Introduction to Data Management CSE 344

Lecture 24: MapReduce

Announcements

- HW7 due tonight
- Final review session this Saturday 3/11
  - EEB 105, 1-2pm

Horizontal Data Partitioning

- Block Partition:
  - Partition tuples arbitrarily s.t. $size(R_1) \approx \ldots \approx size(R_P)$
- Hash partitioned on attribute A:
  - Tuple t goes to chunk i, where i = h(t.A) mod P + 1
  - Recall: calling hash fn's is free in this class
- Range partitioned on attribute A:
  - Partition the range of A into $-\infty = v_0 < v_1 < \ldots < v_P = \infty$
  - Tuple t goes to chunk i, if $v_{i-1} < t.A < v_i$

Uniform Data v.s. Skewed Data

- Let R(K,A,B,C); which of the following partition methods may result in skewed partitions?

- Block partition

Uniform

- Hash-partition
  - On the key K
  - On the attribute A

Uniform

Assuming good hash function

May be skewed

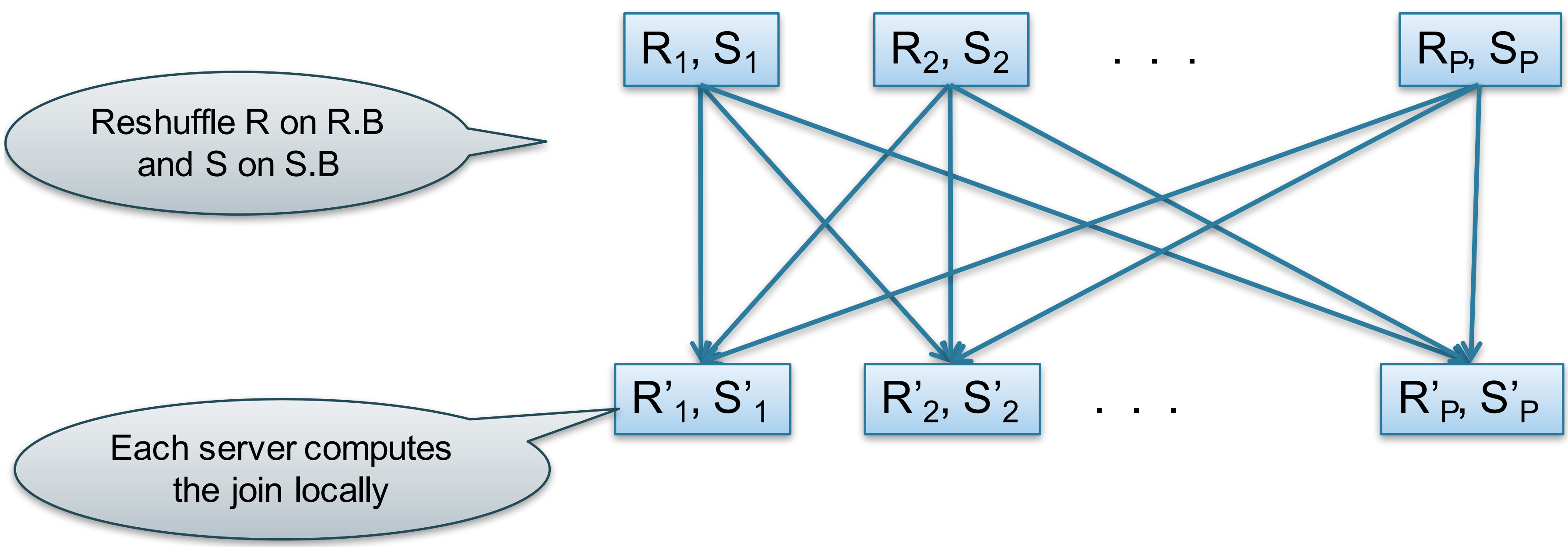
E.g. when all records have the same value of the attribute A, then all records end up in the same
tition

Keep this in mind in the next few slides

Parallel Data Processing @ 1990

Parallel Execution of RA Operators: Partitioned Hash-Join

- **Data**: R(<u>K1</u>, A, B), S(<u>K2</u>, B, C)
- **Query**: R(<u>K1</u>, A, B) ⋈ S(<u>K2</u>, B, C)
  - Initially, both R and S are partitioned on K1 and K2



