Introduction to Data Management CSE 344

Lecture 25: MapReduce and Spark

Today

- MapReduce Review
- Spark

Parallel Data Processing @ 2000

Review: Map Reduce Data Model

Started by Google in 2004

Instance: Files containing (key, value) pairs

Schema: None!

- just like other key-value data models

Query language: a MapReduce program:

- Input: a bag of (key, value) pairs
- Output: a bag of (key, value) pairs
- Implementation in Java (Hadoop), Python, Go, …

Review: Lifecycle of a MR Program

1. Read a lot of data and parse into (key, value) pairs
2. Map: extract something you care about from each (key, value) pair
3. Shuffle output from mappers
   – done internally by implementation
4. Reduce: aggregate, summarize, filter, transform
5. Write the results to files

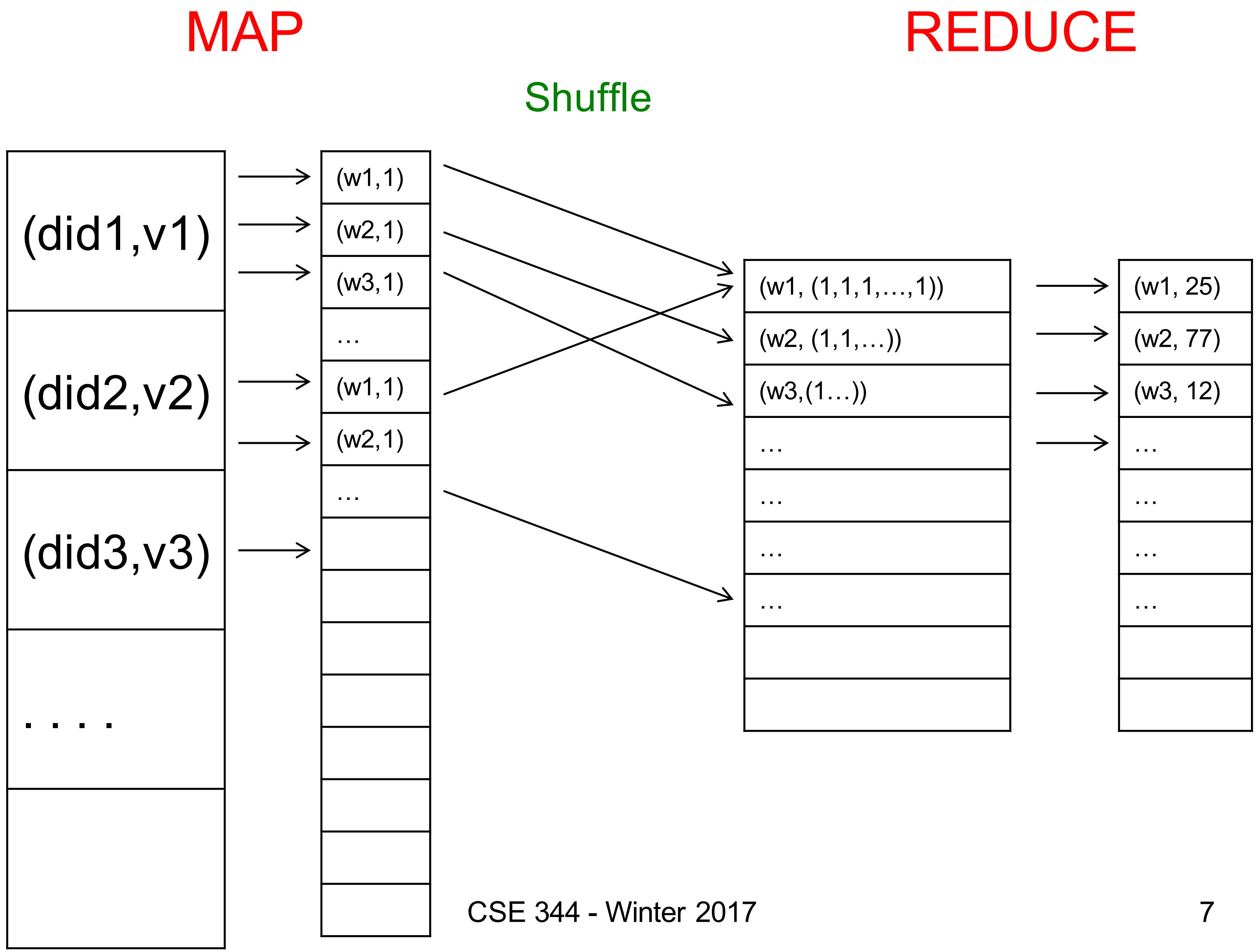
Paradigm stays the same,
change map and reduce
functions for different problems

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 “1”s
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```


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

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

Using MapReduce in Practice:

Implementing RA Operators in MR

Selection $\sigma_{A=42}(R)$

```
map(String relationName, Tuple t):
  if t.A == 42:
    EmitIntermediate(relationName, t);
```

```
reduce(String k, Iterator values):
   for each v in values:
       Emit(v);
```

Selection $\sigma_{A=42}(R)$

```
map(String relationName, Tuple t):
  if t.A == 42:
    EmitIntermediate(relationName, t);
```

```
reduce(String k, Iterator values):
   for each v in values:
        Emit(v);
```

- Reduce isn't really needed
- But MR requires reduce functions

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

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

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

Can’t use hashtable to map A’s to B’s

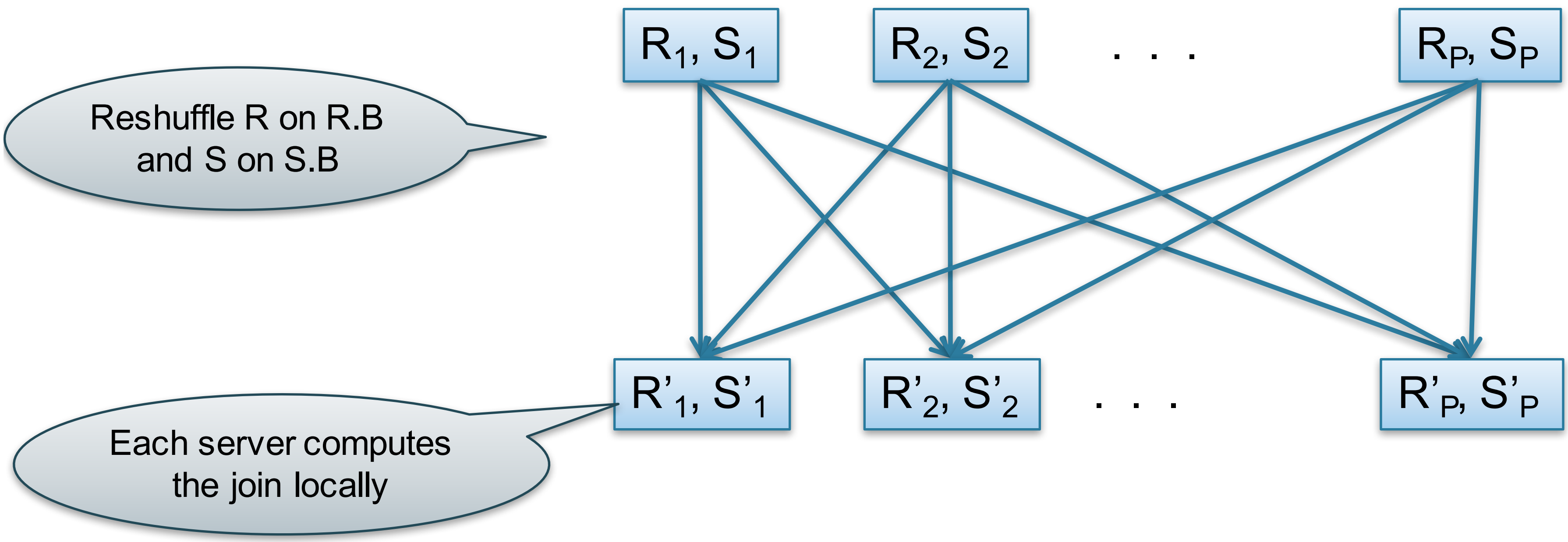
Implementing Join in MR

Two parallel join algorithms that we have seen:

- Partitioned hash-join
- Broadcast join

Parallel Execution of RA Operators: Partitioned Hash-Join

- Data: R(<u>K1</u>,A, B), S(<u>K2</u>, B, C)
- Query: R(<u>K1</u>,A,B) ⋈ S(<u>K2</u>,B,C)
  - Initially, both R and S are partitioned on K1 and K2



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

Partitioned Hash-Join in MR

Or call hash(t.B)

Relation name

value

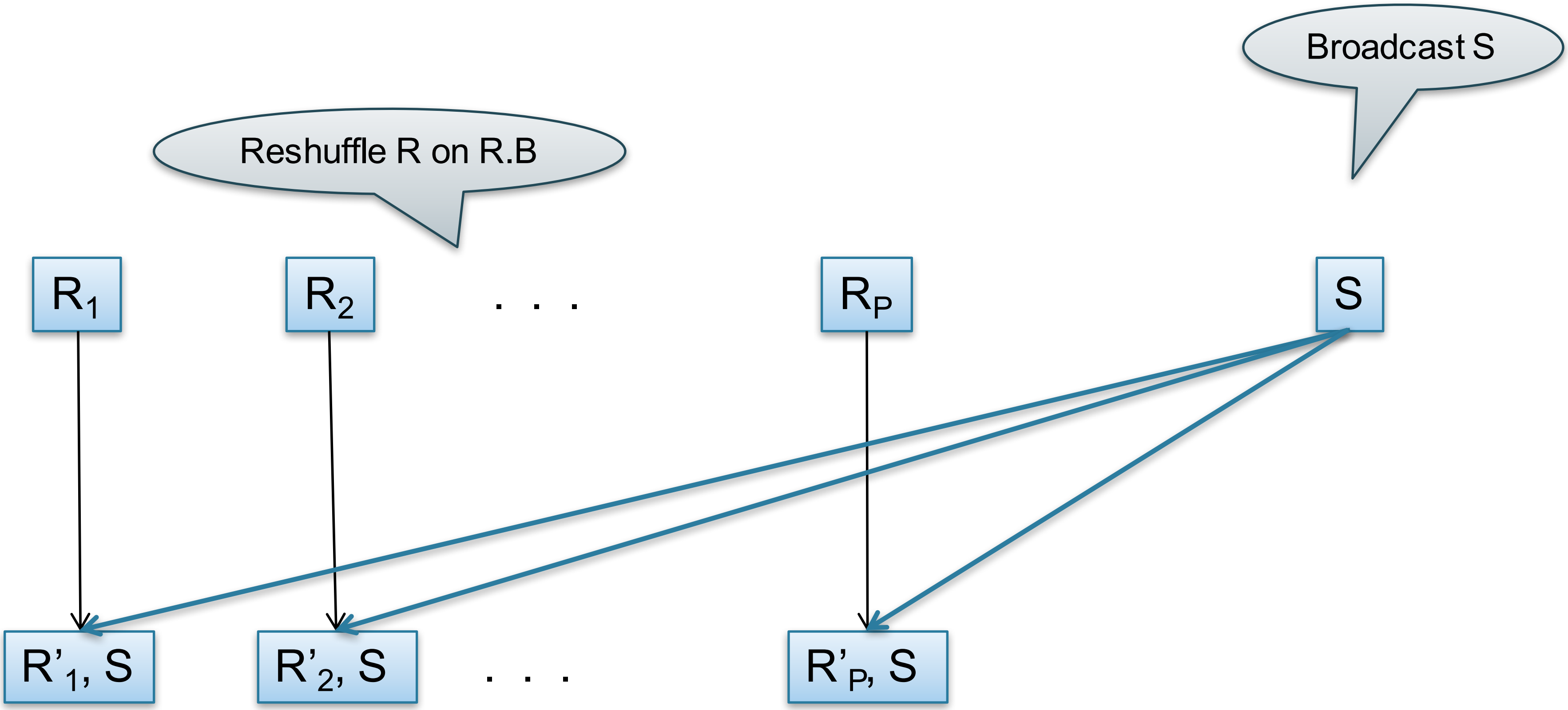
```
map(String relationName, Tuple t):
  switch (relationName):
    case ‘R’: EmitIntermediate(t.B, IntKey(‘R’, value))
    case ‘S’: EmitIntermediate(t.C, IntKey(‘S’, value))
```

All tuples here must join

```
reduce(String k, Iterator values):
  R = []  S = []
  for each v in values:
    switch (v.relationName):
      ‘R’: R.insert(v.value)
      ‘S’: S.insert(v.value)
  for r in R, for s in S:
    Emit(Tuple(r,s))
```

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

Broadcast Join



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

Broadcast Join in MR

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

```
map(String relationName, Tuple [] rs):
   S = readFromNetwork()
   ht = new Hashtable()
   for each w in S:
     ht.insert(w.C, w)

   for each r in ts:
     for each s in ht.find(r.B):
       Emit(Tuple(r,s))
```

Read entire table S, build a Hash Table

Perform join

```
reduce(…):
   /* empty: map-side only */
```

Issues with MapReduce

- Difficult to write complex queries
  - Nested queries
  - Correlated queries?
- Fault tolerance: only persists results between map / reduce
- Need multiple MR jobs: dramatically slows down because each job writes all results to disk

Parallel Data Processing @ 2010

Spark

- Open source system from Berkeley
- Distributed processing over HDFS
- Differences from MapReduce:
  - Not restricted to pairs of mapper and reducer
  - User decides when to persist results

- Details: http://spark.apache.org/examples.html

Spark Data Model

Instance: Resilient Distributed Datasets (RDDs)

Schema: None! (just like MR)

Query language: a Spark program

- Implementation in Scala / Java / SQL
- Scala = extension of Java with functions/closures

RDD

- RDD = Resilient Distributed Datasets
  - A distributed relation, together with its *lineage*
  - Lineage: expression that says how that relation was computed

- 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

How does Spark store lineage?

- A Spark/Scala program consists of:
  - Transformations (map, reduce, join…). Lazily evaluated
  - Only record the function to invoke, actual work not done
  - Actions (count, reduce, save...). Eagerly evaluated
  - Really performs work

Example

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

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

```
lines = spark.textFile(“hdfs://logfile.log”);

errors = lines.filter(_.startsWith(“ERROR”));

sqlerrors = errors.filter(_.contains(“sqlite”));

sqlerrors.collect()
```

Example

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

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

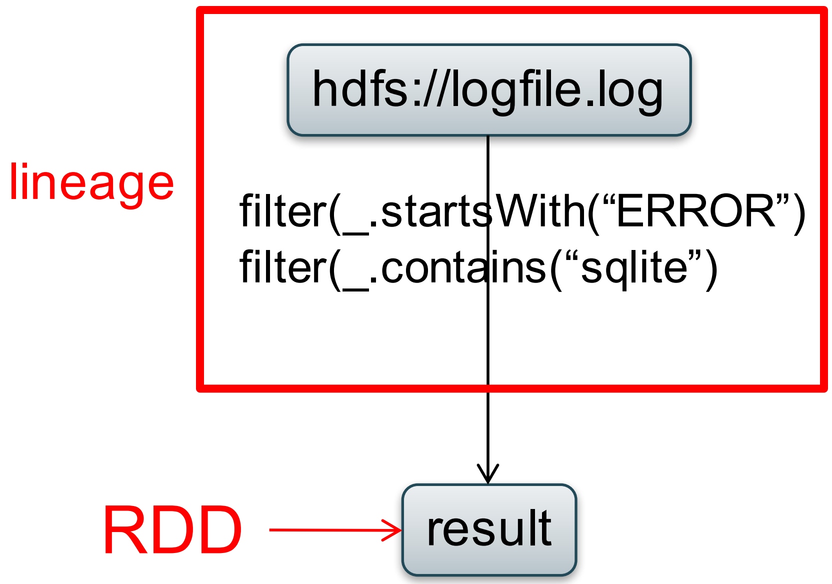
```
lines = spark.textFile(“hdfs://logfile.log”);

errors = lines.filter(_.startsWith(“ERROR”));

sqlerrors = errors.filter(_.contains(“sqlite”));

sqlerrors.collect()
```

Transformations:
Not executed yet…

Action:
triggers execution of entire program

Persistence

```
lines = spark.textFile("hdfs://logfile.log");
errors = lines.filter(_.startsWith("ERROR"));
sqlerrors = errors.filter(_.contains("sqlite"));
result = sqlerrors.collect();
```

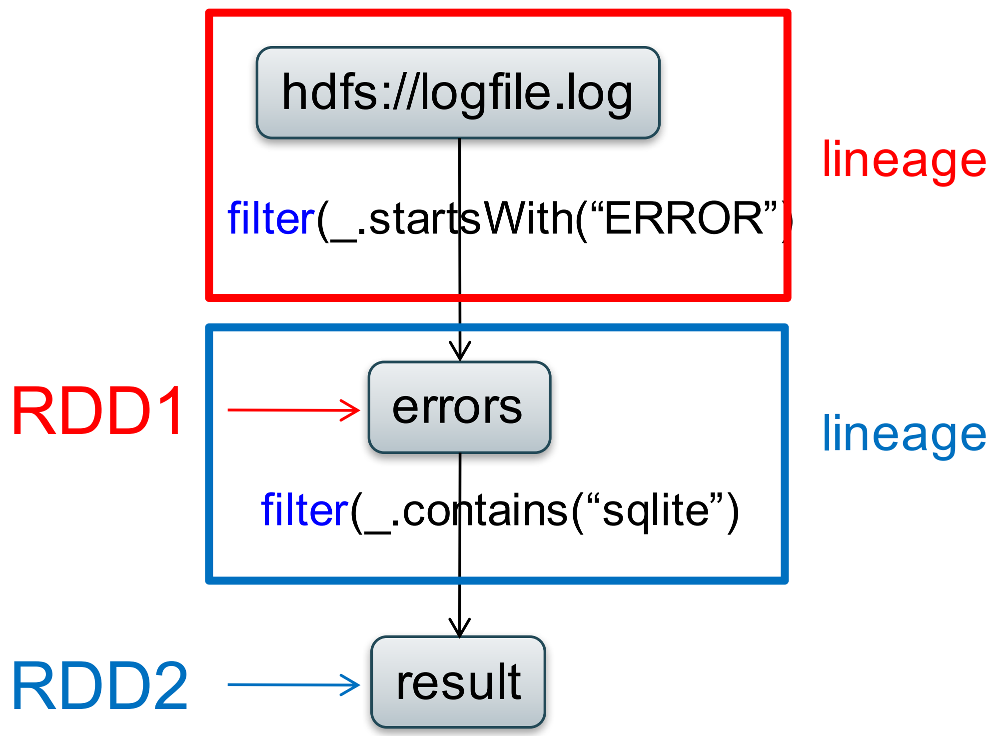


If any server fails before `collect`, then the entire job is restarted

Persistence

```
lines = spark.textFile(“hdfs://logfile.log”);
errors = lines.filter(_.startsWith(“ERROR”));
errors.persist();
sqlerrors = errors.filter(_.contains(“sqlite”));
result = sqlerrors.collect();
```

New RDD



Spark can recompute the result from errors

Transformations:	
map(f : T => U):	RDD[T] => RDD[U]
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

Conclusions

- Parallel databases
  - Predefined relational operators
  - Optimization using standard RA techniques
  - Transaction support is free

- 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