EEL 6761: Cloud Computing Spring 2022 Instructor: J. Fortes
Project Assigned: 4/6/2022 Due: 4/20/2022 @ 5 Pm (use UF E-learning - Canvas)

Deploy an API service for a distributed data processing pipeline on CloudLab

Preamble:

In this project, you will deploy a distributed data processing pipeline that analyzes data streamed from three biodiversity data aggregators: iDigBio (https://idigbio.org/), GBIF (https://gbif.org/), and OBIS (https://obis.org/). You will integrate a messaging queue into the pipeline to collect, buffer, and dispense the streamed data to a data processing engine. For this project you will use Apache Spark as the data processing engine and Apache Kafka as the messaging queue. The integration of Apache Spark with Apache Kafka can be found extensively in the data analysis domain (few examples are provided in Section 9.1 of Cloud Computing for Science and Engineering by Ian Foster and Dennis B. Gannon). Figure 1 shows the overview of the data processing pipeline that you will be implementing. You will deploy it entirely on CloudLab and present your work in class as part of the submission. A one lecture hour will be given to provide hands-on training on how to use CloudLab.

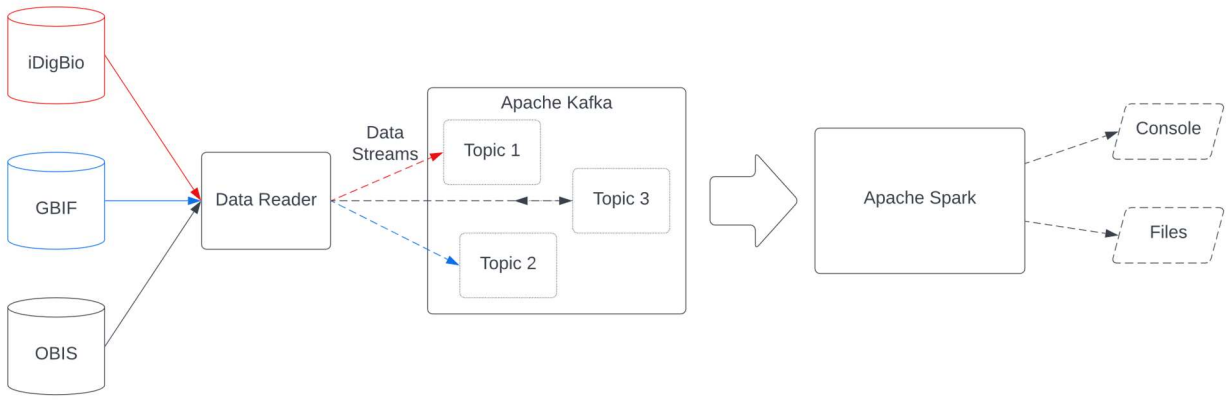

Figure 1: Overview of the data processing pipeline. The colored lines indicate that the data streams are written to specific topics. In this case, all iDigBio streams are written to the topic "Topic 1", GBIF streams are written to the topic "Topic 2", and OBIS streams are written to the topic "Topic 3".

Apache Spark:

Apache Spark (or Spark) is a data processing engine that uses dataflow graphs to process data. Traditionally, large data processing was done in batches by partitioning data into subsets which are then and distributed across multiple compute nodes to be processed in parallel. MapReduce is a popular framework that facilitates this batch processing workflow. In MapReduce, the data is distributed over a distributed file system and the function (task) is applied over each datum generating a <key, value> pair – map. The output of the map process (all the <key, value> pairs) is then consumed the reduce process to generate the desired output. For instance, in the classic word-counting example, the map function processes each word in the input and creates a <word, count> pair. In this case, the count is always 1. The shuffle process orders the <word, count> pairs and the reducer aggregates the total count of each word and emits <word, total count> pairs. Apache Hadoop, precursor to Spark, only allowed batch processing. Spark was developed by representing the MapReduce model as a dataflow graph thereby allowing Spark to process streams (Spark streaming). More information on MapReduce can be found in Section 7.4 and

Section 7.5 of Cloud Computing for Science and Engineering by Ian Foster and Dennis B. Gannon, and at https://www.talend.com/resources/what-is-mapreduce/. More information about Spark can be found in Section 8.1 and Section 8.2 of Cloud Computing for Science and Engineering by Ian Foster and Dennis B. Gannon.

