Big Data

Chapter 19
Principles of Database Management
Lemahieu et. al.
Cambridge

Introduction

- The 5 Vs of Big Data
- Hadoop
- SQL on Hadoop
- Apache Spark

The 5 Vs of Big Data

- Every minute:
  - more than 300,000 tweets are created
  - Netflix subscribers are streaming more than 70,000 hours of video at once
  - Apple users download 30,000 apps
  - Instagram users like almost two million photos
- Big Data encompasses both structured and highly unstructured forms of data

The 5 Vs of Big Data

- **Volume**: the amount of data, also referred to the data “at rest”
- **Velocity**: the speed at which data comes in and goes out, data “in motion”
- **Variety**: the range of data types and sources that are used, data in its “many forms”
- **Veracity**: the uncertainty of the data; data “in doubt”
- **Value**: TCO and ROI of the data

The 5 Vs of Big Data

- Examples:
  - Large-scale enterprise systems
    - e.g., ERP – Enterprise Resource Planning
    - CRM - Customer Relationship Management
    - SCM - Supply chain management
  - Social networks (e.g., Twitter, Weibo, WeChat)
  - Internet of Things
  - Open data

Hadoop

- Open-source software framework used for distributed storage and processing of big datasets
- Can be set up over a cluster of computers built from normal, commodity hardware
- Many vendors offer their implementation of a Hadoop stack (e.g., Amazon, Cloudera, Dell, Oracle, IBM, Microsoft)

Hadoop

- History of Hadoop
- The Hadoop stack

History of Hadoop

- Key building blocks:
  - Google File System: a file system that could be easily distributed across commodity hardware, while providing fault tolerance
  - Google MapReduce: a programming paradigm to write programs that can be automatically parallelized and executed across a cluster of different computers
- Nutch web crawler prototype developed by Doug Cutting
  - Later renamed to Hadoop
- In 2008, Yahoo! open-sourced Hadoop as “Apache Hadoop”

The Hadoop Stack

- Four modules:
  - Hadoop Common: a set of shared programming libraries used by the other modules
  - Hadoop Distributed File System (HDFS): a Java-based file system to store data across multiple machines
  - MapReduce framework: a programming model to process large sets of data in parallel
  - YARN (Yet Another Resource Negotiator): handles the management and scheduling of resource requests in a distributed environment

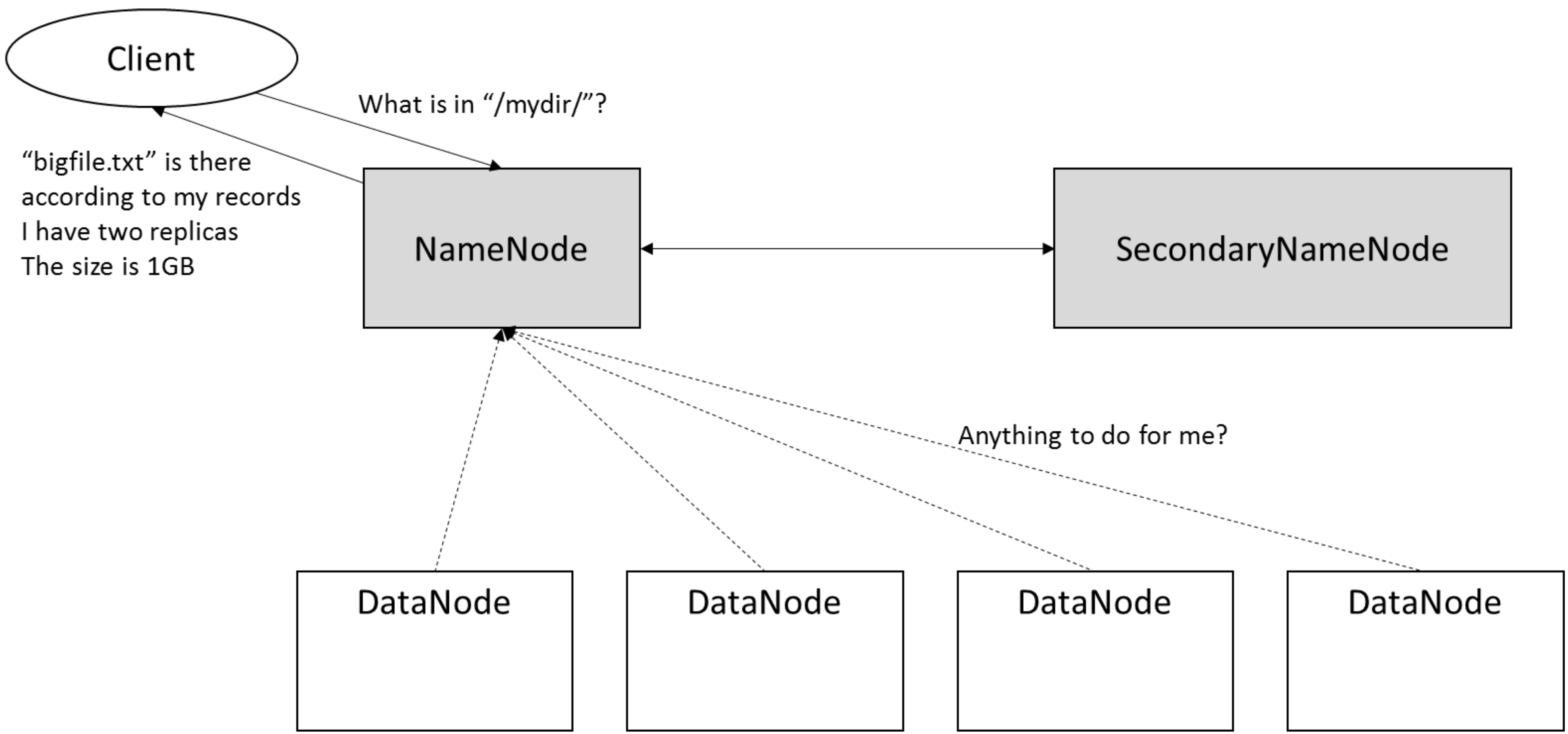
Hadoop Distributed File System (HDFS)

- Distributed file system to store data across a cluster of commodity machines
- High emphasis on fault tolerance
- HDFS cluster is composed of a NameNode and various DataNodes

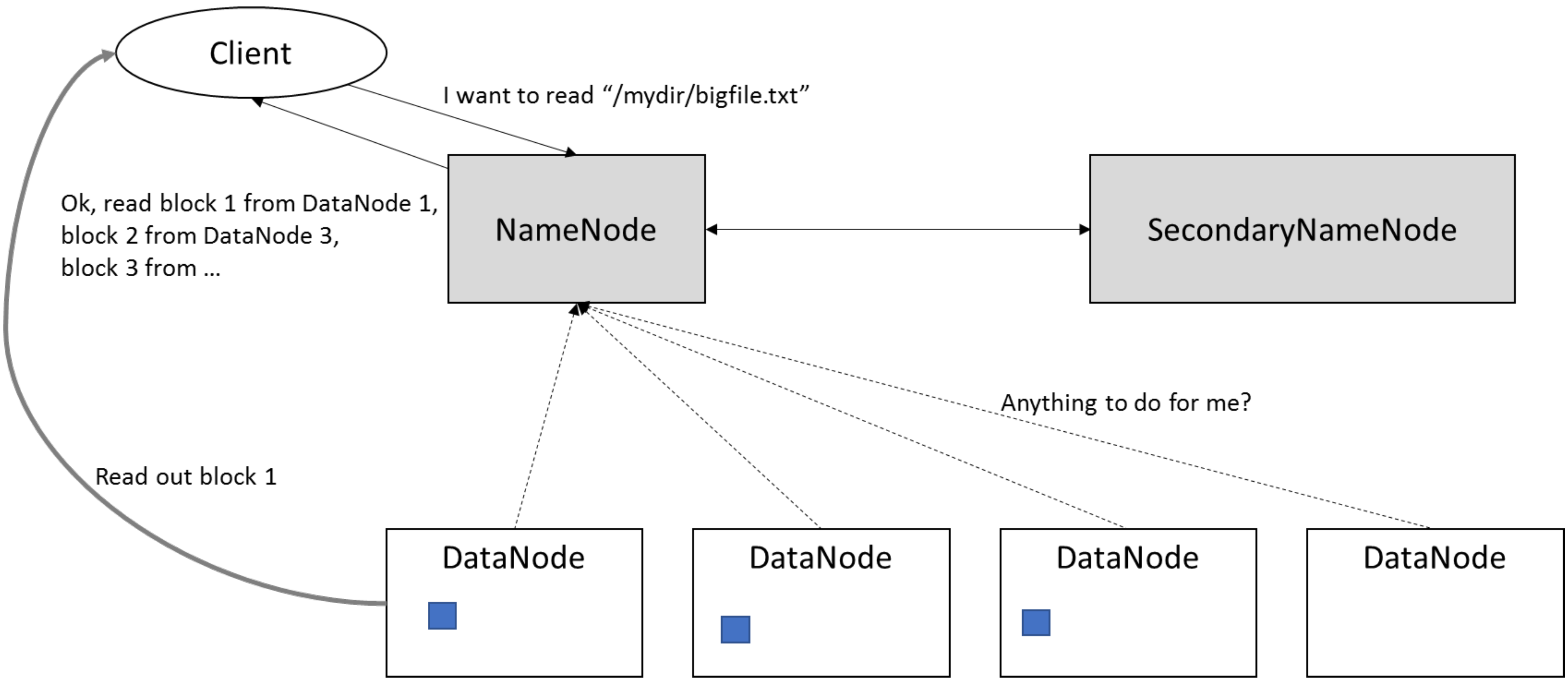
Hadoop Distributed File System (HDFS)

