

# Announcements

- HW 6 releases tonight
  - Due Nov. 20th
  - Waiting for AWS credit can take up to *two days*
  - Sign up early:
    - <https://aws.amazon.com/education/awseducate/apply/>
    - <https://piazza.com/class/jmftm54e88t2kk?cid=452>
- Extended office hours Friday to help with first parts of HW 6: 11:30 to 5:00pm in CSE 023
- Extra office hours 5:30pm today on 2<sup>nd</sup> Floor Breakout

# Introduction to Database Systems

## CSE 414

### Lecture 19: Parallel DBMS

# 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 and Hadoop
- Unit 6: DBMS usability, conceptual design
- Unit 7: Transactions
- Unit 8: Advanced topics (time permitting)

# Why compute in parallel?

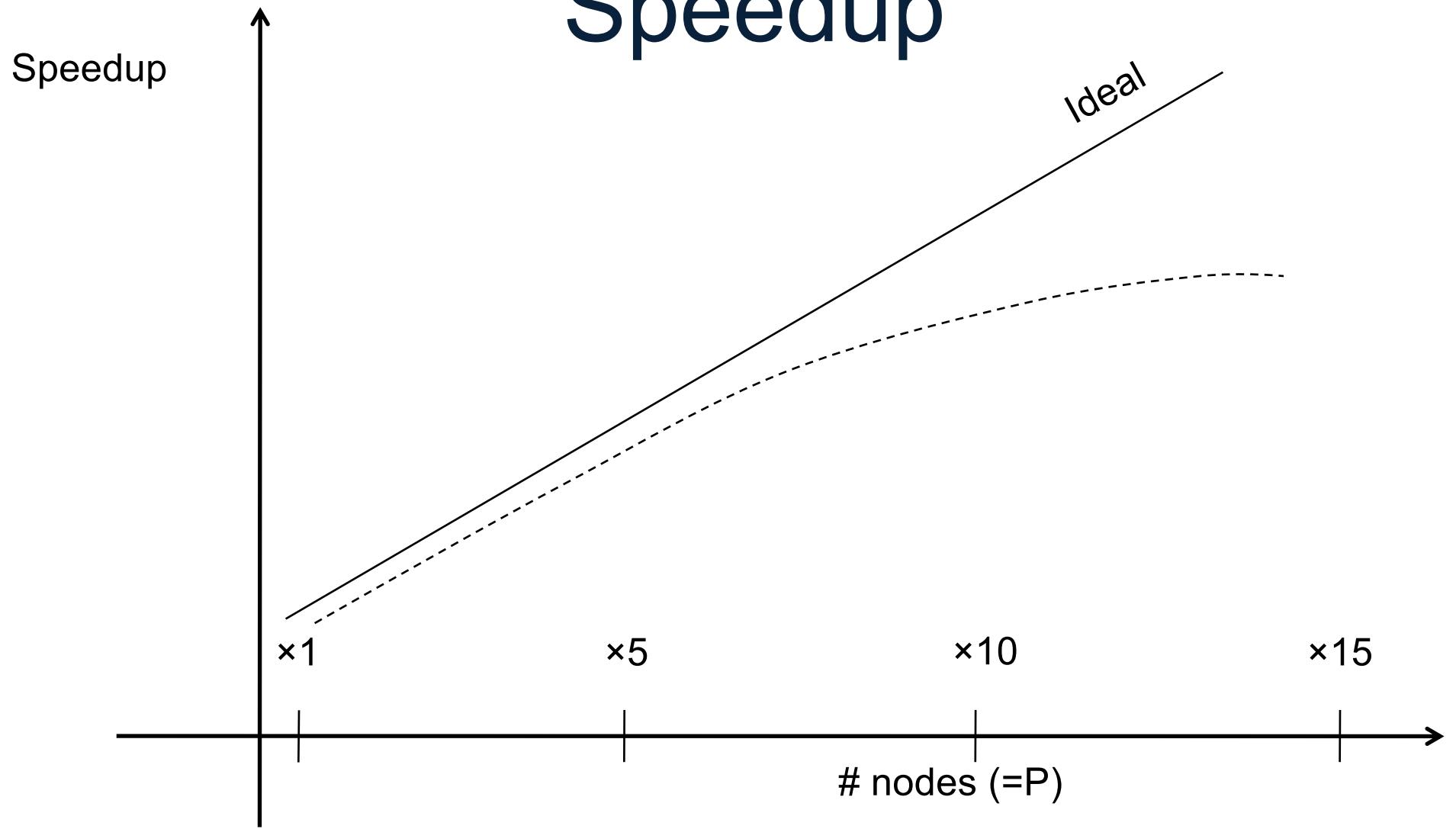
- Multi-cores:
  - Most processors have multiple cores
  - This trend will likely increase in the future
- Big data: too large to fit in main memory
  - Distributed query processing on 100x-1000x servers
  - Widely available now using cloud services
  - Recall HW3

