MapReduce Algorithm Design

WWW 2013 Tutorial, Rio de Janeiro

Jimmy Lin
University of Maryland
Monday, May 13, 2013

Source: Wikipedia (All Souls College, Oxford)

… to building sh*t that works

Source: Wikipedia (Factory)

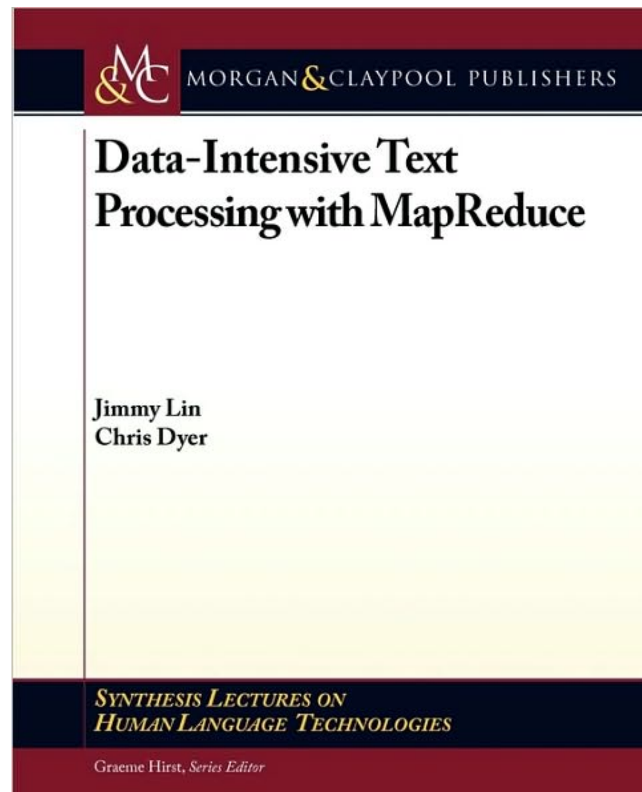
Source: Wikipedia (All Souls College, Oxford)

More about me…

- Past MapReduce teaching experience:
  - Numerous tutorials
  - Several semester-long MapReduce courses

    `http://lintool.github.io/MapReduce-course-2013s/`
- Lin & Dyer MapReduce textbook

    `http://mapreduce.cc/`



Follow me at @lintool

What we'll cover

- Big data
- MapReduce overview
- Importance of local aggregation
- Sequencing computations
- Iterative graph algorithms
- MapReduce and abstract algebra

Focus on design patterns and general principles

What we won't cover

- MapReduce for machine learning (supervised and unsupervised)
- MapReduce for similar item detection
- MapReduce for information retrieval
- Hadoop for data warehousing
- Extensions and alternatives to MapReduce

Big Data

Source: Wikipedia (Hard disk drive)

Google processes 20 PB a day (2008)
crawls 20B web pages a day (2012)

JPMorganChase 150 PB on 50k+ servers
running 15k apps (6/2011)

eBay >10 PB data, 75B DB
calls per day (6/2012)

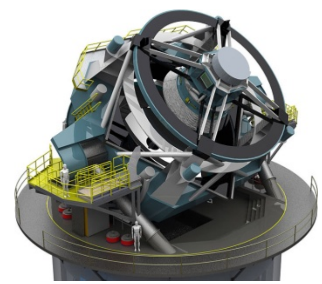
Internet Archive — Wayback Machine: 240B web
pages archived, 5 PB (1/2013)

>100 PB of user data +
500 TB/day (8/2012) — facebook

LHC: ~15 PB a year — CERN

amazon web services — S3: 449B objects, peak 290k
request/second (7/2011)
1T objects (6/2012)

640K ought to be
enough for anybody.



LSST: 6-10 PB a year
(~2015)

SKA: 0.3 – 1.5 EB
per year (~2020)

How much data?

Why big data?

Science

Engineering

Commerce

Source: Wikipedia (Everest)

Science

Emergence of the 4th Paradigm

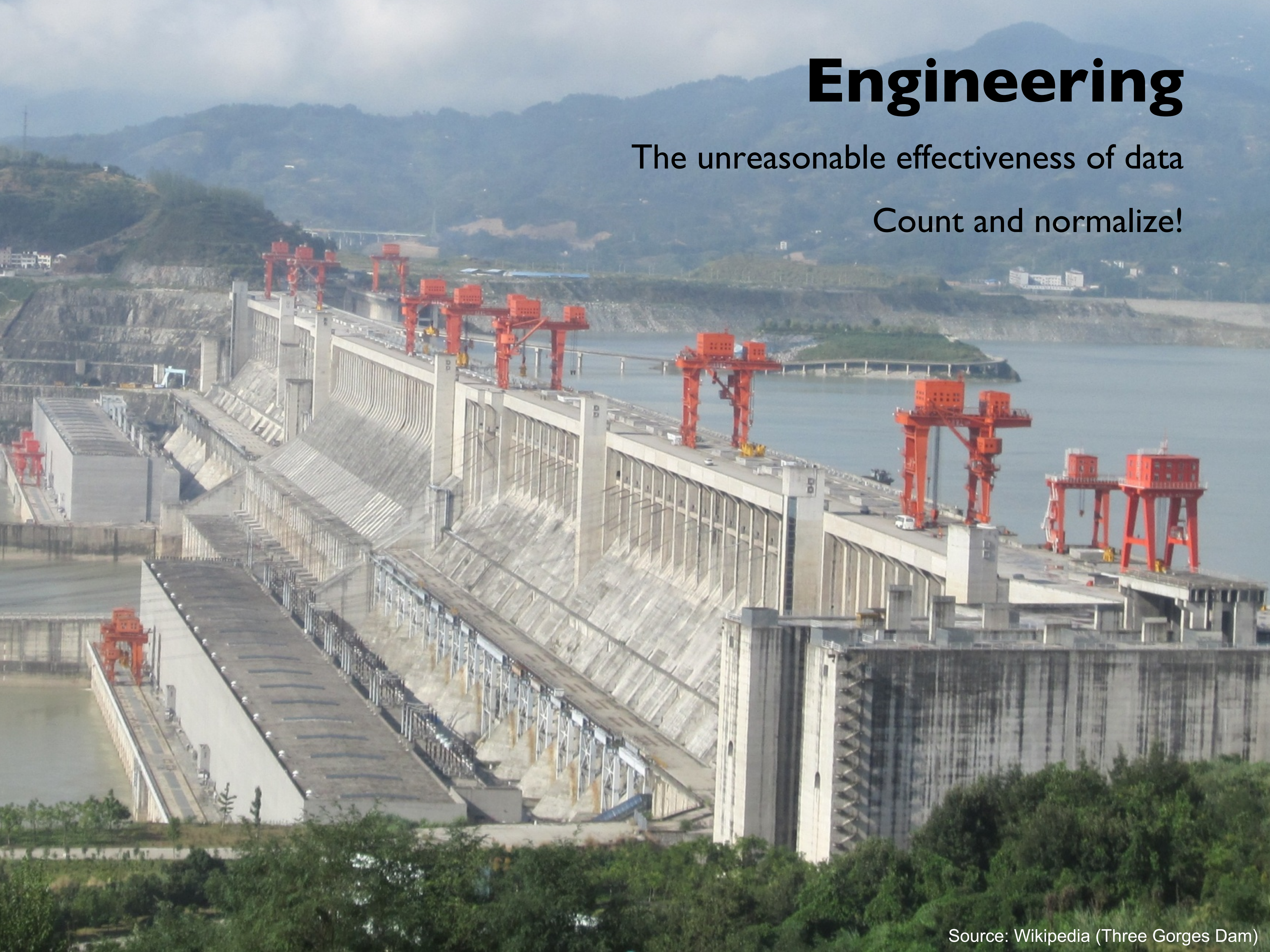
Data-intensive e-Science

Maximilien Brice, © CERN

Engineering

The unreasonable effectiveness of data

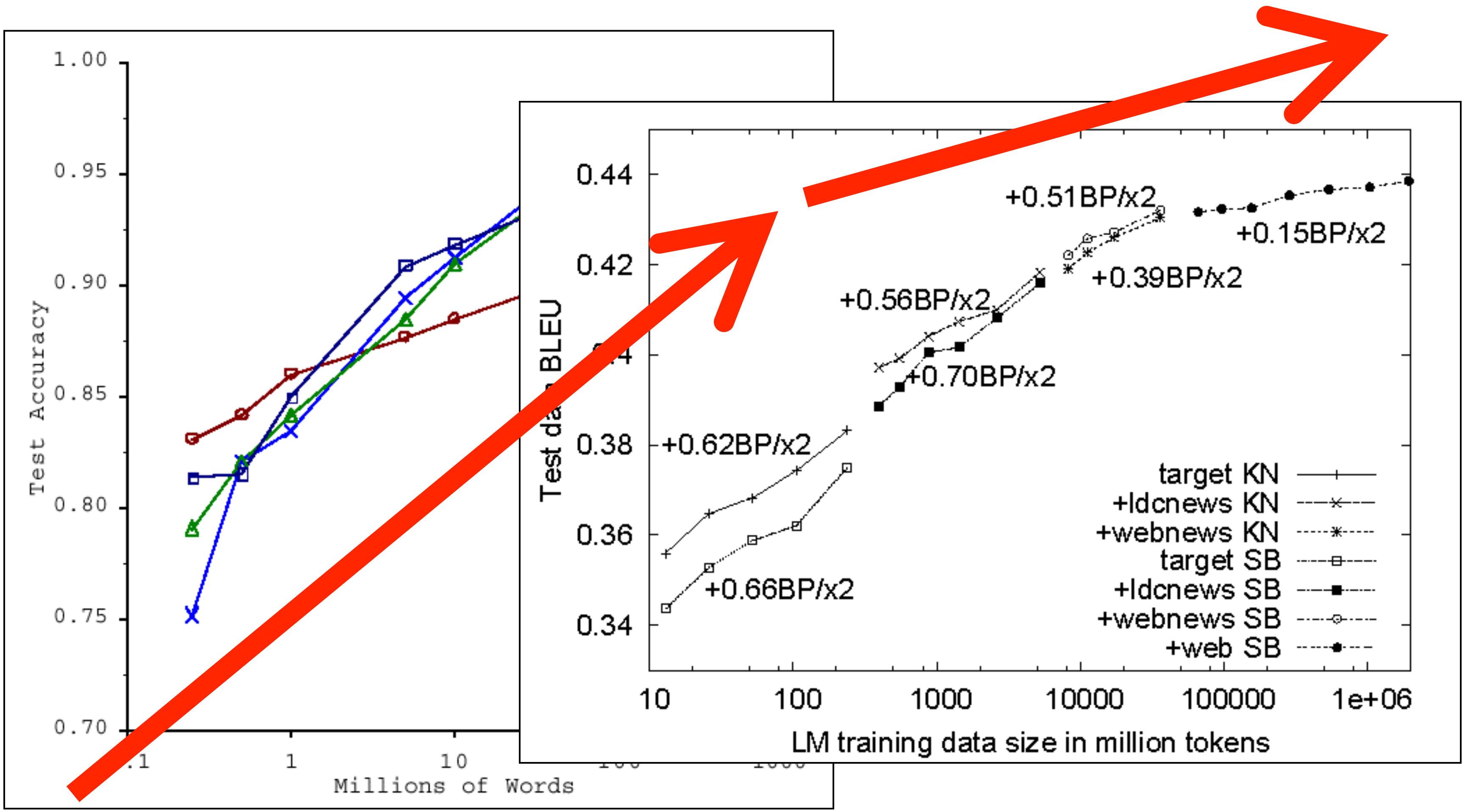
Count and normalize!



Source: Wikipedia (Three Gorges Dam)

No data like more data!

s/knowledge/data/g;



(Banko and Brill, ACL 2001)
(Brants et al., EMNLP 2007)

Know thy customers

Data → Insights → Competitive advantages

Commerce

Source: Wikiedia (Shinjuku, Tokyo)

Why big data?
How big data?

Source: Wikipedia (Noctilucent cloud)


MapReduce
Source: Google

Typical Big Data Problem

- Iterate over a large number of records
- Extract something of interest from each
- Shuffle and sort intermediate results
- Aggregate intermediate results
- Generate final output

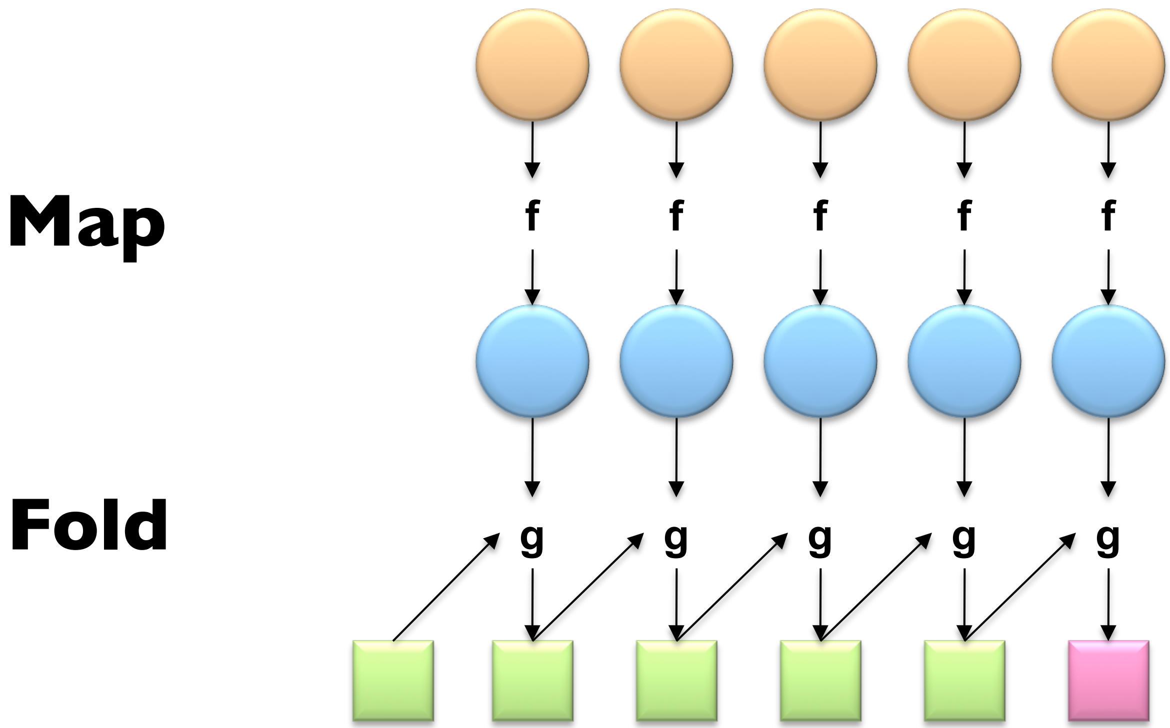
Map

Reduce

Key idea: provide a functional abstraction for these two operations

(Dean and Ghemawat, OSDI 2004)

