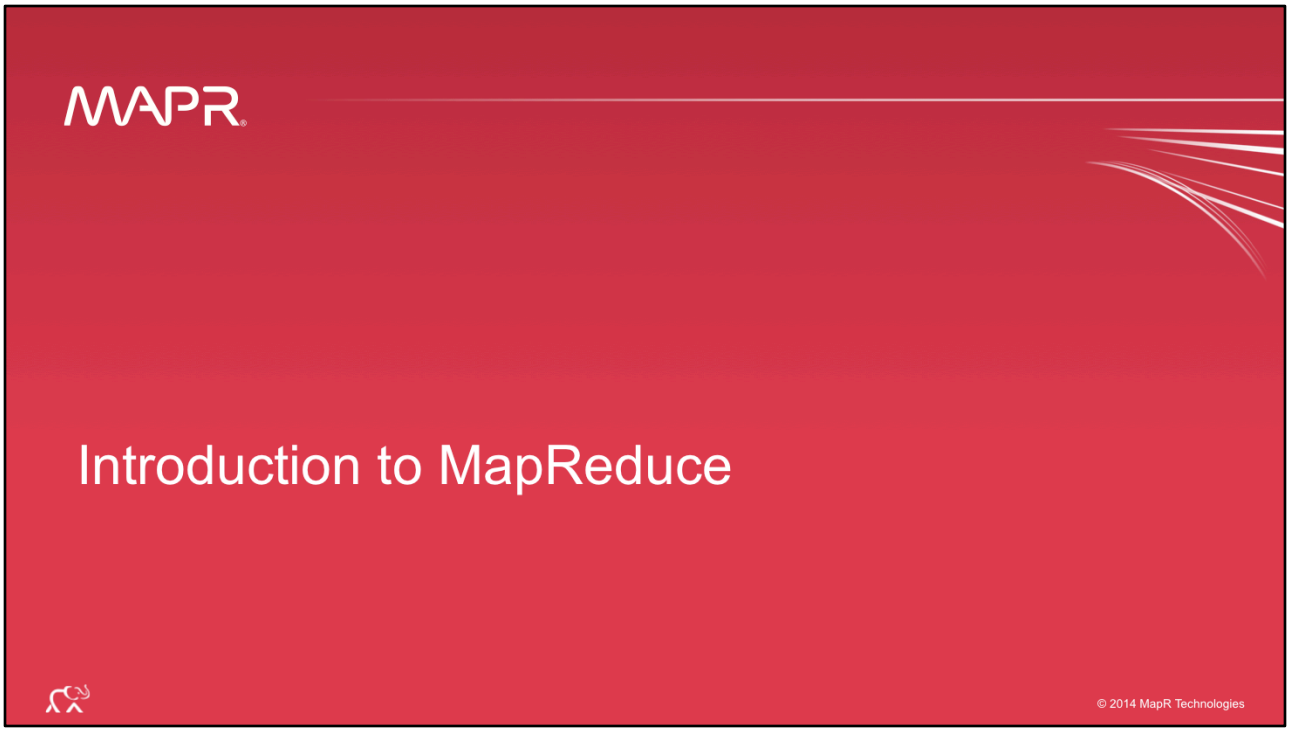

Introduction to MapReduce
© 2014 MapR Technologies

A Few Quotes to Start Us Off

As soon as an Analytical Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will then arise — by what course of calculation can these results be arrived at by the machine in the shortest time?

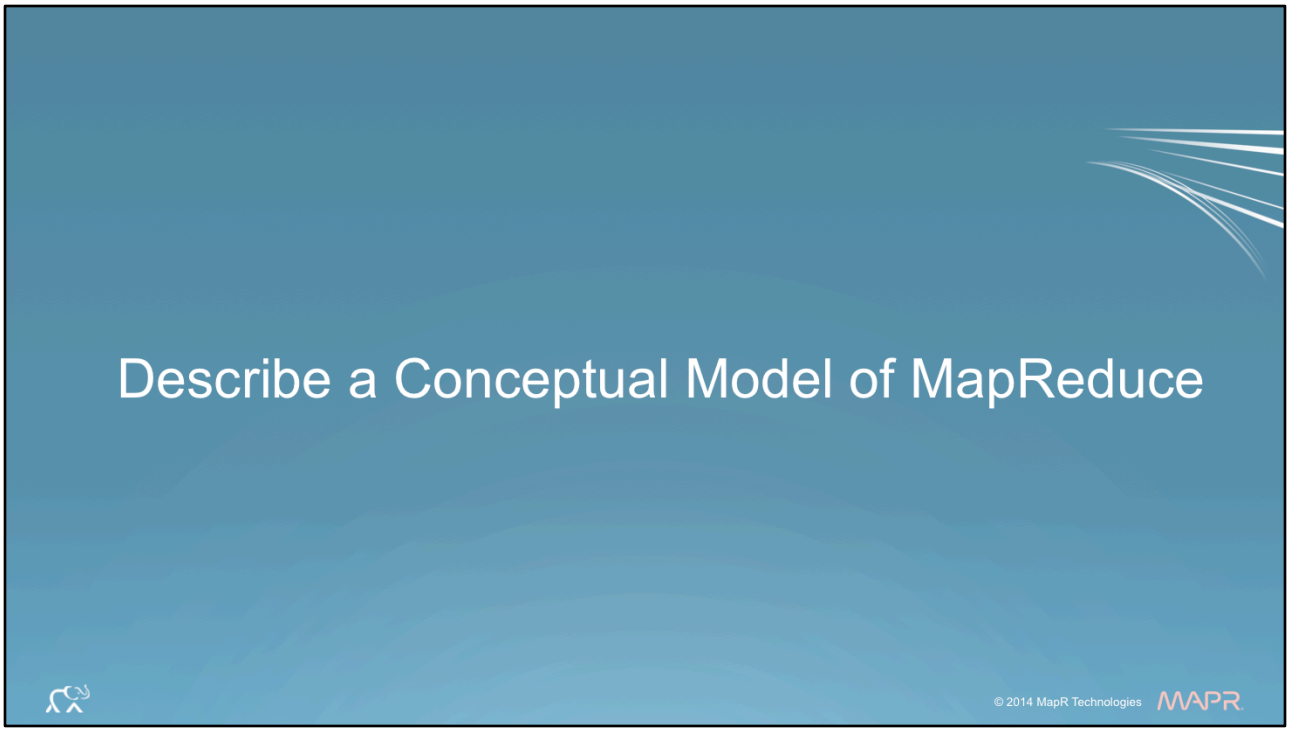
(Charles Babbage, 1864)

Divide et conquera.

(Julius Caesar, 50 B.C.)

Learning Objectives

- Describe a conceptual model a MapReduce
- Discuss the historical perspective of MapReduce
- Discuss and overview of the MapReduce paradigm
- Describe the complete flow of execution and data in Hadoop
- Discuss an overview of the Hadoop job execution framework



In this section, we will discuss an analogy that provides a conceptual model of the MapReduce execution paradigm.

Discuss a Conceptual Example of MapReduce

- Given: 10 decks of playing cards mixed in shoe
- To do: separate into 10 sorted decks

NOTE: this example illustrates how data is sorted and moved through the framework, not how it is transformed.

 5

The conceptual example we're discussing is a set of 10 decks of playing cards mixed in a "shoe" – similar to what you might find at a blackjack table in a casino.

The objective of our exercise is to separate the one set of cards into 10 individual decks, each of which is sorted by suit and face value. The suits of a deck of cards are clubs, diamonds, hearts, and spades. The face values go from ace, two, three, …,, ten, jack, queen, and king.

Define Card sorting Approach #1

- **Phase I:**
  - Get 20 volunteers – one person per 1/20 of shoe
  - Each person separates by suit and sorts by face value
  - *Question: how many piles per person? Total?*
- **Phase II:**
  - Get 4 volunteers – one per suit
    - Each person sorts the 20 corresponding stacks of their suit
    - *Question: how many piles should each person create?*
- Concatenate 4 corresponding piles to make 10 decks

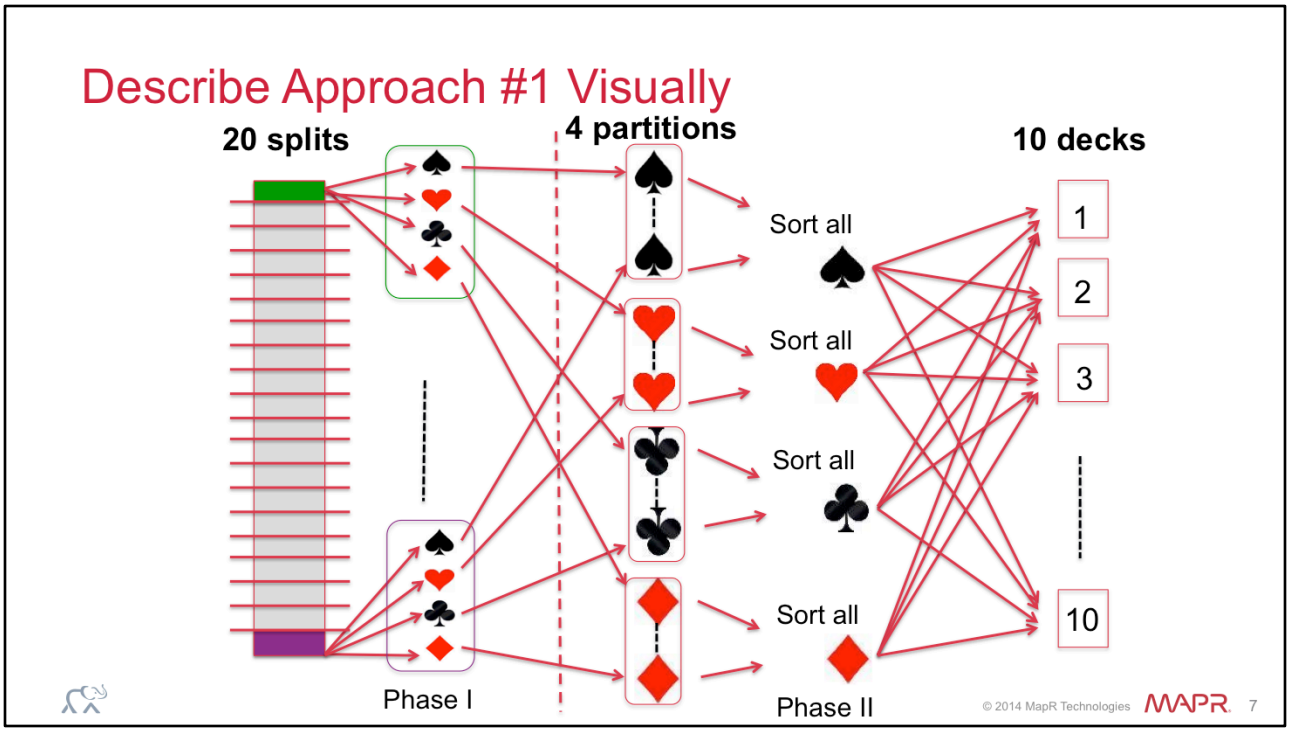
 6

The first approach we consider is as follows. Get 20 volunteers, each of which will sort 1/20th of the shoe. When each person is given their part of the shoe, they will separate cards into piles based on suit and then sort each pile based on its face value. As such, each of the 20 people will have 4 piles (one for clubs, diamonds, hearts, and spades). There will be a total of 20 x 4 = 80 piles of cards at the end of phase one.

