CS 555: DISTRIBUTED SYSTEMS [MAPREDUCE]

Shrideep Pallickara
Computer Science
Colorado State University

Frequently asked questions from the previous class survey

Topics covered in this lecture

- MapReduce

MAPREDUCE

MapReduce: Topics that we will cover

- Why?
- What it is and what it is not?
- The core framework and original Google paper
- Development of simple programs using Hadoop
  - The dominant MapReduce implementation

MapReduce

- It's a **framework** for processing data residing on a large number of computers
- Very powerful framework
  - Excellent for some problems
  - Challenging or not applicable in other classes of problems

What is MapReduce?

- More a framework than a tool
- You are required to **fit** (some folks shoehorn it) your solution into the MapReduce framework
- MapReduce is not a feature, but rather a **constraint**

What does this constraint mean?

- It makes problem solving easier and harder
- Clear boundaries for what you can and cannot do
  - You actually need to consider **fewer options** than what you are used to
- But solving problems with constraints requires planning and a change in your thinking

But what does this get us?

- Tradeoff of being confined to the MapReduce framework?
  - Ability to process data on a large number of computers
  - But, more importantly, **without having to worry about** concurrency, scale, fault tolerance, and robustness

A challenge in writing MapReduce programs

- **Design**!
  - Good programmers can produce bad software due to poor design
  - Good programmers can produce bad MapReduce algorithms
- Only in this case your **mistakes will be amplified**
  - Your job may be distributed on 100s or 1000s of machines and operating on a Petabyte of data

MapReduce: Origins of the design

- Process crawled data and logs of web requests
- Several computations work on this raw data to compute derived data
  - Inverted indices
  - Representation of graph structure of web documents
  - Pages crawled per host
  - Most frequent queries in a day …

Most computations are conceptually straightforward

- But data is large
- Computations must be **scalable**
  - Distributed across thousands of machines
  - To complete in a reasonable amount of time

Complexity of managing distributed computations can …

- Obscure **simplicity** of original computation
- Contributing factors:
  1. How to *parallelize* computation
  2. Distribute the *data*
  3. Handle *failures*

MapReduce was developed to cope with this complexity

- Express simple computations
- Hide messy details of
  - Parallelization
  - Data distribution
  - Fault tolerance
  - Load balancing

MapReduce

- Programming model
- Associated implementation for
  - Processing & Generating large data sets

Programming model

- Computation takes a set of **input** *key/value* pairs
- Produces a set of **output** *key/value* pairs
- Express the computation as two functions:
  - Map
  - Reduce

Map

- Takes an input pair
- Produces a set of intermediate key/value pairs

MapReduce library

- **Groups** all intermediate values with the same intermediate key
- **Passes** them to the Reduce function

Reduce function

- Accepts intermediate *key* I and
  - Set of *values* for that *key*
- **Merge** these *values* together to get
  - Smaller set of *values*

Counting number occurrences of each word in a large collection of documents

```
map (String key, String value)
     //key: document name
     //value: document contents

   for each word w in value
      EmitIntermediate(w, "1")
```


Counting number occurrences of each word in a large collection of documents

```
reduce (String key, Iterator values)
   //key: a word
  //value: a list of counts

  int result = 0;
  for each v in values
     result += ParseInt(v);
  Emit(AsString(result));
```

Sums together all counts emitted for a particular word

MapReduce specification object contains

- Names of
  - Input
  - Output
- Tuning parameters

Map and reduce functions have associated types drawn from different domains

```
map(k1, v1)            → list(k2, v2)

reduce(k2, list(v2)) → list(v2)
```


What's passed to-and-from user-defined functions

- Strings
- User code converts between
  - String
  - Appropriate types

Programs expressed as MapReduce computations: Distributed Grep

- Map
  - Emit line if it matches specified pattern
- Reduce
  - Just copy intermediate data to the output

Term-Vector per Host

- Summarizes important terms that occur in a set of documents <word, frequency>
- Map
  - Emit <hostname, term vector>
  - For each input document
- Reduce function
  - Has all per-document vectors for a given host
  - Add term vectors; discard away infrequent terms
    - <hostname, term vector>

IMPLEMENTATION OF THE RUNTIME

Implementation

- Machines are **commodity** machines
- **GFS** is used to manage the data stored on the disks

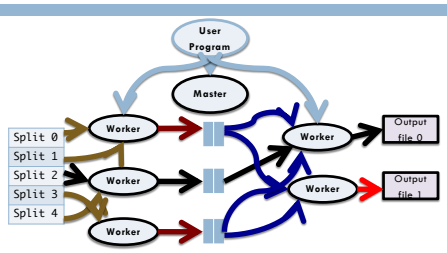
Execution Overview – Part I

- *Maps* distributed across multiple machines
- Automatic partitioning of data into M splits
- Splits processed **concurrently** on different machines

Execution Overview – Part II

- Partition *intermediate* key space into R pieces
- E.g. hash(key) **mod** R
- User specified parameters
  - **Partitioning** function
  - **Number** of partitions (R)

Execution Overview



Execution Overview: Step I
The MapReduce library

- Splits input files into **M** pieces
  - 16-64 MB per piece
- Starts up copies of the program on a cluster of machines

Execution Overview: Step II
Program copies

- One of the copies is a **Master**
- There are **M** map tasks and **R** reduce tasks to assign
- Master
  - Picks *idle* workers
  - Assigns each worker a map or reduce task

Execution Overview: Step III
Workers that are assigned a map task

- Read contents of their input split
- Parses <*key, value*> pairs out of input data
- Pass each pair to user-defined *Map* function
- Intermediate <*key, value*> pairs from *Maps*
  - Buffered in Memory

Execution Overview: Step IV
Writing to disk

- Periodically, **buffered pairs** are written to disk
- These writes are partitioned
  - By the partitioning function
- **Locations** of buffered pairs on local disk
  - *Reported* to back to Master
  - Master *forwards* these locations to reduce workers

Execution Overview: Step V
Reading Intermediate data

- Master notifies *Reduce* worker about locations
- Reduce worker reads buffered data from the **local disks** of *Maps*
- Read *all* intermediate data; sort by intermediate key
  - All occurrences of same key grouped together
  - Many different keys map to the same *Reduce* task

Execution Overview: Step VI
Processing data at the Reduce worker

- Iterate over sorted intermediate data
- For each unique key pass
  - *Key* + set of *intermediate values* to Reduce function
- Output of Reduce function is appended
  - To output file of reduce partition

Execution Overview: Step VII
Waking up the user

- After all Map & Reduce tasks have been completed
- Control returns to the user code

TASK GRANULARITY

Task Granularity

- Subdivide map phase into **M** pieces
- Subdivide reduce phase into **R** pieces
- **M, R** >> number of worker machines
- Each worker performing many different tasks
  - Improves **dynamic load balancing**
  - Speeds up **recovery** during failures

Master Data Structures

- For each Map and Reduce task
  - **State**: {*idle*, *in-progress*, *completed*}
  - Worker **machine** identity
- For each completed Map task store
  - **Location** and **sizes** of **R** intermediate file regions
- Information pushed incrementally to *in-progress* Reduce tasks

Practical bounds on how large M and R can be

- Master must make $O(M + R)$ scheduling decisions
- Keep $O(MR)$ state in memory

The contents of this slide-set are based on the following references

- **JEFFREY DEAN** and **SANJAY GHEMAWAT**: *MapReduce: Simplified Data Processing on Large Clusters.* OSDI 2004: 137-150
- *MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems.* 1st Edition. Donald Miner and Adam Shook. O'Reilly Media ISBN: 978-1449327170. [Chapter 1]