EE 542

Lecture 13: Hadoop and Spark in the Cloud

Internet and Cloud Computing

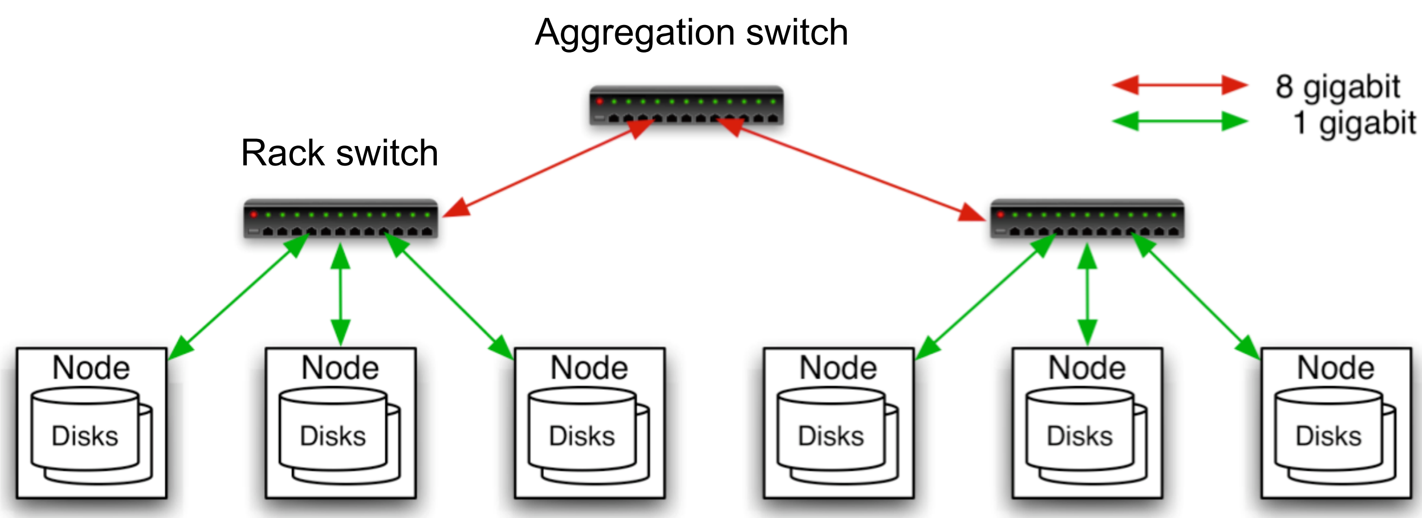
Young Cho
Department of Electrical Engineering
University of Southern California

Some of the Slides borrowed from UC Berkeley AMP Lab

MapReduce

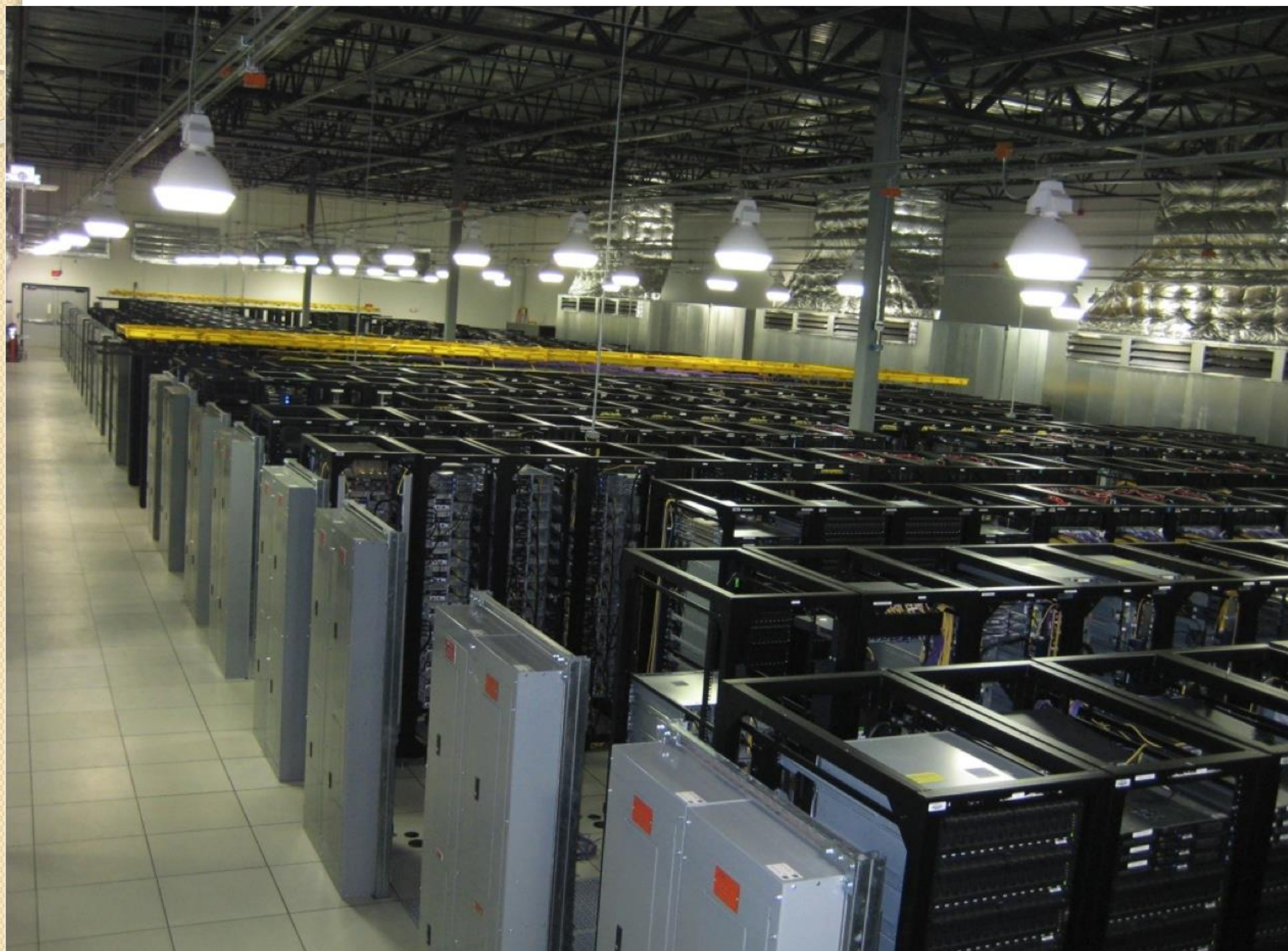
- Programming model for data-intensive computing on commodity clusters
- Pioneered by Google
  - Processes 20 PB of data per day
- Popularized by Apache Hadoop project
  - Used by Yahoo!, Facebook, Amazon, …

Hadoop Cluster



- 40 nodes/rack, 1000-4000 nodes in cluster
- 1 Gbps bandwidth in rack, 8 Gbps out of rack
- Node specs (Facebook):
  8-16 cores, 32 GB RAM, 8 × 1.5 TB disks, no RAID

Hadoop Cluster



Challenges of Cloud Environment

- Cheap nodes fail, especially when you have many
  - Mean time between failures for 1 node = 3 years
  - MTBF for 1000 nodes = 1 day
  - **Solution:** Build fault tolerance into system

- Commodity network = low bandwidth
  - **Solution:** Push computation to the data

