MapReduce

The Story of Sam

Saliya Ekanayake

SALSA HPC Group
http://salsahpc.indiana.edu

Pervasive Technology Institute, Indiana University, Bloomington

Sam’s Mother

- Believed “an apple a day keeps a doctor away”

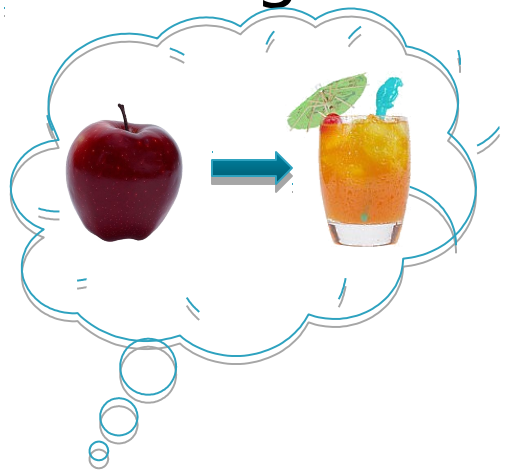
Mother

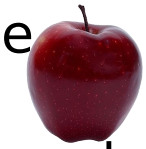
An Apple

Sam

One day

- Sam thought of “drinking” the apple



- He used a  to cut the  and a  to make juice.

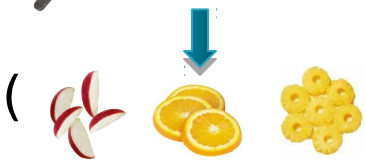




Next Day

- Sam applied his invention to all the fruits he could find in the *fruit basket*

- (map  '(  ))



- (reduce  '(  ))

A *list of values* mapped into another *list of values*, which gets reduced into a *single value*

Classical Notion of MapReduce in Functional Programming

18 Years Later

- Sam got his first job in JuiceRUs for his talent in making juice

Wait!

- Now, it's not just one basket but a whole *container* of fruits

- Also, they produce a *list* of juice types separately

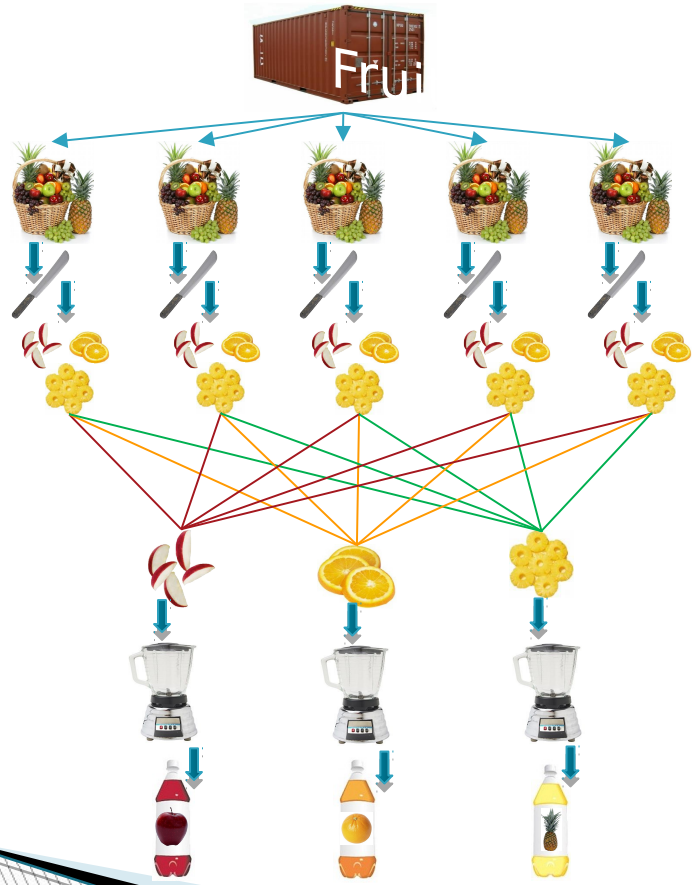
Large data and list of values for output

- But, Sam had just ONE  and ONE 

NOT ENOUGH !!

Brave Sam

- Implemented a *parallel* version of his innovation



Each input to a map is a list of <key, value> pairs

(<a, >, <o, >, <p, >, ...)

A *list of <key, value>* pairs mapped into another *list of <key, value>* pairs which gets grouped by the key and reduced into a *list of values*

Each output of a map is a list of <key, value> pairs

(<a', > , <o', > , <p', > , ...)