# Performance Metrics for Parallel DBMSs

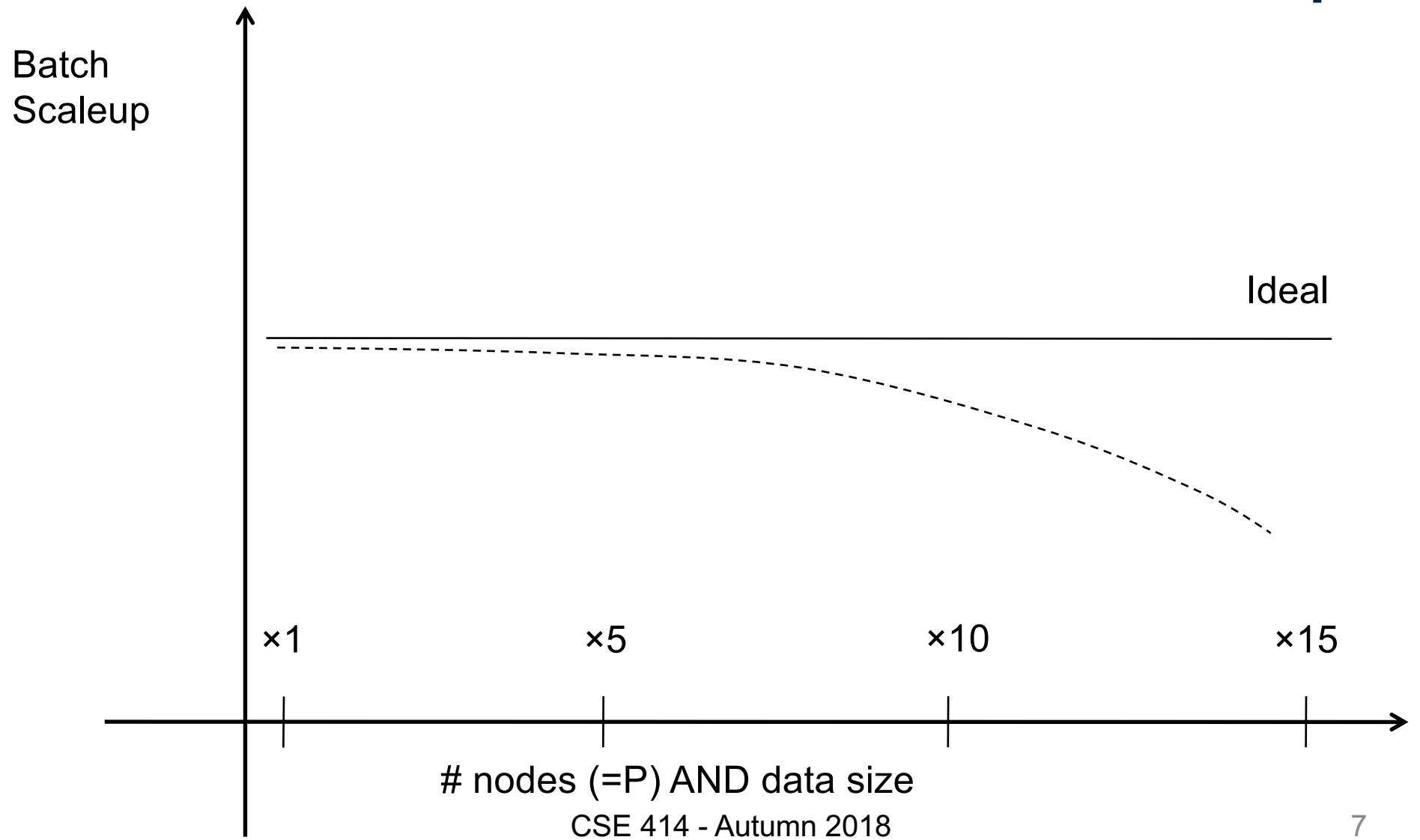
Nodes = processors, computers

- 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



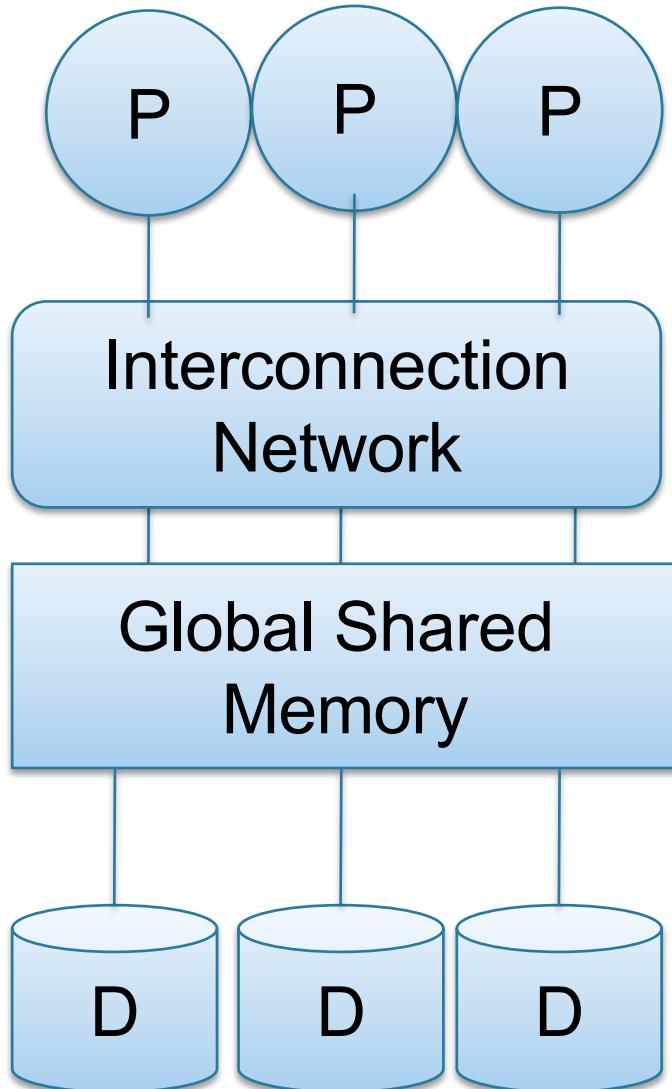
# Why Sub-linear Speedup and Scaleup?

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

# Architectures for Parallel Databases

- Shared memory
- Shared disk
- Shared nothing

# Shared Memory



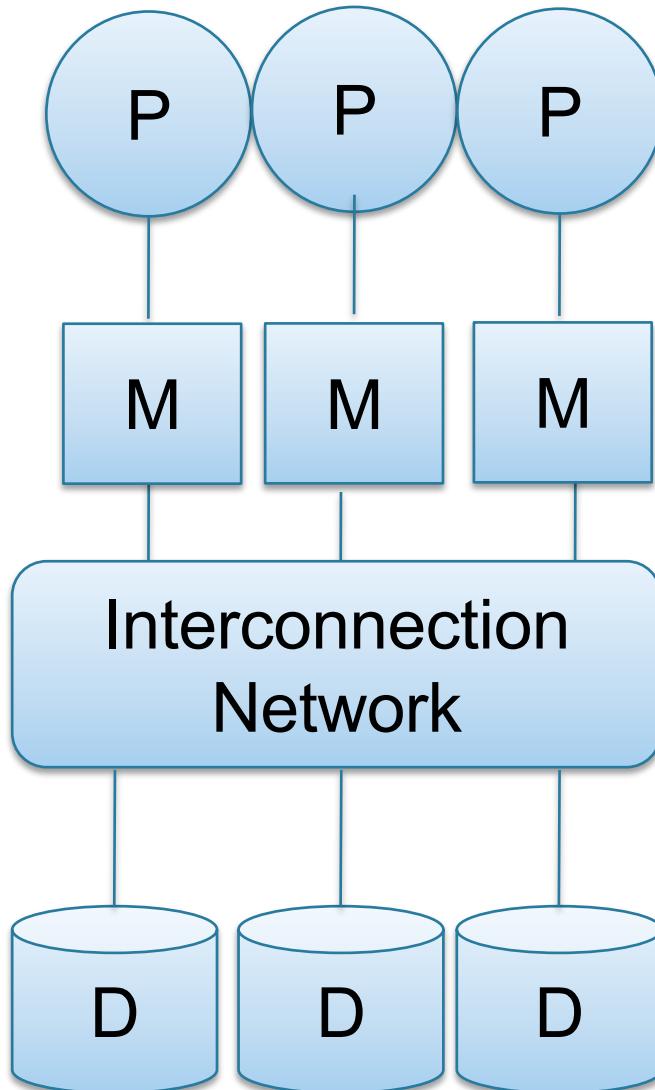
- Nodes share both RAM and disk
- Dozens to hundreds of processors

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
  - last remaining cash cows in the hardware industry

# Shared Disk

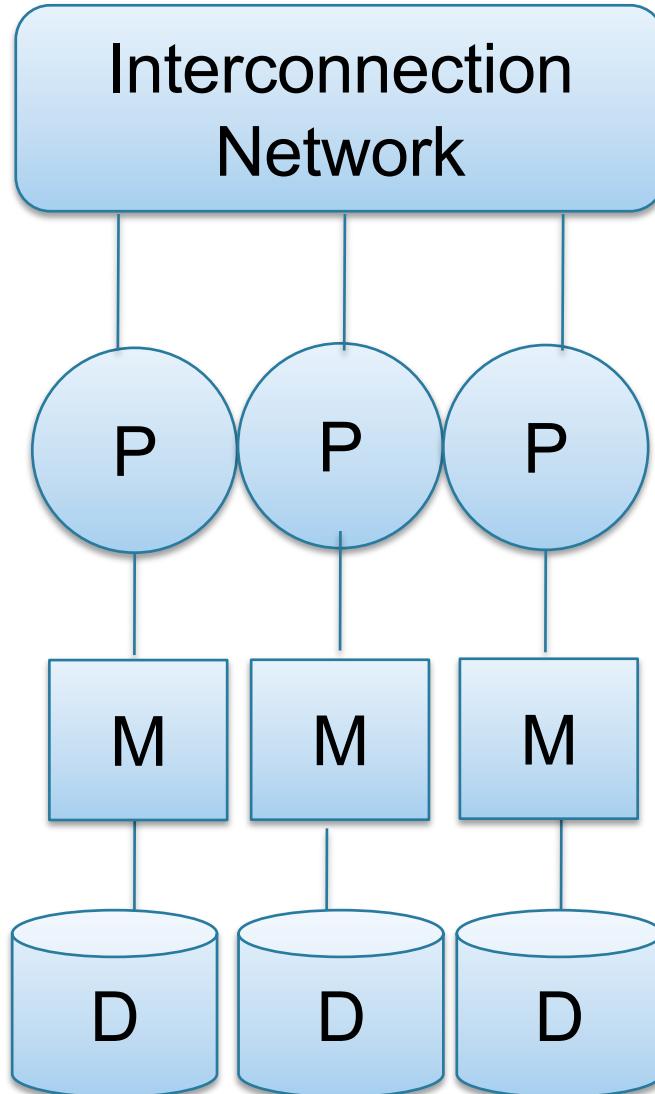


- All nodes access the same disks
- Found in the largest "single-box" (non-cluster) multiprocessors

Example: Oracle

- No need to worry about shared memory
- 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.

no competition over resources (i.e. disks or memory space)

Example: Google Mapreduce

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

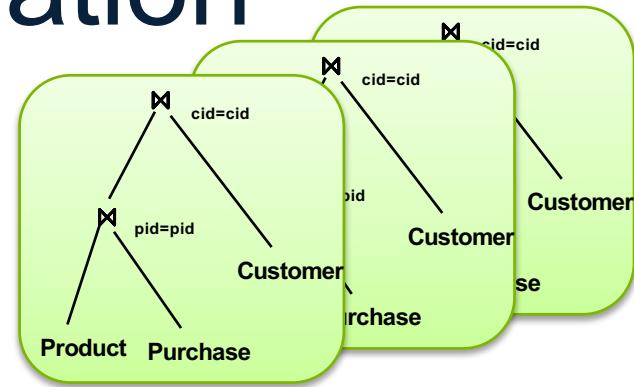
- Easy to maintain and scale
- Most difficult to administer and tune.

