Northeastern University - Seattle

CS6650 Building Scalable Distributed Systems

Professor Ian Gorton

Building Scalable Distributed Systems

Week 12 – Scaling Analytics

Outline

- Motivation
- Hadoop
- Spark

Motivation

Scalable Data Analysis

R (257)	45%
SQL (184)	32%
Python (140)	25%
Java (139)	24%
SAS (121)	21%
MATLAB (83)	15%
C/C++ (73)	13%
Unix shell/awk/gawk/sed (59)	10%
Perl (45)	7.9%
Hadoop/Pig/Hive (35)	6.1%
Lisp (4)	0.7%
Other (70)	12.0%
None (7)	1.2%

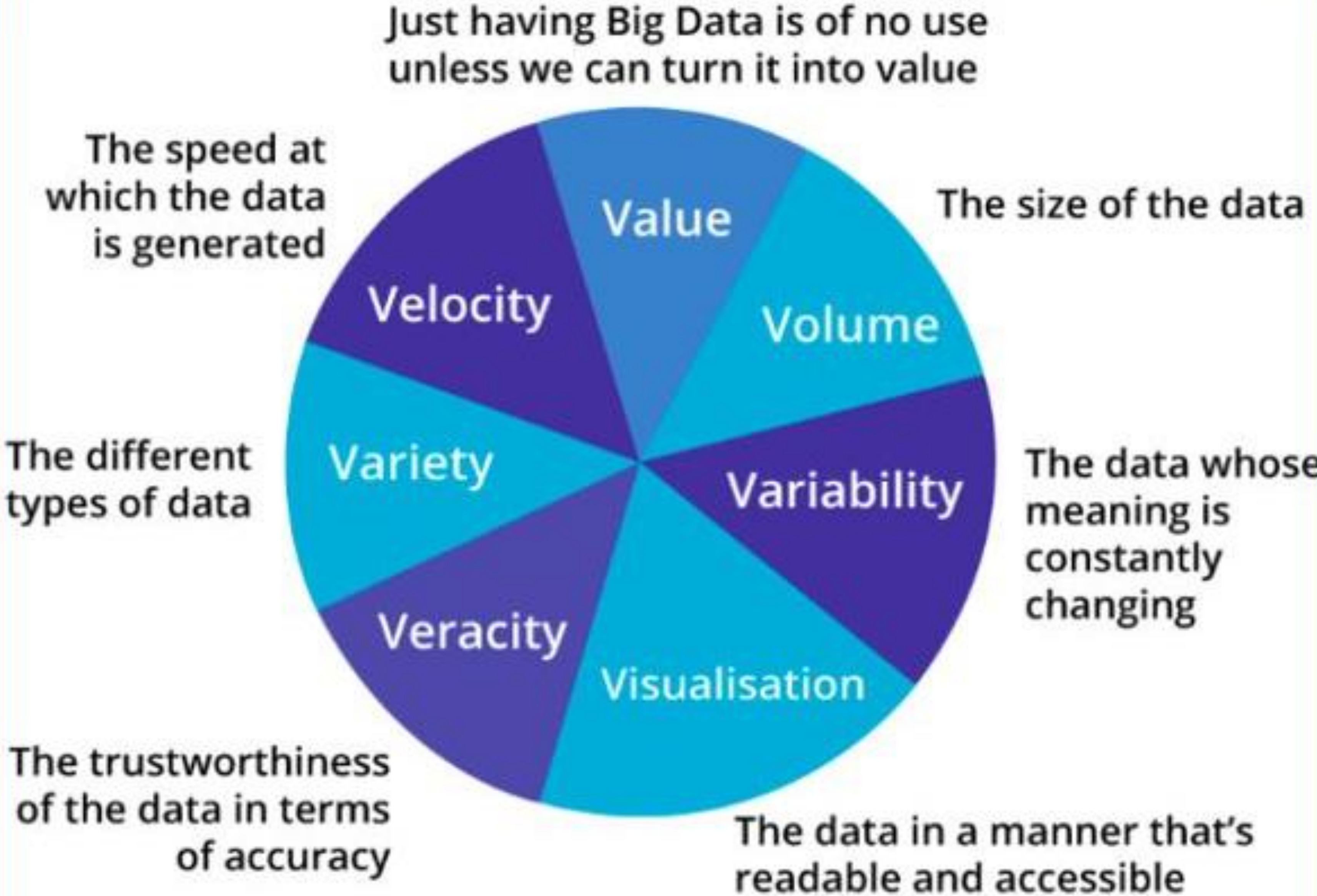
- Many libraries for data analysis, data mining, machine learning available
  - Python/pandas
  - R
  - Matlab
- Suitable for many problems
  - Small data sets
  - experimentation
- As long as they fit on one machine …


Big Data
Analytics

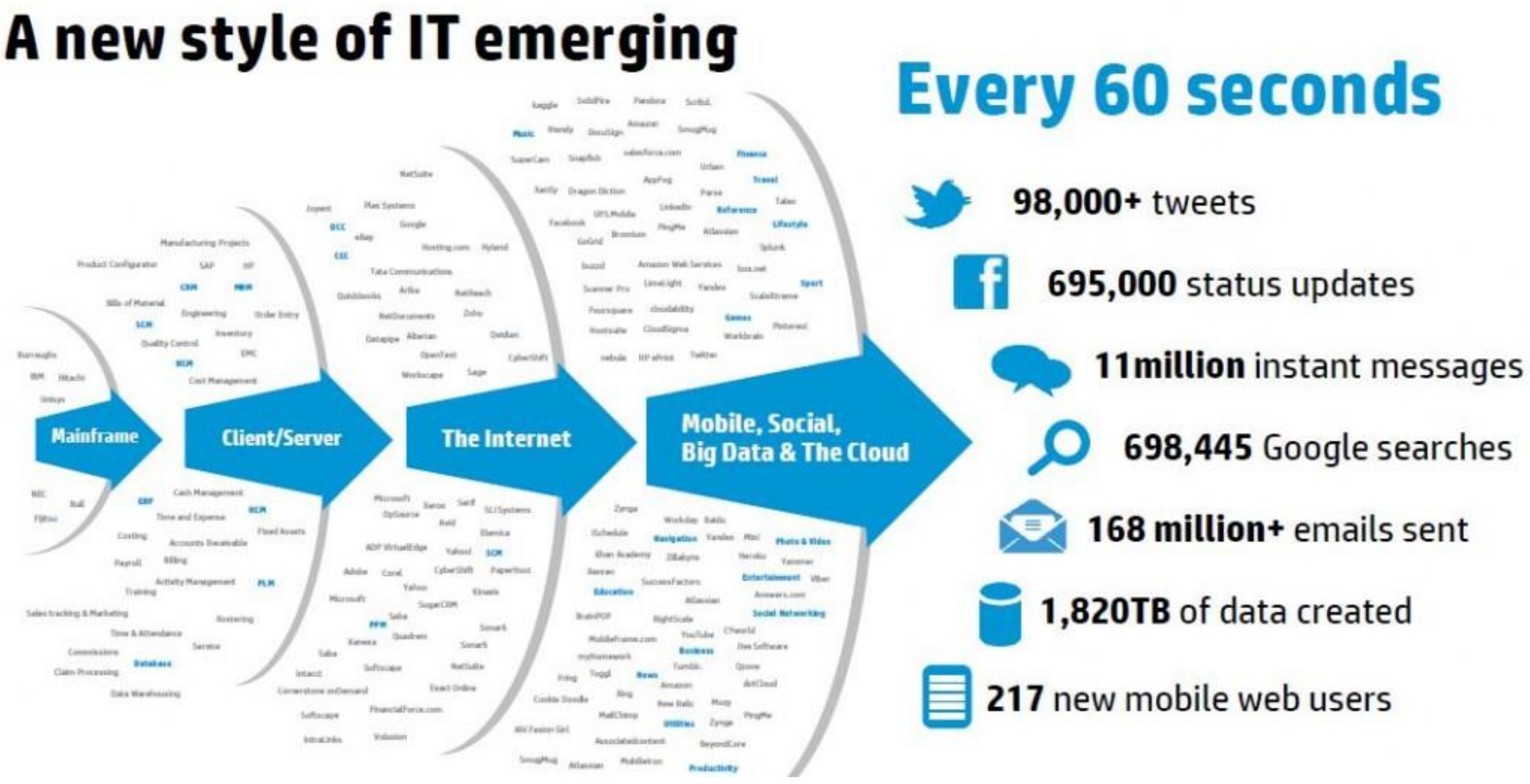
Data Analysis at Scale – Big Data


2019
Loading

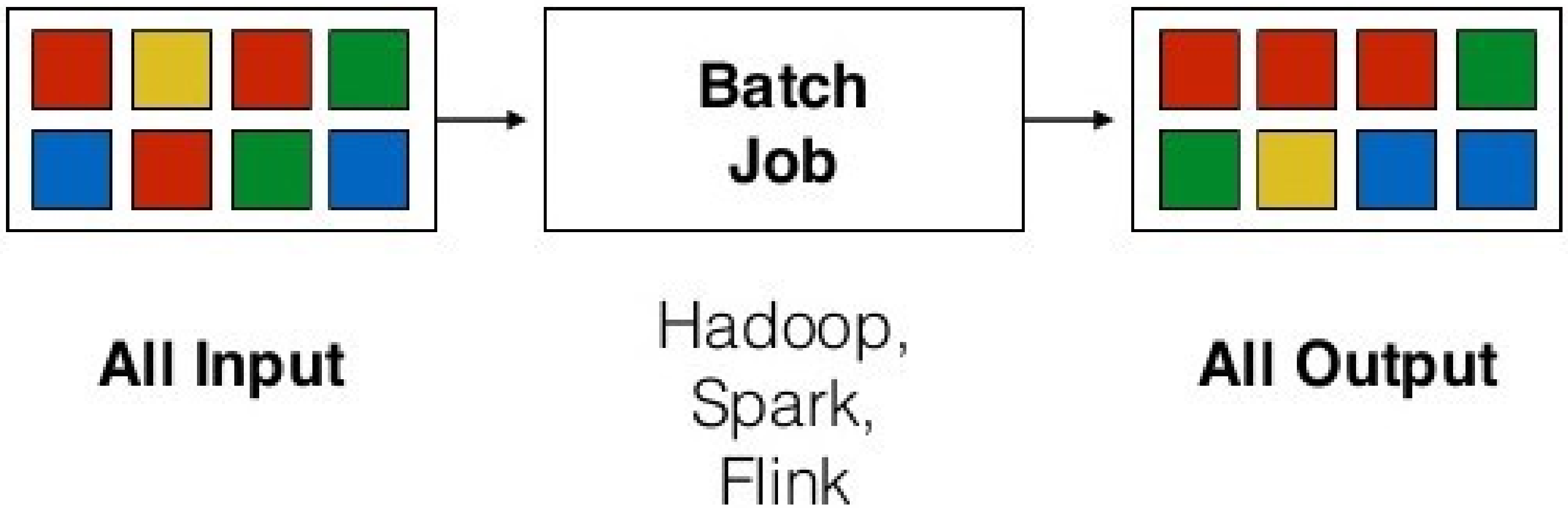
The 7 Vs OF BIG DATA



Big Data


A new style of IT emerging
Mainframe
Client/Server
The Internet
Mobile, Social, Big Data & The Cloud
Every 60 seconds
98,000+ tweets
695,000 status updates
11million instant messages
698,445 Google searches
168 million+ emails sent
1,820TB of data created
217 new mobile web users

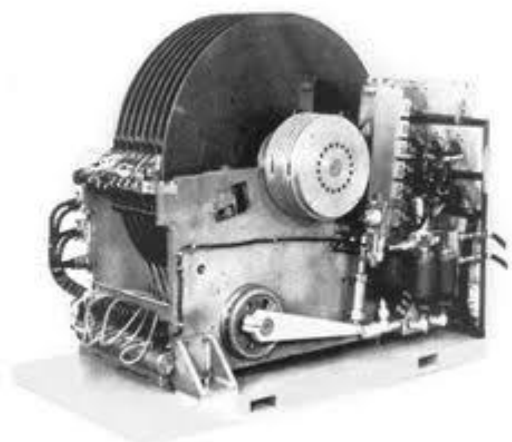
Batch Processing



Batch Processing

- Scheduled periodic processes
  - Daily sales
  - Images uploaded in last hour
- Inputs are big (!)
  - Read only
- Processing transforms inputs to outputs
  - Daily sales analysis by product across stores
  - Sales trend for week/month/year
  - Can have one of more stages
- Can take from minutes to many hours to produce results

Batch Processing



- Big data sets are slow to access sequentially on a single disk, e.g.:
  - 1TB disk at 100MB/s takes 2.5+ hours to read
  - Even slower to write
  - Seek times improving slower than transfer rates
- Parallel access speeds things up
  - Partition 1TB over 100 disks
  - ~2 minutes to read
- Requires replication to provide high reliability
  - Studies show ~8% of hard disks in a data center fail annually

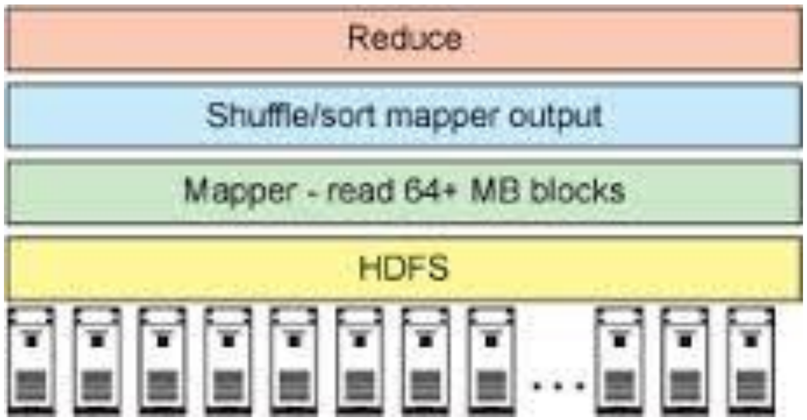
Batch Processing

- Much big data is:
  - Written once, read a lot
  - Often analyzed in its entirety
- This makes it suitable for high-speed streaming reads
  - Minimal seeks times
  - Exploit disk transfer rates
  - Analyze locally to select data that matches a give query
- Enter MapReduce ….

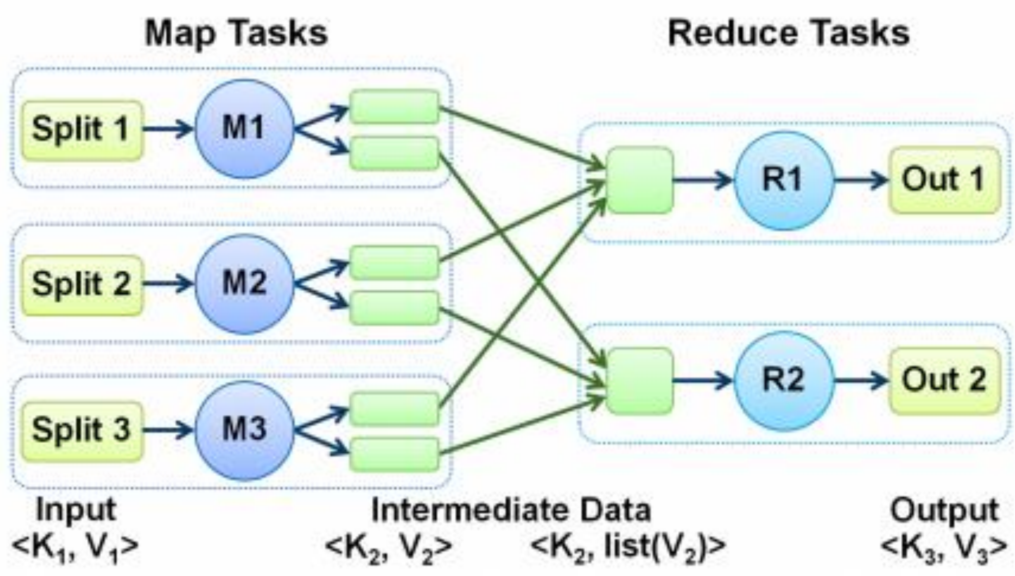
Java 8 Analogy

```
public class UserAverageTest {
  private static List<User> users = Arrays.asList(
      new User(1, "Steve", "Vai", 40),
      new User(4, "Joe", "Smith", 32),
      new User(3, "Steve", "Johnson", 57),
      new User(9, "Mike", "Stevens", 18),
      new User(10, "George", "Armstrong", 24),
      new User(2, "Jim", "Smith", 40),
      new User(8, "Chuck", "Schneider", 34),
      new User(5, "Jorje", "Gonzales", 22),
      new User(6, "Jane", "Michaels", 47),
      new User(7, "Kim", "Berlie", 60)
    );

public static void main(String[] args) {
    double average = users.parallelStream().map(u -> u.age).average().getAsDouble();

    System.out.println("Average User Age: " + average);

  }
}
```

MapReduce (and Hadoop)



- MapReduce: programming model (2004)
- Designed for ***batch processing***
- Hadoop: open source Apache implementation (2006)
- Data stored in Hadoop Distributed File System
  - Distributed across multiple nodes
  - Replicated for fault tolerance

