Data-intensive computing systems

Basic Algorithm Design Patterns

University of Verona
Computer Science Department

Damiano Carra

Acknowledgements

- Credits
  - *Part of the course material is based on slides provided by the following authors*
    - *Pietro Michiardi, Jimmy Lin*

Algorithm Design

- Developing algorithms involve:
  - Preparing the input data
  - Implement the mapper and the reducer
  - Optionally, design the combiner and the partitioner
- How to recast existing algorithms in MapReduce?
  - It is not always obvious how to express algorithms
  - Data structures play an important role
  - Optimization is hard
  - → The designer needs to "bend" the framework
- Learn by examples
  - "Design patterns"
  - Synchronization is perhaps the most tricky aspect

Algorithm Design (cont'd)

- Aspects that are not under the control of the designer
  - Where a mapper or reducer will run
  - When a mapper or reducer begins or finishes
  - Which input key-value pairs are processed by a specific mapper
  - Which intermediate key-value pairs are processed by a specific reducer
- Aspects that can be controlled
  - Construct data structures as keys and values
  - Execute user-specified initialization and termination code for mappers and reducers
  - Preserve state across multiple input and intermediate keys in mappers and reducers
  - Control the sort order of intermediate keys, and therefore the order in which a reducer will encounter particular keys
  - Control the partitioning of the key space, and therefore the set of keys that will be encountered by a particular reducer

Algorithm Design (cont'd)

- MapReduce jobs can be complex
  - Many algorithms cannot be easily expressed as a single MapReduce job
  - Decompose complex algorithms into a sequence of jobs
    - Requires orchestrating data so that the output of one job becomes the input to the next
  - Iterative algorithms require an external driver to check for convergence

- Basic design patterns
  - Local Aggregation
  - Pairs and Stripes
  - Relative frequencies
  - Inverted indexing

Local aggregation

Local aggregation

- Between the Map and the Reduce phase, there is the Shuffle phase
  - Transfer over the network the intermediate results from the processes that produced them to those that consume them
  - Network and disk latencies are expensive
    - Reducing the amount of intermediate data translates into algorithmic efficiency

- We have already talked about
  - Combiners
  - In-Mapper Combiners
  - In-Memory Combiners

In-Mapper Combiners: example

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1                ▷ Tally counts for entire document
6:         for all term t ∈ H do
7:             Emit(term t, count H{t})
```

In-Memory Combiners: example