Spark can be deployed in a single node or over multiple nodes (cluster). In the single node setup, a single instance of Spark is deployed to process the data while in the cluster setup, multiple instances of Spark are deployed on potentially different physical machines. In the cluster setup, one of the Spark instances is chosen as the master while the other Spark instances act as workers. The master node is responsible for distributing jobs amongst its workers and for monitoring the overall progress of the jobs. The master node can also process the job in addition to managing the worker nodes.

Apache Kafka:

Apache Kafka (or Kafka) is a distributed event store and stream-processing platform. Kafka can also be used as a distributed message queue (publish-subscribe) to improve the reliability of the system. In a high throughput streaming environment, data could be lost if the rate of the incoming stream is higher than the throughput of the data processing engine. For instance, data from sensors that are monitoring a wide array of equipment in an assembly plant are critical and the data processing pipeline cannot afford to lose any data. Section 9.6 of Cloud Computing for Science and Engineering by Ian Foster and Dennis B. Gannon discusses briefly about Kafka.

In your setup, you will use Kafka as a distributed message queue to store the incoming data streams temporarily until Spark has the resources to read from the message queue. Kafka groups similar streams of data into topics, as shown in Figure 1, where data streams from a single source are grouped into a single topic. For example, data from iDigBio are grouped into topic 1. Spark will consume the data streams from each topic, i.e., you will write your Spark application in such a way that the queries are run on data from individual data sources. Topics can be further divided into partitions to facilitate high-read throughput. Detailed information on topics and partitions can be found in https://www.instaclustr.com/blog/apache-kafka-architecture/. Kafka, similar to Spark, can be run either in a single node or in a multi-node cluster. The nodes in which Kafka is installed are called brokers and each broker has a unique ID. Each broker can contain one or more topics.

In this project, you will instantiate 3 nodes (physical machines) in CloudLab. On these 3 nodes you will deploy both Kafka and Spark. Figure 2 shows the setup of Kafka and Spark on 3 nodes in CloudLab, as well as the various software components used in cluster mode.

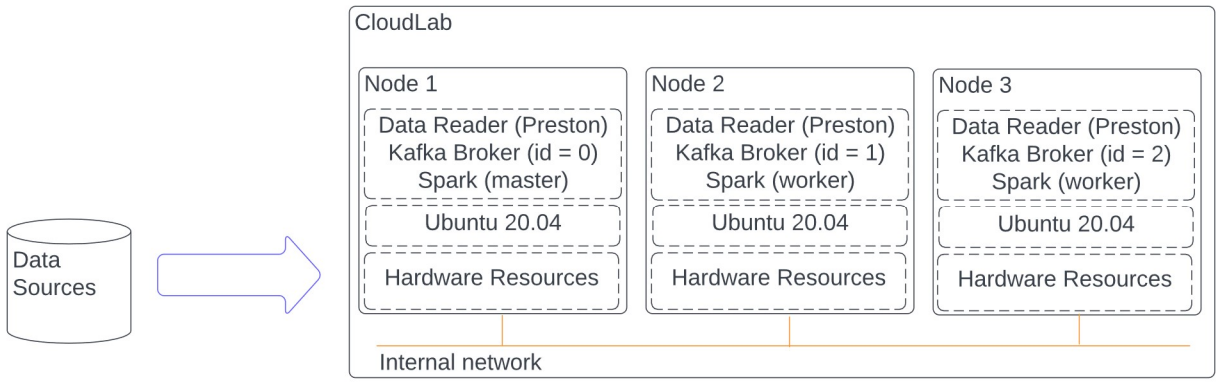

Figure 2: In this example, node 1 runs Spark master. In your implementation, Spark master can run on any node. For brevity, only one data source is shown, however, in your deployment, you will have 3 data sources that are available over the internet. Data readers running on each node will connect with exactly one data source.

