COMP9313: Big Data Management

Lecturer: Xin Cao

Course web site: http://www.cse.unsw.edu.au/~cs9313/

Chapter 4: MapReduce III

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 Weighting

$$w_{i,j} = \mathrm{tf}_{i,j} \cdot \log \frac{N}{n_i}$$

IDF

if n_i is large

$\log$ is less importance

$w_{i,j}$	**weight assigned to term *i* in document *j***
$\mathrm{tf}_{i,j}$	**number of occurrence of term *i* in document *j***
N	**number of documents in entire collection**
n_i	**number of documents with term *i***

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

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

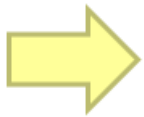
Doc 1 one fish, two fish

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



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

MapReduce: Index Construction

A simple approach:

Each Map task is a document parser

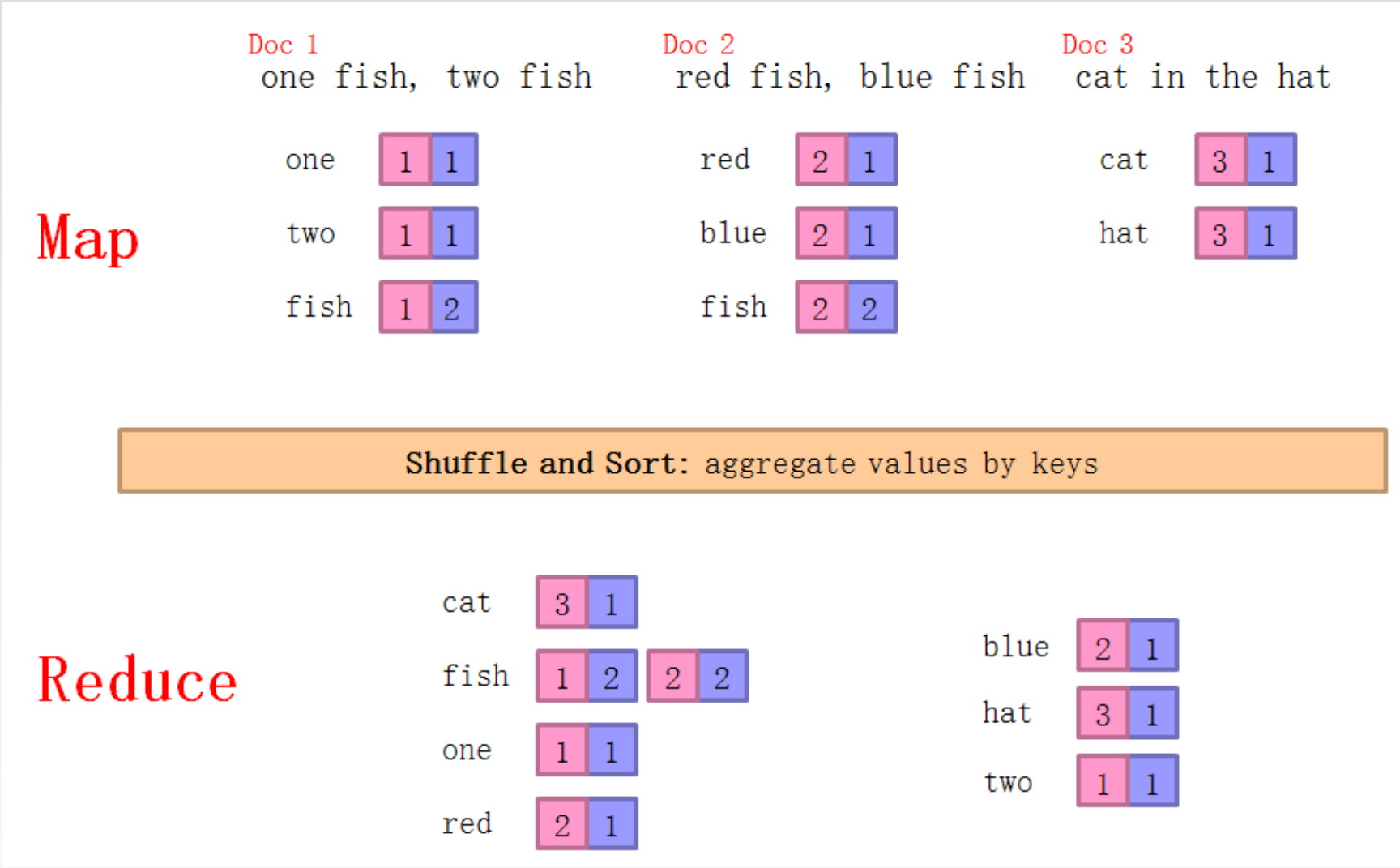
- Input: A stream of documents
  - (1, long ago …), (2, once upon …)
- Output: A stream of (term, [docid, tf]) tuples
  - (long, [1,1]) (ago, [1,1]) … (once, [2,1]) (upon, [2,1]) …

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), …])

If the list is too large?
Run out of mem

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?

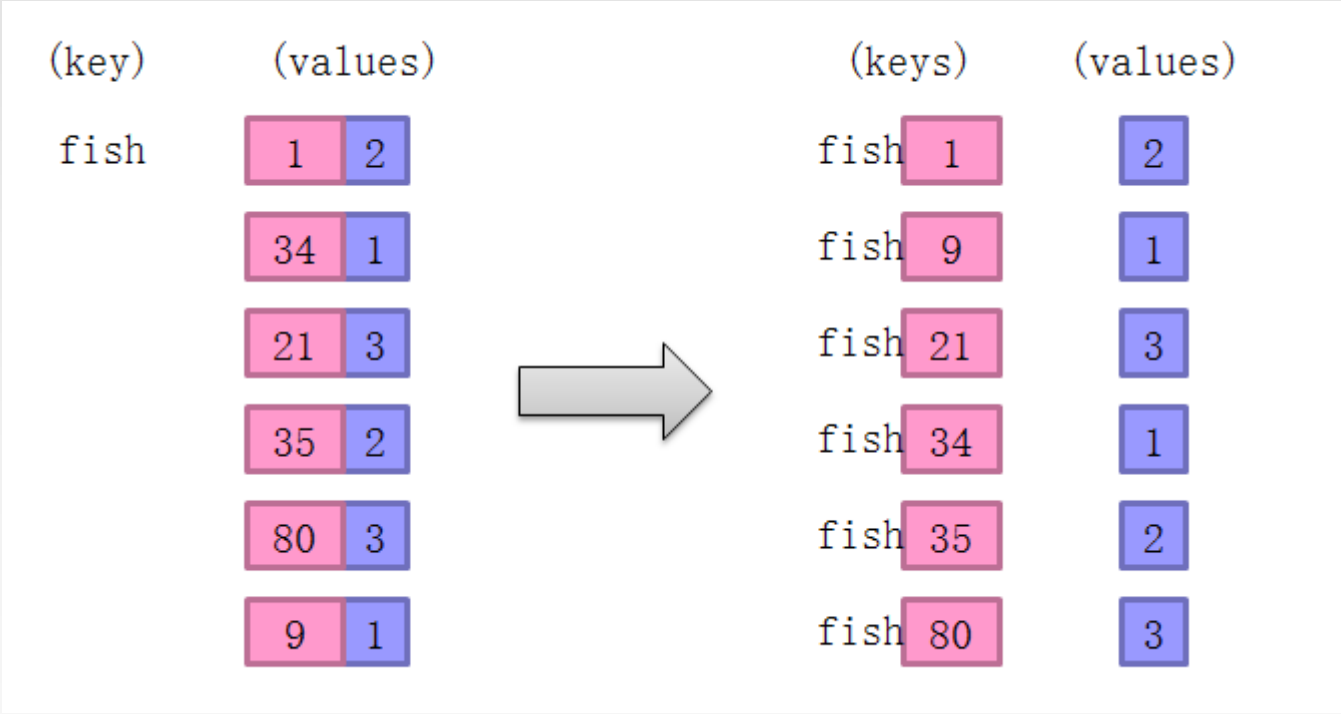
Improvement?

The First Improvement

How to make Hadoop sort the docid, instead of doing it in reducers?

Design pattern: value-to-key conversion, secondary sort

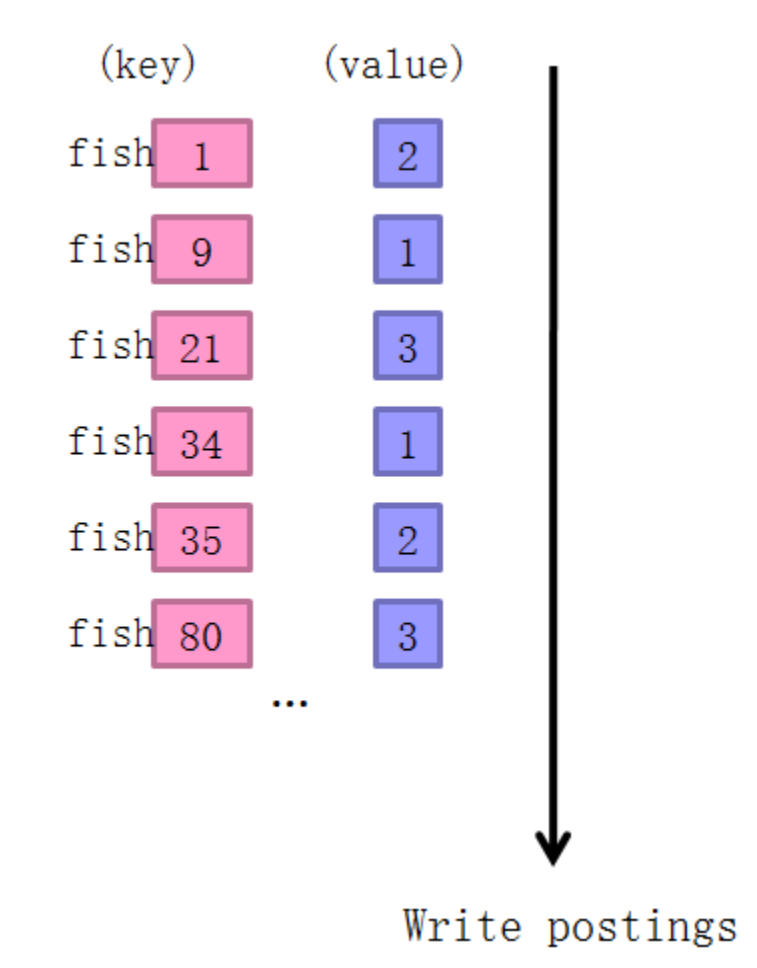
Mapper output a stream of ([term, docid], tf) tuples



Remember: you must implement a partitioner on term!

The Second Improvement

How to avoid buffering all postings associated with key?



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

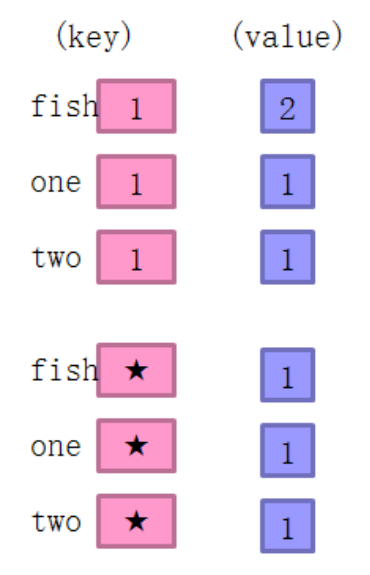
In the mapper:

- 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!

