MapReduce: Simplified Data Processing on Large Clusters

Papers We Love
Bucharest Chapter
October 12, 2015

Hello!

I am Adrian Florea

Architect Lead @ IBM Bucharest Software Lab

What is?

- MapReduce is a programming model and implementation for processing large data sets

- Users specify a Map function that processes a key-value pair to generate a set of intermediate key-value pairs and a Reduce function that aggregates all intermediate values that share the same intermediate key in order to combine the derived data appropriately

map:(k1, v1)->[(k2, v2)]
reduce:(k2, [v2])->[(k3, v3)]

Where else have we seen this?

Map

- function (mathematics)
- map (Java)
- Select (.NET)

Reduce

- fold (functional programming)
- reduce (Java)
- Aggregate (.NET)

```
<R> Stream<R> map(Function<? super T,? extends R> mapper)
```

```
T reduce(T identity,
         BinaryOperator<T> accumulator)
```

```
public static IEnumerable<TResult> Select<TSource, TResult>(
        this IEnumerable<TSource> source,
        Func<TSource, TResult> selector
)
```

```
public static TAccumulate Aggregate<TSource, TAccumulate>(
        this IEnumerable<TSource> source,
        TAccumulate seed,
        Func<TAccumulate, TSource, TAccumulate> func
)
```

Other analogies

```sql
SELECT SalesOrderID, SUM(LineTotal)
FROM SalesOrderDetail
GROUP BY SalesOrderID
```

"To draw an analogy to SQL, map is like the group-by clause of an aggregate query. Reduce is analogous to the aggregate function that is computed over all the rows with the same group-by attribute"

D.J. DeWitt & M. Stonebraker

Divide-and-conquer algorithms

“recursively breaking down a problem into two or more sub-problems of the same (or related) type (divide), until these become simple enough to be solved directly (conquer). The solutions to the sub-problems are then combined to give a solution to the original problem.”

December 6, 2004 4PM

Jeffrey Dean
Google Fellow

Sanjay Ghemawat
Google Fellow

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

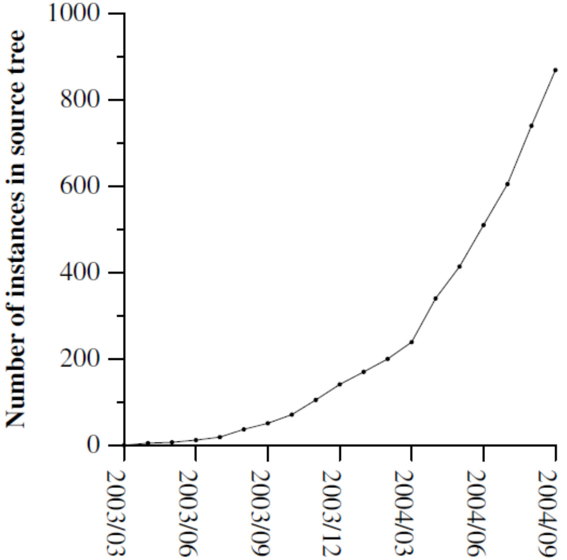
As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

History

- April 1960: John McCarthy introduced the concept of “maplist”
- September 4, 1998: Google founded
- 1998-2003: hundreds of special-purpose large data computation programs in Google
- February 2003: 1st version of MapReduce
- August 2003: MapReduce significant enhancements
- June 18, 2004: Patent US7650331 B1 filed
- December 6, 2004: 1st MapReduce public presentation
- 2005: Hadoop implementation started in Java (Douglass R. Cutting & Michael J. Cafarella)
- September 4, 2007: Hadoop 0.14.1
- January 19, 2010: Patent US7650331 B1 published
- July 6, 2015: Hadoop 2.7.1



