Today’s Agenda

- MapReduce algorithm design
  - How do you express everything in terms of m, r, c, p?
  - Toward “design patterns”

- Indexing / Retrieval
  - Basics of indexing and retrieval
  - Inverted indexing in MapReduce

- Graph Algorithms:
  - Graph problems and representations
  - Parallel breadth-first search
  - PageRank
MapReduce Algorithm Design
MapReduce: Recap

- Programmers must specify:
  - map \( (k, v) \rightarrow <k', v'> \)*
  - reduce \( (k', v') \rightarrow <k', v'> \)*

  - All values with the same key are reduced together

- Optionally, also:
  - partition \( (k', \text{number of partitions}) \rightarrow \text{partition for } k' \)

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

  - combine \( (k', v') \rightarrow <k', v'> \)*

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

- The execution framework handles everything else…
Shuffle and Sort: aggregate values by keys

- Combine:
  - Combine 1, 2
  - Combine 3, 6
  - Combine 5, 2
  - Combine 7, 8

- Partition:
  - Partition
  - Partition
  - Partition

- Reduce:
  - Reduce
  - Reduce
  - Reduce

- Output:
  - s1
  - s2
  - 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

Mapper object
- configure
- map
- close

Reducer object
- configure
- reduce
- close

- state

- one object per task
- one call per input key-value pair
- one call per intermediate key
- API initialization hook
- API cleanup hook
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

Mapper
  ↓
circular buffer (in memory)
  ↓
spills (on disk)
  ↓
Combiner

Combiner
  ↓
merged spills (on disk)
  ↓
intermediate files (on disk)
  ↓
Reducer
  ↓
other reducers

other mappers
Word Count: Baseline

1: class Mapper
2:   method MAP(docid a, doc d)
3:     for all term t ∈ doc d do
4:       EMIT(term t, count 1)

1: class Reducer
2:   method REDUCE(term t, counts [c₁, c₂, ...])
3:     sum ← 0
4:     for all count c ∈ counts [c₁, c₂, ...] do
5:       sum ← sum + c
6:     EMIT(term t, count s)

What’s the impact of combiners?
class Mapper

method MAP(docid a, doc d)

H ← new ASSOCIATIVE ARRAY

for all term t ∈ doc d do

H{t} ← H{t} + 1  \hspace{1cm} \triangleright \text{Tally counts for entire document}

for all term t ∈ H do

EMIT(term t, count H{t})
1: class Mapper
2:    method Initialize
3:        \( H \leftarrow \text{new AssociativeArray} \)
4:    method Map(docid a, doc d)
5:        for all term \( t \in \text{doc} \ d \) do
6:            \( H\{t\} \leftarrow H\{t\} + 1 \) \hfill \triangleright \text{Tally counts across documents}
7:    method Close
8:        for all term \( t \in H \) do
9:            Emit(term \( t \), count \( H\{t\} \) )

Are combiners still needed?

Key: preserve state across input key-value pairs!
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

1: class Mapper
2:     method Map(string t, integer r)
3:         Emit(string t, integer r)

1: class Reducer
2:     method Reduce(string t, integers [r₁, r₂, ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all integer r ∈ integers [r₁, r₂, ...] do
6:             sum ← sum + r
7:             cnt ← cnt + 1
8:         r_avg ← sum/cnt
9:         Emit(string t, integer r_avg)

Why can’t we use reducer as combiner?
Computing the Mean: Version 2

1: class Mapper
2:   method MAP(string t, integer r)
3:       Emit(string t, integer r)

1: class Combiner
2:   method COMBINE(string t, integers [r₁, r₂, ...])
3:       sum ← 0
4:       cnt ← 0
5:       for all integer r ∈ integers [r₁, r₂, ...] do
6:         sum ← sum + r
7:         cnt ← cnt + 1
8:       Emit(string t, pair (sum, cnt)) ▷ Separate sum and count

1: class Reducer
2:   method REDUCE(string t, pairs [(s₁, c₁), (s₂, c₂) ...])
3:       sum ← 0
4:       cnt ← 0
5:       for all pair (s, c) ∈ pairs [(s₁, c₁), (s₂, c₂) ...] do
6:         sum ← sum + s
7:         cnt ← cnt + c
8:         r_avg ← sum/cnt
9:       Emit(string t, integer r_avg)

