Introduction to Data Management CSE 344

Unit 5: Parallel Data Processing

Parallel RDBMS
MapReduce
Spark

(4 lectures)

Introduction to Data Management CSE 344

Spark

Announcement

- HW6 posted
  - We use Amazon Web Services (AWS)
  - Urgent: please sign up for AWS credits (see instructions on the homework)

- No classes on Monday (Veterans' Day)

Class Overview

- Unit 1: Intro
- Unit 2: Relational Data Models and Query Languages
- Unit 3: Non-relational data
- Unit 4: RDMBS internals and query optimization
- Unit 5: Parallel query processing
  - Spark, Hadoop, parallel databases
- Unit 6: DBMS usability, conceptual design
- Unit 7: Transactions
- Unit 8: Advanced topics (time permitting)

Parallelism is of Increasing Importance

- Multi-cores:
  - Most processors have multiple cores
  - This trend will likely increase in the future

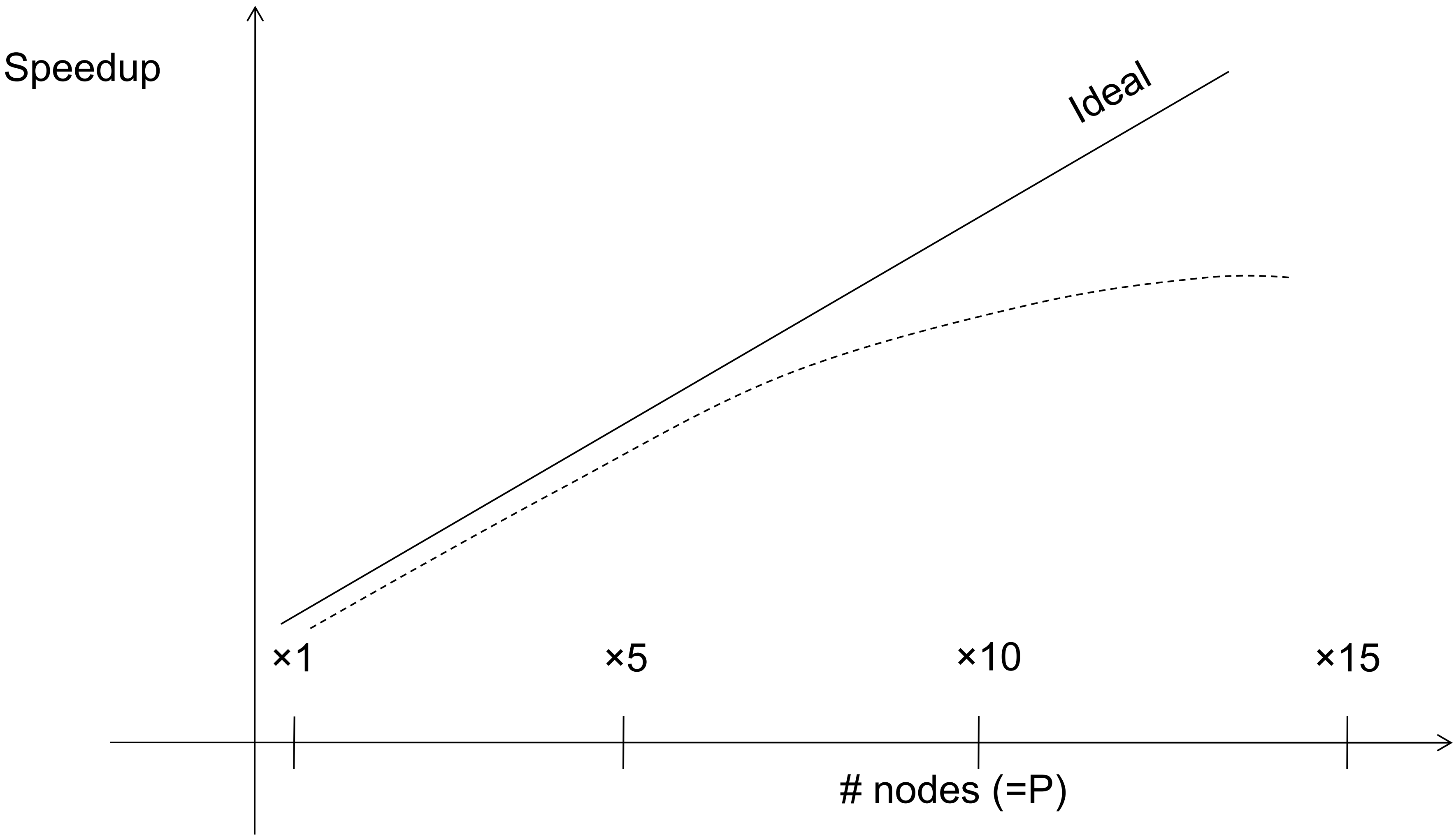
- Big data: too large to fit in main memory
  - Distributed query processing on 100x-1000x servers
  - Widely available now using cloud services

Performance Metrics for Parallel DBMSs

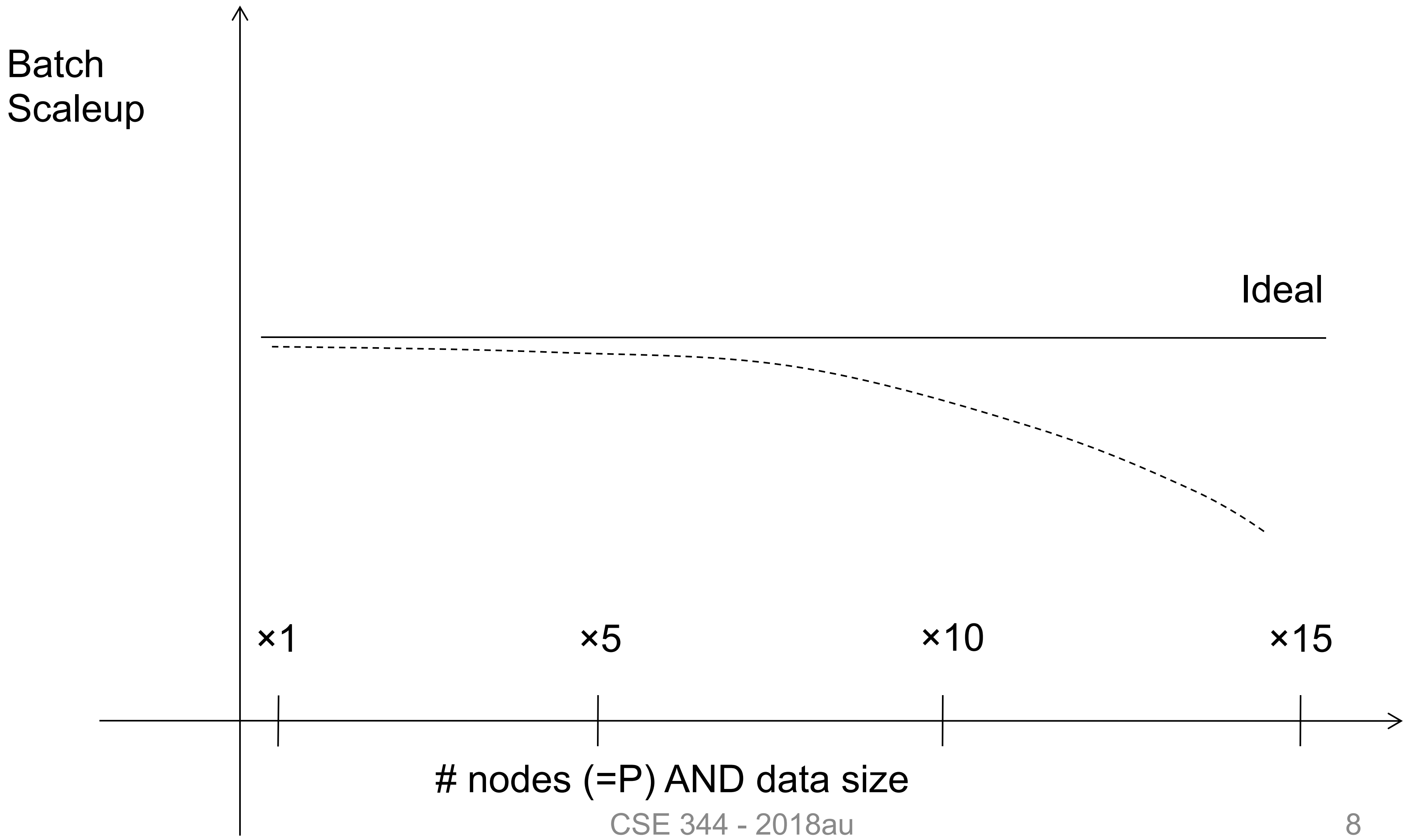
Nodes = processors, computers

- 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?

- Startup cost
  - Cost of starting an operation on many nodes
- Interference
  - Contention for resources between nodes
- Skew
  - Slowest node becomes the bottleneck

Spark

A Case Study of the MapReduce Programming Paradigm

Parallel Data Processing @ 2010

Spark

- Open source system from UC Berkeley
- Distributed processing over HDFS
- Differences from MapReduce (CSE322):
  - 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

- Spark supports interfaces in Java, Scala, and Python
  - Scala: extension of Java with functions/closures
- We will illustrate use the Spark Java interface in this class
- Spark also supports a SQL interface (SparkSQL), and compiles SQL to its native Java interface

Programming in Spark

- A Spark program consists of:
  - Transformations (map, reduce, join…). Lazy
  - Actions (count, reduce, save...). Eager

- Eager: operators are executed immediately

- Lazy: operators are not executed immediately
  - A *operator tree* is constructed in memory instead
  - Similar to a relational algebra tree

Collections in Spark

- RDD<T> = an RDD collection of type T
  - Distributed on many servers, not nested
  - Operations are done in parallel
  - Recoverable via lineage; more later
- Seq<T> = a sequence
  - Local to one server, may be nested
  - Operations are done sequentially

Example

Given a large log file hdfs://logfile.log
retrieve all lines that:

- Start with “ERROR”
- Contain the string “sqlite”

```
s = SparkSession.builder()...getOrCreate();

lines = s.read().textFile(“hdfs://logfile.log”);

errors = lines.filter(l -> l.startsWith(“ERROR”));

sqlerrors = errors.filter(l -> l.contains(“sqlite”));

sqlerrors.collect();
```

Example

Given a large log file hdfs://logfile.log retrieve all lines that:

- Start with “ERROR”
- Contain the string “sqlite”

s has type JavaRDD<String>

```
s = SparkSession.builder()...getOrCreate();

lines = s.read().textFile(“hdfs://logfile.log”);

errors = lines.filter(l -> l.startsWith(“ERROR”));

sqlerrors = errors.filter(l -> l.contains(“sqlite”));

sqlerrors.collect();
```

Example

Given a large log file hdfs://logfile.log retrieve all lines that:

- Start with “ERROR”
- Contain the string “sqlite”

`s` has type `JavaRDD<String>`

```
s = SparkSession.build                    te();

lines = s.read().textFil                   ogfile.log”);

errors = lines.filter(l     l.startsWith(“ERROR”));

sqlerrors = errors.filter(l                        e”));

sqlerrors.collect();
```

Transformation:
Not executed yet…

Action:
triggers execution
of entire program

Example

Recall: anonymous functions
(lambda expressions) starting in Java 8

```
errors = lines.filter(l -> l.startsWith(“ERROR”));
```

is the same as:

```
class FilterFn implements Function<Row, Boolean>{
  Boolean call (Row r)
  { return l.startsWith(“ERROR”); }
}

errors = lines.filter(new FilterFn());
```

Example

Given a large log file hdfs://logfile.log
retrieve all lines that:

- Start with “ERROR”
- Contain the string “sqlite”

```
s = SparkSession.builder()...getOrCreate();

sqlerrors = s.read().textFile(“hdfs://logfile.log”)
                .filter(l -> l.startsWith(“ERROR”))
                .filter(l -> l.contains(“sqlite”))
                .collect();
```

“Call chaining” style

Example

The RDD s:

