Basics of Cloud Computing – Lecture 5

MapReduce Algorithms

Pelle Jakovits
Satish Srirama

Some material adapted from slides by Jimmy Lin, Web-Scale Information Processing Applications course, University of Waterloo (licensed under Creation Commons Attribution 3.0 License)
Outline

• Recap of the MapReduce model
• Example MapReduce algorithms
• Designing MapReduce algorithms
  – How to represent everything using only Map, Reduce, Combiner and Partitioner tasks
  – Managing dependencies in data
  – Using complex data types
MapReduce model

- Programmers specify Map and Reduce functions:
  - `map (k, v) \rightarrow (k', v')`*
    - Applies a user defined function on every input record
    - Values with the same key are grouped together before Reduce phase
  - `reduce (k', [v']) \rightarrow (k'', v'')`*
    - Applies a user defined aggregation function on the list of values
- The execution framework handles everything else!
- Users have opportunity to also define:
  - **Partitioner** - Controls how keys are partitioned between reducers
    - `partition (k, nr. of partitions) \rightarrow partition_id for k`
  - **Combiner** - Mini-reducer applied at the end of the map phase
    - `combine (k', [v']) \rightarrow (k'', v'')`*
Shuffle and Sort: aggregate values by keys

(a, 1) (b, 1) (a, 1)
(c, 1) (b, 1) (c, 1)
(b, 1) (c, 1)
(a, 1) (a, 1) (c, 1)

combine

(a, 2) (b, 1)
(c, 2) (b, 1)
(b, 1) (c, 1)
(a, 2) (c, 1)

reduce

(a, [2, 2])
(b, [1, 1, 1])
(c, [2, 1, 1])

reduce

(a, 4)
(b, 3)
(c, 4)
Typical Hadoop Use Cases

• **Extract, transform and load** (ETL) pipelines
  – Perform transformation, normalization, aggregations on the data
  – Load results into database or data warehouse
  – Ex: Sentiment analysis of review websites and social media data

• **Reporting and analytics**
  – Generate statistics, run ad-hoc queries and information retrieval tasks
  – Ex: Analyzing web clickstream, marketing, CRM, & email data

• **Machine learning**
  – Ex: Building recommender systems for behavioral targeting
  – Ex: Face similarity and recognition over large datasets of images

• **Graph algorithms**
  – Ex: Identifying trends and communities by analyzing social network graph data

[Powered By Hadoop -](https://wiki.apache.org/hadoop/PoweredBy)
MapReduce Jobs

• Tend to be very short, code-wise
  – Identity Reducer is common

• Represent a data flow, rather than a procedure
  – Data „flows“ through Map and Reduce stages

• Can be composed into larger data processing pipelines

• Iterative applications may require repeating the same job multiple times

• Data must be partitioned across many reducers if it is large

• Data will be written into multiple output files if there are more than a single Reduce task
Different MapReduce input formats

- The input types of a MapReduce application are not fixed and depend on the input format that is used

<table>
<thead>
<tr>
<th>InputFormat</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TextInputFormat (Default)</td>
<td>Byte offset of the line (LongWritable)</td>
<td>Line contents Text</td>
</tr>
<tr>
<td>KeyValueInputFormat</td>
<td>User Defined Writable Object (e.g.\ PersonWritable)</td>
<td>User Defined Writable Object</td>
</tr>
<tr>
<td>WholeFileInputFormat</td>
<td>NullWritable</td>
<td>File contents (BytesWritable)</td>
</tr>
<tr>
<td>NLineInputFormat</td>
<td>Byte offset of the line block (LongWritable)</td>
<td>Contents of N lines (Text)</td>
</tr>
<tr>
<td>TableInputFormat (HBase)</td>
<td>Row Key</td>
<td>Value</td>
</tr>
</tbody>
</table>
Designing MapReduce algorithms

• General goal of a MapReduce algorithm:
  – How to produce desired **Output** from the **Input data**?

