DATA516/CSED516 Scalable Data Systems and Algorithms

Lecture 4

Spark, MapReduce, Hive

Announcements

- HW1 is graded and posted (thanks Kexuan!)
- Project proposals due this Friday!
  - Working in team? Only one of you submits
- HW2 (Spark) due on Monday

Distributed or Parallel Query Processing

- Clusters:
  - More servers → more in main memory
  - More servers → more computing power
  - Clusters are now cheaply available in the cloud
  - *Distributed* query procesing
- Multicores:
  - The end of Moore's law
  - *Parallel* query processing

Outline

- Spark
- MapReduce and critique
- Fault Tolerance
- Hive (short)

Next lecture: Parallel databases

Spark

Motivation

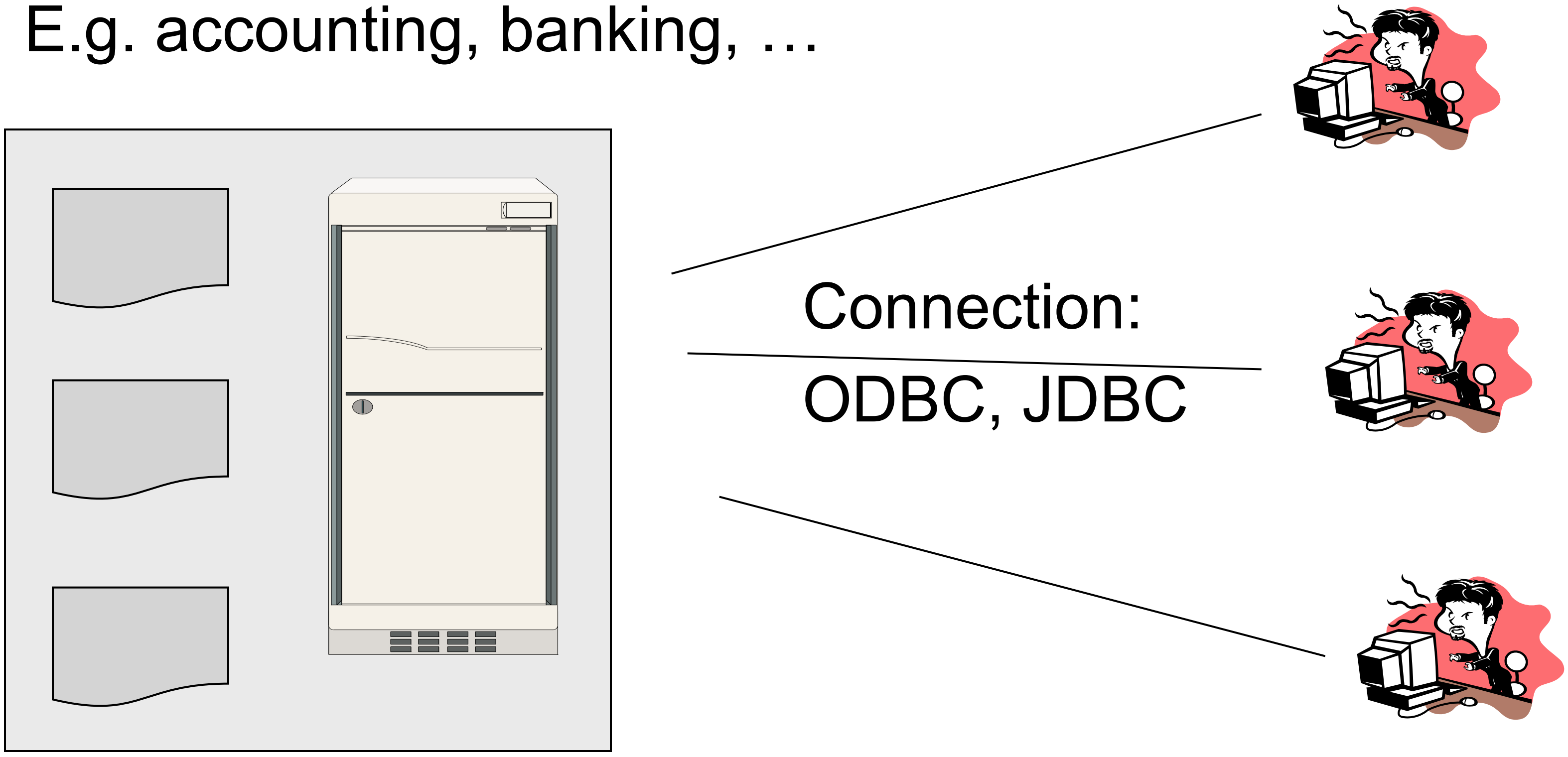
- Limitations of relational database systems:
  - Single server (at least traditionally)
  - SQL is a limited language (eg no iteration)
- Spark:
  - Distributed system
  - Functional language (Java/Scala) good for ML
- Implementation:
  - Extension of MapReduce
  - Distributed physical operators

