Data-Intensive Information Processing Applications — Session #3

# MapReduce Algorithm Design

**Jordan Boyd-Graber**
University of Maryland

Thursday, February 17, 2011

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

# Issues from Last Class

- Everybody has access to the cluster?
- Hardware Sorting
- Names
  - Ying : Jordan Boyd-Graber
  - Ychen126: Yingying Chen
- Input in Hadoop
- What is a node?
- Equal time: Avro

# Input Types

- Recall: FileSplits (split), InputFormat (parse), RecordReader (iterate)
- InputFormat Options
  - TextInputFormat (offset, line text)
  - StreamInputFormat
    - Use StreamXmlRecordReader if values are XML documents
  - KeyValueTextInputFormat (key, line text)
    - Settable delimiter (tab is default)
  - SequenceFileInputFormat (key, binary)
    - Use for binary / serialized input
  - MapFile
    - Just like SequenceFile, but sorted (key must be comparable)
  - Other: HBase, conventional databases

# What is a node?

- Not always 1 node per {computer, core}
- In many cases, nodes are virtual machines running in nodes (e.g. WorldLingo)
- How many nodes per machine depends on typical usage (e.g. IO vs CPU)

# Avro

- Much like protocol buffers
- Uses JSON to compile schema
- Newer, but better connected with Hadoop
  - Could have better integration, but not there yet
- Benifits compared to protocol buffers
  - Schema is transmitted **with** serialization
  - Does **not** require compiling code
- Limitations compared to protocol buffers
  - Schema is transmitted **with** serialization
  - Cannot have nested fields
  - Cannot have null fields
- Again, not required to use them

# Today's Agenda

- "The datacenter *is* the computer"
  - Understanding the design of warehouse-sized computes
- MapReduce algorithm design
  - How do you express everything in terms of m, r, c, p?
  - Toward "design patterns"

## The datacenter *is* the computer

# "Big Ideas"

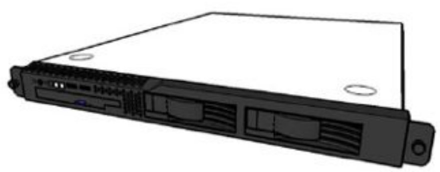
- Scale "out", not "up"
  - Limits of SMP and large shared-memory machines
- Move processing to the data
  - Cluster have limited bandwidth
- Process data sequentially, avoid random access
  - Seeks are expensive, disk throughput is reasonable
- Seamless scalability
  - From the mythical man-month to the tradable machine-hour

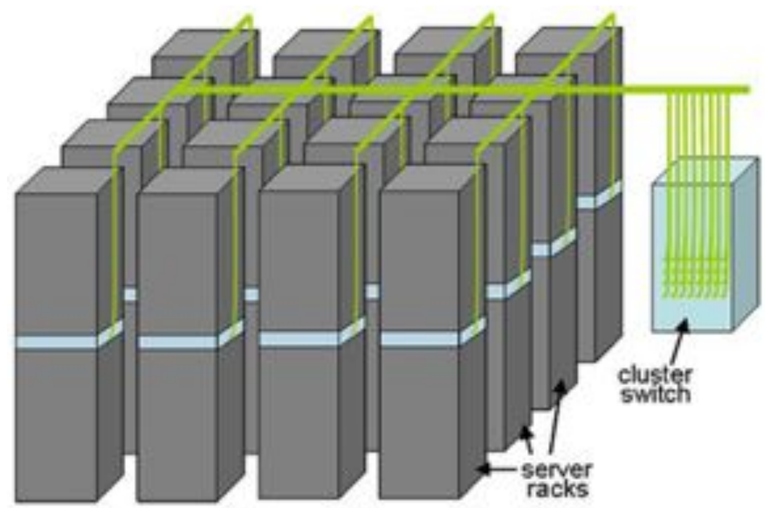
Source: NY Times (6/14/2006)


