IMEN891K / IDSC501 / AXGS-501 · DATA ENGINEERING · FALL 2026

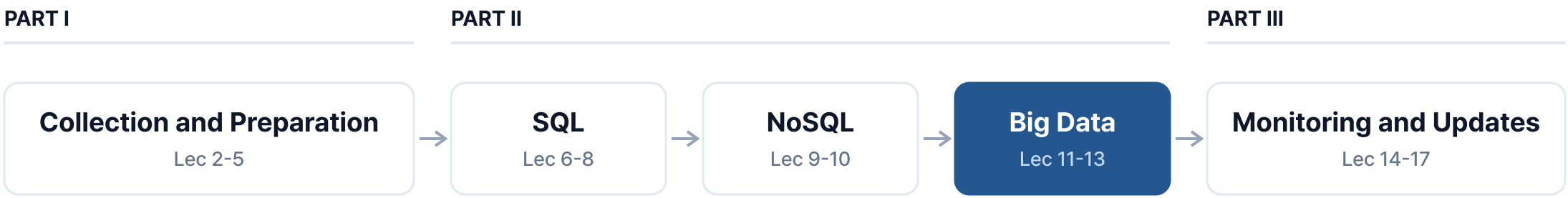
11.

Big Data I: Batch Processing

Sangwoo Mo · POSTECH

Industrial & Management Engineering

Where We Are


PART I
PART II
PART III
Collection and Preparation
Lec 2-5
SQL
Lec 6-8
NoSQL
Lec 9-10
Big Data
Lec 11-13
Monitoring and Updates
Lec 14-17

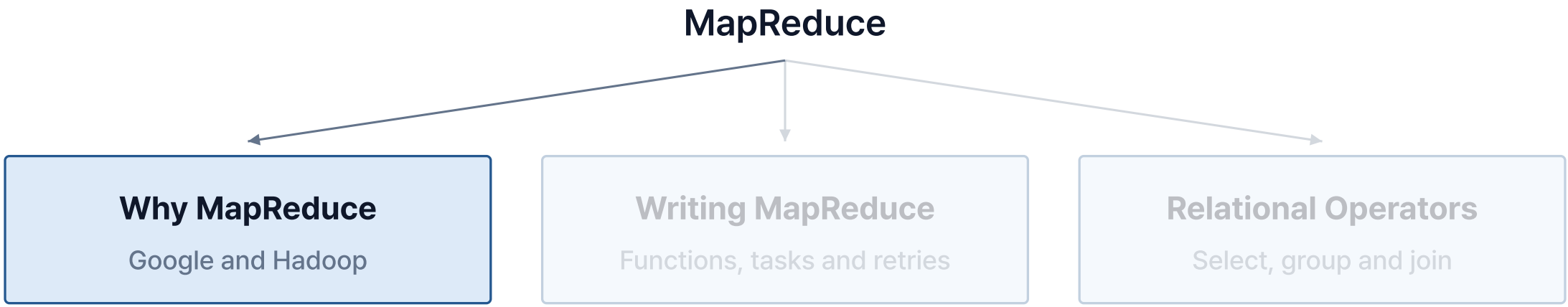
Outline: Big Data I

01	**MapReduce**	write map and reduce functions for distributed jobs
02	**Spark**	compose RDD operations, then describe table queries with DataFrames

01 · MapReduce

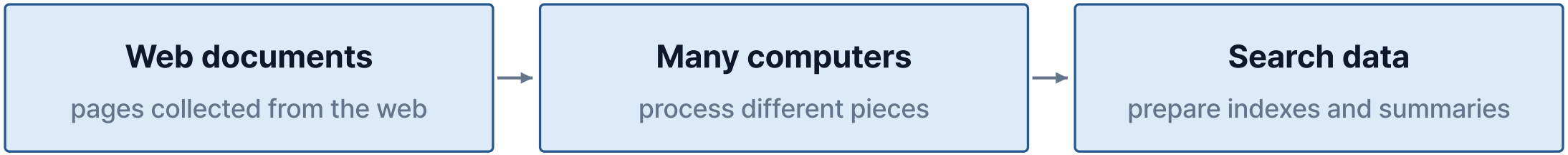
01	**MapReduce**	write map and reduce functions for distributed jobs
02	**Spark**	compose RDD operations, then describe table queries with DataFrames

§ Why MapReduce



Google Had a Data Problem

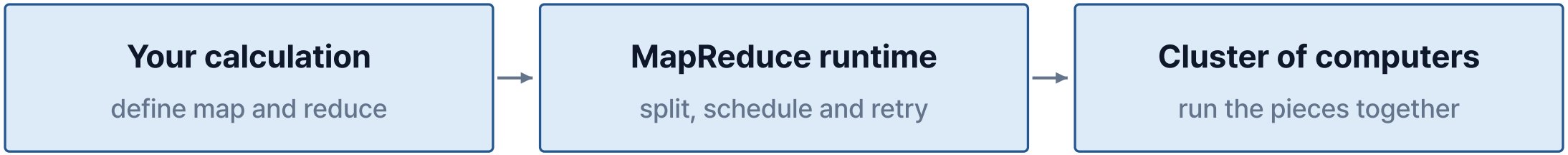
Google needed to turn large collections of web pages and request logs into useful data



Simple calculations became difficult when the input needed many computers

Google Publishes MapReduce

In 2004, Jeffrey Dean and Sanjay Ghemawat at Google published the MapReduce paper



Programmers describe the calculation; the runtime coordinates the computers

Hadoop Makes the Idea Available

Doug Cutting and Mike Cafarella developed Hadoop from the Nutch search project



Hadoop brought distributed storage and MapReduce to an open-source community

The Name Hadoop

The elephant in the logo has a family story behind it



- Hadoop was the name of Doug Cutting's son's toy elephant
- The family nickname became the software project's name

Why Many Machines

Distributed processing divides work across computers when one computer is insufficient

One computer reads everything

All input rows

one computer does all the work

Three computers share the reading

Computer 1: one piece

Computer 2: one piece

Computer 3: one piece

Same question, same answer; different computers handle different rows

Different computers read different rows to answer the same question

Scale Up or Scale Out

Scale up adds resources to one machine; scale out adds machines

Scale up

One larger machine

more memory / processing power

Simpler, until one machine is not enough

Scale out

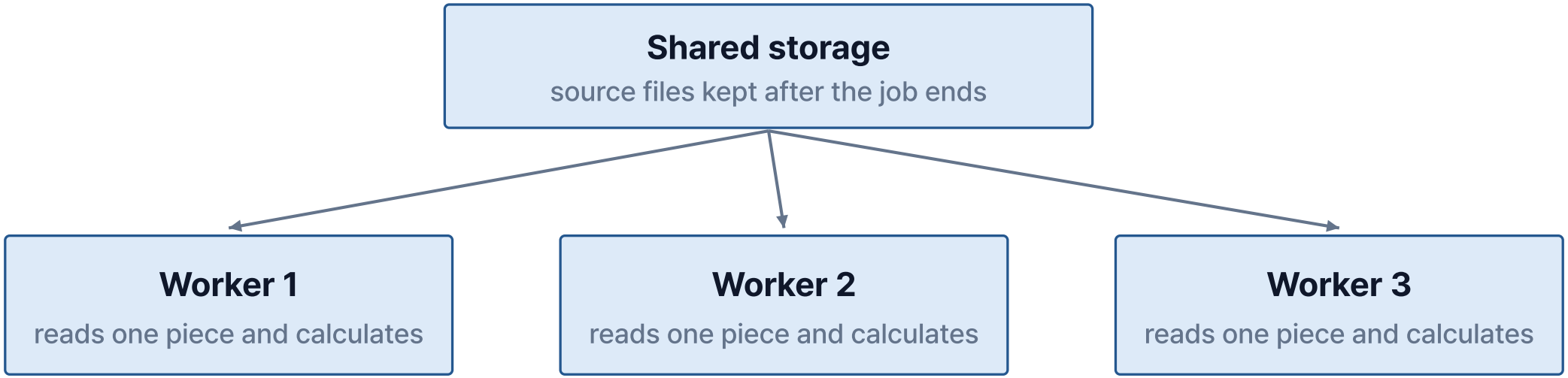
Worker 1 **Worker 2** **Worker 3**

More capacity, plus coordination and transfers

Extra machines add capacity and coordination work

Storage and Compute

Storage keeps the files; workers are processes that read files and calculate results

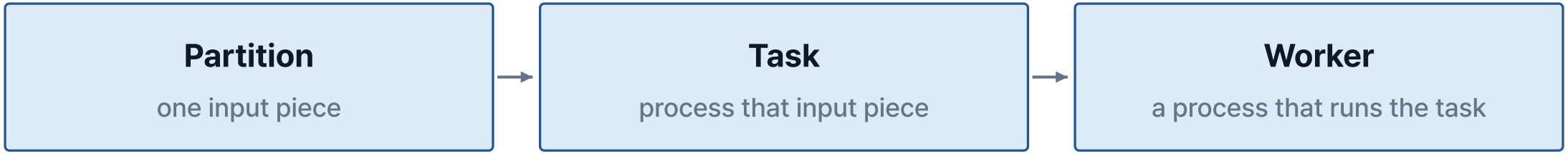


Storage keeps source data; workers run the computation

Durable input and temporary computation have different jobs

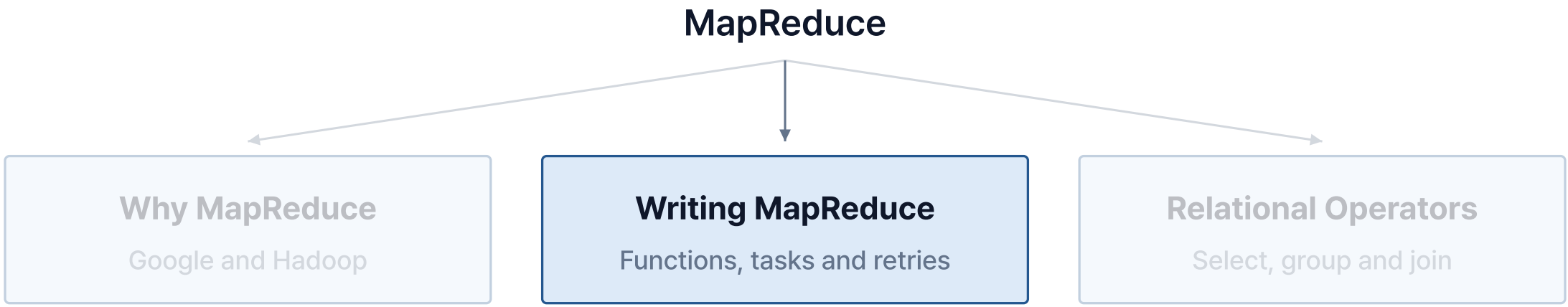
Distributed Files and Tasks

HDFS stores file blocks; MapReduce assigns computation to workers



- A distributed file system divides a large file into blocks
- Replicas keep another copy if a storage server fails
- Workers run tasks over pieces of the stored input

§ Writing MapReduce



Word Count

The input is a collection of documents; the output is one count per word

Document ID	Document contents
d1	spark makes data simple
d2	spark processes data

Question: How many times does each word occur across all documents?

