MapReduce & Hadoop II

Mingshen Sun
The Chinese University of Hong Kong
mssun@cse.cuhk.edu.hk

Outline

- MapReduce Recap
- Design patterns
  - in-mapper combing
  - pairs and stripes
  - order inversion
  - value-to-key conversion

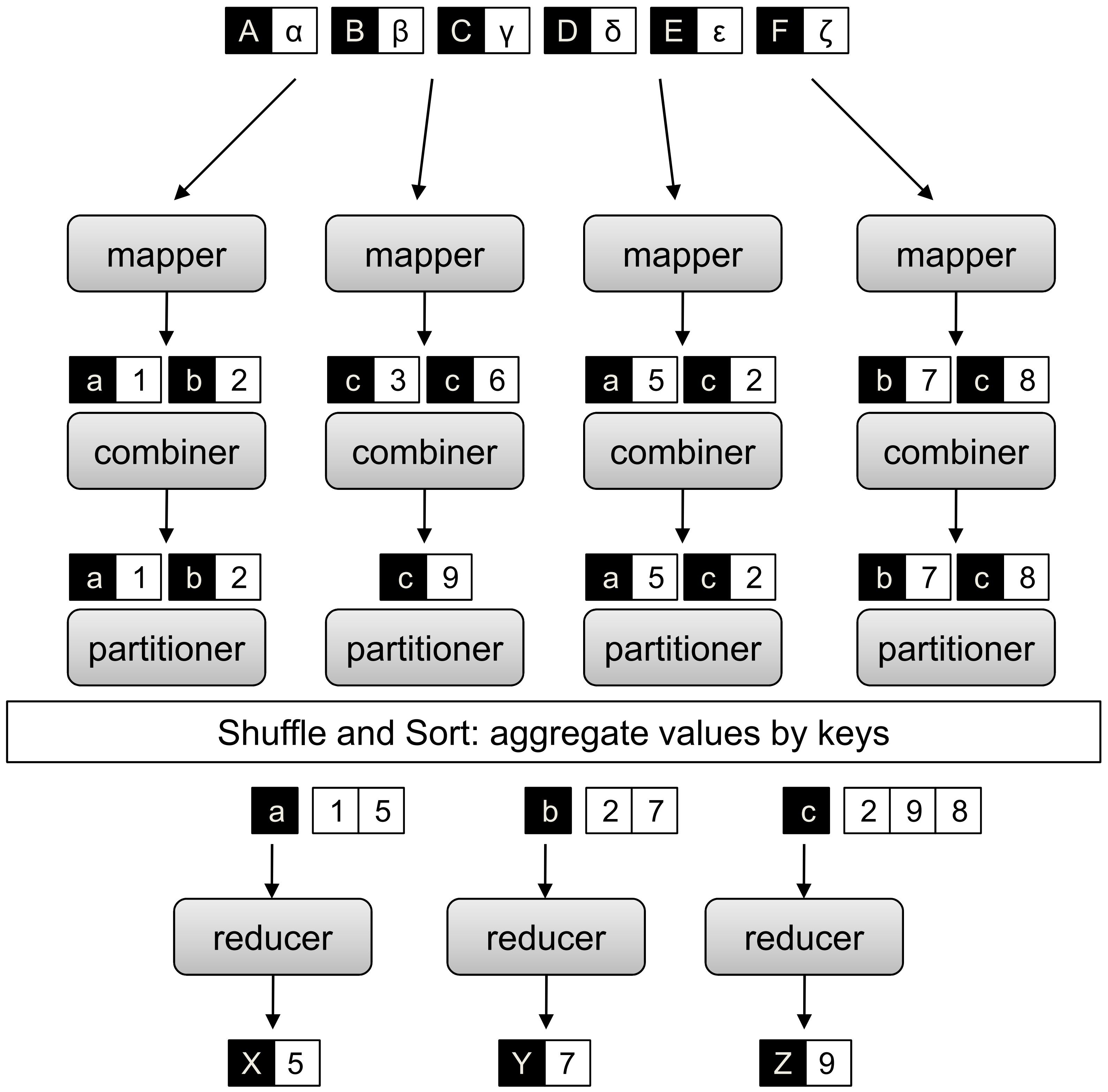
MapReduce Recap

- Input and output: each a set of key/value pairs.
- Tow functions implemented by users.
- **Map** `(k1, v1) -> list(k2, v2)`
  - takes an input key/value pair
  - produces a set of intermediate key/value pairs
- **Reduce** `(k2, list(v2)) -> list(k3, v3)`
  - takes a set of values for an intermediate key
  - produces a set of output value
  - *MapReduce framework guarantees that all values associated with the same key are brought together in the reducer*

MapReduce Recap

- Optional functions:
- **Partition** `(k’, number of partitions) -> partition for k’`
  - dividing up the intermediate key space and assigning intermediate key-value pairs to reducers
  - often a simple hash of the key, e.g., hash(k’) mod n
- **Combine** `(k2, list(v2)) -> list(k2’, v2’)`
  - mini-reducers that run in memory after the map phase
  - used as an optimization to reduce network traffic
  - will be discuss later

MapReduce Recap



Goals

- **Key question**: MapReduce provides an elegant programming model, but how should we recast a multitude of algorithms into the MapReduce model?
- **Goal of this lecture**: provide a guide to MapReduce algorithm design:
    - **design patterns**, which form the building blocks of may problems

Challenges

- MapReduce execution framework handles most complicated details
  - e.g., copy intermediate key-value pairs from mappers to reducers grouped by key during the shuffle and sort stage
- Programmers have little control over MapReduce execution:
  - Where a mapper or reducer runs
  - When a mapper or reduce begins or finishes
  - Which input key-value pairs are processed by a specific mapper
  - Which intermediate key-value pairs are processed by a specific reducer

Challenges

- Things that programmers can control:
  - Construct complex data structures as keys and values to store and communicate partial results
  - Execute user-specified initialization/termination code in a map or reduce task
  - Preserve state in both mappers and reducers across multiple input or intermediate keys
  - Control sort order of intermediate keys, and hence the order of how a reducer processes keys
  - Control partitioning of key space, and hence the set of keys encountered by a reducer

Challenges

- What we really want…
- No inherent bottlenecks as algorithms are applied to increasingly large datasets
    - linear scalability: an algorithm running on twice the amount of data should take only twice as long
    - an algorithm running on twice the number of nodes should only take half as long

Design Patterns

- Combiners and in-mapper combining
    - aggregate map outputs to reduce data traffic being shuffled from mappers to reducers
- Paris and stripes
    - keep track of joint events
- Order inversion
    - sort and control the sequence of computation
- Value-to-key conversion
    - allow secondary sorting

Local Aggregation

- In Hadoop, intermediate results (i.e., map outputs) are written to local disk before being sent over the network
  - network and disk latencies are expensive
- Local aggregation of intermediate results reduces the number of key-value pairs that need to be shuffled from the mappers to the reducers
- Default combiner:
  - provided by the MapReduce framework
  - aggregate map outputs with the same key
  - acts like a mini-reducer

Word Count: Baseline

```
1: class Mapper
2:     method Map(docid $a$, doc $d$)
3:         for all term $t \in$ doc $d$ do
4:             Emit(term $t$, count 1)
```

```
1: class Reducer
2:     method Reduce(term $t$, counts $[c_1, c_2, \ldots]$)
3:         $sum \leftarrow 0$
4:         for all count $c \in$ counts $[c_1, c_2, \ldots]$ do
5:             $sum \leftarrow sum + c$
6:         Emit(term $t$, count $sum$)
```

- What is the number of records being shuffled?
  - without combiners?
  - with combiners?

Implementation in Hadoop

```
public class WordCount {

  public static class TokenizerMapper
         extends Mapper<Object, Text, Text, IntWritable>{

      private final static IntWritable one = new IntWritable(1);
      private Text word = new Text();

      public void map(Object key, Text value, Context context
                      ) throws IOException, InterruptedException {
        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
          word.set(itr.nextToken());
          context.write(word, one);
        }
      }
    }
}
```

Implementation in Hadoop

```
public class WordCount {

  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

}
```

Implementation in Hadoop

```
public class WordCount {

public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = Job.getInstance(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }

}
```

Usage

- Environment

```
export JAVA_HOME=/usr/java/default
export PATH=$JAVA_HOME/bin:$PATH
export HADOOP_CLASSPATH=$JAVA_HOME/lib/tools.jar
```

- Compile & Package

```
$ bin/hadoop com.sun.tools.javac.Main WordCount.java
$ jar cf wc.jar WordCount*.class
```

- Run

```
$ bin/hadoop jar wc.jar WordCount /user/joe/wordcount/
input /user/joe/wordcount/output
```

Word Count: Version 1

```
1: class Mapper
2:     method Map(docid $a$, doc $d$)
3:         $H \leftarrow$ new AssociativeArray
4:         for all term $t \in$ doc $d$ do
5:             $H\{t\} \leftarrow H\{t\} + 1$
6:         for all term $t \in H$ do
7:             Emit(term $t$, count $H\{t\}$)
```

counts for entire document

- in-mapper combining
  - emits a key-value pair for each unique term per document

Word Count: Version 2

```
1: class Mapper
2:     method Initialize
3:         $H \leftarrow$ new AssociativeArray
4:     method Map(docid $a$, doc $d$)
5:         for all term $t \in$ doc $d$ do
6:             $H\{t\} \leftarrow H\{t\} + 1$
7:     method Close
8:         for all term $t \in H$ do
9:             Emit(term $t$, count $H\{t\}$)
```

Setup() in Java

counts *across* documents

Cleanup() in Java

- in-mapper combining
  - recall a map object is created for each map task
  - aggregate all data appearing in the input block processed by the map task

Combiners v.s. In-Mapper Combiners

- Advantages of in-mapper combiners:
  - Provide control over where and how local aggregation takes place. In contrast, semantics of default combiners are underspecified in MapReduce.
  - In-mapper combiners are applied inside the code. Default combiners are applied inside the map outputs (after being emitted by the map task).
- Disadvantages:
  - States are preserved within mappers -> potentially large memory overhead.
  - algorithmic behavior may depend on the order in which input key-value pairs are encountered - > potential order-dependent bugs.

Combiner Design

- Combiner and reducer must share the same signature
  - combiner is treated as mini-reducer
  - combiner input and output key-value types must match reducer input key-value type
- Remember: combiner are optional optimizations
  - with/without combiner should not affect algorithm correctness
  - may be run 0, 1, or multiple times, determined by the MapReduce execution framework
- In Java, you can specify the combiner class as:
  - `public void setCombinerClass(Class<? extends Reducer> cls)`
  - exactly the Reducer type

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 $[r_1, r_2, \ldots]$)
        $sum \leftarrow 0$
        $cnt \leftarrow 0$
        for all integer $r \in$ integers $[r_1, r_2, \ldots]$ do
            $sum \leftarrow sum + r$
            $cnt \leftarrow cnt + 1$
        $r_{avg} \leftarrow sum/cnt$
        Emit(string $t$, integer $r_{avg}$)
```

Pseudo-code for the basic MapReduce algorithm that computes the mean of values associated with the same key.

- Any drawback?
- Can we use reducer as combiner?
  - i.e., set combiner class to be reducer class

Computing the Mean: Version 1

- Mean of the means is not the original mean.
- e.g.,
  - `mean(1, 2, 3, 4, 5) != mean(mean(1, 2), mean(3, 4, 5))`
- It's not a problem for Word Count problem, but it's a problem here.

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 $[r_1, r_2, \ldots]$)
        $sum \leftarrow 0$
        $cnt \leftarrow 0$
        for all integer $r \in$ integers $[r_1, r_2, \ldots]$ do
            $sum \leftarrow sum + r$
            $cnt \leftarrow cnt + 1$
        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 $t$, integer $r_{avg}$)
```

- Does it work? Why?
    - recall that combiners must have the same input and output key-value type
    - Why?
    - combiners are optimizations that cannot change the correctness of the algorithm

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 $t$, integer $r_{avg}$)
```

- Does it work? Why?

Computing the Mean: Version 4

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

- Does it work?
- Do we need a combiner?

Pairs and Stripes

- To illustrate how constructing complex keys and values improves the performance of computation.

A New Running Example

- Problem: building a word co-occurrence matrix over a text collection
    - M = n * n matrix (n = number of unique words)
    - m[i][j] = number of times word w[i] co-occurs with word w[j] within a specific context (e.g., same sentence, same paragraph, same document)
        - it is easy to show that m[i][j] == m[j][i]
- Why this problem is interesting?
    - distributional profiles of words
    - information retrieval
    - statistical natural language processing

Challenge

- Space requirement: O(n^2).
  - too big if we simply store the whole matrix with billions of words in memory
  - a single machine typically cannot keep the whole matrix
- How to use MapReduce to implement this large counting problem?
- Our approach:
  - mappers generate partial counts
  - reducers aggregate partial counts

Pairs

- Each mapper:
  - Emits intermediate key-value pairs with each co-occurring word pair and integer 1
- Each reducer:
  - Sums up all values associated with the same co-occurring word pair
  - MapReduce execution framework guarantees that all values associated with the same key are brought together in the reducer

Pairs

```
1: class Mapper
2:     method Map(docid $a$, doc $d$)
3:         for all term $w \in$ doc $d$ do
4:             for all term $u \in$ Neighbors($w$) do
5:                 Emit(pair $(w, u)$, count 1)        ▷ Emit count for each co-occurrence
```

```
1: class Reducer
2:     method Reduce(pair $p$, counts $[c_1, c_2, \ldots]$)
3:         $s \leftarrow 0$
4:         for all count $c \in$ counts $[c_1, c_2, \ldots]$ do
5:             $s \leftarrow s + c$        ▷ Sum co-occurrence counts
6:         Emit(pair $p$, count $s$)
```

- Can we use the default combiner here?

Stripes

- Each mapper:
  - For each particular word, stores co-occurrence information in an associative array
  - Emits intermediate key-value pairs with words as keys and corresponding associative arrays as values
- Each reducer:
  - Sums all the counts in the associative arrays
  - MapReduce execution framework guarantees that all associative arrays with the same key are brought together in the reducer

Stripes

- Example:

```
(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 emits
    - `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

```
class MAPPER
    method MAP(docid $a$, doc $d$)
        for all term $w \in$ doc $d$ do
            $H \leftarrow$ new ASSOCIATIVEARRAY
            for all term $u \in$ NEIGHBORS($w$) do
                $H\{u\} \leftarrow H\{u\} + 1$                ▷ Tally words co-occurring with $w$
            EMIT(Term $w$, Stripe $H$)
```

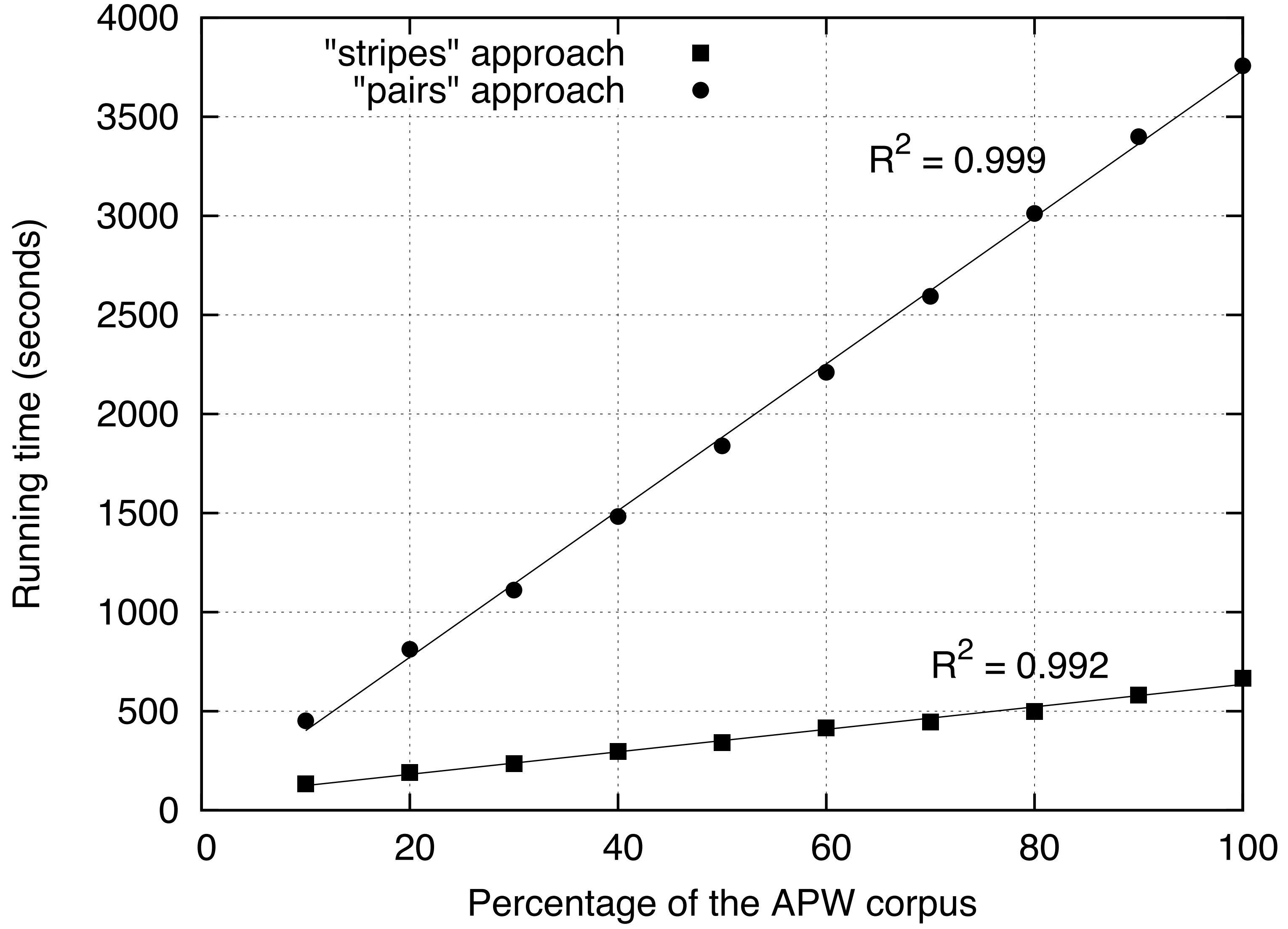
```
class REDUCER
    method REDUCE(term $w$, stripes $[H_1, H_2, H_3, \ldots]$)
        $H_f \leftarrow$ new ASSOCIATIVEARRAY
        for all stripe $H \in$ stripes $[H_1, H_2, H_3, \ldots]$ do
            SUM($H_f, H$)                                  ▷ Element-wise sum
        EMIT(term $w$, stripe $H_f$)
```

- pseudo-code of stripes approach

Pairs v.s. Stripes

- Pairs:
  - Pro: Easy to understand and implement
  - Con: Generate many key-value pairs
- Stripes:
  - Pro: Generate fewer key-value pairs
  - Pro: Make better use of combiners
  - Con: Memory size of associative arrays in mappers could be huge
- Both pairs and stripes can apply in-mapper combining

Pairs v.s. Stripes



- stripes much faster than pairs
- linearity is maintained

Relative Frequencies

- Drawback of co-occurrence counts
  - absolute counts doesn't consider that some words appear more frequently than others
  - e.g., "is" occurs very often by itself
  - doesn't imply "is good" occurs more frequently than "Hello World"
- Estimate relative frequencies instead of counts

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

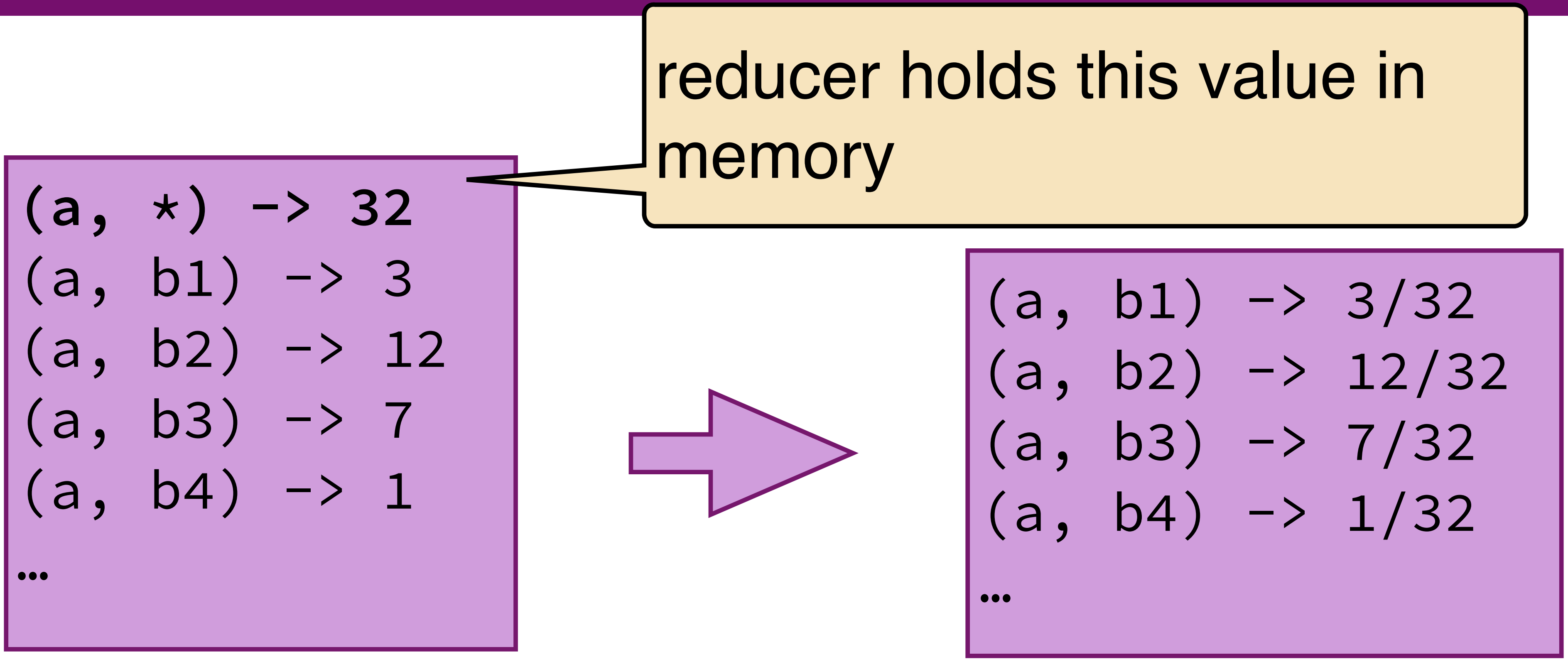
marginal

- How do we apply MapReduce to this problem?

Relative Frequencies

- Computing relative frequencies with the stripes approach is straightforward
  - Sum all the counts in the associative array for each word
  - Why is it possible in MapReduce?
  - Drawback: assuming that each associative array fits into memory
- How to compute relative frequencies with the pairs approach?

Relative Frequencies with Pairs



- Mapper emits (a, *) for every word being observed
- Mapper makes sure same word goes to the same reducer (use partitioner)
- Mapper makes suer (a, *) comes first, before individual counts (how?)
- Reducer holds state to remember the count of (a, *), until all pairs with the word "a" have been computed

Order Inversion

- Why order inversion?
  - 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
- MapReduce allows you to define the order of keys being processed by the reducer
  - shuffle and **sort**

Order Inversion: Idea

- How to use the design pattern of order inversion to compute relative frequencies via the pair approach?
    - Emit a special key-value pair for each co-occurring word for the computation of marginal
    - Control the sort order of the intermediate key so that the marginal count comes before individual counts
    - Define a custom partitioner to ensure all pairs with the same left word are shuffled to the same reducer
    - Preserve state in reducer to remember the marginal count for each word

Secondary Sorting

- MapReduce sorts input to reducers by key
  - values may be arbitrarily ordered
- What if want to sort value also?
- Scenario:
  - sensors record temperature over time
  - each sensor emits (id, time t, temperature v)

Secondary Sorting

- Naive solution
    - each sensor emits
    - id -> (t, v)
    - all readings of sensor id will be aggregated into a reducer
    - buffer values in memory for all id, then sort
- Why is this a bad idea?

Secondary Sorting

- Value-to-key conversion
    - each mapper emits
    - (id, t) -> v
    - let execution framework do the sorting
    - preserve state across multiple key-value pairs to handle processing
    - anything else?
- Main idea: sorting is offloaded from the reducer (in naive approach) to the MapReduce framework

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

Summary

- Design patterns
    - in-mapper combing
    - pairs and stripes
    - order inversion
    - value-to-key conversion

MapReduce Application

- Text retrieval
  - inverted indexing
- Data mining
  - TF-IDF
- Graph algorithm
  - parallel breadth-first search
  - parallel dijkstra’s algorithm
  - PageRank

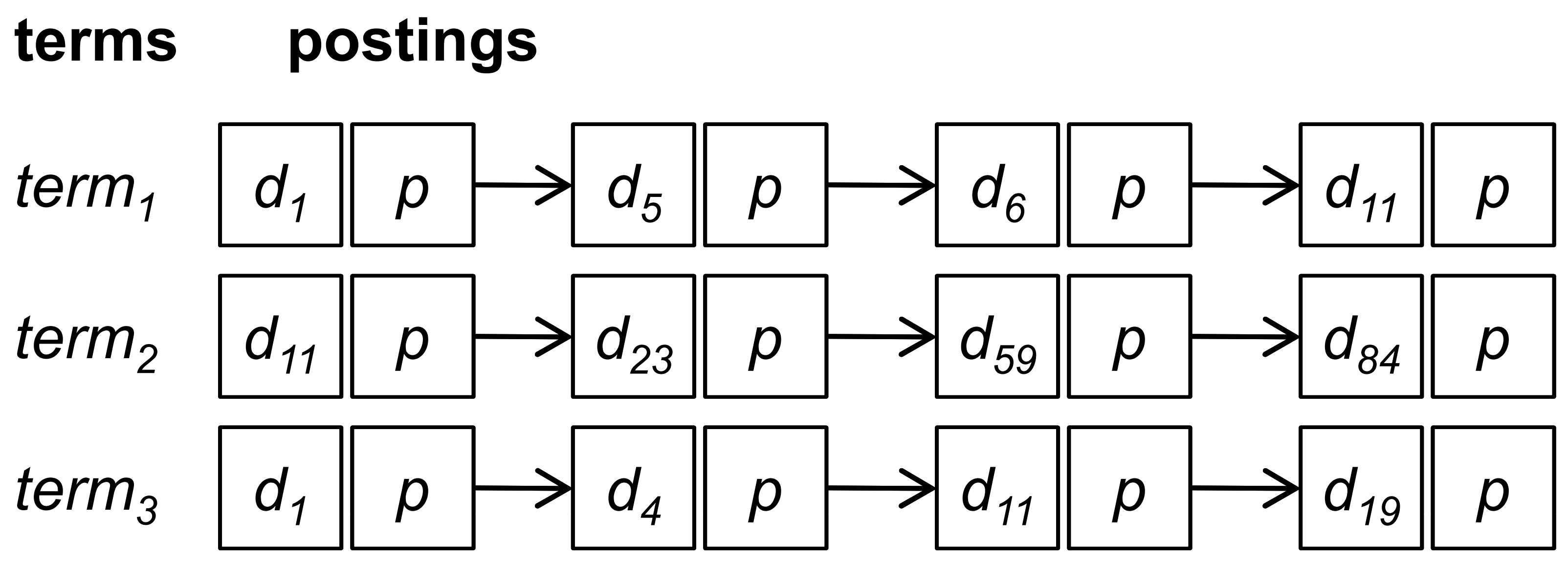
Web Search Problem

- Web search is to retrieve relevant web objects
  - e.g., web pages, PDFs, PPT slides
- Web search problem
  - crawling: gathering web content
  - indexing: constructing search indexing structure
  - retrieval: ranking documents given a query
- Challenge
  - the web is huge
  - billions of web objects, terabytes of information
- Performance goals
  - query latency needs to be small
  - scalable for a large number of documents

Inverted Indexes

- Inverted Index
  - A data structure that given a term provides access to the list of documents that contain the term
  - Used by most full-text search engines today
  - By documents, we mean web objects
- Retrieval engine uses the inverted index to score documents that contain the query terms based on some ranking model
  - e.g., based on term matches, term proximity, term attributes, etc.

Inverted Indexes



- Simple illustration of an inverted index.
  - Each term is associated with a list of postings.
  - Each posting is comprised of a document id and a payload, denoted by p in this case.
  - An inverted index provides quick access to documents ids that contain a term.

Inverted Indexes

- Given a query, retrieval involves fetching postings lists associated with query terms and traversing the postings to compute the result set.
- Simple Boolean retrieval:
  - Apply union (OR) or intersection (AND) of posting lists
- General retrieval:
  - Document scores are ranked
  - Top k documents are returned

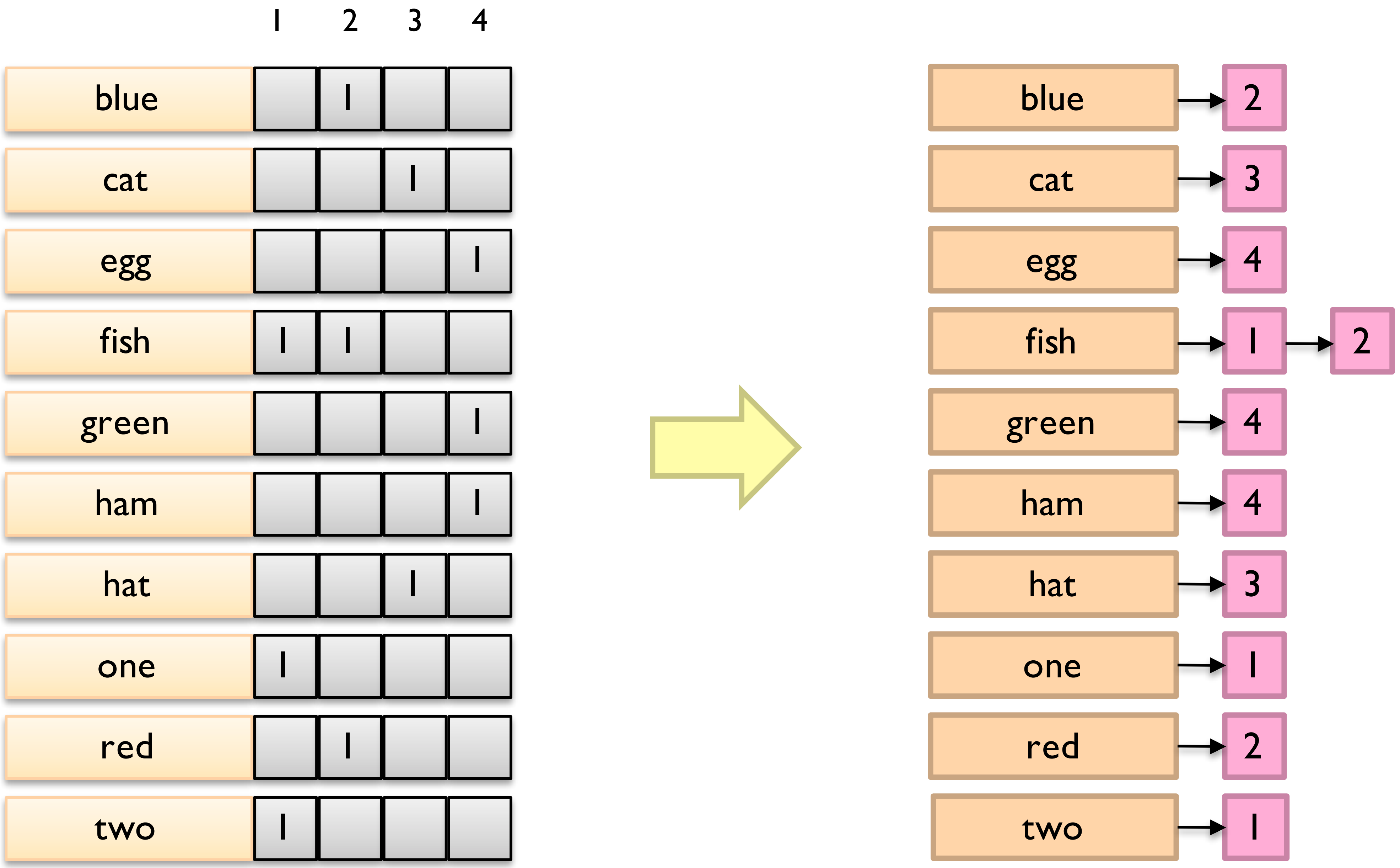
Inverted Indexes

Doc 1
one fish, two fish

Doc 2
red fish, blue fish

Doc 3
cat in the hat

Doc 4
green eggs and ham



	1	2	3	4
blue		1		
cat			1	
egg				1
fish	1	1		
green				1
ham				1
hat			1	
one	1			
red		1		
two	1			

$$\begin{tikzcd}
\text{blue} \arrow[r] & \boxed{2} \\
\text{cat} \arrow[r] & \boxed{3} \\
\text{egg} \arrow[r] & \boxed{4} \\
\text{fish} \arrow[r] & \boxed{1} \arrow[r] & \boxed{2} \\
\text{green} \arrow[r] & \boxed{4} \\
\text{ham} \arrow[r] & \boxed{4} \\
\text{hat} \arrow[r] & \boxed{3} \\
\text{one} \arrow[r] & \boxed{1} \\
\text{red} \arrow[r] & \boxed{2} \\
\text{two} \arrow[r] & \boxed{1}
\end{tikzcd}$$

Inverted Indexes: Construction

- How to construct an inverted index?
- Naive approach:
    - For each document, extract all useful terms, and exclude all stopwords (e.g., “the”, “a”, “of”) and remove affixes (e.g., “dogs” to “dog”)
    - For each term, add the posting (document, payload) to an existing list, or create a posting list if the term is new
- Clearly, naive approach is not scalable if the document collection is huge and each document is large
- Can we use MapReduce?

Baseline Implementation

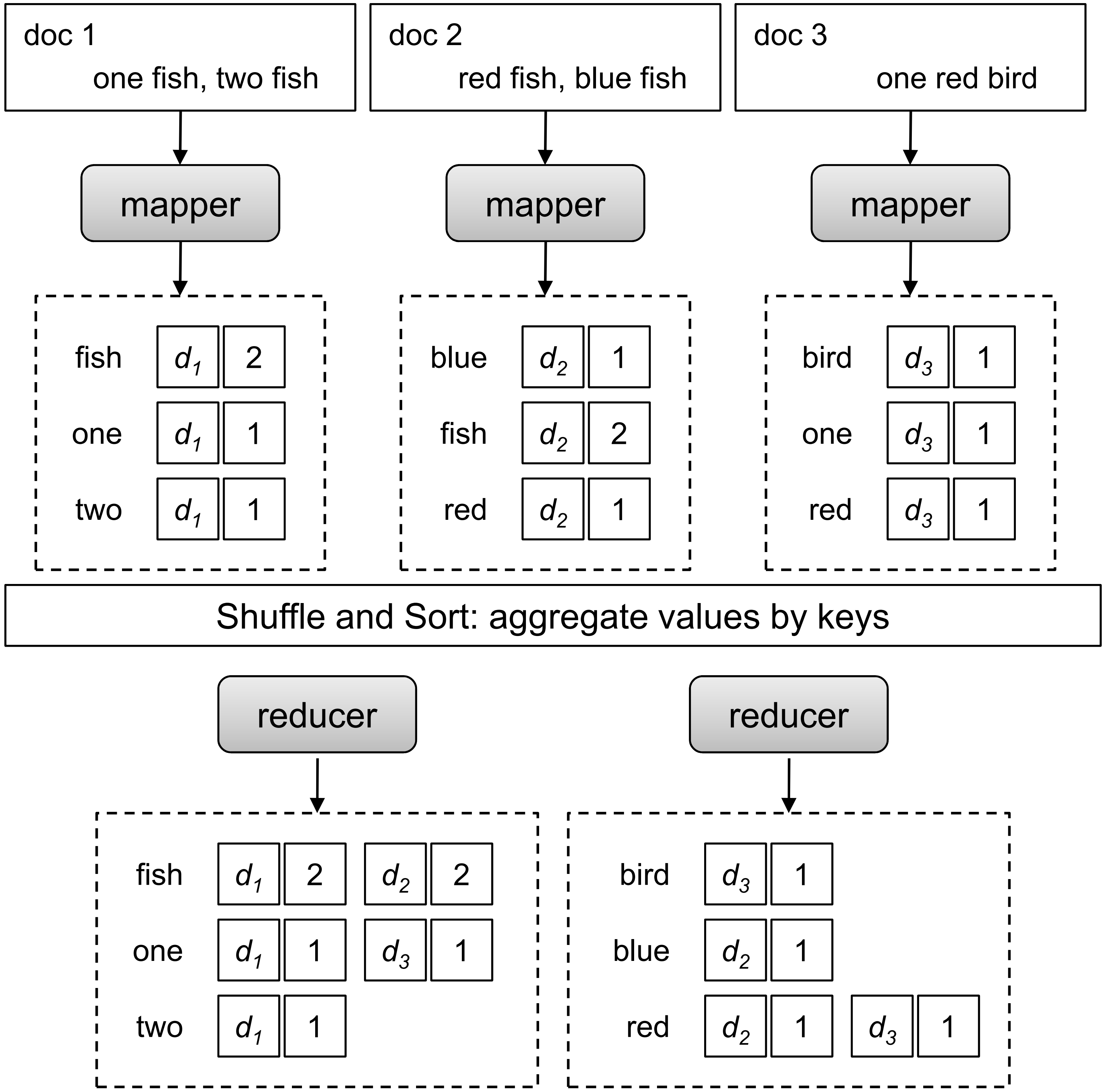
- Our goal: construct an inverted index given a document collection
- Main idea:
  - Input to each mapper:
    - Document IDs (keys)
    - Actual document content (values)
  - What each mapper does:
    - Analyze each document and extract useful terms
    - Compute term frequencies (per document)
- Emit (term, posting)
- What each reducer does
  - Aggregates all observed postings for each term
  - Construct the posting list

Baseline Implementation

```
1: class Mapper
2:     method Map(docid $n$, doc $d$)
3:         $H \leftarrow$ new AssociativeArray
4:         for all term $t \in$ doc $d$ do
5:             $H\{t\} \leftarrow H\{t\} + 1$
6:         for all term $t \in H$ do
7:             Emit(tuple $\langle t, n\rangle$, tf $H\{t\}$)

1: class Reducer
2:     method Initialize
3:         $t_{prev} \leftarrow \emptyset$
4:         $P \leftarrow$ new PostingsList
5:     method Reduce(tuple $\langle t, n\rangle$, tf $[f]$)
6:         if $t \neq t_{prev} \wedge t_{prev} \neq \emptyset$ then
7:             Emit(term $t$, postings $P$)
8:             $P$.Reset()
9:         $P$.Add($\langle n, f\rangle$)
10:        $t_{prev} \leftarrow t$
11:    method Close
12:        Emit(term $t$, postings $P$)
```

Baseline Implementation



Baseline Implementation

- In the shuffle and sort phase, MapReduce framework forms a large, distributed group by the postings of each term
- From reducer's point of view
    - Each input to the reducer is the resulting posting list of a term
    - Reducer may sort the list (if needed), and writes the final output to disk
    - The task of each reducer is greatly simplified! MapReduce framework has done most heavy liftings.

Positional Indexes

Map

Doc 1
one fish, two fish

one	1	1
two	1	1
fish	1	2

Doc 2
red fish, blue fish

red	2	1
blue	2	1
fish	2	2

Doc 3
cat in the hat

cat	3	1
hat	3	1

Shuffle and Sort: aggregate values by keys

Reduce

cat	3	1		
fish	1	2	2	2
one	1	1		
red	2	1		

blue	2	1
hat	3	1
two	1	1

Scalability Issue

- Scalability problem in baseline implementation

```
1: class Mapper
2:     procedure Map(docid $n$, doc $d$)
3:         $H \leftarrow$ new AssociativeArray
4:         for all term $t \in$ doc $d$ do
5:             $H\{t\} \leftarrow H\{t\} + 1$
6:         for all term $t \in H$ do
7:             Emit(term $t$, posting $\langle n, H\{t\}\rangle$)
```

```
1: class Reducer
2:     procedure Reduce(term $t$, postings $[\langle n_1, f_1\rangle, \langle n_2, f_2\rangle \ldots]$)
3:         $P \leftarrow$ new List
4:         for all posting $\langle a, f\rangle \in$ postings $[\langle n_1, f_1\rangle, \langle n_2, f_2\rangle \ldots]$ do
5:             Append($P$, $\langle a, f\rangle$)
6:         Sort($P$)
7:         Emit(term $t$, postings $P$)
```

Any problem?

Scalability Issue

- Assumption of baseline implementation:
  - Reducer has sufficient memory to hold all postings associated with the same term
- Why?
  - The MapReduce framework makes no guarantees about the ordering of values associated with the same key.
  - The reducer first buffers all postings (line 5) and then performs an in-memory sort before writing the postings to disk

Scalability Issue

- How to solve? Key idea is to let MapReduce framework do sorting for us
- Instead of emitting
    - `(term t, posting <docid, f>)`
- Emit
    - **`(tuple <t, docid>, f)`**
- Value-to-key conversion!!

Revised Implementation

- With value-to-key conversion, the MapReduce framework ensures the postings arrive in sorted order (based on <term t, docid>)
- Results can be written to disk directly
- Caution: you need a customized partitioner to ensure that all tuples with the same term are shuffled to the same reducer

Revised Implementation

```
class Mapper
    method Map(docid $n$, doc $d$)
        $H \leftarrow$ new AssociativeArray
        for all term $t \in$ doc $d$ do
            $H\{t\} \leftarrow H\{t\} + 1$
        for all term $t \in H$ do
            Emit(tuple $\langle t, n \rangle$, tf $H\{t\}$)
```

```
class Reducer
    method Initialize
        $t_{prev} \leftarrow \emptyset$
        $P \leftarrow$ new PostingsList
    method Reduce(tuple $\langle t, n \rangle$, tf $[f]$)
        if $t \neq t_{prev} \wedge t_{prev} \neq \emptyset$ then
            Emit(term $t$, postings $P$)
            $P$.Reset()
        $P$.Add($\langle n, f \rangle$)
        $t_{prev} \leftarrow t$
    method Close
        Emit(term $t$, postings $P$)
```

results are directly written to disk

TF-IDF

- Term Frequency – Inverse Document Frequency (TF-IDF)
  - Answers the question “How important is this term in a document”
- Known as a term weighting function
  - Assigns a score (weight) to each term (word) in a document
- Very commonly used in text processing and search
- Has many applications in data mining

TF-IDF Motivation

- Merely counting the number of occurrences of a word in a document is not a good enough measure of its relevance
  - If the word appears in many other documents, it is probably less relevance
  - Some words appear too frequently in all documents to be relevant
  - Known as ‘stopwords’
- TF-IDF considers both the frequency of a word in a given document and the number of documents which contain the word

TF-IDF: Definition

- Term Frequency (TF)
  - Number of times a term appears in a
- document (i.e., the count)
- Inverse Document Frequency (IDF)

$$idf = \log\left(\frac{N}{n}\right)$$

- N: total number of documents
- n: number of documents that contain a term
- TF-IDF
  - TF × IDF

Computing TF-IDF With MapReduce

- Overview of algorithm: 3 MapReduce jobs
- Job 1: compute term frequencies
- Job 2: compute number of documents each word occurs in
- Job 3: compute TD-IDF

Graph: Real-World 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
- PageRank

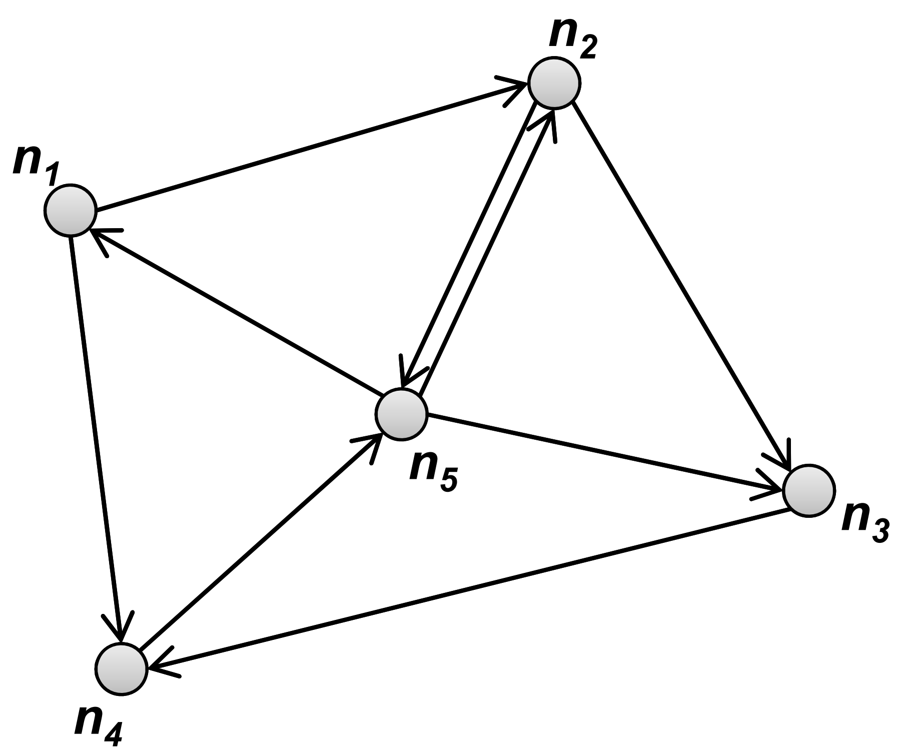
Graphs and MapReduce

- Graph algorithms typically involve:
  - Performing computations at each node: based on node features, edge features, and local link structure
  - Propagating computations: “traversing” the graph
- Challenge:
  - Algorithms running on a single machine and putting the entire graph in memory are not scalable
- Key questions:
  - How do you represent graph data in MapReduce?
  - How do you traverse a graph in MapReduce?

Graph Representations

- Two common representations
  - adjacency matrix
  - adjacency list

- easy to manipulate with linear algebra
- easy algorithmic implementation
- large memory space, esp. for sparse graph



$$\begin{array}{c|ccccc} & n_1 & n_2 & n_3 & n_4 & n_5 \\ \hline n_1 & 0 & 1 & 0 & 1 & 0 \\ n_2 & 0 & 0 & 1 & 0 & 1 \\ n_3 & 0 & 0 & 0 & 1 & 0 \\ n_4 & 0 & 0 & 0 & 0 & 1 \\ n_5 & 1 & 1 & 1 & 0 & 0 \end{array}$$

adjacency matrix

n_1	$[n_2, n_4]$
n_2	$[n_3, n_5]$
n_3	$[n_4]$
n_4	$[n_5]$
n_5	$[n_1, n_2, n_3]$

adjacency lists

- How ever, the shuffle and sort mechanism in MapReduce provides an easy way to group edges by destination nodes.

- much more compact representation
- easy to compute over out-links
- much more difficult to compute over in-links

Single-Source Shortest Path

- Problem: find shortest paths from a source node to all other nodes in the graph
  - Shortest mean smallest hop counts or lowest weights
- Algorithm:
  - Breadth-first-search: for finding minimum hop counts
  - Dijkstra’s algorithm: for finding minimum-cost paths for general graphs

Dijkstra's Algorithm

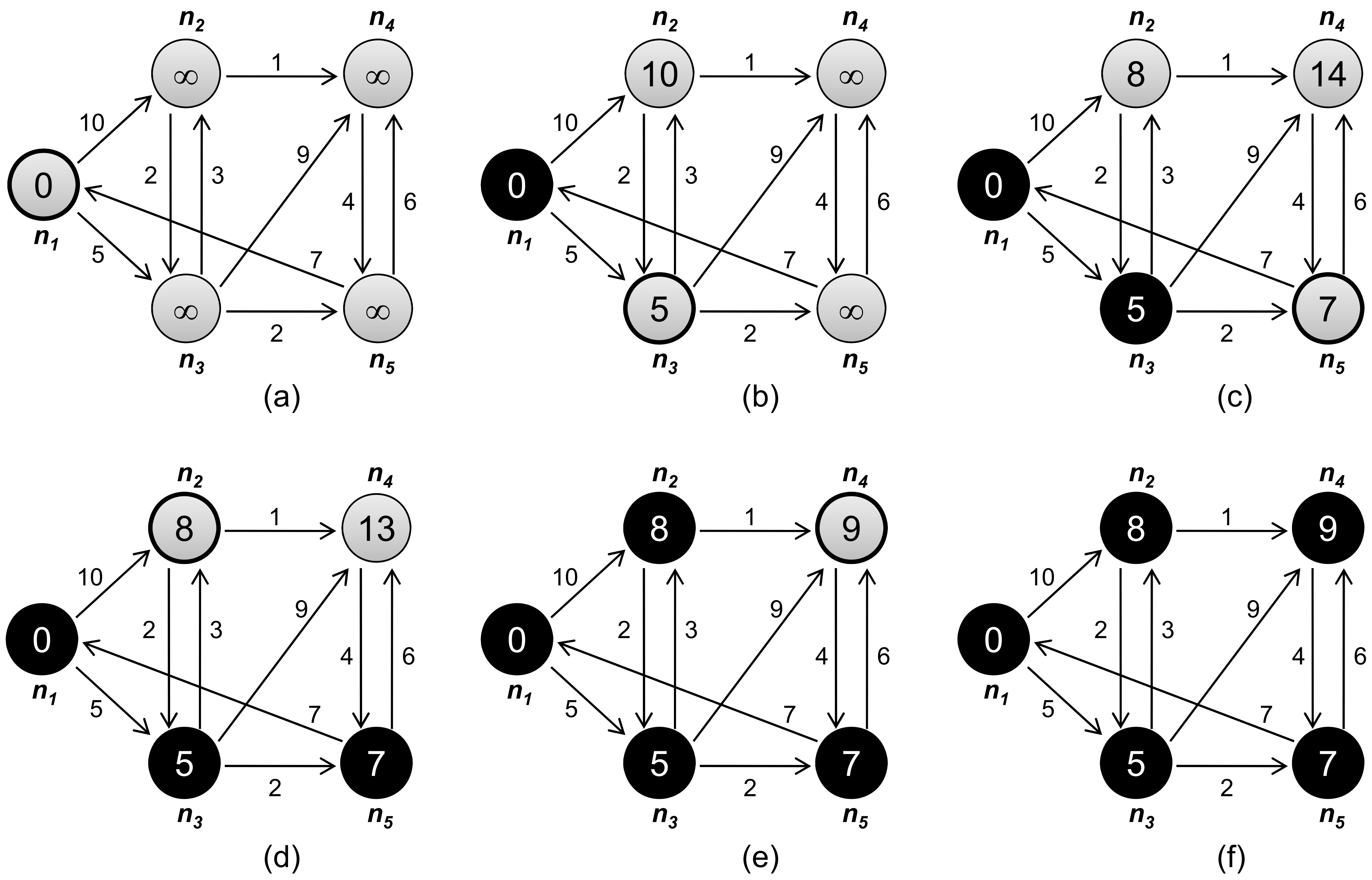


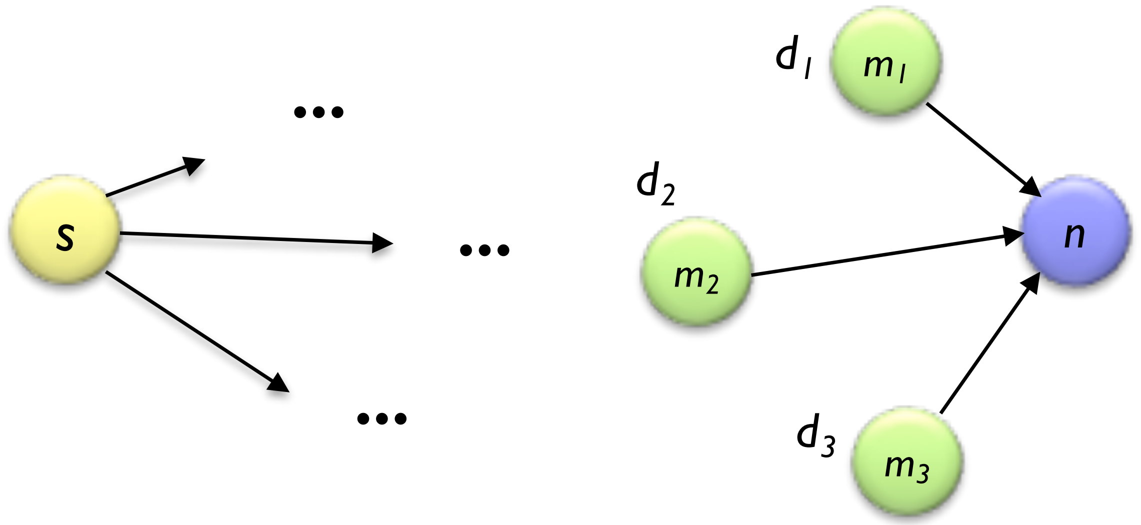
Figure 5.3: Example of Dijkstra's algorithm applied to a simple graph with five nodes, with n_1 as the source and edge distances as indicated. Parts (a)–(e) show the running of the algorithm at each iteration, with the current distance inside the node. Nodes with thicker borders are those being expanded; nodes that have already been expanded are shown in black.

Dijkstra's Algorithm

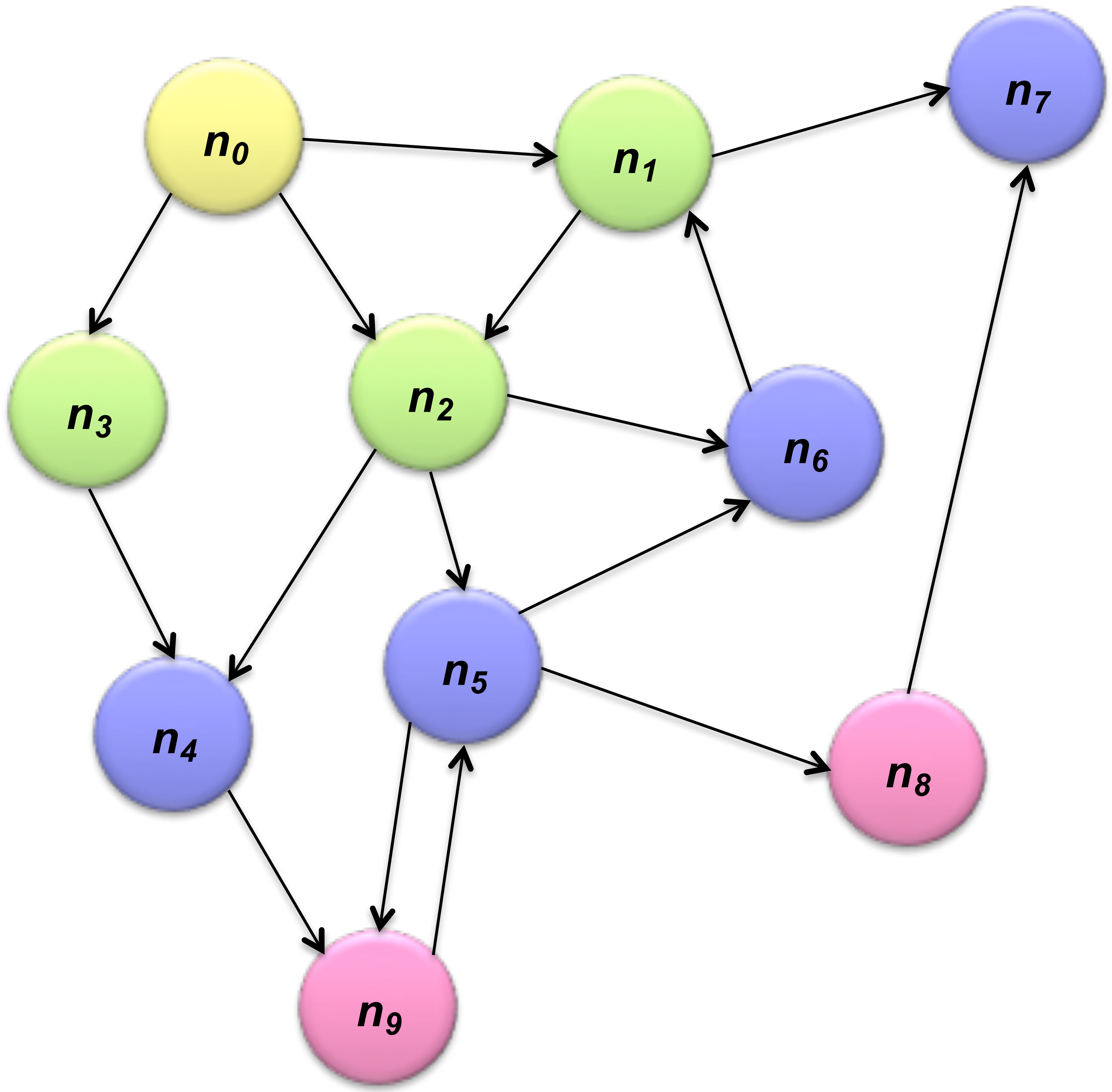
- Dijkstra's algorithm is designed as a sequential algorithm
- Key to Dijkstra's algorithm
  - Priority queue that maintains a globally sorted list of nodes by current distance
  - Not possible in MapReduce, which doesn't provide a mechanism for exchanging global data
- Solution:
  - Brute-force approach: parallel breadth first search
    - Brute force: Try to revisit many nodes that have been visited

Parallel BFS

- 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)



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 = infinity
- Mapper:
    - exit 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 $N$)
        $d \leftarrow N$.Distance
        Emit(nid $n, N$)                                   ▷ Pass along graph structure
        for all nodeid $m \in N$.AdjacencyList do
            Emit(nid $m, d+1$)                        ▷ Emit distances to reachable nodes

class Reducer
    method Reduce(nid $m$, $[d_1, d_2, \ldots]$)
        $d_{min} \leftarrow \infty$
        $M \leftarrow \emptyset$
        for all $d \in$ counts $[d_1, d_2, \ldots]$ do
            if IsNode($d$) then
                $M \leftarrow d$                                ▷ Recover graph structure
            else if $d < d_{min}$ then                      ▷ Look for shorter distance
                $d_{min} \leftarrow d$
        $M$.Distance $\leftarrow d_{min}$                       ▷ Update shortest distance
        Emit(nid $m$, node $M$)
```

Stopping Criterion

- How many iterations are needed in parallel BFS (equal edge weight case)?
- Convince yourself: when a node is first “discovered”, we’ve found the shortest path
- In practice, we iterate the algorithm until all node distances are found (i.e., no more infinity)
- How?
    - Maintain a counter inside the MapReduce program (i.e., count how many node distances are found)
    - Require a non-MapReduce driver program to submit a MapReduce job to iterate the algorithm
    - The driver program checks the counter value before submitting another job

Extend to General Weights

- Difference?
- How many iterations are needed in parallel BFS?
- How do we know that all shortest path distances are found?

Other Graph Algorithms

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

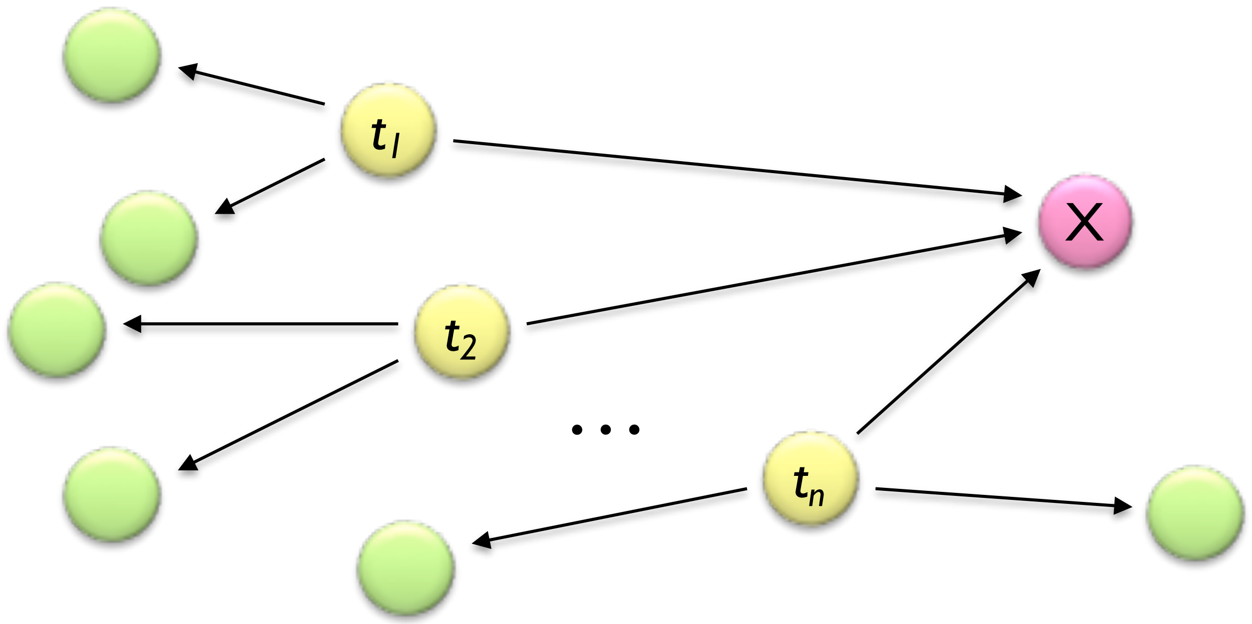
Random Walks Over the Web

- Random surfer model:
  - User starts at a random Web page
  - User randomly clicks on links, surfing from page to page
- PageRank
  - Characterizes the amount of time spent on any given page
  - Mathematically, a probability distribution over pages
- PageRank captures notions of page importance
  - Correspondence to human intuition?
  - One of thousands of features used in web search (query-independent)

PageRank: Definition

- Given page x with inlinks t1…tn, 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

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



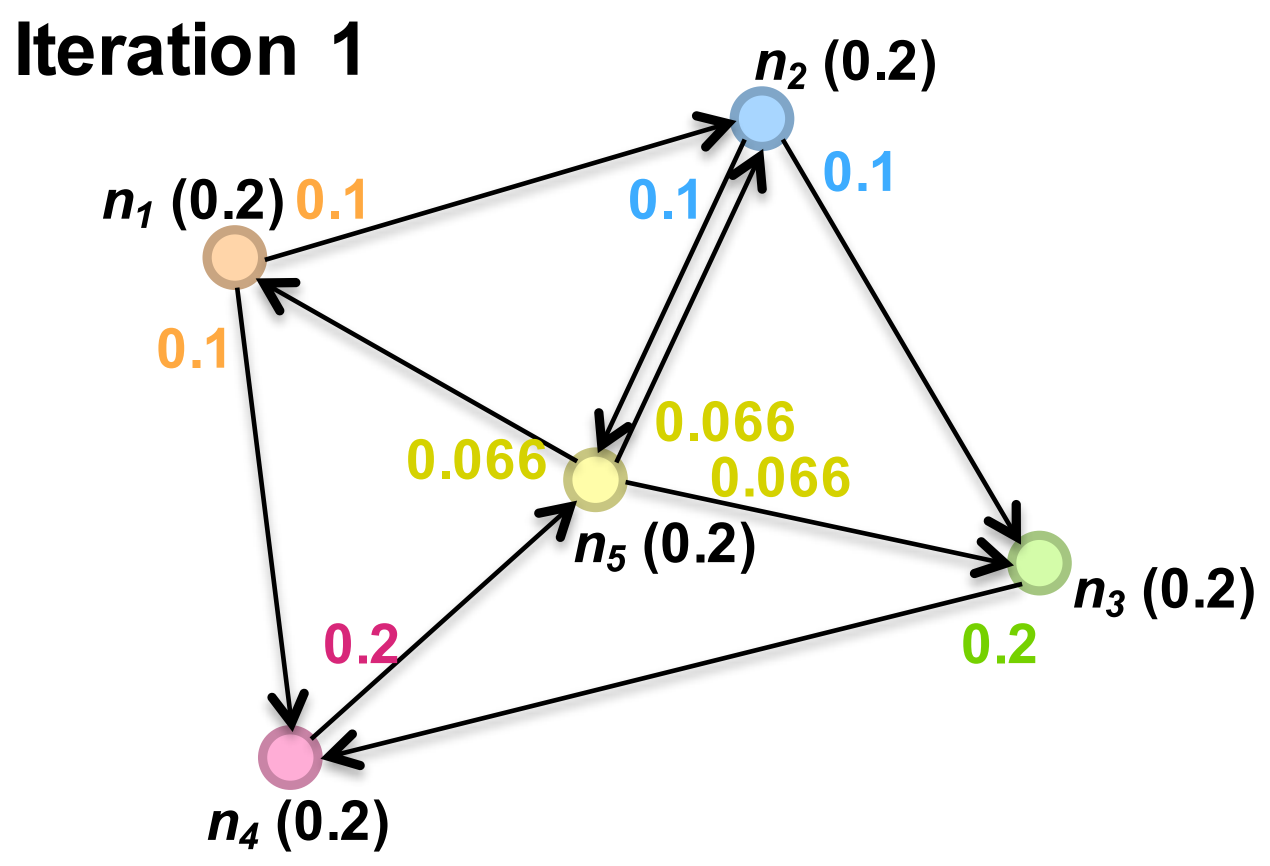
Computing PageRank

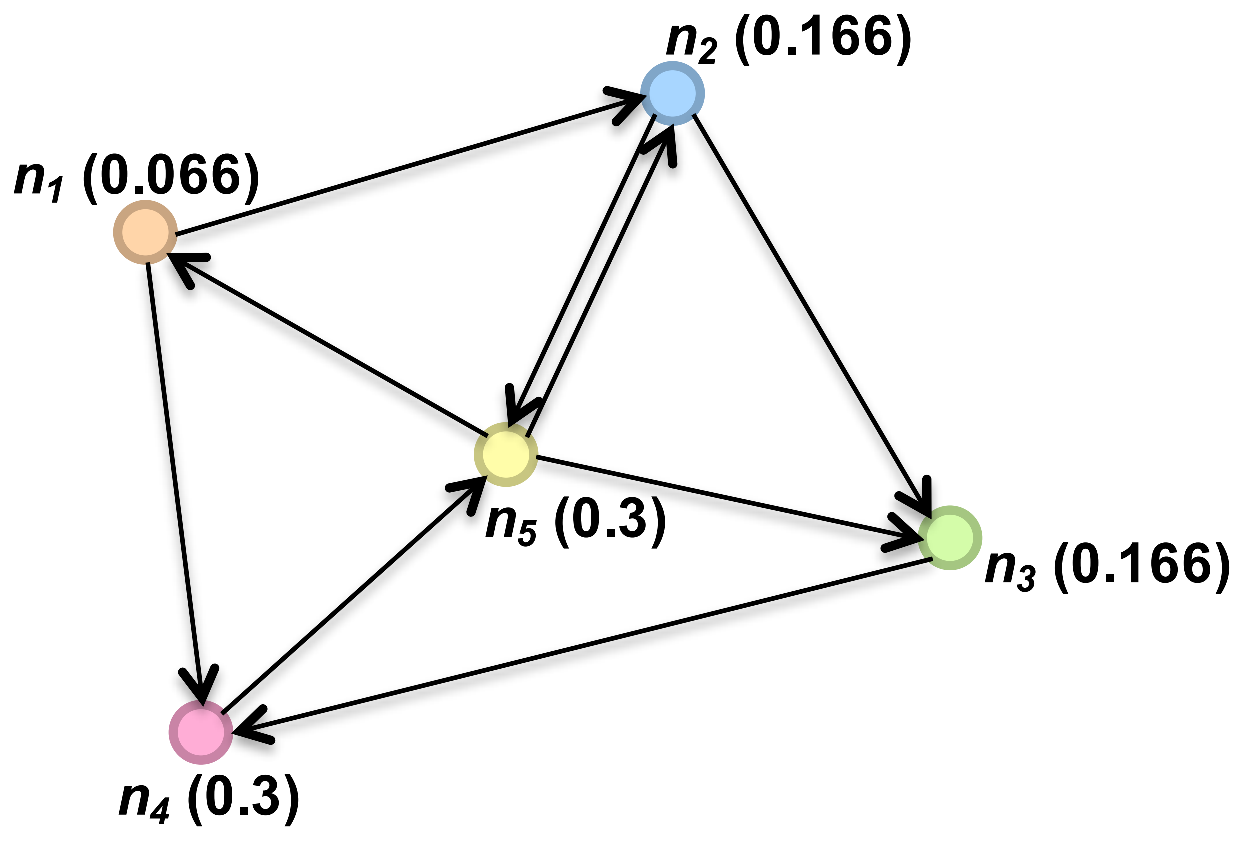
- Properties of PageRank
  - Can be computed iteratively
  - Effects at each iteration are local
- Sketch of algorithm:
  - Start with seed PRi values
  - Each page distributes PRi “credit” to all pages it links to
  - Each target page adds up “credit” from multiple in-bound links to compute PRi+1
 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

Simplified PageRank

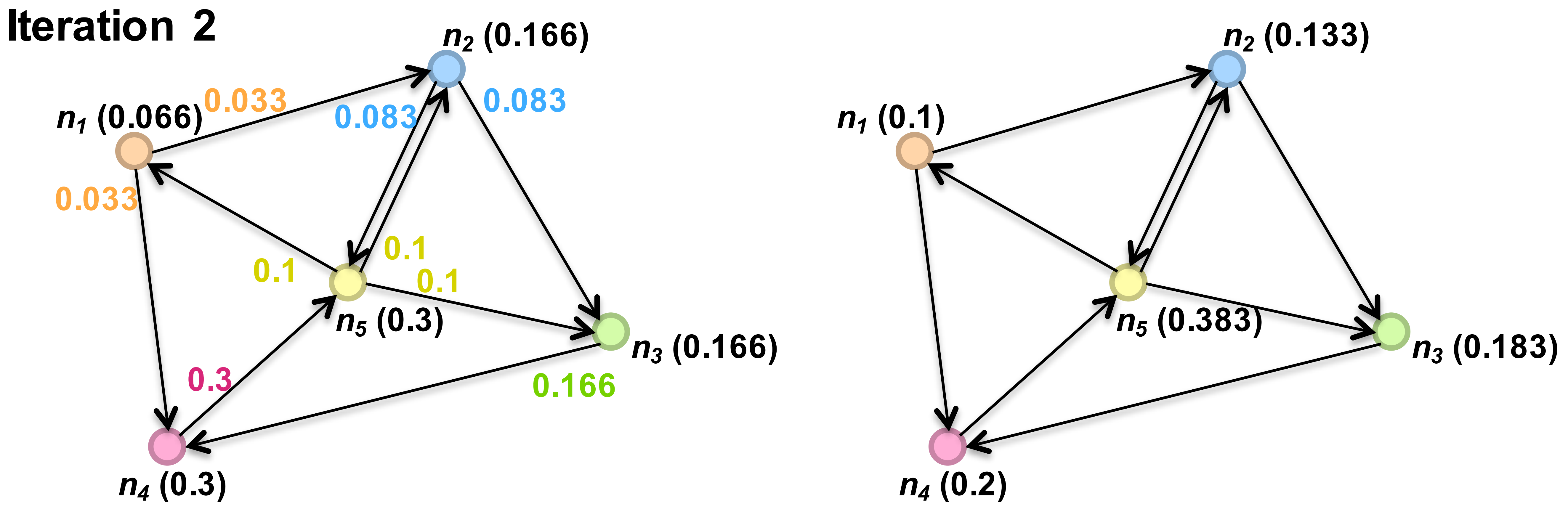
- First, tackle the simple case:
  - No random jump factor
  - No dangling nodes
- Then, factor in these complexities…
  - Why do we need the random jump?
  - Where do dangling nodes come from?

Sample PageRank Iteration (1)

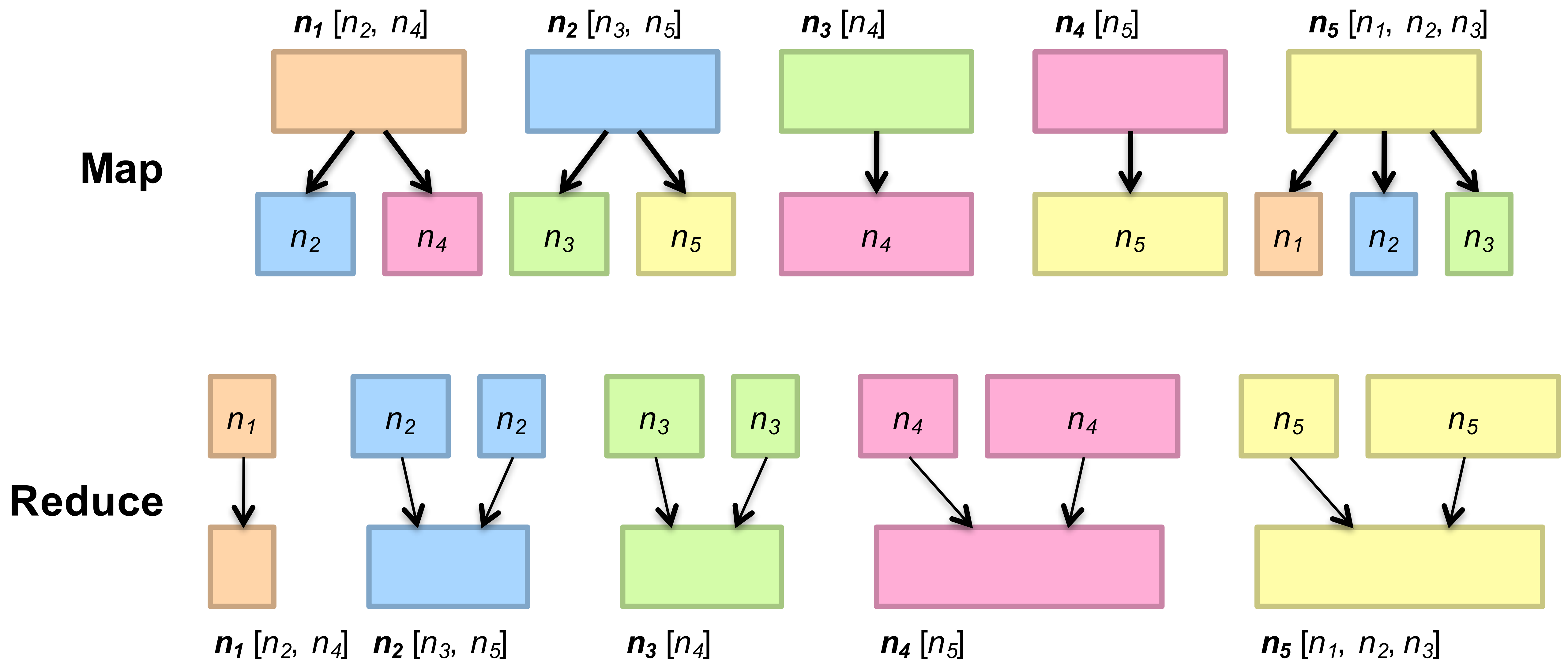




Sample PageRank Iteration (2)



PageRank in MapReduce



PageRank Pseudo-code

```
1: class Mapper
2:     method Map(nid $n$, node $N$)
3:         $p \leftarrow N.\textsc{PageRank}/|N.\textsc{AdjacencyList}|$
4:         Emit(nid $n, N$)                                ▷ Pass along graph structure
5:         for all nodeid $m \in N.\textsc{AdjacencyList}$ do
6:             Emit(nid $m, p$)                            ▷ Pass PageRank mass to neighbors

1: class Reducer
2:     method Reduce(nid $m, [p_1, p_2, \ldots]$)
3:         $M \leftarrow \emptyset$
4:         for all $p \in$ counts $[p_1, p_2, \ldots]$ do
5:             if IsNode($p$) then
6:                 $M \leftarrow p$                        ▷ Recover graph structure
7:             else
8:                 $s \leftarrow s + p$                    ▷ Sums incoming PageRank contributions
9:         $M.\textsc{PageRank} \leftarrow s$
10:        Emit(nid $m$, node $M$)
```

PageRank in MapReduce

- Map phase:
    - For each node, computes how much PageRank mass is emitted as value
- Shuffle and sort phase:
    - Group values passed along the graph edges by destination nodes
- Reduce phase:
    - PageRank mass contributions from all incoming edges are summed to arrive at the updated PageRank value for each node

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
  - …

References

- http://blog.raremile.com/hadoop-demystified/
- Hadoop illuminated: http://hadoopilluminated.com/
- Hadoop: The Definitive Guide, 3rd Edition
- Data-Intensive Text Processing with MapReduce
- CSCI4180: Cloud Computing
- Data-Intensive Computing with MapReduce: http://lintool.github.io/MapReduce-course-2013s