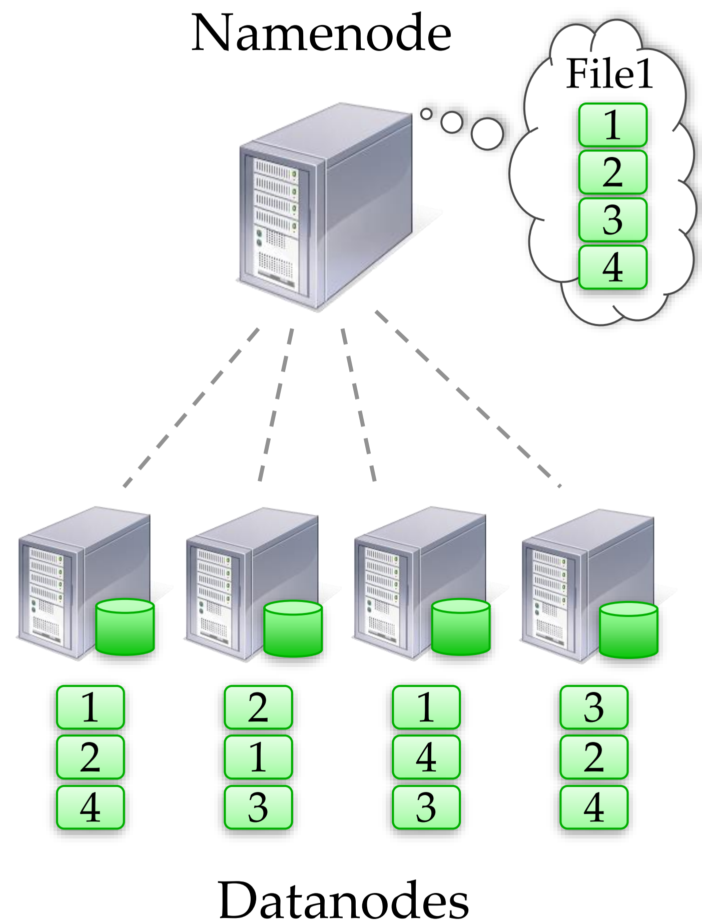
- Programming distributed systems is hard
  - **Solution:** Restricted programming model: users write data-parallel "map" and "reduce" functions, system handles work distribution and failures

Hadoop Components

- Distributed file system (HDFS)
  - Single namespace for entire cluster
  - Replicates data 3x for fault-tolerance
- MapReduce framework
  - Runs jobs submitted by users
  - Manages work distribution & fault-tolerance
  - Colocated with file system

Hadoop Distributed File System

- Files split into 128MB blocks
- Blocks replicated across several datanodes (often 3)
- Namenode stores metadata (file names, locations, etc)
- Optimized for large files, sequential reads
- Files are append-only



MapReduce Execution Details

- Mappers preferentially scheduled on same node or same rack as their input block
  - Minimize network use to improve performance
- Mappers save outputs to local disk before serving to reducers
  - Allows recovery if a reducer crashes
  - Allows running more reducers than # of nodes

Fault Tolerance in MapReduce

1. If a task crashes:
   - Retry on another node
     - OK for a map because it had no dependencies
     - OK for reduce because map outputs are on disk
   - If the same task repeatedly fails, fail the job or ignore that input block

➢ Note: For the fault tolerance to work, *user tasks must be deterministic and side-effect-free*

Fault Tolerance in MapReduce

2. If a node crashes:
   - Relaunch its current tasks on other nodes
   - Relaunch any maps the node previously ran
     - Necessary because their output files were lost along with the crashed node

Fault Tolerance in MapReduce

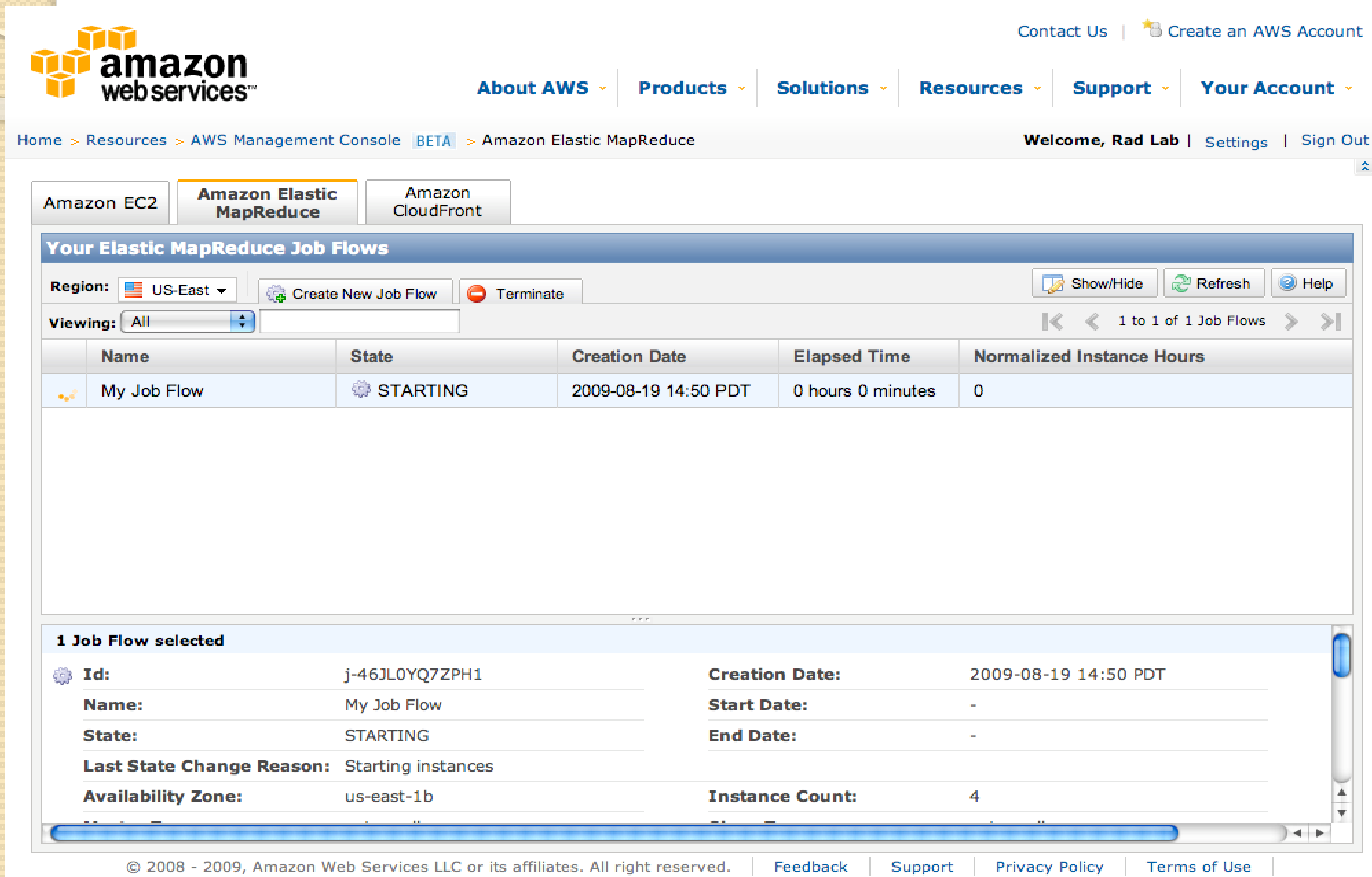
3. If a task is going slowly (straggler):
   - Launch second copy of task on another node
   - Take the output of whichever copy finishes first, and kill the other one

- Critical for performance in large clusters (many possible causes of stragglers)

Amazon Elastic MapReduce

- Web interface and command-line tools for running Hadoop jobs on EC2
- Data stored in Amazon S3
- Monitors job and shuts machines after use

Elastic MapReduce UI



MapReduce

- Programming model for data-intensive computing on commodity clusters
- Pioneered by Google
  - Processes 20 PB of data per day
- Popularized by Apache Hadoop project
  - Used by Yahoo!, Facebook, Amazon, …
- Industry Trending Toward Spark
  - Quickly Being Adopted by Big Data Industry

Limitations of MapReduce

- MapReduce is great at one-pass computation, but inefficient for *multi-pass* algorithms
- No efficient primitives for data sharing
- State between steps goes to distributed file system
- Slow due to replication & disk storage
- No control of data partitioning across steps

Spark Programming Model

- Extends MapReduce with primitives for efficient data sharing
  - "Resilient distributed datasets"
- Open source in Apache Incubator
  - Growing community with 100+ contributors
- APIs in Java, Scala & Python

Resilient Distributed Datasets (RDDs)

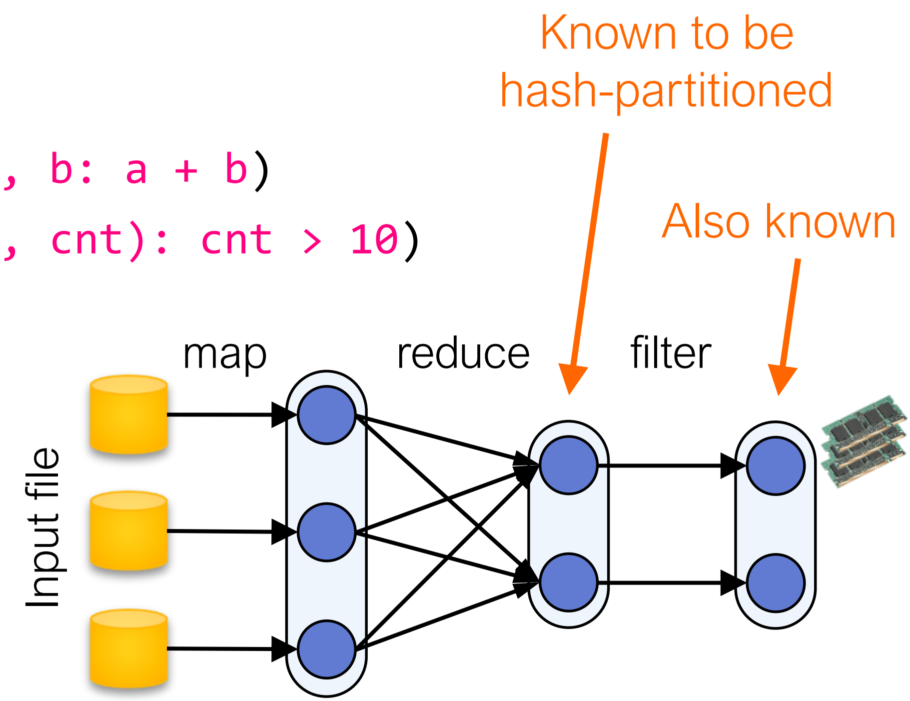
- Collections of objects stored across a cluster
- User-controlled partitioning & storage (RAM, disk, …)
- Automatically rebuilt on failure

```
urls = spark.textFile("hdfs://...")
records = urls.map(lambda s: (s, 1))
counts = records.reduceByKey(lambda a, b: a + b)
bigCounts = counts.filter(lambda (url, cnt): cnt > 10)
```

```
bigCounts.cache()

bigCounts.filter(
  lambda (k,v): "news" in k).count()

bigCounts.join(otherPartitionedRDD)
```



Spark

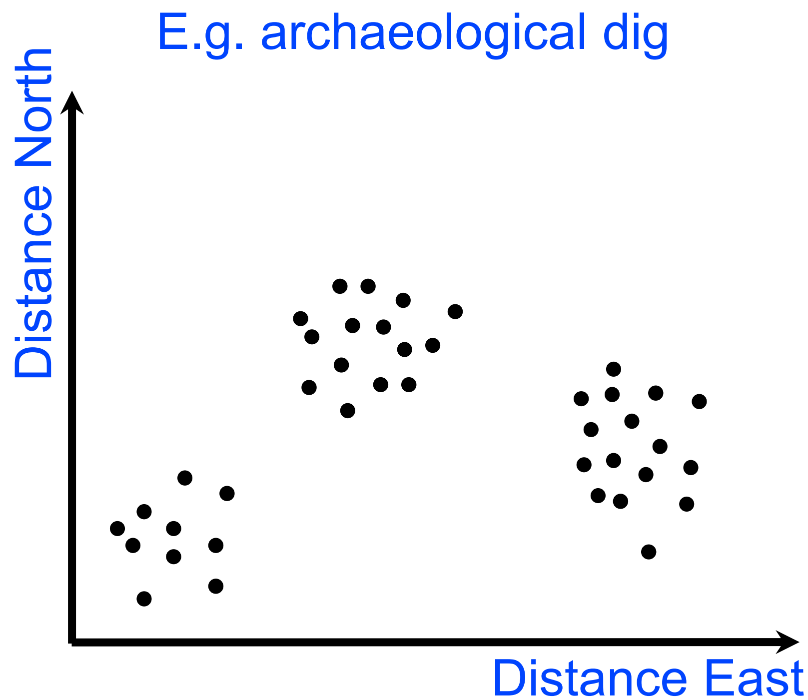
- In-Memory Computation
  - For 64-bit computers TB of data in RAM
  - Designed to transform data in-mem and not in disk
  - Supports parallel distributed processing of data
  - 100x in memory and 10x on disk then Hadoop
- General programming model
  - Use normal sequential programming
  - No need for maps and reduce operations

Spark: Key Advantages

- Ability for On-disk Data Sorting
  - Tuned for large scale of data sorting on disk
  - The world record of on-disk large scale data sorting
- Efficient Use of Cache
  - Mesos which is a distributed system kernel for caching the intermediate dataset
  - Multiple iterations on the cached dataset
- In-memory Tuned Library
  - MLlib library for in-memory tuned operations
- Faster Launch with Virtual Machine

