Hadoop/MapReduce Computing Paradigm

Large-Scale Data Analytics

- MapReduce computing paradigm (E.g., Hadoop) vs. Traditional database systems

vs.

- **Many enterprises are turning to Hadoop**
  - Especially applications generating ***big data***
  - Web applications, social networks, scientific applications

Why Hadoop is able to compete?

vs.


Scalability (petabytes of data, thousands of machines)


Flexibility in accepting all data formats (no schema)


Efficient and simple fault-tolerant mechanism


Commodity inexpensive hardware


Performance (tons of indexing, tuning, data organization tech.)

Interactive processing


Transactions and consistency guarantees (ACID)


Features:

- Provenance tracking
- Annotation management
- ….

What is MapReduce

- **MapReduce is a computing paradigm**
  - A specific mechanism of processing the data

Different Implementations/Systems

- MapReduce ⟷ Relational DBs

What is Hadoop

- Hadoop is a software framework for *distributed processing* of *large datasets* across *large clusters* of computers
  - ***Large datasets*** → Terabytes or petabytes of data
  - ***Large clusters*** → hundreds or thousands of nodes
- Hadoop is open-source implementation for Google ***MapReduce***
- Hadoop is based on a simple programming model called *MapReduce*
- Hadoop is based on a simple data model, *any data will fit*

Compute Cluster

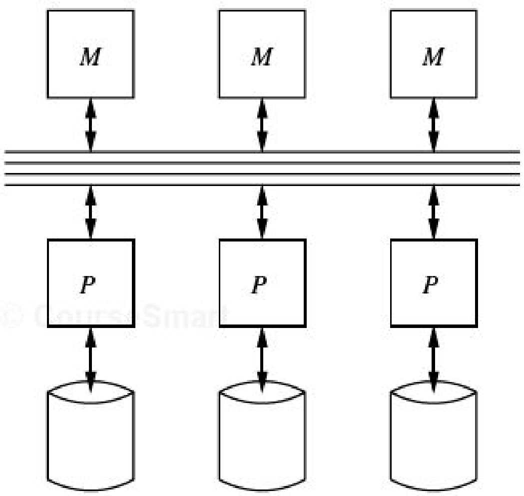
A rack of N machines

One machine ←→ One node

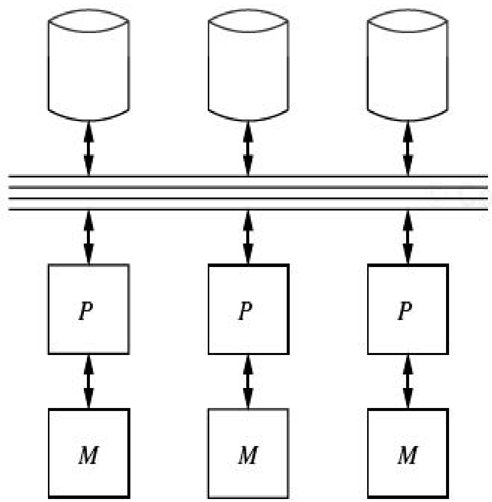
Compute Cluster

- Cluster ➔ Set of machines connected together

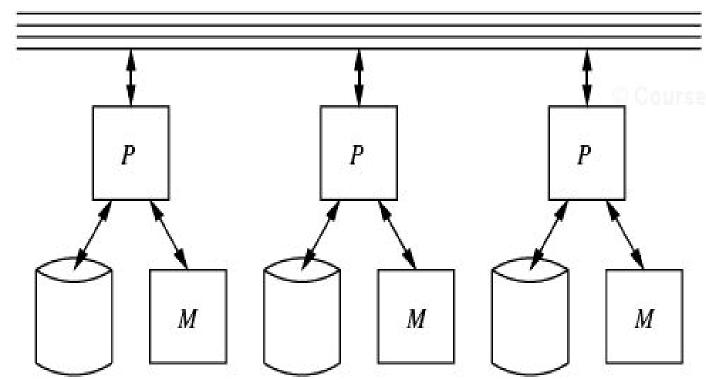
Shared-memory



Shared-disk

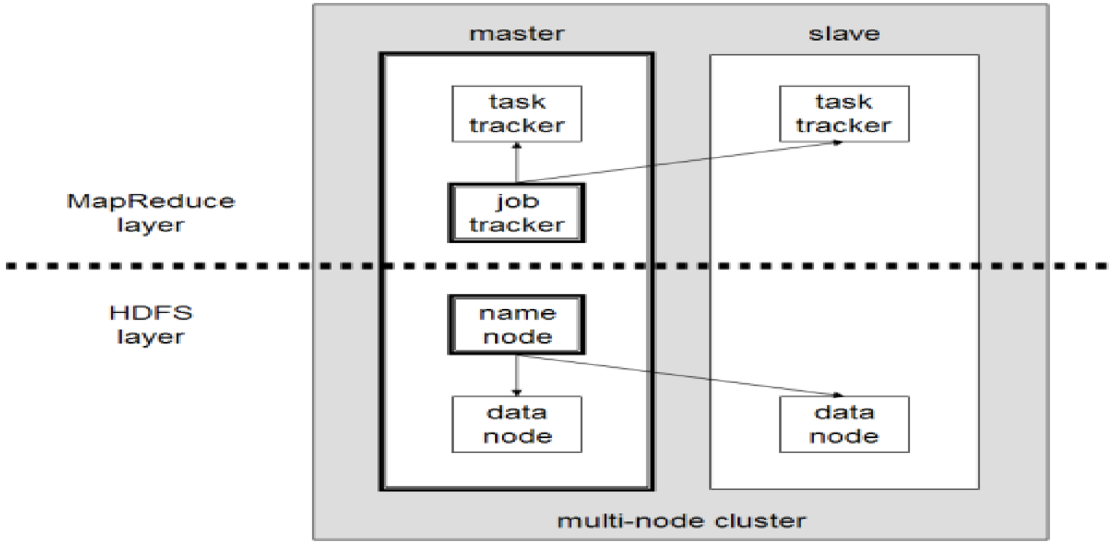


Shared-nothing

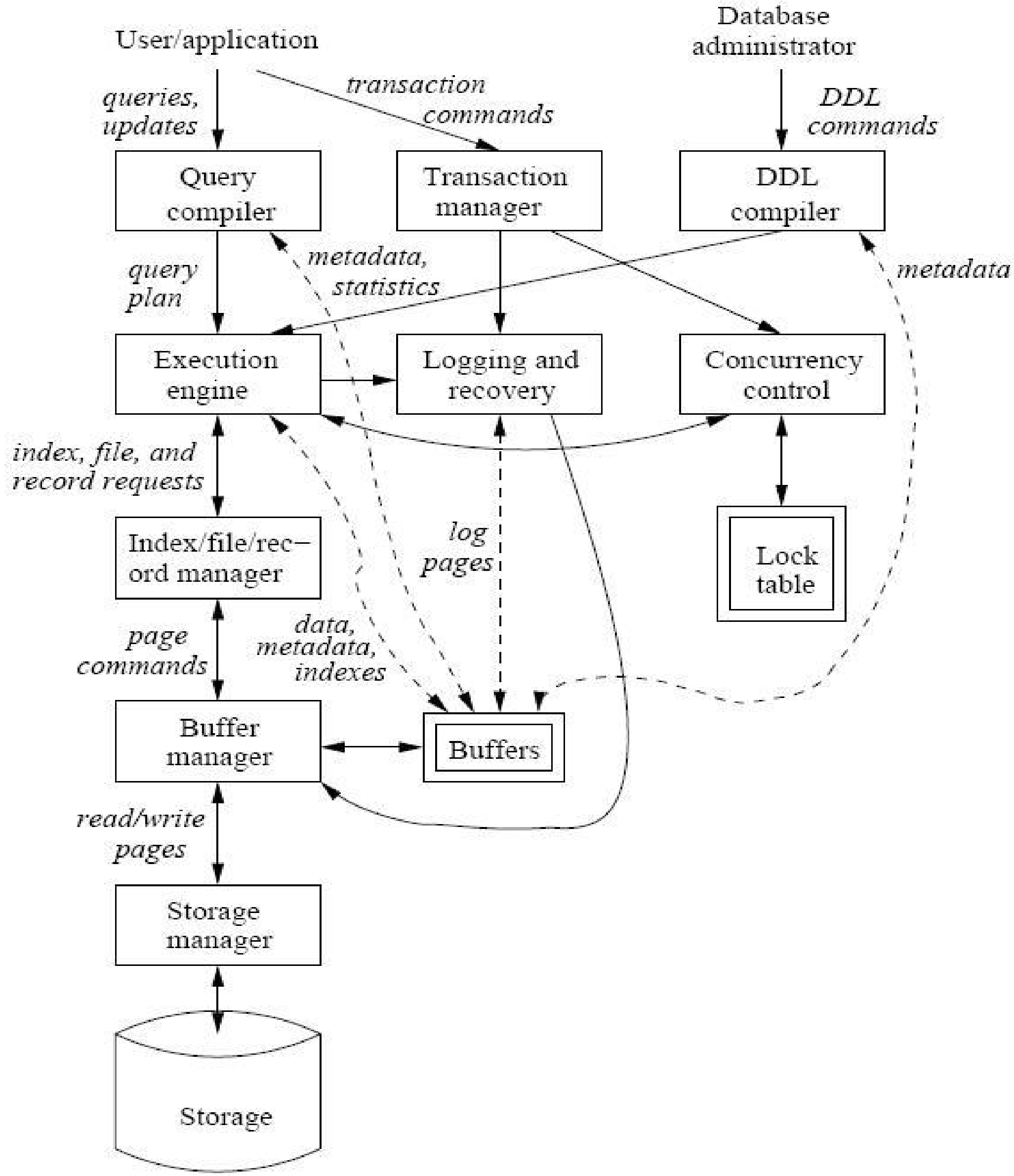


What is Hadoop (Cont'd)

- **Hadoop framework consists on two main layers**
  - Distributed file system (HDFS)
  - Execution engine (MapReduce)



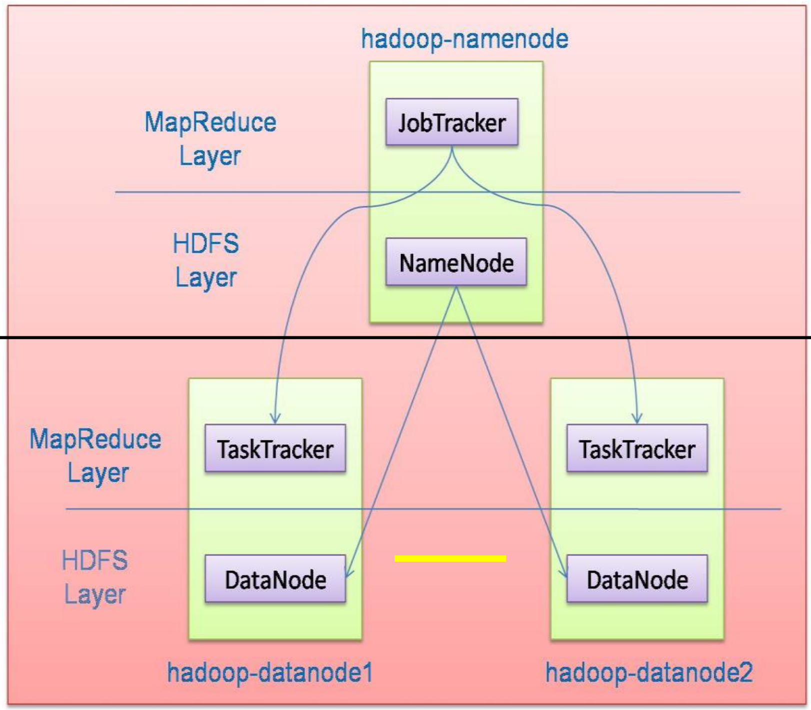
Contrast it with RDBMS



Database management system components

Hadoop Master/Slave Architecture

- Hadoop is designed as a *master-slave* *shared-nothing* architecture



Master node (single node)

Many slave nodes

Design Principles of Hadoop

- Need to process big data
- Need to parallelize computation across thousands of nodes
- **Commodity hardware**
  - Large number of low-end cheap machines working in parallel to solve a computing problem
- This is in contrast to **Parallel DBs**
  - Small number of high-end expensive machines

Design Principles of Hadoop

- **Automatic parallelization & distribution**
  - Hidden from the end-user
- **Fault tolerance and automatic recovery**
  - Nodes/tasks will fail and will recover automatically
- **Clean and simple programming abstraction**
  - Users only provide two functions “map” and “reduce”

