CS150A Database

Wenjie Wang
School of Information Science and Technology
ShanghaiTech University
Dec. 20, 2024

Today:
- MapReduce and Spark:
    - MapReduce
    - Spark
    - Dataframe and Dataset

Readings:
- Lecture note

Acknowledgement: Alvin Cheung@UCBerkeley's scourse notes

No SQL

Motivations

Data Model

Query Language

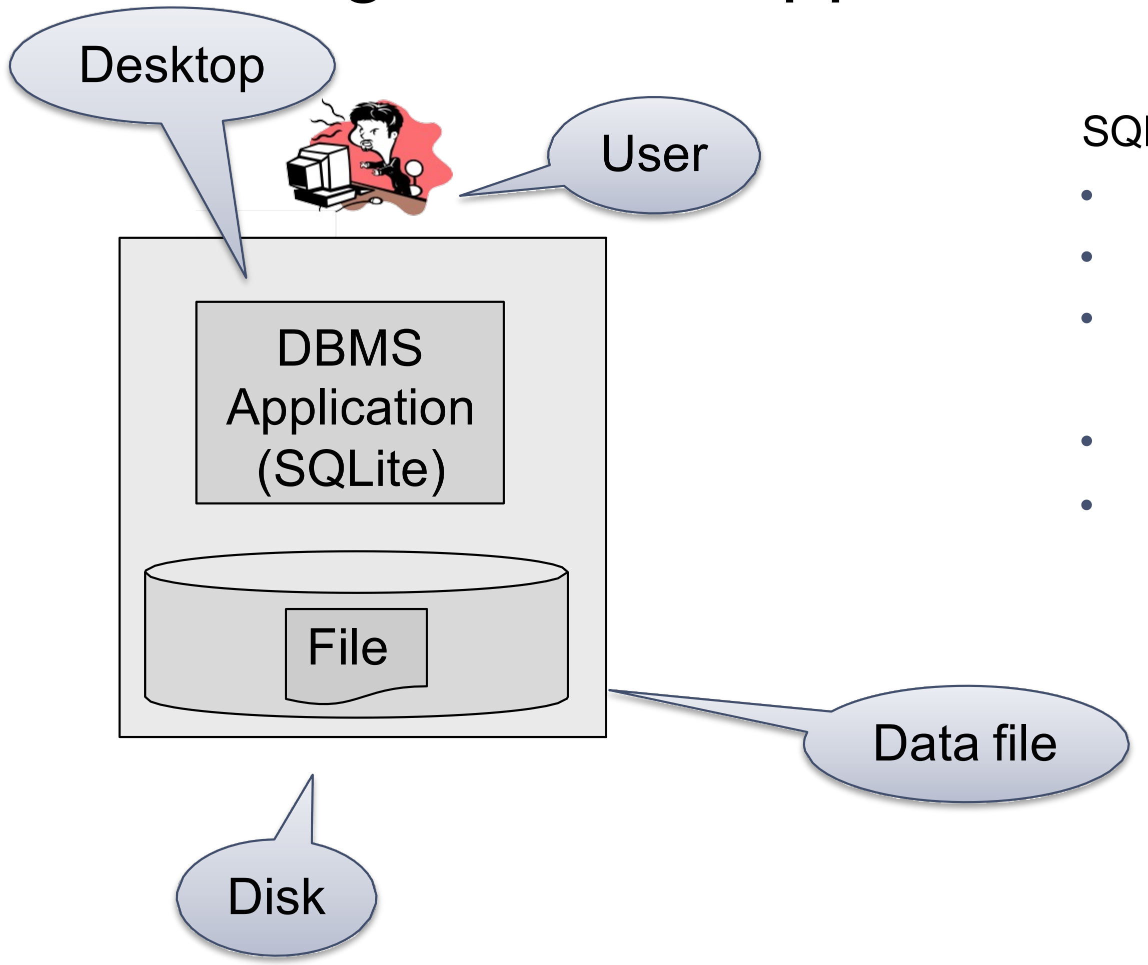
Two Classes of Relational Database Apps

- OLTP (Online Transaction Processing)
  - Queries are simple lookups: 0 or 1 join
    E.g., find customer by ID and their orders
  - Many updates. E.g., insert order, update payment
  - Consistency is critical: we need transactions
- OLAP (Online Analytical Processing)
  - aka “Decision Support”
  - Queries have many joins, and group-by’s
    E.g., sum revenues by store, product, clerk, date
  - No updates

NoSQL Motivation

- Originally motivated by Web 2.0 applications
  - E.g., Facebook, Amazon, Instagram, etc
  - Startups need to scaleup from 10 to 10^7 clients quickly
- Needed: very large scale OLTP workloads
- Give up on consistency, give up OLAP
- NoSQL: reduce functionality
  - Simpler data model
  - Very restricted updates

Structuring RDBMS Apps: “Serverless”

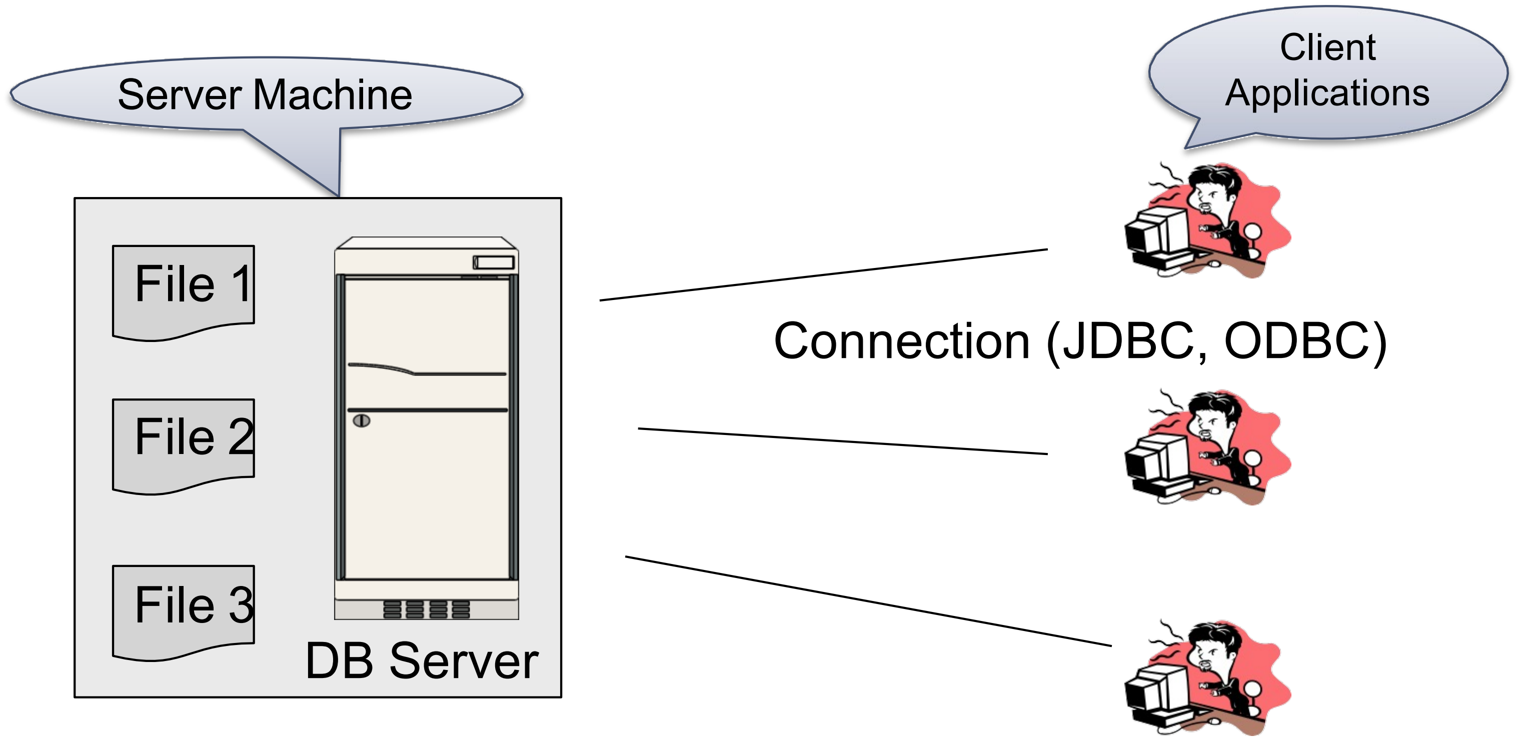


SQLite:

- One data file
- One user
- One DBMS application

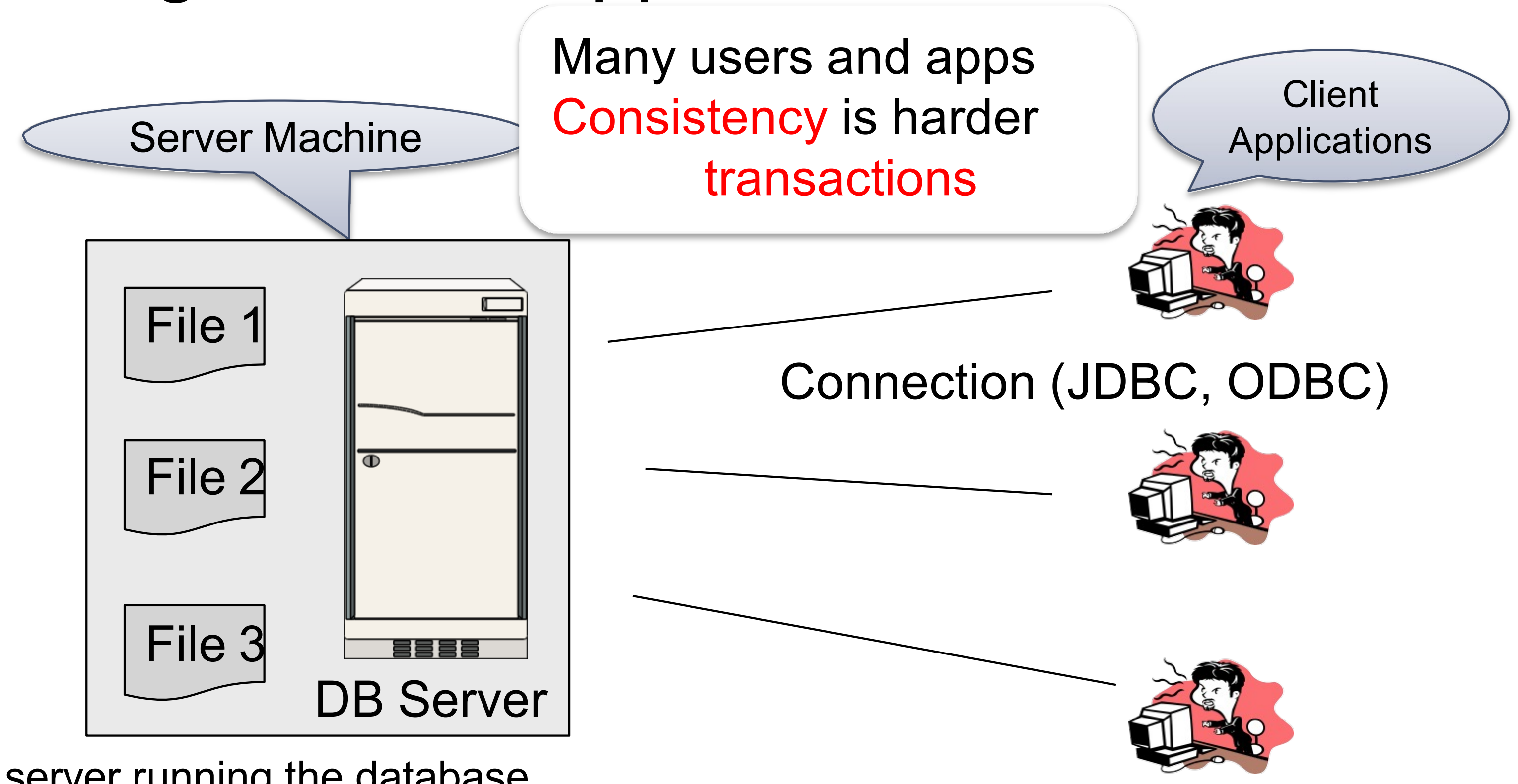
- Consistency is easy
- But only a limited number of scenarios work with such model

Structuring RDBMS Apps: Client-Server



- One server running the database
- Many clients, connecting via the ODBC (Open Database Connectivity) or JDBC (Java Database Connectivity) protocol

Structuring RDBMS Apps: Client-Server

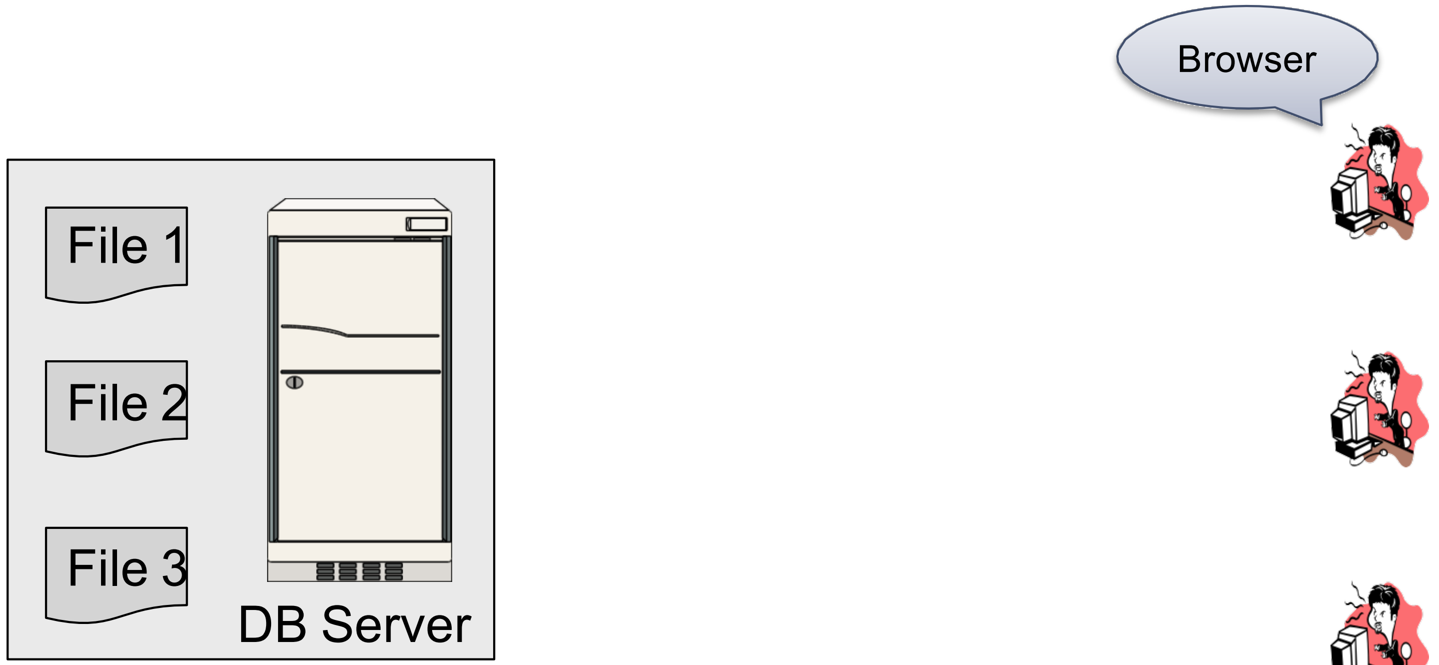


- One server running the database
- Many clients, connecting via the ODBC (Open Database Connectivity) or JDBC (Java Database Connectivity) protocol

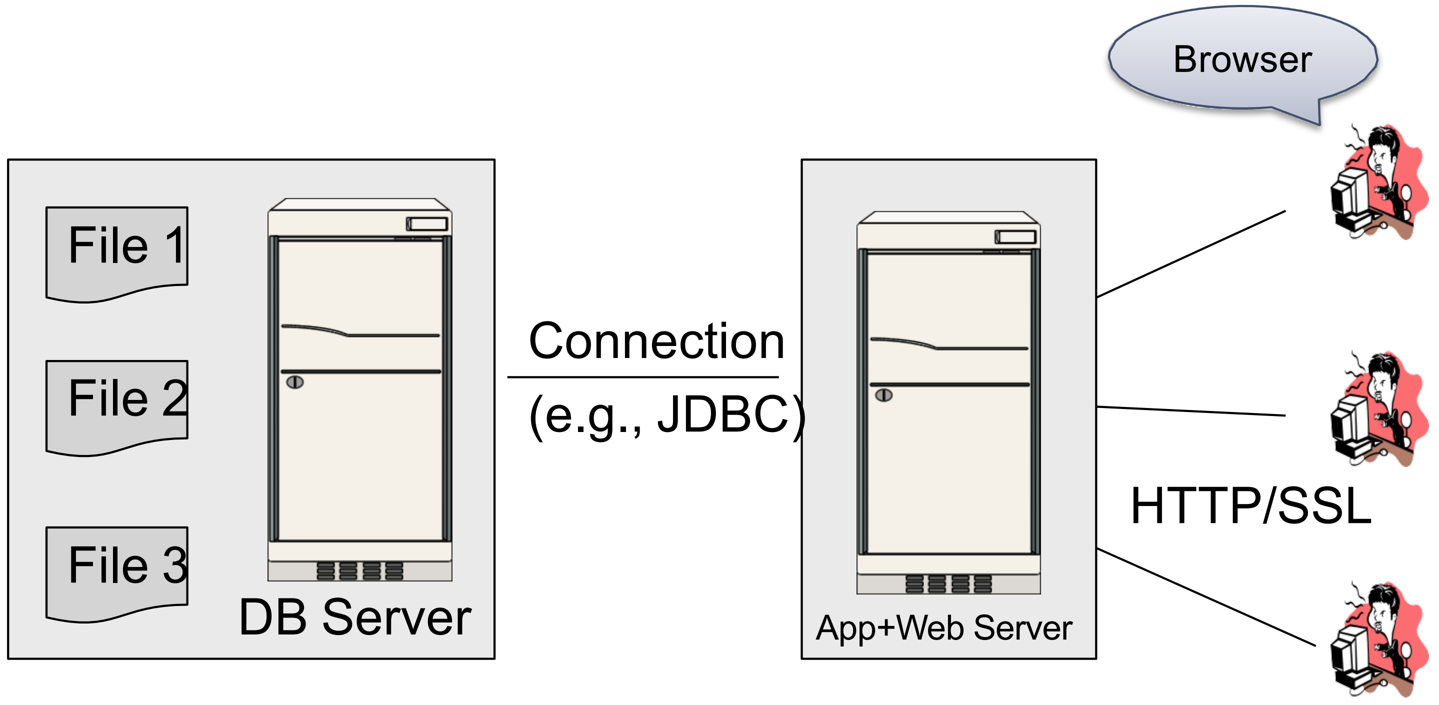
Client-Server

- One *server* that runs the DBMS (or RDBMS):
  - Your own desktop, or
  - Some beefy system, or
  - A cloud service (SQL Azure)
- Many *clients* run apps and connect to DBMS
  - Microsoft’s Management Studio (for SQL Server), or
  - psql (for postgres)
  - Your Java/C++/Python/etc program
- Clients “talk” to server using JDBC/ODBC protocol

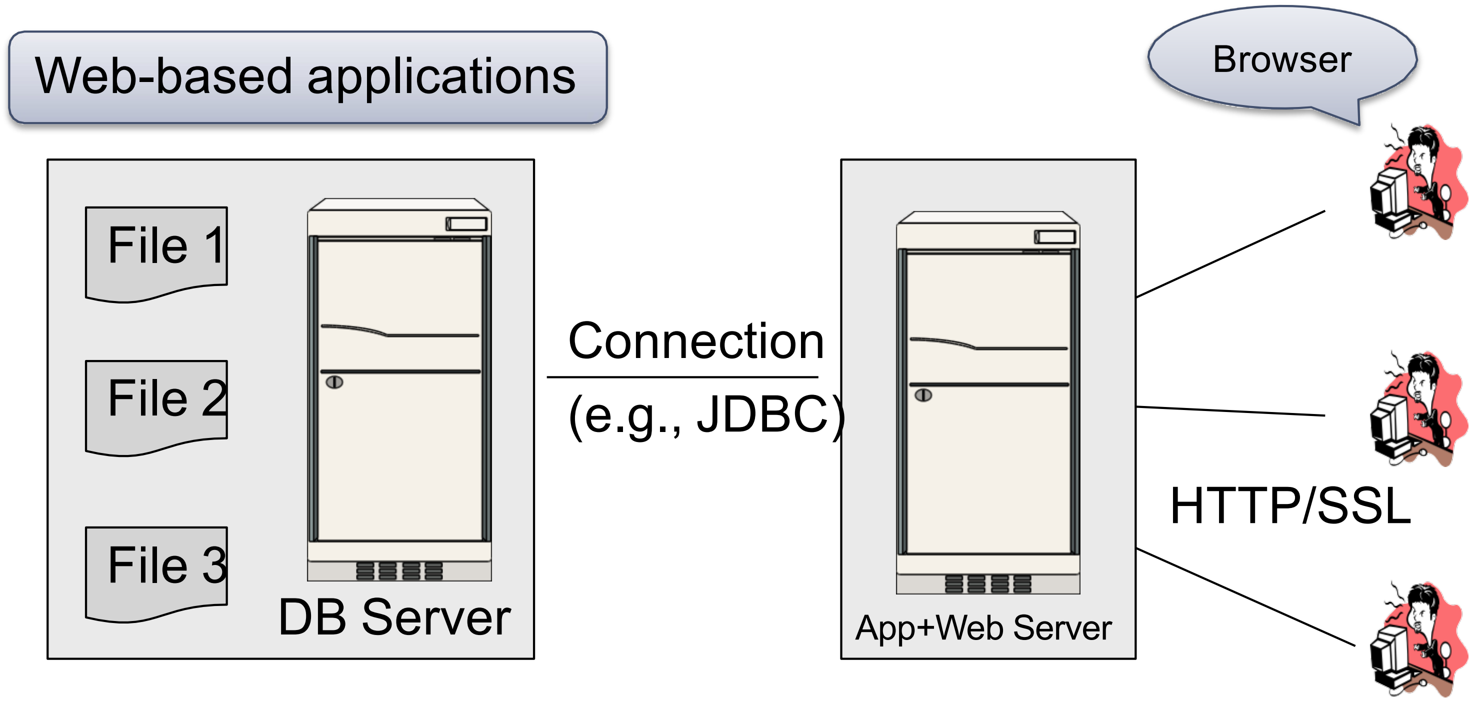
Web Apps: 3 Tier



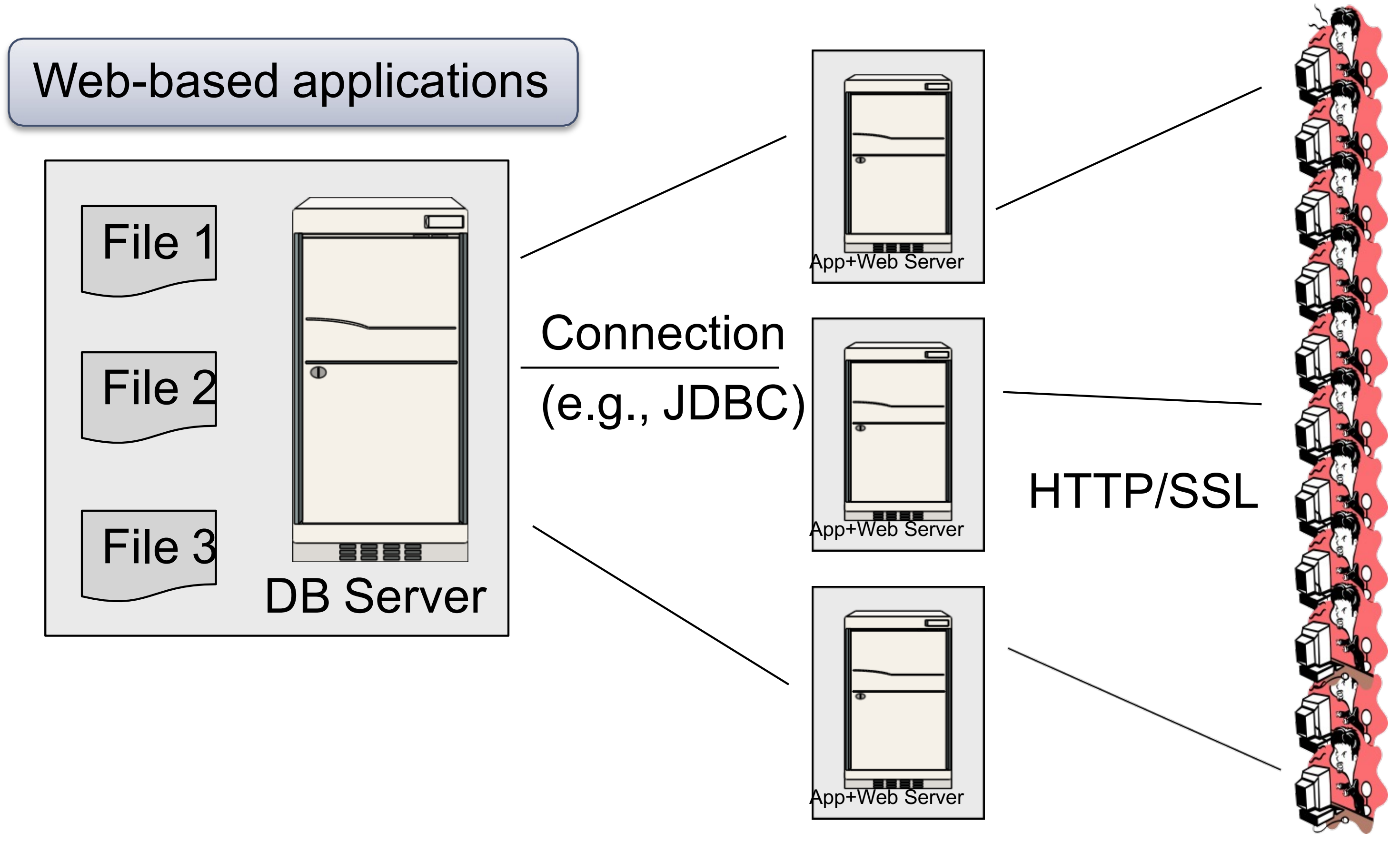
Web Apps: 3 Tier



Web Apps: 3 Tier



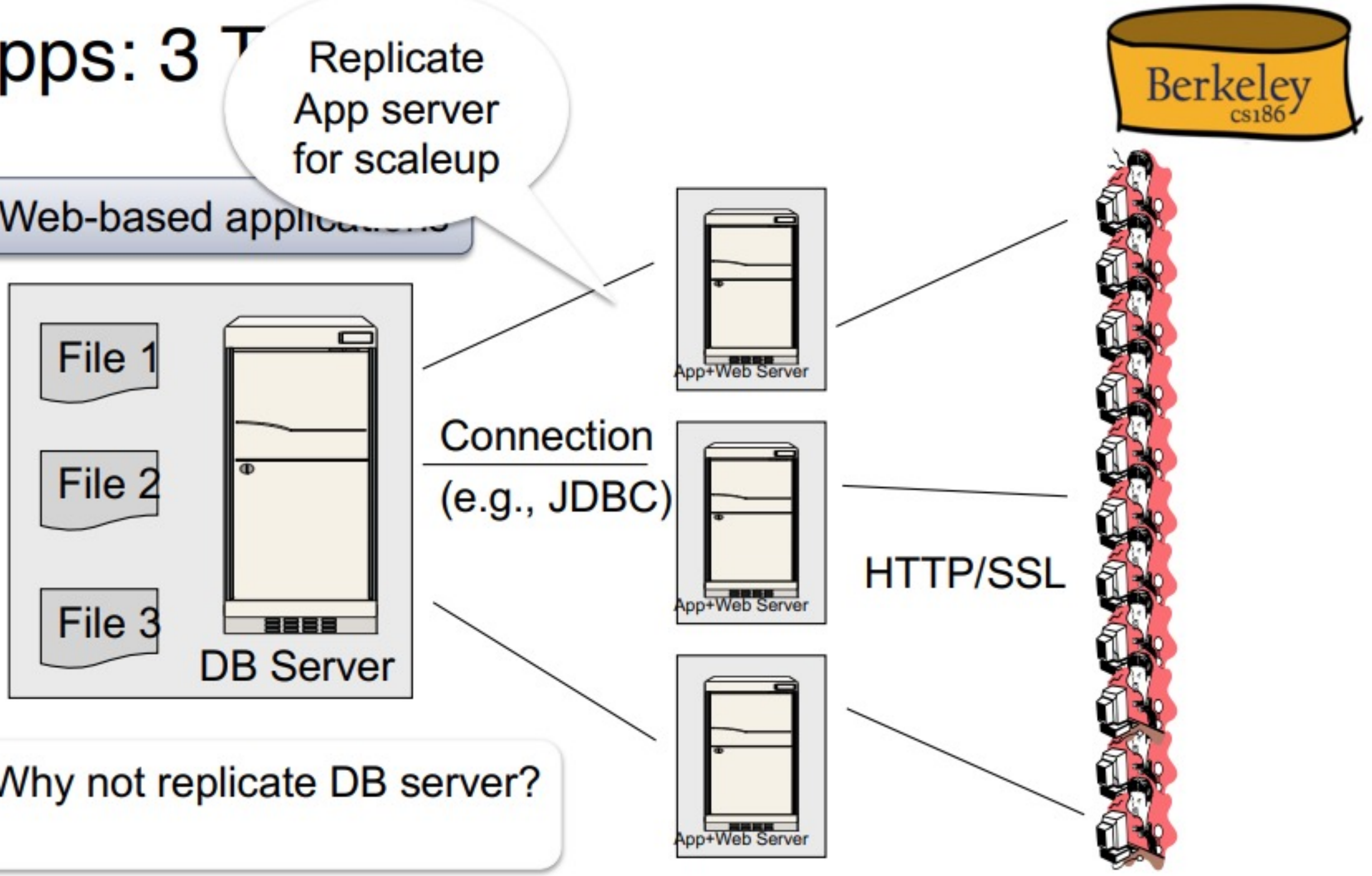
Web Apps: 3 Tier



Web Apps: 3 T

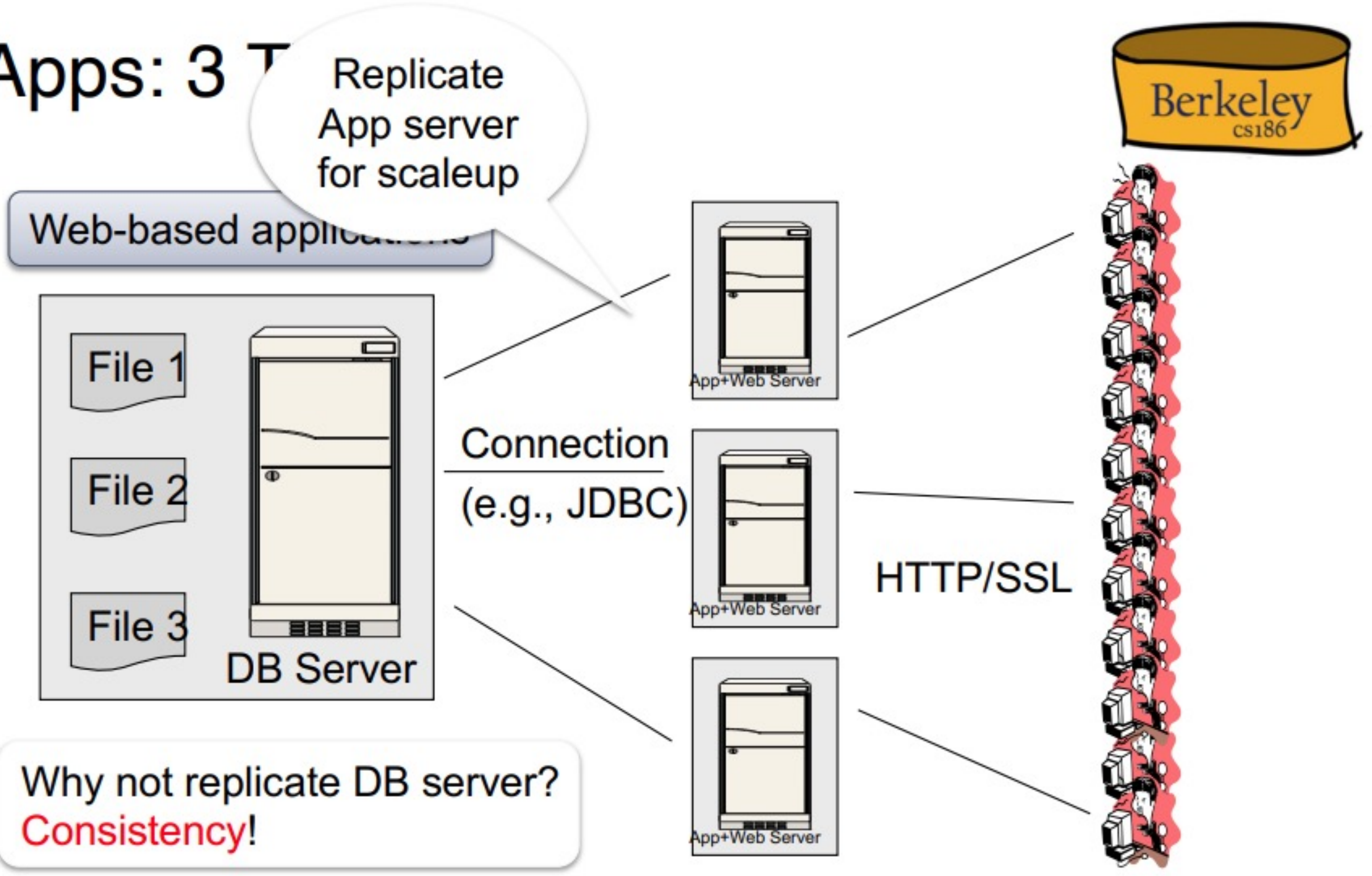
Replicate App server for scaleup

Web-based applic



Why not replicate DB server?

Web Apps: 3 T



Why not replicate DB server?
Consistency!

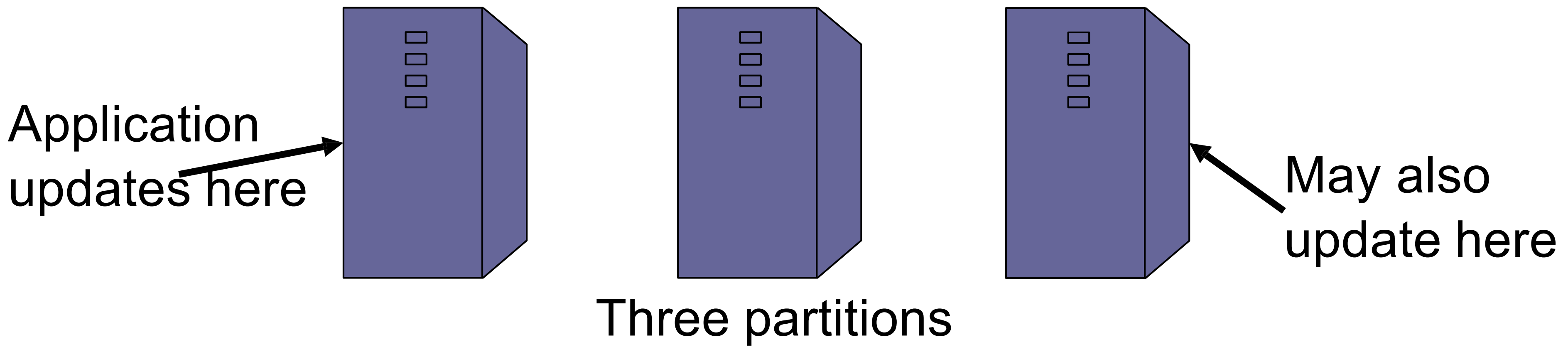
Replicating the Database

- Two basic approaches:
  - Scale up through partitioning – “sharding”
  - Scale up through replication

- Consistency is much harder to enforce

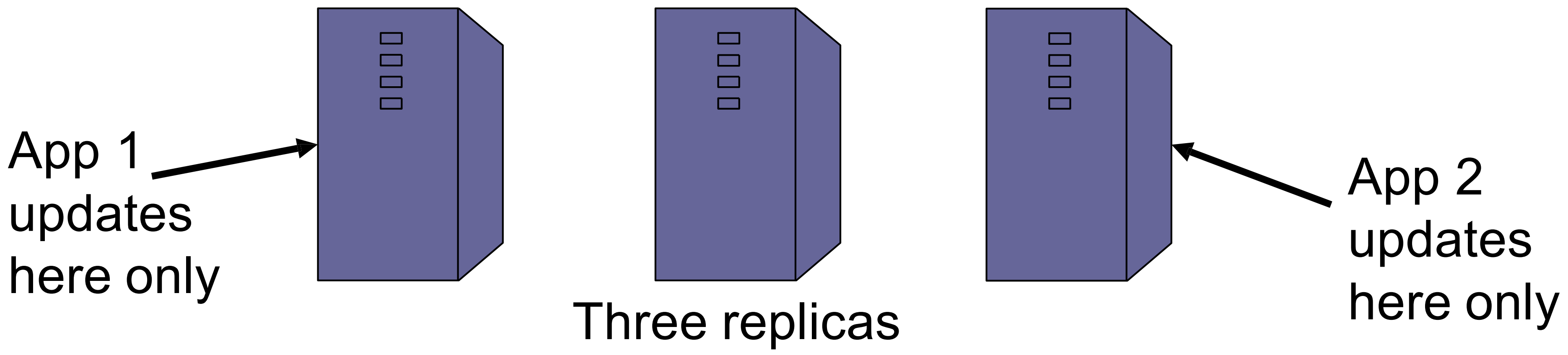
Scale Through Partitioning

- Partition the database across many machines in a cluster
  - Database now fits in main memory
  - Queries spread across these machines
- Can increase throughput
- Easy for writes but reads become expensive!



Scale Through Replication

- Create multiple copies of each database partition
- Spread queries across these replicas
- Can increase throughput and lower latency
- Can also improve fault-tolerance
- Easy for reads but writes become expensive!



Relational Model   NoSQL

- Relational DB: difficult to replicate/partition. E.g., `Supplier(sno,…),Part(pno,…),Supply(sno,pno)`
  - Partition: we may be forced to join across servers
  - Replication: local copy has inconsistent versions
  - Consistency is hard in both cases

- NoSQL: simplified data model
  - Given up on functionality
  - Application must now handle joins and consistency

MongoDB Data Model

MongoDB	DBMS
Database	Database
Collection	Relation
Document	Row/Record
Field	Column

Document = {…, field: value, …}

Where value can be:

- Atomic
- A document
- An array of atomic values
- An array of documents

{ qty : 1, status : "D", size : {h : 14, w : 21}, tags : ["a", "b"] },

[Same as the JSON data model]

Internally stored as BSON = Binary JSON

- Client libraries can directly operate on this natively

MongoDB: History 1

- A prototypical NoSQL database
- Short for hu**mongo**us
- First version in 2009!
- Still very popular
  - IPO in 2017
  - Now worth >7B in market capital (as of 2020)

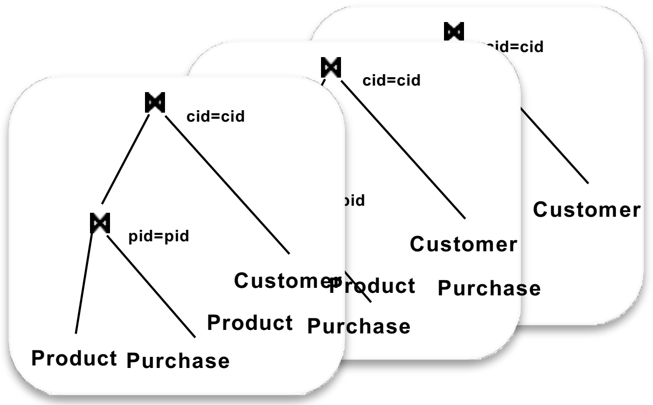
- We have discussed:
  - Single-node relational database systems
  - Parallel relational database systems
  - NoSQL databases

- What about parallel NoSQL databases?
  - That’s what we will discuss next!

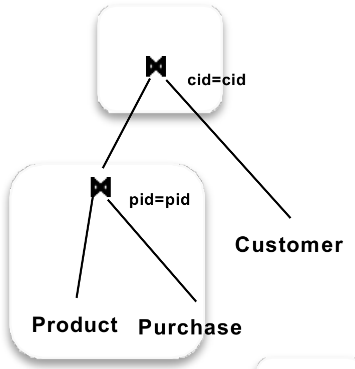
PARALLEL DATA PROCESSING IN THE 20TH CENTURY

Approaches to Parallel Relational Query Evaluation

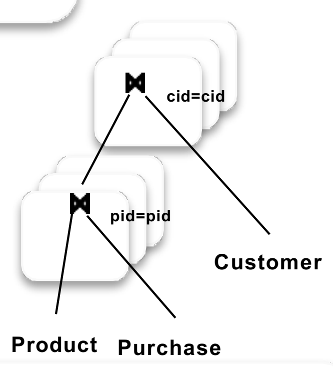
- Inter-query parallelism
  - One query per node
  - Good for transactional (OLTP) workloads



- Inter-operator parallelism
  - Operator per node
  - Good for analytical (OLAP) workloads



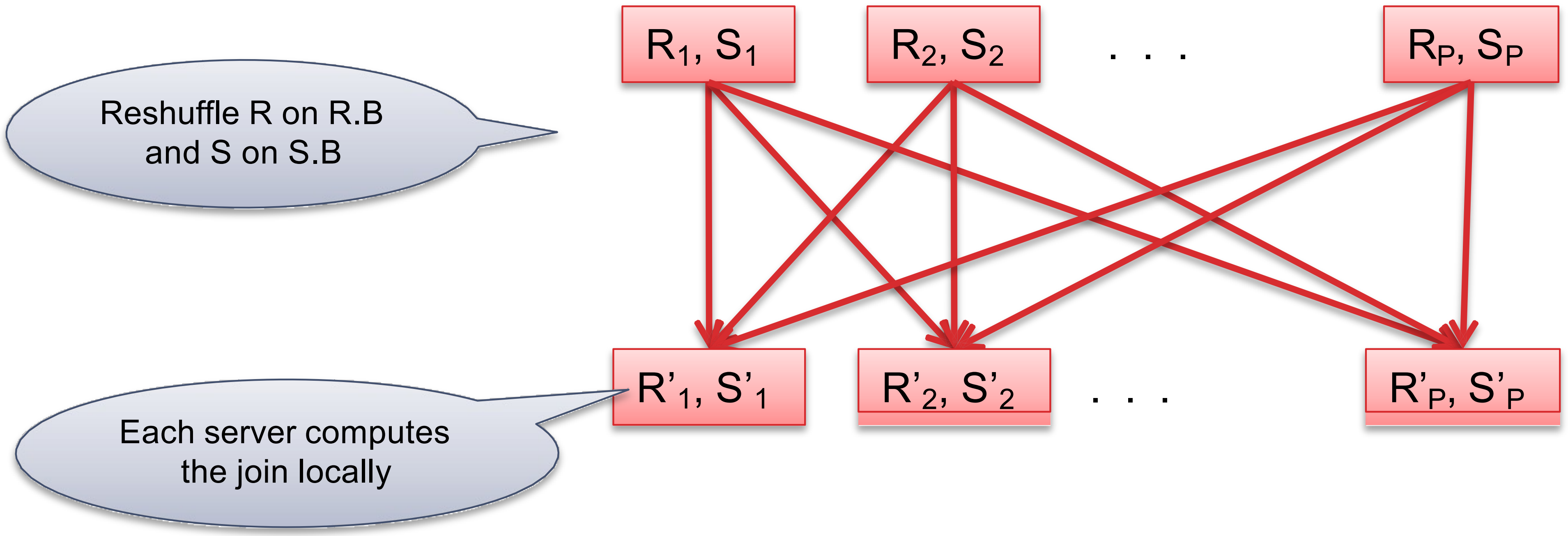
- Intra-operator parallelism
  - Operator on multiple nodes
  - Good for both?



We study only intra-operator parallelism: most scalable

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



Parallel Join Illustration

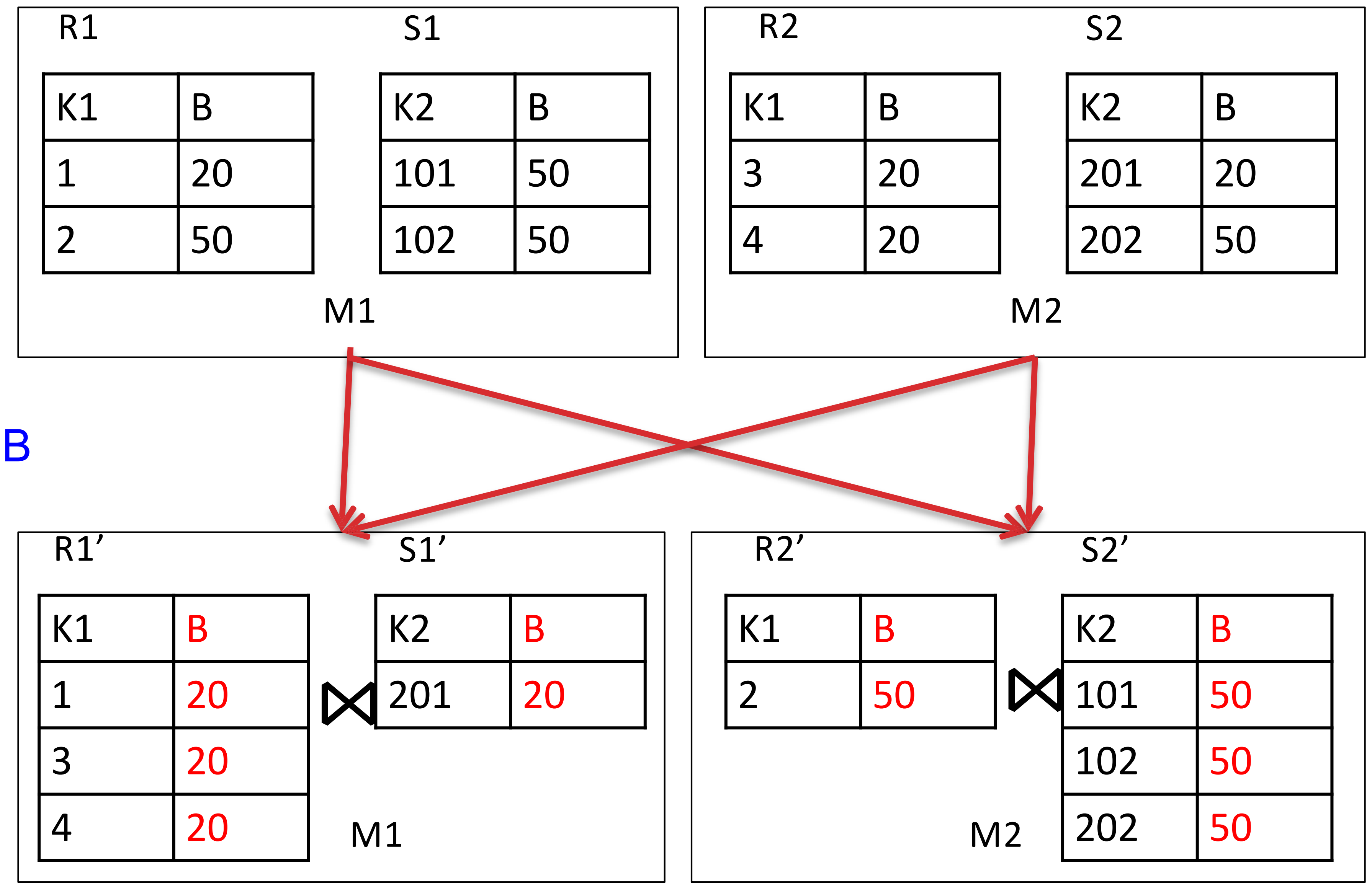
Data: R(K1,A, B), S(K2, B, C)

Query: R(K1,A,B) ⋈ S(K2,B,C)

Partition

Shuffle on B

Local Join



M1

R1

K1	B
1	20
2	50

S1

K2	B
101	50
102	50

M2

R2

K1	B
3	20
4	20

S2

K2	B
201	20
202	50

M1

R1’ ⋈ S1’

K1	B
1	20
3	20
4	20

K2	B
201	20

M2

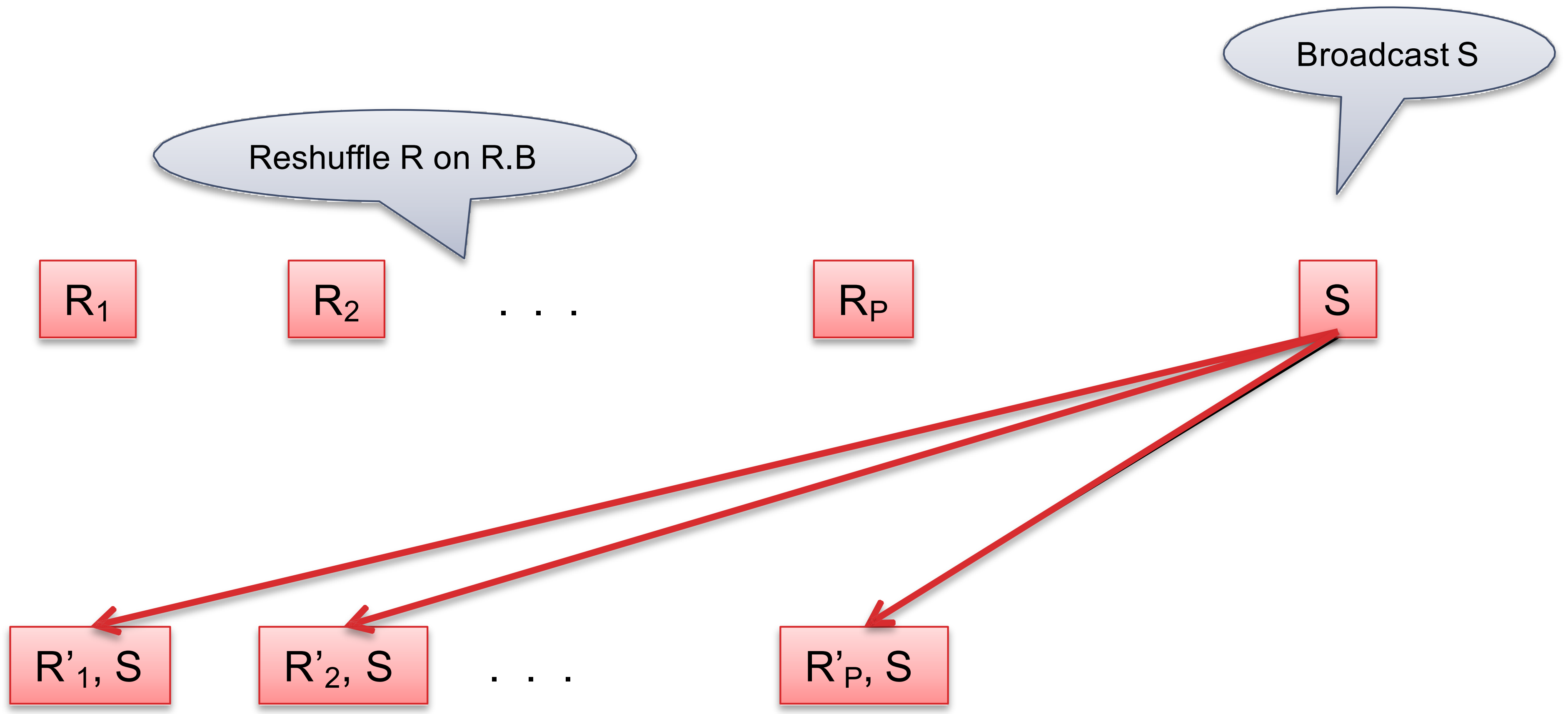
R2’ ⋈ S2’

K1	B
2	50

K2	B
101	50
102	50
202	50

Broadcast Join

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



Why would you want to do this?

Parallel Data Processing @ 2000

Optional Reading

- Original paper: https://www.usenix.org/legacy/events/osdi04/tech/dean.html
- Rebuttal to a comparison with parallel DBs: http://dl.acm.org/citation.cfm?doid=1629175.1629198
- Chapter 2 (Sections 1,2,3 only) of Mining of Massive Datasets, by Rajaraman and Ullman http://i.stanford.edu/~ullman/mmds.html

Motivation

- We learned how to parallelize relational database systems
- While useful, it might incur too much overhead if our query plans consist of simple operations
- 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

MapReduce

- Google: paper published 2004
- Free variant: Hadoop

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

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

slide source: Jeff Dean

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
  - 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

User provides the REDUCE function:

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

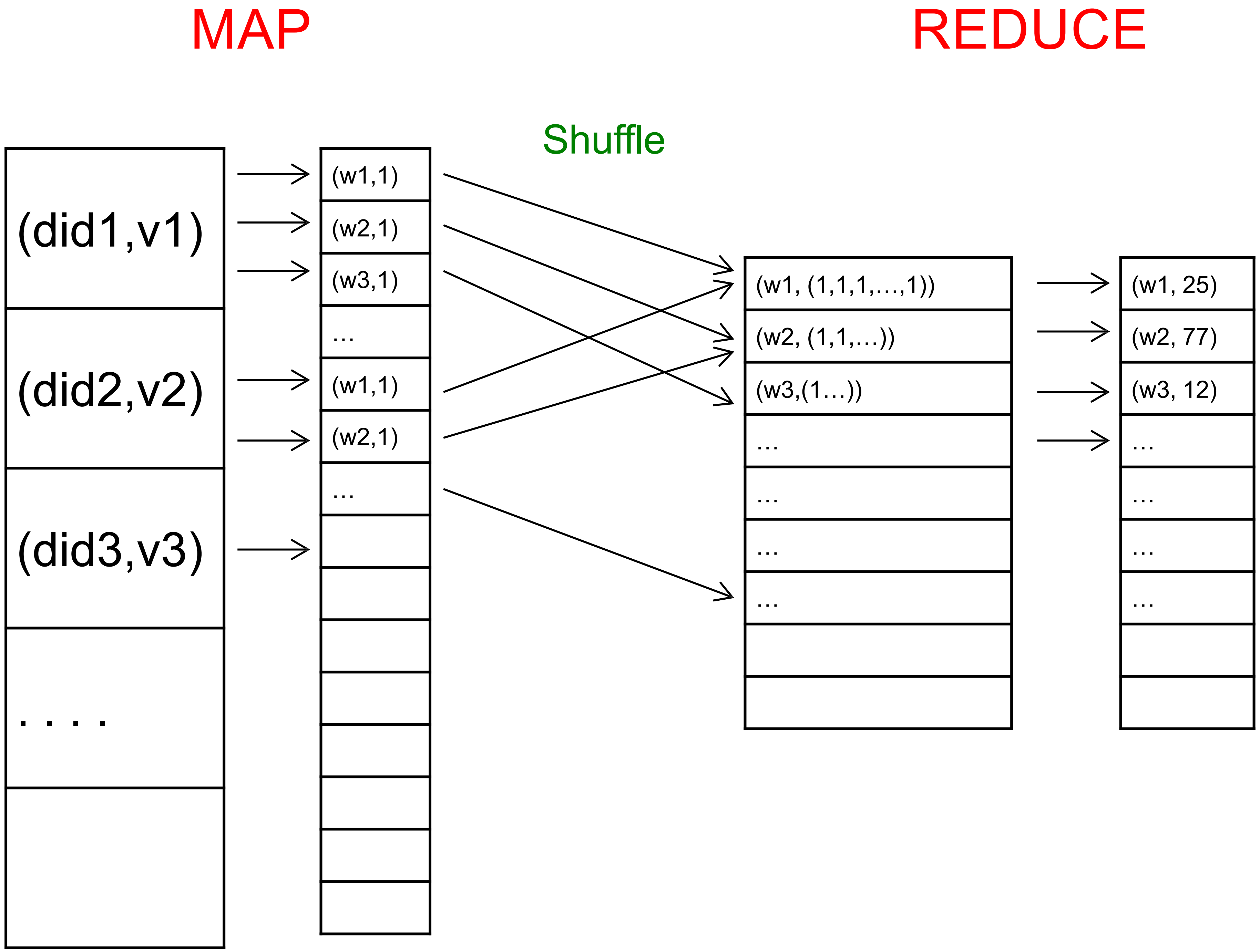
System groups all pairs with the same intermediate key, and passes the bag of values to the REDUCE function

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)

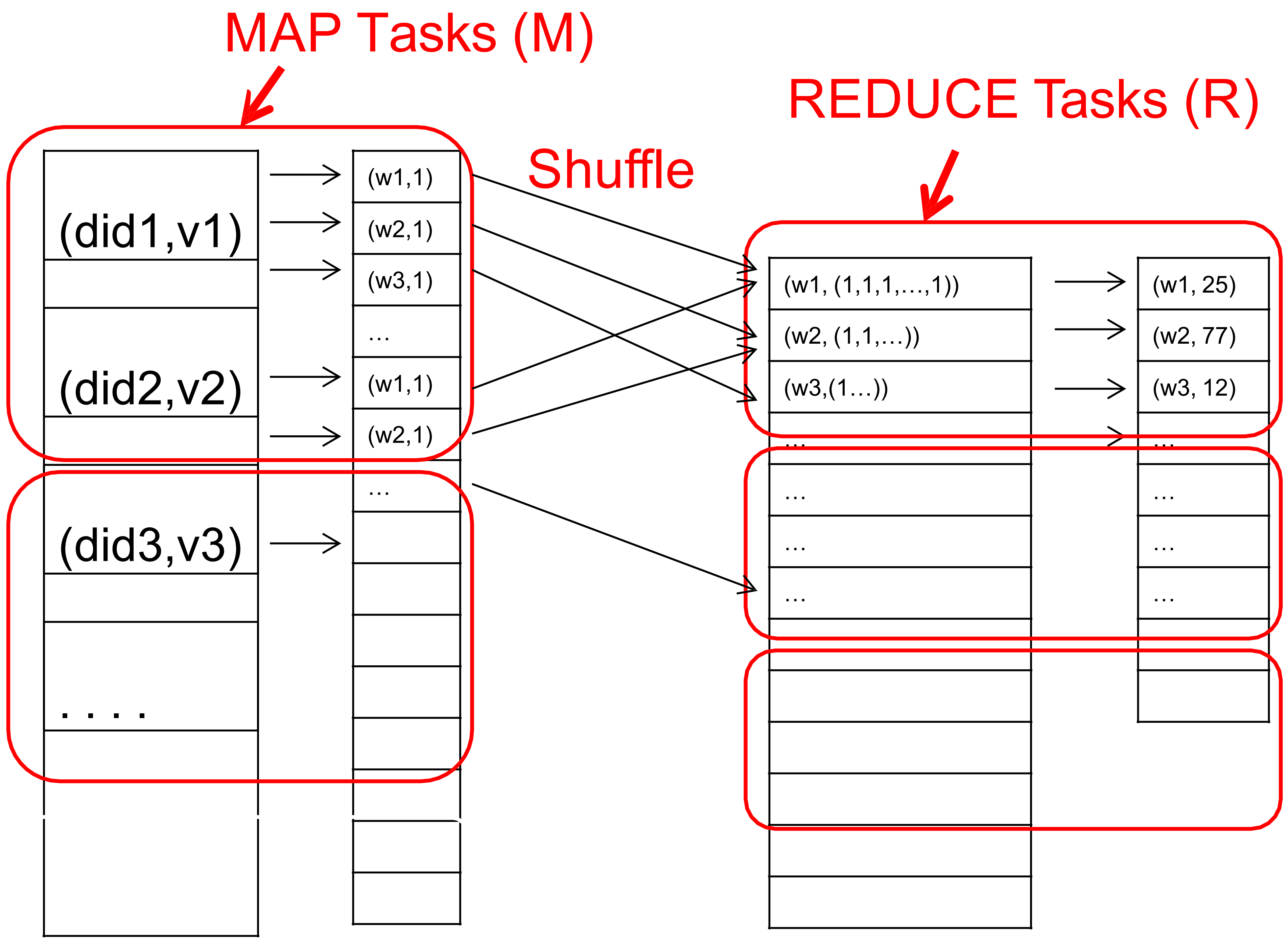
```
map(String key, String value):
   // key: document name
   // value: document contents
   for each word w in value:
        emitIntermediate(w, “1”);
```

```
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
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 4 or 8 per node


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)
…

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

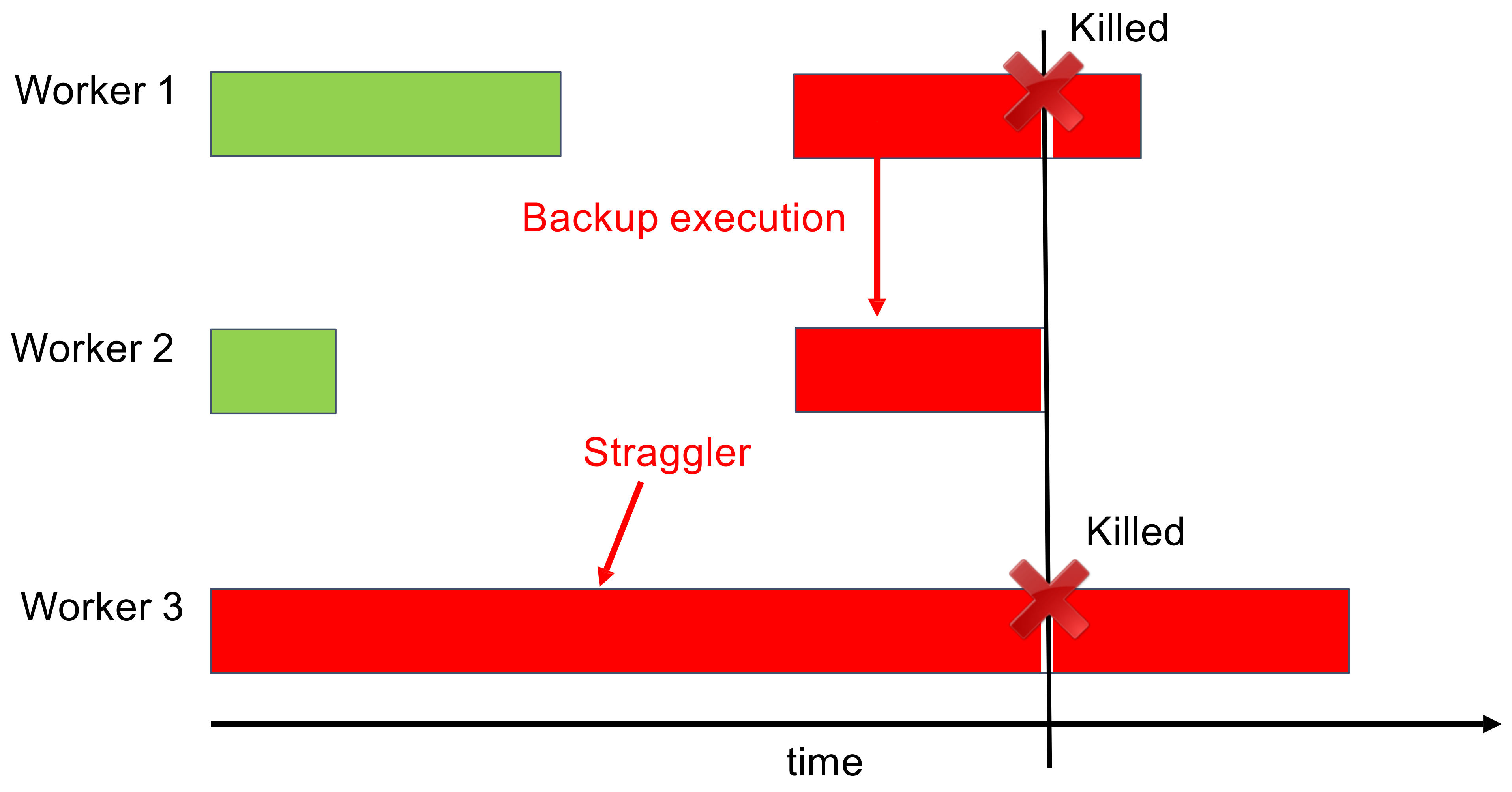
- There is one master node
- Master partitions input file into *M splits*, by key
- Master assigns *workers* (=servers) to the *M map tasks*, keeps track of their progress
- Workers write their output to local disk, partition into *R regions*
- Master assigns workers to the *R reduce tasks*
- Reduce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

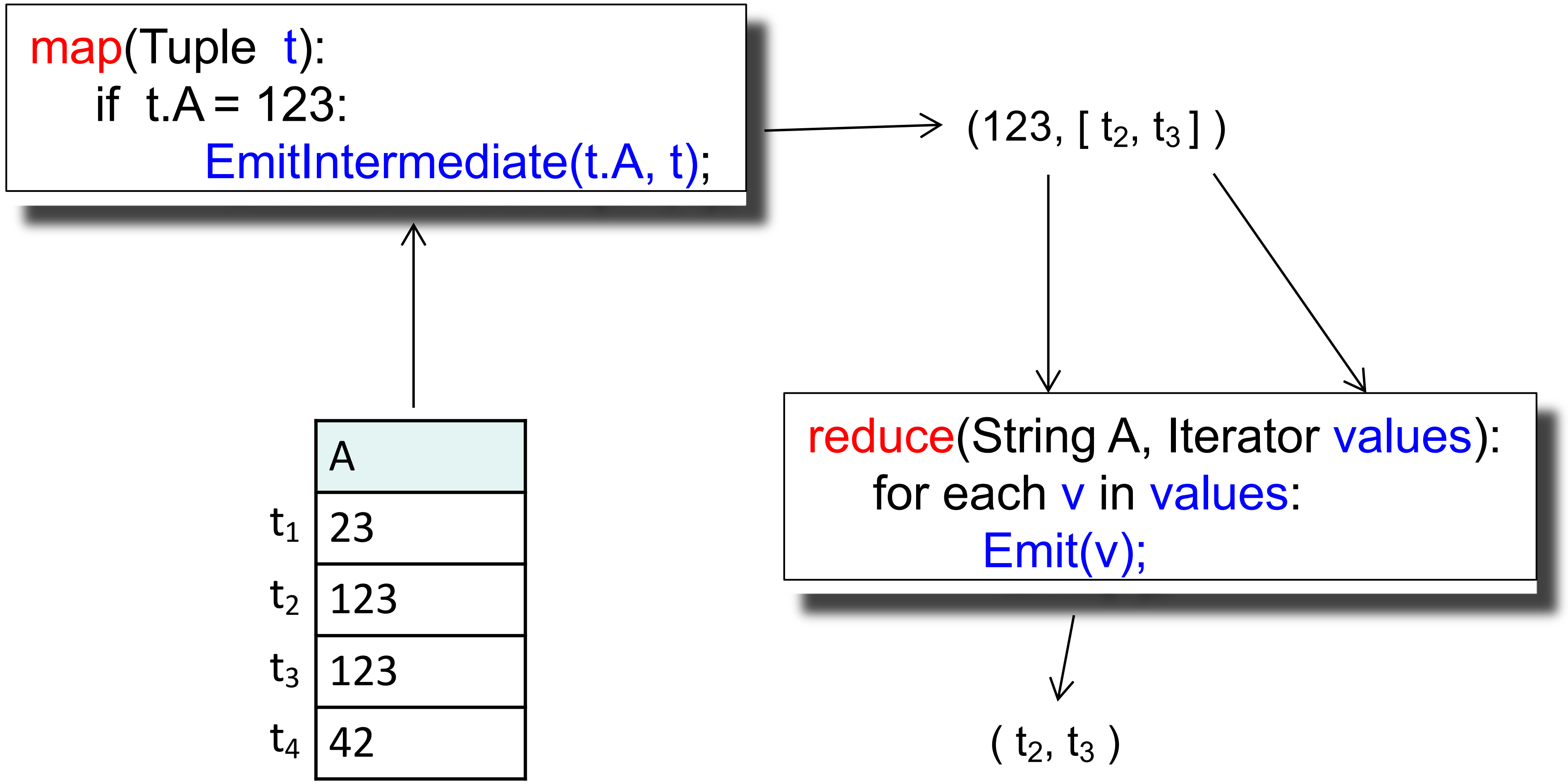
IMPLEMENTING RA OPERATORS IN MR

Relational Operators in MapReduce

Given relations R(A,B) and S(B,C) compute:

- Selection: $\sigma_{A=123}(R)$
- Group-by: $\gamma_{A,sum(B)}(R)$
- Join: $R \bowtie S$

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



```
map(Tuple t):
    if t.A = 123:
        EmitIntermediate(t.A, t);
```

	A
t_1	23
t_2	123
t_3	123
t_4	42

(123, [t_2, t_3])

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

(t_2, t_3)

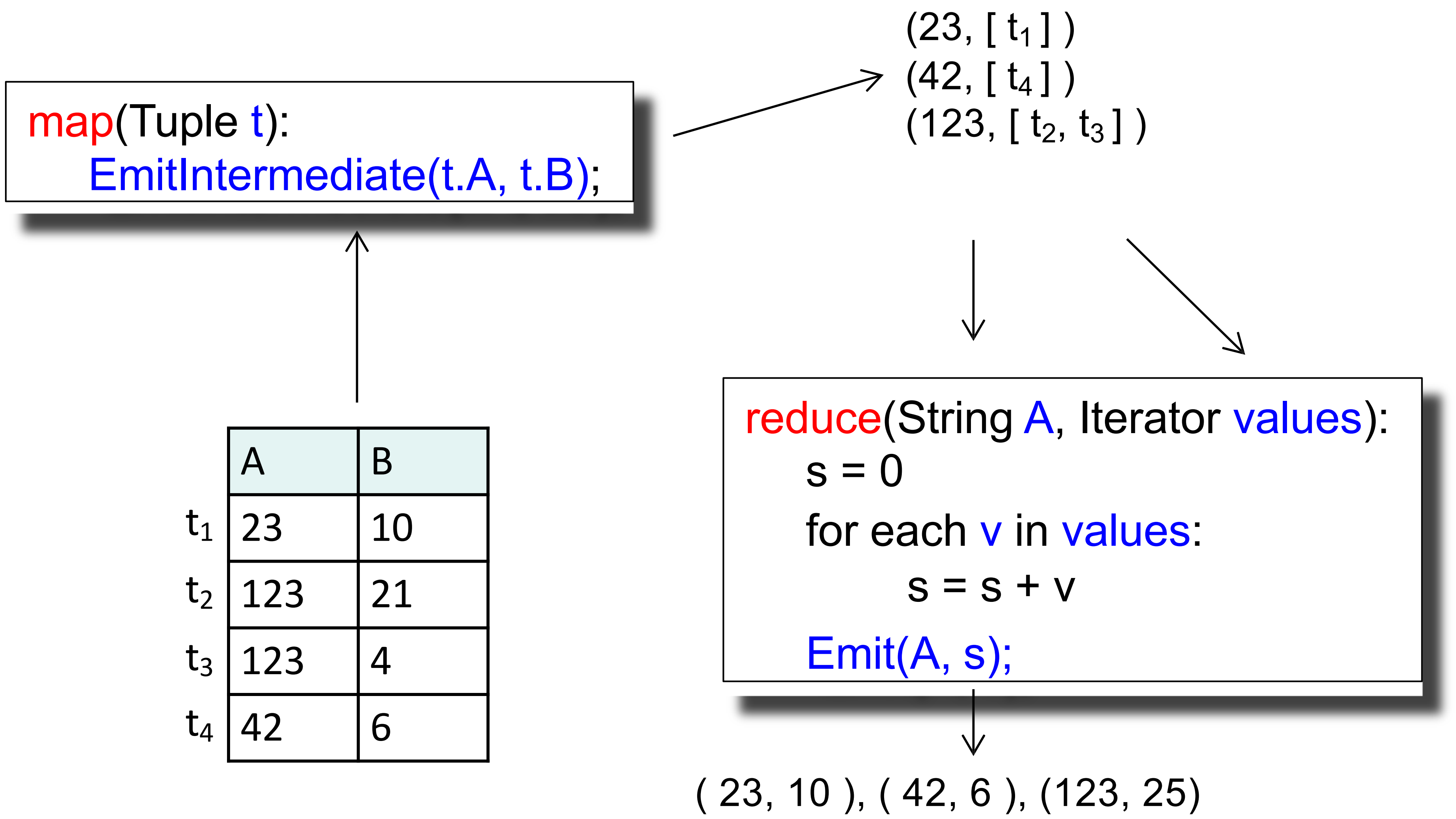
Selection $\sigma_{A=123}(R)$

```
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

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

(23, [t_1])
(42, [t_4])
(123, [t_2, t_3])

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

(23, 10), (42, 6), (123, 25)

Join

Let’s review our parallel join algorithms:

- Partitioned hash-join
- Broadcast join

Partitioned Hash-Join

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

type

actual tuple

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

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

```
reduce(String k, Iterator values):
    R = [ ];  S = [ ];
    for each v in values:
        case v.type of:
            'R':   R.append(v)
            '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)

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

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

Parallel Data Processing @ 2010

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 in 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

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

Data Model: Resilient Distributed Datasets

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

What are the benefits of lazy execution?

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>`

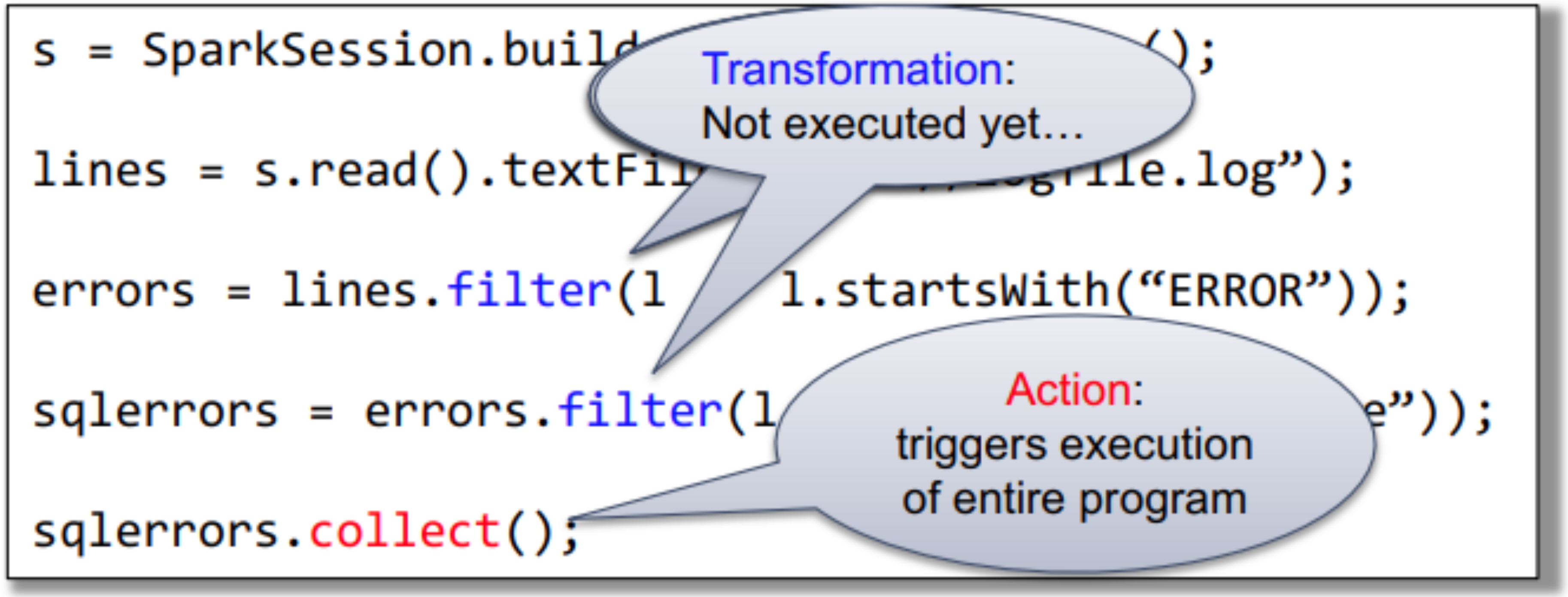
```
s = SparkSession.builder()...getOrCreate();

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.buil                       ();

lines = s.read().textFi              ile.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

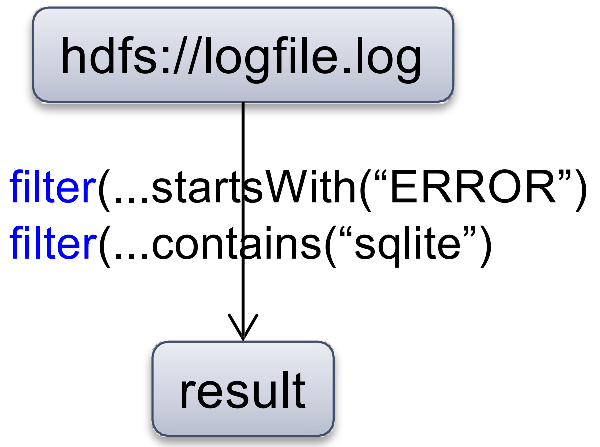
Persistence

```
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:

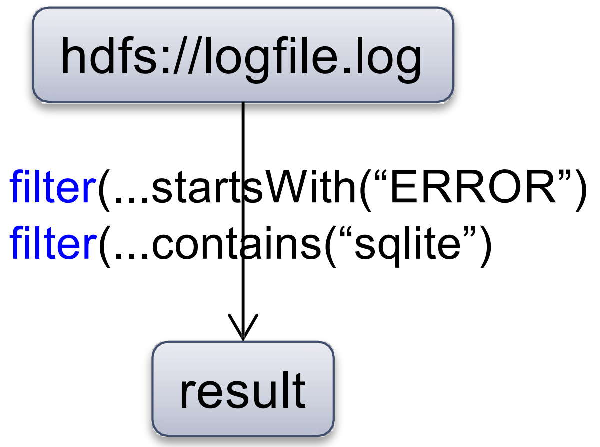


```
lines = s.read().textFile(“hdfs://logfile.log”);
errors = lines.filter(l->l.startsWith(“ERROR”));
sqlerrors = errors.filter(l-
>l.contains(“sqlite”)); sqlerrors.collect();
```

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

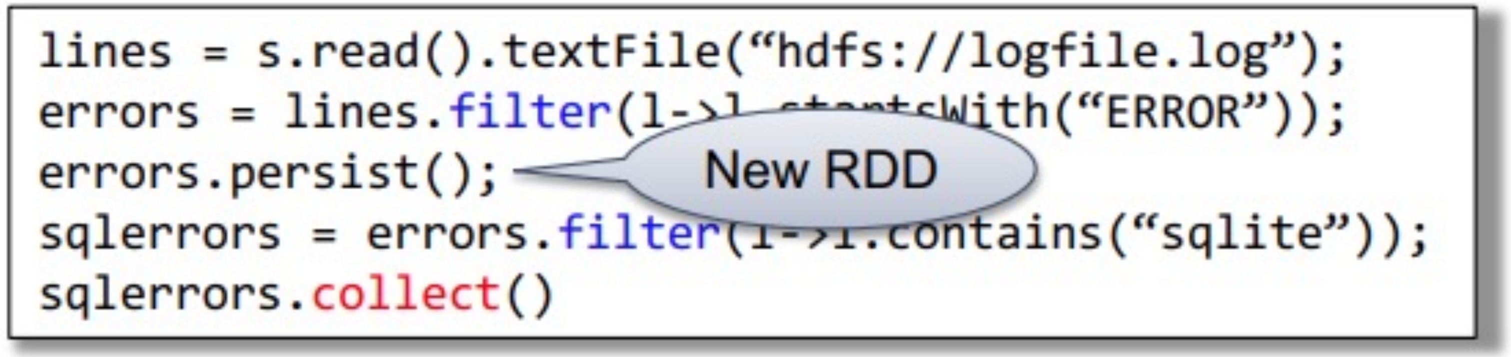
Persistence

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



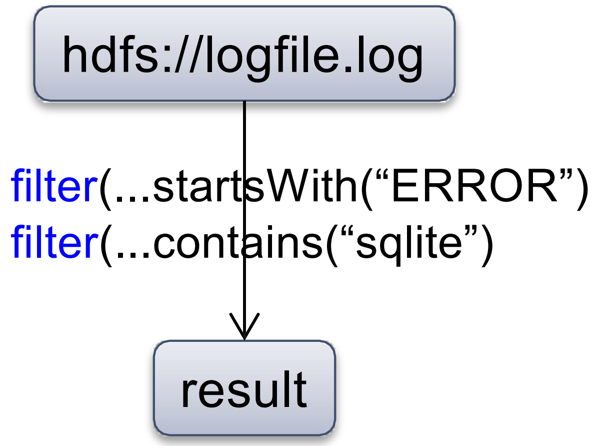
```
lines = s.read().textFile("hdfs://logfile.log");
errors = lines.filter(l->l.startsWith("ERROR"));
errors.persist();
sqlerrors = errors.filter(l->l.contains("sqlite"));
sqlerrors.collect()
```

Spark can recompute the result from errors

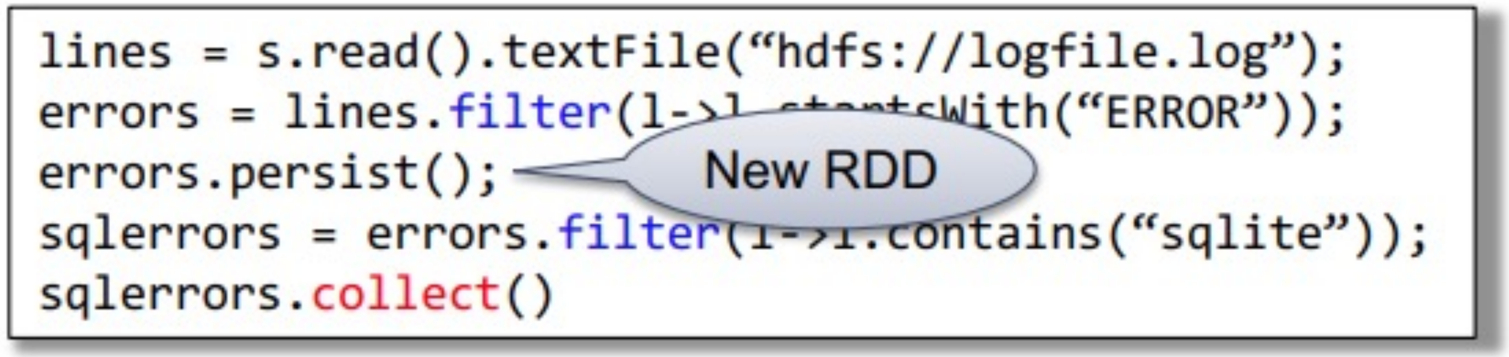
Persistence

RDD:

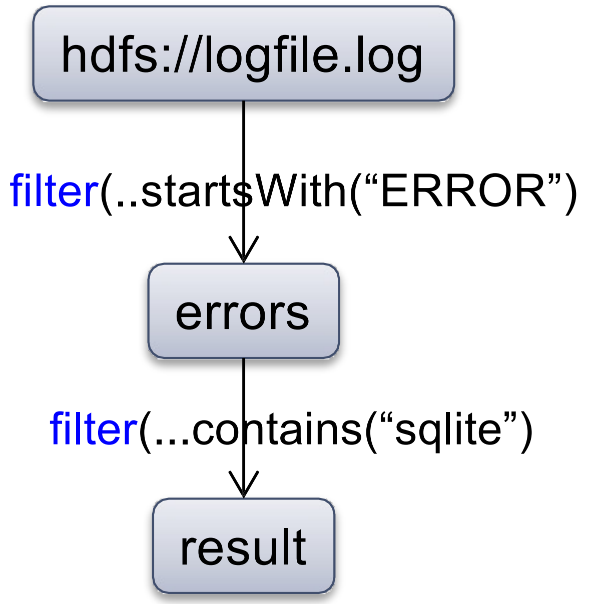
```
lines = s.read().textFile(“hdfs://logfile.log”);
errors = lines.filter(l->l.startsWith(“ERROR”));
sqlerrors = errors.filter(l-
>l.contains(“sqlite”)); sqlerrors.collect();
```



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()
```



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

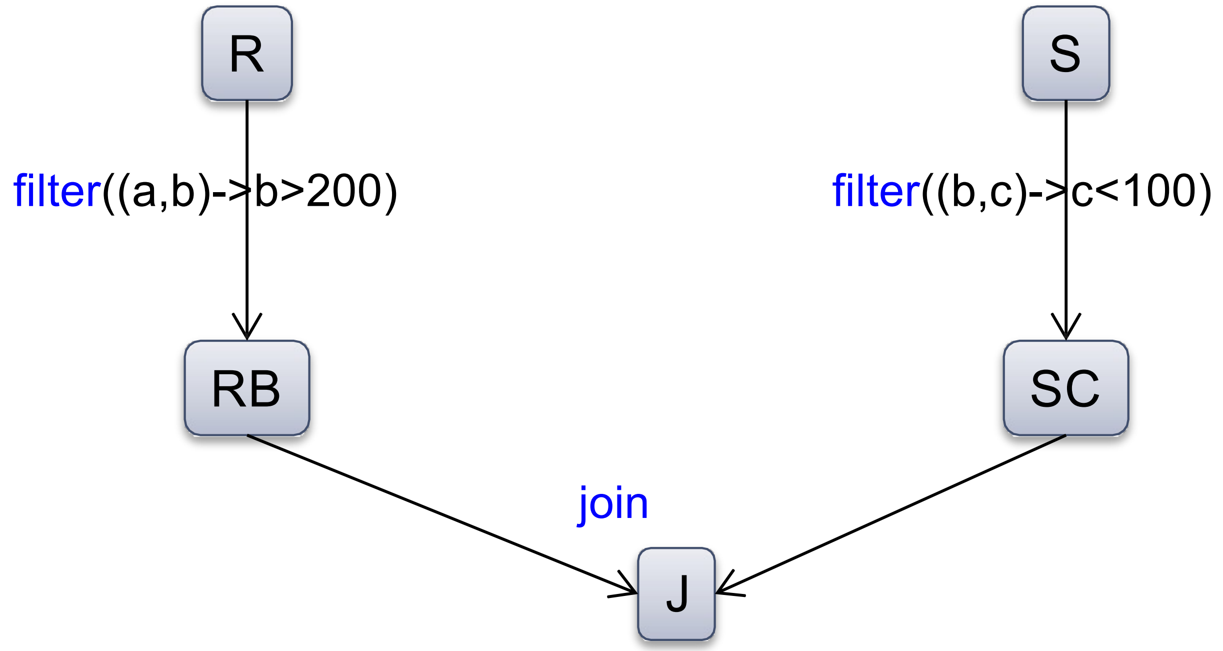
R(A,B)
S(A,C)

```
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/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

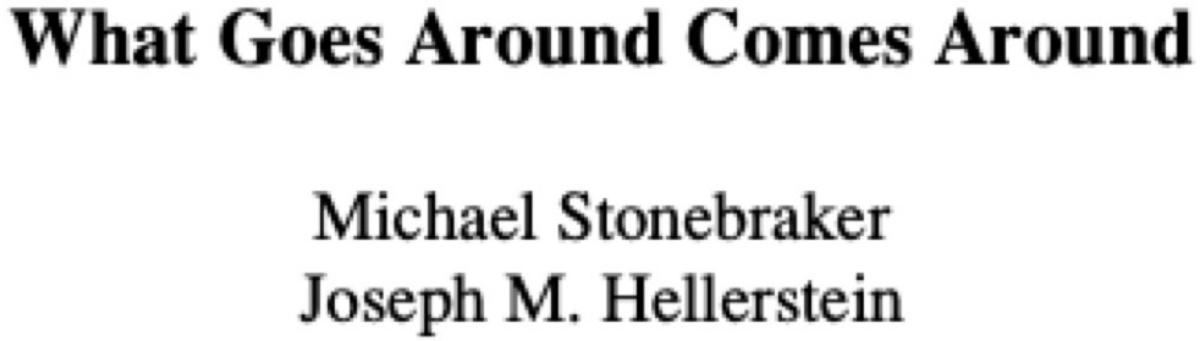
- Organized into *named columns* rather than individual objects
  - Just like a relation
  - Elements are untyped objects called `Row`’s

- Similar API as RDDs with additional methods

  -
    ```
    people = spark.read().textFile(…);
    ageCol = people.col(“age”);
    ageCol.plus(10); // creates a new DataFrame
    ```

Datasets

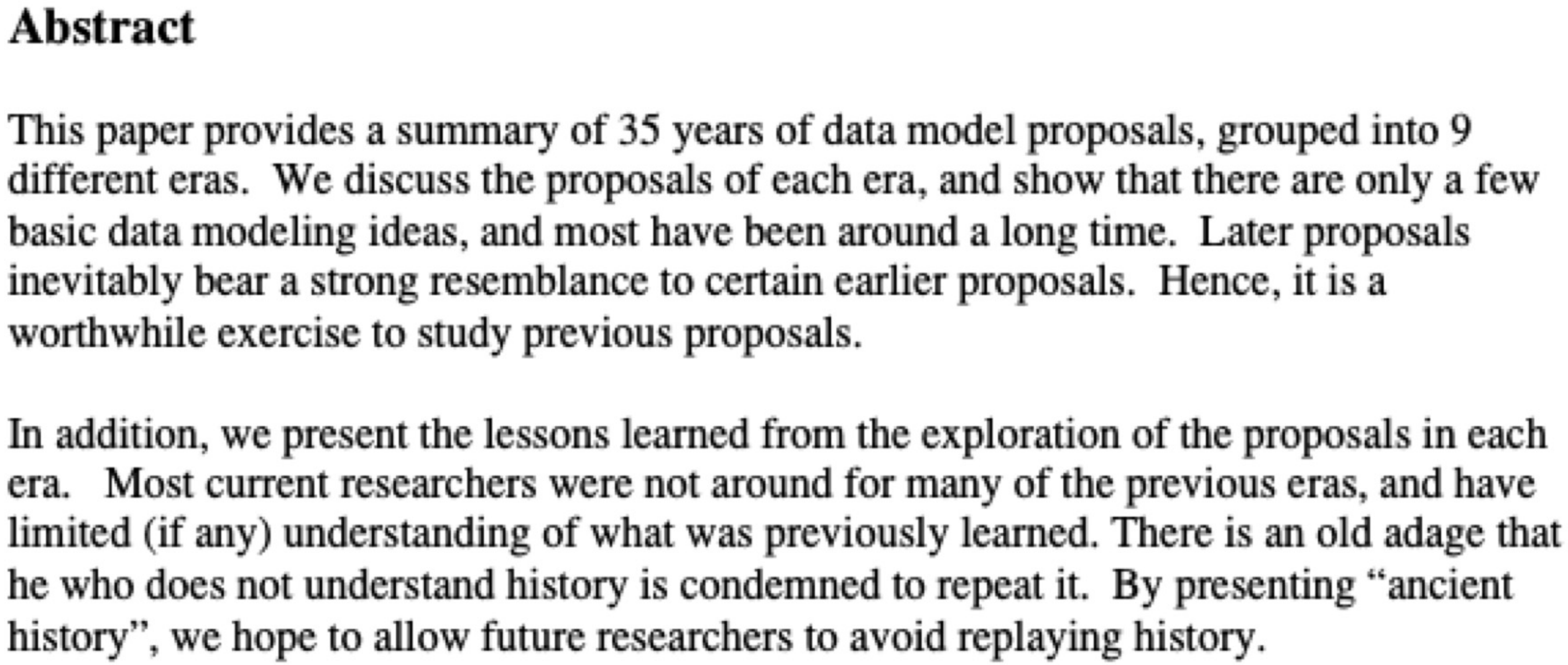
- Similar to DataFrames, except that elements must be typed objects
- E.g.: `Dataset<People>` rather than `Dataset<Row>`
- Can detect errors during compilation time
- DataFrames are aliased as `Dataset<Row>` (as of Spark 2.0)



What Goes Around Comes Around

Michael Stonebraker
Joseph M. Hellerstein

Readings in Database Systems, 5th Edition



Abstract

This paper provides a summary of 35 years of data model proposals, grouped into 9 different eras. We discuss the proposals of each era, and show that there are only a few basic data modeling ideas, and most have been around a long time. Later proposals inevitably bear a strong resemblance to certain earlier proposals. Hence, it is a worthwhile exercise to study previous proposals.

In addition, we present the lessons learned from the exploration of the proposals in each era. Most current researchers were not around for many of the previous eras, and have limited (if any) understanding of what was previously learned. There is an old adage that he who does not understand history is condemned to repeat it. By presenting "ancient history", we hope to allow future researchers to avoid replaying history.

Conclusions

- Parallel databases
  - Predefined relational operators
  - Optimization
  - Transactions and recovery
- 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