Important notes (Read carefully before you start the project):

- Deploy the entire setup locally before you move it to CloudLab. You can use VirtualBox or VMWare Workstation Player to create VMs locally and simulate a 3-node setup in your local machine. Once you deploy the Kafka cluster and the Spark cluster locally, run example code that is shipped with Kafka and Spark (more details are provided in the next section in task 1). This will ensure that your setup works fine and there are no hidden surprises. This is important since, when you port the setup to CloudLab you will likely face other issues with network configuration, etc. that may make it very difficult to diagnose issues with your Spark and Kafka setup.
- CloudLab is a shared environment and researchers usually reserve resources in advance. Do not wait until the last minute to start on the project since there's a high likelihood of the resources being unavailable due to prior reservations.
- Treat the CloudLab machines as your personal computer and set strong passwords where needed. Do not share your private ssh key with anyone. CloudLab admins constantly monitor the network traffic, and you risk losing access to CloudLab resources for the rest of the semester if the CloudLab admins detect any anomalous traffic from your nodes.
- CloudLab has a user forum for discussions and questions. Treat the forum like Stack Overflow. Search for answers before posting questions and follow the community guidelines. CloudLab admins are very helpful and prompt in their response provided you follow the guidelines while posting questions.

Tasks:

There are two tasks in this project, described below:

Task 1: Deploy the distributed data processing pipeline on CloudLab:

a. Make sure you have Java installed in your node before you proceed. Download and install Spark (https://spark.apache.org/downloads.html) on the 3 nodes. Configure Spark to run as a standalone multi-node cluster (refer to https://spark.apache.org/docs/latest/spark-standalone.html). It will be better if you configure Spark to run with a single call to the shell script (start-all.sh). This way

you can create a systemd service for Spark that will run Spark as a service (refer to homework 2 for steps). You can also manually start your Spark cluster whenever you instantiate a new machine in CloudLab.
Hint: When you start Spark using start-all.sh, the Spark process runs until you call stop-all.sh.

b. Download and install Kafka (https://kafka.apache.org/downloads) on the 3 nodes. Configure Kafka to run as a multi-node cluster (to set up a single node, see https://kafka.apache.org/quickstart). To understand how Kafka works refer to the detailed Kafka documentation at https://kafka.apache.org/documentation.html.

c. For the data generator, you will be using a tool called Preston. Preston parses the data from the desired data source, converts the data into JSON format and streams the output. Use the following command (**be sure to remove line breaks**) to install Preston in your node:

```
sudo sh -c 'curl -L https://github.com/bio-
guoda/preston/releases/download/0.3.5/preston.jar > /usr/local/bin/preston && chmod +x
/usr/local/bin/preston'
```

To learn more about Preston refer to the readme at https://github.com/bio-guoda/preston.

d. You will create the following shell script in the 3 nodes. This script clears cached files created by Preston while generating the streams for the project. This ensures that you do not run out of local disk space in your nodes. Call this file *clean-cache*. Do not forget to make the shell script an executable (*chmod +x*).

```
#!/bin/sh
mkdir -p data
cd data

while IFS='$\n' read -r line; do
    # Ignore unnecessary information
    if [ $(echo $line | cut -d " " -f2) = "<http://purl.org/pav/hasVersion>" ]
    then
        # Keep the newest 5 files, delete everything else
        rm -rf $(ls -1t */*/* | tail -n +11)

        # Echo stdin to stdout
        printf "%s\n" "$line"
    fi
done
```

e. You will implement a Kafka producer that will receive the data streams from Preston and store it in the appropriate topic based on the source (refer Figure 1). Preston can be started using the following command:

```
preston track --seed https://idigbio.org | ./clean-cache | preston json-stream 2>
/dev/null
```

Note: You are invoking the "clean-cache" shell script that you created in the previous step in this command. Call the appropriate script if you named the file differently.

