Presentation 2

MapReduce: simplified data processing on large clusters

Dean, Jeffrey, and Sanjay Ghemawat

Communications of the ACM 51, no. 1 (2008): 107-113

Pig latin: a not-so-foreign language for data processing

Olston, Christopher, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins

In Proceedings of the 2008 ACM SIGMOD international conference on Management of data, pp. 1099-1110. ACM, 2008.

John Berlin

October 6, 2016

Old Dominion University
Introduction to Information Retrieval
CS734/834

In The Beginning There Was Big Data

- **What Is It?**
  - Crawled Documents
  - Log Files
  - Databases
- **How Big Is Big Data?**
  - More Data Than A Single Computer Can Handle
  - > 1TB
- **How Do We Process It?**
  - 100 Computers with 1/100 Of The Data?
  - Distribute It All Over The Network?
  - How Are We Going To Coordinate Our Machines?
  - Do Our Programmers Need To Learn A New Language/Tool For This?

"Data: My programming may be inadequate to the task.
Counselor Troi: We're all more than the sum of our parts, Data. You'll have to be more than the sum of your programming."
In Theory, TNG

MapReduce: Simplified Data Processing on Large Clusters

Dean & Ghemawat’s Contribution

- **Created A Framework That Abstracted Away The Complexity From**
  - Parallelization Of The Computation
  - Distribution Of The Data
- **“Devised” A Programming Model To Harness The Framework**
  - Write Smaller Code Modules → Quick Big Data Processing

The MapReduce Programming Model

- **Computation By Two Functions**
  - Map: $(K_1,V_1) \rightarrow List(K_2,V_2)$
  - Reduce: $(K_2, List(V_2)) \rightarrow List(K_3,V_3)$
- **Functions Bound By A Contract**
  - Take A Single Input Format
  - Produce Single Output Format
- **Need Only These Two Functions To Start Map And Reducing**

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

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

Word Count Pseudo Code
section 2.1, p2

MapReduce Is Really Functional Programming λ

- **Pure Functions**
  - Contract Ensures This
  - Input Is Not Modified
  - Produces Same Output Given The Same Input
- **Higher Order Functions**
  - Output Of One Function Is The Input Of The Next
  - Composable: Reduce(Map(Reduce(Map(K_1,V_1))))

    Reduce(Map(Map(K_1,V_1)))
- **Map Only Transforms**
  - Given Input Produces Zero Or More K,V Pairs
- **Reduce Only Accumulates**
  - Combines A List Of V For a Given K

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

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

MapReduce Framework

- **Input Split Into M Splits**
  - Split Size Typically 16-64MB
- **Map Function Distributed**
  - Splits M = Number Of Mappers
- **Reduce Function Distributed**
  - Distribution Based On User Configured Number R
  - Feed Data Based On Partitioning
  - Partition Function Typically: Hash(key) Mod R

