Map/Reduce

Sharma Chakravarthy
Information Technology Laboratory
Computer Science and Engineering Department
The University of Texas at Arlington, Arlington, TX 76009
Email: sharma@cse.uta.edu
URL: http://itlab.uta.edu/sharma

Acknowledgements

- These slides are put together from a variety of sources (both papers and slides/tutorials available on the web)
- Mostly I have tried to: provide my perspective, emphasize aspects that are of interest to this course, and have tried to put forth a consolidated view of Map/Reduce

Tutorial Outline

- ***Map/reduce***
- ***Hadoop***

Data Center as a computer [Patterson, cacm 2008]

- Claim: There are dramatic differences between – developing software for millions to use as a service versus distributing software for millions to run on their PCs
  - Availability, dependability
  - Bandwidth (with low latency) to service large number of users
  - Innovation is fast as the software in their control!
- This has led to distributed data centers

Data Center as a computer [Patterson, cacm 2008]

- What are the useful programming abstractions for such a large system?
- How do thousands of computers behave differently from a small system?
- What must you do differently to run an abstraction on thousands of computers?
- Google proposed a two-phase primitive:
  - Phase 1: maps a user supplied function onto thousands of computers
  - Phase 2: Reduces the returned values from all those instances into a set of results

Map/Reduce

- Is a programming paradigm
- Is a parallel programming paradigm
- Is derived from functional programming (specification vs. procedural programming) (remember SQL is non-procedural)
- Many a times the question asked is:
  - Is there a difference between Map/Reduce and traditional parallel programming?

Data Center as a computer [Patterson, cacm 2008]

- Map/Reduce was born
- Runs on heterogeneous computers (even different generations)
- Runs on heterogeneous OSs
- Scheduler is dynamic and accommodates above (as compared to batch schedulers of Grid computing
- Failures are handled transparently!
- See http://sortbenchmark.org/ for more details
- Google regenerated its index using Map/Reduce
- Fairly easy to program, easy to understand!

Map/Reduce

- Map/Reduce was developed within Google as a mechanism for processing large amounts of raw data; for example, crawled documents or web request logs.
- This data is so large, it must be distributed across thousands of machines in order to be processed in a reasonable time.
- This distribution implies parallel computing since the same computations are performed on each CPU, but with a different dataset.
- Map/Reduce is an abstraction that allows Google engineers to perform simple computations while hiding the details of parallelization, data distribution, load balancing and fault tolerance.

What is Map/Reduce?

- Data-parallel programming model for clusters of commodity machines
- Pioneered by Google
  - Processes 20 PB of data per day
- Popularized by open-source Hadoop project
  - Used by Yahoo!, Facebook, Amazon, …

What is Map/Reduce used for (2)?

- Also used for:
  - Graph mining
  - PageRank calculation
  - Machine learning
  - Shortest path
- Problems are being ported to map/reduce on a daily basis
- Is a popular research area!
- Porting algorithms into map/reduce is not always straightforward!

Note that most/all of these applications are very different from traditional DBMS applications:
very large data sizes
one time computation versus data storage and management
SQL is not a good vehicle/tool for doing this task
Defining schema and loading this data into a DBMS is difficult

What is Map/Reduce used for?

- At Google:
  - Index building for Google Search
  - Article clustering for Google News
  - Statistical machine translation
- At Yahoo!:
  - Index building for Yahoo! Search
  - Spam detection for Yahoo! Mail
- At Facebook:
  - Data mining
  - Ad optimization
  - Spam detection

Note that most/all of these applications are very different from traditional DBMS applications:
very large data sizes
one time computation versus data storage and management
SQL is not a good vehicle/tool for doing this task
Defining schema and loading this data into a DBMS is difficult

What is Map/Reduce used for?

- In research:
  - Analyzing Wikipedia conflicts (PARC)
  - Natural language processing (CMU)
  - Bioinformatics (Maryland)
  - Astronomical image analysis (Washington)
  - Ocean climate simulation (Washington)
  - Graph Mining (UTA)
  - Storm Identification from rainfall data (UTA)

MapReduce Design Goals

1. **Scalability** to large data volumes:
   - Scan 100 TB on 1 node @ 50 MB/s = 23 days
   - Scan on 1000-node cluster = 33 minutes
2. **Cost-efficiency:**
   - Commodity nodes (cheap, but unreliable)
   - Commodity network
   - Automatic fault-tolerance (fewer admins)
   - Easy to use (fewer programmers)

Typical Hadoop Cluster

Map/Reduce

- Automatic parallelization & distribution
- Fault-tolerant
- Provides status and monitoring tools
- Clean abstraction for programmers
- Borrows from functional programming
- Users implement interface of two functions:

map (in_key, in_value) → (int_key, intermediate_value list)

reduce (int_key, intermediate_value list) → (out_key, value list)

In_key, int_key, and out_key need not be same!

The Basics (5)

- Note that we cannot assume that every problem can be parallelized
- Some problems are easier to parallelize and some are inherently sequential
- So it is important to understand whether a problem can be parallelized and to what extent!

- Cryptography hash-chaining computations are very difficult to parallelize
- Parallelizing I/O is difficult (needs additional technology)

Map/Reduce

- Map/Reduce was developed within Google as a mechanism for processing large amounts of raw data; for example, crawled documents or web request logs.
- This data is so large, it must be distributed across thousands of machines in order to be processed in a reasonable time.
- This distribution implies parallel computing since the same computations are performed on each CPU, but with a different dataset.
- Map/Reduce is an abstraction that allows Google engineers to perform simple computations while hiding the details of parallelization, data distribution, load balancing and fault tolerance.

