How-to: Write and Run Apache Giraph Jobs on Apache Hadoop

by Mirko Kämpf February 05, 2014 **16 comments**

Create a test environment for writing and testing Giraph jobs, or just for playing around with Giraph and small sample datasets.

Apache Giraph is a scalable, fault-tolerant implementation of graph-processing algorithms in Apache Hadoop clusters of up to thousands of computing nodes. Giraph is in use at companies like Facebook and PayPal, for example, to help represent and analyze the billions (or even trillions) of connections across massive datasets. Giraph was inspired by **Google's Pregel** framework and integrates well with Apache Accumulo, Apache HBase, Apache Hive, and Cloudera Impala.

Currently, the upstream **"quick start" document** explains how to deploy Giraph on a Hadoop cluster with two nodes running Ubuntu Linux. Although this setup is appropriate for lightweight development and testing, using Giraph with an enterprise-grade **CDH-based cluster** requires a slightly more robust approach.

In this how-to, you will learn how to use Giraph 1.0.0 on top of CDH 4.x using a simple example dataset, and run example jobs that are already implemented in Giraph. You will also learn how to set up your own Giraph-based development environment. The end result will be a setup (not intended for production) for writing and testing Giraph jobs, or just for playing around with Giraph and small sample datasets. (In future posts, I will explain how to implement your own graph algorithms and graph generators as well as how to export your results to **Gephi**, the "Adobe Photoshop for graphs", through Impala and JDBC for further inspection.)

How Giraph Works

As in the core MapReduce framework, in Giraph, all data and workload distribution related details are hidden behind an easy-to-use API. Currently, Giraph API is used on top of MR1 via a slightly unstable mechanism, although a YARN-based implementation is also possible (also to be discussed in a future post). I recommend, to turn off preemption in your cluster if you plan to run Giraph jobs.

In Giraph, a worker node or a slave node is a host (either a physical server or even a virtualized server) that performs the computation and stores data in HDFS. Such workers load the graph and keep the full graph or just a part of it (in case of distributed graph analysis) in memory. Very large graphs are partitioned and distributed across many worker nodes.

(Note: the term *partition* has a different meaning in Giraph. Here, the partitioning of the graph is not necessarily the result of the application of a graph-partitioning algorithm. Rather, it is more a way to group data based on a vertex hash value and the number of workers. This approach is comparable to the partitioning that is done during the shuffle-and-sort phase in MapReduce.)

A Giraph algorithm is an iterative execution of "super-steps" that consist of a message exchange phase followed by an aggregation and node or edge property update phase. While vertices and edges are held in memory, the nodes exchange messages in parallel. Therefore, all worker nodes communicate and send each other small messages, usually of very low data volume.

After this message-exchange phase, a kind of aggregation is done. This leads to an update of the vertex and/or edge properties and a super-step is finished. Now, the next super-step can be executed and nodes communicate again. This approach is known as **Bulk Synchronous Processing (BSP)**. The BSP model is vertex based and generally works with a configurable graph model ***G***<*I*,*V*,*E*,*M*>, where ***I*** is a vertex ID, ***V*** a vertex value, ***E*** an edge value, and ***M*** a message data type, which all implement the Writable interface (which is well known from the Hadoop API).

However, the iterative character of many graph algorithms is a poor fit for the MapReduce paradigm, especially if the graph is a very large one like the full set of interlinked Wikipedia pages. In MapReduce, a data set is streamed into the mappers and aggregation of intermediate results is done in reducers. One MapReduce job can implement one super-step. In a next super-step, the whole graph structure — together with the stored intermediate state of the previous step — have to be loaded from HDFS and stored at the end again. Between processing steps, the full graph is loaded from HDFS and stored there, which is a really large overhead. And let's not forget that the intermediate data requires local storage on each worker node while it passes the shuffle-sort phase.

For very large graphs, it is inefficient to repeatedly store and load the more or less fixed structural data. In contrast, the BSP approach loads the graph only once, at the outset. The algorithms assume that runtime-only messages are passed between the worker nodes. To support fault-tolerant operation the intermediate state of the graph can be stored from time to time, but usually not after each super-step.

MapReduce-based implementations of graph algorithms are demonstrated in the book ***Data-Intensive Text Processing with MapReduce*** (J. Lin, C. Dyer).