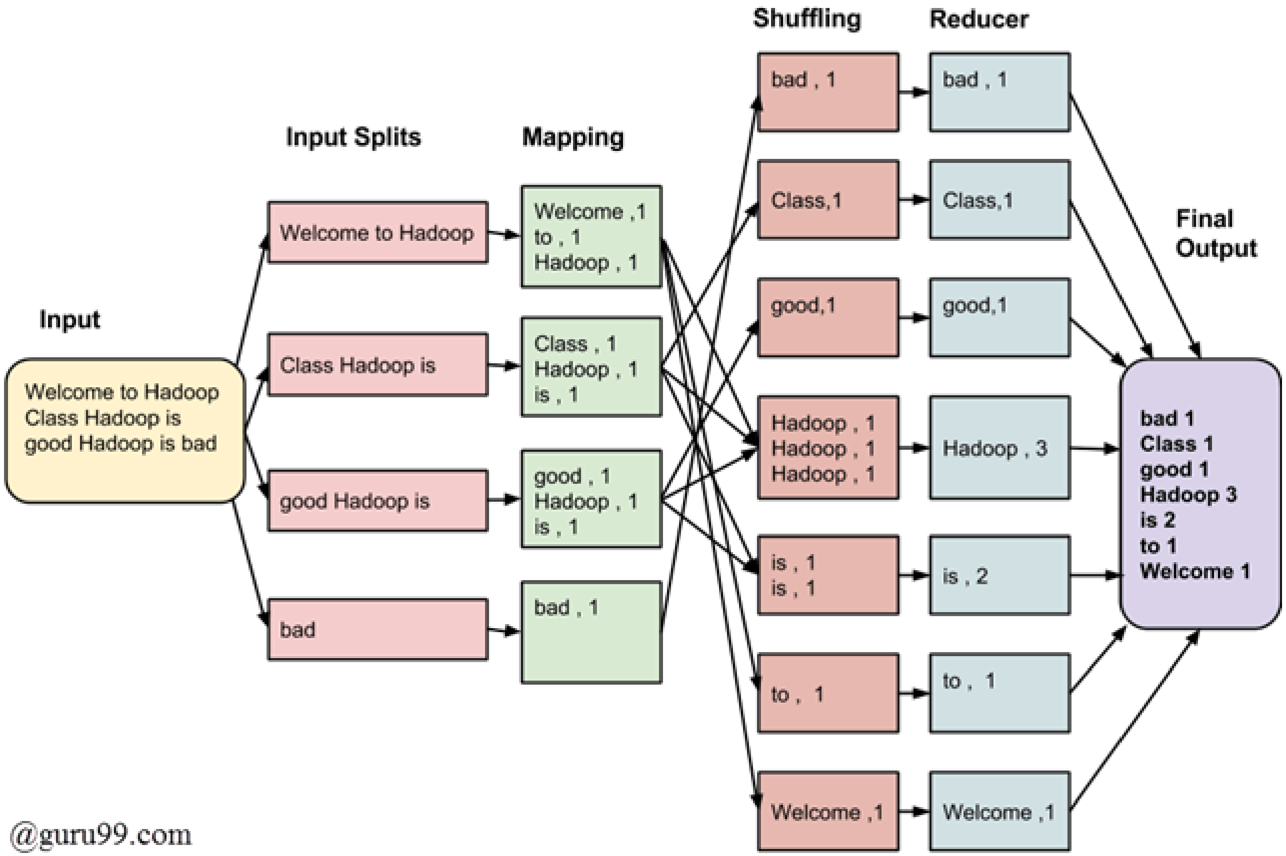


MapReduce: Acyclic Dataflow model

From https://www.cs.duke.edu/starfish/files/hadoop-models.pdf

- Two main phases:
  - Map: process local data to select values relevant for a query
  - Reduce: combine and analyze results emitted from the map phase

Example



https://www.guru99.com/introduction-to-mapreduce.html

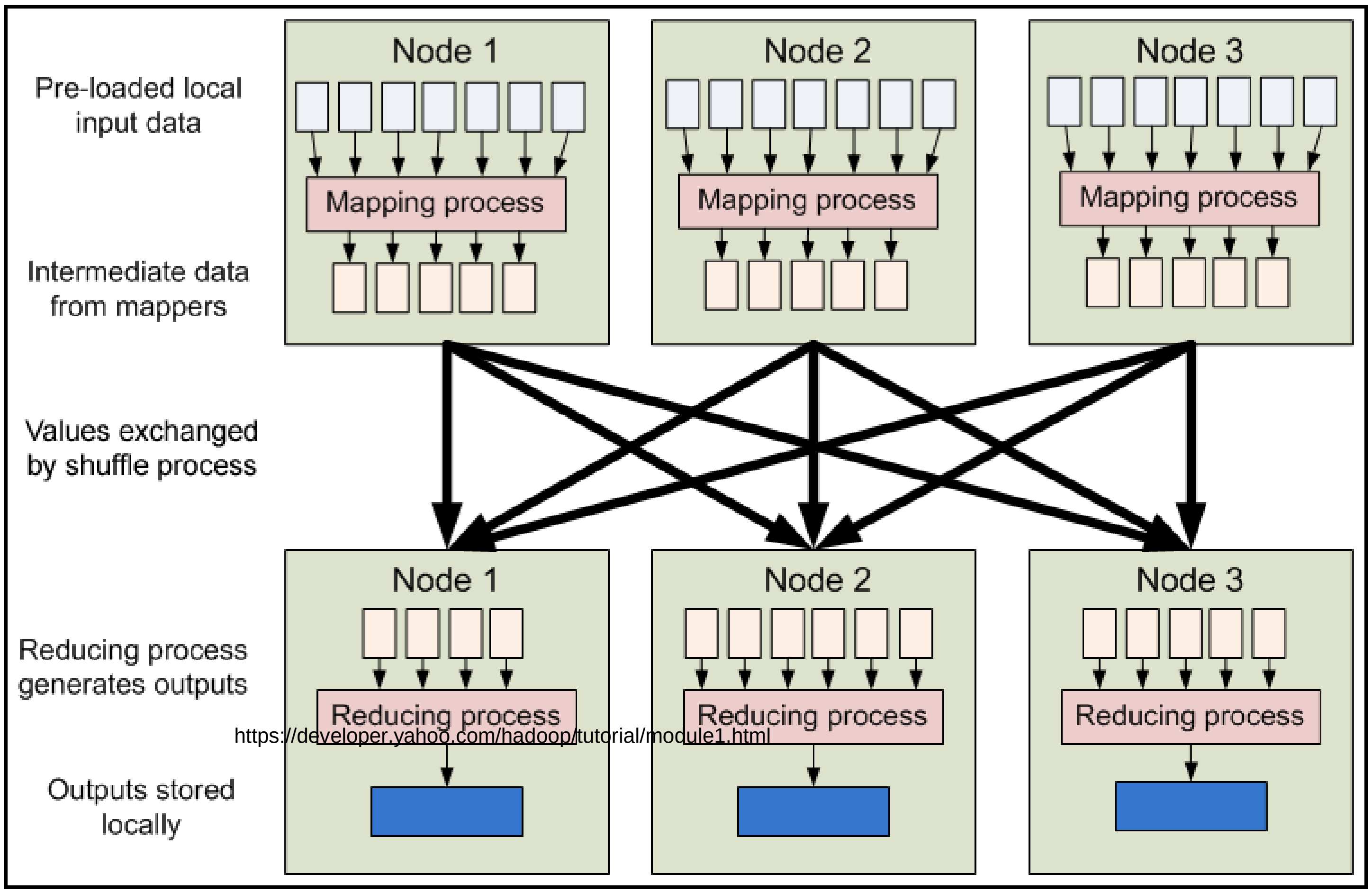
Hadoop Example: Word Count

```
public class WordCount {

  public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
      String line = value.toString();
      StringTokenizer tokenizer = new StringTokenizer(line);
      while (tokenizer.hasMoreTokens()) {
        word.set(tokenizer.nextToken());
        output.collect(word, one);
      }
    }
  }

  public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
      int sum = 0;
      while (values.hasNext()) {
        sum += values.next().get();
      }
      output.collect(key, new IntWritable(sum));
    }
  }
```

Hadoop Example: Word Count

```
public static void main(String[] args) throws Exception {
  JobConf conf = new JobConf(WordCount.class);
  conf.setJobName("wordcount");

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

  conf.setMapperClass(Map.class);
  conf.setCombinerClass(Reduce.class);
  conf.setReducerClass(Reduce.class);

  conf.setInputFormat(TextInputFormat.class);
  conf.setOutputFormat(TextOutputFormat.class);

  FileInputFormat.setInputPaths(conf, new Path(args[0]));
  FileOutputFormat.setOutputPath(conf, new Path(args[1]));

  JobClient.runJob(conf);
}
```

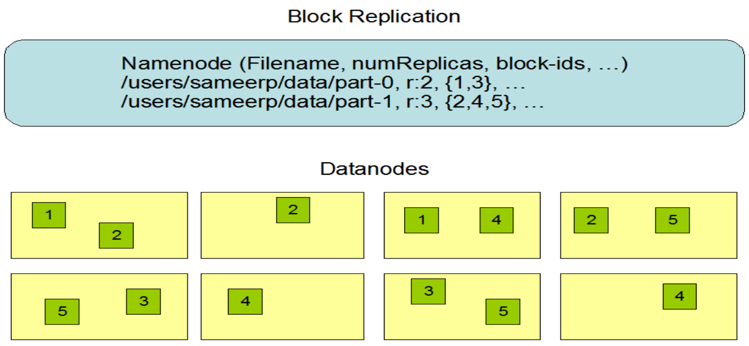
Hadoop – How it works



https://developer.yahoo.com/hadoop/tutorial/module1.html

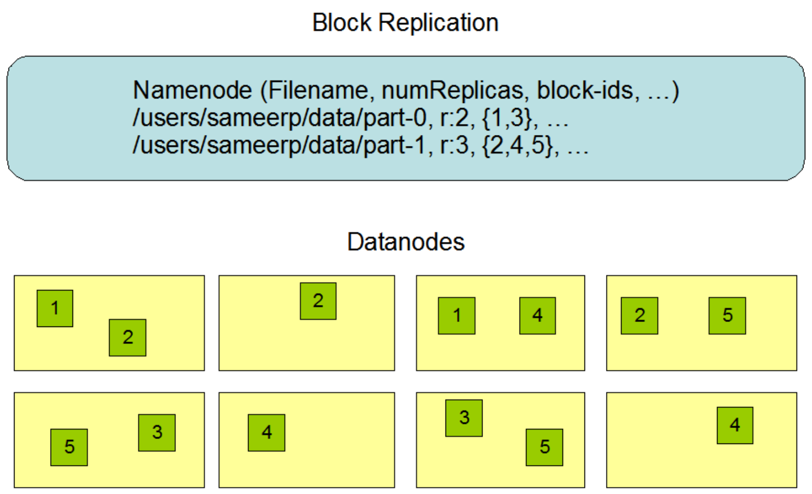
Hadoop Distributed File System

- Distributes and replicates data across many nodes
- Designed to support long streaming reads from disk
  - transfer rate optimized over seek times
  - No local caching of data



http://hadoop.apache.org

Hadoop Distributed File System



- Files broken up into fixed sized blocks (default 64MB)
  - Blocks stored randomly across multiple DataNodes
  - NameNode stores file metadata
  - Balancer utility to distribute blocks across new nodes added to cluster

Hadoop – How it works

- Map Processing:
  - HDFS splits the large input data set into smaller data blocks
  - (64 MB by default) controlled by the property dfs.block.size.
- Data blocks are provided as an input to map tasks.
- Mappers are run on nodes where data resides
  - If possible
  - Locality of access -> faster

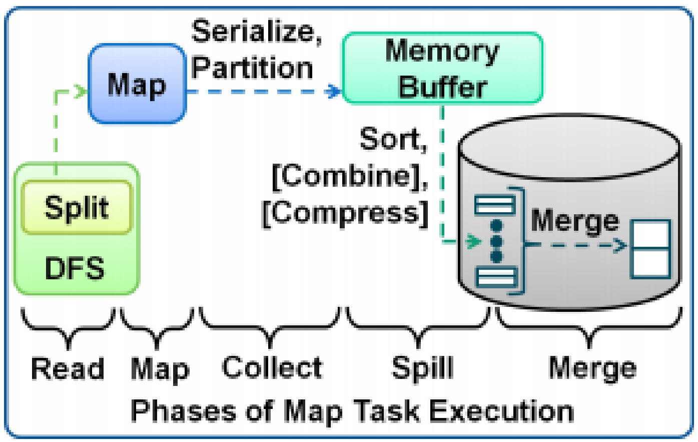
Property name	Type	Default value	Description
mapred.min.split.size	int	1	The smallest valid size in bytes for a file split
mapred.max.split.size[a]	long	Long.MAX_VALUE, that is, 9223372036854775807	The largest valid size in bytes for a file split
dfs.block.size	long	64 MB, that is, 67108864	The size of a block in HDFS in bytes