MapReduce Programming Model

- Data type: key-value *records*
- Map function:

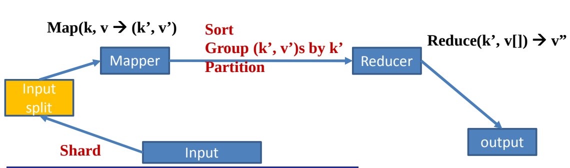
$$(K_{in}, V_{in}) \rightarrow list(K_{inter}, V_{inter})$$

- Reduce function:

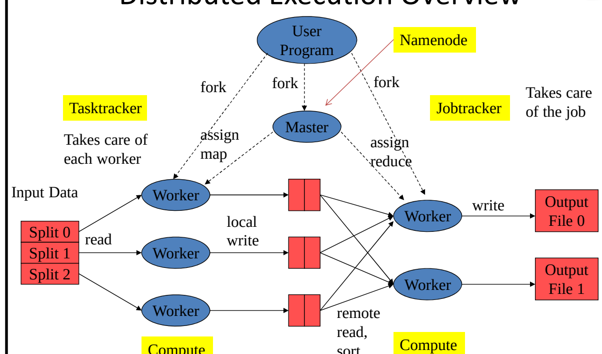
$$(K_{inter}, list(V_{inter})) \rightarrow list(K_{out}, V_{out})$$

MapReduce

- Input: a set of key/value pairs (generated from input data)
- User supplies two functions:
  - map(k,v) → list of (k1,list(v1))
  - reduce(k1, list(v1)) → list of (k2, v2)



Distributed Execution Overview

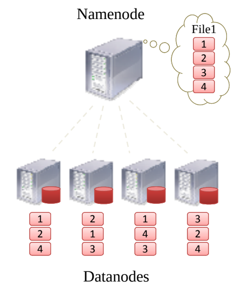


Map/Reduce: Approach

- The MASTER:
  - initializes the data and splits it up according to the number of available WORKERS (# of mappers can be chosen)
  - Sends each WORKER its partition
  - receives the results from each WORKER (its location and some meta information)
- The WORKER:
  - receives the partition (actually its location) from the MASTER
  - performs processing on the partition
  - returns results to MASTER

Hadoop Distributed File System

- Files split into 64MB *blocks (default)*
- Blocks replicated across several *datanodes* (usually 3)
- Single *namenode* stores metadata (file names, block locations, etc)
- Optimized for large files, sequential reads



Map/Reduce

- Map, written by a user of the Map/Reduce library, takes an input pair and produces a set of intermediate key/value pairs.
- The Map/Reduce library groups together all intermediate values associated with the same intermediate key k and passes them to the reduce function.
- Shuffle is an important, transparent step
- The reduce function, also written by the user, accepts an intermediate key k and a set of values for that key. It merges/aggregates these values to form a possibly smaller set of values.

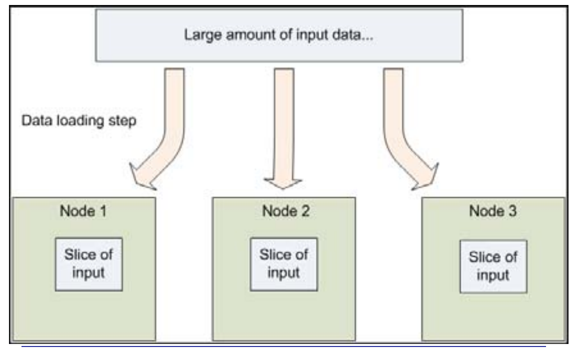
Data Flow

- Input, final output are stored on a distributed file system (e.g., HDFS)
- Scheduler tries to schedule map tasks “close” to physical storage location of input data
- Intermediate results are stored on local FS of map and reduce workers
- Output is often input to another map reduce task
- Not all problems solved with one round of map/reduce computation!

Dealing with Failures

- Map worker failure
  - Map tasks completed or in-progress at worker are reset to idle
  - Failed map tasks are restarted
- Reduce worker failure
  - Only in-progress tasks are reset to idle
  - Failed reduce tasks are restarted
- Master failure
  - MapReduce task is aborted and client is notified

Data distribution across nodes

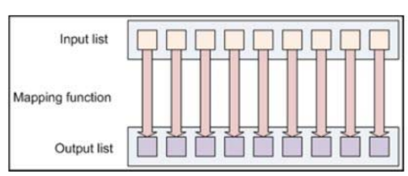


Map/Reduce Execution Overview

- Map invocations are distributed across multiple machines by automatically partitioning the input data into a set of M sp lits/shards. The input shards can be processed in parallel on d ifferent machines.
- Reduce invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., hash(key) mod R).
- The number of partitions (R) and the partitioning function ca n be specified by the user
- The illustration below shows the overall flow of a Map/Reduc e operation. When the user program calls the Map/Reduce f unction, the following sequence of actions occurs (the numb ered labels in the illustration correspond to the numbers in the list below).

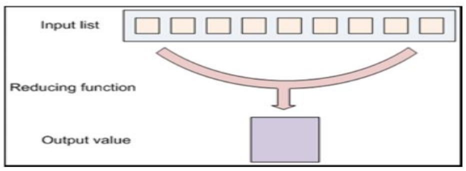
Mapping Lists

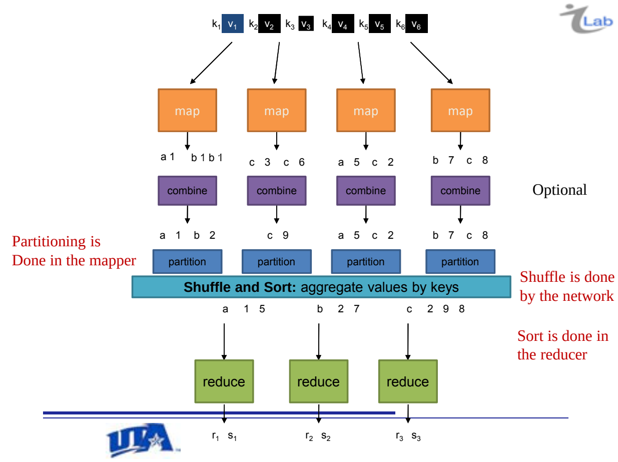
- A list of data elements are provided, one at a time, to a function called mapper, which transforms each element individually to an output element



