CS 2510 – COMPUTER OPERATING SYSTEMS

Cloud Computing

MAPREDUCE

Dr. Taieb Znati
Computer Science Department
University of Pittsburgh

MAPREDUCE Programming Model

- Scaling Data Intensive Application
- MapReduce Framework
  - Map Reduce Data Flow
- MapReduce Execution Model
  - MapReduce Jobtrackers and Tasktrackers
  - Input Data – Splits
  - Scheduling and Synchronization
  - Speculative Execution
  - Partitioners and Combiners

Scaling Data Intensive Application – Example

I can not do everything, but still I can do something; and because I cannot do everything, I will not refuse to do something I can do

Word	Count
And	1
Because	1
But	1
Can	4
Do	4
Everything	2
I	5
Not	3
Refuse	1
Something	2
Still	1
To	1
Will	1

WordCount Program – I

- Define WordCount as Multiset;
- For Each Document in DocumentSet {
-     T = tokenize(document);
-     For Each Token in T {
-     WordCount[token]++;
-    }
- }
- Display(WordCount);

Program Does NOT Scale for Large Number of Documents

WordCount Program – II

- A two-phased program can be used to speed up execution by distributing the work over several machines and combining the outcome from each machine into the final word count
- Phase I – Document Processing
  - Each machine will process a fraction of the document set
- Phase II – Count Aggregation
  - Partial word counts from individual machines are combined into the final word count

WordCount Program – II

Phase I

- Define WordCount as Multiset;
- For Each Document in DocumentSubset {
  - T = tokenize(document);
  - For Each Token in T {
    - WordCount[token]++;
  - }
- } SendToSecondPhase(wordCount);

Phase II

- Define TotalWordCount as Multiset;
- For each WordCount Received from **Phase I** {
  - MultisetAdd (TotalWordCount, WordCount);
  - }

WordCount Program II – Limitations

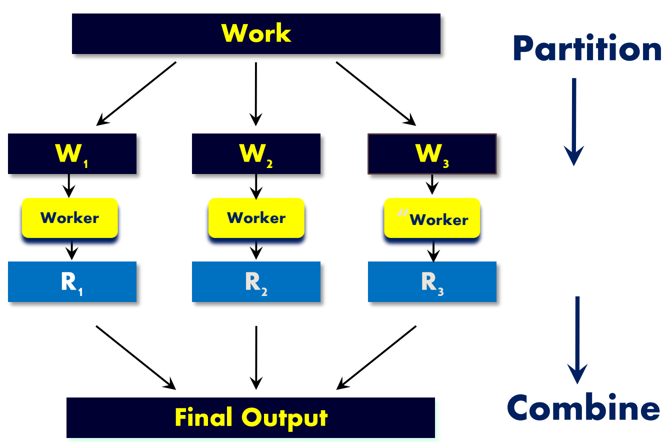
- The program does not take into consideration the location of the documents
  - Storage server can become a bottleneck, if enough bandwidth is not available
    - Distribution of documents across multiple machines removes the central server bottleneck
- Storing WordCount and TotalWordCount in the memory is a flaw
  - When processing large document sets, the number of unique words can exceed the RAM capacity
- In Phase II, the aggregation machine becomes the bottleneck

WordCount Program – Solution

- The aggregation phase must execute in a distributed fashion on a cluster of machines that can run independently
- To achieve this, functionalities must be added
  - Store files over a cluster of processing machines.
  - Design a disk-based hash table permitting processing without being limited by RAM capacity.
  - Partition intermediate data across multiple machines
  - Shuffle the partitions to the appropriate machines

How to Scale Up?

Divide and Conquer



Parallelization Challenges

- How do we assign work units to workers?
- What if we have more work units than workers?
- What if workers need to share partial results?
- How do we aggregate partial results?
- How do we know all the workers have finished?
- What if workers fail?

Parallelization Challenges

- Parallelization problems arise from in several ways
  - Communication between workers, **asynchronously**
    - Workers need to exchange information about their states
  - Concurrent access to shared resources, while preserving "state" consistency
    - Workers need to manipulate data, concurrently
- Cooperation requires synchronization and interprocess communication mechanisms

Distributed Workers Coordination

- Coordinating a large number of workers in a distributed environment is challenging
  - The order in which workers run may be unknown
  - The order in which workers interrupt each other may be unknown
  - The order in which workers access shared data may be unknown
  - Failures further compound the problem!