```
1: class Mapper
2:     method Initialize
3:         H ← new AssociativeArray
4:     method Map(docid a, doc d)
5:         for all term t ∈ doc d do
6:             H{t} ← H{t} + 1                ▷ Tally counts across documents
7:     method Close
8:         for all term t ∈ H do
9:             Emit(term t, count H{t})
```

Algorithmic correctness with local aggregation

- Example
  - We have a large dataset where input keys are strings and input values are integers
  - We wish to compute the mean of all integers associated with the same key
    - In practice: the dataset can be a log from a website, where the keys are user IDs and values are some measure of activity
- Next, a baseline approach
  - We use an identity mapper, which groups and sorts appropriately input key-value pairs
  - Reducers keep track of running sum and the number of integers encountered
  - The mean is emitted as the output of the reducer, with the input string as the key

Example: basic MapReduce to compute the mean of values

1: **class** MAPPER
2: **method** MAP(string t, integer r)
3: EMIT(string t, integer r)

1: **class** REDUCER
2: **method** REDUCE(string t, integers $[r_1, r_2, \ldots]$)
3: $sum \leftarrow 0$
4: $cnt \leftarrow 0$
5: **for all** integer $r \in$ integers $[r_1, r_2, \ldots]$ **do**
6: $sum \leftarrow sum + r$
7: $cnt \leftarrow cnt + 1$
8: $r_{avg} \leftarrow sum/cnt$
9: EMIT(string t, integer r_{avg})

Using the combiners - Wrong approach

- Can we save bandwidth with the in-memory combiners?

Wrong InMemory Mapper

1. **class** MAPPER
2. **method** INITIALIZE
3. $S \leftarrow$ new ASSOCIATIVEARRAY
4. $C \leftarrow$ new ASSOCIATIVEARRAY
5. **method** MAP(string t, integer r)
6. $S\{t\} \leftarrow S\{t\} + r$
7. $C\{t\} \leftarrow C\{t\} + 1$
8. **method** CLOSE
9. **for all** term $t \in S$ **do**
10. EMIT(term t, double $S\{t\}/C\{t\}$)

Using the combiners - Wrong approach

Wrong Reducer

1. **class** REDUCER
2. **method** REDUCE(string t, doubles $[r_1, r_2, \ldots]$)
3. $sum \leftarrow 0$
4. $cnt \leftarrow 0$
5. **for all** double $r \in$ doubles $[r_1, r_2, \ldots]$ **do**
6. $sum \leftarrow sum + r$
7. $cnt \leftarrow cnt + 1$
8. $r_{avg} \leftarrow sum/cnt$
9. EMIT(string t, double r_{avg})

- Some operations are not distributive
  - `Mean`(1,2,3,4,5) ≠ `Mean`(`Mean`(1,2), `Mean`(3,4,5))
  - Hence: a combiner cannot output partial means and hope that the reducer will compute the correct final mean

Using the combiners – Correct approach

- To solve the problem
  - The Mapper partially aggregates results by separating the components to arrive at the mean
  - The **sum** and the **count** of elements are packaged into a **pair**
  - Using the same input string, the combiner emits the pair

Using the combiners – Correct approach

Correct InMemory Mapper

1. **class** MAPPER
2. **method** INITIALIZE
3. $S \leftarrow$ new ASSOCIATIVEARRAY
4. $C \leftarrow$ new ASSOCIATIVEARRAY
5. **method** MAP(string t, integer r)
6. $S\{t\} \leftarrow S\{t\} + r$
7. $C\{t\} \leftarrow C\{t\} + 1$
8. **method** CLOSE
9. **for all** term $t \in S$ **do**
10. EMIT(term t, pair $(S\{t\}, C\{t\})$)

Using the combiners – Correct approach

Correct Reducer

1. **class** REDUCER
2. **method** REDUCE(string t, pairs $[(s_1, c_1), (s_2, c_2), \ldots]$)
3. $sum \leftarrow 0$
4. $cnt \leftarrow 0$
5. **for all** pair $(s, c) \in$ pairs $[(s_1, c_1), (s_2, c_2), \ldots]$ **do**
6. $sum \leftarrow sum + s$
7. $cnt \leftarrow cnt + c$
8. $r_{avg} \leftarrow sum/cnt$
9. EMIT(string t, double r_{avg})

Pairs and stripes

Pairs and stripes

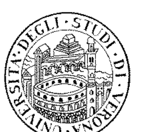
- A common approach in MapReduce: build complex keys
  - Data necessary for a computation are naturally brought together by the framework
- Two basic techniques:
  - Pairs: similar to the example on the average
  - Stripes: uses in-mapper memory data structures
- Next, we focus on a particular problem that benefits from these two methods

Problem statement

- Building word co-occurrence matrices for large corpora
  - The co-occurrence matrix of a corpus is a square $n \times n$ matrix
  - n is the number of unique words (i.e., the vocabulary size)
  - A cell m_{ij} contains the number of times the word w_i co-occurs with word w_j within a specific context
  - Context: a sentence, a paragraph a document or a window of m words
  - NOTE: the matrix may be symmetric in some cases
- Motivation
  - This problem is a basic building block for more complex operations
  - Estimating the distribution of discrete joint events from a large number of observations
  - Similar problem in other domains:
    - Customers who buy this tend to also buy that

Observations

- Space requirements
  - Clearly, the space requirement is $O(n^2)$, where n is the size of the vocabulary
  - For real-world (English) corpora n can be hundreds of thousands of words, or even billion of worlds
- So what's the problem?
  - If the matrix can fit in the memory of a single machine, then just use whatever naive implementation
  - Instead, if the matrix is bigger than the available memory, then paging would kick in, and any naive implementation would break



Word co-occurrence: the Pairs approach

Input to the problem: Key-value pairs in the form of a *docid* and a *doc*

- The mapper:
  - Processes each input document
  - Emits key-value pairs with:
    - Each co-occurring word pair as the key
    - The integer one (the count) as the value
  - This is done with two nested loops:
    - The outer loop iterates over all words
    - The inner loop iterates over all neighbors
- The reducer:
  - Receives pairs relative to co-occurring words
  - Computes an absolute count of the joint event
  - Emits the pair and the count as the final key-value output
    - Basically reducers emit the cells of the matrix

Word co-occurrence: the Pairs approach

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term w ∈ doc d do
4:             for all term u ∈ Neighbors(w) do
5:                 Emit(pair (w, u), count 1)        ▷ Emit count for each co-occurrence

1: class Reducer
2:     method Reduce(pair p, counts [c1, c2, ...])
3:         s ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             s ← s + c                              ▷ Sum co-occurrence counts
6:         Emit(pair p, count s)
```