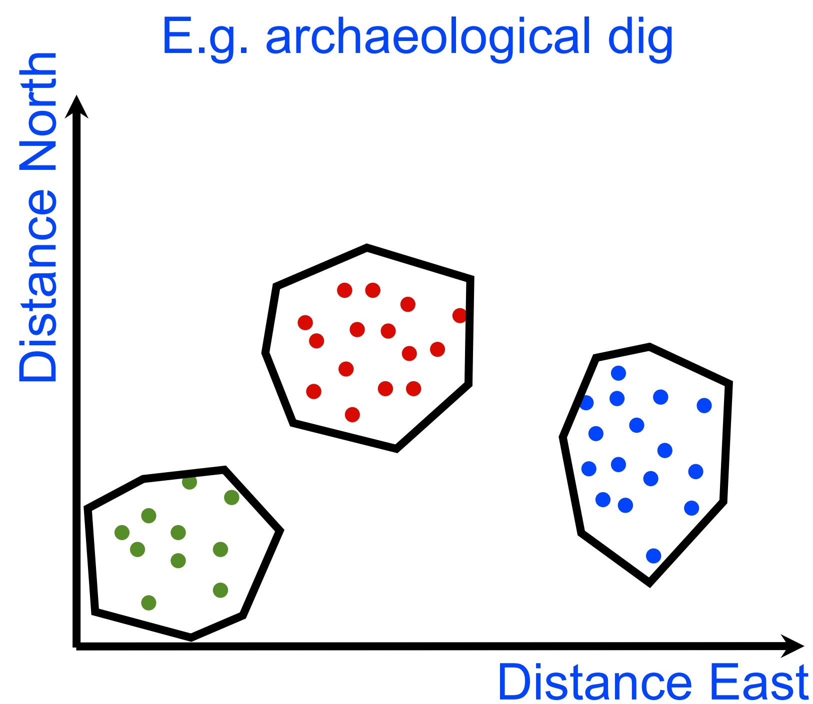
Clustering

Grouping data according to similarity



Clustering

Grouping data according to similarity

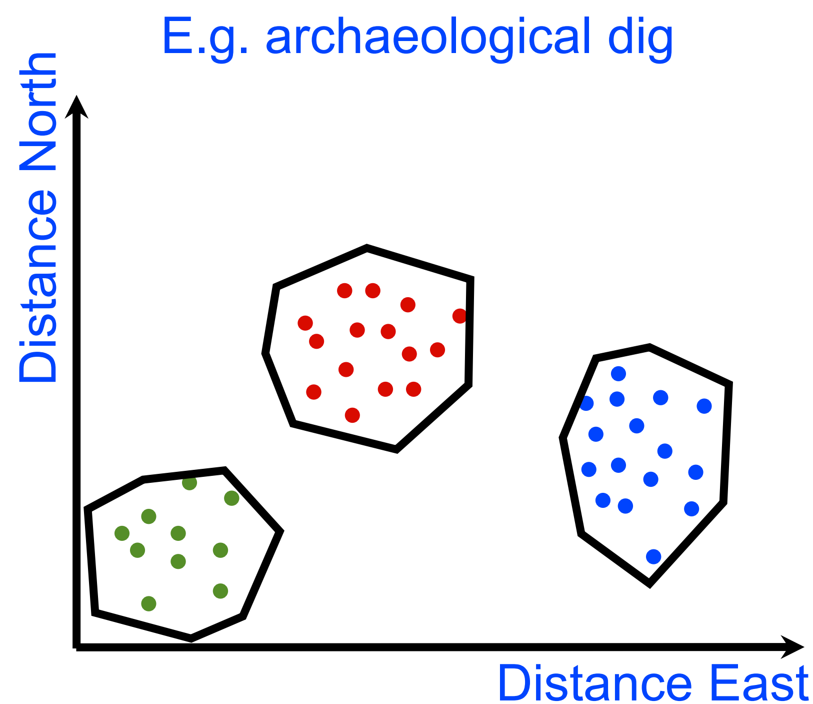


Clustering with Spark

K-Means: preliminaries

Benefits

- Popular
- Fast
- Conceptually straightforward

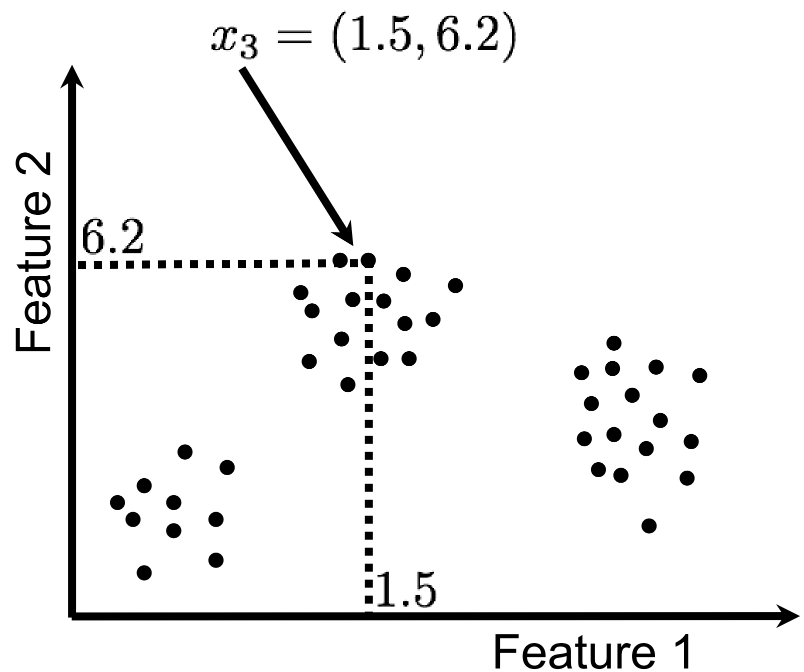


Clustering with Spark

K-Means: preliminaries

Data: Collection of values

```
data = lines.map(line=>
    parseVector(line))
```

