Indian Institute of Science
Bangalore, India
भारतीय विज्ञान संस्थान
बंगलौर, भारत

Department of Computational and Data Sciences

DS256:Jan17 (3:1)

L5,6: MapReduce Algorithm Design

Yogesh Simmhan

31 Jan & 2 Feb, 2017

CDS
Department of Computational and Data Sciences

Map-Only Design
Filtering: Distributed Grep

- Input
  - Lines of text from HDFS
  - “Search String” (e.g. regex), input parameter to job
- Mapper
  - Search line for string/pattern
  - Output matching lines
- Reducer
  - Identity function (output = input), or none at all

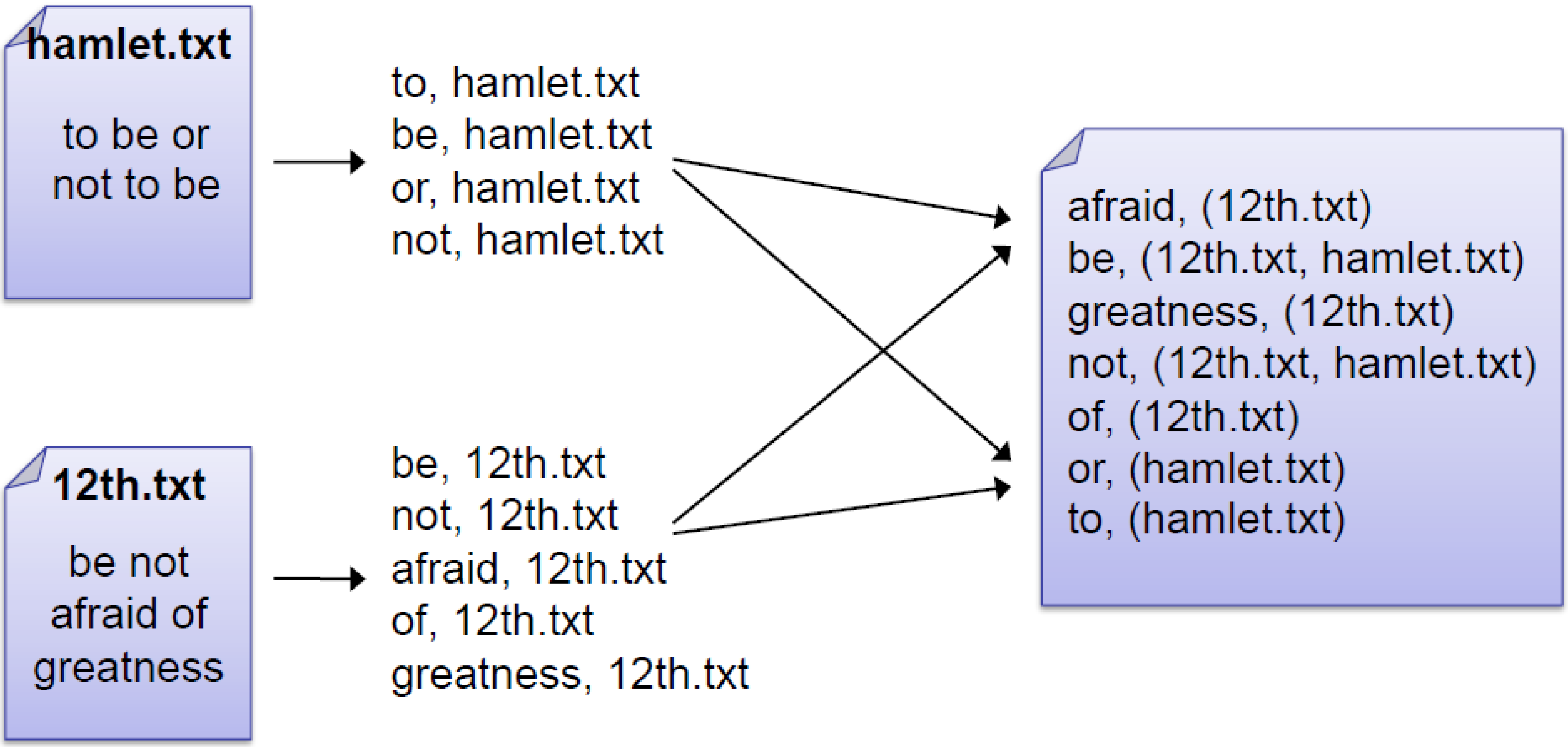
Accumulation (Histogram)

