Data Processing
WWW and search

- Internet introduced a new challenge in the form of a web search engine
  - Web crawler data at a "peta scale"
  - Requirement for efficient indexing to enable fast search (on a continuous basis)
- Addressed via..
  - Google file system (GFS)
    - Large number of replicas distributed widely for fault-tolerance and performance
  - MapReduce
    - Efficient, data parallel computation
MapReduce

- Programming model for processing large data sets with a parallel, distributed algorithm on a cluster
  - Developed to process Google's ~ 20000 petabytes per day problem
  - Supports batch data processing to implement Google search index generation
- Users specify the computation in two steps
  - Recall CS 320 functional programming paradigm
  - **Map**: apply a function across collections of data to compute some information
  - **Reduce**: aggregate information from map using another function (e.g. fold, filter)
  - Sometimes **Shuffle** thrown in between (for maps implementing multiple functions)
MapReduce run-time system

- Automatically parallelizes distribution of data and computation across clusters of machines
- Handles machine failures, communications, and performance issues.
- Initial system described in...
- Re-implemented and open-sourced by Yahoo! as Hadoop
Application examples

- Google
  - Word count
  - Grep
  - Text-indexing and reverse indexing
  - Adwords
  - Pagerank
- Bayesian classification: data mining
- Site demographics
- Financial analytics
- Data parallel computation for scientific applications
  - Gaussian analysis for locating extra-terrestrial objects in astronomy
  - Fluid flow analysis of the Columbia River
Algorithm

- Map: replicate/partition input and schedule execution across multiple machines
- Shuffle: Group by key, sort
- Reduce: Aggregate, summarize, filter or transform
- Output the result
MapReduce example

- Simple word count on a large, replicated corpus of books

How many times does the word “vampire” appear in all “books” in The Twilight Saga?

I heard 6 “Vampire”s
MapReduce

- What about Werewolf and Human?
- Use a map that does multiple counts followed by a shuffle to send to multiple reduce functions
Issue: Single processing model

- Maps with varying execution times cause imbalances
  - Difficult to reallocate load at run-time automatically
- Map computations all done first
  - Reducer blocked until data from map fully delivered
  - Want to stream data from map to reduce
- Batch processing model
  - Bounded, persistent input data in storage
  - Input mapped out, reduced, then stored back again
    - Might want intermediate results in memory for further processing or to send to other processing steps
  - No support for processing and querying indefinite, structured, typed data streams
    - Stock market data, IoT sensor data, gaming statistics
    - Want to support multiple, composable computations organized in a pipeline or DAG
Stream processing systems

- Handle indefinite streams of structured/typed data through pipelines of functions to produce results
  - Programming done via graph construction
  - Graph specify computations and intermediate results
  - Software equivalent to PSU Async
- Several different approaches
  - Stream-only (Apache Storm/Samza)
  - Hybrid batch/stream (Apache Spark/Flink/Beam)
Cloud Dataproc & Dataflow
Google Cloud Dataproc

- Managed Hadoop, Spark, Pig, Hive service
  - Parallel processing of mostly batch workloads including MapReduce
- Hosted in the cloud (since data is typically there)
- Clusters created on-demand within 90 seconds
- Can use pre-emptible VMs (70% cheaper) with a 24-hour lifetime
Google Cloud Dataflow

- Managed stream and batch data processing service
  - Open-sourced into Apache Beam
  - Supports stream processing needed by many real-time applications
  - Supports batch processing via data pipelines from file storage
  - Data brought in from Cloud Storage, Pub/Sub, BigQuery, BigTable
- Transform-based programming model
  - Cluster for implementing pipeline automatically allocated and sized underneath via Compute Engine
  - Work divided automatically across nodes and periodically rebalanced if nodes fall behind
  - Transforms in Java and Python currently
Components

- Graph-based programming model
- Runner
Graph-based programming model

- Programming done at a higher abstraction level
  - Specify a directed acyclic graph using operations (in code, in JSON, or in a GUI)
  - Underlying system pieces together code
- Originally developed in Google Dataflow
  - Spun out to form the basis of Apache Beam to make language independent of vendor
- [https://beam.apache.org/documentation/programming-guide/](https://beam.apache.org/documentation/programming-guide/)
• Example
  • Linear pipeline of transforms that take in and produce data in collections
• More complex pipeline
- Familiar core transform operations
  - ParDo (similar to map)
  - GroupByKey (similar to shuffle)
  - Combine (similar to various fold operations)
  - Flatten/Partition (split up or merge together collections of the same type to support DAG)
Runner

- Run-time system that takes graph and runs job
  - Apache Spark or Apache Flink for local operation
  - Cloud Dataflow for sources on GCP
- Runner decides resource allocation based on graph representation of computation
  - Graph mapped to ComputeEngine VMs automatically in Cloud Dataflow
Example
Labs
Cloud Dataproc Lab #1

- Calculate $\pi$ via massively parallel dart throwing
- Two ways (27 min)
  - Command-line interface
  - Web UI
Computation for calculating $\pi$

- Square with sides of length 1 (Area = 1)
- Circle within has diameter 1 (radius = $\frac{1}{2}$)
  - Area is ?