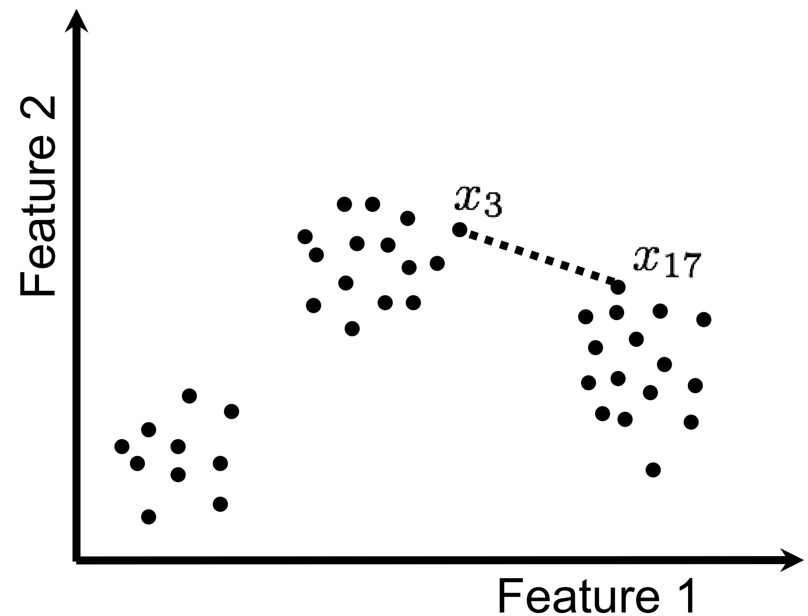


Clustering with Spark

K-Means: preliminaries

Dissimilarity:
Squared Euclidean distance

```
dist = p.squaredDist(q)
```



Clustering with Spark

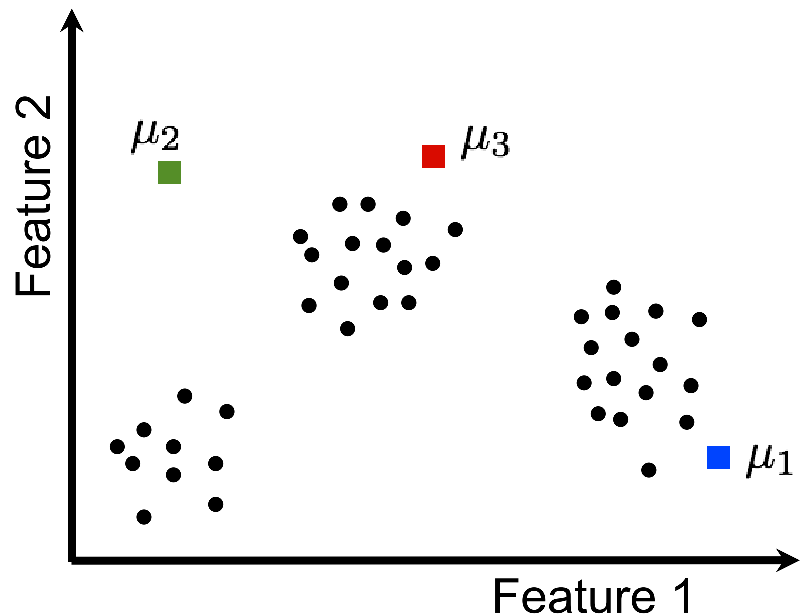
K-Means: preliminaries

K = Number of clusters

$\mu_1, \mu_2, \ldots, \mu_K$

Data assignments to clusters

$S_1, S_2, \ldots, S_K$



Clustering with Spark

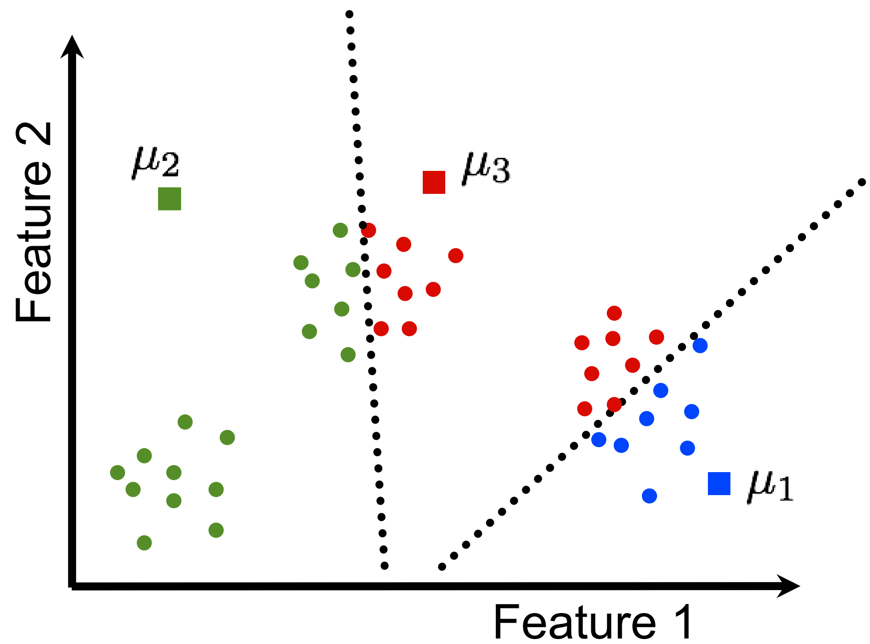
K-Means: preliminaries

K = Number of clusters

$\mu_1, \mu_2, \ldots, \mu_K$

Data assignments to clusters

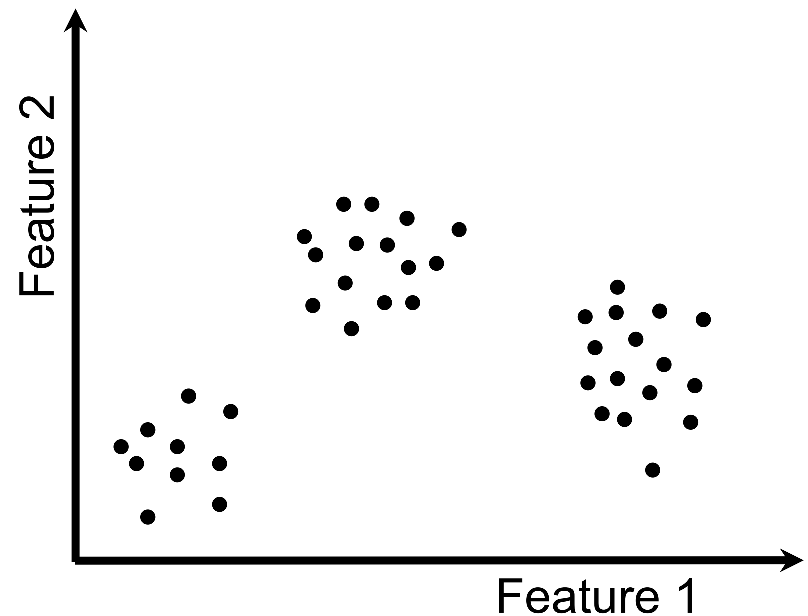
$S_1, S_2, \ldots, S_K$



Clustering with Spark

K-Means Algorithm

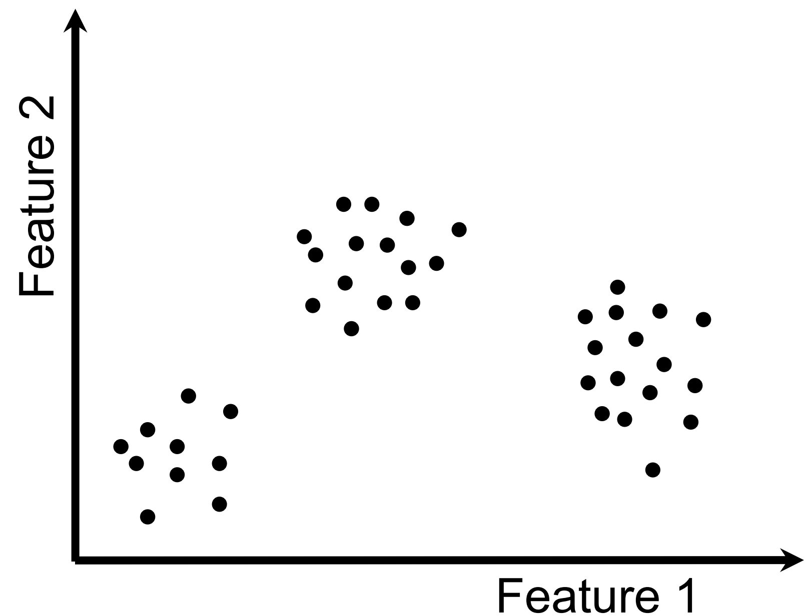
- Initialize K cluster centers
- Repeat until convergence:
    Assign each data point to the cluster with the closest center.
    Assign each cluster center to be the mean of its cluster's data points.



Clustering with Spark

K-Means Algorithm

- Initialize K cluster centers
- Repeat until convergence:
    Assign each data point to the cluster with the closest center.
    Assign each cluster center to be the mean of its cluster's data points.



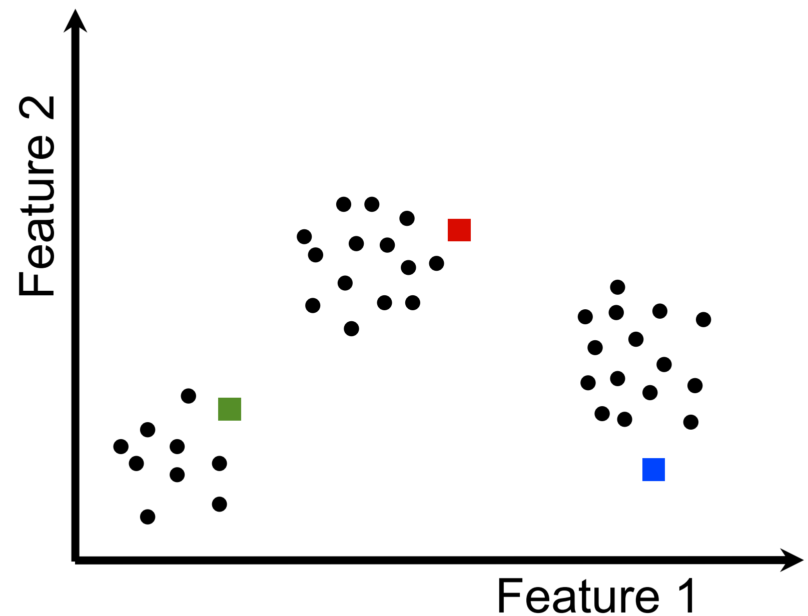
Clustering with Spark

K-Means Algorithm

- Initialize K cluster centers

```
centers = data.takeSample(
    false, K, seed)
```

- Repeat until convergence:
  Assign each data point to the cluster with the closest center.
  Assign each cluster center to be the mean of its cluster's data points.



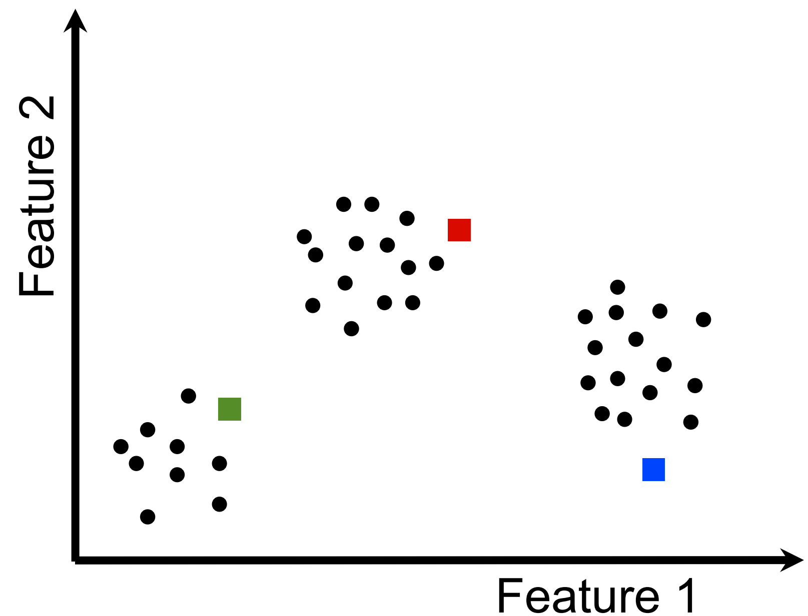
Clustering with Spark

K-Means Algorithm

- Initialize K cluster centers

```
centers = data.takeSample(
    false, K, seed)
```

- Repeat until convergence:
  Assign each data point to the cluster with the closest center.
  Assign each cluster center to be the mean of its cluster's data points.



Clustering with Spark

K-Means Algorithm

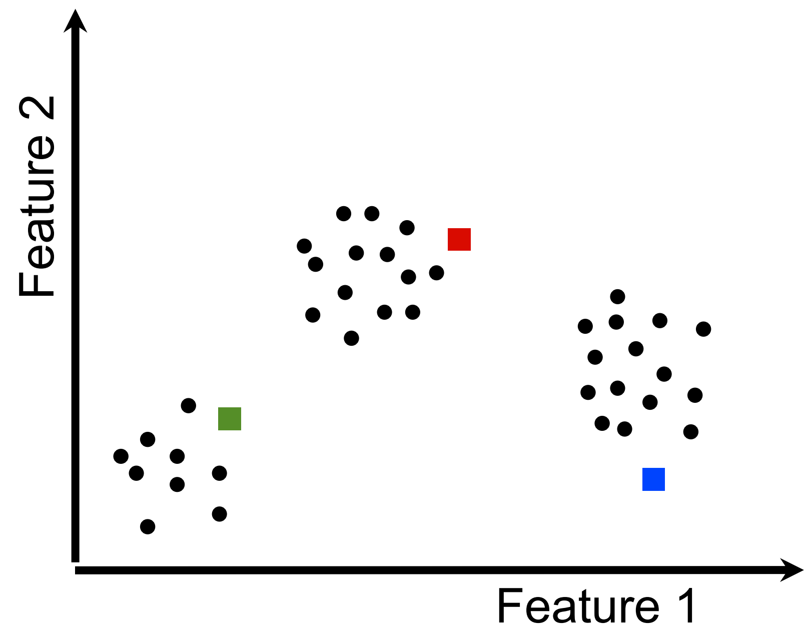
- Initialize K cluster centers

```
centers = data.takeSample(
    false, K, seed)
```

- Repeat until convergence:

  Assign each data point to the cluster with the closest center.

