Algorithms for MapReduce
Assignment 1 released

Cluster admin on vacation... in Florida:
“Greetings from hurricane hit winter haven”
Takeaways
Design MapReduce computations in pseudocode
Optimize a computation, with motivation
Patterns used

Less Important
These specific examples
Problem: Comparing Output

Alice’s Word Counts

<table>
<thead>
<tr>
<th>Word</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>20</td>
</tr>
<tr>
<td>hi</td>
<td>2</td>
</tr>
<tr>
<td>i</td>
<td>13</td>
</tr>
<tr>
<td>the</td>
<td>31</td>
</tr>
<tr>
<td>why</td>
<td>12</td>
</tr>
</tbody>
</table>

Bob’s Word Counts

<table>
<thead>
<tr>
<th>Word</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>20</td>
</tr>
<tr>
<td>hi</td>
<td>2</td>
</tr>
<tr>
<td>i</td>
<td>13</td>
</tr>
<tr>
<td>the</td>
<td>31</td>
</tr>
<tr>
<td>why</td>
<td>12</td>
</tr>
</tbody>
</table>
Problem: Comparing Output

Alice’s Word Counts

```
 a 20  
 hi 2   
 i 13
 the 31
 why 12
```

Bob’s Word Counts

```
 a 20  
 hi 2   
 i 13
 the 31
 why 12
```

Send words to a consistent place
Problem: Comparing Output

Alice's Word Counts

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>20</td>
</tr>
<tr>
<td>hi</td>
<td>2</td>
</tr>
<tr>
<td>i</td>
<td>13</td>
</tr>
<tr>
<td>the</td>
<td>31</td>
</tr>
<tr>
<td>why</td>
<td>12</td>
</tr>
</tbody>
</table>

Bob's Word Counts

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>20</td>
</tr>
<tr>
<td>hi</td>
<td>2</td>
</tr>
<tr>
<td>i</td>
<td>13</td>
</tr>
<tr>
<td>the</td>
<td>31</td>
</tr>
<tr>
<td>why</td>
<td>12</td>
</tr>
</tbody>
</table>

Map

Reduce

Send words to a consistent place: reducers
Problem: Comparing Output

Alice’s Word Counts

<table>
<thead>
<tr>
<th>Word</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>20</td>
</tr>
<tr>
<td>hi</td>
<td>2</td>
</tr>
<tr>
<td>the</td>
<td>31</td>
</tr>
</tbody>
</table>

Bob’s Word Counts

<table>
<thead>
<tr>
<th>Word</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>20</td>
</tr>
<tr>
<td>hi</td>
<td>2</td>
</tr>
<tr>
<td>the</td>
<td>31</td>
</tr>
<tr>
<td>why</td>
<td>12</td>
</tr>
</tbody>
</table>

Map

Unordered Alice/Bob

Reduce

Send words to a consistent place: reducers

Send words to a consistent place: reducers
Comparing Output Detail

Map: \((\text{word}, \text{count}) \mapsto (\text{word}, \text{student}, \text{count})\)

Reduce: Verify both values are present and match.
Deduct marks from Alice/Bob as appropriate.

\(^1\text{The mapper can tell Alice and Bob apart by input file name.}\)
Comparing Output Detail

Map: \((\text{word}, \text{count}) \mapsto (\text{word}, \text{student}, \text{count})\) ¹

Partition: By word

Sort: By \(\text{word}(\text{word}, \text{student})\)

Reduce: Verify both values are present and match.
Deduct marks from Alice/Bob as appropriate.

Exploit sort to control input order

¹The mapper can tell Alice and Bob apart by input file name.
Problem: Comparing Output

Alice’s Word Counts

Map

Reduce

<table>
<thead>
<tr>
<th>Word</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>20</td>
</tr>
<tr>
<td>hi</td>
<td>2</td>
</tr>
<tr>
<td>the</td>
<td>31</td>
</tr>
<tr>
<td>why</td>
<td>12</td>
</tr>
</tbody>
</table>

Bob’s Word Counts

Map

Reduce

<table>
<thead>
<tr>
<th>Word</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>20</td>
</tr>
<tr>
<td>hi</td>
<td>2</td>
</tr>
<tr>
<td>the</td>
<td>31</td>
</tr>
<tr>
<td>why</td>
<td>12</td>
</tr>
</tbody>
</table>

Send words to a consistent place: reducers
Pattern: Exploit the Sort

Without Custom Sort
Reducer buffers all students in RAM
⇒
Might run out of RAM

With Custom Sort
TA appears first, reducer streams through students.
Constant reducer memory.

