Cloud Computing Lecture #3

More MapReduce

Jimmy Lin
The iSchool
University of Maryland

Wednesday, September 10, 2008

Material adapted from: Jimmy Lin. Scalable Language Processing Algorithms for the Masses: A Case Study in Computing Word Co-occurrence Matrices with MapReduce. Proceedings of EMNLP 2008.

This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States
See http://creativecommons.org/licenses/by-nc-sa/3.0/us/ for details

Today's Topics

- Synchronization and coordinating partial results
- Use of complex keys and values

The iSchool
University of Maryland

Managing Dependencies

- Remember: Mappers run in isolation
  - You have no idea in what order the mappers run
  - You have no idea on what node the mappers run
  - You have no idea when each mapper finishes
- Tools for synchronization:
  - Ability to hold state in reducer across multiple key-value pairs
  - Sorting function for keys
  - Partitioner
  - Cleverly-constructed data structures

The iSchool
University of Maryland

Motivating 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)
- Why?
  - Distributional profiles as a way of measuring semantic distance
  - Semantic distance useful for many language processing tasks

"You shall know a word by the company it keeps" (Firth, 1957)

e.g., Mohammad and Hirst (EMNLP, 2006)

The iSchool
University of Maryland

MapReduce: Large Counting Problems

- Term co-occurrence matrix for a text collection
  = specific instance of a large counting problem
  - A large event space (number of terms)
  - A large number of events (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?

The iSchool
University of Maryland

First Try: "Pairs"

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

Note: in all my slides, I denote a key-value pair as k → v

The iSchool
University of Maryland

"Pairs" Analysis

- Advantages
  - Easy to implement, easy to understand
- Disadvantages
  - Lots of pairs to sort and shuffle around (upper bound?)

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 }

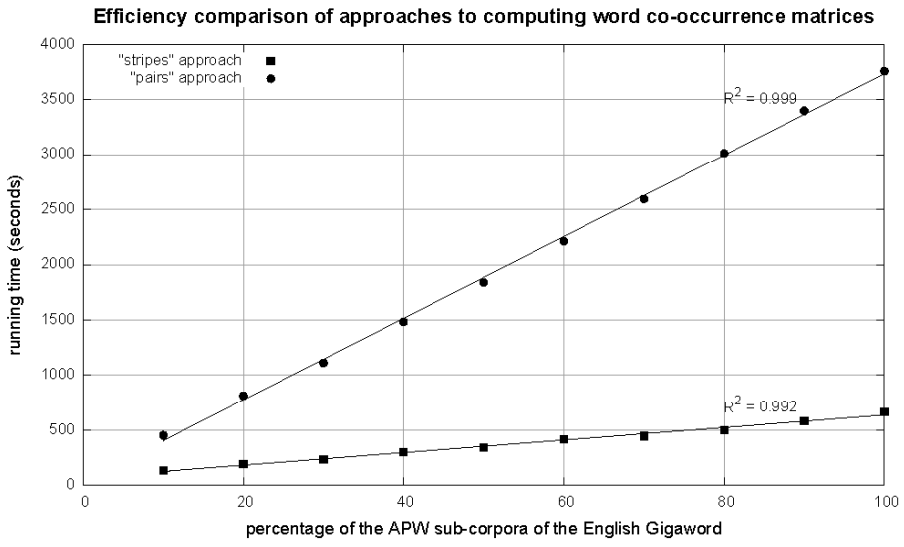
- Each mapper takes a sentence:
  - Generate all co-occurring term pairs
  - For each term, emit a → { b: $count_b$, c: $count_c$, d: $count_d$ … }
- Reducers perform element-wise sum of associative arrays

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


"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 is more heavyweight
  - Fundamental limitation in terms of size of event space

The iSchool
University of Maryland



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)

Conditional Probabilities

- How do we compute conditional probabilities from counts?

$$P(B \mid A) = \frac{\text{count}(A, B)}{\text{count}(A)} = \frac{\text{count}(A, B)}{\sum_{B'} \text{count}(A, B')}$$

- Why do we want to do this?
- How do we do this with MapReduce?

The iSchool
University of Maryland

P(B|A): "Pairs"

(a, *) → 32 **Reducer holds this value in memory**

(a, b_1) → 3
(a, b_2) → 12
(a, b_3) → 7
(a, b_4) → 1
…

(a, b_1) → 3 / 32
(a, b_2) → 12 / 32
(a, b_3) → 7 / 32
(a, b_4) → 1 / 32
…

- For this to work:
  - Must emit extra (a, *) for every b_n in mapper
  - Must make sure all a's get sent to same reducer (use Partitioner)
  - Must make sure (a, *) comes first (define sort order)

The iSchool
University of Maryland

P(B|A): "Stripes"

$a \rightarrow \{b_1:3, b_2:12, b_3:7, b_4:1, \ldots\}$

- Easy!
  - One pass to compute (a, *)
  - Another pass to directly compute P(B|A)

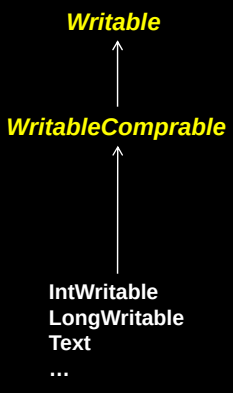
Synchronization in Hadoop

- 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 the pieces together"
  - Each reducer receives all the data it needs to complete the computation
  - Illustrated by the "stripes" approach

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
- Combiners make a big difference!
  - RAM vs. disk and network
  - Arrange data to maximize opportunities to aggregate partial results

Data Types in Hadoop



Defines a de/serialization protocol.
Every data type in Hadoop is a Writable.

Defines a sort order. All keys must be of this type (but not values).

Concrete classes for different data types.

Complex Data Types in Hadoop

- How do you implement complex data types?
- The easiest way:
  - Encoded it as Text, e.g., (a, b) = “a:b”
  - Use regular expressions to parse and extract data
  - Works, but pretty hack-ish
- The hard way:
  - Define a custom implementation of WritableComprable
  - Must implement: readFields, write, compareTo
  - Computationally efficient, but slow for rapid prototyping
- Alternatives:
  - Cloud[9] offers two other choices: Tuple and JSON

Questions?