DATA516/CSED516 Scalable Data Systems and Algorithms

Lecture 4

Spark, MapReduce, Hive

Intro to Parallel Processing

Announcements

- Project proposals due this Friday!
  - Working in team? Only one of you submits

- HW2 (Spark) due on Monday

- Reminder: Jack has no OH this Thursday

Outline

- Spark Review

- MapReduce and critique

- Fault Tolerance

- Hive (short)

Next lecture: Parallel databases (Start Today)

Spark

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

Example

The RDD s:

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

```
sqlerrors = spark.textFile("hdfs://…")
            .filter(x -> x.startsWith("ERROR"))
            .filter(x -> x.contains("sqlite"))
            .collect();
```

Example

Parallel step 1

The RDD s:

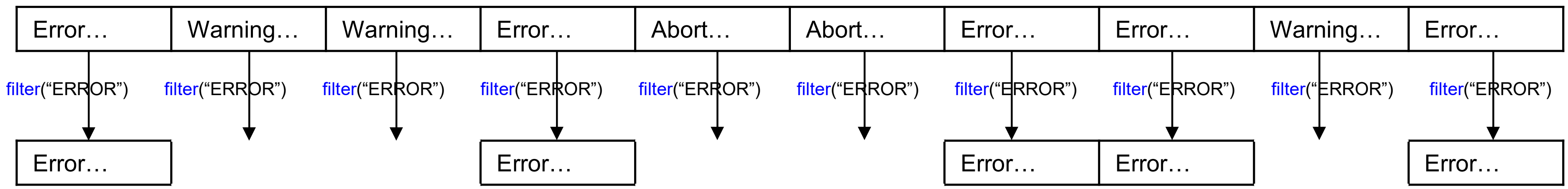
Error…	Warning…	Warning…	Error…	Abort…	Abort…	Error…	Error…	Warning…	Error…
filter(“ERROR”)	filter(“ERROR”)	filter(“ERROR”)	filter(“ERROR”)	filter(“ERROR”)	filter(“ERROR”)	filter(“ERROR”)	filter(“ERROR”)	filter(“ERROR”)	filter(“ERROR”)

```
sqlerrors = spark.textFile(“hdfs://…”)
              .filter(x -> x.startsWith(“ERROR”))
              .filter(x -> x.contains(“sqlite”))
              .collect();
```

Example

Parallel step 1

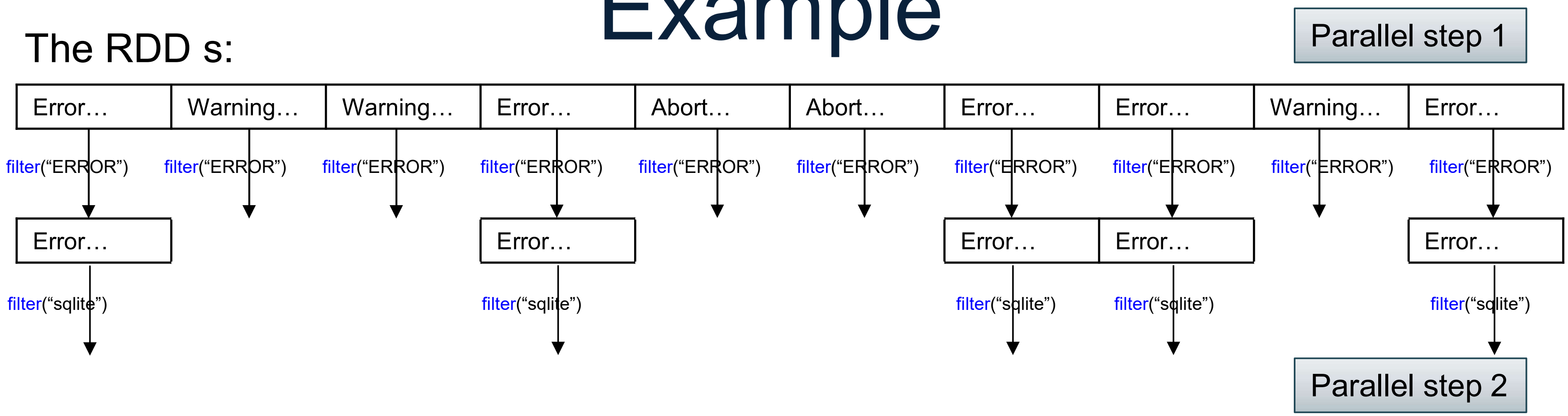
The RDD s:




```
sqlerrors = spark.textFile("hdfs://...")
            .filter(x -> x.startsWith("ERROR"))
            .filter(x -> x.contains("sqlite"))
            .collect();
```

Example

The RDD s:




```
sqlerrors = spark.textFile("hdfs://…")
            .filter(x -> x.startsWith("ERROR"))
            .filter(x -> x.contains("sqlite"))
            .collect();
```

What Am I?

```
val points = spark.textFile(...)
                  .map(parsePoint).persist()
var w = // random initial vector
for (i <- 1 to ITERATIONS) {
  val gradient = points.map{ p =>
    p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
  }.reduce((a,b) => a+b)
  w -= gradient
}
```

[From Zaharia12]

What Am I?

```
val points = spark.textFile(...)
                  .map(parsePoint).persist()
var w = // random initial vector
for (i <- 1 to ITERATIONS) {
  val gradient = points.map{ p =>
    p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
  }.reduce((a,b) => a+b)
  w -= gradient
}
```

Logistic Regression!!

[From Zaharia12]

Spark Ecosystem Growth

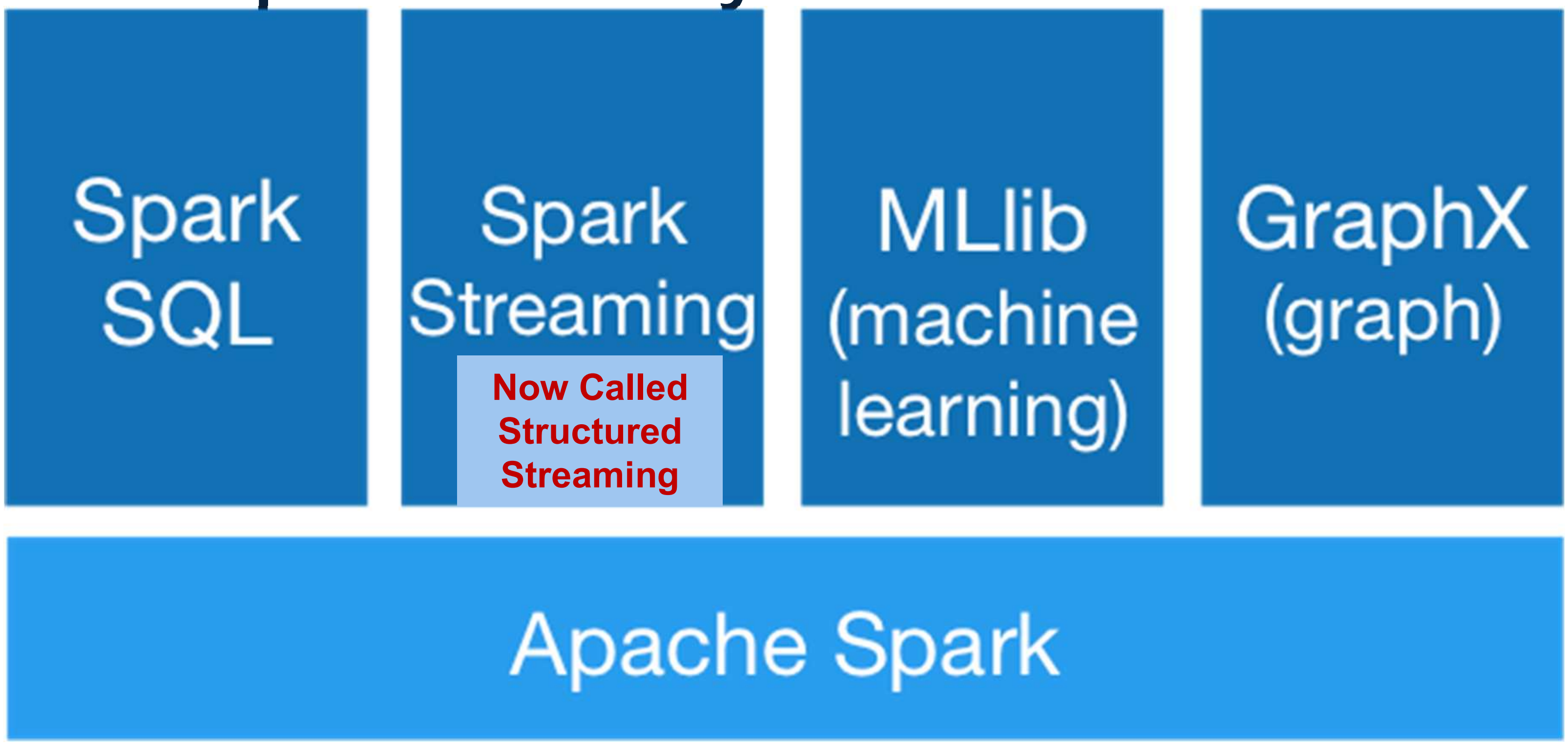


Image from: http://spark.apache.org/

Spark SQL vs Functional Prog. API

- Spark's original functional programming API
  - General
  - But limited opportunities for automatic optimization
- Spark SQL simultaneously
  - Makes Spark accessible to more users
  - Improves opportunities for automatic optimizations

Three Java-Spark APIs

- RDDs: Syntax: JavaRDD<T>
  - T = anything, basically untyped
- Data frames: Dataset<Row>
  - <Row> = a record, dynamically typed
- Datasets: Dataset<Person>
  - <Person> = user defined type
  - Not in Python/R

DataFrames

- Like RDD: immutable distributed collection
- Organized into *named columns*
  - 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

- Like DataFrames, but elements must be typed
- 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)

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?

Outline

- Spark
- MapReduce and critique
- Fault Tolerance
- Hive (short)

MapReduce: References

- Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters. OSDI'04

- D. DeWitt and M. Stonebraker. Mapreduce – a major step backward. In Database Column (Blog), 2008.

MapReduce

- Google:
  - Started around 2000
  - Paper published 2004
  - Discontinued September 2019
- Free variant: Hadoop

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

Distributed File System (DFS)

- For very large files: TBs, PBs
- Each file partitioned into *chunks* (64MB)
- Each chunk replicated (≥3 times) – why?
- Implementations:
  - Google's DFS: GFS, proprietary
  - Hadoop's DFS: HDFS, open source

MapReduce

- Describe the **input** and **output** to map reduce

- Describe the **Map** function

- Describe the **Reduce** function

MapReduce

- Describe the **input** and **output** to map reduce
  - Input: a bag of `(inputkey, value)` pairs
  - Output: a bag of `(outputkey, value)` pairs
- Describe the **Map** function

- Describe the **Reduce** function

MapReduce

- Describe the **input** and **output** to map reduce
  - Input: a bag of `(inputkey, value)` pairs
  - Output: a bag of `(outputkey, value)` pairs
- Describe the **Map** function
  - Input: `(input key, value)`
  - Ouput: bag of `(intermediate key, value)`
- Describe the **Reduce** function

MapReduce

- Describe the **input** and **output** to map reduce
  - Input: a bag of `(inputkey, value)` pairs
  - Output: a bag of `(outputkey, value)` pairs
- Describe the **Map** function
  - Input: `(input key, value)`
  - Ouput: bag of `(intermediate key, value)`
- Describe the **Reduce** function
  - Input: `(intermediate key, bag of values)`
  - Output: bag of output `(values)`

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 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)

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");
```

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));
```

Think "Relational"!

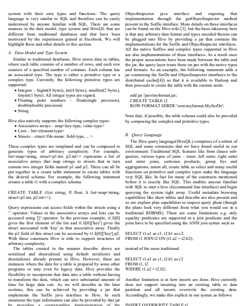
Documents:

Relation

did1


Hive – A Petabyte Scale Data Warehouse Using Hadoop

did2



. . .

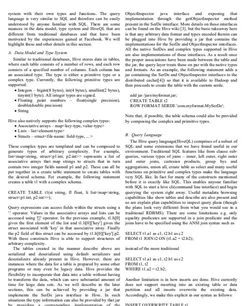
Think “Relational”!

Documents:

did1


Hive – A Petabyte Scale Data Warehouse Using Hadoop

did2



. . .