Classic Models

- Computational Models
  - Master and Slaves
  - Producers and Consumers
  - Readers and Writers
- IPC Models
  - Shared Memory – Threads
  - Message Passing

- To ensure correct execution several mechanisms are needed
  - Semaphores (lock, unlock), Conditional variables (wait, notify, broadcast), Barriers, …
  - Address deadlock, livelock, race conditions, …
  - Makes it difficult to debug parallel execution on clusters of distribute processors

MapReduce – Data-Intensive Programming Model

- MapReduce is a programming model for processing large sets
- Users specify the computation in terms of a **map()** and a **reduce()** function,
  - Underlying runtime system **automatically** parallelizes the computation across large-scale clusters of machines, and
  - Underlying system also handles machine **failures**, efficient communications, and performance issues.
- MapReduce is inspired by the map() and fold() functions commonly used in functional programming

Typical Large-Data Problem

- At a high-level of abstraction, MapReduce codifies a generic "recipe" for processing large data set
  - Iterate over a large number of records
  - Extract something of interest from each

    } **Map**
  - Shuffle and sort intermediate results
  - Aggregate intermediate results
  - Generate final output

    } **Reduce**

Basic Tenet of MapReduce is Enabling a Functional Abstraction for the Map() and Reduce() operations

MapReduce

Functional Programming Paradigm

Imperative Languages and Functional Languages

- The design of the imperative languages is based directly on the von Neumann architecture
  - Efficiency is the primary concern, rather than the suitability of the language for software development
- The design of the functional languages is based on mathematical functions
  - A solid theoretical basis that is also closer to the user, but relatively unconcerned with the architecture of the machines on which programs will run

Fundamentals of Functional Programming Languages

- The basic process of computation is fundamentally different in a FPL than in an imperative language
  - In an imperative language, operations are done and the results are stored in variables for later use
  - Management of variables is a constant concern and source of complexity for imperative programming
- FPL takes a mathematical approach to the concept of a variable
  - Variables are bound to values, not memory locations
  - A variable's value cannot change, which **eliminates assignment** as a possible operation

Characteristics of Pure FPLs

- Pure FP languages tend to
  - Have no side-effects
  - Have no assignment statements
  - Often have no variables!
  - Be built on a small, concise framework
  - Have a simple, uniform syntax
  - Be implemented via interpreters rather than compilers
  - Be mathematically easier to handle

Importance of FP

- FPLs encourage thinking at higher levels of abstraction
  - **It enables programmers to work in units larger than statements of conventional languages**
- FPLs provide a paradigm for parallel computing
  - **Absence of assignment** provide **basis** for **independence of evaluation order**
  - Ability to operate on entire data structures

FPL and IPL – Example

- Summing the integers 1 to 10 in IPL – The computation method is variable assignment

```
total = 0;
for (i = 1; i ≤ 10; ++i)
   total = total+i;
```

- Summing the integers 1 to 10 in FPL – The computation method is function application

```
sum [1..10]
```

22

Lambda Calculus

- The lambda calculus is a formal mathematical system to investigate functions, function application and recursion.
- A lambda expression specifies the parameter(s) and the mapping of a function in the following form
  - λ x . x * x * x – for the function cube (x) = x * x * x
- Lambda expressions describe nameless functions
- Lambda expressions are applied to parameter(s) by placing the parameter(s) after the expression
  - (λ x . x * x * x) 3 => 3*3*3 => 27
  - (λ x,y . (x-y)*(y-x)) (3,5) => (3-5)*(5-3) => -4

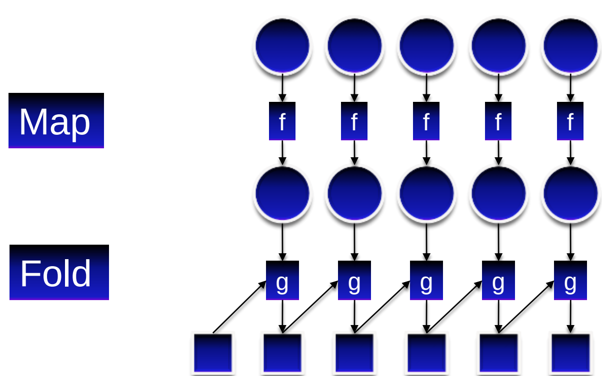
MapReduce

Functional Programming

FPL Map and Fold

- "map" and "fold" – FPL higher-order functions
- (map f list1 [list2 list3 …])
  - (map square (1 2 3 4)) → (1 4 9 16)
- (fold f list […])
  - (fold + (1 4 9 16)) → 30
- (fold + (map square (map – list1 list2))))

Roots in Functional Programming



What is MapReduce?