In phase two, we will get 4 volunteers – each of whom will be assigned to a particular suit. Each person will create 10 piles – one for each deck of cards from the original shoe – and sort each of the 10 piles from smallest (ace) to largest (king) face value.

At the end of phase two, we concatenate as follows:

- Concatenate the first of ten piles of clubs, diamonds, hearts,



The graphic above depicts a visual representation of the first approach.

The person works with the first split (in green), separating cards into 4 piles and sorting each pile based on face value. That same process occurs in parallel with all 20 people in phase one. Once all 20 people are done, phase two begins.

The person working with the spades partition collects all the piles of spades from the 20 people in phase one. He sorts and merges the spades as {A, A, A, A, A, A, A, A, A, A, 2, 2, 2, …. K, K, K, K, K, K, K, K, K}. Then he creates 10 piles of cards by first distributing the aces, then the twos, … and so on until he distributes all the kings. That same process occurs in parallel with all 4 people in phase two.

Identify Salient Points of Approach #1

- More people phase in I has a good effect (up to a point)
- Second phase cannot start until first phase is done
- 4 people in second phase (4 suits)
- Can't add more people in phase II
- We partitioned the cards into suits, but we can select a different partitioner …

 8

A few points of note are described in the slide above.

First, note that we can't start the second phase until the first phase is done. In other words, the person responsible for a given suit in phase two needs all the cards of that suit sorted in phase one before he/she begins.

Second, you can only use 4 people in the second phase because there are only four suits.

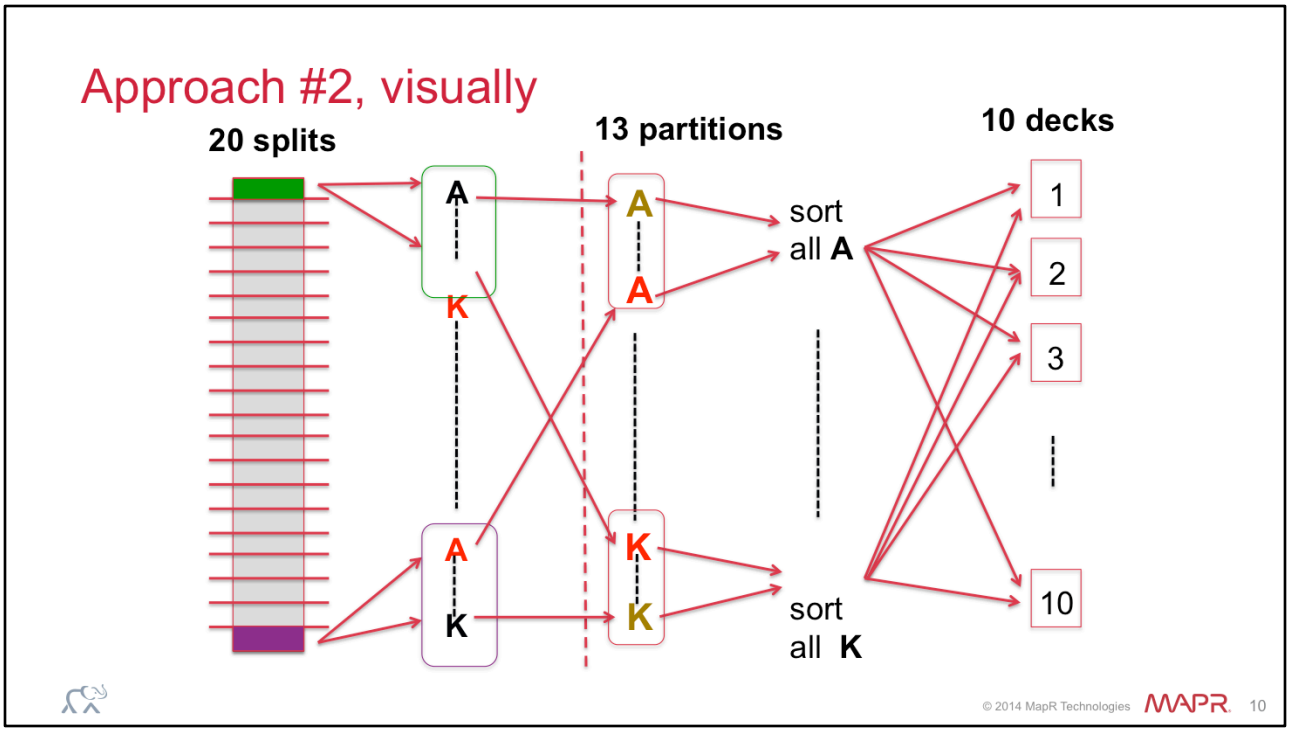
Third, having more people in phase one has a good effect on performance, up to a point. There's a tipping point between parallelism (more is better) and overhead (fewer is better).

Fourth, you can't add more people in the second phase since there are only four suits. What would a fifth person do?

Card Sorting, Approach #2

- Phase I
  - Get 20 volunteers to sort 1/20th of shoe
  - Each person separates by face value and sorts by suit into 13 stacks
- Phase II
  - Select 13 people – one per face value
  - Each person merges stacks of one face value (e.g. all the aces)
  - Each person creates 10 piles for each face-value (130 piles total)
- Concatenate 13 corresponding piles to make 10 decks

The second approach to our card sorting problem is to separate cards by face value and then sort by suit. In the second phase, we are obliged to use exactly 13 people (since there are 13 face values).



The graphic above depicts a visual representation of the second approach.

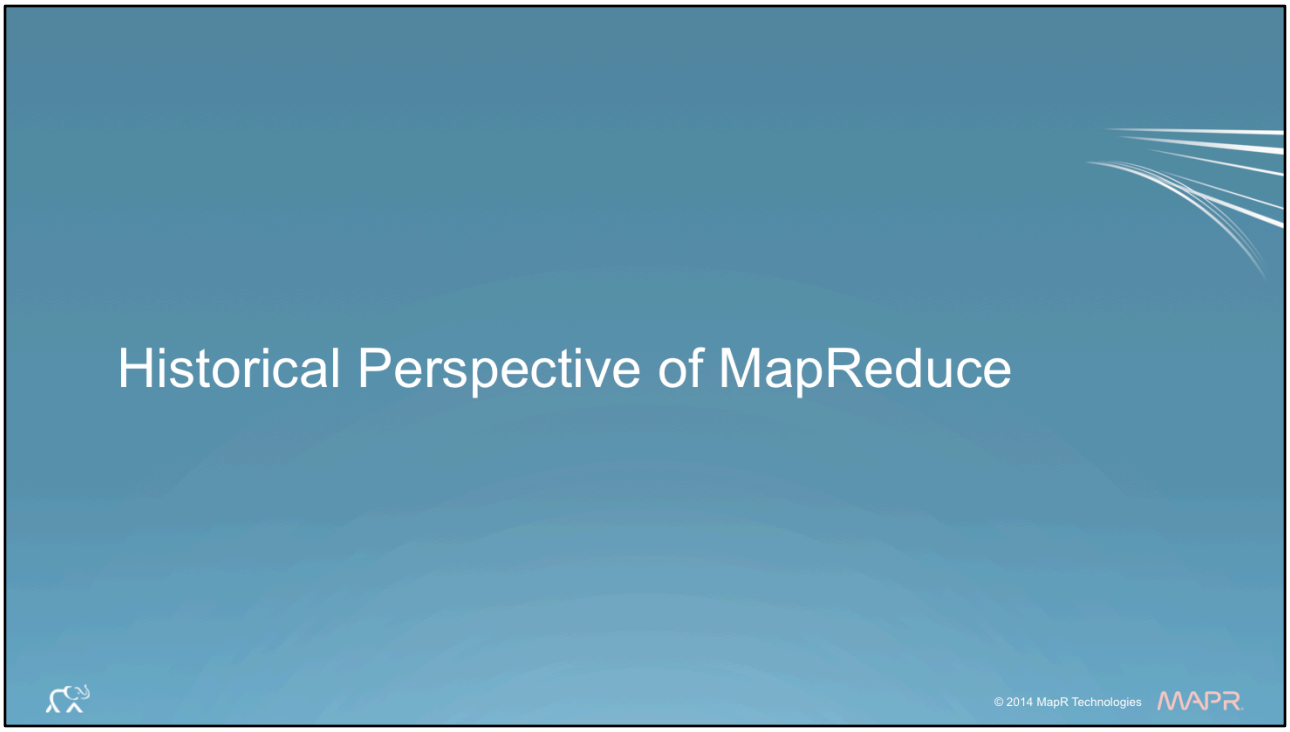
The person works with the first split (in green), separating cards into 13 piles based on face value and and sorting each pile based on suit (perhaps alphabetically – clubs, diamonds, hearts, and then spades). That same process occurs in parallel with all 20 people in phase one. Once all 20 people are done, phase two begins.

The person working with the aces partition collects all the piles of aces from the 20 people in phase one. He sorts and merges the aces as {club, club, club, club, club, club, club, club, club, club, diamond, diamond, diamond, … heart, heart, heart, …, spade, spade, spade}. Then he creates 10 piles of cards by first distributing the clubs, then the diamonds, then the hearts, and

Identify Salient points of Approach #2

- 13 persons in second phase since 13 face values
- We chose face-value as “key” along which to partition data
- Increasing number of people in second phase required picking a different partitioning key
- Is it faster than approach #1?

The primary difference between the two approaches is the number of people we leverage in the second phase. In the second approach, we partitioned the cards based on face value (of which there are 13). All other things equal, more people in the second phase should be faster (up to a point).



In this section, we will discuss a historical perspective of MapReduce.

Lisp Map-Reduce

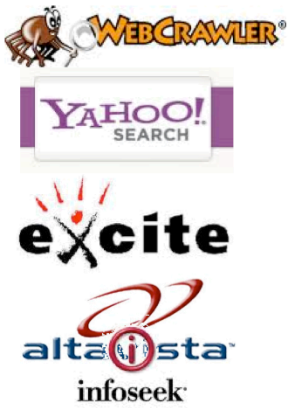
- **Lisp**
  - programming language designed for list processing
  - *Has functions for map and reduce (circa 1970's)*
- **Example: calculate the sum of squares**
  - `(map square '(1 2 3 4)) = (1 4 9 16)`
  - `(reduce + '(1 4 9 16)) = 30`
- **Note that the map function**
  - applies the same logic to each value, ***one value at a time***
  - emits a result for each value
- **Note that the reduce function**
  - applies the same logic to ***all the values*** taken together
  - emits a single result for all the values

The intention of this slide is two-fold. On the one hand, it should be noted that MapReduce was not invented at Google (or Hadoop). Lisp, for example, has a map and reduce set of functionality since the 1970's.

