POLITECNICO
MILANO 1863

SYSTEMS AND METHODS FOR BIG AND UNSTRUCTURED DATA

HADOOP & HDFS

Marco Brambilla

marco.brambilla@polimi.it

@marcobrambi

Agenda

Overview of Hadoop

HDFS

- Introduction
- Architecture
- Commands

Map Reduce

Hadoop

What is Hadoop?

Software platform to easily process vast amounts of data. It includes:

- – MapReduce – offline computing engine
- – HDFS – Hadoop distributed file system
- – Hbase – online data access

… and much more!

Features:

Scalable: It can reliably store and process petabytes.

Economical: It distributes the data and processing across clusters of commonly available computers (in thousands).

Efficient: By distributing the data, it can process it in parallel **on the nodes where the data is located.**

Reliable: It automatically maintains multiple copies of data and automatically redeploys computing tasks based on failures.

What does it do?

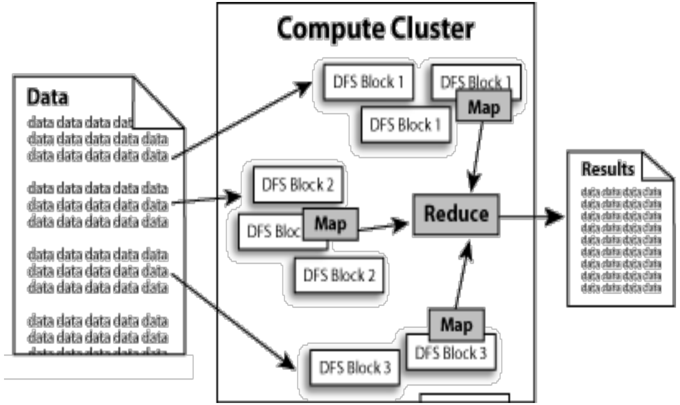
Hadoop implements Google's MapReduce, using HDFS

MapReduce divides applications into many small blocks of work.

HDFS creates multiple replicas of data blocks for reliability, placing them on compute nodes around the cluster.

MapReduce can then process the data where it is located.

Hadoop target is to run on clusters of the order of thousands of nodes.



Hadoop: Assumptions

Hardware *will* fail.

Processing will be run in batches: High throughput as opposed to low latency.

Applications that run on HDFS have large data sets. A typical file in HDFS is gigabytes to terabytes in size.

It should provide high aggregate data bandwidth and scale to hundreds of nodes in a single cluster.

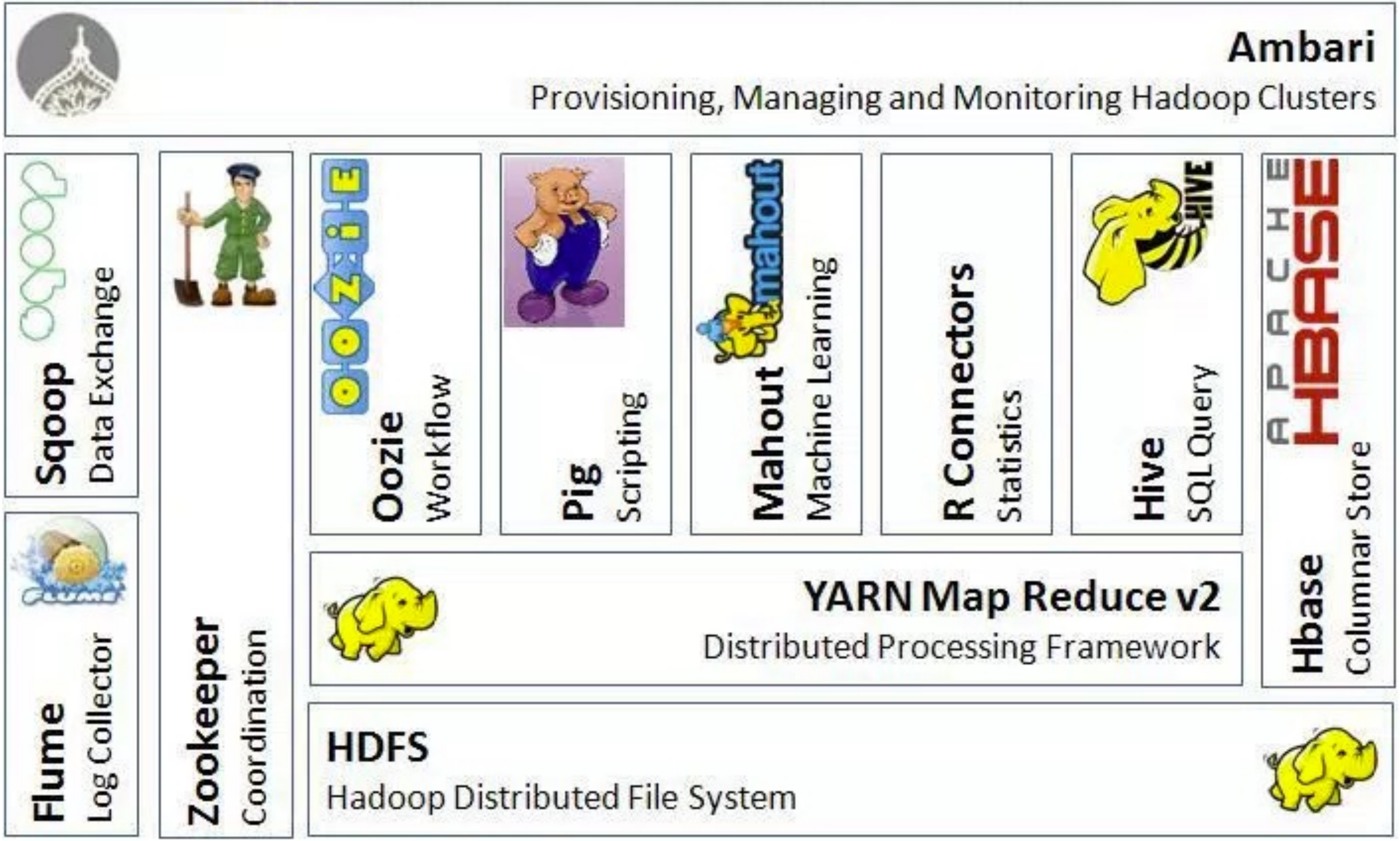
Applications need a **write-once-read-many** access model.

Moving Computation is Cheaper than Moving Data.

Hadoop Components

– Apache Hadoop
– Apache Hive
– Apache Pig
– Apache HBase
– Apache Zookeeper
– Flume, Hue, Oozie, and Sqoop

Apache Hadoop Ecosystem



RDBMS vs. Hadoop

	RDBMS	Hadoop
Data size	Gigabytes	Petabytes
Access	Interactive & Batch	Batch
Updates	Read & write many times	Write once, read many times
Integrity	High	Low
Scaling	Non Linear	Linear
Data representation	Structured	Unstructured, semi-structured

Marco Brambilla.

Hadoop Distributions

Cloudera Distribution for Hadoop (CDH)
Pre-built VMs with most of Cloudera products (Hadoop, etc)

MapR Distribution
MapR Sandbox VM

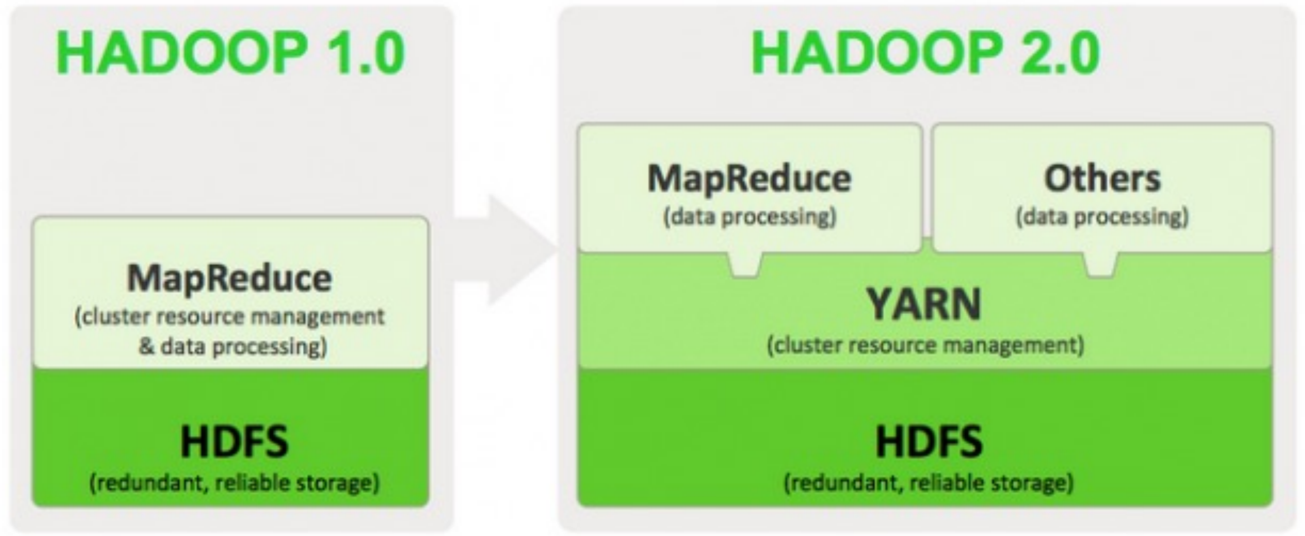
Hortonworks Data Platform (HDP)
Hortonworks Sandbox VM

Oracle Big Data Appliance
VM with Hadoop, Oracle and lots of other tools

Hadoop 1.0 Limitations

- Scalability
  - Maximum Cluster Size – 4000 Nodes
  - Maximum Concurrent Tasks – 40000
  - Coarse synchronization in Job Tracker
- Single point of failure
  - Failure kills all queued and running jobs
  - Jobs need to be resubmitted by users
- Restart is very tricky due to complex state

Hadoop 1.0 → 2.0 (Yarn!)



Marco Brambilla.

