CSE 344

MAY 2ND – MAP/REDUCE

ADMINISTRIVIA

- **HW5 Due Tonight**
- **Practice midterm**
- **Section tomorrow**
  - Exam review

PERFORMANCE METRICS FOR PARALLEL DBMSS

Nodes = processors, computers

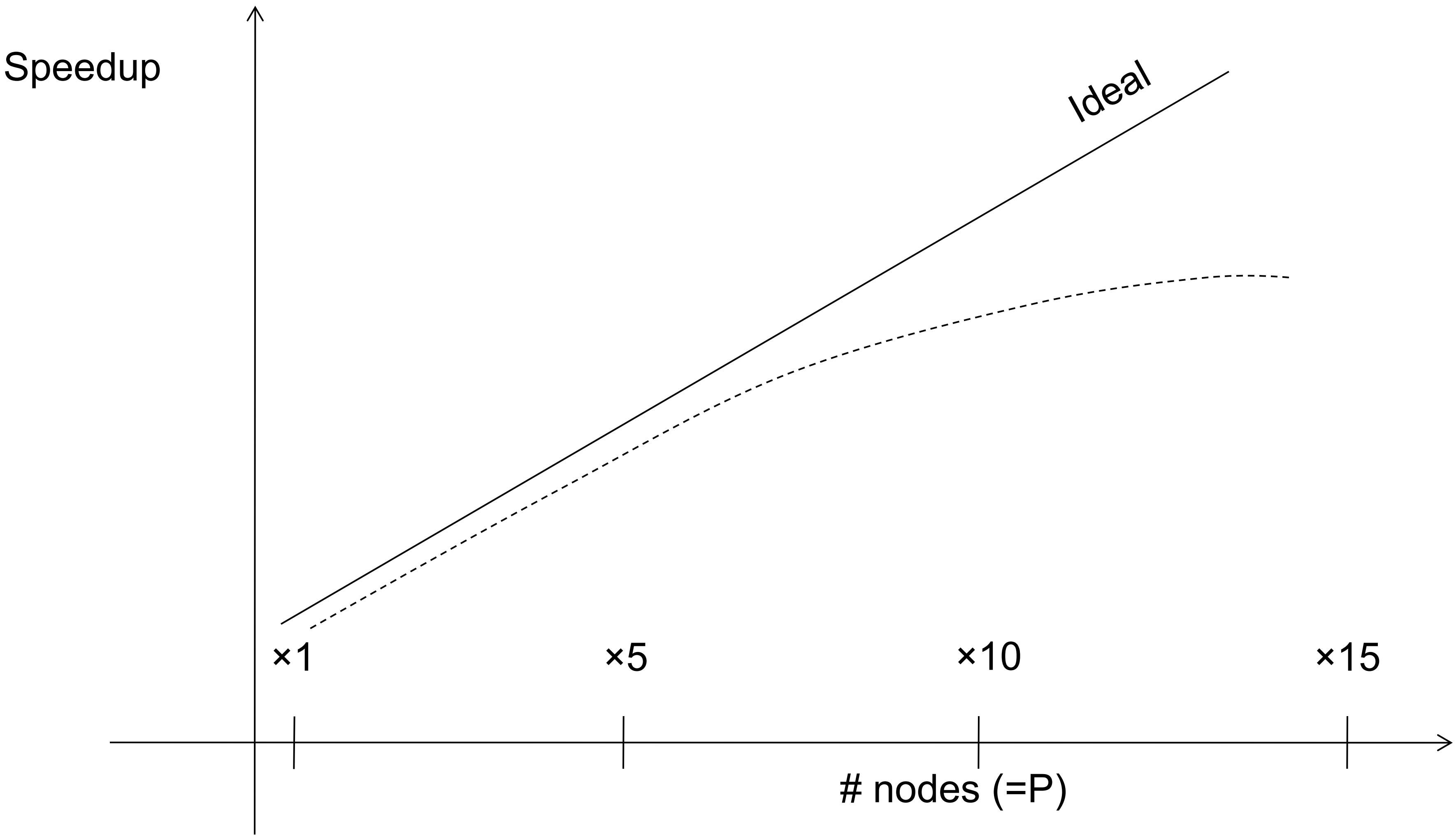
Speedup:

- More nodes, same data ➔ higher speed

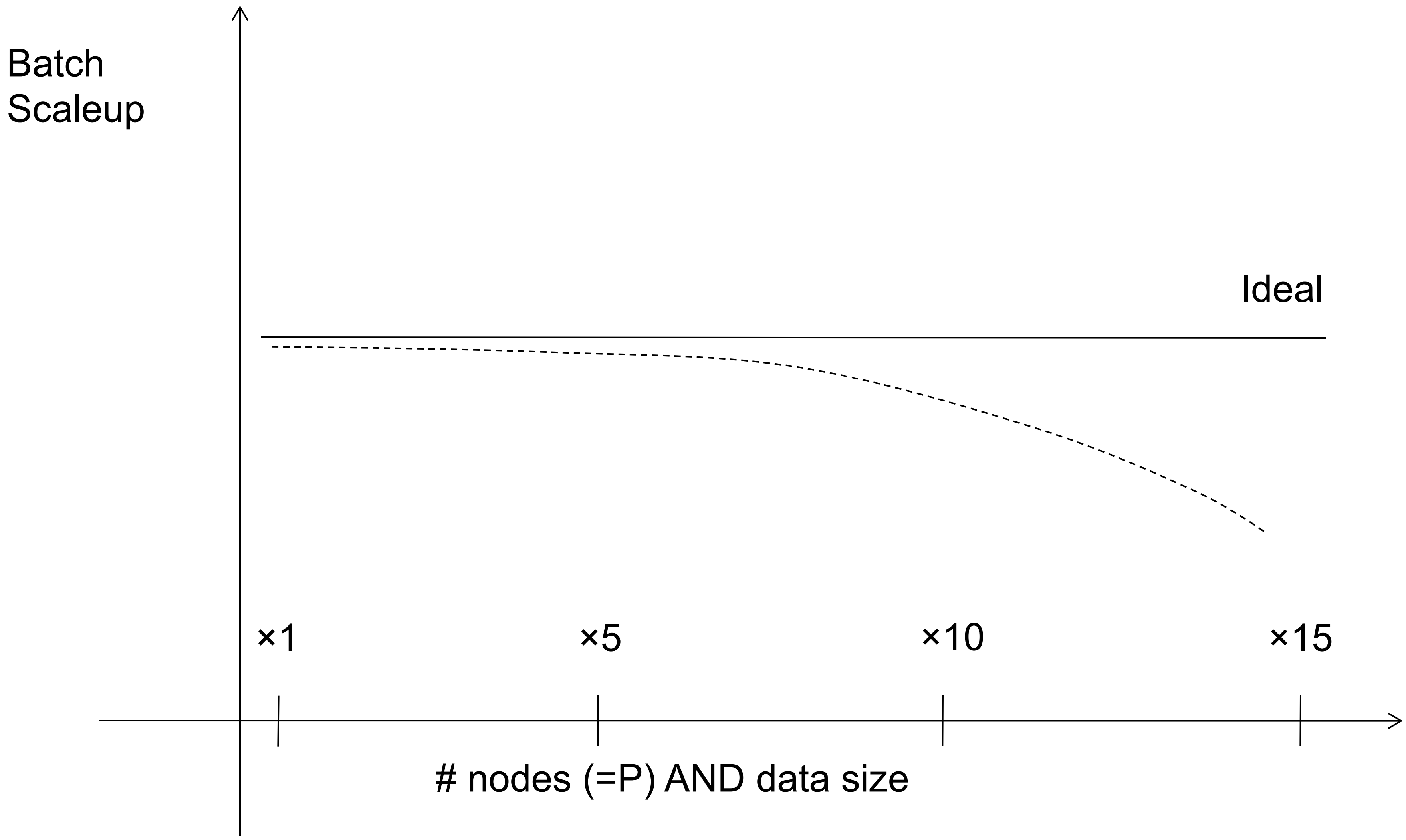
Scaleup:

- More nodes, more data ➔ same speed

LINEAR V.S. NON-LINEAR SPEEDUP



LINEAR V.S. NON-LINEAR SCALEUP



WHY SUB-LINEAR SPEEDUP AND SCALEUP?

Startup cost

- Cost of starting an operation on many nodes

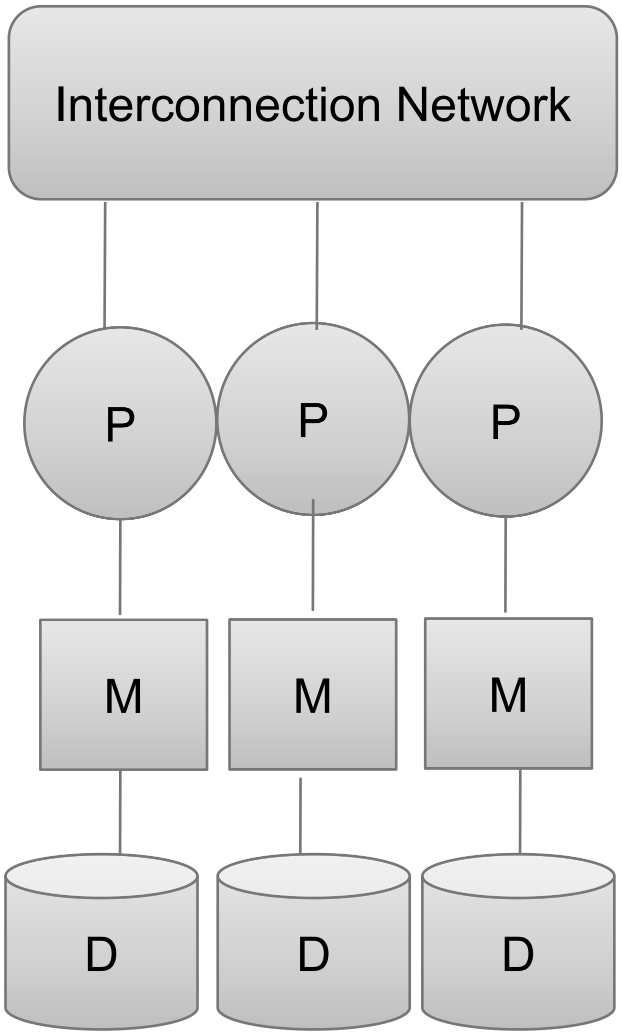
Interference

- Contention for resources between nodes

Skew

- Slowest node becomes the bottleneck

SHARED NOTHING



Cluster of commodity machines on high-speed network

Called "clusters" or "blade servers”

Each machine has its own memory and disk: lowest contention.

Example: Google

Because all machines today have many cores and many disks, shared-nothing systems typically run many "nodes” on a single physical machine.

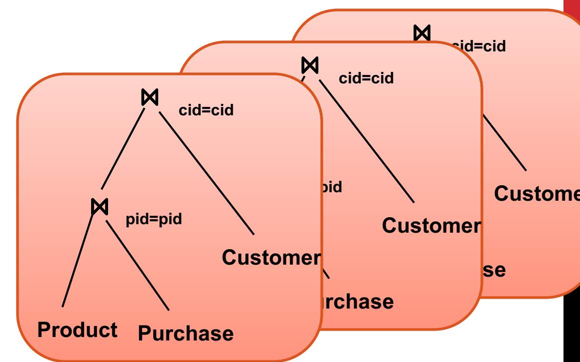
We discuss only Shared Nothing in class

Most difficult to administer and tune.

