Data Management Using MapReduce

M. Tamer Özsu

University of Waterloo

Basics

- For data analysis of very large data sets
  - Highly dynamic, irregular, schemaless, etc.
  - SQL too heavy
- "Embarrassingly parallel problems"
- New, simple parallel programming model
  - Data structured as (key, value) pairs
    - E.g. (doc-id, content), (word, count), etc.
  - Functional programming style with two functions to be given:
    - `Map(k1,v1)` $\rightarrow$ `list(k2,v2)`
    - `Reduce(k2, list (v2))` $\rightarrow$ `list(v3)`
- Implemented on a distributed file system (e.g., Google File System) on very large clusters

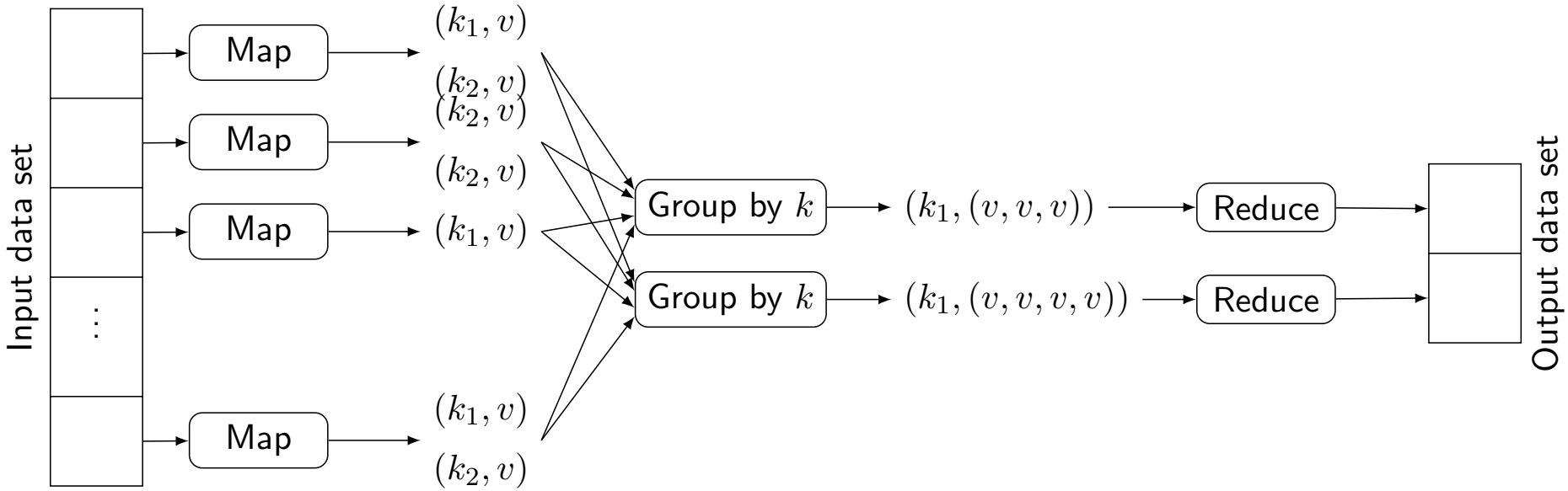
Map Function

- User-defined function
  - Processes input key/value pairs
  - Produces a set of *intermediate* key/value pairs
- Map function I/O
  - **Input:** read a *chunk* from distributed file system (DFS)
  - **Output:** Write to intermediate file on local disk
- MapReduce library
  - Executes map function
  - Groups together all intermediate values with the same key (i.e., generates a set of lists)
  - Passes these lists to reduce functions
- Effect of map function
  - Processes and partitions input data
  - Builds a distributed map (trnasparent to user)
  - Similar to "group by" operaton in SQL

Reduce Function

- User-defined function
  - Accepts one intermediate key and a set of values for that key (i.e., a list)
  - Merges these values together to form a (possibly) smaller set
  - Typically, zero or one output value is generated per invocation
- Reduce function I/O
  - **Input:** read from intermediate files using remote reads on local files of corresponding mapper nodes
  - **Output:** Each reducees writes its output as a file back to DFS
- Effect of map function
  - Similar to aggregation operaton in SQL

MapReduce Processing



Example 1

Assume you are reading the monthly average temperatures for each of the 12 months of a year for a bunch of cities, i.e., each input is the name of the city (key) and the average monthly temperature. Compute the average annual temperature for each city.

- `Map`:
  Input: ⟨`City, Month, MonthAvgTemp`⟩
  1. Create key/value pairs

  Output: ⟨`City, MonthAvgTemp`⟩
- `Reduce`:
  Input: ⟨`City, MonthAvgTemp`⟩
  1. Sort to get ⟨`City, list(MonthAvgTemp)`⟩ (i.e., it combines in a list the monthly average temperatures for a given city)
  2. Compute average over `list(MonthAvgTemp)`

  Output: ⟨`City, AnnualAvgTemp`⟩

Example 2

- Consider EMP (ENAME, TITLE, CITY)
- Query:

```
SELECT CITY, COUNT(*)
FROM EMP
WHERE ENAME LIKE "\%Smith"
GROUP BY CITY
```

