Joins

Mirek Riedewald

Key Learning Goals

- What is an equi-join?
- Give an example for a join that is not an equi-join.
- Write the MapReduce pseudo-code for Reduce-side join and Replicated join.
- Given some input, argue *quantitatively* if Reduce-side (hash + shuffle) or Replicated (partition + broadcast) join will move less data through the network.
- Give an example when you should use hash + shuffle, but not partition + broadcast join.
- Give an example for the opposite.

Key Learning Goals

- What is a semi-join? Give an example to explain it.
- What problem do Bloom filters solve?
- What do we mean by “false positive” and “false negative” in the context of Bloom filters?

Introduction

- The join operator defines a surprisingly common computation pattern that can be applied to many real-world data analysis problems. It combines information across different data sets. While expensive—in the worst case quadratic in input size—it has a regular structure that enables efficient algorithms for computing it.
- 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.

Join Applications

- The relational join operator combines records from one dataset with records in another. It outputs all record pairs that satisfy the join condition. Examples:
  - 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 connect user data to their blog posts and tweets.
  - Joins can be used to analyze correlations and relationships between entities in different data sets. For instance, after joining pairs of records with similar elevation values, scientists can explore if their precipitation values are similar as well.
  - Joins 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 two intermediate stops.

Equi-Joins

- We are given two data sets $S=(s_1,s_2,\ldots)$ and $T=(t_1,t_2,\ldots)$. In database terminology, the records in S and T are *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."
- Consider tables about students and their book reservations. To analyze the GPA distribution of students who reserve certain books, these two tables need to be joined on the SID attribute. (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 ⋈ to express a join operator, annotated with the attributes the tables are joined on.

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

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

$\text{Students} \bowtie_{SID} \text{Reservations}$

Side Note: What Is a Theta-Join?

- Consider tuples $s \in S$ and $t \in T$, and any predicate P(S, T) that is defined over the attributes of S and T. For each pair (s, t), P(s, t) evaluates to either true or false. If it evaluates to true, then the pair is in the theta-join result, otherwise it is not.
  - Equi-joins are a special type of theta-join where P is constrained to only contain equality predicates.

Efficient Join Computation

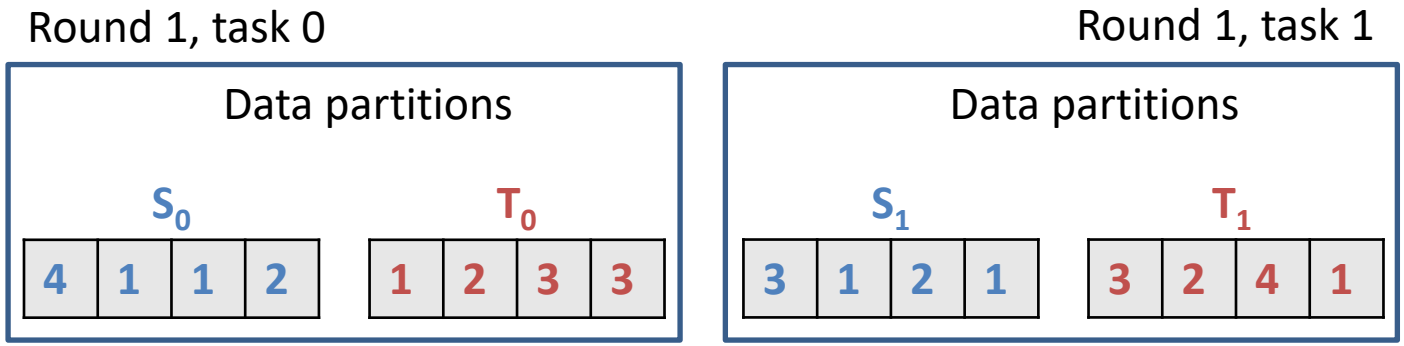
- Any theta-join, including equi-joins, is a subset of the Cartesian product. Hence it could be implemented by first producing all pairs (s_i, t_j) and then filtering out those that do not satisfy the join condition. This algorithm is *inefficient* for joins when most pairs do not match.
- For equi-joins, hash partitioning on join attribute A can dramatically reduce the number of S-T pairs examined.
  - Given k bins, assign each input tuple x to bin number hash(x.A) mod k. Then all S-tuples and T-tuples with the same A-value end up in the same bin. Hence only tuples in the same bin need to be checked for matches.
  - Assuming each input bin receives about the same number of input tuples, the problem of checking |S|·|T| possible pairs is reduced to checking $k \frac{|S|}{k} \frac{|T|}{k}$ pairs, i.e., 1/k of the cost.
- This idea can be applied analogously to equi-joins on multiple attributes by hashing on all join attributes.
- We now discuss how to apply this idea in a distributed context.

Distributed Join: Hash + Shuffle

- The basic idea is to group each input by join attribute and co-locate the matching groups in the same task. This requires shuffling, because any join value $a \in A$ may occur in any input split.
  - Join-attribute values can be assigned to tasks, using a simple hash function.
- Then each join group performs a local Cartesian product between the S-tuples and T-tuples.
  - All these tuples match, because we already separated them based on A-values.

