Advanced Data Management Technologies

Unit 16 — MapReduce

J. Gamper

Free University of Bozen-Bolzano
Faculty of Computer Science
IDSE

Acknowledgements: Much of the information in this unit is from slides of Paul Krzyzanowski, Jerry Zhao, and Jelena Pjesivac-Grbovic.

Outline

1. Introduction
2. MR Programming Model
3. Extensions and Optimizations
4. MapReduce Implementations and Alternatives

Outline

1 Introduction

2 MR Programming Model

3 Extensions and Optimizations

4 MapReduce Implementations and Alternatives

Motivation

In pioneer days they used oxen for heavy pulling, and when one ox couldn't budge a log, they didn't try to grow a larger ox. We shouldn't be trying for bigger computers, but for more systems of computers.

— Grace Hopper

- Many problems cannot be easily scaled to the Web, e.g., $\approx$ 20TB per Google crawl
  - Document inversion
  - PageRank computation
  - Web log mining
- Traditional programming is serial.
- Parallel programming breaks processing into parts that can be executed concurrently on multiple processors.
- Large clusters of commodity Hardware/PCs are networked.
- Challenge
  - Provide a simple framework for distributed/parallel data processing based on the available commmodity hardware.

Simplest Environment for Parallel Processing

- No dependency among data
- Data can be split into equal-size chunks
- Each process can work on a chunk
- Master/worker approach
  - Master
    - Splits data into chunks according to # of workers
    - Sends each worker a chunk
    - Receives the results from each worker
  - Worker
    - Receives a chunk from master
    - Performs processing
    - Sends results to master

Challenges of Parallel/Distributed Processing

- There are dependencies among data
- Identify tasks that can run concurrently
- Identify groups of data that can be processed concurrently
- Not all problems can be parallelized!
- Communication and synchronization between distributed nodes
- Distribute and balance tasks/data to optimize the throughput
- Error handling if node or parts of the network are broken

MapReduce

- A distributed programming model
- Created by Google in 2004 (Jeffrey Dean and Sanjay Ghemawat)
- Inspired by LISP's map and reduce functions
  - Map(function, set of values)
    - Applies function to each value in the set
    - (map 'length'(() (a) (a b) (a b c))) ⇒ (0 1 2 3)
  - Reduce(function, set of values)
    - Combines all the values using a binary function (e.g., +)
    - (reduce '+'(1 2 3 4 5)) ⇒ 15

MapReduce Features

- Complete framework for parallel and distributed computing
- Programmers get a simple but powerful API
  - map function
  - reduce function
- Programmers don't have to worry about handling
  - parallelization
  - data distribution
  - load balancing
  - fault tolerance
- Detects machine failures and redistributes work
- Implementation within hours, not weeks
- Allows to process huge amounts of data (terabytes and petabytes) on thousands of processors.

Outline

1 Introduction

2 MR Programming Model

3 Extensions and Optimizations

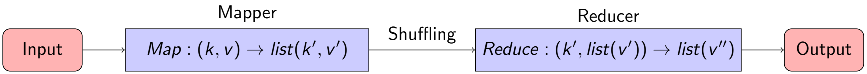
4 MapReduce Implementations and Alternatives

Common Data Processing Pattern

- The following five steps characterize much of our data processing
  1. Iterate over large amounts of data
  2. Extract something of interest
  3. Group things of interest
  4. Aggregate interesting things
  5. Produce output
- MapReduce provides an abstraction of these steps into two operations
  - Map function: combines step 1 + 2
  - Reduce function: combines step 3 + 4 + 5

Basic MapReduce Programming Model

- User specifies two functions that have key/value pairs in input and output
- $Map : (k, v) \rightarrow list(k', v')$
  - Function is applied to each input key/value pair
  - Produces one or more intermediate key/value pairs
- $Reduce : (k', list(v')) \rightarrow list(v'')$
  - All intermediate values for a particular key are first merged
  - Function is applied to each key/(merged) values to aggregate them


Mapper
Reducer
Input
$Map : (k, v) \rightarrow list(k', v')$
Shuffling
$Reduce : (k', list(v')) \rightarrow list(v'')$
Output

- Shuffling is the process of grouping and copying the intermediate data from the mappers' local disk to the reducers

