Introduction to Data Management CSE 344

Lecture 25: MapReduce and Spark

Today

- MapReduce Review
- Parallel Join Algorithms
- Spark

Parallel Data Processing @ 2000

Map Reduce Data Model

Instance: Files!

- where a file = a bag of `(key, value)` pairs

Schema: None!

- just like other key-value data models

Query language: a MapReduce program:

- Input: a bag of `(inputkey, value)` pairs
- Output: a bag of `(outputkey, value)` pairs

Step 1: the MAP Phase

User provides the `map` function:

- Input: `(input key, value)`
- Output: bag of `(intermediate key, value)`

System applies `map` in parallel to all `(input key, value)` pairs in the input file

Step 2: the REDUCE Phase

User provides the `reduce` function:

- Input: `(intermediate key, bag of values w/ same key)`
- Output: bag of output `(values)`

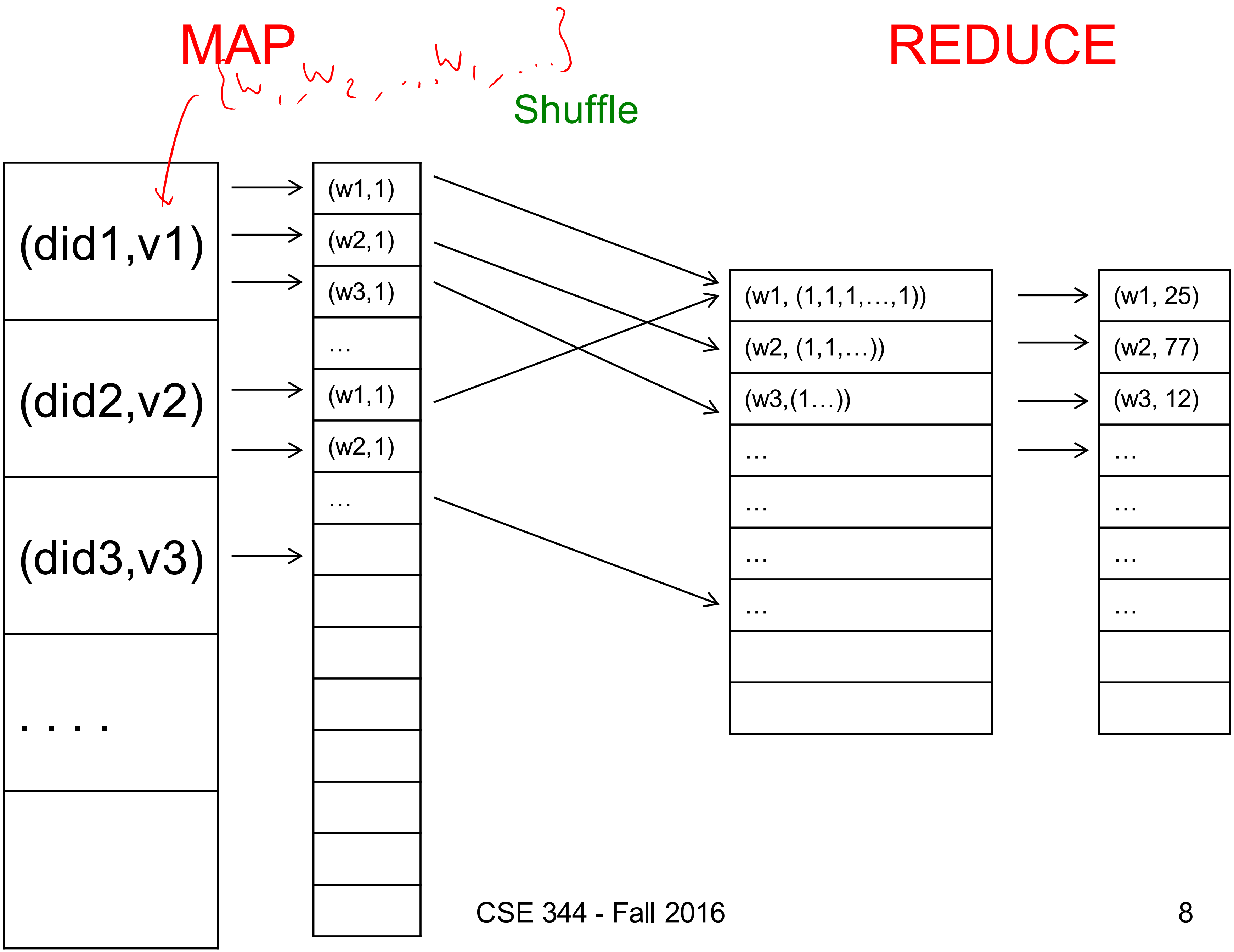
System groups all pairs with the same intermediate key, and passes the bag of values to `reduce`