Relation

Did	Word
did1	Scalable
did1	analysis
did1	on
did1	large
did1	…
did2	system
did2	with
…	

Think “Relational”!

select	word, count(*)
from	Data
group by	word

Relation

Did	Word
did1	Scalable
did1	analysis
did1	on
did1	large
did1	…
did2	system
did2	with
…	

Think “Relational”!

select	word, count(*)
from	Data
group by	word

Relation

Did	Word
did1	Scalable
did1	analysis
did1	on
did1	large
did1	…
did2	system
did2	with
…	

map = group by

reduce = count(…) (or sum(…) or…)

Think “Relational”!

```
select      word, count(*)
from        Data
group by    word
```

Relation

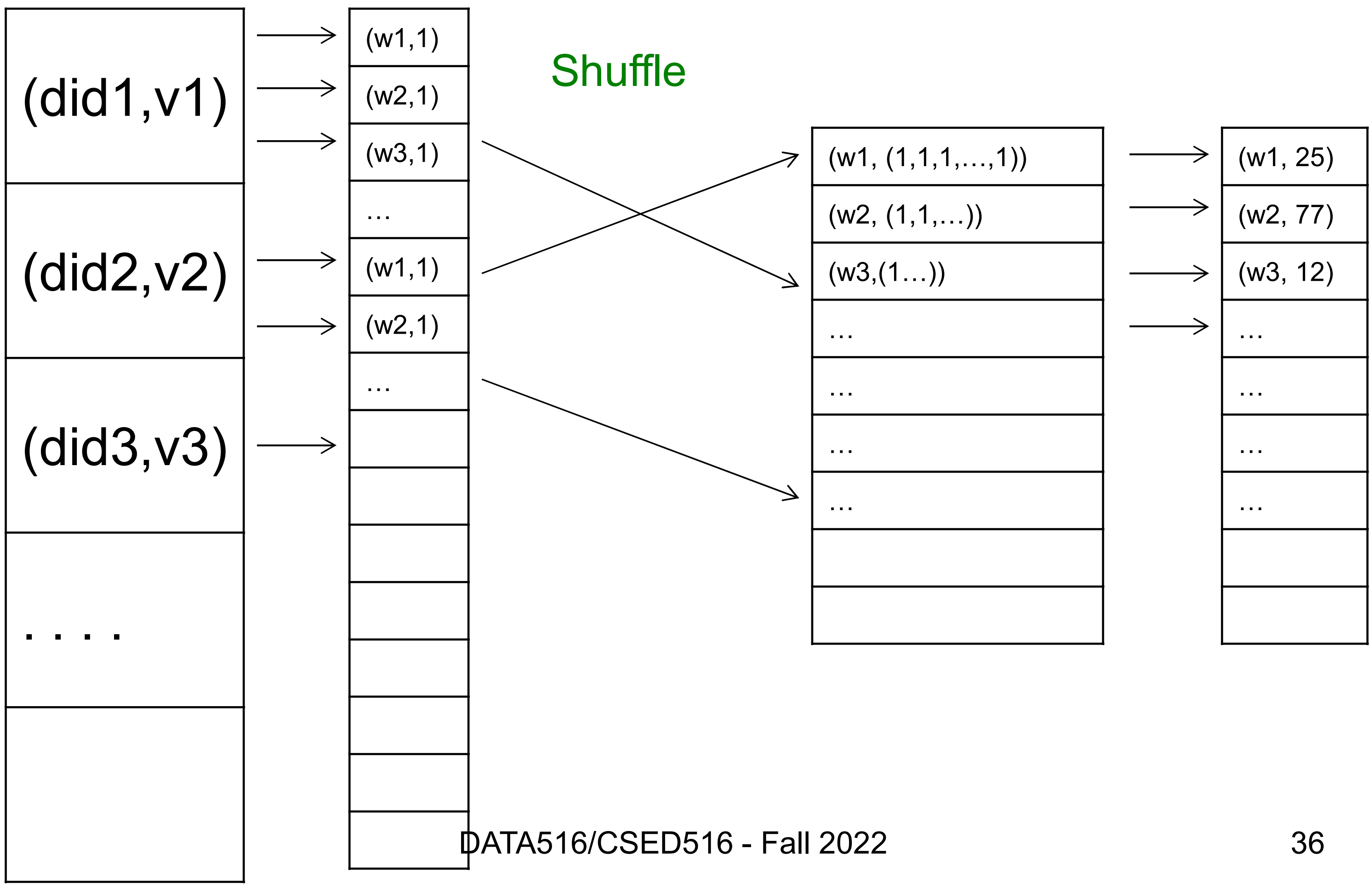
Did	Word
did1	Scalable
did1	analysis
did1	on
did1	large
did1	…
did2	system
did2	with
…	

map = group by

reduce = count(…) (or sum(…) or…)

MapReduce = Group-by-aggregate

MAP REDUCE


(did1,v1)
(did2,v2)
(did3,v3)
. . . .
(w1,1)
(w2,1)
(w3,1)
…
(w1,1)
(w2,1)
…
Shuffle
(w1, (1,1,1,…,1))
(w2, (1,1,…))
(w3,(1…))
…
…
…
…
(w1, 25)
(w2, 77)
(w3, 12)
…
…
…
…

Examples from the paper

Discuss in class how to implement in MR

- Distributed grep
- Count URL access frequency: (URL, count)
- Reverse web-link graph: (URL, (list of URLs))
- Inverted index: (word, (list of URLs))

Jobs v.s. Tasks

- A MapReduce Job
  - One simple "query", e.g. count words in docs
  - Complex queries may require many jobs
- A Map Task, or a Reduce Task
  - A group of instantiations of the map-, or reduce-function, to be scheduled on a single worker

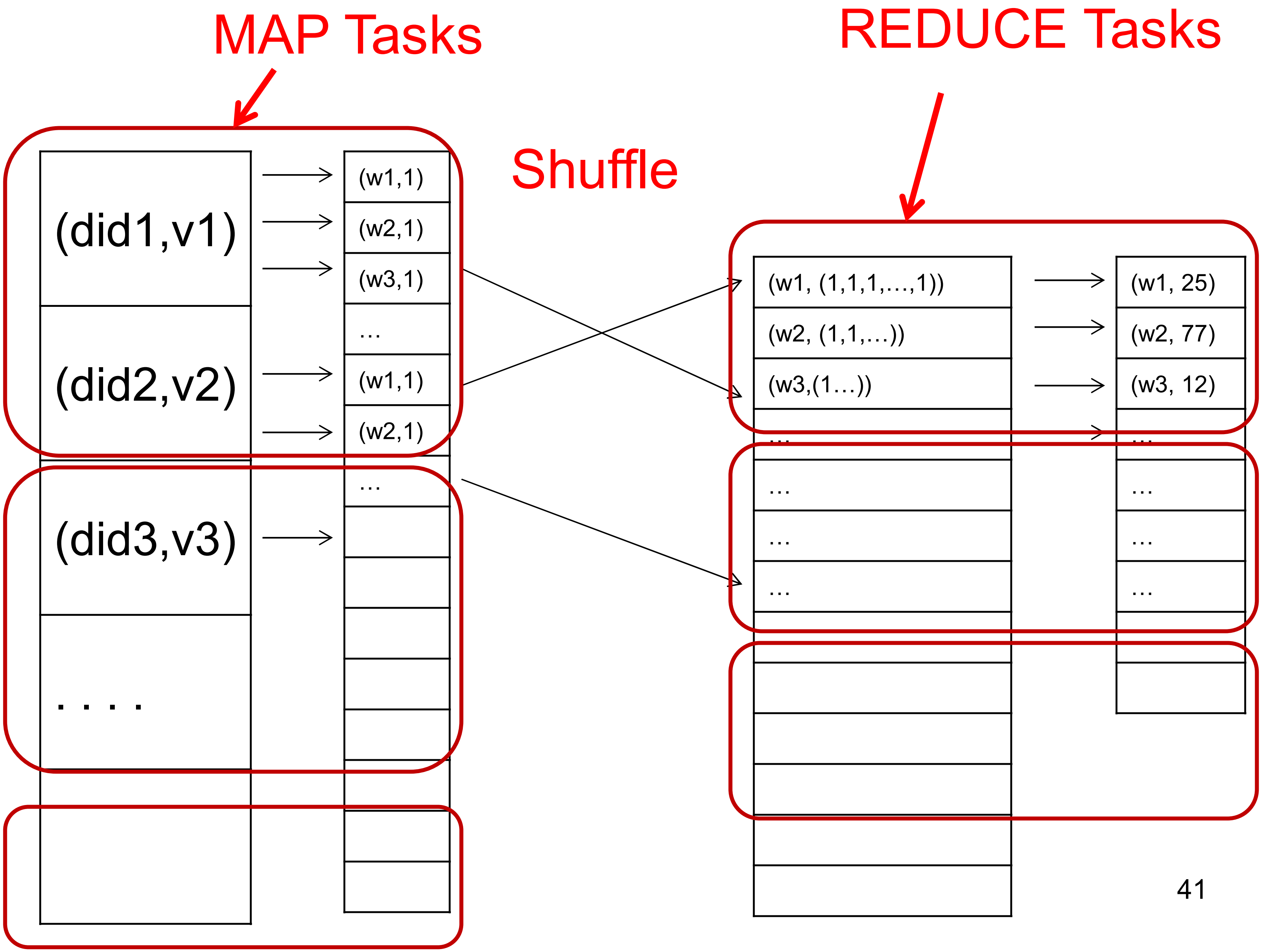
Workers

- 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

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

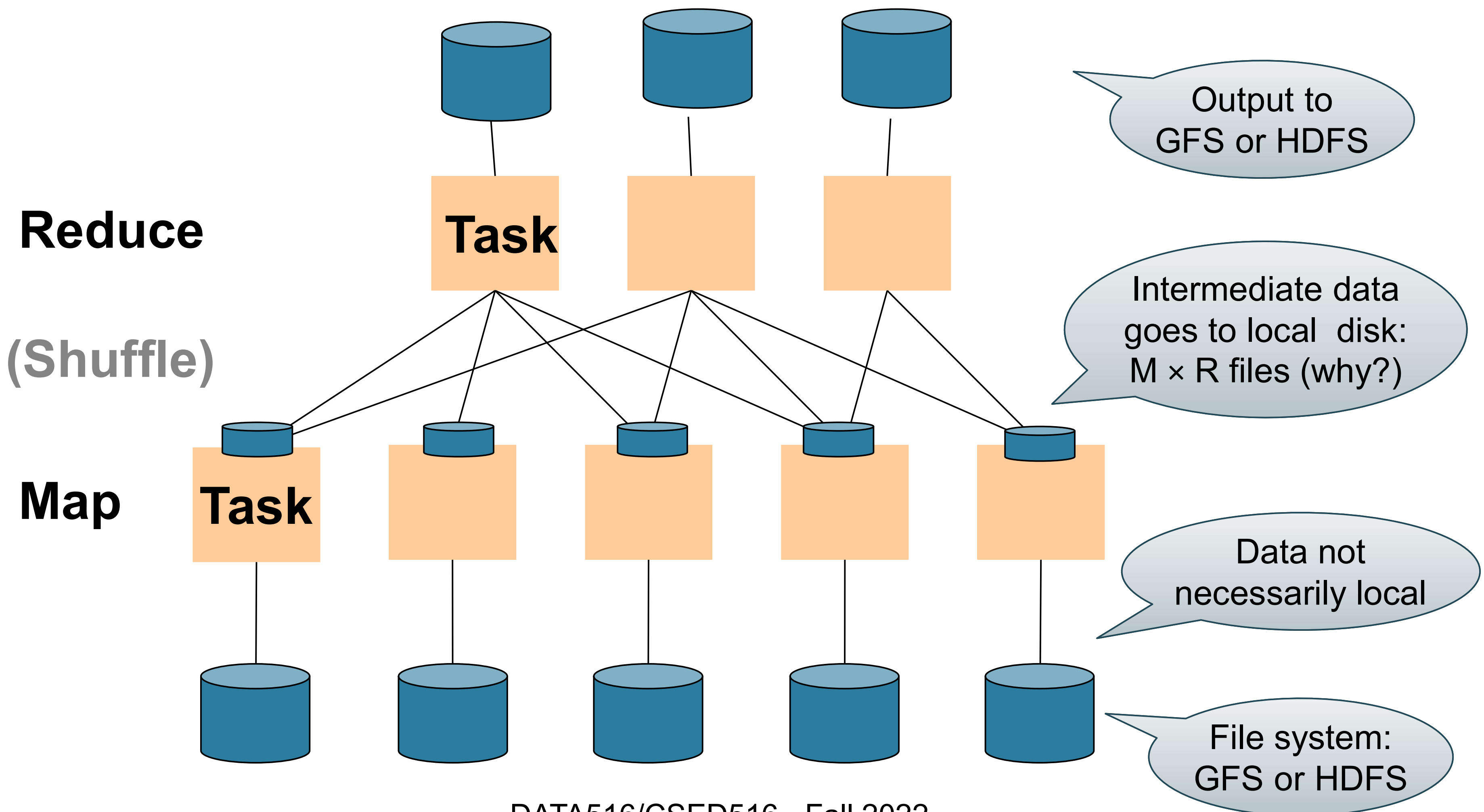

MAP Tasks
REDUCE Tasks
Shuffle
(did1,v1)
(did2,v2)
(did3,v3)
. . . .
(w1,1)
(w2,1)
(w3,1)
…
(w1,1)
(w2,1)
…
(w1, (1,1,1,…,1))
(w2, (1,1,…))
(w3,(1…))
…
…
…
…
(w1, 25)
(w2, 77)
(w3, 12)
…
…
…
…

Choosing Parameters in MR

- Number of map tasks (M):
  - Default: one map task per chunk
  - E.g. data = 64TB, chunk = 64MB ➔ M = 10^6
- Number of reduce tasks (R):
  - No good default; set manually R << M
  - E.g. R = 500 or 5000