Word Count Functions

The mapper emits one per word; the reducer sums values for that word

- 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));
```

The Python walkthrough makes the same two functions explicit

Map: Emit a Word and One

Each yielded pair says that this word occurred once in this document

```
def map_doc(document_id, text):
    for word in text.split():
        yield (word, 1)
```

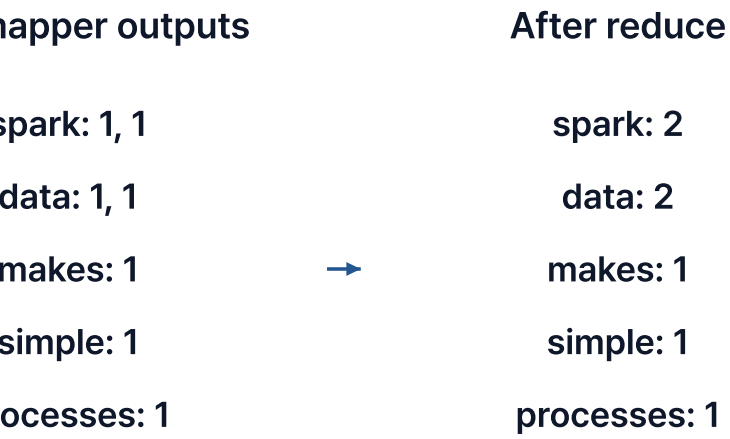


The output key is the word, so occurrences from different documents can meet

Shuffle and Reduce

The runtime groups equal keys before calling reduce_word once for each key

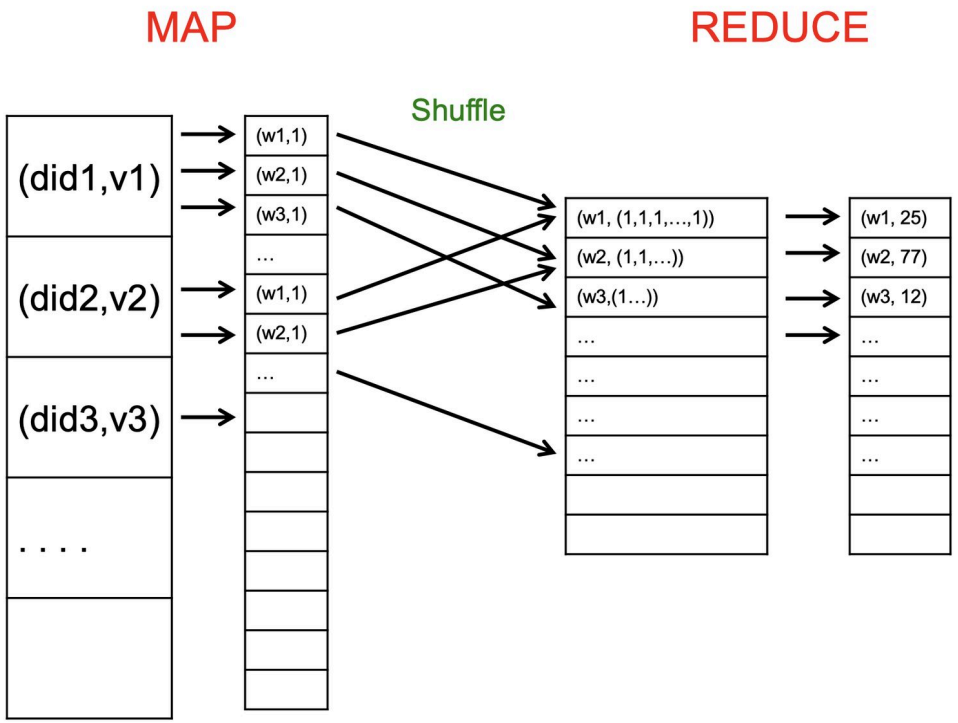
```
def reduce_word(word, counts):
    yield (word, sum(counts))
```



Two documents produce seven tokens and five distinct word counts

Word Count: The Full Data Flow

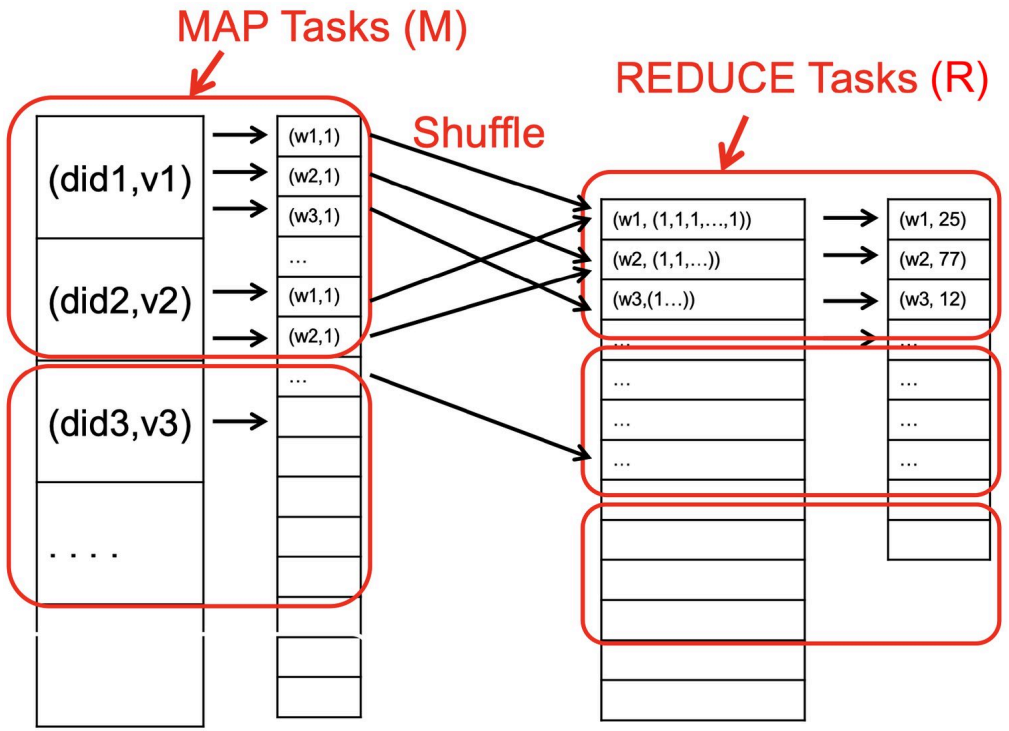
did identifies a document, v is its text, and w is a word



Map emits (word, 1); shuffle groups words; reduce adds each group

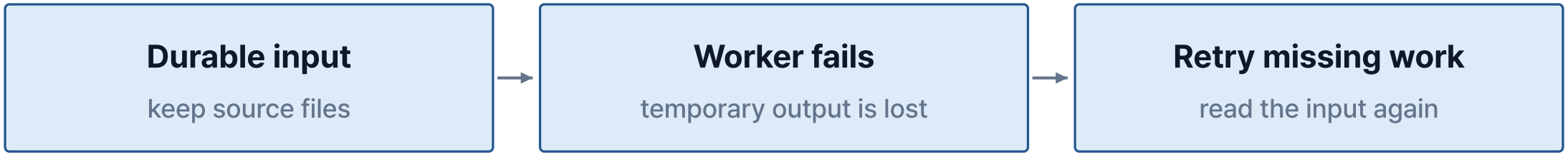
Map Tasks and Reduce Tasks

A task is one unit of work; a worker process can run multiple tasks over time



Map tasks split the input; reduce tasks split the grouped keys

A Worker Can Fail



Retry needs available source data and a repeatable computation

Demo: Follow the Grouped Count

Sensor example: count readings at or above 80°C by machine

Back | Step | Reset

Input partitions	Map: keep + emit	Shuffle: group keys	Reduce: add
M1 70 | M2 90 | M1 85			
M1 95 | M3 75 | M2 60			
M3 88 | M1 82 | M2 78			

Input only: predict which readings pass temp >= 80 and how many remain.

Track M1 from three input partitions to one result

Answer: Follow the Grouped Count

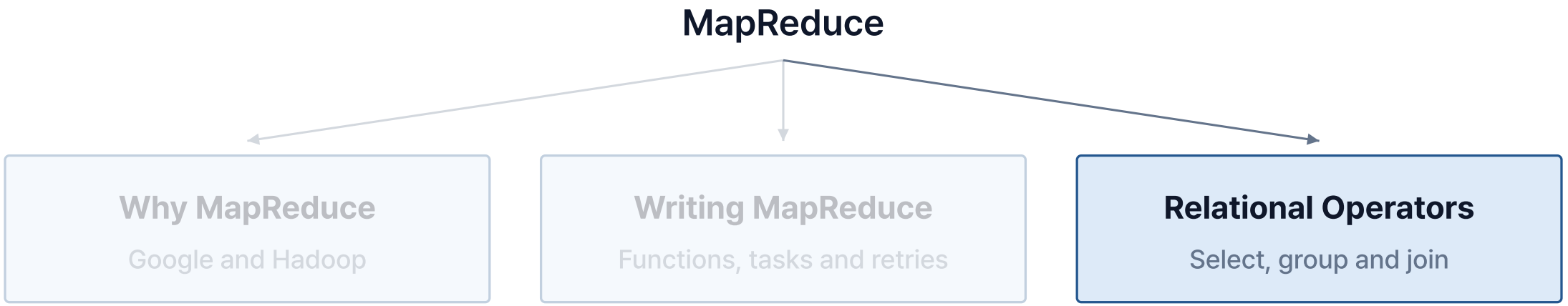
Map emits five pairs, shuffle forms three keys, reduce returns three counts



Selection: temp >= 80 | Key: machine | Value: 1

The output counts sum to five retained readings

§ Relational Operators



Relational Operations in MapReduce

SQL names the result; MapReduce requires you to write how records produce it

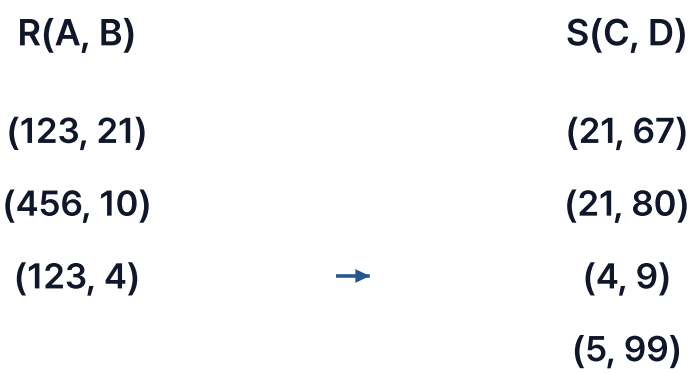
SQL operation	What the MapReduce program must decide
WHERE	which rows map emits
GROUP BY + SUM	grouping key and values to add
JOIN	join key, source tags and matching pairs

The same familiar SQL operation requires explicit record-processing code

The Records for Our SQL Examples

The query determines the intermediate key; a tuple position does not determine its meaning

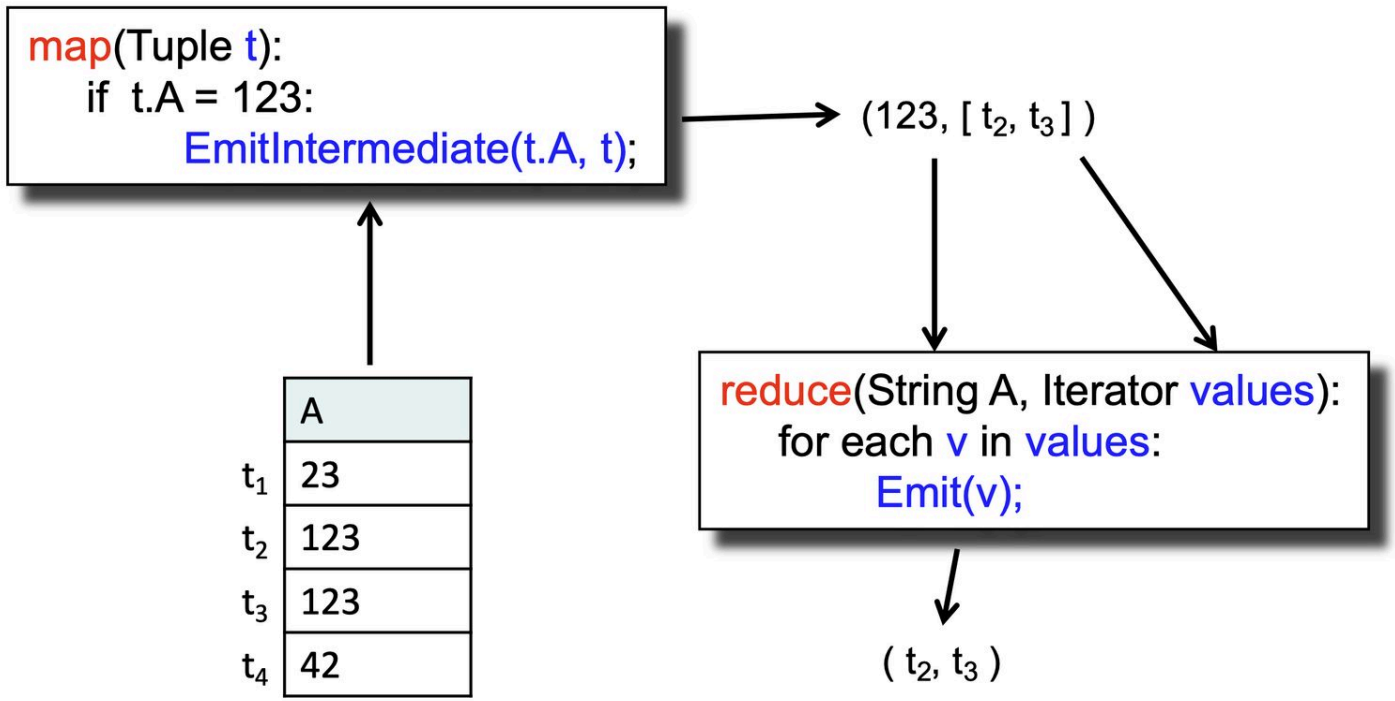
```
R = [(123,21), (456,10), (123,4)]
S = [(21,67), (21,80), (4,9), (5,99)]
```



R has three rows; S has two different rows with C = 21

Selection in MapReduce

SELECT * FROM R WHERE A = 123 keeps only the two matching rows



The mapper checks the condition and emits the retained rows

A Map-Only Selection

A filter can finish in map; Hadoop supports map-only jobs by setting reducers to zero

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

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

Only rows passing A = 123 need to be emitted

Selection: Code and Retained Rows

A WHERE condition decides whether each input row emits an output row

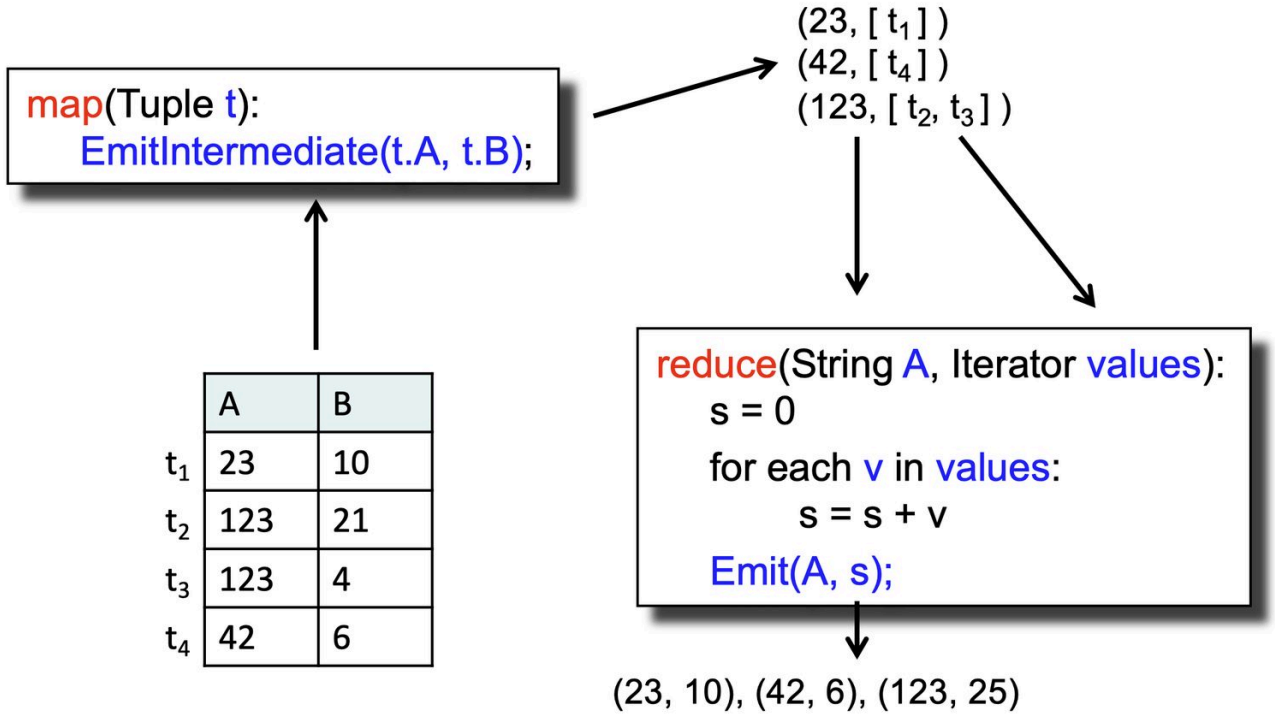
```
# SELECT * FROM R WHERE A = 123
def select_row(a, b):
    if a == 123:
        yield (a, b)
```



Filtering needs no cross-partition grouping

Grouping in MapReduce

SELECT A, SUM(B) FROM R GROUP BY A adds B values within each A



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

Key 123 groups values 21 and 4; reduce returns 25

GROUP BY: Choose the Key and Value

The grouping column becomes the key; the quantity to sum becomes the value

