CSE 444: Database Internals

Lecture 22
MapReduce

Announcements

- HW4 due tonight
- Lab4 due on Friday
- Next lab & hw is the final project!
- Final paper review due on Memorial Day
  - 5TH year master's only
  - Submit sometime around then (flexible)

Final Project Instructions

See course website for details!

1. Design and implementation:
   - There is **a mandatory part** and **extensions**
   - Design, implement, and evaluate one extension
2. Testing and evaluation
   - For your extension, write your own JUnit tests
   - For muliti-process tests, feel free to use scripts
3. Final report

Final Report

- Single-column & single-spaced
- Write your name!
- Structure of the final report
  - Sec 1. Overall System Architecture (2 pages)
    - Can reuse text from lab write-ups
  - Sec 2. Detailed design of the parallel data processing capabilities (3 pages)
  - Sec 3. Discussion (0.5-1 page)

Final Project Grading

- You will get two grades: one grade for your system and one grade for your final report.

References

- MapReduce: Simplified Data Processing on Large Clusters. Jeffrey Dean and Sanjay Ghemawat. OSDI'04
- Mining of Massive Datasets, by Rajaraman and Ullman, http://i.stanford.edu/~ullman/mmds.html
  - Map-reduce (Section 20.2);
  - Chapter 2 (Sections 1,2,3 only)

Outline

- Review high-level MR ideas from 344
- Discuss implementation in greater detail

Map Reduce Review

- Google: [Dean 2004]
- Open source implementation: Hadoop
- MapReduce = high-level programming model and implementation for large-scale parallel data processing

MapReduce Motivation

- Not designed to be a DBMS
- Designed to simplify task of writing parallel programs
  - A simple programming model that applies to many large-scale computing problems
- Hides messy details in MapReduce runtime library:
  - Automatic parallelization
  - Load balancing
  - Network and disk transfer optimizations
  - Handling of machine failures
  - Robustness
  - **Improvements to core library benefit all users of library!**

content in part from: Jeff Dean

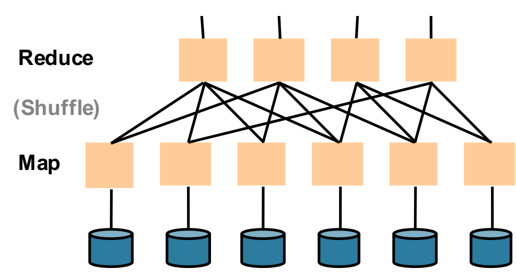
Data Processing at Massive Scale

- Want to process petabytes of data and more
- Massive parallelism:
  - 100s, or 1000s, or 10000s servers (think data center)
  - Many hours
- Failure:
  - If medium-time-between-failure is 1 year
  - Then 10000 servers have one failure / hour

Data Storage: GFS/HDFS

- MapReduce job input is a file
- Common implementation is to store files in a highly scalable file system such as GFS/HDFS
  - GFS: Google File System
  - HDFS: Hadoop File System
  - Each data file is split into M blocks (64MB or more)
  - Blocks are stored on random machines & replicated
  - Files are append only

Observation: Your favorite parallel algorithm…



Typical Problems Solved by MR

- Read a lot of data
- Map: extract something you care about from each record
- Shuffle and Sort
- Reduce: aggregate, summarize, filter, transform
- Write the results

Outline stays the same,
map and reduce change to fit
the problem

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

Step 1: the MAP Phase

User provides the MAP-function:

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

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

Step 2: the REDUCE Phase

User provides the REDUCE function:

- Input: **(intermediate key, bag of values)**
- Output (original MR paper): bag of output **(values)**
- Output (Hadoop): bag of **(output key, values)**

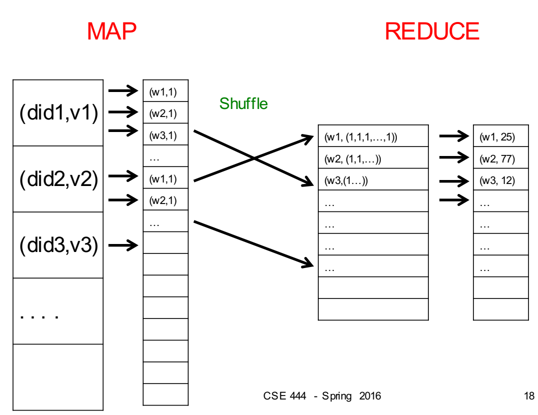
System groups all pairs with the same intermediate key, and passes the bag of values to the REDUCE function