Hands-On

The starting point for the hands-on portion of this how-to is Clouder's QuickStart VM, which contains a virtualized pseudo-distributed Hadoop cluster managed by Cloudera Manager. The VM has many useful modules already pre-installed, including Git client, Maven, and the Eclipse IDE. In this environment, you can deploy and run Giraph benchmarks as functional tests (rather than for performance reasons).

Prepare and Test the Virtual Hadoop cluster

The Cloudera QuickStart VM requires a 64-bit environment and a minimum of 4GB of RAM allocated to the virtual machine.

Our deployment here requires the following software/hardware setup:

- Hardware: (either a real PC or the Cloudera QuickStart VM, download it **here**)
  - Dual-core 2 GHz CPU (64-bit architecture)
  - 4GB RAM (6GB would be much better)
  - 25GB hard drive space
  - 100 Mbps NIC
- CentOS 6.4 (64-bit)
- Admin account (in Quickstart-VM)
  - username: cloudera
  - password: cloudera
- Hostname: localhost
  - IP address: 127.0.0.1
  - Network mask: 255.255.255.0
- Oracle JDK (version 1.6.0_32)
- Apache Hadoop (CDH 4.x)
- Apache Giraph 1.0.0

An accompanying Github project for this tutorial, called **giraphl**, contains related material such as a convenient bootstrap script. This script will save you time as you do not have to type all commands provided in this tutorial. Those who want a more manual experience, however, can follow the steps below.

Test the Hadoop Deployment

First, start the VM and confirm that all required services are up and running. Either you use command `sudo jps`, which should list at least the following services:

- NameNode
- SecondaryNameNode
- DataNode
- JobTracker
- TaskTracker

Or, open Cloudera Manager in a browser at **http://localhost:7180/**.

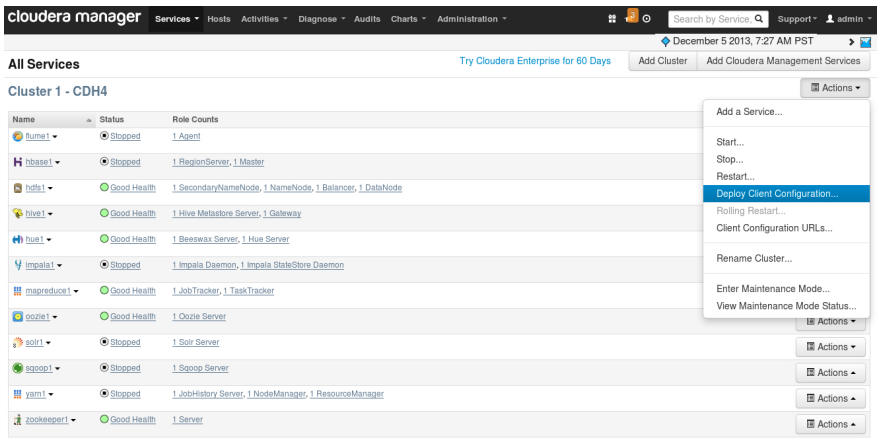

Figure 1: Cloudera Manager shows the service status of a healthy Hadoop cluster in a QuickStart VM. You can deploy the Hadoop client configuration to the local host, which is used during the tutorial.

The HDFS and MapReduce services are required at a minimum for this how-to. If all services are not in a healthy state, start troubleshooting now.

A prerequisite for running MapReduce jobs in the QuickStart VM (because it uses Cloudera Manager) is to deploy the client configuration. If you skip this step, the `hadoop` command will use the local job runner by default. So, you need to do this sequence of steps:

1. Go to Cloudera Manager at **http://localhost:7180/**.
2. Login using “cloudera” for both the username and password.
3. Click the “Actions” menu directly below “Add Cloudera Management Services” and select “Deploy Client Configuration”. (Note: There are several similar dropdowns, so refer to the screenshot in Figure 1 for the correct one.)
4. Click the blue “Deploy Client Configuration” button to confirm your selection. In less than a minute, you’ll see confirmation that the configuration set up in Cloudera Manager is available to the clients of the different services (such as MapReduce) on the cluster.

Running a MapReduce Job

Some MapReduce jobs can now be executed. For a first test, use the TeraGen and TeraSort jobs to create some random test data and to sort the data set via MapReduce.

As user "cloudera", call the following commands:

Run TeraGen:

```
$ hadoop jar /usr/lib/hadoop-0.20-mapreduce/hadoop-examples.jar \
  teragen 50000 TESTFILE
```

Run TeraSort:

```
$ hadoop jar /usr/lib/hadoop-0.20-mapreduce/hadoop-examples.jar \
  terasort TESTFILE TESTFILE.sorted
```

While the jobs are running you can monitor them via the Web user interface:

- NameNode daemon: http://127.0.0.1:50070
- JobTracker daemon: http://127.0.0.1:50030

Your Hadoop cluster is now ready for action. But before writing a Giraph job, let's review some Giraph concepts.