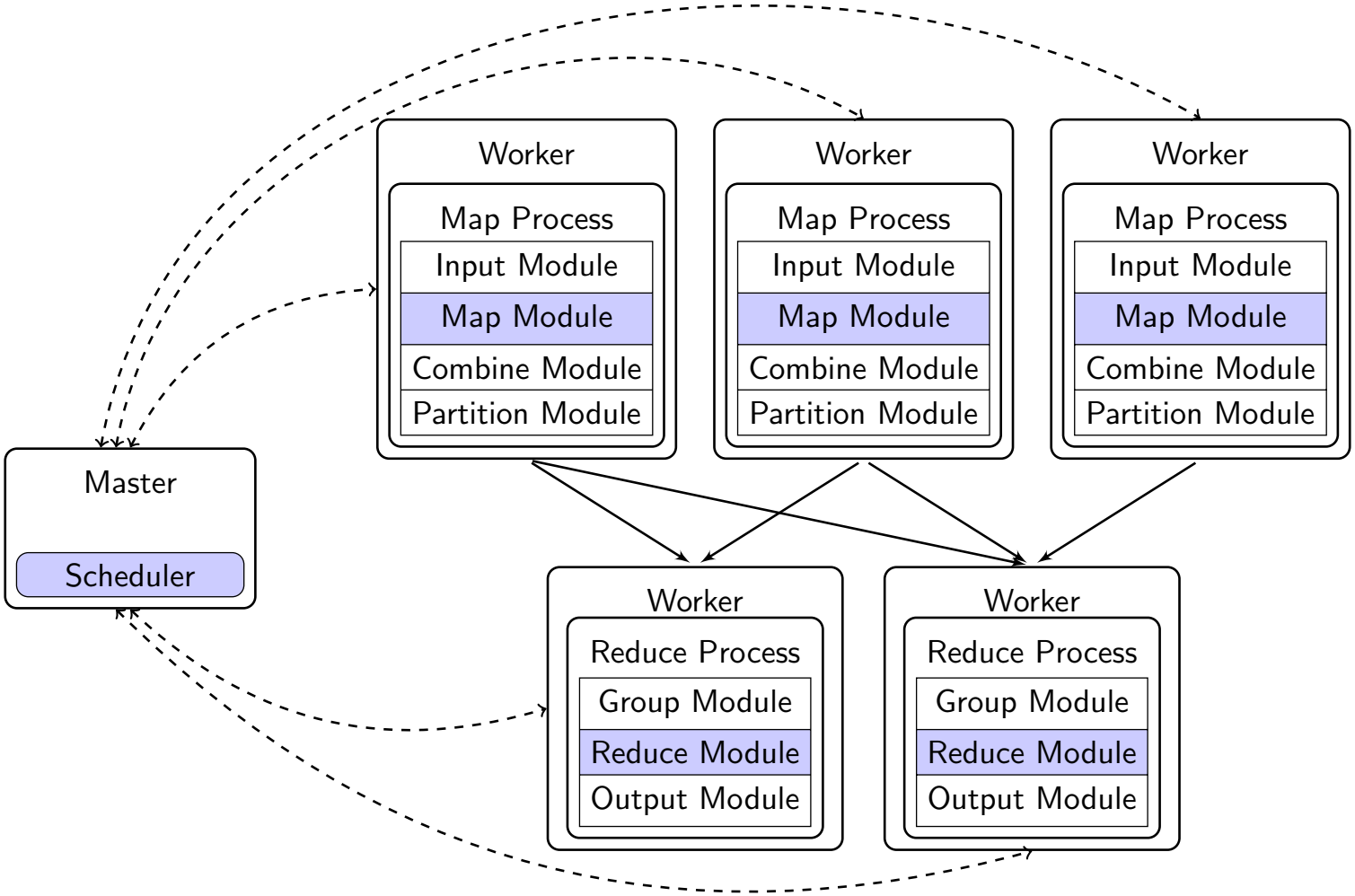
- Map:

```
Input: (TID,emp), Output: (CITY,1)
  if emp.ENAME like "\%Smith" return (CITY,1)
```

