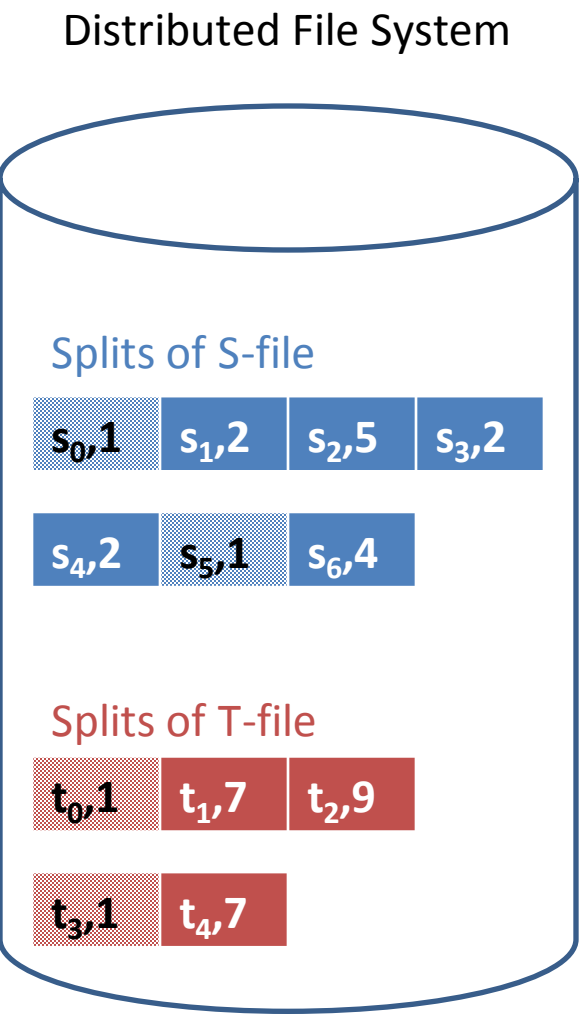
1. Joins in MapReduce

- Wir beginnen mit einem einfachen Equi-Join von $S=(s_1,s_2,\ldots)$ und $T=(t_1,t_2,\ldots)$, beide mit Attribut A:
  - Der Equi-Join $S \bowtie_A T$ ist die Menge aller Paare (s_i,t_j) für die gilt $s_i.A=t_j.A$.
- Es handelt sich auch um einen Equi-Join, wenn die Bedingung eine Konjunktion von solchen Gleichheitsbedingungen auf mehreren Attributen der Eingaberelationen ist.

Verteilte Joinberechnung



- Wie kann garantiert werden dass ein S-Tupel in einem Split der S-Datei mit allen T-Tupeln verknüpft wird, die irgendwo in der T-Datei sein könnten?

- Idee 1: Sende alle Tupel mit dem gleichen A-Wert zum gleichen Task, welcher dann die Verknüpfung ausführt.
- Idee 2: Stelle T auf allen an der Berechnung beteiligten Maschinen zur Verfügung. Tasks lesen nur S-Tupel und greifen auf die (indexierte) lokale Kopie von T zu.

2. Idee 1 in MapReduce: Reduce-Side Algorithmus

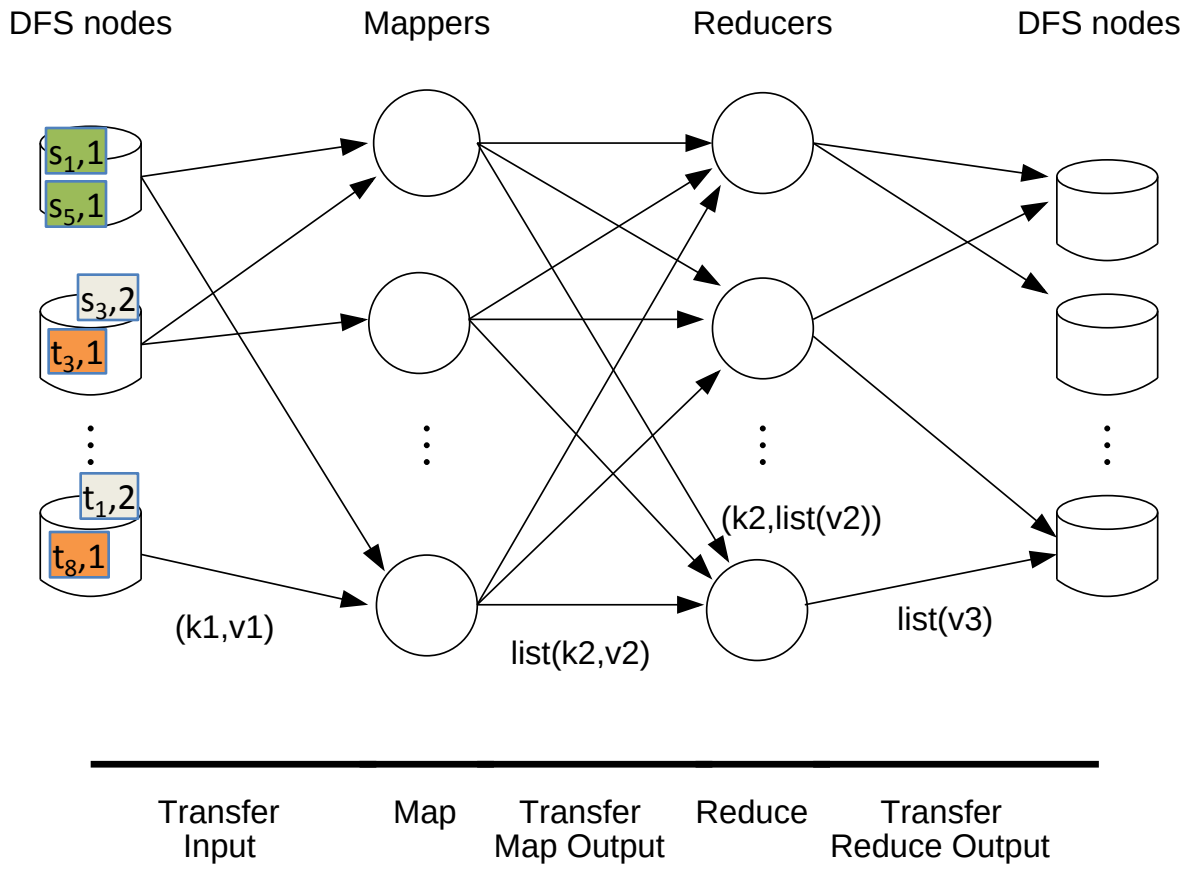
- S und T werden in Gruppen nach Attribut A aufgeteilt. Alle Tupel in der gleichen Gruppe können dann verbunden werden.
- In MapReduce wird das dadurch erreicht das Attribut A als Schlüssel verwendet wird.
- Für jeden Wert von A wird dann die Reduce Funktion ausgeführt.

MapReduce Algorithmus

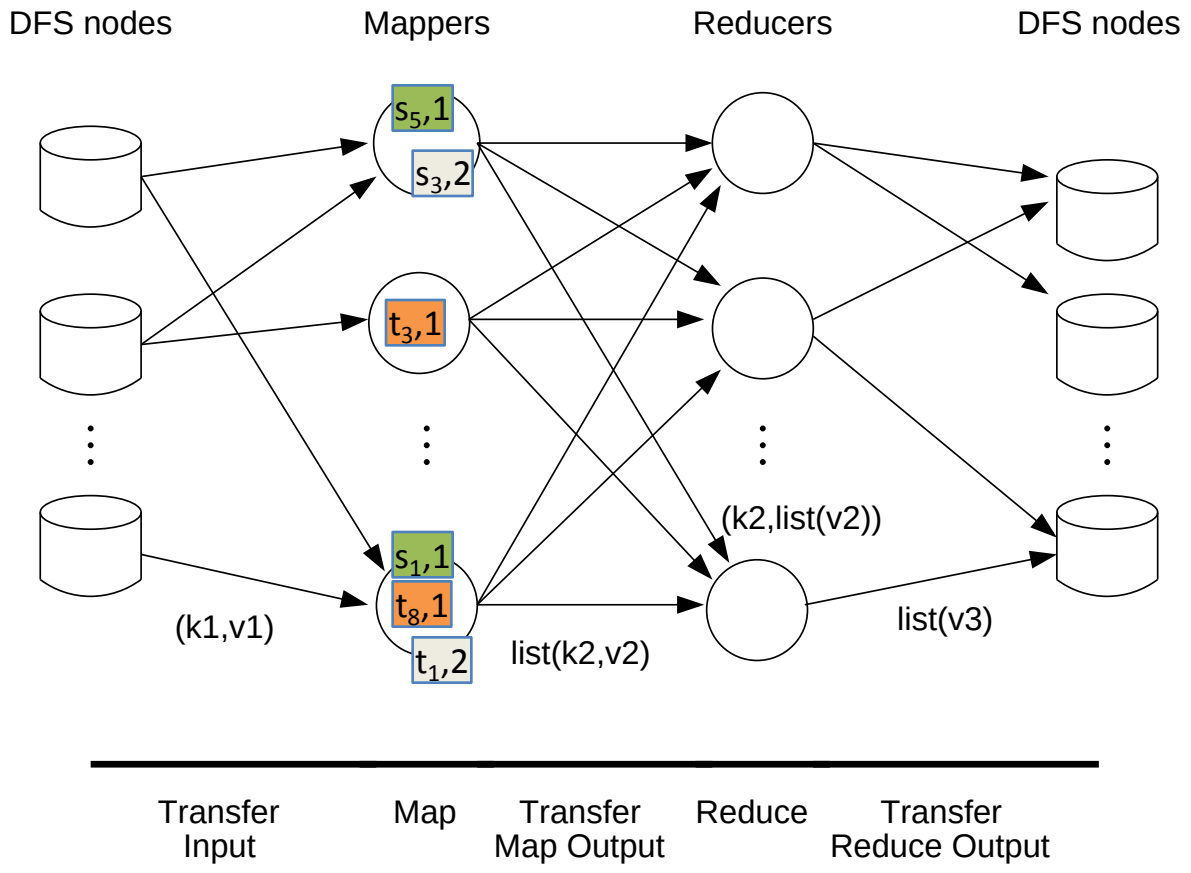
```
map( …, tuple x ) {
  if (x is from S)
    emit( x.A, (x, “S”) )
  else    // x is from T
    emit( x.A, (x, “T”) )
```

```
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) )
}
```

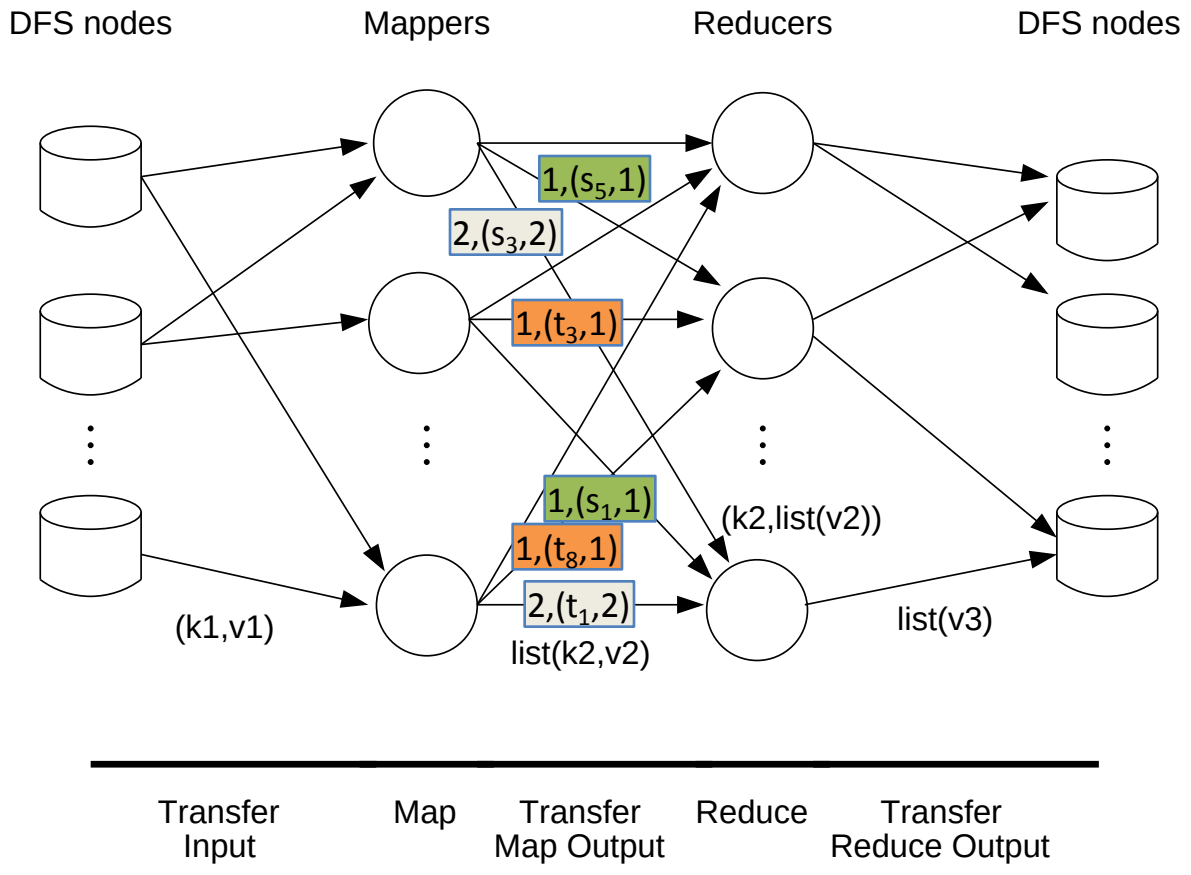
- Im Beispiel sind Tupel ID und Wert von Joinattribut A dargestellt.
- Die Eingabetupel sind verstreut über verschiedene Splits und können von verschiedenen Mappern bearbeitet werden.



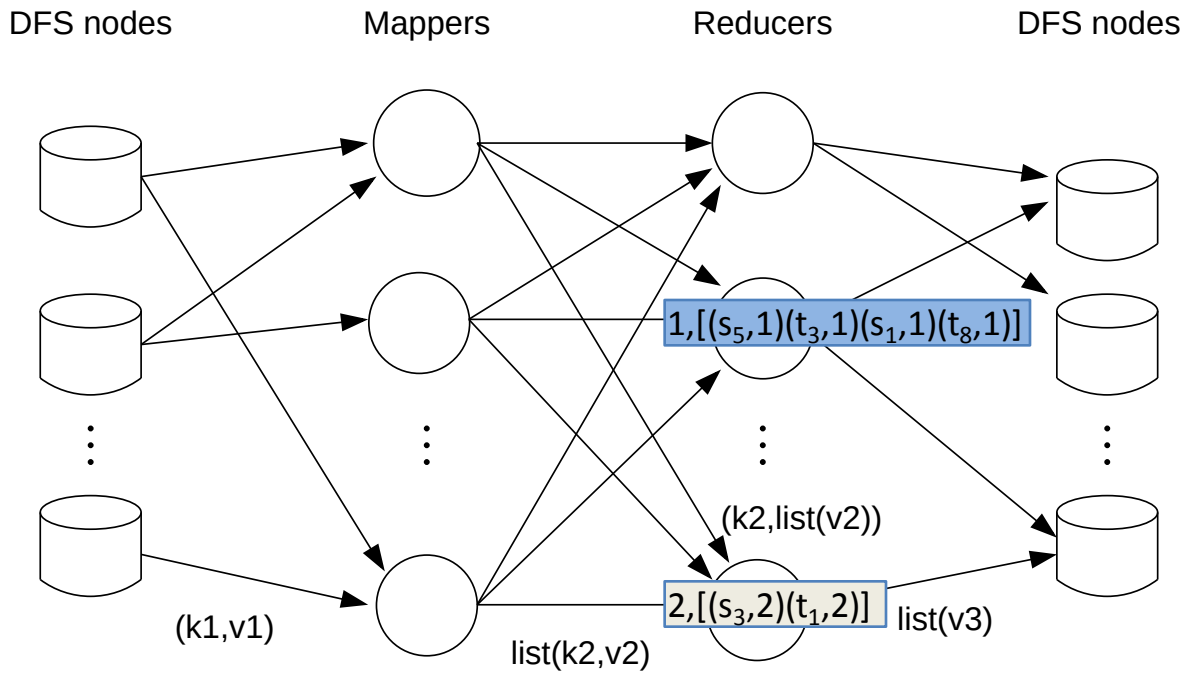
- Die Eingabetupel werden vom verteilten Dateisystem zu den Mappern transferiert.



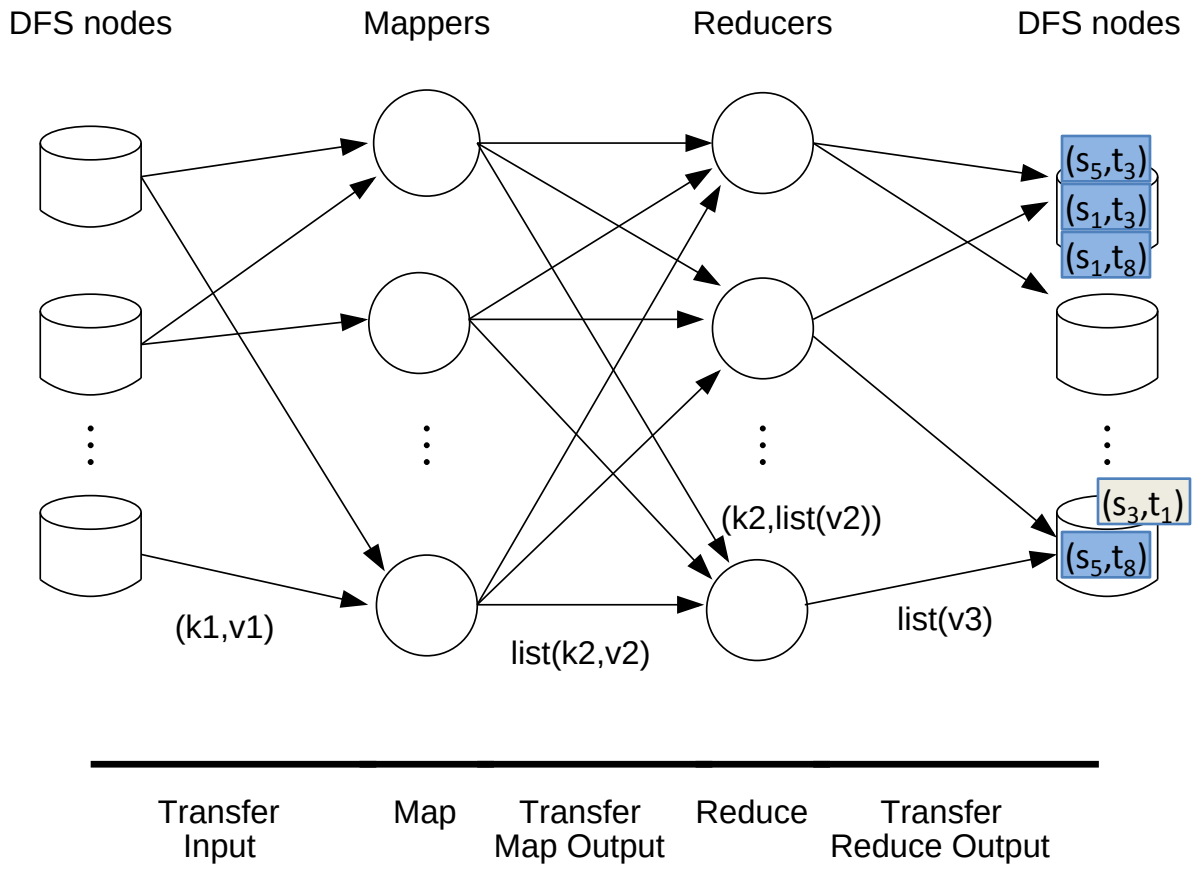
- Die Map Funktion gibt jedes Tupel als Wert, mit seinem A-Wert als Schlüssel, aus.



- Im Beispiel erhalten nur zwei Reducer die Ausgabe von den Mappern—einer für A-Wert 1, der andere für A-Wert 2.



