Processing of massive data: MapReduce

1. Introduction to MapReduce
Origins: the Problem

- Google faced the problem of analyzing huge sets of data (order of petabytes)
- E.g. pagerank, web access logs, etc.
- Algorithm to process data can be reasonable simple
- But to finish it in an acceptable amount of time the task must be split and forwarded to potentially thousands of machines
Origins: the Problem (2)

- Programmers were forced to develop the sw that:
  - Splits data
  - Forwards data and code to participant nodes
  - Checks nodes state to react to errors
  - Retrieves and organizes results
- Tedious, error-prone, time-consuming... and had to be done for each problem
The Solution: MapReduce

- MapReduce is an abstraction to organize parallelizable tasks
- Algorithm has to be adapted to fit MapReduce's main two steps.
  1) **Map**: data processing
     (collecting/grouping/distribution intermediate step)
  2) **Reduce**: data collection and digesting
- The MapReduce framework will take care of data/code transport, nodes coordination, etc.
Example: Word Count

- Given a text, get the number of times each word in the text appears.

```
Text to pass to wc
```

```
<table>
<thead>
<tr>
<th>Map</th>
<th>Intermediate step</th>
<th>Reduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>text: 1 to: 1 pass: 1</td>
<td>text: {1} to: {1,1} pass: {1} wc: {1}</td>
<td>text: {1} to: {2} pass: {1} wc: {1}</td>
</tr>
<tr>
<td>to wc</td>
<td>text: {1} to: {1,1}</td>
<td>text: 1 to: 2 pass: 1 wordcount: 1</td>
</tr>
</tbody>
</table>
```

Processing of massive data: MapReduce – 1. Introduction to MapReduce
Map Definition

• **Map function** signature:
  - takes as input a \((T_{k1}: \text{key}, T_{v1}: \text{value})\) pair
  - generates list of \((T_{k2}: \text{key}, T_{v2}: \text{value})\) pairs

\[
\text{map}(T_{k1}: k1, T_{v1}: v1) \rightarrow \text{list}(T_{k2}: k2, T_{v2}: v2)
\]

• Note that
  - Input and output types can be different
  - In the output list the same key can be repeated
Word Count Example – Map Func.

- In the Word Count example, the Map function could be programmed as follows:

```java
map(String key, String value) {
    // key: doc name, value: doc text
    for each word w in value:
        EmitIntermediate(w, "1");
}
```

```plaintext
("fichero", "text to pass to wc")
```

```plaintext
{("text", 1),("to", 1),
 ("pass", 1),("to", 1),
 ("wc", 1)}
```
Intermediate Step

• The MapReduce software will:
  1) collect all results from nodes running Map step
  2) group the pairs (T_k2: key, T_v2: value) by key
  3) distribute keys among nodes running Reduce step

\[
\begin{align*}
  (k2, v2) & \quad \rightarrow \quad (k2, \{v2, v2', v2''\}) \\
  (k2, v2') & \quad \rightarrow \quad (k3, \{v3\}) \\
  (k3, v3) & \quad \rightarrow \quad (k4, \{v4\}) \\
  (k2, v2'') & \quad \rightarrow \quad (k2, \{v2, v2', v2''\}) \\
  (k4, v4) & \quad \rightarrow \quad (k2, \{v2, v2', v2''\})
\end{align*}
\]

• This step does not depend on the algorithm implementation passed to MapReduce
Word Count Example – Interim Step

- Intermediate results processing would look like:

\[
\begin{align*}
\{(\text{"text"}, 1), \quad & (\text{"to"}, 1), \\
(\text{"pass"}, 1), \quad & (\text{"to"}, 1), \\
(\text{"wc"}, 1)\} & \quad \rightarrow \quad \{(\text{"text"}, \{1\}), \\
(\text{"to"}, \{1,1\}), \quad & (\text{"pass"}, \{1\}), \\
(\text{"wc"}, \{1\})\}
\end{align*}
\]
Reduce Definition

• **Reduce function** signature:
  - takes as input \((T_{k2}:key, \text{list}(T_{v2}:value))\)
  - generates another list of \((T_{v2}:value)\) values

\[
\text{reduce}(T_{k2}:k2, \text{list}(T_{v2}:v2)) \rightarrow (T_{k2}:k2, \text{list}(T_{v2}:v2))
\]

• Note that
  - Processing of each key is independent
  - Typically the resulting list has one element
Word Count Example – Reduce Func

- The Reduce function could be coded as follows:

```java
reduce(String key, List values) {
    // key: word, value: list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(key, result);
}
```

Input:
- ("text", {1})
- ("to", {1,1})
- ("pass", {1})
- ("wc", {1})

