Cloud Computing – Lecture 8

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
• Designing MapReduce algorithms
• Example MapReduce algorithms
• Data Synchronization issues
• Additional notes on MapReduce jobs
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** \((k, \text{nr. of partitions}) \rightarrow \text{partition\_id}\) for \(k\)
  – **Combiner** - Mini-reducer applied at the end of the map phase
    • **combine** \((k', [v']) \rightarrow (k'', v'')\)*
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)** \(\rightarrow\) **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])** \(\rightarrow\) **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 is a set of log files
• Resulting MapReduce algorithm is very similar to WordCount
MapReduce URL Access Frequency

• **Input:** (LineOffset, Line)
• **Output:** (URL, count)
• **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
• Input is a set of text files
• Find all rows in a set of text files that contain a supplied regular expression
MapReduce Distributed Grep

- **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’’)

Lets apply this process at example MapReduce algorithms!
Inverted Index Algorithm

• Generate a **Word -> File** index for each word in the input dataset

• Input is a set of text files

```
Page A
This page contains so much of text

Page B
This page too contains some text

Output
This : A, B
page : A, B
too : B
contains : A, B
so : A
much : A
of : A
text : A, B
some : B
```
MapReduce Inverted Index

- **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]“ )
Inverted Index: Data Flow

A map output

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

Reduced output

This : A, B
page : A, B
too : B
contains : A, B
so : A
much : A
of : A
text : A, B
some : B

B map output

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

Page A

This page contains so much of text

Page B

This page too contains some text
Inverted Index MapReduce pseudocode

```java
map(LineOffset, Line, context):
    pageName = context.getInputSplitFileName()
    for word in Line:
        emit(word, pageName)

reduce(word, values):
    pageList = []
    for 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 in a globally sorted order in the output files

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

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

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

    IF  \[ K1 < K2 \]
    THEN  \[ \text{Partition}(K1) \leq \text{Partition}(K2) \]
Distributed Sort algorithm

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

• Reduce Function
  – Input: (Number, [ _ ])
  – Function: Identity Reducer
  – Output: (Number, _)

(Number, _) is emitted in Reduce for each _
Distributed Sort Data Flow

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

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

Reducer 0 output:
- (023567, '')
- (035567, '')
- (145663, '')
- (195677, '')

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

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

Reducer 1 output:
- (232245, '')
- (278689, '')
- (332432, '')
- (385566, '')

Reducer 4 output:
- (867867, '')
- (888888, '')
- (911234, '')
- (952442, '')
Let's focus on a bit more complex problems
Conditional Probabilities

• For each word A and B in the dataset:
  – What is the chance of word B occurring in a sentence that contains A.

• We can compute conditional probabilities from word 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?
  – Can use WordCount to compute count(A)
  – How to compute count(A,B)?
  – How to gather count(A,B) and count(A) values together to compute division?
Term co-occurrence matrix

• We first need to compute count(A, B) for each term A and B
• 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)

• How large is the resulting matrix?
• How many elements do we need to count?
• Leads us to Large Counting problems

30.03.2020
21/39
Large Counting Problems

• Term co-occurrence matrix for a text collection
  => specific instance of a large counting problem
  – A large event space (number of words)
  – A large number of events (the number of sentences)

• Basic approach
  – Mappers generate partial counts
    • Map output is larger than input -> Data “explosion”
  – Reducers aggregate partial counts into full counts
    • Huge amount of tiny operations

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

- WordCount-like approach
- 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
- Need to use combiners!
“Pairs” Analysis

• Advantages
  – Easy to implement
  – Easy to understand

• Disadvantages
  – Lots of pairs to sort and shuffle around
  – Too many tiny messages to be synchronized
Second approach: “Stripes”

• Idea: group together pairs into an associative array

\[(a, b) \rightarrow 1\]
\[(a, c) \rightarrow 2\]
\[(a, d) \rightarrow 5\]
\[(a, e) \rightarrow 3\]
\[(a, f) \rightarrow 2\]

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

• 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

\[
\begin{align*}
  a \rightarrow \{ \, b: 1, \, d: 5, \, e: 3 \, \} \\
  + a \rightarrow \{ \, b: 1, \, c: 2, \, d: 2, \, f: 2 \, \} \\
  a \rightarrow \{ \, b: 2, \, c: 2, \, d: 7, \, e: 3, \, f: 2 \, \}
\end{align*}
\]
“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
Conditional Probabilities

- For each word A and B in the dataset:
  - What is the chance of word B occurring in a sentence that contains A.

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

- We have now computed count(A,B) values using MapReduce
- How do we compute \( \text{count}(A,B) / \text{count}(A) \)?
  - How do we solve dependencies between separately computed counts?
  - How to make sure \( \text{count} (A) \) is available for every \( \text{count} (A,B') \) in MR
Managing Dependencies in Data

• Remember, Mappers run in isolation
• We can't control:
  – The order in which mappers run
  – On which nodes the mappers run
  – When each mapper finishes
• Available tools for synchronization:
  – Ability to hold state in reducer across multiple key-value pairs
  – Sorting function for keys – to control the order of data
  – Partitioners - to control which data/keys are together
  – Broadcasting data to all Map or Reduce task
  – Cleverly-constructed data structures
P(B|A): “Pairs”

- Co-occurrence matrix already gives us: count(A, B)
- Need to also compute count(A) and count(A,B)/count(A)

(a, _) → 23  
Reducer holds this value in memory

(a, b₁) → 3  
(a, b₁) → 3 / count(a)  
(a, b₁) → 3 / 23
(a, b₂) → 12  
(a, b₂) → 12 / count(a)  
(a, b₂) → 12 / 23
(a, b₃) → 7  
(a, b₃) → 7 / count(a)  
(a, b₃) → 7 / 23
(a, b₄) → 1  
(a, b₄) → 1 / count(a)  
(a, b₄) → 1 / 23

- How can we compute count(a) without changing how the data is grouped?
  - Must also emit an extra (a, _) for every bᵢ in mapper
  - Force all (a, _), (a, bᵢ) to be sent to same reducer using Partitioner
  - Must make sure (a, _) comes first (define sort order)
P(B|A): “Stripes”

• If we use the Stripes approach, we have associative arrays for each term a:

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

• Then computing \( \text{count}(a) \) is easy!
  – No synchronization is required!
  – We can directly compute \( \text{count}(A, B) / \text{count}(A) \) at the end of the Reduce method
  – One pass to compute \( (a, *) \)
  – Another pass to directly compute \( P(B|A) \)
Pairs vs Stripes Issues and Trade-offs

• 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
Synchronization in Hadoop

• **Approach 1:** turn synchronization into an ordering problem
  – Partition keys 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
Notes about 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>
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);
    }
}
Next Lab

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


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