

Introduction to Database Systems

CSE 414

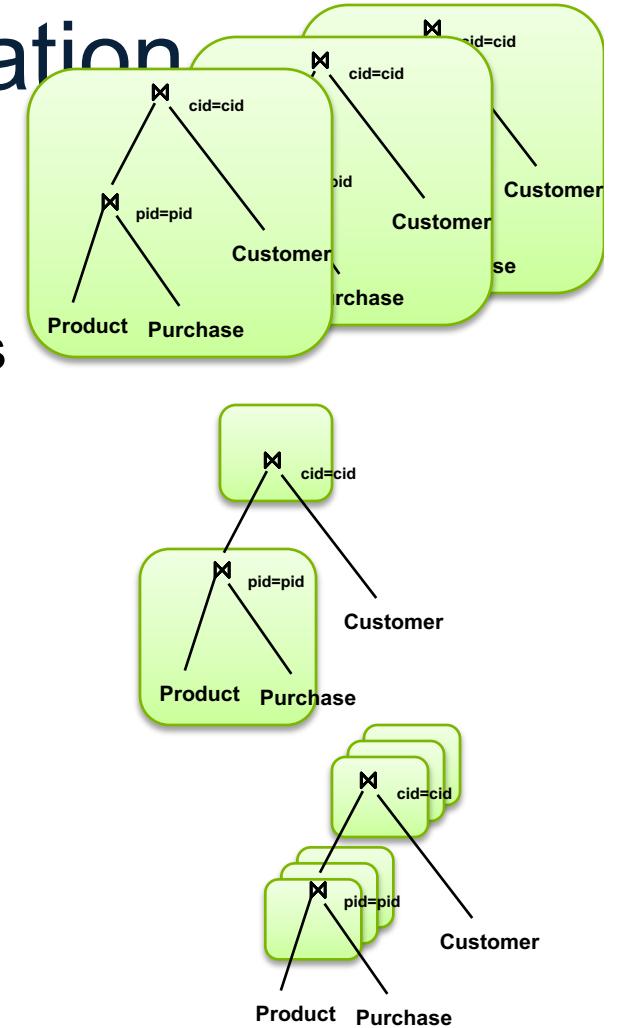
Lecture 18: MapReduce and Spark

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/>
- Extended office hours Friday to help with first parts of HW 6
 - 11:30 to 5:00pm in CSE 023

Approaches to Parallel Query Evaluation

- **Inter-query parallelism**
 - One query per node
 - Good for transactional (OLTP) workloads
- **Inter-operator parallelism**
 - Operator per node
 - Good for analytical (OLAP) workloads
- **Intra-operator parallelism**
 - Operator on multiple nodes
 - Good for both?