- Programming model for expressing distributed computations at a massive scale
- Execution framework for organizing and performing such computations
- Open-source implementation called Hadoop

Mappers And Reducers

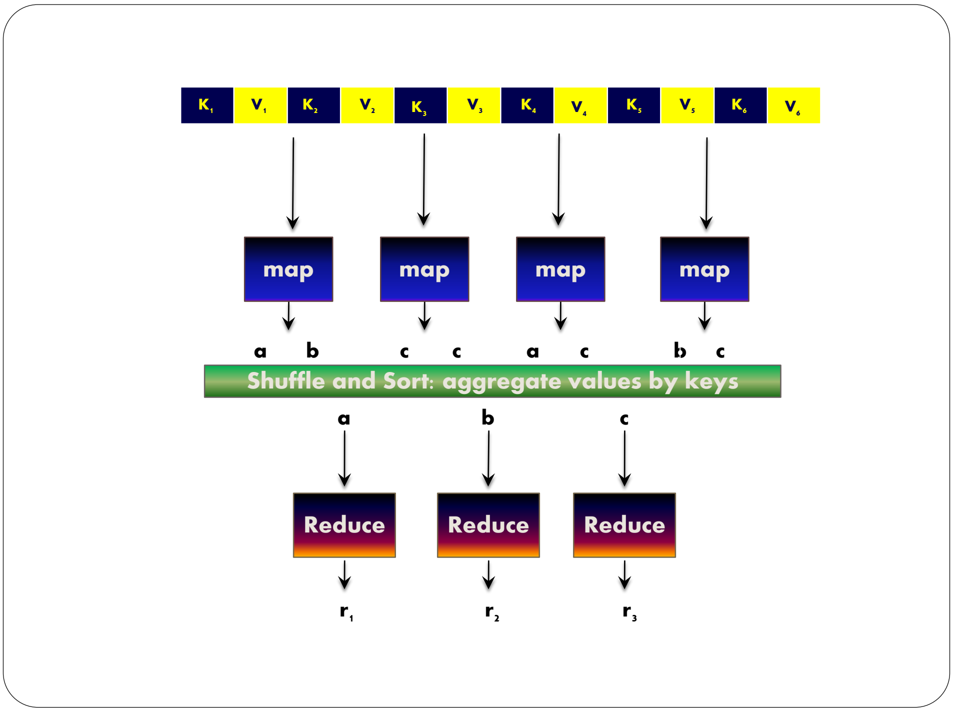
- A mapper is a function that takes as input one ordered (key; value) pair of binary strings.
  - As output the mapper produces a finite multiset of new (key, value) pairs.
  - Mappers operates on **ONE** (key; value) pair at a time
- A reducer is a function that takes as input a binary string k which is the key, and a sequence of values v_1, v_2, …, v_n, which are also binary strings.
  - As output, the reducer produces a multiset of pairs of binary strings $(k,v_{k,1})$, $(k,v_{k,2})$, $(k,v_{k,3})$, … $(k,v_{k,n})$
- Key in output tuples is identical to the key in input tuple.
  - **Consequence** – Mappers can manipulate keys arbitrarily, but Reducers cannot change the keys at all

MapReduce Framework

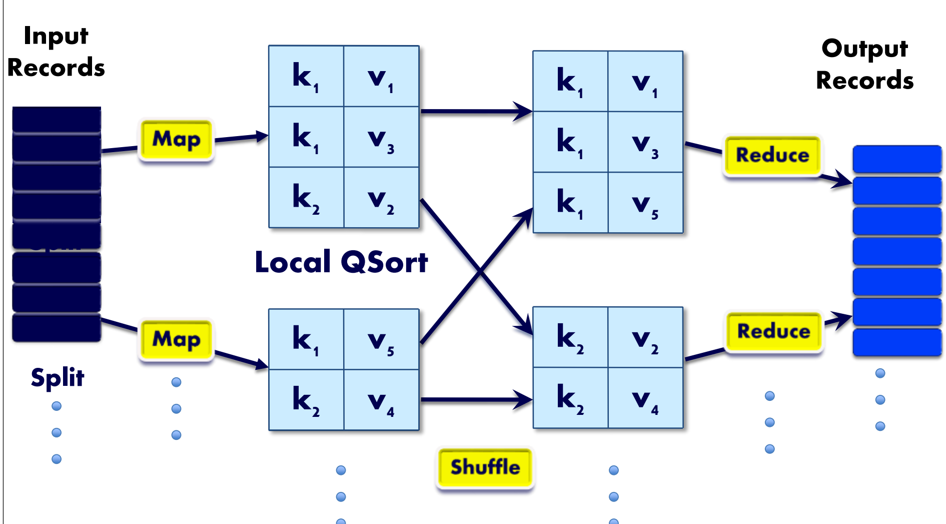
- Programmers specify two functions:
  map (k, v) → <k', v'>*
  reduce (k', v') → <k', v'>*
  - All values, v', with the same key are sent to the same reducer
- The **execution framework** supports a computational runtime environment to handle all issues related to coordinating the parallel execution of a data-intensive computation in a large-scale environment
  - Breaking up the problem into smaller tasks, coordinating workers executions, aggregating intermediate results, dealing with failures and softeare errors, …

