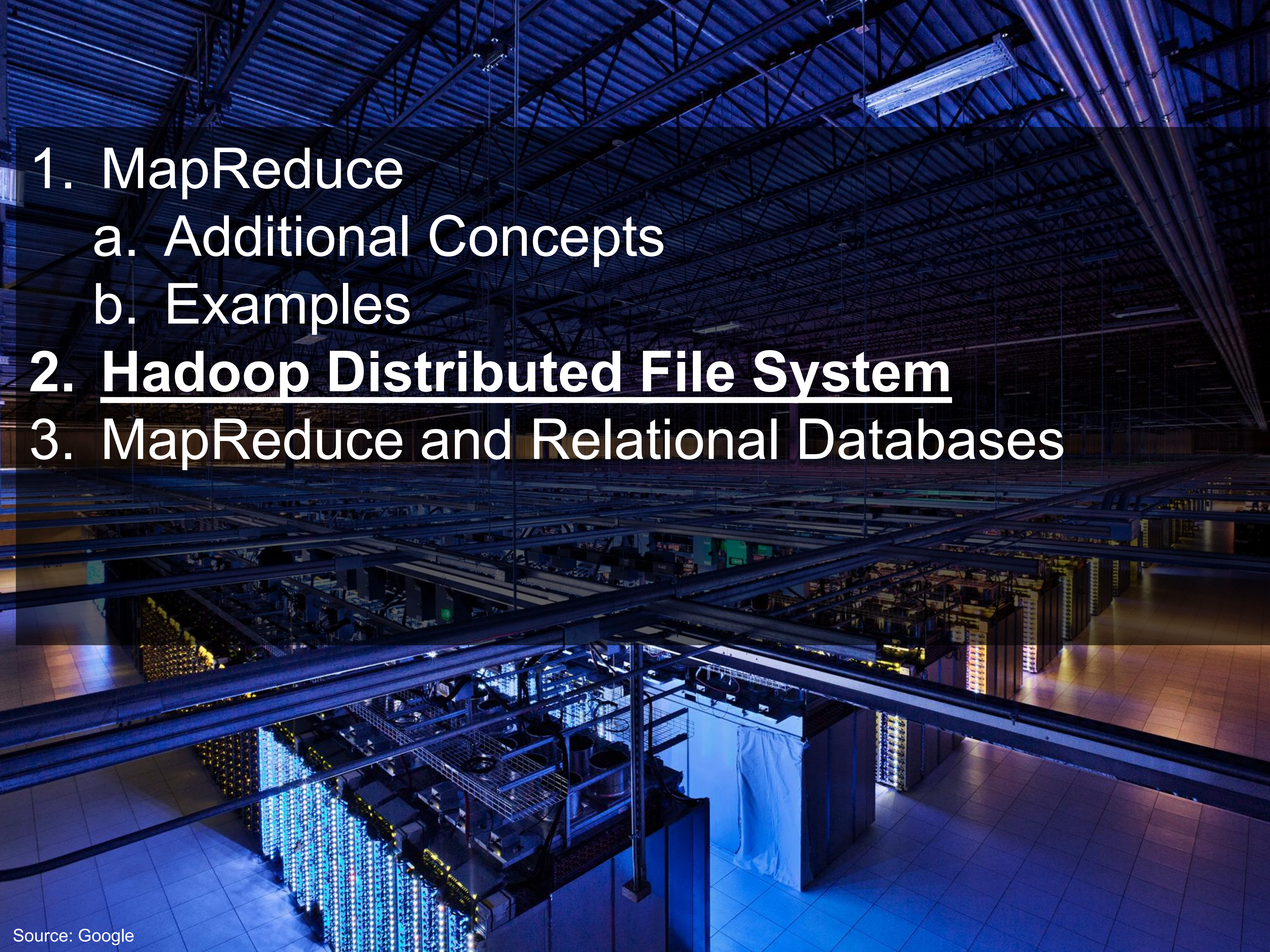
# CS4225/CS5425 Big Data Systems for Data Science

## MapReduce & Databases

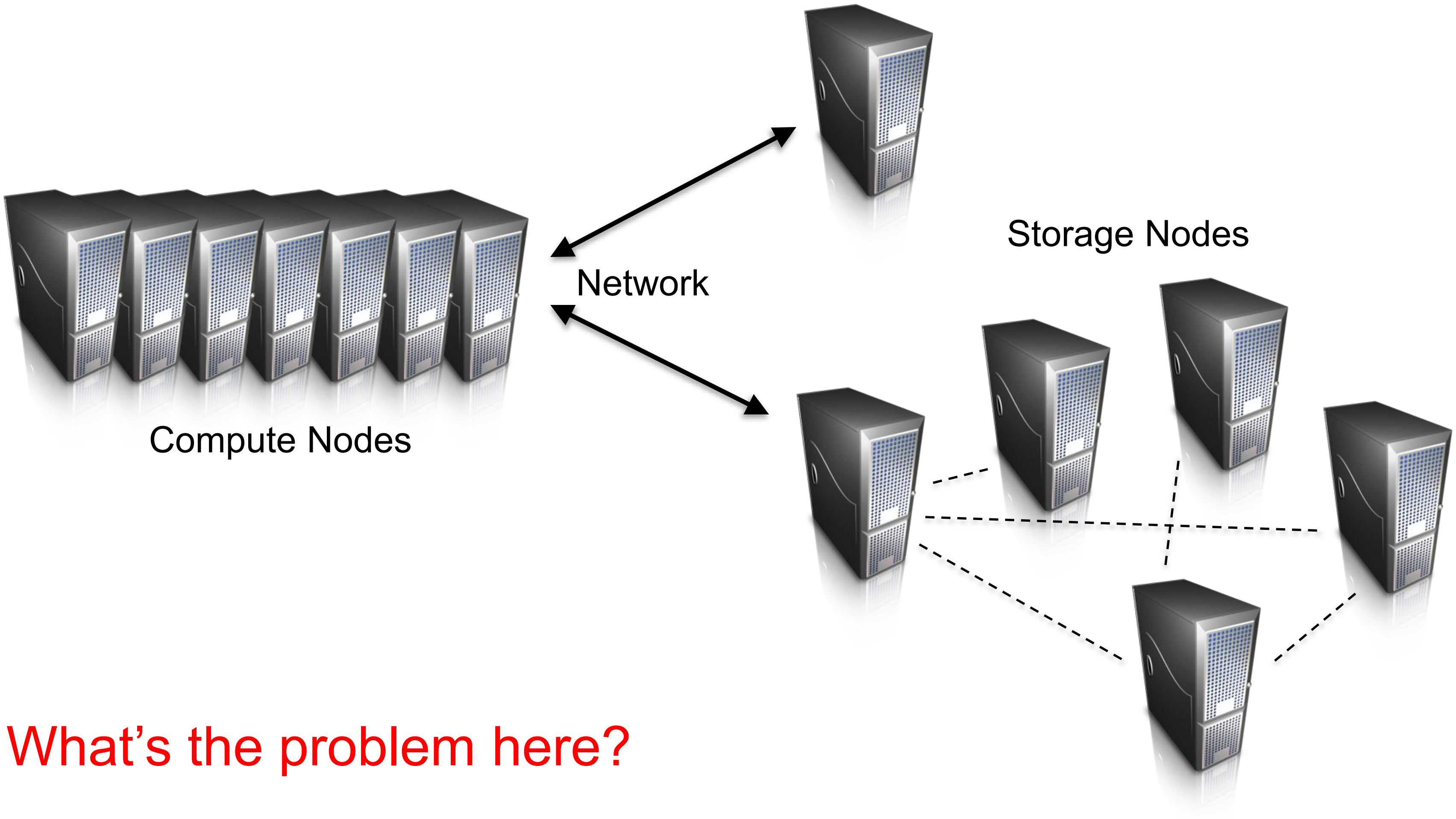
Bingsheng He
School of Computing
National University of Singapore
hebs@comp.nus.edu.sg



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

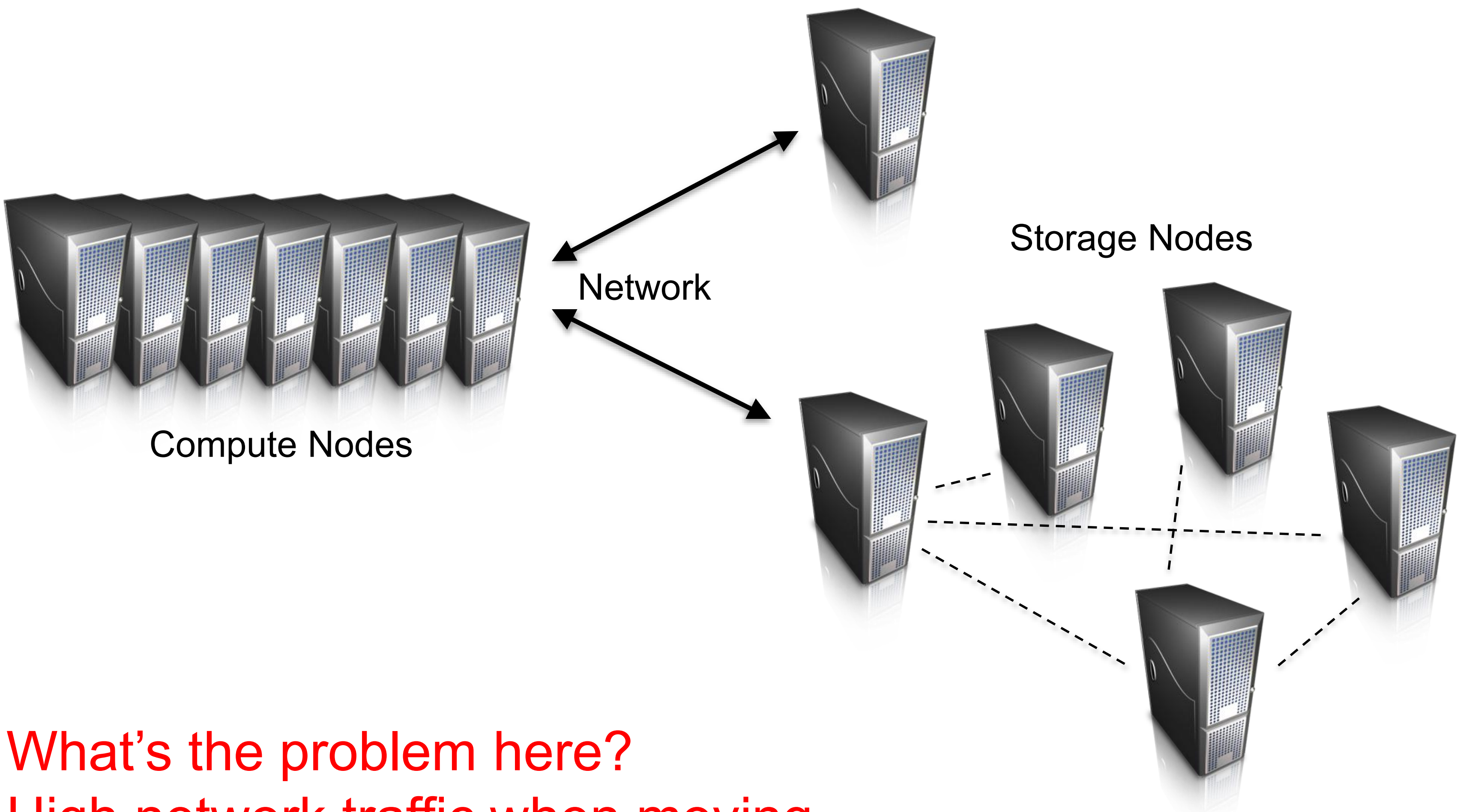
Source: Google

# How do we get data to the workers?




What's the problem here?

