Introduction

- After introducing several design patterns and basic data processing utilities in earlier modules, this module discusses more advanced approaches where information is combined across two input data sets.
- In particular, you will learn about the join operator and about Bloom filters.

1. Joins in MapReduce

- The relational join operator combines records from one data set with records in another. It outputs all record pairs that satisfy a user-provided join condition.
- This makes the join immensely useful for many data analysis problems:
  - In databases, joins are used to connect information stored in different tables. The most well-known example is the join on *foreign keys* to combine information in a normalized database. They can be used for the same purpose when analyzing big *files*, e.g., to find pairs of blog posts and tweets by the same user or from the same user community.
  - Joins can be used to analyze correlations and relationships between entities in different data sets.
  - They are also applicable to graph problems. For instance, given a file containing *direct* flights, the join operator can derive more complex flight connections, e.g., with up to two intermediate stops.
- In this module, the focus is on *equi-joins*. They are probably the most common type of join in practice. In fact, in the literature the term “join” is often used to implicitly mean “equi-join.” More general joins, called *theta-joins*, will be discussed in a future module.

1.1 Equi-Joins

- Let's introduce the equi-join more formally. Assume we are given two data sets $S=(s_1,s_2,\ldots)$ and $T=(t_1,t_2,\ldots)$. Using database terminology, we refer to the records in S and T as *tuples*. Assume that each tuple has an attribute A.
- The equi-join between S and T on attribute A is the set of all pairs (s_i,t_j), such that $s_i \in S$, $t_j \in T$, and $s_i.A=t_j.A$.
  - Property $s_i.A=t_j.A$ is the *join condition*. When the join condition enforces equality between attribute values, the join is called an equi-join. This idea generalizes to more complex equi-joins such as "s.userName = t.person AND s.age = t.age."
- For a concrete example, consider tables of students and book reservations made by these students. To analyze the GPA distribution of students who reserve certain books, these two tables first need to be joined on the SID attribute, which is a unique identifier for the students. (This is an example for a join on a foreign key. SID is a unique identifier, called a key, in the Students table. In the Reservations table, SID serves as a foreign key, "pointing" to the uniquely identified student who reserved the book.)
  - We use $\bowtie$ to express a join operator. For equi-joins it suffices to annotate this operator with the attribute(s) the tables are joined on.

SID	Name	Age	GPA
1	Alice	18	3.5
2	Bob	27	3.4
3	Carla	20	3.8

Students

Reservations

SID	BookID	Date
2	B10	01/17/12
3	B11	01/18/12

Students$\bowtie_{SID}$Reservations

SID	Name	Age	GPA	BookID	Date
2	Bob	27	3.4	B10	01/17/12
3	Carla	20	3.8	B11	01/18/12

1.2 Equi-Join in MapReduce: Reduce-Side Approach

- Let’s first look at the “natural” equi-join algorithm, which performs the actual joining of tuples in the Reducers. Later a specialized algorithm will be introduced for problems where one of the input sets is small.
- Consider S-tuple s, s.A = 1, and T-tuple t, t.A = 1. To produce output tuple (s, t), some task—Map or Reduce—has to receive both s and t. In general, the equi-join can be computed if all S-tuples with A=1 and all T-tuples with A=1 are processed together in the same task.
- This can be achieved for a Reduce task by using join attribute A as the intermediate key. More precisely, Map emits the input record with its A value as the key. Then the Reduce call for that A value receives all matching S- and T-tuples in its input list. To distinguish if a tuple came from S or T, Map adds a flag, “S” or “T”, respectively, to the value component. (Link: Illustration of the data flow during join computation.)
- Note that the Reduce function shown here loads all input values into memory. This is not necessary:
  - If S_list and T_list do not fit into the available memory, they could be written to local files on the Reducer machine. Instead of the nested for-each loops for pairing up the S- and T-tuples, a carefully designed file-I/O optimized algorithm could be used that pages parts of these files into and out of memory when needed.
  - One can also use the *secondary sort* design pattern to ensure that Reduce’s input list is sorted by the data origin flag, e.g., “S” tuples before “T” tuples. Then only one of the lists needs to fit into memory. The other can be accessed through the iterator provided by MapReduce for the Reduce input value list.

1.2 Equi-Join in MapReduce: Reduce-Side Approach

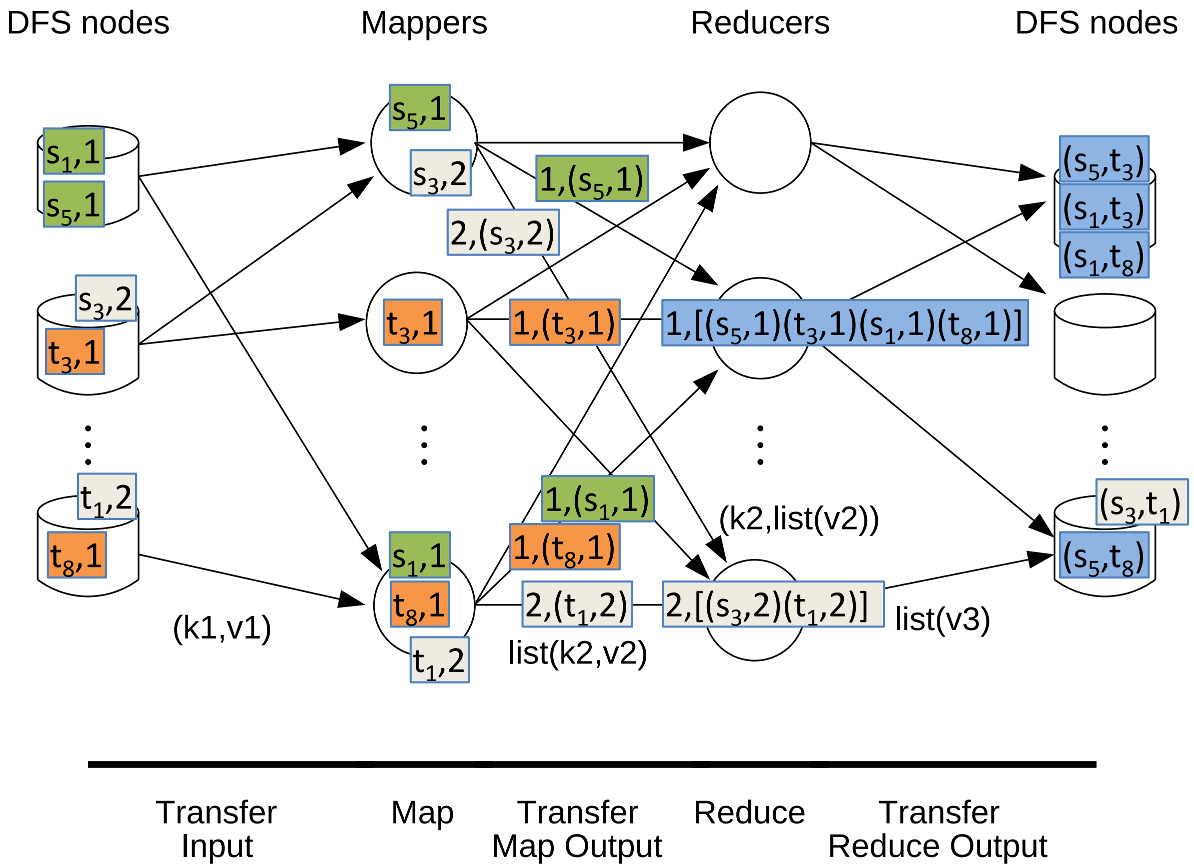
```
map( …, tuple x ) {
 if (x is from S)
  emit( x.A, (x, "S") )
 else    // x is from T
  emit( x.A, (x, "T") )
```

```
reduce( A-value, [(x1, flag1), (x2, flag2),…] ) {
 initialize S_list and T_list

 // Separate the input list by the data set the
 // tuples came from
 for all (x, flag) in input list do
  if (flag = "S")
   S_list.add( x )
  else
   T_list.add( x )

 // Since they have the same A value, each tuple
 // in the S_list "matches" with each
 // tuple in the T_list. Generate all these pairs.
 for each s in S_list
  for each t in T_list
   emit( NULL, (s, t) )
}
```

- In this example, each s_i and t_j is an S- or T-tuple, respectively. For clarity, the value of join attribute A is shown as well, e.g., (s_5,1) indicates s_5.A = 1.
- The input tuples are usually scattered over multiple file splits and might be assigned to different Mappers.
- Since Map emits each tuple with its A value as the key, in the example only two Reducers receive these records—one for key 1 and the other for key 2.
- The Reducers then write their output to files that are chunked up and assigned to distributed file system (DFS) nodes.