Roots in Functional Programming



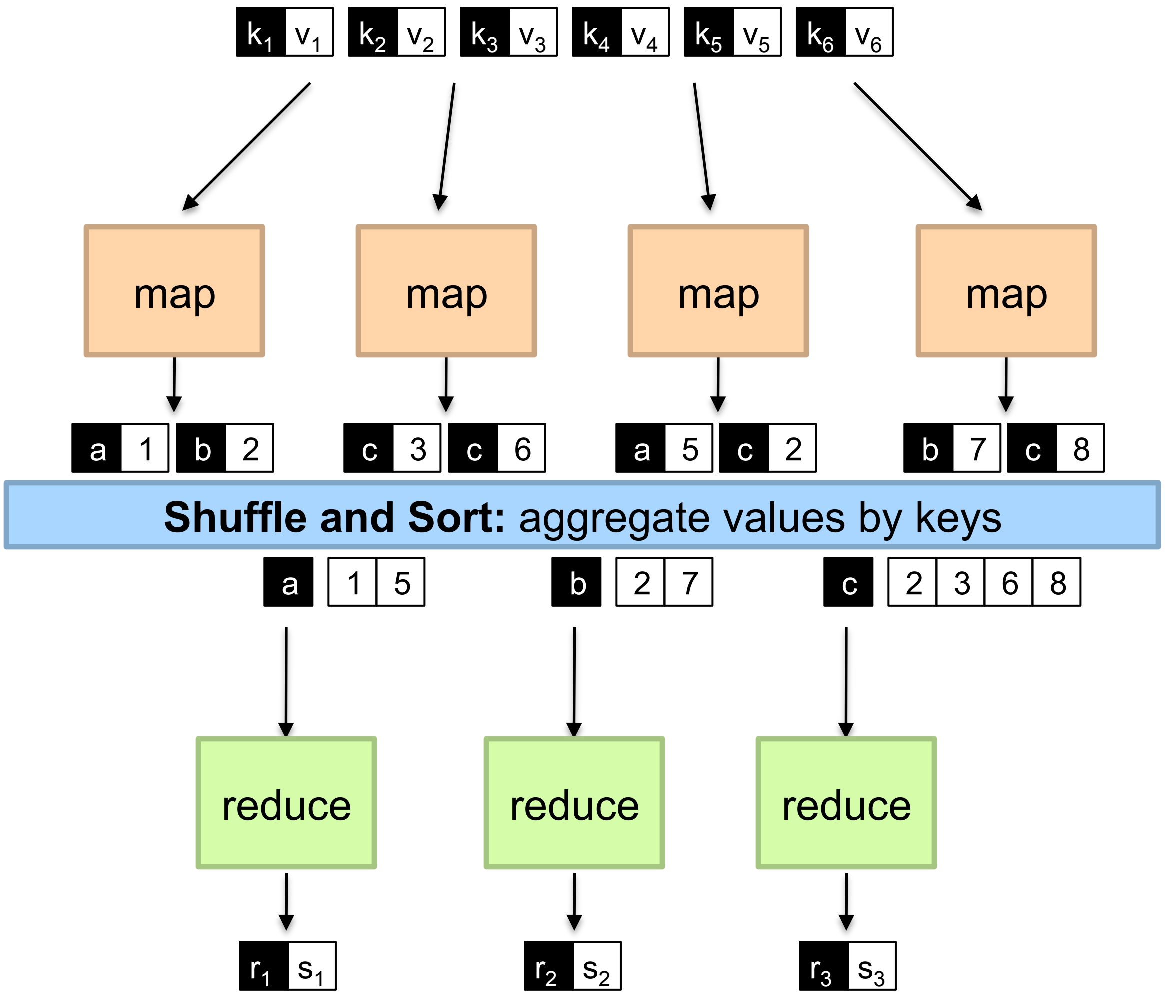
MapReduce

- Programmers specify two functions:

  map $(k_1, v_1) \rightarrow [<k_2, v_2>]$

  reduce $(k_2, [v_2]) \rightarrow [<k_3, v_3>]$

  - All values with the same key are sent to the same reducer
- The execution framework handles everything else…


k_1 v_1 k_2 v_2 k_3 v_3 k_4 v_4 k_5 v_5 k_6 v_6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 3 6 8
reduce
reduce
reduce
r_1 s_1
r_2 s_2
r_3 s_3

MapReduce

- Programmers specify two functions:

  map (k, v) → <k', v'>*

  reduce (k', v') → <k', v'>*

  - All values with the same key are sent to the same reducer
- The execution framework handles everything else…

What's "everything else"?

MapReduce “Runtime”

- Handles scheduling
  - Assigns workers to map and reduce tasks
- Handles “data distribution”
  - Moves processes to data
- Handles synchronization
  - Gathers, sorts, and shuffles intermediate data
- Handles errors and faults
  - Detects worker failures and restarts
- Everything happens on top of a distributed filesystem

MapReduce

- Programmers specify two functions:

  map (k, v) → <k', v'>*

  reduce (k', v') → <k', v'>*

  - All values with the same key are reduced together

- The execution framework handles everything else…

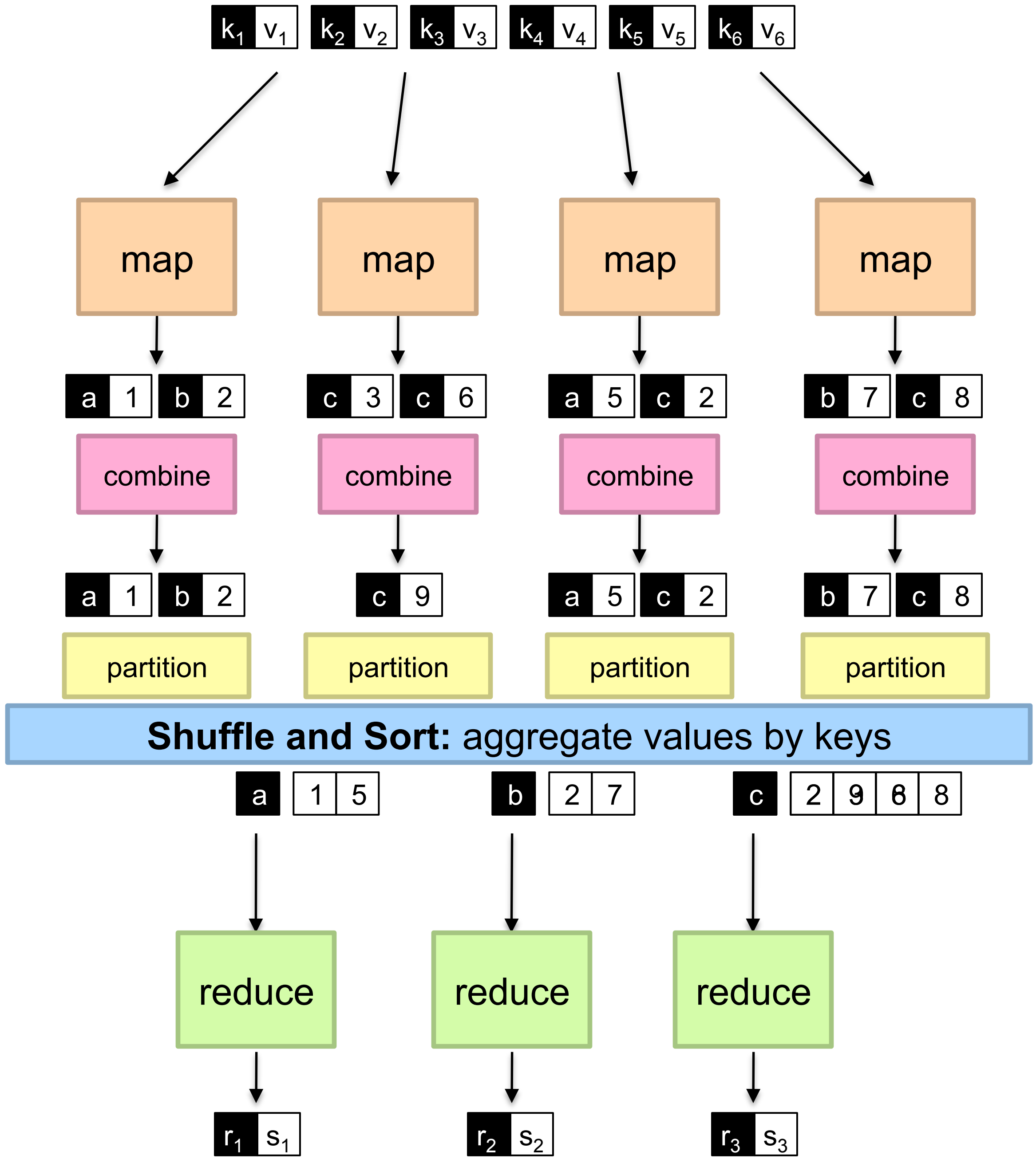
- Not quite…usually, programmers also specify:

  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



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 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

Two more details…

- Barrier between map and reduce phases
  - But intermediate data can be copied over as soon as mappers finish
- Keys arrive at each reducer in sorted order
  - No enforced ordering *across* reducers

What's the big deal?

- Developers need the right level of abstraction
  - Moving beyond the von Neumann architecture
  - We need better programming models
- Abstractions hide low-level details from the developers
  - No more race conditions, lock contention, etc.
- MapReduce separating the *what* from *how*
  - Developer specifies the computation that needs to be performed
  - Execution framework ("runtime") handles actual execution

Source: Google

Source: Google

MapReduce can refer to…

- The programming model
- The execution framework (aka “runtime”)
- The specific implementation

Usage is usually clear from context!

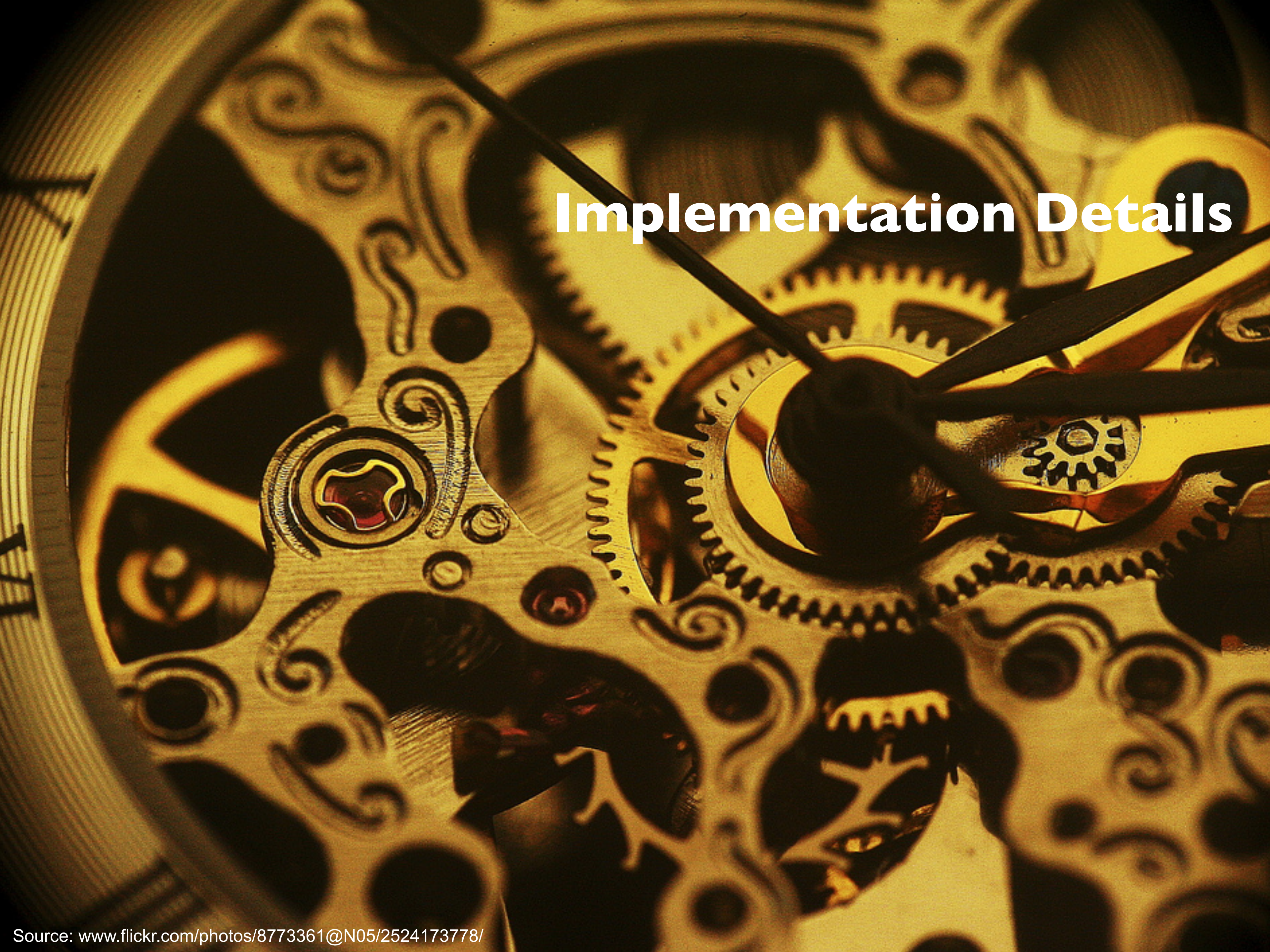
MapReduce Implementations

- Google has a proprietary implementation in C++
  - Bindings in Java, Python
- Hadoop is an open-source implementation in Java
  - Development led by Yahoo, now an Apache project
  - Used in production at Yahoo, Facebook, Twitter, LinkedIn, Netflix, …
  - The *de facto* big data processing platform
  - Rapidly expanding software ecosystem
- Lots of custom research implementations
  - For GPUs, cell processors, etc.

MapReduce algorithm design

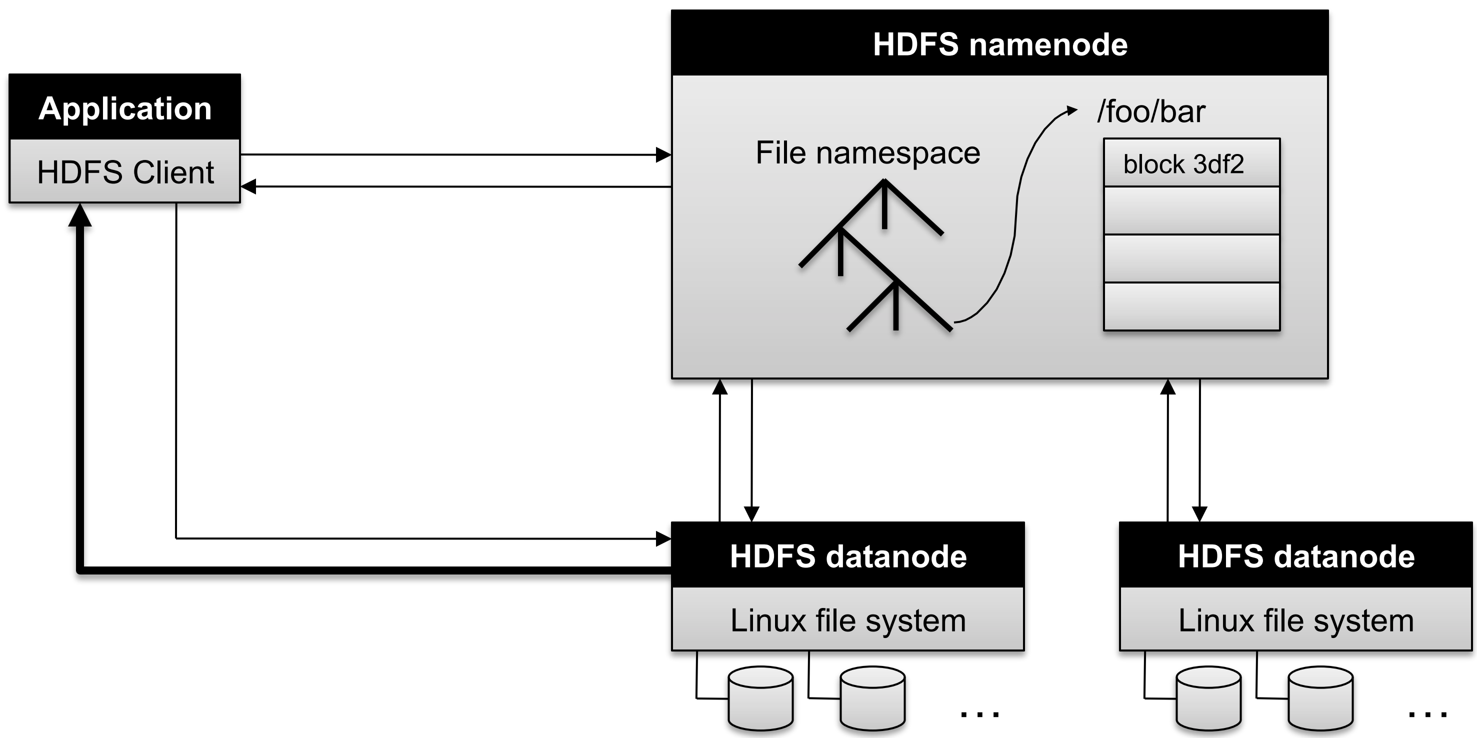
- The execution framework handles "everything else"…
  - Scheduling: assigns workers to map and reduce tasks
  - "Data distribution": moves processes to data
  - Synchronization: gathers, sorts, and shuffles intermediate data
  - Errors and faults: detects worker failures and restarts
- Limited control 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

