Data-Intensive Distributed Computing
CS 431/631 451/651 (Fall 2019)

Part 1: MapReduce Algorithm Design (4/4)

Ali Abedi

These slides are available at https://www.student.cs.uwaterloo.ca/~cs451/

This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States
See http://creativecommons.org/licenses/by-nc-sa/3.0/us/ for details
MapReduce Algorithm Design

How do you express everything in terms of m, r, c, p?
Toward “design patterns”
MapReduce
MapReduce

Programmer specifies four functions:

- **map** \( (k_1, v_1) \rightarrow \text{List}[(k_2, v_2)] \)
- **reduce** \( (k_2, \text{List}[v_2]) \rightarrow \text{List}[(k_3, v_3)] \)

All values with the same key are sent to the same reducer

- **partition** \( (k', p) \rightarrow 0 \ldots p-1 \)
  - Often a simple hash of the key, e.g., \( \text{hash}(k') \mod n \)
  - Divides up key space for parallel reduce operations

- **combine** \( (k_2, \text{List}[v_2]) \rightarrow \text{List}[(k_2, v_2)] \)
  - Mini-reducers that run in memory after the map phase
  - Used as an optimization to reduce network traffic

The execution framework handles everything else...
combine

map

map

map

map

combine

combine

combine

combine

partition

partition

partition

partition

* Important detail: reducers process keys in sorted order

reduce

reduce

reduce

r_1 s_1

r_2 s_2

r_3 s_3
“Everything Else”

Handles scheduling
Assigns workers to map and reduce tasks

Handles “data distribution”
Moves processes to data

Handles synchronization
Gathers, sorts, and shuffles intermediate data

Handles errors and faults
Detects worker failures and restarts
But...

You have limited control over data and execution flow!
All algorithms must be expressed in m, r, c, p

You don’t know:
Where mappers and reducers run
When a mapper or reducer begins or finishes
Which input a particular mapper is processing
Which intermediate key a particular reducer is processing
Tools for Synchronization

Preserving state in mappers and reducers
Capture dependencies across multiple keys and values

Cleverly-constructed data structures
Bring partial results together

Define custom sort order of intermediate keys
Control order in which reducers process keys
Two Practical Tips

Avoid object creation
(Relatively) costly operation
Garbage collection

Avoid buffering
Limited heap size
Works for small datasets, but won’t scale!
Importance of Local Aggregation

Ideal scaling characteristics:
Twice the data, twice the running time
Twice the resources, half the running time

Why can’t we achieve this?
Synchronization requires communication
Communication kills performance

Thus... avoid communication!
Reduce intermediate data via local aggregation
Combiners can help
Distributed Group By in MapReduce

Mapper

circular buffer (in memory)

spills (on disk)

 Mapper

merged spills (on disk)

Combiner

intermediate files (on disk)

Reducer

other reducers

 other mappers

Combiner

intermediate files (on disk)
What’s the impact of combiners?
Are combiners still needed?

class Mapper {
    def map(key: Long, value: String) = {
        val counts = new Map()
        for (word <- tokenize(value)) {
            counts(word) += 1
        }

        for ((k, v) <- counts) {
            emit(k, v)
        }
    }
}
## Performance

Word count on 10% sample of Wikipedia

<table>
<thead>
<tr>
<th></th>
<th>Running Time</th>
<th># Pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>~140s</td>
<td>246m</td>
</tr>
<tr>
<td>Histogram</td>
<td>~140s</td>
<td>203m</td>
</tr>
</tbody>
</table>
Can we do even better?
As shown in the diagram, the process begins with a 'map' operation that takes the input values 

\[ k_1 v_1, k_2 v_2, k_3 v_3, k_4 v_4, k_5 v_5, k_6 v_6 \]

and produces intermediate key-value pairs:

\[ a \ 1 \ b \ 2 \ c \ 3 \ c \ 6 \ a \ 5 \ c \ 2 \ b \ 7 \ c \ 8 \]

