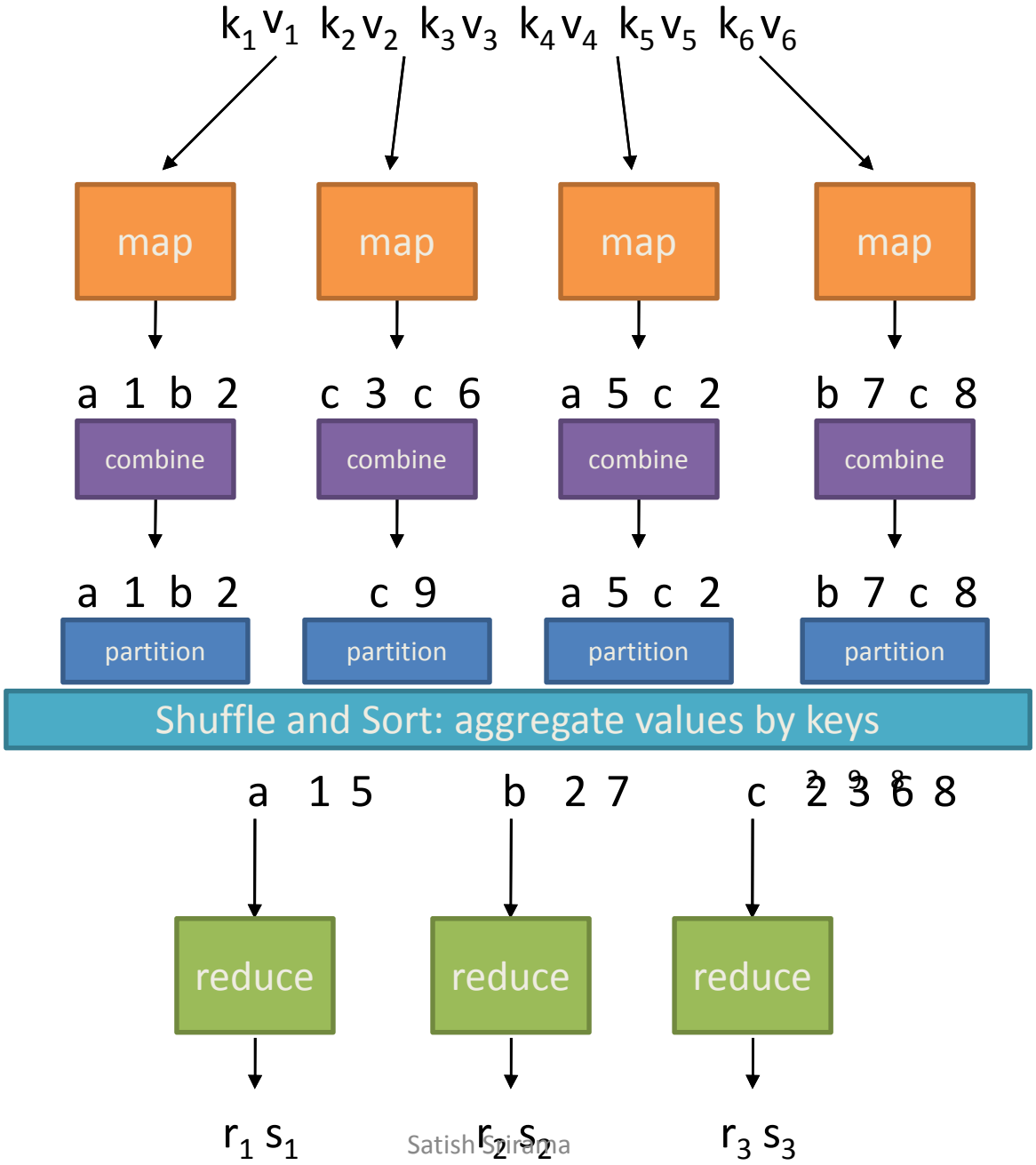
Basics of Cloud Computing – Lecture 5

MapReduce Algorithms

Satish Srirama

Outline

- MapReduce algorithms
- How to write MR algorithms


k1 v1 k2 v2 k3 v3 k4 v4 k5 v5 k6 v6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
combine
combine
combine
combine
a 1 b 2
c 9
a 5 c 2
b 7 c 8
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 3 6 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

Hadoop Usage Patterns

- Extract, transform, and load (ETL)
  - Perform aggregations, transformation, normalizations on the data (e.g. Log files) and load into RDBMS/ data mart
- Reporting and analytics
  - Run ad-hoc queries, analytics and data mining operations on large data
- Data processing pipelines
- Machine learning & Graph algorithms
  - Implement machine learning algorithms on huge data sets
  - Traverse large graphs and data sets, building models and classifiers

MapReduce Examples

- Distributed Grep
- Count of URL Access Frequency
- Reverse Web-Link Graph
- Term-Vector per Host
- Inverted Index
- Distributed Sort

MapReduce Jobs

- Tend to be very short, code-wise
  - IdentityReducer is very common
- “Utility” jobs can be composed
- Represent a data flow, more so than a procedure

Count of URL Access Frequency

- Processing web access logs
- Very similar to word count
- Map
  - processes logs of web page requests and outputs <URL, 1>
- Reduce
  - adds together all values
  - emits a <URL, total count> pairs

Distributed Grep

- Map
  - Emits a line if it matches a supplied pattern
- Reduce
  - Identity function
  - Just copies the supplied intermediate data to the output

Reverse Web-Link Graph

- Map
  - Outputs <*target*, *source*> pairs
  - for each link to a *target* URL found in a page named *source*.
- Reduce
  - Concatenates the list of all source URLs
  - Returns <*target*, list(*source*)>

Sort: Inputs

- A set of files, one value per line.
- Mapper key is file name, line number
- Mapper value is the contents of the line

Sort Algorithm

- Takes advantage of reducer properties:
  - (key, value) pairs are processed in order by key; reducers are themselves ordered
- Mapper: Identity function for value

  (k, v) -> (v, _)
- Reducer: Identity function (k', _) -> (k', "")

Sort: The Trick

- (key, value) pairs from mappers are sent to a particular reducer based on hash(key)
- Must pick the hash function for your data such that
  - K1 < K2 => hash(K1) < hash(K2)
- Used as a test of Hadoop's raw speed

Inverted Index: Inputs

- A set of files containing lines of text

- Mapper key is file name, line number
- Mapper value is the contents of the line

Inverted Index Algorithm

- Mapper: For each word in (file, words), map to (word, file)

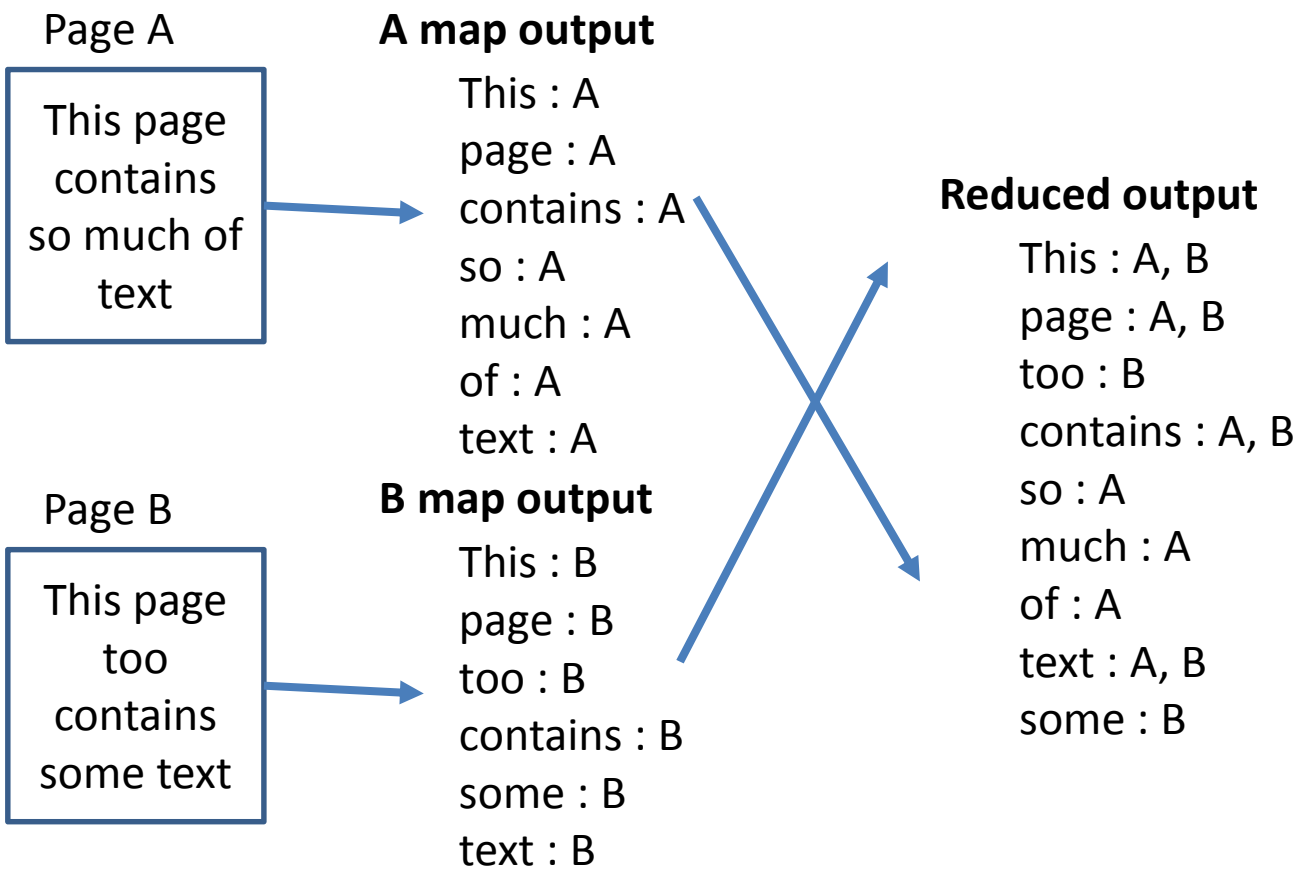
- Reducer: Identity function

Index MapReduce

- map(pageName, pageText):
  foreach word w in pageText:
    emit Intermediate(w, pageName);
  Done

- reduce(word, values):
  foreach pageName in values:
    AddToOutputList(pageName);
  Done
  emitFinal(FormattedPageListForWord);

Index: Data Flow



Let us focus much bigger problems

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

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)

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?

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!

“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

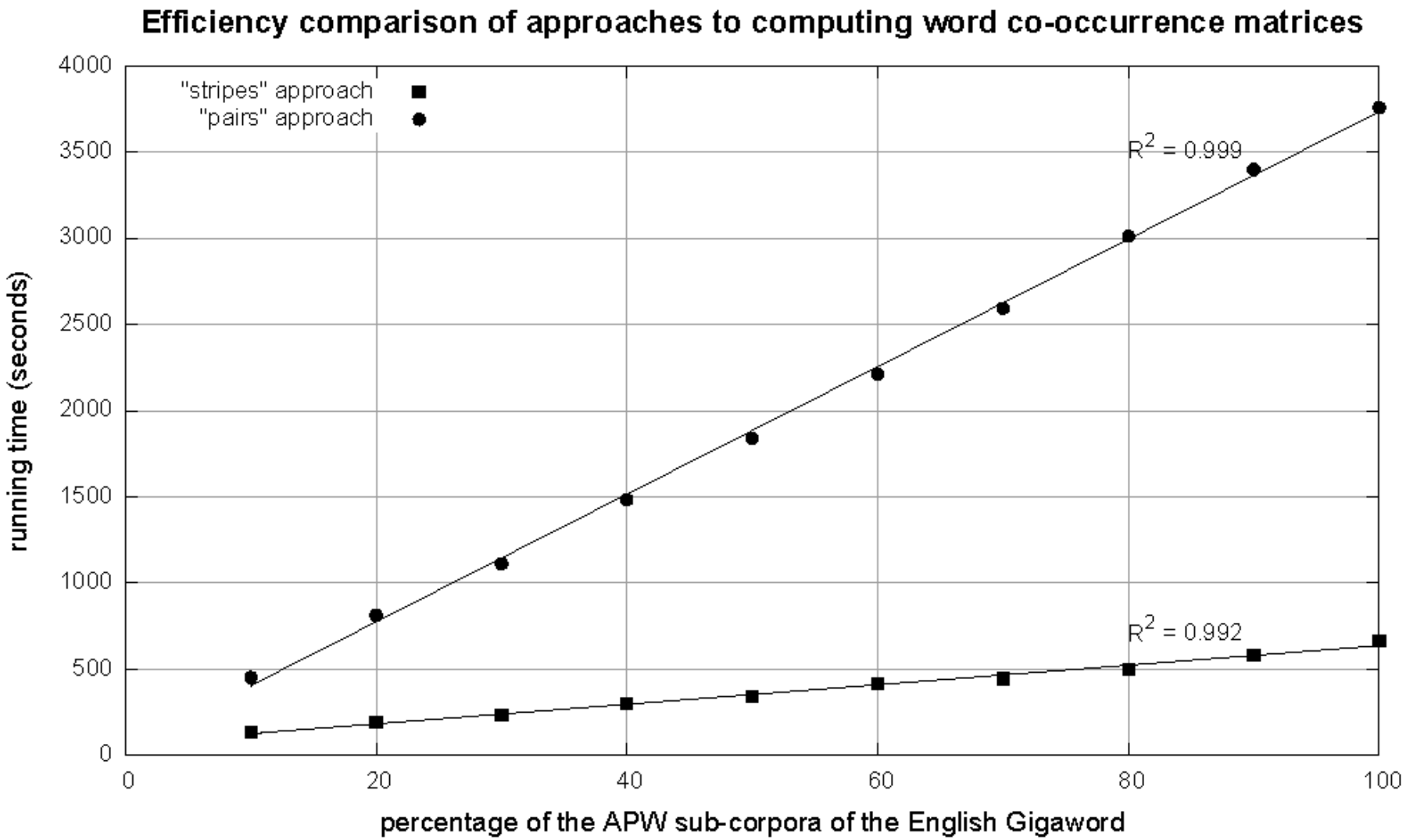
a → { b: 1, c: 2, d: 5, e: 3, f: 2 }

- 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



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?

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

- How do we do this with MapReduce?

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)

P(B|A): “Stripes”

a → {b_1:3, b_2 :12, b_3 :7, b_4 :1, … }

- 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

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

Lab related to the lecture

- MapReduce for data analysis
- Writing better MapReduce algorithms

Next Lecture

- Platform as a Service
  - Google AppEngine

References

- J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters”, OSDI'04: Sixth Symposium on Operating System Design and Implementation, Dec, 2004.
- Data-Intensive Text Processing with MapReduce

Authors: Jimmy Lin and Chris Dyer

http://lintool.github.io/MapReduceAlgorithms/MapReduce-book-final.pdf

Pages 50-57: Pairs and Stripes problem