CS 455: Introduction To Distributed Systems [Hadoop]

What's this hullabaloo about an elephant?

No, not the one named Horton
Who has fun in the Jungle of Nool

This one's named Hadoop, and is just as cool
Crunching through data and having fun

Shrideep Pallickara
Computer Science
Colorado State University

1

Topics covered in this lecture

- Hadoop
  - Application development
  - API

2

HADOOP

3

Hadoop

- Java-based open-source implementation of MapReduce
- Created by Doug Cutting
- Origins of the name Hadoop
  - Stuffed yellow elephant
- Includes HDFS [Hadoop Distributed File System]

4

Hadoop timelines

- Feb 2006
  - Apache Hadoop project officially started
  - Adoption of Hadoop by Yahoo! Grid team
- Feb 2008
  - Yahoo! Announced its search index was generated by a 10,000-core Hadoop cluster
- May 2009
  - 17 clusters with 24,000 nodes

Hadoop Releases

- There are four active releases at the moment
  - 2.7.x
  - 2.8.x
  - 3.1.x
  - 3.2.x
- Last release from the 2.7.x branch (v2.7.7) was on May 31, 2018
  - 2.7.x branch is in maintenance mode
- All 3.x.x branches had releases in 2019

Hadoop Evolution

- 0.20.x series became 1.x series
- 0.23.x was forked from 0.20.x to include some major features
- 0.23 series later became 2.x series
- 2.8.0 is branched off from 2.7.3
- 2.9.0 is branched off from 2.8.2
- 3.0.0 series is branched off from 2.7.0
- 3.1.0 series is branched off from 3.0.0
- 3.2.0 is branched off from 3.1.0

0.23 included several major features

- New MapReduce runtime, called MapReduce 2, implemented on a new system called YARN
  - YARN: Yet Another Resource Negotiator
  - Replaces the "classic" runtime in previous releases
- HDFS federation
  - HDFS namespace can be dispersed across multiple name nodes
- HDFS high-availability
  - Removes name node as a single point of failure; supports standby nodes for failover

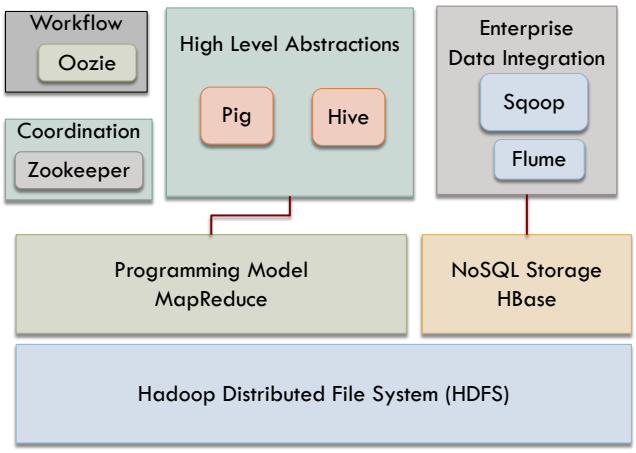
3.2.0 includes major features

- Hadoop Submarine support
  - Hadoop Submarine is a new project that orchestrates Tensorflow programs without modifications on Yarn and provide access to data stored on HDFS
  - Support for GPUs and Docker images
- New/Improved storage connectors
  - ADLS (Azure Datalake Generation 2), Amazon S3, and Amazon DynamoDB
- HDFS storage policies
  - Hierarchical storage – Archival, Disk (default), SSD, and RamDisk
  - Users can define the type of storage when storing data
  - Blocks can be moved between different storage types

Latest Release

- February 6, 2019
  - v3.1.2 released [We will use this for HW3]
- September 22, 2019
  - v3.2.1 released
  - This version is considered stable, but production use not widespread yet.
- v2.9.2 ,v2.8.5, and v2.7.7 were released in 2018.

The Hadoop Ecosystem



MapReduce Jobs

- A MapReduce **Job** is a <u>unit of work</u>
- Consists of:
  - Input Data
  - MapReduce program
  - Configuration information
- Hadoop runs the jobs by dividing it into **tasks**
  - Map tasks
  - Reduce tasks

Types of nodes that control the job execution process [Older Versions]

- **Job tracker**
  - *Coordinates all jobs* by scheduling tasks to run on task trackers
  - Records overall progress of each job
    - If task fails, reschedule on a different task tracker
- **Task tracker**
  - Run tasks and reports progress to job tracker

Types of nodes that control the job execution process [Newer Versions]

- Resource Manager
- Application Manager
- Node manager

Processing a weather dataset

- The dataset is from NOAA
- Stored using a line-oriented format
  - Each line is a record
- Lots of elements being recorded
- We focus on temperature
  - Always present with a fixed width

Format of a record in the dataset

```
0057
332130          # USAF weather station identifier
99999           # WBAN weather station identifier
19500101        # Observation date
300             # Observation time
4
+51317          # latitude (degrees x 1000)
+028783         # longitude (degrees x 1000)
FM-12
+0171           # elevation (meters)
99999
V020
320             # wind direction (degrees)
1               # quality code
…
-0128           # air temperature (degrees Celsius x 10)
1               # quality code
-0139           # dew point temperature (degree Celsius x 10)
```

