

Database Systems CSE 414

Lecture 19: MapReduce (Ch. 20.2)

CSE 414 - Fall 2017

1

Announcements

- HW5 is due tomorrow 11pm
- HW6 is posted and due Nov. 27 11pm
 - Section Thursday on setting up Spark on AWS
 - Create your AWS account **before arriving**
 - Follow the first part of the Spark setup instructions (“Setting up an AWS account”) to get credits for free use
 - <https://courses.cs.washington.edu/courses/cse414/17au/spark/spark-setup.html>
 - note that this **may take a while** to process
 - **Remember to terminate cluster when not in use!!!**
 - **Otherwise, you will be charged lots of \$\$\$\$**

CSE 414 - Fall 2017

2

Optional Reading

- Original paper:
<https://www.usenix.org/legacy/events/osdi04/tech/dean.html>
- Rebuttal to a comparison with parallel DBs:
<http://dl.acm.org/citation.cfm?doid=1629175.1629198>
- Chapter 2 (Sections 1, 2, 3 only) of Mining of Massive Datasets, by Rajaraman and Ullman
<http://i.stanford.edu/~ullman/mmds.html>

CSE 414 - Fall 2017

3

Distributed File System (DFS)

- For very large files: TBs, PBs
- Each file is partitioned into *chunks*, typically 64MB
- Each chunk is replicated several times (≥ 3), on different racks, for fault tolerance
- Implementations:
 - Google’s DFS: **GFS**, proprietary
 - Hadoop’s DFS: **HDFS**, open source

CSE 414 - Fall 2017

4


MapReduce

- Google: paper published 2004
- Free variant: Hadoop
- MapReduce = high-level programming model and implementation for large-scale parallel data processing

CSE 414 - Fall 2017

5

MapReduce Process

- Read a lot of data (records)
- **Map**: extract info you want from each record
- Shuffle and Sort 
- **Reduce**: aggregate, summarize, filter, transform
- Write the results

Paradigm stays the same,
change map and reduce
functions for different problems

CSE 414 - Fall 2017

6

slide source: Jeff Dean

Data Model

Files!

A file = a bag of **(key, value)** pairs

A MapReduce program:

- Input: a bag of **(inputkey, value)** pairs
- Output: a bag of **(outputkey, value)** pairs

CSE 414 - Fall 2017

7

Step 1: the **MAP** Phase

User provides the **MAP**-function:

- Input: **(input key, value)**
- Output: **bag of (intermediate key, value)**

System applies the map function in parallel to all **(input key, value)** pairs in the input file

CSE 414 - Fall 2017

8

Step 2: the **REDUCE** Phase

User provides the **REDUCE** function:

- Input: **(intermediate key, bag of values)**
- Output: bag of output **(key, value)** pairs

System groups all pairs with the same intermediate key, and passes the bag of values to the REDUCE function

CSE 414 - Fall 2017

9

Example

- Counting the number of occurrences of each word in a large collection of documents
- Each Document
 - The **key** = document id (**did**)
 - The **value** = set of words (**word**)

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

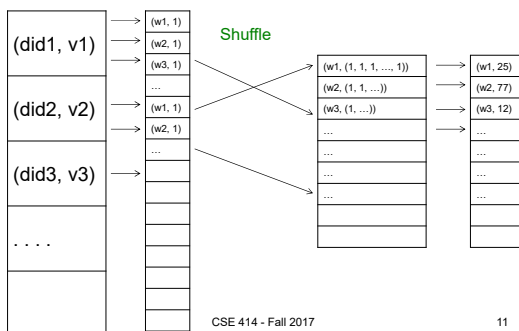
```
reduce(String key, Iterator values):  
  // key: a word  
  // values: a list of counts  
  int result = 0;  
  for each v in values:  
    result += 1;  
  Emit(key, AsString(result));
```

CSE 414 - Fall 2017

10

MAP

REDUCE



CSE 414 - Fall 2017

11

Jobs & Tasks

- A **MapReduce Job**
 - One single “query”, e.g. count the words in all docs
 - More complex queries may consist of multiple jobs
- A **Map Task**, or a **Reduce Task**
 - A group of instantiations of the map-, or reduce-function, which are scheduled on a single worker

CSE 414 - Fall 2017

12

Workers

- A **worker** is a process that executes one task at a time
- Typically there is one worker per processor, hence 4 or 8 per node

CSE 414 - Fall 2017

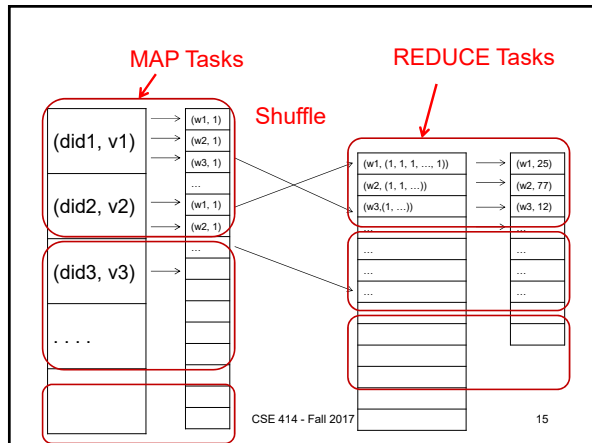
13

Fault Tolerance

- If one server fails once every year...
... then a job with 10,000 servers will fail in less than one hour
- MapReduce handles fault tolerance by writing intermediate files to disk:
 - Mappers write file to local disk
 - Reducers read the files (=reshuffling); if the server fails, the reduce task is restarted on another server

CSE 414 - Fall 2017

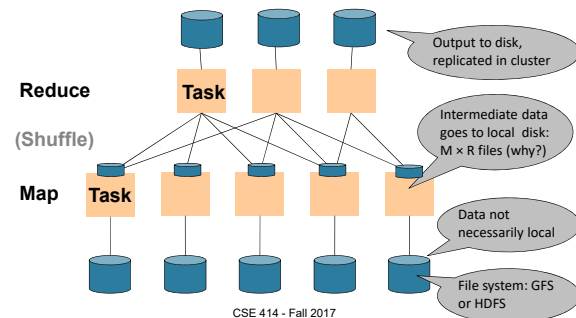
14



CSE 414 - Fall 2017

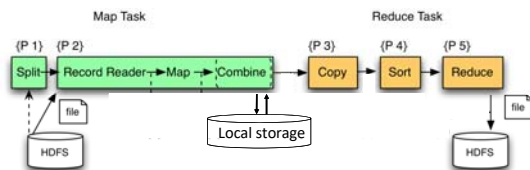
15

MapReduce Execution Details



CSE 414 - Fall 2017

MapReduce Phases



CSE 414 - Fall 2017

17

Implementation

- There is one master node
- Master partitions input file into M splits, by key
- Master assigns *workers* (=servers) to the M map tasks, keeps track of their progress
- Workers write their output to local disk, partition into R regions
- Master assigns workers to the R reduce tasks
- Reduce workers read regions from the map workers' local disks

CSE 414 - Fall 2017

18

Interesting Implementation Details

Worker failure:

- Master pings workers periodically,
- If down, then reassigns the task to another worker

CSE 414 - Fall 2017

19

Interesting Implementation Details

Backup tasks:

- **Straggler** = a machine that takes unusually long time to complete one of the last tasks. E.g.:
 - Bad disk forces frequent correctable errors (30MB/s → 1MB/s)
 - The cluster scheduler has scheduled other tasks on that machine
- Stragglers are a main reason for slowdown
- Solution: *pre-emptive backup execution of the last few remaining in-progress tasks*

CSE 414 - Fall 2017

20

Issues with MapReduce

- Difficult to write more complex queries
- Need multiple MapReduce jobs: dramatically slows down because it writes all results to disk
 - more recent systems work in memory
- Next lecture: Spark

CSE 414 - Fall 2017

21

Relational Operators in MapReduce

Given relations $R(A, B)$ and $S(B, C)$, compute:

- Selection: $\sigma_{A=123}(R)$
- Group-by: $\gamma_{A, \text{sum}(B)}(R)$
- Join: $R \bowtie S$

CSE 414 - Fall 2017

22

Selection $\sigma_{A=123}(R)$

```
map(String value):  
  if value.A = 123:  
    EmitIntermediate(value.key, value);
```

```
reduce(String k, Iterator values):  
  for each v in values:  
    Emit(v);
```

CSE 414 - Fall 2017

23

Selection $\sigma_{A=123}(R)$

```
map(String value):  
  if value.A = 123:  
    EmitIntermediate(value.key, value);
```

```
reduce(String k, Iterator values):  
  for each v in values:  
    Emit(v);
```

No need for reduce.
But need system hacking
to remove reduce from MapReduce

CSE 414 - Fall 2017

24

Group By $V_A, \text{sum}(B)(R)$

```
map(String value):
  EmitIntermediate(value.A, value.B);
```

```
reduce(String k, Iterator values):
  s = 0
  for each v in values:
    s = s + v
  Emit(k, s);
```

CSE 414 - Fall 2017

25

Join

Two simple parallel join algorithms:

- Partitioned hash-join
- Broadcast join

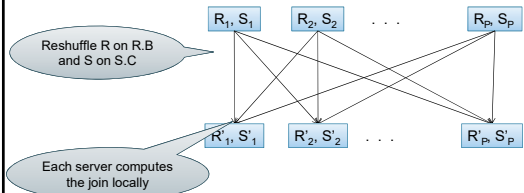
CSE 414 - Fall 2017

26

$R(A, B) \bowtie_{B=C} S(C, D)$

Partitioned Hash-Join

Initially, both R and S are horizontally partitioned



CSE 414 - Fall 2017

27

$R(A, B) \bowtie_{B=C} S(C, D)$

Partitioned Hash-Join

```
map(String value):
  case value.relationName of
    'R': EmitIntermediate(value.B, ('R', value));
    'S': EmitIntermediate(value.C, ('S', value));
```

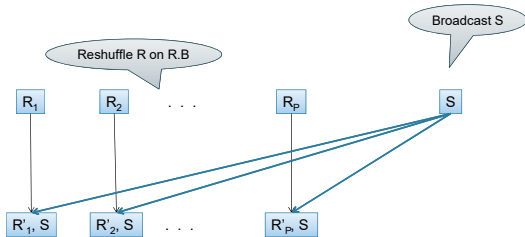
```
reduce(String k, Iterator values):
  R = empty; S = empty;
  for each v in values:
    case v.type of:
      'R': R.insert(v)
      'S': S.insert(v);
  for v1 in R, for v2 in S
    Emit(v1, v2);
```

CSE 414 - Fall 2017

28

$R(A, B) \bowtie_{B=C} S(C, D)$

Broadcast Join



CSE 414 - Fall 2017

29

$R(A, B) \bowtie_{B=C} S(C, D)$

Broadcast Join

```
map(String value):
  open(S); /* over the network */
  hashTbl = new()
  for each w in S:
    hashTbl.insert(w.C, w)
  close(S);

  for each w in hashTbl.find(value.B)
    Emit(v, w);
```

map should read several records of R: value = some group of records

Read entire table S, build a Hash Table

```
reduce(...):
  /* empty: map-side only */
```

CSE 414 - Fall 2017

30

Conclusions

- MapReduce offers a simple abstraction, and handles distribution + fault tolerance
- Speedup/scaleup achieved by allocating dynamically map tasks and reduce tasks to available server. However, skew is possible (e.g. one huge reduce task)
- Writing intermediate results to disk is necessary for fault tolerance, but very slow. Spark replaces this with “Resilient Distributed Datasets” = main memory + lineage

CSE 414 - Fall 2017

31

Conclusions II

- Widely used in industry
 - Google Search, machine learning, etc.
 - looks good on a resume
- Has been generalized (see Google DataFlow)
- Harder to use than necessary
 - language is imperative, not declarative (i.e., you have to actually write code)

CSE 414 - Fall 2017

32