CS 444: Big Data Systems

Chapter 5. Big Data Computing and Processing

Chase Wu

New Jersey Institute of Technology

Some of the slides were provided through the courtesy of Dr. Ching-Yung Lin at Columbia University

Remind -- Apache Hadoop

The Apache™ Hadoop® project develops open-source software for reliable, scalable, distributed computing.

The project includes these modules:

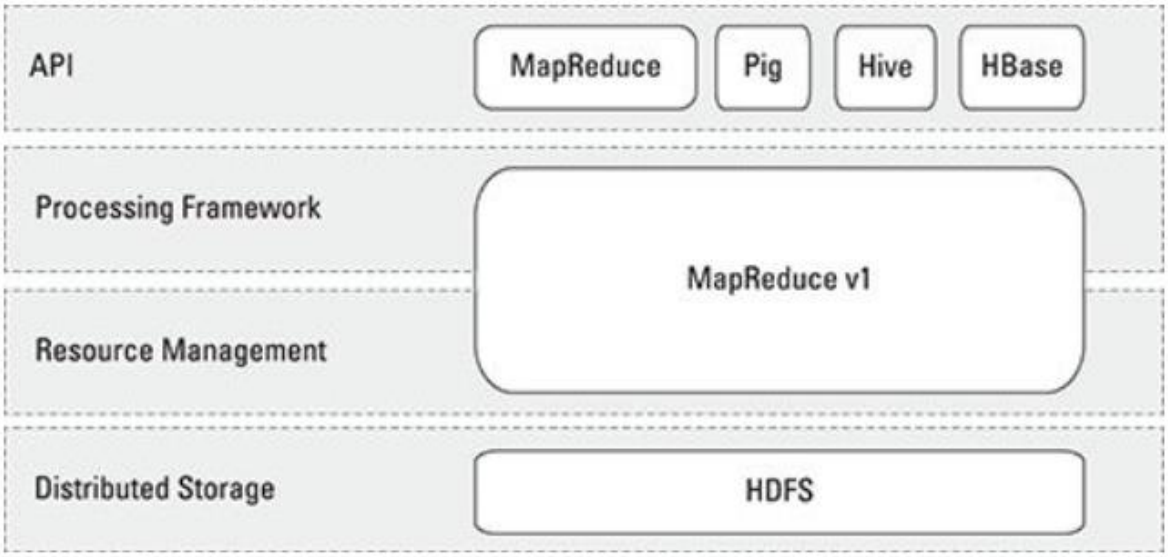
- **Hadoop Common**: The common utilities that support the other Hadoop modules.
- **Hadoop Distributed File System (HDFS™)**: A distributed file system that provides high-throughput access to application data.
- **Hadoop YARN**: A framework for job scheduling and cluster resource management.
- **Hadoop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

http://hadoop.apache.org

Remind -- Hadoop-related Apache Projects

- Ambari™: A web-based tool for provisioning, managing, and monitoring Hadoop clusters.It also provides a dashboard for viewing cluster health and ability to view MapReduce, Pig and Hive applications visually.
- Avro™: A data serialization system.
- Cassandra™: A scalable multi-master database with no single points of failure.
- Chukwa™: A data collection system for managing large distributed systems.
- HBase™: A scalable, distributed database that supports structured data storage for large tables.
- Hive™: A data warehouse infrastructure that provides data summarization and ad hoc querying.
- Mahout™: A Scalable machine learning and data mining library.
- Pig™: A high-level data-flow language and execution framework for parallel computation.
- Spark™: A fast and general compute engine for Hadoop data. Spark provides a simple and expressive programming model that supports a wide range of applications, including ETL, machine learning, stream processing, and graph computation.
- Tez™: A generalized data-flow programming framework, built on Hadoop YARN, which provides a powerful and flexible engine to execute an arbitrary DAG of tasks to process data for both batch and interactive use-cases.
- ZooKeeper™: A high-performance coordination service for distributed applications.

Four distinctive layers of Hadoop 1



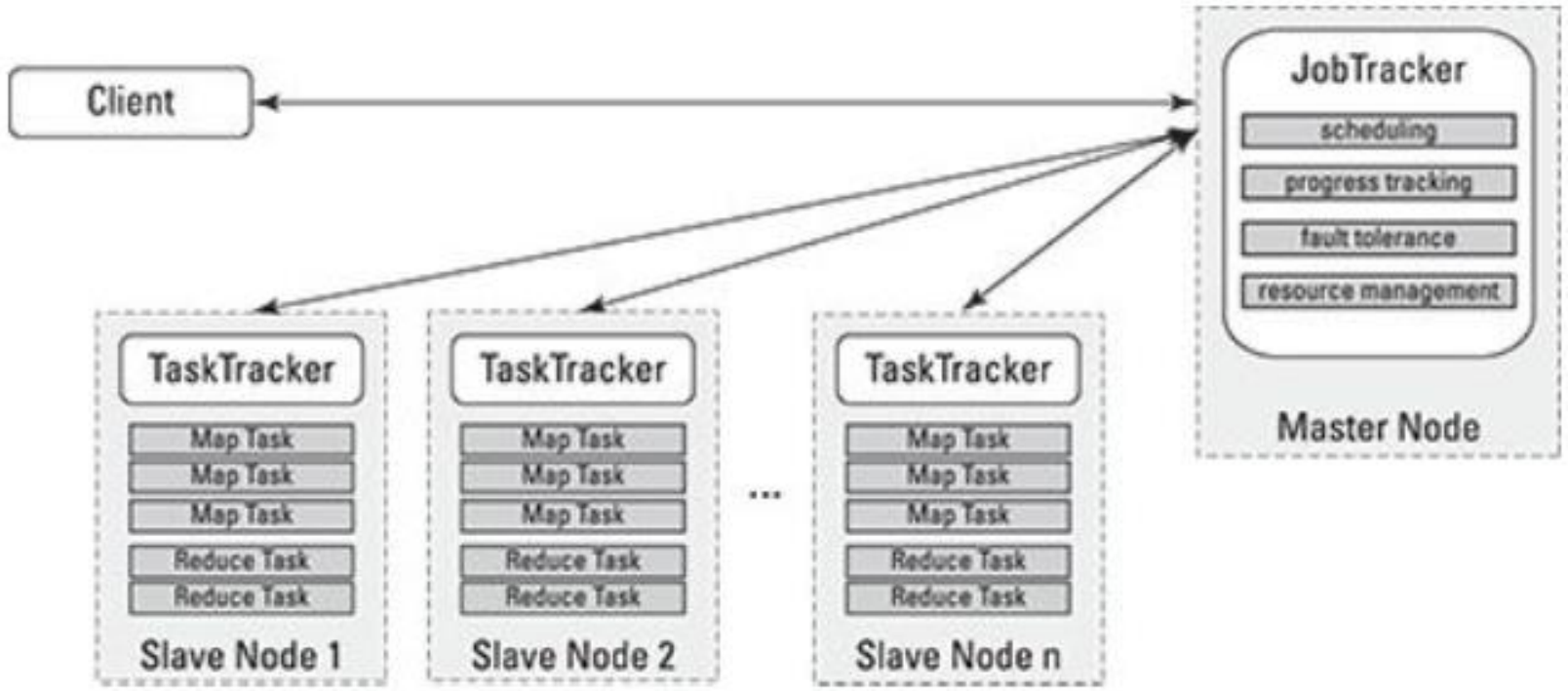
Distributed storage: The Hadoop Distributed File System (HDFS) is the storage layer where the data, interim results, and final result sets are stored.

Resource management: In addition to disk space, all slave nodes in the Hadoop cluster have CPU cycles, RAM, and network bandwidth. A system such as Hadoop needs to be able to parcel out these resources so that multiple applications and users can share the cluster in predictable and tunable ways. This job is done by the JobTracker daemon.

Processing framework: The MapReduce process flow defines the execution of all applications in Hadoop 1. As we saw in Chapter 6, this begins with the map phase; continues with aggregation with shuffle, sort, or merge; and ends with the reduce phase. In Hadoop 1, this is also managed by the JobTracker daemon, with local execution being managed by TaskTracker daemons running on the slave nodes.

Application Programming Interface (API): Applications developed for Hadoop 1 needed to be coded using the MapReduce API. In Hadoop 1, the Hive and Pig projects provide programmers with easier interfaces for writing Hadoop applications, and underneath the hood, their code compiles down to MapReduce.

Haoop 1 execution



1. The client application submits an application request to the JobTracker.
2. The JobTracker determines how many processing resources are needed to execute the entire application.
3. The JobTracker looks at the state of the slave nodes and queues all the map tasks and reduce tasks for execution.
4. As processing slots become available on the slave nodes, map tasks are deployed to the slave nodes. Map tasks are assigned to nodes where the same data is stored.
5. The TaskTracker monitors task progress. If failure, the task is restarted on the next available slot.
6. After the map tasks are finished, reduce tasks process the interim results sets from the map tasks.
7. The result set is returned to the client application.

Limitation of Hadoop 1

- MapReduce is a successful batch-oriented programming model.

- A glass ceiling in terms of wider use:
  - Exclusive tie to MapReduce, which means it could be used only for batch-style workloads and for general-purpose analysis.

- Triggered demands for additional processing modes:
  - Stream data processing (Storm)
  - Message passing (MPI)
  - Graph analysis

➔ Demand is growing for real-time and ad-hoc analysis
➔ Analysts ask many smaller questions against subsets of data and need a near-instant response.
➔ Some analysts are more used to SQL & Relational databases

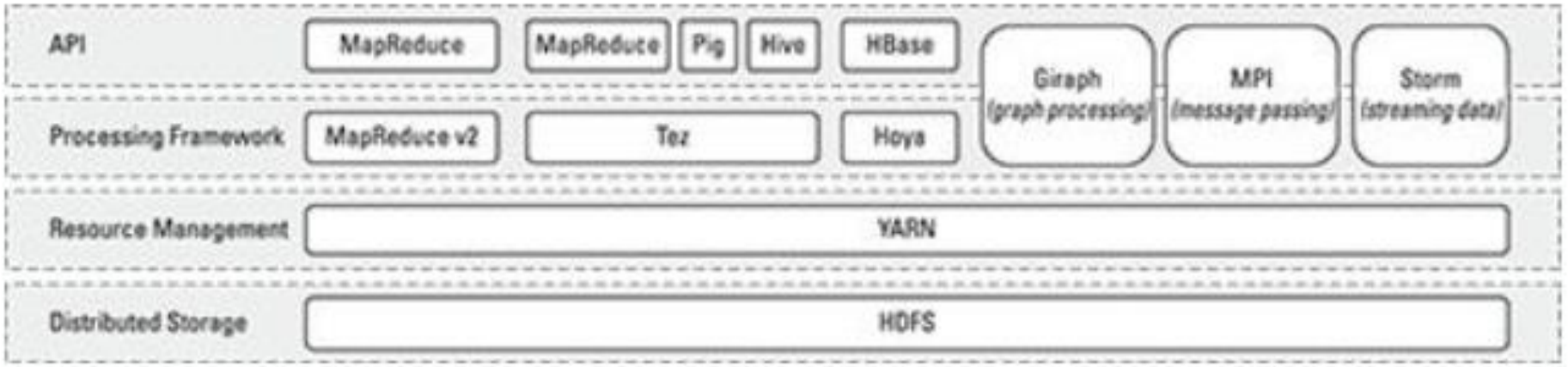
YARN was created to move beyond the limitation of a Hadoop 1 / MapReduce world.

YARN: Resource Management to Support Parallel Computing

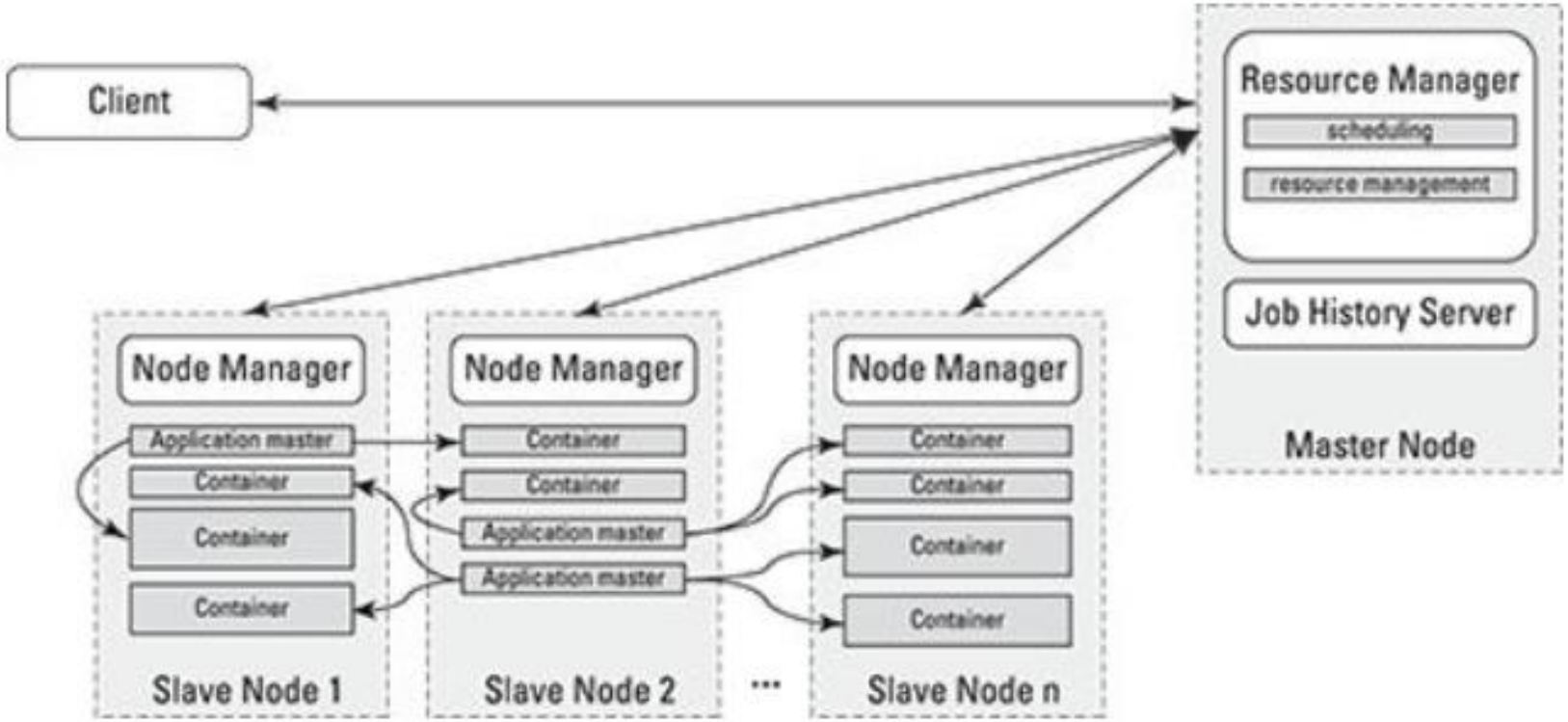
- YARN – Yet Another Resource Negotiator
  - A resource management tool that enables the other parallel processing frameworks to run on Hadoop.
  - A general-purpose resource management facility that can schedule and assign CPU cycles and memory (and in the future, other resources, such as network bandwidth) from the Hadoop cluster to waiting applications.