Anatomy of a Giraph Program

A Giraph program consists of a master, many workers, and an Apache **ZooKeeper** ensemble that maintains the global state of the application or the Giraph job. The active master works as an application coordinator — it synchronizes the super-steps. If the master fails, another master from the ZooKeeper master queue can be activated. (Thus, fault tolerance is built-in.) Workers get a partition of the graph assigned from the master before the super-step starts. Each worker handles the I/O operations, the computational part of the algorithm, and the messaging between the nodes.

Figure 2 illustrates the three phases of the data flow in a Giraph application. It starts with loading the graph (Phase 1): The master assigns the InputSplits to workers, which are responsible for loading the data. This process is similar to classical MapReduce processing, where each mapper task is responsible for processing one InputSplit. Phase 2 is the iterative part of the algorithm, which consists of concatenated super-steps. Finally, each worker contributes the data from all partitions for which it was responsible to the OutputFormat to write results to HDFS, HBase, or even Accumulo.

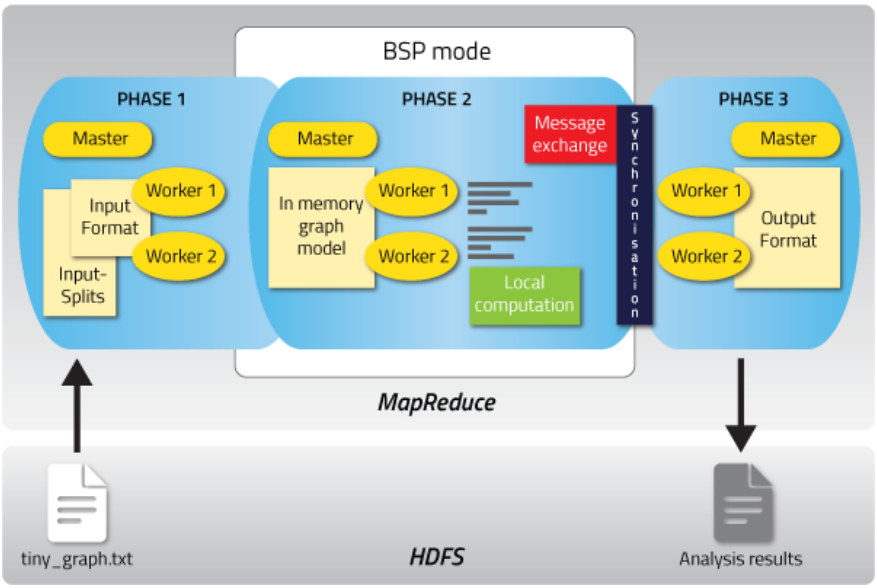

Figure 2: The Giraph data flow in a classical MapReduce implementation

In classical Hadoop 1.x, a map-only job initializes the BSP based implementation of large-scale graph algorithms. After workers load the initial graph data (Phase 1 in Figure 2), the super-steps are executed (Phase 2 in Figure 2). Within each super-step, the method compute(…) of all given vertexes is called. A vertex has the data, which describes the state of the node, and its compute method is part of the implementation of a single algorithm. So you need special vertex implementations for different algorithms.

Some algorithms require an iterative update of a certain node property or vertex value (for example, the page rank of a node). Therefore, you have to store such data outside the compute method as a vertex attribute. All attributes within a vertex are visible within this vertex only and should be in the private part of the vertex class. Don't forget to initialize the variable before the program starts!

One simple example is the single-source shortest path (SSSP) algorithm. Here, you store the current lowest distance to a selected source vertex as a node property in each vertex. After a new message is received, the values can be updated with the new lowest value at the end of each iteration.

Sometimes you need access to node properties from other nodes. To make such values global accessible, use a persistent aggregator. The aggregator has to be initialized before the program starts, and in each super-step, every vertex can send its current value there. Such an aggregator uses a hashmap to store a value for each vertex id. In the case of a persistent aggregator, the data is persistent across all super-steps and the data will be accessible by all vertexes. Developers should be careful to not collect too many large data objects in an aggregator.