- List of courses with number of students enrolled in each (GoI scheme with citizens enrolled in each)
- Input
  - <StudentID, CourseID>
  - <2482, SE256> <6427, SE252> <1635, E0 259>
- Mapper
  - Emit <CourseID, 1>
  - <SE256, 1>, <SE252, 1>, <E0 259, 1>
- Partition
  - By Course ID
- Sort <E0 259, 1>, <SE252, 1>, <SE256, 1>
- Reduce <E0 259, [1,1]>, <SE252, [1]>, <SE256, [1,1,1]>
  - Count number of students per Course.
  - Output <Course ID, Count>
  - <SE256, 2>, <SE252, 1>, <E0 259, 3>

Inverted Index

- Convert from Key:Values to Value:Keys form
  - E.g. <URL, Lines> ➲ <Word:URL[]>
  - Useful for building search index
- Input: <URL, Line>
- Map: `foreach(Word in Line) emit(Word, URL)`
- Combiner: Combine URLs for same Word
- Reduce: `emit(Word, sort(URL[ ]))`

Inverted Index Example



Join

Customers

cfirstname	clastname	cphone	cstreet	czipcode
Tom	Jewett	714-555-1212	10200 Slater	92708
Alvaro	Monge	562-333-4141	2145 Main	90840
Wayne	Dick	562-777-3030	1250 Bellflower	90840

Orders

cfirstname	clastname	cphone	orderdate	soldby
Alvaro	Monge	562-333-4141	2003-07-14	Patrick
Wayne	Dick	562-777-3030	2003-07-14	Patrick
Alvaro	Monge	562-333-4141	2003-07-18	Kathleen
Alvaro	Monge	562-333-4141	2003-07-20	Kathleen

Customers joined to Orders

cfirstname	clastname	cphone	cstreet	czipcode	orderdate	soldby
Alvaro	Monge	562-333-4141	2145 Main	90840	2003-07-14	Patrick
Wayne	Dick	562-777-3030	1250 Bellflower	90840	2003-07-14	Patrick
Alvaro	Monge	562-333-4141	2145 Main	90840	2003-07-18	Kathleen
Alvaro	Monge	562-333-4141	2145 Main	90840	2003-07-20	Kathleen

http://www.tomjewett.com/dbdesign/dbdesign.php?page=join.php

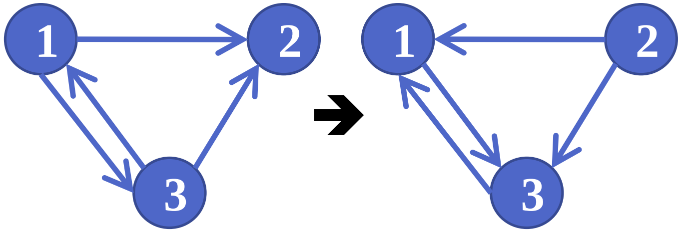
Join

- Given two sets of files, combine the lines having the same key in each file
- Input:
  - <customer_data>, <order_data>
- Mapper:
  - emit <cell, <t1,customer_data>>, <cell, <t2,order_data>>
- Reduce:
  - If only one table ID (customer or order value) present, skip
  - If 2 values present, one from each tables ID
    - Just concatenate and emit the pair
    - <cell, [customer_data, order_data]>
  - If multiple values present for each table ID,
    - Emit cross product of customer_data* and order_data* values, i.e., local join for each cell key
    - <cell, [customer_data*, order_data*]>

Reverse graph edge directions & output in node order

- Input: adjacency list of graph (e.g. 3 nodes and 4 edges)

(3, [1, 2]) (1, [3])
(1, [2, 3]) ➔ (2, [1, 3])
(3, [1])



- node_ids in the output values are also sorted.
  But Hadoop only sorts on keys!
- MapReduce format
  - Input: (3, [1, 2]), (1, [2, 3]).
  - Intermediate: (1, [3]), (2, [3]), (2, [1]), (3, [1]). (reverse edge direction)
  - Out: (1,[3]) (2, [1, 3]) (3, [[1]).

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

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

Advanced Algorithms

Term Co-occurance

- Term co-occurrence matrix for a text collection
  - M = N x N matrix (N = vocabulary size)
  - M_{ij}: number of times *i* and *j* co-occur in some context
    (for concreteness, let's say context = sentence)
- Why?
  - Distributional profiles as a way of measuring semantic distance
  - Semantic distance useful for many language processing tasks

MapReduce: Large Counting Problems

- Term co-occurrence matrix for a text collection = specific instance of a large counting problem
  - A large event space (number of terms)
  - A large number of observations (the collection itself)
  - Goal: keep track of interesting statistics about the events
- How do we compute using MapReduce?
  - Map Input: DocID, DocContent
- 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 Approach

```
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)
```

Figure 3.8: Pseudo-code for the "pairs" approach for computing word co-occurrence matrices from large corpora.