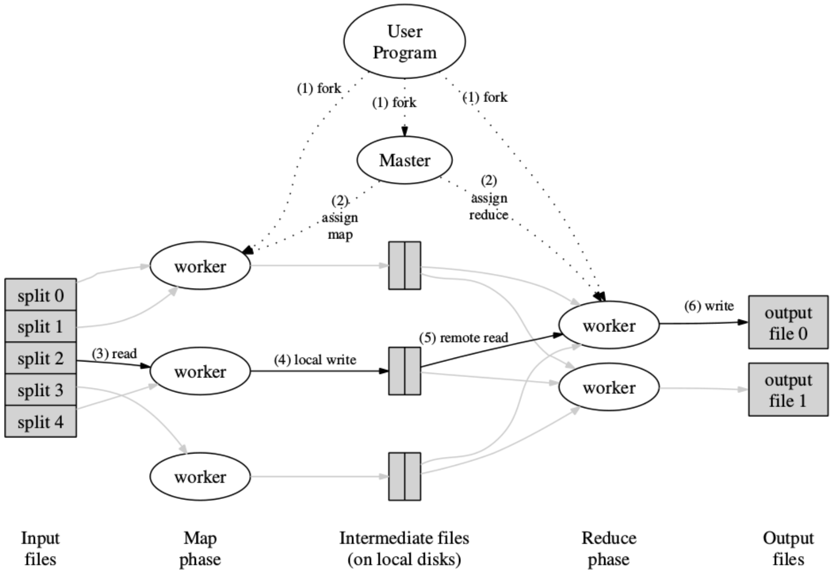


Figure 1: Execution overview

Behind The Scenes Framework Execution

- **Data Lives On The Worker Machines**
  - Split Up On Start Of A MR Job
- **Map Tasks**
  - Master Schedules Map Tasks
  - On End Are Rescheduled If Machine Still Has Unprocessed Data
  - On Failure Master Starts Another On Same Machine Or Machine Close To The Data i.e Same Local Network Switch
- **Reduce Tasks**
  - Constantly Work As Input Is Sorted And Uniquely Keyed
  - Reads Data From Mapper File System Via Network
  - Writes To Final Output Destination

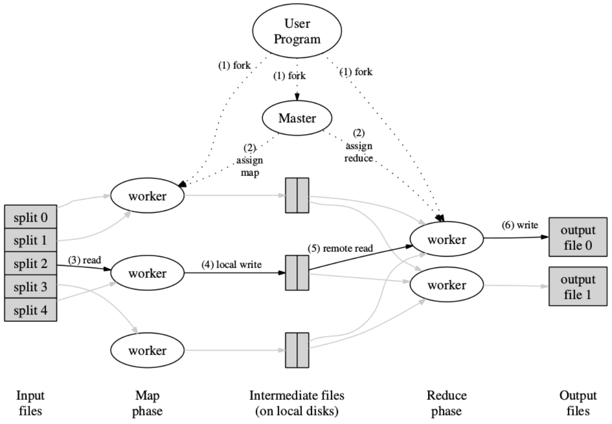


Figure 1: Execution overview

What Does A MapReduce Program Look Like

```cpp
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);
```

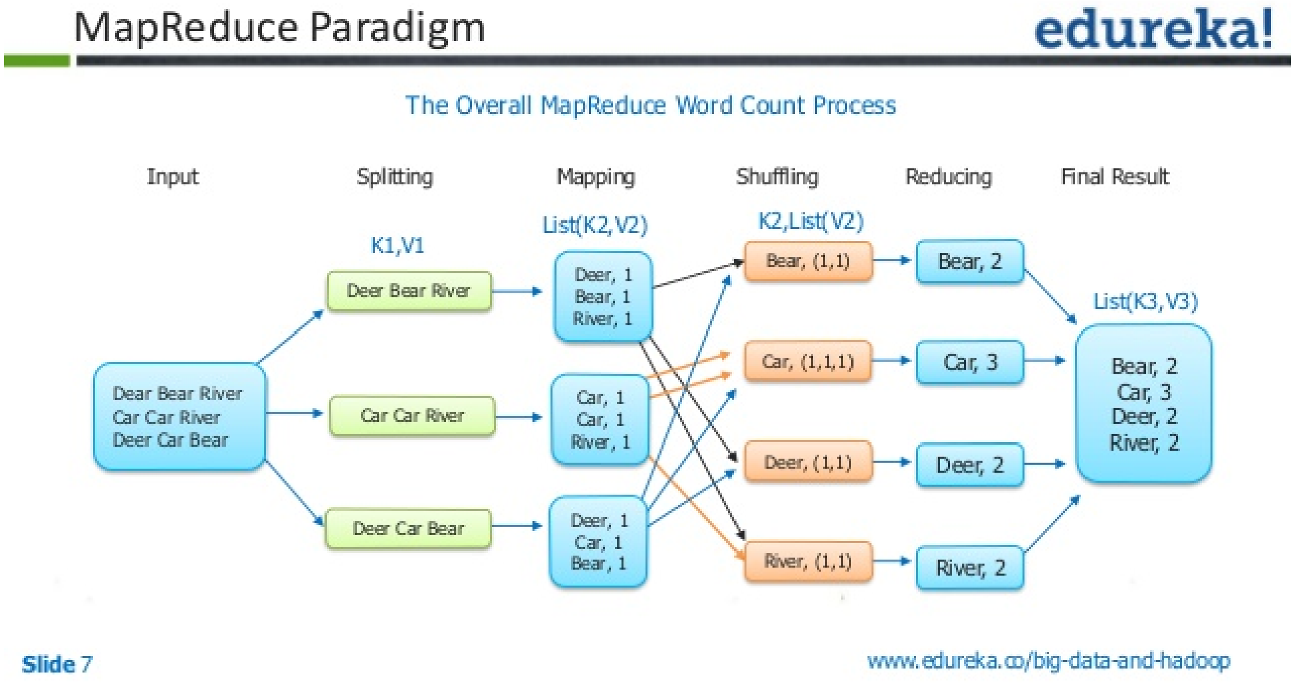
```cpp
// User's reduce function
class Adder : public Reducer {
  virtual void Reduce(ReduceInput* input) {
    // Iterate over all entries with the
    // same key and add the values
    int64 value = 0;
    while (!input->done()) {
      value += StringToInt(input->value());
      input->NextValue();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

