CSC 555 and DSC 333
Mining Big Data
Lecture 2

Alexander Rasin
College of CDM, DePaul University

September 21st, 2021

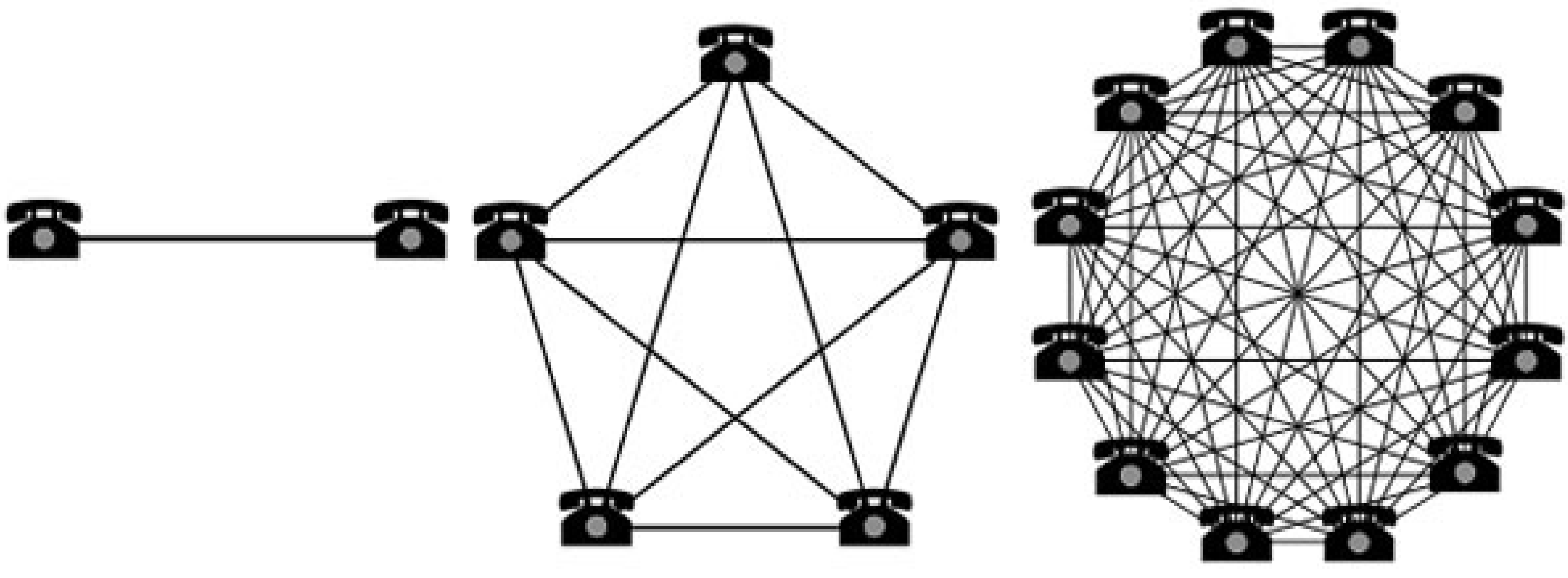
Tonight

- MapReduce/Hadoop
  - Distributed computing
  - Performance considerations
- Hashing
- SQL to MapReduce
- Hive

Distributed Computing

Parallel Computing is Hard

- Fundamental issues



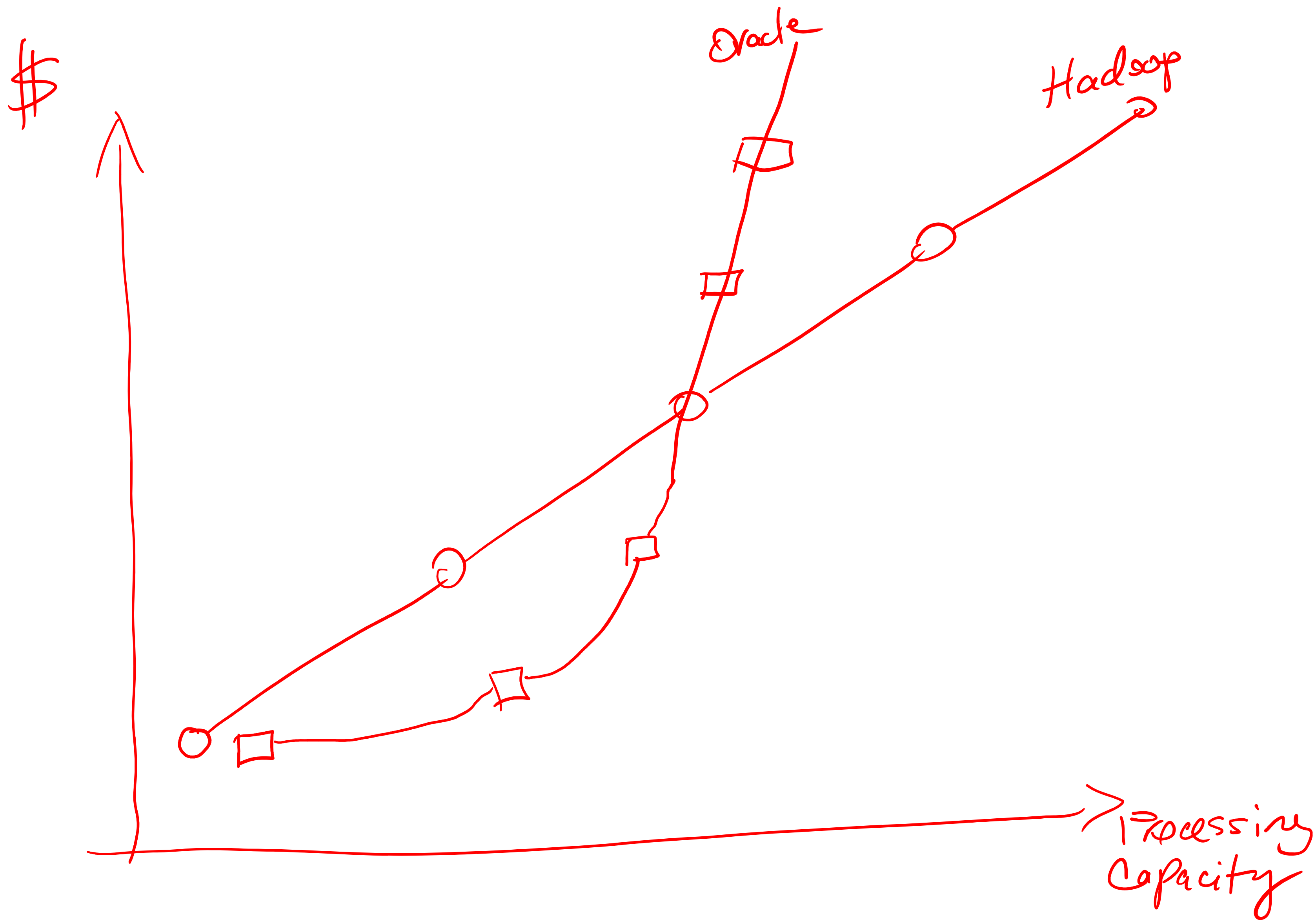
Scale-out vs Scale-up

Scale-out

- Many smaller servers
- Harder to manage
- Easier to expand
- Easier to tolerate failure (*)
- Generally cheaper