Grouped by key

Eac… list>… on the grouping/hashing mechanism)
e.g. <a', (… …)>

Reduced into a list of values

The idea of MapReduce in Data Intensive Computing

Afterwards

- Sam realized,
  - To create his favorite mix fruit juice he can use a *combiner* after the reducers
  - If several <key, value-list> fall into the same group (based on the grouping/hashing algorithm) then use the blender (reducer) separately on each of them
  - The knife (mapper) and blender (reducer) should not contain residue after use – *Side Effect Free*
  - In general reducer should be *associative* and *commutative*

That’s All Folks!

- We think Sam was you ⏪

Dr. Latifur Khan
Department of Computer Science
University of Texas at Dallas

Source:
http://developer.yahoo.com/hadoop/tutorial/module4.html

Commodity Clusters

- MapReduce is designed to efficiently process large volumes of data by connecting many commodity computers together to work in parallel

- A *theoretical* 1000-CPU machine would cost a very large amount of money, far more than 1000 single-CPU or 250 quad-core machines

- MapReduce ties smaller and more reasonably priced machines together into a single cost-effective *commodity cluster*

Isolated Tasks

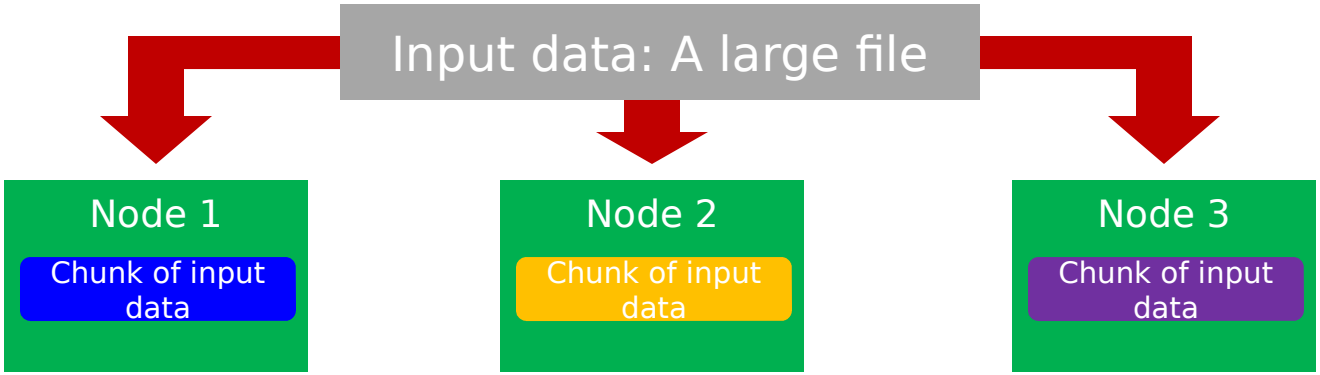
- MapReduce divides the workload into multiple *independent tasks* and schedule them across cluster nodes

- A work performed by each task is done *in isolation* from one another

  - The amount of communication which can be performed by tasks is mainly limited for scalability reasons

Data Distribution

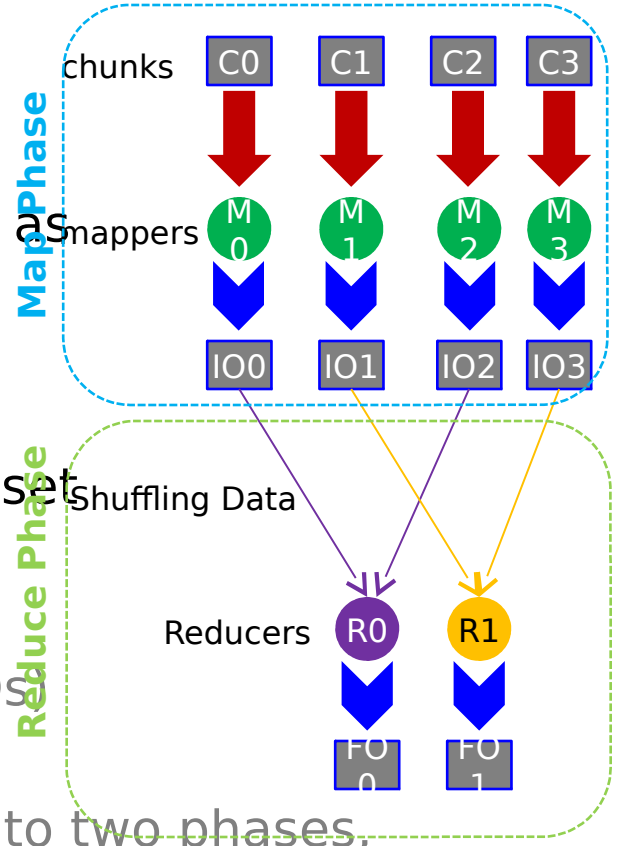
- In a MapReduce cluster, data is distributed to all the nodes of the cluster as it is being loaded in
- An underlying distributed file systems (e.g., GFS) splits large data files into chunks which are managed by different nodes in the cluster