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 += ParseInt(v);
Emit(AsString(result));
```

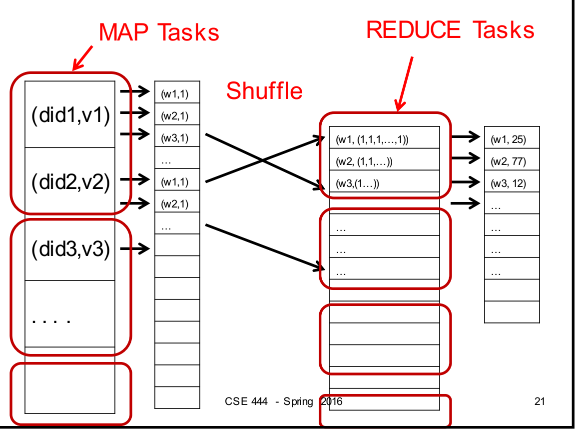



Jobs v.s. Tasks

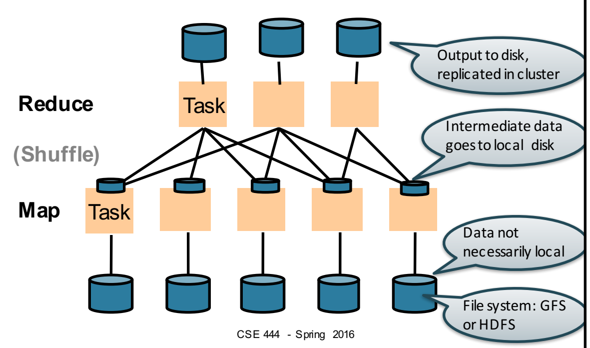
- A MapReduce Job
  - One single "query", e.g. count the words in all docs
  - More complex queries may consists of multiple jobs
- A Map Task, or a Reduce Task
  - A group of instantiations of the map-, or reduce-function, which are scheduled on a single worker

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
- Often talk about "slots"
  - E.g., Each server has 2 map slots and 2 reduce slots



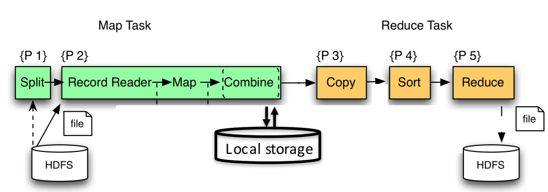
Parallel MapReduce Details



MapReduce Implementation

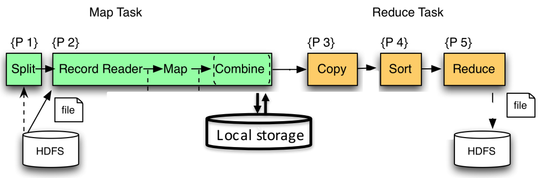
- There is one master node
- Input file gets partitioned further into *M' splits*
  - Each split is a contiguous piece of the input file
  - By default splits correspond to blocks
- Master assigns *workers* (=servers) to the *M' map tasks*, keeps track of their progress
- Workers write their output to local disk
- Output of each map task is partitioned into *R regions*
- Master assigns workers to the *R reduce tasks*
- Reduce workers read regions from the map workers local disks

MapReduce Phases



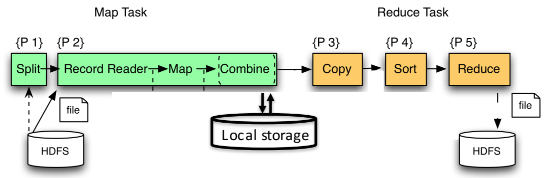
MapReduce Phases

Q: If we compute an aggregate, when can we use a combiner?



MapReduce Phases

Combine runs same code as reduce



Interesting Implementation Details

- Worker failure:
  - Master pings workers periodically,
  - If down then reassigns its task to *another* worker
  - (≠ a parallel DBMS restarts whole query)
- How many map and reduce tasks:
  - Larger is better for load balancing
  - But more tasks also add overheads
  - (≠ parallel DBMS spreads ops across all nodes)

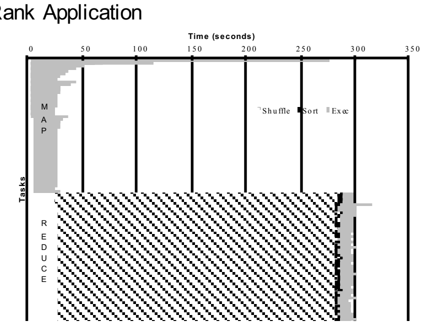
Interesting Implementation Details

Backup tasks:

- *Straggler* = a machine that takes unusually long time to complete one of the last tasks. Eg:
  - 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*

Skew

PageRank Application



The State of MapReduce Systems

- Lots of extensions to address limitations
  - Capabilities to write DAGs of MapReduce jobs
  - Declarative languages (see 344)
  - Ability to read from structured storage (e.g., indexes)
  - Etc.
- Most companies use both types of engines (MR and DBMS), with increased integration
- New systems are emerging: e.g. Spark

See Craig Chamber's talk on Wednesday at 3:30pm in Gates

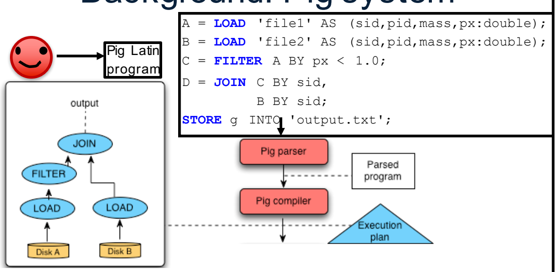
Declarative Languages on MR

- PIG Latin (Yahoo!)
  - New language, like Relational Algebra
  - Open source
- HiveQL (Facebook)
  - SQL-like language
  - Open source
- SQL / Tenzing (Google)
  - SQL on MR
  - Proprietary
  - Morphed into BigQuery

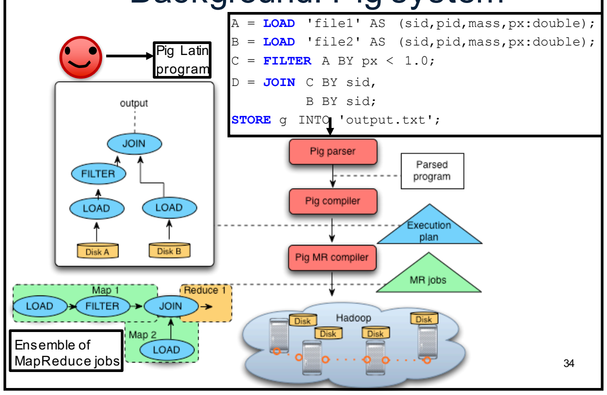
Relational Queries over MR

- Query → query plan
- Each operator → one MapReduce job
- Example: the Pig system

Background: Pig system



Background: Pig system



Doc(key, word)

GroupBy in MapReduce

MapReduce IS A GroupBy!

MAP=GROUP BY, REDUCE=Aggregate

