CS4225/CS5425 Big Data Systems for Data Science

MapReduce & Relational Databases

Bryan Hooi
School of Computing
National University of Singapore
bhooi@comp.nus.edu.sg

School of Computing

Announcements

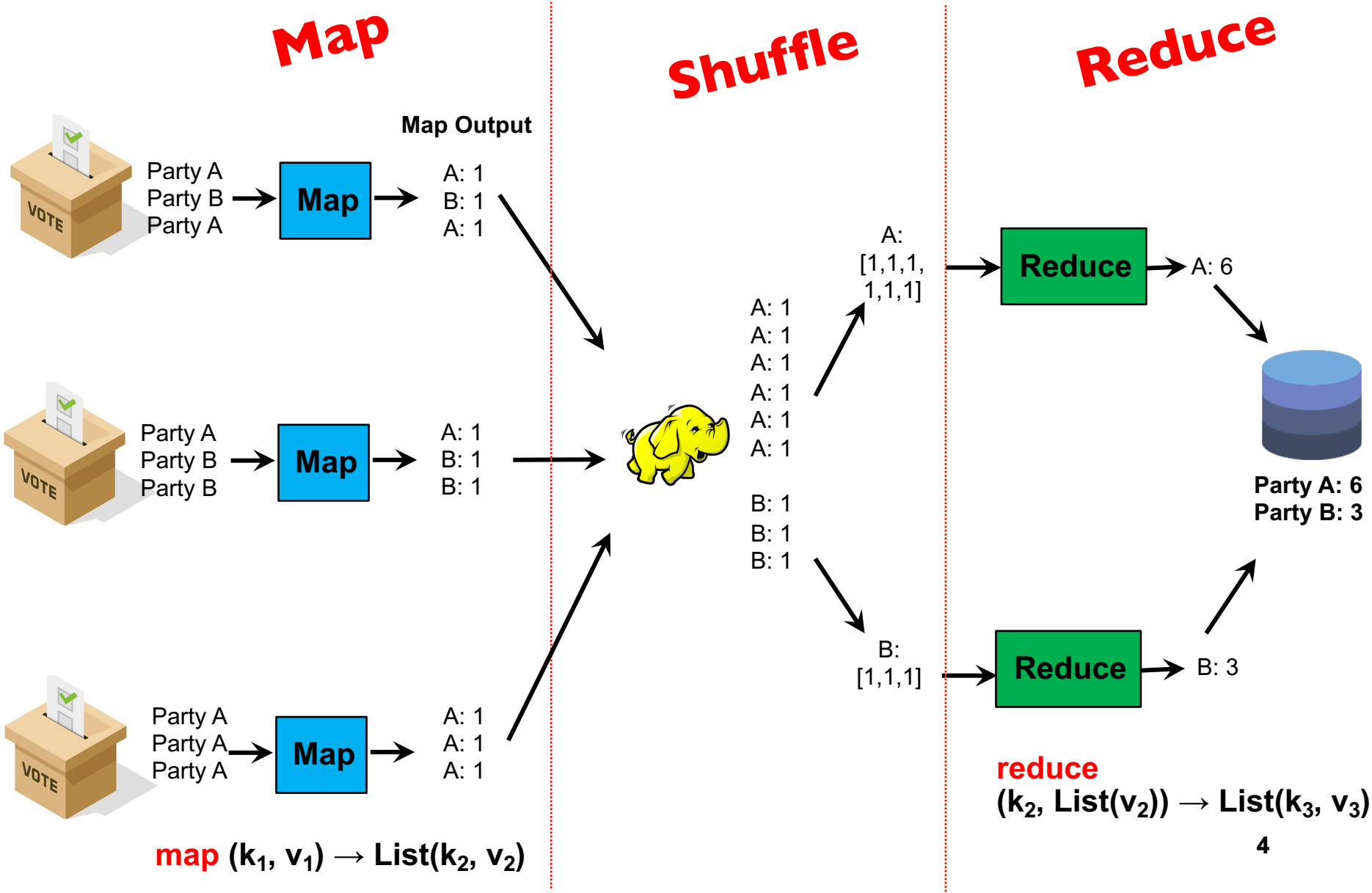
- Tutorial on Hadoop will be held next week (after lecture, 8.30pm – 9.30pm)
- Questions are posted on LumiNUS
- Not graded, but recommended to look through the questions and think about them before the session

Week	Date	Topics	Tutorial	Due Dates
1	12 Aug	Overview and Introduction		
2	29 Aug	MapReduce - Introduction		
3	26 Aug	MapReduce and Relational Databases		
4	2 Sep	MapReduce and Data Mining	Tutorial: Hadoop	Assignment 1 released
5	9 Sep	NoSQL Overview 1		
6	16 Sep	NoSQL Overview 2		
Recess				
7	30 Sep	Apache Spark 1	Tutorial: NoSQL & Spark	Assignment 1 due, Assignment 2 released (3 Oct)
8	7 Oct	Apache Spark 2		
9	14 Oct	Large Graph Processing 1	Tutorial: Graph Processing	
10	21 Oct	Large Graph Processing 2		
11	28 Oct	Stream Processing	Tutorial: Stream Processing	Assignment 2 due (31 Oct)
12	4 Nov	Deepavali – No Class		
13	11 Nov	**Test**		

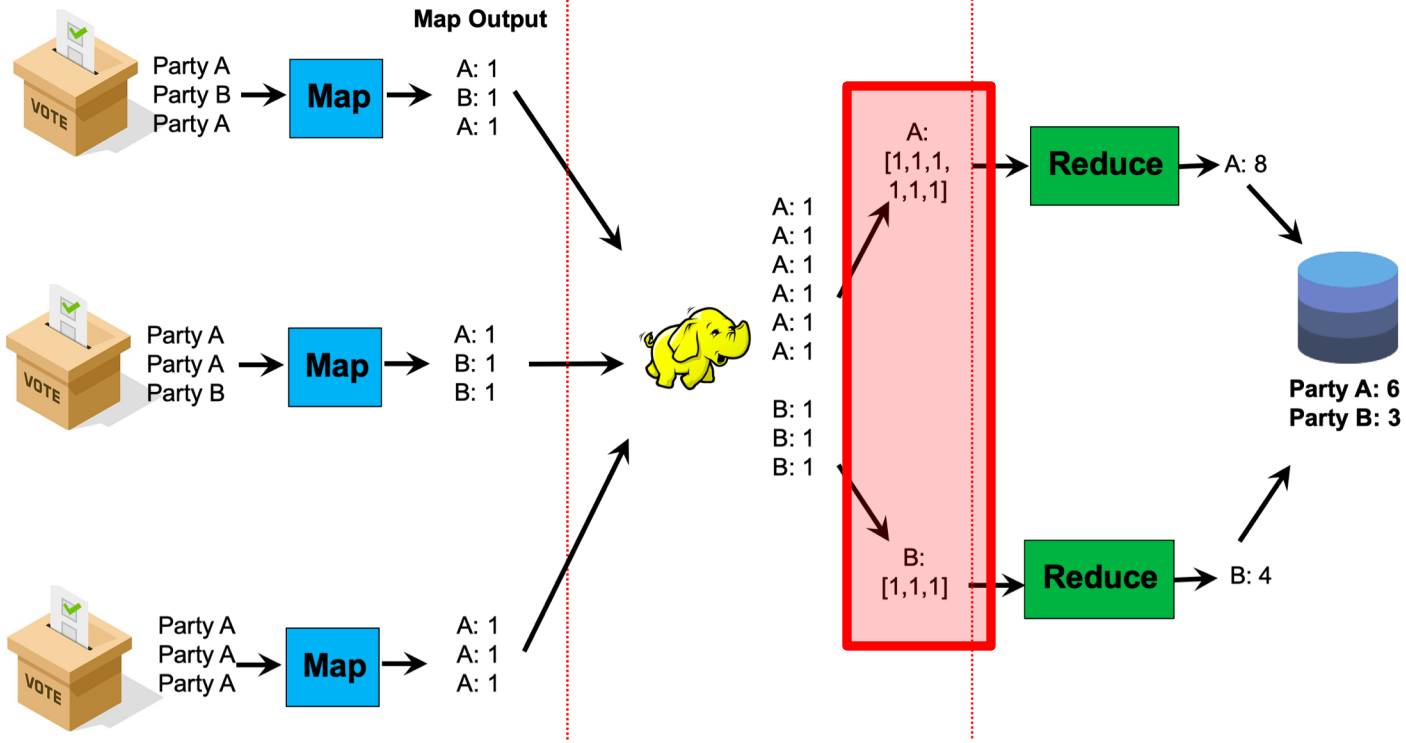
Announcements

- Reminder: if you want to take the test in-class, please fill out the survey in LumiNUS

Recap: MapReduce Algorithm



Recap: Partition Step



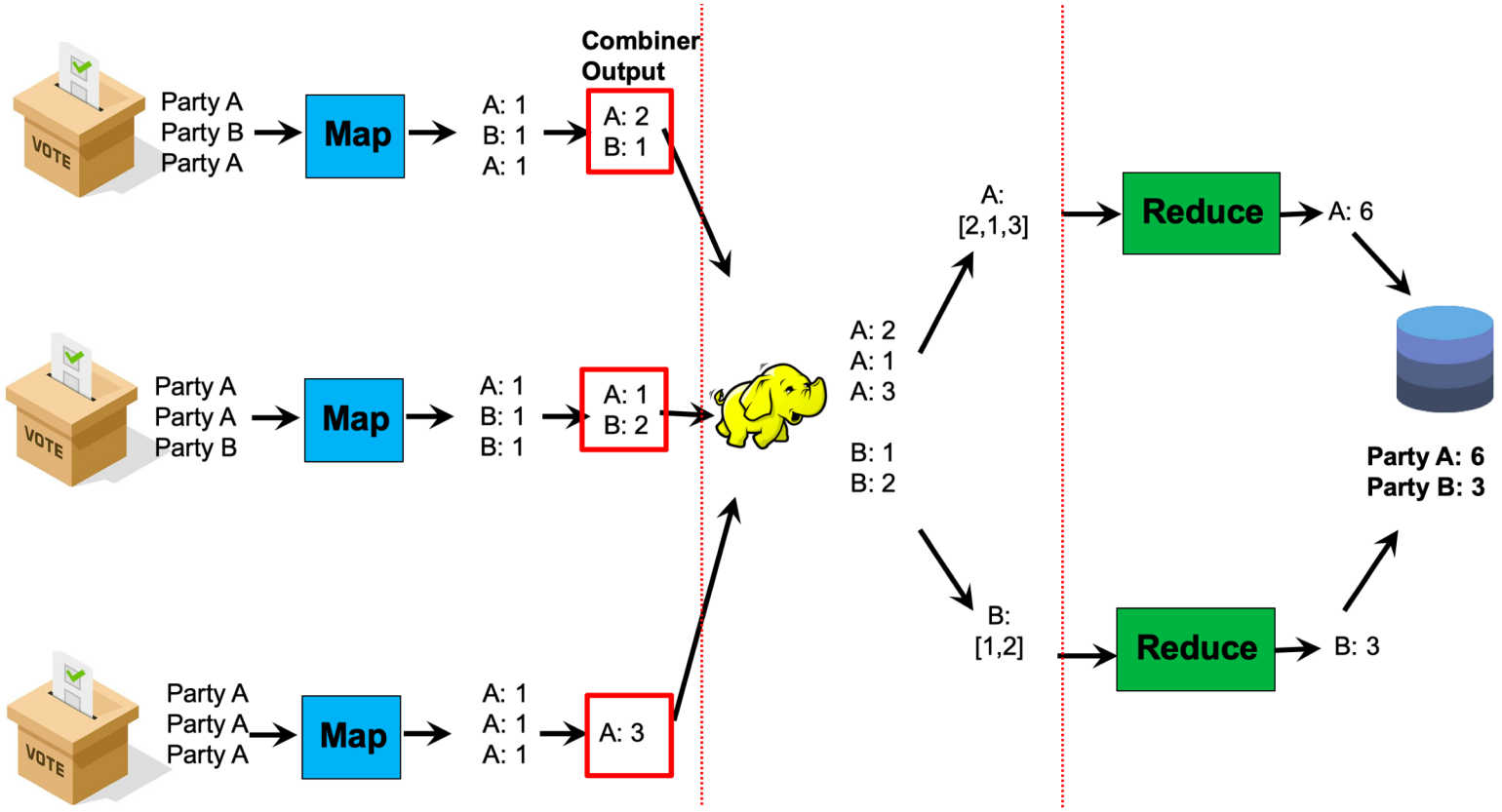
- Note that key A went to reducer 1, and key B went to reducer 2
- By default, the assignment of keys to reducers is determined by a **hash function**

  %

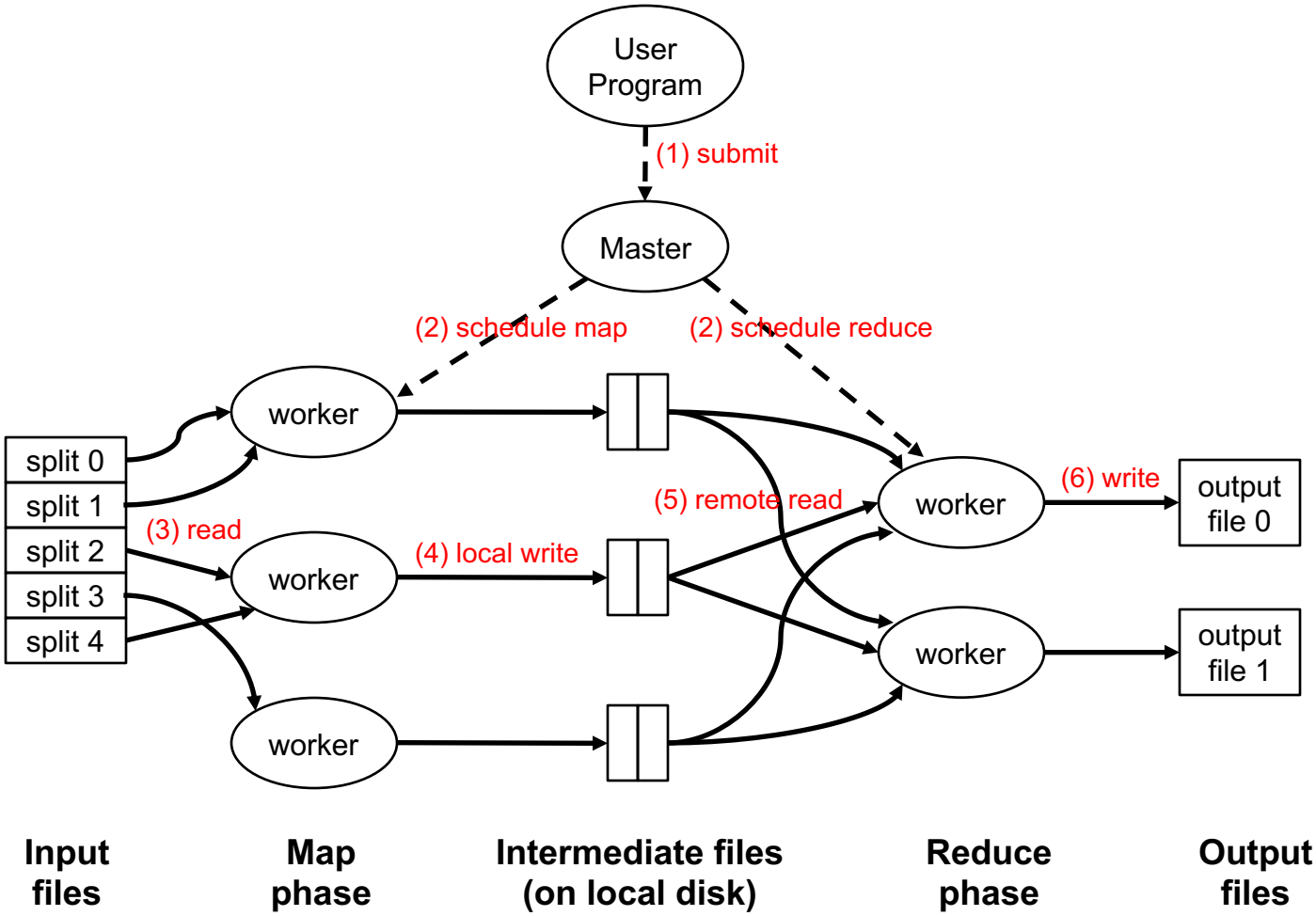
  - e.g., key *k* goes to reducer hash(*k*) mod *num_reducers*
- User can optionally implement a custom partition, e.g. to better spread out the load among reducers (if some keys have much more values than others)