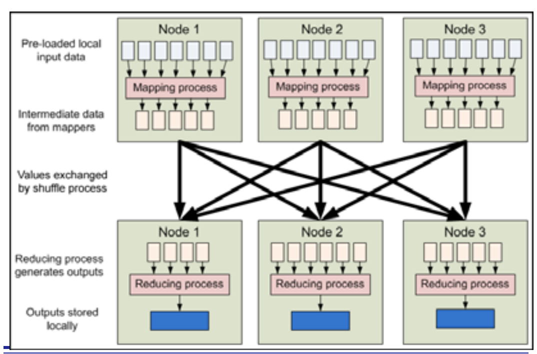
Reducing List

- Reducing lets you aggregate values together. A reducer function receives an iterator of input values from an input list. It them combines these values together, returning an output value





Shuffling in Map/Reduce



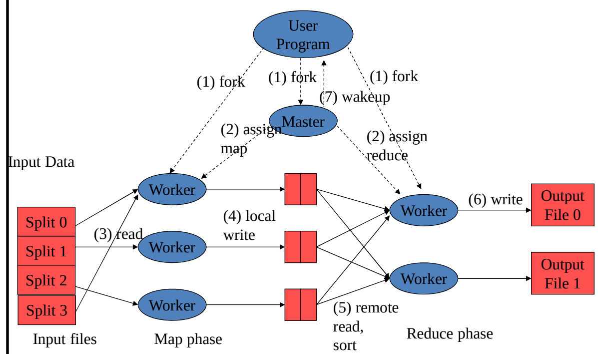
Map/Reduce

- Static load balancer: allocates processes to processors at run time while taking no account of current network load.
- Map, written by a user of the Map/Reduce library, takes an input pair and produces a set of intermediate key/value pairs.
- The Map/Reduce library groups together all intermediate values associated with the same intermediate key *I* and passes them to the reduce function.
  - Shuffle is an important, transparent step
- The reduce function, also written by the user, accepts an intermediate key I and a set of values for that key. It merges together these values to form a possibly smaller set of values.
  - See OSDI 2004 paper

Coordination

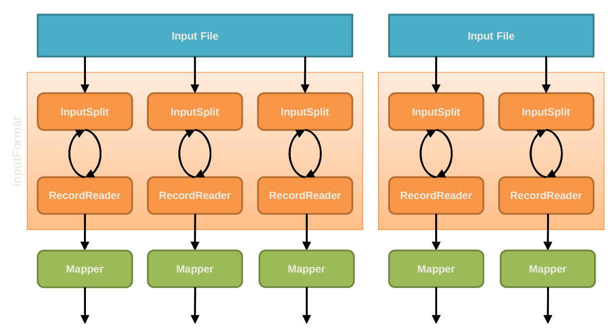
- Master data structures
  - Task status: (idle, in-progress, completed)
  - Idle tasks get scheduled as workers become available
  - When a map task completes, it sends the master the location and sizes of its R intermediate files, one for each reducer
  - Master pushes this info to reducers
- Master pings workers periodically to detect failures

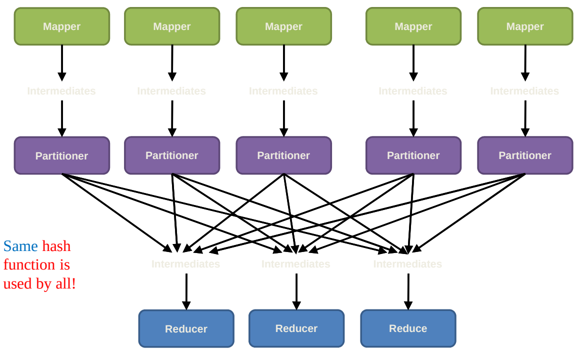
Distributed Execution Overview



Map/Reduce execution overview

- The Map invocations are distributed across multiple machines by automatically partitioning the input data into a set of M splits or *shards*. The input shards can be processed in parallel on different machines.
- Reduce invocations are distributed by partitioning the intermediate key space into R pieces using a partitioning function (e.g., hash(key) mod R).
- The number of partitions (R) and the partitioning function can be specified by the user
- The illustration below shows the overall flow of a Map/Reduce operation. When the user program calls the Map/Reduce function, the following sequence of actions occurs (the numbered labels in the illustration correspond to the numbers in the list below).





Same hash function is used by all!

Partition and shuffle

- After the map tasks have completed, the nodes may still be performing several more map tasks each. But they also begin exchanging the intermediate outputs from the map tasks to where they are required by the reducers.
- This process of moving map outputs to the reducers is known as *shuffling*.
- A different subset of the intermediate key space is assigned to each reduce node; these subsets (known as "partitions") are the inputs to the reduce tasks.
- Each map task may emit (key, value) pairs to any partition; all values for the same key are always reduced together regardless of which mapper is its origin. Therefore, the map nodes must all agree on where to send the different pieces of the intermediate data.

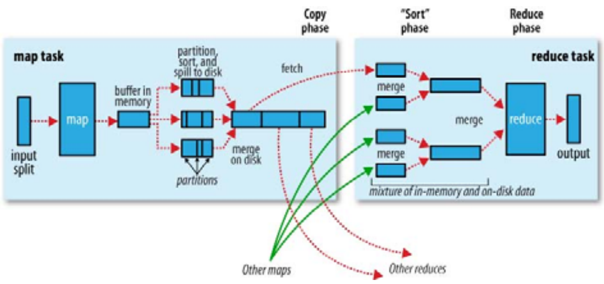
Shuffle and Sort in Hadoop

- Probably the most complex aspect of MapReduce!
- Map side
  - Map outputs are buffered in memory in a circular buffer
  - When buffer reaches threshold, contents are "spilled" to disk
  - Spills merged in a single, partitioned file (sorted within each partition): combiner runs here
- Reduce side
  - First, map outputs are copied over to reducer machine
  - "Sort" is a multi-pass merge of map outputs (happens in memory and on disk)
  - Final merge pass goes directly into reducer

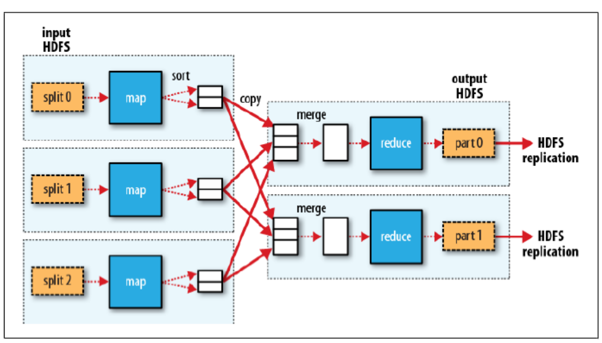
Partition and shuffle

- The *Partitioner* class determines which partition a given (key, value) pair will go to. The default partitioner computes a hash value for the key and assigns the partition based on this result.
- **Sort:** Each reduce task is responsible for reducing the values associated with several intermediate keys. The set of intermediate keys on a single node is automatically sorted by Hadoop before they are presented to the Reducer.

