COMP9313: Big Data Management

Lecturer: Xin Cao

Course web site: http://www.cse.unsw.edu.au/~cs9313/

Assignment Project Exam Help

Ch https://eduassistpro.github.io/ **ce I**

Add WeChat edu_assist_pro

What is MapReduce

- Origin from Google, [OSDI'04]
  - MapReduce: Simplified Data Processing on Large Clusters
  - Jeffrey Dean and Sanjay Ghemawat
- Programming model for parallel data processing
- Hadoop can run MapReduce programs written in various languages: e.g. Java, Ruby,
- For large-scale
  - Exploits large set of commodity
  - Executes process in distributed
  - Offers high availability

Motivation for MapReduce

- A Google server room:

https://www.youtube.com/watch?t=3&v=avP5d16wEp0

Motivation for MapReduce

- Typical big data problem challenges:
  - How do we break up a large problem into smaller tasks that can be executed in parallel?
  - How do we assign tasks to workers distributed across a potentially large number of machines?

  - How do we ensure that the workers get the data they need?
  - How do we ng the different workers?

  - How do we share partial results er that is needed by another?

  - How do we accomplish all of the above in the face of software errors and hardware faults?

Motivation for MapReduce

- There was need for an abstraction that hides many system-level details from the programmer.

- MapReduce addresses this challenge by providing a simple abstraction for the developer, transparently handling most of the details behind the scenes in a ***scalable***, ***robust***, and ***efficient*** manner.

- MapReduce separates the ***what*** fro

Jeffrey (Jeff) Dean

- He is currently a Google Senior Fellow in the Systems and Infrastructure Group
- Designed MapReduce, BigTable, etc.
- One of the most genius engineer, programmer, computer scientist…
- Google “Who is Jeff Dean” and “Jeff Dean facts”

Jeff Dean Facts

- Kenton Varda created "Jeff Dean Facts" as a Google-internal April Fool's joke in 2007.
  - *The speed of light in a vacuum used to be about 35 mph. Then Jeff Dean spent a weekend optimizing physics*

  - *Jeff Dean once bit a spider, the spider got super powers and C readability*

  - *Jeff Dean puts his pants on one but if he had more than two legs, you would see th h is actually O(log n)*

  - *Compilers don't warn Jeff Dean. Jeff Dean warns compilers*
  - *The rate at which Jeff Dean produces code jumped by a factor of 40 in late 2000 when he upgraded his keyboard to USB2.0*