Recap: Combiner

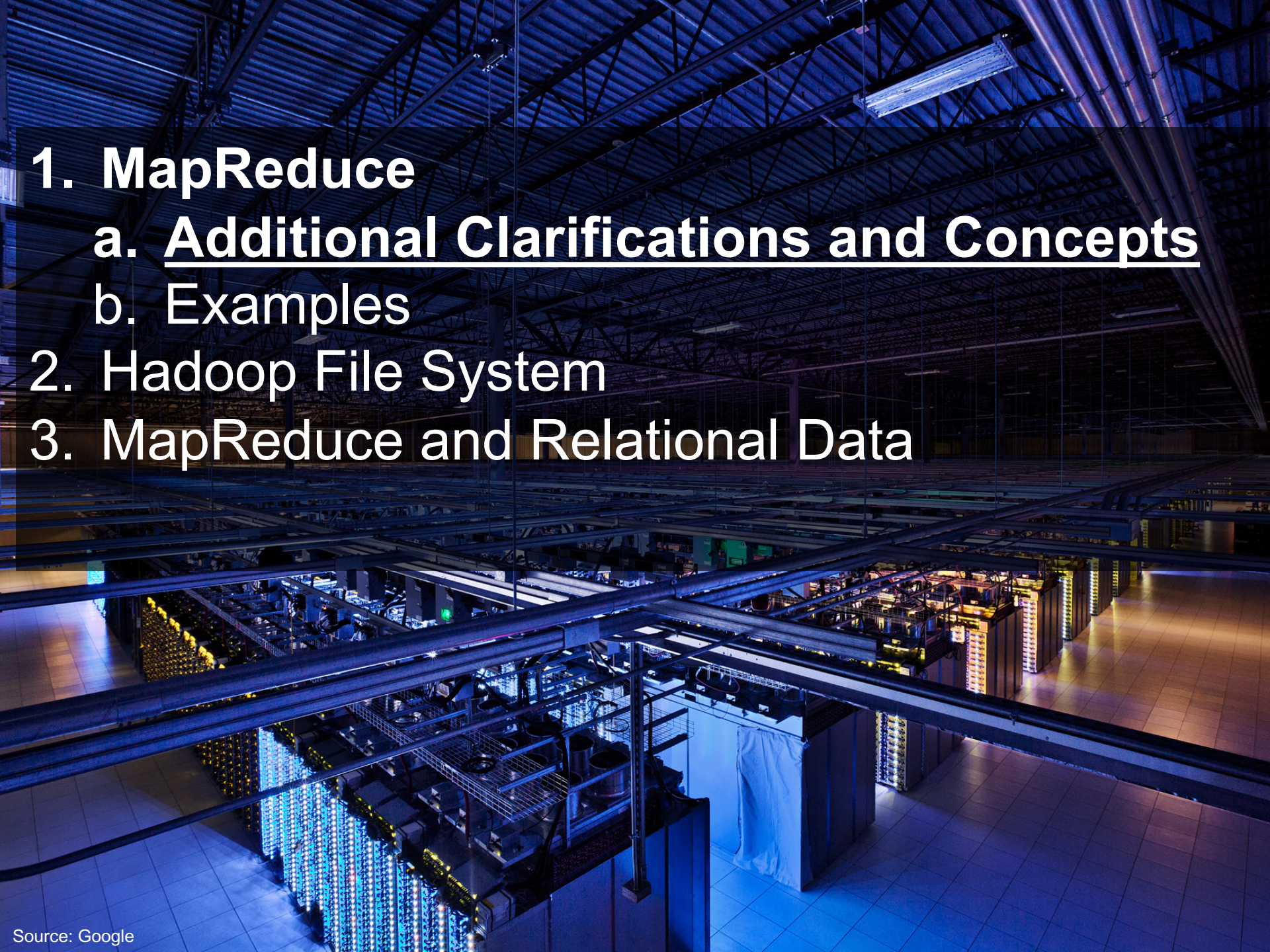
- The user must ensure that the combiner does not affect the correctness of the final output, whether the combiner runs 0, 1, or multiple times



Recap: MapReduce Implementation



Adapted from (Dean and Ghemawat, OSDI 2004)

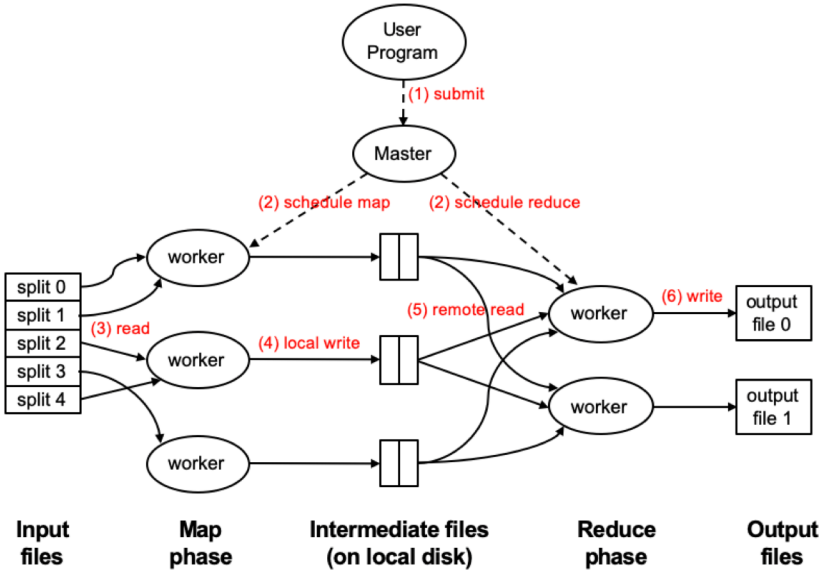


1. **MapReduce**
   a. **Additional Clarifications and Concepts**
   b. Examples
2. Hadoop File System
3. MapReduce and Relational Data

Source: Google

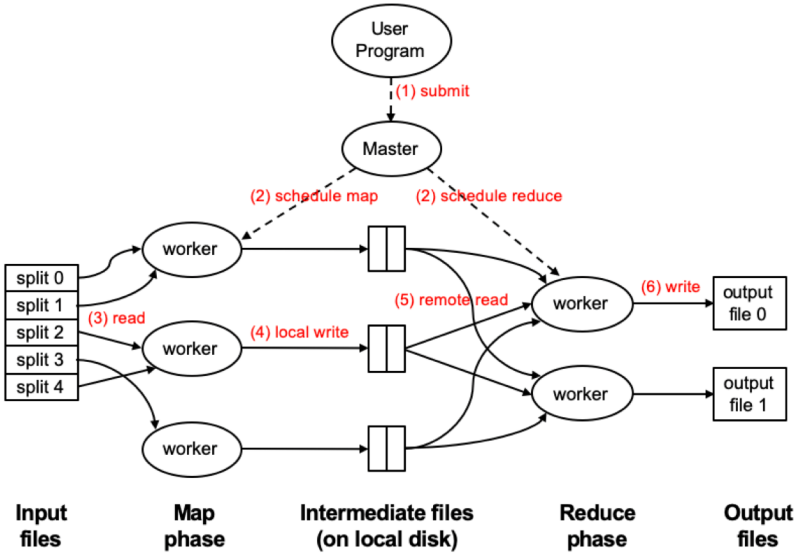
Q: Can a single physical machine handle multiple map tasks, or a combination of map and reduce tasks?

- Yes. When a machine completes a map task (e.g. on split 0), it can be assigned another such task (e.g. split 3).
- Also, machines used for map tasks are often reassigned to reduce tasks once the map phase is done.
- The phrase "mapper" or "reducer" will generally refer to map tasks or reduce tasks, i.e. a job, not a physical machine.



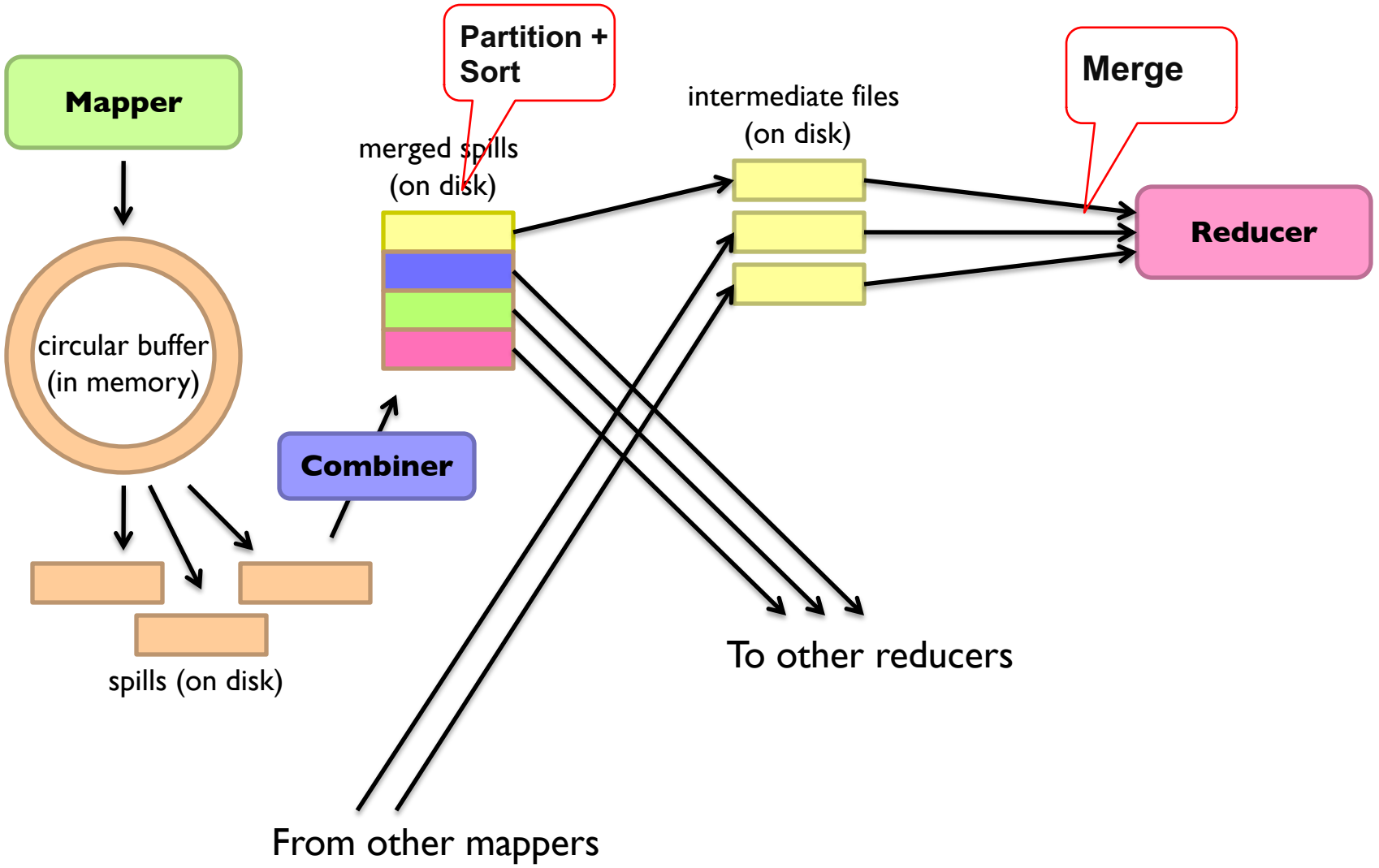
Q: Can a single mapper handle multiple key-value pairs? Can a reducer handle multiple keys?

- Yes to both: each mapper processes all the key-value pairs within a "split". Each split contains a block of input data, e.g. 128MB.
- Each reducer handles all the keys assigned to it by the **partitioner** (which may be user-defined or default)
  - Each reducer's input is **sorted by key:** the keys assigned to a reducer will be fed to it in order of keys. E.g. if a reducer handles keys A and C, it will be fed key A before key C (assuming alphabetical order)

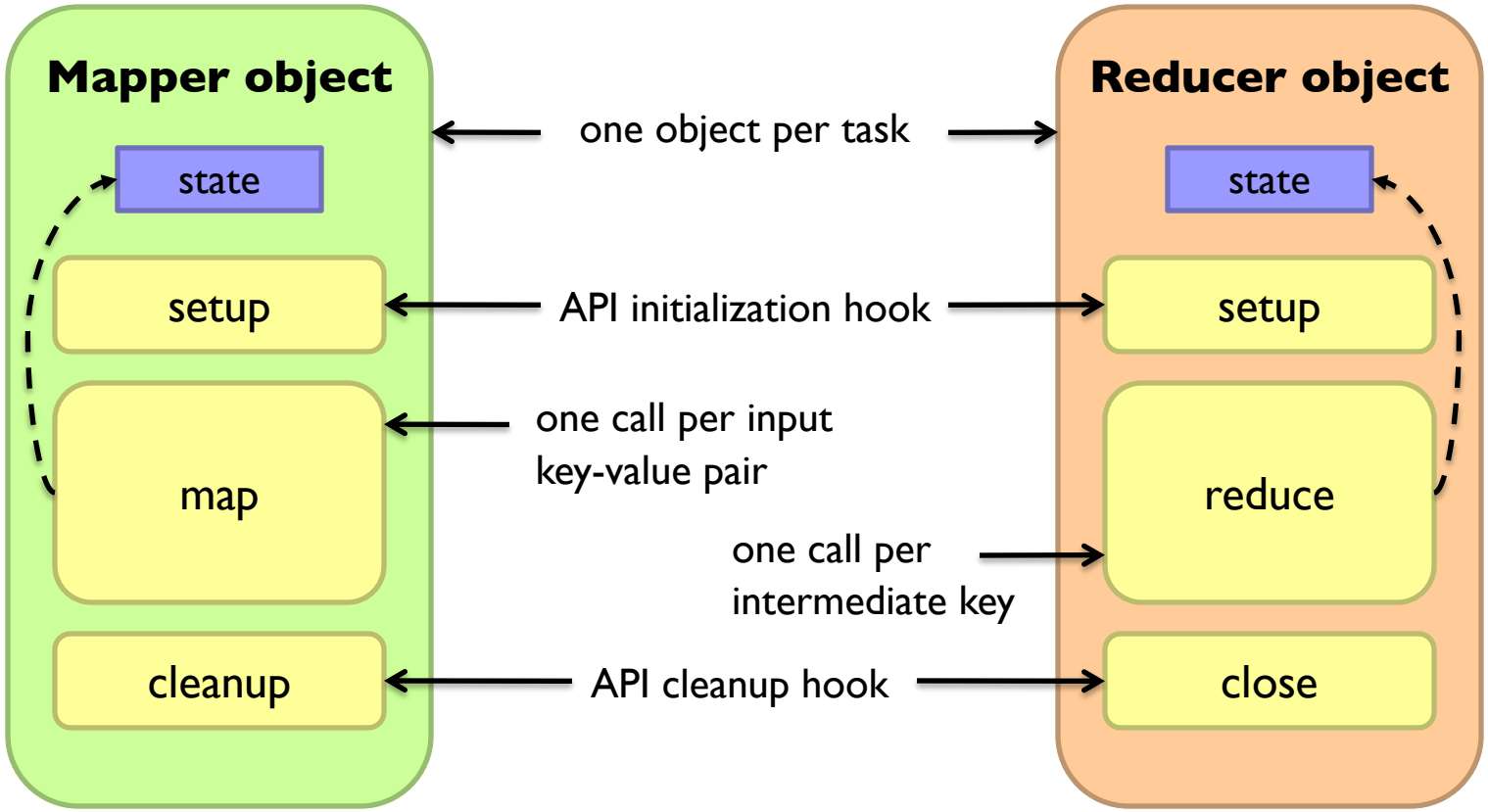


Implementation of Shuffle phase

Optional

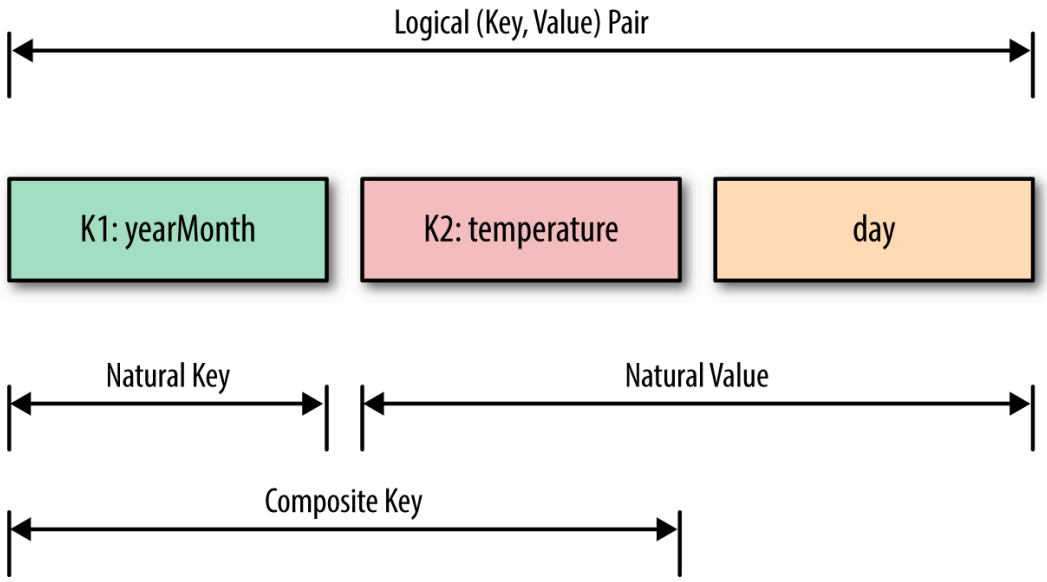


Preserving State in Mappers / Reducers



Secondary Sort

- **Problem**: each reducer's values arrive unsorted. But what if we want them to be sorted (e.g. sorted by temperature)?
- **Solution**: define a new key as (K1, K2), where K1 is the original key ("Natural Key") and K2 is the variable we want to use to sort
  - **Partitioner**: now needs to be customized, to partition by K1 only, not (K1, K2)



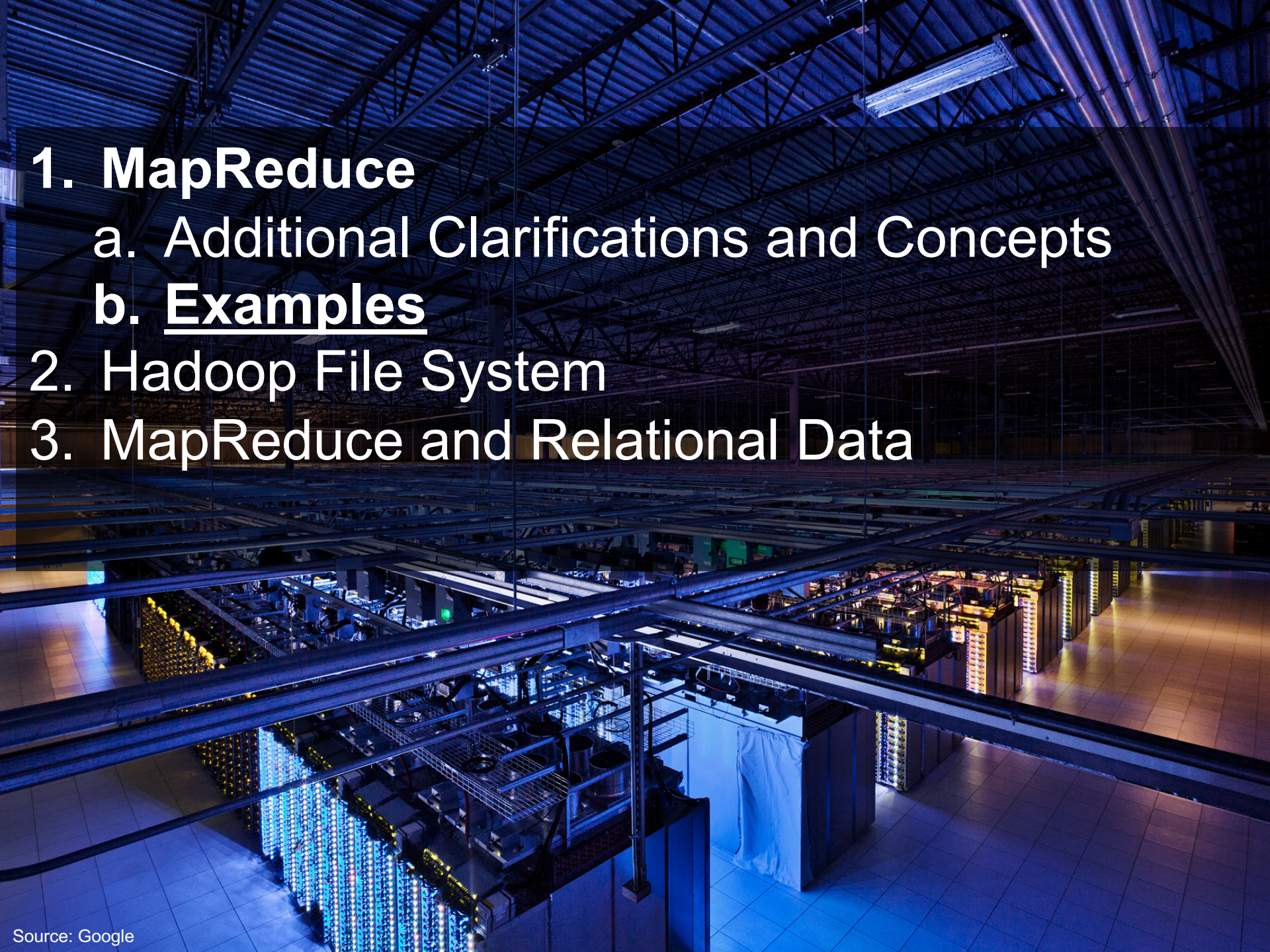
Code Example: Composite Key

```
1  import org.apache.hadoop.io.Writable;
2  import org.apache.hadoop.io.WritableComparable;
3  ...
4  public class DateTemperaturePair
5     implements Writable, WritableComparable<DateTemperaturePair> {
6
7      private Text yearMonth = new Text();                  // natural key
8      private Text day = new Text();
9      private IntWritable temperature = new IntWritable(); // secondary key
10
11     ...
12
13     @Override
14     /**
15      * This comparator controls the sort order of the keys.
16      */
17     public int compareTo(DateTemperaturePair pair) {
18         int compareValue = this.yearMonth.compareTo(pair.getYearMonth());
19         if (compareValue == 0) {
20             compareValue = temperature.compareTo(pair.getTemperature());
21         }
22         //return compareValue;     // sort ascending
23         return -1*compareValue;    // sort descending
24     }
25     ...
26 }
```

Code Example: Custom Partitioner

```
public class DateTemperaturePartitioner
    extends Partitioner<DateTemperaturePair, Text> {

     @Override
     public int getPartition(DateTemperaturePair pair,
                             Text text,
                             int numberOfPartitions) {
         // make sure that partitions are non-negative
         return Math.abs(pair.getYearMonth().hashCode() % numberOfPartitions);
      }
}
```



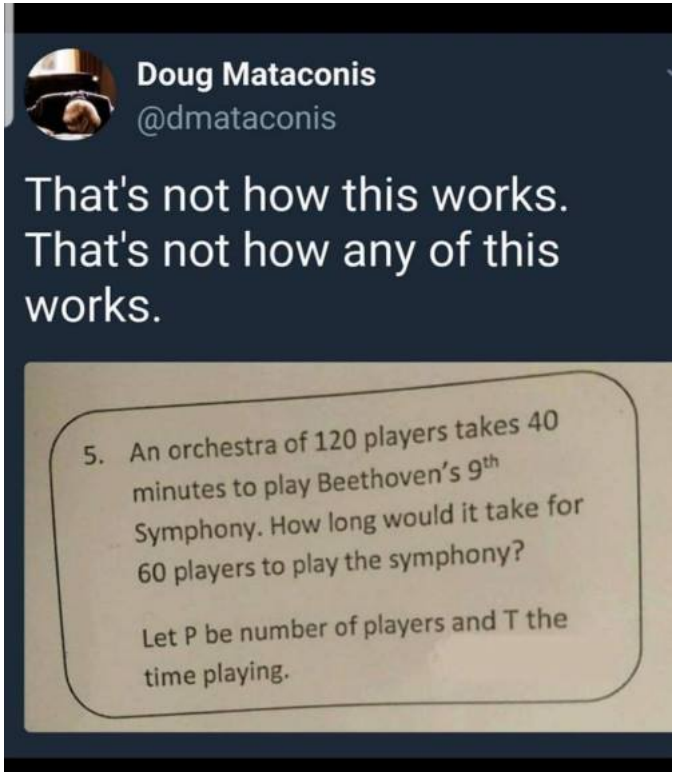
1. **MapReduce**
   a. Additional Clarifications and Concepts
   b. <u>Examples</u>
2. Hadoop File System
3. MapReduce and Relational Data

Source: Google

Performance Guidelines for Basic Algorithmic Design

1. **Linear scalability:** more nodes can do more work in the same time
   - Linear on data size
   - Linear on computer resources
2. **Minimize the amount of I/Os in hard disk and network**
   - Minimize disk I/O; sequential vs. random.
   - Minimize network I/O; bulk send/recvs vs. many small send/recvs
3. **Memory working set** of each task/worker
   - Large memory working set -> high probability of failures.

- Guidelines are applicable to Hadoop, Spark, …



Word Count: Version 0

The mapper emits an intermediate key-value pair for each word in an input document. The reducer sums up all counts for each word.

```
1  class Mapper {
2    def map(key: Long, value: Text) = {
3      for (word <- tokenize(value)) {
4        emit(word, 1)
5    }
6  }
7
8  class Reducer {
9    def reduce(key: Text, values: Iterable[Int]) = {
10     for (value <- values) {
11       sum += value
12     }
13     emit(key, sum)
14   }
15 }
```

What's the key problem of this program?

What's the impact of combiners?

Word Count: Version 1

The mapper builds a histogram of all words in each input document before emitting key-value pairs for unique words observed.

```
1  class Mapper {
2    def map(key: Long, value: Text) = {
3      val counts = new Map()
4      for (word <- tokenize(value)) {
5        counts(word) += 1
6      }
7      for ((k, v) <- counts) {
8        emit(k, v)
9      }
10   }
11 }
```

Key idea: locally aggregate counts in mapper

Are combiners still of any use?

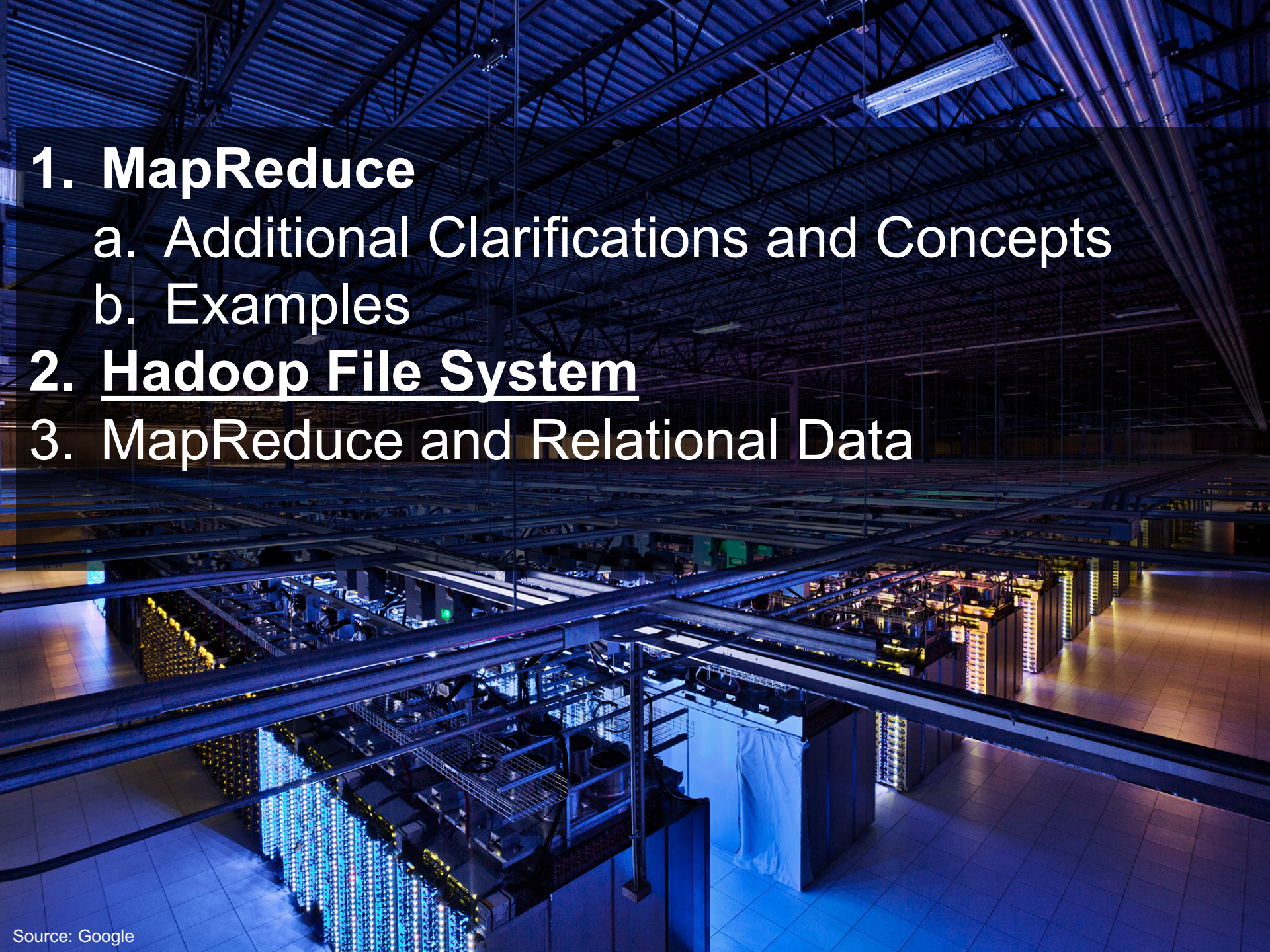
Word Count: Version 2

The mapper builds a histogram of all input documents processed before emitting key-value pairs for unique words observed.

```
1  class Mapper {
2    val counts = new Map()
3
4    def map(key: Long, value: Text) = {
5      for (word <- tokenize(value)) {
6        counts(word) += 1
7      }
8    }
9
10   def cleanup() = {
11     for ((k, v) <- counts) {
12       emit(k, v)
13     }
14   }
15 }
```

Key idea: preserve state across input key-value pairs!

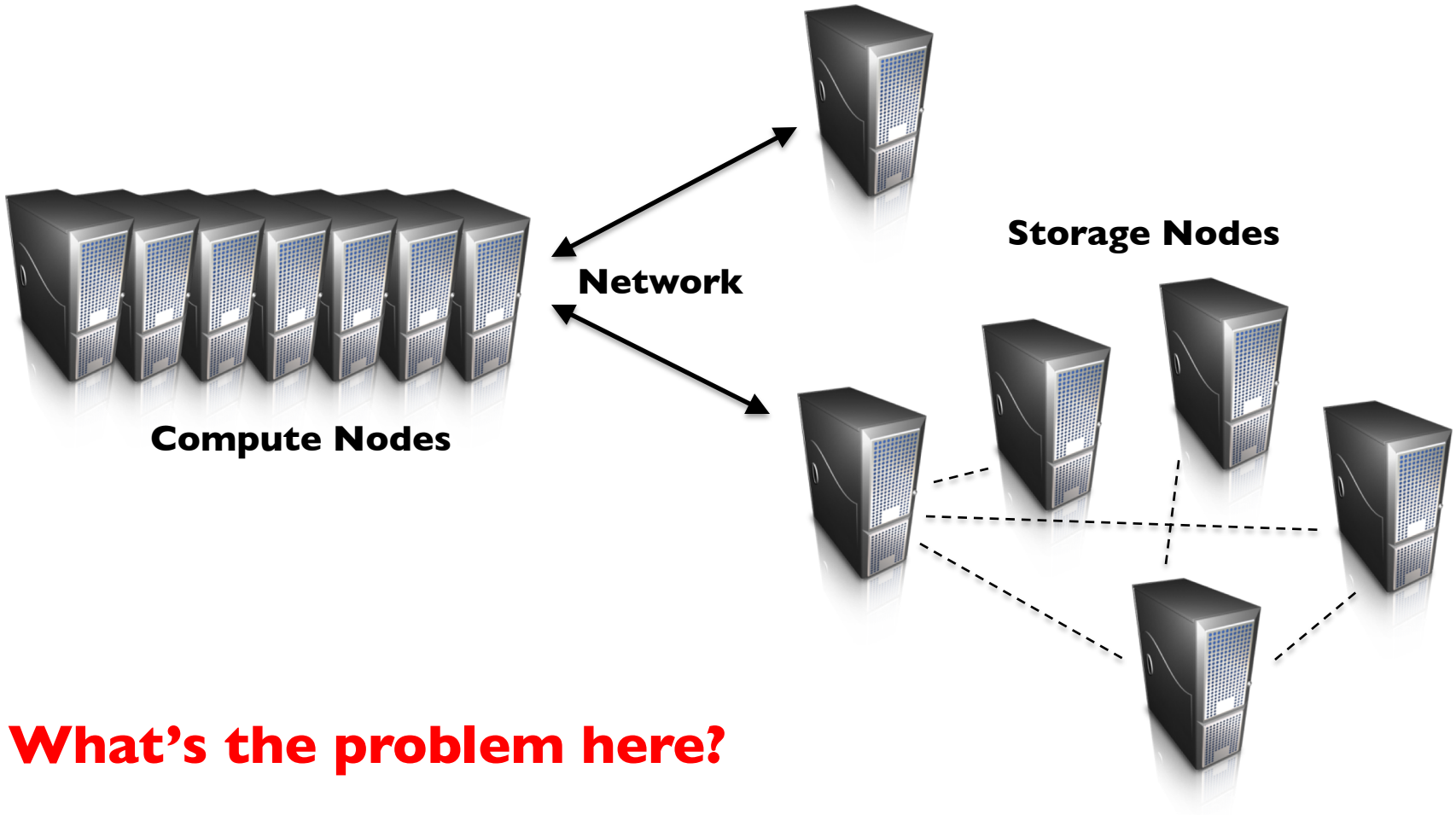
Are combiners still of any use?



1. **MapReduce**
   a. Additional Clarifications and Concepts
   b. Examples
2. **Hadoop File System**
3. MapReduce and Relational Data

Source: Google

How do we get data to the workers?



What's the problem here?

Distributed File System

- Don't move data to workers… move workers to the data!
  - Store data on the local disks of nodes in the cluster
  - Start up the workers on the node that has the data local
- A distributed file system is the answer
  - GFS (Google File System) for Google's MapReduce
  - HDFS (Hadoop Distributed File System) for Hadoop

GFS: Assumptions

- Commodity hardware instead of "exotic" hardware
  - Scale "out", not "up"
- High component failure rates
  - Inexpensive commodity components fail all the time
- "Modest" number of huge files
- Files are write-once, mostly appended to
- Large streaming reads instead of random access
  - High sustained throughput over low latency

GFS slides adapted from material by (Ghemawat et al., SOSP 2003)

GFS: Design Decisions

- Files stored as chunks
  - Fixed size (64MB for GFS, 128MB for HDFS)
- Reliability through replication
  - Each chunk replicated across 3+ chunkservers
- Single master to coordinate access, keep metadata
  - Simple centralized management

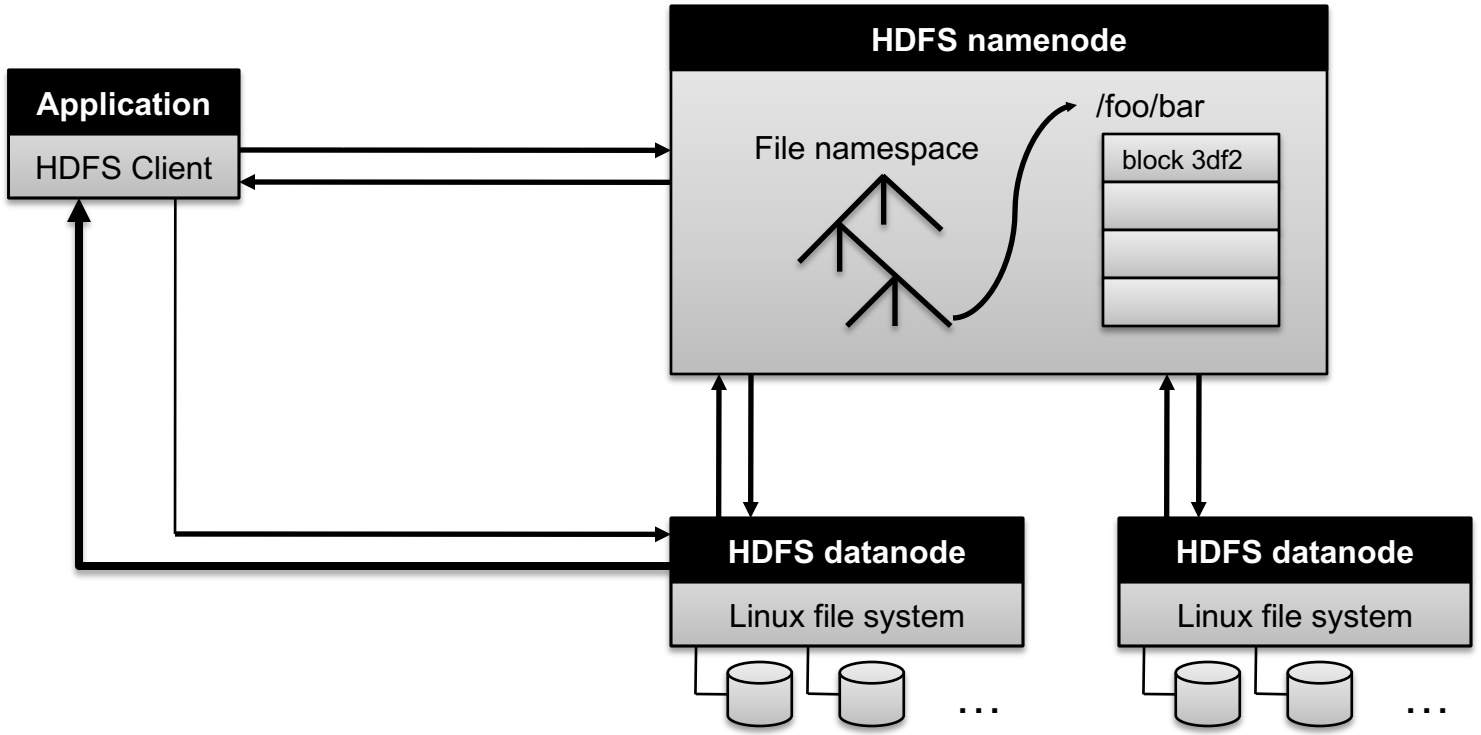
HDFS = GFS clone (same basic ideas)

From GFS to HDFS

- Terminology differences:
  - GFS master = Hadoop namenode
  - GFS chunkservers = Hadoop datanodes
- Differences:
  - Implementation
  - Performance

For the most part, we'll use Hadoop terminology…

HDFS Architecture



Reading files:

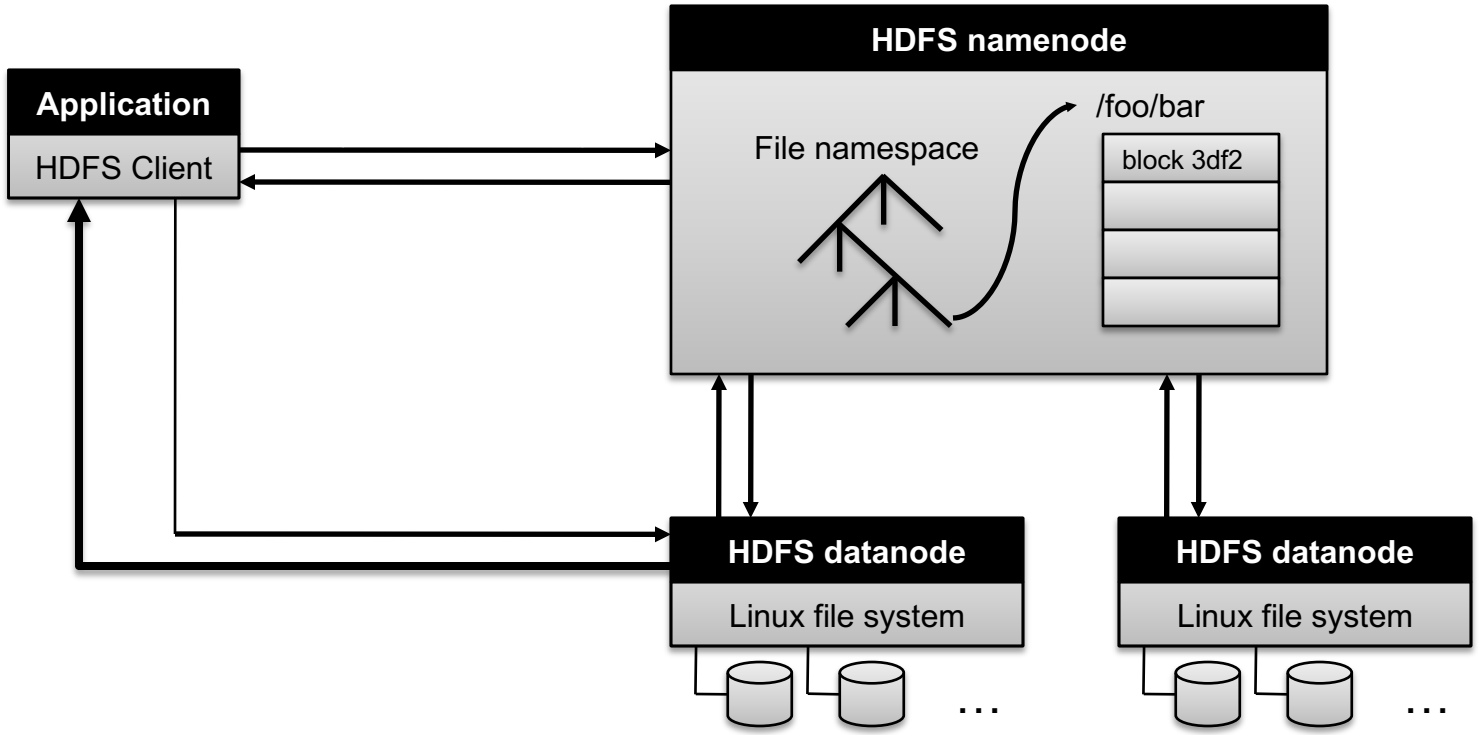
Step 1: application sends request to namenode

Step 2: namenode searches name space and directory, and directs request to a datanode

Step 3: datanode sends the data to client

Adapted from (Ghemawat et al., SOSP 2003)

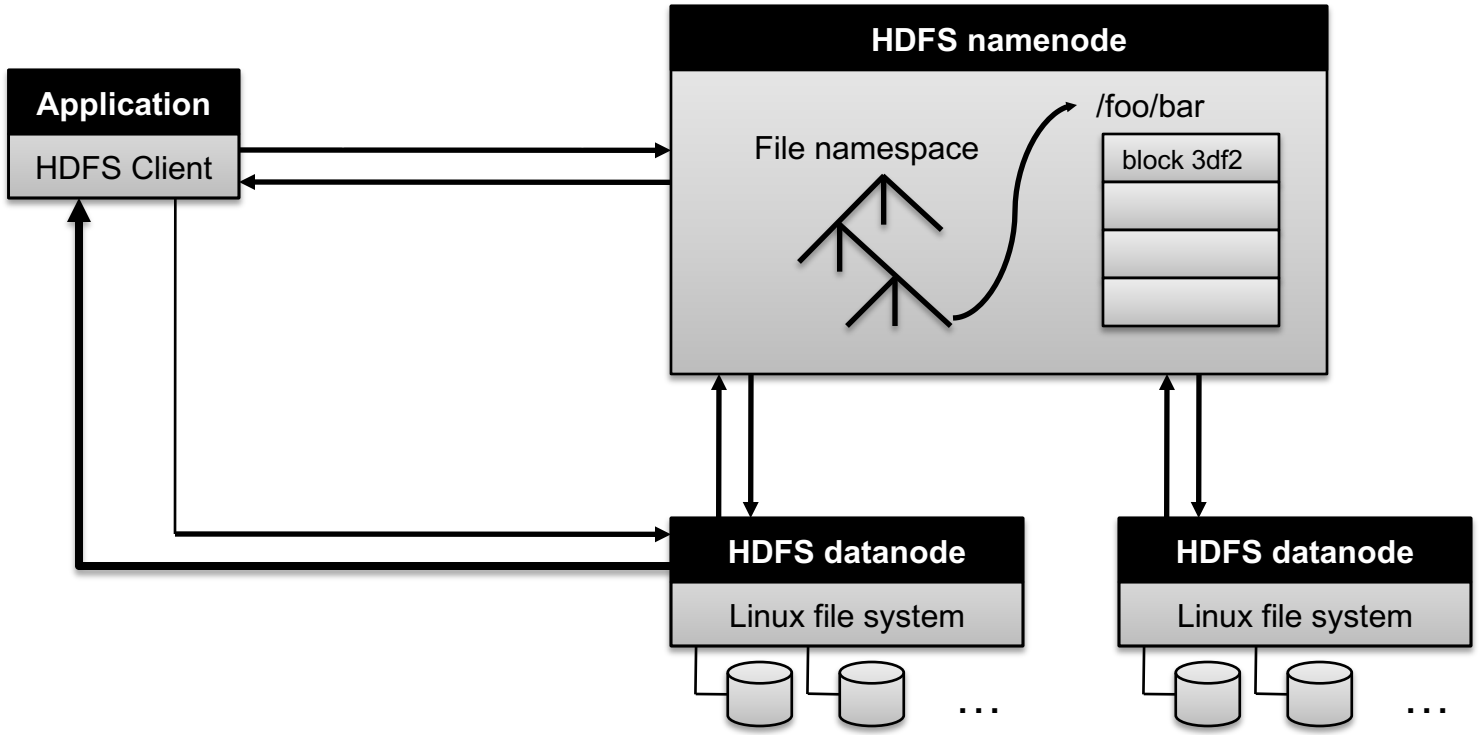
HDFS Architecture



Note: no actual data blocks pass through namenode! The namenode only handles metadata (i.e. filenames and locations)

Adapted from (Ghemawat et al., SOSP 2003)

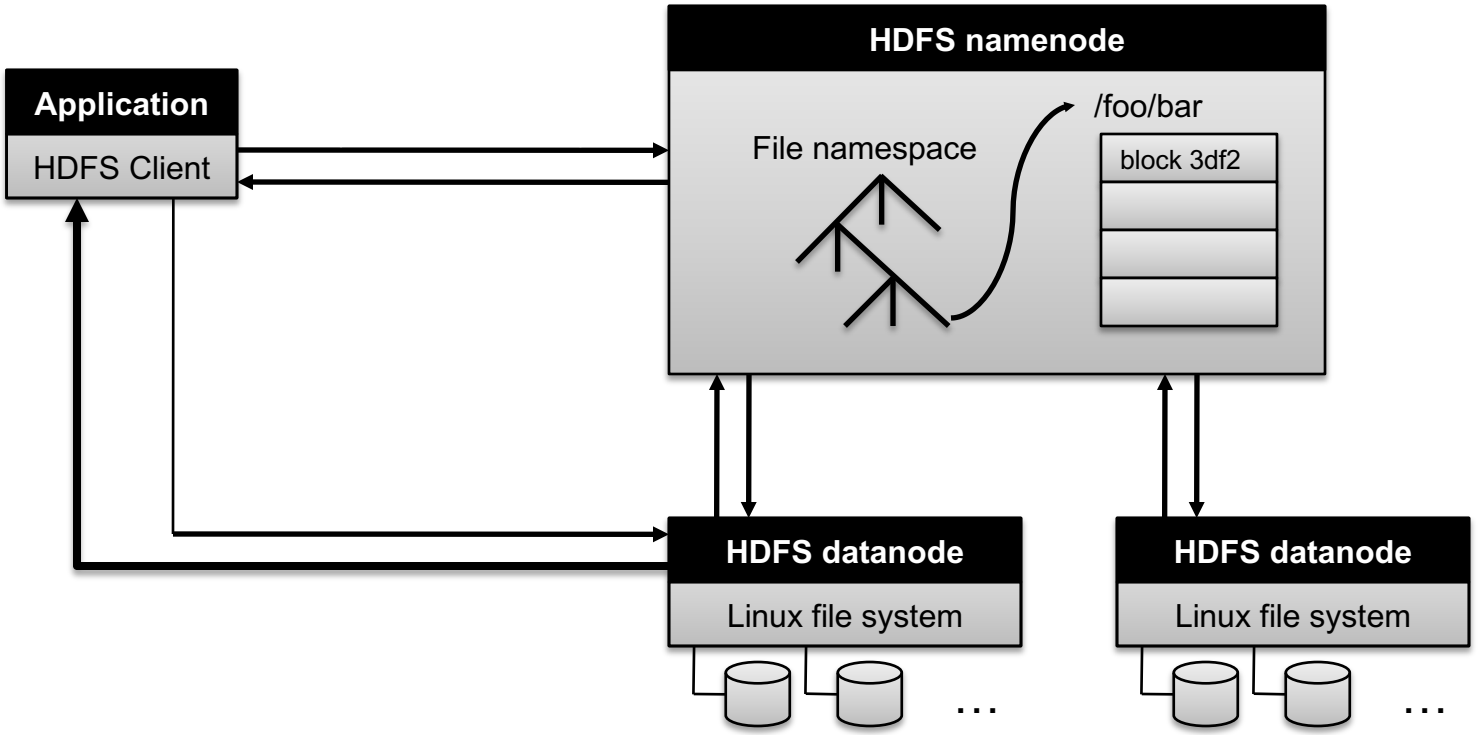
HDFS Architecture



Q: How to perform replication when writing data?

Adapted from (Ghemawat et al., SOSP 2003)

HDFS Architecture



Q: How to perform replication when writing data?

A: Namenode decides which datanodes are to be used as replicas; generally replicas on different racks are chosen. The 1st datanode forwards data blocks to the 1st replica, which forwards them to the 2nd replica, and so on.

Adapted from (Ghemawat et al., SOSP 2003)

Namenode Responsibilities

- Managing the file system namespace:
  - Holds file/directory structure, metadata, file-to-block mapping, access permissions, etc. Coordinating file operations:
  - Directs clients to datanodes for reads and writes
  - No data is moved through the namenode
- Maintaining overall health:
  - Periodic communication with the datanodes
  - Block re-replication and rebalancing
  - Garbage collection

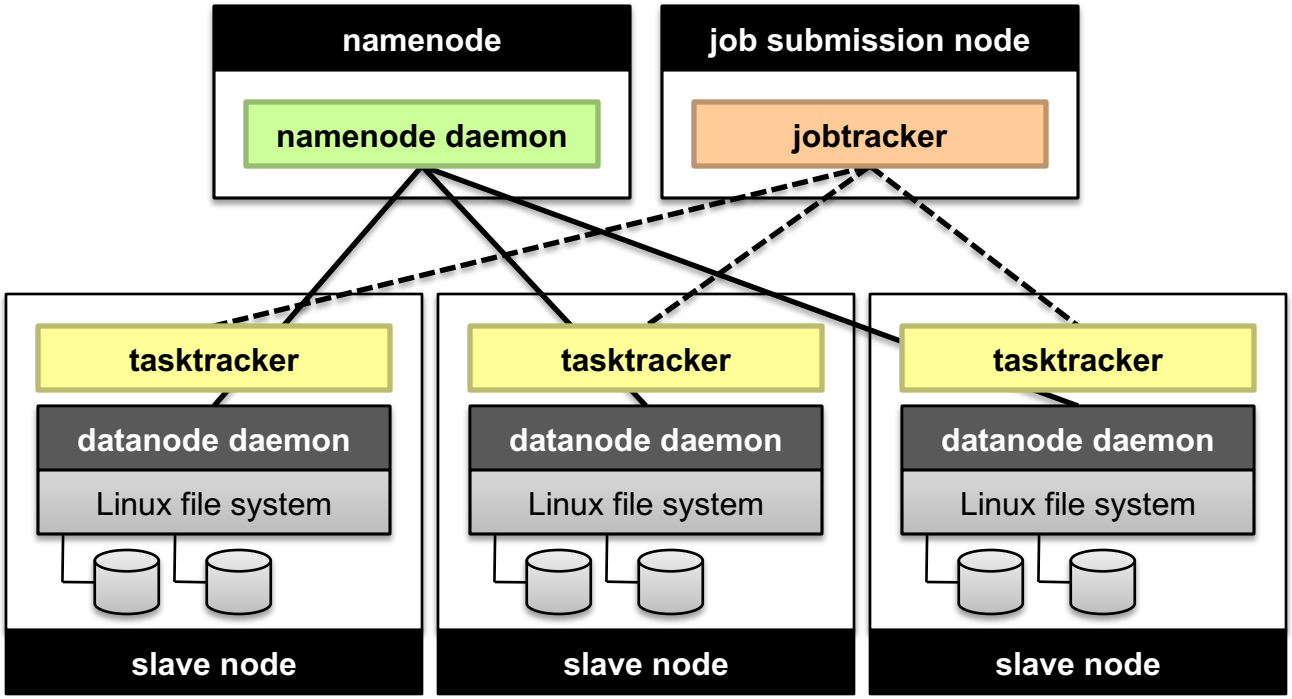
- **Q**: What if the namenode's data is lost?

Namenode Responsibilities

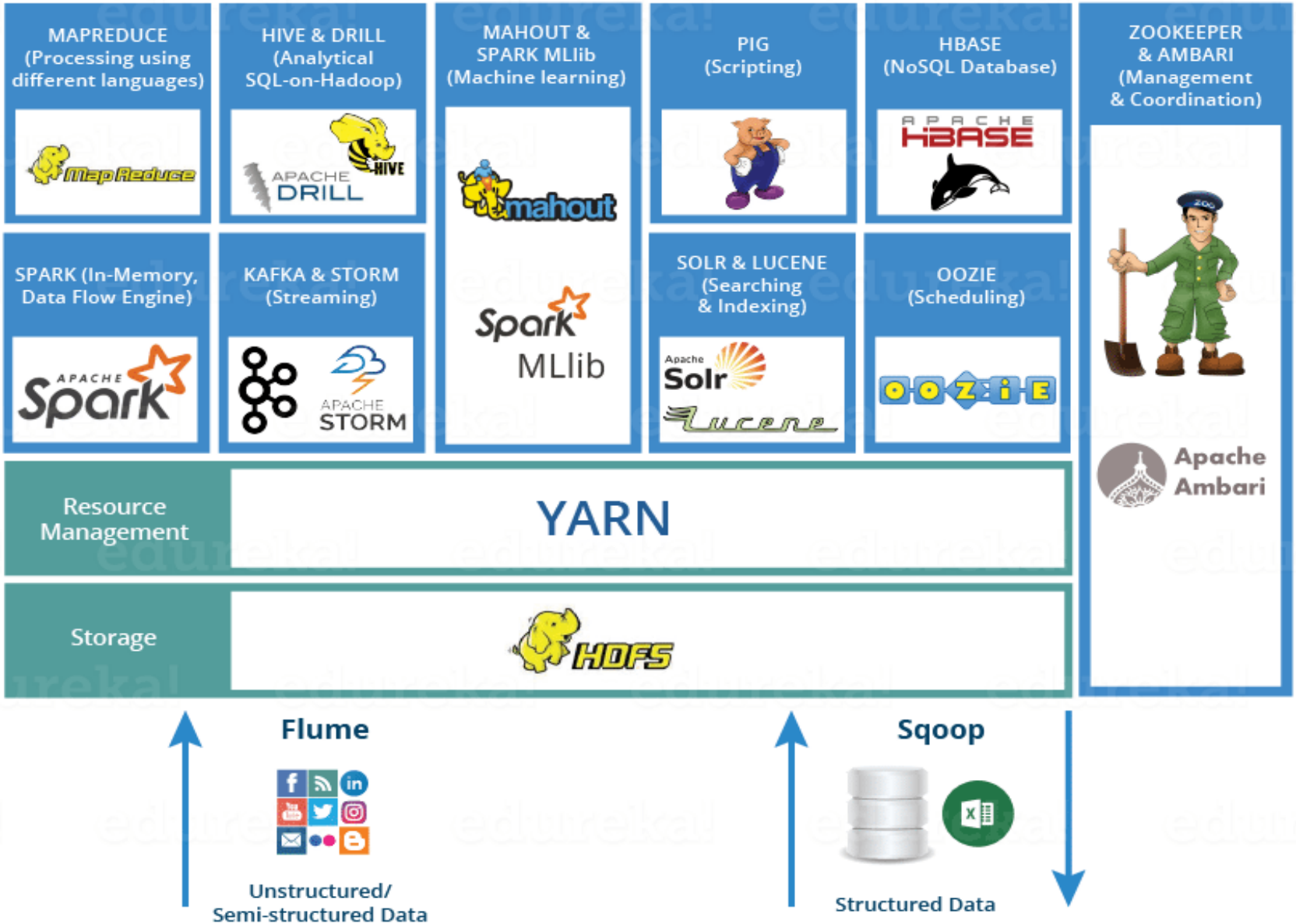
- Managing the file system namespace:
  - Holds file/directory structure, metadata, file-to-block mapping, access permissions, etc. Coordinating file operations:
  - Directs clients to datanodes for reads and writes
  - No data is moved through the namenode
- Maintaining overall health:
  - Periodic communication with the datanodes
  - Block re-replication and rebalancing
  - Garbage collection

- **Q**: What if the namenode's data is lost?
- **A**: All files on the filesystem cannot be retrieved since there is no way to reconstruct them from the raw block data. Fortunately, Hadoop provides 2 ways of improving resilience, through backups and secondary namenodes (out of syllabus, but you can refer to Resources for details)

Putting everything together…



Hadoop Ecosystem



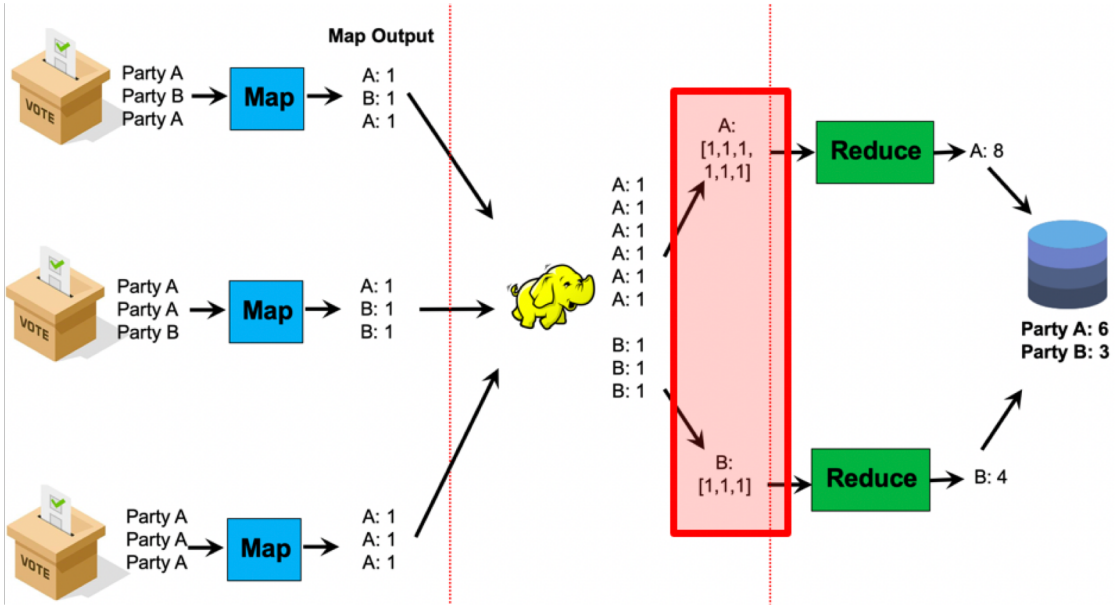
Q: Which statements about Hadooop's partitioner are correct?

1. The partitioner determines which keys are processed on which mappers.
2. The partitioner can improve performance when some keys are much more common than others.
3. The partitioner is run in between the map and reduce phases.

Q: Which statements about Hadooop's partitioner are correct?

1. The partitioner determines which keys are processed on which mappers.
2. The partitioner can improve performance when some keys are much more common than others.
3. The partitioner is run in between the map and reduce phases.

- **A:** F, T, T



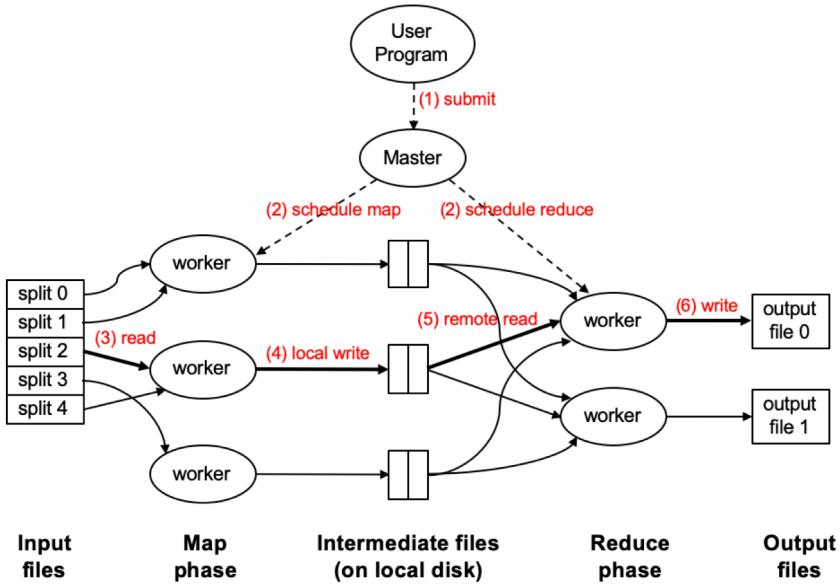
Q: True or false: the Shuffle stage of MapReduce is run within the Master node.

1. True
2. False

Q: True or false: the Shuffle stage of MapReduce is run within the Master node.

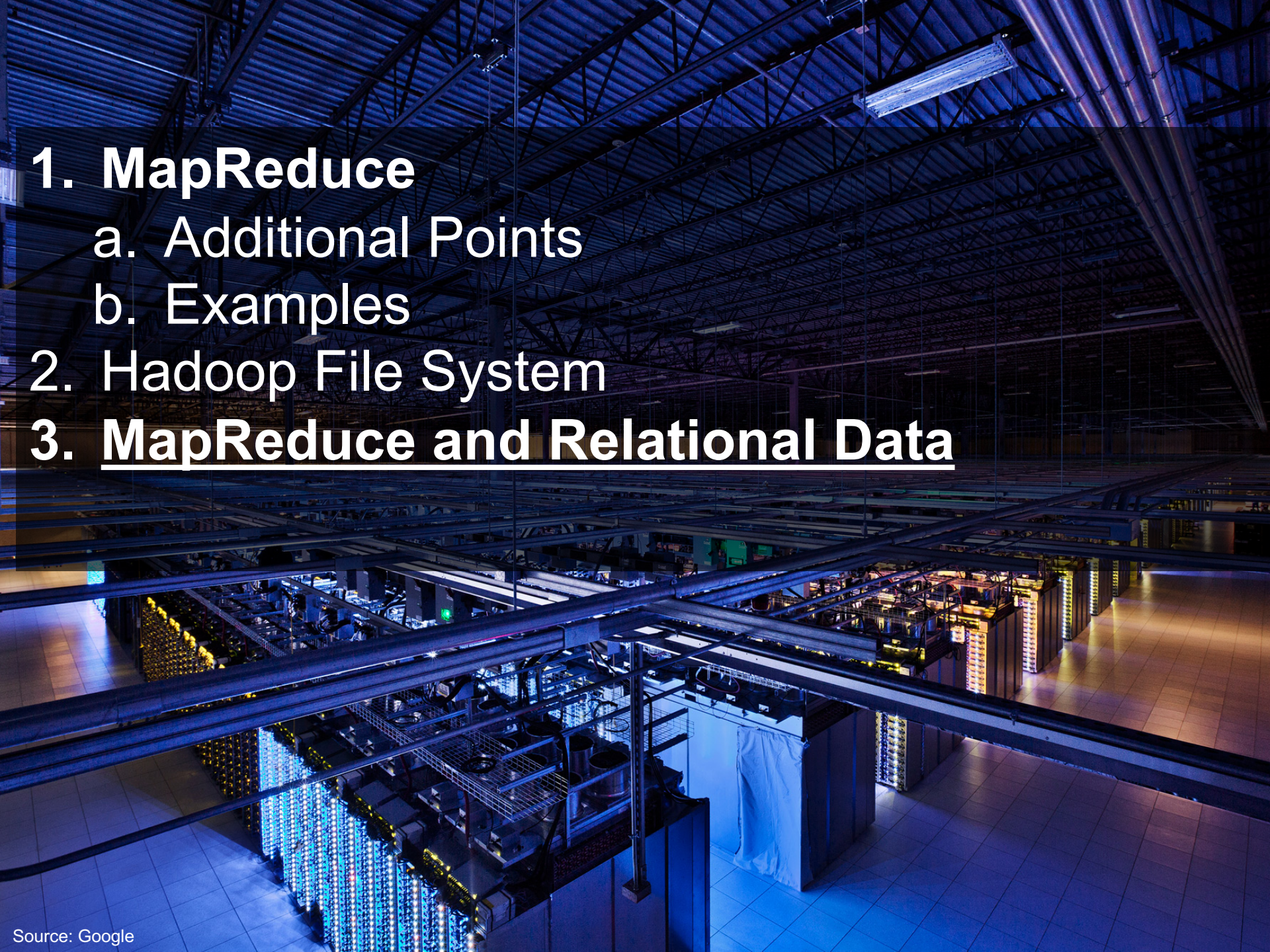
1. True
2. False

A: False (it runs in the worker nodes)



Resources

- Hadoop: The Definitive Guide (by Tom White)
- Hadoop Wiki
  - Introduction
    - http://wiki.apache.org/lucene-hadoop/
  - Getting Started
    - http://wiki.apache.org/lucene-hadoop/GettingStartedWithHadoop
  - Map/Reduce Overview
    - http://wiki.apache.org/lucene-hadoop/HadoopMapReduce
    - http://wiki.apache.org/lucene-hadoop/HadoopMapRedClasses
  - Eclipse Environment
    - http://wiki.apache.org/lucene-hadoop/EclipseEnvironment
- Javadoc
  - http://lucene.apache.org/hadoop/docs/api/
- YARN
  - https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

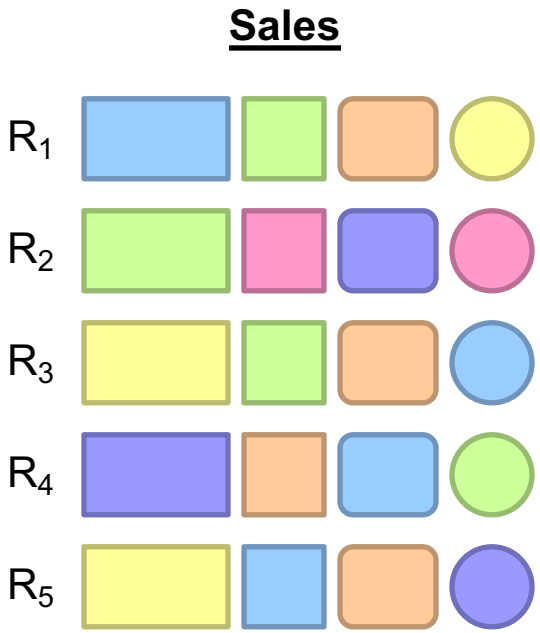


1. **MapReduce**
   a. Additional Points
   b. Examples
2. Hadoop File System
3. **MapReduce and Relational Data**

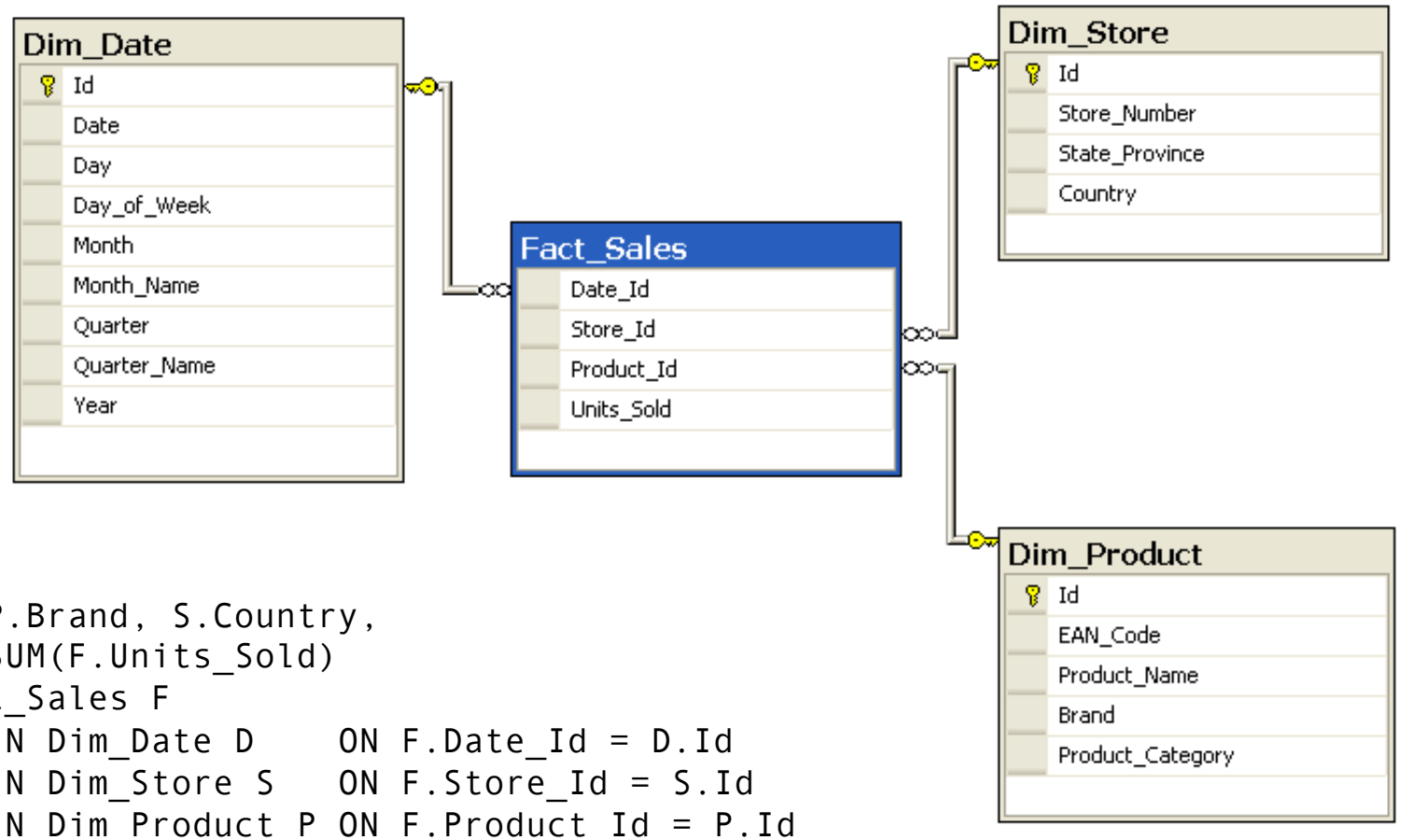
Source: Google

Relational Databases

- A relational database is comprised of tables.
- Each table represents a relation = collection of tuples (rows).
- Each tuple consists of multiple fields.



Structure of Data Warehouses

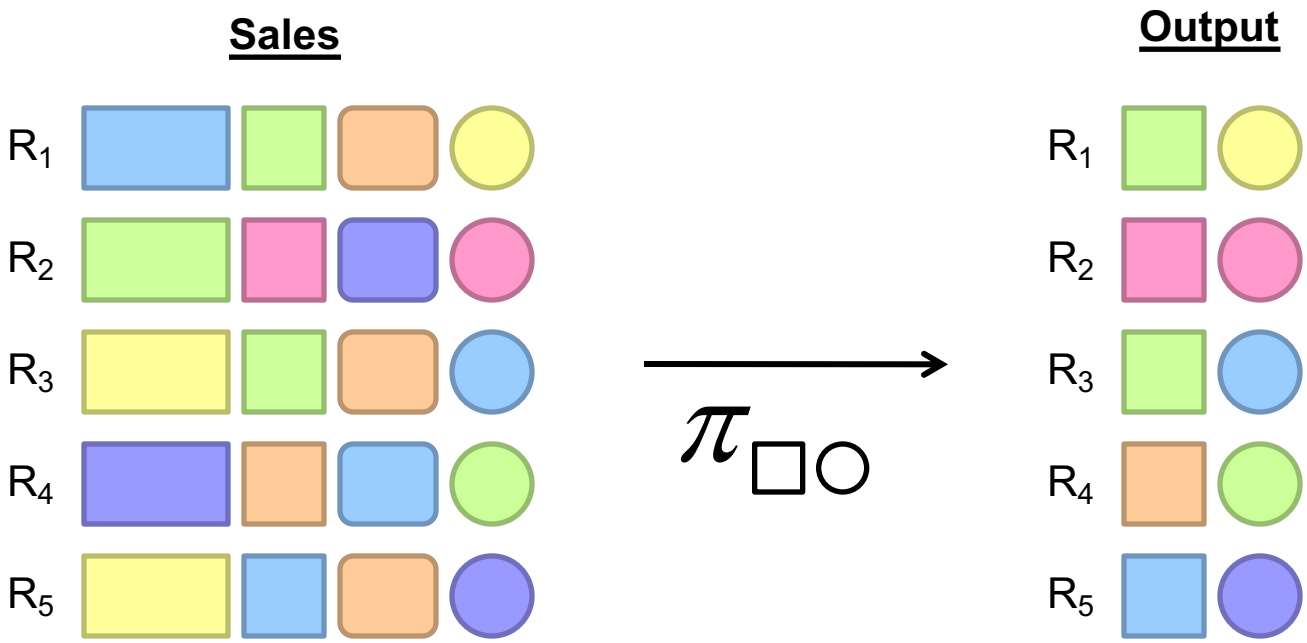



```
SELECT  P.Brand, S.Country,
        SUM(F.Units_Sold)
FROM Fact_Sales F
INNER JOIN Dim_Date D    ON F.Date_Id = D.Id
INNER JOIN Dim_Store S   ON F.Store_Id = S.Id
INNER JOIN Dim_Product P ON F.Product_Id = P.Id
WHERE D.YEAR = 1997 AND P.Product_Category = 'tv'
GROUP BY P.Brand, S.Country;
```


Source: Wikipedia (Star Schema)

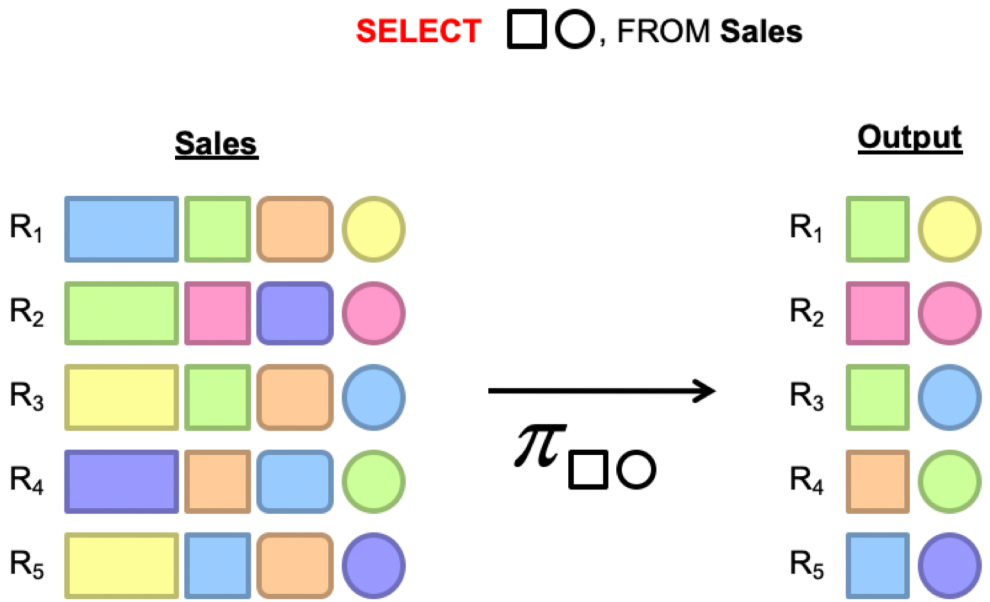
Projection

SELECT □○, FROM **Sales**



Projection in MapReduce

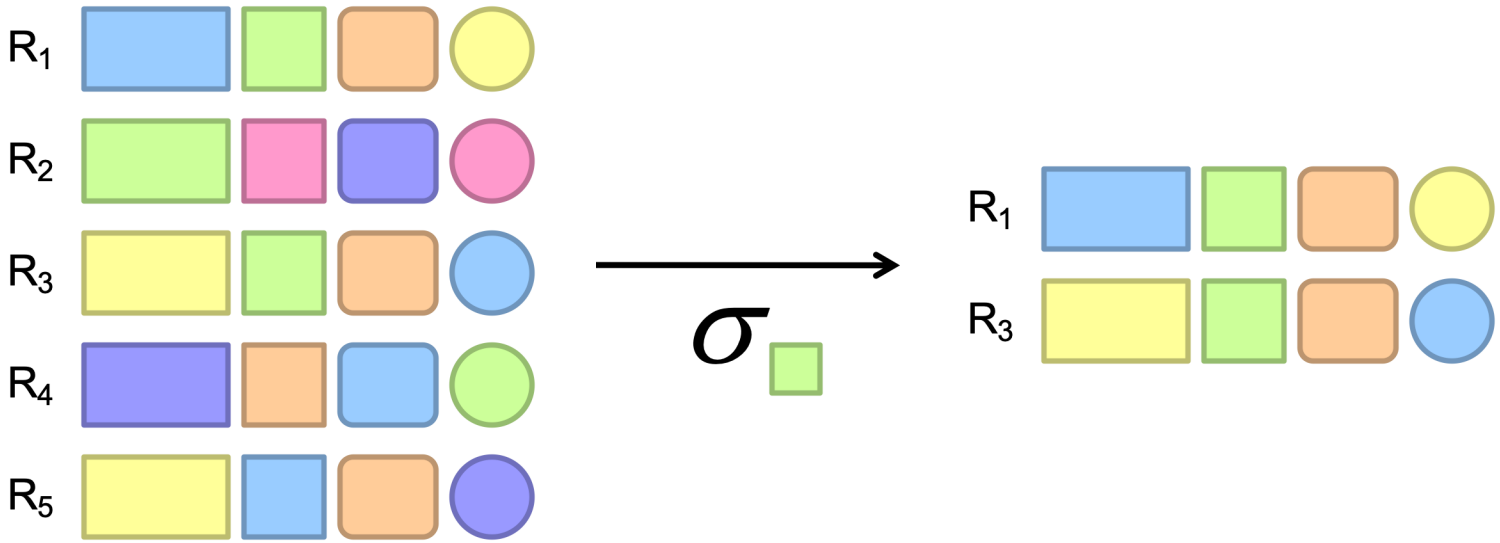
- **Map**: take in a tuple (with tuple ID as key), and emit new tuples with appropriate attributes
- No reducer needed (⇒ no need shuffle step)



Selection

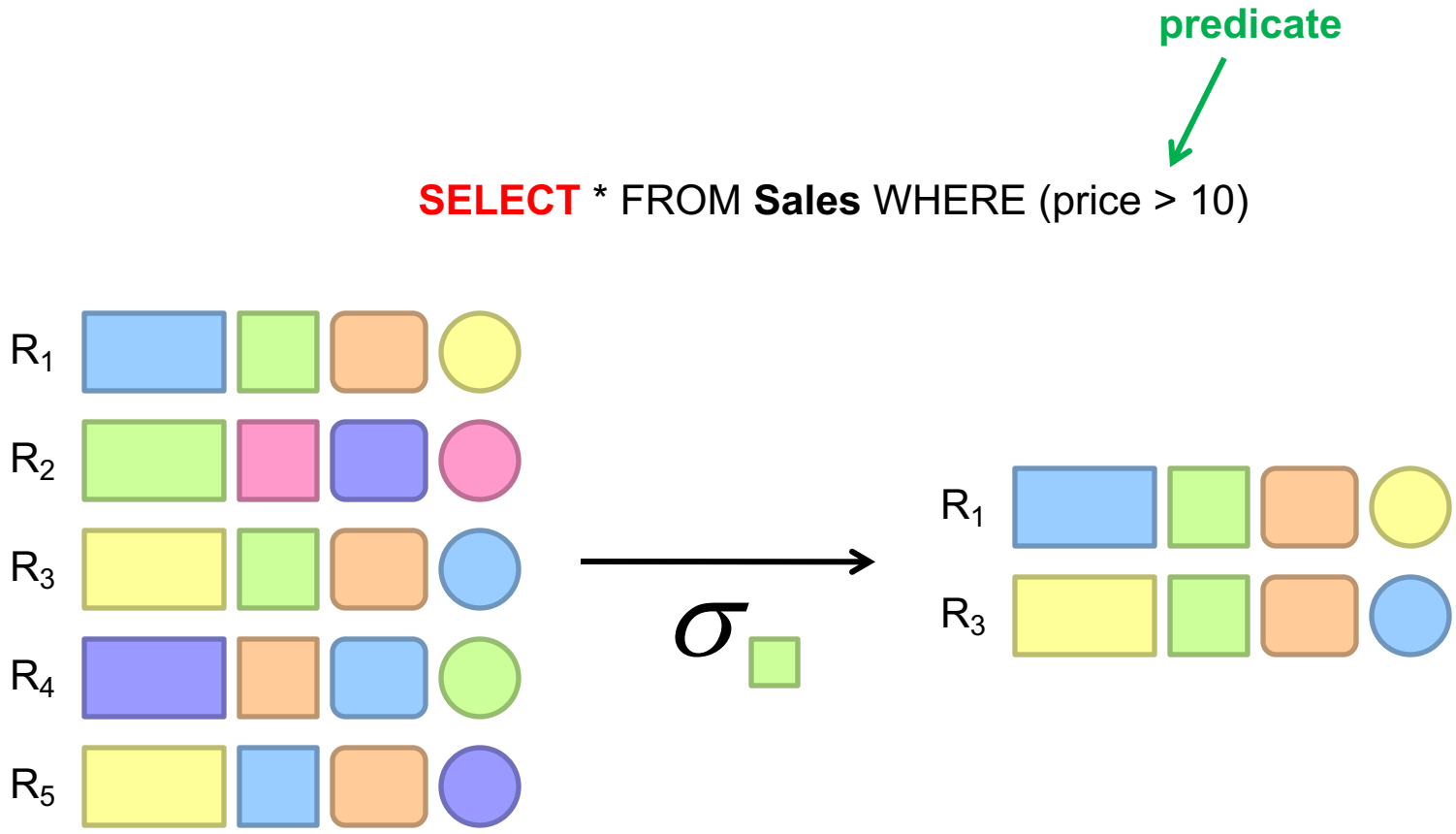
predicate

SELECT * FROM **Sales** WHERE (price > 10)



Selection in MapReduce

- **Map**: take in a tuple (with tuple ID as key), and emit only tuples that meet the predicate
- No reducer needed

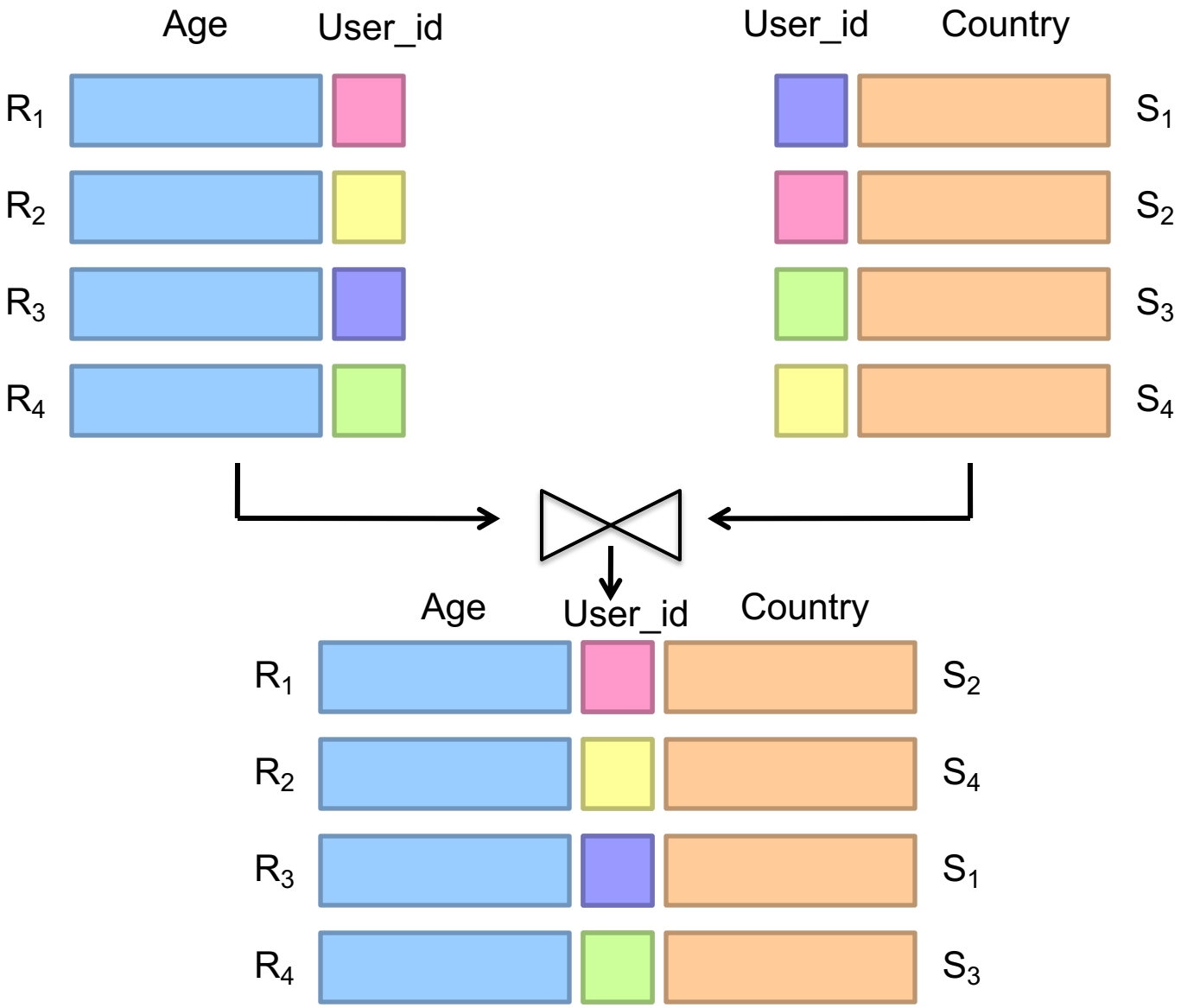


Group by… Aggregation

- Example: What is the average sale price per product?
- In SQL:
  - SELECT product_id, AVG(price) FROM sales GROUP BY product_id
- In MapReduce:
  - Map over tuples, emit <product_id, price>
  - Framework automatically groups values by keys
  - Compute average in reducer
  - Optimize with combiners

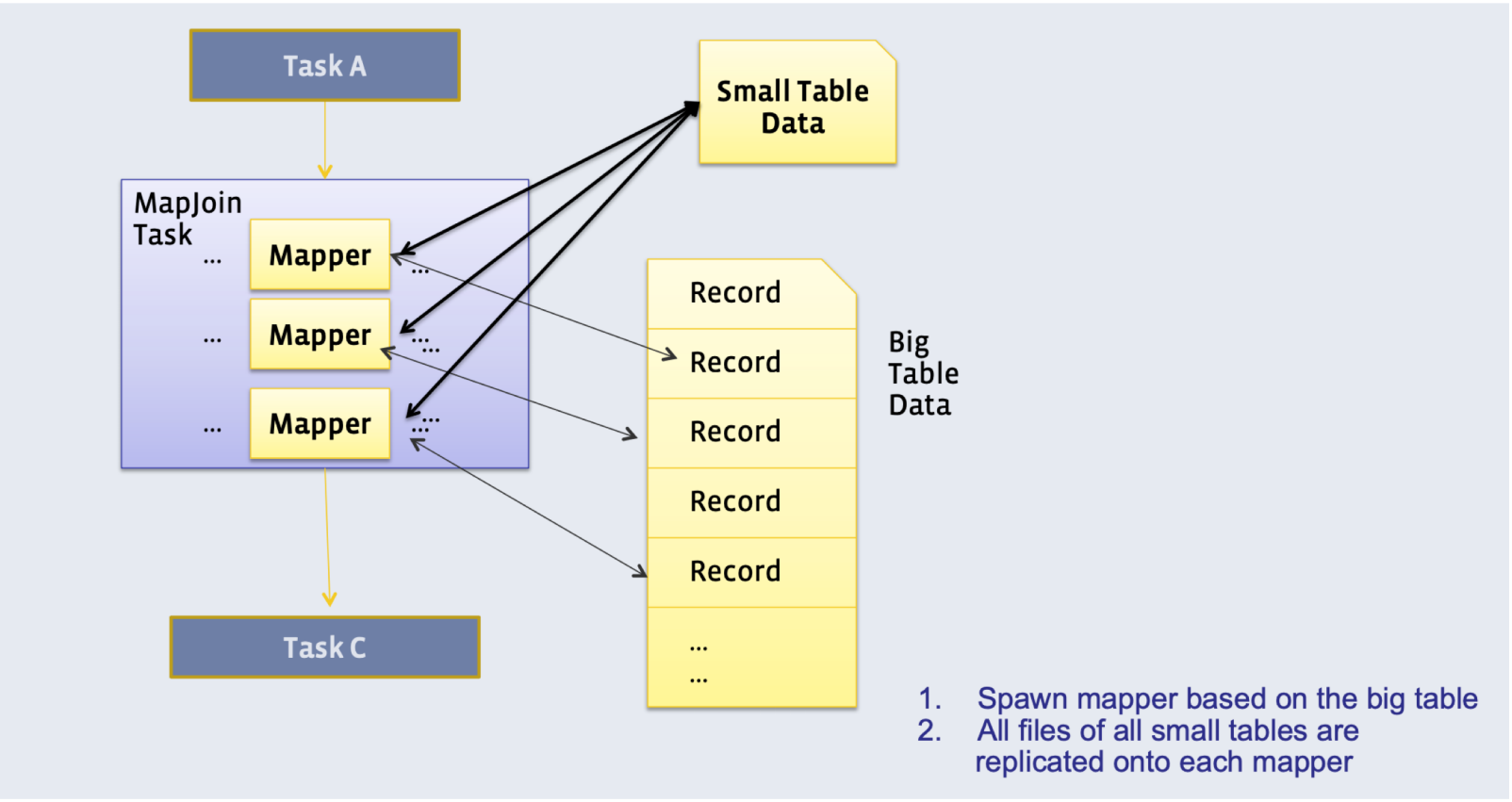
Relational Joins

Relational Joins ('Inner Join')



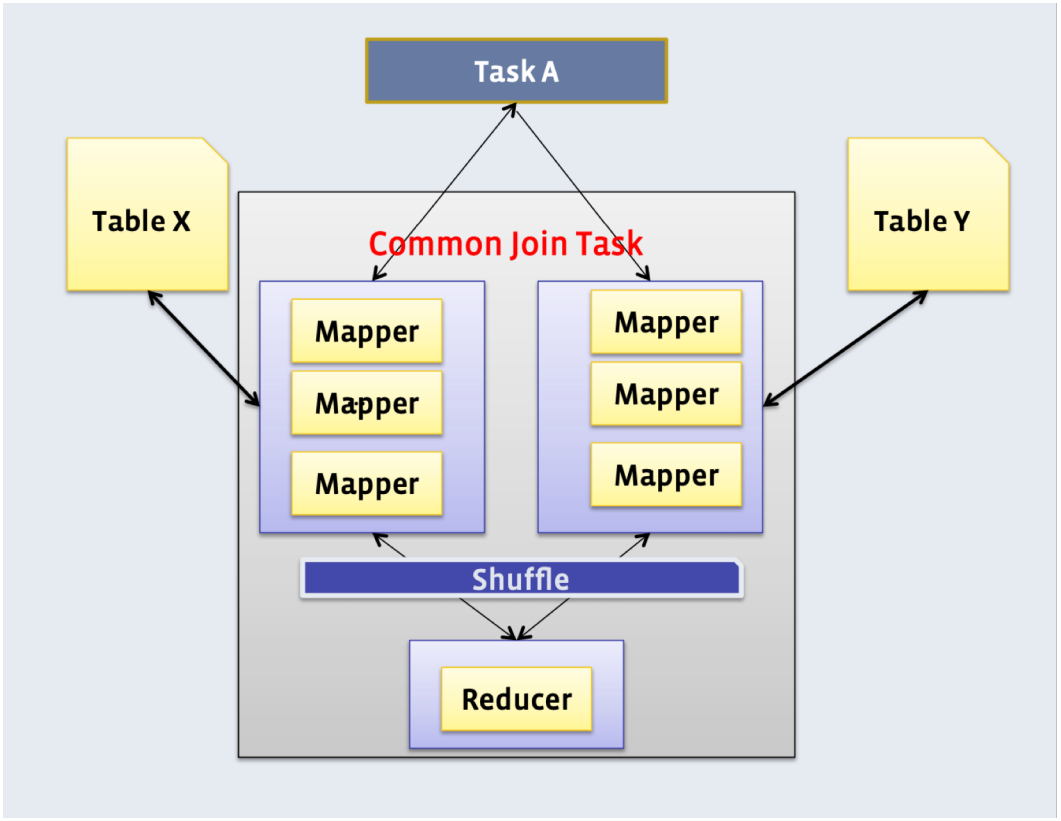
Method 1: Map-Side Join

- Requires one of the tables to fit in memory
  - All mappers store a copy of the small table
  - They iterate over the big table, and join the records with the small table



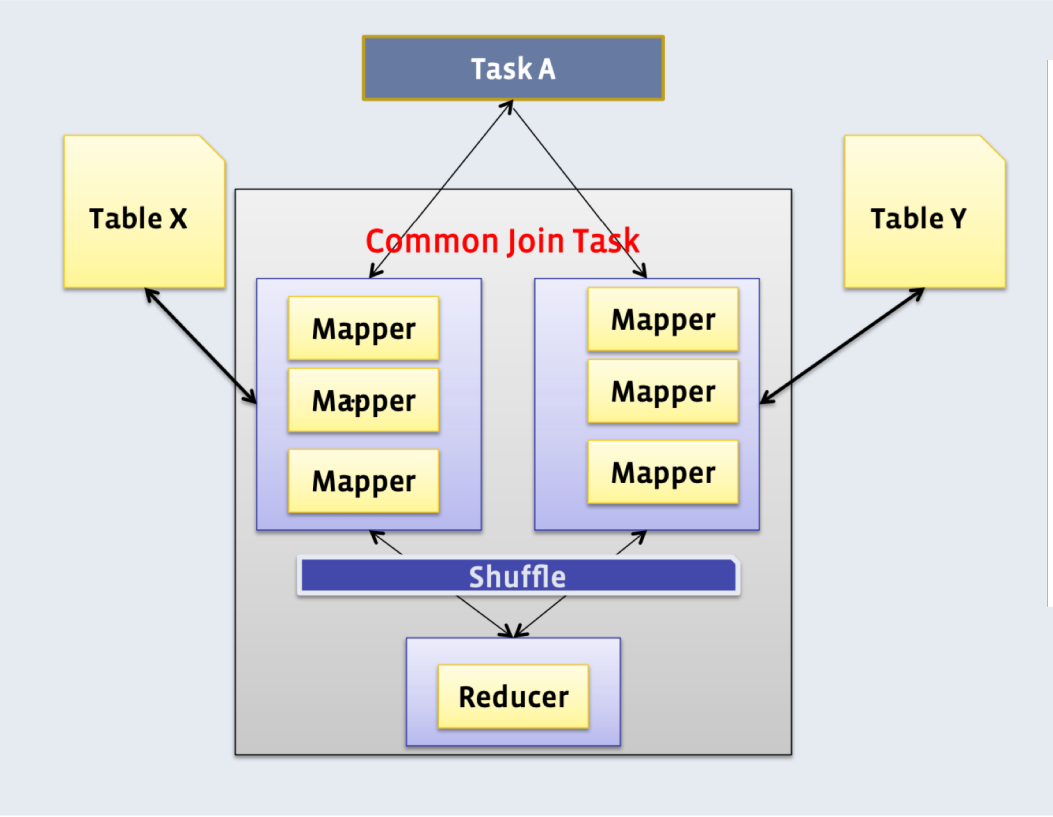
Method 2: Reduce-side ('Common') Join

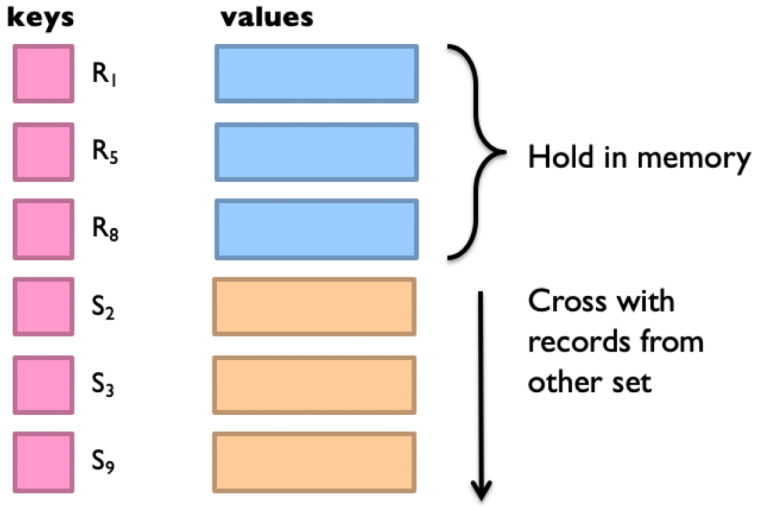
- Doesn't require a dataset to fit in memory, but slower than map-side join
  - Different mappers operate on each table, and emit records, with key as the variable to join by



Method 2: Reduce-side ('Common') Join

- In reducer: we can use **secondary sort** to ensure that all keys from table X arrive before table Y
  - Then, hold the keys from table X in memory and cross them with records from table Y





Further Reading

- Chapter 6, "Data-Intensive Text Processing with MapReduce", by Jimmy Lin. http://lintool.github.io/MapReduceAlgorithms/ed1n/MapReduce-algorithms.pdf
- Foto N. Afrati and Jeffrey D. Ullman. 2010. Optimizing joins in a map-reduce environment. In *Proceedings of the 13th International Conference on Extending Database Technology* (EDBT '10). http://infolab.stanford.edu/~ullman/pub/join-mr.pdf.

Acknowledgement

- "Join Strategies in Hive" (Facebook; Liyin Tang, Namit Jain)
- "Data Algorithms", July 2015, O'Reilly Media, Inc.
- "Data-Intensive Text Processing with MapReduce", Jimmy Lin and Chris Dyer
- Original CS4225 slides by He Bingsheng