Error…	Warning…	Warning…	Error…	Abort…	Abort…	Error…	Error…	Warning…	Error…

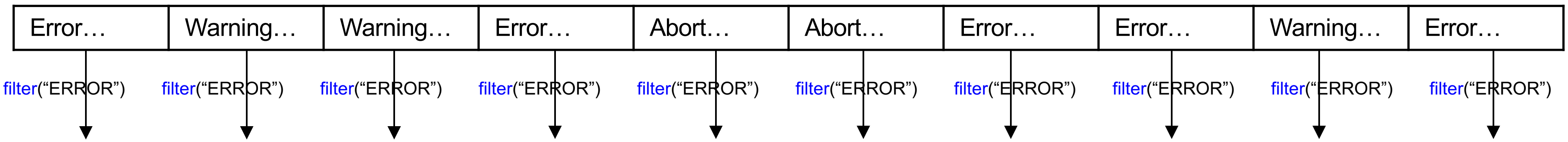
```
s = SparkSession.builder()...getOrCreate();

sqlerrors = s.read().textFile("hdfs://logfile.log")
             .filter(l -> l.startsWith("ERROR"))
             .filter(l -> l.contains("sqlite"))
             .collect();
```

Example

Parallel step 1

The RDD s:



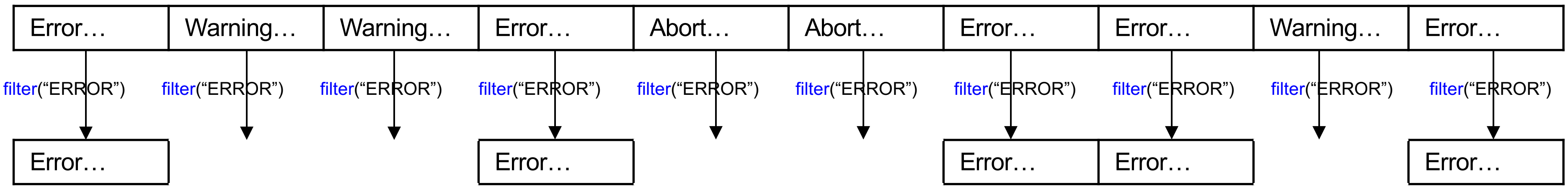

```
s = SparkSession.builder()...getOrCreate();

sqlerrors = s.read().textFile(“hdfs://logfile.log”)
                .filter(l -> l.startsWith(“ERROR”))
                .filter(l -> l.contains(“sqlite”))
                .collect();
```

Example

Parallel step 1

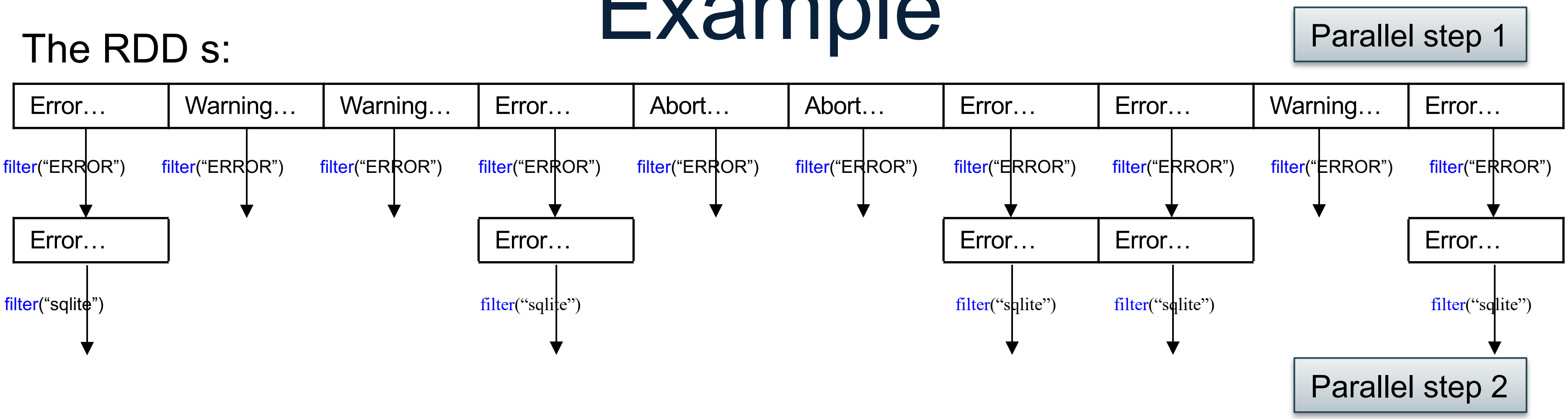
The RDD s:




```
s = SparkSession.builder()...getOrCreate();

sqlerrors = s.read().textFile("hdfs://logfile.log")
              .filter(l -> l.startsWith("ERROR"))
              .filter(l -> l.contains("sqlite"))
              .collect();
```

Example




```
s = SparkSession.builder()...getOrCreate();

sqlerrors = s.read().textFile("hdfs://logfile.log")
              .filter(l -> l.startsWith("ERROR"))
              .filter(l -> l.contains("sqlite"))
              .collect();
```

Fault Tolerance

- When a job is executed on x100 or x1000 servers, the probability of a failure is high
- Example: if a server fails once/year, then a job with 10000 servers fails once/hour
- Different solutions:
  – Parallel database systems: restart. Expensive.
  – MapReduce: write everything to disk, redo. Slow.
  – Spark: redo only what is needed. Efficient.

Resilient Distributed Datasets

- RDD = Resilient Distributed Dataset
  - Distributed, immutable and records 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

Persistence

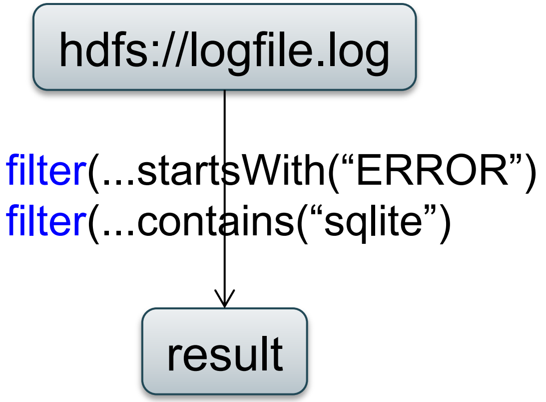
```
lines = s.read().textFile("hdfs://logfile.log");
errors = lines.filter(l->l.startsWith("ERROR"));
sqlerrors = errors.filter(l->l.contains("sqlite"));
sqlerrors.collect();
```

If any server fails before the end, then Spark must restart

Persistence

```
lines = s.read().textFile("hdfs://logfile.log");
errors = lines.filter(l->l.startsWith("ERROR"));
sqlerrors = errors.filter(l->l.contains("sqlite"));
sqlerrors.collect();
```

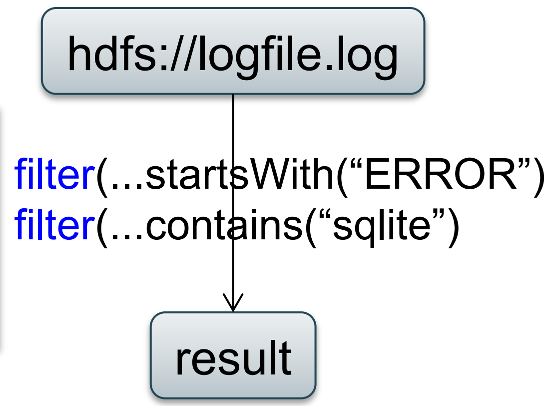
RDD:



If any server fails before the end, then Spark must restart

Persistence

RDD:




```
lines = s.read().textFile("hdfs://logfile.log");
errors = lines.filter(l->l.startsWith("ERROR"));
sqlerrors = errors.filter(l->l.contains("sqlite"));
sqlerrors.collect();
```

If any server fails before the end, then Spark must restart

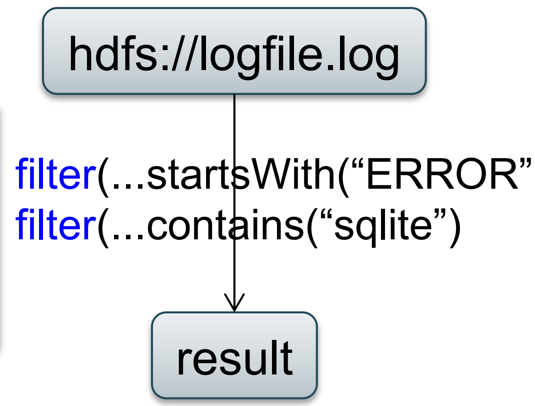
```
lines = s.read().textFile("hdfs://logfile.log");
errors = lines.filter(l->l.startsWith("ERROR"));
errors.persist();
sqlerrors = errors.filter(l->l.contains("sqlite"));
sqlerrors.collect()
```

New RDD

Spark can recompute the result from errors

Persistence

RDD:

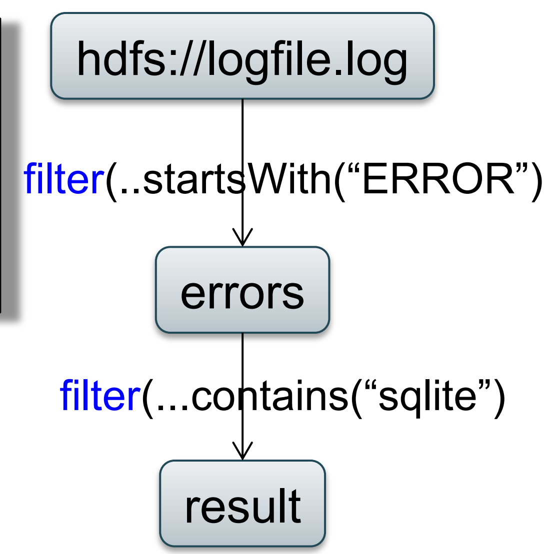



```
lines = s.read().textFile(“hdfs://logfile.log”);
errors = lines.filter(l->l.startsWith(“ERROR”));
sqlerrors = errors.filter(l->l.contains(“sqlite”));
sqlerrors.collect();
```

If any server fails before the end, then Spark must restart

```
lines = s.read().textFile(“hdfs://logfile.log”);
errors = lines.filter(l->l.startsWith(“ERROR”));
errors.persist();
sqlerrors = errors.filter(l->l.contains(“sqlite”));
sqlerrors.collect()
```

New RDD



Spark can recompute the result from errors

R(A,B)
S(A,C)

```
SELECT count(*)  FROM R, S
WHERE R.B > 200 and S.C < 100  and R.A = S.A
```

Example

```
R = strm.read().textFile("R.csv").map(parseRecord).persist();
S = strm.read().textFile("S.csv").map(parseRecord).persist();
```

Parses each line into an object

persisting on disk

R(A,B)
S(A,C)

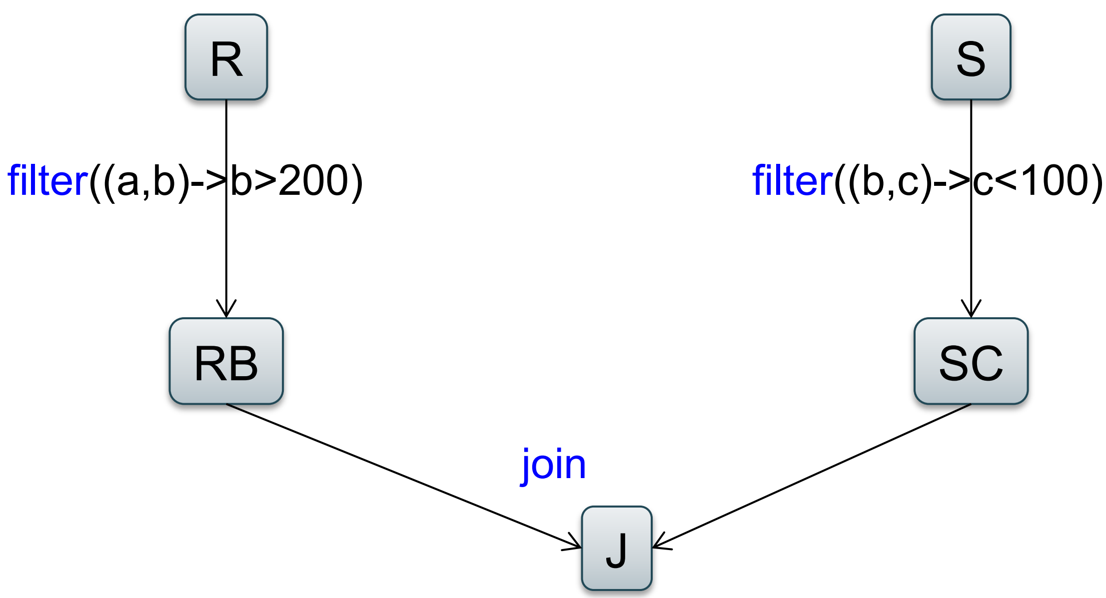
```
SELECT count(*)  FROM R, S
WHERE R.B > 200 and S.C < 100  and R.A = S.A
```