We discuss only Shared Nothing in class

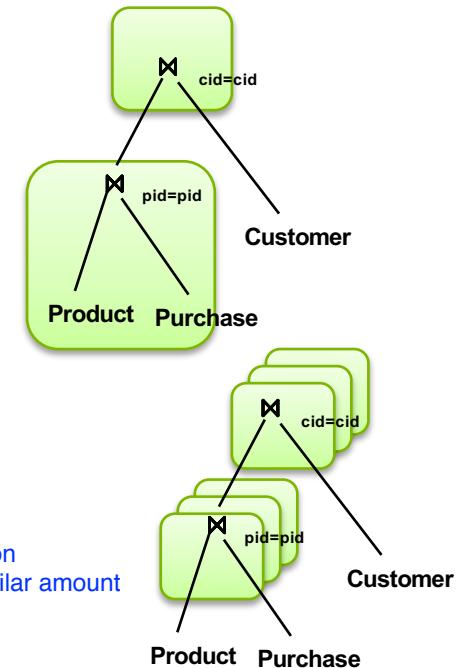
# Approaches to Parallel Query Evaluation

- **Inter-query parallelism**
  - Transaction per node
  - Good for transactional workloads
    - small update queries; very isolated queries.
- **Inter-operator parallelism**
  - Operator per node
  - Good for analytical workloads
- **Intra-operator parallelism**
  - Operator on multiple nodes
  - Good for both?

each machine runs full query on its own; as queries come in just divide them to one machine



one machine does an operator, then sends the data elsewhere



One operator run on ALL machines; spread computation evenly across machines assuming they each store similar amount of data/tuples

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:  $\gamma_{A,\text{sum}(B)}(R)$ 
  - 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

vertical partition: each machine stores subset of the attributes!

each machine only stores some subset of the tuples

- Data is “horizontally partitioned” on many servers
- Operators may require data reshuffling
- First let’s discuss how to distribute data across multiple nodes / servers

moving data/tuples between machines

# Horizontal Data Partitioning

Data:

ALL OF DATA

<u>K</u>	A	B
...	...	

Servers:

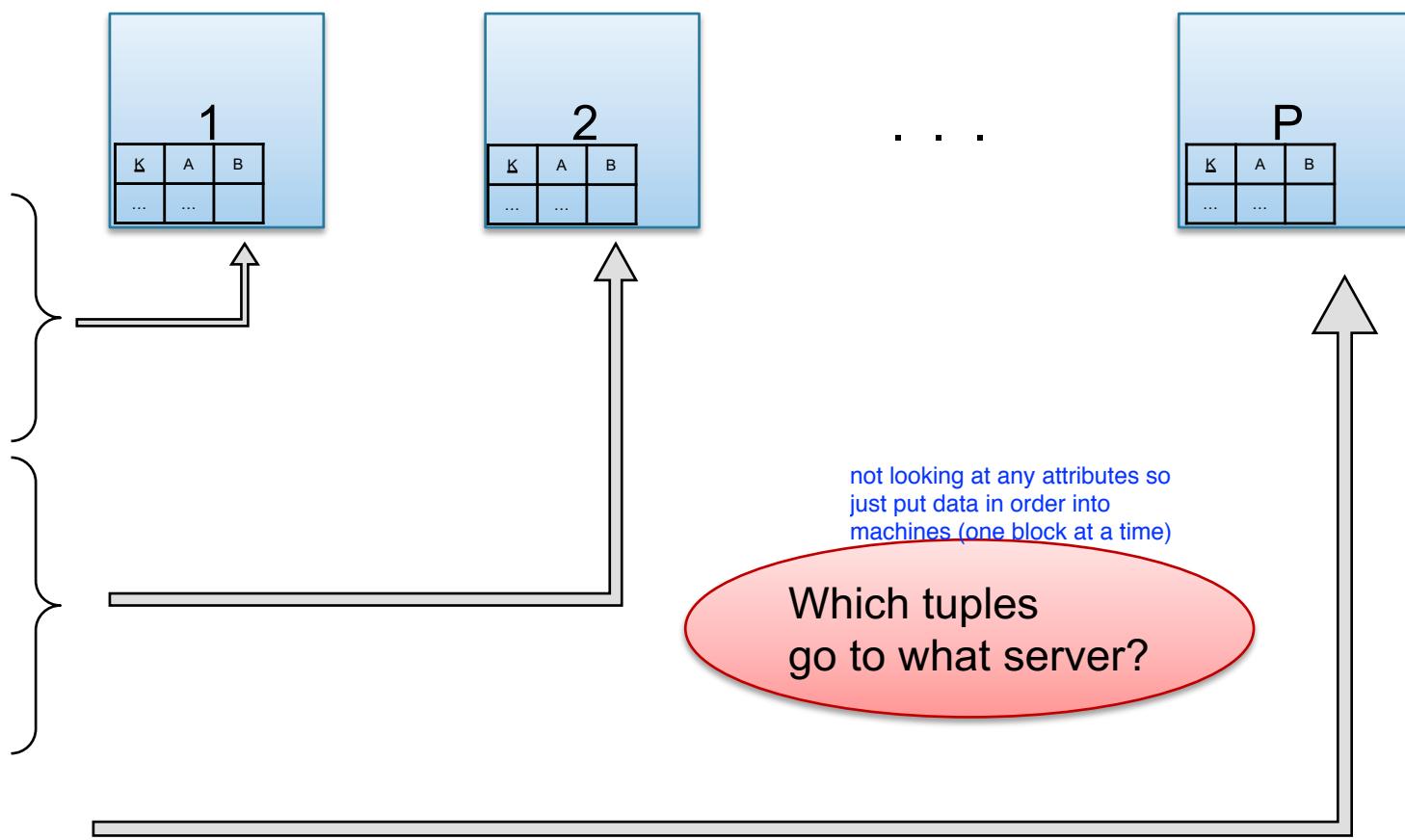


# Horizontal Data Partitioning

Data:

K	A	B
...	...	

Servers:



# Horizontal Data Partitioning

- ★• **Block Partition:** example above → just put tuples in order into machine one until its full, machine 2 until its full, etc...
  - Partition tuples arbitrarily s.t.  $\text{size}(R_1) \approx \dots \approx \text{size}(R_P)$
- ★• **Hash partitioned on attribute A:** choose a specific attribute of the tuples, and apply a hash function to that attribute for each tuple; the result determines which machine the tuple will go on to.
  - Tuple  $t$  goes to chunk  $i$ , where  $i = h(t.A) \bmod P + 1$
  - Recall: calling hash fn's is <sup>no slow down</sup> free in this class compared to disk read time
- **Range partitioned on attribute A:** choose an attribute of interest and sort tuples onto machine so that each machine contains a specific range of values of that attribute
  - Partition the range of  $A$  into  $-\infty = v_0 < v_1 < \dots < v_P = \infty$
  - 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?

i.e. some data attribute values might be grouped together rather than randomly distributed.

- Block partition**



- Hash-partition**

- On the key K
- On the attribute A

because the keys are unique!



Assuming good hash function

if the attribute A is skewed  
(i.e. consider all values of A are the same)



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

Keep this in mind in the next few slides

# Parallel Execution of RA Operators: Grouping

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

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

How to compute group by if:

- $R$  is hash-partitioned on  $A$  ?

As we are hash-partitioned on  $A$ , all tuples with same values of  $A$ . Are placed on same machine. Thus, each machine can easily run separately and group by values of  $A$  and then simply collect the result at the end.

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

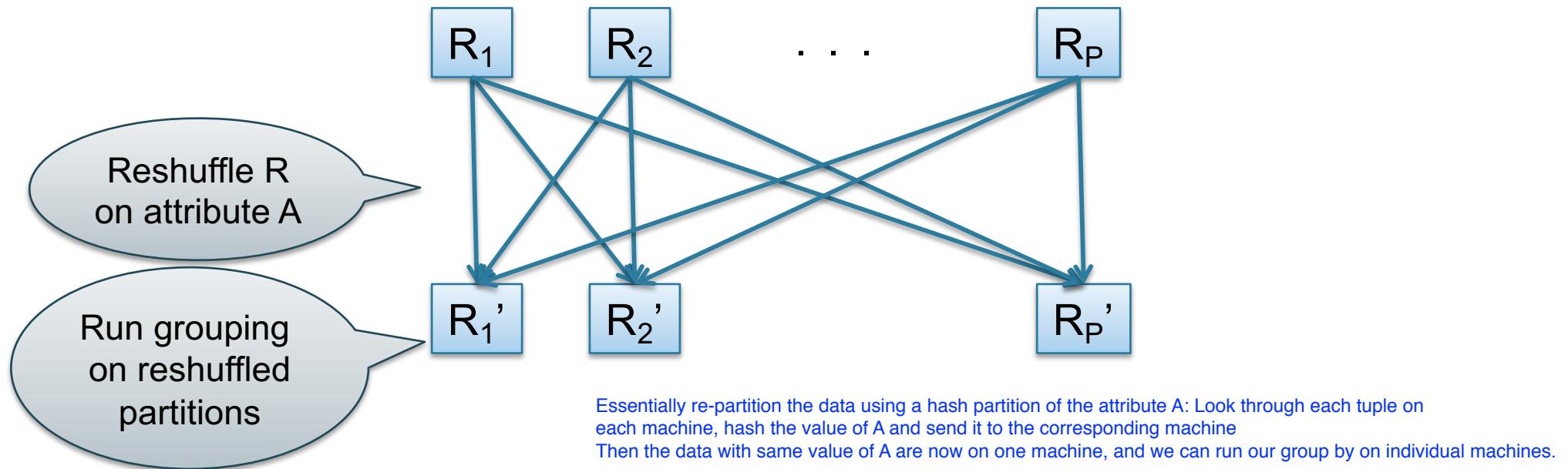
we have to do a data shuffle

# Parallel Execution of RA Operators: Grouping

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

**Query:**  $\text{V}_{A, \text{sum}(C)}(R)$

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



# Speedup and Scaleup

How does the algorithm above for group by perform as we change the number of nodes and/or data size?