Source: www.robinmajumdar.com

Source: Bonneville Power Administration

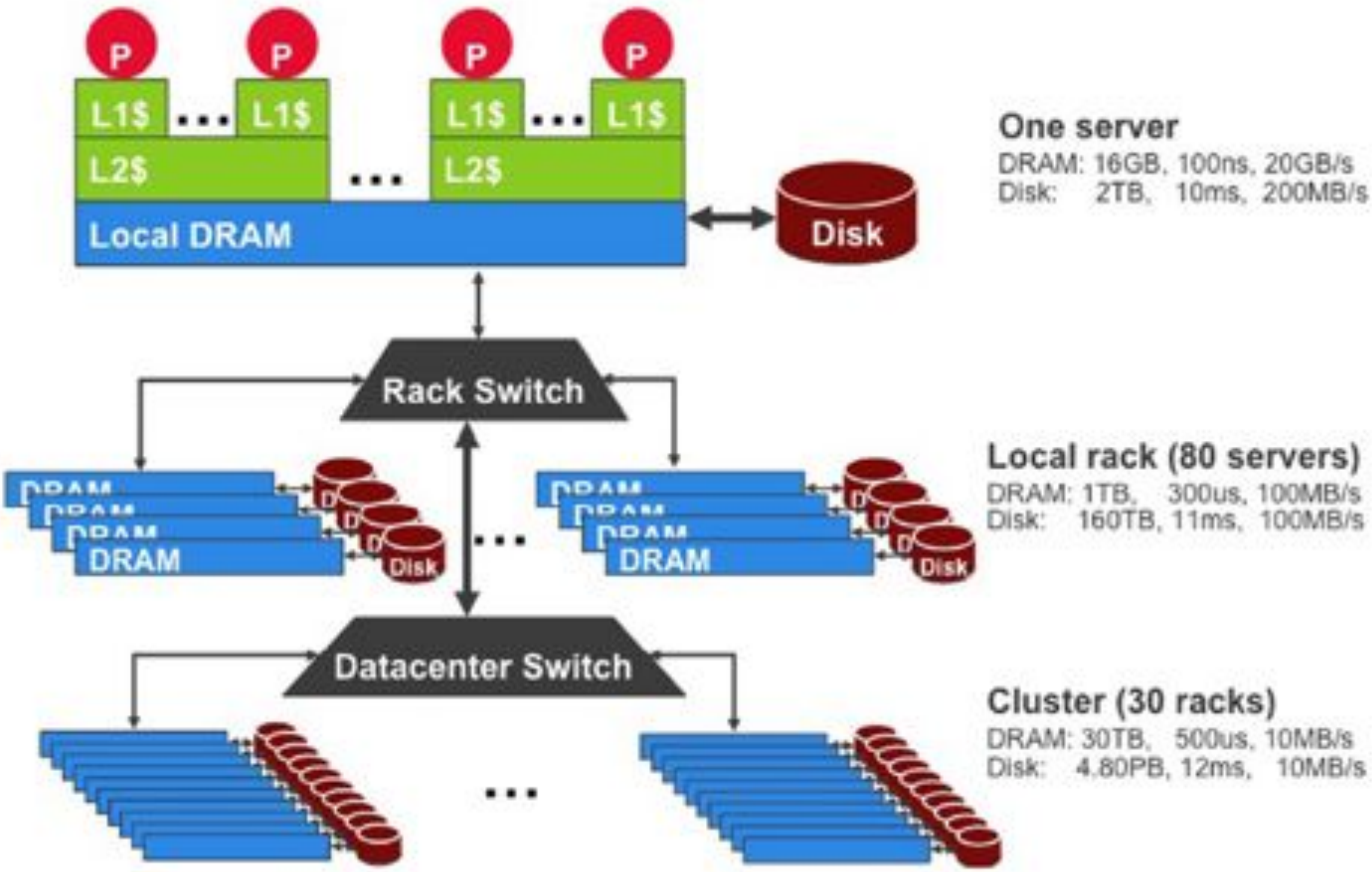
# Building Blocks





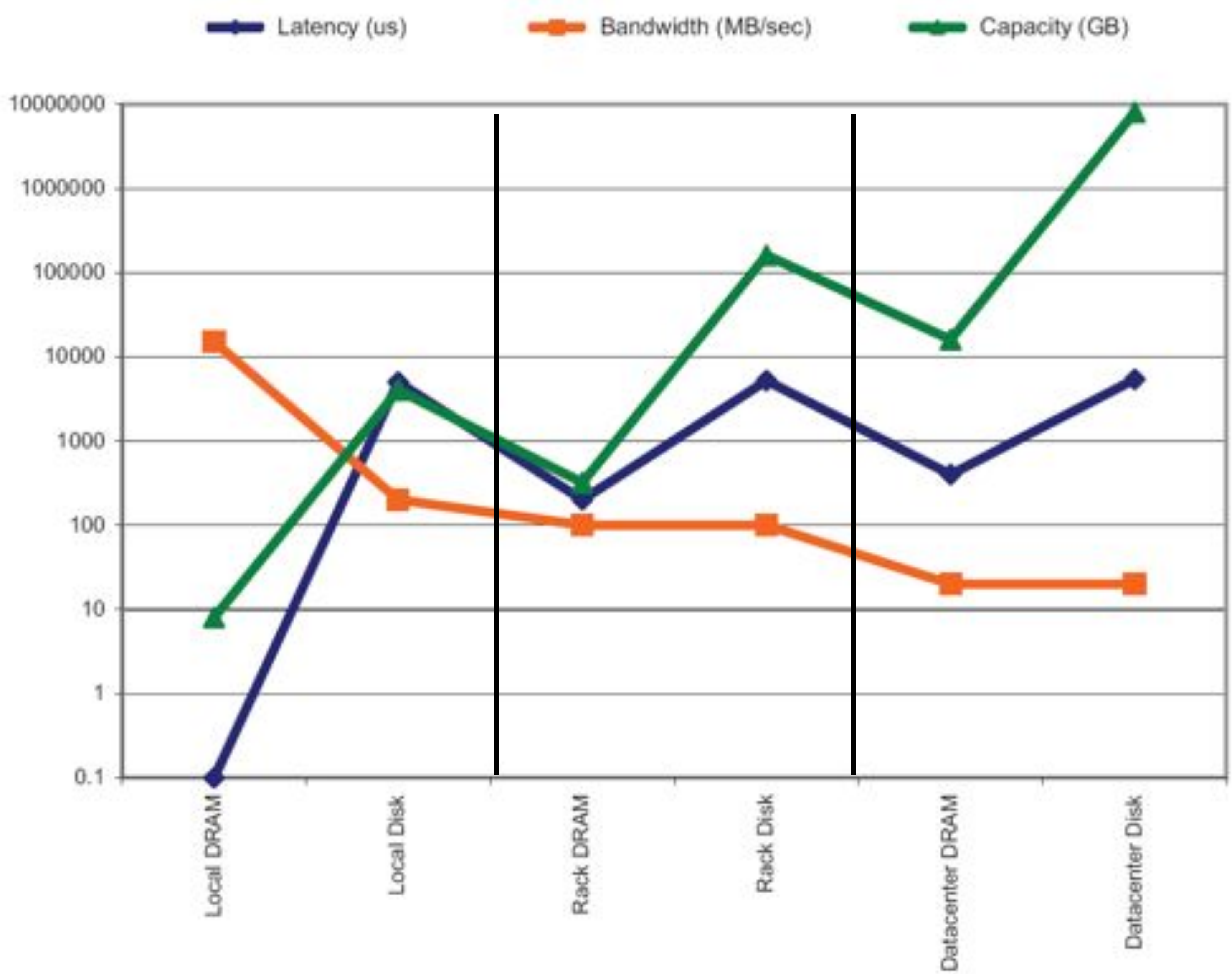

Source: Barroso and Urs Hölzle (2009)

# Storage Hierarchy



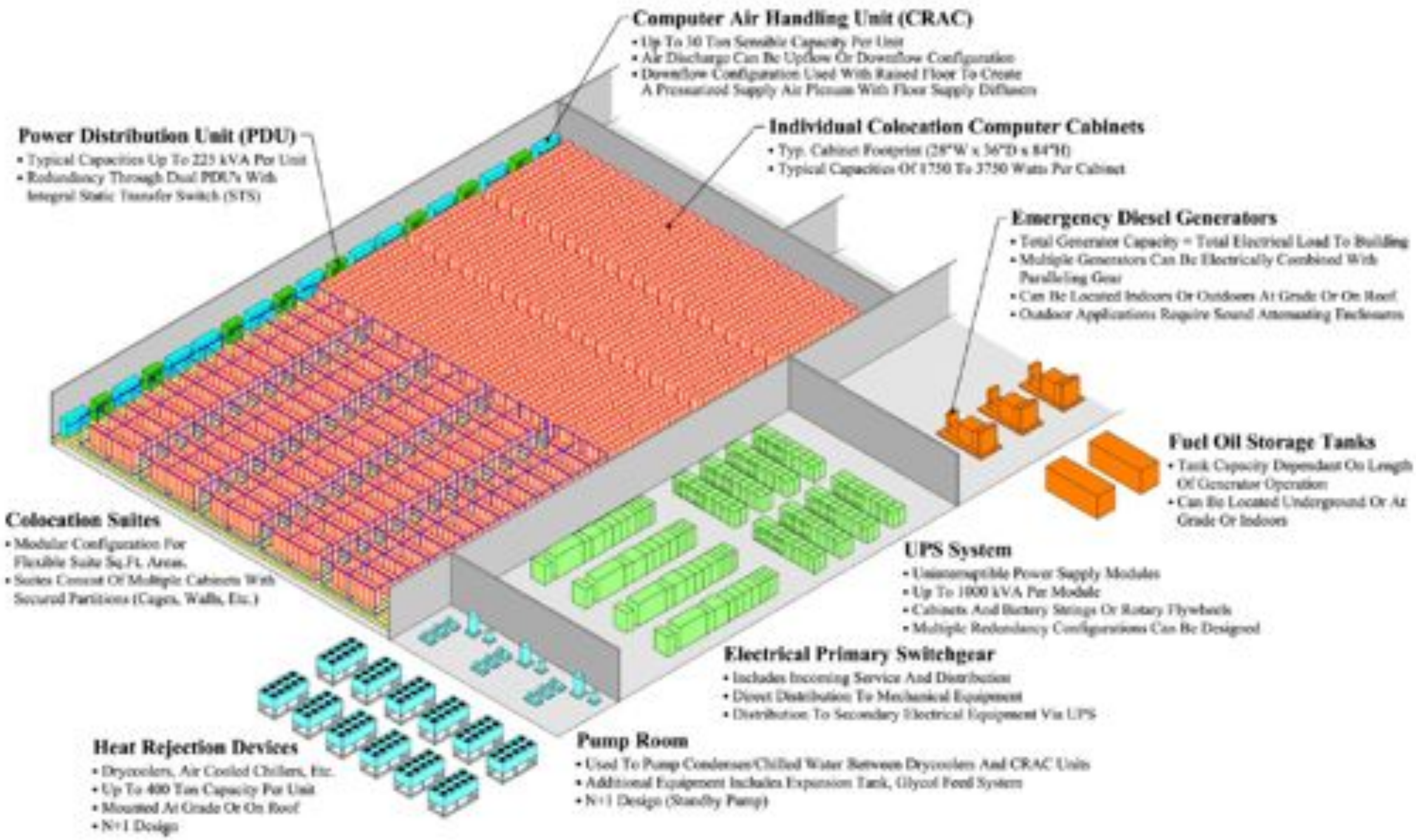

Source: Barroso and Urs Hölzle (2009)

# Storage Hierarchy




Source: Barroso and Urs Hölzle (2009)

# Anatomy of a Datacenter




Source: Barroso and Urs Hölzle (2009)

# Why commodity machines?

| | HP INTEGRITY SUPERDOME-ITANIUM2 | HP PROLIANT ML350 G5 |
|---|---|---|
| Processor | 64 sockets, 128 cores (dual-threaded), 1.6 GHz Itanium2, 12 MB last-level cache | 1 socket, quad-core, 2.66 GHz X5355 CPU, 8 MB last-level cache |
| Memory | 2,048 GB | 24 GB |
| Disk storage | 320,974 GB, 7,056 drives | 3,961 GB, 105 drives |
| TPC-C price/performance | \$2.93/tpmC | \$0.73/tpmC |
| price/performance (server HW only) | \$1.28/transactions per minute | \$0.10/transactions per minute |
| Price/performance (server HW only) (no discounts) | \$2.39/transactions per minute | \$0.12/transactions per minute |

Source: Barroso and Urs Hölzle (2009); performance figures from late 2007

# Why commodity machines?

- Diminishing returns for high-end machines
- Power usage is lower for mid-range machines
- If you‘re doing it right, many processes are memory

Source: Barroso and Urs Hölzle (2009); performance figures from late 2007

# What about communication?

- Nodes need to talk to each other!
  - SMP: latencies ~100 ns
  - LAN: latencies ~100 $\mu$s
- Scaling “up” vs. scaling “out”
  - Smaller cluster of SMP machines vs. larger cluster of commodity machines
  - E.g., 8 128-core machines vs. 128 8-core machines
  - Note: no single SMP machine is big enough
- Let’s model communication overhead…

Source: analysis on this an subsequent slides from Barroso and Urs Hölzle (2009)

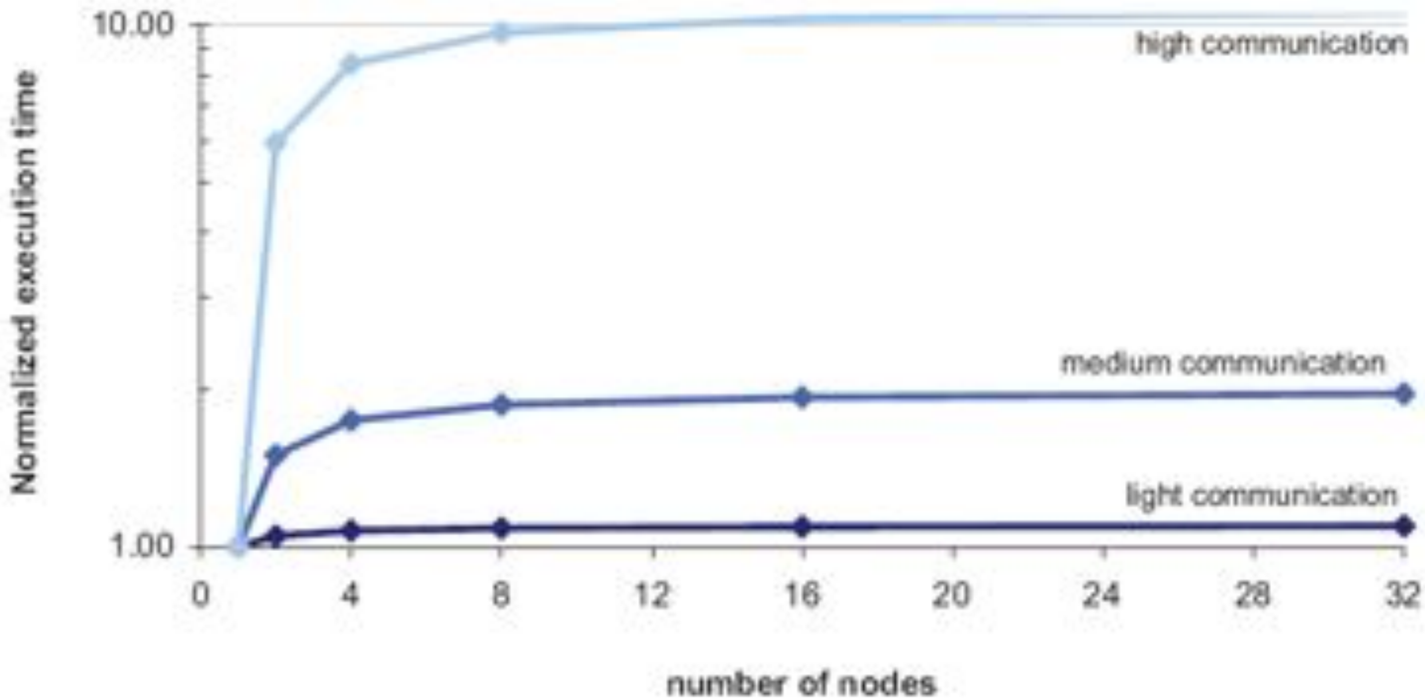
# Modeling Communication Costs

- Simple execution cost model:
  - Total cost = cost of computation + cost to access global data
  - Fraction of local access inversely proportional to size of cluster
  - $n$ nodes (ignore cores for now)

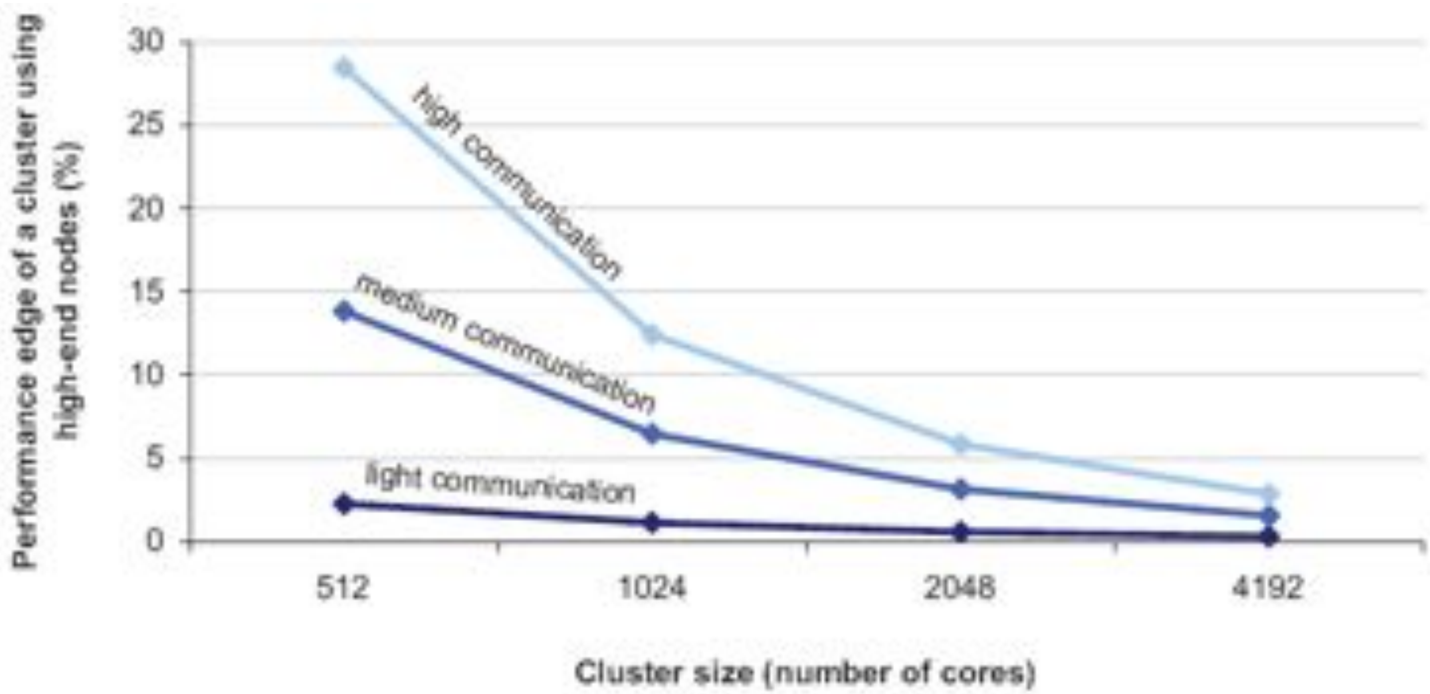
$$1 \text{ ms} + f \times [100 \text{ ns} \times n + 100\ \mu\text{s} \times (1 - 1/n)]$$