Example

```
R = strm.read().textFile("R.csv").map(parseRecord).persist();
S = strm.read().textFile("S.csv").map(parseRecord).persist();
RB = R.filter(t -> t.b > 200).persist();
SC = S.filter(t -> t.c < 100).persist();
J = RB.join(SC).persist();
J.count();
```

transformations

action



Recap: Programming in Spark

- A Spark/Scala program consists of:
  - Transformations (map, reduce, join…). Lazy
  - Actions (count, reduce, save...). Eager
- RDD<T> = an RDD collection of type T
  - Partitioned, recoverable (through lineage), not nested
- Seq<T> = a sequence
  - Local to a server, may be nested

Transformations:	
map(f : T -> U):	RDD<T> -> RDD<U>
flatMap(f: T -> Seq(U)):	RDD<T> -> RDD<U>
filter(f:T->Bool):	RDD<T> -> RDD<T>
groupByKey():	RDD<(K,V)> -> RDD<(K,Seq[V])>
reduceByKey(F:(V,V)-> V):	RDD<(K,V)> -> RDD<(K,V)>
union():	(RDD<T>,RDD<T>) -> RDD<T>
join():	(RDD<(K,V)>,RDD<(K,W)>) -> RDD<(K,(V,W))>
cogroup():	(RDD<(K,V)>,RDD<(K,W)>)-> RDD<(K,(Seq<V>,Seq<W>))>
crossProduct():	(RDD<T>,RDD<U>) -> RDD<(T,U)>

Actions:	
count():	RDD<T> -> Long
collect():	RDD<T> -> Seq<T>
reduce(f:(T,T)->T):	RDD<T> -> T
save(path:String):	Outputs RDD to a storage system e.g., HDFS