- Even though the file chunks are distributed across several machines, they form *a single namesapce*

MapReduce: A Bird's-Eye View

- In MapReduce, chunks are processed in isolation by tasks called *Mappers*
- The outputs from the mappers are denoted as intermediate outputs (IOs) and are brought into a second set of tasks called *Reducers*
- The process of bringing together IOs into a set of Reducers is known as *shuffling process*
- The Reducers produce the final outputs (FOs)
- Overall, MapReduce breaks the data flow into two phases, *map phase* and *reduce phase*

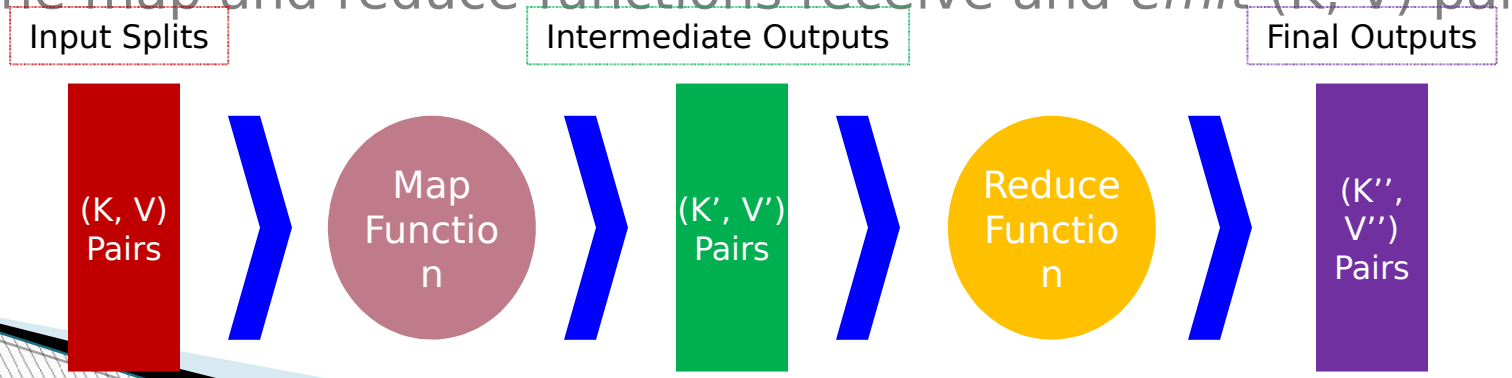


Keys and Values

- The programmer in MapReduce has to specify two functions, the *map function* and the *reduce function* that implement the Mapper and the Reducer in a MapReduce program

- In MapReduce data elements are always structured as key-value (i.e., (K, V)) pairs

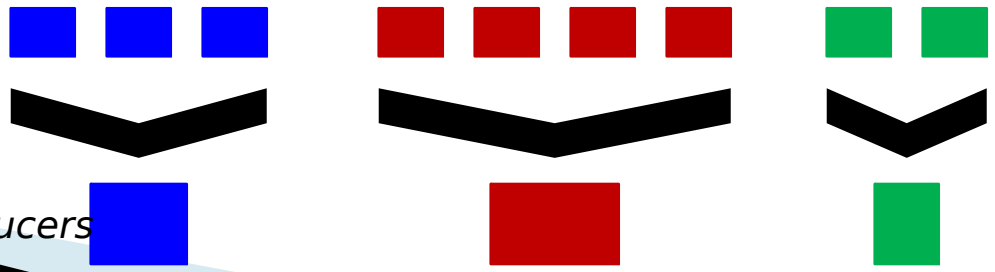
- The map and reduce functions receive and *emit* (K, V) pairs



Partitions

- In MapReduce, intermediate output values are not usually reduced together
- *All values with the same key are presented to a single Reducer together*
- More specifically, a different subset of intermediate key space is assigned to each Reducer
- These subsets are known as *partitions*