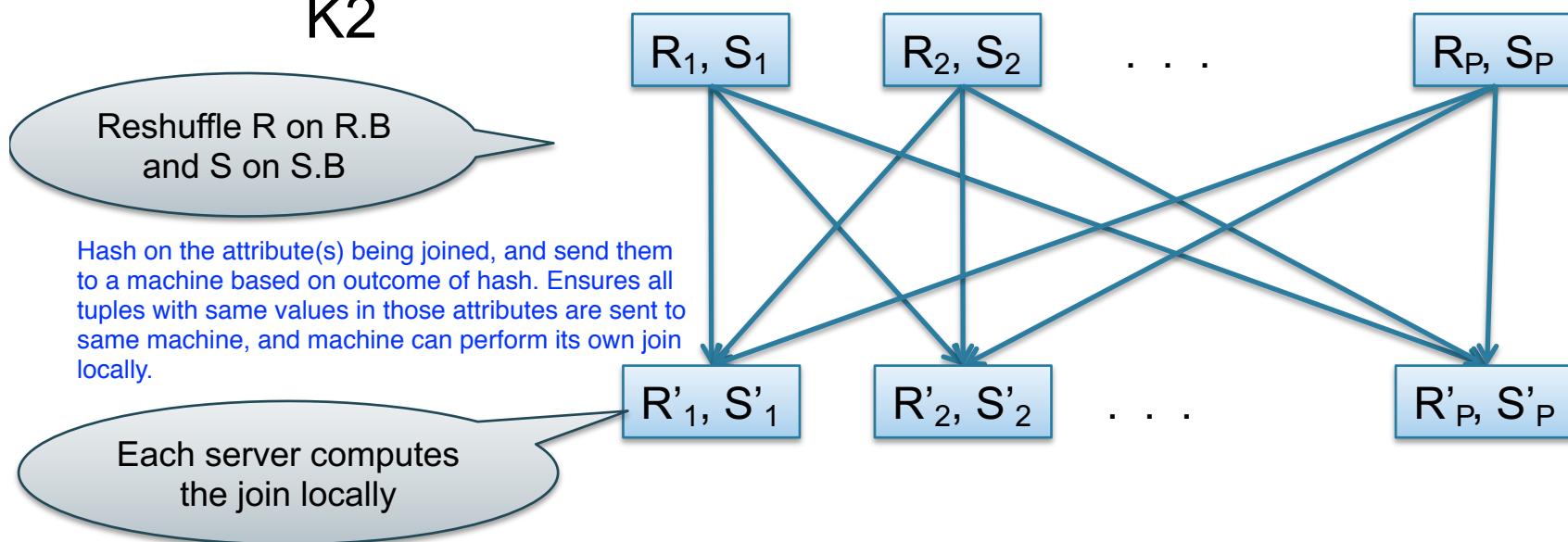


We study only intra-operator parallelism: most scalable

Parallel Data Processing in the 20th Century

Parallel Execution of RA Operators: Partitioned Hash-Join

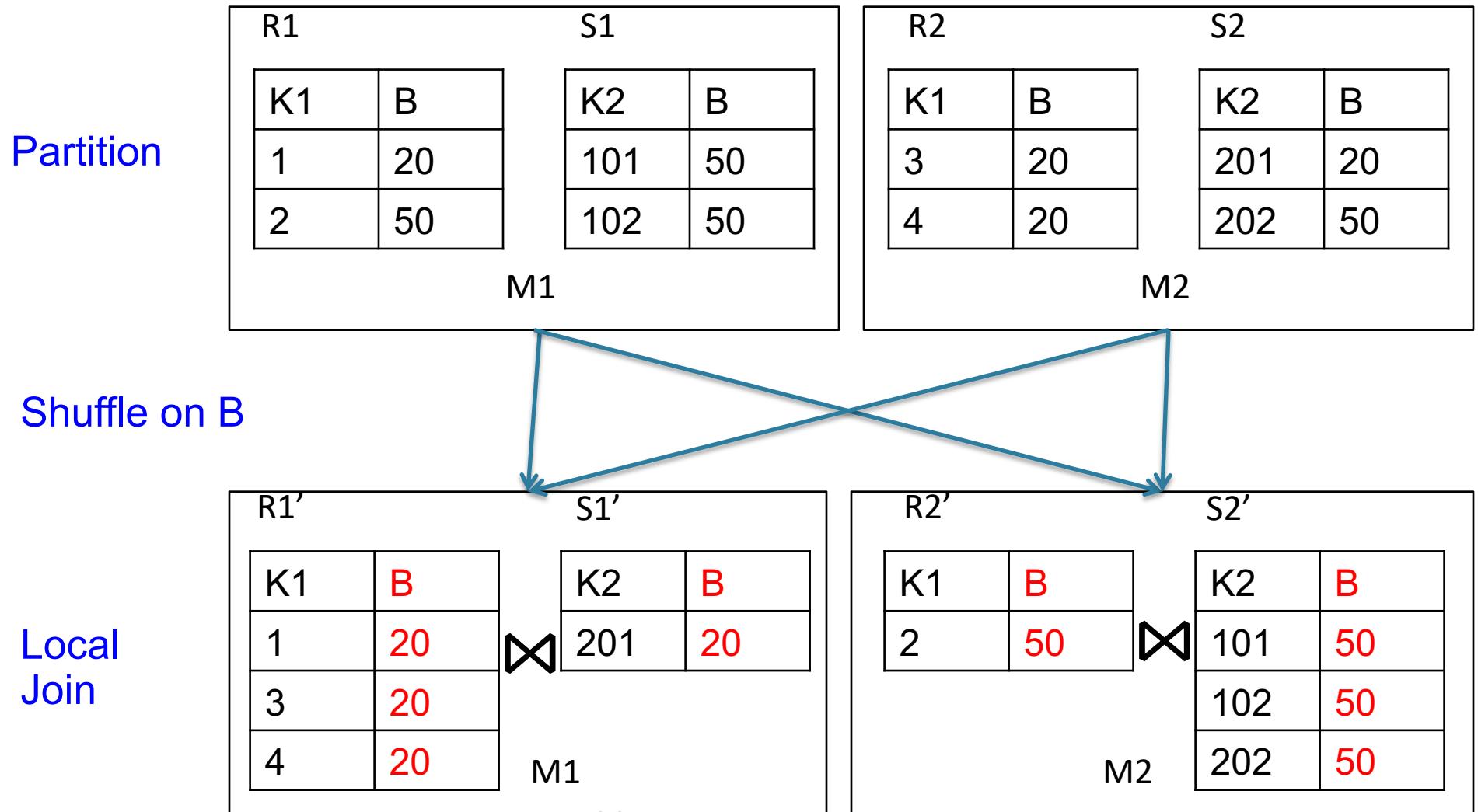
- **Data:** $R(\underline{K1}, A, B)$, $S(\underline{K2}, B, C)$
- **Query:** $R(\underline{K1}, A, B) \bowtie S(\underline{K2}, B, C)$
 - Initially, both R and S are partitioned on K1 and K2



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

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

Parallel Join Illustration

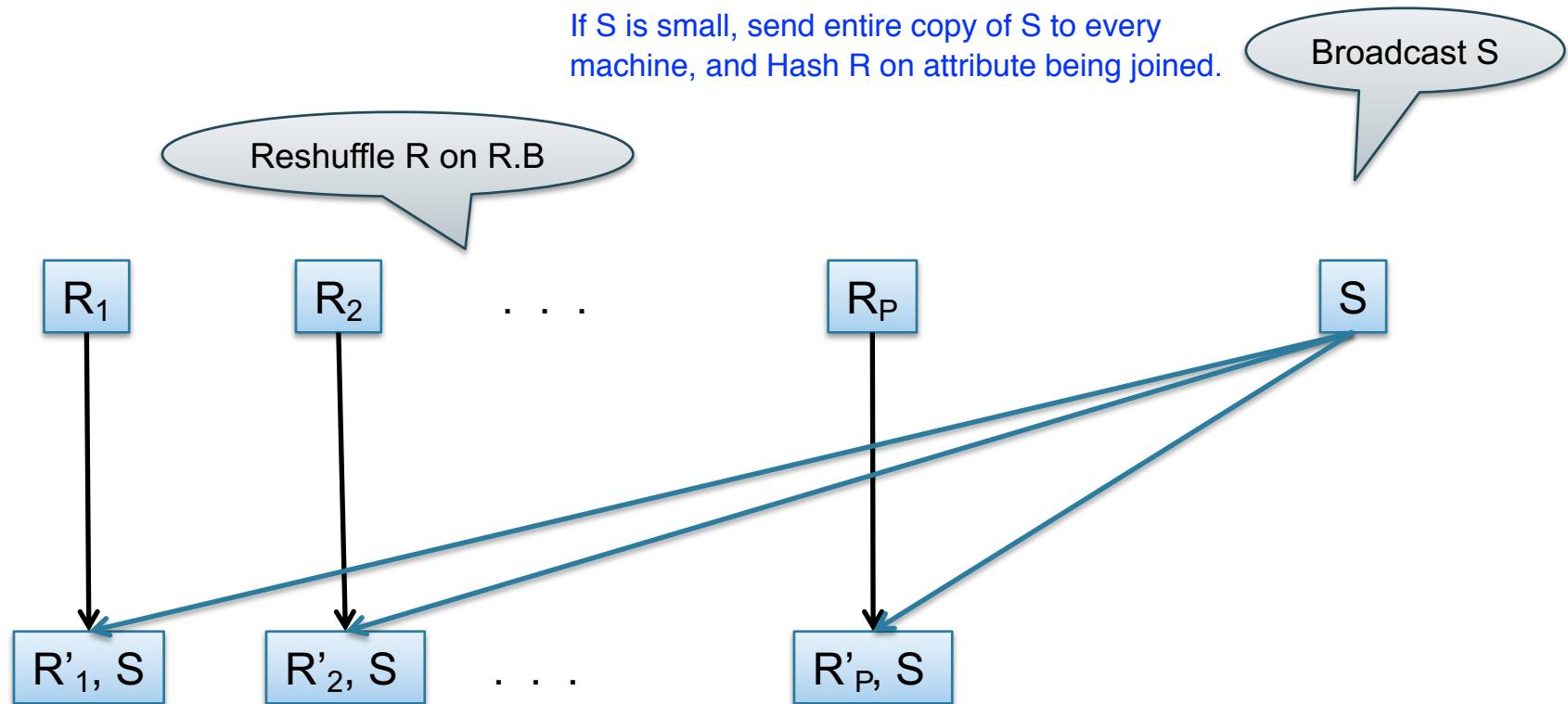


Data: $R(A, B), S(C, D)$

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

Broadcast Join

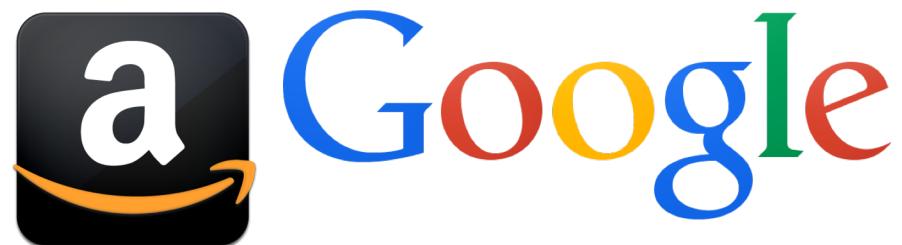
If S is small, send entire copy of S to every machine, and Hash R on attribute being joined.



Why would you want to do this?



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 (≥ 3), on different racks, for fault tolerance
- Implementations:
 - Google's DFS: **GFS**, proprietary
 - Hadoop's DFS: **HDFS**, open source

MapReduce

- Google: paper published 2004
- Free variant: Hadoop
- MapReduce = high-level programming model and implementation for large-scale parallel data processing

Typical Problems Solved by MR

- Read a lot of data
 - **Map**: extract something you care about from each record
 - Shuffle and Sort
 - **Reduce**: aggregate, summarize, filter, transform
 - Write the results
- Paradigm stays the same,
change map and reduce
functions for different problems

Data Model

Files!

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

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)`
- Output: 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