➔ Starting from Hadoop 2, YARN has converted Hadoop from simply a batch processing engine into a platform for many different modes of data processing
  - From traditional batch to interactive queries to streaming analysis.

Hadoop 2 Data Processing Architecture

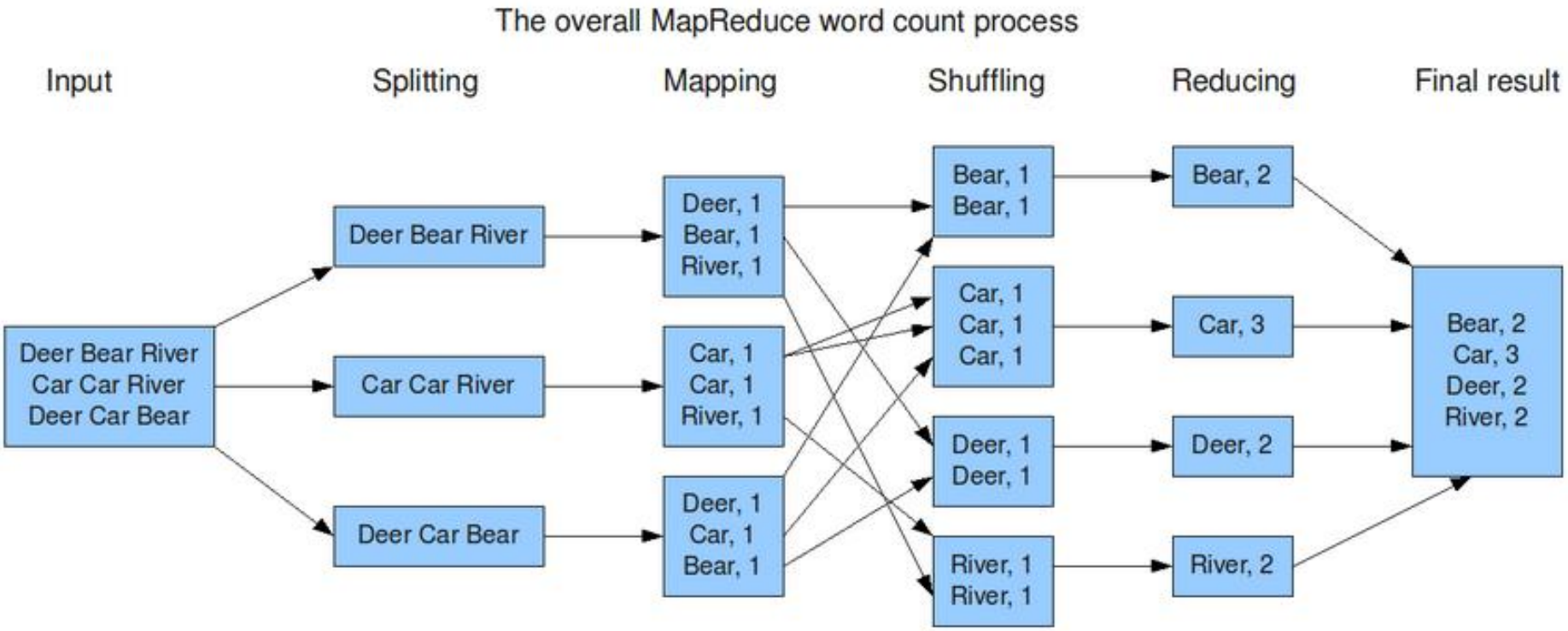


YARN's application execution



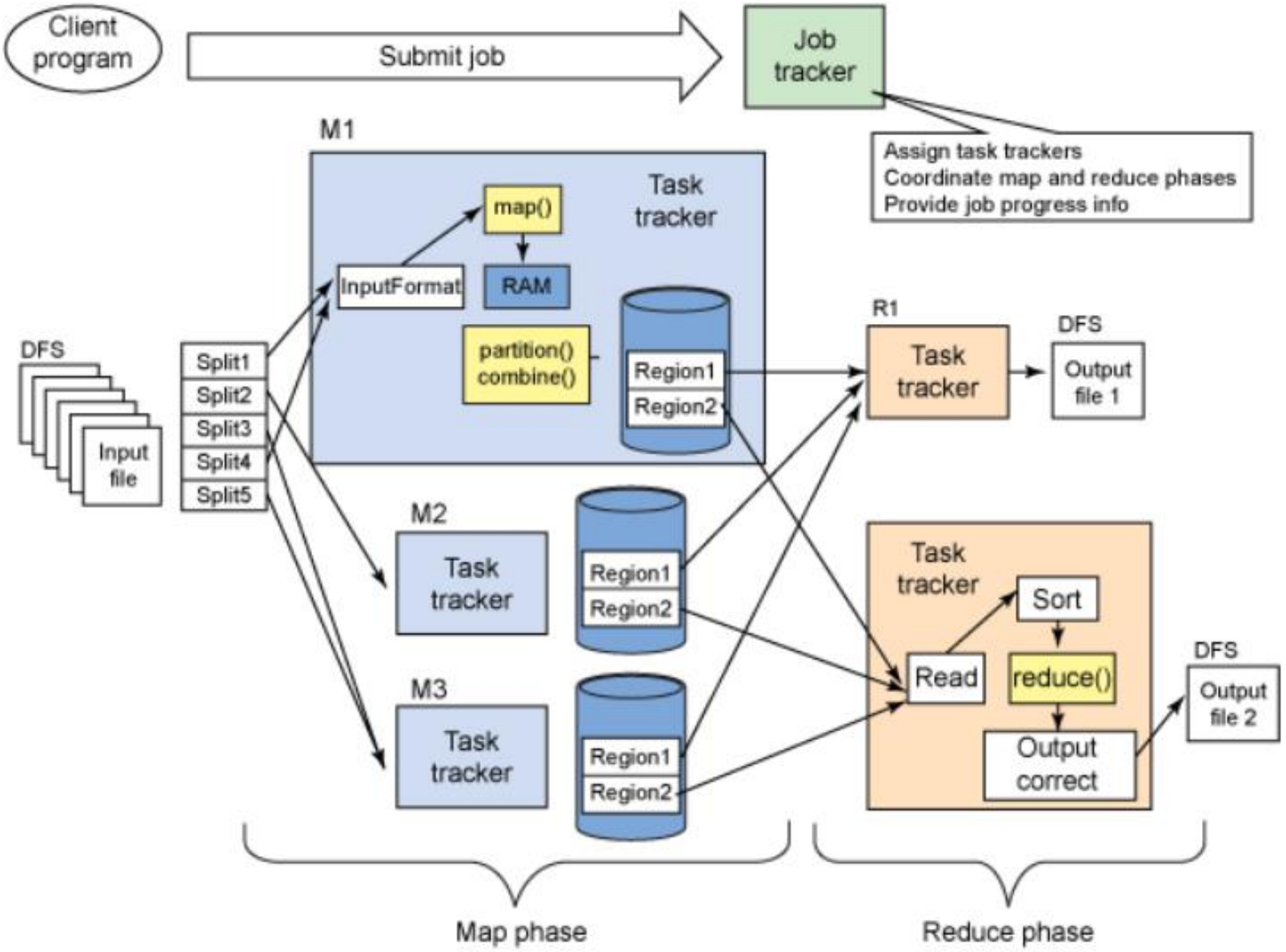
- The client submits an application to the Resource Manager.
- The Resource Manager asks a Node Manager to create an Application Master Instance (AMI) and starts up.
- Application Master initializes itself and register with the Resource Manager
- Application Master figures out how many resources are needed to execute the application.
- Application Master then requests the necessary resources from the Resource Manager. It sends heartbeat message to the Resource Manager throughout its lifetime.
- The Resource Manager accepts the request and queue up.
- As the requested resources become available on the slave nodes, the Resource Manager grants the Application Master leases for containers on specific slave nodes.
- …. ➔ only need to decide on how much memory tasks can have.

MapReduce WordCount revisit



http://www.alex-hanna.com

MapReduce Data Flow (Hadoop 1)



http://www.ibm.com/developerworks/cloud/library/cl-openstack-deployhadoop/

Spark

Fast, Interactive, Language-Integrated Cluster Computing

Download source release:

www.spark-project.org

Spark Goals

Extend the MapReduce model to better support two common classes of analytics applications:

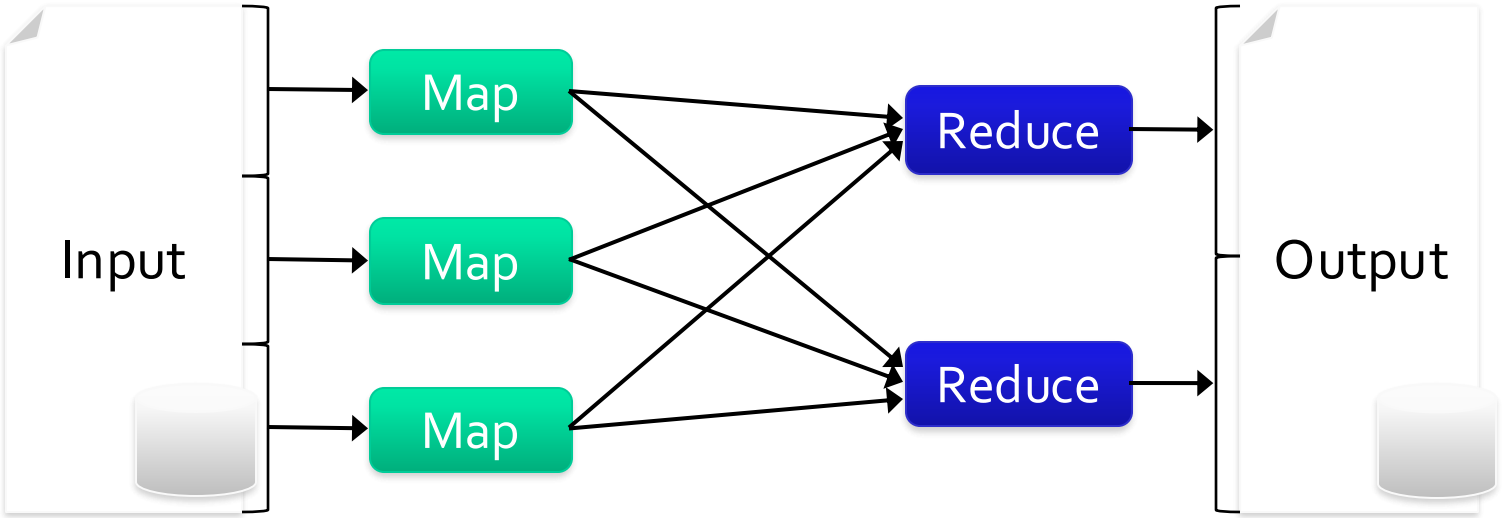
- **Iterative** algorithms (machine learning, graphs)
- **Interactive** data mining (user query)

Enhance programmability:

- Integrate into Scala programming language
- Allow interactive use from Scala interpreter

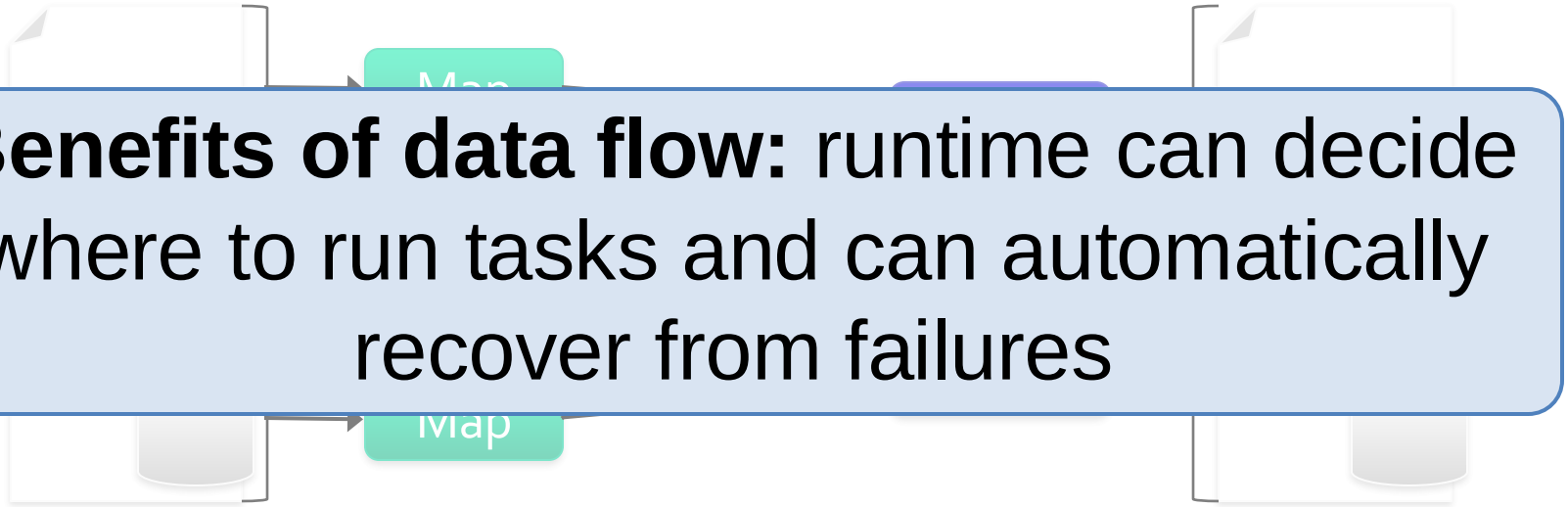
Motivation

Most current cluster programming models are based on *acyclic data flow* from stable storage to stable storage



Motivation

Most current cluster programming models are based on *acyclic data flow* from stable storage to stable storage



Benefits of data flow: runtime can decide where to run tasks and can automatically recover from failures

Motivation

Acyclic data flow is inefficient for applications that repeatedly reuse a *working set* of data:

- **Iterative** algorithms (machine learning, graphs)
- **Interactive** data mining tools (R, Excel, Python)

With current frameworks, applications must reload data from stable storage on each query, which is time consuming!