- Mapper emits many intermediate pairs (cell values)
- Combiner operates on sparse keys

TB Sec 3.2, Lin, et al,

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

a → { b: 1, c: 2, d: 5, e: 3, f: 2 }

- Each mapper takes a sentence:
  - Generate all co-occurring term pairs
  - For each term, emit a → { b: $count_b$, c: $count_c$, d: $count_d$ … }

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 }

- Reducers perform element-wise sum of associative arrays

Stripes Approach

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

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

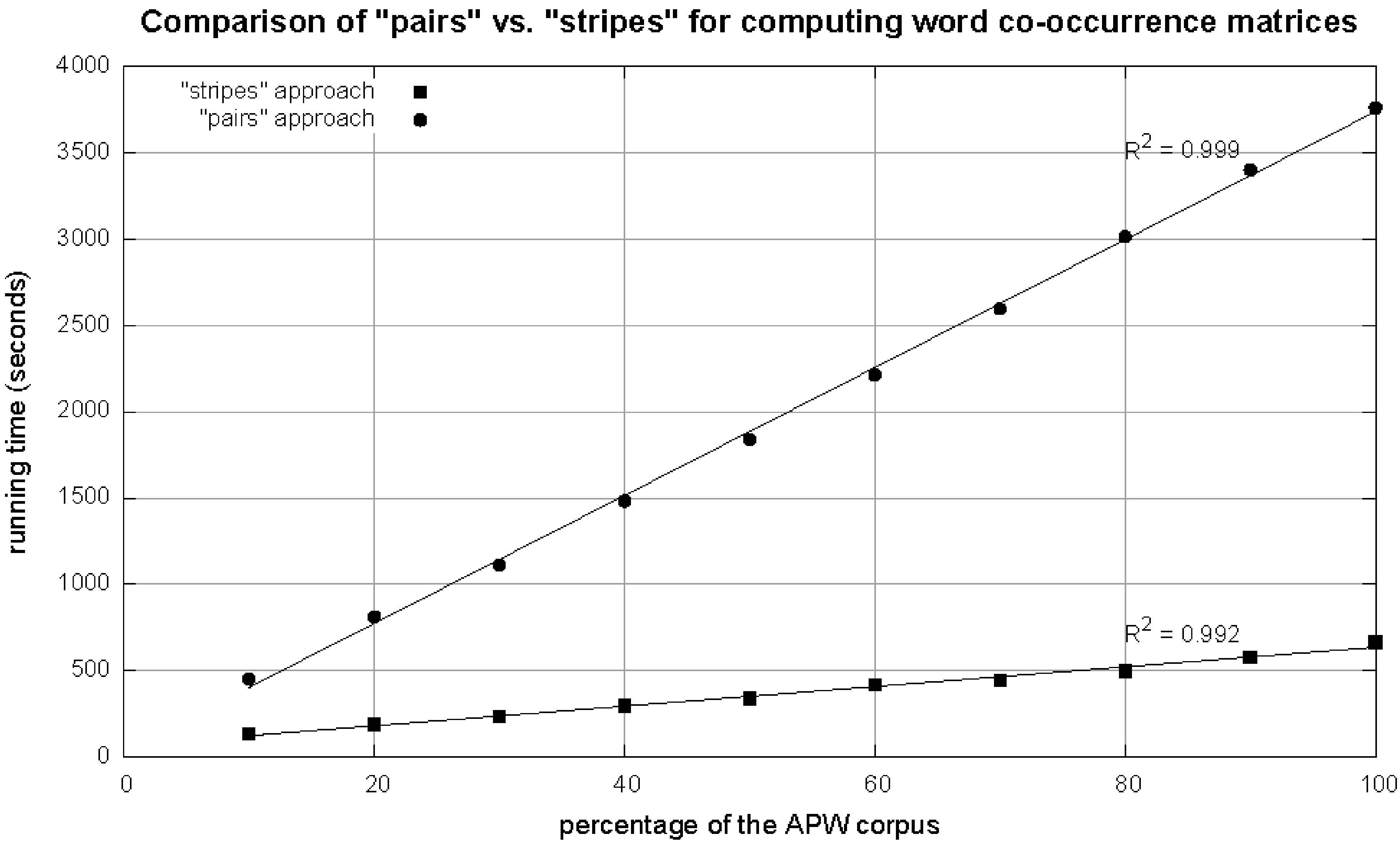
Figure 3.9: Pseudo-code for the "stripes" approach for computing word co-occurrence matrices from large corpora.

- Mapper emits entire row at a time
- Combiner & Reducer operate on fewer keys
- Need to store entire row in memory!