Spark 2.0

The DataFrame and Dataset Interfaces

DataFrames

- Like RDD, also an immutable distributed collection of data
- Organized into *named columns* rather than individual objects
  - Just like a relation
  - Elements are untyped objects called `Row`’s
- Similar API as RDDs with additional methods
  - ```
    people = spark.read().textFile(…);
    ageCol = people.col(“age”);
    ageCol.plus(10); // creates a new DataFrame
    ```
    ```

Datasets

- Similar to DataFrames, except that elements must be typed objects
- E.g.: `Dataset<People>` rather than `Dataset<Row>`
- Can detect errors during compilation time
- DataFrames are aliased as `Dataset<Row>` (as of Spark 2.0)
- You will use both Datasets and RDD APIs in HW6

Datasets API: Sample Methods

- Functional API
  - `agg(Column expr, Column... exprs)`
    Aggregates on the entire Dataset without groups.
  - `groupBy(String col1, String... cols)`
    Groups the Dataset using the specified columns, so that we can run aggregation on them.
  - `join(Dataset<?> right)`
    Join with another DataFrame.
  - `orderBy(Column... sortExprs)`
    Returns a new Dataset sorted by the given expressions.
  - `select(Column... cols)`
    Selects a set of column based expressions.
- "SQL" API
  - `SparkSession.sql("select * from R");`
- Look familiar?

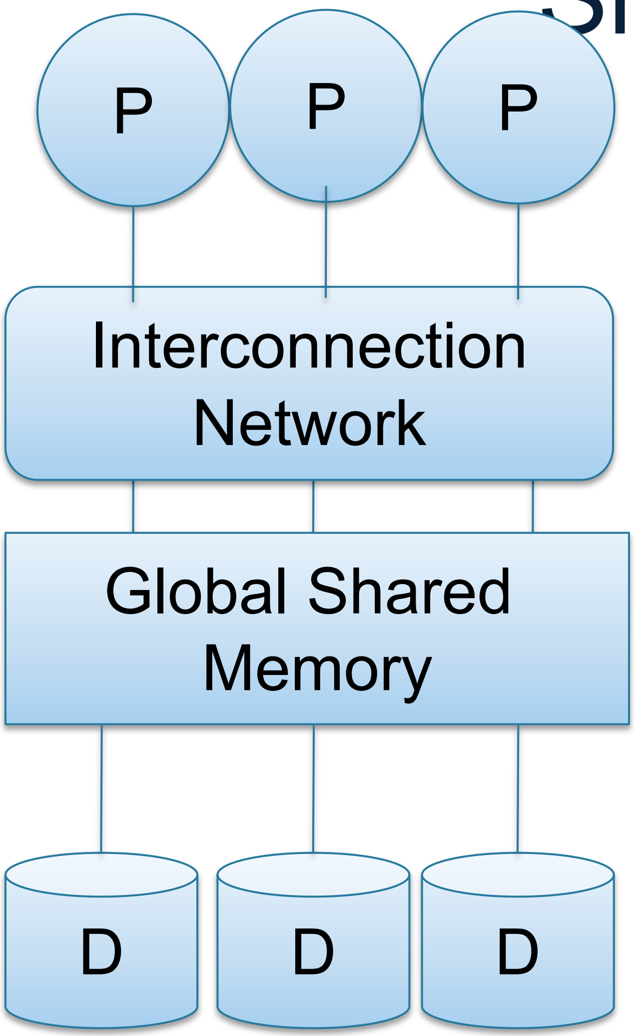
Introduction to Data Management CSE 344

Parallel Databases

Architectures for Parallel Databases

- Shared memory
- Shared disk
- Shared nothing

Shared Memory



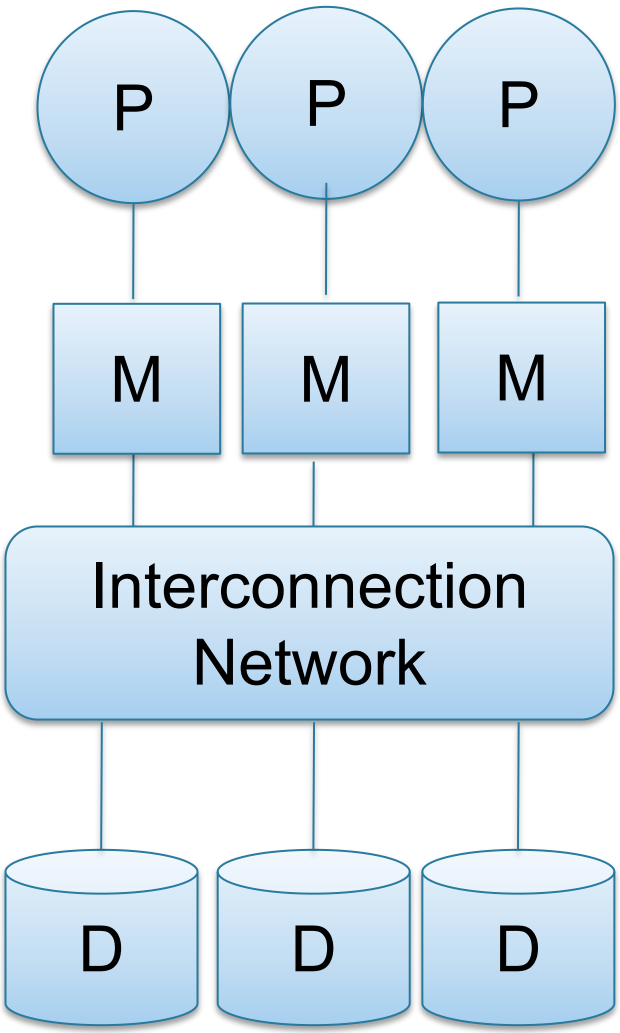
- Nodes share both RAM and disk
- Dozens to hundreds of processors

Example: SQL Server runs on a single machine and can leverage many threads to speed up a query

- check your HW3 query plans

- Easy to use and program
- Expensive to scale

Shared Disk



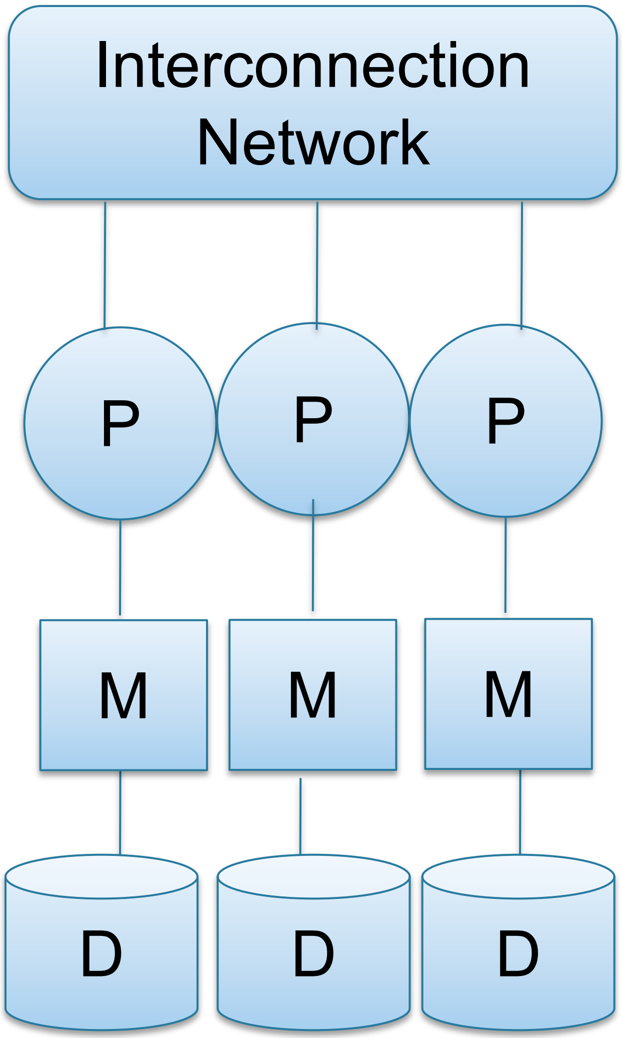
- All nodes access the same disks
- Found in the largest "single-box" (non-cluster) multiprocessors

Example: Oracle

- No more memory contention

- Harder to program
- Still hard to scale: existing deployments typically have fewer than 10 machines

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: Spark

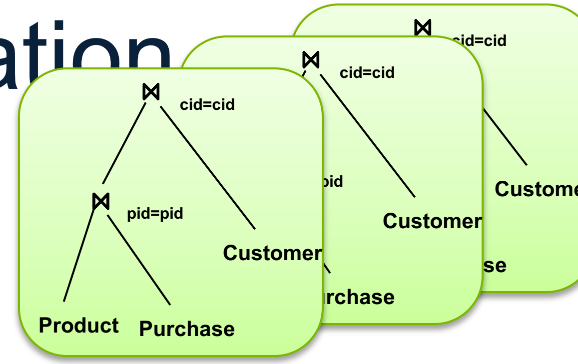
Because all machines today have many cores and many disks, shared-nothing systems typically run many "nodes” on a single physical machine.

- Easy to maintain and scale
- Most difficult to administer and tune.

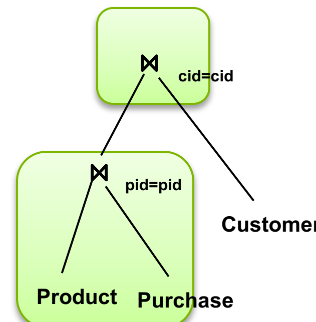
We discuss only Shared Nothing in class

Approaches to Parallel Query Evaluation

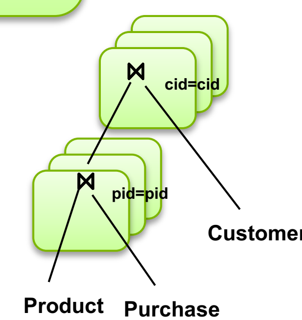
- Inter-query parallelism
  - Transaction per node
  - Good for transactional workloads



- 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

Single Node Query Processing (Review)

Given relations R(A,B) and S(B, C), no indexes:

- Selection: $\sigma_{A=123}(R)$
  - Scan file R, select records with A=123
- Group-by: $\gamma_{A,sum(B)}(R)$
  - Scan file R, insert into a hash table using A as key