Data: R(K1,A, B), S(K2, B, C)
Query: R(K1,A,B) ⋈ S(K2,B,C)

Parallel Join Illustration

Partition

M1

R1

K1	B
1	20
2	50

S1

K2	B
101	50
102	50

M2

R2

K1	B
3	20
4	20

S2

K2	B
201	20
202	50

Shuffle on B

Local Join

M1

R1'

K1	B
1	20
3	20
4	20

⋈

S1'

K2	B
201	20

M2

R2'

K1	B
2	50

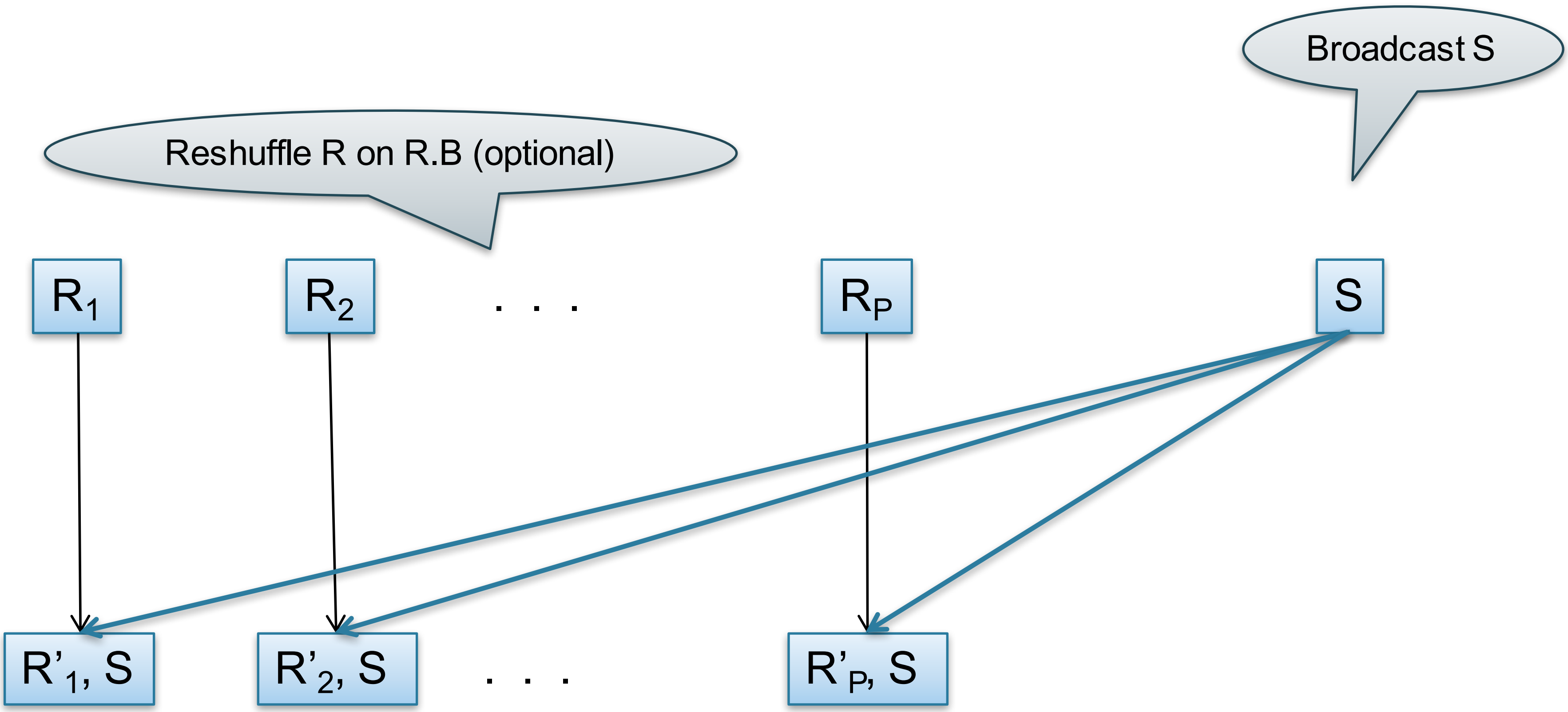
⋈

S2'

K2	B
101	50
102	50
202	50

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

Broadcast Join

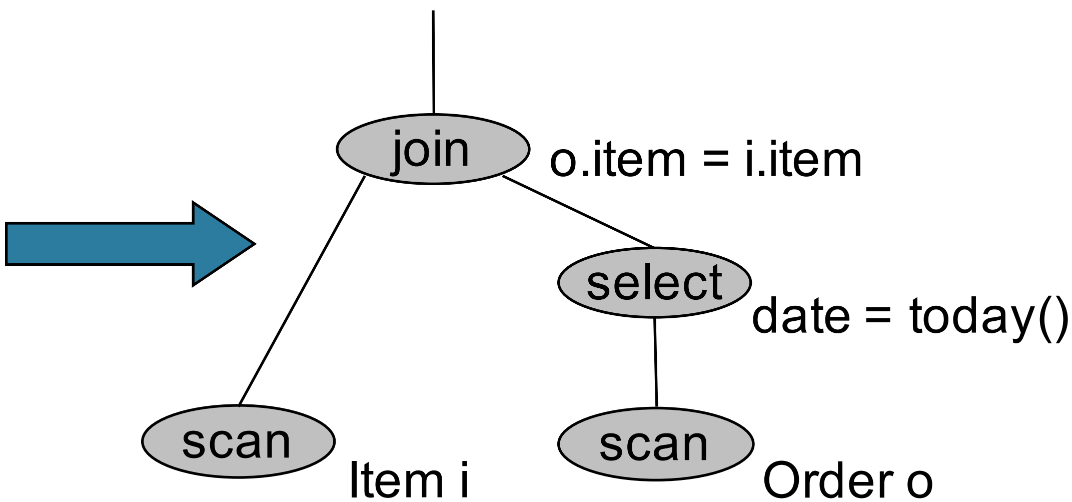


Why would you want to do this?

Putting it Together: Example Parallel Query Plan

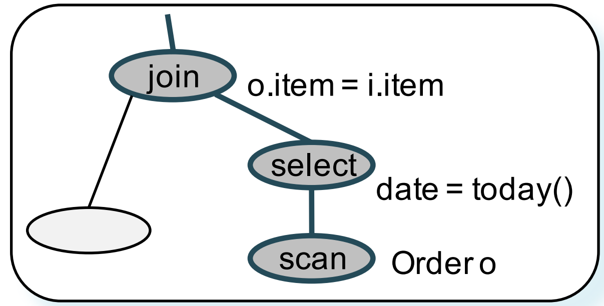
Find all orders from today, along with the items ordered

