MapReduce: Algorithm Design Patterns
Juliana Freire

Some slides borrowed from Jimmy Lin, Jeff Ullman, Jerome Simeon, and Jure Leskovec.

Designing Algorithms for MapReduce
- Need to adapt to a restricted model of computation
- Goals
  - Scalability: adding machines will make the algo run faster
  - Efficiency: resources will not be wasted
- The translation some algorithms into MapReduce isn’t always obvious

Towards Scalable Hadoop Algorithms
- 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

Tools for Synchronization
- 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
- Execute initialization and termination code before and after map/reduce tasks
Preserving State

Mapper object

Reducer object

configure

map

configure

one call per input key-value pair

close

API initialization hook

one call per intermediate key

API cleanup hook

One object per task

One call per intermediate key

API cleanup hook

Mapper object

Reducer object

configure

map

configure

one call per input key-value pair

close

API initialization hook

one call per intermediate key

API cleanup hook

One object per task

One call per intermediate key

API cleanup hook

Pattern 1: Local Aggregation

• Use combiners
• Do aggregation inside mappers

Word Count: Baseline

1. class Mapper
2. method Map(key, doc)
3. for all term t ∈ doc.
4. H[t] = H(t) + 1
5. Endmap

1. class Reducer
2. method Reduce(term, counts [t1, t2, ...])
3. sum = 0
4. for all count c ∈ counts [t1, t2, ...]
5. sum += c
6. Emit(term, count sum)

Suppose the collection has a total of \( n \) terms and \( d \) distinct terms. What are the communication costs for this mapreduce job?

What are the communication costs if we add a combiner?

Word Count: Aggregate in Mapper

H(dog) =\text{E} + 1
H(cat) =\text{E} + 1
H(dog) =\text{E} + 1

Are combiners still needed?

Word Count: Aggregate in Mapper (v. 2)

H(dog) =\text{E} + 1
H(cat) =\text{E} + 1
H(dog) =\text{E} + 1

Are combiners still needed?

DESIGN PATTERNS
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
  – Explicit control aggregation
  – Speed

  Why is this faster than actual combiners?

• Disadvantages
  – Explicit memory management required – if associative array grows too big, it will not fit in memory!
  – Preserving state across multiple key-value pairs may lead to potential for order-dependent bugs

• Not a problem for word count...

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
  – Explicit control aggregation
  – Speed

  Why is this faster than actual combiners?

No need to write all intermediate key-value pairs to disk!

Limiting Memory Usage

• To limit memory usage when using the in-mapper combining technique, block input key-value pairs and flush in-memory data structures periodically
  – E.g., counter variable that keeps track of the number of input key-value pairs that have been processed

• Memory usage threshold needs to be determined empirically: with too large a value, the mapper may run out of memory, but with too small a value, opportunities for local aggregation may be lost

• Note: Hadoop physical memory is split between multiple tasks that may be running on a node concurrently – difficult to coordinate resource consumption

How/when can local aggregation help with reduce stragglers?

When value distribution is skewed
Combiner Design

- Combiners and reducers share same method signature
  - Sometimes, reducers can serve as combiners
  
  *When is this the case?*

**Computing the Mean: Version 1**

```
class Mapper {
  method Map(string t, integer r) {
    sum = 0
    for all integer r in integers [r1,r2,...] do
      sum = sum + r
    return (string t, integer sum)
  }
}
```

```
class Reducer {
  method Reduce(string t, integers [r1,r2,...]) {
    sum = 0
    for all integer r in integers [r1,r2,...] do
      sum = sum + r
    return sum
  }
}
```

```
Mean(1; 2; 3; 4; 5) ?=?
Mean(Mean(1; 2), Mean(3; 4; 5))
```

Can we use the reducer as a combiner?

**How would you fix this?**

```
class Mapper {
  method Map(string t, integer r) {
    sum = 0
    for all integer r in integers [r1,r2,...] do
      sum = sum + r
    return (string t, integer sum)
  }
}
```

```
class Reducer {
  method Reduce(string t, integers [r1,r2,...]) {
    sum = 0
    for all integer r in integers [r1,r2,...] do
      sum = sum + r
    return sum
  }
}
```

```
Mean(1; 2; 3; 4; 5) ?=?
Mean(sum/5)
```

**Does this work?**

```
class Mapper {
  method Map(string t, integer r) {
    sum = 0
    for all integer r in integers [r1,r2,...] do
      sum = sum + r
    return (string t, integer sum)
  }
}
```

```
class Reducer {
  method Reduce(string t, integers [r1,r2,...]) {
    sum = 0
    for all integer r in integers [r1,r2,...] do
      sum = sum + r
    return sum
  }
}
```

```
Mean((user_id, session_id, session_length))
```

**When is the case?**

- Often, not...works only when reducer is commutative and associative

- Remember: combiners are an optional optimization
  - Should not affect algorithm correctness
  - May be run 0, 1, or multiple times

- Example: find average of all integers associated with the same key
  - Access logs: (user_id, session_id, session_length)
Computing the Mean: Version 3

1. class Mapper
2. method Map(docId, doc) 
3. for all term x in doc do 
4. emit (x, x) -> count
5. end for
6. end method

1. class Reducer
2. method Reduce(key, counts) 
3. emit (key, sum(counts))
4. end method

Pattern 2: Pairs and Stripes

• Keep track of joint events across a large number of observations
  – Common in natural language processing
  – Point-of-sale analysis to identify correlated product purchases
  • E.g., if customer buys milk she also buys bread
  • Assist in inventory management and product placement on store shelves

Example: Term co-occurrence matrix for a text collection

- M = N x 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)

Computing the Mean: Version 4

1. class Mapper
2. method Map(docId, doc) 
3. emit (docId, doc) 
4. end method

1. class Reducer
2. method Reduce(key, counts) 
3. emit (key, sum(counts))
4. end method

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!

