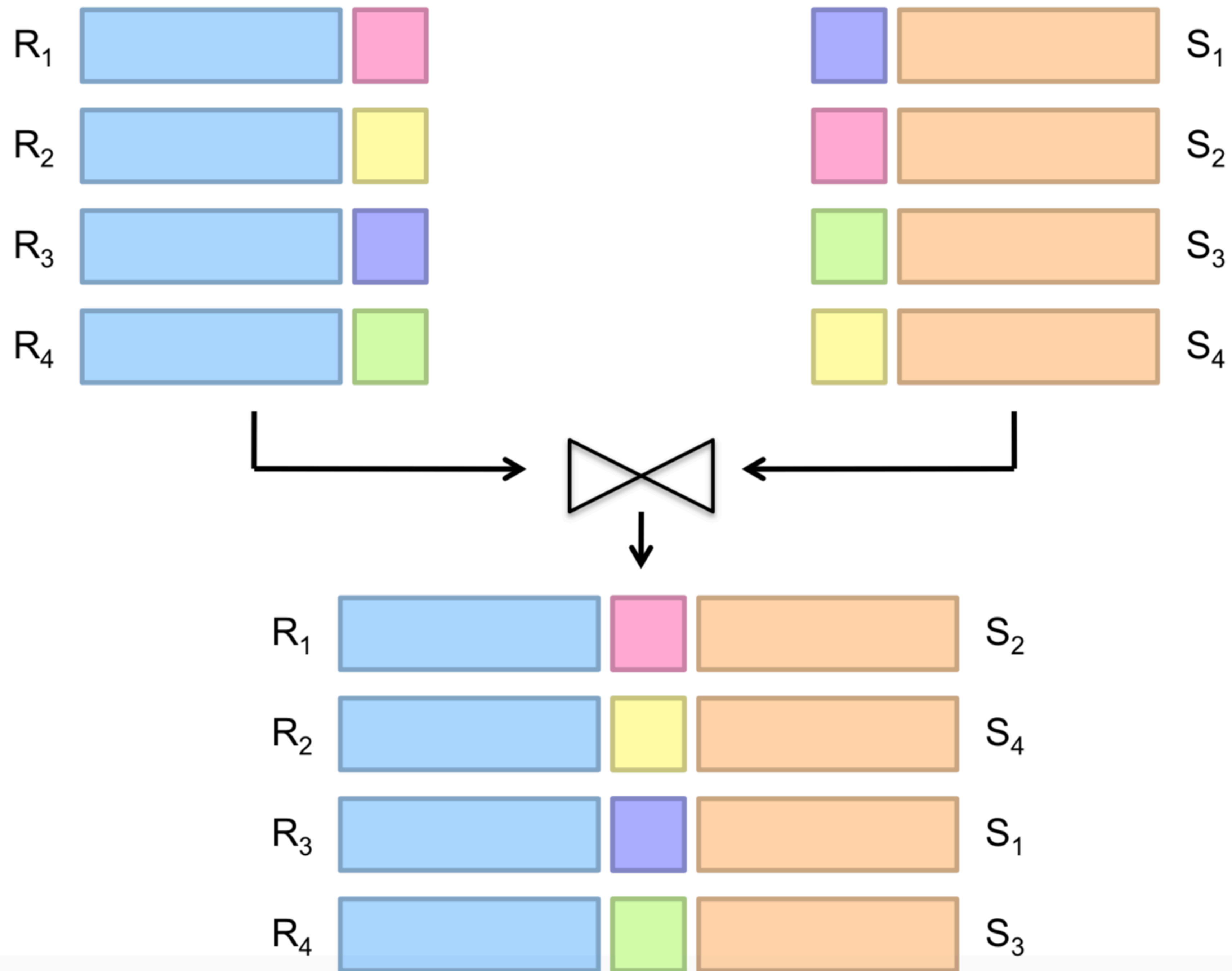
Relational Algebra & MapReduce 2

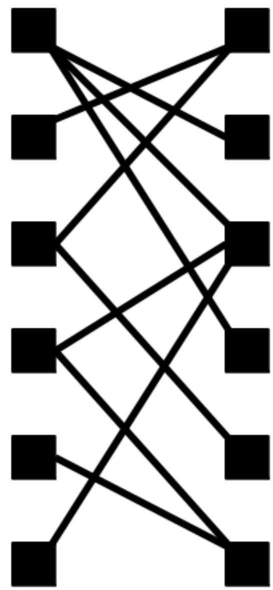
Relational Algebra

- Primitives
  - Projection (π)
  - Selection (σ)
  - Cartesian product ($\times$)
  - Set union ($\cup$)
  - Set difference ($-$)
  - Rename (ρ)
- Other operations
  - Join ($\bowtie$)
  - Group by… aggregation