Scale-up

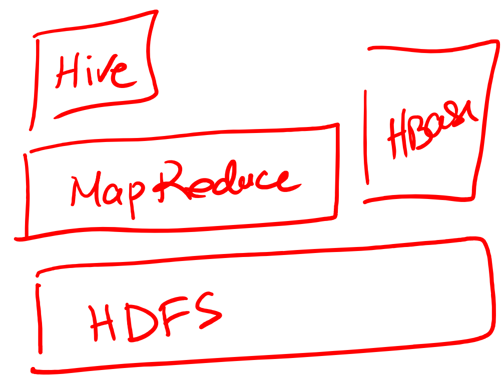
- Few large servers
- Easier to manage
- Difficult/expensive to expand
- Failure is a serious setback
- More expensive



$
Oracle
Hadoop
Processing
Capacity

Hadoop Execution

- Every job represented as
  - Map function
  - (Optional) Combine function
  - Reduce function

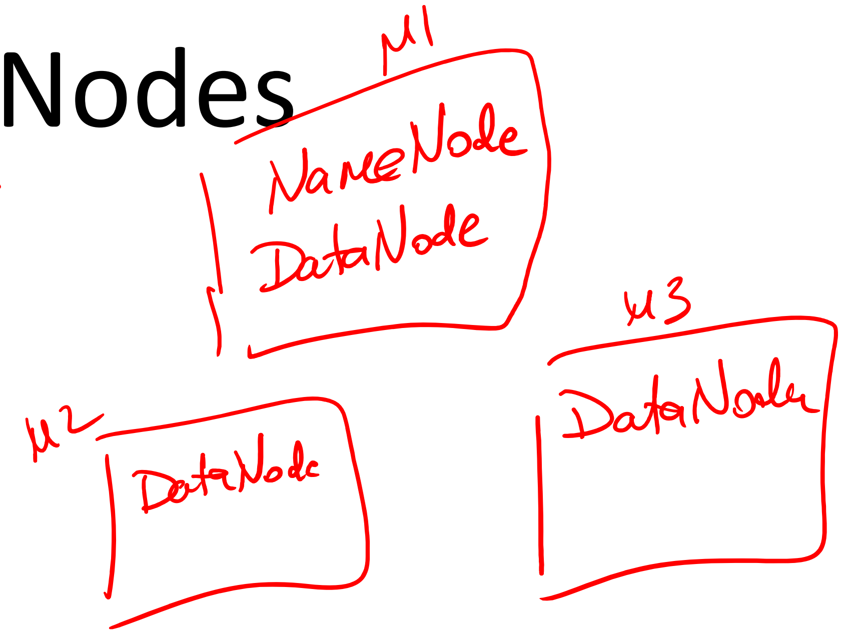


- Provides convenient failure/retry semantics
- Extensive Ecosystem
  - Hive, Pig, Spark, Storm, HBase, Mahout, Zookeeper, Sqoop, HUE, Oozie, Avro, Flume, …

Types of Nodes

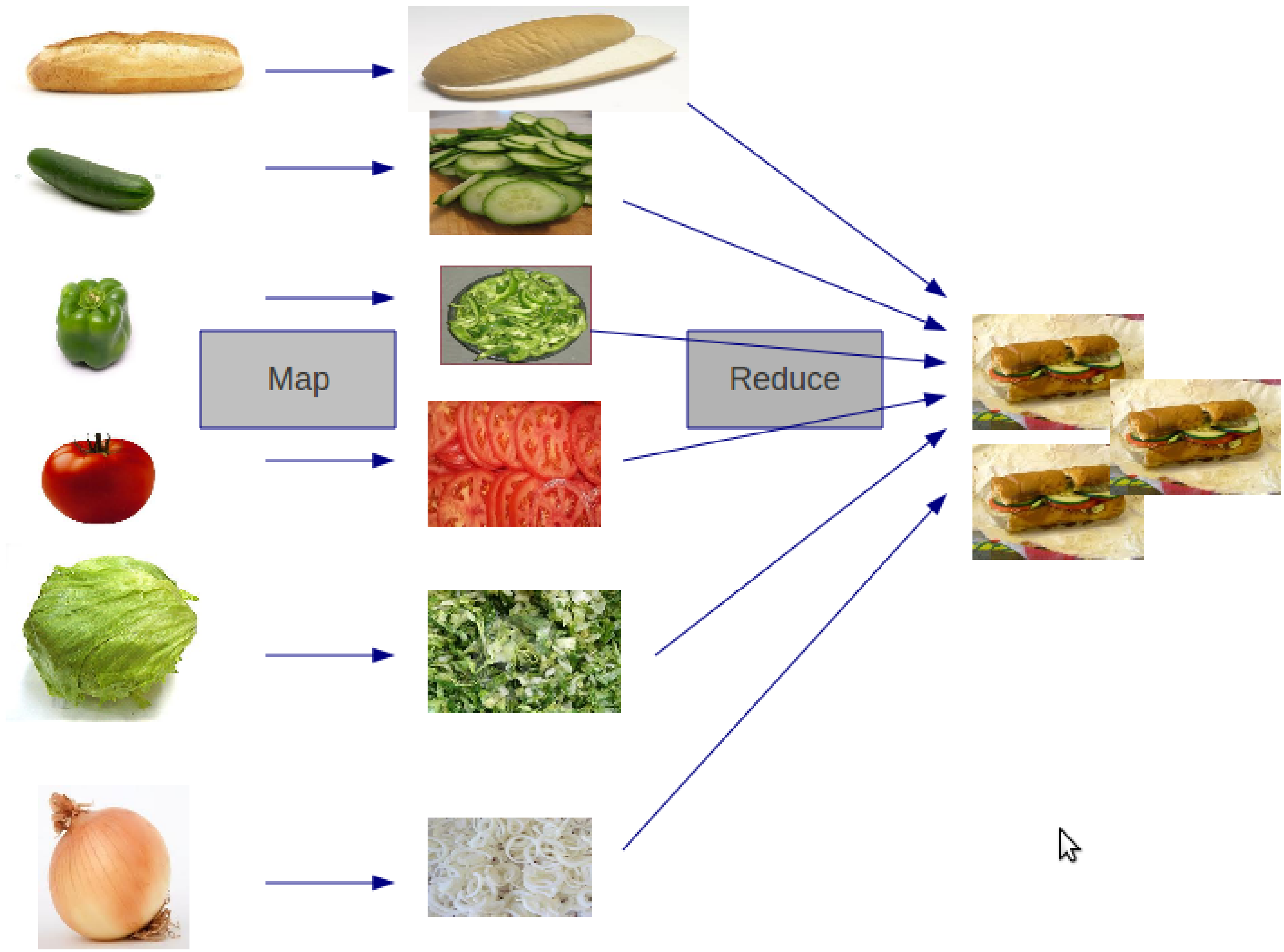
supervisor

- HDFS Nodes
  - NameNode
  - DataNode
- MapReduce Nodes
  - JobTracker
  - TaskTracker



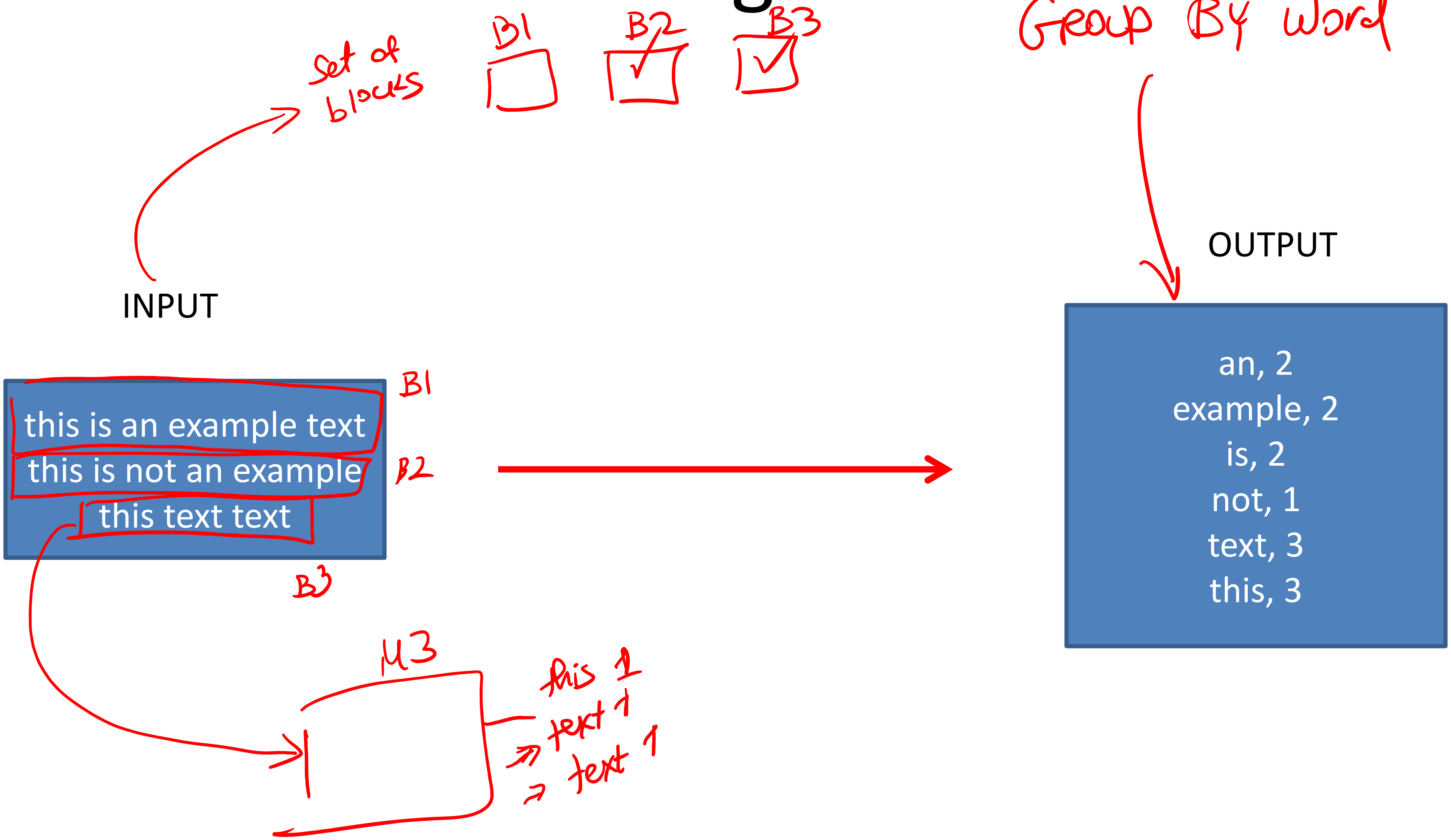
- ResourceManager
- NodeManager

- Other nodes (checkpoint, balancer, etc)


Map
Reduce

Counting Words

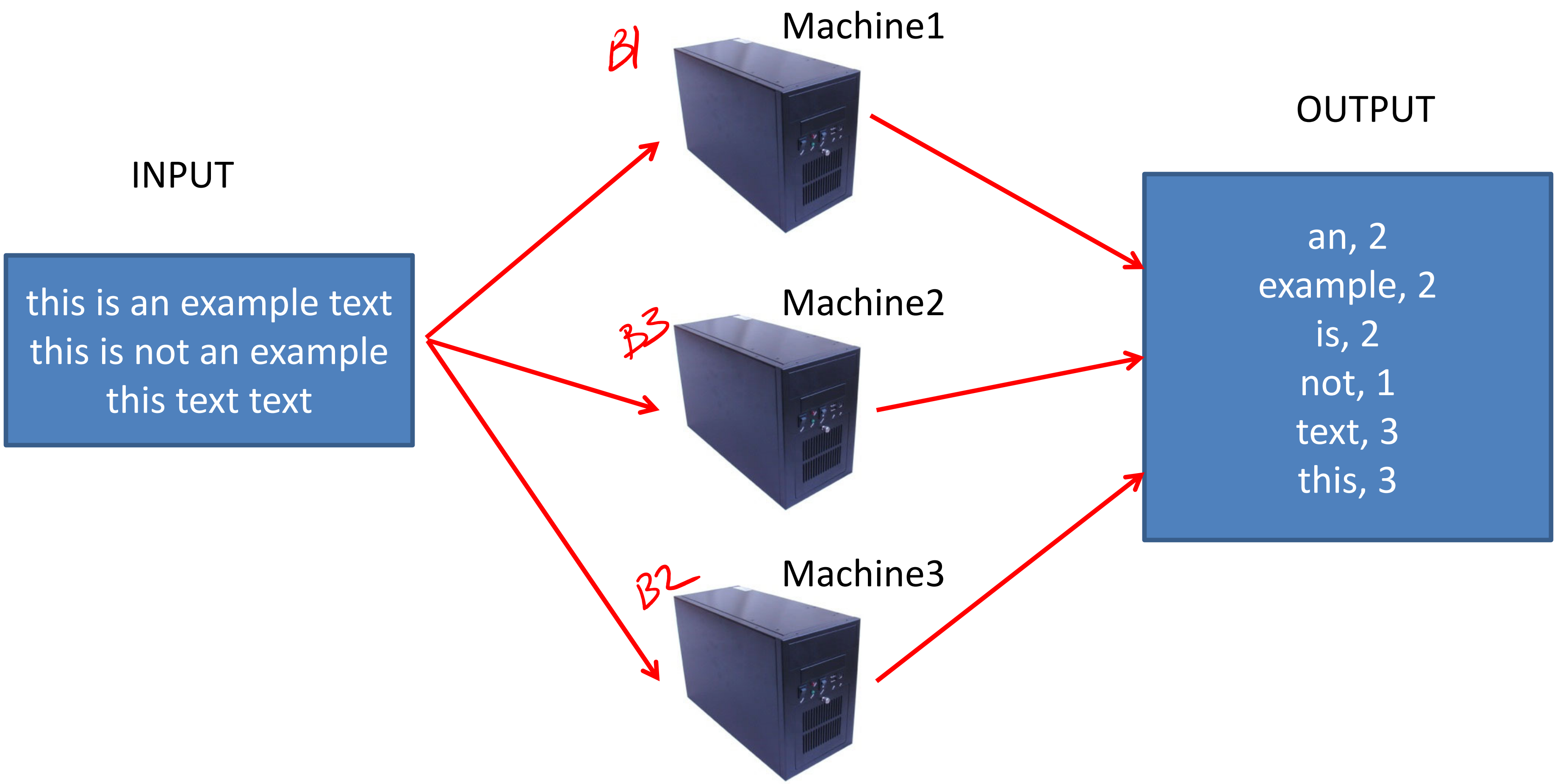
select word, count(*)
FROM MyWords
Group By word

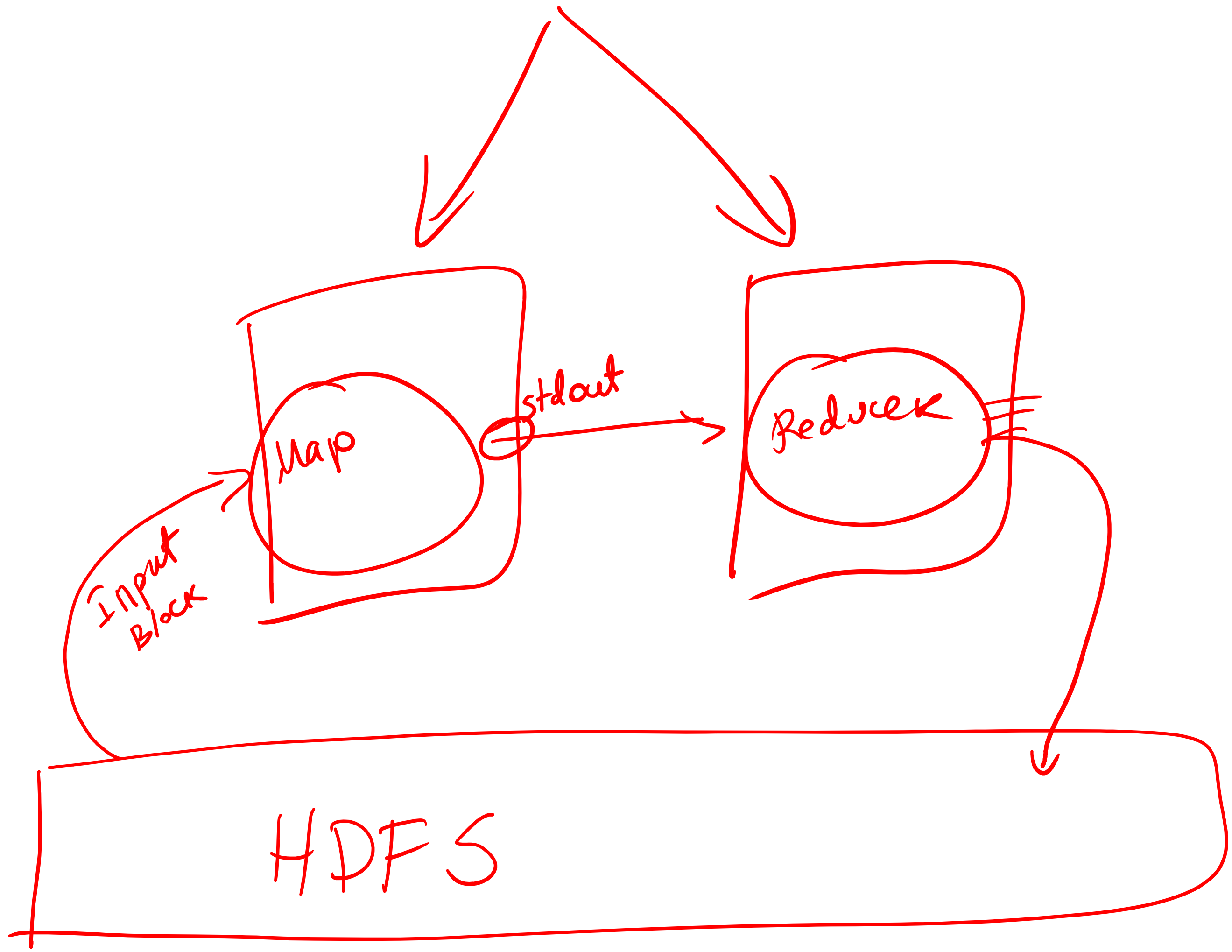


- Applied on **per-block** basis
- Assigned to a machine

Map Worker

Applied on **per-block** basis



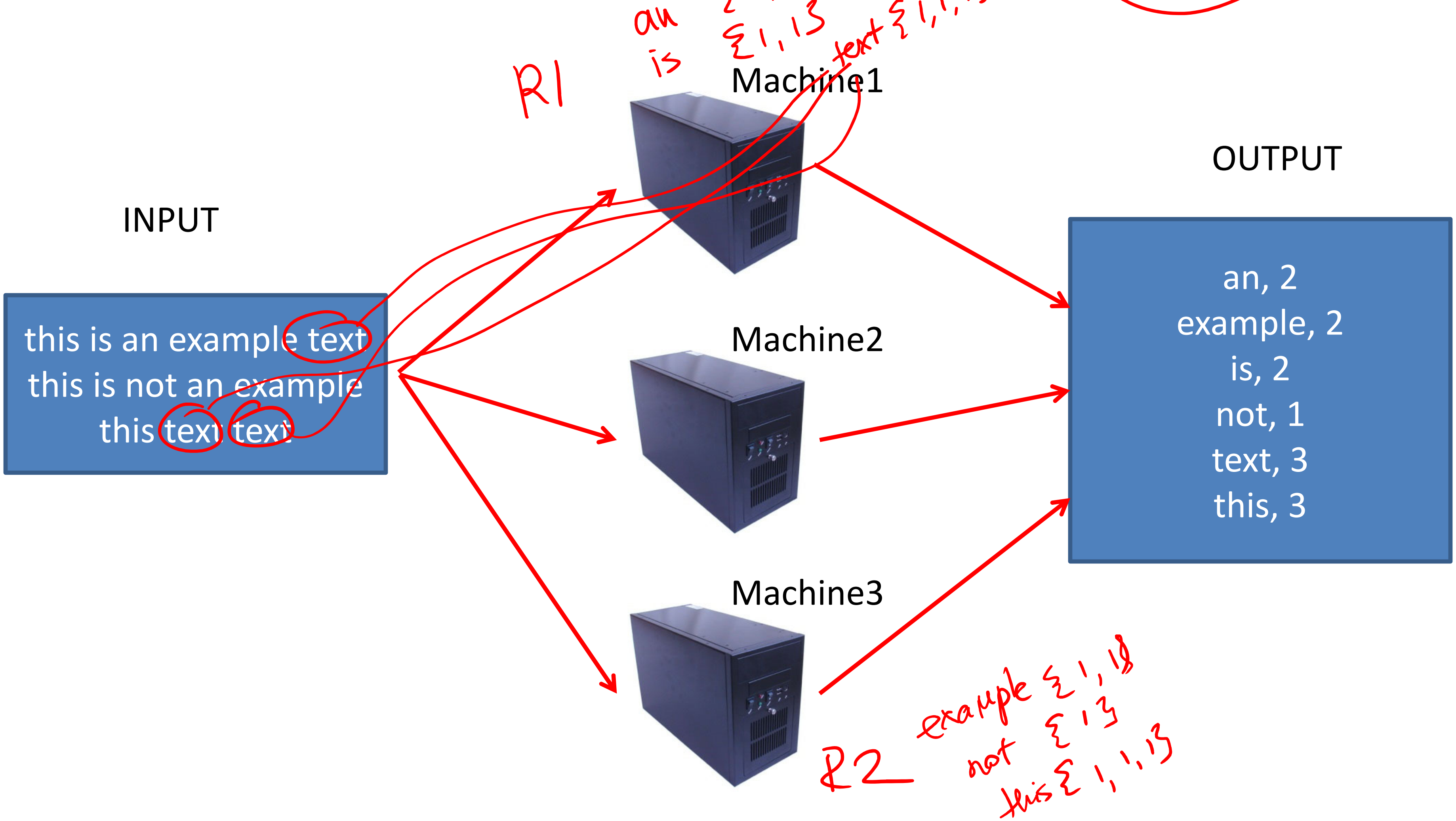


stdout
Map
Reducer
Input Block
HDFS

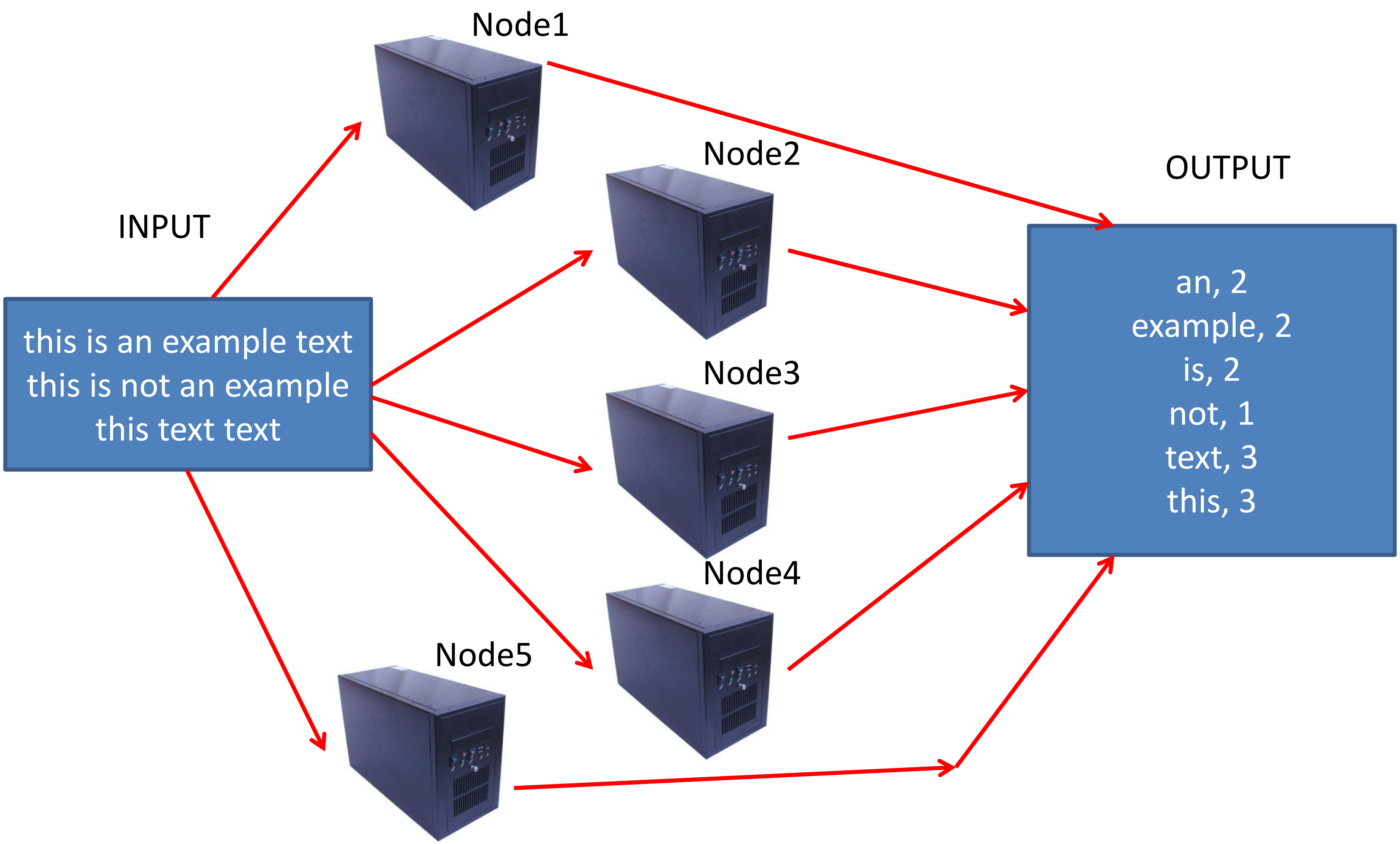
Reduce Worker

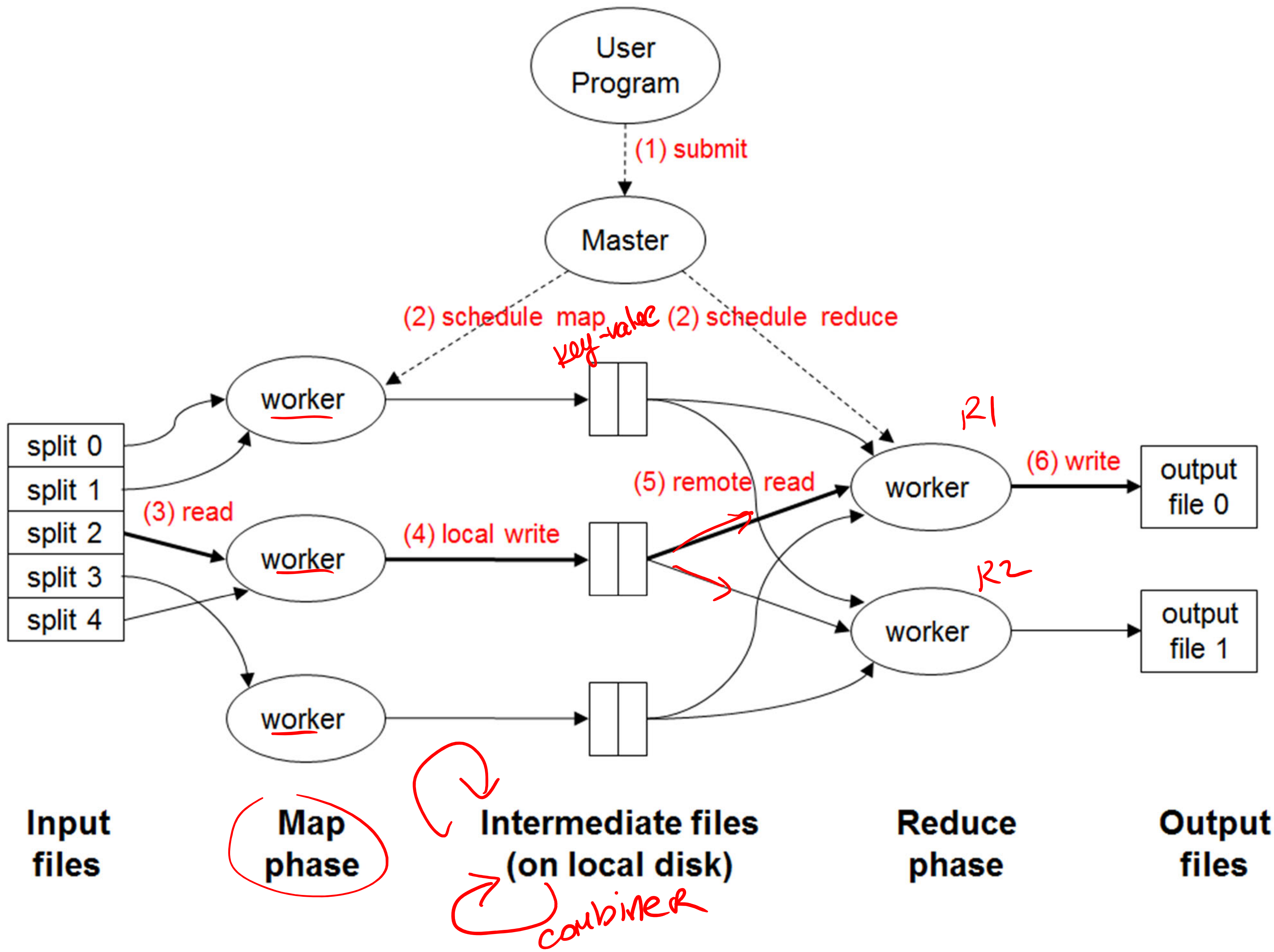
Applied on **per-key** basis

file1
an 2
is 2
text 3



Distributed Word Count





User Program
(1) submit
Master
(2) schedule map
key-value
(2) schedule reduce
split 0
split 1
split 2
split 3
split 4
(3) read
worker
worker
worker
(4) local write
(5) remote read
R1
R2
worker
worker
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disk)
combiner
Reduce phase
Output files

Java Code: Map

```
/**
 * Counts the words in each line.
 * For each line of input, break the line into words and emit them as
 * (<b>word</b>, <b>1</b>).
 */
public static class MapClass extends MapReduceBase
  implements Mapper<LongWritable, Text, Text, IntWritable> {

  private final static IntWritable one = new IntWritable(1);
  private Text word = new Text();

  public void map(LongWritable key, Text value,
                  OutputCollector<Text, IntWritable> output,
                  Reporter reporter) throws IOException {
    String line = value.toString();
    StringTokenizer itr = new StringTokenizer(line);     line.split(' ')
    while (itr.hasMoreTokens()) {
      word.set(itr.nextToken());
      output.collect(word, one);
    }
  }
}
```

Python Code: Map

```
# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        #
        # tab-delimited; the trivial word count is 1
        print '%s\t%s' % (word, 1)
```

'this is text'

['this', 'is', 'text']

→

this \t 1
is \t 1
text \t 1

Java Code: Reduce

```
/**
 * A reducer class that just emits the sum of the input values.
 */
public static class Reduce extends MapReduceBase
  implements Reducer<Text, IntWritable, Text, IntWritable> {

  public void reduce(Text key, Iterator<IntWritable> values,
                     OutputCollector<Text, IntWritable> output,
                     Reporter reporter) throws IOException {
    int sum = 0;
    while (values.hasNext()) {
      sum += values.next().get();
    }
    output.collect(key, new IntWritable(sum));
  }
}
```

```
# input comes from STDIN
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()

    # parse the input we got from mapper.py
    word, count = line.split('\t', 1)

    # convert count (currently a string) to int
    try:
        count = int(count)
    except ValueError:
        # count was not a number, so silently
        # ignore/discard this line
        continue

    # this IF-switch only works because Hadoop sorts map output
    # by key (here: word) before it is passed to the reducer
    if current_word == word:
        current_count += count
    else:
        if current_word:
            # write result to STDOUT
            print '%s\t%s' % (current_word, current_count)
        current_count = count
        current_word = word
```

Performance in Hadoop

- Run WordCount with different settings
- Use 1-node and 5-node Hadoop cluster
- File inputs
  - Small (0.5MB), Medium (182MB), Large (5.6GB)

128 | 54

- Results
  - Small (1-node : **0.5min** 5-nodes : **0.5min**)
  - Medium (1-node : **1.5min** 5-nodes : **1min**)
  - Large (1-node : **26 min** 5-nodes : **<8min**)

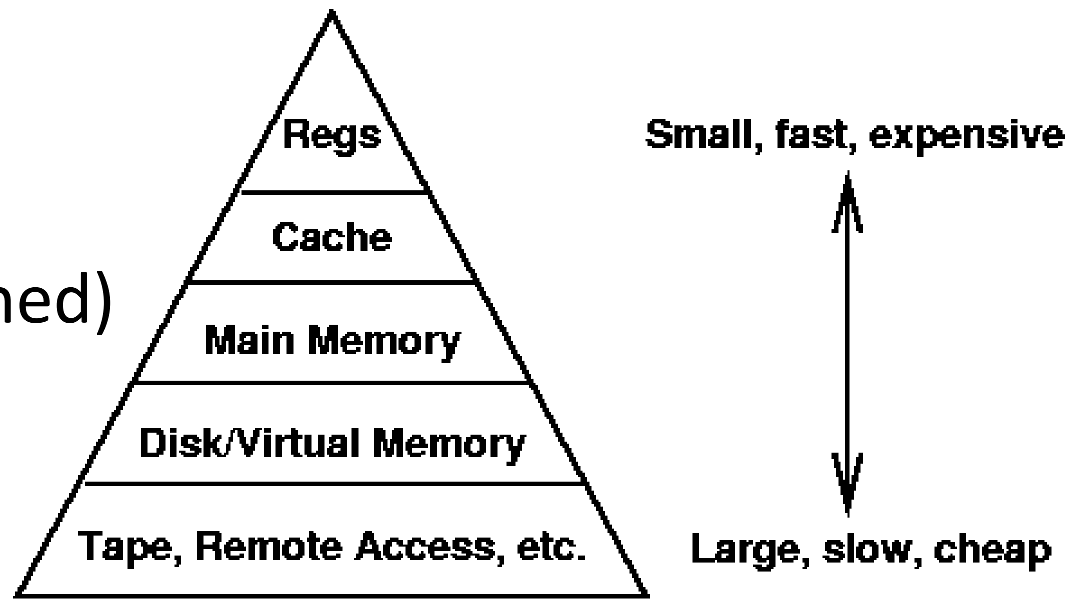
A Break

Runtime / Cost

- Accessing the disk
  - Reading from disk is expensive
- Communication cost
  - Network slower than disk
  - Rack to Rack (Data center to data center)
- Execution cost
  - Actual tasks (map, reduce, etc.)

Disk Access

- Disks are cheap but slow
  - Spark (in RAM)
  - Compression
- Seek vs Read
  - Small files (batched)
  - In place updates



Network Transfer

- Transferring data may be slow
  - Local access/disk is faster
  - Computation is much faster
  - Distance (neighbor, other rack, other datacenter)
- Network saturation
  - Same pipe / joins
  - N^2 communication cost

File Balance in HDFS

- Where do map tasks read the files?
- Balance the workload
  - Worst case => entire file is on one node

- Why unbalanced?
  - New (or replacement) nodes are added
  - Extra drives added, heterogeneous machines

How is HDFS (Map) balanced?

- Blocks written to disk at random
  - (hopefully)
- No automatic rebalancing
  - Replace file (i.e. write again)
  - Turn replication off/on (rewrite duplicates)
  - Run a rebalancer

Reduce Balancing

- How many values does each reducer process?
- Values are assigned across reducers
  - Hashing function (default or custom)
- Worst case => all keys go to just one reducer

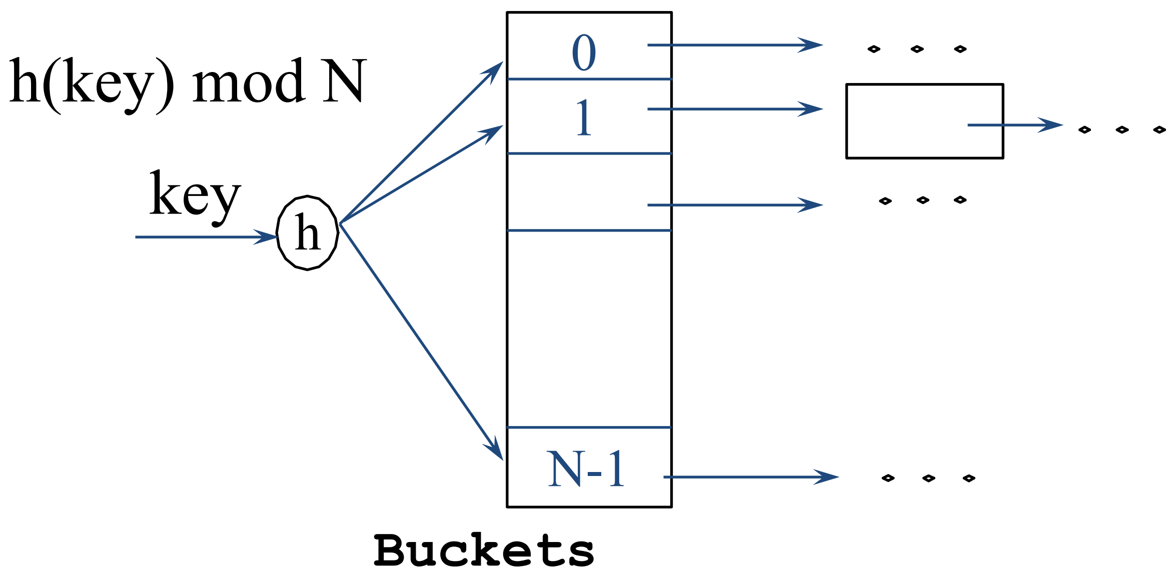
What is Hashing?

- The **Hash House Harriers** (abbreviated to **HHH** or **H3**, or referred to simply as **hashing**) is an international group of non-competitive running social clubs. An event organized by a club is known as a **hash** or **hash run**, with participants calling themselves **hashers** or **hares, hounds, harriers, and harriets**.

HARE AND HOUNDS

Hashing

- **h(k) MOD N** (N = # of buckets)
  - bucket to which data entry with key *k* belongs.



Hashing

- Buckets correspond to Reducers
- Hash function works on **key** field of record **r**. Use its value MOD N to distribute values over range 0 ... N-1.
  - **h**(key) = (a * key + b) usually works well
  - a and b are constants

- Node may have different number of reducers

Hadoop Partitioner

```
public class HashPartitioner<K, V> extends
Partitioner<K, V> {
  /** Use {@link Object#hashCode()} to partition. */

  public int getPartition(K key, V value,
                          int numReduceTasks) {
    return (key.hashCode() & Integer.MAX_VALUE) %
numReduceTasks;
  }
}
```

MOD

Sorting Hat

Describing MapReduce algorithms

- Initially pseudo-code like
  - Later we will use python
- Discuss examples for Map and Reduce tasks
  - Describe the tasks
  - <u>What</u> does Map do?
    - Per block: identify the key, identify the value
  - <u>What</u> does Reduce do?
    - Per key: apply the function

Queries with MapReduce

Key value
Map ID other Columns

SELECT * FROM Students

- Hadoop
  - Parses the input file every time
  - Applies filter when reading/parsing

print(ID + '/t' + restofrow)

SELECT * FROM Students

WHERE age > 30 AND Year < 2022;

if year < 2022 :
print(ID + '\t' + restofrow)

Queries with MapReduce

	Key	Value
Map	Age	1
Reducer	Age	SUM(1)

SELECT Age, COUNT(*)

FROM Student

GROUP BY Age

Avg(Grade)

Map	Age	Grade
Reducer	Age	Avg(Grade)

K1 31 {A, B, C, A B}

33 {C, C, C, C, C}

K2 32 {A, A, A, A, A}

SELECT Age, Year COUNT(*)

FROM Student

	key	Value
Map	Age_Year	1
Reduce	Age, Year	SUM(1)

GROUP BY Age, Year

31_2022 1t 1

31_2023

Set Union

- MapReduce has to parse both files

		Key	Value
S	Map1	ID_S	1
F	Map2	ID_F	1
	Reducer	ID	—

- SQL:

```
SELECT ID FROM Student
UNION
SELECT ID FROM Faculty;
```

Union with MapReduce

- Need to process 2 files
  - Input == directory with 2 files
- Map
  - Process each file
  - Emit compatible keys
- Reduce
  - Emit <u>single</u> output for every <u>key</u>

Set Intersection

7, t, S

7 S

7 F

- MapReduce has to parse both files

	Key	Value	
Map1	ID_S	~~1~~	S
Map2	ID_F	~~1~~	F

- SQL:

    SELECT ID FROM Students

    INTERSECT

    SELECT ID FROM Faculty;

Reducer

7	~~{1}~~	{S}
8	~~{1,1}~~	{S,S,F}
9	~~{1,1}~~	{F,F}

Intersection with MapReduce

- Process two input files
- Need to identify which key is from where
  - Two different Mappers
- Reduce
  - Iterate through keys
  - Check if at least one from each file appears
  - Emit that key
- Does file order matter?

Hive Architecture

Student.txt

ID, Name, Year
1, Alex, 2015
2, Janu, 2016
3, Jane, 2117

Hive Execution Flow

- Parse the query
- Get metadata from MetaStore
- Create a logical plan
- Optimize the plan
- Create a physical plan
  - DAG of MR jobs

Hive Examples

CREATE TABLE u_data (userid INT,
movieid INT,
rating INT,
unixtime STRING)
ROW FORMAT DELIMITED FIELDS
TERMINATED BY '\t' STORED AS TEXTFILE; (not compressed)

- show tables; describe u_data;
- wget http://www.grouplens.org/system/files/ml-100k.zip

LOAD DATA LOCAL INPATH 'ml-100k/u.data'
OVERWRITE INTO TABLE u_data;

Using Hive

- SELECT COUNT(*) FROM u_data;
- SELECT * FROM u_data WHERE userid = 449;
- SELECT userid, AVG(rating) from u_data GROUP BY userid;
- SELECT userid, AVG(rating) as AR from u_data GROUP BY userid ORDER BY AR;

Next Time:

- Distributed system/Hadoop performance
- Compression and public-private keys
- Hive
- Read:
  - Mining of Massive Datasets
    - Section 2.3
  - Hadoop: The Definitive Guide
    - pp 3-10, Chapter1: Data! Through Grid Computing
    - Pp 43-49, Chapter 3: Design of HDFS through HDFS Federation
    - Pp 50-53 Chapter 3: Command-line Interface and Basic Filesystem Operations
    - Pp 69- 71 Chapter 3: Data flow