1.2.1 Implementation Details

- Usually the S-tuples and T-tuples are stored in separate files, in particular when S and T have different types or formats. This would cause a problem for Mappers as discussed so far: If there is only one Mapper class for a MapReduce job, how can it read files and parse tuples differently depending on it being a split of the file containing S-tuples or T-tuples?
- For cases where a MapReduce job has to process input files of different formats, Hadoop offers the MultipleInputs class in org.apache.hadoop.mapreduce.lib.input. It allows creation of different input paths and defining specialized Mappers for each—all in the same MapReduce job.
- This functionality can be exploited for the join problem:
  - Create two Mapper classes, Smapper and Tmapper. Smapper parses only S-tuples and adds "S" to the emitted value; Tmapper parses only T-tuples and adds "T" to the emitted value.
  - In the driver, both Mappers need to be set up with their appropriate input paths using commands like
    - MultipleInputs.addInputPath(job, new Path(args[0]),…, Smapper.class);
    - MultipleInputs.addInputPath(job, new Path(args[1]),…, Tmapper.class);

1.2.2 Discussion

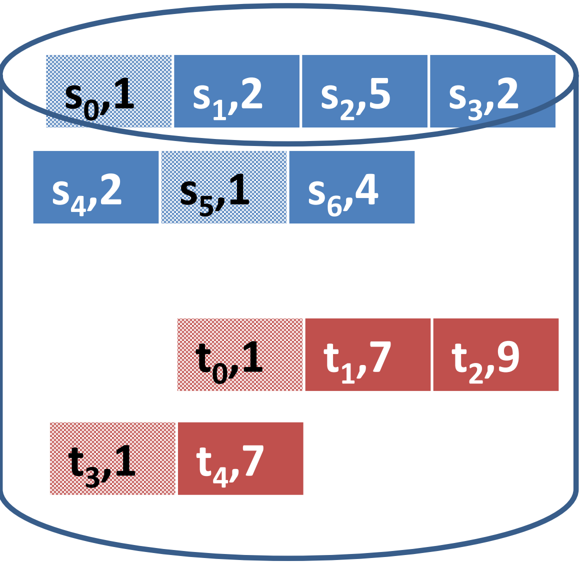
- A naïve join implementation could check for each possible pair (s, t), s∈S and t∈T, if s.A=t.A. This algorithm would explore |S|·|T| possible combinations, which is quadratic in the (already large) input size. In practice the join output size tends to be much smaller than |S|·|T|, hence this algorithm is generally considered inefficient.
- The Reduce-side join algorithm, like similar approaches from the relational database literature, avoids checking non-matching tuples. It groups the input by the join attribute (a linear-cost operation), then only combines tuples in the same group, which by design are matching.
- It might be hard to imagine that one could do any better than this. Unfortunately, there are several drawbacks:
  - The approach does not balance load well when the input is highly skewed. For instance, assume 50% of the S- and T-tuples have join attribute value A=1. The Reduce task responsible for A=1 would receive 50% of the entire input and would produce a huge fraction of the output.
  - The approach does not scale well for join attributes with small domains. For instance, assume we join based on gender and there are only two genders in the data: male and female. This means that Reduce-side work can be distributed over at most two Reduce tasks. No matter how many machines are available, all but two would be idle during the Reduce phase.
  - By using the join attribute(s) as the intermediate key, the algorithm is inherently limited to equi-joins. It does not generalize to non-equi joins, such as inequality conditions (S.A < T.A) or band-joins (|S.A – T.A| < ε).
  - Both S and T are completely read in the Map phase. Then the entire input is shuffled around and copied over to the Reducers, where it is read again.

1.3 Map-Only Join (“Replicated Join”)

- Let’s try to improve the Reduce-side join algorithm by starting with the last problem: the cost of transferring the entire input from Mappers to Reducers and then reading it again on the Reducers. A *Map-only* implementation would avoid this, but it might seem that joins need a Reduce phase. How else would one be able to guarantee that an S-tuple in some split of a big S-file would be joined with a matching T-tuple that could be in any split of the T-file? The illustration below shows an example. Each of the S-tuples with A=1 is in a different split, hence would be processed by a different Mapper. The matching T-tuples also occur in different splits.
- A Map-only join algorithm can be achieved by copying the entire data set T to every Mapper:
  - Send the entire T-file to all Mapper machines. This can be done most efficiently by using Hadoop’s distributed file cache (class DistributedCache).
  - The Map task’s setup function loads T from the file cache into a task-local data structure, e.g., a hash index on join attribute A.
  - The Map function processes only input set S (not T!). For each S-tuple, it probes the hash index to find all matching T-tuples.

1.3 Map-Only Join (“Replicated Join”)

Distributed File System



Splits of S-file

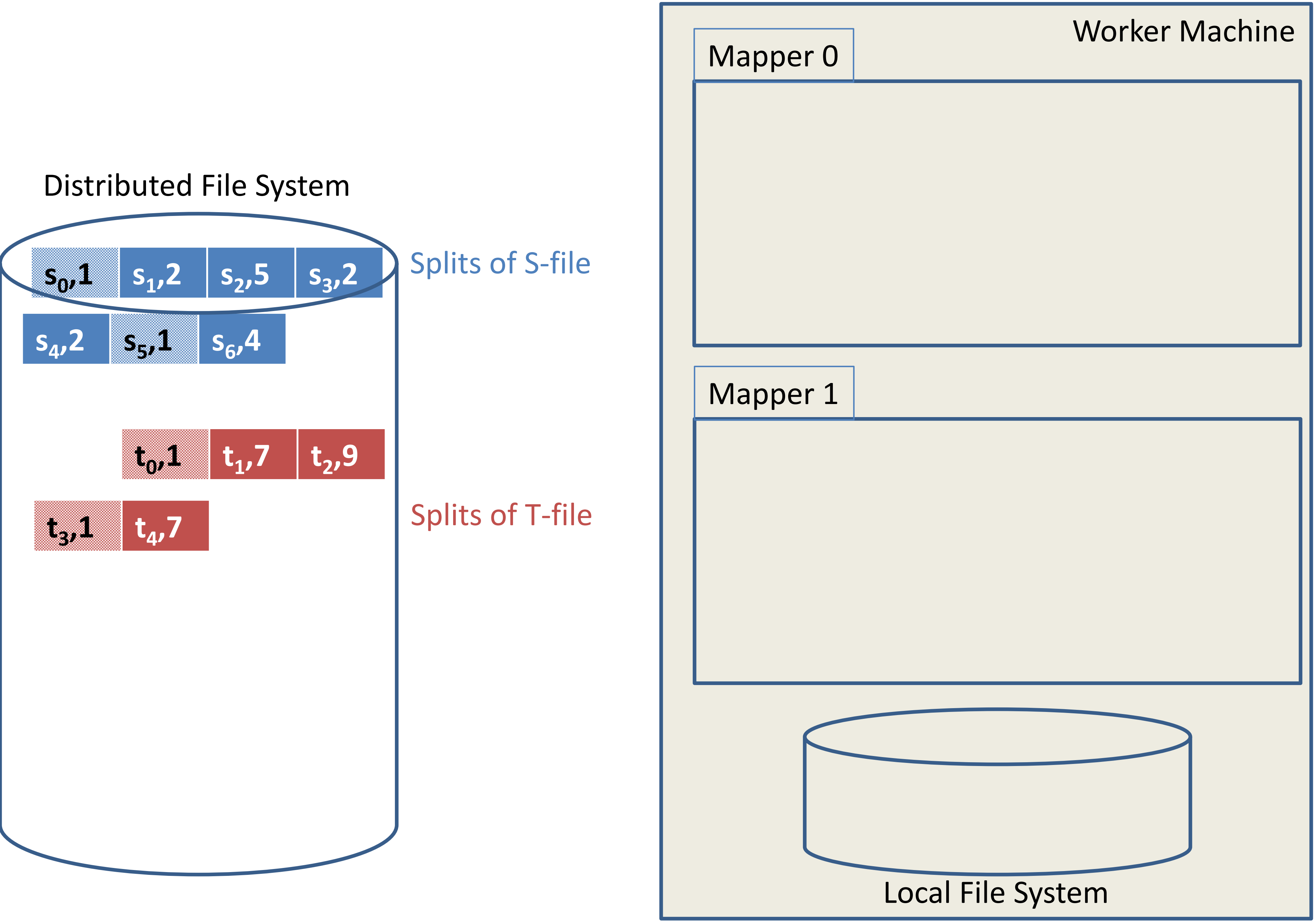
Splits of T-file

```
Class Mapper {
 // Index H maps a join attribute
 // value to all T-tuples with that value
 hashIndex H

 setup() {
   // Load data set T from the distributed file
   // cache into H, indexing on join attribute A.
   H = new hashMap
   for each tuple t in Distributed Cache
     H.insert( t.A, t )
 }

 map(…, S-tuple s ) {
   // The index lookup returns an iterator to
   // access all matching T-tuples in H.
   for each tuple t in H.lookup( s.A ) do
     emit( NULL, (s, t) )
 }

 cleanup() { clean up H }
}
```