Example

- Counting the number of occurrences of each word in a large collection of documents
- Each Document
  - The key = document id (did)
  - The value = set of words (word)

```
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");
```

```
reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(AsString(result));
```


MAP
{w, w2, ..., w1, ...}
Shuffle
REDUCE
(did1,v1)
(did2,v2)
(did3,v3)
. . . .
(w1,1)
(w2,1)
(w3,1)
…
(w1,1)
(w2,1)
…
(w1, (1,1,1,…,1))
(w2, (1,1,…))
(w3,(1…))
…
…
…
…
(w1, 25)
(w2, 77)
(w3, 12)
…
…
…
…

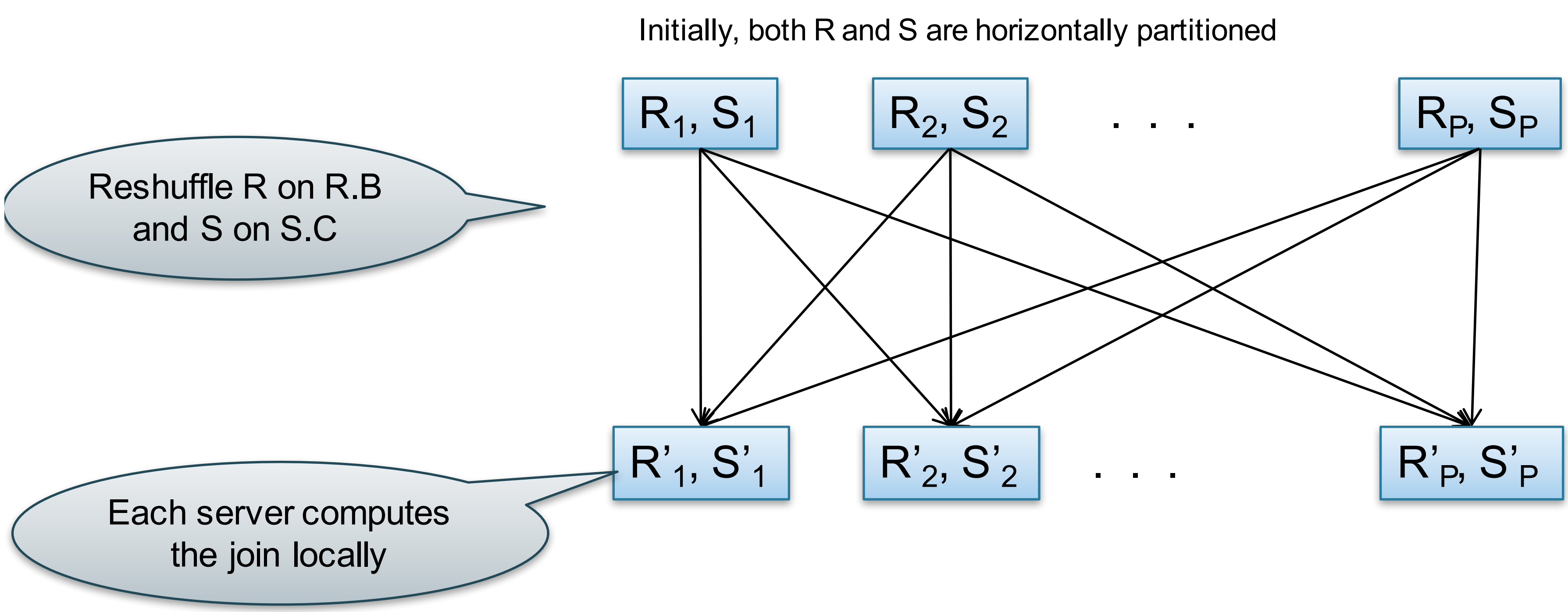
Fault Tolerance

- If one server fails once every year…
  ... then a job with 10,000 servers will fail in less than one hour