MapReduce Example

- Compute the total adRevenue for the following relation:
  `UserVisits(sourceIP, destURL, adRevenue, userAgent, ...)`:
- Map function
  - Assumes that input tuples are strings separated by "|"
  - Generates key/value pairs `(sourceIP, adRevenue)`

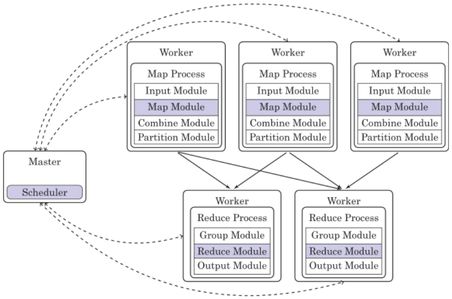
```
map(String key, String value);
String[] array = value.split(‘‘|’’);
EmitIntermediate(array[0], ParseFloat(array[2]);
```

- Reduce function
  - Intermediate key/value pairs are grouped into (sourceIP, [adRevenue1, . . .])
  - Sum of adRevenue values for each sourceIP are output

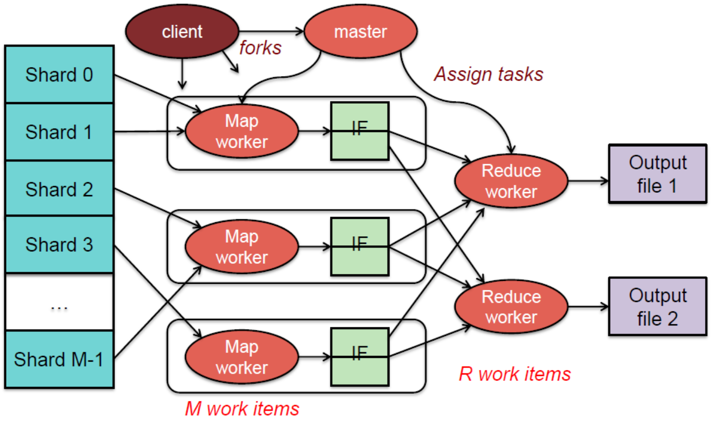
```
reduce(String key, Iterator values);
float totalRevenue = 0;
while values.hasNext() do
    totalRevenue += values.next();
Emit(key, totalRevenue);
```

MapReduce Architecture

- MapReduce processing engine has two types of nodes:
  - Master node: controls the execution of the tasks;
  - Worker nodes: responsible for the `map` and `reduce` tasks.
- Basic MapReduce engine includes the following modules:
  - Scheduler: assigns map and reduce tasks to worker nodes
  - Map module: scans a data chunk and invokes the `map` function
  - Reduce module: pulls intermediate key/values pairs from the mappers, merges the data by keys, and applies the reduce function

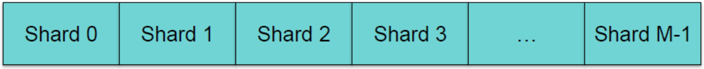


MapReduce Execution Overview



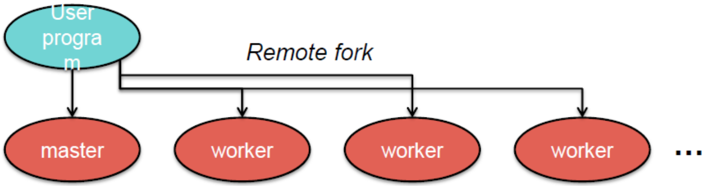
MR Step 1: Split Input Files

- Input can be many files or a single big file.
- Break up the input data into M pieces (typically 64 MB)



MR Step 2: Fork Processes

- Start up many copies of the program on a cluster of machines
  - One master node: scheduler & coordinator
  - Lots of worker nodes
- Idle workers are assigned either
  - map tasks (each works on a shard) – there are M map tasks/workers
  - reduce tasks (each works on intermediate files) – there are R reduce tasks ($R = \#$ of partitions defined by the user)

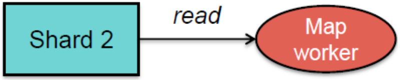


MR Step 3: Map Task

- Reads contents of the input shard assigned to it
- Parses key/value pairs out of the input data
- Passes each pair to the user-defined map function