Distributed File System

s_0,1 s_1,2 s_2,5 s_3,2

s_4,2 s_5,1 s_6,4

Splits of S-file

t_0,1 t_1,7 t_2,9

t_3,1 t_4,7

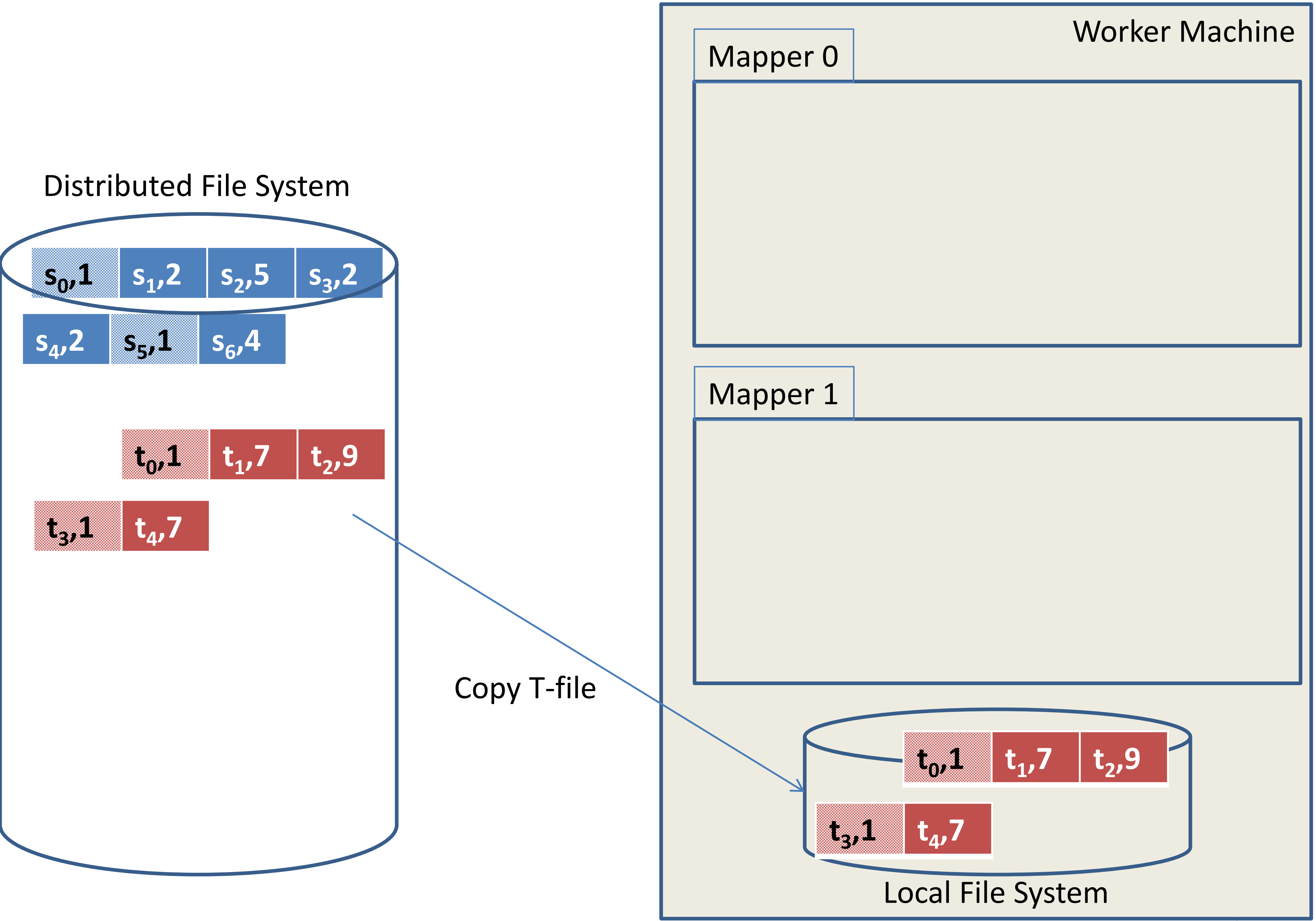
Splits of T-file

Worker Machine

Mapper 0

Mapper 1

Local File System



Distributed File System

s_0,1 s_1,2 s_2,5 s_3,2

s_4,2 s_5,1 s_6,4

t_0,1 t_1,7 t_2,9

t_3,1 t_4,7

Copy T-file

Worker Machine

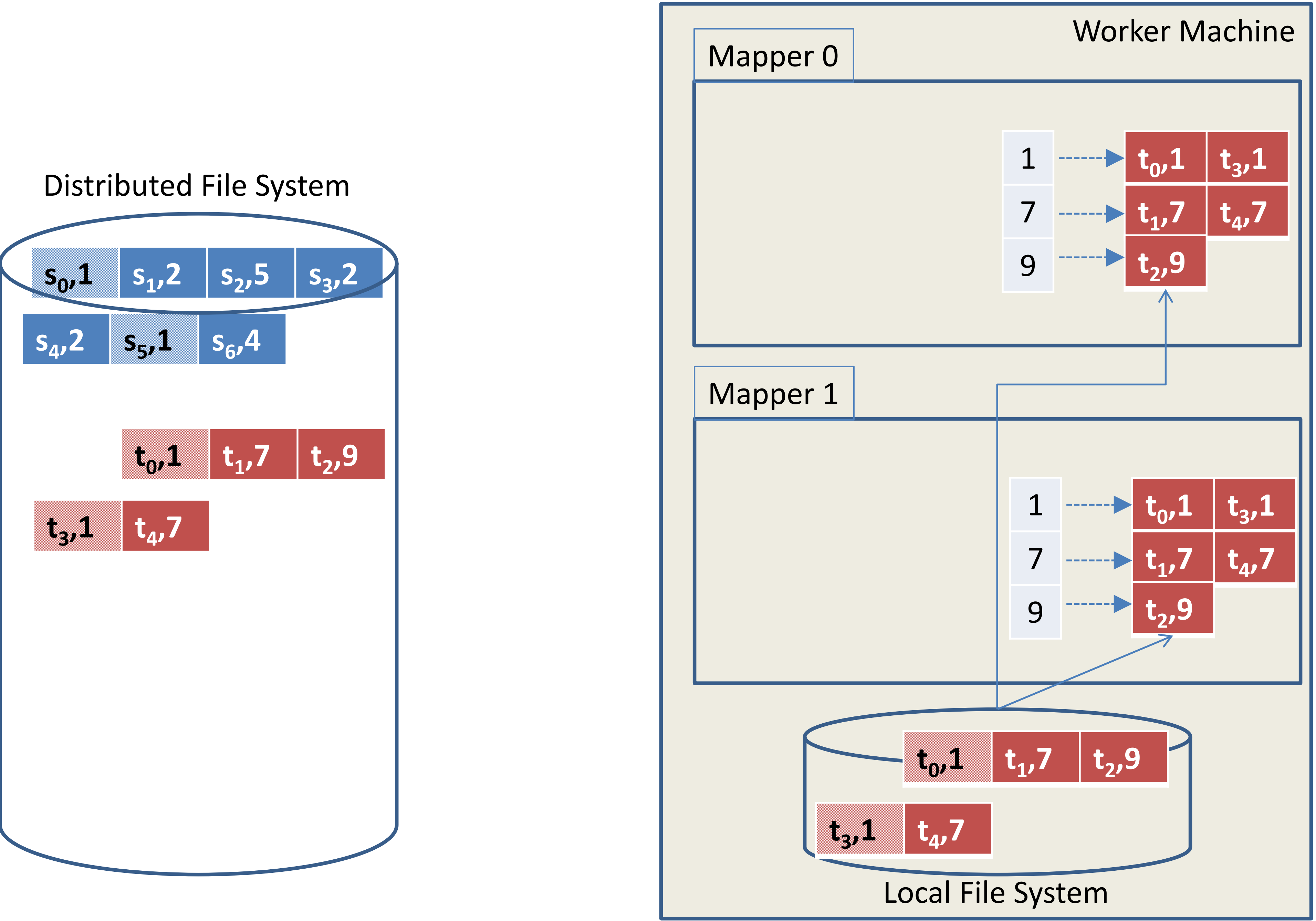
Mapper 0

Mapper 1

t_0,1 t_1,7 t_2,9

t_3,1 t_4,7

Local File System



Distributed File System

s_0,1 s_1,2 s_2,5 s_3,2

s_4,2 s_5,1 s_6,4

t_0,1 t_1,7 t_2,9

t_3,1 t_4,7

Worker Machine

Mapper 0

1 → t_0,1 t_3,1