We will give higher marks to scalable solutions
(even if yours runs on small data)
Problem: Averaging

We’re given temperature readings from cities:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Francisco</td>
<td>22</td>
</tr>
<tr>
<td>Edinburgh</td>
<td>14</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>23</td>
</tr>
<tr>
<td>Edinburgh</td>
<td>12</td>
</tr>
<tr>
<td>Edinburgh</td>
<td>9</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>21</td>
</tr>
</tbody>
</table>

Find the average temperature in each city.

Map: (city, temperature) \(\mapsto\) (city, temperature)

Reduce: Count, sum temperatures, and divide.
Problem: Averaging

We’re given temperature readings from cities:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Francisco</td>
<td>22</td>
</tr>
<tr>
<td>Edinburgh</td>
<td>14</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>23</td>
</tr>
<tr>
<td>Edinburgh</td>
<td>12</td>
</tr>
<tr>
<td>Edinburgh</td>
<td>9</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>21</td>
</tr>
</tbody>
</table>

Find the average temperature in each city.

Map: \((\text{city}, \text{temperature}) \mapsto (\text{city}, \text{temperature})\)
Combine: Same as reducer?
Reduce: Count, sum temperatures, and divide.
Problem: Averaging

We’re given temperature readings from cities:

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>San Francisco</td>
<td>22</td>
</tr>
<tr>
<td>Edinburgh</td>
<td>14</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>23</td>
</tr>
<tr>
<td>Edinburgh</td>
<td>12</td>
</tr>
<tr>
<td>Edinburgh</td>
<td>9</td>
</tr>
<tr>
<td>Los Angeles</td>
<td>21</td>
</tr>
</tbody>
</table>

Find the average temperature in each city.

**Map:** \((\text{city}, \text{temperature}) \mapsto (\text{city}, \text{count} = 1, \text{temperature})\)

**Combine:** Sum count and temperature fields.

**Reduce:** Sum count, sum temperatures, and divide.
Pattern: Combiners

Combiners reduce communication by aggregating locally. Many times they are the same as reducers (i.e. summing). …but not always (i.e. averaging).
PROGRAMMING FOR A DATA CENTRE
Programming for a data centre

- Understanding the design of warehouse-sized computes
  - Different techniques for a different setting
  - Requires quite a bit of rethinking
- MapReduce algorithm design
  - How do you express everything in terms of `map()`, `reduce()`, `combine()`, and `partition()`?
  - Are there any design patterns we can leverage?
Building Blocks
Funny story about sense of scale…

One server
DRAM: 16GB, 100ns, 20GB/s
Disk: 2TB, 10ms, 200MB/s

Local rack (80 servers)
DRAM: 1TB, 300us, 100MB/s
Disk: 160TB, 11ms, 100MB/s

Cluster (30 racks)
DRAM: 30TB, 500us, 10MB/s
Disk: 4.80PB, 12ms, 10MB/s
Scaling up vs. out

• No single machine is large enough
  – Smaller cluster of large SMP machines vs. larger cluster of commodity machines (e.g., 8 128-core machines vs. 128 8-core machines)

• Nodes need to talk to each other!
  – Intra-node latencies: ~100 ns
  – Inter-node latencies: ~100 µs

• Let’s model communication overhead
Modelling communication overhead

• Simple execution cost model:
  – Total cost = cost of computation + cost to access global data
  – Fraction of local access inversely proportional to size of cluster
  – \(n\) nodes (ignore cores for now)

\[
1 \text{ ms} + f \times [100 \text{ ns} \times (1/n) + 100 \text{ µs} \times (1 - 1/n)]
\]

• Light communication: \(f = 1\)
• Medium communication: \(f = 10\)
• Heavy communication: \(f = 100\)

• What is the cost of communication?
Overhead of communication

- Light communication
- Medium communication
- Heavy communication

The graph shows the normalised execution cost against the number of cores for different levels of communication overhead.
Seeks vs. scans

• Consider a 1TB database with 100 byte records
  – We want to update 1 percent of the records
• Scenario 1: random access
  – Each update takes ~30 ms (seek, read, write)
  – $10^8$ updates = ~35 days
• Scenario 2: rewrite all records
  – Assume 100MB/s throughput
  – Time = 5.6 hours(!)
• Lesson: avoid random seeks!
## Numbers everyone should know

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache reference</td>
<td>0.5</td>
</tr>
<tr>
<td>Branch mispredict</td>
<td>5</td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7</td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>25</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100</td>
</tr>
<tr>
<td>Send 2K bytes over 1 Gbps network</td>
<td>20,000</td>
</tr>
<tr>
<td>Read 1 MB sequentially from memory</td>
<td>250,000</td>
</tr>
<tr>
<td>Round trip within same datacenter</td>
<td>500,000</td>
</tr>
<tr>
<td>Disk seek</td>
<td>10,000,000</td>
</tr>
<tr>
<td>Read 1 MB sequentially from disk</td>
<td>20,000,000</td>
</tr>
<tr>
<td>Send packet CA → Netherlands → CA</td>
<td>150,000,000</td>
</tr>
</tbody>
</table>
Optimising computation

