15-319 / 15-619
Cloud Computing

Recitation 13
April 14th 2015
Overview

• Last week’s reflection
  – Project 4.1
• Budget issues
  – Tagging, 15619Project
• This week’s schedule
  – Unit 5 - Modules 18
  – Project 4.2
• Demo
• Twitter Analytics: The 15619Project
Reflections on P4.1

• End-to-End Application using MapReduce, H-Base and web frontend
  • Text Corpus -> NGrams -> Language Model
  • Web app querying HBase
• FAQs
  • Unable to load data into HBase from Reducer
  • Ans: Use the correct jars, build on the instance with the right dependencies, try on small datasets first
• Secret to MapReduce: Start small
Budget Issues

• Tag all resources (Piazza @1656)
  • Amazon EBS General Purpose SSD
  • Amazon EBS Provisioned IOPS
  • Amazon EBS Snapshot

• Untagged resources will be counted towards your weekly project
  • Keep a buffer
Module to Read

• UNIT 5: Distributed Programming and Analytics Engines for the Cloud
  – Module 16: Introduction to Distributed Programming for the Cloud
  – Module 17: Distributed Analytics Engines for the Cloud: MapReduce
  – Module 18: Distributed Analytics Engines for the Cloud: Spark
  – Module 19: Distributed Analytics Engines for the Cloud: GraphLab
Project 4

• Project 4.1
  – MapReduce Programming Using YARN

• Project 4.2
  – Iterative Programming Using Apache Spark

• Project 4.3
  – Graph Programming Using GraphLab
**Typical MapReduce Job**

- Simplistic view of a MapReduce job

![Diagram of MapReduce job]

- You simply write code for the
  - Mapper
  - Reducer

- Inputs are read from disk and outputs are written to disk
  - Intermediate data is spilled to local disk
Iterative MapReduce Jobs

- Some applications require iterative processing
- Eg: Machine Learning, etc.

- MapReduce: Data is always **spilled** to disk
  - Added overhead for each iteration
  - Can we keep data in memory? Across Iterations?
  - How do you manage this?
Resilient Distributed Datasets (RDDS)

• RDDs are
  – in-memory
  – read-only objects
  – partitioned across the cluster
    • partitioned across machines based on a range or the hash of a key in each record
Operations on RDDs

• Loading
  >>>input_RDD = sc.textFile("text.file")

• Transformation
  – Apply an operation and derive a new RDD
    >>>transform_RDD = input_RDD.filter(lambda x: "abcd" in x)

• Action
  – Computations on an RDD that return a single object
    >>>print "Number of “abcd”:" + transform_RDD.count()
RDDs and Fault Tolerance

- Actions create new RDDs
- Instead of replication, recreate RDDs on failure
- Use RDD lineage
  - RDDs store the transformations required to bring them to current state
  - Provides a form of resilience even though they are in-memory
The Spark Framework

RDD Objects

```
val rdd1 = sc.parallelize(Array(1, 2, 3, 4, 5))
val rdd2 = sc.parallelize(Array(6, 7, 8, 9, 10))
val result = rdd1.join(rdd2)
  .groupBy((k, v) => k)
  .mapValues(v => v.map(_ + 10))
  .filter(v => v._2 > 10)
  .collect()
```

Spark Client (Application Master)

- Scheduler and RDD Graph
- Trackers

Task Scheduler

- Cluster Manager

Worker

- Threads
- Block Manager
  - BlockInfo
  - MemoryStore
  - DiskStore
  - ShuffleBlockManager
Spark Ecosystem

- **Spark SQL**
  - Allows running of SQL-like queries against RDDs

- **Spark Streaming**
  - Run spark jobs against streaming data

- **MLlib**
  - Machine learning library

- **GraphX**
  - Graph-parallel framework
Project 4.2

- Build a basic search engine!
  - Index pages
  - Run PageRank
- No need to include an input text predictor
  - Unless you want to and have time and budget to spare
Project 4.2 - Overview

- Use the Wikipedia English dataset
- Build a search engine for Wikipedia documents
- Find and order the pages using
  - Term Frequency - Inverse Document Frequency (TF-IDF)
  - PageRank
Project 4.2 - Two Parts

1. Generate TF-IDF
   – Get weighted measurements of individual words in the document

2. Run PageRank
   – Rank pages according to the incoming and outgoing links
TF-IDF

• Term Frequency (TF)
  – The count of the number of times a word has appeared in a document

• Inverse Document Frequency (IDF)
  – The total number of documents / number of documents in which a particular word has appeared

• For a term $i$ in document $j$

\[ w_{i,j} = tf_{i,j} \times \log \left( \frac{N}{df_i} \right) \]

- $tf_{i,j}$ = number of occurrences of $i$ in $j$
- $df_i$ = number of documents containing $i$
- $N$ = total number of documents
PageRank

• Give pages ranks (scores) based on links to them
• A page that has:
  – Links from many pages ⇒ high rank
  – Link from a high-ranking page ⇒ high rank

PageRank

- For each Page $i$ in dataset, Rank of $i$ can be computed:

$$R[i] = 0.15 + \sum_{j \in \text{Nbr}(i)} w_{ji} \times R[j]$$

- Iterate until $R[i]$ converges
- Formula to be implemented for 4.2 is slightly more complex. Read carefully!!!
PageRank in Spark (Scala)
(Note: This is a simpler version of PageRank, than P4.2)

```scala
val links = spark.textFile(...).map(...).persist()
var ranks = // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS)
{
    // Build an RDD of (targetURL, float) pairs
    // with the contributions sent by each page
    val contribs = links.join(ranks).flatMap
    {
        (url, (links, rank)) =>
        links.map(dest => (dest, rank/links.size))
    }

    // Sum contributions by URL and get new ranks
    ranks = contribs.reduceByKey((x,y) => x+y)
    .mapValues(sum => a/N + (1-a)*sum)
}
```
Launching a Spark Cluster

- Use the Spark-EC2 scripts
- Command line options to specify instance types and spot pricing
- Spark is an in-memory system
  - r3 EC2 instances are a good match as they have a lot of RAM
  - test with a single instance first
- Develop and test your scripts on a portion of the dataset before launching a cluster
Spark Shell

- Like the python shell
- Run commands interactively
- Demo in second half of recitation
- On the master, execute (from /root)
  - ./spark/bin/spark-shell
  - ./spark/bin/pyspark
Grading

- Use submitter file to autograde your answers

- For TF-IDF
  - Run the command `./submitter -t`
    - Uploads code immediately
    - Instant feedback (like Project 4.1)

- For PageRank
  - Run the command `./submitter -p`
    - Uploads code immediately
    - Instant feedback (like Project 4.1)
Upcoming Deadlines

● 15619Project Phase3 Live Test
  ○ Due: Wed, 18:59PM ET April 15th

● 15619Project Final Report
  ○ Due: Sun 11:59PM ET April 19th

● P4.2 Iterative Programming using Spark
  ○ Due: 11:59PM ET April 26th (Sunday)

● Quiz 5, Dist. Prog. & Analytics Engines
  ○ Due: 11:59PM ET April 24th (Friday)
Questions?
TWITTER ANALYTICS:
THE 15619PROJECT
Query 5: Twitter Rankster

- Request: a list of userids and a date range
- You should award points to the users based on these rules:
  - +1 per unique tweet sent by the user in the time interval
  - +3 per friend (based on the maximum value of user.friends_count in the time interval)
  - +5 per follower (based on the maximum value of user.followers_count in the time interval)
Query 5: Twitter Rankster

GET /q5?userlist=12,14,16,18,20&start=2010-01-01&end=2014-12-31

Team,1234-5678-1234,1234-5678-1234,1234-5678-1234
12,173
16,155
14,99
20,99
18,55
Query 6: Hermit Finder

- Request: A range of userids
- You should count the number of users where:
  - userid is between M and N inclusive,
  - has at least one tweet but none of his/her tweets contain location information.

GET /q6?m=0&n=9999999999

Team,1234-5678-1234,1234-5678-1234,1234-5678-1234,1234-5678-1234
55811730
What’s due soon?

- Phase 3 Live Test Deadline
  - Submission of one URL by 18:59 ET (Pittsburgh) Wed 4/15
    - Live Test from 8 PM to midnight ET
  - Choose one database
  - Can only use m1.large or cheaper t1, t2, m1, m3 instances
  - Fix Q1, Q2, Q3, Q4 if your Phase 2 did not go well
  - New queries Q5 and Q6.
  - Phase 3 counts for 60% of 15619Project grade
Phase 3 Report [VERY IMPORTANT]

- Start early
- Document your steps
- Identify and isolate the performance impact of each change you make
- Document your ideas and experiments

MAKE A QUANTITATIVE, DATA-DRIVEN REPORT
Live Test

- 30 minutes warm-up
- 3 hours Q1 - Q6
- 30 minutes mix-Q1Q2Q3Q4Q5Q6
- Preparing for the live test
  - Choose a database based on your observations from previous phases and all six queries
  - Caching known requests will not work (unless you are smart)
  - Need to have all Qs running at the same time
  - Don’t expect testing in sequence
  - Avoid bottlenecks in mixed queries
Warnings

● Avoid tagging penalties

● Keep a watch on budget.
  ○ $55 (phase + livetest)

● Check correctness before Live Test

● Start early