Hash Join With Shuffling

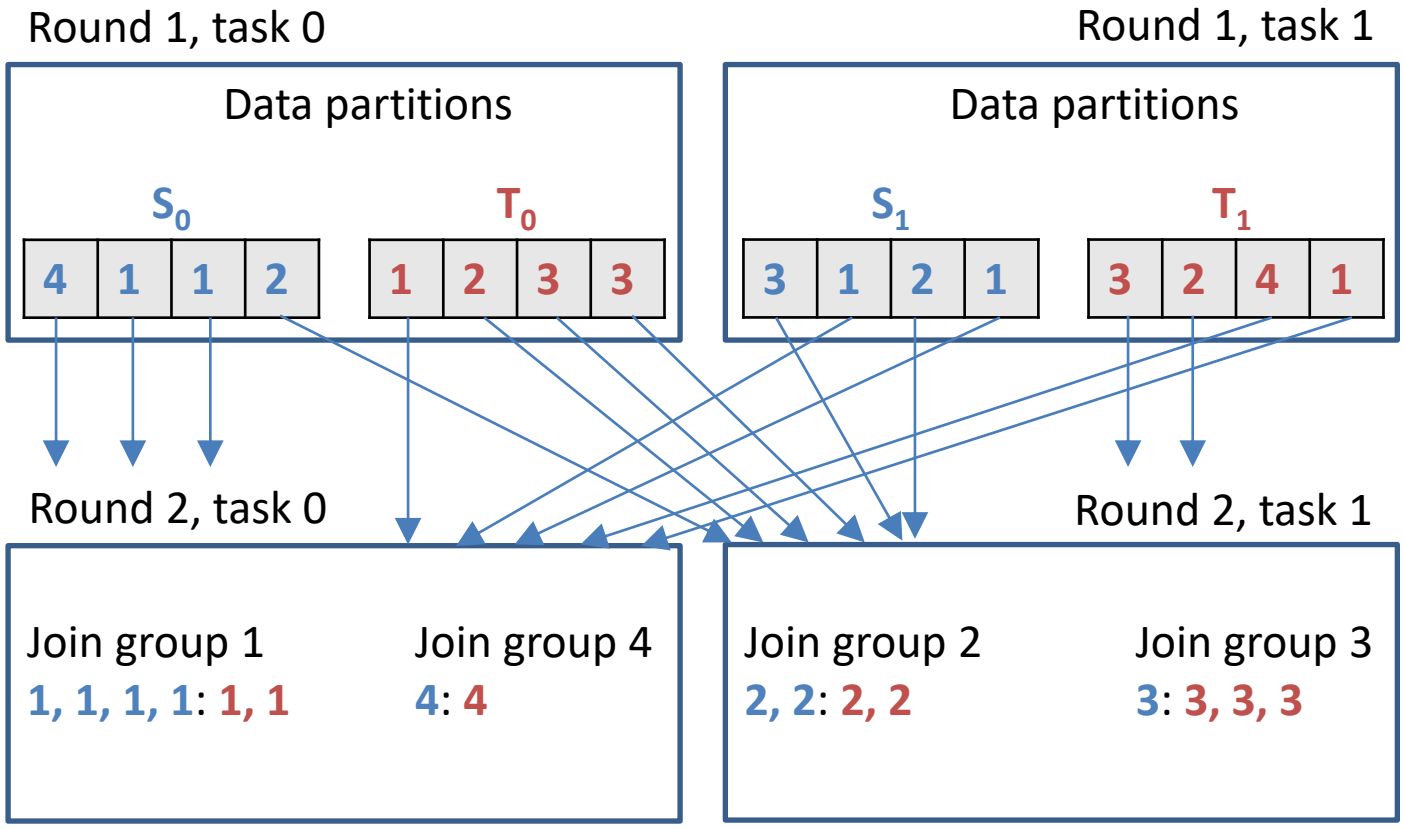
Initial data location



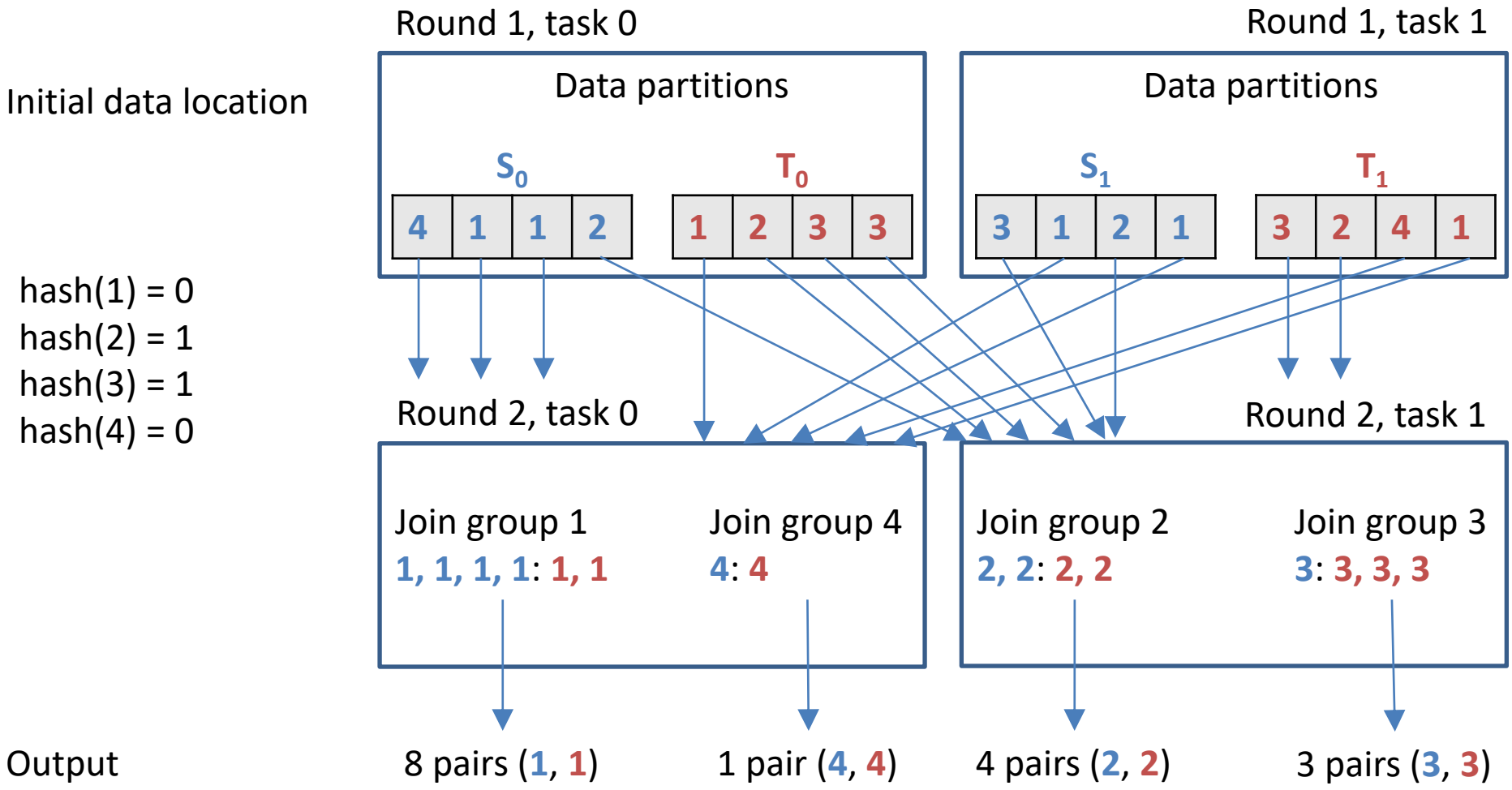
Hash Join With Shuffling

Initial data location

hash(1) = 0
hash(2) = 1
hash(3) = 1
hash(4) = 0



Hash Join With Shuffling

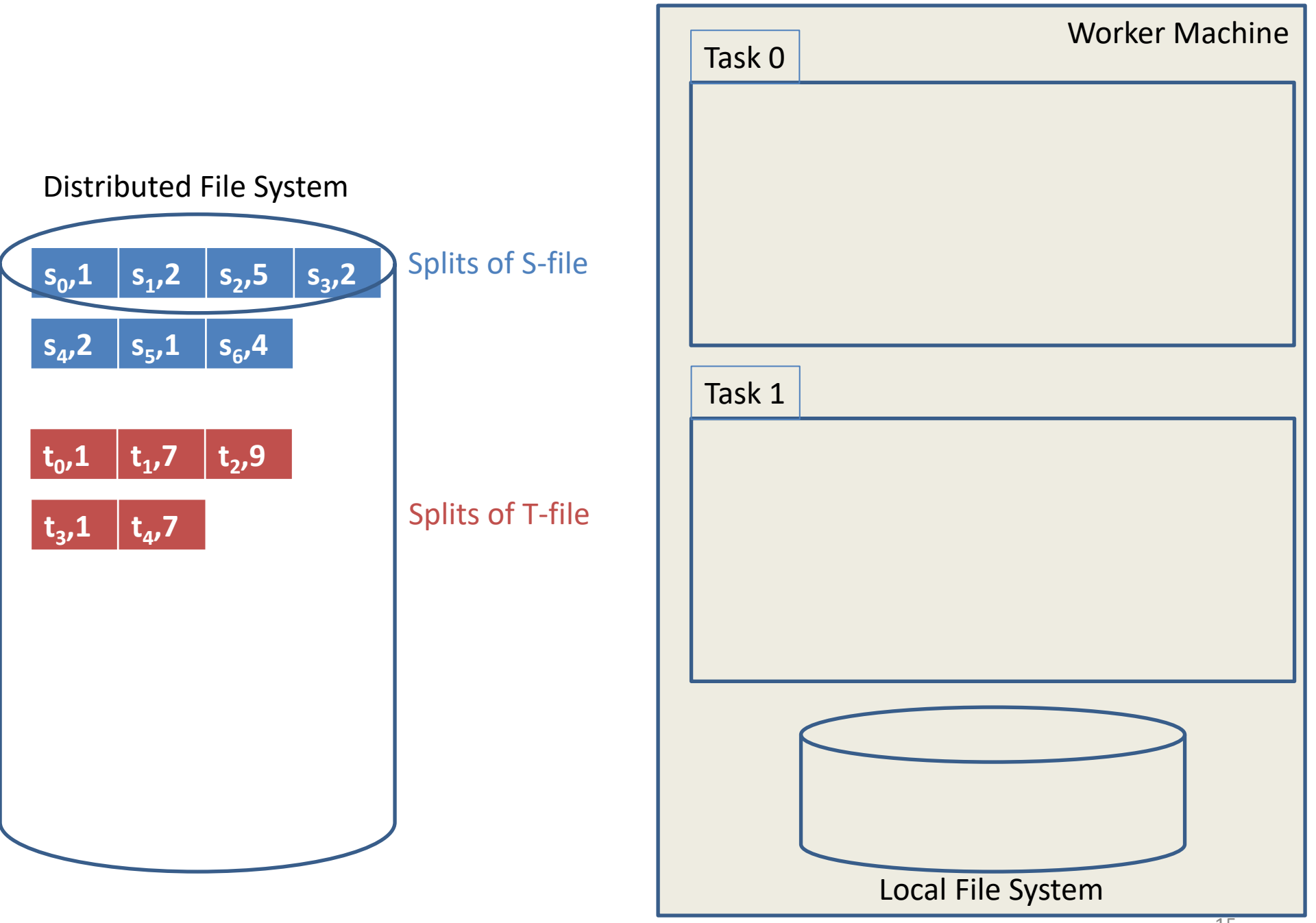


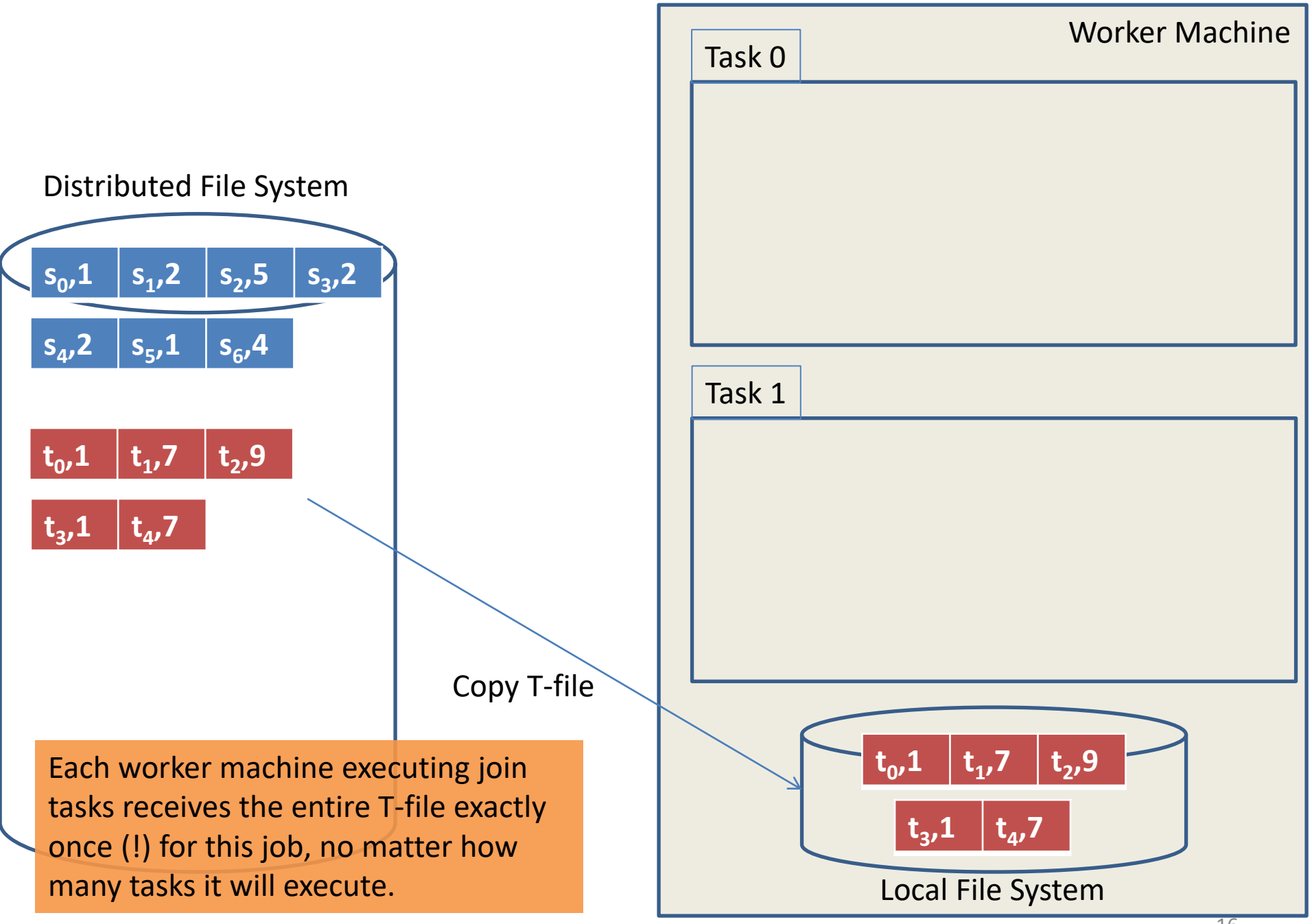
Discussion

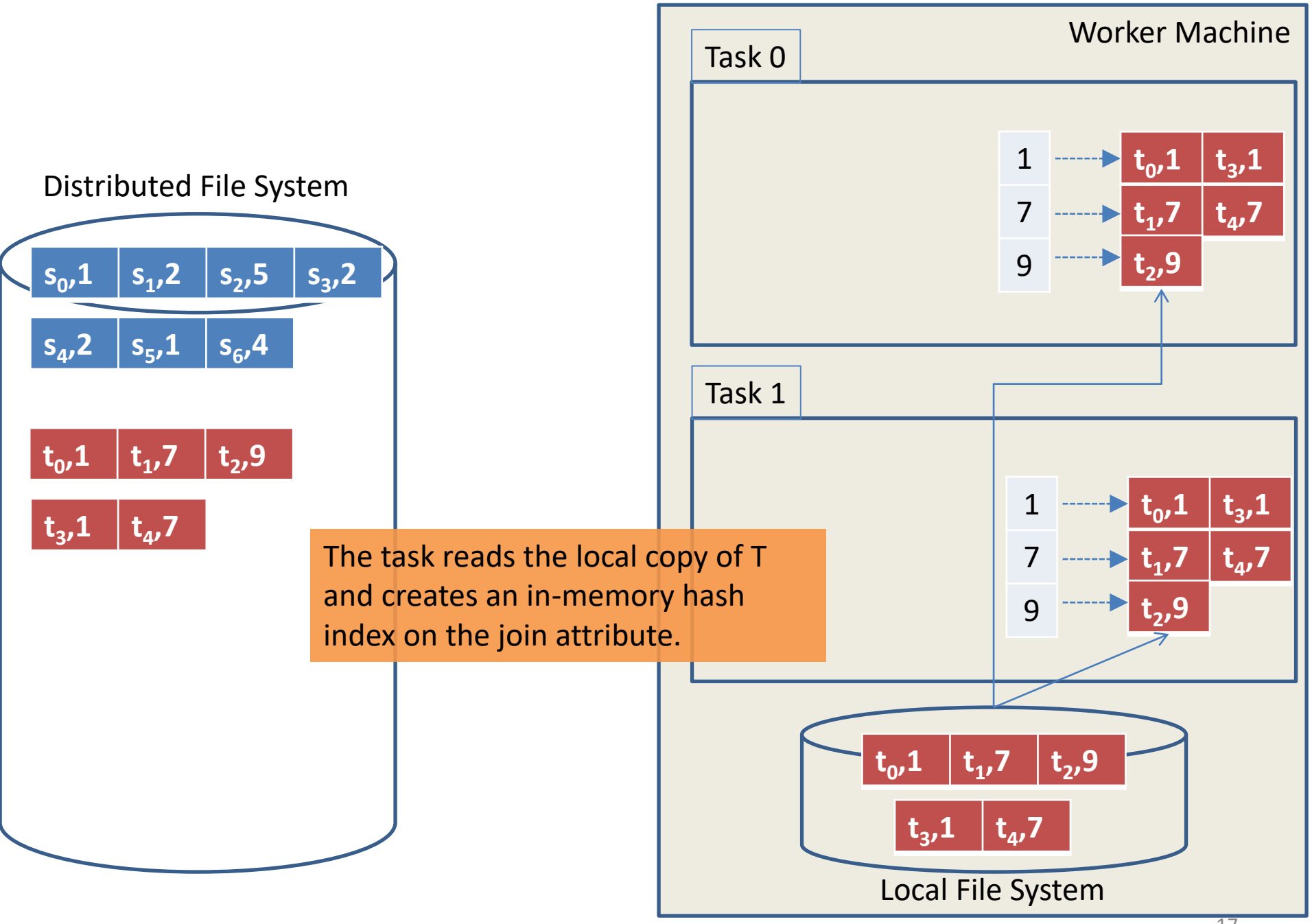
- The hash + shuffle algorithm avoids checking non-matching tuples. It groups the input by the join attribute (a linear-cost operation), then only combines tuples in the same group, which by design are matching. It might be hard to imagine that one could do any better than this. Unfortunately, there are several drawbacks:
  - The approach does not balance load well when the input is highly skewed. Consider the join group for A=1 in the example. If the data had even more tuples with A=1, then the round-2 task receiving that join group could become a bottleneck.
  - The approach does not scale well for join attributes with small domains. If only k distinct join attribute values occur in the input (k=4 in the example), then there are only k different join groups. No matter how many machines are available, all but k of them would be idle in round 2.
  - Hash-partitioning on the join attribute inherently limits the algorithm to equi-joins. It does not generalize to non-equi joins, such as inequality conditions (S.A < T.A) or band-joins ($|S.A - T.A| < \varepsilon$).
  - Both S and T are read twice—in round 1 and again in round 2.

Distributed Join: Partition + Broadcast

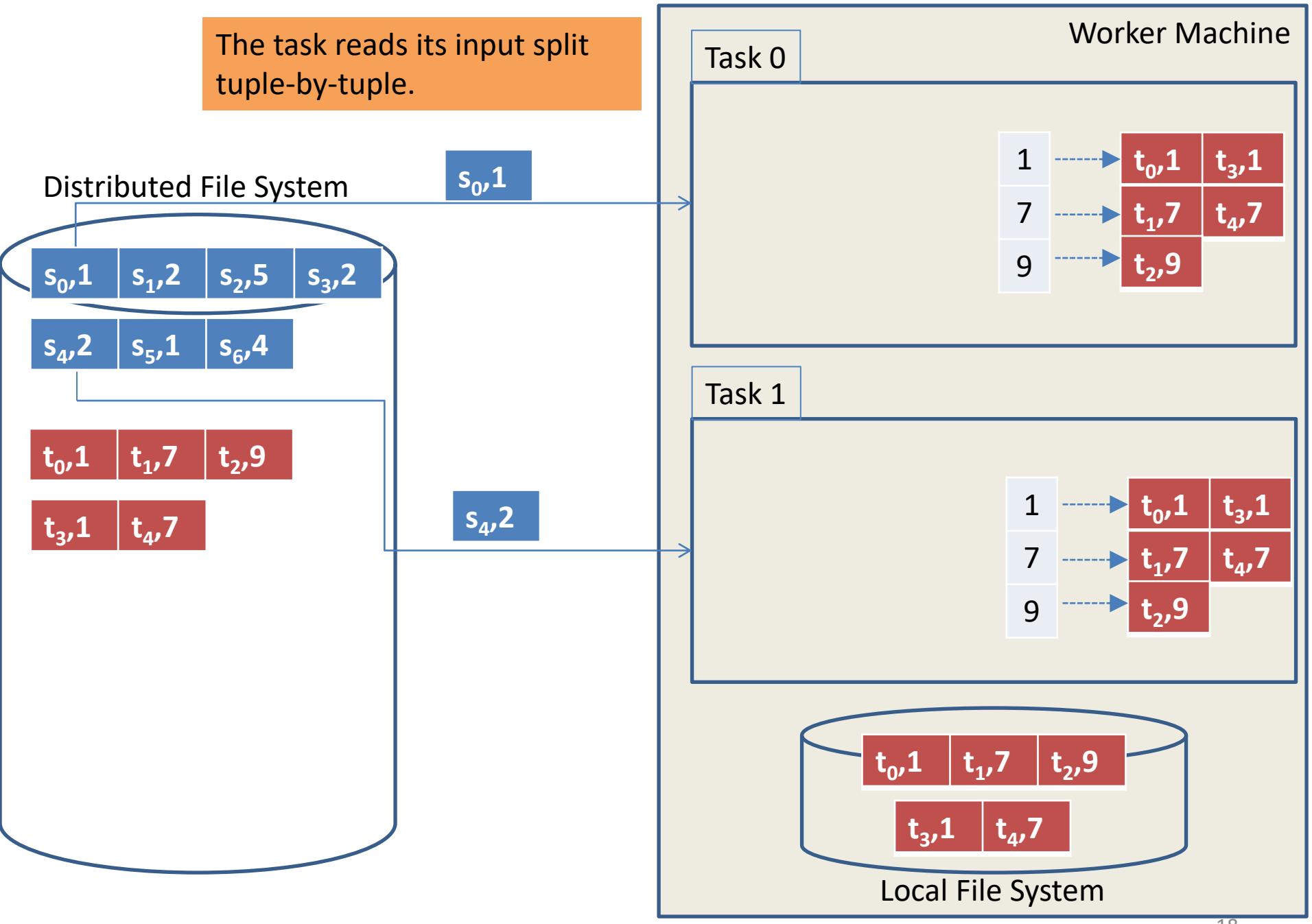
- Let us attempt to address the shortcomings of hash + shuffle by starting with the last problem: Can we compute the entire join in a single round?
  - This may seem impossible, because tuples with the same A-value could be anywhere in the input file. Without data shuffling, how would one guarantee that an S-tuple is joined with a matching T-tuple that could be in any split of the T-file?
- Let us start with a gedankenexperiment: Assume a task receives an arbitrary input split from S. Which tuples from T should be sent to that task so that it can compute all matches?
  - Since the S-split could contain any A-value, we must send the *entire input T* to the task. Otherwise it might miss matches for its S-tuples.
- This analysis forms the foundation of the partition + broadcast algorithm:
  - Each task receives an arbitrary subset of S-tuples, e.g., a file split. It also receives a copy of the entire T, i.e., we partition S and broadcast T.
  - For efficient broadcasting, notice that a worker might execute multiple tasks. Instead of requesting T repeatedly, the worker should hold on to the first copy it receives, and re-use it for all tasks.
  - For efficient local lookups, the task could store T in a hash index that maps each $a \in A$ to the set of all T-tuples t with $t.A = a$.
- We show an example to illustrate this algorithm next.