Deploying Giraph

In order to download Giraph from a repository and **build** it, you need Git and Maven 3 installed. Both packages are preinstalled and configured in QuickStart VM. Giraph can support multiple versions of Hadoop, and here you'll use the hadoop-2.0.0 profile.

First, choose a directory for Giraph. (Most people use /usr/local/giraph, but this is not a strong requirement.) You should define this location as a system variable called GIRAPH_HOME and export it via the .bashrc file in the home directory of the user cloudera. Therefore, you have to edit /home/cloudera/.bashrc. Add the following line:

```
export GIRAPH_HOME=/usr/local/giraph
```

(Note: At the time of this writing, it was not possible to compile giraph-1.1.0. Therefore, the latest stable release, version 1.0.0, is used here.)

Let's define a version and a profile identifier variable by adding:

```
export GV=1.0.0
export PRO=hadoop_2.0.0
```

to the file /home/cloudera/.bashrc and refresh the environment variables with:

```
$ source ~/.bashrc
```

Clone the Giraph project from a Github mirror:

```
$ sudo mkdir /usr/local/giraph
$ cd $GIRAPH_HOME
$ cd ..
$ sudo git clone https://github.com/apache/giraph.git
$ sudo chown -R cloudera:hadoop giraph
```

Check out the stable branch, and build Giraph by running the following commands. (Note: there is no space character between -D and skipTests):

```
$ cd $GIRAPH_HOME
$ git checkout release-$GV
$ mvn package -DskipTests -Dhadoop=non_secure -P $PRO
```

The argument `-DskipTests` will skip the testing phase to save some time. As you are not working with a secure cluster setup, you have to select the non-secure mode by using the option `-Dhadoop=non_secure`.

The build procedure may take a while. You may have to execute the build command multiple times. But finally you should see output similar to the following:

```
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] Apache Giraph Parent .....................      SUCCESS [0.720s]
[INFO] Apache Giraph Core .......................      SUCCESS [21.551s]
[INFO] Apache Giraph Hive I/O ..................     SUCCESS [10.896s]
[INFO] Apache Giraph Examples ..............     SUCCESS [7.088s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 40.526s
[INFO] Finished at: Thu Dec 05 08:36:52 PST 2013
[INFO] Final Memory: 43M/569M
[INFO] ------------------------------------------------------------------------
```

At the end, you should have the distributable Giraph JAR file:

```
giraph-core/target/giraph-$GV-for-$PRO-alpha-jar-with-dependencies.jar
```

and the Giraph examples JAR file:

```
giraph-examples/target/giraph-examples-$GV-for-$PRO-alpha-jar-with-dependencies.jar
```

For convenience, I prefer to create symbolic links to the jar files. Usually such links are managed by the Linux `alternatives` command.

```
$ cd $GIRAPH_HOME
$ sudo ln -s \
giraph-core/target/giraph-$GV-for-$PRO-alpha-jar-with-dependencies.jar \
giraph-core.jar
$ sudo ln -s \
giraph-examples/target/giraph-examples-$GV-for-$PRO-alpha-jar-with-dependencies.jar \
giraph-ex.jar
```

Preparing Sample Data

The next step is to prepare some example data sets in HDFS. The official Giraph quick-start tutorial uses the `tiny_graph.txt` file, which contains a small test network. (Later, this network can be loaded via `JsonLongDoubleFloatDoubleVertexInputFormat` input format.)

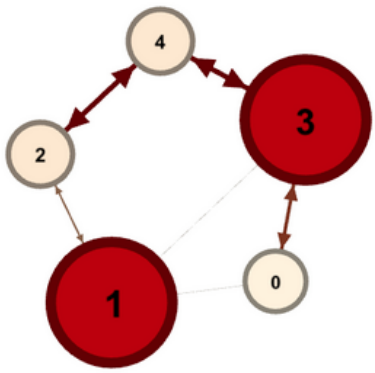

Figure 3: The tiny sample graph plotted in Gephi.

Copy the text below:

[0,0,[[1,1],[3,3]]]
[1,0,[[0,1],[2,2],[3,1]]]
[2,0,[[1,2],[4,4]]]
[3,0,[[0,3],[1,1],[4,4]]]
[4,0,[[3,4],[2,4]]]

into a new empty file named `tiny_graph.txt` in your QuickStart VM and then copy this file to HDFS into a directory named ginput, which is in your HDFS home directory.

