COMP 322: Fundamentals of Parallel Programming

Lecture 35: Map Reduce

Vivek Sarkar
Department of Computer Science, Rice University
vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

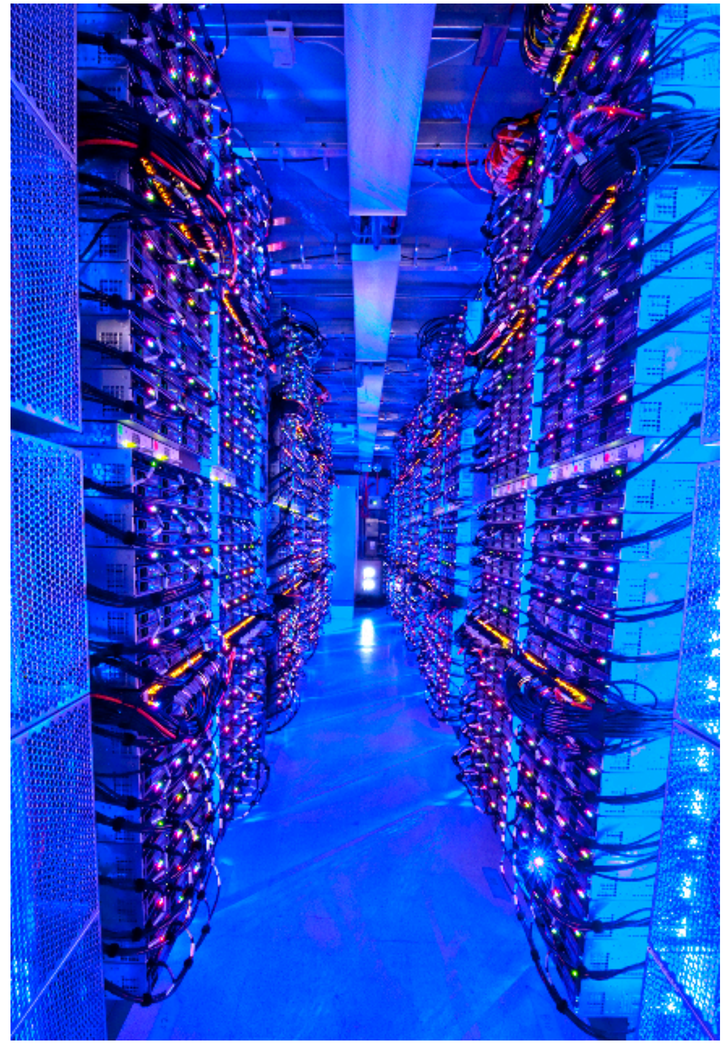
Outline

- <u>**Map Reduce Programming Model and Runtime System**</u>

- **Map Reduce Algorithms**

Mainstream trends

Personal Mobile Devices (PMD): Relying on wireless networking, Apple, Nokia, … build $500 smartphone and tablet computers for individuals
=> Objective C, Android OS



Cloud Computing: Using Local Area Networks, Amazon, Google, … build $200M Warehouse Scale Computers with 100,000 servers for Internet Services for PMDs
=> MapReduce, Ruby on Rails

Motivation: Large Scale Data Processing

- Want to process terabytes of raw data
  —documents found by a web crawl
  —web request logs
- Produce various kinds of derived read-only/append-only data
  —inverted indices
    – e.g. mapping from words to locations in documents
  —various representations of graph structure of documents
  —summaries of number of pages crawled per host
  —most frequent queries in a given day
  —...
- Input data is large
- Need to parallelize computation so it takes reasonable time
  —need hundreds/thousands of CPUs
- Need for fault tolerance