Typical Big Data Problem

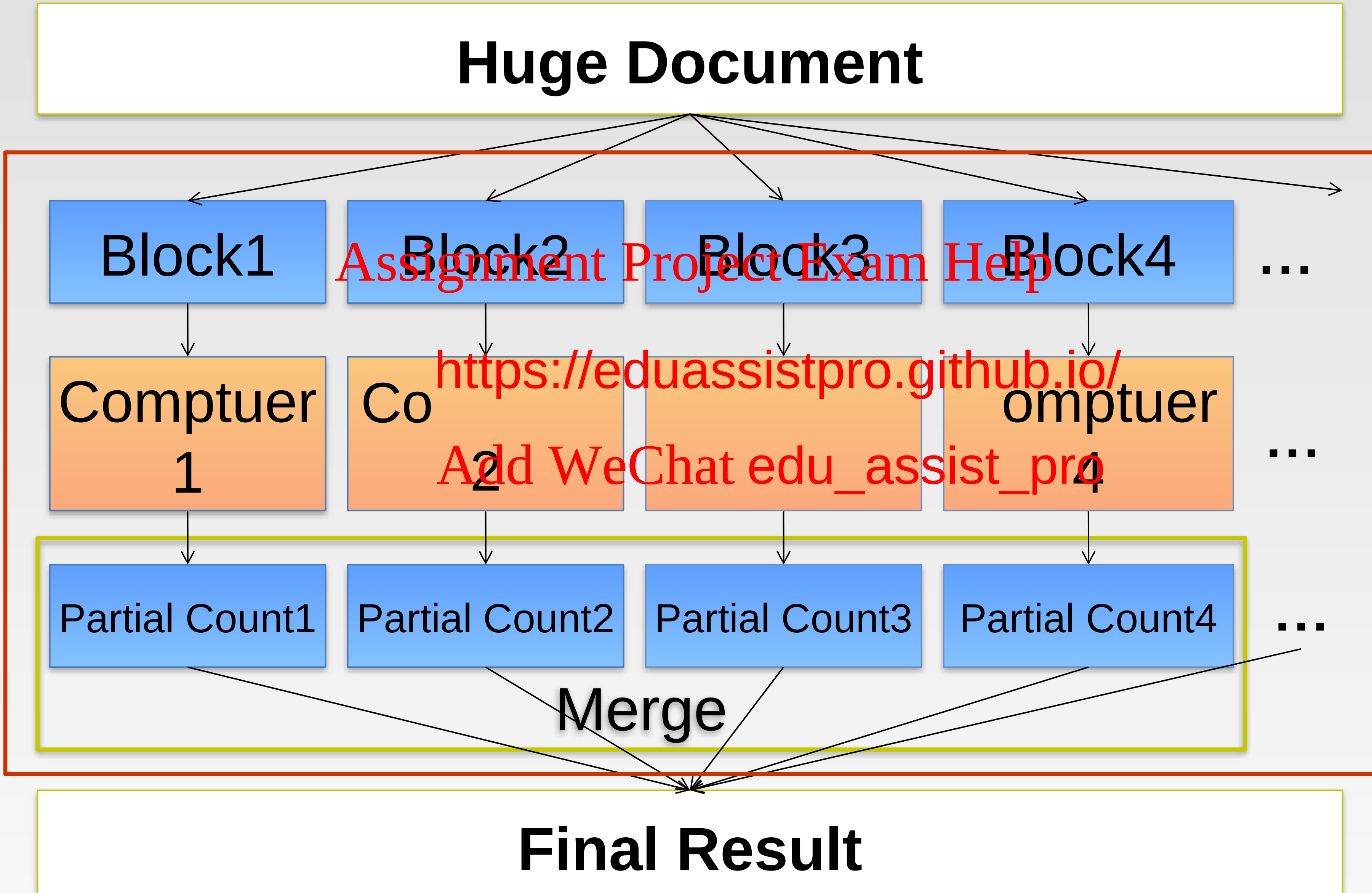
- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output

Map

educe

Key idea: provide l abstraction for these two oper

Distributed Word Count



The Idea of MapReduce

- Inspired by the map and reduce functions in functional programming
- We can view map as a transformation over a dataset
  - This transformation is specified by the function *f*
  - Each functional application happens in isolation
  - The application of *f* to each element of a dataset can be parallelized i
- We can view re n
  - The aggregation is defined by t
  - Data locality: elements in the lis ght together”
  - If we can group elements of the list, also the reduce phase can proceed in parallel
- The framework coordinates the map and reduce phases:
  - Grouping intermediate results happens in parallel

Data Structures in MapReduce

- Key-value pairs are the basic data structure in MapReduce
  - Keys and values can be: integers, float, strings, raw bytes
  - They can also be arbitrary data structures

- The design of MapReduce algorithms involves:

  - Imposing th y datasets
    - E.g.: for t keys may be URLs and values may be the HTM

  - In some algorithms, input keys .g., wordcount), in others they uniquely identify a record
  - Keys can be combined in complex ways to design various algorithms

Map and Reduce Functions

- Programmers specify two functions:
  - **map** $(k_1, v_1) \rightarrow$ list $[<k_2, v_2>]$
    - Map transforms the input into key-value pairs to process
  - **reduce** $(k_2,$ list $[v_2]) \rightarrow [<k_3, v_3>]$
    - Reduce aggregates the list of values for each key
    - All value the same reducer
  - list $[<k_2, v_2>]$ y k_2 as $(k_2,$ list $[v_2])$
- The MapReduce environment takes verything else…
- A complex program can be decomposed as a succession of Map and Reduce tasks

Everything Else?

- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles “data distribution”
  - Moves processes to data
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles errors
  - Detects wor
- Everything happens on top of a dist tem (HDFS)
- You don’t know:
  - Where mappers and reducers run
  - When a mapper or reducer begins or finishes
  - Which input a particular mapper is processing
  - Which intermediate key a particular reducer is processing

A Brief View of MapReduce

Shuffle and Sort

- Shuffle
  - Input to the Reducer is the sorted output of the mappers. In this phase the framework fetches the relevant partition of the output of all the mappers, via HTTP.
- Sort
  - The framework groups Reducer inputs by keys (since different Mappers ma this stage.
- Hadoop framew t step .