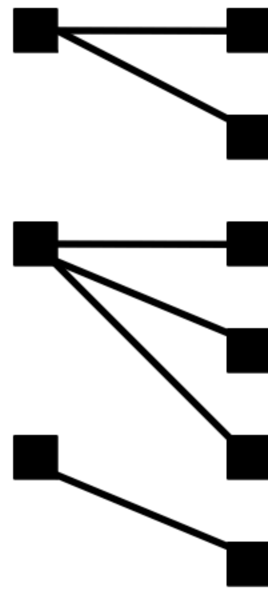
Relational Join



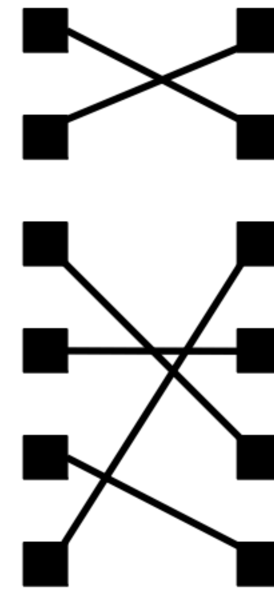
Types of Relations



Many-to-Many



One-to-Many



One-to-One

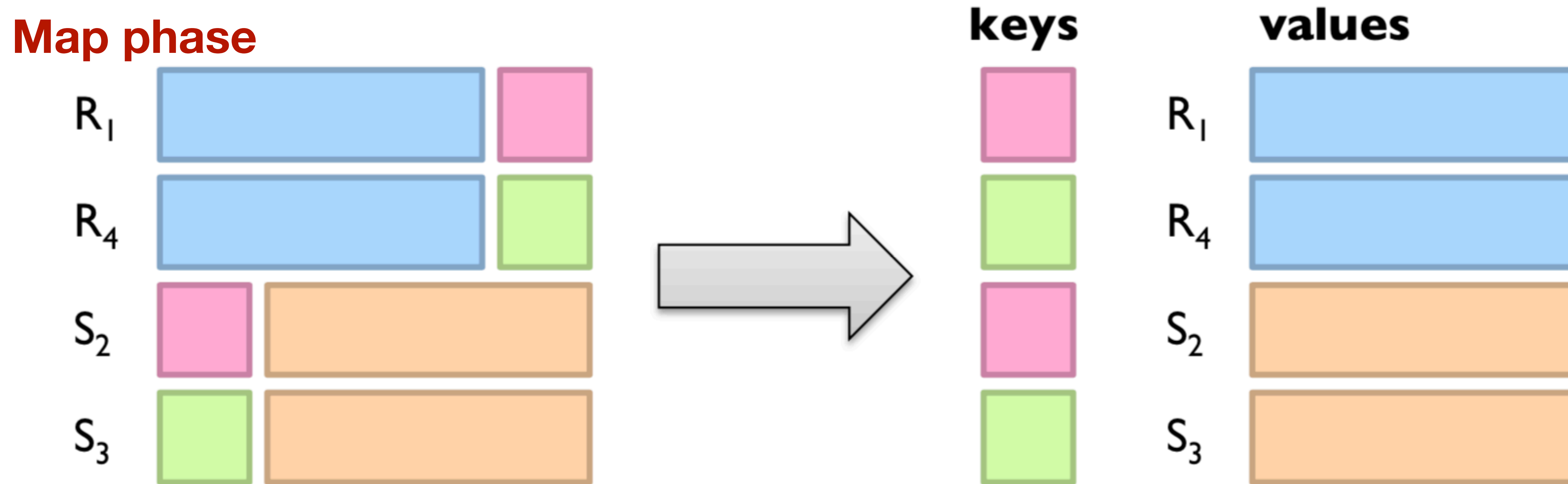
Join in MapReduce

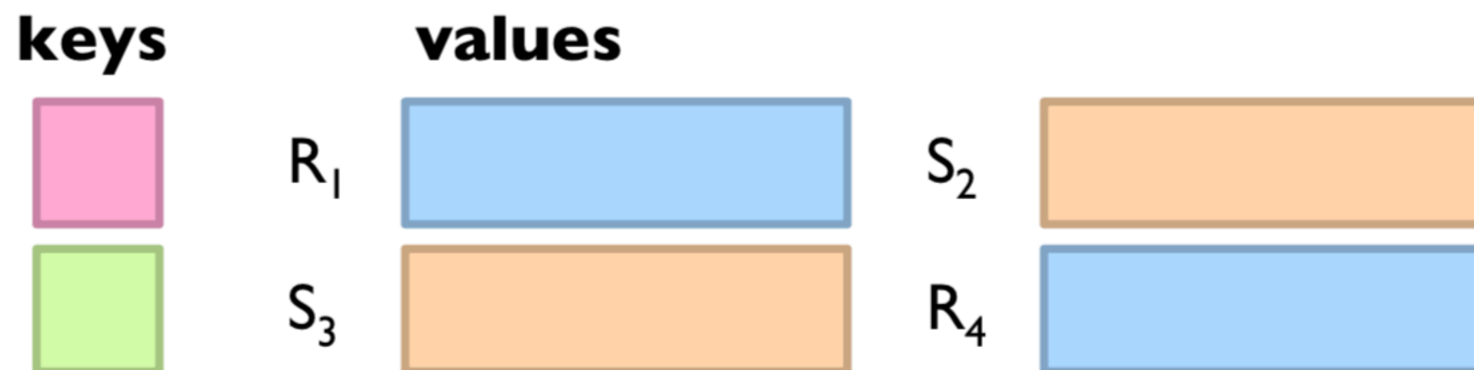
- Reduce-side join
  - join is done by the reducers
- Map-side join
  - join is done my the mappers

Reduce-side Join

- Key idea: group by join key
- Mapper:
  - read tuples from two datasets/directories
  - emit key-values as:
    - tuple plus tag to indicate from which dataset the record came as value (tag, tuple)
    - join attribute as key
  - framework group tuples that share the same key
- Reducer:
  - all values from the two datasets with the same join key arrives at the same reducer
  - perform the join

Reduce-side Join
1-to-1





* order of values is not guaranteed, use the tag to check the origin

Example

- given the following datasets, we want to know for each customer: name, amount, date

customer

Cust ID	First Name	Last Name
4000001	Kristina	Chung

transactions

Trans ID	Date	Cust ID	Amount
0000000	06-26-2011	4000001	40.33

Example

- given the following datasets, we want to know for each customer: name, amount, date
- solution:
  - Map phase:
    - we need mapper for each dataset; for example CustomerMapper, TransactionMapper
    - emit /context.write()
      - the needed attributes with relation name (customer or transactions) as value
      - the join key (custID) as the key
  - reduce phase
    - receives all tuples of the same join key
  - job conf
    - add the two mappers

```
MultipleInputs.addInputPath(job, new Path(args[0]),TextInputFormat.class, CustomerMapper.class);
MultipleInputs.addInputPath(job, new Path(args[1]),TextInputFormat.class, TransactionMapper.class);
```

customer

Cust ID	First Name	Last Name
4000001	Kristina	Chung

transactions

Trans ID	Date	Cust ID	Amount
0000000	06-26-2011	4000001	40.33

References

- Data-Intensive Text Processing with MapReduce
  - pdf available on https://lintool.github.io/MapReduceAlgorithms/MapReduce-book-final.pdf
  - Reduce-side join full example
    - https://www.edureka.co/blog/mapreduce-example-reduce-side-join/