The other purpose of this slide is to further explain the map/reduce paradigm through how it is done in Lisp. The example shown has a map of the square function on an input list from 1 to 4. The square function, since it is mapped, will apply to each of the inputs and produce a single output per input (i.e. 1, 4, 9, 16), The addition function reduces the list and produces a single output (the sum of the input).

Web Search Engines

Circa 1995



Now

So how did we get here? Google is the poster child for the power of the MapReduce paradigm which underlies Hadoop. Google was the 19th search engine to enter a crowded market. You might recall some of these old search brands, but you *might not*… because within a few short years, Google emerged and dominated the search market.

Simplified Web Search Engine Approach

- Crawl the Web
- Sort the pages by URL
- Remove the junk
- Create an inverted index
- Rank the results

The high-level approach described above is taken from http://www.google.com/intl/en/insidesearch/howsearchworks/thestory/. Without getting into the details of each step, it is straightforward to understand conceptually what this approach does.

1. Crawl the Web

This involves using so-called "spiders" to crawl the web, following links within web pages to get to other web pages. Overall, this is the most time-consuming step.

2. Sort the pages by URL

The pages are sorted based on their URL.

3. Remove the junk

A good search engine should remove "junk" from a known list

Word Count Algorithm from Google White Paper

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat
jeff@google.com, sanjay@google.com
Google,Inc.