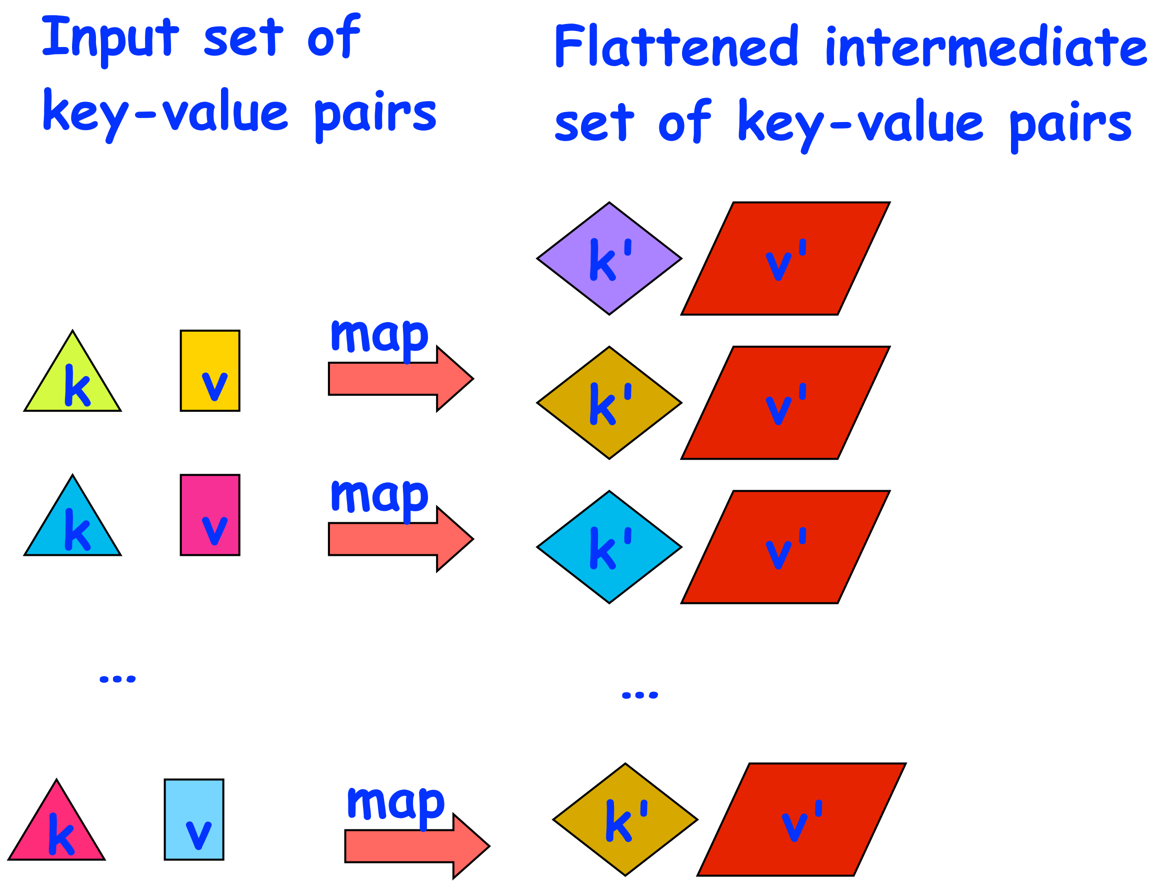
MapReduce Solution

- Apply Map function f to user supplied record of key-value pairs
- Compute set of intermediate key/value pairs
- Apply Reduce operation g to all values that share same key to combine derived data properly
  —Often produces smaller set of values
- User supplies Map and Reduce operations in functional model so that the system can parallelize them, and also re-execute them for fault tolerance

