CS 378 – Big Data Programming

Lecture 2
Map-Reduce
MapReduce

• Large data sets are not new

• What characterizes a problem suitable for MR?
  – Most or all of the data is processed
    • But viewed in small increments
    • For the most part, map and reduce tasks are stateless
  – Write once, read multiple times
    • Data Warehouse has this intended usage (write once)
  – Unstructured data vs. structured/normalized

• Data pipelines are common
  – Chain of MR jobs, with intermediate results
# MapReduce

Table 1-1, Hadoop – The Definitive Guide

<table>
<thead>
<tr>
<th></th>
<th>Traditional RDBMS</th>
<th>MapReduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Size</td>
<td>Gigabytes</td>
<td>Petabytes</td>
</tr>
<tr>
<td>Access</td>
<td>Interactive and batch</td>
<td>Batch</td>
</tr>
<tr>
<td>Updates</td>
<td>Read and write many times</td>
<td>Write once, read many</td>
</tr>
<tr>
<td>Structure</td>
<td>Static schema</td>
<td>Dynamic schema</td>
</tr>
<tr>
<td>Integrity</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Scaling</td>
<td>Nonlinear</td>
<td>Linear</td>
</tr>
</tbody>
</table>
MapReduce

- Tom White, in *Hadoop: The Definitive Guide*

- “MapReduce works well on unstructured or semistructured data because it is designed to interpret the data at processing time. In other words, the input keys and values for MapReduce are not intrinsic properties of the data, but they are chosen by the persona analyzing the data.”
MapReduce

• When writing a MapReduce program ...
  – You don’t know the size of the data
  – You don’t know the extent of the parallelism

• MapReduce tries to collocate the data with the compute node
  – Parallelize the I/O
  – Make the I/O local (versus across network)
MapReduce

• As the name implies, for each problem we’ll write
  – Map method/function
  – Reduce method/function

• Terms from functional programming

• Map
  – Apply a function to each input, output the result

• Reduce
  – Given a list of inputs, compute some output value
The number of reduce tasks is not governed by the size of the input, but instead is specified independently. In "The Default MapReduce Job" on page 227, you will see how to choose the number of reduce tasks for a given job.

When there are multiple reducers, the map tasks partition their output, each creating one partition for each reduce task. There can be many keys (and their associated values) in each partition, but the records for any given key are all in a single partition. The partitioning can be controlled by a user-defined partitioning function, but normally the default partitioner—which buckets keys using a hash function—works very well.

The data flow for the general case of multiple reduce tasks is illustrated in Figure 2-4.

This diagram makes it clear why the data flow between map and reduce tasks is colloquially known as "the shuffle," as each reduce task is fed by many map tasks. The shuffle is more complicated than this diagram suggests, and tuning it can have a big impact on job execution time, as you will see in "Shuffle and Sort" on page 208.

Finally, it's also possible to have zero reduce tasks. This can be appropriate when you don't need the shuffle because the processing can be carried out entirely in parallel (a few examples are discussed in "NLineInputFormat" on page 247). In this case, the only off-node data transfer is when the map tasks write to HDFS (see Figure 2-5).

Combiner Functions

Many MapReduce jobs are limited by the bandwidth available on the cluster, so it pays to minimize the data transferred between map and reduce tasks. Hadoop allows the user to specify a combiner function to be run on the map output, and the combiner...
Map Function

• Map input is a stream of key/value pairs
  – Web logs: Server name (key), log entry (value)
  – Sensor reading: sensor ID (key), sensed values (value)
  – Document ID (key), contents (value)

• Map function processes each input pair in turn

• For each input pair, the map function can (but isn’t required) to emit one or more key/value pairs
  – Key/value pair(s) derived from the input key/value pair
  – Does not need to be the same key or value data type
Reduce Function

• Reduce input is a stream of key/value-list pairs
  – These are the key value pairs emitted by the map function

• Reduce function processes each input pair in turn

• For each input pair, the reduce function can (but isn’t required) to emit a key/value pair
  – Key value pair derived from the input key/value-list pair
  – Does not need to be the same key or value data type
WordCount Example

• For an input text file of arbitrary size, or
• Multiple text files of arbitrary size, or
• An arbitrary number of documents

• Count the number occurrences of all the words that appear in the input.
• Output:
  – word1, count
  – word2, count
  – ...

WordCount Example - Map

• Map input is a stream of key/value pairs
  – File position in bytes (key), line of text (value)

• Map function processes each input pair in turn
  – Extract each word from the line of text
  – Emits a key/value pair for each word: <the-word, 1>

• For each input pair, the map function emits multiple key/value pairs
  – Key is a text string (the word), value is a number
WordCount Example - Reduce

- Reduce input is a stream of key/value-list pairs
  - These are the key value pairs emitted by the map function
  - Key is a text string (the word), value is a list of some number of the value “1”
  - Hadoop has grouped data together by key

- Reduce function processes each input pair in turn
  - Sums the values in the value-list

- For each input pair, the reduce function emits a key/value pair
  - Key is a text string (the word), value is total count for that word
The number of reduce tasks is not governed by the size of the input, but instead is specified independently. In "The Default MapReduce Job" on page 227, you will see how to choose the number of reduce tasks for a given job.

When there are multiple reducers, the map tasks partition their output, each creating one partition for each reduce task. There can be many keys (and their associated values) in each partition, but the records for any given key are all in a single partition. The partitioning can be controlled by a user-defined partitioning function, but normally the default partitioner—which buckets keys using a hash function—works very well.

The data flow for the general case of multiple reduce tasks is illustrated in Figure 2-4. This diagram makes it clear why the data flow between map and reduce tasks is colloquially known as "the shuffle," as each reduce task is fed by many map tasks. The shuffle is more complicated than this diagram suggests, and tuning it can have a big impact on job execution time, as you will see in "Shuffle and Sort" on page 208.

Finally, it's also possible to have zero reduce tasks. This can be appropriate when you don't need the shuffle because the processing can be carried out entirely in parallel (a few examples are discussed in "NLineInputFormat" on page 247). In this case, the only off-node data transfer is when the map tasks write to HDFS (see Figure 2-5).

Combiner Functions

Many MapReduce jobs are limited by the bandwidth available on the cluster, so it pays to minimize the data transferred between map and reduce tasks. Hadoop allows the user to specify a combiner function to be run on the map output, and the combiner...
MapReduce
(from cubrid.org)
Java and Maven Review

• Directory structure expected by maven (supported in IDEs):
  – Project directory (example name: bdp)
  – Source code directory: bdp/src/main/java
  – The Java package structure appears in the “java” directory
  – Ex: bdp/src/main/java/com/refactorlabs/cs378/assign1
  – A class defined in the com.refactorlabs.cs378/assign1 package placed here
  – Ex: bdp/src/main/java/com/refactorlabs/cs378/assign1/WordCount.java

• Easy setup - Create you project directory
  – Place pom.xml in this directory
  – Place WordCount.java as shown above
  – Import the maven pom.xml into your IDE.
Assignment Artifacts

• For each assignment, there will be one or more artifacts to submit:
  – Java code
    • Source files in one directory (for easy inspection)
    • Source files in `src/main/java/...` structure (use “tar”)
  – Build info: `pom.xml` file used for maven
    • An initial `pom.xml` file will be provided, and we’ll expand this during the semester
  – Program outputs
    • Extracted from HDFS

• Artifacts required for each assignment will be listed.
Assignment 1

• Build a JAR file
• Upload to AWS S3
• Create a cluster using Elastic MapReduce (EMR)
• Run your map-reduce job on EMR cluster
• Download the output