```
# SELECT A, SUM(B) FROM R GROUP BY A
def map_sum(a, b):
    yield (a, b)

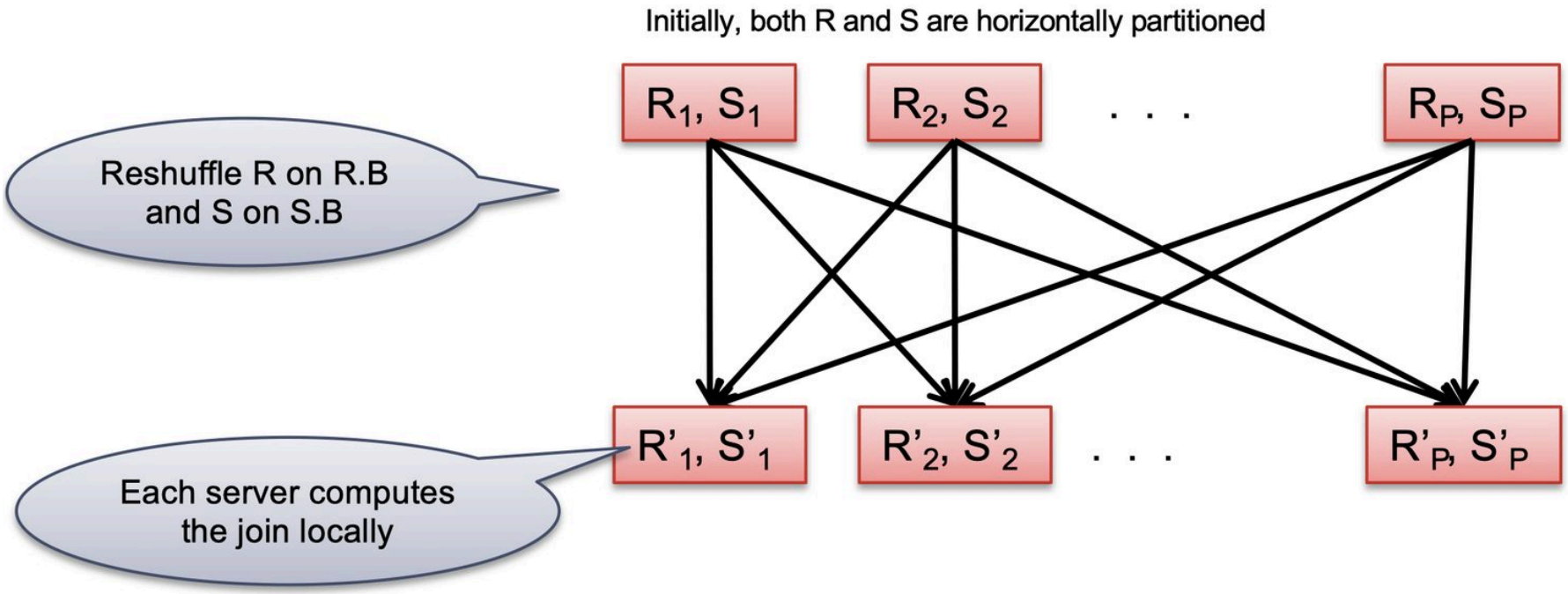
def reduce_sum(a, values):
    yield (a, sum(values))
```



Changing emitted values from B to 1 changes SUM into COUNT

Shuffle Join: Move Both Inputs

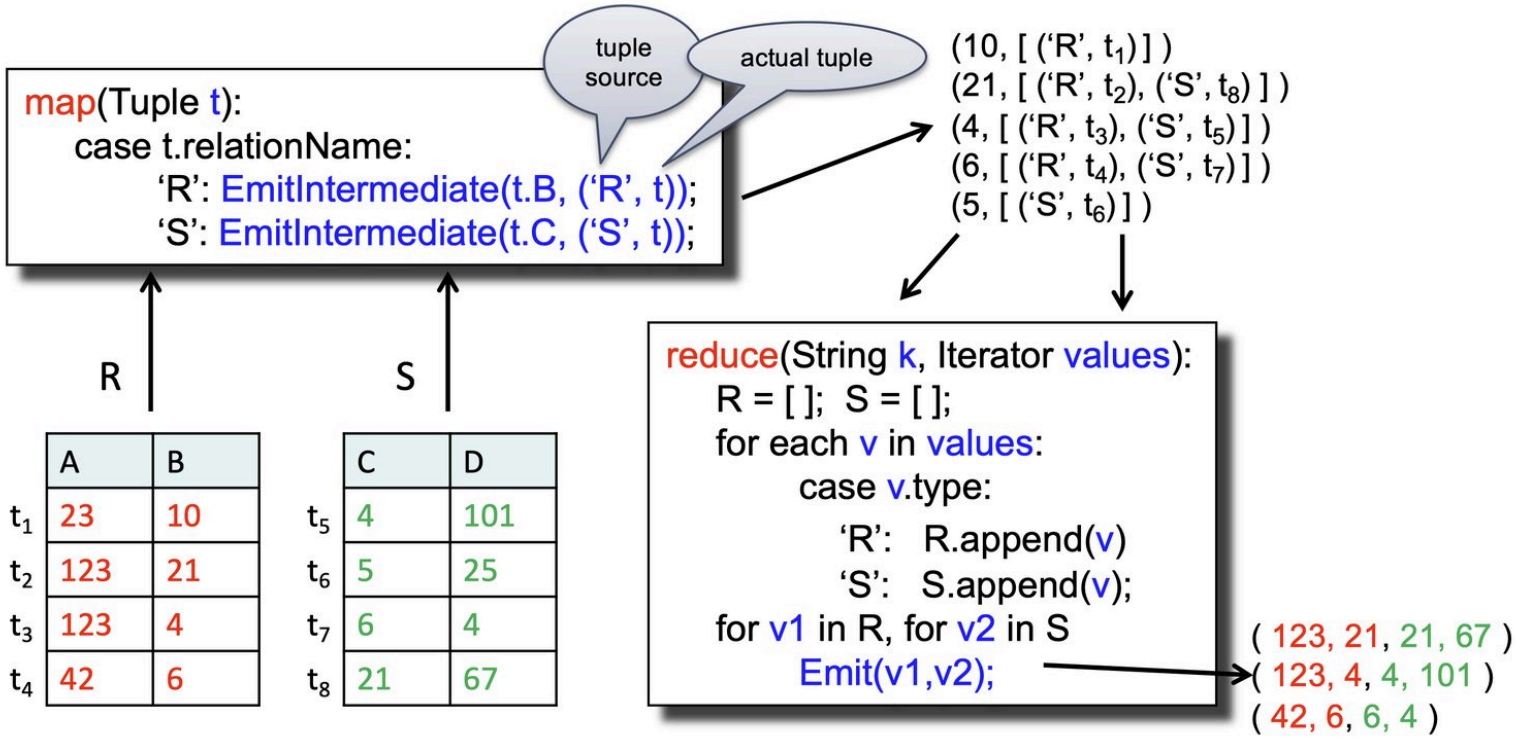
Matching join keys must meet even when the two input tables start on different servers



Redistribute both inputs by the join key, then join locally

Shuffle Join: Match Rows Within a Key

A source tag tells reduce which rows came from R and which came from S



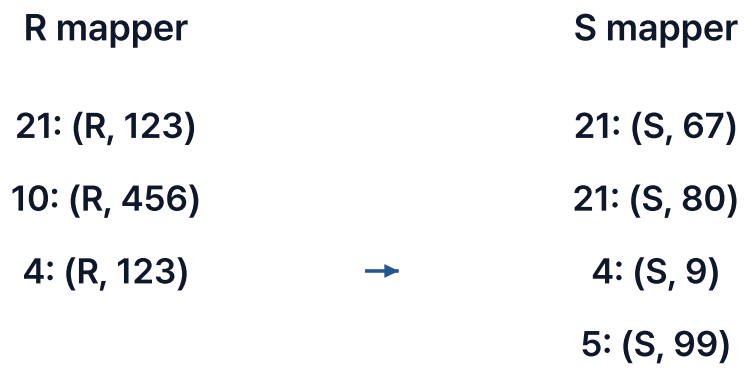
Within one key, reduce pairs each R row with each matching S row

JOIN: Emit the Join Key and a Source Tag

Both mappers emit the same kind of key: R.B on the left and S.C on the right

```
# SELECT R.A, R.B, S.D
# FROM R JOIN S ON R.B = S.C
def map_r(a, b):
    yield (b, ("R", a))

def map_s(c, d):
    yield (c, ("S", d))
```



Key 21 brings one R value and two S values together

JOIN: Pair the Values Within One Key

The nested loops produce every R × S pair within one key