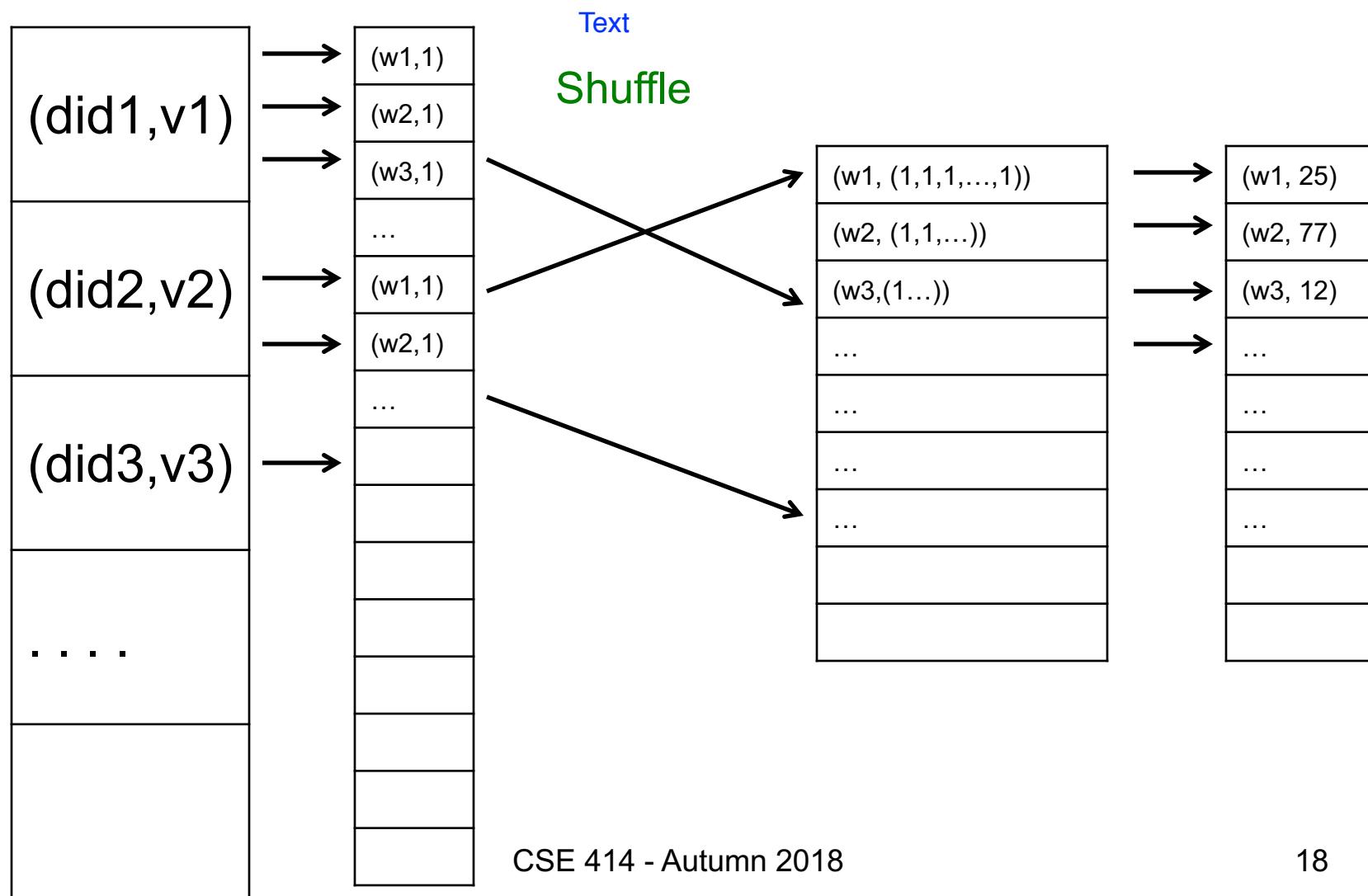
- Counting the number of occurrences of each word in a large collection of documents
- Each Document
 - The **key** = document id (**did**)
 - The **value** = set of words (**word**)

```
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



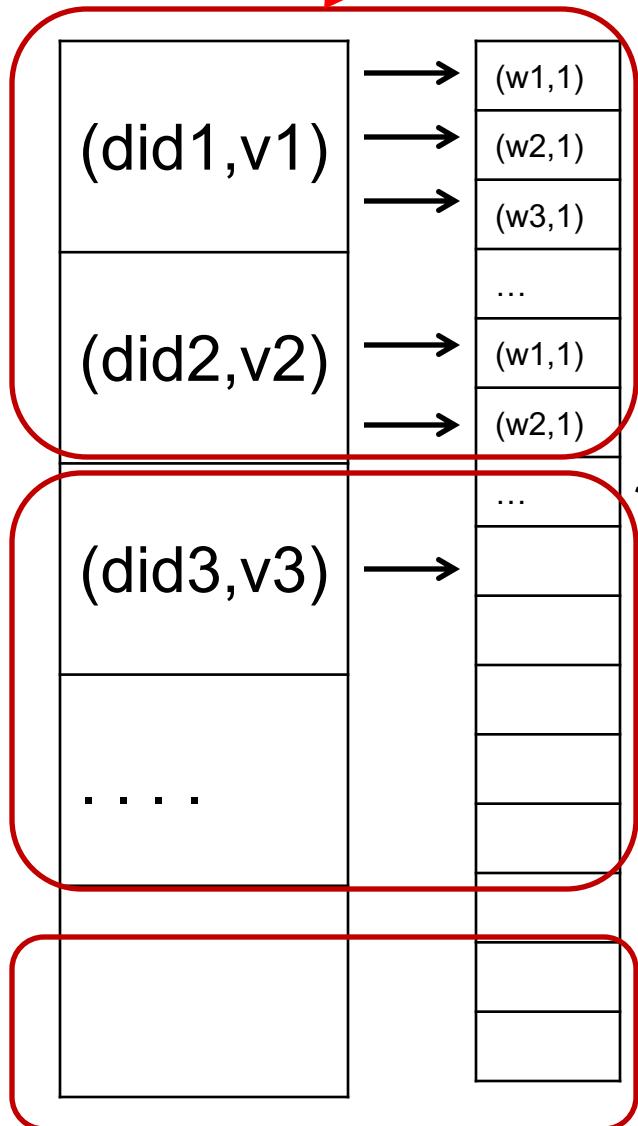
Workers

- 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

map each tuple to some key-value pair

MAP Tasks (M)

separate processors
handle each of these tasks

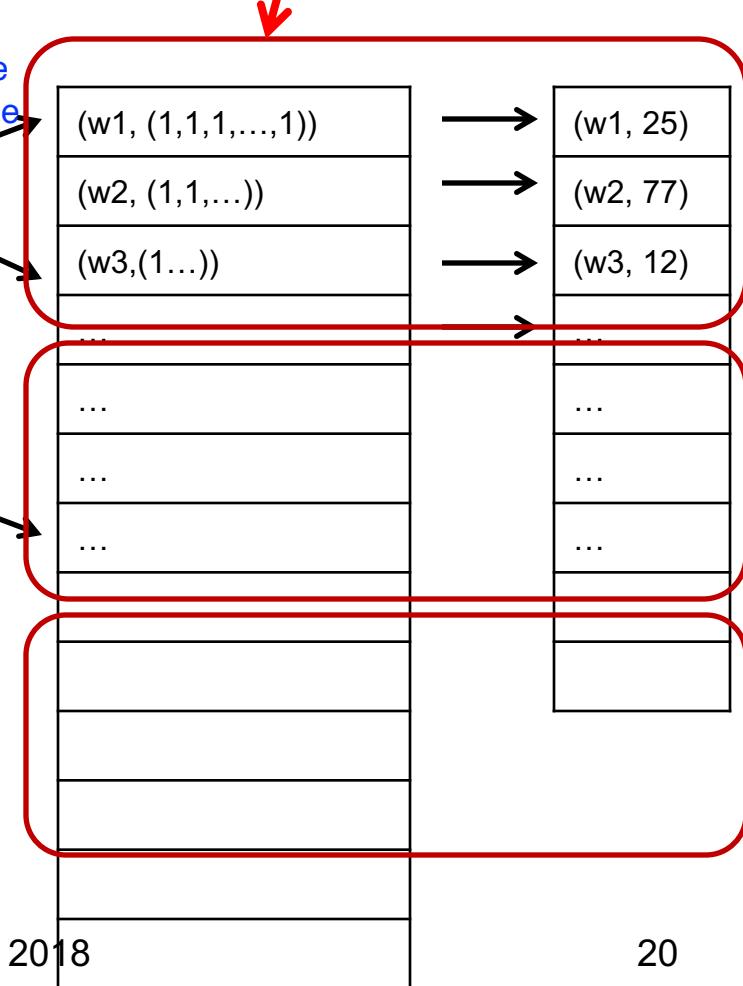