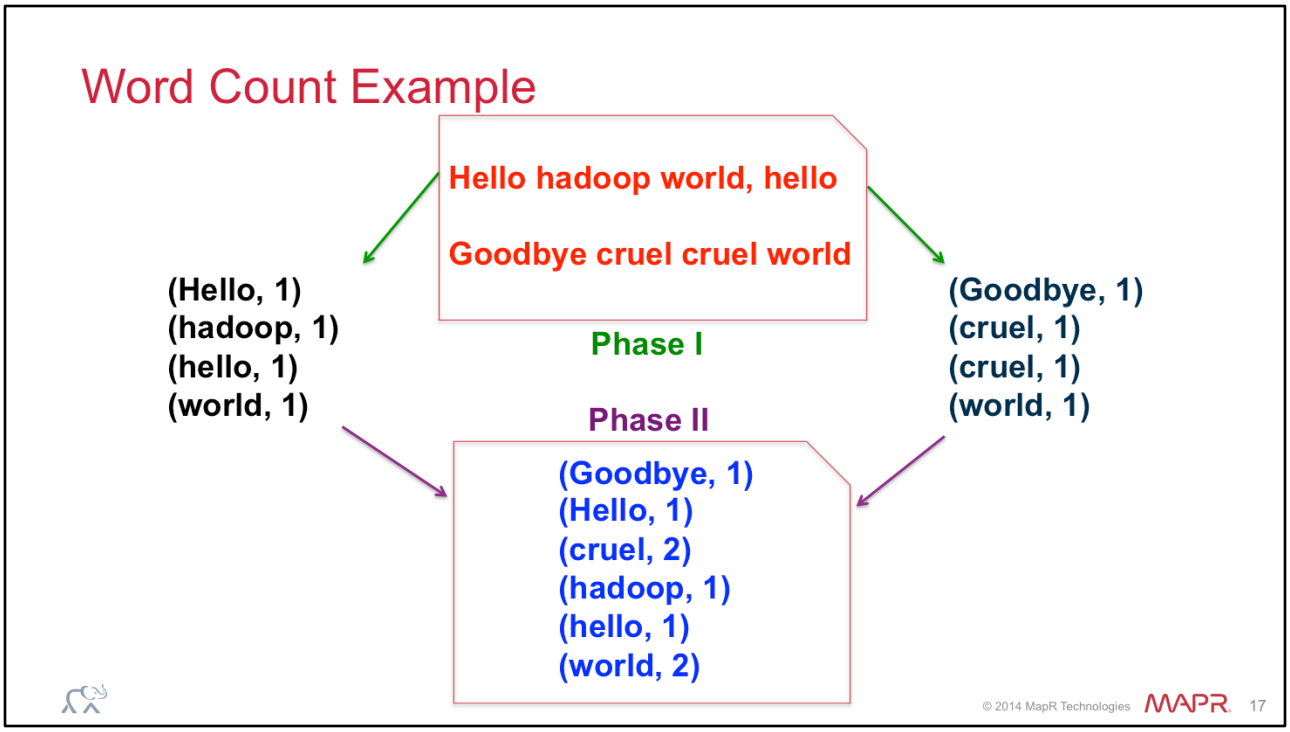
```
map(String key, String value):
   // key: document name
   // value: document contents
   for each word w in value:
      EmitIntermediate(w,"1");

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

 MAPR 16

The word count algorithm shown above is taken directly from the seminal paper on MapReduce from Dean and Ghemawat. Algorithms for counting words existed before this paper was published, obviously, but the mechanism for doing it using the MapReduce framework was novel. The algorithm is quite simple. This is true of most MapReduce programs as we will later see.

The map method takes as input a key and a value, where the key represents the name of a document and the value is the contents of the document. The map method loops through each word in the document and emits a 2-tuple (word, 1).

The reduce method takes as input a key and a list of values, where the key represents a word, and the list of values is the list of counts (i.e. "1" 's) for that word. The reduce method



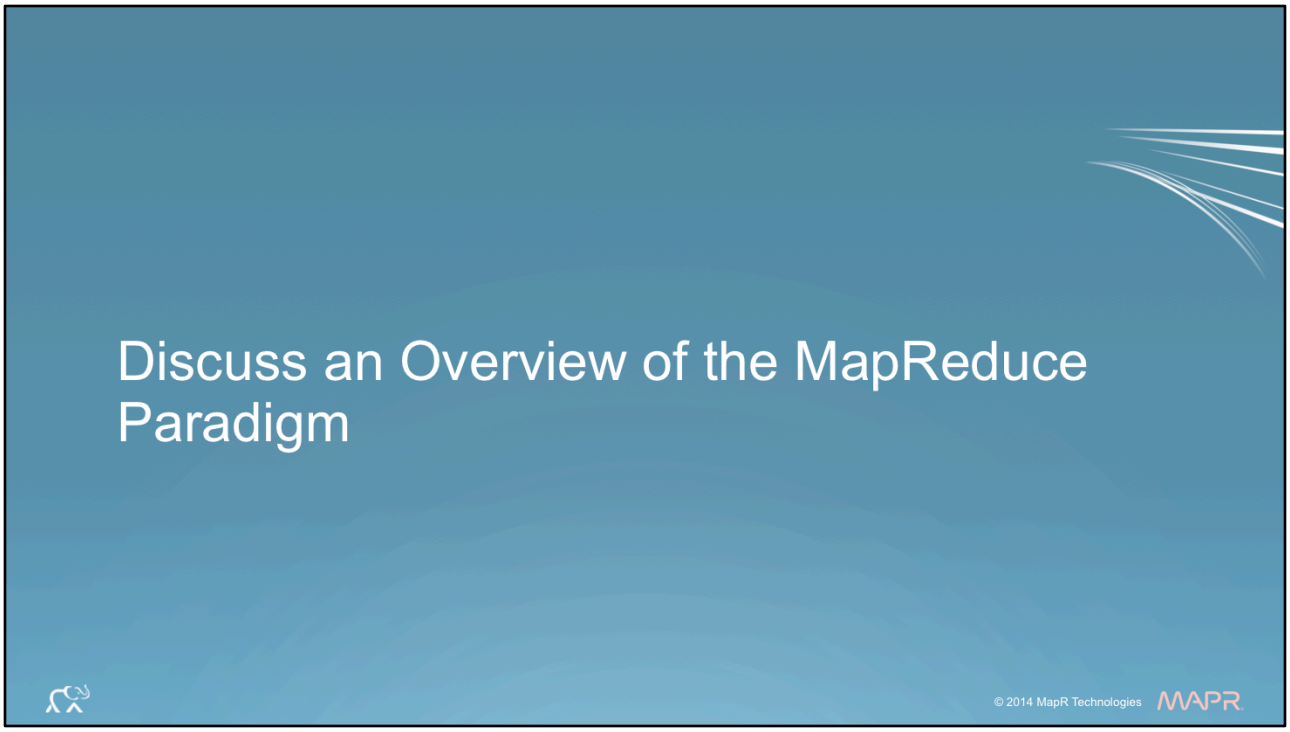
In summary, each mapper in the map phase takes an input list (e.g. “Hello hadoop world”) and maps it to a list of key-value pairs (e.g. (Hello, 1), (hadoop, 1), (world, 1)). Note that even in the input list “Hello hadoop world”, there is a key and a value, though the key is not obvious. We will discuss this later.

Each reducer in the reduce phase takes an input of a key and the list of values associated with that key (e.g. (world, 1), (world, 1)) and emits a single output (e.g. (world, 2)).

How Can Word Count (or a variation) Be Used for Building a Web Search Engine?

- Crawl the Web
  - count frequency of terms in Web documents
- Sort the pages by URL
  - done automatically by framework
- Remove the junk
  - use a MapReduce filter design pattern
- Create an inverted index
  - emit location instead of count
- Rank the results
  - count the frequency of URLs, number of hits, …

The slide above revisits the web search engine approach in the context of wordcount and the MapReduce framework. Of the 5 basic phases described, wordcount (or a small variation of it) could be used. Note that the sorting phase is actually done by the framework. Removing junk involves building and using a filter – a task that can be implemented with MapReduce as well.



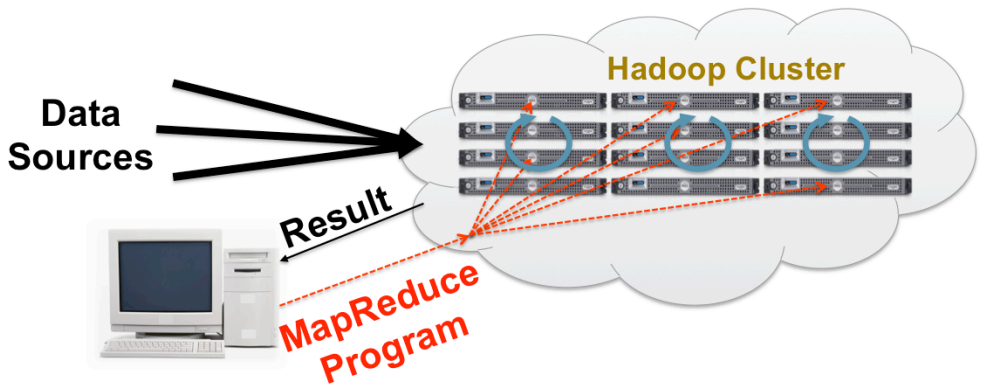
In this section, we discuss an overview of the MapReduce paradigm.

Describe a Summary of Hadoop MapReduce

- **3 phases (technically, not 2)**
  - Map
  - Shuffle/sort/merge
  - Reduce
- **Map phase**
  - Split input amongst nodes
  - Each node processes local data set and emit key-value pairs
- **Shuffle/merge phase**
  - Transfer results from mappers to reducers
  - Merge results in each partition by key
- **Reduce phase**
  - Read all results for each key
  - Perform application-specific logic on data

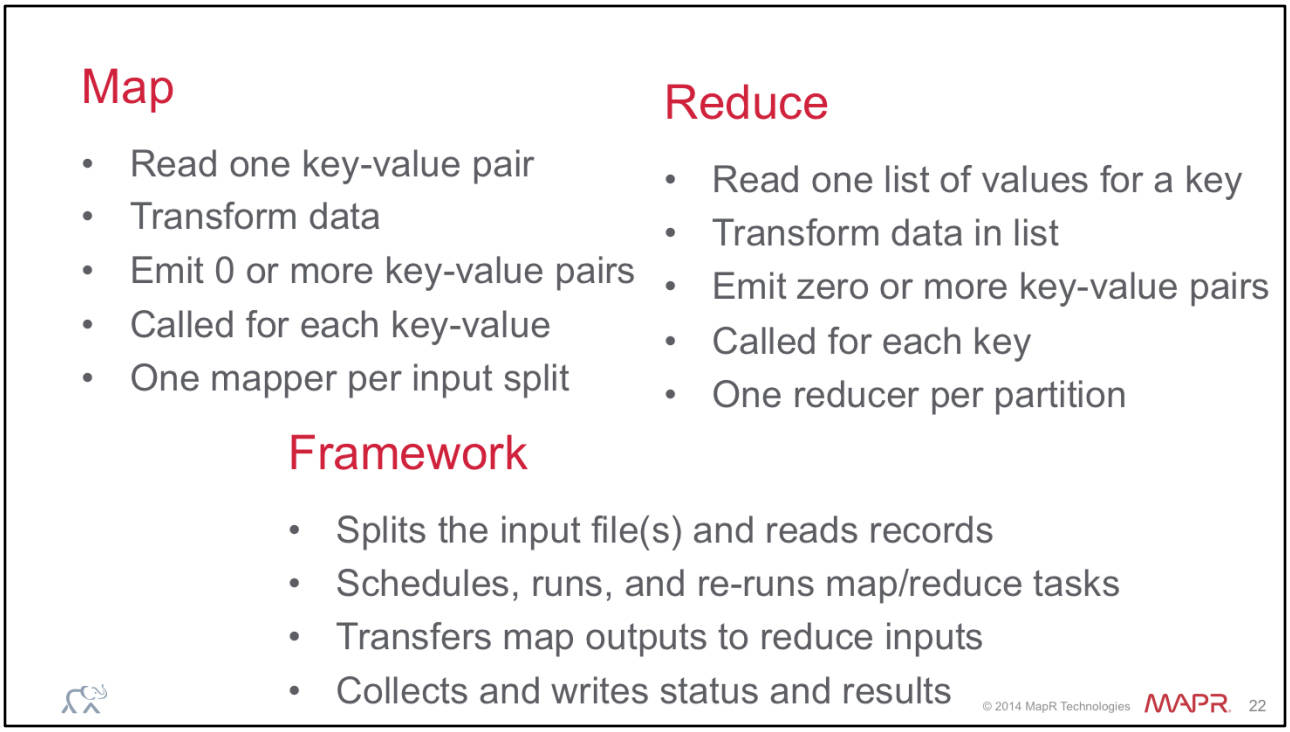
The slide above shows a high-level summary of the Hadoop MapReduce computational paradigm. There are actually 3 phases (map, shuffle/sort, and reduce) in MapReduce. The map phase is split amongst the task tracker nodes where the data is located. Each node in the map phase emits key-value pairs based on input one record at a time. The shuffle/sort phase is handled by the Hadoop framework. Output from the mappers is sent to the reducers as partitions. The reduce phase is split amongst the partitions in which each reduce reads a key and iterable list of values associated with that key. Reducers emit zero or more key-value pairs based on the application logic.

Describe the Hadoop Runtime Paradigm



Recall that the MapReduce paradigm is based on sending compute to where data resides.

We collect source data on the Hadoop cluster, either by bulk copying data in, or better yet, by simply accumulating data in the cluster over time. When we kick off a MapReduce job, Hadoop sends map and reduce tasks to appropriate servers in the cluster, and the framework manages all the details of data passing between nodes. Much of the compute happens on nodes with data on local disks, which minimizes network traffic. Finally, we can read back the result from the cluster.