```
def reduce_join(key, tagged):
    rows = list(tagged)
    rs = [v for t, v in rows if t == "R"]
    ss = [v for t, v in rows if t == "S"]
    for a in rs:
        for d in ss:
            yield (a, key, d)
```



One reducer call can emit zero, one or many joined rows

JOIN: All Keys and the Final Result

An inner join emits only keys with rows on both sides

Rows per key		Final (A, B, D)
21: 1 R × 2 S = 2		(123, 21, 67)
4: 1 R × 1 S = 1		(123, 21, 80)
10: 1 R × 0 S = 0	→	(123, 4, 9)
5: 0 R × 1 S = 0		

The three output rows preserve both matches for key 21

Caveat: Duplicate Join Keys

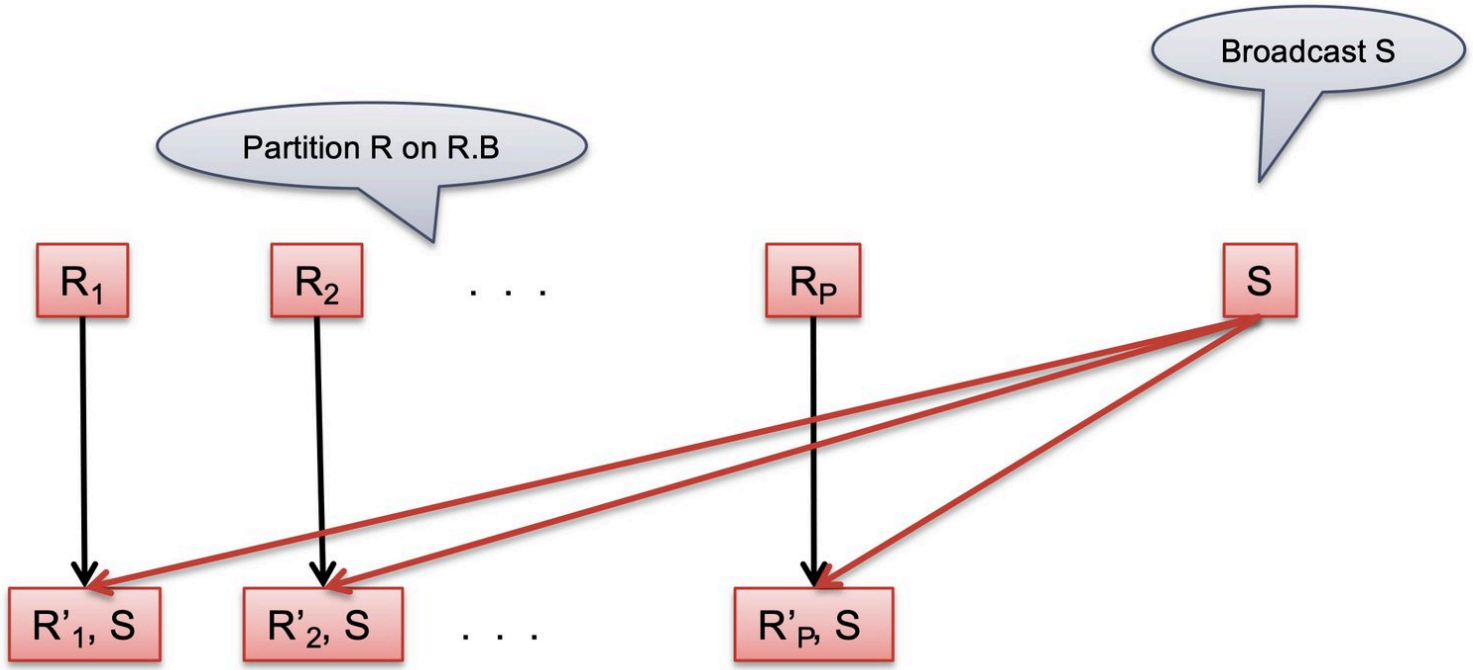
Example change: add (789, 21) to R; inspect key 21 only

R values at key 21	S values at key 21	All four pairs
A = 123	D = 67	(123, 21, 67)
A = 789	D = 80	(123, 21, 80)
		(789, 21, 67)
		(789, 21, 80)

Two rows on each side create four combinations

Broadcast Join: Copy the Small Input

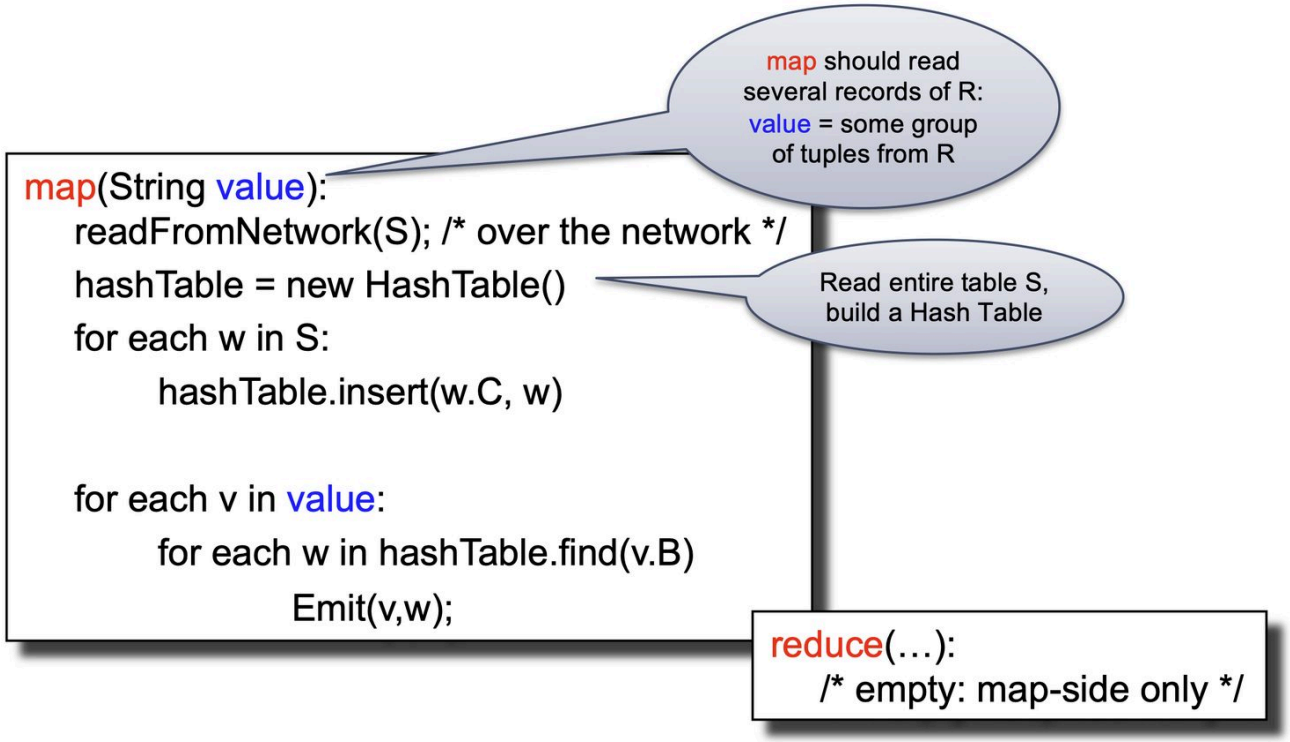
If S is small enough, each worker can receive a copy while R stays partitioned



Copy S to the workers instead of shuffling the large input

Broadcast Join: Work Inside Map

Each mapper builds a lookup table for S and probes it with its own R rows



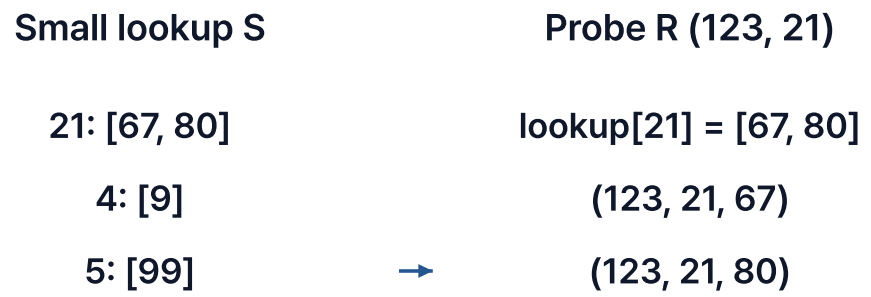
The join can finish in map when each worker holds the small input

Broadcast JOIN: Keep Every Match

Each worker probes a local copy of the small table using its own R rows

```
from collections import defaultdict
lookup = defaultdict(list)
for c, d in S:  # The small input
    lookup[c].append(d)

def join_local(a, b):
    for d in lookup.get(b, []):
        yield (a, b, d)
```



A list per key preserves the two matches; a single-value dictionary loses one

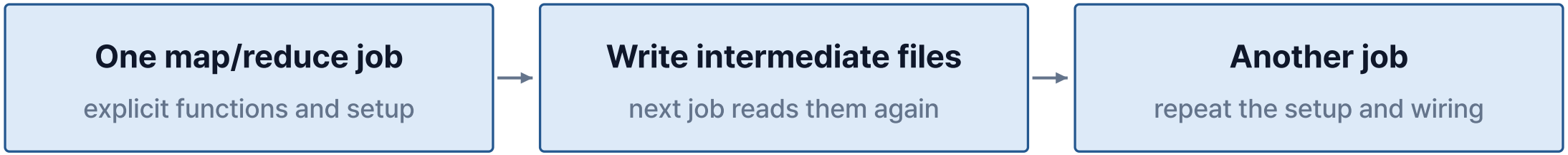
Two Join Strategies

The data sizes help determine which input movement is practical

	Shuffle join	Broadcast join
Input movement	Redistribute both inputs by key	Copy the small input to each worker
Useful when	Both inputs are large	One input fits in each worker’s memory
Main cost	Network transfer and uneven keys	Copies and memory for the small input

Why Move Beyond MapReduce

More complex calculations require more job code and intermediate data handling

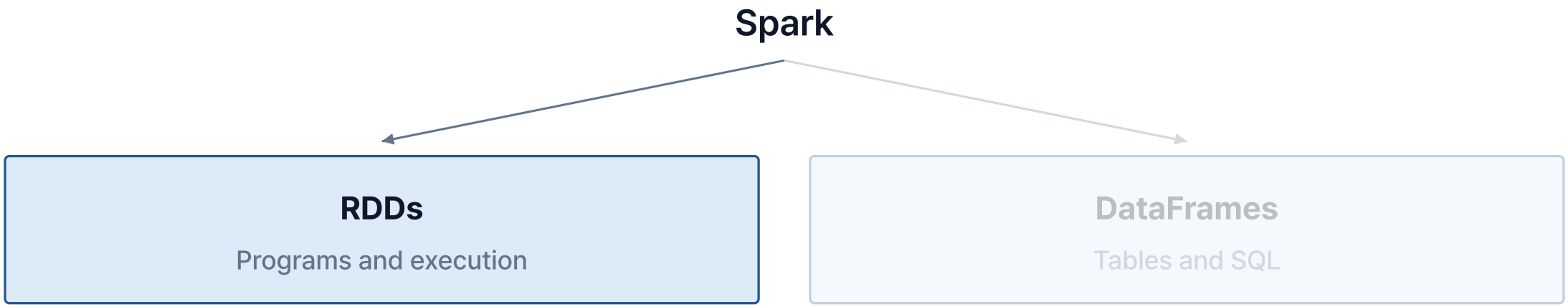


- Complex queries need several connected jobs
- You write the steps and the intermediate data flow

02 · Spark

01	MapReduce	write map and reduce functions for distributed jobs
02	**Spark**	compose RDD operations, then describe table queries with DataFrames

§ RDDs


Spark
RDDs
Programs and execution
DataFrames
Tables and SQL

Spark Begins at Berkeley

A university research project grew into an open-source processing engine



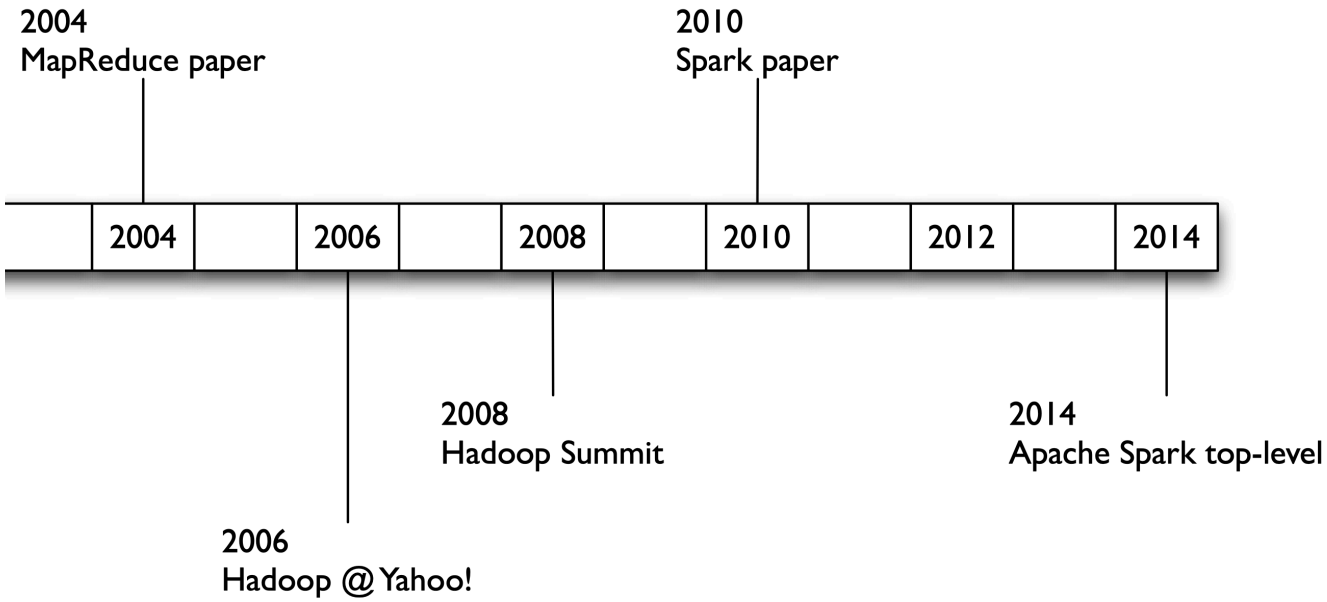




- 2009: research at UC Berkeley
- 2010: open-source release and the Spark paper
- 2013: moved to Apache; 2014: became a top-level Apache project

Spark: A Brief History

The MapReduce paper appeared in 2004; the Spark paper followed in 2010



A research idea became an open-source project and a broader community

A Shorter Word Count Program

The same word-count task needs much less job wiring in the Spark example



WordCount in 50+ lines of Java MR

