MapReduce

SUNNIE S CHUNG
CIS 612
MapReduce

- Challenges in distributed processing/analysis
  - Extremely large data sets (petabytes of data)
  - Data is distributed across many systems (potentially thousands)
  - Individual systems are not aware of all data
  - Data is too large to be held or processed by any singular system

- How to distribute and process a task that works over petabytes of data?
  - How to parallelize the computation
  - How to distribute the data
  - How to handle failures
Word Count over a Given Set of Web Pages

Can we do word count in parallel?
MapReduce

- MapReduce was introduced as a programming model by Google in 2004*
- Spreads the task of processing data out over many systems
- Key-Value based system
  - Elements of divide and conquer
  - Makes use of the concept of hashing
- Used for extremely parallel data processing/analysis
- Highly scalable
- Failure tolerant

* MapReduce: Simplified Data Processing on Large Clusters, OSDI’04: Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004
MapReduce

How does MapReduce work?

Three primary steps are used to run a MapReduce job:

- Map
- Shuffle
- Reduce

Data is read in a parallel fashion across many different nodes in a cluster (Map):

- Groups are identified for processing the input data, then output

The data is then shuffled into these groups (Shuffle):

- All data with a common group identifier (Key) is then sent to a single location

Each group is then processed atomically across the nodes in parallel. (Reduce):

- Each node will receive one or more groups to process
- Each group is then processed and the results of the union of each group is the result of the entire job.
MapReduce Example 1: Map Function for Word Count

Map Function

- Input: key/value pair
- Output: Intermediate key/value pair

```java
map(String key, String value):
    // Key: Document Name
    // Value: Document Contents
    for each word w in values
        EmitIntermediate(w, "1")
```
MapReduce: Programming Model

Declaration of Independence

When in the Course of human events it becomes necessary for one people to dissolve the political bands which have connected them with another and to assume among the powers of the earth, the separate and equal station to which the Laws of Nature and of Nature's God entitle them, a decent respect to the opinions of mankind requires that they should declare the causes which impel them to the separation.

map(String key, String value):
   // Key: Document Name
   // Value: Document Contents
   for each word w in values
      EmitIntermediate(w, “1”)
MapReduce Example 1: Reduce Function for Word Count

Reduce Function
- Input: intermediate key/value pair
- Output: results

reduce(String key, String value):
    // Key: word
    // Value: a list of counts
    int result = 0;
    for each v in values:
        result += parseInt(v);
    Emit(As(String(result))’
MapReduce Programming Model

<table>
<thead>
<tr>
<th></th>
<th>Count</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>for</td>
<td>1</td>
<td>among</td>
</tr>
<tr>
<td>one</td>
<td>1</td>
<td>and</td>
</tr>
<tr>
<td>people</td>
<td>1</td>
<td>another</td>
</tr>
<tr>
<td>to</td>
<td>1</td>
<td>assume</td>
</tr>
<tr>
<td>dissolve</td>
<td>1</td>
<td>bands</td>
</tr>
<tr>
<td>the</td>
<td>1</td>
<td>connected</td>
</tr>
<tr>
<td>political</td>
<td>1</td>
<td>dissolve</td>
</tr>
<tr>
<td>bands</td>
<td>1</td>
<td>for</td>
</tr>
<tr>
<td>which</td>
<td>1</td>
<td>have</td>
</tr>
<tr>
<td>have</td>
<td>1</td>
<td>one</td>
</tr>
<tr>
<td>connected</td>
<td>1</td>
<td>people</td>
</tr>
<tr>
<td>them</td>
<td>1</td>
<td>political</td>
</tr>
<tr>
<td>with</td>
<td>1</td>
<td>the</td>
</tr>
<tr>
<td>another</td>
<td>1</td>
<td>them</td>
</tr>
<tr>
<td>and</td>
<td>1</td>
<td>to</td>
</tr>
<tr>
<td>to</td>
<td>1</td>
<td>which</td>
</tr>
<tr>
<td>assume</td>
<td>1</td>
<td>with</td>
</tr>
<tr>
<td>among</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>the</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Output: <word, count> Pairs
MapReduce Example 2: Word Count

- Reads in files from the HDFS
- Returns the counts of all words located in the files
- Map:
  
  Mapper (line_number, line_contents)
  
  for each word in line_contents
  
  emit(word, 1)

- Reduce:
  
  Reducer (word, values[])
  
  sum = 0
  
  for each value in values
  
  sum = sum + value
  
  emit(word, sum)
Word-Counting

“You jump, I jump.”
Both jump.