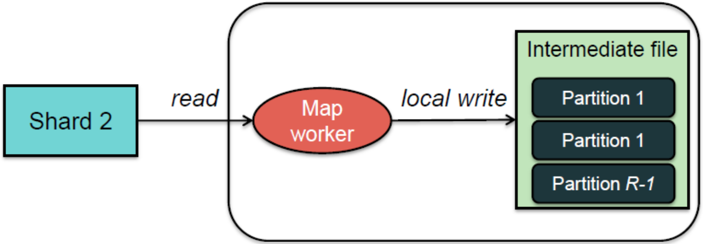
$$map : (k, v) \rightarrow list(k', v')$$

  which produces intermediate key/value pairs
- They are buffered in local memory



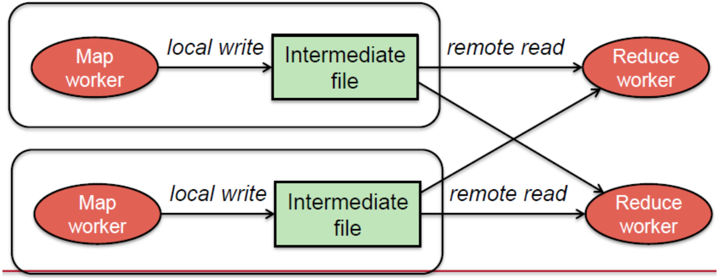
MR Step 4: Intermediate Files and Partitioning

- Intermediate key/value pairs are periodically written from memory to local disk.
- Thereby, key/value pairs are sorted by keys and grouped into R partitions
- Default partitioning function: $hash(key) \mod R$
- Master node is notified about the position of the intermediate result
- Reduce nodes will read the associated partition from every Map node



MR Step 5: Sorting

- Reduce worker gets notified by the master about the location of intermediate files for its partition.
- Uses RPCs to read the data from the local disks of the map workers.
- When the reduce worker reads intermediate data:
  - it merge-sorts the data from the different map tasks by the intermediate keys
  - such that all occurrences of the same key are grouped together.

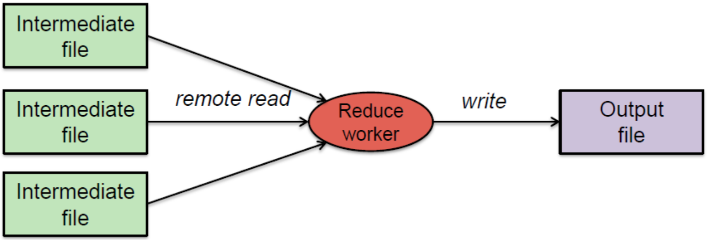


MR Step 6: Reduce Task

- Key and set of intermediate values for that key is given to the reduce function:

$$reduce : (k', [v'_1, v'_2, v'_3, v'_4, \ldots]) \rightarrow list(v'')$$

- The output of the Reduce function is appended to an output file.
- The reduce function can only start when all mappers are done!



MR Step 7: Return to User

- When all map and reduce tasks have completed, the master wakes up the user program.
- The MapReduce call in the user program returns and the program can resume execution.
- Output of MapReduce is available in R output files.

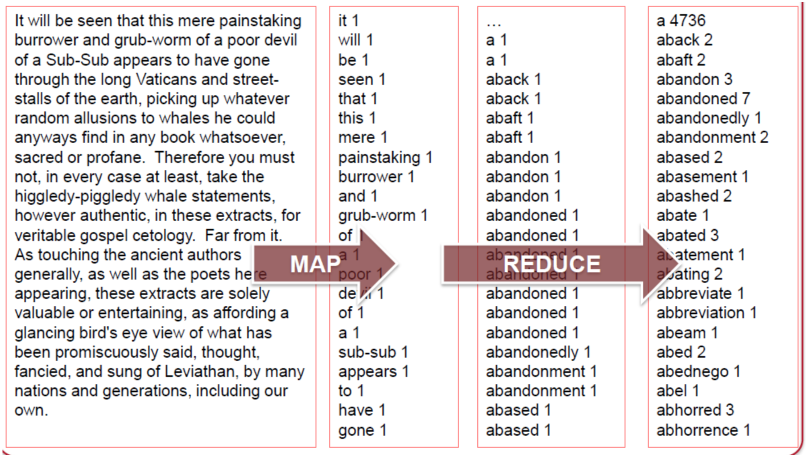
Word Count Example/1

- Task: Count # of occurrences of each word in a collection of documents
- Input: Large number of text documents
- Output: Word count across all the documents
- MapReduce solution
  - Map: Parse data and output (*word*, "1") for every word in a document.
  - Reduce: For each word, sum all occurrences and output (*word*, *total_count*)