Next, create some working directories in HDFS where you will store the input data sets and results:

```
$ hadoop fs -mkdir ginput
$ hadoop fs -mkdir goutput
```

To copy the manually created input file to HDFS, you should use the Hadoop filesystem shell. Please make the current directory in your shell the one in which you stored the `tiny_graph.txt` file and then type:

```
$ hadoop fs -put tiny_graph.txt ginput
```

The meaning of the data in each line is:

```
[source_id,source_value,[[dest_id, edge_value],...]]
```

The `source_value` property describes the node and the `edge_value` is an edge property (such as the correlation link strength). There are five nodes and 12 directed edges in this graph.

Another example data set is available **here**. Download and decompress the file, then upload it to HDFS. But first you have to install wget via yum package manager.

```
$ sudo yum install wget
$ wget http://ece.northwestern.edu/~aching/shortestPathsInputGraph.tar.gz
$ tar zxvf shortestPathsInputGraph.tar.gz
```

```
$ hadoop fs -put shortestPathsInputGraph ginput
```

With those files in place, you can now run some Giraph jobs.

Running Giraph Jobs

The Giraph library offers MapReduce programs, called GiraphJobs, but no additional services for the Hadoop cluster. Therefore, you do not have to deploy any configuration or services on your running Hadoop cluster; rather, just submit the existing map-only jobs. As soon as you do that via the hadoop command-line tool, you have to specify all required libraries. In this case, that will be the giraph-core.jar and the giraph-ex.jar files. The first one contains all Giraph-related code and has to be deployed via the `-libjars` option. This means that the jar files specified here are added to the distributed cache. During runtime, those libraries are available in the Java classpath on all cluster nodes running a task, which belongs to the submitted job.

Giraph jobs are MapReduce jobs. Although mappers usually do not communicate with other mappers, Giraph uses MapReduce only during the initialization phase and mapper-to-mapper communication is actually required.

A ZooKeeper quorum provides a global shared memory. You tell Giraph about the configuration of the ZooKeeper servers with the following property:

```
-Dgiraph.zkList=<zknode1>:<zkport1>,<zknode2>:<zkport2> ...
```

The Cloudera QuickStart VM has a running HBase cluster with ZooKeeper running on port 2181. So, use that instead of Giraph's default ZooKeeper port, which is assumed to be port 22181.

The rest of this section will show you how to run Giraph jobs and what parameters are required. More details about using other existing algorithms and designing your own implementations will be explained in a subsequent post.

PageRankBenchmark

There are some benchmarks implemented in the package org.apache.giraph.benchmark. Run the PageRankBenchmark just to verify the setup and the build.

```
$ hadoop jar giraph-ex.jar org.apache.giraph.benchmark.PageRankBenchmark
-Dgiraph.zkList=127.0.0.1:2181 -libjars giraph-core.jar \
-e 1 -s 3 -v -V 50 -w 1
```

Parameter Name	Description
`-e` `--edgesPerVertex`	Number of edges per vertex that are generated
`-s`	Number of super-steps to execute before finishing the job
`-v`	Run the job in verbose mode.
`-V` `--aggregateVertices`	Aggregate the vertices.
`-w` `--workers`	Number of workers (the number of mappers that have to be started in parallel)

SimpleShortestPathsVertex

The SimpleShortestPathsVertex job will be started to explain the specific command-line properties, which are slightly different from a classical Hadoop job.

It reads a graph from an input file stored in HDFS to compute the length of the shortest paths from one chosen source node to all other nodes. The current implementation of the algorithm will only process the first vertex in the input file. If you need the shortest path for all available vertexes, multiple jobs would have to be started. In each run you would specify the vertex id with a custom argument:

```
-ca SimpleShortestPathsVertex.source=2
```

Here we will use the JsonLongDoubleFloatDoubleVertexInputFormat and IdWithValue- TextOutputFormat output file formats. The result file is a simple text file where each line consists of **target_id length** for each vertex in the graph. Length is the shortest path to a target node, starting from the single source node, defined by the custom property.

```
$ hadoop jar giraph-ex.jar org.apache.giraph.GiraphRunner \
-Dgiraph.zkList=127.0.0.1:2181 -libjars giraph-core.jar \
org.apache.giraph.examples.SimpleShortestPathsVertex \
-vif org.apache.giraph.io.formats.JsonLongDoubleFloatDoubleVertexInputFormat \
```

```
-vip /user/training/ginput/tiny_graph.txt \
-of org.apache.giraph.io.formats.IdWithValueTextOutputFormat \
-op /user/training/goutput/shortestpathsC2 \-ca SimpleShortestPathsVertex.source=2 \
```