Master

Worker - Map Instance
(You, 1)
(jump, 1)

Worker - Reduce Instance
You, 1
I, 1

Worker - Map Instance
(l, 1)
(jump, 1)

Worker - Reduce Instance
Both, 1
jump, 3

Worker - Map Instance
(Both, 1)
(jump, 1)

Reducer pulls the data

Intermediary result stored on Mapper’s local disk

Final output written to DFS

Input file from Distributed File System (DFS), e.g. GFS
Another Example

- Count of URL Access Frequency
  - Map
    - Input: Log of Web Page Requests
    - Output: <URL,1> Intermediate Pair
  - Reduce
    - Input: Intermediate Pairs
    - Output: <URL, Total Count> Pair
MapReduce

- A MapReduce (MR) program consists of generally two user defined stages.
  - Map:
    - Map(Key_p, Value_p) → list(Key_i, Value_i)
      - Takes in a data element
      - Outputs Key-Value pairs
  - Reduce:
    - Reduce(Key_i, list(Value_i)) → list(Key_s, Value_s)
      - Takes in a Key and a collection of Values
      - Outputs results
**Map Phase Process**

**Map Phase:**

- Many different Map Tasks will be created across the different nodes in the distributed cluster

- Each task will receive a block of the data to be processed

- For each input element in that block, it will make a call to $\text{Map}(\text{Key}_p, \text{Value}_p)$

- For each call of the $\text{Map}(\text{Key}_p, \text{Value}_p)$, it will emit a list of Key-Value pairs derived from the input element in the form $\text{list}(\text{Key}_i, \text{Value}_i)$

- All of the list($\text{Key}_i, \text{Value}_i$) from the different Map Tasks and calls to Map will then be shuffled across the network to the different Reduce Tasks

- This shuffling produces the $<\text{Key}_i, \text{list}(\text{Value}_i)>$ data collection that are the input to the Reduce Phase
MapReduce Process Flow

Diagram of a MapReduce Job Flow
Reduce Phase Process

- **Reduce Phase:**
  - The Reduce phase of MapReduce takes in a group defined by a unique key: \((Key_i)\) and a list of values: \((\text{list}(Value_i))\)

- Many different reduce tasks will be created across the different nodes in the distributed cluster
  - Each task will make a call to \(\text{Reduce}(Key_i, \text{list}(Value_i))\) for every reduce group that it receives
  - Each call will process that reduce group atomically
The MapReduce Framework (by Google)
Overall MapReduce Word Count Process
Automatic Parallel Execution in MapReduce (Google)

Handles failures automatically, e.g., restarts tasks if a node fails; runs multiples copies of the same task to avoid a slow task slowing down the whole job.
**MapReduce: Word Count**

Data in HDFS

- Input File 1
  - This is line one
- Input File 2
  - But this is line two
- Input File 3
  - And I am line three

Node 1

- Map
  - Input: This is line one
  - Output:
    - <"this", 1>
    - <"is", 1>
    - <"line", 1>
    - <"one", 1>
- Reduce

Node 2

- Map
  - Input: But this is line two
  - Output:
    - <"but", 1>
    - <"is", 1>
    - <"line", 1>
    - <"two", 1>
- Reduce

Node 3

- Map
  - Input: And I am line three
  - Output:
    - <"and", 1>
    - <"i", 1>
    - <"am", 1>
    - <"line", 1>
    - <"three", 1>
- Reduce

Shuffle

Write output to HDFS

Output:

- <"is", 2>
- <"one", 1>
- <"and", 1>
- <"line", 3>
- <"i", 1>
- <"three", 1>
MapReduce in Hadoop (1)

Figure 2-2. MapReduce data flow with a single reduce task
MapReduce in Hadoop (2)

Figure 2-3. MapReduce data flow with multiple reduce tasks
Figure 2-4. MapReduce data flow with no reduce tasks
Data Flow in a MapReduce Program in Hadoop