    - Light communication: $f = 1$
    - Medium communication: $f = 10$
    - Heavy communication: $f = 100$
- What are the costs in parallelization?

# Cost of Parallelization



# Advantages of scaling "up"




So why not?

# Seeks vs. Scans

- Consider a 1 TB database with 100 byte records
  - We want to update 1 percent of the records
- Scenario 1: random access
  - Each update takes ~30 ms (seek, read, write)
  - $10^8$ updates = ~35 days
- Scenario 2: rewrite all records
  - Assume 100 MB/s throughput
  - Time = 5.6 hours(!)
- Lesson: avoid random seeks!

Source: Ted Dunning, on Hadoop mailing list

# Justifying the “Big Ideas”

- Scale “out”, not “up”
  - Limits of SMP and large shared-memory machines
- Move processing to the data
  - Cluster have limited bandwidth
- Process data sequentially, avoid random access
  - Seeks are expensive, disk throughput is reasonable
- Seamless scalability
  - From the mythical man-month to the tradable machine-hour

# Numbers Everyone Should Know*

| | |
|---|---|
| L1 cache reference | 0.5 ns |
| Branch mispredict | 5 ns |
| L2 cache reference | 7 ns |
| Mutex lock/unlock | 25 ns |
| Main memory reference | 100 ns |
| Send 2K bytes over 1 Gbps network | 20,000 ns |
| Read 1 MB sequentially from memory | 250,000 ns |
| Round trip within same datacenter | 500,000 ns |
| Disk seek | 10,000,000 ns |
| Read 1 MB sequentially from disk | 20,000,000 ns |
| Send packet CA → Netherlands → CA | 150,000,000 ns |

* According to Jeff Dean (LADIS 2009 keynote)

# MapReduce Algorithm Design

# MapReduce: Recap

- Programmers must specify:

  **map** (k, v) → <k’, v’>*

  **reduce** (k’, v’) → <k’, v’>*

  - All values with the same key are reduced together

- Optionally, also:

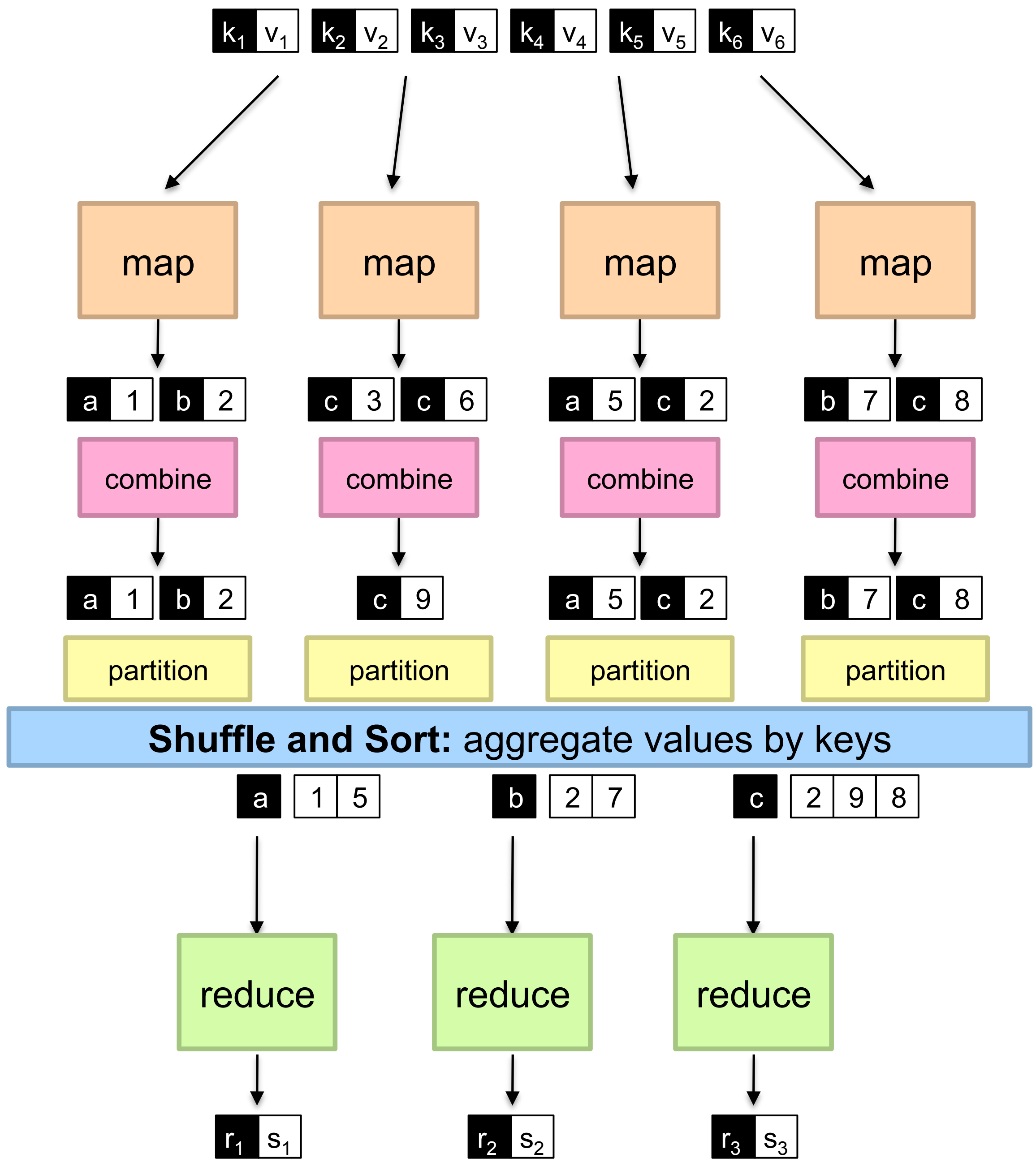
  **partition** (k’, number of partitions) → partition for k’

  - Often a simple hash of the key, e.g., hash(k’) mod n
  - Divides up key space for parallel reduce operations

  **combine** (k’, v’) → <k’, v’>*

  - Mini-reducers that run in memory after the map phase
  - Used as an optimization to reduce network traffic

- The execution framework handles everything else…



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 9 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

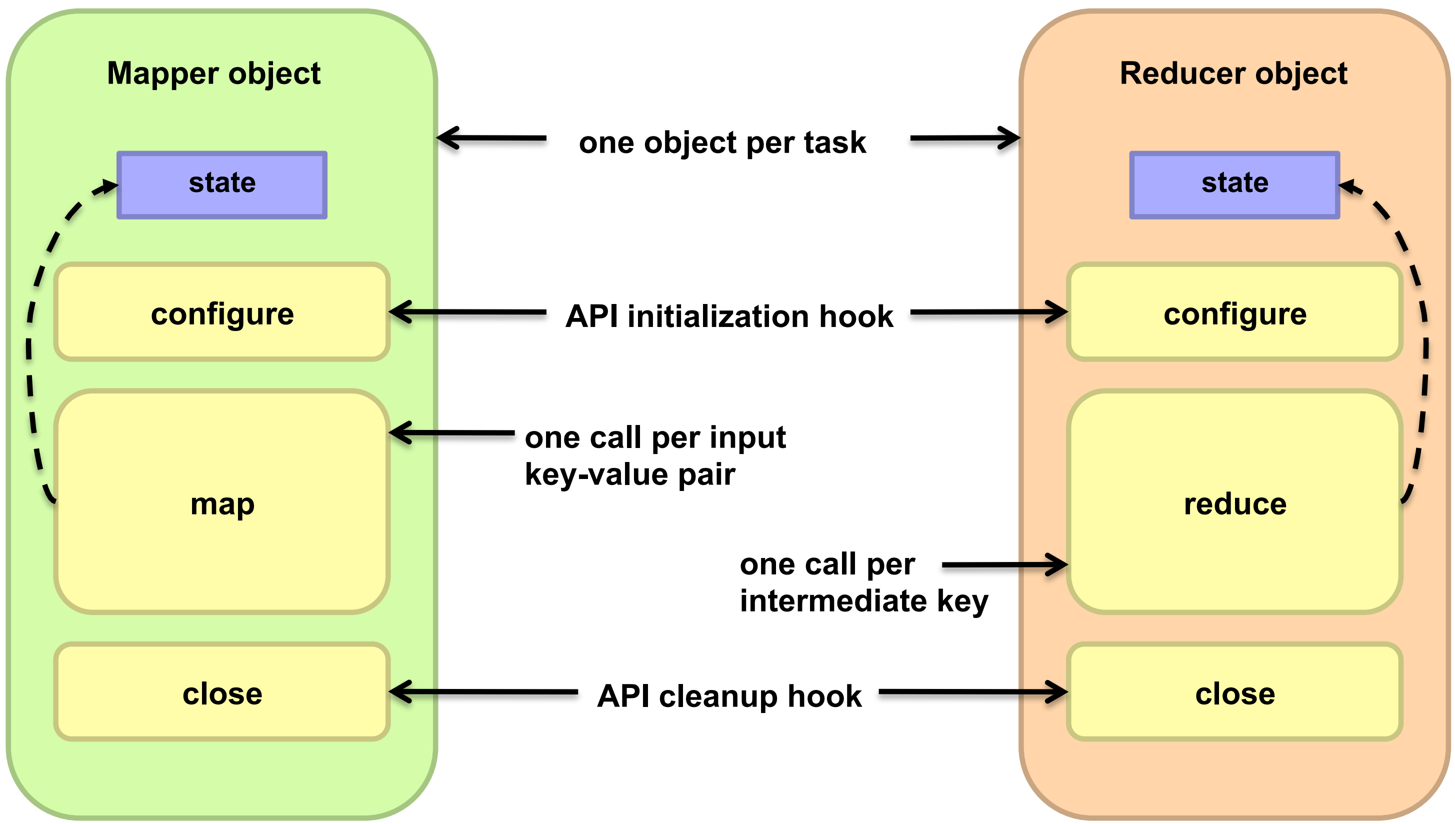
# “Everything Else”

- The execution framework handles everything else…
  - Scheduling: assigns workers to map and reduce tasks
  - “Data distribution”: moves processes to data
  - Synchronization: gathers, sorts, and shuffles intermediate data
  - Errors and faults: detects worker failures and restarts
- Limited control over data and execution flow
  - All algorithms must expressed in m, r, c, p
- You don’t know:
  - Where mappers and reducers run
  - When a mapper or reducer begins or finishes
  - Which input a particular mapper is processing
  - Which intermediate key a particular reducer is processing

# 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

# Preserving State



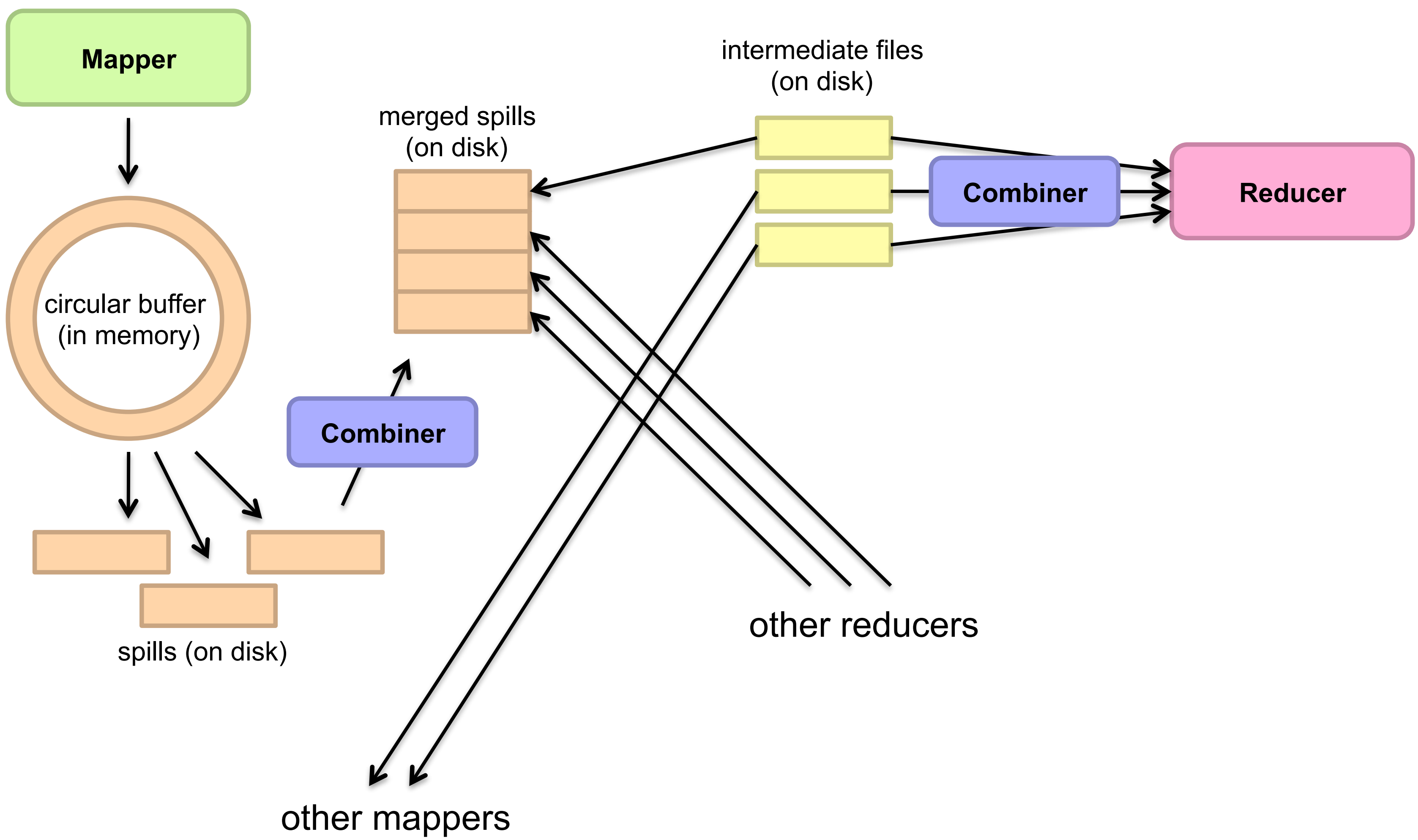
# Scalable Hadoop Algorithms: Themes

- Avoid object creation
  - Inherently costly operation
  - Garbage collection
- Avoid buffering
  - Limited heap size
  - Works for small datasets, but won’t scale!

# Importance of Local Aggregation

- 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

# Shuffle and Sort



# Word Count: Baseline

```
class Mapper
    method Map(docid a, doc d)
        for all term t ∈ doc d do
            Emit(term t, count 1)

class Reducer
    method Reduce(term t, counts [c1, c2, ...])
        sum ← 0
        for all count c ∈ counts [c1, c2, ...] do
            sum ← sum + c
        Emit(term t, count s)
```

**What's the impact of combiners?**

# Word Count: Version 1

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

**Are combiners still needed?**

# Word Count: Version 2

```
class Mapper
    method Initialize
        H ← new AssociativeArray
    method Map(docid a, doc d)
        for all term t ∈ doc d do
            H{t} ← H{t} + 1                ▷ Tally counts across documents
    method Close
        for all term t ∈ H do
            Emit(term t, count H{t})
```

Key: preserve state across input key-value pairs!

Are combiners still needed?

# 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
  - Speed
  - Why is this faster than actual combiners?
- Disadvantages
  - Explicit memory management required
  - Potential for order-dependent bugs

# Combiner Design

- Combiners and reducers share same method signature
  - Sometimes, reducers can serve as combiners
  - Often, not…
- Remember: combiner are optional optimizations
  - Should not affect algorithm correctness
  - May be run 0, 1, or multiple times
- Example: find average of all integers associated with the same key

# Computing the Mean: Version 1

```
class Mapper
    method Map(string t, integer r)
        Emit(string t, integer r)
```

```
class Reducer
    method Reduce(string t, integers [r_1, r_2, ...])
        sum ← 0
        cnt ← 0
        for all integer r ∈ integers [r_1, r_2, ...] do
            sum ← sum + r
            cnt ← cnt + 1
        r_avg ← sum/cnt
        Emit(string t, integer r_avg)
```

**Why can’t we use reducer as combiner?**

# Computing the Mean: Version 2

```
class Mapper
    method Map(string t, integer r)
        Emit(string t, integer r)

class Combiner
    method Combine(string t, integers [r1, r2, ...])
        sum ← 0
        cnt ← 0
        for all integer r ∈ integers [r1, r2, ...] do
            sum ← sum + r
            cnt ← cnt + 1
        Emit(string t, pair (sum, cnt))                ▷ Separate sum and count

class Reducer
    method Reduce(string t, pairs [(s1, c1), (s2, c2) ...])
        sum ← 0
        cnt ← 0
        for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
            sum ← sum + s
            cnt ← cnt + c
        r_avg ← sum/cnt
        Emit(string t, integer r_avg)
```

**Why doesn't this work?**

# Computing the Mean: Version 3

**class** MAPPER
    **method** MAP(string $t$, integer $r$)
        EMIT(string $t$, pair $(r, 1)$)

**class** COMBINER
    **method** COMBINE(string $t$, pairs $[(s_1, c_1), (s_2, c_2) \ldots]$)
        $sum \leftarrow 0$
        $cnt \leftarrow 0$
        **for all** pair $(s, c) \in$ pairs $[(s_1, c_1), (s_2, c_2) \ldots]$ **do**
            $sum \leftarrow sum + s$
            $cnt \leftarrow cnt + c$
        EMIT(string $t$, pair $(sum, cnt)$)

**class** REDUCER
    **method** REDUCE(string $t$, pairs $[(s_1, c_1), (s_2, c_2) \ldots]$)
        $sum \leftarrow 0$
        $cnt \leftarrow 0$
        **for all** pair $(s, c) \in$ pairs $[(s_1, c_1), (s_2, c_2) \ldots]$ **do**
            $sum \leftarrow sum + s$
            $cnt \leftarrow cnt + c$
        $r_{avg} \leftarrow sum/cnt$
        EMIT(string $t$, pair $(r_{avg}, cnt)$)

Fixed?

# Computing the Mean: Version 4

```
class Mapper
    method Initialize
        S ← new AssociativeArray
        C ← new AssociativeArray
    method Map(string t, integer r)
        S{t} ← S{t} + r
        C{t} ← C{t} + 1
    method Close
        for all term t ∈ S do
            Emit(term t, pair (S{t}, C{t}))
```

Are combiners still needed?

# Algorithm Design: Running 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

# 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 observations (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 sum up counts associated with these pairs
- Use combiners!

# Pairs: Pseudo-Code

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

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

# “Pairs” Analysis

- Advantages
  - Easy to implement, easy to understand
- Disadvantages
  - Lots of pairs to sort and shuffle around (upper bound?)
  - 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 → { b: 1, c: 2, d: 5, e: 3, f: 2 }
(a, e) → 3
(a, 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 } |

**Key: cleverly-constructed data structure brings together partial results**

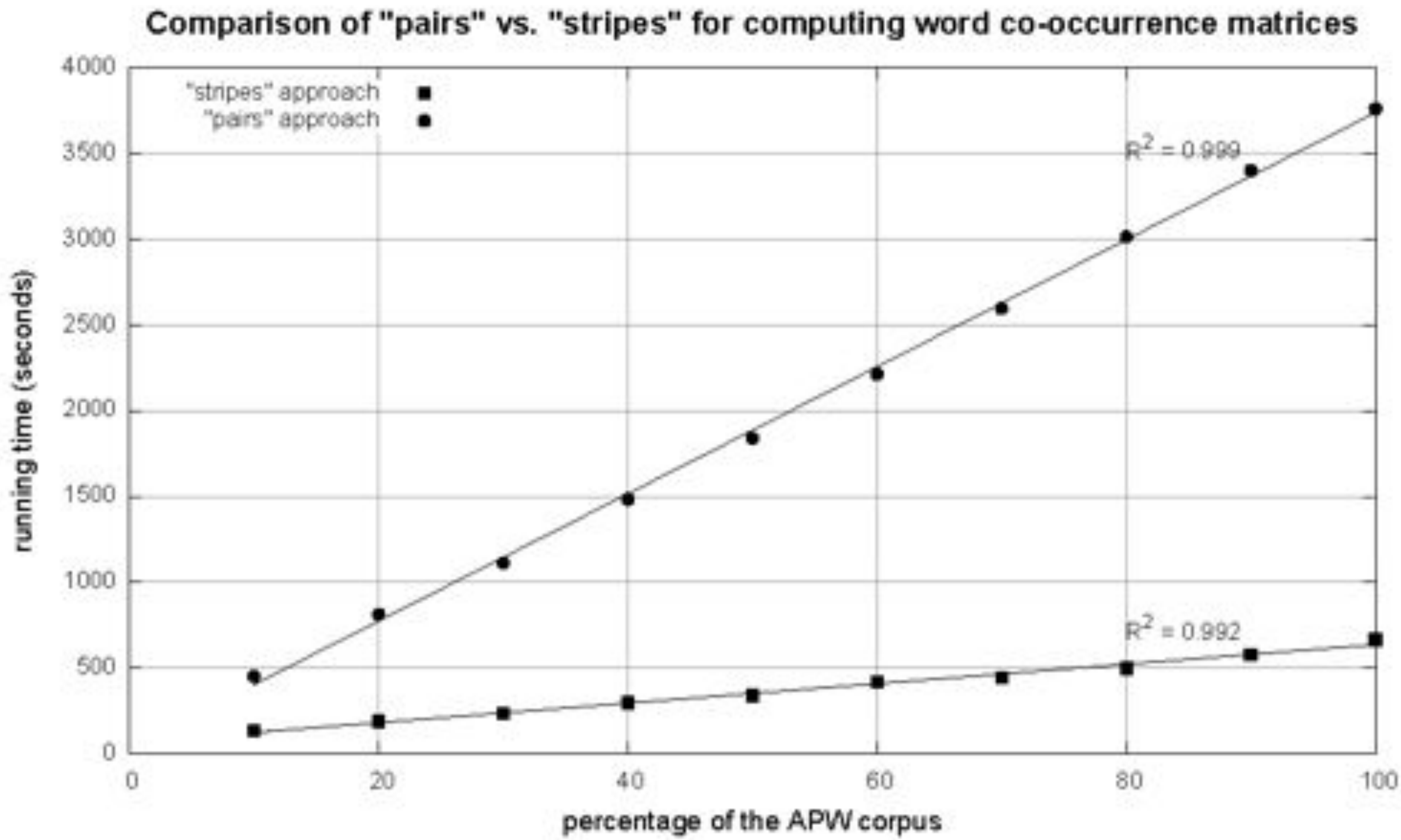
# Stripes: Pseudo-Code

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

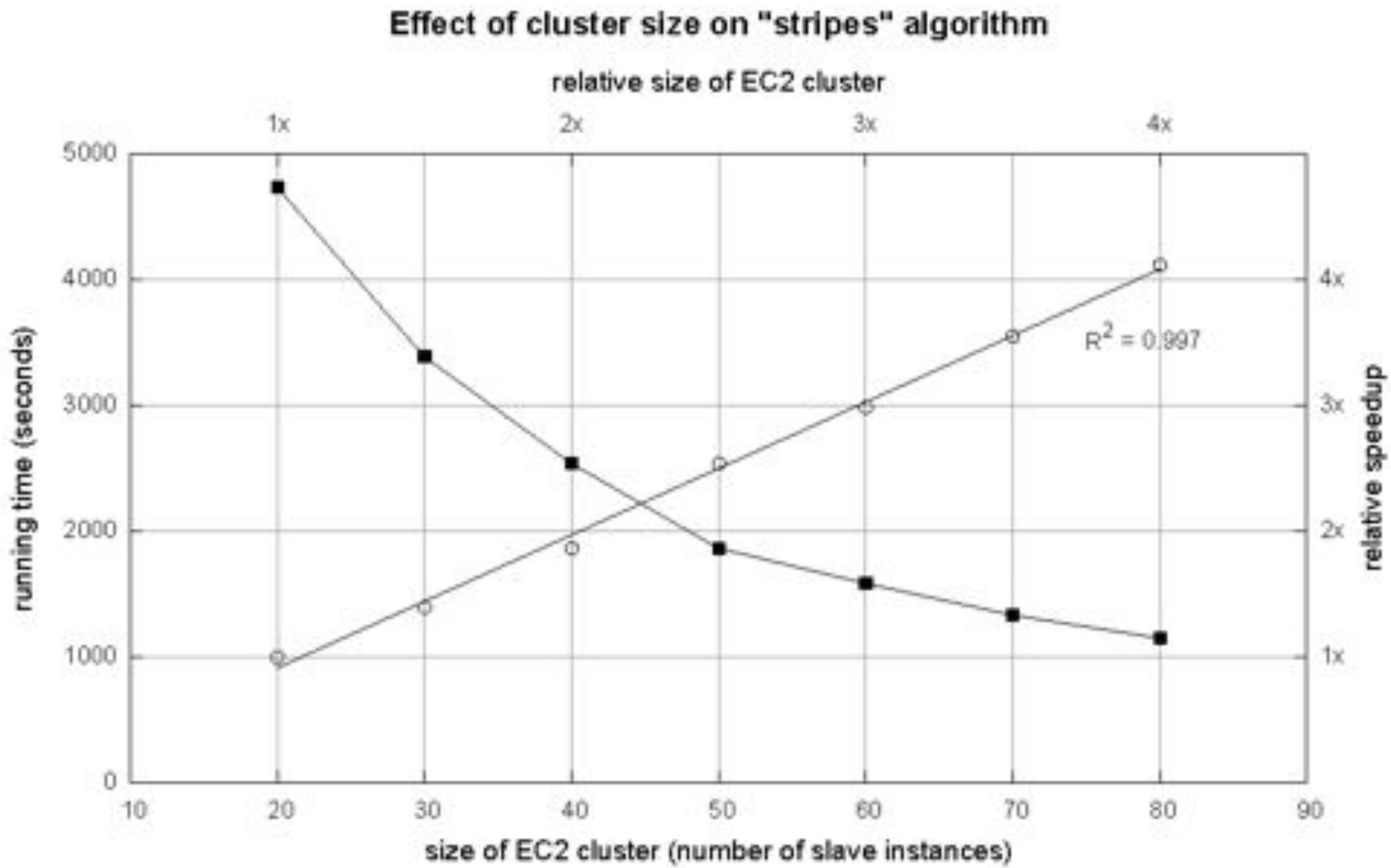
```
class Reducer
    method Reduce(term w, stripes [H1, H2, H3, ...])
        Hf ← new AssociativeArray
        for all stripe H ∈ stripes [H1, H2, H3, ...] do
            Sum(Hf, H)                              ▷ Element-wise sum
        Emit(term w, stripe Hf)
```

# “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
  - 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)



Effect of cluster size on "stripes" algorithm
relative size of EC2 cluster
1x
2x
3x
4x
5000
4000
3000
2000
1000
0
running time (seconds)
relative speedup
4x
3x
2x
1x
R2 = 0.997
10
20
30
40
50
60
70
80
90
size of EC2 cluster (number of slave instances)

# Relative Frequencies

- How do we estimate relative frequencies from counts?

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

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

# f(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 f(B|A)

# f(B|A): "Pairs"

$(a, *) \rightarrow 32$ **Reducer holds this value in memory**

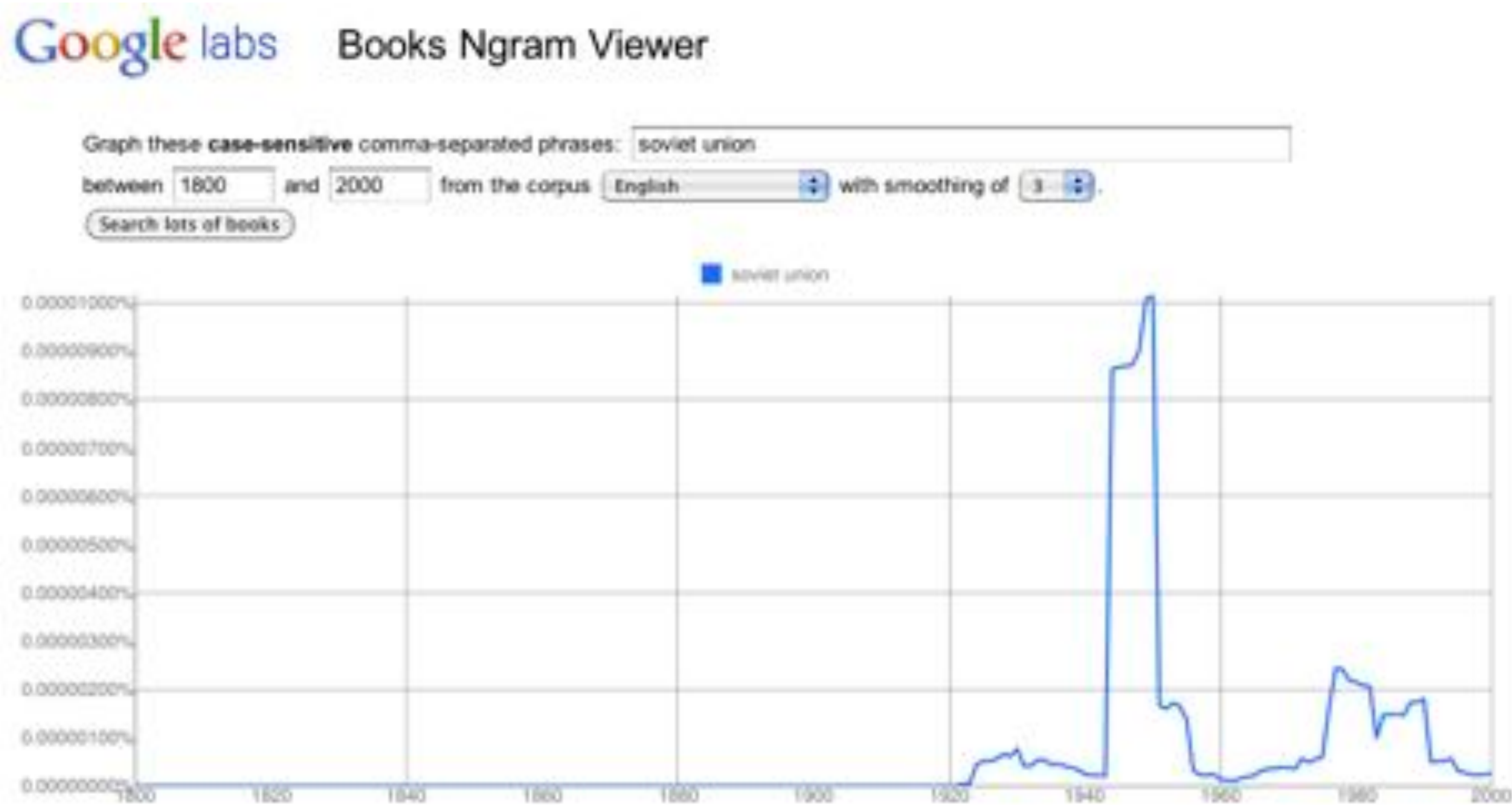
| | | |
|---|---|---|
| $(a, b_1) \rightarrow 3$ | → | $(a, b_1) \rightarrow 3 / 32$ |
| $(a, b_2) \rightarrow 12$ | | $(a, b_2) \rightarrow 12 / 32$ |
| $(a, b_3) \rightarrow 7$ | | $(a, b_3) \rightarrow 7 / 32$ |
| $(a, b_4) \rightarrow 1$ | | $(a, b_4) \rightarrow 1 / 32$ |
| … | | … |

- 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)
  - Must hold state in reducer across different key-value pairs

# “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
- Optimizations
  - Apply in-memory combining pattern to accumulate marginal counts
  - Should we apply combiners?

# Order Inversion for Bigrams



# N-Gram Probability

- Given the phrase „I pity the“, what is the the probability of the next word being „fool“?
- Requires counting up the number of times „I pity the fool“ appears in the corpus and dividing by the number of times „I pity the“ appears.
- Useful for spelling correction, machine translation, speech recognition
- When N=2, bigrams

# Digging In: Bigram Example

- Run the program:
  - hadoop jar cloud9.jar edu.umd.cloud9.example.bigram.BigramRelativeFrequency /tmp/wiki /umd-lin/jbg/output/bigram 15
- Take a look at the ouput:
  - Hadoop jar cloud9.jar edu.umd.cloud9.example.bigram.AnalyzeBigramRelativeFrequency /umd-lin/jbg/output/bigram
- Definition
  - Mapper<LongWritable, Text, PairOfStrings, FloatWritable>
  - Reducer<PairOfStrings, FloatWritable, PairOfStrings, FloatWritable>

# Digging In: Bigram Mapper

```
public void map(LongWritable key, Text value, Context context) {
    String line = value.toString();
    String prev = null;
    StringTokenizer itr = new StringTokenizer(line);
    while (itr.hasMoreTokens()) {
        String cur = itr.nextToken();
         if (prev == null) continue;
        bigram.set(prev, cur);
        context.write(bigram, one);
        bigram.set(prev, "*");
        context.write(bigram, one);
    }
    prev = cur;
}
}
```

# Digging In: Bigram Reducer

```
public void reduce(PairOfStrings key, Iterable<FloatWritable> values, Context context) {
    float sum = 0.0f;
    Iterator<FloatWritable> iter = values.iterator();
    while (iter.hasNext()) sum += iter.next().get();
    if (key.getRightElement().equals("*")) {
        value.set(sum);
        marginal = sum;
    } else {
        value.set(sum / marginal);
        context.write(key, value);
    }
}
```

# 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

# Digging In: Pairs

- Datatype:
  - import edu.umd.cloud9.io.PairOfStrings
- Definitions:

```
Reducer<PairOfStrings, IntWritable, PairOfStrings, IntWritable>
Mapper<LongWritable, Text, PairOfStrings, IntWritable>
```

- Mapper

```
public void map(LongWritable key, Text line, Context context) {
  String[] terms = line.toString().split("\\s+");
  for (int i = 0; i < terms.length; i++) {
   String term = terms[i];
   for (int j = i - window; j < i + window + 1; j++) {
       // OMITTED: Check to make sure valid pair
       pair.set(term, terms[j]);
       context.write(pair, one);
}}}
```

# Digging In: Pairs

- Reducer

```
public void reduce(PairOfStrings key, Iterable<IntWritable> values, Context
context) {
  Iterator<IntWritable> iter = values.iterator();
  int sum = 0;
      while (iter.hasNext()) {sum += iter.next().get();}
  SumValue.set(sum);
  context.write(key, SumValue);
}
```

# Digging In: Stripes

- Datatype:
  - import edu.umd.cloud9.io.fastuil.String2IntOpenHashMapWritable;
- Definitions

```
Mapper<LongWritable, Text, Text, String2IntOpenHashMapWritable>
Reducer<Text, String2IntOpenHashMapWritable, Text,
        String2IntOpenHashMapWritable>
```

- Mapper

```
map(LongWritable key, Text line, Context context) {
  String[] terms = line.toString().split("\\s+");
  for (int i = 0; i < terms.length; i++) {
    String term = terms[i];
    map.clear();
    for (int j = i - window; j < i + window + 1; j++) map.put(terms[j], 1);
    textKey.set(term);
    context.write(textKey, map);
  }
}
```

# Digging In: Stripes

- Reducer

```
public void reduce(Text key, Iterable<String2IntOpenHashMapWritable> values,
Context context) {
    Iterator<String2IntOpenHashMapWritable> iter = values.iterator();
    String2IntOpenHashMapWritable map = new String2IntOpenHashMapWritable();
    while (iter.hasNext()) map.plus(iter.next());
   context.write(key, map);
}
```

# Secondary Sorting

- MapReduce sorts input to reducers by key
  - Values may be arbitrarily ordered
- What if want to sort value also?
  - E.g., $k \rightarrow (v_1, r), (v_3, r), (v_4, r), (v_8, r)\ldots$

# 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, $(k, v_1)$
  - 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, bring local


Source: Wikipedia (Japanese rock garden)