The above command will output a continuous stream of JSON data from the iDigBio source, where each line is a "specimen record" in JSON format, e.g.,

```
{ "http://rs.tdwg.org/dwc/terms/kingdom": "Animalia", "http://rs.tdwg.org/dwc/ter ... }
{ "http://rs.tdwg.org/dwc/terms/kingdom": "Plantae", "http://rs.tdwg.org/dwc/ter ... }
{ "http://rs.tdwg.org/dwc/terms/kingdom": "Fungi", "http://rs.tdwg.org/dwc/ter ... }
...
```

You must stream the data into your Kafka producer for further processing. To stream data from the other two sources, replace "`--seed https://idigbio.org`" with "`--seed https://gbif.org`" for GBIF and "`--seed https://obis.org`" for OBIS. This will be used in the task 2 of this project.

f. You will implement a Spark map-reduce application that will read stream data from Kafka and calculate the following over 2-minute windows for each data source:
   1) Count the number of plant, animal, and fungi records
   2) Count the number of records processed from each data source URL
   3) Count the total number of unique species across all records

   Refer to https://dwc.tdwg.org/list/ for full list of labels and the example values that you can expect to see in the data. To identify if a record (or data) is a plant, or animal, or fungi, check its field named "http://rs.tdwg.org/dwc/terms/kingdom" for the values "Plantae", "Animalia", and "Fungi". The field "http://rs.tdwg.org/dwc/terms/scientificName" holds the species name of each record.

   Note: You will need to save the results somewhere that all of your nodes can access. This may be HDFS, a Kafka topic, or a shared file system. In task 2, below, you will implement an API call to retrieve the results.

g. To test your Kafka + Spark pipeline, run 3 instances of Preston, one for each data source. For each instance of Preston, stream the results to a topic corresponding to the URL of the data source. For example, you should have one topic named "https://idigbio.org", one named "https://gbif.org", and another named "https://obis.org". You will trigger all 3 instances at the same time. Your Spark application must process at least 10 windows of data, i.e., at least 20 minutes of streamed data. Your output should be a table for each data source in the following format:

Window #	Number of plant records	Number of animal records	Number of fungi records	Total number of unique species
1	A	B	C	D

   Note 1: Your Spark application need not output the table. You can post-process the output to derive the table.
   Note 2: Your Spark application and Kafka producer must be running and waiting to ingest data before you start Preston.
   Note 3: You must configure Kafka to discard messages after a short period of time, e.g., 2 minutes. This can be done by setting a retention policy in Kafka's config/server.properties file or as an option in your command used to create each Kafka topic.

Task 2: Deploy an API:

You will implement a web API that allows users to control, monitor, and read results from the distributed system you deployed in task 1. The API will expose at least three commands: two to manage data streaming into Kafka, and one *parameterized* command to query for results from Spark.

a. To begin implementing the web API, you must create a web server. With Python, a simple web server can be created using the web.py package (installed using `pip install web.py`). See https://webpy.org/ for quick-start information and https://webpy.org/docs/0.3/tutorial for a more detailed introduction to programming a server to parse request URLs, routing the requests to handlers, sending responses, and starting up the server to listen for incoming requests on a specified port. You may choose any port between 10000 and 65535. Do not use ports that are generally reserved for "well-known" purposes such as HTTP (80) and HTTPS (443) and are therefore common targets of malicious attacks.

b. Implement an API call to begin streaming data from a URL. This will involve starting a Preston process to download records from the URL (as described earlier) and streaming the output into a Kafka producer. The producer should select a Kafka broker (i.e., stream data to the IP of the node that runs the broker) using a round-robin rule: the first added source should stream data to the Kafka broker on "node 0", the second should stream data to the broker on "node 1", and so on. For this project, the list of sources is restricted to three aforementioned data sources (iDigBio, GBIF, and OBIS).

Request	`GET /addSource?url=[URL]`

Response Code	If the value of the "url" parameter is in the list of sources, return
	`200 OK`
	Otherwise,
	`400 Bad Request`

Response Body	Empty

c. Implement an API call to list the set of data source URLs that are currently being used to stream data into Kafka

Request	`GET /listSources`
Response Code	`200 OK`

Response Body	A list of active data source URLs, e.g.
	`https://idigbio.org`
	`https://obis.org`

d. Implement an API call to retrieve results that have been calculated by Spark.

