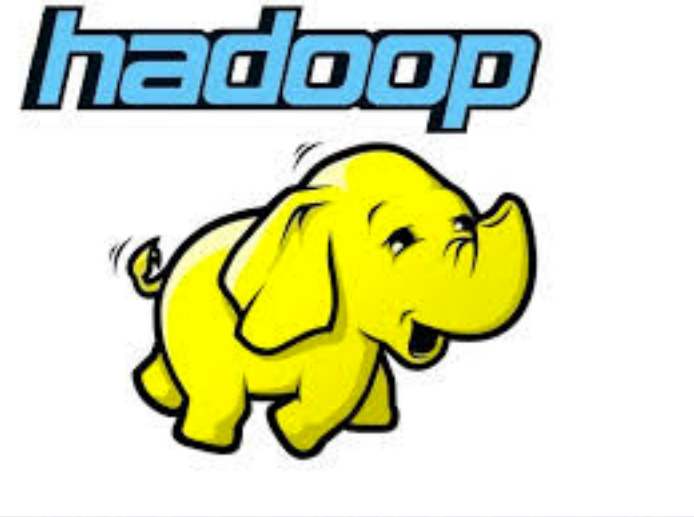


BIG DATA and AI
for business

MapReduce Programming

Decisions, Operations & Information Technologies
Robert H. Smith School of Business
Fall, 2020

What we'll cover…

UNIVERSITY OF MARYLAND
ROBERT H. SMITH SCHOOL OF BUSINESS

- **MapReduce**
  - Required components
    - Input, Mapper, Reducer, Driver, and Output
  - Optional components
    - Combiner and Partitioner
  - Customize components
    - Input / output
    - Data type
  - Chaining jobs
- **Examples**
  - Single-source shortest path for a large graph

MapReduce

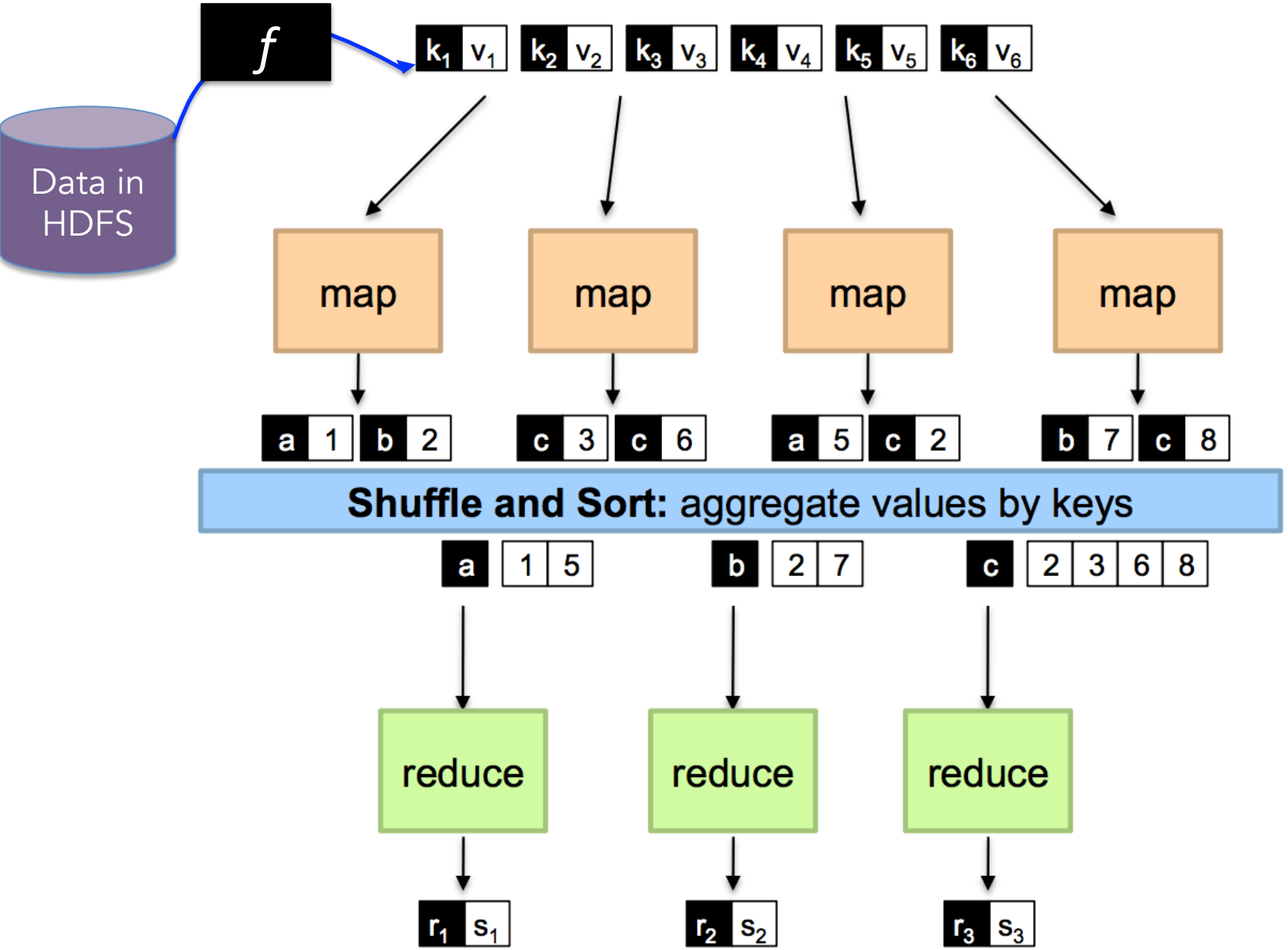
- **Everything** in MapReduce is <**key, value**> pair

- Programmers create/implement three functions/methods:

  map(key1, value1) → **[**<key2, value2>**]**

  reduce(key2, **[**value2**]**) → <key3, value3>

  driver() / main()


f
Data in HDFS
k1 v1
k2 v2
k3 v3
k4 v4
k5 v5
k6 v6
map
map
map
map
a 1
b 2
c 3
c 6
a 5
c 2
b 7
c 8
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 3 6 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

(Lin, WWW 2013)

MapReduce: map

- map(key1, value1) → [<key2, value2>]

MapReduce: map

- Input:
  - Key: byte offset of the line
  - Value: the content of the line
  - E.g., "I am taking big data course at UMD summer school 2018. I really enjoy summer time here at UMD."
- Output:
  - (I, 1), (am, 1), (taking, 1), (big, 1), (data, 1), (course, 1), (at, 1), (UMD, 1), (summer, 1), (school, 1), (2018, 1), (., 1), (I, 1), (really, 1), (enjoy, 1), (summer, 1), (time, 1), (here, 1), (at, 1), (UMD, 1), (., 1)

MapReduce: reduce

- reduce(key2, [value2]) → <key3, value3>

MapReduce: reduce

- Input:
  - (I, 1), (am, 1), (taking, 1), (big, 1), (data, 1), (course, 1), (at, 1), (UMD, 1), (summer, 1), (school, 1), (2018, 1), (., 1), (I, 1), (really, 1), (enjoy, 1), (summer, 1), (time, 1), (here, 1), (at, 1), (UMD, 1), (., 1)
- Output:
  - (I, 2), (am, 1), (taking, 1), (big, 1), (data, 1), (course, 1), (at, 2), (UMD, 2), (summer, 2), (school, 1), (2018, 1), (., 2), (really, 1), (enjoy, 1), (time, 1), (here, 1)

MapReduce: driver

- Driver in Hadoop (job configuration)
  - Specify where to load input and where to put output
  - Specify input format and output format
  - Specify mapper, combiner, partitioner, and reducer
  - Specify number of mappers, reducers, etc.
  - …

MapReduce

- Programmers create three functions/methods:

  map(key1, value1) → **[**<key2, value2>**]**

  reduce(key2, **[**value2**]**) → <key3, value3>

  driver() / main()

- The execution framework in Hadoop handles everything else…

What's "everything else"?