- In general, MapReduce had very many parameters that required expertise to tune

MapReduce Execution Details



Discussion

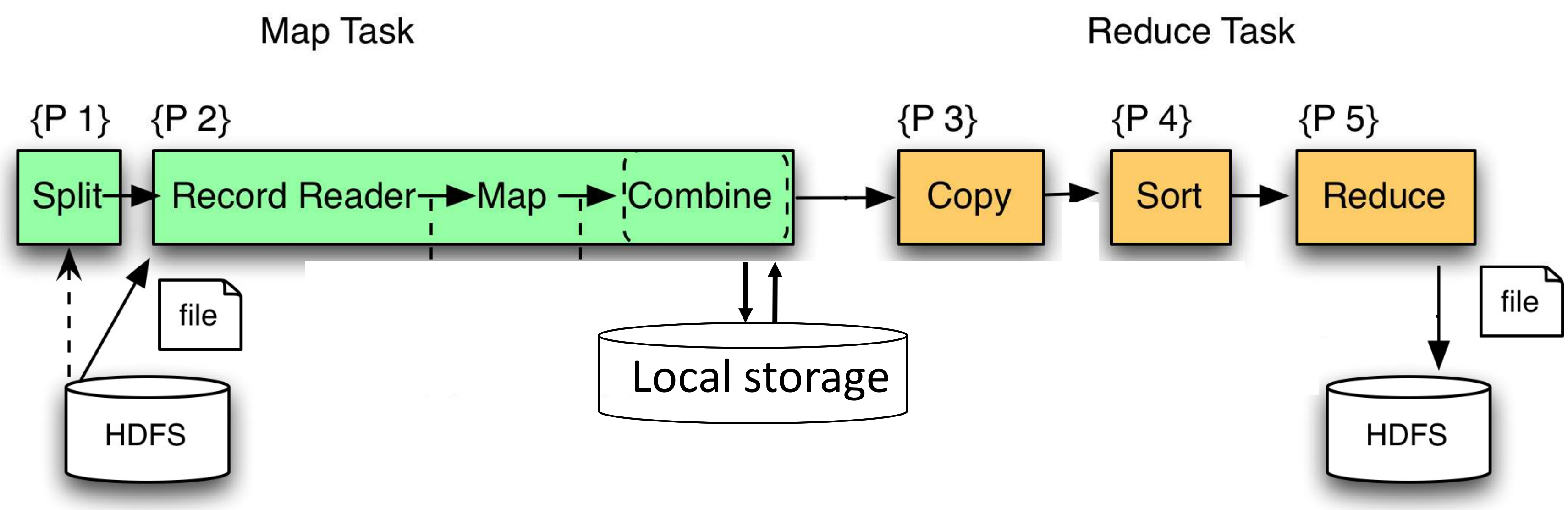
Why doesn't MR determine the number of reduce tasks R dynamically, after all map tasks finish?

Discussion

Why doesn’t MR determine the number of reduce tasks R dynamically, after all map tasks finish?

Because each map tasks needs to write its output into R file; so R must be known before the map tasks start

MapReduce Phases



Riddle

- The combiner function performs an optimization that you already know
- Which one?

Riddle

- The combiner function performs an optimization that you already know
- Which one?

- Pushing aggregates down

Riddle

- The combiner function performs an optimization that you already know
- Which one?

```
Temp=
  select server, word, count(*) as c
  from Data
  group by server, word
```

- Pushing aggregates down:
  - Each mapper groups by word

Riddle

- The combiner function performs an optimization that you already know
- Which one?

```
Temp=
  select server, word, count(*) as c
  from Data
  group by server, word
```

- Pushing aggregates down:
  – Each mapper groups by word
  – Reducers perform final group-by

```
Output =
  select word, sum(c)
  from Temp
  group by word
```

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

MapReduce v.s. Databases

Blog by DeWitt and Stonebraker

MapReduce v.s. Databases

Blog by DeWitt and Stonebraker

- “Schemas are good”

MapReduce v.s. Databases

Blog by DeWitt and Stonebraker

- “Schemas are good”
- “Indexes”

MapReduce v.s. Databases

Blog by DeWitt and Stonebraker

- "Schemas are good"
- "Indexes"
- "Skew" (MR mitigates it somewhat, how?)

MapReduce v.s. Databases

Blog by DeWitt and Stonebraker

- “Schemas are good”
- “Indexes”
- “Skew” (MR mitigates it somewhat, how?)
- The M * R problem – what is it?

MapReduce v.s. Databases

Blog by DeWitt and Stonebraker

- “Schemas are good”
- “Indexes”
- “Skew” (MR mitigates it somewhat, how?)
- The M * R problem – what is it?
- “Parallel databases uses push (to sockets) instead of pull” – what’s the point?

Outline

- Spark
- MapReduce and critique
- Fault Tolerance
- Hive (short)

Fault Tolerance

Fault Tolerance

- Traditional RDBMs:
  - Major concern: recover after failure

- Massively distributed systems:
  - Probability of failure increases w/ no. of workers and length of job

Fault Tolerance

Example:

- if a server fails once/year…
- … a job with 10000 servers fails once/hour

Fault Tolerance

How is fault tolerance handled in each system?

- **MapReduce**: if a worker fails then

- **Spark**:

Fault Tolerance

How is fault tolerance handled in each system?

- **MapReduce**: if a worker fails then
  - All its completed map tasks need re-executed
  - Its in-progress reduce task needs re-executed

- **Spark**:

Fault Tolerance

How is fault tolerance handled in each system?

- **MapReduce**: if a worker fails then
  - All its completed map tasks need re-executed
  - Its in-progress reduce task needs re-executed
- **Spark**: will discuss next

Approach

New abstraction: Resilient Distributed Datasets

RDD properties

- Parallel data structure
- Can be persisted in memory
- Fault-tolerant
- Users can manipulate RDDs with rich set of operators

Resilient Distributed Datasets

- RDD = Resilient Distributed Dataset
  - Distributed, immutable.
  - Records 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

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
in memory
or on disk

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

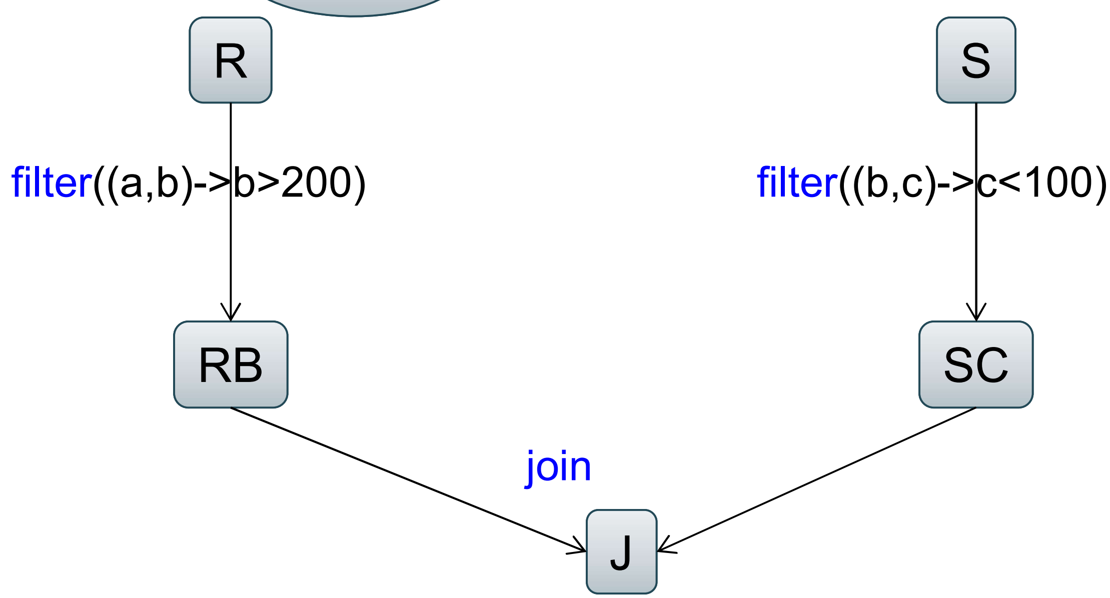
SELECT count(*) FROM R, S
WHERE R.B > 200 and S.C < 100 and R.A = S.A