• The cluster management software orchestrates the computation
• But we can still optimise the computation
  – Just as we can write better code and use better algorithms and data structures
  – At all times confined within the capabilities of the framework
• Cleverly-constructed data structures
  – Bring partial results together
• Sort order of intermediate keys
  – Control order in which reducers process keys
• Partitioner
  – Control which reducer processes which keys
• Preserving state in mappers and reducers
  – Capture dependencies across multiple keys and values
Preserving State

Mapper object
- state
- setup
- map
- cleanup

Reducer object
- state
- setup
- reduce
- close

API initialization hook
one call per input key-value pair

API cleanup hook
one object per task

one call per intermediate key
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
Word count: baseline

class Mapper

    method map(docid a, doc d)
        for all term t in d do
            emit(t, 1);

class Reducer

    method reduce(term t, counts [c1, c2, ...])
        sum = 0;
        for all counts c in [c1, c2, ...] do
            sum = sum + c;
        emit(t, sum);
class Mapper
    
    method map(docid a, doc d)
        H = associative_array(term \rightarrow count)
        for all term t in d do
            H[t]++;
        for all term t in H[t] do
            emit(t, H[t]);

Local aggregation reduces further computation
class Mapper

    method initialise()
    H = associative_array(term \rightarrow count);

    method map(docid a, doc d)
    for all term t in d do
        H[t]++;

    method close()
    for all term t in H[t] do
        emit(t, H[t]);

Compute sums across documents!
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
Combiner design

• Combiners and reducers share same method signature
  – Effectively they are map-side reducers
  – Sometimes, reducers can serve as combiners
  – Often, not…

• Remember: combiners are optional optimisations
  – Should not affect algorithm correctness
  – May be run 0, 1, or multiple times

• Example: find average of integers associated with the same key
Algorithm design: term co-occurrence

• 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
    (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
Using MapReduce for large counting problems

- Term co-occurrence matrix for a text collection is a 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
    method map(docid a, doc d)
        for all w in d do
            for all u in neighbours(w) do
                emit(pair(w, u), 1);

class Reducer
    method reduce(pair p, counts [c1, c2, ...])
        sum = 0;
        for all c in [c1, c2, ...] do
            sum = sum + c;
        emit(p, sum);
Analysing pairs

• 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: count_b, c: count_c, d: count_d … }

•Reducers perform element-wise sum of associative arrays

a → { b: 1, c: 2, d: 5, e: 3, f: 2 }
a → { b: 1, c: 2, d: 7, e: 3, f: 2 }

Cleverly-constructed data structure brings together partial results
Stripes: pseudo-code

class Mapper
    method map(docid a, doc d)
        for all w in d do
            H = associative_array(string → integer);
            for all u in neighbours(w) do
                H[u]++;
            emit(w, H);

class Reducer
    method reduce(term w, stripes [H1, H2, ...])
        H_f = associative_array(string → integer);
        for all H in [H1, H2, ...] do
            sum(H_f, H); // sum same-keyed entries
            emit(w, H_f);
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
  – Fundamental limitation in terms of size of event space
Comparison of "pairs" vs. "stripes" for 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)
Effect of cluster size on "stripes" algorithm

Relative size of EC2 cluster

Running time (seconds)

Size of EC2 cluster (number of slave instances)

Relative speedup

$R^2 = 0.997$
Distributed Grep

Mapper    Keep lines matching “secret”
Reducer    NONE

Tip: save bandwidth by skipping reducers entirely.
Efficiency Tips

- Avoid sending data over the network
- Balance work across machines
- Use constant/bounded memory in each machine
- Combiners can help (but not if the keys are unique)
- Use secondary sort to order data for you
- Less computation in mappers or reducers
- ...
Debugging at scale

• Works on small datasets, won’t scale… why?
  – Memory management issues (buffering and object creation)
  – Too much intermediate data
  – Mangled input records

• Real-world data is messy!
  – There’s no such thing as consistent data
  – Watch out for corner cases
  – Isolate unexpected behavior, bring local
Summary

• Further delved into computing using MapReduce
• Introduced map-side optimisations
• Discussed why certain things may not work as expected
• Need to be really careful when designing algorithms to deploy over large datasets
• What seems to work on paper may not be correct when distribution/parallelisation kick in