- MapReduce handles fault tolerance by writing intermediate files to disk:
  - Mappers write file to local disk
  - Reducers read the files (=reshuffling); if the server fails, the reduce task is restarted on another server

$R(A,B) \bowtie_{B=C} S(C,D)$

Partitioned Hash-Join



$R(A,B) \bowtie_{B=C} S(C,D)$

Partitioned Hash-Join

```
map(String relationName, String s):
  Tuple t = parse(s);
  switch (relationName):
    case 'R': EmitIntermediate(t.B, KV('R', t));
    case 'S': EmitIntermediate(t.C, KV('S', t));
```

key

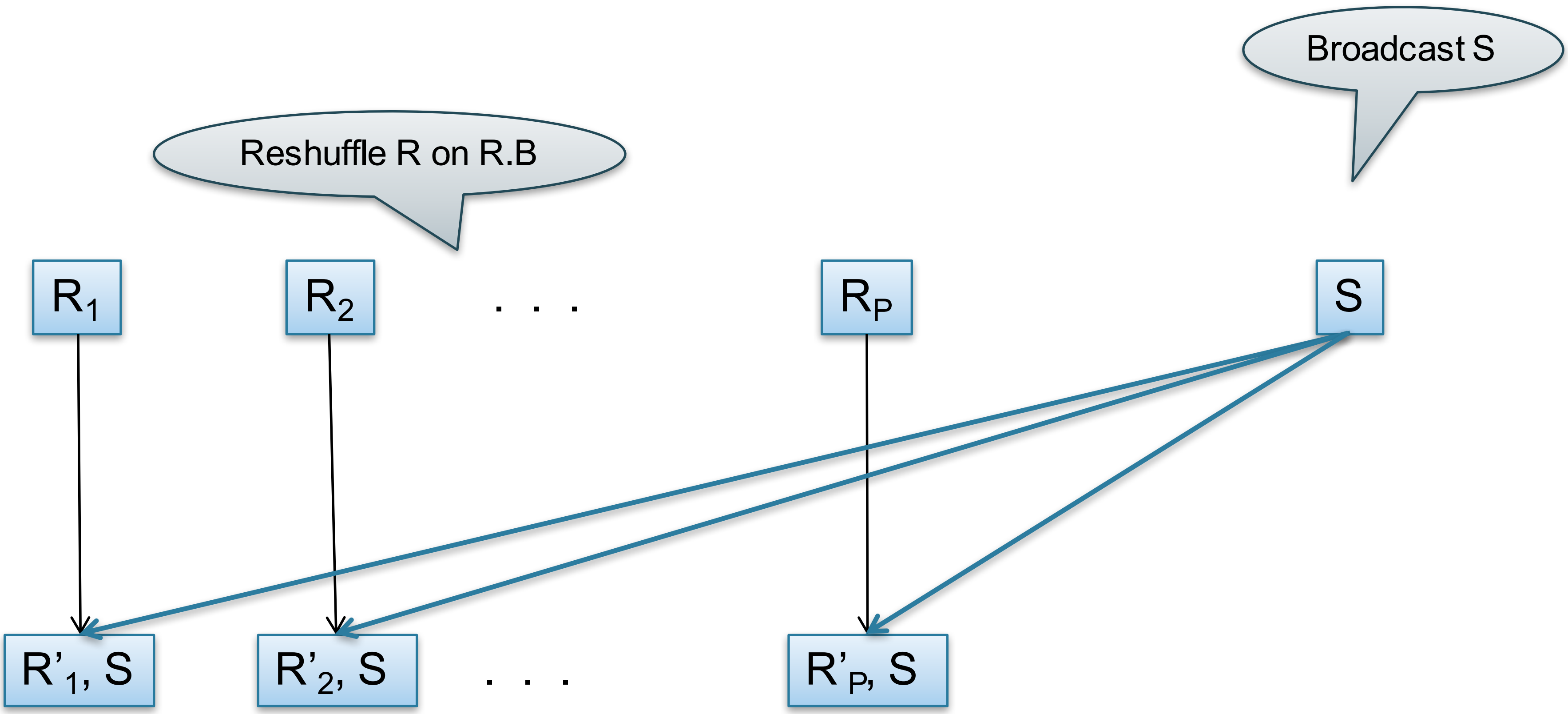
Or call hash(t.B)

```
reduce(String k, Iterator values):
  R = []; S = [];
  for each v in values:
    switch (v.relationName):
      'R': R.insert(v);
      'S': S.insert(v);
  for r in R, for s in S
    Emit(Tuple(r,s));
```