Different colors represent different keys (potentially) from different Mappers



Partitions are the input to Reducers

MapReduce

- In this part, the following concepts of MapReduce will
  be described:

  - Basics
  - A close look at MapReduce data flow
  - Additional functionality
  - Scheduling and fault-tolerance in MapReduce
  - Comparison with existing techniques and models

Hadoop

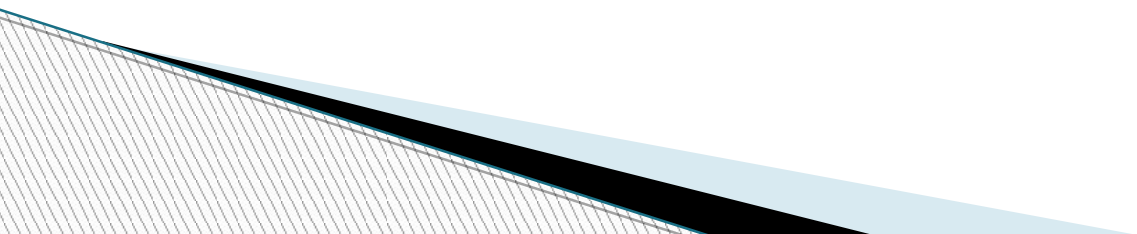
- Since its debut on the computing stage, MapReduce has frequently been associated with *Hadoop*
- Hadoop is an open source implementation of MapReduce and is currently enjoying wide popularity
- Hadoop presents MapReduce as an analytics engine and under the hood uses a distributed storage layer referred to as Hadoop Distributed File System (*HDFS*)
- HDFS mimics Google File System (*GFS*)

Distributed File Systems

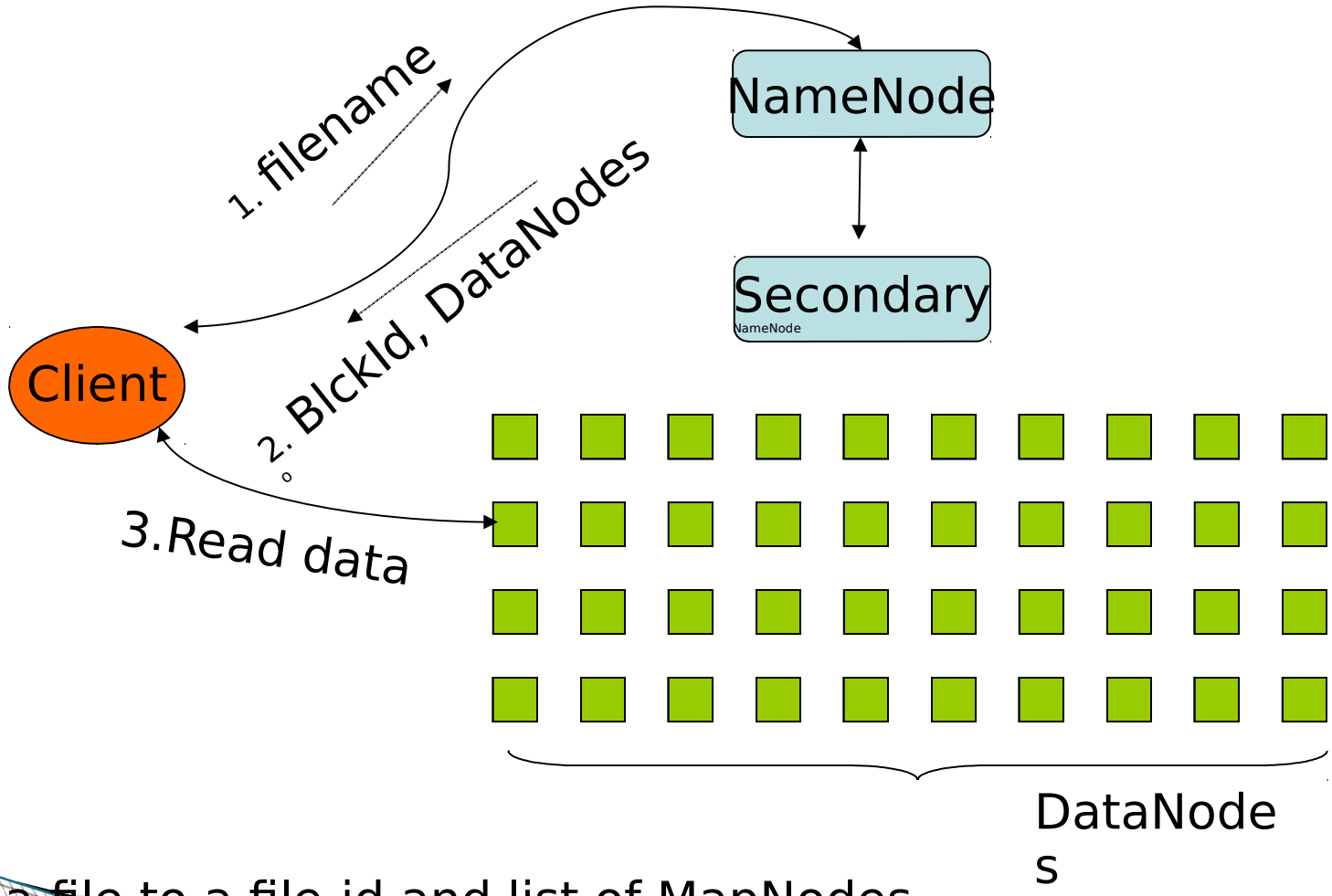
- Highly scalable distributed file system for large data-intensive applications.
  - E.g. 10K nodes, 100 million files, 10 PB
- Provides redundant storage of massive amounts of data on cheap and unreliable computers
  - Files are replicated to handle hardware failure
  - Detect failures and recovers from them
- Provides a platform over which other systems like MapReduce, BigTable operate.

