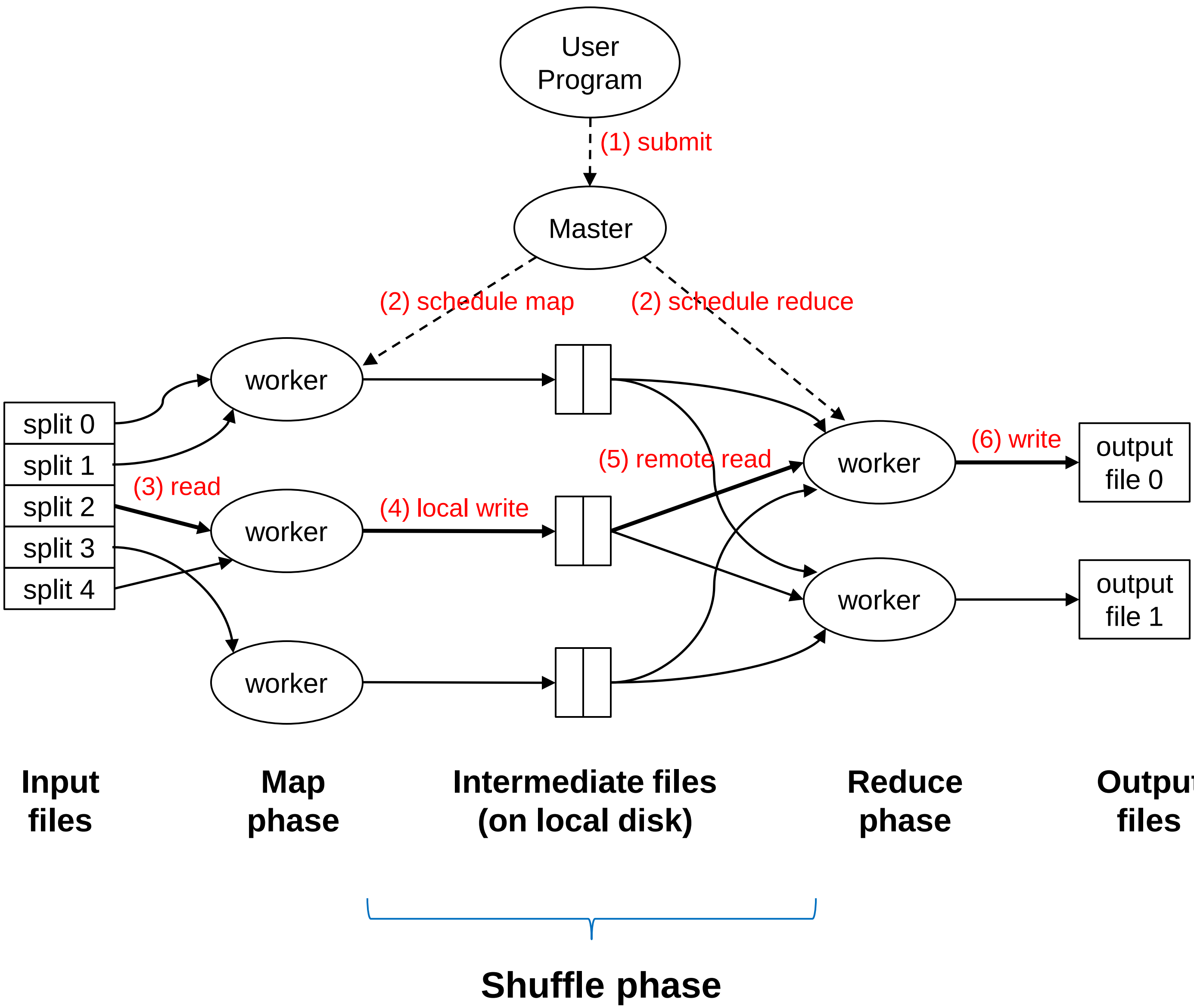
# Announcement

- Feel free to post questions/discussions within each lab group.
- Coding assignment consultation sessions today:
  - 3:45-4:45pm, LT 15 (TA: Jiqing and Xihao)
  - 8:15-9:15pm, I3-AUD (TA: Zhiheng and Edward)

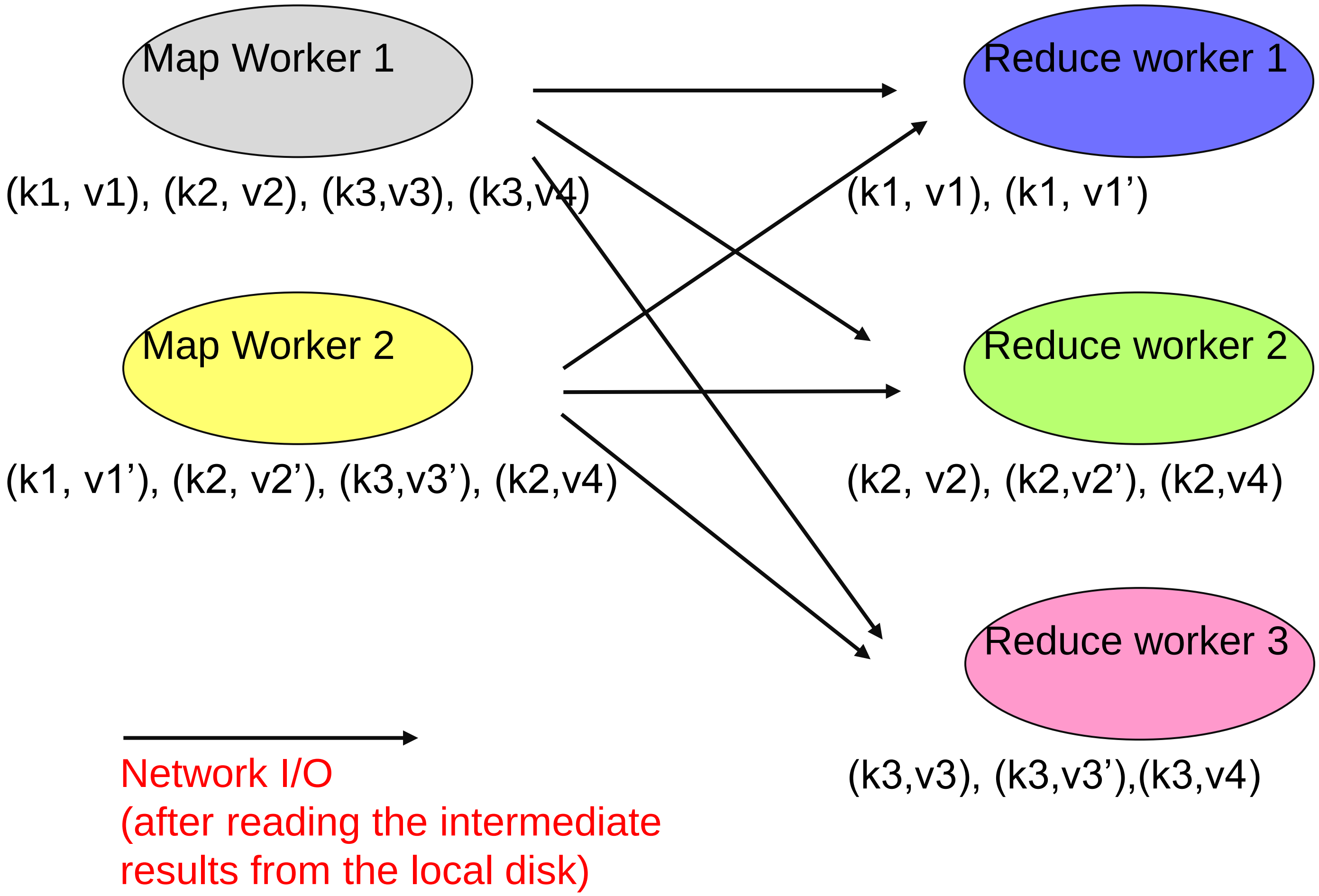
# Recap: MapReduce Implementation




Adapted from (Dean and Ghemawat, OSDI 2004)