- Randomly throw darts into square
  - What does the ratio of darts in the circle to the total darts correspond to?
  - What expression as a function of darts approximates $\pi$?
• Algorithm
  • Spawn 1000 dart-throwers (map)
  • Collect counts (reduce)
• Modified computation on quadrant
  • Randomly pick x and y uniformly between 0,1 and calculate "inside" to get ratio
  • Dart is inside orange when $x^2 + y^2 < 1$

```python
def inside(p):
    x, y = random.random(), random.random()
    return x*x + y*y < 1

count = sc.parallelize(xrange(0, NUM_SAMPLES)).filter(inside).count()
print "Pi is roughly %f" % (4.0 * count / NUM_SAMPLES)
```
Provisioning and Using a Managed Hadoop/Spark Cluster with Cloud Dataproc (Command Line) (20 min)

- Enable API

```bash
gcloud services enable dataproc.googleapis.com
```

- Skip to end of Step 4
  - Set zone to `us-west1-b` (substitute zone for rest of lab)

```bash
gcloud config set compute/zone us-west1-b
```

- Set name of cluster in `CLUSTERNAME` environment variable to `<username>-dplab`

```bash
CLUSTERNAME=${USER}-dplab
```
• Create a cluster with tag "codelab" in us-west1-b

```
gcloud dataproc clusters create ${CLUSTERNAME} \\
    --scopes=cloud-platform \\
    --tags codelab \\
    --zone=us-west1-b
```

• Go to Compute Engine to see the nodes created

<table>
<thead>
<tr>
<th>Node</th>
<th>Zone</th>
</tr>
</thead>
<tbody>
<tr>
<td>wuchang-dplab-m</td>
<td>us-west1-b</td>
</tr>
<tr>
<td>wuchang-dplab-w-0</td>
<td>us-west1-b</td>
</tr>
<tr>
<td>wuchang-dplab-w-1</td>
<td>us-west1-b</td>
</tr>
</tbody>
</table>
Note the current time, then submit job specifying

- 1000 workers
- stdout and stderr sent to output.txt via >&
- Command placed in the background via ending &

```bash
gcloud dataproc jobs submit spark --cluster ${CLUSTERNAME} \
--class org.apache.spark.examples.SparkPi \
--jars file:///usr/lib/spark/examples/jars/spark-examples.jar -- 1000 \
> output.txt &
```

List the jobs periodically via

```bash
gcloud dataproc jobs list --cluster ${CLUSTERNAME}
```

When done, note the time. How long did it take?

Examine `output.txt` via `less` to find the string "Pi is"

What is the estimate for $\pi$?
• Show the cluster to find the `numInstances` used for the master and the workers (save to a file if necessary)

```bash
gcloud dataproc clusters describe ${CLUSTERNAME}
```

• Allocate two pre-emptible machines to the cluster

```bash
gcloud dataproc clusters update ${CLUSTERNAME} --num-preemptible-workers=2
```

• Repeat listing to see Config section they show up in

```bash
gcloud dataproc clusters describe ${CLUSTERNAME}
```

• Show them in Compute Engine
Repeat with new setup

- Note the current time, then submit job again, saving result to a different file

```
gcloud dataproc jobs submit spark --cluster ${CLUSTERNAME} \  
   --class org.apache.spark.examples.SparkPi \  
   --jars file:///usr/lib/spark/examples/jars/spark-examples.jar -- 1000 \  
   >& output2.txt &
```

- List the jobs periodically via

```
gcloud dataproc jobs list --cluster ${CLUSTERNAME}
```

- When done, note the time. How long did it take?
  - What is the estimate for π?
• ssh into the master node
  `gcloud compute ssh ${CLUSTERNAME}-m --zone=us-west1-b`

• Once logged in, get the hostname
  `hostname`

• List the cluster to show all VMs
  `gcloud dataproc clusters list`

• Then logout
• Skip Step 10
• Delete cluster
  `gcloud dataproc clusters delete ${CLUSTERNAME}`

• Ensure no instances from cluster are running on Compute Engine before continuing to Step 12
Version #1: Command-line interface