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);
SC = S.filter(t -> t.c < 100);
J = RB.join(SC);
J.count();
```

transformations

action



RDD Details

- An RDD is a partitioned collection of records
  - RDD's are typed: RDD[Int] is an RDD of integers
  - Records are Java/Python objects
- An RDD is read only
  - This means no updates to individual records
  - This is to contrast with in-memory key-value stores
- To create an RDD
  - Execute a deterministic operation on another RDD
  - Or on data in stable storage
  - Example operations: map, filter, and join

RDD Materialization

- Users control persistence and partitioning

- Persistence
  - Materialize this RDD in memory

- Partitioning
  - Users can specify key for partitioning an RDD

Outline

- Spark
- MapReduce and critique
- Fault Tolerance
- Hive (short)

Hive

- Facebook's implementation of SQL over MR
- Supports subset of SQL
- Uses MapReduce runtime (pros/cons?)
  – Note: this is similar to Google's FlumeJava

Hive

- Facebook's implementation of SQL over MR
- Supports subset of SQL
- Uses MapReduce runtime (pros/cons?)
  - Note: this is similar to Google's FlumeJava
- Optimizations:

Hive

- Facebook's implementation of SQL over MR
- Supports subset of SQL
- Uses MapReduce runtime (pros/cons?)
  - Note: this is similar to Google's FlumeJava
- Optimizations:
  - Column pruning

Hive

- Facebook's implementation of SQL over MR
- Supports subset of SQL
- Uses MapReduce runtime (pros/cons?)
  - Note: this is similar to Google's FlumeJava
- Optimizations:
  - Column pruning
  - Predicate push-down

Hive

- Facebook's implementation of SQL over MR
- Supports subset of SQL
- Uses MapReduce runtime (pros/cons?)
  - Note: this is similar to Google's FlumeJava
- Optimizations:
  - Column pruning
  - Predicate push-down
  - Partition pruning

Hive

- Facebook’s implementation of SQL over MR
- Supports subset of SQL
- Uses MapReduce runtime (pros/cons?)
  - Note: this is similar to Google’s FlumeJava
- Optimizations:
  - Column pruning
  - Predicate push-down
  - Partition pruning
  - Map-side join = ”broadcast join” (discuss in class)

Hive

- Facebook's implementation of SQL over MR
- Supports subset of SQL
- Uses MapReduce runtime (pros/cons?)
  – Note: this is similar to Google's FlumeJava
- Optimizations:
  – Column pruning
  – Predicate push-down
  – Partition pruning
  – Map-side join = ”broadcast join” (discuss in class)
  – Join reordering

Discussion

- Parallel database systems: since the 80s
- MapReduce: around 2000
- Hive: built on MapReuce
- Spark: “better” MapReduce around 2010
- Snowflake, Aurora: cloud, parallel databases; around 2015

Quick comparison (next slides)

MapReduce v.s. Spark

MapReduce	Spark
• Job = Map+Reduce	• Job = any query
• Language = Java	• Language ≈ RA
• Data = untyped	• Data = has schema
• Optimization = no	• Optimization = yes but limited: missing stats on base data

Spark v.s. RDBMS

- Query language = its own proprietary
- Optimizer = limited
- Runtime = its own proprietary
- External functions = yes; very useful in ML

- Query language = SQL
- Optimizer = full scale
- Runtime = efficient SQL query engine
- External functions = no

Outline

- Spark Review
- MapReduce and critique
- Fault Tolerance
- Hive (short)

Next lecture: $\boxed{\text{Parallel databases (Start Today)}}$

Parallel Databases

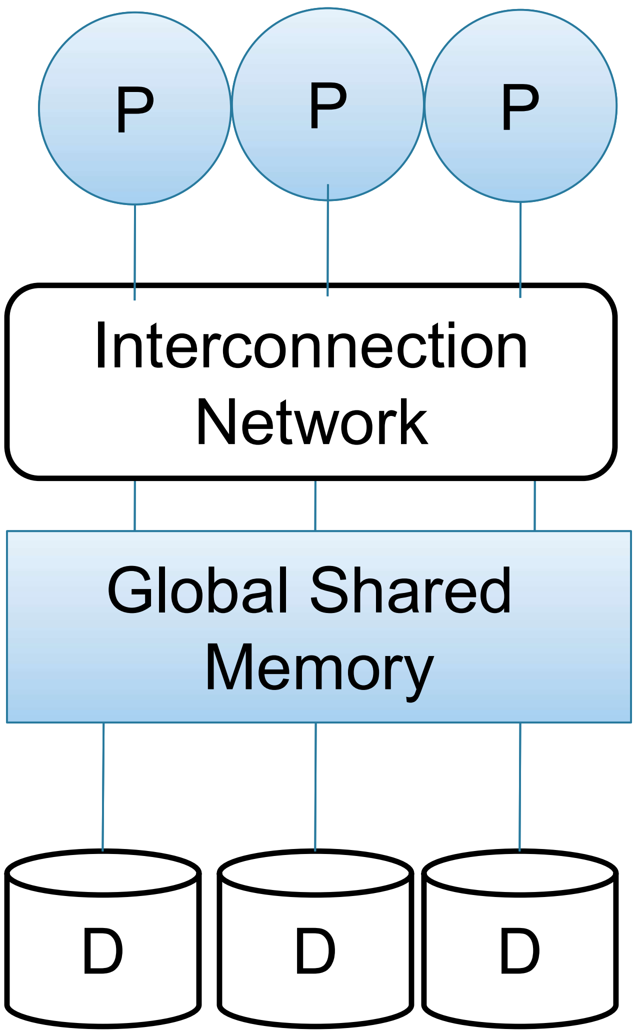
Outline

- Basic notions
- Distributed query processing algorithms (Start)
- Skew (will continue next lecture)

Architectures for Parallel Databases

- Shared memory
- Shared disk
- Shared nothing

Shared Memory

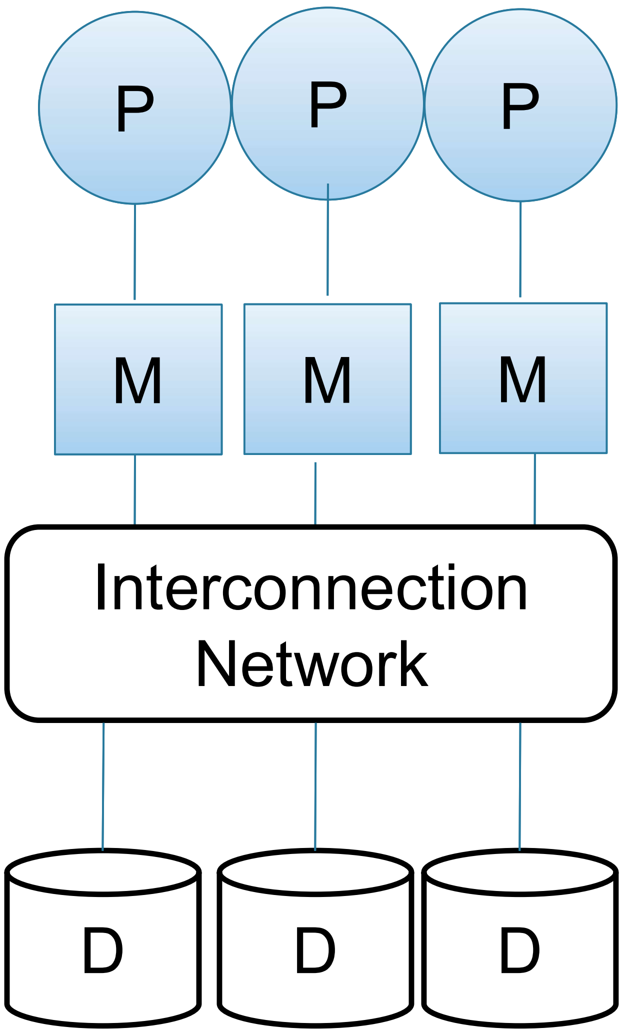


- SMP = symmetric multiprocessor
- Nodes share RAM and disk
- 10x … 100x processors

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

- Easy to use and program
- Expensive to scale

Shared Disk



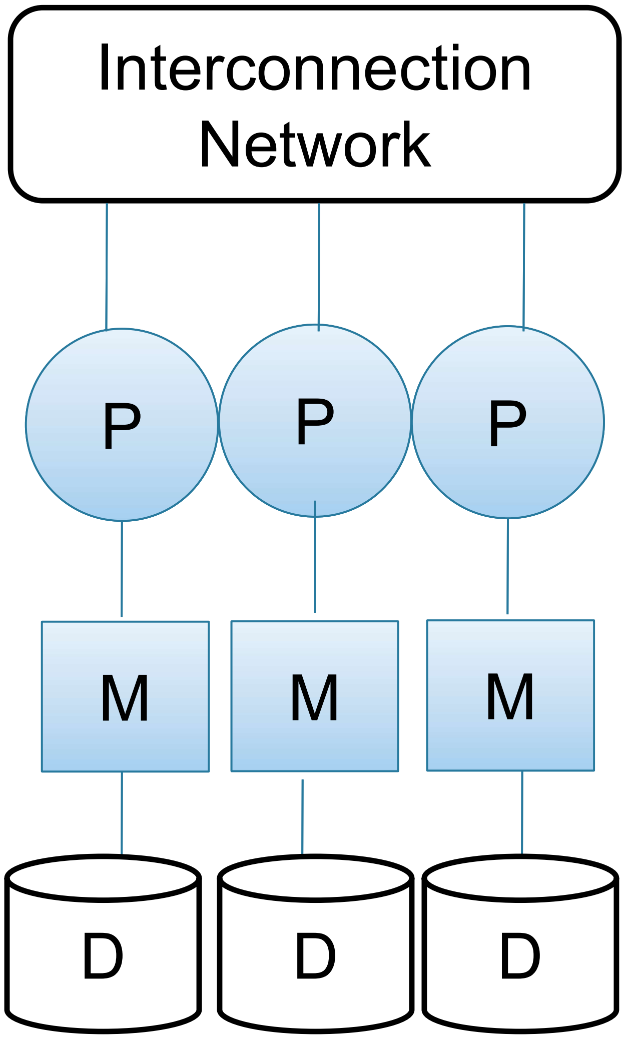
- All nodes access same disks
- 10x processors

- Example: Oracle

- No more memory contention

- Harder to program
- Still hard to scale

Shared Nothing



- Cluster of commodity machines
- Called "clusters" or "blade servers"
- Each machine: own memory & disk
- Up to x1000-x10000 nodes
- Example: redshift, spark, snowflake

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.

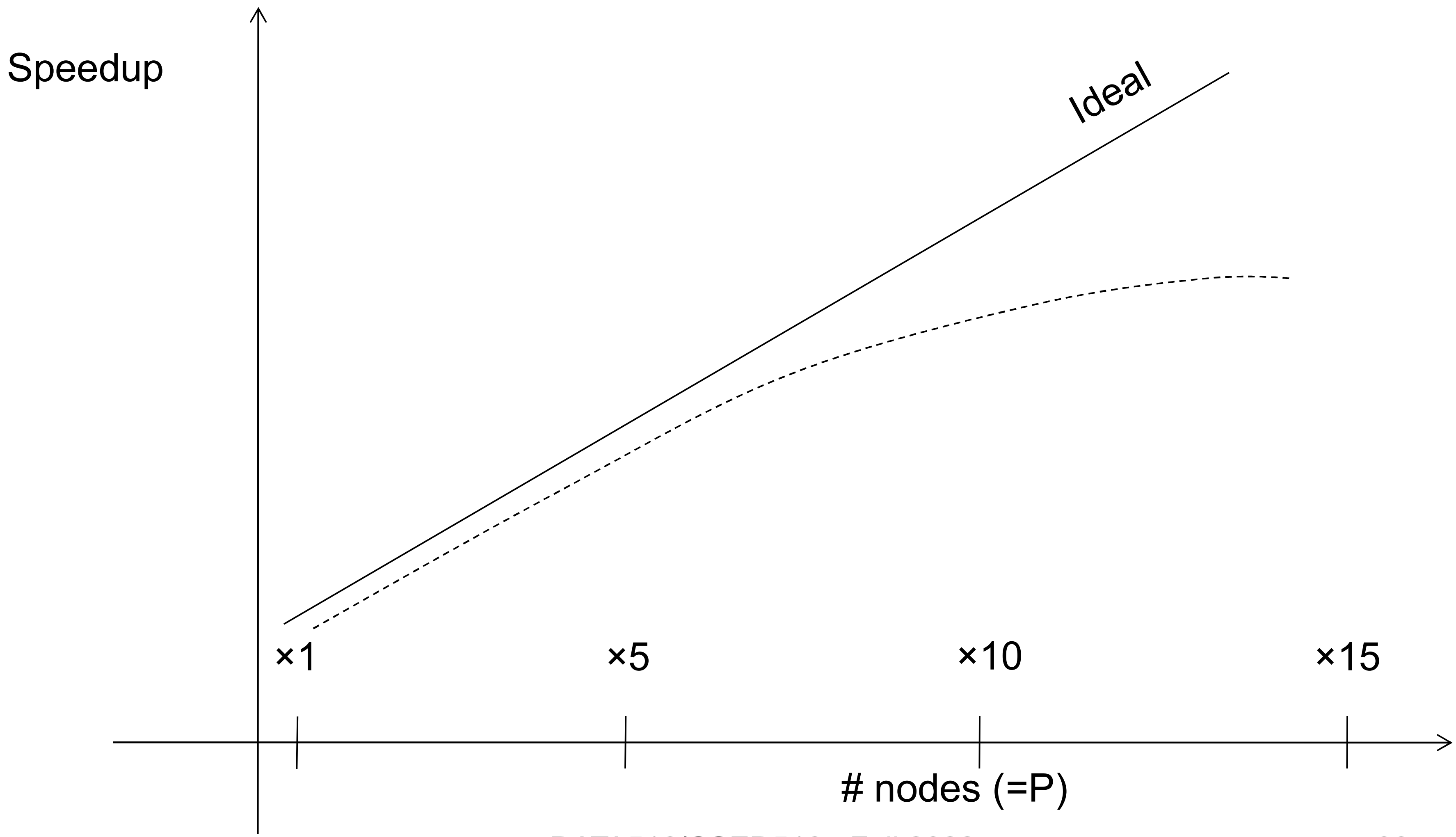
Performance Metrics

Nodes = processors = computers

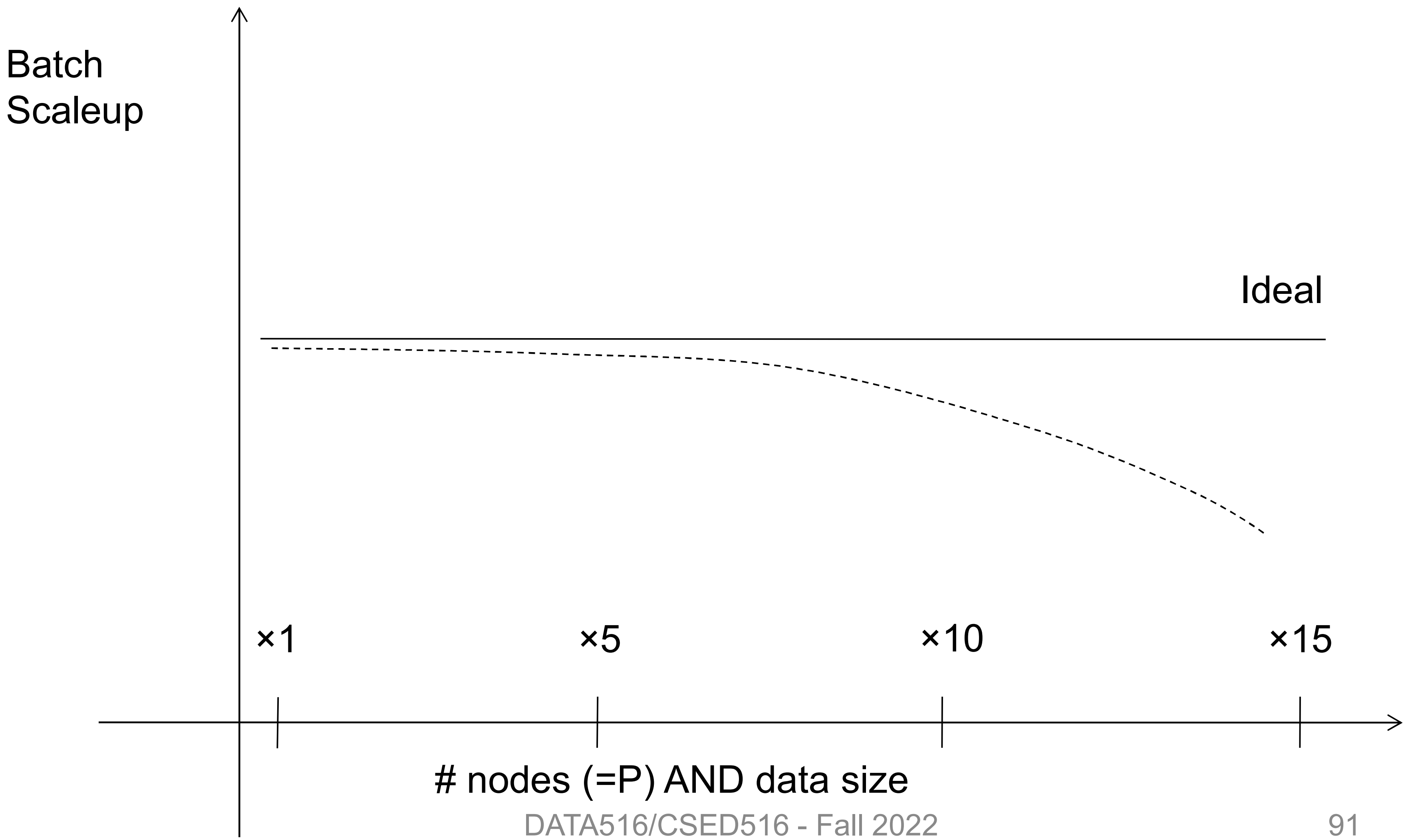
- Speed Up:
  - More nodes, same data ➔ higher speed
- Scale Up:
  - More nodes, more data ➔ same speed

Disclaimer: *Scale Up* is often mis-used as *Speed Up*

Linear v.s. Non-linear Speedup



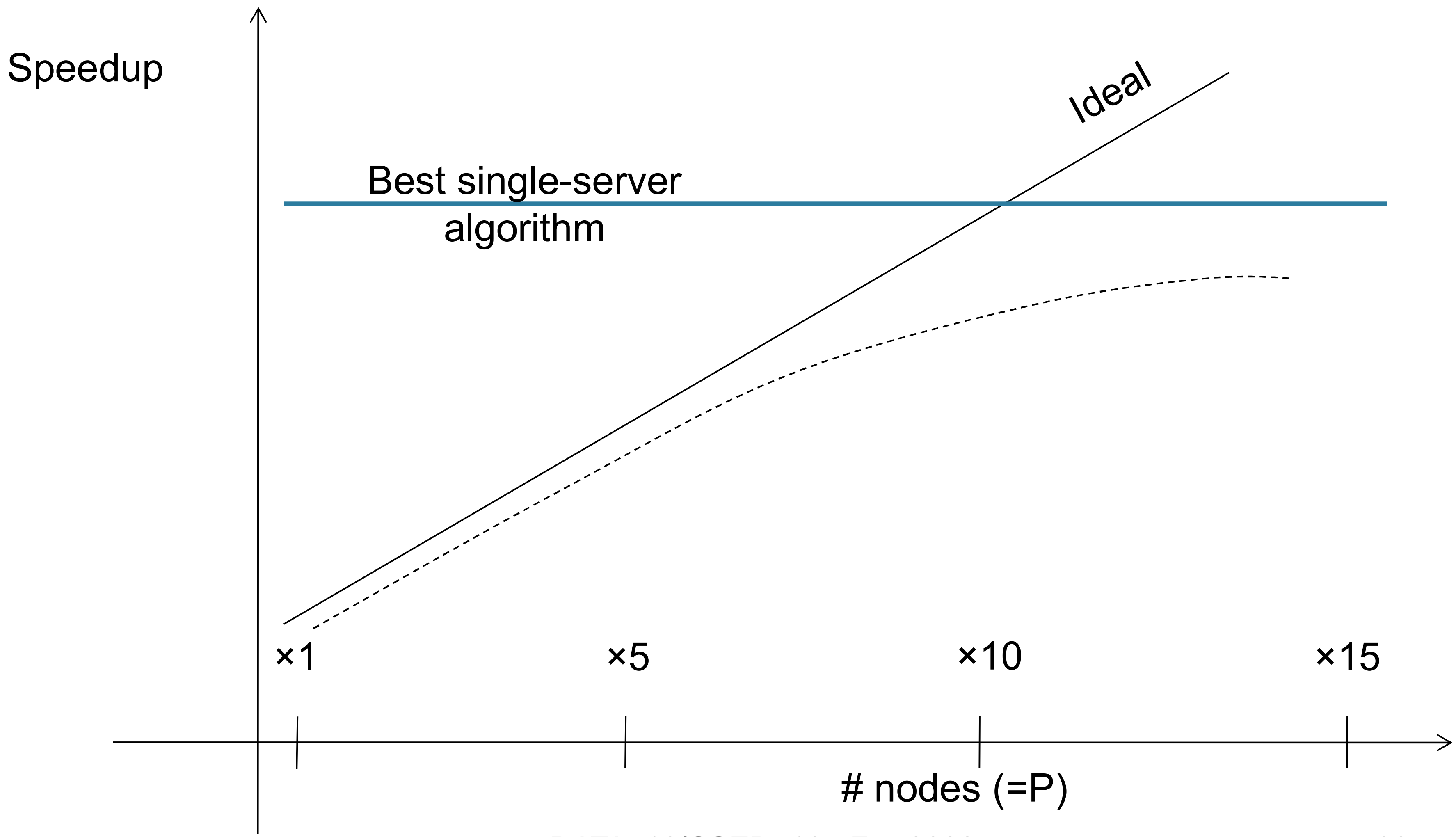
Linear v.s. Non-linear Scaleup



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

“Scalability but at what cost?”



Discussion

Parallel/distributed data processing:

- Scales up* to more data:
  - More servers can hold more data

- Speedup w/ number of nodes:
  - Harder to achieve
  - But can get there in with more nodes/future research

* “Scale-up” is often used informally, like here

Outline

- Basic notions