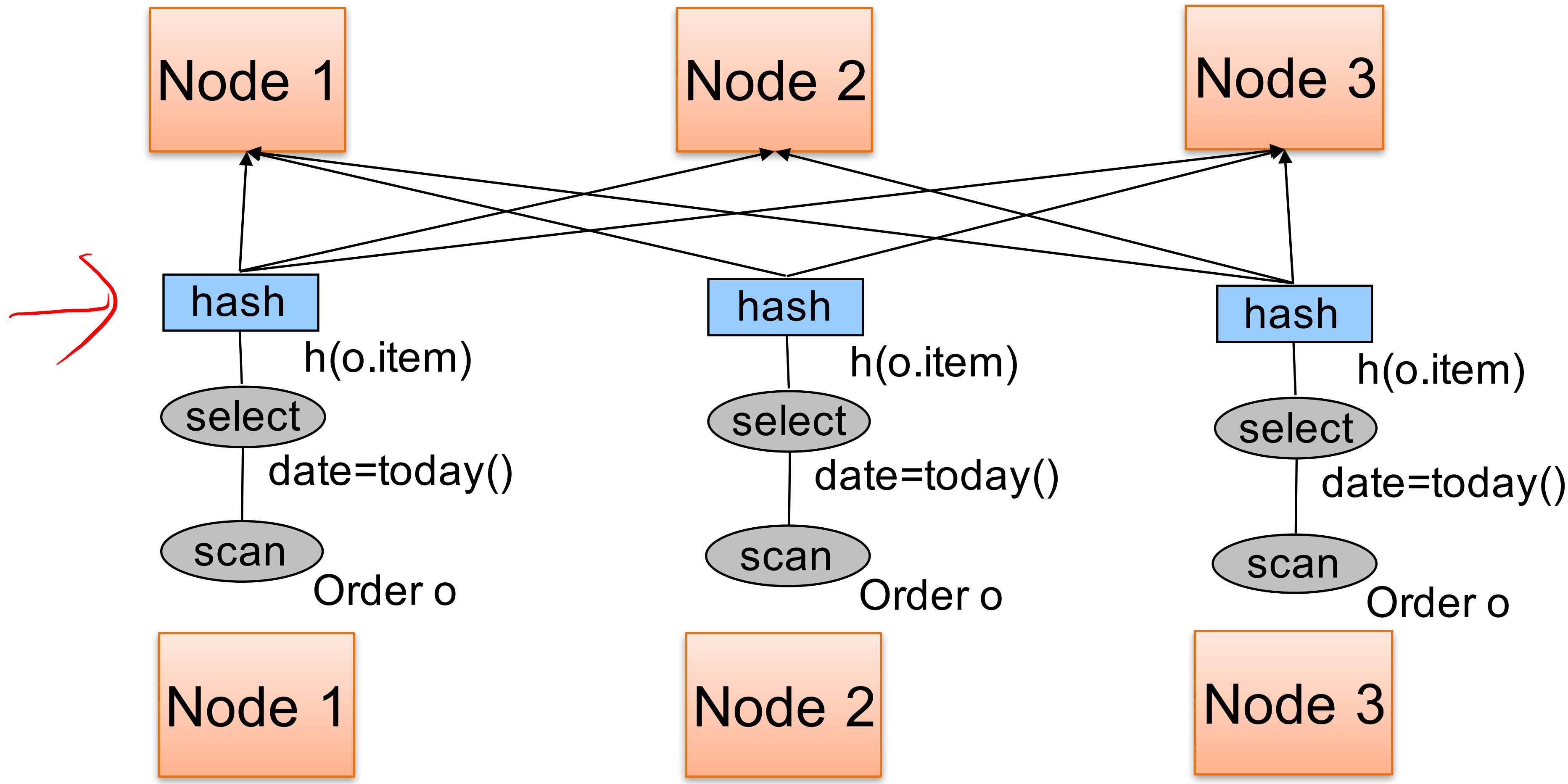
```
SELECT *
  FROM Order o, Line i
 WHERE o.item = i.item
   AND o.date = today()
```



Order(oid, item, date), Line(item, …)