Implementation Details



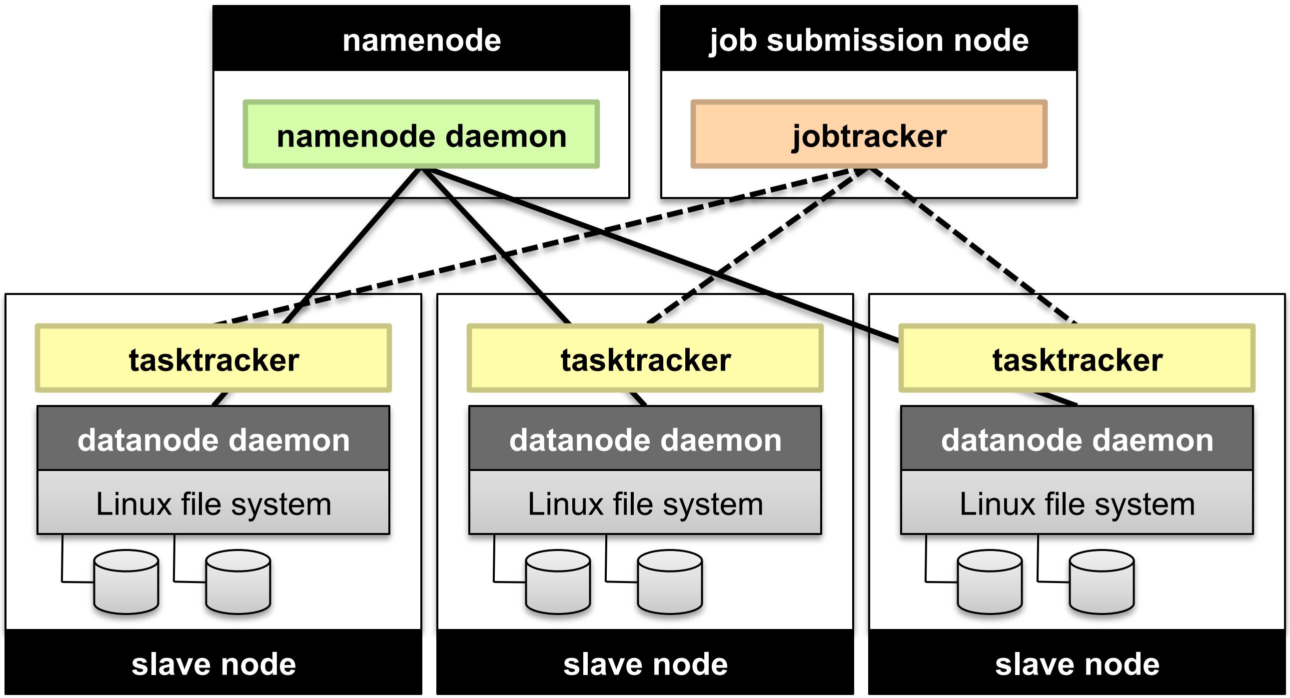
Source: www.flickr.com/photos/8773361@N05/2524173778/

HDFS Architecture

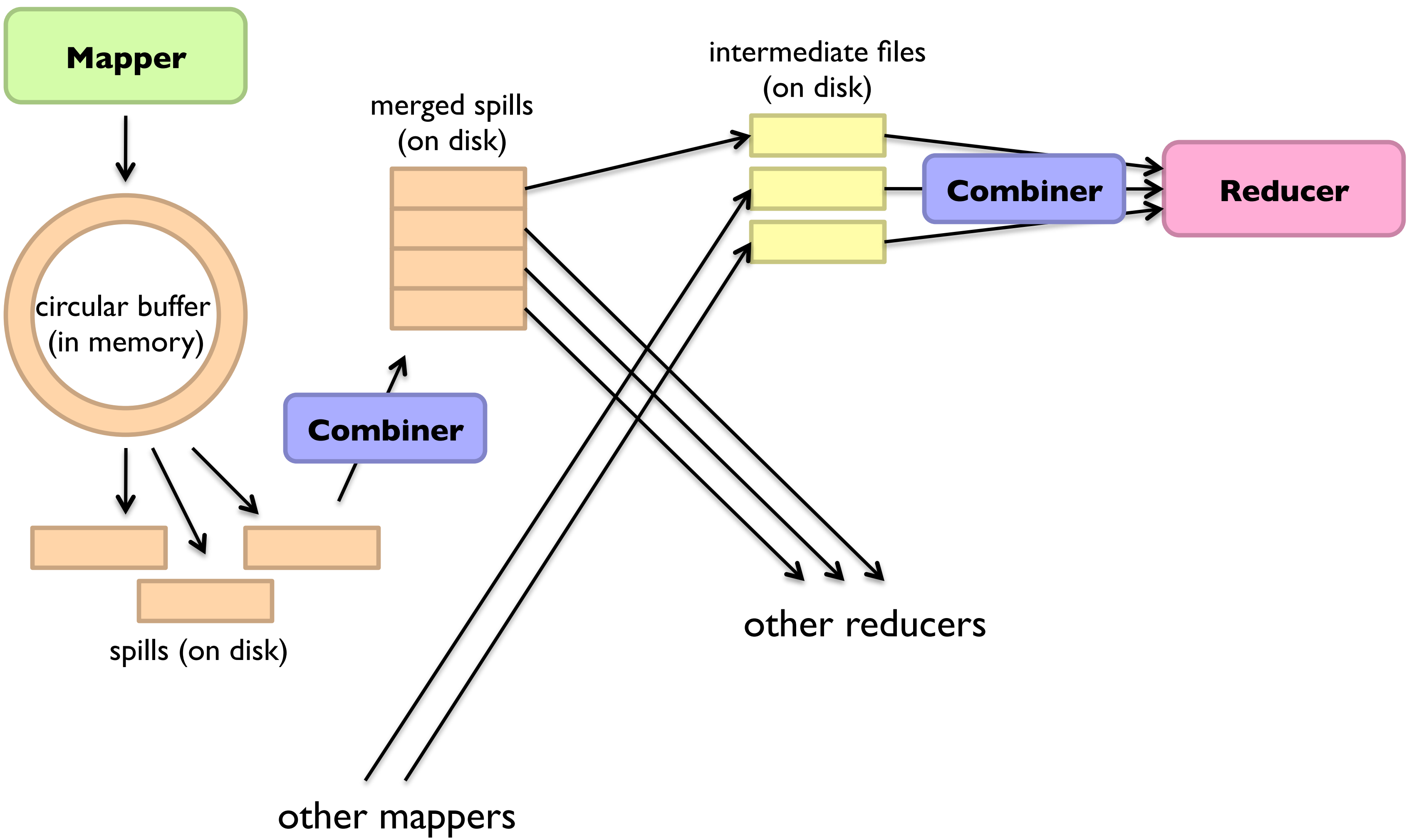


Adapted from (Ghemawat et al., SOSP 2003)

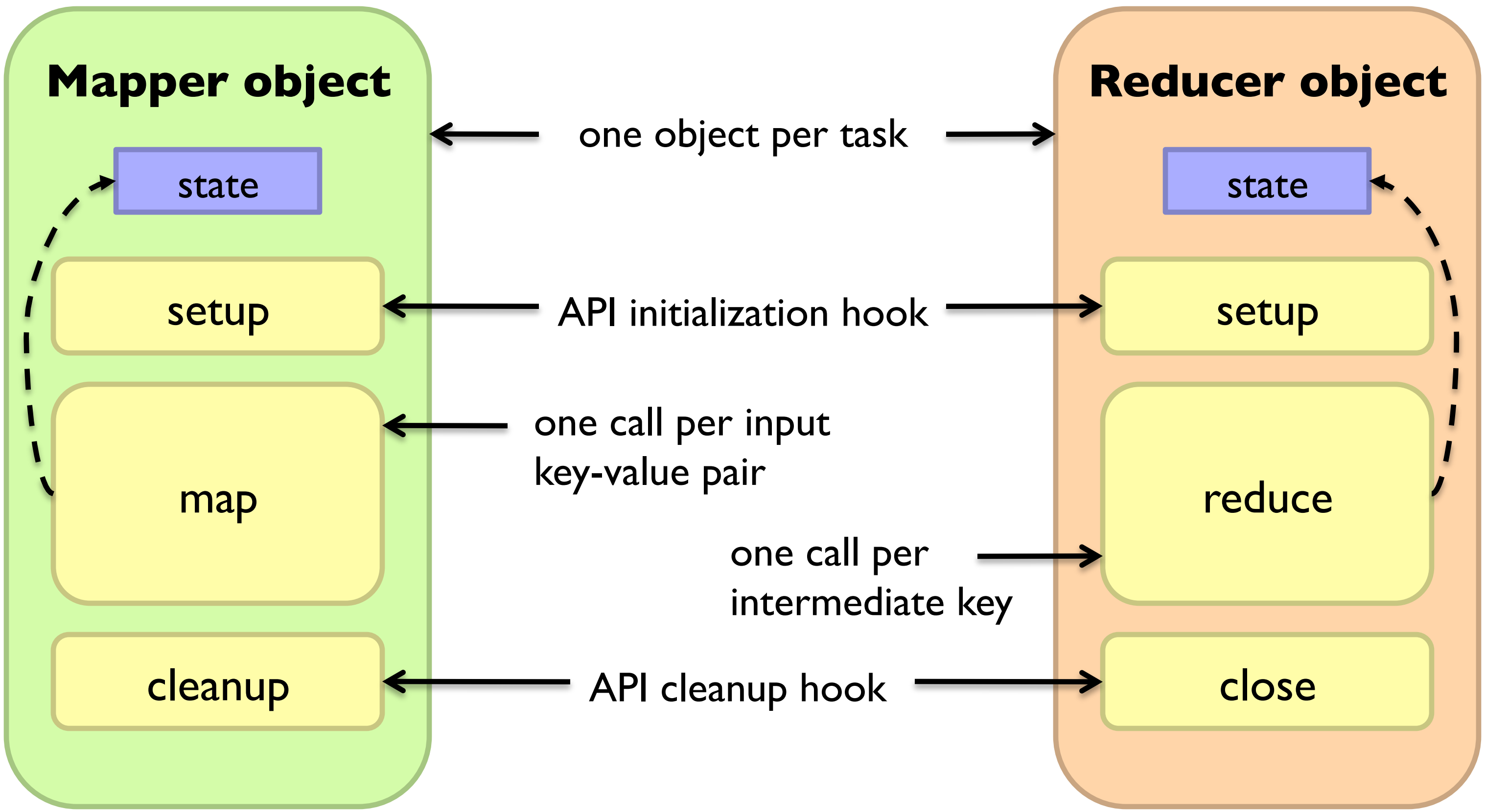
Putting everything together…



Shuffle and Sort



Preserving State



Implementation Don'ts

- Don't unnecessarily create objects
  - Object creation is costly
  - Garbage collection is costly
- Don't buffer objects
  - Processes have limited heap size (remember, commodity machines)
  - May work for small datasets, but won't scale!

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?

Source: www.flickr.com/photos/bunnieswithsharpteeth/490935152/

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 (network is slow!)
- Thus… avoid communication!
  - Reduce intermediate data via local aggregation
  - Combiners can help

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 idea: 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 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))

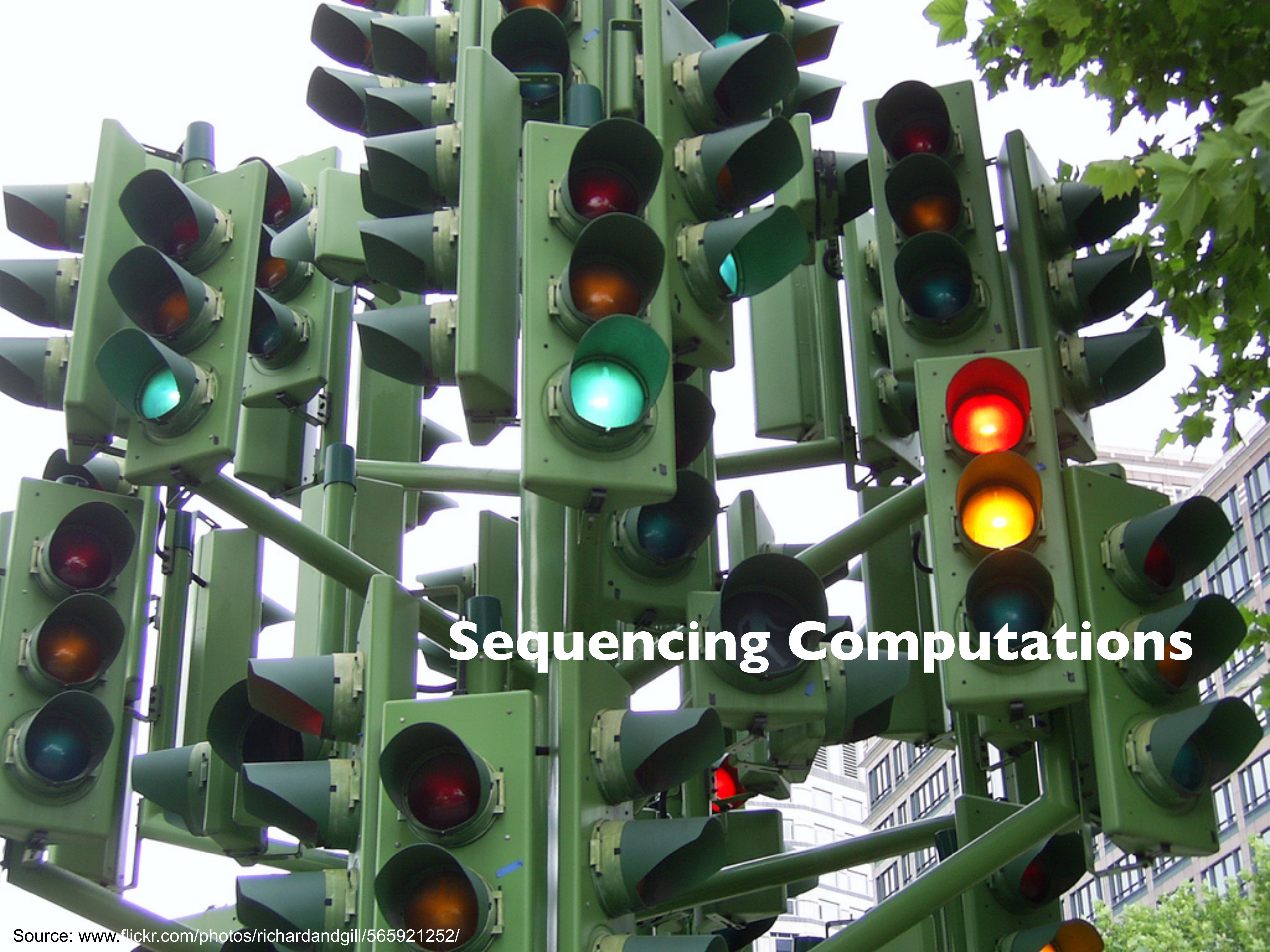
class Reducer
  method Reduce(string t, pairs $[(s_1, c_1), (s_2, c_2) \ldots]$)
    $sum \leftarrow 0$
    $cnt \leftarrow 0$
    for all pair $(s, c) \in$ pairs $[(s_1, c_1), (s_2, c_2) \ldots]$ **do**
      $sum \leftarrow sum + s$
      $cnt \leftarrow cnt + c$
    $r_{avg} \leftarrow sum/cnt$
 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



Sequencing Computations

Source: www.flickr.com/photos/richardandgill/565921252/

Sequencing Computations

1. Turn synchronization into a sorting problem
   - Leverage the fact that keys arrive at reducers in sorted order
   - Manipulate the sort order and partitioning scheme to deliver partial results at appropriate junctures
2. Create appropriate algebraic structures to capture computation
   - Build custom data structures to accumulate partial results

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
  - Basis for large classes of more sophisticated algorithms

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

$(a, b) \rightarrow 1$
$(a, c) \rightarrow 2$
$(a, d) \rightarrow 5$
$(a, e) \rightarrow 3$
$(a, f) \rightarrow 2$

$a \rightarrow \{ b: 1, c: 2, d: 5, e: 3, f: 2 \}$

- 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

$$
\begin{array}{ll}
 & a \rightarrow \{ b: 1, \quad\quad d: 5, e: 3 \} \\
+ & a \rightarrow \{ b: 1, c: 2, d: 2, \quad\quad f: 2 \} \\
\hline
 & a \rightarrow \{ b: 2, c: 2, d: 7, e: 3, f: 2 \}
\end{array}
$$