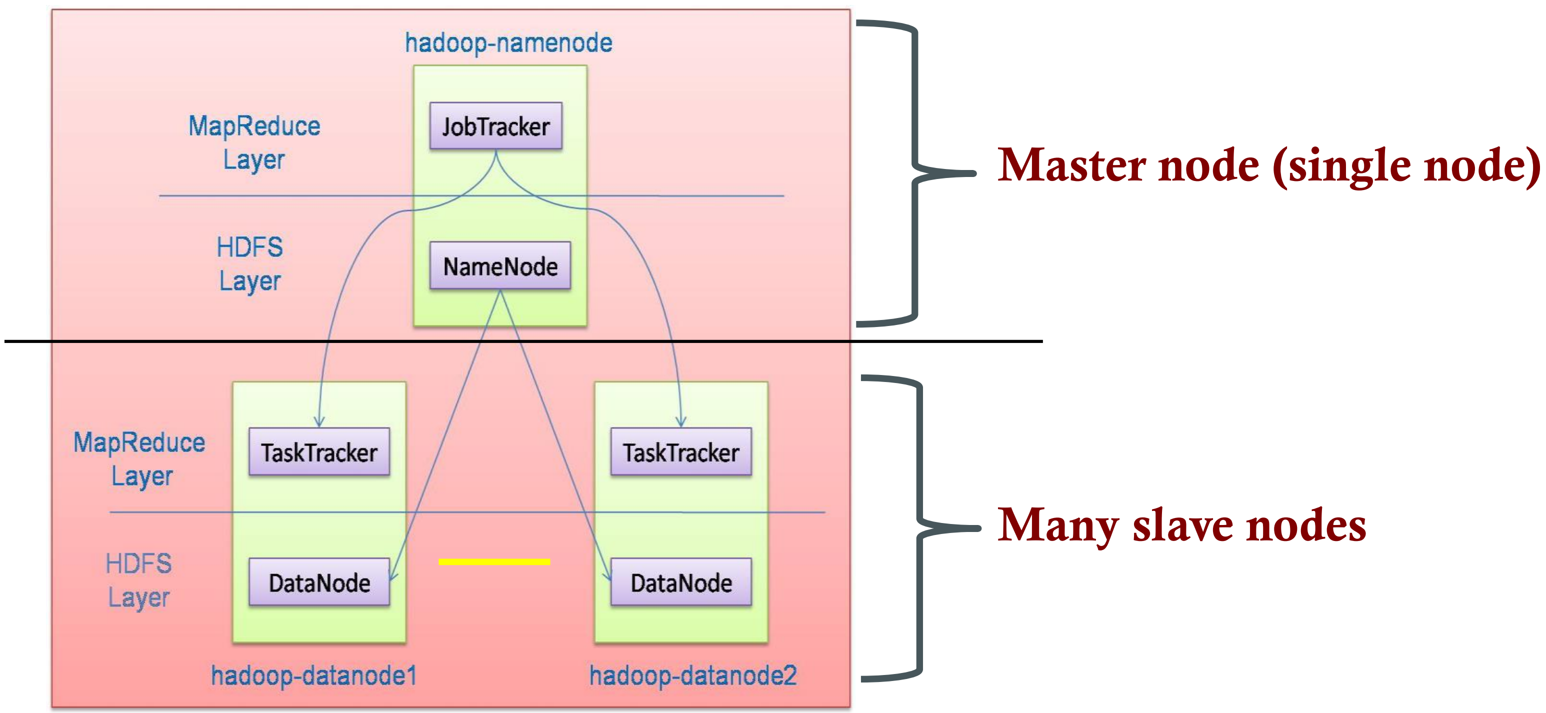
Who Uses MapReduce/Hadoop

- Google: Inventors of MapReduce computing paradigm
- Yahoo: Developing Hadoop open-source of MapReduce
- IBM, Microsoft, Oracle
- Facebook, Amazon, AOL, NetFlix
- Many others + universities and research labs

Hadoop: How it Works

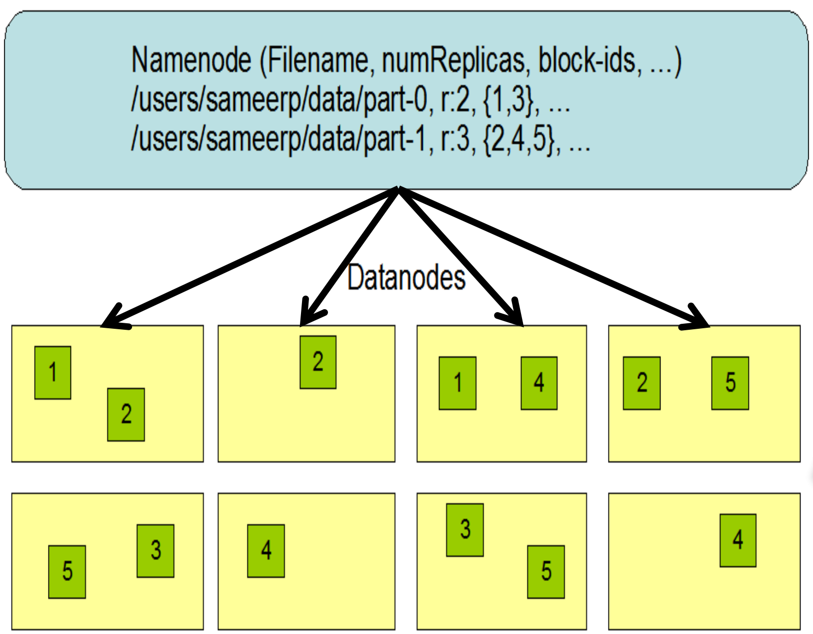
Hadoop Architecture

- Distributed file system (HDFS)
- Execution engine (MapReduce)



Hadoop Distributed File System (HDFS)

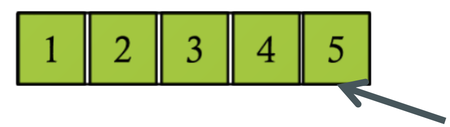
Block Replication



Centralized namenode

- Maintains metadata info about files

File *F*



Blocks (64 MB)

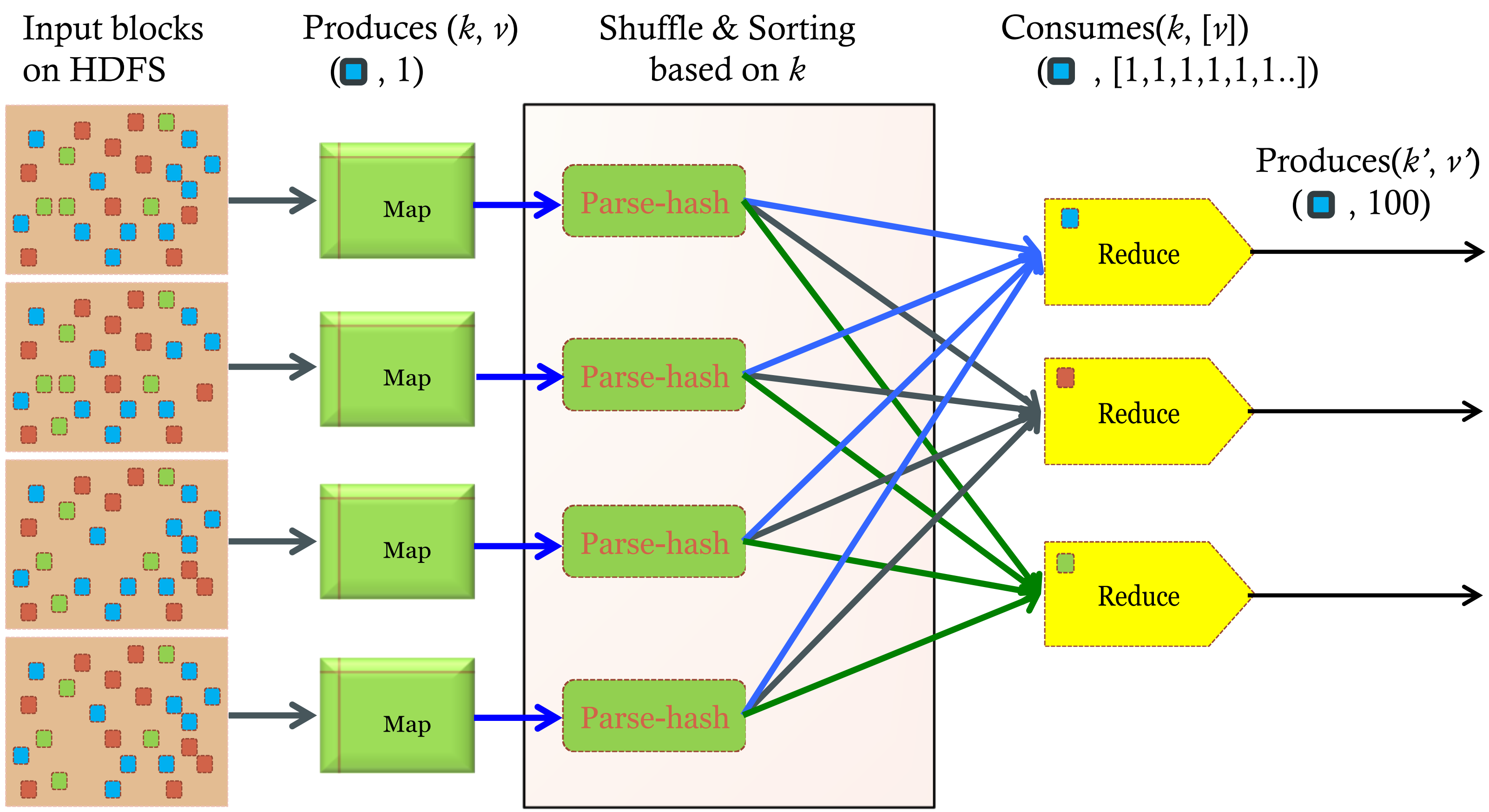
Many datanode (1000s)

- Store the actual data
- Files are divided into blocks
- Each block is replicated *N* times (Default = 3)

Main Properties of HDFS

- ***Large:*** A HDFS instance may consist of thousands of server machines, each storing part of the file system's data
- ***Replication:*** Each data block is replicated many times (default is 3)
- ***Failure:*** Failure is the norm rather than exception
- ***Fault Tolerance:*** Detection of faults and quick, automatic recovery from them is a core architectural goal of HDFS
  - Namenode is consistently checking Datanodes

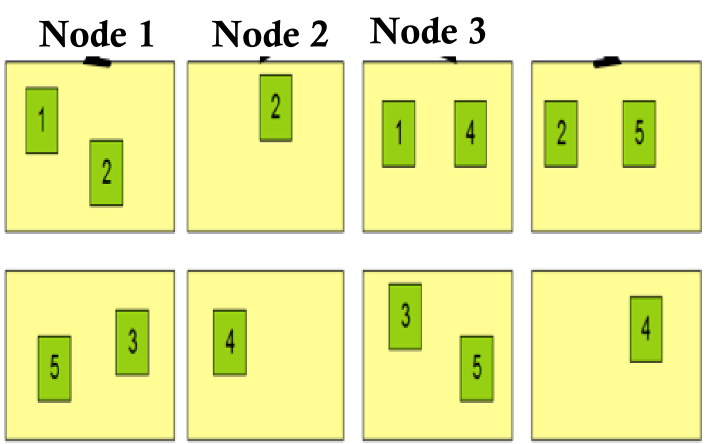
Map-Reduce Execution Engine
(Example: Color Count)



Users only provide the "Map" and "Reduce" functions

Properties of MapReduce Engine

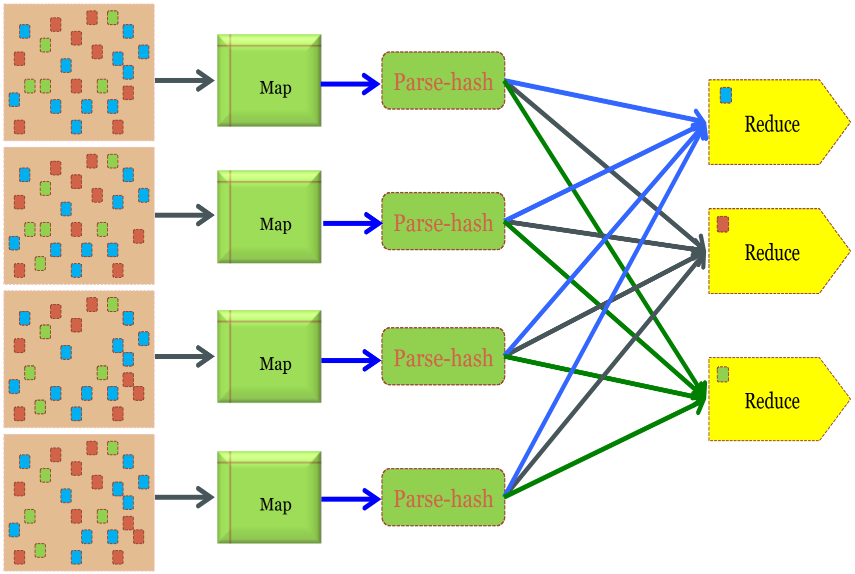
- **Job Tracker is the master node (runs with the namenode)**
  - Receives the user's job
  - Decides on how many tasks will run (number of mappers)
  - Decides on where to run each mapper (concept of locality)



- This file has 5 Blocks → run 5 map tasks
- Where to run the task reading block "1"
  - *Try to run it on Node 1 or Node 3*

Properties of MapReduce Engine (Cont'd)

- **Task Tracker is the slave node (runs on each datanode)**
  - Receives the task from Job Tracker
  - Runs the task until completion (either map or reduce task)
  - Always in communication with the Job Tracker reporting progress