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

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

Word Count Example/2



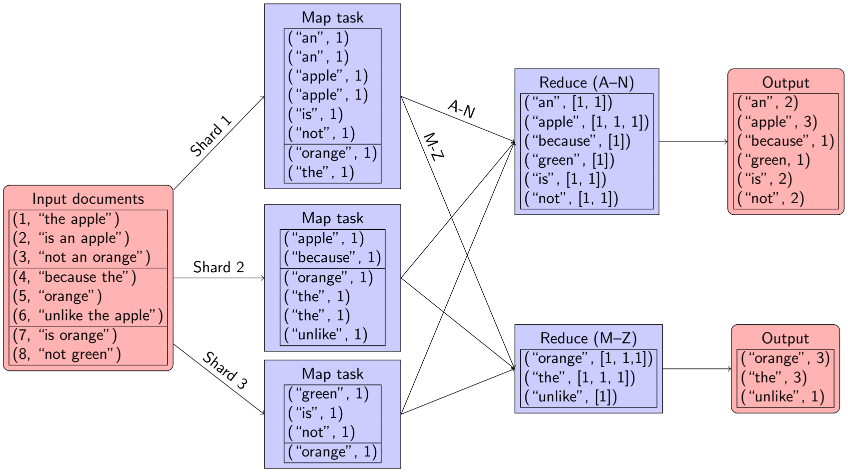
It will be seen that this mere painstaking burrower and grub-worm of a poor devil of a Sub-Sub appears to have gone through the long Vaticans and street-stalls of the earth, picking up whatever random allusions to whales he could anyways find in any book whatsoever, sacred or profane. Therefore you must not, in every case at least, take the higgledy-piggledy whale statements, however authentic, in these extracts, for veritable gospel cetology. Far from it. As touching the ancient authors generally, as well as the poets here appearing, these extracts are solely valuable or entertaining, as affording a glancing bird's eye view of what has been promiscuously said, thought, fancied, and sung of Leviathan, by many nations and generations, including our own.

```
it 1
will 1
be 1
seen 1
that 1
this 1
mere 1
painstaking 1
burrower 1
and 1
grub-worm 1
of 1
a 1
poor 1
devil 1
of 1
a 1
sub-sub 1
appears 1
to 1
have 1
gone 1
```

```
…
a 1
a 1
aback 1
aback 1
abaft 1
abaft 1
abandon 1
abandon 1
abandon 1
abandoned 1
abandoned 1
abandoned 1
abandoned 1
abandoned 1
abandoned 1
abandoned 1
abandonedly 1
abandonment 1
abandonment 1
abased 1
abased 1
```

```
a 4736
aback 2
abaft 2
abandon 3
abandoned 7
abandonedly 1
abandonment 2
abased 2
abasement 1
abashed 2
abate 1
abated 3
abatement 1
abating 2
abbreviate 1
abbreviation 1
abeam 1
abed 2
abednego 1
abel 1
abhorred 3
abhorrence 1
```

Word Count Example/3



Outline

1 Introduction

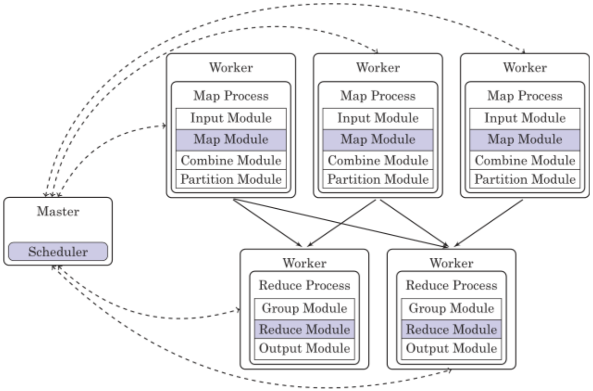
2 MR Programming Model

3 Extensions and Optimizations

4 MapReduce Implementations and Alternatives

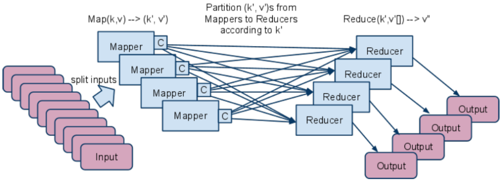
MR Extensions and Optimizations

- To improve efficiency and usability, the basic MR architecture (scheduler, map module and reduce module) is usually extended by additional modules that can be customized by the user.