APPROACHES TO PARALLEL QUERY EVALUATION

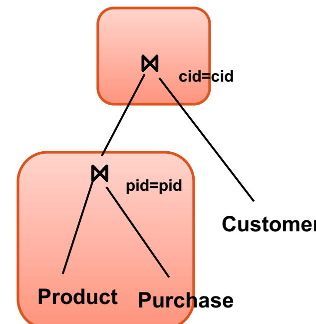
Inter-query parallelism

- Transaction per node
- Good for transactional workloads



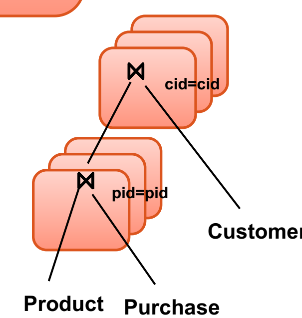
Inter-operator parallelism

- Operator per node
- Good for analytical workloads



Intra-operator parallelism

- Operator on multiple nodes
- Good for both?



We study only intra-operator parallelism: most scalable

DISTRIBUTED QUERY PROCESSING

Data is horizontally partitioned on many servers

Operators may require data reshuffling

First let's discuss how to distribute data across multiple nodes / servers

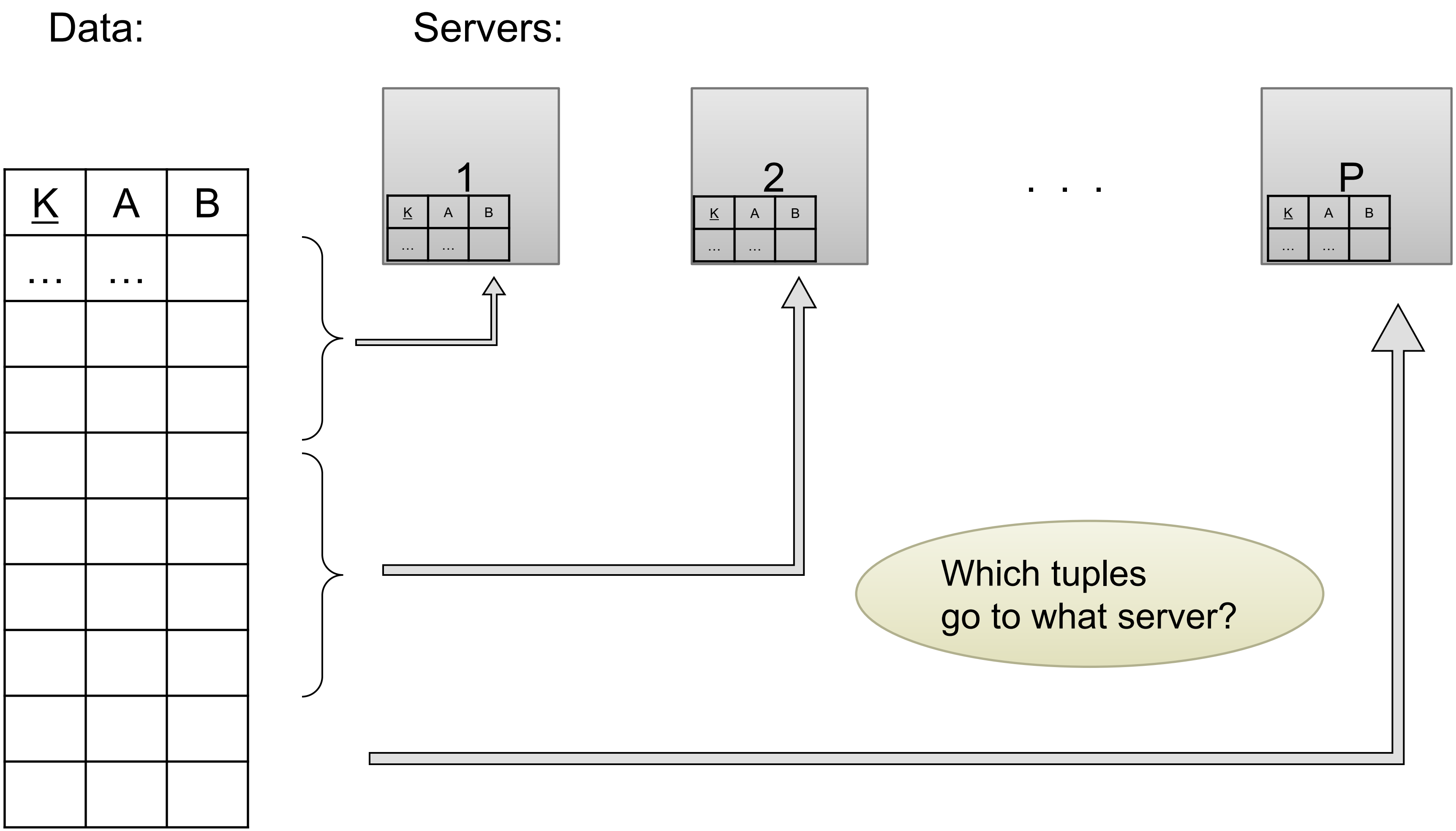
HORIZONTAL DATA PARTITIONING

Data:

<u>K</u>	A	B
…	…	

Servers:

HORIZONTAL DATA PARTITIONING



HORIZONTAL DATA PARTITIONING

Block Partition:

- Partition tuples arbitrarily s.t. size(R_1)≈ … ≈ 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(<u>K</u>,A,B,C); which of the following partition methods may result in skewed partitions?

Block partition — Uniform

Hash-partition — Uniform

Assuming good hash function

- On the key K
- On the attribute A

Range partition — May be skewed

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

Keep this in mind in the next few slides

PARALLEL EXECUTION OF RA OPERATORS: GROUPING

Data: R(<u>K</u>,A,B,C)

Query: $\gamma_{A,sum(C)}(R)$

How to compute group by if:

R is hash-partitioned on A ?

R is block-partitioned ?

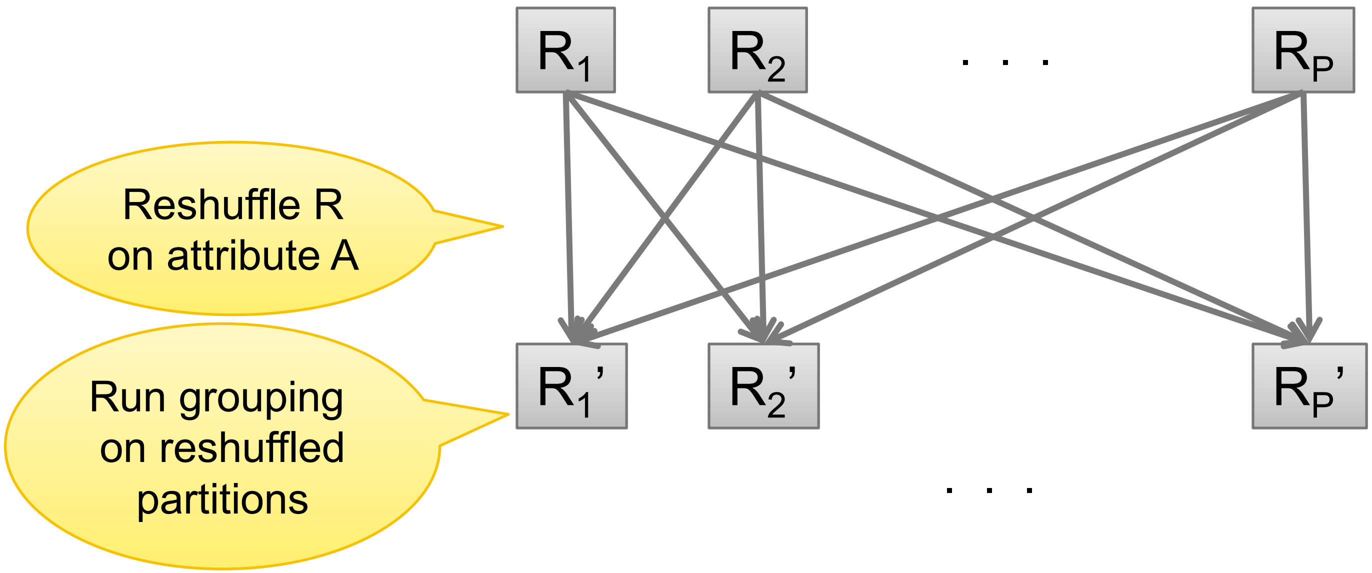
R is hash-partitioned on K ?

PARALLEL EXECUTION OF RA OPERATORS: GROUPING

Data: R(<u>K</u>,A,B,C)

Query: $\gamma_{A,sum(C)}(R)$

R is block-partitioned or hash-partitioned on K



SPEEDUP AND SCALEUP

Consider:

- Query: $\gamma_{A,sum(C)}(R)$
- Runtime: only consider I/O costs

If we double the number of nodes P, what is the new running time?

- Half (each server holds ½ as many chunks)

If we double both P and the size of R, what is the new running time?

- Same (each server holds the same # of chunks)

But only if the data is without skew!

SKEWED DATA

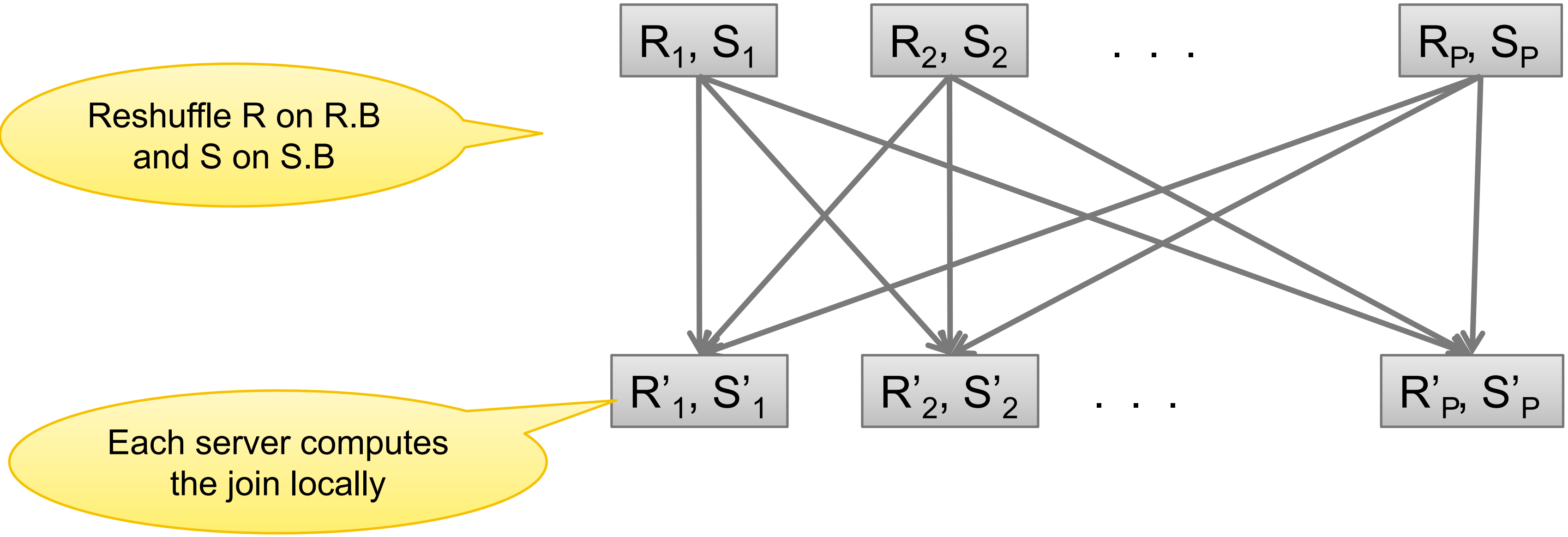
- R(<u>K</u>,A,B,C)
- Informally: we say that the data is skewed if one server holds much more data that the average
- E.g. we hash-partition on A, and some value of A occurs many times
- Then the server holding that value will be skewed

