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

```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
spill(cache)
```

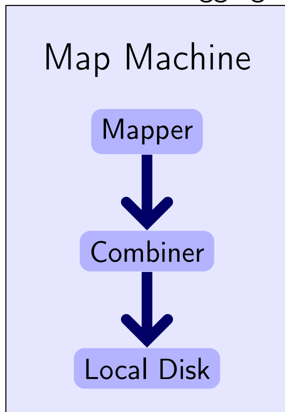
Solution: bounded size

```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

Combiners formalize the local aggregation we just did:



Specifying a Combiner

Hadoop bas built-in support for combiners:

```
hadoop jar hadoop-streaming-2.7.3.jar                 Run Hadoop
-files count_map.py,count_reduce.py                  Copy to workers
-input /data/assignments/ex1/webSmall.txt            Read text file
-output /user/$USER/combined                         Write here
-mapper count_map.py                                 Simple mapper
-combiner count_reduce.py                            Combiner sums
-reducer count_reduce.py                             Reducer sums
```

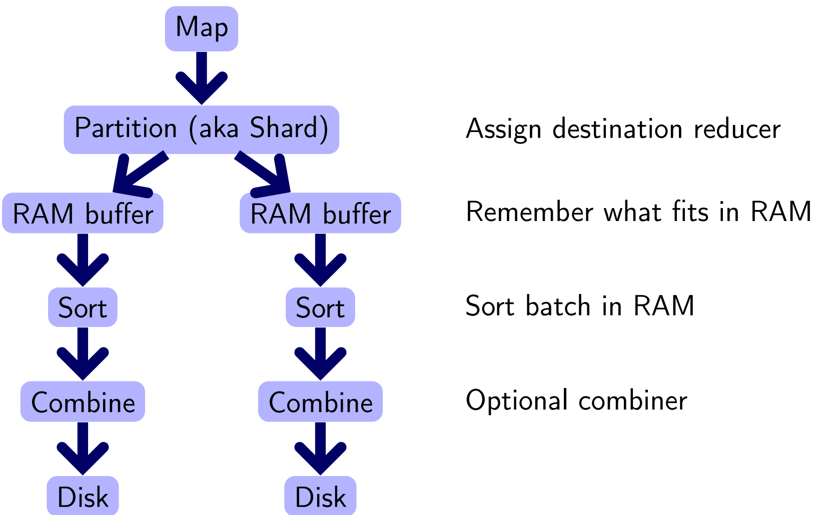
Specifying a Combiner

Hadoop bas built-in support for combiners:

```
hadoop jar hadoop-streaming-2.7.3.jar                Run Hadoop
-files count_map.py,count_reduce.py                  Copy to workers
-input /data/assignments/ex1/webSmall.txt            Read text file
-output /user/$USER/combined                         Write here
-mapper count_map.py                                 Simple mapper
-combiner count_reduce.py                            Combiner sums
-reducer count_reduce.py                             Reducer sums
```

How is this implemented?

Mapper's Initial Sort



Assign destination reducer

Remember what fits in RAM

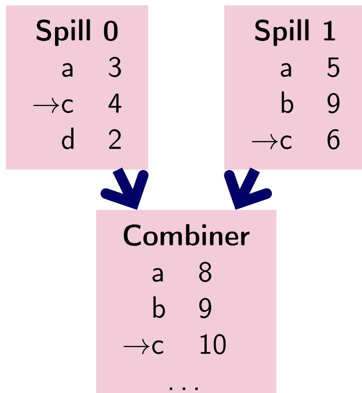
Sort batch in RAM

Optional combiner

Merge Sort

When the mapper runs out of RAM, it spills to disk.
$\Longrightarrow$ Chunks of sorted data called "spills".

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



Combiner Summary

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.
$\Longrightarrow$ Don't change the key.

Combiner Efficiency: Sort vs Hash Table

Hadoop sorts before combining
$\implies$ Duplicate keys are sorted $\implies$ slow

Our in-mapper implementation used a hash table.
Also reduces Java $\leftrightarrow$ Python overhead.

In-mapper is usually faster, but we'll let you use either one.

Problem: Averaging

We're given temperature readings from cities:

Key	Value
San Francisco	22
Edinburgh	14
Los Angeles	23
Edinburgh	12
Edinburgh	9
Los Angeles	21

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:

Key	Value
San Francisco	22
Edinburgh	14
Los Angeles	23
Edinburgh	12
Edinburgh	9
Los Angeles	21

Find the average temperature in each city.

Map: (city, temperature) $\mapsto$ (city, temperature)
Combine: Same as reducer?
Reduce: Count, sum temperatures, and divide.

Problem: Averaging

We're given temperature readings from cities:

Key	Value
San Francisco	22
Edinburgh	14
Los Angeles	23
Edinburgh	12
Edinburgh	9
Los Angeles	21

Find the average temperature in each city.

Map: (city, temperature) $\mapsto$ (city, count $= 1$, temperature)
Combine: Sum count and temperature fields.
Reduce: Sum count, sum temperatures, and divide.

Pattern: Combiners

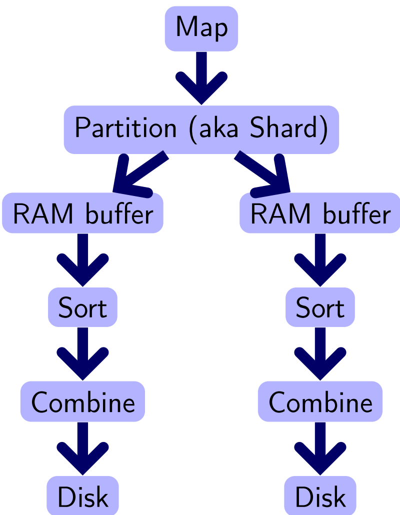
Combiners reduce communication by aggregating locally.

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



Custom partitioner

Custom sort function

Problem: Comparing Output

Alice's Word Counts

a	20
hi	2

i	13
the	31

why	12

Bob's Word Counts

a	20
why	12

hi	2
i	13
the	31

Problem: Comparing Output

Alice's Word Counts

a	20
hi	2

i	13
the	31

why	12

Bob's Word Counts

a	20
why	12

hi	2
i	13
the	31

a	20
a	20
hi	2
hi	2

the	31
the	31

i	13
i	13
why	12
why	12

Send words to a consistent place

Problem: Comparing Output

Alice's Word Counts

Bob's Word Counts

Map

a	20
hi	2

i	13
the	31

why	12

a	20
why	12

hi	2
i	13
the	31

Reduce

a	20
a	20
hi	2
hi	2

the	31
the	31

i	13
i	13
why	12
why	12

Send words to a consistent place: reducers

Problem: Comparing Output

Alice's Word Counts

Bob's Word Counts

Map

a	20
hi	2

i	13
the	31

why	12

a	20
why	12

hi	2
i	13
the	31

Reduce

a	20
a	20
hi	2
hi	2

the	31
the	31

i	13
i	13
why	12
why	12

Unordered Alice/Bob

Send words to a consistent place: reducers

Comparing Output Detail

Map: (word, count) $\mapsto$ (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: (word, count) $\mapsto$ (word, student, count) [1]

Partition: By word

Sort: By ~~word~~(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

Alice's Word Counts

Bob's Word Counts

Map

a	20
hi	2

i	13
the	31

why	12

Bob:

a	20
why	12

hi	2
i	13
the	31

Reduce

a	20
a	20
hi	2
hi	2

the	31
the	31

i	13
i	13
why	12
why	12

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 Coocurrence

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 → { b: 1, c: 2, d: 5, e: 3, f: 2 }
(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,        d: 5, e: 3 }
a → { b: 1, c: 2,  d: 2,        f: 2 }
a → { b: 2, 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, …])
    Hf = assoiative_array(string → integer);
    for all H in [H1, H2, …] do
      sum(Hf, H);    // 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

THE UNIVERSITY *of* EDINBURGH
informatics

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

"stripes" approach ■
"pairs" approach ●

$R^2 = 0.999$
$R^2 = 0.992$

running time (seconds): 0, 500, 1000, 1500, 2000, 2500, 3000, 3500, 4000

percentage of the APW corpus: 0, 20, 40, 60, 80, 100

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)

THE UNIVERSITY *of* EDINBURGH
informatics

Effect of cluster size on "stripes" algorithm

relative size of EC2 cluster

1x 2x 3x 4x

running time (seconds)

5000
4000
3000
2000
1000
0

relative speedup

4x
3x
2x
1x

R^2 = 0.997

10 20 30 40 50 60 70 80 90

size of EC2 cluster (number of slave instances)