Analyzing the dataset

- What's the highest recorded temperature for each year in the dataset?
- See how programs are written
  - Using Unix tools
  - Using MapReduce

Using awk
Tool for processing line-oriented data

```
#! /usr/bin/env bash
for year in all/*
do
   echo –ne ‘basename $year .gz’ ”\t”
   gunzip –c $year | \
       awk ‘{ temp=substr($0, 88, 5) + 0;
                    q=substr($0, 93, 1);
                    if (temp !=9999 && q ~ /[01459]/ &&
                        temp > max) max = temp }
     END {print max}’
done
```

Sample output that is produced

```
% ./max_temperature.sh

1901        317
1902        244
1903        289
1904        256
1905        283
 …
```


To speed things up, we need to be able to do this processing on multiple machines

- STEP 1: **Divide** the work and execute concurrently on multiple machines
- STEP 2: **Combine** results from independent processes
- STEP 3: **Deal with failures** that might take place in the system

The Hollywood principle

Don't call us, we'll call you.

- Useful software development technique
- Object's (or component's) initial condition and **ongoing life cycle** is handled by its *environment*, rather than by the object itself
- Typically used for implementing a class/component that must fit into the constraints of an existing framework

Doing the analysis with Hadoop

- Break the processing into two phases
  - Map and Reduce
  - Each phase has <*key, value*> pairs as input and output
- Specify two functions
  - Map
  - Reduce

The map phase

- Choose a `Text` input format
  - Each line in the dataset is given as a text *value*
  - *key* is the offset of the beginning of the line from the beginning of the file
- Our map function
  - Pulls out year and the air temperature
  - Think of this as a data preparation phase
    - Reducer will work on data generated by the maps

How the data is represented in the actual file

```
0067011990999991950051507004...9999999N9+00001+99999999999...
0043011990999991950051512004...9999999N9+00221+99999999999...
0043011990999991950051518004...9999999N9-00111+99999999999...
0043012650999991949032412004...0500001N9+01111+99999999999...
0043012650999991949032418004...0500001N9+00781+99999999999...
```

How the lines in the file are presented to the map function by the framework

keys: Line offsets within the file

```
(0,   0067011990999991950051507004...9999999N9+00001+99999999999...)
(106, 0043011990999991950051512004...9999999N9+00221+99999999999...)
(212, 0043011990999991950051518004...9999999N9-00111+99999999999...)
(318, 0043012650999991949032412004...0500001N9+01111+99999999999...)
(424, 0043012650999991949032418004...0500001N9+00781+99999999999...)
```

The lines are presented to the map function as key-value pairs

Map function

- Extract year and temperature from each record and emit output

```
(1950, 0)
(1950, 22)
(1950, -11)
(1949, 111)
(1949, 78)
```

The output from the map function

- Processed by the MapReduce framework *before* being sent to the reduce function
  - **Sort** and **group** <*key, value*> pairs by key
- In our example, each year appears with a list of all its temperature readings

```
(1949, [111, 78])
(1950, [0, 22, -11])
...
```


What about the reduce function?

- All it has to do now is iterate through the list supplied by the maps and pick the max reading
- Example output at the reducer?

```
(1949, 111)
(1950,  22)
...
```

What does the actual code to do all of this look like?

1. Map functionality
2. Reduce functionality
3. Code to run the job

The map function is represented by an abstract `Mapper` class

- Declares an abstract `map()` method
- `Mapper` class is a generic type
  - 4 formal type parameters
  - Specifies input key, input value, output key, and output value

The `Mapper` for our example

```
public class MaxTemperatureMapper extends
     Mapper <LongWritable, Text, Text, IntWritable> {

   private final int MISSING = 9999;

   public void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {
       String line = value.toString();
       String year = line.substring(15, 19);
       int airTemperature;
       if (line.charAt(87) ==  `+` ) {
          airTemperature = Integer.parseInt(line.substring(88, 92));
       } else {
         airTemperature = Integer.parseInt(line.substring(87, 92));
       }
       String quality = line.substring(92, 93);
       if (airTemperature != MISSING && quality.matches("[01459]") {
        context.write(new Text(year), new IntWritable(airTemperature));
      }
   }
}
```


Rather than use built-in Java types, Hadoop uses its own set of basic types

- Optimized for **network serialization**
- These are in the `org.apache.hadoop.io` package
  - `LongWritable` corresponds to Java `Long`
  - `Text` corresponds to Java `String`
  - `IntWritable` corresponds to Java `Integer`

But the `map()` method also had `Context`

- You use this to write the output
- In our example
  - Year was written as a `Text` object
  - Temperature was wrapped as an `IntWritable`

More about Context

- A context object is available at any point of the MapReduce execution
- Provides a convenient mechanism for exchanging required system and job-wide information
- Context coordination happens only when an appropriate phase (driver, map, reduce) of a MapReduce job starts.
  - Values set by one mapper are not available in another mapper but is available in any reducer