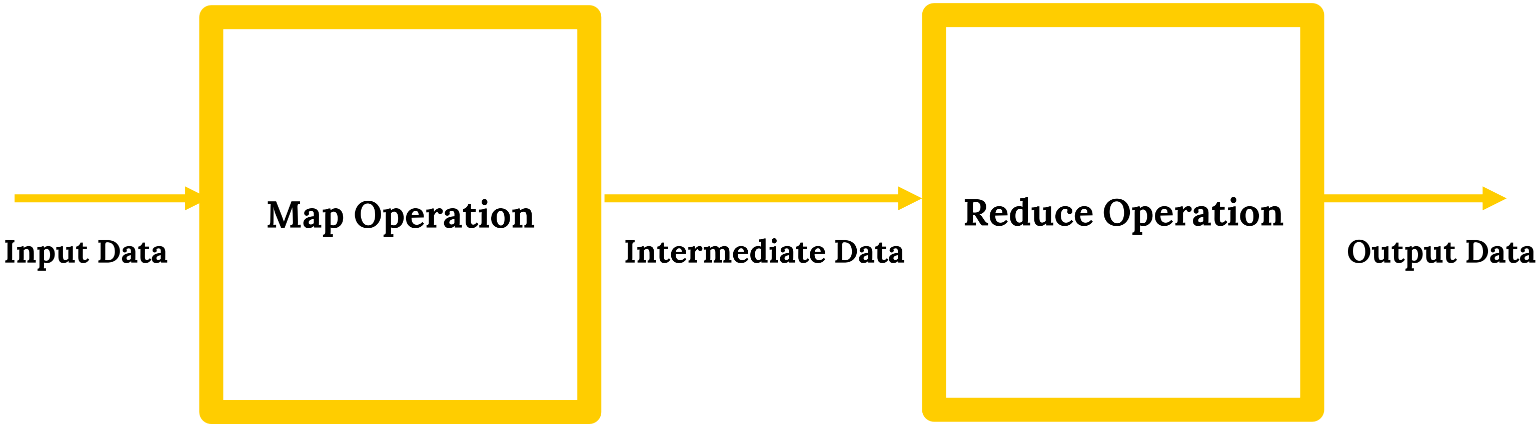
Distribution issues

- Communication and routing
  - which nodes should be involved?
  - what transport protocol should be used?
  - threads/events/connections management
  - remote execution of your processing code?
- Fault tolerance and fault detection
- Load balancing / partitioning of data
  - heterogeneity of nodes
  - skew in data
  - network topology
- Parallelization strategy
  - algorithmic issues of work splitting

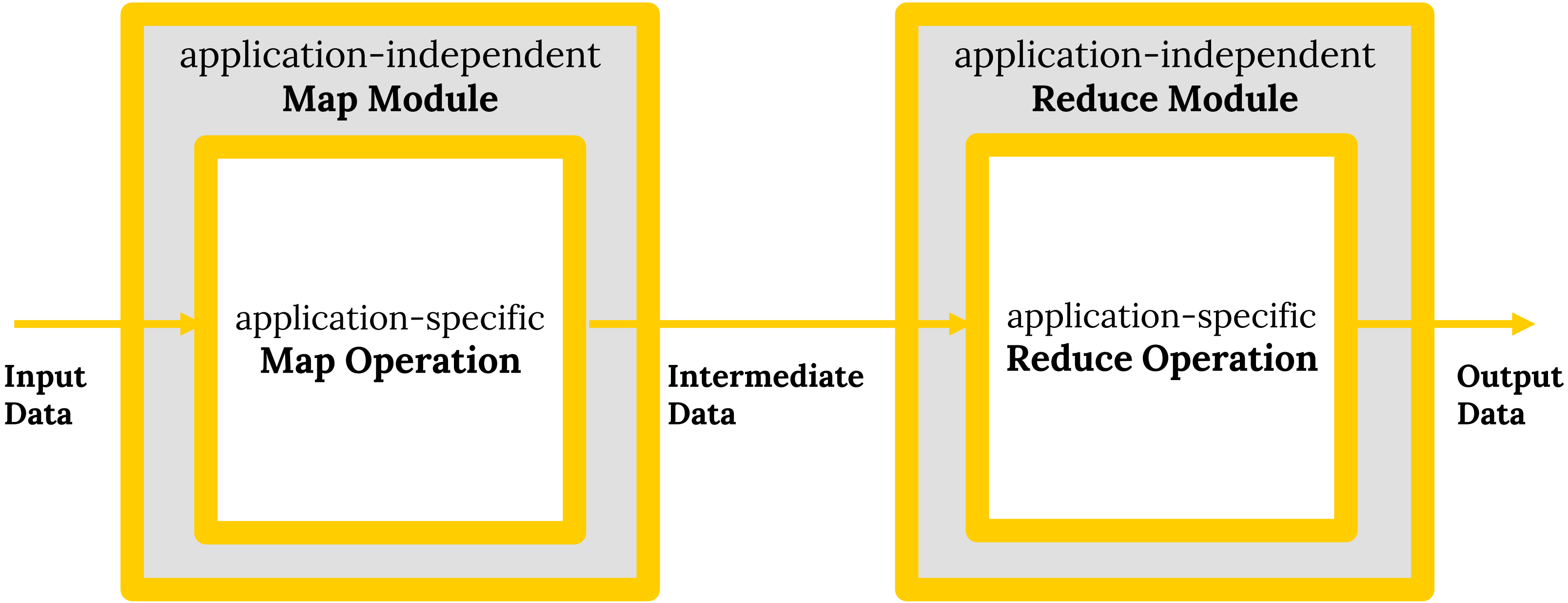
"without having to deal with failures, the rest of the support code just isn't so complicate"