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 128 MB block size
  - Each block replicated on multiple DataNodes
- **Intelligent Client**
  - Client can find location of blocks
  - Client accesses data directly from DataNode

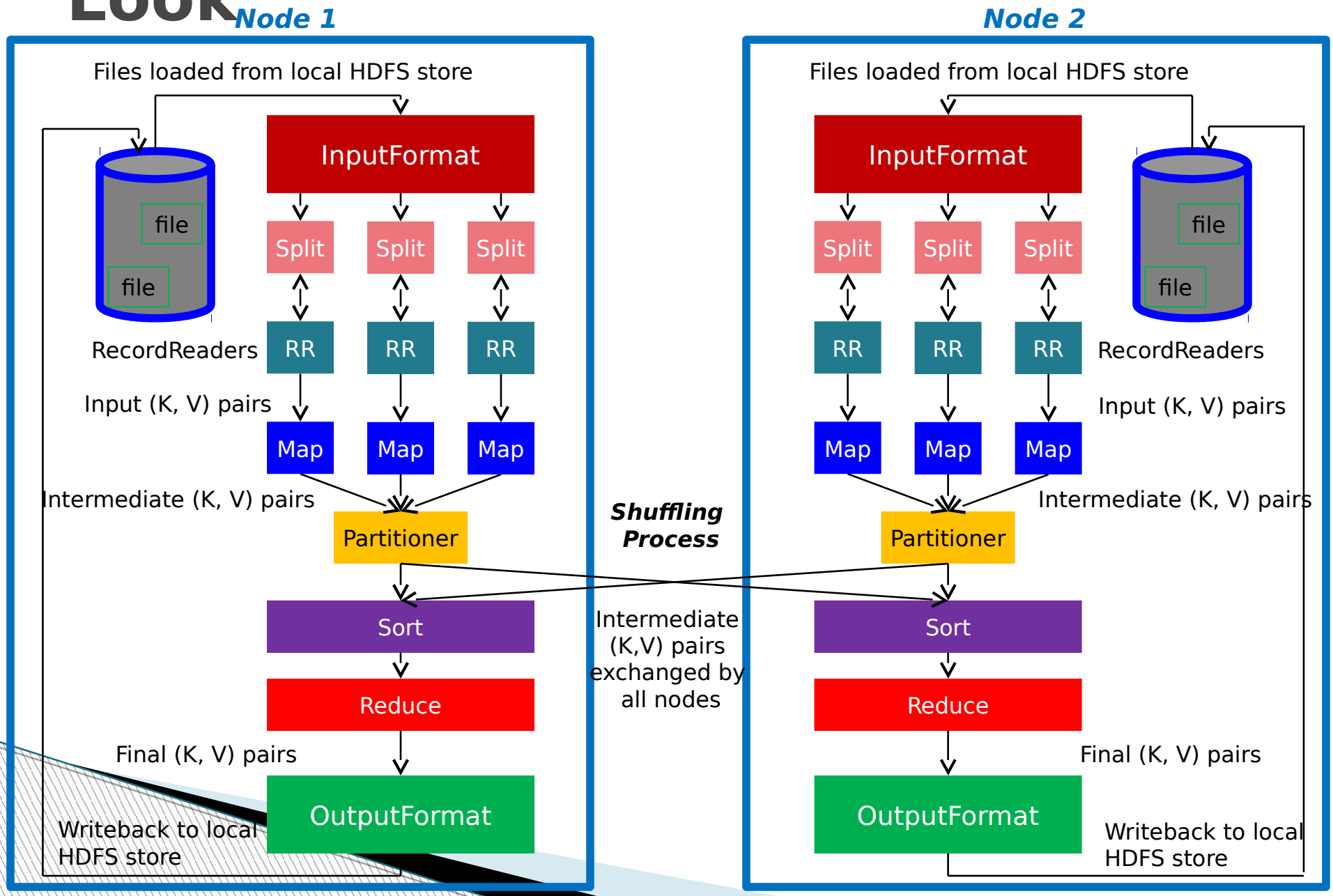


HDFS Architecture



aps a file to a file-id and list of MapNodes
ps a block-id to a physical location on disk

Hadoop MapReduce: A Closer Look

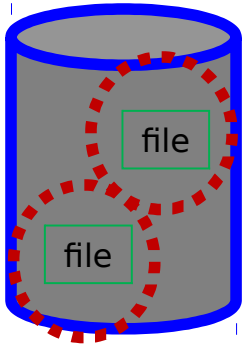


Input Files

- *Input files* are where the data for a MapReduce task is initially stored

- The input files typically reside in a distributed file system (e.g. HDFS)

- The format of input files is arbitrary
  - Line-based log files
  - Binary files
  - Multi-line input records
  - Or something else entirely

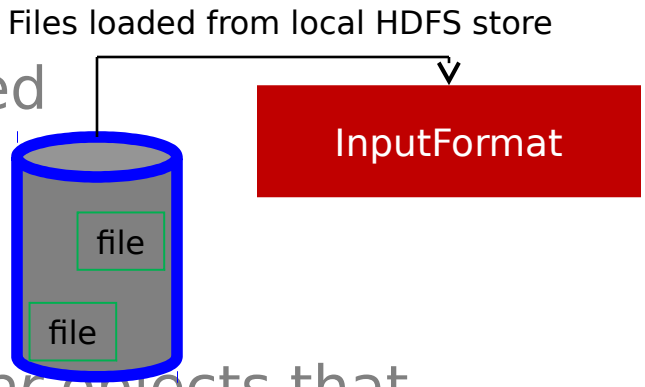


InputFormat

- How the input files are split up and read is defined by the *InputFormat*

  - InputFormat is a class that does the following:

    - Selects the files that should be used for input