MapReduce "Runtime"

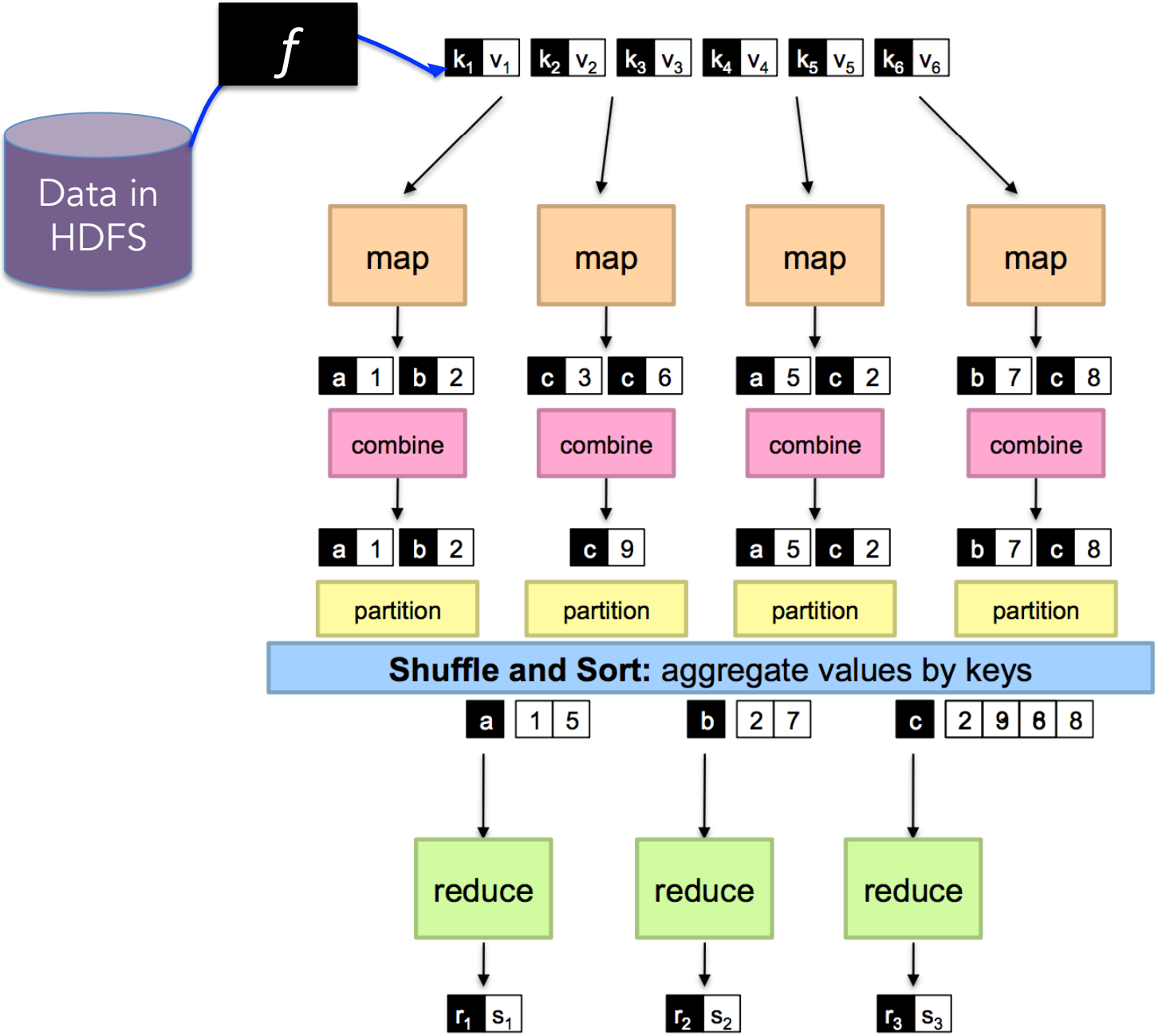
- Handles scheduling
  - Assigns mappers and reducers to do tasks
- Handles data distribution
  - Moves processes to data
- Handles synchronization
  - Collects, sorts, and shuffles intermediate data
- Handles errors and faults
  - Detects failures and restarts
- Everything happens on top of a distributed file system

Optional functions

- Programmers create three functions/methods
- The execution framework in Hadoop handles everything else…
- Usually, programmers also include the following two functions to optimize performance:

  combine (k, v) → <k, v′>
  - Mini-reducers that run in memory after the map phase
  - Used as an optimization to reduce network traffic

  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


f
Data in HDFS
k1 v1
k2 v2
k3 v3
k4 v4
k5 v5
k6 v6
map
map
map
map
a 1 b 2
c 3 c 6
a 5 c 2
b 7 c 8
combine
combine
combine
combine
a 1 b 2
c 9
a 5 c 2
b 7 c 8
partition
partition
partition
partition
Shuffle and Sort: aggregate values by keys
a 1 5
b 2 7
c 2 9 8 8
reduce
reduce
reduce
r1 s1
r2 s2
r3 s3

(Lin, WWW 2013)

Combiner

- In MapReduce framework, usually the output from the map tasks is large and data transfer between map and reduce tasks will be high
- Combiner functions summarize the map output records with the same key and output of combiner will be sent over network to actual reduce task as input
- The combiner does not have its own interface and **it must implement Reducer interface** and reduce() method of combiner will be called on each map output key
- reduce() method **must have the same input and output key-value types as the reducer class**
- **Hadoop doesn't guarantee on how many times a combiner will be called** for each map output key
- In most cases, the reducer is used as the combiner

Combiner

- (key, [value]) → <key, value'>

Partitioner

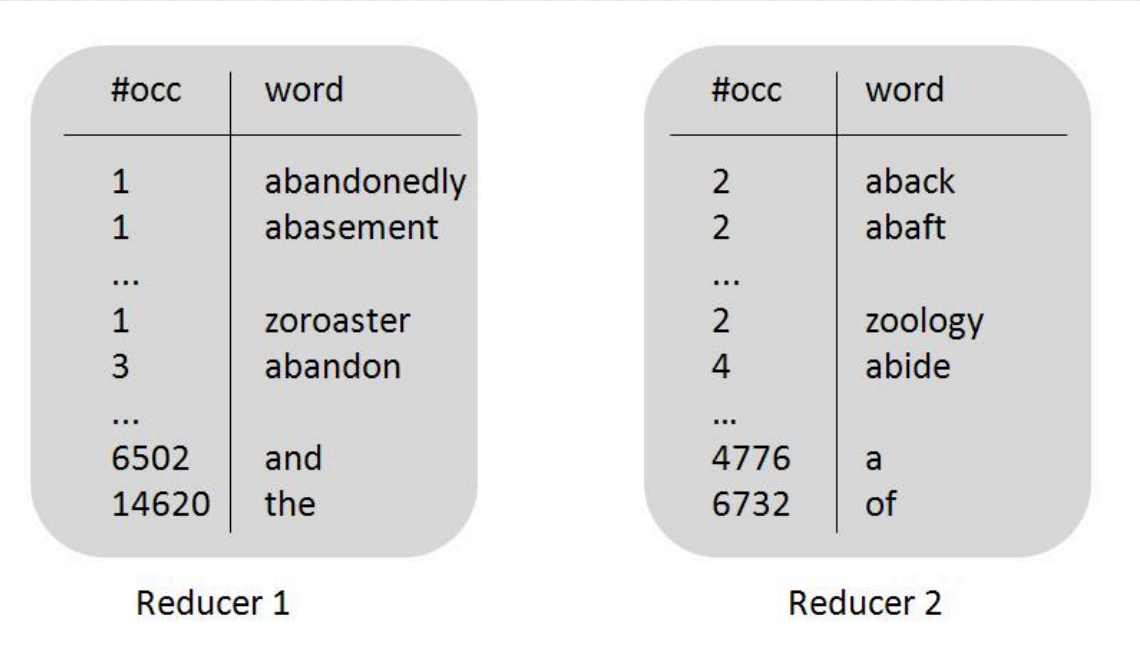
- The mechanism sending specific key-value pairs to specific reducers is called partitioning (the key-value pairs space is partitioned among the reducers)
- The default partitioner is *HashPartitioner*
  - Hashes a record's key to determine which partition (and thus which reducer) the record belongs in
  - The number of partition = the number of reduce tasks for the job

Why partition?

- It has a direct impact on the overall performance of your job: a poorly designed partitioning function will not evenly distributes the charge over the reducers, potentially loosing all the interest of the map/reduce distributed infrastructure.
- It maybe sometimes necessary to control the key/value pairs partitioning over the reducers

Example

- Your input is a (huge) set of tokens and their corresponding number of occurrences
- You want to sort them by number of occurrences



#occ	word
1	abandonedly
1	abasement
...	
1	zoroaster
3	abandon
...	
6502	and
14620	the

Reducer 1

#occ	word
2	aback
2	abaft
...	
2	zoology
4	abide
...	
4776	a
6732	of

Reducer 2

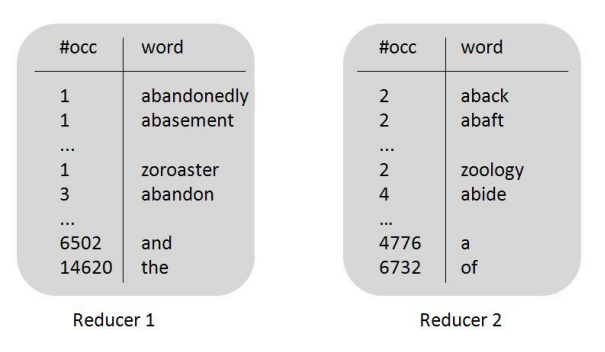
Case 1

- Your input is a (huge) set of tokens and their corresponding number of occurrences
- You want to sort them by number of occurrences
  - 95% tokens occur once
  - Using 2 reducers

