Introduction to Database Systems
CSE 414

Lecture 20: Map-Reduce and Spark

Announcements

- HW6 is two parts
  - Running your Spark code locally
  - Running your Spark code on AWS
- Do all the local coding first, then run on AWS last.
- Useful: http://spark.apache.org/docs/latest/rdd-programming-guide.html

Typical Problems Solved by MR

- Read a lot of data
- Map: extract something you care about from each record (i.e do something with each row)
- Shuffle and Sort
- Reduce: aggregate, summarize, filter, transform
- Write the results (into a new file)

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

Sounds familiar after HW5?

A MapReduce program:

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

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)

values for matching keys are combined into an iterable

(key, value) taken from the output of emitIntermediate
In this case:

key = word
values = list of 1's, one for each occurrence of word

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

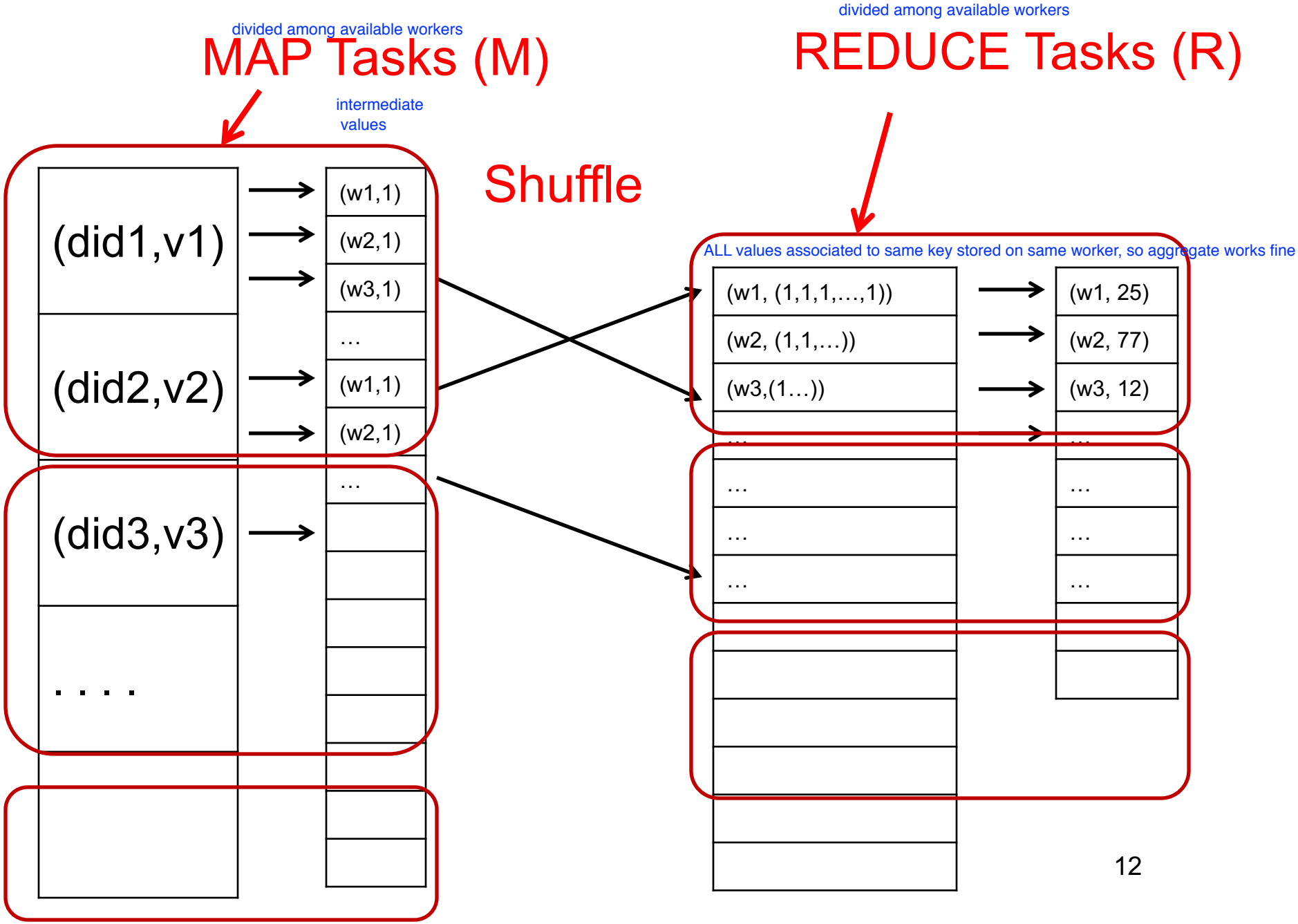
```
reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
    for each v in values:
      result += ParseInt(v);
  emit(AsString(result));
```

Workers

how do we break this up into a parallel program?