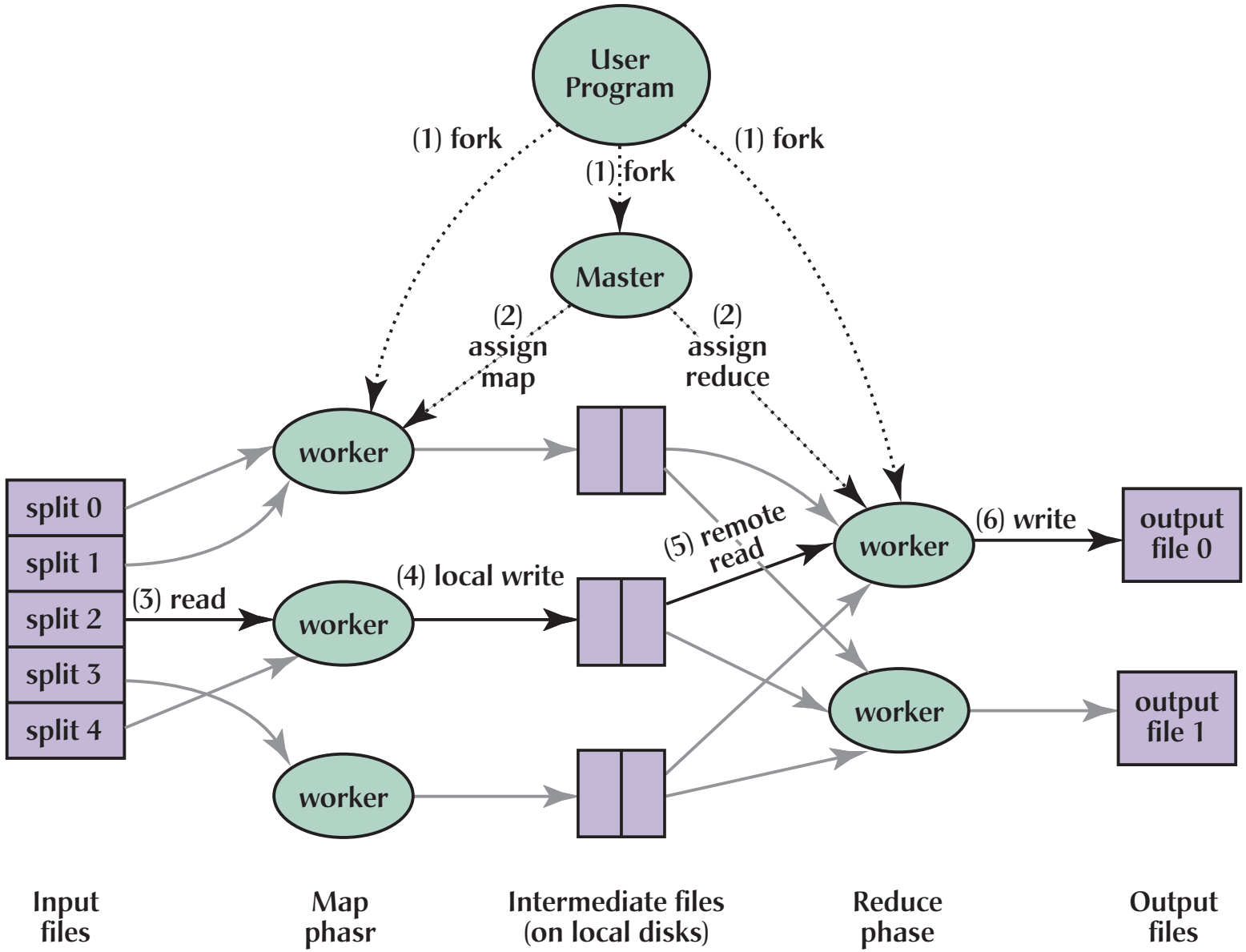
- Reduce:

```
Input: (CITY,list(1)), Output: (CITY,SUM(list(1)))
  return (CITY,SUM(1*))
```

MapReduce Architecture



Execution Flow with Architecture



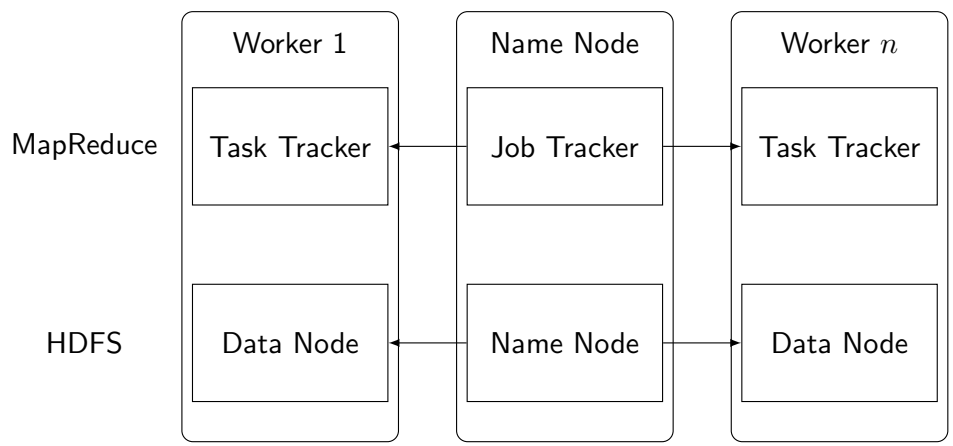
From: J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Comm. ACM*, 2008.

Characteristics

- Flexibility
  - User can write any `map` and `reduce` function code
  - No need to know how to parallelize
- Scalability
  - Elastic scalability
  - Automatic load balancing
- Efficiency
  - Simple: parallel scan
  - No database loading
- Fault tolerance
  - Worker failure
    - Master pings workers periodically; assumes failure if no response
    - Tasks (both map and reduce) on failed workers scheduled on a different worker node
  - Master failure
    - Checkpoints of master state
    - Recovery after failure $\rightarrow$ progress can halt
  - Replication on distributed data store

Hadoop

- Most popular MapReduce implementation – developed by Yahoo!
- Two components
  - Processing engine
  - HDFS: Hadoop Distributed Storage System – others possible
  - Can be deployed on the same machine or on different machines
- Processes
  - Job tracker: hosted on the master node and implements the schedule
  - Task tracker: hosted on the worker nodes and accepts tasks from job tracker and executes them
- HDFS
  - Name node: stores how data are partitioned, monitors the status of data nodes, and data dictionary
  - Data node: Stores and manages *data chunks* assigned to it



Hadoop UDF Functions

Phase	Name	Function
Map	InputFormat::getSplit	Partition the input data into different splits. Each split is processed by a mapper and may consist of several chunks.
	RecordReader::next	Define how a split is divided into items. Each item is a key/value pair and used as the input for the `map` function.
	Mapper::map	Users can customize the `map` function to process the input data. The input data are transformed into some intermediate key/value pairs.
	WritableComparable::compareTo	The comparison function for the key/-value pairs.
	Job::setCombinerClass	Specify how the key/value pair are aggregated locally.
Shuffle	Job::setPartitionerClass	Specify how the intermediate key/value pairs are shuffled to different reducers.
Reduce	Job::setGroupingComparatorClass	Specify how the key/value pairs are grouped in the reduce phase.
	Reducer::reduce	Users write their own `reduce` functions to perform the corresponding jobs.