  - When a new key is equal to an existing one, add B to the value
- Join: $R \bowtie S$
  - Scan file S, insert into a hash table using B as key
  - Scan file R, probe the hash table using B

Distributed Query Processing

- Data is horizontally partitioned on servers

- Operators may require data reshuffling

Horizontal Data Partitioning

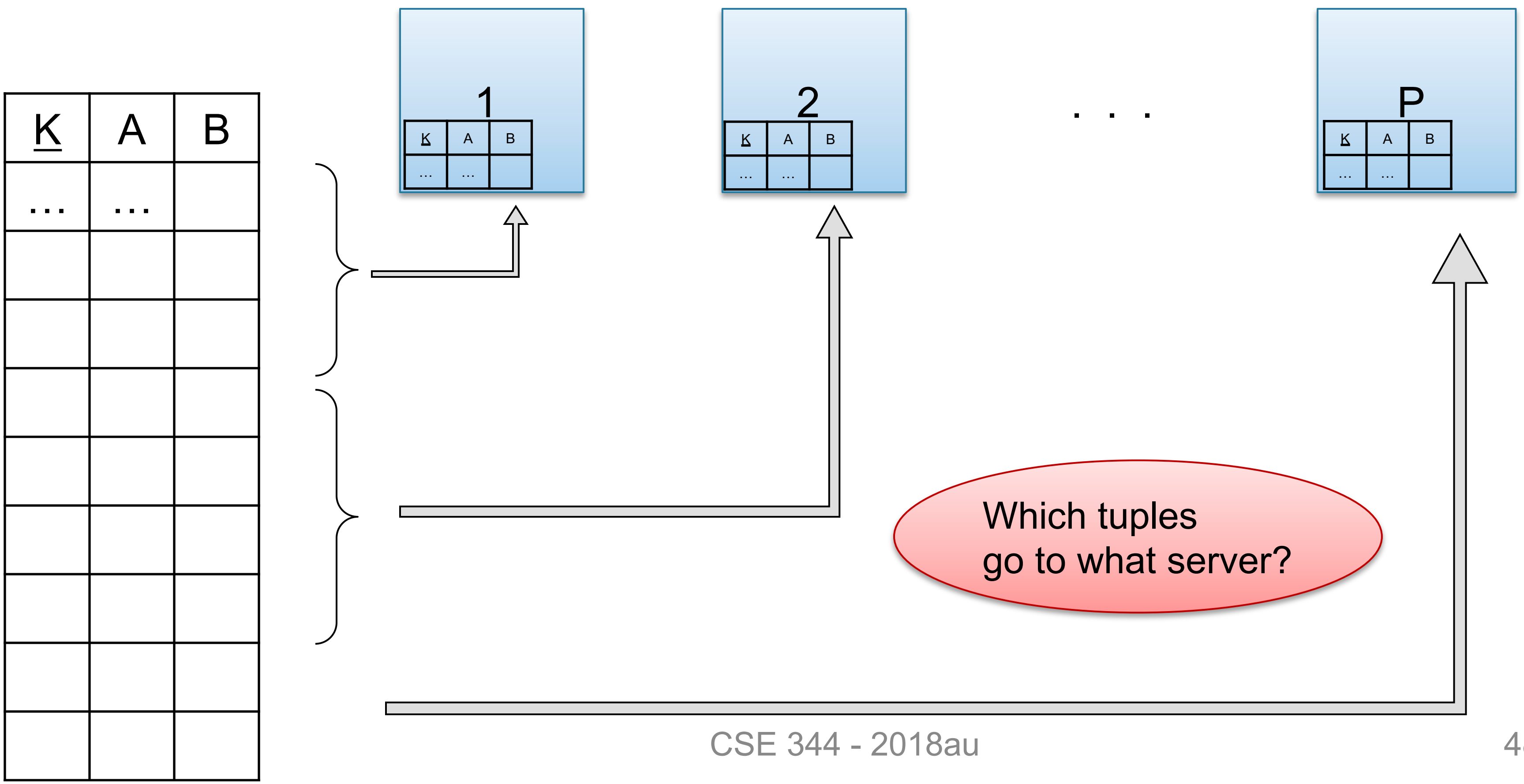
Data:

K	A	B
…	…	

Servers:

Horizontal Data Partitioning

Data: Servers:



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 tition

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:

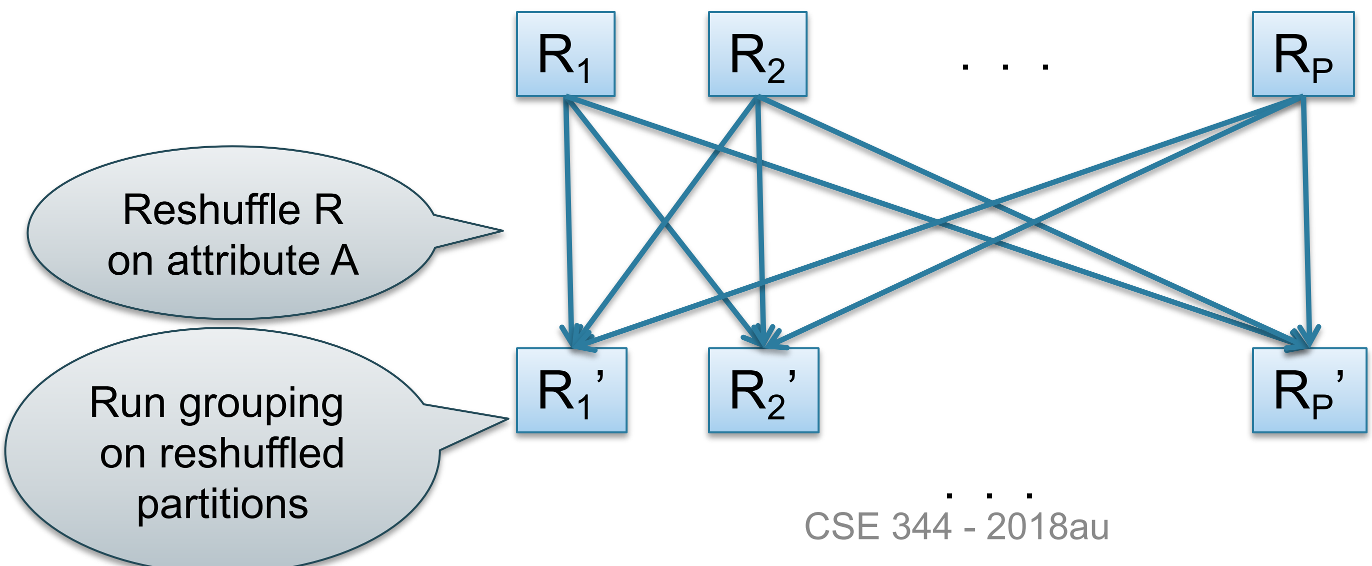
- R is hash-partitioned on A ?
- R is block-partitioned ?
- R is hash-partitioned on K ?

Parallel Execution of RA Operators: Grouping

Data: R(K,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 records)
- If we double both P and the size of R, what is the new running time?
  - Same (each server holds the same # of records)

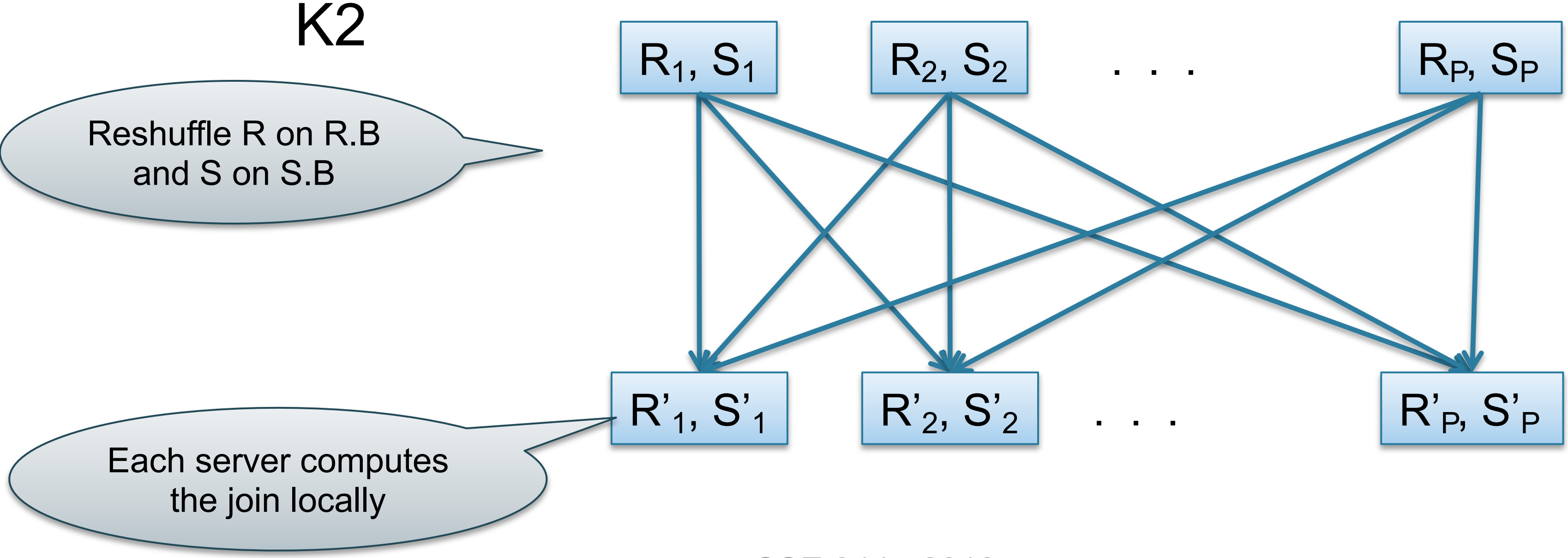
But only if the data is without skew!

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 very many times (“Justin Bieber”)
- Then the server holding that value will be skewed

Parallel Execution of RA Operators: Partitioned Hash-Join

- Data: R(K1, A, B), S(K2, B, C)
- Query: R(K1, A, B) $\bowtie$ S(K2, 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

⋈

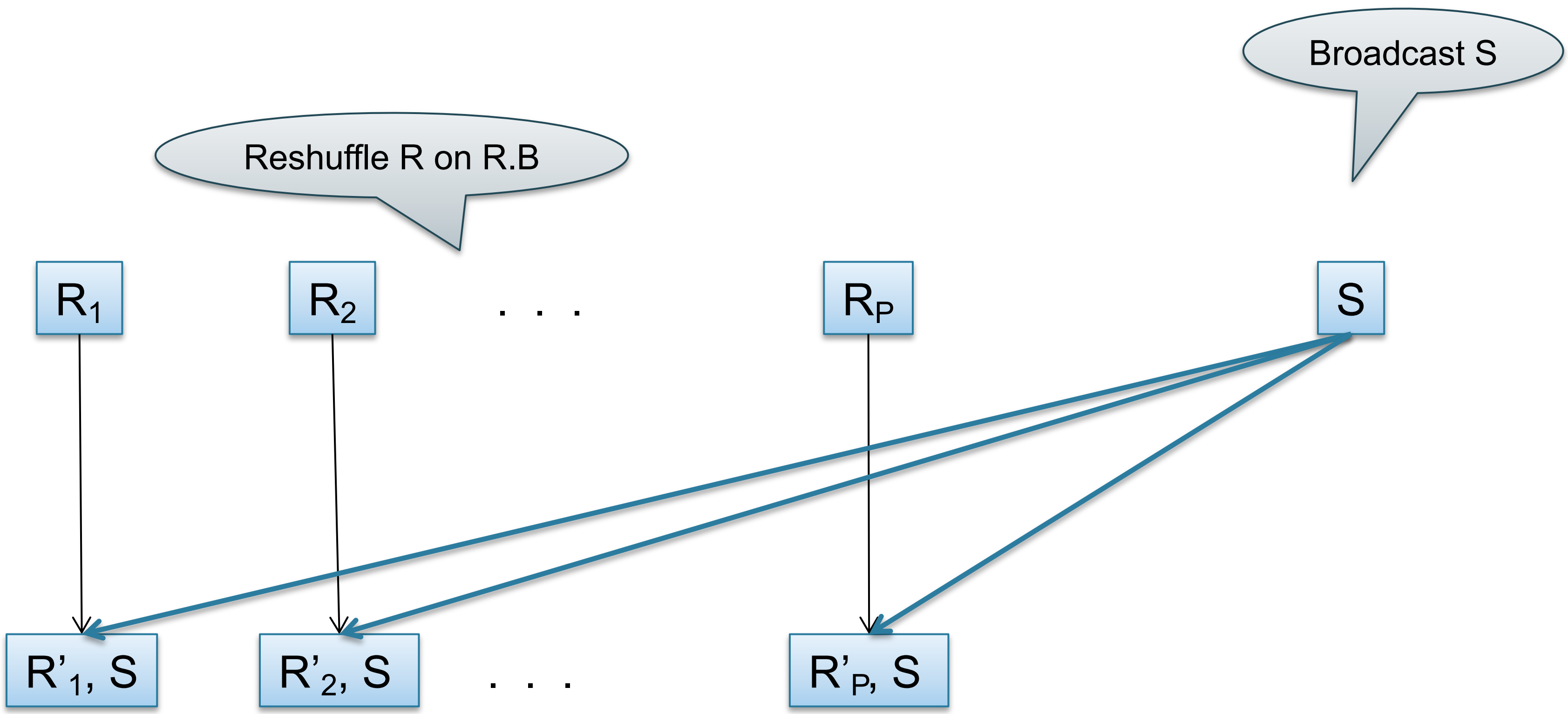
S2'

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

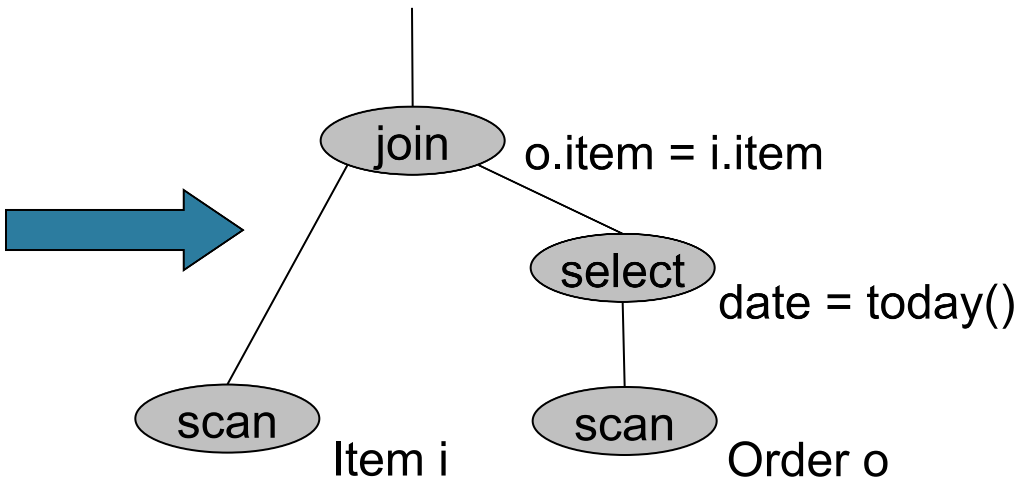


Why would you want to do this?

Putting it Together: Example Parallel Query Plan

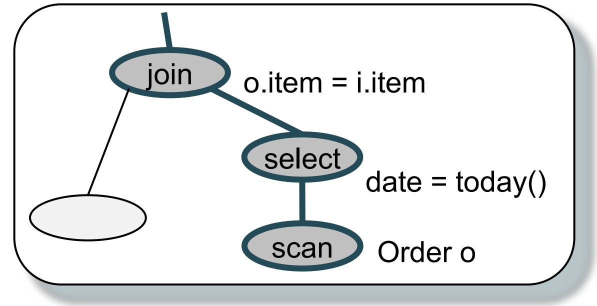
Find all orders from today, along with the items ordered

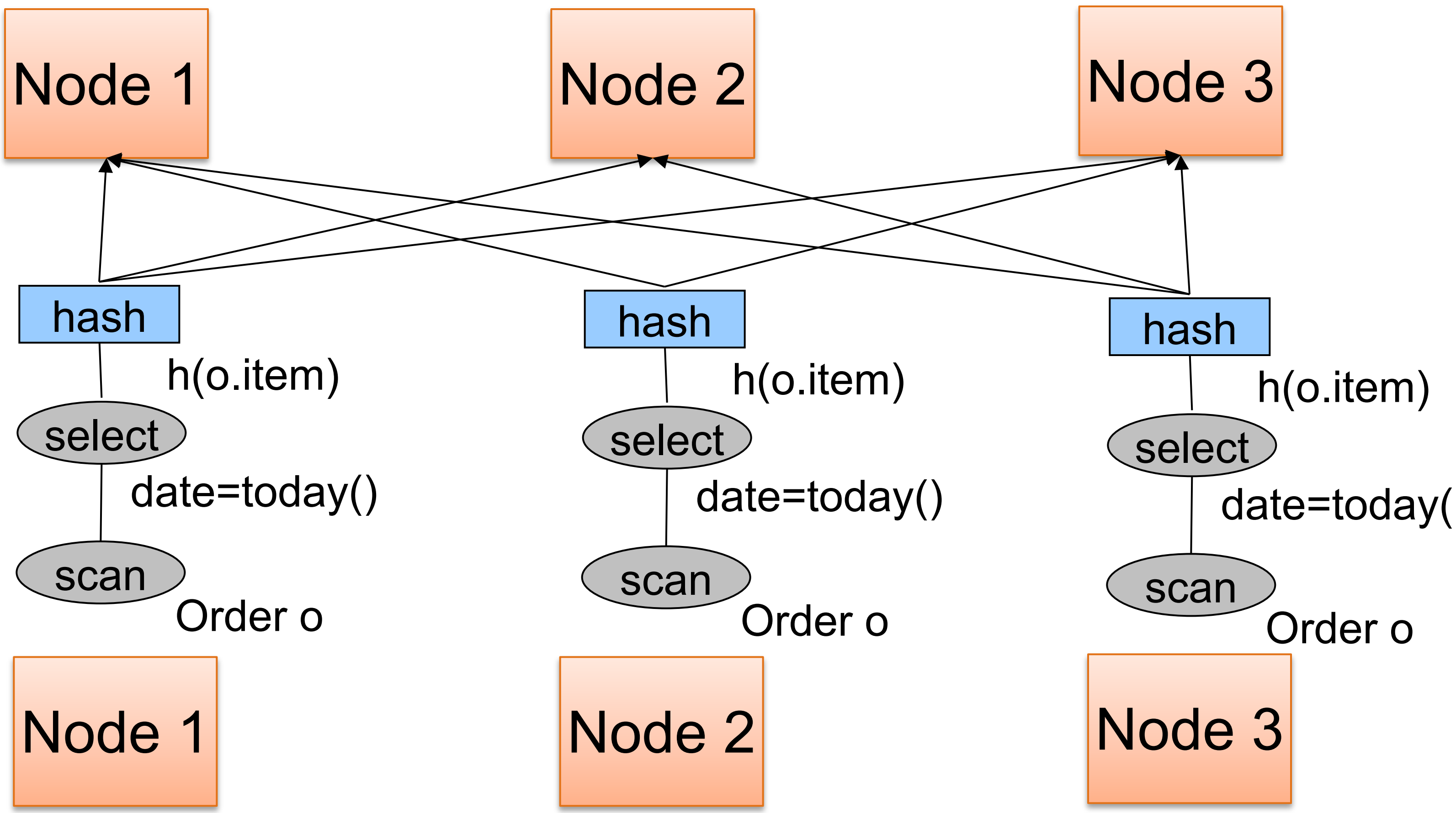
```
SELECT *
  FROM Order o, Line i
 WHERE o.item = i.item
   AND o.date = today()
```



Order(oid, item, date), Line(item, …)

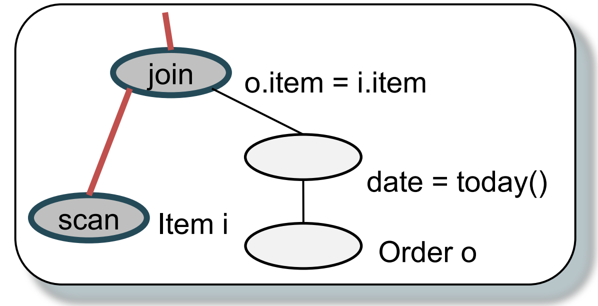
Example Parallel Query Plan

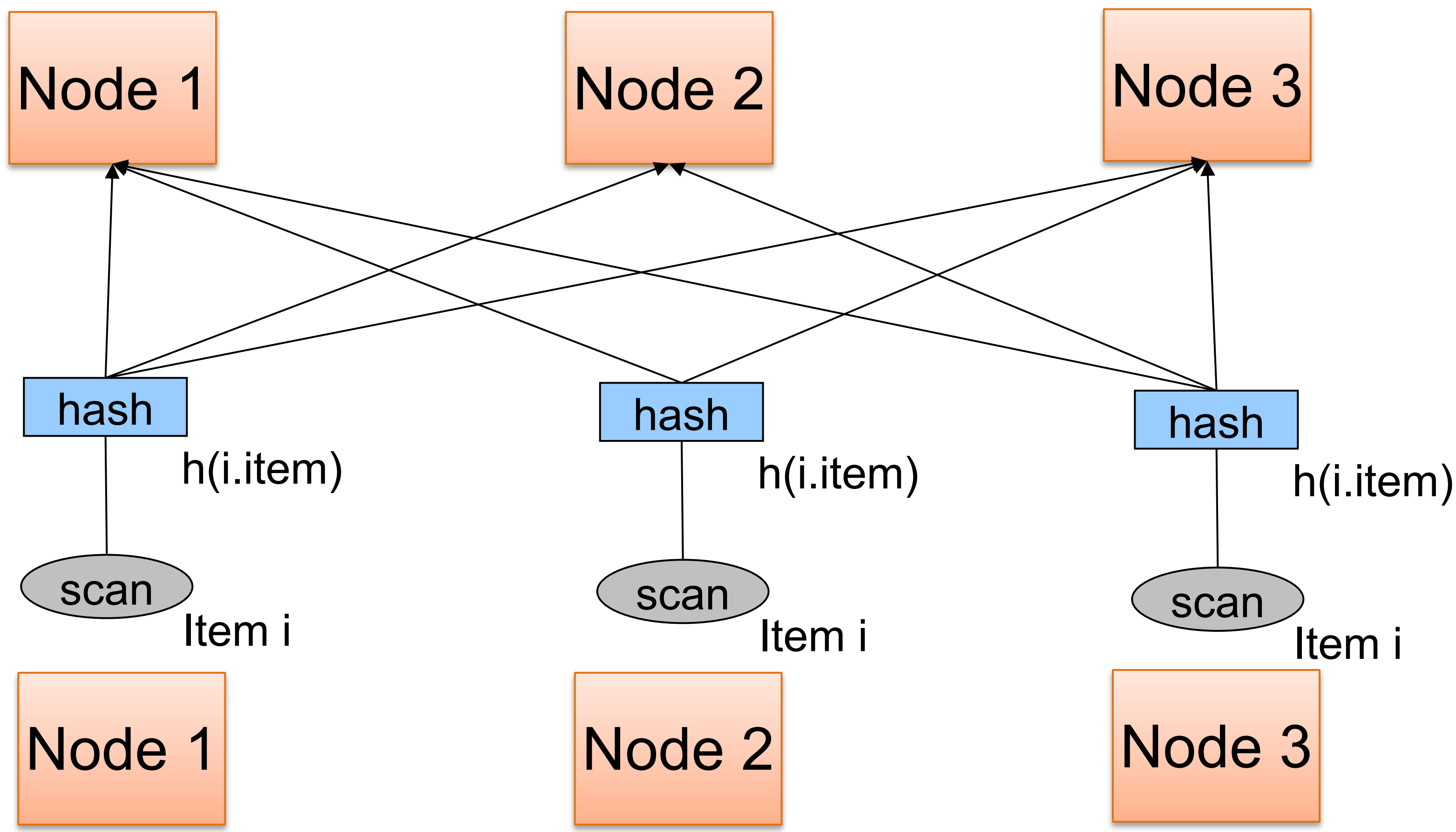




Order(oid, item, date), Line(item, …)

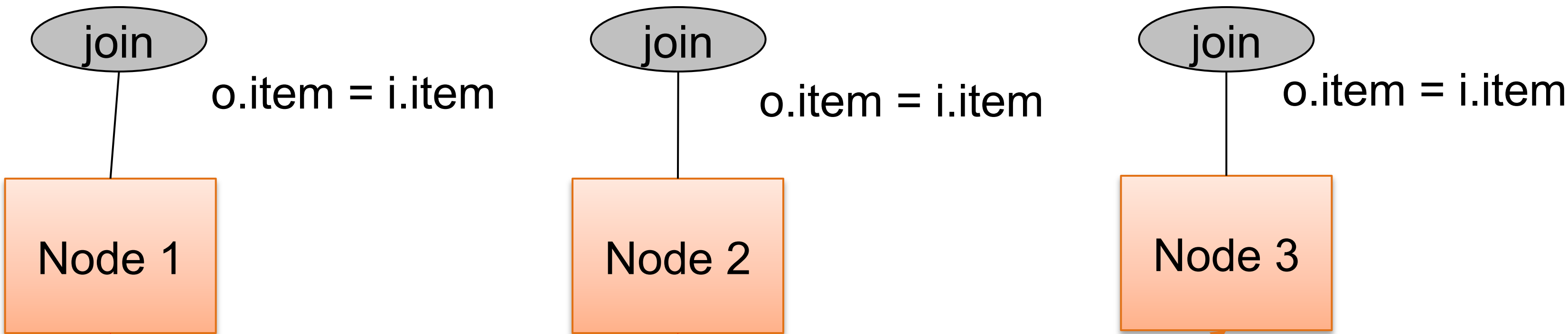
Example Parallel Query Plan





Order(oid, item, date), Line(item, …)

Example Parallel Query Plan



contains all orders and all
lines where hash(item) = 3

contains all orders and all
lines where hash(item) = 2

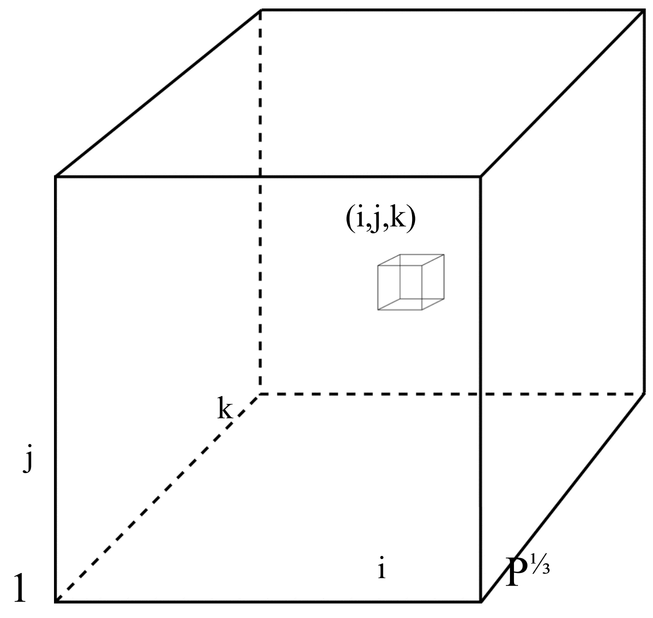
contains all orders and all
lines where hash(item) = 1

A Challenge

- Have P number of servers (say P=27 or P=1000)
- How do we compute this Datalog query in one step?
- Q(x,y,z) :- R(x,y), S(y,z), T(z,x)

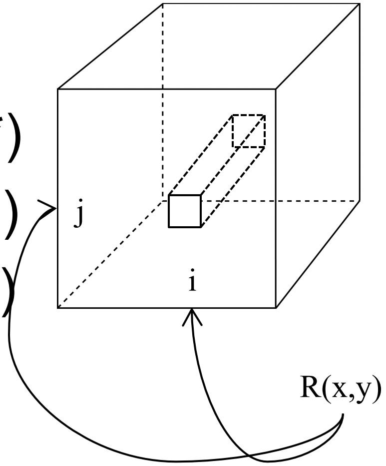
A Challenge

- Have P number of servers (say P=27 or P=1000)
- How do we compute this Datalog query in one step?
  Q(x,y,z) = R(x,y),S(y,z),T(z,x)
- Organize the P servers into a cube with side $P^{⅓}$
  - Thus, each server is uniquely identified by (i,j,k), i,j,k≤$P^{⅓}$



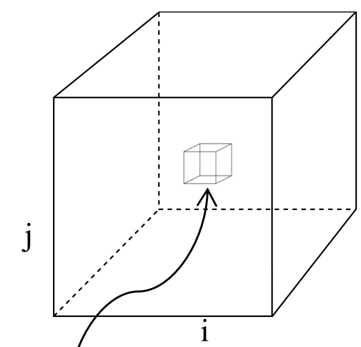
HyperCube Join

- Have P number of servers (say P=27 or P=1000)
- How do we compute this Datalog query in one step?