- Customized partitioning function
  - Partition <#occ, token> pair instead of just #occ to balance load for two reducers

Case 2

- Your job's input is a (huge) set of tokens and their corresponding number of occurrences
- You want to sort them by number of occurrences

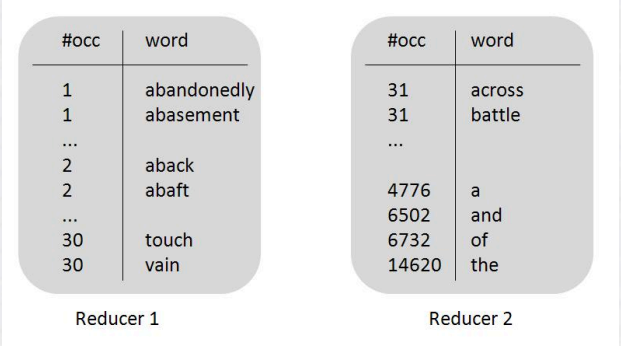


#occ	word
1	abandonedly
1	abasement
...	
1	zoroaster
3	abandon
...	
6502	and
14620	the

Reducer 1

#occ	word
2	aback
2	abaft
...	
2	zoology
4	abide
...	
4776	a
6732	of

Reducer 2



#occ	word
1	abandonedly
1	abasement
...	
2	aback
2	abaft
...	
30	touch
30	vain

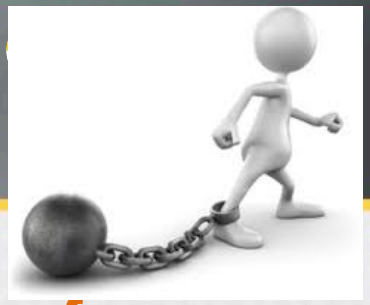
Reducer 1

#occ	word
31	across
31	battle
...	
4776	a
6502	and
6732	of
14620	the

Reducer 2

- Customized partitioning function
  - All the tokens having a number of occurrences inferior to N (here 30) are sent to reducer 1 and the others are sent to reducer 2, resulting in two partitions

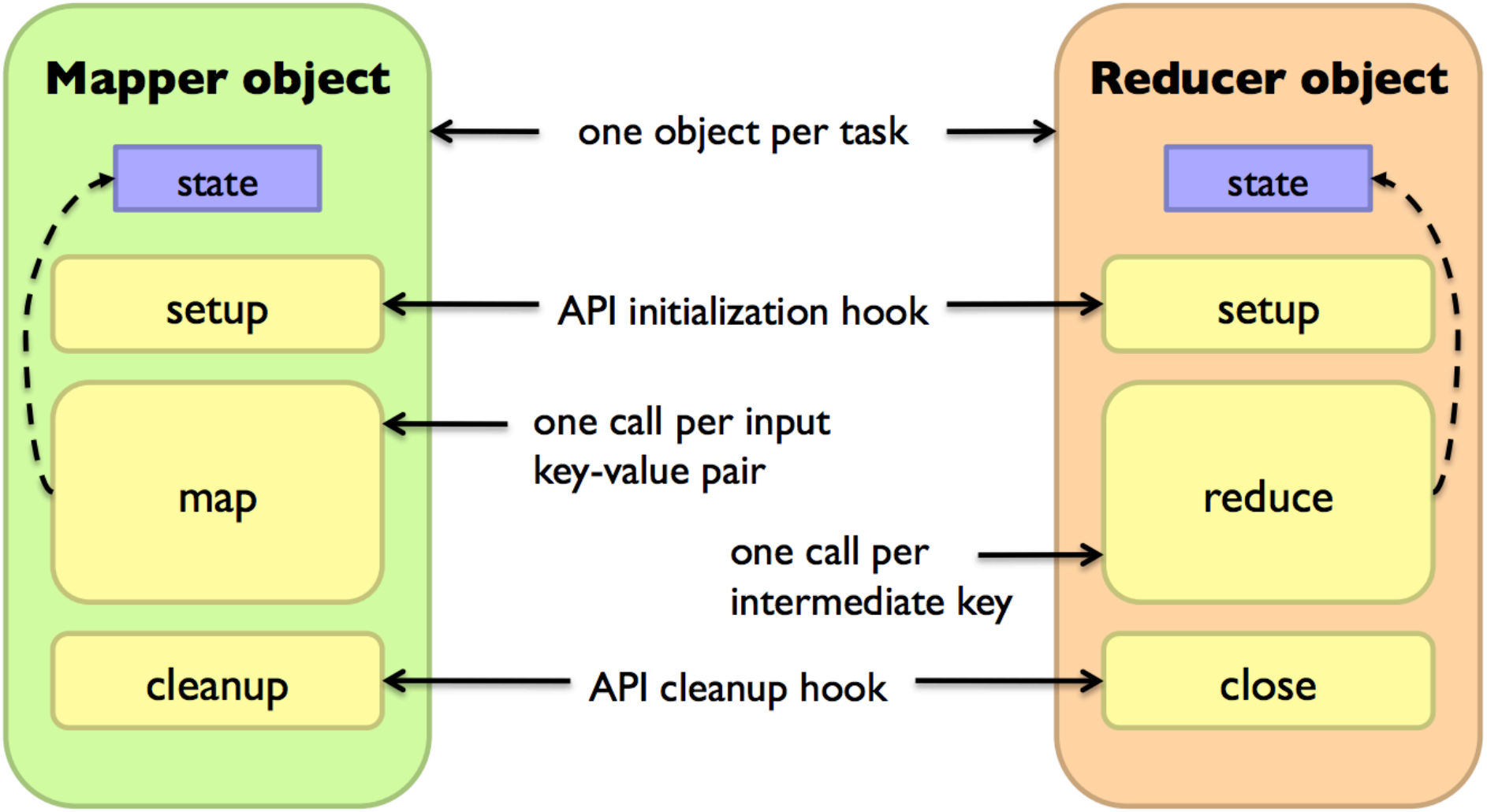
More about MapReduce…



- **The same key must be sent to the same reducer for reduce tasks**
- **Key arrived at each reducer in a sorted order**
  - No enforced ordering across reducers
- **The number of partitions is based on the number of reducers**

- Limitations
  - Limited control over data and execution flow
    - All algorithms must expressed in map, reduce, combine, or partition
  - You **CAN NOT** control:
    - Where mappers and reducers run
    - When a mapper or reducer begins or finishes
    - Which input a particular mapper is processing
    - Which intermediate key a particular reducer is processing