7 → t_1,7 t_4,7

9 → t_2,9

Mapper 1

1 → t_0,1 t_3,1

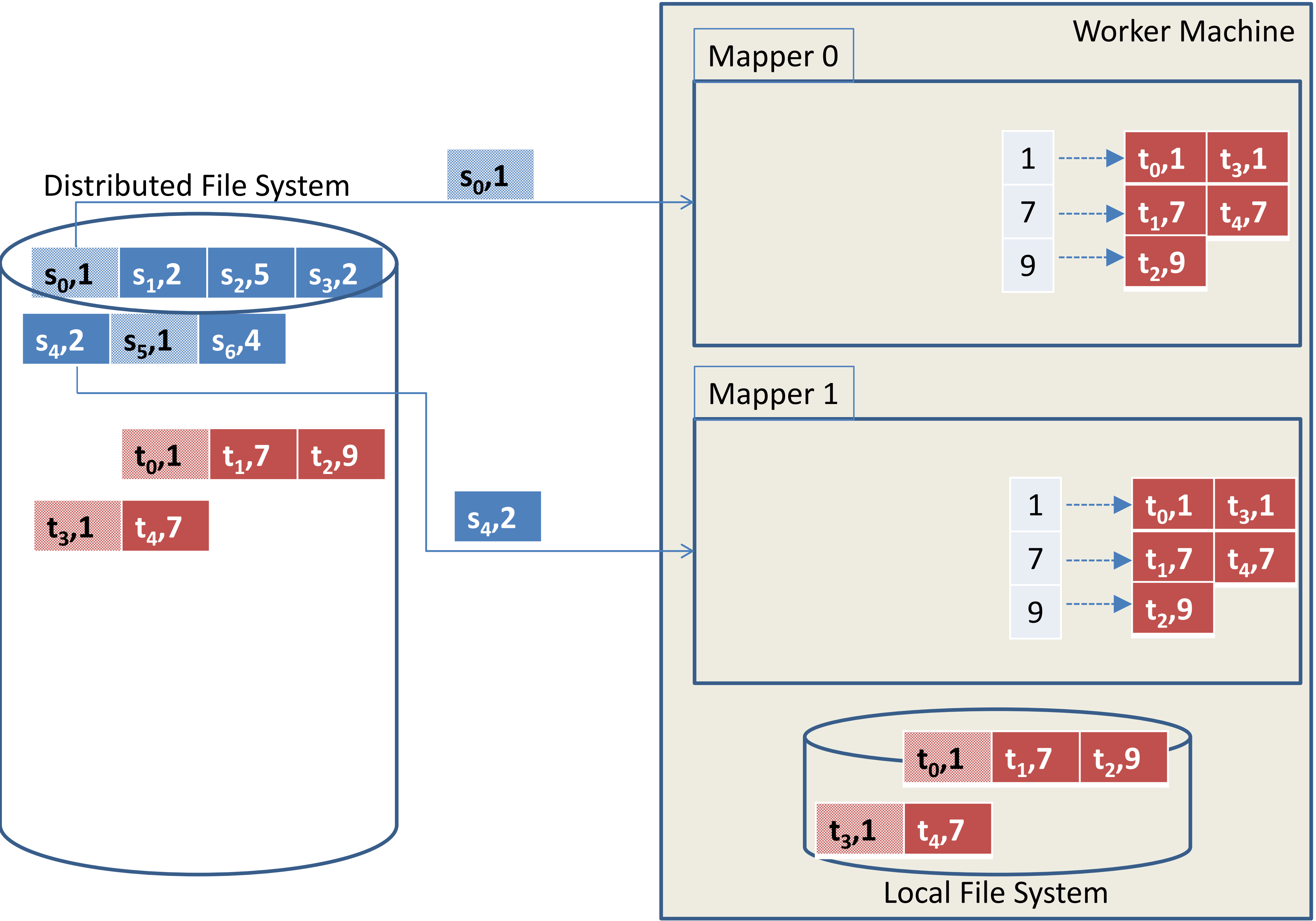
7 → t_1,7 t_4,7

9 → t_2,9

t_0,1 t_1,7 t_2,9

t_3,1 t_4,7

Local File System



Worker Machine

Mapper 0

1 → t_0,1 t_3,1

7 → t_1,7 t_4,7

9 → t_2,9

Distributed File System

s_0,1

s_0,1 s_1,2 s_2,5 s_3,2

s_4,2 s_5,1 s_6,4

Mapper 1

1 → t_0,1 t_3,1

7 → t_1,7 t_4,7

9 → t_2,9

t_0,1 t_1,7 t_2,9

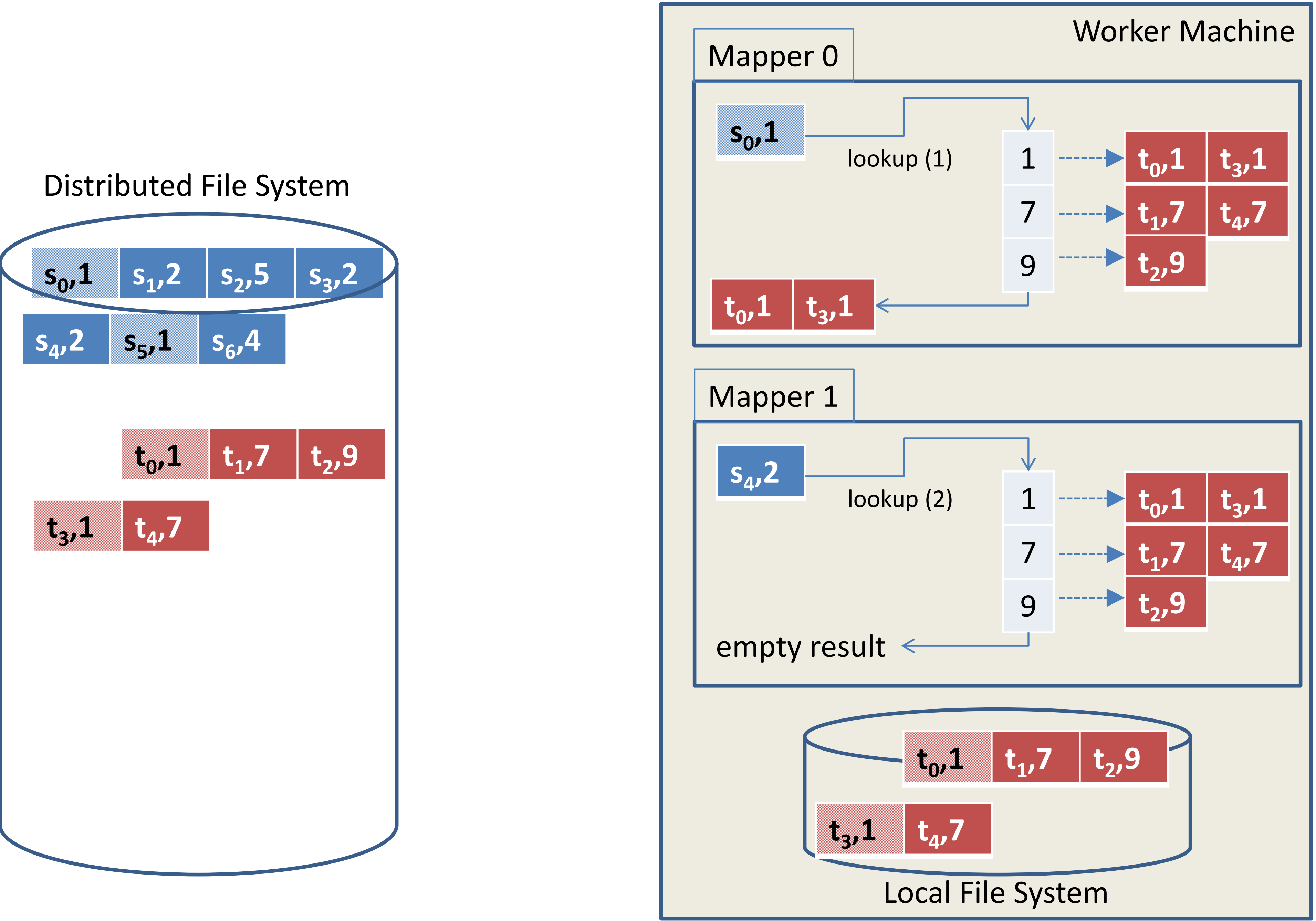
t_3,1 t_4,7

s_4,2

t_0,1 t_1,7 t_2,9

t_3,1 t_4,7

Local File System

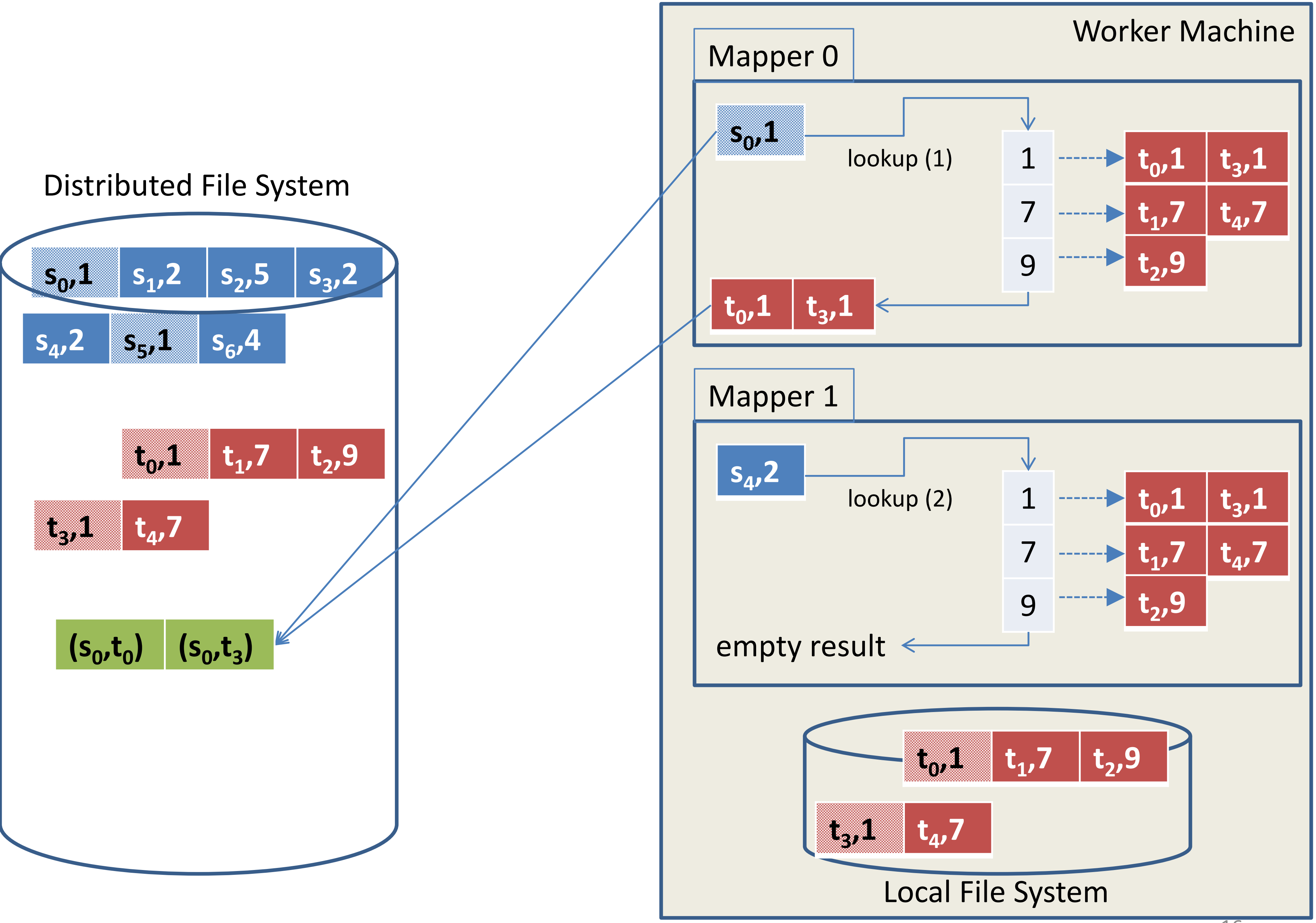

Distributed File System

Worker Machine

Mapper 0

Mapper 1

