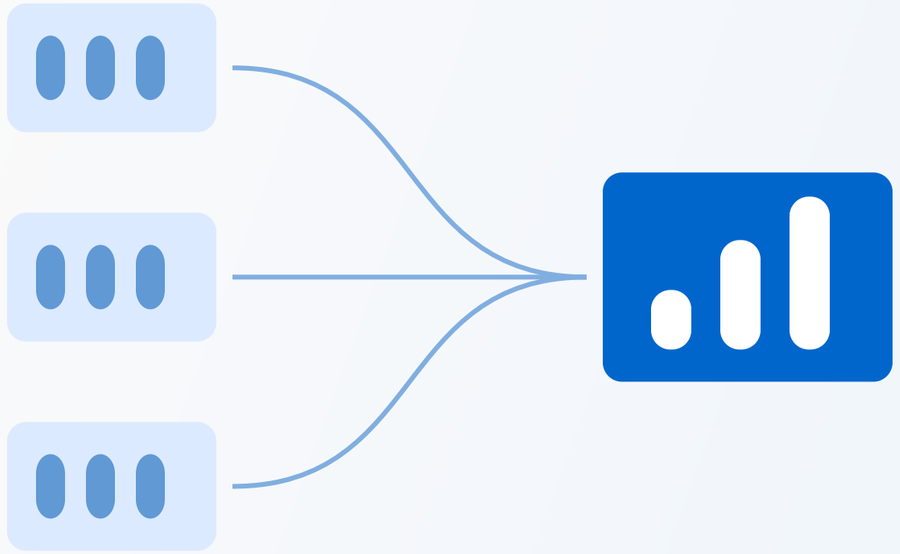
# Spark

From a sales question to a checked Spark result



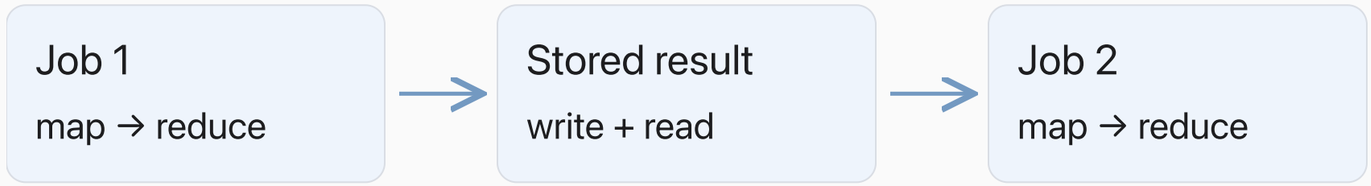
# Why Spark?

**Spark plans a connected sequence of distributed operations, from reading data to writing results.**

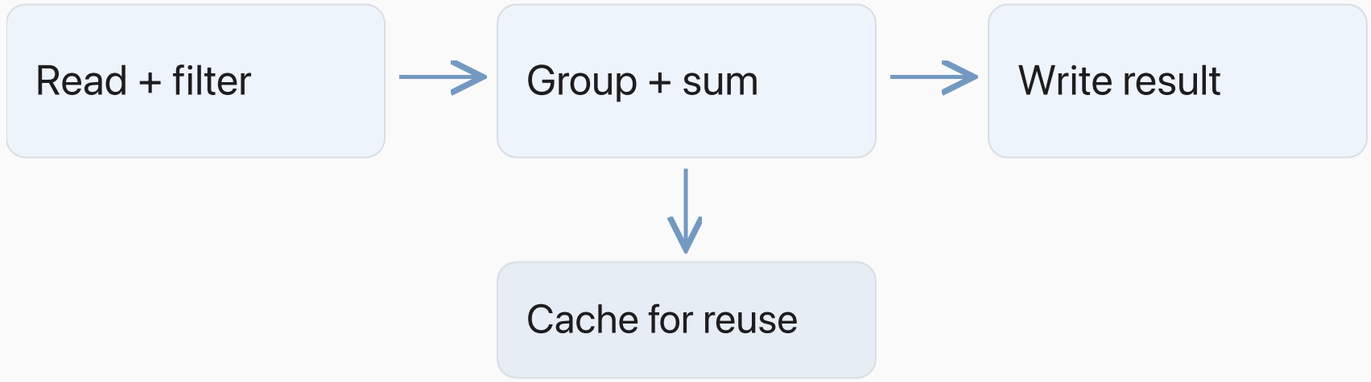
It can pipeline compatible work and reuse cached results across queries.

Shuffles, spills and durable writes still use network or disk. Performance depends on the workload.