```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;
}
```

Work Frequency Appendix A Page 13

In Action

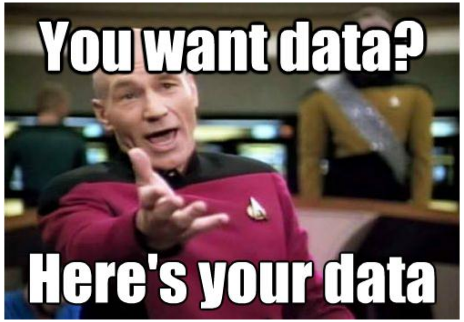


http://image.slidesharecdn.com/hadoopmr-150613152227-lva1-app6891/95/xml-parsing-with-map-reduce-7-638.jpg?cb=1434209024

Google Has Solved Our Problems Again

- **MapReduce Requires Little Knowledge Of:**
  - Parallel Programming
  - Distributed Systems
- **MapReduces Framework**
  - Scales Well On Large Clusters
  - Easily Adoptable To Many Real World Problem

- **BUT Not Open Source**
- **The Google Viewpoint On It**
  - We Designed It, Built It For Ourselves (In C++, For GFS)
  - We Told You How To Do It
  - Now Build it Yourself.



Some Background For The Next Paper: Hadoop

- **Communities Java Implementation Of Googles MapReduce**
  - Good Guy Apache
- **Full Implementation**
  - Hadoop File System
  - Programming Model
  - Executor

Code I Wrote For CS495 →

Spring 2014

```java
/*
  this class takes line and emites only
  the productId and count of 1
*/
public class GetIdsMapper extends  Mapper<LongWritable, Text, Text, IntWritable> {
  @Override
  public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException

    String line = value.toString();
    StringTokenizer tokenizer = new StringTokenizer(line);

    while (tokenizer.hasMoreTokens()) {
      String token = tokenizer.nextToken();
      if(token.equals("product/productId:")){
        context.write(new Text(tokenizer.nextToken()),new IntWritable(1));
      }
    }
  }

}
```

```java
public class CountIdsReducer extends Reducer<Text, IntWritable,Text, IntWritable>
  @Override
  public void reduce (Text key, Iterable<IntWritable> values, Context context) thr
    int sum = 0;
    for(IntWritable val : values)
      sum += val.get();
    context.write(key,new IntWritable(sum));
  }
}
```

But Not Everyone Wants To Use Java

- **Making Hadoop Do The Dew Can Get Tricky**
- **Must Make Your Types**
  - Only Primitive Types Supported i.e IntWritable
  - Gotta Write Your Own Complex Key Value Types
- **No Easy Way To Do**
  - Map/Reduce Chaining
  - Secondary Sorting
- **Projects Are Large →**

Pig Latin: A Not-So-Foreign Language for Data Processing

Olston, Reed, Srrivastavas, Kumar, Tomkins Contribution

- Created A Data Processing Environment Back By Hadoop
- Made Processing Big Data SQL Like
- Easier Nonstandard Operations On The Data

Motivation Behind Pig

- Addresses The Short Comings Of Hadoop
- Joins, Multi-Stage Processing Was A Pain
- Conditional Selection
- Custom Code Everywhere

Developers Understand SQL Like Expression Of The Computation Better Than Java

https://www.dynadot.com/community/image?image_id=1169

Pig A Dataflow Language

- **Nested Data Model**
  - term_info: (termId, termString), position: (termId,documentId,pos)
  - Much Like Databases
  - Captures The Flow Of The Data
- **UDFs: User Defined Functions**
  - Express The Computation Not Implement It