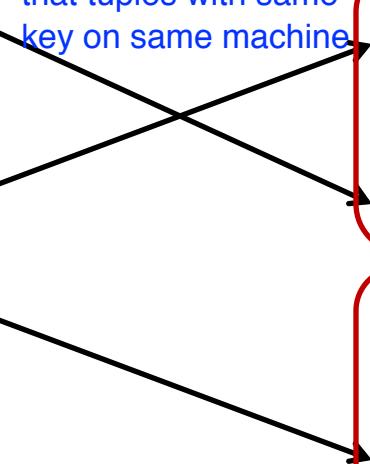


REDUCE Tasks (R)

combine values from matching keys,
and perform some sort of aggregation
on them

Shuffle

hash join etc, so
that tuples with same
key on same machine



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

Implementation

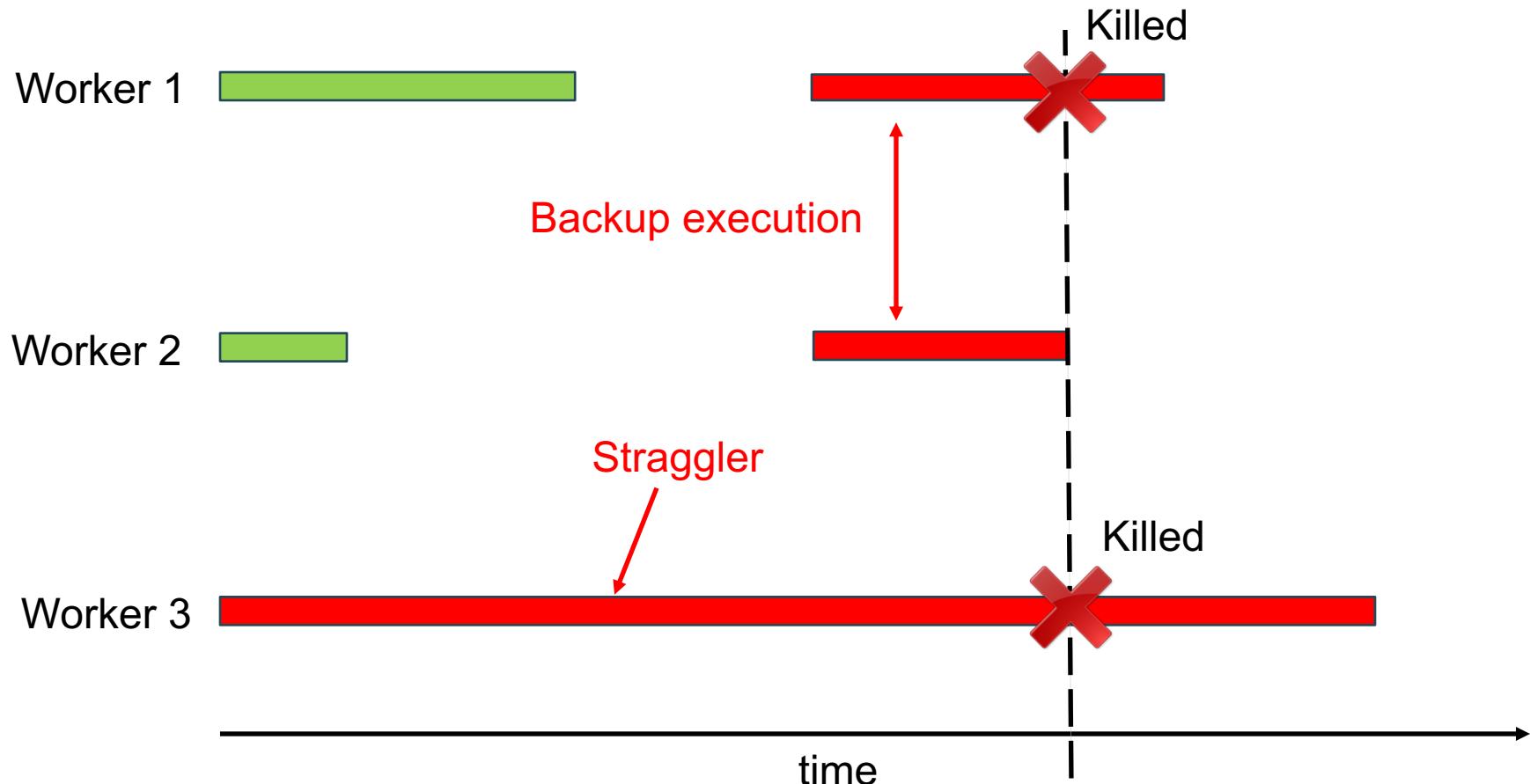
- There is one master node
- Master partitions input file into M splits, by key
- Master assigns *workers* (=servers) to the M map tasks, keeps track of their progress