MapReduce "Runtime" Basic Functions

- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles "data distribution"
  - Moves processes to data, not data to processes
- Handles synchronization among workers
  - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults, dynamically
  - Detects worker failures and restarts



MapReduce Data Flow



MapReduce

- Programmers specify two functions:
  map (k, v) → <k', v'>*
  reduce (k', v') → <k', v'>*
  - All values with the same key are reduced together
- The execution framework handles everything else!
- **Not quite …** Usually, programmers also specify:
  partition (k', number of partitions) → partition for k'
  - Often a simple hash of the key, e.g., hash(k') mod N
  - Divides up key space for parallel reduce operations

  combine (k', v') → <k', v'>*
  - Mini-reducers that run in memory after the map phase
  - Used as an optimization to reduce network traffic

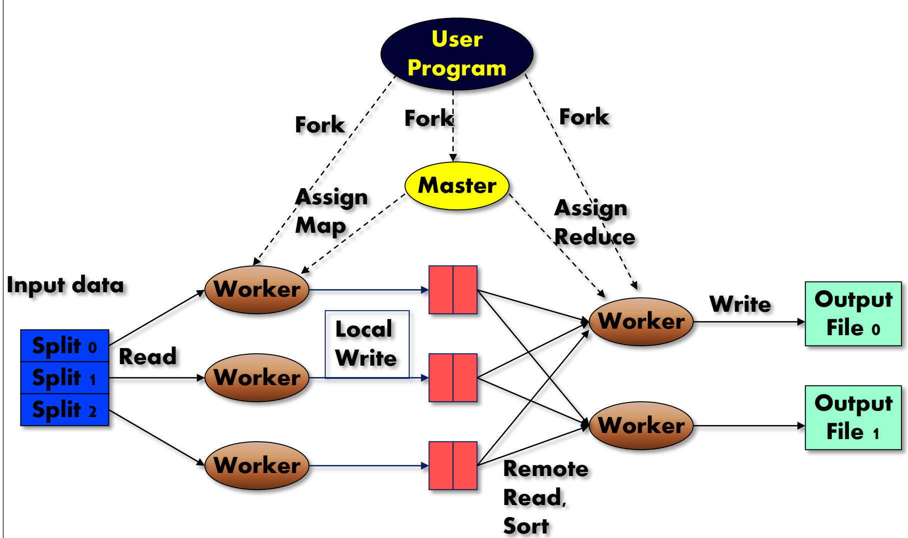
MapReduce Design Issues

- Barrier between map and reduce phases
  - To enhance performance the process of copying intermediate data can start early
- Keys arrive at each reducer in sorted order
  - No enforced ordering *across* reducers

MapReduce Implementations

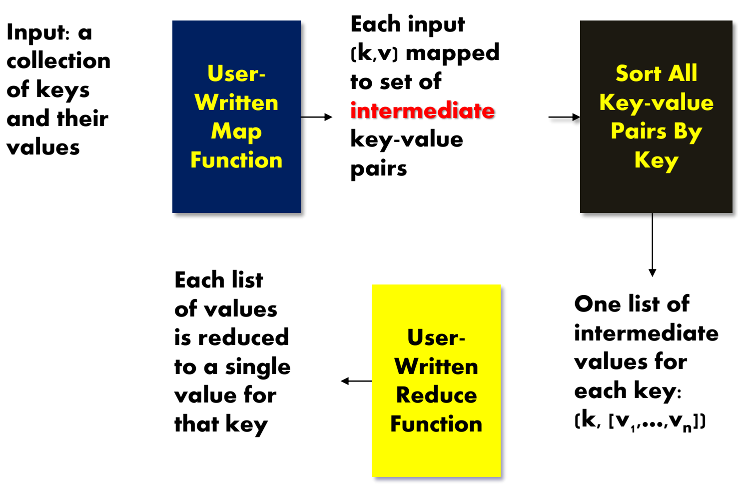
- Google has a proprietary implementation in C++
  - Bindings in Java, Python
- Hadoop is an open-source implementation in Java
  - Development led by Yahoo, used in production
  - Now an Apache project
  - Rapidly expanding software ecosystem
- Lots of custom research implementations
  - For GPUs, cell processors, etc.