Local File System



Distributed File System

s_0,1 s_1,2 s_2,5 s_3,2

s_4,2 s_5,1 s_6,4

t_0,1 t_1,7 t_2,9

t_3,1 t_4,7

(s_0,t_0) (s_0,t_3)

Worker Machine

Mapper 0

s_0,1

lookup (1)

1 → t_0,1 t_3,1

7 → t_1,7 t_4,7

9 → t_2,9

t_0,1 t_3,1

Mapper 1

s_4,2

lookup (2)

1 → t_0,1 t_3,1

7 → t_1,7 t_4,7

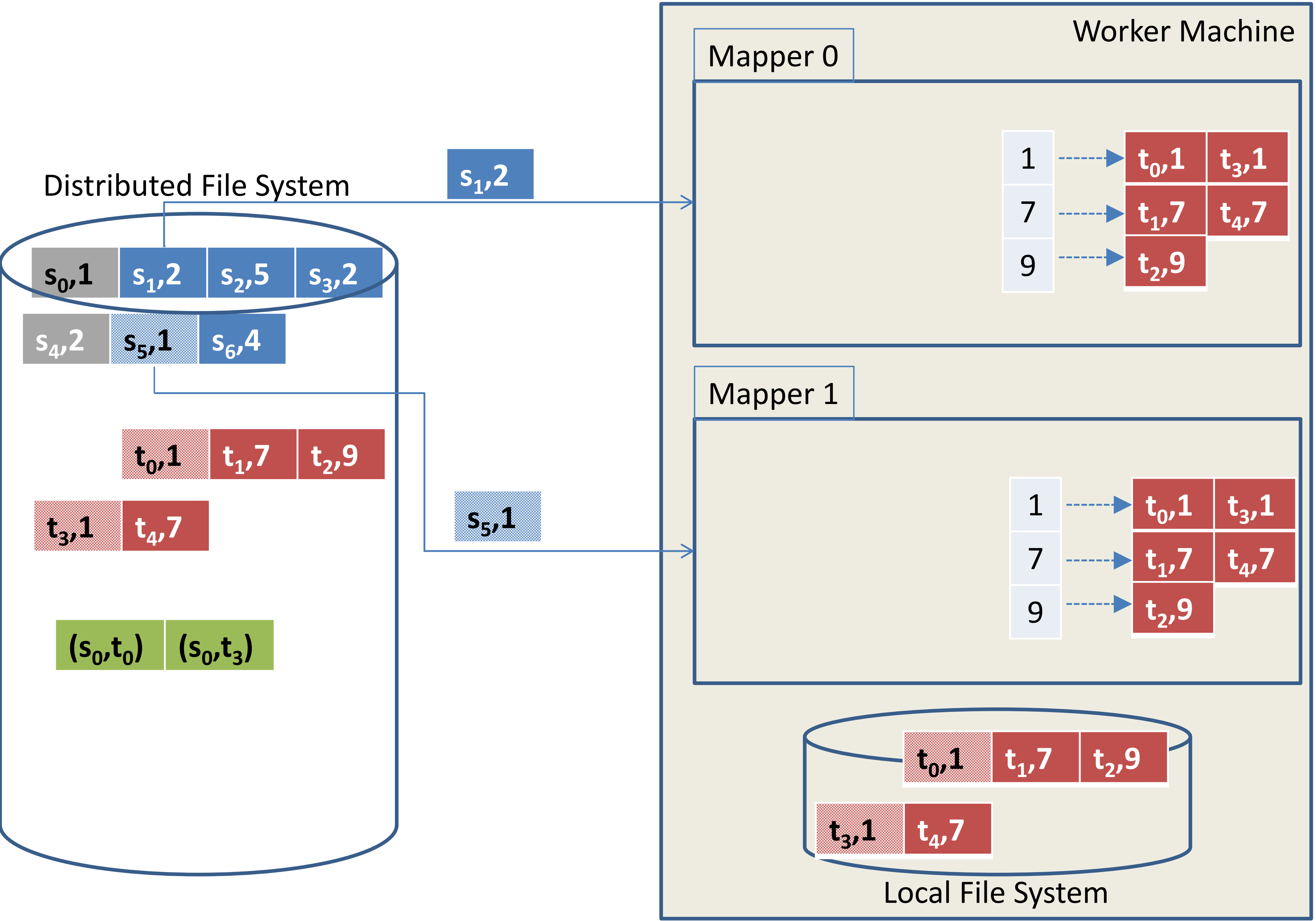
9 → t_2,9

empty result

t_0,1 t_1,7 t_2,9

t_3,1 t_4,7

Local File System



Worker Machine

Mapper 0

1 → t_0,1 t_3,1

7 → t_1,7 t_4,7

9 → t_2,9

Distributed File System

s_1,2

s_0,1 s_1,2 s_2,5 s_3,2

s_4,2 s_5,1 s_6,4

t_0,1 t_1,7 t_2,9

t_3,1 t_4,7

(s_0,t_0) (s_0,t_3)