Why doesn’t this work?
Computing the Mean: Version 3

```java
1: class Mapper
2:     method MAP(string t, integer r)
3:         Emit(string t, pair (r, 1))

1: class Combiner
2:     method COMBINE(string t, pairs [(s1, c1), (s2, c2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:         Emit(string t, pair (sum, cnt))

1: class Reducer
2:     method REDUCE(string t, pairs [(s1, c1), (s2, c2) ...])
3:         sum ← 0
4:         cnt ← 0
5:         for all pair (s, c) ∈ pairs [(s1, c1), (s2, c2) ...] do
6:             sum ← sum + s
7:             cnt ← cnt + c
8:             r_avg ← sum/cnt
9:         Emit(string t, pair (r_avg, cnt))
```

Fixed?
Computing the Mean: Version 4

```java
1: class Mapper
2:    method Initialize
3:        S ← new AssociativeArray
4:        C ← new AssociativeArray
5:    method Map(string t, integer r)
6:        S{t} ← S{t} + r
7:        C{t} ← C{t} + 1
8:    method Close
9:        for all term t ∈ S do
10:            Emit(term t, pair (S{t}, C{t}))
```

Are combiners still needed?
Algorithm Design: Running Example

- Term co-occurrence matrix for a text collection
  - $M = N \times N$ 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 \((a, b) \rightarrow \text{count}\)

- Reducers sum up counts associated with these pairs

- Use combiners!
Pairs: Pseudo-Code

1: class Mapper
2:   method MAP(dcid a, doc d)
3:     for all term w ∈ doc d do
4:       for all term u ∈ NEIGHBORS(w) do
5:         EMIT(pair (w, u), count 1) \quad \triangleright \text{Emit count for each co-occurrence}

1: class Reducer
2:   method REDUCE(pair p, counts [c₁, c₂, \ldots])
3:     s ← 0
4:     for all count c ∈ counts [c₁, c₂, \ldots] do
5:       s ← s + c \quad \triangleright \text{Sum co-occurrence counts}
6:     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, e) → 3
  - (a, f) → 2

- Each mapper takes a sentence:
  - Generate all co-occurring term pairs
  - For each term, emit a → { b: countb, c: countc, d: countd … }

- Reducers perform element-wise sum of associative arrays

  - a → { b: 1, c: 2, d: 5, e: 3, f: 2 }
  - a → { b: 1,         d: 5, e: 3 }
  - a → { b: 1, c: 2, d: 7, e: 3, f: 2 }

Key: cleverly-constructed data structure brings together partial results
Stripes: Pseudo-Code

1: class Mapper
2:   method Map(docid a, doc d)
3:     for all term $w \in$ doc $d$ do
4:         $H \leftarrow$ new AssociativeArray
5:         for all term $u \in$ Neighbors($w$) do
6:             $H\{u\} \leftarrow H\{u\} + 1$ \quad \triangleright \text{Tally words co-occurring with } \ w
7:     Emit(Term $w$, Stripe $H$)

1: class Reducer
2:   method Reduce(term $w$, stripes [$H_1, H_2, H_3, \ldots$])
3:       $H_f \leftarrow$ new AssociativeArray
4:       for all stripe $H \in$ stripes [$H_1, H_2, H_3, \ldots$] do
5:           Sum($H_f, H$) \quad \triangleright \text{Element-wise sum}
6:       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
Comparison of "pairs" vs. "stripes" for computing word co-occurrence matrices

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

running time (seconds)

size of EC2 cluster (number of slave instances)

relative speedup

$R^2 = 0.997$
Relative Frequencies

- How do we estimate relative frequencies from counts?

\[ f(B|A) = \frac{\text{count}(A, B)}{\text{count}(A)} = \frac{\text{count}(A, B)}{\sum_{B'} \text{count}(A, B')} \]

- 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 \((a, \ast)\)
  - Another pass to directly compute \(f(B|A)\)
f(B | A): “Pairs”

Reducer holds this value in memory

(a, *) → 32

(a, b1) → 3
(a, b2) → 12
(a, b3) → 7
(a, b4) → 1

(a, b1) → 3 / 32
(a, b2) → 12 / 32
(a, b3) → 7 / 32
(a, b4) → 1 / 32

For this to work:

- Must emit extra (a, *) for every bn 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?
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
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, v1)
  - 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
Indexing and Retrieval
First, nomenclature...

- **Information retrieval (IR)**
  - Focus on textual information (= text/document retrieval)
  - Other possibilities include image, video, music, ...

- **What do we search?**
  - Generically, “collections”
  - Less-frequently used, “corpora”

- **What do we find?**
  - Generically, “documents”
  - Even though we may be referring to web pages, PDFs, PowerPoint slides, paragraphs, etc.
The Central Problem in Search

Do these represent the same concepts?

Searcher

Concepts

Query Terms
“tragic love story”

Author

Concepts

Document Terms
“fateful star-crossed romance”
Abstract IR Architecture

Query Representation Function

Query Representation

Comparison Function

Hits

Representation Function

Document Representation

Index

document acquisition (e.g., web crawling)
How do we represent text?

- Remember: computers don’t “understand” anything!

- “Bag of words”
  - Treat all the words in a document as index terms
  - Assign a “weight” to each term based on “importance” (or, in simplest case, presence/absence of word)
  - Disregard order, structure, meaning, etc. of the words
  - Simple, yet effective!

- Assumptions
  - Term occurrence is independent
  - Document relevance is independent
  - “Words” are well-defined
What’s a word?

天主教教宗若望保祿二世因感冒再度住進醫院。
這是他今年第二度因同様的病因住院。

وقال مارك ريجيفر - المناطق باسم الخارجية الإسرائيلية - إن شارون قبيل الدعوة وسيقوم للزيارة الأولى بشريزة تونس، التي كانت لفترة طويلة المقر الرسمي لمنظمة التحرير الفلسطينية، بعود خروجه من لبنان عام 1982.

Выступая в Мещанском суде Москвы экс-глава ЮКОСа заявил не совершал ничего противозаконного, в чем обвиняет его генпрокуратура России.

भारत सरकार ने आर्थिक सर्वेक्षण में वित्तीय वर्ष 2005-06 में सात फीसदी विकास दर हासिल करने का आकलन किया है और कर सुधार पर जोर दिया है

日米連合で台頭中国に対処…アーミテージ前副長官提言

조재영 기자= 서울시는 25일 이명박 시장이 `행정중심복합도시" 건설안에 대해 `군대라도 동원해 막고싶은 심정"이라고 말했다는 일부 언론의 보도를 부인했다.
McDonald's slims down spuds

Fast-food chain to reduce certain types of fat in its french fries with new cooking oil.

NEW YORK (CNN/Money) - McDonald's Corp. is cutting the amount of "bad" fat in its french fries nearly in half, the fast-food chain said Tuesday as it moves to make all its fried menu items healthier.

But does that mean the popular shoestring fries won't taste the same? The company says no. "It's a win-win for our customers because they are getting the same great french-fry taste along with an even healthier nutrition profile," said Mike Roberts, president of McDonald's USA.

But others are not so sure. McDonald's will not specifically discuss the kind of oil it plans to use, but at least one nutrition expert says playing with the formula could mean a different taste.

Shares of Oak Brook, Ill.-based McDonald's (MCD: down $0.54 to $23.22, Research, Estimates) were lower Tuesday afternoon. It was unclear Tuesday whether competitors Burger King and Wendy's International (WEN: down $0.80 to $34.91, Research, Estimates) would follow suit. Neither company could immediately be reached for comment.

“Bag of Words”

14 × McDonalds
12 × fat
11 × fries
8 × new
7 × french
6 × company, said, nutrition
5 × food, oil, percent, reduce, taste, Tuesday

…
Counting Words...

Documents → Bag of Words → Inverted Index

- case folding, tokenization, stopword removal, stemming
  - syntax, semantics, word knowledge, etc.
Boolean Retrieval

- Users express queries as a Boolean expression
  - AND, OR, NOT
  - Can be arbitrarily nested

- Retrieval is based on the notion of sets
  - Any given query divides the collection into two sets: retrieved, not-retrieved
  - Pure Boolean systems do not define an ordering of the results
Inverted Index: Boolean Retrieval

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  1  1
  cat  1  1
  egg  1  1  1
  fish  1  1
  green  1  1
  ham  1  1
  hat  1  1
  one  1
  red  1  1
  two  1

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

- Users express queries as a Boolean expression
  - AND, OR, NOT
  - Can be arbitrarily nested

- Retrieval is based on the notion of sets
  - Any given query divides the collection into two sets: retrieved, not-retrieved
  - Pure Boolean systems do not define an ordering of the results
Strengths and Weaknesses

- **Strengths**
  - Precise, if you know the right strategies
  - Precise, if you have an idea of what you’re looking for
  - Implementations are fast and efficient

- **Weaknesses**
  - Users must learn Boolean logic
  - Boolean logic insufficient to capture the richness of language
  - No control over size of result set: either too many hits or none
  - **When do you stop reading?** All documents in the result set are considered “equally good”
  - **What about partial matches?** Documents that “don’t quite match” the query may be useful also
Ranked Retrieval

- Order documents by how likely they are to be relevant to the information need
  - Estimate relevance \( (q, di) \)
  - Sort documents by relevance
  - Display sorted results

- User model
  - Present hits one screen at a time, best results first
  - At any point, users can decide to stop looking

- How do we estimate relevance?
  - Assume document is relevant if it has a lot of query terms
  - Replace relevance \( (q, di) \) with \( \text{sim}(q, di) \)
  - Compute similarity of vector representations
Assumption: Documents that are “close together” in vector space “talk about” the same things.

Therefore, retrieve documents based on how close the document is to the query (i.e., similarity ~ “closeness”)

Vector Space Model
Similarity Metric

- Use “angle” between the vectors:

\[
\cos(\theta) = \frac{d_j \cdot d_k}{\|d_j\| \|d_k\|}
\]

\[
sim(d_j, d_k) = \frac{d_j \cdot d_k}{\|d_j\| \|d_k\|} = \frac{\sum_{i=1}^{n} w_{i,j} w_{i,k}}{\sqrt[\big]{\sum_{i=1}^{n} w_{i,j}^2} \sqrt[\big]{\sum_{i=1}^{n} w_{i,k}^2}}
\]

- Or, more generally, inner products:

\[
sim(d_j, d_k) = d_j \cdot d_k = \sum_{i=1}^{n} w_{i,j} w_{i,k}
\]
Term Weighting

- Term weights consist of two components
  - Local: how important is the term in this document?
  - Global: how important is the term in the collection?

- Here’s the intuition:
  - Terms that appear often in a document should get high weights
  - Terms that appear in many documents should get low weights

- How do we capture this mathematically?
  - Term frequency (local)
  - Inverse document frequency (global)
TF.IDF Term Weighting

\[ w_{i,j} = \text{tf}_{i,j} \cdot \log \frac{N}{n_i} \]

- \( w_{i,j} \): weight assigned to term \( i \) in document \( j \)
- \( \text{tf}_{i,j} \): number of occurrence of term \( i \) in document \( j \)
- \( N \): number of documents in entire collection
- \( n_i \): number of documents with term \( i \)
Inverted Index: TF.IDF

Doc 1
one fish, two fish

Doc 2
red fish, blue fish

Doc 3
cat in the hat

Doc 4
green eggs and ham

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>blue</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cat</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>egg</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>fish</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>green</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ham</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>hat</td>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>one</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>red</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>two</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>blue</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>cat</td>
<td>1</td>
<td></td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>egg</td>
<td>1</td>
<td></td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>fish</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>green</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ham</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>hat</td>
<td>1</td>
<td></td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>one</td>
<td>1</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>red</td>
<td>1</td>
<td></td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>two</td>
<td>1</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Positional Indexes

- Store term position in postings
- Supports richer queries (e.g., proximity)
- Naturally, leads to larger indexes...
Inverted Index: Positional Information

<table>
<thead>
<tr>
<th>Doc 1</th>
<th>one fish, two fish</th>
<th>Doc 2</th>
<th>red fish, blue fish</th>
<th>Doc 3</th>
<th>cat in the hat</th>
<th>Doc 4</th>
<th>green eggs and ham</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Word</th>
<th>tf</th>
<th>df</th>
</tr>
</thead>
<tbody>
<tr>
<td>blue</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>cat</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>egg</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>fish</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>green</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ham</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>hat</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>one</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>red</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>two</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>blue</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>cat</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>egg</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>fish</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>green</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>ham</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>hat</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>one</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>red</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>two</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Retrieval in a Nutshell

- Look up postings lists corresponding to query terms
- Traverse postings for each query term
- Store partial query-document scores in accumulators
- Select top $k$ results to return
Retrieval: Document-at-a-Time

- Evaluate documents one at a time (score all query terms)
  - blue
    - 9 2 21 1
    - 35 1
  - fish
    - 1 2 9 1 21 3
    - 34 1 35 2 80 3
- Accumulators
  - (e.g. priority queue)
  - Document score in top k?
    - Yes: Insert document score, extract-min if queue too large
    - No: Do nothing

- Tradeoffs
  - Small memory footprint (good)
  - Must read through all postings (bad), but skipping possible
  - More disk seeks (bad), but blocking possible
Retrieval: Query-At-A-Time

- Evaluate documents one query term at a time
  - Usually, starting from most rare term (often with tf-sorted postings)

```
[9 2 21 1 35 1 ...]

[1 2 9 1 21 3 34 1 35 2 80 3 ...]
```

- Tradeoffs
  - Early termination heuristics (good)
  - Large memory footprint (bad), but filtering heuristics possible

Accumulators (e.g., hash)
MapReduce it?

- The indexing problem
  - Scalability is critical
  - Must be relatively fast, but need not be real time
  - Fundamentally a batch operation
  - Incremental updates may or may not be important
  - For the web, crawling is a challenge in itself

- The retrieval problem
  - Must have sub-second response time
  - For the web, only need relatively few results

Perfect for MapReduce!

Uh… not so good…
Indexing: Performance Analysis

- Fundamentally, a large sorting problem
  - Terms usually fit in memory
  - Postings usually don’t
- How is it done on a single machine?
- How can it be done with MapReduce?
- First, let’s characterize the problem size:
  - Size of vocabulary
  - Size of postings
Vocabulary Size: Heaps’ Law

\[ M = kT^b \]

- \( M \) is vocabulary size
- \( T \) is collection size (number of documents)
- \( k \) and \( b \) are constants

Typically, \( k \) is between 30 and 100, \( b \) is between 0.4 and 0.6

- Heaps’ Law: linear in log-log space
- Vocabulary size grows unbounded!
Heaps’ Law for RCV1


- k = 44
- b = 0.49

First 1,000,020 terms:
- Predicted = 38,323
- Actual = 38,365

Manning, Raghavan, Schütze, Introduction to Information Retrieval (2008)
Postings Size: Zipf’s Law

\[ cf_i = \frac{c}{i} \]

\( cf \) is the collection frequency of \( i \)-th common term
\( c \) is a constant

- **Zipf’s Law**: (also) linear in log-log space
  - Specific case of Power Law distributions
- **In other words**:
  - A few elements occur very frequently
  - Many elements occur very infrequently
Zipf’s Law for RCV1


Manning, Raghavan, Schütze, Introduction to Information Retrieval (2008)
Power Laws are everywhere!

MapReduce: Index Construction

- Map over all documents
  - Emit term as key, (docno, tf) as value
  - Emit other information as necessary (e.g., term position)
- Sort/shuffle: group postings by term
- Reduce
  - Gather and sort the postings (e.g., by docno or tf)
  - Write postings to disk
- MapReduce does all the heavy lifting!
Inverted Indexing with MapReduce

<table>
<thead>
<tr>
<th>Map</th>
<th>Doc 1</th>
<th>Doc 2</th>
<th>Doc 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>one fish, two fish</td>
<td>red fish, blue fish</td>
<td>cat in the hat</td>
</tr>
<tr>
<td>one</td>
<td>1 1</td>
<td>red</td>
<td>cat</td>
</tr>
<tr>
<td>two</td>
<td>1 1</td>
<td>blue</td>
<td>hat</td>
</tr>
<tr>
<td>fish</td>
<td>1 2</td>
<td>fish</td>
<td></td>
</tr>
</tbody>
</table>

Shuffle and Sort: aggregate values by keys

<table>
<thead>
<tr>
<th>Reduce</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>3 1</td>
<td>blue</td>
<td></td>
</tr>
<tr>
<td>fish</td>
<td>1 2 2</td>
<td>hat</td>
<td></td>
</tr>
<tr>
<td>one</td>
<td>1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>red</td>
<td>2 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Inverted Indexing: Pseudo-Code

1: class Mapper
2: procedure MAP(docid n, doc d)
3:     H ← new AssociativeArray
4:     for all term t ∈ doc d do
5:         H{t} ← H{t} + 1
6:     for all term t ∈ H do
7:         EMIIT(term t, posting ⟨n, H{t}⟩)

1: class Reducer
2: procedure REDUCE(term t, postings [⟨a₁, f₁⟩, ⟨a₂, f₂⟩ ...])
3:     P ← new List
4:     for all posting ⟨a, f⟩ ∈ postings [⟨a₁, f₁⟩, ⟨a₂, f₂⟩ ...] do
5:         APPEND(P, ⟨a, f⟩)
6:     SORT(P)
7:     EMIIT(term t, postings P)
Positional Indexes

- Store term position in postings
- Supports richer queries (e.g., proximity)
- Naturally, leads to larger indexes...
Inverted Indexing: Pseudo-Code

1: class Mapper
2:     procedure MAP(docid n, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1
6:         for all term t ∈ H do
7:             EMIT(term t, posting ⟨n, H{t}⟩)

1: class Reducer
2:     procedure REDUCE(term t, postings [(a₁, f₁), (a₂, f₂) . . .])
3:         P ← new List
4:         for all posting ⟨a, f⟩ ∈ postings [(a₁, f₁), (a₂, f₂) . . .] do
5:             APPEND(P, ⟨a, f⟩)
6:         SORT(P)
7:         EMIT(term t, postings P)
Scalability Bottleneck

- Initial implementation: terms as keys, postings as values
  - Reducers must buffer all postings associated with key (to sort)
  - What if we run out of memory to buffer postings?
- Uh oh!
Another Try...

How is this different?
- Let the framework do the sorting
- Term frequency implicitly stored
- Directly write postings to disk!

Where have we seen this before?
Postings Encoding

Conceptually:

```
fish  1  2  9  1  21  3  34  1  35  2  80  3  ...
```

In Practice:

- Don’t encode docnos, encode gaps (or $d$-gaps)
- But it’s not obvious that this save space...

```
fish  1  2  8  1  12  3  13  1  1  2  45  3  ...
```
Overview of Index Compression

- Byte-aligned vs. bit-aligned
- Non-parameterized bit-aligned
  - Unary codes
  - $\gamma$ codes
  - $\delta$ codes
- Parameterized bit-aligned
  - Golomb codes (local Bernoulli model)

Want more detail? Read *Managing Gigabytes* by Witten, Moffat, and Bell!
Unary Codes

- $x \geq 1$ is coded as $x-1$ one bits, followed by 1 zero bit
  - $3 = 110$
  - $4 = 1110$
- Great for small numbers… horrible for large numbers
  - Overly-biased for very small gaps

Watch out! Slightly different definitions in different textbooks
**γ codes**

- $x \geq 1$ is coded in two parts: length and offset
  - Start with binary encoded, remove highest-order bit = offset
  - Length is number of binary digits, encoded in unary code
  - Concatenate length + offset codes

- **Example: 9 in binary is 1001**
  - Offset = 001
  - Length = 4, in unary code = 1110
  - $γ$ code = 1110:001

- **Analysis**
  - Offset = $\lfloor \log x \rfloor$
  - Length = $\lceil \log x \rceil + 1$
  - Total = 2 $\lceil \log x \rceil + 1$
δ codes

- Similar to γ codes, except that length is encoded in γ code

- Example: 9 in binary is 1001
  - Offset = 001
  - Length = 4, in γ code = 11000
  - δ code = 11000:001

- γ codes = more compact for smaller numbers
- δ codes = more compact for larger numbers
Golomb Codes

- $x \geq 1$, parameter $b$:
  - $q + 1$ in unary, where $q = \left\lfloor \frac{x - 1}{b} \right\rfloor$
  - $r$ in binary, where $r = x - qb - 1$, in $\lceil \log b \rceil$ or $\lfloor \log b \rfloor$ bits

- Example:
  - $b = 3$, $r = 0, 1, 2$ ($0, 10, 11$)
  - $b = 6$, $r = 0, 1, 2, 3, 4, 5$ ($00, 01, 100, 101, 110, 111$)
  - $x = 9$, $b = 3$: $q = 2$, $r = 2$, code = $110:11$
  - $x = 9$, $b = 6$: $q = 1$, $r = 2$, code = $10:100$

- Optimal $b \approx 0.69 \left( \frac{N}{df} \right)$
  - Different $b$ for every term!
## Comparison of Coding Schemes

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0:0</td>
<td>0:00</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>10:0</td>
<td>100:0</td>
<td>0:10</td>
<td>0:01</td>
</tr>
<tr>
<td>3</td>
<td>110</td>
<td>10:1</td>
<td>100:1</td>
<td>0:11</td>
<td>0:100</td>
</tr>
<tr>
<td>4</td>
<td>1110</td>
<td>110:00</td>
<td>101:00</td>
<td>10:0</td>
<td>0:101</td>
</tr>
<tr>
<td>5</td>
<td>11110</td>
<td>110:01</td>
<td>101:01</td>
<td>10:10</td>
<td>0:110</td>
</tr>
<tr>
<td>6</td>
<td>111110</td>
<td>110:10</td>
<td>101:10</td>
<td>10:11</td>
<td>0:111</td>
</tr>
<tr>
<td>7</td>
<td>1111110</td>
<td>110:11</td>
<td>101:11</td>
<td>110:0</td>
<td>10:00</td>
</tr>
<tr>
<td>8</td>
<td>11111110</td>
<td>1110:000</td>
<td>11000:000</td>
<td>110:10</td>
<td>10:01</td>
</tr>
<tr>
<td>9</td>
<td>111111110</td>
<td>1110:001</td>
<td>11000:001</td>
<td>110:11</td>
<td>10:100</td>
</tr>
<tr>
<td>10</td>
<td>1111111110</td>
<td>1110:010</td>
<td>11000:010</td>
<td>1110:0</td>
<td>10:101</td>
</tr>
</tbody>
</table>

Witten, Moffat, Bell, Managing Gigabytes (1999)
## Index Compression: Performance

Comparison of Index Size (bits per pointer)

<table>
<thead>
<tr>
<th></th>
<th>262</th>
<th>1918</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary</td>
<td>15</td>
<td>20</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>6.51</td>
<td>6.63</td>
</tr>
<tr>
<td>$\delta$</td>
<td>6.23</td>
<td>6.38</td>
</tr>
<tr>
<td>Golomb</td>
<td>6.09</td>
<td>5.84</td>
</tr>
</tbody>
</table>

**Recommend best practice**

**Bible**: King James version of the Bible; 31,101 verses (4.3 MB)

**TREC**: TREC disks 1+2; 741,856 docs (2070 MB)

Witten, Moffat, Bell, Managing Gigabytes (1999)
Chicken and Egg?

But wait! How do we set the Golomb parameter \( b \)?

Recall: optimal \( b \approx 0.69 \) (N/df)

We need the \( df \) to set \( b \)…

But we don’t know the \( df \) until we’ve seen all postings!

Write directly to disk

Sound familiar?
Getting the df

- In the mapper:
  - Emit “special” key-value pairs to keep track of $df$

- In the reducer:
  - Make sure “special” key-value pairs come first: process them to determine $df$

- Remember: proper partitioning!
Getting the df: Modified Mapper

Input document…

Emit normal key-value pairs…

Emit “special” key-value pairs to keep track of df…
Getting the df: Modified Reducer

First, compute the \( df \) by summing contributions from all “special” key-value pairs…

Important: properly define sort order to make sure “special” key-value pairs come first!

Write postings directly to disk
MapReduce it?

- The indexing problem
  - Scalability is critical
  - Must be relatively fast, but need not be real time
  - Fundamentally a batch operation
  - Incremental updates may or may not be important
  - For the web, crawling is a challenge in itself

- The retrieval problem
  - Must have sub-second response time
  - For the web, only need relatively few results

Perfect for MapReduce!

Uh... not so good...
Retrieval with MapReduce?

- MapReduce is fundamentally batch-oriented
  - Optimized for throughput, not latency
  - Startup of mappers and reducers is expensive
- MapReduce is not suitable for real-time queries!
  - Use separate infrastructure for retrieval…
Important Ideas

- Partitioning (for scalability)
- Replication (for redundancy)
- Caching (for speed)
- Routing (for load balancing)

The rest is just details!
Term vs. Document Partitioning
Graph Algorithms
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
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

- Graph algorithms typically 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?
Representing Graphs

- $G = (V, E)$

- Two common representations
  - Adjacency matrix
  - Adjacency list
Adjacency Matrices

Represent a graph as an $n \times n$ square matrix $M$

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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

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
- First, a refresher: Dijkstra’s Algorithm
Dijkstra’s Algorithm Example

Example from CLR
Dijkstra’s Algorithm Example

Example from CLR
Dijkstra’s Algorithm Example

Example from CLR
Dijkstra’s Algorithm Example

Example from CLR
Dijkstra’s Algorithm Example

Example from CLR
Dijkstra’s Algorithm Example
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
- First, a refresher: Dijkstra’s Algorithm
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 \( a \)
  - \( \text{DistanceTo}(s) = 0 \)
  - For all nodes \( p \) reachable from \( s \), \( \text{DistanceTo}(p) = 1 \)
  - For all nodes \( n \) reachable from some other set of nodes \( M \), \( \text{DistanceTo}(n) = 1 + \min(\text{DistanceTo}(m), m \in M) \)
Visualizing Parallel BFS
From Intuition to Algorithm

- **Data representation:**
  - Key: node \( n \)
  - Value: \( d \) (distance from start), adjacency list (list of nodes reachable from \( n \))
  - Initialization: for all nodes except for start node, \( d = \infty \)

- **Mapper:**
  - \( \forall m \in \text{adjacency list}: \text{emit} (m, d + 1) \)

- **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 “known 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 \((n, \text{adjacency list})\) as well
BFS Pseudo-Code

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

1: class Reducer
2:   method Reduce(nid m, [d1, d2, ...])
3:       d_min ← ∞
4:       M ← ∅
5:   for all d ∈ counts [d1, d2, ...] do
6:       if IsNode(d) then
7:         M ← d ▷ Recover graph structure
8:       else if d < d_min then
9:         d_min ← d ▷ Look for shorter distance
10:        M.Distance ← d_min ▷ Update shortest distance
11:       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
- Now answer the question...
  - Six degrees of separation?
- Practicalities of implementation in MapReduce
Comparison to Dijkstra

- Dijkstra’s algorithm is more efficient
  - At any step it only pursues edges from the minimum-cost path inside the 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?
Weighted Edges

- Now add positive weights to the edges
  - Why can’t edge weights be negative?
- Simple change: adjacency list now includes a weight $w$ for each edge
  - In mapper, emit $(m, d + wp)$ instead of $(m, d + 1)$ for each node $m$
- That’s it?
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
- Now answer the question...
  - Six degrees of separation?
- Practicalities of implementation in MapReduce
Additional Complexities

search frontier
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

- Now answer the question...
  - Six degrees of separation?

- Practicalities of implementation in MapReduce
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

- Key questions:
  - How do you represent graph data in MapReduce?
  - How do you traverse a graph in MapReduce?
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
  - Note: query-independent
PageRank: Defined

Given page $x$ with inlinks $t1 \ldots tn$, where

- $C(t)$ is the out-degree of $t$
- $\alpha$ 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 $P_{RI}$ values
  - Each page distributes $P_{RI}$ “credit” to all pages it links to
  - Each target page adds up “credit” from multiple in-bound links to compute $P_{RI+1}$
  - Iterate until values converge
Simplified PageRank

- First, tackle the simple case:
  - No random jump factor
  - No dangling links

- Then, factor in these complexities...
  - Why do we need the random jump?
  - Where do dangling links 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:     \( p \leftarrow N.\text{PageRank} / |N.\text{AdjacencyList}| \)
4:     Emit(nid n, N) \quad \triangleright \text{Pass along graph structure}
5:   for all nodeid \( m \in N.\text{AdjacencyList} \) do
6:     Emit(nid m, p) \quad \triangleright \text{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 \text{counts} \ [\( p_1, p_2, \ldots \)] \) do
5:     if IsNode(p) then
6:       \( M \leftarrow p \) \quad \triangleright \text{Recover graph structure}
7:     else
8:       \( s \leftarrow s + p \) \quad \triangleright \text{Sums incoming PageRank contributions}
9:     M.PageRank \leftarrow s
10:    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
    
    \[ p' = \alpha \left( \frac{1}{|G|} \right) + (1 - \alpha) \left( \frac{m}{|G|} + p \right) \]

  - \( p \) is PageRank value from before, \( p' \) is updated PageRank value
  - \( |G| \) is the number of nodes in the graph
  - \( m \) is the missing PageRank mass
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?
Beyond PageRank

- Link structure is important for web search
  - PageRank is one of many link-based features: HITS, SALSA, etc.
  - One of many thousands of features used in ranking...

- Adversarial nature of web search
  - Link spamming
  - Spider traps
  - Keyword stuffing
  - ...

Efficient Graph Algorithms

- Sparse vs. dense graphs
- Graph topologies
Local Aggregation

- Use combiners!
  - In-mapper combining design pattern also applicable
- Maximize opportunities for local aggregation
  - Simple tricks: sorting the dataset in specific ways
Questions?