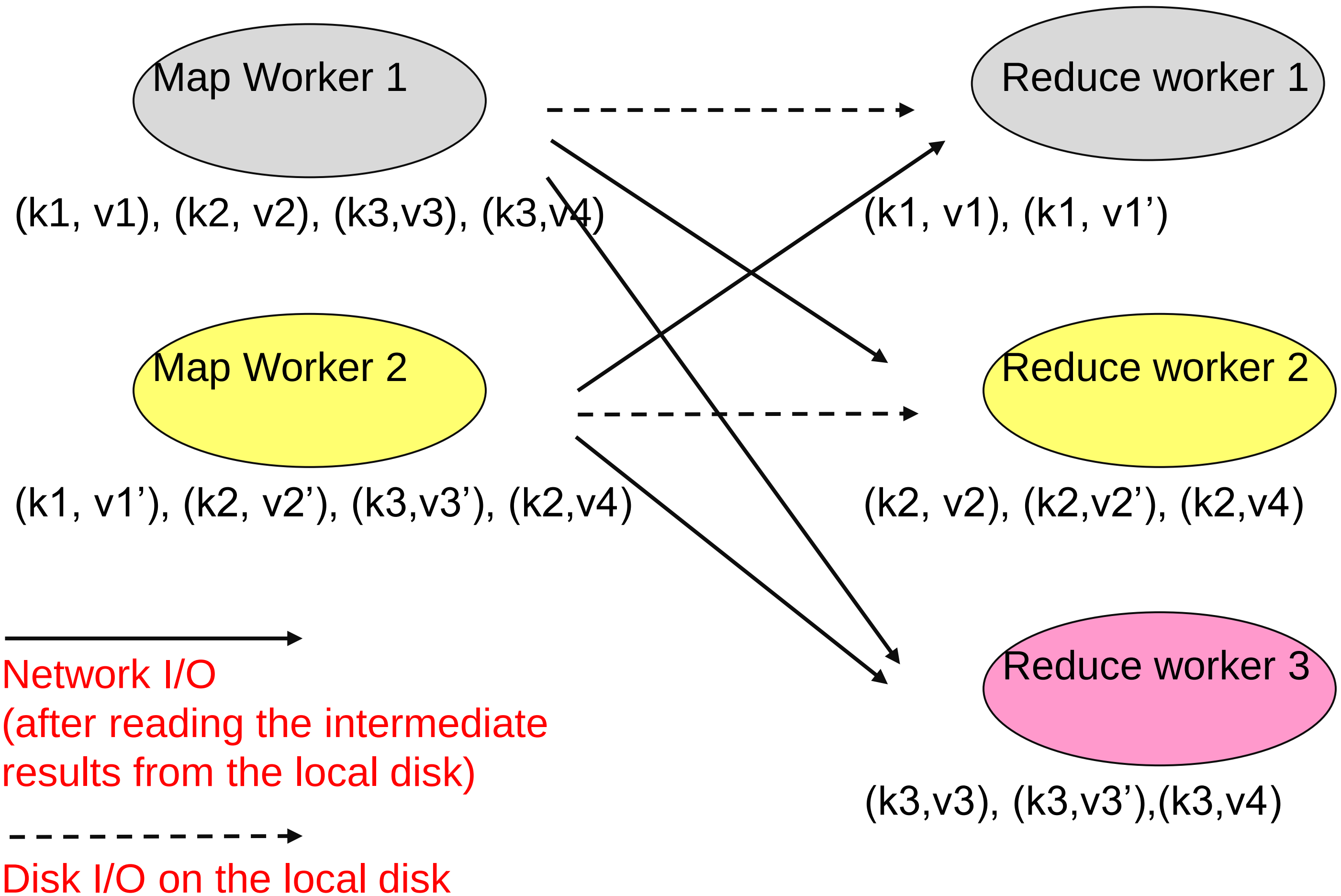
# Recap: The amount of network I/O in shuffle

- Case 1: Map Workers are different to Reduce Workers



# Recap: The amount of network I/O in shuffle

- Case 2: Two map Workers are reused in reduce phase.



# Recap: Performance Guidelines for Basic Algorithmic Design

- **Linear scalability:** more nodes can do more work in the same time
  - Linear on data size
  - Linear on computer resources
- **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
- **Memory working set** of each task/worker
  - Large memory working set -> high memory requirements / probability of out-of-memory errors.
- **Load balance among tasks**
  - Load imbalance -> long execution time / large memory working set.
- Guidelines are applicable to Hadoop, Spark, …

# Recap: A Step-by-Step Performance Analysis Guide

- Scalability analysis
  - **Max** number of Map tasks: Will the number of mapper tasks increase linearly as the input size increases?
  - **Max** number of Reduce tasks: Will the number of reducer tasks increase linearly as the input size increases?
- I/O analysis: input + intermediate results + output
  - The amount of disk I/O from each Map/Reduce task
  - The amount of network I/O from each Map/Reduce task
  - The amount of network I/O in shuffle (= amount of intermediate results from map tasks)
- Memory working set: intermediate results
  - The amount of memory consumption from each map/reduce task
- Load balance: worst case analysis
  - The worst execution time for all map/reduce tasks
  - The largest memory working set for all map/reduce tasks

# Recap: Scalability Analysis

- Assume that one worker can run one Map or Reduce task
- Linear scalability:
  - Given W workers, we can run W tasks at the same time.
  - The key question will be: how many tasks does the job have?
- We calculate:
  - **Max** number of Map tasks
  - **Max** number of Reduce tasks
- We analyze:
  - Will the number of mapper tasks increase linearly as the input size increases?
  - Will the number of reducer tasks increase linearly as the input size increases?

# Recap: Word Count V0: Scalability Analysis

```
class Mapper {
  def map(key: Long, value: Text) = {
    for (word <- tokenize(value)) {
      emit(word, 1)
    }
}

class Reducer {
  def reduce(key: Text, values: Iterable[Int]) = {
    for (value <- values) {
      sum += value
    }
    emit(key, sum)
  }
}
```

**<u>Scalability analysis for Map:</u>**

*** Max number of map tasks**

**= input size / chunk size**

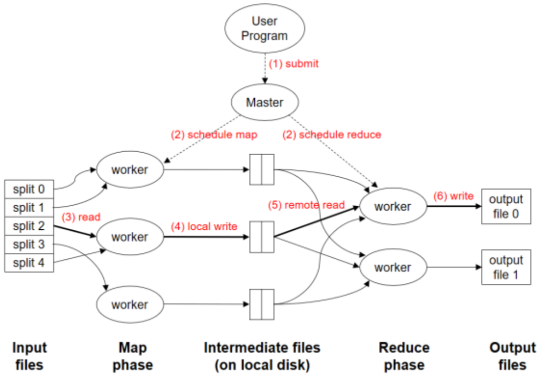
**<u>Scalability analysis for Reduce:</u>**

*** Max number of reduce tasks**

**= number of distinct keys**

# Recap: I/O Analysis

- For a Mapreduce job, we have the following I/O components:
  - Reading input from HDFS: mainly disk I/O
  - Shuffle and sort: Disk and network I/O
  - Output: Disk and network I/O




- We calculate:
  - The amount of disk I/O from each Map/Reduce task
  - The amount of network I/O from each Map/Reduce task
  - The amount of network I/O in shuffle (= amount of intermediate results from map tasks)

# Recap: Word Count V0: I/O Analysis

```
class Mapper {
  def map(key: Long, value: Text) = {
    for (word <- tokenize(value)) {
      emit(word, 1)
  }
}
```