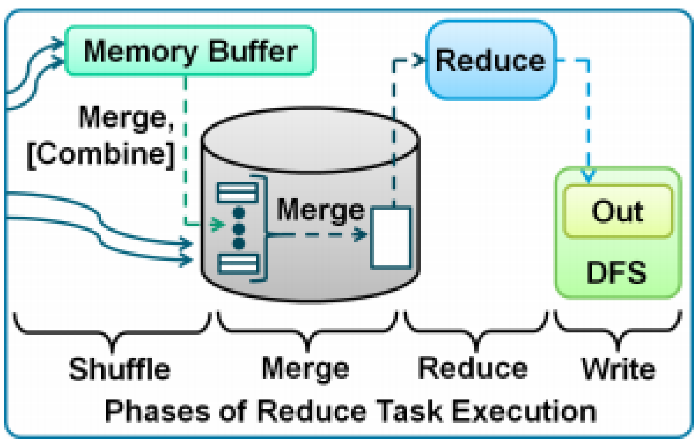
Hadoop – How it works

- block split into key value pairs based on application defined *Input Format* class
  - Split-up the input file(s) into logical InputSplits, each of which assigned to an individual Mapper.
- The map function is invoked for every key value pair in the input.
- Output generated by map function is written to a local circular memory buffer, associated with each map.
  - 100 MB by default and can be controlled by the property **io.sort.mb.....**

Hadoop – Map and Reduce Phases

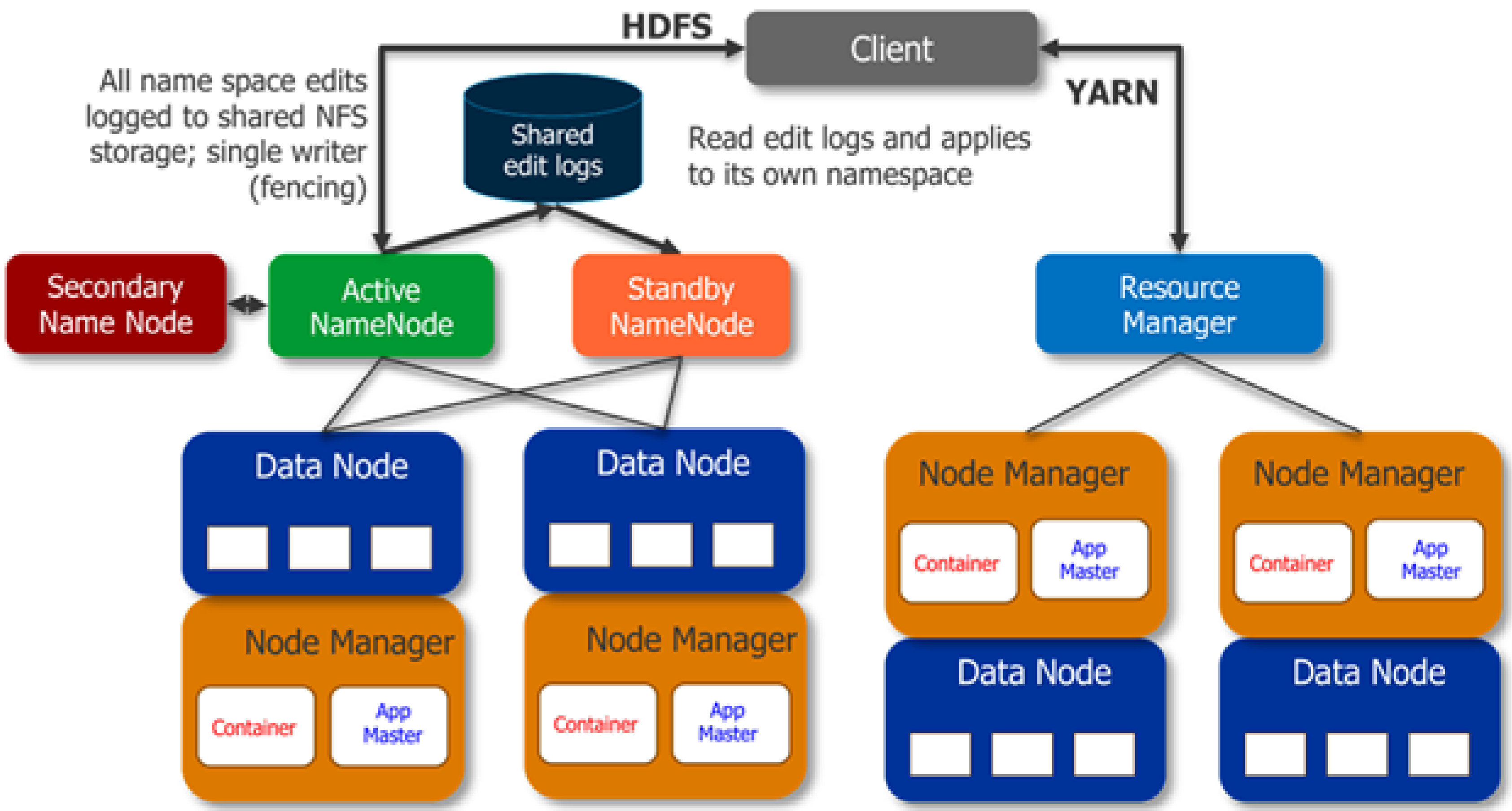
- **Spill**: When the buffer size reaches a threshold size controlled by *io.sort.spill.percen*t (default 0.80 or 80%), a background thread starts to spill the contents to disk. While the spill takes place map continues to write data to the buffer unless it is full. Spills are written in round-robin fashion to the directories specified by *themapred.local.dir* property, in a job-specific subdirectory. A new spill file is created each time the memory buffer reaches to spill threshold.