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(<u>K1</u>,A, B), S(<u>K2</u>, B, C)

Query: R(<u>K1</u>,A,B) $\bowtie$ S(<u>K2</u>,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

$\bowtie$

S1’

K2	B
201	20

M2

R2’

K1	B
2	50

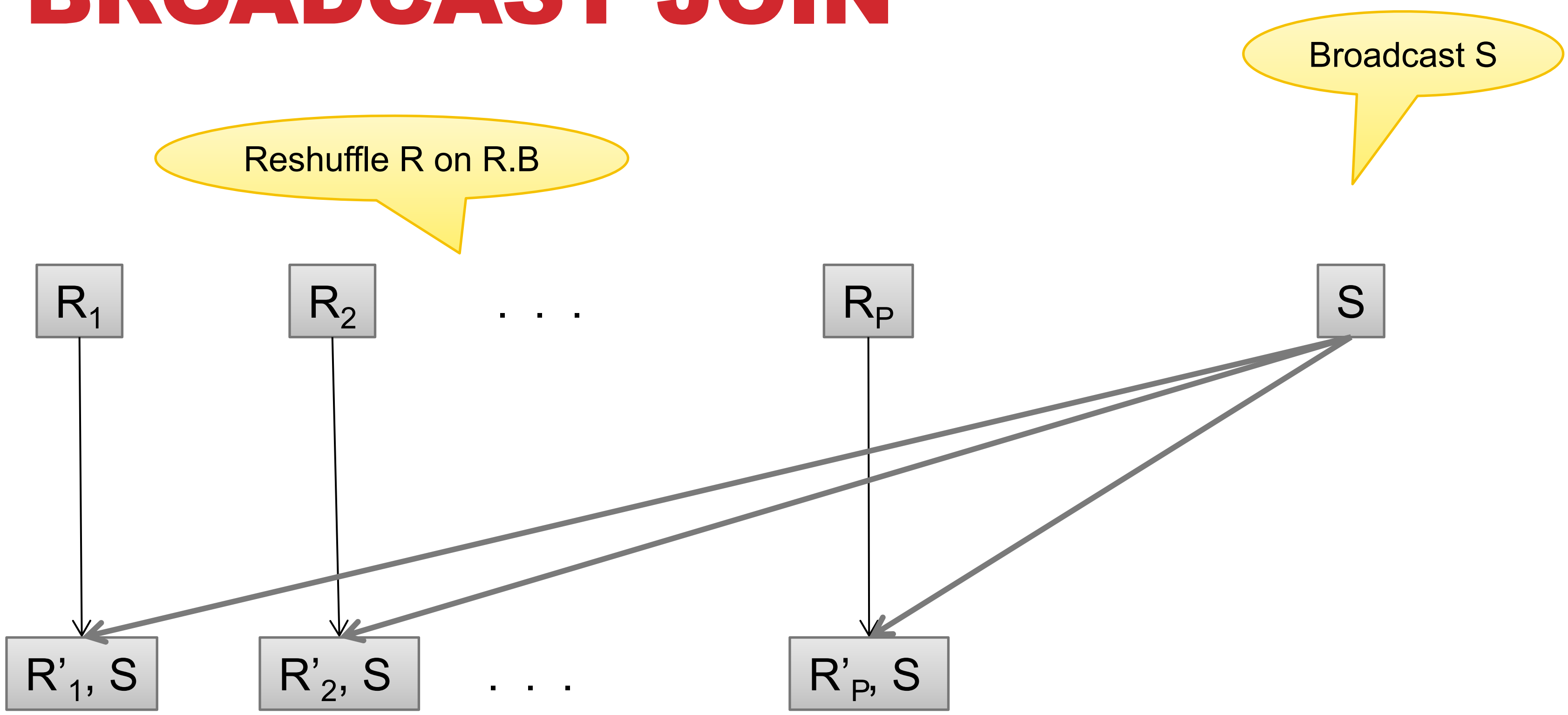
$\bowtie$

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



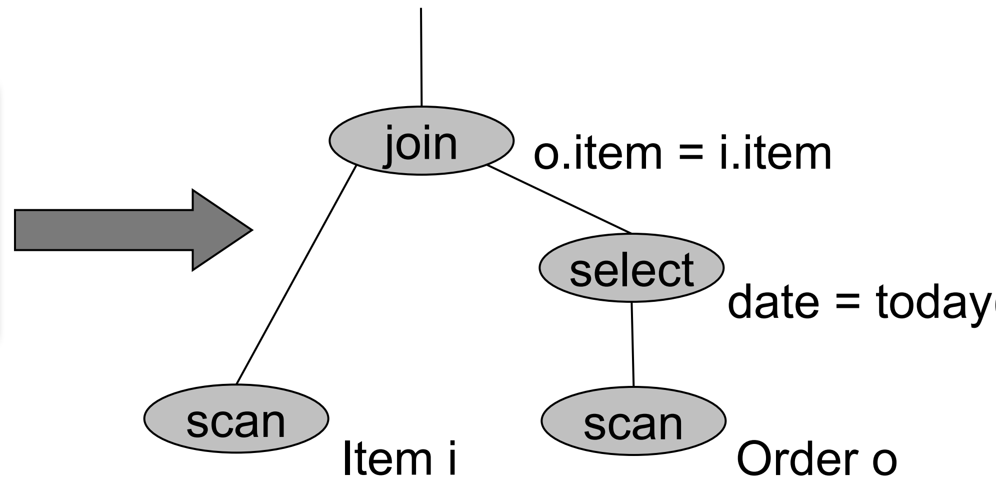
Why would you want to do this?