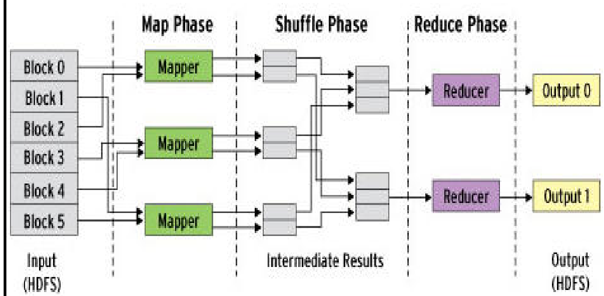
Partition, shuffle, sort



Partition, shuffle, sort

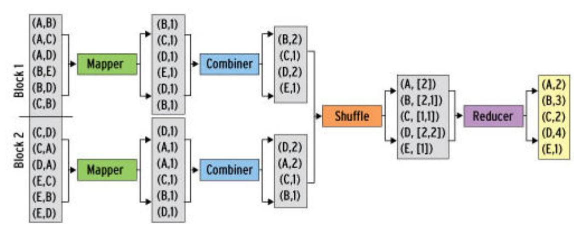


Map tasks and mapper nodes

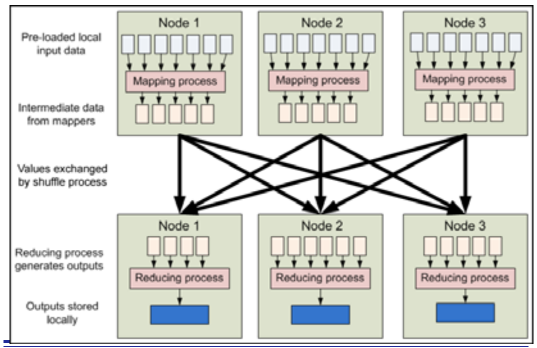


An example

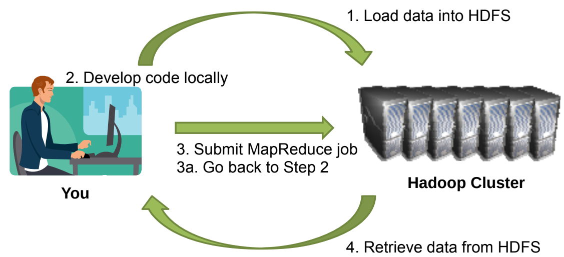
- Counting letters



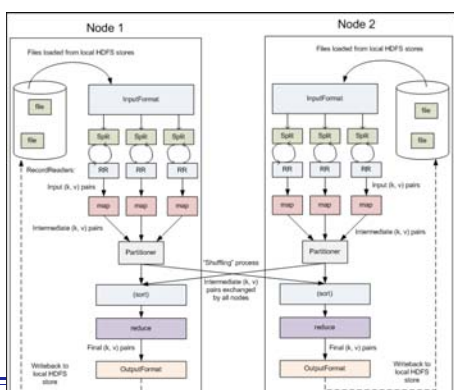
Map and reduce on nodes with data



Hadoop Workflow



A closer look



Input Files

- This is where the data for a Map/Reduce task is initially stored. While this does not need to be the case, the input files typically reside in HDFS.
- The format of these files is arbitrary; while line-based log files can be used, we could also use a binary format, multi-line input records, or something else entirely. It is typical for these input files to be very large -- tens of gigabytes or more.

Input Format

- How these input files are split up and read is defined by the InputFormat. An InputFormat is a class that provides the following functionality:
  - Selects the files or other objects that should be used for input
  - Defines the *InputSplits* that break a file into tasks
  - Provides a factory for *RecordReader* objects that read the file
- Several InputFormats are provided with Hadoop. An abstract type is called *FileInputFormat*; all InputFormats that operate on files inherit functionality and properties from this class.
- When starting a Hadoop job, FileInputFormat is provided with a path containing files to read. The FileInputFormat will read all files in this directory. It then divides these files into one or more InputSplits each. You can choose which InputFormat to apply to your input files for a job by calling the setInputFormat() method of the *JobConf*object that defines the job.

InputFormats provided by Map/Reduce

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 key, val 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

InputSplits

- An InputSplit describes a unit of work that comprises a single *map task* in a Map/Reduce program. A Map/Reduce program applied to a data set, collectively referred to as a *Job*, is made up of several (possibly several hundred) tasks. Map tasks may involve reading a whole file; they often involve reading only part of a file. By default, the FileInputFormat and its descendants break a file up into 64 MB chunks (the same size as blocks in HDFS). You can control this value.
- By processing a file in chunks, 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.

InputSplits

- Even more importantly, since the various blocks that make up the file may be spread across several different nodes in the cluster, it allows tasks to be scheduled on each of these different nodes; the individual blocks are thus all processed locally, instead of needing to be transferred from one node to another.
- Of course, while log files can be processed in this piece-wise fashion, some file formats are not amenable to chunked processing. By writing a custom InputFormat, you can control how the file is broken up (or is not broken up) into splits.
- The InputFormat defines the list of tasks that make up the mapping phase; each task corresponds to a single input split. The tasks are then assigned to the nodes in the system based on where the input file chunks are physically resident. An individual node may have several dozen tasks assigned to it. The node will begin working on the tasks, attempting to perform as many in parallel as it can.

Record Reader

- The InputSplit has defined a slice of work, but does not describe how to access it. The*RecordReader* class actually loads the data from its source and converts it into (key, value) pairs suitable for reading by the Mapper.
- The RecordReader instance is defined by the InputFormat.
- The default InputFormat, *TextInputFormat*, provides a *LineRecordReader*, which treats each line of the input file as a new value. The key associated with each line is its byte offset in the file.
- The RecordReader is invoked repeatedly on the input until the entire InputSplit has been consumed. Each invocation of the RecordReader leads to another call to the map() method of the Mapper.

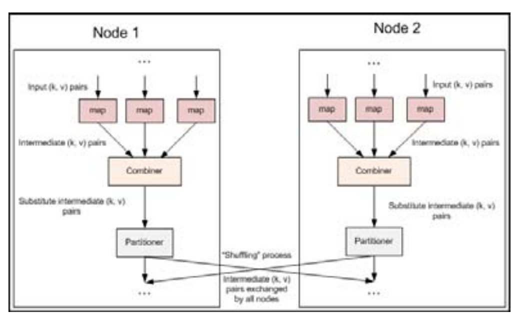
Mapper

- The Mapper performs the interesting **user-defined** work of the first phase of the Map/Reduce program.
- Given a key and a value, the map() method emits (key, value) pair(s) which are forwarded to the Reducers.
- A new instance of Mapper is instantiated in a separate Java process for each map task (InputSplit) that makes up part of the total job input.
- The individual mappers are intentionally not provided with a mechanism to communicate with one another in any way. This allows the reliability of each map task to be governed solely by the reliability of the local machine.
- The map() method receives two parameters in addition to the key and the value: output collector and reporter objects

Reduce

- A Reducer instance is created for each reduce task. This is an instance of **user-provided** code that performs the second important phase of job-specific work.
- For each key in the partition assigned to a Reducer, the Reducer's reduce() method is called once. This receives a key as well as an iterator over all the values associated with the key. The values associated with a key are returned by the iterator in an undefined order.
- The Reducer also receives as parameters *OutputCollector* and *Reporter* objects; they are used in the same manner as in the map() method.

Additional Map/Reduce functionality



Combiner

- The pipeline showed earlier omits a processing step which can be used for optimizing bandwidth usage by your Map/Reduce job.
- Called the *Combiner,* this pass runs after the Mapper and before the Reducer. Usage of the Combiner is optional.
- If this pass is suitable for your job, instances of the Combiner class are run on every node that has run map tasks.
- The Combiner will receive as input all data emitted by the Mapper instances on a given node.
- The output from the Combiner is then sent to the Reducers, instead of the output from the Mappers. The Combiner is **a** "mini-reduce" process which operates only on data generated by one machine**.**

An Application: Word Count

A simple Map/Reduce program can be written to determine count of words in a set of files. For example, if we had:

foo.txt: Sweet, this is the foo file, sweet

bar.txt: This is the bar file, very sweet

We would expect the output to be:

sweet	3
this	2
is	2
the	2
foo	1
bar	1
file	2
Very	1

Word Count example

- Input: Large number of text documents
- Task: Compute word count across all the document
- Solution
  - Mapper:
    - For every word in a document output (word (key), "1" (value))
  - Reducer:
    - Sum all occurrences of words and output (word, total_count)

Word Count Solution

```
//Pseudo-code for "word counting"
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 word_count = 0;
  for each v in values:
     word_count += ParseInt(v);
  Emit(key, AsString(word_count));
```

No types, just strings

Example: Word Count

```
def mapper(line):
  foreach word in line.split():
    output(word, 1)

def reducer(key, values):
  output(key, sum(values))
```

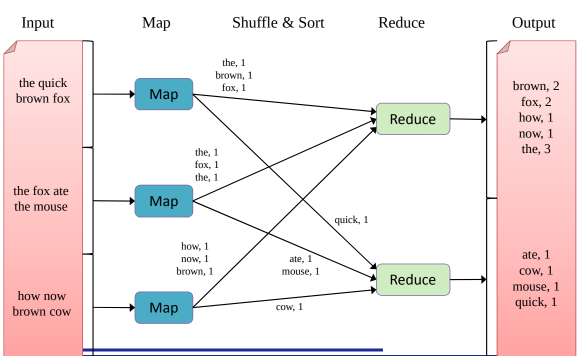
An Optimization: The Combiner

- A combiner is a local aggregation function for repeated keys produced by same map
- For associative ops. like sum, count, max
- Decreases size of intermediate data

- Example: local counting for Word Count:

```
def combiner(key, values):
  output(key, sum(values))
```

Word Count Execution



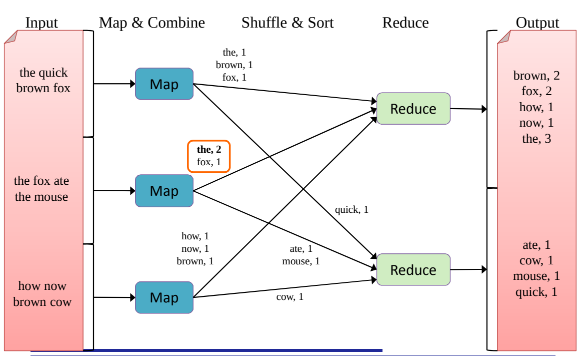
Combiner

- Word count is a prime example where a Combiner is useful. The Word Count program emits a (*word*, 1) pair for every instance of every word it sees. So if the same document contains the word "cat" 3 times, the pair ("cat", 1) is emitted three times; all of these are then sent to the Reducer.
- By using a Combiner, these can be condensed into a single ("cat", 3) pair to be sent to the Reducer. Now each node only sends a single value to the reducer for each word – drastically reducing the total bandwidth required for the shuffle process, and speeding up the job.

Combiner

- The best part is that we may not need to write any additional code to take advantage of this!
- If a reduce function is both *commutative* and *associative*, then it can be used as a Combiner as well.
- The Combiner should be an instance of the *Reducer* interface. If your Reducer itself cannot be used directly as a Combiner because of commutativity or associativity, you might still be able to write a third class to use as a Combiner for your job.
- Commutative means A o B is the same as B o A
- Associative means (A o B) o C is the same as A o (B o C)
- Count is both commutative and associative (as are +, max)
- - is not associative, concat is not commutative!