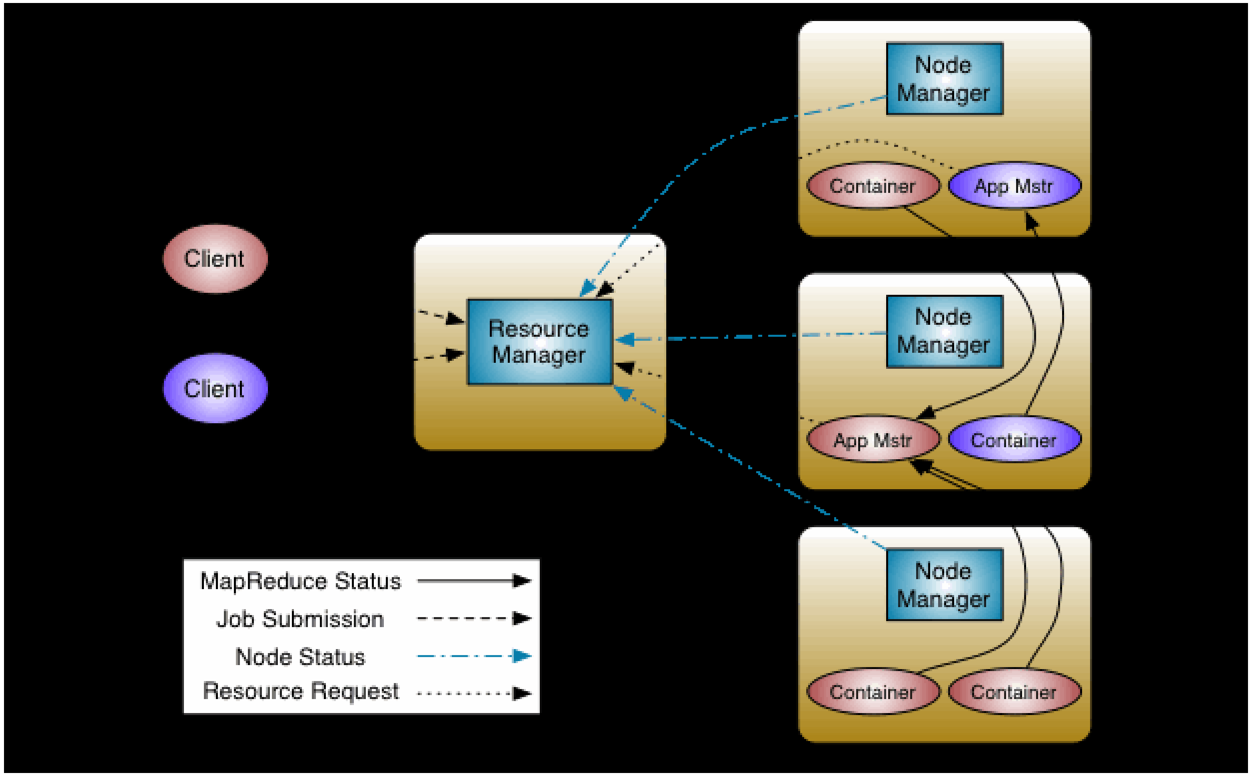


From https://www.cs.duke.edu/starfish/files/hadoop-models.pdf

Hadoop 2.0 Architecture

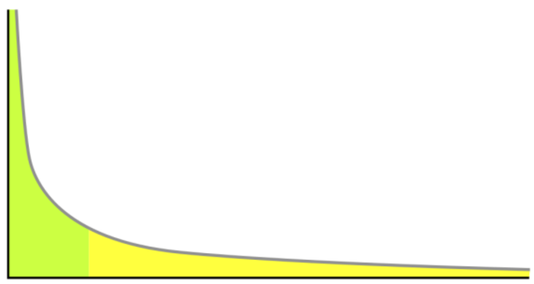


YARN



https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

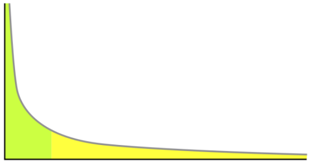
Hadoop Performance Considerations



- Running many jobs simultaneously can dramatically lower performance
  - Contention for disk access
  - HDFS not optimized for seeks
- Execution time for map phase determined by slowest mapper
  - Many jobs exhibit long tail distributions
  - Stragglers
  - Need to carefully design data partitions to attempt to evenly distribute work across mappers

Hadoop Performance Tuning

- Highly configurable behavior
  - Over 200 configuration parameters
  - ~30 can greatly effect performance
- Lets look at some examples ....



Hadoop Performance Tuning

dfs.block.size :

Specifies the size of data blocks in which the input data set is split

mapred.compress.map.output

Specifies whether to compress output of maps.

mapred.map/reduce.tasks.speculative.execution:

When a task (map/reduce) runs slower (due to hardware degradation or software mis-configuration) than expected. The Job Tracker runs another equivalent task as a backup on another node. This is known as speculative execution. The output of the task which finishes first is taken and the other task is killed.

Further Reading: MapReduce

- Apache Pig
  - High level procedural language – Pig Latin – for Hadoop
- Apache Hive
  - Data warehouse functionality for Hadoop
  - Stores metadata in a RDBMS
  - SQL-like HiveQL for data summation/query/analysis
    - Translated to directed acyclic graph of MapReduce jobs

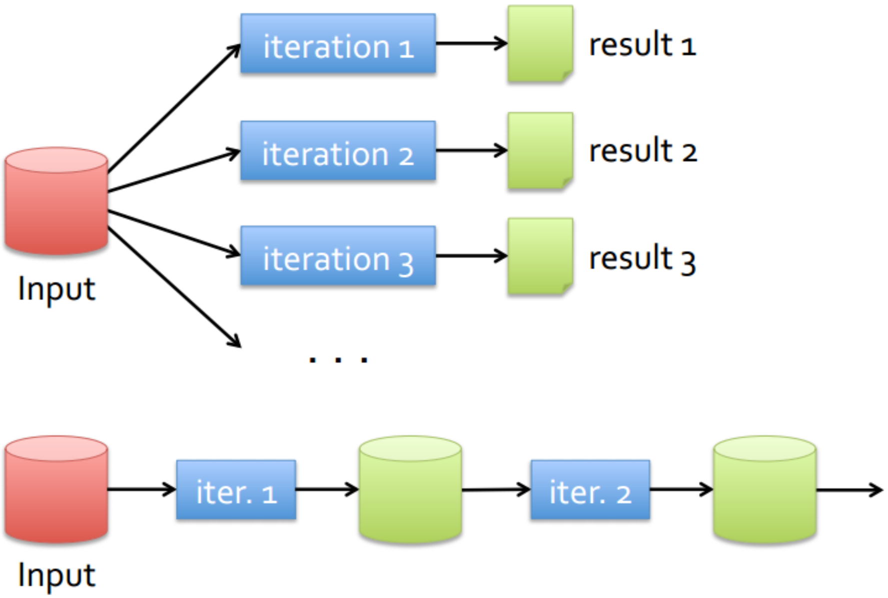
Apache Spark

Apache Spark (https://spark.apache.org/)