Parameter Name	Description
`-vif`	The VertexInputFormat for the job
`-vip`	The path in HDFS from which the VertexInputFormat loads the data
`-of`	The OutputFormat for the job
`-op`	The path in HDFS to which the OutputFormat writes the data
`-ca`	A custom argument is defined as a key value pair

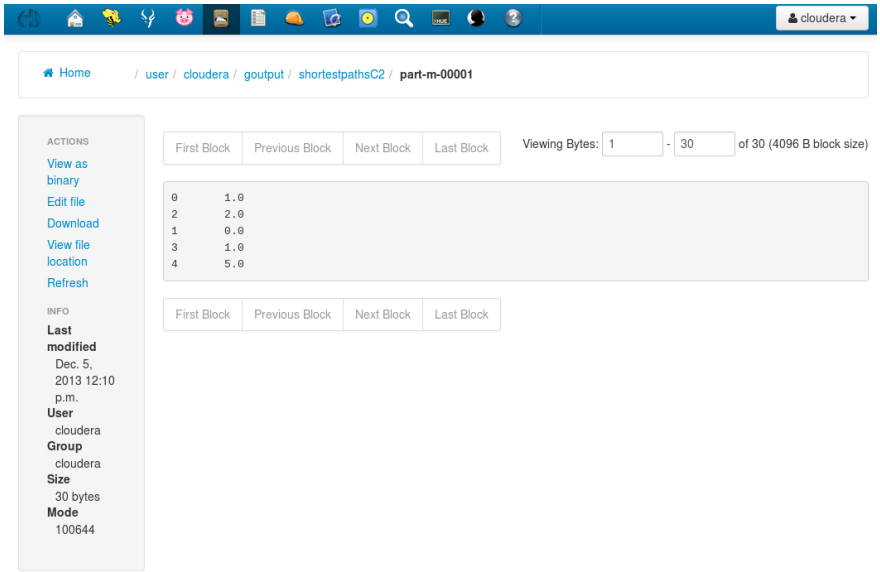

Figure 4: The file browser app in Hue shows the content of the result file, which was generated by our first Giraph job.

Conclusion

Congratulations, you now have a running Hadoop cluster on your desktop and a fresh installation of the latest stable Giraph release, which is version 1.0.0. You should also now know how a Giraph job works and how to run existing algorithms.

In future posts, you will get a deep dive into Giraph development and also discover how to export data to Gephi via Hive, Impala (via JDBC) to visualize large graphs.

Any feedback is welcome in comments!

Mirko Kämpf is the lead instructor for the Cloudera Administrator Training for Apache Hadoop for Cloudera University.

Filed under:

General

Graph Processing

Hadoop

16 Responses

PRAVEEN SRIPATI / FEBRUARY 05, 2014 / 2:00 PM

Lately Cloudera has been pushing Spark, which has Graphx. When to use Giraph and when Graphx?

Praveen

JUSTIN KESTELYN (@KESTELYN) / FEBRUARY 06, 2014 / 2:45 PM

Praveen,

Cloudera is agnostic on this issue; neither Graphx nor Giraph are in CDH today. You should probably ask contributors to those projects directly.

DANNY / FEBRUARY 14, 2014 / 6:15 AM

It seems that Giraph 1.1.0 can only be compiled with Java 7:

https://github.com/apache/giraph/commit/ac93c3b6c5bce5f22b293b29df91663ca7d7ce63

Is there a possibility to install the CDH4 cluster through the CDM with Java7?

GHUFRAN / FEBRUARY 19, 2014 / 12:07 PM

Hello,

Thank you for this fantastic article. I have set-up Giraph via your quick start vm, but have had trouble running the SimpleShortestPathsVertex example. I received warnings saying output format is not recognised then the giraph job runs and fails producing no output in the hadoop fs. I get the same error in my own personal set-up of Giraph as well and was wondering if I was missing something or if this was a current problem with the Giraph itself?

below is the output recieved from the quick start vm terminal:

14/02/18 09:44:25 INFO utils.ConfigurationUtils: No edge input format specified. Ensure your InputFormat does not require one.
14/02/18 09:44:25 INFO utils.ConfigurationUtils: Setting custom argument [SimpleShortestPathsVertex.source] to [2] in GiraphConfiguration
14/02/18 09:44:25 WARN job.GiraphConfigurationValidator: Output format vertex index type is not known
14/02/18 09:44:25 WARN job.GiraphConfigurationValidator: Output format vertex value type is not known
14/02/18 09:44:25 WARN job.GiraphConfigurationValidator: Output format edge value type is not known
14/02/18 09:44:25 INFO job.GiraphJob: run: Since checkpointing is disabled (default), do not allow any task retries (setting mapred.map.max.attempts = 0, old value = 4)
14/02/18 09:44:26 WARN mapred.JobClient: Use GenericOptionsParser for parsing the arguments. Applications should implement Tool for the same.
14/02/18 09:44:28 INFO mapred.JobClient: Running job: job_201402180900_0002
14/02/18 09:44:29 INFO mapred.JobClient: map 0% reduce 0%
14/02/18 09:44:47 INFO mapred.JobClient: Job complete: job_201402180900_0002
14/02/18 09:44:47 INFO mapred.JobClient: Counters: 4
14/02/18 09:44:47 INFO mapred.JobClient: Job Counters
14/02/18 09:44:47 INFO mapred.JobClient: Total time spent by all maps in occupied slots (ms)=16399
14/02/18 09:44:47 INFO mapred.JobClient: Total time spent by all reduces in occupied slots (ms)=0
14/02/18 09:44:47 INFO mapred.JobClient: Total time spent by all maps waiting after reserving slots (ms)=0
14/02/18 09:44:47 INFO mapred.JobClient: Total time spent by all reduces waiting after reserving slots (ms)=0

Thanks,

Ghufran

GHUFRAN / FEBRUARY 19, 2014 / 4:13 PM

I seemed to of solved my problem, by using the following class as the -of:

IdWithValueTextOutputFormat$IdWithValueVertexWriter

Ghufran

STEFAN BESKOW / FEBRUARY 24, 2014 / 8:56 AM

Hi.

Thanks for the great article. It was very helpful to get Giraph up and running on a Cloudera VM. Now I'm also trying to get Giraph up and running on Hadoop 2.0.0-cdh4.2.0 using a cluster with 60 nodes. When I run the sample application org.apache.giraph.examples.SimpleShortestPathsVertex with just 1 worker it works fine, but when I specify more than 1 worker it throws exception java.lang.IllegalArgumentException: checkLocalJobRunnerConfiguration as shown below. Is there a way to pass a command line parameter to Giraph so that it doesn't use the local job runner or do I need to update any of the Hadoop configuration files for this to work?

Here is the command I use to run sample application with 2 workers:
hadoop jar giraph-examples-1.0.0-for-hadoop-2.0.0-cdh4.2.0-jar-with-dependencies.jar org.apache.giraph.GiraphRunner -Dgiraph.zkList=rdcgrd001.unx.sas.com:2181 -libjars giraph-examples-1.0.0-for-hadoop-2.0.0-cdh4.2.0-jar-with-dependencies.jar org.apache.giraph.examples.SimpleShortestPathsVertex -vif org.apache.giraph.io.formats.JsonLongDoubleFloatDoubleVertexInputFormat -vip /user/stbesk/input/tiny_graph.txt -of org.apache.giraph.io.formats.IdWithValueTextOutputFormat -op /user/stbesk/output/shortestpathsC2 -ca SimpleShortestPathsVertex.source=2 -w 2 -ca giraph.SplitMasterWorker=true

Here is the exception:
14/02/24 00:20:23 INFO utils.ConfigurationUtils: No edge input format specified. Ensure your InputFormat does not require one.

14/02/24 00:20:23 INFO utils.ConfigurationUtils: Setting custom argument [SimpleShortestPathsVertex.source] to [2] in GiraphConfiguration
14/02/24 00:20:23 INFO utils.ConfigurationUtils: Setting custom argument [giraph.SplitMasterWorker] to [true] in GiraphConfiguration
14/02/24 00:20:23 WARN job.GiraphConfigurationValidator: Output format vertex index type is not known
14/02/24 00:20:23 WARN job.GiraphConfigurationValidator: Output format vertex value type is not known
14/02/24 00:20:23 WARN job.GiraphConfigurationValidator: Output format edge value type is not known
14/02/24 00:20:23 INFO job.GiraphJob: run: Since checkpointing is disabled (default), do not allow any task retries (setting mapred.map.max.attempts = 0, old value = 4)
Exception in thread “main” java.lang.IllegalArgumentException: checkLocalJobRunnerConfiguration: When using LocalJobRunner, must have only one worker since only 1 task at a time!
at org.apache.giraph.job.GiraphJob.checkLocalJobRunnerConfiguration(GiraphJob.java:151)
at org.apache.giraph.job.GiraphJob.run(GiraphJob.java:225)
at org.apache.giraph.GiraphRunner.run(GiraphRunner.java:94)
at org.apache.hadoop.util.ToolRunner.run(ToolRunner.java:70)
at org.apache.hadoop.util.ToolRunner.run(ToolRunner.java:84)
at org.apache.giraph.GiraphRunner.main(GiraphRunner.java:124)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)
at java.lang.reflect.Method.invoke(Method.java:597)
at org.apache.hadoop.util.RunJar.main(RunJar.java:208)

Appreciate any help.