These are then partitioned into groups based on their keys:

\[ a \ 1 \ b \ 2 \ c \ 9 \ a \ 5 \ c \ 2 \ b \ 7 \ c \ 8 \]

The partitioned data is then combined:

\[ a \ 1 \ b \ 2 \ c \ 9 \ a \ 5 \ c \ 2 \ b \ 7 \ c \ 8 \]

Finally, the values are reduced to produce the output:

\[ r_1 s_1, r_2 s_2, r_3 s_3 \]

An important detail to note is that reducers process keys in sorted order.

 logical view
MapReduce API*

Mapper<$K_{in}, V_{in}, K_{out}, V_{out}>$

void setup(Mapper.Context context)
**Called once at the start of the task**
void map($K_{in}$ key, $V_{in}$ value, Mapper.Context context)
**Called once for each key/value pair in the input split**
void cleanup(Mapper.Context context)
**Called once at the end of the task**

Reducer<$K_{in}, V_{in}, K_{out}, V_{out}>$/Combiner<$K_{in}, V_{in}, K_{out}, V_{out}>$

void setup(Reducer.Context context)
**Called once at the start of the task**
void reduce($K_{in}$ key, Iterable<$V_{in}$> values, Reducer.Context context)
**Called once for each key**
void cleanup(Reducer.Context context)
**Called once at the end of the task**

*Note that there are two versions of the API!*
Preserving State

Mapper object

- state
- setup
- map
- cleanup

Reducer object

- state
- setup
- reduce
- cleanup

- one object per task
- one call per input key-value pair
- one call per intermediate key

API initialization hook
API cleanup hook
Pseudo-Code

class Mapper {
  def setup() = {
    ...
  }

  def map(key: Long, value: String) = {
    ...
  }

  def cleanup() = {
    ...
  }
}
class Mapper {
    val counts = new Map()

    def map(key: Long, value: String) = {
        for (word <- tokenize(value)) {
            counts(word) += 1
        }
    }

    def cleanup() = {
        for ((k, v) <- counts) {
            emit(k, v)
        }
    }
}

Are combiners still needed?

Key idea: preserve state across input key-value pairs!
Design Pattern for Local Aggregation

“In-mapper combining”
Fold the functionality of the combiner into the mapper by preserving state across multiple map calls

Advantages
Speed
Why is this faster than actual combiners?

Disadvantages
Explicit memory management required
Potential for order-dependent bugs
# Performance

Word count on 10% sample of Wikipedia

<table>
<thead>
<tr>
<th></th>
<th>Running Time</th>
<th># Pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>~140s</td>
<td>246m</td>
</tr>
<tr>
<td>Histogram</td>
<td>~140s</td>
<td>203m</td>
</tr>
<tr>
<td>IMC</td>
<td>~80s</td>
<td>5.5m</td>
</tr>
</tbody>
</table>
Combiner Design

Combiners and reducers share same method signature
sometimes, reducers can serve as combiners
often, not...

Remember: combiner are optional optimizations
should not affect algorithm correctness
may be run 0, 1, or multiple times

Example: find average of integers associated with the same key
Computing the Mean: Version 1

class Mapper {
    def map(key: String, value: Int) = {
        emit(key, value)
    }
}

class Reducer {
    def reduce(key: String, values: Iterable[Int]) {
        for (value <- values) {
            sum += value
            cnt += 1
        }
        emit(key, sum/cnt)
    }
}

Why can’t we use reducer as combiner?
class Mapper {
    def map(key: String, value: Int) =
        emit(key, value)
}
class Combiner {
    def reduce(key: String, values: Iterable[Int]) = {
        for (value <- values) {
            sum += value
            cnt += 1
        }
        emit(key, (sum, cnt))
    }
}
class Reducer {
    def reduce(key: String, values: Iterable[Pair]) = {
        for ((s, c) <- values) {
            sum += s
            cnt += c
        }
        emit(key, sum/cnt)
    }
}