Solution:
Resilient Distributed Datasets (RDDs)

- Allow apps to keep working sets in memory for efficient reuse
- Retain the attractive properties of MapReduce
  - Fault tolerance, data locality, scalability
- Support a wide range of applications

Programming Model

Two stages: **transformations** followed by **actions**

- Core structure: Resilient distributed datasets (RDDs)
  - Immutable, partitioned collections of objects
  - Created through parallel *transformations* (map, filter, groupBy, join, …) on data in stable storage
  - Can be *cached* for efficient reuse
- Perform multiple various *Actions* on RDDs
  - Count, reduce, collect, save, …

Note that

- Before Spark 2.0, the main programming interface of Spark was the Resilient Distributed Dataset (RDD)
- After Spark 2.0, RDDs are replaced by Dataset
  - Strongly-typed like a RDD, but with richer optimizations
  - The RDD interface is still supported

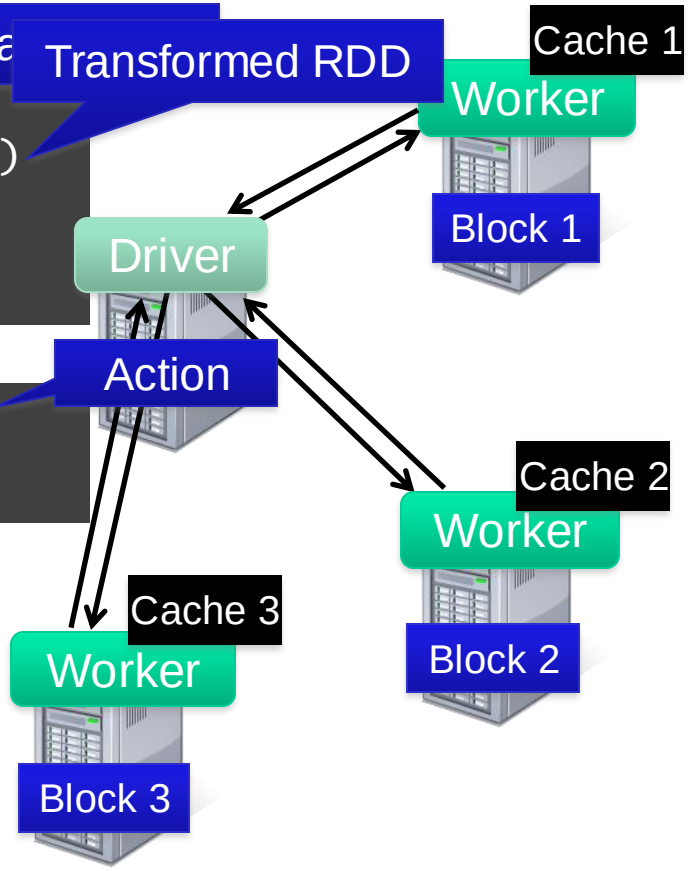
Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

Scala

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()
```

```
cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
```



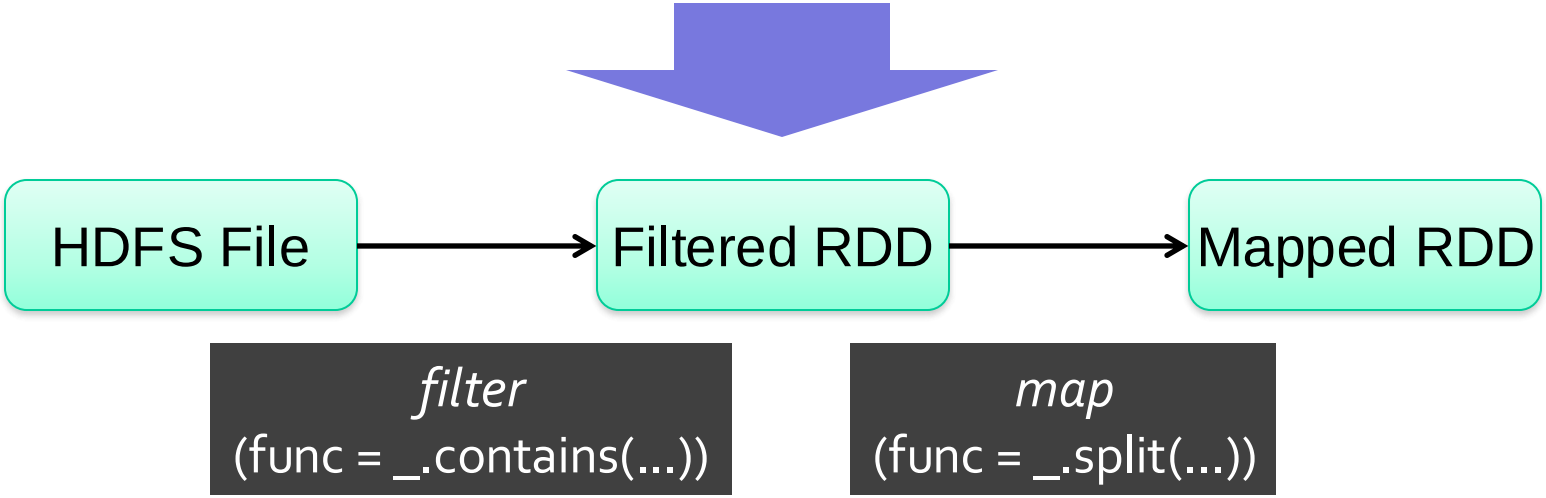
Result: scaled to 1 TB data in 5-7 sec
(vs 170 sec for on-disk data)

RDD Fault Tolerance

RDDs maintain *lineage* information that can be used to reconstruct lost partitions

Ex:

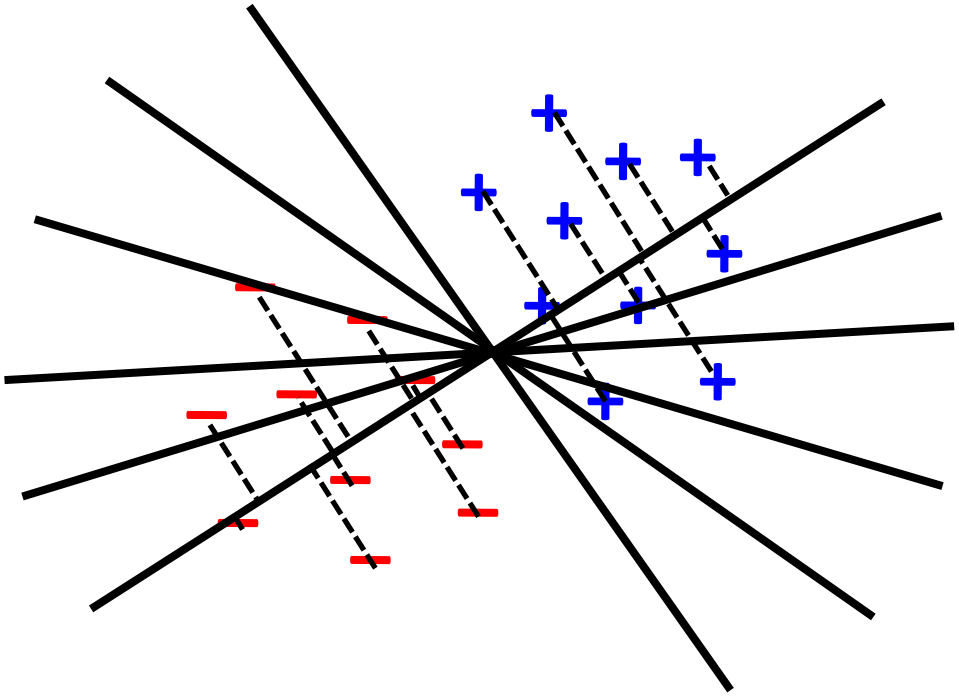
```
messages = textFile(...).filter(_.startsWith("ERROR"))
                        .map(_.split('\t')(2))
```



Example: Logistic Regression

Goal: find the best line separating two sets of points

The found line can be used to classify new points.



Example: Logistic Regression

Scala

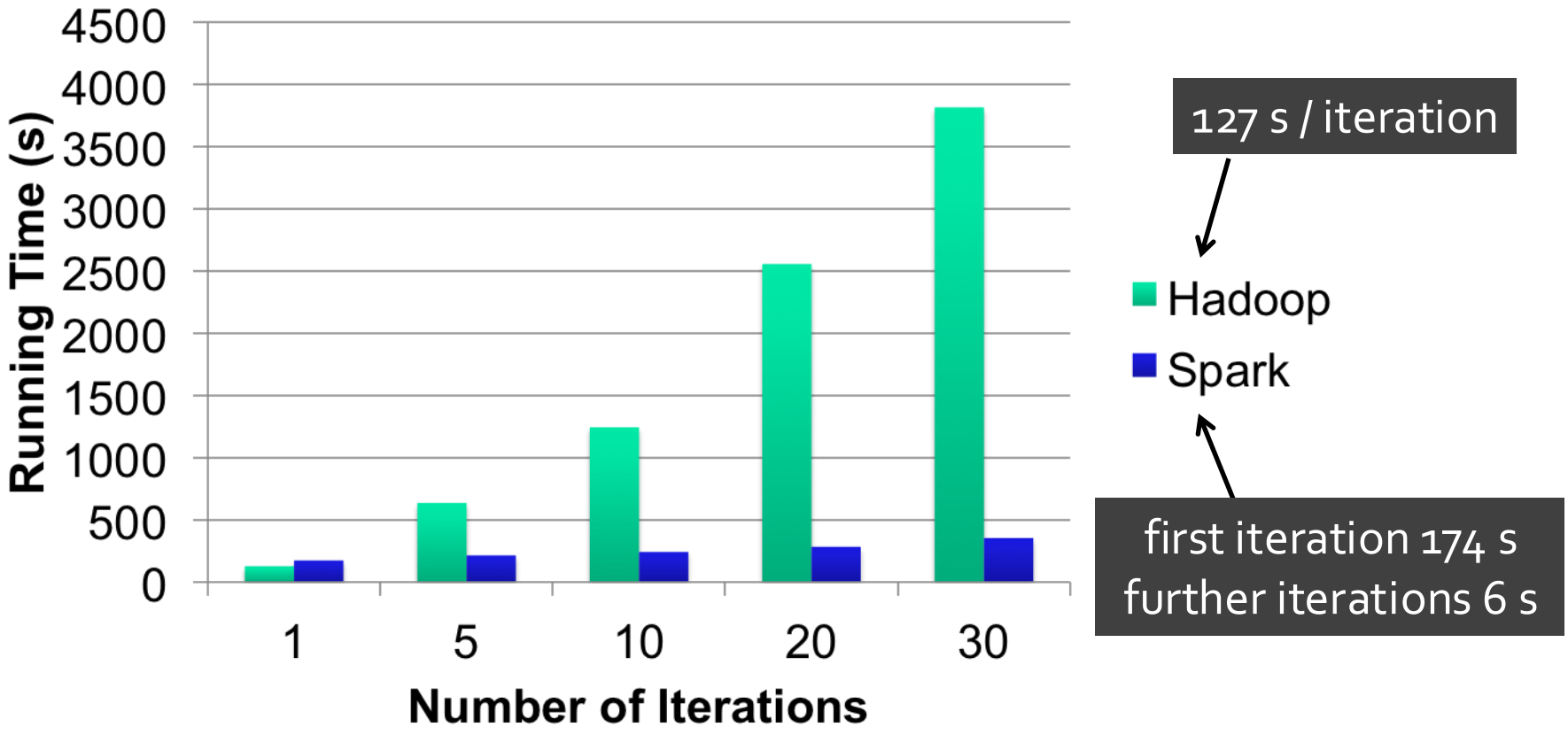
```
val data = spark.textFile(...).map(readPoint).cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = data.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}