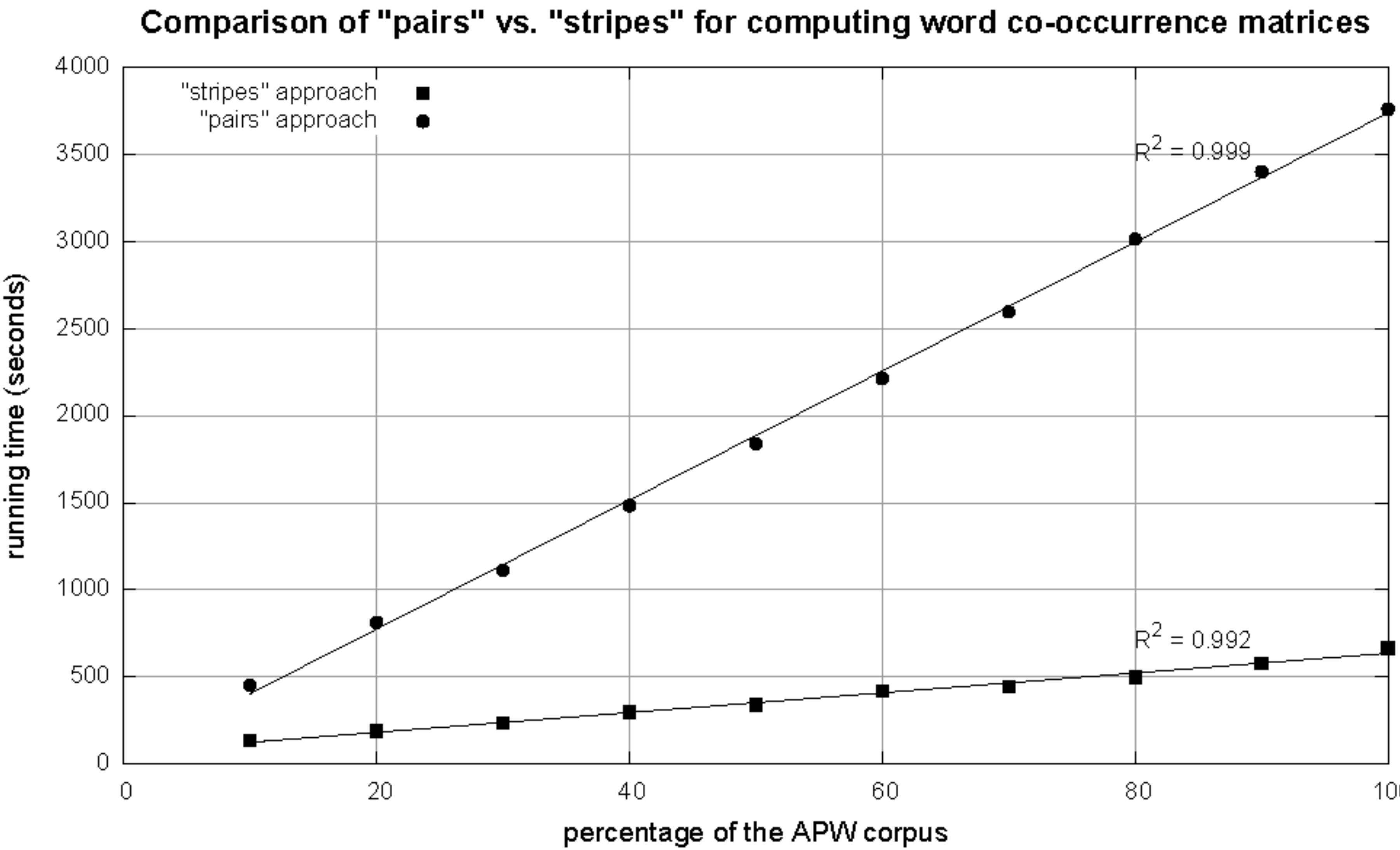
Key idea: cleverly-constructed data structure for aggregating partial results

Stripes: Pseudo-Code

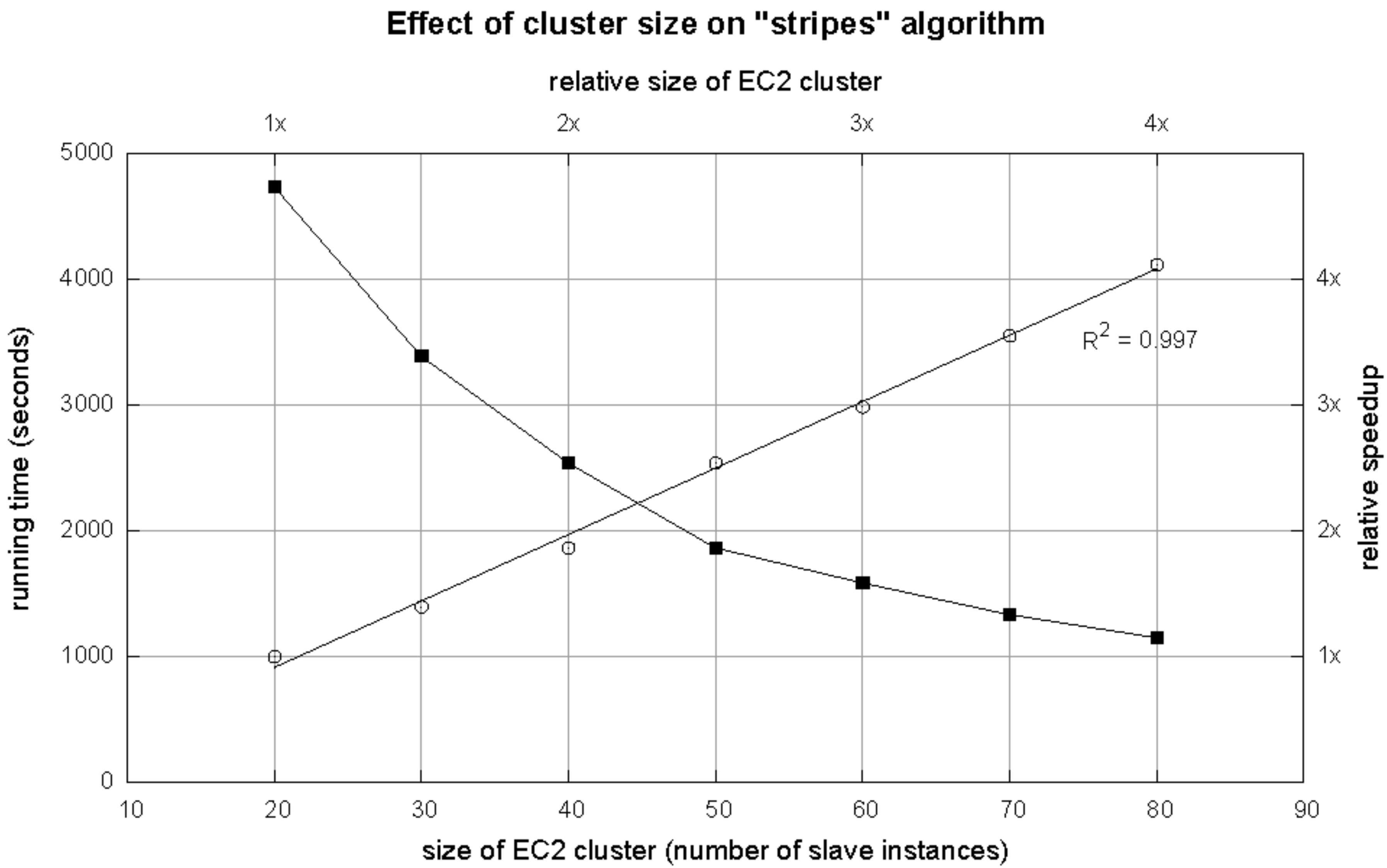
```
class Mapper
    method Map(docid a, doc d)
        for all term w ∈ doc d do
            H ← new AssociativeArray
            for all term u ∈ Neighbors(w) do
                H{u} ← H{u} + 1                ▷ Tally words co-occurring with w
            Emit(Term w, Stripe H)

class Reducer
    method Reduce(term w, stripes [H_1, H_2, H_3, ...])
        H_f ← new AssociativeArray
        for all stripe H ∈ stripes [H_1, H_2, H_3, ...] do
            Sum(H_f, H)                        ▷ Element-wise sum
        Emit(term w, stripe H_f)
```

“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
running time (seconds)
5000
4000
3000
2000
1000
0
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?

$$f(B|A) = \frac{N(A,B)}{N(A)} = \frac{N(A,B)}{\sum_{B'} N(A,B')}$$

- 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"

- What's the issue?
  - Computing relative frequencies requires marginal counts
  - But the marginal cannot be computed until you see all counts
  - Buffering is a bad idea!
- Solution:
  - What if we could get the marginal count to arrive at the reducer first?

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:
  - Take advantage of sorted key order at reducer to sequence computations
  - Get the marginal counts to arrive at the reducer before the joint counts
- Optimization:
  - Apply in-memory combining pattern to accumulate marginal counts

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 to accumulate partial results
  - 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

Source: http://www.flickr.com/photos/guvnah/7861418602/

Hidden Markov Models

An HMM $\lambda = (A, B, \Pi)$ is characterized by:

- N states: $Q = \{q_1, q_2, \ldots q_N\}$
- N x N Transition probability matrix $A = [a_{ij}]$

$$a_{ij} = p(q_j|q_i) \qquad \sum_j a_{ij} = 1 \quad \forall i$$

- V observation symbols: $O = \{o_1, o_2, \ldots o_V\}$
- N x |V| Emission probability matrix $B = [b_{iv}]$

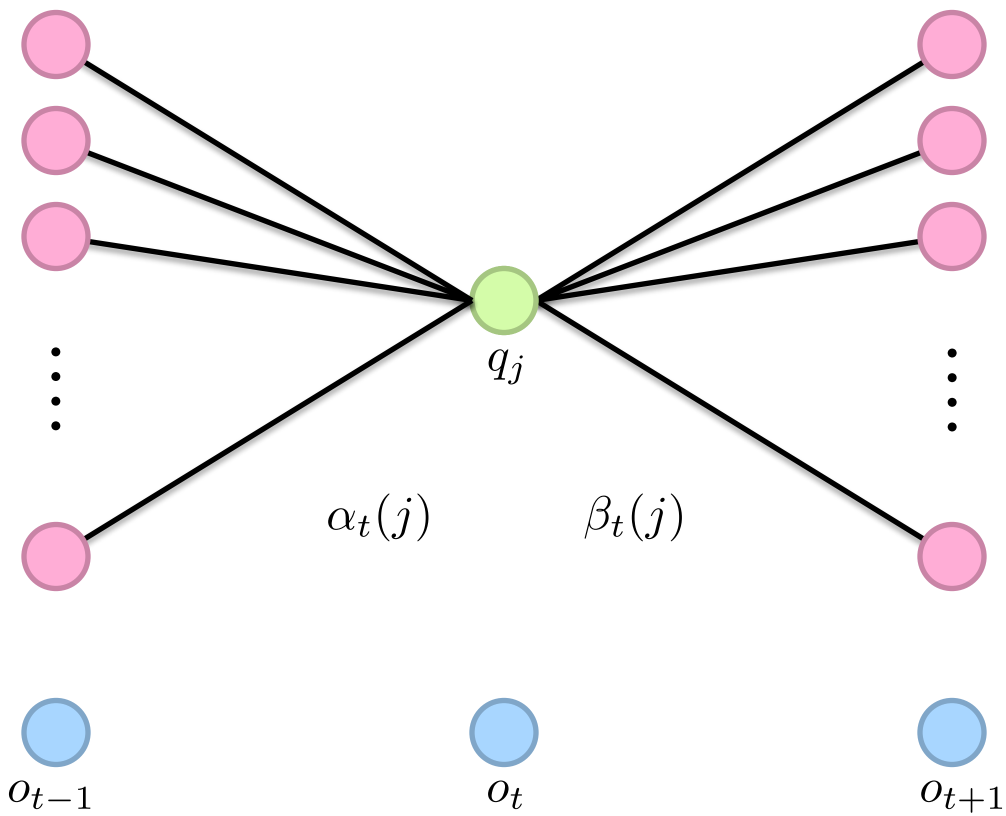
$$b_{iv} = b_i(o_v) = p(o_v|q_i)$$

- Prior probabilities vector $\Pi = [\pi_i, \pi_2, \ldots \pi_N]$

$$\sum_{i=1}^{N} \pi_i = 1$$

Forward-Backward

$$\alpha_t(j) = P(o_1, o_2 \ldots o_t, q_t = j|\lambda) \qquad \beta_t(j) = P(o_{t+1}, o_{t+2} \ldots o_T|q_t = i, \lambda)$$



Estimating Emissions Probabilities

- Basic idea:

$$b_j(v_k) = \frac{\text{expected number of times in state } j \text{ and observing symbol } v_k}{\text{expected number of times in state } j}$$

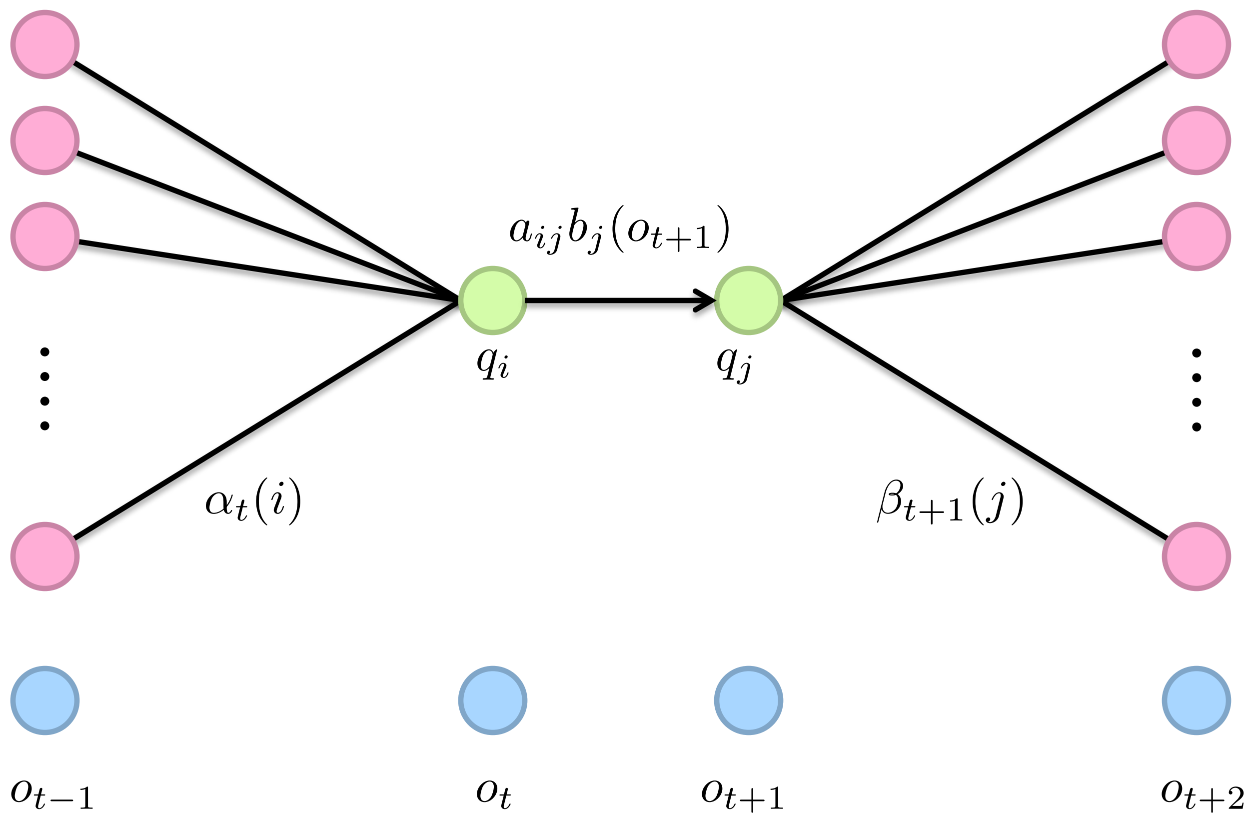
- Let's define:

$$\gamma_t(j) = \frac{P(q_t = j, O|\lambda)}{P(O|\lambda)} = \frac{\alpha_t(j)\beta_t(j)}{P(O|\lambda)}$$

- Thus:

$$\hat{b}_j(v_k) = \frac{\sum_{i=1 \cap O_t = v_k}^{T} \gamma_t(j)}{\sum_{i=1}^{T} \gamma_t(j)}$$

Forward-Backward



Estimating Transition Probabilities

- Basic idea:

$$a_{ij} = \frac{\text{expected number of transitions from state } i \text{ to state } j}{\text{expected number of transitions from state } i}$$

- Let's define:

$$\xi_t(i,j) = \frac{\alpha_t(i) a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)}{P(O|\lambda)}$$

- Thus:

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \sum_{j=1}^{N} \xi_t(i,j)}$$

MapReduce Implementation: Mapper

class Mapper
  method Initialize(integer *iteration*)
    $\langle \mathcal{S}, \mathcal{O} \rangle \leftarrow$ ReadModel
    $\theta \leftarrow \langle A, B, \pi \rangle \leftarrow$ ReadModelParams(*iteration*)
  method Map(sample *id*, sequence $\mathbf{x}$)
    $\alpha \leftarrow$ Forward($\mathbf{x}, \theta$)
    $\beta \leftarrow$ Backward($\mathbf{x}, \theta$)
    $I \leftarrow$ new AssociativeArray
    for all $q \in \mathcal{S}$ **do**
      $I\{q\} \leftarrow \alpha_1(q) \cdot \beta_1(q)$
    $O \leftarrow$ new AssociativeArray of AssociativeArray
    for $t = 1$ to $|\mathbf{x}|$ **do**
      for all $q \in \mathcal{S}$ **do**
        $O\{q\}\{x_t\} \leftarrow O\{q\}\{x_t\} + \alpha_t(q) \cdot \beta_t(q)$
      $t \leftarrow t + 1$
    $T \leftarrow$ new AssociativeArray of AssociativeArray
    for $t = 1$ to $|\mathbf{x}| - 1$ **do**
      for all $q \in \mathcal{S}$ **do**
        for all $r \in \mathcal{S}$ **do**
          $T\{q\}\{r\} \leftarrow T\{q\}\{r\} + \alpha_t(q) \cdot A_q(r) \cdot B_r(x_{t+1}) \cdot \beta_{t+1}(r)$
      $t \leftarrow t + 1$
    Emit(string '*initial*', stripe I)
    for all $q \in \mathcal{S}$ **do**
      Emit(string '*emit from* ' + q, stripe $O\{q\}$)
      Emit(string '*transit from* ' + q, stripe $T\{q\}$)

$$\hat{b}_j(v_k) = \frac{\sum_{i=1 \cap O_t = v_k}^{T} \gamma_t(j)}{\sum_{i=1}^{T} \gamma_t(j)}$$

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \sum_{j=1}^{N} \xi_t(i,j)}$$

$$\gamma_t(j) = \frac{\alpha_t(j)\beta_t(j)}{P(O|\lambda)}$$

$$\xi_t(i,j) = \frac{\alpha_t(i) a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)}{P(O|\lambda)}$$

MapReduce Implementation: Reducer

class COMBINER
  method COMBINE(string t, stripes $[C_1, C_2, \ldots]$)
    $C_f \leftarrow$ new ASSOCIATIVEARRAY
    for all stripe $C \in$ stripes $[C_1, C_2, \ldots]$ **do**
      SUM(C_f, C)
    EMIT(string t, stripe C_f)

class REDUCER
  method REDUCE(string t, stripes $[C_1, C_2, \ldots]$)
    $C_f \leftarrow$ new ASSOCIATIVEARRAY
    for all stripe $C \in$ stripes $[C_1, C_2, \ldots]$ **do**
      SUM(C_f, C)
    $z \leftarrow 0$
    for all $\langle k, v \rangle \in C_f$ **do**
      $z \leftarrow z + v$
    $P_f \leftarrow$ new ASSOCIATIVEARRAY
    for all $\langle k, v \rangle \in C_f$ **do**
      $P_f\{k\} \leftarrow v/z$
    EMIT(string t, stripe P_f)

$$\hat{b}_j(v_k) = \frac{\sum_{i=1 \cap O_t = v_k}^{T} \gamma_t(j)}{\sum_{i=1}^{T} \gamma_t(j)}$$

$$\hat{a}_{ij} = \frac{\sum_{t=1}^{T-1} \xi_t(i,j)}{\sum_{t=1}^{T-1} \sum_{j=1}^{N} \xi_t(i,j)}$$

$$\gamma_t(j) = \frac{\alpha_t(j)\beta_t(j)}{P(O|\lambda)}$$

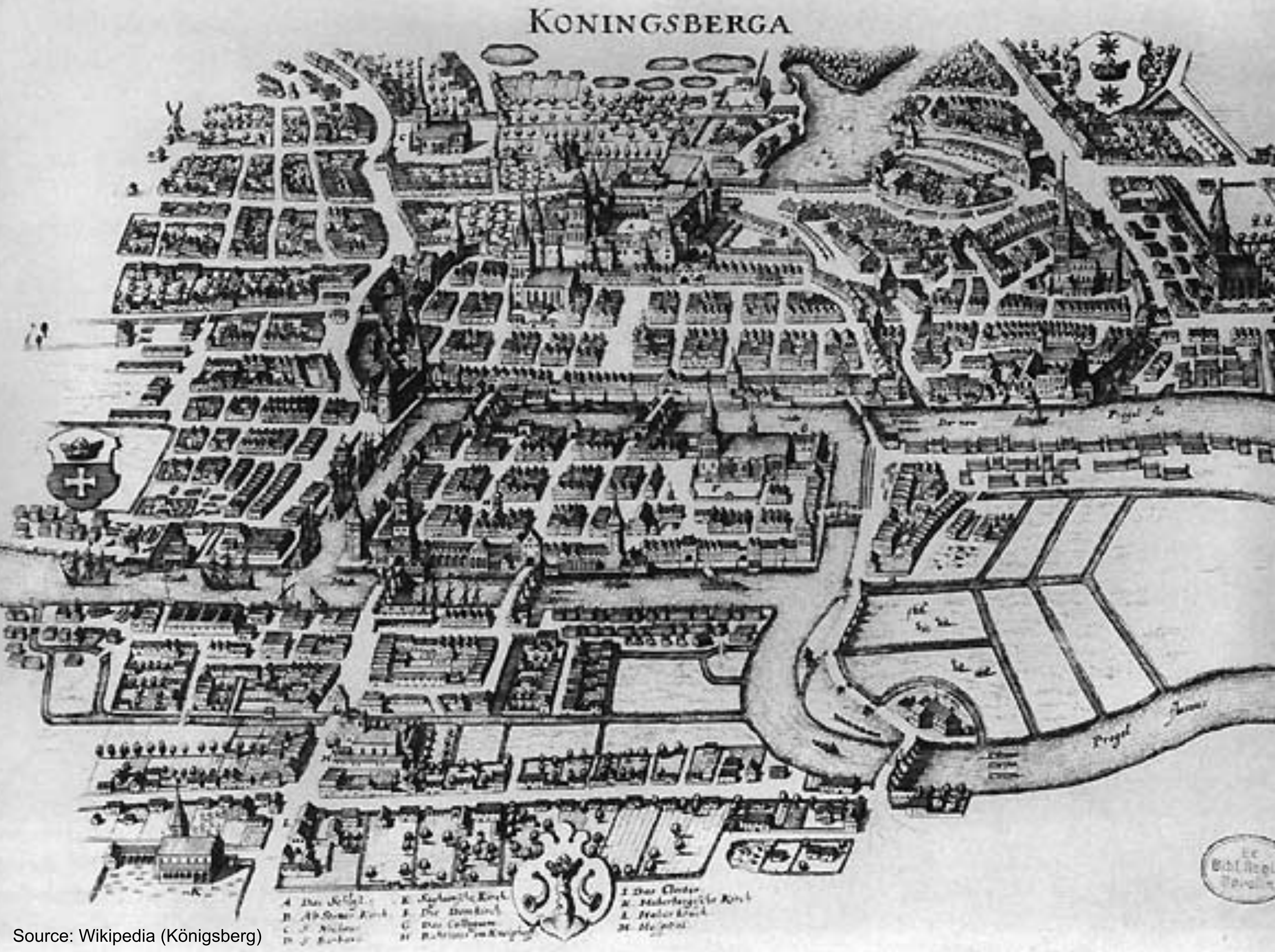
$$\xi_t(i,j) = \frac{\alpha_t(i) a_{ij} b_j(o_{t+1}) \beta_{t+1}(j)}{P(O|\lambda)}$$

Iterative Algorithms: Graphs

Source: Wikipedia (Water wheel)

What's a graph?

- G = (V,E), where
  - V represents the set of vertices (nodes)
  - E represents the set of edges (links)
  - Both vertices and edges may contain additional information
- Different types of graphs:
  - Directed vs. undirected edges
  - Presence or absence of cycles
- Graphs are everywhere:
  - Hyperlink structure of the web
  - Physical structure of computers on the Internet
  - Interstate highway system
  - Social networks



Source: Wikipedia (Königsberg)

Source: Wikipedia (Kaliningrad)

Some Graph Problems

- Finding shortest paths
  - Routing Internet traffic and UPS trucks
- Finding minimum spanning trees
  - Telco laying down fiber
- Finding Max Flow
  - Airline scheduling
- Identify "special" nodes and communities
  - Breaking up terrorist cells, spread of avian flu
- Bipartite matching
  - Monster.com, Match.com
- And of course... PageRank

Graphs and MapReduce

- A large class of graph algorithms involve:
  - Performing computations at each node: based on node features, edge features, and local link structure
  - Propagating computations: "traversing" the graph
- Key questions:
  - How do you represent graph data in MapReduce?
  - How do you traverse a graph in MapReduce?

In reality: graph algorithms in MapReduce suck!

Representing Graphs

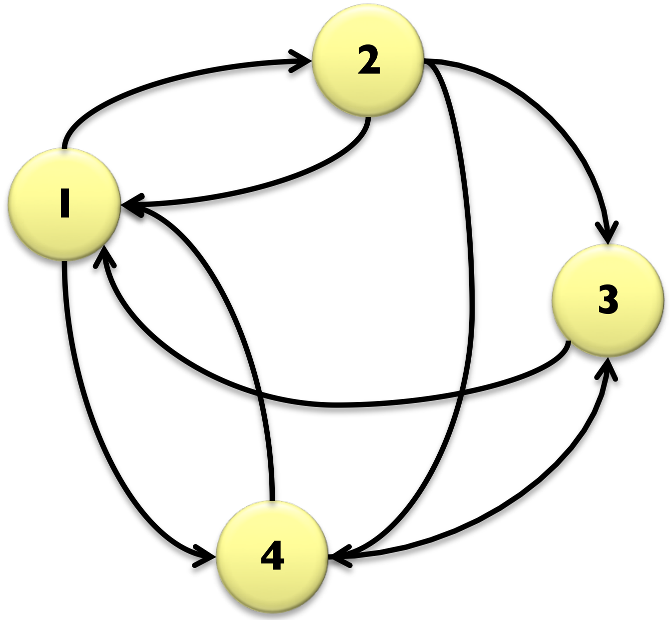
- G = (V, E)
- Two common representations
  - Adjacency matrix
  - Adjacency list

Adjacency Matrices

Represent a graph as an *n* x *n* square matrix *M*

- $n = |V|$
- $M_{ij} = 1$ means a link from node *i* to *j*

	1	2	3	4
1	0	1	0	1
2	1	0	1	1
3	1	0	0	0
4	1	0	1	0



Adjacency Matrices: Critique

- Advantages:
  - Amenable to mathematical manipulation
  - Iteration over rows and columns corresponds to computations on outlinks and inlinks
- Disadvantages:
  - Lots of zeros for sparse matrices
  - Lots of wasted space

Adjacency Lists

Take adjacency matrices… and throw away all the zeros

	1	**2**	**3**	**4**
1	0	1	0	1
2	1	0	1	1
3	1	0	0	0
4	1	0	1	0

→

1: 2, 4
2: 1, 3, 4
3: 1
4: 1, 3

Adjacency Lists: Critique

- Advantages:
  - Much more compact representation
  - Easy to compute over outlinks
- Disadvantages:
  - Much more difficult to compute over inlinks

Single-Source Shortest Path

- **Problem:** find shortest path from a source node to one or more target nodes
  - Shortest might also mean lowest weight or cost
- Single processor machine: Dijkstra's Algorithm
- MapReduce: parallel breadth-first search (BFS)

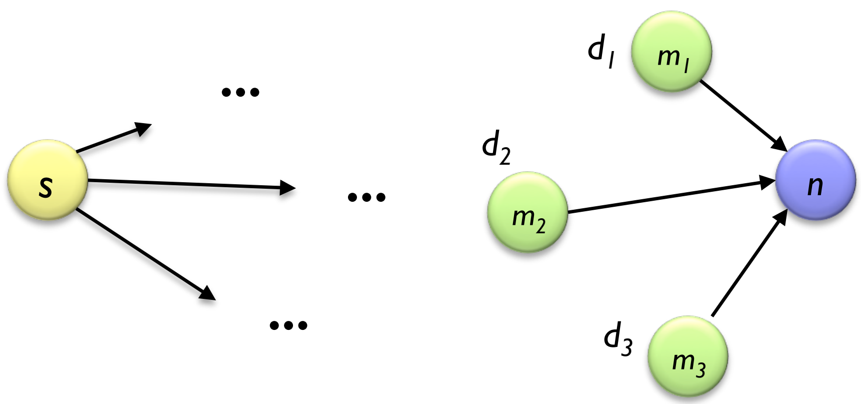
Finding the Shortest Path

- Consider simple case of equal edge weights
- Solution to the problem can be defined inductively
- Here's the intuition:
  - Define: *b* is reachable from *a* if *b* is on adjacency list of *a*

    DistanceTo(s) = 0
  - For all nodes p reachable from s,

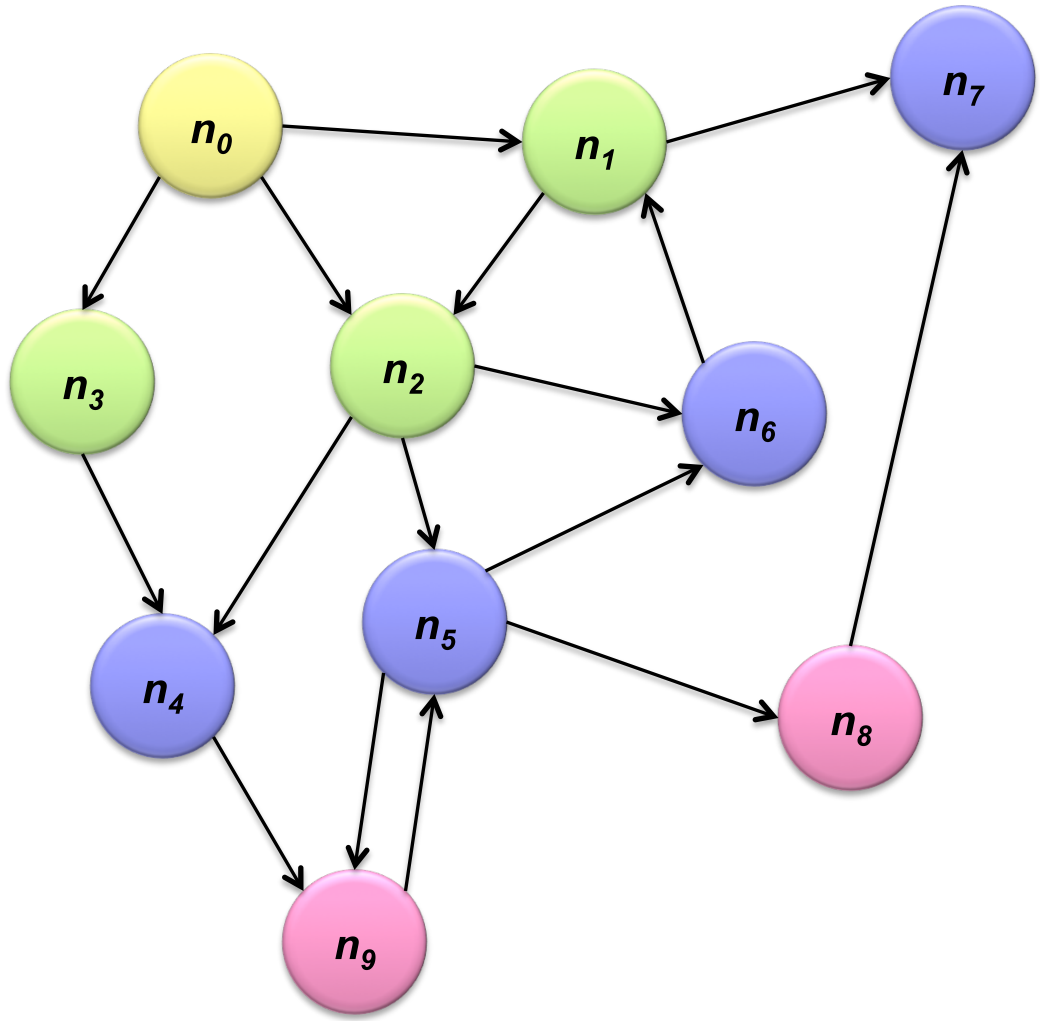
    DistanceTo(p) = 1
  - For all nodes n reachable from some other set of nodes M,

    DistanceTo(n) = 1 + min(DistanceTo(m), $m \in M$)




Source: Wikipedia (Wave)

Visualizing Parallel BFS



From Intuition to Algorithm

- Data representation:
  - Key: node *n*
  - Value: *d* (distance from start), adjacency list (nodes reachable from *n*)
  - Initialization: for all nodes except for start node, $d = \infty$
- Mapper:
  - $\forall m \in$ adjacency list: emit $(m, d + 1)$
- Sort/Shuffle
  - Groups distances by reachable nodes
- Reducer:
  - Selects minimum distance path for each reachable node
  - Additional bookkeeping needed to keep track of actual path

Multiple Iterations Needed

- Each MapReduce iteration advances the “frontier” by one hop
  - Subsequent iterations include more and more reachable nodes as frontier expands
  - Multiple iterations are needed to explore entire graph
- Preserving graph structure:
  - Problem: Where did the adjacency list go?
  - Solution: mapper emits (*n*, adjacency list) as well

BFS Pseudo-Code

```
class Mapper
    method Map(nid n, node N)
        d ← N.Distance
        Emit(nid n, N)                                  ▷ Pass along graph structure
        for all nodeid m ∈ N.AdjacencyList do
            Emit(nid m, d + 1)                          ▷ Emit distances to reachable nodes

class Reducer
    method Reduce(nid m, [d1, d2, . . .])
        d_min ← ∞
        M ← ∅
        for all d ∈ counts [d1, d2, . . .] do
            if IsNode(d) then
                M ← d                                   ▷ Recover graph structure
            else if d < d_min then                      ▷ Look for shorter distance
                d_min ← d
        M.Distance ← d_min                              ▷ Update shortest distance
        Emit(nid m, node M)
```

Stopping Criterion

- When a node is first discovered, we've found the shortest path
  - Maximum number of iterations is equal to the diameter of the graph
- Practicalities of implementation in MapReduce

Comparison to Dijkstra

- Dijkstra's algorithm is more efficient
  - At each step, only pursues edges from minimum-cost path inside frontier
- MapReduce explores all paths in parallel
  - Lots of "waste"
  - Useful work is only done at the "frontier"
- Why can't we do better using MapReduce?

Single Source: Weighted Edges

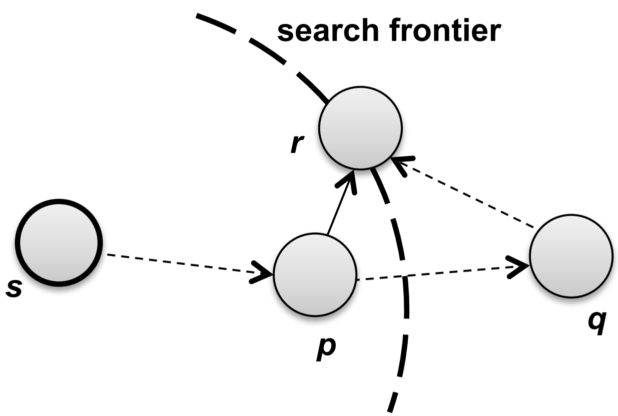
- Now add positive weights to the edges
  - Why can't edge weights be negative?
- Simple change: add weight *w* for each edge in adjacency list
  - In mapper, emit (m, $d + w_p$) instead of (m, $d + 1$) for each node *m*
- That's it?

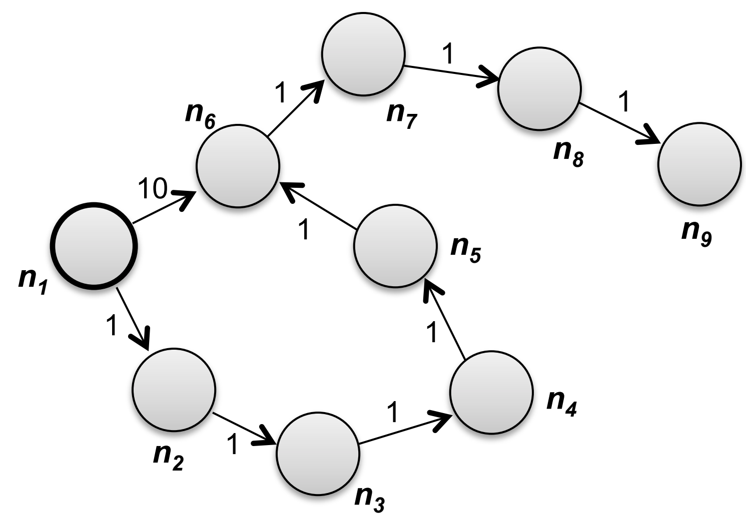
Stopping Criterion

- How many iterations are needed in parallel BFS (positive edge weight case)?
- When a node is first discovered, we've found the shortest path

Not true!

Additional Complexities





Stopping Criterion

- How many iterations are needed in parallel BFS (positive edge weight case)?
- Practicalities of implementation in MapReduce

All-Pairs?

- Floyd-Warshall Algorithm: difficult to MapReduce-ify…
- Multiple-source shortest paths in MapReduce: run multiple parallel BFS *simultaneously*
  - Assume source nodes $\{s_0, s_1, \ldots s_n\}$
  - Instead of emitting a single distance, emit an array of distances, with respect to each source
  - Reducer selects minimum for each element in array
- Does this scale?

Application: Social Search

Source: Wikipedia (Crowd)

Social Search

- When searching, how to rank friends named "John"?
  - Assume undirected graphs
  - Rank matches by distance to user
- Naïve implementations:
  - Precompute all-pairs distances
  - Compute distances at query time
- Can we do better?

Landmark Approach (aka sketches)

- Select *n* seeds $\{s_0, s_1, \ldots s_n\}$
- Compute distances from seeds to every node:

  A = [2, 1, 1]
  B = [1, 1, 2]
  C = [4, 3, 1]
  D = [1, 2, 4]

  - What can we conclude about distances?
  - Insight: landmarks bound the maximum path length
- Lots of details:
  - How to more tightly bound distances
  - How to select landmarks (random isn't the best…)
- Use multi-source parallel BFS implementation in MapReduce!

Source: Wikipedia (Wave)

Graphs and MapReduce

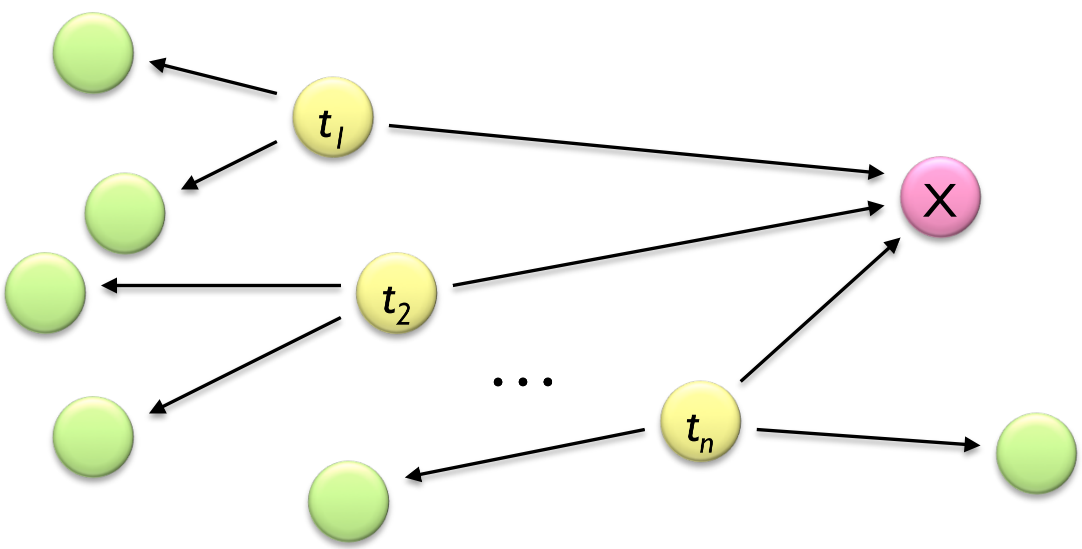
- A large class of graph algorithms involve:
  - Performing computations at each node: based on node features, edge features, and local link structure
  - Propagating computations: “traversing” the graph
- Generic recipe:
  - Represent graphs as adjacency lists
  - Perform local computations in mapper
  - Pass along partial results via outlinks, keyed by destination node
  - Perform aggregation in reducer on inlinks to a node
  - Iterate until convergence: controlled by external “driver”
  - Don’t forget to pass the graph structure between iterations

PageRank

Given page x with inlinks $t_1 \ldots t_n$, where

- $C(t)$ is the out-degree of t
- α is probability of random jump
- N is the total number of nodes in the graph

$$PR(x) = \alpha \left(\frac{1}{N} \right) + (1 - \alpha) \sum_{i=1}^{n} \frac{PR(t_i)}{C(t_i)}$$



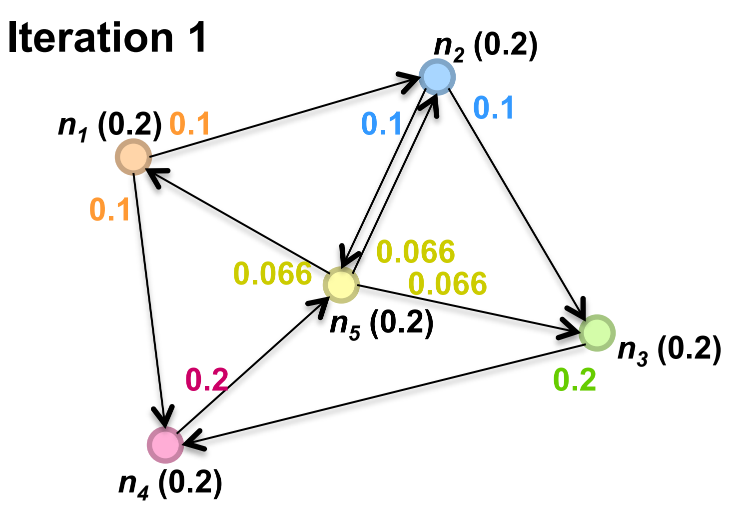
Computing PageRank

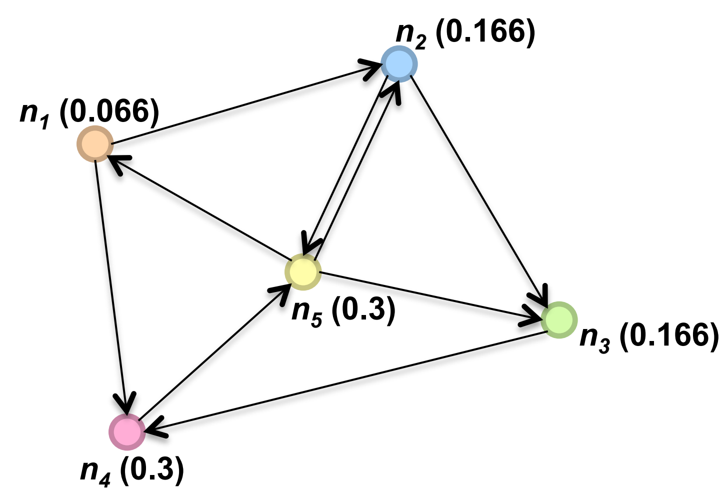
- Properties of PageRank
  - Can be computed iteratively
  - Effects at each iteration are local
- Sketch of algorithm:
  - Start with seed PR_i values
  - Each page distributes PR_i “credit” to all pages it links to
  - Each target page adds up “credit” from multiple in-bound links to compute PR_{i+1}
  - Iterate until values converge

Simplified PageRank

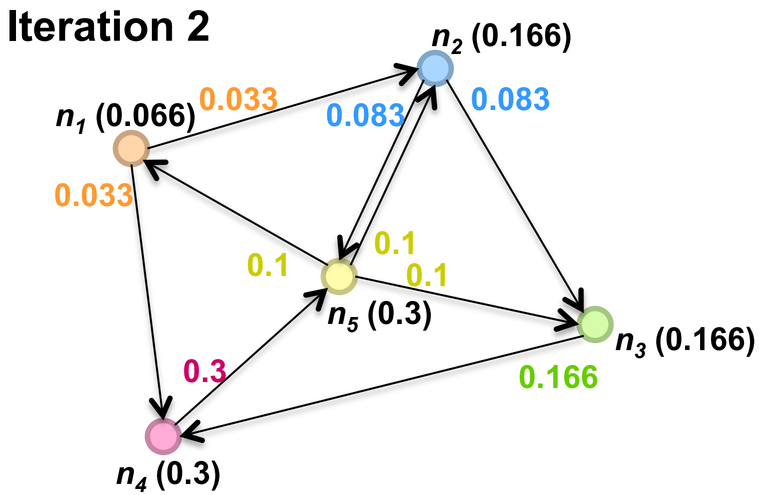
- First, tackle the simple case:
  - No random jump factor
  - No dangling nodes
- Then, factor in these complexities…
  - Why do we need the random jump?
  - Where do dangling nodes come from?

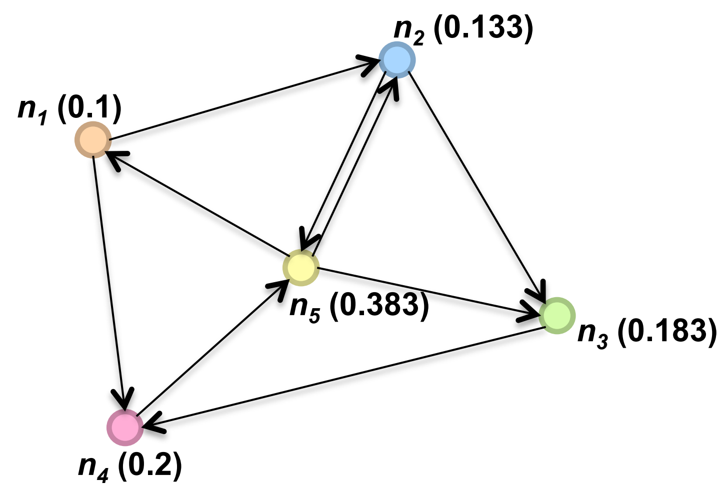
Sample PageRank Iteration (1)



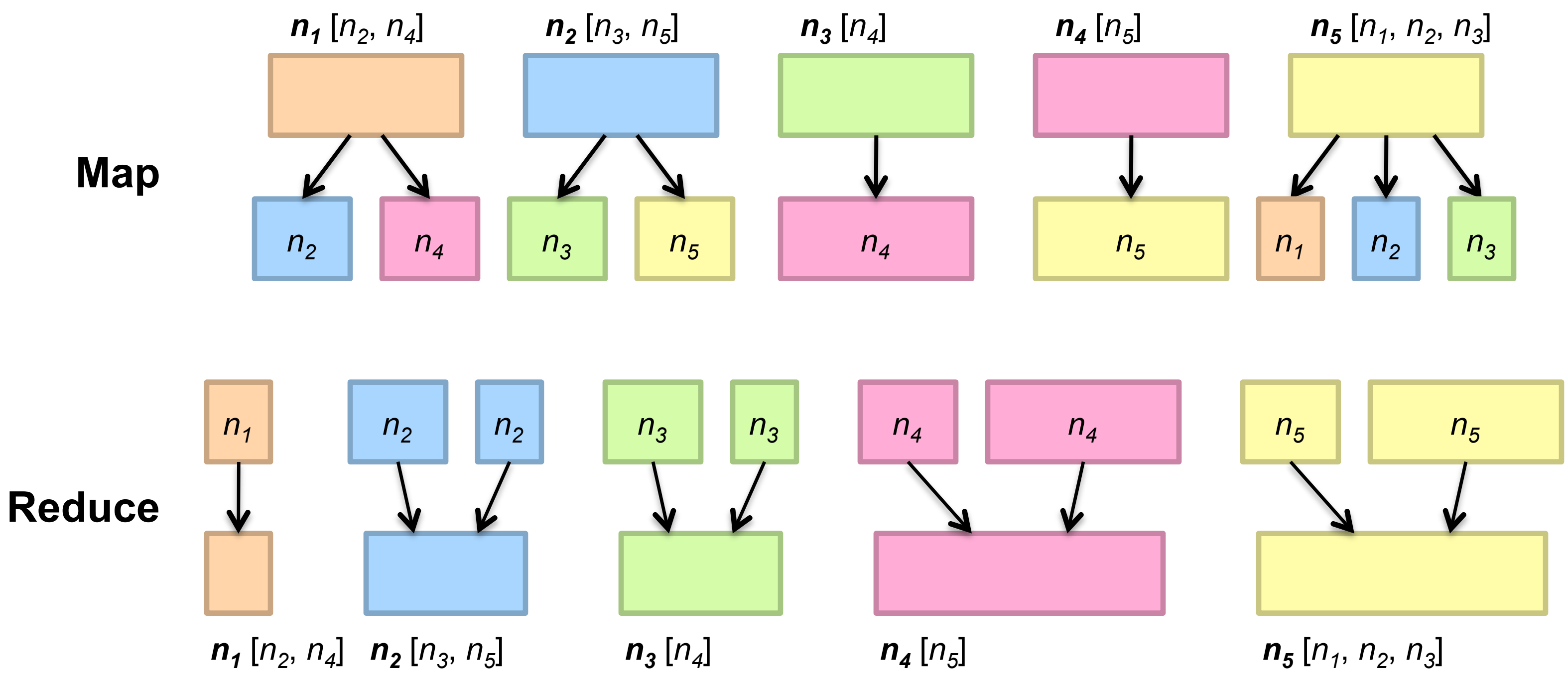


Sample PageRank Iteration (2)





PageRank in MapReduce



PageRank Pseudo-Code

```
class Mapper
    method Map(nid n, node N)
        p ← N.PageRank/|N.AdjacencyList|
        Emit(nid n, N)                                  ▷ Pass along graph structure
        for all nodeid m ∈ N.AdjacencyList do
            Emit(nid m, p)                              ▷ Pass PageRank mass to neighbors
```

```
class Reducer
    method Reduce(nid m, [p1, p2, . . .])
        M ← ∅
        for all p ∈ counts [p1, p2, . . .] do
            if IsNode(p) then
                M ← p                                   ▷ Recover graph structure
            else
                s ← s + p                               ▷ Sums incoming PageRank contributions
        M.PageRank ← s
        Emit(nid m, node M)
```

Complete PageRank

- Two additional complexities
  - What is the proper treatment of dangling nodes?
  - How do we factor in the random jump factor?
- Solution:
  - Second pass to redistribute "missing PageRank mass" and account for random jumps

$$p' = \alpha \left(\frac{1}{N} \right) + (1 - \alpha) \left(\frac{m}{N} + p \right)$$

  - *p* is PageRank value from before, *p*' is updated PageRank value
  - *N* is the number of nodes in the graph
  - *m* is the missing PageRank mass
- Additional optimization: make it a single pass!

PageRank Convergence

- Alternative convergence criteria
  - Iterate until PageRank values don't change
  - Iterate until PageRank rankings don't change
  - Fixed number of iterations
- Convergence for web graphs?
  - Not a straightforward question
- Watch out for link spam:
  - Link farms
  - Spider traps
  - …

Beyond PageRank

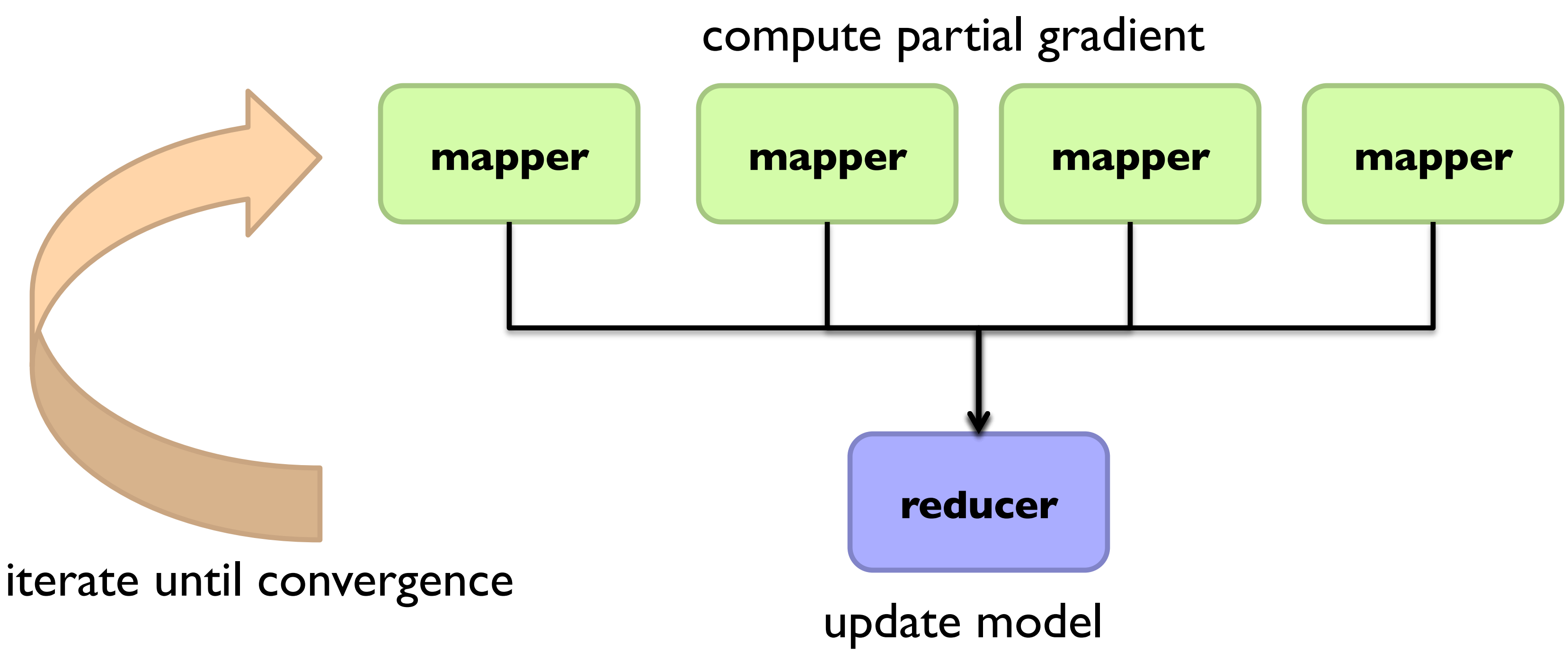
- Variations of PageRank
  - Weighted edges
  - Personalized PageRank
- Variants on graph random walks
  - Hubs and authorities (HITS)
  - SALSA

Other Classes of Graph Algorithms

- Subgraph pattern matching
- Computing simple graph statistics
  - Degree vertex distributions
- Computing more complex graph statistics
  - Clustering coefficients
  - Counting triangles

Batch Gradient Descent in MapReduce

$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \gamma^{(t)} \frac{1}{n} \underbrace{\underbrace{\sum_{i=0}^{n} \nabla \ell(f(\mathrm{x}_i; \theta^{(t)}), y_i)}_{\text{mappers}}}_{\text{single reducer}}$$



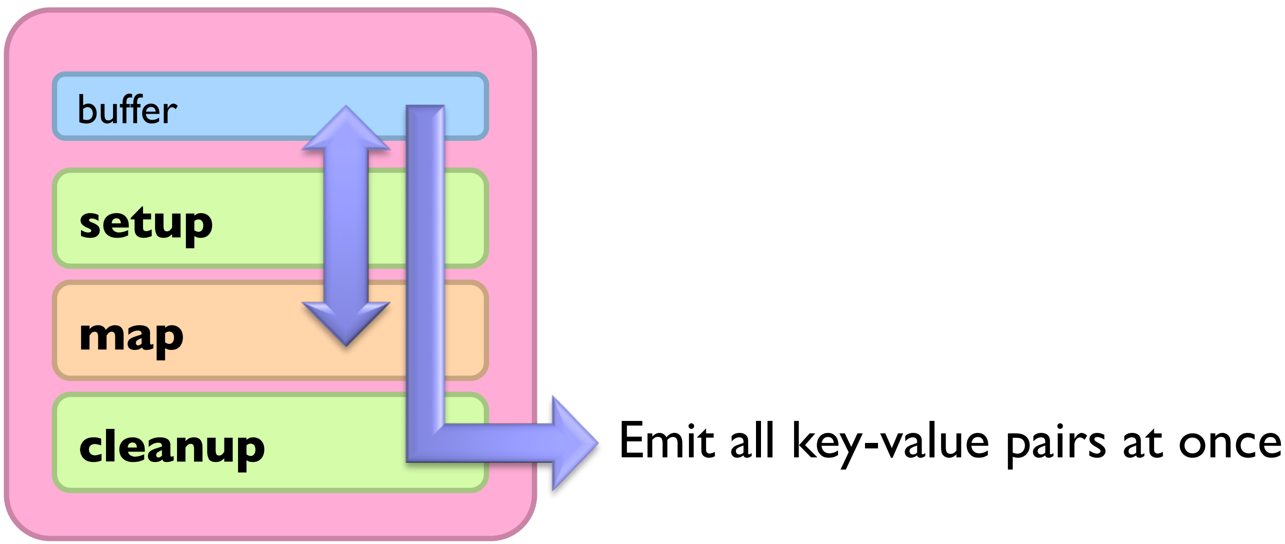
Source: http://www.flickr.com/photos/fusedforces/4324320625/

MapReduce sucks at iterative algorithms

- Hadoop task startup time
- Stragglers
- Needless graph shuffling
- Checkpointing at each iteration

In-Mapper Combining

- Use combiners
  - Perform local aggregation on map output
  - Downside: intermediate data is still materialized
- Better: in-mapper combining
  - Preserve state across multiple map calls, aggregate messages in buffer, emit buffer contents at end
  - Downside: requires memory management

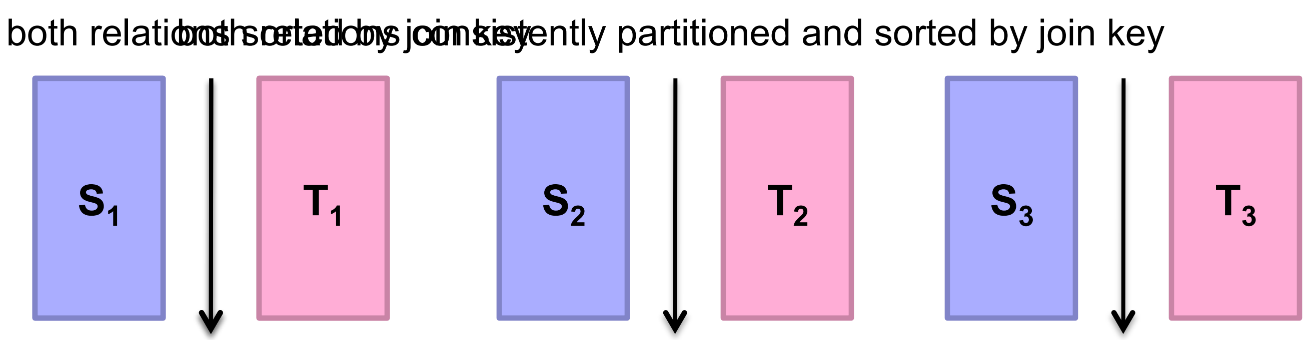


Better Partitioning

- Default: hash partitioning
  - Randomly assign nodes to partitions
- Observation: many graphs exhibit local structure
  - E.g., communities in social networks
  - Better partitioning creates more opportunities for local aggregation
- Unfortunately, partitioning is **hard**!
  - Sometimes, chick-and-egg…
  - But cheap heuristics sometimes available
  - For webgraphs: range partition on domain-sorted URLs

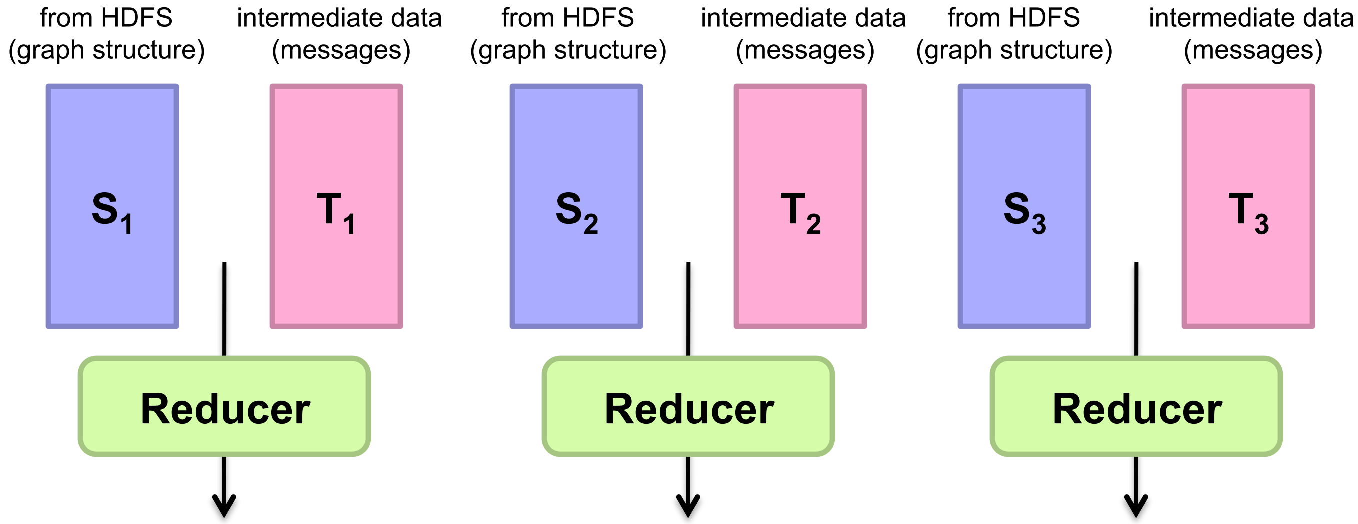
Schimmy Design Pattern

- Basic implementation contains two dataflows:
  - Messages (actual computations)
  - Graph structure (“bookkeeping”)
- Schimmy: separate the two dataflows, shuffle only the messages
  - Basic idea: merge join between graph structure and messages



Do the Schimmy!

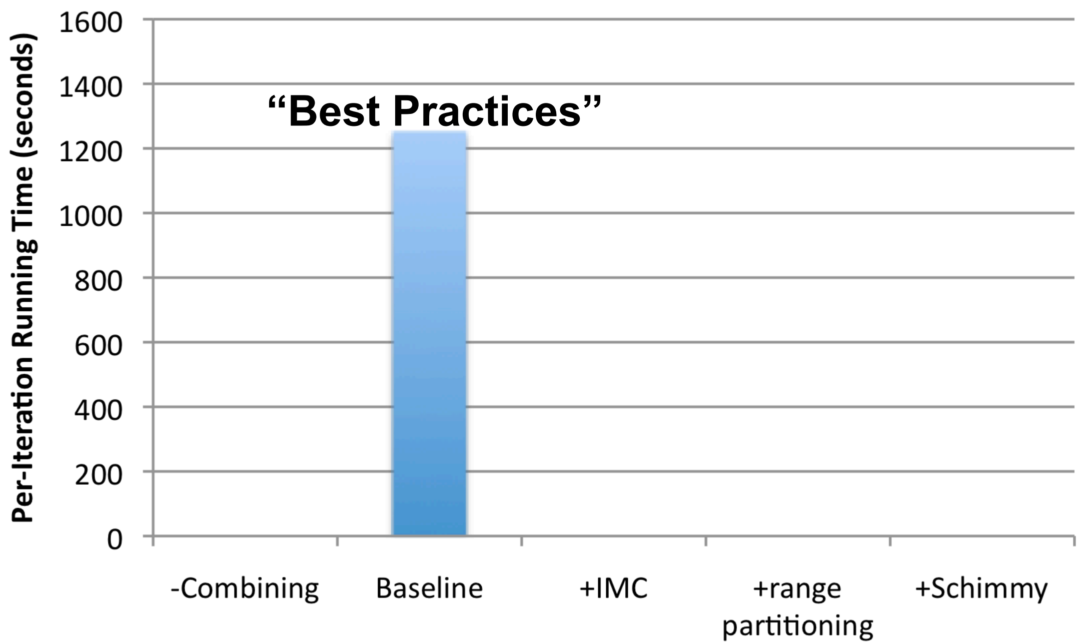
- Schimmy = reduce side parallel merge join between graph structure and messages
  - Consistent partitioning between input and intermediate data
  - Mappers emit only messages (actual computation)
  - Reducers read graph structure directly from HDFS



Experiments

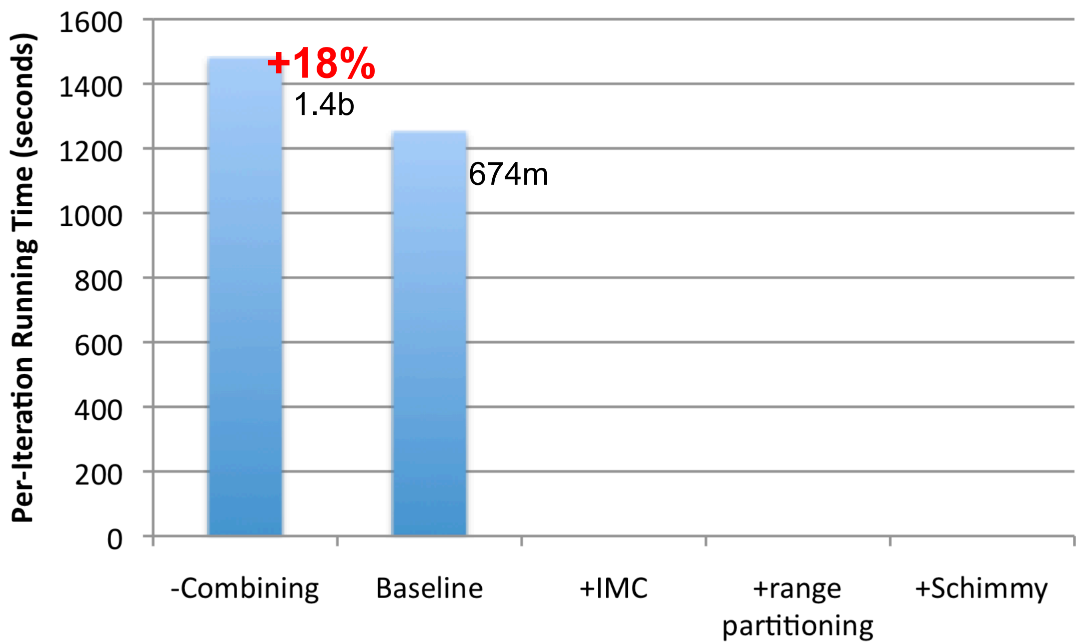
- Cluster setup:
  - 10 workers, each 2 cores (3.2 GHz Xeon), 4GB RAM, 367 GB disk
  - Hadoop 0.20.0 on RHELS 5.3
- Dataset:
  - First English segment of ClueWeb09 collection
  - 50.2m web pages (1.53 TB uncompressed, 247 GB compressed)
  - Extracted webgraph: 1.4 billion edges, 7.0 GB
  - Dataset arranged in crawl order
- Setup:
  - Measured per-iteration running time (5 iterations)
  - 100 partitions

Results



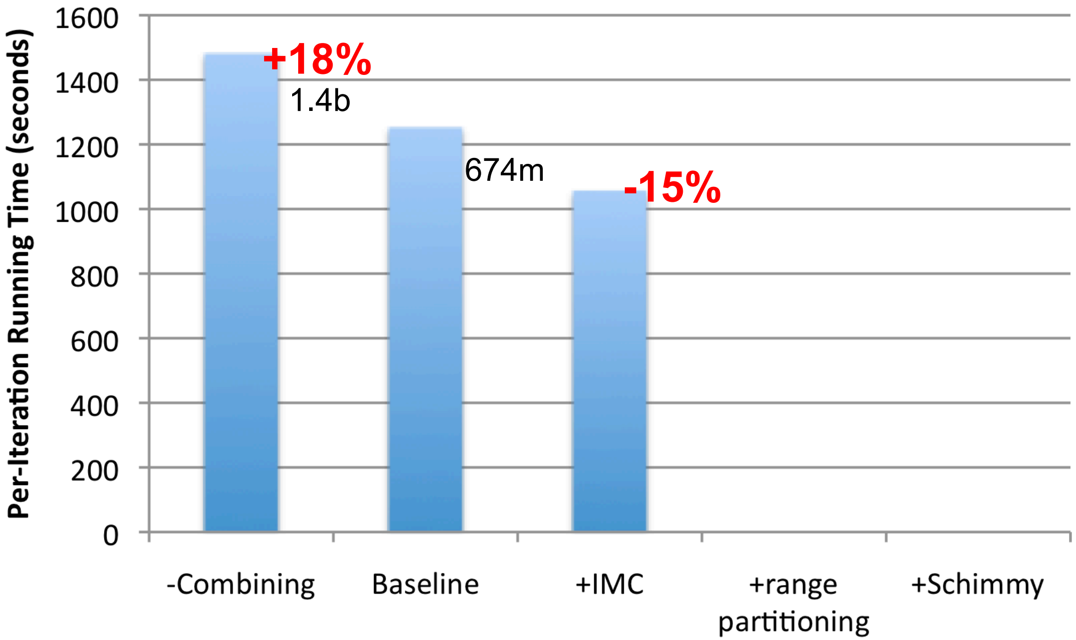
“Best Practices”
Per-Iteration Running Time (seconds)
1600
1400
1200
1000
800
600
400
200
0
-Combining
Baseline
+IMC
+range partitioning
+Schimmy

Results



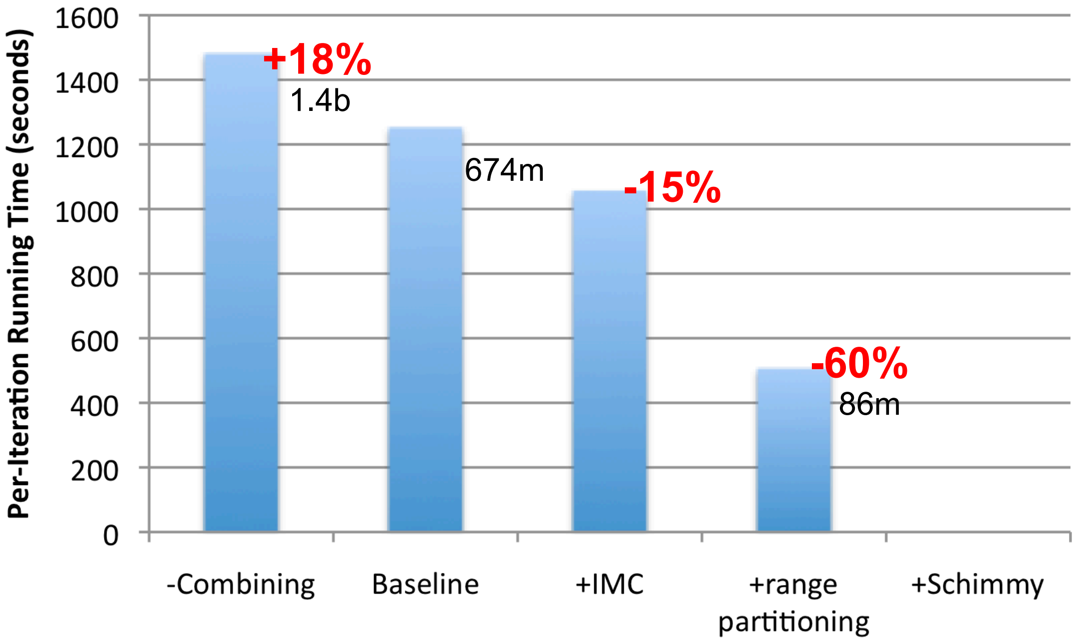
1600
1400
1200
1000
800
600
400
200
0
Per-Iteration Running Time (seconds)
+18%
1.4b
674m
-Combining
Baseline
+IMC
+range partitioning
+Schimmy

Results



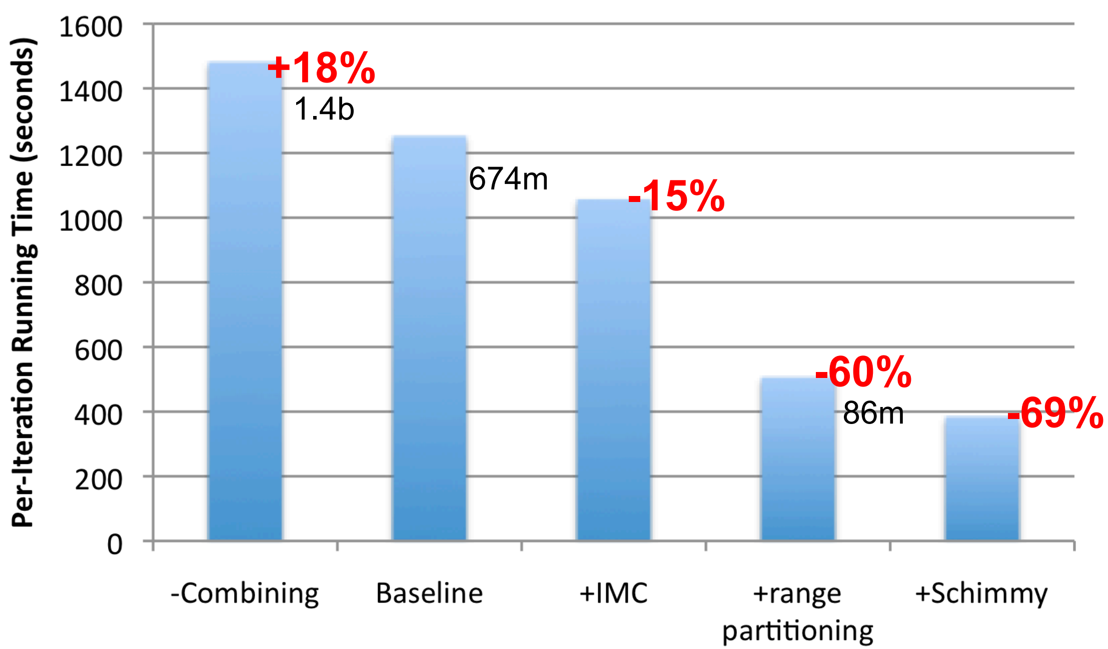
Per-Iteration Running Time (seconds)
1600
1400
1200
1000
800
600
400
200
0
+18%
1.4b
674m
-15%
-Combining
Baseline
+IMC
+range partitioning
+Schimmy

Results

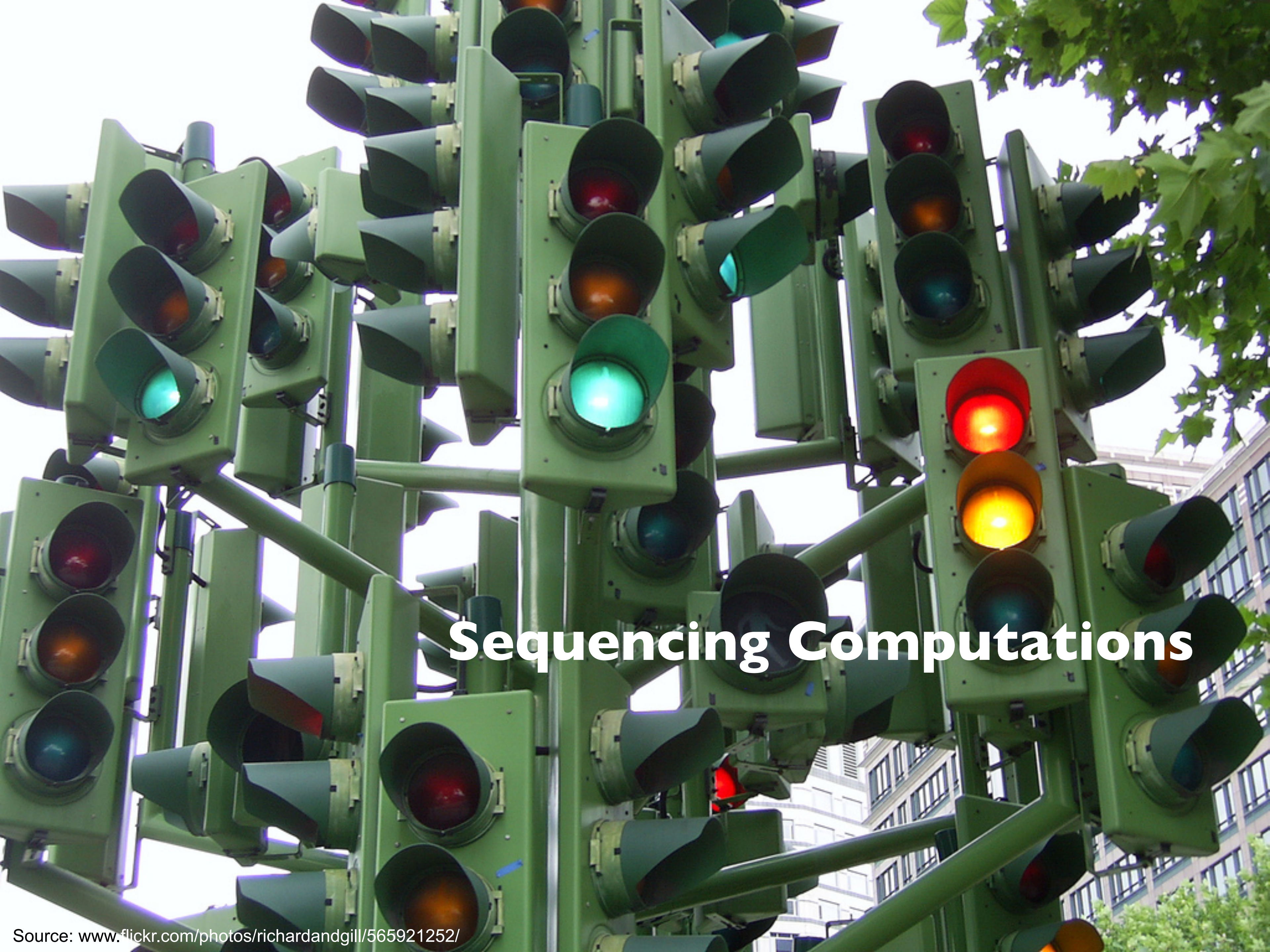


Per-Iteration Running Time (seconds)
1600
1400
1200
1000
800
600
400
200
0
+18%
1.4b
674m
-15%
-60%
86m
-Combining
Baseline
+IMC
+range partitioning
+Schimmy

Results



Per-Iteration Running Time (seconds)
1600
1400
1200
1000
800
600
400
200
0
+18%
1.4b
674m
-15%
-60%
86m
-69%
-Combining
Baseline
+IMC
+range partitioning
+Schimmy



Sequencing Computations

Source: www.flickr.com/photos/richardandgill/565921252/

Sequencing Computations

1. Turn synchronization into a sorting problem
   - Leverage the fact that keys arrive at reducers in sorted order
   - Manipulate the sort order and partitioning scheme to deliver partial results at appropriate junctures
2. Create appropriate algebraic structures to capture computation
   - Build custom data structures to accumulate partial results

Monoids!

Monoids!

- What's a monoid?
- An algebraic structure with
  - A single associative binary operation
  - An identity
- Examples:
  - Natural numbers form a commutative monoid under + with identity 0
  - Natural numbers form a commutative monoid under × with identity 1
  - Finite strings form a monoid under concatenation with identity “”
  - …

Monoids and MapReduce

- Recall averaging example: why does it work?
  - AVG is non-associative
  - Tuple of (sum, count) forms a monoid under element-wise addition
  - Destroy the monoid at end to compute average
  - Also explains the various failed algorithms
- "Stripes" pattern works in the same way!
  - Associate arrays form a monoid under element-wise addition

Go forth and monoidify!

Abstract Algebra and MapReduce

- Create appropriate algebraic structures to capture computation
- Algebraic properties
  - Associative: order doesn't matter!
  - Commutative: grouping doesn't matter!
  - Idempotent: duplicates don't matter!
  - Identity: this value doesn't matter!
  - Zero: other values don't matter!
  - …
- Different combinations lead to monoids, groups, rings, lattices, etc.

Recent thoughts, see: Jimmy Lin. Monoidify! Monoids as a Design Principle for Efficient MapReduce Algorithms. arXiv:1304.7544, April 2013.

Source: Guy Steele

Source: Google