Hadoop MapReduce Brief Data Flow

- 1. Mappers read from HDFS
- 2. Map output is partitioned by key and sent to Reducers
- 3. Reducers sort input by key
- 4. Reduce output is written to HDFS
- Intermediate results are stored on local FS of Map and Reduce workers

“Hello World” in MapReduce

“Hello World” in MapReduce

- Input:
  - Key-value pairs: (docid, doc) of a file stored on the distributed filesystem
  - docid : unique identifier of a document
  - doc: is the text of the document itself

- Mapper:
  - Takes an in ... line

  - Emits intermediate key-value p ... the key and the integer is the value

- The framework:
  - Guarantees all values associated with the same key (the word) are brought to the same reducer
- The reducer:
  - Receives all values associated to some keys
  - Sums the values and writes output key-value pairs: the key is the word and the value is the number of occurrences

Coordination: Master

- Master node takes care of coordination:
  - 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 push

- Master pings workers periodically to

Dealing with Failures

- Map worker failure
  - Its task is reassigned to another map worker
  - Reduce workers are notified when task is rescheduled on another worker
- Reduce worker failure

  - Its task is re ker
  - Reduce task starting mapper tasks as well)

- Master failure

  - MapReduce task is aborted and client is notified
- Robust
  - Google's experience: lost 1600 of 1800 machines once!, but finished fine

Where the Magic Happens

- Implicit between the map and reduce phases is a parallel "group by" operation on intermediate keys
  - Intermediate data arrive at each reducer in order, sorted by the key
  - No ordering is guaranteed across reducers

- Output keys fro DFS

  - The output may consist of *r* disti e *r* is the number of reducers

  - Such output may be the input to a subsequent MapReduce phase

- Intermediate keys (used in shuffle and sort) are transient:
  - They are not stored on the distributed filesystem
  - They are "spilled" to the local disk of each machine in the cluster

Write Yo

nt in Java?

MapReduce Program

- A MapReduce program consists of the following 3 parts:
  - Driver → main (would trigger the map and reduce methods)
  - Mapper
  - Reducer
  - It is better to include the map reduce and main methods in 3 different cla

- Check detailed information of all cla
  https://hadoop.apache.org/docs/r2 ... es-noframe.html

Mapper

```
public static class TokenizerMapper
    extends Mapper<Object, Text, Text, IntWritable>{
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
```



```
        public void                    ontext context) throws
            I                          n {
```



```
            StringTokenizer itr =
                StringToke          tring());
```



```
            while (itr.hasMoreTokens()) {
                word.set(itr.nextToken());
                context.write(word, one);
            }
        }
}
```

Mapper Explanation

- Maps input key/value pairs to a set of intermediate key/value pairs.

//Map class header

public static class TokenizerMapper

extends Mapper<Object, Text, Text, IntWritable>{

- Class Mapper<KEYIN,VALUEIN,KEYOUT,VALUEOUT>
  - KEYIN,V oc)
  - KEYOUT rd, 1)

// IntWritable: A serializable and comp r integer

private final static IntWritable one = new IntWritable(1);

//Text: stores text using standard UTF8 encoding. It provides methods to serialize, deserialize, and compare texts at byte level

private Text word = new Text();

//hadoop supported data types for the key/value pairs, in package org.apache.hadoop

What is Writable?

- Hadoop defines its own “box” classes for strings (Text), integers (IntWritable), etc.

- All values must implement interface Writable

- All keys must im parable

- Writable is a serializable object whi a simple, efficient, serialization protocol

Mapper Explanation (Cont’)

//Map method header

public void map(Object key, Text value, Context context) throws IOException, InterruptedException

- Object key/Text value: Data type of the input Key and Value to the mapper

- Context: An inner class of Mapper, used to store the context of a running task output by either the Mapper or t tputs or the output of the job

- Exceptions: IOException, Interr
- This function is called once for each key/value pair in the input split. Your application should override this to do your job.

Mapper Explanation (Cont')

```
//Use a string tokenizer to split the document into words
StringTokenizer itr = new StringTokenizer(value.toString());
//Iterate through each word and a form key value pairs
while (itr.hasMoreTokens()) {
//Assign each work from the tokenizer (of String type) to a Text 'word'
        word.set(it
//Form key value pa                > using context
        context.write(word, one);
}
```


- Map function produces Map.Context object
  - Map.context() takes *(k, v)* elements
- Any *(WritableComparable, Writable)* can be used

Reducer

```
public static class IntSumReducer
      extends Reducer<Text,IntWritable,Text,IntWritable> {
            private IntWritable result = new IntWritable();

            public void reduce(Text key, Iterable<IntWritable> values,
            Context co                                        rruptedException{
                  in
                  for (IntWritable val :
                        sum += val;
                  }
                  result.set(sum);
                  context.write(key, result);
            }
}
```

Reducer Explanation

```
//Reduce Header similar to the one in map with different key/value data type
public static class IntSumReducer
      extends Reducer<Text, IntWritable, Text, IntWritable>
//data from map will be <"word",{1,1,..}>, so we get it with an Iterator and thus we can go through the sets of values
public void reduce(Text key, Iterable<IntWritable> values,
Context context) thro                                    n{
```



```
//Initaize a variable 'su
```



```
            int sum = 0;
//Iterate through all the values with respect t        up all of them
```



```
            for (IntWritable val : values) {
                        sum += val.get();
            }
// Form the final key/value pairs results for each word using context
            result.set(sum);
            context.write(key, result);
```

Main (Driver)

```
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setReducerCl
    job.setOutputKey
    job.setOutputValueClass(IntWritable
    FileInputFormat.addInputPath(job, n                    ]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
```

Main(The Driver)

- Given the Mapper and Reducer code, the short main() starts the MapReduction running
- The Hadoop system picks up a bunch of values from the command line on its own
- Then the main() also specifies a few key parameters of the problem in the Job object

- Job is the prima e a map-reduce job to the Hadoop fra hat Map and Reduce classes to use and the format of the ut files)

- Other parameters, i.e. the number o use, are optional and the system will determine good values for them if not specified
- Then the framework tries to faithfully execute the job as-is described by Job

Main Explanation

//Creating a Configuration object and a Job object, assigning a job name for identification purposes

```
Configuration conf = new Configuration();
Job job = Job.getInstance(conf, "word count");
```

- Job Class: It allows the user to configure the job, submit it, control its execution, and query the state. Normally the user creates the application, job via Job and then submits the

//Setting the job's jar file by finding the provi n

```
job.setJarByClass(WordCount.class);
```


//Providing the mapper and reducer class names

```
job.setMapperClass(TokenizerMapper.class);
job.setReducerClass(IntSumReducer.class);
```

//Setting configuration object with the Data Type of output Key and Value for map and reduce

```
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(IntWritable.class);
```

Main Explanation (Cont')

```
//The hdfs input and output directory to be fetched from the command line
FileInputFormat.addInputPath(job, new Path(args[0]));
FileOutputFormat.setOutputPath(job, new Path(args[1]));
//Submit the job to the cluster and wait for it to finish.
System.exit(job.waitForCompletion(true) ? 0 : 1);
```

Make It Running ！

- Configure environment variables

export JAVA_HOME=…

export PATH=${JAVA_HOME}/bin:${PATH}

export HADOOP_CLASSPATH=${JAVA_HOME}/lib/tools.jar

- Compile WordCount.java and create a jar:

$ hadoop com.sun.tools.j

$ jar cf wc.jar WordCount

- Put files to HDFS

$ hdfs dfs –put YOURFILES input

- Run the application

$ hadoop jar wc.jar WordCount input output

- Check the results

$ hdfs dfs –cat output/*

Make It Running ！

- Given two files:
  - file1: Hello World Bye World
  - file2: Hello Hadoop Goodbye Hadoop
- The first map emits:
  - < Hello, 1> < World, 1> < Bye, 1> < World, 1>
- The second ma
  - < Hello, 1> Hadoop, 1>
- The output of the job is:
  - < Bye, 1> < Goodbye, 1> < Had o, 2> < World, 2>

Mappers and Reducers

- Need to handle more data? Just add more Mappers/Reducers!
- No need to handle multithreaded code ☺
  - Mappers and Reducers are typically single threaded and deterministic
    - Determinism allows for restarting of failed jobs
  - Mappers/Re t of each other
    - In Hadoo

Combiners

- Often a Map task will produce many pairs of the form (k,v_1), (k,v_2), … for the same key k
  - E.g., popular words in the word count example
- Combiners are a general mechanism to reduce the amount of intermediate data, thus saving network time

  - They could be thought of as “mini-reducers”
- Warning!

  - The use of combiners must be t ly

    - Optional in Hadoop: the corr algorithm cannot depend on computation (or even execution) of the combiners
    - A combiner operates on each map output key. It must have the same output key-value types as the Mapper class.
    - A combiner can produce summary information from a large dataset because it replaces the original Map output
  - Works only if reduce function is commutative and associative
    - In general, reducer and combiner are not interchangeable

Combiners in WordCount

- Combiner combines the values of all keys of a single mapper node (single machine):

- Much less data needs to be copied and shuffled!
- If combiners take advantage of all opportunities for local aggregation we have at most $m \times V$ intermediate key-value pairs
  - m: number of mappers
  - V: number of unique terms in the collection
- Note: not all mappers will see all terms

Combiners in WordCount

- In WordCount.java, you only need to add the follow line to Main:

  job.setCombinerClass(IntSumReducer.class);

  - This is because in this example, Reducer and Combiner do the same thing
  - Note: Most cases this is not true!

  - You need to
- Given two files:

  - file1: Hello World Bye World
  - file2: Hello Hadoop Goodbye H

- The first map emits:
  - < Hello, 1> < World, 2> < Bye, 1>
- The second map emits:
  - < Hello, 1> < Hadoop, 2> < Goodbye, 1>

Partitioner

- Partitioner controls the partitioning of the keys of the intermediate map-outputs.
  - The key (or a subset of the key) is used to derive the partition, typically by a *hash function*.
  - The total number of partitions is the same as the number of reduce tasks for the job.
    - This cont s the intermediate key (and hen uction.
- System uses HashPartitioner by de
  - hash(key) mod R
- Sometimes useful to override the hash function:
  - E.g., ***hash(hostname(URL)) mod R*** ensures URLs from a host end up in the same output file
    - https://www.unsw.edu.au/faculties and https://www.unsw.edu.au/about-us will be stored in one file
- Job sets Partitioner implementation (in Main)

MapReduce: Recap

- Programmers must specify:
  - map $(k_1, v_1) \rightarrow [(k_2, v_2)]$
  - reduce $(k_2, [v_2]) \rightarrow [<k_3, v_3>]$
  - All values with the same key are reduced together
- Optionally, also:
  - combine (k_2,
    - Mini-red e map phase
    - Used as an optimization to r traffic
  - partition (k_2, number of partition k_2
    - Often a simple hash of the key, e.g., hash(k_2) mod n
    - Divides up key space for parallel reduce operations
- The execution framework handles everything else…

MapReduce: Recap (Cont')

- Divides input into fixed-size pieces*, **input splits***
  - Hadoop creates one map task for each split
  - Map task runs the user-defined map function for each *record* in the split
- Size of splits
  - Small size is r machine will be able to process
  - But if splits are too small, the ov aging the splits dominate the total execution tim
  - For most jobs, a good split size size of a HDFS block, 64MB(default)
- Data locality optimization
  - Run the map task on a node where the input data resides in HDFS
  - This is the reason why the split size is the same as the block size

MapReduce: Recap (Cont')

- Map tasks write their output to local disk (not to HDFS)
  - Map output is intermediate output
  - Once the job is complete the map output can be thrown away
  - So storing it in HDFS with replication would be overkill
  - If the node of map task fails, Hadoop will automatically rerun the map task on

- Reduce tasks d locality

  - Input to a single reduce task is tput from all mappers

  - Output of the reduce is stored in HDFS for reliability
- The number of reduce tasks is not governed by the size of the input, but is specified independently

MapReduce: Recap (Cont')

- When there are multiple reducers, the map tasks partition their output:
  - One partition for each reduce task
  - The records for every key are all in a single partition
  - Partitioning can be controlled by a user-defined partitioning function

More Detailed MapReduce Dataflow

MapReduce: Recap

Another Example ： Analysis of Weather Dataset

- Data from NCDC(National Climatic Data Center)
  - A large volume of log data collected by weather sensors: e.g. temperature
- Data format
  - Line-oriented ASCII format
  - Each record has many elements
  - We focus on the temperature element
  - Data files are organiz
  - There is a directory fo containing a gzipped file for each weather station with its readings for th
- Query
  - What’s the highest recorded global temperature for each year in the dataset?

Year Temperature

```
0067011990999991950051507004...9999999N9+00001+9999999999
9...
0043011990999991950051512004...9999999N9+00221+9999999999
9...
0043011990999991950051518004...9999999N9-
00111+9999999999...
0043012650999991949032412004...0500001N9+01111+9999999999
9...
0043012650999991949032418004...0500001N9+00781+9999999999
```

Contents of data files

```
% ls raw/1990 | head
010010-99999-1990.gz
010014-99999-1990.gz
010015-99999-1990.gz
010016-99999-1990.gz
010017-99999-1990.gz
010030-99999-1990.gz
010040-99999-1990.gz
010080-99999-1990.gz
010100-99999-1990.gz
010150-99999-1990.gz
```

List of data files

Analyzing the Data with Unix Tools

- To provide a performance baseline
- Use *awk* for processing line-oriented data
- Complete run for the century took **42 minutes** on a single EC2 High-CPU Extra Large Instance


```
% ./max_temperature.sh
1901    317
1902    244
1903    289
1904    256
1905    283
...
```

How Can We Parallelize This Work?

- To speed up the processing, we need to run parts of the program in **parallel**
- Challenges?
  - Divide the work into even distribution is not easy
    - File size for different years varies
  - Combining t
    - Get the r rature for each chunk
  - We are still limited by the proce of a single machine
    - Some datasets grow beyon f a single machine
- To use **multiple machines**, we need to consider a variety of complex problems
  - Coordination: Who runs the overall job?
  - Reliability: How do we deal with failed processes?
- **Hadoop** can take care of these issues

MapReduce Design

- We need to answer these questions:
  - What are the map input key and value types?
  - What does the mapper do?
  - What are the map output key and value types?
  - Can we use a combiner?
  - Is a partition
  - What does t
  - What are the reduce output key s?

- And: What are the file formats?
  - For now we are using text files
  - We may use binary files

MapReduce Types

- General form

  map: (K1, V1) → list(K2, V2)

  reduce: (K2, list(V2)) → list(K3, V3)

- Combine function

  - The same form as the reduce fu its output types
  - Output type is the same as Map
  - The combine and reduce functions may be the same

- Partition function

  partition: (K2, V2) → integer

  - Input intermediate key and value types
  - Returns the partition index

MapReduce Design

- Identify the input and output of the problem
  - Text input format of the dataset files (input of mapper)
    - Key: offset of the line (unnecessary)
    - Value: each line of the files (string)
  - Output (output of reducer)
    - Key: yea
    - Value: m

- Decide the MapReduce data types
  - Hadoop provides its own set of
    - optimized for network serialization
    - org.apache.hadoop.io package
  - In WordCount, we have used Text and IntWritable
  - Key must implement interface WritableComparable
  - Value must implement interface Writable

Writable Wrappers

Writable Class Hierarchy

What does the Mapper Do?

- Pull out the year and the temperature
  - Indeed in this example, the map phase is simply data preparation phase
  - Drop bad records(filtering)

Input File

of Map Function (key, value)

Input of Map Function (key, value)

```
(0, 0067011990999991950051507004...9999999N9+00001+99999999999...)
(106, 0043011990999991950051512004...9999999N9+00221+99999999999...)
(212, 0043011990999991950051518004...9999999N9-00111+99999999999...)
(318, 0043012650999991949032412004...0500001N9+01111+99999999999...)
(424, 0043012650999991949032418004...0500001N9+00781+99999999999...)
```

Map

```
(1950, 0)
(1950, 22)
(1950, -11)
(1949, 111)
(1949, 78)
```

Map Input and Output

- Input
  - Key: offset of the line (unnecessary)
    - The dataset is quite large and contains a huge number of lines
    - LongWritable
  - Value: each line of the files (string)
    - Text
- Output
  - Key: year
    - Both string or integer forma
    - Text/IntWritable
  - Value: temperature
    - Integer is already enough to store it
    - IntWritable
- Combiner and Partitioner?

What does the Reducer Do?

- Reducer input
  - (year, [temperature1, temperature2, temperature3, …])

- Scan all values received for the key, and find out the maximum one

- Reducer output
  - Key: year
    - String/IntWritable
  - Value: maximum temperature
    - IntWritable

MapReduce Design of NCDC Example

- The output from the map function is processed by MapReduce framework
  - Sorts and groups the key-value pairs by key

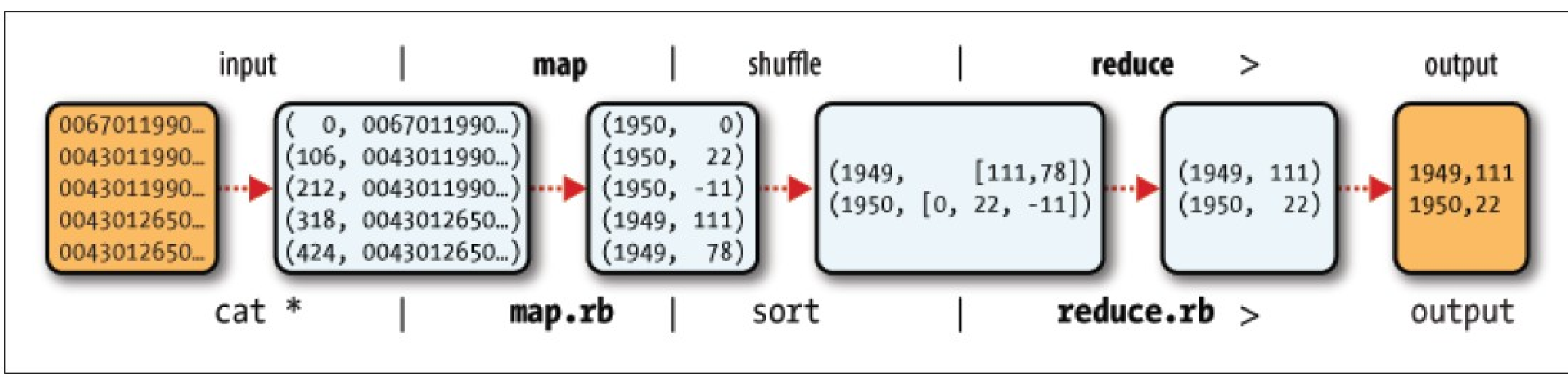
Sort and Group By

```
(1949, [111, 78])
(1950, [0, 22, -11])
```


- Reduce function itera the maximum value

Reduce


```
(1949, [111, 78])
(1950, [0, 22, -11])
```



Java Implementation of the Example

```
public class MaxTemperatureMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
  private static final int MISSING = 9999;

  @Override
  public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
    String line = value.toString();
    String year = line.substring(15, 19);
    int airTemperature;
    if (line.charAt(87) == '+') {
      airTemperature = Integer.
    } else {
      airTemperature = Integer.parseInt(line.substring(87, 92
    }
    String quality = line.substring(92, 93);
    if (airTemperature != MISSING && quality.matches("[01459]")) {
      context.write(new Text(year), new IntWritable(airTemperature));
    }
  }
}
```

Java Implementation of the Example

```
public class MaxTemperatureReducer
                extends Reducer<Text, IntWritable, Text, IntWritable> {
  @Override
  public void reduce(Text key, Iterable<IntWritable> values,
      Context context) throws IOException, InterruptedException {
    int maxValue = Integer.MIN_VALUE;
    for (IntWritable valu
      maxValue = Math.
    }
    context.write(key, new IntWritable(maxV
  }
}
```

Java Implementation of the Example

```
public class MaxTemperatureWithCombiner {
//specify the usage of the job
  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: MaxTemperatureWithCombiner <input path> " + "<output path>");
      System.exit(-1);
    }
//Construct a job object to configure, control and run the job
    Job job = new Job();
    job.setJarByClass(MaxTemp
    job.setJobName("Max temper
//Specify input and output pat
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
//Specify map and reduce classes, also a combiner
    job.setMapperClass(MaxTemperatureMapper.class);
    job.setCombinerClass(MaxTemperatureReducer.class);
    job.setReducerClass(MaxTemperatureReducer.class);
//Specify output type
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
//submit the job and wait for completion
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```


Codes can be found here:
http://hadoopbook.com/code.html

References

- Chapter 2, Hadoop The Definitive Guide