Review: Single Client

E.g. data analytics

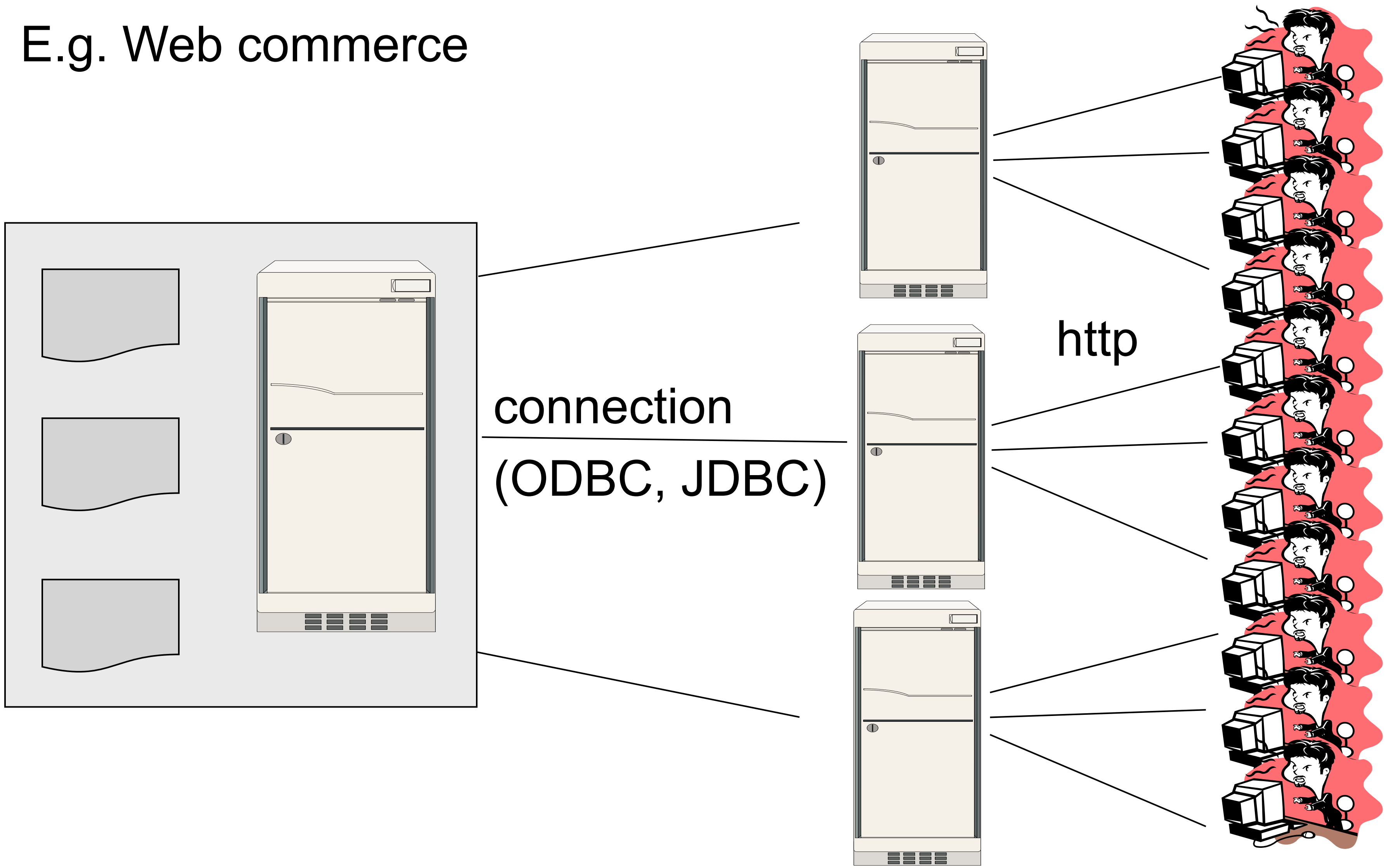
Review: Client-Server

E.g. accounting, banking, …

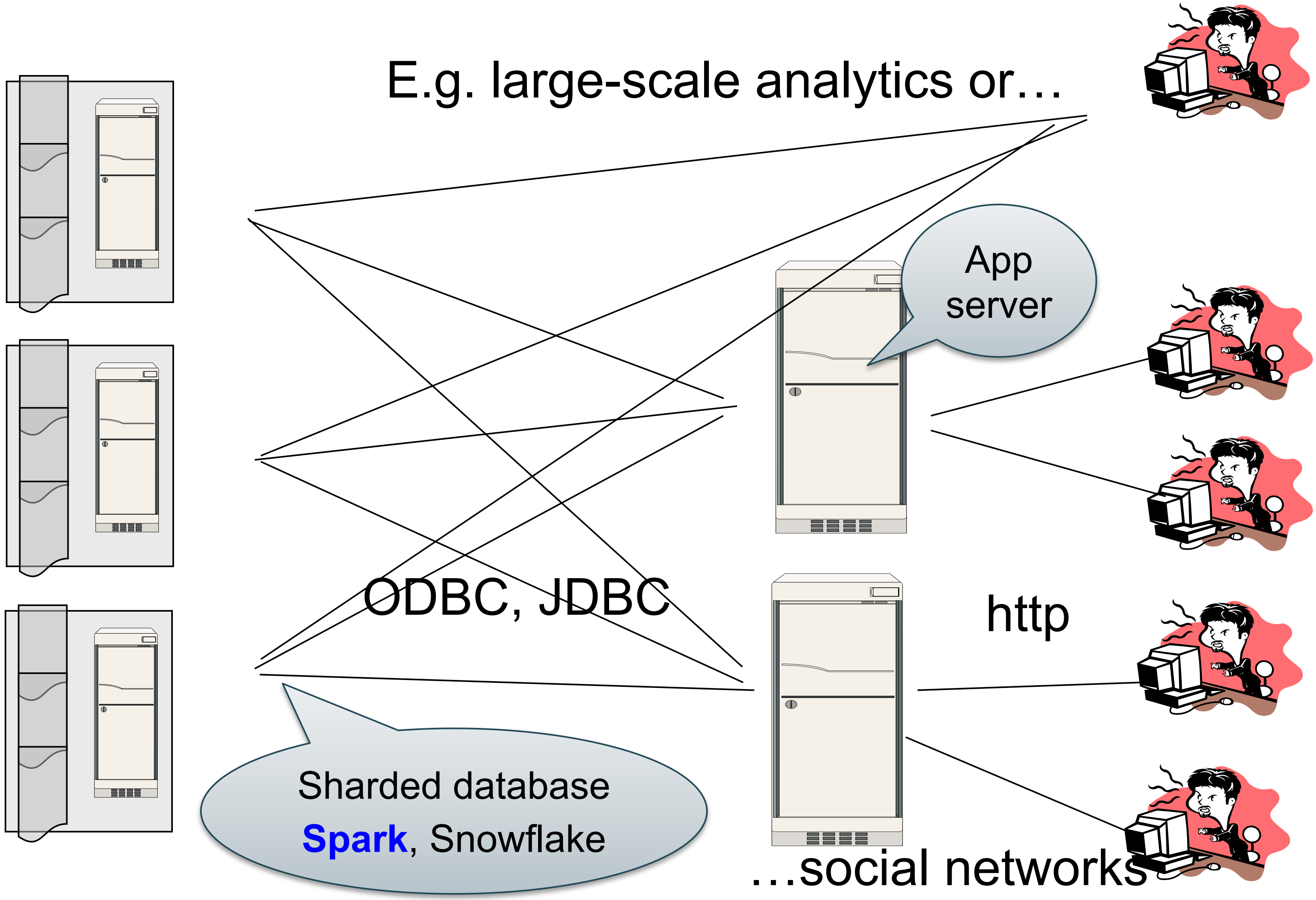


Review: Three-tier

E.g. Web commerce



Review: Distributed Database



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

- Distributed on many servers, not nested
- Operations are done in parallel
- Recoverable via lineage; more later

Seq<T> = a sequence

- Local to one server, may be nested
- Operations are done sequentially

Example from paper, new syntax

Search logs stored in HDFS

```
// First line defines RDD backed by an HDFS file
lines = spark.textFile(“hdfs://…”)

// Now we create a new RDD from the first one
errors = lines.filter(x -> x.startsWith(“Error”))

// Persist the RDD in memory for reuse later
errors.persist()
errors.collect()
errors.filter(x -> x.contains(“MySQL”)).count()
```

Example from paper, new syntax

Search logs stored in HDFS

```
// First line defines RDD backed by an HDFS file
lines = spark.textFile("hdfs://…")

// Now we create a new RDD from the first one
errors = lines.filter(x -> x.startsWith("Error"))

// Persist the RDD in memory for reuse later
errors.persist()
errors.collect()
errors.filter(x -> x.contains("MySQL")).count()
```

Transformation: Not executed yet…

Example from paper, new syntax

Search logs stored in HDFS

```
// First line defines RDD backed by an HDFS file
lines = spark.textFile(“hdfs://…”)

// Now we create a new RDD from the first one
errors = lines.filter(x -> x.startsWith(“Error”))

// Persist the RDD in memory for reuse later
errors.persist()
errors.collect()
errors.filter(x -> x.contains(“MySQL”)).count()
```