Extensions and Optimizations in Map Process

- Input module
  - Responsible for recognizing the input data with different input formats and splitting the input data into key/value pairs.
  - Supports different storage systems, e.g., text files, binary files, databases
- Combine module
  - combine: $(k', list(v')) \rightarrow list(k', v'')$
  - Mini-reducer that runs in the mapper to reduce the number of key/value paris shuffled to the reducer (reduce network traffic)



- Partition module
  - Divides up the intermediate key space for parallel reduce operations,
    - specifies which key/value pairs are shuffled to which reducers
  - Default partition function: $f(k') = hash(k') \mod \#reducers$

Extensions and Optimizations in Reduce Process

- Output module
  - Similar to input module, but for the output
- Group module
  - Specifies how to merge data received from different mappers into one sorted run in the reduce phase
  - Example: if the map output key is a composition ($sourceIP, destURL$), the group function can only compare a subset ($sourceIP$)
  - Thus, the reduce function is applied to the key/value pairs with the same $sourceIP$.

Word Count Example: Combiner Function

```
combine(String key, Iterator values);
// key: a word; values: a list of counts
int partial_word_count = 0;
foreach v in values do
    partial_word_count += ParseInt(v);
Emit(key, AsString(partial_word_count));
```

Relative Word Frequency Example: Naive Solution

- Input: Large number of text documents
- Task: Compute relative word frequency across all documents
  - Relative frequency is calculated with respect to the total word count

- A naive solution with basic MapReduce model requires two MR cycles
  - MR1: count number of all words in these documents
  - MR2: count number of each word and divide it by the total count from MR1

- Can we do it better?

Features of Google's MR Implementation

- Google's MapReduce implementation offers two nice features
  - Ordering guarantee of reduce keys
    - Reducer processes the (key, list(value))-pairs in the order of the keys
  - Auxiliary functionality: $\mathit{EmitToAllReducers}(k, v)$
    - Sends k/v-pair to all reducers

Rel. Word Frequency Example: Advanced Solution

- The features in the previous slide allow better solution to compute the relative word frequency
  - Only one MR cycle is needed
  - Every map task sends its total word count with key "" to all reducers (in addition to the word count "1" for each single word)
  - The sum of values with key "" gives the total number of words
  - Key "" will be the first key processed by the reducer
    - Thus, total number of words is known before processing individual words

Rel. Word Frequency Example: Mapper/Combiner

```
map(String key, String value);
// key: document name; value: document contents
int word_count = 0;
foreach word w in value do
    EmitIntermediate(w, "1");
    word_count++;
EmitIntermediateToAllReducers("", AsString(word_count));
```

```
combine(String key, Iterator values);
// key: a word; values: a list of counts
int partial_word_count = 0;
foreach v in values do
    partial_word_count += ParseInt(v);
Emit(key, AsString(partial_word_count));
```

Rel. Word Frequency Example: Reducer

```
reduce(String key, Iterator values);
// key: a word; values: a list of counts
if key == "" then
    total_word_count = 0;
    foreach v in values do
        total_word_count += ParseInt(v);
else
    // key != ""
    int word_count = 0;
    foreach v in values do
        word_count += ParseInt(v);
    Emit(key, AsString(word_count / total_word_count));
```

Other Examples

- Distributed grep (search for words)
  - Task: Search for words in lots of documents
  - Map: emit a line if it matches a given pattern
  - Reduce: just copy the intermediate data to the output
- Count URL access frequency
  - Task: Find the frequency of each URL in web logs
  - Map: process logs of web page access; output $<$URL, 1$>$
  - Reduce: add all values for the same URL
- Inverted index
  - Task: Find what documents contain a specific word
  - Map: parse document, emit $<$word, document-ID$>$ pairs
  - Reduce: for each word, sort the corresponding document IDs
    Emit a $<$word, list(document-ID)$>$-pair
    The set of all output pairs is an inverted index