Order(oid, item, date), Line(item, …)

EXAMPLE PARALLEL QUERY PLAN

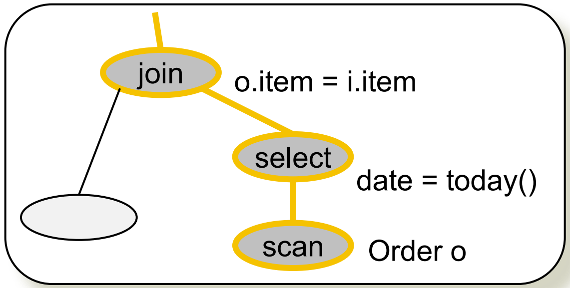
Find all orders from today, along with the items ordered

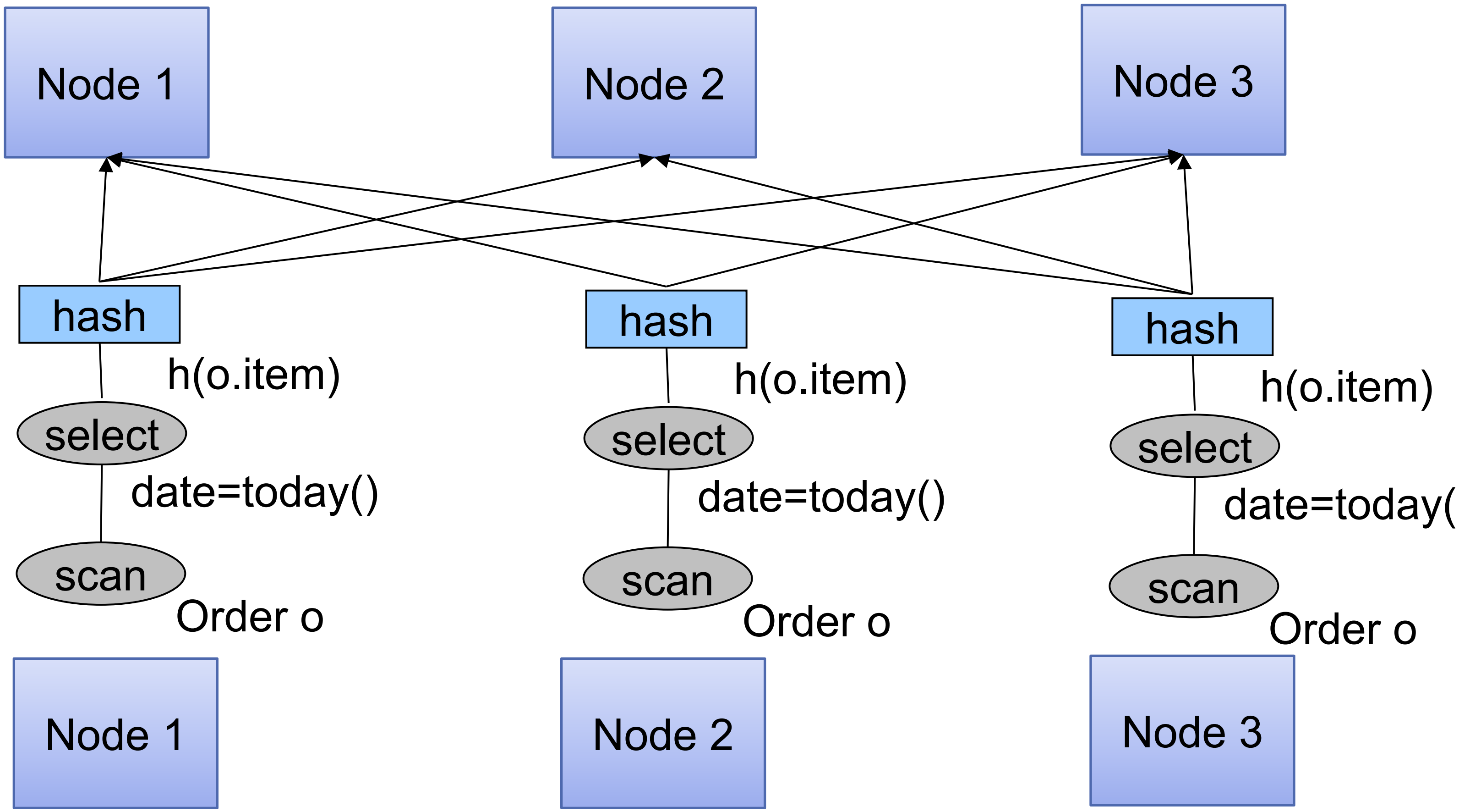
```
SELECT *
  FROM Order o, Line i
 WHERE o.item = i.item
   AND o.date = today()
```



Order(oid, item, date), Line(item, …)