- Distributed query processing algorithms
- Skew (will continue next lecture)

Distributed Query Processing Algorithms

Horizontal Data Partitioning

Table

R

sid	name	…	…

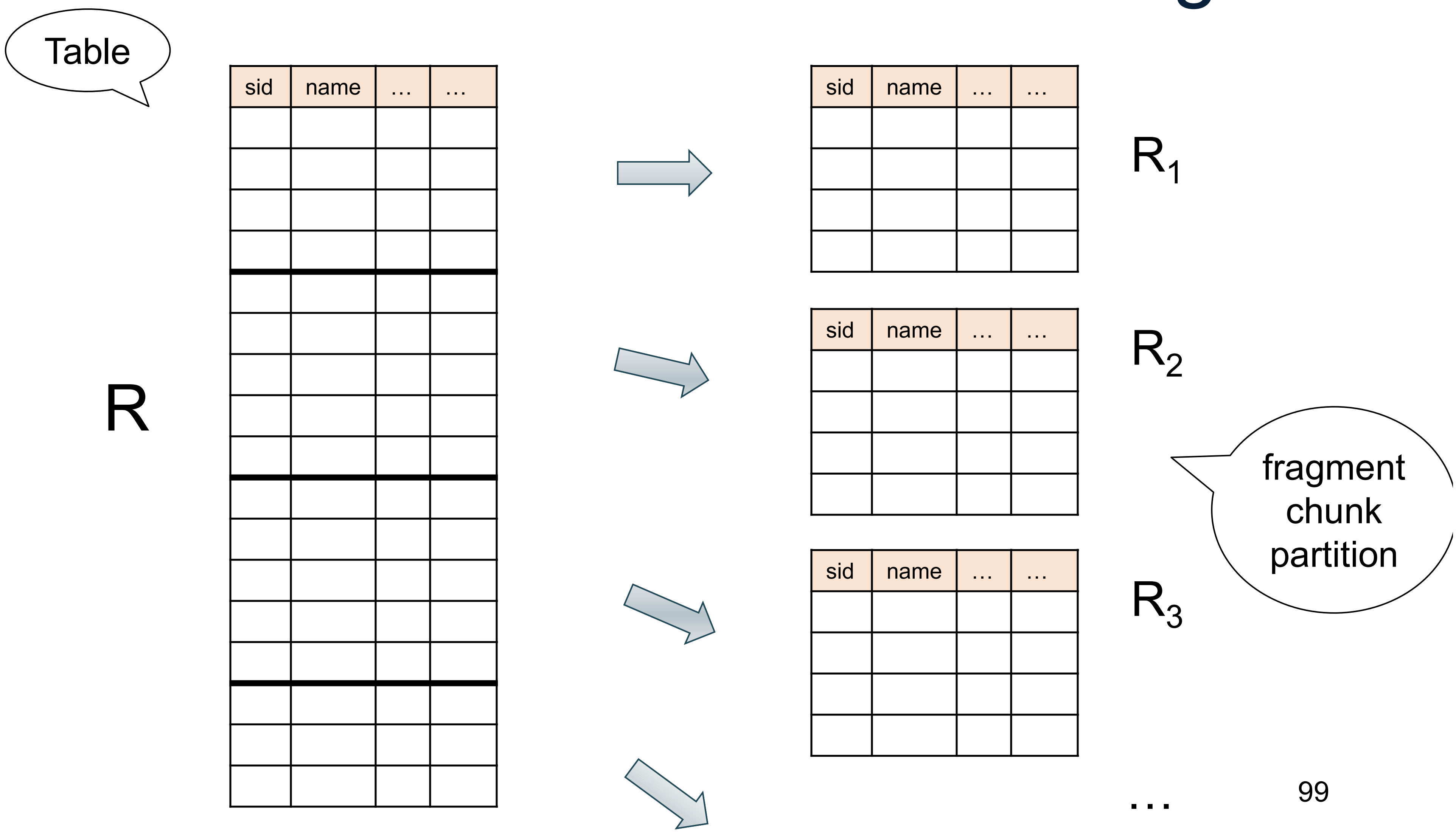
Horizontal Data Partitioning

Table

R

sid	name	…	…

Horizontal Data Partitioning



Horizontal Data Partitioning

- Block Partition, a.k.a. Round Robin:
  - 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

- 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 $v_{i-1} < t.A < v_i$

Notations

p = number of servers (nodes) that hold the chunks

When a relation R is distributed to p servers,
we draw the picture like this:

R_1 R_2 R_P

Here R_1 is the fragment of R stored on server 1, etc

$$R = R_1 \cup R_2 \cup \cdots \cup R_P$$

Uniform Load and Skew

- $|R|$ = N tuples, then $|R_1| + |R_2| + \ldots + |R_p| = N$
- We say the load is uniform when:
  $$|R_1| \approx |R_2| \approx \ldots \approx |R_p| \approx N/p$$
- Skew means that some load is much larger:
  $$\max_i |R_i| >> N/p$$

We design algorithms for uniform load, discuss skew later

Parallel Algorithm

- Selection σ
- Join $\bowtie$
- Group by γ

Parallel Selection

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

Query: $\sigma_{A=v}(R)$, or $\sigma_{v1<A<v2}(R)$

- Block partitioned:
- Hash partitioned:
- Range partitioned:

Parallel Selection

Data: R(K, A, B, C)

Query: $\sigma_{A=v}(R)$, or $\sigma_{v1<A<v2}(R)$

- Block partitioned:
  - All servers need to scan
- Hash partitioned:
- Range partitioned:

Parallel Selection

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

Query: $\sigma_{A=v}(R)$, or $\sigma_{v1<A<v2}(R)$

- Block partitioned:
  - All servers need to scan
- Hash partitioned:
  - Point query: only one server needs to scan
  - Range query: all servers need to scan
- Range partitioned:

Parallel Selection

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

Query: $\sigma_{A=v}(R)$, or $\sigma_{v1<A<v2}(R)$

- Block partitioned:
  - All servers need to scan
- Hash partitioned:
  - Point query: only one server needs to scan
  - Range query: all servers need to scan
- Range partitioned:
  - Only some servers need to scan

Parallel GroupBy

Data: $R(\underline{K}, A, B, C)$

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

Discuss in class how to compute in each case:

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

Parallel GroupBy

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

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

Discuss in class how to compute in each case:

- R is hash-partitioned on A
  - Each server i computes locally $\gamma_{A,sum(C)}(R_i)$
- R is block-partitioned or hash-partitioned on K

Parallel GroupBy

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

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

Discuss in class how to compute in each case:

- R is hash-partitioned on A
  - Each server i computes locally $\gamma_{A,sum(C)}(R_i)$
- R is block-partitioned or hash-partitioned on K
  - Need to reshuffle data on A first (next slide)
  - Then compute locally $\gamma_{A,sum(C)}(R_i)$

Basic Parallel GroupBy

Data: R($\underline{K}$, A, B, C)

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

- R is block-partitioned or hash-partitioned on K

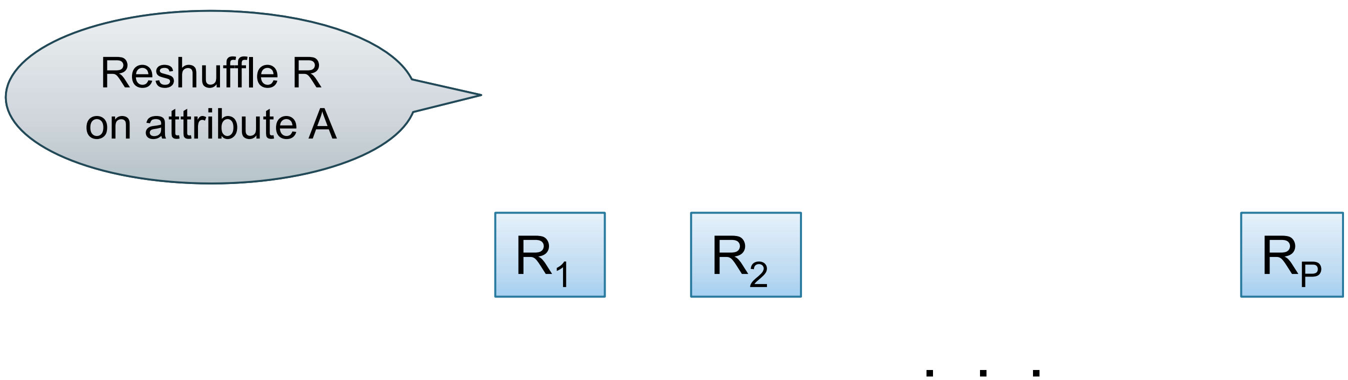
R_1 R_2 . . . R_P

Basic Parallel GroupBy

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

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

- R is block-partitioned or hash-partitioned on K

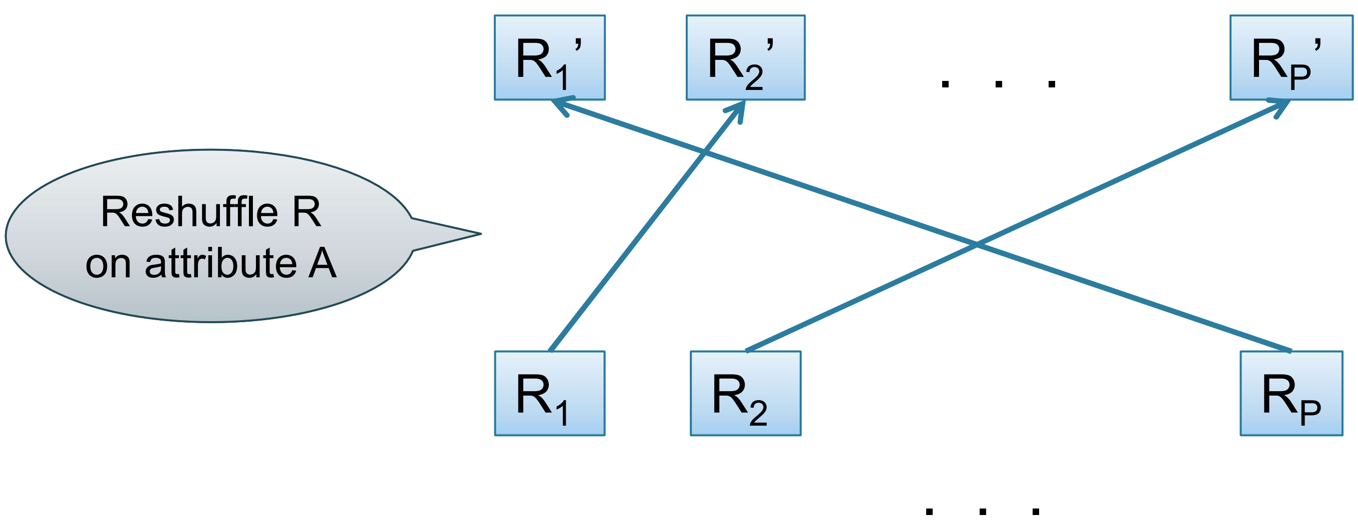


Basic Parallel GroupBy

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

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

- R is block-partitioned or hash-partitioned on K

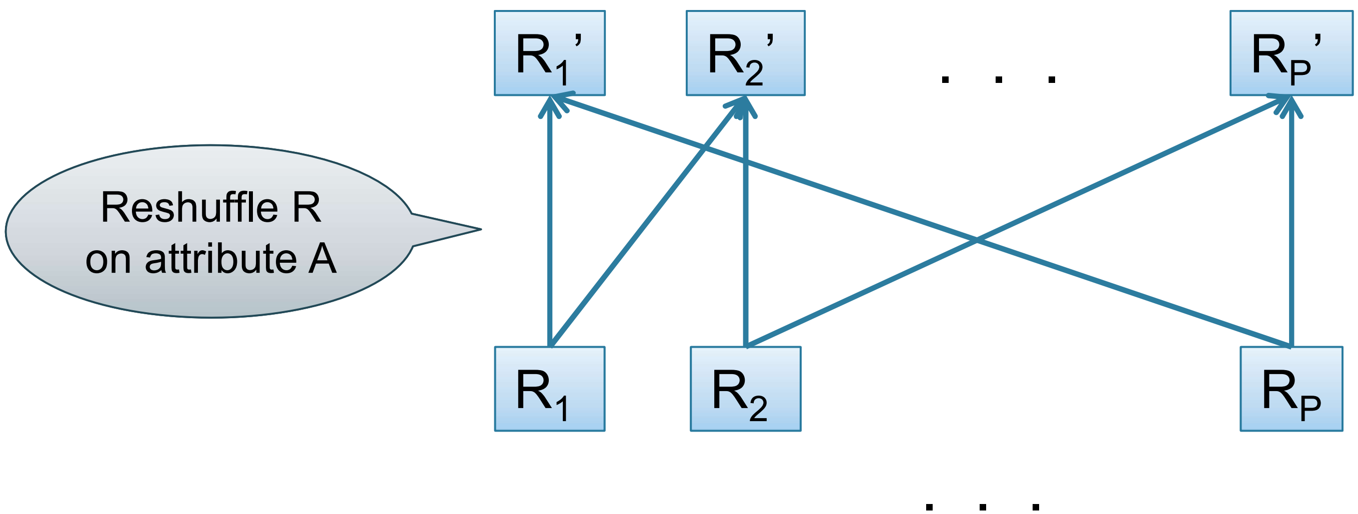


Basic Parallel GroupBy

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

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

- R is block-partitioned or hash-partitioned on K

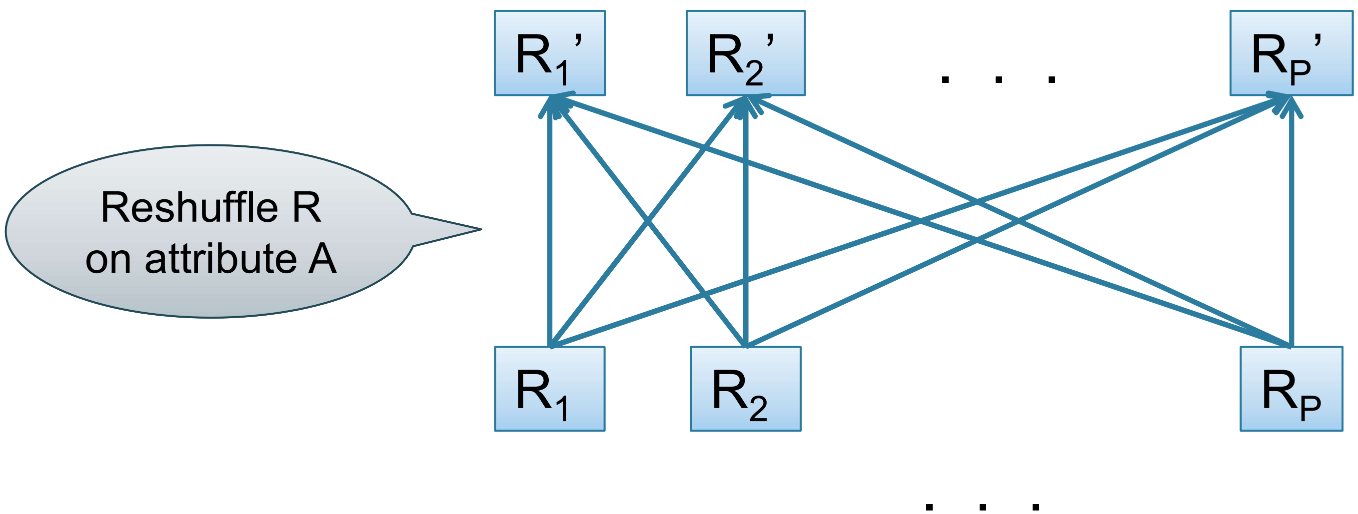


Basic Parallel GroupBy

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

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

- R is block-partitioned or hash-partitioned on K

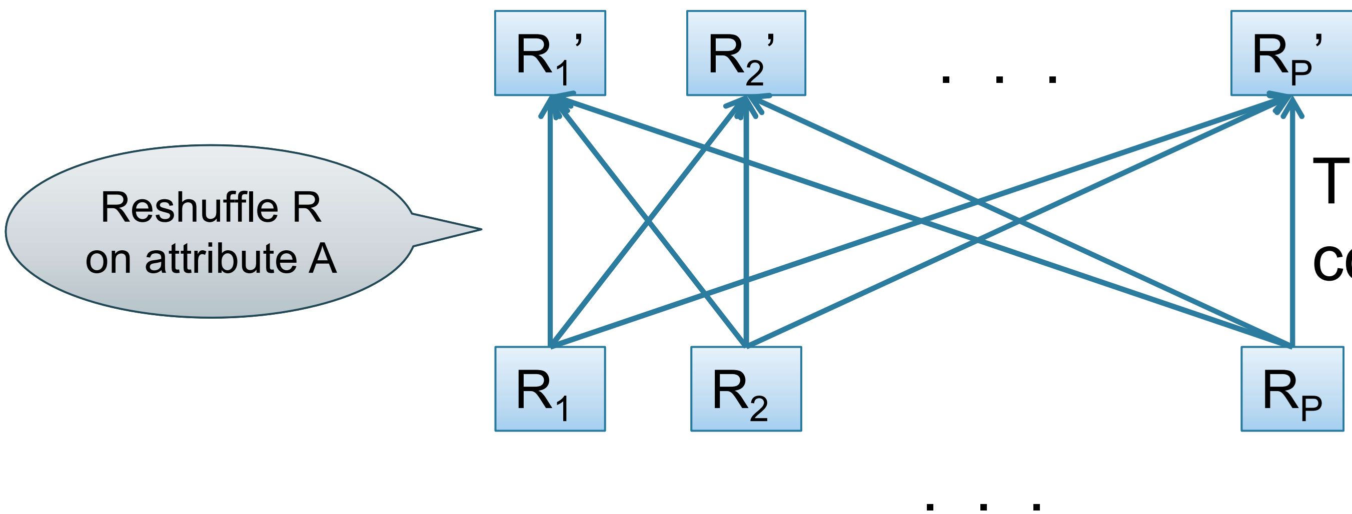


Basic Parallel GroupBy

Data: R(K, A, B, C)

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

- R is block-partitioned or hash-partitioned on K



This is done in ***one*** communication step

Reshuffling

- Nodes send data over the network
- Many-many communications possible
- Throughput:
  - Better than disk
  - Worse than main memory

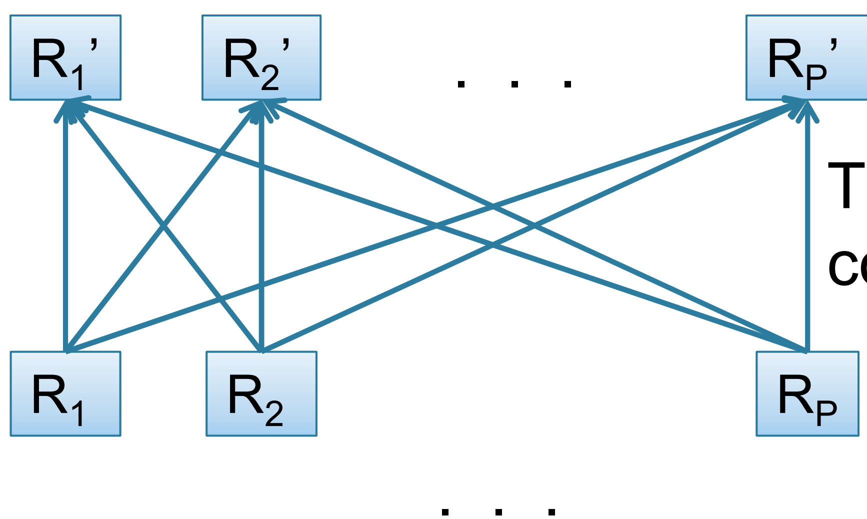
Basic Parallel GroupBy

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

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

- R is block-partitioned or hash-partitioned on K

Reshuffle R on attribute A



This is done in *one* communication step

Can you think of an optimization?

GroupBy/Union Commutativity

	city	…	qant
	Seattle		10
	LA		20
	Seattle		30
	NY		40

	city	…	qant
	LA		22
	NY		33
	LA		44
	Austin		55

	city	…	qant
	Seattle		66
	LA		77
	NY		88
	LA		99