```
SELECT word, sum(1)
FROM Doc
GROUP BY word
```


Joins in MapReduce

- If MR is GROUP-BY plus AGGREGATE, then how do we compute R(A,B) ⋈ S(B,C) using MR?

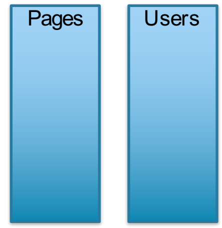
Joins in MapReduce

- If MR is GROUP-BY plus AGGREGATE, then how do we compute R(A,B) ⋈ S(B,C) using MR?
- Answer:
  - Map: group R by R.B, group S by S.B
    - Input = either a tuple R(a,b) or a tuple S(b,c)
    - Output = (b,R(a,b)) or (b,S(b,c)) respectively
  - Reduce:
    - Input = (b,{R(a1,b),R(a2,b),…,S(b,c1),S(b,c2),…})
    - Output = {R(a1,b),R(a2,b),…} × {S(b,c1),S(b,c2),…}
    - In practice: improve the reduce function (next…)

Users(name,age)
Pages(userName,url)

Join in MR

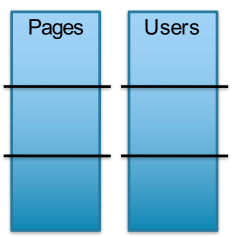
```
Users = load 'users' as (name, age);
Pages = load 'pages' as (userName, url);
Jnd = join Users by name, Pages by userName;
```



Users(name,age)
Pages(userName,url)

Join in MR

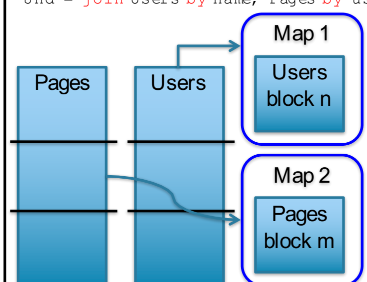
```
Users = load 'users' as (name, age);
Pages = load 'pages' as (userName, url);
Jnd = join Users by name, Pages by userName;
```



Users(name,age)
Pages(userName,url)

Join in MR

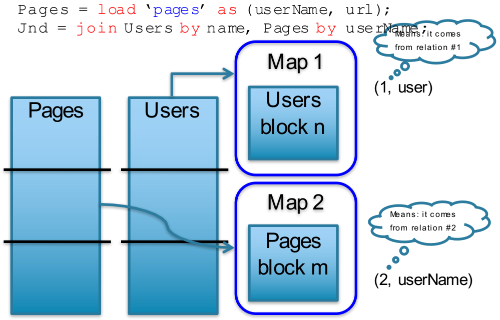
```
Users = load 'users' as (name, age);
Pages = load 'pages' as (userName, url);
Jnd = join Users by name, Pages by userName;
```



Users(name,age)
Pages(userName,url)

Join in MR

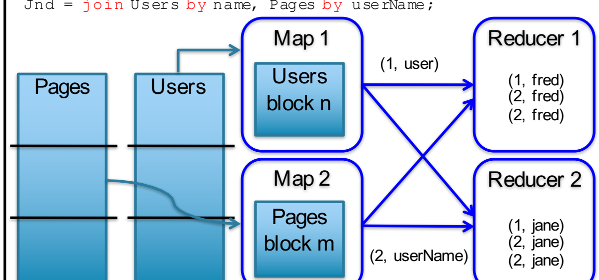
```
Users = load 'users' as (name, age);
Pages = load 'pages' as (userName, url);
Jnd = join Users by name, Pages by userName;
```



Users(name,age)
Pages(userName,url)

Join in MR

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (userName, url);
Jnd = join Users by name, Pages by userName;
```