- Consider:
  - Query:  $\gamma_{A,\text{sum}(C)}(R)$
  - Runtime: only consider I/O costs
- If we double the number of nodes  $P$ , what is the new running time?
  - Half (each server holds  $\frac{1}{2}$  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!

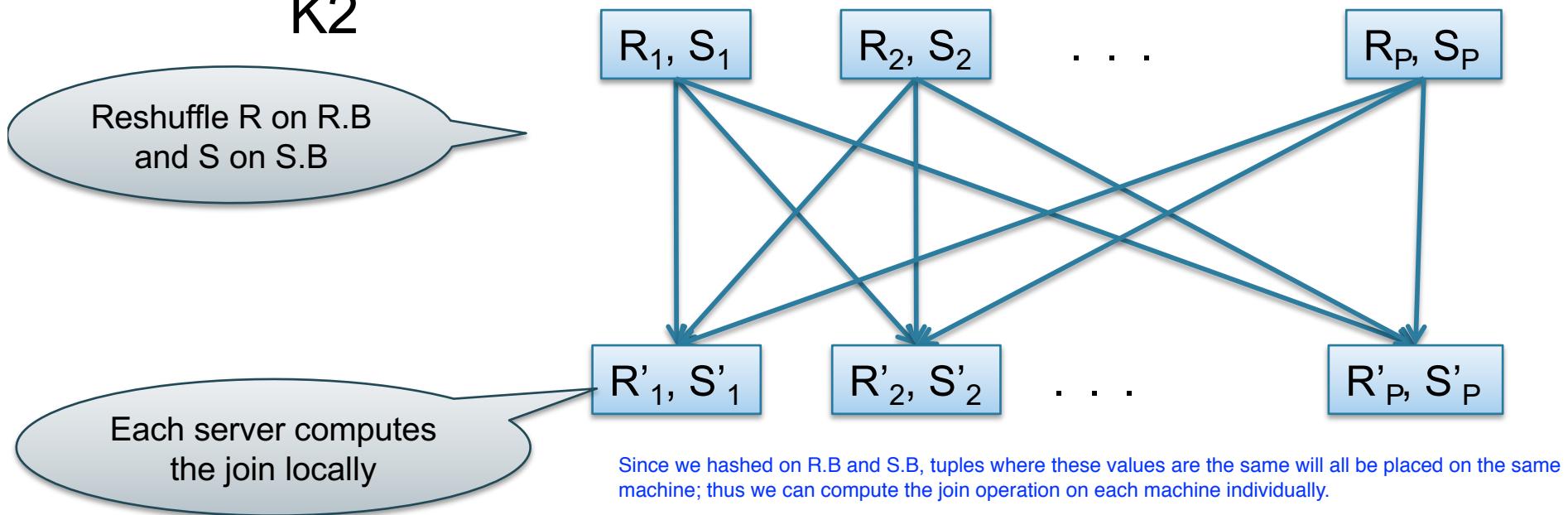
# Skewed Data

- $R(\underline{K}, A, B, C)$
- Informally: we say that the data is skewed if one server holds much more data than the average
  - (after a hash partition on attribute A)
- 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 Data Processing in the 20<sup>th</sup> Century

# Parallel Execution of RA Operators: Partitioned Hash-Join

- Data:  $R(\underline{K1}, A, B)$ ,  $S(\underline{K2}, B, C)$
- Query:  $R(\underline{K1}, A, \textcolor{red}{B}) \bowtie S(\underline{K2}, \textcolor{red}{B}, C)$ 
  - 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)  $\bowtie$  S(K2,B,C)

# Parallel Join Illustration

R and S are blocked partitioned onto our two machines

Partition

R1		S1		R2		S2	
K1	B	K2	B	K1	B	K2	B
1	20	101	50	3	20	201	20
2	50	102	50	4	20	202	50
Machine		M1		M2			

Shuffle on B

Local  
Join

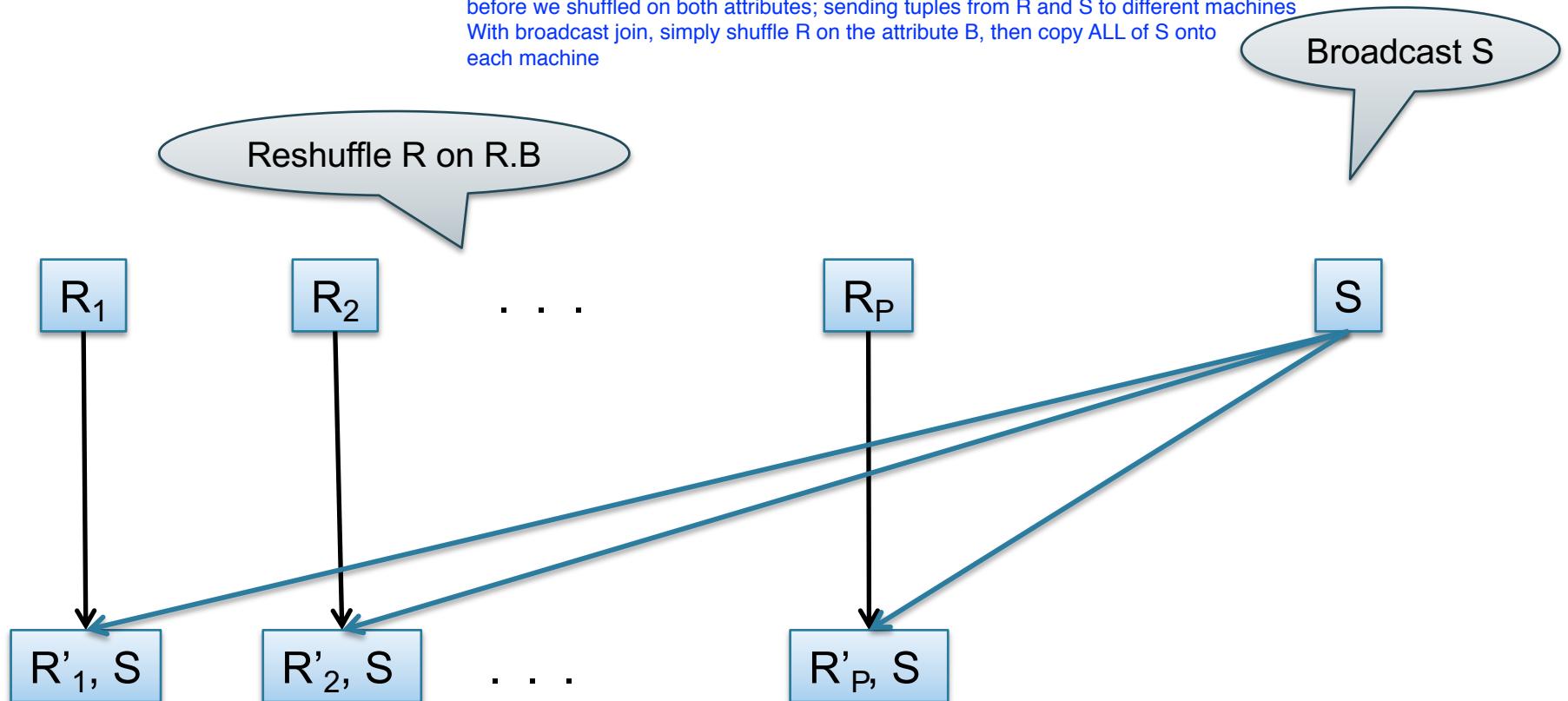
R1'		S1'		R2'		S2'	
K1	B	K2	B	K1	B	K2	B
1	20	201	20	2	50	101	50
3	20					102	50
4	20					202	50
M1		M2					

Data:  $R(A, B)$ ,  $S(C, D)$

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

## Broadcast Join

before we shuffled on both attributes; sending tuples from R and S to different machines  
With broadcast join, simply shuffle R on the attribute B, then copy ALL of S onto each machine



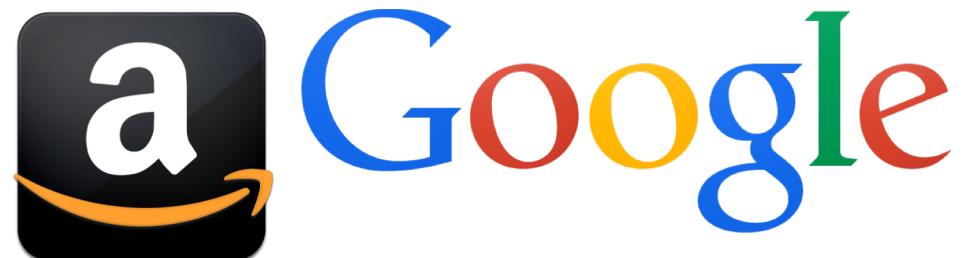
Why would you want to do this?

when S is a relatively small table so S fits  
easily in the memory of one machine



# Parallel Data Processing @ 2000

MAP reduce paradigm



# 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 take one piece of data and extract a piece of information about it you are interested in
- Shuffle and Sort
- **Reduce:** aggregate, summarize, filter, transform take multiple pieces of data from the map
- Write the results Paradigm stays the same, change map and reduce functions for different problems

# Data Model

Files!

