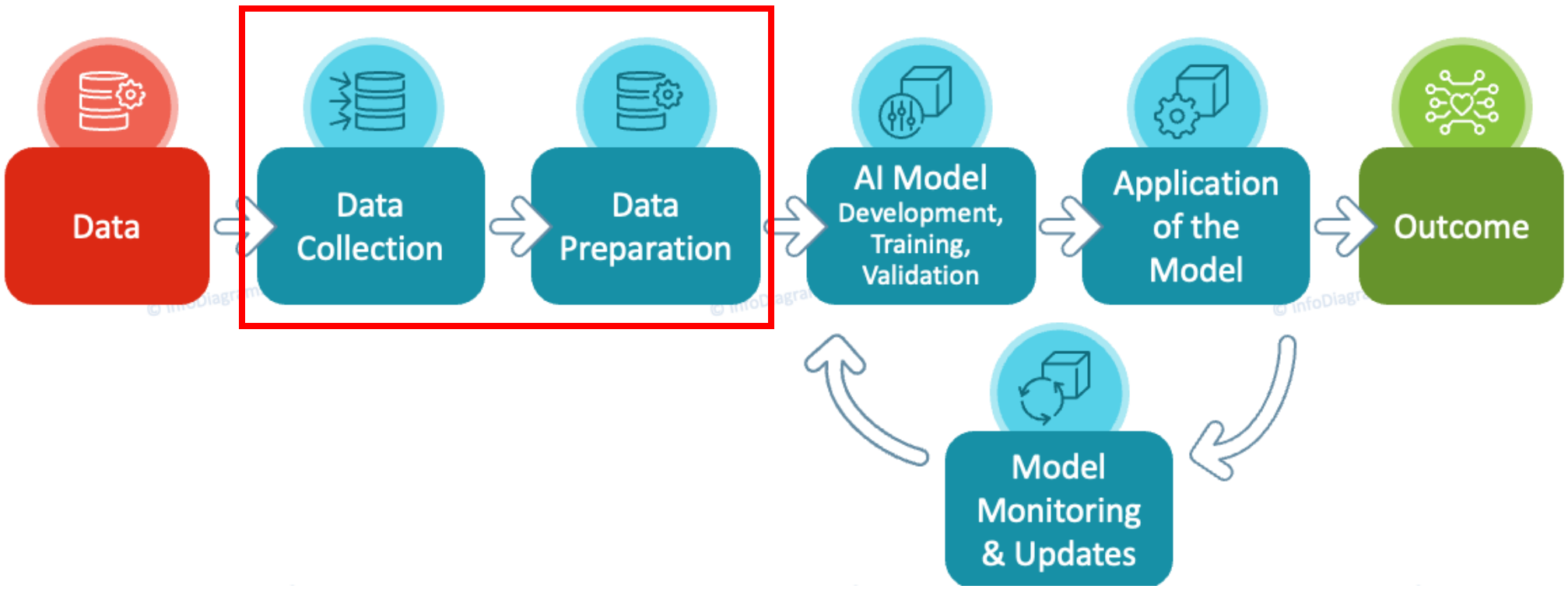
IMEN891K/IDSC501 Data Engineering

12. **Big Data I**: MapReduce

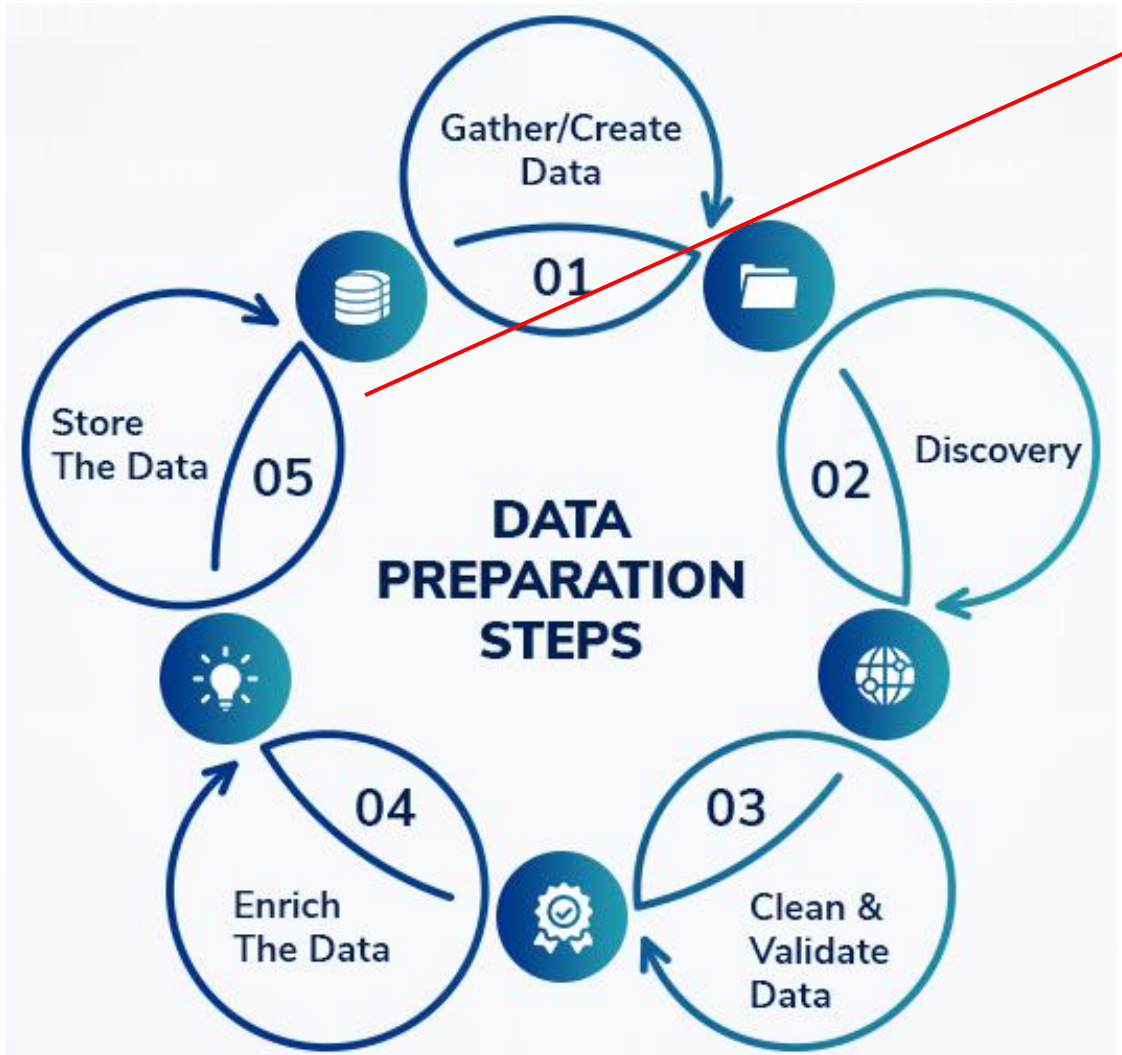
Fall 2025

Instructor: **Sangwoo Mo**

Recap: Course Overview



Recap: Course Overview



Lecture 7-15

Disclaimer

The NoSQL part was heavily copy-pasted from Berkeley CS186

Introduction to Database Systems

UC Berkeley, Fall 2024



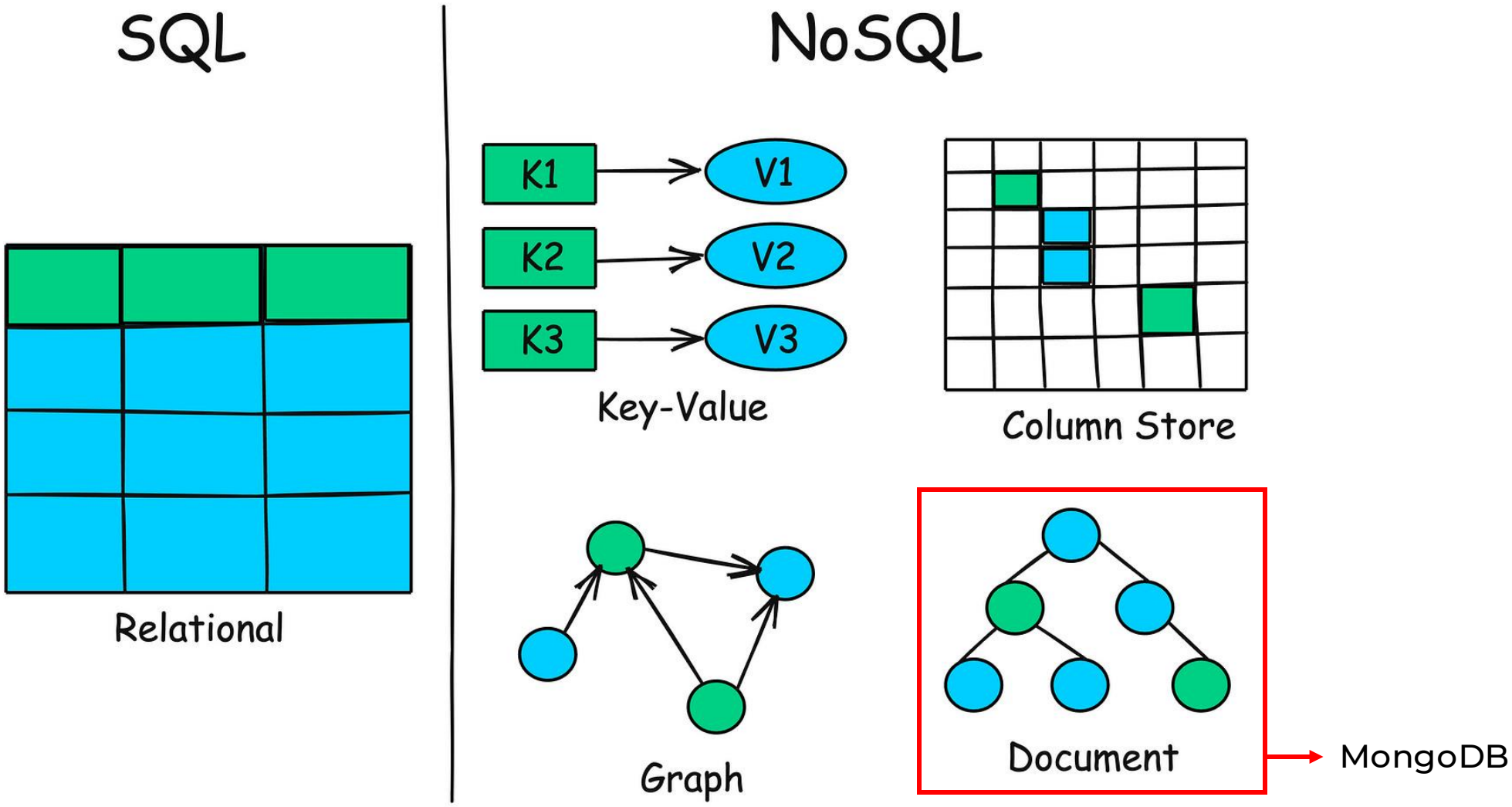
Alvin Cheung
akcheung@cs.berkeley.edu

Pronouns: he/him/his

OH: Tues 11am-noon on zoom

Hi all! I'm an EECS faculty and my group works on data systems research. Looking forward to meeting you all!

Recap: Types of NoSQL



https://blog.algomaster.io/p/sql-vs-nosql-7-key-differences

Recap: MongoDB Query Language

- Input = collections, output = collections

- Three main types of queries in the query language
  - Retrieval: Restricted SELECT-WHERE-ORDER BY-LIMIT type queries
  - Aggregation: A bit of a misnomer; a general pipeline of operators
    - Can capture Retrieval as a special case
    - But worth understanding Retrieval queries first…
  - Updates

Recap: MongoDB Query Language

- Input = collections, output = collections
- Three main types of queries in the query language
  - Retrieval: Restricted SELECT-WHERE-ORDER BY-LIMIT type queries
  - Aggregation: A bit of a misnomer; a general pipeline of operators