```
val f = sc.textFile(inputPath)
val w = f.flatMap(l => l.split(" ")).map(word => (word, 1)).cache()
w.reduceByKey(_ + _).saveAsText(outputPath)
```

WordCount in 3 lines of Spark

Compose operations instead of wiring a new job around each pair of functions

Word Count with Spark RDDs

The same Word Count now uses a chain of distributed collection operations

```
sc = spark.sparkContext
lines = sc.parallelize([
    "spark makes data simple",
    "spark processes data"], 2)
words = lines.flatMap(lambda s: s.split())
pairs = words.map(lambda w: (w, 1))
counts = pairs.reduceByKey(
    lambda a, b: a + b)
result = sorted(counts.collect())
```

Same two documents

2 lines

7 words

7 (word, 1) pairs

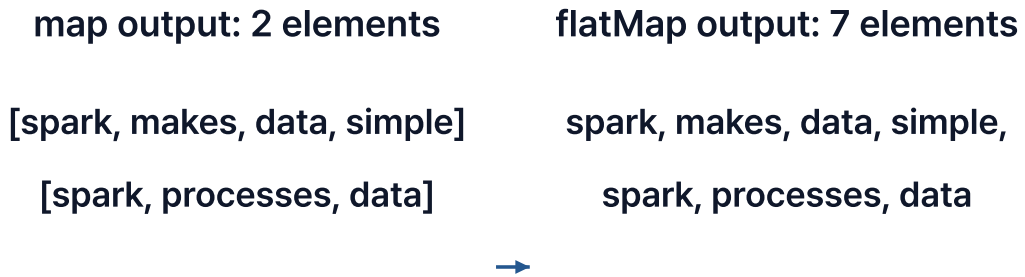
5 (word, count) pairs

Variable names identify the intermediate datasets in the graph

flatMap: Lines Become Words

map preserves one output element per line; flatMap flattens each line’s word list