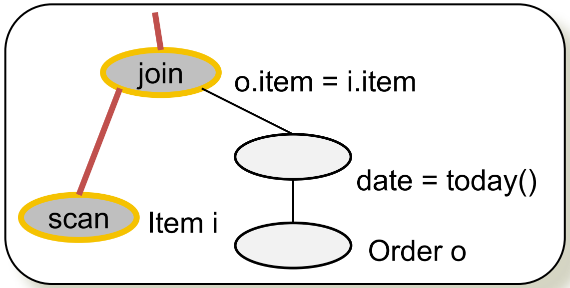
PARALLEL QUERY PLAN

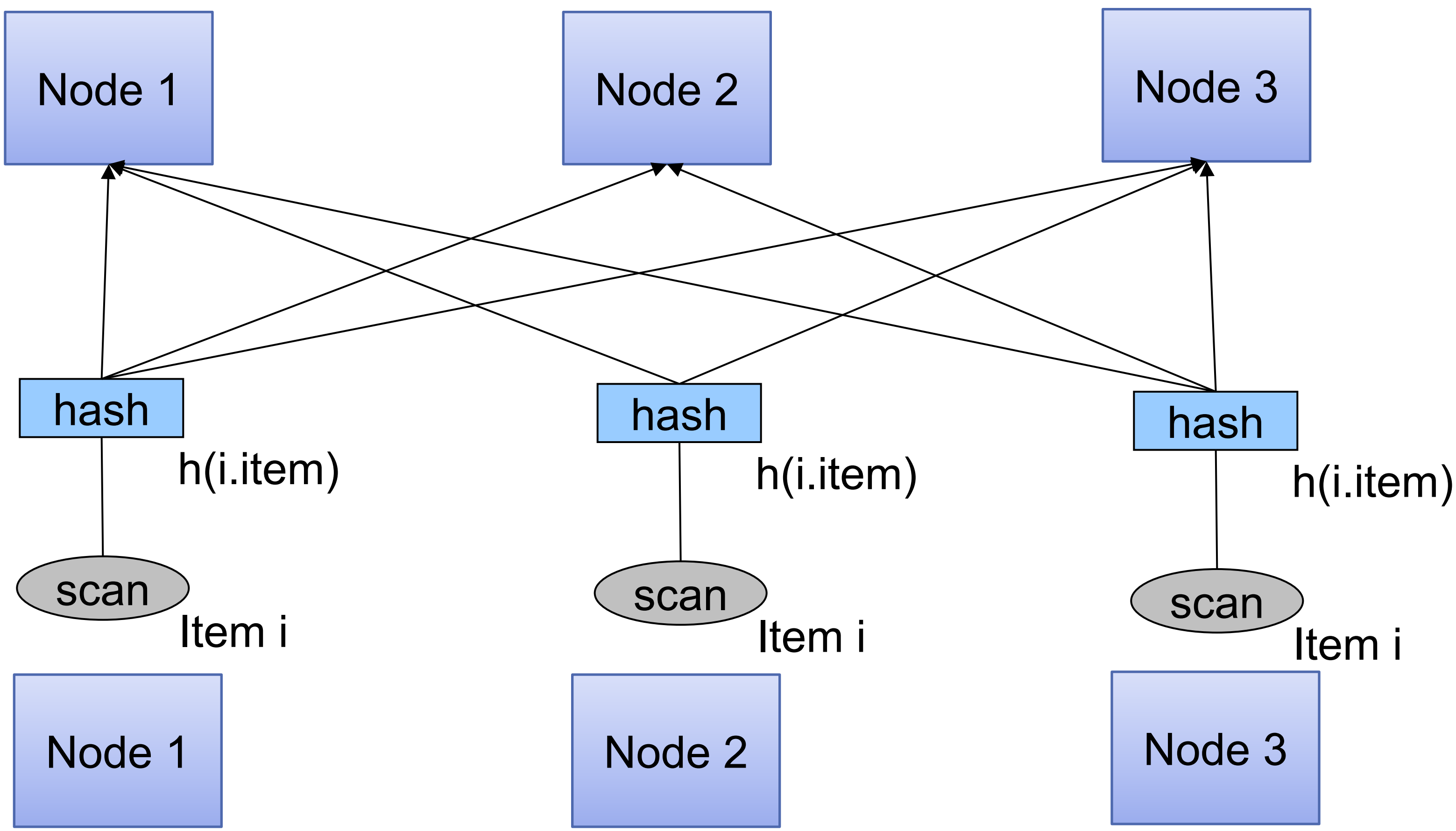




Order(oid, item, date), Line(item, …)