S. Ghemawat

MapReduce model



MapReduce system



Original article

```
#include "mapreduce/mapreduce.h"

// User's map function
class WordCounter : public Mapper {
	public:
	virtual void Map(const MapInput& input) {
		const string& text = input.value();
		const int n = text.size();
		for (int i = 0; i < n; ) {
			// Skip past leading whitespace
			while ((i < n) && isspace(text[i]))
				i++;
			// Find word end
			int start = i;
			while ((i < n) && !isspace(text[i]))
				i++;
			if (start < i)
				Emit(text.substr(start,i-start),"1");
		}
	}
};

REGISTER_MAPPER(WordCounter);

// User's reduce function
class Adder : public Reducer {
	virtual void Reduce(ReduceInput* input) {
		// Iterate over all entries with the
		// same key and add the values
		int64 value = 0;
		while (!input->done()) {
			value += StringToInt(input->value());
			input->NextValue();
		}
		// Emit sum for input->key()
		Emit(IntToString(value));
	}
};

REGISTER_REDUCER(Adder);
```

Hadoop wiki

```
import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class WordCount {

public static class Map extends 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,
		Context context)
		throws IOException, InterruptedException {
		String line = value.toString();
		StringTokenizer tokenizer
			= new StringTokenizer(line);
		while (tokenizer.hasMoreTokens()) {
			word.set(tokenizer.nextToken());
			context.write(word, one);
		}
	}
}

public static class Reduce extends Reducer<Text, IntWritable,
	Text, IntWritable> {

	public void reduce(Text key, Iterable<IntWritable> values,
		Context context)
		throws IOException, InterruptedException {
		int sum = 0;
		for (IntWritable val : values) {
			sum += val.get();
		}
		context.write(key, new IntWritable(sum));
	}
}
```

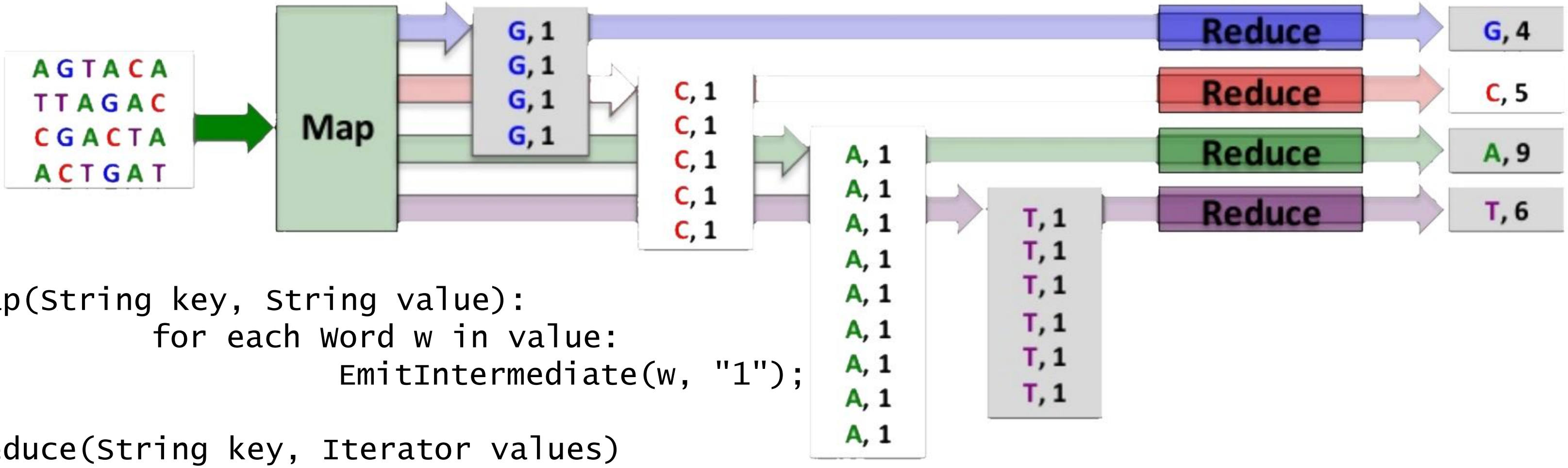
```cpp
int main(int argc, char** argv) {
        ParseCommandLineFlags(argc, argv);
        MapReduceSpecification spec;
        // Store list of input files into "spec"
        for (int i = 1; i < argc; i++) {
                MapReduceInput* input = spec.add_input();
                input->set_format("text");
                input->set_filepattern(argv[i]);
                input->set_mapper_class("WordCounter");
        }
        // Specify the output files:
        // /gfs/test/freq-00000-of-00100
        // /gfs/test/freq-00001-of-00100
        // ...
        MapReduceOutput* out = spec.output();
        out->set_filebase("/gfs/test/freq");
        out->set_num_tasks(100);
        out->set_format("text");
        out->set_reducer_class("Adder");
        // Optional: do partial sums within map
        // tasks to save network bandwidth
        out->set_combiner_class("Adder");
        // Tuning parameters: use at most 2000
        // machines and 100 MB of memory per task
        spec.set_machines(2000);
        spec.set_map_megabytes(100);
        spec.set_reduce_megabytes(100);
        // Now run it
        MapReduceResult result;
        if (!MapReduce(spec, &result)) abort();
        // Done: 'result' structure contains info
        // about counters, time taken, number of
        // machines used, etc.
        return 0;
}
```