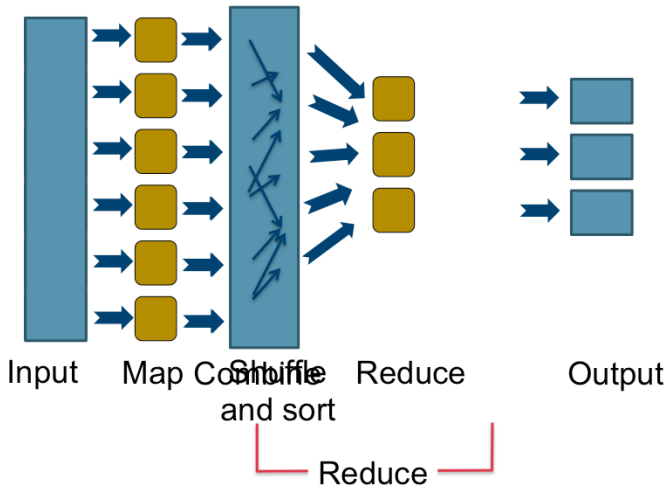


MapReduce programs perform a sequence transformations on Key-Value pairs.

Here are the major components you have to be familiar with when programming for MapReduce.

The Map function performs independent record transformations, including the potential for dropping or replicating records. The map function receives a key-value pair of some generic types, which we'll represent as K1 and V1, respectively, and it outputs a list of 0 or more key-value pairs of some other types, K2 and V2. The input types and output types are often different, but they can be the same. You might be thinking, "Well, not all data is naturally structured as key-value pairs," but the MapReduce framework accommodates this,

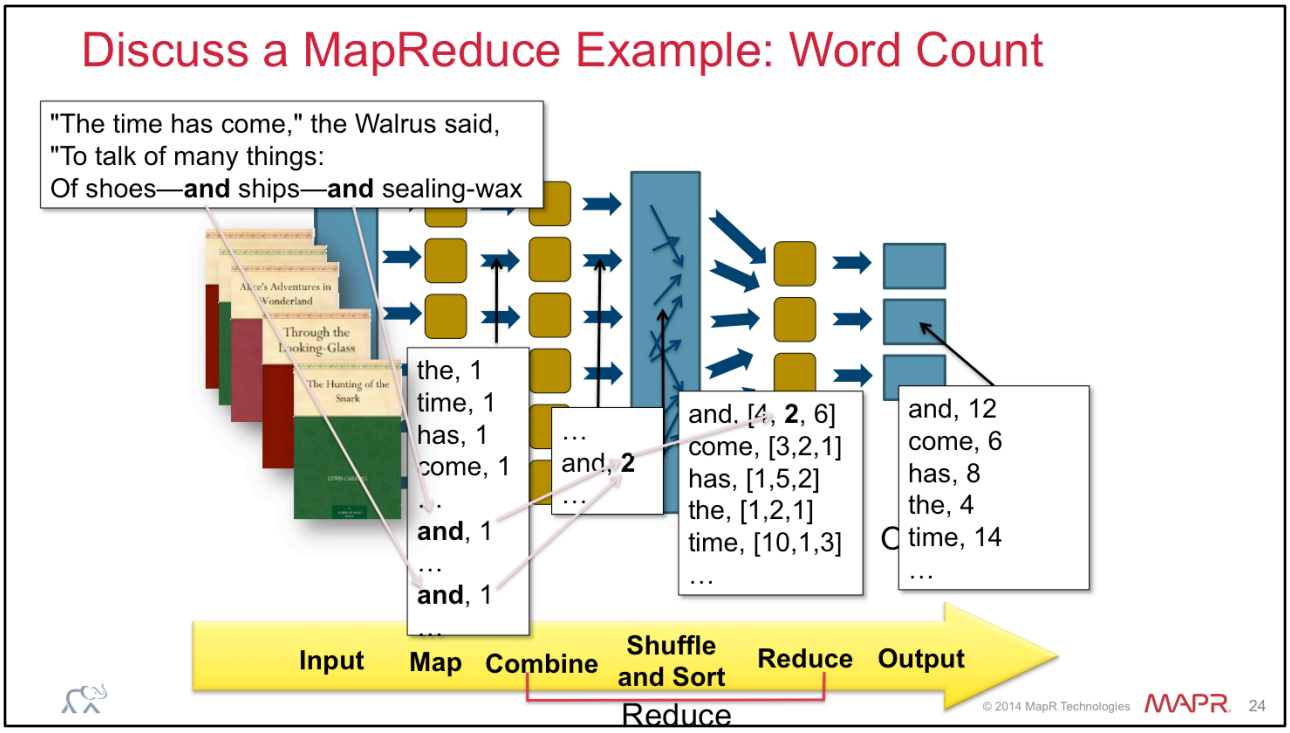
Describe the MapReduce Flow



Let's step through the flow of a MapReduce.

We start off with our input, maybe one file, maybe many files.

- The framework breaks up the input into "splits". Each split contains many records. Records can be any type of information: text, audio data, structured records, or anything. Each split typically corresponds to a block of data on a node where the data resides, but the programmer doesn't need to be aware of this.
- Each split is processed by a map task. Each record in a split is passed, independently of other records, to the map() function. The map() function receives each record as a key-value pair, although in some cases it might only be interested in either the keys or the values. The Mapper emits key-value pairs in response to the input record. It can emit zero records, or it can emit lot of records.



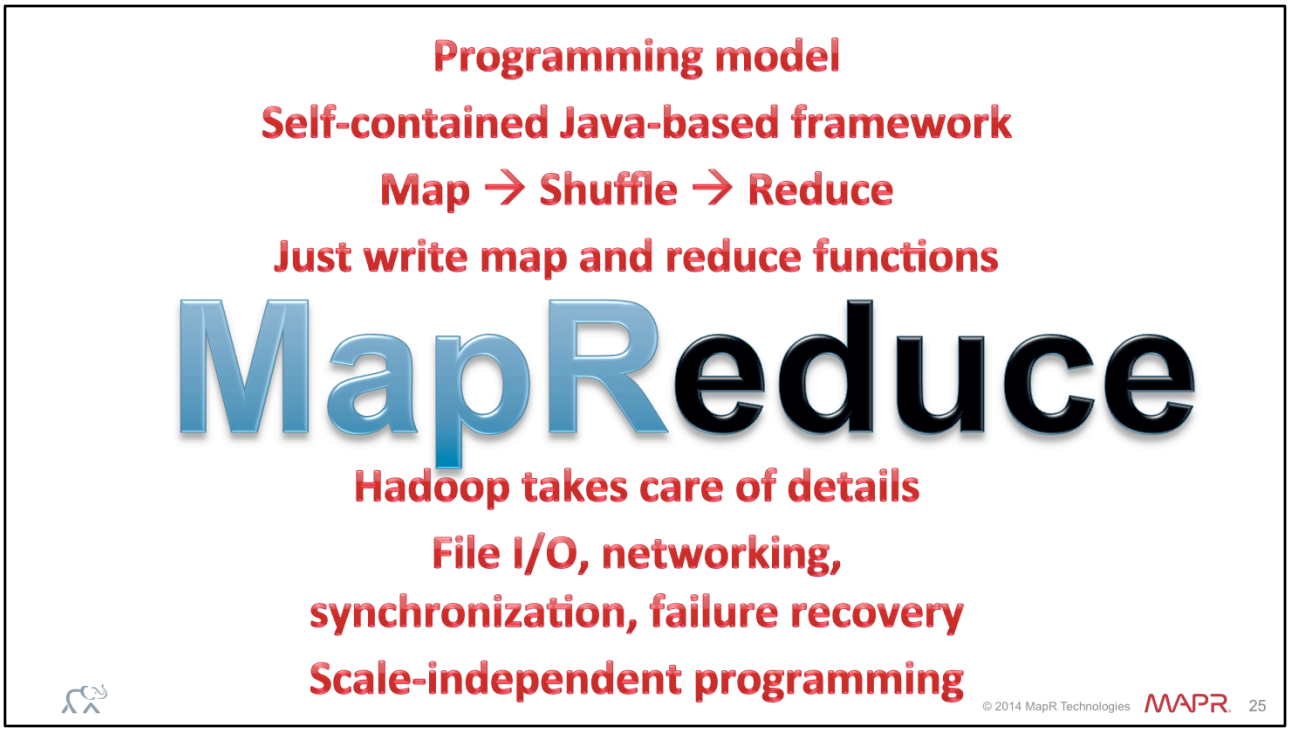
Let's step through an example word-count program using MapReduce to count the occurrences of each word in a set of input text files.

WordCount is like the "Hello World" program for MapReduce, but it's not useless, because counting particular strings is a critical part of lots of real-world programs.

1. As input, let's say we have all Lewis Carroll's books, and we want to count the occurrence of every word.

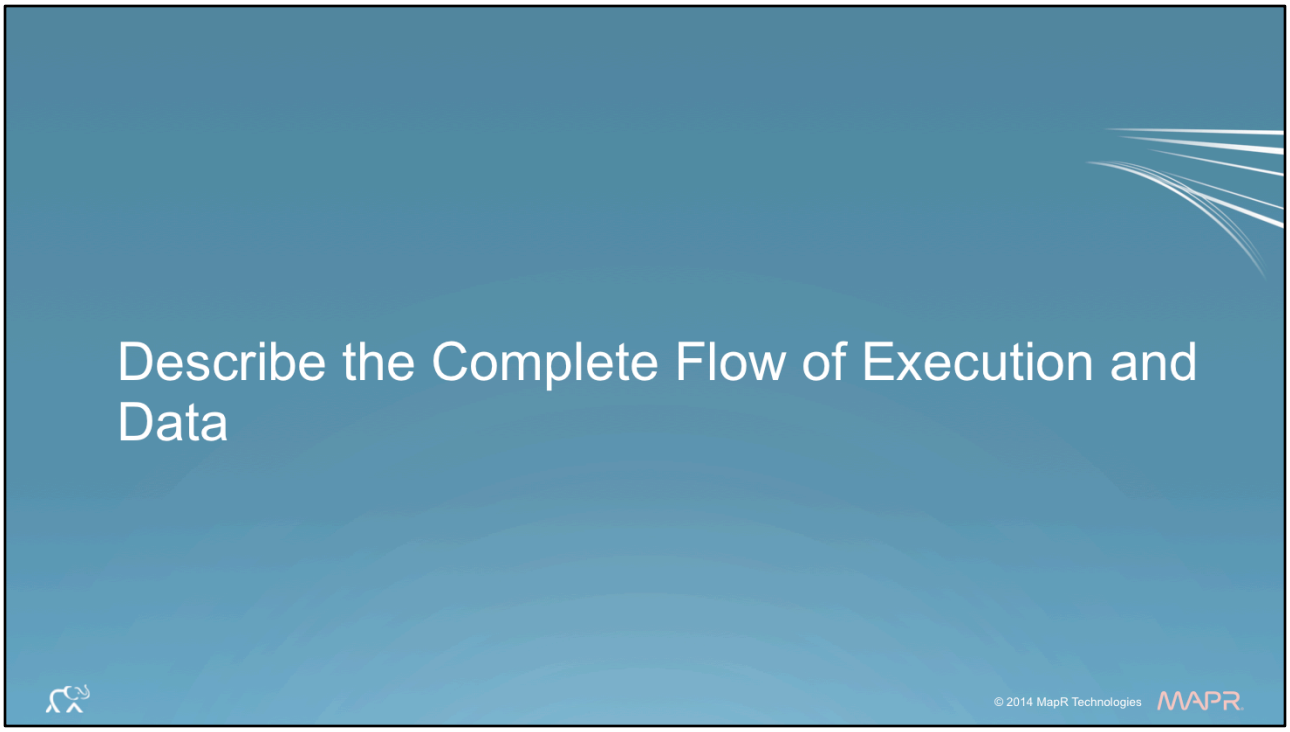
2. Hadoop divides this into "input splits."

3. One of the nodes contains Tweedledee's poem, "The time has come to talk of many things" and so forth. In the case of text input, every line of text is a record. Each record gets fed



Let's review MapReduce concepts:

- MapReduce is a programming model for distributed computing. Hadoop is based in Java, but you can develop in other languages too.
- Hadoop is self-contained. All packages you need to write MapReduce programs are included, so you don't need to know other frameworks before you can start with Hadoop.
- MapReduce programs execute in three stages: A Map stage, a Shuffle stage, and a Reduce stage.
- In the simplest case, you just write a Map and Reduce function.
- Hadoop takes care of issuing tasks via remote procedure call, verifying task completion, and copying data around the cluster. So your code doesn't get cluttered trying to manage …
- File I/O, process synchronization, or recovering from failures.



In this section, we will examine the complete flow of data and execution through the MapReduce framework.

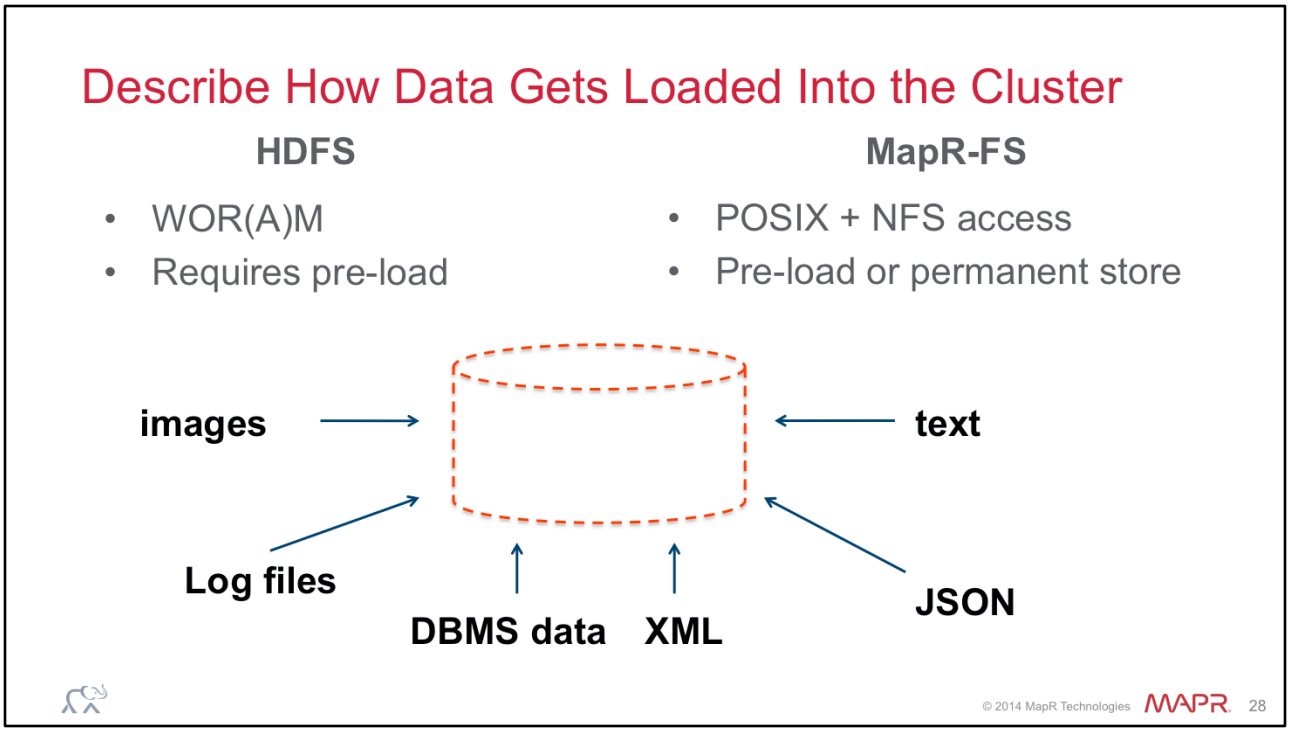
Define a Simplified MapReduce Workflow

- ***Load*** data into the cluster
- ***Analyze*** the data
- ***Store*** results in the cluster
- ***Read*** the results from the cluster

NOTE: MapReduce jobs themselves are often run in series – each job transforming the data in a particular way.

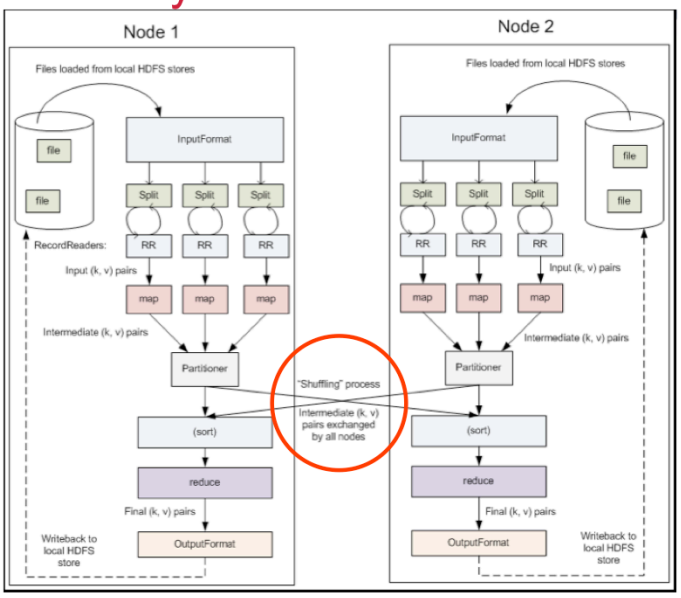
An over-simplified version of a MapReduce workflow is defined in the slide above. You must first load the data into the cluster (or use MapR-FS to store the data in production). You analyze the data with MapReduce and store the results in the HDFS/ MapR-FS file system. You then perform your business logic analysis on the results that are read in from the cluster.

