Algorithms for MapReduce
Assignment 1 released
Due 16:00 on 20 October
Correctness is not enough!
Most marks are for efficiency.
Combining, Sorting, and Partitioning

...and algorithms exploiting these options.

Important: learn and apply optimization tricks.
Less important: these specific examples.
Last lecture: hash table has unbounded size

```
#!/usr/bin/python3
import sys
def spill(cache):
    for word, count in cache.items():
        print(word + " \t" + str(count))

    cache = {}
    for line in sys.stdin:
        for word in line.split():
            cache[word] = cache.get(word, 0) + 1
    spill(cache)
```

Combiners
Partition and Sort
Pairs vs Stripes
Solution: bounded size

```python
#!/usr/bin/python3
import sys
def spill(cache):
    for word, count in cache.items():
        print(word + " \t " + str(count))

cache = {}
for line in sys.stdin:
    for word in line.split():
        cache[word] = cache.get(word, 0) + 1
    if (len(cache) >= 10):  # Limit 10 entries
        spill(cache)
        cache.clear()
spill(cache)
```

Combiners
- Partition and Sort
- Pairs vs Stripes
Combiners formalize the local aggregation we just did:
Specifying a Combiner

Hadoop has built-in support for combiners:

```
hadoop jar hadoop-streaming-2.7.3.jar
  -files count_map.py,count_reduce.py
  -input /data/assignments/ex1/webSmall.txt
  -output /user/$USER/combined
  -mapper count_map.py
  -combiner count_reduce.py
  -reducer count_reduce.py
```

Run Hadoop
Copy to workers
Read text file
Write here
Simple mapper
Combiner sums
Reducer sums
Hadoop has built-in support for combiners:

```shell
hadoop jar hadoop-streaming-2.7.3.jar
-files count_map.py,count_reduce.py
-input /data/assignments/ex1/webSmall.txt
-output /user/$USER/combined
-mapper count_map.py
-combiner count_reduce.py
-reducer count_reduce.py
```

Run Hadoop
Copy to workers
Read text file
Write here
Simple mapper
Combiner sums
Reducer sums

How is this implemented?

Combiners
Partition and Sort
Pairs vs Stripes
Mapper’s Initial Sort

- **Map**
- **Partition (aka Shard)**
  - RAM buffer
    - **Sort**
    - **Combine**
      - Disk
    - RAM buffer
      - **Sort**
      - **Combine**
        - Disk
  - **Assign destination reducer**
  - **Remember what fits in RAM**
  - **Sort batch in RAM**
  - **Optional combiner**

**Combiners**
- **Partition and Sort**
- **Pairs vs Stripes**
Merge Sort

When the mapper runs out of RAM, it spills to disk.

Chunks of sorted data called “spills”.

Mappers merge their spills into one per reducer.
Reducers merge input from multiple mappers.

```
<table>
<thead>
<tr>
<th>Spill 0</th>
<th>Spill 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>a   3</td>
<td>a   5</td>
</tr>
<tr>
<td>→ c 4</td>
<td>b   9</td>
</tr>
<tr>
<td>d   2</td>
<td>→ c 6</td>
</tr>
</tbody>
</table>

Combiner
```

```
| a   8         |
| b   9         |
| → c 10        |
| ...           |
```
Combiners optimize merge sort and reduce network traffic. They may run in:

- Mapper initial sort
- Mapper merge
- Reducer merge
Combiner FAQ

Hadoop might not run your combiner at all!

Combiners will see a mix of mapper and combiner output.

Hadoop won’t partition or sort combiner output again.
⇒ Don’t change the key.
Hadoop sorts before combining

⇒ Duplicate keys are sorted ⇒ slow

Our in-mapper implementation used a hash table.
Also reduces Java ↔ Python overhead.

In-mapper is usually faster, but we’ll let you use either one.
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:** (city, temperature) \(\rightarrow\) (city, 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: \((\text{city, temperature}) \mapsto (\text{city, count} = 1, \text{temperature})\)

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).
Custom Partitioner and Sorting Function
Mapper’s Initial Sort

Map

Partition (aka Shard)

RAM buffer

Sort

Combine

Disk

Custom partitioner

Custom sort function

Combiners
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>
<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>the</td>
<td>31</td>
</tr>
<tr>
<td>why</td>
<td>12</td>
</tr>
</tbody>
</table>
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>

Send words to a consistent place
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>
```

Map

Reduce

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

<table>
<thead>
<tr>
<th>Alice’s Word Counts</th>
<th>Bob’s Word Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>a 20</td>
<td>a 20</td>
</tr>
<tr>
<td>hi 2</td>
<td>hi 2</td>
</tr>
<tr>
<td>the 31</td>
<td>the 31</td>
</tr>
<tr>
<td>why 12</td>
<td>why 12</td>
</tr>
</tbody>
</table>

#### Map

<table>
<thead>
<tr>
<th>Alice’s Word Counts</th>
<th>Bob’s Word Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>a 20</td>
<td>a 20</td>
</tr>
<tr>
<td>hi 2</td>
<td>hi 2</td>
</tr>
<tr>
<td>the 31</td>
<td>the 31</td>
</tr>
<tr>
<td>why 12</td>
<td>why 12</td>
</tr>
</tbody>
</table>

#### Reduce

<table>
<thead>
<tr>
<th>Alice’s Word Counts</th>
<th>Bob’s Word Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>a 20</td>
<td>a 20</td>
</tr>
<tr>
<td>a 20</td>
<td>a 20</td>
</tr>
<tr>
<td>hi 2</td>
<td>hi 2</td>
</tr>
<tr>
<td>hi 2</td>
<td>hi 2</td>
</tr>
<tr>
<td>the 31</td>
<td>the 31</td>
</tr>
<tr>
<td>the 31</td>
<td>the 31</td>
</tr>
<tr>
<td>i 13</td>
<td>i 13</td>
</tr>
<tr>
<td>i 13</td>
<td>i 13</td>
</tr>
<tr>
<td>why 12</td>
<td>why 12</td>
</tr>
<tr>
<td>why 12</td>
<td>why 12</td>
</tr>
</tbody>
</table>

Send words to a consistent place: reducers

- Unordered Alice/Bob
Comparing Output Detail

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

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

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

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

Partition: By word

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

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

Exploit sort to control input order

\(^1\)The mapper can tell Alice and Bob apart by input file name.
# Problem: Comparing Output

<table>
<thead>
<tr>
<th>Alice’s Word Counts</th>
<th>Bob’s Word Counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>a 20</td>
<td>a 20</td>
</tr>
<tr>
<td>hi 2</td>
<td>hi 2</td>
</tr>
<tr>
<td>the 31</td>
<td>the 31</td>
</tr>
<tr>
<td>why 12</td>
<td>why 12</td>
</tr>
</tbody>
</table>

## Map

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

## Reduce

- Ordered Alice/Bob

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.
Problem: Word Cooccurrence

Count pairs of words that appear in the same line.
First try: pairs

• Each mapper takes a sentence:
  – Generate all co-occurring term pairs
  – For all pairs, emit (a, b) → count
• Reducers sum up counts associated with these pairs
• Use combiners!
Pairs: pseudo-code

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 }
+ 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 \rightarrow integer);
            for all u in neighbours(w) do
                H[u]++;
            emit(w, H);

class Reducer
    method reduce(term w, stripes [H1, H2, ...])
        Hf = associative_array(string \rightarrow integer);
        for all H in [H1, H2, ...] do
            sum(H, Hf); // sum same-keyed entries
        emit(w, Hf);
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

$R^2 = 0.997$