- 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


divided among available workers
MAP Tasks (M)
intermediate values
Shuffle
divided among available workers
REDUCE Tasks (R)
ALL values associated to same key stored on same worker, so aggregate works fine
(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)
…
…
…
…

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 slows down process a lot by introducing a lot of disk writes/reads
  - 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

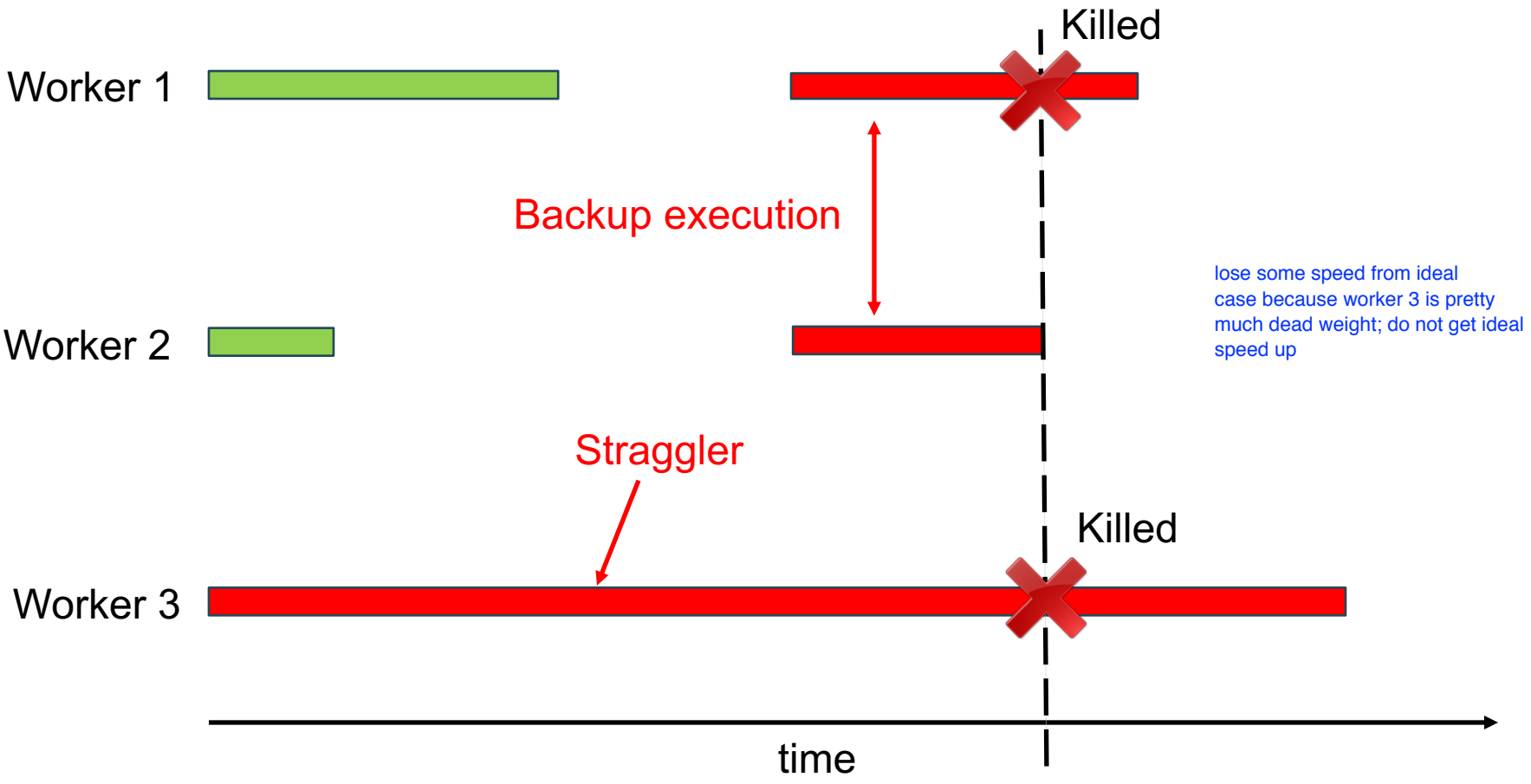
Interesting Implementation Details

Backup tasks: if one machine is significantly slower than others, like skew in data causing one machine to have much more work than others, or machine is faulty, etc.