    - Can capture Retrieval as a special case
    - But worth understanding Retrieval queries first…
  - Updates
- All queries are invoked as
  - db.collection.operation1(…).operation2(…)…
    - collection: name of collection
  - Unlike SQL which lists many tables in a FROM clause, MQL is centered around manipulating a single collection

Recap: Retrieval

db.collection.**find**(<*predicate*>, optional <*projection*>)

returns documents that match <*predicate*>

keep fields as specified in <*projection*>

both <*predicate*> and <*projection*> expressed as documents

in fact, most things are documents!

Recap: Retrieval

```
db.inventory.find(
            { tags : red },
            {_id : 0, instock : 0} )
.sort ( { "dim.0": -1, item: 1 } )
.limit (2)
```

Recap: Retrieval

```
db.inventory.find(
            { tags : red },
            {_id : 0, instock : 0} )
.sort ( { "dim.0": -1, item: 1 } )
.limit (2)
```

FROM

WHERE

SELECT

ORDER BY

LIMIT

Recap: Retrieval

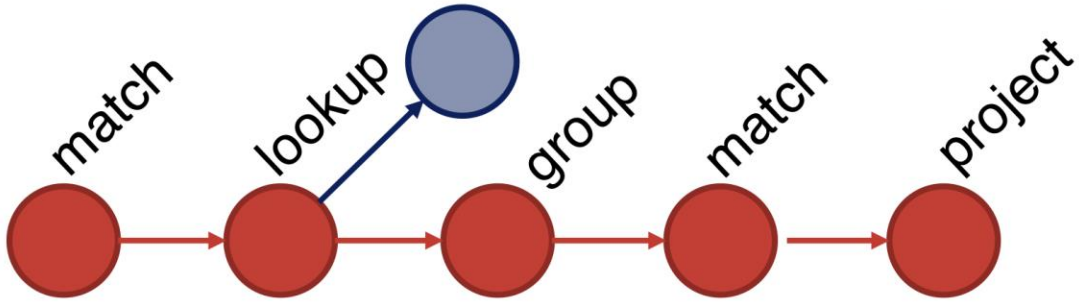
```
find() =   SELECT <projection>
           FROM Collection
           WHERE <predicate>
limit() =  LIMIT
sort()  =  ORDER BY
```

```
db.inventory.find(
           { tags : red },
           {_id : 0, instock : 0} )
.sort ( { "dim.0": -1, item: 1 } )
.limit (2)
```

```
FROM
WHERE
SELECT
ORDER BY
LIMIT
```

Recap: Aggregation

- Composed of a linear *pipeline* of *stages*
- Each stage corresponds to one of:
  - match // first arg of find ( )
  - project // second arg of find ( ) but more expressiveness
  - sort/limit // same
  - group
  - unwind
  - lookup
  - … lots more!!
- Each stage manipulates the existing collection in some way



- Syntax:

```
db.collection.aggregate ( [
 { $stage1Op: { } },
 { $stage2Op: { } },
  …
 { $stageNOp: { } }
] )
```

Recap: Grouping

```
$group : {
        _id: <expression>, // Group By Expression
        <field1>: { <aggfunc1> : <expression1> },
        ... }
```

Returns one document per unique group, indexed by _id

Agg.func. can be standard ops like $sum, $avg, $max

Recap: Grouping

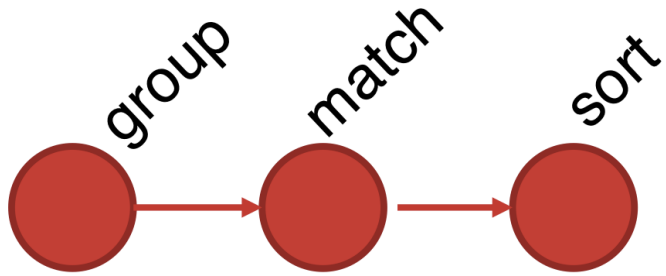
Find states with population > 15M, sort by decending order

```
db.zips.aggregate( [
{ $group: { _id: "$state", totalPop: { $sum: "$pop" } } },
{ $match: { totalPop: { $gte: 15000000 } } },
{ $sort : { totalPop : -1 } }
 ] )
```

GROUP BY

AGGS.

match after group = HAVING



Recap: Grouping

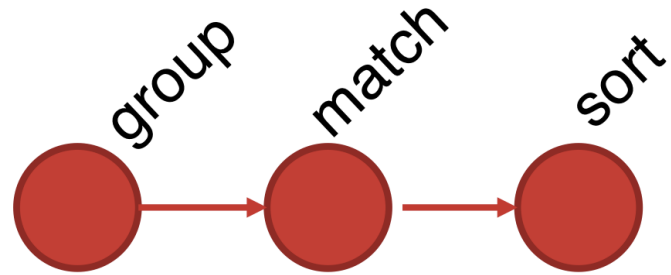
Find states with population > 15M, sort by decending order

```
db.zips.aggregate( [
{ $group: { _id: "$state", totalPop: { $sum: "$pop" } } },
{ $match: { totalPop: { $gte: 15000000 } } },
{ $sort : { totalPop : -1 } }
 ] )
```

GROUP BY

AGGS.

match after group = HAVING




```
SELECT state AS id, SUM(pop) AS totalPop
FROM zips
GROUP BY state
HAVING totalPop >= 15000000
ORDER BY totalPop DESCENDING
```

Q: what would the SQL query for this be?

Motivation of Distributed Systems

- Data models:
  - Key-value
  - JSON
- Single node implementation: MongoDB

- Next: *parallel* NoSQL databases!

Motivation of Distributed Systems

- We learned how to parallelize relational database systems
- While useful, it might incur too much overhead if our query plans consist of simple operations
- MapReduce is a NoSQL programming model for such computation
- First, let’s study how data is stored in such systems

Distributed File System (DFS)

- For very large files: TBs, PBs
- Each file is partitioned into *chunks*, typically 64MB
- Each chunk is replicated several times (≥3), on different racks, for fault tolerance
- Implementations:
  - Google's DFS: GFS, proprietary
  - Hadoop's DFS: HDFS, open source

MapReduce

- Google: paper published 2004
- Free variant: Hadoop 

- MapReduce: high-level programming model and implementation for large-scale parallel data processing

Typical Problems Solved by MapReduce

- Read a lot of data
- Map: extract something you care about from each record
- Shuffle and Sort
- Reduce: aggregate, summarize, filter, transform
- Write the results

Paradigm stays the same,
change map and reduce
functions for different problems

Data Model

Files!

A file = a bag of `(key, value)` pairs

A MapReduce program:

- Input: a bag of `(inputkey, value)` pairs
- Output: a bag of `(outputkey, value)` pairs
  - outputkey is optional

Step 1: the MAP Phase

User provides the MAP-function:

- Input: `(input key, value)`
- Output: bag of `(intermediate key, value)`

System applies the map function in parallel to all `(input key, value)` pairs in the input file

Step 2: the SHUFFLE Phase

This is completely internal to the system

- Input: Pairs of `(intermediate key, value)`
- Output: Pairs of
  `(intermediate key, bag of values with the same intermediate key)`

I.e., group all values with the same key

Step 3: the REDUCE Phase

User provides the REDUCE function:

- Input: `(intermediate key, bag of values)`
- Output: bag of output `(values)`

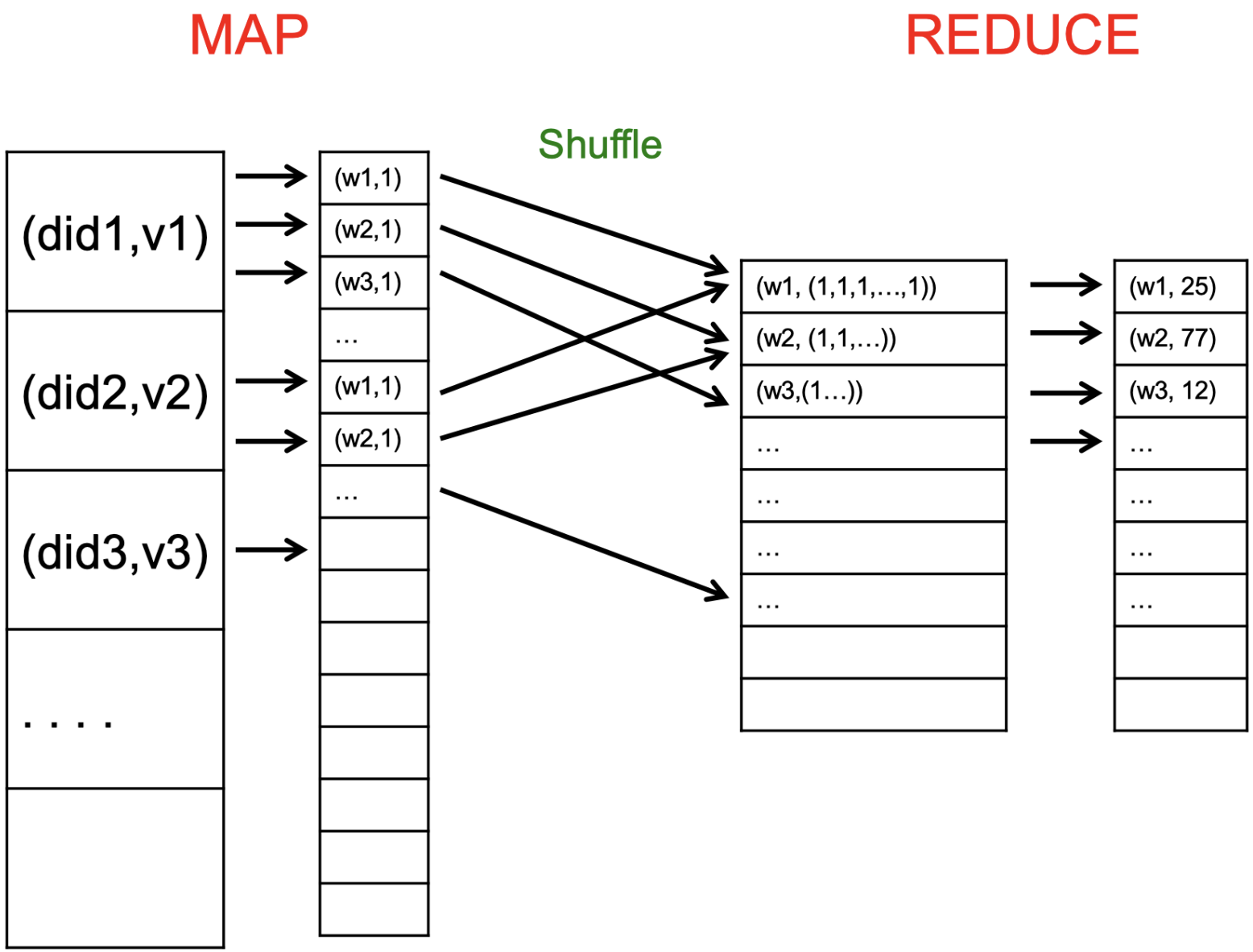
System groups all pairs with the same intermediate key,
and passes the bag of values to the REDUCE function

Example

- Counting the number of occurrences of each word in a large collection of documents
- Each Document
  - The key = document id (did)
  - The value = set of words (word)

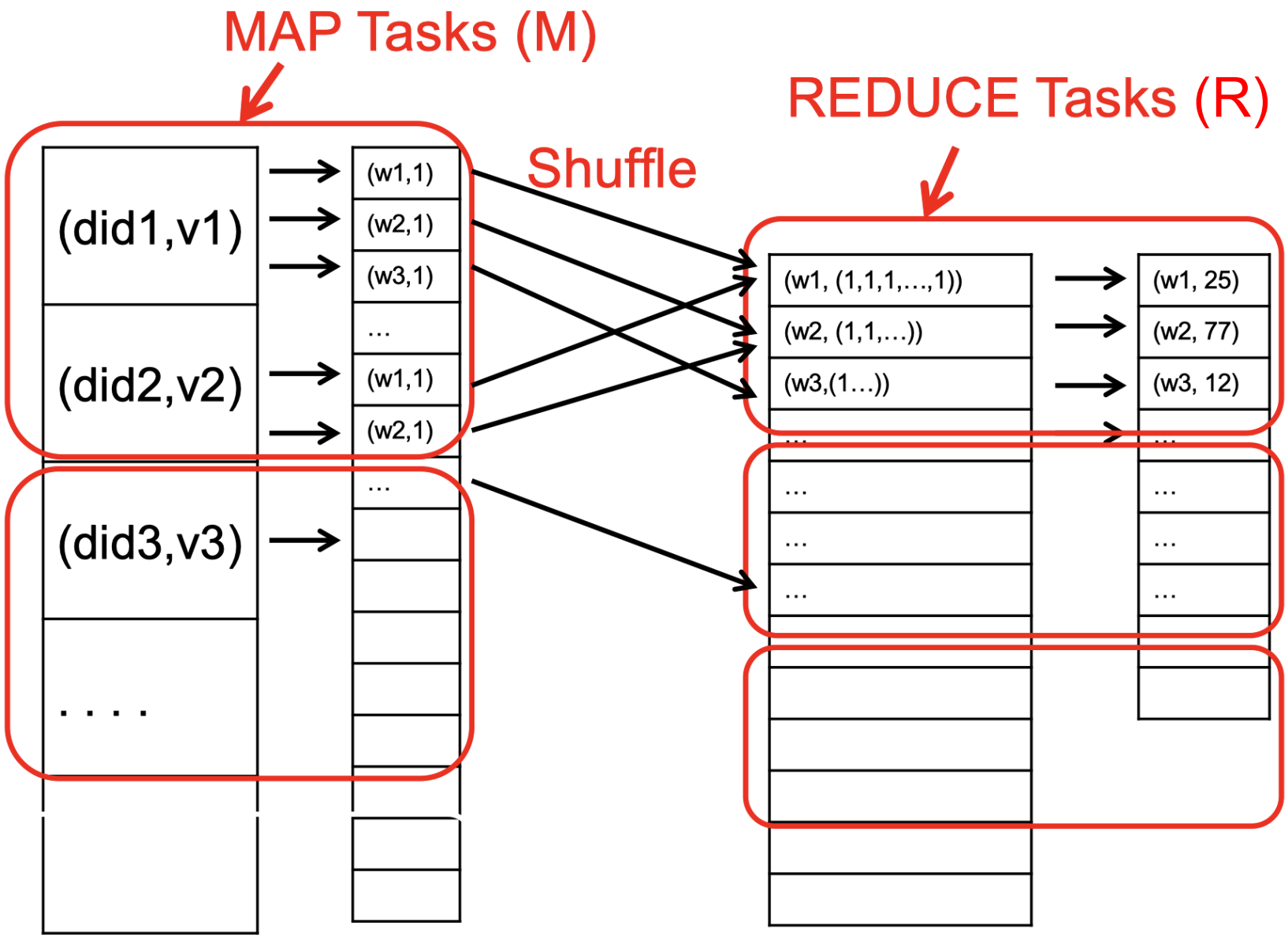
```
map(String key, String value):
   // key: document name
   // value: document contents
   for each word w in value:
        emitIntermediate(w, “1”);
```

```
reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
    for each v in values:
      result += ParseInt(v);
  emit(AsString(result));
```


MAP
REDUCE
Shuffle
(did1,v1)
(did2,v2)
(did3,v3)
. . . .
(w1,1)
(w2,1)
(w3,1)
…
(w1,1)
(w2,1)
…
(w1, (1,1,1,…,1))
(w2, (1,1,…))
(w3,(1…))
…
…
…
…
(w1, 25)
(w2, 77)
(w3, 12)
…
…
…
…

Workers

- A worker is a process that executes one task at a time
- Typically there is one worker per processor, hence 4 or 8 per node


MAP Tasks (M)
REDUCE Tasks (R)
Shuffle
(did1,v1)
(did2,v2)
(did3,v3)
. . . .
(w1,1)
(w2,1)
(w3,1)
…
(w1,1)
(w2,1)
…
(w1, (1,1,1,…,1))
(w2, (1,1,…))
(w3,(1…))
…
…
…
…
(w1, 25)
(w2, 77)
(w3, 12)
…
…
…

Fault Tolerance

- If one server fails once every year…
  ... then a job with 10,000 servers will fail in less than one hour

- MapReduce handles fault tolerance by writing intermediate files to disk:
  - Mappers write file to local disk
  - Reducers read the files (=reshuffling); if the server fails, the reduce task is restarted on another server

Implementation

- There is one master node
- Master partitions input file into *M splits*, by key
- Master assigns *workers* (=servers) to the *M map tasks*, keeps track of their progress
- Workers write their output to local disk, partition into *R regions*
- Master assigns workers to the *R reduce tasks*
- Reduce workers read regions from the map workers' local disks

USING MAPREDUCE IN PRACTICE:

IMPLEMENTING RA OPERATORS IN MR

Relational Operators in MapReduce

Given relations R(A,B) and S(B,C) compute:

- Selection:
- Group-by:
- Join:

Relational Operators in MapReduce

Given relations R(A,B) and S(B,C) compute:

- Selection: $\sigma_{A=123}(R)$

```
SELECT *
FROM R
WHERE A = 123;
```

- Group-by:
- Join:

Relational Operators in MapReduce

Given relations R(A,B) and S(B,C) compute:

- Selection: $\sigma_{A=123}(R)$
- Group-by: $\gamma_{A,sum(B)}(R)$ →

```
SELECT A, SUM(B)
FROM R
GROUP BY A;
```

- Join:

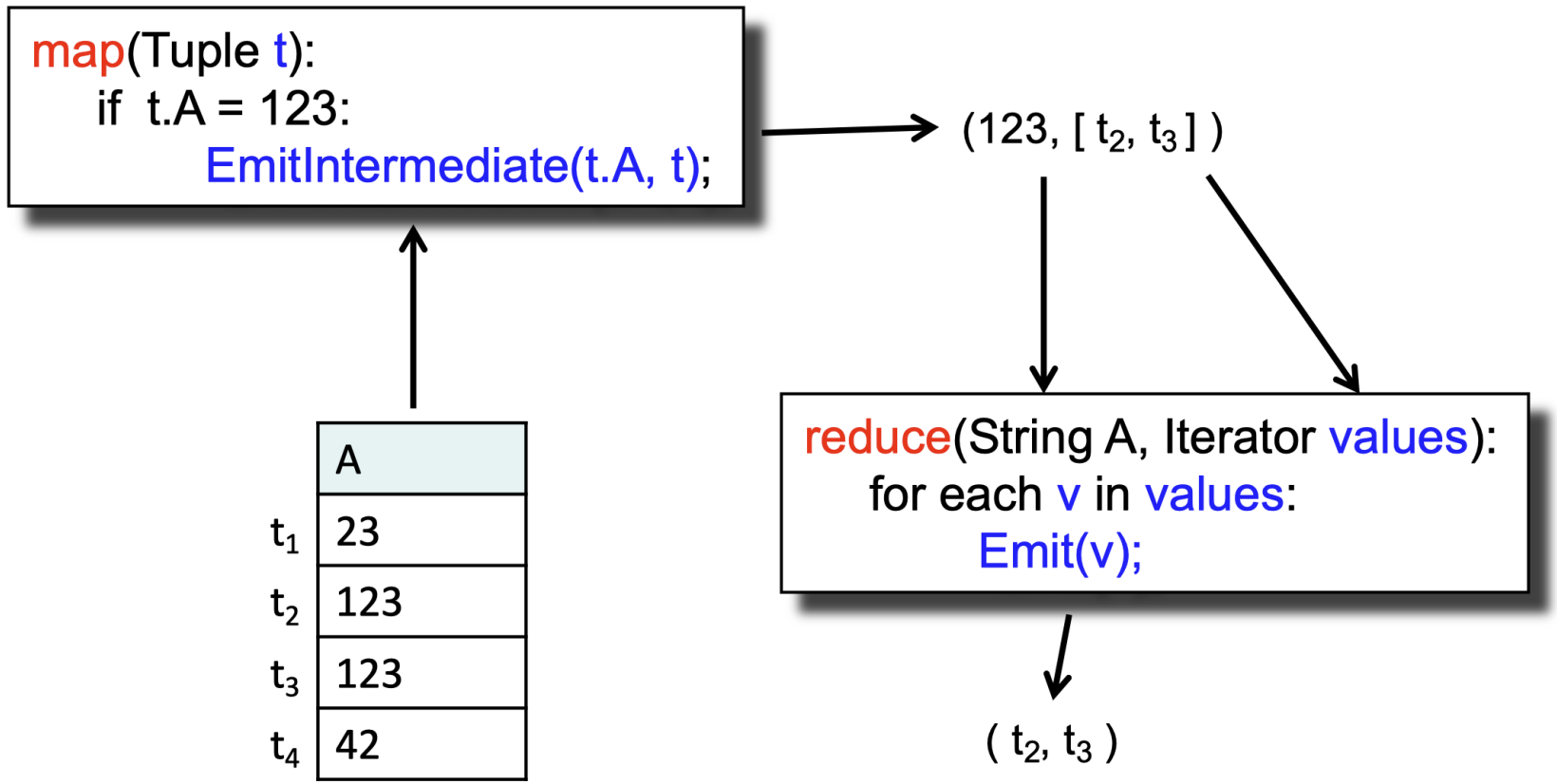
Relational Operators in MapReduce

Given relations R(A,B) and S(B,C) compute:

- Selection: $\sigma_{A=123}(R)$
- Group-by: $\gamma_{A,sum(B)}(R)$
- Join: $R \bowtie S$ ⟶

```
SELECT *
FROM R
JOIN S ON R.B = S.B;
```

Selection $\sigma_{A=123}(R)$



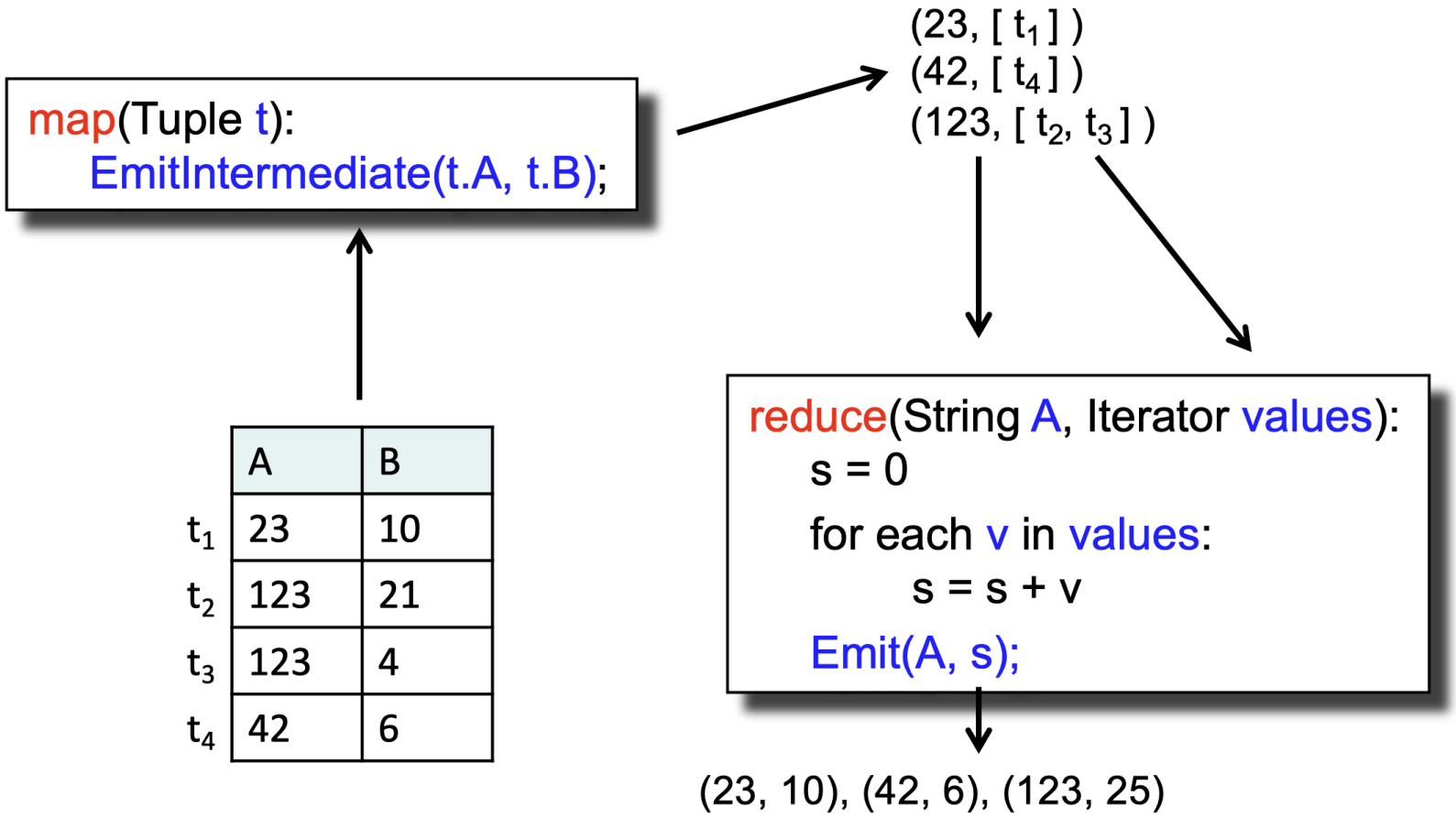
Selection $\sigma_{A=123}(R)$

```
map(Tuple t):
    if  t.A = 123:
            EmitIntermediate(t.A, t);
```

```
reduce(String A, Iterator values):
    for each v in values:
        Emit(v);
```

No need for reduce.
But need system hacking in Hadoop
to remove reduce from MapReduce

Group By $\gamma_{A,\,\mathrm{sum}(B)}(R)$

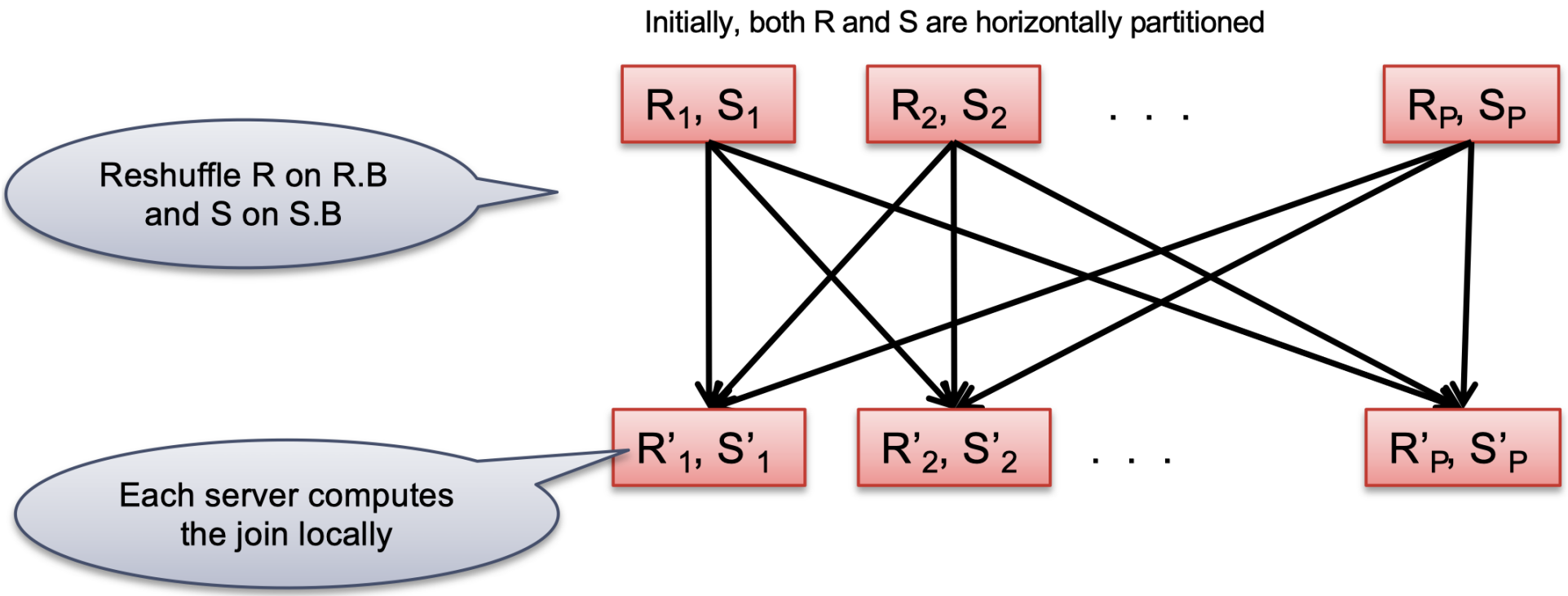


	A	B
t_1	23	10
t_2	123	21
t_3	123	4
t_4	42	6

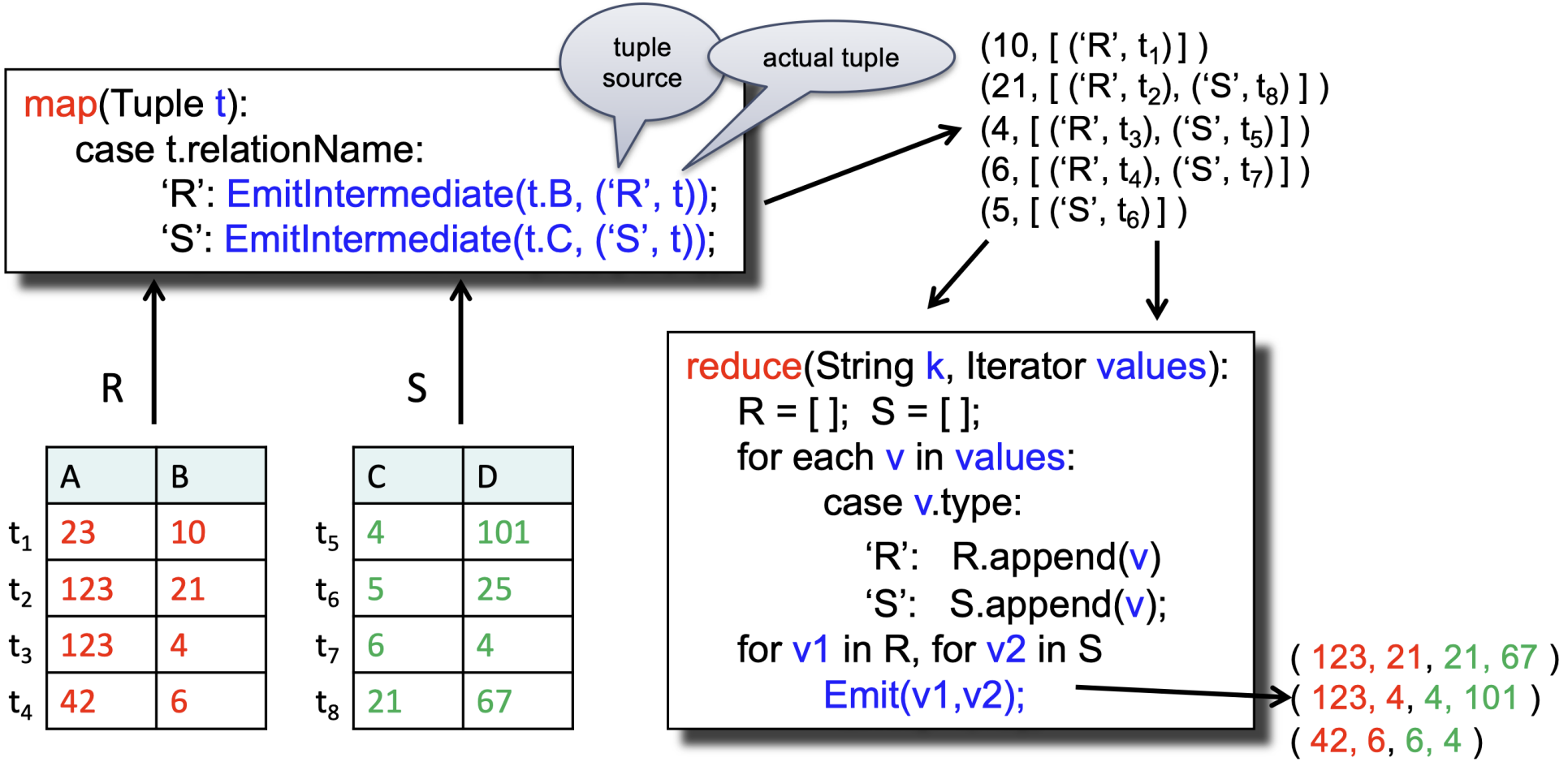
Join

- Partitioned hash-join
- Broadcast join

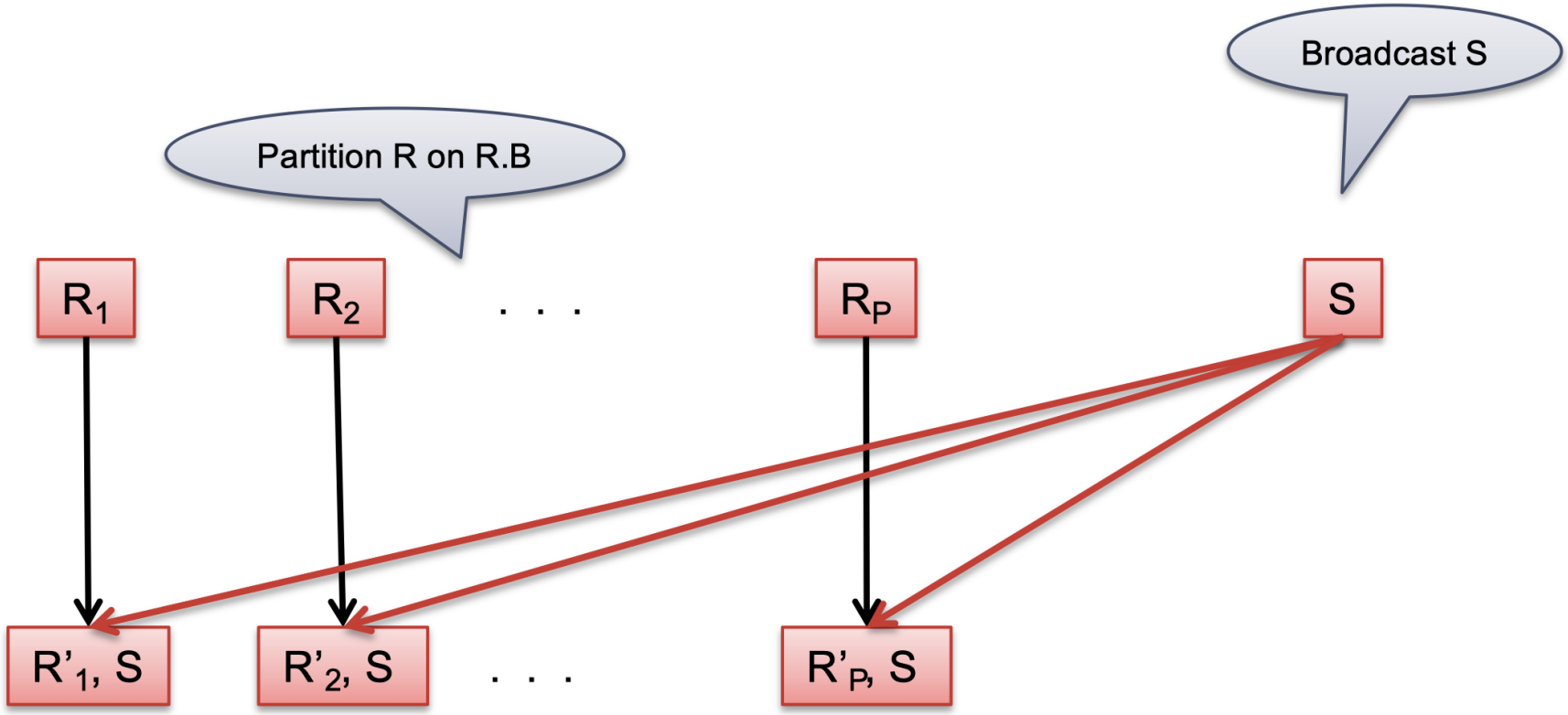
Partitioned Hash-Join (a.k.a. Shuffle Join)



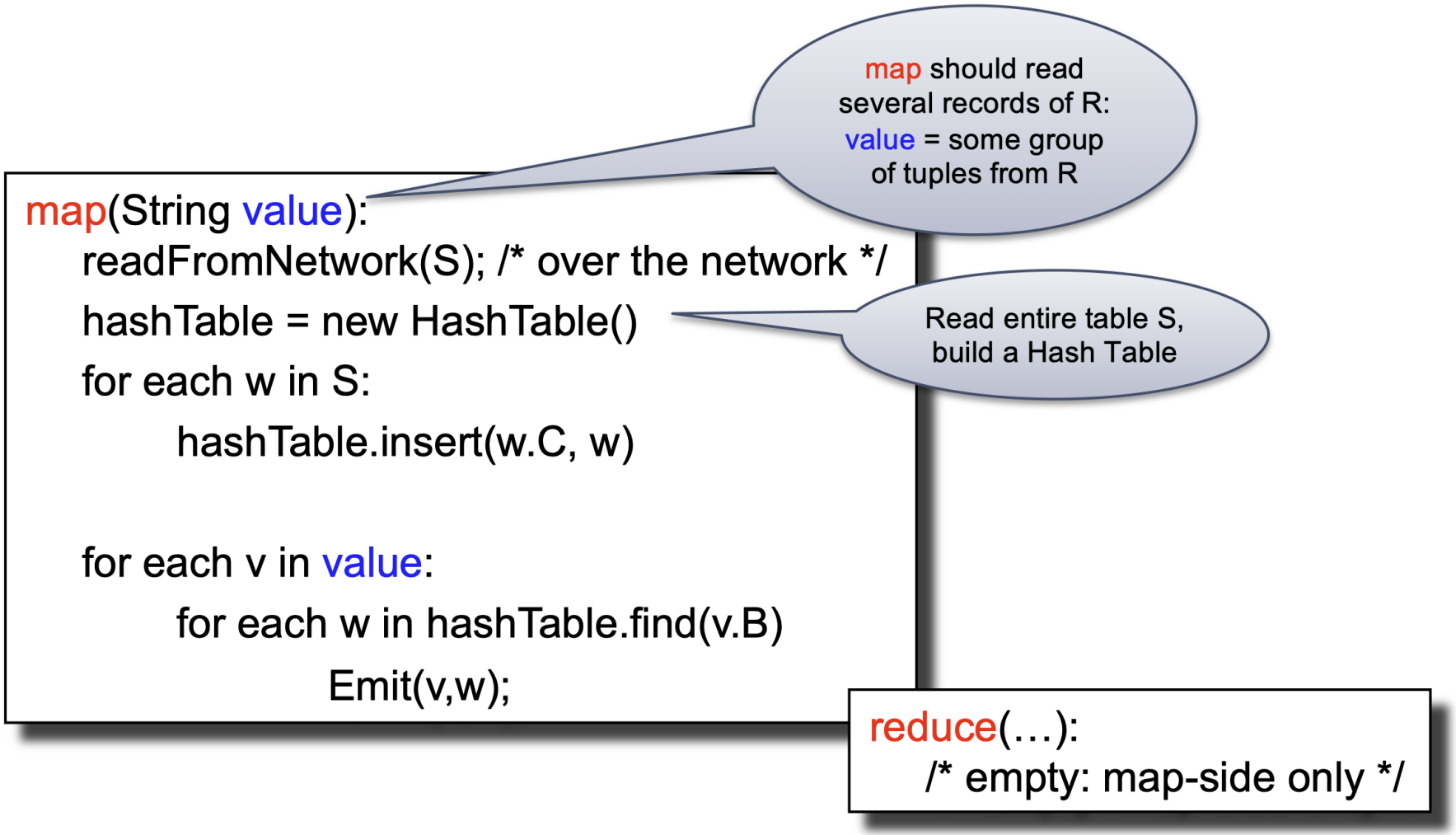
Partitioned Hash-Join (a.k.a. Shuffle Join)



Broadcast Join (a.k.a Map-Side Join)



Broadcast Join (a.k.a Map-Side Join)



Comparison of Join Algorithms

Category	Partitioned Hash Join	Broadcast Join
When to Use	When **both tables are large**, or data is not already partitioned by the join key	When **one table is very small** (a few MBs to tens of MBs) and can be replicated to all worker nodes
Data Scale	Large ↔ Large (e.g., fact–fact join)	Large ↔ Small (e.g., fact–dimension join)
Data Movement	Both tables are shuffled across the network	Only the small table is sent to all nodes (broadcasted)
Performance	More network I/O → slower, but **stable for large data**	No shuffle → **very fast**, but **memory-dependent**
Advantages	• Handles very large datasets reliably • Even workload distribution	• Very fast (no shuffle) • Minimal network cost
Limitations	• High shuffle cost • Longer execution time	• Small table size limit • Risk of out-of-memory errors
Rule of Thumb	**"If both are big → use partitioned hash join."**	**"If one is small → use broadcast join."**

MapReduce Summary

- MapReduce offers a simple abstraction, and handles distribution + fault tolerance
- Speedup from dynamically allocating map and reduce tasks and to servers. Skew is possible!
- Write intermediate results to disk for fault tolerance but can be slow

MapReduce Summary

- MapReduce offers a simple abstraction, and handles distribution + fault tolerance
- Speedup from dynamically allocating map and reduce tasks and to servers. Skew is possible!
- Write intermediate results to disk for fault tolerance but can be slow

- Spark replaces this with “Resilient Distributed Datasets” = main memory + lineage

Issues with MapReduce

- Difficult to write more complex queries
  - Everything has to be expressed as map-reduce
- Need multiple MapReduce jobs: dramatically slows down because it writes all (intermediate) results to disk

Spark

- Open source system developed right here!
- Distributed processing over HDFS
- Differences from MapReduce:
  - Multiple steps, including iterations
  - Stores intermediate results in main memory
  - Closer to relational algebra (familiar to us)
- Details: http://spark.apache.org

Data Model: Resilient Distributed Datasets (RDD)

- RDD = Resilient Distributed Datasets
  - A distributed, immutable relation, together with its *lineage*
  - Lineage: expression that says how that relation was computed (e.g., a relational algebra plan)
- All inputs, outputs, and intermediates are stored as RDDs
- If a server crashes, its RDD in main memory is lost. However, the driver (=master node) knows the lineage, and will simply recompute the lost partition of the RDD

Data Model: Resilient Distributed Datasets (RDD)

> **RDD = main memory + lineage**

1. **Main memory** → Eliminates redundant disk I/O, enabling fast iterative computation (e.g., ML, graph algorithms).
2. **Lineage** → Stores only *how* data was derived, so Spark can recover lost data without checkpoints.

This design gave Spark:

- ~10–100× faster performance than Hadoop
- The ability to unify **batch, streaming, and SQL** in one engine
- Fault tolerance without heavy disk overhead

MapReduce vs RDD

> Hadoop: "**Write to disk after every step**"
> Spark: "**Keep everything in memory and execute as one logical plan.**"

Example:

- Hadoop:

```scss
Map() → write to HDFS → Reduce() → write to HDFS → next job...
```

- Spark:

```python
rdd.map(...).filter(...).groupBy(...).collect()
```

→ Spark builds a **DAG** internally and executes it efficiently in memory.

MapReduce vs RDD

Category	Hadoop MapReduce	Spark (RDD)
Core Idea	Disk-based **batch execution**	In-memory **resilient computation**
Data Storage	Writes intermediate results to **HDFS** after every stage	Keeps data in **memory (RDD)**, reconstructs via **lineage** if lost
Fault Tolerance	**Re-executes tasks** on failure (checkpoint-based)	**Recomputes** lost partitions from lineage graph
Performance	High disk I/O overhead (read/write between every step)	10–100× faster via in-memory execution
Iteration / Pipeline	Each iteration = new MapReduce job	Builds a **DAG** of transformations executed together
Programming Model	Limited to `map` / `reduce`, Java-heavy	Rich functional API (`map`, `filter`, `join`, `reduceByKey`), Python/Scala-friendly
Flexibility	Batch only	Unified engine for batch, streaming, and SQL