- InputFormat
- Map function
- Partitioner
- Sorting & Merging
- Combiner
- Shuffling
- Merging
- Reduce function
- OutputFormat

\[
\begin{align*}
\text{Input Format} & : \text{data} \rightarrow K_1, V_1 \\
\text{Mapper} & : K_1, V_1 \rightarrow K_2, V_2 \\
\text{Combiner} & : K_2, \text{iter}(V_2) \rightarrow K_2, V_2 \\
\text{Partitioner} & : K_2, V_2 \rightarrow \text{int} \\
\text{Reducer} & : K_2, \text{iter}(V_2) \rightarrow K_3, V_3 \\
\text{Out. Format} & : K_3, V_3 \rightarrow \text{data}
\end{align*}
\]
Figure 6-4. Shuffle and sort in MapReduce
Execution Overview
Step 1

User Program invokes the Map Reduce Library

Splits the input file into M pieces

Starts up copies of the program on a cluster of machines
Step 2

**Master / Workers**

- Designate one Master
- Designate M Map Workers
- Designate R Reduce Workers
Step 3

Execute the Map Function

The map workers read the contents of the Input Split and generate intermediate

The intermediate <key, value> pairs are buffered into memory.
Step 4

Buffer / Partition / Notify

Periodically, the buffered pairs are written to the local disk.

The information is partitioned into R regions.

The location of these partitions is sent to the Master.
Step 5

Execute the Reduce Function

RPC is used to read the Partitioned pairs from their local locations

Once all data is read, it must be sorted
Step 6

Iterate and Reduce

The Reduce Worker iterates over the Intermediate Results.

Unique Keys are passed into the user defined Reduce Function.

The output is appended to a final output file for this reduce partition.
Step 7

Provide Results

- Wake up the user program
- Return back to user code
- Generally these output files will be passed into some other application versus combining the results into a single file.
The Master

- Stores the state and identity of all workers

- Central control for passing the intermediate file regions from the map tasks to the reduce tasks.
Fault Tolerance - Monitoring

- Hadoop MapReduce is fault tolerant with hardware/ networking failure, and input/program error.
- Slave nodes send heartbeat messages to the master node periodically.
- Master considers a node is dead by absence of heartbeat message
- No further requests are sent to dead nodes.
Fault Tolerance - Monitoring

- Three things to recover
  - Data Block on failed machine [by DFS]
  - Map work/result on failed machine [by MapReduce]
  - Reduce work/result on failed machine [by MapReduce]
Data Block Recovery

- A data block need to be recovered (adding duplicated copy) when
  - A slave node is down
  - A hard disk of a slave is down
  - A data block is corrupted
  - Replication factor is increased (by administrator)
- Recall each block has multiple copies on different nodes
- As long as some copy is intact, master node can issue commands of creating new copies of the block on available nodes to meet the required number of copies.
Map work recovery

- If a node is down, all the map work assigned to it will be executed by other available nodes.
  - Finished map work on a failed node also need to be re-executed because the map output is stored locally on the failed node and hence unreachable.
- If a map work instance hangs, it will be tried again on other available node, the not responding instance will be killed.
- Reducer only needs to load the re-executed Map output from other machines.
- No further impact on reducer because real reduce work will not start until all maps finish.
Reduce Work Recovery

- If a node fails, its *unfinished* reduce work will be assigned to other available nodes.
- No impact on maps, i.e. maps need not be re-executed for failed reducers
  - Because map output is still hold on local hard disk. New reducers only need to pull the output again
- Finished reduce work on a failed node does not need to be re-executed because final output has been written into DFS.
Skipping Bad Records

- An option in MapReduce of skipping bad records which cause crashes.
- Sometimes ignoring some bad records is acceptable. E.g., for statistical analysis on large data.
- A signal is sent to the Master indicating a crash and on which record (line) it happens.
- If more than one signals on the same record are received, the master can tell next running node to skip that bad record.
Fault Tolerance

- Data replication across nodes.
  - If one machine is dead, other machines can still get access to the data
  - As long as data is accessible, any machine can do the job – “push computation to data” – preferably the one which holds a copy

- Strict Staging
  - Reduce can not start until all Maps finish (though copy and sorting could start in advance)
  - Chained MR programs do not overlap – Depended job must finish first before a depending job starts
  - Effect: Failure at one stage only lead to re-execution at this stage