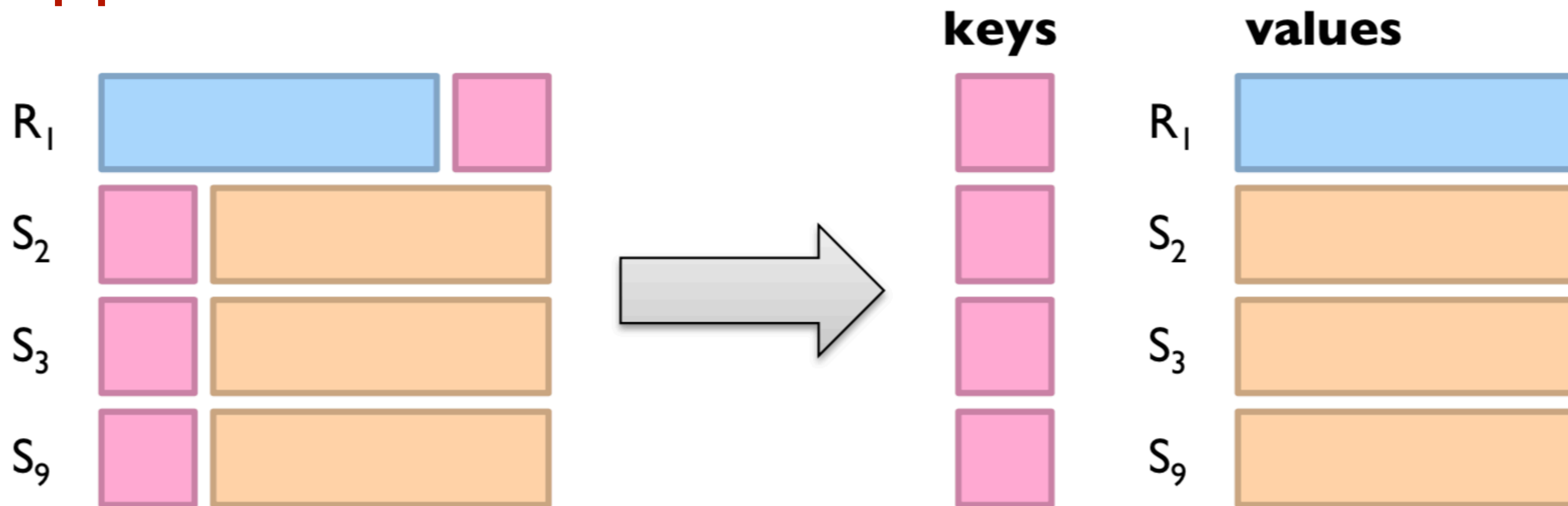
Reduce-side Join
1-to-1

- Map phase
  - the map reads the tuple from one of the two datasets
  - emit key-value pair
    - where key is the join key
    - value consists of
      1. the tuple (row), it can be the whole row or part of it
      2. with a tag to indicate from which dataset it originates
    - for example from dataset S; key-value -> (joinKey, “S” + “,” + tuple)

Reduce-side Join
1-to-1

- Reduce phase
  - reducer receive all values for the same key
  - the framework guarantees that all values of the same key goes to the same reducer
  - but the order of values is not guaranteed
  - for the 1-to-1 join the reducer will receive two values for the join key
    - one from each dataset
  - reducer can read the two values, keep them in memory
  - the value consist of the tuple from the dataset plus a tag indicating from which dataset the value originates
    - for example, (“S”, tuple) indicates that this tuple is from S

Reduce-side Join
1-to-many



Reduce-side Join 1-to-many

- Map phase is the same as we have in the Reduce-side join 1-to-1
- Reduce phase
  - we need to cross the 1 tuple with many tuples
    - but remember that the order of values is not guaranteed
  - if we are joining tuples from relation **R** (one) with relation **S** (many)
  - we need to cross the one value from **R** with all tuples from **S** that have the same join key and we don't know when the value from **R** comes

First Solution

- buffer every thing in memory
  - pick the value from **R** and hold it in memory
    - each value is tagged with dataset name (either **S** or **R**)
  - cross it with all values from **S**
- This solution might cause memory error if there is no enough memory at the reducer node
- This problem requires a secondary sort of values

Second Solution

- Instead of making the reducer does the sort
  - which might cause out of memory problem
- We utilize the framework to do the sorting
  - the framework is already doing the sort based on the key
  - we update it to do a secondary sort as well for the values
- In the mappers: Move part of the value to the key in order to form a composite key and let the framework handle the sort. is known as value-to-key conversion design pattern

Secondary Sort (Map update)

- Modify the mapper:
  - instead of emitting the the join key as the intermediate key
  - emit a composite key: join key and tuple id (whether it is from S or R relation)

Secondary Sort (framework)

- We need two modifications:

1. Sort:

    - define the sort order to be first based on the join key,
    - and then sort tuple from **R** to be before tuples from **S**

Secondary Sort (framework)

- We need two modifications:

  2. Partitioner:

     - must pay attention to the join key only to make sure that all composite keys with same join key end up in the same reducer