    - Defines the *InputSplits* that break a file
    - Provides a factory for *RecordReader* objects that read the file



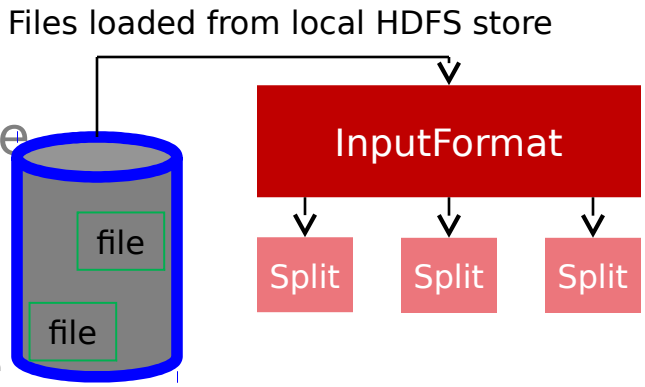
InputFormat Types

- Several InputFormats are provided with Hadoop:

InputFormat	Description	Key	Value
TextInputFormat	Default format; reads lines of text files	The byte offset of the line	The line contents
KeyValueInputFormat	Parses lines into (K, V) pairs	Everything up to the first tab character	The remainder of the line
SequenceFileInputFormat	A Hadoop-specific high-performance binary format	user-defined	user-defined

Input Splits

- An *input split* describes a unit of work that comprises a single map task in a MapReduce program
- By default, the InputFormat breaks a file up into 64MB splits
- By dividing the file into splits, we allow several map tasks to operate on a single file in parallel
- If the file is very large, this can improve performance significantly through parallelism
- Each map task corresponds to a *single* input split