s_5,1

Mapper 1

1 → t_0,1 t_3,1

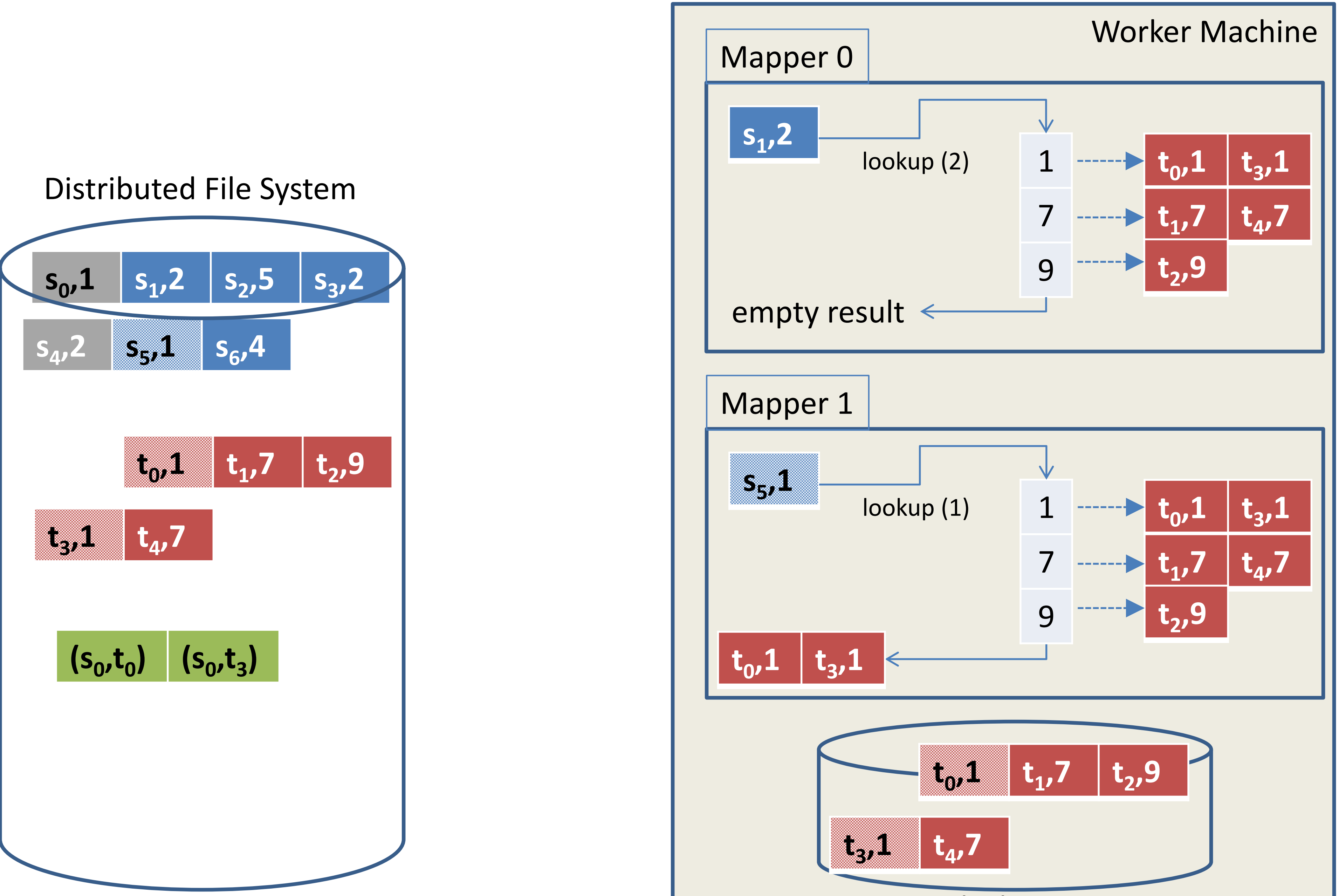
7 → t_1,7 t_4,7

9 → t_2,9

t_0,1 t_1,7 t_2,9

t_3,1 t_4,7

Local File System



Distributed File System

s_0,1 s_1,2 s_2,5 s_3,2

s_4,2 s_5,1 s_6,4

t_0,1 t_1,7 t_2,9

t_3,1 t_4,7

(s_0,t_0) (s_0,t_3)

Worker Machine

Mapper 0

s_1,2

lookup (2)

1 7 9

t_0,1 t_3,1

t_1,7 t_4,7

t_2,9

empty result

Mapper 1

s_5,1

lookup (1)

1 7 9

t_0,1 t_3,1

t_1,7 t_4,7

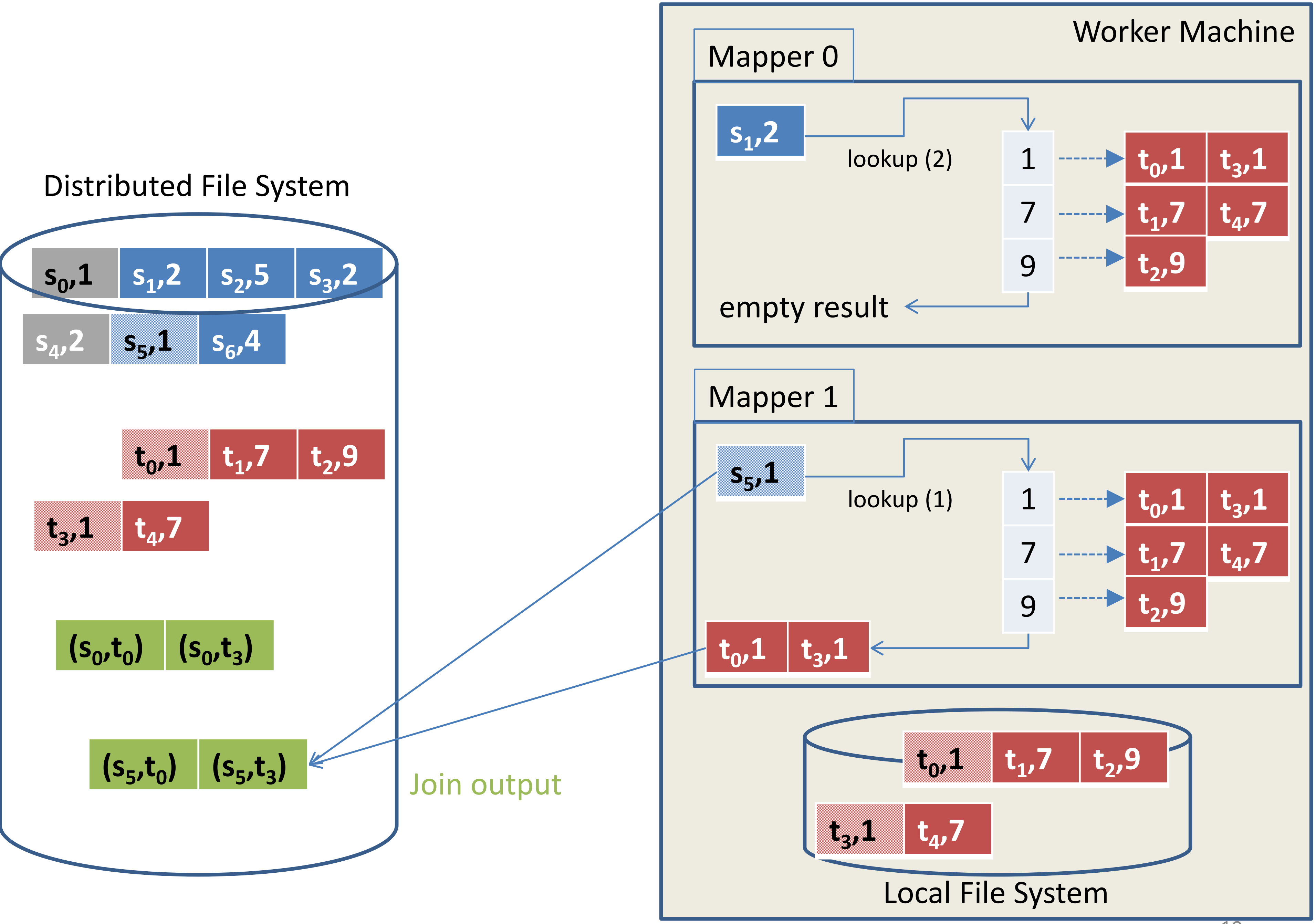
t_2,9

t_0,1 t_3,1

t_0,1 t_1,7 t_2,9

t_3,1 t_4,7

Local File System



Distributed File System

s_0,1 s_1,2 s_2,5 s_3,2

s_4,2 s_5,1 s_6,4

t_0,1 t_1,7 t_2,9

t_3,1 t_4,7

(s_0,t_0) (s_0,t_3)