println("Final w: " + w)
```

Logistic Regression Performance



Spark Applications

In-memory data mining on Hive data (Conviva)

Predictive analytics (Quantifind)

City traffic prediction (Mobile Millennium)

Twitter spam classification (Monarch)

Collaborative filtering via matrix factorization

…

Data Processing Frameworks Built on Spark

Pregel on Spark (Bagel)

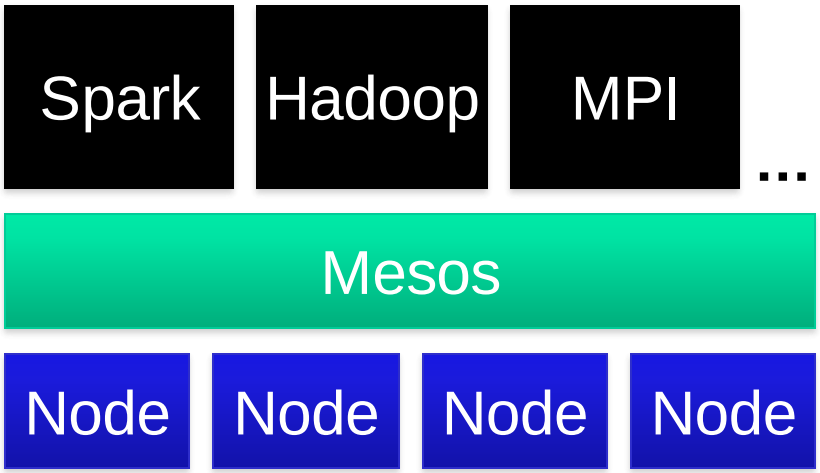
- Google message passing model for graph computation
- 200 lines of code

Hive on Spark (Shark)

- 3000 lines of code
- Compatible with Apache Hive
- ML operators in Scala

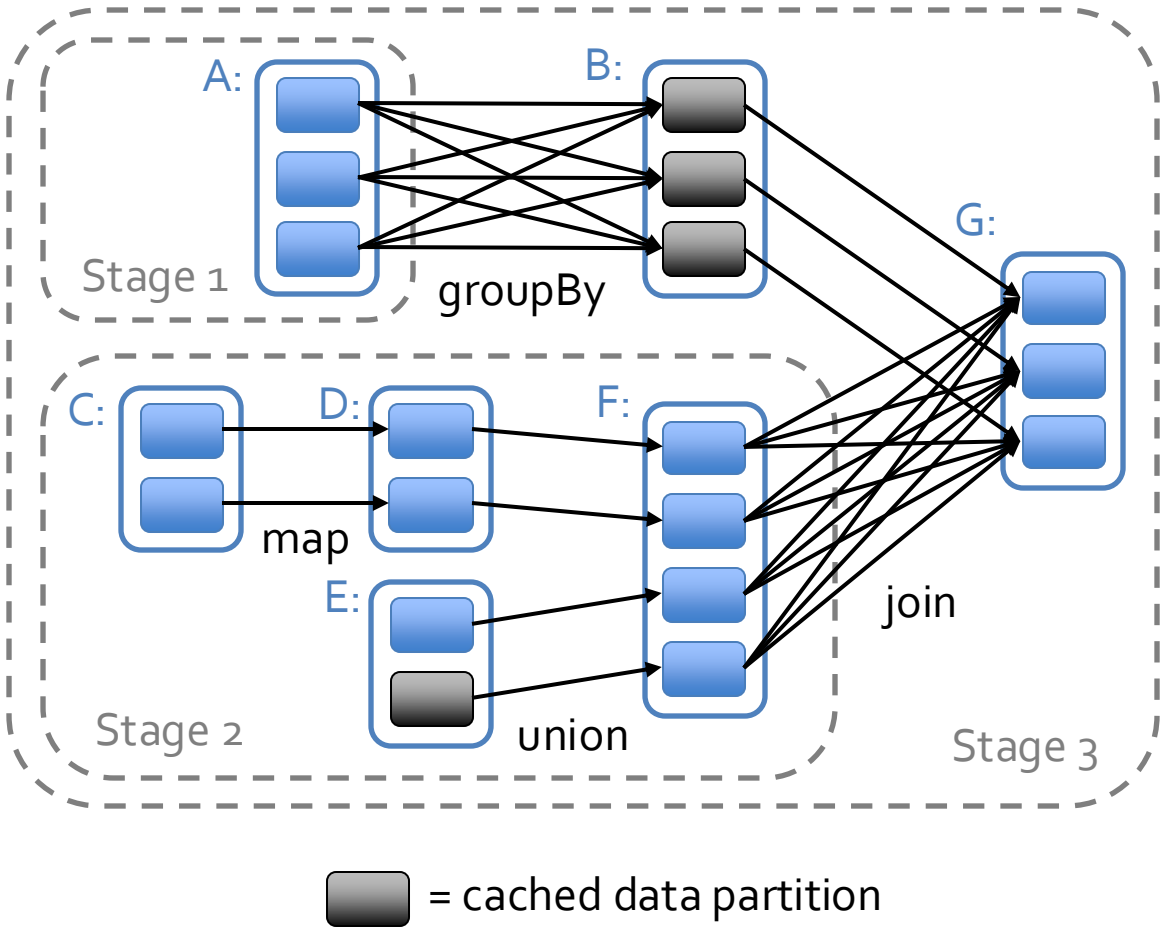
Implementation

- Runs on Apache Mesos to share resources with Hadoop & other apps
- Can read from any Hadoop input source (e.g. HDFS)
- No changes to Scala compiler



Spark Scheduler

- Dryad-like DAGs
- Pipelines functions within a stage
- Cache-aware work reuse & locality
- Partitioning-aware to avoid shuffles



Interactive Spark

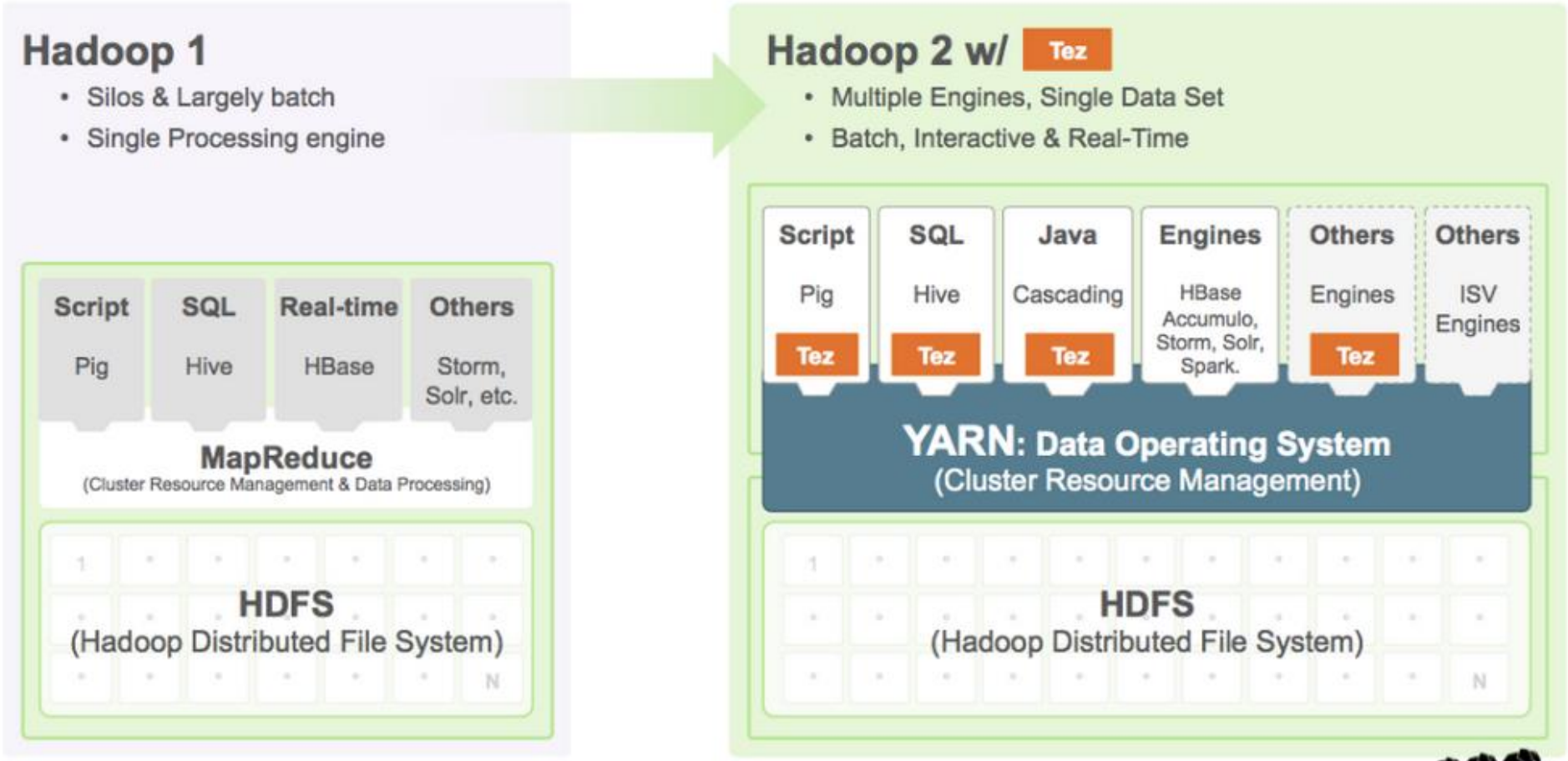
- Modified Scala interpreter to allow Spark to be used interactively from the command line
- Requires two changes:
  - Modified wrapper code generation so that each line typed has references to objects for its dependencies
  - Distribute generated classes over the network

Spark Operations

Transformations (define a new RDD)	map filter sample groupByKey reduceByKey sortByKey	flatMap union join cogroup cross mapValues
Actions (return a result to driver program)	collect reduce count save lookupKey	

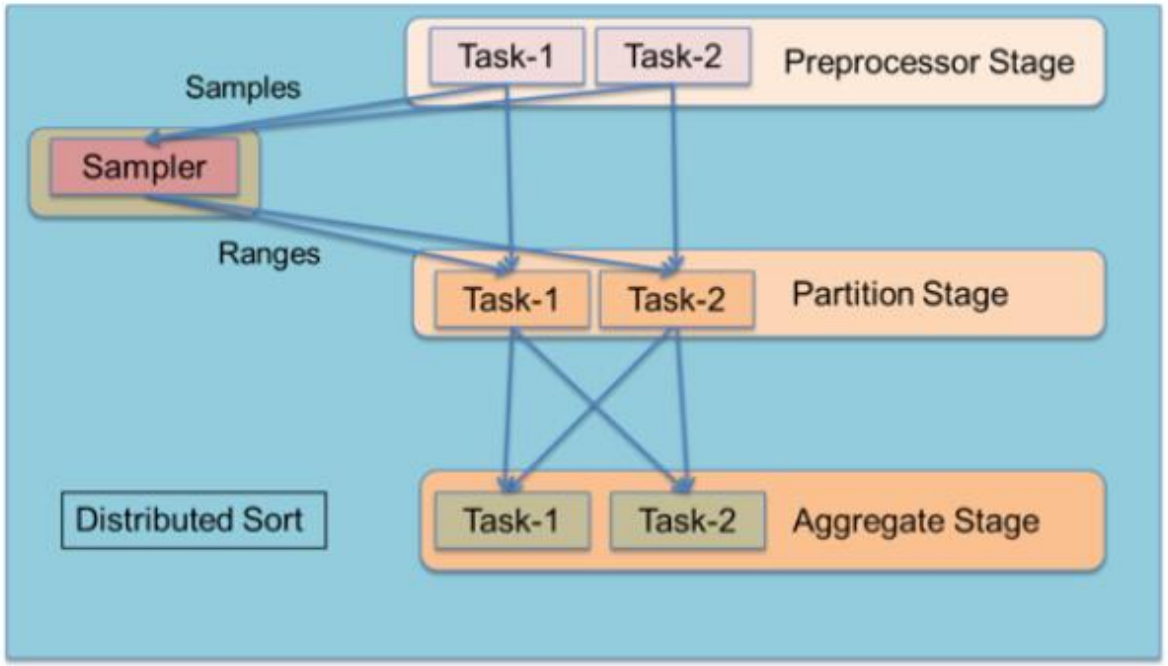
Apache Tez

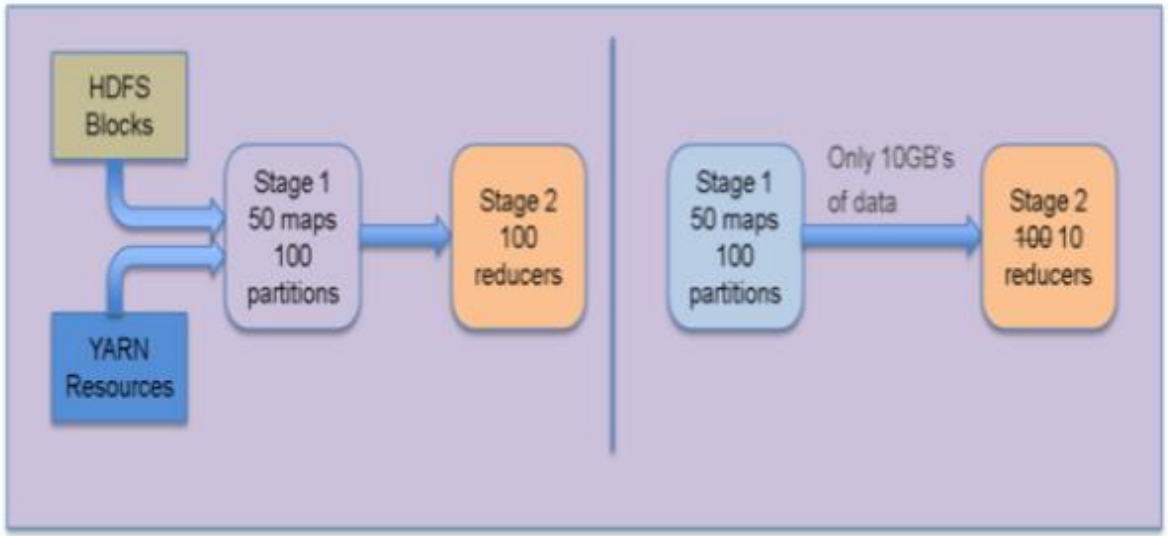
The Apache TEZ® project is aimed at building an application framework which allows for a complex directed-acyclic-graph (DAG) of tasks for processing data. It is currently built atop Apache Hadoop YARN.



Hortonworks

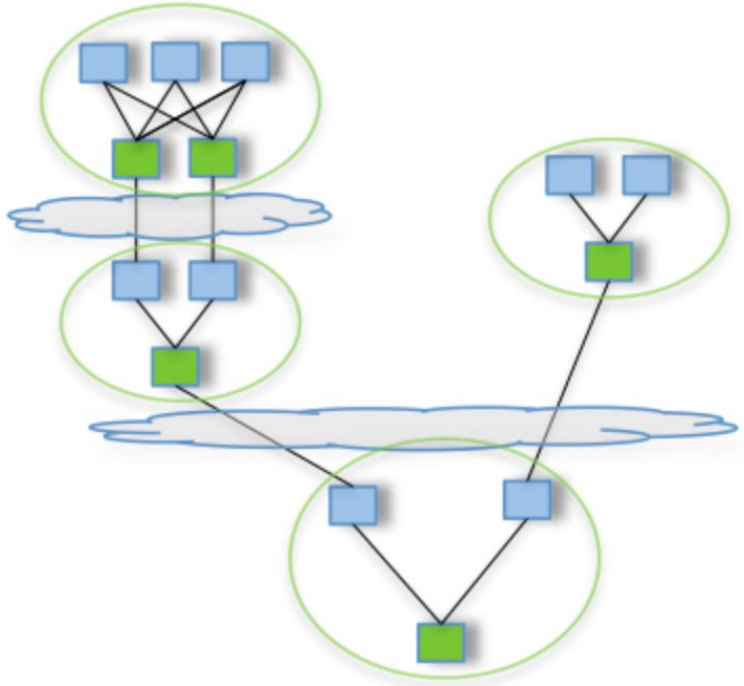
Tez's characteristics



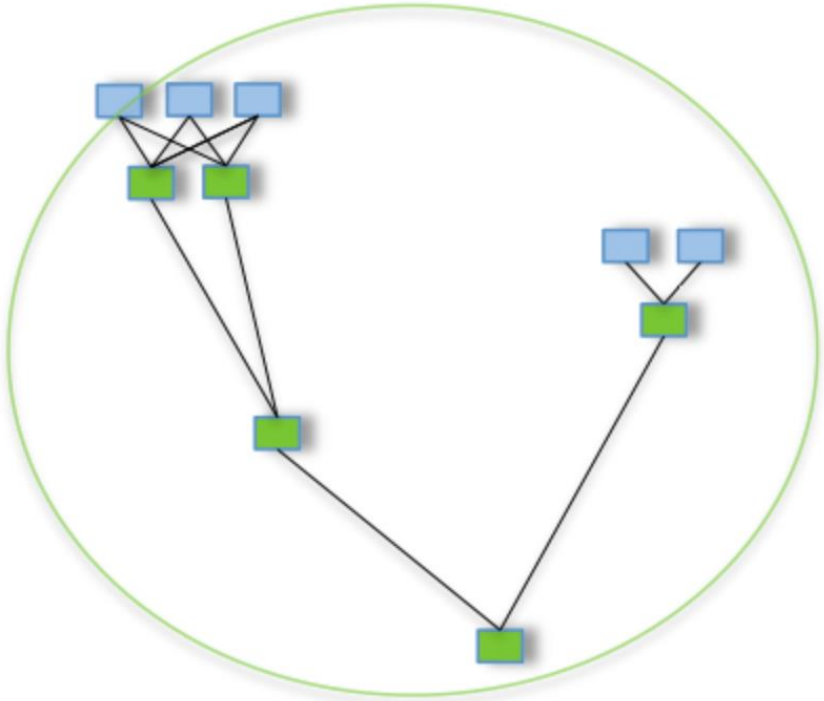


- Dataflow graph with vertices to express, model, and execute data processing logic
- Performance via Dynamic Graph Reconfiguration
- Flexible Input-Processor-Output task model
- Optimal Resource Management

By allowing projects like Apache Hive and Apache Pig to run a complex DAG of tasks, Tez can be used to process data, which earlier took multiple MR jobs, now in a single Tez job as shown below.



Pig/Hive - MR



Pig/Hive - Tez

Apache Storm

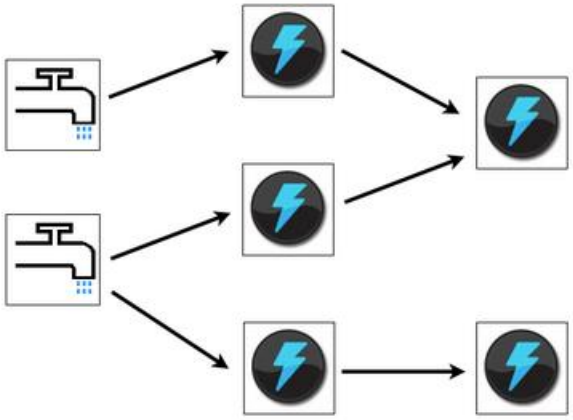
Stream Processing

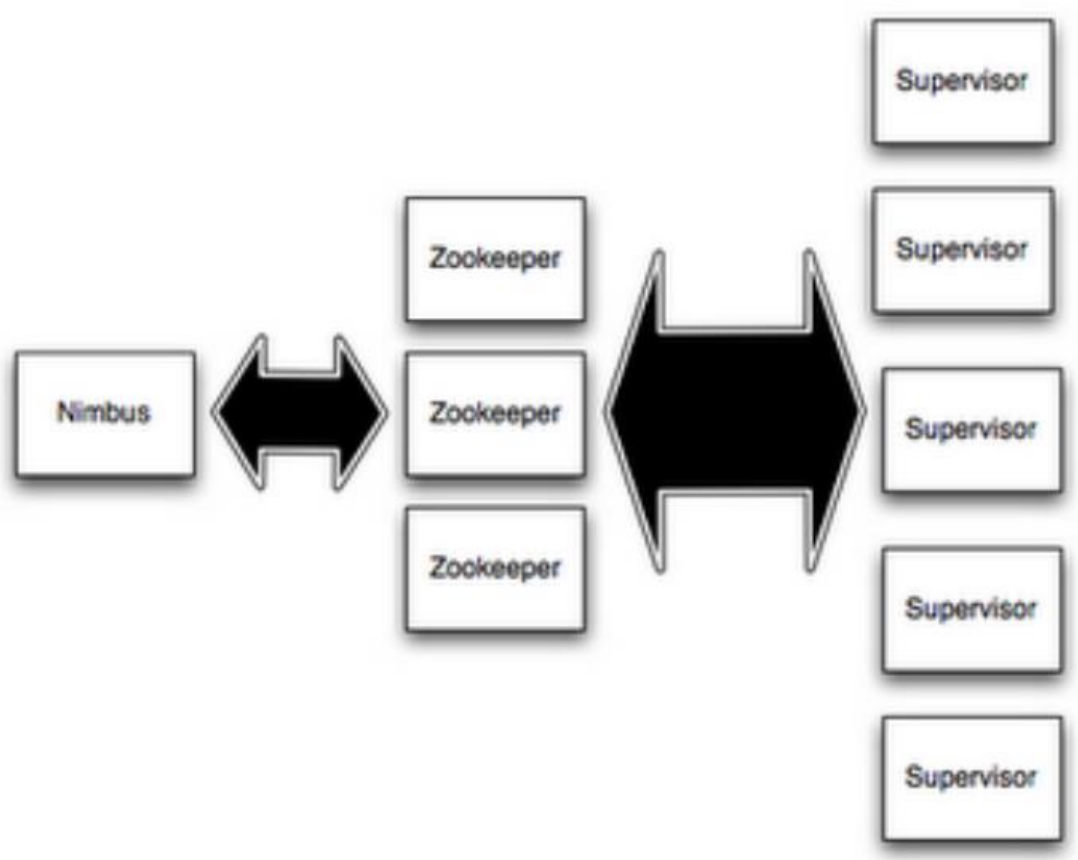
-- On Hadoop, you run MapReduce jobs; On Storm, you run Topologies.

-- Two kinds of nodes on a Storm cluster:

  -- the master node runs “Nimbus”

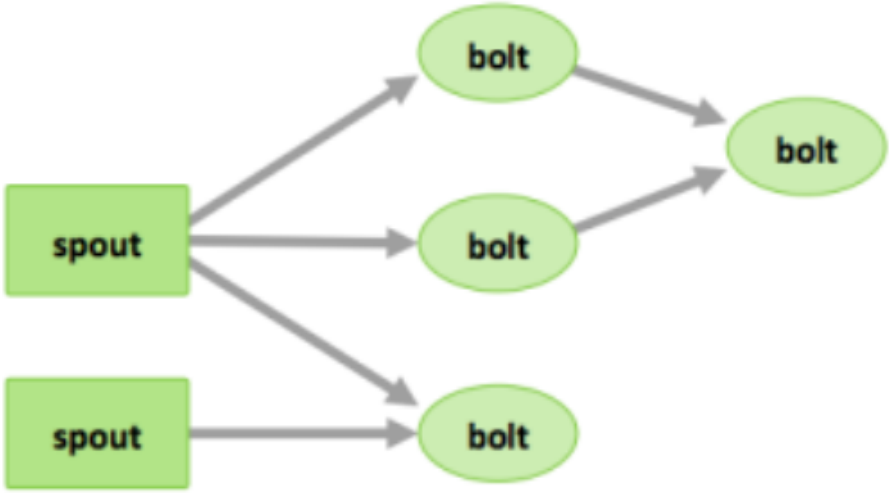
  -- the worker nodes called the Supervisor.





How Storm processes data?

- **Tuples**– an ordered list of elements. For example, a “4-tuple” might be (7, 1, 3, 7)
- **Streams** – an unbounded sequence of tuples.
- **Spouts** –sources of streams in a computation (e.g. a Twitter API)
- **Bolts** – process input streams and produce output streams. They can: run functions; filter, aggregate, or join data; or talk to databases.
- **Topologies** – the overall calculation, represented visually as a network of spouts and bolts (as in the following diagram)



Storm's Goals and Plans

Initiative Goals

Streams in HDP

Bringing stream data processing to enterprise Apache Hadoop and Hortonworks Data Platform.

Storm on YARN

Use the YARN Hadoop operating system to allow multiple workloads to be applied to Hadoop data simutaneously.

Enterprise Readiness

Bring baseline high availability, management, authentication and advanced scheduling to Storm.

Phase 1

Delivered
Storm 0.9.1
(HDP 2.1)

- Manage & Monitor via Amb
- Kafka, HBase & HDFS Connectors
- Windows Support

Phase 2

Coming Soon

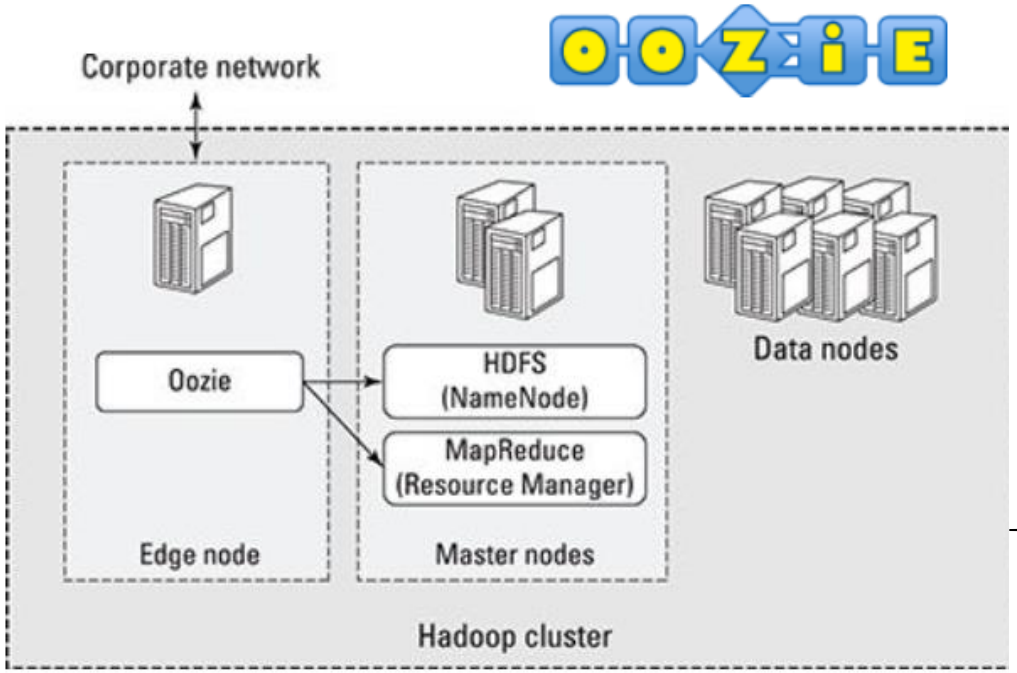
- Storm-on-YARN
- Ingest & Notification for JMS
- Data Persistance: EDWs & RDBMS'
- Kerberos Support for Nimbus
- User Authorization for Topologies
- Hive Connector for Hive Table Updates

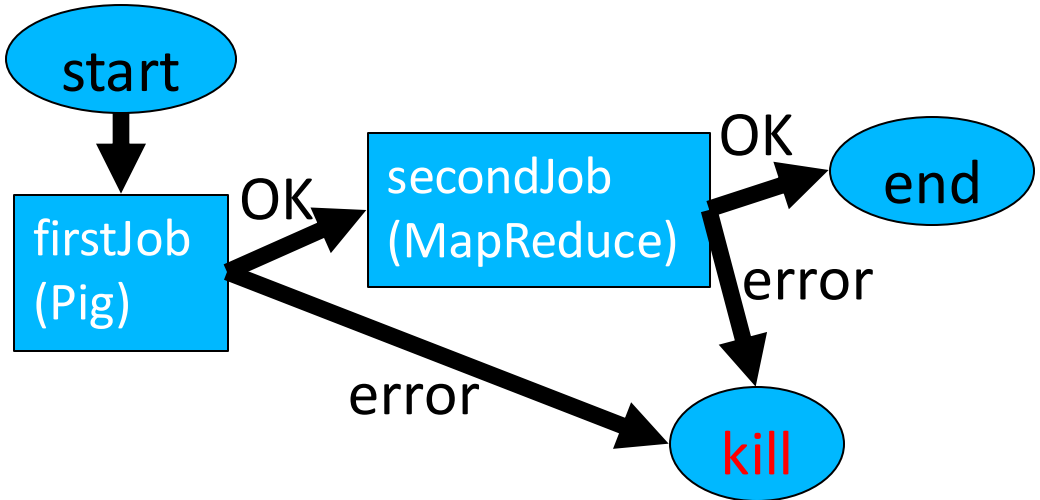
Phase 3

- Nimbus HA Management & Setup w/ Ambari
- Advanced Scheduler
- Ambari for Topology Management & Monitoring
- Simplified topology development

Oozie Workflow Scheduler for Hadoop

- Oozie supports a wide range of job types, including Pig, Hive, and MapReduce, as well as jobs coming from Java programs and Shell scripts.

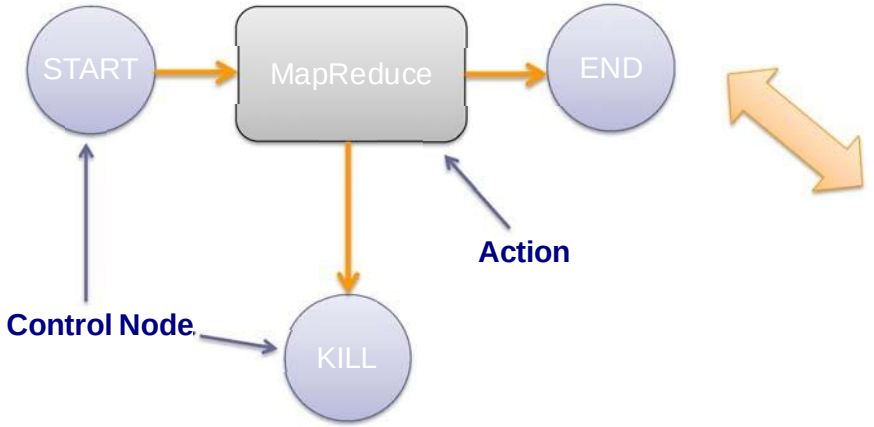