TB Sec 3.2, Lin, et al,

"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)

TB Sec 3.2, Lin, et al,

Relative Frequencies

- How do we estimate relative frequencies from counts?

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

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

f(B|A): "Stripes"

- Easy!
  - One pass to compute (a, *)
  - Another pass to directly compute f(B|A)

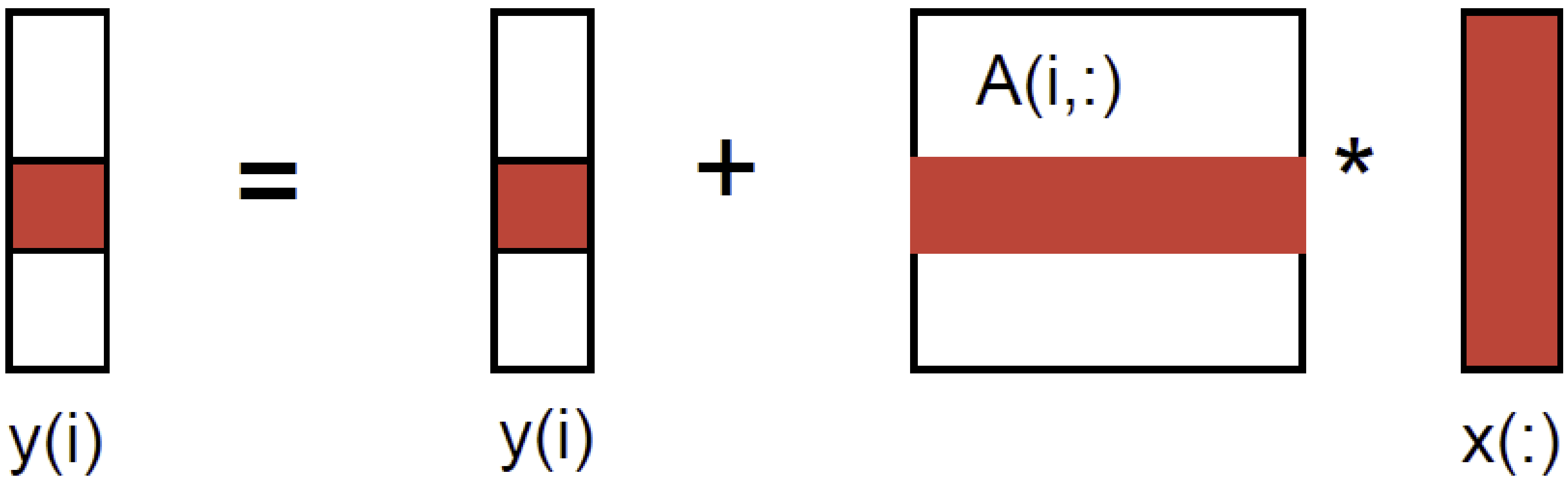
f(B|A): "Pairs"

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

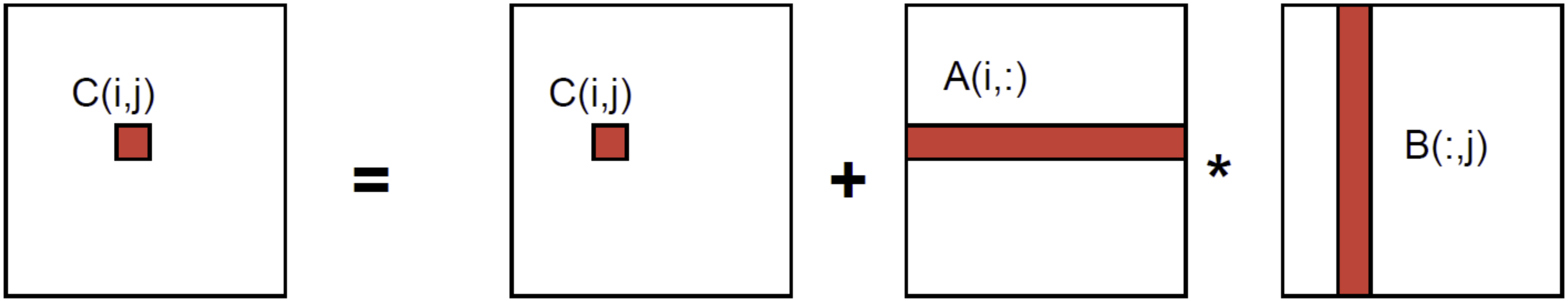
$(a, b_1) \rightarrow 3$	$(a, b_1) \rightarrow 3 / 32$
$(a, b_2) \rightarrow 12$	$(a, b_2) \rightarrow 12 / 32$
$(a, b_3) \rightarrow 7$	$(a, b_3) \rightarrow 7 / 32$
$(a, b_4) \rightarrow 1$	$(a, b_4) \rightarrow 1 / 32$
…	…