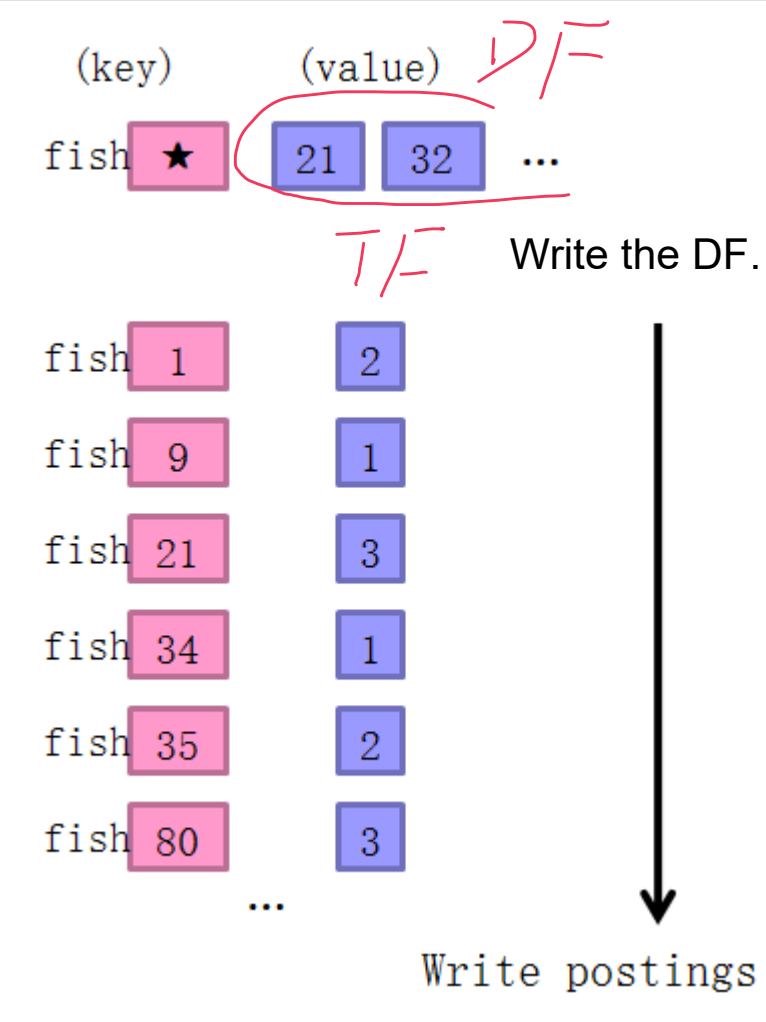


Emit normal key-value pairs…

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!

MapReduce SequenceFile

File operations based on binary format rather than text format

SequenceFile class prvoides a persistent data structure for binary key-value pairs, e.g.,

Key: timestamp represented by a LongWritable

Value: quantity being logged represented by a Writable

Use SequenceFile in MapReduce:

job.setinputFormatClass(SequenceFileOutputFormat.class);

job.setOutputFormatClass(SequenceFileOutputFormat.class);

In Mapreduce by default *TextInputFormat*

MapReduce Input Formats

InputSplit

- A **chunk** of the input processed by a single map
- Each split is divided into records
- Split is just a reference to the data (doesn't contain the input data)

```
public interface InputSplit extends Writable {

  long getLength() throws IOException;

  String[] getLocations() throws IOException;

}
```

RecordReader

- Iterate over records
- Used by the map task to generate record key-value pairs

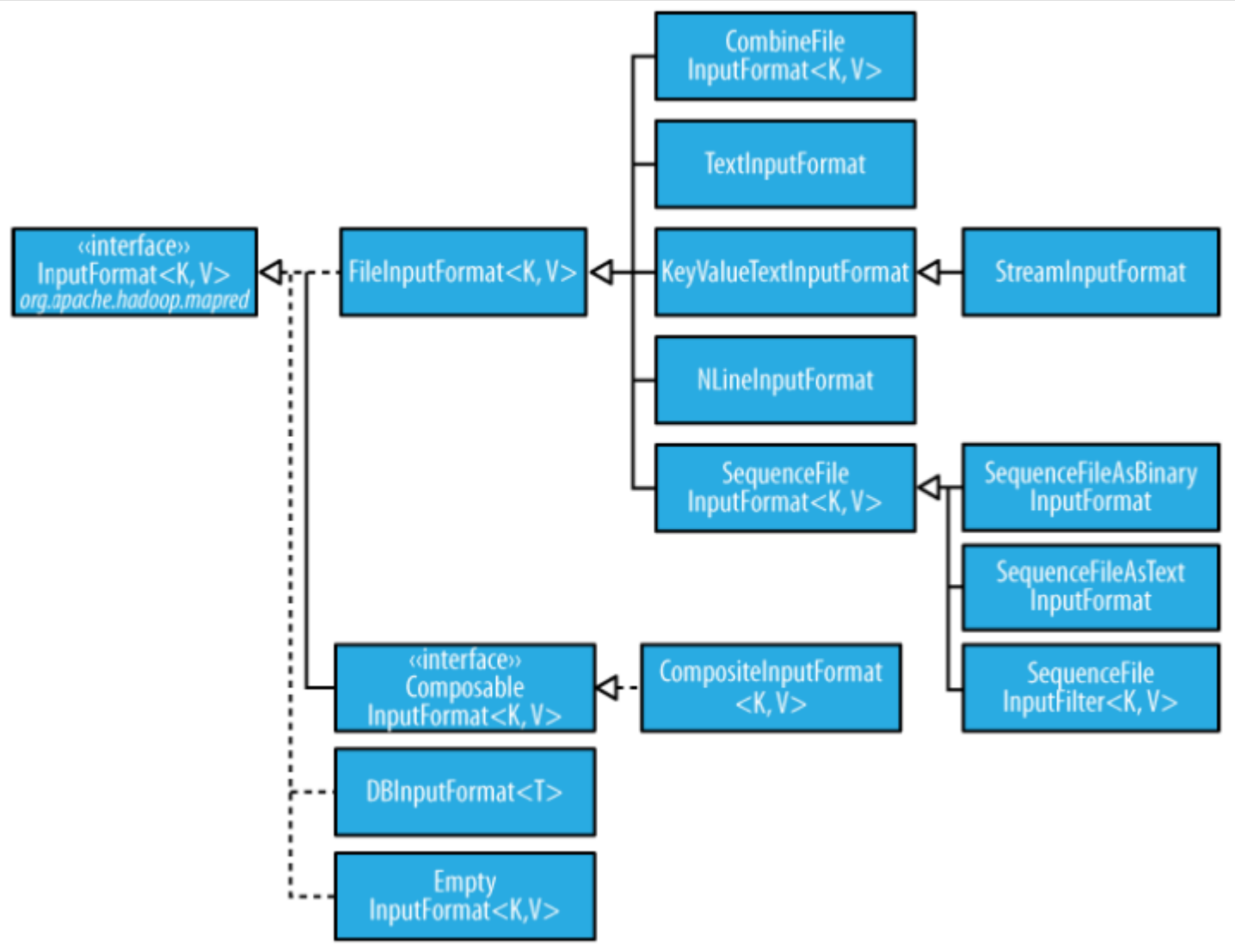
As a MapReduce application programmer, we do not need to deal with InputSplit directly, as they are created in InputFormat

In MapReduce, by default TextInputFormat and LineRecordReader

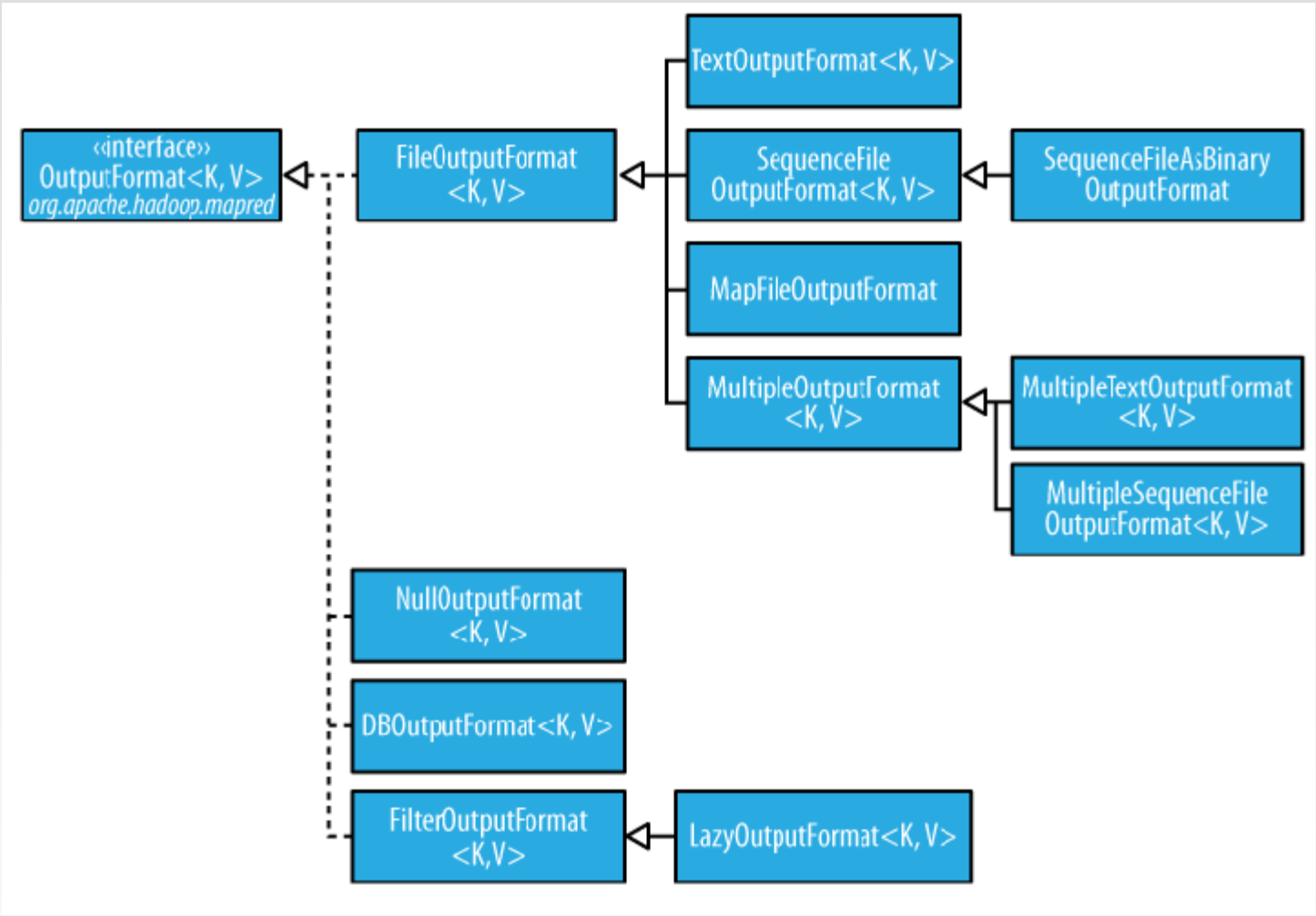
read data line by line

Not suitable for Json XML

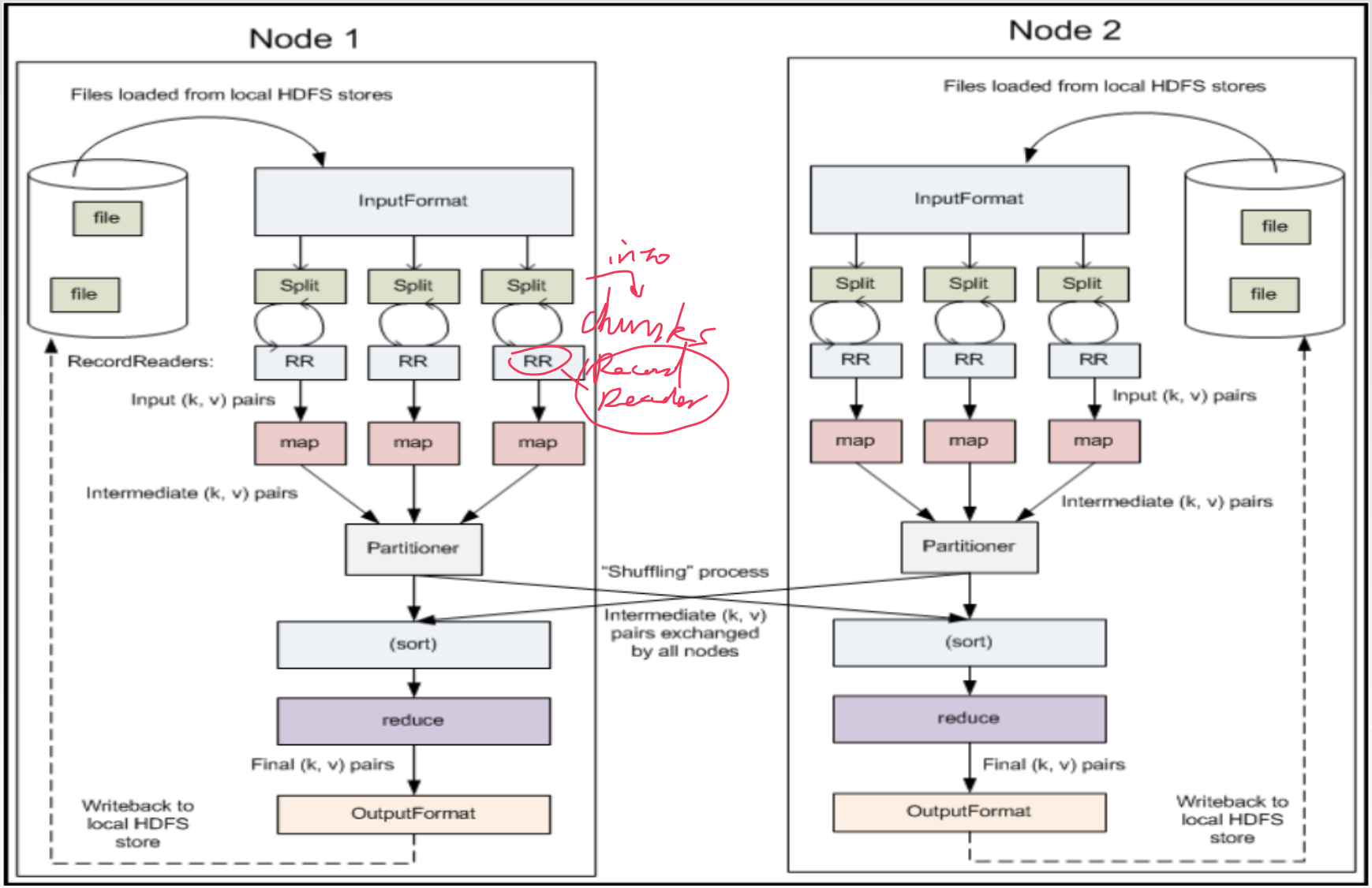
MapReduce InputFormat



MapReduce OutputFormat



Detailed Hadoop MapReduce Data Flow



Creating Inverted Index

Given you a large text file containing the contents of huge amount of webpages, in which each webpage starts with “<DOC>” and ends with “</DOC>”, your task is to create an inverted index for these documents.

A sample file

Procedure:

start with <doc>
ends with </doc>

- Implement a custom RecordReader
- Implement a custom InputFormat, and overwrite the CreateRecordReader() function to return your self-defined RecordReader object
- Configure the InputFormat class in the main function using job.setInputFormatClass()

Try to finish this task using the sample file

Graph Data Processing in MapReduce

What's a Graph?

G = (V,E), where

- V represents the set of vertices (nodes)
- E represents the set of edges (links)
- Both vertices and edges may contain additional information

Different types of graphs:

- Directed vs. undirected edges
- Presence or absence of cycles

Graphs are everywhere:

- Hyperlink structure of the Web
- Physical structure of computers on the Internet
- Interstate highway system
- Social networks

Graph Analytics

General Graph

- Count the number of nodes whose degree is equal to 5
- Find the diameter of the graphs

Web Graph

- Rank each webpage in the web graph or each user in the twitter graph using PageRank, or other centrality measure

Transportation Network

- Return the shortest or cheapest flight/road from one city to another

Social Network

- Detect a group of users who have similar interests

Financial Network

- Find the path connecting two suspicious transactions;

… …

Graphs and MapReduce

Graph algorithms typically involve:

- Performing computations at each node: based on node features, edge features, and local link structure
- Propagating computations: “traversing” the graph

Key questions:

- How do you represent graph data in MapReduce?
- How do you traverse a graph in MapReduce?

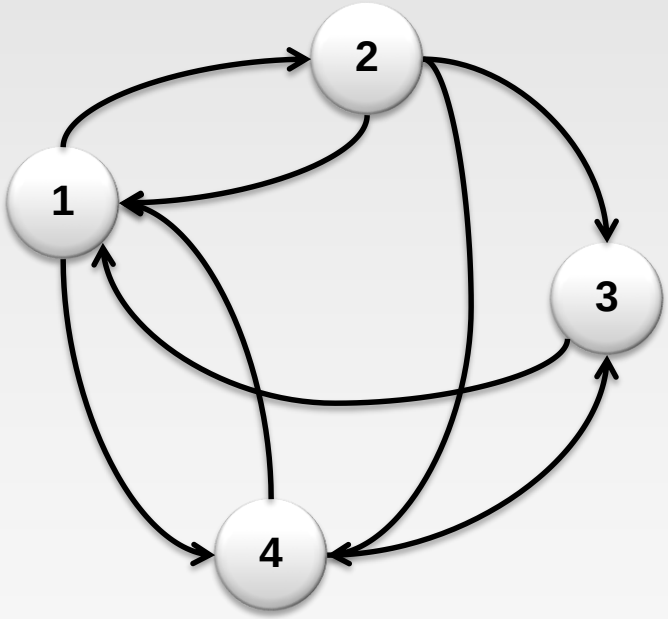
Representing Graphs

Adjacency Matrices: Represent a graph as an n x n square matrix M

$n = |V|$

$M_{ij} = 1$ means a link from node i to j

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



Adjacency Matrices: Critique

Advantages:

- Amenable to mathematical manipulation
- Iteration over rows and columns corresponds to computations on outlinks and inlinks

Disadvantages:

- Lots of zeros for sparse matrices
- Lots of wasted space

Representing Graphs

Adjacency Lists: Take adjacency matrices… and throw away all the zeros

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

1: 2, 4
2: 1, 3, 4
3: 1
4: 1, 3

Adjacency Lists: Critique

Advantages:

- Much more compact representation
- Easy to compute over outlinks

Disadvantages:

- Much more difficult to compute over inlinks

Single-Source Shortest Path

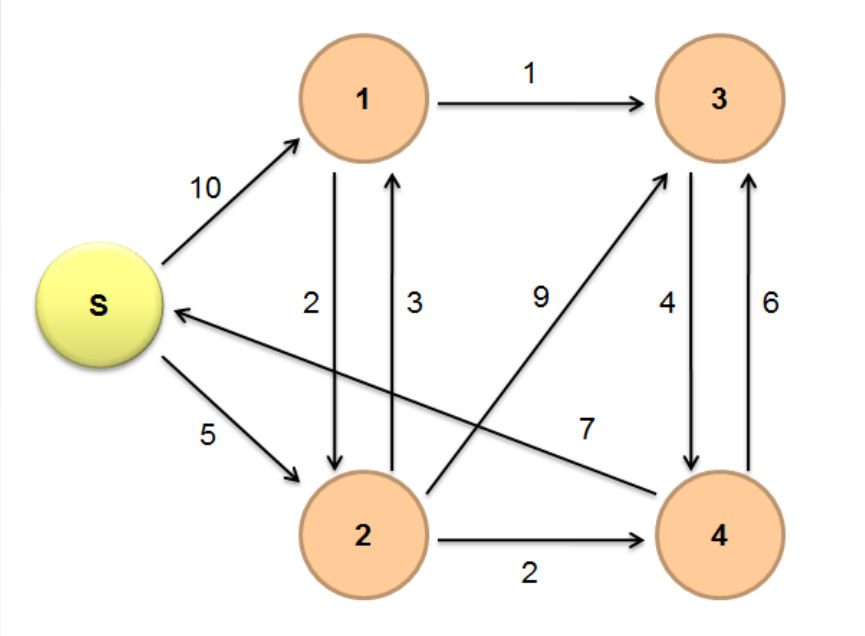
Single-Source Shortest Path (SSSP)

Problem: find shortest path from a source node to one or more target nodes

Shortest might also mean lowest weight or cost

Dijkstra's Algorithm:

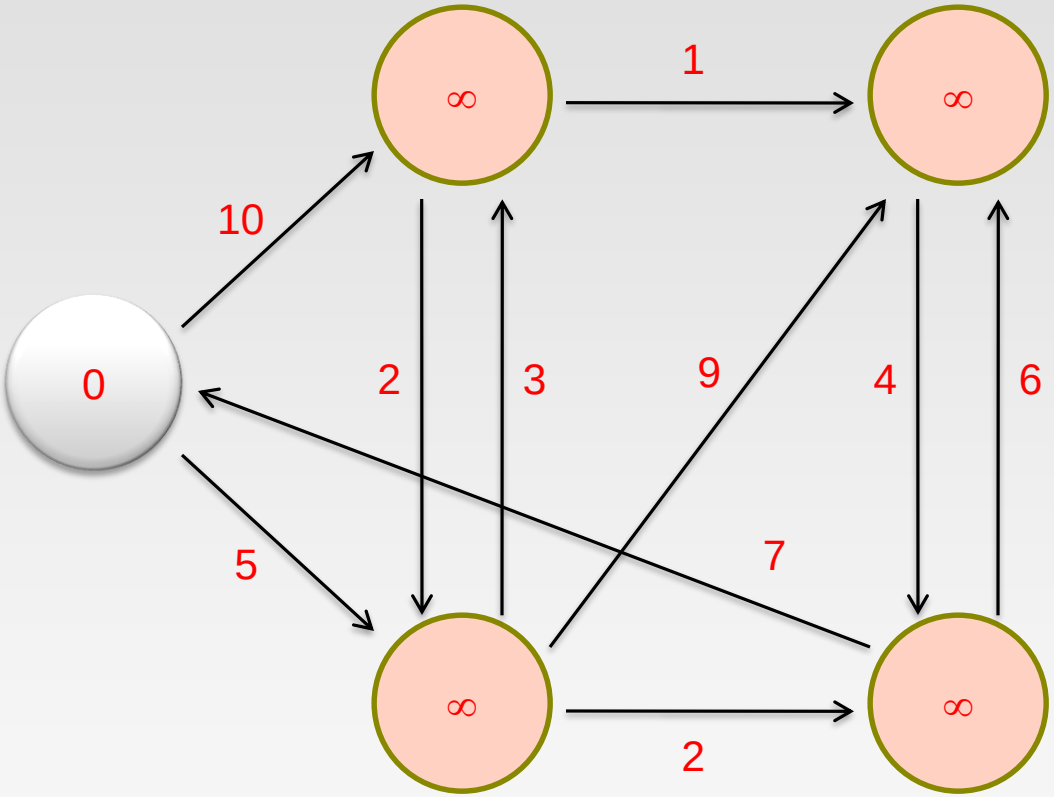
For a given source node in the graph, the algorithm finds the shortest path between that node and every other



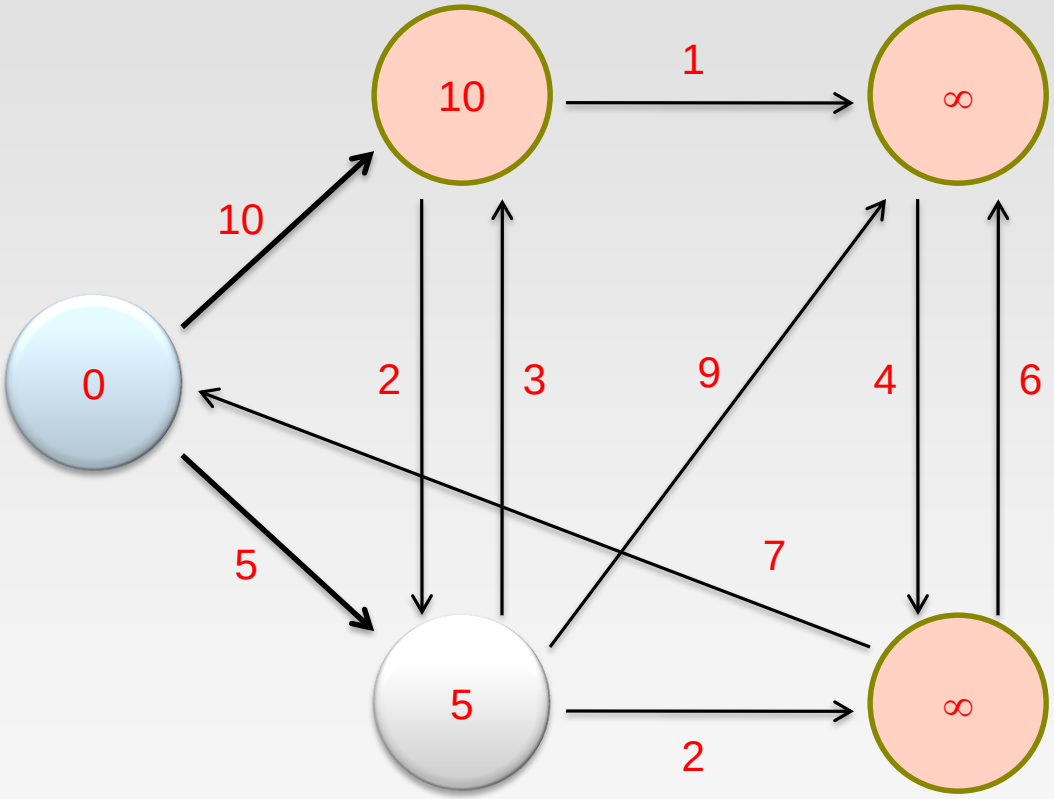
Dijkstra's Algorithm

```
1:  DIJKSTRA(G, w, s)
2:      d[s] ← 0
3:      for all vertex v ∈ V do
4:          d[v] ← ∞
5:      Q ← {V}
6:      while Q ≠ ∅ do
7:          u ← EXTRACTMIN(Q)
8:          for all vertex v ∈ u.ADJACENCYLIST do
9:              if d[v] > d[u] + w(u, v) then
10:                 d[v] ← d[u] + w(u, v)
```

Dijkstra's Algorithm Example

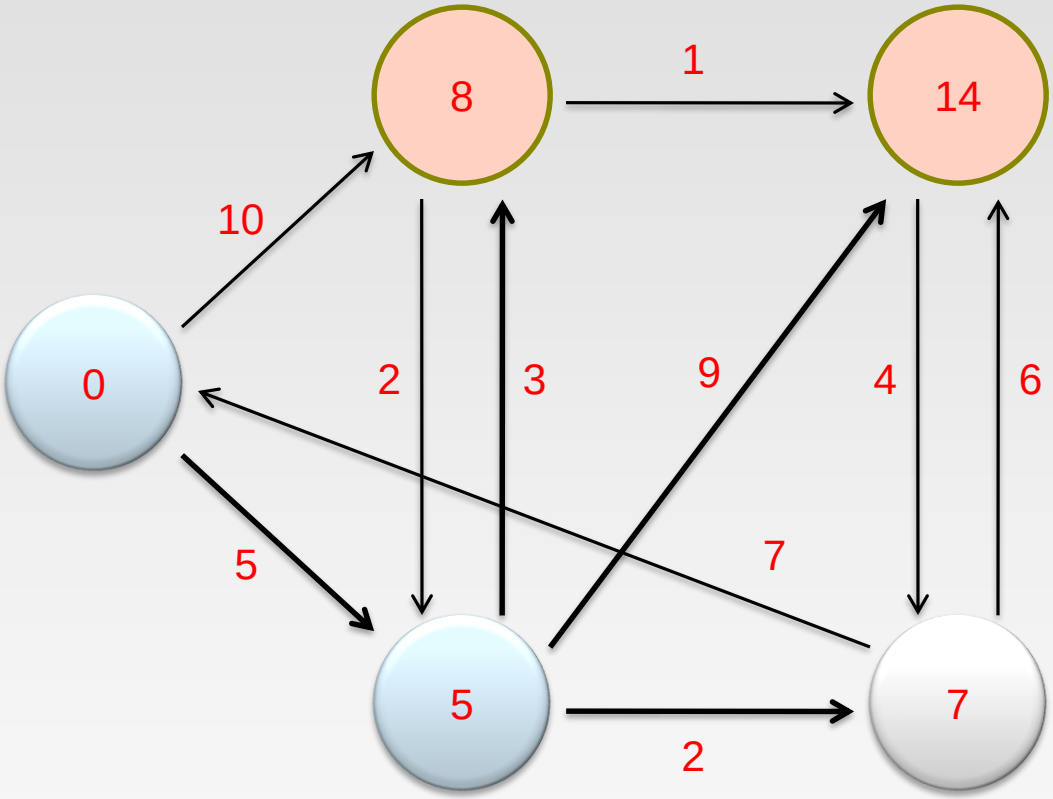


Dijkstra's Algorithm Example

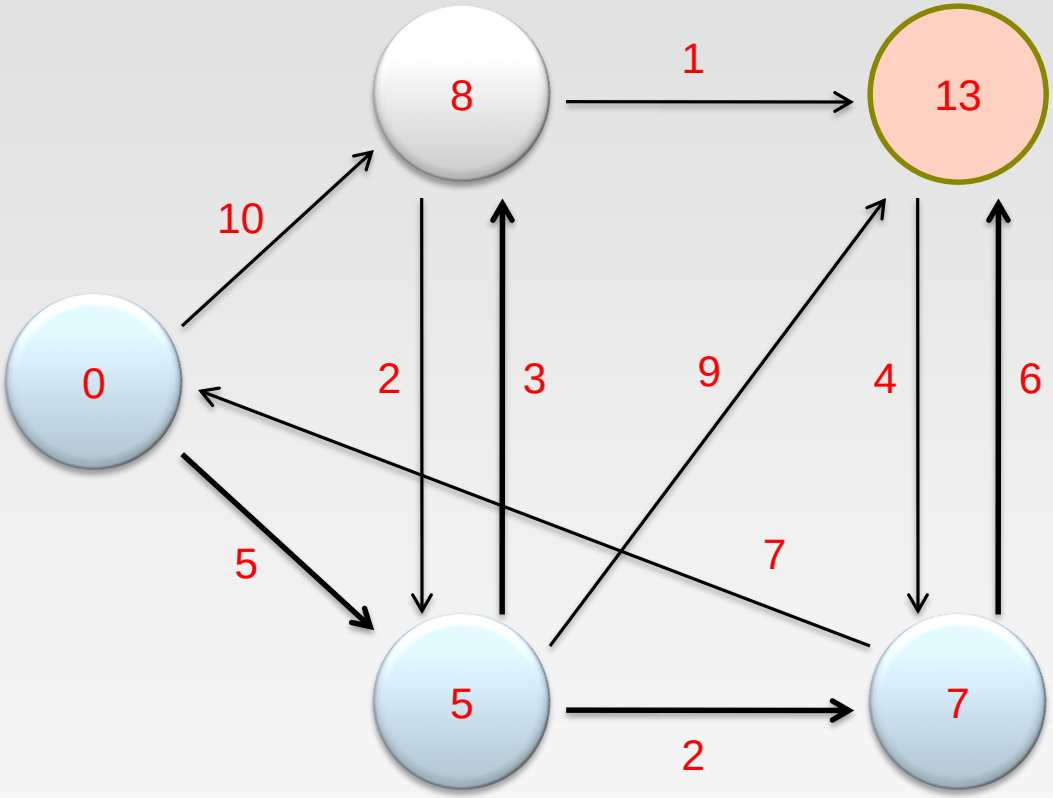


Example from CLR

Dijkstra's Algorithm Example

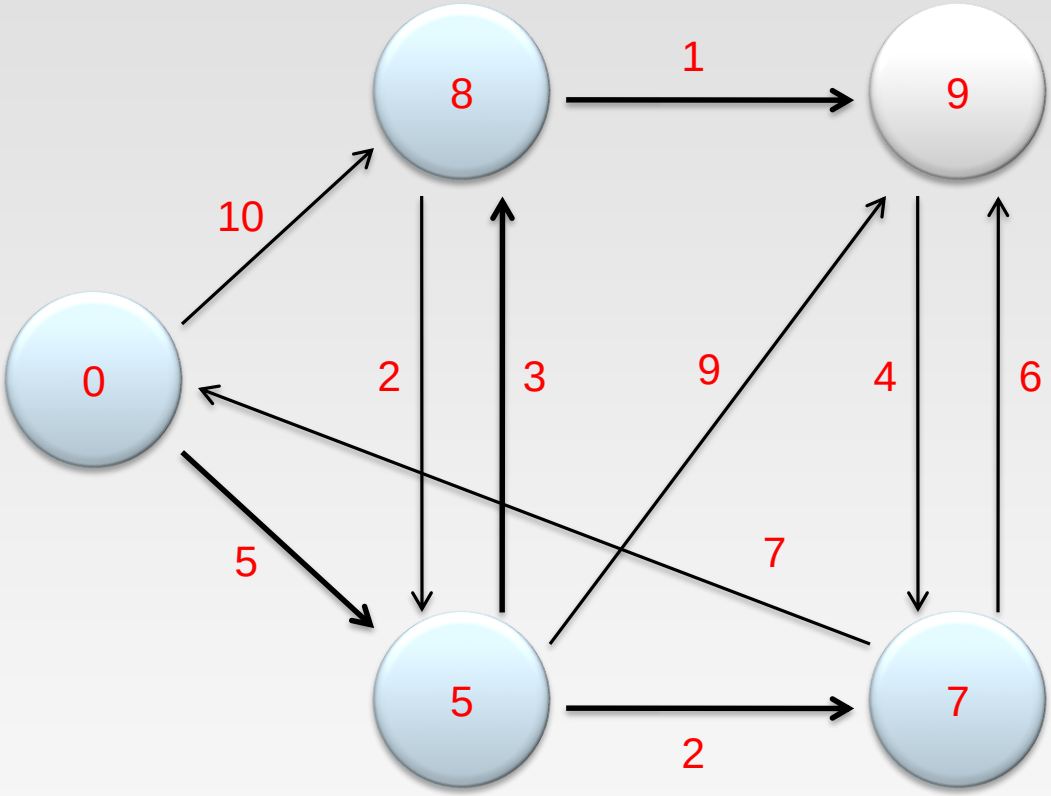


Dijkstra's Algorithm Example



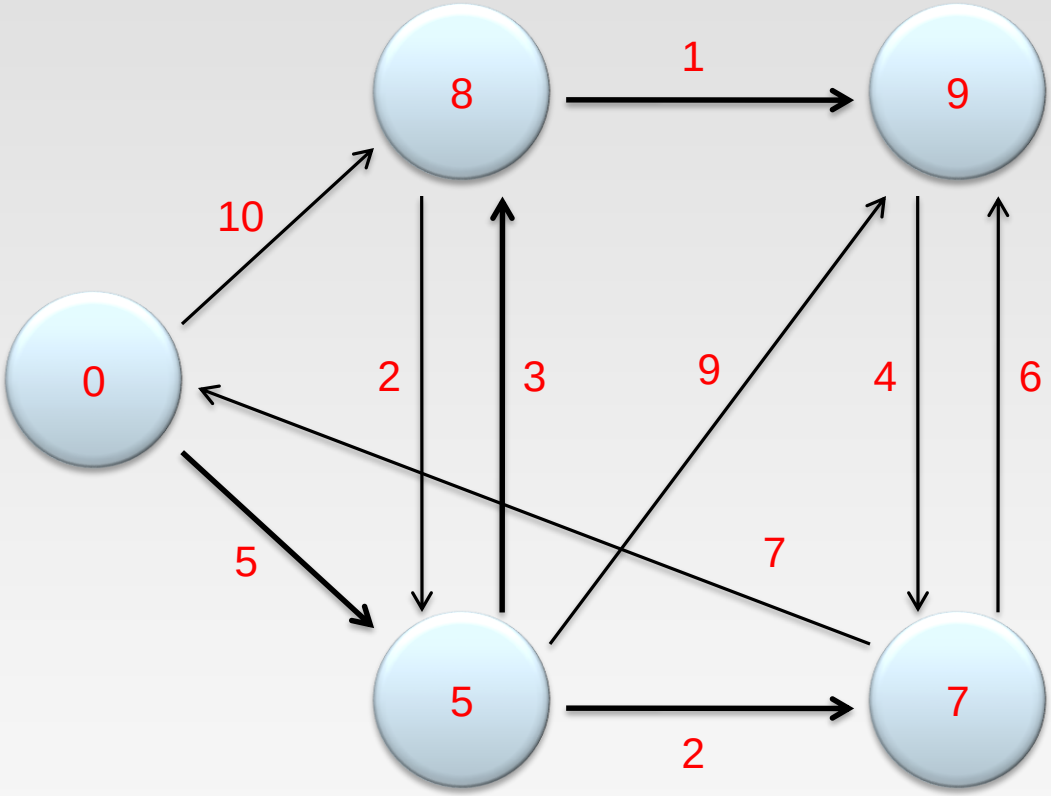
Example from CLR

Dijkstra's Algorithm Example



Example from CLR

Dijkstra's Algorithm Example



Finish!

Single Source Shortest Path

Problem: find shortest path from a source node to one or more target nodes

Shortest might also mean lowest weight or cost

Single processor machine: Dijkstra's Algorithm

MapReduce: parallel Breadth-First Search (BFS)

Finding the Shortest Path

Consider simple case of equal edge weights

Solution to the problem can be defined inductively

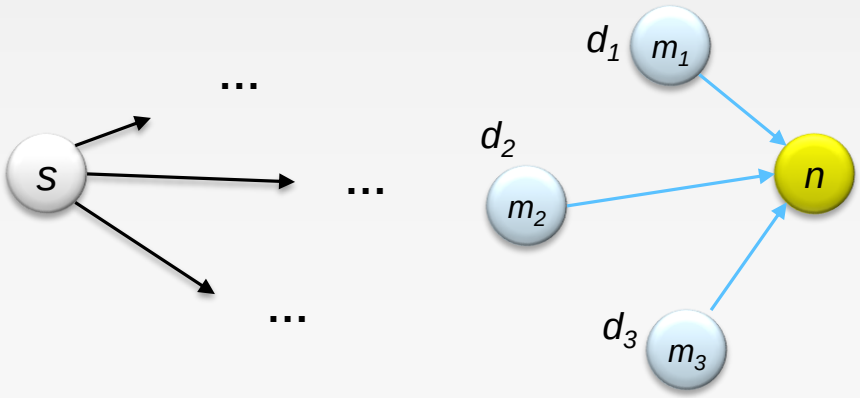
Here's the intuition:

Define: *b* is reachable from *a* if *b* is on adjacency list of *a*

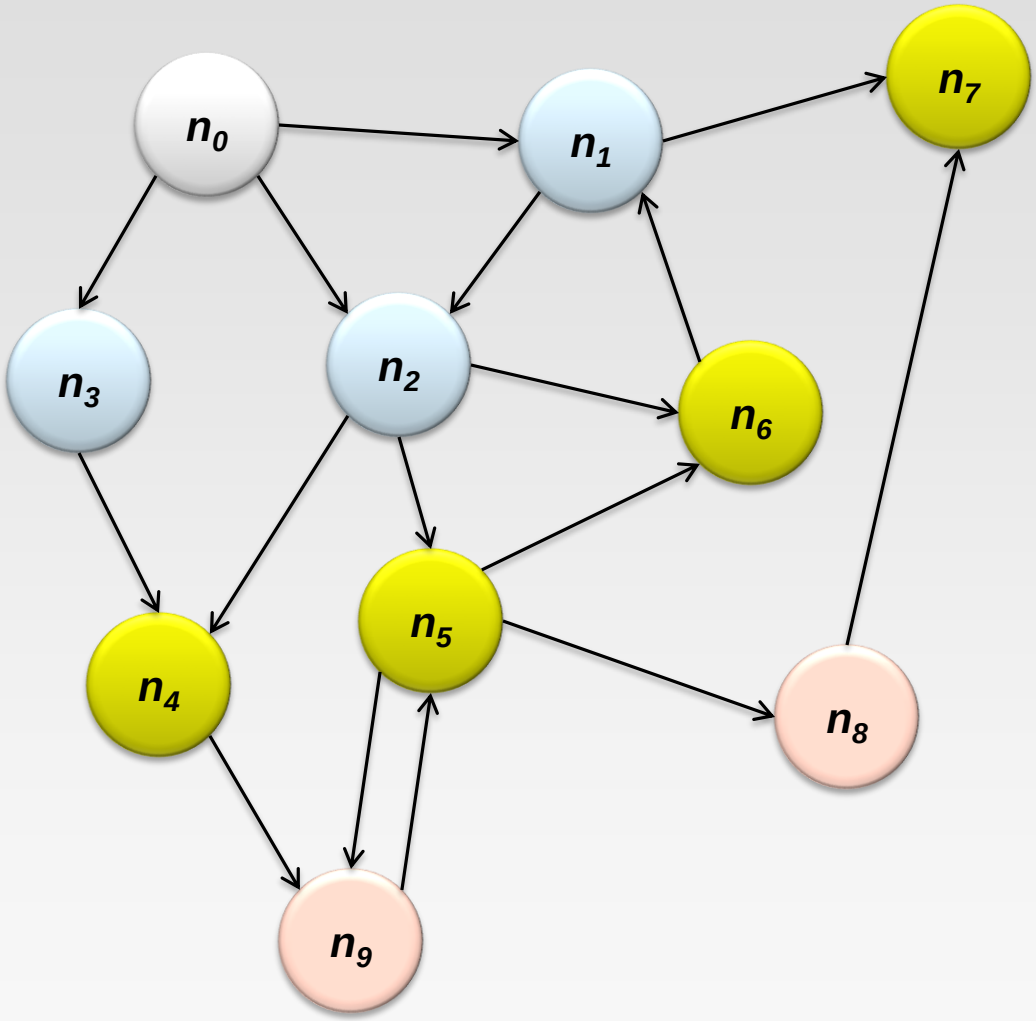
DISTANCETO(*s*) = 0

For all nodes *p* reachable from *s*,
DISTANCETO(*p*) = 1

For all nodes *n* reachable from some other set of nodes *M*,
DISTANCETO(*n*) = 1 + min(DISTANCETO(*m*), $m \in M$)



Visualizing Parallel BFS



From Intuition to Algorithm

Data representation:

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

Mapper:

- $\forall m \in$ adjacency list: emit (m, $d + 1$)

Sort/Shuffle

- Groups distances by reachable nodes

Reducer:

- Selects minimum distance path for each reachable node
- Additional bookkeeping needed to keep track of actual path

Multiple Iterations Needed

Each MapReduce iteration advances the “known frontier” by one hop

- Subsequent iterations include more and more reachable nodes as frontier expands
- The input of Mapper is the output of Reducer in the previous iteration
- Multiple iterations are needed to explore entire graph

Preserving graph structure:

- Problem: Where did the adjacency list go?
- Solution: mapper emits (*n*, adjacency list) as well

BFS Pseudo-Code

Equal Edge Weights (how to deal with weighted edges?)

Only distances, no paths stored (how to obtain paths?)

```
class Mapper
    method Map(nid n, node N)
    d ← N.Distance
    Emit(nid n,N)                          //Pass along graph structure
    for all nodeid m ∈ N.AdjacencyList do
        Emit(nid m, d+1)                   //Emit distances to reachable nodes
```

```
class Reducer
    method Reduce(nid m, [d1, d2, . . .])
    d_min ← ∞
    M ← ∅
    for all d ∈ counts [d1, d2, . . .] do
        if IsNode(d) then
            M ← d                          //Recover graph structure
        else if d < d_min then             //Look for shorter distance
            d_min ← d
    M.Distance ← d_min                     //Update shortest distance
    Emit(nid m, node M)
```

Stopping Criterion

How many iterations are needed in parallel BFS (equal edge weight case)?

Convince yourself: when a node is first “discovered”, we’ve found the shortest path

Now answer the question...

The diameter of the graph, or the greatest distance between any pair of nodes

Six degrees of separation?

- If this is indeed true, then parallel breadth-first search on the global social network would take at most six MapReduce iterations.

Until all nodes get valid distance

Implementation in MapReduce

The actual checking of the termination condition must occur outside of MapReduce.

The driver (main) checks to see if a termination condition has been met, and if not, repeats.

Hadoop provides a lightweight API called “counters”.

- It can be used for counting events that occur during execution, e.g., number of corrupt records, number of times a certain condition is met, or anything that the programmer desires.
- Counters can be designed to count the number of nodes that have distances of ∞ at the end of the job, the driver program can access the final counter value and check to see if another iteration is necessary.

MapReduce Counters

Instrument Job's metrics

Gather statistics

- Quality control – confirm what was expected.
  - E.g., count invalid records
- Application level statistics.

Problem diagnostics

Try to use counters for gathering statistics instead of log files

Framework provides a set of built-in metrics

For example bytes processed for input and output

User can create new counters

Number of records consumed

Number of errors or warnings

Built-in Counters

Hadoop maintains some built-in counters for every job.

Several groups for built-in counters

- File System Counters – number of bytes read and written
- Job Counters – documents number of map and reduce tasks launched, number of failed tasks
- Map-Reduce Task Counters– mapper, reducer, combiner input and output records counts, time and memory statistics

User-Defined Counters

You can create your own counters

Counters are defined by a Java enum

- serves to group related counters
- E.g.,

```
enum Temperature {
        MISSING,
        MALFORMED
}
```

Increment counters in Reducer and/or Mapper classes

Counters are global: Framework accurately sums up counts across all maps and reduces to produce a grand total at the end of the job

Implement User-Defined Counters

Retrieve Counter from Context object

Framework injects Context object into map and reduce methods

Increment Counter's value

Can increment by 1 or more

```
parser.parse(value);
if (parser.isValidTemperature()) {
  int airTemperature = parser.getAirTemperature();
  context.write(new Text(parser.getYear()),
      new IntWritable(airTemperature));
} else if (parser.isMalformedTemperature()) {
  System.err.println("Ignoring possibly corrupt input: " + value);
  context.getCounter(Temperature.MALFORMED).increment(1);
} else if (parser.isMissingTemperature()) {
  context.getCounter(Temperature.MISSING).increment(1);
}
```

Implement User-Defined Counters

Get Counters from a finished job in Java

Counter counters = job.getCounters()

Get the counter according to name

Counter c1 = counters.findCounter(Temperature.MISSING)

Enumerate all counters after job is completed

```
for (CounterGroup group : counters) {
        System.out.println("* Counter Group: " + group.getDisplayName() + " (" +
        group.getName() + ")");
        System.out.println("  number of counters in this group: " + group.size());
        for (Counter counter : group) {
                System.out.println("  - " + counter.getDisplayName() + ": " +
                counter.getName() + ": "+counter.getValue());
        }
}
```

How to Find the Shortest Path?

The parallel breadth-first search algorithm only finds the shortest distances.

Store “back-pointers” at each node, as with Dijkstra's algorithm

- Not efficient to recover the path from the back-pointers

A simpler approach is to emit paths along with distances in the mapper, so that each node will have its shortest path easily accessible at all times

- The additional space requirement is acceptable

Mapper

```
public static class TheMapper extends Mapper<LongWritable, Text, LongWritable, Text> {
    public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
        Text word = new Text();
        String line = value.toString();//looks like 1 0 2:3:
        String[] sp = line.split(" ");//splits on space
        int distanceadd = Integer.parseInt(sp[1]) + 1;
        String[] PointsTo = sp[2].split(":");
        for(int i=0; i<PointsTo.length; i++){
            word.set("VALUE "+distanceadd);//tells me to look at distance value
            context.write(new LongWritable(Integer.parseInt(PointsTo[i])), word);
            word.clear(); }
        //pass in current node's distance (if it is the lowest distance)
        word.set("VALUE "+sp[1]);
        context.write( new LongWritable( Integer.parseInt( sp[0] ) ), word );
        word.clear();
        word.set("NODES "+sp[2]);//tells me to append on the final tally
        context.write( new LongWritable( Integer.parseInt( sp[0] ) ), word );
        word.clear();
    }
}
```

Reducer

```
public static class TheReducer extends Reducer<LongWritable, Text, LongWritable, Text> {
    public void reduce(LongWritable key, Iterable<Text> values, Context context) throws
IOException, InterruptedException {
        String nodes = "UNMODED";
        Text word = new Text();
        int lowest = INFINITY;//start at infinity

        for (Text val : values) {
            //looks like NODES/VALUES 1 0 2:3:, we need to use the first as a key
            String[] sp = val.toString().split(" ");//splits on space
            //look at first value
            if(sp[0].equalsIgnoreCase("NODES")){
                nodes = null;
                nodes = sp[1];
            }else if(sp[0].equalsIgnoreCase("VALUE")){
                int distance = Integer.parseInt(sp[1]);
                lowest = Math.min(distance, lowest);
            }
        }
        word.set(lowest+" "+nodes);
        context.write(key, word);
        word.clear();
    }
}
```

https://github.com/himank/Graph-Algorithm-MapReduce/blob/master/src/DijikstraAlgo.java

BFS Pseudo-Code (Weighted Edges)

The adjacency lists, which were previously lists of node ids, must now encode the edge distances as well

- Positive weights!

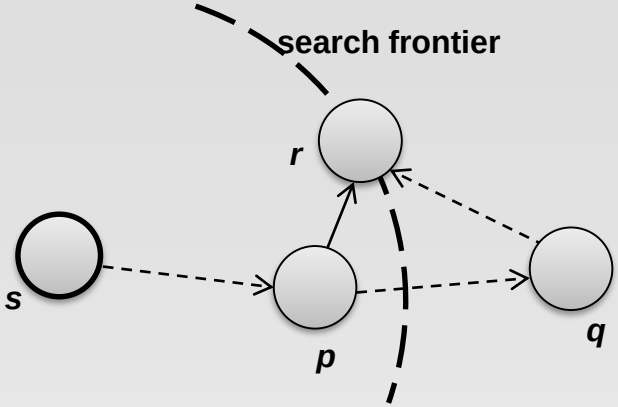
In line 6 of the mapper code, instead of emitting d + 1 as the value, we must now emit d + w, where w is the edge distance

The termination behaviour is very different!

- How many iterations are needed in parallel BFS (positive edge weight case)?
- Convince yourself: when a node is first "discovered", we've found the shortest path

Not true!

Additional Complexities



Assume that *p* is the current processed node

In the current iteration, we just “discovered” node r for the very first time.

We've already discovered the shortest distance to node *p*, and that the shortest distance to *r* so far goes through *p*

Is *s->p->r* the shortest path from *s* to *r*?

The shortest path from source *s* to node *r* may go outside the current search frontier

It is possible that *p->q->r* is shorter than *p->r*!

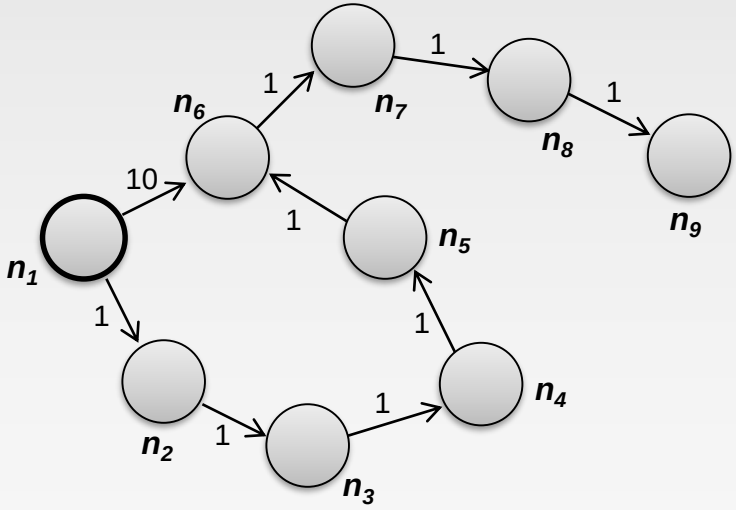
We will not find the shortest distance to *r* until the search frontier expands to cover *q*.

How Many Iterations Are Needed?

In the worst case, we might need as many iterations as there are nodes in the graph minus one updated # nodes - 1

A sample graph that elicits worst-case behaviour for parallel breadth-first search.

Eight iterations are required to discover shortest distances to all nodes from n_1.



Example (only distances)

Input file:

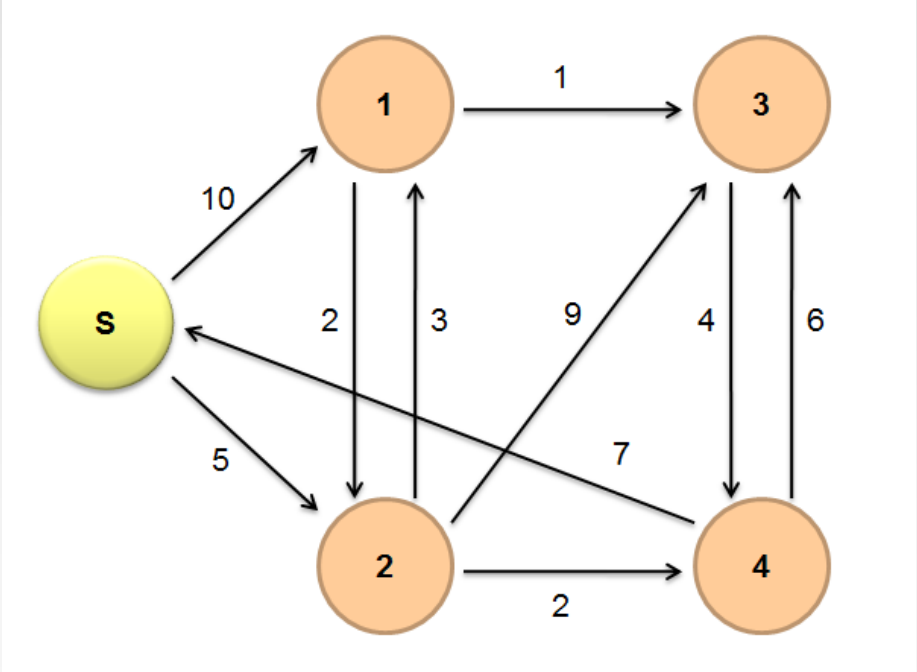
s --> 0 | n1: 10, n2: 5

n1 --> ∞ | n2: 2, n3:1

n2 --> ∞ | n1: 3, n3:9, n4:2

n3 --> ∞ | n4:4

n4 --> ∞ | s:7, n3:6



Iteration 1

Map:

Read s --> 0 | n1: 10, n2: 5

Emit: (n1, 10), (n2, 5), and the adjacency list (s, n1: 10, n2: 5)

The other lists will also be read and emit, but they do not contribute, and thus ignored

Reduce:

Receives: (n1, 10), (n2, 5), (s, <0, (n1: 10, n2: 5)>)

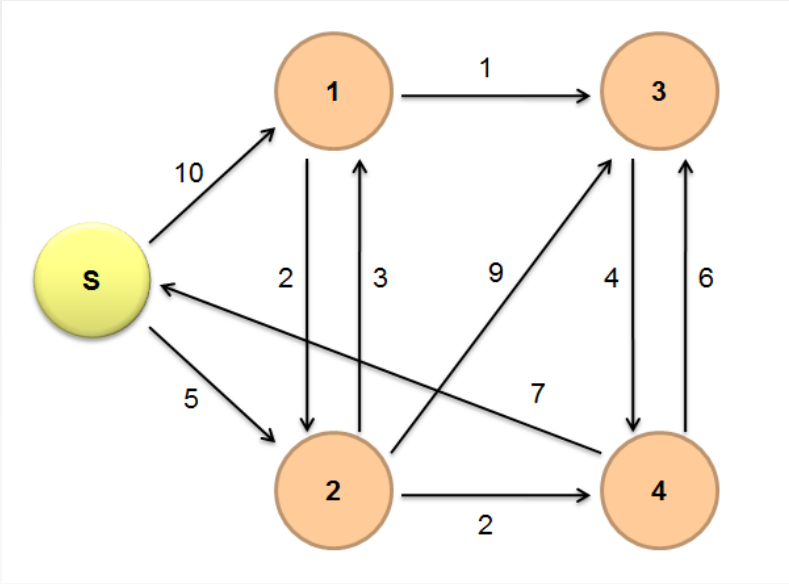
The adjacency list of each node will also be received, ignored in example

Emit:

s --> 0 | n1: 10, n2: 5

n1 --> 10 | n2: 2, n3:1

n2 --> 5 | n1: 3, n3:9, n4:2



Iteration 2

Map:

Read: n1 --> 10 | n2: 2, n3:1

Emit: (n2, 12), (n3, 11), (n1, <10, (n2: 2, n3:1)>)

Read: n2 --> 5 | n1: 3, n3:9, n4:2

Emit: (n1, 8), (n3, 14), (n4, 7), (n2, <5, (n1: 3, n3:9, n4:2)>)

Ignore the processing of the other lists

Reduce:

Receives: (n1, (8, <10, (n2: 2, n3:1)>)), (n2, (12, <5, n1: 3, n3:9, n4:2>)), (n3, (11, 14)), (n4, 7)

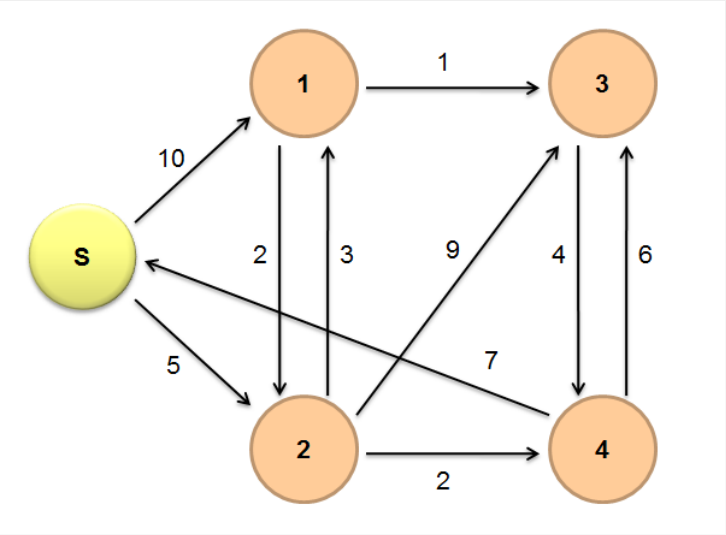
Emit:

n1 --> 8 | n2: 2, n3:1

n2 --> 5 | n1: 3, n3:9, n4:2

n3 --> 11 | n4:4

n4 --> 7 | s:7, n3:6



Iteration 3

Map:

Read: n1 --> 8 | n2: 2, n3:1

Emit: (n2, 10), (n3, 9), (n1, <8, (n2: 2, n3:1)>)

Read: n2 --> 5 | n1: 3, n3:9, n4:2 (**Again!**)

Emit: (n1, 8), (n3, 14), (n4, 7), (n2, <5, (n1: 3, n3:9, n4:2)>)

Read: n3 --> 11 | n4:4

Emit: (n4, 15), (n3, <11, (n4:4)>)

Read: n4 --> 7 | s:7, n3:6

Emit: (s, 14), (n3, 13), (n4, <7, (s:7, n3:6)>)

Reduce:

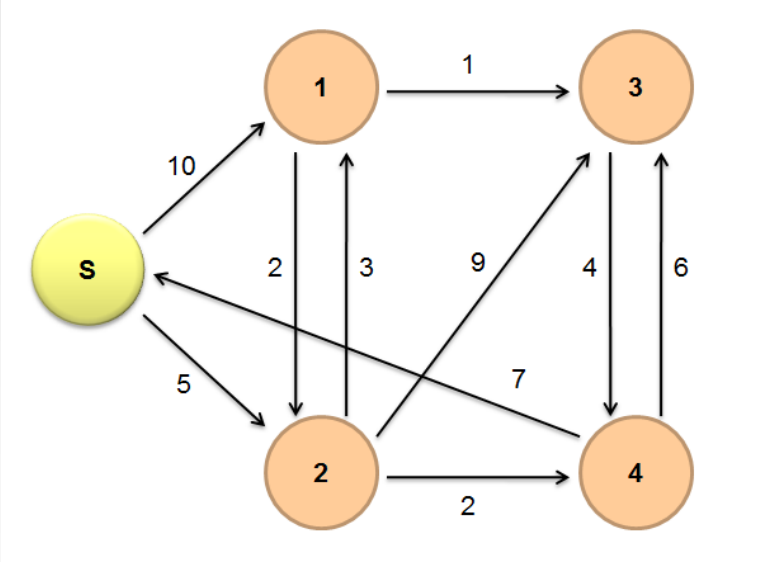
Emit:

n1 --> 8 | n2: 2, n3:1

n2 --> 5 | n1: 3, n3:9, n4:2

n3 --> 9 | n4:4

n4 --> 7 | s:7, n3:6



Iteration 4

In order to avoid duplicated computations, you can use a status value to indicate whether the distance of the node has been modified in the previous iteration.

Map:

Read: n1 --> 8 | n2: 2, n3:1 (**Again!**)

Emit: (n2, 10), (n3, 9), (n1, <8, (n2: 2, n3:1)>)

Read: n2 --> 5 | n1: 3, n3:9, n4:2 (**Again!**)

Emit: (n1, 8), (n3, 14), (n4, 7), (n2, <5, (n1: 3, n3:9, n4:2)>)

Read: n3 --> 9 | n4:4

Emit: (n4, 13), (n3, <9, (n4:4)>)

Read: n4 --> 7 | s:7, n3:6 (**Again!**)

Emit: (s, 14), (n3, 13), (n4, <7, (s:7, n3:6)>)

Reduce:

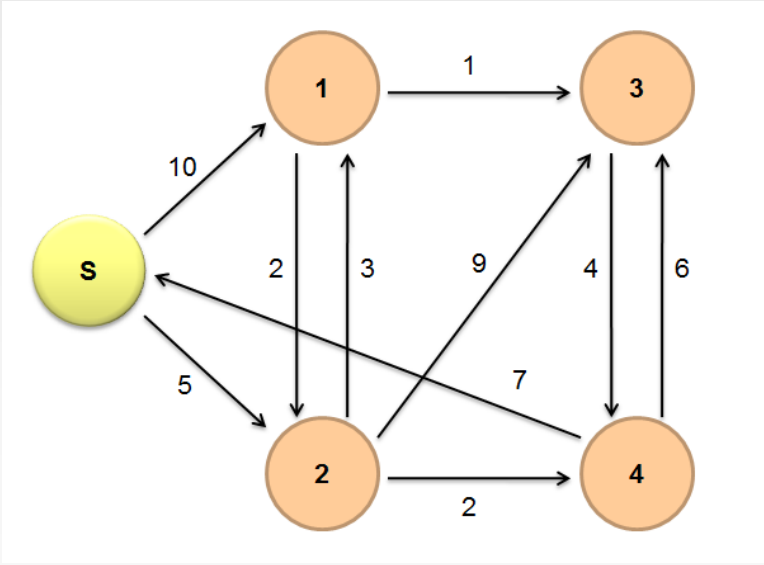
Emit:

n1 --> 8 | n2: 2, n3:1

n2 --> 5 | n1: 3, n3:9, n4:2

n3 --> 9 | n4:4

n4 --> 7 | s:7, n3:6



counter = 0

No updates. Terminate.

Comparison to Dijkstra

Dijkstra's algorithm is more efficient

- At any step it only pursues edges from the minimum-cost path inside the frontier

MapReduce explores all paths in parallel

- Lots of "waste"
- Useful work is only done at the "frontier"

Why can't we do better using MapReduce?

Graphs and MapReduce

Graph algorithms typically involve:

- Performing computations at each node: based on node features, edge features, and local link structure
- Propagating computations: “traversing” the graph

Generic recipe:

- Represent graphs as adjacency lists
- Perform local computations in mapper
- Pass along partial results via outlinks, keyed by destination node
- Perform aggregation in reducer on inlinks to a node
- Iterate until convergence: controlled by external “driver”
- Don’t forget to pass the graph structure between iterations

Issues with MapReduce on Graph Processing

MapReduce Does not support iterative graph computations:

External driver. Huge I/O incurs

No mechanism to support global data structures that can be accessed and updated by all mappers and reducers

- Passing information is only possible within the local graph structure – through adjacency list
- Dijkstra's algorithm on a single machine: a global priority queue that guides the expansion of nodes
- Dijkstra‘s algorithm in Hadoop, no such queue available. Do some “wasted” computation instead

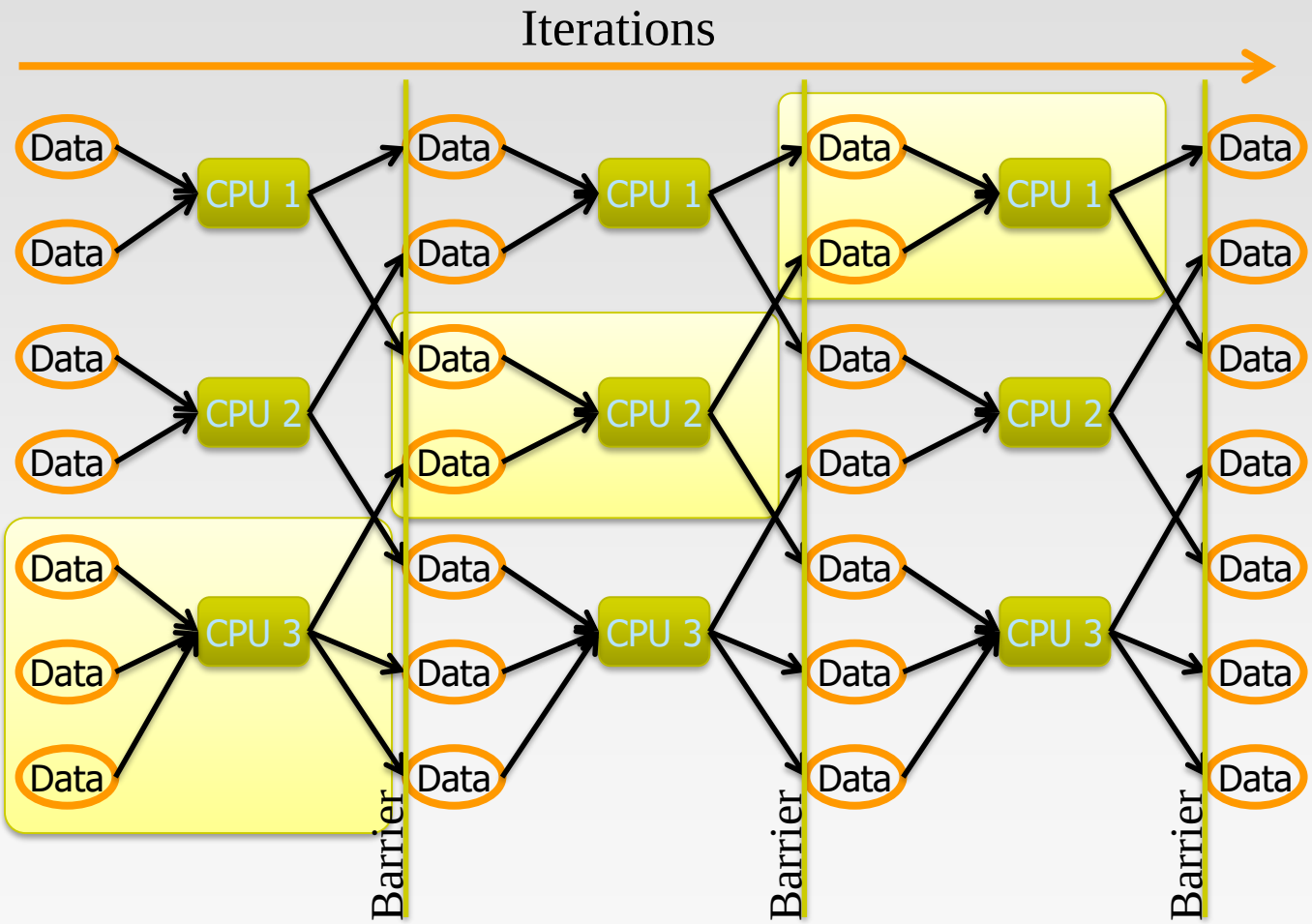
MapReduce algorithms are often impractical on large, dense graphs.

The amount of intermediate data generated is on the order of the number of edges.

For dense graphs, MapReduce running time would be dominated by copying intermediate data across the network.

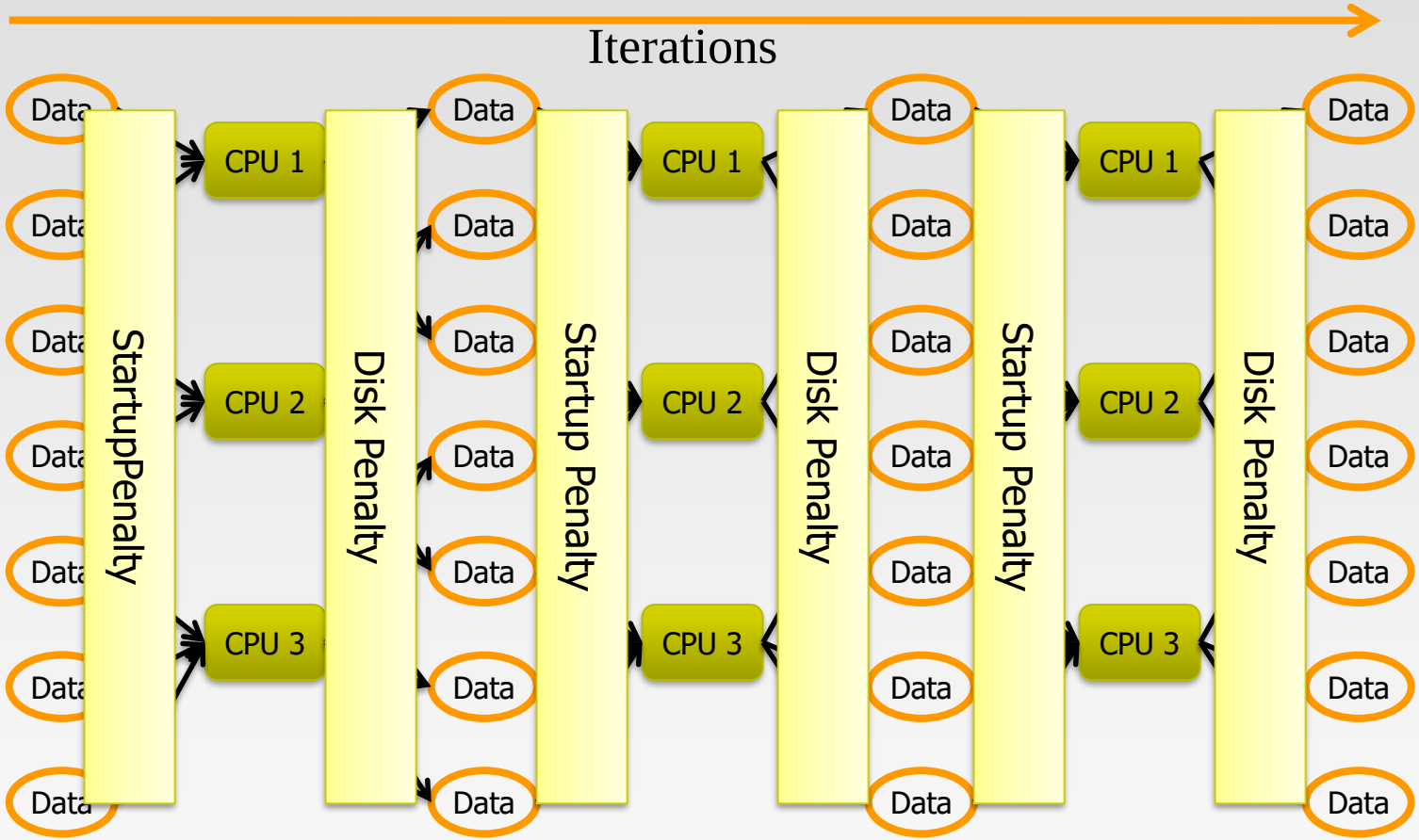
Iterative MapReduce

Only a subset of data needs computation:



Iterative MapReduce

System is not optimized for iteration:



Better Partitioning

Default: hash partitioning

- Randomly assign nodes to partitions

Observation: many graphs exhibit local structure

- E.g., communities in social networks
- Better partitioning creates more opportunities for local aggregation

Unfortunately, partitioning is **hard**!

- Sometimes, chick-and-egg…
- But cheap heuristics sometimes available
- For webgraphs: range partition on domain-sorted URLs

Graphs and MapReduce

Graph algorithms typically involve:

- Performing computations at each node: based on node features, edge features, and local link structure
- Propagating computations: “traversing” the graph

Generic recipe:

- Represent graphs as adjacency lists
- Perform local computations in mapper
- Pass along partial results via outlinks, keyed by destination node
- Perform aggregation in reducer on inlinks to a node
- Iterate until convergence: controlled by external “driver”
- Don’t forget to pass the graph structure between iterations

Issues with MapReduce on Graph Processing

MapReduce Does not support iterative graph computations:

External driver. Huge I/O incurs

No mechanism to support global data structures that can be accessed and updated by all mappers and reducers

- Passing information is only possible within the local graph structure – through adjacency list
- Dijkstra's algorithm on a single machine: a global priority queue that guides the expansion of nodes
- Dijkstra‘s algorithm in Hadoop, no such queue available. Do some “wasted” computation instead

MapReduce algorithms are often impractical on large, dense graphs.

The amount of intermediate data generated is on the order of the number of edges.

For dense graphs, MapReduce running time would be dominated by copying intermediate data across the network.

MapReduce Advantages

Automatic Parallelization:

- Depending on the size of RAW INPUT DATA ➔ instantiate multiple MAP tasks
- Similarly, depending upon the number of intermediate <key, value> partitions ➔ instantiate multiple REDUCE tasks

Run-time:

- Data partitioning
- Task scheduling
- Handling machine failures
- Managing inter-machine communication

Completely transparent to the programmer/analyst/user

Methods to Write MapReduce Jobs

Typical – usually written in Java

- MapReduce 2.0 API
- MapReduce 1.0 API

Streaming

- Uses stdin and stdout
- Can use any language to write Map and Reduce Functions
  - C#, Python, JavaScript, etc…

Pipes

- Often used with C++

Abstraction libraries

- Hive, Pig, etc… write in a higher level language, generate one or more MapReduce jobs

Number of Maps and Reduces

Maps

The number of maps is usually driven by the total size of the inputs, that is, the total number of blocks of the input files.

The right level of parallelism for maps seems to be around 10-100 maps per-node, although it has been set up to 300 maps for very cpu-light map tasks.

If you expect 10TB of input data and have a blocksize of 128MB, you'll end up with 82,000 maps, unless Configuration.set(MRJobConfig.NUM_MAPS, int) (which only provides a hint to the framework) is used to set it even higher.

Reduces

The right number of reduces seems to be 0.95 or 1.75 multiplied by (<*no. of nodes*> * <*no. of maximum containers per node*>)

With 0.95 all of the reduces can launch immediately and start transferring map outputs as the maps finish. With 1.75 the faster nodes will finish their first round of reduces and launch a second wave of reduces doing a much better job of load balancing.

Use job.setNumReduceTasks(int) to set the number

References

Data-Intensive Text Processing with MapReduce. Jimmy Lin and Chris Dyer. University of Maryland, College Park.

Hadoop The Definitive Guide. Hadoop I/O, and MapReduce Features chapters.

Chapter 5. Mining of Massive Datasets.

End of Chapter 4

Practices

Practice: Design MapReduce Algorithms

Counting total enrollments of two specified courses

Input Files: A list of students with their enrolled courses

Jamie: COMP9313, COMP9318

Tom: COMP9331, COMP9313

… …

Mapper selects records and outputs initial counts

Input: Key – student, value – a list of courses

Output: (COMP9313, 1), (COMP9318, 1), …

Reducer accumulates counts

Input: (COMP9313, [1, 1, …]), (COMP9318, [1, 1, …])

Output: (COMP9313, 16), (COMP9318, 35)

Practice: Design MapReduce Algorithms

Remove duplicate records

Input: a list of records

2013-11-01 aa
2013-11-02 bb
2013-11-03 cc
2013-11-01 aa
2013-11-03 dd

Mapper

Input (record_id, record)

Output (record, “”)

- E.g., (2013-11-01 aa, “”), (2013-11-02 bb, “”), …

Reducer

Input (record, [“”, “”, “”, …])

- E.g., (2013-11-01 aa, [“”, “”]), (2013-11-02 bb, [“”]), …

Output (record, “”)

Practice: Design MapReduce Algorithms

Assume that in an online shopping system, a huge log file stores the information of each transaction. Each line of the log is in format of "userID\t product\t price\t time". Your task is to use MapReduce to find out the top-5 expensive products purchased by each user in 2016

Mapper:

Input(transaction_id, transaction)

initialize an associate array H(UserID, priority queue Q of log record based on price)

map(): get local top-5 for each user

cleanup(): emit the entries in H

not applicable if user purchase a user several times

Reducer:

Input(userID, list of queues[])

get top-5 products from the list of queues

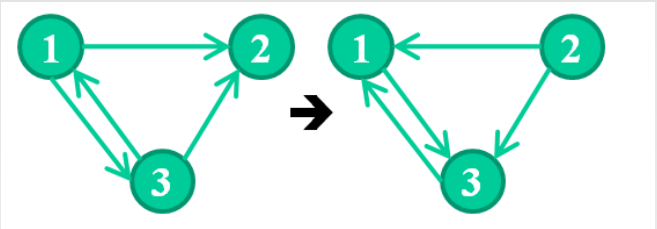
Practice: Design MapReduce Algorithms

Reverse graph edge directions & output in node order

Input: adjacency list of graph (3 nodes and 4 edges)

(3, [1, 2])
(1, [2, 3]) ➔

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



Note, the node_ids in the output values are also sorted. But Hadoop only sorts on keys!

Solutions: Secondary sort

Practice: Design MapReduce Algorithms

Map

Input: (3, [1, 2]), (1, [2, 3]).

Intermediate: (1, [3]), (2, [3]), (2, [1]), (3, [1]). (reverse direction)

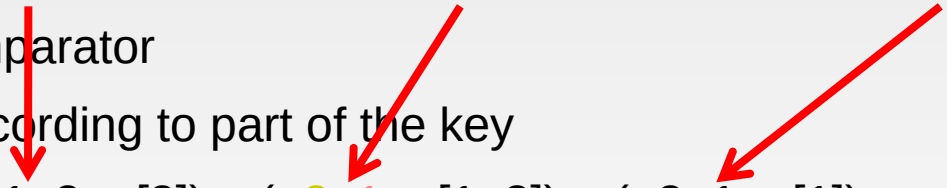
Output: (<1, 3>, [3]), (<2, 3>, [3]), (<2, 1>, [1]), (<3, 1>, [1]).

- Copy node_ids from value to key.

Partition on Key.field1, and Sort on whole Key (both fields)

Input: (<1, 3>, [3]), (<2, 3>, [3]), (<2, 1>, [1]), (<3, 1>, [1])

Output: (<1, 3>, [3]), (<2, 1>, [1]), (<2, 3>, [3]), (<3, 1>, [1])



Grouping comparator

Merge according to part of the key

Output: (<1, 3>, [3]), (<2, 1>, [1, 3]), (<3, 1>, [1])
this will be the reducer's input

Reducer

Merge according to part of the key

Output: (1, [3]), (2, [1, 3]), (3, [1])

Practice: Design MapReduce Algorithms

Calculate the common friends for each pair of users in Facebook. Assume the friends are stored in format of Person->[List of Friends], e.g.: A -> [B C D], B -> [A C D E], C -> [A B D E], D -> [A B C E], E -> [B C D]. Your result should be like:

(A B) -> (C D)

(A C) -> (B D)

(A D) -> (B C)

(B C) -> (A D E)

(B D) -> (A C E)

(B E) -> (C D)

(C D) -> (A B E)

(C E) -> (B D)

(D E) -> (B C)

Practice: Design MapReduce Algorithms

Mapper:

Input(user u, List of Friends [f_1, f_2, …,])

map(): for each friend f_i, emit (<u, f_i>, List of Friends [f_1, f_2, …,])

Reducer:

Input(user u, list of friends lists[])

Get the intersection from all friends lists

Example: http://stevekrenzel.com/articles/finding-friends