(s_5,t_0) (s_5,t_3)

Join output

Worker Machine

Mapper 0

s_1,2

lookup (2)

1 → t_0,1 t_3,1

7 → t_1,7 t_4,7

9 → t_2,9

empty result

Mapper 1

s_5,1

lookup (1)

1 → t_0,1 t_3,1

7 → t_1,7 t_4,7

9 → t_2,9

t_0,1 t_3,1

t_0,1 t_1,7 t_2,9

t_3,1 t_4,7

Local File System

1.3 Discussion

- How does the Replicated join (i.e., the Map-only approach) compare to the Reduce-side join algorithm? The Map-only approach eliminates the high cost of the shuffle-and-sort and Reduce phases. So, it seems that there is a clear winner. Unfortunately, things are more complicated:
  - The entire T-file is sent to all machines that execute Map tasks. When using n machines, Replicated join reads T and then sends n·|T| data in addition to reading S. The Reduce-side join sends |S|+|T| data from Mappers to Reducers, in addition to reading S and T. Unless |T| is about n times smaller than |S|, Replicated join might therefore send even more data through the network.
  - Data set T has to be small enough so that the hash index fits into memory. If T is larger, one could resort to a disk-based index. However, cost would increase significantly due the I/O operations for transferring index pages repeatedly from disk to memory. The code would also be significantly more complex than for the simple in-memory hash index.
- In summary, Replicated join will work best if T is much smaller than S, ideally so small that it fits into memory on a Mapper machine.
- However, Replicated join has one major advantage over Reduce-side join: It generalizes easily to any theta join! [Link: What exactly is a theta join?]
  - To see why this holds, notice that for each pair (s, t) of tuples $s \in S$ and $t \in T$, there is exactly one Map task that receives both s (because it is in the input split for this task) and t (because every Mapper loads the entire set T). The Map function processing input tuple s can go through the entire list of T-tuples (instead of using the hash index) and evaluate predicate P for each pair consisting of s and a tuple from T.
- In a future module we will discuss a generalization of Replicated join and Reduce-side join that works for any theta-join and can perform even better.

2. Computing a Semi-Join

- The semi-join is similar to a join, but its output tuples have the attributes from only one of the two input data sets. Formally, a semi-join is defined as follows:
  - As for the join, we are given two data sets $S=(s_1,s_2,\ldots)$ and $T=(t_1,t_2,\ldots)$, such that each tuple has an attribute A.
  - The *left semi-join* between S and T on attribute A is the set of all tuples $s_i \in S$ for which there exists a tuple $t_j \in T$, such that $s_i.A=t_j.A$. The right semi-join is defined analogously.
- Like for joins, the above definition can be generalized to allow arbitrary Boolean functions over the attributes of pair (s_i, t_j), not just equality. This generalization is not considered further in this course.

Students

SID	Name	Age	GPA
1	Alice	18	3.5
2	Bob	27	3.4
3	Carla	20	3.8

Reservations

SID	BookID	Date
2	B10	01/17/12
3	B11	01/18/12
2	B11	01/20/12

“Regular” equi-join:
Students$\bowtie_{SID}$Reservations

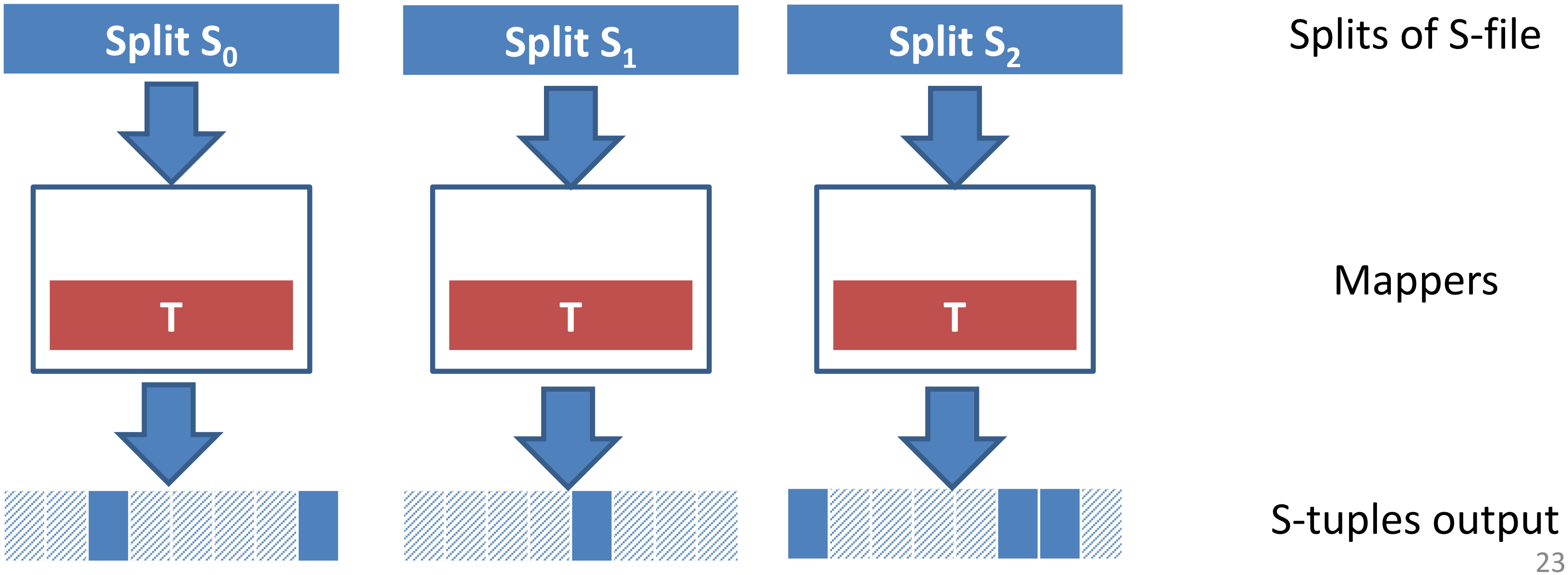
SID	Name	Age	GPA	BookID	Date
2	Bob	27	3.4	B10	01/17/12
2	Bob	27	3.4	B11	01/20/12
3	Carla	20	3.8	B11	01/18/12

Left semi-join:
Students$\ltimes_{SID}$Reservations

SID	Name	Age	GPA
2	Bob	27	3.4
3	Carla	20	3.8

2.1 Exact Solution

- One can view the left semi-join between S and T as a more complex type of *filter*. Instead of checking if an S-tuple s satisfies some predicate over its attributes (e.g., s.Age > 20), one has to check if it has a match in T.
  - Recall that filters are an example of *per-record computations*, which can be implemented with a Map-only job. Since now the "filter" is the entire data set T, each Mapper needs to have access to T.
- Alternatively, one can implement the semi-join as a regular equi-join and make sure (1) tuples from S are not duplicated in the output and (2) only attributes from S are passed to the output record.
  - This can be implemented using the Reduce-side join approach.
  - Interestingly, if the semi-join is implemented using the Replicated join approach, it will be identical to the algorithm derived based on the filter idea above: broadcast T to all Mappers, then have a Map call check if the incoming S-tuple has a match in T.
- Schematic drawing of the algorithm that replicates T on all Mappers and uses it as a filter for incoming S-tuples:



2.2 Toward Approximate Solutions

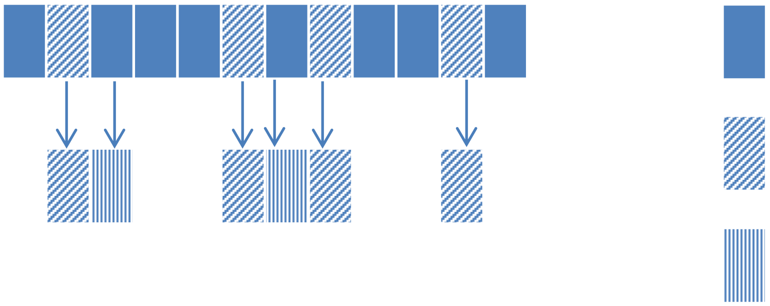
- As for the equi-join, broadcasting T to all Mapper machines will be expensive for large T. And if T does not fit into Mapper memory, expensive local I/O will further affect performance.
- How about sending a *subset*, e.g., a random sample, of T instead? A small enough subset would address the cost issue. Unfortunately, it has the drawback that some output tuples might be missing. This can be a problem, in particular if the user is looking for rare events of high importance.
- Can one find a different approximate solution that also limits the amount of data copied and loaded into Mapper memory, but *guarantees that no semi-join output tuple will be missed*? It turns out that *Bloom filters* are the answer.