Similar to our NoSQL (SQL++) paradigm

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

Sounds familiar after HW5?

A MapReduce program:

- Input: a bag of (**inputkey, value**) pairs
- Output: a bag of (**outputkey, value**) pairs
  - **outputkey** is optional

# Step 1: the MAP Phase

User provides the **MAP**-function:

- Input: (**input key, value**)  
i.e. our tuple
- Output: bag of (**intermediate key, value**)

write some java code for Hadoop

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**)

combine ALL values associated to a given word/key

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

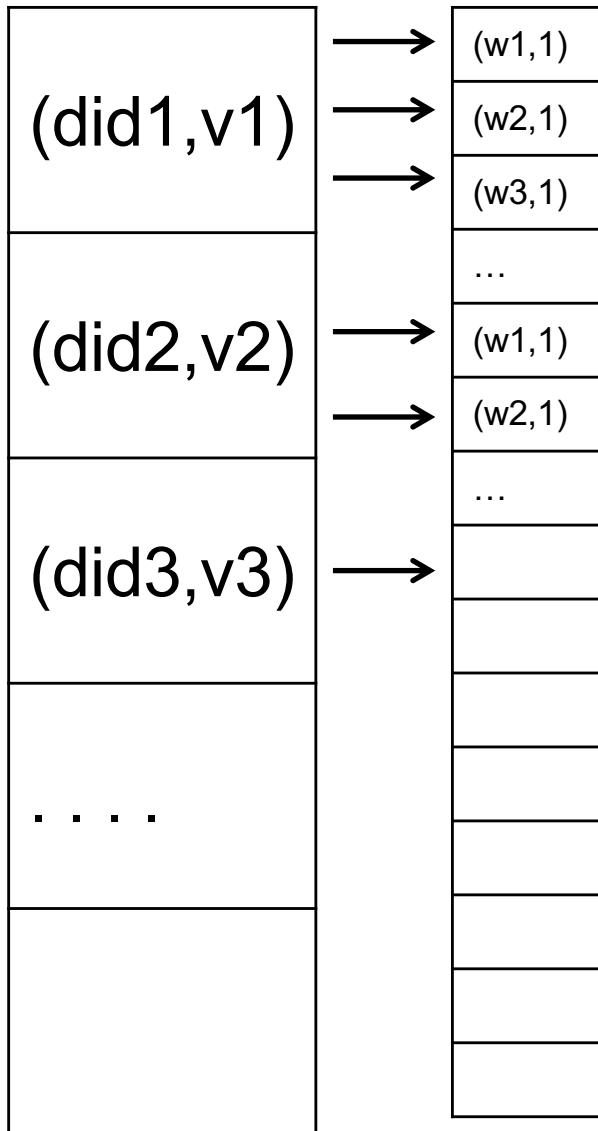
i.e. this word w appeared one time

newKey newValue

```
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

for each document  
create a key for  
each word, with value  
1

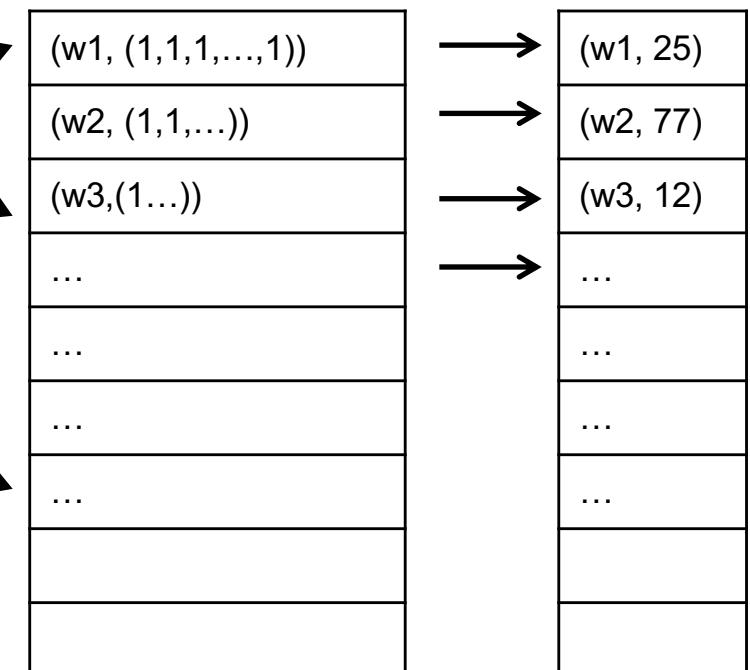


# REDUCE

Hash by key and combine  
values into a list/iterator

## Shuffle

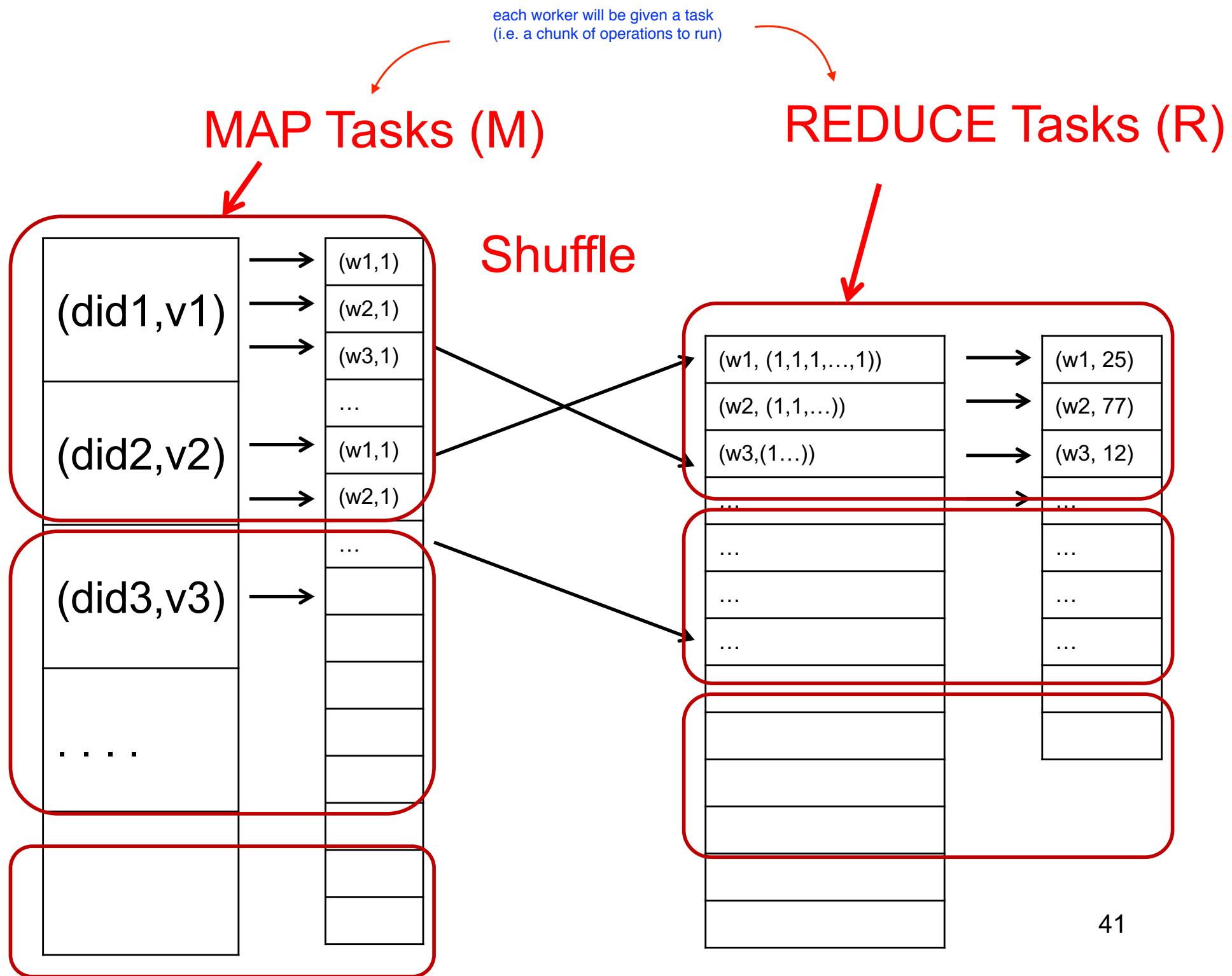
apply reduce  
function



# Workers

one processor

- 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
- instead of directly sending files to next machine, write to disk  
this way if a server fails, they can just read from disk wherever they left off

# Implementation

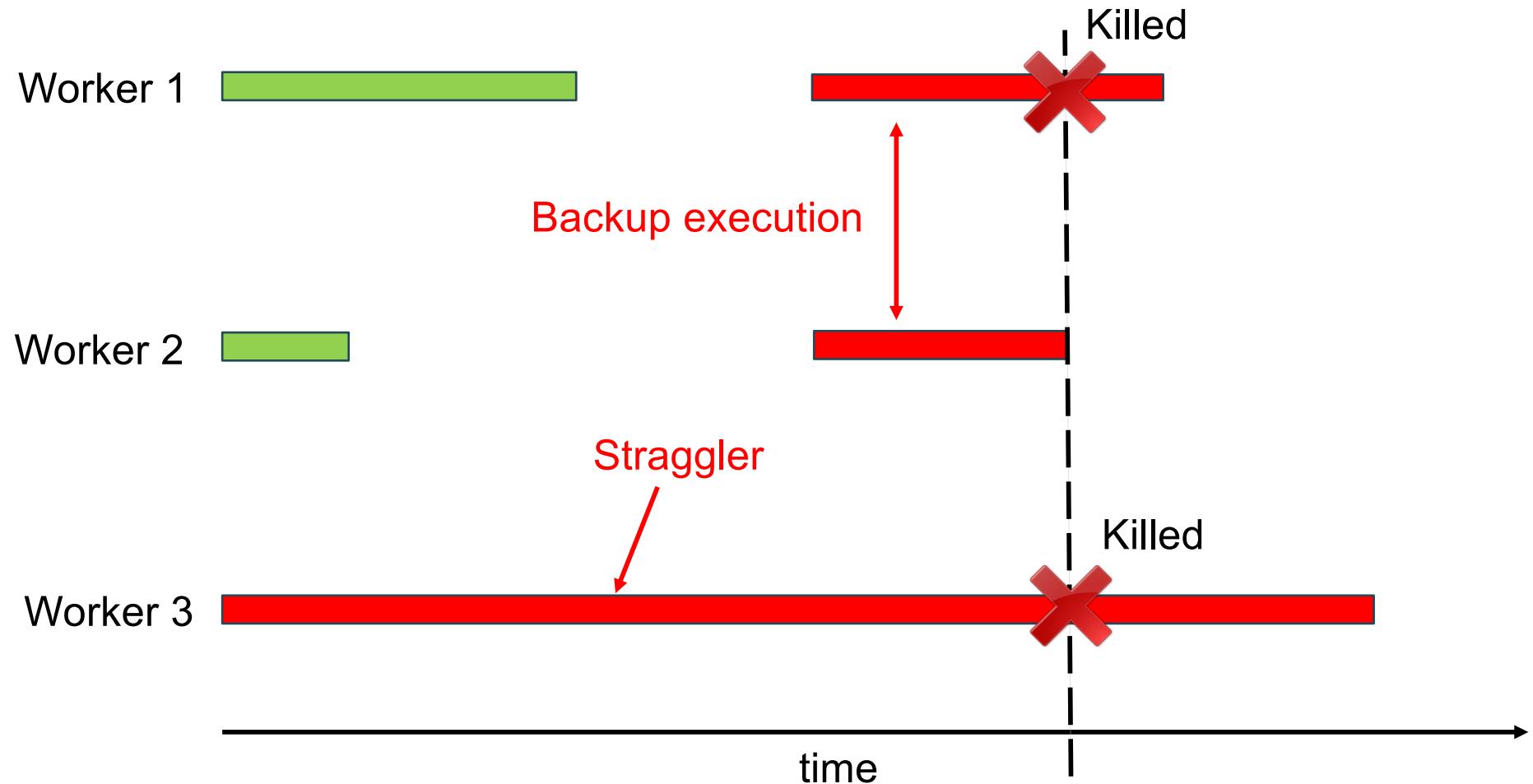
- There is one master node
- Master partitions input file into  $M$  splits, by key
- Master assigns *workers* (=servers) to the  $M$  map tasks, keeps track of their progress
- Workers write their output to local disk, partition into  $R$  regions
- Master assigns workers to the  $R$  reduce tasks
-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