Word Count with Combiner



Word frequency example

- Used for decrypting, comparison of documents etc.
- Input: Large number of text documents
- Task: Compute word frequency (ratio) across all the document
  - Need to compute total count as well as individual count
- A naive solution with basic Map/Reduce model requires two Map/Reduces
  - MR1: count number of all words in these documents
    - Use combiners
  - MR2: count number of each word and divide it by the total count from MR1

Word frequency example

- Can we do better?
- Two nice features of Google's MapReduce implementation
  - Ordering guarantee of reduce key
  - Auxiliary functionality: EmitToAllReducers(k, v)
- A nice trick: To compute the total number of words in all documents
  - Every map task sends its total world count with key "" to ALL reducer splits
  - Key "" will be the first key processed by reducer
    - Sum of its values → total number of words!
- Requires only 1 map and 1 reduce instead of 2 each in a chain

Word frequency solution: Mapper with combiner

```
map(String key, String value):
// key: document name, value: document contents
  int word_count = 0;
  for each word w in value:
     EmitIntermediate(w, "1");
     word_count++;
  EmitIntermediateToAllReducers("", AsString(word_count));
combine(String key, Iterator values):
// Combiner for map output
// key: a word, values: a list of counts
   int partial_word_count = 0;
   for each v in values:
      partial_word_count += ParseInt(v);
      Emit(key, AsString(partial_word_count));
```


Word frequency solution: reducer

```
reduce(String key, Iterator values):
// Actual reducer, key: a word
// values: a list of counts
   if (is_first_key):
      assert("" == key); // sanity check
      total_word_count_ = 0;
      for each v in values:
         total_word_count_ += ParseInt(v)
   else
      assert("" != key); // sanity check
      int word_count = 0;
      for each v in values:
         word_count += ParseInt(v);
   Emit(key, AsString(word_count / total_word_count_));
```


Computing averages

- Average income in a city
- From census or other data
- Joins two inputs

SSTable 1: (SSN, {Personal Information})
123456:(John Smith;Sunnyvale, CA)
123457:(Jane Brown;Mountain View, CA)
123458:(Tom Little;Mountain View, CA)

SSTable 2: (SSN, {year, income})
123456:(2007,$70000),(2006,$65000),(2005,$6000),...
123457:(2007,$72000),(2006,$70000),(2005,$6000),...
123458:(2007,$80000),(2006,$85000),(2005,$7500),...

- Task: Compute average income in each city in 2007
- Note: **Both inputs sorted by SSN (it does not have to be)**

Computing averages

Mapper 1a
Input: SSN → personal info
Output: (ssn, city)

Mapper 1b:
input: SSN → annual incomes
output: (SSN, 2007 income)

reducer 1:
input: SSN, {city, 2007 income}
Output: (SSN, [City, 2007 income])

Mapper 2:
Input: SSN → [city, 2007 income]
Output: (City, 2007 income)

reducer 2:
Input: City → 2007 incomes
output: (City, AVG (2007 Incomes))

Average income in a joined solution

- The previous example showed a sorted input.
- But it does not have to be sorted.
- The input can be a single one as in our project case!

Mapper
Input: SSN → personal info and incomes
Output: (City, 2007 income)

Reducer
Input: City → {2007 incomes}
output: (City, AVG (2007 Incomes))

Approach

- What should be done in the mapper?
  - Specify key value output for each tuple
- Do we need a combiner? If so, what should it do
  - How does it transform the key, value produced by the mapper?
- Should we use the default partitioning?
  - Default partitioning is done on the key produced by the mapper. # partitions is based on the number of reducers.
- Should we provide a custom partitioning?
  - If so, what should it be
- What is received by the reducer? How do we process the key, list for each key identified in the mapper

Another Example

Suppose, we have the following input (can be any delimiter separated)

Id	Name	Age	Gender	Salary
1201	gopal	45	Male	50,000
1202	manisha	40	Female	50,000
1203	khalil	34	Male	30,000
1204	prasanth	30	Male	30,000
1205	kiran	20	Male	40,000
1206	laxmi	25	Female	35,000
1207	bhavya	20	Female	15,000
1208	reshma	19	Female	15,000
1209	kranthi	22	Male	22,000
1210	Satish	24	Male	25,000
1211	Krishna	25	Male	25,000
1212	Arshad	28	Male	20,000
1213	lavanya	18	Female	8,000

- **We have to compute highest salaried employee by gender in different age groups: below 20, between 21 to 30, and above 30**
- **Essentially compute Histograms for Male and Female separately**

Another Example

Suppose, we have the following input (can be any delimiter separated)

Id	Name	Age	Gender	Salary
1201	gopal	45	Male	50,000
1202	manisha	40	Female	50,000
1203	khalil	34	Male	30,000
1204	prasanth	30	Male	30,000
1205	kiran	20	Male	40,000
1206	laxmi	25	Female	35,000
1207	bhavya	20	Female	15,000
1208	reshma	19	Female	15,000
1209	kranthi	22	Male	22,000
1210	Satish	24	Male	25,000
1211	Krishna	25	Male	25,000
1212	Arshad	28	Male	20,000
1213	lavanya	18	Female	8,000

What key, value should the Mapper output?

- **We have to compute highest salaried employee by gender in different age groups: up to 20, between 21 to 30, and above 30**
- **Essentially compute Histograms for Male and Female separately**

Approach

- What should be done in the mapper?
  - Key: gender
  - Value: entire record (or needed portions)
- Do we need a combiner?
  - For this problem we **do not** need a combiner!
- Should we use the default partitioning?
  - Default partitioning will partition the key, namely, gender. That is not what we want
- Should we provide a custom partitioning?
  - Yes, but on what?
  - Read the key value pair. Access the age and create 3 partition based on the age groups given (if 5 age groups, need 5 partitions)

Approach

- With the partitioning, how many reducers are used?
  - 3
- Each partition is sent to a different reducer
- What do the key value pairs look like in each partition?
  - Partition 0:
    - Female <(bhavya, 20, female, 15000), {reshma, 19, female, 14000) (lavanya, 18, female, 8000)>
    - Male <(kiran, 20, male, 15000)>
  - Partition 1:
    - Female <(laxmi, 25, female, 35000)>
    - Male <(Satish, 24, male, 25000), …> total 5
  - Partition 2:

Approach

- What does the reducer do?
  - Reducer 0:
    - Female <(bhavya, 20, female, 15000), {reshma, 19, female, 14000) (lavanya, 18, female, 8000)>
    - Male <(kiran, 20, male, 40000)>
  - Compute the highest for each <key, list> coming to that reducer!
  - Reducer 1:
    - Female <(laxmi, 25, female, 35000)>
    - Male <(Satish, 24, male, 25000), …> total 5
  - Reducer 2:
  - You can see the code for this example at https://www.tutorialspoint.com/map_reduce/map_reduce_partitioner.htm

Approach

- Final output will be in 3 files each generated by a reducer.
  - Reducer 0:
    - Female <(bhavya, 20, female, 15000), {reshma, 19, female, 14000) (lavanya, 18, female, 8000)>
    - Male <(kiran, 20, male, 40000)>
- Output in Part-00000
  - Female 15000
  - Male 40000
- Output in Part-00001
  - Female 35000
  - Male 31000
- Output in Part-00002
  - Female 51000
  - Male 50000

Fault tolerance

- This fault tolerance underscores the need for program execution to be side-effect free.
- If Mappers and Reducers had individual identities and communicated with one another or the outside world, then restarting a task would require the other nodes to communicate with the new instances of the map and reduce tasks, and the re-executed tasks would need to reestablish their intermediate state (remember cascading rollbacks or aborts)
- This process is notoriously complicated and error-prone in the general case.
- Map/Reduce simplifies this problem drastically by eliminating task identities or the ability for task partitions to communicate with one another. An individual task sees only its own direct inputs and knows only its own outputs, to make this failure and restart process clean and dependable.

MR Fault tolerance and DBMS Recovery

- This fault tolerance underscores the need for program execution to be side-effect free.
- This requirement is also needed/used in the recovery of a DBMS using logs.
- If a transaction were to communicate with outside (i.e., outside of reading and writing from disks, and with others), recovery becomes very complicated and may not even be feasible.
- DBMS recovery aims at restoring the state of the DBMS to a consistent state so that transactions aborted can be re-executed from a consistent state
- It also requires that each transaction leaves the DBMS in a consistent state if it completes!
- ACID property (which is much stronger than what is used in MR) is guaranteed in a DBMS

Chaining jobs

- Not every problem can be solved with a Map/Reduce program, but fewer still are those which can be solved with a single Map/Reduce job. Many problems can be solved with Map/Reduce, by writing several Map/Reduce steps which run in series to accomplish a goal:
- Map1 -> Reduce1 -> Map2 -> Reduce2 -> Map3...
- You can easily chain jobs together in this fashion by writing multiple driver methods, one for each job. Call the first driver method, which uses JobClient.runJob() to run the job and wait for it to complete. When that job has completed, then call the next driver method, which creates a new JobConf object referring to different instances of *Mapper* and *Reducer*, etc.

Chaining examples

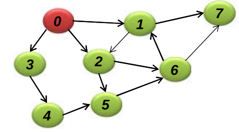
- Suppose you want to compute
  - Single Source Shortest Path
  - Page Rank
  - Graph substructures
- The above problems cannot be done in one iteration.
- This means several map/reduce pairs have to be chained to solve the problem!

Chaining jobs

- The first job in the chain should write its output to a path which is then used as the input path for the second job. This process can be repeated for as many jobs are necessary to arrive at a complete solution to the problem.
- Many problems which at first seem impossible in Map/Reduce can be accomplished by dividing one job into two or more.
- Hadoop provides another mechanism for managing batches of jobs with dependencies between jobs. Rather than submit a JobConf to the JobClient's runJob() or submitJob() methods, org.apache.hadoop.mapred.jobcontrol.Job objects can be created to represent each job;
- Dependencies can be accommodated

Iteration 1

Mapper outputs



Key	Value
1	1
2	1
3	1

Key	Value
1	[2,7]

Key	Value
2	[5,6]

Key	Value
3	[4]

Key	Value
4	[5]

Key	Value
5	[6]

Key	Value
6	[1,7]

Vertex 0 does not emit its distance or adj list as it is the start node

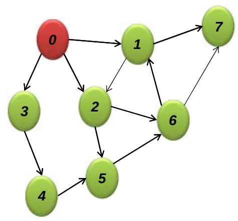
Vertex 7 also does not emit anything as its adjacency list is empty

Node Id	Adj List	Distance
0	[1,2,3]	0
1	[2,7]	Null
2	[5,6]	Null
3	[4]	Null
4	[5]	Null
5	[6]	Null
6	[1,7]	Null
7	[]	Null

P1 To Mapper

P2 To Mapper

Graph Input



Node Id	Adj List	Distance
0	[1,2,3]	0
1	[2,7]	Null
2	[5,6]	Null
3	[4]	Null
4	[5]	Null
5	[6]	Null
6	[1,7]	Null
7	[]	Null

Distance indicates distance of the node from the source
Initially only the self distance is known so all other distance is null

Iteration 1 – Combiner in action

Mapper Outputs

Key	Value
1	[2,7] , 1

Key	Value
2	[5,6] , 1

Key	Value
3	[4] , 1

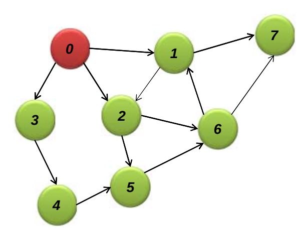
Key	Value
4	[5]

Key	Value
5	[6]

Key	Value
6	[1,7]

Combiner combines same keys in a mapper

It is an in-mapper reducer



Reducer Inputs

Iteration 1 – Reducer in action

Reducer has key as vertex id followed by values.

If number of values < 2 →
if value has "["
Not reached
Else: reached with distance

If number of values >= 2 values contain the distances from the source to this node and the adjacency list.