```
<workflow-app name="SampleWorkflow"
xmlns="uri:oozie:workflow:0.1">
 <start to="firstJob"/>
 <action name="firstJob">
  <pig>...</pig>
  <ok to="secondJob"/>
  <error to="kill"/>
 </action>
 <action name="secondJob">
  <map-reduce>...</map-reduce>
  <ok to="end" />
  <error to="kill" />
 </action>
 <end name="end"/>
 <kill name="kill">
  <message>"Killed job."</message>
 </kill>
</workflow-app>
```

A sample Oozie XML file

Action and Control Nodes




```
<workflow-app name="foo-wf"..
<start to="[NODE-NAME]"/>
<map-reduce>
...
...
</map-reduce>
<kill name="[NODE-NAME]">
<message>Error occurred</message
</kill>
<end name="[NODE-NAME]"/>
</workflow-app>
```

- **Control Flow**
  - start, end, kill
  - decision
  - fork, join
- **Actions**
  - MapReduce
  - Java
  - Pig
  - Hive
  - HDFS commands

- **Workflows begin with START node**
- **Workflows succeed with END node**
- **Workflows fail with KILL node**
- **Several actions support JSP Expression Language (EL)**

- **Oozie Coordination Engine can trigger workflows by**
  - Time (Periodically)
  - Data Availability (Data appears in a directory)

Schedule Workflow by Time

```
<coordinator-app name="sampleCoordinator"
        frequency="${coord:days(1)}"
        start="2014-06-01T00:01Z "
        end="2014-06-01T01:00Z "
        timezone="UTC"
        xmlns="uri:oozie:coordinator:0.1">
<controls>...</controls>
<action>

    <workflow>
      <app-path>${workflowAppPath}</app-path>
    </workflow>
  </action>
</coordinator-app>
```

Schedule Workflow by Time and Data Availability

```
<coordinator-app name="sampleCoordinator"
        frequency="${coord:days(1)}"
        start="${startTime}"
        end="${endTime}"
        timezone="${timeZoneDef}"
        xmlns="uri:oozie:coordinator:0.1">
 <controls>...</controls>
 <datasets>
  <dataset name="input" frequency="${coord:days(1)}" initial-instance="${startTime}" timezone="${timeZoneDef}">
     <uri-template>${triggerDatasetDir}</uri-template>
  </dataset>
 </datasets>
 <input-events>
     <data-in name="sampleInput" dataset="input">
     <instance>${startTime}</instance>
  </data-in>
 </input-events>
  <action>
   <workflow>
     <app-path>${workflowAppPath}</app-path>
   </workflow>
  </action>
</coordinator-app>
```

Install Oozie

- **$ mkdir <OOZIE_HOME>/libext**
- **Download ExtJS and place under <OOZIE_HOME>/libext**
  - ext-2.2.zip
- **Place Hadoop libs under libext**
  - $ cd <OOZIE_HOME>
  - $ tar xvf oozie-hadooplibs-3.1.3-cdh4.0.0.tar.gz
  - $ cp oozie-3.1.3-cdh4.0.0/hadooplibs/hadooplib-2.0.0- cdh4.0.0/*.jar libext/
- **Configure Oozie with components under libext**
  - $ bin/oozie-setup.sh
- **Create environment variable for default url**
  - export OOZIE_URL=http://localhost:11000/oozie
  - This allows you to use $oozie command without providing url
- **Update oozie-site.xml to point to Hadoop configuration**