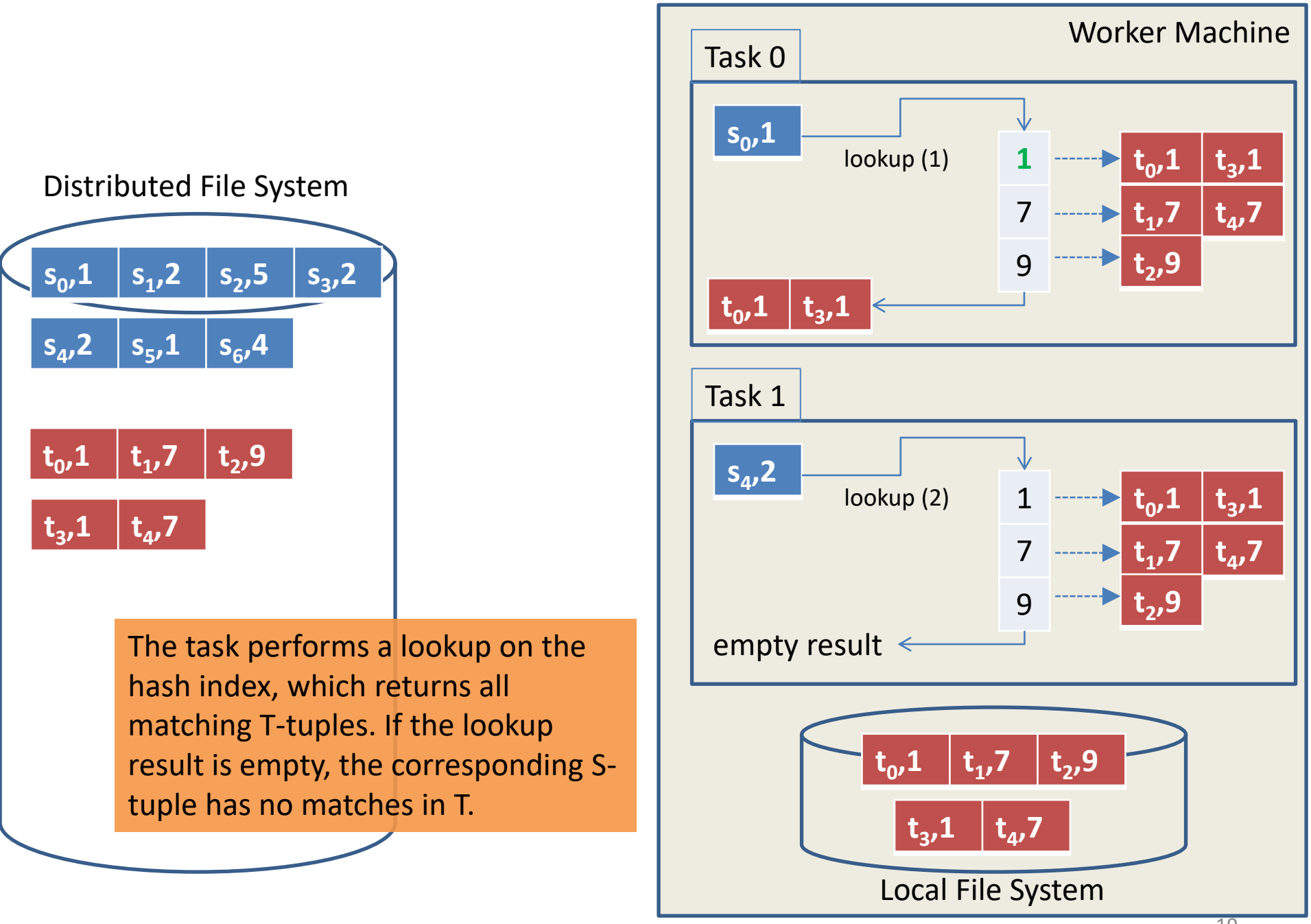

Distributed File System
$s_0,1$ $s_1,2$ $s_2,5$ $s_3,2$
Splits of S-file
$s_4,2$ $s_5,1$ $s_6,4$
$t_0,1$ $t_1,7$ $t_2,9$
$t_3,1$ $t_4,7$
Splits of T-file
Worker Machine
Task 0
Task 1
Local File System

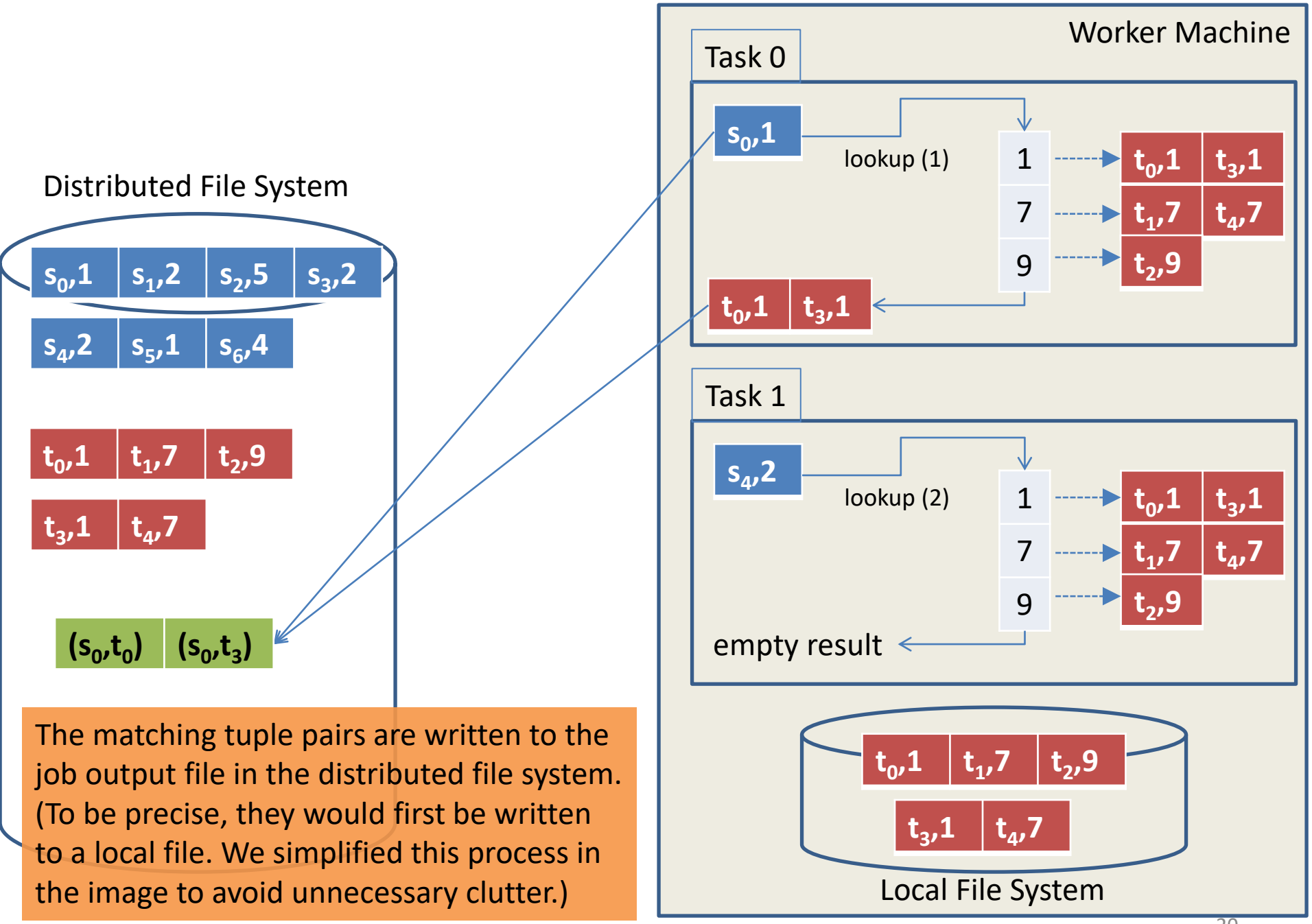

Distributed File System
s_0,1
s_1,2
s_2,5
s_3,2
s_4,2
s_5,1
s_6,4
t_0,1
t_1,7
t_2,9
t_3,1
t_4,7
Copy T-file
Worker Machine
Task 0
Task 1
t_0,1
t_1,7
t_2,9
t_3,1
t_4,7
Local File System
Each worker machine executing join tasks receives the entire T-file exactly once (!) for this job, no matter how many tasks it will execute.


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
Task 0
1
7
9
t_0,1
t_3,1
t_1,7
t_4,7
t_2,9
Task 1
1
7
9
t_0,1
t_3,1
t_1,7
t_4,7
t_2,9
t_0,1
t_1,7
t_2,9
t_3,1
t_4,7
Local File System
The task reads the local copy of T and creates an in-memory hash index on the join attribute.

The task reads its input split tuple-by-tuple.

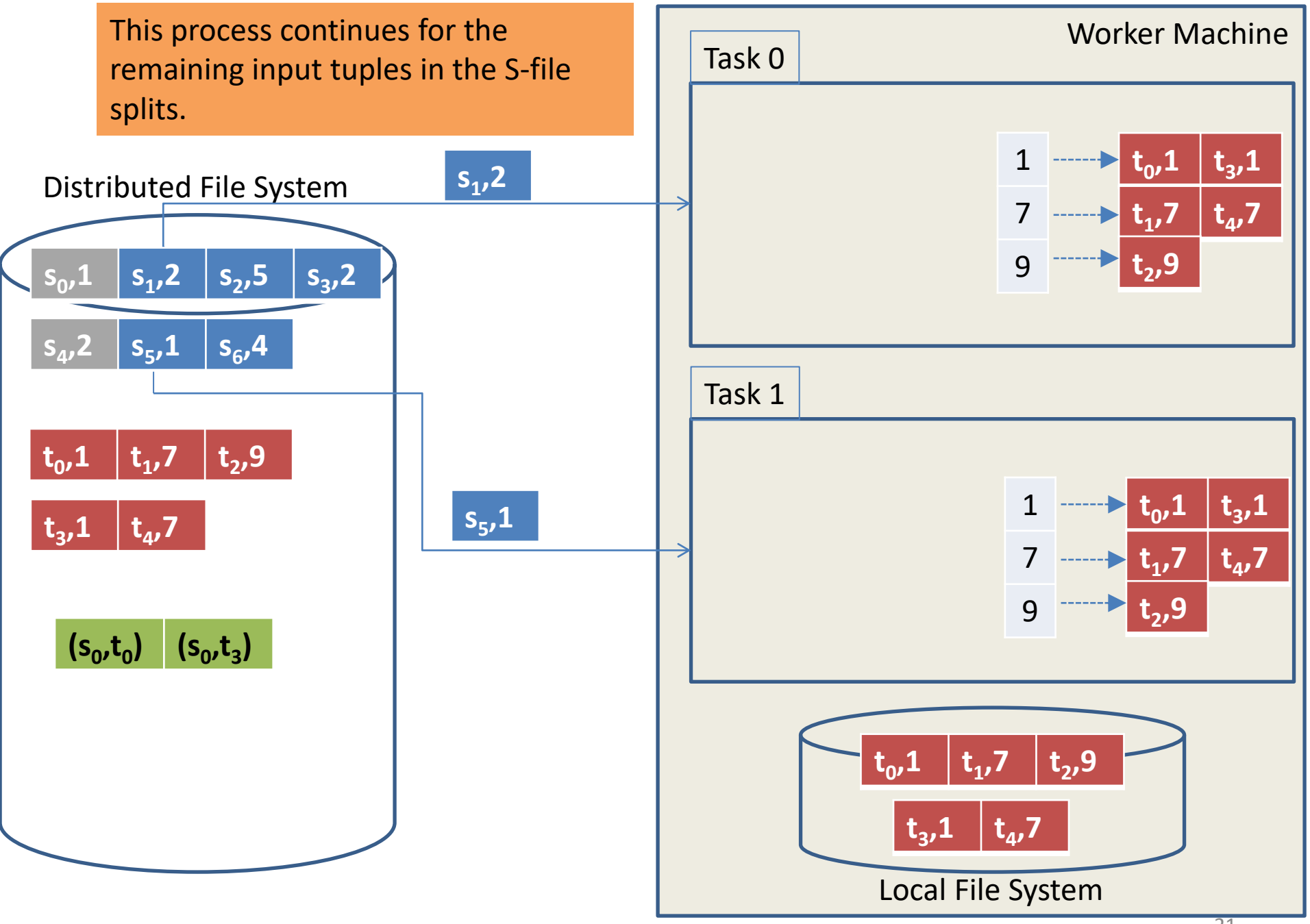



Distributed File System
$s_0,1$ $s_1,2$ $s_2,5$ $s_3,2$
$s_4,2$ $s_5,1$ $s_6,4$
$t_0,1$ $t_1,7$ $t_2,9$
$t_3,1$ $t_4,7$
The task performs a lookup on the hash index, which returns all matching T-tuples. If the lookup result is empty, the corresponding S-tuple has no matches in T.
Worker Machine
Task 0
$s_0,1$
lookup (1)
1
7
9
$t_0,1$ $t_3,1$
$t_0,1$ $t_3,1$
$t_1,7$ $t_4,7$
$t_2,9$
Task 1
$s_4,2$
lookup (2)
1
7
9
empty result
$t_0,1$ $t_3,1$
$t_1,7$ $t_4,7$
$t_2,9$
$t_0,1$ $t_1,7$ $t_2,9$
$t_3,1$ $t_4,7$
Local File System

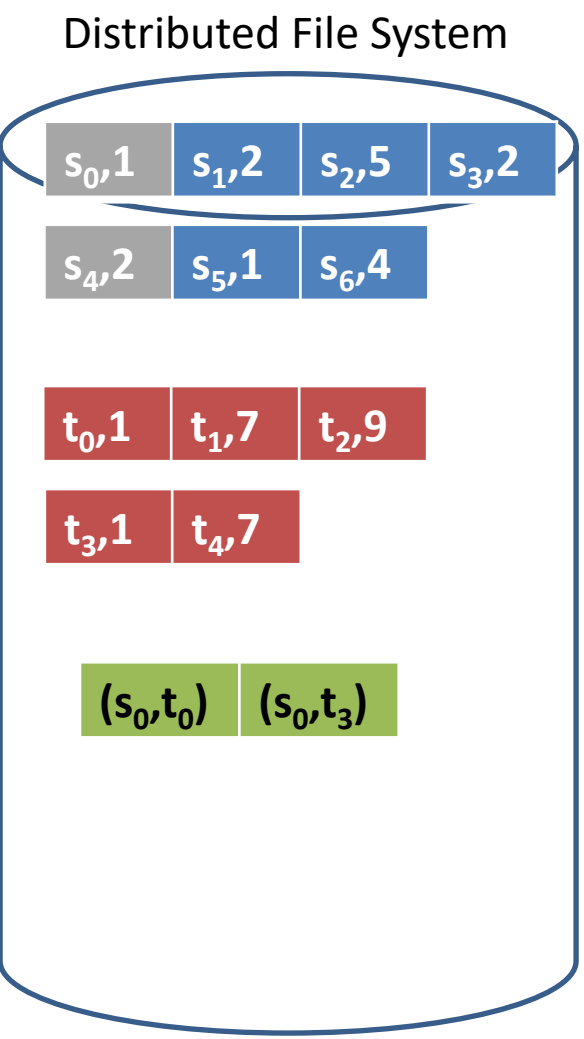


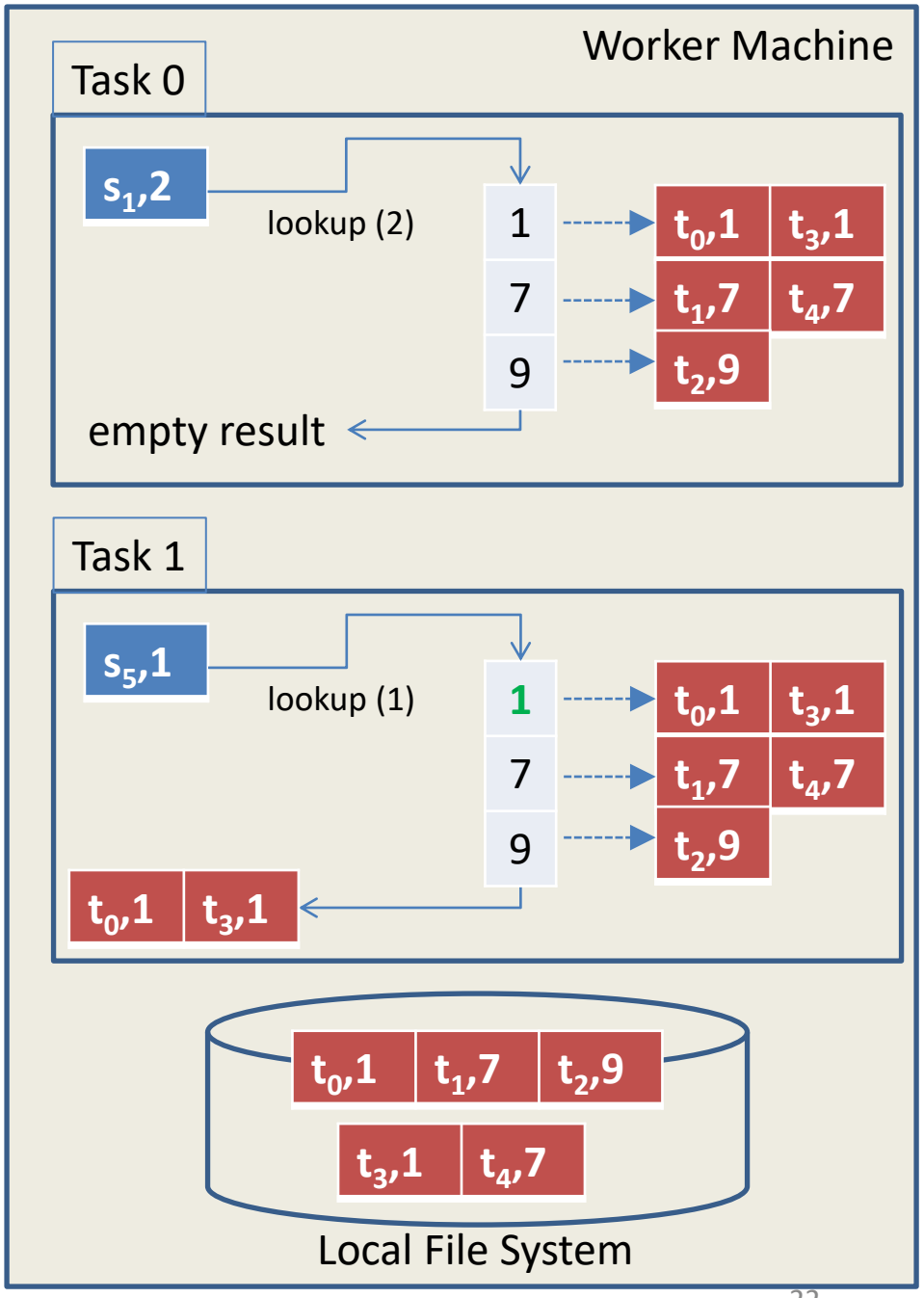
The matching tuple pairs are written to the job output file in the distributed file system. (To be precise, they would first be written to a local file. We simplified this process in the image to avoid unnecessary clutter.)