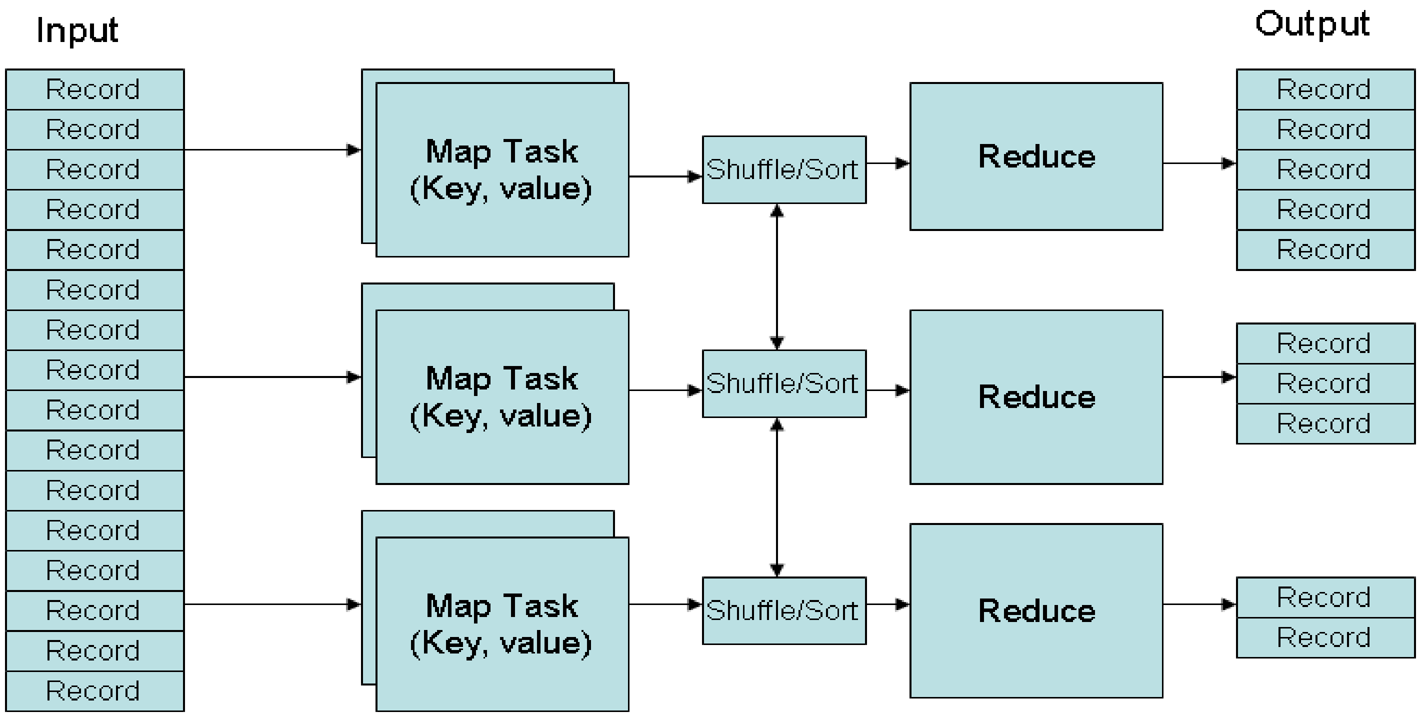


In this example, 1 map-reduce job consists of 4 map tasks and 3 reduce tasks

Key-Value Pairs

- Mappers and Reducers are users' code (provided functions)
- Just need to obey the Key-Value pairs interface
- **Mappers:**
  - Consume <key, value> pairs
  - Produce <key, value> pairs
- **Reducers:**
  - Consume <key, <list of values>>
  - Produce <key, value>
- **Shuffling and Sorting:**
  - Hidden phase between mappers and reducers
  - Groups all similar keys from all mappers, sorts and passes them to a certain reducer in the form of <key, <list of values>>

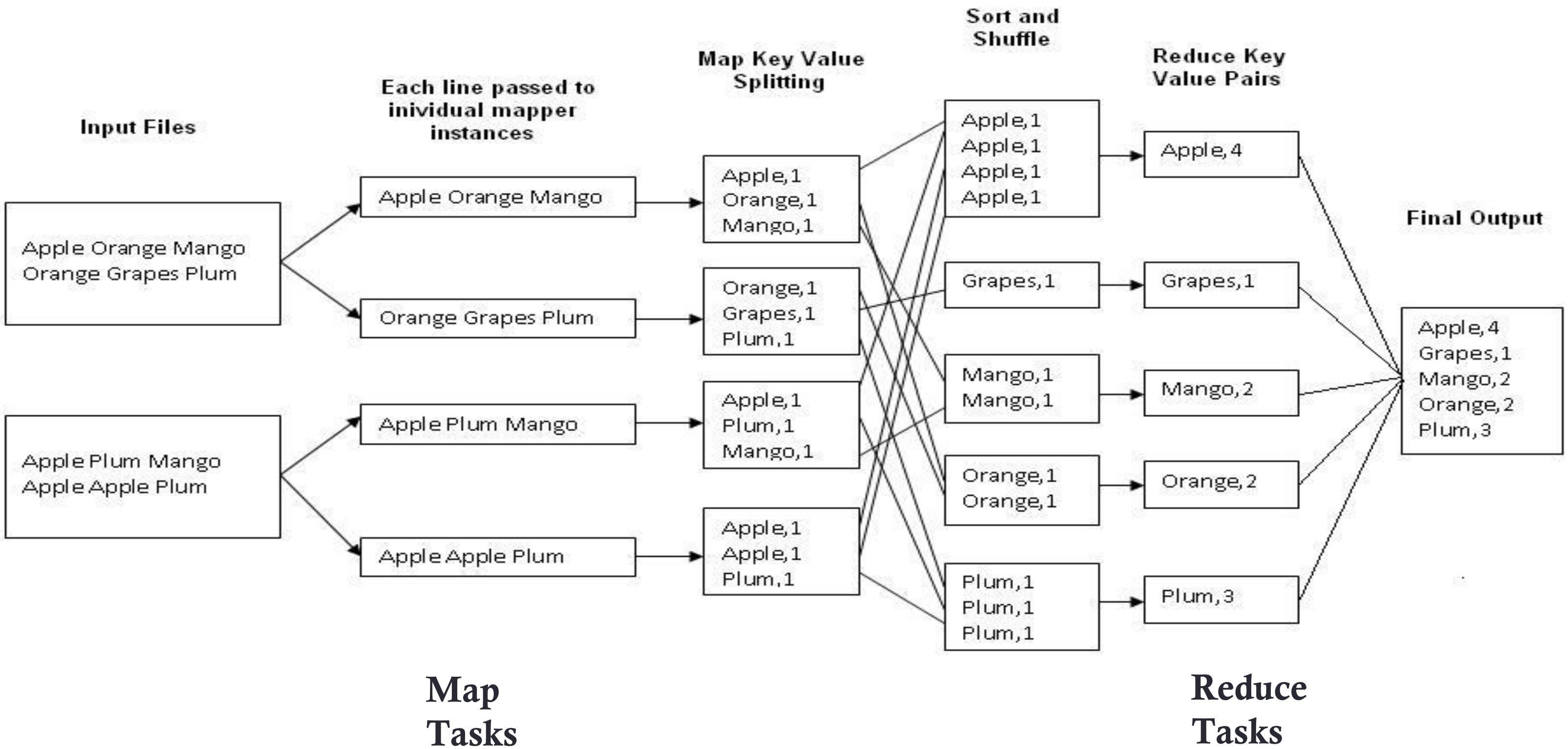
MapReduce Phases



Deciding on what will be the key and what will be the value ➔ developer's responsibility