- NameNode
  - A server that holds all the metadata regarding the stored files (i.e., registry)
  - Manages incoming file system operations
  - Maps data blocks (parts of files) to DataNodes
- DataNode
  - Handles file read and write requests
  - Creates, deletes, and replicates data blocks among their disk drives
  - Continuously loop, asking the NameNode for instructions
- Note: size of one data block is typically 64 megabytes

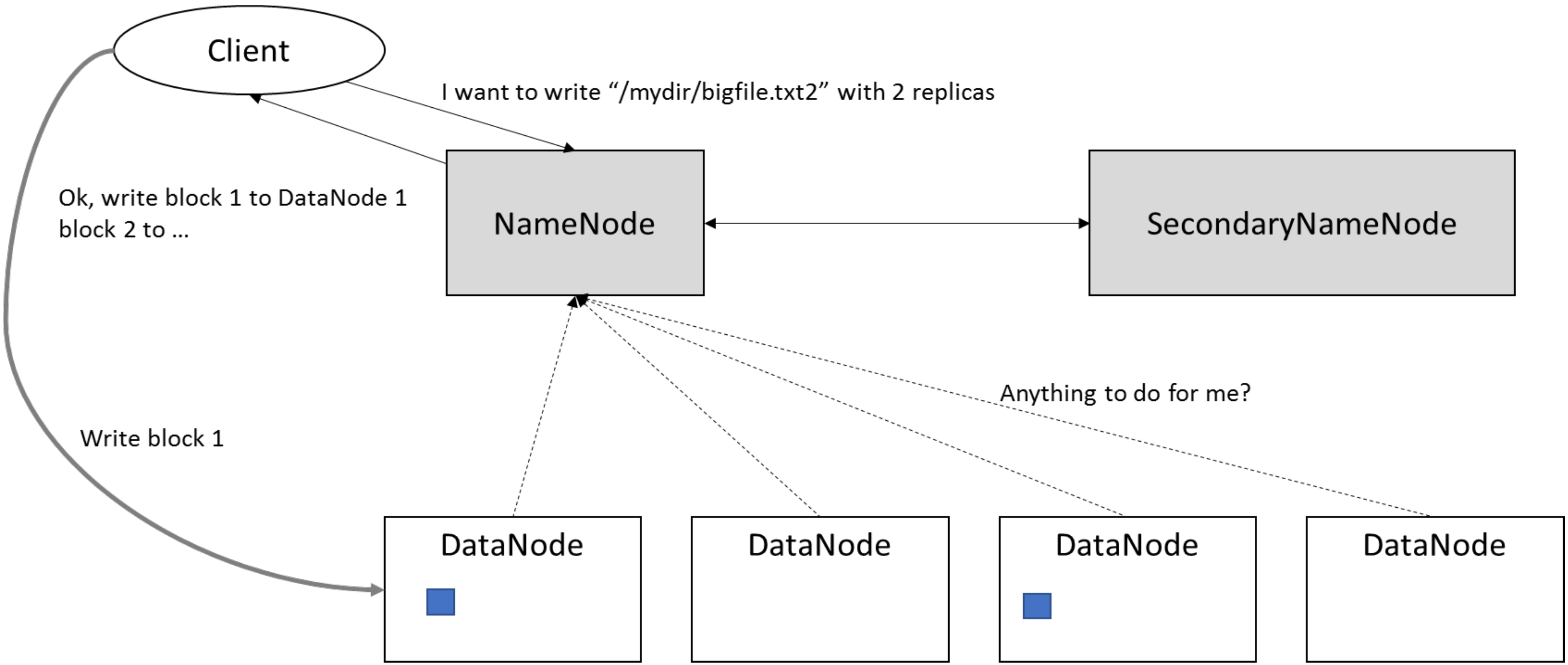
Hadoop Distributed File System (HDFS)



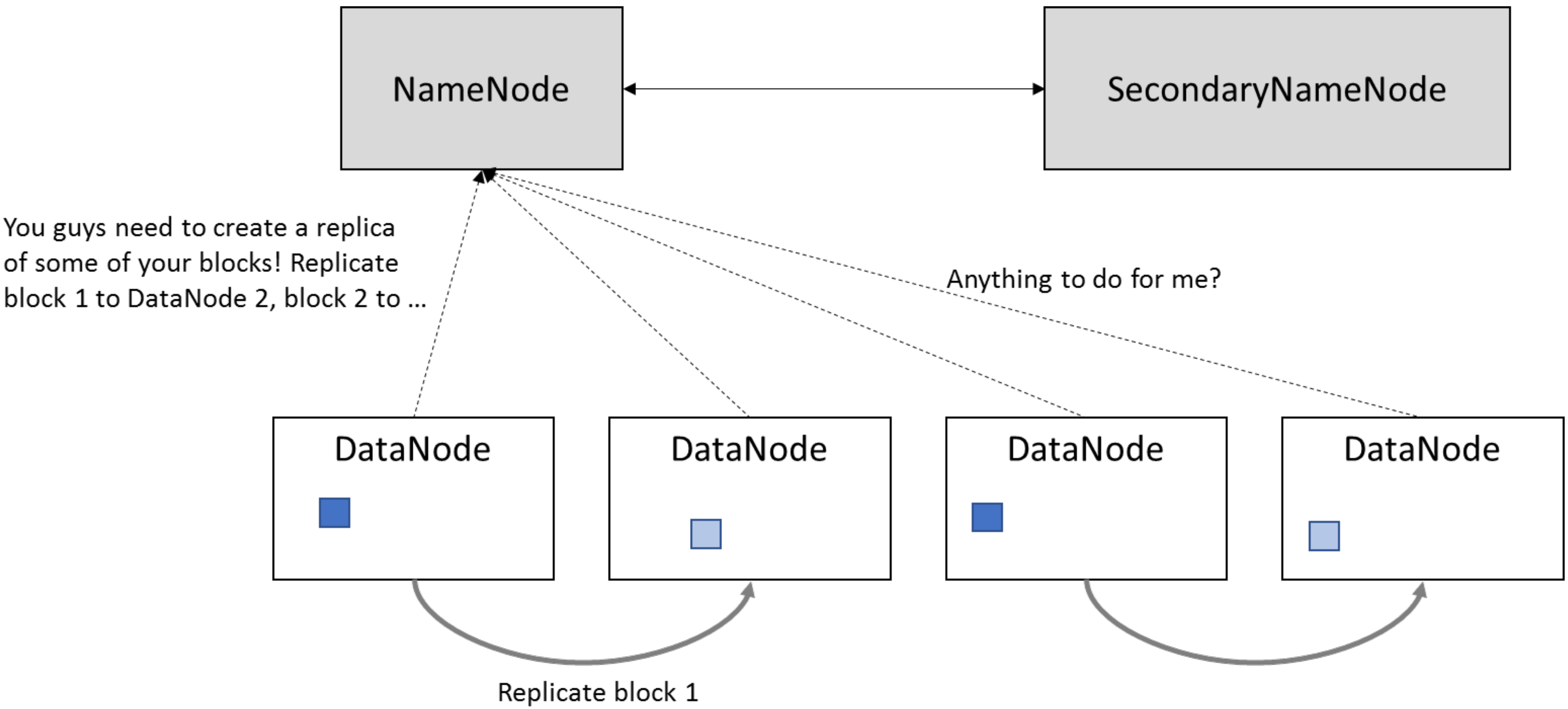
Hadoop Distributed File System (HDFS)



Hadoop Distributed File System (HDFS)



Hadoop Distributed File System (HDFS)



Hadoop Distributed File System (HDFS)

- HDFS provides a native Java API to allow for writing Java programs that can interface with HDFS

```
String filePath = "/data/all_my_customers.csv";
Configuration config = new Configuration();
org.apache.hadoop.fs.FileSystem hdfs =
org.apache.hadoop.fs.FileSystem.get(config);
org.apache.hadoop.fs.Path path = new
org.apache.hadoop.fs.Path(filePath);
org.apache.hadoop.fs.FSDataInputStream inputStream =
hdfs.open(path);
byte[] received = new byte[inputStream.available()];
inputStream.readFully(received);
```

Hadoop Distributed File System (HDFS)

```
// ...
org.apache.hadoop.fs.FSDataInputStream inputStream = hdfs.open(path);
byte[] buffer=new byte[1024]; // Only handle 1KB at once
int bytesRead;
while ((bytesRead = in.read(buffer)) > 0) {
    // Do something with the buffered block here
}
```

Hadoop Distributed File System (HDFS)

`hadoop fs -mkdir mydir`	Create a directory on HDFS
`hadoop fs -ls`	List files and directories on HDFS
`hadoop fs -cat myfile`	View a file’s content
`hadoop fs -du`	Check disk space usage on HDFS
`hadoop fs -expunge`	Empty trash on HDFS
`hadoop fs -chgrp mygroup myfile`	Change group membership of a file on HDFS
`hadoop fs -chown myuser myfile`	Change file ownership of a file on HDFS
`hadoop fs -rm myfile`	Delete a file on HDFS
`hadoop fs -touchz myfile`	Create an empty file on HDFS
`hadoop fs -stat myfile`	Check the status of a file (file size, owner, etc.)
`hadoop fs -test -e myfile`	Check if a file exists on HDFS
`hadoop fs -test -z myfile`	Check if a file is empty on HDFS
`hadoop fs -test -d myfile`	Check if myfile is a directory on HDFS

MapReduce

- Programming paradigm made popular by Google and subsequently implemented by Apache Hadoop
- Focus on scalability and fault tolerance
- A map–reduce pipeline starts from a series of values and maps each value to an output using a given mapper function

MapReduce

- High-level Python example
  - **Map**

```
>>> numbers = [1,2,3,4,5]
>>> numbers.map(lambda x : x * x) # Map a
function to our list
[1,4,9,16,25]
```

  - **Reduce**

```
>>> numbers.reduce(lambda x : sum(x) + 1)
# Reduce a list using given function
16
```

MapReduce

- A MapReduce pipeline in Hadoop starts from a list of key–value pairs, and maps each pair to one or more output elements
- The output elements are also key–value pairs
- Next, the output entries are grouped so all output entries belonging to the same key are assigned to the same worker (e.g., physical machine)
- These workers then apply the reduce function to each group, producing a new list of key–value pairs
- The resulting, final outputs can then be sorted

MapReduce

- Reduce-workers can already get started on their work even although not all mapping operations have finished yet
- Implications:
  - The reduce function should output the same key–value structure as the one emitted by the map function
  - The reduce function itself should be built in such a way that it provides correct results, even if called multiple times

MapReduce

- In Hadoop, MapReduce tasks are written in Java
- To run a MapReduce task, a Java program is packaged as a JAR archive and launched as:
  `hadoop jar myprogram.jar TheClassToRun [args...]`

MapReduce

- Example: Java program to count the appearances of a word in a file

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {
  // Following fragments will be added here
}
```

MapReduce

- Define mapper function as a class extending the built-in `Mapper<KeyIn, ValueIn, KeyOut, ValueOut>` class, indicating which type of key–value input pair we expect and which type of key–value output pair our mapper will emit

MapReduce

```
public static class MyMapper extends Mapper<Object, Text, Text, IntWritable> {
		// Our input key is not important here, so it can just be any generic object.  Our input value is a piece of text (a line)
		// Our output key will also be a piece of text (a word). Our output value will be an integer.

		public void map(Object key, Text value, Context context)  throws IOException, InterruptedException {
			// Take the value, get its contents, convert to lowercase,
			// and remove every character except for spaces and a-z values:
			String document = value.toString().toLowerCase().replaceAll("[^a-z\\s]", "");
			// Split the line up in an array of words
			String[] words = document.split(" ");

			// For each word...
			for (String word : words) {
				// "context" is used to emit output values
				// Note that we cannot emit standard Java types such as int, String, etc. Instead, we need to use
				// a org.apache.hadoop.io.* class such as Text (for string values) and IntWritable (for integers)

				Text textWord = new Text(word);
				IntWritable one = new IntWritable(1);

				// ... simply emit a (word, 1) key-value pair:
				context.write(textWord, one);
			}
		}
}
```

MapReduce

Input key–value pairs	
Key <Object>	Value <Text>
0	This is the first line
23	And this is the second line, and this is all

Mapped key–value pairs	
Key <Text>	Value <IntWritable>
this	1
is	1
the	1
first	1
line	1
and	1
…	…

MapReduce

- Reducer function is specified as a class extending the built-in `Reducer<KeyIn, ValueIn, KeyOut, ValueOut>` class

MapReduce

```
public static class MyReducer extends Reducer<Text, IntWritable, Text, IntWritable> {

        public void reduce(Text key, Iterable<IntWritable> values, Context context)
                        throws IOException, InterruptedException {
                int sum = 0;
                IntWritable result = new IntWritable();
                // Summarise the values so far...
                for (IntWritable val : values) {
                        sum += val.get();
                }
                result.set(sum);
                // ... and output a new (word, sum) pair
                context.write(key, result);
        }
}
```

MapReduce

Mapped key–value pairs	
Key <Text>	Value <IntWritable>
this	1
is	1
the	1
first	1
line	1
and	1
this	1
is	1

Mapped key–value pairs for “this”	
Key <Text>	Value <IntWritable>
this	1
this	1

Reduced key–value pairs for “this”	
Key <Text>	Value <IntWritable>
this	1 + 1 = 2

MapReduce

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();

    // Set up a MapReduce job with a sensible short name:
    Job job = Job.getInstance(conf, "wordcount");

    // Tell Hadoop which JAR it needs to distribute
    // to the workers.
    // We can easily set this using setJarByClass
    job.setJarByClass(WordCount.class);

    job.setMapperClass(MyMapper.class);
    job.setReducerClass(MyReducer.class);

    // What does the output look like?
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    // Our program expects two arguments, the first one is the input
    // file on HDFS
    // Tell Hadoop our input is in the form of TextInputFormat
    // (Every line in the file will become value to be mapped)
    TextInputFormat.addInputPath(job, new Path(args[0]));

    // The second argument is the output directory on
    // HDFS
    Path outputDir = new Path(args[1]);
    // Tell Hadoop what our desired output structure is
    FileOutputFormat.setOutputPath(job, outputDir);

    // Delete the output directory if it exists
    FileSystem fs = FileSystem.get(conf);
    fs.delete(outputDir, true);

    // Stop after our job has completed
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

MapReduce

```
hadoop jar wordcount.jar WordCount /users/me/dataset.txt /users/me/output/
```

Command Prompt

```
[root@sandbox Desktop]$ hadoop jar wordcount.jar WordCount /users/me/dataset.txt /users/me/output/
17/03/16 15:14:23 INFO impl.TimelineClientImpl: Timeline service address: http://sandbox.hortonworks.com:8188/ws/v1/timeline/
17/03/16 15:14:23 INFO client.RMProxy: Connecting to ResourceManager at sandbox.hortonworks.com/172.17.0.2:8050
17/03/16 15:14:23 INFO client.AHSProxy: Connecting to Application History server at sandbox.hortonworks.com/172.17.0.2:10200
17/03/16 15:14:23 WARN mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed.
                       Implement the Tool interface and execute your application with ToolRunner to remedy this.
17/03/16 15:14:23 INFO input.FileInputFormat: Total input paths to process : 1
17/03/16 15:14:23 INFO lzo.GPLNativeCodeLoader: Loaded native gpl library
17/03/16 15:14:23 INFO lzo.LzoCodec: Successfully loaded & initialized native-lzo library
                       [hadoop-lzo rev 7a4b57bedce694048432dd5bf5b90a6c8ccdba80]
17/03/16 15:14:24 INFO mapreduce.JobSubmitter: number of splits:1
17/03/16 15:14:24 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1489673597052_0001
17/03/16 15:14:24 INFO impl.YarnClientImpl: Submitted application application_1489673597052_0001
17/03/16 15:14:25 INFO mapreduce.Job: The url to track the job: http://sandbox.hortonworks.com:8088/proxy/application_1489673597052_0001/
17/03/16 15:14:25 INFO mapreduce.Job: Running job: job_1489673597052_0001
17/03/16 15:14:42 INFO mapreduce.Job: Job job_1489673597052_0001 running in uber mode : false
17/03/16 15:14:42 INFO mapreduce.Job: map 0% reduce 0%
17/03/16 15:14:49 INFO mapreduce.Job: map 100% reduce 0%
17/03/16 15:14:57 INFO mapreduce.Job: map 100% reduce 100%
17/03/16 15:14:57 INFO mapreduce.Job: Job job_1489673597052_0001 completed successfully
17/03/16 15:14:57 INFO mapreduce.Job: Counters: 49
        File System Counters
                FILE: Number of bytes read=5269
                FILE: Number of bytes written=298885
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations-0
                HDFS: Number of bytes read-2826
                HDFS: Number of bytes written=2069
                HDPS: Number of read operations=6
```

MapReduce

```
$ hadoop fs -ls /users/me/output
Found 2 items
-rw-r—r-- 1 root    hdfs    0    2017-05-20  15:11         /users/me/output/_SUCCESS
-rw-r—r-- 1 root    hdfs    2069  2017-05-20  15:11        /users/me/output/part-r-00000
```

```
$ hadoop fs -cat /users/me/output/part-r-00000
and       2
first     1
is        3
line      2
second    1
the       2
this      3
```

MapReduce

- MapReduce task can consist of more than mappers and reducers
- Can also include
  - Partitioners
  - Combiners
  - Shufflers
  - Sorters

MapReduce
Partitioner

- condition in processing an input dataset.
- partition phase takes place after the Map phase and before the Reduce phase.
- The number of partitioners is equal to the number of reducers.
  - partitioner will divide the data according to the number of reducers.

MapReduce
Partitioner

- A partitioner partitions the key-value pairs of intermediate Map-outputs.
- partitions data using a user-defined condition,
  - works like a hash function.

MapReduce
Partitioner

- Code for below example can be found at https://www.tutorialspoint.com/map_reduce/map_reduce_partitioner.htm
- Dataset

Id	Name	Age	Gender	Salary

- Task
  - find highest salaried employee by gender in different age groups (for example, below 20, between 21 to 30, above 30).

MapReduce
Partitioner

- Map task:
  - Read the value (record data), which comes as input value from the argument list in a string.
  - Using the split function, separate the gender and store in a string variable.
- Output of map task
  - gender data and the record data value as key-value pairs.

MapReduce
Partitioner

- Partitioner input
  - input key-value paired data output from mapper
  - key = Gender field value in the record.
  - value = Whole record data value of that gender.
  - The whole data of key-value pairs are segmented into three collections of key-value pairs
    - Based on age
- Partitioner output
  - key-value pairs are segmented into three collections of key-value pairs.

MapReduce
Partitioner

- Reducer works individually on each collection.
- Reducer Input – three reducers with different collection of key-value pairs.
  - key = gender field value in the record.
  - value = the whole record data of that gender.

MapReduce
Partitioner

- Output of Reducer
  - set of key-value pair data in three collections of different age groups.
  - max salary from the Male collection in each age group respectively
  - max salary from the Female collection in each age group respectively

MapReduce
Partitioner

- the three collections of key-value pair data are stored in three different files as the output.
- requirements and specifications of these jobs should be specified in the Configurations –
  - Job name
  - Input and Output formats of keys and values
  - Individual classes for Map, Reduce, and Partitioner tasks

MapReduce
Partitioner

- Output will be in three files
  - three partitioners and three Reducers
- Files generated by HDFS
  - Part-00000
  - Part-00001
  - Part-00002

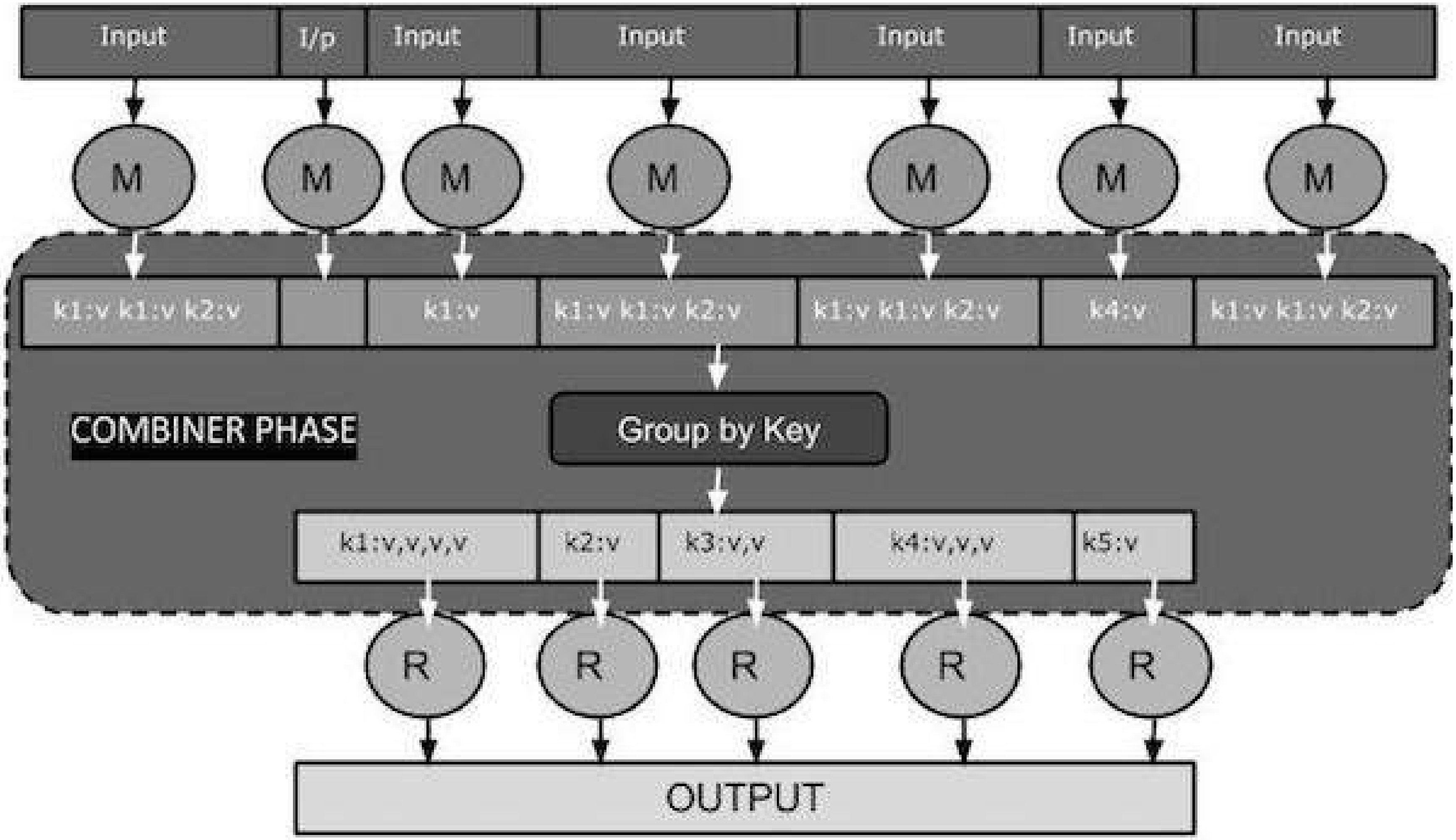
MapReduce
Combiners

- Code can be found at

  https://www.tutorialspoint.com/map_reduce/map_reduce_combiners.htm
- optional class
  – accepts inputs from Map class and passes output key-value pairs to Reducer class.
- summarizes map output records with the same key
- output (key-value collection) of combiner sent over network to the actual Reducer task as input

MapReduce
Combiners

- Combiner class used in between Map class and Reduce class to reduce volume of data transfer between Map and Reduce.
  - Usually, output of map task is large and data transferred to reduce task is high.


Input
I/p
Input
Input
Input
Input
Input
M
M
M
M
M
M
M
k1:v k1:v k2:v
k1:v
k1:v k1:v k2:v
k1:v k1:v k2:v
k4:v
k1:v k1:v k2:v
COMBINER PHASE
Group by Key
k1:v,v,v,v
k2:v
k3:v,v
k4:v,v,v
k5:v
R
R
R
R
R
OUTPUT

MapReduce
Combiners

- implements Reducer interface's reduce() method.
- operates on each map output key.
  - must have same output key-value types as Reducer class
- can produce summary information from a large dataset because it replaces original Map output
- optional - helps segregating data into multiple groups for Reduce phase

MapReduce
Combiners

- **Input text file:**

What do you mean by Object

What do you know about Java

What is Java Virtual Machine

How Java enabled High Performance

- **Key value pairs**

<1, What do you mean by Object>

<2, What do you know about Java>

<3, What is Java Virtual Machine>

<4, How Java enabled High Performance>

MapReduce
Combiners

- Key-value pairs input to Map phase
- Output of Map phase

<What,1> <do,1> <you,1> <mean,1> <by,1> <Object,1>

<What,1> <do,1> <you,1> <know,1> <about,1> <Java,1>

<What,1> <is,1> <Java,1> <Virtual,1> <Machine,1>

<How,1> <Java,1> <enabled,1> <High,1> <Performance,1>

- This is input to combiner

MapReduce
Combiners

- Output of combiner

<What,1,1,1> <do,1,1> <you,1,1> <mean,1> <by,1> <Object,1>

<know,1> <about,1> <Java,1,1,1>

<is,1> <Virtual,1> <Machine,1>

<How,1> <enabled,1> <High,1> <Performance,1>

- This is input to reducer

MapReduce
Combiners

- Output from Reducer

<What,3> <do,2> <you,2> <mean,1> <by,1> <Object,1>

<know,1> <about,1> <Java,3>

<is,1> <Virtual,1> <Machine,1>

<How,1> <enabled,1> <High,1> <Performance,1>

MapReduce

- Constructing MapReduce programs requires a certain skillset in terms of programming
- Tradeoffs in terms of speed, memory consumption, and scalability

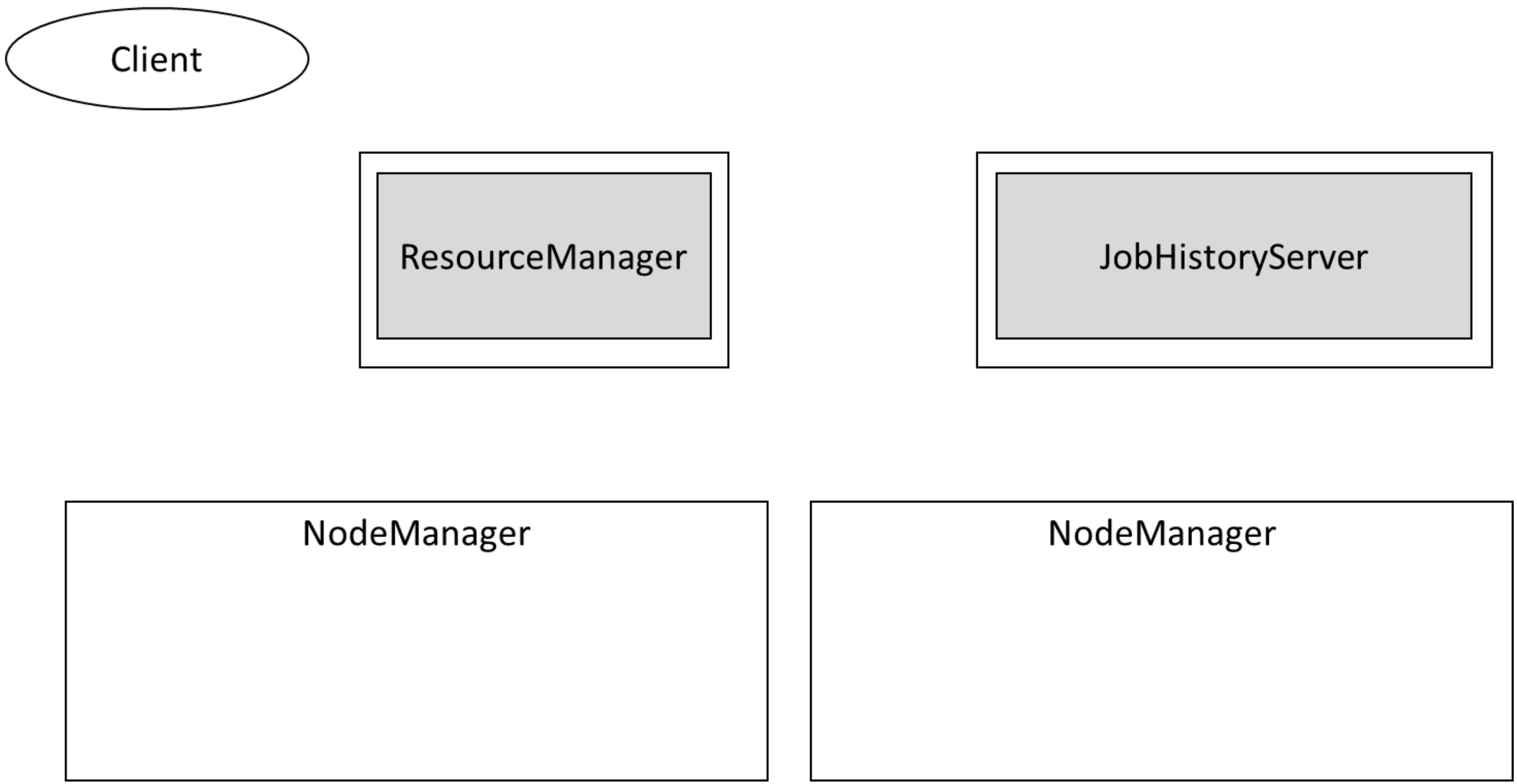
Yet Another Resource Negotiator (YARN)

- Yet Another Resource Negotiator (YARN) distributes a MapReduce program across different nodes and takes care of coordination

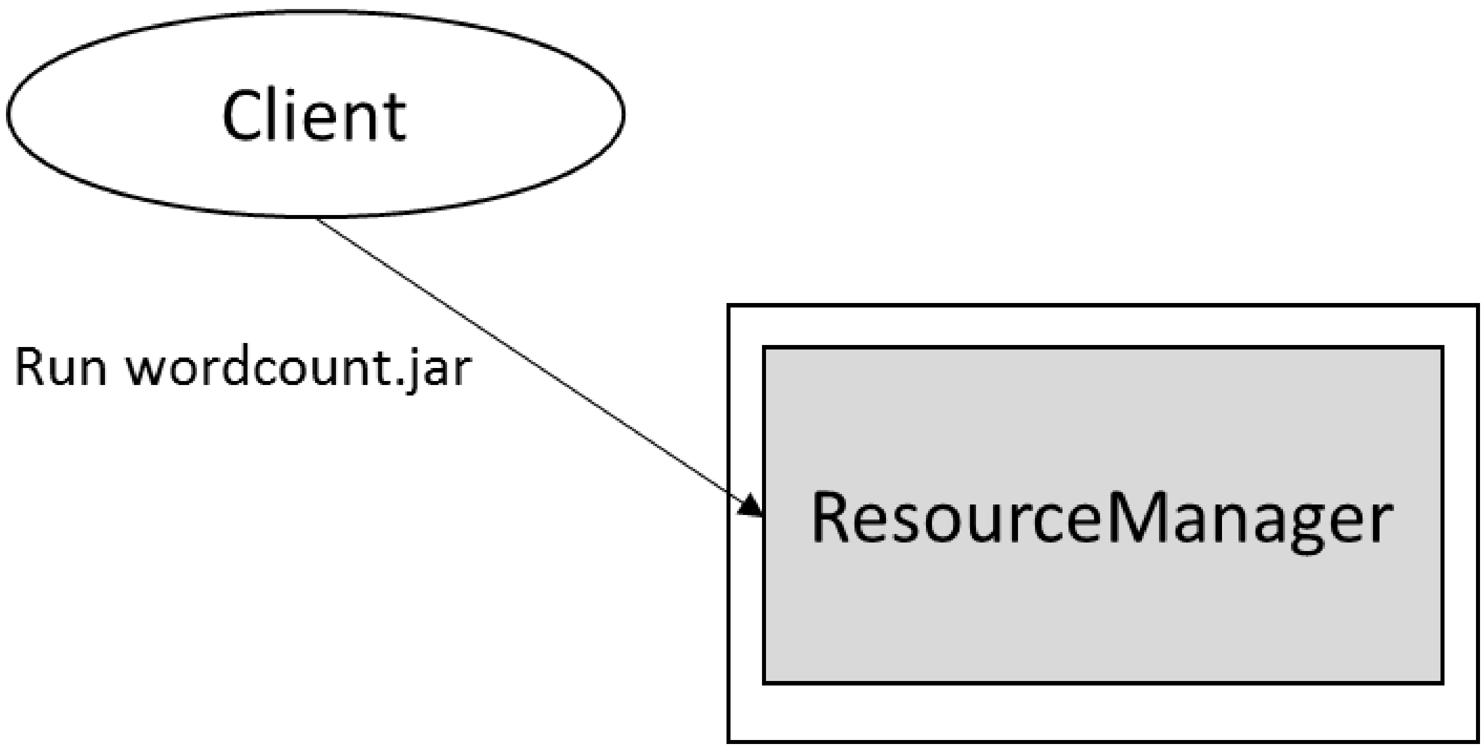
Yet Another Resource Negotiator (YARN)

- Three important services
  - ResourceManager: a global YARN service that receives and runs applications (e.g., a MapReduce job) on the cluster
  - JobHistoryServer: keeps a log of all finished jobs
  - NodeManager: responsible for overseeing resource consumption on a node
    - ApplicationMaster - responsible for the execution of a single application.
      - Asks for containers from the Resource Manage) and executes specific programs (e.g., the main of a Java class) on the obtained containers

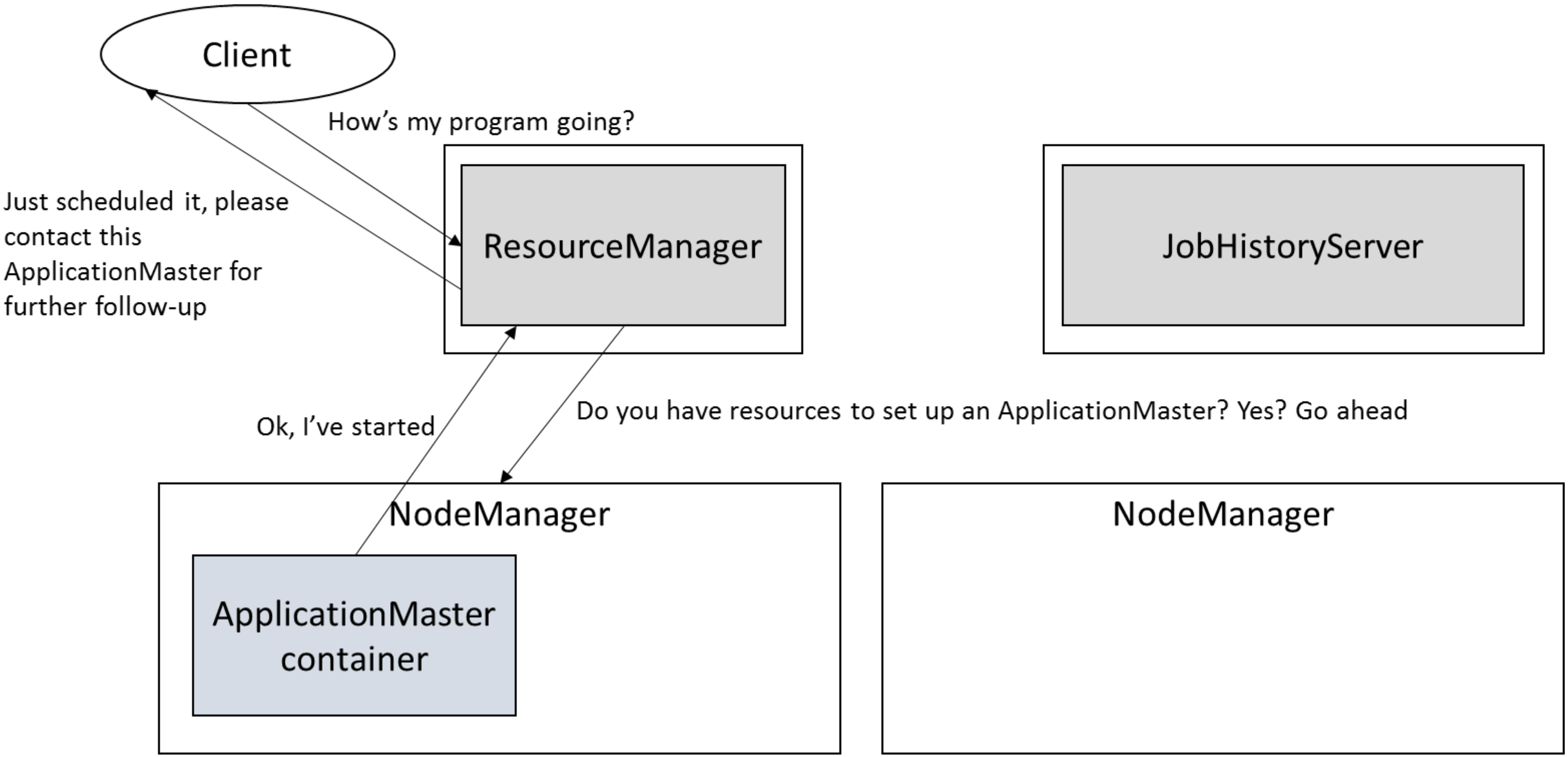
Yet Another Resource Negotiator (YARN)



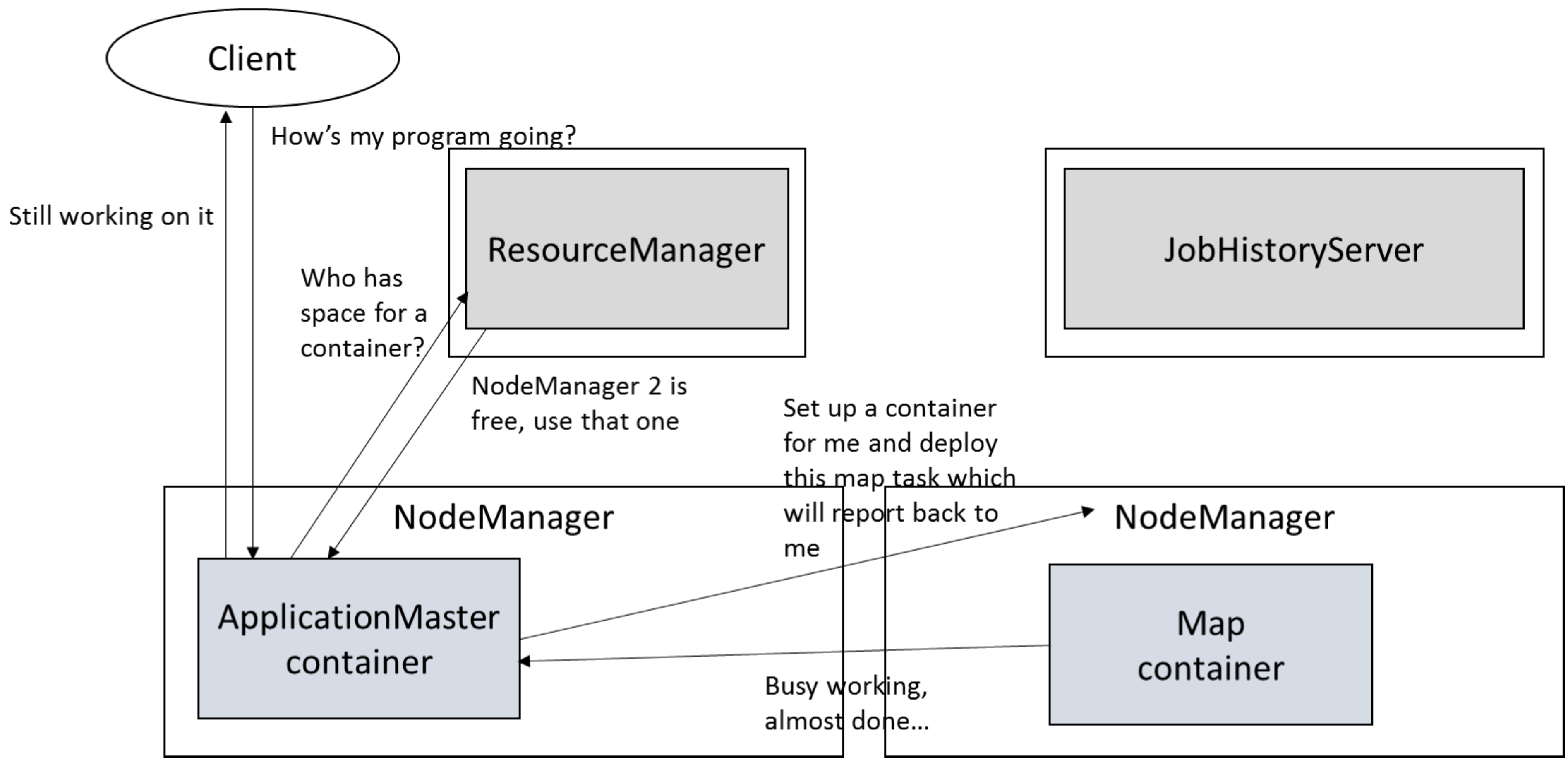
Yet Another Resource Negotiator (YARN)



Yet Another Resource Negotiator (YARN)



Yet Another Resource Negotiator (YARN)



Yet Another Resource Negotiator (YARN)

- Complex setup
- Allows running programs and applications other than MapReduce

SQL on Hadoop

- MapReduce very complex when compared to SQL
- Need for a more database-like setup on top of Hadoop

SQL on Hadoop

- HBase
- Pig
- Hive

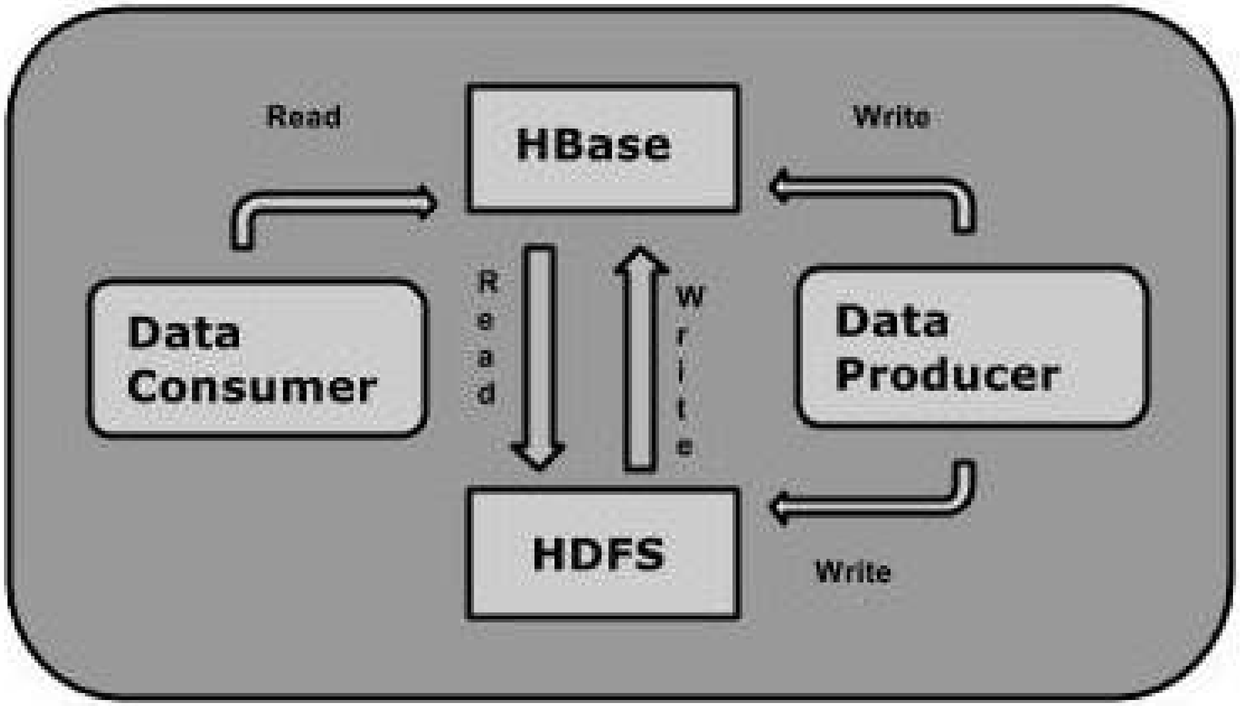
HBase

- Hadoop - batch processing
  - data accessed in a sequential manner.
- Need to access any point of data in a single unit of time (random access).

HBase

- distributed column-oriented database built on top of Hadoop file system.
- horizontally scalable.
- similar to Google's big table - to provide quick random access to huge amounts of structured data.
- provides random real-time read/write access to data in the Hadoop File System.
- Structured and semi-structured data

HBase


Read
HBase
Write
Data Consumer
Read
Write
Data Producer
HDFS
Write

HBase

HDFS	HBase
distributed file system - for storing large files	database built on top of the HDFS
does not support fast individual record lookups	fast lookups
high latency batch processing	low latency access to single rows from billions of records (Random access)
sequential access of data	uses Hash tables and provides random access and stores data in indexed HDFS files for faster lookups

HBase

- NoSQL-like data storage platform
  - No typed columns, triggers, advanced query capabilities, etc.
- Offers a simplified structure and query language in a way that is highly scalable and can tackle large volumes

HBase

- Similar to RDBMSs, HBase organizes data in tables with rows and columns
- HBase table consists of multiple rows
- A row consists of a row key and one or more columns with values associated with them
- Rows in a table are sorted alphabetically by the row key

HBase

- Each column in HBase is denoted by a column family and qualifier (separated by a colon, “:”)
- A column family physically co-locates a set of columns and their values
- Every row has the same column families, but not all column families need to have a value per row
- Each cell in a table is hence defined by a combination of the row key, column family and column qualifier, and a timestamp

HBase

- Example: HBase table to store and query users
- The row key will be the user id
- column families:qualifiers
  - name:first
  - name:last
  - email (without a qualifier)

HBase

```
hbase(main):001:0> create 'users', 'name', 'email'
0 row(s) in 2.8350 seconds

=> Hbase::Table - users



hbase(main):002:0> describe 'users'
Table users is ENABLED
users
COLUMN FAMILIES DESCRIPTION
{NAME => 'email', BLOOMFILTER => 'ROW', VERSIONS => '1', IN_MEMORY => 'false', K
EEP_DELETED_CELLS => 'FALSE', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', C
OMPRESSION => 'NONE', MIN_VERSIONS => '0', BLOCKCACHE => 'true', BLOCKSIZE => '6
5536', REPLICATION_SCOPE => '0'}
{NAME => 'name', BLOOMFILTER => 'ROW', VERSIONS => '1', IN_MEMORY => 'false', KE
EP_DELETED_CELLS => 'FALSE', DATA_BLOCK_ENCODING => 'NONE', TTL => 'FOREVER', CO
MPRESSION => 'NONE', MIN_VERSIONS => '0', BLOCKCACHE => 'true', BLOCKSIZE => '65
536', REPLICATION_SCOPE => '0'}
2 row(s) in 0.3250 seconds
```

HBase

```
hbase(main):003:0> list 'users'
TABLE
users
1 row(s) in 0.0410 seconds

=> ["users"]
```

HBase

```
hbase(main):006:0> put 'users', 'seppe', 'name:first', 'Seppe'
0 row(s) in 0.0200 seconds

hbase(main):007:0> put 'users', 'seppe', 'name:last', 'vanden Broucke'
0 row(s) in 0.0330 seconds

hbase(main):008:0> put 'users', 'seppe', 'email', 'seppe.vandenbroucke@kuleuven'
0 row(s) in 0.0570 seconds
```

```
hbase(main):009:0> scan 'users'
ROW                     COLUMN+CELL
 seppe                  column=email:, timestamp=1495293082872, value=seppe.vanden
                        broucke@kuleuven.be
 seppe                  column=name:first, timestamp=1495293050816, value=Seppe
 seppe                  column=name:firstt, timestamp=1495293047100, value=Seppe
 seppe                  column=name:last, timestamp=1495293067245, value=vanden Broucke

1 row(s) in 0.1170 seconds
```

HBase

```
hbase(main):011:0> get 'users', 'seppe'
COLUMN                        CELL
 email:                       timestamp=1495293082872,
value=seppe.vandenbroucke@kuleuven.be
 name:first                   timestamp=1495293050816, value=Seppe
 name:firstt                  timestamp=1495293047100, value=Seppe
 name:last                    timestamp=1495293067245, value=vanden Broucke
4 row(s) in 0.1250 seconds
```

```
hbase(main):018:0> put 'users', 'seppe', 'email', 'seppe@kuleuven.be'
0 row(s) in 0.0240 seconds
```

```
hbase(main):019:0> get 'users', 'seppe', 'email'
COLUMN                        CELL
 email:                       timestamp=1495293303079, value=seppe@kuleuven.be
1 row(s) in 0.0330 seconds
```

HBase

- HBase's query facilities are very limited
- Essentially a key–value, distributed data store with simple get/put operations
- Includes facilities to write MapReduce programs
- Hbase (similar to Hadoop) doesn't perform well on less than five HDFS DataNodes with an additional NameNode
  - Only makes the effort worthwhile when you can invest in, set up, and maintain at least 6–10 nodes

Pig

- Yahoo! Developed “Pig” which was made open source as Apache Pig in 2007
- High-level platform for creating programs that run on Hadoop (in Pig Latin), which uses MapReduce underneath
- Somewhat resembles querying facilities of SQL
- procedural language platform used to develop a script for MapReduce operations.

Pig

```
timesheet = LOAD 'timesheet.csv' USING PigStorage(',');
raw_timesheet = FILTER timesheet by $0 > 100;
timesheet_logged = FOREACH raw_timesheet GENERATE $0 AS
driverId, $2 AS hours_logged, $3 AS miles_logged;
grp_logged = GROUP timesheet_logged by driverId;
sum_logged = FOREACH grp_logged GENERATE group as driverId,
SUM(timesheet_logged.hours_logged) as sum_hourslogged,
SUM(timesheet_logged.miles_logged) as sum_mileslogged;
```

Pig

- Some have argued that RDBMSs and SQL are substantially faster than MapReduce – and hence Pig
  - Especially for relatively sized, structured data
- Pig Latin is relatively procedural versus declarative SQL
- No wide adoption

Hive

- Initially developed by Facebook but open-sourced afterwards
- Data warehouse solution offering SQL querying facilities on top of Hadoop
- Converts SQL-like queries to a MapReduce pipeline
- Also offers a JDBC and ODBC interface
- Can run on top of HDFS, as well as other file systems

Hive

- Hive Metastore stores metadata for each table such as its schema and location on HDFS (using an RDBMS)
- Driver service is responsible to receive and handle incoming queries
  - Query is first converted to an abstract syntax tree, which is then converted to a directed acyclic graph representing an execution plan
  - The directed acyclic graph will contain a number of MapReduce stages and tasks
- Optimizer optimizes the directed acyclic graph
- Executer sends MapReduce stages to Hadoop's resource manager (e.g., YARN) and monitors their progress

Hive

- HiveQL does not completely follow the full SQL-92 standard
  - e.g., lacks strong support for indexes, transactions, materialized views, and only has limited subquery support
- Example:

```
SELECT genre, SUM(nrPages) FROM books
GROUP BY genre
```

Hive

- HiveQL also allows querying datasets other than structured tables as long as it is possible to express a statement to extract data out of them in a tabular format
- Example: Word count

```
CREATE TABLE docs (line STRING); --
create a docs table-- load in file
from HDFS to docs table, overwrite
existing data:
```

Hive

```
LOAD DATA INPATH '/users/me/doc.txt' OVERWRITE INTO
TABLE docs;

-- perform word count
SELECT word, count(1) AS count
FROM (
  SELECT explode(split(line, '\s')) AS word FROM docs
) t
GROUP BY t.word
ORDER BY t.word;
```

Hive

- One difference with traditional RDBMS is that Hive does not enforce the schema at the time of loading the data
  - Hive: schema-on-read
  - RDBMS: schema-on-write
- Recent versions of Hive support full ACID transaction management
- Performance and speed of SQL queries still forms the main disadvantage of Hive today
  - Solutions to bypass MapReduce (e.g., Apache Tez, Cloudera Impala, Facebook Presto)

Apache Spark

- Open-source alternative for MapReduce
- New programming paradigm centered on a data structure called the resilient distributed dataset (RDD), which can be distributed across a cluster of machines and is maintained in a fault-tolerant way
- RDDs can enable the construction of iterative programs that have to visit a dataset multiple times, as well as more interactive or exploratory programs
- Many orders of magnitude faster than MapReduce implementations
- Rapidly adopted by many Big Data vendors

Apache Spark

- Similar to Hadoop, Spark works with HDFS and requires a cluster manager (e.g., YARN)
- Key components
  - Spark Core
  - Spark SQL
  - MLib, Spark Streaming, GraphX

Spark Core

- Foundation for all other components
- Provides functionality for task scheduling and a set of basic data transformations that can be used through many programming languages (e.g., Java, Python, Scala, and R)
- RDDs are the primary data abstraction in Spark
  - Designed to support in-memory data storage and operations, distributed across a cluster

Spark Core

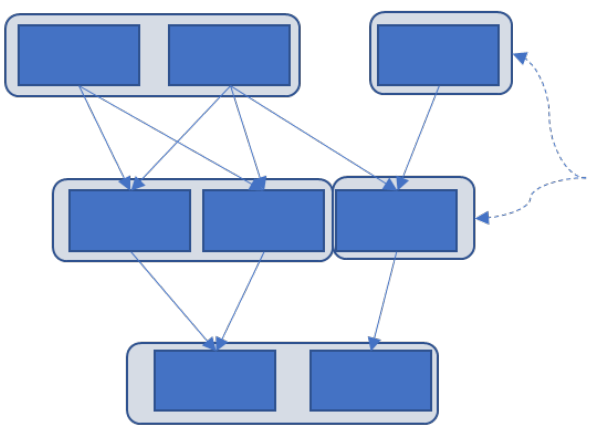
- Once data are loaded into an RDD, two basic types of operations can be performed:
  - Transformation, which creates a new RDD through changing the original one
  - Actions (such as counts), which measure but do not change the original data
- Transformations are lazily evaluated
  - They are not executed until a subsequent action has a need for the result
- RDDs will also be kept as long as possible in memory
- A chain of RDD operations gets compiled by Spark into a directed acyclic graph which is then spread out and calculated over the cluster

Spark Core

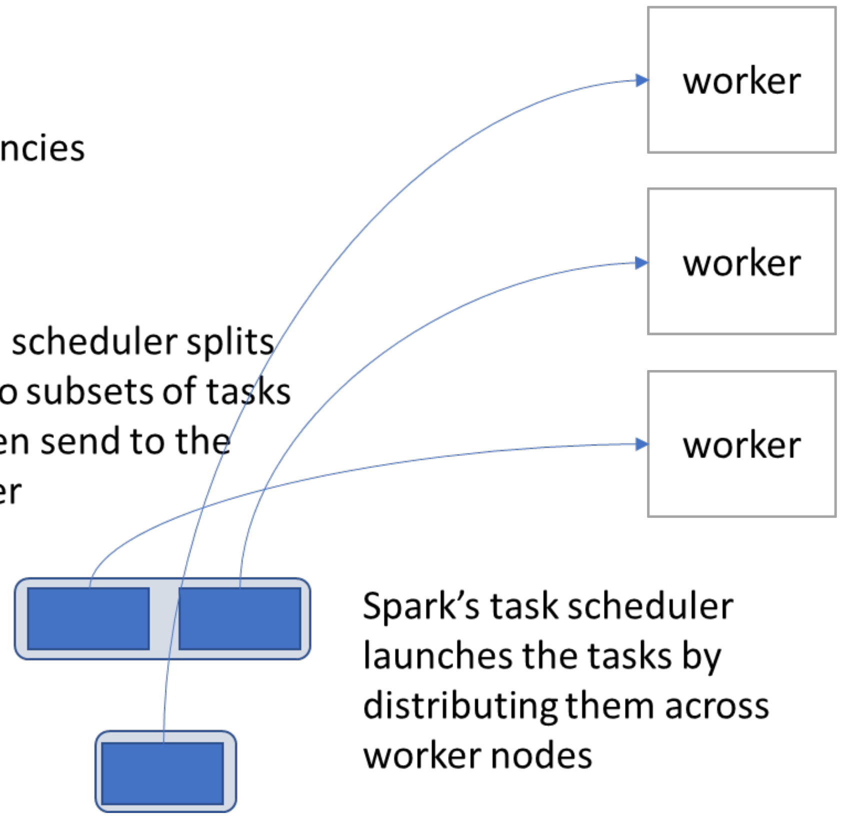
A programmer writes a Spark program using its API:

```
rdd1.join(rdd2).groupBy(…).filter(…)
```

Based on this, Spark builds a directed acyclic graph of operations with their dependencies



Spark's graph scheduler splits the graph into subsets of tasks which are then send to the task scheduler



Spark's task scheduler launches the tasks by distributing them across worker nodes

Spark Core

- Spark's RDD API is relatively easy to work with compared to writing MapReduce programs

```
# Set up connection to the Spark cluster
sconf = SparkConf()
sc = SparkContext(master='', conf=sconf)

# Load in an RDD from a text file, the RDD will represent a collection of
# text strings (one for each line)
text_file = sc.textFile("myfile.txt")

# Count the word occurrences
counts = text_file.flatMap(lambda line: line.split(" ")) \
.map(lambda word: (word, 1)) \
.reduceByKey(lambda a, b: a + b)

print(counts)
```

Spark SQL

- Spark SQL runs on top of Spark Core and introduces another data abstraction called DataFrames
- DataFrames can be created from RDDs by specifying a schema on how to structure the data elements in the RDD, or can be loaded in directly from various sorts of file formats
- Even although DataFrames continue to use RDDs behind the scenes, they represent themselves to the end-user as a collection of data organized into named columns

Spark SQL

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("Spark
example").getOrCreate()

# Create a DataFrame object by reading in a file
df = spark.read.json("people.json")

df.show()
# | age|    name|
# +----+--------+
# |null|   Seppe|
# |  30|Wilfried|
# |  19|    Bart|
# +----+--------+

# DataFrames are structured in columns and rows:
df.printSchema()
# root
# |-- age: long (nullable = true)
# |-- name: string (nullable = true)
```

Spark SQL

```
df.select("name").show()
# +--------+
# |    name|
# +--------+
# |   Seppe|
# |Wilfried|
# |    Bart|
# +--------+

# SQL-like operations can now easily be expressed:
df.select(df['name'], df['age'] + 1).show()
# +--------+---------+
# |    name|(age + 1)|
# +--------+---------+
# |   Seppe|     null|
# |Wilfried|       31|
# |    Bart|       20|
# +--------+---------+
```

Spark SQL

```
df.filter(df['age'] > 21).show()
# +---+--------+
# |age|    name|
# +---+--------+
# | 30|Wilfried|
# +---+--------+

df.groupBy("age").count().show()
# +----+-----+
# | age|count|
# +----+-----+
# |  19|    1|
# |null|    1|
# |  30|    1|
# +----+-----+
```

Spark SQL

- Spark implements a full SQL query engine that can convert SQL statements to a series of RDD transformations and actions

```
# Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")
```

```
sqlDF = spark.sql("SELECT * FROM people WHERE age > 21")
sqlDF.show()

# +---+--------+
# |age|    name|
# +---+--------+
# | 30|Wilfried|
# +---+--------+
```

MLlib, Spark Streaming, and GraphX

- MLlib is Spark's machine learning library
  - Offers classification, regression, clustering, and recommender system algorithms
- MLlib was originally built directly on top of the RDD abstraction
- New MLlib version works directly with SparkSQL's DataFrames based API

MLlib, Spark Streaming, and GraphX

- Spark Streaming leverages Spark Core and its fast scheduling engine to perform streaming analytics
- Spark Streaming provides another high-level concept called the DStream (discretized stream), which represents a continuous stream of data
  - Internally a DStream is represented as a sequence of RDD fragments
- DStreams provide windowed computations, which allow applying transformations over a sliding window of data

MLlib, Spark Streaming, and GraphX

- Example: word counting

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext
sc = SparkContext("local[2]", "StreamingWordCount")
ssc = StreamingContext(sc, 1)

# Create a DStream that will connect to server.mycorp.com:9999 as a source
lines = ssc.socketTextStream("server.mycorp.com ", 9999)

# Split each line into words
words = lines.flatMap(lambda line: line.split(" "))

# Count each word in each batch
pairs = words.map(lambda word: (word, 1))
wordCounts = pairs.reduceByKey(lambda x, y: x + y)

# Print out first ten elements of each RDD generated in the wordCounts Dstream
wordCounts.pprint()

# Start the computation
ssc.start()
ssc.awaitTermination()
```

MLlib, Spark Streaming, and GraphX

- GraphX is Spark’s component implementing programming abstractions to deal with graph-based structures, again based on the RDD abstraction.
- GraphX comes with a set of fundamental operators and algorithms (such as PageRank) to work with graphs and simplify graph analytics tasks

Conclusion

- The 5 Vs of Big Data
- Hadoop
- SQL on Hadoop
- Apache Spark