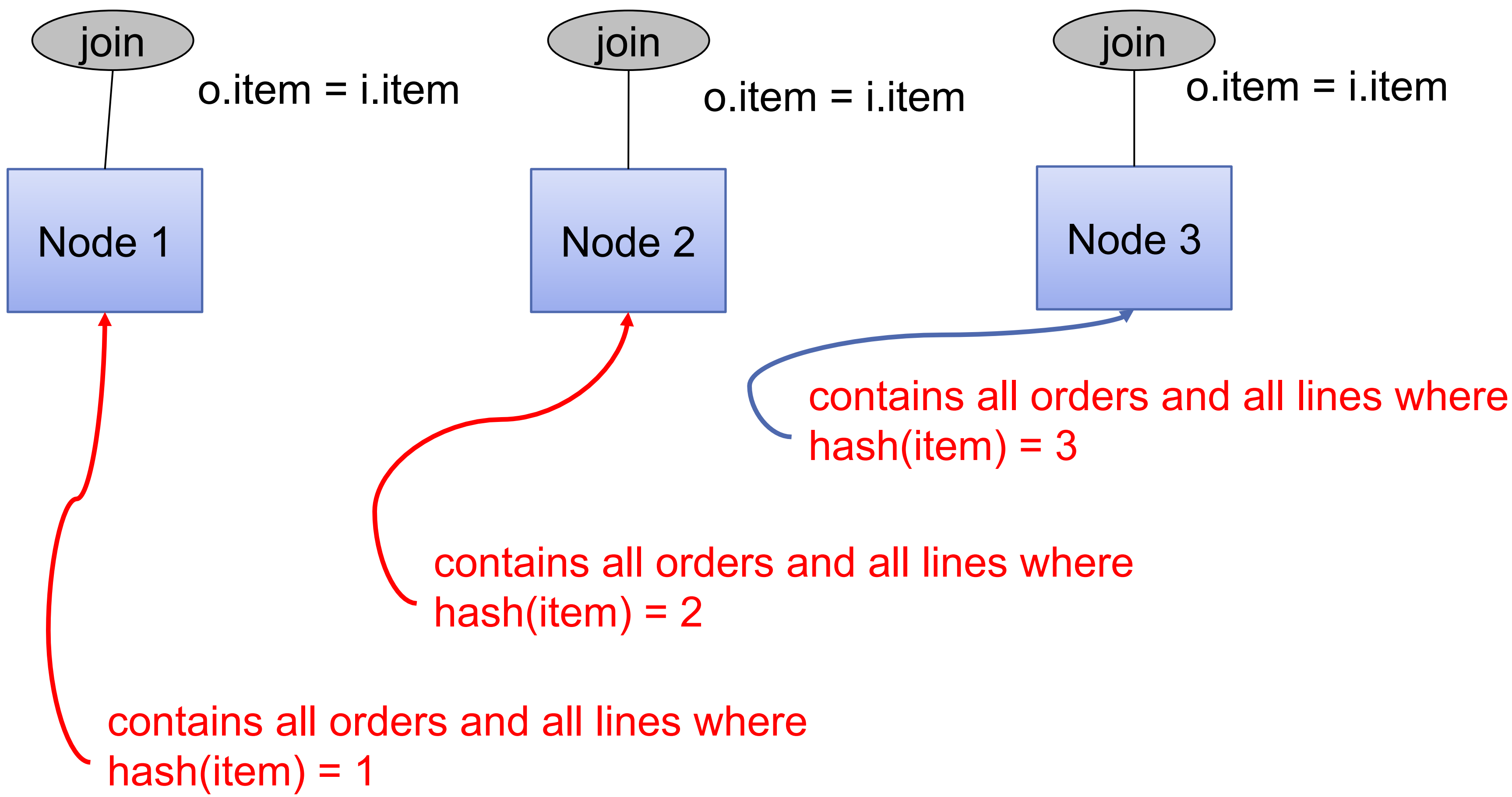
PARALLEL QUERY PLAN





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

EXAMPLE PARALLEL QUERY PLAN



MOTIVATION

We learned how to parallelize relational database systems

While useful, it might incur too much overhead if our query plans consist of simple operations

MapReduce is a programming model for such computation

First, let's study how data is stored in such systems

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

MAPREDUCE

Google: paper published 2004

Free variant: Hadoop

MapReduce = high-level programming model and implementation for large-scale parallel data processing

TYPICAL PROBLEMS SOLVED BY MR

Read a lot of data

Map: extract something you care about from each record

Shuffle and Sort

Reduce: aggregate, summarize, filter, transform

Write the results

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

DATA MODEL

Files!

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

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)`

Ouput: bag of `(intermediate key, value)`

System applies the map function 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)`

Output: bag of output `(values)`

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

EXAMPLE

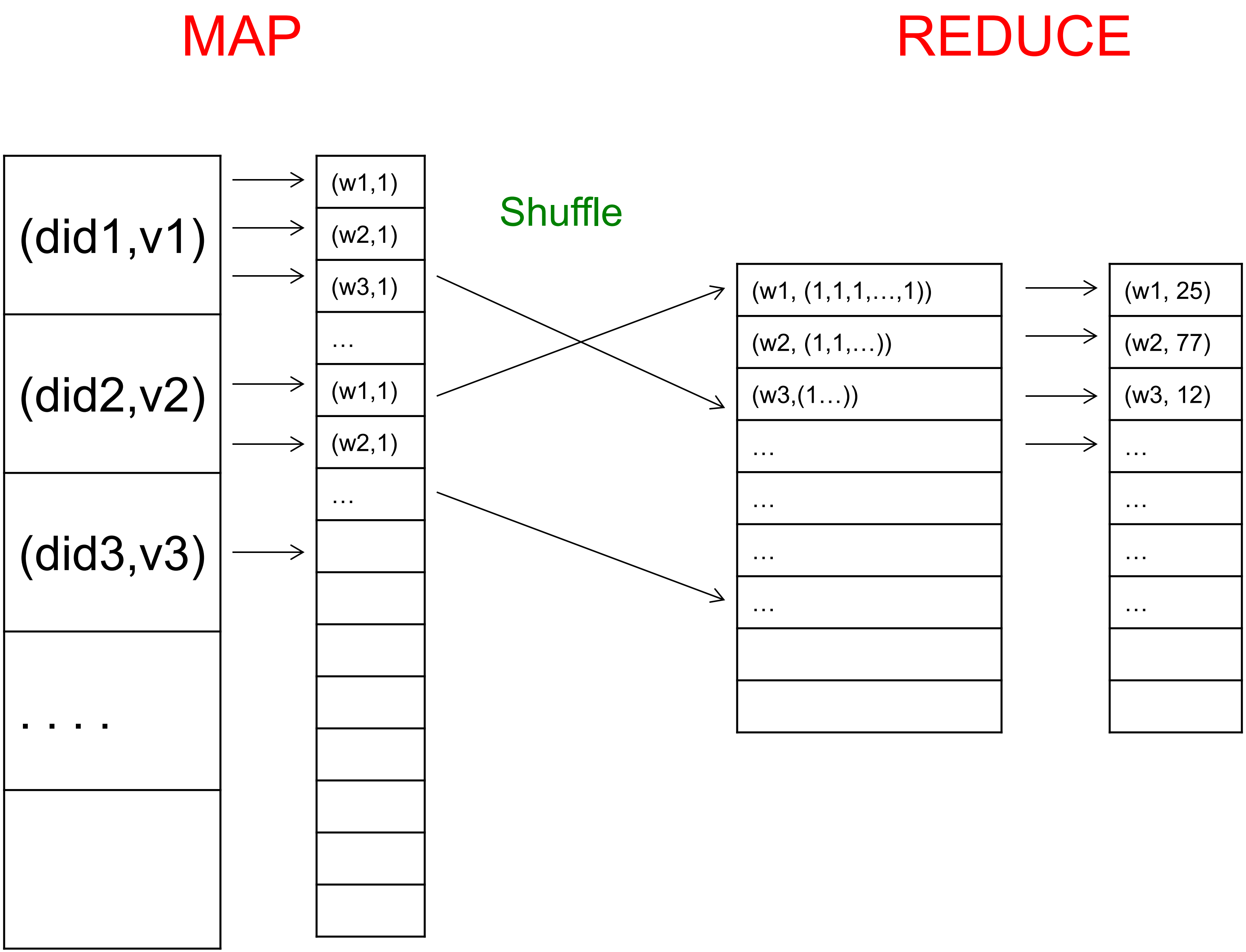
Counting the number of occurrences of each word in a large collection of documents

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
REDUCE
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)
…
…
…
…