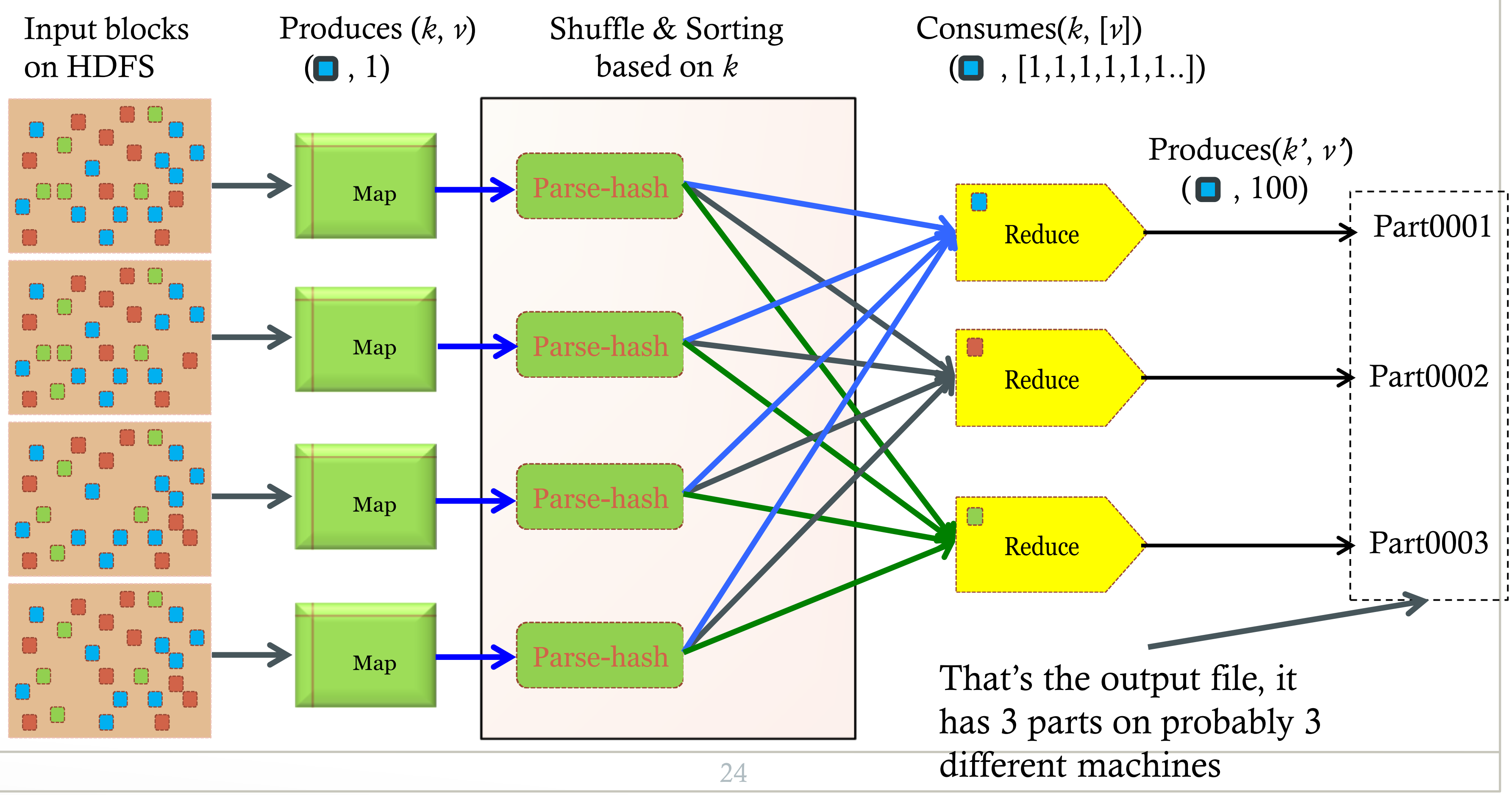
Example 1: Word Count

- **Job: Count the occurrences of each word in a data set**



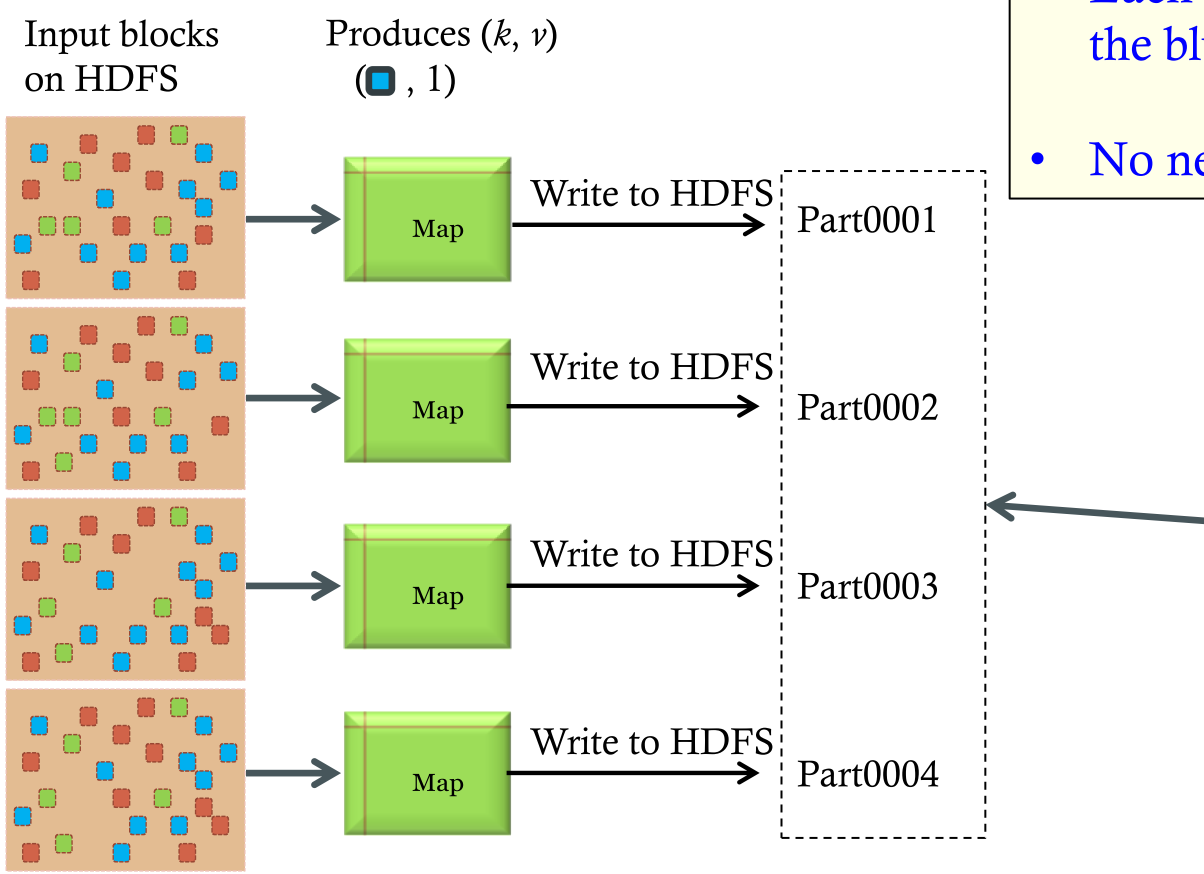
Example 2: Color Count

Job: Count the number of each color in a data set



Example 3: Color Filter

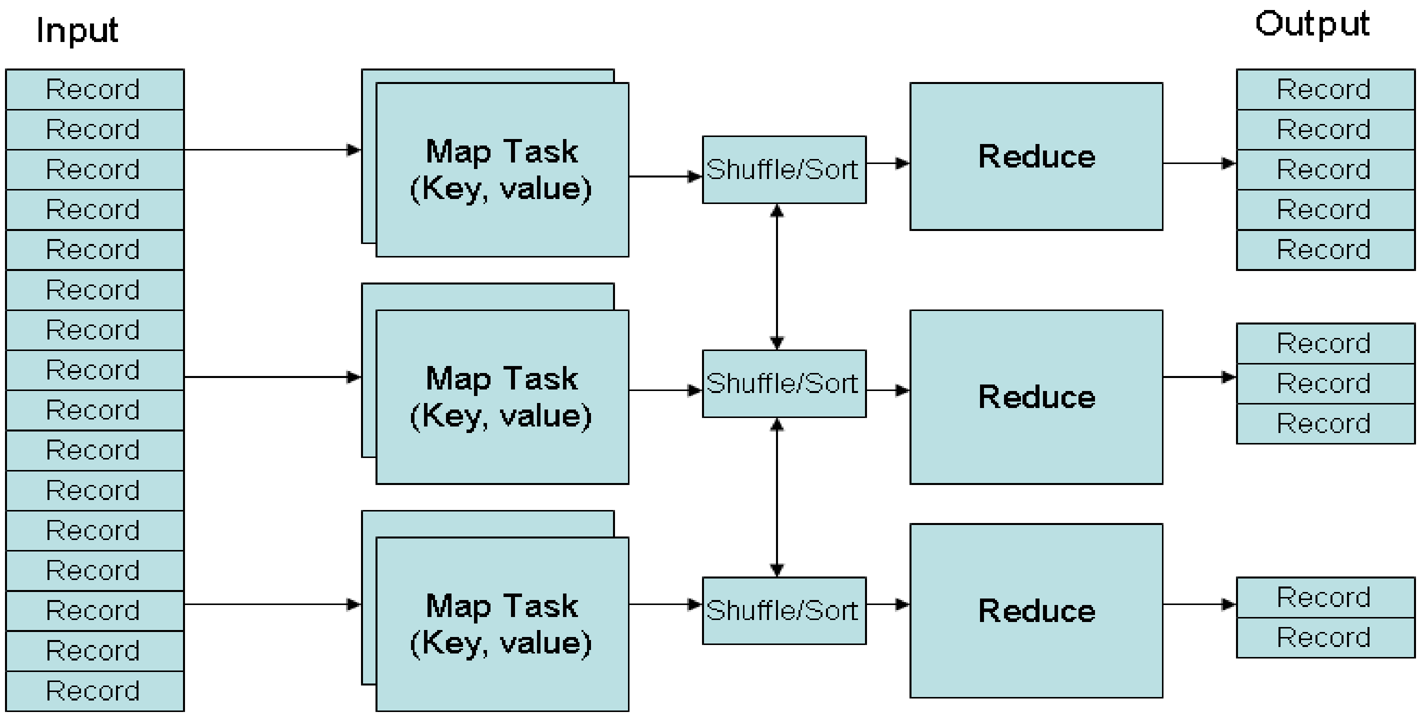
Job: Select only the blue and the green colors



- Each map task will select only the blue or green colors
- No need for reduce phase

That's the output file, it has 4 parts on probably 4 different machines

MapReduce Phases



Deciding on what will be the key and what will be the value ➔ developer's responsibility

Processing Granularity

- **Mappers**
  - Run on a record-by-record bases
  - Your code processes that record and may produce
    - Zero, one, or many outputs
- **Reducers**
  - Run on a group-of-records bases (having same key)
  - Your code processes that group and may produce
    - Zero, one, or many outputs

How it looks like in Java

File Edit Options Buffers Tools Java Help

```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);
  }}
```

--:--- **mapreduce.java** All L9 (Java/l Abbrev)--

Wrote /home/shivnath/Desktop/mapreduce.java

Provide implementation for Hadoop's Mapper abstract class

Map function

Provide implementation for Hadoop's Reducer abstract class

Reduce function

Job configuration

Optimization 1

- **In Color Count example, assume I know that the number of colors is small → can we optimize the map-side**

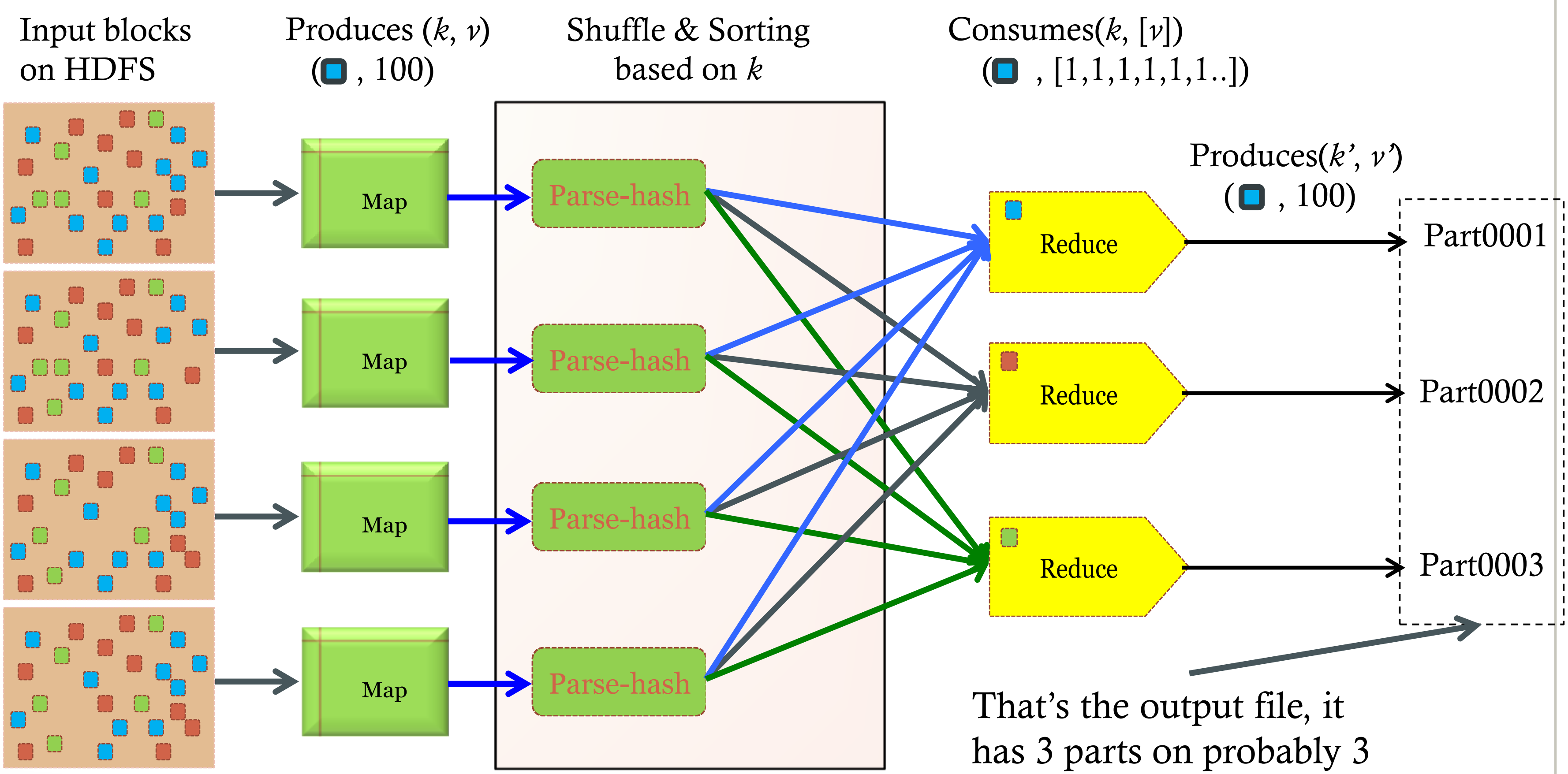
  - Each map function can have a small main-memory hash table (color, count)
  - With each line, update the hash table and produce nothing
  - When done, report each color and its local count

	10
	5
	7
	20

Gain: Reduce the amount of shuffled/sorted data over the network
Q1: Where to build the hash table?
Q2: How to know when done?

Optimization 1: Takes Place inside Mappers

Saves network messages (Typically very expensive phase)



That's the output file, it has 3 parts on probably 3 different machines

Inside the Mapper Class

Called once after all records (Here you can produce the output)

Method Summary	
protected void	`cleanup(Mapper.Context context)` Called once at the end of the task.
protected void	`map(KEYIN key, VALUEIN value, Mapper.Context context)` Called once for each key/value pair in the input split.
void	`run(Mapper.Context context)` Expert users can override this method for more complete control over the execution of the Mapper.
protected void	`setup(Mapper.Context context)` Called once at the beginning of the task.

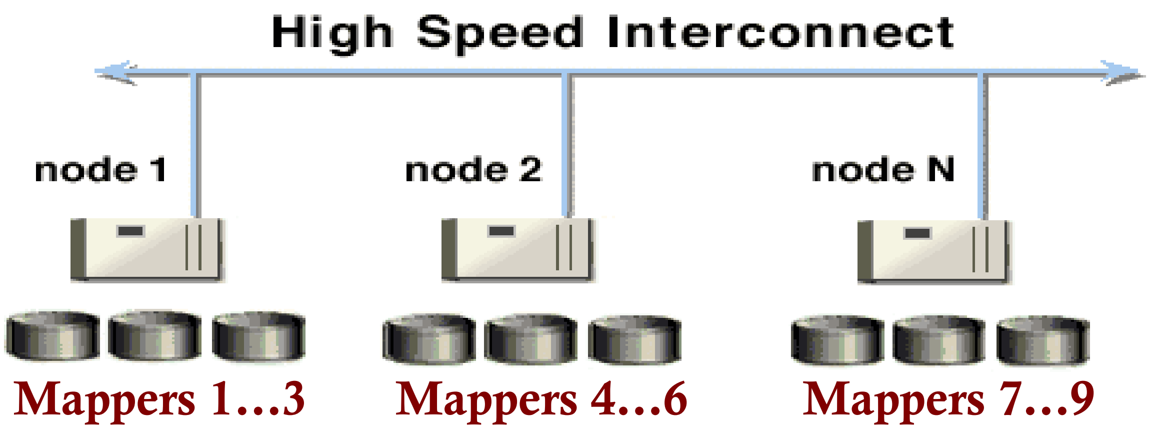
Called for each record

Called once before any record (Here you can build the hash table)

Reducer has similar functions…

Optimization 2: Map-Combine-Reduce

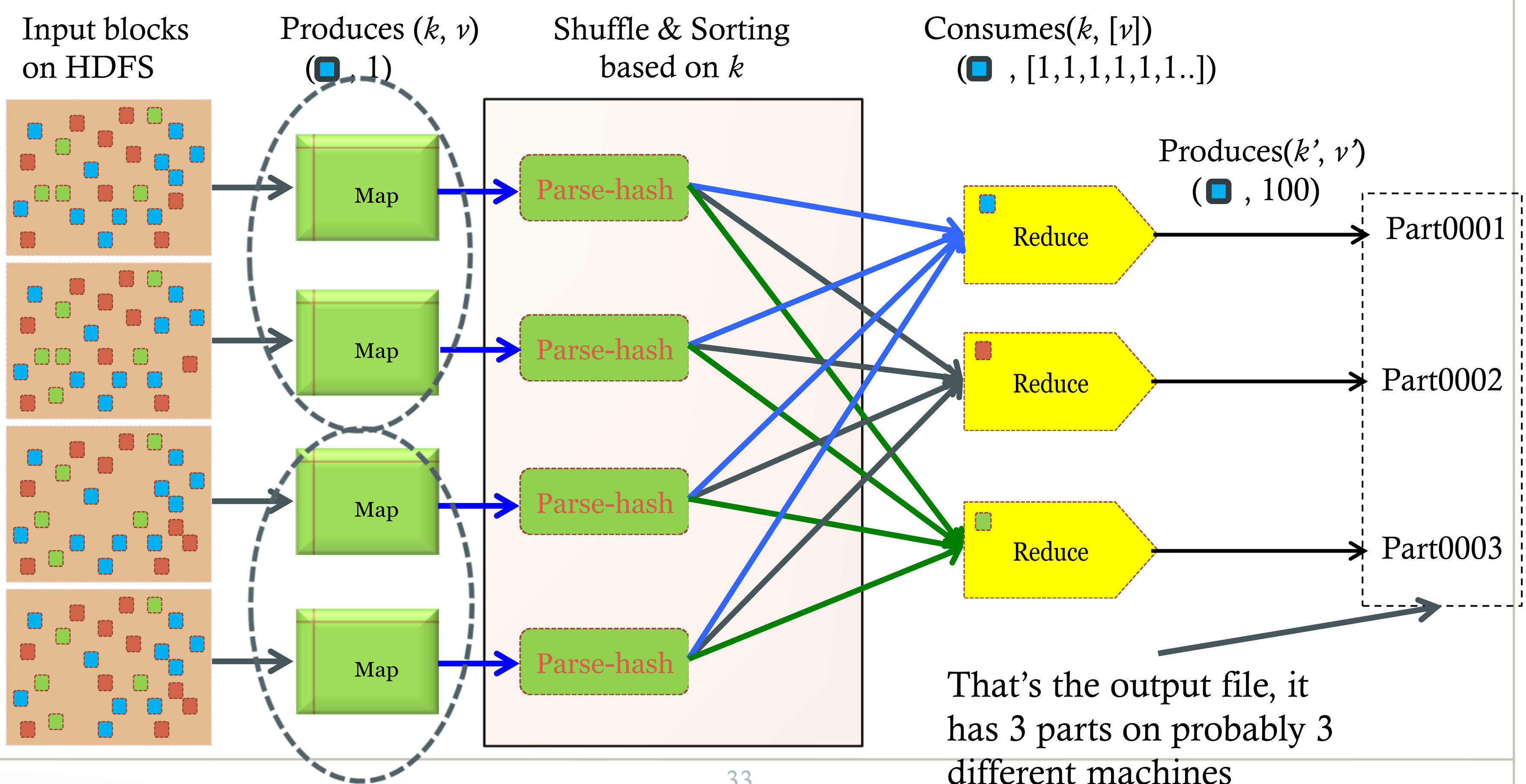
- What about partially aggregating the results from mappers on each machine