HDFS

HDFS Overview

Responsible for storing data on the cluster

Data files are split into blocks and distributed across the nodes in the cluster

Each block is replicated multiple times

HDFS Basic Concepts

HDFS is a file system written in Java based on the Google's GFS

Provides redundant storage for massive amounts of data

HDFS Basic Concepts

HDFS works best with a smaller number of large files

- Millions as opposed to billions of files
- Typically 100MB or more per file

Files in HDFS are write once

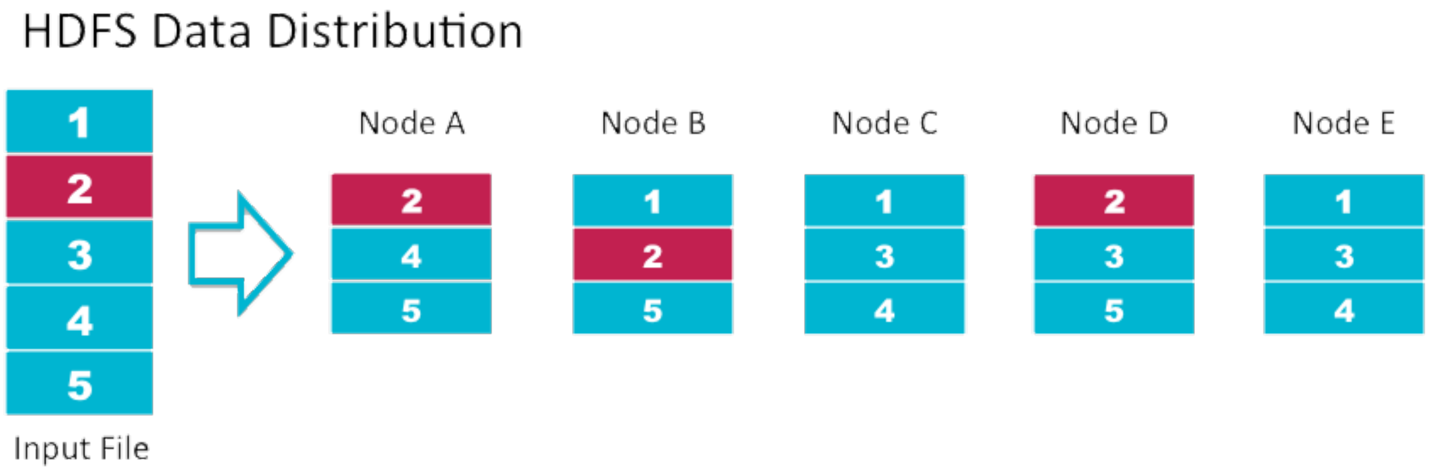
Optimized for streaming reads of large files and not random reads

How are Files Stored

- Files are split into blocks
- Blocks are split across many machines at load time
    - Different blocks from the same file will be stored on different machines
- Blocks are replicated across multiple machines
- The NameNode keeps track of which blocks make up a file and where they are stored

Data Replication

Default replication is 3-fold



Marco Brambilla.

Goals of HDFS

- Very Large Distributed File System
  - 10K nodes, 100 million files, 10PB
- Assumes Commodity Hardware
  - Files are replicated to handle hardware failure
  - Detect failures and recover from them
- Optimized for Batch Processing
  - Data locations exposed so that computations can move to where data resides
  - Provides very high aggregate bandwidth

Distributed File System

Single Namespace for entire cluster

Data Coherency

- **Write-once-read-many** access model
- Client can only append to existing files

Files are broken up into blocks

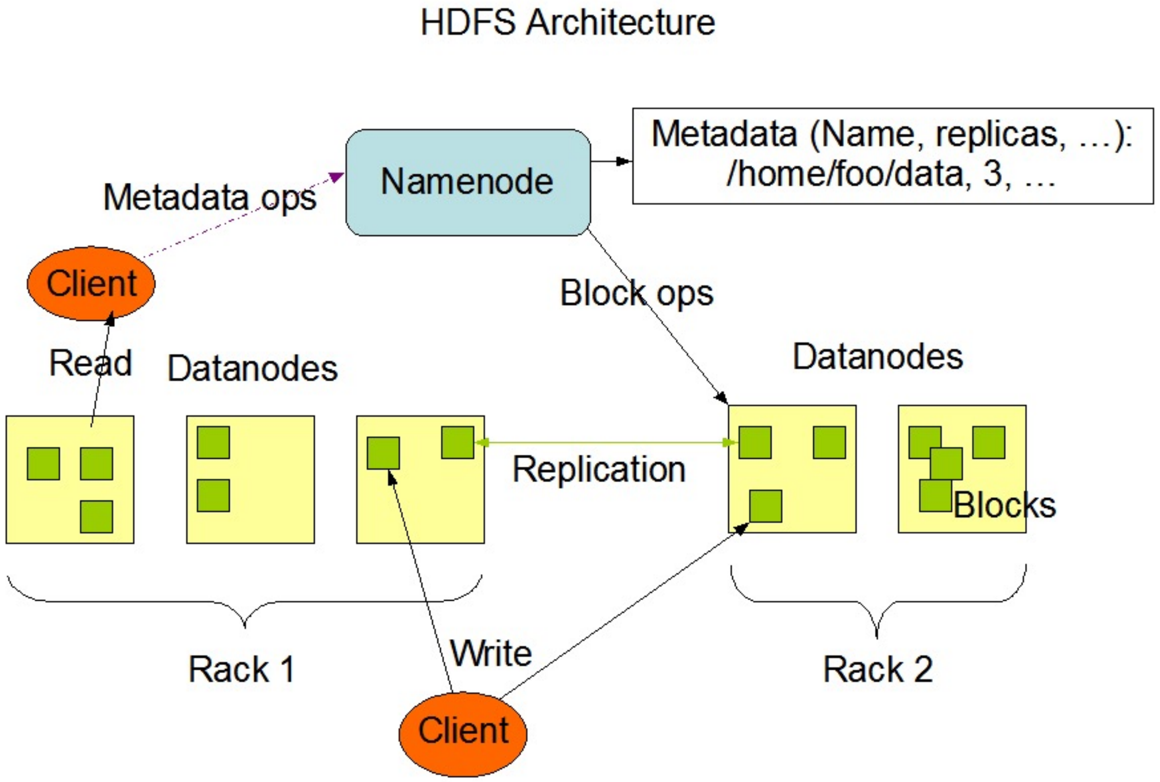
- Typically 64MB block size
- Each block replicated on multiple DataNodes

Intelligent Client

- Client can find location of blocks
- Client accesses data directly from DataNode

HDFS Architecture

HDFS Architecture



Marco Brambilla.

Functions of a NameNode

- Manages File System Namespace
  - Maps a file name to a set of blocks
  - Maps a block to the DataNodes where it resides
- Cluster Configuration Management
- Replication Engine for Blocks

NameNode Metadata

- Metadata in Memory
  - The entire metadata is in main memory
  - No demand paging of metadata
- Types of metadata
  - List of files
  - List of Blocks for each file
  - List of DataNodes for each block
  - File attributes, e.g. creation time, replication factor
- A Transaction Log
  - Records file creations, file deletions etc

DataNode

- A Block Server
  - Stores data in the local file system (e.g. ext3)
  - Stores metadata of a block (e.g. CRC)
  - Serves data and metadata to Clients
- Block Report
  - Periodically sends a report of all existing blocks to the NameNode
- Facilitates Pipelining of Data
  - Forwards data to other specified DataNodes

Block Placement

Strategy

- One replica on local node
- Second replica on a remote rack
- Third replica on same remote rack
- Additional replicas are randomly placed

Clients read from nearest replicas

Heartbeats

- DataNodes send hearbeat to the NameNode
  - Once every 3 seconds
- NameNode uses heartbeats to detect DataNode failure

Replication Engine

- NameNode detects DataNode failures
  - Chooses new DataNodes for new replicas
  - Balances disk usage
  - Balances communication traffic to DataNodes

Data Correctness

- Use Checksums to validate data
  - Use CRC32
- File Creation
  - Client computes checksum per 512 bytes
  - DataNode stores the checksum
- File access
  - Client retrieves the data and checksum from DataNode
  - If Validation fails, Client tries other replicas

NameNode Failure

- A single point of failure in HDFS 1
- Transaction Log stored in multiple directories
  - A directory on the local file system
  - A directory on a remote file system (NFS/CIFS)

Data Pieplining

Client retrieves a list of DataNodes on which to place replicas of a block

Client writes block to the first DataNode

The first DataNode forwards the data to the next node in the Pipeline

When all replicas are written, the Client moves on to write the next block in file

Rebalancer

- Goal: % disk full on DataNodes should be similar
  - Usually run when new DataNodes are added
  - Cluster is online when Rebalancer is active
  - Rebalancer is throttled to avoid network congestion

Secondary NameNode

Copies FsImage and Transaction Log from Namenode to a temporary directory

Merges FSImage and Transaction Log into a new FSImage in temporary directory

Uploads new FSImage to the NameNode

- Transaction Log on NameNode is purged

User Interface

Commads for HDFS User:

hadoop dfs -mkdir /foodir
hadoop dfs -cat /foodir/myfile.txt
hadoop dfs -rm /foodir/myfile.txt