• To define a MapReduce algorithm, we need to define:
  
 1. **Map Function**
     • What is **Map Input** (Key, Value) pair
     • What is **Map Output** (Key, Value) pair
     • **Map Function**: Input (Key, Value) → Output (Key, Value)

 2. **Reduce Function**
     • What is Reduce **Input** (Key, [Value]) pair
     • What is Reduce **Output** (Key, Value) pair
     • **Reduce Function**: Input (Key, [Value]) → Output (Key, Value)

Let's look at a few Example MapReduce algorithms
MapReduce Examples

• Counting URL Access Frequency
• Distributed Grep
• Distributed Sort
• Inverted Index
• Conditional Probabilities
Counting URL Access Frequency

• Process web access logs to count how often each URL was visited
  – **Input:** (LineOffset, Line)
  – **Output:** (URL, count)

• Very similar to the MapReduce WordCount algorithm

• **Map function**
  – Processes one log record at a time
  – Emit (URL, 1) if an URL appears in log record

• **Reduce function**
  – Sum together all values
  – Emit (URL, total_count) pair
Distributed Grep

- Distributed version of the Linux command line Grep command
- Find all rows in a set of text files that contain a supplied regular expression
  - **Input**: (LineOffset, Line)
  - **Output**: (LineOffset, Line)

- **Map function**
  - Emits a line **ONLY** if it matches the supplied regular expression

- **Reduce function**
  - Identity function
  - Emits all input data as (Key, Value) pairs without modifications
MapReduce Algorithm Design Process

1. Structure of the input data \(\rightarrow\) Defines **Job Input** (Key, Value)
2. Desired result \(\rightarrow\) Defines **Job Output** (Key'', Value'')
3. If the desired result can be computed **without shuffling data**:
   - **Map Function**: Job Input (Key, Value) \(\rightarrow\) Job Output (Key'', Value'')
   - **Reduce Function**: Use **Identity** function!
4. If data **needs to be shuffled**:
   - **Map Function**:
     - How should data be grouped \(\rightarrow\) Defines Map Output Key'
     - What values are needed in Reduce task \(\rightarrow\) Defines Map Output Value'
     - **Function**: Job Input (Key, Value) \(\rightarrow\) Map Output (Key', Value')
   - **Reduce Function**:
     - **Input**: Based on Map Output: (Key’, [Value’])
     - **Function**: Reduce Input (Key’, [Value’]) \(\rightarrow\) Job Output (Key’’, Value’’)

13.03.2019
Satish Srirama
Inverted Index Algorithm

- Generate a **Word to File** index for each word in the input dataset
- **Input**: Set of text files
- **Output**: For each word, return a list of files it appeared in
- **Map Function**
  - **Input**: (LineOffset, Line)
  - **Function**: Extract words from the line of text.
  - **Output**: (word, fileName)
- **Reduce Function**
  - **Input**: (word, [fileName])
  - **Function**: Concatenate list of file names into a single string
  - **Output**: (word, “[fileName]“)
Index: Data Flow

Page A

This page contains so much of text

A map output
This : A
page : A
contains : A
so : A
much : A
of : A
text : A

Page B

This page too contains some text

B map output
This : B
page : B
too : B
contains : B
some : B
text : B

Reduced output
This : A, B
page : A, B
too : B
contains : A, B
so : A
much : A
of : A
text : A, B
some : B
Inverted Index MapReduce pseudocode

**map** (LineOffset, Line, context):
   pageName = context.getInputSplitFileName()
   foreach word in Line:
      emit(word, pageName)

**reduce** (word, values):
   pageList = []
   foreach pageName in values:
      pageList.add(pageName)
   emit(word, str(set(pageList)))
Distributed Global Sort

• Task is to sort a very large list of numerical values
• Each value is in a separate line inside a text file
• **Input:** A set of text files
• **Output:** values are in a globally sorted order in the output files

• Can be used as a benchmark to measure the raw throughput of the MapReduce cluster
Sort: The Trick

• Take advantage of Reducer properties:
  – (Key, Value) pairs are processed in order by key
  – (Key, Value) pairs from mappers are sent to a particular reducer based on Partition(key) function

• Change the Partition function
  – Must use a partition function such that:

\[
\text{IF } K_1 < K_2 \text{ THEN Partition}(K_1) \leq \text{Partition}(K_2)
\]
Distributed Sort algorithm

• **Map Function**
  – **Input:** ( LineOffset, Line )
  – **Function:** Move the value into the Key
  – **Output:** ( Line, _ )