This process continues for the remaining input tuples in the S-file splits.

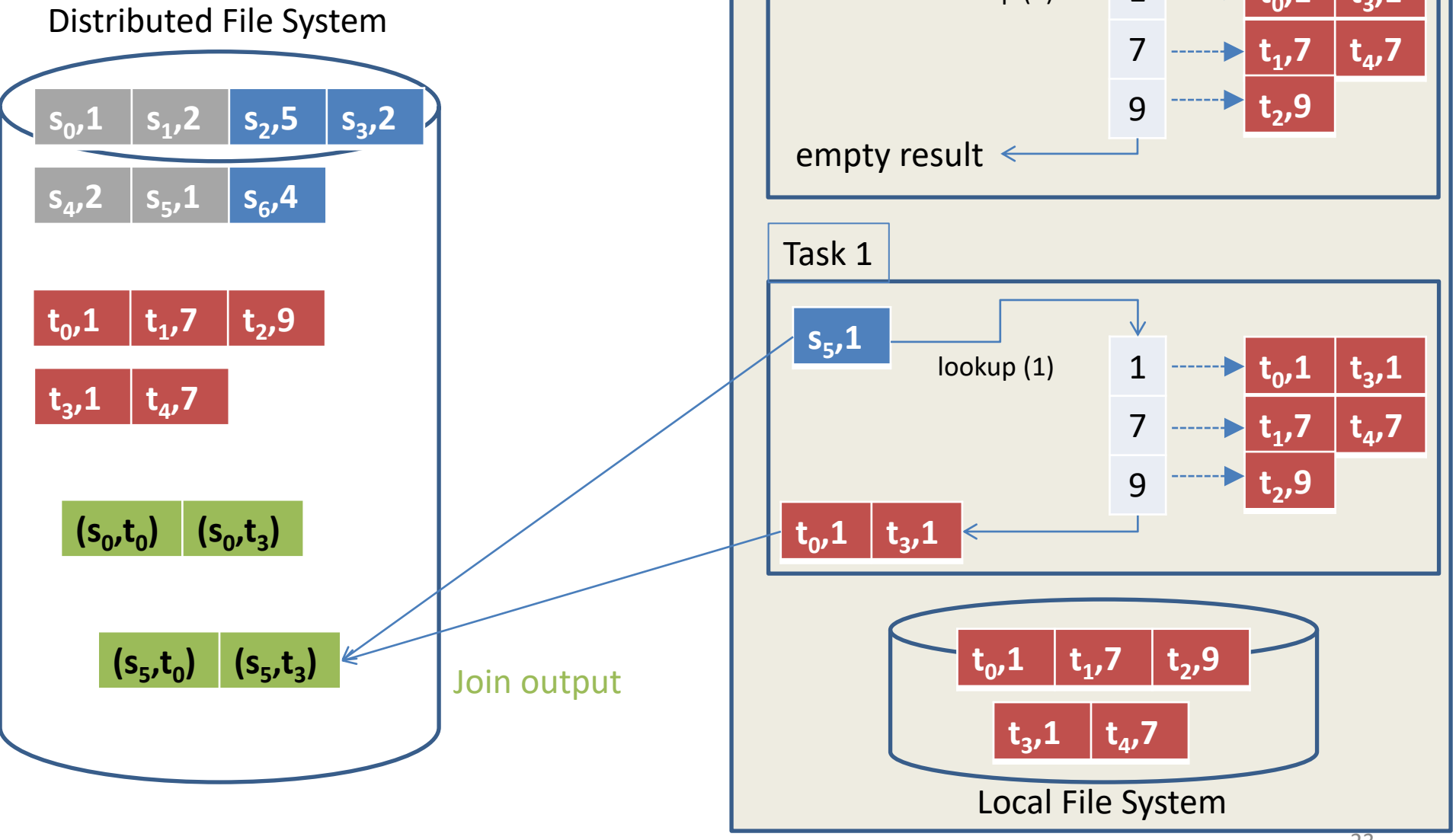


The task performs a lookup on the hash index, which returns all matching T-tuples.





The matching tuple pairs are written to the job output file in the distributed file system. The next steps will not produce any more output.



Discussion

- Partition + broadcast works best if T is much smaller than S and *fits in memory*. Otherwise, T needs to be paged in and out from disk, increasing code complexity and access cost.
- This algorithm generalizes to any theta-join!
  - To see why, notice that for each pair (s, t) of tuples $s \in S$ and $t \in T$, there is exactly one task that receives both s (because it is in the input split for this task) and t (because every task receives the entire T). The task processing input tuple s can go through the entire list of T-tuples (instead of using the hash index) and evaluate the join predicate for each pair (s, t).
- In a future module we will discuss a generalization of hash + shuffle and partition + broadcast that works for any theta-join and can perform even better.

Which Join Algorithm is Better?

- Hash + shuffle groups by join attribute and hence does not balance load well for join attributes with skewed distribution or small domain. Partition + broadcast can create as many partitions as needed. It tends to distribute load evenly, because every task receives the same amount of input.
- Hash + shuffle is limited to equi-joins. Partition + broadcast can implement any theta-join condition.
- Hash + shuffle can join two large inputs, but for partition + broadcast, one of the inputs should fit in memory.
- Which of the two will transfer less data through the network? This depends:
  - In partition + broadcast, the entire T-file is sent to all machines that execute tasks. When using n machines, it reads T once and then sends $n \cdot |T|$ data in addition to reading S.
  - The hash + shuffle algorithm sends $|S|+|T|$ data from round 1 to round 2, in addition to reading S and T.
  - Unless $|T|$ is about n times smaller than $|S|$, partition + broadcast might send more data through the network than hash + shuffle, despite not shuffling.

Cost Reduction through Co-Partitioning

- Sometimes the same dataset participates in multiple joins on the same key, or in a group-by followed by a join on the same key.
- In those cases, it pays off to hash-partition both inputs by the join key once—using the same Partitioner—and to maintain the partitions on the worker machines for repeated re-use. This is called co-partitioning or co-grouping.
  - This is not possible in Hadoop MapReduce, because each job reads the data "from scratch" from HDFS.
- Co-partitioning enables the use of the hash + shuffle algorithm, *without the need for shuffling*!

Cogrouping in Spark

- rdd1.cogroup(rdd2) transformation: for pair RDD rdd1 of type (K, V) and pair RDD rdd2 of type (K, W), it returns a pair RDD of type (K, (Iterable(V), Iterable(W)))—one element for each key that exists in either input.
  - If a key exists only in one but not the other input, then the corresponding iterator is empty.
  - Elements in the Iterable can be accessed as usual with foreach().
- Cogroup is like an outer join, except the matching pairs are implicitly stored in the Iterables.

Let us now see how the join algorithms are expressed and implemented in Hadoop MapReduce, Spark, and DBMS.

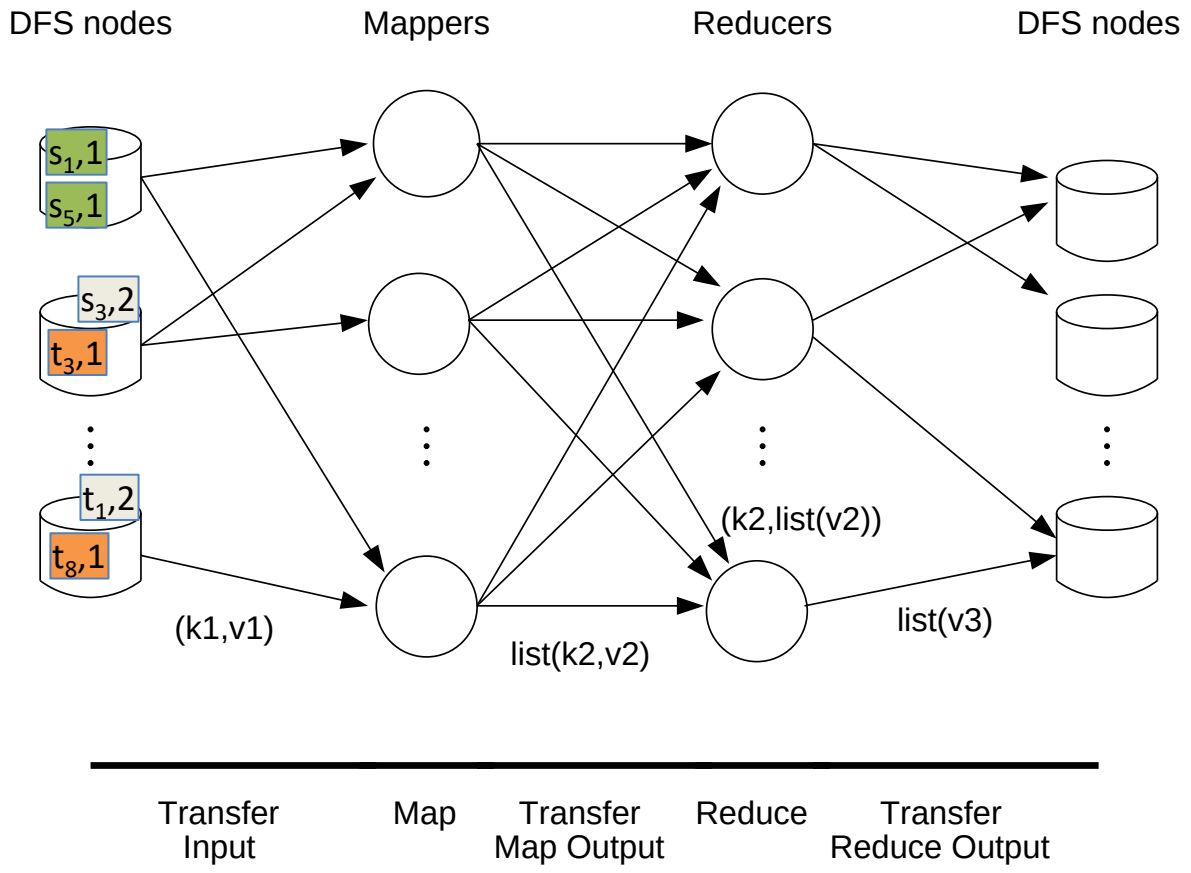
Hash + Shuffle in MapReduce

- In MapReduce, this algorithm is called Reduce-side join—for obvious reasons. It is implemented by making join attribute A the intermediate key and by using a hash Partitioner. For each input tuple, Map emits the tuple with its A value as the key. The Reduce call for an A value receives all matching S- and T-tuples. To distinguish where a tuple came from, Map adds flag "S" or "T" to the value.
- The Reduce function shown loads all input values into memory. If S_list and T_list do not fit in memory, they can be written to local files. Instead of simple nested loops pairing up S- and T-tuples, a file-I/O optimized algorithm could be used to page chunks of these files in and out of memory when needed.
- One can also use secondary sort 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 in memory. The other can be accessed through the iterator provided by MapReduce for the Reduce input value list.

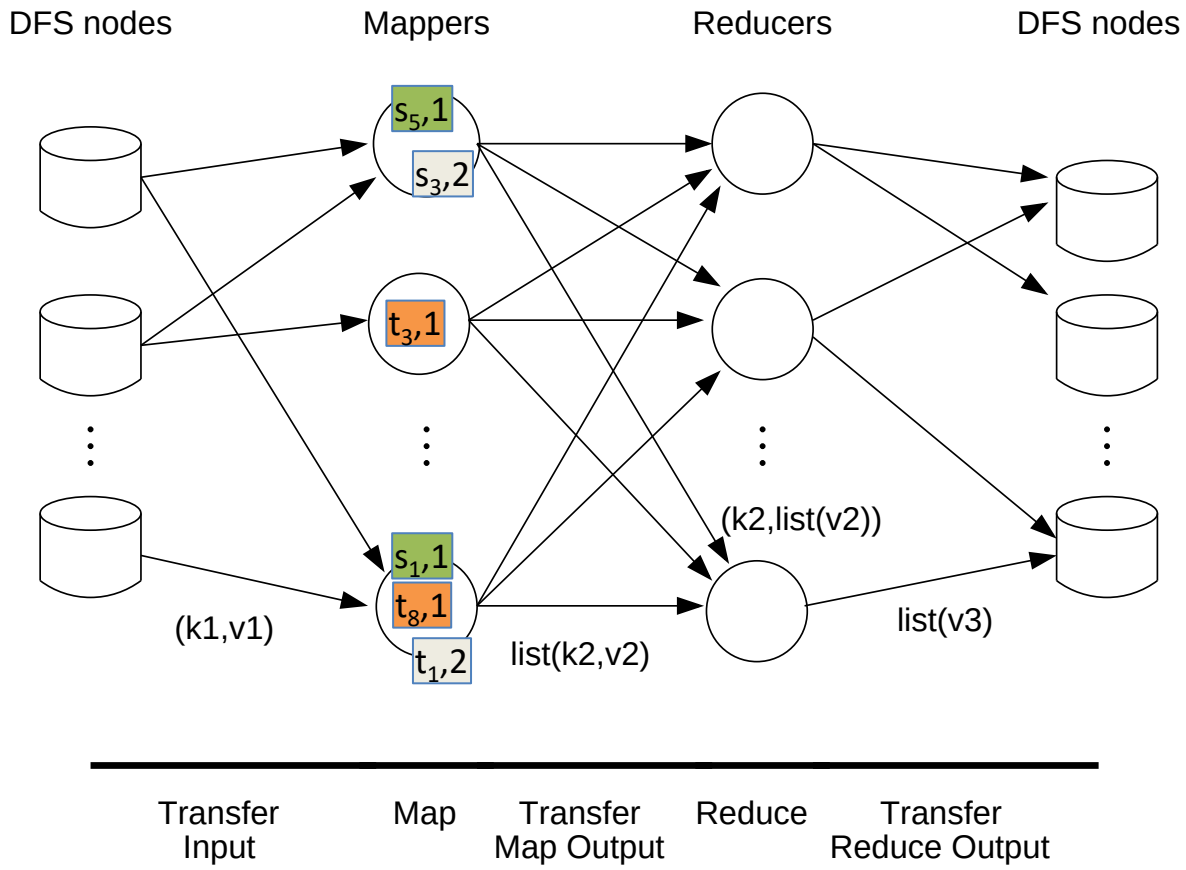
```
map( …, tuple x )
  if (x is from S)
    emit( x.A, (x, "S") )
  else    // x is from T
    emit( x.A, (x, "T") )
```