Outline

1 Introduction

2 MR Programming Model

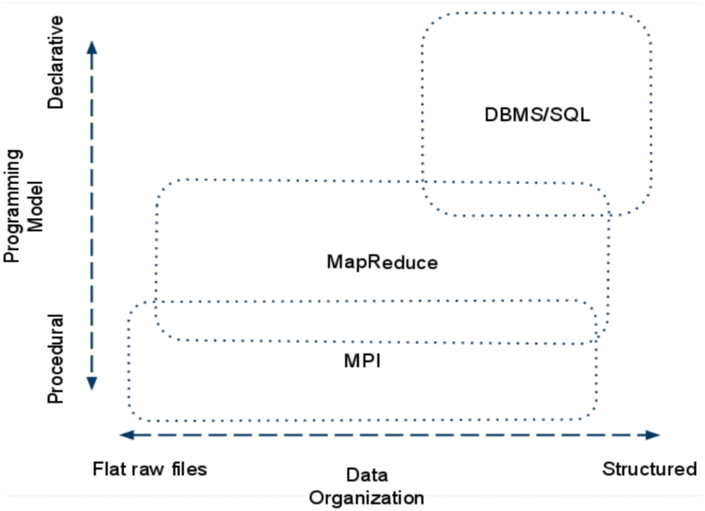
3 Extensions and Optimizations

4 **MapReduce Implementations and Alternatives**

Comparing MapReduce and RDBMS

	Traditional RDBMS	**MapReduce**
Data size	Gigabytes	Petabytes
Access	Interactive and batch	Batch
Updates	Read and write many times	Write once, read many times
Structure	Static schema	Dynamic schema
Integrity	High	Low
Scaling	Nonlinear	Linear

Comparing MPI, MapReduce, and RDBMS/1



Comparing MPI, MapReduce, and RDBMS/2

	MPI	**MapReduce**	**DBMS/SQL**
What they are	A general parrellel programming paradigm	A programming paradigm and its associated execution system	A system to store, manipulate and serve data
Programming Model	Messages passing between nodes	Restricted to Map/Reduce operations	Declarative on data query/retrieving; stored procedures
Data organization	No assumption	"files" can be sharded	Organized data structures
Data to be manipulated	Any	k, v-pairs: string	Tables with rich types
Execution model	Nodes are independent	Map/Shuffle/Reduce, Checkpointing/Backup, Physical data locality	Transaction, Query/operation optimization, Materialized view
Usability	Steep learning curve; difficult to debug	Simple concept; Could be hard to optimize	Declarative interface; Could be hard to debug in runtime
Key selling point	Flexible to accommodate various applications	Plow through large amount of data with commodity hardware	Interactive querying the data; Maintain a consistent view across clients

Different MapReduce Implementations

- Google MapReduce
  - Original proprietary implementation
  - Based on proprietary infrastructures
    - GFS(SOSP'03), MapReduce(OSDI'04) , Sawzall(SPJ'05), Chubby (OSDI'06), Bigtable(OSDI'06)
    - and some open source libraries
  - Support C++, Java, Python, Sawzall, etc.
- Apache Hadoop MapReduce
  - Most common (open-source!) implementation
  - Built on specs defined by Google
  - Plus the whole equivalent package, and more
    - HDFS, Map-Reduce, Pig, Zookeeper, HBase, Hive
  - Used by Yahoo!, Facebook, Amazon and Google-IBM NSF cluster
- Amazon Elastic MapReduce
  - Uses Hadoop MapReduce running on Amazon EC2
- Dryad
  - Proprietary, based on Microsoft SQL servers
  - Dryad(EuroSys'07), DryadLINQ(OSDI'08)
  - Michael's Dryad TechTalk@Google (Nov.'07)