Why doesn’t this work?
class Mapper {
    def map(key: String, value: Int) =
        emit(key, (value, 1))
}

class Combiner {
    def reduce(key: String, values: Iterable[Pair]) = {
        for ((s, c) <- values) {
            sum += s
            cnt += c
        }
        emit(key, (sum, cnt))
    }
}

class Reducer {
    def reduce(key: String, values: Iterable[Pair]) = {
        for ((s, c) <- values) {
            sum += s
            cnt += c
        }
        emit(key, sum/cnt)
    }
}
class Mapper {
    val sums = new Map()
    val counts = new Map()

    def map(key: String, value: Int) = {
        sums(key) += value
        counts(key) += 1
    }

    def cleanup() = {
        for (key <- counts.keys) {
            emit(key, (sums(key), counts(key)))
        }
    }
}

Are combiners still needed?
## Performance

200m integers across three char keys

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>Scala</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>~120s</td>
<td>~120s</td>
</tr>
<tr>
<td>V3</td>
<td>~90s</td>
<td>~120s</td>
</tr>
<tr>
<td>V4</td>
<td>~60s</td>
<td>~90s (default HashMap)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>~70s (optimized HashMap)</td>
</tr>
</tbody>
</table>
MapReduce API*

Mapper<$K_{in}, V_{in}, K_{out}, V_{out}>$

void setup(Mapper.Context context)

Called once at the start of the task

void map($K_{in}$ key, $V_{in}$ value, Mapper.Context context)

Called once for each key/value pair in the input split

void cleanup(Mapper.Context context)

Called once at the end of the task

Reducer<$K_{in}, V_{in}, K_{out}, V_{out}>$ / Combiner<$K_{in}, V_{in}, K_{out}, V_{out}>$

void setup(Reducer.Context context)

Called once at the start of the task

void reduce($K_{in}$ key, Iterable<$V_{in}>$ values, Reducer.Context context)

Called once for each key

void cleanup(Reducer.Context context)

Called once at the end of the task

*Note that there are two versions of the API!
Algorithm Design: Running Example

Term co-occurrence matrix for a text collection

\[ M = N \times N \text{ matrix (} N = \text{vocabulary size)} \]

\[ M_{ij} : \text{number of times} \ i \text{ and} \ j \text{ co-occur in some context} \]

(for concreteness, let’s say context = sentence)

Why?

Distributional profiles as a way of measuring semantic distance

Semantic distance useful for many language processing tasks

Applications in lots of other domains
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 observations (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 Try: “Pairs”

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

Reducers sum up counts associated with these pairs
Use combiners!
class Mapper {
    def map(key: Long, value: String) = {
        for (u <- tokenize(value)) {
            for (v <- neighbors(u)) {
                emit((u, v), 1)
            }
        }
    }
}

class Reducer {
    def reduce(key: Pair, values: Iterable[Int]) = {
        for (value <- values) {
            sum += value
        }
        emit(key, sum)
    }
}
class Partitioner {
  def getPartition(key: Pair, value: Int, numTasks: Int): Int = {
    return key.left % numTasks
  }
}

Pairs: Pseudo-Code
One more thing...
“Pairs” Analysis

Advantages
Easy to implement, easy to understand

Disadvantages
Lots of pairs to sort and shuffle around (upper bound?)
Not many opportunities for combiners to work
Another Try: “Stripes”

Idea: group together pairs into an associative array

(a, b) → 1
(a, c) → 2
(a, d) → 5
(a, e) → 3
(a, f) → 2

Each mapper takes a sentence:
Generate all co-occurring term pairs
For each term, emit a → { b: \text{count}_b, c: \text{count}_c, d: \text{count}_d ... }

