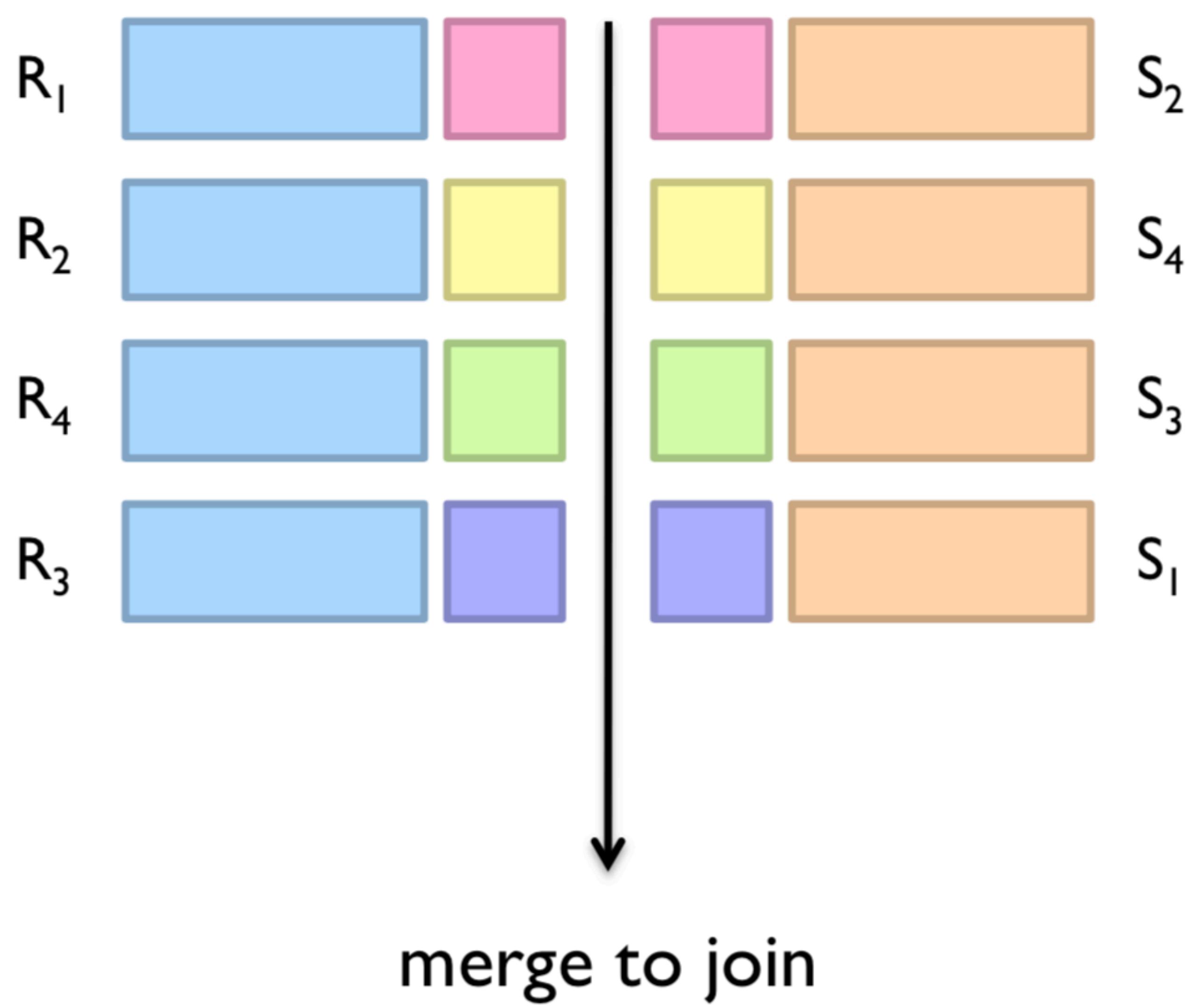
Relation Algebra In MapReduce 3

Map-side Join

- known as sort-merge join



Map-side Join

- Works if
  - the two datasets are partitioned in the same way using the join key
  - each partition is sorted by the join key
- For example if we have two datasets R & S, if
  - both are divided into n files, partitioned & sorted
  - then, we simply join first partition from S with first partition from R

Map-side Join

- MapReduce implementation
  - map over one dataset (the larger one), read from other corresponding partition which is in memory
  - In Hadoop MapReduce there is something called Distributed Cache
    - if you have one small file then you can ship it with the MapReduce code
    - it will be copied in every node running map task
    - the node will load this file in memory so it will be available when needed

Difference (-)

- To get the difference between two relations
- The two relations must have the same schema
  - same attributes
- R - S -> tuples in R but not in S

Difference in MapReduce

- Two relations R and S, we want to do R - S
- Mappers
  - for each tuple *t* emit a key-value pair
    - key is the tuple itself *t*
    - value is a tag indicating the dataset containing the tuple; if it is from relation R then value = “R”
  - so key-value is (t,”R”) or (t,”S”)
- Reducers:
  - get all values related to the same tuple *t* which could be one or two
  - key t might have the following possible values [“R”,”S”], [“S”,”R”] [“S”] or [“R”]
    - if *t* is associated with [“R”], then emit (t , t)
    - if *t* is associated with [“R”,”S”], [“S”,”R”] [“S”], then ignore it

Union (∪)

- Union between two relation will result in a relation that has rows from either of them or both (no duplicates)
- The two relations must have the same schema; same attributes

Union in MapReduce

- R ∪ S
- Mappers:
  - for each tuple *t* from ’S’ or ‘R’
  - emit key-value pairs: (*t, t*)
- Reducers:
  - for each key t, reducer will get either one or two values
    - one value: if either R or S has the tuple
    - two values: if both relations has the tuple
    - in either case, reducer emit (t , t) once

Intersection (∩)

- Intersection between two relations R ∩ S will result in having a relation that contains tuples which exist in both
    - both must have the same schema

Intersection in MapReduce

- $R \cap S$
- Mappers:
  - for each tuple t emit key-value pair (t, t)
  - it does not matter from which relation the tuple comes
- Reducers
  - reducer will get for each key *t* one or two values
  - if key has list of values (2), then we know both relations have the tuple
    - emit (t, t)
  - if the key has one value then ignore it

Summary

- MapReduce algorithms for processing relational data:
  - Group by, sorting, partitioning are handled automatically by MapReduce framework
  - Selection, projection, and other computations (e.g., AVG, MIN, …), are performed either in mapper or reducer
  - Multiple strategies for relational joins
    - Reduce-side join
    - Map-Side

Need for Higher-level Language

- Complex operations require multiple MapReduce jobs

  - Example: top ten URLs in terms of average time spent
    - Input data: url, user, spent time
    - We need two MapReduce jobs
      - one for computing the average,
      - a second one reads the output from the first job and get the top 10

- Might require to write a lot of code and multiple jobs
- Therefore, we need higher-level language
  - for example Pig Latin: next

Computing Mean

- Find average of integers associated with the same key

  SELECT key, AVG(value) FROM r GROUP BY key;

  - input to the mapper is key-value pairs (key, value)
    - for example: (key1, 2) , (key1, 4)
  - final output should be (key , average(values))
    - mean = sum / cnt = (2 + 4)/2
    - -> (key1 , 3)

Implementing Mean in MapReduce

- There are three ways to compute the mean
  - with and without including the combiner
- form groups and find two possible implementation of the Mean
  - up to you what to do in Map, Reduce, combiner

Computing the Mean (v1)

```
class Mapper {
  def map(key: Text, value: Int, context: Context) = {
    context.write(key, value)
  }
}

class Reducer {
  def reduce(key: Text, values: Iterable[Int], context: Context) {
    for (value <- values) {
      sum += value
      cnt += 1
    }
    context.write(key, sum/cnt)
  }
}
```

Computing the Mean (v2)

```
class Mapper {
  def map(key: Text, value: Int, context: Context) =
    context.write(key, value)
}
class Combiner {
  def reduce(key: Text, values: Iterable[Int], context: Context) = {
    for (value <- values) {
      sum += value
      cnt += 1
    }                          Pair
    context.write(key, (sum, cnt))
  }
}
class Reducer {
  def reduce(key: Text, values: Iterable[Pair], context: Context) = {
    for (value <- values) {
      sum += value.left
      cnt += value.right
    }
    context.write(key, sum/cnt)
  }
}
```

Computing the Mean (v3)

```
class Mapper {
  def map(key: Text, value: Int, context: Context) =
    context.write(key, (value, 1))
}
class Combiner {
  def reduce(key: Text, values: Iterable[Pair], context: Context) = {
    for (value <- values) {
      sum += value.left
      cnt += value.right
    }
    context.write(key, (sum, cnt))
  }
}
class Reducer {
  def reduce(key: Text, values: Iterable[Pair], context: Context) = {
    for (value <- values) {
      sum += value.left
      cnt += value.right
    }
    context.write(key, sum/cnt)
  }
}
```

Need for Higher-level Language

- Complex operations require multiple MapReduce jobs
  - Example: top ten URLs in terms of average time spent
  - MapReduce job for computing the average, a second one reads the output from the first job and sort and get the top 10
- Might require to write a lot of code and multiple jobs
- Therefore, we need higher-level language
  - for example Pig Latin: next