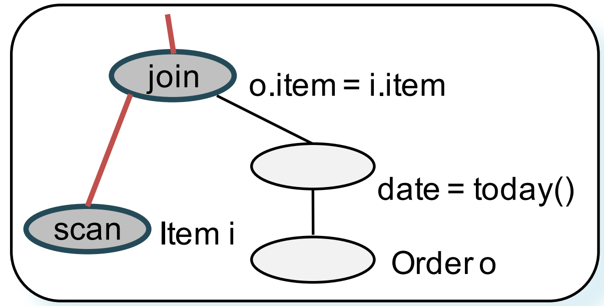
Example Parallel Query Plan

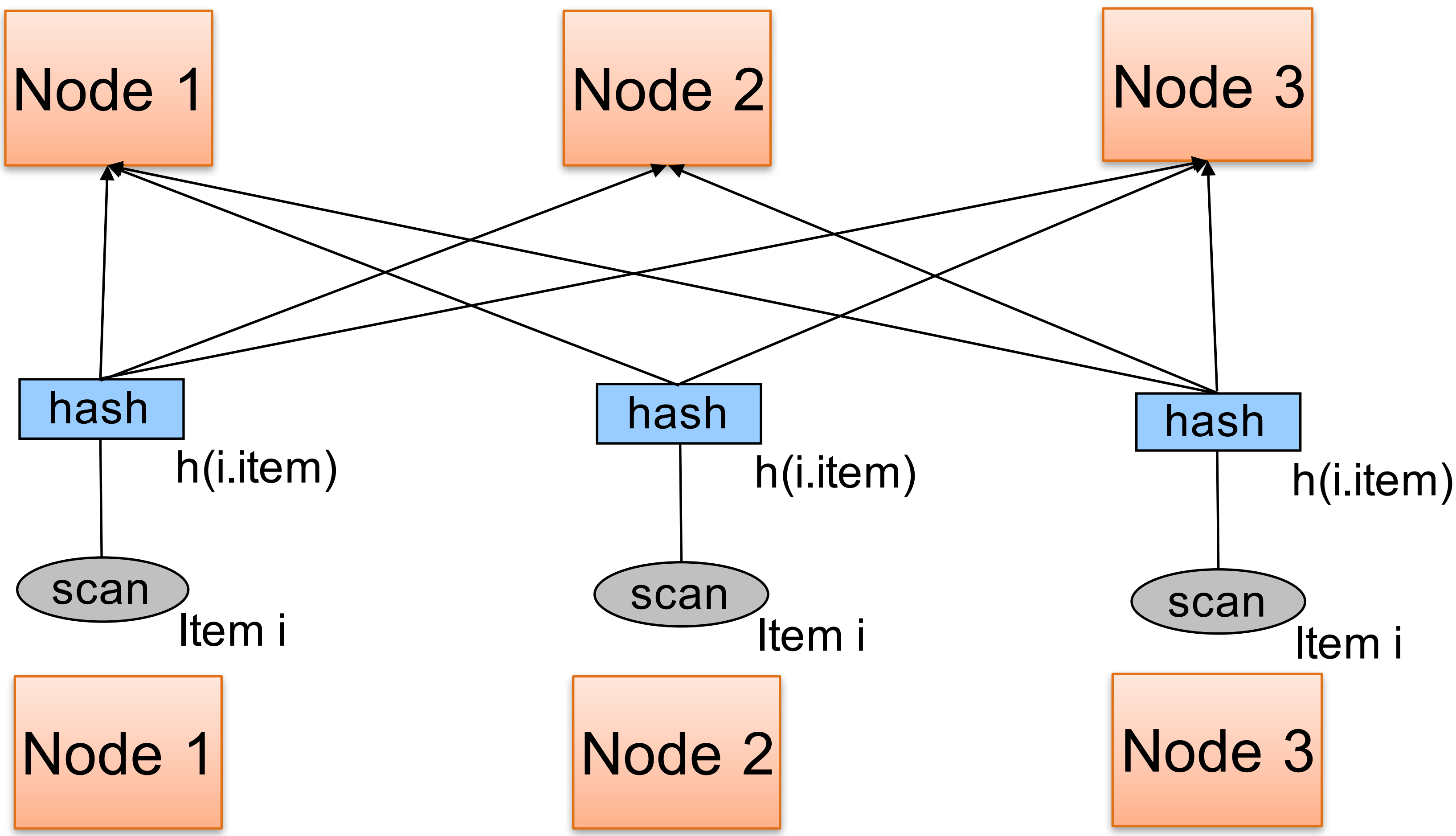




Order(oid, item, date), Line(item, …)