MapReduce: 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)
  – Space requirement: n^2
  – Goal: keep track of interesting statistics about the events
• Basic approach
  – Mappers generate partial counts
  – Reducers aggregate partial counts
• Real-world English corpora can be hundreds of thousands of words, or even billions of words in web-scale collections

Pairs: Pseudo-Code

1. class Mapper
2. method Map(docId, doc) 
3. for all term x in doc do 
4. emit (x, doc) 
5. end for
6. end method

1. class Reducer
2. method Reduce(key, counts) 
3. emit (key, sum(counts))
4. end method

Note the use of a complex key.
"Pairs" Analysis

- Advantages
  - Easy to implement, easy to understand

- Disadvantages
  - Lots of pairs to sort and shuffle around
  - 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

  a → (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 → { b: count_b, c: count_c, d: count_d, ... }

  Reducers perform element-wise sum of associative arrays
  - (a, b) → 1
  - (a, c) → 2
  - (a, d) → 5
  - (a, e) → 3
  - (a, f) → 2

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

Stripes: Pseudo-Code

```
1: class Mapper
2:  method Map(word a, doc d)
3:     for all term w ∈ doc do
4:         H = new AssociativeArray
5:         Emit(Term w, Stripes H)
6:  end for
7: end class

2: class Reducer
3:  method Reduce(term w, stripes [H1, H2, H3, ... ])
4:     H = new AssociativeArray
5:     for all stripes H ∈ stripes [H1, H2, H3, ... ] do
6:         S(H, H) = Element-wise sum
7:     end for
8:     Emit(term w, stripe H)
9: end class
```

What are the advantages of stripes?

What about combiners?

- Both algorithms can benefit from the use of combiners, since the respective operations in their reducers (addition and element-wise sum of associative arrays) are both commutative and associative.

- Are combiners equally effective in both pairs and stripes?

"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 – higher serialization and de-serialization overhead
  - Fundamental limitation in terms of size of event space

Relative Frequencies

- Absolute counts do not take into account the fact that some words appear more frequently than others, e.g., "the"

- How do we estimate relative frequencies from counts? What proportion of time does B appear in the context of A?

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

- How do we do this with MapReduce?
f(B|A): “Stripes”

- For this to work:
  - Must emit extra (a, *) for every b, in mapper
  - Must make sure all a’s get sent to same reducer (use partitioner)
  - Must make sure (a, *) comes first (define sort order)
  - Must hold state in reducer across different key-value pairs

f(B|A): “Pairs”

- Reducer holds marginal value in memory
  - (a, b) → 3
  - (a, b) → 12
  - (a, b) → 7
  - (a, b) → 1
  - ... ... ... ...

- For this to work:
  - Must emit extra (a, *) for every b, in mapper
  - Must make sure all a’s get sent to same reducer (use partitioner)
  - Must make sure (a, *) comes first (define sort order)
  - Must hold state in reducer across different key-value pairs

Pattern 3: “Order Inversion”

- Common design pattern
  - Computing relative frequencies requires marginal counts
  - But marginal cannot be computed until you see all counts
  - Buffering is a bad idea!
  - Trick: getting the marginal counts to arrive at the reducer before the joint counts

Pattern 4: Secondary Sorting

- MapReduce sorts input to reducers by key
  - Values may be arbitrarily ordered
  - What if want to sort values also?
    - E.g., k → (v₁, r₁), (v₂, r₂), (v₃, r₃), (v₄, r₄)...

Synchronization: Pairs vs. Stripes

- Approach 1: turn synchronization into an ordering problem
  - Sort keys into correct order of computation
  - Partition key space so that each reducer gets the appropriate set of partial results
  - Hold state in reducer across multiple key-value pairs to perform computation
    - Illustrated by the “pairs” approach
- Approach 2: construct 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: Solutions

- **Solution 1:**
  - Buffer values in memory, then sort
  - Why is this a bad idea?
- **Solution 2:**
  - "Value-to-key conversion" design pattern: form composite intermediate key, \((k, v_i)\)
  - Let 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

- Cleverly-constructed data structures
  - Bring data 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

Issues and Tradeoffs

- 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
- Local aggregation
  - Opportunities to perform local aggregation varies
  - Combiners make a big difference
  - Combiners vs. in-mapper combining
  - RAM vs. disk vs. network

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!
  - Word count: how many unique words in Wikipedia?
  - There’s no such thing as “consistent data”
  - Watch out for corner cases
  - Isolate unexpected behavior

Cost Measures for Algorithms

1. Communication cost = total I/O of all processes.
2. Elapsed communication cost = max of I/O along any path.
3. (Elapsed) computation costs analogous, but count only running time of processes.
Example: Cost Measures

- For a map-reduce algorithm:
  - Communication cost = input file size + 2 × (sum of the sizes of all files passed from Map processes to Reduce processes) + the sum of the output sizes of the Reduce processes.
  - Elapsed communication cost is the sum of the largest input + output for any map process, plus the same for any reduce process.