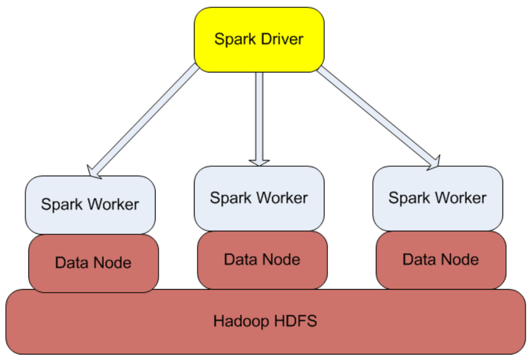
- General purpose cluster computing framework
- Developed in the AMPLab at UC Berkeley.
- Apache top-level project in Feb 2014
- Java, Scala, Python APIs
- Large (e.g. 10-100x) performance gains over Hadoop for certain types of applications
  - Repeatedly reuse/share data across a set of operations
    - Machine learning, graph processing
  - Interactive data mining

Iterative Algorithms

- In Hadoop, large overheads incurred due to read/writing data to stable storage in-between iterations



Spark Basics



- Eliminate overheads of disk accesses by keeping these shared data sets in memory
- Spark provides:
  - Programming model – *Resilient Distributed Datasets (RDDs)* – that support coarse grained operations
  - Fault tolerance model so that data loss can be addressed by recomputation of lost data

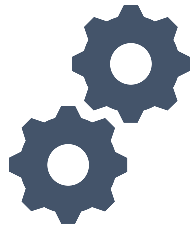
Resilient Distributed Datasets

A data parallel programming model for fault tolerant distributed datasets

Partitioned collections with caching

Transformations (define new RDDs), actions (compute results)

Restricted shared variables (broadcast, accumulators)



Distributes data across slices in a cluster:

Runs 1 task per slice

Spark chooses value automatically (typically 2-4 slices per CPU)

Or set in parallelize function

Simple Example

```
JavaSparkContext sc;

//create a parallel data set
//with 5 slices
List<Integer> data =
Arrays.asList(1, 2, 3, 4, 5);
JavaRDD<Integer> distData =
sc.parallelize(data, 5);

//create a text file RDD with
//a slice per HDFS block
JavaRDD<String> distFile =
sc.textFile("data.txt");
```

RDD Operations

- Transformations
  - Create a new RDD from an existing one
  - map, filter, distinct, union, join, repartition, sortByKey etc
  - Can cache a new RDD for later use
- Actions
  - Return a result to the driver program
  - reduce(func), foreach(func), count, takeSample, etc
- All operations are ‘lazy’
  - Only executed when an action is called to compute a result
  - Supports optimization of Spark execution

```
JavaRDD<String> lines = sc.textFile("data.txt");
// transform …
JavaRDD<Integer> lineLengths = lines.map(s -> s.length());
// cache …
lineLengths.persist();
// compute result
int totalLength = lineLengths.reduce((a, b) -> a + b);
```

Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER	Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.

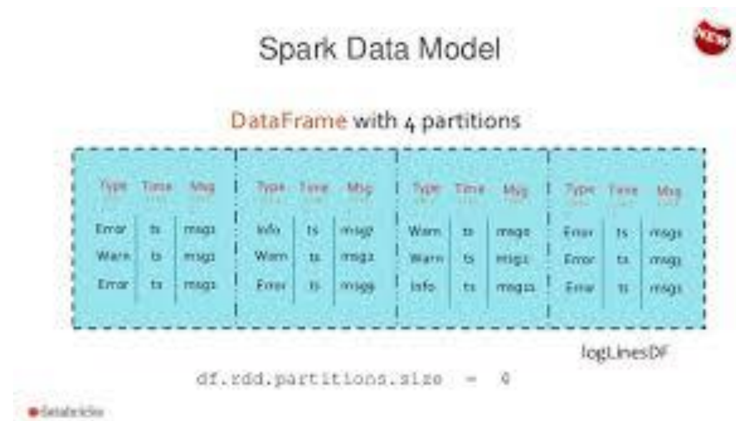
RDD Persistence

- RDDs can be selectively cached between operations
- Various options:
    - Leave in memory ➔ fast
    - Serialize to save space
    - Spill to disk only if the data is expensive to compute
    - Fault tolerance allows immediate partition recovery for computation
        - All data is fault tolerant in that it can always be recomputed, but with latency costs

DataFrames

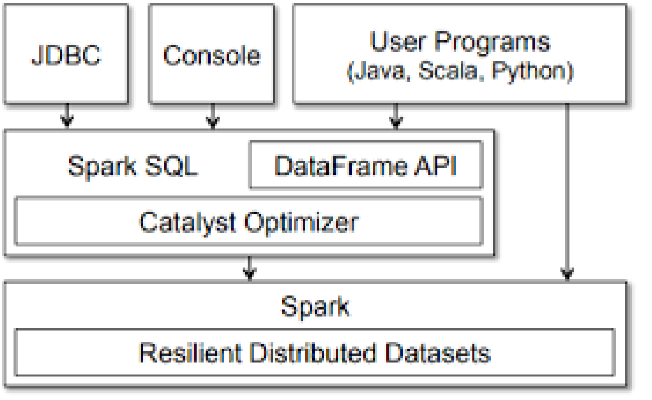
- Built on RDDs
- Column-based data structures
- Partitioned
- Untyped
- Column-based queries

df.filter(col(“Type”).like(“Error”)).show();



DataSets

- Add type safety
- Defined by a Java class
- SQL based API




```
Dataset<Person> people =
spark.read().parquet("...").as(Encoders.bean(Person.class));

Dataset<Person> oldPeople = spark.sql("SELECT * FROM
people WHERE age>65");
```

Example

```
// To create Dataset<Row> using SparkSession
   Dataset<Row> people = spark.read().parquet("...");
   Dataset<Row> department = spark.read().parquet("...");

   people.filter(people.col("age").gt(30))
     .join(department, people.col("deptId").equalTo(department.col("id")))
     .groupBy(department.col("name"), people.col("gender"))
     .agg(avg(people.col("salary")), max(people.col("age")));
```

Spark Ecosystem

Spark Examples

RDD Guide

https://spark.apache.org/docs/2.4.4/rdd-programming-guide.html#resilient-distributed-datasets-rdds

Code Examples

http://spark.apache.org/examples.html

Summary

- Much data is written once and processed many times
- Massive data sets need to be batch processed
- Hadoop implements MapReduce model
- Spark provides powerful analytics and scales across memory and disk