       - otherwise if partitioner is based on the composite key, then data from the two relation with same join key might end up in different reducers

Secondary Sort (Reduce update)

- now when the reducer read a new key, it is guaranteed that the first value is from **R**, thus
  - reducer holds this value in memory
  - cross it with other values from **S**

in Reducer

keys **values**

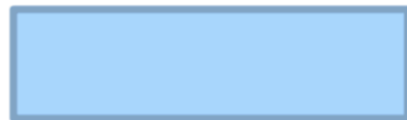
R_1 New key encountered: hold in memory

Cross with records from other dataset

S_2

S_3

S_9

R_4  New key encountered: hold in memory

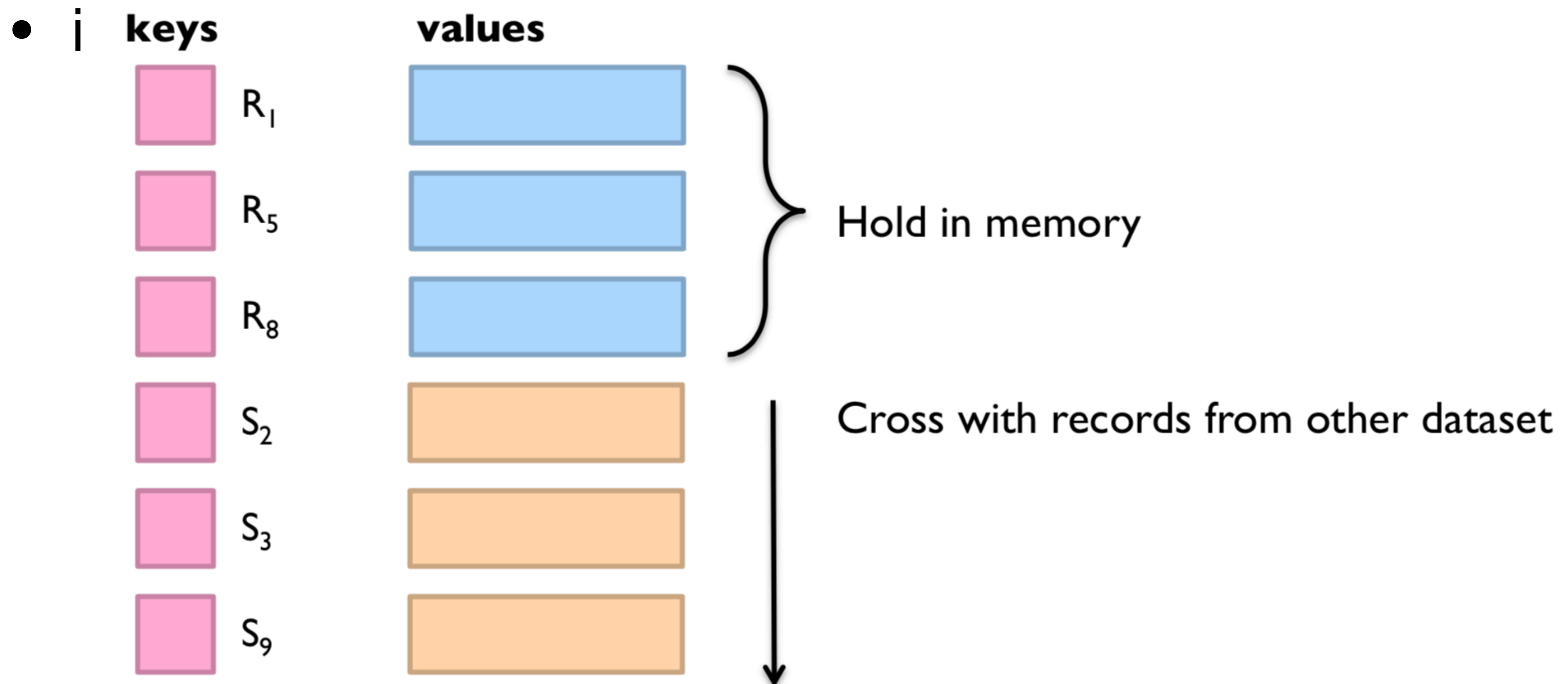
Cross with records from other dataset

S_3

S_7

Reduce-side Join many-to-many

- same idea
- j



Limitations

- The idea of holding values from S in memory will work, assuming that tuples from S can fit in memory

Map-side Join

- known as sort-merge join