Reducers perform element-wise sum of associative arrays

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

Key idea: cleverly-constructed data structure brings together partial results
class Mapper {
    def map(key: Long, value: String) = {
        for (u <- tokenize(value)) {
            val map = new Map()
            for (v <- neighbors(u)) {
                map(v) += 1
            }
            emit(u, map)
        }
    }
}

class Reducer {
    def reduce(key: String, values: Iterable[Map]) = {
        val map = new Map()
        for (value <- values) {
            map += value
        }
        emit(key, map)
    }
}
“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 more heavyweight
- Overhead associated with data structure manipulations
- Fundamental limitation in terms of size of event space
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)

Comparison of "pairs" vs. "stripes" for computing word co-occurrence matrices

- "stripes" approach
- "pairs" approach

R² = 0.999
R² = 0.992
Effect of cluster size on "stripes" algorithm

relative size of EC2 cluster

running time (seconds)

graph with data points labeled "stripes" and "pairs"
Tradeoffs

Pairs:
Generates a *lot* more key-value pairs
Less combining opportunities
More sorting and shuffling
Simple aggregation at reduce

Stripes:
Generates fewer key-value pairs
More opportunities for combining
Less sorting and shuffling
More complex (slower) aggregation at reduce
Relative Frequencies

How do we estimate relative frequencies from counts?

\[ f(B|A) = \frac{N(A, B)}{N(A)} = \frac{N(A, B)}{\sum_{B'} N(A, B')} \]

Why do we want to do this?

How do we do this with MapReduce?
f(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 f(B|A)

\[
f(B|A) = \frac{N(A, B)}{N(A)} = \frac{N(A, B)}{\sum_{B'} N(A, B')}\]
f(B|A): “Pairs”

What’s the issue?
Computing relative frequencies requires marginal counts
But the marginal cannot be computed until you see all counts
Buffering is a bad idea!

Solution:
What if we could get the marginal count to arrive at the reducer first?
**f(B|A): “Pairs”**

Reducer holds this value in memory

\[(a, *) \rightarrow 32\]

\[(a, b_1) \rightarrow 3\]
\[(a, b_2) \rightarrow 12\]
\[(a, b_3) \rightarrow 7\]
\[(a, b_4) \rightarrow 1\]

\[
\frac{(a, b_1) \rightarrow 3}{32} \quad \frac{(a, b_2) \rightarrow 12}{32} \quad \frac{(a, b_3) \rightarrow 7}{32} \quad \frac{(a, b_4) \rightarrow 1}{32}
\]

**For this to work:**

- Emit extra \((a, *)\) for every \(b_n\) in mapper
- Make sure all a’s get sent to same reducer (use partitioner)
- Make sure \((a, *)\) comes first (define sort order)
- Hold state in reducer across different key-value pairs

\[
f(B|A) = \frac{N(A, B)}{N(A)} = \frac{N(A, B)}{\sum_{B'} N(A, B')}
\]
“Order Inversion”

Common design pattern:
Take advantage of sorted key order at reducer to sequence computations
Get the marginal counts to arrive at the reducer before the joint counts

Additional optimization
Apply in-memory combining pattern to accumulate marginal counts
Synchronization: Pairs vs. Stripes

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

Approach 2: data structures that bring partial results together
Each reducer receives all the data it needs to complete the computation
Illustrated by the “stripes” approach
Secondary Sorting

MapReduce sorts input to reducers by key

Values may be arbitrarily ordered

What if we want to sort value also?

E.g., \( k \rightarrow (v_1, r), (v_3, r), (v_4, r), (v_8, r) \ldots \)
Secondary Sorting: Solutions

Solution 1

Buffer values in memory, then sort
Why is this a bad idea?

Solution 2

“Value-to-key conversion” : form composite intermediate key, \((k, v_1)\)
Let the execution framework do the sorting
Preserve state across multiple key-value pairs to handle processing
Anything else we need to do?
Recap: Tools for Synchronization

- Preserving state in mappers and reducers
  Capture dependencies across multiple keys and values

- Cleverly-constructed data structures
  Bring partial results together

- Define custom sort order of intermediate keys
  Control order in which reducers process keys