Request	`GET /count?by=[BY]`
	
	Valid parameter values:
	`by=source`
	`by=kingdom`
	`by=totalSpecies`

Response Code	If the value of the "by" parameter is a valid value, return `200 OK` Otherwise,

	`400 Bad Request`

Response Body	For `by=source`, include a list of active data source URLs and the number of records streamed from each, e.g.,
	`https://idigbio.org 98`
	`https://gbif.org 124`
	`https://obis.org 22`
	
	For `by=kingdom`, include a list of the unique "kingdom" names that have been processed and the number of records of each kingdom, e.g.,
	`Animalia 51`
	`Plantae 94`
	
	For `by=totalSpecies`, print "Species" and the number of unique species that have been observed, e.g.,
	`Species 48`

e. If request is made with an unrecognized URL path (ignoring parameters, i.e., everything including and after a "?"), respond with a "`404 Not Found`" response code.

Note: When your web server provides a `400 Bad Request` response code, you may choose to include an explanation in the body to help with debugging. For example, if an invalid parameter is supplied, it helps to explain this in the body, e.g., respond to "`/count?kee=kingdom`" with "Unrecognized parameter 'kee'".

f. **(EXTRA CREDIT)** Implement an API call to stop streaming data from a URL. This will require you to keep track of which URLs are being used as data sources and to be able to kill the corresponding data streaming process.

Request	GET /removeSource?url=[URL]

Response Code	If the value of the "url" parameter is in the list of sources, return
	`200 OK`
	Otherwise,
	`400 Bad Request`

Response Body	Empty

g. **(EXTRA CREDIT)** Enhance the "count" function with a "windows" parameter to allow users to retrieve multiple windows of data. The parameter will specify exactly how many windows of data should be returned. If "windows=all" is provided, return all saved window data.

Request	GET /count?by=[BY]&windows=[#WINDOWS] Default parameter values: windows=1

Response Code	If the value of the "by" parameter is valid and the value of "windows" is either equal to "all" or greater than 0, return
	`200 OK`
	Otherwise,
	`400 Bad Request`

Response Body	The same as before, except include the data from multiple windows. Additionally, prefix each line with the timestamp of the window that provided the data, e.g.,
	`1 Animalia 51`
	`1 Plantae 94`
	`2 Animalia 65`
	`2 Plantae 83`
	`3 Fungi 11`

h. **(EXTRA CREDIT)** Implement token-based authentication for all API calls by requiring a "token" parameter to be specified. Only process requests that provide a valid token, i.e., a string value that is known by both the user and the web server. If a valid token is not provided, all calls should respond with `401 Unauthorized`. You may add an unauthenticated API call (e.g., "getToken") to allow users to request a token, or you may simply use a fixed string value. Additional security can be added by assigning an expiration date to each token, or by only accepting tokens provided from the same IP address that requested the token, or by any other means.

i. **(EXTRA CREDIT)** Use a signed SSL certificate to enable secure communication using HTTPS.

j. **(EXTRA CREDIT)** If your API web server communicates using HTTPS and uses token authentication, you may *forward* traffic on port 443 (HTTPS) to your API web server (which should still be listening on port 10000 or above). This way, users will not be required to specify the port that your server listens on.

Submission guidelines:

1. You will be uploading a PDF (<your_last_name>_project.pdf) as part of the project submission. The PDF must contain:
    a. The 10 windows of Spark outputs as described in part g of task 1.
    b. Brief description of your setup i.e. any scripts that you implemented along with its purpose.
    c. A copy of the source code for Spark and any scripts that you implemented to complete the project pasted towards the end of the PDF.

2. Your API will be tested using cURL, e.g.,

```
- curl https://[your-server]:[port]/addSource?url=https://idigbio.org
- curl https://[your-server]:[port]/listSources
- curl https://[your-server]:[port]/count?by=kingdom
```

   Malformed requests (e.g., undefined commands, invalid parameter values, etc.) will also be tested.

3. You will meet with the TAs for a live demo, where you will be asked to execute several API calls. You will also be expected to explain how Kafka, Spark, Preston, and the API web server work together to implement a web API.

Submission policy:

- All submissions are expected by the deadline specified in the homework assignment. Grade is automatically reduced by **25% for every late day**.