Mapper and Reducer objects



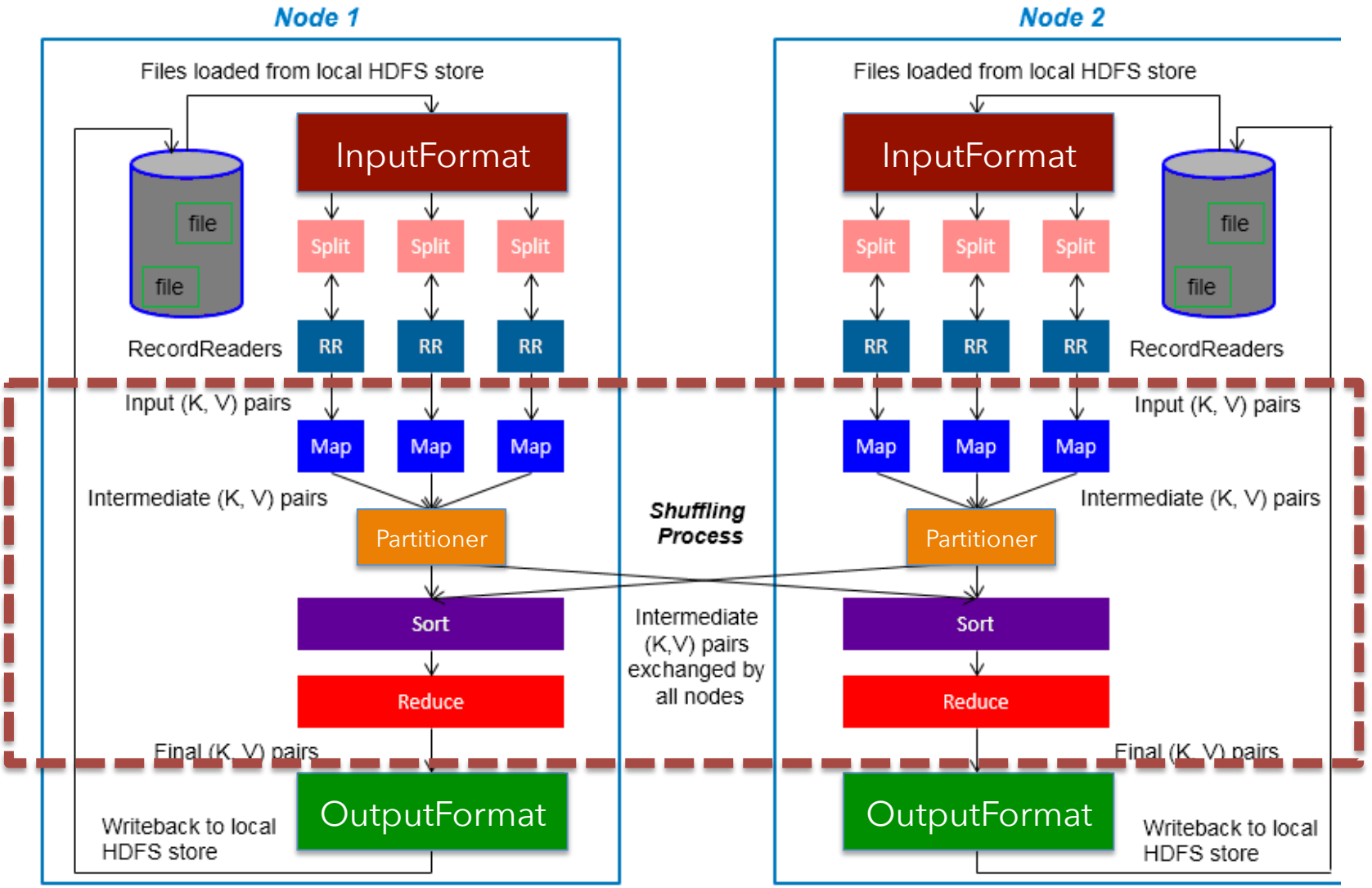
Setup() / Cleanup() methods

- Simply "hooks" for you to have a chance to do something **before** and **after** your map / reduce tasks
  - What typically happens during setup()
    - Read parameters from the configuration object to customize your processing logic
  - What typically happens during cleanup()
    - Clean up any resources you may have allocated
    - Flush out any accumulation of aggregate results

Setup() example

- For example, we want to exclude certain words from being counted (e.g. stop words such as "the", "a", "be", etc...).
- When we configure MapReduce Job, we can pass a list (comma-delimited) of these words as a parameter (key-value pair) into the configuration object.
- Then in the map code, during setup(), we can acquire stop words and store them in some global variable (global variable to the map task) and exclude counting these words during the map logic.

MapReduce flow



InputFormat

- How input files are split up and read is defined by the InputFormat Class
- **Three things**
  1. Validate the input configuration for the job (i.e., checking that the data is there)
  2. Split the input blocks and files into logical chunks of type *InputSplit*, each of which is assigned to a map task for processing
  3. Create the *RecordReader* implementation to be used to create key/value pairs from the raw InputSplit. These pairs are sent one by one to their mapper

Types of InputFormat

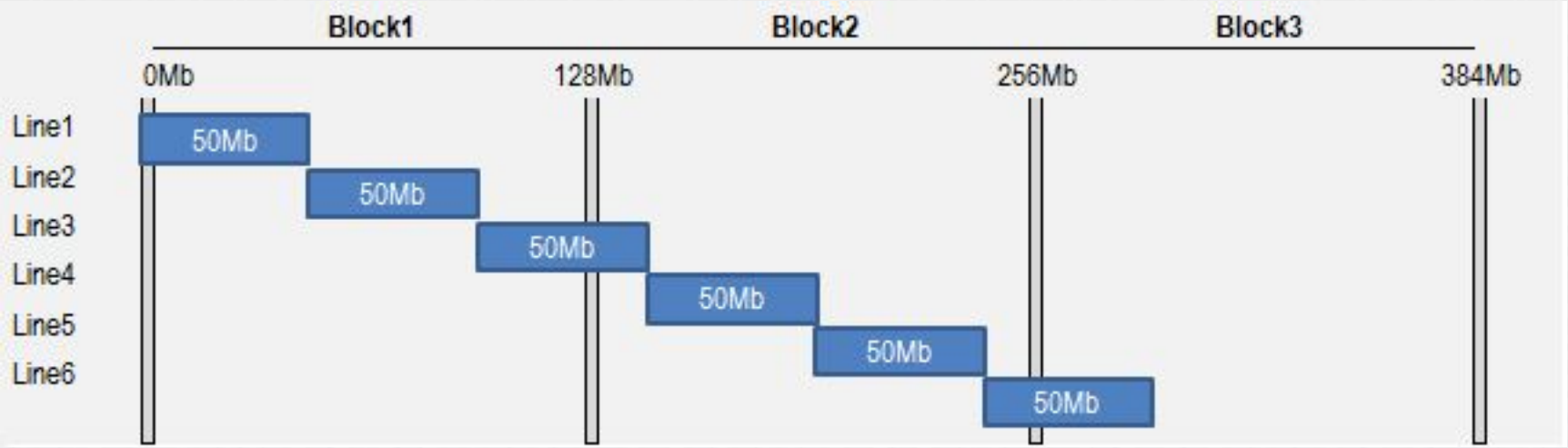
InputFormat	Description	Key	Value
TextInputFormat	Default format Read lines of text files	The byte offset of the line	The line contents
KeyValueInputFormat	Parses lines into key, value pairs	Everything up to the first tab	The reminder of the line
SequenceFileInputFormat	A hadoop-specific high performance binary format	User-defined	User-define
NLineInputFormat	Each split is guaranteed to have exactly N lines. Set by mapred.line.input.format.lineper map property	The byte offset of the first line of N lines	The contents of N lines

- **Usage:**
  - TextInputFormat is useful for unformatted data or line-based records like log file
  - KeyValueInputFormat is useful for reading the output of one MapReduce job as the input to another
  - SequenceFileInputFormat reads special binary files that are specific to Hadoop

InputSplits

- An InputSplit describes a unit of work that comprises a single mapper task in a MapReduce program
- **By default**, the FileInputFormat and its descendants break a file up to 128Mb chunks
  - mapred.min.split.size
  - Overriding this parameter in the JobConf object used in the driver class
- Map tasks are performed in parallel
  - mapred.tasktracker.map.tasks.maximum for on-node parallelism

InputSplits (1)



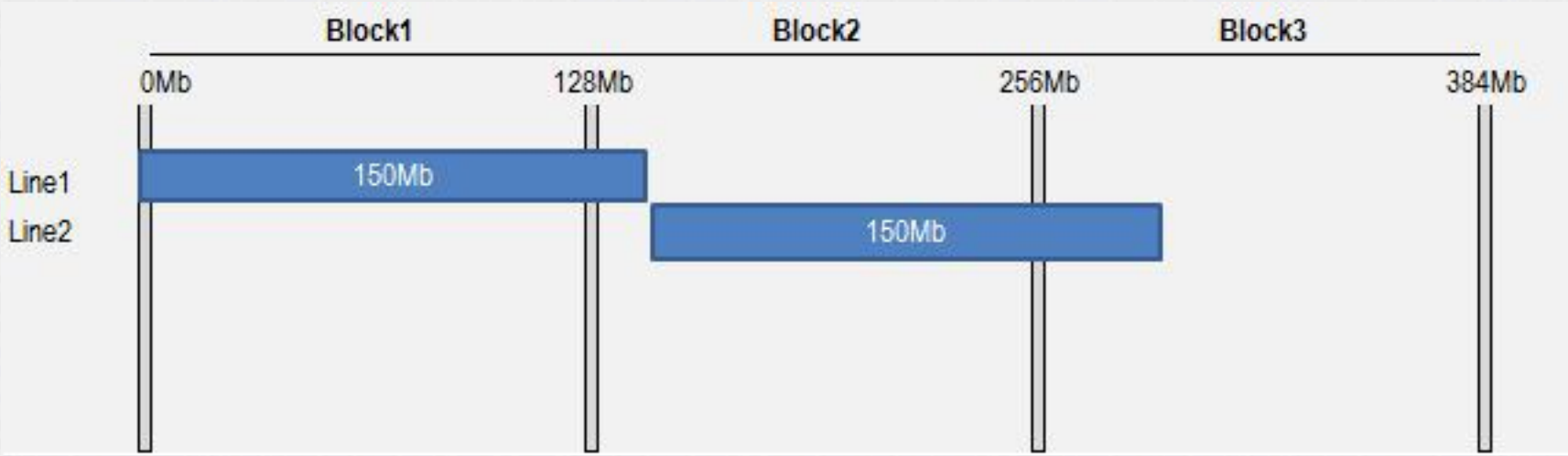
InputSplits (1)

The first Reader: First part of Line3 will be read locally from Block B1, second part will be read remotely from Block B2 (by the mean of FSDataInputStream), and a complete record will be finally sent as key / value to Mapper 1

The second Reader: starts on Block B2, at position 128Mb. Because 128Mb is not the start of a file, there are strong chance our pointer is located somewhere in an existing record that has been already processed by previous Reader. We need to skip this record by jumping out to the next available EOL, found at position 150Mb. Actual start of RecordReader 2 will be at 150Mb instead of 128Mb

	Start	Actual Start	End	Line(s)
Mapper1	B1:0	B1:0	B2:150	L1,L2,L3
Mapper2	B2:128	B2:150	B3:300	L4,L5,L6
Mapper3	B3:256	B3:300	B3:300	N/A

InputSplits (2)



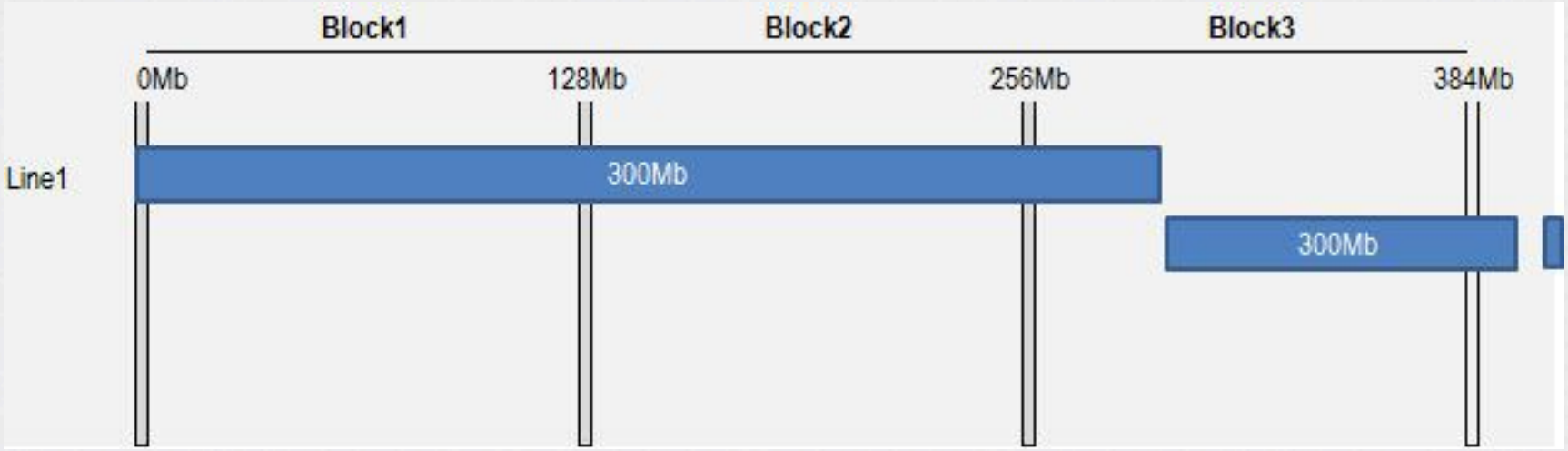
InputSplits (2)

Reader 1 will start reading from block B1, position 0. It will read line L1 locally until end of its split (128Mb), and will then continue reading remotely on B2 until EOL (150Mb)

Reader 2 will not start reading from 128Mb, but from 150Mb, and until B3:300

	Start	Actual Start	End	Line(s)
Mapper1	B1:0	B1:0	B2:150	L1
Mapper2	B2:128	B2:150	B3:300	L2
Mapper3	B3:256	B3:300	B3:300	N/A

InputSplits (3)



InputSplits (3)

Reader 1 will start reading locally from B1:0 until B1:128, then remotely all bytes available on B2, and finally remotely on B3 until EOL is reached (300Mb). There is here some overhead as we're trying to read a lot of data that is not locally available

Reader 2 will start reading from B2:128 and will jump out to next available record located at B3:300. Its new start position (B3:300) is actually greater than its maximum position (B2:256). This reader will therefore not provide Mapper 2 with any key / value

Reader 3 will start reading from B3:300 to B5:600

	Start	Actual Start	End	Line(s)
Mapper1	B1:0	B1:0	B3:300	L1
Mapper2	B2:128	B3:300	B2:256	N/A
Mapper3	B3:256	B3:300	B5:600	L2

RecordReader

It actually loads the data from its source and converts it into (key, value) paris

The default InputFormat: *TextInputFormat*, provides a line *RecordReader*, which treats each line of the input file as a new value

The *recordReader* is invoked repeatedly on the input until the entire InputSplit has been consumed

Each invocation of the *RecordReader* leads to a call to the *map()* function of the Mapper

Customized InputFormat

- **Subclass** the FileInputFormat class rather than implement InputFormat directly
- Override the ***createRecordReader()*** method, which returns an instance of ***RecordReader***: an object that can read from the input source

OutputFormat

- The (key, value) pairs provided to the context are then written to output files
- Each reducer writes to a separate file in a common output directory
- These files will typically be named part-nnnnn, where nnnnn is the partition id associated with the reduce task

Types of OutputFormat

OutputFormat	Description
TextOutputFormat	Default format; writes lines in "key \t value" format
SequenceFileOutputFormat	Writes binary files suitable for reading into subsequent MapReduce jobs
NullOutputFormat	Disregards its inputs

- The NullOutputFormat generates no output files and disregards any (key, value) paris passed to it by the context
- It is useful if you are **explicitly writing your own output files** in the reduce method, and do not want additional output files generated by the Hadoop framework

Data type

- Hadoop provides *writable* interface based data types for serialization and de-serialization of data storage in HDFS and mapreduce computations
  - Serialization: object data → byte stream data for transmission over a network across different nodes in a cluster **or** for persistent data storage
  - De-serialization: byte stream data →object data for reading data from HDFS

Constrains on Key-value

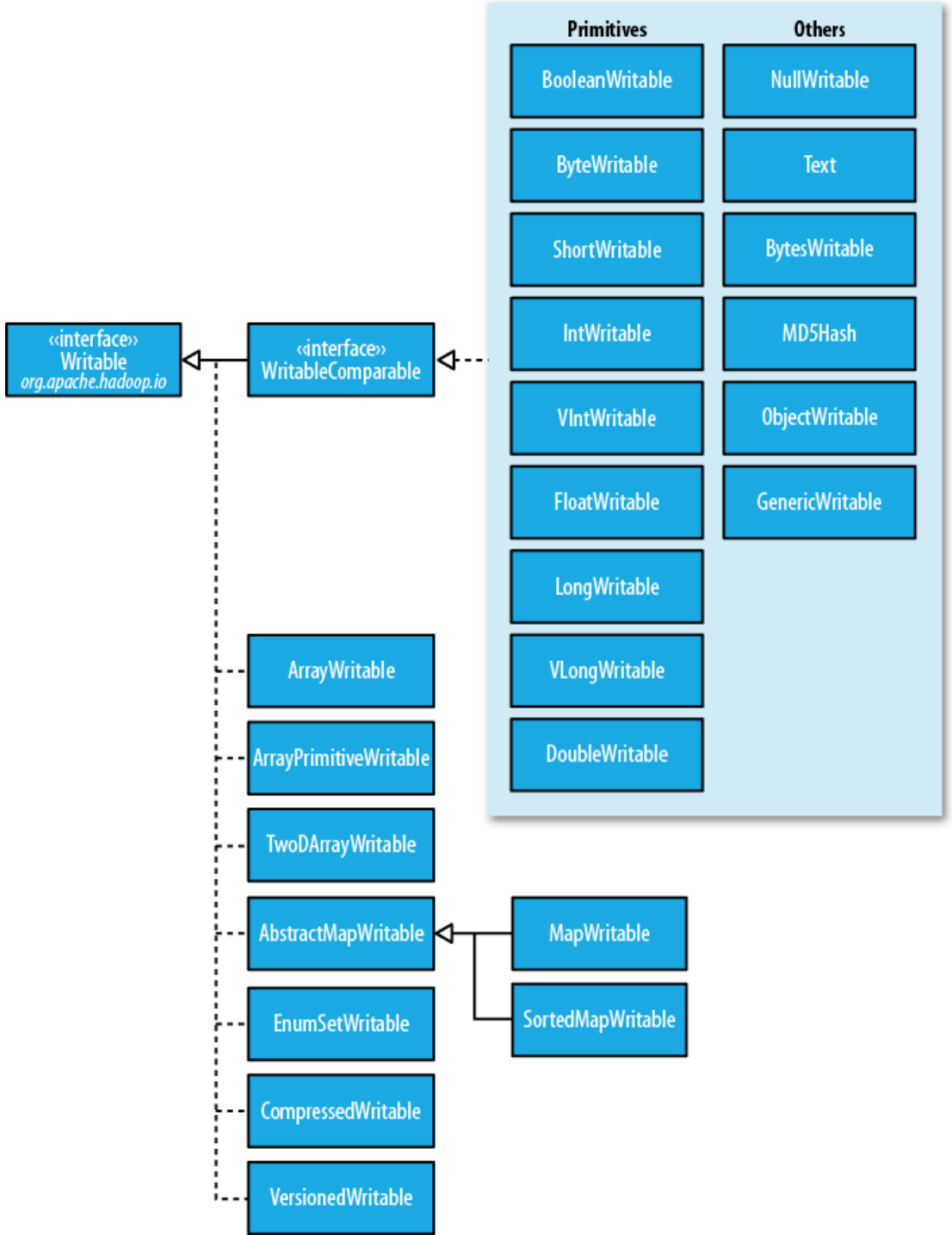
- Any data type used for a **Value** field in mapper or reducer input/output **must implement Writable** interface
- Any data type used for a **Key** field in mapper or reducer input/output must implement WritableComparable interface **along with Comparable** interface to compare the keys of this type with each other for sorting purposes

Writable classes

- All the Writable wrapper classes have a **get()** and a **set()** method for retrieving and sorting the wrapped value
- Primitive writable classes
  - BooleanWritable
  - ByteWritable
  - IntWritable
  - VIntWritable (variable length integer type)
  - FloatWritable
  - LongWritable
  - VLongWritable (variable length long type)
  - DoubleWritable

Writable classes

- Array writable classes
  - The elements of these arrays must be other writable objects like IntWritable or LongWritable
    - ArrayWritable
    - TwoDArrayWritable
- Map writable classes
- Other writable classes
  - NullWritable – special type of Writable representing a null value
  - TextWritable – Writable equivalent of java.lang.String and its max size is 2GB
  - …


Primitives
Others
BooleanWritable
NullWritable
ByteWritable
Text
ShortWritable
BytesWritable
IntWritable
MD5Hash
VIntWritable
ObjectWritable
FloatWritable
GenericWritable
LongWritable
VLongWritable
DoubleWritable
«interface»
Writable
org.apache.hadoop.io
«interface»
WritableComparable
ArrayWritable
ArrayPrimitiveWritable
TwoDArrayWritable
AbstractMapWritable
MapWritable
SortedMapWritable
EnumSetWritable
CompressedWritable
VersionedWritable

Customized data type example

- Assume we want an object representation of a vehicle to be the type of your key or value. Three properties of a vehicle has taken in to consideration (manufacturer, VIN, mileage)
- E.g.,

Toyota 1GCCS148X48370053 10000
Toyota 1FAPP64R1LH452315 40000
BMW WP1AA29P58L263510 10000
BMW JM3ER293470820653 60000
Nissan 3GTEC14V57G579789 10000
Nissan 1GNEK13T6YJ290558 25000
Honda 1GC4KVBG6AF244219 10000
Honda 1FMCU5K39AK063750 30000

Glossary

Term	Meaning
Job	The whole process to execute: the input data, the mapper and reducer execution and the output data
Task	Every job is divided among several mappers and reducers; a task is the job portion that goes to every single mapper and reducer
Split	The input file is split into several splits (the suggested size is the HDFS block size, 128 Mb)
Record	The split is read from mapper by default a line at a time: each line is a record. Using a class extending FileInputFormat, the record can be composed by more than one line
Partition	The set of all key-value pairs that will be sent to a single reducer. The default partitioner uses a hash function on the key to determine to which reducer send the data

Numbers

- How many mappers?
  - Exactly equals to the number of input splits
  - The split size is controlled by setting various Hadoop properties
    - mapred.min.split.size [minimumSize]
    - mapred.max.split.size [maximumSize]
    - dfs.block.size [blockSize]
  - split size is calculated by the formula: max(minimumSize, min(maximumSize, blockSize))
- How many reducers?
  - Specified by property or set in the job

Chaining jobs

- Many problems can be solved with MapReduce by writing **several MapReduce steps** which run in series to accomplish a goal
- Run the same mapper and reducer multiple times with slight alternations such as change of input and output files. Each iteration can use the previous iteration's output as input
  - map1->reduce1->map2->reduce2->…

Chaining methods (1)

- First create the JobConf object "job1" for the first job and set all parameters with "input" as input directory and "temp" as output directory. Execute this job: JobClient.run(job1)
- Immediately below it, create the JobConf object "job2" for the second job and set all parameters with "temp" as input directory and "output" as output directory. Finally execute second job: JobClient.run(job2)

Chaining methods (2)

- Create two JobConf objects and set all parameters in them just like (method (1)) except that you don't use JobClient.run
- Create two job objects with jobconfs as parameters:
  - Job job1=new Job(jobconf1)
  - Job job2=new Job(jobconf2)
- Using the jobControl object, you specify the job dependencies and then run the jobs
  - JobControl jbcntrl = new JobControl("jbcntrl")
  - jbcntrl.addJob(job1)
  - jbcntrl.addJob(job2)
  - job2.addDependingJob(job1)
  - jbcntrl.run()

Chaining methods (3)

UNIVERSITY OF MARYLAND
ROBERT H. SMITH SCHOOL OF BUSINESS

- Using **counters** and termination conditions

```
10/01/07 01:58:29 INFO mapred.JobClient: Running job: job_200912300823_0013
10/01/07 01:58:30 INFO mapred.JobClient:  map 0% reduce 0%
10/01/07 01:58:40 INFO mapred.JobClient:  map 100% reduce 0%
10/01/07 01:58:46 INFO mapred.JobClient:  map 100% reduce 100%
10/01/07 01:58:48 INFO mapred.JobClient: Job complete: job_200912300823_0013
10/01/07 01:58:48 INFO mapred.JobClient: Counters: 18
10/01/07 01:58:48 INFO mapred.JobClient:   Job Counters
10/01/07 01:58:48 INFO mapred.JobClient:     Launched reduce tasks=1
10/01/07 01:58:48 INFO mapred.JobClient:     Launched map tasks=2
10/01/07 01:58:48 INFO mapred.JobClient:     Data-local map tasks=2
10/01/07 01:58:48 INFO mapred.JobClient:   FileSystemCounters
10/01/07 01:58:48 INFO mapred.JobClient:     FILE_BYTES_READ=2521661
10/01/07 01:58:48 INFO mapred.JobClient:     HDFS_BYTES_READ=1259430
10/01/07 01:58:48 INFO mapred.JobClient:     FILE_BYTES_WRITTEN=5043392
10/01/07 01:58:48 INFO mapred.JobClient:     HDFS_BYTES_WRITTEN=366678
10/01/07 01:58:48 INFO mapred.JobClient:   Map-Reduce Framework
10/01/07 01:58:48 INFO mapred.JobClient:     Reduce input groups=33783
10/01/07 01:58:48 INFO mapred.JobClient:     Combine output records=0
10/01/07 01:58:48 INFO mapred.JobClient:     Map input records=22109
10/01/07 01:58:48 INFO mapred.JobClient:     Reduce shuffle bytes=2521667
10/01/07 01:58:48 INFO mapred.JobClient:     Reduce output records=33783
10/01/07 01:58:48 INFO mapred.JobClient:     Spilled Records=430274
10/01/07 01:58:48 INFO mapred.JobClient:     Map output bytes=2091381
10/01/07 01:58:48 INFO mapred.JobClient:     Map input bytes=1257289
10/01/07 01:58:48 INFO mapred.JobClient:     Combine input records=0
10/01/07 01:58:48 INFO mapred.JobClient:     Map output records=215137
10/01/07 01:58:48 INFO mapred.JobClient:     Reduce input records=215137
```

Counters

- Define your own counters to track any kind of statistics about the records you are manipulating in the mapper and the reducer
- Hadoop maintains some built-in counters for every job, which reports various metrics for your job
- User-defined counters that can be incremented or decremented by driver, mapper, or reducer
- Counters are global

Counters

- Declaring an enum representing your counters
- The enum name is the group of the counter, and each field of the enum is the name of the counter that will be reported in this same group
- Incrementing the desired counters from your map and reduce methods through the Context of your mapper or reducer

Counters

```
public class StringUtils {
        public static boolean startsWithDigit(String s){
                if( s == null || s.length() == 0 )
                        return false;
                return Character.isDigit(s.charAt(0));
        }
        public static boolean startsWithLetter(String s){
                if( s == null || s.length() == 0 )
                        return false;
                return Character.isLetter(s.charAt(0));
        }
}
```

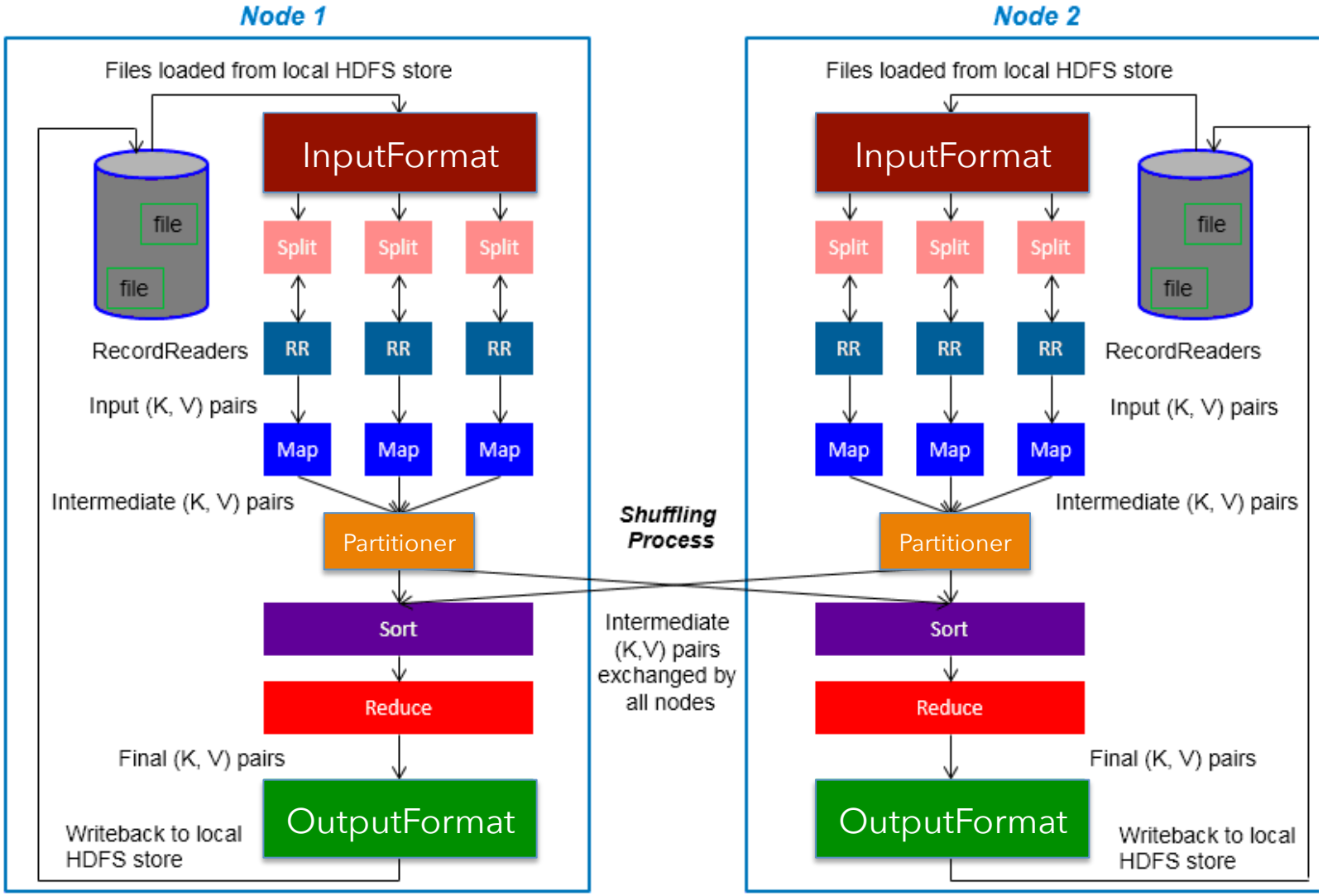
```
static enum WordsNature{
        STARTS_WITH_DIGIT, STARTS_WITH_LETTER, ALL
}
```

```
public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
        int sum = 0;
        String token = key.toString();

        if( StringUtils.startsWithDigit(token) ){
                context.getCounter(WordsNature.STARTS_WITH_DIGIT).increment(1);
        }
        else if( StringUtils.startsWithLetter(token) ){
                context.getCounter(WordsNature.STARTS_WITH_LETTER).increment(1);
        }
        context.getCounter(WordsNature.ALL).increment(1);

        for (IntWritable value : values) {
                sum += value.get();
        }
        context.write(key, new IntWritable(sum));
}
```

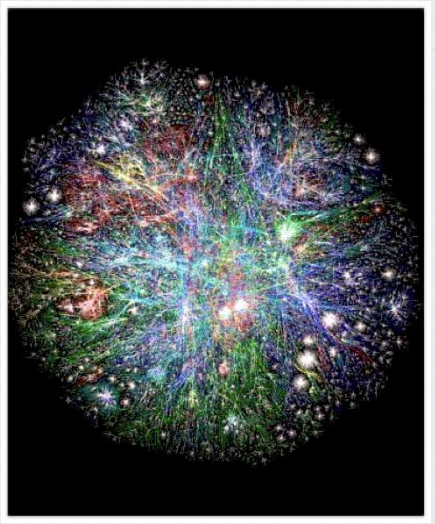
MapReduce flow




Iterative graph algorithm: single source shortest path

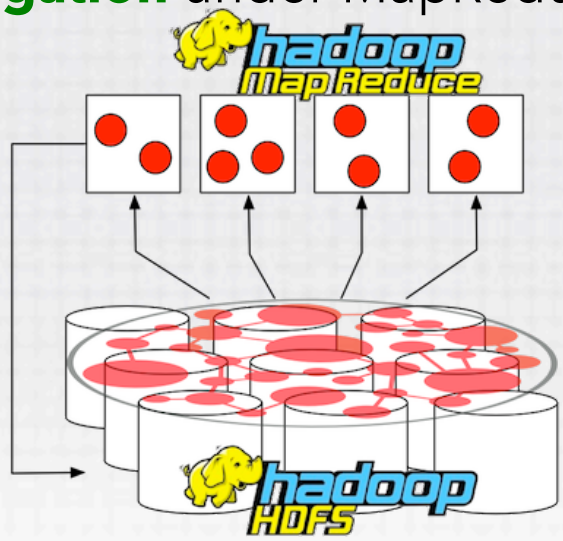
What is a graph?

- G = (V, E), where
  - V represents the set of nodes
  - E represents the set of links
  - Links and nodes may contain weights
- Different types of graphs
  - Directed vs. undirected
  - Weighted vs. unweighted
  - Acyclic vs. cyclic
- Networks/graphs are everywhere:
  - Social networks
  - Traffic networks
  - Web link networks



Graph and MapReduce

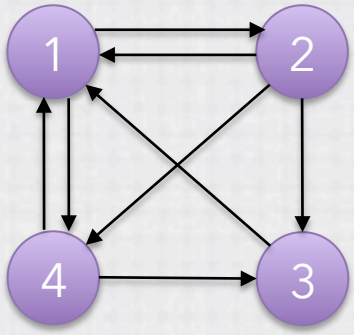
- Graph algorithms typically involves:
  - Computing something about nodes and links
  - Propagating information to the neighbors
- **Key issues:**
  - Graph **representation** under MapReduce framework
  - Information **propagation** under MapReduce framework



Graph representation

- Adjacency matrix

	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 list

1: 2, 4

2: 1, 3, 4

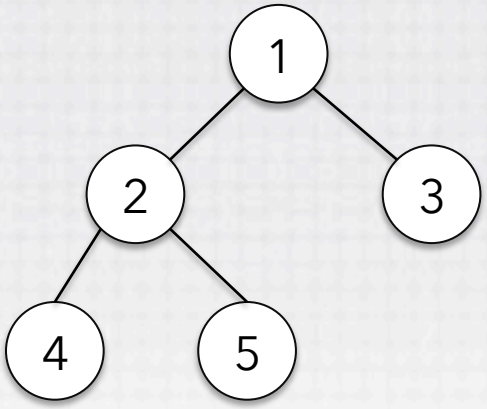
3: 1

4: 1, 3

Single source shortest path

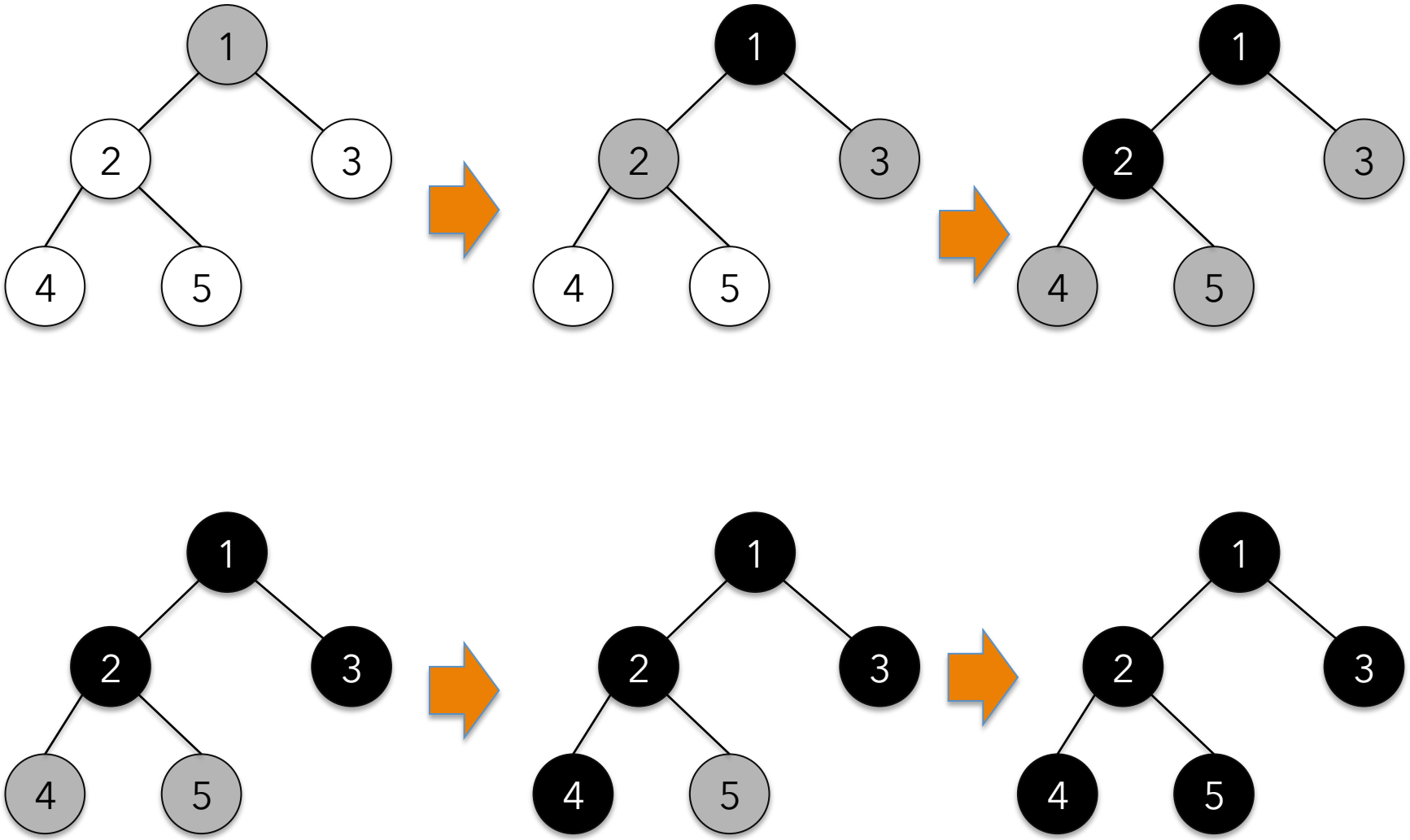
- **Problem**: find shortest path from a source node to one or more target nodes

- Single processor: Dijkstra's algorithm
- MapReduce: parallel breadth-first search (BFS)



Algorithm design

- One way of performing the BFS is by coloring the nodes and traversing according to the color of the nodes.
  - WHITE(unvisited), GRAY(visited) and BLACK(finished)
- At the beginning, all nodes are colored white.
- The source node is colored gray.
- The gray node indicates that it is visited and its neighbors should be processed.
- All the nodes adjacent to a gray node that are white are changed to be gray colored.
- The original gray node is colored.
- The process continues until there are no more gray nodes to process in the graph.


1
2
3
4
5
1
2
3
4
5
1
2
3
4
5
1
2
3
4
5
1
2
3
4
5
1
2
3
4
5

MapReduce design

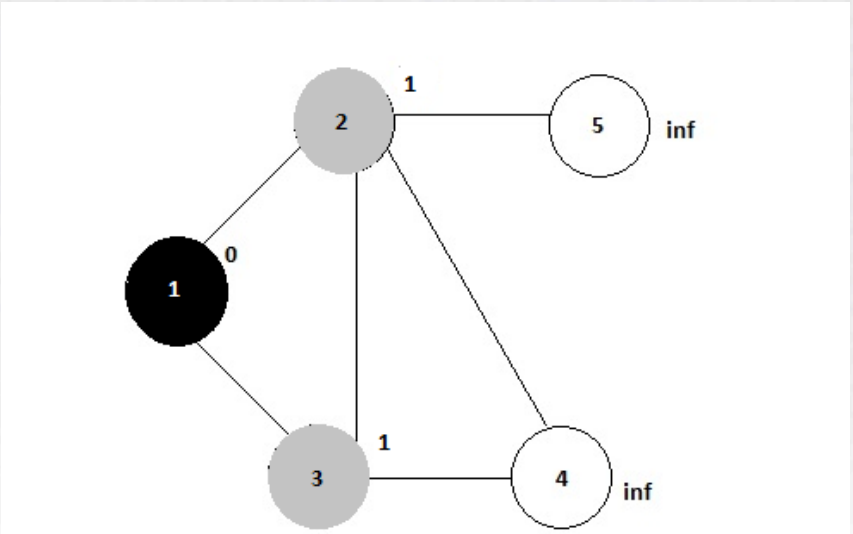
UNIVERSITY OF MARYLAND
ROBERT H. SMITH SCHOOL OF BUSINESS

- Data preparation (input format)
  - source<tab>adjacency list | distance from source | color | parent node
- Mapper
  - The nodes colored WHITE or BLACK are emitted as such
  - For each node that is colored GRAY, a new node is emitted with the distance incremented by one and the color set to GRAY. The original GRAY colored node is set to BLACK color and it is also emitted
- Reducer
  - Make a new node which combines all information for this single node id that is for each key. The new node should have the full list of edges, the minimum distance, the darkest color, and the parent/predecessor node

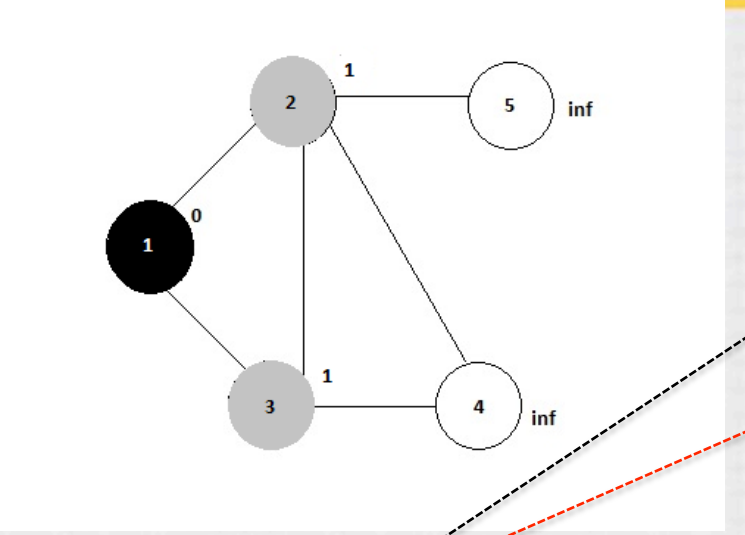
Example

- Input:

  1<tab>2,3,|0|BLACK|source
  2<tab>1,3,4,5,|1|GRAY|1
  3<tab>1,4,2,|1|GRAY|1
  4<tab>2,3,|Integer.MAX_VALUE|WHITE|null
  5<tab>2,|Integer.MAX_VALUE|WHITE|null



Mapper



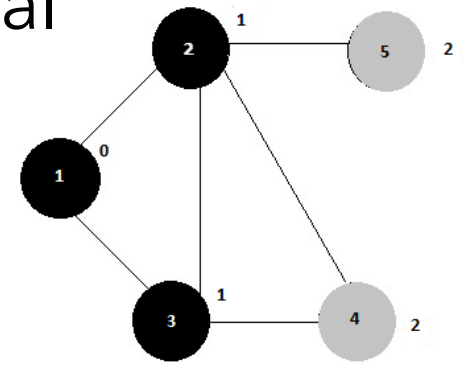
1<tab>2,3,|0|BLACK|source
2<tab>1,3,4,5,|1|GRAY|1
3<tab>1,4,2,|1|GRAY|1
4<tab>2,3,|Integer.MAX_VALUE|WHITE|null
5<tab>2,|Integer.MAX_VALUE|WHITE|null

1<tab>2,3,|0|BLACK|source
1<tab>2,3,|2|GRAY|2
3<tab>1,4,2,|2|GRAY|2
4<tab>2,3,|2|GRAY|2
5<tab>2,|2|GRAY|2
2<tab>1,3,4,5,|1|BLACK|1
1<tab>2,3,|2|GRAY|3
4<tab>2,3,|2|GRAY|3
2<tab>1,3,4,5,|2|GRAY|3
3<tab>1,4,2,|1|BLACK|1
4<tab>2,3,|Integer.MAX_VALUE|WHITE|null
5<tab>2,|Integer.MAX_VALUE|WHITE|null

Reducer

- For the same key, choose the darkest

and the minimum distance as the final value



1<tab>2,3,|0|BLACK|source
1<tab>2,3,|2|GRAY|2
3<tab>1,4,2,|2|GRAY|2
4<tab>2,3,|2|GRAY|2
5<tab>2,|2|GRAY|2
2<tab>1,3,4,5,|1|BLACK|1
1<tab>2,3,|2|GRAY|3
4<tab>2,3,|2|GRAY|3
2<tab>1,3,4,5,|2|GRAY|3
3<tab>1,4,2,|1|BLACK|1
4<tab>2,3,|Integer.MAX_VALUE|WHITE|null
5<tab>2,|Integer.MAX_VALUE|WHITE|null

1<tab>2,3,|0|BLACK|source
5<tab>2,|2|GRAY|2
2<tab>1,3,4,5,|1|BLACK|1
4<tab>2,3,|2|GRAY|**2**
3<tab>1,4,2,|1|BLACK|1