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

  // Separate the input list by the input the tuples came from
  for all (x, flag) in input list do
    if (flag = "S") then 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( s, t )
```

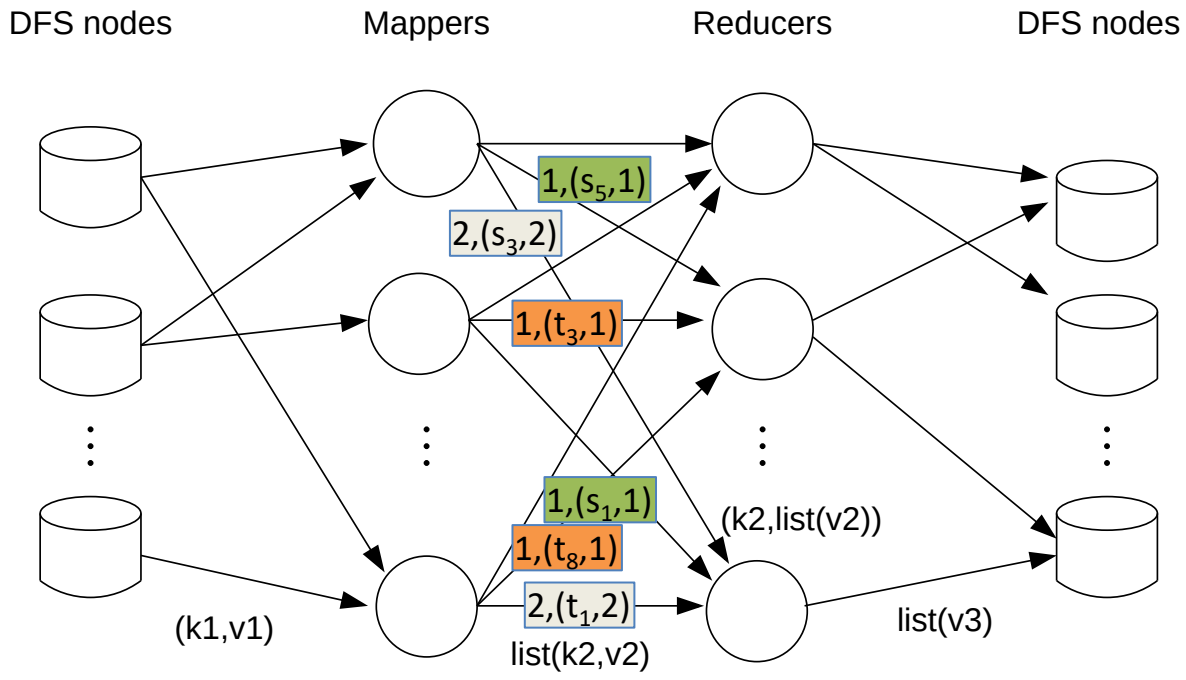
- In this example, each s_i and t_j is an S- or T-tuple, respectively. For clarity, the value of join attribute A is shown as well, e.g., (s_5,1) indicates s_5.A = 1.
- The matching input tuples are usually scattered over multiple file splits and might be assigned to different Mappers.



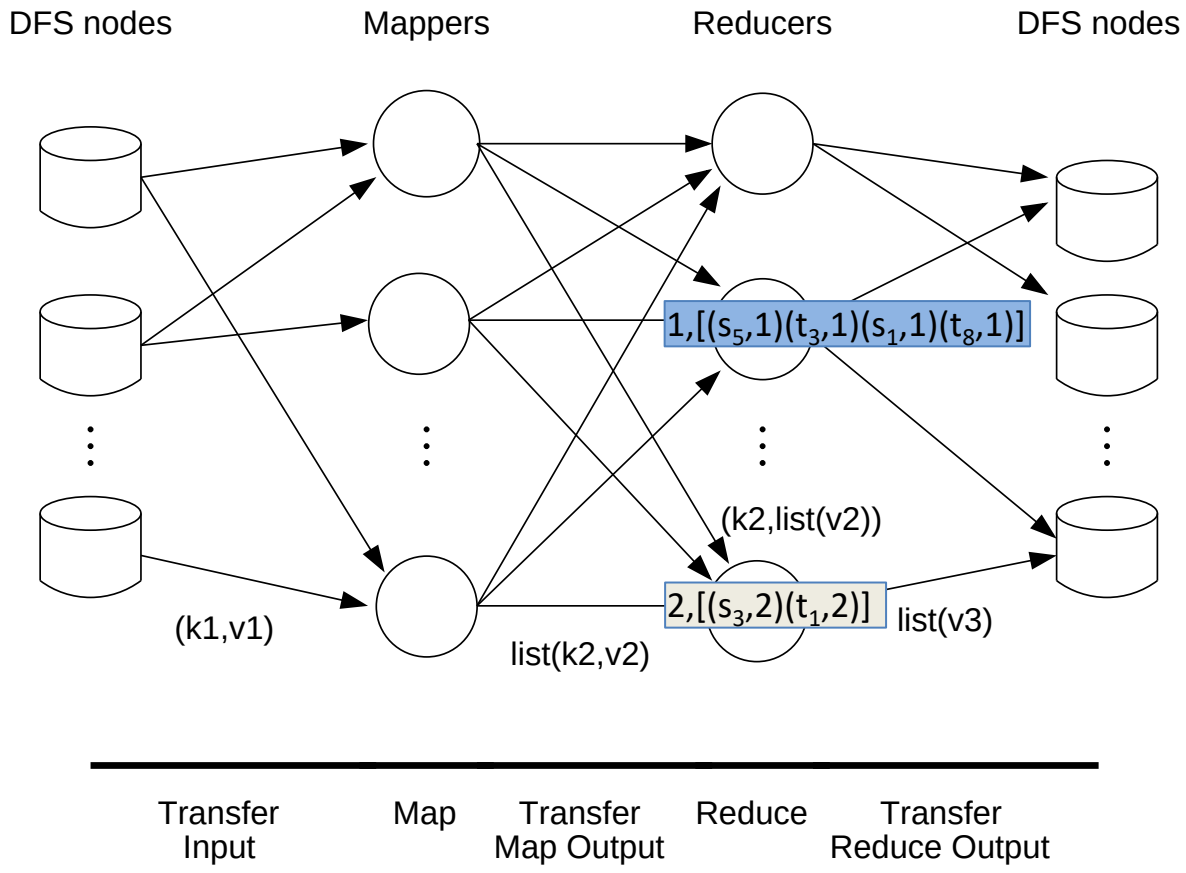
- The input tuples are transferred to the Mappers.



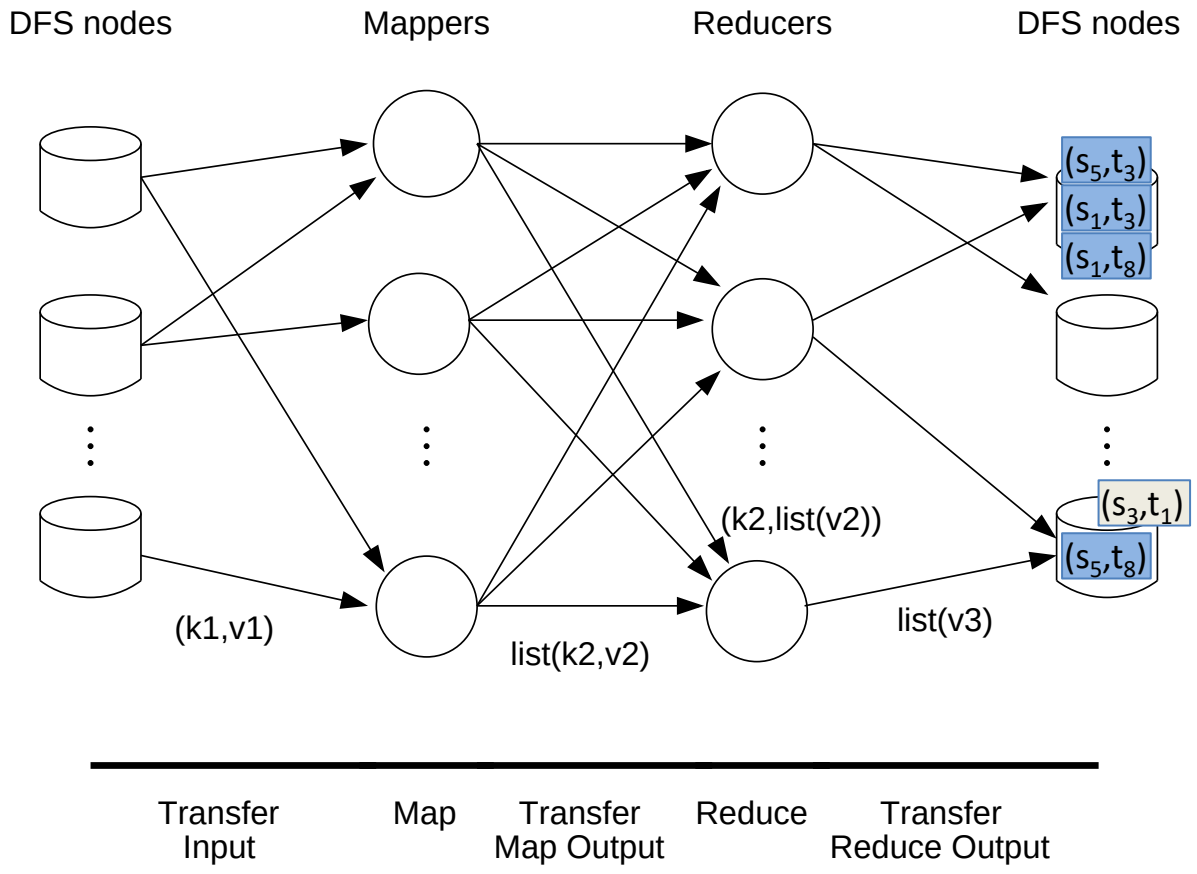
- Map emits each tuple with its A value as the key.



- In this example, only two Reducers receive the Map output—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.



Implementation Details

- Usually S and T are stored in separate files, especially when S and T have different schemas or formats. This causes 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 or T?
- For cases where a MapReduce job processes 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, S_Mapper and T_Mapper. S_Mapper parses only S-tuples and adds flag "S" to the emitted value; T_Mapper parses only T-tuples and adds flag "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]),…, S_Mapper.class);
    - MultipleInputs.addInputPath(job, new Path(args[1]),…, T_Mapper.class);

Real Code

- Check out the example code from the Miner/Shook book at http://khoury.northeastern.edu/home/mirek/code/ReduceSideJoinDriver.java
- It joins a file of users on the user ID attribute with another file containing their comments. Notice the use of MultipleInputs.
- In addition to the equi-join as discussed before (called “inner join” in the program) it also supports variations called “outer joins.”

Partition + Broadcast in MapReduce

- This algorithm is called Replicated join or Map-only join in MapReduce.
- It sets S as the job input and broadcasts T through the file cache:
  - 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 S (not T). For each S-tuple, it probes the hash index to find all matching T-tuples.

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

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

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

Real Code

- Check out the example code from the Miner/Shook book at http://khoury.northeastern.edu/home/mirek/code/ReplicatedJoinDriver.java
- It uses the Map-only approach to join a file of users with another file with their comments based on a user ID attribute.
- The user file is copied to all Mappers, using the DistributedCache. (This feature is now deprecated. Find out what replaces it.) The setup() function reads it and constructs a hash index of users on their ID.
- For an input comment record, Map extracts the user ID and uses it to look up the matching user record in the hash index. Since the user ID uniquely identifies a user, the lookup will never return more than one user.

Joins in a DBMS

- Any join can be expressed in SQL as SELECT * FROM S, T WHERE joinCondition(S, T). A typical equi-join condition is S.A = T.A AND S.B = T.B.
- Instead, one can also specify inequalities such as S.A < T.A or |S.A – T.A| < 2, or even a user-defined Boolean function over the attributes of S and T.
- The DBMS optimizer will automatically choose the join implementation it considers superior. Options include block nested loop, index nested loop, sort-merge join, and hash join.
  - Hash + shuffle is the parallel version of the hash join.
  - The parallel version of index nested loop join is the partition + broadcast algorithm.

Joins in Spark

- In Spark, join options and their implementation depend on the data representation used: RDD versus DataSet and DataFrame.
- The following discussion on joins in Spark is partially based on [H. Karau and R. Warren. High Performance Spark. O'Reilly, 2017]

Join Basics

- Spark only offers equi-joins. For pair RDDs, there are join, leftOuterJoin, rightOuterJoin, and fullOuterJoin.
- When called on an RDD of type (K, V), with an RDD of type (K, W) as input, they return an RDD of type
  - (K, (V, W)) for the join
  - (K, (V, Option(W))) for the left outer join
  - (K, (Option(V), W)) for the right outer join
  - (K, (Option(V), Option(W))) for the full outer join.
  - Here Option(T) represents any object of type T or None (i.e., NULL). This is needed, because outer joins emit results also for elements from one RDD that have no matches in the other.
- These operations have versions that take a Partitioner object or a number of partitions as parameter.
  - If no Partitioner or number of partitions is specified, Spark takes the Partitioner of the first RDD. If the input RDDs have no Partitioners, Spark uses the hash Partitioner with either the default number of partitions set in spark.default.partitions, or, if the default is not defined, the largest number of partitions of the input RDDs.

Cartesian Product

- rdd1.cartesian(rdd2) transformation: for rdd1 of type T and rdd2 of type U, it returns an RDD of type (T, U). This RDD contains all pairs of elements from rdd1 and rdd2.
- This operation tends to be very expensive. If rdd1 has m elements and rdd2 has n, then there are m·n results.
- It can be used to implement any *theta-join*, but often more efficient solutions exist, e.g., as we discussed for equi-joins.
  - Challenge: How is cartesian() implemented in Spark?

Hash + Shuffle Join with Pair RDDs

- In the example below, the key is of type Long. Note that in the result RDD, the key is of type Long as well. The value is of type (Double, String), i.e., the value columns from left and right input.
  - There is no need to store both input keys, because they have the same value. The join here is an equi-join!
- By default, the join is implemented as hash + shuffle.

```
def joinOnKey( leftRDD : RDD[(Long, Double)], rightRDD : RDD[(Long, String )])
  : RDD[(Long, (Double, String))]= {
    val joinedRDD = leftRDD.join(rightRDD)
}
```

Avoiding Join Shuffling

- The pair-RDD join can only avoid shuffling if both RDDs are co-partitioned, i.e., both have a known Partitioner and the Partitioners are identical.
  - This situation may arise when joining after a _ByKey operation, as shown in the example from the Karau/Warren book below. There reduceByKey explicitly assigns the right input's Partitioner to the aggregated left input.
- We will see another example when discussing PageRank.

```
def joinWithoutShuffle(leftRDD: RDD[(Long, Double)], rightRDD: RDD[(Long, String)])
  : RDD[(Long, (Double, String))]= {
   // If rightRDD has a known Partitioner we should use that, otherwise it has a default
   // hash parttioner, which we can reconstruct by getting the number of partitions.
   val rightDataPartitioner = rightRDD.partitioner match {
     case (Some(p)) => p
     case (None) => new HashPartitioner(rightRDD.partitions.length)
   }
   val maxValueLeftData = leftRDD.reduceByKey(rightDataPartitioner, (x, y) => if(x > y) x else y)
   maxValueLeftData.join(rightRDD)
}
```

Partition + Broadcast with Pair RDDs

- This is called broadcast hash join in Spark.
- It is not supported for RDDs, but only in Spark SQL for DataFrames. The example from the Karau/Warren book on the next page shows an implementation for pair RDDs in user code.
  - It collects the smaller RDD as a map (i.e., a dictionary data structure) to the driver, who broadcasts it to all workers.
  - Notice the use of mapPartitions for pairing up the matching tuples. The iterator goes through the partition of the large RDD, looking up matches in the dictionary that stores the small RDD as a mapping from key to value.

Partition + Broadcast for RDDs in User Code

```
def manualBroadCastHashJoin[K : Ordering : ClassTag, V1 : ClassTag, V2 : ClassTag]
  (bigRDD : RDD[(K, V1)], smallRDD : RDD[(K, V2)])= {
    val smallRDDLocal: Map[K, V2] = smallRDD.collectAsMap()
    bigRDD.sparkContext.broadcast(smallRDDLocal)
    bigRDD.mapPartitions(iter => {
      iter.flatMap{
        case (k,v1 ) =>
          smallRDDLocal.get(k) match {
            case None => Seq.empty[(K, (V1, V2))]
            case Some(v2) => Seq((k, (v1, v2)))
          }
        }
      }, preservesPartitioning = true)
    }
    //end:coreBroadCast[]
}
```

Simpler Version for RDDs

```
// There is a size limit for broadcasting a collection, which can be increased.

// Turn the small pair RDD into a hash-map style collection for broadcasting
val smallAsMap = sc.broadcast( smallRDD.collect.toMap )

// For each row in the large RDD, find all values associated with the same key
in the hash map and return all corresponding value pairs
largeRDD.flatMap {
   case(keyLarge, valueLarge) => smallAsMap.value.get(keyLarge).map {
      valueSmall => (keyLarge, (valueLarge, valueSmall))
   }
}
```

Joins with DataFrame and DataSet

- Like in a DBMS, the Spark optimizer automatically attempts to find the best join execution plan. This includes reordering of operations to apply early projections and selections (discussed soon).
- The user cannot control the Partitioner manually. That is also handled by the optimizer.
- The join example below shows that since there is no explicit key in a DataSet or DataFrame, the user must specify the join columns—here SID.
- In addition to the (inner) join as discussed so far, DBMS and Spark DataFrame/DataSet also support outer joins. Outer joins preserve input tuples that have no match in the other input.

```
studentDF.join(reservationDF, studentDF("SID") === reservationDF("SID"))

// Shorter version of the same
studentDF.join(reservationDF, "SID")
```

Self-Join with DataFrames

- By default, the self-join creates a result where column names are duplicated. To be able to refer to the different columns, they must be renamed. In Spark, like in SQL, we can define an alias for a DataFrame name using “as.”
  - Notice the specification of the join condition in the where() function, like in SQL’s WHERE clause.

```
studentDF.as(“S1”).join(studentDF.as(“S2”)).where($”S1.SID” === $”S2.SID”)
```

Partition + Broadcast with DataFrames

- The Spark optimizer automatically selects the partition + broadcast algorithm when appropriate.
  - We can give a hint to Spark SQL by calling broadcast on the smaller DataFrame as shown below.
  - The optimizer behavior can be controlled through spark.sql.conf.autoBroadcastJoinThreshold.
- Notice the shorter version of the join condition: only the join column is specified, implying an equi-join on it.
- The query plan selected by the optimizer can be shown by calling queryExecution.executedPlan in Spark SQL.

```
reservationDF.join(broadcast(studentDF), “SID”)
```

Challenge Question

- Why does DataSet offer a separate joinWith function?

Challenge Question

- Why does DataSet offer a separate joinWith function? Here is the definition from the DataSet API (Spark 2.3.2, September 2018):
  - joinWith: “This is similar to the relation join function with one important difference in the result schema. Since joinWith preserves objects present on either side of the join, the result schema is similarly nested into a tuple under the column names _1 and _2. This type of join can be useful both for preserving type-safety with the original object types as well as working with relational data where either side of the join has column names in common.”

Reducing Join Cost

- Joins often are building blocks of a more complex computation. This may be exploited to reduce cost. We focus on projections (removal of fields from each tuple) and selections (filters).
- As an example, consider finding the names of all 20-year-old students who reserved books in 2012. The SQL query is shown below.

Students

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

```
SELECT name
FROM Students S, Reservations R
WHERE S.SID = R.SID
   AND S.age = 20
   AND R.date.year = 2012
```

Reservations

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

Order of Operations

- All three execution plans below are equivalent. Which one will have the lowest cost?
  - Symbol π is the projection, i.e., removes columns.
  - Symbol σ is the selection (filter), i.e., removes rows.

Plan 1: $\pi_{\text{name}}(\sigma_{\text{age=20}\wedge\text{date.year=2012}}(\text{Students} \bowtie_{SID} \text{Reservations}))$

Plan 2: $\pi_{\text{name}}\sigma_{\text{age=20}\wedge\text{date.year=2012}}(\pi_{\text{SID,name,age}}(\text{Students}) \bowtie_{SID} \pi_{\text{SID,date}}(\text{Reservations}))$

Plan 3: $\pi_{\text{name}}((\pi_{\text{SID,name}}\ \sigma_{\text{age=20}}(\text{Students})) \bowtie_{SID} (\pi_{\text{SID}}\ \sigma_{\text{date.year=2012}}(\text{Reservations})))$

Query Plan Discussion

- Plan 1 computes the (expensive) join first, then removes disqualified result rows and columns.
- Plan 2 applies early projection, removing irrelevant columns early.
  - Notice that in addition to SID, which is needed for the join, all attributes needed by the final selection and projection operator must be preserved as well.
- Plan 3 applies both projection and selection as early as possible. Since those operators are much cheaper than the join, this usually results in huge cost savings: the expensive join is applied to much smaller inputs, in terms of the number of both rows and columns.
  - Early selection and projection are classic DBMS optimization techniques.

Operation Order in Practice

- In a DBMS, the optimizer will automatically apply early projection and selection.
- In Hadoop MapReduce, the programmer has to hard-code the right strategy.
  - In Reduce-side join, early projections and selections would be applied in the Mappers.
- In Spark, use of DataFrame and DataSet enables automatic application of early projection and selection, even if the program states a different order.
  - The Spark optimizer automatically re-orders the operations.
  - Spark (pair) RDDs (as of September 2018) do not support this automatic re-ordering of operations. Hence the programmer has to hard-code the best order.

Advanced Distributed Equi-Join Algorithms

- For a theoretical analysis with lower bounds and (asymptotically) matching upper bounds see [P. Beame, P. Koutris, and D. Suciu. Skew in parallel query processing. In Proc. ACM SIGMOD, pages 212-223, 2014].
- Our research group proposed the best skew-resistant distributed equi-join algorithm (as of late 2020). It comes with a constant-factor approximation guarantee relative to an optimal solution for random load assignment. [R. Li, M. Riedewald, and X. Deng. Submodularity of Distributed Join Computation. In Proc. ACM SIGMOD, pages 1237-1252, 2018]
- Both approaches address skew by partitioning "heavy hitter" groups, creating additional input duplicates in the process. This is discussed in the end of this module.

We explore next the semi-join, a close relative of the join.

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 semi-joins on multiple attributes and by allowing arbitrary Boolean functions over the attributes of pair (s_i, t_j), not just equality.

Semi-Join Example

- Recall the equi-join of Students and Reservations on the SID attribute, which produces pairs combining reservations with detailed student information. Assume the reservation details are not needed and one simply wants to analyze students who made a reservation, e.g., to find out if books are reserved mostly by more senior students.
- The (left) semi-join of Students and Reservations on the SID attribute would deliver this result. The left semi-join selects tuples from the left table (Students in the example expression) that have a matching tuple in the right table (Reservations in the example). By definition, the semi-join result is identical to the result of the regular equi-join, but without the Reservations attributes and without duplication of student information.

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

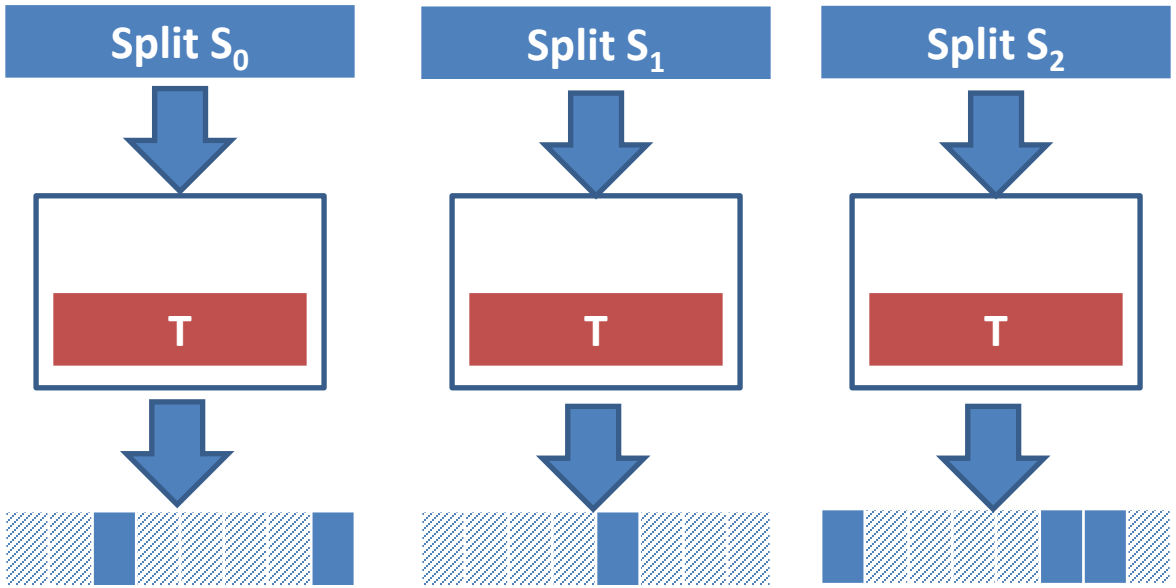
SID	Name	Age	GPA	BookID	Date
2	Bob	27	3.4	B10	01/17/12
2	Bob	27	3.4	B11	01/20/12
3	Carla	20	3.8	B11	01/18/12

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

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

Exact Semi-Join Computation

- 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 checks if it has a match in T. This is shown below.
  - Filters are an example of *per-record computations*, which can be implemented without shuffling (Map-only job). Since now the "filter" is the entire data set T, each task 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.
  - Interestingly, if the semi-join is implemented using partition + broadcast, it is identical to the algorithm derived based on the filter idea above.



Semi-Join in Spark

- Spark DataFrames and DataSets directly support semi-joins by simply selecting the join type as shown below.

```
studentDF.join(reservationDF, studentDF("SID") === reservationDF("SID"), "left-semi")
```

Toward Approximate Solutions

- As for the equi-join, broadcasting T to all worker machines will be expensive for large T. And if T does not fit in memory, expensive local I/O will further affect performance.
- How about sending a *subset*, e.g., a random sample, of T instead? A smaller subset reduces cost but has the drawback that some output tuples might be missing. This can be a problem, if the user is looking for rare events of high importance.
- Can we find a different solution that also works with a smaller version of T, but *guarantees that no semi-join output tuple will be missed*? It turns out that Bloom filters are the answer.

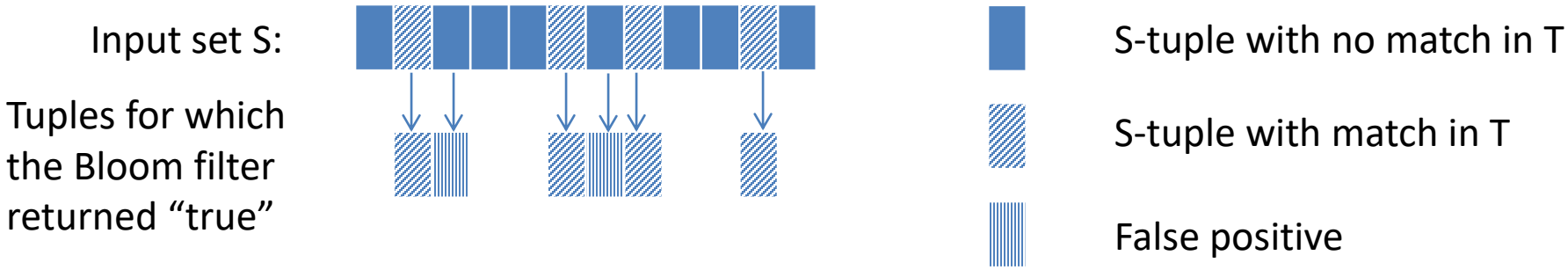
Bloom Filter Introduction

- The Bloom filter provides useful big-data analysis capabilities by representing very large sets in limited storage space.

- Using it for implementing semi-joins also illustrates a combination of algorithm design principles from per-record computation, join, and approximate computation.

Bloom Filter Properties

- A Bloom filter is a *probabilistic* data structure that determines if a tuple s occurs in a data set T. It is probabilistic in the following sense:
  - If the Bloom filter outputs "true," i.e., claims $s \in T$, then the answer might be incorrect. More precisely, one cannot be sure if s is in T or not. If the Bloom filter outputs "false," i.e., that $s \notin T$, then the answer is guaranteed to be correct.
  - In summary, the Bloom filter will not produce **false negatives** (S-tuples that are in T, but for which it outputs "false"), but it might produce **false positives** (tuples that are *not* in T, but for which it outputs "true").



Bloom Filter Challenge Question 1

- Are there other important uses for Bloom filters?

Bloom Filter Challenge Question 1

- Are there other important uses for Bloom filters?
  - 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 analyzing an event is expensive. To reduce processing cost, the analyst wants to focus on the critical events only. Let T be a set of signatures of critical events. Event $s \in S$ 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 cannot be critical. This helps cut cost.
  - However, it would not completely eliminate processing of non-critical events, because of the false positives.

Bloom Filter Challenge Question 2

- When using Bloom filters to eliminate irrelevant tuples, wouldn't we cut cost even more if the Bloom filter returned no false positives at all?

Bloom Filter Challenge Question 2

- When using Bloom filters to eliminate irrelevant tuples, 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. If T is very large, it might exceed available memory and it would be expensive to copy it to all machines.
  - 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 (space, lookup time), but increases the probability of false positives.
    - 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.

Let us see the Bloom filter in action.

Bloom Filter Construction

- For ease of presentation, let T be a set of integers.
- The Bloom filter's base data structure is an array of m bits. This bit array stores the encodings of all elements of set T.
- The Bloom filter also contains k independent hash functions h_1,…, h_k. Each hash function h maps a given integer i to another integer h(i) between 0 and m-1 (both inclusive). Intuitively, for input value i the hash function h turns on the bit at position h(i) in the bit array. For best results, the hash function output should be uniformly distributed in interval [0, m-1].
- The Bloom filter is constructed by initializing the bit array with zeroes and then inserting the tuples from T one-by-one. Tuple $t \in T$ is inserted by setting the bits in the bit array at positions $h_1(t)$, $h_2(t)$,…, $h_k(t)$ all to 1.

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

0	0	0	0	0	0	0	0	0	0

T = {2, 5, 6}

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

T = {2, 5, 6}

Insert 2 into the Bloom filter:

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

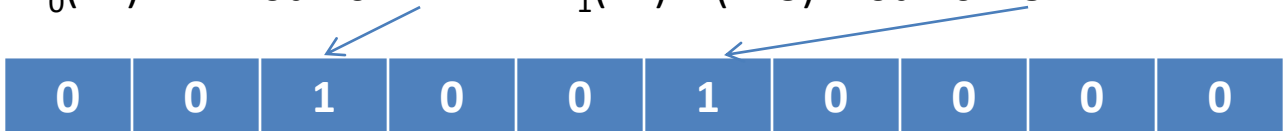
$h_0(t) = t \bmod 10$ $h_1(t) = (t+3) \bmod 10$

0	0	0	0	0	0	0	0	0	0

T = {2, 5, 6}

Insert 2 into the Bloom filter:

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



Insert 5 into the Bloom filter:

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

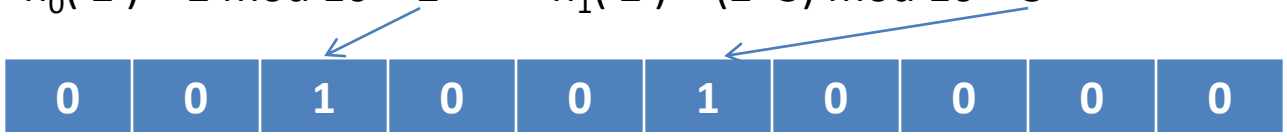
$h_0(t) = t \bmod 10$ $h_1(t) = (t+3) \bmod 10$

0	0	0	0	0	0	0	0	0	0

T = {2, 5, 6}

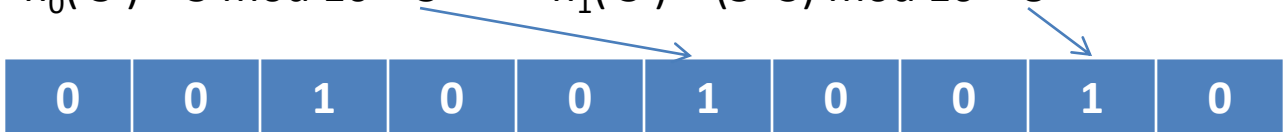
Insert 2 into the Bloom filter:

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



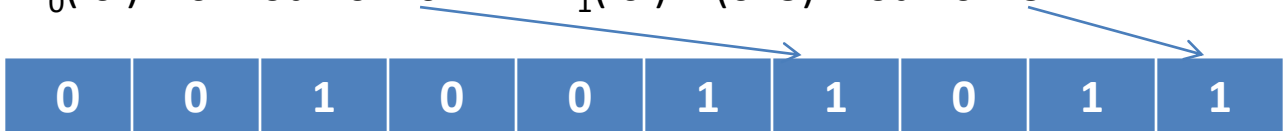
Insert 5 into the Bloom filter:

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



Insert 6 into the Bloom filter:

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



Querying the Bloom Filter

- After the Bloom filter is constructed, it is used to check if a given S-tuple s is in T. To determine the answer, the Bloom filter checks the bit values at positions $h_1(s)$, $h_2(s)$,…, $h_k(s)$ in the bit array. If all these bits are set, it will return "true." Otherwise it will return "false."
- In the example, for 2 and 18, the Bloom filter outputs the correct answer. It is clear by construction that the Bloom filter will always return "true" for S-tuples occurring in T.
- For 9 the Bloom filter's response is a false positive. Value 9 is not in T, but the Bloom filter cannot determine this because the corresponding bits are set.
- The example illustrates how bits set by elements of set T can result in *spurious* patterns that suggest the existence of other tuples in the set. Use of a larger bit array reduces the probability of spurious results.

Is value 2 in set T?

Bloom filter output:

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

true

Is value 2 in set T?

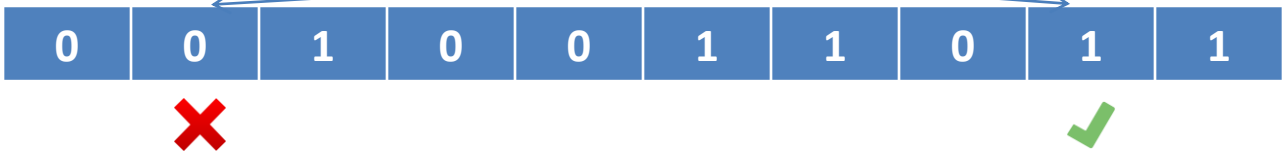
Bloom filter output:

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

true

Is value 18 in set T?

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



false

Is value 2 in set T?

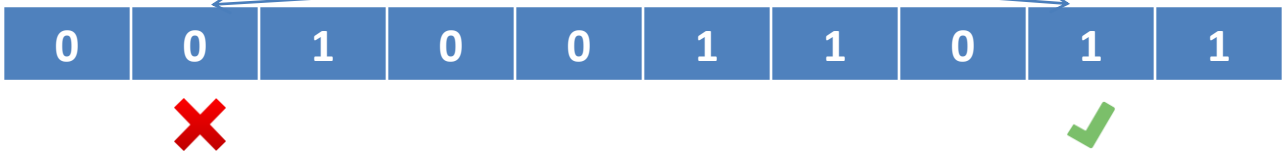
Bloom filter output:

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

true

Is value 18 in set T?

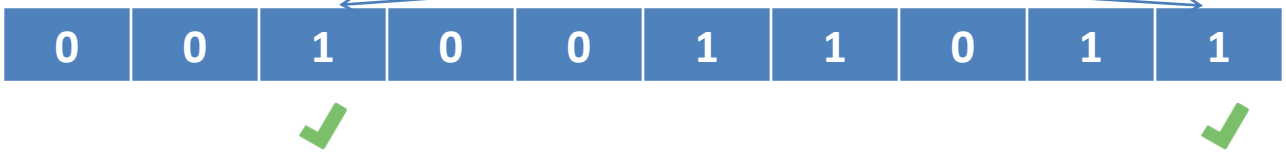
$h_0(18) = 18 \bmod 10 = 8$ $h_1(18) = (18+3) \bmod 10 = 1$



false

Is value 9 in set T?

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



true

Approximate Semi-Join with Bloom Filter in MapReduce

- 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 dataset T from file cache into B
    B = new BloomFilter
    for each tuple t in File Cache
      B.insert( t )
  }

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

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

Summary

- When information is combined across different data sets, distributed-algorithm design becomes more challenging.
- The hash + shuffle join often works well in practice but requires two rounds of computation to group data by join-attribute value.
- When one of the inputs fits in memory, the partition + broadcast algorithm can compute the join in a single round, avoiding shuffling. On the other hand, the smaller input must be broadcast to all worker machines.
- Big-data analysis tends to be resource intensive, but resource consumption can be reduced when the user settles for approximate answers. The Bloom filter provides approximate information about set membership. Its size is tunable, at the cost of a higher false positive rate. On the other hand, it never returns false negatives.

References

- P. Beame, P. Koutris, and D. Suciu. Skew in parallel query processing. In Proc. ACM SIGMOD, pages 212-223, 2014
  - https://scholar.google.com/scholar?cluster=15843596658611202 90&hl=en&as_sdt=0,22
- R. Li, M. Riedewald, and X. Deng. Submodularity of Distributed Join Computation. In Proc. ACM SIGMOD, pages 1237-1252, 2018
  - https://dl.acm.org/citation.cfm?id=3183728

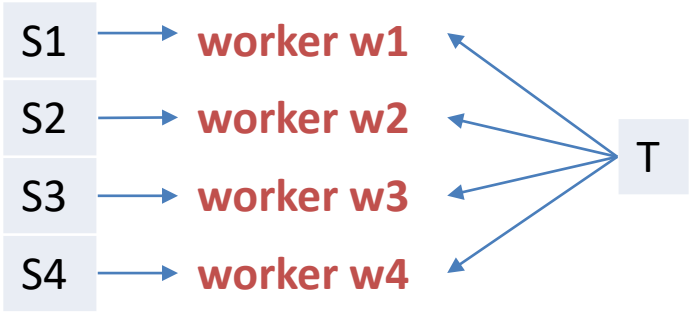
For those interested in more advanced material, we now present an overview of our paper on dealing with skew in equi-joins [R. Li, M. Riedewald, and X. Deng. Submodularity of Distributed Join Computation. In Proc. ACM SIGMOD, pages 1237-1252, 2018].

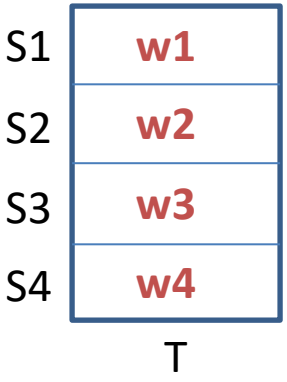
Skew Matters

- Overloaded workers delay job completion. When does this become a problem?
  - Large join groups: This can happen for commonly observed real-world distributions such as Zipf, Power Law, and the 80-20 rule.
    - Zipf and Power Law distributions are common in social networks (a few people have many followers, many have a few).
    - 80-20 captures the idea that often 20% of entities capture 80% of some value of interest, e.g., income distribution.
  - Scaling out: Skew increases with increasing number of machines, because the size of the largest groups becomes larger *relative* to average load per machine.