• **Reduce Function**
  – **Input:** (Line, [ _ ] )
  – **Function:** Identity Reducer
  – **Output:** ( Line, _ )
Distributed Sort Data Flow

File A
- 023567
- 911234
- 278689
- 867867
- 232245
- 145663

A map output
- (023567, "")
- (911234, "")
- (278689, "")
- (867867, "")
- (232245, "")
- (145663, ")

File B
- 385566
- 888888
- 952442
- 332432
- 195677
- 035567

B map output
- (385566, "")
- (888888, "")
- (952442, "")
- (332432, "")
- (195677, "")
- (035567, ")

Reducer 0 output
- (023567, "")
- (035567, ")
- (145663, ")
- (195677, ")

Reducer 1 output
- (232245, "")
- (278689, ")
- (332432, ")
- (385566, ")

Reducer 9 output
- (867867, ")
- (888888, ")
- (911234, ")
- (952442, ")
Let us focus on a bit more complex problems
Managing Dependencies in Data

• Remember: Mappers run in isolation
  – You have no control over the order in which mappers run
  – You have no control over: on which nodes the mappers run
  – You have no control over: when each mapper finishes

• Tools for synchronization:
  – Ability to hold state in reducer across multiple key-value pairs
  – Sorting function for keys
  – Partitioners
  – Broadcasting/replicating values
  – Cleverly-constructed data structures
Motivating Example

• Term co-occurrence matrix for a text collection
  – \( M = N \times N \) matrix (\( N \) = vocabulary size)
  – \( M_{ij} \): number of times \( i \) and \( j \) co-occur in some context (let’s say context = sentence)

• Why?
  – Distributional profiles as a way of measuring semantic distance
  – Semantic distance useful for many language processing tasks

“You shall know a word by the company it keeps” (Firth, 1957)
MapReduce: Large Counting Problems

- Term co-occurrence matrix for a text collection
  = specific instance of a large counting problem
  - A large event space (number of terms)
  - A large number of events (the collection itself)
  - Goal: keep track of interesting statistics about the events

- Basic approach
  - Mappers generate partial counts
  - Reducers aggregate partial counts

How do we aggregate partial counts efficiently?
First approach: “Pairs”

• Each mapper takes a sentence:
  – Generate all co-occurring term pairs
  – For all pairs, emit \((a, b) \rightarrow \text{count}\)

• Reducers sums up counts associated with these pairs

• Output of the MapReduce job:
  – **Key**: WordA, WordB
  – **Value**: count

• Use combiners!
“Pairs” Analysis

• Advantages
  – Easy to implement, easy to understand

• Disadvantages
  – Lots of pairs to sort and shuffle around (upper bound?)
Second approach: “Stripes”

• Idea: group together pairs into an associative array

\[
\begin{align*}
(a, b) & \rightarrow 1 \\
(a, c) & \rightarrow 2 \\
(a, d) & \rightarrow 5 \\
(a, e) & \rightarrow 3 \\
(a, f) & \rightarrow 2
\end{align*}
\]

\[
a \rightarrow \{ b: 1, c: 2, d: 5, e: 3, f: 2 \}
\]

• Each mapper takes a sentence:
  – Generate all co-occurring term pairs
  – For each term, emit \( a \rightarrow \{ b: \text{count}_b, c: \text{count}_c, d: \text{count}_d \ldots \} \)

•Reducers perform element-wise sum of associative arrays

\[
\begin{align*}
a & \rightarrow \{ b: 1, d: 5, e: 3 \} \\
+ \ a & \rightarrow \{ b: 1, c: 2, d: 2, f: 2 \} \\
\hline
a & \rightarrow \{ b: 2, c: 2, d: 7, e: 3, f: 2 \}
\end{align*}
\]
“Stripes” Analysis

• Advantages
  – Far less sorting and shuffling of key-value pairs
  – Can make better use of combiners

• Disadvantages
  – More difficult to implement
  – Underlying object is more heavyweight
  – Fundamental limitation in terms of size of event space
Efficiency comparison of approaches to computing word co-occurrence matrices

Cluster size: 38 cores
Data Source: Associated Press Worldstream (APW) of the English Gigaword Corpus (v3), which contains 2.27 million documents (1.8 GB compressed, 5.7 GB uncompressed)
Conditional Probabilities

• What is the chance of word B occurring in a sentence that contains word A.