- **Step 12**
  - **Learn More**
    - Dataproc Documentation: [https://cloud.google.com/dataproc/overview](https://cloud.google.com/dataproc/overview)
    - [Getting Started with Dataproc using the Console](https://codelabs.developers.google.com/codelabs/cloud-dataproc-gcloud)
  - Click on "Getting Started…" link to do the lab via the console

- Version #1: Provisioning and Using a Managed Hadoop/Spark Cluster with Cloud Dataproc (Command Line) (20 min)
  - [https://codelabs.developers.google.com/codelabs/cloud-dataproc-gcloud](https://codelabs.developers.google.com/codelabs/cloud-dataproc-gcloud)
Skip steps 1, 2, 3
Step 4
- Goto Cloud Dataproc
- Create a cluster in `us-west1-b` with master and worker nodes set to `n1-standard-2` VMs
- Click "Submit a Job", choose region and cluster just created
- Set job type to Spark
- Set name of main jar
  - Java version
- Set args to 1000
  - # of tasks
- Set location of jar

<table>
<thead>
<tr>
<th>Job ID</th>
<th>job-9541a6f4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Region</td>
<td>us-west1</td>
</tr>
<tr>
<td>Cluster</td>
<td>cluster-60b8</td>
</tr>
<tr>
<td>Job type</td>
<td>Spark</td>
</tr>
<tr>
<td>Main class or jar</td>
<td>org.apache.spark.examples.SparkPi</td>
</tr>
<tr>
<td>Arguments</td>
<td>1000</td>
</tr>
<tr>
<td>Jar files</td>
<td>file:///usr/lib/spark/examples/jars/spark-examples.jar</td>
</tr>
</tbody>
</table>
- Start job and wait a minute for completion
- Upon completion, click on job, then click on Line wrapping to see output
Version #2: Web UI

- Delete cluster
Cloud Dataproc Lab #1

• Version #2: Introduction to Cloud Dataproc: Hadoop and Spark on Google Cloud Platform (7 min)
  • https://codelabs.developers.google.com/codelabs/cloud-dataproc-starter
Cloud Dataflow Lab #1

- Run a Big Data Text Processing Pipeline in Cloud Dataflow (21 min)
  - Generates a histogram of words in a file
  - Default input file
    - `gs://dataflow-samples/shakespeare/kinglear.txt`
  - Default output files at...
    - `gs://${YOUR_OUTPUT_PREFIX}/`
  - Done in Java, but equivalents in Python, Go (typical languages used for data processing)
**Dataflow**
- Split file into collection of lines
- Split lines into collection words
- Count words to generate collection of key-value pairs
- Extract word counts to generate collection of formatted output strings
- Output results to files
- Instantiate options from command-line arguments
- Create a Dataflow pipeline with the options
- Setup pipeline with transforms
  - Call "ReadLines" on input file specified in options
  - Call CountWords() transform
  - Take histogram and generate formatted output (i.e. Strings
  - Call "WriteCounts" on output file specified in options to write formatted output
- Note: Every transform supports an "apply" method
  - Chained as in functional programming
Main transform `CountWords()` done in two steps
- ParDo `ExtractWordsFn()` that takes collections of lines and generates collections of words via call to `split`
- Built-in transform `Count()` that takes collections of words and generates collections of word-counts (key-value pairs)
Create output string from key-value collection that contains histogram information
• List the APIs to see the range of services available
  `gcloud services list --available`

• To enable a service like the Cloud Datastore API, the command would be
  `gcloud services enable datastore.googleapis.com`

• From the list, enable the following services
  • Google Cloud Datastore API
  • Google Dataflow API
  • Stackdriver Logging API
  • Google Cloud Storage
  • Google Cloud Storage JSON API
  • BigQuery API
  • Google Cloud Pub/Sub API
• Create a multi-regional bucket in Cloud Storage
• Launch Cloud Shell (note that the steps for setting the project are not necessary)
• Create an Apache Maven project
  • Maven is a build automation tool used to construct application (compiles code, run tests, creates JAR file)

```
mvn archetype:generate \  -DarchetypeArtifactId=google-cloud-dataflow-java-archetypes-examples \  -DarchetypeGroupId=com.google.cloud.dataflow \  -DarchetypeVersion=1.9.0 \  -DgroupId=com.example \  -DartifactId=first-dataflow \  -Dversion="0.1" \  -DinteractiveMode=false \  -Dpackage=com.example
```

• `first-dataflow` directory created
• Cloud Dataflow SDK for Java installed along with example pipelines
• Set environment variables for your `PROJECT_ID` and storage bucket
  
  ```shell
  export PROJECT_ID=$DEVSHELL_PROJECT_ID
  export BUCKET_NAME=<your_bucket_name>
  ```

• Ensure `GOOGLE_APPLICATION_CREDENTIALS` environment variable is set and pointing to a valid JSON containing service account information
  
  • [https://console.cloud.google.com/apis/credentials/service_accountkey?_ga=2.254517962.33815533.1524620040](https://console.cloud.google.com/apis/credentials/service_accountkey?_ga=2.254517962.33815533.1524620040)

• Change into project directory
  
  ```shell
  cd first-dataflow
  ```
• Use Apache Maven to build word-counting application
  • Compile code, run tests, package results into JAR files

```
mvn compile exec:java \
   -Dexec.mainClass=com.example.WordCount \
   -Dexec.args="--project=${PROJECT_ID} \
   --stagingLocation=gs://${BUCKET_NAME}/staging/ \
   --output=gs://${BUCKET_NAME}/output \
   --runner=BlockingDataflowPipelineRunner"
```
Examine timing information and worker execution history by clicking on job.

**wordcount-wuchangfeng-0606030124**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>End time</th>
<th>Elapsed time</th>
<th>Start time</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>wordcount-wuchangfeng-0606030124</td>
<td>Batch</td>
<td>Jun 5, 2018, 8:05:03 PM</td>
<td>2 min 54 sec</td>
<td>Jun 5, 2018, 8:02:10 PM</td>
<td>Succeeded</td>
</tr>
</tbody>
</table>

Worker history

- **ReadLines**: Succeeded 0 sec
- **WordCount.CountWords**: Succeeded 3 sec
- **ParDo(FormatAsText)**: Succeeded 0 sec
- **WriteCounts**: Succeeded 4 sec

<table>
<thead>
<tr>
<th>Target workers</th>
<th>Timestamp</th>
<th>Rationale</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Jun 5, 2018, 8:03:39 PM</td>
<td>Stopping worker pool.</td>
</tr>
<tr>
<td>1</td>
<td>Jun 5, 2018, 8:02:19 PM</td>
<td>Starting a pool of 1 workers.</td>
</tr>
</tbody>
</table>
• View output in bucket

<table>
<thead>
<tr>
<th>Name</th>
<th>Size</th>
<th>Type</th>
<th>Storage class</th>
</tr>
</thead>
<tbody>
<tr>
<td>output-00000-of-00003</td>
<td>13.94 KB</td>
<td>text/plain</td>
<td>Multi-Regional</td>
</tr>
<tr>
<td>output-00001-of-00003</td>
<td>16.72 KB</td>
<td>text/plain</td>
<td>Multi-Regional</td>
</tr>
<tr>
<td>output-00002-of-00003</td>
<td>17.14 KB</td>
<td>text/plain</td>
<td>Multi-Regional</td>
</tr>
<tr>
<td>staging/</td>
<td></td>
<td>Folder</td>
<td></td>
</tr>
</tbody>
</table>

clean-up (bucket)
Cloud Dataflow Lab #1

- Run a Big Data Text Processing Pipeline in Cloud Dataflow (21 min)
  - [https://codelabs.developers.google.com/codelabs/cloud-dataflow-starter](https://codelabs.developers.google.com/codelabs/cloud-dataflow-starter)
Data Science Lab #1

- Do one of the labs in GCP’s Cloud Quest (except the first one on Earthquake data)
  - [https://codelabs.developers.google.com/cloud-quest-scientific-data](https://codelabs.developers.google.com/cloud-quest-scientific-data)
  - Descriptions here:
    - [https://cloud.google.com/blog/big-data/2017/07/new-hands-on-labs-for-scientific-data-processing-on-google-cloud-platform](https://cloud.google.com/blog/big-data/2017/07/new-hands-on-labs-for-scientific-data-processing-on-google-cloud-platform)

- End of labs for notebook
Extra
MapReduce Engine

- MapReduce requires a distributed file system and an engine that can distribute, coordinate, monitor and gather the results.
- Hadoop provides that engine through (the file system we discussed earlier) and the JobTracker + TaskTracker system.
- JobTracker is simply a scheduler.
- TaskTracker is assigned a Map or Reduce (or other operations); Map or Reduce run on node and so is the TaskTracker; each task is run on its own JVM on a node.
Example

- Baseball analytics for 2016 MLB World Series
  - BigQuery (data storage)
  - Cloud Dataflow (data processing)
  - Cloud Datalab (data visualization)
Large scale data splits

Map <key, 1> <key, value> pair

Reducers (say, Count)

- Count
- Count
- Count

P-0000 (count1)
P-0001 (count2)
P-0002 (count3)