

Introduction to Database Systems

CSE 414

Lecture 21: Spark Wrap-up

Announcements

-

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

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));
```

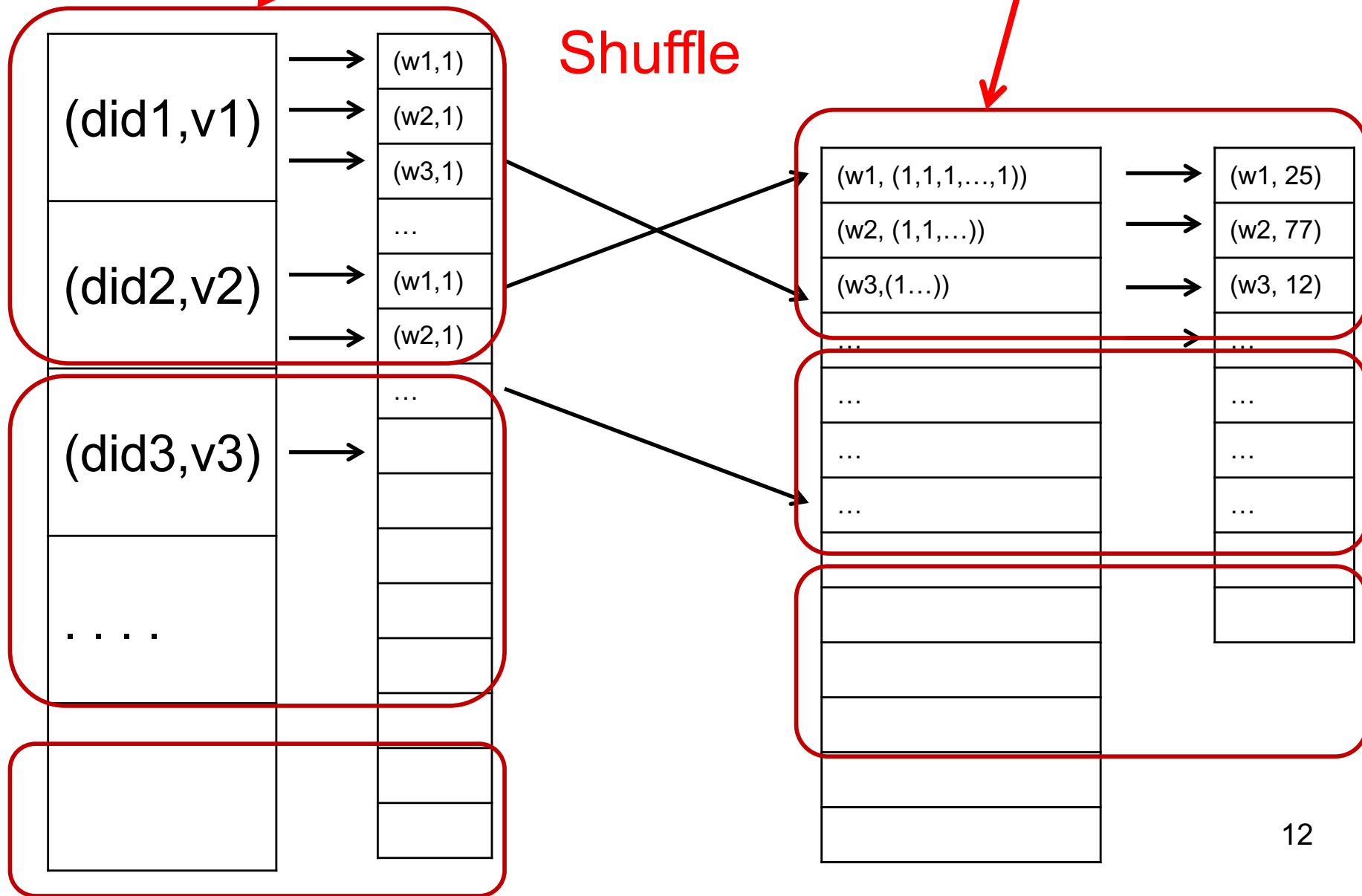
Workers

- 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 Tasks (M)

REDUCE Tasks (R)

Shuffle



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

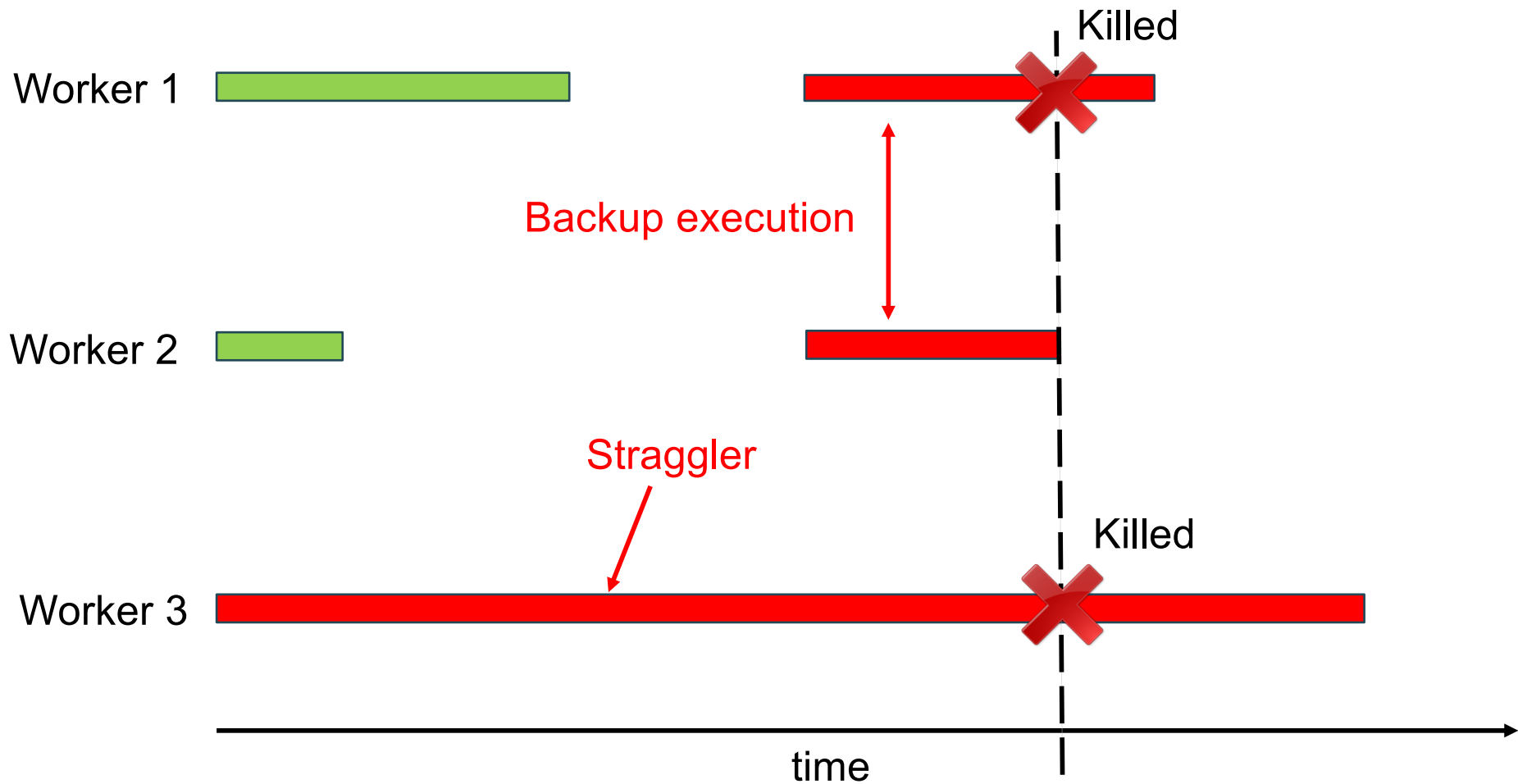
- There is one master node
- Master partitions input file into *M splits*, by key
- Master assigns *workers* (=servers) to the *M map tasks*, keeps track of their progress
- Workers write their output to local disk, partition into *R regions*
- Master assigns workers to the *R reduce tasks*
- Reduce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



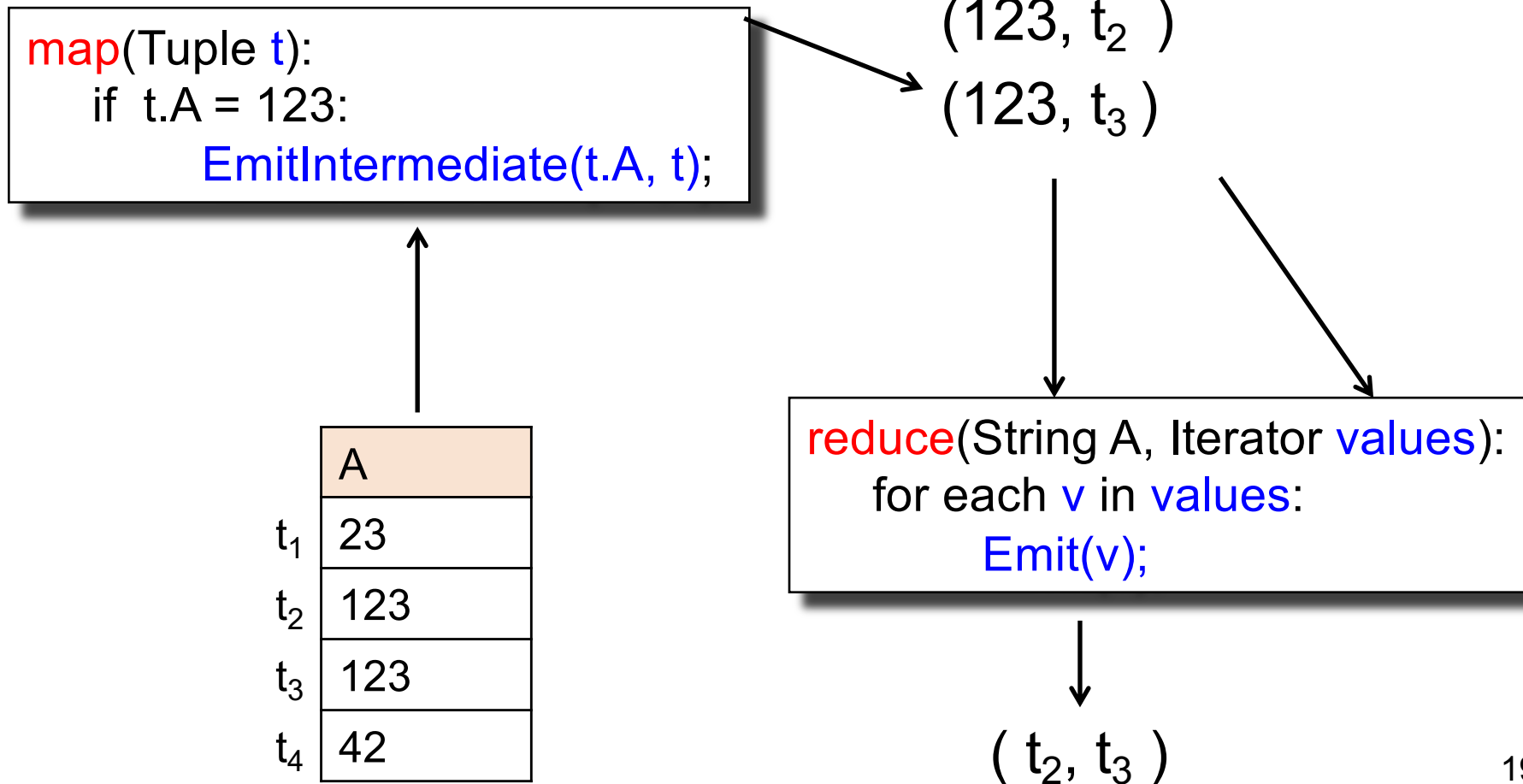
Using MapReduce in Practice: Implementing RA Operators in MR

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:** $R \bowtie S$ (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 $\gamma_{A, \text{sum}(B)}(R)$

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

	A	B
t ₁	23	10
t ₂	123	21
t ₃	123	4
t ₄	42	6

(23, 10)

(42, 6)

(123, 21)

(123, 4)

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

(23, 10), (42, 6), (123, 25)²¹

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

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
 - Closer to relational algebra (familiar to you)
- Details:
<http://spark.apache.org/examples.html>

Spark

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

Programming in Spark

- A Spark program consists of:
 - Transformations (map, reduce, join...). **Lazy**
 - Actions (count, reduce, save...). **Eager**
- **Eager**: operators are executed immediately
- **Lazy**: operators are not executed immediately
 - A *operator tree* is constructed in memory instead
 - Similar to a relational algebra tree

Collections in Spark

- $\text{RDD}\langle T \rangle$ = an RDD collection of type T
 - Distributed on many servers, not nested
 - Operations are done in parallel
 - Recoverable via lineage; more later
 - We use JavaRDD in HW 6
- $\text{Seq}\langle T \rangle$ = a sequence
 - Local to one server, may be nested
 - Operations are done sequentially

Transformations:

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

Transformations:

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

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

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

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

Example

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

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

`s` has type `JavaRDD<String>`

```
s = SparkSession.builder().readTextFile("hdfs://logfile.log");
lines = s.read().textFile("hdfs://logfile.log");
errors = lines.filter(l => l.startsWith("ERROR"));
sqliteErrors = errors.filter(l => l.contains("sqlite"));
sqliteErrors.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:

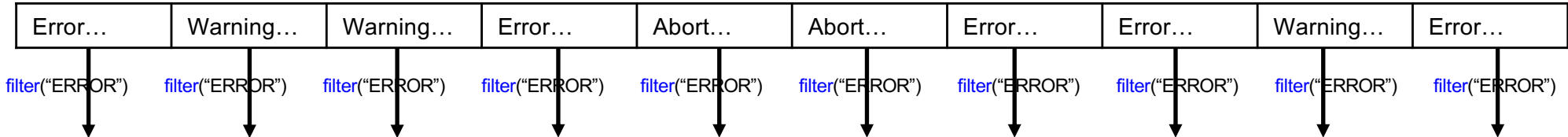
Error...	Warning...	Warning...	Error...	Abort...	Abort...	Error...	Error...	Warning...	Error...
----------	------------	------------	----------	----------	----------	----------	----------	------------	----------

```
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



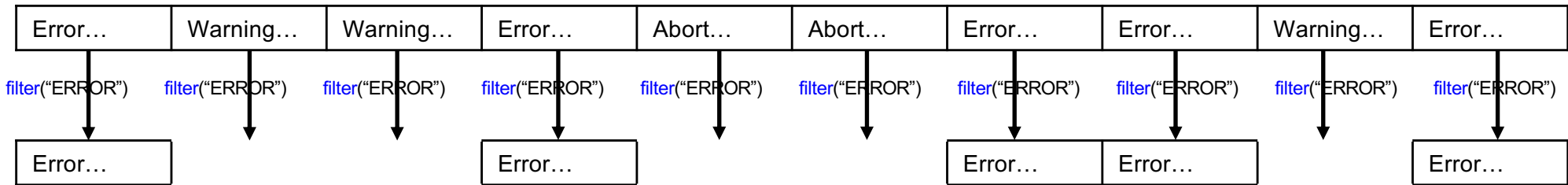
```
s = SparkSession.builder()...getOrCreate();

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

Example

The RDD s:

Parallel step 1

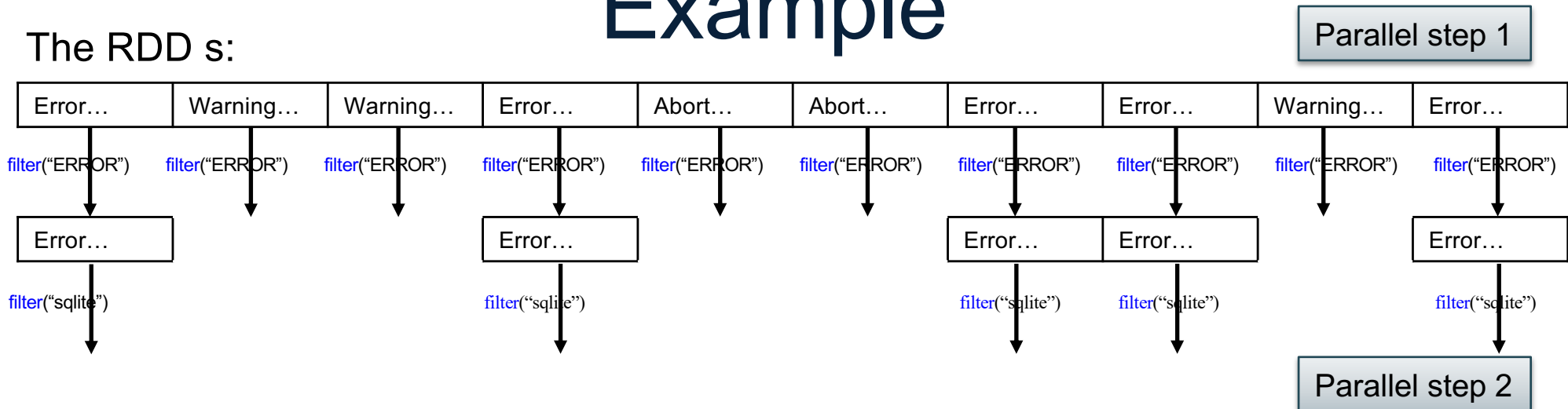


```
s = SparkSession.builder()...getOrCreate();

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();
```

Fault Tolerance

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

Resilient Distributed Datasets

- RDD = Resilient Distributed Dataset
 - Distributed, immutable and records its *lineage*
 - Lineage = expression that says how that relation was computed = a relational algebra plan
- Spark stores intermediate results as RDD
- If a server crashes, its RDD in main memory is lost. However, the driver (=master node) knows the lineage, and will simply recompute the lost partition of the RDD

Persistence

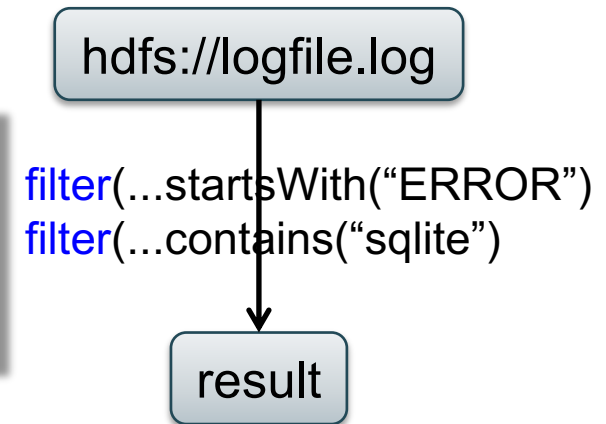
```
lines = s.read().textFile("hdfs://logfile.log");  
errors = lines.filter(l->l.startsWith("ERROR"));  
sqlerrors = errors.filter(l->l.contains("sqlite"));  
sqlerrors.collect();
```

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

Persistence

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

RDD:

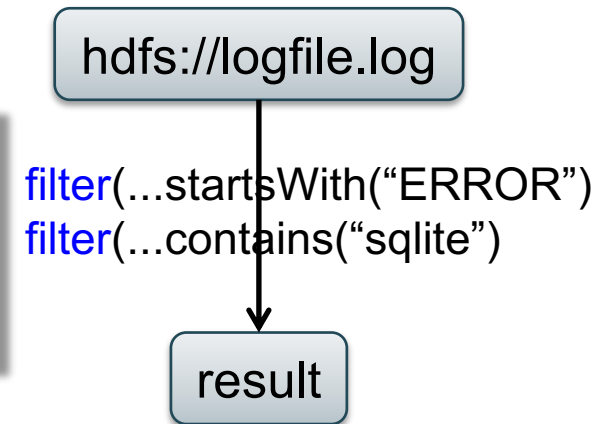


If any server fails before the end, then Spark must restart

Persistence

```
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();
```

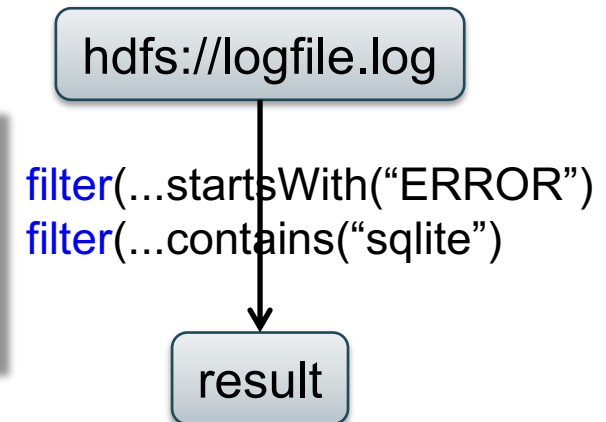
New RDD

Spark can recompute the result from errors

Persistence

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

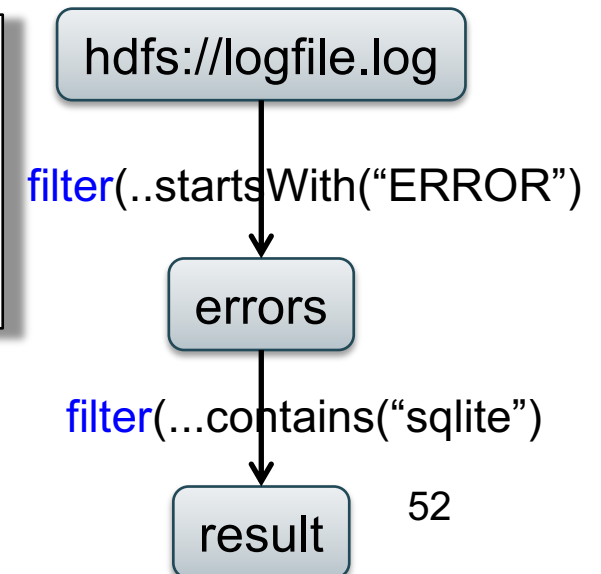
RDD:



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)

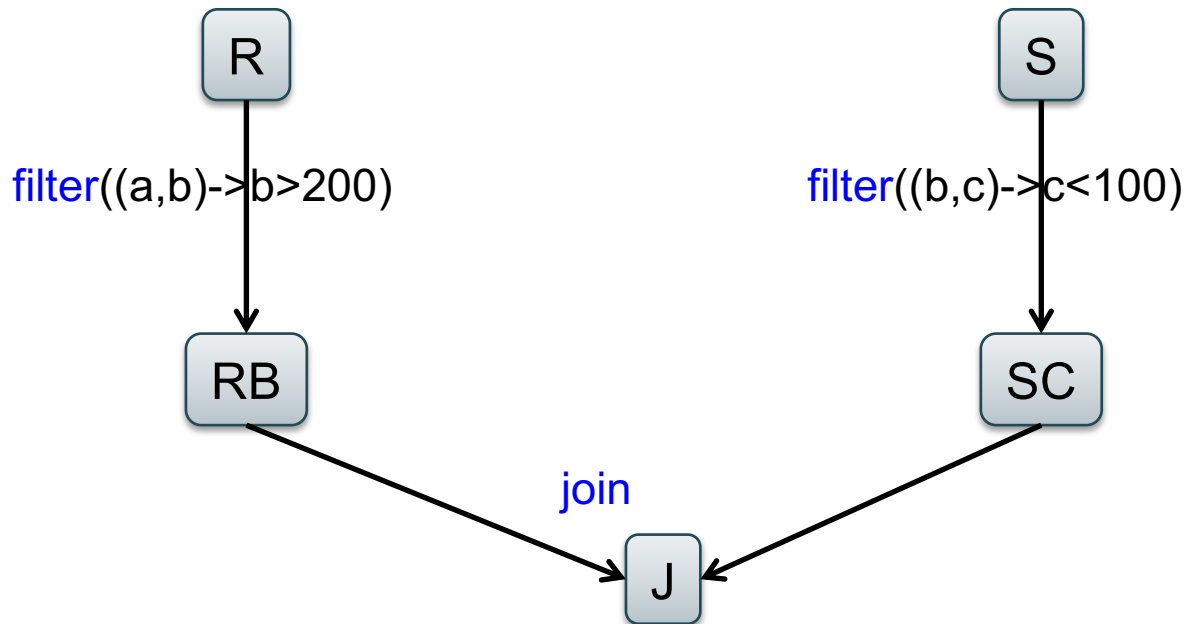
```
SELECT count(*) FROM R, S  
WHERE R.B > 200 and S.C < 100 and R.A = S.A
```

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**
- $\text{RDD}\langle T \rangle$ = 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

Transformations:

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

Transformations:

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