What Cost Measures Mean

- Either the I/O (communication) or processing (computation) cost dominates.
  - Ignore one or the other.
- Total costs tell what you pay in rent from your friendly neighborhood cloud.
- Elapsed costs are wall-clock time using parallelism.

JOINS IN MAPREDUCE

Join By Map-Reduce

- Our first example of an algorithm in this framework is a map-reduce example.
- Compute the natural join $R(A,B) \bowtie S(B,C)$.
- $R$ and $S$ each are stored in files.
- Tuples are pairs (a,b) or (b,c).

Map-Reduce Join – (2)

- Use a hash function $h$ from B-values to 1..k.
- A Map process turns input tuple $R(a,b)$ into key-value pair $(b,(a,R))$ and each input tuple $S(b,c)$ into $(b,(c,S))$.

Map-Reduce Join – (3)

- Map processes send each key-value pair with key $b$ to Reduce process $h(b)$.
  - Hadoop does this automatically; just tell it what $k$ is.
- Each Reduce process matches all the pairs $(a,R)$ with all $(b,(c,S))$ and outputs $(a,b,c)$. 
Cost of Map-Reduce Join

- Total communication cost = $O(|R|+|S|+|R \bowtie S|)$.
- Elapsed communication cost = $O(s)$.
  - We’re going to pick $k$ and the number of Map processes so I/O limit $s$ is respected.
  - We put a limit $s$ on the amount of input or output that any one process can have. $s$ could be:
    - What fits in main memory
    - What fits on local disk
  - With proper indexes, computation cost is linear in the input + output size.
  - So computation costs are like communication costs.

Three-Way Join

- We shall consider a simple join of three relations, the natural join $R(A,B) \bowtie S(B,C) \bowtie T(C,D)$.
- One way: cascade of two 2-way joins, each implemented by map-reduce.
- Fine, unless the 2-way joins produce large intermediate relations.

Example: Large Intermediate Relations

- $A =$ “good pages”; $B, C =$ “all pages”; $D =$ “spam pages.”
- $R, S, T$ each represent links.
- 3-way join = “path of length 3 from good page to spam page.”
- $R \bowtie S =$ paths of length 2 from good page to any; $S \bowtie T =$ paths of length 2 from any page to spam page.

Another 3-Way Join

- Reduce processes use hash values of entire $S(B,C)$ tuples as key.
- Choose a hash function $h$ that maps $B$- and $C$-values to $k$ buckets.
- There are $k^2$ Reduce processes, one for each $(B$-bucket, $C$-bucket) pair.

Mapping for 3-Way Join

- We map each tuple $S(b,c)$ to $(h(b), h(c), (S, b, c))$.
- We map each $R(a,b)$ tuple to $(h(b), y, (R, a, b))$ for all $y = 1, 2, \ldots, k$.
- We map each $T(c,d)$ tuple to $(x, h(c), (T, c, d))$ for all $x = 1, 2, \ldots, k$.

Assigning Tuples to Reducers

- Keys
- Values
- Diagram showing how tuples are assigned to reducers based on hash values.
Job of the Reducers

- Each reducer gets, for certain B-values $b$ and C-values $c$:
  1. All tuples from $R$ with $B = b$,
  2. All tuples from $T$ with $C = c$, and
  3. The tuple $S(b, c)$ if it exists.
- Thus it can create every tuple of the form $(a, b, c, d)$ in the join.

RUNNING MAPREDUCE JOBS

Hadoop Workflow

1. Load data into HDFS
2. Develop code locally
3a. Go back to Step 2
3. Submit MapReduce job
4. Retrieve data from HDFS

On Amazon: With EC2

1. Allocate Hadoop cluster
2. Develop code locally
3a. Go back to Step 2
3. Submit MapReduce job
4. Retrieve data from HDFS
5. Clean up!

On Amazon: EC2 and S3

Copy from S3 to HDFS
Copy from HFDS to S3

Debugging Hadoop

- First, take a deep breath
- Start small, start locally
- Strategies
  - Learn to use the webapp
  - Where does println go?
  - Don’t use println, use logging
  - Throw RuntimeExceptions
References

- Data Intensive Text Processing with MapReduce, Lin and Dyer (Chapter 3)
- Mining of Massive Data Sets, Rajaraman et al. (Chapter 2)
- Hadoop tutorial: https://developer.yahoo.com/hadoop/tutorial/
  module4.html
- Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters
  http://labs.google.com/papers/mapreduce.html
  http://labs.google.com/papers/gfs.html