

25 YEARS ANNIVERSARY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY



HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Chapter 6

Batch processing - part 1

MapReduce

Data processing: MapReduce

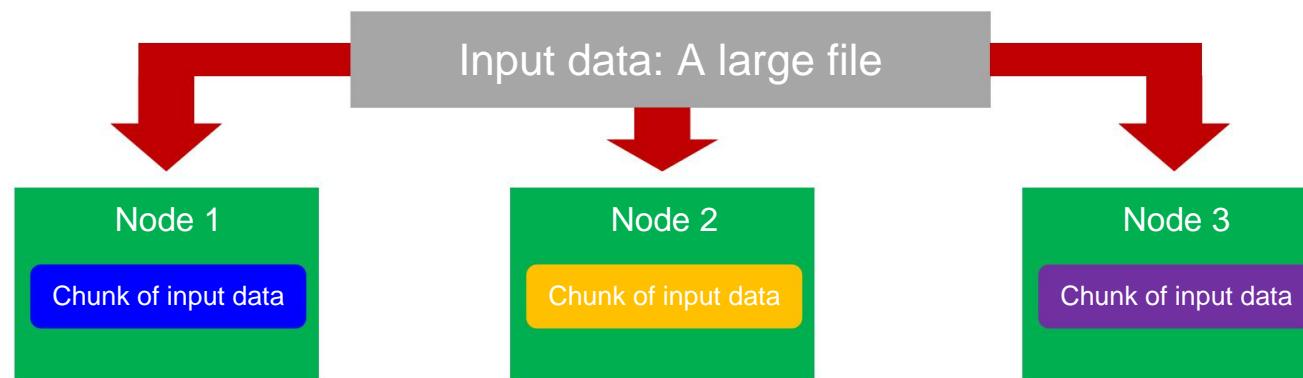
- MapReduce framework is the Hadoop default data processing engine
- MapReduce is a programming model for data processing
 - it is not a language, a style of data processing created by Google
- The beauty of MapReduce
 - Simplicity
 - Flexibility
 - Scalability

a MR job = {Isolated Tasks}n

- MapReduce divides the workload into multiple *independent tasks* and schedules them across cluster nodes
- A work performed by each task is done *in isolation* from one another for scalability reasons • The communication overhead required to keep the data on the nodes synchronized at all times will prevent the model from performing reliably and efficiently at large scale

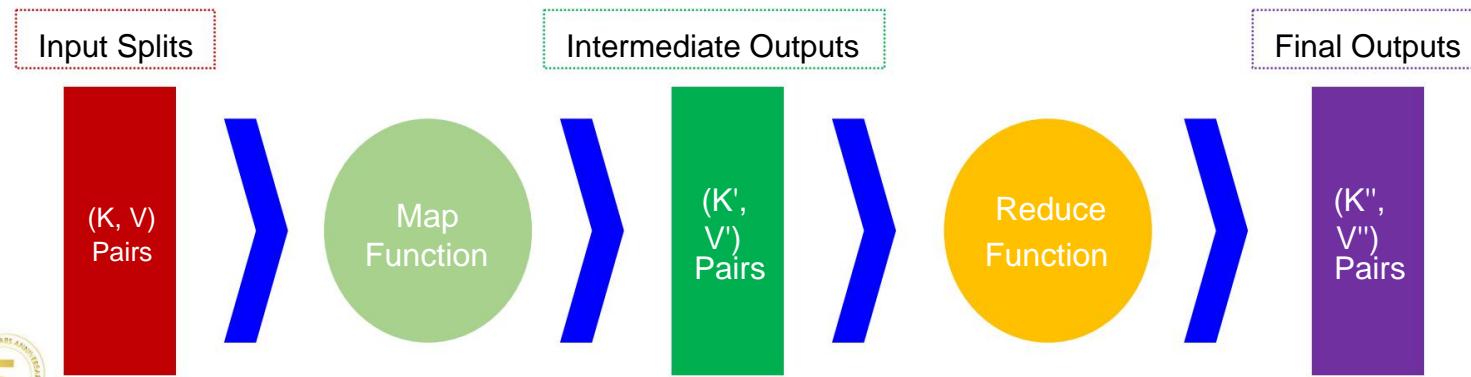
Data Distribution

- In a MapReduce cluster, data is usually managed by a distributed file systems (eg, HDFS)
- Move code to data and not data to code



Keys and Values

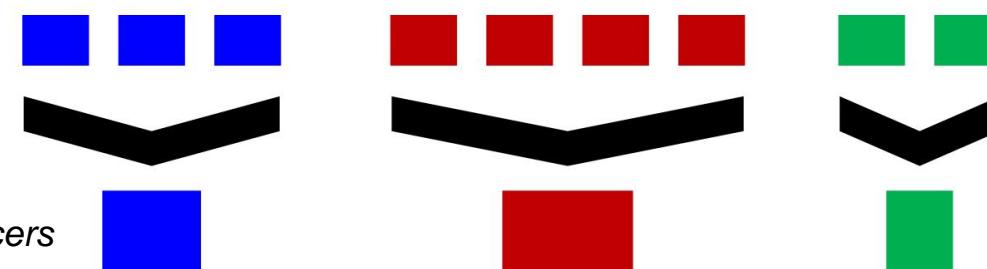
- The programmer in MapReduce has to specify two functions, the *map function* and the *reduce function* that implements the Mapper and the Reducer in a MapReduce program
- In MapReduce data elements are always structured as key-value (ie, (K, V)) pairs
- The map and reduce functions receive and *emit* (K, V) pairs



Partitions

- § A different subset of intermediate key space is assigned to each Reducer
- § These subsets are known as *partitions*

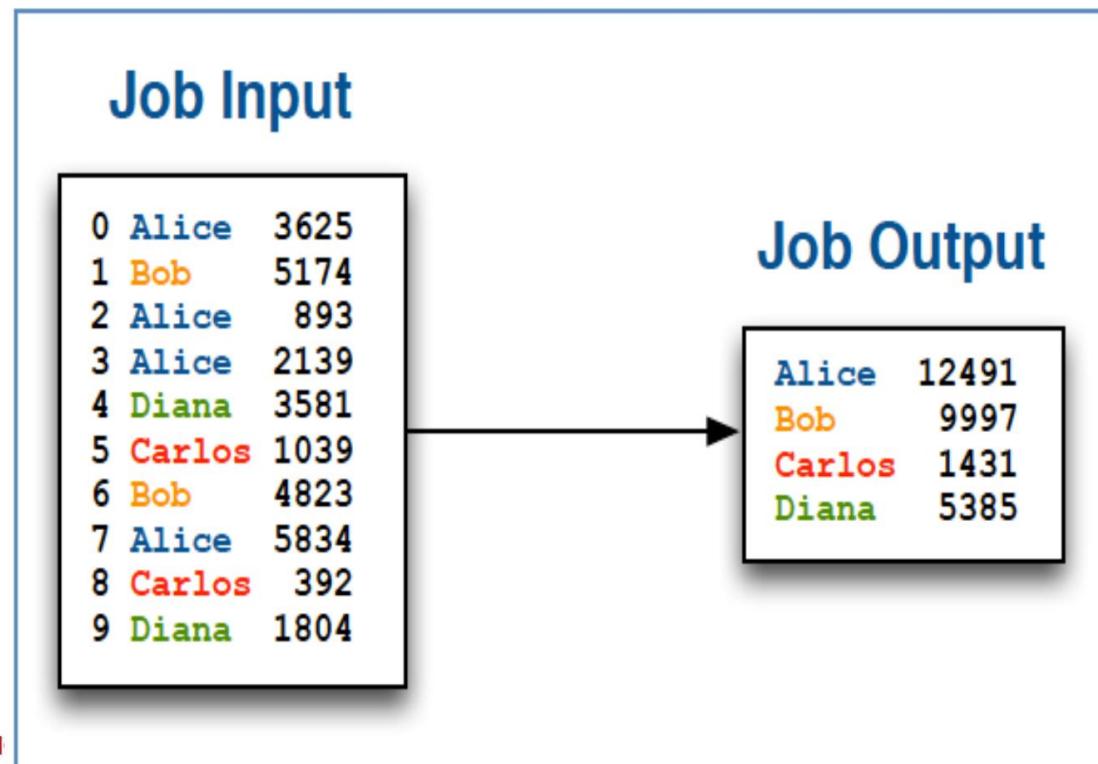
Different colors represent different keys (potentially) from different Mappers



Partitions are the input to Reducers

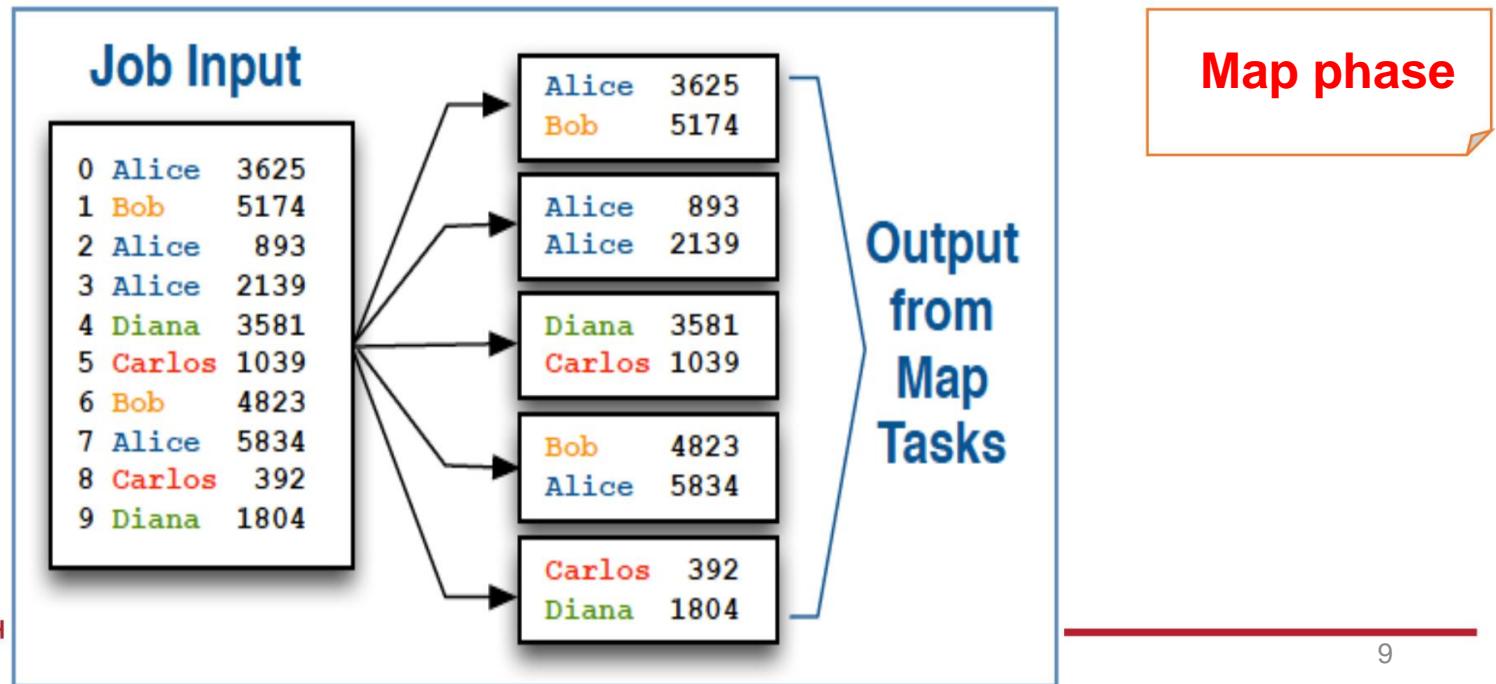
MapReduce example

- Input: text file containing order ID, employee name, and sale amount
- Output: sum of all sales per employee



Map phase

- Hadoop splits jobs into many individual map tasks
 - Number of map tasks is determined by the amount of input data
 - Each map task receives a portion of the overall job input to process
 - Mappers process one input record at a time
 - For each input record, they emit zero or more records as output
- In this case, the map task simply parses the input record
 - And then emits the name and price fields for each as output



Shuffle & sort

- Hadoop automatically sorts and merges output from all map tasks •

This intermediate process is known as the **shuffle and sort** • The result is supplied to reduce tasks

Map Task #1 Output

Alice	3625
Bob	5174

Map Task #2 Output

Alice	893
Alice	2139

Map Task #3 Output

Diana	3581
Carlos	1039

Map Task #4 Output

Bob	4823
Alice	5834

Map Task #5 Output

Carlos	392
Diana	1804

Input to Reduce Task #1

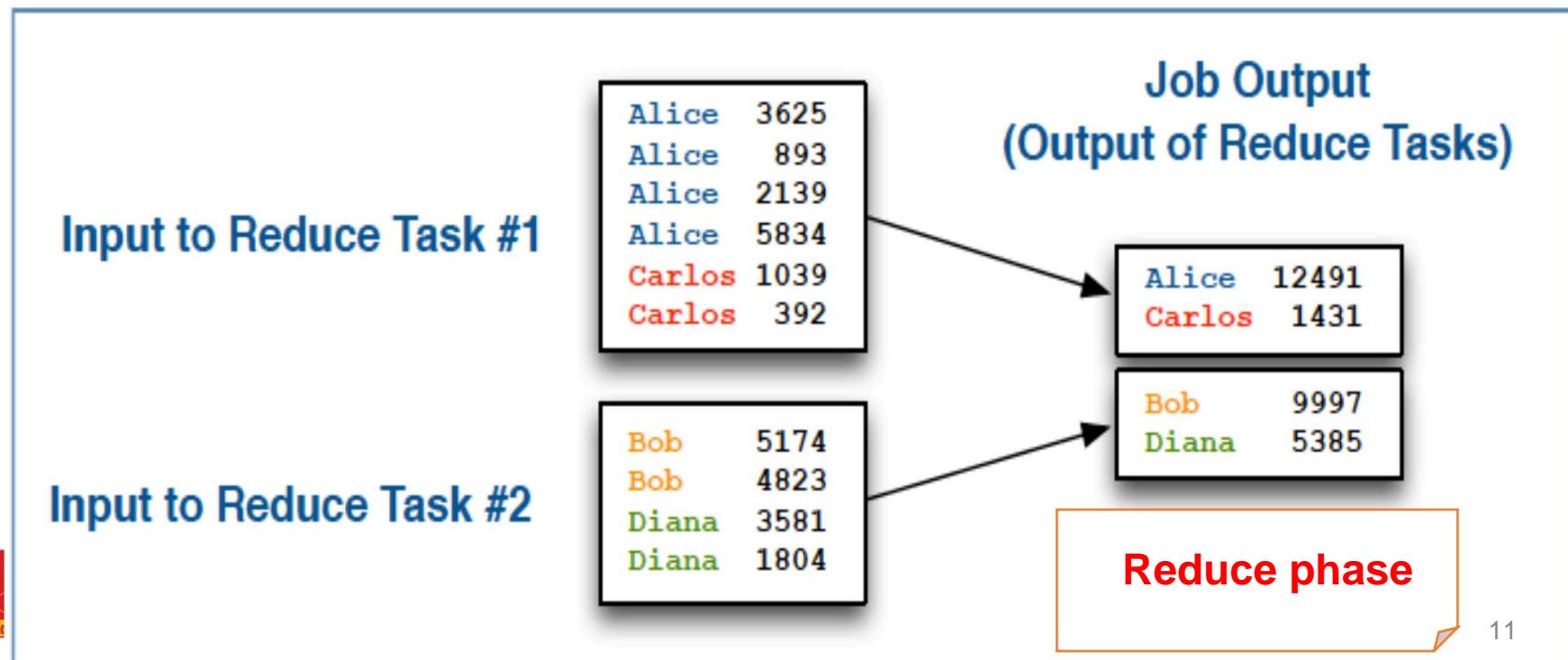
Alice	3625
Alice	893
Alice	2139
Alice	5834
Carlos	1039
Carlos	392

Bob	5174
Bob	4823
Diana	3581
Diana	1804

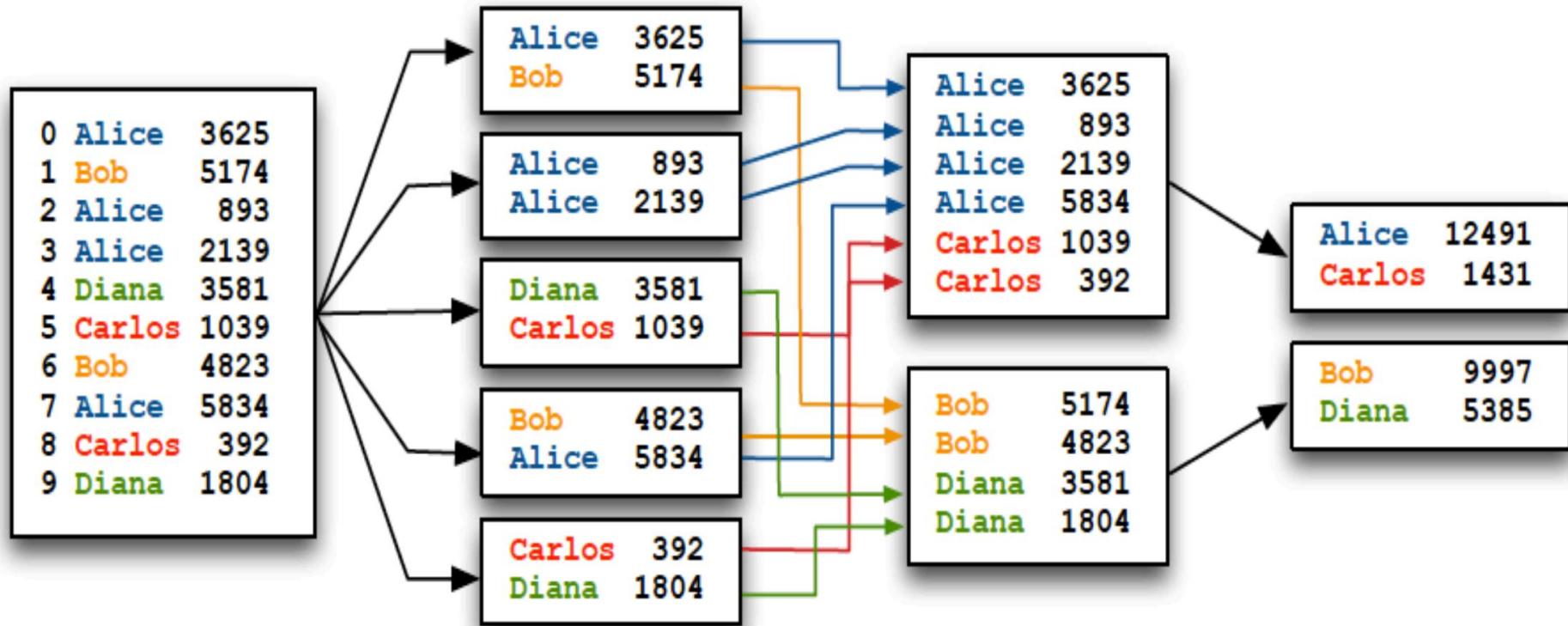
Shuffle & sort phase

Reduce phase

- Reducer input comes from the shuffle and sort process
 - As with map, the reduce function receives one record at a time
 - A given reducer receives all records for a given key
 - For each input record, reduce can emit zero or more output records
- Our reduce function simply sums total per person
 - And emits employee name (key) and total (value) as output

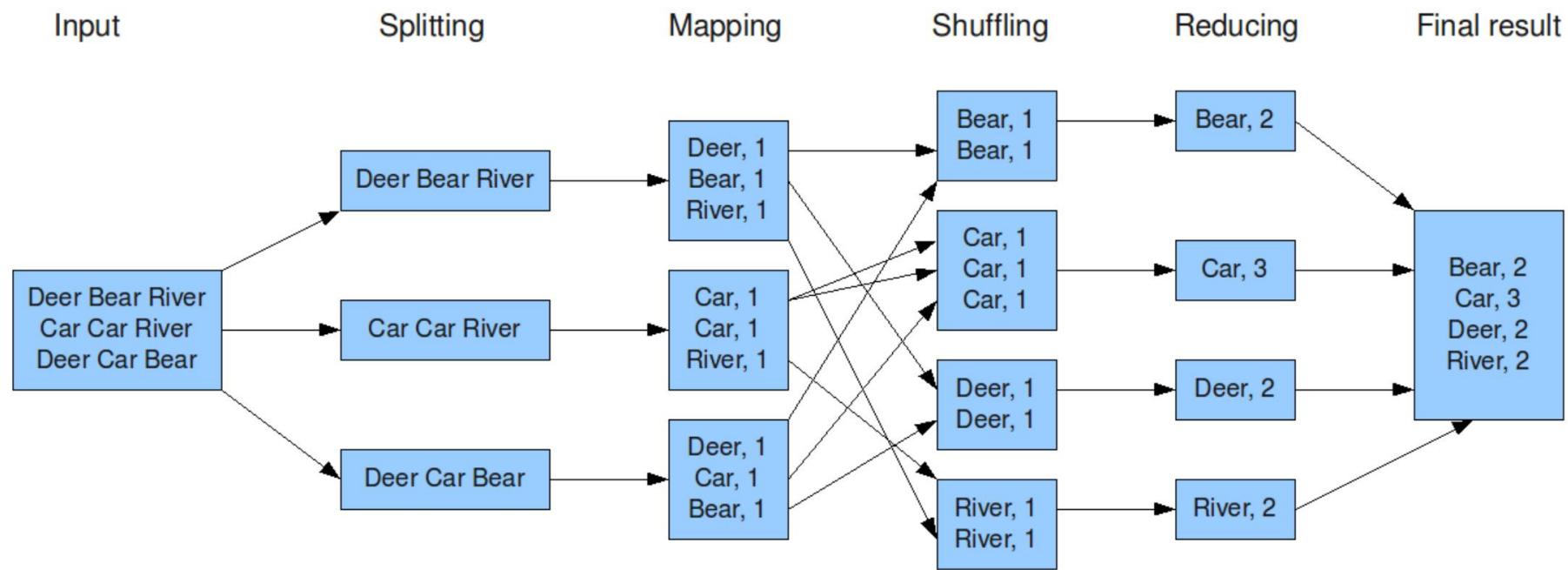


Data flow for the entire MapReduce job

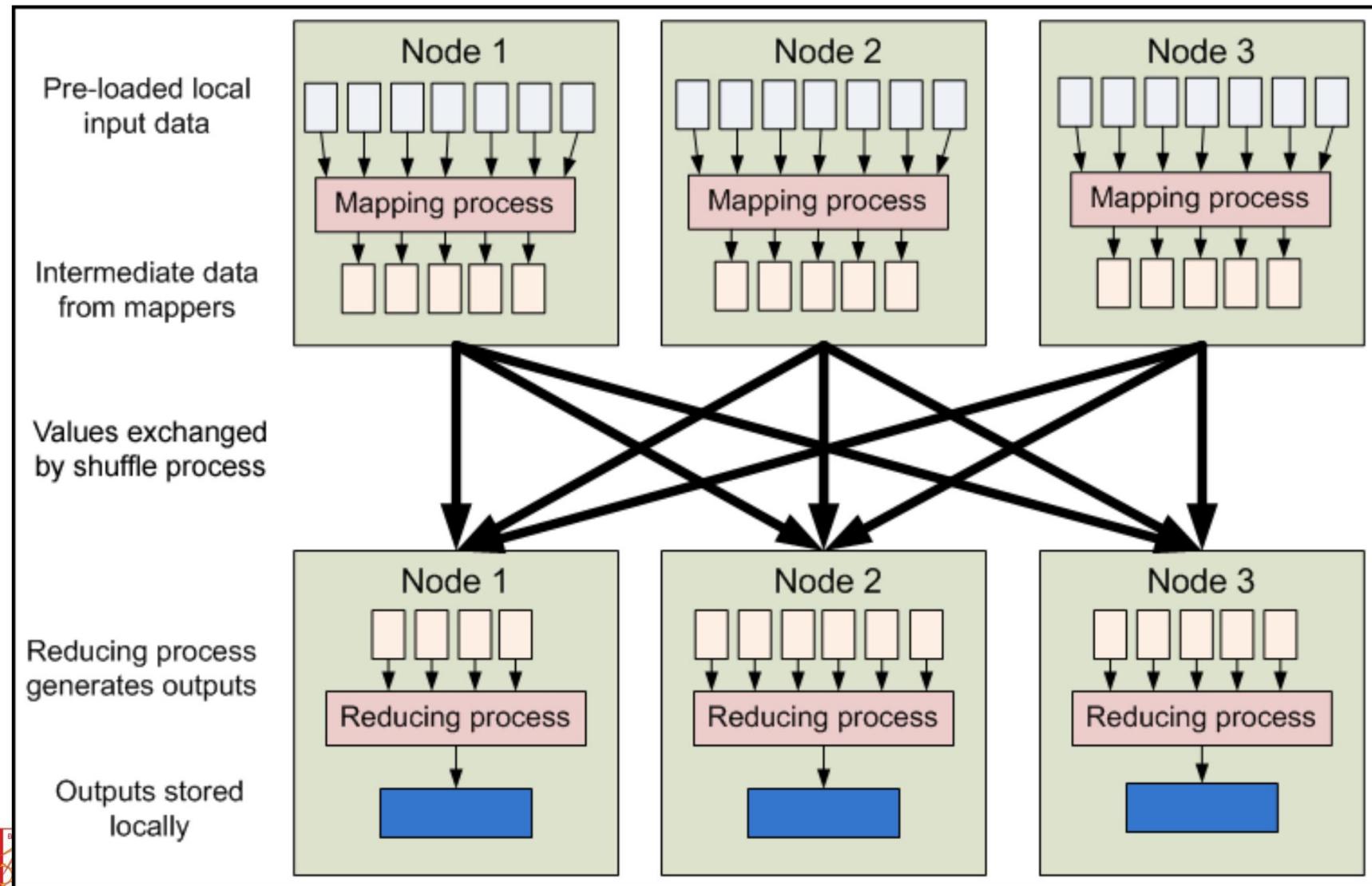


Word Count Dataflow

The overall MapReduce word count process



MapReduce - Dataflow



Example: Word Count (1)

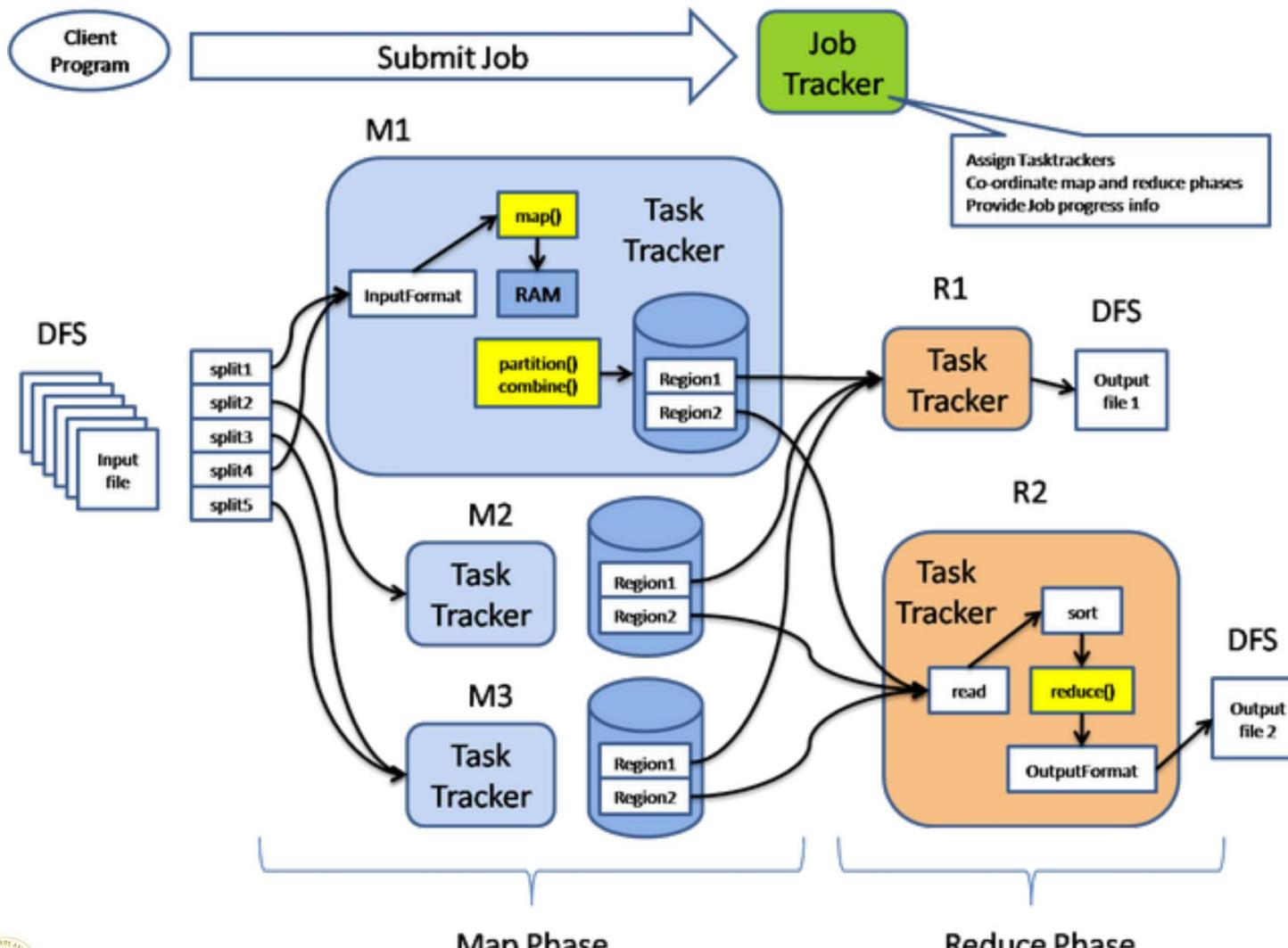
```
9 import org.apache.hadoop.mapreduce.Job;
10 import org.apache.hadoop.mapreduce.Mapper;
11 import org.apache.hadoop.mapreduce.Reducer;
12 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
13 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
14 import org.apache.hadoop.util.GenericOptionsParser;
15
16
17
18
19 public class WordCount {
20 public static void main(String [] args) throws Exception
21 {
22 Configuration c=new Configuration();
23 String[] files=new GenericOptionsParser(c,args).getRemainingArgs();
24 Path input=new Path(files[0]);
25 Path output=new Path(files[1]);
26 Job j=new Job(c,"wordcount");
27 j.setJarByClass(WordCount.class);
28 j.setMapperClass(MapForWordCount.class);
29 j.setReducerClass(ReduceForWordCount.class);
30 j.setOutputKeyClass(Text.class);
31 j.setOutputValueClass(IntWritable.class);
32 FileInputFormat.addInputPath(j, input);
33 FileOutputFormat.setOutputPath(j, output);
34 System.exit(j.waitForCompletion(true)?0:1);
35 }
```

Example: Word Count (2)

```
36 public static class MapForWordCount extends Mapper<LongWritable, Text, Text, IntWritable>{
37     public void map(LongWritable key, Text value, Context con) throws IOException, InterruptedException
38     {
39         String line = value.toString();
40         String[] words = line.split(",");
41         for(String word: words )
42         {
43             Text outputKey = new Text(word.toUpperCase().trim());
44             IntWritable outputValue = new IntWritable(1);
45             con.write(outputKey, outputValue);
46         }
47     }
48 }
49
50 public static class ReduceForWordCount extends Reducer<Text, IntWritable, Text, IntWritable>
51 {
52     public void reduce(Text word, Iterable<IntWritable> values, Context con) throws IOException, InterruptedException
53     {
54         int sum = 0;
55         for(IntWritable value : values)
56         {
57             sum += value.get();
58         }
59         con.write(word, new IntWritable(sum));
60     }
}
```



Map reduce life cycle



MapReduce algorithms

(C) <https://courses.cs.washington.edu/courses/cse490h/08au/lectures.htm>

Algorithms for MapReduce

- Sorting
- Searching
- TF-IDF
- BFS
- PageRank
- More advanced algorithms

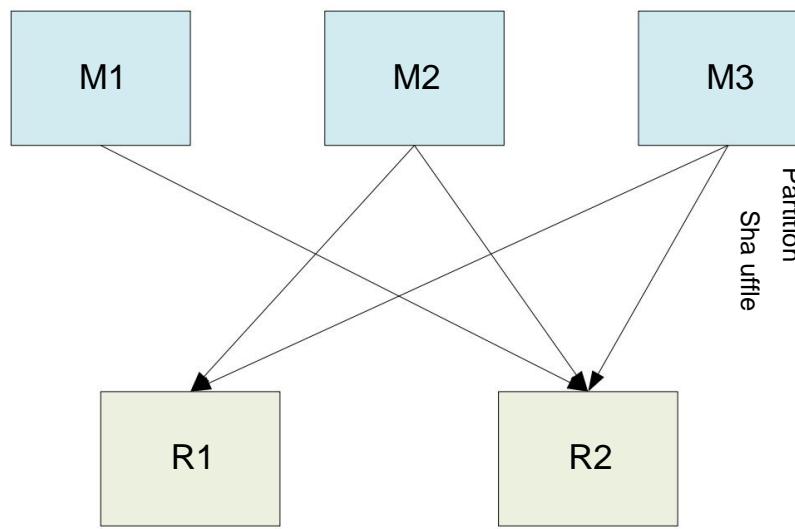
Sort algorithm

- Used as a test of Hadoop's raw speed •
Essentially “IO drag race” • Input
- A set
 - of files, one value per line •
Mapper key is file name, line number •
Mapper value is the contents of the line

Idea

- Takes advantage of reducer properties: (key, value) pairs are processed in order by key; reducers are ordered themselves
- Mapper: Identity function for value
 (k, v) to $(v, _)$
- Reducer: Identity function $(k', _) \rightarrow (k', "")$

Idea (2)



- (key, value) pairs from mappers are sent to a particular reducer based on $\text{hash}(\text{key})$
- Must pick the hash function for your data such that $k_1 < k_2 \Rightarrow \text{hash}(k_1) < \text{hash}(k_2)$

Search algorithm

- Input
 - A set of files containing lines of text
 - A search pattern to find
- Mapper key is file name, line number • Mapper value is the contents of the line • Search pattern sent as special parameter

Search algorithm

- Mapper •

Given (filename, some text) and “pattern”, if “text” matches
“pattern” output (filename, _)

- Reducer •

Identity function

Optimization

- Once a file is found to be interesting, we only need to mark it that way once
- Use *Combiner* function to fold redundant (filename, _) pairs into a single one
 - Reduces network I/O

TF-IDF algorithm

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

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

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

$$tfidf = tf \cdot idf$$

- $|D|$: total number of documents in the corpus
- $|\{d : t_i \in d\}|$: number of documents where the term t_i appears (that is $n_i \neq 0$).

Observation

- Information needed
 - Number of times term X appears in a given document
 - Number of terms in each document
 - Number of documents X appears in total number of documents

Job 1: Word frequency in each document

- Mapper •

- Input: (docname, contents) •

- Output: ((word, docname), 1)

- Reducers

- Sums counts for word in document • Outputs

- ((word, docname), n) • Combiner is the

- same as Reducer

Job 2: Word counts for documents

- Mapper •

- Input: ((word, docname), n) •

- Output: (docname, (word, n))

- Reducers

- Sums frequency of individual n's in same document • Feeds original

- data through • Outputs ((word,

- docname), (n, N)) • = ѿ !

Job 3: Word frequency in corpus

- Mapper •

Input: ((word, docname), (n, N)) •

Output: (word, (docname, n, N, 1))

- Reducers

- Number of documents where the term *word* appear d

- Outputs ((word, docname), (n, N, d))

Job 4: Calculate TF-IDF

- Mapper •

- Input: ((word, docname), (n, N, d)) •

- Assume D is known (or, easy MR to find it) •

- Output ((word, docname), TF*IDF)

- Reducers

- Just the identity function

Final thoughts on TF-IDF

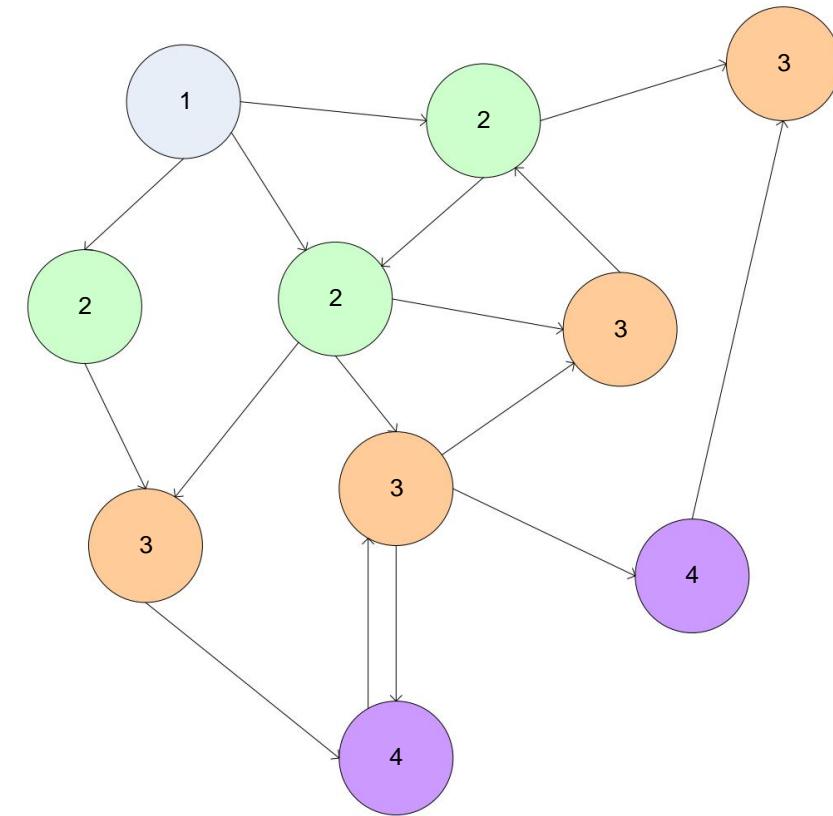
- Several small jobs add up to full algorithm
- Lots of code reuse possible
 - Stock classes exist for aggregation, identity
- Jobs 3 and 4 can actually be done at once in the same reducer, saving a write/read cycle
- Very easy to handle medium-large scale, but must take care to ensure flat memory usage for largest scale

Breadth-first search algorithm

- 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

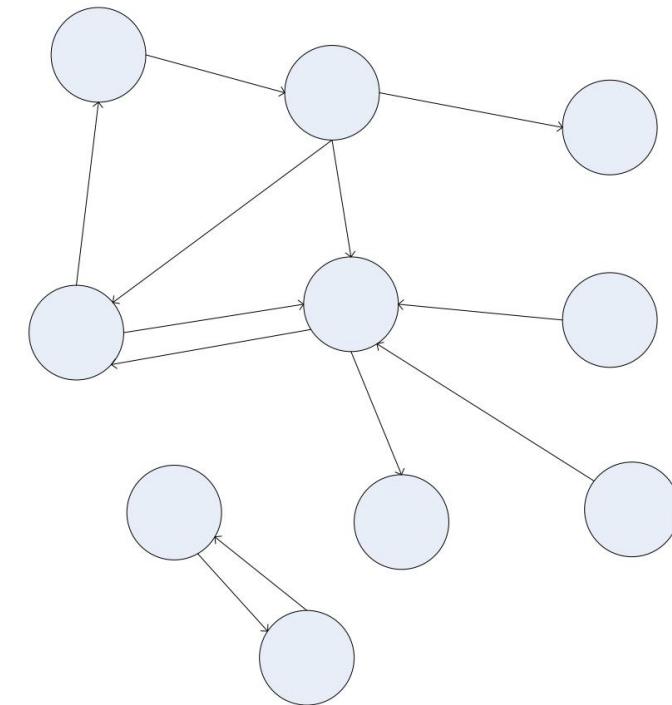
- Problem
 - This doesn't "fit" into MapReduce
- Solution
 - Iterated passes through MapReduce – map some nodes, result includes additional nodes which are fed into successively MapReduce passes

Breadth-first search & MapReduce

- Problem
 - 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 neighbors



Direct references

- Structure is inherent to objects
- 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 matrices

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

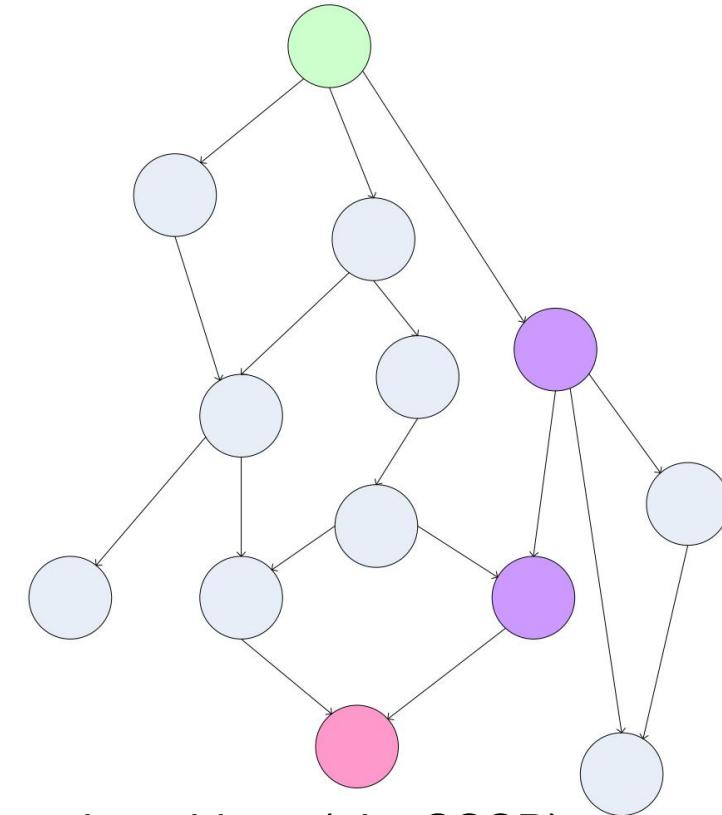
0	1	2	3	4
1	0	1	0	1
2	1	0	1	1
3	0	1	0	0
4	1	0	1	0

Adjacency matrices: Sparse representation

- Adjacency matrix for most large graphs (eg, 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. (aka SSSP)

Finding the shortest path: Intuition

- We can define the solution to this problem inductively:

$\text{DistanceTo}(\text{startNode}) = 0$

- For all nodes n directly reachable from startNode ,

$\text{DistanceTo}(n) = 1$

- For all nodes n reachable from some other set of nodes S ,

$\text{DistanceTo}(n) = 1 + \min(\text{DistanceTo}(m), m \in S)$

From intuition to algorithm

- A map task receives a node n as a key, and $(D, \text{points-to})$ as its value
 - D is the distance to the node from the start
 - points-to is a list of nodes reachable from n
 - $\forall p \in \text{points-to}, \text{emit } (p, D+1)$
- Reduce task gathers possible distances to a given p and selects the minimum one

Discussion

- This MapReduce task can advance the known frontier by one hop
- 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 $(n, \text{points-to})$ 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+wp) instead of (p, D+1) for each node p
 - Works for positive-weighted graphs

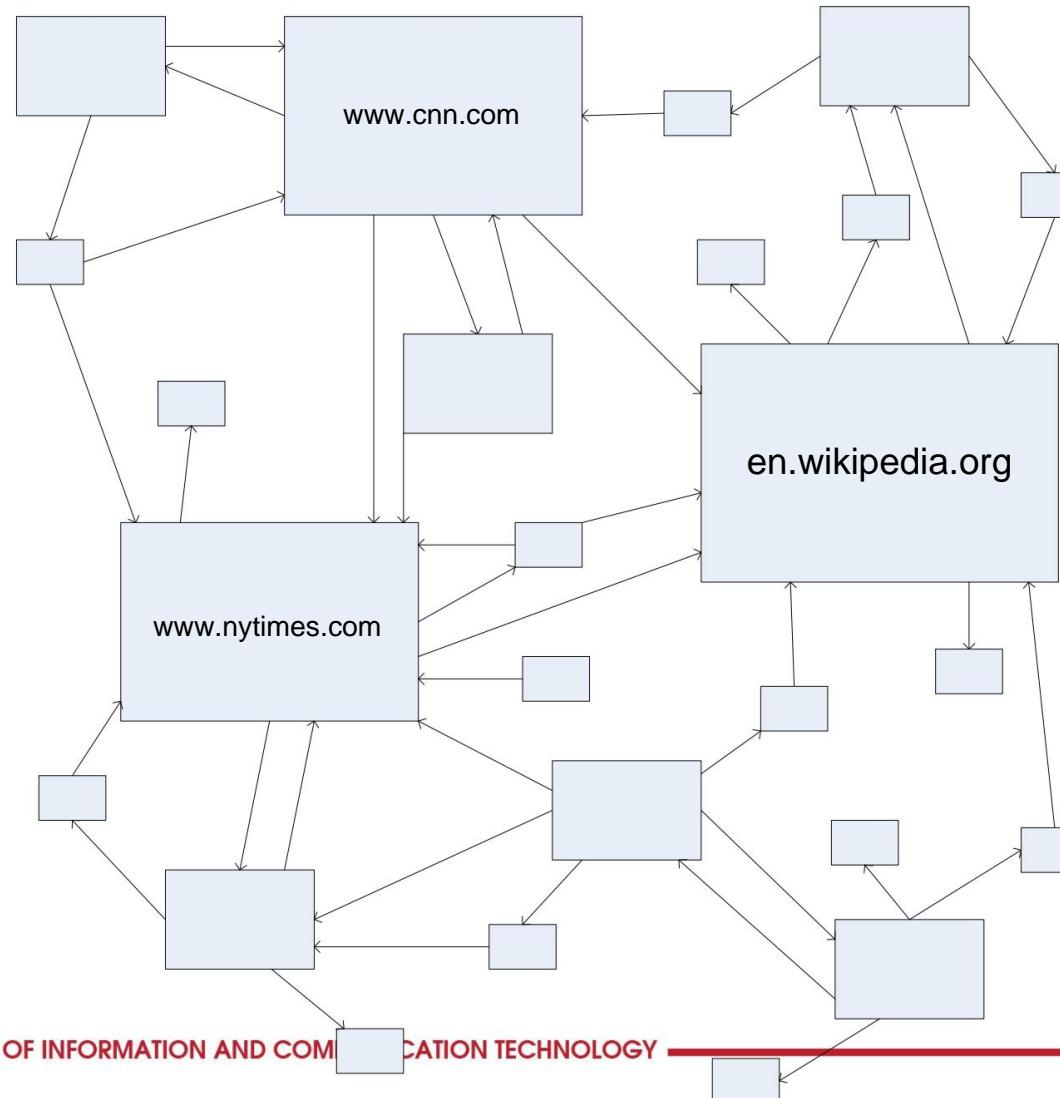
Comparison to Dijkstra

- Dijkstra's algorithm is more efficient because at any step it only pursues edges from the minimum-cost path inside the frontier
- MapReduce version explores all paths in parallel; not as efficient overall, but the architecture is more scalable
- Equivalent to Dijkstra for weight=1 case

PageRank: Random walks over the Web

- If a user starts at a random web page and surfs by clicking links and tentatively entering new URLs, what is the probability that s/he will arrive at a given page?
- The PageRank of a page captures this notion
 - More “popular” or “worthwhile” pages get a higher rank

PageRank: Visually



PageRank: Formula

- Given page A, and pages T₁ through T_n linking to A, PageRank is defined as:
 - $PR(A) = (1-d) + d \left(\frac{PR(T_1)}{C(T_1)} + \dots + \frac{PR(T_n)}{C(T_n)} \right)$
- C(P) is the cardinality (out-degree) of page P
- d is the damping (“random URL”) factor

PageRank: Intuition

- Calculation is iterative: PR_{i+1} is based on PR_i
- Each page distributes its PR_i to all pages it links to. Linkees add up their awarded rank fragments to find their PR_{i+1}
- d is a tunable parameter (usually = 0.85) encapsulating the “random jump factor”

$$\text{PR}(A) = (1-d) + d \left(\frac{\text{PR}(T_1)}{C(T_1)} + \dots + \frac{\text{PR}(T_n)}{C(T_n)} \right)$$

PageRank: First implementation

- Create two tables 'current' and 'next' holding the PageRank for each page. Seed 'current' with initial PR values
- Iterate over all pages in the graph, distribute PR from 'current' into 'next' of links
- `current := next; next := fresh_table();`
- Go back to iteration step or end if converged

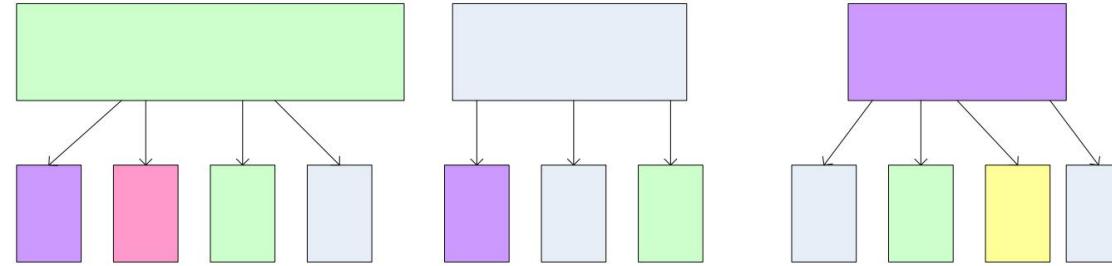
Distribution of the algorithm

- Key insights allowing parallelization:
 - The 'next' table depends on 'current', but not on any other rows of 'next'
 - Individual rows of the adjacency matrix can be processed in parallel
 - Sparse matrix rows are relatively small

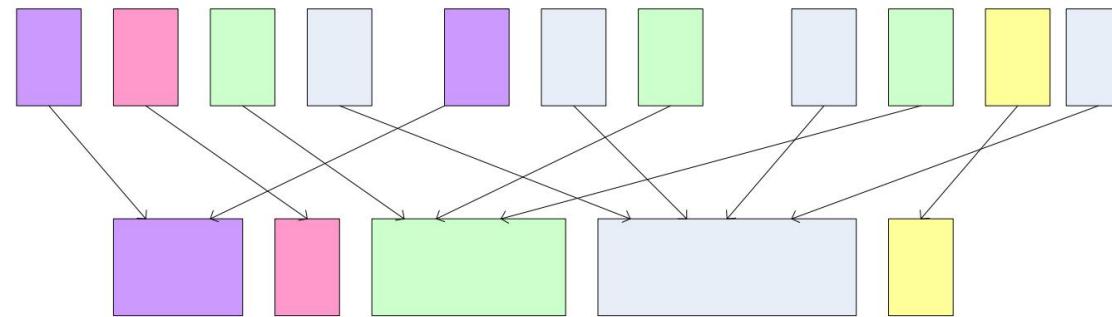
Distribution of the algorithm

- Consequences of insights:
 - We can map each row of 'current' to a list of PageRank "fragments" to assign to linkees
 - These fragments can be reduced into a single PageRank value for a page by summing
 - Graph representation can be even more compact; Since each element is simply 0 or 1, only transmit column numbers where it's 1

Map step: break page rank into even fragments to distribute to link targets



Reduce step: add together fragments into next PageRank



Iterate for next step...

Phase 1: Parse HTML

- Map task takes (URL, page content) pairs and maps them to (URL, (PRinit, list-of-urls))
 - PRinit is the “seed” PageRank for URL
 - list-of-urls contains all pages pointed to by URL
- Reduce task is just the identity function

Phase 2: PageRank distribution

- Map task takes (URL, (cur_rank, url_list))
 - For each u in url_list, emit (u, cur_rank/|url_list|)
 - Emit (URL, url_list) to carry the points-to list along through iterations
- Reduce task gets (URL, url_list) and many (URL, val) values
 - Sum vals and fix up with d
 - Emit (URL, (new_rank, url_list))

$$\text{PR}(A) = (1-d) + d \left(\frac{\text{PR}(T_1)}{C(T_1)} + \dots + \frac{\text{PR}(T_n)}{C(T_n)} \right)$$

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 Phase 2 into another Phase 2 iteration

Remark

- MapReduce runs the “heavy lifting” in iterated computation
- Key element in parallelization is independent PageRank computations in a given step
- Parallelization requires thinking about minimum data partitions to transmit (eg, compact representations of graph rows)
 - Even the implementation shown today doesn't actually scale to the whole Internet; but it works for intermediate-sized graphs

References

- Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51.1 (2008): 107-113.
- Lin, Jimmy, and Chris Dyer. "Data-intensive text processing with MapReduce." *Synthesis Lectures on Human Language Technologies* 3.1 (2010): 1-177.
- Lee, Kyong-Ha, et al. "Parallel data processing with MapReduce: a survey." *AcM SIGMOD Record* 40.4 (2012): 11-20.



25
YEARS ANNIVERSARY
SOICT

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you
for your
attention!!!