- *Straggler* = a machine that takes unusually long time to complete one of the last tasks. E.g.: much longer than others
  - 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*

schedule same task to multiple machines; after machines finish their task, assign slow task to already finished machines;

Straggler Example



lose some speed from ideal case because worker 3 is pretty much dead weight; do not get ideal speed up

Using MapReduce in Practice:

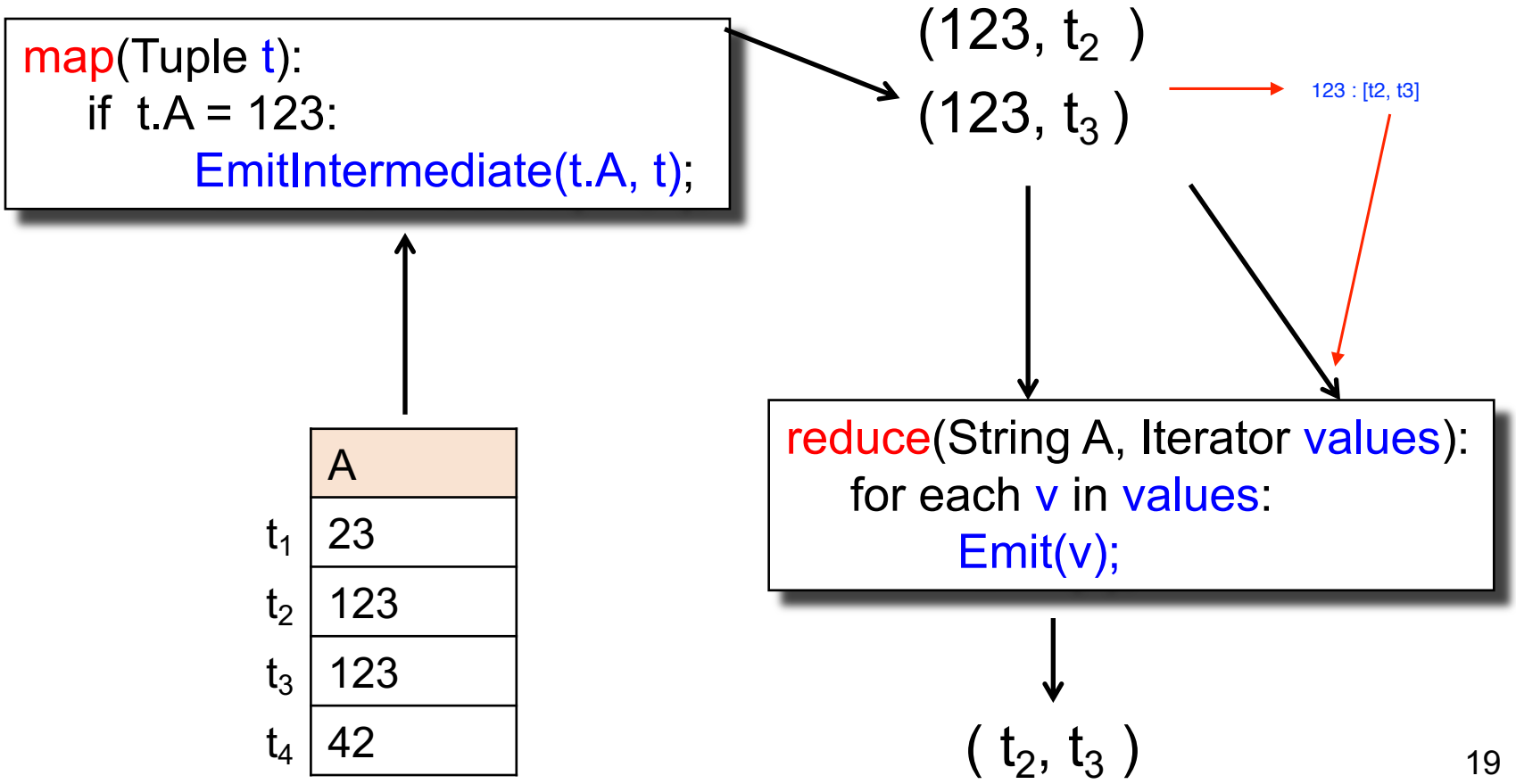
Implementing RA Operators in MR

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: $R \bowtie S$~~ (Saved for later)

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



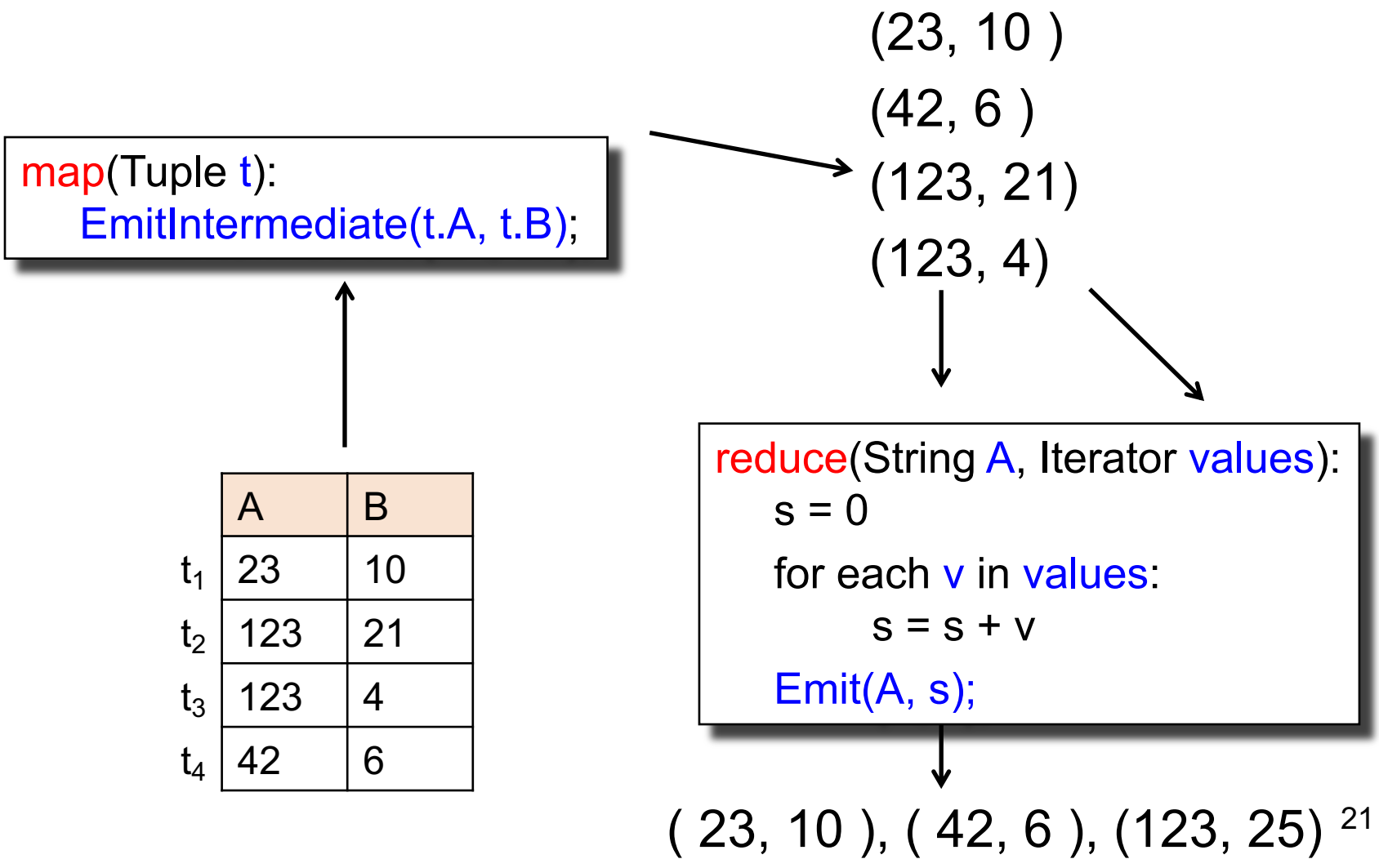
Selection $\sigma_{A=123}(R)$

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



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

Conclusions

- MapReduce offers a simple abstraction, and handles distribution + fault tolerance
- Speedup/scaleup achieved by allocating dynamically map tasks and reduce tasks to available server. However, skew is possible (e.g., one huge reduce task)

as all values associated to a given key are sent to same machine
(Otherwise reduce step couldnt work)

- Writing intermediate results to disk is necessary for fault tolerance, but very slow.
- Spark replaces this with “Resilient Distributed Datasets” = main memory + lineage

RDD

object stored in main memory that keeps track of lineages?

Spark

A Case Study of the MapReduce Programming Paradigm

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?

Parallel Data Processing @ 2010

Spark

- Open source system from UC Berkeley
- Distributed processing over HDFS
- Differences from MapReduce (CSE 322):
  - Multiple steps, including iterations
  - Stores intermediate results in main memory NOT disk
  - 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

functional programming language

- 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 (collection of data (i.e. tuples/ Row objects))
  - Distributed on many servers, not nested
  - Operations are done in parallel
  - Recoverable via lineage; more later
  - We use JavaRDD in HW 6
- Seq<T> = a sequence
  - Local to one server, may be nested
  - Operations are done sequentially

Transformations:		
map(f : T -> U):	RDD<T> -> RDD<U>	map from one type to another (i.e. extract some columns)
mapToPair(f : T -> K, V):	RDD<T> -> RDD<K, V>	
flatMap(f: T -> Seq(U)):	RDD<T> -> RDD<U>	
filter(f:T->Bool):	RDD<T> -> RDD<T>	function that evaluates true or false for every value
groupByKey():	RDD<(K,V)> -> RDD<(K,Seq[V])>	
reduceByKey(F:(V,V)-> V):	RDD<(K,V)> -> RDD<(K,V)>	For matching keys, combine the values in a way described by function F
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

Transformations:	
map(f : T -> U):	RDD<T> -> RDD<U>
mapToPair(f : T -> K, V):	RDD<T> -> RDD<K, V>
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

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

in line functions!

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

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

is the same as:

function F from R to Boolean

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

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

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 l)
  { 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 has type JavaRDD<String>`

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

lines will be a javaRDD of type String (i.e. each row of our datatable is a line in our file)
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                                  logfile.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

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…

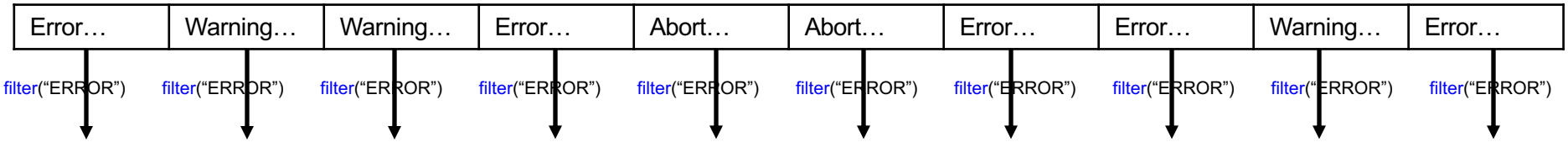
```
s = SparkSession.builder()...getOrCreate();

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

Example

The RDD s:

Parallel step 1

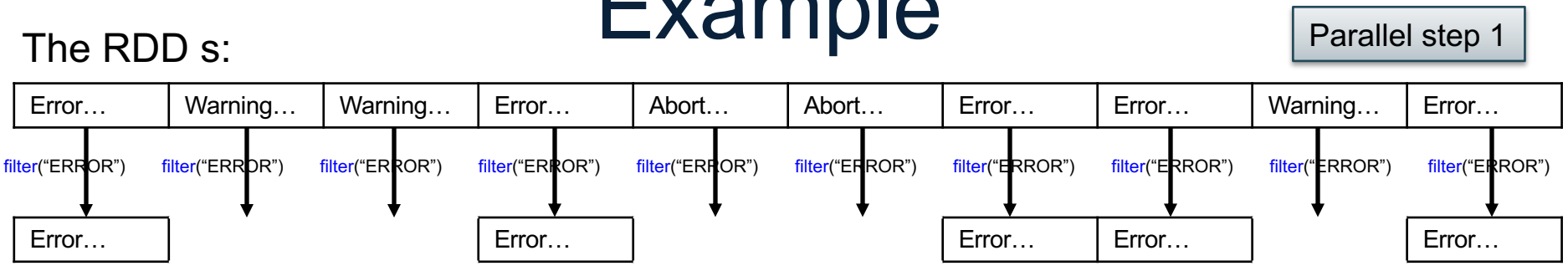



```
s = SparkSession.builder()...getOrCreate();

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

Example

The RDD s:

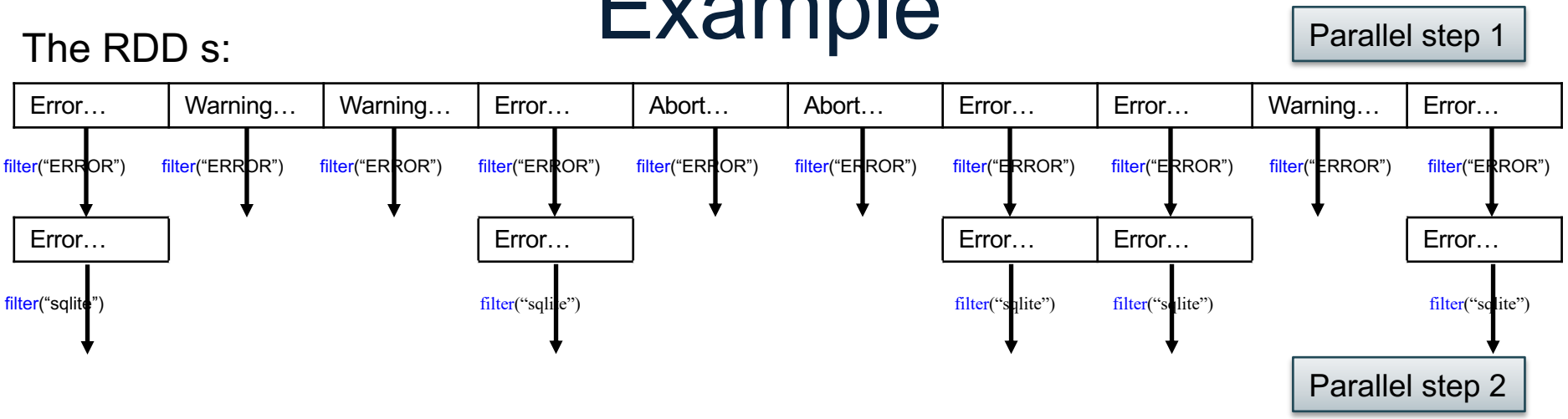



```
s = SparkSession.builder()...getOrCreate();

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

Example

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();
```

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

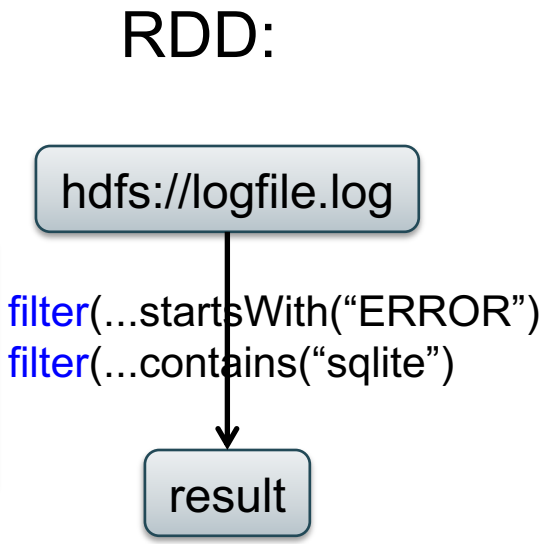
Persistence

```
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

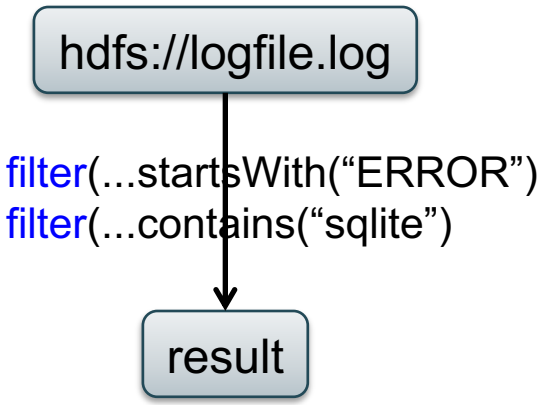
```
lines = s.read().textFile(“hdfs://logfile.log”);
errors = lines.filter(l->l.startsWith(“ERROR”));
sqlerrors = errors.filter(l->l.contains(“sqlite”));
sqlerrors.collect();
```



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();
```



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()
```

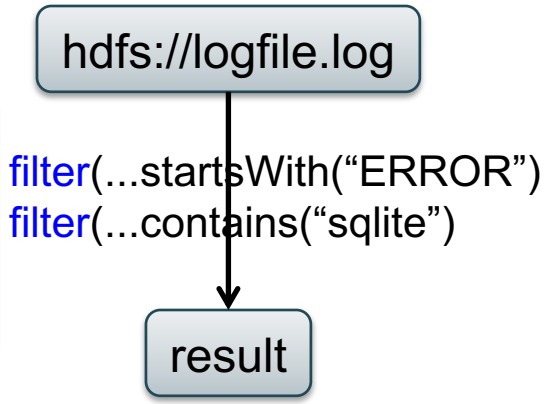
New RDD

Spark can recompute the result from errors

Persistence

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

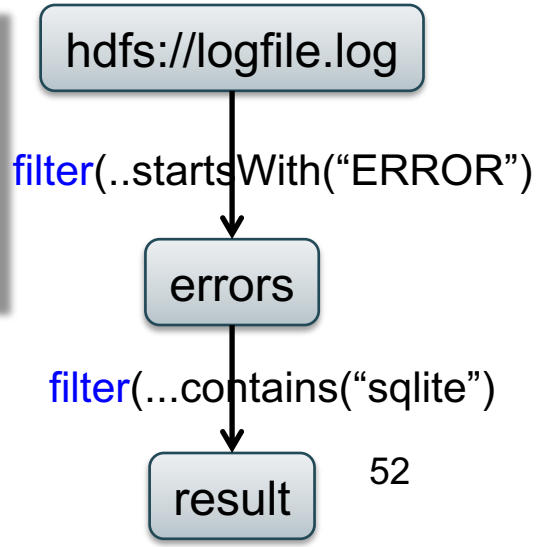
RDD:



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

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)

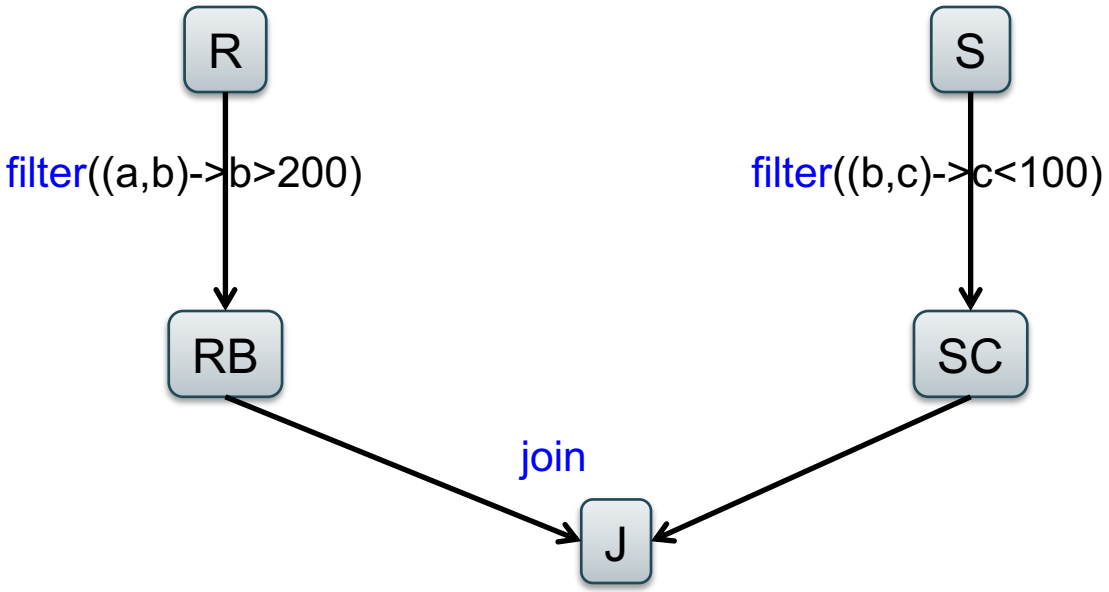
```
SELECT count(*)  FROM R, S
WHERE R.B > 200 and S.C < 100  and R.A = S.A
```

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>
mapToPair(f : T -> K, V):	RDD<T> -> RDD<K, V>
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

Transformations:	
map(f : T -> U):	RDD<T> -> RDD<U>
mapToPair(f : T -> K, V):	RDD<T> -> RDD<K, V>
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?

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