Output after Reduce:
- ("text", {1})
- ("to", {2})
- ("pass", {1})
- ("wc", {1})
Summing Up Map and Reduce Defs

\[ \text{map}(T_{k1:k1}, T_{v1:v1}) \rightarrow \text{list}(T_{k2:k2}, T_{v2:v2}) \]

Intermediate:

collection, sorting and
distribution of map
results.

\[ (k2, v2) \]
\[ (k2, v2') \]
\[ (k2, v2'') \]

\[ \rightarrow (k2, \{v2, v2', v2''\}) \]

\[ \text{reduce}(T_{k2:k2}, \text{list}(T_{v2:v2})) \rightarrow (T_{k2:k2}, \text{list}(T_{v2:v2})) \]
The Combiner function

- It is an *optional* step right after Map
- Typically, consists on applying the Reduce function locally in the Map node before sending the results
- It can reduce the bandwidth and disk space consumed by the Map output data
- Applicable if the reduce function is commutative and associative
  - In the previous wordcount example it can be applied
Processing of massive data: MapReduce – 1. Introduction to MapReduce

**Distributed MapReduce**

Map process, distributed among **Map Nodes**

- Intermediate process
  1) collects values by key
  2) sorts by key

Reduce process, distributed among **Reduce Nodes**
Distributed Implementations Hints

- In “big data” jobs is more efficient moving code than moving data.
- Map (and Reduce) nodes work independently.
- A special *master* node plans the distribution of tasks.
  - It also monitors nodes, who periodically report their status.
  - Failure tolerance is simple: if some node fails the master can replace it immediately (the rest do not need to be notified about it).
- Map tasks can run in parallel, and Reduce tasks can work in parallel (but, *can Reduce tasks work at the same time than Map tasks?*)
Distributed Implementations Hints (2)

• User specifies:
  
  • $M$: Number of slots or pieces the input data is split into (i.e., number of Map tasks to run)
  
  • $R$: Number of pieces the intermediate data is split into (i.e., number of Reduce tasks to run)
    
    - To assign intermediate data to Reduce nodes some function must be used. For example:

      $$\text{hash}(k2) \mod R$$

      That function usually balances data fairly among Reduce partitions

• Typical Google job: $M=200,000; R=5,000; 2,000$ machines
Processing of massive data: MapReduce – 1. Introduction to MapReduce

Distributed Implementation Schema

User Program → Master

- Split 0 → Map node
- Split 1 → Map node
- Split 2 → Map node
- ... → Map node
- Split M → Map node

Distributed file system (GFS, HDFS...)

M Map tasks

Local write

Remote Read

Reduce node

- Reduce node
- Reduce node
- ... Reduce tasks

R local disk buffers

Output file 0

Output file 1

Distributed file system (GFS, HDFS...)

New Trends In Distributed Systems
MSc Software and Systems
More MapReduce Examples

• Grep (distributed): look for lines that match a pattern
  - Map → Checks text line by line, it emits matching lines
  - Reduce → Identity

• Count of URL access frequency: count in web request logs how many times each URL has been accessed
  - Map → Each time an URL appears, emit <URL, 1>
  - Reduce → Add together all values, emit <URL, sum>

• Pages that link to a certain URL (reverse web graph)
  - Map → For each URL in some page, emit <URL, page>
  - Reduce → Identity
Applications of MapReduce

- Three main areas:
  - Text tokenization, indexing, and search
  - Creation of other kinds of data structures (e.g., graphs)
  - Data mining and machine learning

- By sector:
  - Originally developed and used by web companies: Google, Facebook, Yahoo!, Ebay, Adobe, Twitter, Last.fm, LinkedIn...
  - The scientific community is also applying it to large datasets:
    - Statistical algorithms (k-means, linear regression...); image processing; genetic sequences search

- Check [http://wiki.apache.org/hadoop/PoweredBy](http://wiki.apache.org/hadoop/PoweredBy)!
Beyond MapReduce

- MapReduce has a 'low semantic interface'
  - I.e.: Map and Reduce operations are too simple
  - Complex manipulation of data is cumbersome (for example, compared with the flexibility of SQL)
- Higher level languages have been proposed
  - They work on top of MapReduce, transforming queries into MapReduce operations
  - Examples: DryadLINQ, Sawzall, PygLatin, Hive

```
hive> CREATE TABLE pokes (foo INT, bar STRING);
hive> SELECT a.foo FROM pokes WHERE a.bar='test';
```
Bibliography

- (Paper) “MapReduce: Simplified Data Processing on Large Data Clusters”. Jeffrey Dean, Sanjay Ghemawat. OSDI 2004


Processing of massive data: MapReduce – 1. Introduction to MapReduce

(Example with Hadoop)