- Workers write their output to local disk, partition into R regions
- Master assigns workers to the R reduce tasks
- Reduce workers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



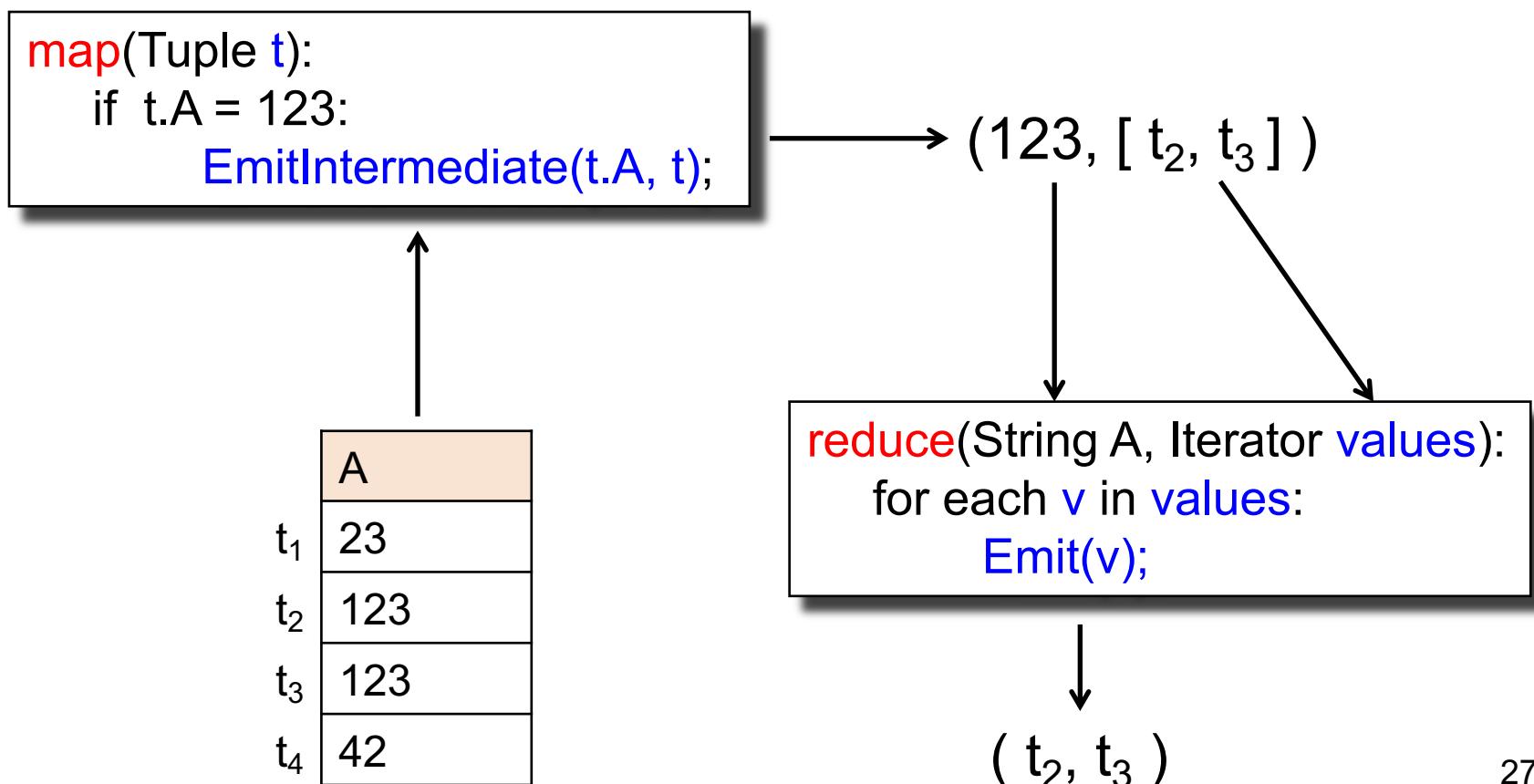
Using MapReduce in Practice: Implementing RA Operators in MR

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

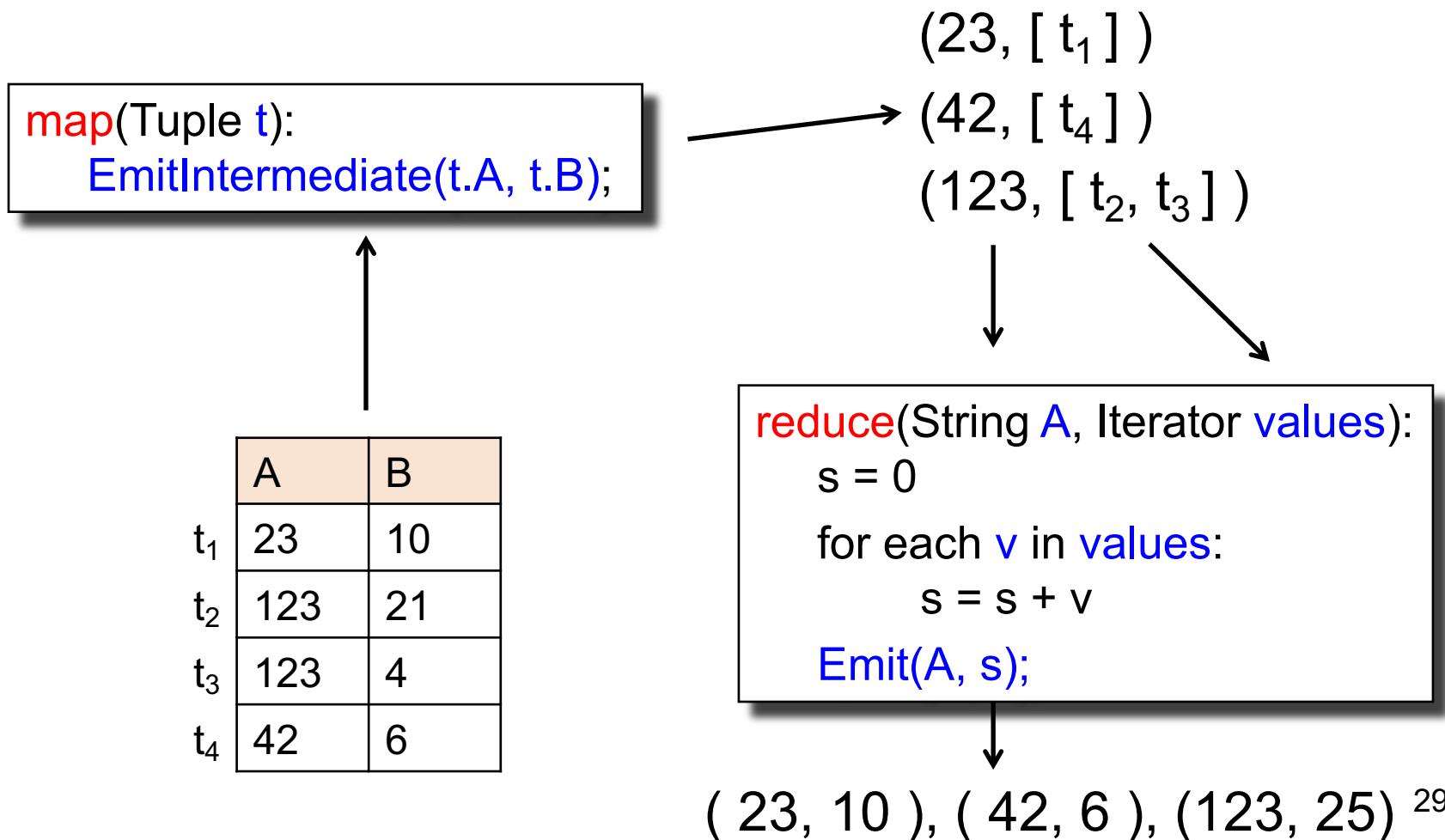
```
map(Tuple t):  
  if t.A = 123:  
    EmitIntermediate(t.A, t);
```

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



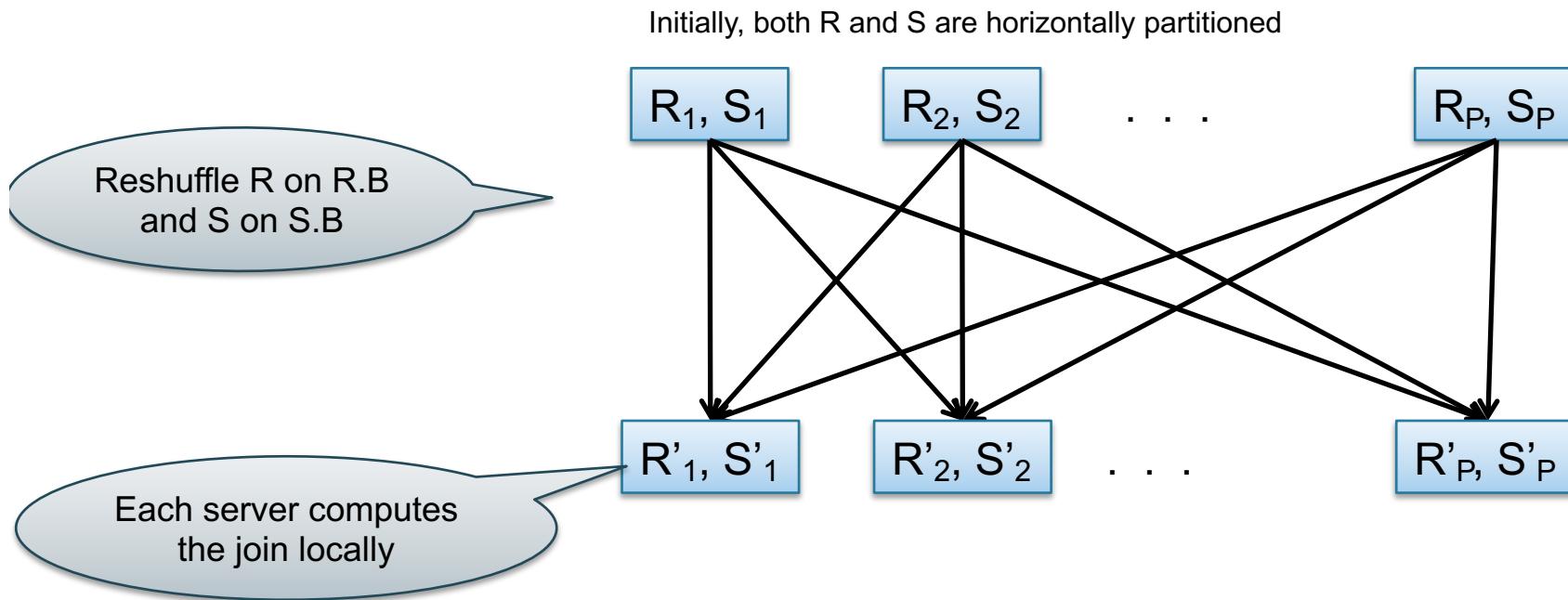
Join

Two simple parallel join algorithms:

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

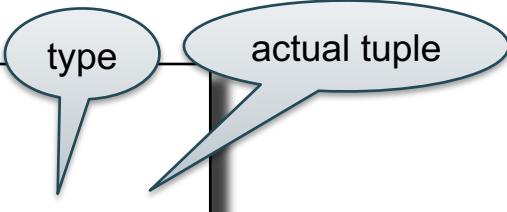
$$R(A,B) \bowtie_{B=C} S(C,D)$$

Partitioned Hash-Join



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

Partitioned Hash-Join



map(Tuple *t*):

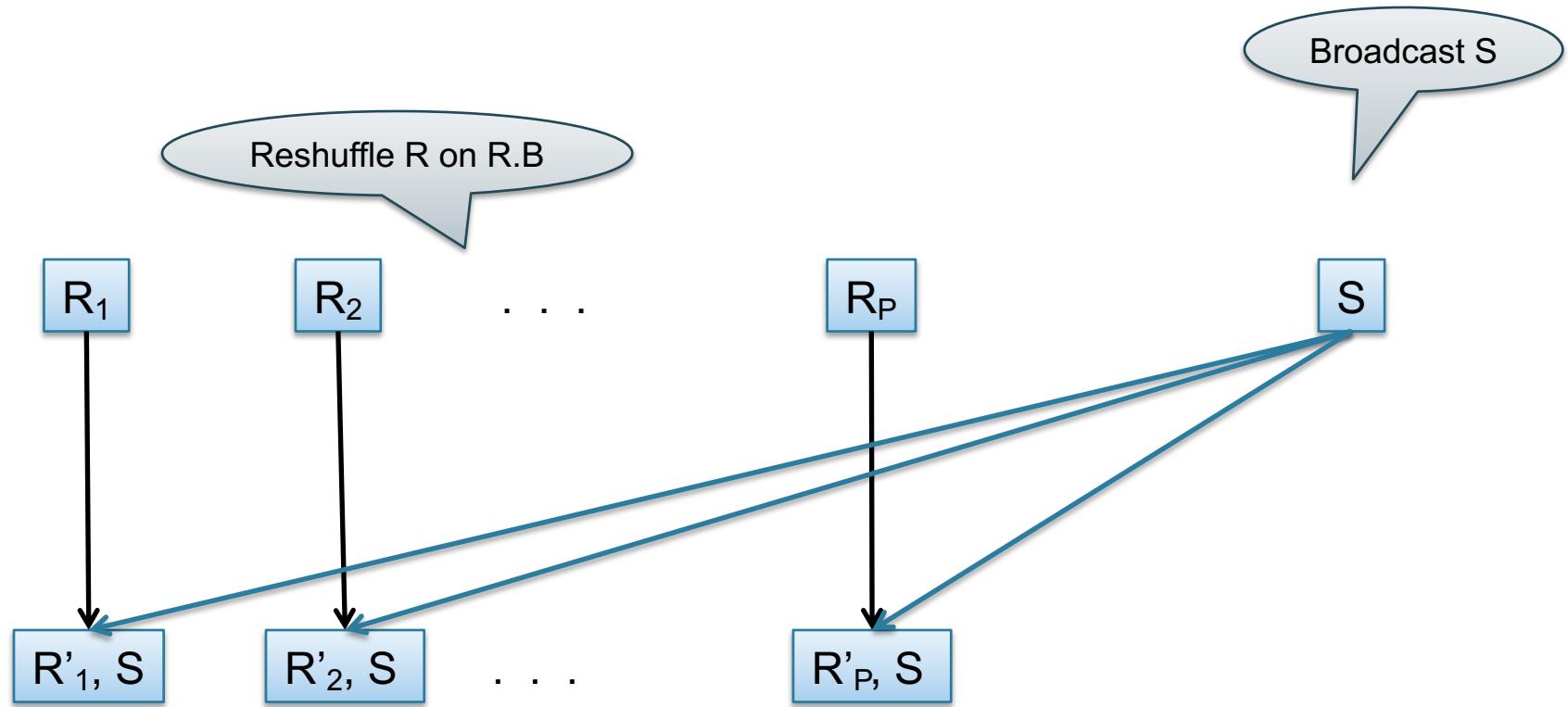
```
case t.relationName of
    'R': EmitIntermediate(t.B, ('R', t));
    'S': EmitIntermediate(t.C, ('S', t));
```

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)
```

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

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

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?

The RDD Interface

Collections in Spark

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

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, 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();
```

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

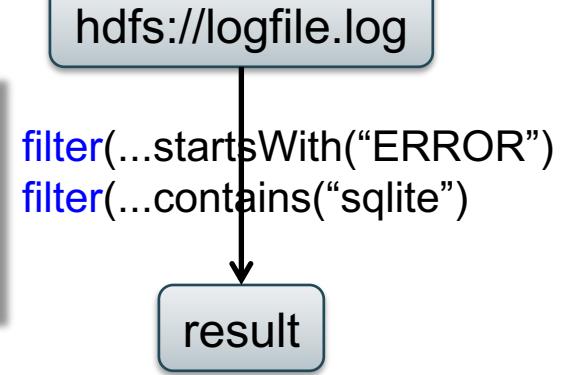
```
lines = s.read().textFile("hdfs://logfile.log");
errors = lines.filter(l->l.startsWith("ERROR"));
sqlerrors = errors.filter(l->l.contains("sqlite"));
sqlerrors.collect();
```

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

Persistence

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

RDD:

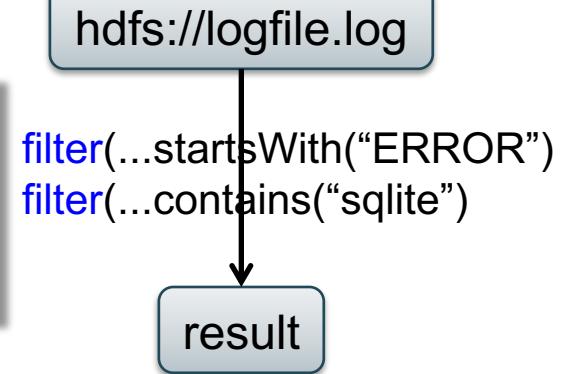


If any server fails before the end, then Spark must restart

Persistence

```
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

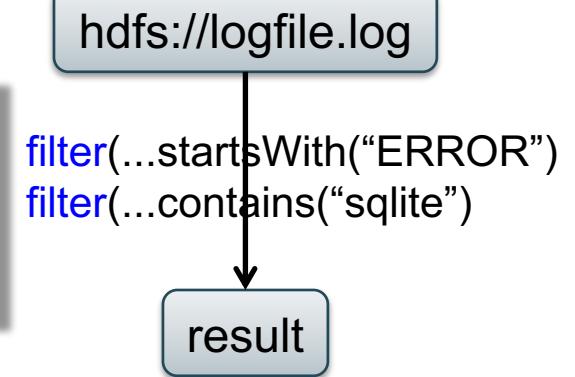
```
lines = s.read().textFile("hdfs://logfile.log");
errors = lines.filter(l->l.startsWith("ERROR"));
errors.persist();           New RDD
sqlerrors = errors.filter(l->l.contains("sqlite"));
sqlerrors.collect()
```

Spark can recompute the result from errors

Persistence

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

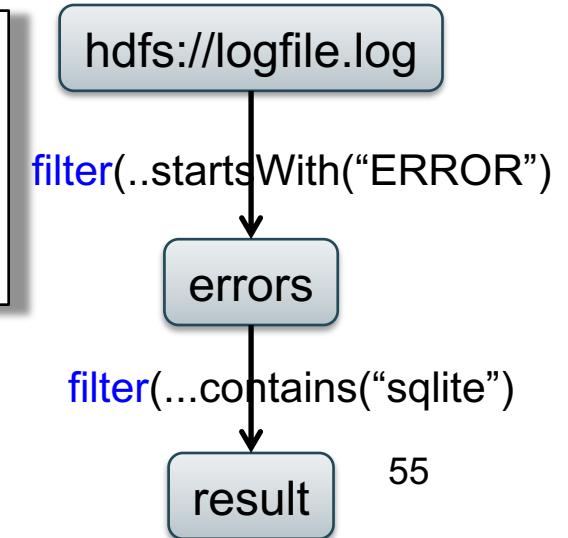
RDD:



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

```
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

CSE 414 - Autumn 2018

55

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)

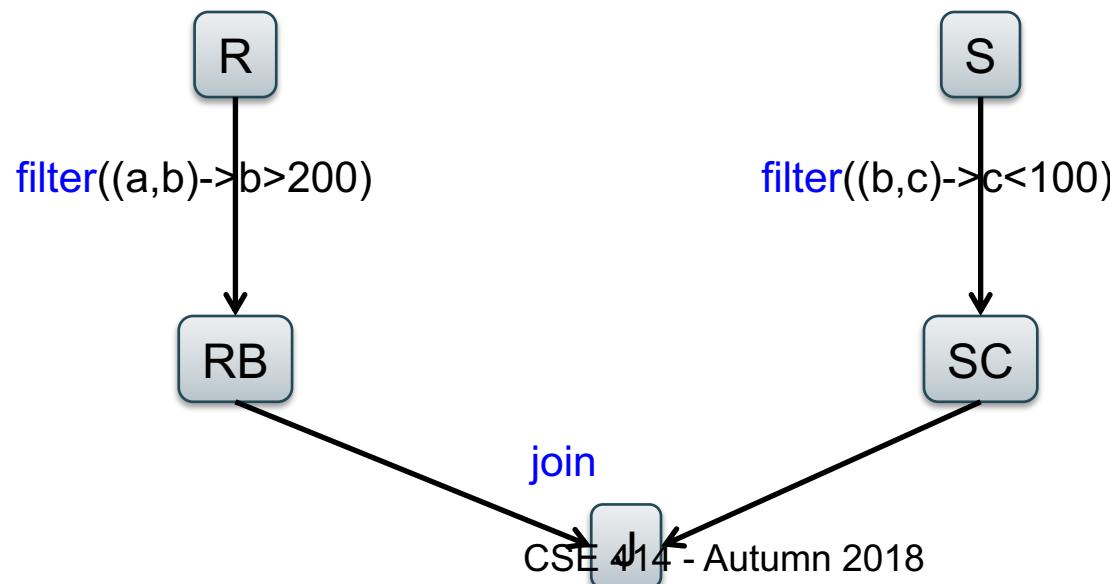
SELECT count(*) FROM R, S
WHERE R.B > 200 and S.C < 100 and R.A = S.A

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

Actions:	
<code>count():</code>	<code>RDD<T> -> Long</code>
<code>collect():</code>	<code>RDD<T> -> Seq<T></code>
<code>reduce(f:(T,T)->T):</code>	<code>RDD<T> -> T</code>
<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