Word co-occurrence: the Stripes approach

Input to the problem: Key-value pairs in the form of a *docid* and a *doc*

- The mapper:
  - Same two nested loops structure as before
  - Co-occurrence information is first stored in an associative array
  - Emit key-value pairs with words as keys and the corresponding arrays as values
- The reducer:
  - Receives all associative arrays related to the same word
  - Performs an element-wise sum of all associative arrays with the same key
  - Emits key-value output in the form of word, associative array
    - Basically, reducers emit rows of the co-occurrence matrix

Word co-occurrence: the Stripes approach

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term w ∈ doc d do
4:             H ← new AssociativeArray
5:             for all term u ∈ Neighbors(w) do
6:                 H{u} ← H{u} + 1            ▷ Tally words co-occurring with w
7:             Emit(Term w, Stripe H)

1: class Reducer
2:     method Reduce(term w, stripes [H1, H2, H3, ...])
3:         Hf ← new AssociativeArray
4:         for all stripe H ∈ stripes [H1, H2, H3, ...] do
5:             Sum(Hf, H)                      ▷ Element-wise sum
6:         Emit(term w, stripe Hf)
```

Pairs and Stripes, a comparison

- The pairs approach
  - Generates a large number of key-value pairs (also intermediate)
  - The benefit from combiners is limited, as it is less likely for a mapper to process multiple occurrences of a word
  - Does not suffer from memory paging problems
- The stripes approach
  - More compact
  - Generates fewer and shorted intermediate keys
    - The framework has less sorting to do
  - The values are more complex and have serialization/deserialization overhead
  - Greatly benefits from combiners, as the key space is the vocabulary
  - Suffers from memory paging problems, if not properly engineered

Relative frequencies

"Relative" Co-occurrence matrix

- Problem statement
  - Similar problem as before, same matrix
  - Instead of absolute counts, we take into consideration the fact that some words appear more frequently than others
    - Word w_i may co-occur frequently with word w_j simply because one of the two is very common
  - We need to convert absolute counts to relative frequencies $f(w_j \mid w_i)$
    - What proportion of the time does w_j appear in the context of w_i ?
- Formally, we compute:

$$f(w_j|w_i) = N(w_i,w_j) \ / \ \Sigma_{w'} N(w_i,w')$$

  - $N(\cdot, \cdot)$ is the number of times a co-occurring word pair is observed
  - The denominator is called the marginal

Computing relative frequencies: the stripes approach

- The stripes approach
  - In the reducer, the counts of all words that co-occur with the conditioning variable (w_i) are available in the associative array
  - Hence, the sum of all those counts gives the marginal
  - Then we divide the the joint counts by the marginal and we're done

```
1. class Reducer
2.   method Reduce(term w, stripes [H1, H2, ...])
3.     Hf ← new AssociativeArray
4.     for all stripe H ∈ stripes [H1, H2, ...] do
5.       Sum(Hf, H)                 // Element-wise sum
6.     cnt ← Count(Hf)              // ∑i Hf(ui)
7.     for all term u ∈ Hf do
8.       Hf{u} ← Hf{u}/cnt
9.     Emit(Term w, Stripe Hf)
```

Computing relative frequencies: the pairs approach

- The pairs approach
  - The reducer receives the pair (w_i , w_j) and the count
  - From this information alone it is not possible to compute $f(w_j \mid w_i)$
  - Fortunately, as for the mapper, also the reducer can preserve state across multiple keys
    - We can buffer in memory all the words that co-occur with w_i and their counts
    - This is basically building the associative array in the stripes method
- Problems:
  - Pairs that have the same first word can be processed by different reducers
    - E.g., (house, window) and (house, door)
  - The marginal is required before processing a set of pairs
    - E.g., we need to know the sum of all the occurrences of (house, *)

Computing relative frequencies: the pair approach

- We must define an appropriate partitioner
  - The default partitioner is based on the hash value of the intermediate key, modulo the number of reducers
  - For a complex key, the raw byte representation is used to compute the hash value
    - Hence, there is no guarantee that the pair (dog, aardvark) and (dog,zebra) are sent to the same reducer
  - What we want is that all pairs with the same left word are sent to the same reducer
- We must define the sort order of the pair
  - In this way, the keys are first sorted by the left word, and then by the right word (in the pair)
  - Hence, we can detect if all pairs associated with the word we are conditioning on (w_i) have been seen
  - At this point, we can use the in-memory buffer, compute the relative frequencies and emit

Computing relative frequencies: order inversion

- The key is to properly sequence data presented to reducers
  - If it were possible to compute the marginal in the reducer before processing the join counts, the reducer could simply divide the joint counts received from mappers by the marginal
  - The notion of "before" and "after" can be captured in the ordering of key-value pairs
  - The programmer can define the sort order of keys so that data needed earlier is presented to the reducer before data that is needed later

Computing relative frequencies: order inversion

- Recall that mappers emit pairs of co-occurring words as keys
- The mapper:
  - additionally emits a "special" key of the form $(w_i , *)$
  - The value associated to the special key is one, that represents the contribution of the word pair to the marginal
  - Using combiners, these partial marginal counts will be aggregated before being sent to the reducers
- The reducer:
  - We must make sure that the special key-value pairs are processed before any other key-value pairs where the left word is w_i
  - We also need to modify the partitioner as before, i.e., it would take into account only the first word

Computing relative frequencies: the pair approach

```
1. class Mapper
2.   method Map(docID a, doc d)
3.     for all term w ∈ doc d do
4.       for all term u ∈ Neighbors(w) do
5.         Emit(Pair (w, u), Integer 1)
6.         Emit(Pair (w, *), Integer 1)
```

```
1.  class Reducer
2.    method Initialize
3.      cnt ← 0
4.    method Reduce(pair p, integers [r1, r2, ...])
5.      if p.getSecond() == * then
6.        cnt ← 0
7.        for all integer r ∈ integers [r1, r2, ...] do
8.          cnt ← cnt + r
9.      else
10.       sum ← 0
11.       for all integer r ∈ integers [r1, r2, ...] do
12.         sum ← sum + r
13.       Emit(Pair p, double sum/cnt)
```

Note:
The partitioner is not shown here

Using in-mapper combiners

```
1.  class Mapper
2.    method Initialize
3.      H ← new AssociativeArray
4.    method Map(docID a, doc d)
5.      for all term w ∈ doc d do
6.        for all term u ∈ Neighbors(w) do
7.          Emit(Pair (w, u), Integer 1)
8.          H{w} ← H{w} + 1
9.    method Close
10.     for all term w ∈ H do
11.       Emit(Pair (w, *), Integer H{w})
```

Computing relative frequencies: order inversion

- Memory requirements:
  - Minimal, because only the marginal (an integer) needs to be stored
  - No buffering of individual co-occurring word
  - No scalability bottleneck

- Key ingredients for order inversion
  - Emit a special key-value pair to capture the marginal
  - Control the sort order of the intermediate key, so that the special key-value pair is processed first
  - Define a custom partitioner for routing intermediate key-value pairs
  - Preserve state across multiple keys in the reducer

Inverted indexing

Inverted indexing

❑ Quintessential large-data problem: Web search

- A web crawler gathers the Web objects and store them
- Inverted indexing
  - Given a term *t* → Retrieve relevant web objects that contains *t*
- Document ranking
  - Sort the relevant web objects

❑ Here we focus on the inverted indexing

- For each term *t*, the output is a list of documents and the number of occurrences of the term *t*

Inverted indexing: visual solution