Users(name,age)
Pages(userName,url)

Join in MR

```
Users = load 'users' as (name, age);
Pages = load 'pages' as (userName, url);
Jnd = join Users by name, Pages by userName;
```

```
map([String key], String value):
   // value.relation is either 'Users' or 'Pages'
   if value.relation='Users':
        EmitIntermediate(value.name, (1, value));
   else // value.relation='Pages':
        EmitIntermediate(value.userName, (2, value));
```

```
reduce(String user, Iterator values):
   Users = empty; Pages = empty;
   for each v in values:
     if v.type = 1: Users.insert(v)
     else Pages.insert(v);
   for v1 in Users, for v2 in Pages
        Emit(v1,v2);
```


Parallel DBMS vs MapReduce

- Parallel DBMS
  - Relational data model and schema
  - Declarative query language: SQL
  - Many pre-defined operators: relational algebra
  - Can easily combine operators into complex queries
  - Query optimization, indexing, and physical tuning
  - Streams data from one operator to the next without blocking
  - Can do more than just run queries: Data management
    - Updates and transactions, constraints, security, etc.

Parallel DBMS vs MapReduce

- MapReduce
  - Data model is a file with key-value pairs!
  - No need to "load data" before processing it
  - Easy to write user-defined operators
  - Can easily add nodes to the cluster (no need to even restart)
  - Uses less memory since processes one key-group at a time
  - Intra-query fault-tolerance thanks to results on disk
  - Intermediate results on disk also facilitate scheduling
  - Handles adverse conditions: e.g., stragglers
  - Arguably more scalable… but also needs more nodes!