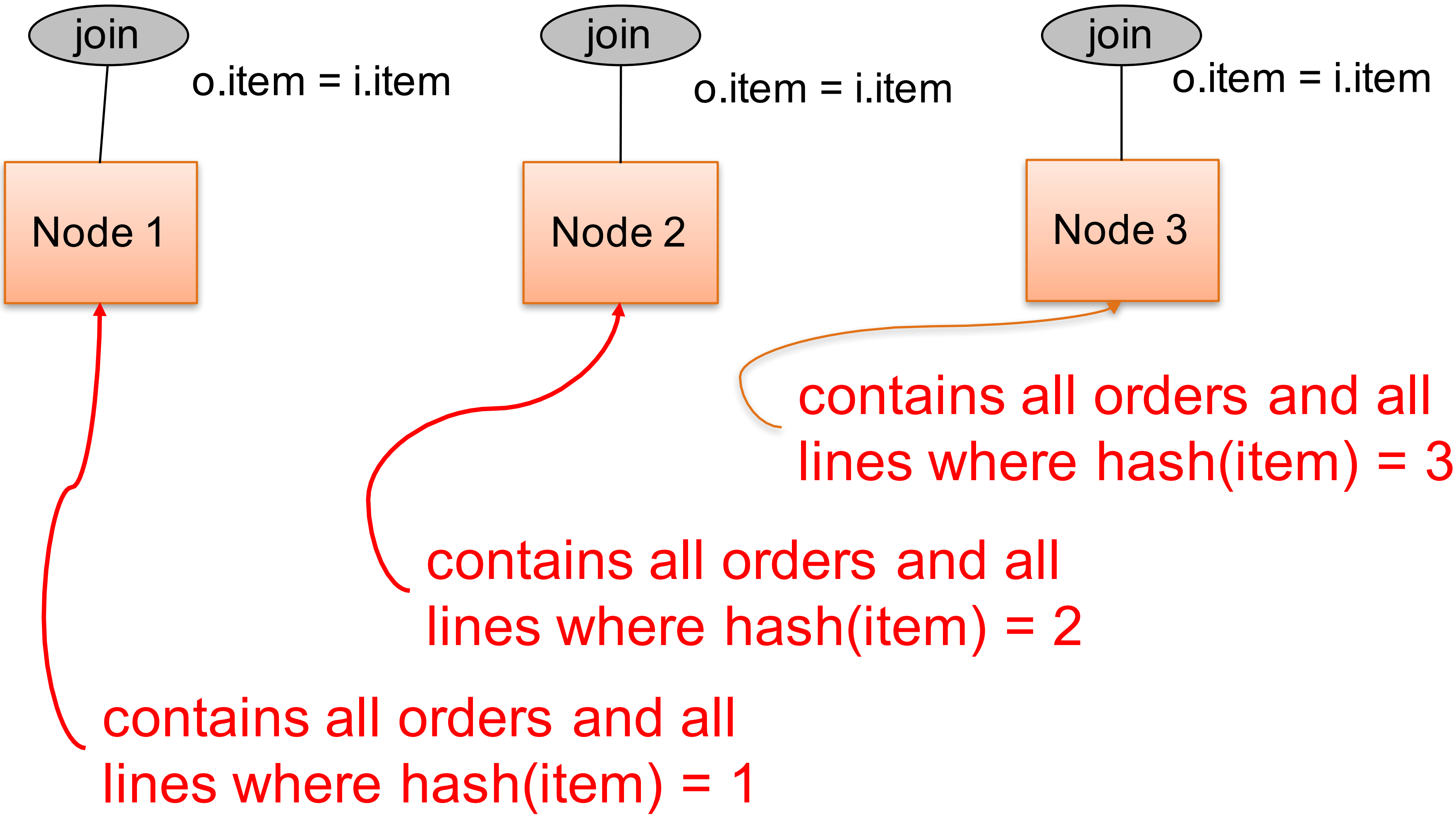
Example Parallel Query Plan





Order(<u>oid</u>, item, date), Line(item, …)

Example Parallel Query Plan



Parallel Data Processing @ 2000

What's wrong with the relational data model?

- Recall lecture on NoSQL
- For parallel data processing:
  - Want to control both data distribution and query processing
  - Want simpler programming model
    - "I don't want to learn SQL!" (non 344 student)
  - Fault tolerance is important

Map Reduce Data Model

Started by Google in 2004

Instance: Files containing (key, value) pairs

Schema: None!
- just like other key-value data models

Query language: a MapReduce program:
- Input: a bag of (key, value) pairs
- Output: a bag of (key, value) pairs
- Implementation in Java (Hadoop), Python, Go, …

Distributed File System (DFS)

- For very large files: TBs, PBs
- Each file is partitioned into *chunks*, typically 64MB
- Each chunk is replicated several times (≥3), on different racks, for fault tolerance
- Implementations:
  - Google's DFS: GFS, proprietary
  - Hadoop's DFS: HDFS, open source

Lifecycle of a MR Program

1. Read a lot of data and parse into (key, value) pairs
2. Map: extract something you care about from each (key, value) pair
3. Shuffle output from mappers
   – done internally by implementation
4. Reduce: aggregate, summarize, filter, transform
5. Write the results to files

Paradigm stays the same,
change map and reduce
functions for different problems

Step 2: the MAP Phase

User provides the MAP-function:

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

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

Step 3: the shuffle phase

- System groups all pairs generated by MAPpers with the same intermediate key
- Passes the bag of values to the REDUCE function in next stage
- Example: given map output:
  ("a", 1), ("b", 1), ("a", 2), ("c", 1), ("c", 5)

  Shuffle produces the output:
  ("a", [1,2]), ("b", [1]), ("c", [1,5])

- key This is just another (key, value) pair!
  value

Step 4: the REDUCE Phase

User provides the REDUCE function:

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

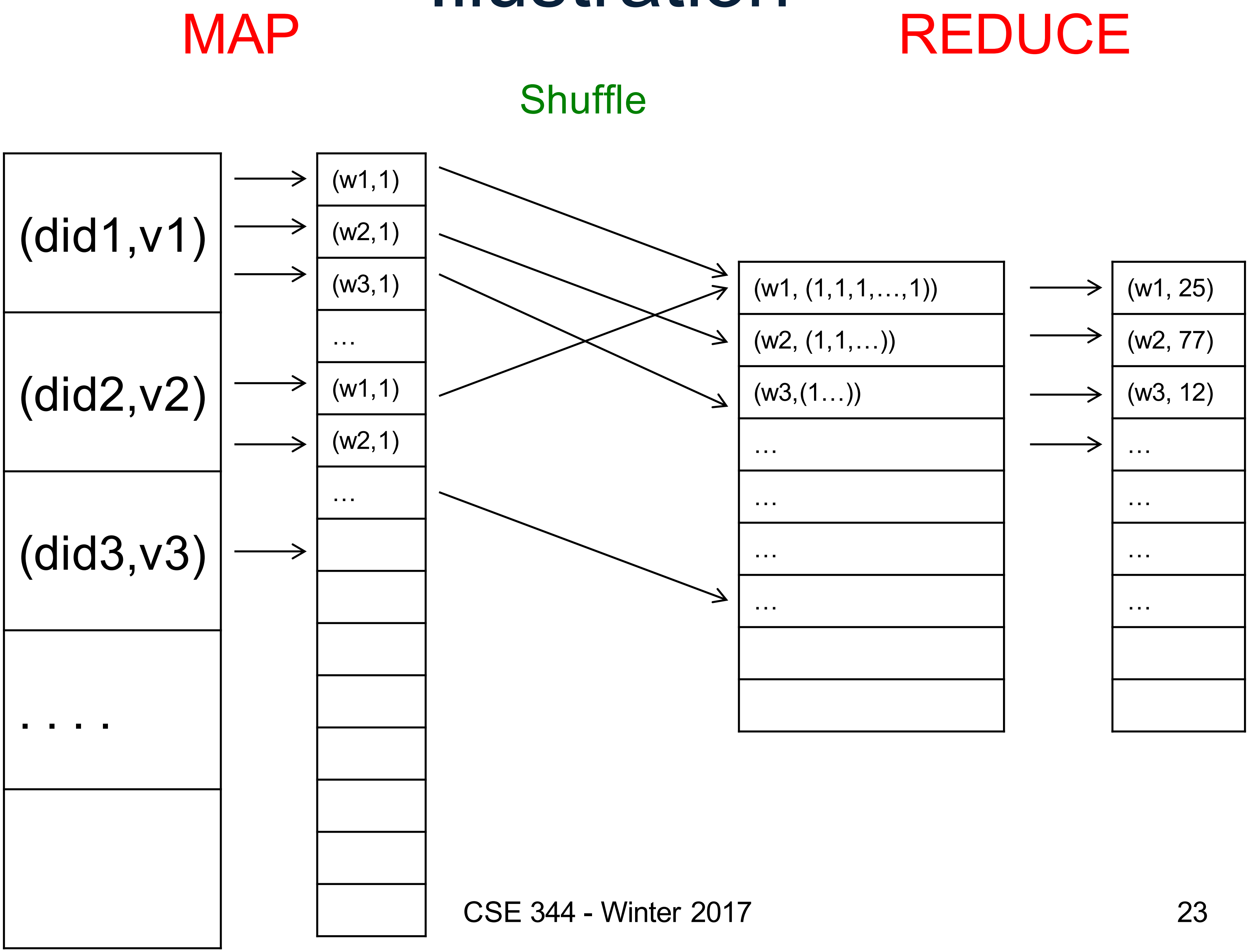
Example

- 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 "1"s
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

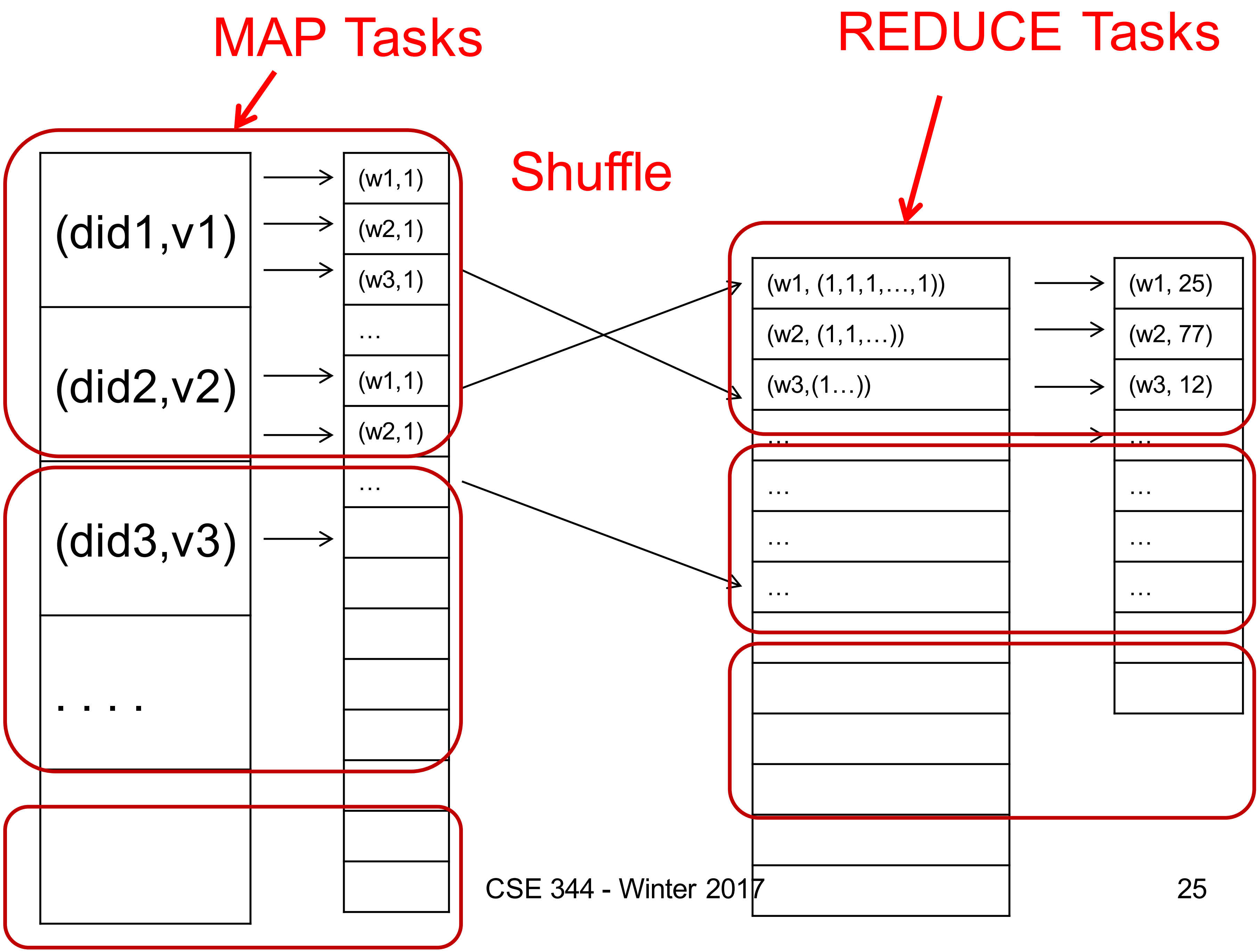
Illustration



Jobs v.s. Tasks

- A MapReduce Job
  - One single "query," e.g., count the words in all docs
  - More complex queries may consists of multiple jobs

- A Map Task, or a Reduce Task
  - A group of instantiations of the map-, or reduce-function, which are scheduled on a single worker



MAP Tasks
REDUCE Tasks
Shuffle
(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)
…
…
…
…

Workers

- A worker is a process that executes one task at a time

- Typically there is one worker per processor, hence 4 or 8 per node

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