```
<property>
<name>oozie.service.HadoopAccessorService.hadoop.configurations</name>
<value>*=/home/hadoop/Training/CDH4/hadoop-2.0.0-cdh4.0.0/conf</value>
</property>
```

- **Setup Oozie database**
  - $./bin/ooziedb.sh create -sqlfile oozie.sql -run DB Connection.

Install Oozie

- **Update core-site.xml to allow Oozie become "hadoop" and for that user to connect from any host**

```
<property>
<name>hadoop.proxyuser.hadoop.groups</name>
<value>*</value>
<description>Allow the superuser oozie to impersonate any members of the group group1 and
group2</description>
</property>
<property>
<name>hadoop.proxyuser.hadoop.hosts</name>
<value>*</value>
<description>The superuser can connect only from host1 and host2 to impersonate a
user</description>
</property>
```

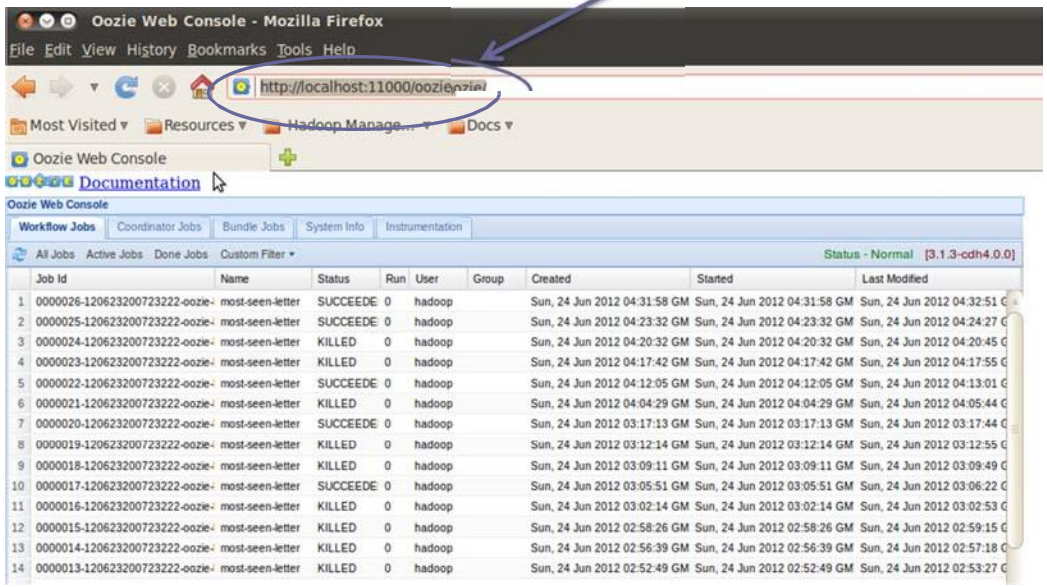
Start Oozie

```
$ oozie-start.sh
```

```
Setting OOZIE_HOME:          /home/hadoop/Training/CDH4/oozie-3.1.3-cdh4.0.0
Setting OOZIE_CONFIG:        /home/hadoop/Training/CDH4/oozie-3.1.3-cdh4.0.0/conf
Sourcing:                    /home/hadoop/Training/CDH4/oozie-3.1.3-cdh4.0.0/conf/oozie-env.sh
  setting OOZIE_LOG=/home/hadoop/Training/logs/oozie
setting CATALINA_PID=/home/hadoop/Training/hadoop_work/pids/oozie.pid
Setting OOZIE_CONFIG_FILE:   oozie-site.xml
Setting OOZIE_DATA:   /home/hadoop/Training/CDH4/oozie-3.1.3-cdh4.0.0/data  Using
  OOZIE_LOG:          /home/hadoop/Training/logs/oozie
Setting OOZIE_LOG4J_FILE:    oozie-log4j.properties  Setting OOZIE_LOG4J_RELOAD:
  10
Setting OOZIE_HTTP_HOSTNAME: localhost  Setting OOZIE_HTTP_PORT:      11000
Setting OOZIE_ADMIN_PORT:  11001
...
...
...
```

```
$ oozie admin -status
System mode: NORMAL
```

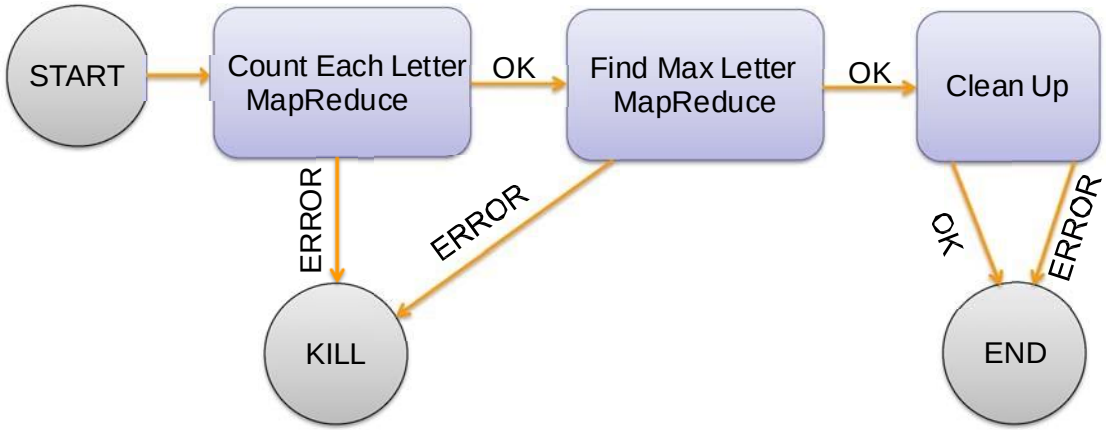
http://localhost:11000/oozie/

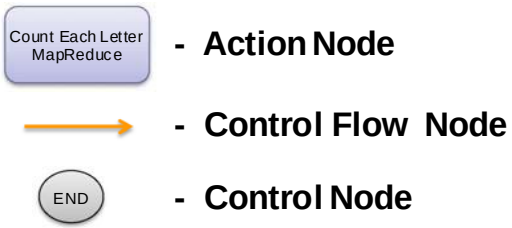


Running Oozie Examples

- **Extract examples packaged with Oozie**
  - $ cd $OOZIE_HOME
  - $ tar xvf oozie-examples.tar.gz
- **Copy examples to HDFS from user's home directory**
  - $ hdfs dfs -put examples examples
- **Run an example**
  - $ oozie job -config examples/apps/map-reduce/job.properties - run
- **Check Web Console**
  - http://localhost:11000/oozie/

An example workflow





This source is in *HadoopSamples* project under `/src/main/resources/mr/workflows`

Workflow definition

```
<workflow-app xmlns="uri:oozie:workflow:0.2" name="most-seen-letter">
    <start to="count-each-letter"/>
    <action name="count-each-letter">
        <map-reduce>
            <job-tracker>${jobTracker}</job-tracker>
            <name-node>${nameNode}</name-node>
            <prepare>
                <delete path="${nameNode}${outputDir}"/>
                <delete path="${nameNode}${intermediateDir}"/>
            </prepare>
            <configuration>
                        ...
                <property>
                    <name>mapreduce.job.map.class</name>
                    <value>mr.wordcount.StartsWithCountMapper</value>
                    </property>
                                        ...
                    </configuration>
                    </map-reduce>
        <ok to="find-max-letter"/>
        <error to="fail"/>
    </action>
...
```

START Action Node to count-each-letter MapReduce action

MapReduce have optional Prepare section

Pass property that will be set on MapReduce job's Configuration object

In case of success, go to the next job; in case of failure, go to fail node

Package and Run Your Workflow

1. **Create application directory structure with workflow definitions and resources**
   - Workflow.xml, jars, etc..
2. **Copy application directory to HDFS**
3. **Create application configuration file**
   - specify location of the application directory on HDFS
   - specify location of the namenode and resource manager
4. **Submit workflow to Oozie**
   - Utilize oozie command line
5. **Monitor running workflow(s)**

- **Two options**
  - Command line ($oozie)
  - Web Interface (http://localhost:11000/oozie)

Oozie Application Directory

- **Must comply to directory structure spec**

```
mostSeenLetter-oozieWorkflow
|--lib/
|     |--HadoopSamples.jar
|--workflow.xml
```

Application Workflow Root

Libraries should be placed under lib directory

Workflow.xml defines workflow

Bi-Annual Data Exposition

Every other year, at the Joint Statistical Meetings, the Graphics Section and the Computing Section join in sponsoring a special Poster Session called **The Data Exposition** , but more commonly known as **The Data Expo**. All of the papers presented in this Poster Session are reports of analyses of a common data set provided for the occasion. In addition, all papers presented in the session are encouraged to report the use of graphical methods employed during the development of their analysis and to use graphics to convey their findings.

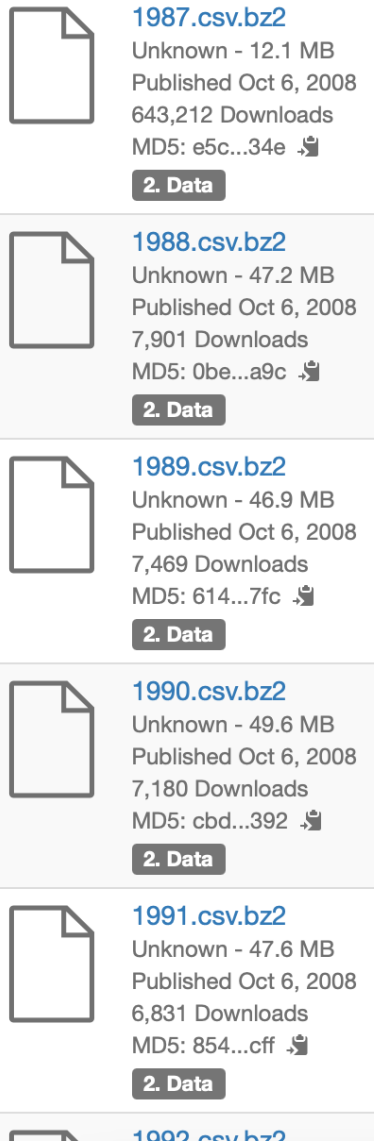
Data sets

- 2013: Soul of the Community
- 2011: Deepwater horizon oil spill
- 2009: Airline on time data
- 2006: NASA meteorological data. Electronic copy of entries
- 1997: Hospital Report Cards
- 1995: U.S. Colleges and Universities
- 1993: Oscillator time series & Breakfast Cereals
- 1991: Disease Data for Public Health Surveillance
- 1990: King Crab Data
- 1988: Baseball
- 1986: Geometric Features of Pollen Grains
- 1983: Automobiles

http://stat-computing.org/dataexpo/

Airline On-time Performance Dataset

- Data Source: Airline On-time Performance data set (flight data set).
  - All the logs of domestic flights from the period of October 1987 to April 2008.
  - Each record represents an individual flight where various details are captured:
    - Time and date of arrival and departure
    - Originating and destination airports
    - Amount of time taken to taxi from the runway to the gate.
  - Download it from:
    - http://stat-computing.org/dataexpo/2009/
    - https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/HG7NV7



Flight Data Schema

	Name	Description
1	Year	1987-2008
2	Month	1-12
3	DayofMonth	1-31
4	DayOfWeek	1 (Monday) - 7 (Sunday)
5	DepTime	actual departure time (local, hhmm)
6	CRSDepTime	scheduled departure time (local, hhmm)
7	ArrTime	actual arrival time (local, hhmm)
8	CRSArrTime	scheduled arrival time (local, hhmm)
9	UniqueCarrier	unique carrier code
10	FlightNum	flight number
11	TailNum	plane tail number
12	ActualElapsedTime	in minutes
13	CRSElapsedTime	in minutes
14	AirTime	in minutes
15	ArrDelay	arrival delay, in minutes
16	DepDelay	departure delay, in minutes
17	Origin	origin IATA airport code
18	Dest	destination IATA airport code
19	Distance	in miles
20	TaxiIn	taxi in time, in minutes
21	TaxiOut	taxi out time in minutes
22	Cancelled	was the flight cancelled?
23	CancellationCode	reason for cancellation
24	Diverted	1 = yes, 0 = no
25	CarrierDelay	in minutes
26	WeatherDelay	in minutes
27	NASDelay	in minutes
28	SecurityDelay	in minutes
29	LateAircraftDelay	in minutes

MapReduce Use Case Example – flight data

- Problem: count the number of flights for each carrier
- Solution using a serial approach (not MapReduce):

Listing 6-1: Pseudocode for Calculating The Number of Flights By Carrier Serially

```
create a two-dimensional array
create a row for every airline carrier
  populate the first column with the carrier code
  populate the second column with the integer zero

for each line of flight data
  read the airline carrier code
  find the row in the array that matches the carrier code
    increment the counter in the second column by one

print the totals for each row in the two-dimensional array
```

MapReduce Use Case Example – flight data

- Problem: count the number of flights for each carrier
- Solution using MapReduce (parallel way):

Listing 6-2: Pesudocode for Calculating The Number of Flights By Carrier in Parallel

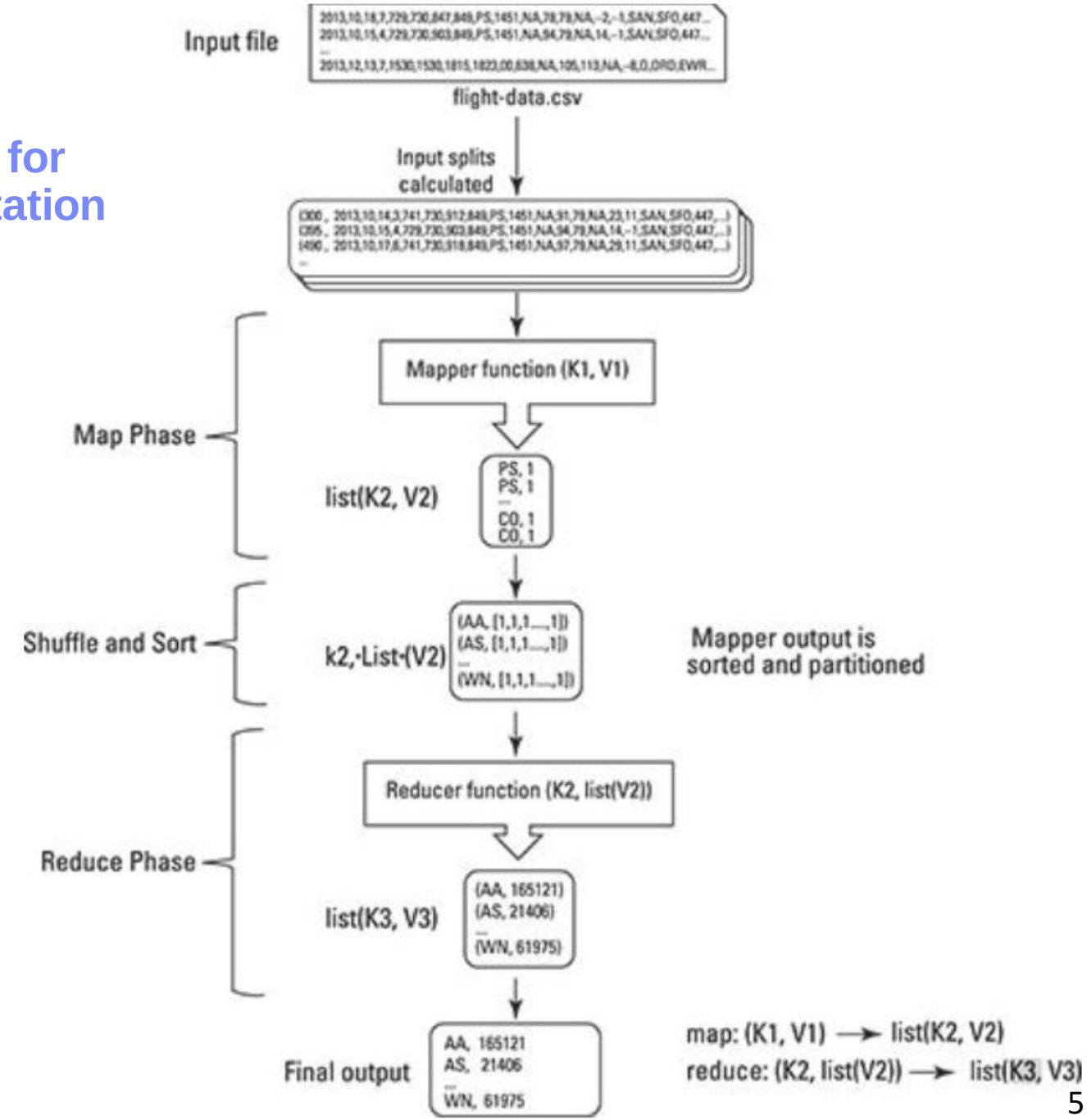
```
Map Phase:
for each line of flight data
  read the current record and extract the airline carrier code
  output the airline carrier code and the number one as a key/value pair

Shuffle and Sort Phase:
  read the list of key/value pairs from the map phase
  group all the values for each key together
    each key has a corresponding array of values
  sort the data by key
  output each key and its array of values

Reduce Phase:
  read the list of carriers and arrays of values from the shuffle and sort phase
  for each carrier code
    add the total number of ones in the carrier code's array of values together

print the totals for each row in the two-dimensional array
```

MapReduce steps for flight data computation



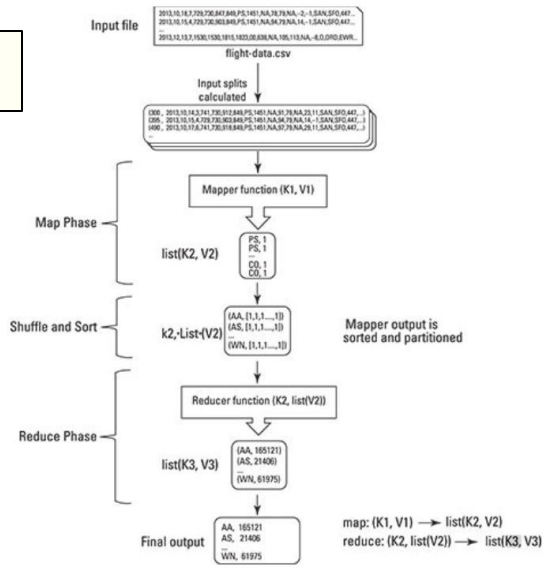
FlightsByCarrier application

Create FlightsByCarrier.java:

Listing 6-3: The FlightsByCarrier Driver Application

```
@@1
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class FlightsByCarrier {
    public static void main(String[] args) throws Exception
    @@2
    Job job = new Job();
    job.setJarByClass(FlightsByCarrier.class);
    job.setJobName("FlightsByCarrier");
```



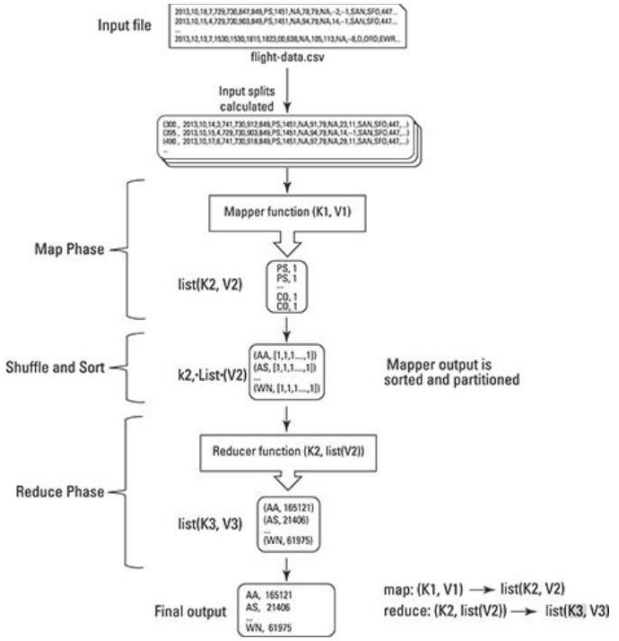
FlightsByCarrier application

```
@@3
TextInputFormat.addInputPath(job, new Path(args[0]));
job.setInputFormatClass(TextInputFormat.class);

@@4
job.setMapperClass(FlightsByCarrierMapper.class);
job.setReducerClass(FlightsByCarrierReducer.class);

@@5
TextOutputFormat.setOutputPath(job, new Path(args[1]));
job.setOutputFormatClass(TextOutputFormat.class);
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);

@@6
job.waitForCompletion(true);
      }
}
```

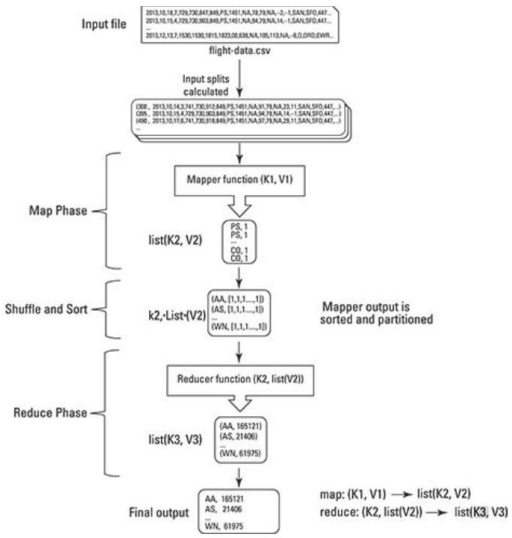


FlightsByCarrier Mapper

Listing 6-4: The FlightsByCarrier Mapper Code

```
 @@1
import java.io.IOException;
import au.com.bytecode.opencsv.CSVParser;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Mapper;

@@2
public class FlightsByCarrierMapper extends
        Mapper<LongWritable, Text, Text, IntWritable> {
    @Override
    @@3
    protected void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
      @@4
      if (key.get() > 0) {
       String[] lines = new
      CSVParser().parseLine(value.toString());
       @@5
       context.write(new Text(lines[8]), new IntWritable(1));
      }
    }
}
```



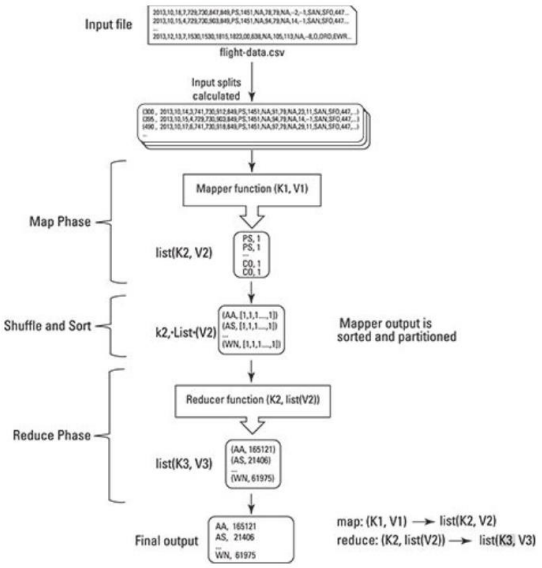
FlightsByCarrier Reducer

Listing 6-5: The FlightsByCarrier Reducer Code

```
 @@1
import java.io.IOException;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.Reducer;

@@2
public class FlightsByCarrierReducer extends
        Reducer<Text, IntWritable, Text, IntWritable> {
    @Override
    @@3
    protected void reduce(Text token, Iterable<IntWritable> counts,
            Context context) throws IOException, InterruptedException {
        int sum = 0;

        @@4
        for (IntWritable count : counts) {
                sum+= count.get();
        }
        @@5
        context.write(token, new IntWritable(sum));
    }
}
```



Run the code

To run the FlightsByCarrier application, follow these steps:

Go to the directory with your Java code and compile it using the following command:

```
javac -classpath $CLASSPATH MapRed/FlightsByCarrier/*.java
```

Build a JAR file for the application by using this command:

```
jar cvf FlightsByCarrier.jar *.class
```

Run the driver application by using this command:

```
hadoop jar FlightsByCarrier.jar FlightsByCarrier /user/root/airline-data/2008.csv /user/root/output/flightsCount
```

See Result

Show the job's output file from HDFS by running the command

```
hadoop fs -cat /user/root/output/flightsCount/part-r-00000
```

You see the total counts of all flights completed for each of the carriers in 2008:

```
AA    165121
AS    21406
CO    123002
DL    185813
EA    108776
HP    45399
NW    108273
PA (1)  16785
PI    116482
PS    41706
TW    69650
UA    152624
US    94814
WN    61975
```

Using Pig Script for fast application development

- Problem: calculate the total miles flown for all flights flown in one year
- How much work is needed using MapReduce?
- What if we use Pig?
  - totalmiles.pig

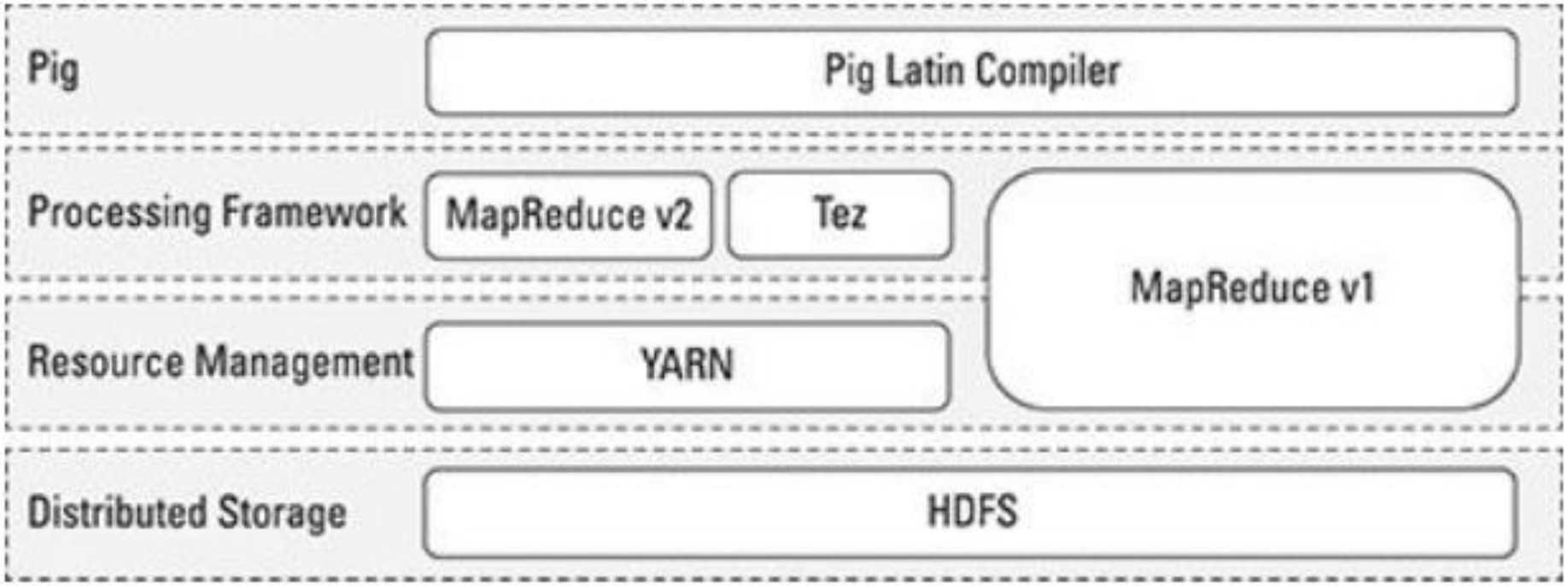
```
records = LOAD '2013_subset.csv' USING PigStorage(',') AS
   (Year,Month,DayofMonth,DayOfWeek,DepTime,CRSDepTime,ArrTime,\
   CRSArrTime,UniqueCarrier,FlightNum,TailNum,ActualElapsedTime,\
   CRSElapsedTime,AirTime,ArrDelay,DepDelay,Origin,Dest,\
   Distance:int,TaxiIn,TaxiOut,Cancelled,CancellationCode,\
   Diverted,CarrierDelay,WeatherDelay,NASDelay,SecurityDelay,\
   LateAircraftDelay);
milage_recs = GROUP records ALL;
tot_miles = FOREACH milage_recs GENERATE SUM(records.Distance);

STORE tot_miles INTO /user/root/totalmiles;
```

- Execute it: pig totalmiles.pig
- See result: hdfs dfs –cat /user/root/totalmiles/part-r-00000

➔ 775009272

Pig: a data flow language



At its core, Pig Latin is a *dataflow* language, where you define a data stream and a series of transformations that are applied to the data as it flows through your application. This is in contrast to a *control flow* language (like C or Java), where you write a series of instructions. In control flow languages, we use constructs like loops and conditional logic (like an if statement). You won't find loops and if statements in Pig Latin.

Data flow language

- Define a data stream
  - Typically using "LOAD …"
- Apply a series of transformations to the data
  - FILTER, GROUP, COUNT, DUMP, etc.

Pig example for WordCount

```
input_lines = LOAD '/tmp/my-copy-of-all-pages-on-internet' AS (line:chararray);

-- Extract words from each line and put them into a pig bag
-- datatype, then flatten the bag to get one word on each row
words = FOREACH input_lines GENERATE FLATTEN(TOKENIZE(line)) AS word;

-- filter out any words that are just white spaces
filtered_words = FILTER words BY word MATCHES '\\w+';

-- create a group for each word
word_groups = GROUP filtered_words BY word;

-- count the entries in each group
word_count = FOREACH word_groups GENERATE COUNT(filtered_words) AS count, group AS word;

-- order the records by count
ordered_word_count = ORDER word_count BY count DESC;
STORE ordered_word_count INTO '/tmp/number-of-words-on-internet';
```

Characteristics of Pig

Most Pig scripts start with the LOAD statement to read data from HDFS. In this case, we're loading data from a .csv file. Pig has a data model it uses, so next we need to map the file's data model to the Pig data mode. This is accomplished with the help of the USING statement. (More on the Pig data model in the next section.) We then specify that it is a comma-delimited file with the PigStorage(',') statement followed by the AS statement defining the name of each of the columns.

Aggregations are commonly used in Pig to summarize data sets. The GROUP statement is used to aggregate the records into a single record mileage_recs. The ALL statement is used to aggregate all tuples into a single group. Note that some statements — including the following SUM statement — requires a preceding GROUP ALL statement for global sums.

FOREACH ... GENERATE statements are used here to transform columns data. In this case, we want to count the miles traveled in the records_Distance column. The SUM statement computes the sum of the record_Distance column into a single-column collection total_miles.

The DUMP operator is used to execute the Pig Latin statement and display the results on the screen.DUMP is used in interactive mode, which means that the statements are executable immediately and the results are not saved. Typically, you will either use the DUMP or STORE operators at the end of your Pig script.

Pig vs. SQL

In comparison to SQL, Pig

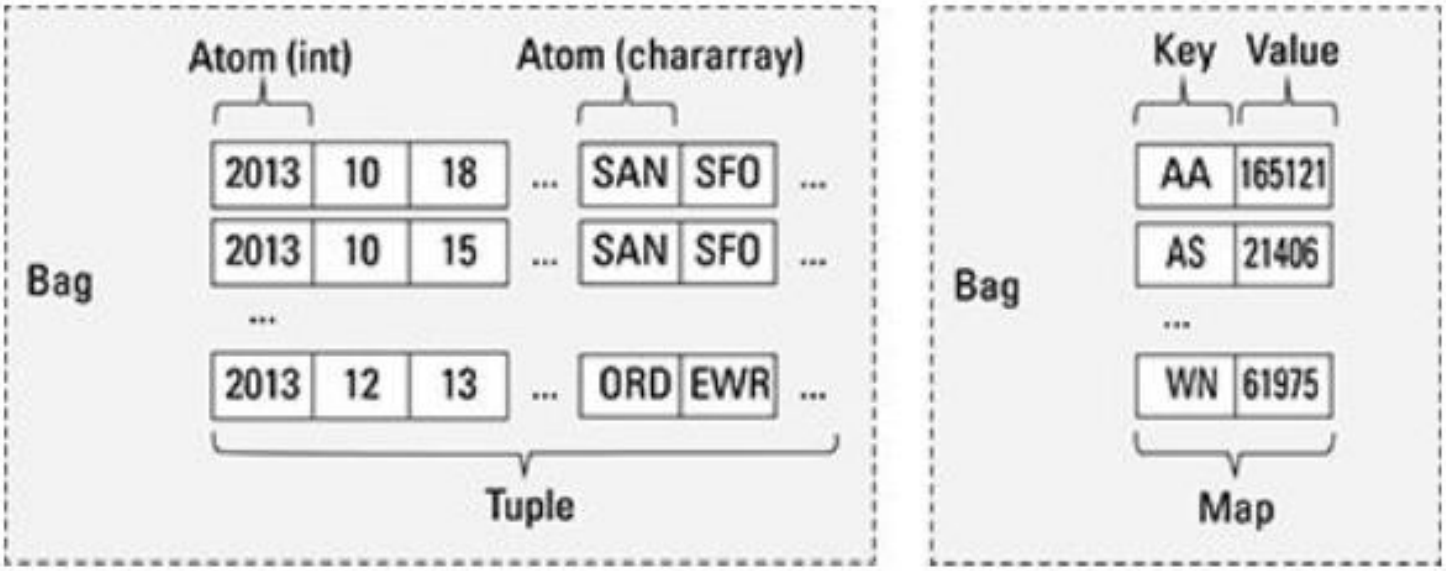
1. uses lazy evaluation,
2. uses ETL (Extract-Transform-Load),
3. is able to store data at any point during a pipeline,
4. declares execution plans,
5. supports pipeline splits.

On the other hand, it has been argued DBMSs are substantially faster than the MapReduce system once the data is loaded, but that loading the data takes considerably longer in the database systems. It has also been argued RDBMSs offer out of the box support for column-storage, working with compressed data, indexes for efficient random data access, and transaction-level fault tolerance.

Pig Latin is procedural and fits very naturally in the pipeline paradigm while SQL is instead declarative. In SQL users can specify that data from two tables must be joined, but not what join implementation to use. Pig Latin allows users to specify an implementation or aspects of an implementation to be used in executing a script in several ways.

Pig Latin programming is similar to specifying a query execution plan.

Pig Data Types and Syntax



Atom: An atom is any single value, such as a string or number

Tuple: A tuple is a record that consists of a sequence of fields. Each field can be of any type.

Bag: A bag is a collection of non-unique tuples.

Map: A map is a collection of key value pairs.

Pig Latin Operators

Operation	*Operator*	*Explanation*
Data Access	LOAD/STORE	Read and Write data to file system
	DUMP	Write output to standard output (stdout)
	STREAM	Send all records through external binary
	FOREACH	Apply expression to each record and output one or more records
	FILTER	Apply predicate and remove records that don't meet condition
	GROUP/COGROUP	Aggregate records with the same key from one or more inputs
	JOIN	Join two or more records based on a condition
Transformations	CROSS	Cartesian product of two or more inputs
	ORDER	Sort records based on key
	DISTINCT	Remove duplicate records
	UNION	Merge two data sets
	SPLIT	Divide data into two or more bags based on predicate
	LIMIT	subset the number of records

Pig Latin Expressions

Expressions

In Pig Latin, expressions are language constructs used with the FILTER, FOREACH, GROUP, and SPLIT operators as well as the eval functions.

Expressions are written in conventional mathematical infix notation and are adapted to the UTF-8 character set. Depending on the context, expressions can include:

- Any Pig data type (simple data types, complex data types)
- Any Pig operator (arithmetic, comparison, null, boolean, dereference, sign, and cast)
- Any Pig built-in function.
- Any user-defined function (UDF) written in Java.

In Pig Latin,

- An arithmetic expression could look like this:

```
X = GROUP A BY f2*f3;
```

- A string expression could look like this, where a and b are both chararrays:

```
X = FOREACH A GENERATE CONCAT(a,b);
```

- A boolean expression could look like this:

```
X = FILTER A BY (f1==8) OR (NOT (f2+f3 > f1));
```

Pig User-Defined Functions (UDFs)

```
-- myscript.pig
REGISTER myudfs.jar;
A = LOAD 'student_data' AS (name: chararray, age: int, gpa: float);
B = FOREACH A GENERATE myudfs.UPPER(name);
DUMP B;
```

Command to run the script.

```
java -cp pig.jar org.apache.pig.Main -x local myscript.pig
```

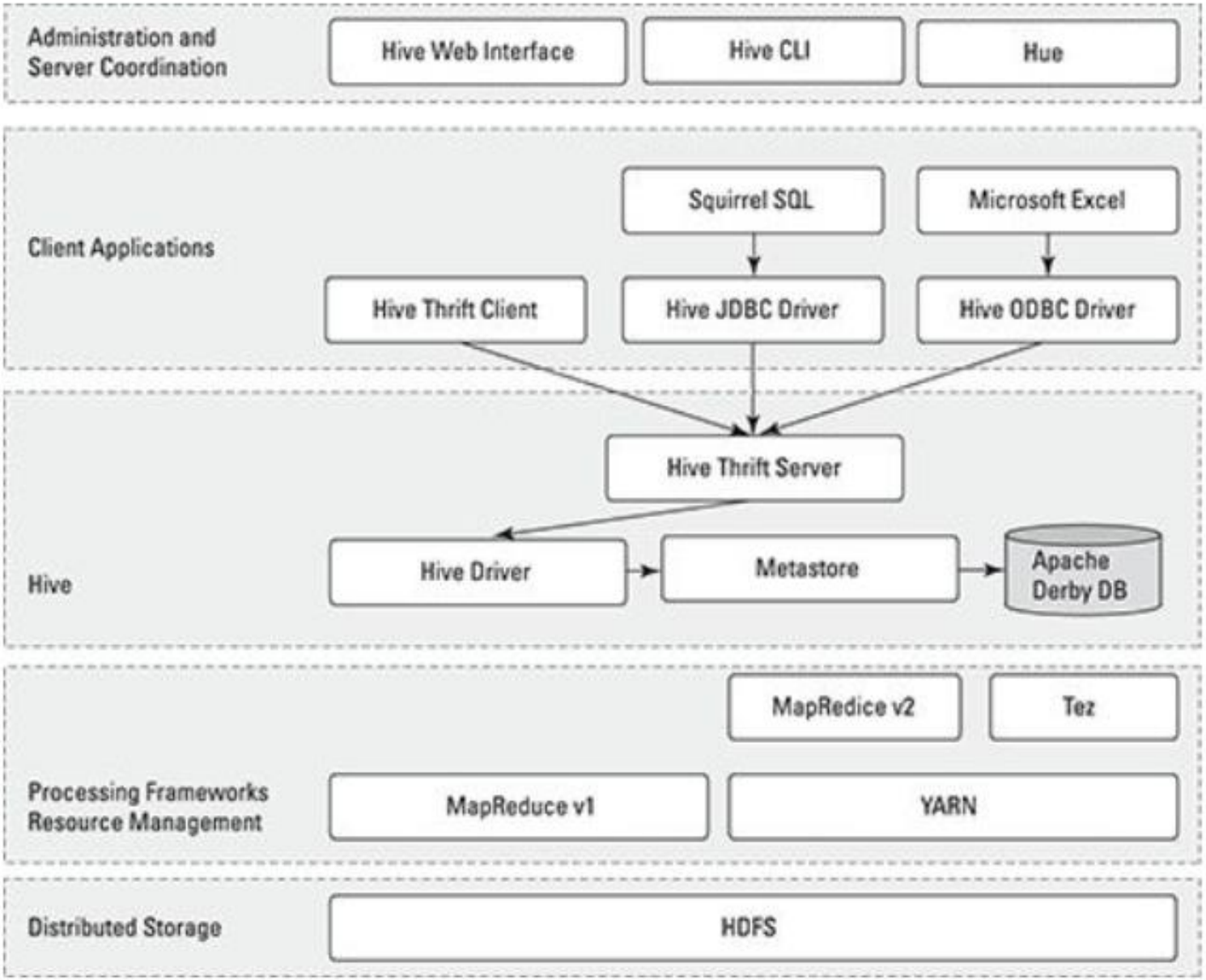
The .java contains UPPER

```
package myudfs;
import java.io.IOException;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.Tuple;
import org.apache.pig.impl.util.WrappedIOException;

public class UPPER extends EvalFunc (String)
{
    public String exec(Tuple input) throws IOException {
        if (input == null || input.size() == 0)
            return null;
        try{
            String str = (String)input.get(0);
            return str.toUpperCase();
        }catch(Exception e){
            throw WrappedIOException.wrap("Caught exception processing input row ", e);
        }
    }
}
```

https://pig.apache.org/docs/r0.7.0/udf.html

Apache Hive


Administration and Server Coordination
Hive Web Interface
Hive CLI
Hue
Client Applications
Squirrel SQL
Microsoft Excel
Hive Thrift Client
Hive JDBC Driver
Hive ODBC Driver
Hive Thrift Server
Hive
Hive Driver
Metastore
Apache Derby DB
MapRedice v2
Tez
Processing Frameworks Resource Management
MapReduce v1
YARN
Distributed Storage
HDFS

Using Hive to Create a Table

```
(A) $ $HIVE_HOME/bin hive --service cli
(B) hive> set hive.cli.print.current.db=true;
(C) hive (default)> CREATE DATABASE ourfirstdatabase;
OK
Time taken: 3.756 seconds
(D) hive (default)> USE ourfirstdatabase;
OK
Time taken: 0.039 seconds
(E) hive (ourfirstdatabase)> CREATE TABLE our_first_table (
                   > FirstName      STRING,
                   > LastName       STRING,
                   > EmployeeId      INT);
OK
Time taken: 0.043 seconds
hive (ourfirstdatabase)> quit;
(F) $ ls /home/biadmin/Hive/warehouse/ourfirstdatabase.db
our_first_table
```

Another Hive Example

```
(A) CREATE TABLE IF NOT EXISTS FlightInfo2007 (
Year SMALLINT, Month TINYINT, DayofMonth TINYINT, DayOfWeek TINYINT,
DepTime SMALLINT, CRSDepTime SMALLINT, ArrTime SMALLINT, CRSArrTime SMALLINT,
UniqueCarrier STRING, FlightNum STRING, TailNum STRING,
ActualElapsedTime SMALLINT, CRSElapsedTime SMALLINT,
AirTime SMALLINT, ArrDelay SMALLINT, DepDelay SMALLINT,
Origin STRING, Dest STRING,Distance INT,
TaxiIn SMALLINT, TaxiOut SMALLINT, Cancelled SMALLINT,
CancellationCode STRING, Diverted SMALLINT,
CarrierDelay SMALLINT, WeatherDelay SMALLINT,
NASDelay SMALLINT, SecurityDelay SMALLINT, LateAircraftDelay SMALLINT)
COMMENT 'Flight InfoTable'
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
STORED AS TEXTFILE
TBLPROPERTIES ('creator'='Bruce Brown', 'created_at'='Thu Sep 19 10:58:00 EDT 2013');
```

Exercises

1. Download Airline Data and one of your own selected datasets from Stat-Computing.org
3. Learn to use PIG. You can try the example in the reference
4. Use Oozie to schedule a few jobs
5. Try HBase. Use your own example
6. Try Hive. Use your own example

Bi-Annual Data Exposition

Every other year, at the Joint Statistical Meetings, the Graphics Section and the Computing Section join in sponsoring a special Poster Session called **The Data Exposition** , but more commonly known as **The Data Expo**. All of the papers presented in this Poster Session are reports of analyses of a common data set provided for the occasion. In addition, all papers presented in the session are encouraged to report the use of graphical methods employed during the development of their analysis and to use graphics to convey their findings.

Data sets

- 2013: Soul of the Community
- 2011: Deepwater horizon oil spill
- 2009: Airline on time data
- 2006: NASA meteorological data. Electronic copy of entries
- 1997: Hospital Report Cards
- 1995: U.S. Colleges and Universities
- 1993: Oscillator time series & Breakfast Cereals
- 1991: Disease Data for Public Health Surveillance
- 1990: King Crab Data
- 1988: Baseball
- 1986: Geometric Features of Pollen Grains
- 1983: Automobiles

http://stat-computing.org/dataexpo/

A project that combines all you've learned so far on big data storage, management, computing, and processing will be assigned after the midterm exam.