Commands for HDFS Administrator

hadoop dfsadmin -report
hadoop dfsadmin -decommision datanodename

Web Interface

http://host:port/dfshealth.jsp

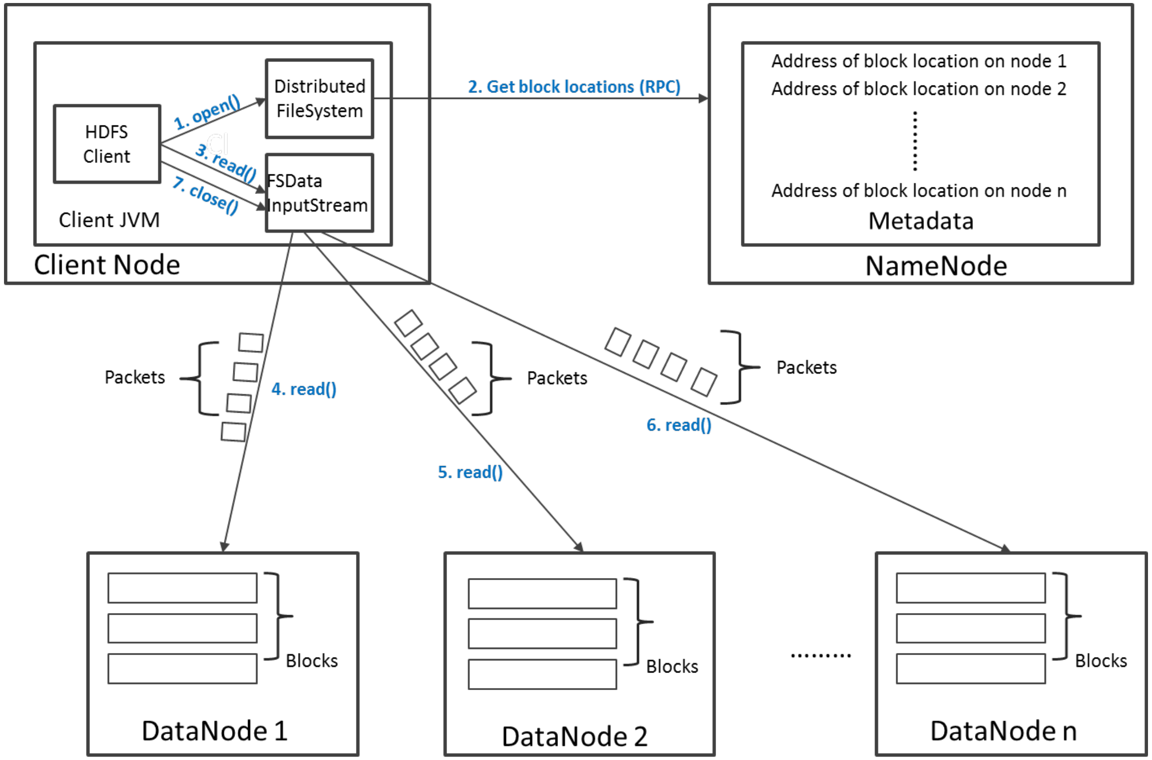
Data Retrieval

When a client wants to retrieve data

- Communicates with the NameNode to determine which blocks make up a file and on which data nodes those blocks are stored

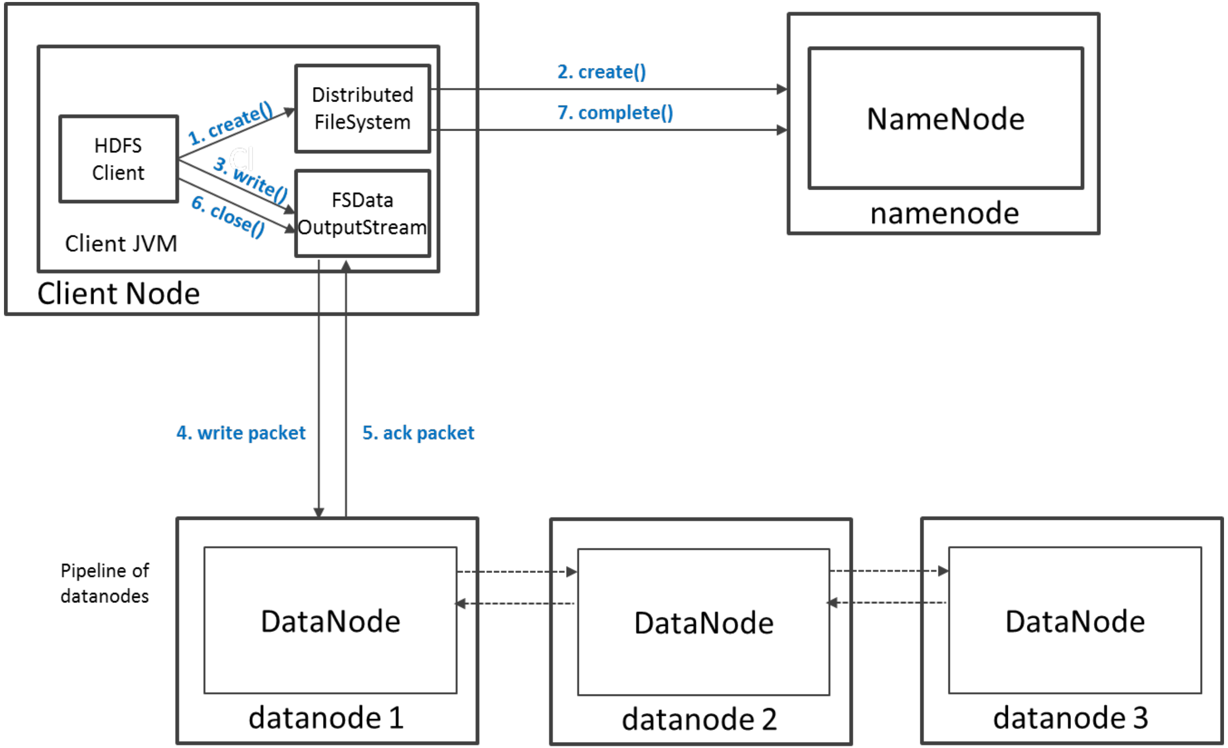
- Then communicated directly with the data nodes to read the data

Read Operation in HDFS



Marco Brambilla.

Write Operation in HDFS



Marco Brambilla.

HDFS Security

Authentication to Hadoop

- Simple – insecure way of using OS username to determine hadoop identity
- Kerberos – authentication using kerberos ticket
- Set by `hadoop.security.authentication=simple|kerberos`

File and Directory permissions are same like in POSIX

- read (r), write (w), and execute (x) permissions
- also has an owner, group and mode
- enabled by default (`dfs.permissions.enabled=true`)

ACLs are used for implemention permissions that differ from natural hierarchy of users and groups

- enabled by `dfs.namenode.acls.enabled=true`

HDFS Configuration

HDFS Defaults

Block Size – 64 MB

Replication Factor – 3

Web UI Port – 50070

HDFS conf file - `/etc/hadoop/conf/hdfs-site.xml`

Interfaces to HDFS

Java API (`DistributedFileSystem`)

C wrapper (`libhdfs`)

HTTP protocol

WebDAV protocol

Shell Commands

However the command line is one of the simplest and most familiar

HDFS Commands

HDFS – Shell Commands

There are two types of shell commands

User Commands

- `hdfs dfs` – runs filesystem commands on the HDFS
- `hdfs fsck` – runs a HDFS filesystem checking command

Administration Commands

- `hdfs dfsadmin` – runs HDFS administration commands

HDFS – User Commands (dfs)

List directory contents

```
hdfs dfs -ls
hdfs dfs -ls /
hdfs dfs -ls -R /var
```

Display the disk space used by files

```
hdfs dfs -du -h /
hdfs dfs -du /hbase/data/hbase/namespace/
hdfs dfs -du -h /hbase/data/hbase/namespace/
hdfs dfs -du -s /hbase/data/hbase/namespace/
```

HDFS – User Commands (dfs)

Copy data to HDFS

```
hdfs dfs -mkdir tdata
hdfs dfs -ls
hdfs dfs -copyFromLocal tutorials/data/geneva.csv tdata
hdfs dfs -ls -R
```

Copy the file back to local filesystem

```
cd tutorials/data/
hdfs dfs -copyToLocal tdata/geneva.csv geneva.csv.hdfs
md5sum geneva.csv geneva.csv.hdfs
```

HDFS – User Commands (acls)

List acl for a file

```
hdfs dfs -getfacl tdata/geneva.csv
```

List the file statistics – (%r – replication factor)

```
hdfs dfs -stat "%r" tdata/geneva.csv
```

Write to hdfs reading from stdin

```
echo "blah blah blah" | hdfs dfs -put - tdataset/tfile.txt
hdfs dfs -ls -R
hdfs dfs -cat tdataset/tfile.txt
```

HDFS – User Commands (fsck)

Removing a file

```
hdfs dfs -rm tdataset/tfile.txt
hdfs dfs -ls –R
```

List the blocks of a file and their locations

```
hdfs fsck /user/cloudera/tdata/geneva.csv -
files -blocks –locations
```

Print missing blocks and the files they belong to

```
hdfs fsck / -list-corruptfileblocks
```

HDFS – Adminstration Commands

Comprehensive status report of HDFS cluster

```
hdfs dfsadmin -report
```

Prints a tree of racks and their nodes

```
hdfs dfsadmin -printTopology
```

Get the information for a given datanode (like ping)

```
hdfs dfsadmin -getDatanodeInfo
localhost:50020
```

HDFS – Advanced Commands

Get a list of namenodes in the Hadoop cluster

```
hdfs getconf -namenodes
```

Dump the NameNode fsimage to XML file

```
cd /var/lib/hadoop-hdfs/cache/hdfs/dfs/name/current
hdfs oiv -i fsimage_0000000000000003388 -o
/tmp/fsimage.xml -p XML
```

The general command line syntax is

hdfs command [genericOptions] [commandOptions]

Other Interfaces to HDFS

HTTP Interface

```
http://quickstart.cloudera:50070
```

MountableHDFS – FUSE

```
mkdir /home/cloudera/hdfs
sudo hadoop-fuse-dfs dfs://quickstart.cloudera:8020
/home/cloudera/hdfs
```

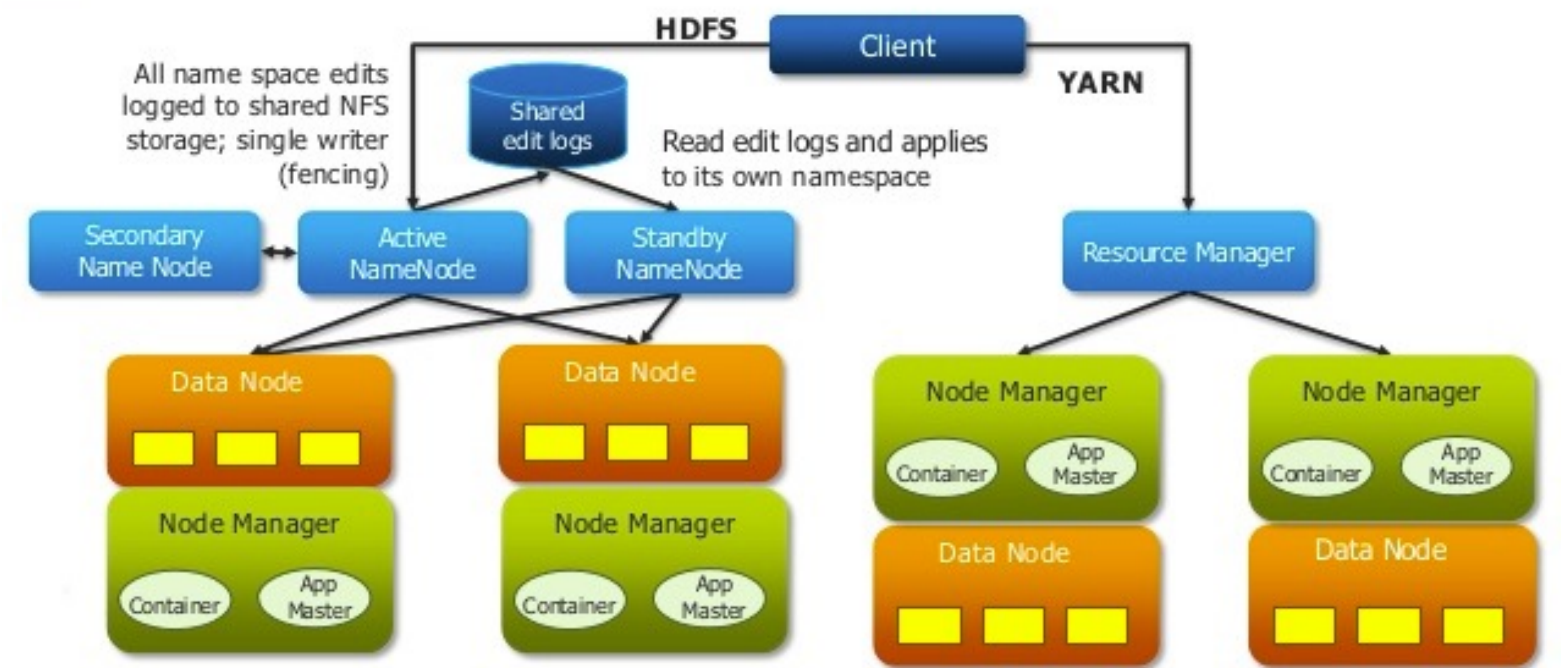
Once mounted all operations on HDFS can be performed using standard Unix utilities such as 'ls', 'cd', 'cp', 'mkdir', 'find', 'grep',

Hadoop 2.0

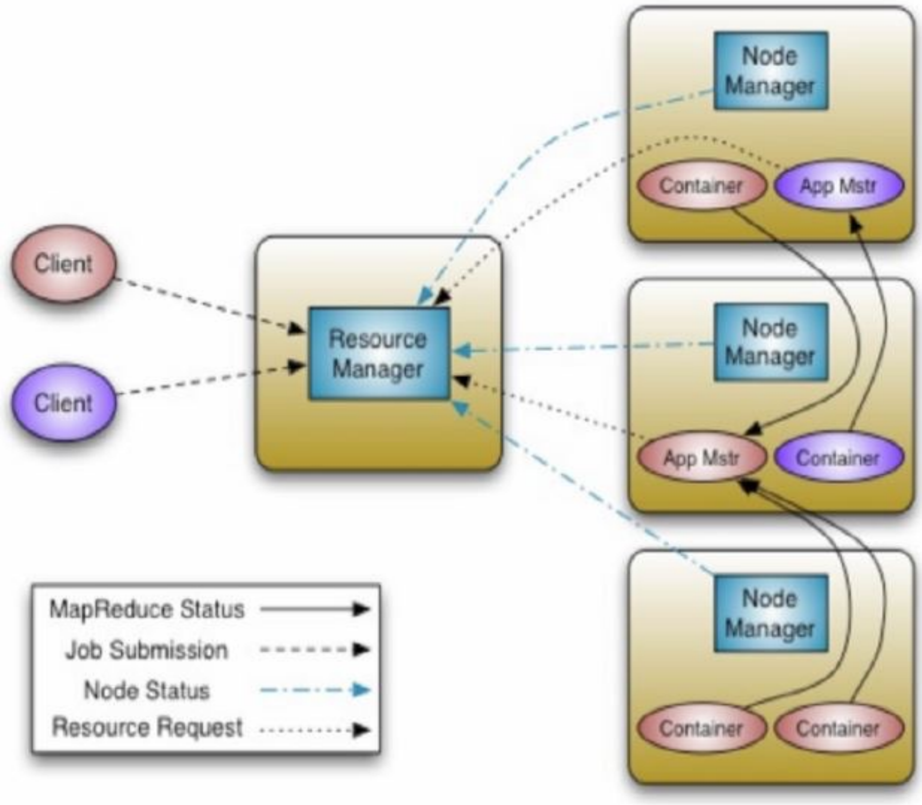
YARN

- Splits up the two major functions of JobTracker
    - Global Resource Manager - Cluster resource management
    - Application Master - Job scheduling and monitoring (one per application). The Application Master negotiates resource containers from the Scheduler, tracking their status and monitoring for progress. Application Master itself runs as a normal *container*.
- Tasktracker
    - NodeManager (NM) - A new per-node slave is responsible for launching the applications' containers, monitoring their resource usage (cpu, memory, disk, network) and reporting to the Resource Manager.
- YARN maintains compatibility with existing MapReduce applications and users.

Hadoop 2.0 Architecture - YARN



Marco Brambilla.



- Scalability - Clusters of 6,000-10,000 machines
  - Each machine with 16 cores, 48G/96G RAM, 24TB/36TB disks
  - 100,000+ concurrent tasks
  - 10,000 concurrent jobs

Classic MapReduce vs. YARN

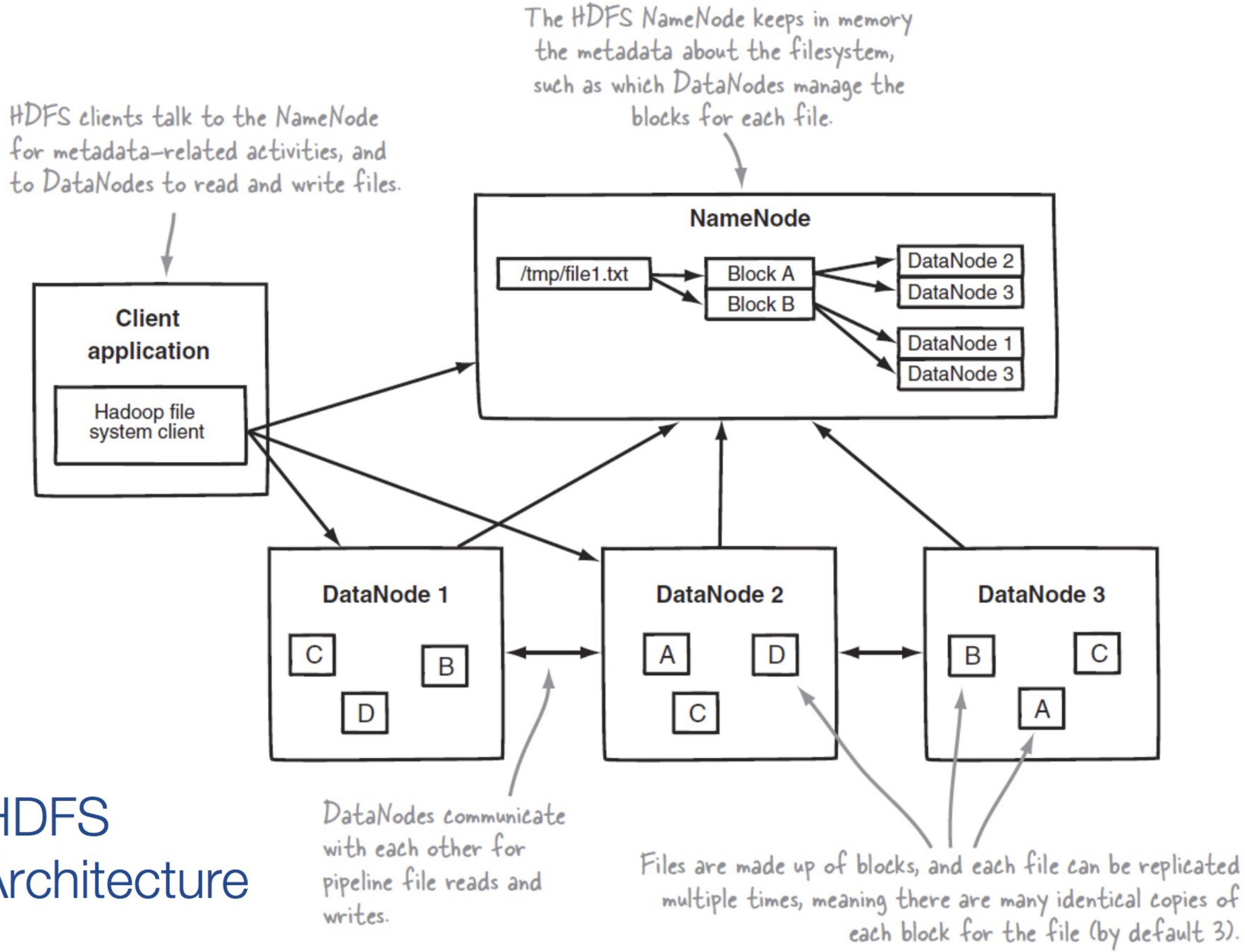
- Fault Tolerance and Availability
  - Resource Manager
    - No single point of failure – state saved in ZooKeeper
    - Application Masters are restarted automatically on RM restart
  - Application Master
    - Optional failover via application-specific checkpoint
    - MapReduce applications pick up where they left off via state saved in HDFS
- Wire Compatibility
  - Protocols are wire-compatible
  - Old clients can talk to new servers
  - Rolling upgrades

Classic MapReduce vs. YARN

- Support for programming paradigms other than MapReduce (Multi tenancy)
  - Tez – Generic framework to run a complex DAG
  - HBase on YARN(HOYA)
  - Machine Learning: Spark
  - Graph processing: Giraph
  - Real-time processing: Storm
  - Enabled by allowing the use of paradigm-specific application master
  - *Run all on the same Hadoop cluster!*

Concluding...

HDFS Summary



HDFS Architecture

Map Reduce

Agenda

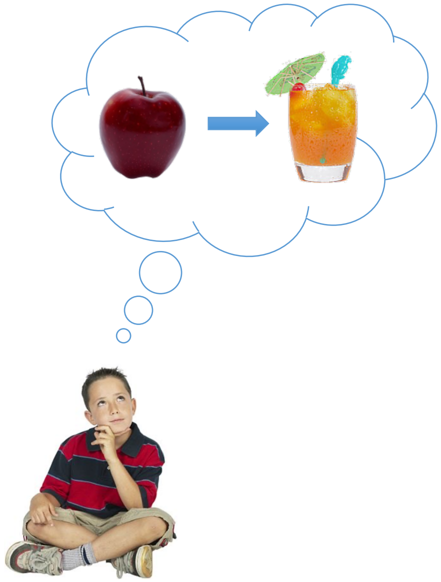
Intro & Intuition

Definition

Typical Problems

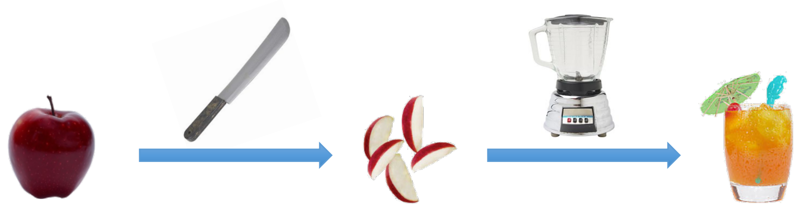
The Juice-making business

1. Wish



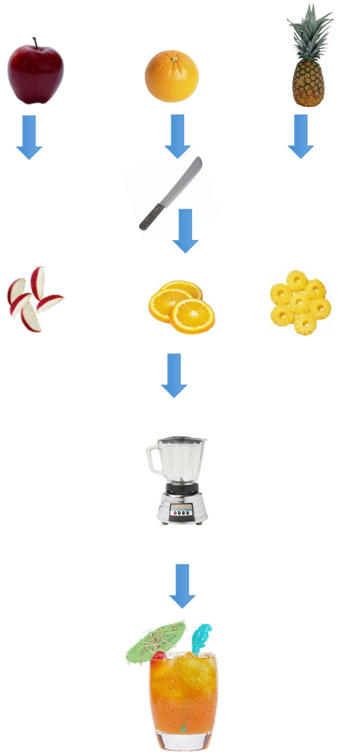
The Juice-making business

2. Prototype



Next Day

Apply it to all the fruits in the *fruit basket*



Marco Brambilla.

Industrialization

A juice making giant is making juice

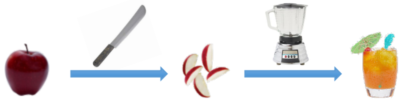
- whole container of fruits

- juice of different fruits separately

Marco Brambilla.

Why?

The operations themselves are conceptually simple

- Making juice 
- Indexing
- Recommendations etc

But, the data to process is **HUGE!!!**

- Google processes over 50 PB of data every day

Sequential execution just won’t scale up

Why?

Parallel execution achieves greater efficiency

But, parallel programming is hard

- Parallelization
  - Race Conditions
  - Debugging
- Fault Tolerance
- Data Distribution
- Load Balancing

MapReduce

> “*MapReduce is a programming model and an associated implementation for processing and generating large data sets*”

Programming model

Abstractions to express simple computations

Implementation (existing)

Takes care of the gory stuff: Parallelization, Fault Tolerance, Data Distribution and Load Balancing

MapReduce Advantages

- Automatic parallelization, distribution
- I/O scheduling
  - Load balancing
  - Network and data transfer optimization
- Fault tolerance
  - Handling of machine failures

Need more power: Scale out, not up!

Map? Reduce?

Mappers read in data from the filesystem, and output (typically) modified data

Reducers collect all of the mappers output on the keys, and output (typically) reduced data

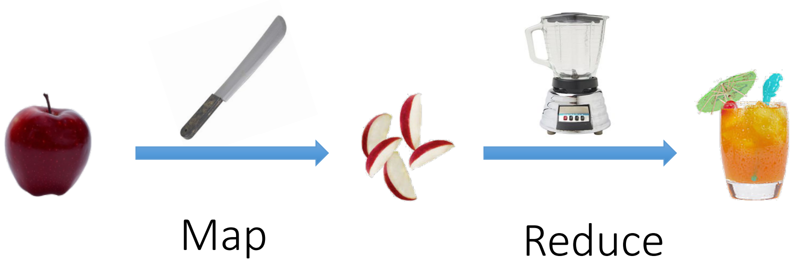
The output data is written to disk

All data is in terms of *key-value* pairs

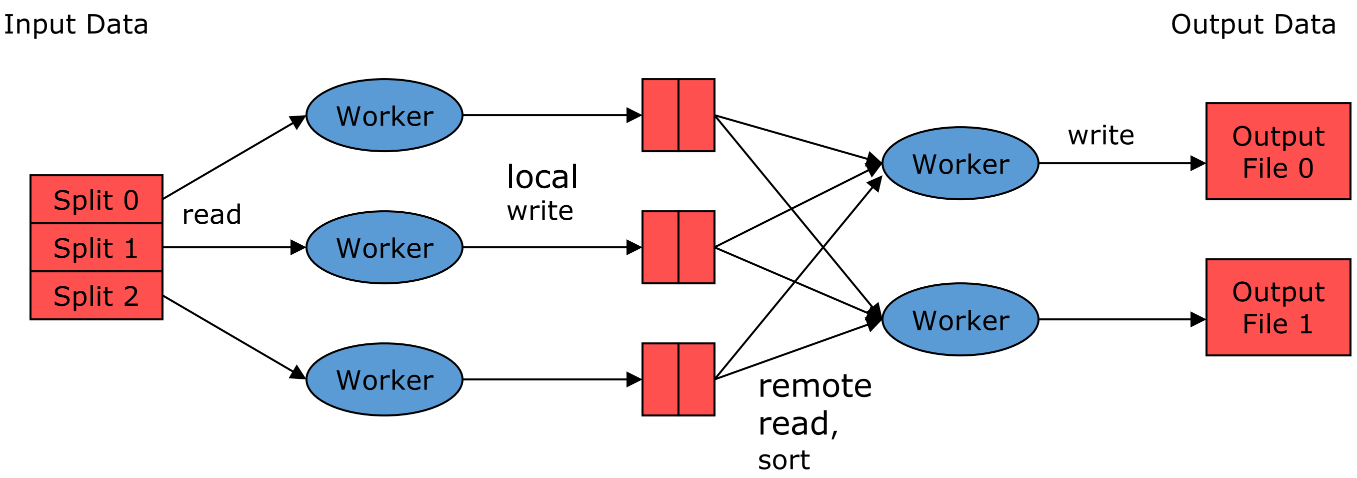
Typical problem solved by MapReduce

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

Example of Map-Reduce



MapReduce workflow



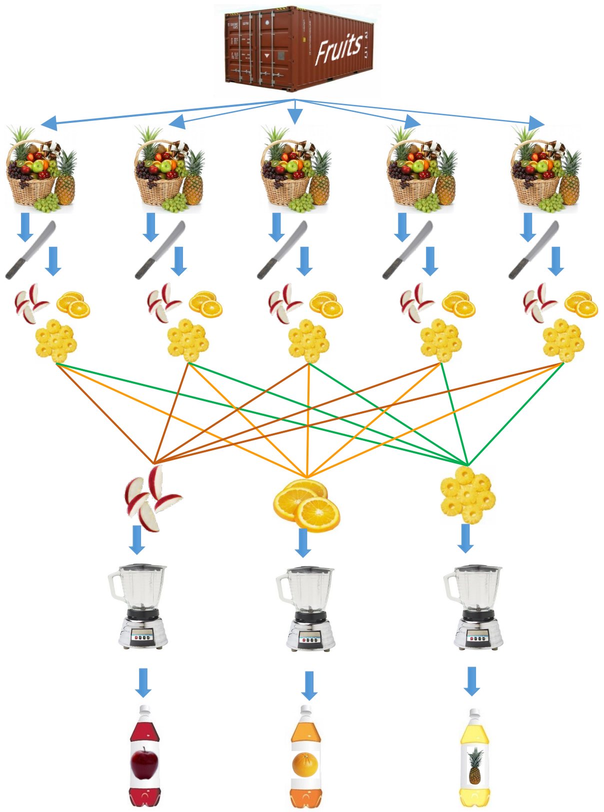
Map
extract something you care about from each record

Reduce
aggregate, summarize, filter, or transform

Marco Brambilla.

Complete MapReduce Example

A *parallel* version of the process



Marco Brambilla.

Programming Model

To generate a set of output key-value pairs from a set of input key-value pairs

$\{ < k_i, v_i >\} \rightarrow \{ < k_o, v_o >\}$

Expressed using two abstractions:

- Map task
  - $<k_i, v_i> \rightarrow \{ < k_{int}, v_{int} > \}$
- Reduce task
  - $< k_{int}, \{v_{int}\} > \rightarrow < k_o, v_o >$

Library

- aggregates all the all intermediate values associated with the same intermediate key
- passes the intermediate key-value pairs to ***reduce*** function

Mapper

Reads in input pair <Key,Value>
Outputs a pair <K’, V’>

Let’s count number of each word in user queries (or Tweets/Blogs)
The input to the mapper will be <queryID, QueryText>:

```
<Q1,“The teacher went to the store. The store was closed; the
store opens in the morning. The store opens at 9am.” >
```

The output would be:

```
<The, 1> <teacher, 1> <went, 1> <to, 1> <the, 1> <store,1> <the, 1> <store,
1> <was, 1> <closed, 1> <the, 1> <store,1> <opens, 1> <in, 1> <the, 1>
<morning, 1> <the 1> <store, 1> <opens, 1> <at, 1> <9am, 1>
```

Reducer

Accepts the Mapper output, and aggregates values on the key

- For our example, the reducer input would be:
  - <The, 1> <teacher, 1> <went, 1> <to, 1> <the, 1> **<store, 1>** <the, 1> <store, 1> <was, 1> <closed, 1> <the, 1> **<store, 1>** <opens,1> <in, 1> <the, 1> <morning, 1> <the 1> **<store, 1>** <opens, 1> <at, 1> <9am, 1>
- The output would be:
  - <The, 6> <teacher, 1> <went, 1> <to, 1> **<store, 3>** <was, 1> <closed, 1> <opens, 1> <morning, 1> <at, 1> <9am, 1>

Key Components

Input Splitter

- Is responsible for splitting your input into multiple chunks
- These chunks are then used as input for your mappers
- Splits on logical boundaries. The default is 64MB per chunk
    - Depending on what you're doing, 64MB might be a LOT of data! You can change it
- Typically, you can just use one of the built in splitters, unless you are reading in a specially formatted file

Mapper

Reads in input pair <K,V> (a section as split by the input splitter)

Outputs a pair <K’, V’>

Ex. For our Word Count example, with the following input: “The teacher went to the store. The store was closed; the store opens in the morning. The store opens at 9am.”

The output would be:

<The, 1> <teacher, 1> <went, 1> <to, 1> <the, 1> <store, 1> <the, 1> <store, 1> <was, 1> <closed, 1> <the, 1> <store, 1> <opens, 1> <in, 1> <the, 1> <morning, 1> <the 1> <store, 1> <opens, 1> <at, 1> <9am, 1>

Reducer

- Accepts the Mapper output, and collects values on the key
  - All inputs with the same key *must* go to the same reducer!
- Input is typically sorted, output is output exactly as is
- For our example, the reducer input would be:
  - <The, 1> <teacher, 1> <went, 1> <to, 1> <the, 1> <store, 1> <the, 1> <store, 1> <was, 1> <closed, 1> <the, 1> <store, 1> <opens, 1> <in, 1> <the, 1> <morning, 1> <the 1> <store, 1> <opens, 1> <at, 1> <9am, 1>
- The output would be:
  - <The, 6> <teacher, 1> <went, 1> <to, 1> <store, 3> <was, 1> <closed, 1> <opens, 1> <morning, 1> <at, 1> <9am, 1>

Partitioner (Shuffler)

Decides which pairs are sent to which reducer

Default is simply:

- Key.hashCode() % numOfReducers

Custom partitioning is often required, for example, to produce a total order in the output. User can override to:

- Provide (more) uniform distribution of load between reducers
- Some values might need to be sent to the same reducer
  - Ex. To compute the relative frequency of a pair of words <W1, W2> you would need to make sure all of word W1 are sent to the same reducer
- Binning of results

How?

- Should implement *Partitioner* interface
- Set by calling `conf.setPartitionerClass(MyPart.class)`

Combiner

- Essentially an intermediate reducer
- Is optional
- Reduces output from each mapper, reducing bandwidth and sorting
- Cannot change the type of its input
  - Input types must be the same as output types

Output Committer

Is responsible for taking the reduce output, and committing it to a file

Typically, this committer needs a corresponding input splitter (so that another job can read the input)

Again, usually built-in committers are good enough, unless you need to output a special kind of file

Master

Responsible for scheduling & managing jobs

Scheduled computation should be close to the data if possible

- Bandwidth is expensive! (and slow)
- This relies on a Distributed File System (GFS / HDFS)!

If a task fails to report progress (such as reading input, writing output, etc), crashes, the machine goes down, etc, it is assumed to be stuck, and is killed, and the step is re-launched (with the same input)

The Master is handled by the framework, no user code is necessary

Master

HDFS can replicate data to be local if necessary for scheduling

Because our nodes are (or at least should be) deterministic

- The Master can restart failed nodes
  - Nodes should have no side effects!
- If a node is the last step, and is completing slowly, the master can launch a second copy of that node
  - This can be due to hardware isuses, network issues, etc.
  - First one to complete wins, then any other runs are killed

Data Typing: Writables

Are types that can be serialized / deserialized to a stream

Are required to be input/output classes, as the framework will serialize your data before writing it to disk

User can implement this interface, and use their own types for their input/output/intermediate values

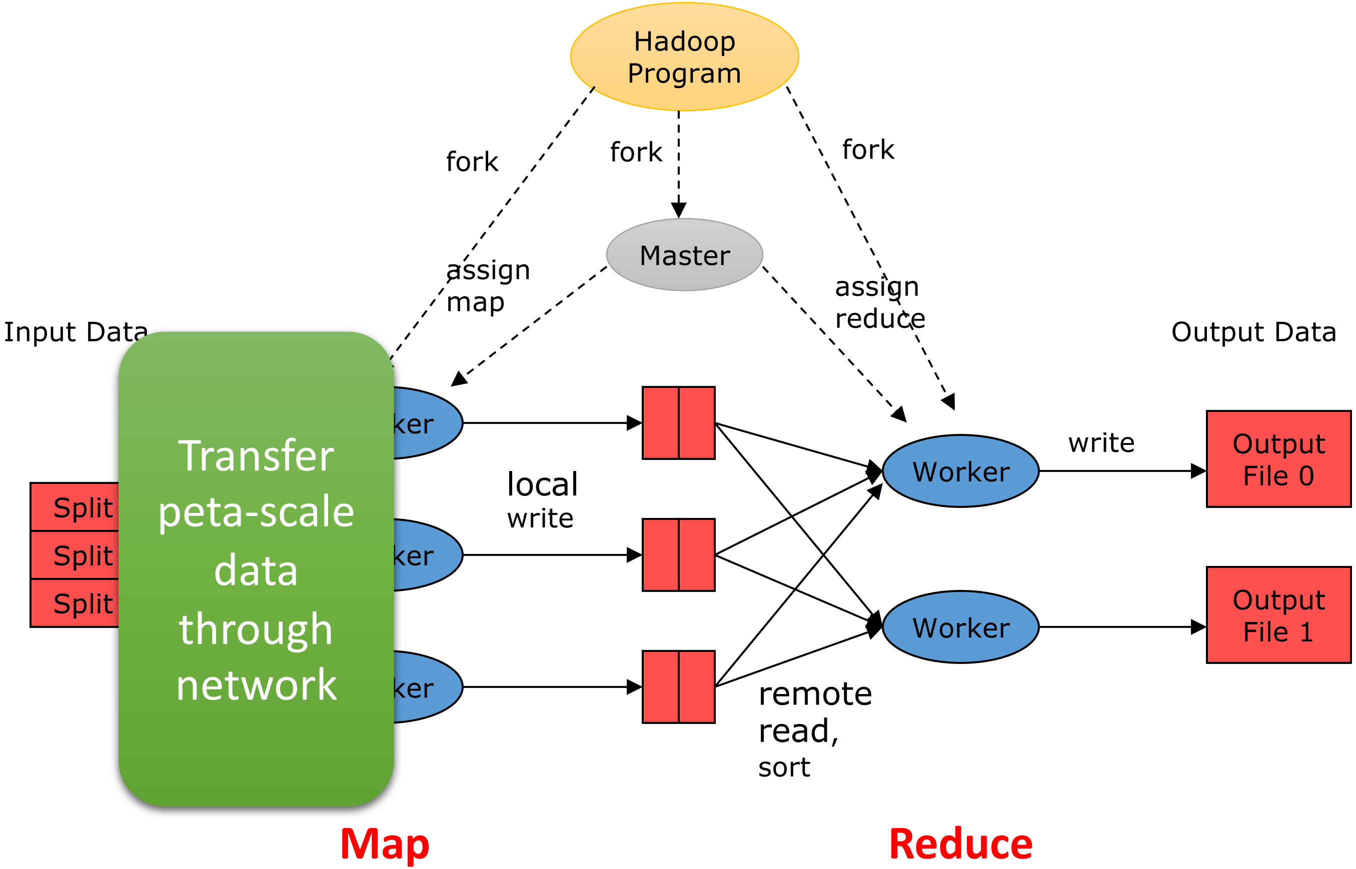
There are default for basic values, like Strings, Integers, Longs, etc.

Can also handle store, such as arrays, maps, etc.

Your application needs at least six writables

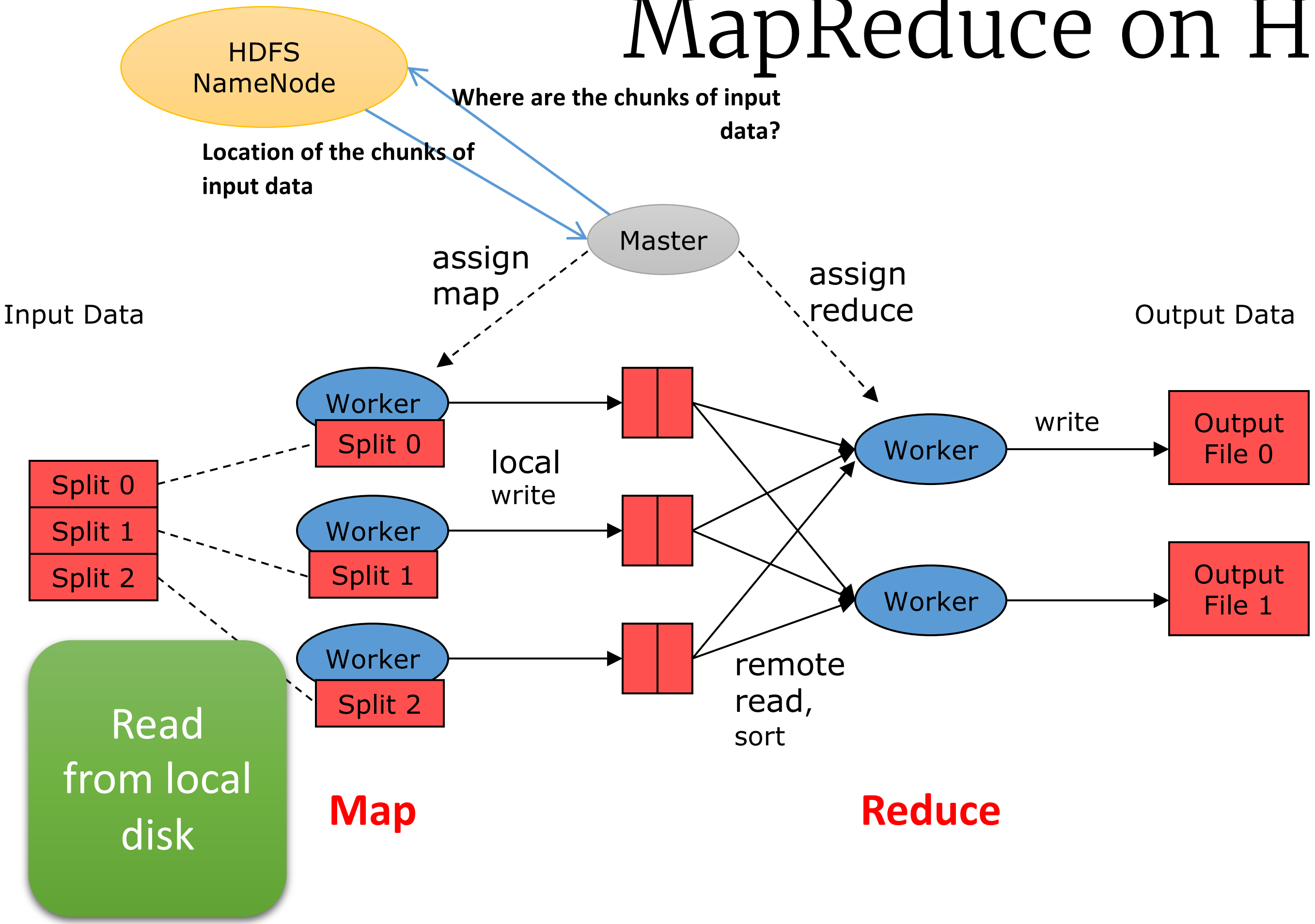
- 2 for your input
- 2 for your intermediate values (Map <-> Reduce)
- 2 for your output

MapReduce



Marco Brambilla.

MapReduce on HDFS



Marco Brambilla.

Execution

System determines:

- *M*: no. of map tasks

User specifies:

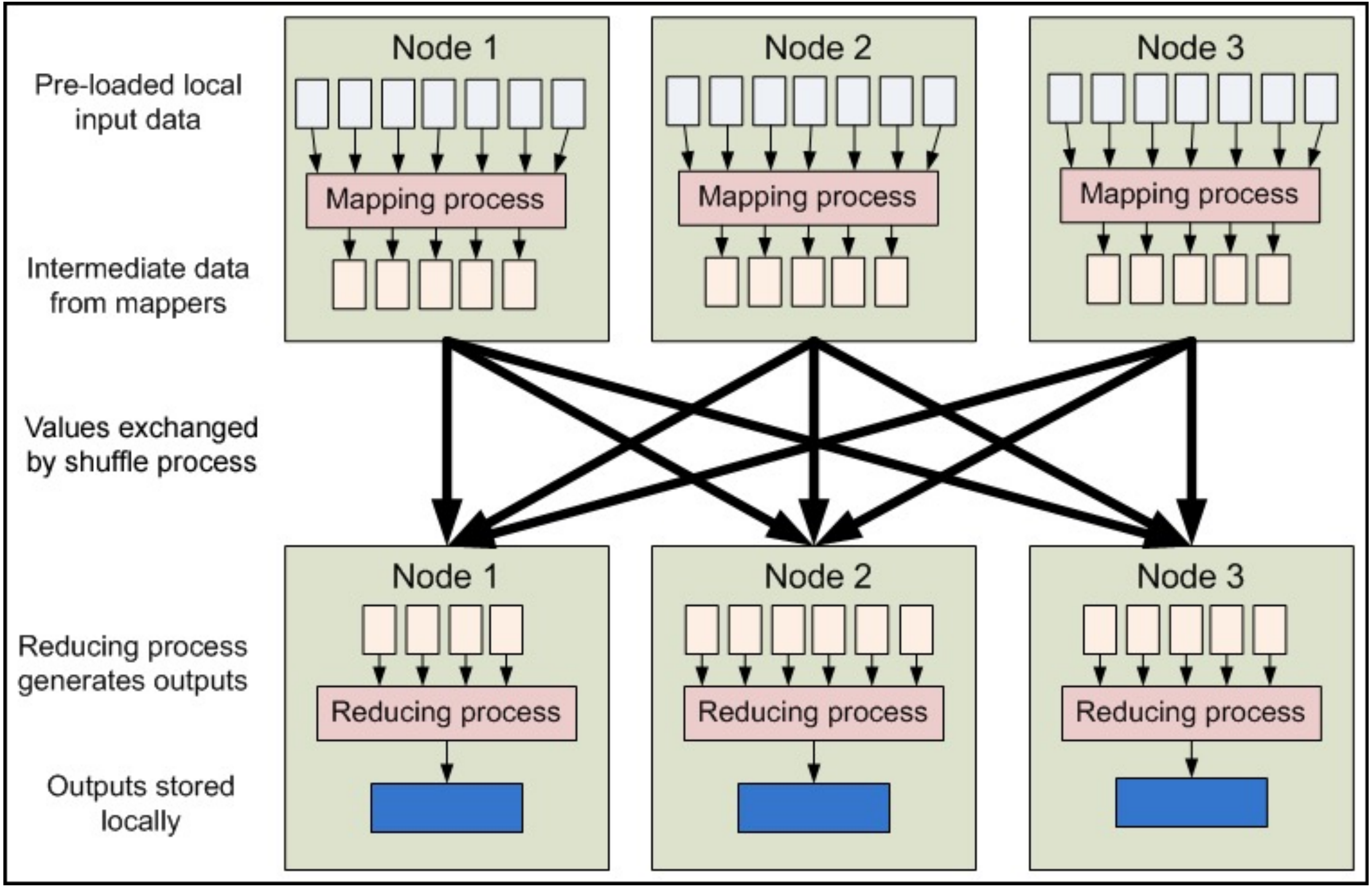
- *R*: no. of reduce tasks

Map Phase

- input is partitioned into *M* splits
- map tasks are distributed across multiple machines

Reduce Phase

- reduce tasks are distributed across multiple machines
- intermediate keys are partitioned (using partitioning function) to be processed by desired reduce task


Node 1
Node 2
Node 3
Pre-loaded local input data
Mapping process
Mapping process
Mapping process
Intermediate data from mappers
Values exchanged by shuffle process
Node 1
Node 2
Node 3
Reducing process generates outputs
Reducing process
Reducing process
Reducing process
Outputs stored locally

Example: Word Count

```
map(String input_key, String input_value):
  // input_key: document name
  // input_value: document contents
  for each word w in input_value:
    EmitIntermediate(w, "1");
```

<“Sam”, “1”>, <“Apple”, “1”>,
<“Sam”, “1”>, <“Mom”, “1”>,
<“Sam”, “1”>, <“Mom”, “1”>,

```
reduce(String output_key, Iterator intermediate_values):
  // output_key: a word
  // output_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v);
  Emit(AsString(result));
```

<“Sam” , [“1”,”1”,”1”]>,
<“Apple” , [“1”]>,
<“Mom” , [“1”, “1”]>

“3”
“1”
“2”

Marco Brambilla.

Complete MapReduce Job

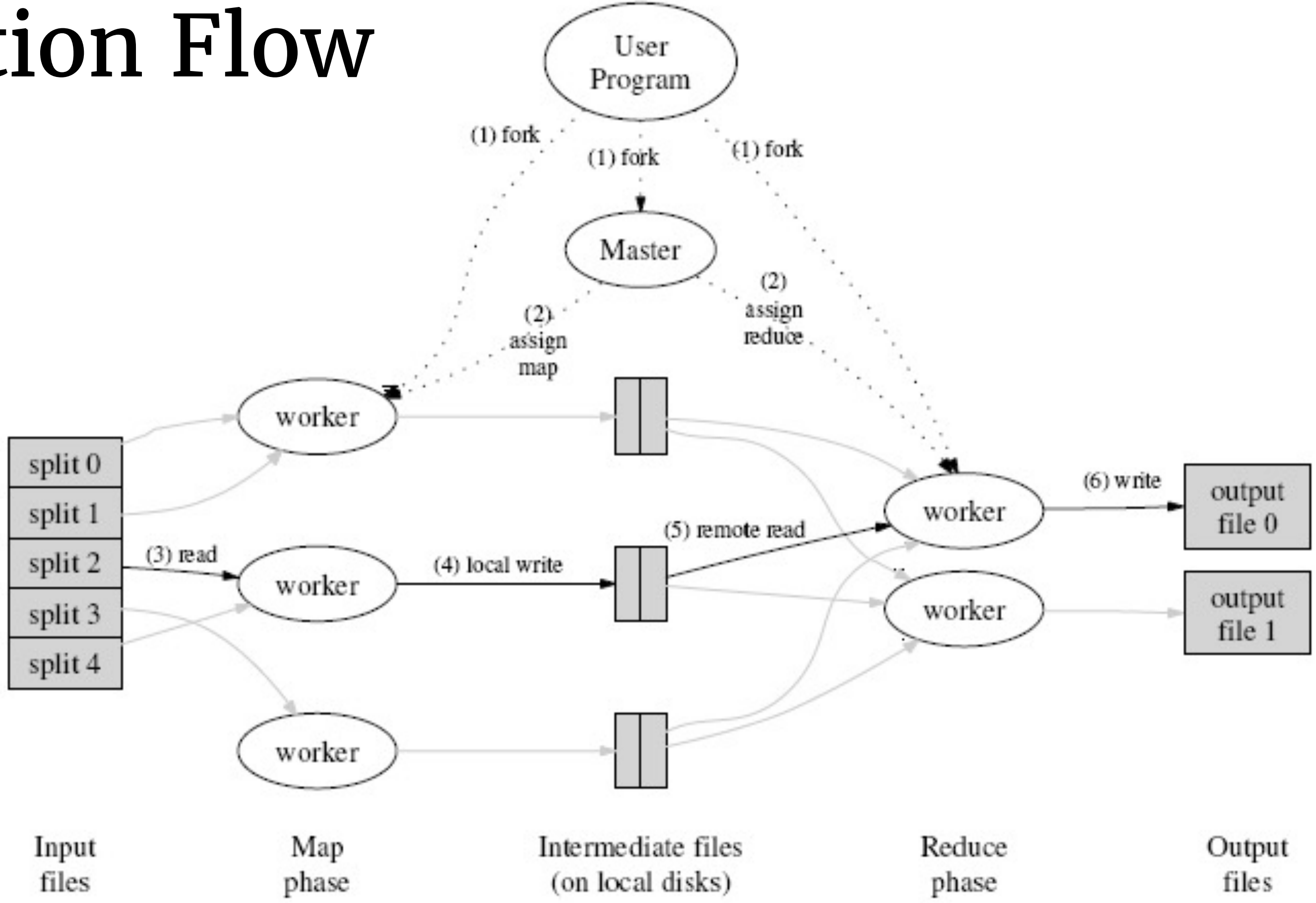
```java
public class WordCount {

  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));
  }}

  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);
  }}
```

Mapper

Reducer

Run this program as a MapReduce job

Execution Flow



Master Data Structures

For each task

- State { *idle*, *in-progress*, *completed* }
- Identity of the worker machine

For each completed map task

- Size and location of intermediate data

Some further handy tools

Combiners
Compression
Counters
Speculation
Zero Reduces

Combiners

When *maps* produce many repeated keys

- It is often useful to do a local aggregation following the *map*
- Done by specifying a *Combiner*
- Goal is to decrease size of the transient data
- Combiners have the same interface as Reduces, and often are the same class
- Combiners must **not** side effects, because they run an intermdiate number of times
- In *WordCount*, *conf.setCombinerClass(Reduce.class);*

Compression

Compressing the outputs and intermediate data will often yield huge performance gains

- Can be specified via a configuration file or set programmatically
- Set *mapred.output.compress* to *true* to compress job output
- Set *mapred.compress.map.output* to *true* to compress map outputs

Compression Types (*mapred(.map)?.output.compression.type)*

- "block" - Group of keys and values are compressed together
- "record" - Each value is compressed individually
- Block compression is almost always best

Compression Codecs *(mapred(.map)?.output.compression.codec)*

- Default (zlib) - slower, but more compression
- LZO - faster, but less compression

Counters

Often Map/Reduce applications have countable events

For example, framework counts records in to and out of Mapper and Reducer

To define user counters:

```
static enum Counter {EVENT1, EVENT2};
reporter.incrCounter(Counter.EVENT1, 1);
```

Define nice names in a MyClass_Counter.properties file

```
CounterGroupName=MyCounters
EVENT1.name=Event 1
EVENT2.name=Event 2
```

Speculative execution

The framework can run multiple instances of slow tasks

- Output from instance that finishes first is used
- Controlled by the configuration variable *mapred.speculative.execution*
- Can dramatically bring in long tails on jobs

Zero Reduces

Frequently, we only need to run a filter on the input data

- No sorting or shuffling required by the job
- Set the number of reduces to 0
- Output from maps will go directly to OutputFormat and disk

Properties

Fault Tolerance

- Worker failure – handled via re-execution
  - Identified by no response to heartbeat messages
  - *In-progress* and *Completed* map tasks are re-scheduled
  - Workers executing reduce tasks are notified of re-scheduling
  - *Completed* reduce tasks are not re-scheduled
- Master failure
  - Rare
  - Can be recovered from checkpoints
  - All tasks abort

Disk Locality

Leveraging HDFS

Map tasks are scheduled close to data

- on nodes that have input data
- if not, on nodes that are nearer to input data
  - Ex. Same network switch

Conserves network bandwidth

Task Granularity

No. of map tasks > no. of worker nodes

- Better load balancing
- Better recovery

But, increases load on Master

- More scheduling decisions
- More states to be saved

M could be chosen w.r.t to block size of the file system

- to effectively leverage locality

R is usually specified by users

- Each reduce task produces one output file

Stragglers

- Slow workers delay completion time
  - Bad disks with soft errors
  - Other tasks eating up resources
  - Strange reasons like processor cache being disabled
- Start back-up tasks as the job nears completion
  - First task to complete is considered

Refinement: Partitioning Function

Identifies the desired reduce task

- Given the intermediate key and R

Default partitioning function

- $hash(key)$ **mod** R

Important to choose well-balanced partitioning functions

- If not, reduce tasks may delay completion time

Refinement: Combiner Function

- Mini-reduce phase before the intermediate data is sent to reduce
- Significant repetition of intermediate keys possible
    - Merge values of intermediate keys before sending to reduce tasks
- Similar to reduce function
- Saves network bandwidth

Refinement: Skipping Bad Records

- Map/Reduce tasks may fail on certain records due to bugs
  - Ideally, debug and fix
  - Not possible if third-party code is buggy
- When worker dies, Master is notified of the record
- If more than one worker dies on the same record
  - Master re-schedules the task and asks to skip the record

New Trend: Disk-locality Irrelevant

Assumes disk bandwidth exceeds network bandwidth

Network speeds fast improving

Disk speeds have stagnated

Next step: attain memory-locality

Scheduling

By default, Hadoop uses FIFO to schedule jobs.

Alternate scheduler options:

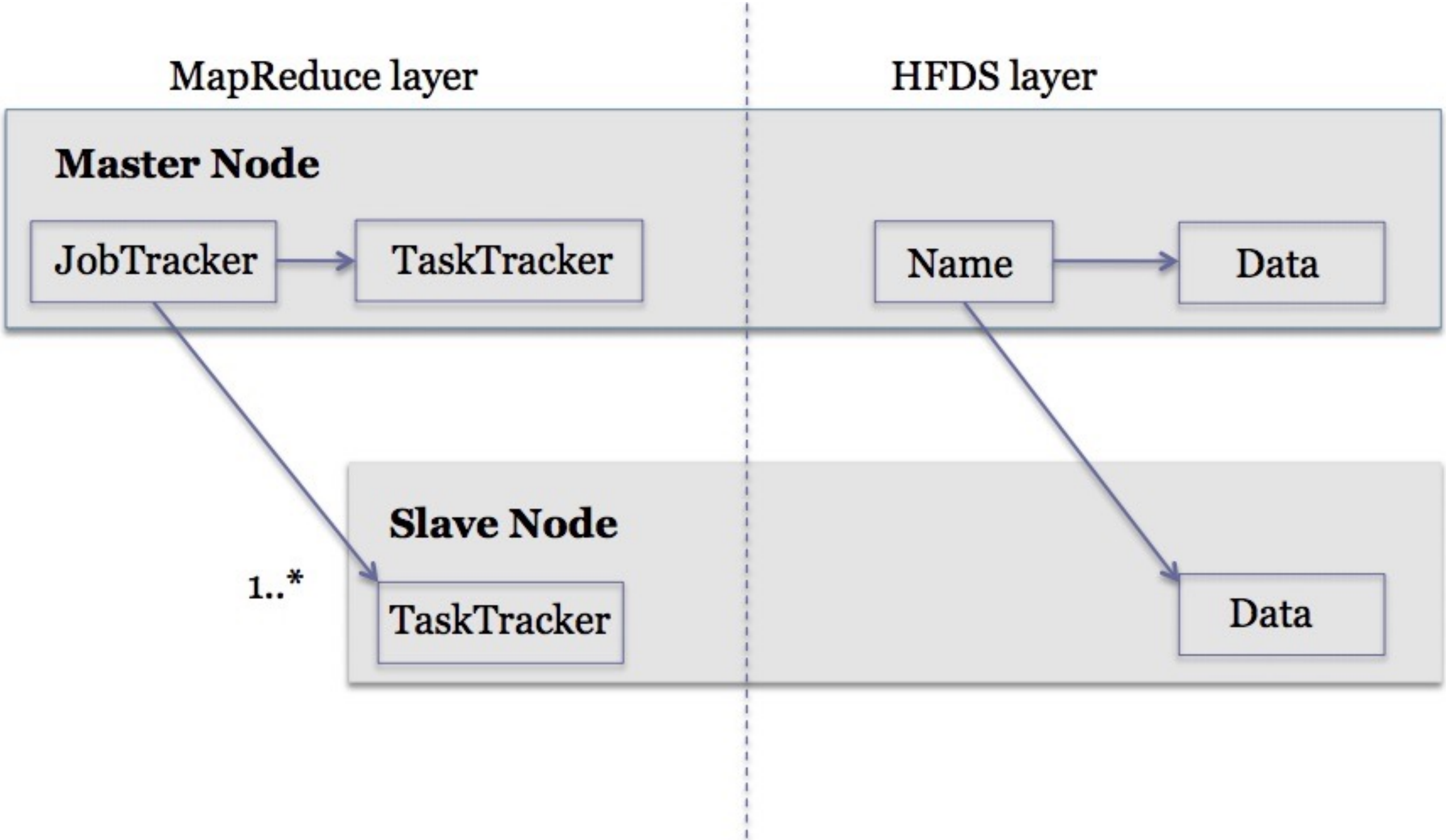
capacity and *fair*

Capacity Scheduler

- Developed by Yahoo
- Jobs are submitted to queues
- Jobs can be prioritized
- Queues are allocated a fraction of the total resource capacity
- Free resources are allocated to queues beyond their total capacity
- No preemption once a job is running

Fair scheduler

- Developed by Facebook
- Provides fast response times for small jobs
- Jobs are grouped into Pools
- Each pool assigned a guaranteed minimum share
- Excess capacity split between jobs
- By default, jobs that are uncategorized go into a default pool.
- Pools have to specify the minimum number of map slots, reduce slots, and a limit on the number of running jobs


MapReduce layer
HFDS layer
Master Node
JobTracker
TaskTracker
Name
Data
Slave Node
1..*
TaskTracker
Data

Conclusion

- Easy to use scalable programming model for large-scale data processing on clusters
  - Allows users to focus on computations
- Hides issues of
  - parallelization, fault tolerance, data partitioning & load balancing
- Achieves efficiency through disk-locality
- Achieves fault-tolerance through replication