- A ***combiner*** is a ***reducer*** that runs on each machine to partially aggregate (*that's a user code*) mappers' outputs from this machine
- Then, combiners output is shuffled/sorted for reducers

Optimization 2: Outside Mappers, But on Each Machine

Combiner runs on each node to partially aggregate the local mappers' output



Tell Hadoop to use a Combiner

File Edit Options Buffers Tools Java Help

```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);
  }}
```

```
--:---  mapreduce.java   All L9    (Java/l Abbrev)----------------------------------------
Wrote /home/shivnath/Desktop/mapreduce.java
```

Not all jobs can use a combiner

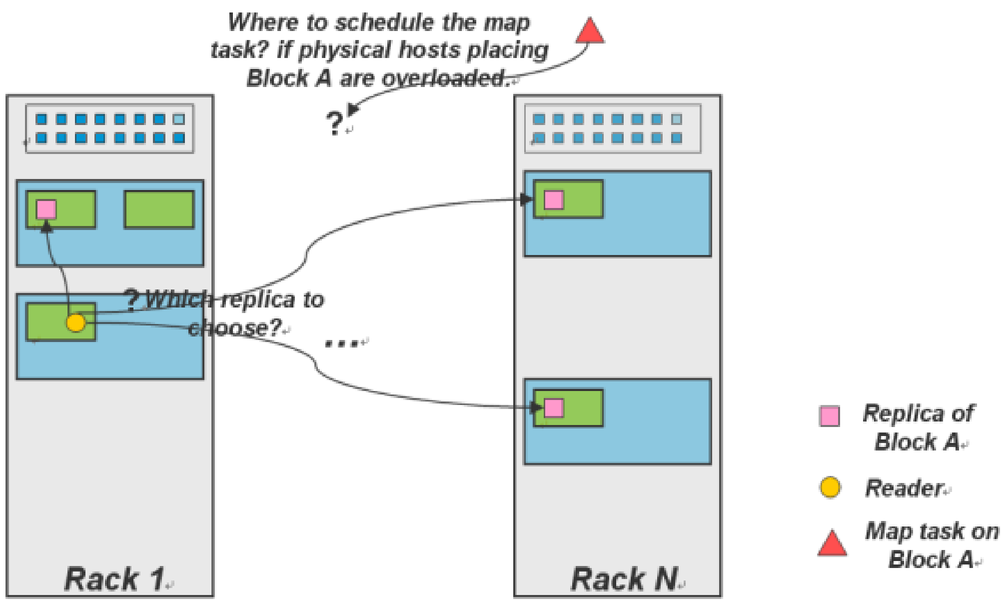
Use a combiner

Optimizations 3: Speculative Execution

- If one node is slow, it will slow the entire job
- **Speculative Execution:** Hadoop automatically runs each task multiple times in parallel on different nodes
  - First one finishes, the others will be killed

Optimizations 4: Locality

- **Locality:** try to run the map code on the same machine that has the relevant data
  - If not possible, then machine in the same rack
  - Best effort, no guarantees



Translating DB Operations to Hadoop Jobs

DB Operations

- Select (Filter)
- Projection
- Grouping and aggregation
- Duplicate Elimination
- Join

Selection: σ

- **Select: σ_c *(R)*:**
  - *c* is a condition on R's attributes
  - Select subset of tuples from R that satisfy **selection condition *c***

R

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

$\sigma_{((A=B) \wedge (D>5))}$ **(R)**

A	B	C	D
α	α	1	7
β	β	23	10

Select *
From R
Where R.A = R.B
And R.D > 5;

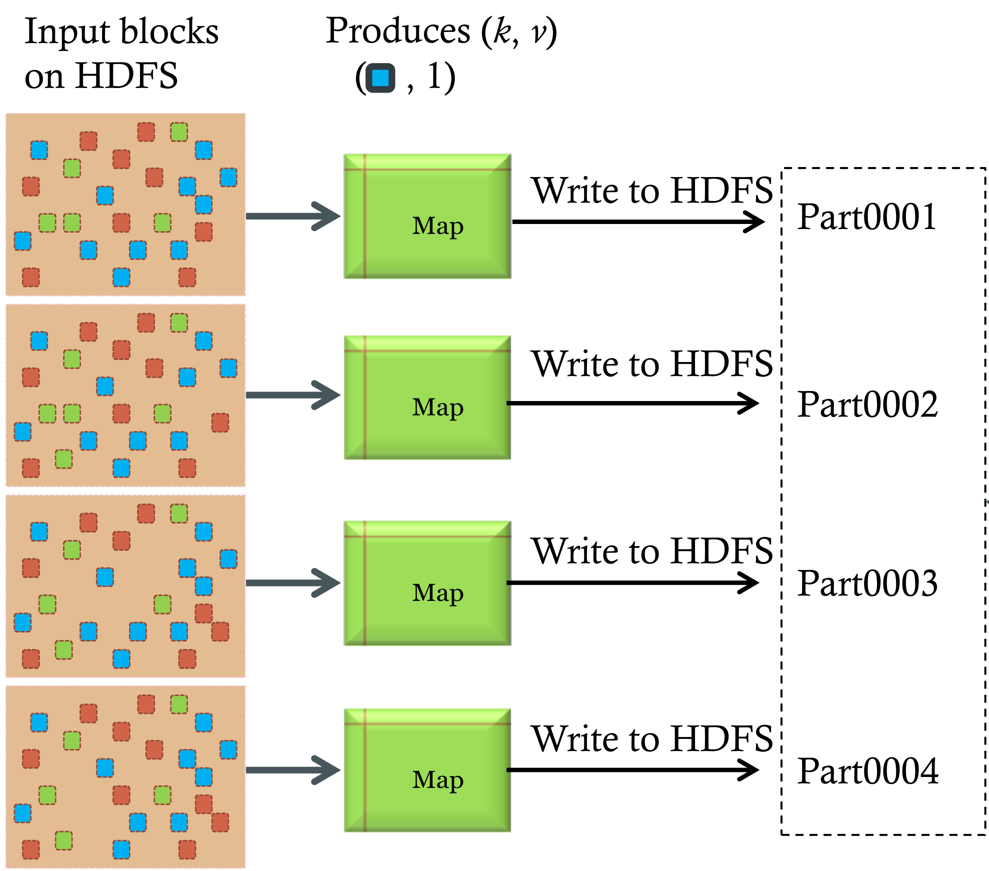
$\sigma_{(D > C)}$ **(R)**

A	B	C	D
α	α	1	7
α	β	5	7

In Hadoop, Selection is implemented as a Map-Only Job

Back to Color Filter

Job: Select only the blue and the green colors



- Each map task will select only the blue or green colors
- No need for reduce phase

That's the output file, it has 4 parts on probably 4 different machines

Projection: π

- $\pi_{A1, A2, \ldots, An}$ (R), with A1, A2, …, An $\in$ attributes A_R
  - returns all tuples in R, but only columns A1, A2, …, An
- A1, A2, …, An are called ***Projection List***

Rename column A to V

Compute this expression and call it X

$\pi_{C,\ V \leftarrow A,\ X \leftarrow C*3+B}$ (R)

Select C, A as V, C*3+B as X
From R;

R

A	B	C
1	2	5
3	4	6
1	2	7
1	2	8

C	V	X
5	1	17
6	3	22
7	1	23
8	1	26

In Hadoop, Projection is implemented as a Map-Only Job

Grouping & Aggregation

- **Aggregation function** takes a collection of values and returns a single value as a result
  - **avg**: average value
  - **min**: minimum value
  - **max**: maximum value
  - **sum**: sum of values
  - **count**: number of values

- **Grouing & Aggregate operation** in relational algebra
  - $\gamma_{g1,g2,\ \ldots gm,\ F1(A1),\ F2(A2),\ \ldots Fn(An)}(R)$
  - *R* is a relation or any relational-algebra expression
  - *g1, g2, …gm* is a list of attributes on which to group (can be empty)
  - Each ***Fi*** is an aggregate function applied on attribute ***Ai*** within each group

Grouping & Aggregation Operator: Example

R

A	B	C
α	α	7
α	β	7
β	β	3
β	β	10

S

branch_name	*account_number*	*balance*
Perryridge	A-102	400
Perryridge	A-201	900
Brighton	A-217	750
Brighton	A-215	750
Redwood	A-222	700

$\gamma_{\text{sum(c)}}$**(R)**

sum(*c*)
27

$\gamma_{\text{branch\_name,sum(balance)}}$**(S)**

branch_name	**sum**(*balance*)
Perryridge	1300
Brighton	1500
Redwood	700

Select sum(C)
From R;

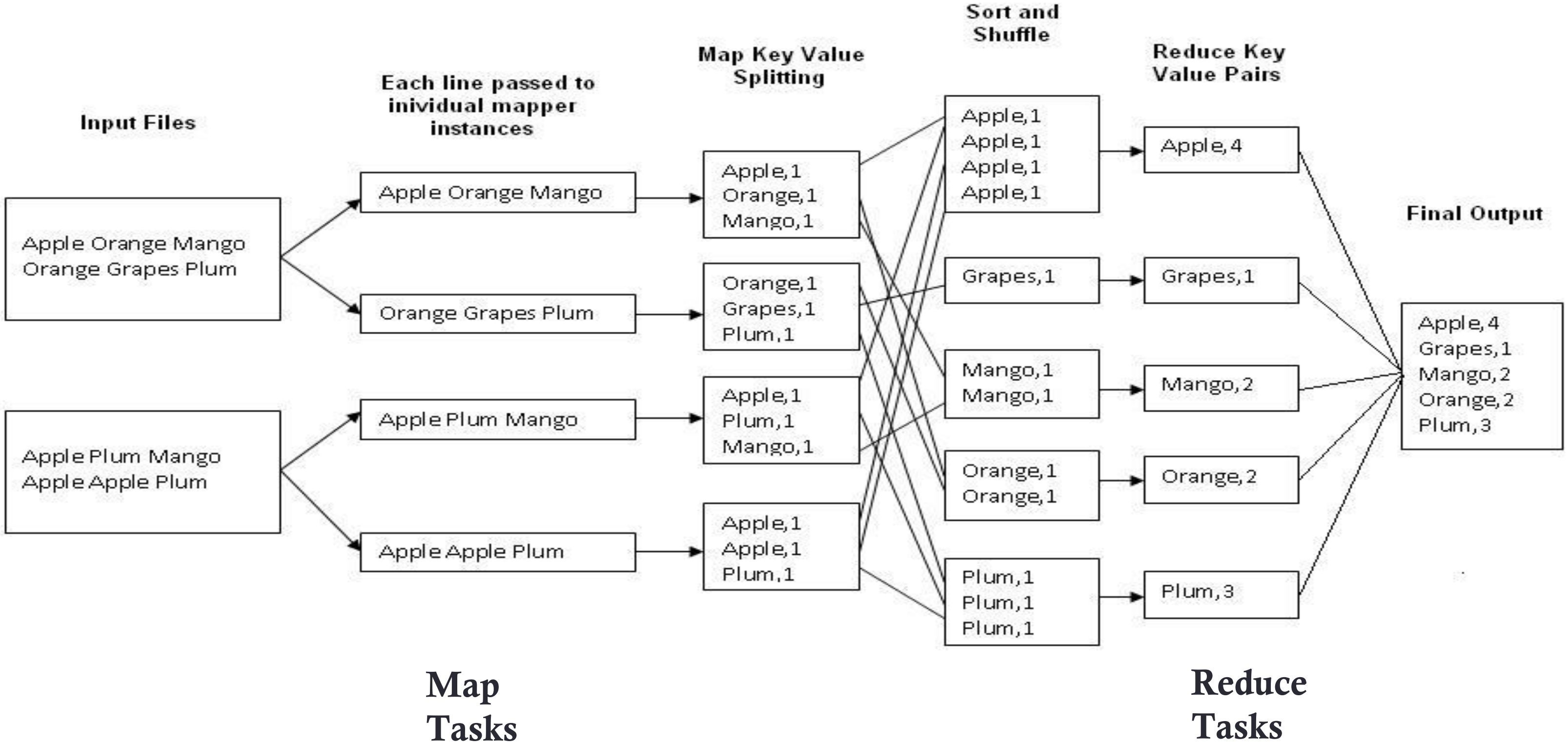
Select sum(balance)
From S
Group By branch_name;

In Hadoop, Grouping & Aggregation is implemented as a Map-Reduce Job

What is the key/value?

