Principles of Computer Science II

Large Scale Computation

Ioannis Chatzigiannakis

Sapienza University of Rome

Lecture 22

Problem: Lots of data

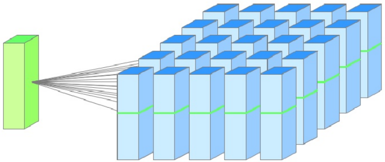
- Example: Homo sapiens high coverage assembly GRCh37
  - 27478 contigs
  - contig length total 3.2 Gb.
  - chromosome length total 3.1 Gb.
- One computer can read 30-35MB/sec from disc
  - $\sim$ 10 months to read the data
- $\sim$ 100 hard drives just to store the data in compressed format
- Even more to do something with the data.

Spread the work over many machines

- Good news: same problem with 1000 machines: $\leq$ 1 hour
- Bad news: concurrency
  - communication and coordination
  - recovering from machine failure
  - status reporting
  - debugging
  - optimization
- Bad news 2: repeat for every problem you want to solve

Computing Clusters

- Many racks of computers
- Thousands of machines per cluster
- Limited bandwidth between racks



Computing Environment

- Each machine has 2-4 CPUs
  - Typically quad-core
  - Future machines will have more cores
- 1-6 locally-attached disks
  - $\sim$ 10TB of disk
- Overall performance more important than peak performance of single machines
- Reliability
  - In 1 server environment, it may stay up for three years (1000 days)
  - If you have 10000 servers, expect to lose 10 each day
- Ultra reliable hardware still fails
  - We need to keep in mind cost of each machine

Map Reduce Computing Paradigm

- A simple programming model
  - Applies to large-scale computing problems
- Hides difficulties of concurrency
  - automatic parallelization
  - load balancing
  - network and disk transfer optimization
  - handling of machine failures
  - robustness
  - improvements to core libraries benefit all users of library

A typical problem

- Read a lot of data
- Map: extract something important from each record
- Shuffle and sort
- Reduce: aggregate, summarize, filter or transform
- Write the results

In more details

- Programmer specifies two primary methods:
  - map$(k, v) \quad \to \quad < k', v' > *$
    - Takes a key-value pair and outputs a set of key-value pairs
    - There is one Map call for every (k, v) pair
  - reduce$(k', < v' > *) \quad \to \quad < k', v' > *$
    - All values v' with same key k' are reduced together and processed in v' order
    - There is one Reduce function call per unique key k'
- All v' with same k' are reduced together, in order.

An example: Frequencies in DNA sequence

A typical exercise for a new engineer in his/her first week:

- Input files with one document per record
- Specify a map function that takes a key/value pair
  - key = document URL
  - value = document contents
- Output of map function is (potentially many) key/value pairs.
- In this case, output:
  (word, 1) once per word in the document

“document 1”, “CTGGGCTAA”
converted to
(C, 1), (T, 1), (G, 1), . . .

An example: Frequencies in DNA sequence

- MapReduce library gathers together all pairs with the same key (shuffle/sort)
- The reduce function combines the values for a key
- In this example:

key = “A”	key = “G”	key = “C”	key = “T”
values = 1, 1	values = 1, 1, 1	values = 1, 1	values = 1, 1
summarize	summarize	summarize	summarize
2	3	2	2

- Output of reduce paired with key and saved

(A, 3), (G, 3), (C, 2), (T, 2)

An example: Frequencies in DNA sequence

```
s = 'CTGGGCTAA'
seq = list(s)  # ['C', 'T', 'G', 'G', 'G', 'C', 'T', 'A', 'A']
sc.parallelize(seq)\
  .map(lambda symbol: (symbol, 1))\
  .reduceByKey(add)\
  .collect()
```

Output:
```
[('A', 2), ('C', 2), ('G', 3), ('T', 2)]
```

Fault tolerance: handled via re-execution

- On worker failure:
  - Detect failure via periodic heartbeats
  - Re-execute completed and in-progress map tasks
  - Re-execute in progress reduce tasks
  - Task completion committed through master
- On master failure:
  - Restart execution

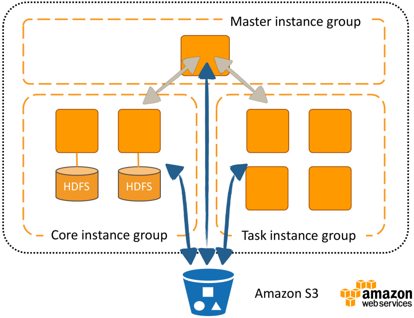
AWS Elastic Map Reduce

- Managed Hadoop framework on EC2 instances.
- AWS EMR splits large processing jobs into smaller jobs and distributes them across many compute nodes in a Hadoop cluster.
- Easily run and scale open-source big data frameworks:
  - Apache Spark
  - Apache Flink
  - Apache Hive
  - Presto
  - Apache HBase
  - ...
- EMR Notebooks.