All tuples here must join

$R(A,B) \bowtie_{B=C} S(C,D)$

Broadcast Join



$R(A,B) \bowtie_{B=C} S(C,D)$

Broadcast Join

map should read several records of R: value = some group of records

```
map(String key, String s):
   Tuple [] rs = parse(s)
   S = readFromNetwork();
   ht = new Hashtable()
   for each w in S:
     ht.insert(w.C, w)

   for each r in rs:
     for each s in ht.find(r.B):
       Emit(Tuple(r,s));
```

Read entire table S, build a Hash Table

```
reduce(…):
   /* empty: map-side only */
```

Let's consider a 3-way join

- How do we compute this query?
  Q(x,y,z) :- R(x,y),S(y,z),T(z,x)

- This computes all "triangles"
- e.g., let Follows(x,y) be all pairs of Twitter users s.t. x follows y.
- If R=S=T=Follows, then Q computes all triples of people that follow each other.

Q(x,y,z) :- R(x,y),S(y,z),T(z,x)

Shuffle join

Partition

P1

R1

x	y
1	2
3	2

S1

y	z
4	7
4	9

T1

z	x
1	1
3	3

P2

R2

x	y
5	4
7	6

S2

y	z
2	3
2	9

T2

z	x
1	1
3	1

P3

R3

x	y
8	6
9	6

S3

y	z
6	7
6	9

T3

z	x
1	1
3	1

$Q(x,y,z)$:- $R(x,y), S(y,z), T(z,x)$

Shuffle join

Partition

P1

R1

x	y
1	2
3	2

S1

y	z
4	7
4	9

T1

z	x
1	1
3	3

P2

R2

x	y
5	4
7	6

S2

y	z
2	3
2	9

T2

z	x
1	1
3	1

P3

R3

x	y
8	6
9	6

S3

y	z
6	7
6	9

T3

z	x
1	1
3	1

Shuffle

Local Join

P1

R1’

x	y
1	2
3	2

S1’

y	z
2	3
2	9

T1

z	x
1	1
3	3

P2

R2’

x	y
5	4

S2’

y	z
4	7
4	9

T2

z	x
1	1
3	1

P3

R3’

x	y
7	6
8	6
9	6

S3’

y	z
6	7
6	9

T3

z	x
1	1
3	1

Local Join

R1′ ⋈ S1′

x	y	z
1	2	3
3	2	9
…	…	…

T1

z	x
1	1
3	3

P1

R2′ ⋈ S2′

x	y	z
5	4	7
5	4	9
…	…	…

T2

z	x
1	1
3	1

P2

R3′ ⋈ S3′

x	y	z
7	6	7
8	6	7
…	…	…

T3

z	x
1	1
3	1

P3

Local Join

R1' ⋈ S1'

x	y	z
1	2	3
3	2	9
…	…	…

T1

z	x
1	1
3	3

P1

R2' ⋈ S2'

x	y	z
5	4	7
5	4	9
…	…	…

T2

z	x
1	1
3	1

P2

R3' ⋈ S3'