- For this to work:
  - Must emit extra (a, *) for every b_n in mapper
  - Must make sure all a's get sent to same reducer (use partitioner)
  - Must make sure (a, *) comes first (define sort order)
  - Must hold state in reducer across different key-value pairs

Matrix-Vector Multiply



Matrix-Matrix Multiply



Sparse Matrix Multiplication

$$\overset{\textbf{A}}{\begin{bmatrix} 10 & & 20 \\ & 30 & 40 \\ 50 & 60 & 70 \end{bmatrix}} \times \overset{\textbf{B}}{\begin{bmatrix} -1 & \\ -2 & -3 \\ & -4 \end{bmatrix}} = \overset{\textbf{C}}{\begin{bmatrix} -10 & -80 \\ -60 & -250 \\ -170 & -460 \end{bmatrix}}$$

- Task: Compute product C = A·B
- Assume most matrix entries are 0

- Motivation
  - Core problem in scientific computing
  - Challenging for parallel execution
  - Demonstrate expressiveness of Map/Reduce

Sparse Matrix Multiplication

A

$$\begin{bmatrix} 10 & & 20 \\ & 30 & 40 \\ 50 & 60 & 70 \end{bmatrix}$$

$1 \xrightarrow[A]{10} 1$

$1 \xrightarrow[A]{20} 3$

$2 \xrightarrow[A]{30} 2$

$2 \xrightarrow[A]{40} 3$

$3 \xrightarrow[A]{50} 1$

$3 \xrightarrow[A]{60} 2$

$3 \xrightarrow[A]{70} 3$

B

$$\begin{bmatrix} -1 & \\ -2 & -3 \\ & -4 \end{bmatrix}$$

$1 \xrightarrow[B]{-1} 1$

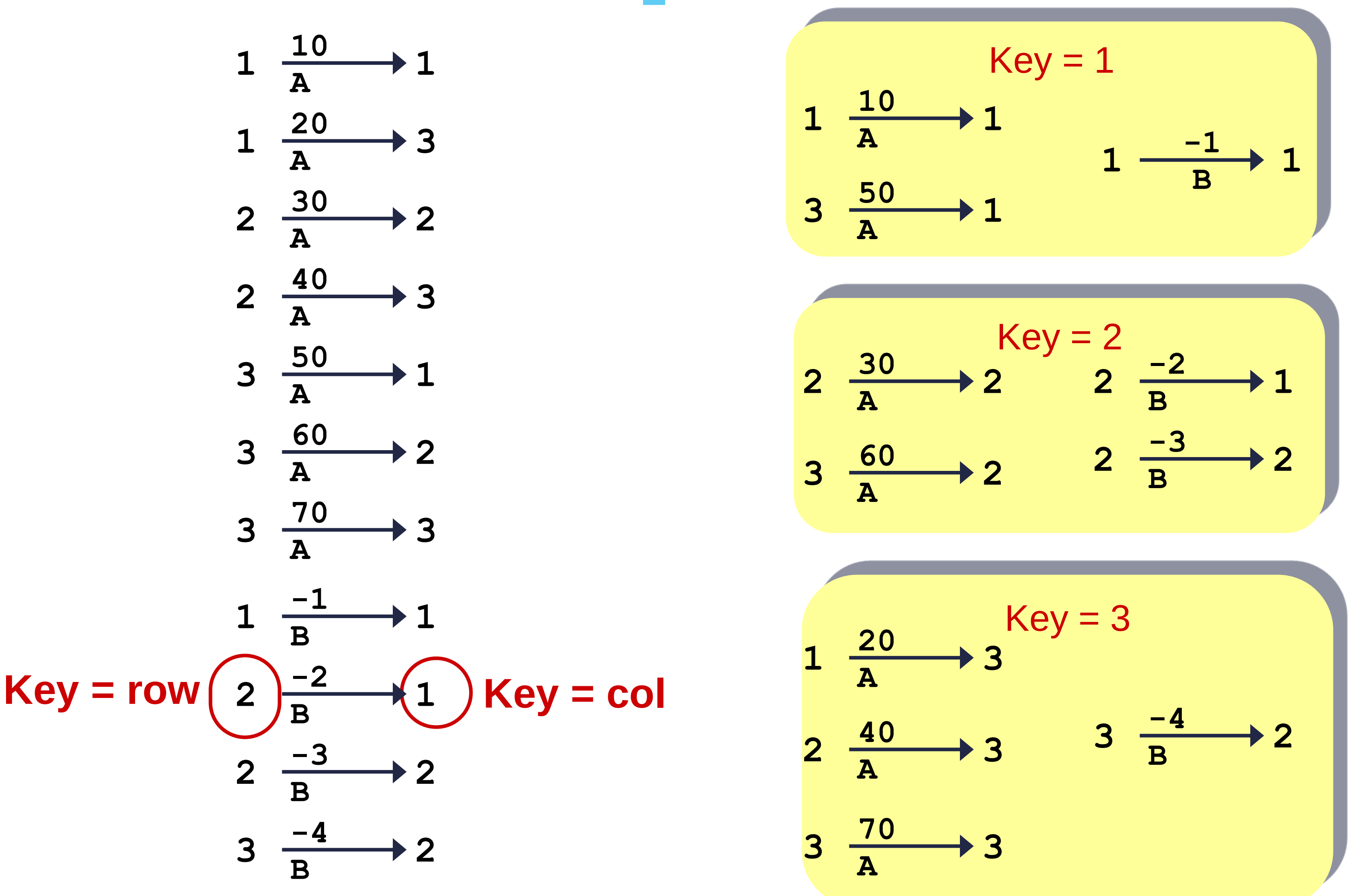
$2 \xrightarrow[B]{-2} 1$

$2 \xrightarrow[B]{-3} 2$

$3 \xrightarrow[B]{-4} 2$

- Represent matrix as list of nonzero entries
  - ⟨row, col, value, matrixID⟩
- Strategy
  - Phase 1: Compute all products ai,k · bk,j
  - Phase 2: Sum products for each entry i,j
  - Each phase involves a Map/Reduce

Phase 1: Map



- Group values ai,k and bk,j according to key k

Phase 1: Reduce

Key = 1

$1 \xrightarrow[A]{10} 1$

$3 \xrightarrow[A]{50} 1$

X $1 \xrightarrow[B]{-1} 1$

Key = 2

$2 \xrightarrow[A]{30} 2$

$3 \xrightarrow[A]{60} 2$

X $2 \xrightarrow[B]{-2} 1$

$2 \xrightarrow[B]{-3} 2$

Key = 3

$1 \xrightarrow[A]{20} 3$

$2 \xrightarrow[A]{40} 3$

$3 \xrightarrow[A]{70} 3$

X $3 \xrightarrow[B]{-4} 2$

$1 \xrightarrow[C]{-10} 1$

$3 \xrightarrow[A]{-50} 1$

$2 \xrightarrow[C]{-60} 1$

$2 \xrightarrow[C]{-90} 2$

$3 \xrightarrow[C]{-120} 1$

$3 \xrightarrow[C]{-180} 2$

$1 \xrightarrow[C]{-80} 2$

$2 \xrightarrow[C]{-160} 2$

$3 \xrightarrow[C]{-280} 2$

- Generate all products ai,k · bk,j

Phase 2: Map

$1 \xrightarrow[C]{-10} 1$

$3 \xrightarrow[A]{-50} 1$

$2 \xrightarrow[C]{-60} 1$

$2 \xrightarrow[C]{-90} 2$

$3 \xrightarrow[C]{-120} 1$

$3 \xrightarrow[C]{-180} 2$

$1 \xrightarrow[C]{-80} 2$

$2 \xrightarrow[C]{-160} 2$

$3 \xrightarrow[C]{-280} 2$

Key = row,col

Key = 1,1	$1 \xrightarrow[C]{-10} 1$
Key = 1,2	$1 \xrightarrow[C]{-80} 2$
Key = 2,1	$2 \xrightarrow[C]{-60} 1$
Key = 2,2	$2 \xrightarrow[C]{-90} 2$ $2 \xrightarrow[C]{-160} 2$
Key = 3,1	$3 \xrightarrow[C]{-120} 1$ $3 \xrightarrow[A]{-50} 1$
Key = 3,2	$3 \xrightarrow[C]{-280} 2$ $3 \xrightarrow[C]{-180} 2$

- Group products ai,k · bk,j with matching values of i and j

Phase 2: Reduce

Key = 1,1: $1 \xrightarrow[C]{-10} 1$

Key = 1,2: $1 \xrightarrow[C]{-80} 2$

Key = 2,1: $2 \xrightarrow[C]{-60} 1$

Key = 2,2: $2 \xrightarrow[C]{-90} 2$, $2 \xrightarrow[C]{-160} 2$

Key = 3,1: $3 \xrightarrow[C]{-120} 1$, $3 \xrightarrow[A]{-50} 1$

Key = 3,2: $3 \xrightarrow[C]{-280} 2$, $3 \xrightarrow[C]{-180} 2$

final entries

$1 \xrightarrow[C]{-10} 1$

$1 \xrightarrow[C]{-80} 2$

$2 \xrightarrow[C]{-60} 1$

$2 \xrightarrow[C]{-250} 2$

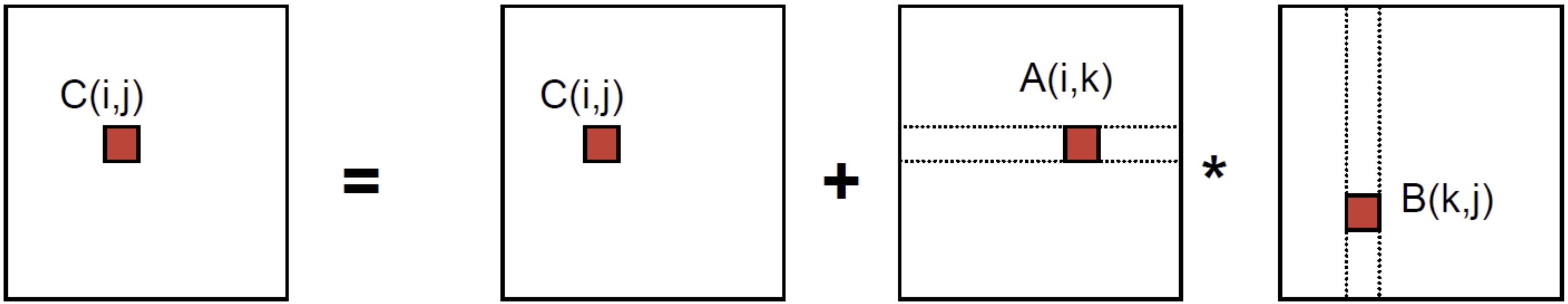
$3 \xrightarrow[C]{-170} 1$

$3 \xrightarrow[C]{-460} 2$

C

$$\begin{array}{cc} -10 & -80 \\ -60 & -250 \\ -170 & -460 \end{array}$$

Block Matrix Multiply



PageRank

- Centrality measure of web page quality based on the web structure
  - How important is this vertex in the graph?
- Random walk
  - Web surfer visits a page, randomly clicks a link on that page, and does this repeatedly.
  - How frequently would each page appear in this surfing?
- Intuition
  - Expect high-quality pages to contain “endorsements” from many other pages thru hyperlinks
  - Expect if a high-quality page links to another page, then the second page is likely to be high quality too

Lin, Ch 5.3 PAGERANK

PageRank, recursively

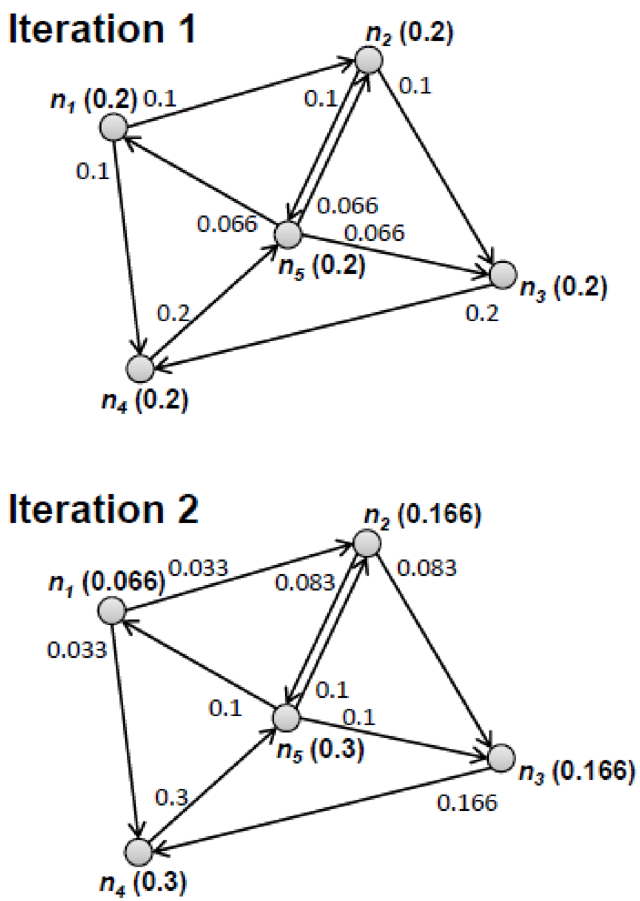
$$P(n) = \alpha \left(\frac{1}{|G|} \right) + (1-\alpha) \sum_{m \in L(n)} \frac{P(m)}{C(m)}$$

- P(n) is PageRank for webpage/URL ‘n’
  - Probability that you’re in vertex ‘n’
- |G| is number of URLs (vertices) in graph
- α is probability of random jump
- L(n) is set of vertices that link to ‘n’
- C(m) is out-degree of ‘m’

PageRank Iterations

α=0

Initialize P(n)=1/|G|



Lin, Fig 5.7

PageRank using MapReduce

```
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                           ▷ Sum incoming PageRank contributions
        M.PageRank ← s
        Emit(nid m, node M)
```

Lin, Fig 5.8

PageRank using MapReduce

- MR run over multiple iterations (typically 30)
  - *The graph structure itself must be passed from iteration to iteration!*
- Mapper will
  - Initially, load adjacency list and initialize default PR
    - `<v1, <v2>+>`
  - Subsequent iterations will load adjacency list and new PR
    - `<v1, <v2>+, pr1>`
  - Emit two types of messages from Map
    - PR messages and Graph Structure Messages
- Reduce will
  - Reconstruct the adjacency list for each vertex
  - Update the PageRank values for the vertex based on neighbour's PR messages
  - Write adjacency list and new PR values to HDFS, to be used by next Map iteration
    - `<v1, <v2>+, pr1'>`

Inverted Indexes Revisited

- Each Map task parses one or more webpages
  - Input: A stream of webpages (WARC)
  - Output: A stream of (term, URL) tuples
    - (long, http://gb.com) (long, http://gb.com) (ago, http://gb.com) … (long, http://jn.in) (years, http://jn.in) (ago, http://jn.in) …
- Shuffle sorts by key and routes tuples to Reducers
- Reducers convert streams of keys into streams of inverted lists
  - Sorts the values for a key (why?) and builds an inverted list
  - Output: (long, [http://gb.com, http://jn.in]), (ago, [http://gb.com, http://jn.in]), (years, [http://jn.in])

Optimizations & Extensions

- URL sizes may be large
  - Replace URLs with unique longs, URL ID
    - Mapping from URL ID to URL saved as a file
    - Inverted Index has <term, [URL ID]+>
  - Skip stop words with lot of matching URLs
  - Use combiners
- Partition term by prefix alphabet(s)
  - One reducer for each term starting with "a", "b", etc.
  - Part file from each reducer has terms with unique a starting letter
- Additional metadata
  - Idea: Include a mapping from URL ID to <URL, PageRank>?
  - Include "term frequency" of term occurrence per URL ID in Inverted Index?

Challenge

- Even using URL IDs, all IDs per term *may not fit in reduce memory for sorting*
  - E.g. 17M URLs in 1% of CC data.
  - Say 1000 unique words per URL.
  - So 17B keys and values generated by Mappers.
  - Say 50,000 unique words (keys) in English
  - One key would on average have 17B/50K=340K URL IDs
    - Peak values would be much higher
- Use a value-to-key conversion design pattern
  - Let MR perform sorting, Reducer just emits result

```
class Mapper
    method Map(docid n, doc d)
        H ← new AssociativeArray
        for all term t ∈ doc d do
            H{t} ← H{t} + 1
        for all term t ∈ H do
            Emit(tuple ⟨t, n⟩, tf H{t})

class Reducer
    method Initialize
        t_prev ← ∅
        P ← new PostingsList
    method Reduce(tuple ⟨t, n⟩, tf [f])
        if t ≠ t_prev ∧ t_prev ≠ ∅ then
            Emit(term t, postings P)
            P.Reset()
        P.Add(⟨n, f⟩)
        t_prev ← t
    method Close
        Emit(term t, postings P)
```

- Mapper emits <**<term, URL ID>**, tf>
    - i.e. compound key
- Partitioner sends all terms to the same reducer
- Per reducer, MR sorts based on compound key <term, URL ID>
- Only one value for each compound key
- Reduce task gets list of term and URL ID in sorted order
    - When new term seen, flush index for “prev” term and start new term
- E.g.
    - <<Ago, 1>, tf1>
    - <<Ago, 7>, tf7>
        - Flush <Ago, [<1,tf1>,<7,tf7>]
    - <<Long, 3>, tf3>
    - <<Long, 4>, tf4>
    - <<Long, 6>, tf6>
        - Flush <Long, [<3,tf3>,<4,tf4>,<6,tf6>]

Lin, Figure 4.4

Lookup of Terms

- Each Map task loads one of the index files, say, by alphabet
- Input terms e.g. "t1 & t2 & t3" passed to each Map task as AND search
- Map does lookup and sends <URL ID, t_i> to reducer
  - Optionally send <<PR, URL ID>, t_i> for sorting by PR
- Reducer does set intersection of all t_i for a URL ID
  - If all terms match, looks up URL for the URL ID
  - If PR stored for each URL, that is returned too

Thank You!