**<u>I/O analysis for map task:</u>**

- **Input Disk I/O= 128MB**
- **Intermediate results = very small (i.e., can be ignored)**
- **Output Disk I/O= all <word, 1> pairs, #pairs= #words in the chunk**
- **Network I/O= very small**

**<u>I/O analysis for shuffling:</u>**

**Network I/O = all <word, 1> pairs, #pairs= #words in the chunk**

# Recap: Word Count V0: I/O Analysis

```
class Mapper {
  def map(key: Long, value: Text) = {
    for (word <- tokenize(value)) {
      emit(word, 1)
  }
}

class Reducer {
  def reduce(key: Text, values: Iterable[Int]) = {
    for (value <- values) {
      sum += value
    }
    emit(key, sum)
  }
}
```

**<u>I/O analysis for reduce task:</u>**

- **Input Disk I/O= very small //already counted in shuffling**
- **Intermediate results = very small**
- **Output Disk I/O= very small**
- **Network I/O= very small**

# Recap: Memory Consumption Analysis

- For Map/Reduce function, look for the memory allocation:
  - Variables
  - Intermediate data structures
- We calculate:
  - The total amount of memory consumption by all those variables/data structures.

# Recap: Word Count V0: Memory Consumption

```
class Mapper {
  def map(key: Long, value: Text) = {
    for (word <- tokenize(value)) {
      emit(word, 1)
    }
}

class Reducer {
  def reduce(key: Text, values: Iterable[Int]) = {
    for (value <- values) {
      sum += value
    }
    emit(key, sum)
  }
}
```

**Memory analysis for Map:**

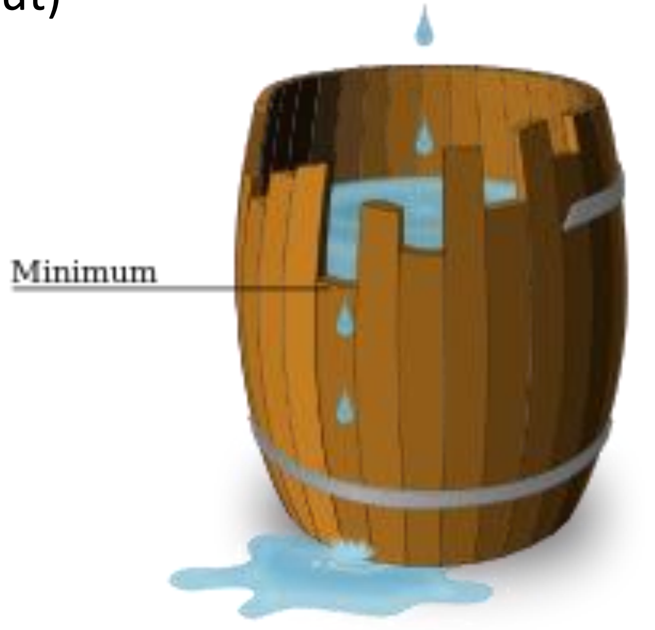
*** Memory working set**
**= very small**

**Memory analysis for Reduce:**

*** Memory working set**
**= very small**

# Recap: Load Balance

- Avoid unevenly overloading some compute nodes while other compute nodes are left idle.
- Cannikin Law (“Buckets effect”)
- Worst case analysis for all map/reduce tasks
  - The worst execution time (sensitive to input)
  - The largest memory consumption



# Recap: Word Count V0: Worst Case Analysis

```
class Mapper {
  def map(key: Long, value: Text) = {
    for (word <- tokenize(value)) {
      emit(word, 1)
  }
}

class Reducer {
  def reduce(key: Text, values: Iterable[Int]) = {
    for (value <- values) {
      sum += value
    }
    emit(key, sum)
  }
}
```

**Worst case analysis for Map:**

*** Memory working set**

**= Very small**

*** Execution time**

**= Parsing the chunk**

**Worst case analysis for Reduce:**

*** Memory working set**

**= Very small**

*** Execution time**

**= Parsing the entire input**

**(all documents have the same word)**

# Recap: Summary of Issues in Word Count Version 0

- Scalability
  - Max number of reducer tasks= number of distinct keys
- I/O
  - Map: Output Disk I/O= all <word, 1> pairs, #pairs= #words in the chunk
  - Shuffle: Network I/O= all <word, 1> pairs, #pairs= #words in the chunk
- Memory working set
  - No issue
- Load balance
  - Reduce: Execution time= Parsing the entire input (all documents have the same word)

**What's the impact of combiners?**

# CS4225/CS5425 Big Data Systems for Data Science

## MapReduce & Relational Databases

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

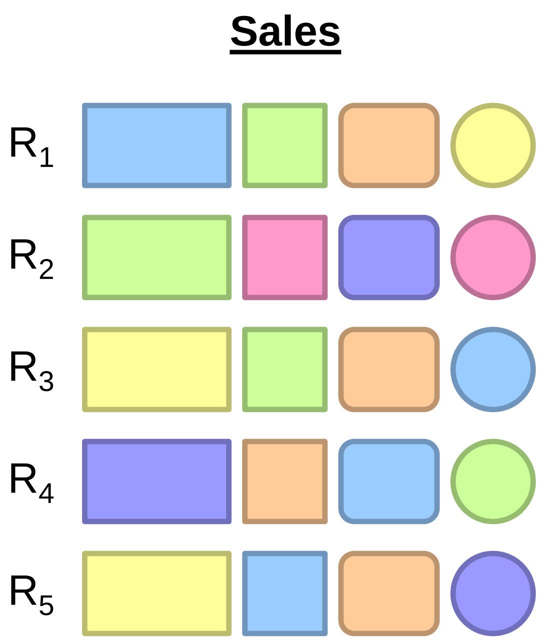
NUS National University of Singapore
School of Computing

# Learning Objectives

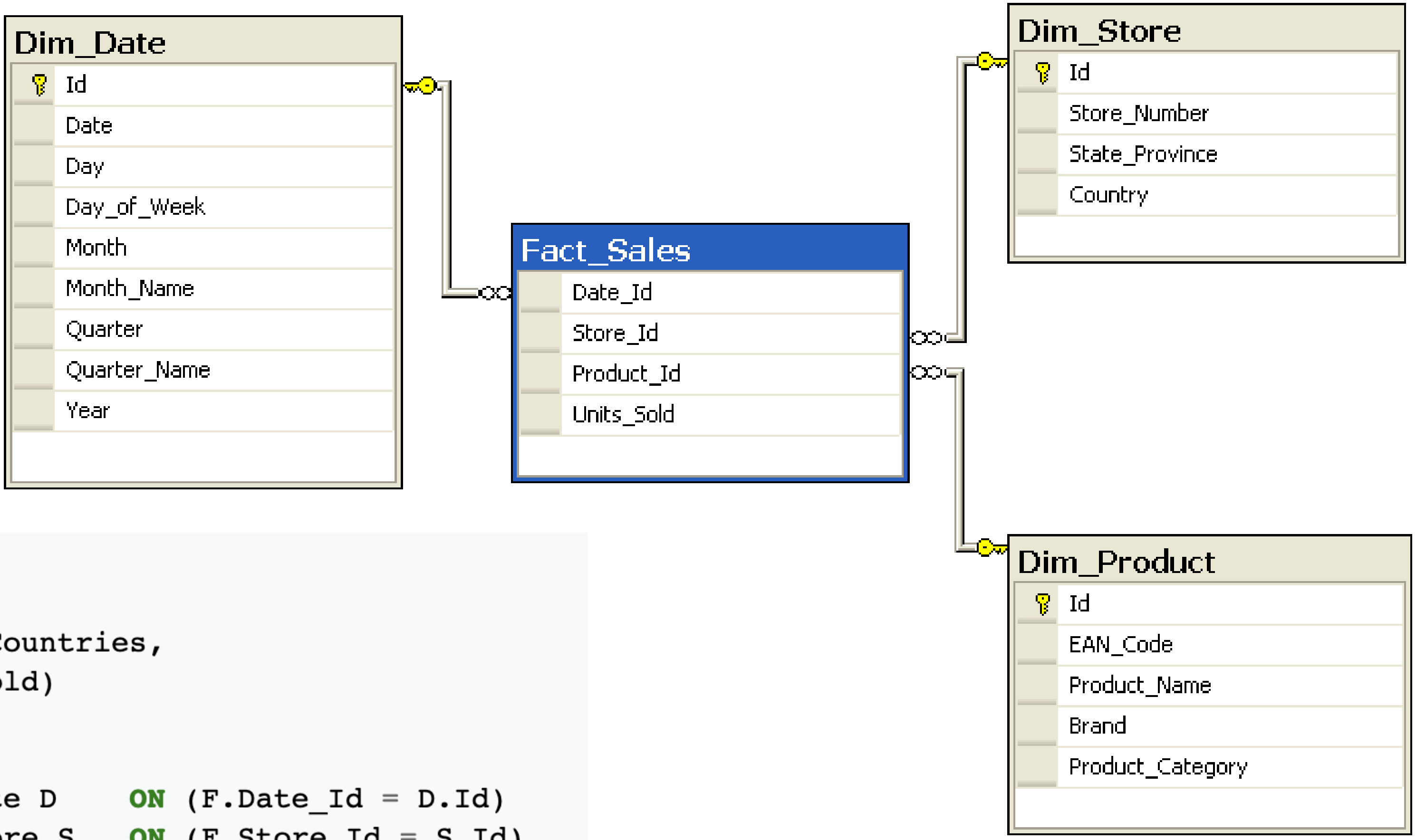
- Learn how to process big relational data with MapReduce.
- Learn algorithmic design of MapReduce.

# 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

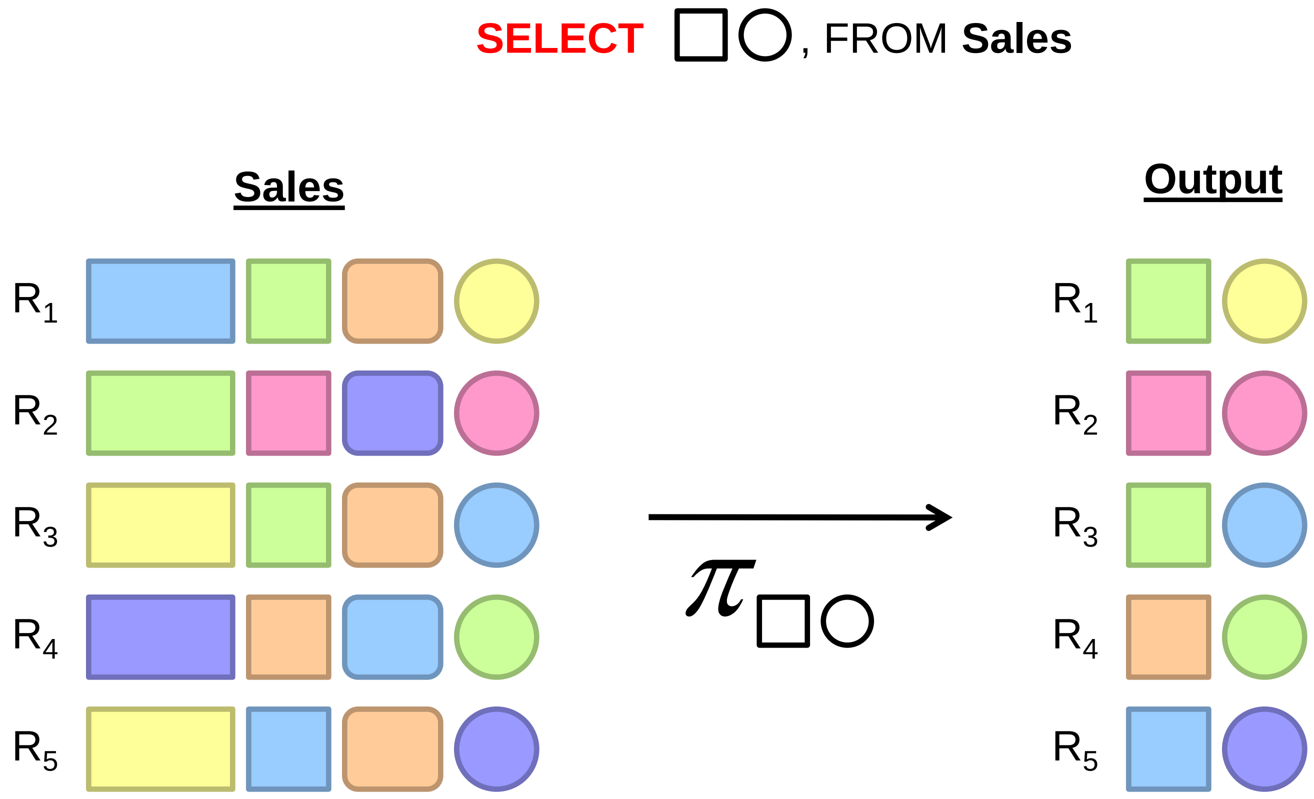



```sql
SELECT
    P.Brand,
    S.Country AS Countries,
    SUM(F.Units_Sold)

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)

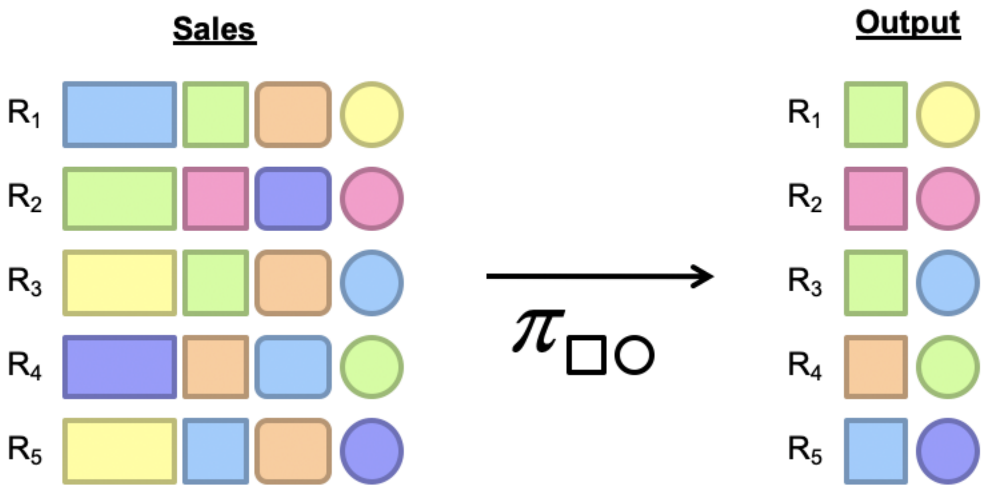
# Projection



# 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)

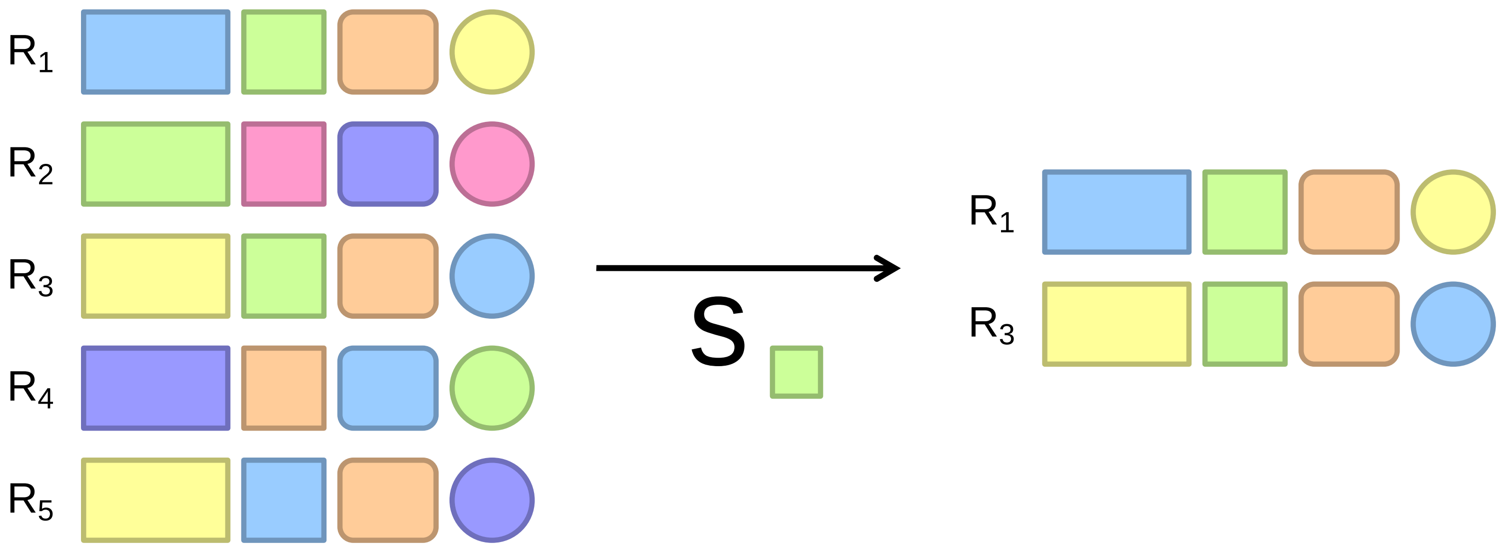
SELECT □○, FROM **Sales**



# Selection

predicate

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

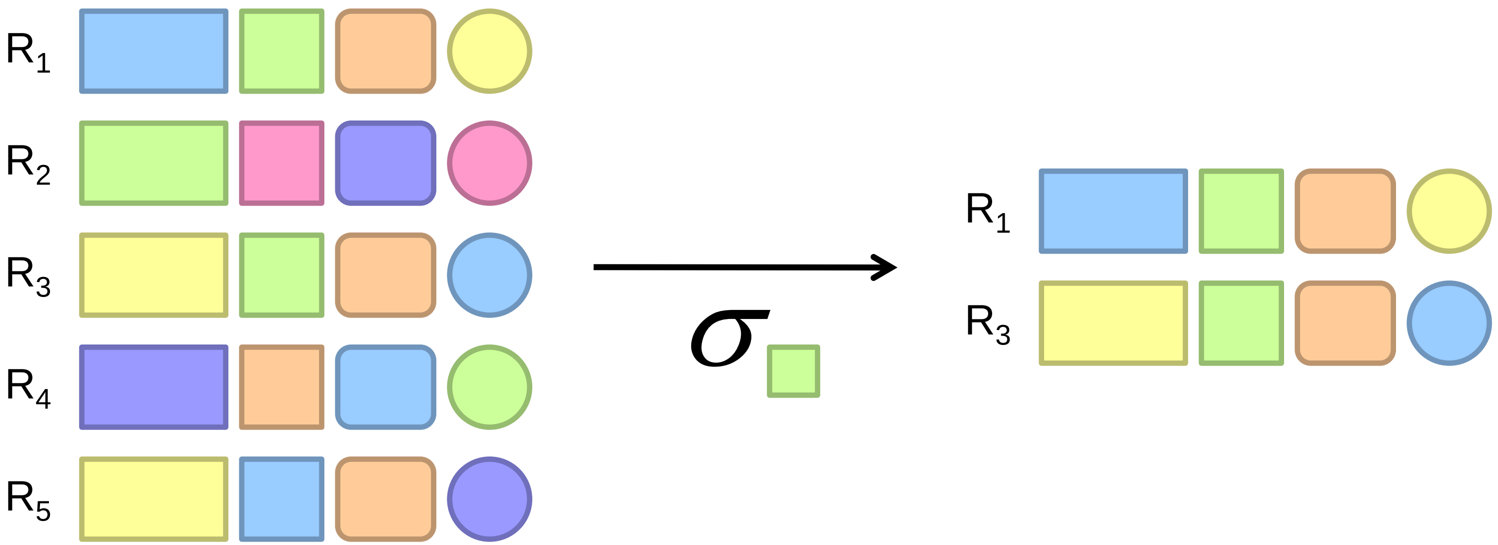


# Selection in MapReduce

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

predicate

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

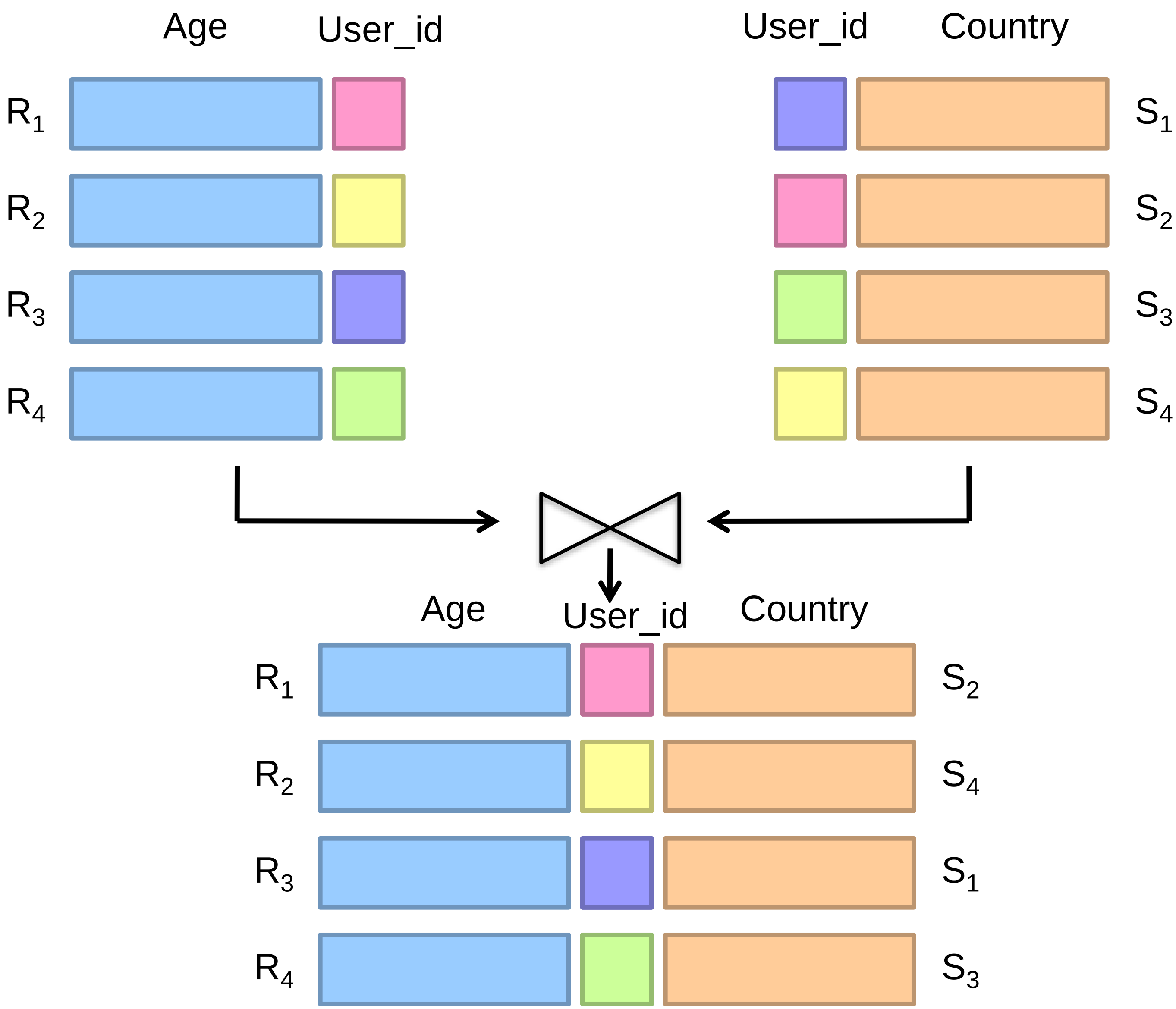


# Group by… Aggregation

- 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')



# An Overview

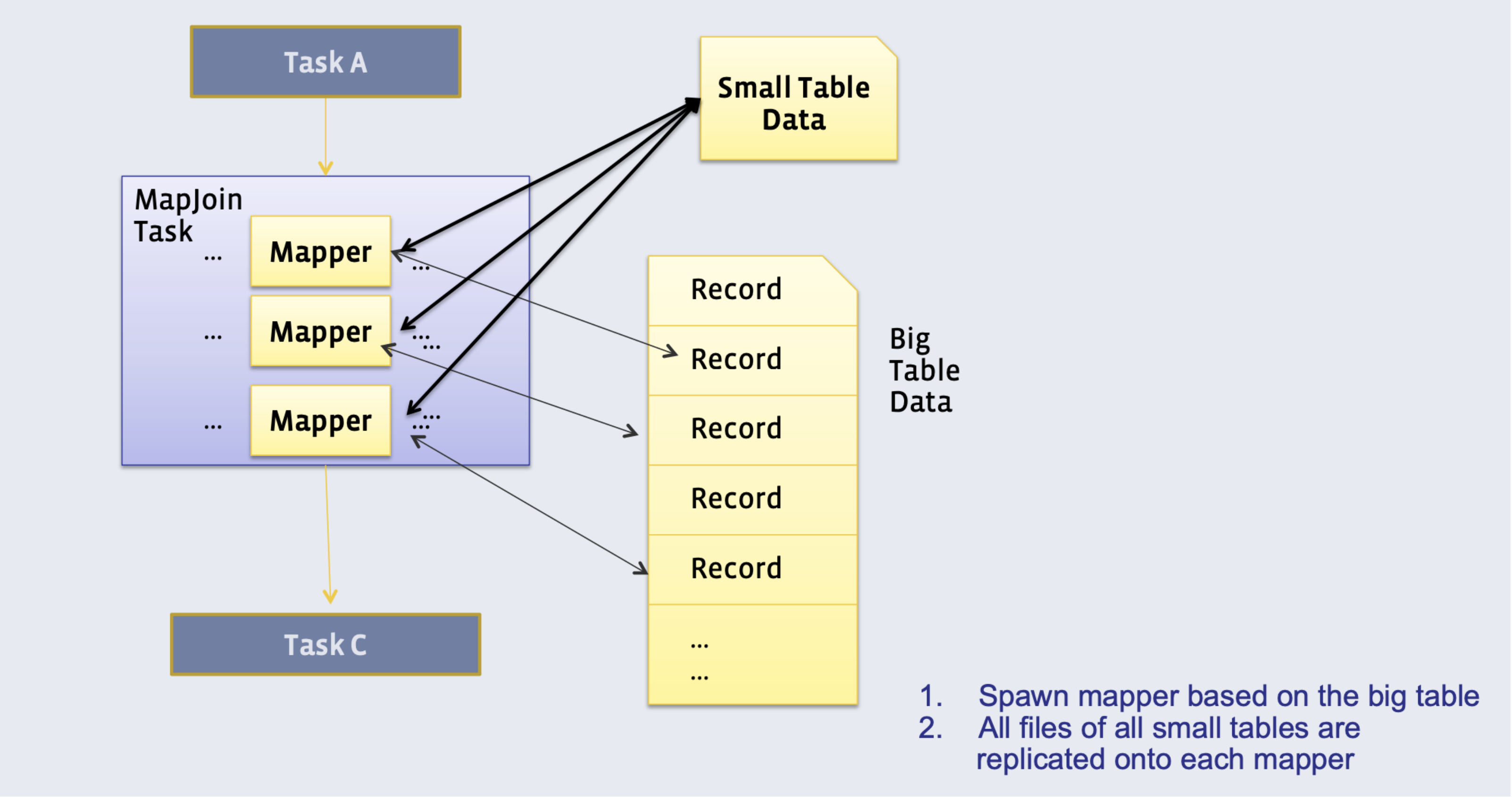
- Join implementations are complex
- “One size does not fit all”
- We mainly talk about two methods.
  - If (one of the input tables is **small**):

    Choose method 1: Broadcast Join
  - Else:

    Choose method 2: Reduce-side Join

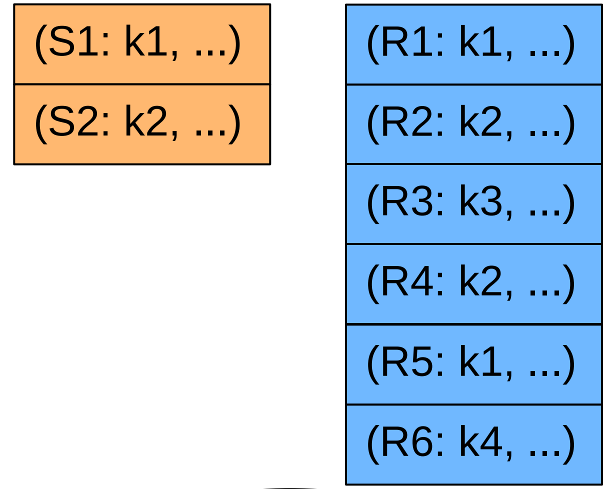
# Method 1: Broadcast Join

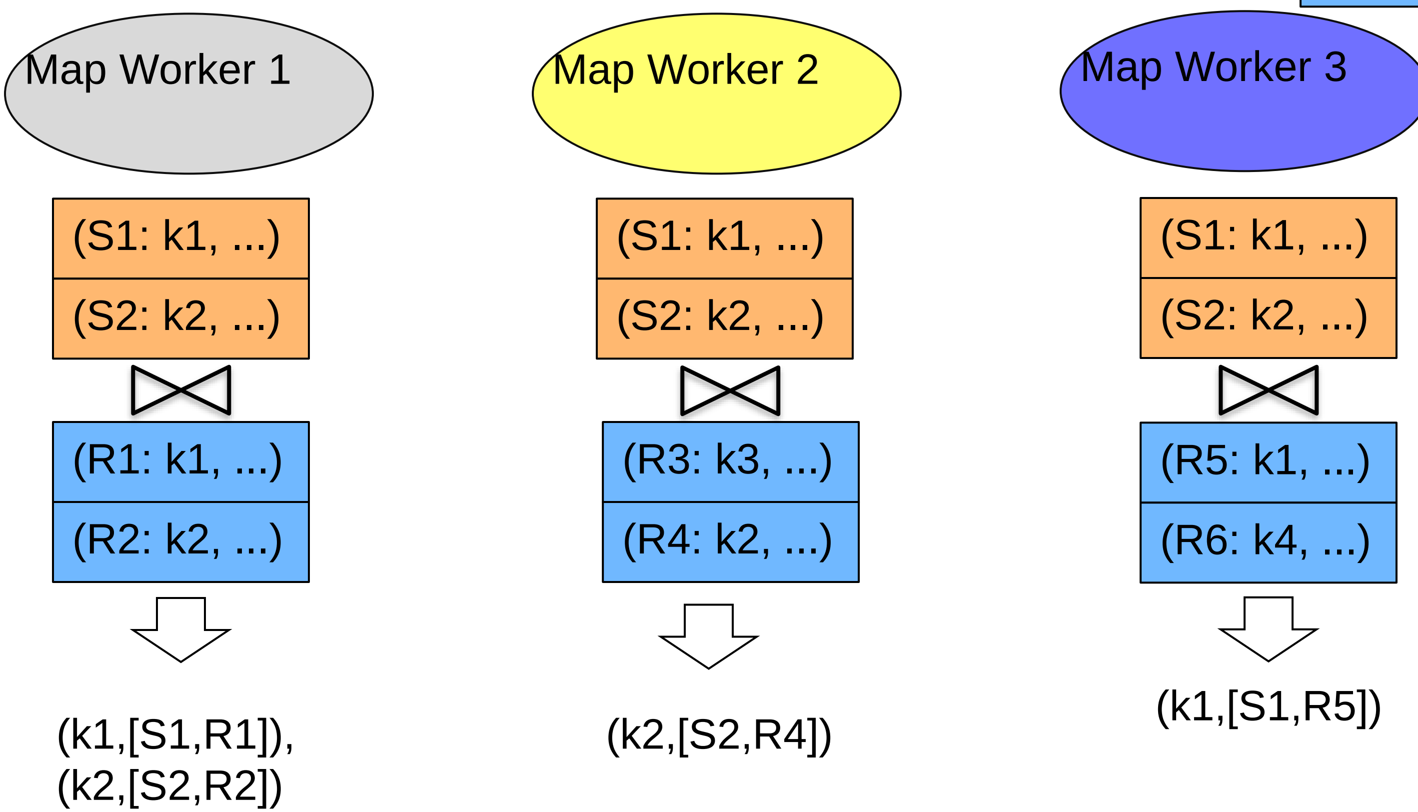
- Requires one of the tables to fit in main memory of individual servers
  - All mappers store a copy of the small table
  - They iterate over the big table, and join the records with the small table



# An Example

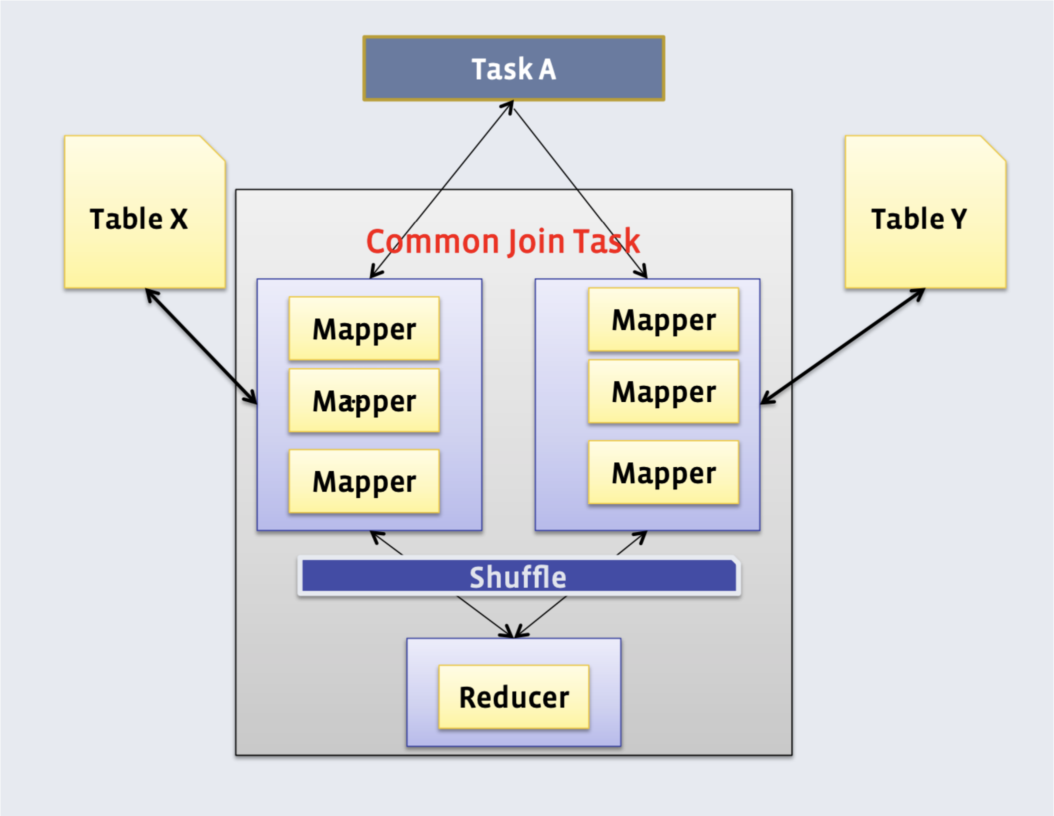
- Small table S with 2 tuples
- Large table R with 6 tuples



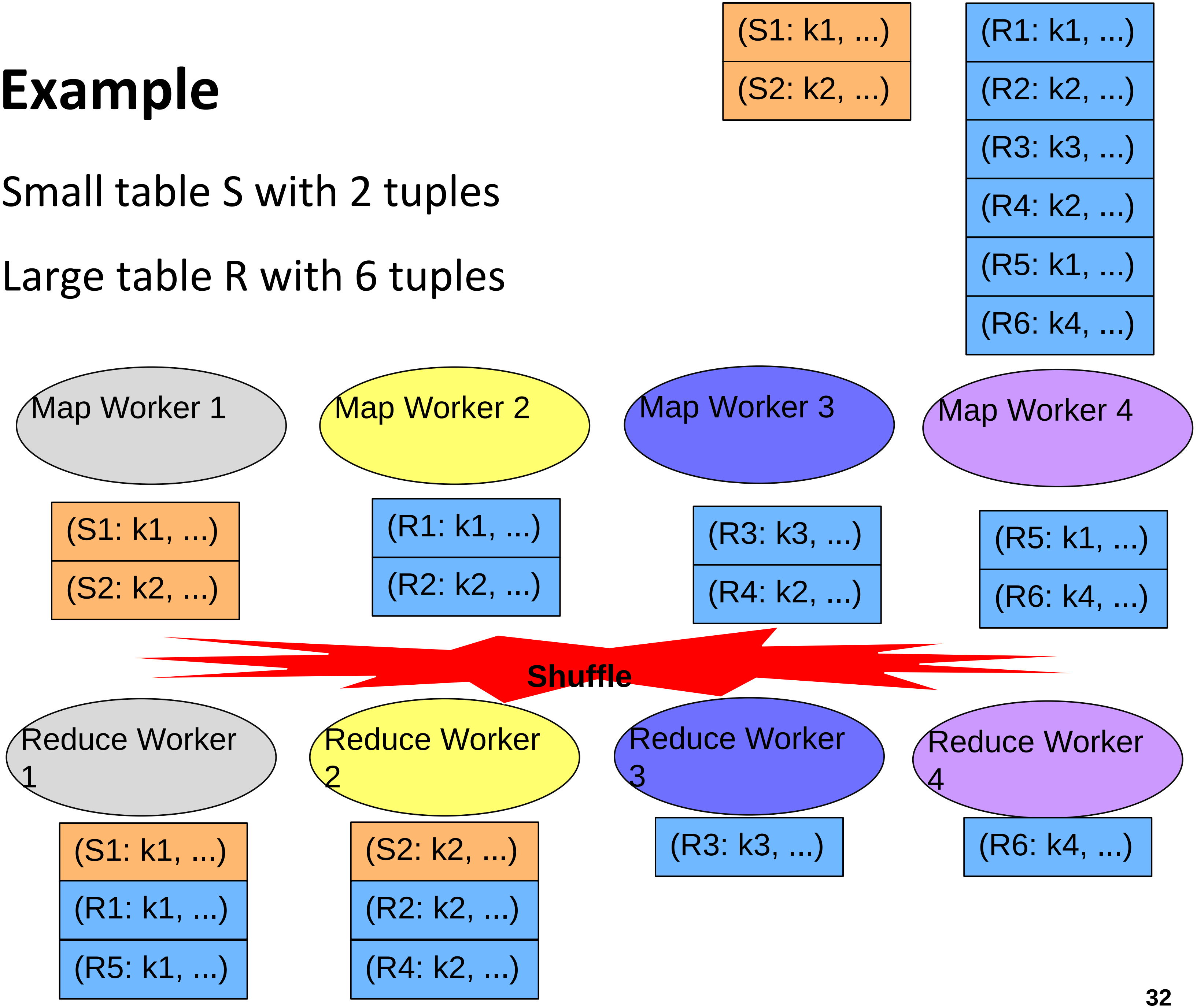
# Method 2: Reduce-side ('Common') Join

- 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



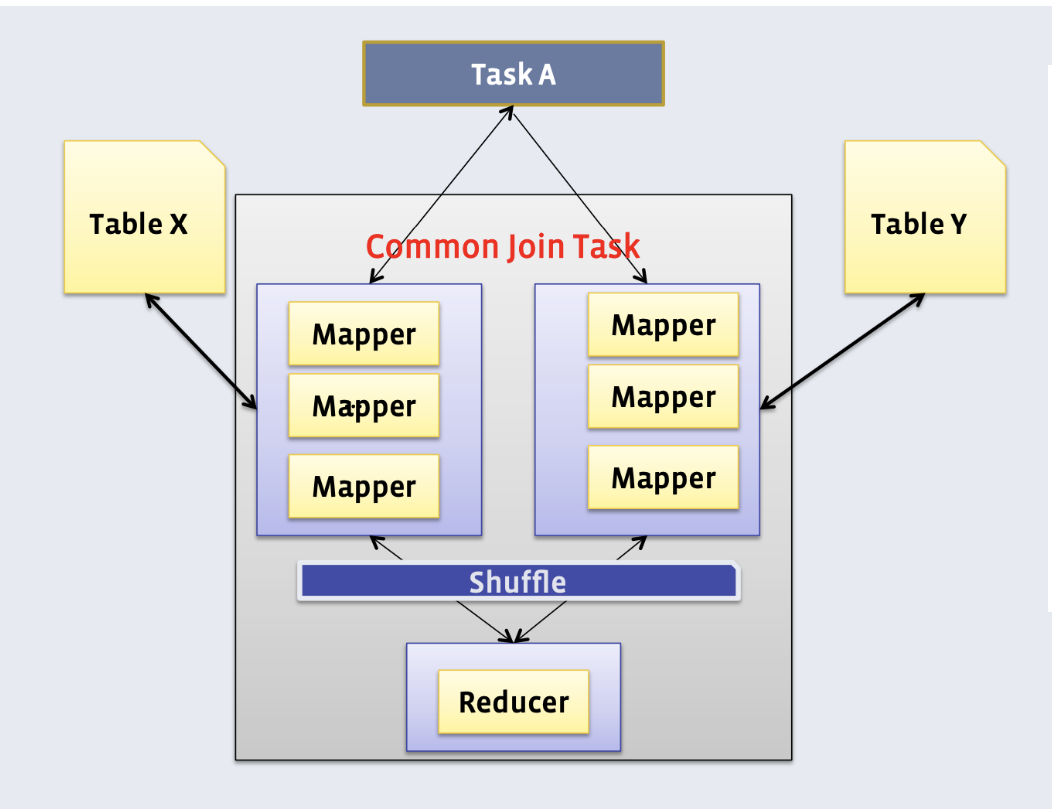
# An Example

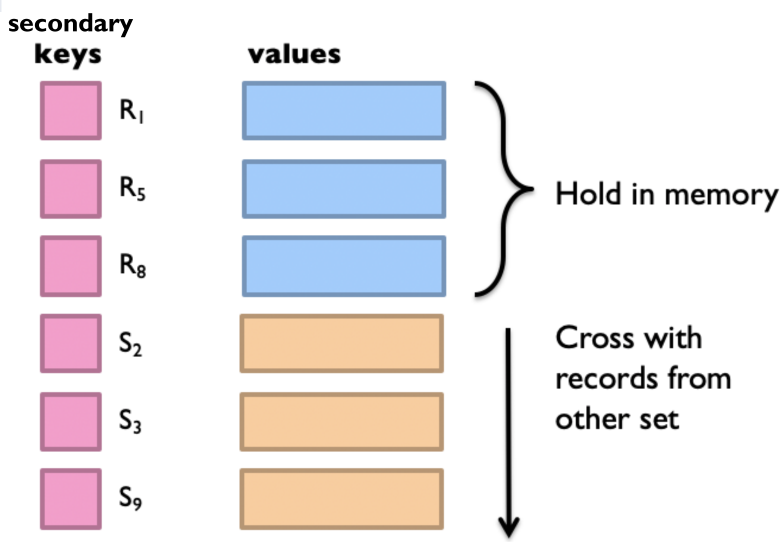
- Small table S with 2 tuples
- Large table R with 6 tuples



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

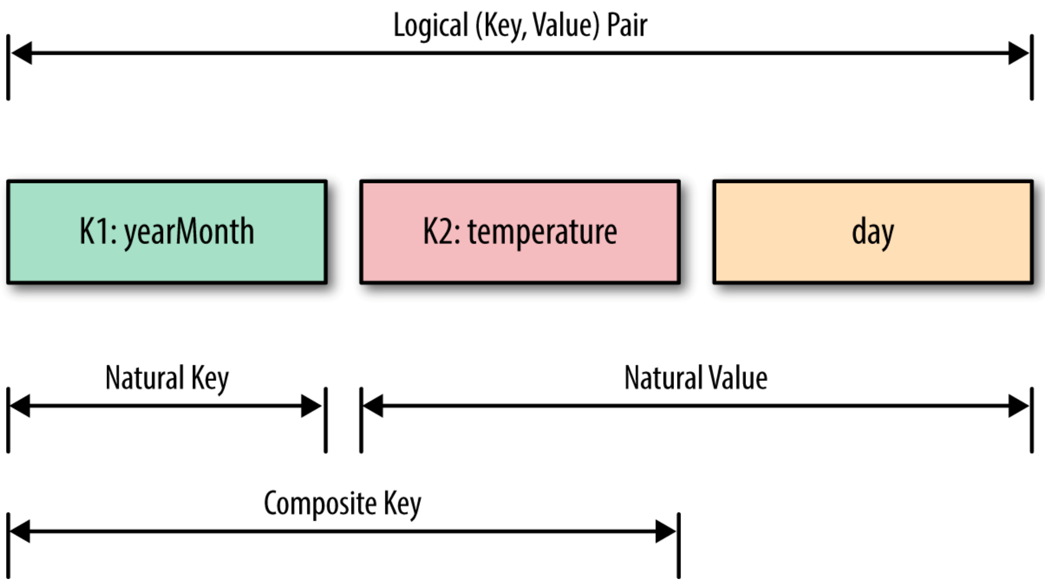
- In reducer: we can use **secondary sort** to ensure that all keys from table X arrive before table Y
  - Then, hold the keys from table X in memory and cross them with records from table Y



# Secondary Sort

- (note: not required knowledge for class)
- **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 'composite 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)



# 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
- Bryan Hooi