The reduce function is represented by an abstract `Reducer` class

- Declares an abstract `reduce()` method
- `Reducer` class is a generic type
  - 4 formal type parameters
  - Used to specify the input and output types of the reduce function
  - The input types should **match** the *output types of the map function*
    - In the example, `Text` and `IntWritable`

The Reducer

```
public class MaxTemperatureReducer extends
     Reducer <Text, IntWritable, Text, IntWritable> {

   public void reduce(Text key, Iterable<IntWritable> values,
                      Context context)
      throws IOException, InterruptedException {

       int maxValue = Integer.MIN_VALUE;

       for (IntWritable value : values) {
          maxValue = Math.max(maxValue, value.get());
       }

       context.write(key, new IntWritable(maxValue));
   }
}
```

The code to run the MapReduce job

```
public class MaxTemperature {
   public static main(String[] args) throws Exception {
      Job job = Job.getInstance();
      job.setJarByClass(MaxTemperature.class);
      job.setJobName("Max temperature");

      FileInputFormat.addInputPath(job, new Path(args[0]));
      FileOutputFormat.setOutputPath(job, new Path(args[1]));

      job.setMapperClass(MaxTemperatureMapper.class);
      job.setReducerClass(MaxTemperatureReducer.class);

      job.setOutputKeyClass(Text.class);
      job.setOutputValueClass(IntWritable.class);

      System.exit(job.waitForCompletion(true) ? 0: 1);
   }
}
```


Details about the Job submission [1/3]

- Code must be packaged in a JAR file for Hadoop to distribute over the cluster
  - `setJarByClass()` causes Hadoop *to locate relevant JAR file* by looking for JAR that contains this class
- Input and output paths must be specified next
  - `addInputPath()` can be *called more than once*
  - `setOutputPath()` specifies the output directory
    - Directory *should not exist* before running the job
    - Precaution to prevent data loss

Details about the Job submission [2/3]

- The methods `setOutputKeyClass()` and `setOutputValueClass()`
  - Control the output types of the map and reduce functions
  - If they are different?
    - Map output types can be set using `setMapOutputKeyClass()` and `setMapOutputValueClass()`

Details about the Job submission. [3/3]

- The `waitforCompletion()` method **submits** the job and **waits** for it to complete
  - The `boolean` argument is a *verbose* flag; if set, progress information is printed on the console
- Return value of `waitforCompletion()` indicates success (`true`) or failure (`false`)
  - In the example this is the program's exit code (0 or 1)

API DIFFERENCES

The old and new MapReduce APIs

- The new API favors abstract classes over interfaces
  - Make things easier to evolve
- New API is in `org.apache.hadoop.mapreduce` package
  - Old API can be found in `org.apache.hadoop.mapred`
- New API makes use of context objects
  - `Context` unifies roles of `JobConf`, `OutputCollector`, and `Reporter` from the old API

The old and new MapReduce APIs

- In the new API, job control is done using the `Job` class rather than using the `JobClient`
- Output files are named slightly differently
  - Old API: Both map and reduce outputs are named `part-nnnn`
  - New API: Map outputs are named `part-m-nnnn` and reduce outputs are named `part-r-nnnn`

The old and new MapReduce APIs

- The new API's `reduce()` method passes values as `Iterable` rather than as `Iterator`
  - Makes it **easier to iterate** over values using the `for-each` loop construct

```
for (VALUEIN value: values) {
    …
}
```

MapReduce tasks & Split strategies

45

Hadoop divides the input to a MapReduce job into fixed-sized pieces

- These are called **input-splits** or just splits
- Creates **one map task per split**
  - Runs *user-defined map function* for each **record** in the split

46

Split strategy: Having many splits

- Time taken to process split is small compared to processing the whole input
- Quality of **load balancing** increases as splits become *fine-grained*
  - Faster machines process proportionally more splits than slower machines
  - Even if machines are identical, this feature is desirable
    - Failed tasks get relaunched, and there are other jobs executing concurrently

Split strategy: If the splits are too small

- **Overheads** for managing splits and map task creation dominates total job execution time
- Good split size tends to be an HDFS **block**
  - This could be changed for a cluster or specified when each file is created

Scheduling map tasks

- Hadoop does its best to run a map task on the *node where input data resides* in HDFS
  - **Data locality**
- What if all three nodes holding the HDFS block replicas are busy?
  - Find free map slot on node in the same rack
  - Only when this is not possible, is an off-rack node utilized
    - Inter-rack network transfer

Why the optimal split size is the same as the block size …

- Largest size of input that can be stored on a single node
- If split size spanned two blocks?
  - Unlikely that any HDFS node has stored *both* blocks
  - Some of the split *will have to be transferred* across the network to node running the map task
    - Less efficient than operating on local data without the network movement

Contents of this slide set are based on the following references

- *Tom White. Hadoop: The Definitive Guide. 3rd Edition. Early Access Release. O'Reilly Press. ISBN: 978-1-449-31152-0. Chapters 1 and 2.*
- Boris Lublinsky, Kevin Smith, and Alexey Yakubovich. Professional Hadoop Solutions. Wiley Press. Chapter 3.