- Solution: partition the heavy hitters, i.e., join groups that are "large" and hence create a high load. This trades input duplication for better load balance as we discuss next.
  - Which groups should be partitioned and how?

Previous Work

Partition-Broadcast



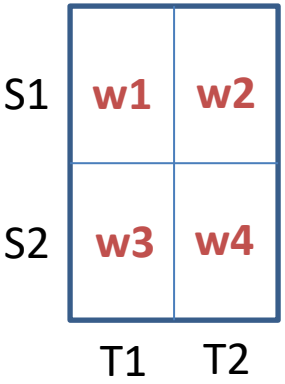


S: #workers partitions

T: #workers copies

Heuristic without optimality guarantees

Beame, Koutris, Suciu 2014



Partitioning determined by optimization goal:

Minimize max input assigned to any worker

Solution: optimal within polylog factor of #workers

Note: Constants do not matter for their analysis, but they do for query optimization.

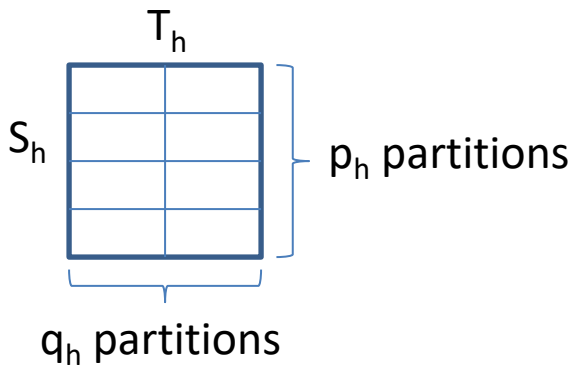
Our Solution

- Computation model: random assignment of (sub) groups to workers. When does this model apply?
  - When we choose a hash function randomly from a family (universal hashing).
  - When a given system uses a “blackbox” hash Partitioner, i.e., we do not know the function.
- We define load as a weighted sum of input and output
- If two partitionings have the same load expectation, we prefer the one with lower variance. It has the same total cost, but a higher probability of better load balance.
  - If we can find the minimum-variance partitioning for a given load expectation threshold, then we can generate a sequence of Pareto-optimal expectation-variance combinations.
  - From those combinations, we then pick the one that minimizes estimated running time.

Toward an Optimization Problem

- We solve the problem of minimizing load variance, subject to an upper limit on load expectation.
  - Intuitively, this maximizes load balancing for a given upper bound on total load.
- To solve this problem, we express load expectation E and variance V in terms of partitioning parameters.
  - Let p_h and q_h denote the number of partitions of input S and T, respectively, for heavy hitter join attribute value h. Setting p_h and q_h to 1 implies that the join group is not further partitioned.

Rectangular Group Partitioning



This applies to "general" join problems:

select … from S, T where S.A = T.B

select … from S as S1, S as S2 where S1.A = S2.B

$$\text{load} = \gamma I + \alpha O$$

$$E[L] = \frac{1}{w}\sum_{h\in H}(\gamma q_h|S_h| + \gamma p_h|T_h| + \alpha|S_h||T_h|)$$

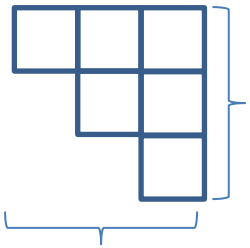
$$V[L] = \frac{w-1}{w^2}\sum_{h\in H} p_h q_h \left(\gamma\frac{|S_h|}{p_h} + \gamma\frac{|T_h|}{q_h} + \alpha\frac{|S_h||T_h|}{p_h q_h}\right)^2$$

L: random variable capturing load on a worker machine

w: number of worker machines

H: set of heavy hitters

Triangular Group Partitioning



For natural self-join:
select … from S as S1, S as S2 where S1.A = S2.A

$$\text{load} = \gamma I + \alpha O$$

$$E[L] = \frac{1}{w}\sum_{h\in H}(\gamma p_h|S_h| + \alpha|S_h|^2)$$

$$V[L] = \frac{w-1}{w^2}\sum_{h\in H}\left(2-\frac{1}{p_h}\right)\left(\gamma|S_h| + \frac{\alpha}{p_h}|S_h|^2\right)^2$$

L: random variable capturing load on a worker machine
w: number of worker machines
H: set of heavy hitters

Problem Analysis

- We carefully analyze the properties of the load variance and expectation formulas to find efficient solutions.
- Monotonicity and submodularity of the objective function (negative load variance) enable us to apply powerful greedy heuristics that provide strong near-optimality guarantees for the solutions they find.

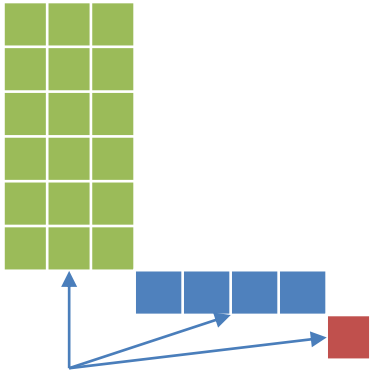
Load-Variance Properties

- Monotonically decreasing in p_h and q_h, if
  - $\alpha \geq \gamma \frac{p_h+1}{|S_h|}$ and $\alpha \geq \gamma \frac{q_h+1}{|T_h|}$ (rectangular partitioning)
  - always (foreign-key join)
  - $\alpha \geq \gamma \frac{p_h}{|S_h|}$ (triangular partitioning)
- Submodular as well (for negative variance)
  - Slightly stronger requirement for general rectangular case: $\alpha \geq \sqrt{2}\gamma \frac{p_h+1}{|S_h|}$ and $\alpha \geq \sqrt{2}\gamma \frac{q_h+1}{|T_h|}$

Variance Optimization

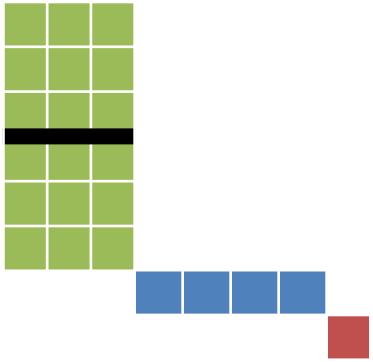
- Maximizing negative load variance subject to an upper limit on expectation is a submodular maximization problem with Knapsack constraints.
- A simple greedy solution guarantees a factor $\frac{(1-e^{-1})}{2} \approx 0.32$ approximation.
  - The greedy algorithm increases the number of partitions with the greatest ratio of variance reduction to input duplication increase.
  - The cost of an iteration of this algorithm is linear in the number of groups considered.
- Another greedy solution guarantees a stronger $(1 - e^{-1})$ approximation, but its computation cost is higher by a factor that is cubic in the number of groups considered.

Example Execution

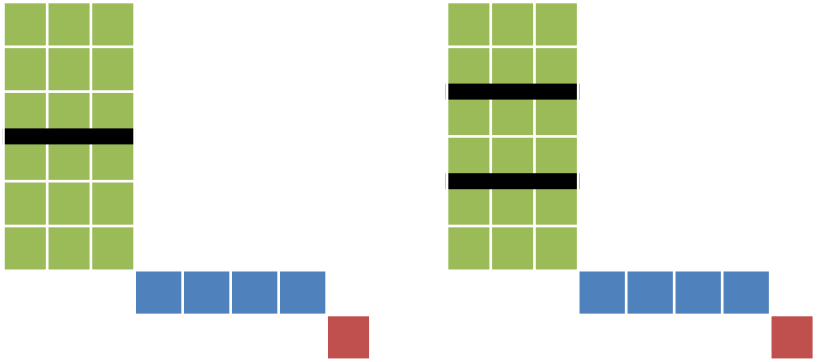


Join groups

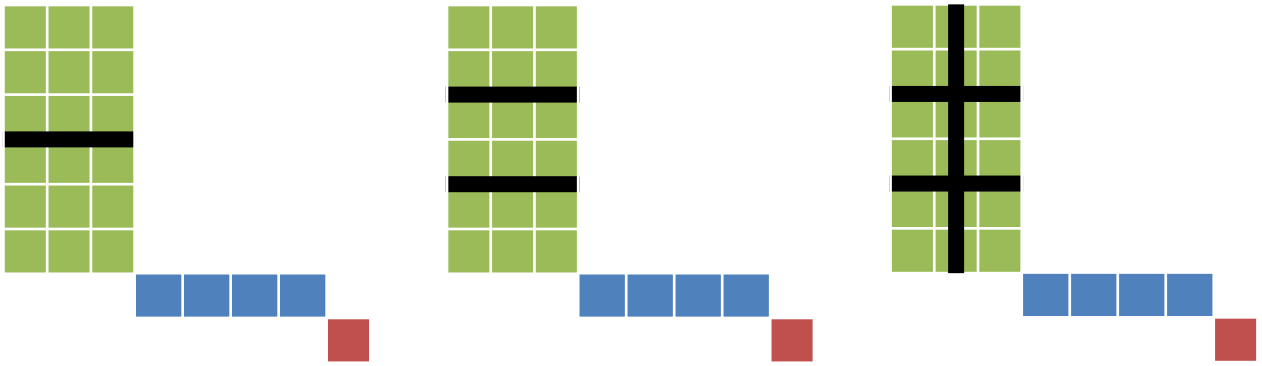
Example Execution



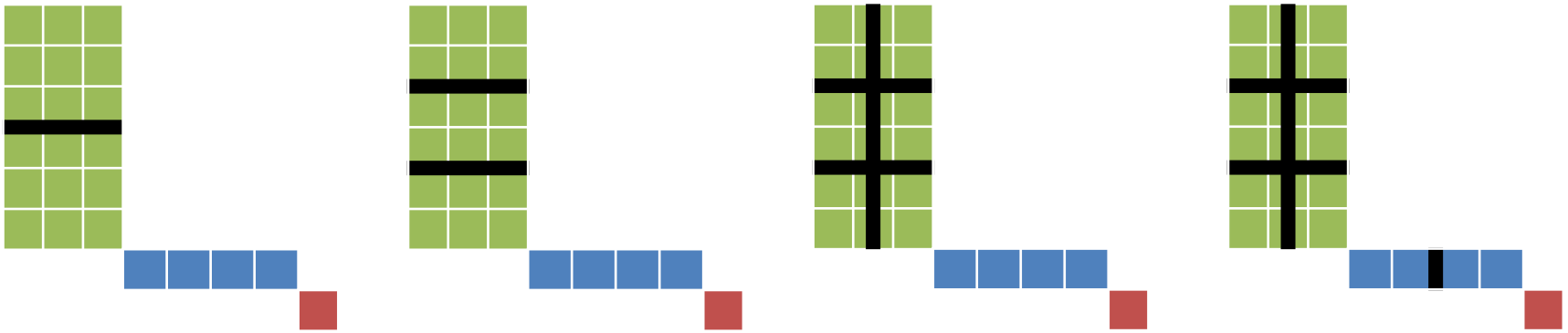
Example Execution

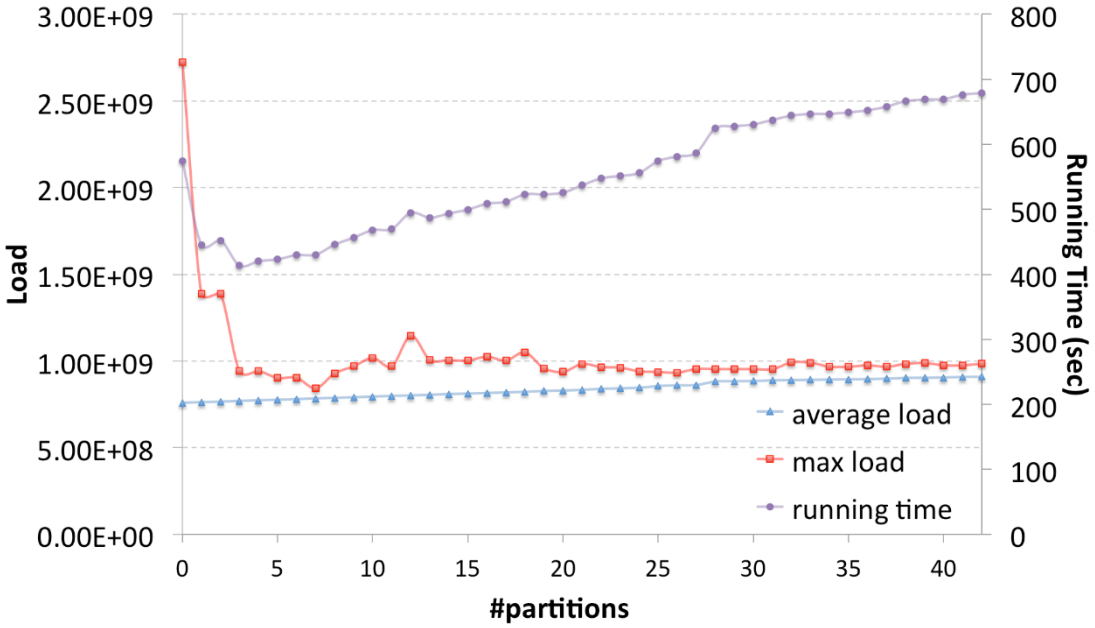


Example Execution



Example Execution





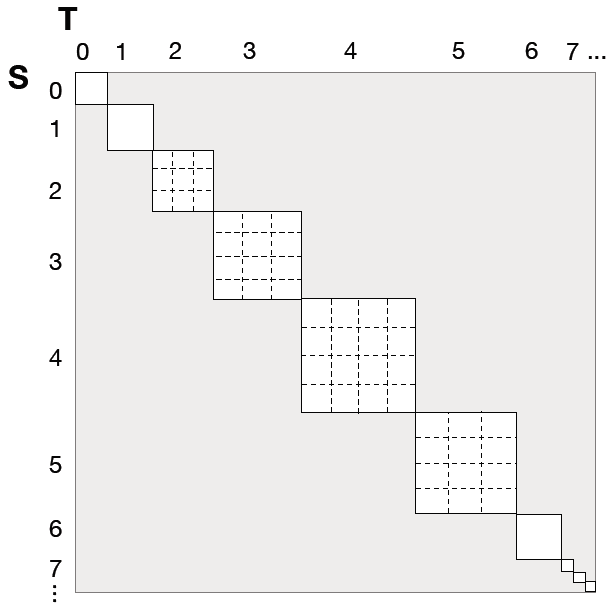
Running-Time Minimization

- When do we stop partitioning?
  - More partitions improve load balance but increase total load. When is the benefit of better load balance not worth the increase in total cost anymore?
- We need a cost model—actually a makespan-prediction model! (Makespan captures end-to-end running time of a schedule of tasks.) This model needs to do two things:
  - Estimate shuffle time based on data size.
  - Estimate maximum "local" processing time on a worker.
- Note: we can replace random load assignment with deterministic assignment in the algorithm.
  - The Least-Loaded Decreasing (LLD) heuristic guarantees a 4/3 approximation of optimal.
  - It stops partitioning earlier than for random assignment, resulting in better performance in practice.

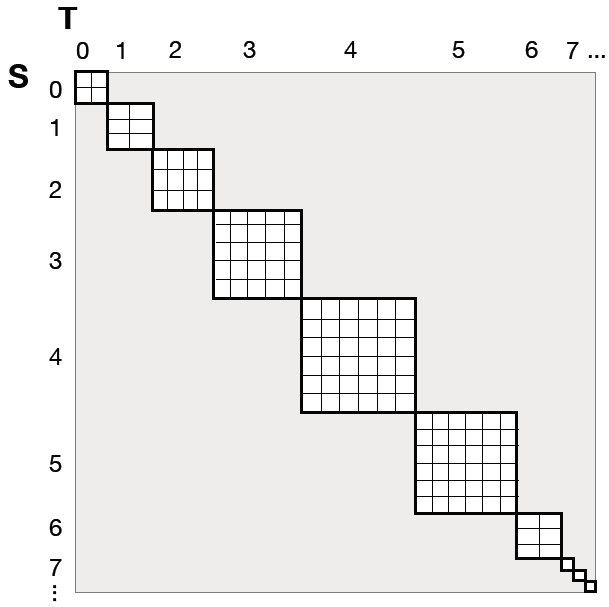
Experiments

- Cluster14: 8 machines (quad-core 2.4Ghz, 8GB RAM), 2 cores per worker used
- EMR50: 51 m1.medium machines (1 vCPU, 3.75GB RAM, moderate network performance)
- Various synthetic and real data sets
  - Up to 2 billion input tuples
  - 2 to 1657 columns
  - Output size between 10^7 and $1.6*10^{13}$ tuples
- Queries: JOIN (full result), JOIN-AGG (1 output per group), and JOIN-Fib (Fibonacci(1000) computed per output tuple to simulate expensive per-output processing)

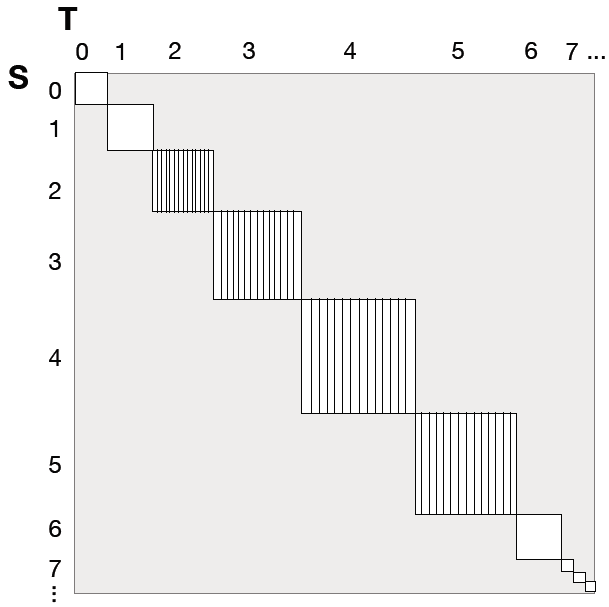
Partition Example (JOIN, cloud-200k, Cluster14)



BKS



ExpVar



PaBr

JOIN on Cluster14 (Running Time in Sec)

Data sets	ExpVar	NoPar	PaBr	PaBr+	BKS	BKS+
Zipf-100k-1.0-1g*	1570	1837	1764	1660	1789	1789
Zipf-500k-1.0-1g*	1661	1961	1968	1728	1817	1817
Zipf-90k-1.0-1g	1789	3198	2059	1900	1873	1869
Zipf-200k-1.0-50m	755	5005	1137	769	1181	913
Zipf-285k-1.0-10m	1352	7965	2160	1324	1507	1448
Zipf-100k-1.0	208	1270	299	201	240	217
Zipf-500k-0.25	1765	4599	1937	1756	3199	1808
Zipf-500k-0.5	2125	7676	3663	2048	3668	2535
Zipf-500k-1.0	4033	27455	5389	4114	5338	4310
Cloud-200k	1144	4448	1612	1204	1594	1215
TPC-H-cust-nation	433	1294	448	448	597	467

Ratio to best competitor: 0.95…1.04

Ratio to best competitor without cost model: 0.58…0.90

Self-JOIN on Cluster14

Data sets	ExpVar	CIK	CIK+
Sj-zipf-200k-1.0-50m	843	1434	1395
Sj-zipf-500k-0.25	1923	2827	2827
Sj-cloud-200k	1220	2263	2263

Ratio to best competitor: 0.54…0.68
Ratio to best competitor without cost model: 0.54…0.68

JOIN-AGG on Cluster14

Data sets	ExpVar	NoPar	PaBr	PaBr+	BKS	BKS+
Zipf-5m-0	333	378	378	325	378	365
Zipf-5m-[0.5,0]	321	418	474	304	391	379
Zipf-5m-[1,0]	351	495	894	474	461	359
Zipf-5m-0.5	353	591	693	498	494	454
Zipf-5m-1.0	562	1502	989	924	694	643
Cloud-5m	314	1291	473	446	329	325
ebird-all	1367	3000	3222	3000	1927	1871
ebird-basic	598	1051	1842	1051	939	907

Ratio to best competitor: 0.66…1.06
Ratio to best competitor without cost model: 0.64…0.88

JOIN-AGG on EMR50

Data sets	ExpVar	NoPar	PaBr	PaBr+	BKS	BKS+
BKS-bad-case	2072	3212	5419	2072	3514	2072
Cloud-5m	440	3652	1088	1088	546	546
ebird-basic	380	3410	1022	684	464	425

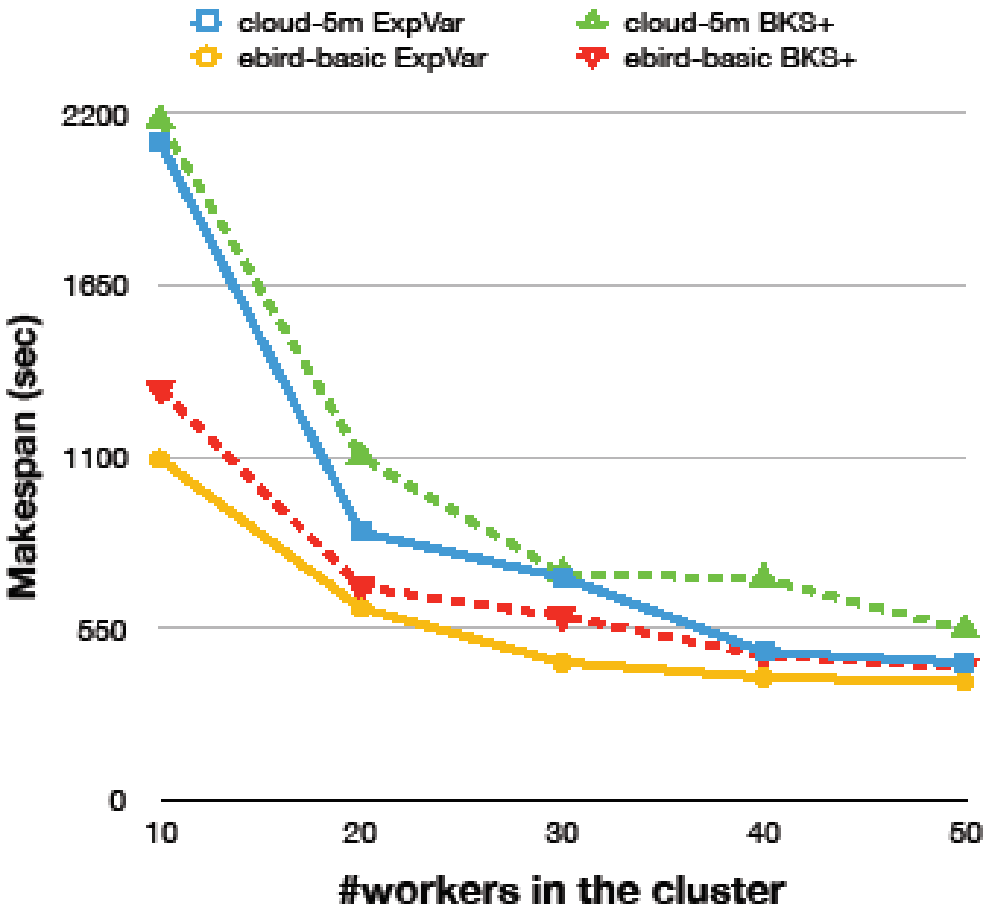
Ratio to best competitor: 0.81…1.00
Ratio to best competitor without cost model: 0.65…0.82

JOIN-Fib on Cluster14

Data sets	ExpVar	NoPar	PaBr	PaBr+	BKS	BKS+
Zipf-90k-1.0-1g	1858	3529	3560	1950	2019	1933
Zipf-500k-1.0	3983	14509	7246	4057	7196	4136
Cloud-200k	1965	7014	2871	1981	3401	2158

Ratio to best competitor: 0.96…0.99
Ratio to best competitor without cost model: 0.55…0.92

Speedup on Amazon Cloud



Conclusions and Future Work

- We started our analysis with a random-load-assignment model. This resulted in an algorithm for near-optimal greedy partitioning of heavy-hitter join groups to address input skew.
- The algorithm also works well for deterministic load assignment.
  - It optimizes for the given problem, not worst-case complexity.
  - Our partitioning algorithm creates easy-to-distribute homogeneous pieces. This plays a role in explaining why the partitions also give good results for deterministic load assignment.
- Our algorithm only needs count statistics for large groups. In contrast to most previous work, no pre-defined heavy-hitter threshold is needed.
- Next step: generalize to skew-driven partitioning for multi-way joins, i.e., joins between more than two inputs.