Emit the minimum of the distance values and the adjacency list

Reducer Input

Key	Value
1	[2,7] , 1

Key	Value
2	[5,6] , 1

Key	Value
3	[4] , 1

Key	Value
4	[5]

Key	Value
5	[6]

Key	Value
6	[1,7]

Reducer Output

Key	Value
1	[2,7]
1	1

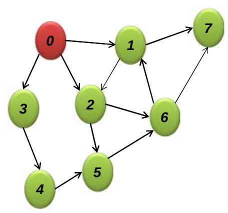
Key	Value
2	[5,6]
2	1

Key	Value
3	[4]
3	1

Key	Value
4	[5]

Key	Value
5	[6]

Key	Value
6	[1,7]



Iteration i (i = 2) - Combiner in action

Individual map task output

Key	Value
1	[2,7]
2	2
7	2
1	1

Key	Value
4	[5]

Key	Value
5	[6]

Key	Value
4	2
3	[4]
3	1

Key	Value
6	[1,7]

Key	Value
5	2
6	2
2	[5,6]
2	1

Combiner Output

Key	Value
1	[2,7] , 1

Key	Value
6	2

Key	Value
2	2, [5,6] , 1

Key	Value
5	2

Key	Value
3	[4] , 1

Key	Value
4	[5]

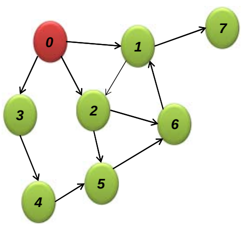
Key	Value
7	2

Key	Value
5	[6]

Key	Value
4	2

Key	Value
6	[1,7]

Reducer Input



Iteration i (i = 2) – Mapper in action

Mapper Input

Key	Value
1	[2,7]
1	1

M1

Key	Value
2	[5,6]
2	1

M1

Key	Value
3	[4]
3	1

M1

Key	Value
4	[5]

M2

Key	Value
5	[6]

M2

Key	Value
6	[1,7]

M2

Mapper Output

Key	Value
1	[2,7]
2	2
7	2
1	1

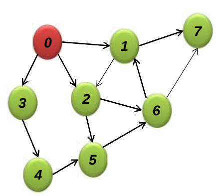
Key	Value
4	2
3	[4]
3	1

Key	Value
5	2
6	2
2	[5,6]
2	1

Key	Value
4	[5]

Key	Value
5	[6]

Key	Value
6	[1,7]



Iteration i (i = 2) - Reducer in action

Reducer Input

Key	Value
1	[2,7] , 1

Key	Value
2	2, [5,6] , 1

Key	Value
3	[4] , 1

Key	Value
4	2, [5]

Key	Value
5	2, [6]

Key	Value
6	2, [1,7]

Key	Value
7	2

Reducer Output

Key	Value
1	[2,7]
1	1

Key	Value
4	2
4	[5]

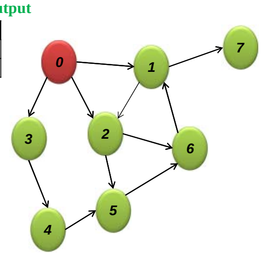
Key	Value
2	[5,6]
2	1

Key	Value
5	2
5	[6]

Key	Value
3	[4]
3	1

Key	Value
6	2
6	[1,7]

Key	Value
7	2

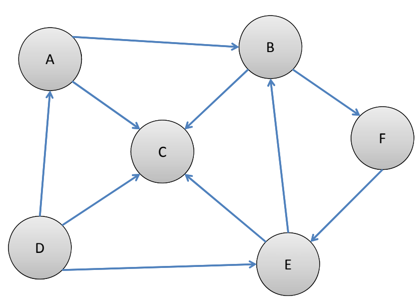


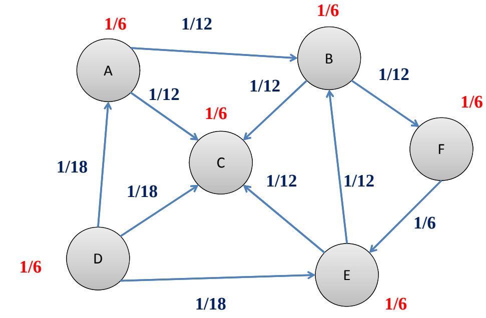
Continue iterating now for i > 2

Takeaways

- Convergence
  - The algorithm will converge when the distance of nodes from source do not change across iterations
  - Need to check convergence criteria periodically to stop iterations (additional processing)
  - The algorithm keeps the distance and not the path (extra bookkeeping for storing the path)

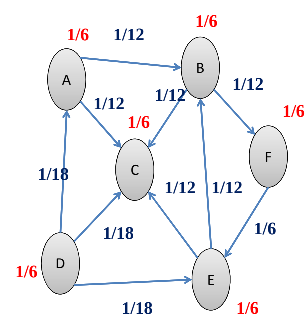
Page Rank





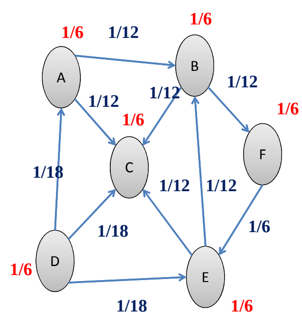
1/6	1/6	1/6	1/6	1/6	1/6

	A	B	C	D	E	F
A	0	1	1	0	0	0
B	0	0	1	0	0	1
C	0	0	0	0	0	0
D	1	0	1	0	1	0
E	0	1	1	0	0	0
F	0	0	0	0	1	0



1/6	1/6	1/6	1/6	1/6	1/6

	A	B	C	D	E	F
A	0	0.5	0.5	0	0	0
B	0	0	0.5	0	0	0.5
C	0	0	0	0	0	0
D	0.33	0	0.33	0	0.33	0
E	0	0.5	0.5	0	0	0
F	0	0	0	0	1	0



Summary

- Additional features such as pipes and streaming are available in Hadoop.
- If you are familiar with C++ or Java, it is not very difficult to understand the basic concept and use it
- Of course, if you want to use advanced features, you need to learn them
- Much easier than using a DBMS for some jobs where the data is in free format; will discuss more of this later!

Page Rank Trivia

- Mapper emits weights across all out links
- Reducer aggregates weights from all in links
- Reducer outputs go into another mapper
- We keep on iterating until values do not change across iterations
- Power law method converges in ~30 iterations

Questions !