```
groups = GROUP urls BY category;
output = FOREACH groups GENERATE
              category, top10(urls);
```

Pig’s Data Model

- **Atom: Simple Atomic Value Such As A String, “John”**
- **Tuple: Sequence Of Fields Of Any Datatype, (“John”,”Berlin”)**
- **Bag: Collection Of Tuples**

$$\left\{ \begin{array}{c} (\text{‘alice’},\ \text{‘lakers’}) \\ (\text{‘alice’},\ (\text{‘iPod’},\ \text{‘apple’})) \end{array} \right\}$$

- **Map: Collection Of Data Items**

$$\left[\begin{array}{c} \text{‘fan of’} \to \left\{ \begin{array}{c} (\text{‘lakers’}) \\ (\text{‘iPod’}) \end{array} \right\} \\ \text{‘age’} \to 20 \end{array} \right]$$

Pig Latin: Igspay Omputationcay Anguagelay

```
expanded_queries = FOREACH queries GENERATE
                   userId, expandQuery(queryString);
```

```
real_queries =
           FILTER queries BY NOT isBot(userId);
```

```
join_result  =  JOIN results BY queryString,
                revenue BY queryString;
```

```
grouped_revenue = GROUP revenue BY queryString;
query_revenues = FOREACH grouped_revenue{
                   top_slot = FILTER revenue BY
                              adSlot eq 'top';
                   GENERATE queryString,
                            SUM(top_slot.amount),
                            SUM(revenue.amount);
                 };
```

```
grouped_revenue = GROUP revenue BY queryString;
query_revenues = FOREACH grouped_revenue GENERATE
             queryString,
             SUM(revenue.amount) AS totalRevenue;
```

```
grouped_data  =  COGROUP results BY queryString,
                         revenue BY queryString;
```

Examples Of How Pig Latin Is Used To Do
Map Reduce Operations.
Section 3.3-3.7, p5-7

How Does This Work With Hadoop/MapReduce

- **Pig Latin Script → Logical Plan**
  - Data Model and Operations Have Their Own
  - Each Step Is Checked For Validity
  - Assignment Constructs Plan From Input

```
join_result  =  JOIN results BY queryString,
                revenue BY queryString;
```

- **Logical Plan → MapReduce Compilation**

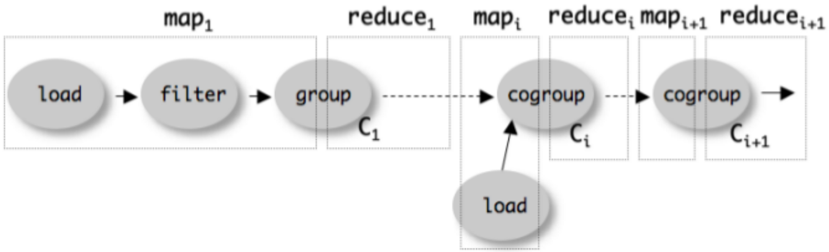


Figure 3: Map-reduce compilation of Pig Latin.

Conclusion

- **MapReduce Provided The Framework For**
  - Efficient Big Data Processing
  - Programming Model To Do It
  - Framework For Distributing The Computation
- **Pig**
  - Abstracted Away The Complexity Of The Computation
  - Made It Easier To Write MapReduce Jobs