## Chained MapReduce jobs




## Spark: connected operations

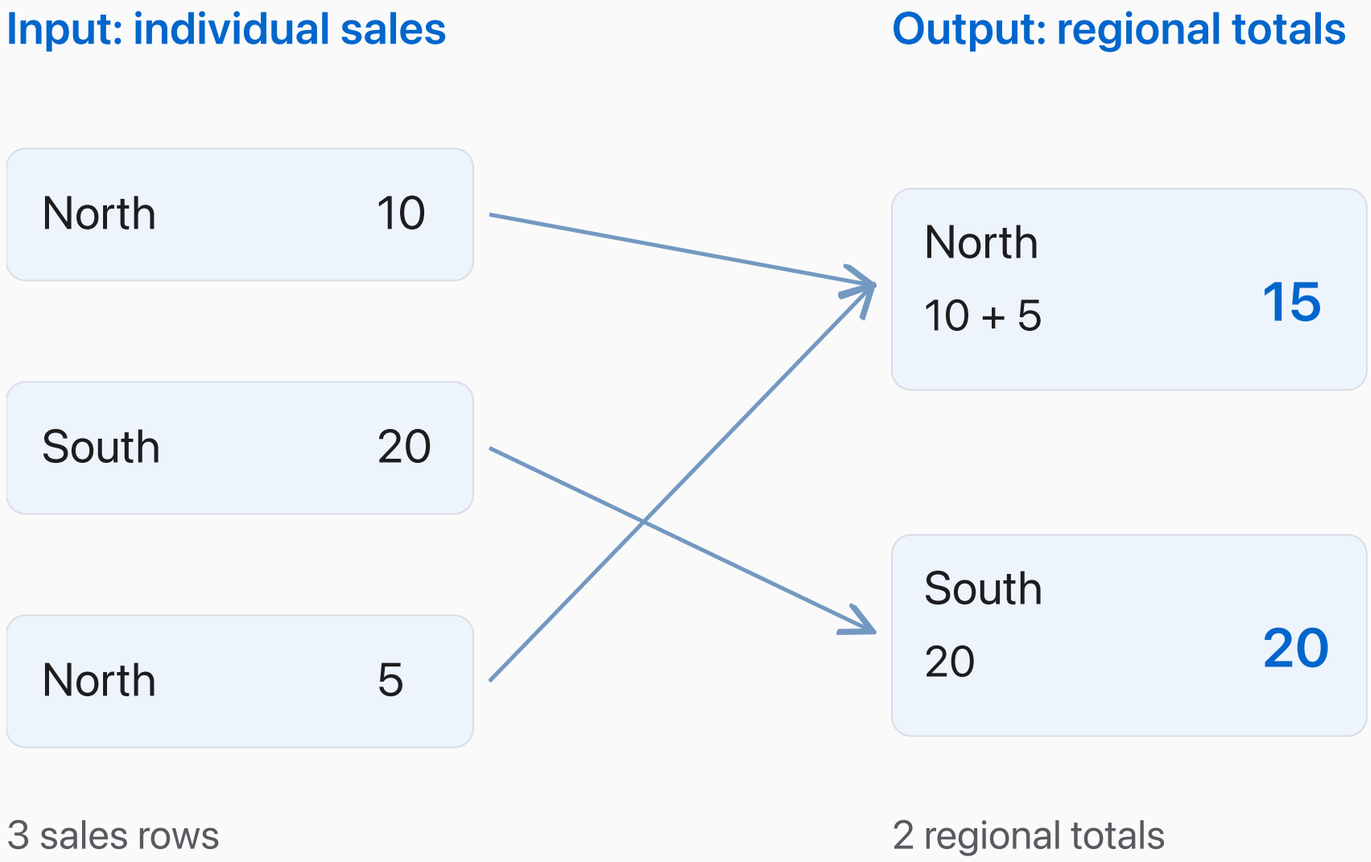


# Sales by region

**A retailer has three sales: North 10, South 20 and North 5. What total belongs to each region?**

One input row is one sale. One output row is one region, so North combines two sales.

The checked answer is North 15 and South 20; the total amount remains 35.



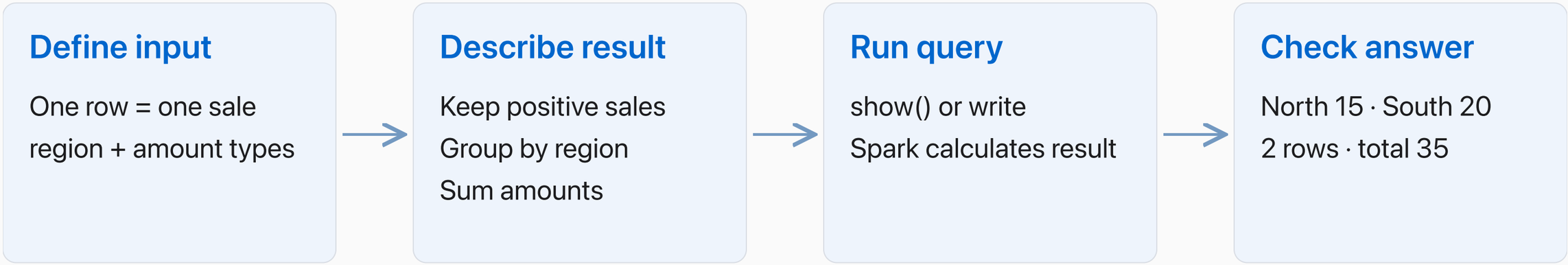
# Define the regional totals

Define the accepted sales and the meaning of each output row.

# The Spark workflow

Start with what one sale row means. Then describe the result, ask Spark to calculate it, and check the saved answer.

The required answer stays North 15 and South 20 through every step; names and types, operations, execution and checks must agree.



# RDDs and DataFrames

RDD (Resilient Distributed Dataset) holds distributed records. A DataFrame holds distributed rows with named, typed columns.

Both represent the same sales. Named DataFrame columns suit this regional query; RDD functions offer a lower-level record view.

## RDD

Tuples in this example

```
("North", 10)
("South", 20)
("North", 5)
```

Functions transform records

## DataFrame

Named columns + schema

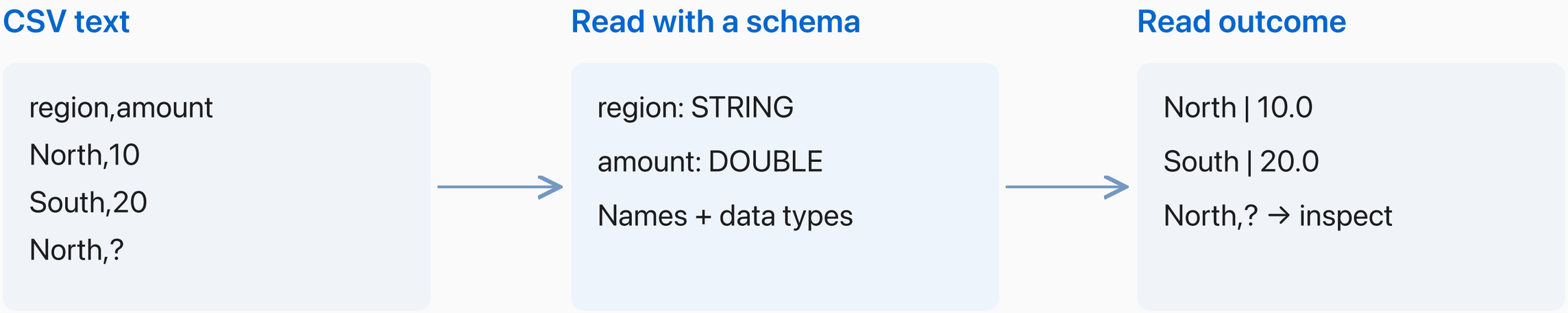
| region: STRING | amount: DOUBLE |
| --- | --- |
| North | 10 |
| South | 20 |
| North | 5 |

Expressions refer to named columns

# Schema

A schema states the name and data type of each column, so Spark can interpret CSV text as typed rows. In the notebook, spark is the SparkSession used to read it.

North,10 becomes region = North and amount = 10.0. North,? needs a data-quality check; declared types alone do not prove every value is valid.