# Interesting Implementation Details

## 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



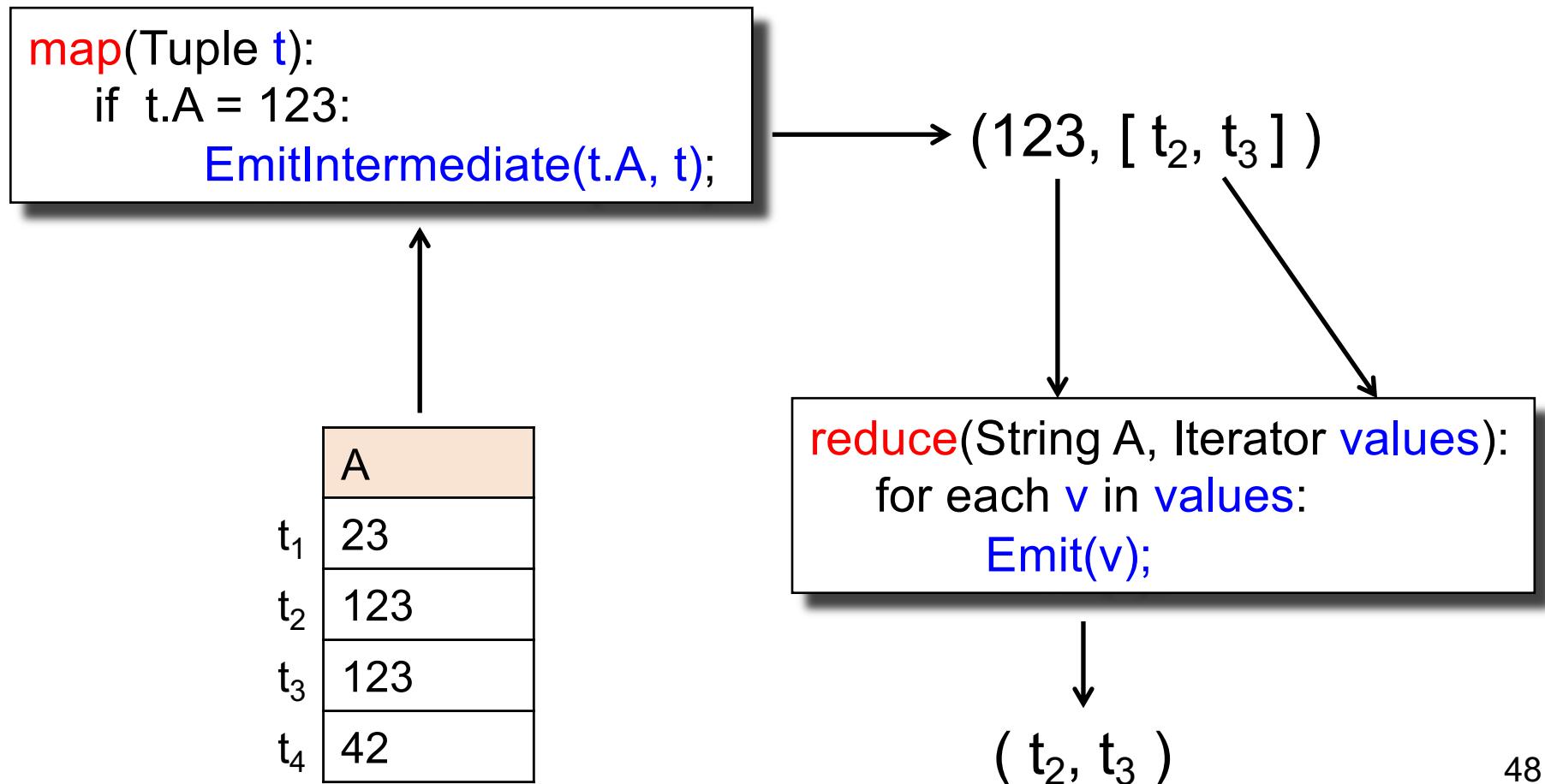
# Using MapReduce in Practice: Implementing RA Operators in MR

# Relational Operators in MapReduce

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

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

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



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

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

~~reduce(String A, 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 $\gamma_{A,\text{sum}(B)}(R)$

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

(23, [ t<sub>1</sub> ] )  
→ (42, [ t<sub>4</sub> ] )  
(123, [ t<sub>2</sub>, t<sub>3</sub> ] )

	A	B
t <sub>1</sub>	23	10
t <sub>2</sub>	123	21
t <sub>3</sub>	123	4
t <sub>4</sub>	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) <sup>50</sup>

# Join

Two simple parallel join algorithms:

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

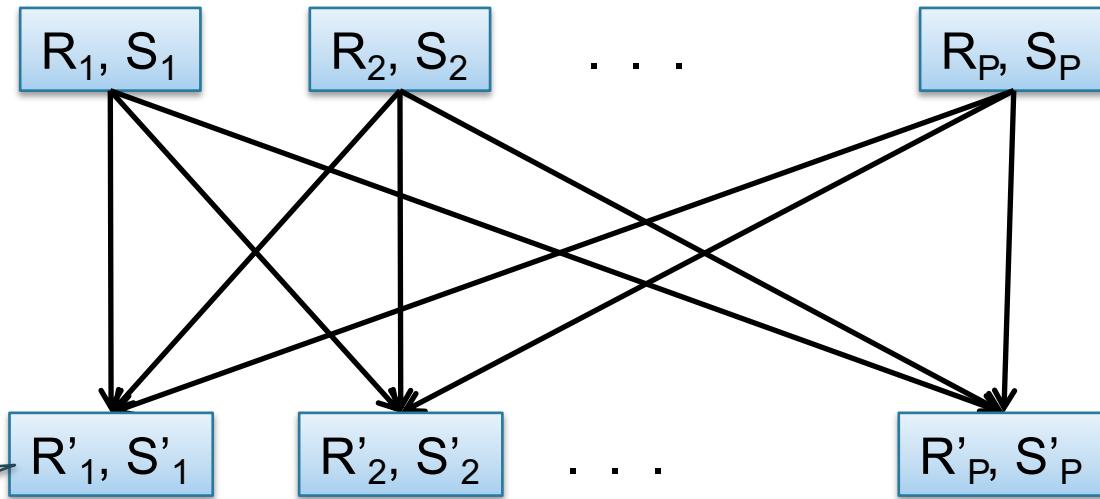
$$R(A,B) \bowtie_{B=C} S(C,D)$$

# Partitioned Hash-Join

Initially, both R and S are horizontally partitioned

Reshuffle R on R.B  
and S on S.B

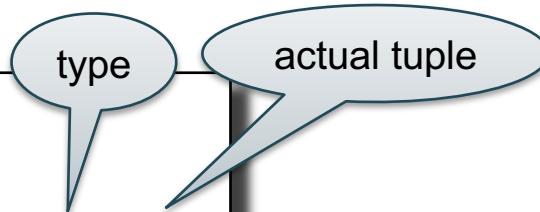
Each server computes  
the join locally



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

# Partitioned Hash-Join

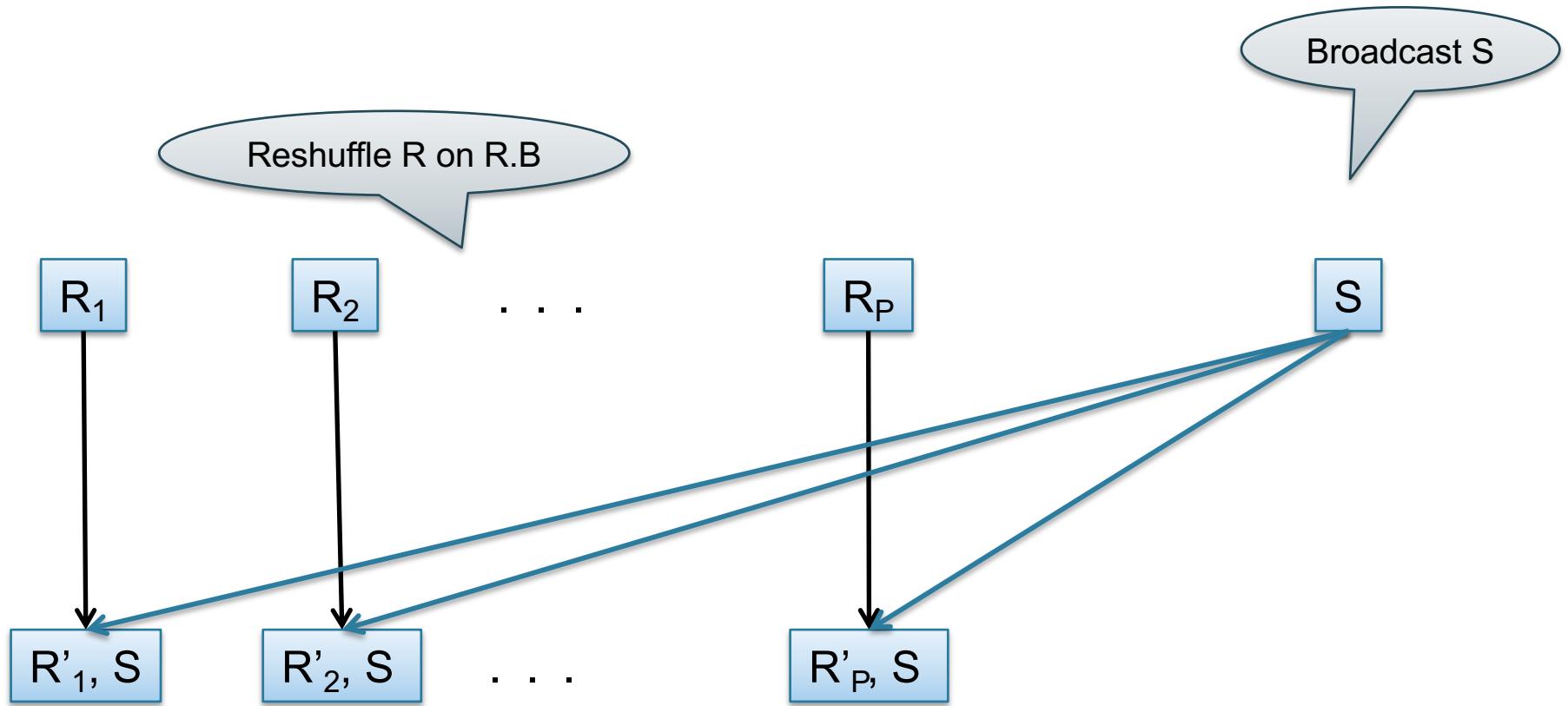
```
map(Tuple t):
    case t.relationName of
        'R': EmitIntermediate(t.B, ('R', t));
        'S': EmitIntermediate(t.C, ('S', t));
```



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

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

# Broadcast Join



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

# Broadcast Join

