Review: MapReduce and Spark

Alvin Cheung

Spring 2023

Motivation

- We learned how to parallelize relational database systems
- While useful, scaling up relational databases is challenging
- MapReduce is a programming model for such computation
- First, let's study how data is stored in such systems

Distributed File System (DFS)

- For very large files: TBs, PBs
- Each file is partitioned into *chunks*, typically 64MB
- Each chunk is replicated several times (≥3), on different racks, for fault tolerance
- Implementations:
  - Google's DFS: GFS, proprietary
  - Hadoop's DFS: HDFS, open source

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

slide source: Jeff Dean

Data Model

Files!

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

Project 6 anyone?

A MapReduce program:

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

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

System groups all pairs with the same intermediate key, and passes the bag of values to the REDUCE function

User provides the REDUCE function:

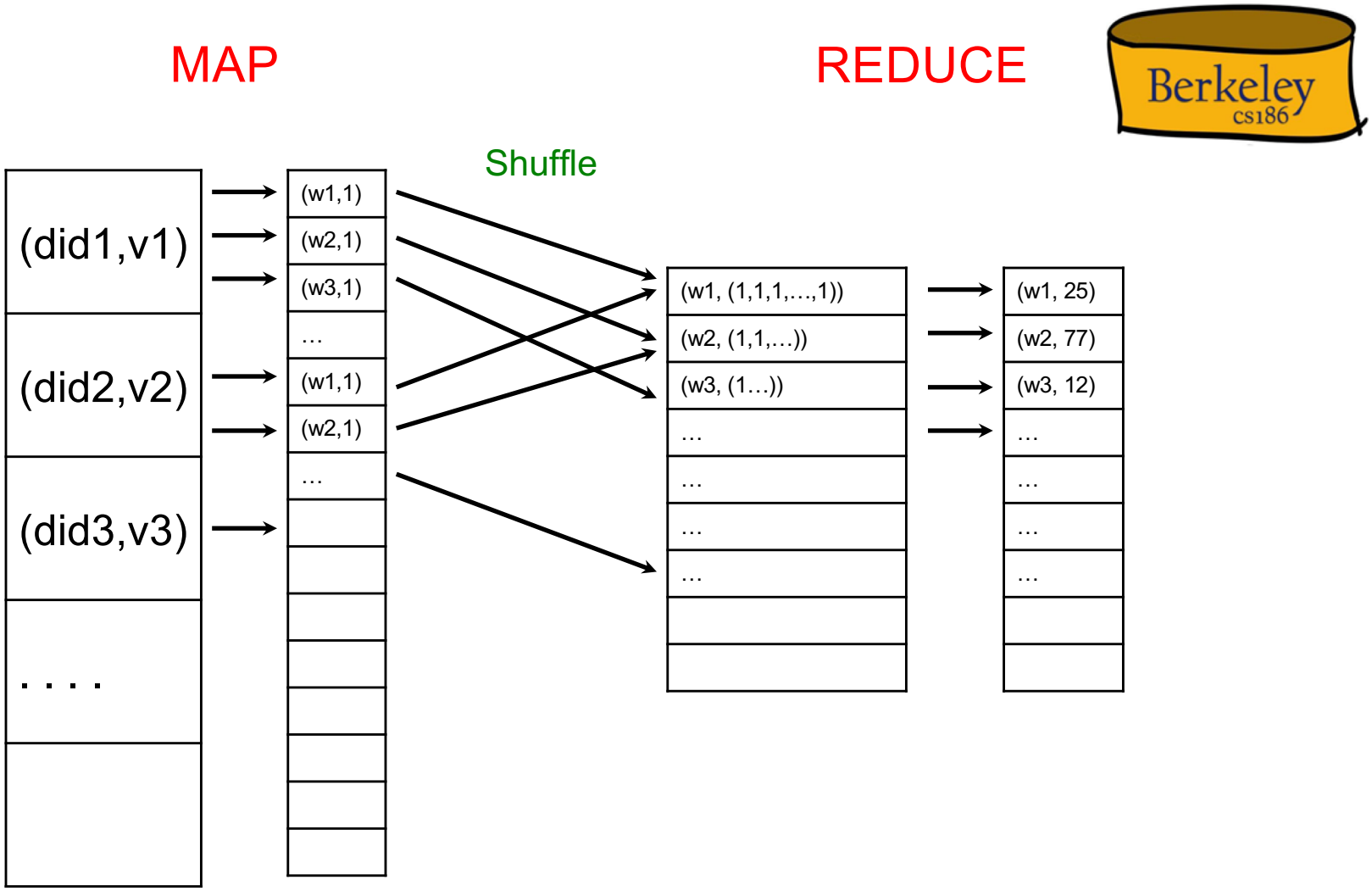
- Input: `(intermediate key, bag of values)`
- Output: bag of output `(values)`

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);
```

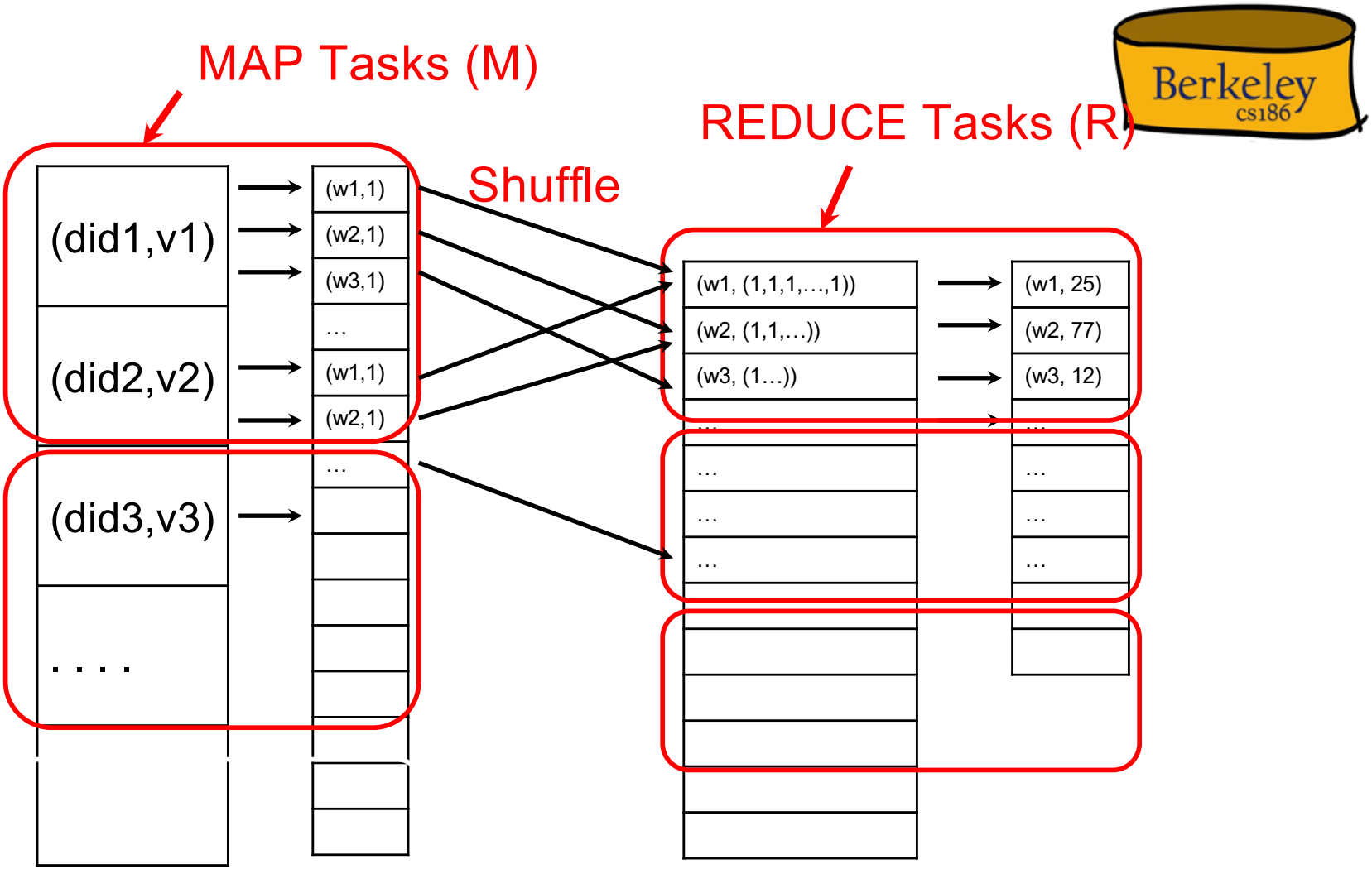
```
reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int sum = 0;
  for each v in values:
      sum += v;
  emit(key, sum);
```


MAP
REDUCE
Berkeley
cs186
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)
…
…
…
…

Workers

- A worker is a process that executes one task at a time

- Typically there is one worker per processor, hence 8 or 16 per node


MAP Tasks (M)
REDUCE Tasks (R)
Shuffle
Berkeley
cs186
(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)
…
…
…
…

Interesting Implementation Details

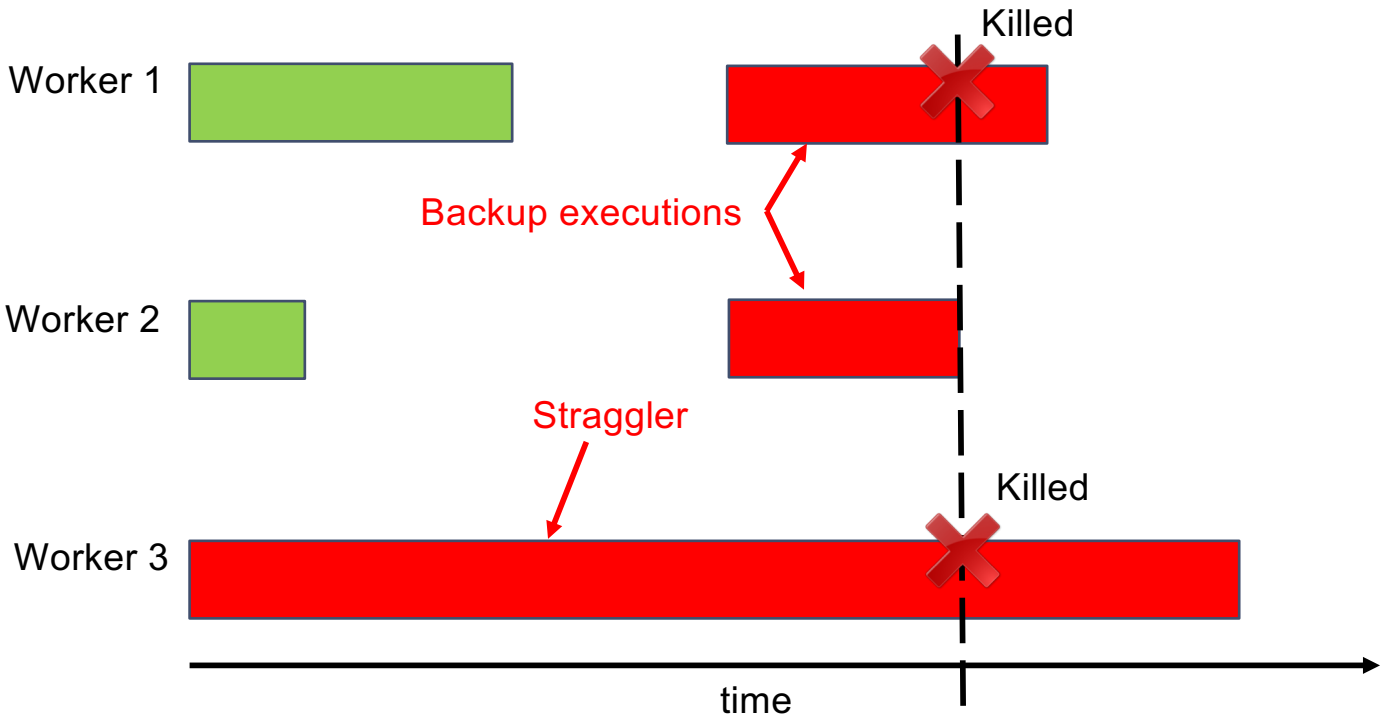
Backup tasks:

- *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:

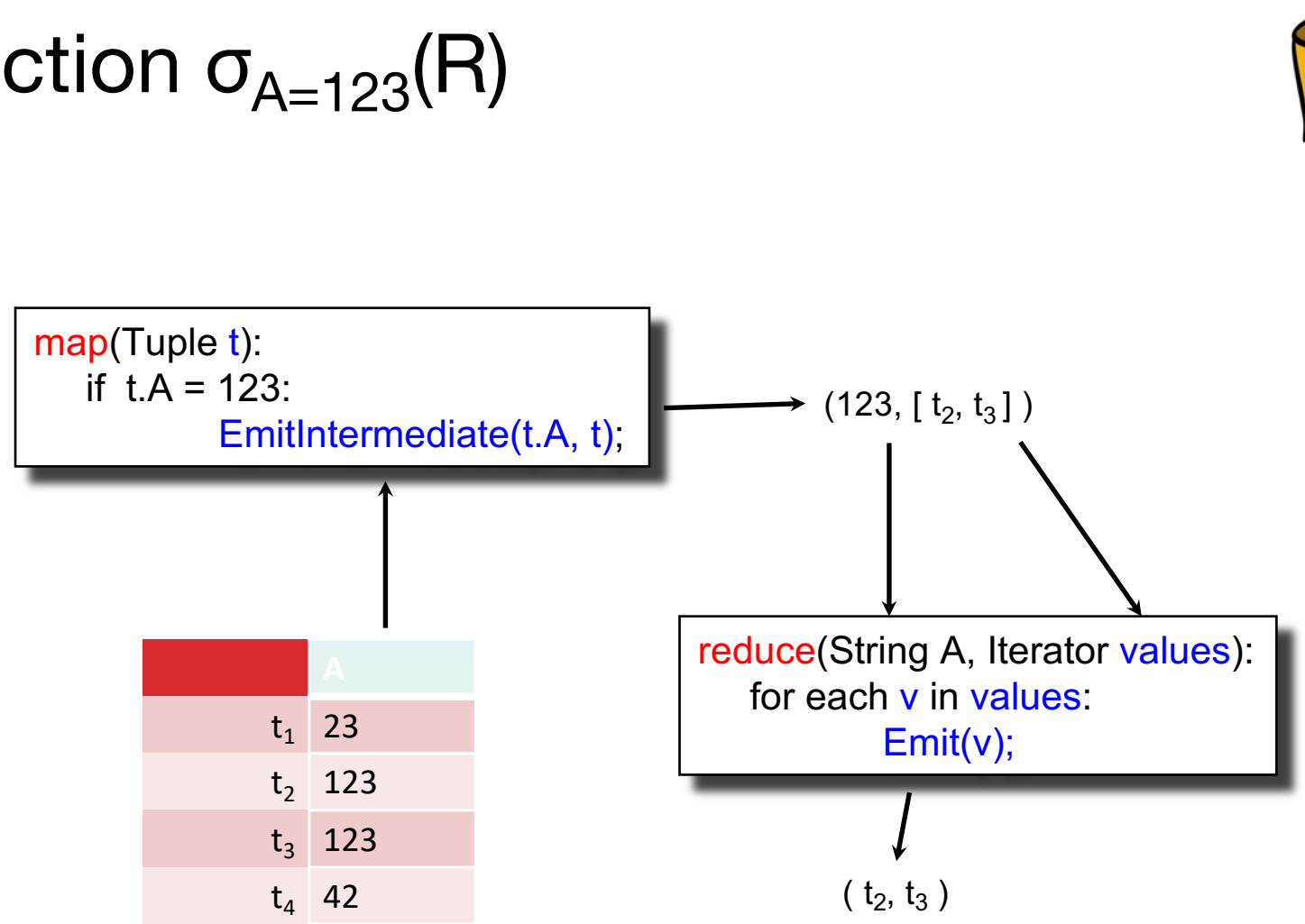
Implementing relational algebra operators in MapReduce

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: $R \bowtie S$

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



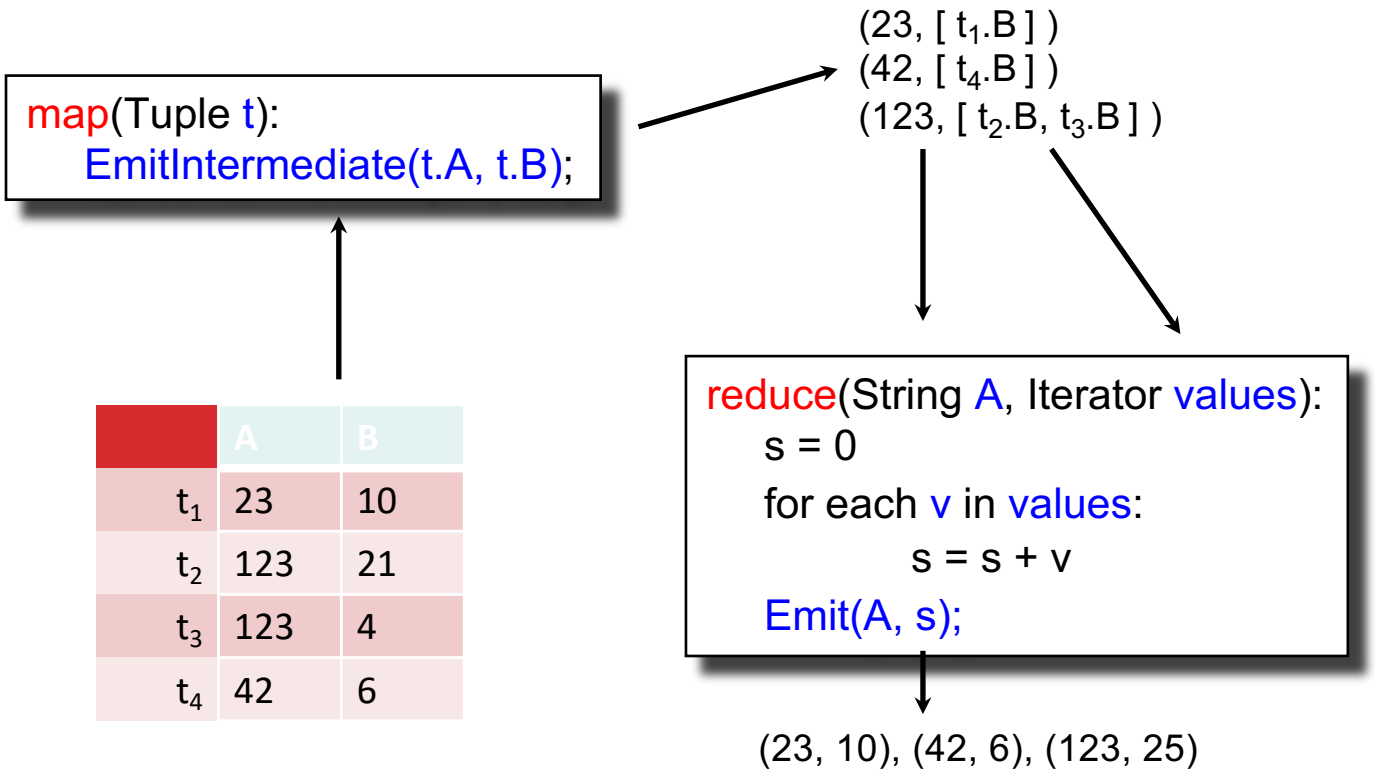
Selection $\sigma_{A=123}(R)$


```
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

Join

Let's implement our parallel join algorithms using MapReduce:

- Partitioned hash-join
- Broadcast join

Partitioned Hash-Join

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


```
map(Tuple t):
    switch (t.relationName):
    case 'R': EmitIntermediate(t.B, t);
    case 'S': EmitIntermediate(t.C, t);
```

R

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

S

	C	D
t_5	4	101
t_6	5	25
t_7	6	4
t_8	21	67

(10, [t_1])
(21, [t_2, t_8])
(4, [t_3, t_5])
(6, [t_4, t_7])
(5, [t_6])

```
reduce(String k, Iterator values):
    R = [ ];  S = [ ];
    for each v in values:
        switch (v.relationName):
        case 'R':   R.append(v)
        case 'S':   S.append(v);
    for v1 in R, for v2 in S
        Emit(v1,v2);
```

(123, 21, 21, 67)
(123, 4, 4, 101)
(42, 6, 6, 4)

Broadcast Join

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


```
map(String value):
    readFromNetwork(S);
    hashTable = new HashTable()
    for each w in S:
            hashTable.insert(w.C, w)

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

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

Read entire table S, build a Hash Table

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

Issues with MapReduce

- Difficult to write more complex queries
  - Everything has to be expressed as map-reduce

- Need multiple MapReduce jobs: dramatically slows down because it writes all (intermediate) results to disk

Spark

- Open source system developed right here!
- Distributed processing over HDFS
- Differences from MapReduce:
  - Multiple steps, including iterations
  - Stores intermediate results in main memory
  - Closer to relational algebra
- Details: http://spark.apache.org

Data Model: Resilient Distributed Datasets

- RDD = Resilient Distributed Datasets
  - A distributed, immutable data set, together with its *lineage*
  - Lineage = expression that says how that data set was computed (e.g., a relational algebra plan)
- Spark stores intermediate results as RDD
- If a server crashes, its RDD in main memory is lost. However, the driver (= leader node) knows the lineage, and will simply recompute the lost partition of the RDD

Programming in Spark

- A Spark program consists of:
  - Transformations (map, reduceByKey, 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                    ();

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

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

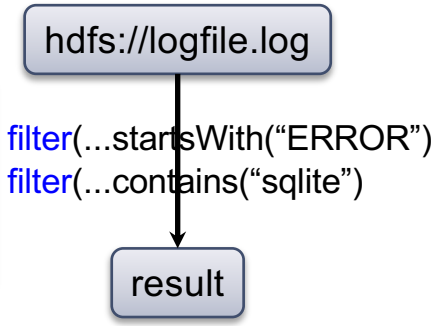

```
lines = s.read().textFile(“hdfs://logfile.log”);
errors = lines.filter(l->l.startsWith(“ERROR”));
sqlerrors = errors.filter(l->l.contains(“sqlite”));
sqlerrors.collect();
```

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

Persistence

RDD:


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

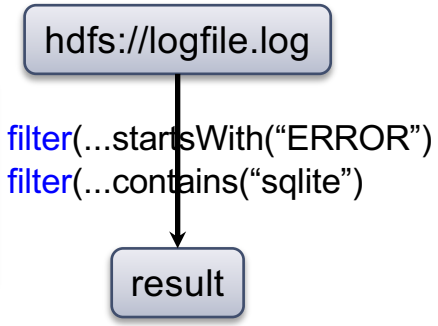


If any server fails before the end, then Spark must restart

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()
```

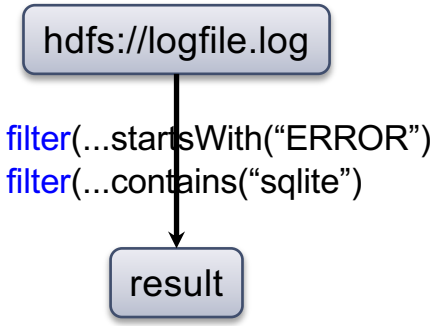
New RDD

Spark can recompute the result from errors

Persistence

RDD:

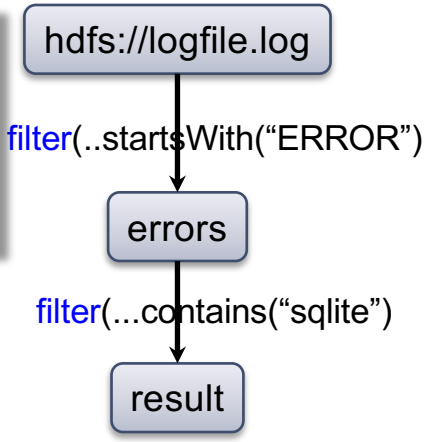

```
lines = s.read().textFile(“hdfs://logfile.log”);
errors = lines.filter(l->l.startsWith(“ERROR”));
sqlerrors = errors.filter(l->l.contains(“sqlite”));
sqlerrors.collect();
```



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

Example

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

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


```
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

Example

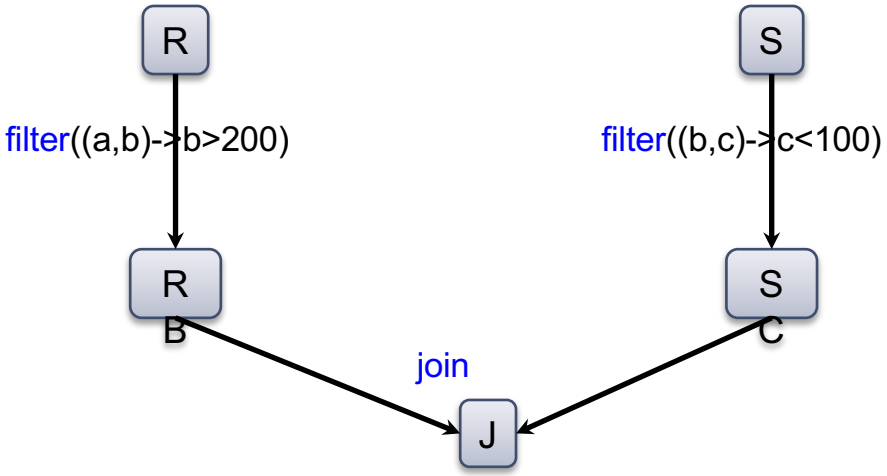
R(A,B)
S(A,C)

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


```
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 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

DataFrames

- Like RDD, also an immutable distributed collection of data
- Organized into *named columns* rather than individual objects
  - Elements are untyped objects called `Row`'s
  - Just like a relation
- 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)

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?

Make sure you understand these!

- MapReduce
  - Why bother?
  - Data model
  - How computation is structured
- Spark
  - What's the problem with MapReduce?
  - Data model
  - What did they add on top of how MR structures its computation?