Distributed Execution Overview



"Hello World": Word Count

Map-Reduce Framework for Word Count



WordCount PseudoCode

Map

- map(String filename, String document) { List<String> T = tokenize(document);
- for each token in T { emit ((String)token, (Integer) 1); }
- }

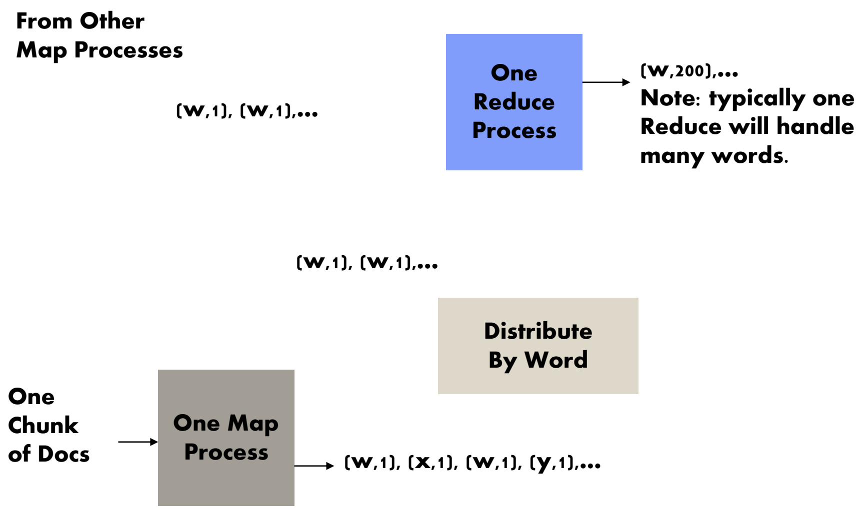
Reduce

- reduce(String token, List<Integer> values) { Integer sum = 0;
- for each value in values {
- sum = sum + value;
- }
- emit ((String)token, (Integer) sum);
- }

Implementation of WordCount()

- A program forks a *master* process and many *worker* processes.
- Input is partitioned into some number of *splits*.
- Worker processes are assigned either to perform Map on a split or Reduce for some set of intermediate keys.

Execution of WordCount



Responsibility of the Master

1. Assign Map and Reduce tasks to Workers.
2. Check that no Worker has died (because its processor failed).
3. Communicate results of Map to the Reduce tasks.

Communication from Map to Reduce

- Select a number *R* of reduce tasks.
- Divide the intermediate keys into *R* groups,
  - Use an efficient hashing function
- Each Map task creates, at its own processor, *R* files of intermediate key-value pairs, one for each Reduce task.

Map Reduce Execution Framework

Dr. Taieb Znati
Computer Science Department
University of Pittsburgh

MAPREDUCE Execution Framework

- MapReduce Execution Model
  - MapReduce Jobtrackers and Tasktrackers
  - Input Data – Splits
  - MapReduce Execution Issues
    - Scheduling and Synchronization
    - Speculative Execution
    - Partitioners and Combiners

MapReduce Data Flow

- A MapReduce job is a unit of work to be performed
  - Job consists of the **MapReduce Program**, the **Input data** and the **Configuration Information**
- The MapReduce job is divided it into two types of tasks – map tasks and reduce tasks
  - It is not uncommon for MapReduce jobs to have thousands of individual tasks to be assigned to cluster nodes
- The Input data is divided into fixed-size pieces called **splits**
  - One map task is created for each split
  - The user-defined map function is run on each split
- Configuration information indicates where the input lies, and the output is stored

Lifecycle of a MapReduce Job

File Edit Options Buffers Tools Java Help

```java
public class WordCount {

  public static class Map extends MapReduceBase implements
                Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, OutputCollector<Text, IntWritable>
                output, Reporter reporter) throws IOException {
      String line = value.toString();
      StringTokenizer tokenizer = new StringTokenizer(line);
      while (tokenizer.hasMoreTokens()) {
        word.set(tokenizer.nextToken());
        output.collect(word, one);
  }}}

  public static class Reduce extends MapReduceBase implements
                Reducer<Text, IntWritable, Text, IntWritable> {
    public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text,
                IntWritable> output, Reporter reporter) throws IOException {
      int sum = 0;
      while (values.hasNext()) { sum += values.next().get(); }
      output.collect(key, new IntWritable(sum));
  }}

  public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(Map.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);
    conf.setInputFormat(TextInputFormat.class);
    conf.setOutputFormat(TextOutputFormat.class);
    FileInputFormat.setInputPaths(conf, new Path(args[0]));
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));

    JobClient.runJob(conf);
  }}
```

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

Map function

Reduce function

Job Execution

Jobtrackers and Takstrackers

- Two types of nodes control the job execution process — Jobtraccker and Tasktrackers
- A jobtracker coordinates all the jobs run on the system by scheduling tasks to run on Tasktrackers
- Tasktrackers run tasks and send progress reports to the jobtracker
- Jobtracker keeps a record of the overall progress of each job
  - If a task fails, the jobtracker can reschedule it on a different Tasktracker

Scheduling and Synchronization

Jobtracker – Scheduling and Coordination

- In large jobs, the total number of tasks may exceed the number of tasks that can be run on the cluster concurrently,
  - The Jobtracker must maintain a task queue and assign nodes to waiting tasks as the nodes become available.
- Another aspect of Jobtracker's responsibiliies involves coordination among tasks belonging to different jobs
  - Jobs from different users, for example
- Designing a large-scale, shared resource to support several users simultaneously in a predictable, transparent and policy-driven fashion is challenging!

MapReduce Stragglers

- The speed of a MapReduce job is sensitive to the stragglers' performance – tasks that take an usually long time to complete
  - The map phase of a job is only as fast as the slowest map task.
  - The running time of the slowest reduce task determines the completion time of a job
- Stragglers may result from unreliable hardware
  - A machine recovering from frequent hardware errors may become significantly slower
- The barrier between the map and reduce tasks further compounds the problem

MapReduce – Speculative Execution

- Speculative execution is an optimization technique to improve job running times, in the presence of stragglers
  - Base on speculative execution, an identical copy of the same task is executed on a different machine,
  - The result of the task that finishes first is used
- Google has reported that speculative execution can achieve 44% performance improvement

MapReduce – Speculative Execution

- Both map and reduce tasks can be speculatively executed, but the technique is better suited to map tasks than reduce tasks
  - This is due to the fact that each copy of the reduce task needs to pull data over the network.
- Speculative execution cannot adequately address cases where stragglers are caused by a skew in the distribution of values associated with intermediate keys
  - In these cases, tasks responsible for processing the most frequent elements run much longer than the typical task
    - More efficient local aggregation may be required

MapReduce – Synchronization

- In MapReduce, synchronization is needed when mappers and reduces exchange intermediate output and state information
  - Intermediate key-value pairs must be grouped by key, which requires the execution a distributed sort process involving all the nodes that executed map tasks and all the nodes that will execute reduce tasks
    - The “shuffle and sort” process involves copying intermediate data over the network

MapReduce – Synchronization

- A MapReduce job with **M** mappers and **R** reducers may involves up to **M•R** distinct copy operations
  - Each mapper intermediate output goes to every reducer
- No reducer can start until all the mappers have finished emitting key-value pairs and all intermediate key-value pairs have been shuffled and sorted
  - Necessary to guarantee that all values associated with the same key have been gathered.
  - This is an important departure from functional programming, where aggregation can begin as soon as values are available.
- For improvement start copying intermediate key-value pairs over the network to the nodes running the reducers as soon as each mapper finishes

Data Locality Optimization

Input Division – Split Size

- Fine-grained splits increase parallelism and improves fault-tolerance
  - Small splits reduce the processing time of each split and allows faster machines to process proportionally more splits over the course of the job than slower machines
  - Load-balancing can be achieved more efficiently with small splits
  - The impact of failure, when combined with load-balancing, can be reduced significantly with fine-grained splits
- Too small splits increases the overhead of managing splits
  - Map task creation dominates the total job execution time.

Data Locality – Input Data

- Data locality Optimization
  - The map task should be run on a node where the input data resides
  - The **optimal split size** is the same as the largest size of input that can be guaranteed to be stored on a **single** node.
    - If the split is larger than what one node can store, data transfer on across the network to the node running the map task is required
      - May result in significant communication overhead, and reduces efficiency

Data Locality – Map Output

- Output produced by map tasks should be stored locally, **NOT** at a **distributed storage**
  - Map output is intermediate – Processed by reduce tasks to produce the final output
    - Map output is no needed upon completion of the job
- Map output should NOT be replicated to overcome failure
  - It is more efficient to restart the map task upon failure than replicating the output produced by map tasks

Data Locality – Reduce Tasks

- Reduce tasks cannot typically take advantage of data locality
  - Input to a single reduce task is normally the output from all mappers.
- The sorted map outputs have to be transferred across the network to the node where the reduce task is running
  - The outputs are merged and then passed to the user-defined reduce function

Reduce Task Output – Replication

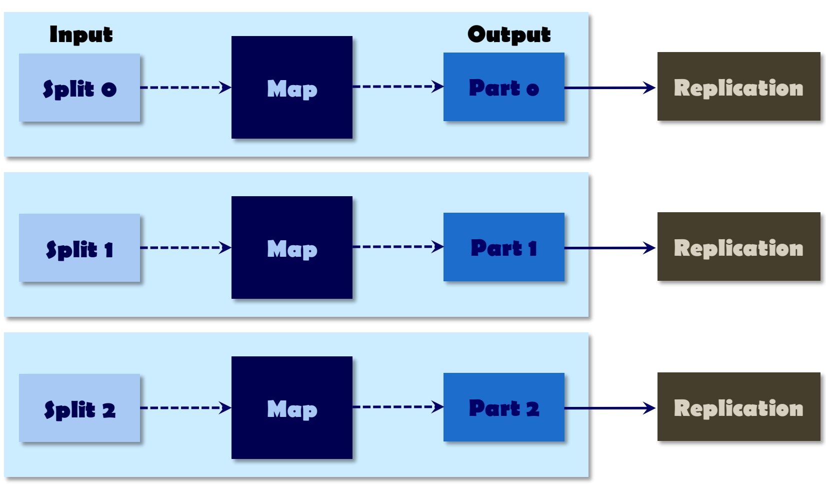
- Multiple replicas of the reduce output are normally stored reliably,
  - First replica is stored on the local node,
  - Other replicas being stored on off-rack nodes.
- To increase efficiency, the writing of the reduce output must reduce the amount of network bandwidth consumed
  - The replication process must be streamlined and efficient

Partitioners and Combiners

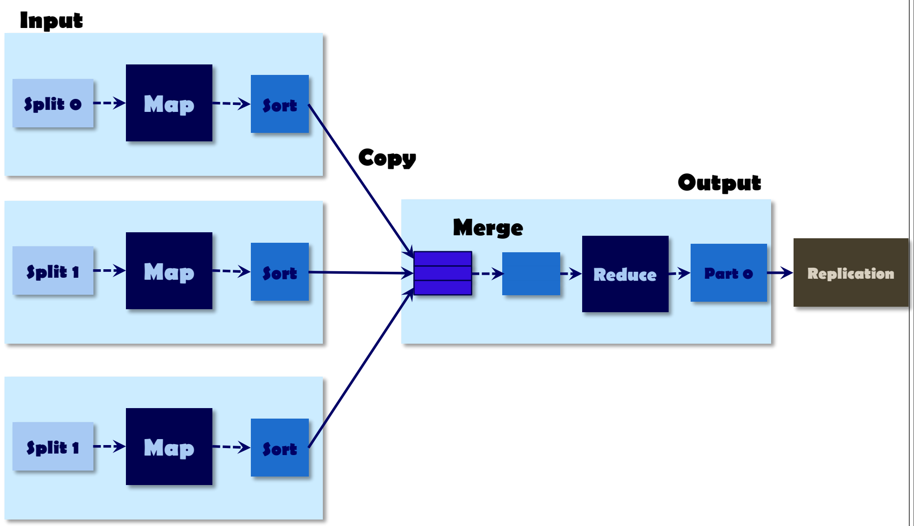
Number of Reduce Tasks

- The number of map reducers is typically specified independently
  - It is not only governed by the size of the input, but also the type of the application
- MapReduce data flow can be specified in different ways
  - No reduce tasks
  - Single reduce task
  - Multiple reduce tasks