```
SELECT city, sum(quant)
FROM R
GROUP BY city
```

GroupBy/Union Commutativity

	city	…	qant
	Seattle		10
	LA		20
	Seattle		30
	NY		40

	city	…	qant
	LA		22
	NY		33
	LA		44
	Austin		55

	city	…	qant
	Seattle		66
	LA		77
	NY		88
	LA		99

Q: What is sum for Seattle?

```
SELECT city, sum(quant)
FROM R
GROUP BY city
```

GroupBy/Union Commutativity

	city	…	qant
	Seattle		**10**
	LA		20
	Seattle		**30**
	NY		40

	city	…	qant
	LA		22
	NY		33
	LA		44
	Austin		55

	city	…	qant
	Seattle		**66**
	LA		77
	NY		88
	LA		99

Q: What is sum for Seattle?
A: 106

```
SELECT city, sum(quant)
FROM R
GROUP BY city
```

GroupBy/Union Commutativity

	city	…	qant
	Seattle		**10**
	LA		20
	Seattle		**30**
	NY		40

Sum here = 40

Q: What is sum for Seattle?
A: 106

	city	…	qant
	LA		22
	NY		33
	LA		44
	Austin		55

```
SELECT city, sum(quant)
FROM R
GROUP BY city
```

	city	…	qant
	Seattle		**66**
	LA		77
	NY		88
	LA		99

Sum here = 66

GroupBy/Union Commutativity

	city	…	qant
	Seattle		**10**
	LA		20
	Seattle		**30**
	NY		40

Sum here = 40

Q: What is sum for Seattle?
A: 106

	city	…	qant
	LA		22
	NY		33
	LA		44
	Austin		55

SELECT city, sum(quant)
FROM R
GROUP BY city

	city	…	qant
	Seattle		**66**
	LA		77
	NY		88
	LA		99

Sum here = 66

$$\gamma_{city,sum(q)}(R_1 \cup R_2 \cup R_3) =$$

GroupBy/Union Commutativity

	city	…	qant
	Seattle		**10**
	LA		20
	Seattle		**30**
	NY		40

Sum here = 40

Q: What is sum for Seattle?
A: 106

	city	…	qant
	LA		22
	NY		33
	LA		44
	Austin		55