```
map(String value):
```

```
    readFromNetwork(S); /* over the network */
```

```
    hashTable = new HashTable()
```

```
    for each w in S:
```

```
        hashTable.insert(w.C, w)
```

```
for each v in value:
```

```
    for each w in hashTable.find(v.B)
```

```
        Emit(v,w);
```

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

Read entire table S,  
build a Hash Table

reduce(...):  
/\* empty: map-side only \*/

# HW6

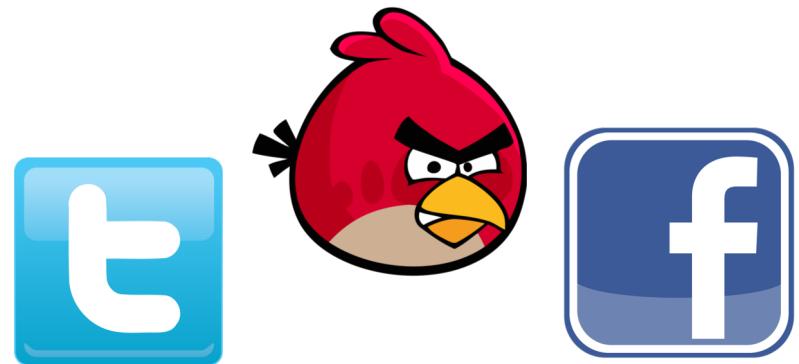
- HW6 will ask you to write SQL queries and MapReduce tasks using Spark
- You will get to “implement” SQL using MapReduce tasks
  - Can you beat Spark’s implementation?

# Spark

## A Case Study of the MapReduce Programming Paradigm



# 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

# Spark

- Open source system from UC 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

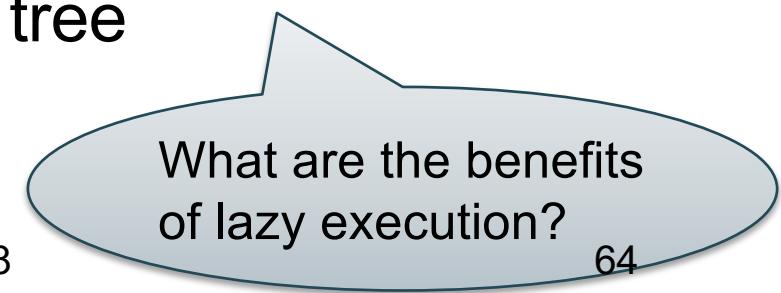
- Spark supports interfaces in Java, Scala, and Python
  - Scala: extension of Java with functions/closures
- We will illustrate use the Spark Java interface in this class
- Spark also supports a SQL interface (SparkSQL), and compiles SQL to its native Java interface

# Resilient Distributed Datasets

- RDD = Resilient Distributed Datasets
  - A distributed, immutable 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

# 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



What are the benefits  
of lazy execution?

64

# The RDD Interface

# Collections in Spark

- $\text{RDD}\langle T \rangle$  = an RDD collection of type T
  - Partitioned, recoverable (through lineage), not nested
- $\text{Seq}\langle T \rangle$  = a sequence
  - Local to a server, may be nested

# 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”

lines, errors, sqerrors  
have type JavaRDD<String>

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

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

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

sqerrors = errors.filter(l -> l.contains("sqlite"));

sqerrors.collect();
```