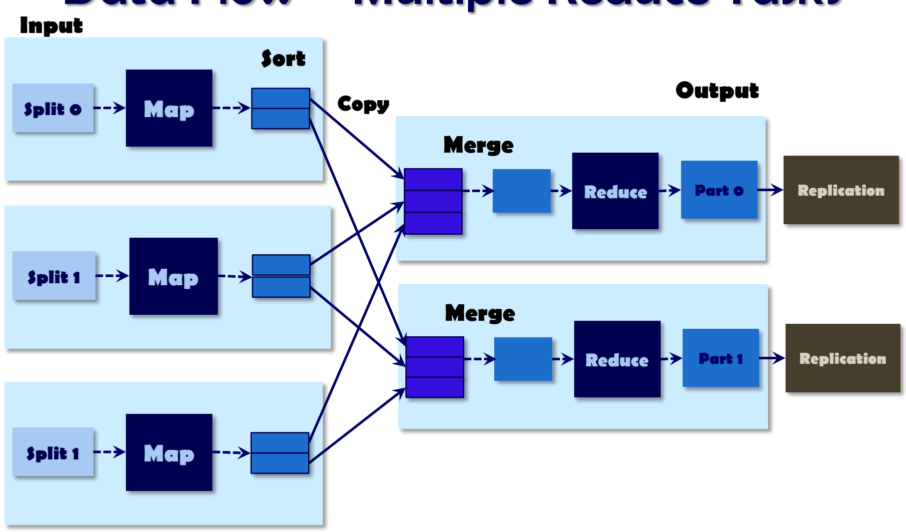
Data Flow – No Reduce Tasks



Data Flow – Single Reduce Task



Data Flow – Multiple Reduce Tasks



Data Flow – Multiple Reduce Tasks

- Map tasks partition their output, each creating one partition for each reduce task
  - There can be many keys and their associate values in each partition
    - The records for every key are all in a single partition
  - The partition can be controlled by user-defined partition function
    - The use of a hash function typically works well
- The "Shuffle", data flow between map and reduce, is a complicated process whose tuning can have a big impact on the job execution

MapReduce – Combiners

- Combiner functions can be used to minimize the data transferred between map and reduce tasks
  - Combiners are particularly useful when MapReduce jobs are limited by the bandwidth available on the cluster
    - *Combiners are user-specified functions*
- Combiner functions run on the map output
  - The combiner's output is then fed to the reduce function
- Since it is an optimization function, there is no guarantee how many times combiners are called for a particular map output record, if at all
  - Calling the combiner zero, one, or many times should produce the same output from the reducer.

Combiner Example – Max Temperature

- Assume that the mappers produce the following output
  - Mapper 1 – (1950, 0), (1950, 20), (1950, 10)
  - Mapper 2 – (1950, 25), (1950, 15)
- Reduce function is called with a list of all the values
  - (1950, [0, 20, 10, 25, 15]) with output (1950, 25)
- Using a combiner for each map output results in:
  - Combiner 1 – (1950, 20)
  - Combiner 2 – (1950, 25)
- Reduce function is called with (1950, [20,25]) with output (1950, 25)

Combiner Property

- The combiner function calls can be expressed as follows:
  - Max(0, 20, 10, 25, 15) = Max(Max(0, 20, 10), Max(25, 15)) = Max(20, 25)=25
  - Max() is commonly referred to as **distributive**
- Not all function exhibit distributive property
  - Mean(0, 20, 10, 25, 15) = (0+20+10+25+15)/5=14
  - Mean(Mean(0, 20, 10), Mean(25, 15)) = Mean (10, 20) = 15
- Combiners do not replace reducers
  - Reducers are still needed to process recorders with the same key from different maps

Conclusion – Part I

- MapReduce Execution Framework
  - MapReduce Jobtrackers and Tasktrackers
  - Input Data – Splits
  - MapReduce Execution Issues
    - Scheduling and Synchronization
    - Speculative Execution
    - Partitioners and Combiners

Conclusion – Part II

- Scaling Data Intensive Application
- MapReduce Framework
  - MapReduce Overview
  - Map Reduce Data Flow
    - Map Function
    - Partition Function
    - Compare Function
    - Reduce Function

Reference

- Data-Intensive Text Processing with MapReduce, Jimmy Lin and Chris Dyer.
- MapReduce: Simplified Data Processing on Large Clusters, Jeffrey Dean and Sanjay Ghemawat,
- The Google File System, Sanjay Ghemawat, Howard Gobioff, and Shun-TakLeung,