2.3 Bloom Filter Details

- A Bloom filter is a *probabilistic* data structure that determines if a tuple s occurs in a given data set T. [Link: How does it work?] It is probabilistic in the following sense:
  - If the Bloom filter outputs “true”, i.e., claiming s∈T, the answer might be incorrect. More precisely, one cannot be sure if s is indeed in T or not.
  - On the other hand, if the Bloom filter outputs “false”, i.e., that s∉T, the answer is guaranteed to be correct.
  - In summary, the Bloom filter will not produce **false negatives** (S-tuples that actually are in T, but for which it outputs “false”), but it might produce **false positives** (tuples that actually are *not* in T, but for which it outputs “true”).
- Why would one want to use such a data structure that might give incorrect answers?
  - It helps eliminate irrelevant tuples, even though it might *not* eliminate *all* irrelevant tuples.
  - For example, assume S is a huge data set of events and that analyzing an event is expensive. To reduce processing cost, the analyst would like to focus on the critical events only. Let T be a set of signatures of critical events. Event s∈S only needs to be processed, if it matches a signature in T. Using a Bloom filter, any event s for which the Bloom filter returns “false” can be safely ignored—it is guaranteed to not be critical. This way it helps cut cost.
  - However, it would not completely eliminate processing of non-critical events, because of the false positives.
- Wouldn’t we cut cost even more if the Bloom filter returned no false positives at all?
  - True. Ideally we would like to use the exact set T to eliminate *all* non-critical events. Unfortunately, this is not always achievable. In particular, if T is very large, it might exceed available memory and it would be expensive to copy it to all machines using it.
  - The Bloom filter’s winning proposition is the ability to customize its size as desired. This introduces a tradeoff: smaller Bloom filter size implies lower cost for testing membership in T, but also increases the probability of false positives.
    - Formally, the false positive rate of the Bloom filter is approximately $\left(1-e^{-kn/m}\right)^k$, where k, m, and n refer to the number of hash functions used, the number of bits in the filter, and the number of tuples in T, respectively.

Input set S:

Tuples for which the Bloom filter returned “true”



▮ S-tuple with no match in T

▨ S-tuple with match in T

▥ False positive

$h_0(t) = t \bmod 10$ $h_1(t) = (t+3) \bmod 10$

0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**

T = {2, 5, 6}

Insert 2 into the Bloom filter:

$h_0(2) = 2 \bmod 10 = 2$ $h_1(2) = (2+3) \bmod 10 = 5$

0	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**

Insert 5 into the Bloom filter:

$h_0(5) = 5 \bmod 10 = 5$ $h_1(5) = (5+3) \bmod 10 = 8$

0	**0**	**1**	**0**	**0**	**1**	**0**	**0**	**1**	**0**

Insert 6 into the Bloom filter:

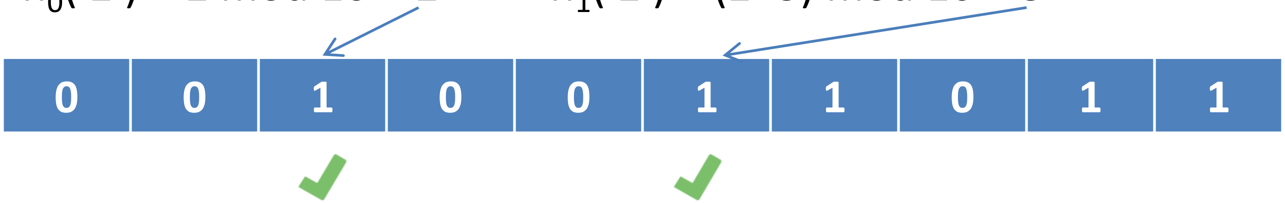
$h_0(6) = 6 \bmod 10 = 6$ $h_1(6) = (6+3) \bmod 10 = 9$

0	**0**	**1**	**0**	**0**	**1**	**1**	**0**	**1**	**1**

Bloom filter output:

Is value 2 in set T?

$h_0(2) = 2 \bmod 10 = 2$ $h_1(2) = (2+3) \bmod 10 = 5$

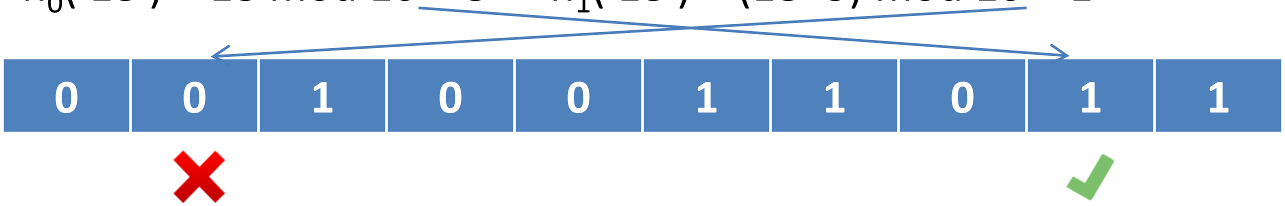


0	0	1	0	0	1	1	0	1	1

true

Is value 18 in set T?

$h_0(18) = 18 \bmod 10 = 8$ $h_1(18) = (18+3) \bmod 10 = 1$

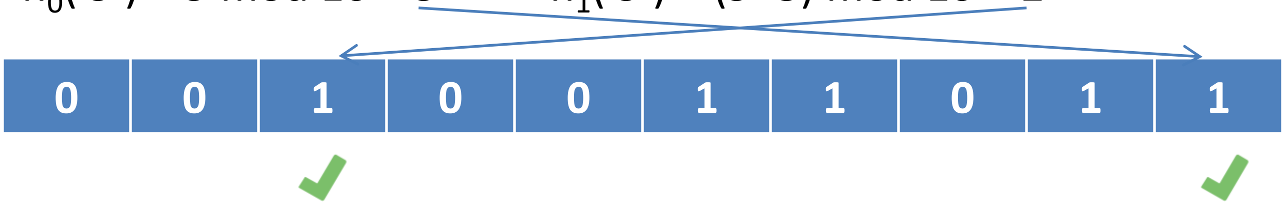


0	0	1	0	0	1	1	0	1	1

false

Is value 9 in set T?

$h_0(9) = 9 \bmod 10 = 9$ $h_1(9) = (9+3) \bmod 10 = 2$



0	0	1	0	0	1	1	0	1	1

true

2.4 Approximate Semi-Join Using a Bloom Filter

- This algorithm is virtually identical to the Replicated equi-join. Instead of creating a hash index, the Mapper creates the Bloom filter. And instead of emitting output *pairs*, only the S-tuple is emitted.

```
Class Mapper {
 BloomFilter B

 setup() {
  // Load data set T from the distributed file
  // cache into B.
  B = new BloomFilter
  for each tuple t in Distributed Cache
    B.insert( t )
 }

 map(…, S-tuple s ) {
  if ( B.lookup(s) )
    emit( NULL, s )
 }

 cleanup() { clean up B }
}
```

3. Summary

- When information is combined across different data sets, distributed algorithm design becomes more challenging.
- For equi-joins, the “natural” Reduce-side join approach requires both Map and Reduce phase. Map is used to send input records to the appropriate Reducers, while Reduce performs the actual join work.
- For problems where one of the two input sets is small, the Replicated join algorithm eliminates the need for a Reduce phase.
- Big data analysis tends to be inherently resource intensive. The way to achieve reasonable performance often is to settle for approximate answers. Bloom filters support approximate information about set membership. Their size is tunable, at the cost of a higher false positive rate. On the other hand, they are guaranteed to never produce false negatives. The latter makes them an ideal choice for applications where one wants to eliminate irrelevant input tuples at low cost.