  Q(x,y,z) = R(x,y),S(y,z),T(z,x)
- Organize the P servers into a cube with side $P^{⅓}$
  – Thus, each server is uniquely identified by (i,j,k), $i,j,k \leq P^{⅓}$
- Step 1:
  – Each server sends R(x,y) to all servers (h(x),h(y),*)
  – Each server sends S(y,z) to all servers (*,h(y),h(z))
  – Each server sends T(x,z) to all servers (h(x),*,h(z))



HyperCube Join

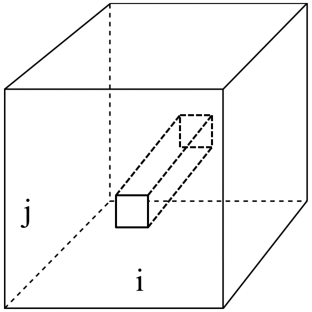
- Have P number of servers (say P=27 or P=1000)
- How do we compute this Datalog query in one step?
  Q(x,y,z) = R(x,y),S(y,z),T(z,x)
- Organize the P servers into a cube with side $P^{⅓}$
  - Thus, each server is uniquely identified by (i,j,k), $i,j,k \leq P^{⅓}$
- Step 1:
  - Each server sends R(x,y) to all servers (h(x),h(y),*)
  - Each server sends S(y,z) to all servers (*,h(y),h(z))
  - Each server sends T(x,z) to all servers (h(x),*,h(z))



- Final output:
  - Each server (i,j,k) computes the query R(x,y),S(y,z),T(z,x) locally

HyperCube Join

- Have P number of servers (say P=27 or P=1000)
- How do we compute this Datalog query in one step?
  Q(x,y,z) = R(x,y),S(y,z),T(z,x)
- Organize the P servers into a cube with side $P^{1/3}$
  - Thus, each server is uniquely identified by (i,j,k), i,j,k≤$P^{1/3}$
- Step 1:
  - Each server sends R(x,y) to all servers (h(x),h(y),*)
  - Each server sends S(y,z) to all servers (*,h(y),h(z))
  - Each server sends T(x,z) to all servers (h(x),*,h(z))



- Final output:
  - Each server (i,j,k) computes the query R(x,y),S(y,z),T(z,x) locally
- Analysis: each tuple R(x,y) is replicated at most $P^{1/3}$ times

$Q(x,y,z) = R(x,y),S(y,z),T(z,x)$

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

What if

h(x): h(1) = h(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

What if

h(x): h(1) = h(3)

Local Join

R1’

x	y
1	2
3	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
1	2

S3’

y	z
2	3

T3

z	x
3	3

P3: (3, 2, 3)

Introduction to Data Management
CSE 344

MapReduce

Announcements

- HW6 due tomorrow
- HW7 to be posted tomorrow; HW8 next week
- Makeup lectures:
  - Tuesday, Nov. 27, Room EEB 105
  - Tuesday, Dec. 4, Room EEB 105
- Canceled lectures:
  - Wednesday, Dec. 5
  - Friday, Dec. 7

Check the calendar!

Parallel Data Processing @ 2000

Optional Reading

- Original paper: https://www.usenix.org/legacy/events/osdi04/tech/dean.html
- Rebuttal to a comparison with parallel DBs: http://dl.acm.org/citation.cfm?doid=1629175.1629198
- Chapter 2 (Sections 1,2,3 only) of Mining of Massive Datasets, by Rajaraman and Ullman http://i.stanford.edu/~ullman/mmds.html

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 ($\geq$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

slide source: Jeff Dean

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)`

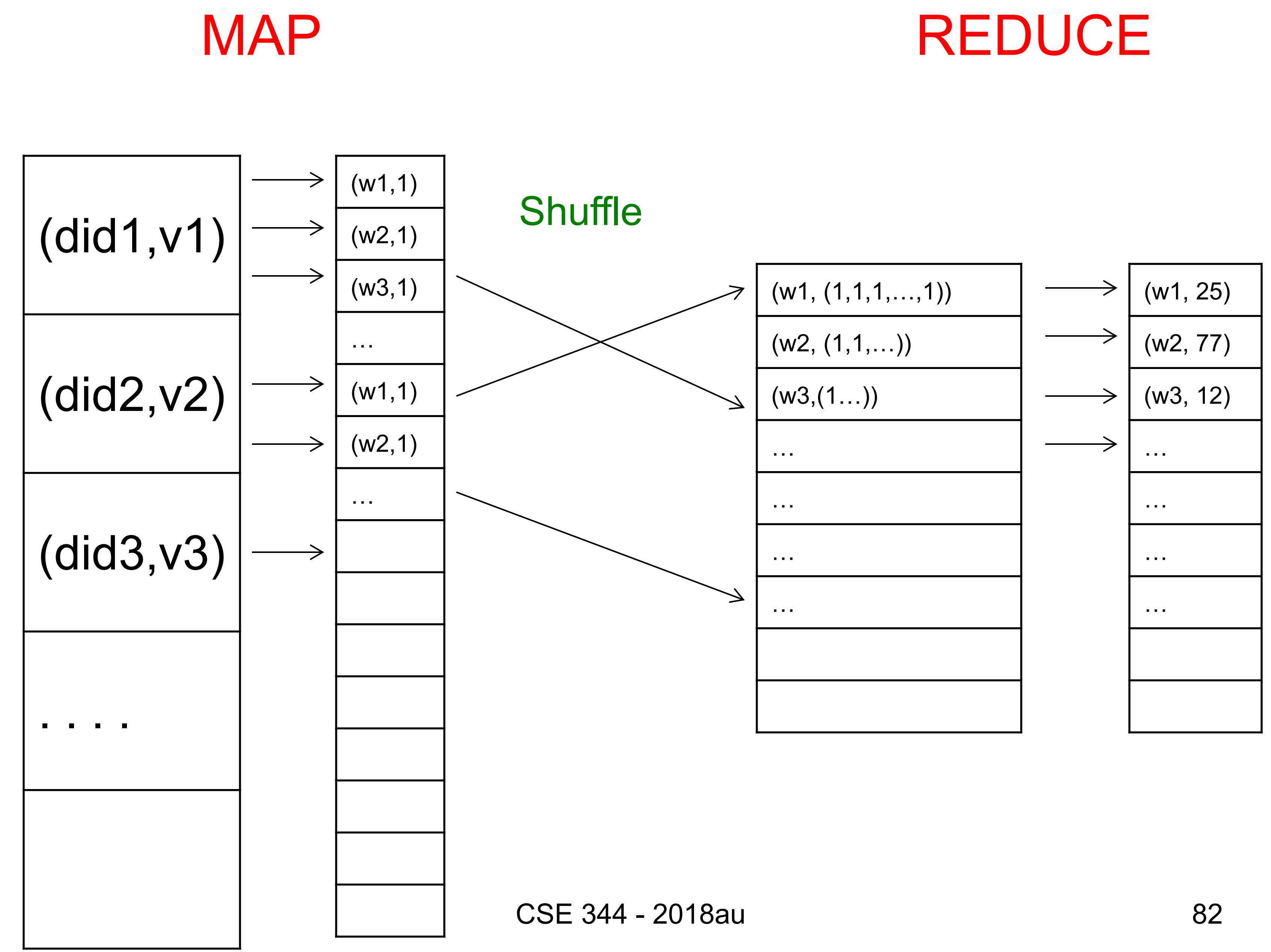
System groups all pairs with the same intermediate key, and passes the bag of values to the REDUCE function

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)
…
…
…
…

Comparison to Spark

MapReduce in Spark:

- `col.flatMap(f)` applies in parallel the predicate f to all elements x of the partitioned collection; for each x, f(x) is a list; then flatMap returns their concatenation

- `col.reduceByKey(g)` applies in parallel the function g to all elements with a common key

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

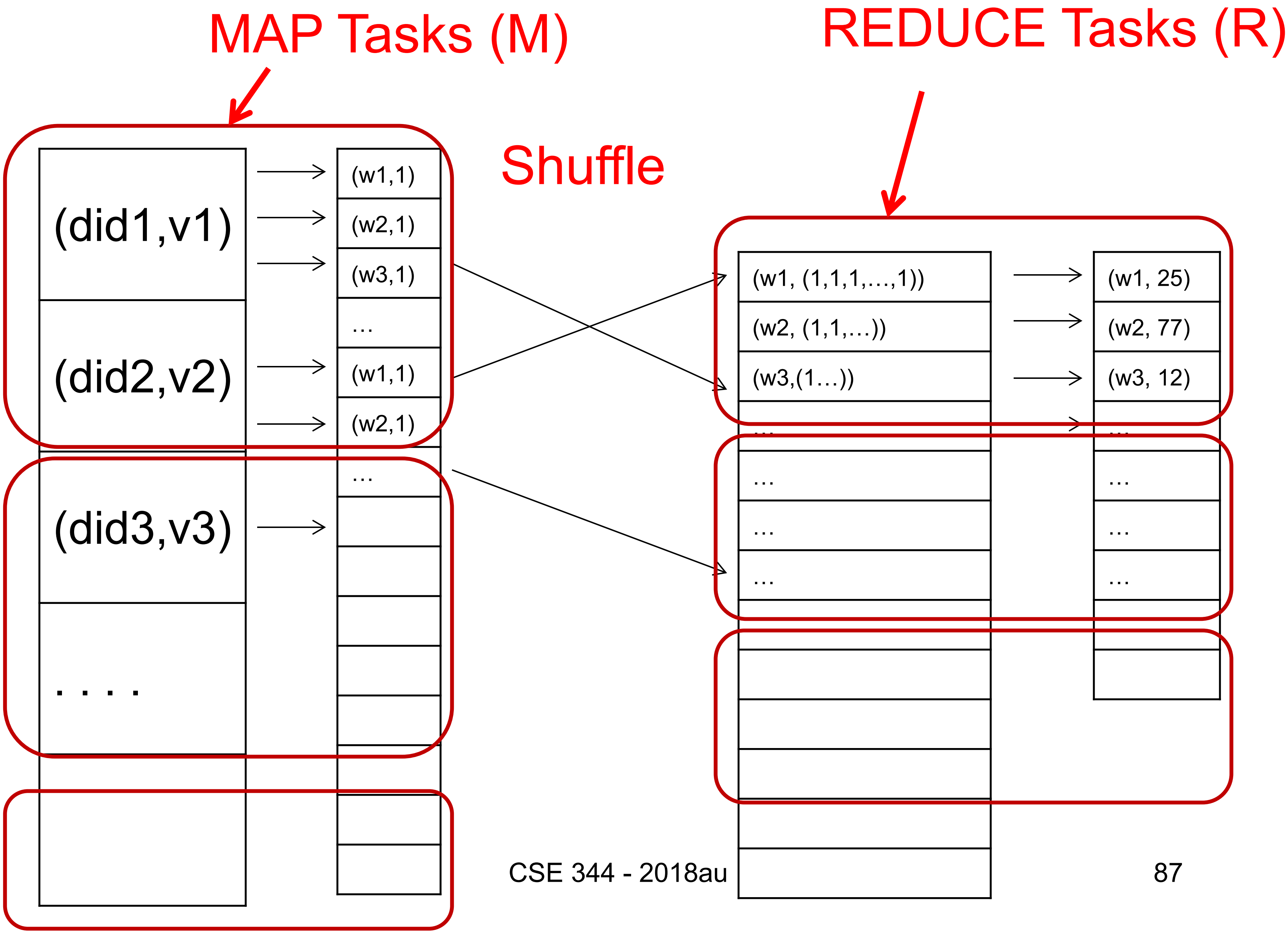
Workers

- 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

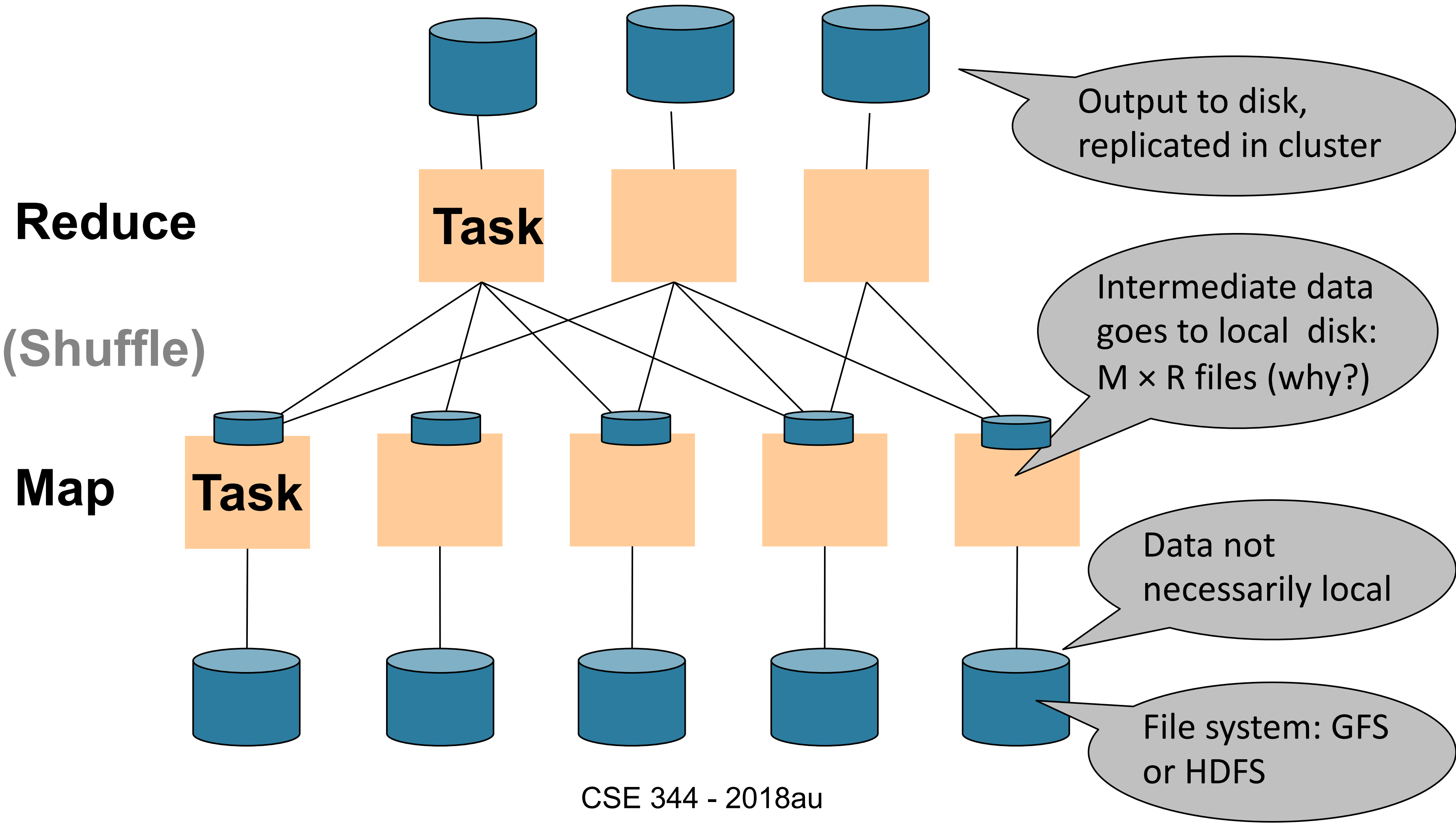
Fault Tolerance

MapReduce handles fault tolerance by writing intermediate files to disk:

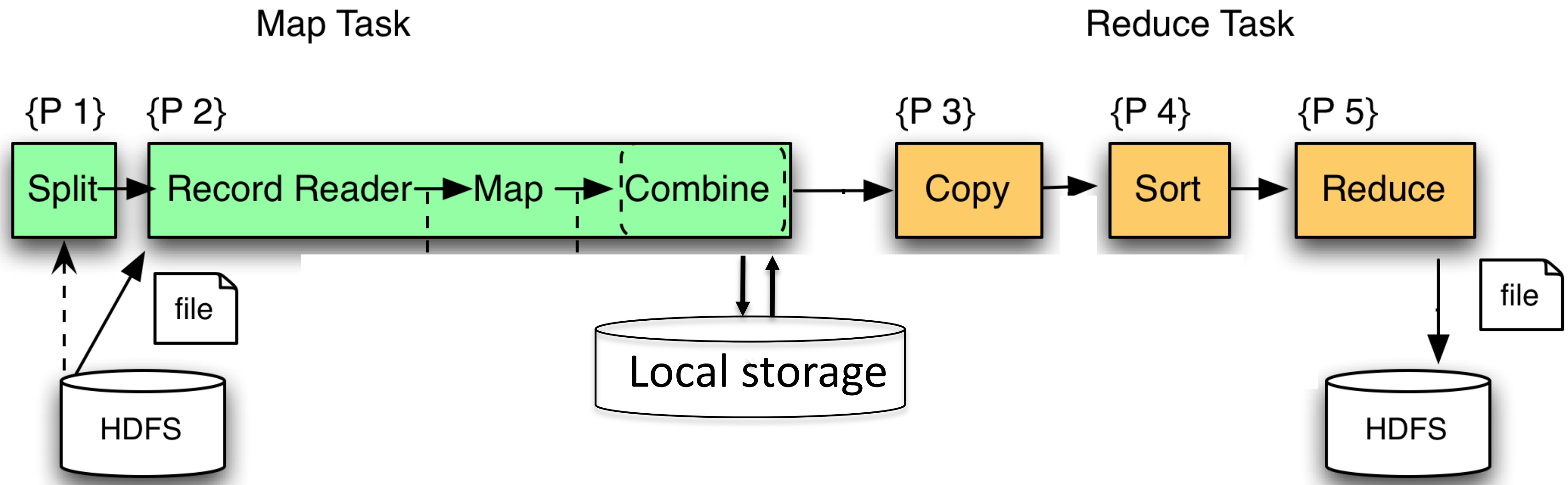
- Mappers write file to local disk
- Reducers read the files (=reshuffling); if the server fails, the reduce task is restarted on another server


MAP Tasks (M)
REDUCE Tasks (R)
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)
…
…
…

MapReduce Execution Details



MapReduce Phases



Implementation

- There is one master node
- Master partitions input file into *M splits*, by key
- Master assigns *workers* (=servers) to the *M map tasks*, keeps track of their progress
- Workers write their output to local disk, partition into *R regions*
- Master assigns workers to the *R reduce tasks*
- Reduce workers read regions from the map workers' local disks

Interesting Implementation Details

Worker failure:

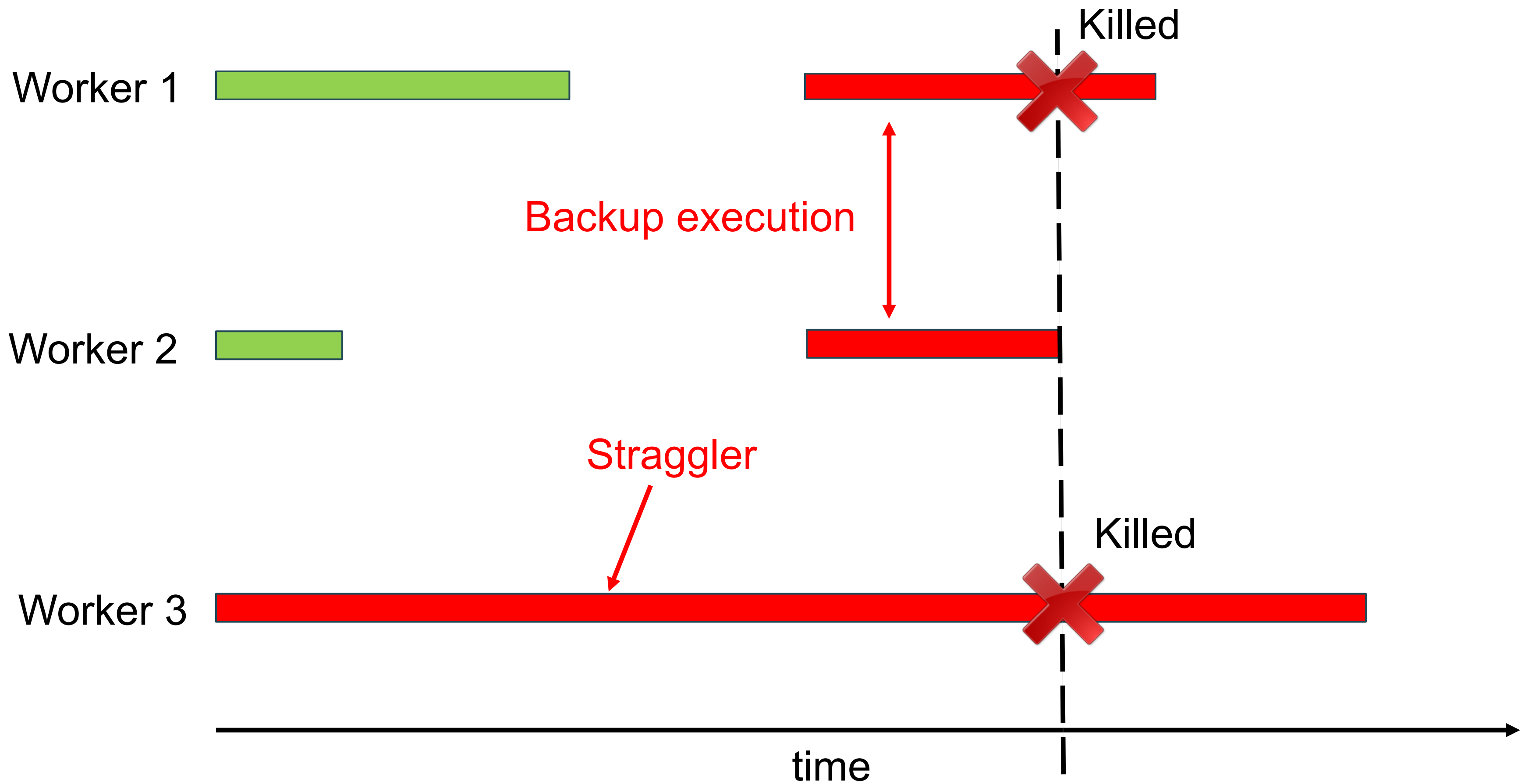
- Master pings workers periodically,
- If down then reassigns the task to another worker

Interesting Implementation Details

Backup tasks:

- *Straggler* = a machine that takes unusually long time to complete one of the last tasks. E.g.:
  - Bad disk forces frequent correctable errors (30MB/s → 1MB/s)
  - The cluster scheduler has scheduled other tasks on that machine
- Stragglers are a main reason for slowdown
- Solution*: pre-emptive backup execution of the last few remaining in-progress tasks*

Straggler Example



Using MapReduce in Practice:

Implementing RA Operators in MR

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

$(123, [\ t_2, t_3\ ]\ )$

	A
t_1	23
t_2	123
t_3	123
t_4	42

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

$(\ t_2, t_3\ )$

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

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

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

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

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

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

$(23, [\ t_1\ ])$

$(42, [\ t_4\ ])$

$(123, [\ t_2, t_3\ ])$

	A	B
t_1	23	10
t_2	123	21
t_3	123	4
t_4	42	6

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

(23, 10), (42, 6), (123, 25)

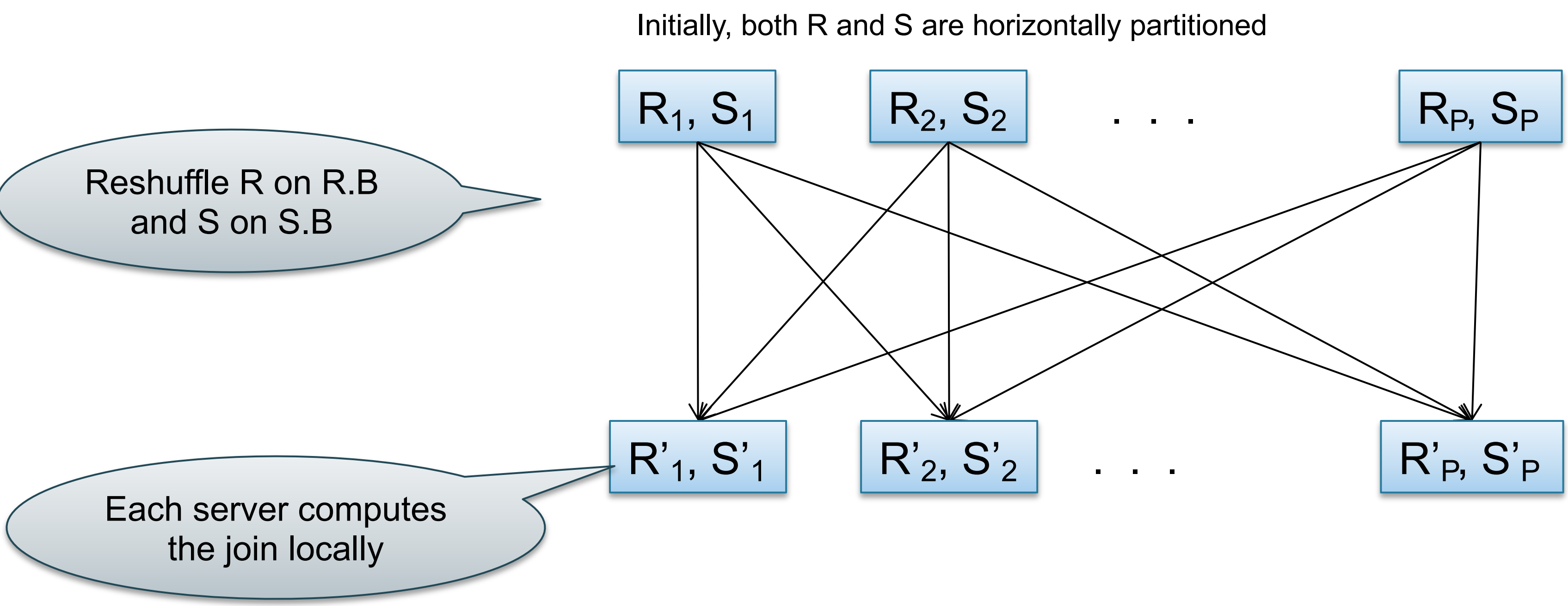
Join

Two simple parallel join algorithms:

- Partitioned hash-join (we saw it, will recap)
- Broadcast join

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

Partitioned Hash-Join



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

Partitioned Hash-Join

actual tuple