Transformation: Not executed yet…

Action: triggers execution of entire program

Anonymous Functions

A.k.a. lambda expressions, starting in Java 8

```
errors = lines.filter(x -> x.startsWith("Error"))
```

Chaining Style

```
sqlerrors = spark.textFile("hdfs://…")
            .filter(x -> x.startsWith("ERROR"))
            .filter(x -> x.contains("sqlite"))
            .collect();
```

Example

The RDD s:

Error…	Warning…	Warning…	Error…	Abort…	Abort…	Error…	Error…	Warning…	Error…

```
sqlerrors = spark.textFile("hdfs://…")
            .filter(x -> x.startsWith("ERROR"))
            .filter(x -> x.contains("sqlite"))
            .collect();
```

Example

The RDD s:

Parallel step 1

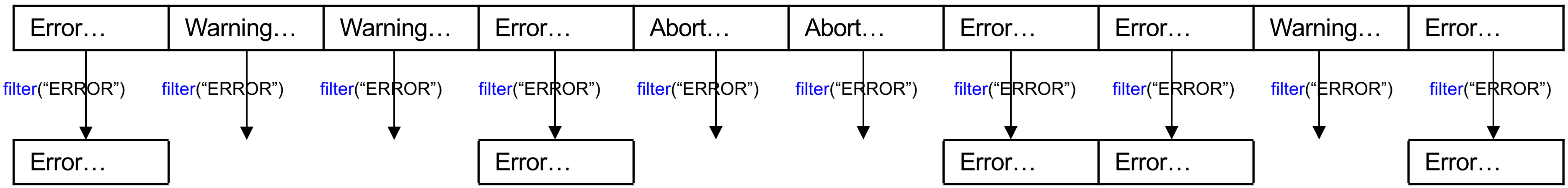
Error…	Warning…	Warning…	Error…	Abort…	Abort…	Error…	Error…	Warning…	Error…
filter("ERROR")	filter("ERROR")	filter("ERROR")	filter("ERROR")	filter("ERROR")	filter("ERROR")	filter("ERROR")	filter("ERROR")	filter("ERROR")	filter("ERROR")

```
sqlerrors = spark.textFile("hdfs://…")
              .filter(x -> x.startsWith("ERROR"))
              .filter(x -> x.contains("sqlite"))
              .collect();
```

Example

Parallel step 1

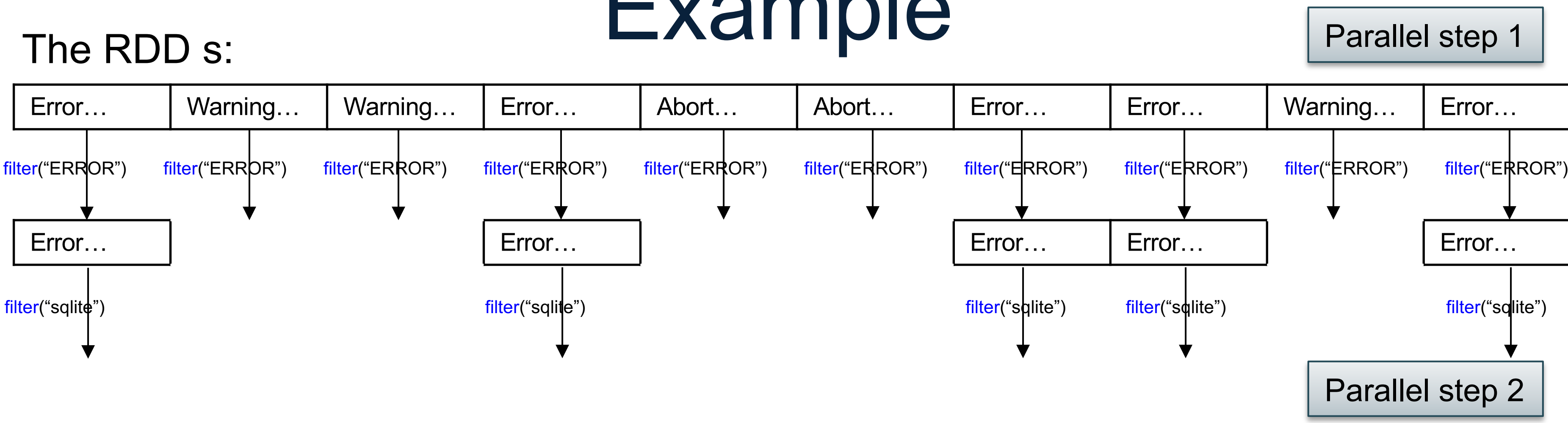
The RDD s:




```
sqlerrors = spark.textFile("hdfs://…")
            .filter(x -> x.startsWith("ERROR"))
            .filter(x -> x.contains("sqlite"))
            .collect();
```

Example

The RDD s:




```
sqlerrors = spark.textFile("hdfs://…")
            .filter(x -> x.startsWith("ERROR"))
            .filter(x -> x.contains("sqlite"))
            .collect();
```

More on Programming Interface

Large set of pre-defined transformations:

- Map, filter, flatMap, sample, groupByKey, reduceByKey, union, join, cogroup, crossProduct, …

Small set of pre-defined actions:

- Count, collect, reduce, lookup, and save

Programming interface includes **iterations**

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

More Complex Example

```
val points = spark.textFile(...)
                  .map(parsePoint).persist()
var w = // random initial vector
for (i <- 1 to ITERATIONS) {
  val gradient = points.map{ p =>
    p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
  }.reduce((a,b) => a+b)
  w -= gradient
}
```

[From Zaharia12]

Spark Ecosystem Growth

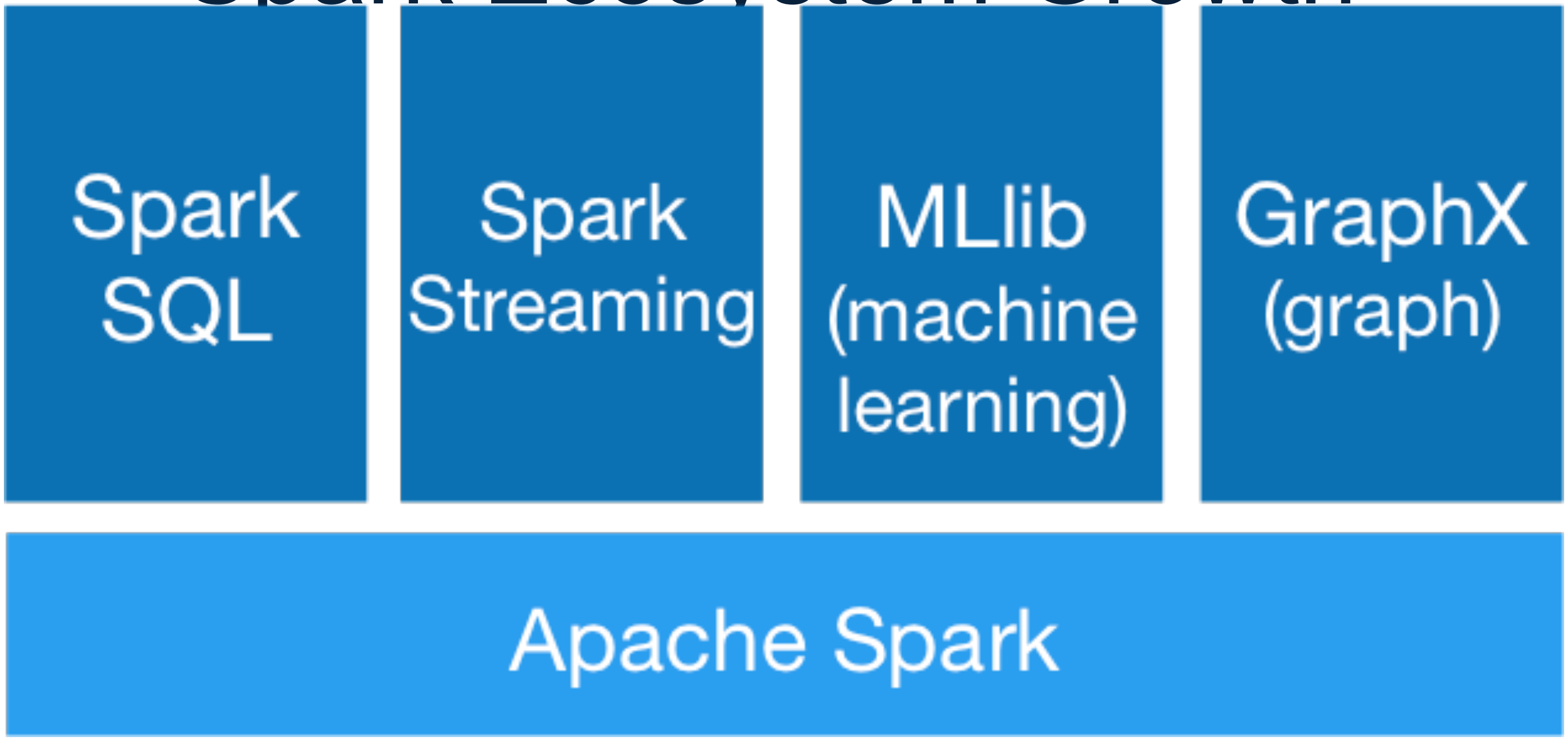


Image from: http://spark.apache.org/

Spark SQL vs Functional Prog. API

- Spark's original functional programming API
  - General
  - But limited opportunities for automatic optimization
- Spark SQL simultaneously
  - Makes Spark accessible to more users
  - Improves opportunities for automatic optimizations

Three Java-Spark APIs

- RDDs: Sytnax: JavaRDD<T>
  - T = anything, basically untyped
  - Distributed, main memory
- Data frames: Dataset<Row>
  - <Row> = a record, dynamically typed
  - Distributed, main memory or external (e.g. SQL)
- Datasets: Dataset<Person>
  - <Person> = user defined type
  - Distributed, main memory (not external)

DataFrames

- Like RDD: immutable distributed collection
- Organized into *named columns*
  - 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

- Like DataFrames, but elements must be typed
- 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)

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?

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

Outline

- Spark
- MapReduce and critique
- Fault Tolerance
- Hive (short)

Next lecture: Parallel databases

MapReduce: References

- Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters. OSDI'04

- D. DeWitt and M. Stonebraker. Mapreduce – a major step backward. In Database Column (Blog), 2008.

MapReduce

- Google:
  - Started around 2000
  - Paper published 2004
  - Discontinued September 2019
- Free variant: Hadoop

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

Distributed File System (DFS)

- For very large files: TBs, PBs
- Each file partitioned into *chunks* (64MB)
- Each chunk replicated (≥3 times) – why?
- Implementations:
  - Google's DFS: GFS, proprietary
  - Hadoop's DFS: HDFS, open source

MapReduce

- Describe the **input** and **output** to map reduce

- Describe the **Map** function

- Describe the **Reduce** function

MapReduce

- Describe the **input** and **output** to map reduce
  - Input: a bag of `(inputkey, value)` pairs
  - Output: a bag of `(outputkey, value)` pairs
- Describe the **Map** function

- Describe the **Reduce** function

MapReduce

- Describe the **input** and **output** to map reduce
  - Input: a bag of `(inputkey, value)` pairs
  - Output: a bag of `(outputkey, value)` pairs
- Describe the **Map** function
  - Input: `(input key, value)`
  - Ouput: bag of `(intermediate key, value)`
- Describe the **Reduce** function

MapReduce

- Describe the **input** and **output** to map reduce
  - Input: a bag of `(inputkey, value)` pairs
  - Output: a bag of `(outputkey, value)` pairs
- Describe the **Map** function
  - Input: `(input key, value)`
  - Ouput: bag of `(intermediate key, value)`
- Describe the **Reduce** function
  - Input: `(intermediate key, bag of values)`
  - Output: bag of output `(values)`

Step 1: the MAP Phase

User provides the MAP-function:

- Input: `(input key, value)`
- Ouput: bag of `(intermediate key, value)`

System applies the map function in parallel to all `(input key, value)` pairs in 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)

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");
```

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));
```

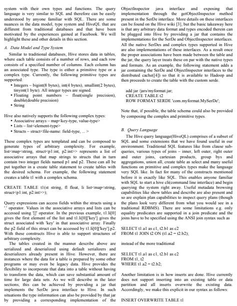
Think “Relational”!

Documents: Relation

did1

Hive – A Petabyte Scale Data Warehouse Using Hadoop

did2



. . .

Think "Relational"!

Documents:

did1


Hive – A Petabyte Scale Data Warehouse Using Hadoop

did2

. . .

Relation

Did	Word
did1	Scalable
did1	analysis
did1	on
did1	large
did1	…
did2	system
did2	with
…	

Think "Relational"!

```
select     word, count(*)
from       Data
group by   word
```

Relation

Did	Word
did1	Scalable
did1	analysis
did1	on
did1	large
did1	…
did2	system
did2	with
…	

Think "Relational"!

```
select     word, count(*)
from       Data
group by   word
```

map = group by

reduce = count(…) (or sum(…) or…)

Relation

Did	Word
did1	Scalable
did1	analysis
did1	on
did1	large
did1	…
did2	system
did2	with
…	

Think “Relational”!

```
select     word, count(*)
from       Data
group by   word
```

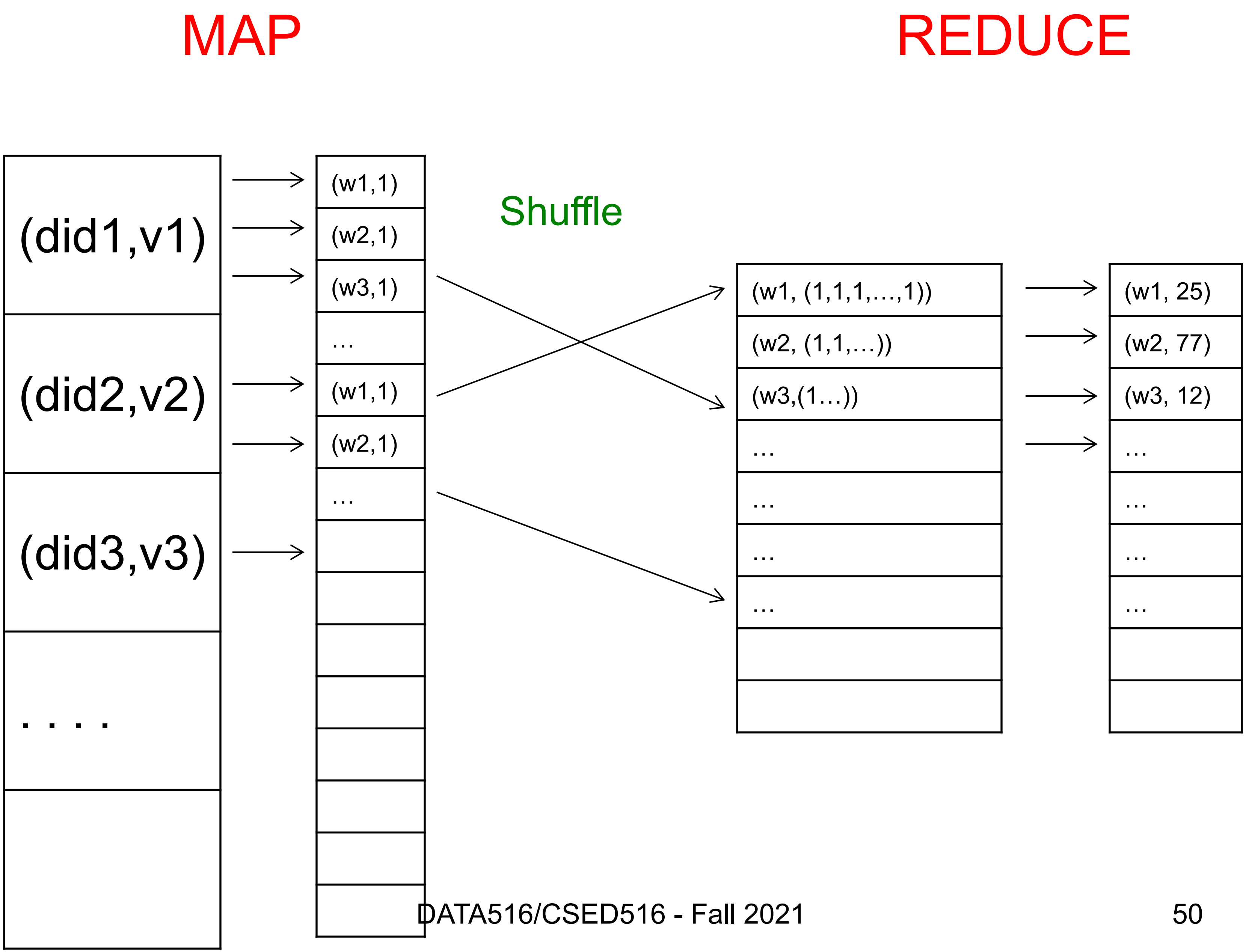
Relation

Did	Word
did1	Scalable
did1	analysis
did1	on
did1	large
did1	…
did2	system
did2	with
…	

map = group by

reduce = count(…) (or sum(…) or…)

MapReduce = Group-by-aggregate


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)
…
…
…
…

Examples from the paper

Discuss in class how to implement in MR

- Distributed grep
- Count URL access frequency: (URL, count)
- Reverse web-link graph: (URL, (list of URLs))
- Inverted index: (word, (list of URLs)

Jobs v.s. Tasks

- A MapReduce Job
  - One simple “query”, e.g. count words in docs
  - Complex queries may require many jobs

- A Map Task, or a Reduce Task
  - A group of instantiations of the map-, or reduce-function, to be scheduled on a single worker

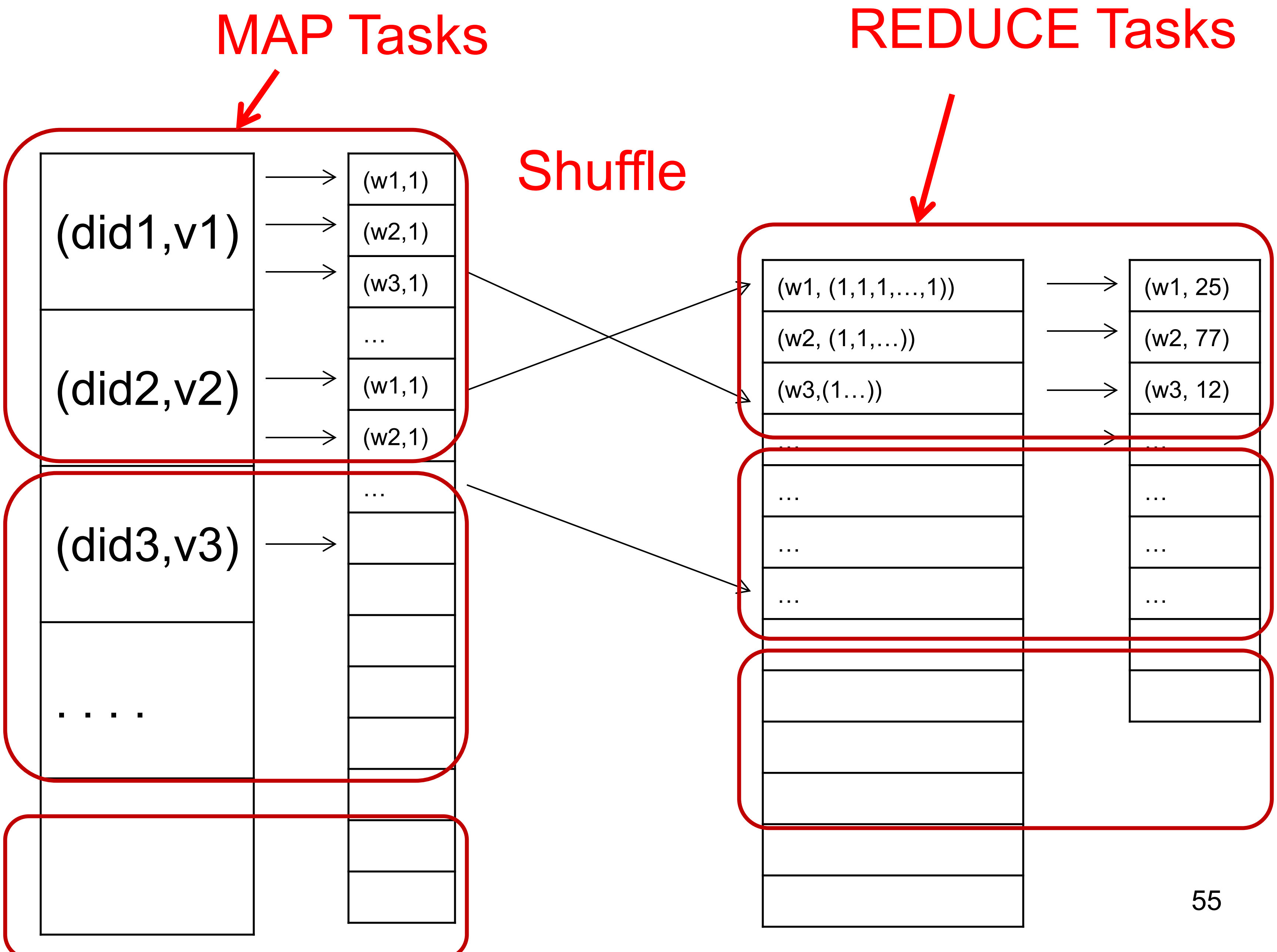
Workers

- 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

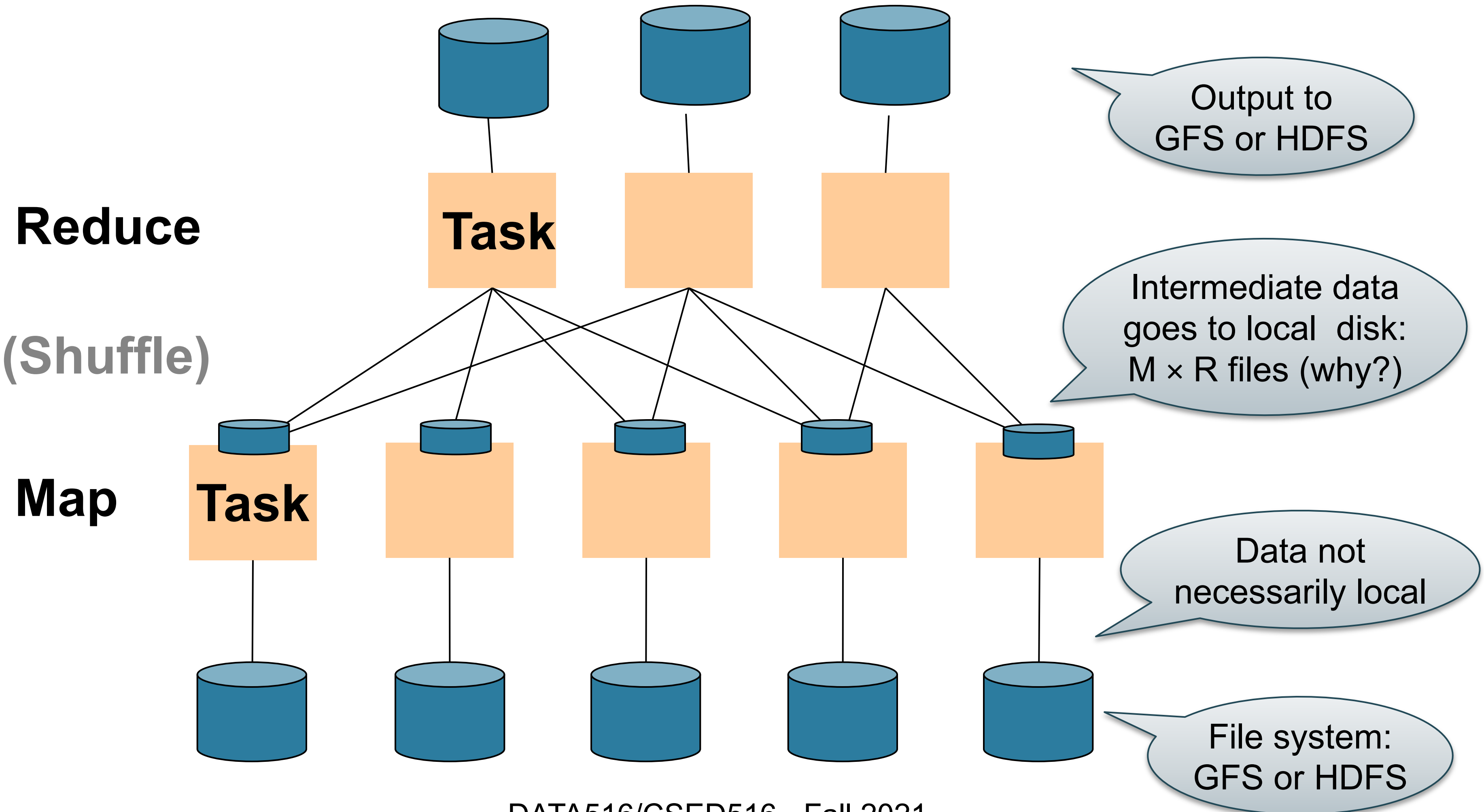


MAP Tasks
REDUCE Tasks
Shuffle
(did1,v1)
(did2,v2)
(did3,v3)
. . . . .
(w1,1)
(w2,1)
(w3,1)
…
(w1,1)
(w2,1)
…
(w1, (1,1,1,…,1))
(w2, (1,1,…))
(w3,(1…))
…
…
…
…
(w1, 25)
(w2, 77)
(w3, 12)
…
…
…
…

Choosing Parameters in MR

- Number of map tasks (M):
  - Default: one map task per chunk
  - E.g. data = 64TB, chunk = 64MB → $M = 10^6$
- Number of reduce tasks (R):
  - No good default; set manually $R << M$
  - E.g. R = 500 or 5000
- In general, MapReduce had very many parameters that required expertise to tune

MapReduce Execution Details



Discussion

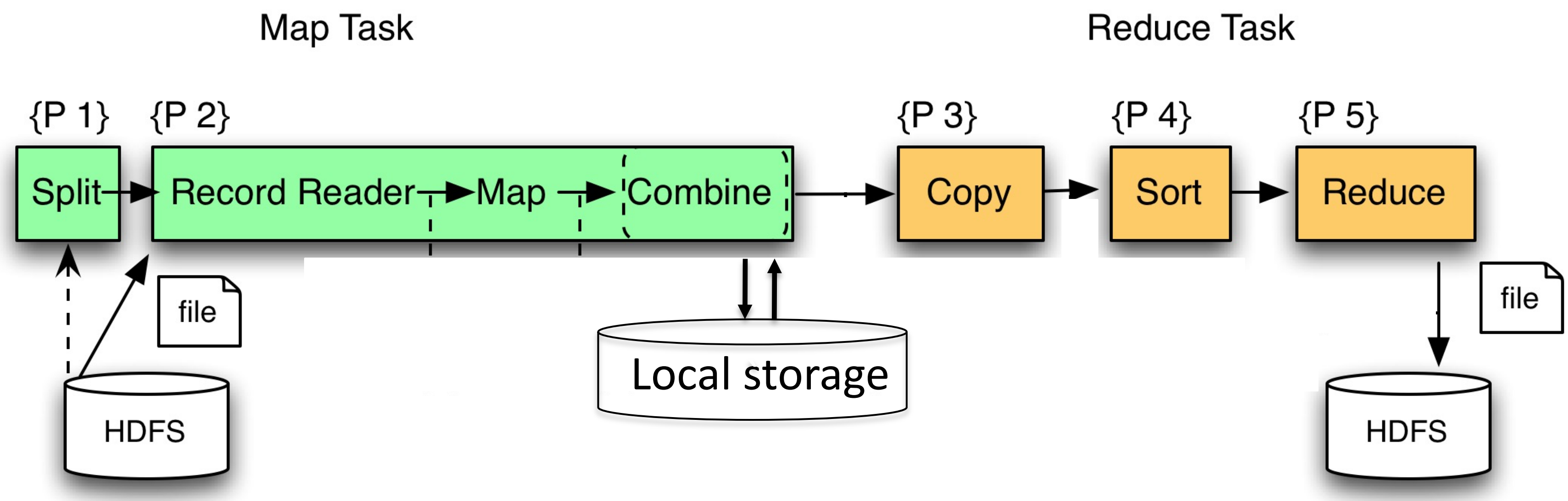
Why doesn't MR determine the number of reduce tasks R dynamically, after all map tasks finish?

Discussion

Why doesn’t MR determine the number of reduce tasks R dynamically, after all map tasks finish?

Because each map tasks needs to write its output into R file; so R must be known before the map tasks start

MapReduce Phases



Riddle

- The combiner function performs an optimization that you already know
- Which one?

Riddle

- The combiner function performs an optimization that you already know
- Which one?

- Pushing aggregates down

Riddle

- The combiner function performs an optimization that you already know
- Which one?

```
Temp=
  select server, word, count(*) as c
  from Data
  group by server, word
```

- Pushing aggregates down:
  - Each mapper groups by word

Riddle

- The combiner function performs an optimization that you already know
- Which one?

Temp=
select server, word, count(*) as c
from **Data**
group by server, word

- Pushing aggregates down:
  - Each mapper groups by word
  - Reducers perform final group-by

Output =
select word, sum(c)
from **Temp**
group by word

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

MapReduce v.s. Databases

Blog by DeWitt and Stonebraker

MapReduce v.s. Databases

Blog by DeWitt and Stonebraker

- “Schemas are good”

MapReduce v.s. Databases

Blog by DeWitt and Stonebraker

- “Schemas are good”
- “Indexes”

MapReduce v.s. Databases

Blog by DeWitt and Stonebraker

- “Schemas are good”
- “Indexes”
- “Skew” (MR mitigates it somewhat, how?)

MapReduce v.s. Databases

Blog by DeWitt and Stonebraker

- “Schemas are good”
- “Indexes”
- “Skew” (MR mitigates it somewhat, how?)
- The M * R problem – what is it?

MapReduce v.s. Databases

Blog by DeWitt and Stonebraker

- “Schemas are good”
- “Indexes”
- “Skew” (MR mitigates it somewhat, how?)
- The M * R problem – what is it?
- “Parallel databases uses push (to sockets) instead of pull” – what’s the point?

Outline

- Spark
- MapReduce and critique
- Fault Tolerance
- Hive (short)

Next lecture: Parallel databases

Fault Tolerance

Fault Tolerance

- Traditional RDBMs:
  - Major concern: recover after failure
  - FT: not a concern

- Massively distributed systems:
  - Probability of failure increases w/ no. of workers and length of job

Fault Tolerance

Example:

- if a server fails once/year…
- … a job with 10000 servers fails once/hour

Fault Tolerance

How is fault tolerance handled in each system?

- **MapReduce**: if a worker fails then

- **Spark**:

Fault Tolerance

How is fault tolerance handled in each system?

- **MapReduce**: if a worker fails then
  - All its completed map tasks need re-executed
  - Its in-progress reduce task needs re-executed

- **Spark**:

Fault Tolerance

How is fault tolerance handled in each system?

- **MapReduce**: if a worker fails then
  - All its completed map tasks need re-executed
  - Its in-progress reduce task needs re-executed: this is possible because the map tasks still have intermediate data on their local disks
- **Spark**:

Fault Tolerance

How is fault tolerance handled in each system?

- **MapReduce**: if a worker fails then
  - All its completed map tasks need re-executed
  - Its in-progress reduce task needs re-executed: this is possible because the map tasks still have intermediate data on their local disks
- **Spark**: will discuss next

Approach

New abstraction: Resilient Distributed Datasets

RDD properties

- Parallel data structure
- Can be persisted in memory
- Fault-tolerant
- Users can manipulate RDDs with rich set of operators

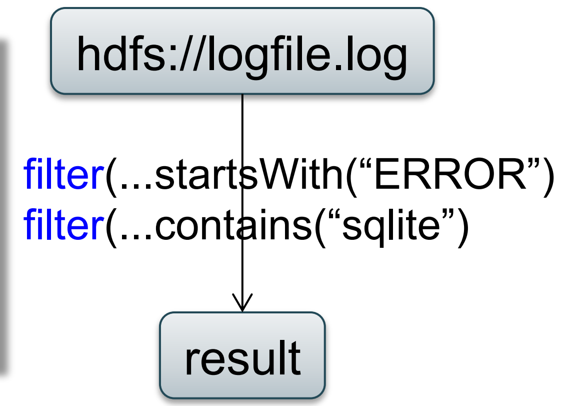
Resilient Distributed Datasets

- RDD = Resilient Distributed Dataset
  - Distributed, immutable.
  - Records 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

RDDs

RDD:

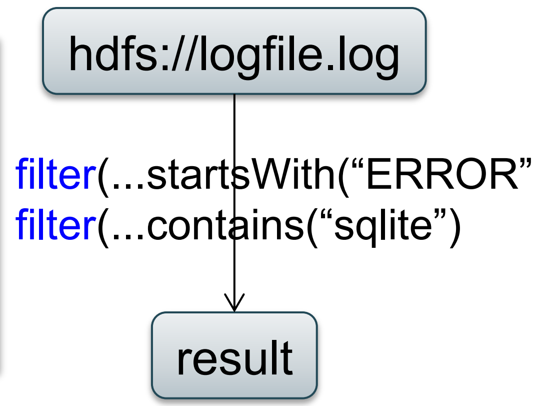
```
lines   = spark.textFile("hdfs://...")
result = lines.filter(l -> l.startsWith("ERROR"))
              .filter(l -> l.contains("sqlite"))
result.collect();
```



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

RDDs

RDD:




```
lines   = spark.textFile("hdfs://...")
result = lines.filter(l -> l.startsWith("ERROR"))
                        .filter(l -> l.contains("sqlite"))
result.collect();
```

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

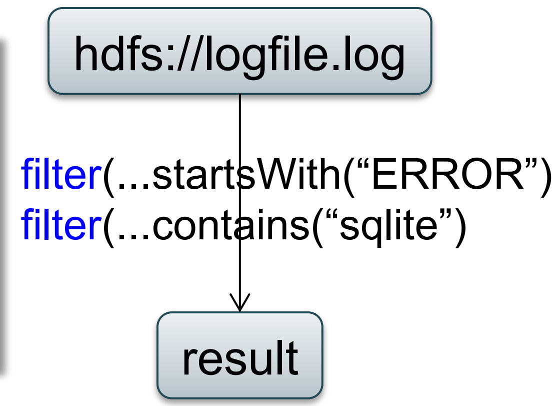
New RDD

```
lines   = spark.textFile("hdfs://...")
errors = lines.filter(l -> l.startsWith("ERROR"))
result = errors.filter(l -> l.contains("sqlite"))
result.collect();
```

RDDs

RDD:

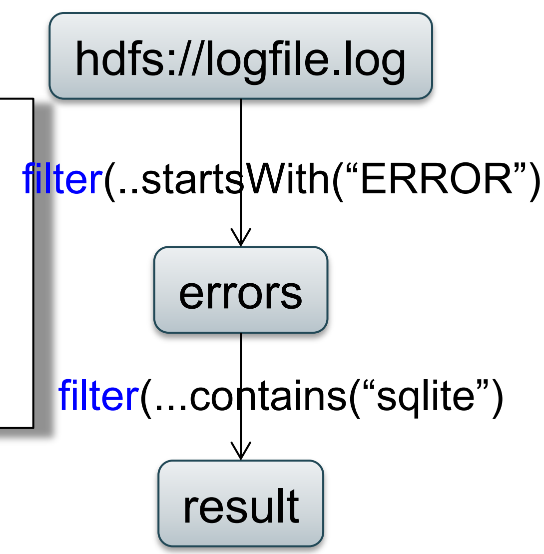
```
lines  = spark.textFile("hdfs://…")
result = lines.filter(l -> l.startsWith("ERROR"))
              .filter(l -> l.contains("sqlite"))
result.collect();
```



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

New RDD

```
lines  = spark.textFile("hdfs://…")
errors = lines.filter(l -> l.startsWith("ERROR"))
result = errors.filter(l -> l.contains("sqlite"))
result.collect();
```



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
in memory
or on disk

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

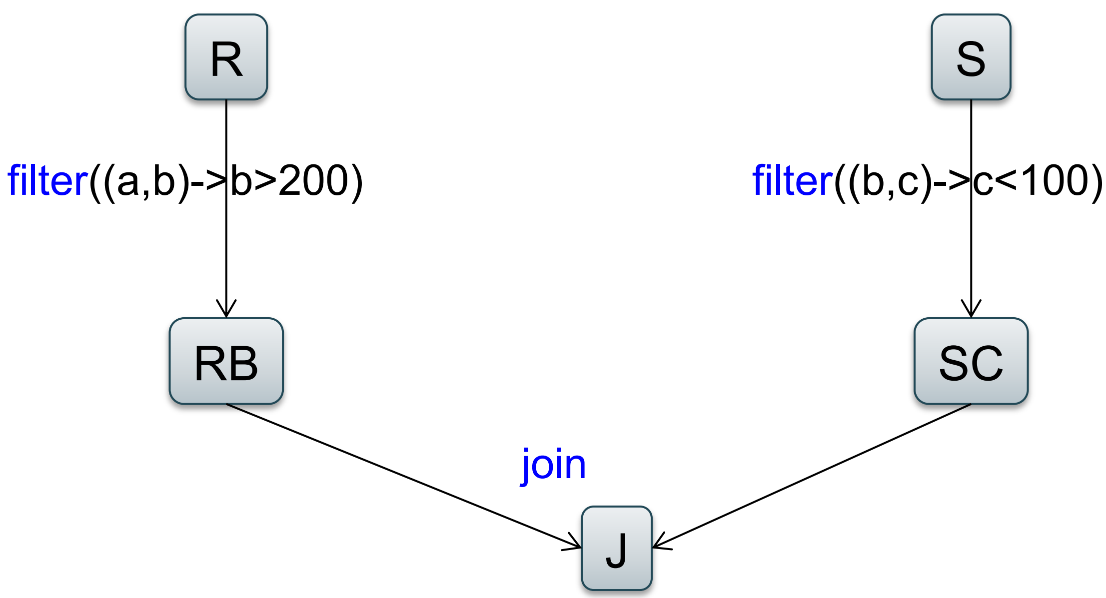
```
SELECT count(*)  FROM R, S
WHERE R.B > 200 and S.C < 100  and R.A = S.A
```

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



RDD Details

- An RDD is a partitioned collection of records
  - RDD's are typed: RDD[Int] is an RDD of integers
  - Records are Java/Python objects
- An RDD is read only
  - This means no updates to individual records
  - This is to contrast with in-memory key-value stores
- To create an RDD
  - Execute a deterministic operation on another RDD
  - Or on data in stable storage
  - Example operations: map, filter, and join

RDD Materialization

- Users control persistence and partitioning

- Persistence
  - Materialize this RDD in memory

- Partitioning
  - Users can specify key for partitioning an RDD

Outline

- Spark
- MapReduce and critique
- Fault Tolerance
- Hive (short)

Next lecture: Parallel databases

Hive

- Facebook's implementation of SQL over MR
- Supports subset of SQL
- Uses MapReduce runtime (pros/cons?)
  - Note: this is similar to Google's FlumeJava

Hive

- Facebook's implementation of SQL over MR
- Supports subset of SQL
- Uses MapReduce runtime (pros/cons?)
  - Note: this is similar to Google's FlumeJava
- Optimizations:

Hive

- Facebook's implementation of SQL over MR
- Supports subset of SQL
- Uses MapReduce runtime (pros/cons?)
  - Note: this is similar to Google's FlumeJava
- Optimizations:
  - Column pruning

Hive

- Facebook’s implementation of SQL over MR
- Supports subset of SQL
- Uses MapReduce runtime (pros/cons?)
  - Note: this is similar to Google’s FlumeJava
- Optimizations:
  - Column pruning
  - Predicate push-down

Hive

- Facebook's implementation of SQL over MR
- Supports subset of SQL
- Uses MapReduce runtime (pros/cons?)
  - Note: this is similar to Google's FlumeJava
- Optimizations:
  - Column pruning
  - Predicate push-down
  - Partition pruning

Hive

- Facebook’s implementation of SQL over MR
- Supports subset of SQL
- Uses MapReduce runtime (pros/cons?)
  - Note: this is similar to Google’s FlumeJava
- Optimizations:
  - Column pruning
  - Predicate push-down
  - Partition pruning
  - Map-side join = ”broadcast join” (discuss in class)

Hive

- Facebook's implementation of SQL over MR
- Supports subset of SQL
- Uses MapReduce runtime (pros/cons?)
  - Note: this is similar to Google's FlumeJava
- Optimizations:
  - Column pruning
  - Predicate push-down
  - Partition pruning
  - Map-side join = ”broadcast join” (discuss in class)
  - Join reordering

Discussion

- Parallel database systems: since the 80s
  - Will discuss next lecture
- MapReduce: around 2000
- Hive: built on MapReuce
- Spark: “better” MapReduce around 2010
- Snowflake, Aurora: cloud, parallel databases; around 2015 (next lecture)

Quick comparison (next slides)

MapReduce v.s. Spark

- Job = Map+Reduce
- Language = Java
- Data = untyped
- Optimization = no

- Job = any query
- Language ≈ RA
- Data = has schema
- Optimization = yes but limited: missing stats on base data

Spark v.s. RDBMS (e.g. Snowflake)

- Query language = its own proprietary
- Optimizer = limited
- Runtime = its own proprietary
- External functions = yes; very useful in ML

- Query language = SQL
- Optimizer = full scale
- Runtime = efficient SQL query engine
- External functions = no