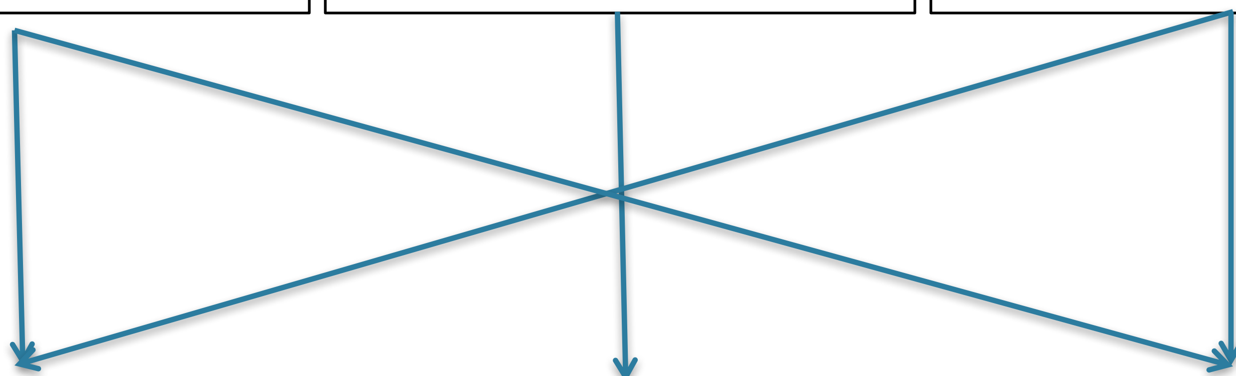
x	y	z
7	6	7
8	6	7
…	…	…

T3

z	x
1	1
3	1

P3

Shuffle



Local Join

R1' ⋈ S1'

x	y	z
1	2	3
1	2	9
…	…	…

T1'

z	x
1	1
1	1
3	1

P1

R1' ⋈ S1'

x	y	z
5	4	7
3	2	9
5	4	9

T2'

z	x
3	3

P2

R1' ⋈ S1'

x	y	z
7	6	7
8	6	7
…	…	…

T3'