# How do we get data to the workers?




What's the problem here?
High network traffic when moving data to compute nodes!

# HDFS: Assumptions

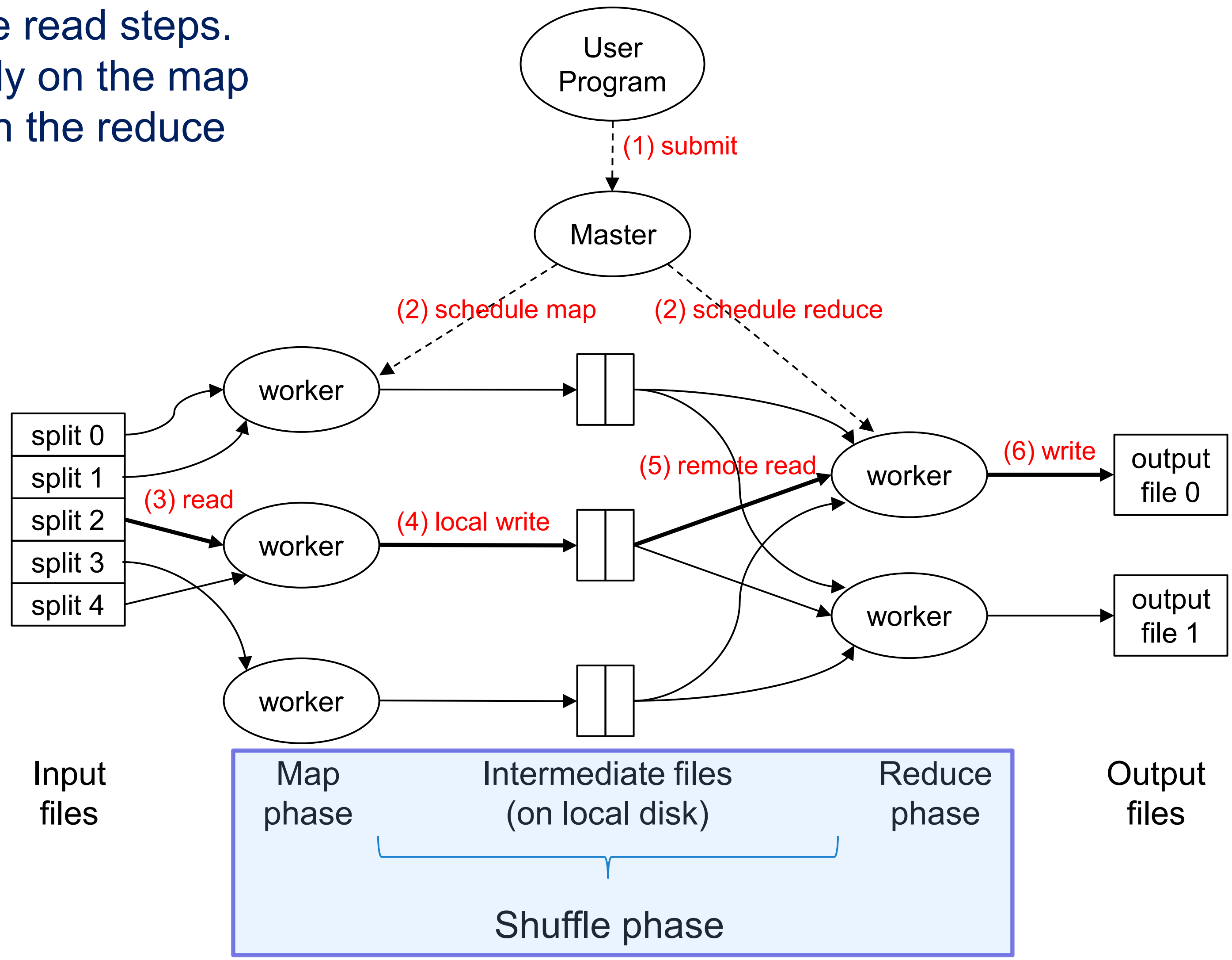
- Commodity hardware instead of “exotic” hardware
  - Scale “out”, not “up”
- High component failure rates
  - Inexpensive commodity components fail all the time
- “Modest” number of huge files
- Large sequential reads instead of random access
  - Leads to better throughput due to sequential access

# Design Decisions

- Files stored as chunks (or blocks)
  - Fixed size (by default 128MB)
- Reliability through replication
  - Each chunk is stored as multiple replicas (by default 3)
- Single master to coordinate access, keep metadata
  - Simple centralized management

# [Recap] MapReduce Implementation

Clarification of "shuffle phase": the "shuffle phase" is comprised of the local write and remote read steps. Thus, it happens partly on the map workers, and partly on the reduce workers.



# [Recap] HDFS: 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
- Large sequential reads instead of random access
  - Leads to better throughput due to sequential access

# [Recap] Design Decisions

- Files stored as chunks (or blocks)
  - Fixed size (by default 128MB)
- Reliability through replication
  - Each chunk is stored as multiple replicas (by default 3)
- Single master to coordinate access, keep metadata
  - Simple centralized management

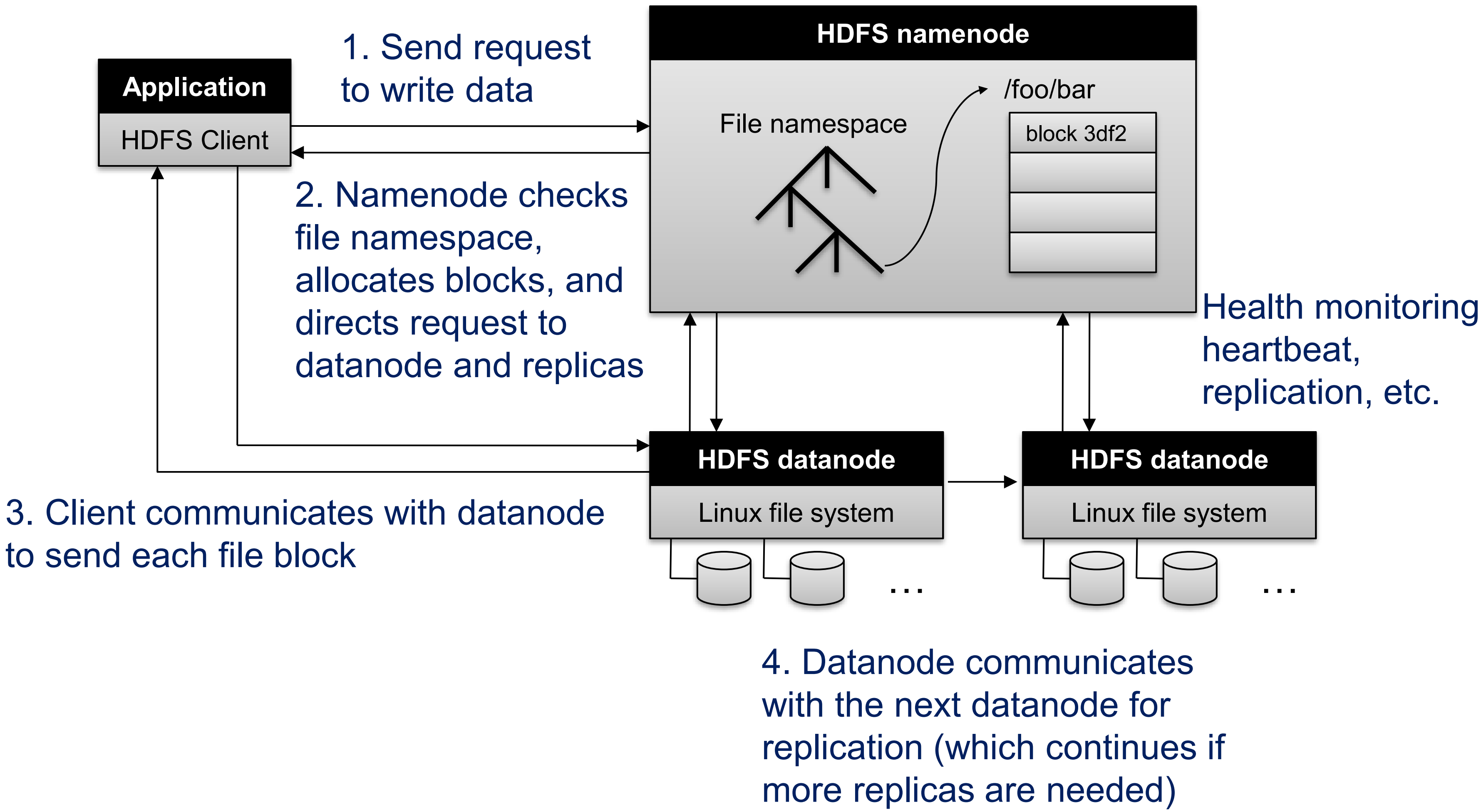
# HDFS: Interface

- Mostly similar to the interface of regular file systems (`-ls`, `-rm`, `-mkdir`, `-cat` etc.), with extra commands to transfer files to and from the local filesystem
- Designed for **write-once, read-many**
  - Does not support modifying files, other than appending
- Can handle very large files, while providing fault tolerance and replication

| Command | Description |
|---|---|
| -rm | Removes file or directory |
| -ls | Lists files with permissions and other details |
| -mkdir | Creates a directory named path in HDFS |
| -cat | Shows contents of the file |
| -rmdir | Deletes a directory |
| -put | Uploads a file or folder from a local disk to HDFS |
| -rmr | Deletes the file identified by path or folder and subfolders |
| -get | Moves file or folder from HDFS to local file |
| -count | Counts number of files, number of directory, and file size |
| -df | Shows free space |
| -getmerge | Merges multiple files in HDFS |
| -chmod | Changes file permissions |
| -copyToLocal | Copies files to the local system |
| -Stat | Prints statistics about the file or directory |
| -head | Displays the first kilobyte of a file |
| -usage | Returns the help for an individual command |
| -chown | Allocates a new owner and group of a file |

Note: exact functionality / syntax is out of scope; no need to memorize

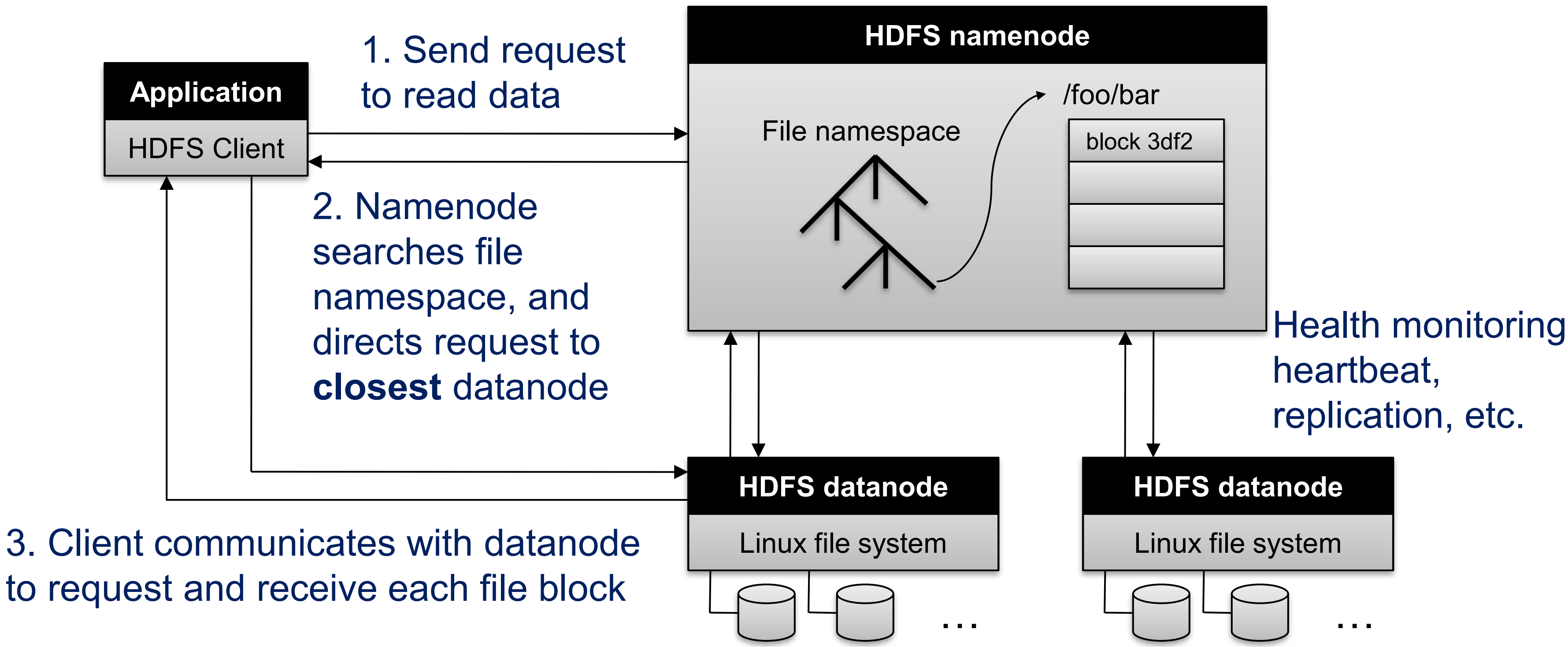
# HDFS Architecture: Writing Data






Adapted from (Ghemawat et al., SOSP 2003)

# HDFS Architecture: Reading Data



# Namenode Responsibilities

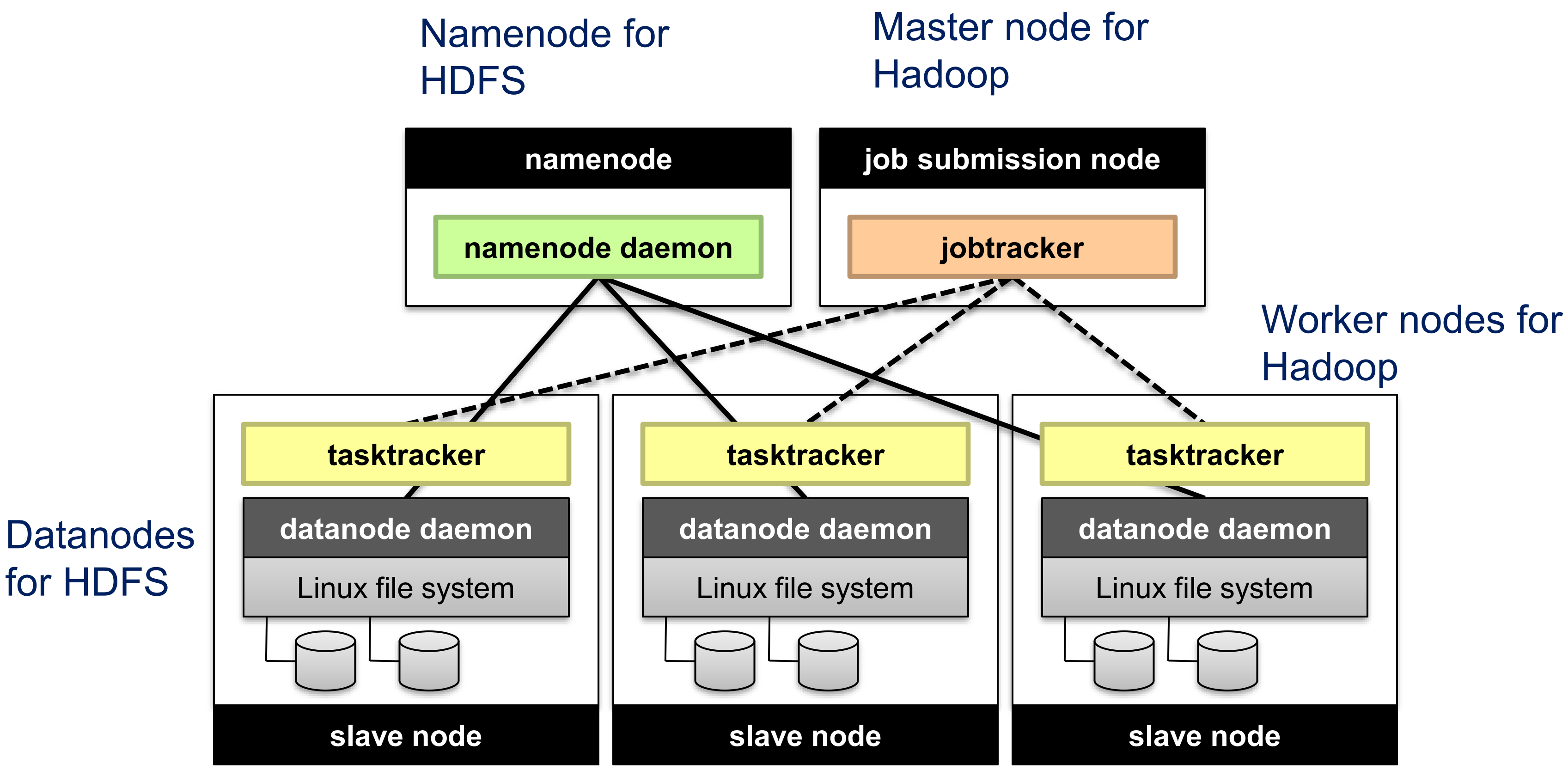
- Managing the file system namespace:
  - Holds file/directory structure, metadata, file-to-block mapping, access permissions, etc.
  - Directs clients to datanodes for reads and writes
  - No data is moved through the namenode
- Maintaining overall health:
  - “Heartbeat” (periodic communication with the datanodes)
  - Block rebalancing (ensure data is distributed close to evenly)
- 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.
  - Directs clients to datanodes for reads and writes
  - No data is moved through the namenode
- Maintaining overall health:
  - “Heartbeat” (periodic communication with the datanodes)
  - Block rebalancing (ensure data is distributed close to evenly)

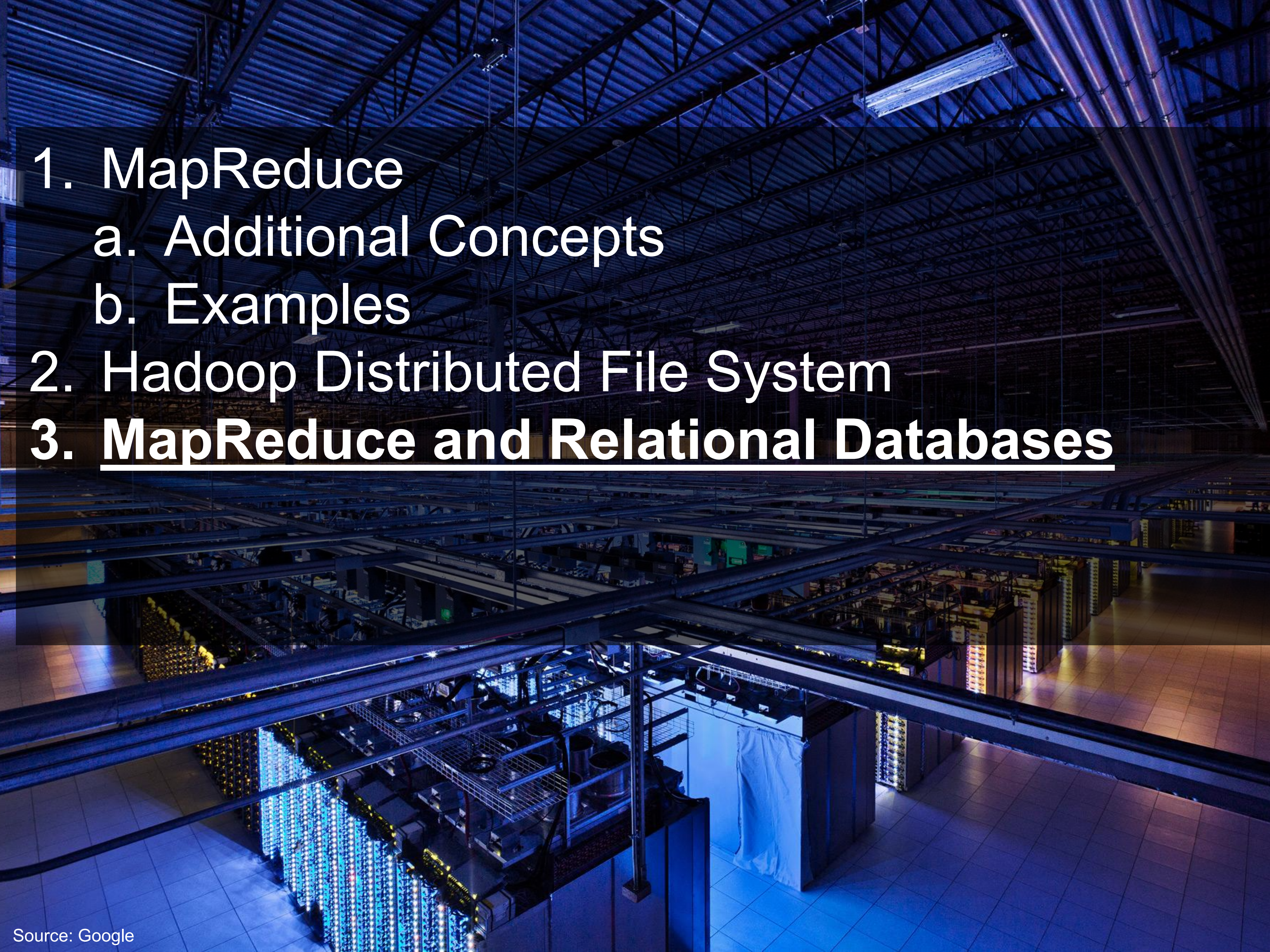
- Q: What if the namenode’s data is lost?
- A: All files on the filesystem cannot be retrieved since there is no way to reconstruct them from the raw block data.
  - Fortunately, Hadoop provides 2 ways of improving resilience, through backups and checkpointing (out of scope, but you can refer to Resources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

# Putting everything together…




The same set of nodes are used for both HDFS and Hadoop. Hadoop tries to schedule map tasks to run on the machines that already contain the needed data (called **data locality**; or "moving the task to the data")

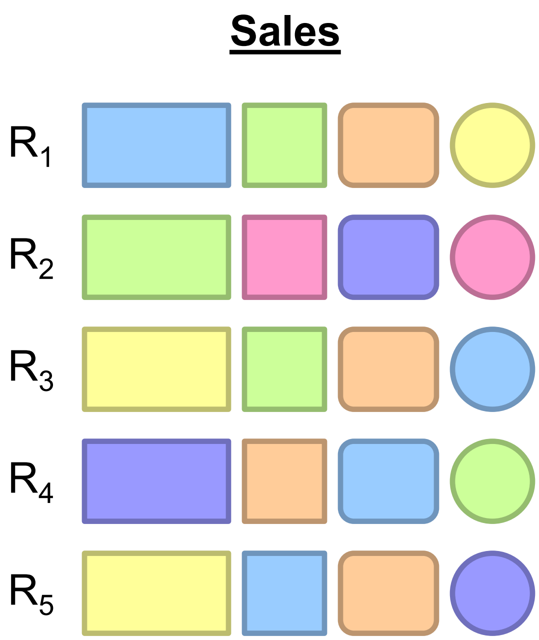
1. MapReduce
    a. Additional Concepts
    b. Examples
2. Hadoop Distributed File System
3. **MapReduce and Relational Databases**



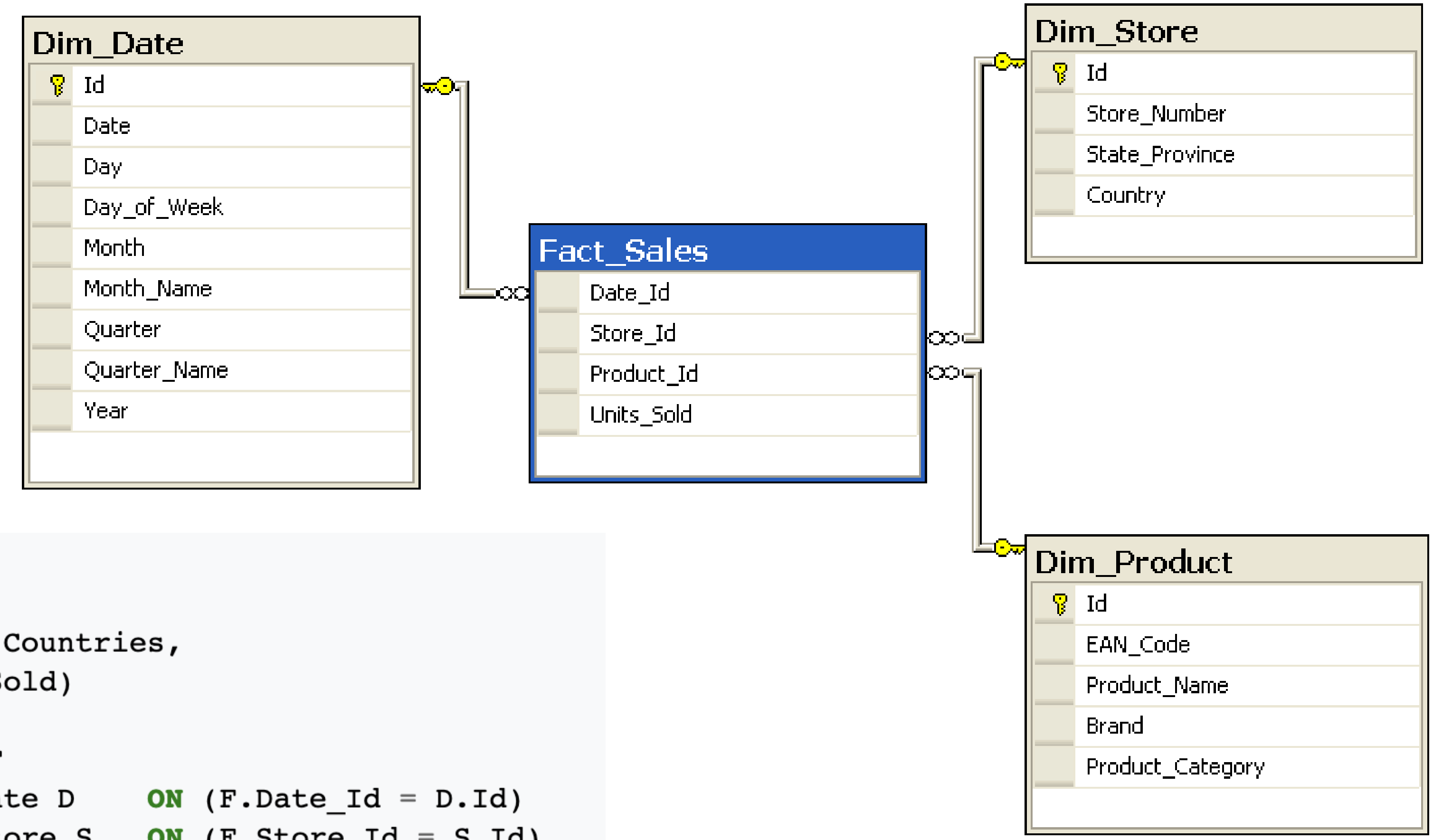
Source: Google

# Relational Databases

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



# Star Schema and SQL Queries



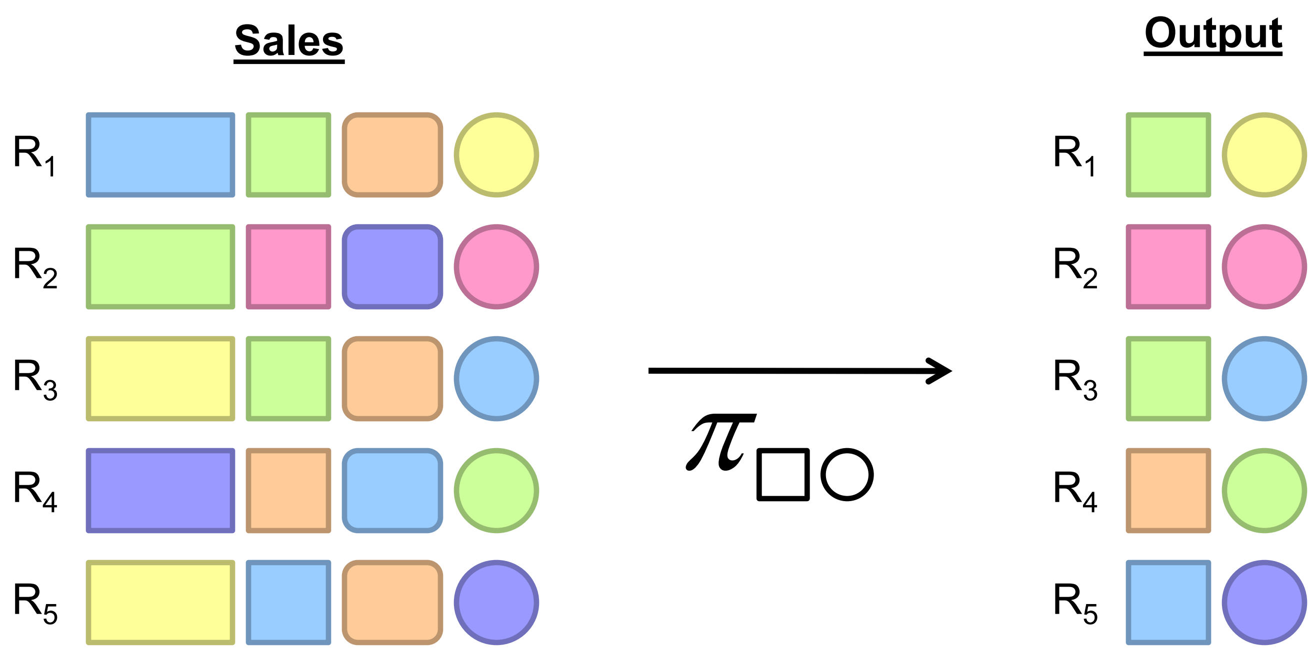

```sql
SELECT
    P.Brand,
    S.Country AS Countries,
    SUM(F.Units_Sold)

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)

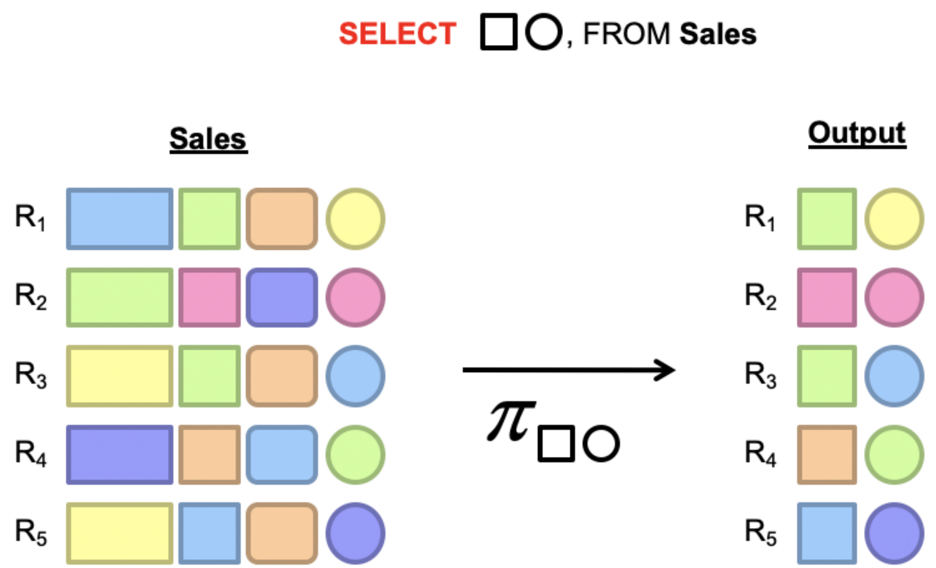
# Projection

**SELECT** □ ○, FROM **Sales**



# Projection in MapReduce

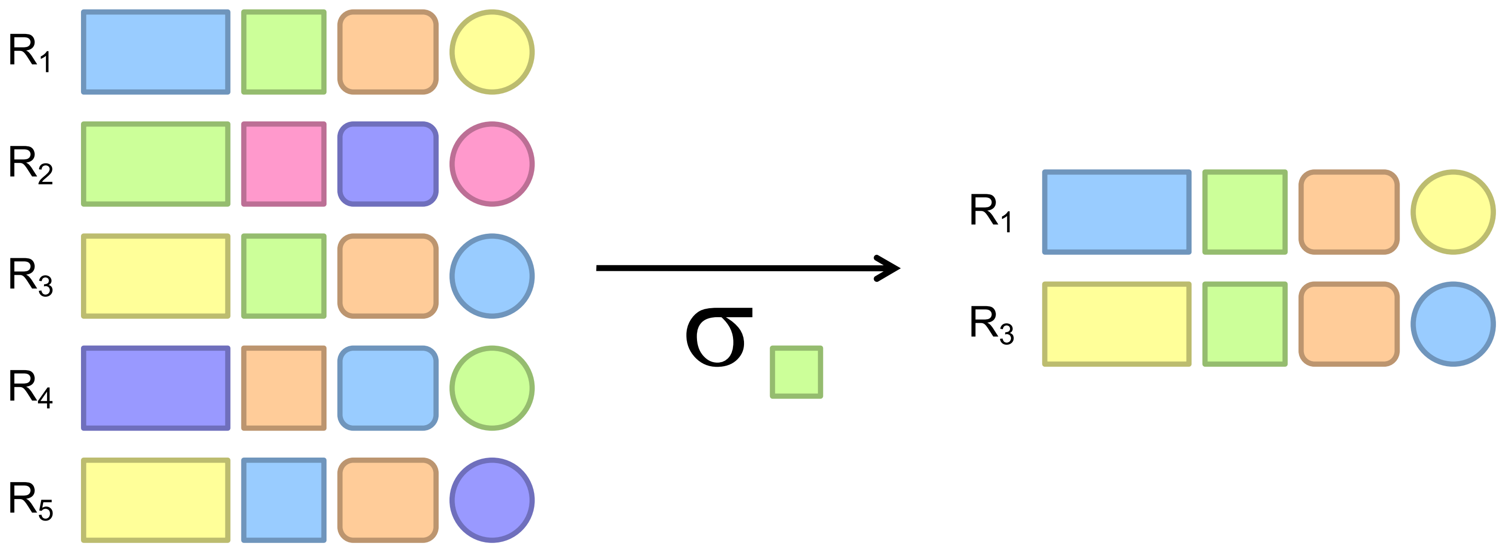
- **Map**: take in a tuple (with tuple ID as key), and emit new tuples with appropriate attributes
- **No reducer needed** (this is a **'map-only job':** the output of the map phase is directly used as the final output. There is no shuffle step, making this job more efficient)



# Selection

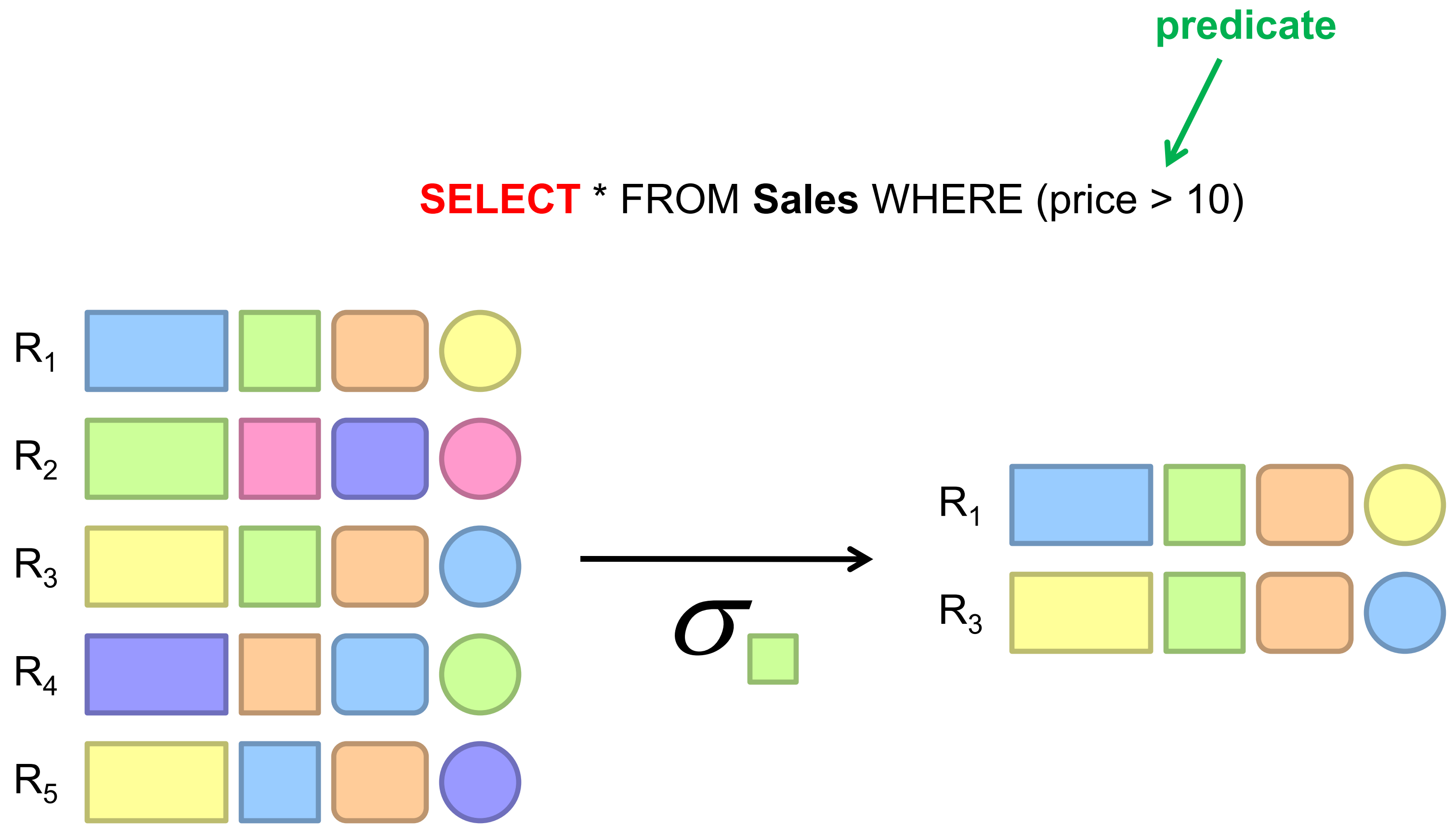
predicate

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



# Selection in MapReduce

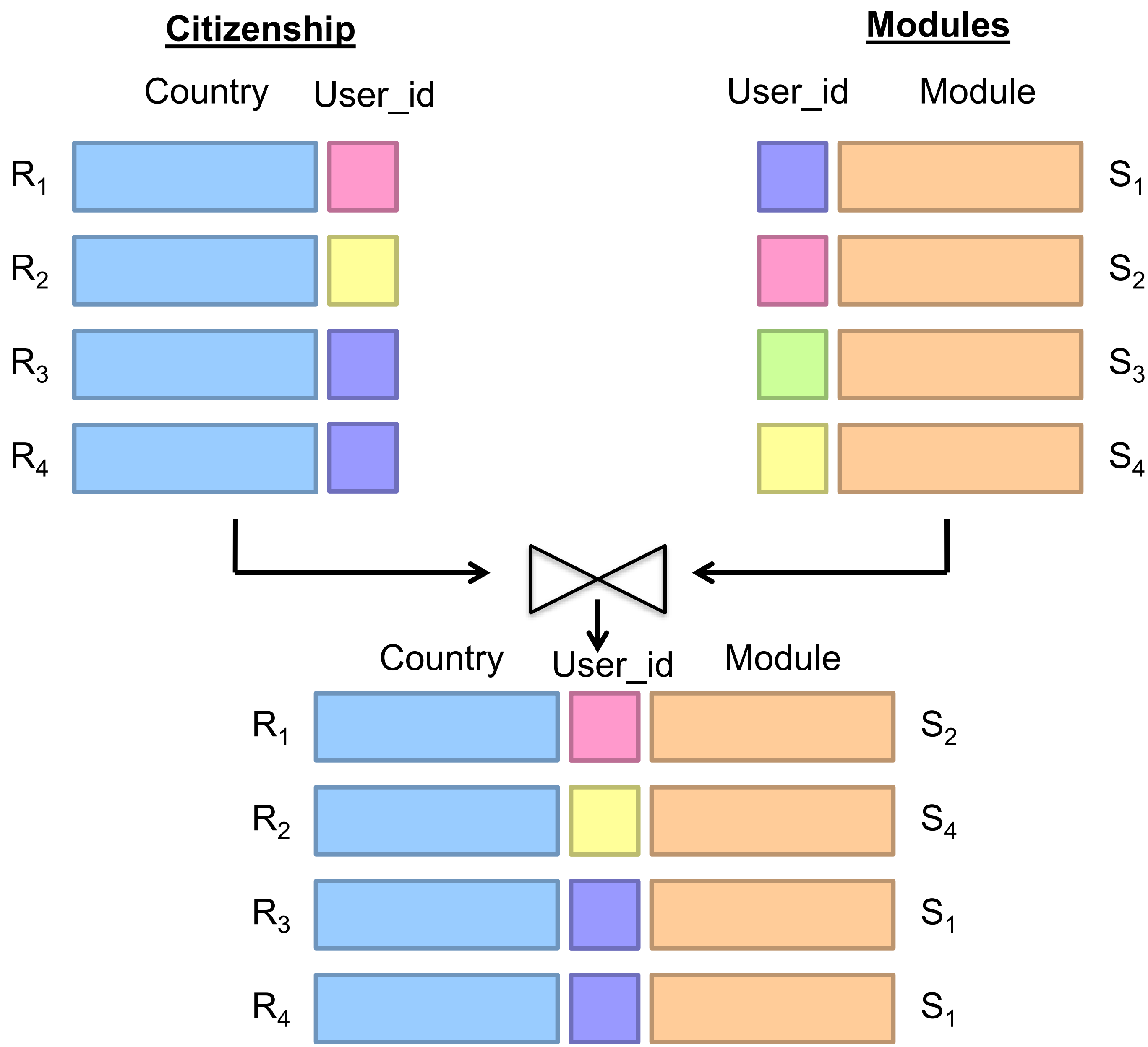
- **Map**: take in a tuple (with tuple ID as key), and emit only tuples that meet the predicate
- **No reducer needed**



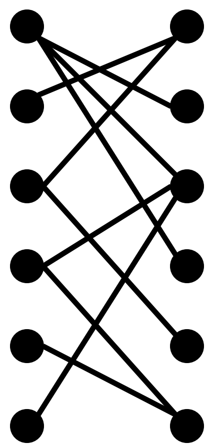
# Group by… Aggregation

- 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 these tuples by key
  - Compute average in reducer
  - Optimize with combiners

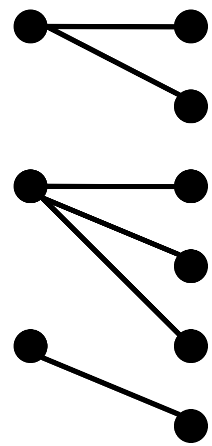
# Relational Joins (‘Inner Join’)



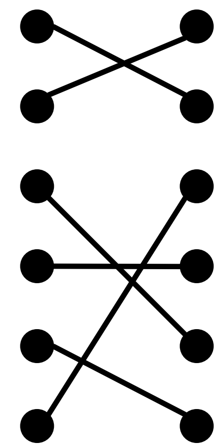
# Types of Relationships



**Many-to-Many**



**One-to-Many**

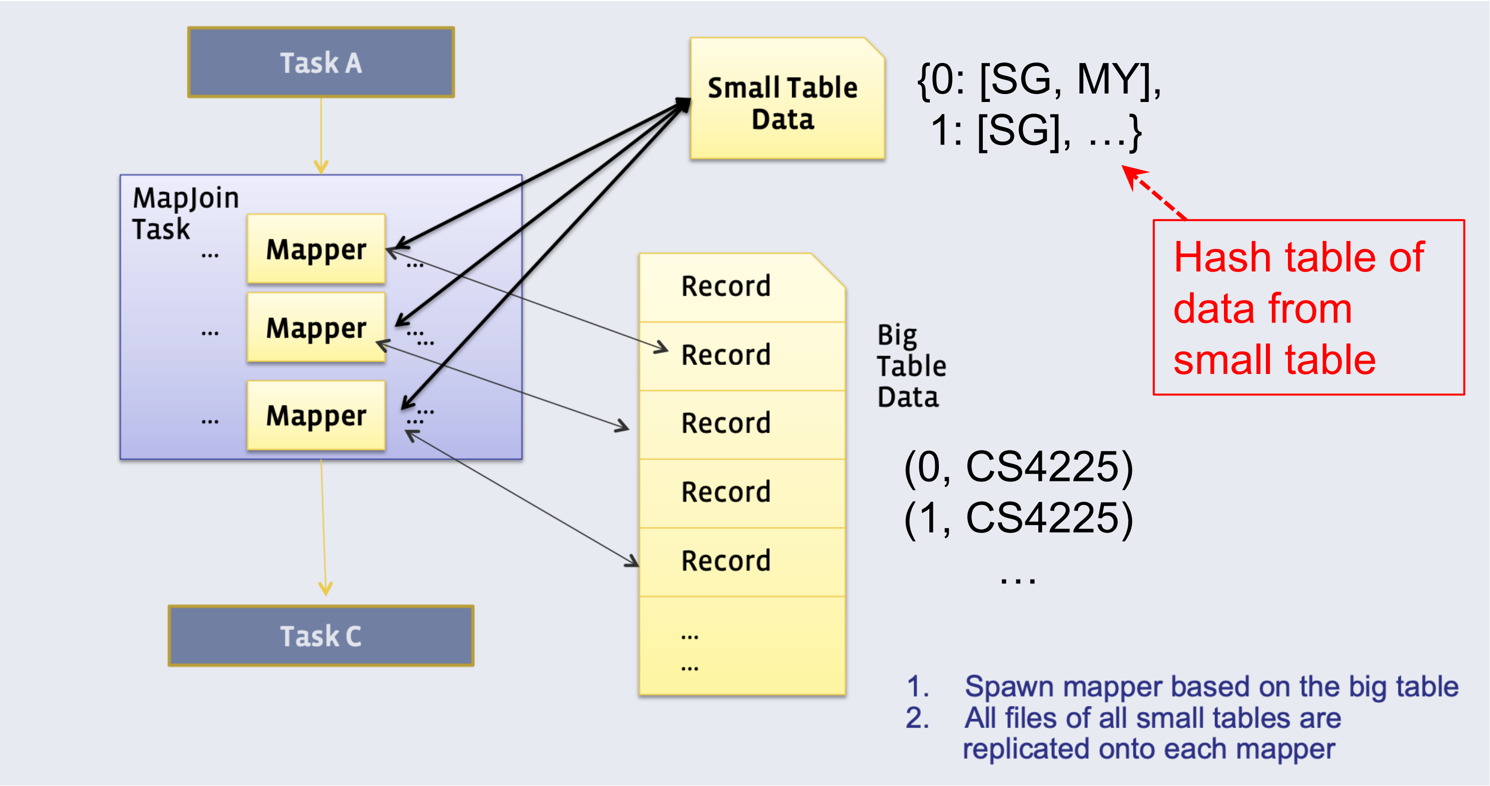


**One-to-One**

- This is a “many to many” relationship: a particular user ID can appear multiple times, in both tables

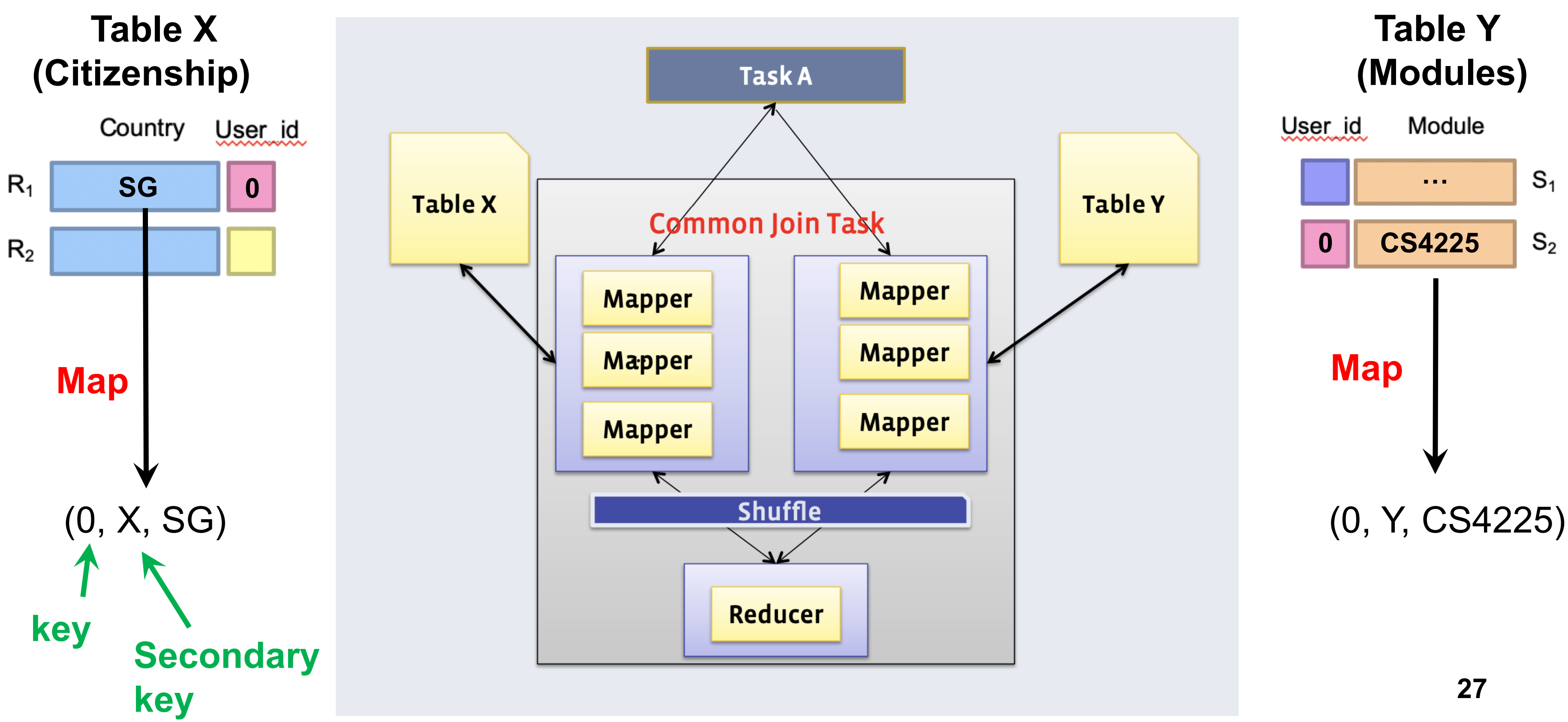
# Method 1: Broadcast (or 'Map') Join

- Requires one of the tables to fit in memory
  - All mappers store a copy of the small table (for efficiency: we convert it to a **hash table**, with keys as the keys we want to join by)
  - They iterate over the big table, and join the records with the small table



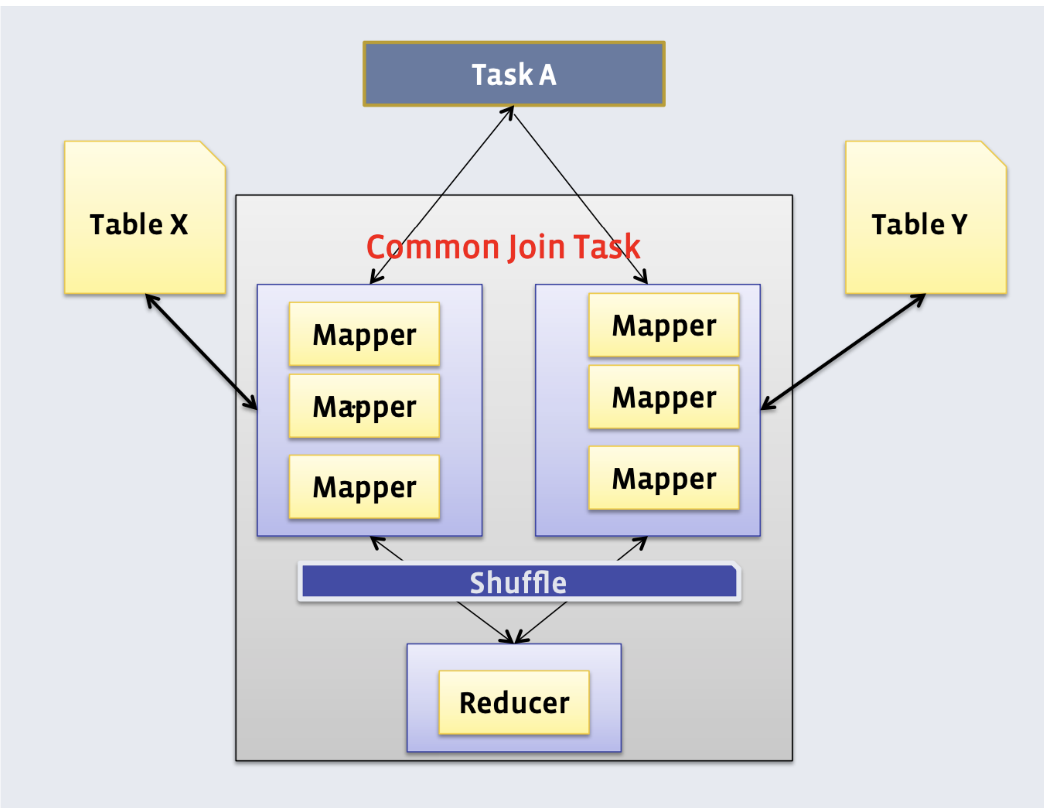
# Method 2: Reduce-side (or 'Common') Join

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

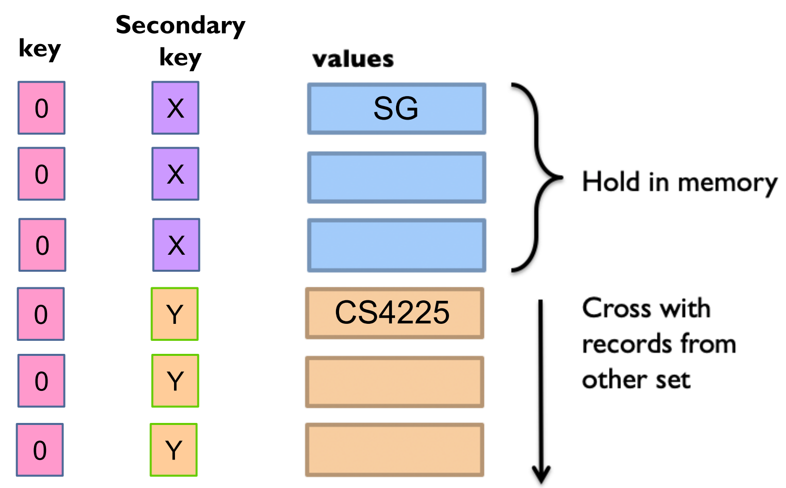


# Method 2: Reduce-side (or 'Common') Join

- 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




**In reduce function for key 0:**



# Resources

- Hadoop: The Definitive Guide (by Tom White)
- Hadoop Wiki
  - https://hadoop.apache.org/docs/current/

# Take-away

- Big data systems may not automatically bring performance benefits
- Performance analysis and algorithm design are still needed to efficiently and effectively develop various applications.

# Take-away in the AI Era

- 🧠 What AI cannot decide for you (yet)
  - Whether a problem should be solved by **SQL or MapReduce**
  - Whether data should be **moved to computation or computation to data**
  - Whether performance is limited by **network, disk, or memory**
- **Your design matters (AI can assist)**
  - Choosing between abstractions (SQL vs MapReduce)
  - Reasoning about data locality
  - Selecting join strategies under constraints

# 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.

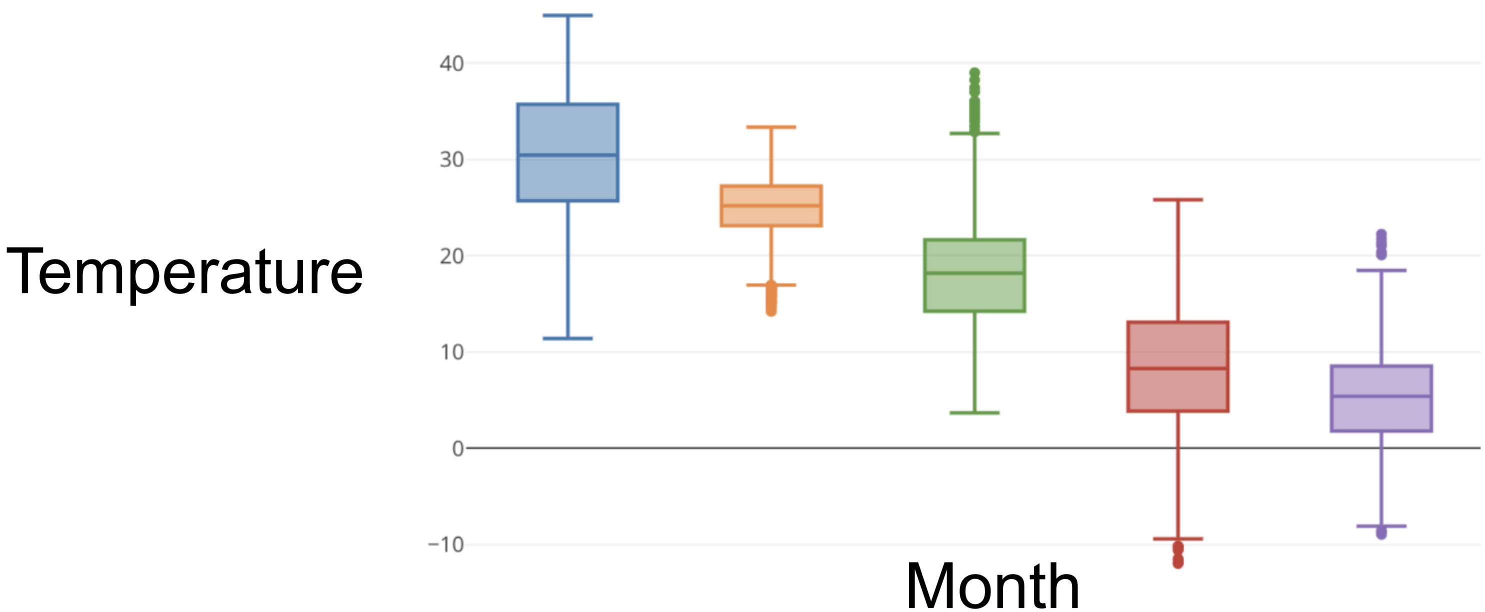
# Acknowledgement

- Slides adopted/revised from
  - Jimmy Lin, http://lintool.github.io/UMD-courses/bigdata-2015-Spring/
  - Bryan Hooi
- Some slides are also adopted/revised from
  - Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. 2014. Mining of Massive Datasets (2nd ed.). Cambridge University Press. http://www.mmds.org/

# Supplementary Slides

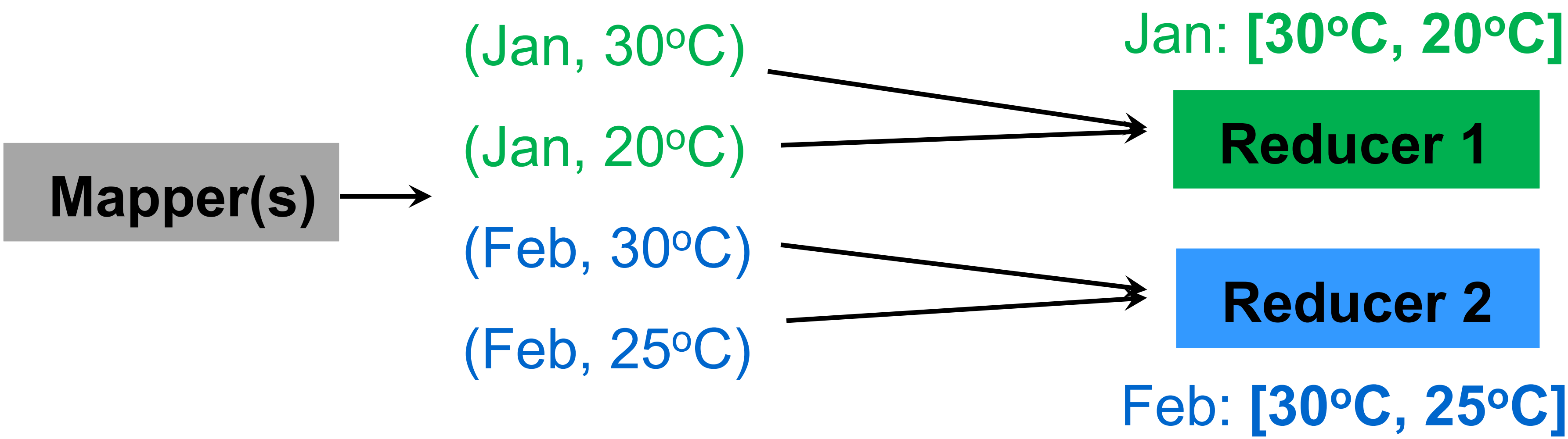
# Secondary Sort: Overview

- **Example**: Suppose our map function emits (month, temperature) tuples, and we want to compute some statistics for each month, like the median, 25% quantile, 75% quantile, etc.



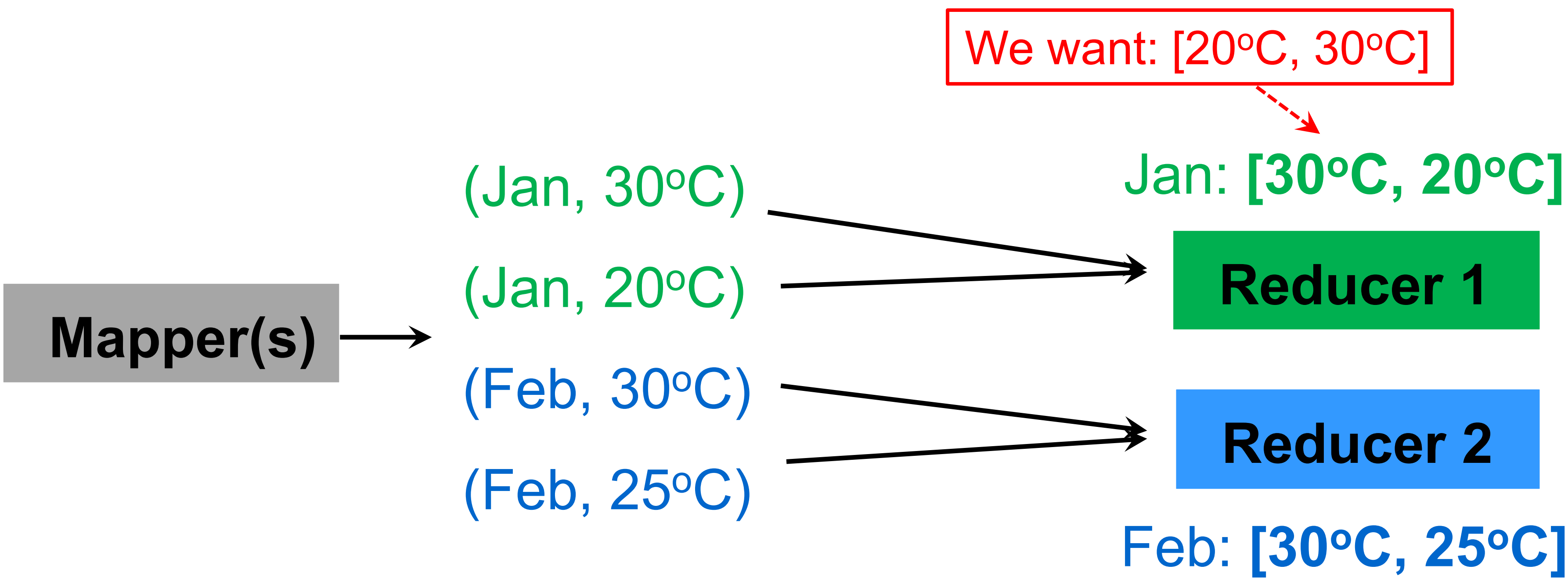
# Secondary Sort: Overview

- **Example**: Suppose our map function emits (month, temperature) tuples, and we want to compute some statistics for each month, like the median, 25% quantile, 75% quantile, etc.
- **Recall that**: At each reduce task, the **keys arrive sorted**, but the **values arrive unsorted** (i.e., in an arbitrary order)



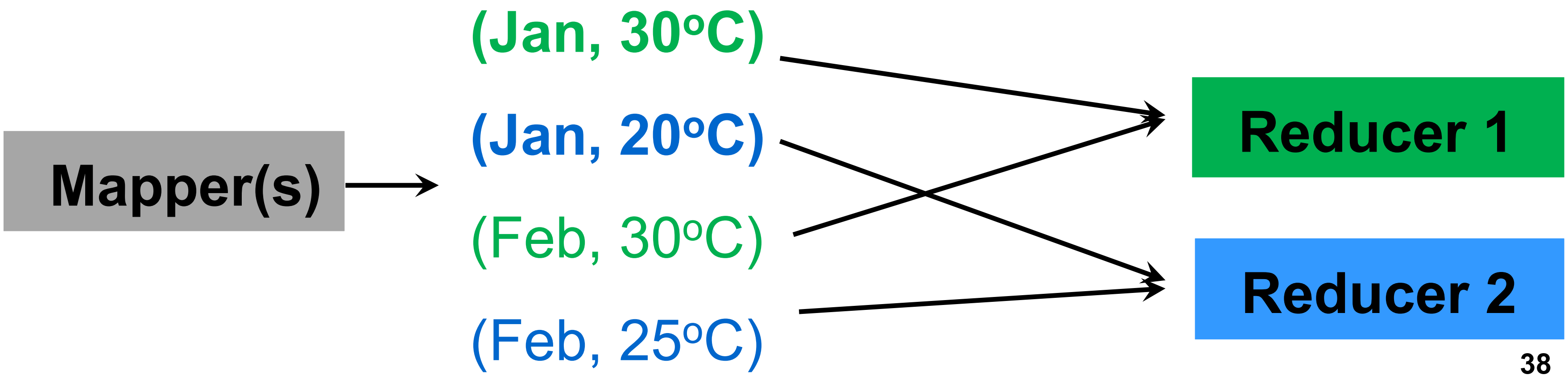
# Secondary Sort: Overview

- **Goal of Secondary Sort**: It is a “trick” allowing the reducer to **receive values in sorted order**
  - Why could this be useful? Having the values arrive in sorted order could make it more convenient to compute certain statistics, particularly for data containing timestamps, and also for medians and quantiles, as in this case (especially for very large datasets)



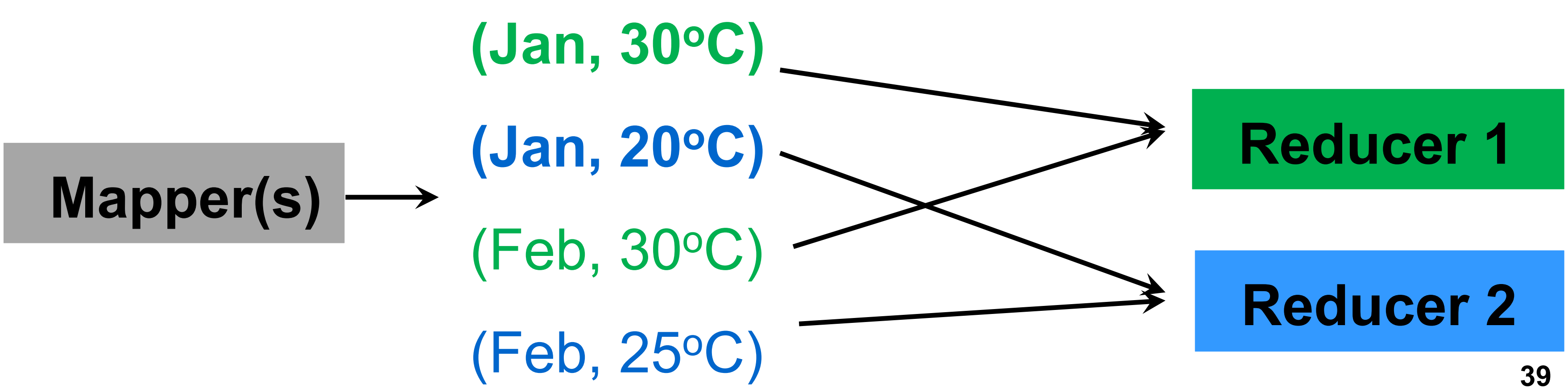
# Secondary Sort: Approach

- Combine (month, temperature) together into a **composite key**.
  - Further, define a **custom comparator** to compare by month first, then by temperature



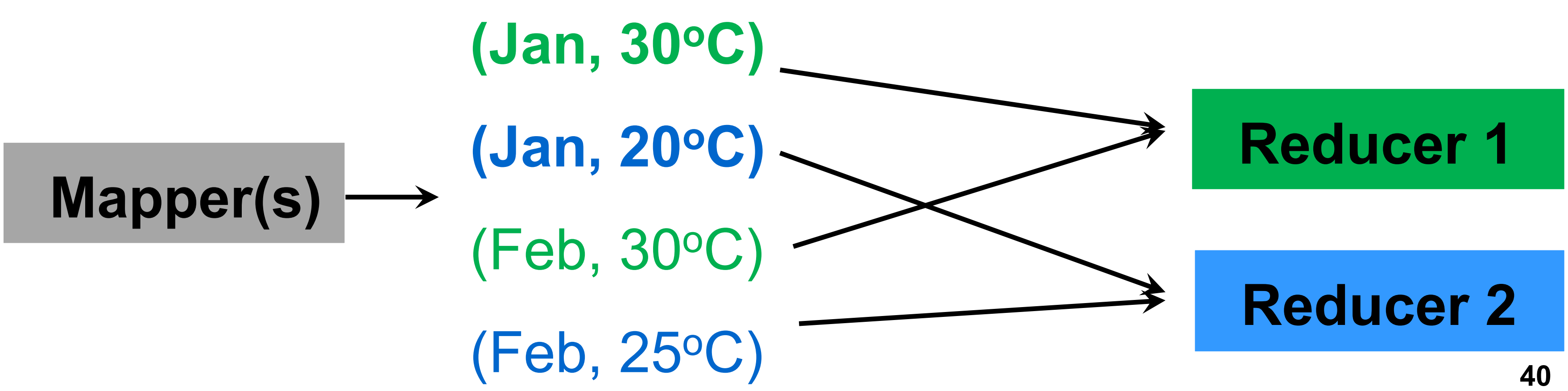
# Secondary Sort: Approach

- Combine (month, temperature) together into a **composite key**.
  - Further, define a **custom comparator** to compare by month first, then by temperature (i.e., if month is tied, compare by temperature)
- Let's consider: what should the partitioner function be?
  - First let's assume that we simply used the default partitioner, which partitions based by hashing values of the composite key.



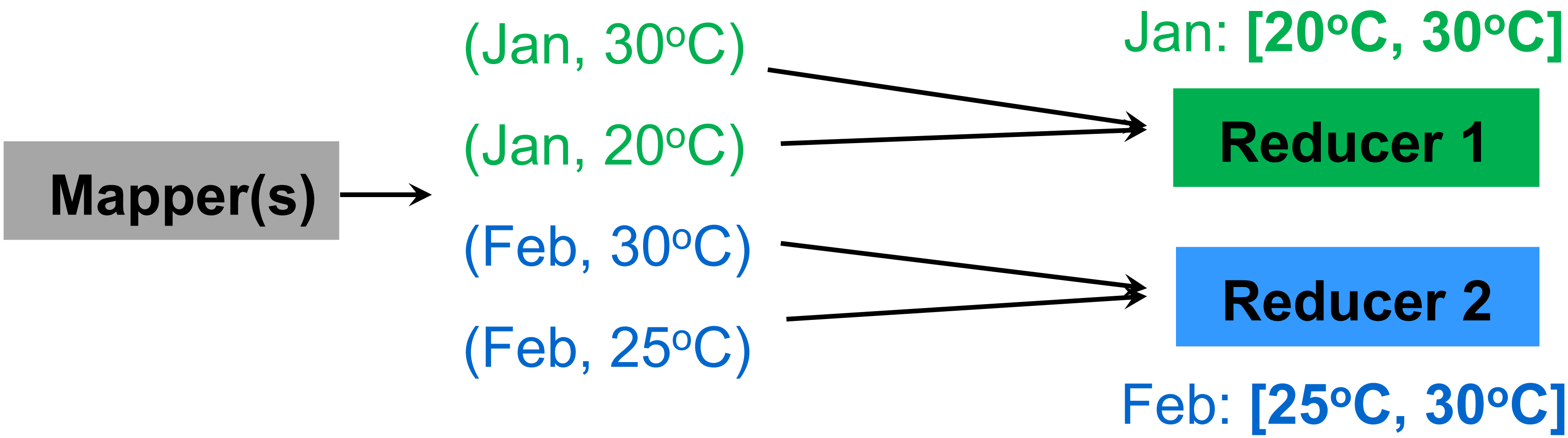
# Secondary Sort: Example

- The 4 tuples below have 4 different "values" of the composite key.
  - (Jan, 30ºC) and (Jan, 20ºC) have different values of the composite key, so they can be partitioned into different reduce tasks
- Unfortunately, this means we now can't compute the desired statistics for each month (e.g., as the January tuples are split into different reducers)



# Secondary Sort: Example

- To fix this problem we need to implement a **custom partitioner**, to partition by month only.
- Now we see that 1) data for the same month goes to the same reducer, and 2) at each reducer, data arrives sorted by temperature, since the MapReduce framework always sorts the data by key before giving it to each reducer.



# Code Example: Composite Key

```
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
...
public class DateTemperaturePair
   implements Writable, WritableComparable<DateTemperaturePair> {

    private Text yearMonth = new Text();                 // natural key
    private Text day = new Text();
    private IntWritable temperature = new IntWritable(); // secondary key

    ...

    @Override
    /**
     * This comparator controls the sort order of the keys.
     */
    public int compareTo(DateTemperaturePair pair) {
        int compareValue = this.yearMonth.compareTo(pair.getYearMonth());
        if (compareValue == 0) {
            compareValue = temperature.compareTo(pair.getTemperature());
        }
        //return compareValue;    // sort ascending
        return -1*compareValue;   // sort descending
    }
    ...
}
```

Define our composite key, which is a **pair** consisting of a primary key (**`yearMonth`**) and secondary key (**`temperature`**)

Define comparator: compare by **`yearMonth`** first; if tie, compare by **`temperature`**

# Code Example: Custom Partitioner

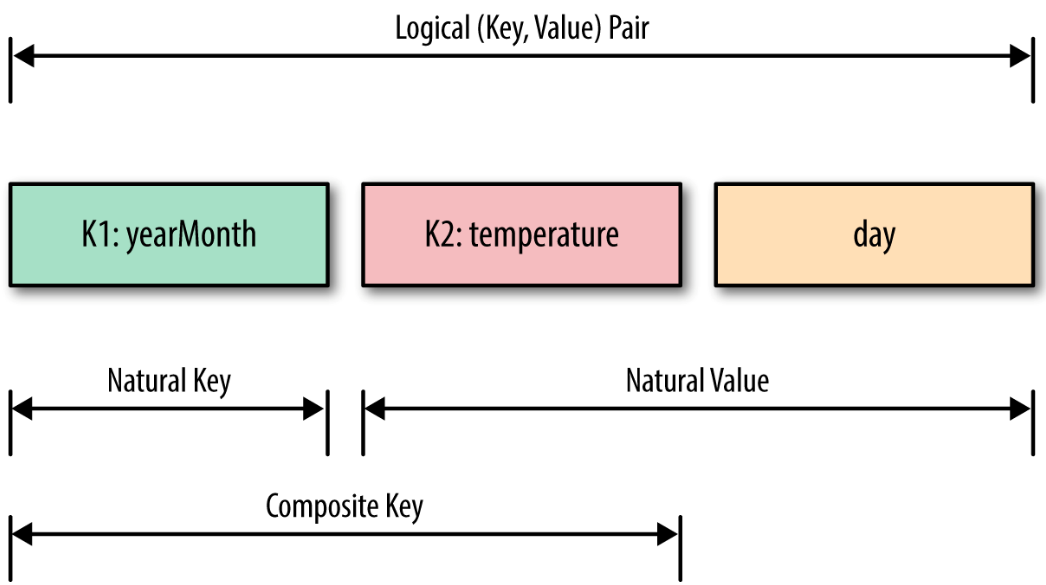
```
public class DateTemperaturePartitioner
    extends Partitioner<DateTemperaturePair, Text> {

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

Define custom partitioner

Partition by **`yearMonth`**
only (not **`temperature`**)

# Secondary Sort: Summary

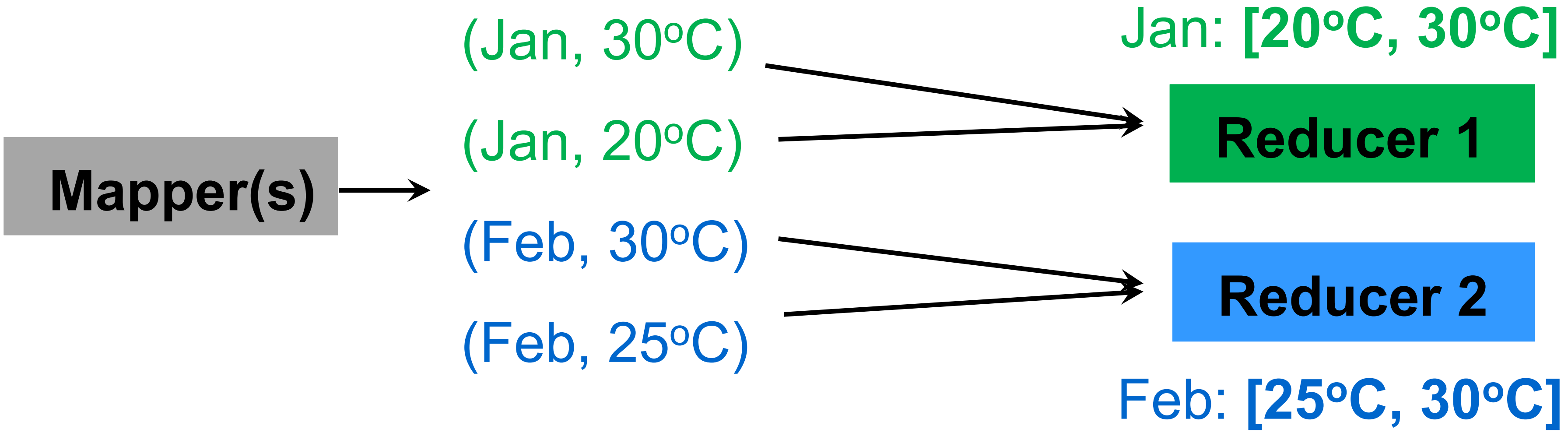
- **General Approach**: define a new 'composite key' as (K1, K2), where K1 is the original key ("Natural Key") and K2 is the variable we want to use to sort
  - Custom Comparator: compare by K1 first, then by K2
  - Partitioner: now needs to be customized, to partition by K1 only, not (K1, K2)



# Secondary Sort: Q&A

Q: Could we just sort the data ourselves in the reducer?

A: Yes, but it may be expensive (in memory & time) if there are a lot of tuples for one key. Secondary sort is effectively "borrowing" the inbuilt distributed sorting process of Hadoop to sort the tuples for us.



# Preserving State in Map / Reduce Tasks

- Recall that a map task calls the map function multiple times
- In fact, we can store state variables, that can be shared across multiple calls to the map function within a map task: