Basics of Cloud Computing – Lecture 5

MapReduce Algorithms

Pelle Jakovits

Satish Srirama

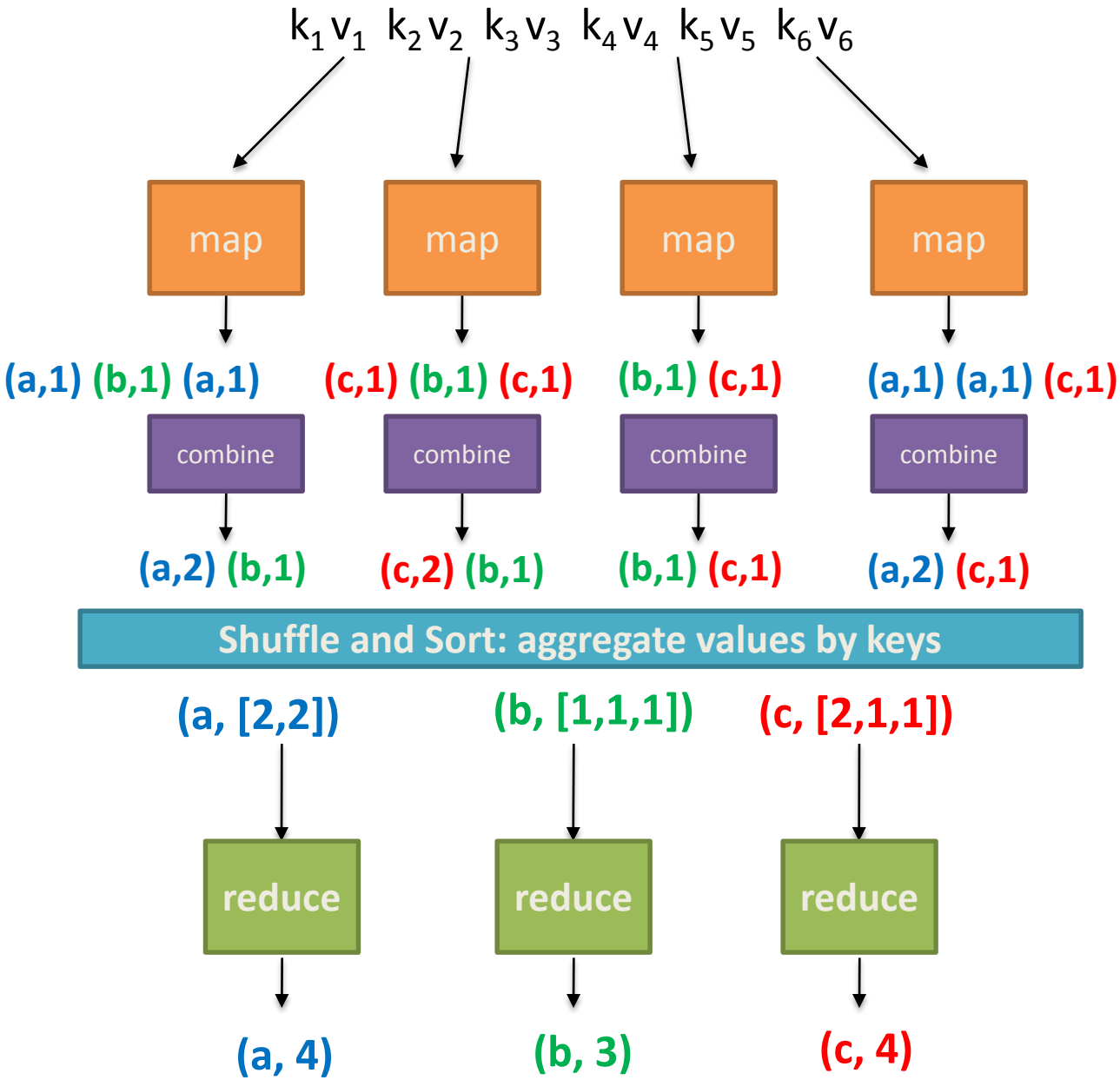
Some material adapted from slides by Jimmy Lin, Web-Scale Information Processing Applications course, University of Waterloo (licensed under Creation Commons Attribution 3.0 License)

Outline

- Recap of the MapReduce model
- Example MapReduce algorithms
- Designing MapReduce algorithms
  - How to represent everything using only Map, Reduce, Combiner and Partitioner tasks
  - Managing dependencies in data
  - Using complex data types

MapReduce model

- Programmers specify Map and Reduce functions:
  - `map (k, v) → (k', v')*`
    - Applies a user defined function on every input record
    - Values with the same key are grouped together before Reduce phase
  - `reduce (k', [v']) → (k'', v'')*`
    - Applies a user defined aggregation function on the list of values
- The execution framework handles everything else!
- Users have opportunity to also define:
  - **Partitioner -** Controls how keys are partitioned between reducers
    - `partition (k, nr. of partitions) → partition_id for k`
  - **Combiner -** Mini-reducer applied at the end of the map phase
    - `combine (k', [v']) → (k'', v'')*`


k1 v1 k2 v2 k3 v3 k4 v4 k5 v5 k6 v6
map
map
map
map
(a,1) (b,1) (a,1)
(c,1) (b,1) (c,1)
(b,1) (c,1)
(a,1) (a,1) (c,1)
combine
combine
combine
combine
(a,2) (b,1)
(c,2) (b,1)
(b,1) (c,1)
(a,2) (c,1)
Shuffle and Sort: aggregate values by keys
(a, [2,2])
(b, [1,1,1])
(c, [2,1,1])
reduce
reduce
reduce
(a, 4)
(b, 3)
(c, 4)

Typical Hadoop Use Cases

- **Extract, transform and load** (ETL) pipelines
  - Perform transformation, normalization, aggregations on the data
  - Load results into database or data warehouse
  - Ex: Sentiment analysis of review websites and social media data
- **Reporting and analytics**
  - Generate statistics, run ad-hoc queries and information retrieval tasks
  - Ex: Analyzing web clickstream, marketing, CRM, & email data
- **Machine learning**
  - Ex: Building recommender systems for behavioral targeting
  - Ex: Face similarity and recognition over large datasets of images
- **Graph algorithms**
  - Ex: Identifying trends and communities by analyzing social network graph data

Powered By Hadoop - https://wiki.apache.org/hadoop/PoweredBy

MapReduce Jobs

- Tend to be very short, code-wise
  - Identity Reducer is common
- Represent a data flow, rather than a procedure
  - Data „flows“ through Map and Reduce stages
- Can be composed into larger data processing pipelines
- Iterative applications may require repeating the same job multiple times
- Data must be partitioned across many reducers if it is large
- Data will be written into multiple output files if there are more than a single Reduce task

Different MapReduce input formats

- The input types of a MapReduce application are not fixed and depend on the input format that is used

InputFormat	Key	Value
TextInputFormat (Default)	Byte offset of the line (LongWritable)	Line contents Text
KeyValueInputFormat	User Defined Writable Object *e.g. PersonWritable*	User Defined Writable Object
WholeFileInputFormat	NullWritable	File contents (BytesWritable)
NLineInputFormat	Byte offset of the line block (LongWritable)	Contents of N lines (Text)
TableINputFormat (HBase)	Row Key	Value

Designing MapReduce algorithms

- General goal of a MapReduce algorithm:
  - How to produce desired **Output** from the **Input data**?
- To define a MapReduce algorithm, we need to define:
  1. **Map Function**
     - What is **Map Input (Key, Value)** pair
     - What is **Map Output (Key, Value)** pair
     - **Map Function: Input (Key, Value) → Output (Key, Value)**
  2. **Reduce Function**
     - What is Reduce **Input (Key, [Value])** pair
     - What is Reduce **Output (Key, Value)** pair
     - **Reduce Function: Input (Key, [Value]) → Output (Key, Value)**

Lets look at a few Example MapReduce algorithms

MapReduce Examples

- Counting URL Access Frequency
- Distributed Grep
- Distributed Sort
- Inverted Index
- Conditional Probabilities

Counting URL Access Frequency

- Process web access logs to count how often each URL was visited
  - **Input:** (LineOffset, Line)
  - **Output:** (URL, count)
- Very similar to the MapReduce WordCount algorithm
- **Map function**
  - Processes one log record at a time
  - Emit (URL, 1) if an URL appears in log record
- **Reduce function**
  - Sum together all values
  - Emit (URL, total_count) pair

Distributed Grep

- Distributed version of the Linux command line Grep command
- Find all rows in a set of text files that contain a supplied regular expression
  - **Input:** (LineOffset, Line)
  - **Output:** (LineOffset, Line)
- **Map function**
  - Emits a line **ONLY** if it matches the supplied regular expression
- **Reduce function**
  - Identity function
  - Emits all input data as (Key, Value) pairs without modifications

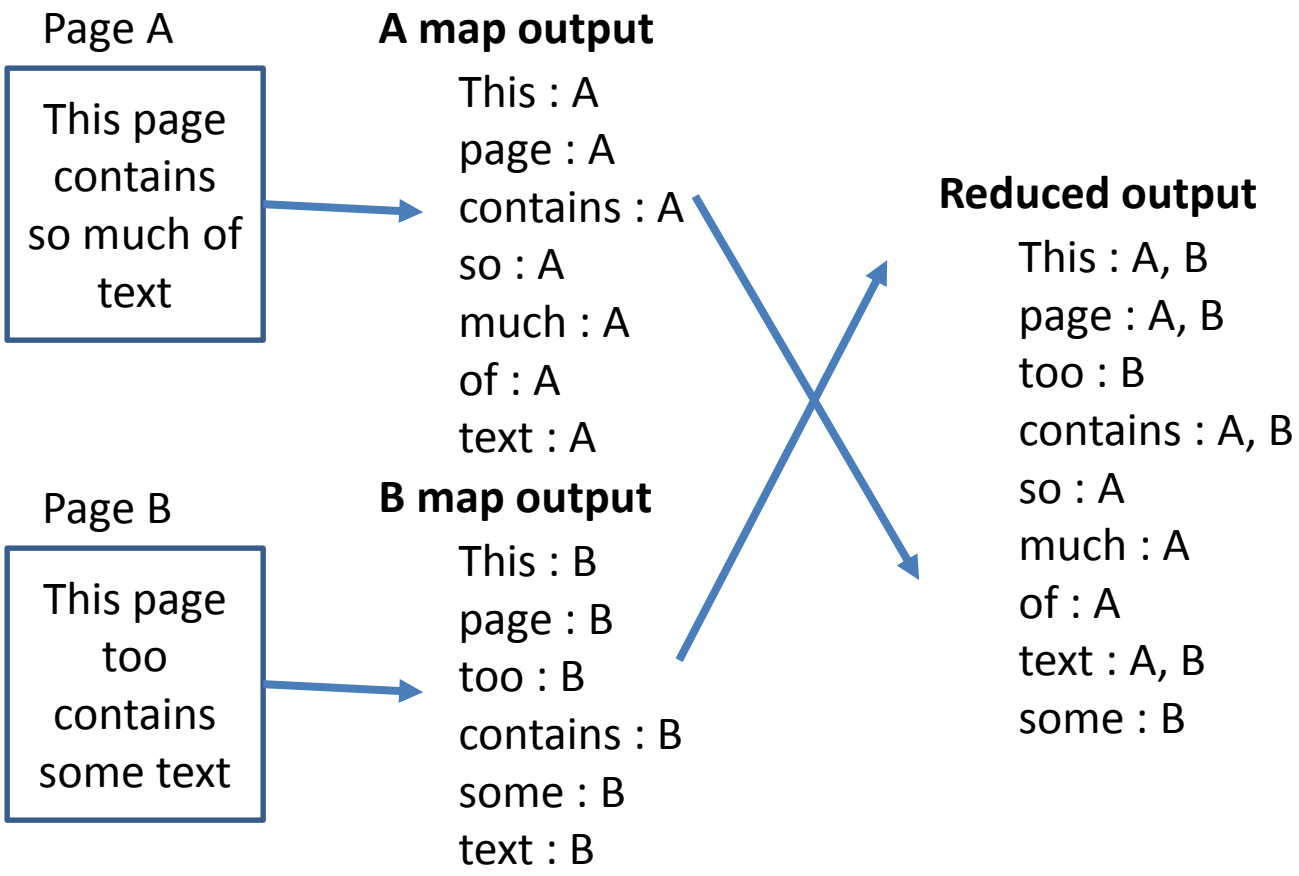
MapReduce Algorithm Design Process

1. Structure of the input data → Defines **Job Input (Key, Value)**
2. Desired result → Defines **Job Output (Key'', Value'')**
3. If the desired result can be computed **without shuffling data**:
   - **Map Function: Job Input (Key, Value) → Job Output (Key'', Value'')**
   - **Reduce Function:** Use **Identity** function!
4. If data **needs to be shuffled**:
   - **Map Function:**
     - How should data be grouped → Defines Map Output **Key'**
     - What values are needed in Reduce task → Defines Map Output **Value'**
     - **Function: Job Input (Key, Value) → Map Output (Key', Value')**
   - **Reduce Function:**
     - **Input:** Based on Map Output: **(Key', [Value'])**
     - **Function: Reduce Input (Key', [Value']) → Job Output (Key'', Value'')**

Inverted Index Algorithm

- Generate a **Word to File** index for each word in the input dataset
- **Input:** Set of text files
- **Output:** For each word, return a list of files it appeared in
- **Map Function**
  - **Input:** (LineOffset, Line)
  - **Function:** Extract words from the line of text.
  - **Output:** (word, fileName)
- **Reduce Function**
  - **Input:** (word, [fileName])
  - **Function:** Concatenate list of file names into a single string
  - **Output:** (word, “[fileName]“)

Index: Data Flow



Inverted Index MapReduce pseudocode

```
map(LineOffset, Line, context):
        pageName = context.getInputSplitFileName()
        foreach word in Line:
                emit(word, pageName)

reduce(word, values):
        pageList = []
        foreach pageName in values:
                pageList.add(pageName)
        emit(word, str(set(pageList)))
```

Distributed Global Sort

- Task is to sort a very large list of numerical values
- Each value is in a separate line inside a text file
- **Input:** A set of text files
- **Output:** values are in a globally sorted order in the output files

- Can be used as a benchmark to measure the raw throughput of the MapReduce cluster

Sort: The Trick

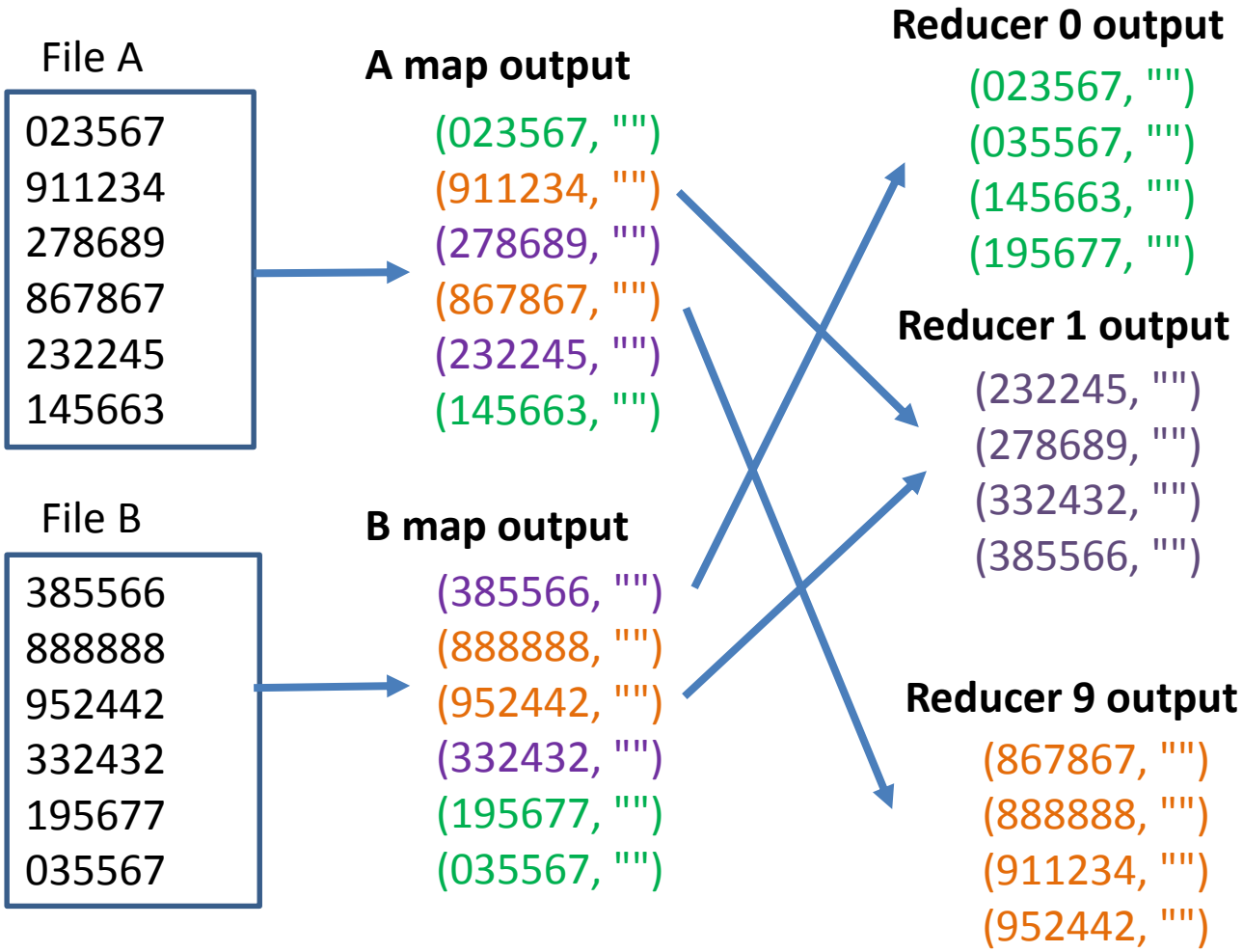
- Take advantage of Reducer properties:
  - (Key, Value) pairs are processed in order by key
  - (Key, Value) pairs from mappers are sent to a particular reducer based on Partition(key) function
- Change the Partition function
  - Must use a partition function such that:

```
IF K1 < K2 THEN Partition(K1) <= Partition(K2)
```

Distributed Sort algorithm

- **Map Function**
  - **Input:** (LineOffset, Line)
  - **Function:** Move the value into the Key
  - **Output:** (Line, _)
- **Reduce Function**
  - **Input:** (Line, [_])
  - **Function:** Identity Reducer
  - **Output:** (Line, _)

Distributed Sort Data Flow



Let's focus on a bit
more complex problems

Term co-occurrence matrix

- 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 (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)

- How large is the resulting matrix?
- How many elements do we need to count?

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

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 approach: “Pairs”

- WordCount-like approach
- 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!

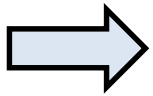
“Pairs” Analysis

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

Second approach: “Stripes”

- Idea: group together pairs into an associative array

```
(a, b) → 1
(a, c) → 2
(a, d) → 5
(a, e) → 3
(a, f) → 2
```



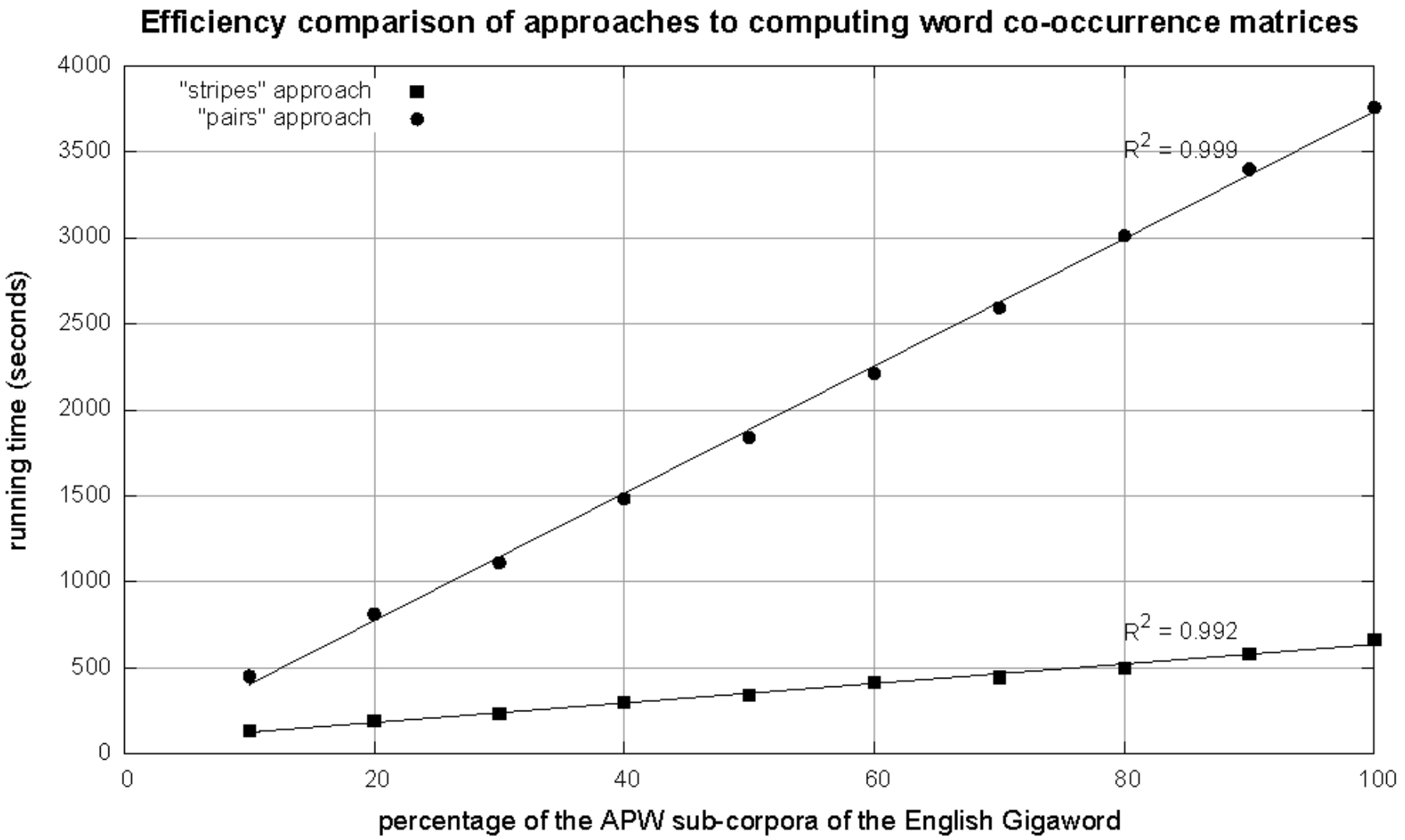
```
a → { b: 1, c: 2, d: 5, e: 3, f: 2 }
```

- 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)

Managing Dependencies in Data

- Remember, Mappers run in isolation. We can't control:
  - The order in which mappers run
  - On which nodes the mappers run
  - When each mapper finishes
- Available tools for synchronization:
  - Ability to hold state in reducer across multiple key-value pairs
  - Sorting function for keys
  - Partitioners
  - Broadcasting/replicating values
  - Cleverly-constructed data structures

Conditional Probabilities

- What is the chance of word B occurring in a sentence that contains word A.
- How do we compute conditional probabilities from counts?

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

- How do we compute this with MapReduce?

P(B|A): “Pairs”

- Co-occurance matrix already gives us: count(A, B)
- Need to also compute count(A)

(a, *) → 23 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 / 23
(a, b_2) → 12 / 23
(a, b_3) → 7 / 23
(a, b_4) → 1 / 23

- How can we compute count(a) without changing how the data is grouped?
  - Must also emit an 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
  - Partition key space so that each reducer gets the appropriate set of partial results
  - Sort keys into correct order of computation
  - 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 to use more complex data types as Keys and Values?
- The easiest way:
  - Encode it as a composed String, 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 more efficient, but slow for rapid prototyping

Custom Hadoop WritableComparable Object

```
public class MyKey implements WritableComparable {
     private int ID;
     private long phone_num;

     public void write(DataOutput out) {
         out.writeInt(ID);
         out.writeLong(phone_num);
     }

     public void readFields(DataInput in) {
         ID = in.readInt();
         phone_num = in.readLong();
     }
     public int compareTo(MyKey o) {
         int res = Integer.compare(this.ID, o.ID);
         if (res != 0)
             return res;
         return Long.compare(this.phone_num, o.phone_num);
     }
}
```

Next Lab

- Creating a new MapReduce application
  - Analyzing an open dataset
  - Parsing CSV files
  - Aggregating data using simple statistical functions

Next Lecture

- Platform as a Service (PaaS) model
  - Google AppEngine
  - Elastic MapReduce (EMR)
    - MapReduce platform as a Service

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.
- Jimmy Lin and Chris Dyer, "**Data-Intensive Text Processing with MapReduce**"
  http://lintool.github.io/MapReduceAlgorithms/MapReduce-book-final.pdf
  Pages 50-57: Pairs and Stripes problem