```
SELECT city, sum(quant)
FROM R
GROUP BY city
```

	city	…	qant
	Seattle		**66**
	LA		77
	NY		88
	LA		99

Sum here = 66

$$\gamma_{city,sum(q)}(R_1 \cup R_2 \cup R_3) =$$

$$= \gamma_{city,sum(q)}\left(\gamma_{city,sum(q)}(R_1) \cup \gamma_{city,sum(q)}(R_2) \cup \gamma_{city,sum(q)}(R_3)\right)$$

Basic Parallel GroupBy

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

Basic Parallel GroupBy

Data: R(<u>K</u>, A, B, C)
Query: $\gamma_{A,sum(C)}(R)$

Step 0: [Optimization] each server i computes local group-by:
$$T_i = \gamma_{A,sum(C)}(R_i)$$

Basic Parallel GroupBy

Data: R(<u>K</u>, A, B, C)
Query: $\gamma_{A,sum(C)}(R)$

Step 0: [Optimization] each server i computes local group-by:

$$T_i = \gamma_{A,sum(C)}(R_i)$$

Step 1: partitions tuples in T_i using hash function h(A):

$$T_{i,1}, T_{i,2}, \ldots, T_{i,p}$$

then send fragment $T_{i,j}$ to server j

Basic Parallel GroupBy

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

Step 0: [Optimization] each server i computes local group-by:

$$T_i = \gamma_{A,sum(C)}(R_i)$$

Step 1: partitions tuples in T_i using hash function h(A):

$$T_{i,1}, T_{i,2}, \ldots, T_{i,p}$$

then send fragment $T_{i,j}$ to server j

Step 2: receive fragments, union them, then group-by

$$R_j' = T_{1,j} \cup \ldots \cup T_{p,j}$$

$$Answer_j = \gamma_{A, sum(C)} (R_j')$$

Pushing Aggregates Past Union

Which other rules can we push past union?

- Sum?
- Count?
- Avg?
- Max?
- Median?

Pushing Aggregates Past Union

Which other rules can we push past union?

- Sum?
- Count?
- Avg?
- Max?
- Median?

Distributive	Algebraic	Holistic
$sum(a_1+a_2+\ldots+a_9)=$ $sum(sum(a_1+a_2+a_3)+$ $sum(a_4+a_5+a_6)+$ $sum(a_7+a_8+a_9))$	avg(B) = sum(B)/count(B)	median(B)

Example Query with Group By

```
SELECT a, sum(b) as sb
FROM R WHERE c > 0
GROUP BY a
```

Example Query with Group By

```
SELECT a, sum(b) as sb
FROM R WHERE c > 0
GROUP BY a
```

$$\gamma_{\text{a, sum(b)} \rightarrow \text{sb}}$$

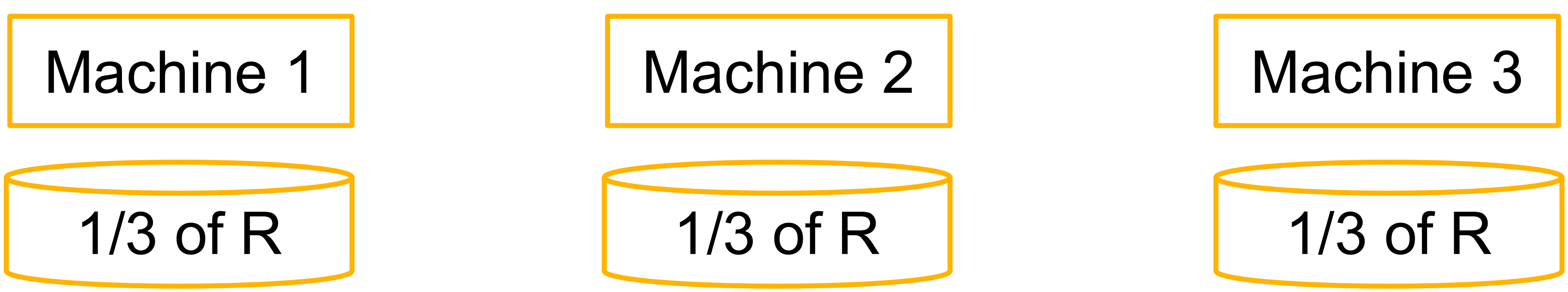
$$\sigma_{c>0}$$

$$R$$

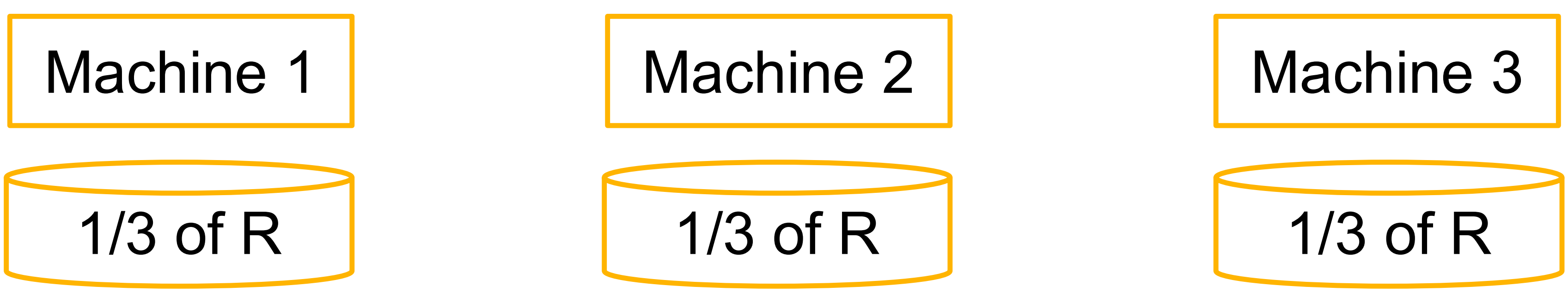
Example Query with Group By