Back to Word Count

- **Job: Count the occurrences of each word in a data set**



Duplicate Elimination: δ (R)

- Delete all duplicate records
- Convert a bag to a set

Select Distinct *
From R;

R

A	B
1	2
3	4
1	2
1	2

d (R)

A	B
1	2
3	4

In Hadoop, duplicate elimination is implemented as a Map-Reduce Job

What is the key/value?

Join: $R \bowtie_C S$

- Theta Join is cross product, with condition C
- It is defined as : $R \bowtie_C S = (\sigma_C (R \times S))$

R

A	B
1	2
3	2

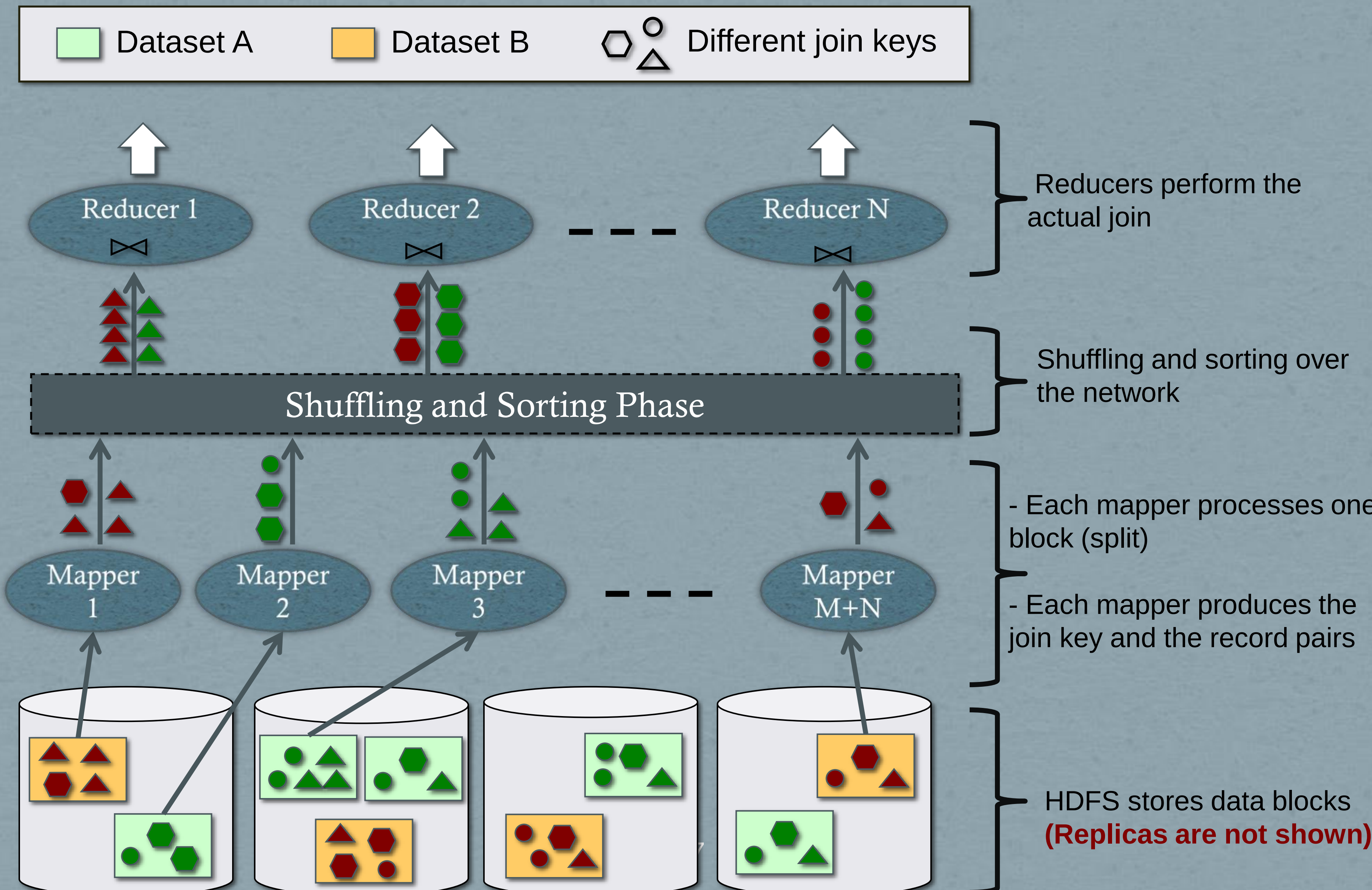
S

D	C
2	3
4	5
4	5

$R \bowtie_{R.A>=S.C} S$

A	B	D	C
3	2	2	3

Joining Two Large Datasets: Re-Partition Join

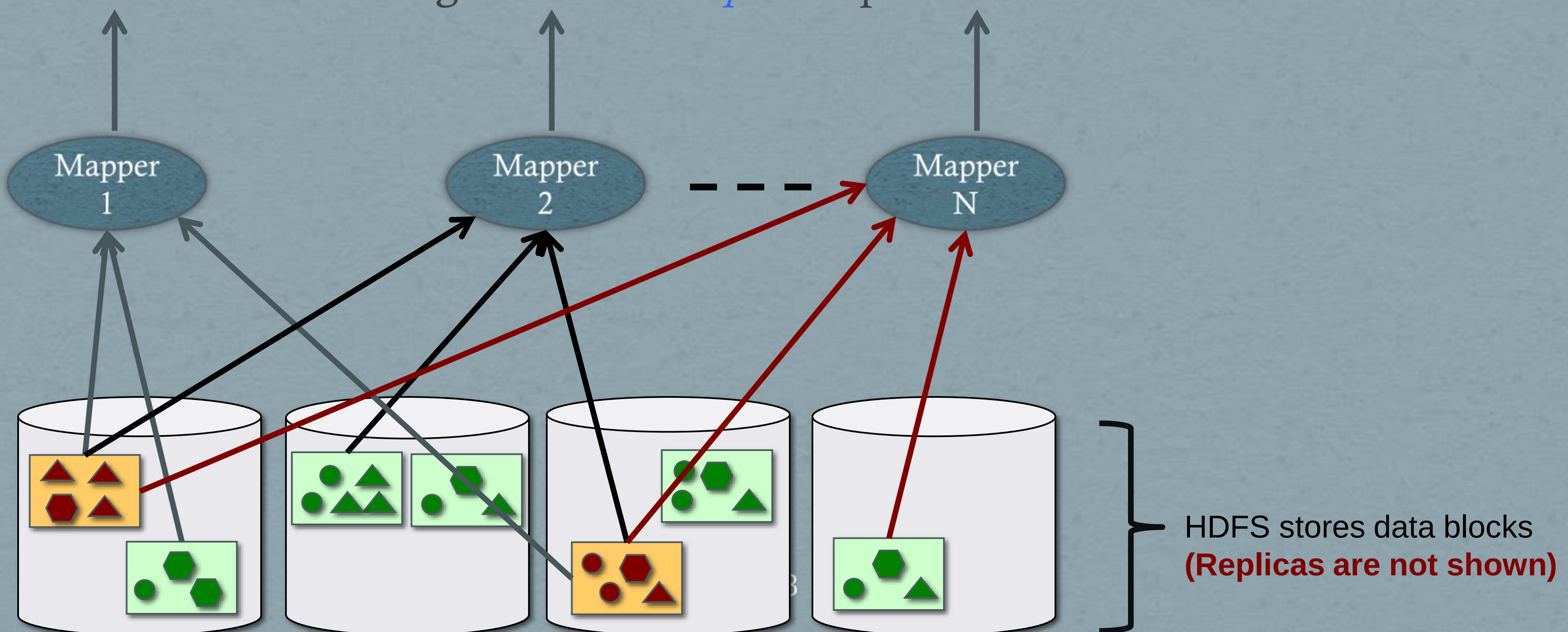


Joining Large Dataset (A) with Small Dataset (B)

Broadcast/Replication Join

Dataset A   Dataset B   Different join keys

- Every map task processes one block from A and the entire B
- Every map task performs the join *(MapOnly job)*
- Avoid the shuffling and reduce *expensive* phases



Translating DB Operations to Hadoop Jobs (**Summary**)

- Select (Filter) → Map-only job
- Projection → Map-only job
- Grouping and aggregation → Map-Reduce job
- Duplicate Elimination → Map-Reduce job
  - **Map (Key= hash code of the tuple, Value= tuple itself)**
- Join → Map-Reduce job

Hadoop Black-Box Model

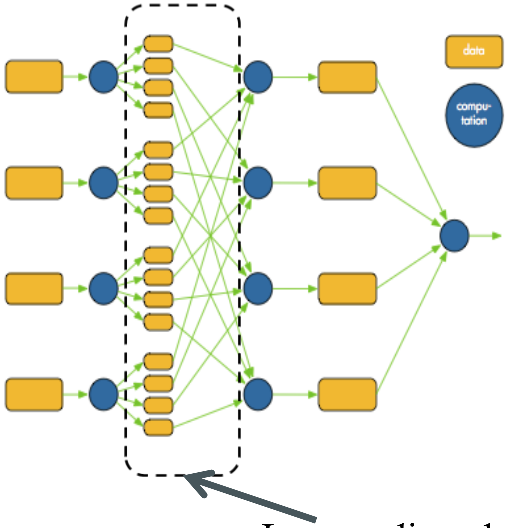
- **DBMSs have an Open-Box Model**
  - Data are known (DB Schema)
  - Queries are known (written in SQL)

- **Hadoop has a Black-Box Model**
  - Data are not known (files of unknown structure)
  - Jobs are also unknown (written in java)

Hadoop does very limited optimizations

Hadoop Fault Tolerance

- Intermediate data between mappers and reducers are *materialized* to simple & straightforward fault tolerance



Intermediate data (materialized)

- **What if a task fails (map or reduce)?**
  - Tasktracker detects the failure
  - Sends message to the jobtracker
  - Jobtracker re-schedules the task
- **What if a datanode fails?**
  - Both namenode and jobtracker detect the failure
  - All tasks on the failed node are re-scheduled
  - Namenode replicates the users' data to another node
- **What if a namenode or jobtracker fails?**
  - The entire cluster is down

Simplicity Comes From…

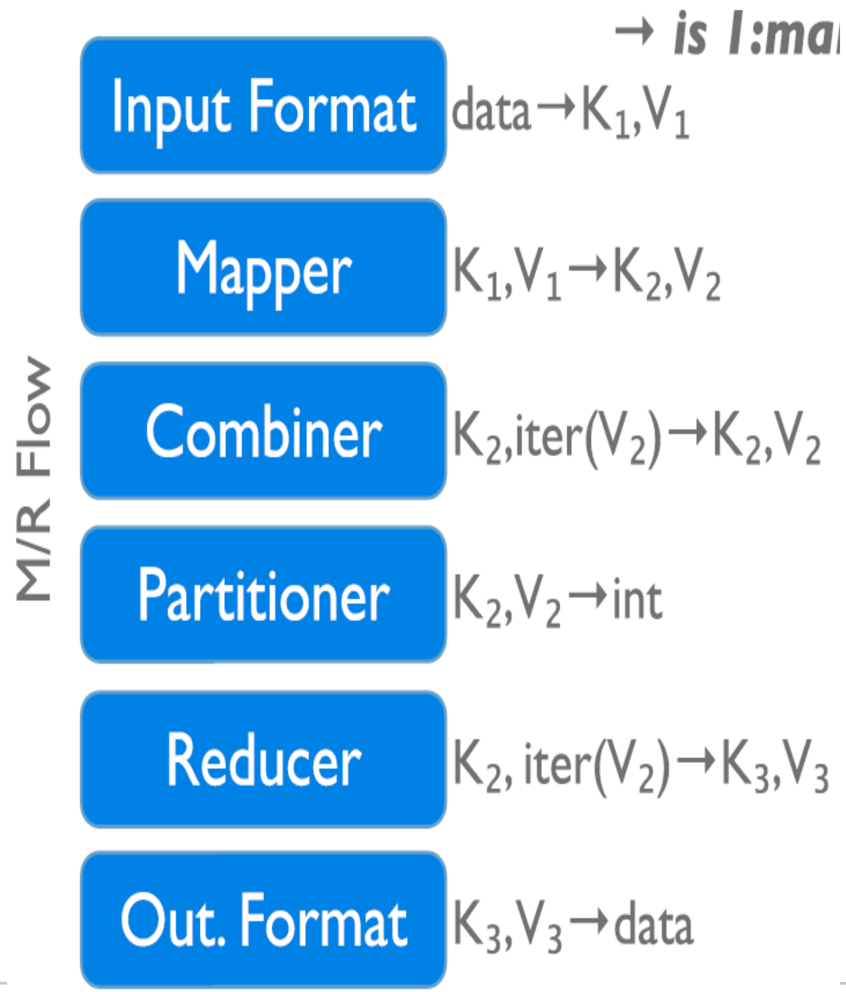
- Jobs are read-only (do not change or modify data)

- Intermediate data between mappers & reducers is materialized until job finishes

More About Execution Phases

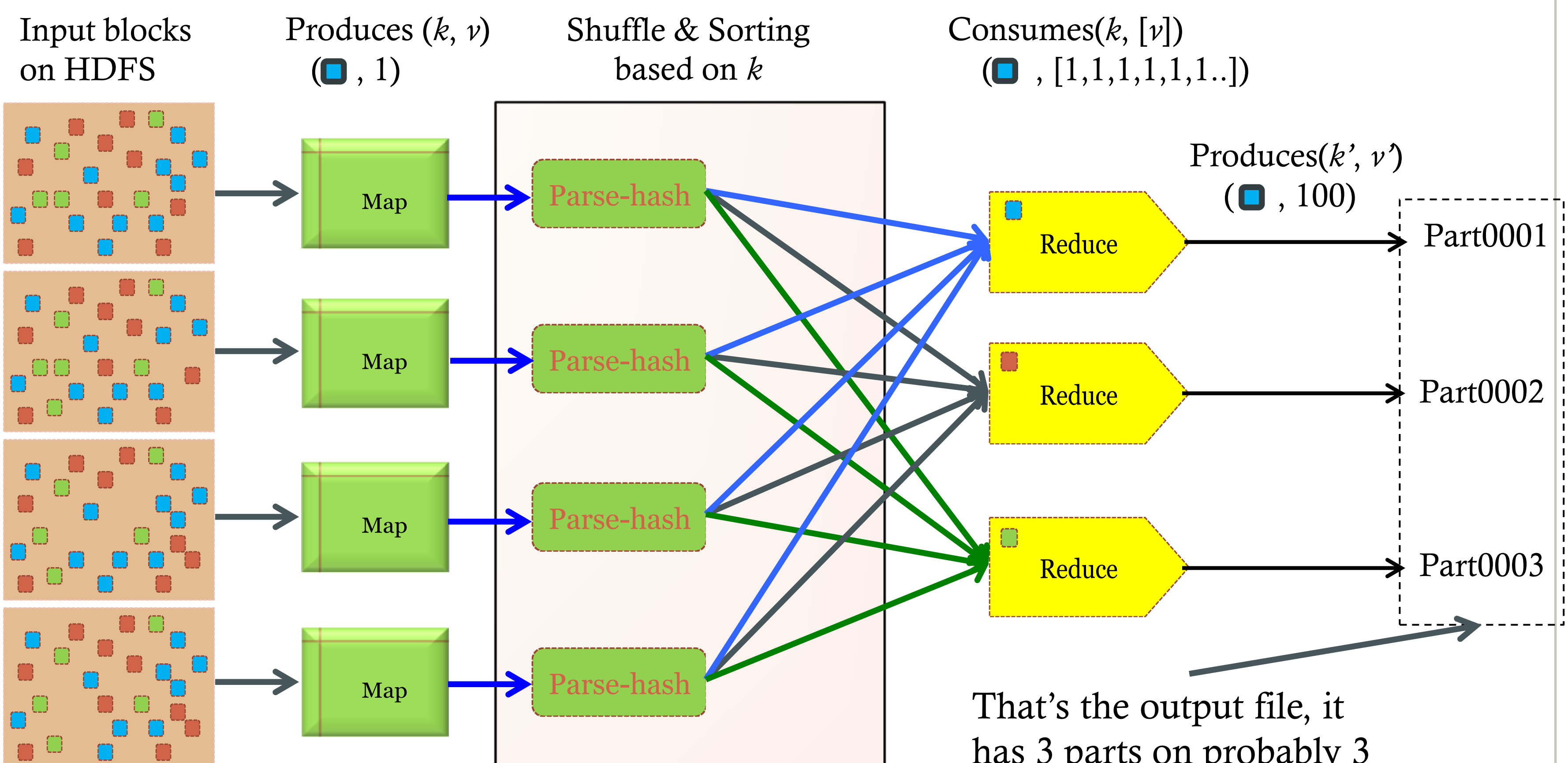
Execution Phases

- InputFormat
- Map function
- Partitioner
- Sorting & Merging
- Combiner
- Shuffling
- Merging
- Reduce function
- OutputFormat



Reminder about Covered Phases

Job: Count the number of each color in a data set



That's the output file, it has 3 parts on probably 3 different machines

Partitioners

- **The output of the mappers need to be partitioned**
  - # of partitions = # of reducers
  - The same key in all mappers must go to the same partition (and hence same reducer)

- **Default partitioning is hash-based**

- **Users can customize it as they need**

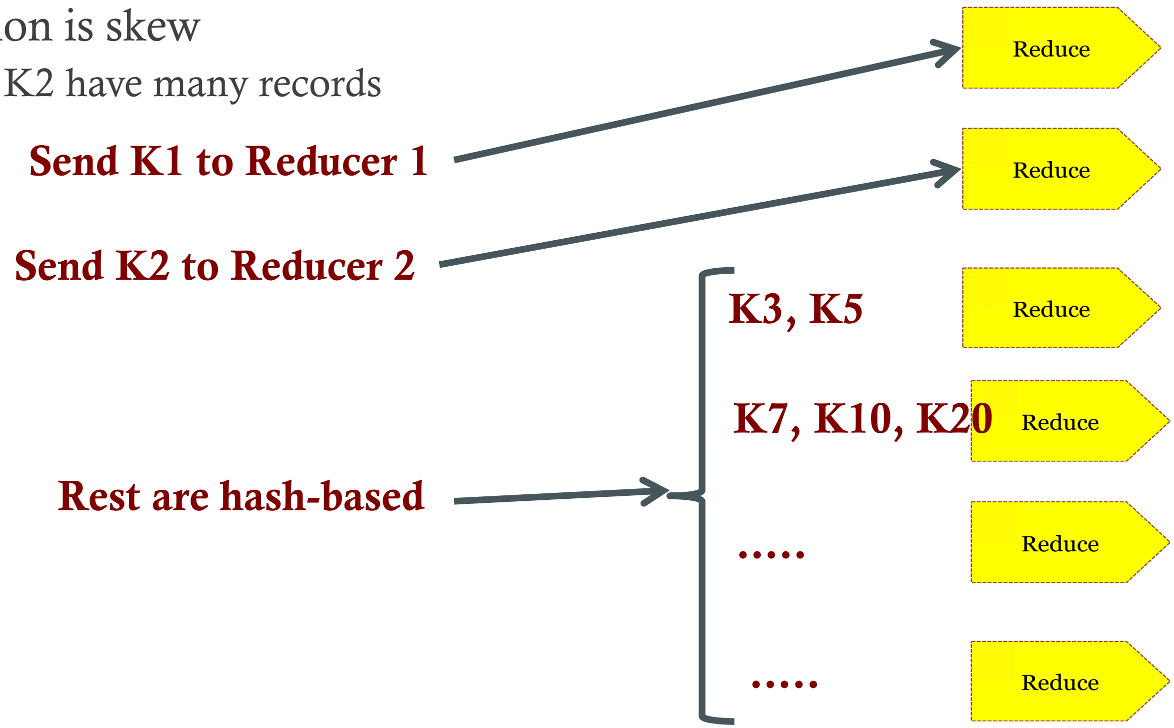
Customized Partitioner

```java
package org.apache.hadoop.examples.textpair;

import org.apache.hadoop.io.Writable;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.Partitioner;

/**
 * the hash partitioner
 *
 * @author yingyib
 *
 */
public class FirstPartitioner implements Partitioner<TextPair, Writable> {

        @Override
        public void configure(JobConf job) {
        }

        @Override
        public int getPartition(TextPair key, Writable value, int numPartitions) {
                return Math.abs(key.getFirst().hashCode()) % numPartitions;
        }
}
```

Returns a partition Id

Optimization: Balance the Load among Reducers

- Assume we have N reducers
- Many keys {k1, k2, …, Km}
- Distribution is skew
  - K1 and K2 have many records

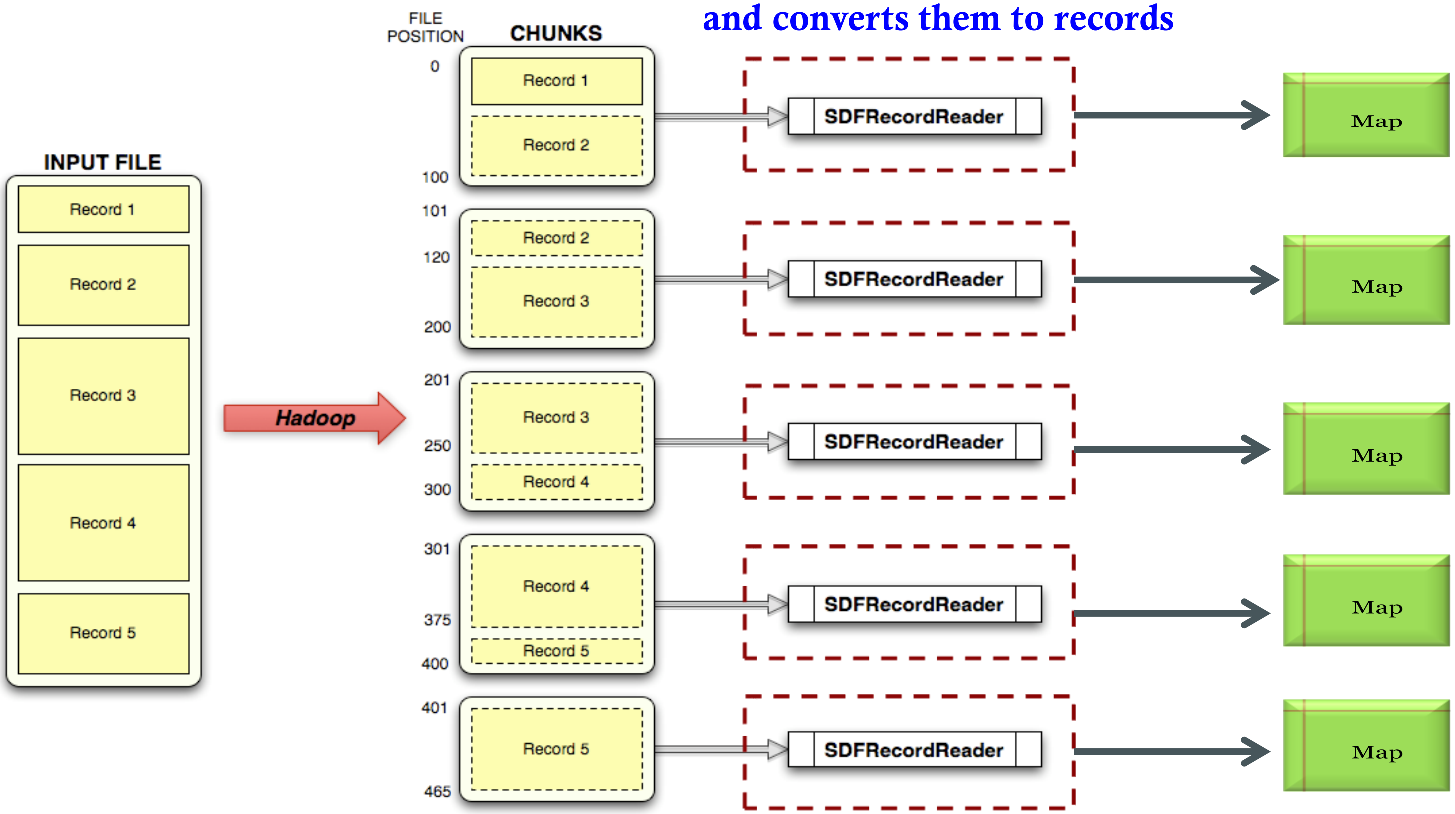


Input/Output Formats

- **Hadoop's data model ← Any data in any format will fit**
  - Text, binary, in a certain structure
- **How Hadoop understands and reads the data ??**
- **The *input format* is the piece of code that understands the data and how to reads it**
  - Hadoop has several built-in input formats to use
  - Text files, binary sequence files

Input Formats

Record reader reads bytes
and converts them to records



Tell Hadoop which Input/Output Formats

File Edit Options Buffers Tools Java Help

```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);
  }}
```