Thanks.
Stefan

MIRKO / MARCH 04, 2014 / 12:16 PM

Did you deploy the client configuration the right way? If a client has not the right cluster config it might use the local mode, so please check this setup.

ALEXANDER RIGGERS / APRIL 09, 2014 / 6:23 AM

Hello Mirko

I was wondering if you know where I can find an overview of the supported algorithms for Apache Giraph? The API documentation is really messy and I can not figure out which algorithms they support.

Kind regards,
Alex

GLENN STRYCKER / APRIL 22, 2014 / 1:00 PM

How was the Gephi plot for tiny_graph.txt generated? That is, Gephi does not recognize the format when I load that file, so I’m curious if I should use a different extension. Which graph file format uses [source_id,source_value, [[dest_id,edge_value],…]] ??

Is there a quick and handy way to convert this to a .dot, .glm, .gexf, or .gephi file?

MIRKO / MAY 10, 2014 / 12:35 AM

Glenn, the graph was plotted with Gephi and manually translated file of the format:

source target weight
0,1,1
0,3,3
1,0,1
1,2,2
1,3,1
2,1,2
2,4,4
3,0,3
3,1,1
3,4,4
4,3,4
4,2,4

How to translate the format or how to request a large graph to show this in Gephi, you will see in the part, which is already on its way. We use the Hadoop-Gephi-Connector to request large filtered edge lists from Impala and Hive.

MIRKO / MAY 10, 2014 / 12:09 PM

Alexander,
I think you should look here: **http://grafos.ml/index.html#Okapi**
This project uses Giraph but wraps more docu around it.
Another alternative would be to use GraphLab, which works also on HDFS data but does not use MapReduce.

POOJA / JUNE 13, 2014 / 9:04 AM

Really nice blog. I was able to install Giraph without facing any issue. But getting the below error when trying to run

SimpleShortestPathVertex example.

Exception in thread “main” java.io.IOException: Error opening job jar: giraph-ex.jar
at org.apache.hadoop.util.RunJar.main(RunJar.java:135)
Caused by: java.util.zip.ZipException: error in opening zip file
at java.util.zip.ZipFile.open(Native Method)
at java.util.zip.ZipFile.(ZipFile.java:127)
at java.util.jar.JarFile.(JarFile.java:135)
at java.util.jar.JarFile.(JarFile.java:72)
at org.apache.hadoop.util.RunJar.main(RunJar.java:133)

MIRKO / JUNE 24, 2014 / 9:17 AM

POOJA, I think you missed this step:

$ cd $GIRAPH_HOME
$ sudo ln -s
giraph-core/target/giraph-$GV-for-$PRO-alpha-jar-with-dependencies.jar
giraph-core.jar
$ sudo ln -s
giraph-examples/target/giraph-examples-$GV-for-$PRO-alpha-jar-with-dependencies.jar
giraph-ex.jar

… after the maven build command. Please check if the symbolic link “giraph-ex.jar” points to the right jar and if you have the permissions.

MATTHEW CORNELL / AUGUST 05, 2014 / 12:15 PM

The SimpleShortestPathsVertex output is for source = 1, not 2.

ESSIE / DECEMBER 01, 2014 / 8:05 PM

Hi, am new to hadoop. when I run
‘git checkout release-$GV’, i get the following error

fatal: Not a git repository (or any of the parent directories): .git

CRISTIAN VIDAL SILVA / MARCH 31, 2015 / 4:39 AM

Dear.

I successfully installed Giraph on Cloudera… Right now I want to test my own code, or compile and test Giraph examples… How can do it?

I hope you can give me tips ASAP.

Greetings from Chile.

Leave a comment

Name REQUIRED

Email REQUIRED
(WILL NOT BE PUBLISHED)

Website

Comment

Leave Comment

Prove you're human! *

5 – five =