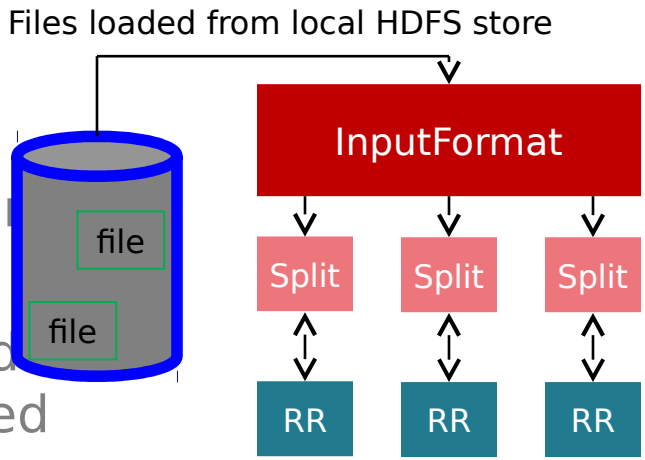
RecordReader

- The input split defines a slice of work but does not describe how
  to access it

- The *RecordReader* class actually loads data from its source and converts it into (K, V) pairs suitable for reading by Mappers

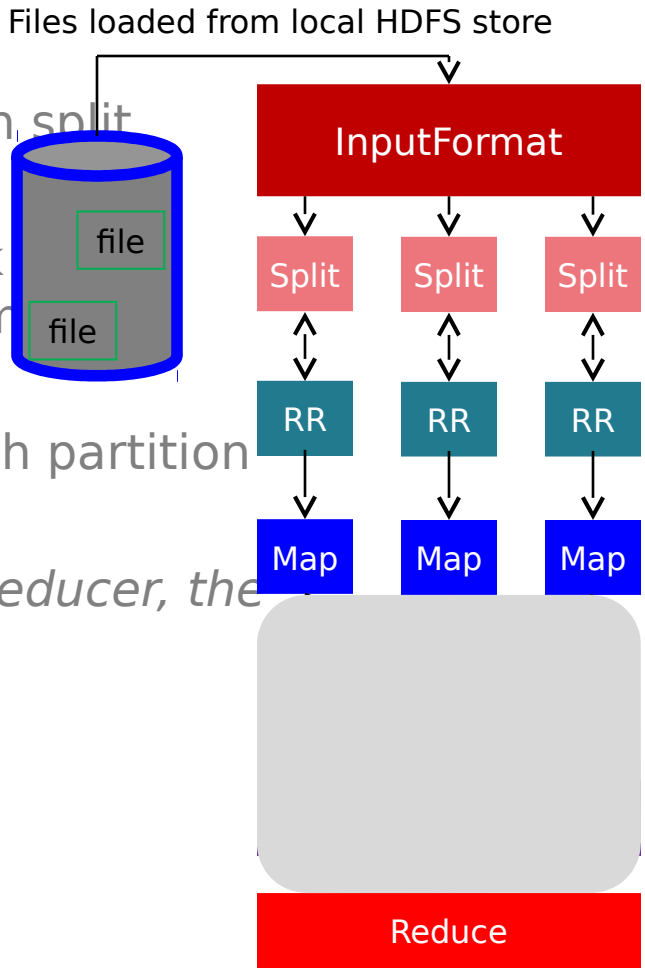
- The RecordReader is invoked repeatedly on the input until the entire split is consu

- Each invocation of the RecordReader lead to another call of the map function defined by the programmer



