Chapter 4

Distributed Algorithms based on MapReduce

- Applications

Acknowledgements

- MapReduce Algorithms - Understanding Data Joins：http://codingjunkie.net/mapreduce-reduce-joins/
- Joins with Map Reduce： https://chamibuddhika.wordpress.com/2012/02/26/joins-with-map-reduce/
- MapReduce Example: Reduce Side Join in Hadoop MapReduce: https://www.edureka.co/blog/mapreduce-example-reduce-side-join/
- Map Reduce Advanced - Relational Join: https://www.hackerrank.com/challenges/map-reduce-advanced-relational-join
- Map Side Join Vs. Join: https://www.edureka.co/blog/map-side-join-vs-join/
- Linux grep command: https://www.computerhope.com/unix/ugrep.htm.
- MapReduce Algorithms CSE 490H.
- Applications of Map-Reduce. Team 3, CS 4513 – D08, WPI.
- Data-Intensive Scalable Computing with MapReduce, CS290N, Spring, 2010, Spinnaker Labs, Inc.
- PageRank: http://www.cnblogs.com/rubinorth/p/5799848.html
- PageRank with MapReduce: http://www.cnblogs.com/fengfenggirl/p/pagerank-introduction.html

MapReduce Algorithm Design

- Aspects that are **NOT** under the control of the designer
  - Where a mapper or reducer will run
  - When a mapper or reducer begins or finishes
  - Which input key-value pairs are processed by a specific mapper
  - Which intermediate key-value paris are processed by a specific reducer
- Aspects that **CAN** be controlled
  - Construct data structures as keys and values
  - Execute user-specified initialization and termination code for mappers and reducers (pre-process and post-process)
  - Preserve state across multiple input and intermediate keys in mappers and reducers (in-mapper combining)
  - Control the sort order of intermediate keys, and therefore the order in which a reducer will encounter particular keys (order inversion)
  - Control the partitioning of the key space, and therefore the set of keys that will be encountered by a particular reducer (partitioner)

Algorithms for MapReduce

1. Distributed Grep
2. Secondary sort
3. Relational join
4. Search engine
5. Graph (BFS)
6. PageRank

Distributed Grep

Distributed Grep

- **G**lobally search a **R**egular **E**xpression and **P**rint
- A command-line utility for searching plain-text data sets for lines that match a regular expression
- Syntax:
  - Standalone (more efficient):

    grep [*OPTIONS*] PATTERN [*FILE*...]
  - Pipelined with cat/awk:

    cat [FILE] | grep PATTERN [| more/less]

Simple usages

```
user@computer: ~/documents/html
$ grep "our products" product-listing.html
<p>You will find that all of our products are impeccably designed and
meet the highest manufacturing standards available <em>anywhere.</em><
/p>
$
```

```
$ grep --color -i -n -r "our products" *
product-details.html:27:<p><b>OUR PRODUCTS</b></p>
product-details.html:59:<p class="products-searchbox">To search a comp
rehensive list of our products, type your search term in the box below
 and click the magnifying glass.</p>
product-listing.html:18:<p>You will find that all of our products are
impeccably designed and meet the highest manufacturing standards avail
able <em>anywhere.</em></p>
product-listing.html:23:<p class="listing">Our products are manufactur
ed using only the finest top-grain leather.</p>
product-overview.html:30:<p>To learn more about our products, please e
mail us at the link below.</p>
shop/shopping-cart.html:14:<p>Our products ship to you in 12 hours or
less, guaranteed.</p>
shop/shopping-cart.html:17:<p class="section-head">The most popular of
 our products are listed here:</p>
xml/dynamic-content.xml:29:<page01 ref="prods_01_a" ref_coord="83,12,1
7"<![CDATA[<p>Click here to email us about our products.</p>]]></page0
1>
$
```

--color: output in color

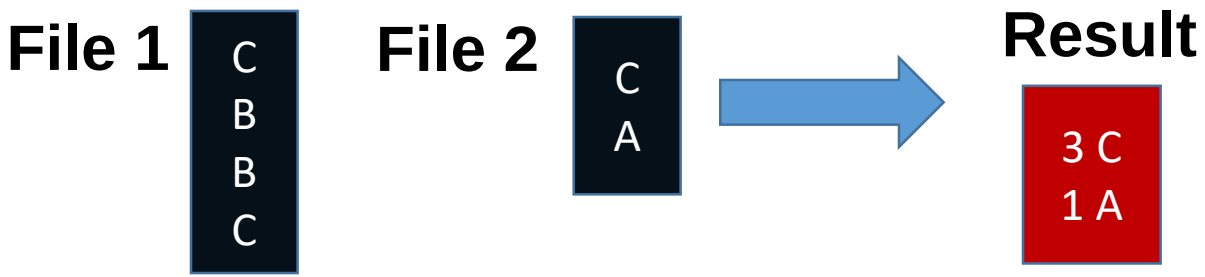
-n: show line numbers of successful matches

-i: case-insensitive match

-r: recursively search

Example

- **grep -Eh 'A|C' in/* | sort | uniq -c | sort –nr**
  - counts lines in all files in <inDir> that match <regex> and displays the counts in descending order

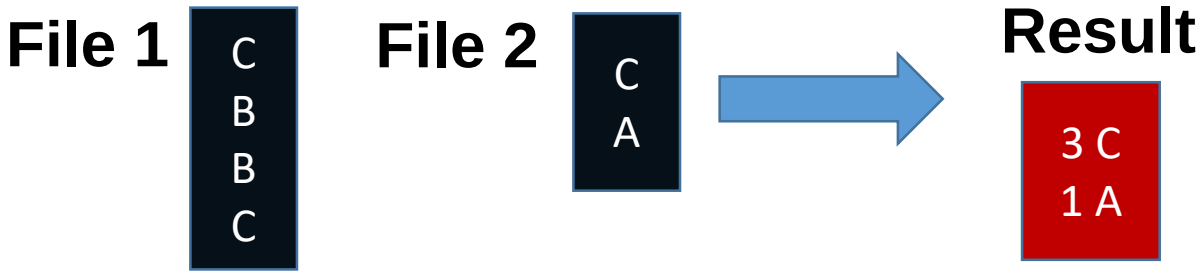


  - Analyzing web server access logs to find the top requested pages that match a given pattern

Distributed Grep

- Map function in this case:
  - input is (file offset, line)
  - output is either:
    - an empty list [] (the line does not match)
    - a key-value pair [(line, 1)] (if it matches)
- Reduce function in this case:
  - input is (line, [1, 1, ...])
  - output is (line, n) where n is the number of 1s in the list.

Distributed Grep



Map tasks:
(0, C) -> [(C, 1)]
(2, B) -> []
(4, B) -> []
(6, C) -> [(C, 1)]
(0, C) -> [(C, 1)]
(2, A) -> [(A, 1)]

Reduce tasks:
(A, [1]) -> (A, 1)
(C, [1, 1, 1]) -> (C, 3)

Secondary Sort

What is Secondary Sort

- MapReduce sorts input to reducers by key
  - Values may be arbitrarily ordered

- What if want to sort value as well?
  - E.g., $k \rightarrow (v_1, r), (v_3, r), (v_4, r), (v_8, r)\ldots$
  - Google's MapReduce implementation provides built-in functionality
  - Unfortunately, Hadoop does not support

- **Secondary Sort**: sorting values associated with a key in the reduce phase, also called "**value-to-key conversion**"

Example 1

- Sensor data from a scientific experiment: there are *m* sensors each taking readings on continuous basis

  (t1, m1, r_{80521})

  (t1, m2, r_{14209})

  (t1, m3, r_{76742})

  …

  (t2, m1, r_{21823})

  (t2, m2, r_{66508})

  (t2, m3, r_{98347})

- We wish to reconstruct the activity at each individual sensor over time
- In a MapReduce program, a mapper may emit the following pair as the intermediate result

  m_1 -> (t_1, r_{80521})

  - We need to sort the value according to the timestamp

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 (m_1, t_1)
    - The mapper emits (m_1, t_1) -> r_{80521}
  - Let execution framework do the sorting
  - Preserve state across multiple key-value pairs to handle processing
  - Anything else we need to do?
    - Sensor readings are split across multiple keys. Reducers need to know when all readings of a sensor have been processed
    - All pairs associated with the same sensor are shuffled to the same reducer (use partitioner)

Example 2

- Consider the temperature data from a scientific experiment. Columns are *year*, *month*, *day*, and daily *temperature*, respectively:

```
2012, 01, 01, 5
2012, 01, 02, 45
2012, 01, 03, 35
2012, 01, 04, 10
...
2001, 11, 01, 46
2001, 11, 02, 47
2001, 11, 03, 48
2001, 11, 04, 40
...
2005, 08, 20, 50
2005, 08, 21, 52
2005, 08, 22, 38
2005, 08, 23, 70
```

```
2012-01:  5, 10, 35, 45, ...
2001-11: 40, 46, 47, 48, ...
2005-08: 38, 50, 52, 70, ...
```

- We want to output the temperature for every year-month with the values sorted in ascending order.

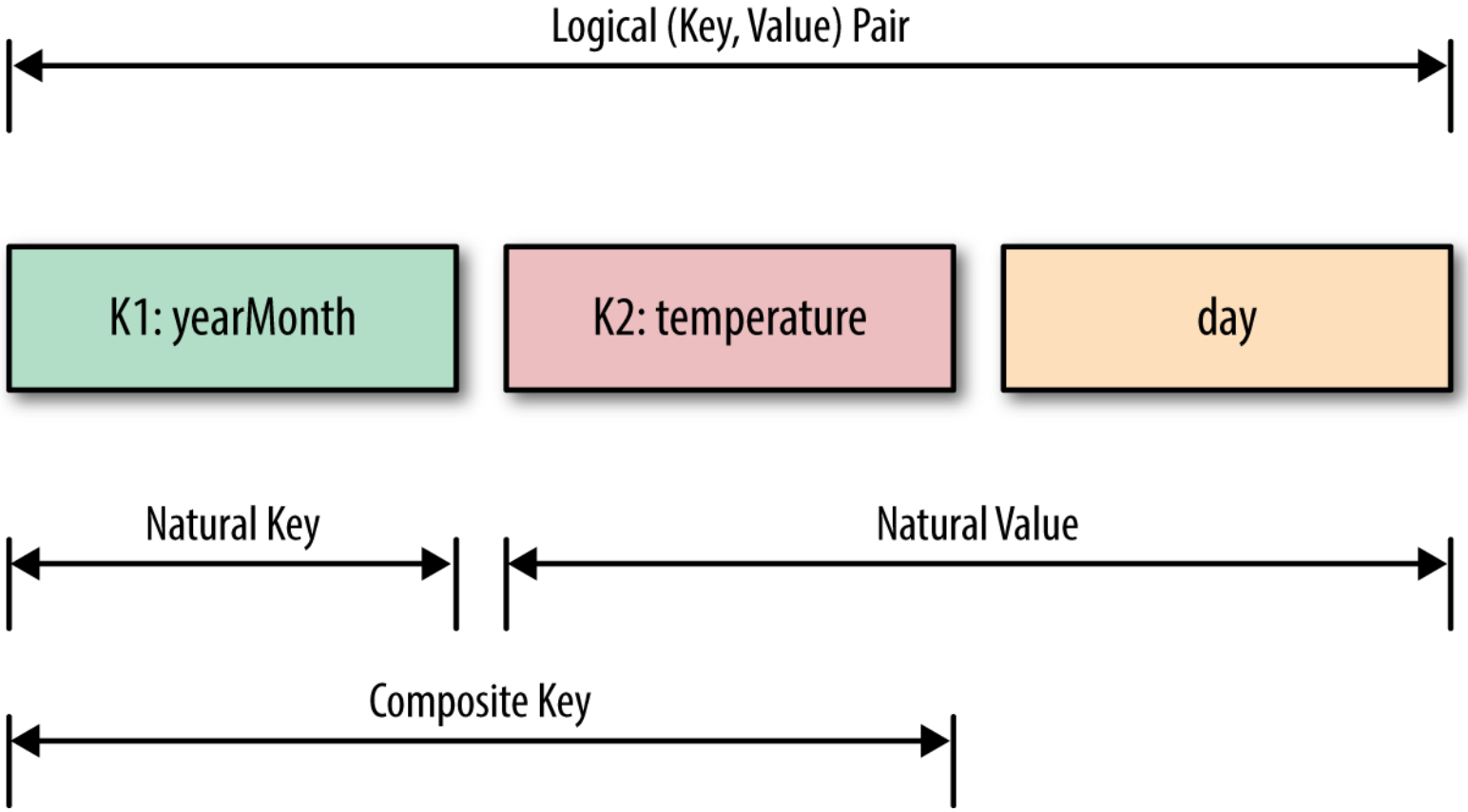
Solutions

- Use the **Value-to-Key Conversion** design pattern:
  - form a composite intermediate key, (K, V), where V is the secondary key. Here, K is called a *natural key*. To inject a value (i.e., V) into a reducer key, simply create a composite key
    - K: year-month
    - V: temperature data

- Let the MapReduce execution framework do the sorting (rather than sorting in memory, let the framework sort by using the cluster nodes).

- Preserve state across multiple key-value pairs to handle processing. Write your own partitioner: partition the mapper's output by the natural key (year-month).

Secondary Sorting Keys



Customize The Composite Key

```
public class DateTemperaturePair
        implements Writable, WritableComparable<DateTemperaturePair> {
            private Text yearMonth = new Text(); // natural key
            private IntWritable temperature = new IntWritable(); // secondary key
            … …
            @Override
            /**
            * This comparator controls the sort order of the keys.
            */
            public int compareTo(DateTemperaturePair pair) {
                int compareValue = this.yearMonth.compareTo(pair.getYearMonth());
                if (compareValue == 0) {
                    compareValue = temperature.compareTo(pair.getTemperature());
                }
                return compareValue; // sort ascending
            }
            … …
}
```

Customize The Partitioner

```
public class DateTemperaturePartitioner
        extends Partitioner<DateTemperaturePair, Text> {
            @Override
            public int getPartition(DateTemperaturePair pair, Text text, int numberOfPartitions) {
            // make sure that partitions are non-negative
                return Math.abs(pair.getYearMonth().hashCode() % numberOfPartitions);
            }
}
```

Utilize the natural key only for partitioning

Grouping Comparator

- Controls which keys are grouped together for a single call to Reducer.reduce() function.

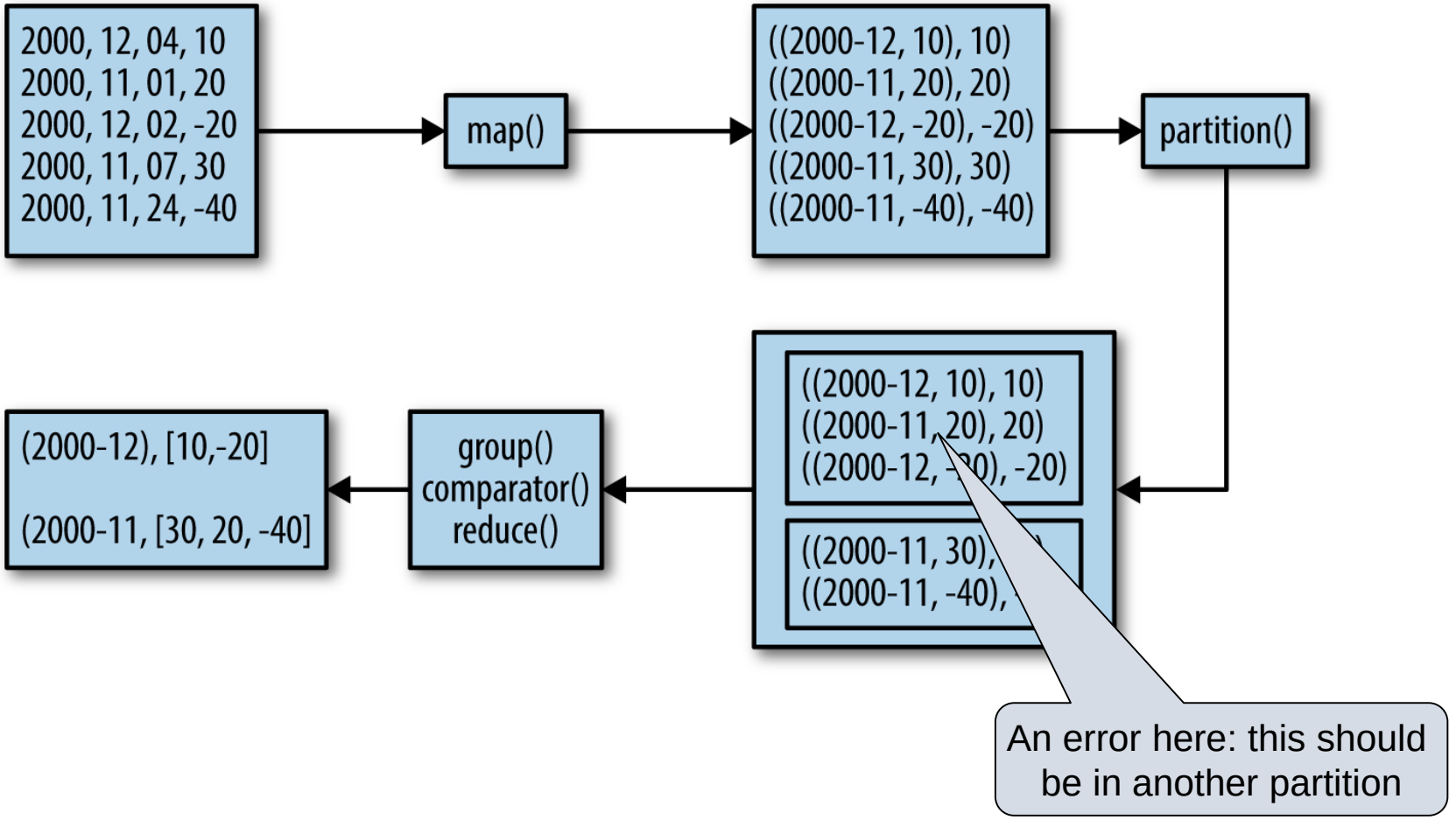
```
public class DateTemperatureGroupingComparator extends WritableComparator {
        … …
        protected DateTemperatureGroupingComparator(){
                super(DateTemperaturePair.class, true);
        }
        @Override
        /* This comparator controls which keys are grouped together into a si
method */
        public int compare(WritableComparable wc1, WritableComparable wc2) {
                DateTemperaturePair pair = (DateTemperaturePair) wc1;
                DateTemperaturePair pair2 = (DateTemperaturePair) wc2;
                return pair.getYearMonth().compareTo(pair2.getYearMonth());
        }
}
```

Consider the natural key only for grouping

- Configure the grouping comparator using Job object:

```
job.setGroupingComparatorClass(YearMonthGroupingComparator.class);
```

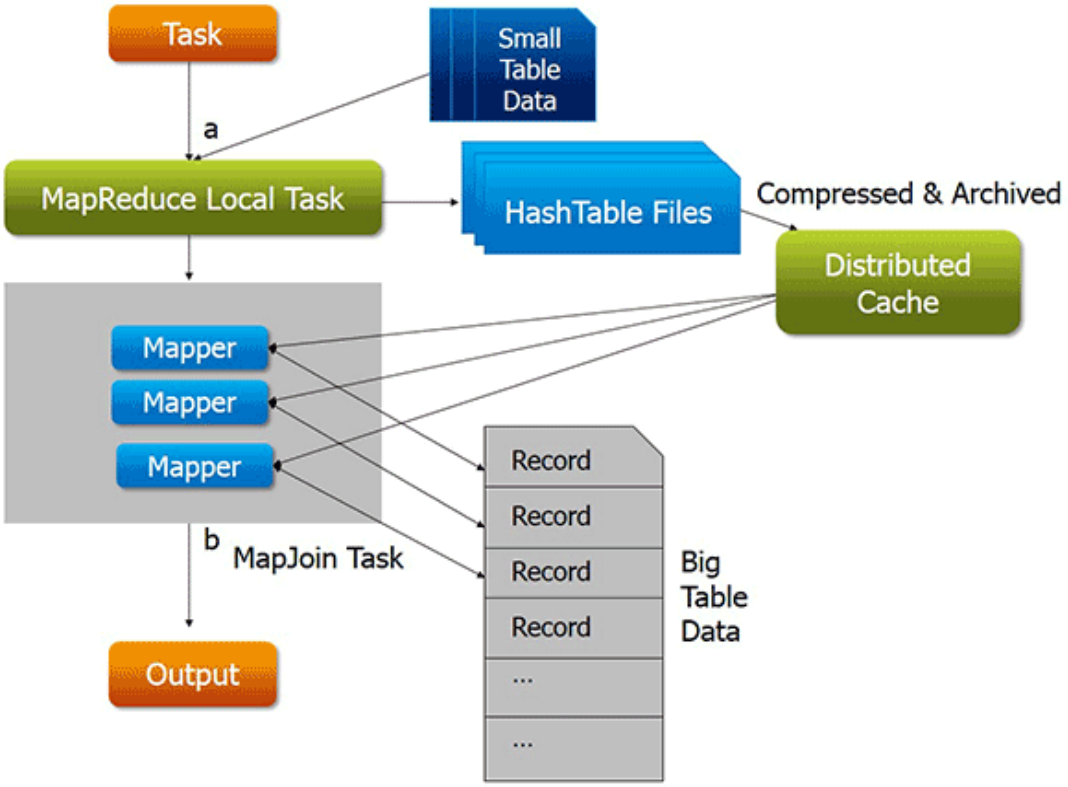
Data Flow


2000, 12, 04, 10
2000, 11, 01, 20
2000, 12, 02, -20
2000, 11, 07, 30
2000, 11, 24, -40
map()
((2000-12, 10), 10)
((2000-11, 20), 20)
((2000-12, -20), -20)
((2000-11, 30), 30)
((2000-11, -40), -40)
partition()
((2000-12, 10), 10)
((2000-11, 20), 20)
((2000-12, -20), -20)
((2000-11, 30),
((2000-11, -40),
group()
comparator()
reduce()
(2000-12), [10,-20]
(2000-11, [30, 20, -40]
An error here: this should be in another partition

Relational join

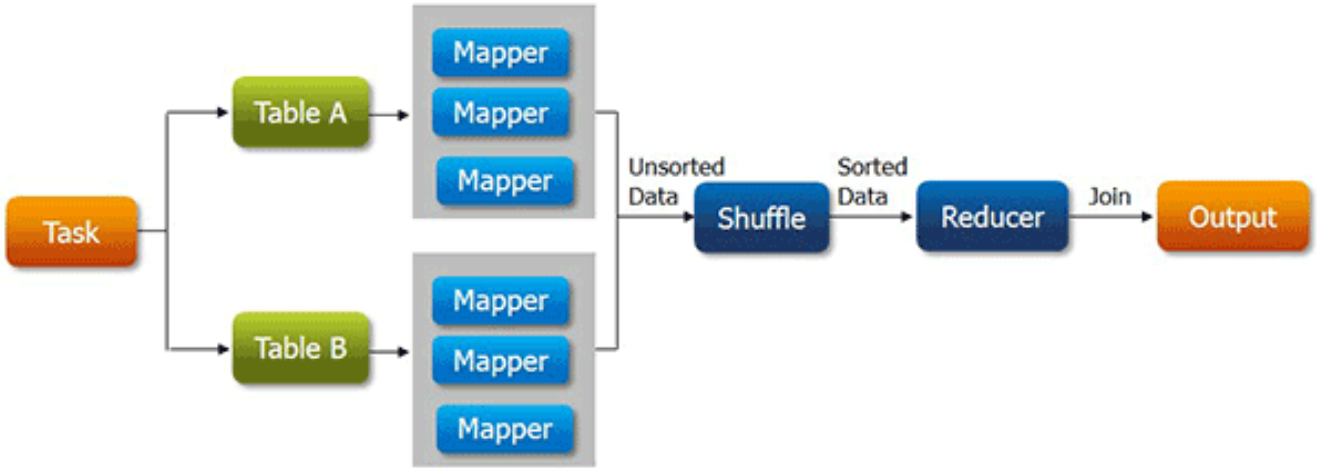
Joins in MapReduce

- **Map Side Join**
  - The join operation is performed in the map phase itself.
  - It is mandatory that the input to each map is partitioned and sorted according to the keys.



Joins in MapReduce

- **Reduce Side Join**
  - The reducer is responsible for performing the join operation.
  - The sorting and shuffling phase sends the values having identical keys to the same reducer and therefore, by default, the data is organized for us.
  - It is comparatively simple and easier to implement than the map side join.



Example: Relational data

- 2 relational tables: Employees, Department

Employees

Name	Age	Dept_Id
Alex	26	2
Ben	24	2
Sara	34	5

Department

Dept_Id	Name
5	Mkt
2	Eng
3	Sales

- SQL query of joins:

SELECT Employees.Name, Employees.Age, Department.Name
FROM Employees
INNER JOIN Department
ON Employees.Dept_Id=Department.Dept_Id

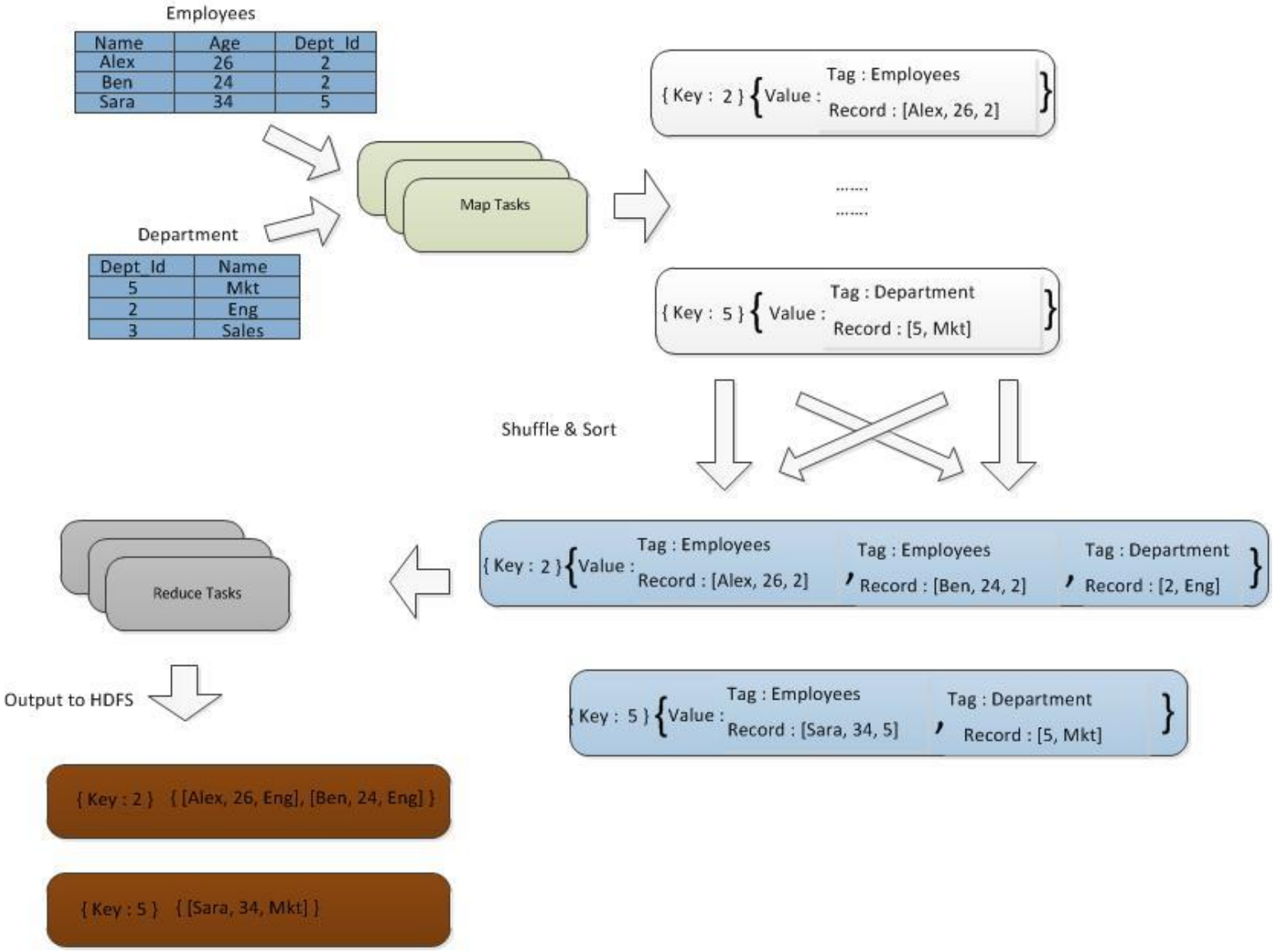
Reduce Side Join: Diagram

- **Map** is responsible for emitting the join predicate values along with the corresponding record from each table.
- Records having same **department id** in both tables will end up at on same reducer.
- **Reducers** then do the joining of records having same department id.
- It is also required to **tag each record** to indicate from which table the record originated so that joining happens between records of two tables.

Reduce Side Join: Steps

1. Mapper reads the input data which are to be combined based on common column or join key.
2. The mapper processes the input and adds a tag to the input to distinguish the input belonging from different sources or data sets or databases.
3. The mapper outputs the intermediate key-value pair where the key is nothing but the join key.
4. After the sorting and shuffling phase, a key and the list of values is generated for the reducer.
5. The reducer joins the values present in the list with the key to give the final aggregated output.

Reduce Side Join: Diagram



Reduce Side Join: Pseudo Code

```
map (K table, V rec) {
        dept_id = rec.Dept_Id
        tagged_rec.tag = table
        tagged_rec.rec = rec
        emit(dept_id, tagged_rec)
}

reduce (K dept_id, list<tagged_rec> tagged_recs)  {
        for (tagged_rec : tagged_recs) {
                for (tagged_rec1 : taagged_recs) {
                        if (tagged_rec.tag != tagged_rec1.tag) {
                        joined_rec = join(tagged_rec, tagged_rec1)
                }
                emit (tagged_rec.rec.Dept_Id, joined_rec)
        }
 }
```

Map Side Join

- Aka. Replicated Join
- One relation has to fit in to memory.
- The smaller table is replicated to each node (using Distributed Cache) and loaded to the memory.
- The join happens at map side without reducer involvement, which significantly speeds up the process (since this avoids shuffling all data across the network even-though most of the records not matching are later dropped).

Map Side Join: Pseudo Code

```
map (K table, V rec) {
        // Get smaller table records having this Dept_Id
        list recs = lookup(rec.Dept_Id)
        for (small_table_rec : recs) {
                joined_rec = join (small_table_rec, rec)
        }
        emit (rec.Dept_id, joined_rec)
}
```

Map Side Join: Filtering

- If the smaller table doesn't fit the memory it may be possible to **prune the contents** of it if filtering expression has been specified in the query.
- Here a smaller data set can be derived from Department table by filtering out records having department names other than "Eng". Now it may be possible to do replicated map side join with this smaller data set.
- SQL Query:

```
SELECT Employees.Name, Employees.Age, Department.Name
FROM Employees INNER JOIN Department
ON Employees.Dept_Id=Department.Dept_Id
WHERE Department.Name="Eng"
```

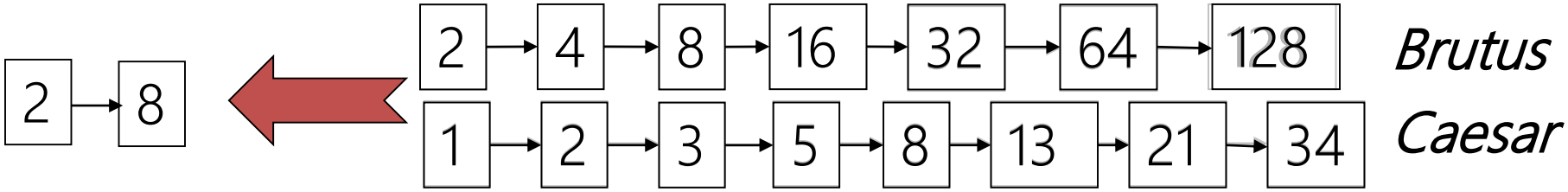
Search Engine

Search Engine

- Information retrieval (IR)
  - Focus on textual information (= text/document retrieval)
  - Other possibilities include image, video, music, …
- **Boolean** text retrieval
  - Each document or query is treated as a "bag" of words or terms. Word sequence is not considered
  - Query terms are combined logically using the Boolean operators AND, OR, and NOT.
    - E.g., ((data AND mining) AND (NOT text))
  - Retrieval
    - Given a Boolean query, the system retrieves every document that makes the query logically true.
    - Exact match
  - The retrieval results are usually quite **poor** because *term frequency is not considered* and *results are not ranked*

Boolean Query Processing: AND

- Consider processing the query: ***Brutus*** *AND* ***Caesar***
  - Locate ***Brutus*** in the Dictionary;
    - Retrieve its postings.
  - Locate ***Caesar*** in the Dictionary;
    - Retrieve its postings.
  - "Merge" the two postings:
    - Walk through the two postings simultaneously, in time linear in the total number of postings entries



If the list lengths are *x* and *y*, the merge takes O(*x*+*y*) operations.

Crucial: postings sorted by docID.

Search Using Inverted Index

- Given a query *q*, search processing has the following steps:
  - Step 1 (**Vocabulary search**): find each term/word in q in the inverted index.
  - Step 2 (**Results merging**): Merge results to find documents that contain all or some of the words/terms in *q*.
  - Step 3 (**Rank score computation**): To rank the resulting documents/pages, using:
    - content-based ranking
    - link-based ranking
    - Not used in Boolean retrieval

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…

Search Engine

Inverted Index

Inverted Index

- The inverted index of a document collection is basically a **data structure** that
  - attaches each distinctive term with a list of all documents that contains the term
  - The documents containing a term are sorted in the list

- In retrieval, it takes constant time to find the documents that contains a query term
- Multiple query terms are also easy handle as we will see soon

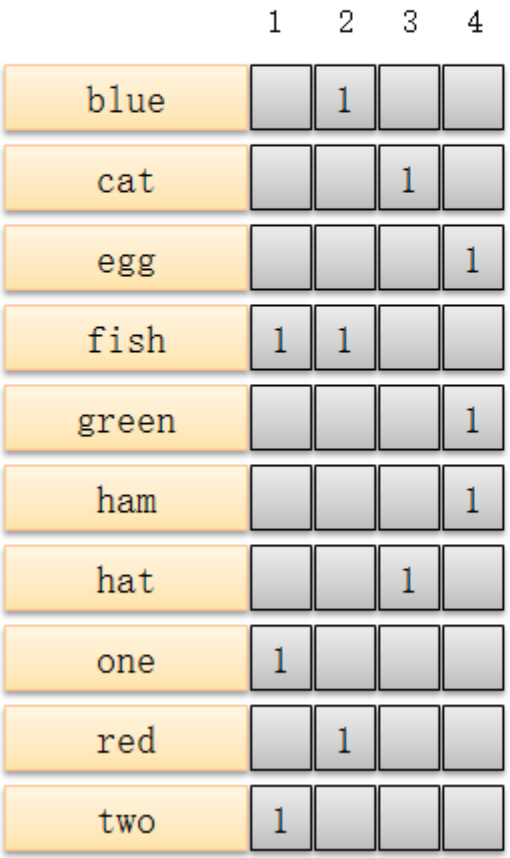
Inverted Index

Doc 1
one fish, two fish

Doc 2
red fish, blue fish

Doc 3
cat in the hat

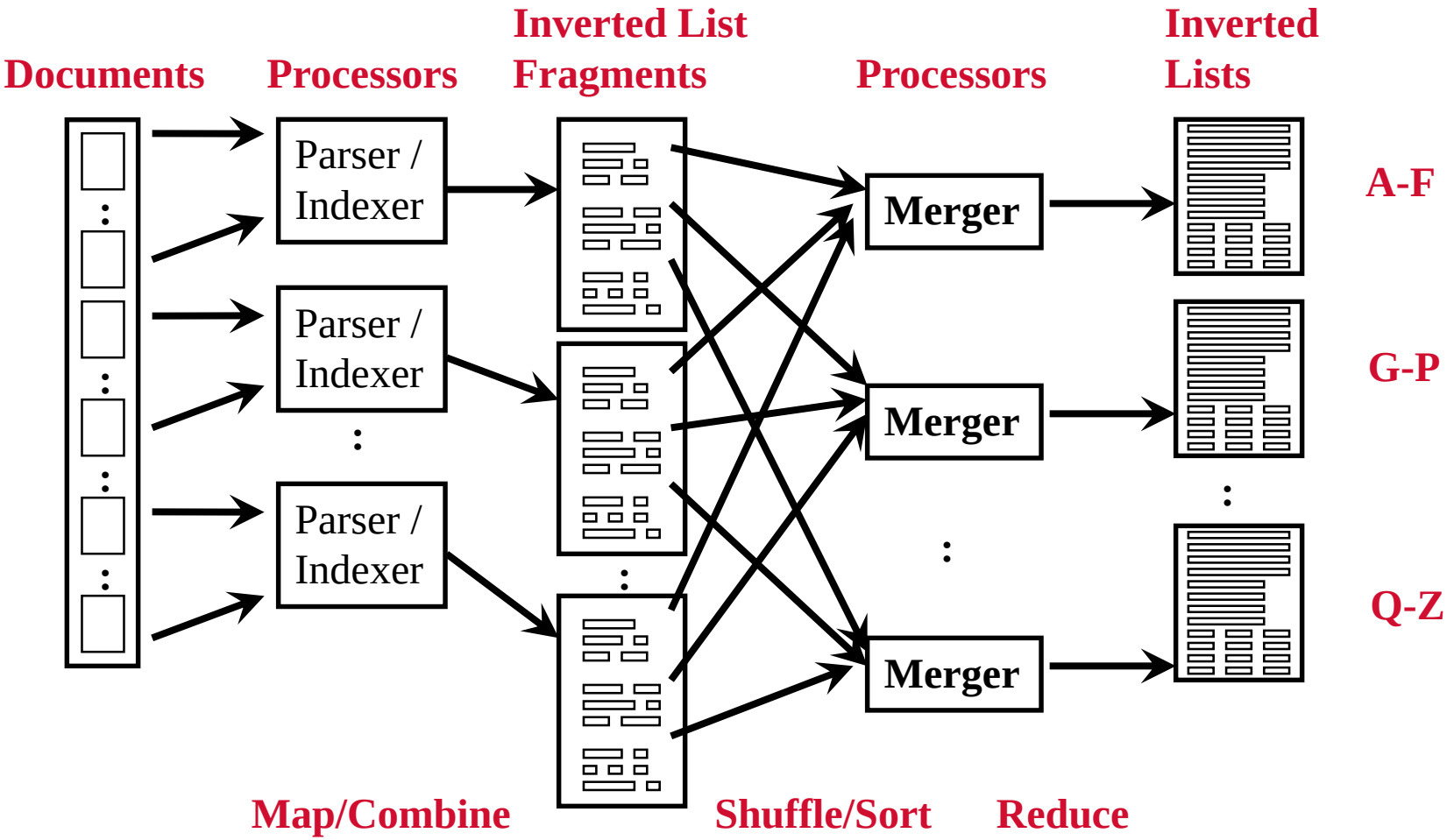
Doc 4
green eggs and ham



MapReduce: Index Construction

- Input: documents: (*docid*, *doc*), ..
- Output: (term, [*docid*, *docid*, …])
  - E.g., (long, [1, 23, 49, 127, …])
    - The *docid* are sorted !! (used in query phase)
  - *docid* is an internal document id, e.g., a unique integer. Not an external document id such as a URL
- How to do it in MapReduce?

Using MR to Construct Indexes



MapReduce Implementation

- A simple approach:
  - Each Map task is a document parser
    - Input: A stream of documents
      - (1, long ago …), (2, once upon …)
    - Output: A stream of (term, docid) tuples
      - (long, 1) (ago, 1) … (once, 2) (upon, 2) …
  - Reducers convert streams of keys into streams of inverted lists
    - Input: (long, [1, 127, 49, 23, …])
    - The reducer sorts the values for a key and builds an inverted list
      - Longest inverted list must fit in memory
    - Output: (long, [1, 23, 49, 127, …])
- Problems?
  - Inefficient
  - *docid*s are sorted in reducers
  - Secondary sort?

Search Engine

Term Weighting

Ranked Text Retrieval

- Order documents by how likely they are to be relevant
  - Estimate relevance(q, d_i)
  - 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, d_i) with sim(q, d_i)
  - Compute similarity of vector representations

- Vector space model, cosine similarity, language models, …

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?
  - TF: Term frequency (local)
  - IDF: Inverse document frequency (global)

TF-IDF

- Term Frequency – Inverse Document Frequency
  - Relevant to text processing
  - Common web analysis algorithm

$$\mathrm{tf_i} = \frac{n_i}{\sum_k n_k}$$

$$\mathrm{idf_i} = \log \frac{|D|}{|\{d : t_i \in d\}|}$$

$$\mathrm{tfidf} = \mathrm{tf} \cdot \mathrm{idf}$$

- $|D|$: total number of documents in the corpus
- $|\{d : t_i \in d\}|$: number of documents where the term t_i appears

Information We Need

1. Number of times term X appears in a given document
2. Number of terms in each document
3. Number of documents X appears in
4. Total number of documents

Job 1: Word Frequency in Doc

- Mapper
  - Input: (docname, contents)
  - Output: ((word, docname), 1)
- Reducer
  - Sums counts for word in document
  - Outputs ((word, docname), *n*)
- Combiner is same as Reducer

Job 2: Word Counts For Docs

- Mapper
  - Input: ((word, docname), *n*)
  - Output: (docname, (word, *n)*)
- Reducer
  - Sums frequency of individual *n*'s in same doc
  - Feeds original data through
  - Outputs ((word, docname), (*n*, *N)*)

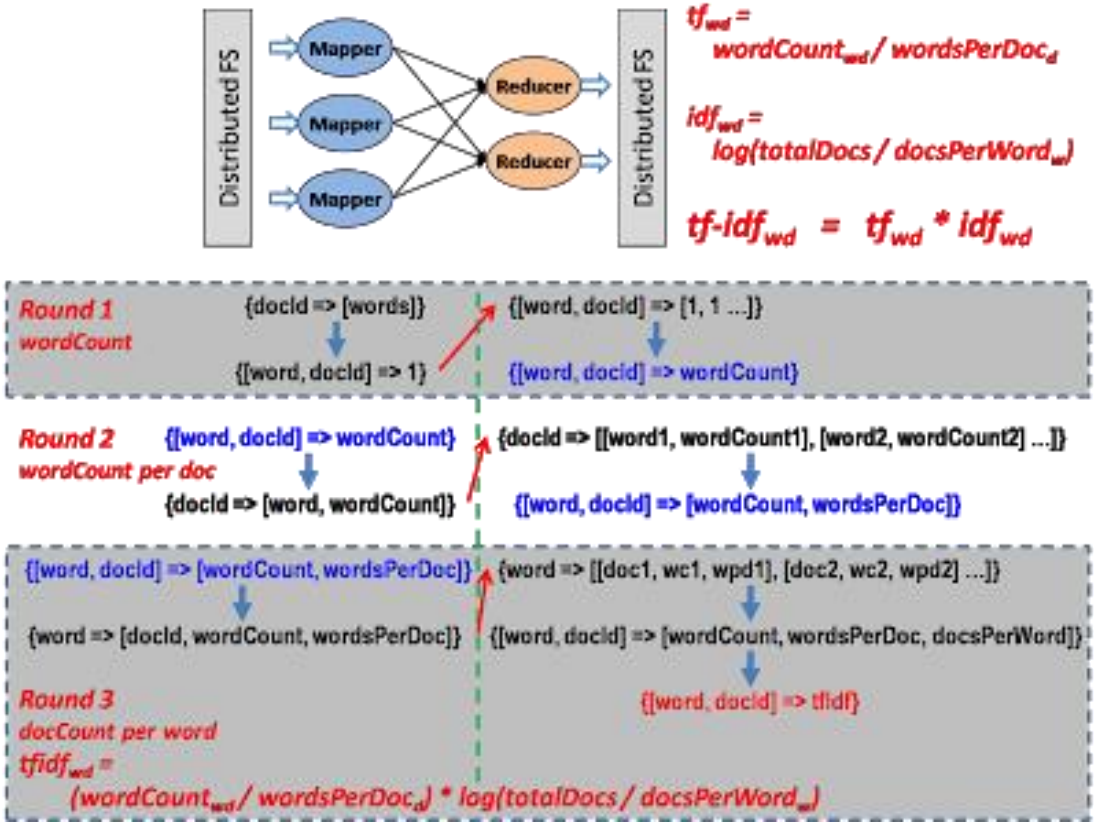
Job 3: Word Frequency In Corpus

- Mapper
  - Input: ((word, docname), (*n, N)*)
  - Output: (word, (docname, *n*, *N*, 1))
- Reducer
  - Sums counts for word in corpus
  - Outputs ((word, docname), (*n*, *N*, *m*))

- The weight of term *word* in *docname* is:

  $tfidf_{word} = (n/N) \times \log(|D|/m)$

Solution Overview



Working At Scale

- Buffering (doc, *n*, *N*) counts while summing 1's into *m* may not fit in memory
  - How many documents does the word "the" occur in?
- Possible solutions
  - Ignore very-high-frequency words
  - Write out intermediate data to a file
  - Use another MR pass

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

Search Engine

Index Construction

MapReduce: Index Construction

- Input: documents: (*docid*, *doc*), ..
- Output: (*t*, [(*docid*, w_t), (*docid*, *w*), …])
  - w_t represents the term weight of *t* in *docid*
  - E.g., (long, [(1, 0.5), (23, 0.2), (49, 0.3), (127,0.4), …])
    - The docid are sorted !! (used in query phase)
- How this problem differs from the previous one?
  - TF computing
    - Easy. Can be done within the mapper
  - IDF computing
    - Known only after all documents containing a term *t* processed
  - Input and output of map and reduce?

Inverted Index: TF-IDF

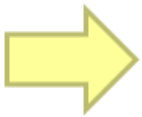
Doc 1 one fish, two fish

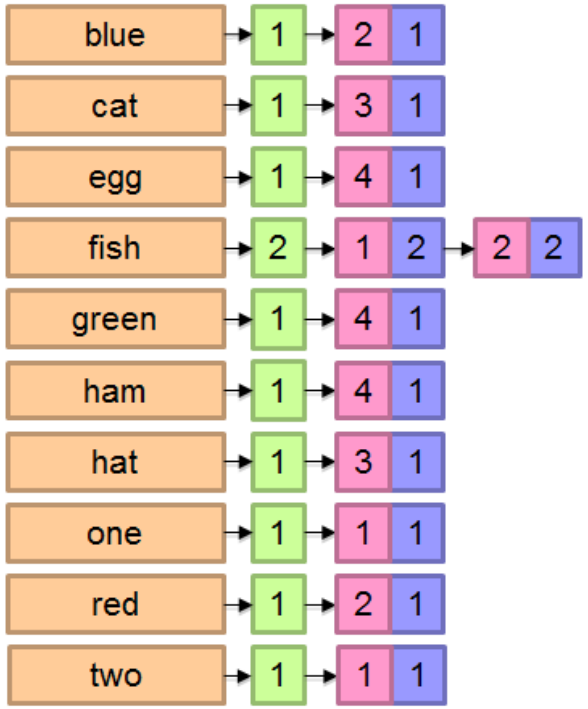
Doc 2 red fish, blue fish

Doc 3 cat in the hat

Doc 4 green eggs and ham

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

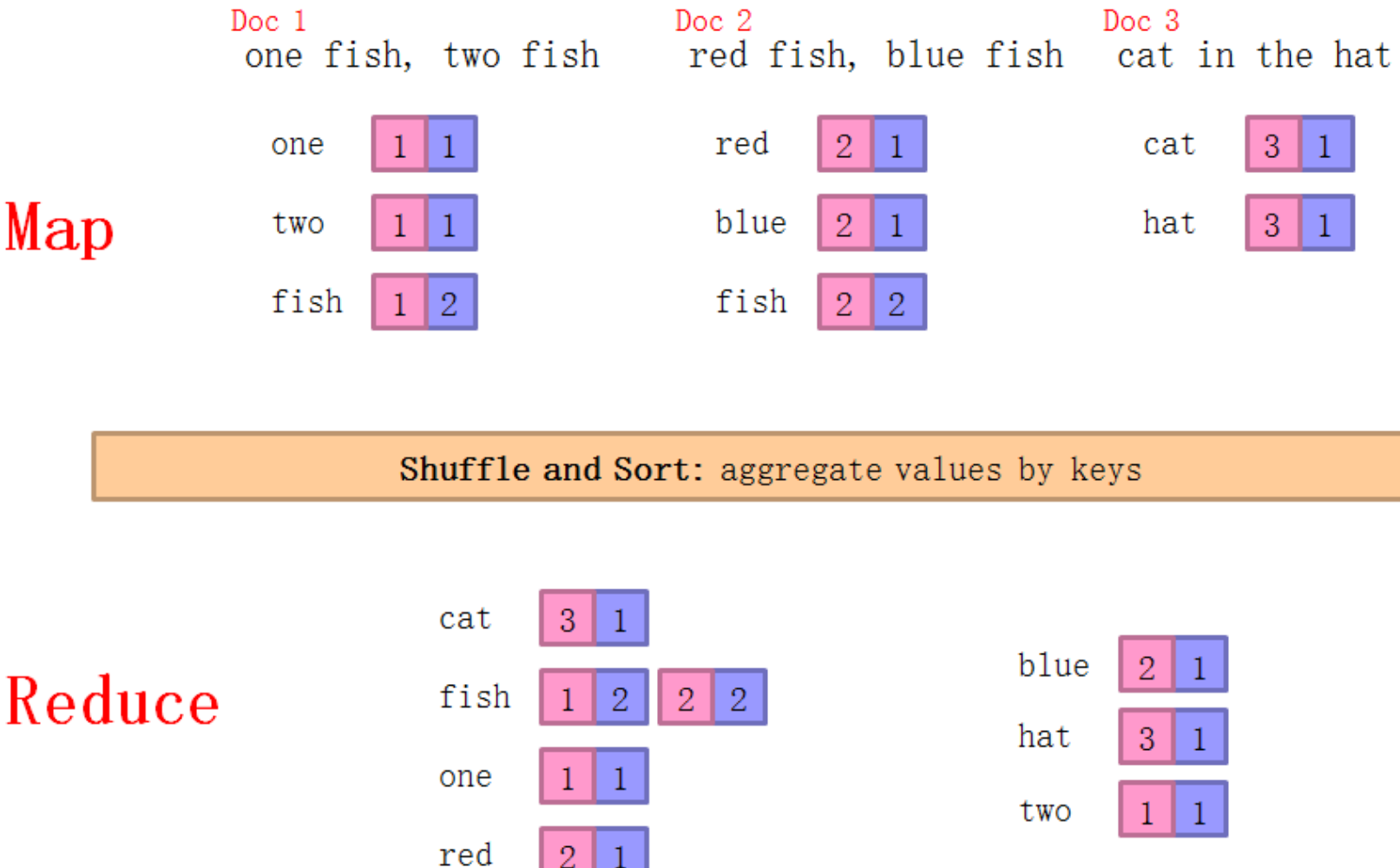




MapReduce: Index Construction

- A simple approach:
  - Each Map task is a document parser
    - Input: A stream of documents
      - (1, long ago …), (2, once upon …)
    - Output: A stream of (*term*, [*docid*, *tf*]) tuples
      - (long, [1,1]) (ago, [1,1]) … (once, [2,1]) (upon, [2,1]) …
  - Reducers convert streams of keys into streams of inverted lists
    - Input: (long, {[1,1], [127,2], [49,1], [23,3] …})
    - The reducer sorts the values for a key and builds an inverted list
      - Compute TF and IDF in reducer!
    - Output: (long, [(1, 0.5), (23, 0.2), (49, 0.3), (127,0.4), …])

MapReduce: Index Construction



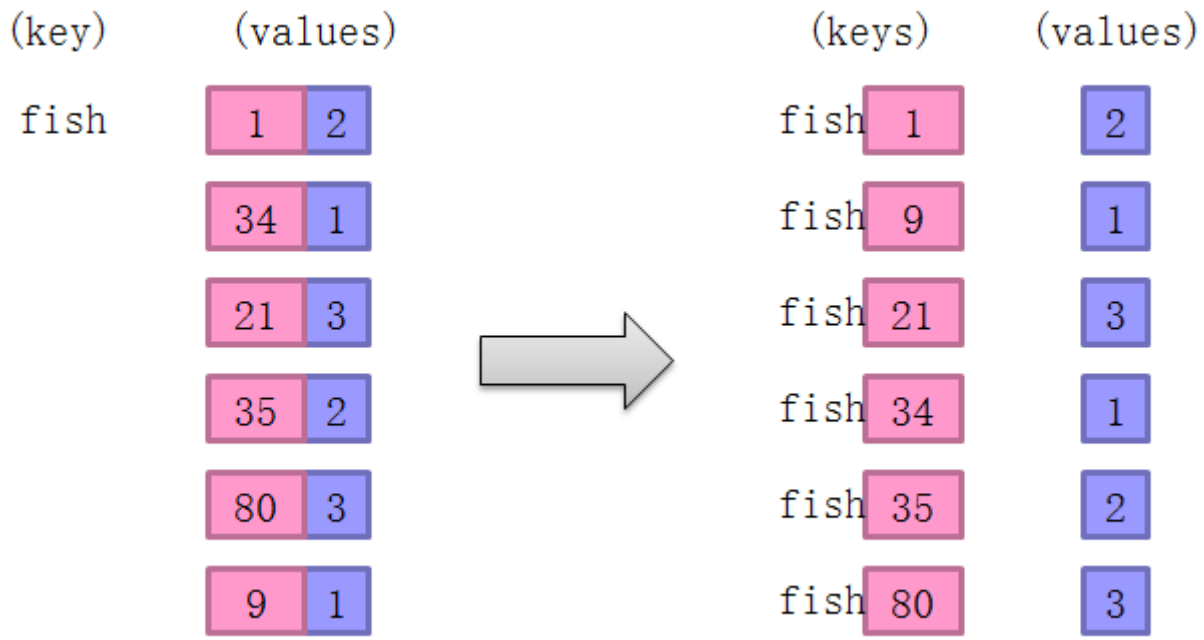
MapReduce: Index Construction

- Inefficient: terms as keys, postings as values
  - docids are sorted in reducers
  - IDF can be computed only after all relevant documents received
  - Reducers must buffer all postings associated with key (to sort)
    - What if we run out of memory to buffer postings?
  - Improvement?

The First Improvement

- How to make Hadoop sort the docid, instead of doing it in reducers?
- Design pattern: value-to-key conversion, secondary sort

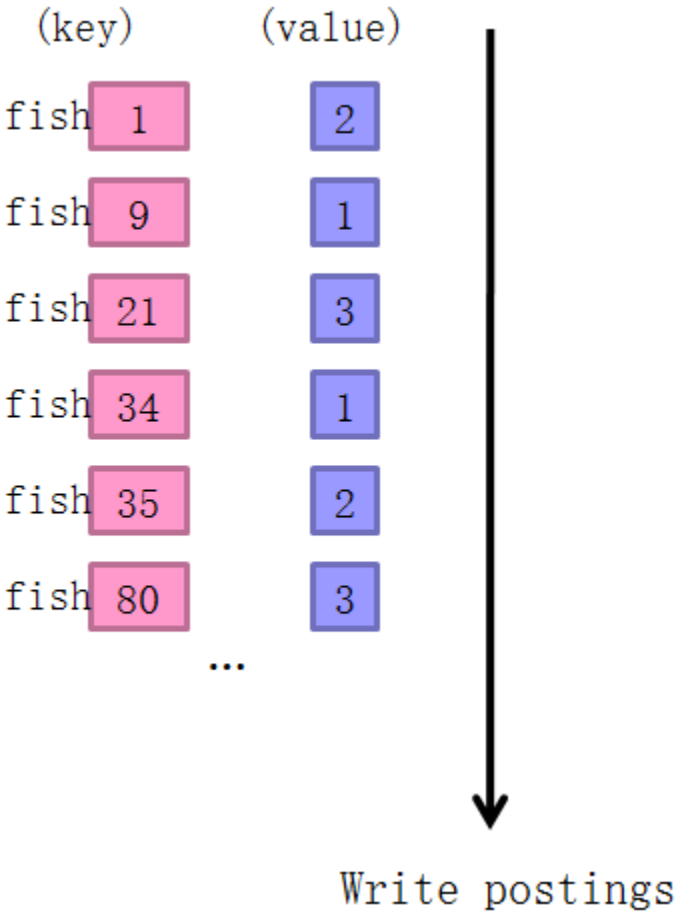
- Mapper output a stream of ([*term*, *docid*], *tf*) tuples



- Remember: you must implement a partitioner on term!

The Second Improvement

- How to avoid buffering all postings associated with



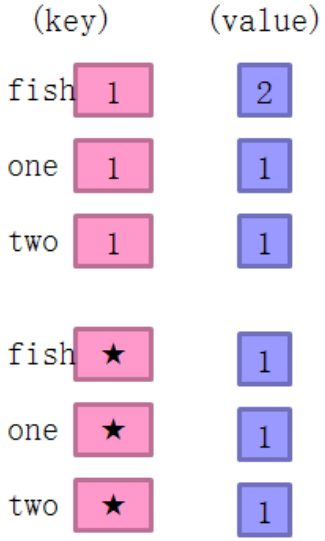
We'd like to store the DF at the front of the postings list

But we don't know the DF until we've seen all postings!

Sound familiar?
Design patter: Order inversion

The Second Improvement

- 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!

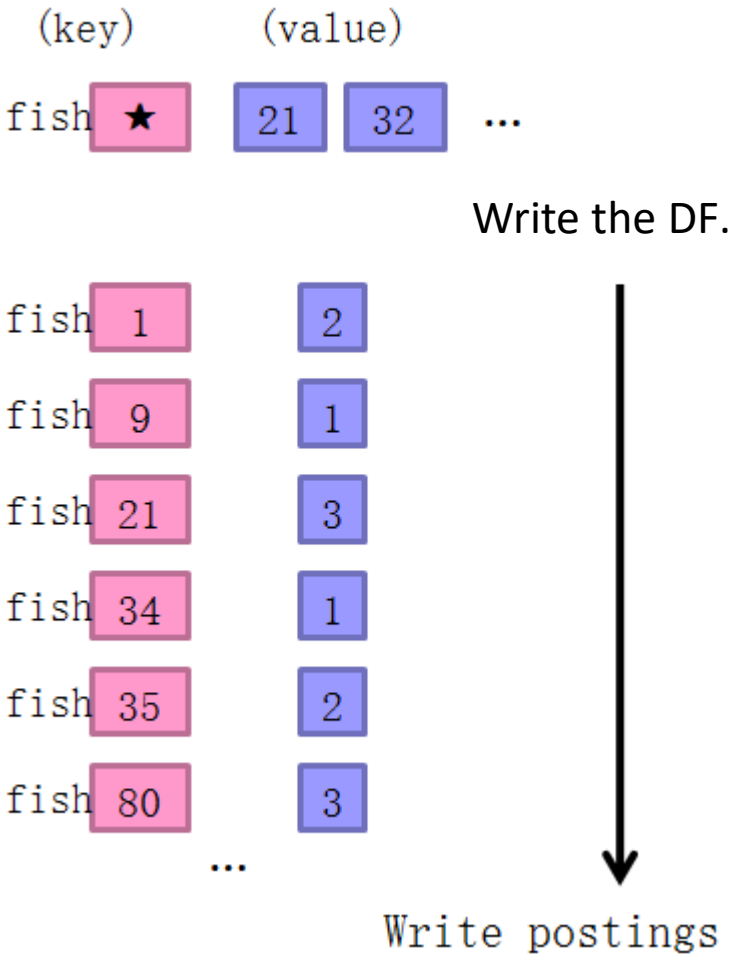


Emit normal key-value pairs…

Emit “special” key-value pairs to keep track of df…

Doc1: one fish, two fish

The Second Improvement



First, compute the DF by summing contributions from all “special” key-value pair…

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

Graph (BFS)

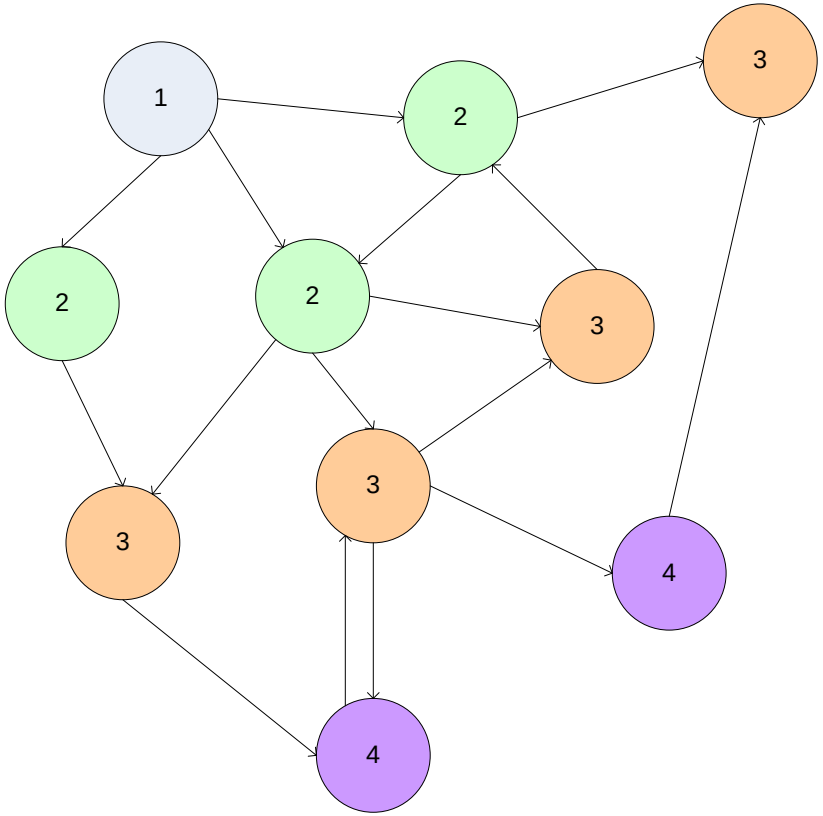
BFS: Motivating Concepts

- Performing computation on a graph data structure requires processing at each node
- Each node contains node-specific data as well as links (edges) to other nodes
- Computation must traverse the graph and perform the computation step

- *How do we traverse a graph in MapReduce? How do we represent the graph for this?*

Breadth-First Search

- Breadth-First Search is an *iterated* algorithm over graphs
- Frontier advances from origin by one level with each pass



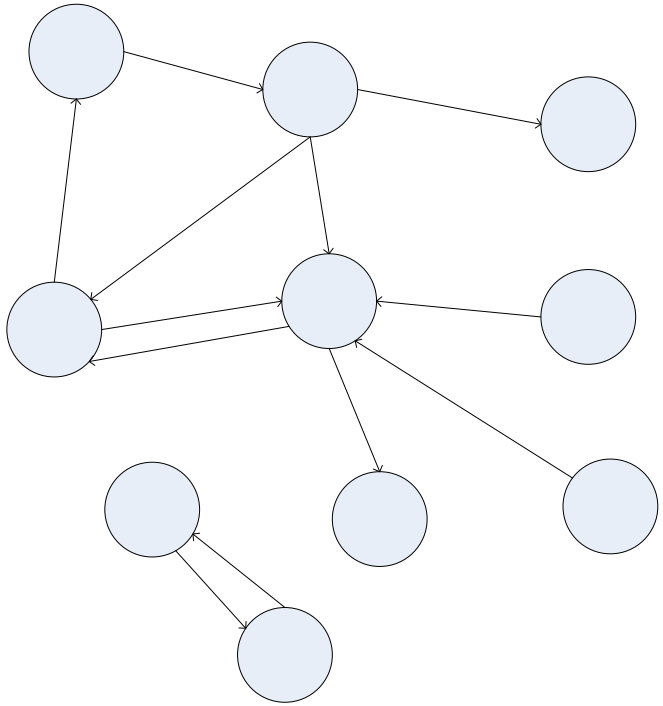
Breadth-First Search & MapReduce

- **Problem 1**: This doesn't “fit” into MapReduce
- Solution: Iterated passes through MapReduce – map some nodes, result includes additional nodes which are fed into successive MapReduce passes

- **Problem 2**: Sending the entire graph to a map task (or hundreds/thousands of map tasks) involves an enormous amount of memory
- Solution: Carefully consider how we represent graphs

Graph Representations

- The most straightforward representation of graphs uses references from each node to its neighbours



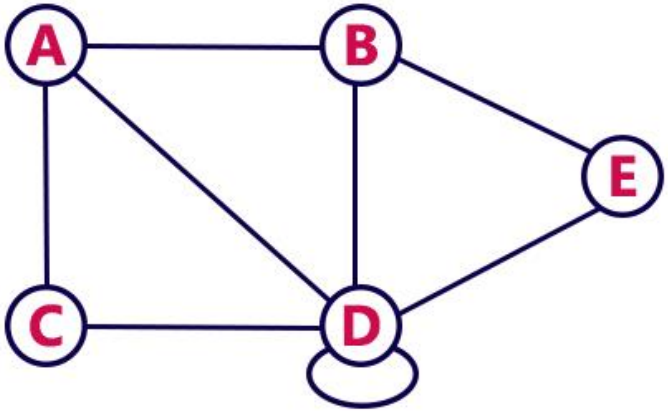
Direct References

- Structure is inherent to object
- Iteration requires linked list “threaded through” graph
- Requires common view of shared memory (synchronization!)
- Not easily serializable

```
class GraphNode
{
  Object data;
  Vector<GraphNode> out_edges;
  GraphNode iter_next;
}
```

Adjacency Matrix

- Another classic graph representation. M[*i*][*j*]= '1' implies a link from node *i to j.*
- Naturally encapsulates iteration over nodes



Sparse Matrix Representation

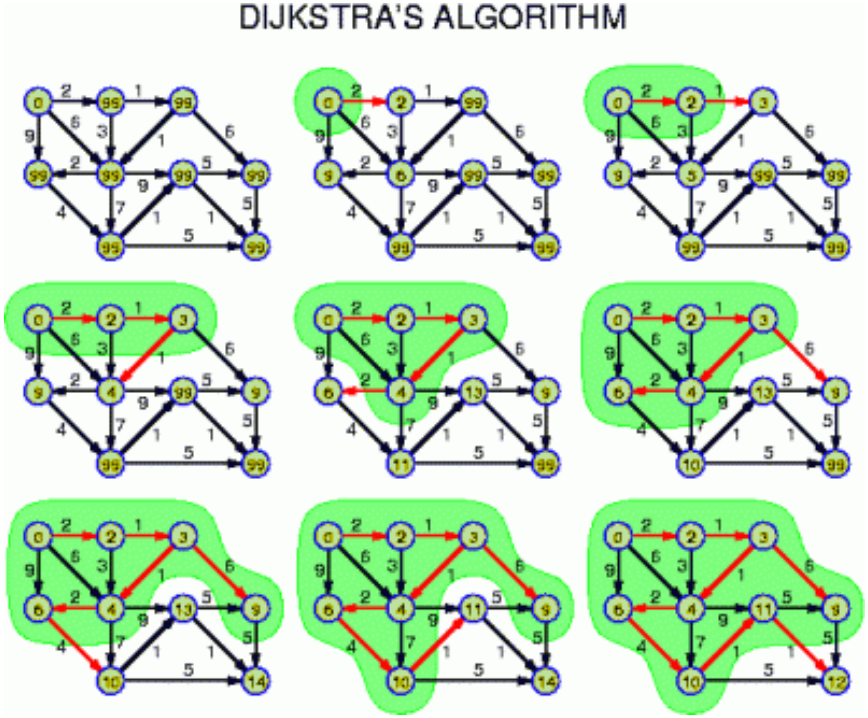
- Adjacency matrix for most large graphs (e.g., the web) will be overwhelmingly full of zeros.
- Each row of the graph is absurdly long
- Sparse matrices only include non-zero elements

1: (3, 1), (18, 1), (200, 1)
2: (6, 1), (12, 1), (80, 1), (400, 1)
3: (1, 1), (14, 1)
…

1: 3, 18, 200
2: 6, 12, 80, 400
3: 1, 14
…

Finding the Shortest Path

- A common graph search application is finding the shortest path from a start node to one or more target nodes
- Commonly done on a single machine with *Dijkstra's Algorithm*
- Can we use BFS to find the shortest path via MapReduce?



This is called the single-source shortest path problem. (a.k.a. SSSP)

Intuition

- We can define the solution to this problem inductively:
  - DistanceTo(*startNode*) = 0
  - For all nodes *n* directly reachable from *startNode*, DistanceTo(*n*) = 1
  - For all nodes *n* reachable from other set of nodes *S*,

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

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 = ∞
- A map task receives a node *n as* a key, and *(d, points-to)* as its value
  - *d* is the distance to the node from the start
  - *points-to* is a list of nodes reachable from *n*
  - $\forall p \in$ points-to, emit (*p*, *d*+1)
- Sort/Shuffle
  - Groups distances by reachable nodes
- Reduce task gathers possible distances to a given *p* and selects the minimum one

What This Gives Us

- Each MapReduce iteration can advance 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
- To perform the whole BFS, a non-MapReduce component then feeds the output of this step back into the MapReduce task for another iteration
  - Problem: Where'd the points-to list go?
  - Solution: Mapper emits (*n*, points-to) as well

Blow-up and Termination

- This algorithm starts from one node
- Subsequent iterations include many more nodes of the graph as frontier advances
- Does this ever terminate?
  - Yes! Eventually, routes between nodes will stop being discovered and no better distances will be found. When distance is the same, we stop
  - Mapper should emit *(n, D)* to ensure that "current distance" is carried into the reducer

Adding weights

- Weighted-edge shortest path is more useful than cost==1 approach
- Simple change: points-to list in map task includes a weight *w* for each pointed-to node
  - emit (p, $D+w_p$) instead of (p, D+1) for each node p
  - Works for positive-weighted graph

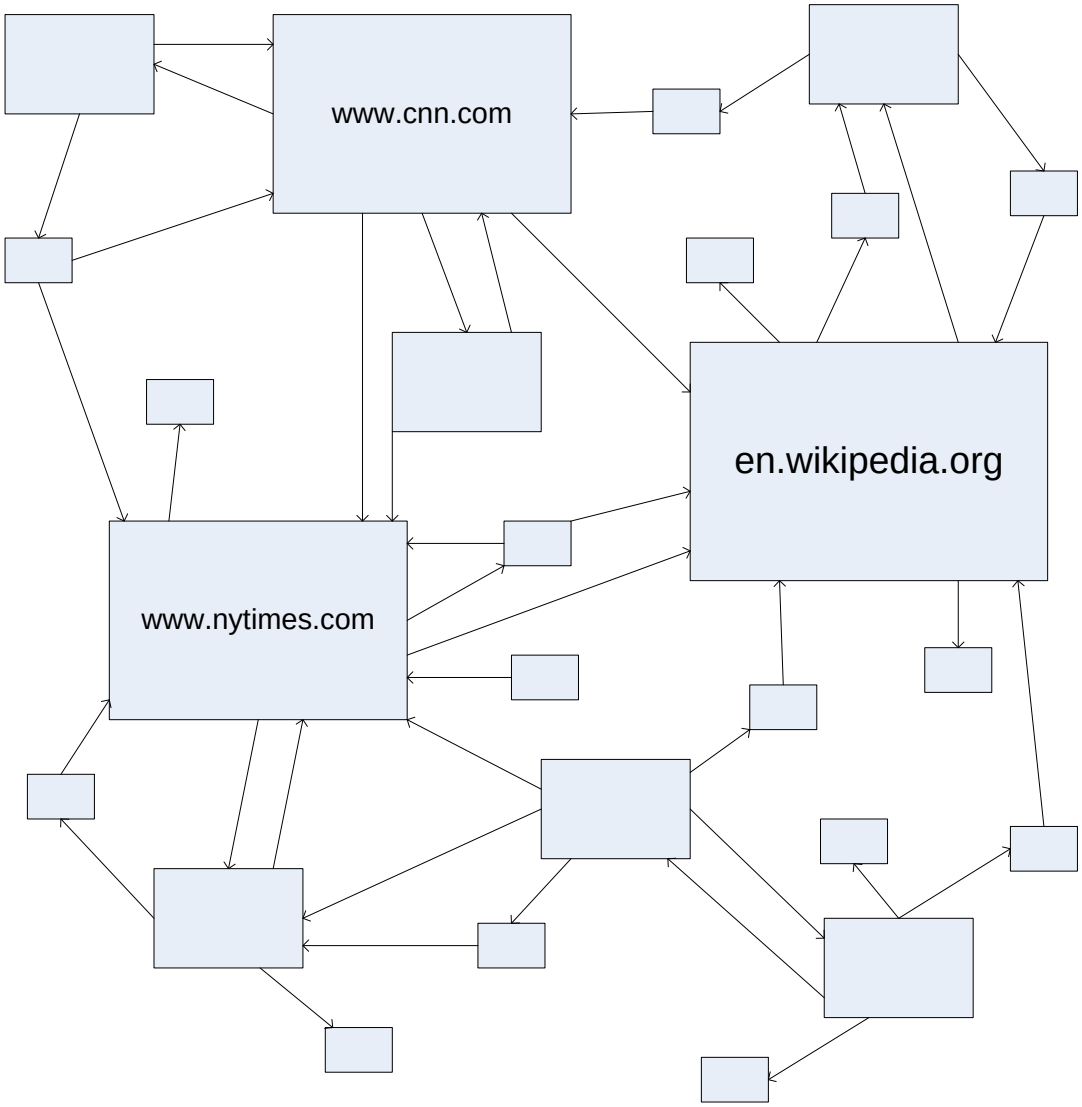
PageRank

Intuition

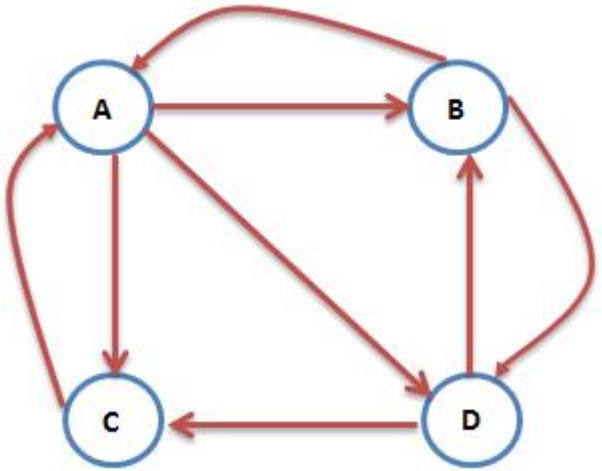
- Named after **Larry Page**

- Works by counting the **number** and **quality** of links to a page to determine a rough estimate of **how important** the website is
- Intuitive description: a page has high rank if the sum of the ranks of its backlinks is high. This covers both the case:
  - When a page has many backlinks
  - When a page has a few highly ranked backlinks

Visually



PageRank in Formula



$$PR(A) = PR(B) + PR(C)$$

$$PR(A) = \frac{PR(B)}{2} + \frac{PR(C)}{1}$$

$$PR(A) = \frac{PR(B)}{2} + \frac{PR(C)}{4}$$

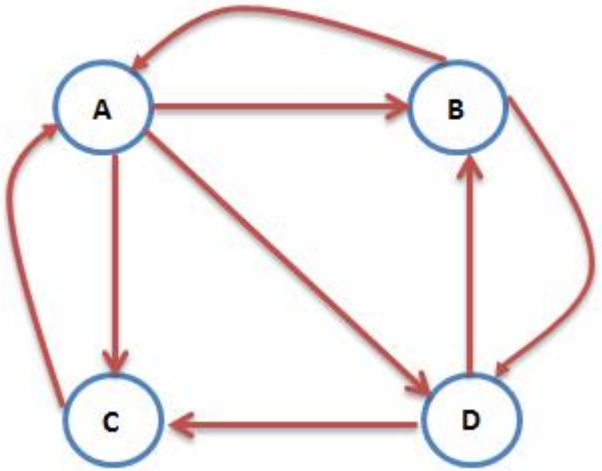
$$PR(A) = \alpha\left(\frac{PR(B)}{2}\right) + \frac{(1-\alpha)}{4}$$

Formula:

$$PR(p_i) = \alpha \sum_{p_j \in M_{p_i}} \frac{PR(pj)}{L(pj)} + \frac{(1-\alpha)}{N}$$

where M_{p_i} is the out-link set of page p_i, $L(p_j)$ is the number of out-links of p_j, N is the total number of web pages, α is usually set to 0.85.

PageRank in Matrix



$$M = \begin{bmatrix} 0 & 1/2 & 1 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}$$

$$V_1 = MV_0 = \begin{bmatrix} 0 & 1/2 & 1 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix} = \begin{bmatrix} 9/24 \\ 5/24 \\ 5/24 \\ 5/24 \end{bmatrix}$$

Multiply the PageRank vector **v** with transition matrix **M** to get the next moment's probability distribution **X**:

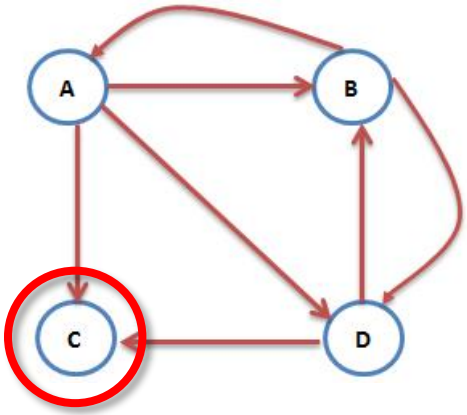
$$\begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix} \begin{bmatrix} 9/24 \\ 5/24 \\ 5/24 \\ 5/24 \end{bmatrix} \begin{bmatrix} 15/48 \\ 11/48 \\ 11/48 \\ 11/48 \end{bmatrix} \cdot \begin{bmatrix} 11/32 \\ 7/32 \\ 7/32 \\ 7/32 \end{bmatrix} \cdots \begin{bmatrix} 3/9 \\ 2/9 \\ 2/9 \\ 2/9 \end{bmatrix}$$

X = **Mv**

In the final state, PageRank will coverge and vector **X** will be the same as vector **V**.

PageRank in Matrix: Problems

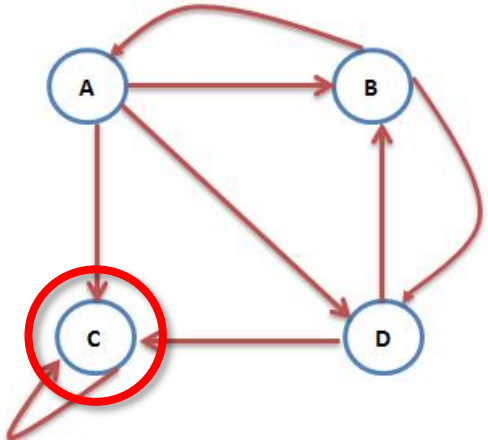
- Termination node



$$M = \begin{bmatrix} 0 & 1/2 & 0 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix} \begin{bmatrix} 3/24 \\ 5/24 \\ 5/24 \\ 5/24 \end{bmatrix} \begin{bmatrix} 5/48 \\ 7/48 \\ 7/48 \\ 7/48 \end{bmatrix} \begin{bmatrix} 21/288 \\ 31/288 \\ 31/288 \\ 31/288 \end{bmatrix} \cdots \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

- Spider Trap



$$M = \begin{bmatrix} 0 & 1/2 & 0 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 1 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}$$

$$\begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix} \begin{bmatrix} 3/24 \\ 5/24 \\ 11/24 \\ 5/24 \end{bmatrix} \begin{bmatrix} 5/48 \\ 7/48 \\ 29/48 \\ 7/48 \end{bmatrix} \begin{bmatrix} 21/288 \\ 31/288 \\ 205/288 \\ 31/288 \end{bmatrix} \cdots \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

PageRank in Matrix: Solution

- Set a random redirect probability: (1-α)
- The iteration equation is rewritten as:

$$V' = \alpha MV + (1-\alpha)e$$

$$V_1 = \alpha M V_0 + (1-\alpha)e = 0.8 \times \begin{bmatrix} 0 & 1/2 & 0 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 1 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix} + 0.2 \times \begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix} = \begin{bmatrix} 9/60 \\ 13/60 \\ 25/60 \\ 13/60 \end{bmatrix}$$

$$\begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix} \begin{bmatrix} 9/60 \\ 13/60 \\ 25/60 \\ 13/60 \end{bmatrix} \begin{bmatrix} 41/300 \\ 53/300 \\ 153/300 \\ 53/300 \end{bmatrix} \begin{bmatrix} 15/4500 \\ 707/4500 \\ 2543/4500 \\ 707/4500 \end{bmatrix} \cdots \begin{bmatrix} 15/148 \\ 19/148 \\ 95/148 \\ 19/148 \end{bmatrix}$$

Map & Reduce

- Map
  - Input
    - **Key** = URL of website
    - **Value** = PR, Outgoing links from that website
  - Output
    - **Key** = Outgoing link URL
    - **Value** = Original Website URL PR / Outgoing links number
- Reduce
  - Input
    - **Key** = Outgoing link URL
    - **Value** = All links to outgoing link URL
  - Output
    - **Key** = Outgoing link URL
    - **Value** = Newly computed PR, Outgoing links number from document of outgoing link URL

$$< y, \{x_1, \dots, xn\} >$$

$$\text{for } i = 1 \text{ to } n\text{: emit} < xi, \frac{PR(y)}{out(y)} >$$

$$< x, \left[\frac{PR(y_1)}{out(y_1)}, \dots, \frac{PR(y_m)}{out(ym)}\right] >$$

$$\text{compute: } PR(x) = \frac{1-\alpha}{N} + \alpha \sum_{y \to x} \frac{PR(y)}{out(y)}$$

$$\text{emit: } < xi, \frac{PR(x)}{out(x)} >$$

Finishing up

- A subsequent component determines whether convergence has been achieved (Fixed number of iterations? Comparison of key values?)
- If so, write out the PageRank lists - done!
- Otherwise, feed output of this MapReduce round into another MapReduce job iteration

End of Chapter 4