- Die Reducer schreiben ihre Ausgabe zurück auf das verteilte Dateisystem.



Diskussion

- Der Reduce-side Join Algorithmus vermeidet es, Paare von nicht verknüpfbaren Tupeln zu testen—so wie ein Hash Join.
- Er gruppiert die Eingabe nach dem Joinattribut (lineare Kosten), dann verknüpft er die Tupel in der gleichen Gruppe. Diese lokale Operation ist ein kartesisches Produkt.
- Leider gibt es mehrere Nachteile…

Diskussion

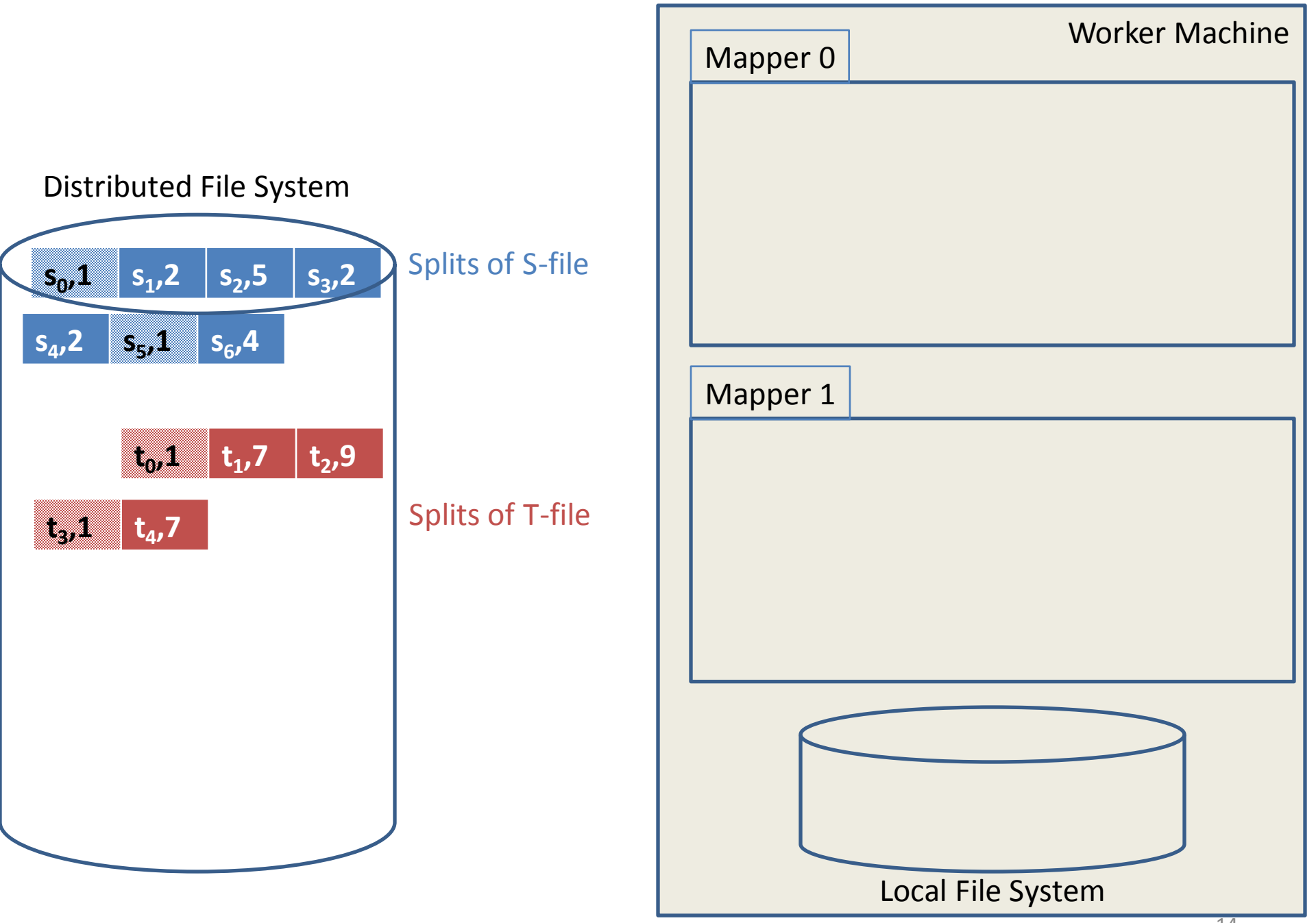
- Schlechte Lastverteilung bei schiefer Verteilung der Joinattributwerte: Alle Tupel mit dem gleichen Joinattributwert müssen im gleichen Reduce Aufruf bearbeitet werden.
- Skaliert nicht für Joinattribute mit kleiner Anzahl verschiedener Werte: Bei k verschiedenen Werten gibt es höchstens k Partitionen.
- Die Idee des Joinattributes als Schlüssel generalisiert nicht zu anderen Joinbedingungen, z.B. Ungleichheit ($S.A < T.A$) oder Band-Joins ($|S.A - T.A| < \varepsilon$).
- S und T werden zweimal übertragen: vom verteilten Dateisystem zu Mappern und von Mappern zu Reducern.

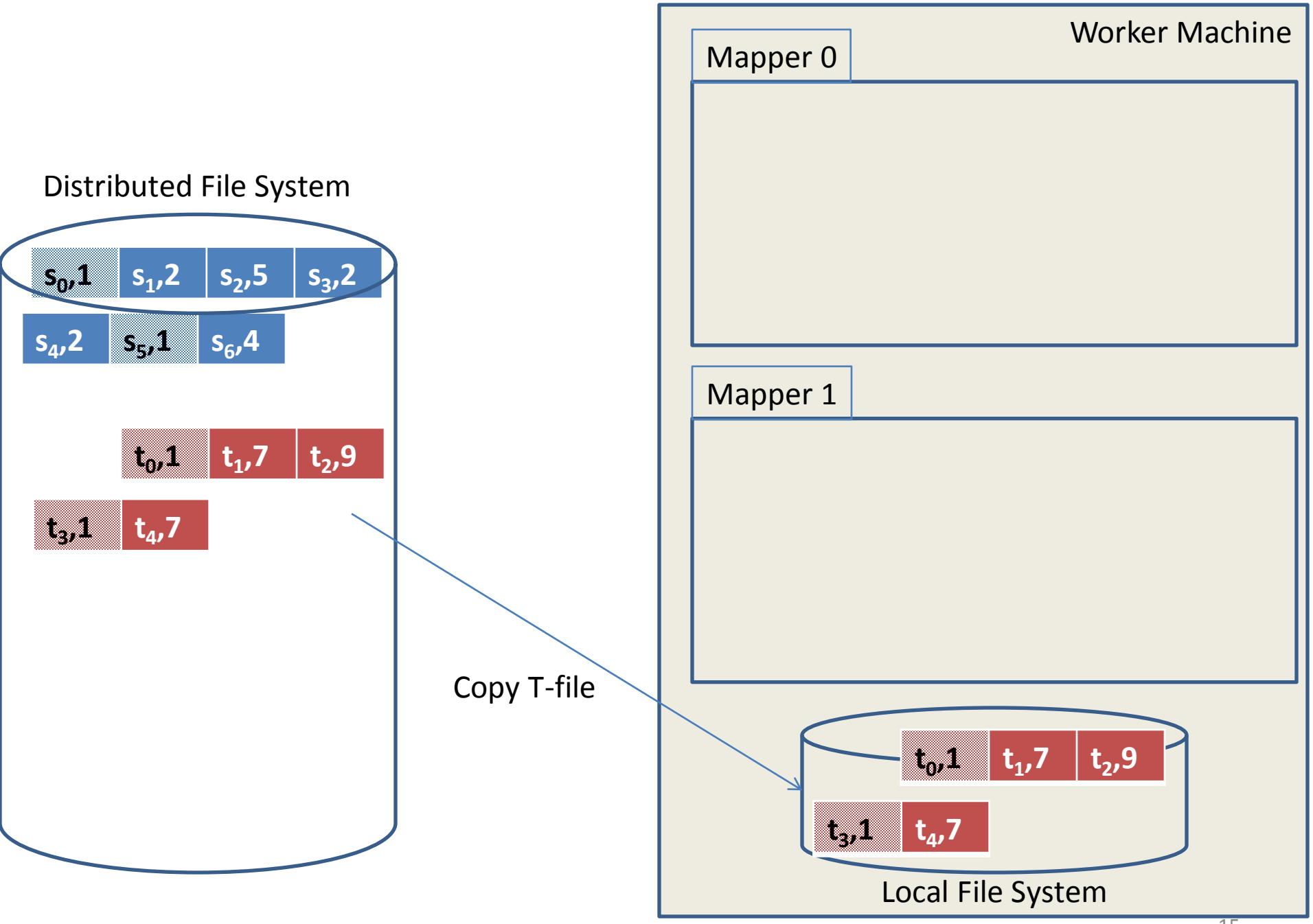
3. Idee 2: Map-Only Join (“Replicated Join”)

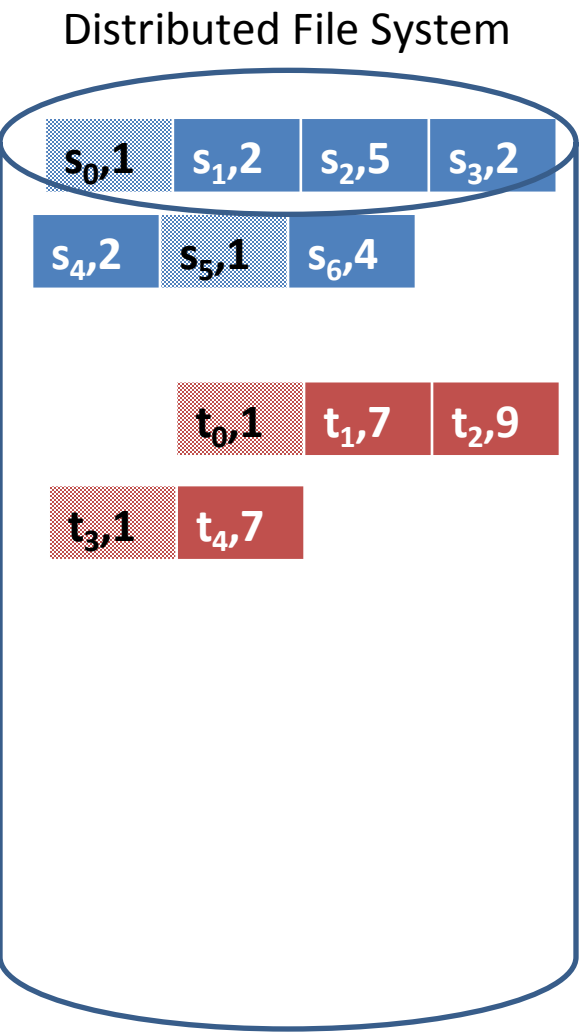
- Datei T wird mithilfe des verteilten Dateipuffers (Hadoop class *DistributedCache*) an alle Knoten verteilt.
- Die setup Funktion des Map Tasks lädt T aus dem Puffer und kreiert eine Task-lokale Datenstruktur, z.B. einen Hash-Index.
- Die Map Funktion bearbeit nur S als Eingabe. Für jedes eingelesene S-Tupel benutzt sie den Index um alle passenden T-Tupel zu finden.

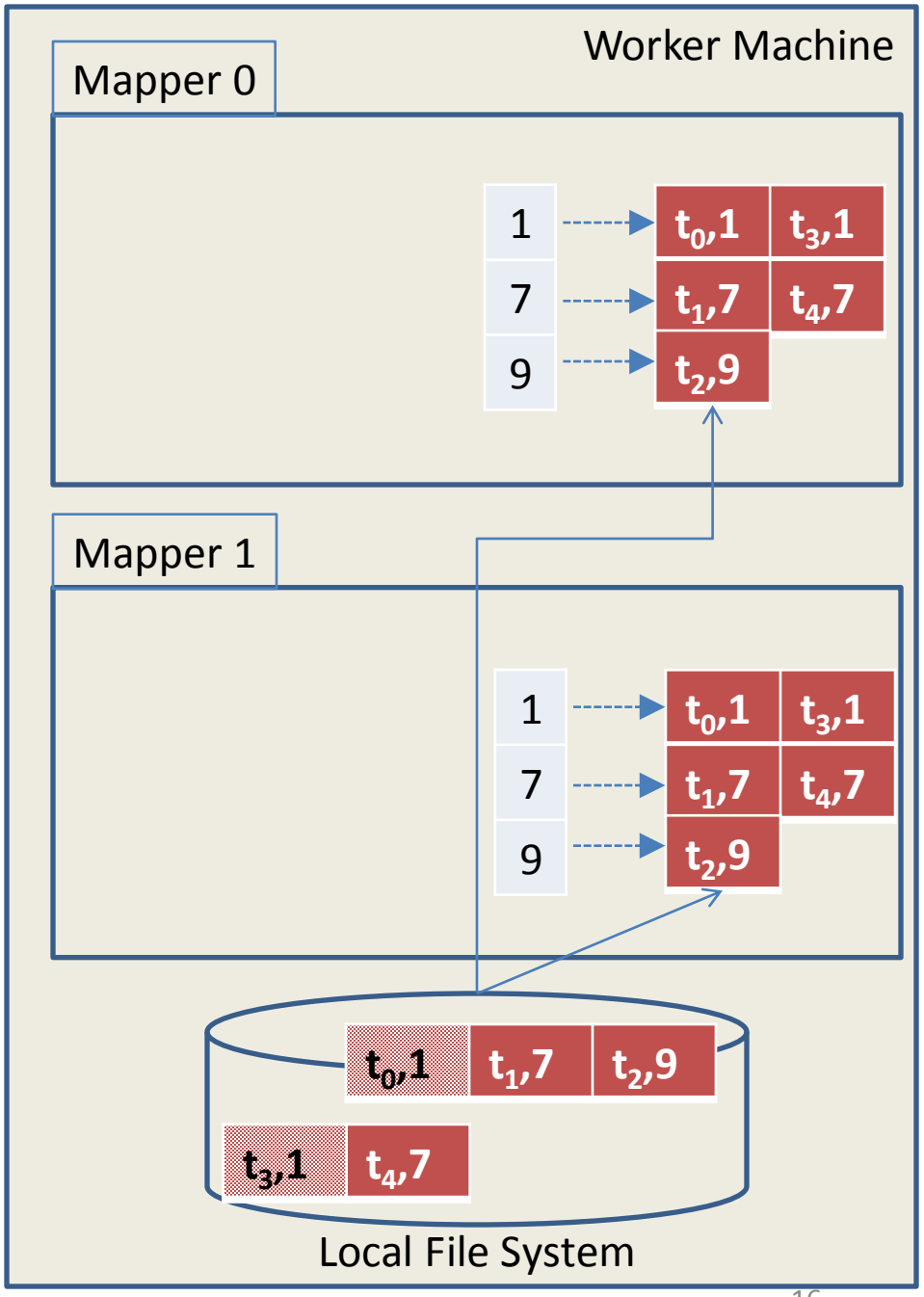
MapReduce Algorithm

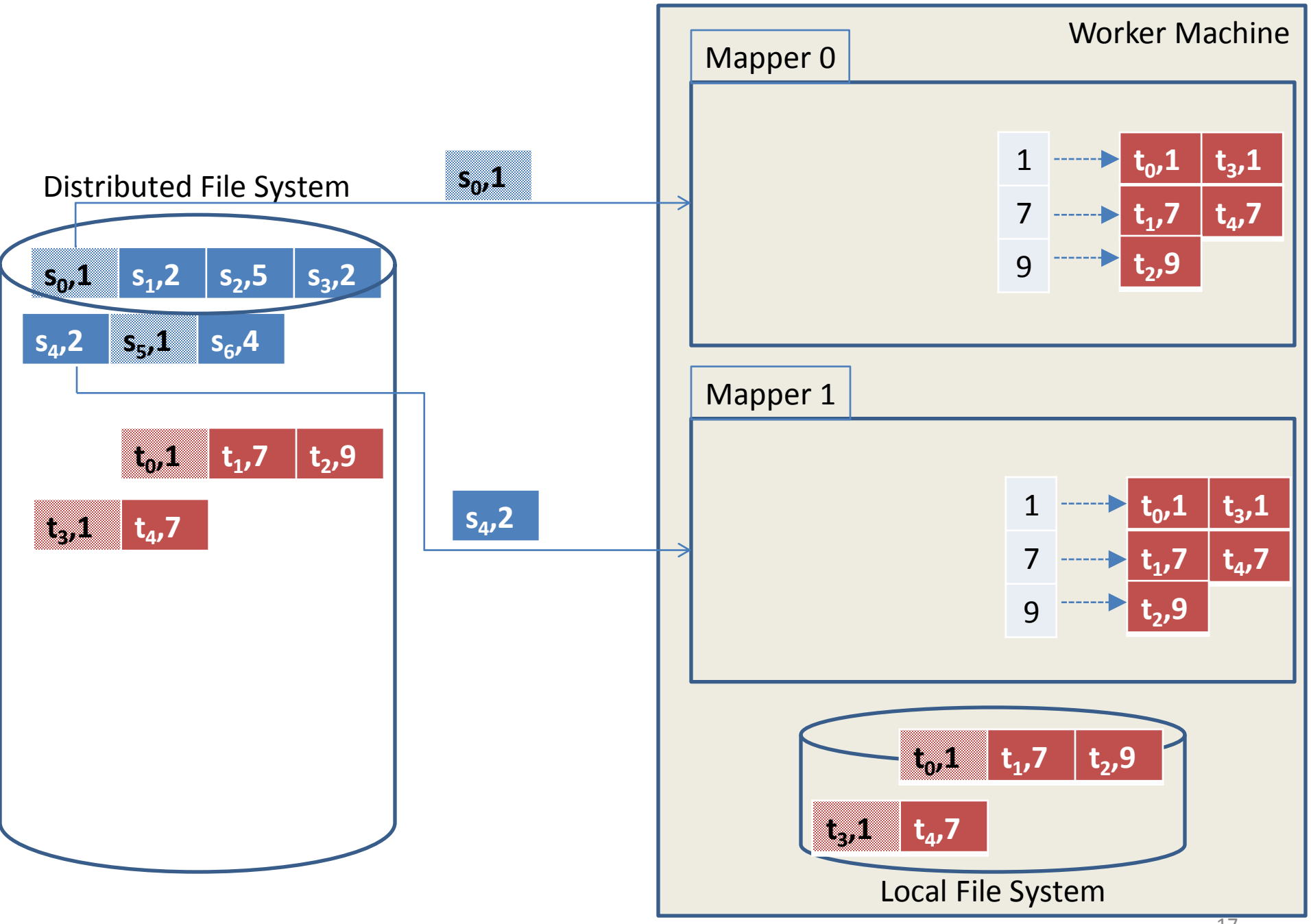
```
Class Mapper {
  // Index H maps a join attribute value to all T-tuples with that value
  hashIndex H

  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 }
}
```

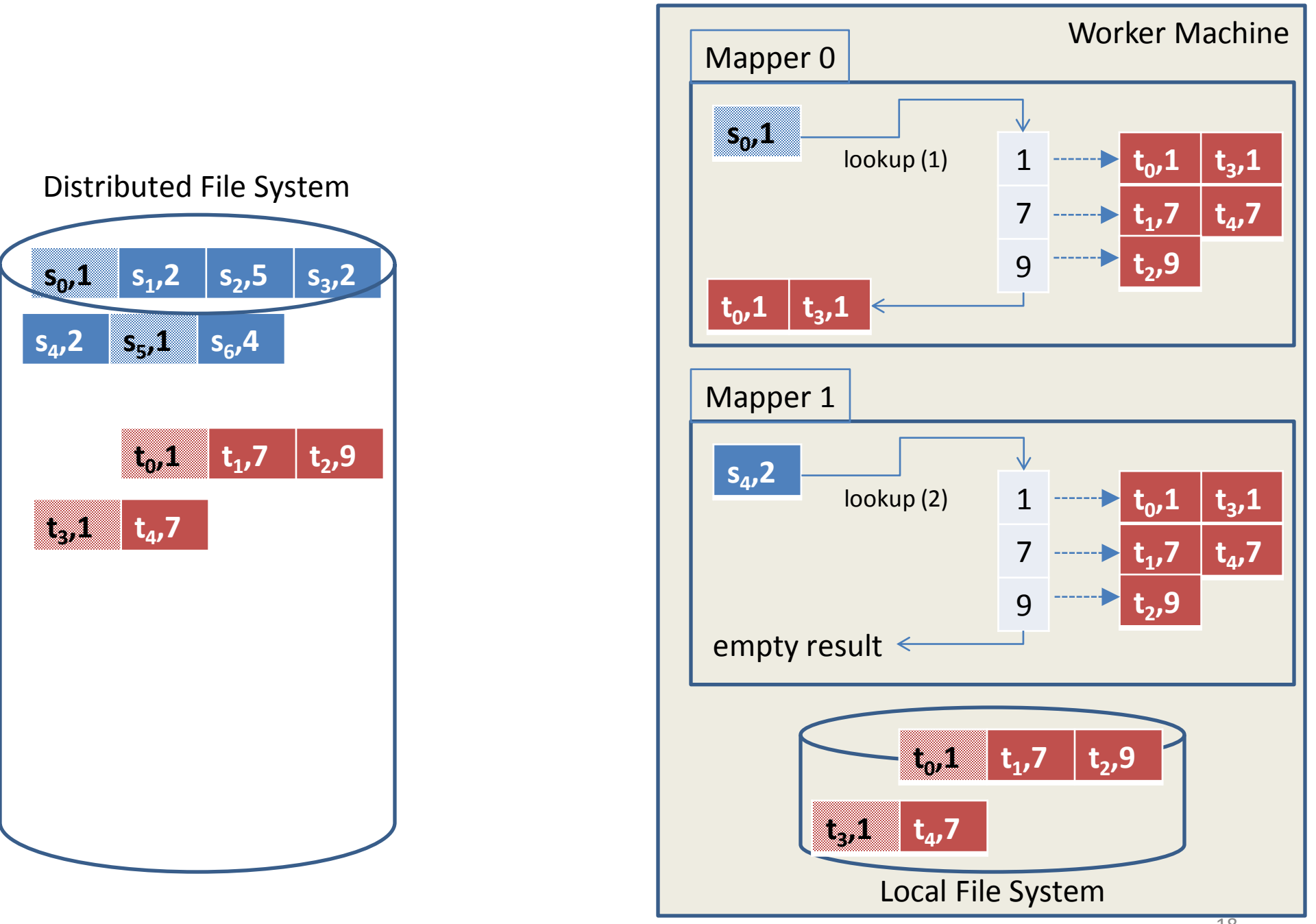

Distributed File System
$s_0,1$
$s_1,2$
$s_2,5$
$s_3,2$
Splits of S-file
$s_4,2$
$s_5,1$
$s_6,4$
$t_0,1$
$t_1,7$
$t_2,9$
$t_3,1$
$t_4,7$
Splits of T-file
Worker Machine
Mapper 0
Mapper 1
Local File System

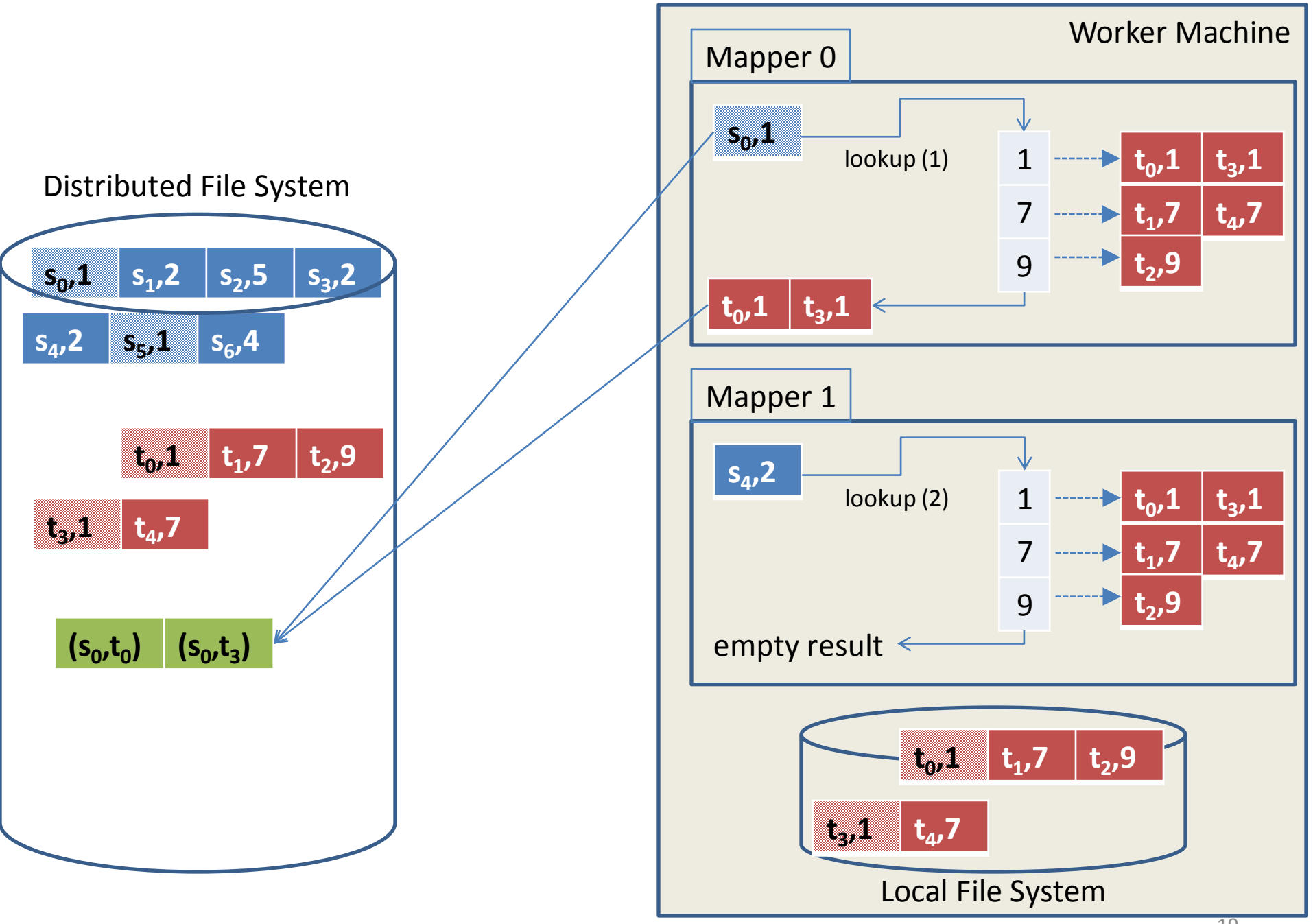

Distributed File System
$s_0,1$
$s_1,2$
$s_2,5$
$s_3,2$
$s_4,2$
$s_5,1$
$s_6,4$
$t_0,1$
$t_1,7$
$t_2,9$
$t_3,1$
$t_4,7$
Copy T-file
Worker Machine
Mapper 0
Mapper 1
$t_0,1$
$t_1,7$
$t_2,9$
$t_3,1$
$t_4,7$
Local File System

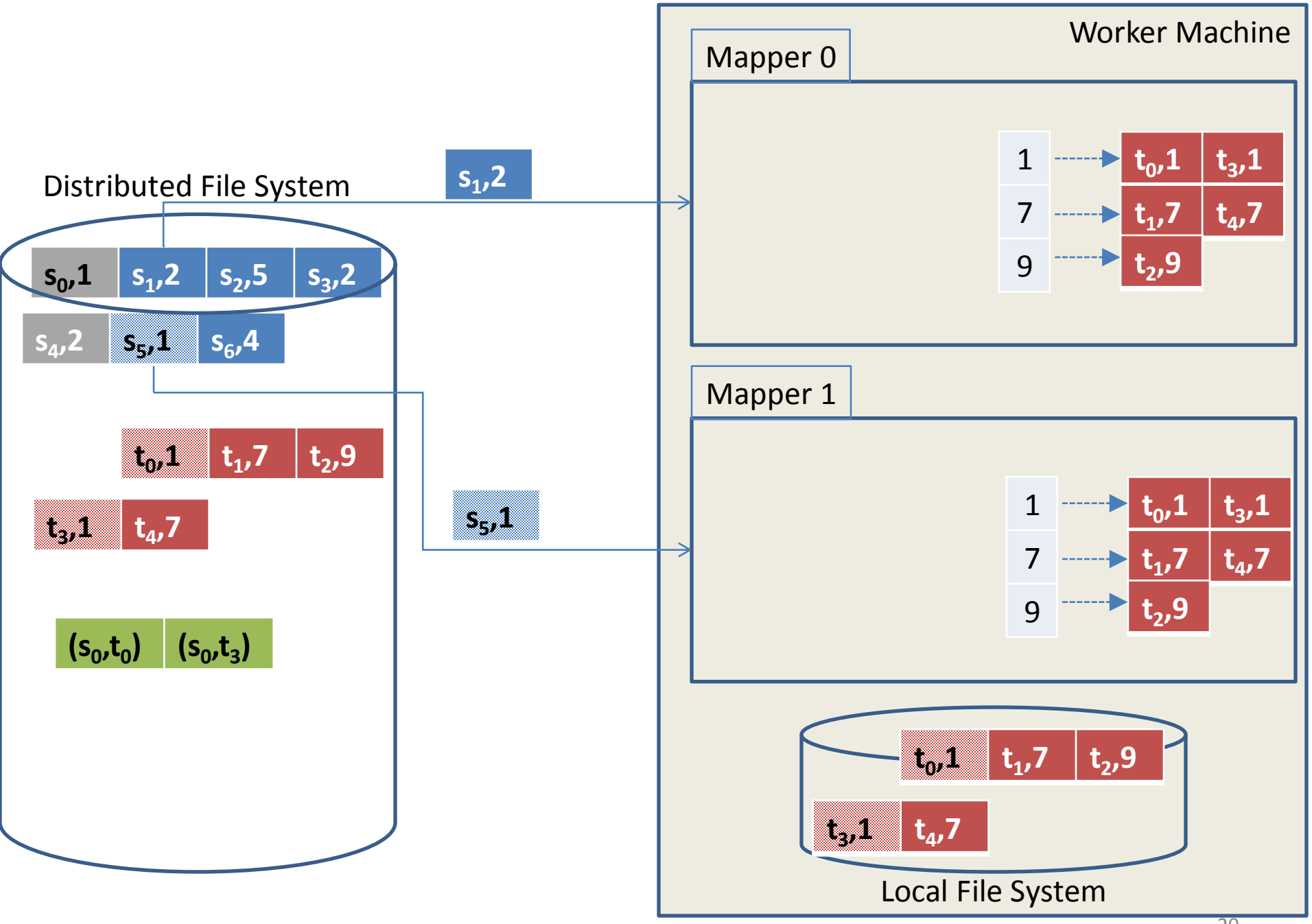

Distributed File System
$s_0,1$
$s_1,2$
$s_2,5$
$s_3,2$
$s_4,2$
$s_5,1$
$s_6,4$
$t_0,1$
$t_1,7$
$t_2,9$
$t_3,1$
$t_4,7$

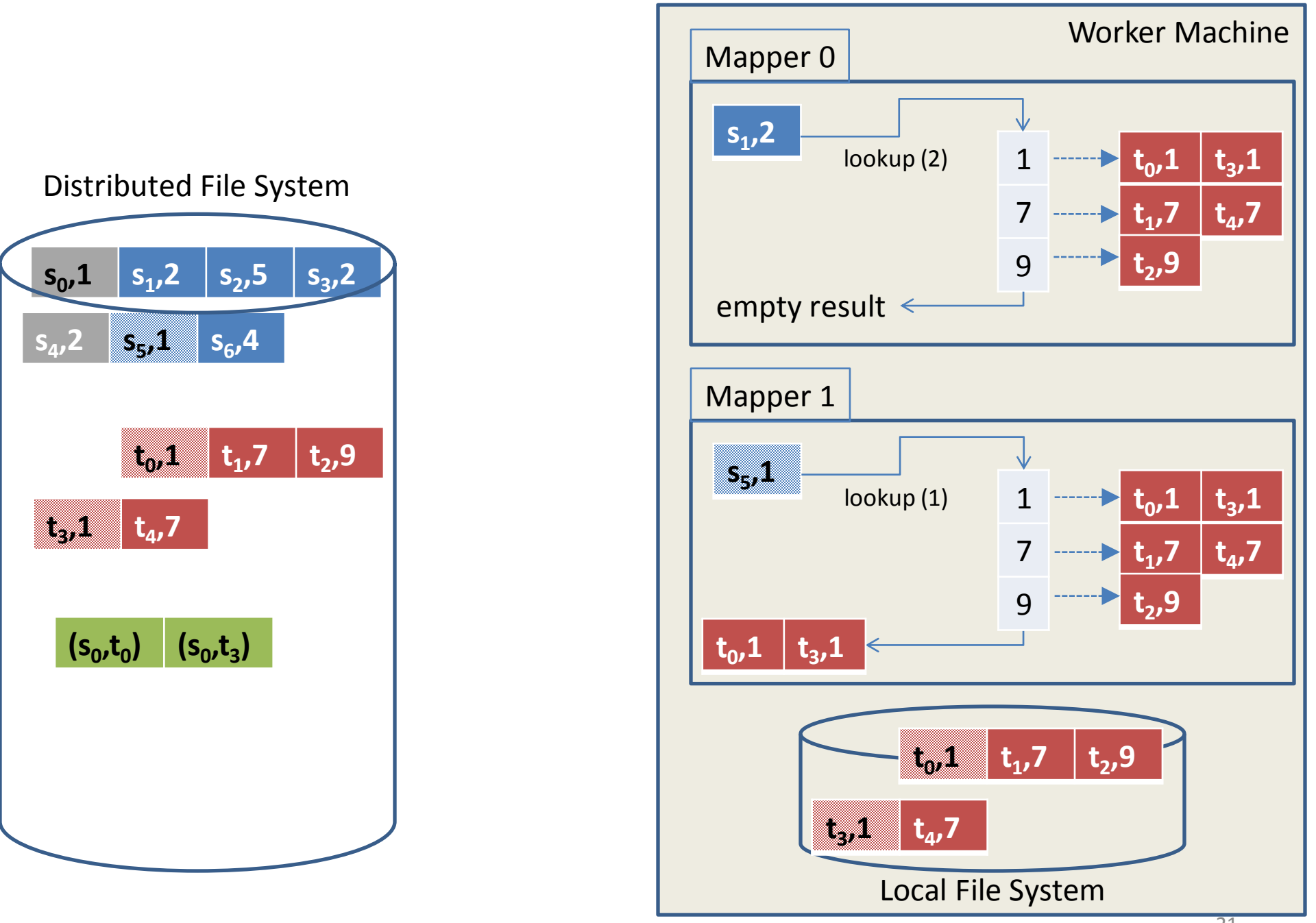

Worker Machine
Mapper 0
1
7
9
$t_0,1$
$t_3,1$
$t_1,7$
$t_4,7$
$t_2,9$
Mapper 1
1
7
9
$t_0,1$
$t_3,1$
$t_1,7$
$t_4,7$
$t_2,9$
$t_0,1$
$t_1,7$
$t_2,9$
$t_3,1$
$t_4,7$
Local File System

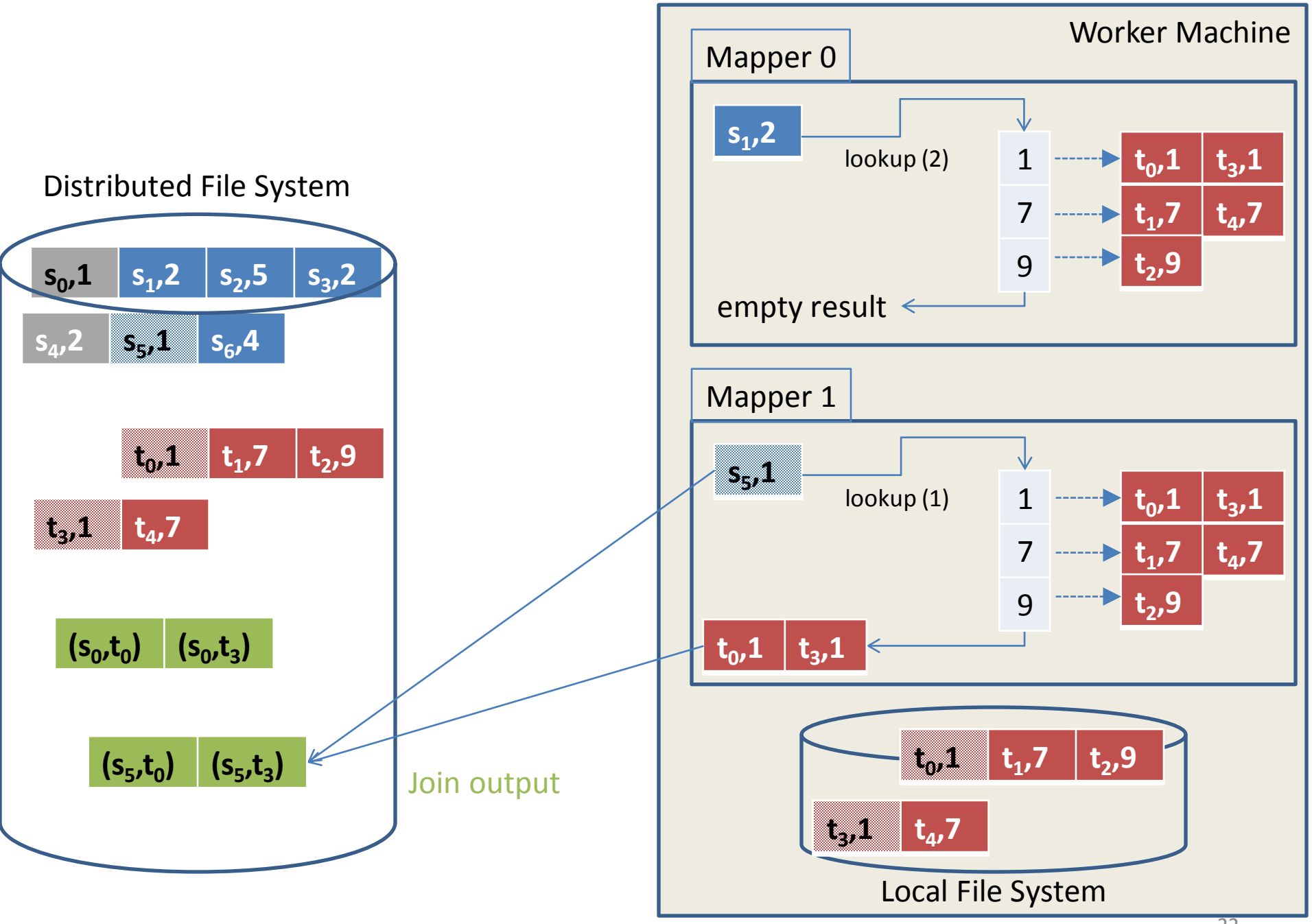

Distributed File System
$s_0,1$ $s_1,2$ $s_2,5$ $s_3,2$
$s_4,2$ $s_5,1$ $s_6,4$
$t_0,1$ $t_1,7$ $t_2,9$
$t_3,1$ $t_4,7$
$s_0,1$
$s_4,2$
Worker Machine
Mapper 0
1
7
9
$t_0,1$ $t_3,1$
$t_1,7$ $t_4,7$
$t_2,9$
Mapper 1
1
7
9
$t_0,1$ $t_3,1$
$t_1,7$ $t_4,7$
$t_2,9$
$t_0,1$ $t_1,7$ $t_2,9$
$t_3,1$ $t_4,7$
Local File System


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
$s_0,1$
lookup (1)
1
7
9
$t_0,1$ $t_3,1$
$t_1,7$ $t_4,7$
$t_2,9$
$t_0,1$ $t_3,1$
Mapper 1
$s_4,2$
lookup (2)
1
7
9
$t_0,1$ $t_3,1$
$t_1,7$ $t_4,7$
$t_2,9$
empty result
$t_0,1$ $t_1,7$ $t_2,9$
$t_3,1$ $t_4,7$
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
$s_0,1$
lookup (1)
1
7
9
$t_0,1$ $t_3,1$
$t_1,7$ $t_4,7$
$t_2,9$
$t_0,1$ $t_3,1$
Mapper 1
$s_4,2$
lookup (2)
1
7
9
$t_0,1$ $t_3,1$
$t_1,7$ $t_4,7$
$t_2,9$
empty result
$t_0,1$ $t_1,7$ $t_2,9$
$t_3,1$ $t_4,7$
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)
$s_1,2$
$s_5,1$
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


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
$t_0,1$ $t_3,1$
$t_1,7$ $t_4,7$
$t_2,9$
$t_0,1$ $t_3,1$
$t_1,7$ $t_2,9$
$t_3,1$ $t_4,7$
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
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

Diskussion

- Der Replicated Join scheint alle Probleme des Reduce-side Joins zu beheben:
  - Schlechte Lastverteilung bei schiefer Verteilung der Joinattributwerte
  - Skaliert nicht für Joinattribute mit kleiner Anzahl verschiedener Werte
  - Beschränkung auf Equi-Joins
  - Doppelter Transfer von S und T
- Dann ist er also perfekt?

Diskussion

- Replicated Join kann insgesamt mehr Daten über das Netzwerk übertragen als Reduce-side Join.
  - Replicated Join
    - Liest $|T|$, sendet $n \cdot |T|$ zu den n Mapper Maschinen
    - Liest $|S|$
  - Reduce-side Join
    - Liest $|S|+|T|$
    - Sendet $|S|+|T|$ von Mappern zu Reducern
- Datensatz T sollte in den Speicher passen.
  - Ansonsten sind teure Plattenzugriffe nötig.
  - Das Programm wird komplizierter als mit einem einfachen Hash Index im Speicher.

Diskussion

- Insgesamt bietet sich Replicated Join nur dann an wenn T viel kleiner als S ist, und idealerweise ganz in den Arbeitsspeicher passt.
- Allerdings hat Replicated Join einen großen Vorteil gegenüber Reduce-side Join, weil er jeden Theta-Join berechnen kann!
- Können wir noch bessere Algorithmen für Theta-Joins finden?

Größe von S	Größe von T	Equi-Join	Theta-Join
Groß	Groß	Reduce-side Join (Probleme bei schiefen Verteilungen, wenigen verschiedenen Werten)	???
Groß	Klein	Replicated Join (Reduce-side Join)	Replicated Join
Klein	Groß	Replicated Join (Reduce-side Join)	Replicated Join
Klein	Klein	Einfach	Einfach

4. Theta-Joins

- Dieser Teil basiert auf einer Veröffentlichung von Prof. Riedewalds Arbeitsgruppe in der führenden Datenbankenkonferenz (ACM SIGMOD).

Problem Definition

- Theta-Join generalisiert den Equi-Join:
  - Für S={s_1, s_2,…} and T={t_1, t_2,…}, finde alle Paare (s_i, t_j) die ein gegebenes Prädikat P(s_i, t_j) erfüllen.
- Ein typisches Beispiel sind Ungleichheits-Bedingungen, z.B. um nah beieinander liegende Messungen zwischen zwei Wetterdatensätzen zu finden.
- Das Ziel besteht darin, jedes gegebene Theta-Join Problem so auf Arbeitsknoten zu verteilen, dass Antwortzeit (von Job Start bis Ende) minimiert wird.

4.1 1-Bucket-Random Algorithmus

- Gegeben zwei ganze Zahlen A und B, kreiert der Algorithmus A·B Tasks, von denen jeder etwa 1/A von S und 1/B von T erhält.
  - Diese Partitionen enthalten zufällig ausgewählte Daten, was für sehr ausgeglichene Lastverteilung sorgt.
- Das Endresultat ist die Vereinigung der Einzelresultate.
- Hauptidee: weise jedes S-Tupel zufällig an eine von A Zeilen zu; und jedes T-Tupel an eine von B Spalten.

1-Bucket-Random: Map

```
map(…, tuple x) {
  if (x is from S) {
    // Select a random integer from range [0,…, A-1]
    row = random( 0, A-1 )

    // Emit the tuple for all keys in the selected "row".
    for key = (row * B) to (row * B + B – 1)
      emit( key, (x, "S") )
  }
  else {     // x is from T
    // Select a random integer from range [0,…, B-1]
    col = random( 0, B-1 )

    // Emit the tuple for all keys in the selected "column".
    // This requires skipping B region numbers forward from
    // start region key equal to col.
    for key = col to ((A-1)*B + col) step B
      emit( key, (x, "T") )
  }
}
```

0	1	2
3	4	5

Partitionierung für A=2 und B=3. Nummern zeigen Schlüssel der einzelnen Regionen.

1-Bucket-Random: Reduce

```
reduce( regionID, [(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 )

  // Any appropriate (non-parallel) join implementation
  // can be used to join S_list and T_list
  joinResult = myFavoriteJoinAlgorithm( S_list, T_list )
  for each tuple t in joinResult
    emit( t )
}
```

Eigenschaften

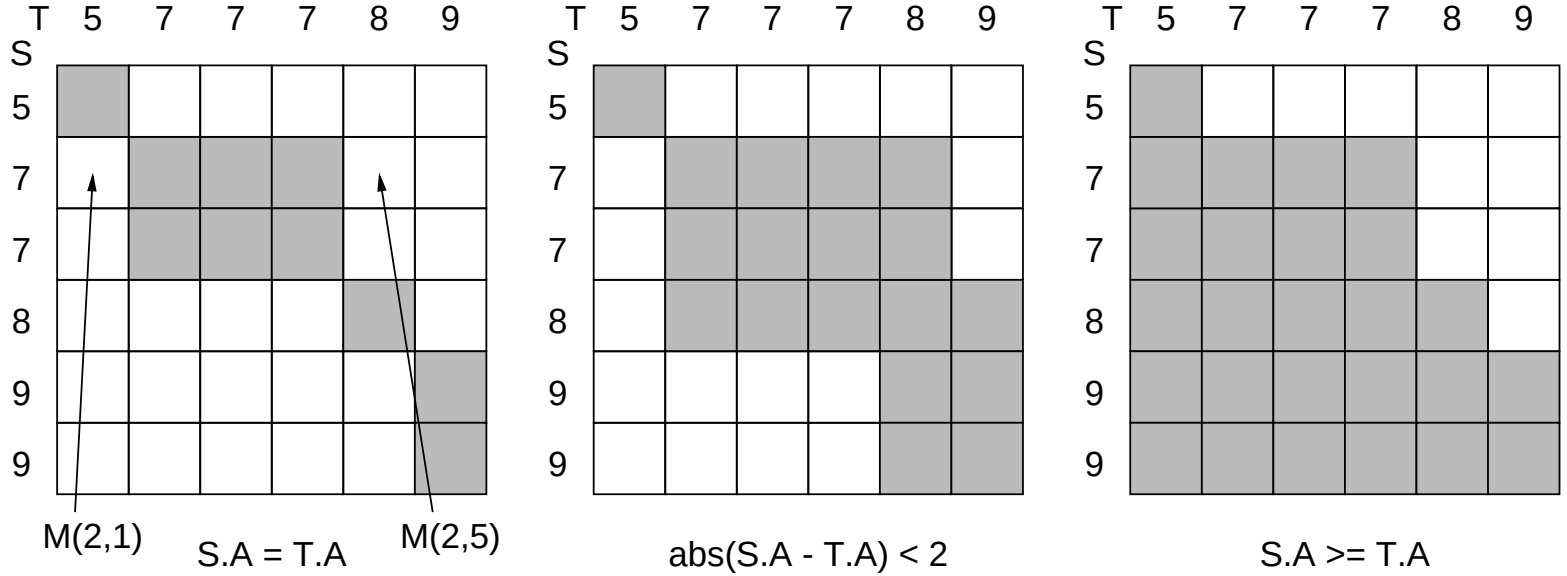
- Fast perfekte Eingabelastverteilung praktisch garantiert (Chernoff Ungleichung)
- Erwartungswert der Ein- und Ausgabelast ist perfekt verteilt
- Für eine gewünschte Anzahl Tasks können wir immer gute Werte für A und B finden
  - Höchstens $(2 + \frac{1}{A} + \frac{1}{B})/2$ mal die untere Schranke des idealen Eingabeanteils (genereller Algorithmus)
    - Faktor 1,1 für A=B=10
  - Höchstens $\frac{(A+1)(B+1)}{A \cdot B}$ mal die untere Schranke des idealen Ausgabeanteils (Erwartungswert)
    - Faktor 1,21 für A=B=10

Verbesserung von 1-Bucket-Random

- Jedes Eingabetupel von S wird B mal kopiert, jedes von T wird A mal kopiert.
- Ausgabetupel werden nicht kopiert.
- Da die Last fast perfekt verteilt wird, kann 1-Bucket-Random nur verbessert werden indem man *weniger Eingabekopien* generiert.
- Um das zu erreichen müssen die Eigenschaften des Joinprädikats ausgenutzt werden.

4.2 Theta-Join Matrix

- Theta-Join = Teilmenge des kartesischen Produkts S×T, das jedes S-Tupel mit jedem T-Tupel verbindet.
- Darum kann jeder Theta-Join in einer Matrix M mit |S| Zeilen und |T| Spalten dargestellt werden.
- Zelle M(i, j) repräsentiert Paar (s_i, t_j). Sie hat Wert “true”, falls (s_i, t_j) das Joinprädikat erfüllt; ansonsten “false”.



Matrix Überdeckung

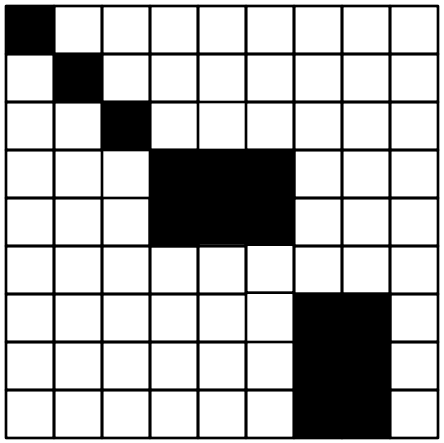
- Jede true-wertige Zelle muss von einem Task berechnet werden. Dies führt zu einem Überdeckungsproblem wo alle true-wertigen Zellen von einem Schlüssel überdeckt werden müssen.
- False-wertige Zellen brauchen nicht überdeckt werden.
  - Ihre Überdeckung führt nicht zu falschen Resultaten, da der Task die Erfüllung des Joinprädikats testen kann.
- Keine Zelle kann von mehr als einem Schlüssel überdeckt werden, um Duplikate zu vermeiden.
  - Duplikatentfernung ist teuer und benötigt Herkunftsdaten für jedes Ausgabetupel, um Duplikate von legitimen gleichen Ausgabetupeln zu unterscheiden. Und Operatoren die die Joinausgabe weiterbearbeiten, z.B. sie zählen, können nicht schon in der Reduce Phase des Joins ausgeführt werden.

Reduzierte Überdeckung

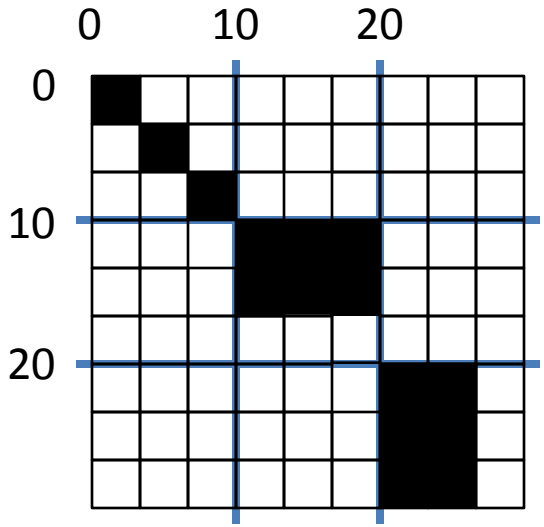
- 1-Bucket-Random ist praktisch optimal in der Klasse der Theta-Join Algorithmen die die gesamte Joinmatrix überdecken.
- Um besser zu sein, muss man den zu überdeckenden Teil der Matrix erheblich verringern.
- Da alle true-wertigen Zellen überdeckt werden müssen, kann dies nur über Identifikation von false-wertigen Zellen geschehen. Aber wie findet man diese, ohne den Join selbst zu berechnen?

Ausnutzung von Daten- und Joineigenschaften

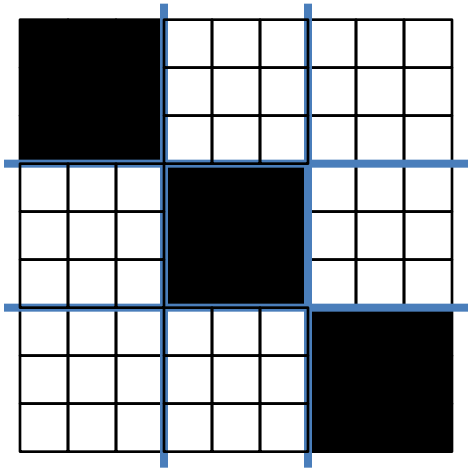
- Für eine Region der Matrix muss bewiesen werden (unter geringem Kostenaufwand), dass dort keine true-wertigen Zellen sein können.
- Dazu müssen die Eigenschaften von Daten in der Region eingeschränkt werden.
- Für Equi-Join S.A = T.A kann dies folgendermaßen erreicht werden. Angenommen A hat Werte zwischen 0 und 30, die in Intervalle [0,10), [10, 20) und [20, 30) aufgeteilt sind. Ein S-Tupel in Intervall [10,20) kann nicht mit T-Tupeln im Intervall [20,30) verbunden werden usw. Auf diese Weise können die meisten Intervall-Kombinationen verworfen werden. Für eine entsprechend organisierte Join-Matrix bedeutet dies, dass ein großer Teil nicht überdeckt werden braucht, was Eingabekopien drastisch reduziert.



Join matrix for equi-join S.A = T.A.



Intervall-partitionierte Matrix. Jedes S-Tupel gehört in genau eine der drei Zeilenpartitionen; jedes T-Tupel in genau eine der drei Spaltenpartitionen.



Zellen die überdeckt werden müssen (schwarz markiert). Die Gleichheitsbedingung des Equi-Joins eliminiert 6 von 9 Blöcken.

Da nur 1/3 der Matrix überdeckt werden muss, liegt die untere Schranke für Eingabegröße pro Task deutlich unter der für Algorithmen die die gesamte Matrix überdecken.

M-Bucket Algorithmen

- Nachdem Regionen mit zu überdeckenden Zellen identifiert wurden, müssen sie mit r Schlüsseln überdeckt werden. Aufgrund der komplexeren Struktur ist dies schwieriger als für die gesamte Matrix.
- Wir haben hierfür zwei Heuristiken vorgeschlagen, M-Bucket-I und M-Bucket-O, welche sich auf Eingabe- beziehungsweise Ausgabeverteilung konzentrieren.

Praktische Relevanz

- Für eine große Klasse von Joins ist 1-Bucket-Random fast optimal
  - Benutzer-definierte Joinprädikate
  - Komplexe Joinprädikate
  - Wenn Ausgabegröße >> Eingabegröße
- Experimente mit echten und synthetischen Daten zeigten dass 1-Bucket-Random und die M-Bucket Algorithmen in der Praxis sehr effektiv und effizient sind.

The Big Picture

- Reduce-side Join ist ein Spezialfall von M-Bucket-I.
- Replicated Join ist ein Spezialfall von 1-Bucket-Random.
- 1-Bucket-Random und die M-Bucket Algorithmen können Hauptspeicher-optimiert arbeiten indem man Parameter r entsprechend erhöht.
- Anstatt Eingabe- und Ausgabegröße individuell zu betrachten könnte man auch kombinierte Metriken in Betracht ziehen, z.B. ihre Summe.
- Gleichmäßige Verteilung der Ausgabelast is schwierig.
  - Klassisches Problem der Selektivitätsschätzung in Datenbanken.
  - 1-Bucket-Random umgeht dies dank Randomisierung.

5. Wichtige Punkte

- Datenverteilung stellt die wohl wichtigste Herausforderung für verteilte Verarbeitung von großen Datenmengen dar. Sie sollte zwei Eigenschaften aufweisen:
  - Jeder Knoten erhält einen kleinen Teil der Daten.
  - Jeder Knoten kann seine Arbeit unabhängig von den anderen ausführen, mit geringen Datenaustausch.
- Modellierung des Verteilungsproblems als Matrix-Überdeckung vereinfacht Algorithmus-Design und Analyse.
- Randomisierung spielt eine wichtige Rolle bei der Konvertierung einer Matrix-Überdeckung in einen verteilten Algorithmus. Sie kann das Beweisen von Eigenschaften vereinfachen, z.B. von unteren und oberen Schranken.

Bonusmaterial

Korrektheit von 1-Bucket-Random

- Nehme ein beliebiges Paar (s,t), $s \in S$ und $t \in T$.
- Tupel s wird allen Schlüsseln in genau einer Zeile zugewiesen; t allen Schlüsseln in genau einer Spalte.
- Darum gibt es genau einen Schlüssel—wo Zeile von s und Spalte von t überschneiden—der *beide* erhält.
- Die Reduce Ausführung für diesen Schlüssel generiert Ausgabe (s,t), falls diese das Joinprädikat erfüllt.
- Keine andere Reduce Ausführung kann (s,t) generieren.

Lastverteilung

- Mit Hilfe der Chernoff Ungleichung kann gezeigt werden, dass für realistische Datengröße die Wahrscheinlichkeit dass ein Task mehr als 5% über seinem idealen Eingabeanteil erhält, praktisch Null ist.
- Es kann auch gezeigt werden, dass die Erwartung des Anteils and der Ausgabe den idealen Wert 1/AB aufweist.

Bestimmen von A und B

- Sei *r* die gewünschte Anzahl von Teilproblemen.
- Um A und B zu bestimmen, benötigt man nur minimale Information über die Daten, nämlich das Verhältnis $|S|/|T|$.
  - Das ist weniger als die Kardinalitäten $|S|$ und $|T|$, welche einem 1-Klassen Histogramm entsprechen.
- Sei ohne Beschränkung der Allgemeinheit $|S| \leq |T|$ and sei $C = |S|/|T|$.
  - Falls $C < 1/r$, dann setze $A = 1$ und $B = r$.
  - Andernfalls, d.h. für $C \geq 1/r$, setze $A = \left\lfloor \sqrt{C \cdot r} \right\rfloor$ und $B = \left\lfloor \sqrt{C^{-1} \cdot r} \right\rfloor$.

Effektivität: Eingabeverteilung

- Wir beweisen, dass mit den obigen Werten für A und B kein Task mehr als $(2 + \frac{1}{A} + \frac{1}{B})/2$ mal die untere Schranke des idealen Eingabeanteils erhält.
  - Schlechtester Fall: Für A=B=1 ist der Faktor 2.
  - Für realistischere A=B=10 ist der Faktor nur 1,1.
- Das Result gilt nur für Join Algorithmen die nicht spezielle Eigenschaften des Joinprädikats ausnutzen. (Für auf bestimmte Jointypen spezialisierte Algorithmen kann die untere Schranke niedriger liegen, was zu einem höheren Faktor führt.)
- Die Garantie ist probabilistisch, gilt aber für realistische Eingabegröße mit praktisch 100%.

Effektivität: Ausgabeverteilung

- Wir beweisen , dass mit den obigen Werten für A und B kein Task mehr als $\frac{(A+1)(B+1)}{A \cdot B}$ mal die untere Schranke des idealen Ausgabeanteils generiert.
  - Schlechtester Fall: Für A=B=1 ist der Faktor 4.
  - Für realistischere A=B=10 ist der Faktor nur 1,21.
- Dieses Resultat gilt nur für die *erwartete* Ausgabegröße. Es gilt allerdings sogar dann, wenn die untere Schranke für Algorithmen die nur auf bestimmte Jointypen spezialisiert sind in Betracht gezogen wird.