Comparison of MapReduce Implementations

Name	Language	File System	Index	Master Server	Multiple Job Support
Hadoop	Java	HDFS	No	Name Node and Job Tracker	Yes
Cascading	Java	HDFS	No	Name Node and Job Tracker	Yes
Sailfish	Java	HDFS + $\mathcal{I}$-file	No	Name Node and Job Tracker	Yes
Disco	Python and Erlang	Distributed Index	Disco Server	No	No
Skynet	Ruby	MySQL or Unix File System	No	Any node in the cluster	No
FileMap	Shell and Perl Scripts	Unix File System	No	Any node in the cluster	No
Themis	Java	HDFS	No	Name Node and Job Tracker	Yes

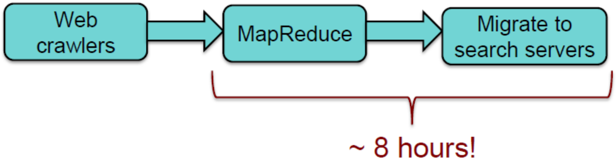
- Other implementations
  - Oracle provides a MapReduce implementation by using its parallel pipelined table functions and parallel operations
  - New DBMSs provide built-in MR support, e.g., Greenplum (http://www.greenplum.com), Aster (http://www.asterdata.com/), MongoDB (http://www.mongodb.org)
  - Some stream systems, such as IBM's SPADE, are also enhanced with MR

MapReduce @ Google/1

- Google's hammer for 80% of data crunching
    - Large-scale web search indexing
    - Clustering problems for Google News
    - Produce reports for popular queries, e.g. Google Trend
    - Processing of satellite imagery data
    - Language model processing for statistical machine translation
    - Large-scale machine learning problems
    - Just a plain tool to reliably spawn large number of tasks
        - e.g. parallel data backup and restore

MapReduce @ Google/2

- MapReduce was used to process web data collected by Google's crawlers.
  - Extract the links and metadata needed to search the pages
  - Determine the site's PageRank
  - Move results to search servers
  - The process took around eight hours.



- Web has become more dynamic
  - An 8+ hour delay is a lot for some sites
- Goal: refresh certain pages within seconds
- Search framework updated in 2009-2010: Caffeine
  - Index updated by making direct changes to data stored in BigTable
- MapReduce is still used for many Google services

What is Hadoop?/1

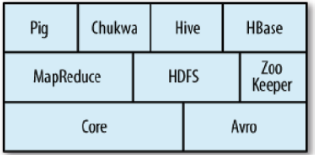
- A software framework that supports data-intensive distributed applications.
- It enables applications to work with thousands of nodes and petabytes of data.
- Hadoop was inspired by Google's MapReduce and Google File System (GFS).
- Hadoop is a top-level Apache project being built and used by a global community of contributors, using the Java programming language.
- Yahoo! has been the largest contributor to the project, and uses Hadoop extensively across its businesses.

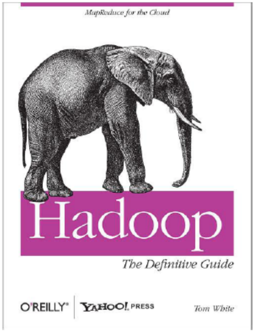
What is Hadoop?/2

http://hadoop.apache.org/

Hadoop: not only a Map/Reduce implementation!

- HDFS – distributed file system
- Pig – high level query language (SQL like)
- HBase – distributed column store
- Hive – Hadoop based data warehouse
- ZooKeeper, Chukwa, Pipes/Streaming, …





Who uses Hadoop?

- Yahoo!
  - More than 100,000 CPUs in >36,000 computers.
- Facebook
  - Used in reporting/analytics and machine learning and also as storage engine for logs.
  - A 1100-machine cluster with 8800 cores and about 12 PB raw storage.
  - A 300-machine cluster with 2400 cores and about 3 PB raw storage.
  - Each (commodity) node has 8 cores and 12 TB of storage.

Hadoop API/1

- Input
  - Set of files that are spread out over the Hadoop Distributed File System (HDFS)
- **Map phase/tasks**
  - Record reader
    - Translates an input shard/split into key-value pairs (records).
  - Map
    - Applies the map function.
  - Combiner
    - An optional localized reducer to aggregate values of a single mapper.
    - Is an optimization and can be called 0, 1, or several times.
    - No guarantee how often it is called!
  - Partitioner
    - Takes the intermediate key-value pairs from the mapper and splits them up into shards (one shard per reducer).