```
nested = lines.map(lambda s: s.split())
words = lines.flatMap(lambda s: s.split())
```

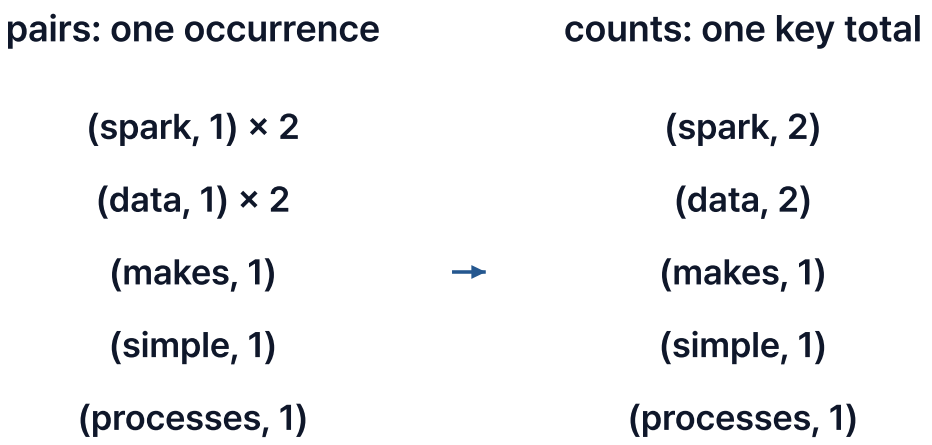


The function is the same; the resulting element type changes

map and reduceByKey: Word Counts

reduceByKey combines values for each word; it may combine locally before the shuffle

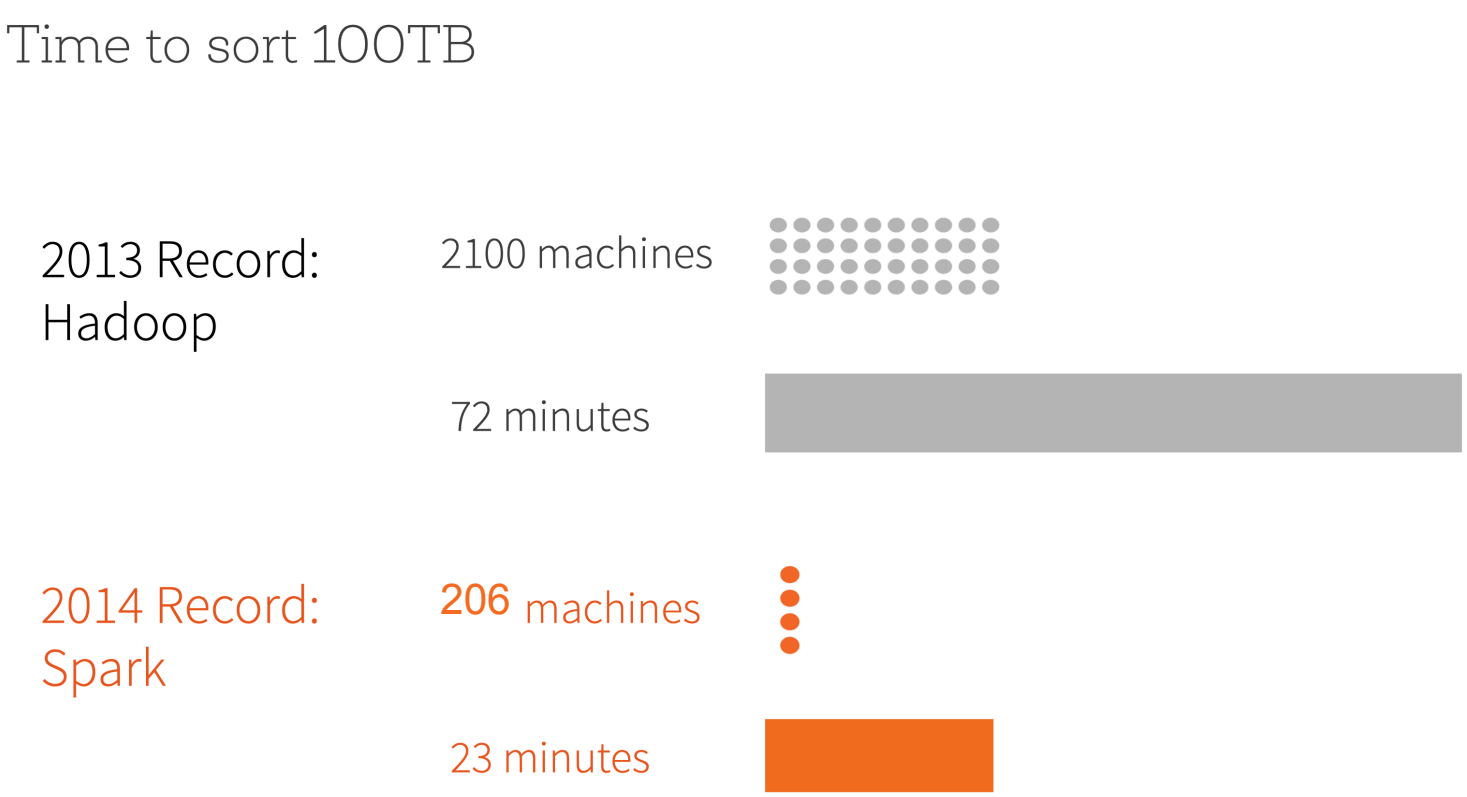
```
pairs = words.map(lambda w: (w, 1))
counts = pairs.reduceByKey(
    lambda a, b: a + b)
```



The result has five keys, and the counts still sum to seven

Spark's 2014 Sorting Record

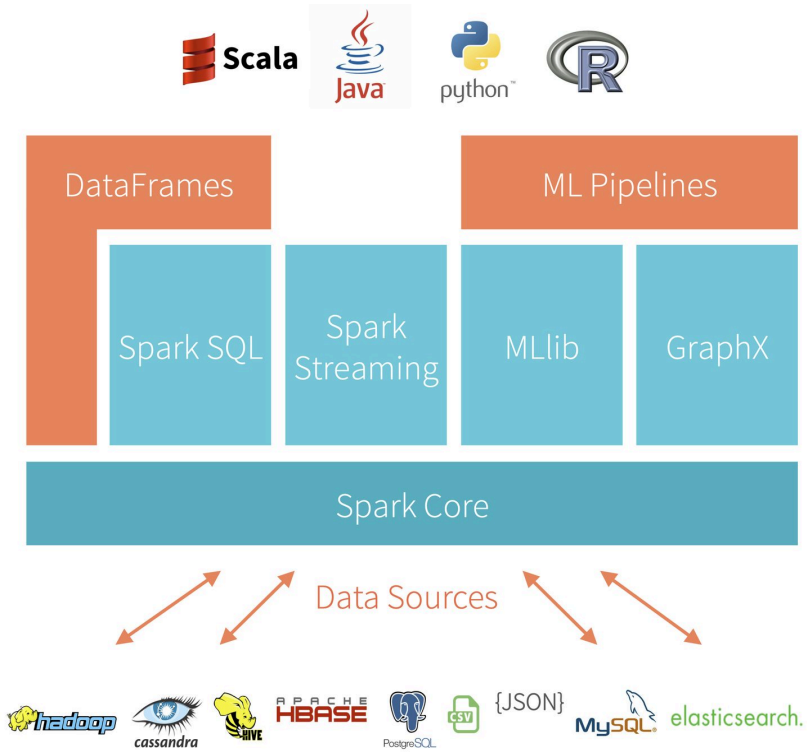
Historical sorting records drew attention to Spark as a practical processing engine



Different clusters and runs: 72 minutes in 2013, 23 minutes in 2014

Spark Libraries and Languages

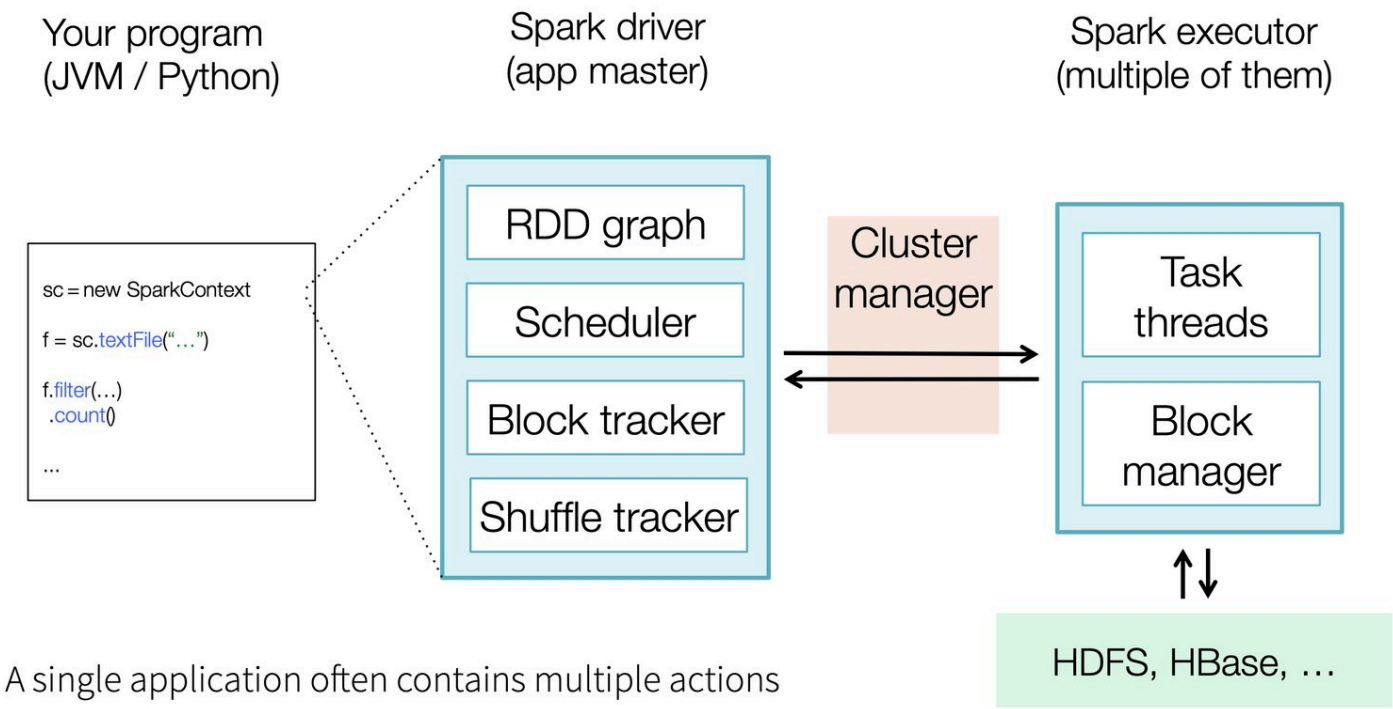
One execution engine supports several kinds of work and several programming languages



Today’s route uses Python, DataFrames and Spark SQL

Driver, Executors and Storage

The driver coordinates the program; executor processes run tasks on the data

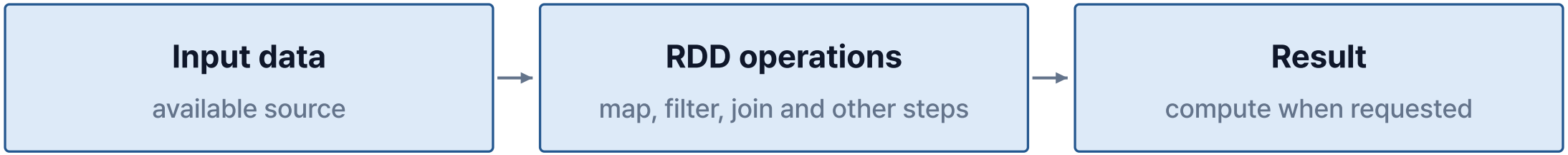


Spark handles the execution machinery behind the shorter program

Resilient Distributed Datasets

An RDD gives distributed data a programmable collection interface

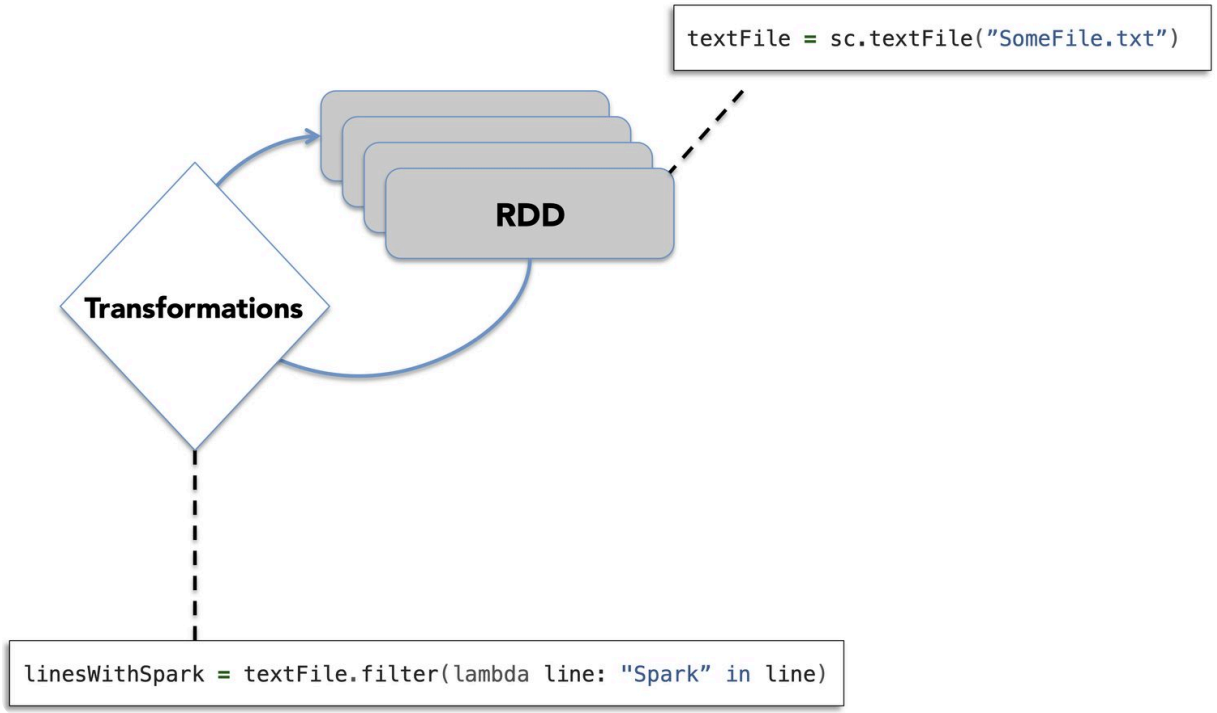
RDD: an immutable collection divided into partitions, with a record of how to rebuild it



- Distributed: partitions can run on different workers
- Resilient: lineage records how lost partitions can be recomputed

Transformations Build Another RDD

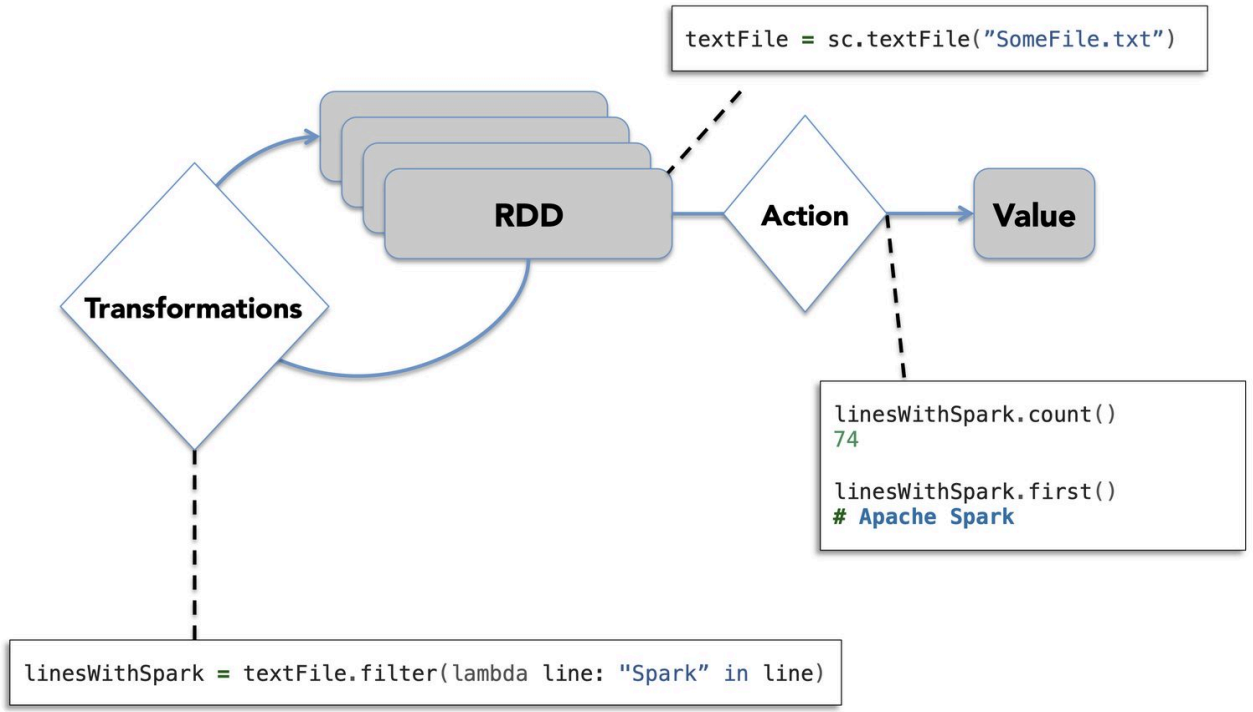
A transformation such as filter describes a new distributed collection



The program builds the calculation before requesting its result

Actions Request Results

An action such as count() requests a value and triggers the required computation

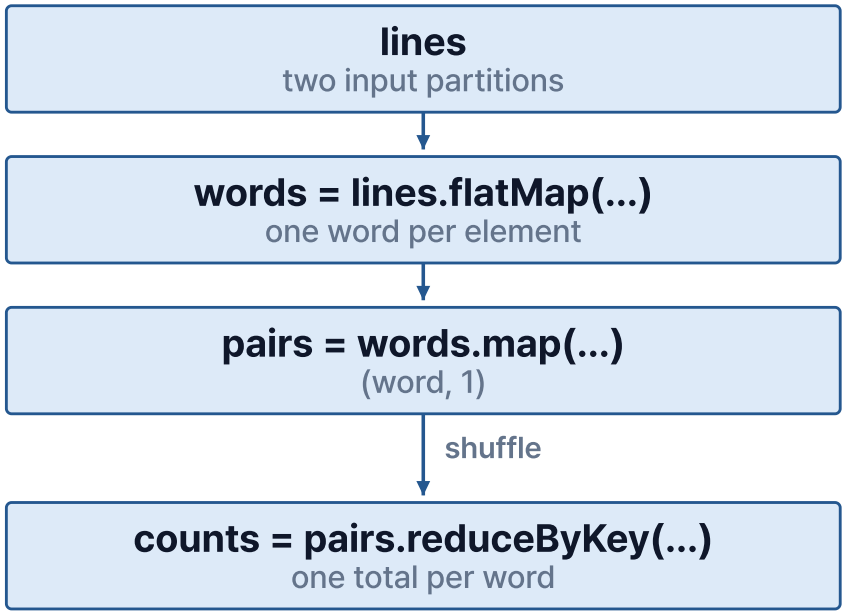


Transformations return RDDs; actions return values or save output

Lineage: Code Becomes a Dependency Graph

Lineage records parent RDDs and operations; the DAG shows these dependencies

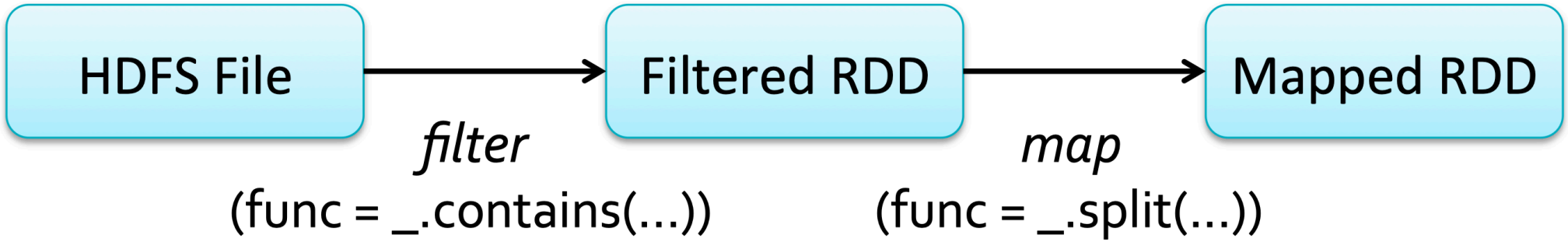
```
# Reuse lines from the Word Count example
words = lines.flatMap(lambda s: s.split())
pairs = words.map(lambda w: (w, 1))
counts = pairs.reduceByKey(
    lambda a, b: a + b)
```



The graph uses the same variable names and functions as the code

Lineage: File, Filter and Map

A file, a filter and a map form the same kind of dependency chain



Parent data and recorded functions describe how to produce each derived RDD

An Action Executes the Required DAG

An action requests the answer and runs the calculation defined by the transformations