```
SELECT a, sum(b) as sb
FROM R WHERE c > 0
GROUP BY a
```

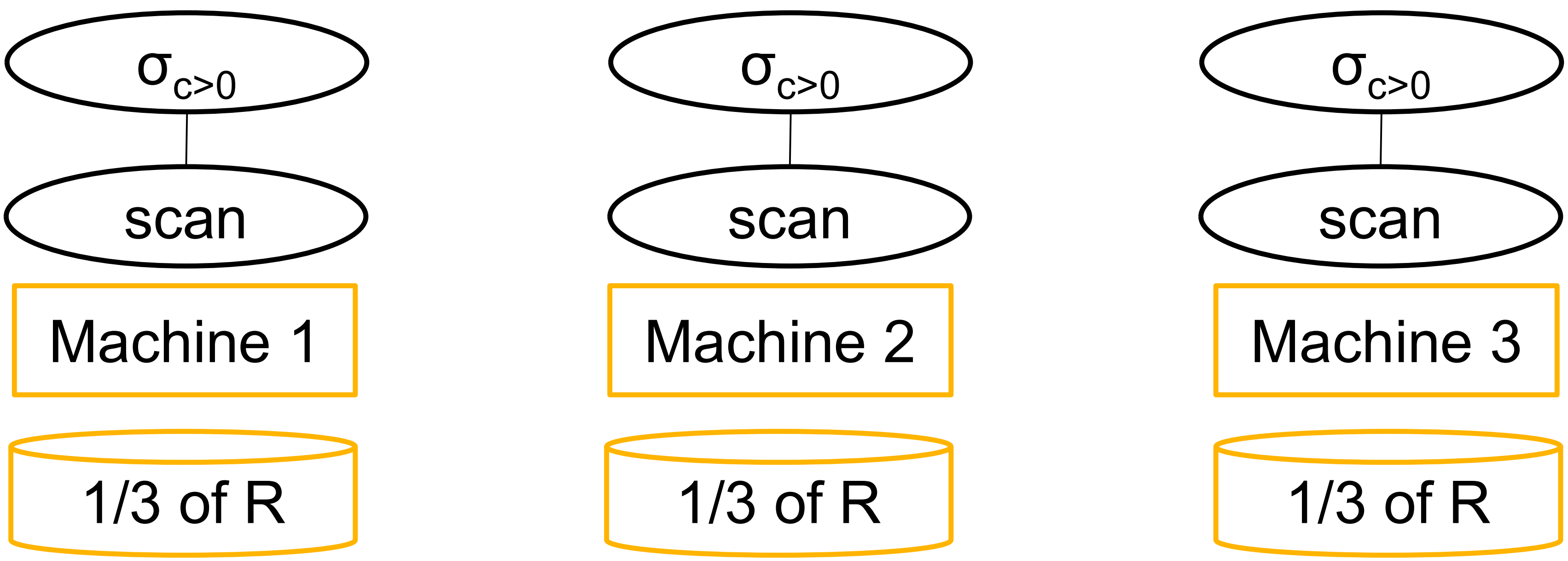
$$\gamma_{a,\ sum(b) \rightarrow sb}$$
$$|$$
$$\sigma_{c>0}$$
$$|$$
$$R$$



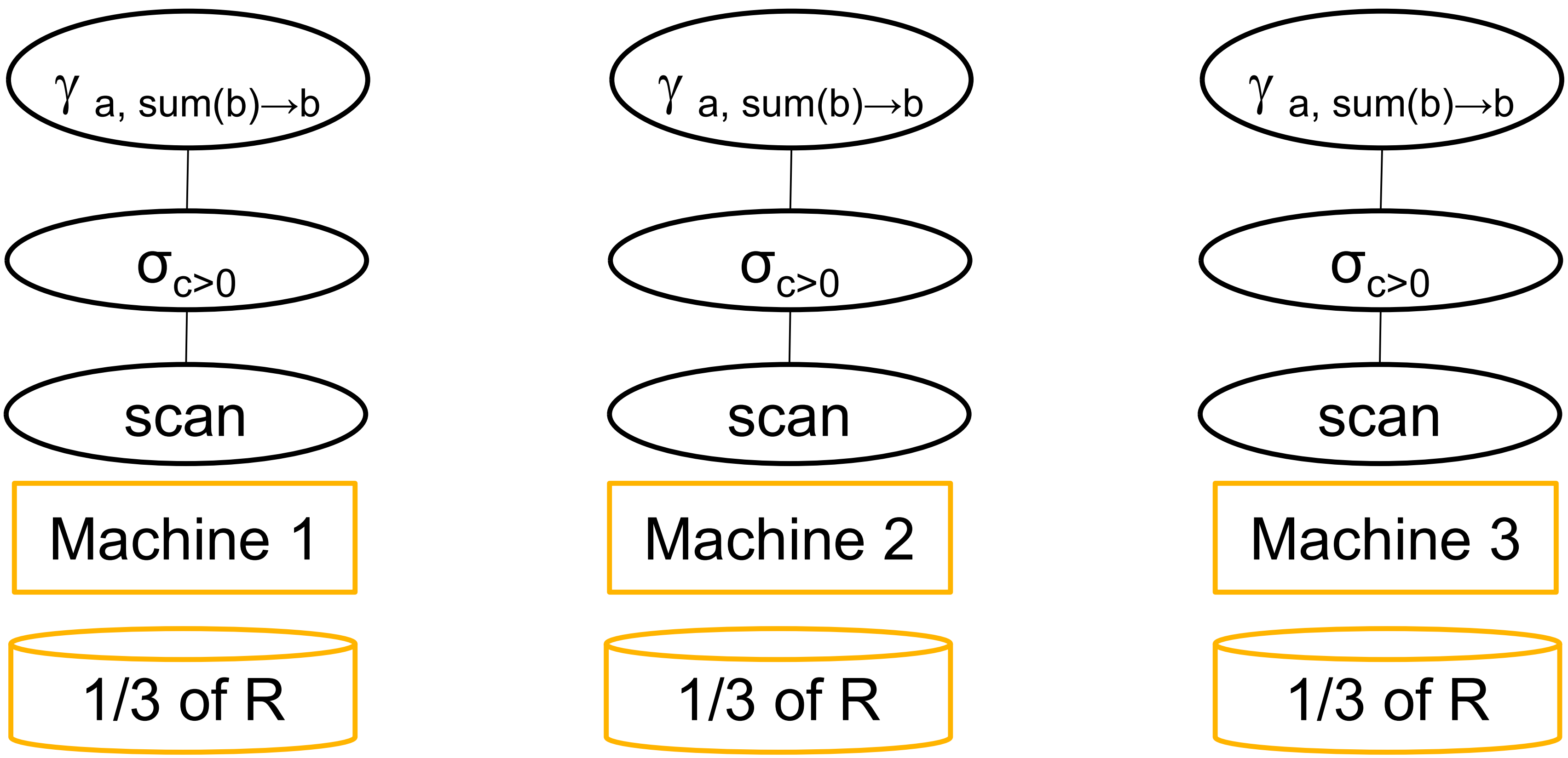
SELECT a, sum(b) as sb FROM R WHERE c > 0 GROUP BY a



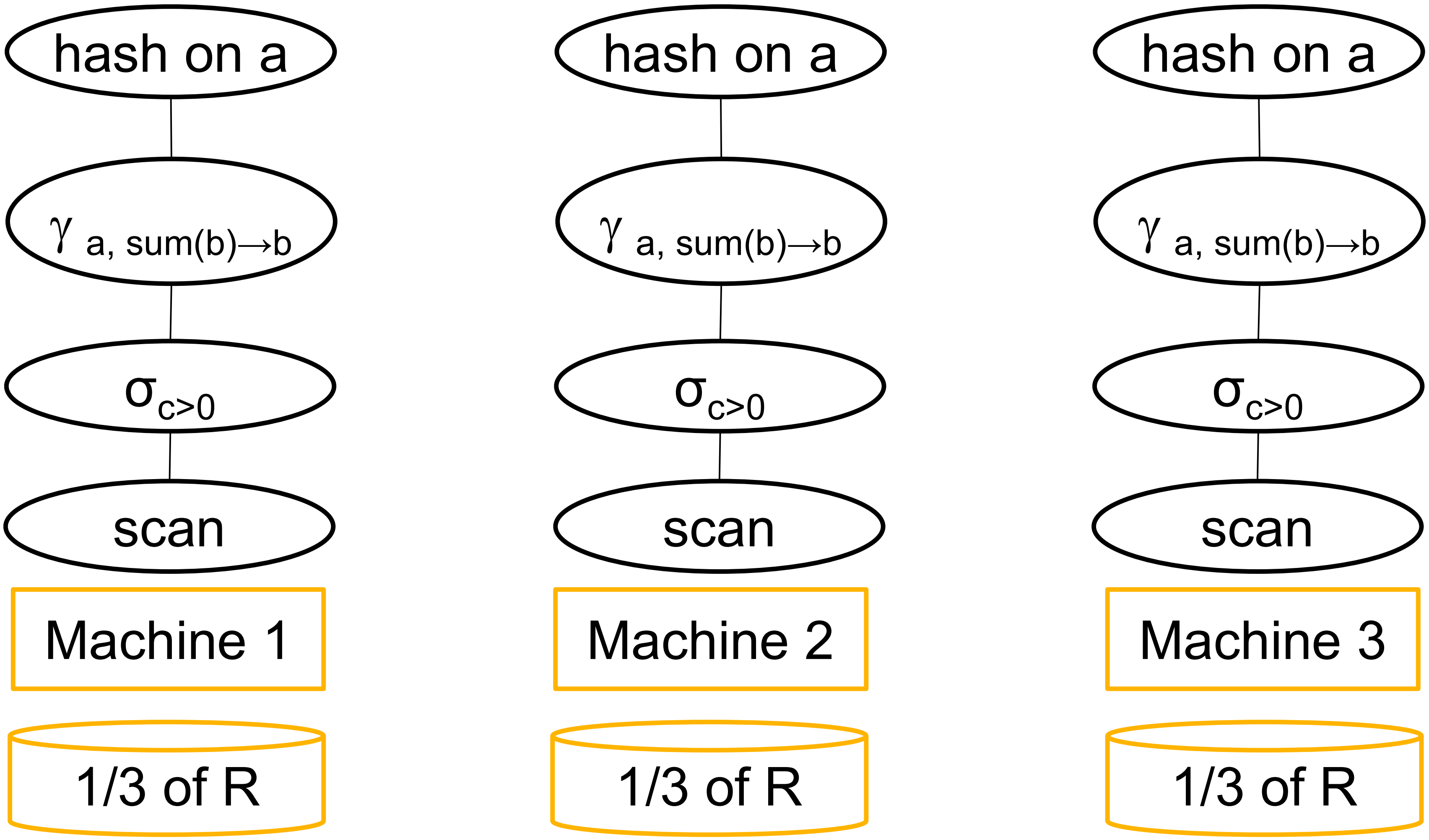
```
SELECT a, sum(b) as sb    FROM R   WHERE c > 0 GROUP BY a
```

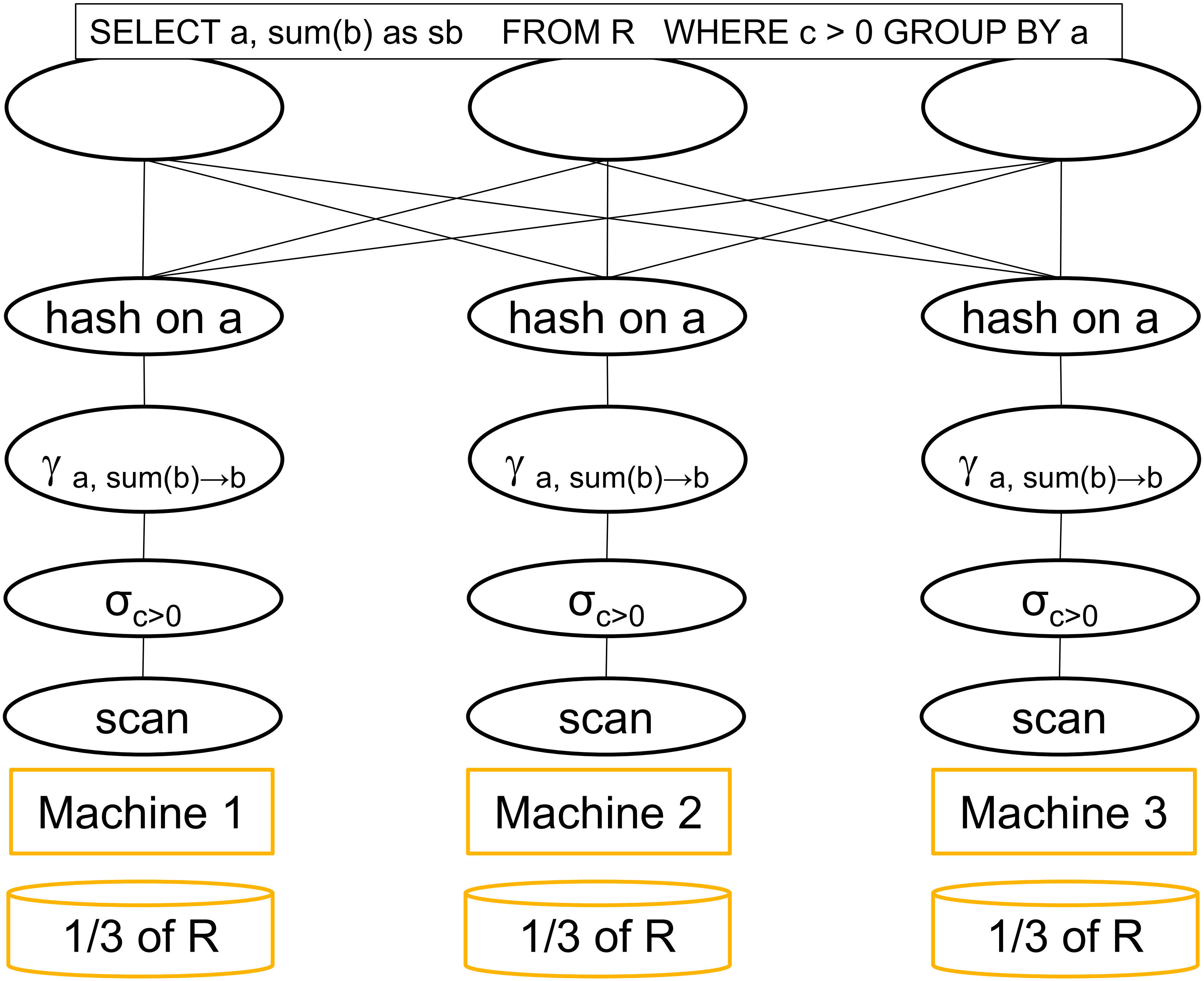


SELECT a, sum(b) as sb FROM R WHERE c > 0 GROUP BY a

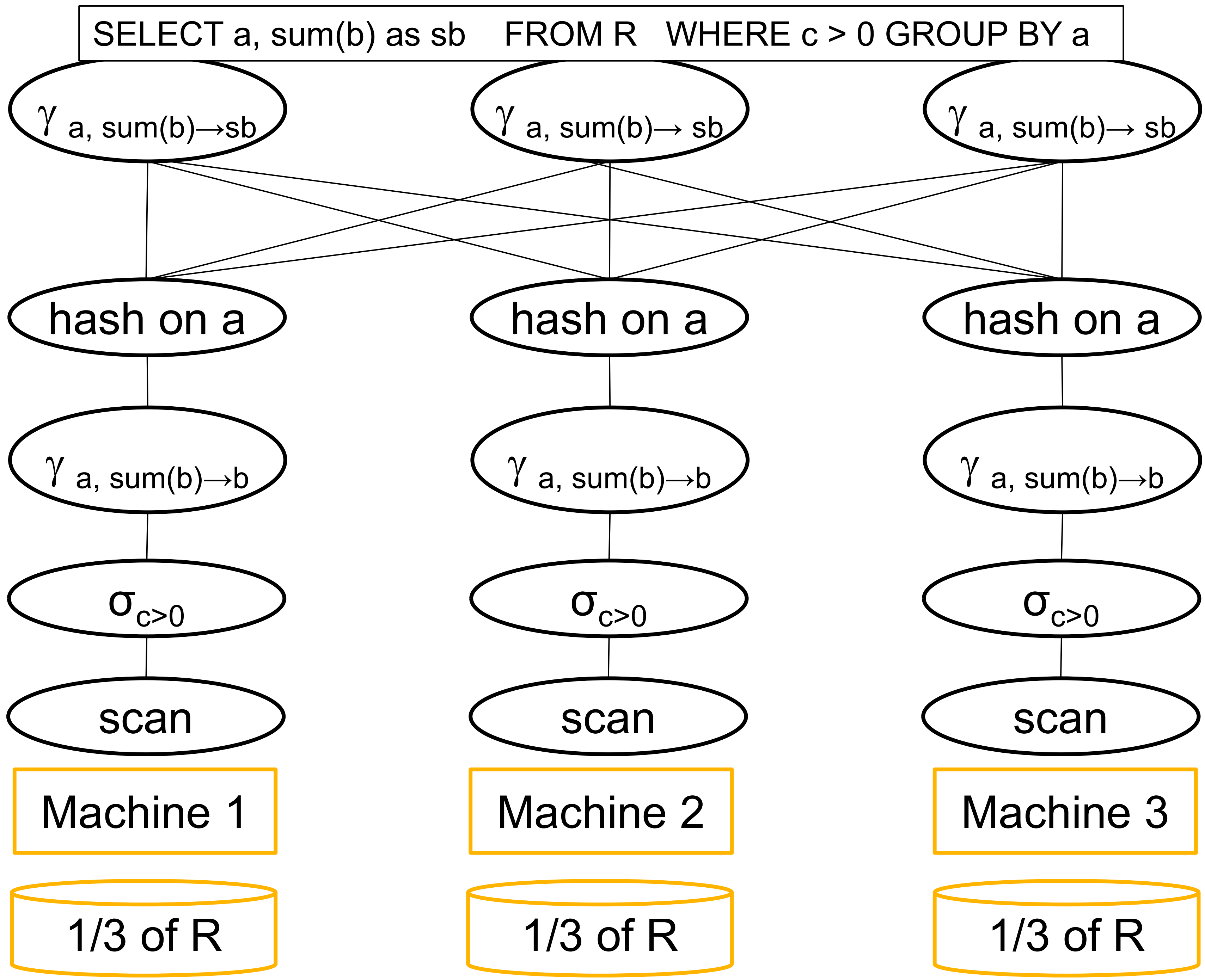


SELECT a, sum(b) as sb FROM R WHERE c > 0 GROUP BY a





SELECT a, sum(b) as sb FROM R WHERE c > 0 GROUP BY a
hash on a
hash on a
hash on a
$\gamma_{a,\ sum(b) \rightarrow b}$
$\gamma_{a,\ sum(b) \rightarrow b}$
$\gamma_{a,\ sum(b) \rightarrow b}$
$\sigma_{c>0}$
$\sigma_{c>0}$
$\sigma_{c>0}$
scan
scan
scan
Machine 1
Machine 2
Machine 3
1/3 of R
1/3 of R
1/3 of R


SELECT a, sum(b) as sb FROM R WHERE c > 0 GROUP BY a
γ a, sum(b)→sb
γ a, sum(b)→ sb
γ a, sum(b)→ sb
hash on a
hash on a
hash on a
γ a, sum(b)→b
γ a, sum(b)→b
γ a, sum(b)→b
σc>0
σc>0
σc>0
scan
scan
scan
Machine 1
Machine 2
Machine 3
1/3 of R
1/3 of R
1/3 of R

Speedup and Scaleup

Consider the query $\gamma_{A,sum(C)}(R)$
Assume the local runtime for group-by is linear O(|R|)

If we double number of nodes P, what is the runtime?

If we double both P and size of R, what is the runtime?

Speedup and Scaleup

Consider the query $\gamma_{A,sum(C)}(R)$
Assume the local runtime for group-by is linear $O(|R|)$

If we double number of nodes P, what is the runtime?

- Half (chunk sizes become ½)

If we double both P and size of R, what is the runtime?

- Same (chunk sizes remain the same)

Speedup and Scaleup

Consider the query $\gamma_{A,sum(C)}(R)$
Assume the local runtime for group-by is linear O(|R|)

If we double number of nodes P, what is the runtime?

- Half (chunk sizes become ½)

If we double both P and size of R, what is the runtime?

- Same (chunk sizes remain the same)

But only if the data is without skew!