Hadoop API/2

- **Reduce phase/tasks**
  - Shuffle and sort
    - Reads the output files written by all of the partitioners and downloads them to the local machine.
    - The individual data are sorted by the intermediate key into one large data list $\rightarrow$ group equivalent keys together.
    - This step is not customizable, i.e., completely done by the system.
    - Only customization is to specify a `Comparator` class for sorting the data.
  - Reduce
    - Apply the reduce function.
  - Output format
    - Translates the final key-value pairs from the `reduce` function into a customized output format.
    - The output is written to HDFS.

WordCount Example in Hadoop – Mapper

```
public class WordCountMapper extends Mapper<LongWritable, Text, Text, IntWritable> {

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

    @Override
    public void map(LongWritable key, Text value, final Context context) throws IOException,
        InterruptedException {

        String line = value.toString().toLowerCase();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }

    }
}
```

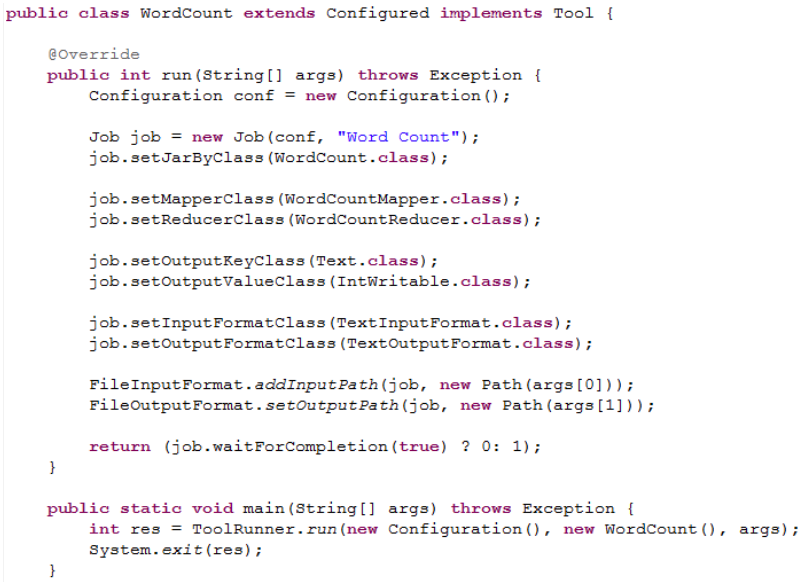
- `Mapper` class with abstract `map` function.
- Four parameters: type of input key, input value, output key, output value.
- Hadoop provides its own set of data types that are optimized for network serialization, e.g., `Text` (= `String`) or `IntWritable` (= `int`).
- `map` has 3 parameters: key, value, context where to write the output.

WordCount Example in Hadoop – Reducer

```java
public class WordCountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {

    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context context)
        throws IOException, InterruptedException {

        int sum = 0;
        for (IntWritable value : values) {
            sum += value.get();
        }
        context.write(key, new IntWritable(sum));

    }

}
```

- `Reducer` class with abstract `reduce` function.
- Four parameters: type of input key, input value, output key, output value.
- `reduce` has 3 parameters: key, value, context where to write the output.
- Input types of `reduce` must match the output types of `map`.

WordCount Example in Hadoop – Main



```java
public class WordCount extends Configured implements Tool {

    @Override
    public int run(String[] args) throws Exception {
        Configuration conf = new Configuration();

        Job job = new Job(conf, "Word Count");
        job.setJarByClass(WordCount.class);

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);

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

        job.setInputFormatClass(TextInputFormat.class);
        job.setOutputFormatClass(TextOutputFormat.class);

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        return (job.waitForCompletion(true) ? 0: 1);
    }

    public static void main(String[] args) throws Exception {
        int res = ToolRunner.run(new Configuration(), new WordCount(), args);
        System.exit(res);
    }
```

Limitations of MapReduce

- Batch-oriented
- Not suited for near-real-time processes
- Cannot start a new phase until the previous has completed
  - Reduce cannot start until all Map workers have completed
- Suffers from “stragglers” – workers that take too long (or fail)

Summary

- MapReduce is a framework for distributed and parallel data processing
- Simple programming model with a map and reduce function
- Handles automatically parallelization, data distribution, load balancing and fault tolerance
- Allows to process huge amounts of data by commodity hardware.
- Different MapReduce implementations are available