Operations on Sets of Key-Value Pairs

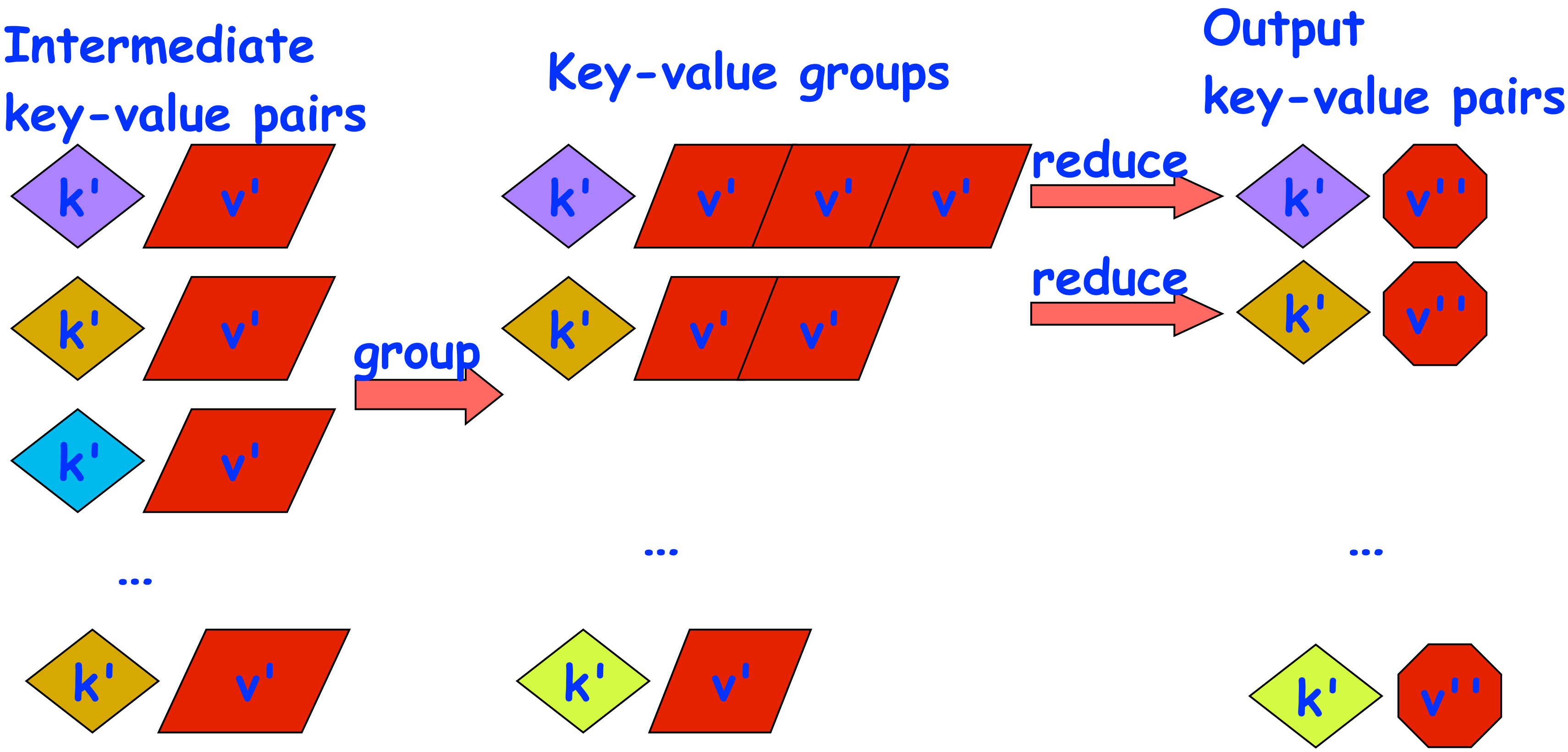
- Input set is of the form {(k1, v1), . . . (kn, vn)}, where (ki, vi) consists of a key, ki, and a value, vi.
  - Assume that the key and value objects are immutable, and that equality comparison is well defined on all key objects.
- Map function f generates sets of intermediate key-value pairs, f(ki,vi) = {(k1′ ,v1′),...(km′,vm′)}. The kj′ keys can be different from ki key in the input of the map function.
  - Assume that a flatten operation is performed as a post-pass after the map operations, so as to avoid dealing with a set of sets.
- Reduce operation groups together intermediate key-value pairs, {(k′, vj′)} with the same k’, and generates a reduced key-value pair, (k′,v″), for each such k’, using reduce function g

MapReduce: The Map Step



Source: http://infolab.stanford.edu/~ullman/mining/2009/mapreduce.ppt

MapReduce: The Reduce Step



Source: http://infolab.stanford.edu/~ullman/mining/2009/mapreduce.ppt

WordCount example

Input: set of words

Output: set of (word,count) pairs

Algorithm:

1. For each input word W, emit (W, 1) as a key-value pair (map step).

2. Group together all key-value pairs with the same key (reduce step).

3. Perform a sum reduction on all values with the same key(reduce step).

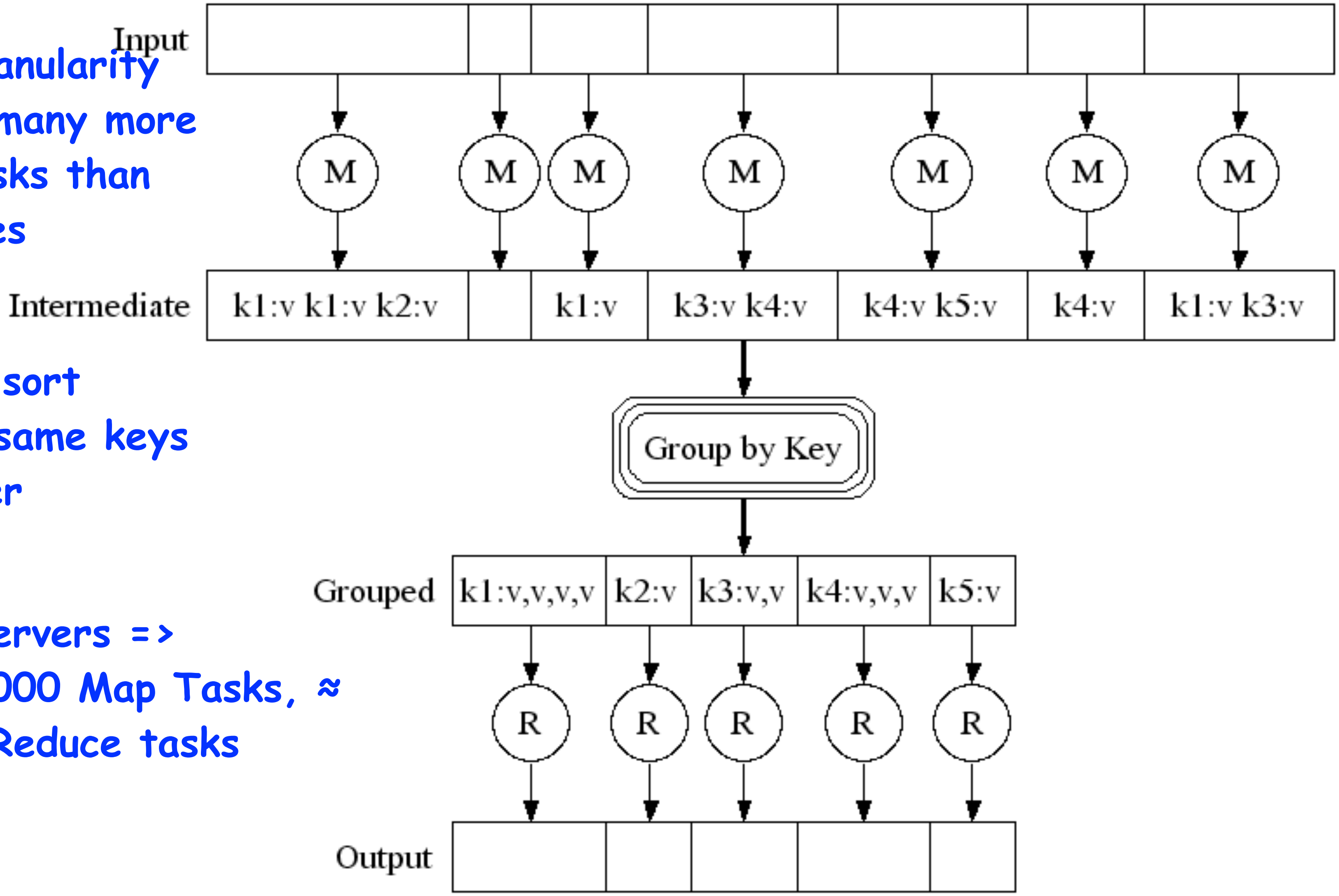
- **All map operations in step 1 can execute in parallel with only local data accesses**
- **Step 2 may involve a major reshuffle of data as all key-value pairs with the same key are grouped together.**
- **Step 3 performs a standard reduction algorithm for all values with the same key, and in parallel for different keys.**

MapReduce Execution

Fine granularity tasks: many more map tasks than machines

Bucket sort to get same keys together

2000 servers => ≈ 200,000 Map Tasks, ≈ 5,000 Reduce tasks

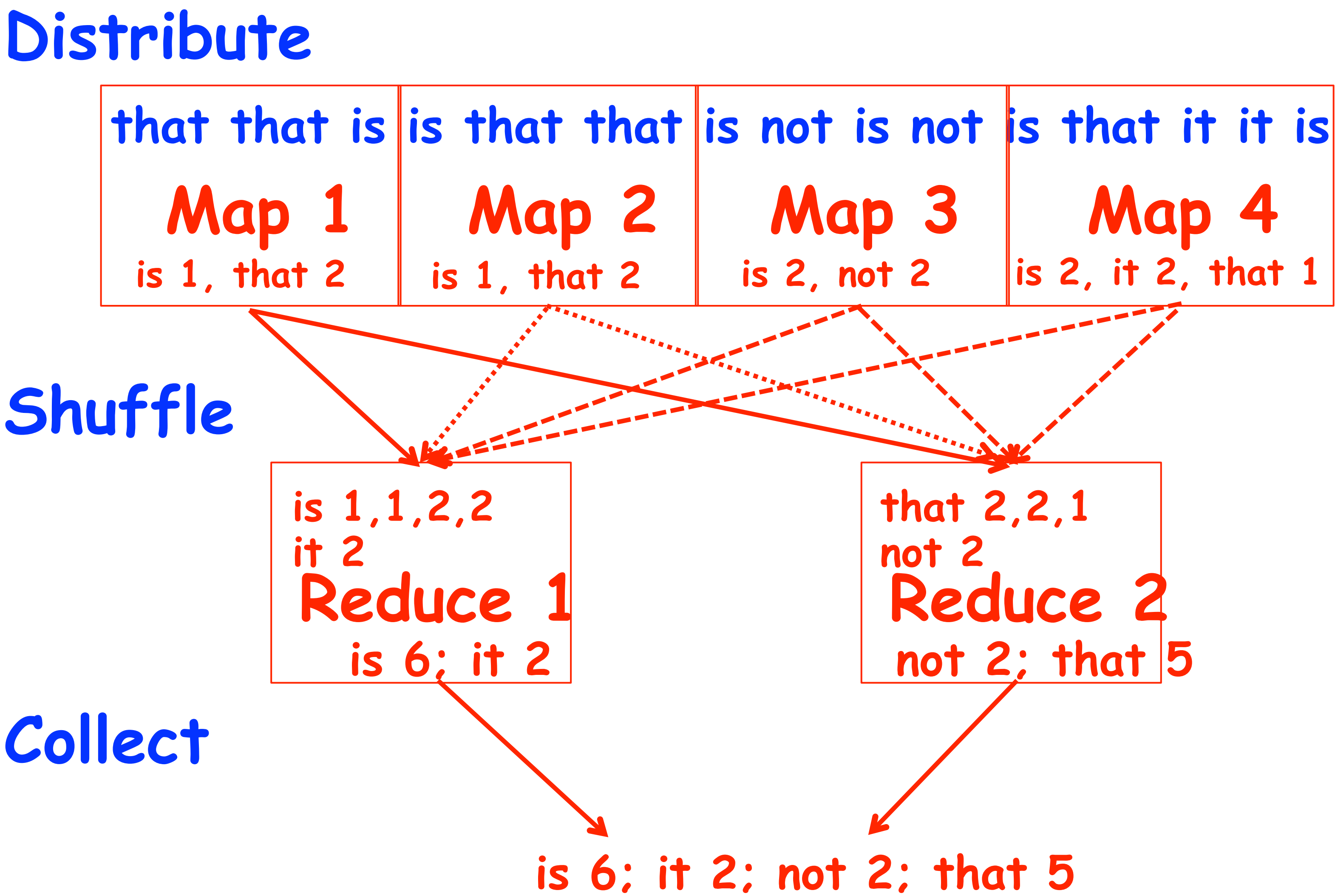


PseudoCode for WordCount

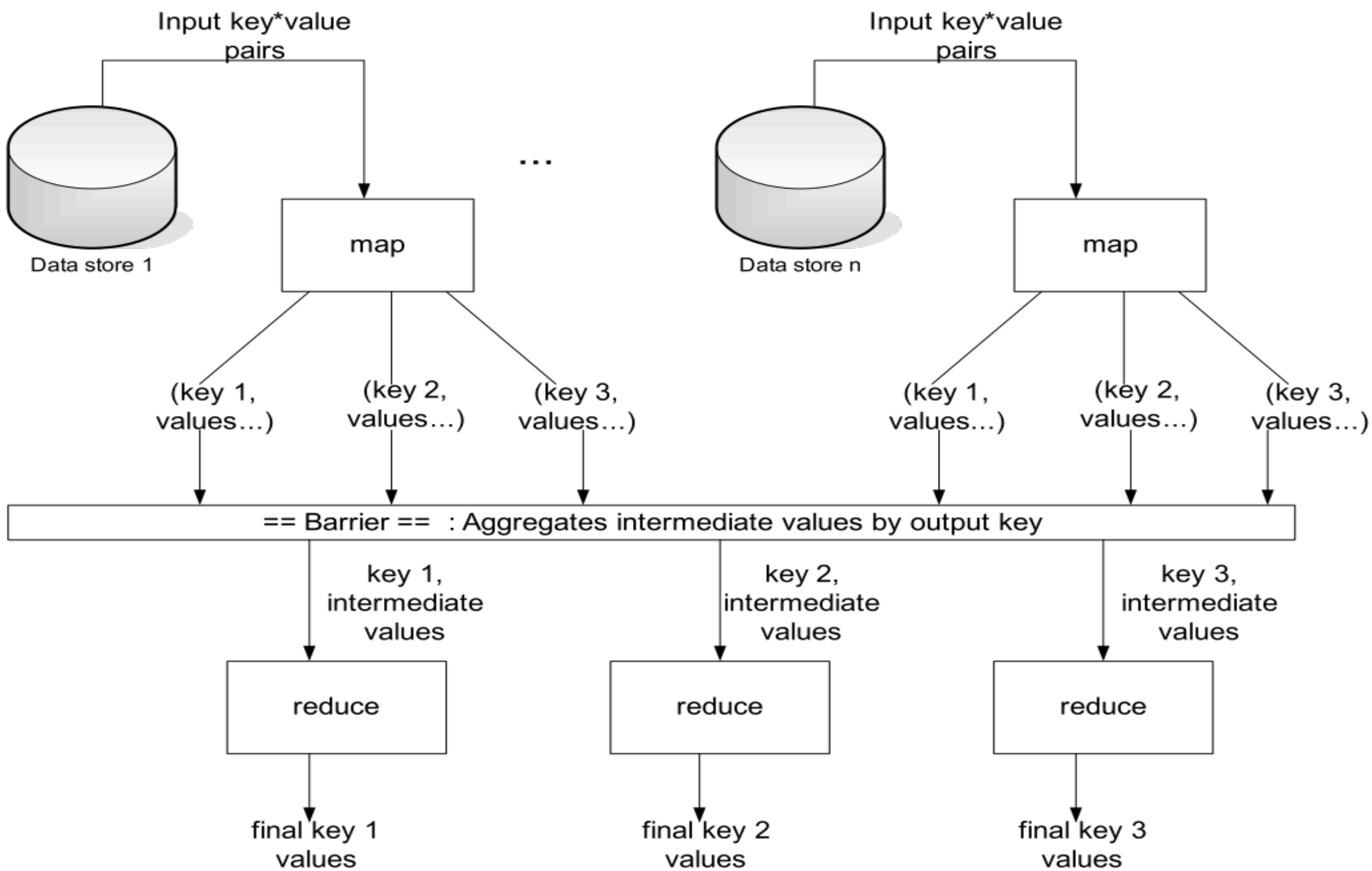
```
map(String input_key, String input_value):
  // input_key: document name
  // input_value: document contents
  for each word w in input_value:
    EmitIntermediate(w, "1"); // Produce count of words

reduce(String output_key, Iterator intermediate_values):
  // output_key: a word
  // intermediate_values: a list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v); // get integer from key-value
  Emit(AsString(result));
```

Example Execution of WordCount Program



Overall schematic for MapReduce framework on a data center cluster



MapReduce is a Data-Parallel form of the "Divide and Conquer" Pattern

- **Map:**
  - Slice data into "shards" or "splits", distribute these to workers, compute sub-problem solutions
  - `map(in_key,in_value)->list(out_key,intermediate value)`
    - Processes input key/value pair
    - Produces set of intermediate pairs
- **Reduce:**
  - Collect and combine sub-problem solutions
  - `reduce(out_key,list(intermediate_value))->list(out_value)`
    - Combines all intermediate values for a particular key
    - Produces a set of merged output values
- **Easy to use: focus on problem, let MapReduce library deal with messy details**

MapReduce Failure Handling

- **On worker failure:**
  - —Detect failure via periodic heartbeats
  - —Re-execute completed and in-progress map tasks
  - —Re-execute in progress reduce tasks
  - —Task completion committed through master
- **Master failure:**
  - —Could handle, but don't yet (master failure unlikely)
- **Robust: lost 1600 of 1800 machines once, but finished fine**

MapReduce Redundant Execution

- Slow workers significantly lengthen completion time
  - Other jobs consuming resources on machine
  - Bad disks with soft errors transfer data very slowly
  - Weird things: processor caches disabled (!!)
- Solution: Near end of phase, spawn backup copies of incomplete tasks
  - Whichever one finishes first "wins"
- Effect: Dramatically shortens job completion time
  - 3% more resources, large tasks 30% faster

MapReduce Locality Optimization during Scheduling

- **Master scheduling policy:**
  - —Asks GFS (Google File System) for locations of replicas of input file blocks
  - —Map tasks typically split into 64MB (== GFS block size)
  - —Map tasks scheduled so GFS input block replica are on same machine or same rack
- **Effect: Thousands of machines read input at local disk speed**
- **Without this, rack switches limit read rate**

Additional Optimization: Combiner Functions

- "Combiner" functions can run on same machine as a mapper
- Causes a mini-reduce phase to occur before the real reduce phase, to save bandwidth

Summary of MapReduce API

- Programmers must specify:

  map (k, v) → list(<k', v'>)

  reduce (k', list(v')) → <k'', v''>

    All values with the same key are reduced together

Optionally, also:

  partition (k', number of partitions) → 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') → <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…

Google Uses MapReduce For …

- **Web crawl**: Find outgoing links from HTML documents, aggregate by target document
- **Google Search**: Generating inverted index files using a compression scheme
- **Google Earth**: Stitching overlapping satellite images to remove seams and to select high-quality imagery
- **Google Maps**: Processing all road segments on Earth and render map tile images that display segments
- More than 10,000 MR programs at Google in 4 years,
  run 100,000 MR jobs per day (2008)

MapReduce Popularity at Google

	Aug-04	Mar-06	Sep-07	Sep-09
Number of MapReduce jobs	29,000	171,000	2,217,000	3,467,000
Average completion time (secs)	634	874	395	475
Server years used	217	2,002	11,081	25,562
Input data read (TB)	3,288	52,254	403,152	544,130
Intermediate data (TB)	758	6,743	34,774	90,120
Output data written (TB)	193	2,970	14,018	57,520
Average number servers / job	157	268	394	488

Outline

- **Map Reduce Programming Model and Runtime System**

- **<u>Map Reduce Algorithms</u>**

Algorithms for MapReduce

- Sorting
- Searching
- Indexing
- Classification
- TF-IDF
- Breadth-First Search / SSSP
- PageRank
- Clustering

Sort Algorithm

- Takes advantage of reducer properties: (key, value) pairs are processed in order by key; reducers are themselves ordered by hash function
- Mapper: Identity function for value

  $(k, v) \rightarrow (v, \_)$

- Reducer: Identity function (k’, _) -> (k’, “”)
- Trick: (key, value) pairs from mappers are sent to a particular reducer based on hash(key)
  - Must pick the hash function for your data such that $k_1 < k_2 \Rightarrow hash(k_1) < hash(k_2)$

Inverted Index: Data flow

Foo

This page contains so much text

Foo map output

contains: Foo
much: Foo
page : Foo
so : Foo
text: Foo
This : Foo

Bar

My page contains text too

Bar map output

contains: Bar
My: Bar
page : Bar
text: Bar
too: Bar

Reduced output

contains: Foo, Bar
much: Foo
My: Bar
page : Foo, Bar
so : Foo
text: Foo, Bar
This : Foo
too: Bar

- **Let's try this out in Worksheet #35!**

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 (that is $n_i \neq 0$).**

Information We Need

- 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 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*))**

Job 4: Calculate TF-IDF

- **Mapper**
  - —Input: ((word, docname), (n, N, m))
  - —Assume D is known (or, easy MR to find it)
  - —Output ((word, docname), TF*IDF)
- **Reducer**
  - —Just the identity function

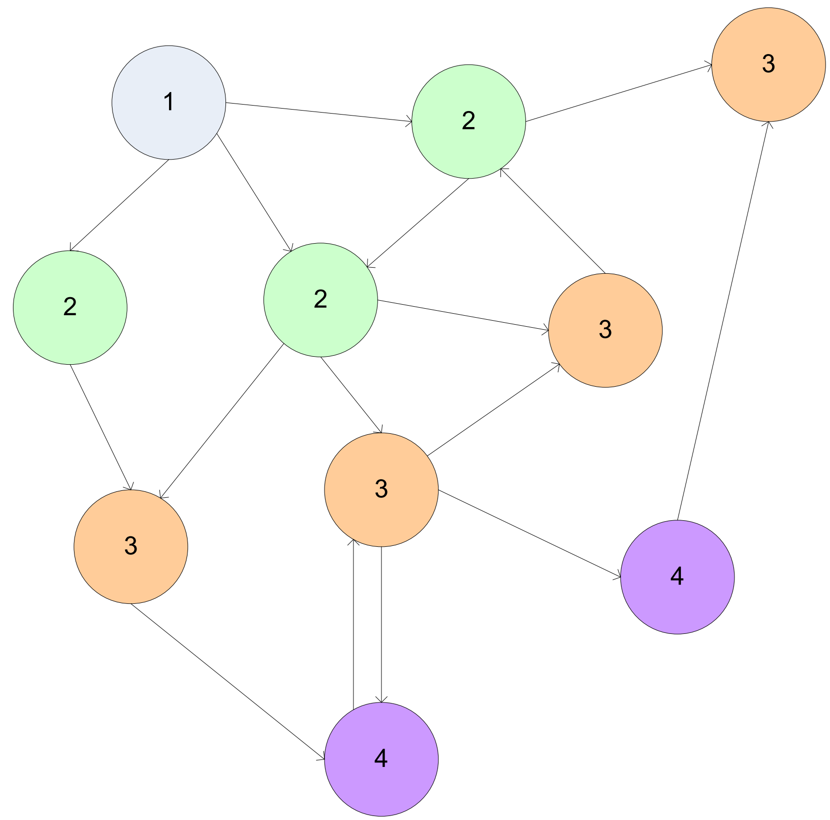
Breadth-First Search (BFS): Motivating Concepts

- **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: 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**

Adjacency Matrices

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

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

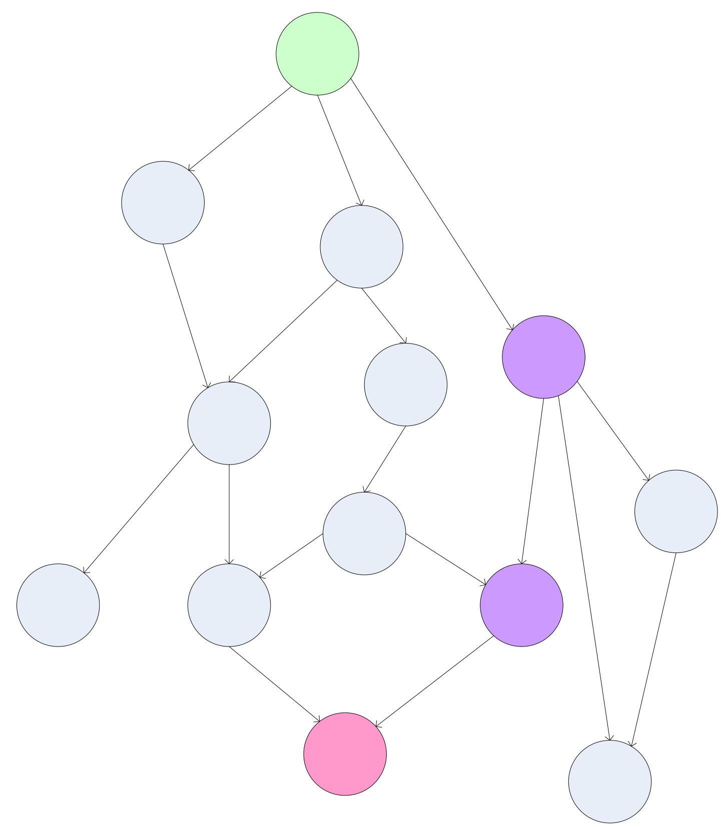
Adjacency Matrices: Sparse Representation

- **Adjacency matrix for most large graphs (e.g., the web) will be overwhelmingly full of zeros.**
- **Each row of the graph is too long to store in a dense manner**
- **Sparse matrices only include non-zero elements**

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)

Finding the Shortest Path: 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 some other set of nodes *S*,**

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

<u>Algorithm</u>:

- **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)**

- **Reduce task gathers possible distances to a given *p* and selects the minimum one**

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
- 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

Summary of Warehouse Scale Computing and Map Reduce

- Request-Level Parallelism
  - —High request volume, each largely independent of other
  - —Use replication for better request throughput, availability
- MapReduce Data Parallelism
  - —Map: Divide large data set into pieces for independent parallel processing
  - —Reduce: Combine and process intermediate results to obtain final result
- WSC CapEx vs. OpEx
  - —Economies of scale mean WSC can sell computing as a utility
  - —Servers currently dominate capital expense, and power distribution, cooling infrastructure dominate operating expense

Worksheet #35: Inverted Index

Name 1: ___________________________ Name 2: ___________________________

Assume an input set of key-value pairs of the form (file, word). Define the map and reduce functions to get an inverted index consisting of (word, file) key-value pairs.