CSE 544
Principles of Database Management Systems

Lecture 13 – Parallel Programming Models: Map Reduce and Spark

Announcements

- Project Milestone due on Friday
- HW4 due next Friday
- Today’s office hour: 3-3:30, 4-4:30

Map Reduce

- Google: [Dean 2004]
- Open source implementation: Hadoop

- MapReduce = high-level programming model and implementation for large-scale parallel data processing

Map Reduce Motivation

- Not designed to be a DBMS
- Designed to simplify task of writing parallel programs
- Hides messy details in MapReduce runtime library:
  - Automatic parallelization
  - Load balancing
  - Network and disk transfer optimizations
  - Handling of machine failures
  - Robustness

Distributed File System

- GFS: Google File System (proprietary)
- HDFS: Hadoop File System (open source)

- Each data file is split into M blocks (64MB or more)
- Blocks are stored on random machines & replicated
- Files are append only

MapReduce Data Model

Files!

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

A MapReduce program:

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

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)`

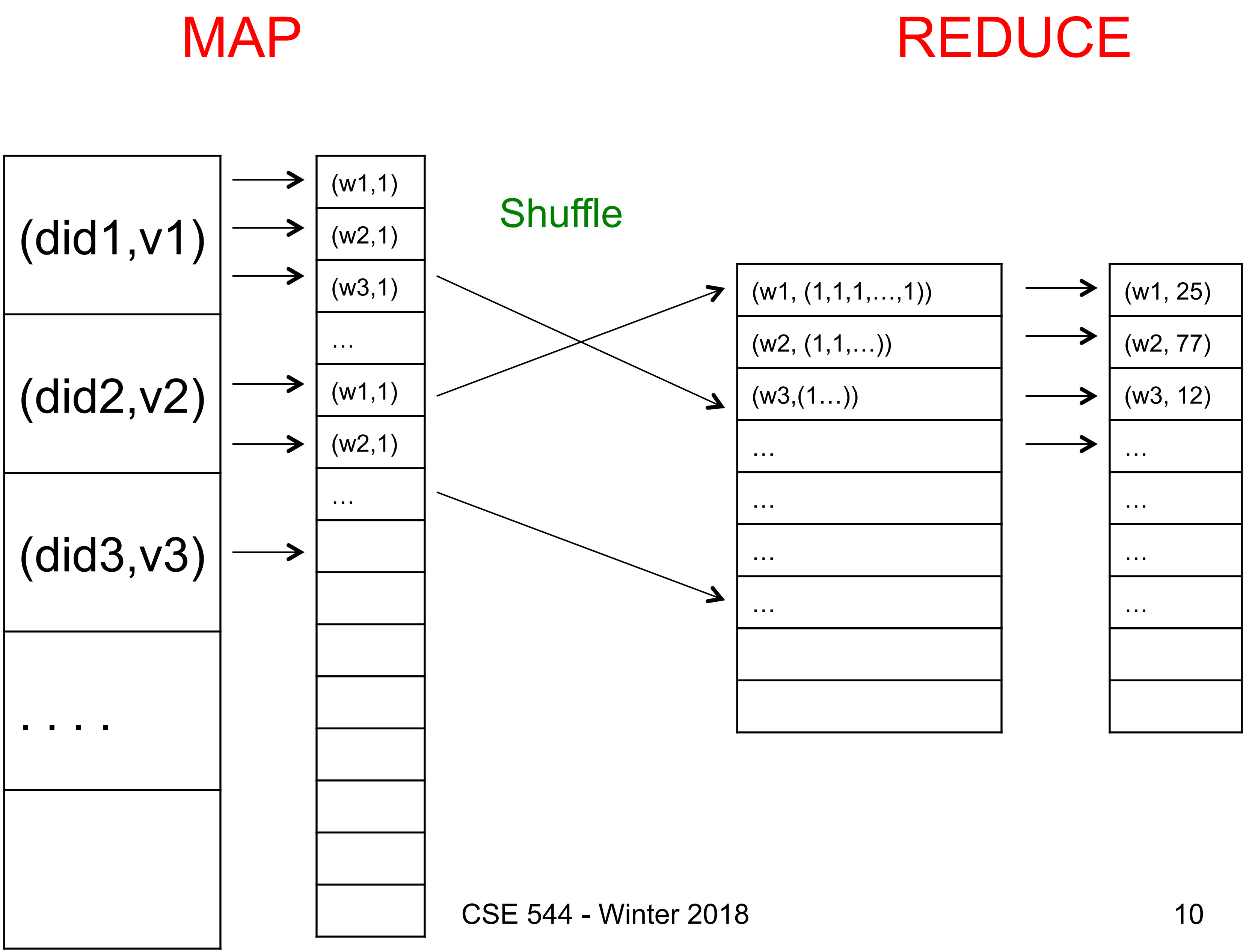
System groups all pairs with the same `intermediate-key`, and applies the reduce function in parallel

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

Jobs v.s. Tasks

- A MapReduce Job
  - One single "query", e.g. count the words in all docs
  - More complex queries may consists of multiple jobs
- A Map Task, or a Reduce Task
  - A group of instantiations of the map-, or reduce-function, which are scheduled on a single worker

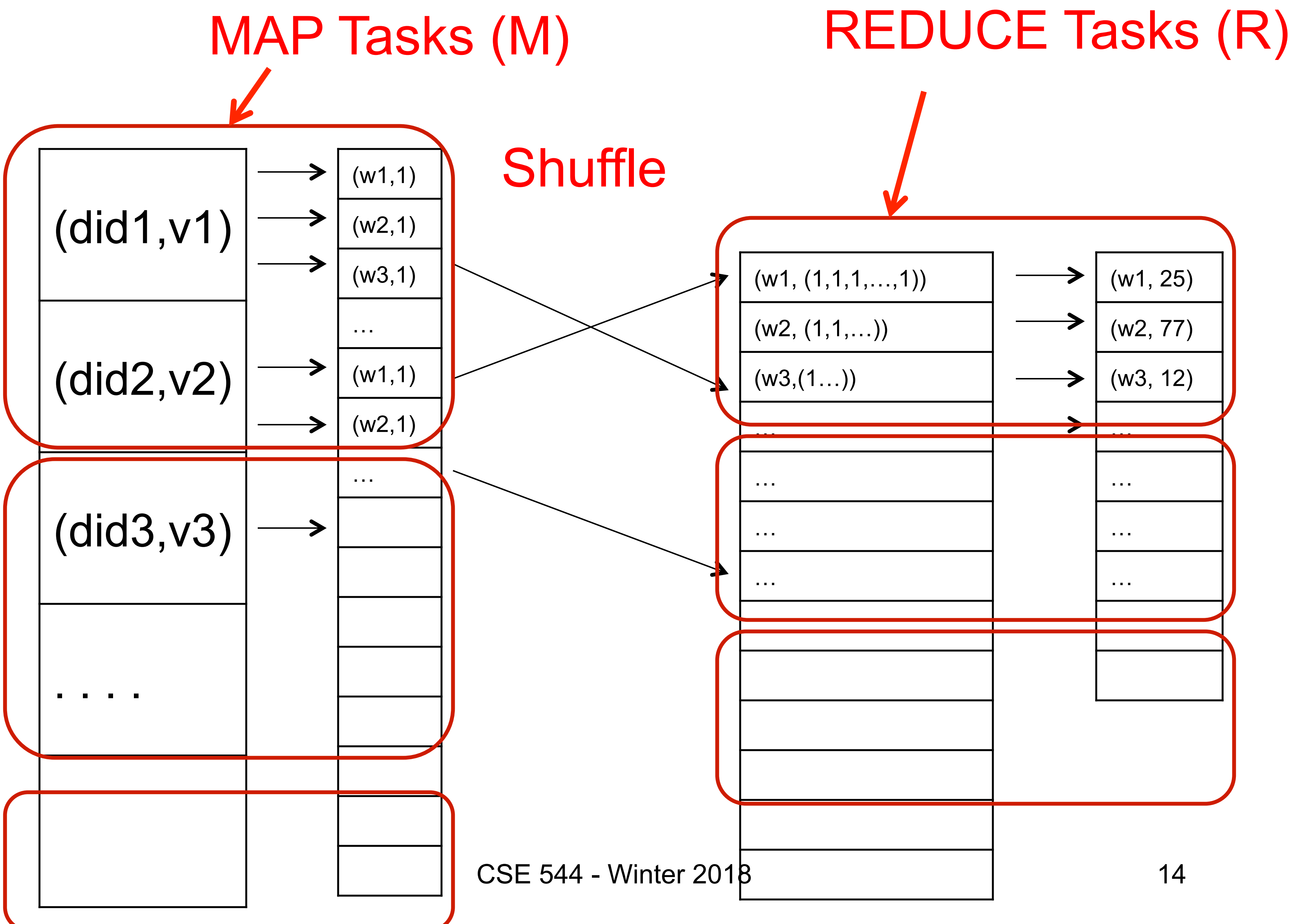
Workers

- 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

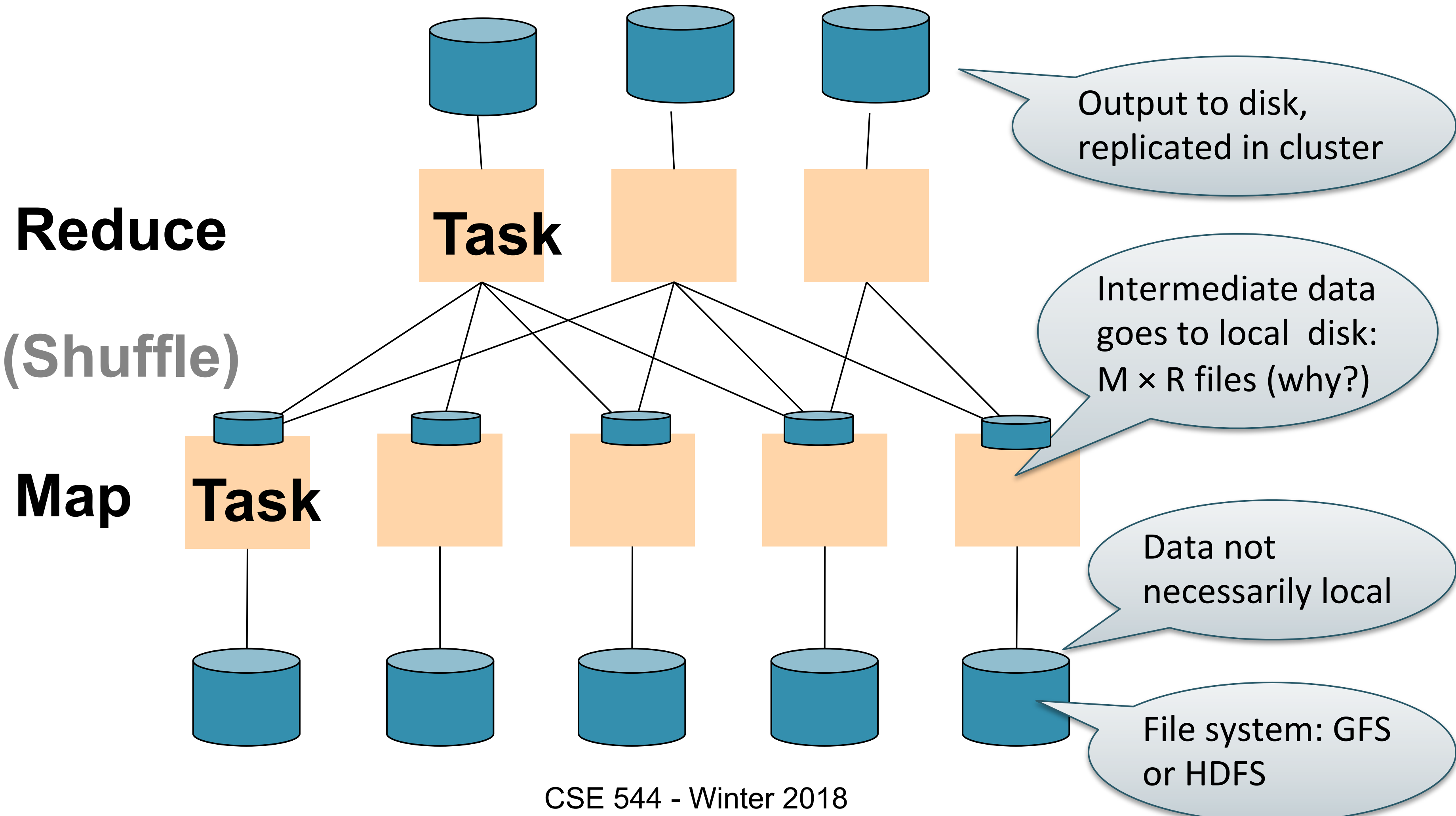
Fault Tolerance

- If one server fails once every year…
  ... then a job with 10,000 servers will fail in less than one hour

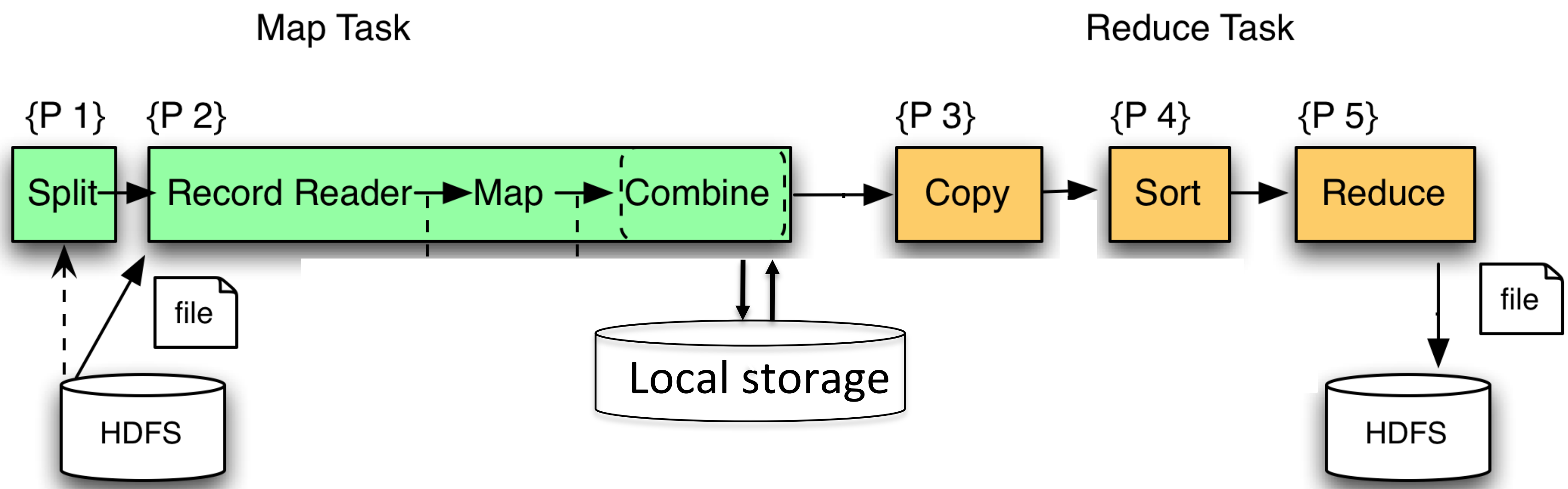
- MapReduce handles fault tolerance by writing intermediate files to disk:
  - Mappers write file to local disk
  - Reducers read the files (=reshuffling); if the server fails, the reduce task is restarted on another server


MAP Tasks (M)
REDUCE Tasks (R)
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)
…
…
…
…

MapReduce Execution Details



MapReduce Phases



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

Worker failure:

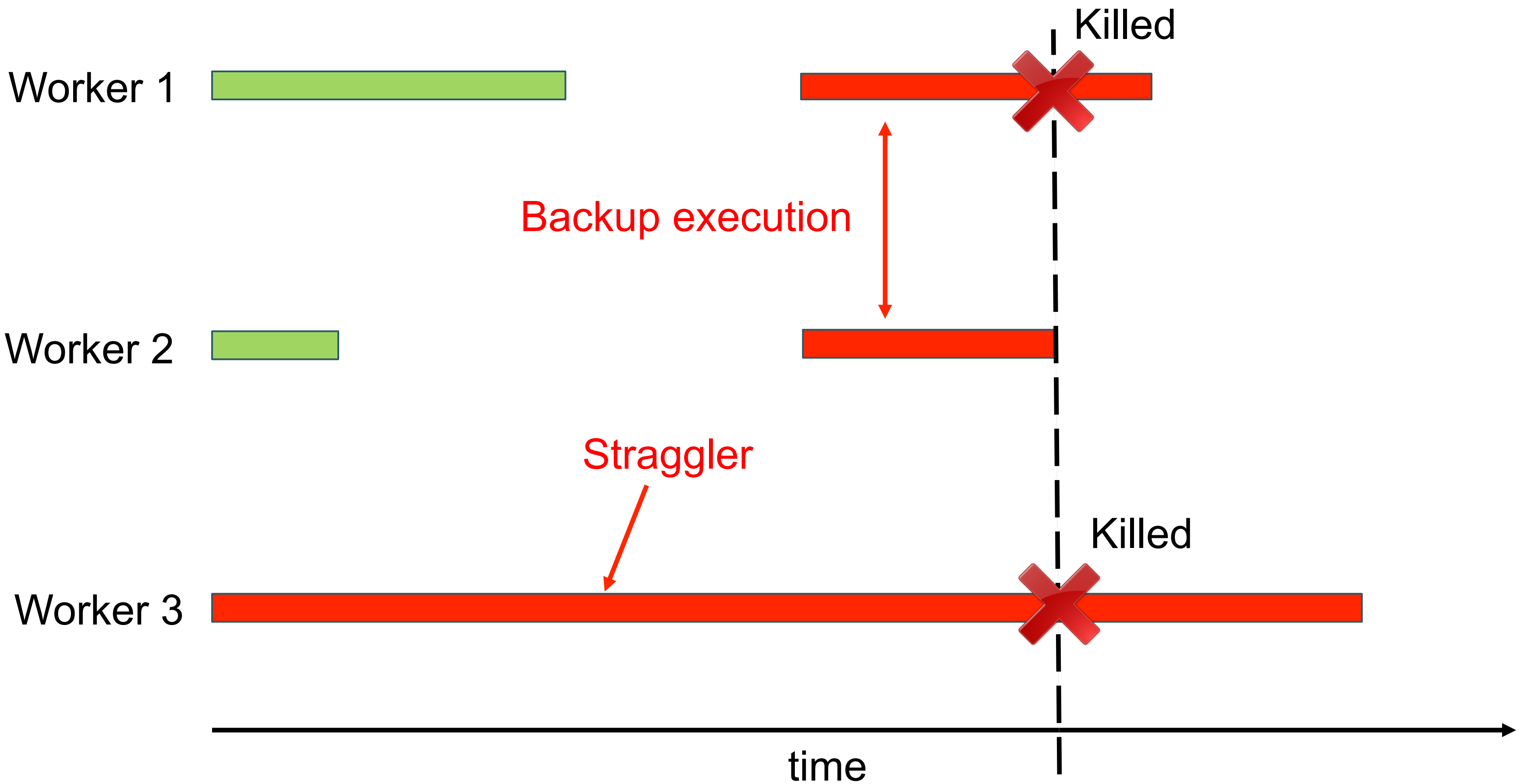
- Master pings workers periodically,
- If down then reassigns the task to another worker

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: $\gamma_{A,sum(B)}(R)$
- Join: $R \bowtie S$

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

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

```
map(String value):
    if  value.A = 123:
            EmitIntermediate(value.key, value);
```

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

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

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

```
map(String value):
    if  value.A = 123:
          EmitIntermediate(value.key, value);
```

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

No need for reduce.
But need system hacking in Hadoop
to remove reduce from MapReduce

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

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

```
map(String value):
    EmitIntermediate(value.A, value.B);
```

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

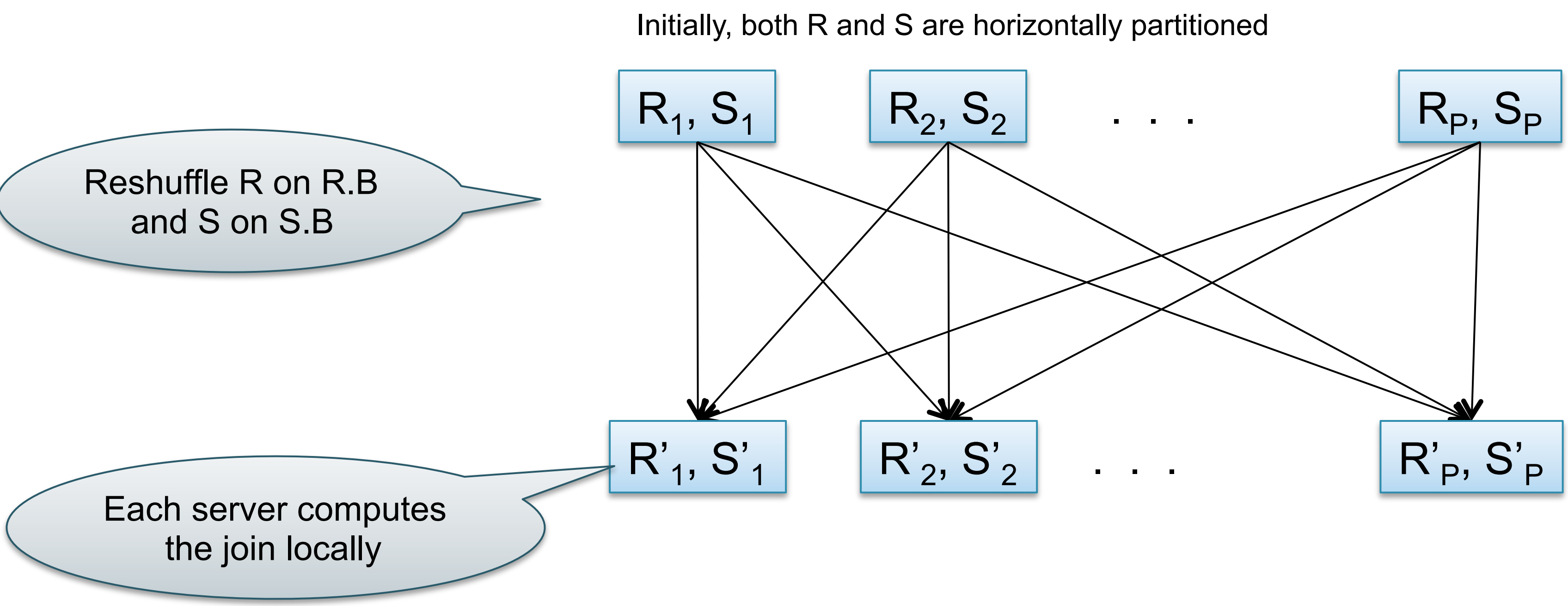
Join

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



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

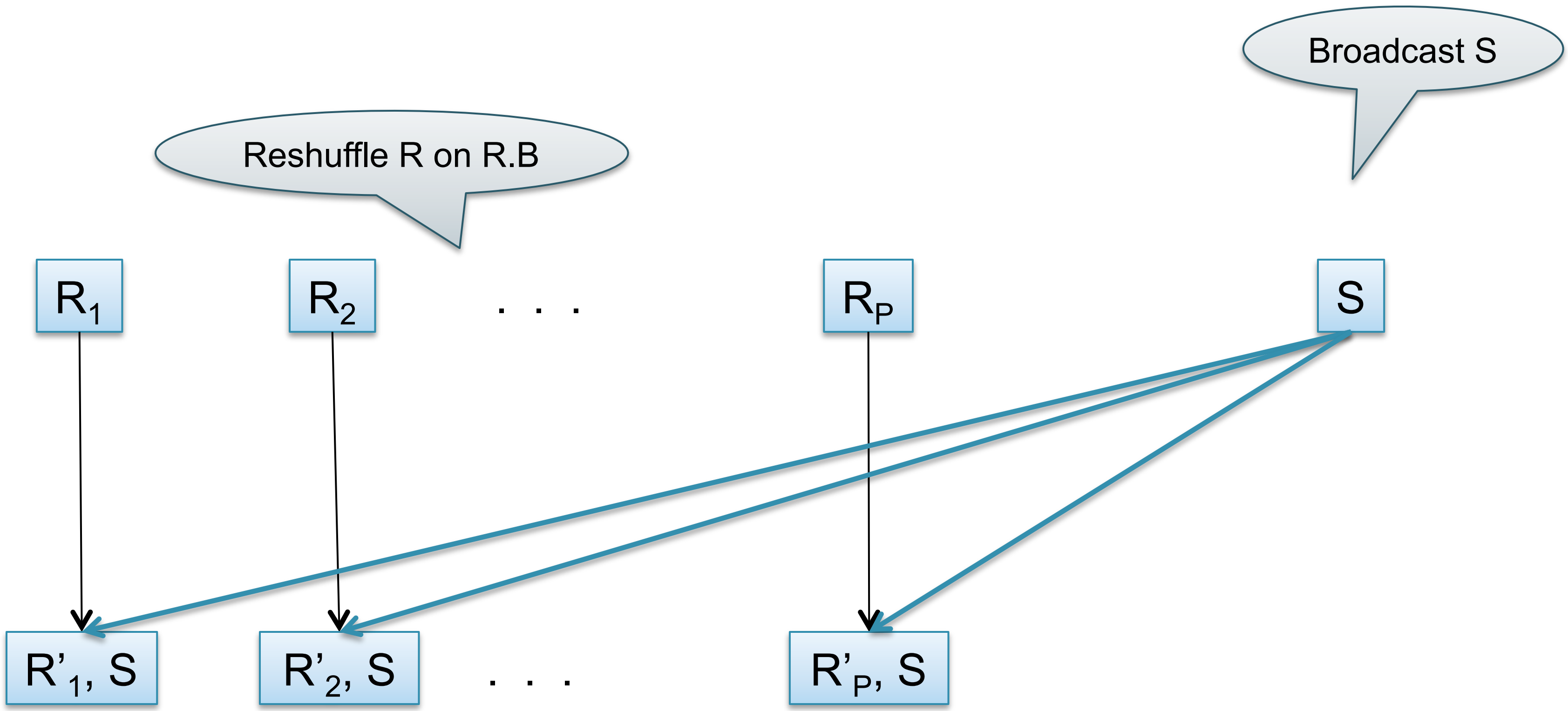
Partitioned Hash-Join

```
map(String value):
    case value.relationName of
        'R': EmitIntermediate(value.B, ('R', value));
        'S': EmitIntermediate(value.C, ('S', value));
```

```
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):
    open(S); /* over the network */
    hashTbl = new()
    for each w in S:
        hashTbl.insert(w.C, w)
    close(S);

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

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

Read entire table S, build a Hash Table

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

Spark

A Case Study of the MapReduce Programming Paradigm

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

The RDD Interface

Collections in Spark

- 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

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.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

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) = 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

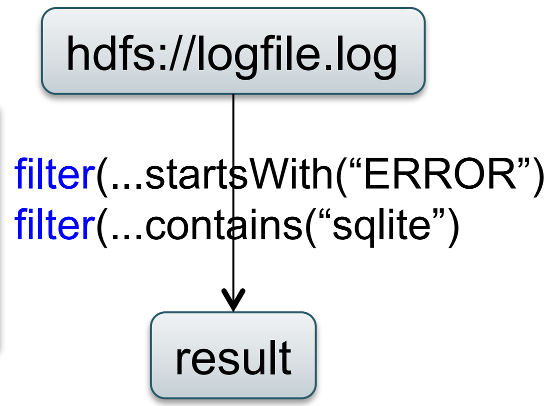
```
lines = s.read().textFile(“hdfs://logfile.log”);
errors = lines.filter(l->l.startsWith(“ERROR”));
sqlerrors = errors.filter(l->l.contains(“sqlite”));
sqlerrors.collect();
```

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();
```

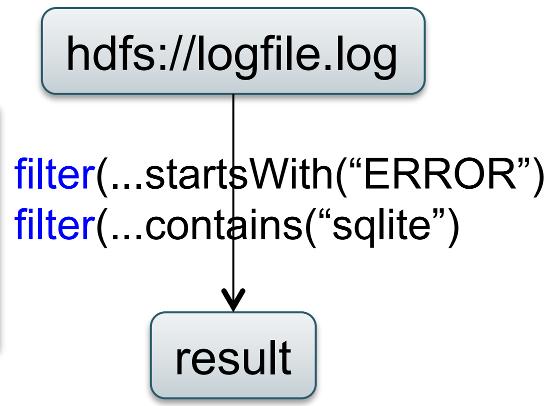
RDD:



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

Persistence

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

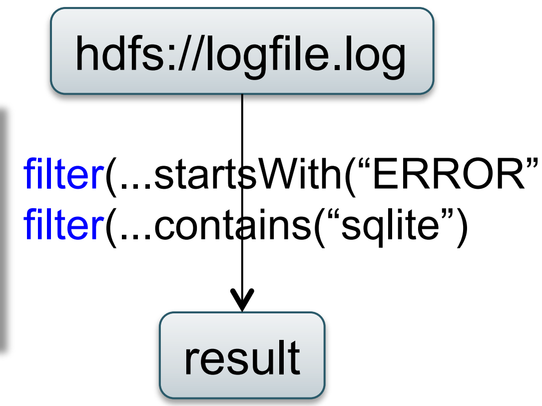
```
lines = s.read().textFile("hdfs://logfile.log");
errors = lines.filter(l->l.startsWith("ERROR"));
errors.persist();
sqlerrors = errors.filter(l->l.contains("sqlite"));
sqlerrors.collect()
```

New RDD

Spark can recompute the result from errors

Persistence

RDD:

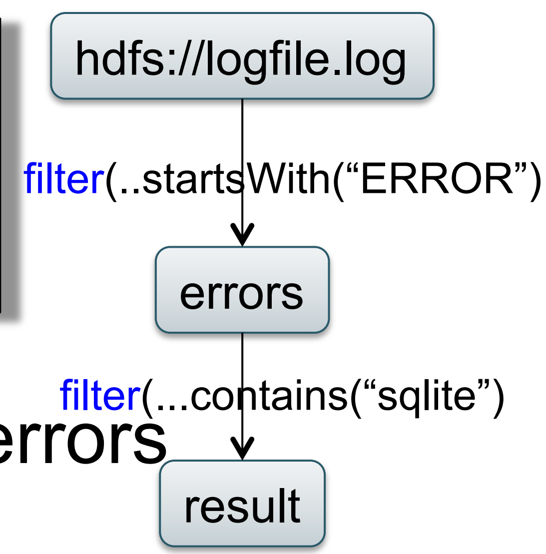



```
lines = s.read().textFile("hdfs://logfile.log");
errors = lines.filter(l->l.startsWith("ERROR"));
sqlerrors = errors.filter(l->l.contains("sqlite"));
sqlerrors.collect();
```

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();
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 = 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)

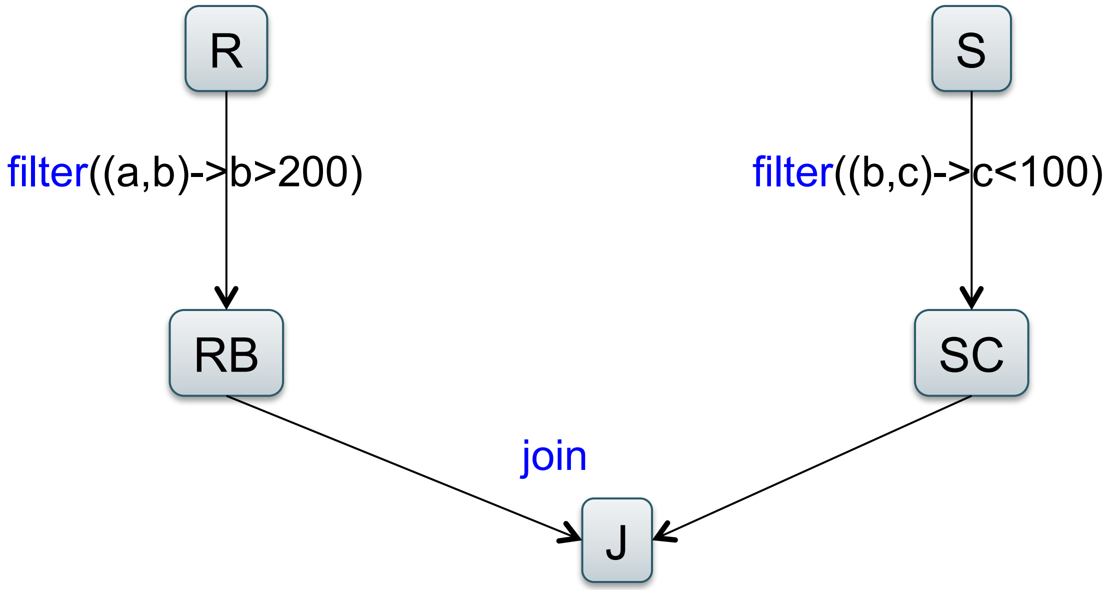
```
SELECT count(*)  FROM R, S
WHERE R.B > 200 and S.C < 100  and R.A = S.A
```

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

- 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>
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 HW4

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