```
map(Tuple t):
    case t.relationName of
        'R': EmitIntermediate(t.B, t);
        'S': EmitIntermediate(t.C, t);
```

(5, [t_1, t_3, t_4])
(6, [t_2, t_5])

```
reduce(String k, Iterator values):
    R = empty;  S = empty;
    for each v in values:
        case v.relationName of:
            'R':   R.insert(v)
            'S':   S.insert(v);
    for v1 in R, for v2 in S
        Emit(v1,v2);
```

(t_1, t_4), (t_3, t_4)
(t_2, t_5)

R

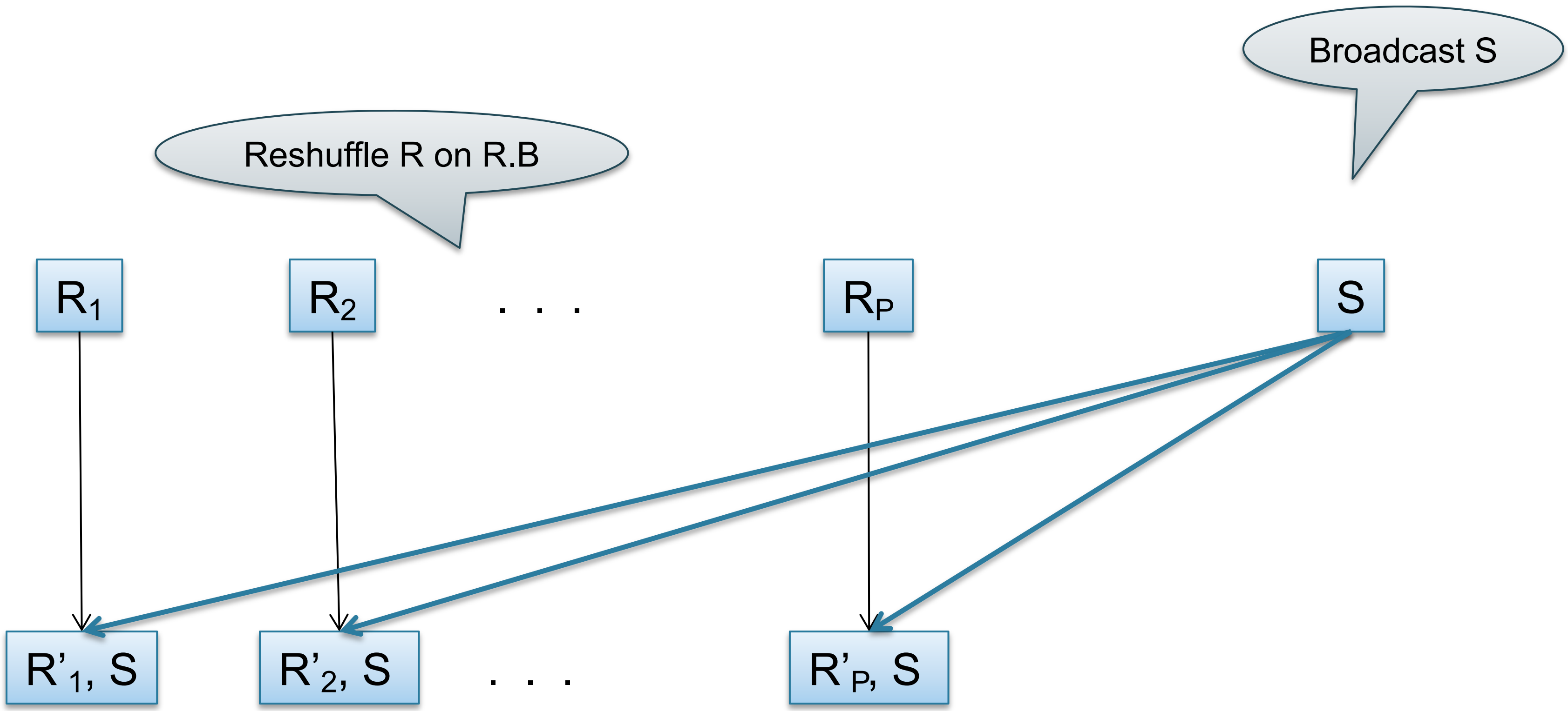
	A	B
t_1	10	5
t_2	20	6
t_3	20	5

S

	C	D
t_4	5	2
t_5	6	10

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

Broadcast Join



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

Broadcast Join

```
map(String value):
    open(S); /* over the network */
    hashTbl = new()
    for each w in S:
        hashTbl.insert(w.C, w)
    close(S);

    for each v in value:
        for each w in hashTbl.find(v.B)
            Emit(v,w);
```

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

Read entire table S, build a Hash Table

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

Summary

- MapReduce: simple abstraction, distribution, fault tolerance
- Load balancing (needed for speedup): by dynamic task allocation; skew still possible
- Writing intermediate results to disk is necessary for fault tolerance, but very slow.
- Spark replaces this with “Resilient Distributed Datasets” = main memory + lineage