Note that MapReduce is a simplified programming model and as such, you usually need to write several MapReduce jobs to completely analyze your data.



How does the data get into the distributed file system in the first place? There are several tools available for bringing in your data from various structured and unstructured data sources. For example, the Sqoop ecosystem tool can be used to import SQL data into your distributed file system. Flume can be used to import log data into your distributed file system. Also note that since MapR-FS is fully POSIX-compliant, read-write file-system with underlying mirroring/snapshotting capability and NFS support, you could host your production data on MapR-FS rather than only use it for staging your input. With that in mind, you could consider HDFS as analogous to a CDROM whereas MapR-FS is analagous to a disk.

Describe a Summary of Execution and Data Flow

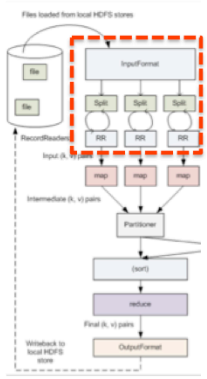


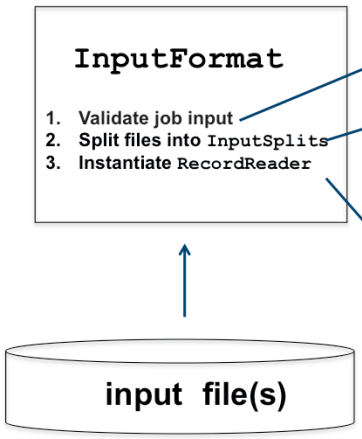
http://developer.yahoo.com/hadoop/tutorial/module4.html

The graphic above depicts the complete flow of data and execution for a MapReduce job. In summary,

1. Data is loaded from the HDFS file system
2. The job defines the input format of the data
3. Data is split between different map() methods running on all the nodes
4. Record readers parse out the data into key-value pairs that serve as input into the map() methods
5. The map() method produces key-value pairs that are sent to the partitioner
6. When there are multiple reducers, the mapper creates one partition for each reduce task.
7. The key-value pairs are sorted by key in each partition
8. The reduce() method takes the intermediate key-value pairs and reduces them to a final list of key-value pairs
9. The job defines the output format of the data

Define the `InputFormat` Class





Does the input exist?

One `InputSplit` per mapper

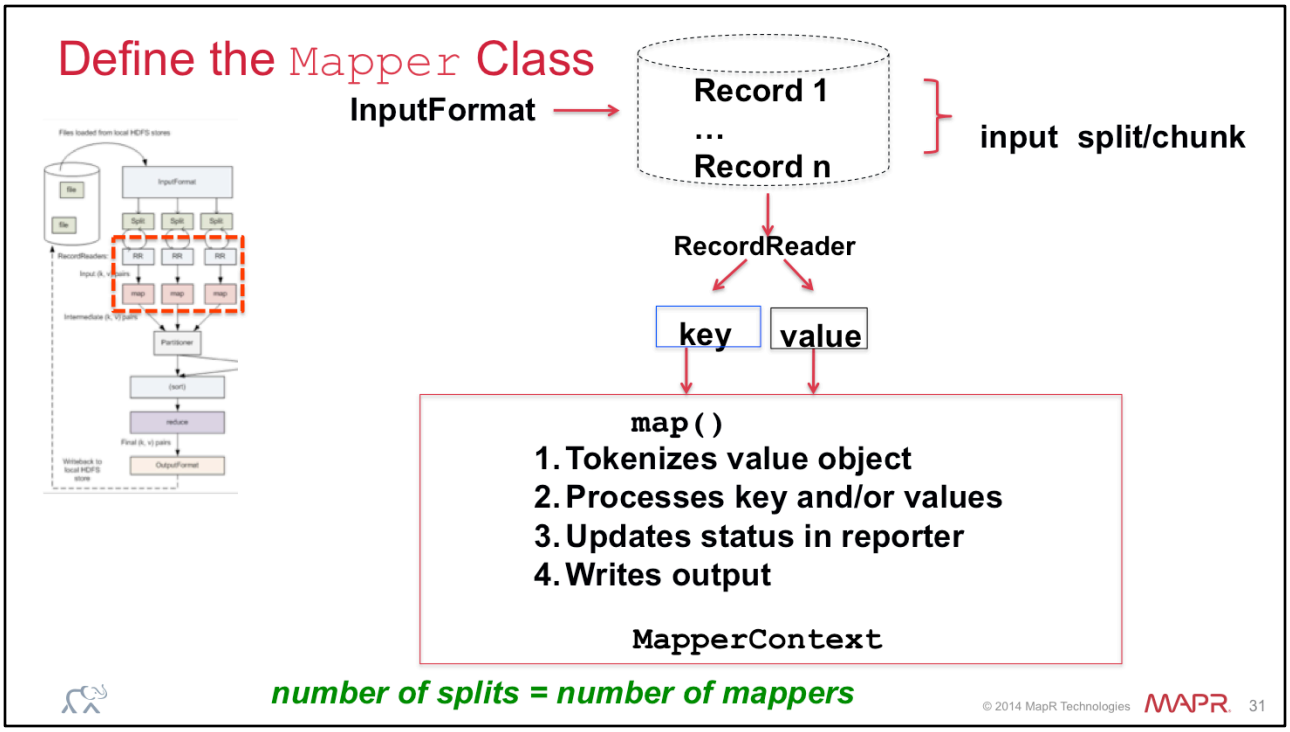
NOTE: these are logical splits – physical blocks are already written to data nodes

Iterates over records in split

As described in the slide above, the InputFormat object is responsible for validating the job input, splitting the files amongst the mappers, and instantiating the RecordReader.

By default, the size of an input split is equal to the size of the block. In Hadoop, the default block size is 64M. In MapR, the equivalent structure is called "chunk" and has default size 256M.

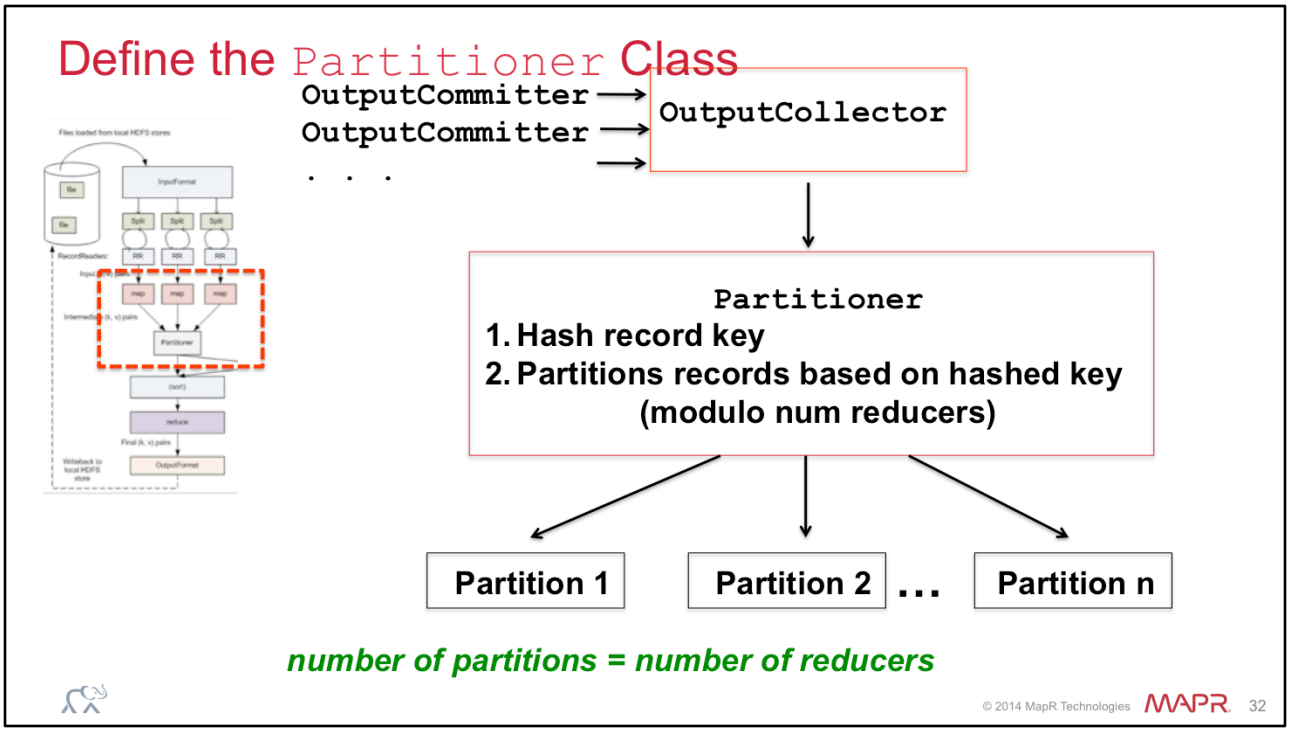
Each InputSplit references a set of records, each of which will be broken up into a key and value for the mapper. The work of splitting up the data is done before your map() process even begins running. The job execution framework will attempt to place the mapping task as close to the data as possible. Once the task is assigned to a node, that TaskTracker passes the InputSplit to the RecordReader constructor. The RecordReader



The map() method is implemented as part of the Mapper interface. The map() method has three arguments: Writable key, Writable value, and context.

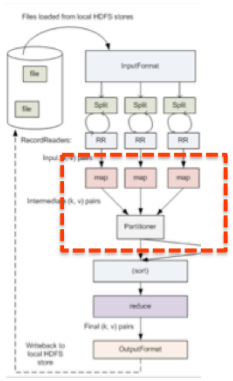
The RecordReader defines the key for the map() method as the byte offset of the record in the input file, and the value is simply the line at the byte offset. A reporter is provided to the map() method and optionally used to update status.

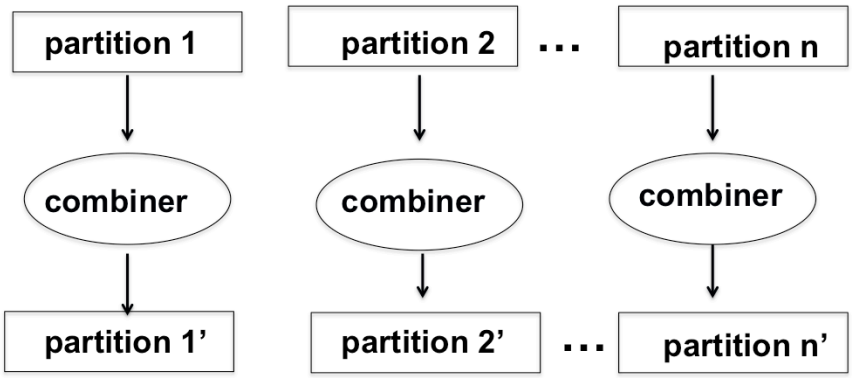
The map method tokenizes the input value and then processes those tokens. For each token, the map() method may write the key and value to the OutputCommitter. The OutputCommitter object collects the output from all the calls to the map method and then passes that data to the Partitioner, described next.



The partitioner takes the output generated by the map() method, hashes the record key, and creates partitions based on the hashed key. Each partition is destined for a single reducer, so all the records with the same key will be written to the same partition. This is the behavior of the default partitioner – you could override that and provide your own behavior.

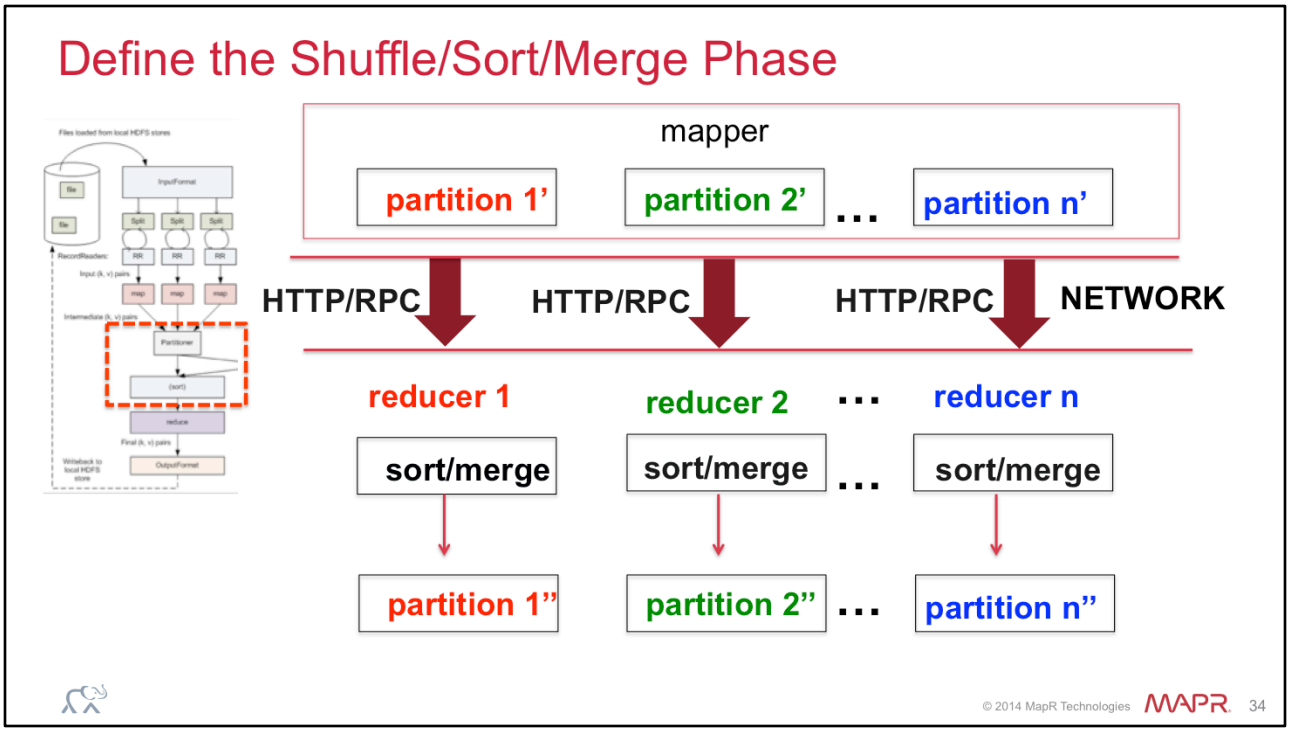
Define the (Optional) `Combiner` Class





 33

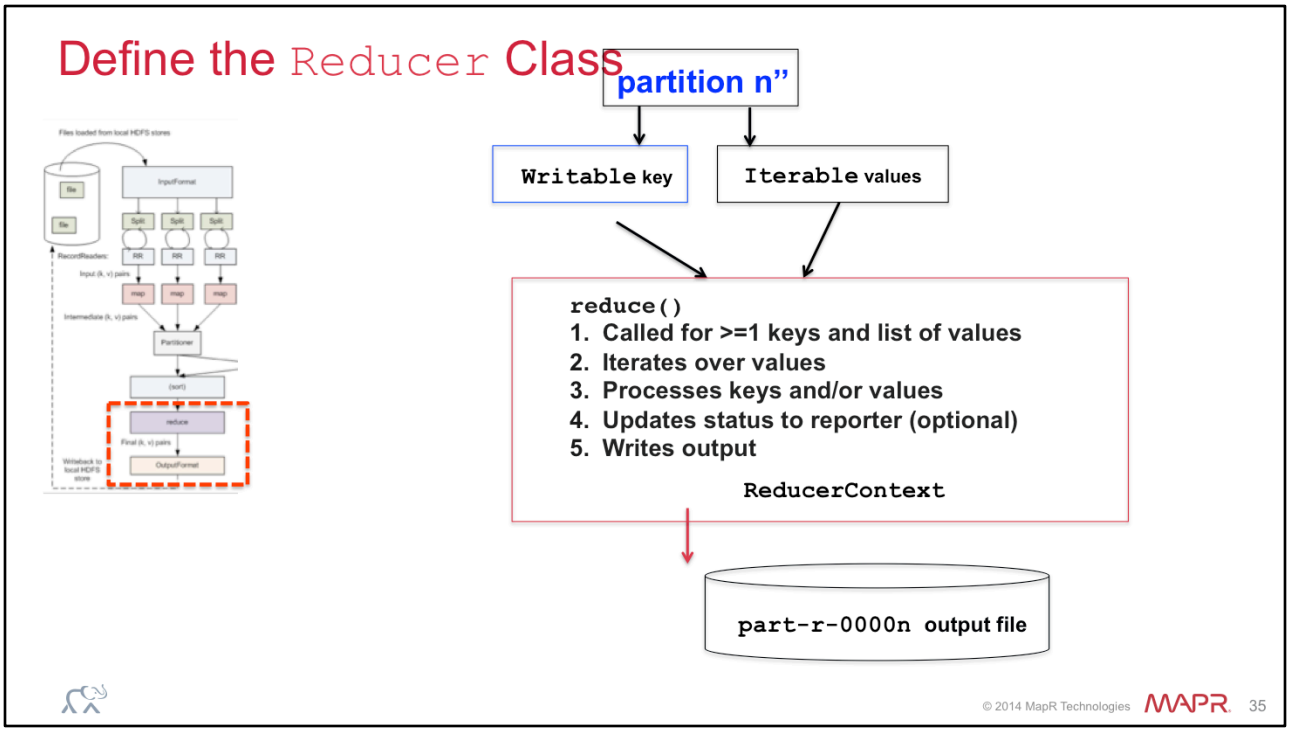
There is no default implementation of the combiner – you have to implement yourself and set it in the job configuration. If the combiner function is defined, then it is invoked before the output file is written. The motivation for implementing a combiner is to reduce the intermediate values of the mappers before they are sent over the network to enhance performance. Usually the reducer is used as the combiner, but it requires that the reducer function be both associative and commutative. In other words, if the reducer function provides the same output regardless of the order and grouping of the inputs, then you can repurpose the reducer as the combiner.



The partitions now are sorted and merged in preparation for sending to the reducers. Once an intermediate partition is complete, it is sent to the reducer over the network using a protocol (Hadoop uses HTTP whereas MapR uses RPC). The available number of threads the reduce task has for performing this step is 5 by default but is configurable.

When a map task is finished, it signals its parent task tracker which in turn notifies the jobtracker. For a given job, the jobtracker knows the mapping between map outputs and task trackers. The reducer periodically asks the job tracker for map until it has retrieved them all.

Task trackers wait until they are told to delete thee intermediate results by the job tracker, which is after the job has completed. The map outputs are copied to the reduce task



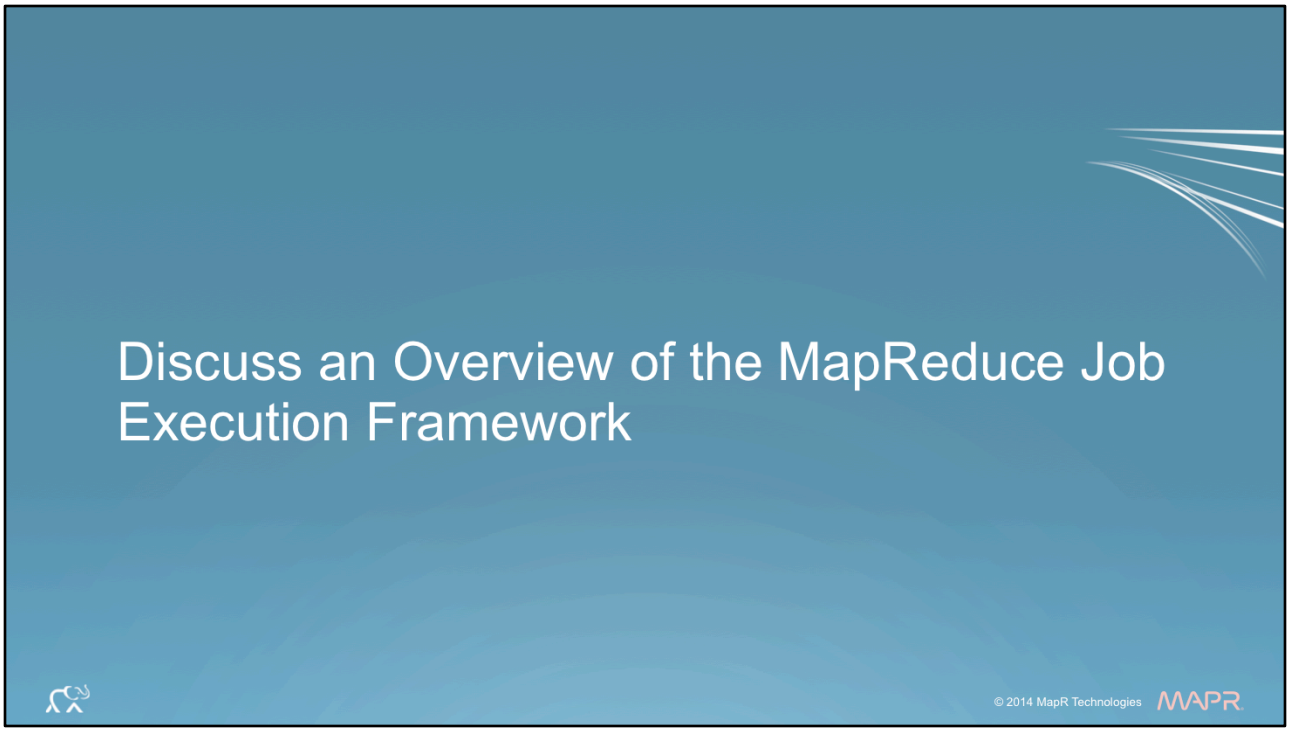
The reduce() method is called for each key and the list of values associated with that key. The reduce() method processes each iterated value and writes the key and reduced list to the output collector. The key, while writable, should not be altered by the reduce() method as the framework will reuse this key until the job is complete.

Identify the Results From a MapReduce Job

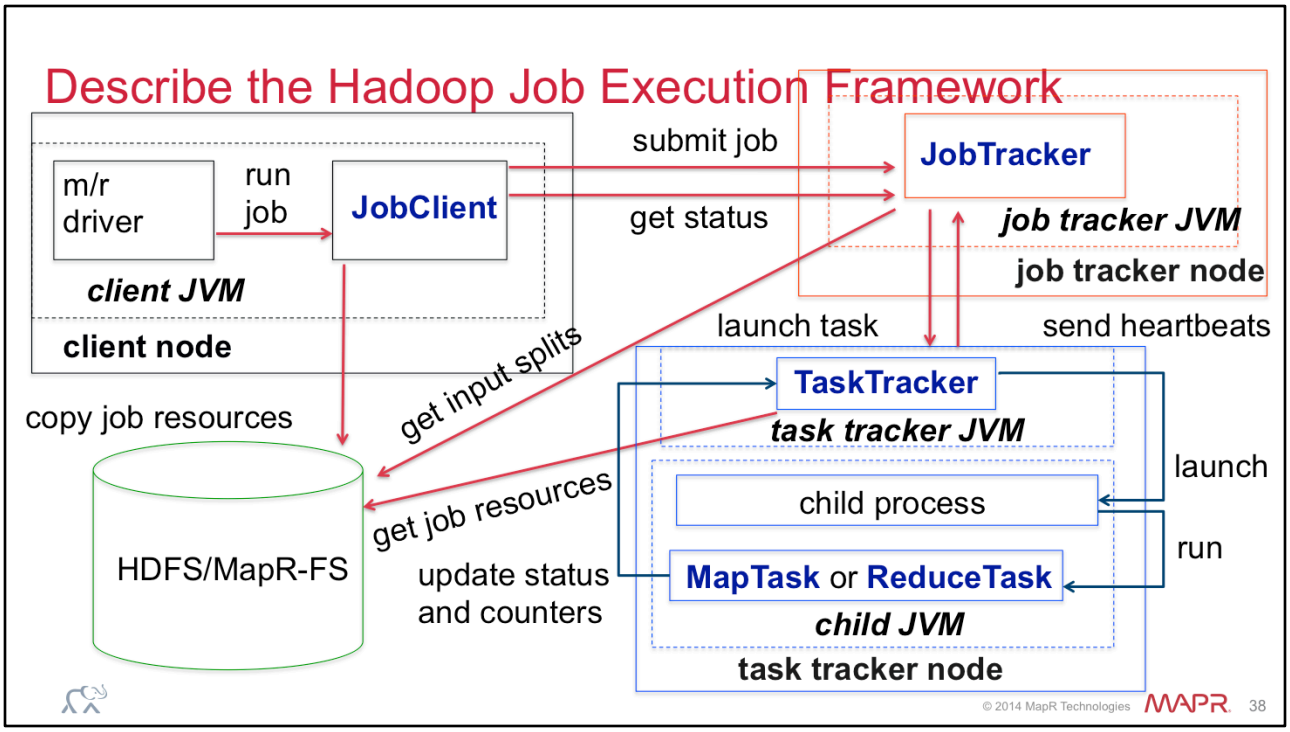
- **Output directory contents:**
  - `_SUCCESS`
  - `_logs/history*`
  - `part-r|m-00001`
  - `part-r|m-00002`
  - `...`
  - `part-r|m-n`
- **Description of output files:**
  - key-value pairs
  - one file per reducer
  - format defined by user

The results of a Map-Reduce job are written to a directory specified by the user. The contents of this directory are identified in the slide above. An empty file named _SUCCESS is created to indicate that the job completed successfully (but not necessarily without errors). A trace of each reduce() method is captured in the _logs/history* files. The output of the reduce() method itself is captured in files named part-r-00001, part-r-00002, … (one for each reducer). By keeping the files distinct for each reducer, there is no need to account for concurrency.

The format of the output files is in the form of one key-value pair per line. The actual data format of the key and value is defined by the programmer in the reduce() method. Both text and binary (BLOB) data types are supported.



In this section, we discuss an overview of the MapReduce job execution framework.



The graphic above describes how Hadoop executes map-reduce jobs. A user runs a map-reduce (m/r) program on the client node which instantiates a JobClient object. The JobClient submits the job to the JobTracker. The job tracker instantiates a Job object which gets sent to the appropriate task tracker. The task tracker launches a child process which in turns runs the map or reduce task. The task continuously updates the task tracker with status and counters.

Discuss How the Heartbeat is Used

- **Heartbeats tell JT which TTs are alive**
- **When JT stops receiving heartbeats from a TT**
  - JT reschedules tasks on failed TT to other TTs
  - JT marks TT as down (won't schedule subsequent tasks)
- **Heartbeat also contains the following info:**
  - Task status
  - Task counters
  - Data read/write status (aka "block reports")

When task trackers send heartbeats to the job tracker, they include other information as described in the slide above. Similarly, data nodes piggyback their heartbeats with status reports (called block reports). Usually, a task tracker serves as a data node too.

Discuss Hadoop Job Scheduling

- **2 schedulers available in Hadoop:**
  - Fair scheduler
  - Capacity scheduler
- **Defined by**
  `mapred.jobtracker.taskScheduler` (`mapred-site.xml`)
- **Some terms to define:**
  - FIFO ➔ first-in first-out
  - Pre-empt ➔ kill one running program to let another program run instead

Fair scheduler (default)
Resources shared evenly across pools
Each user has its own pool by default
Can configure custom pools
Can configure guarenteed minimum access to pools (prevent starvation)
Support pre-emption

Capacity scheduler
Resources shared across queues
Admin configures hierarchical queues to reflect organizations and their weighted access to resources
Can configure soft and hard capacity limits to users within a queue
Queues have ACLs to prevent rogues from accessing queue
Supports resource-based scheduling (memory)

Define the Hadoop Fair Scheduler

- **Pools**
  - Pool = set of jobs
  - User-configurable priority between jobs within the same pool
  - Default 1 user per pool
- **Scheduling algorithm**
  - Divide each pool's min maps/reduces among jobs
  - When slot is free, allocate to a job below min share (or most starved)
  - Pre-empt long-running jobs to meet minimum guarantees
- **Per-pool configuration** (`fair-scheduler.xml`)
  - `minMaps`, `minReduces`
  - `maxRunningJobs`
  - `minSharePreemptionTimeout`

Goal of the fair scheduler is to give all users equitable access to pool resources

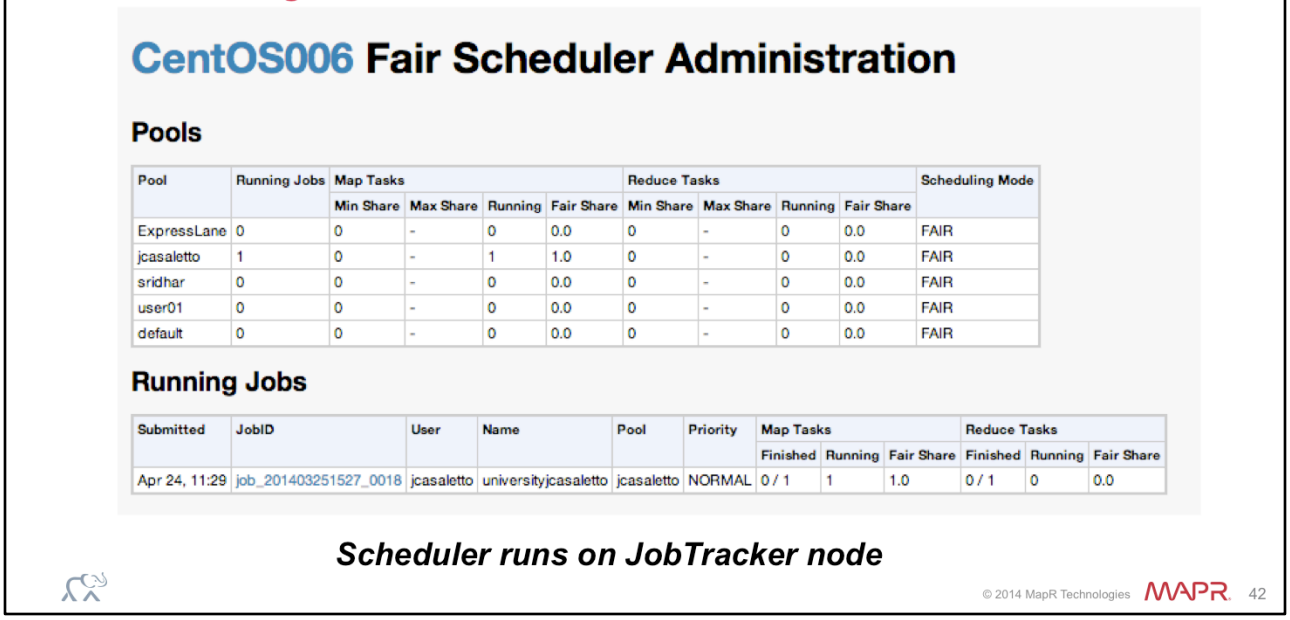
Each pool has configurable guaranteed capacity (= slots)

Jobs placed in flat pools (default = 1 pool per user)

Jobs from "over-using" users can be preempted

developed at Facebook

Examining the Fair Scheduler in MCS



CentOS006 Fair Scheduler Administration

Pools

Pool	Running Jobs	Map Tasks				Reduce Tasks				Scheduling Mode
		Min Share	Max Share	Running	Fair Share	Min Share	Max Share	Running	Fair Share	
ExpressLane	0	0	-	0	0.0	0	-	0	0.0	FAIR
jcasaletto	1	0	-	1	1.0	0	-	0	0.0	FAIR
sridhar	0	0	-	0	0.0	0	-	0	0.0	FAIR
user01	0	0	-	0	0.0	0	-	0	0.0	FAIR
default	0	0	-	0	0.0	0	-	0	0.0	FAIR

Running Jobs

Submitted	JobID	User	Name	Pool	Priority	Map Tasks			Reduce Tasks		
						Finished	Running	Fair Share	Finished	Running	Fair Share
Apr 24, 11:29	job_201403251527_0018	jcasaletto	universityjcasaletto	jcasaletto	NORMAL	0 / 1	1	1.0	0 / 1	0	0.0

Scheduler runs on JobTracker node

You can examine the configuration and status of the Hadoop fair scheduler in the MCS (or by using the Web UI running on the job tracker), as shown in the slide above.

Define the Hadoop Capacity Scheduler

- **Queue**
  - Queue = set of jobs
  - May be hierarchically organized
  - Shares assigned to queues as percentages of total cluster resources
- **Scheduling algorithm**
  1. Allocate slots to queues based on percentage of shares
  2. FIFO scheduling within each queue
- **Configuration** (`capacity-scheduler.xml`)
  - per-queue configuration (e.g. priority support)
  - Per-user configuration (e.g. max jobs per user)

The goal of the capacity scheduler give all queues access to cluster resources

Each queue has configurable guaranteed capacity (= slots)

Jobs placed in hierarchical queues (default = 1 queue per cluster)

Jobs within queue are FIFO

developed at Yahoo

Identify Limitations in the Hadoop Execution Framework

Aspect	Limitation
scalability	Single job tracker restricts job throughput
availability	Single job tracker and namenode introduces SPOF
inflexibility	Map and reduce slots are not interchangeable
scheduler optimization	Framework does not optimize scheduling of jobs
program support	Framework is restricted to map and reduce programs

MapReduce v2 (aka YARN) specifically addresses some of these issues

The table above identifies some of the limitations of the Hadoop execution framework. Some of these limitations are addressed in the MapR distribution – discussed later in this section. Inflexlbility and program support are being addressed by YARN.