```
# Transformations above define counts
result = sorted(counts.collect())
# [("data", 2), ("makes", 1),
#  ("processes", 1), ("simple", 1),
#  ("spark", 2)]
```

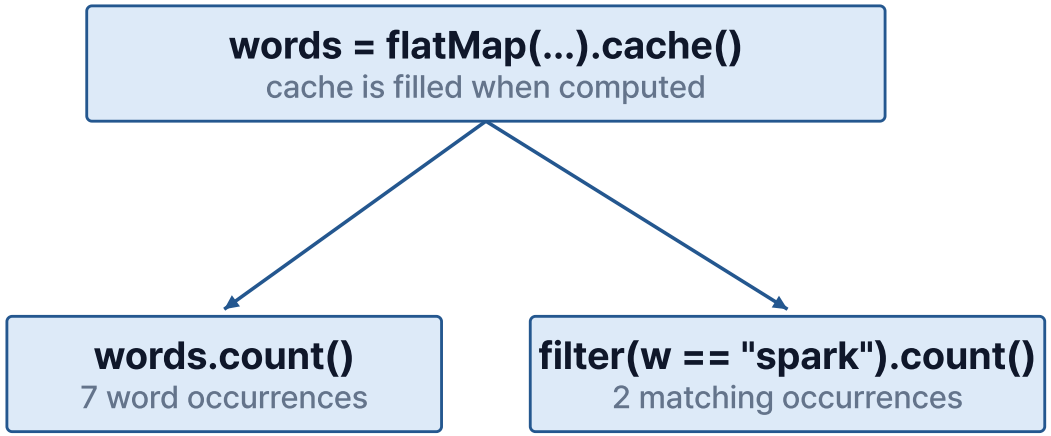
Define the calculation		Request the answer
lines → words		counts.collect()
words → (word, 1)		run the required operations
group and add by word	→	return five word counts

Transformations describe the calculation; an action requests its result

Cache and Two Actions on One RDD

Two actions reuse the same intermediate words; cache and lineage serve different roles

```
words = lines.flatMap(lambda s:
s.split()).cache()
n = words.count()  # 7, fills the cache
k = words.filter(lambda w: w ==
"spark").count()
# k == 2; reuse available cached words
words.unpersist()
```

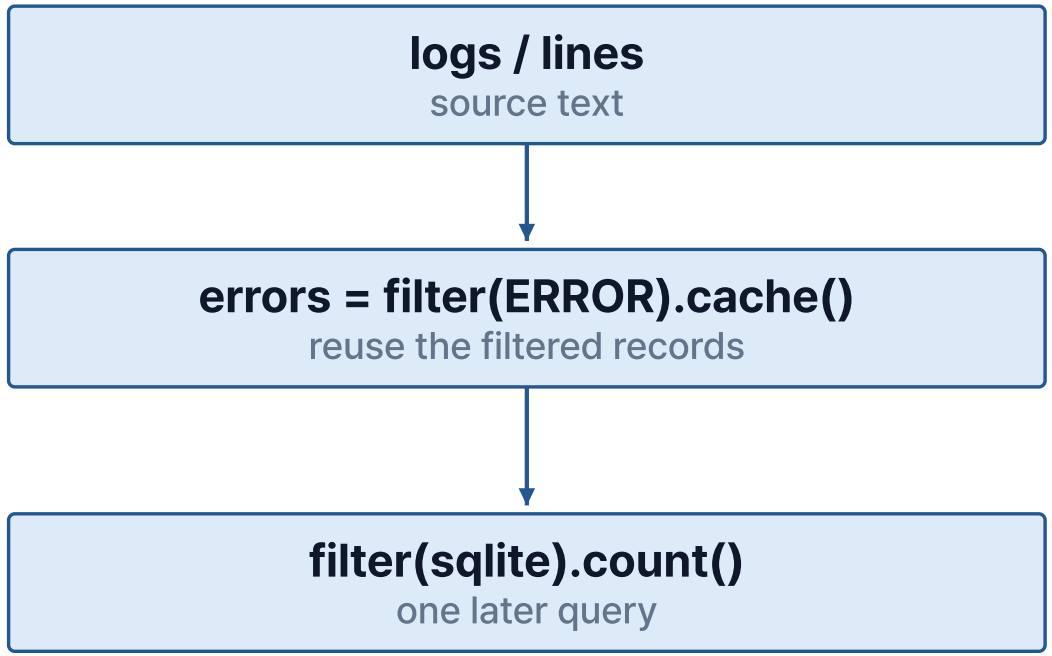


The first action counts all seven words; the second counts two spark occurrences

Cache: Repeated Log Queries

The filtered log records can serve more than one later query

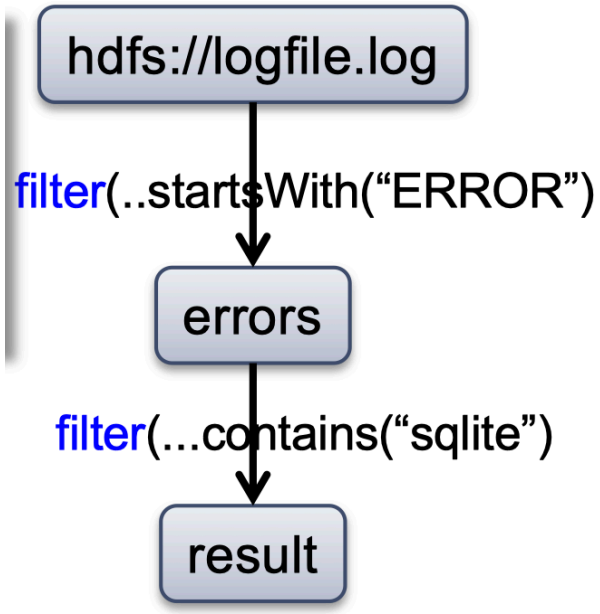
```
lines = sc.textFile("logs/")
errors = lines.filter(
    lambda s:
s.startswith("ERROR")).cache()
errors.count()
errors.filter(lambda s: "sqlite" in
s).count()
errors.unpersist()
```



The first action computes errors; later queries can reuse its cached partitions

Cache: Reusing Filtered Logs

This log diagram shows where an intermediate result can be reused



Cached errors are computed data; lineage remains the recipe for recomputation

Lineage and One Missing Partition

Spark uses available parent data and recorded operations to recompute missing work

Recorded dependency		If its pairs are missing
input partition P1		read available P1 again
flatMap(split)		apply the recorded functions
map(word, 1)	→	rebuild the missing output

Lineage describes a recipe; cache holds computed data

The Same Join in Spark

Keep R.A = 123, match R.B with S.C, then count the three joined rows

```
# Same R(A, B) and S(C, D)
# Keep A = 123, then join B = C
# Expected (A, B, D):
# (123, 21, 67)
# (123, 21, 80)
# (123, 4, 9)
```

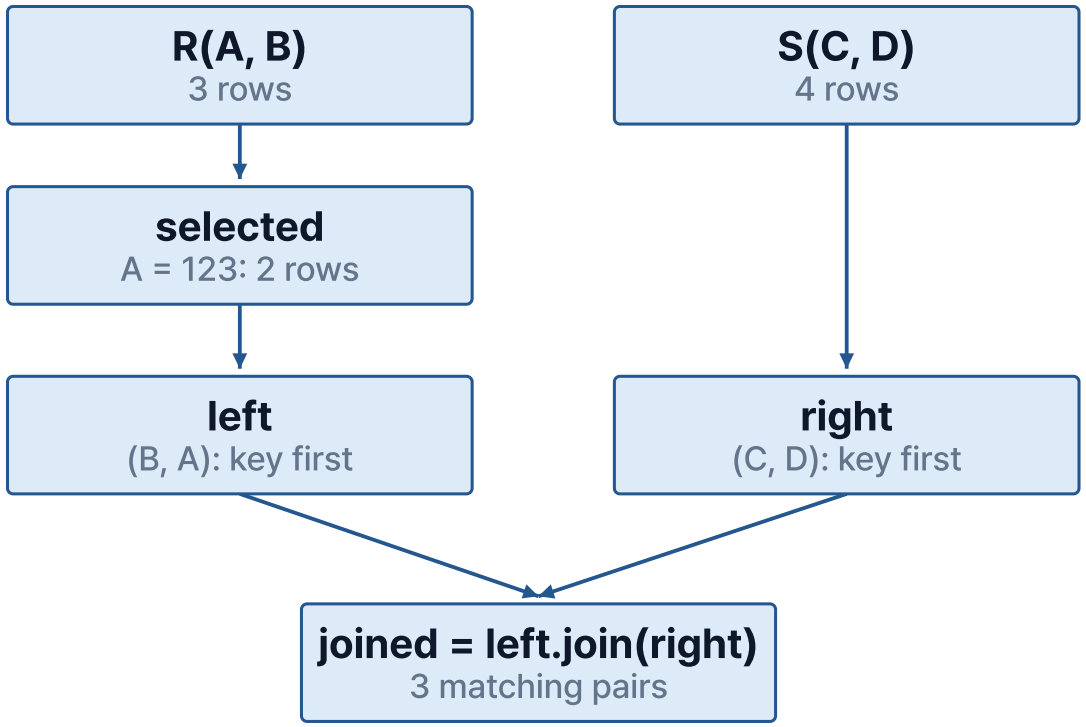


The two matches at key 21 must survive in every API

A Join Query as RDD Operations

Pair RDD join uses the first tuple element as its key; re-key R from (A, B) to (B, A)

```
R = sc.parallelize([(123,21),(456,10),
(123,4)])
S = sc.parallelize([(21,67),(21,80),(4,9),
(5,99)])
selected = R.filter(lambda r: r[0] == 123)
left = selected.map(lambda r: (r[1], r[0]))
right = S  # Already (C, D)
joined = left.join(right)
answer = joined.count()  # 3
```



The two input paths meet at join; count requests the joined row count

RDD JOIN: Read the Nested Output

join preserves the key and nests the two values; map arranges the desired output columns

```
rows = joined.map(
    lambda kv: (kv[1][0], kv[0], kv[1][1]))
# Result elements are now (A, B, D)
result = sorted(rows.collect())
```

join: (key, (left, right))		map: (A, B, D)
(21, (123, 67))		(123, 21, 67)
(21, (123, 80))		(123, 21, 80)
(4, (123, 9))	→	(123, 4, 9)

The record structure explains every index in the lambda

Case study: A Filtered Join

This query filters both inputs before joining on their shared A key

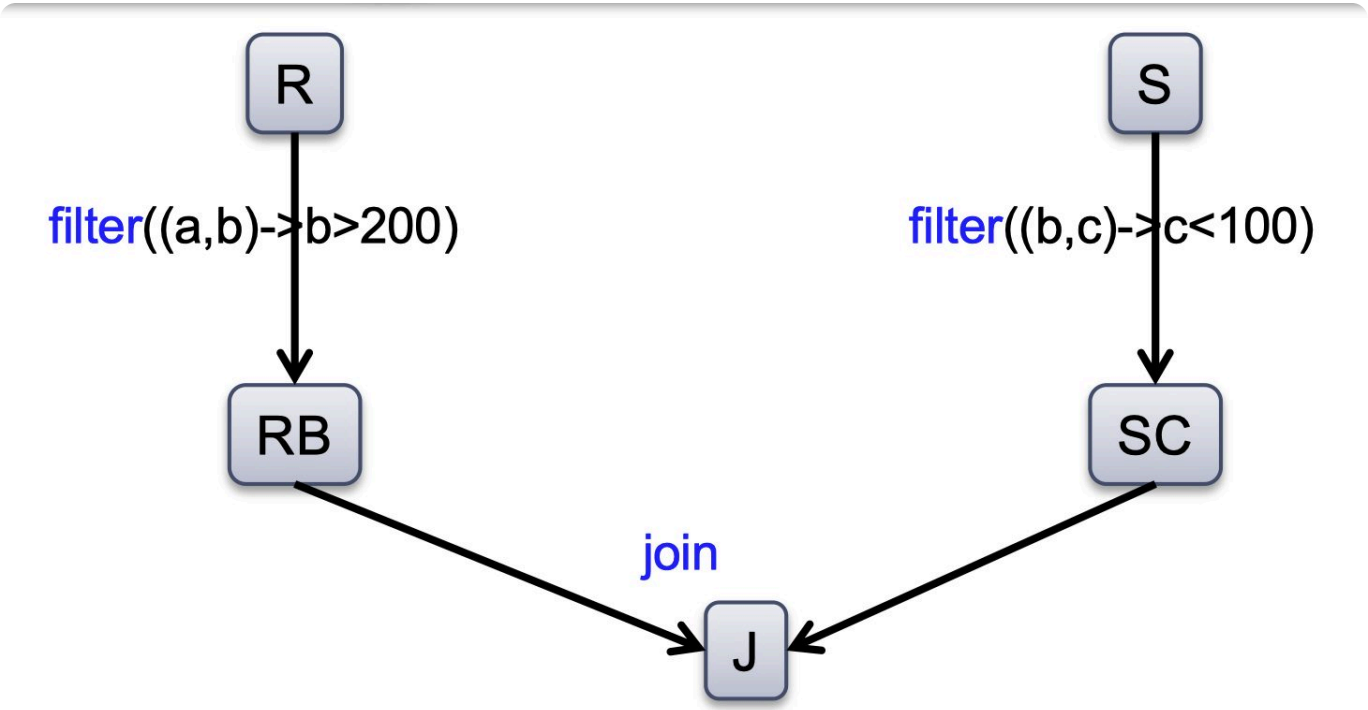
```
# R(A, B) and S(A, C)
R = sc.parallelize([(1,250),(2,180),
(3,300)])
S = sc.parallelize([(1,50),(1,120),(3,80)])
RB = R.filter(lambda r: r[1] > 200)
SC = S.filter(lambda s: s[1] < 100)
J = RB.join(SC)
answer = J.count()  # 2
```



Two retained keys produce two joined rows

Case study: The Filtered Join DAG

R and S follow separate filter paths and meet at the join



The two input branches explain the structure of the filtered-join code

Case study: The Same Filtered Join as Tables

RDD functions, DataFrame columns and SQL express the same filtered join

```
R = spark.createDataFrame(
    [(1,250),(2,180),(3,300)], ["A","B"])
S = spark.createDataFrame(
    [(1,50),(1,120),(3,80)], ["A","C"])
J = R.filter(R.B > 200).join(
    S.filter(S.C < 100), on="A")
answer = J.count()  # 2
```

Equivalent SQL

```
SELECT COUNT(*)
FROM R JOIN S
ON R.A = S.A
WHERE R.B > 200
AND S.C < 100
```

→

Joined (A, B, C)

(1, 250, 50)

(3, 300, 80)

Both the RDD program and the table query count two matches

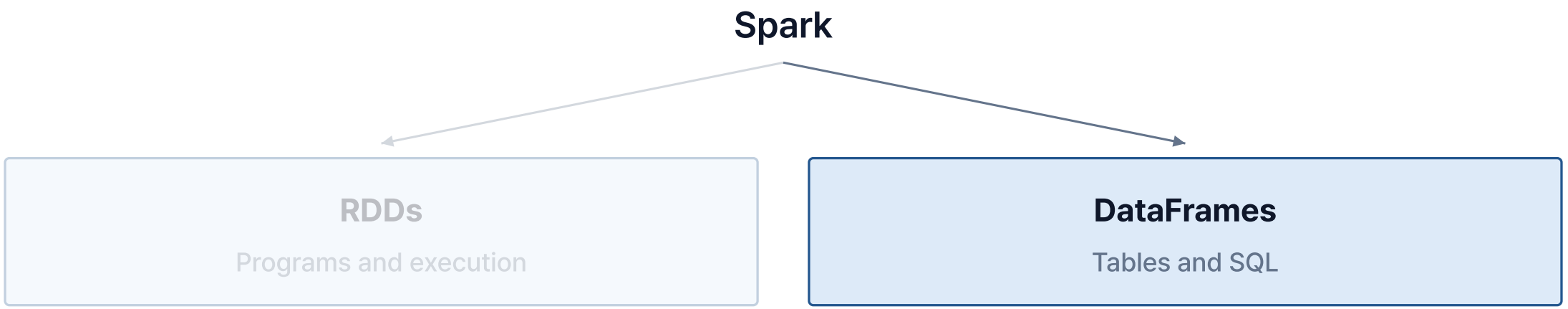
What RDDs Simplify

RDDs remove much of the job wiring, but the program still works with individual objects

MapReduce program	Spark RDD program
Write mapper and reducer classes	Call map, filter, join or reduceByKey
Wire a sequence of jobs and intermediate files	Compose operations in one program
Manage key/value logic for each operation	Reuse built-in distributed operations

We still write functions and choose the operation sequence

§ DataFrames


Spark
RDDs
Programs and execution
DataFrames
Tables and SQL

Why DataFrames

Named columns let us express the query as table operations

```
R.filter(lambda row: row[1] > 200)
```

What is row[1]? The programmer must know the record layout.

- A DataFrame gives columns names and types
- Table operations describe the query in a form Spark can optimize

The Same Join with DataFrames

Column names replace tuple positions and manual re-keying

```
R = spark.createDataFrame(
    [(123,21),(456,10),(123,4)], ["A","B"])
S = spark.createDataFrame(
    [(21,67),(21,80),(4,9),(5,99)],
["C","D"])
rows = (R.filter(R.A == 123)
    .join(S, R.B == S.C)
    .select("A", "B", "D"))
answer = rows.count()  # 3
```



filter chooses rows; select chooses output columns

The Same Join in Spark SQL

The same table query produces the same three rows through Spark SQL

```
R.createOrReplaceTempView("R")
S.createOrReplaceTempView("S")
rows = spark.sql("""
    SELECT R.A, R.B, S.D
    FROM R JOIN S ON R.B = S.C
    WHERE R.A = 123
""")
rows.show()
answer = rows.count()  # 3
```

All three APIs agree

(123, 21, 67)

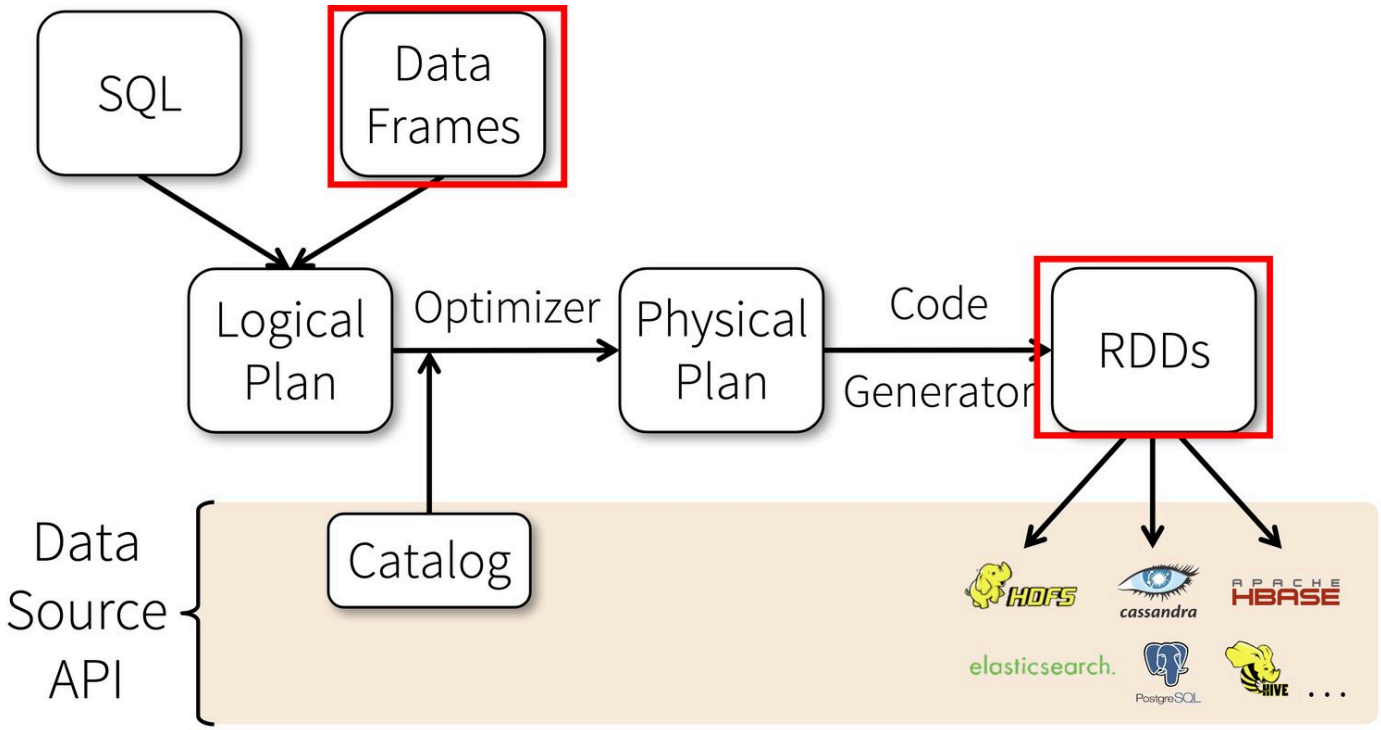
(123, 21, 80)

(123, 4, 9)

The engine chooses the physical join strategy for the table query

DataFrames and the Optimizer

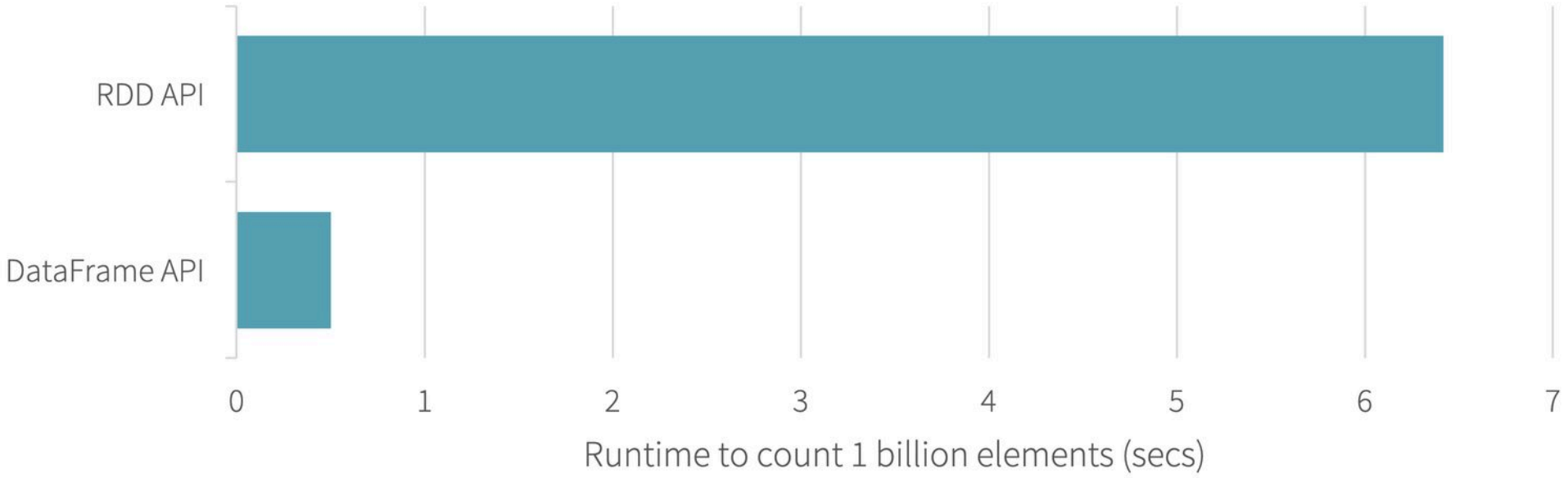
SQL and DataFrame expressions produce a plan the optimizer can inspect



Describe the table query; let Spark choose an execution plan

A Historical DataFrame Performance Example

This historical count benchmark illustrates how a table API can enable optimized execution



The pictured workload is one example; the speed difference depends on the query and environment

MapReduce, RDDs and DataFrames

Each step lets us express more of the task through a higher-level interface

	MapReduce	Spark RDD	Spark DataFrame / SQL
You write	map/reduce functions and job setup	Functions and operation chains	Columns, table operations or SQL
You manage	Keys, source tags and job boundaries	Object layout and operation sequence	Schema and query meaning
The engine provides	Distribution, shuffle and retries	Distributed operators and lineage	Those execution services plus SQL optimization

Takeaway: MapReduce, RDDs and DataFrames

1. **MapReduce**: write map/reduce functions, keys and job setup
2. **Spark RDDs**: compose distributed operations in a shorter program
3. **DataFrames and SQL**: describe named-column queries and use the optimizer