Google

[MapReduce and PageRank]

Please do not forget to download the illustration page:

lec24-mapReduce-Illustrated.pdf

Map Reduce

Motivation:

- Large-scale data processing
- Distributed computing: want to use thousands of machines to do the job, but don't want to micro-manage

MapReduce:

- Automatically distribute jobs across machines
- Hide internal management: fault tolerance (nice failure semantic: simply retry), scheduling, monitoring and status updates
- Implemented on GFS
    - Distributed computing uses files to do the job
    - Hence, distributed file system is useful for distributed computing

Map and Reduce functions:

- Initially, the input values must be a set of key-value pairs
    - In real life, a job works on millions of key-value pairs
    - This key-value pairs is set up by the clients and they are stored in a GFS file
    - Hence, the job will read chunks to get the data
- Map(key, value) function
    - You implement your Map(key, value) function
    - The server will take the key-value pairs and execute your Map() function
    - Your Map() function should generate a set of intermediate key-value pairs
- Reduce(key, value**s**) function
    - You implement your Reduce(key, values) function
    - The server will take the **intermediate** key-value pairs generated by your Map(), then merge all pairs with same key to get key-value**s** pairs
    - Then the server will execute your Reduce() function on these intermediate key-value pairs

Example: Simple Math

Given a set of integers, compute the sum of their square values.
e.g. 1 2 3 4 → 1 + 4 + 9 + 16 → 30

```
Map(key, value) {
   Generate (1, value*value)
}

Reduce(key, values) {
   Int sum = 0;
   For (all values)
      sum += values[i];
}
```

Count words in all web pages

Inputs consists of url-content pairs.

```
Map(key, value) {
   // key is url
   // value is the content of the url
   For each word W in the content
      Generate(W, 1);
}

Reduce(key, values) {
   // key is word (W)
   // values are basically all 1s
   Sum = Sum all 1s in values
   // generate word-count pairs
    Generate (key,sum);
}
```

(Note: like a producer-consumer problem; map produces pairs, reduce consumes pairs)

Grep

Input consists of (url+offset, single line).
(Note: you as the client needs to setup such inputs and store it in a GFS file)

```
Map(key, value) {
   // key = url + offset
   // value = single line
```

```
   str = key concat value
   If value matches regexp
      Generate(str, 1)
}

Reduce(key, value) {
   // key = url+offset, line
   // values = all 1s (the sum is the total count)
   No need to do anything, just regenerate the line
   Generate(key, “”);
}
```

Reverse web-link graph

Go to google advanced search:
"find pages that link to the page:" cnn.com

```
Map(key, value) {
   // key = url
   // value = content
   For each url, linking to target
      Generate(output target, source);
}

Reduce(key, values) {
   // key = target url
   // values = all urls that point to the target url
   For each values
      Generate(key, values[i]);
}
```

Effectiveness of MapReduce

- In april 2003, one google software uses map-reduce
- In sep 2004, almost 1000 google software uses map-reduce
- (almost exponential growth)

Distributed Execution

- GFS Cluster
    - Google has thousands of machine that can used to run jobs
    - Again, jobs must be rewritten in map-reduce way
- Partition input key/value pairs
    - If inputs have not been stored in GFS cluster, create a file
    - And GFS will save the file into chunks and distribute chunks across machines
- Run Map() tasks in parallel

- After all maps are complete, servers consolidate all generated values for each unique generated key
- Next, servers will partition these intermediate results (again into GFS chunks), and run reduce() in parallel

Job processing:

1. client submits "grep" job to job tracker, indicating code and input files
2. jobtracker breaks input file into k chunks (e.g. 6 chunks), assign 6 machines to work on the 6 chunks (called task-trackers)
3. after map is done, tasktrackers exchange map-output to build reduce() keyspace
4. jobtracker breaks reduce() keyspace into m chunks (e.g. 4)
5. reduce() output may go to NDFS

Pipelining:

- Problem: we send a task to a machine and waiting for the output from that machine might be slow. Hence break the task into several "maps". E.g. 1 map has 100,000 key-value pairs. For each map that is done, we can assign another machine to work the reduce.
- Good things: we can see partial results faster
- Example: break big thing to map1, 2, and 3. but only assign 2 servers. Worker A and B. Worker A does 1 and 3, and B does 2. When A1 finishes, another worker C can reduce the output of A1.

Fault-tolerance:

- Detect failure via periodic heartbeats
- If map() or reduce() fails, simply reexecute()

Refinement: Redundant execution:

- If there is a slow machine/component (due to high load, or bad disks, or other bugs), the final result is delayed
- Solution: create redundant execution, especially for map/reduce tasks that have not been completed.
    - Hence, this shows another power of MapReduce, a very powerful abstraction for distributed computing

Conclusion:

- Rewrote problem in map-reduce format
    - What's the whole idea? Common format for inputs: key-value pairs
- MapReduce is an abstraction for distributed computing
    - Abstraction → simplify large-scale computations
- "Fun"
    - Focus on problem (rather than implementation)
    - Let the internal mechanisms deal with the details

Page Rank

Search problem:

- Search for "flu"
- Thousands of matching pages.
- How to put most relevant pages at the top hits?
- In other words, how to make people more effective by giving relevant hits

Solution 1 (In the beginning):

- In the old days: put the word "flu" 10,000 times in your page
- Then your page will appear as one of the top results
- Hence, lots of malicious people out there who knows this simple algorithm, and "game" the search algorithm

Solution 2 (Simple PageRank, 1998):

- Utilize the linked structure of hypertext
- Links counting as votes for the page
- In other words, look which page is famous/important?
- How to define "famous/important", by counting how many links to the page
    - If page is pointed once, rank = 1
    - If page is pointed twice, rank = 2
- "democratic nature"
- Problem:
    - Once, people know this algorithm, they can game the system
    - I can create thousands of pages that point to a page that I want to advertise
    - And that's what people did!

Solution 3 (Weighted PageRank):

- Analyze the page that votes for a page
- If page X links to page Y, is page X itself important/famous?
- Rank of a page is the sum of ranks of all pages that link to the page
- Rank of a page is high if all the voters are also important
- "not fully democratic"
- Problem:
    - Commercial interest can still build lots of backlinks

Solution 4 (Add Host Name Detection)

- See if link is accessed from the same host name
- For example: myserver.com/file.html points to myserver.com/access.html
- If that's the case, reduce the rank (because this is internal link)
- But if a page is pointed by other host, increase the rank
- For example: myserver.com/file.html points to mybook.com/access.html
- Problem:
    - Still can game the search engine

- Ex: A company can create lots of host names (mybook.com and myserver.com belong to the same company)

Solution 5 (Add Host-IP Detection)

- If a same company, usually host names are host on the same machine
- Ex: although myserver.com/file.html points to mybook.com/access.html, myserver.com and mybook.com are host in the same machine (e.g. with IP address 128.105.167.124)
- Hence, use that information.
- Specifically, if the first three octets (e.g. 128.105.167) of two host names are the same, then reduce rank
- If not, increase rank
- Problem:
  - This algorithm solves commercial interests that come from small companies. Small companies cannot have lots of machines to host their webpage. But still, big companies can buy lots of machines.

Solution 6 (Hilltop, 2003)

- Google bought Hilltop patent
- We know that there are expert (or authoritative) pages out there (e.g. cnn.com, espn.com)
- Hence, trust expert pages
- If a page is pointed by expert pages, give higher ranks
- Problems:
  - Need to know expert pages concerning each topic
  - Hard because need to annotate expert pages manually

Solution 7 (Add User's Selections)

- Useful/important pages are usually selected by search users
- Hence, why not incorporate their selections to increase the rank?

Recap: PageRank and MapReduce

- To create ranks, we need a **program** to **scan** all web pages, and the algorithm
- Hence, the inputs to MapReduce function for this program are all the pages
  - Key = url, value = content
- These web pages are already stored in the GFS servers
- Hence, this program can be run in parallel
- Storage affinity
  - Ex: to scan cnn.com, this program is run on a machine that stores cnn.com, etc.
  - Hence, no need to send cnn.com pages across machines

Summary:

- Lots of unanswered problems
  - Ex: If you are a specific researcher. Mostly, the pages you are looking is unique. Hence, not always appear at the top.

    - Or find a specific setup problem. If there are only 100 people who face the problem, and the solution is available on the web, your search results might not give what you want directly.
- Lots of future in search
    - How to make search engine “do your job”
    - I want to do X, and lots of explanations are scattered, how to combine them?
- Final words
    - The foundation of the search algorithm is theory (if you like theory, this is a promising field)
    - To execute the theory efficiently, we need to know systems-design (i.e. PageRank will not be attractive if there is no system solutions like GFS and MapReduce)