MapReduce Implementations

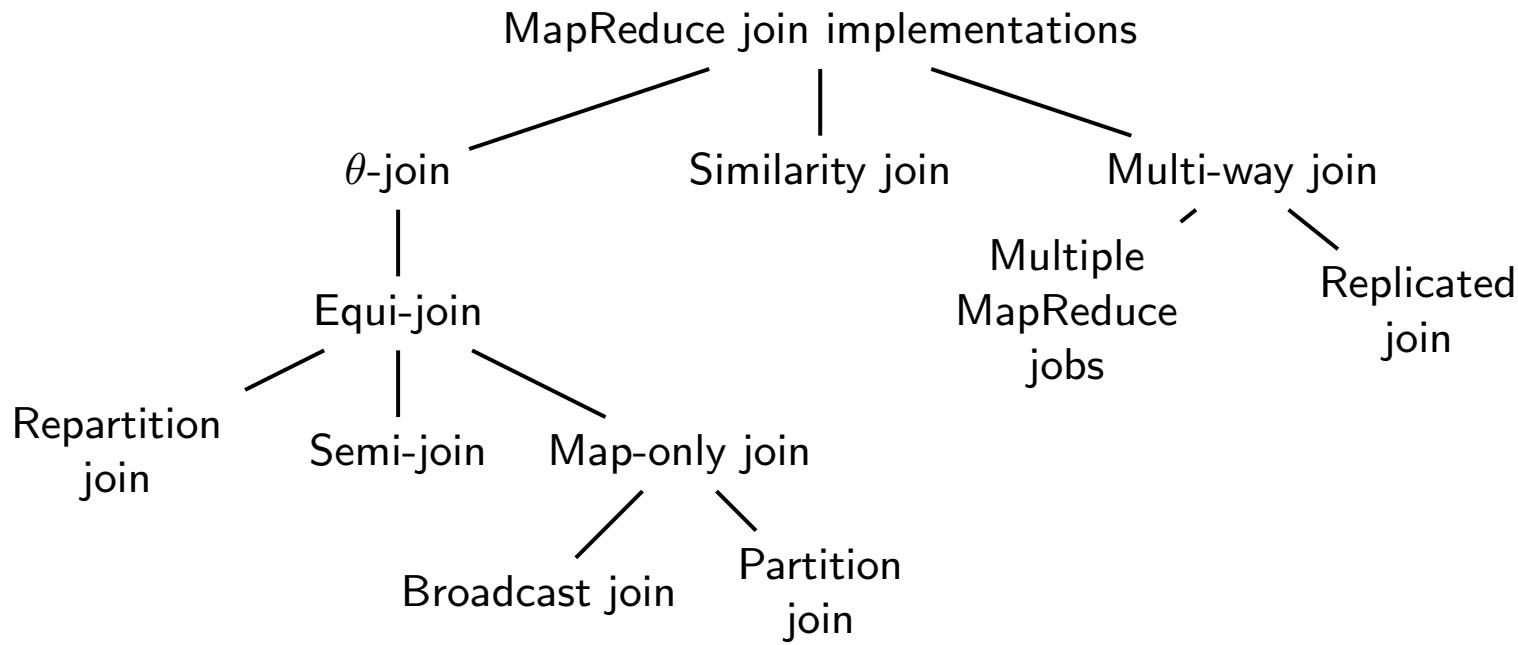
Name	Language	File System	Index	Master Server	Multiple Job Support
Hadoop	Java	HDFS	No	Name Node and Job Tracker	Yes
Disco	Python and Erlang	Distributed Index	Disco Server	No	No
Skynet	Ruby	MySQL or Unix File System	No	Any node in the cluster	No
FileMap	Shell and Perl Scripts	Unix File System	No	Any node in the cluster	No
Twister	Java	Unix File System	No	One master node in broker network	Yes
Cascading	Java	HDFS	No	Name Node and Job Tracker	Yes