  Assign each cluster center to be the mean of its cluster's data points.



Clustering with Spark

K-Means Algorithm

- Initialize K cluster centers

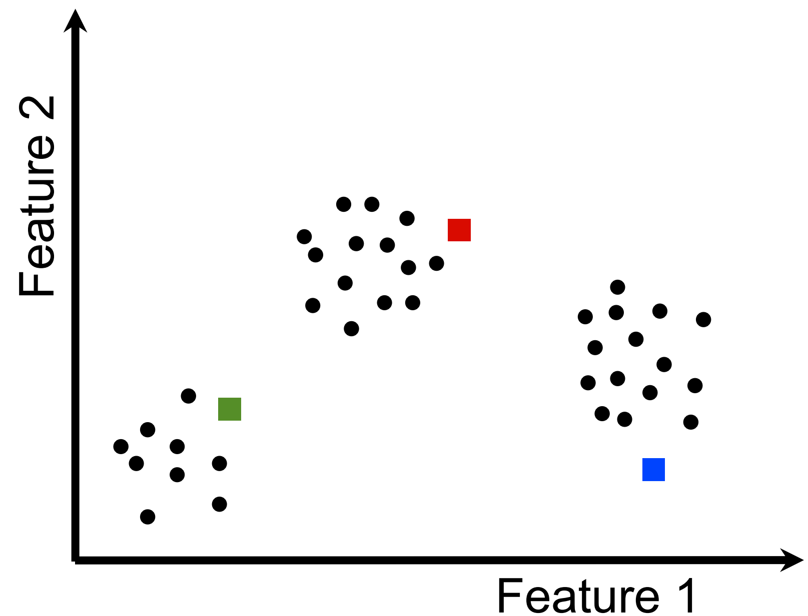
```
centers = data.takeSample(
    false, K, seed)
```

- Repeat until convergence:

```
closest = data.map(p =>

(closestPoint(p,centers),p))
```

Assign each cluster center to be the mean of its cluster's data points.



Clustering with Spark

K-Means Algorithm

- Initialize K cluster centers

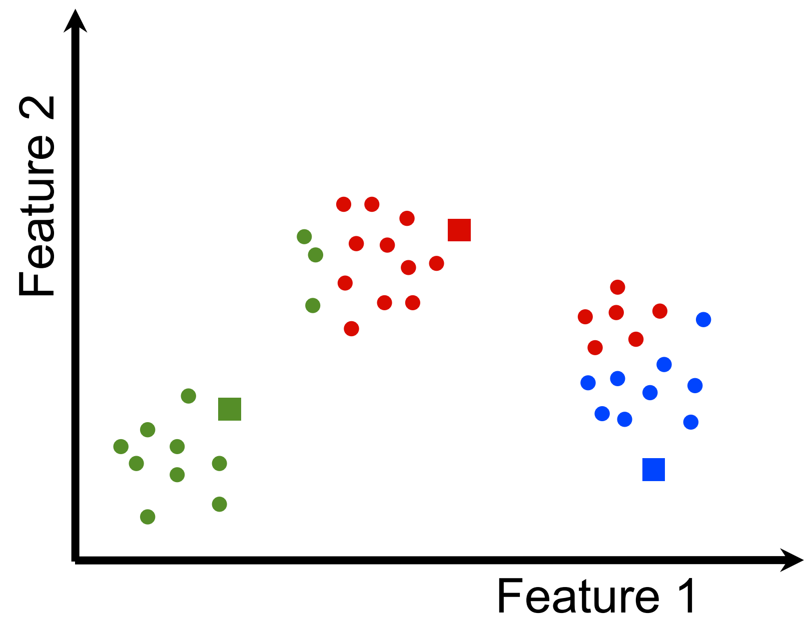
```
centers = data.takeSample(
    false, K, seed)
```

- Repeat until convergence:

```
closest = data.map(p =>

(closestPoint(p,centers),p))
```

Assign each cluster center to be the mean of its cluster’s data points.



Clustering with Spark

K-Means Algorithm

- Initialize K cluster centers

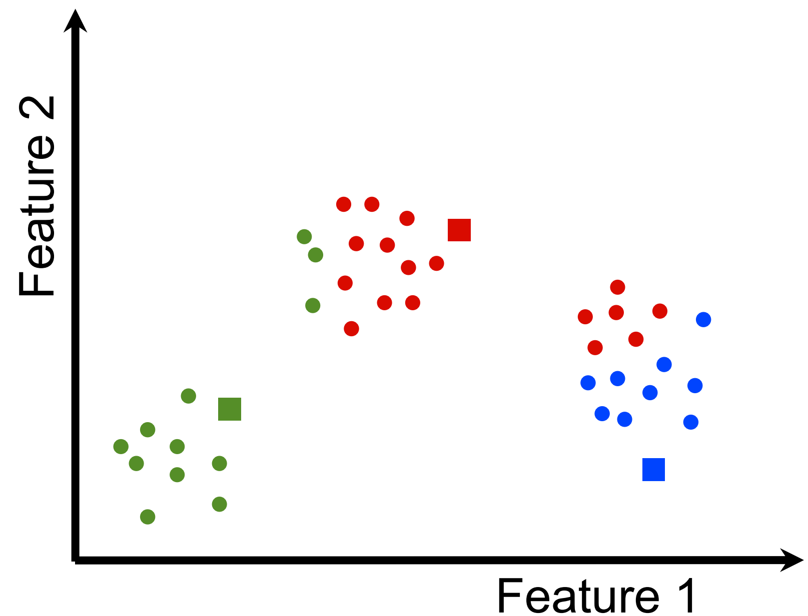
```
centers = data.takeSample(
    false, K, seed)
```

- Repeat until convergence:

```
closest = data.map(p =>

(closestPoint(p,centers),p))
```

Assign each cluster center to be the mean of its cluster’s data points.



Clustering with Spark

K-Means Algorithm

- Initialize K cluster centers

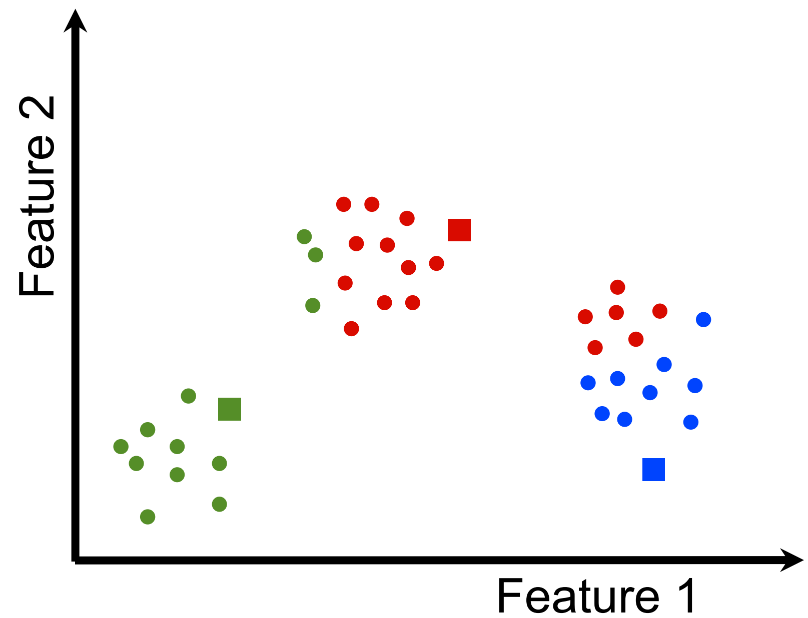
```
centers = data.takeSample(
    false, K, seed)
```

- Repeat until convergence:

```
closest = data.map(p =>

(closestPoint(p,centers),p))

pointsGroup =
    closest.groupByKey()
```



Clustering with Spark

K-Means Algorithm

- Initialize K cluster centers

```
centers = data.takeSample(
    false, K, seed)
```

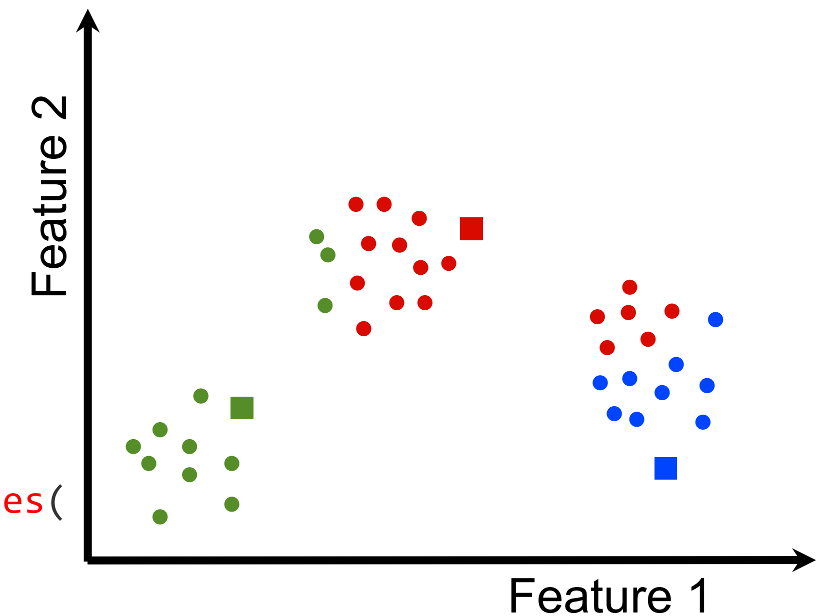
- Repeat until convergence:

```
closest = data.map(p =>

(closestPoint(p,centers),p))

pointsGroup =
    closest.groupByKey()

newCenters = pointsGroup.mapValues(
    ps => average(ps))
```



Clustering with Spark

K-Means Algorithm

- Initialize K cluster centers

```
centers = data.takeSample(
    false, K, seed)
```

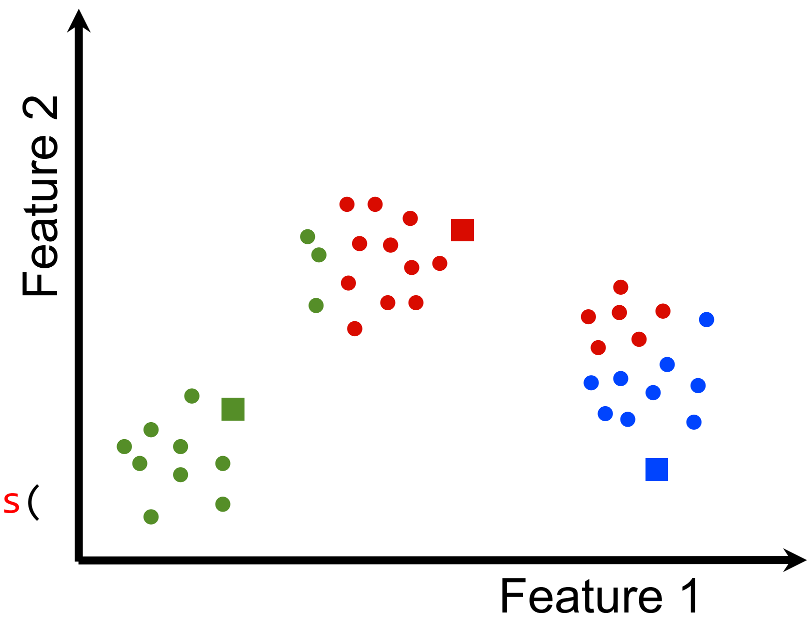
- Repeat until convergence:

```
closest = data.map(p =>

(closestPoint(p,centers),p))
```

```
pointsGroup =
    closest.groupByKey()
```

```
newCenters = pointsGroup.mapValues(
    ps => average(ps))
```



Clustering with Spark

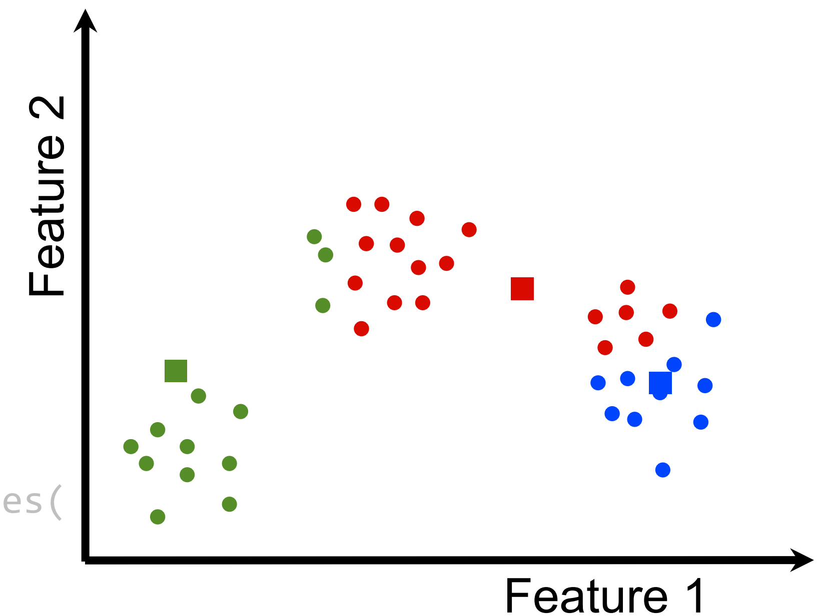
K-Means Algorithm

- Initialize K cluster centers

```
centers = data.takeSample(
    false, K, seed)
```

- Repeat until convergence:

```
closest = data.map(p =>
   (closestPoint(p,centers),p))
pointsGroup =
    closest.groupByKey()
newCenters = pointsGroup.mapValues(
    ps => average(ps))
```



Clustering with Spark

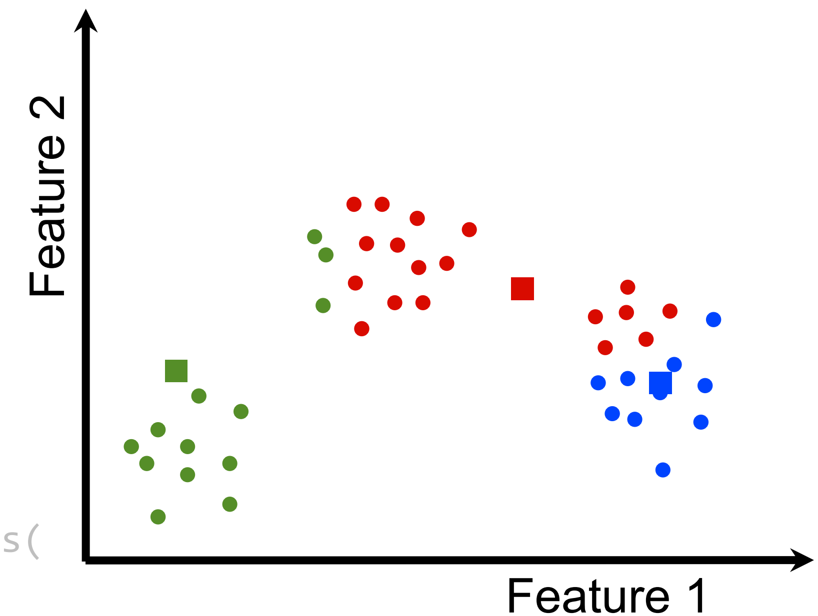
K-Means Algorithm

- Initialize K cluster centers

```
centers = data.takeSample(
    false, K, seed)
```

- Repeat until convergence:

```
while (dist(centers,
             newCenters) > ε)
closest = data.map(p =>
   (closestPoint(p,centers),p))
pointsGroup =
    closest.groupByKey()
newCenters =pointsGroup.mapValues(
    ps => average(ps))
```



Clustering with Spark

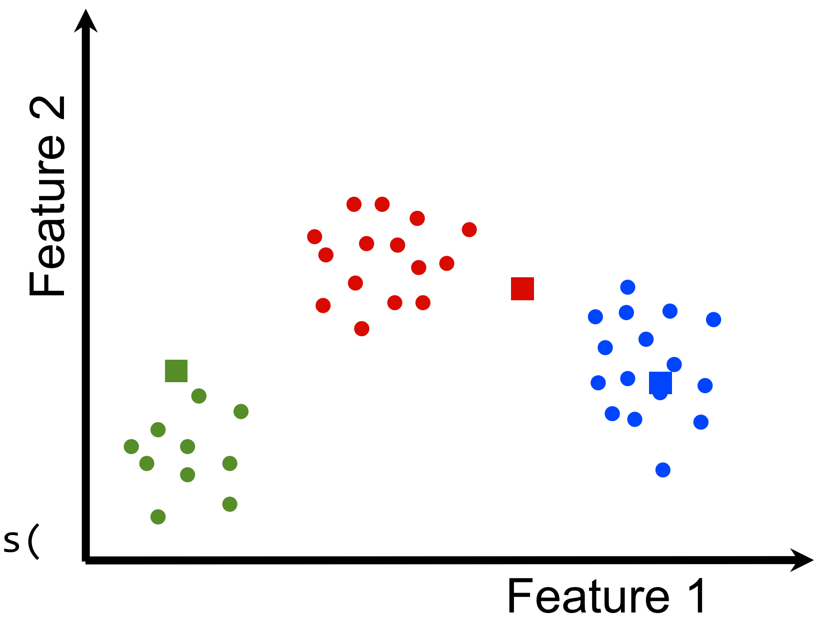
K-Means Algorithm

- Initialize K cluster centers

```
centers = data.takeSample(
    false, K, seed)
```

- Repeat until convergence:

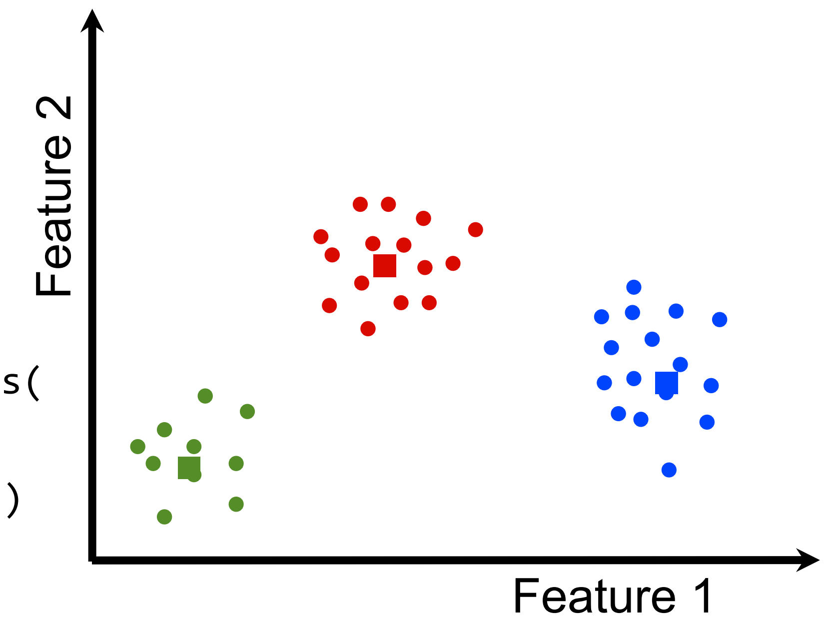
```
while (dist(centers,
            newCenters) > ε)
closest = data.map(p =>
   (closestPoint(p,centers),p))
pointsGroup =
    closest.groupByKey()
newCenters =pointsGroup.mapValues(
    ps => average(ps))
```



Clustering with Spark

K-Means Source

```
centers = data.takeSample(
    false, K, seed)
while (d > ε)
{
  closest = data.map(p =>
     (closestPoint(p,centers),p))
  pointsGroup =
      closest.groupByKey()
  newCenters =pointsGroup.mapValues(
      ps => average(ps))
  d = distance(centers, newCenters)
  centers = newCenters.map(_)
}
```

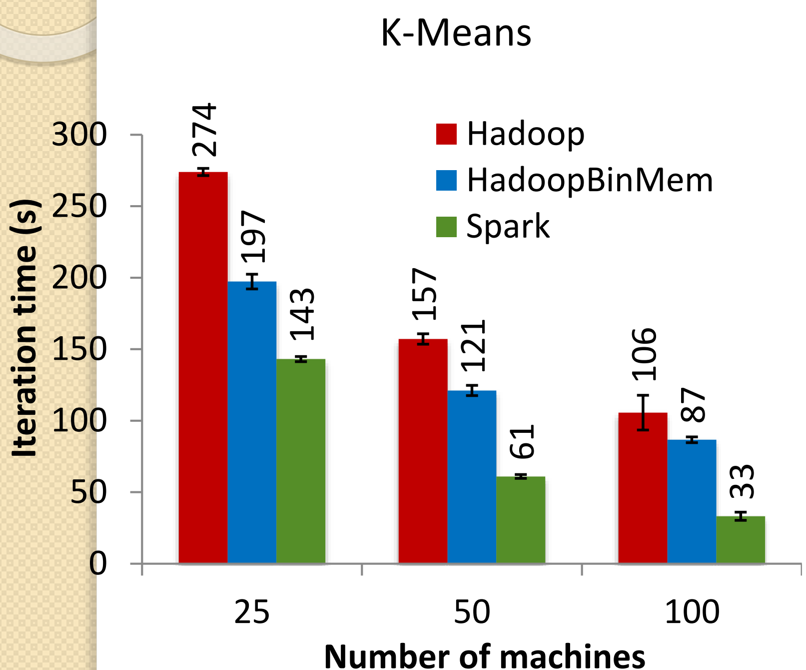


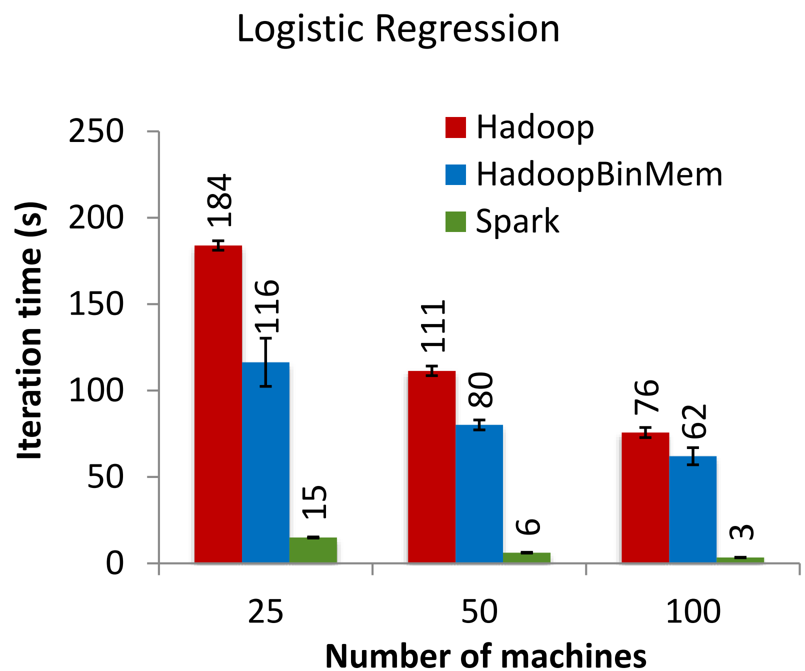
Clustering with Spark

Ease of use

- Interactive shell:
  - Useful for featurization, pre-processing data
- Lines of code for K-Means
  - Spark ~ 90 lines
  - Hadoop/Mahout ~ 4 files, > 300 lines

Performance





[Zaharia et. al, NSDI'12]

K-Means in MLlib

- http://spark.apache.org/docs/latest/mllib-clustering.html#k-means
- Available for Multiple Languages
  - Scala
  - Java
  - Python