z	x

P3

$Q(x,y,z) \text{ :- } R(x,y),S(y,z),T(z,x)$

Broadcast join

Partition

P1

R1

x	y
1	2
3	2

S1

y	z
4	7
4	9

T1

z	x
1	1
3	3

P2

R2

x	y
5	4
7	6

S2

y	z
2	3
2	9

T2

z	x
9	5
3	1

P3

R3

x	y
8	6
9	6

S3

y	z
6	7
6	9

T3

z	x
7	1
3	1

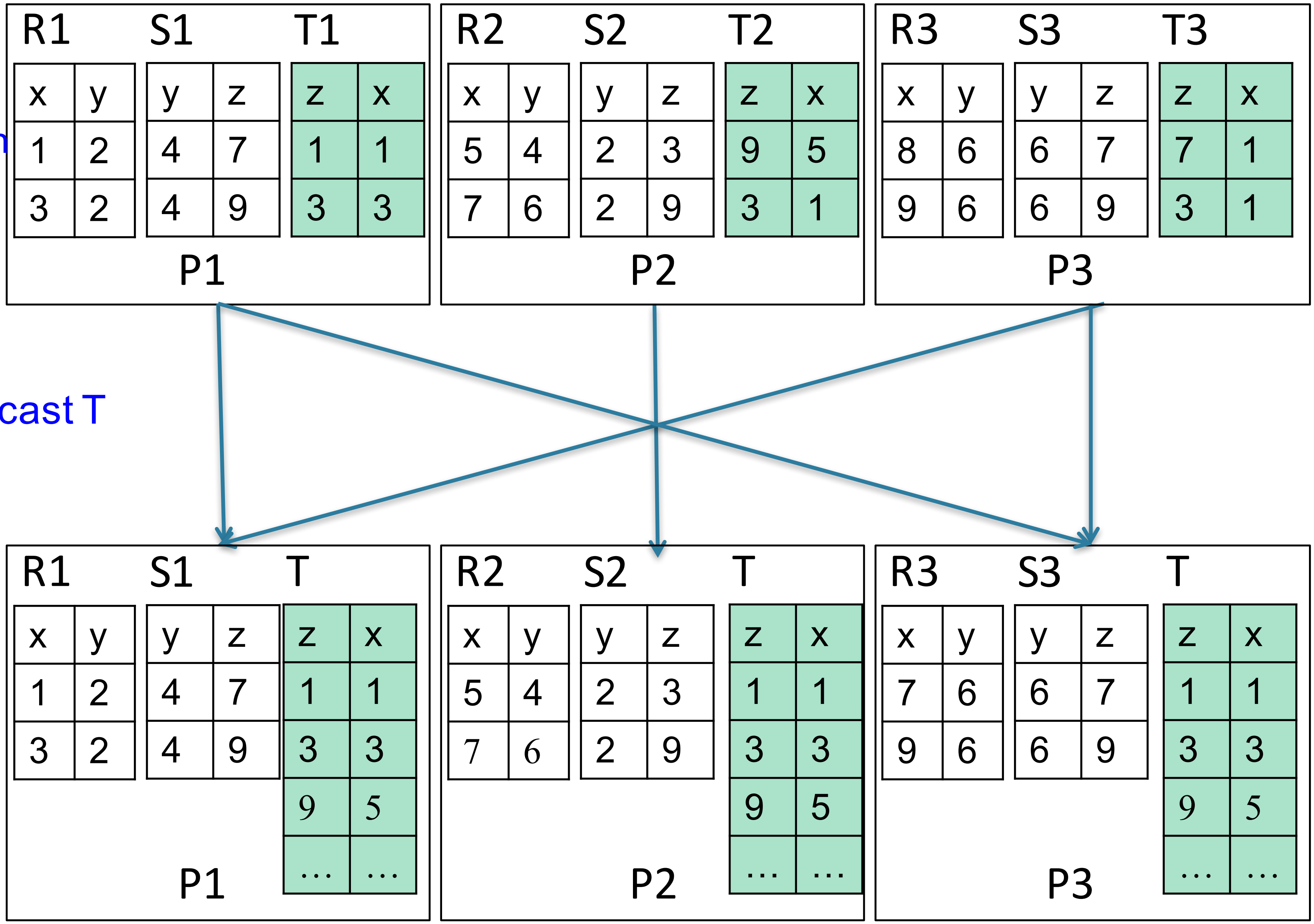
$Q(x,y,z)$:- $R(x,y),S(y,z),T(z,x)$

Broadcast join

Partition

Broadcast T

Local Join



Broadcast S …

Q(x,y,z) :- R(x,y),S(y,z),T(z,x)

Broadcast + shuffle join

Partition

P1

R1

x	y
1	2
3	2

S1

y	z
4	7
4	9

T1

z	x
1	1
3	3

P2

R2

x	y
5	4
7	6

S2

y	z
2	3
2	9

T2

z	x
9	5
3	1

P3

R3

x	y
8	6
9	6

S3

y	z
6	7
6	9

T3

z	x
7	1
3	1

Q(x,y,z) :- R(x,y),S(y,z),T(z,x)

Broadcast + shuffle join

Partition

P1

R1

x	y
1	2
3	2

S1

y	z
4	7
4	9

T1

z	x
1	1
3	3

P2

R2

x	y
5	4
7	6

S2

y	z
2	3
2	9

T2

z	x
9	5
3	1

P3

R3

x	y
8	6
9	6

S3

y	z
6	7
6	9

T3

z	x
7	1
3	1

Shuffle R, S + Broadcast T

Local Join

P1

R1'

x	y
1	2
3	2

S1'

y	z
2	7
2	9

T

z	x
1	1
3	3
9	5
…	…

P2

R2'

x	y
5	4

S2'

y	z
4	3
4	9

T

z	x
1	1
3	3
9	5
…	…

P3

R3'

x	y
7	6
8	6
9	6

S3'

y	z
6	7
6	9

T

z	x
1	1
3	3
9	5
…	…

Issues

- Shuffle join
  - Requires two shuffles
- Broadcast join
  - Requires two broadcasts
- Shuffle + broadcast join
  - Redundant broadcast
- Can we do better?

Hypercube join

- Assign each server a specific value of (x,y,z)
  - e.g., P1: (1,2,3) means P1 gets all tuples with x=1, y=2, and z=3
  - Join can now proceed in one step!

- What if we have more values of x than # of servers?
  - Use modulo!

Q(x,y,z) :- R(x,y),S(y,z),T(z,x)

Hypercube join

Partition

P1

R1

x	y
1	2
3	2

S1

y	z
4	7
4	9

T1

z	x
1	1
3	3

P2

R2

x	y
5	4
7	6

S2

y	z
2	3
2	9

T2

z	x
9	5
3	1

P3

R3

x	y
8	6
9	6

S3

y	z
6	7
6	9

T3

z	x
7	1
3	1

Shuffle

Local Join

R1'

x	y
1	2

S1'

y	z
2	7

T1

z	x
7	1

P1: (1, 2, 7)

R2'

x	y
1	2

S2'

y	z
2	3

T2

z	x
3	1

P2: (1, 2, 3)

R3'

x	y
3	2

S3'

y	z
2	3

T3

z	x
3	3

P3: (3, 2, 3)

Q(x,y,z) :- R(x,y),S(y,z),T(z,x)

Hypercube join

Partition

P1

R1

x	y
1	2
3	2

S1

y	z
4	7
4	9

T1

z	x
1	1
3	3

P2

R2

x	y
5	4
7	6

S2

y	z
2	3
2	9

T2

z	x
9	5
3	1

P3

R3

x	y
8	6
9	6

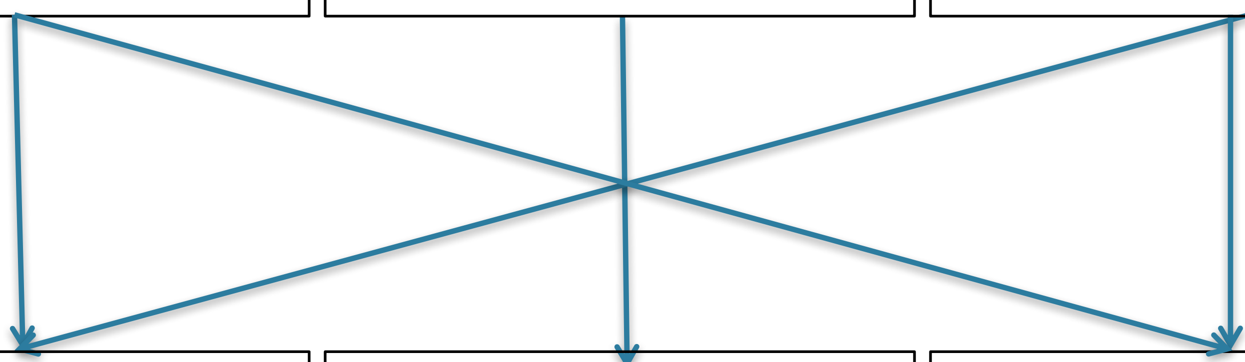
S3

y	z
6	7
6	9

T3

z	x
7	1
3	1

Shuffle



Local Join

R1’

x	y
1	2
5	4

S1’

y	z
2	7
4	7

T1

z	x
7	1

P1: (1, 2, 7) (1, 4, 7)
(5, 2, 7) (5, 4, 7)

R2’

x	y
1	2

S2’

y	z
2	3

T2

z	x
3	1

P2: (1, 2, 3)

R3’

x	y
3	2
8	6

S3’

y	z
2	3

T3

z	x
3	3

P3: (3, 2, 3)
(8, 6, 3)

Parallel Data Processing @ 2010

Issues with MapReduce

- Difficult to write more complex queries

- Need multiple MapReduce jobs: dramatically slows down because it writes all results to disk

Implementing Relational Operators in MapReduce

Given relations R(A,B) and S(B, C) compute:

- Selection: $\sigma_{A=123}(R)$
- Group-by: $\gamma_{A,sum(B)}(R)$
- Join: $R \bowtie S$

Selection $\sigma_{A=123}(R)$

```
map(String key, Tuple t):
    if  t.A = 123:
          EmitIntermediate(key, t);
```

```
reduce(String k, Iterator values):
    for each v in values:
          Emit(v);
```

Selection $\sigma_{A=123}(R)$

```
map(String key, Tuple t):
    if  t.A = 123:
          EmitIntermediate(key, t);
```

```
reduce(String k, Iterator values):
    for each v in values:
          Emit(v);
```

No need for reduce.
But need system hacking
to remove reduce from MapReduce

Group By $\gamma_{A,sum(B)}(R)$

```
map(String key, Tuple t):
    EmitIntermediate(t.A, t.B);
```

Can't use hashtable to map A's to B's

```
reduce(String k, Iterator values):
    s = 0
    for each v in values:
        s = s + v
    Emit(k, v);
```

Spark

- Open source system from Berkeley
- Distributed processing over HDFS
- Differences from MapReduce:
  - Multiple steps, including iterations
  - Stores intermediate results in main memory
  - Closer to relational algebra (familiar to you)
- Details: http://spark.apache.org/examples.html

Spark Interface

- Spark supports a Scala interface
- Scala = extension of Java with functions/closures
- We will illustrate Scala/Spark in the lectures

- Spark also supports a SQL interface, and compiles SQL to its Scala interface
- For HW8: you only need the SQL interface!

RDD

- RDD = Resilient Distributed Datasets
  - A distributed relation, together with its *lineage*
  - Lineage = expression that says how that relation was computed = a relational algebra plan
- Spark stores intermediate results as RDD
- If a server crashes, its RDD in main memory is lost. However, the driver (=master node) knows the lineage, and will simply recompute the lost partition of the RDD