EMR: Benefits

- Easy to use – interact using Jupyter via web.
- Low cost
  - Pay a per-instance rate for every second used, with a one-minute minimum charge.
- Elastic
  - For short-running jobs, you can spin up and spin down clusters and pay per second for the instances used.
  - For long-running workloads, you can create highly available clusters that automatically scale to meet demand.
- Reliable
- Secure
- Flexible

Apache EMR Architecture



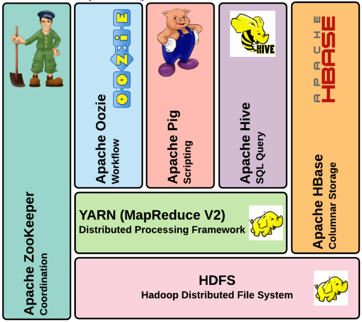
AWS EMR and Apache Hadoop

- The Elastic Map Reduce is built on top Apache Hadoop.
- An open-source Java software framework that supports massive data processing across a cluster of instances.
- Distributed processing across the instances that make up the cluster.
- It can run on a single instance or thousands of instances.
- Elastic auto-scaling of cluster.
- Provides a fault-tolerant processing environment.

Apache Hadoop

- Apache Hadoop includes the following modules:
  - Hadoop Common: The common utilities that support the other Hadoop modules.
  - Hadoop Distributed File System (HDFS): A distributed file system that provides high-throughput access to application data.
  - Hadoop YARN: A framework for job scheduling and cluster resource management.
  - Hadoop MapReduce: A YARN-based system for parallel processing of large data sets.
  - Hadoop Ozone: An object store for Hadoop.

Apache Hadoop Ecosystem



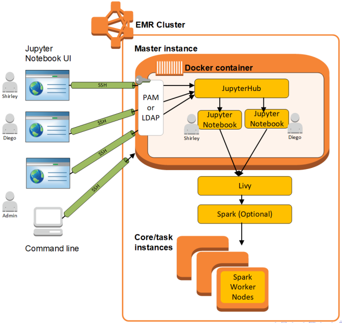
Apache Spark on AWS EMR

- Started in 2009 as a research project at UC Berkley's AMPLab.
- An open-source, distributed processing system used for big data workloads.
- In contrast to Hadoop, uses in-memory caching to achieve high speed-ups.
  - Optimized query execution for fast analytic queries against data of any size.
- Development APIs in Java, Scala, Python and R.
- Supports code reuse across multiple workloads-batch processing:
  - interactive queries, real-time analytics, machine learning, and graph processing.

EMR Notebooks

- EMR Notebooks is a Jupyter Notebook environment built in to the Amazon EMR console.
- Quickly create Jupyter notebooks, attach them to Spark clusters
- Use Jupyter Notebook editor to remotely run queries and code.
- Open, attach multiple notebooks to a single cluster, and re-use a notebook on different clusters.
- You can start a cluster, attach an EMR notebook for analysis, and then terminate the cluster.
- You can also close a notebook attached to one running cluster and switch to another.
- Multiple users can attach notebooks to the same cluster simultaneously.

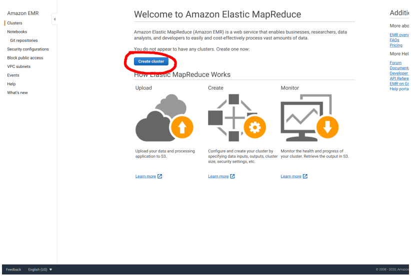
EMR Notebooks architecture



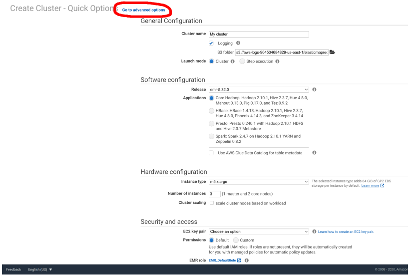
Considerations when using EMR Notebooks

- User notebooks and files are saved to the file system on the master node.
- This is ephemeral storage that does not persist through cluster termination.
- When a cluster terminates, this data is lost if not backed up.
- EMR Notebooks support persistance to S3.
- EMR Notebooks supports connection with GitHub repositories.

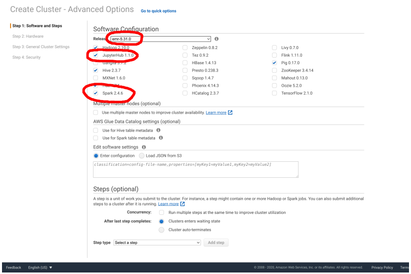
Create Cluster



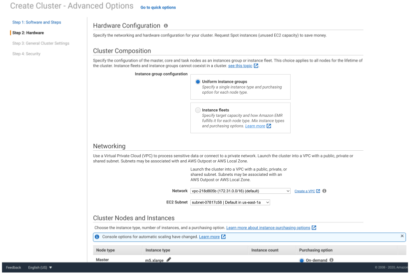
Quick Options



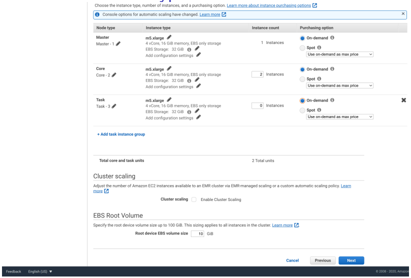
Software Configuration



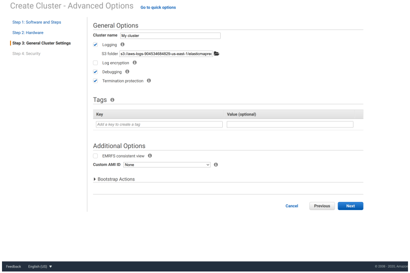
Hardware Configuration



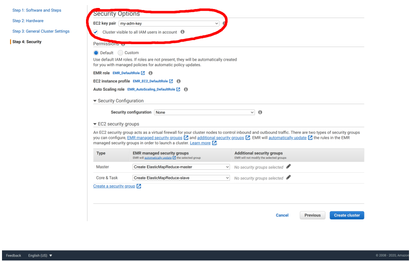
Choose Instance Types



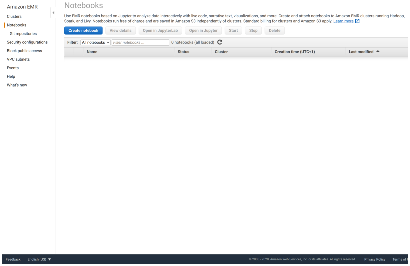
General Cluster Settings



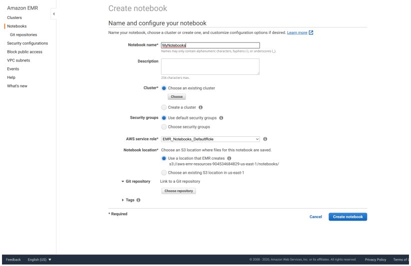
Security Options



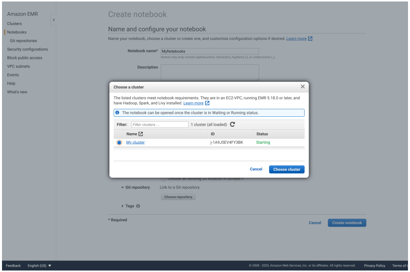
EMR Notebooks



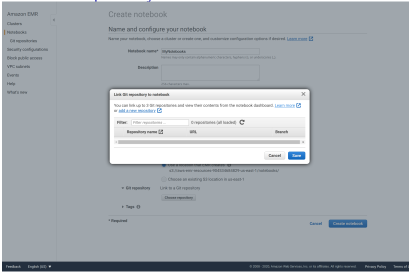
Create notebook



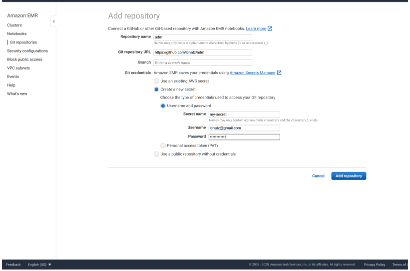
Choose a cluster



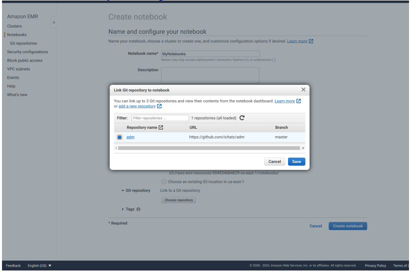
Link Git repository to notebook



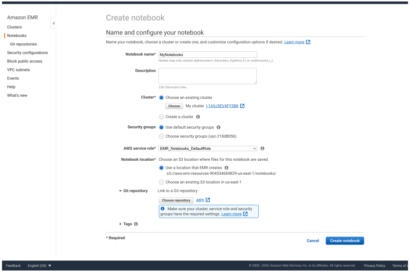
Add repository



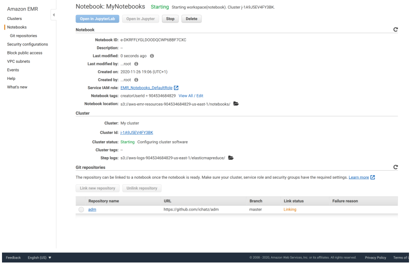
Link Git repository to notebook



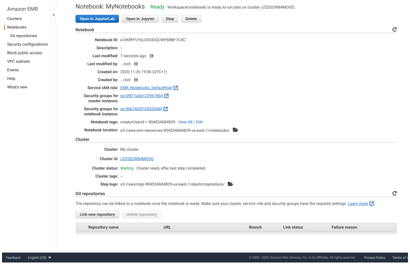
Create notebook



Starting notebook

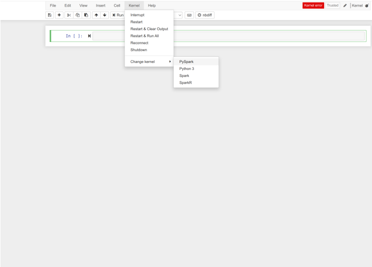


EMR Notebooks: ready

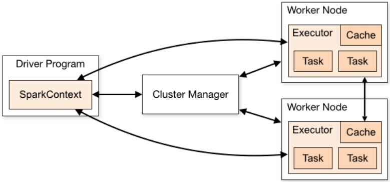


Jupyter Tree

Choose PySpark Kernel



Spark Driver Program



- Map/Reduce operations are issued to the cluster manager.
- Map/Reduce operations work on a given dataset.
- The dataset is encoded using the RDD structure.

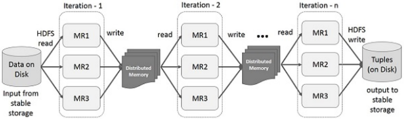
Spark Context

- SparkContext is the entry point to any spark functionality.
- A SparkContext represents the connection to a Spark cluster.
- Used to create RDD and broadcast variables on that cluster.
- Only one SparkContext should be active per session.

Resilient Distributed Datasets

- A fundamental data structure of Spark.
- Spark makes use RDD to achieve faster and efficient MapReduce operations.
- An immutable distributed read-only collection of objects.
  - immutable = state cannot change after it is constructed.
  - Can contain any type of Python, Java, or Scala objects, including user-defined classes.
- Two ways to construct an RDD:
  1. Referencing a dataset in an external storage system: S3, HDFS, HBase, ...
  2. Through Map/Reduce opreations.
- RDD is divided into logical partitions.
  - Each logical partition may be computed on different nodes of the cluster.

Iterative Operations on MapReduce



- Reuse intermediate results across multiple computations in multi-stage applications.
- Each Map/Reduce operation works on a given/input RDD.
- Each Map/Reduce operation constructs/outputs a new RDD.
- If the Distributed memory (RAM) is not sufficient to store intermediate RDD, then it will store those results on the disk.

Dataset

- Download dataset `http://hplgit.github.io/bioinf-py/data/yeast_chr1.txt`
- Upload dataset to S3

Retrieve dataset

- Use the variable sc to access the Spark Context.
- Load the text file to Spark.

```
raw_data = sc.textFile("s3://.../yeast_chr1.txt")
```

- textFile Creates a new RDD object.
- Provides a reference to the dataset - no data loaded yet.

```
raw_data.count()
```

- Retrieves the dataset from s3.
- Creates 1 entry in the RDD for each line of the text file.
- Counts the entries in the RDD object.

```
raw_data.take(5)
```

- Take the first 5 elements of the RDD.

Spark Monitoring Job Progress

Sample App

```
def splitLine(line):
  pairs = []
  symbols = list(line)
  if len(booksymbols) > 1:
    for symbol in symbols:
      pairs.append([symbol, 1])

  return pairs

pairs = raw_data.flatMap(splitLine)
print(pairs.take(10))

final = pairs.reduceByKey(lambda a,b: a + b)
print(final.collect())
```

Profile most-frequent k-mer

CGGGGCTGGGTCGTCACATTCCCCTTTCGATA
TTTGAGGGTGCCCAATAACCAAAGCGGACAAA
GGGATGCCGTTTGACGACCTAAATCAACGGCC
AAGGCCAGGAGCGCCTTTGCTGGTTCTACCTG
AATTTTCTAAAAAGATTATAATGTCGGTCCTC
CTGCTGTACAACTGAGATCATGCTGCTTCAAC
TACATGATCTTTTGTGGATGAGGGAATGATGC

- Seven (7) 32-nucleotide DNA sequences
- A "secret" pattern P=ATGCAACT of length $l = 8$ implanted.
- Can you reconstruct P by analyzing the DNA sequences?

An example

CGGGGCTATGCAACTGGGTCGTCACATTCCCCTTTCGATA
TTTGAGGGTGCCCAATAAATGCAACTCCAAAGCGGACAAA
GGATGCAACTGATGCCGTTTGACGACCTAAATCAACGGCC
AAGGATGCAACTCCAGGAGCGCCTTTGCTGGTTCTACCTG
AATTTTCTAAAAAGATTATAATGTCGGTCCATGCAACTTC
CTGCTGTACAACTGAGATCATGCTGCATGCAACTTTCAAC
TACATGATCTTTTGATGCAACTTGGATGAGGGAATGATGC

- The same DNA sequences with the implanted pattern ATGCAACT
- Can you spot the locations of the implanted pattern?

An example

CGGGGCTATGCAACTGGGTCGTCACATTCCCCTTTCGATA
TTTGAGGGTGCCCAATAAATGCAACTCCAAAGCGGACAAA
GGATGCAACTGATGCCGTTTGACGACCTAAATCAACGGCC
AAGGATGCAACTCCAGGAGCGCCTTTGCTGGTTCTACCTG
AATTTTCTAAAAAGATTATAATGTCGGTCCATGCAACTTC
CTGCTGTACAACTGAGATCATGCTGCATGCAACTTTCAAC
TACATGATCTTTTGATGCAACTTGGATGAGGGAATGATGC

- Same as before but showing the implant locations.
- Devise an MapReduce algorithm to automatically identify the implanted pattern
- Length l is known.
- Sequence: `https://goo.gl/xN7WvE`

```
from operator import add

conf = SparkConf().setAppName("Profile").setMaster("local")
sc = SparkContext(conf=conf)

val raw_data = sc.wholeTextFiles("pattern.txt")

def splitLine(line):
  pairs = []
  if len(line) > 1:
    for symbol in range(0, len(line)-8):
      pairs.append((line[symbol:symbol+8], 1))

  return pairs

pairs = raw_data.flatMap(splitLine) \
                .reduceByKey(add) \
                .sortBy(lambda a: -a[1])
print(pairs.take(5))
```

Profile most-Frequent first appearing k-mer

CGGGGCTGGGTCGTCACATTCCCCTTTCGATA
TTTGAGGGTGCCCAATAACCAAAGCGGACAAA
GGGATGCCGTTTGACGACCTAAATCAACGGCC
AAGGCCAGGAGCGCCTTTGCTGGTTCTACCTG
AATTTTCTAAAAAGATTATAATGTCGGTCCTC
CTGCTGTACAACTGAGATCATGCTGCTTCAAC
TACATGATCTTTTGTGGATGAGGGAATGATGC

- Identify most-frequent first-appearing k-mer
- In first line, 3-mer CGG appears before GGC, TGG, GTC, ...

```python
from operator import add

conf = SparkConf().setAppName("Profile").setMaster("local")
sc = SparkContext(conf=conf)

raw_data = sc.textFile("yeast_chr1.txt")

def splitLine(line):
  pairs = []
  for symbol in range(0, len(line)-6, 3):
      for second in range(symbol+3, len(line)-6, 3):
          pairs.append(((line[symbol:symbol+3],
                         line[second:second+3]), 1))

  return pairs

pairs = raw_data.flatMap(splitLine) \
                .reduceByKey(add) \
                .sortBy(lambda a: -a[1])
print(pairs.take(10))
```