• How do we compute conditional probabilities from counts?

\[ P(B|A) = \frac{\text{count}(A, B)}{\text{count}(A)} = \frac{\text{count}(A, B)}{\sum_{B'} \text{count}(A, B')} \]

• How do we compute this with MapReduce?
P(B | A): “Pairs”

• Co-occurrence matrix already gives us: \( \text{count}(A, B) \)
• Need to also compute \( \text{count}(A) \)

\[
\begin{align*}
(a, b_1) \rightarrow 3 & \quad \Rightarrow \quad (a, b_1) \rightarrow 3 / 23 \\
(a, b_2) \rightarrow 12 & \quad \Rightarrow \quad (a, b_2) \rightarrow 12 / 23 \\
(a, b_3) \rightarrow 7 & \quad \Rightarrow \quad (a, b_3) \rightarrow 7 / 23 \\
(a, b_4) \rightarrow 1 & \quad \Rightarrow \quad (a, b_4) \rightarrow 1 / 23 \\
\end{align*}
\]

Reducer holds this value in memory

• How can we compute \( \text{count}(a) \) without changing how the data is grouped?
  – Must also emit an extra \((X, \star)\) for every term \(X\) in mapper
  – Must make sure all \(a\)’s get sent to same reducer (use Partitioner)
  – Must make sure \((a, \star)\) key is processed first (sort order) in Reduce
P(B|A): “Stripes”

\[ a \rightarrow \{ b_1 : 3, b_2 : 12, b_3 : 7, b_4 : 1, \ldots \} \]

- Easy!
  - One pass to compute \((a, *)\)
  - Another pass to directly compute \(P(B|A)\)
Synchronization in Hadoop

• **Approach 1:** turn synchronization into an ordering problem
  – Partition key space so that each reducer gets the appropriate set of partial results
  – Sort keys into correct order of computation
  – Hold state in reducer across multiple key-value pairs to perform computation
  – Illustrated by the “pairs” approach

• **Approach 2:** construct data structures that “bring the pieces together”
  – Each reducer receives all the data it needs to complete the computation
  – Illustrated by the “stripes” approach
Issues and Tradeoffs

• Number of key-value pairs
  – Object creation overhead
  – Time for sorting and shuffling pairs across the network

• Size of each key-value pair
  – De/serialization overhead

• Combiners make a big difference!
  – RAM vs. disk and network
  – Arrange data to maximize opportunities to aggregate partial results
Complex Data Types in Hadoop

• How to use more complex data types as Keys and Values?
  • The easiest way:
    – Encode it as a composed String, e.g., (a, b) = “a;b”
    – Use regular expressions to parse and extract data
    – Works, but pretty hack-ish
  • The hard way:
    – Define a custom implementation of WritableComparable
    – Must implement: readFields, write, compareTo
    – Computationally more efficient, but slow for rapid prototyping
public class MyKey implements WritableComparable {
    private int ID;
    private long phone_num;

    public void write(DataOutput out) {
        out.writeInt(ID);
        out.writeLong(phone_num);
    }

    public void readFields(DataInput in) {
        ID = in.readInt();
        phone_num = in.readLong();
    }

    public int compareTo(MyKey o) {
        int res = Integer.compare(this.ID, o.ID);
        if (res != 0)
            return res;
        return Long.compare(this.phone_num, o.phone_num);
    }
}

Custom Hadoop WritableComparable Object
Multiple shuffle stages

• Data in a MapReduce application can only be shuffled once!
• There is no way to "move" data between Map and Reduce tasks once the Map or Reduce stage has started
• What if a single aggregation is not enough?
  – Does the data need to be re-grouped?
    • Yes: We may need to create multiple MapReduce jobs and execute them in a sequence
    • No: Reduce can apply multiple aggregation functions on the same list of values
Next Lab

• Creating a new MapReduce application
  – Analyzing an open dataset
  – Parsing CSV files
  – Aggregating data using simple statistical functions
Next Lecture

• Platform as a Service (PaaS) model
  – Google AppEngine
  – Elastic MapReduce (EMR)
    • MapReduce platform as a Service
References


• Jimmy Lin and Chris Dyer, "Data-Intensive Text Processing with MapReduce"
Pages 50-57: Pairs and Stripes problem