# Example

Given a large log file hdfs://logfile.log

retrieve all lines that:

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

lines, errors, sqlerrors  
have 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();
```

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

# MapReduce Again...

Steps in Spark resemble MapReduce:

- `col.filter(p)` applies in parallel the predicate  $p$  to all elements  $x$  of the partitioned collection, and returns collection with those  $x$  where  $p(x) = \text{true}$
- `col.map(f)` applies in parallel the function  $f$  to all elements  $x$  of the partitioned collection, and returns a new partitioned collection

# 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

RDD:

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

hdfs://logfile.log

filter(...startsWith("ERROR"))  
filter(...contains("sqlite"))

result

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();
```

hdfs://logfile.log

filter(...startsWith("ERROR"))  
filter(...contains("sqlite"))

result

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();           New RDD
sqlerrors = errors.filter(l->l.contains("sqlite"));
sqlerrors.collect()
```

Spark can recompute the result from errors

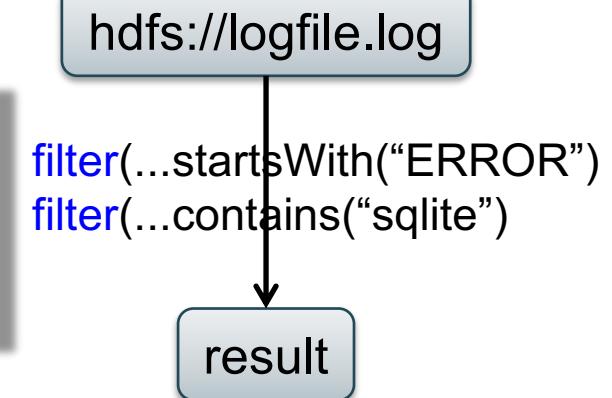
CSE 414 - Autumn 2018

75

# Persistence

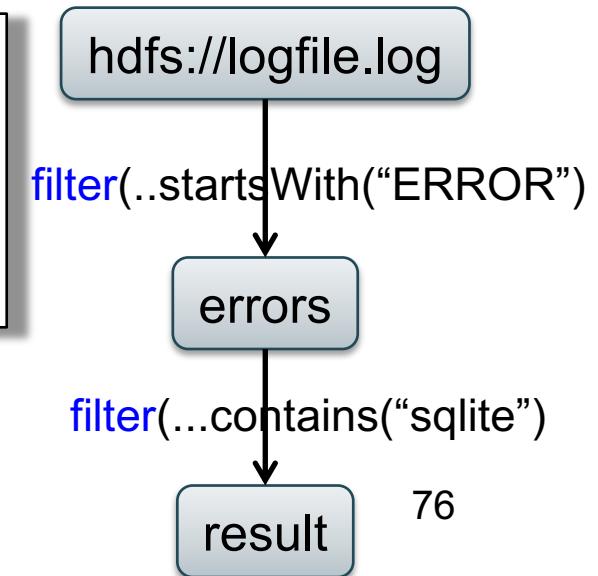
RDD:

```
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();           New RDD
sqlerrors = errors.filter(l->l.contains("sqlite"));
sqlerrors.collect()
```



Spark can recompute the result from errors

CSE 414 - Autumn 2018

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

Parses each line into an object

persisting on disk

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

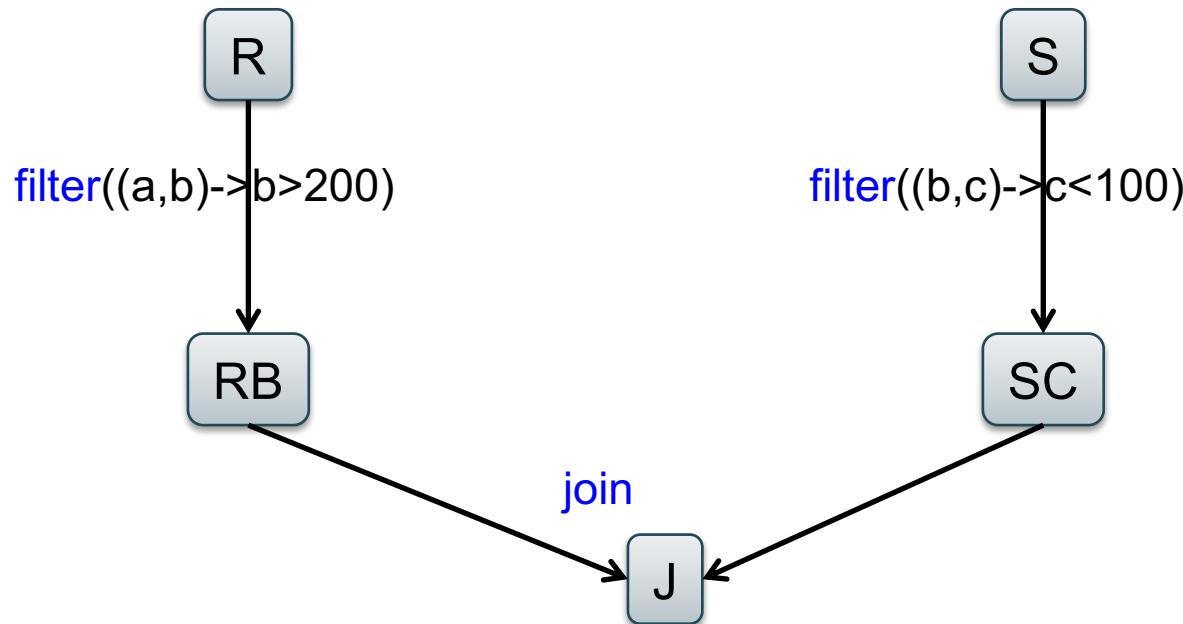
SELECT count(\*) FROM R, S  
WHERE R.B > 200 and S.C < 100 and R.A = S.A

## Example

```
R = s.read().textFile("R.csv").map(parseRecord).persist();  
S = s.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**
- $\text{RDD} < T >$  = an RDD collection of type T
  - Partitioned, recoverable (through lineage), not nested
- $\text{Seq} < T >$  = a sequence
  - Local to a server, may be nested

## Transformations:

<code>map(f : T -&gt; U):</code>	$\text{RDD}(T) \rightarrow \text{RDD}(U)$
<code>flatMap(f: T -&gt; Seq(U)):</code>	$\text{RDD}(T) \rightarrow \text{RDD}(U)$
<code>filter(f:T-&gt;Bool):</code>	$\text{RDD}(T) \rightarrow \text{RDD}(T)$
<code>groupByKey():</code>	$\text{RDD}((K,V)) \rightarrow \text{RDD}((K,\text{Seq}[V]))$
<code>reduceByKey(F:(V,V)-&gt; V):</code>	$\text{RDD}((K,V)) \rightarrow \text{RDD}((K,V))$
<code>union():</code>	$(\text{RDD}(T), \text{RDD}(T)) \rightarrow \text{RDD}(T)$
<code>join():</code>	$(\text{RDD}((K,V)), \text{RDD}((K,W))) \rightarrow \text{RDD}((K,(V,W)))$
<code>cogroup():</code>	$(\text{RDD}((K,V)), \text{RDD}((K,W))) \rightarrow \text{RDD}((K,(\text{Seq}[V],\text{Seq}[W])))$
<code>crossProduct():</code>	$(\text{RDD}(T), \text{RDD}(U)) \rightarrow \text{RDD}((T,U))$

## Actions:

<code>count():</code>	$\text{RDD}(T) \rightarrow \text{Long}$
<code>collect():</code>	$\text{RDD}(T) \rightarrow \text{Seq}(T)$
<code>reduce(f:(T,T)-&gt;T):</code>	$\text{RDD}(T) \rightarrow T$
<code>save(path:String):</code>	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?

# Conclusions

- Parallel databases
  - Predefined relational operators
  - Optimization
  - Transactions
- MapReduce
  - User-defined map and reduce functions
  - Must implement/optimize manually relational ops
  - No updates/transactions
- Spark
  - Predefined relational operators
  - Must optimize manually
  - No updates/transactions