Original article

```java
public static void main(String[] args) throws Exception {
        Configuration conf = new Configuration();

        Job job = new Job(conf, "wordcount");

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

        job.setMapperClass(Map.class);
        job.setReducerClass(Reduce.class);

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

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

        job.waitForCompletion(true);
}
}
```

Hadoop wiki

MapReduce model in practice



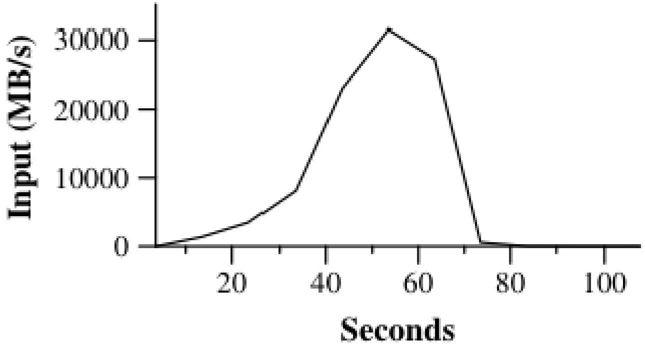

```
map(String key, String value):
        for each Word w in value:
                EmitIntermediate(w, "1");

reduce(String key, Iterator values)
        int result = 0;
        for each v in values:
                result += ParseInt(v);
        Emit(key, AsString(result));
```

map:(k1, v1)->[(k2, v2)]
reduce:(k2, [v2])->[(k3, v3)]

How long does it take to go through 1 TB?

Sequentially: 3 hours

MapReduce: startup overhead 70 seconds + computation 80 seconds



Environment

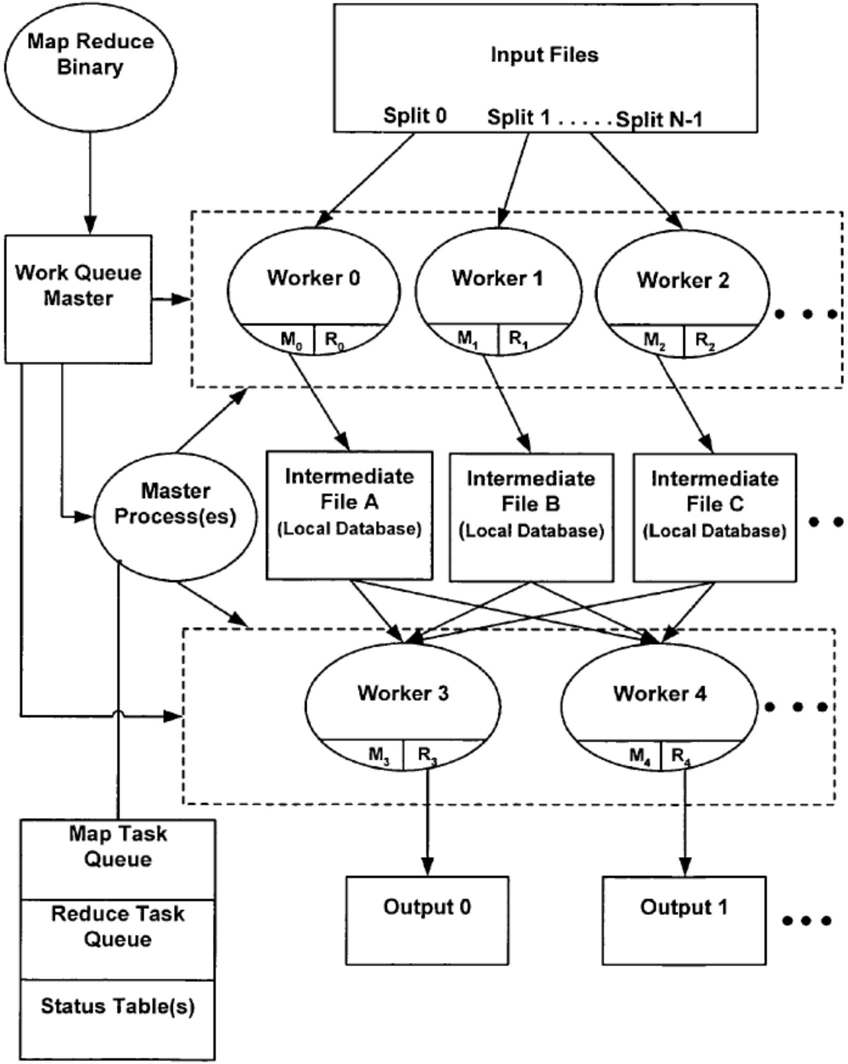
- 1800 Linux dual-processorx86 machines, 2-4 GB memory
- Fast Ethernet/Giga Ethernet
- Inexpensive IDE disks and a distributed Google File System

Take a walk through a Google data center

[Tianhe-2, #1 supercomputer](): 3,120,000 cores, 1,5 PB total memory

Execution diagram

- Master process is a task itself initiated by the WQM and is responsible for assigning all other tasks to worker processes
- Each worker invokes at least a map thread and a reduce thread
- If a worker fails, its tasks are reassigned to another worker process
- When WQM receives a job, it allocates the job to the master that calculates and requires M+R+1 processes to be allocated to the job
- WQM responds with the process allocation info (can result less processes) to the master that will manage the performance of the job
- Reduce tasks begin work when the master informs them that there are intermediate files ready
- Input data (files/DB/memory) are splitted in data blocks (16-64 MB) automatically or configurable
- The worker to which a map task has been assigned applies the map() operator to the respective input data block
- When the worker completes the task, it informs the master of the status
- Master informs workers where to find intermediate data and schedules their reading
- Workers (3 & 4) sort the intermediate key-value pairs, then merge (by applying reduce()) them and write to output



Workflow diagram

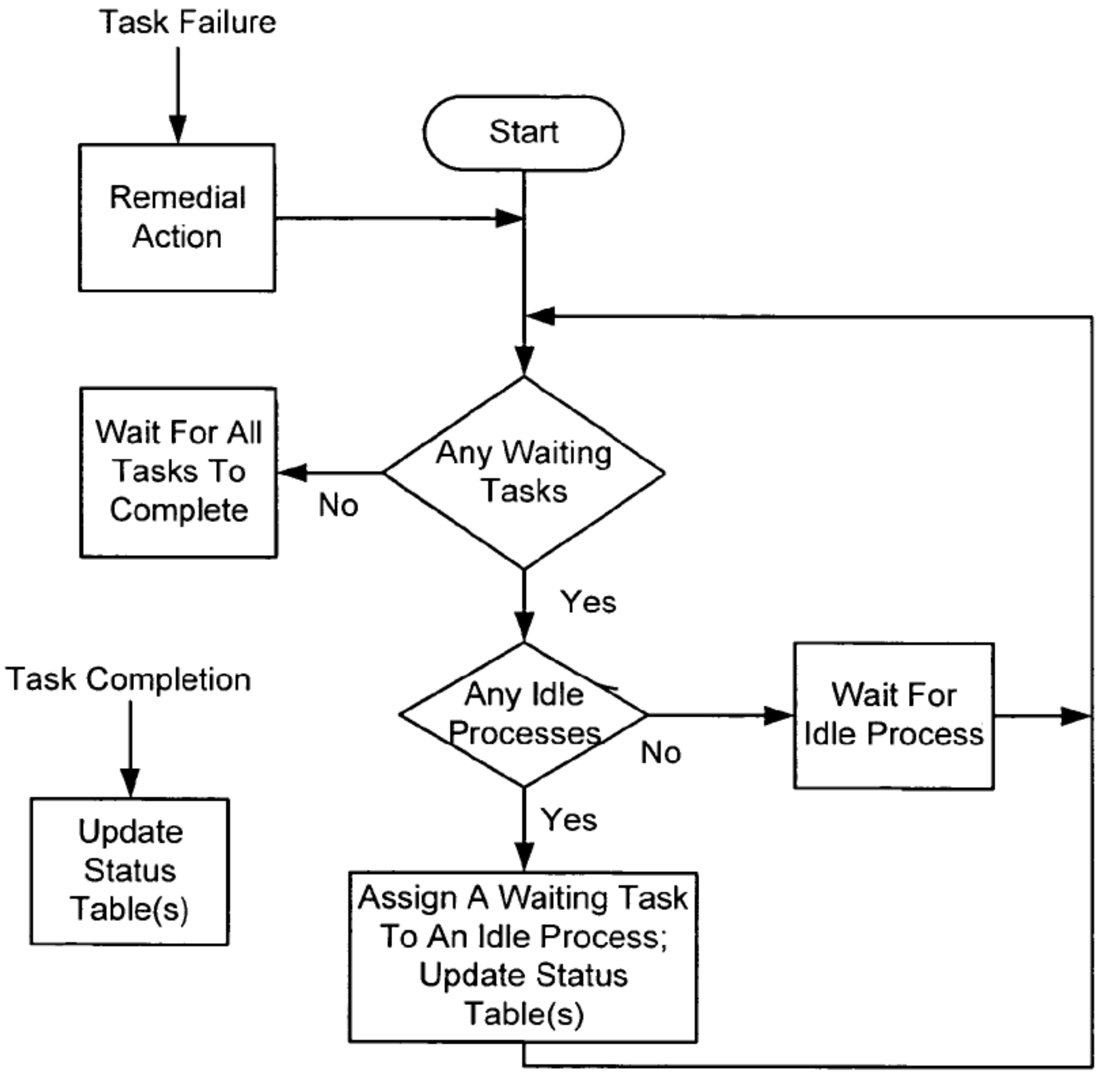
- When a process completes a task it informs WQM which updates the status tables
- When WQM discovers one process failed, it assign its tasks to a new process and updates the status tables

Task Status Table

- TaskID
- Status (InProgress, Waiting, Completed, Failed)
- ProcessID
- InputFiles (Input, Intermediate)
- OutputFiles

Process Status Table

- ProcessID
- Status (Idle, Busy, Failed)
- Location (CPU ID, etc.)
- Current (TaskID, WQM)



Questions from the audience @ original paper presentation

- Q: Wanted to know of any task that could not be handled using MapReduce?

  A: join operations could not be performed with the current model

- Q: Wondered how MapReduce differs from parallel databases?

  A: MapReduce is stored across a large number of machines as compared to parallel databases,

  the abstractions are fairly simple to use in MapReduce, and

  MapReduce also benefits greatly from locality optimizations

Bibliography

- J. Dean, S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", OSDI'04, Dec. 6, 2004
- J. Dean, S. Ghemawat, "System and method for efficient large-scale data processing", Patent US 7650331 B1, Jan. 19, 2010
- S. Ghemawat, J. Dean, J. Zhao, M. Austern, A. Spector, "Google Technology RoundTable: Map Reduce", Aug. 21, 2008 – Youtube
- P. Mahadevan, "OSDI'04 Conference Reports", ;LOGIN: Vol. 30, No. 2, Apr. 2005, p. 61
- R. Jacotin, "Lecture: The Google MapReduce", SlideShare, October 3, 2014

Thanks!

Any *questions* ?

You can find me at