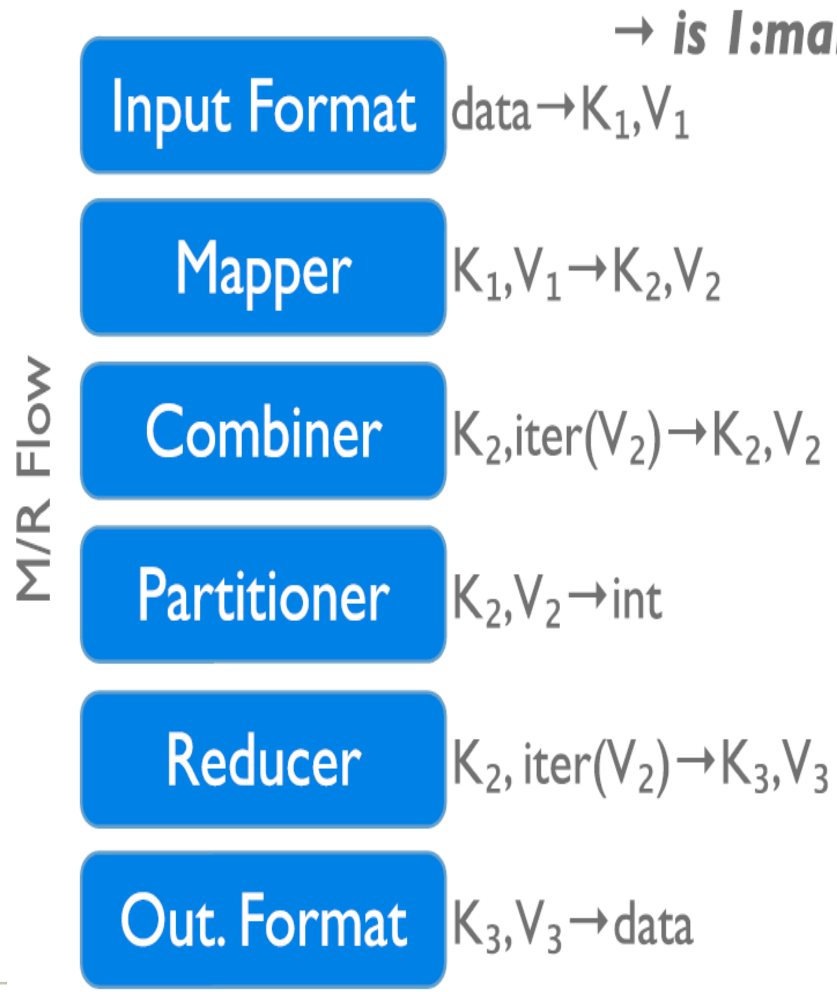
--:--- **mapreduce.java** All L9 (Java/l Abbrev)--

Wrote /home/shivnath/Desktop/mapreduce.java

Define the formats

We Covered All Execution Phases

- **InputFormat**
- **Map function**
- **Partitioner**
- **Sorting & Merging**
- **Combiner**
- **Shuffling**
- **Merging**
- **Reduce function**
- **OutputFormat**

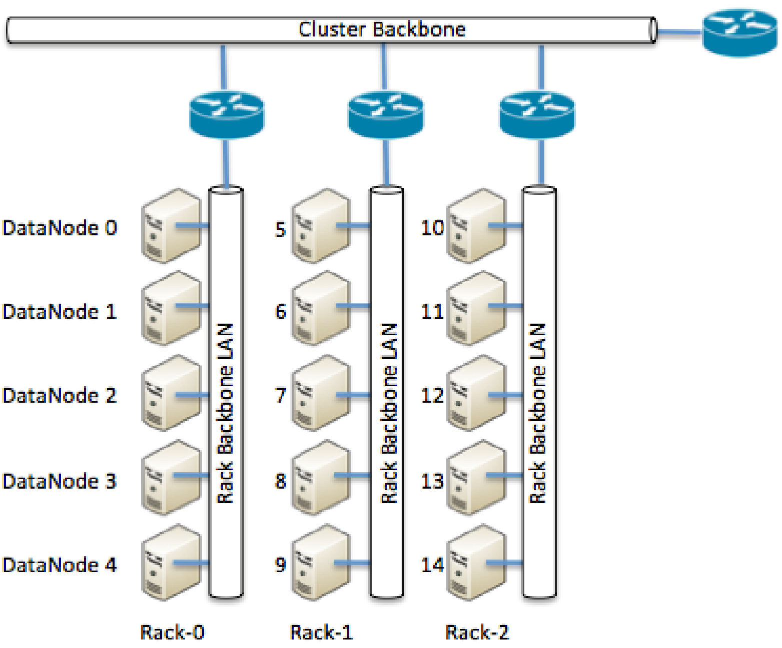


More on HDFS

HDFS and Placement Policy

Default Placement Policy

- ***First copy*** is written to the node creating the file (write affinity)
- ***Second copy*** is written to a data node within the same rack
- ***Third copy*** is written to a data node in a different rack
- **Objective:** load balancing & fault tolerance

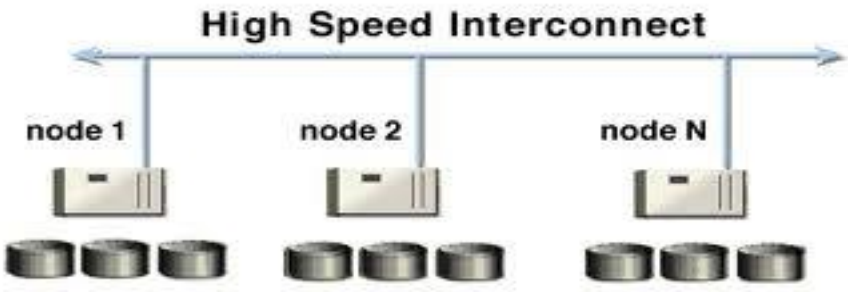


Rack-aware replica placement

Safemode Startup

- On startup Namenode enters Safemode (few seconds).
- Each DataNode checks in with Heartbeat and BlockReport.
- Namenode verifies that each block has acceptable number of replicas
- If things are fine ➔ Namenode exits Safemode
- If some blocks are under replicated
  - Replicate these blocks to other Datanodes
  - Then, exit safemode

The Communication Protocol



- All HDFS communication protocols are layered on top of the TCP/IP protocol
- A client establishes a connection to a configurable TCP port on the Namenode machine. It talks ClientProtocol with the Namenode
- The Datanodes talk to the Namenode using Datanode protocol
- **File transfers are done directly between datanodes**
  - **Does not go though the namenode**

Configuration

- Several files control Hadoop's cluster configurations
  - **Mapred-site.xml:** map-reduce parameters
  - **Hdfs-site.xml:** HDFS parameters
  - **Matsers:** Which node(s) are the master ones
  - **Slaves:** which nodes are the slaves
- Hadoop has around 190 parameters
  - Mostly 10-20 are the effective ones

Web Interface

HDFS Interface

NameNode 'mach1:8000'

Started: Fri Aug 22 10:50:32 GMT 2008
Version: 0.17.1.r81212
Compiled: Mon Aug 11 03:42:21 GMT 2008 by rs3e24
Upgrades: There are no updates in progress

Applications Places Sun Jun 10, 10:43 AM T V Ganesh

localhost Hadoop Map/Re… | Running Hadoop On Ubun… | Blogger: Giga thoughts… | My Stats — WordPress.co…

localhost:50030/jobtracker.jsp

Gmail | Facebook | Welcome, Tinniam … | Blogger: Dashboard

localhost Hadoop Map/Reduce Ad

State: RUNNING
Started: Sun Jun 10 10:10:10 IST 2012
Version: 1.0.3, r1335192
Compiled: Tue May 8 20:16:59 UTC 2012 by hortonfo
Identifier: 201206101010

Cluster Summary (Heap Size is 15.06 MB/989.

Running Map Tasks	Running Reduce Tasks	Total Submissions	Nodes	Occupied Map Slots	Occupied Reduce Slots
0	0	2	1	0	0

Scheduling Information

Queue Name	State	Scheduling Information
default	running	N/A

Filter (Jobid, Priority, User, Name)
Example: 'user:smith 3200' will filter by 'smith' only in the user field and '3200' in all fields

Running Jobs

none

Completed Jobs

Jobid	Priority	User	Name	Map % Complete	Ma… To…
job_201206101010_0003	NORMAL	root	grep-search	100.00%	9

localhost Hadoop Map… | tvganesh@localhost:/… | hadoop.doc - LibreOffi…

Hadoop job_200709211549_0003 on localhost

User: hadoop
Job Name: streamjob34453.jar
Job File: /usr/local/hadoop-datastore/hadoop-hadoop/mapred/system/job_200709211549_0003/job.xml
Status: Succeeded
Started at : Fri Sep 21 16:07:10 CEST 2007
Finished at: Fri Sep 21 16:07:26 CEST 2007
Finished in: 16sec

Kind	% Complete	Num Tasks	Pending	Running	Complete	Killed	Failed/Killed Task Attempts
map	100.00%	3	0	0	3	0	0 / 0
reduce	100.00%	1	0	0	1	0	0 / 0

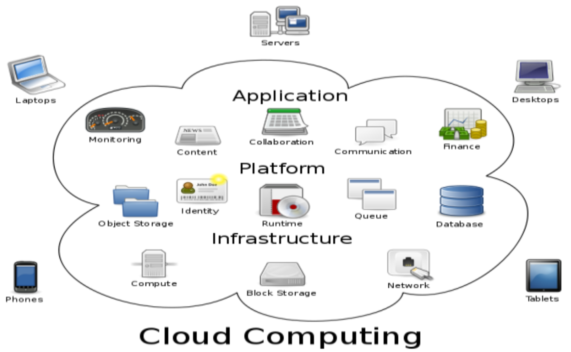
	Counter	Map	Reduce	Total
Job Counters	Launched map tasks	0	0	3
	Launched reduce tasks	0	0	1
	Data-local map tasks	0	0	3
Map-Reduce Framework	Map input records	77,637	0	77,637
	Map output records	103,909	0	103,909
	Map input bytes	3,659,910	0	3,659,910
	Map output bytes	1,083,767	0	1,083,767
	Reduce input groups	0	85,095	85,095
	Reduce input records	0	103,909	103,909
	Reduce output records	0	85,095	85,095

Change priority from NORMAL to: VERY_HIGH HIGH LOW VERY_LOW

Bigger Picture: Hadoop vs. Other Systems

	Distributed Databases	Hadoop
Computing Model	- Notion of transactions - Transaction is the unit of work - ACID properties, Concurrency control	- Notion of jobs - Job is the unit of work - No concurrency control
Data Model	- Structured data with known schema - Read/Write mode	- Any data will fit in any format - (un)(semi)structured - ReadOnly mode
Cost Model	- Expensive servers	- Cheap commodity machines
Fault Tolerance	- Failures are rare - Recovery mechanisms	- Failures are common over thousands of machines - Simple yet efficient fault tolerance
Key Characteristics	- Efficiency, optimizations, fine-tuning	- Scalability, flexibility, fault tolerance

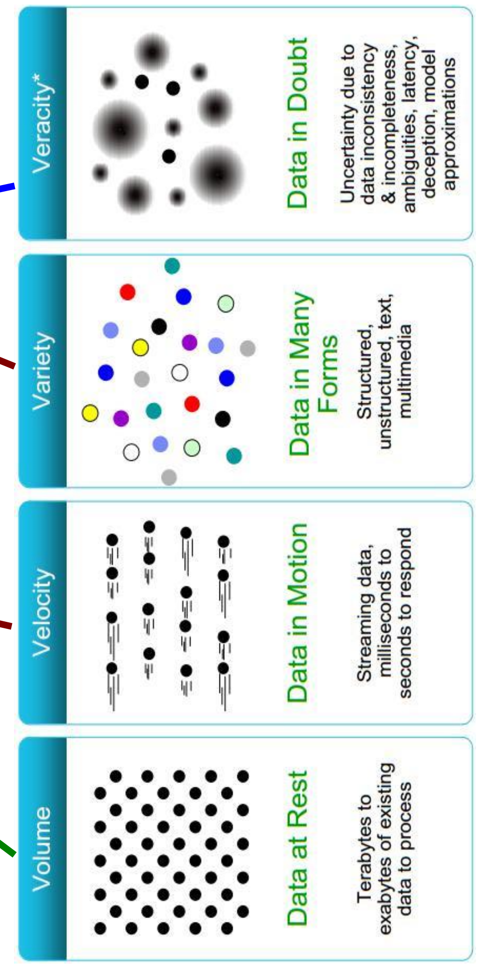
- *Cloud Computing*
  - A computing model where any computing infrastructure can run on the cloud
  - Hardware & Software are provided as remote services
  - Elastic: grows and shrinks based on the user's demand
  - Example: Amazon EC2



Recall…DBMS

- **Data is nicely structured (known in advance)**
- **Data is correct & certain**
- **Data is relatively static & small-mid size**

- **Access pattern: Mix Read/Write**
- **Notion of transactions**

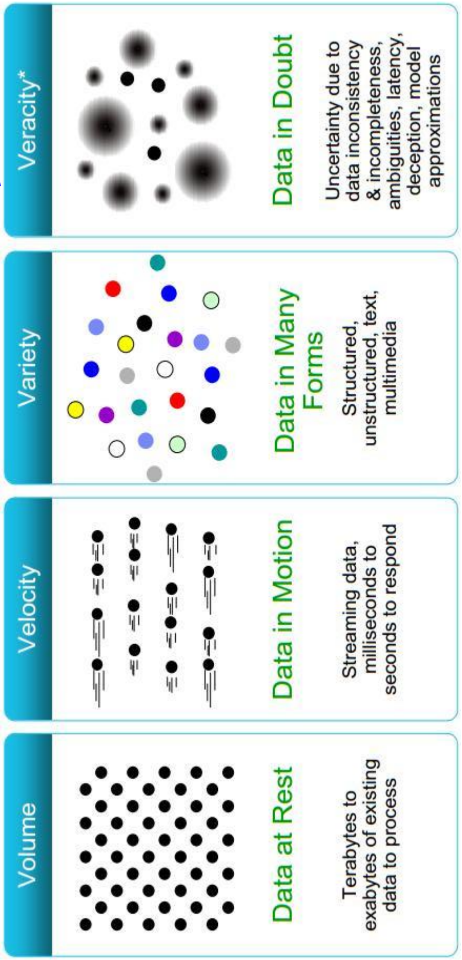


In Big Data: ***It is read only,***
No notion of transactions

What About Hadoop

- **Any structure will fit**
- **Data is correct & certain**
- **Data is static, but scales to petabytes**

- **Access pattern: Read-Only**
- **Notion of jobs**



In Big Data: ***It is read only,***
No notion of transactions