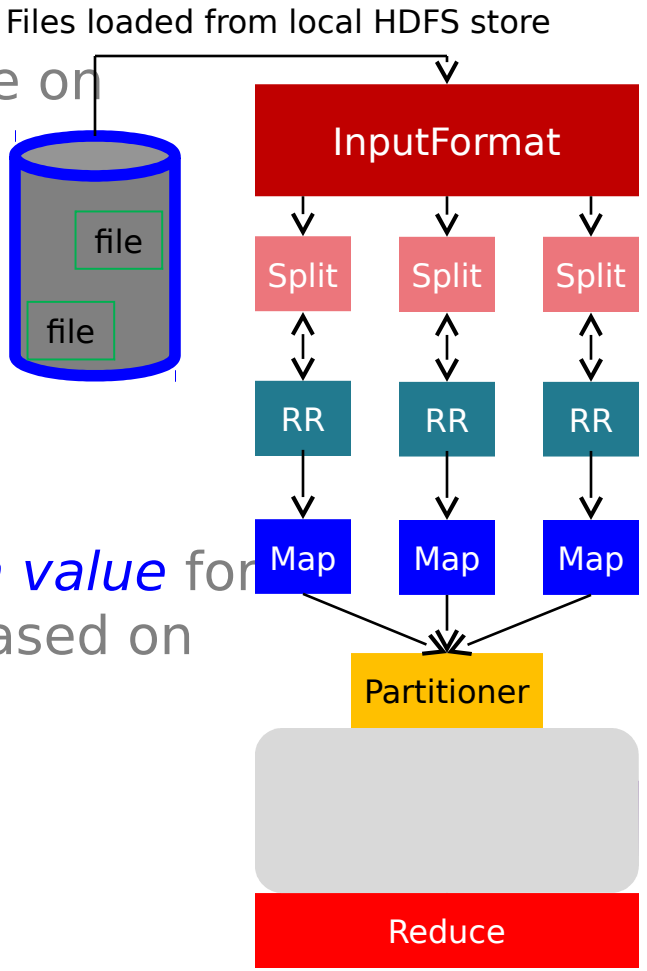
Mapper and Reducer

- The *Mapper* performs the user-defined work of the first phase of the MapReduce program
- A new instance of Mapper is created for each split
- The *Reducer* performs the user-defined work of the second phase of the MapReduce program
- A new instance of Reducer is created for each partition
- *For each key in the partition assigned to a Reducer, the Reducer is called once*



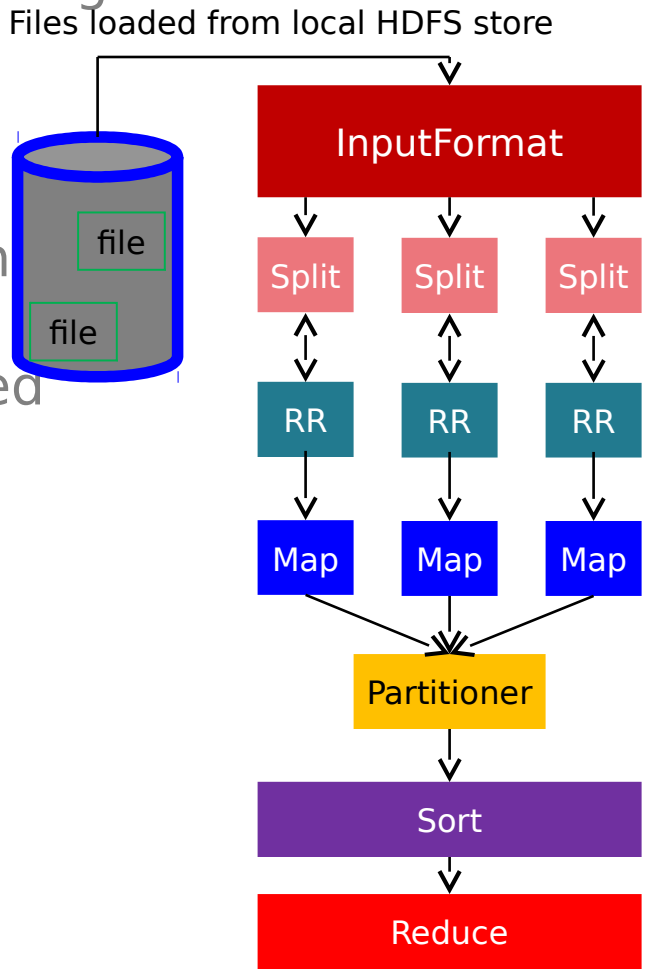
Partitioner

- Each mapper may emit (K, V) pairs to *any* partition
- Therefore, the map nodes must all agree on where to send different pieces of intermediate data
- The *partitioner* class determines which partition a given (K,V) pair will go to
- The default partitioner computes *a hash value* for given key and assigns it to a partition based on this result



Sort

- Each Reducer is responsible for reducing the values associated with (several) intermediate keys
- The set of intermediate keys on a sin node is *automatically sorted* by MapReduce before they are presented to the Reducer



OutputFormat

- The *OutputFormat* class defines the way (K,V) pairs produced by Reducers are written to output files
- The instances of OutputFormat provided by Hadoop write to files on the local disk or in H
- Several OutputFormats are provided by Hadoop:

OutputFormat	Description
TextOutputFormat	Default; writes lines in "key \t value" format
SequenceFileOutputFormat	Writes binary files suitable for reading into subsequent MapReduce jobs
NullOutputFormat	Generates no output files