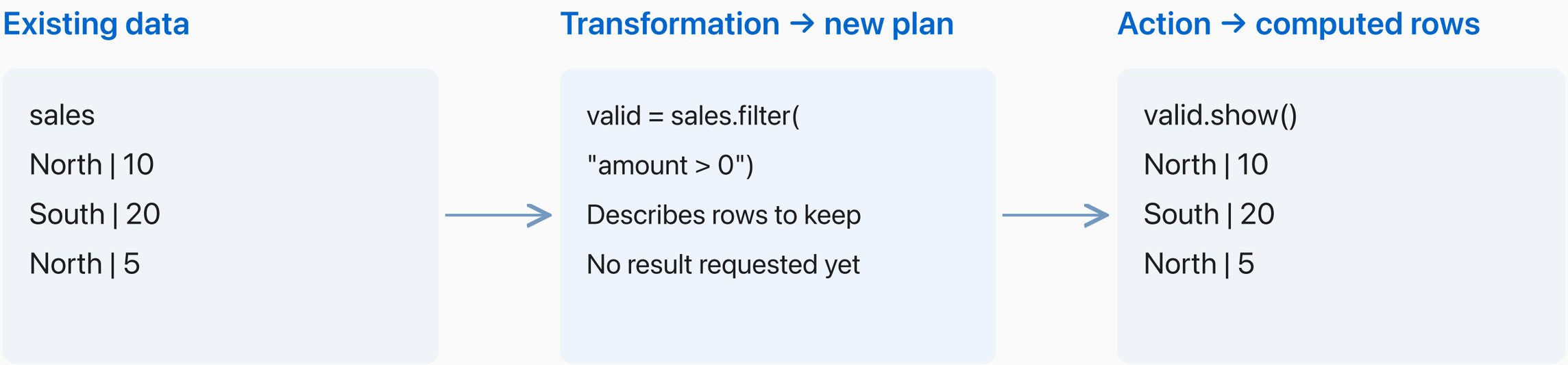

```
spark.read.schema("region STRING, amount DOUBLE").csv("sales.csv", header=True)
```

printSchema() reports types; inspect actual rows for invalid amounts.

# Transformations and actions

A transformation such as filter, select or groupBy describes a new distributed result. Spark records the work to do; the original DataFrame stays available.

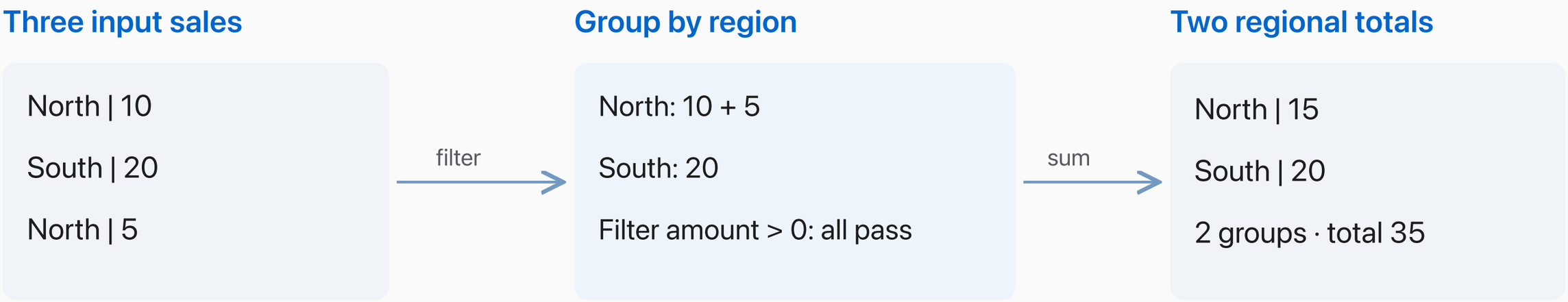
An action such as show(), count() or write needs a result. Spark then executes the required plan.



# Regional aggregation

Start with three sale rows: North 10, South 20 and North 5. Filtering amount > 0 keeps all three.

Grouping brings the two North amounts together; summing produces one row per region. The code beneath the diagram expresses these same steps.



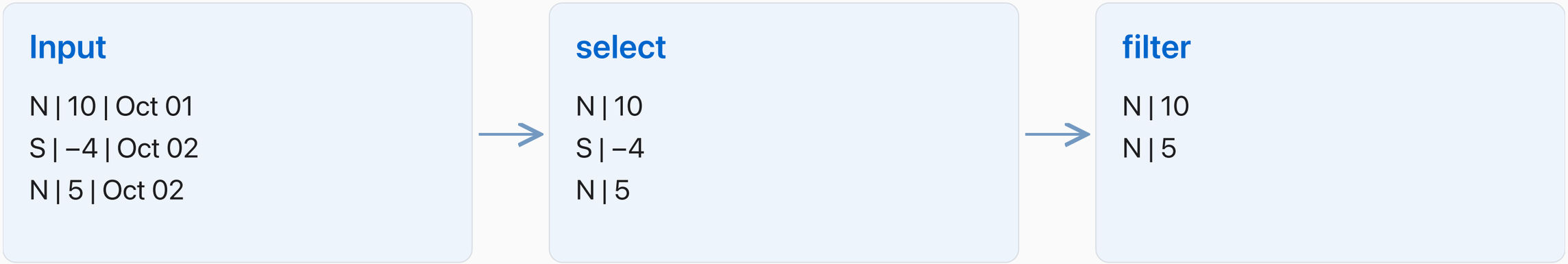

```
from pyspark.sql import functions as F
valid = sales.filter(F.col("amount") > 0)
totals = valid.groupBy("region").agg(F.sum("amount").alias("total"))
totals.show()
```

# Projection and filtering

Projection with select keeps chosen columns; filtering keeps rows that pass a condition. In this temporary variation, South −4 is filtered out.

Projection changes columns; filtering changes rows. This variation leaves North 10 and North 5.

select("region", "amount") → filter("amount > 0"). N = North; S = South.



# NULL values

NULL means an amount is missing, not that it equals zero. In this variation, test 10, NULL and −4 with amount > 0.

The positive-sales rule keeps 10; NULL and −4 do not pass. This variation does not change the original three-sale result.

| amount | amount > 0 | Filter result |
| --- | --- | --- |
| 10 | True | Keep 10 |
| NULL | Unknown / NULL | Exclude from this filtered result |
| −4 | False | Exclude −4 |
| Surviving sum | 10 | Does not prove missing values were harmless |

# Actions

An action asks Spark to execute a DataFrame plan. show, count, collect and write return different kinds of result.

Choose the destination deliberately: collect gathers all rows at the driver; write stores a distributed dataset. A small final answer can still need a large scan or shuffle.

| Operation | Destination / result | Study distinction |
| --- | --- | --- |
| show(5) | Display up to five rows | A sample does not prove full correctness. |
| count() | One row-count value | Counts rows, not the sum of amount. |
| collect() | All result rows in driver memory | Can exhaust driver memory. |
| write.parquet(path) | Dataset files in storage | Several part-files can form one dataset. |
| limit(5) | New DataFrame plan | Transformation; not a collected Python list. |

# SQL and DataFrame API

SQL and the DataFrame API can express the same Spark query. A temporary view lets SQL refer to the existing sales DataFrame during this session; it does not save a durable table.

This SQL uses the same positive filter, region grouping and amount sum as the DataFrame query. Changing SUM(amount) to COUNT(*) would answer a different question.

```
sales.createOrReplaceTempView("sales")
totals_sql = spark.sql("""
    SELECT region, SUM(amount) AS total
    FROM sales
    WHERE amount > 0
    GROUP BY region
""")
totals_sql.show()
```

### Same intended result

North | 15.0

South | 20.0

# Which expressions run, and what does count return?

Use the three original sales rows. Predict each variable's type and the final answer.

Distinguish a grouped count expression from the action that counts DataFrame rows.

```
a = sales.filter("amount > 0")
b = a.limit(2)
c = a.groupBy("region").count()
d = c.count()
```

## Questions

1. Which lines build plans?
2. Which line requests the result?
3. What are the two rows in c?
4. What is d? Are b's rows fixed?

# Grouped counts

a, b and c are DataFrames; d is the integer 2 returned by the final action.

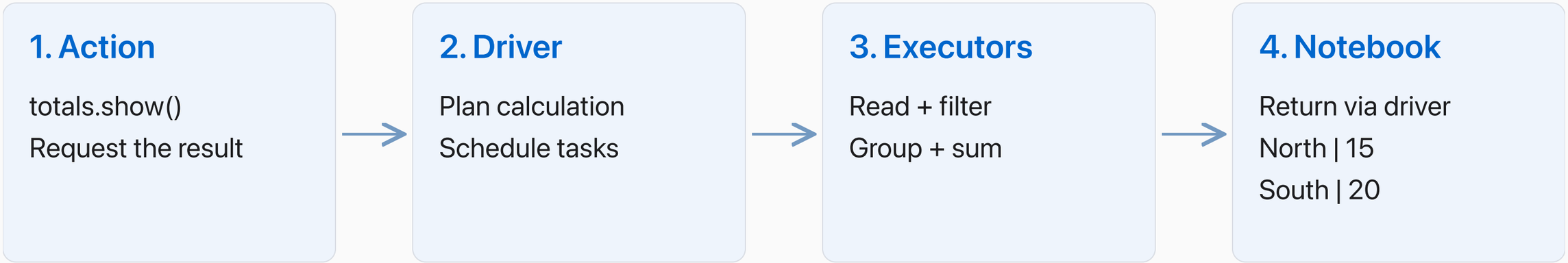
a.groupBy("region").count() builds a lazy aggregation; c.count() is an action. The final action does not use b.

| Expression | Meaning | Result |
| --- | --- | --- |
| a = sales.filter("amount > 0") | Keep positive sales | Plan for 3 rows |
| b = a.limit(2) | Bound the result, without ordering | Plan for up to 2 unspecified rows |
| c = a.groupBy("region").count() | Count sales inside each region | North: 2; South: 1 |
| d = c.count() | Count rows in the grouped result | 2 groups → integer 2 |

# Action: Running the query

An action requests computation of a DataFrame plan. Here totals.show() computes the regional sums for North 10, South 20 and North 5.

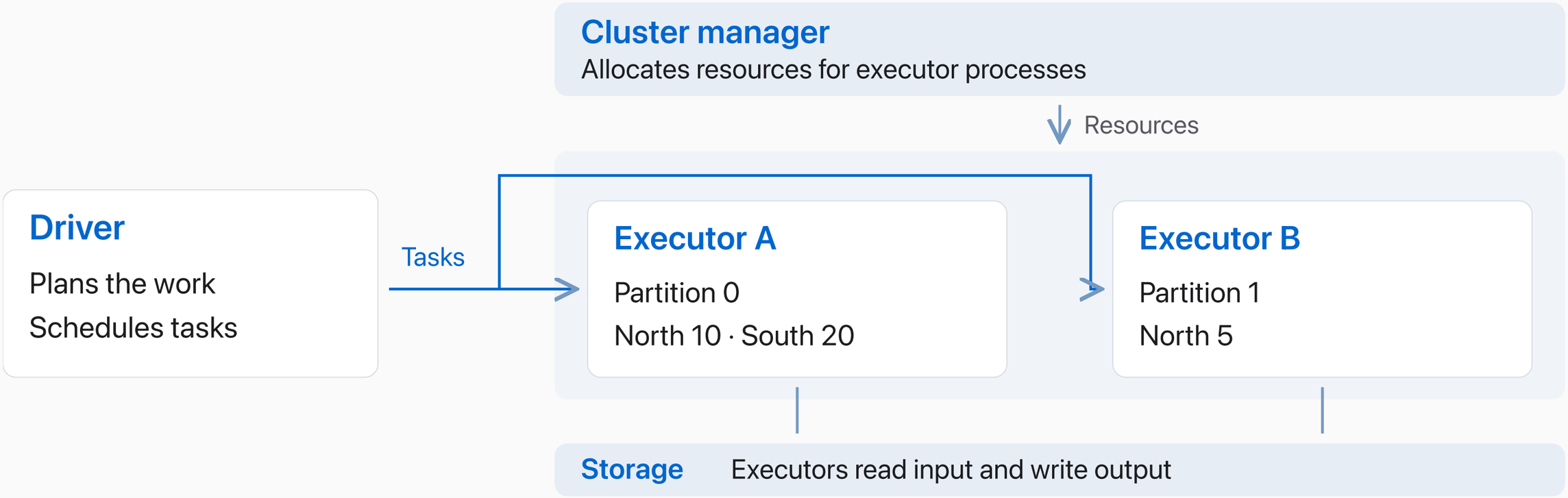
The driver plans and schedules the work; executors process data partitions. The cluster manager supplies executor resources.



# Driver, manager, executors

An action asks the driver to schedule work. The cluster manager allocates resources; executor processes read and process data partitions.

One task processes one partition in a stage. An executor is a process that can run tasks, not a name for an entire machine.

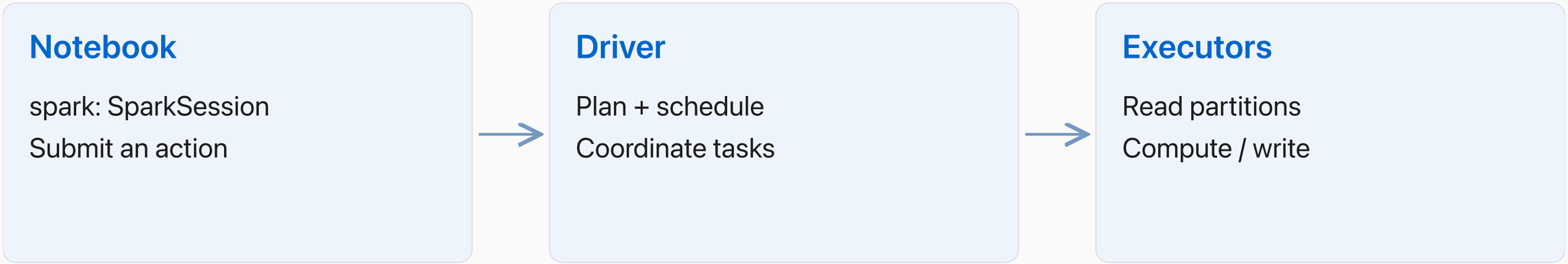


# Notebook to executors

In the notebook, spark is a SparkSession used to express the query. The driver coordinates its execution; executors read storage directly.

The notebook submits work through the Spark application. Sales rows need not pass through the notebook or driver.

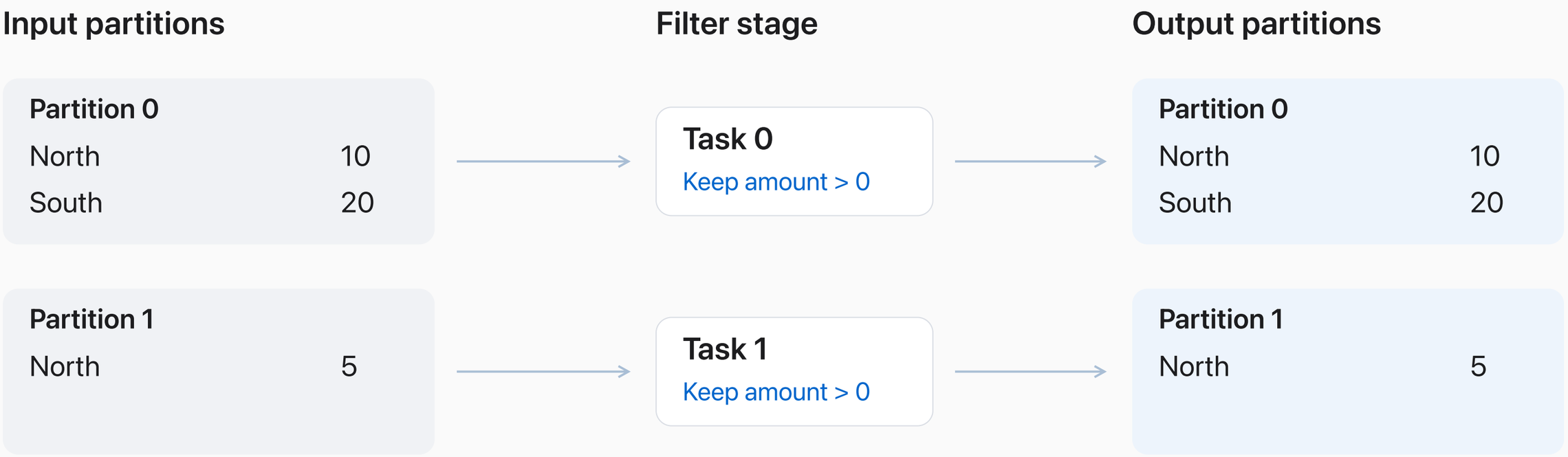
Cluster manager allocates resources; executors access storage directly.



# Partitions

A partition is a slice of distributed data. The three original sales occupy two input partitions, so this filter stage has one task for each partition.

Three rows in two partitions produce two tasks in this stage. Row count, partition count and machine count differ.

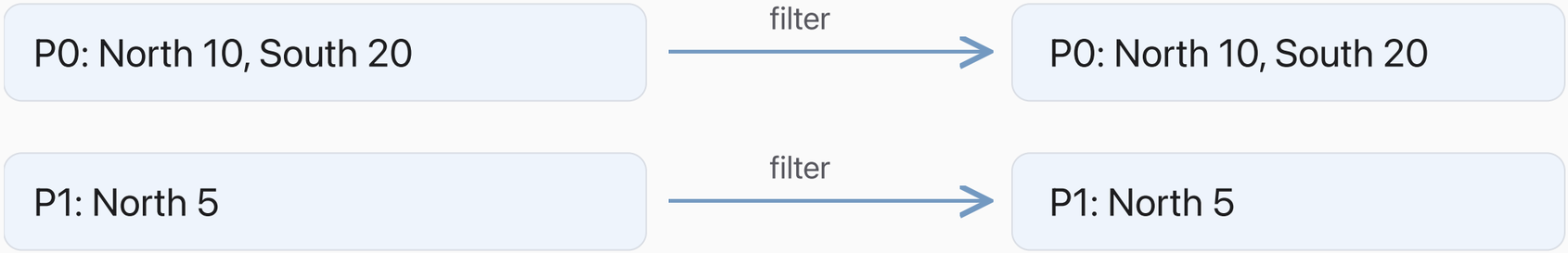


# Narrow transformations

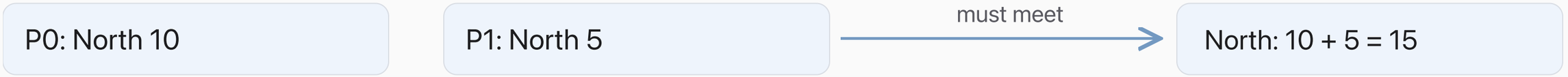
A narrow transformation can process each input partition without moving rows to another partition. Here filter and select work locally before totals are merged.

The positive filter keeps all three sales locally. North 10 and North 5 still need to meet for the regional sum.

## Positive filter · independent lanes




## North total · cross-partition dependency



# Shuffle

A shuffle redistributes intermediate data between partitions so rows with matching keys can be combined. North 10 and North 5 begin apart.

The North contributions can meet after redistribution and sum to 15. Shuffle moves data and can add network or disk cost.

### Local partial sums

| Partition | Region | Sum |
|---|---|---|
| 0 | North | 10 |
| 0 | South | 20 |
| 1 | North | 5 |

### Group matching regions

**North**

10 + 5

**South**

20

### Merge

15

20

# Execution hierarchy

An application contains work requested by actions. Jobs consist of stages; each stage runs tasks over partitions.

A shuffle can separate stages. Count the level being asked about: rows, partitions, tasks, stages and machines are different quantities.