MapReduce Languages

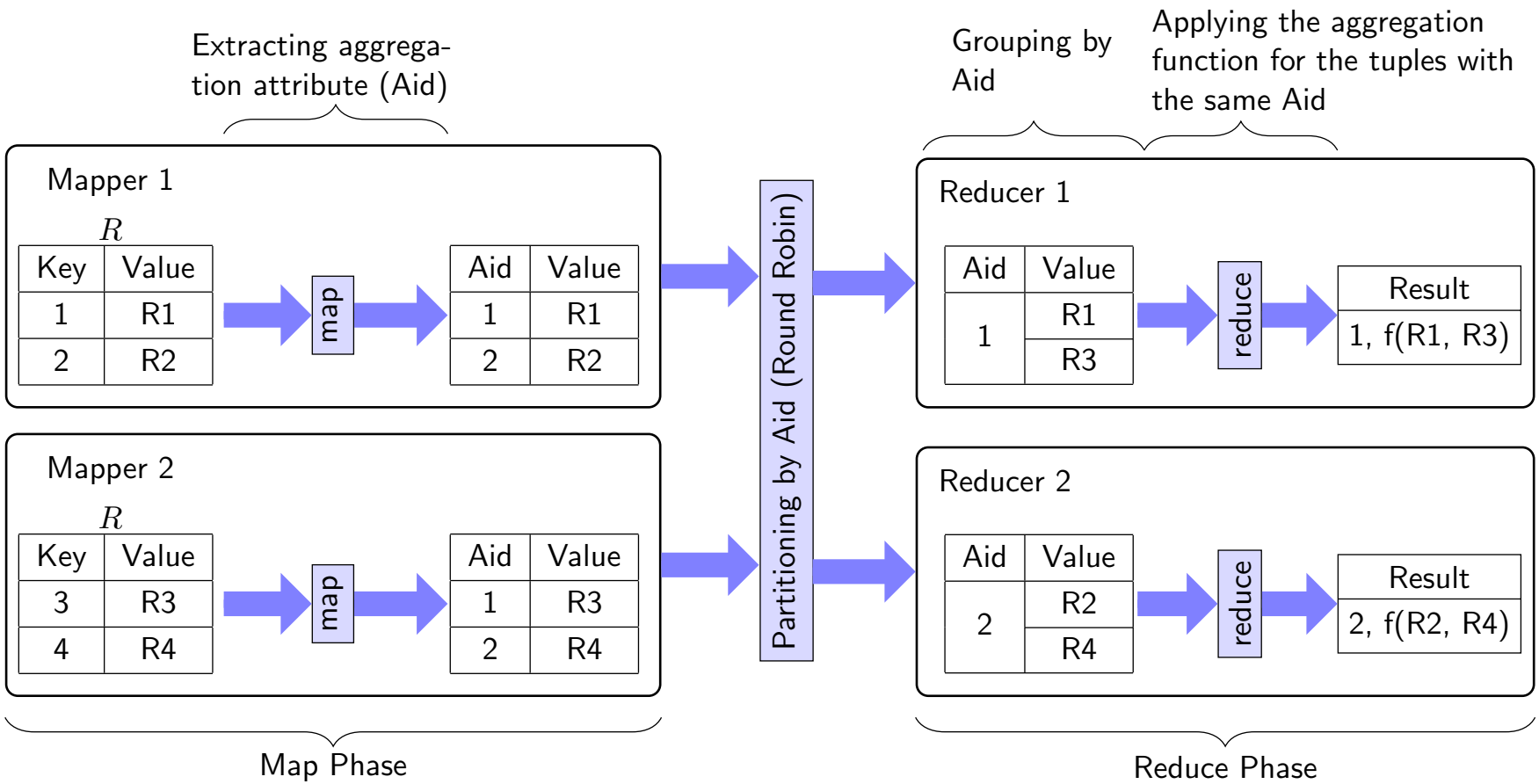
	Hive	Pig
Language	Declarative SQL-like	Dataflow
Data model	Nested	Nested
UDF	Supported	Supported
Data partition	Supported	Not supported
Interface	Command line, web, JDBC/ODBC server	Command line
Query optimization	Rule based	Rule based
Metastore	Supported	Not supported

MapReduce Implementations of Database Operators

- `Select` and `Project` can be easily implemented in the map function
- Aggregation is not difficult (see next slide)
- `Join` requires more work



Aggregation



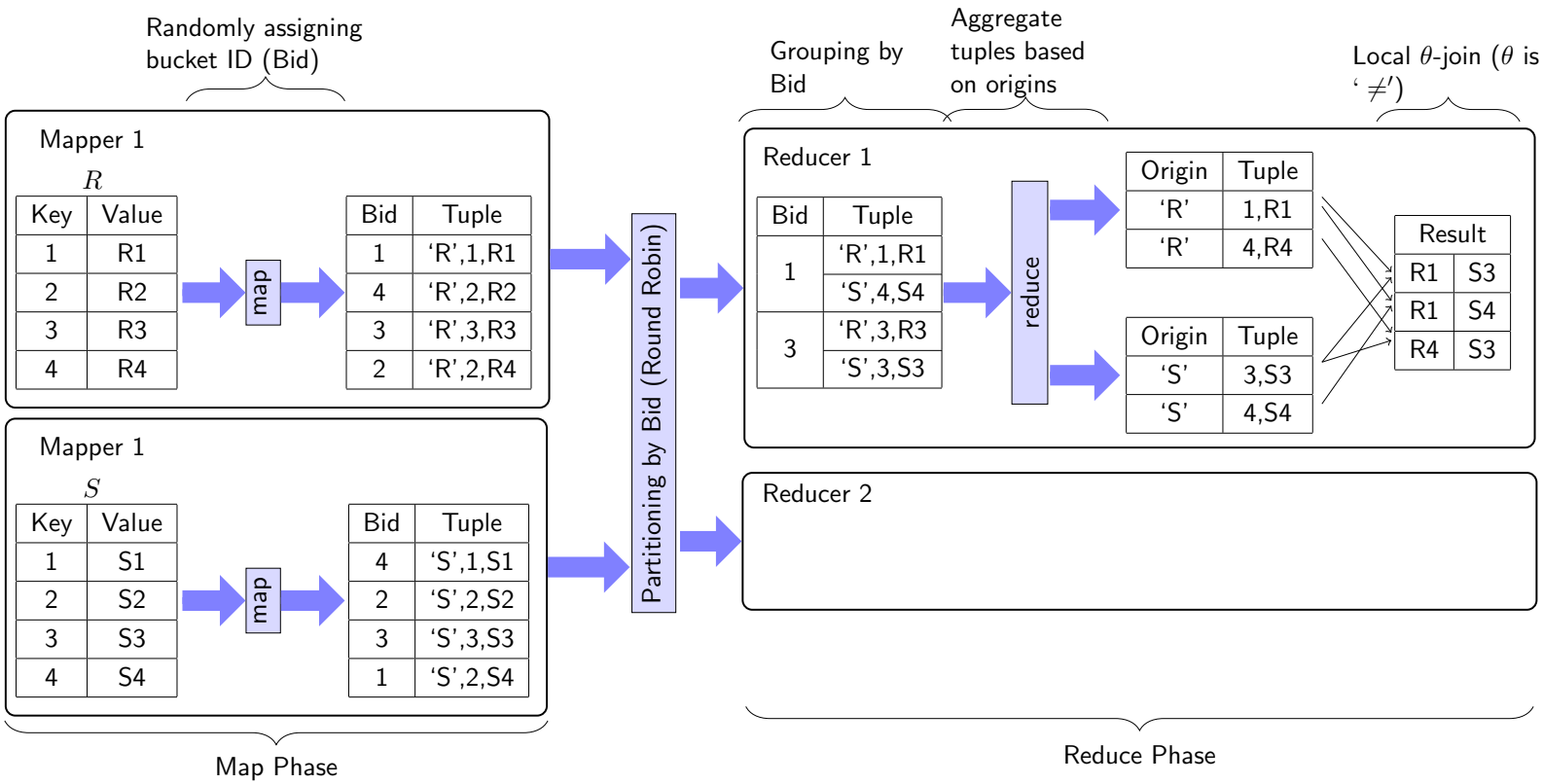
θ-Join

Baseline implementation of $R(A, B) \bowtie S(B, C)$

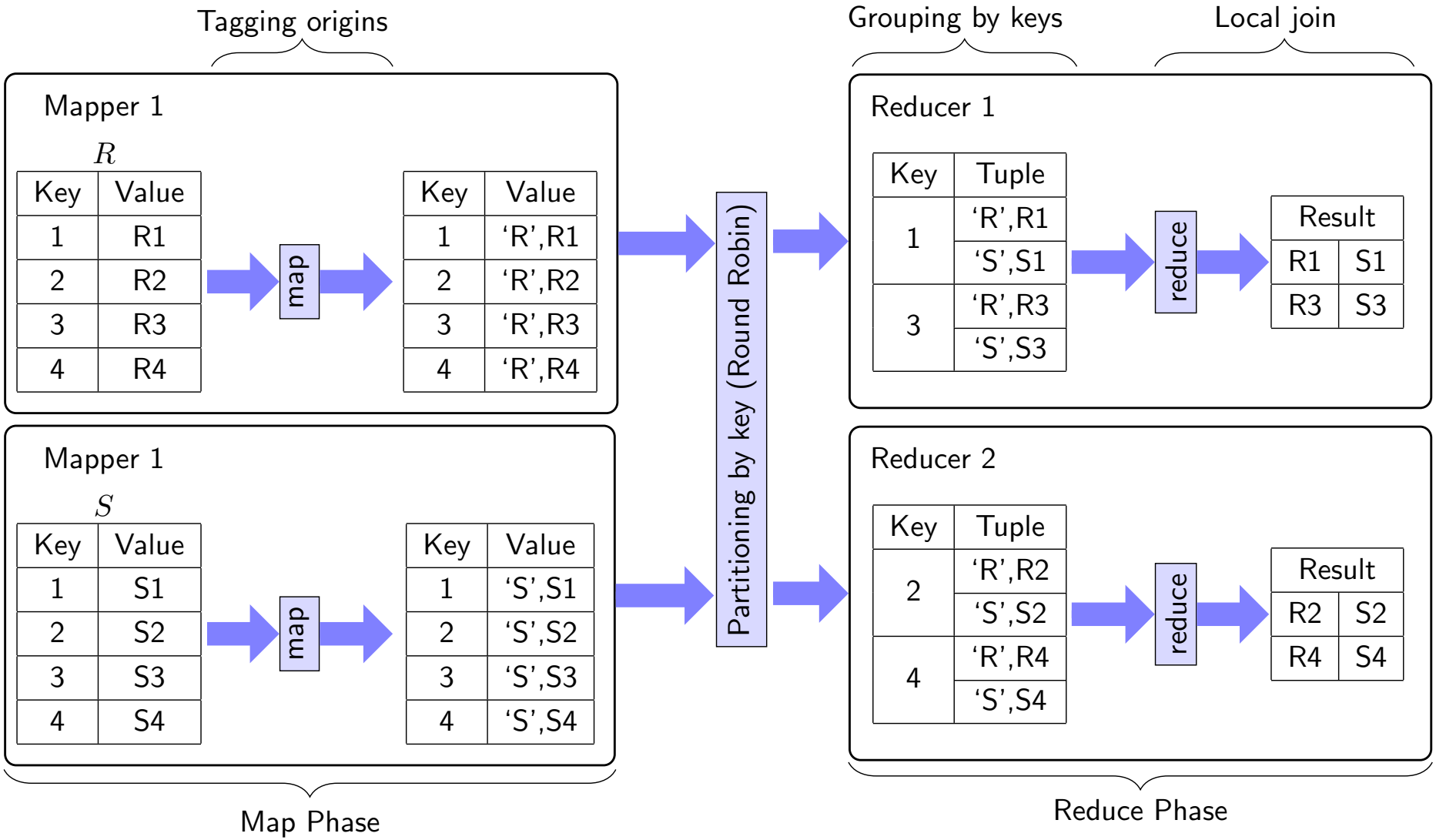
1. Partition R and assign each partition to mappers
2. Each mapper takes $\langle a, b \rangle$ tuples and converts them to a list of key/value pairs of the form $(b, \langle a, R \rangle)$
3. Each reducer pulls the pairs with the same key
4. Each reducer joins tuples of R with tuples of S

θ-Join

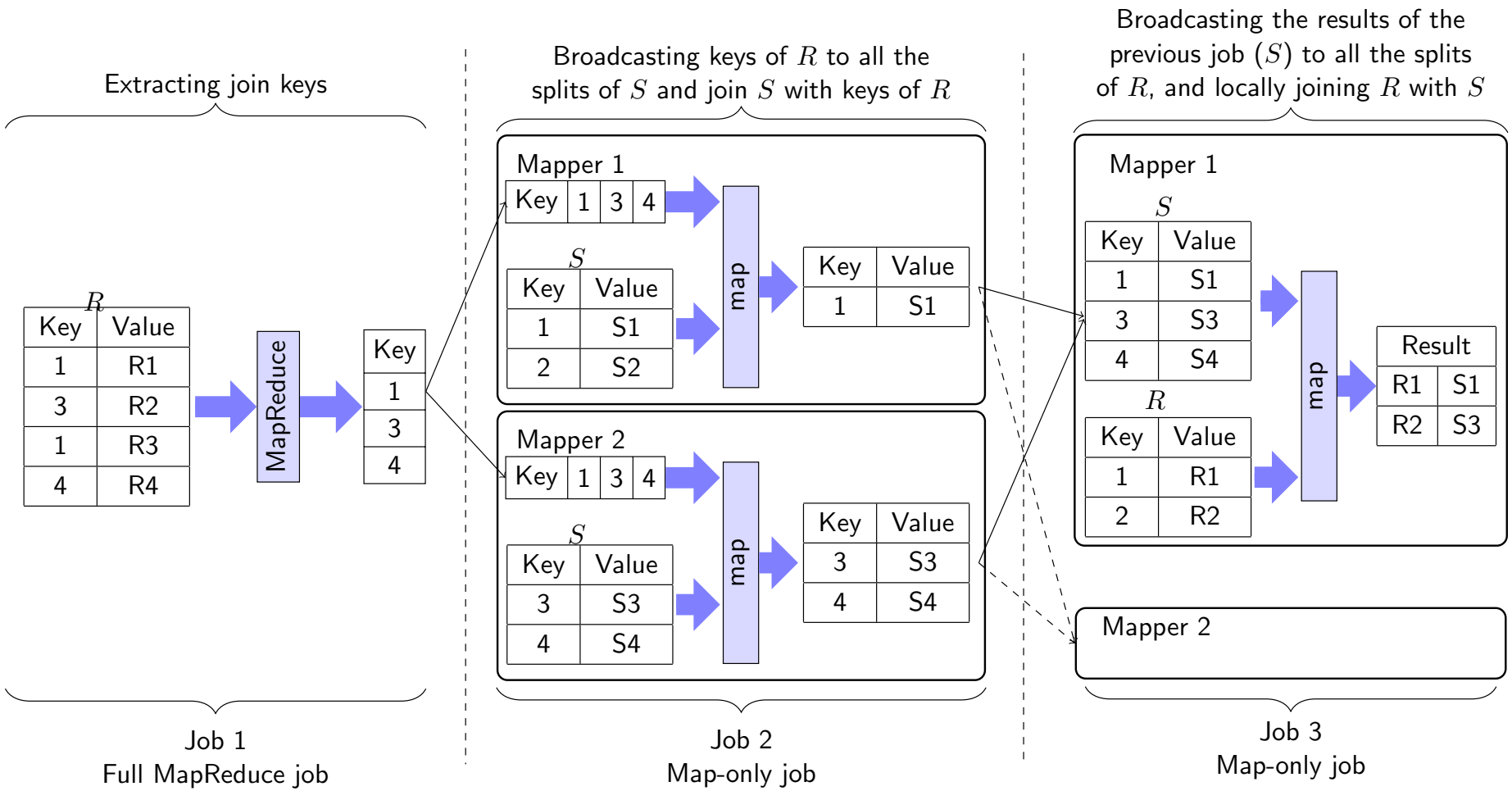
- If θ equals $=$ (i.e., equijoin)
  - Repartition join
  - Semijoin-based join
  - Map-only join
- If θ equals $\neq$



Repartition Join

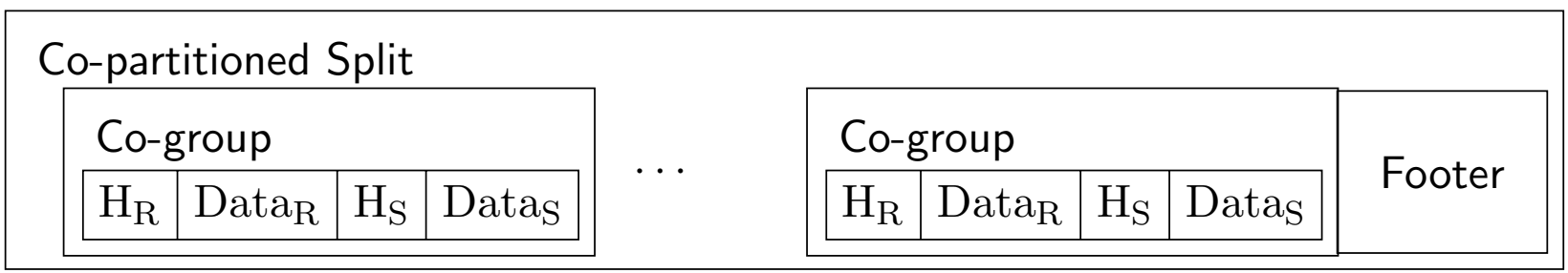


Semijoin-based Join



Map-only Join

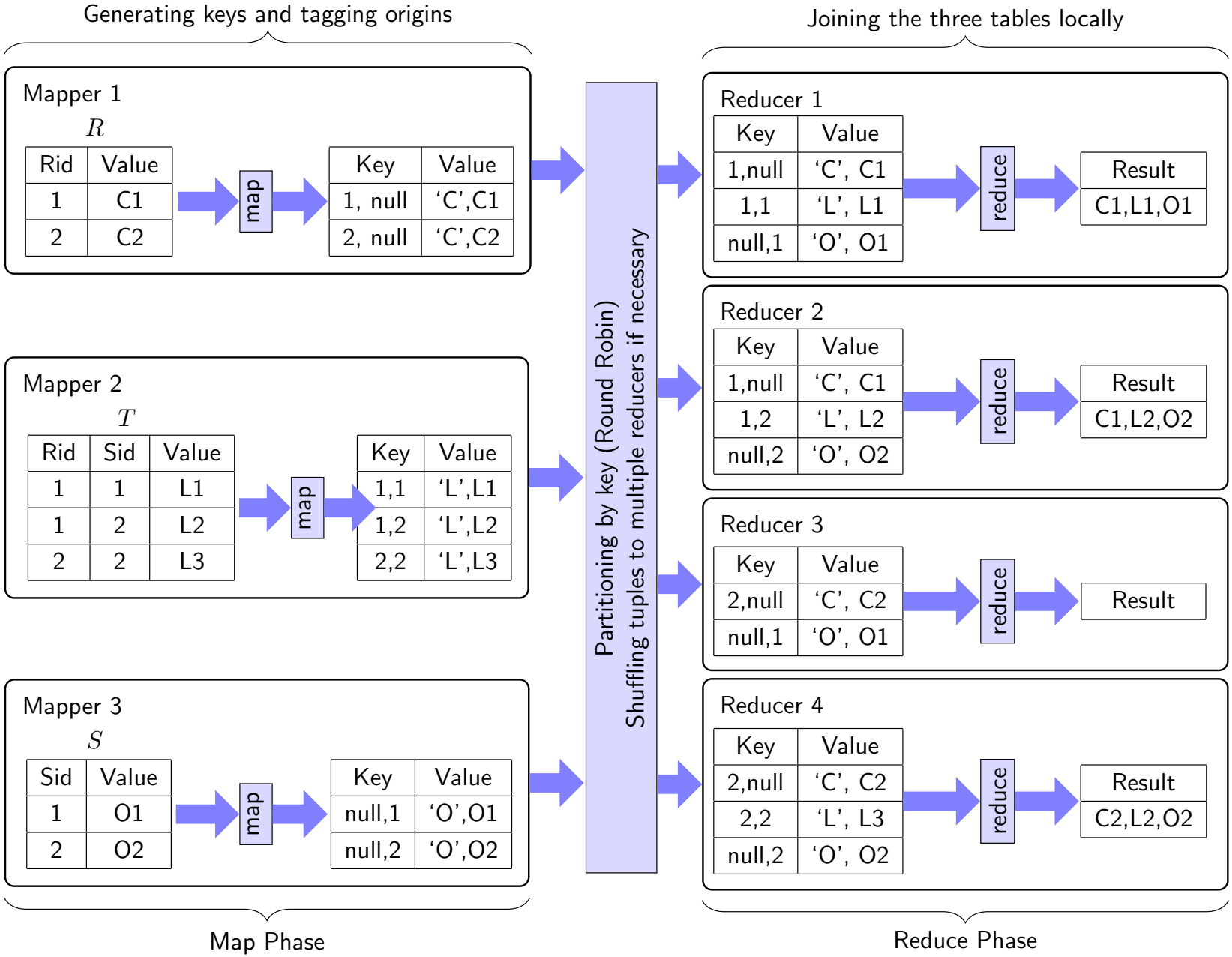
- Broadcast join: If inner relation $\ll$ outer relation $\rightarrow$ no shuffling
  - Map phase similar to third job of semijoin-based join
  - Each mapper loads the full inner table to build an in-memory hash; scan the outer relation
- Trojan join: Relations are already co-partitioned on the join key $\rightarrow$ all tuples of both relations are co-located on the same node
  - Co-partitioning implemented by one job
  - Scheduler loads co-partitioned data chunks in the same mapper to perform a local join



Multiway Join

- Multiple MapReduce jobs
  - $R \bowtie S \bowtie T \rightarrow (R \bowtie S) \bowtie T$
  - Each join implemented in one MapReduce job
  - Join ordering problem
- Replicated join
  - Single MapReduce job

Replicated Join



DBMS on MapReduce

	HadoopDB	Llama	Cheetah
Language	SQL-like	Simple interface	SQL
Storage	Row store	Column store	Hybrid store
Data compression	No	Yes	Yes
Data partition	Horizontally partitioned	Vertically partitioned	Horizontally partitioned at chunk level
Indexing	Local index in each database instance	No index	Local index for each data chunk
Query optimization	Rule based optimization plus local optimization by PostgreSQL	Column-based optimization, late materialization and processing multiway join in one job	Multi-query optimization, materialized views