Introduction to Database Systems
CSE 414

Lecture 17: MapReduce and Spark
Announcements

• Midterm this Friday in class!
  – Review session tonight
  – See course website for OHs
  – Includes everything up to Monday’s lecture

• HW6 released
  – Not due until next Friday 5/11
  – No WQ6 (Yay!)
Approaches to Parallel Query Evaluation

• **Inter-query parallelism**
  – One query per node
  – Good for transactional (OLTP) workloads

• **Inter-operator parallelism**
  – Operator per node
  – Good for analytical (OLAP) workloads

• **Intra-operator parallelism**
  – Operator on multiple nodes
  – Good for both?

We study only intra-operator parallelism: most scalable
Parallel Data Processing in the 20th Century
Parallel Execution of RA Operators: Partitioned Hash-Join

- **Data**: $R(K_1, A, B), S(K_2, B, C)$
- **Query**: $R(K_1, A, B) \bowtie S(K_2, B, C)$
  - 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

Reshuffle $R'$ on $R'.B$ and $S'$ on $S'.B$
Parallel Join Illustration

Data: $R(K_1, A, B)$, $S(K_2, B, C)$
Query: $R(K_1, A, B) \bowtie S(K_2, B, C)$

Partition

Shuffle on B

Local Join
Data: R(A, B), S(C, D)
Query: R(A,B) \bowtie_{B=C} S(C,D)
Parallel Data Processing @ 2000
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
Motivation

• We learned how to parallelize relational database systems

• While useful, it might incur too much overhead if our query plans consist of simple operations

• MapReduce is a programming model for such computation

• First, let’s study how data is stored in such systems
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
MapReduce

- Google: paper published 2004
- Free variant: Hadoop

- MapReduce = high-level programming model and implementation for large-scale parallel data processing
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

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

Files!

A file = a bag of \((\text{key}, \text{value})\) pairs

Sounds familiar after HW5?

A MapReduce program:
• Input: a bag of \((\text{inputkey}, \text{value})\) pairs
• Output: a bag of \((\text{outputkey}, \text{value})\) pairs
  – outputkey is optional
Step 1: the MAP Phase

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

System applies the map function in parallel to all \((\text{input key}, \text{value})\) pairs in the input file.
Step 2: the REDUCE Phase

User provides the REDUCE function:

- Input: \((\text{intermediate key, bag of values})\)
- Output: bag of output \((\text{values})\)

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));
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 (M) → REDUCE Tasks (R) → Shuffle

(did1,v1) → (w1,1) → (w2,1) → (w3,1) → ...
(did2,v2) → (w1,1) → (w2,1) → ...
(did3,v3) → ...
...
...
...

(w1, (1,1,1,...,1)) → (w1,25) →...
(w2, (1,1,...)) → (w2,77) →...
(w3,(1...)) → (w3,12) →...

...
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
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

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*
Straggler Example

Worker 1

Worker 2

Worker 3

Backup execution

Straggler

Killed

Killed
Using MapReduce in Practice:
Implementing RA Operators in MR
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**: $R \bowtie S$
Selection $\sigma_{A=123}(R)$

map(Tuple t):
   if t.A = 123:
       EmitIntermediate(t.A, t);

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

<table>
<thead>
<tr>
<th>A</th>
<th>t1</th>
<th>t2</th>
<th>t3</th>
<th>t4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>23</td>
<td>123</td>
<td>123</td>
<td>42</td>
</tr>
</tbody>
</table>
Selection $\sigma_{A=123}(R)$

map(Tuple t):
  if t.A = 123:
    EmitIntermediate(t.A, t);

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

No need for reduce.
But need system hacking in Hadoop
  to remove reduce from MapReduce
Group By $\gamma_{A, \text{sum}(B)}(R)$

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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁</td>
<td>23</td>
</tr>
<tr>
<td>t₂</td>
<td>123</td>
</tr>
<tr>
<td>t₃</td>
<td>123</td>
</tr>
<tr>
<td>t₄</td>
<td>42</td>
</tr>
</tbody>
</table>

reduce(String A, Iterator values):

\[
\begin{align*}
\text{s} &= 0 \\
\text{for each } v \text{ in values:} \\
\text{s} &= s + v \\
\text{Emit}(A, s);
\end{align*}
\]

\[(23, [ t₁ ]), (42, [ t₄ ]), (123, [ t₂, t₃ ])\]
Join

Two simple parallel join algorithms:

• Partitioned hash-join (we saw it, will recap)

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

Partitioned Hash-Join

Initially, both $R$ and $S$ are horizontally partitioned

- Reshuffle $R$ on $R.B$ and $S$ on $S.B$
- Each server computes the join locally

Diagram:

- $R_1, S_1$
- $R_2, S_2$
- $\ldots$
- $R_P, S_P$

- $R'_1, S'_1$
- $R'_2, S'_2$
- $\ldots$
- $R'_P, S'_P$
\[ R(A,B) \bowtie_{B=C} S(C,D) \]

**Partitioned Hash-Join**

**map(Tuple t):**

- case t.relationName of
  - ‘R’: EmitIntermediate(t.B, ('R', t));
  - ‘S’: EmitIntermediate(t.C, ('S', t));

**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);
Broadcast Join

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

**Broadcast Join**

**map** (String value):
- readFromNetwork(S); /* over the network */
- hashTable = new HashTable()
- for each w in S:
  - hashTable.insert(w.C, w)
- for each v in value:
  - for each w in hashTable.find(v.B)
    - Emit(v, w);

**reduce** (...):
/* empty: map-side only */
HW6

• HW6 will ask you to write SQL queries and MapReduce tasks using Spark

• You will get to “implement” SQL using MapReduce tasks
  – Can you beat Spark’s implementation?