## Application = driver + executors

Action requests computation → one or more jobs

### Stage A · before shuffle

Task 0 → partition 0
Task 1 → partition 1

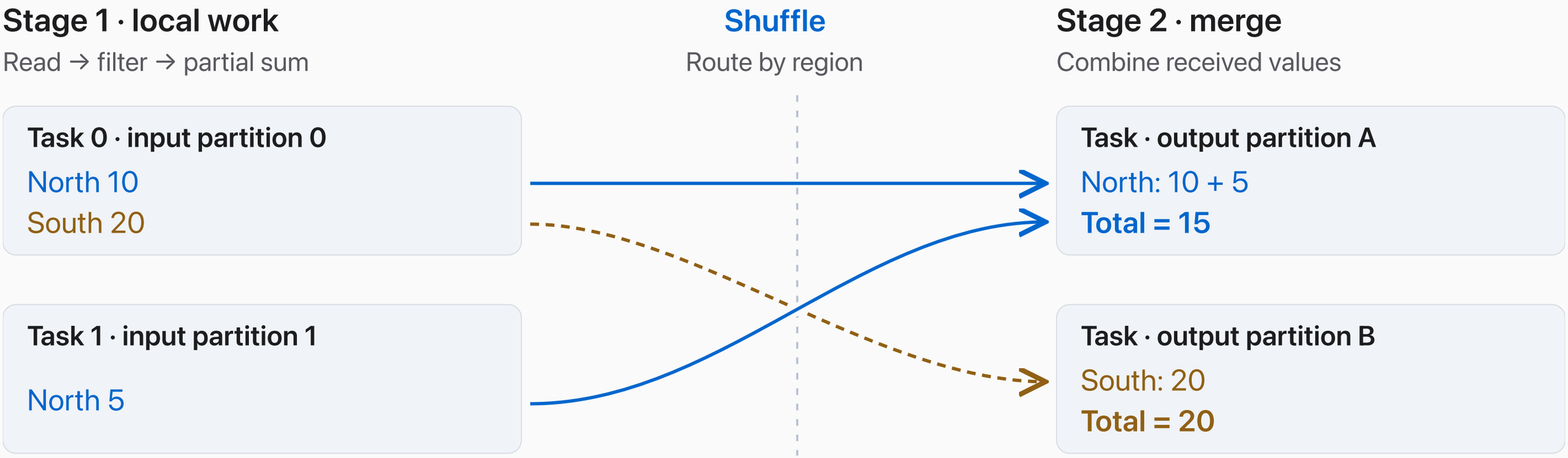
### Stage B · after shuffle

Task 0 → output partition 0
Task 1 → output partition 1

# Stage boundaries

A stage groups tasks that can run without a shuffle between them. The next stage needs the preceding stage's shuffle output before it can merge the results.

Here two input tasks run independently. The shuffle brings North 10 and North 5 together; the next stage merges them into North 15.

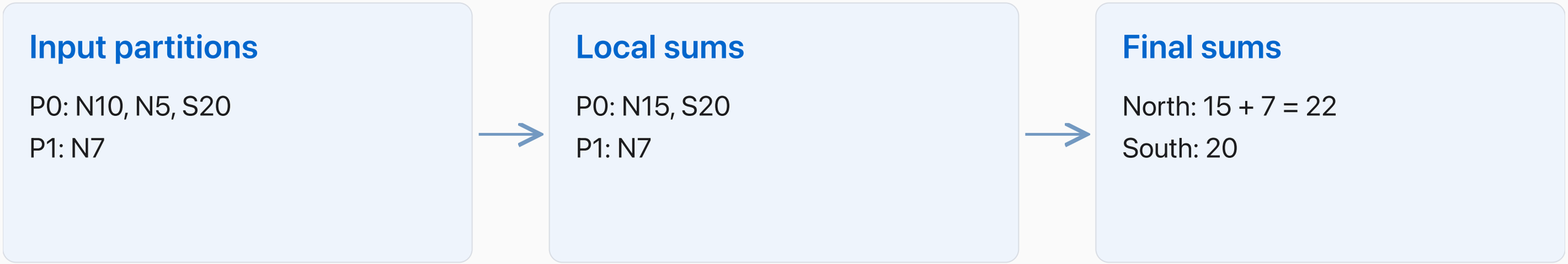


# Local combining

Local combining makes partial totals inside each input partition before data moves. In this four-sale variation, add North 7 to the original sales.

P0 can send North 15 and South 20 instead of three raw sales. Combined with P1's North 7, North becomes 22.

Four raw records → three partial records. N = North; S = South.

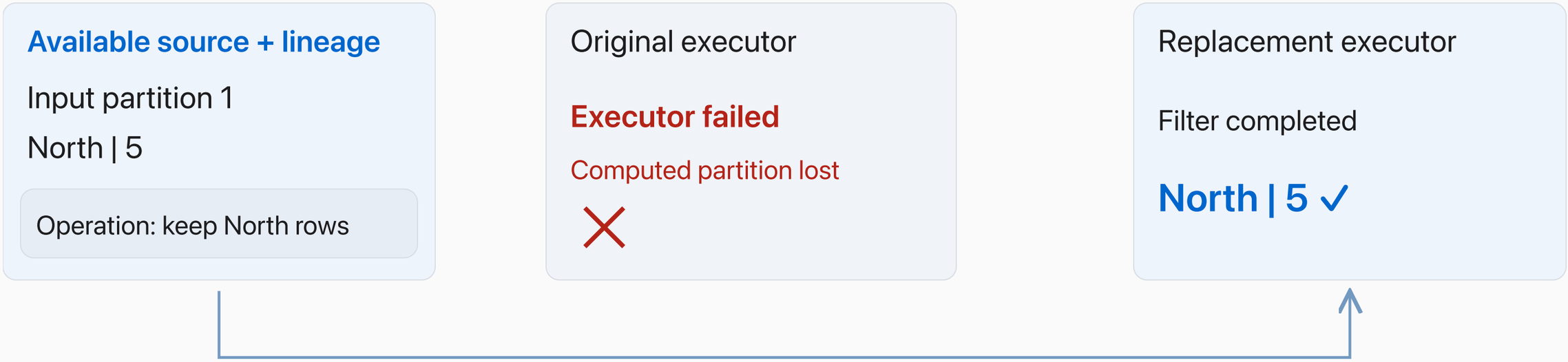


# Lineage

Lineage records how Spark derived a computed partition. If it is lost, Spark can rerun the steps using available input and resources.

Recomputation cannot replace a missing source or correct bad logic. A cached partition is not a durable backup.

**Lost computed partition → recompute from source → restored result**



# Physical plans

A physical plan describes the operations Spark chooses to produce the requested result. For positive sales summed by region, it may scan, filter, combine locally, exchange and finish the sum.

explain("formatted") reveals the plan selected before execution. Runtime evidence in the Spark UI can show costs and adaptive changes.

```
Conceptual plan · read bottom to top

Final aggregate: sum by region
  Exchange: partition by region
    Partial aggregate: local sums
      Filter: amount > 0
        Scan: region, amount
```

## What to identify

Scan → source and required columns

Filter → which rows qualify

Exchange → redistribution boundary

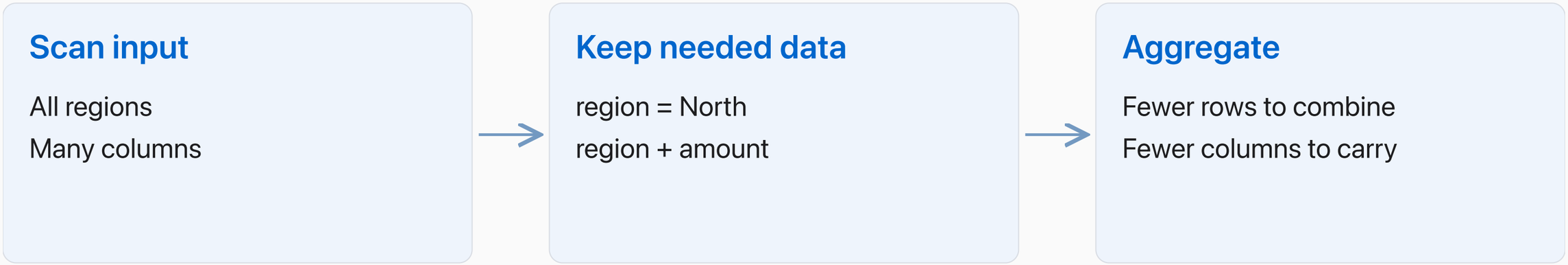
Aggregate → local and final work

# Early filtering

If the question changes to North-only sales, filtering South before grouping can reduce downstream work.

Early row and column reduction can reduce later work; source-level read savings depend on support.

Preserve required columns and query meaning when moving or adding reductions.



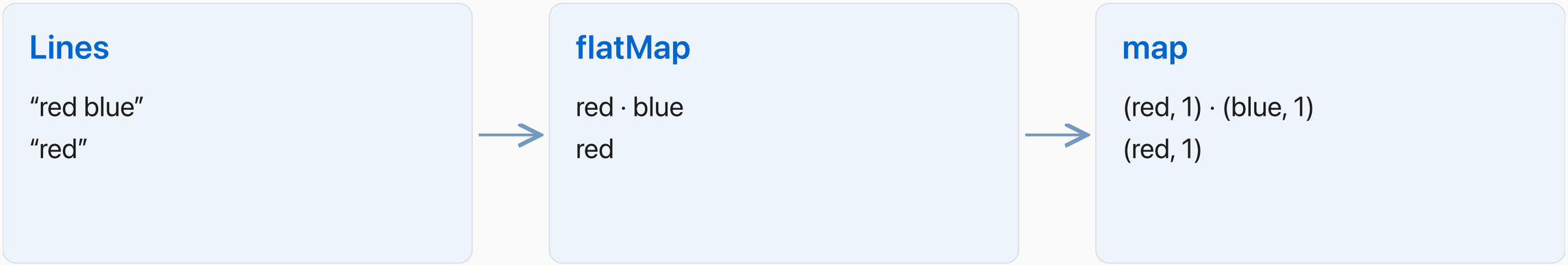
# Keys and joins

Keys combine values in an aggregation and match rows in a join.

# RDD word records

In an RDD, flatMap splits each input line into individual word records. map then pairs each word with 1.

Each pair represents one occurrence; repeated words will be combined by key.



# Word count with two APIs

RDD reduceByKey adds values for equal word keys. collect() requests the small RDD result and brings it to the driver.

Lab 07 uses DataFrame split, explode and groupBy("word").count() to obtain the same grouped counts.

```
lines = sc.parallelize(["red blue", "red"])
words = lines.flatMap(lambda s: s.split())
pairs = words.map(lambda w: (w, 1))
counts = pairs.reduceByKey(lambda a, b: a + b)
counts.collect()
```

## Lab 07 DataFrame route

split lines → explode words

groupBy("word").count()

red | 2; blue | 1

show() requests the result

# Duplicate join keys

A join matches rows by a key. A sale with two matching lookup rows appears twice; here North maps to both A and B.

Define key uniqueness and join type before trusting totals. A left join preserves unmatched left rows but still duplicates repeated matches.

| Sale | Matches in lookup | Inner-join rows |
| --- | --- | --- |
| North \| 10 | A and B | North \| 10 \| A; North \| 10 \| B |
| North \| 5 | A and B | North \| 5 \| A; North \| 5 \| B |
| South \| 20 | No matching lookup row | No output row |
| Before: 3 rows; sum 35 | After: 4 rows; sum 30 | Duplication and loss happen together. |

# Left joins

A left join keeps each sale even when no lookup row matches. With this duplicate North lookup, matching sales still appear twice.

Left joins preserve unmatched left rows, but repeated matches still multiply them.

| Sale | Left-join result | Amount contribution |
| --- | --- | --- |
| North 10 | North 10 A; North 10 B | 20 |
| North 5 | North 5 A; North 5 B | 10 |
| South 20 | South 20 NULL | 20 |
| Total | 5 output rows | 50, compared with original 35 |

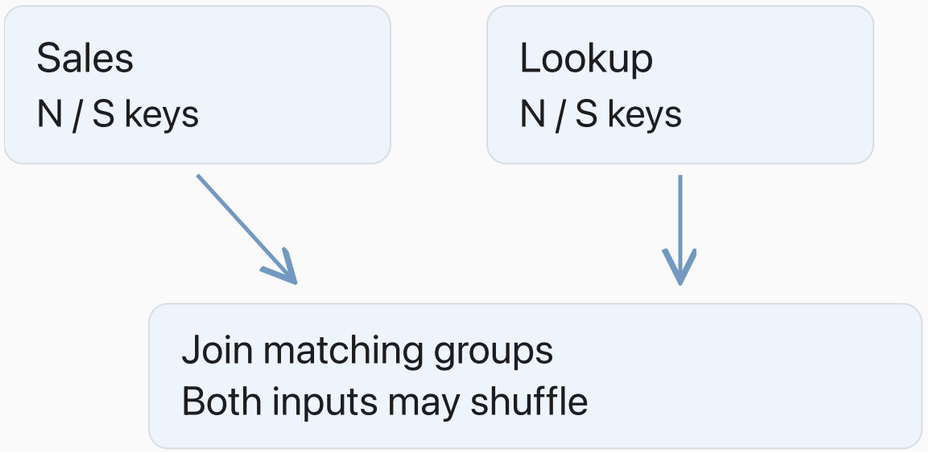
# Join strategies

A shuffle join moves rows by key; a broadcast join copies a suitable small lookup to executors. Both must preserve the checked join result.

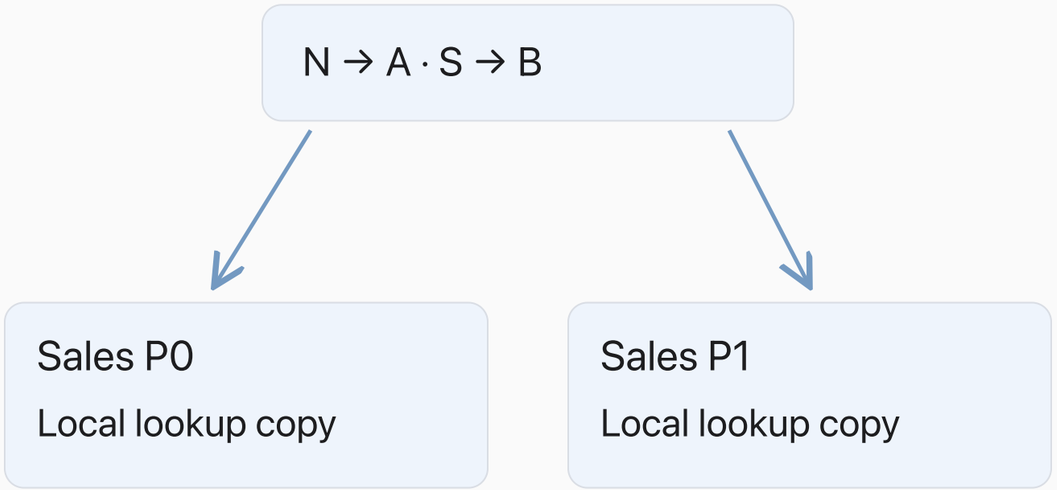
Broadcast may avoid shuffling the large sales side, but lookup bytes, executor memory and join support matter. Confirm the chosen plan.

The join result must stay the same. Broadcast requires a lookup small enough for each executor.

## Shuffle join · matching keys move together




## Broadcast join · copy a small lookup



# Performance diagnosis

Use task evidence to identify idle capacity, uneven work, repeated computation and spill.

# Spark UI evidence

The Spark UI is the application’s runtime view of jobs, stages and tasks. It shows whether time is lost to idle capacity, skew or spill.

Use task duration, input size, shuffle bytes and spill together to identify the likely bottleneck before changing settings.

| Observed pattern | Inspect next | Possible explanation |
| --- | --- | --- |
| One large, slow task | Key distribution + input size | Skew or uneven work |
| Many tasks spill | Task size + executor memory | Execution memory pressure |
| Few tasks; idle slots | Partitions + available capacity | Insufficient parallel work |
| Driver fails during collect | Result size + driver memory | Gathered result is too large |

# Task slots

A task slot means capacity to run one task at a time in this example. Four equal tasks and two available slots require two waves.

Two slots run two tasks together, then the next two. Adding partitions creates work units; adding capacity helps only if enough work can run.

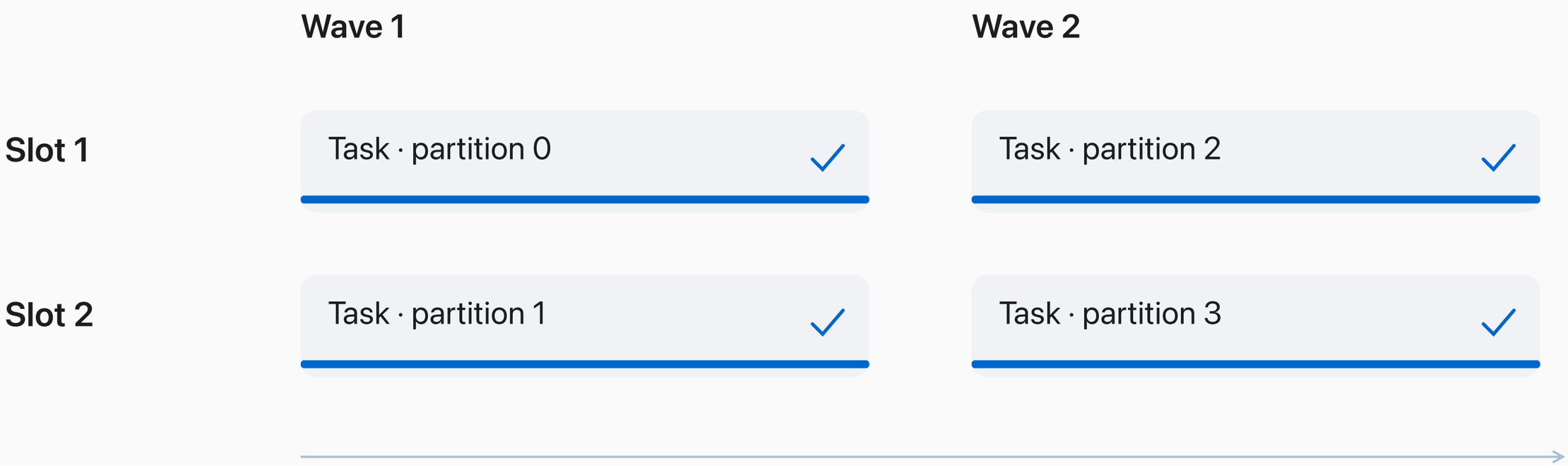


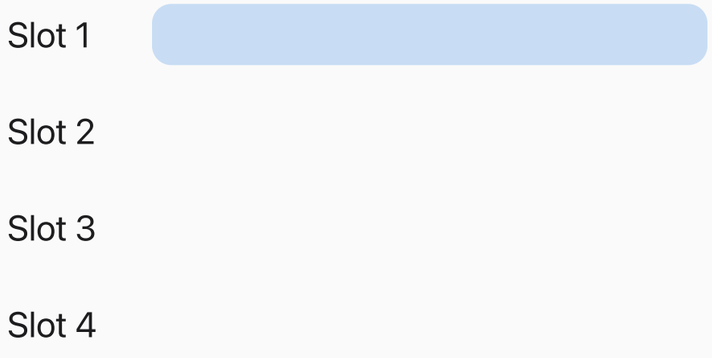

Illustration: equal task durations.

# