Introduction to Data Management
CSE 344

Lecture 24: MapReduce
Announcements

• HW7 due yesterday
  – With 2 late days due tomorrow

• Final next Friday (in class)
  – 60 min (will give 75). Shorter than examples.
  – Monday will be last lecture covered by exam.

• Evaluation:
  https://uw.iasystem.org/survey/179899
Putting it Together:
Example Parallel Query Plan

Find all orders from today, along with the items ordered

```
SELECT *
FROM Order o, Line i
WHERE o.item = i.item
AND o.date = today()
```
Example Parallel Query Plan

Order(oid, item, date), Line(item, …)
Example Parallel Query Plan

Order(oid, item, date), Line(item, …)
Example Parallel Query Plan

Order(oid, item, date), Line(item, …)

Node 1

Join

o.item = i.item

contains all orders and all lines where hash(item) = 1

Node 2

Join

o.item = i.item

contains all orders and all lines where hash(item) = 2

Node 3

Join

o.item = i.item

contains all orders and all lines where hash(item) = 3
What’s wrong with the relational data model?

• From last class - NoSQL
  - nice not to have a schema

• For parallel data processing:
  – Want to control both data distribution and query processing
  – Want simpler programming model
    • “I don’t want to learn SQL!” (non 344 student)
  – Fault tolerance is important
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
Distributed File System (DFS)
The Problem

• Want to calculate in link counts for the entire web.

• What you have:
  – 30 billion webpages stored in GFS
  – In 100MB chunks on 10,000 nodes

• What you want:
  – List of pairs url : count (in bound link count)
The Solution

• Step 1
  – For each html document create keys {url -> count}
  – Distributed across all GFS nodes

• Step 2
  – Partition keys on url over cluster.

• Step 3
  – On each node sum up the total count of inbound links for each URL.

This is Map Reduce
Map Reduce Data Model

Started by Google in 2004

**Instance**: Files containing (key, value) pairs

**Schema**: None!
- just like other key-value data models

**Query language**: a MapReduce program:
- Input: a bag of (key, value) pairs
- Output: a bag of (key, value) pairs
- Implementation in Java (Hadoop), Python, Go, …
Lifecycle of a MR Program

1. Read a lot of data and parse into (key, value) pairs
2. Map: extract something you care about from each (key, value) pair
3. Shuffle output from mappers
   - done internally by implementation
4. Reduce: aggregate, summarize, filter, transform
5. Write the results to files

Paradigm stays the same, change map and reduce functions for different problems
Step 2: the MAP Phase

User provides the MAP-function:
• Input: \textbf{(key, value)}
• Output: bag of \textbf{(intermediate key, value)}

System applies the map function in parallel to all \textbf{(key, value)} pairs in the input file
Step 3: the shuffle phase

- System groups all pairs generated by MAPpers with the same intermediate key
- Passes the bag of values to the REDUCE function in next stage
- Example: given map output:
  ("a", 1), ("b", 1), ("a", 2), ("c", 1), ("c", 5)

  Shuffle produces the output:
  ("a", [1,2]), ("b", [1]), ("c", [1,5]) and partitions

- This is just another (key, value) pair!
Step 4: the **REDUCE** Phase

User provides the **REDUCE** function:

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

```java
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 "1"s
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```
Map Reduce Data Model

Key Points:

**Instance:** Files containing (key, value) pairs

**Schema:** None!
• just like other key-value data models

**Query language:** a MapReduce program (No SQL):
• Input: a bag of (key, value) pairs
• Output: a bag of (key, value) pairs
• Implementation in Java (Hadoop), Python, Go, …
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
MAP Tasks

(\text{did1,v1})

(\text{did2,v2})

(\text{did3,v3})

\ldots

\text{(w1,1)}

\text{(w2,1)}

\text{(w3,1)}

\ldots

...\text{(w1,1)}

\text{(w2,1)}

\ldots

\text{Shuffle}

\text{REDUCE Tasks}

(\text{w1, (1,1,1,\ldots,1)})

(\text{w2, (1,1,\ldots)})

(\text{w3,(1\ldots)})

\ldots

\text{\ldots}

\text{\ldots}

\text{\ldots}

\text{\ldots}

\text{\ldots}

\text{\ldots}

\text{\ldots}

\ldots

\text{(w1, 25)}

\text{(w2, 77)}

\text{(w3, 12)}

\ldots

\text{\ldots}

\text{\ldots}

\text{\ldots}

\text{\ldots}

\text{\ldots}

CSE 344 - Summer 2016
How can we optimize the shuffle?

Run Reduce function on each Node before shuffle.

New step: Combine
How can we optimize the shuffle?
MapReduce Phases

Map Task

{P 1} Split → Record Reader → Map → Combine

{P 2} → Local storage

Reduce Task

{P 3} Copy → Sort → Reduce

{P 4} → {P 5} → HDFS

HDFS

file
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
Interesting Implementation Details

Worker failure:

• Master pings workers periodically,

• If down then reassigns the task to another worker
Fault Tolerance

- If one server fails once every year, how long before a job running on 10,000 servers fails
  - 1h 10 min! \( \frac{10,000}{365 \times 24} \)

- 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
MapReduce Execution Details

Map

Task

Reduce

Task

(Shuffle)

Intermediate data goes to local disk

Output to disk, replicated in cluster

Data not necessarily local

File system: GFS or HDFS

CSE 344 - Summer 2017
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*
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**: $R \Join S$
Selection $\sigma_{A=42}(R)$

```python
def map(String relationName, Tuple t):
    if t.A == 42:
        EmitIntermediate(relationName, t);
```

```python
def reduce(String k, Iterator values):
    for each v in values:
        Emit(v);
```
Selection $\sigma_{A=42}(R)$

```java
map(String relationName, Tuple t):
    if t.A == 42:
        EmitIntermediate(relationName, t);
```

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

- Reduce isn’t really needed
- But MR requires reduce functions
Group By $\gamma_{A,\text{sum}(B)}(R)$

map(String relationName, Tuple t):
   EmitIntermediate(t.A, t.B);

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

Two parallel join algorithms that we have seen:

- Partitioned hash-join
- Broadcast join
Parallel Execution of RA Operators: Partitioned Hash-Join

- **Data**: $R(K_1, A, B)$, $S(K_2, B, C)$
- **Query**: $R(K_1, A, B) \Join S(K_2, B, C)$
  - Initially, both $R$ and $S$ are partitioned on $K_1$ and $K_2$

- Reshuffle $R$ on $R.B$ and $S$ on $S.B$
- Each server computes the join locally
Partitioned Hash-Join in MR

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

**map**\( (\text{String relationName, Tuple } t) \):

\[
\begin{align*}
\text{switch (relationName):} \\
\text{case 'R': EmitIntermediate(t.B, IntKey('R', value));} \\
\text{case 'S': EmitIntermediate(t.C, IntKey('S', value));}
\end{align*}
\]

**reduce**\( (\text{String } k, \text{ Iterator values}) \):

\[
\begin{align*}
R & = []; \quad S = []; \\
\text{for each } v \text{ in values:} \\
\text{switch (v.relationName):} \\
\text{'R': R.insert(v);} \\
\text{'S': S.insert(v);} \\
\text{for } r \text{ in } R, \text{ for } s \text{ in } S \\
\text{ Emit(Tuple(r,s));}
\end{align*}
\]

Or call hash(t.B)

All tuples here must join
Data: \( R(A, B), S(C, D) \)
Query: \( R(A, B) \bowtie_{B=C} S(C, D) \)

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

**Broadcast Join in MR**

```java
map(Tuple [] rs):
    S = readFromNetwork();
    ht = new Hashtable()
    for each w in S:
        ht.insert(w.C, w)

    for each r in ts:
        for each s in ht.find(r.B):
            Emit(Tuple(r,s));

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

- **map** should read several records of R: 
  - value = some group of records
- Read entire table S, build a Hash Table
Issues with MapReduce

• Difficult to write more complex queries

• Need multiple MapReduce jobs: dramatically slows down because it writes all results to disk

• We will talk about Spark in the next lecture
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.

- Next time: Spark replaces this with “Resilient Distributed Datasets” = main memory + lineage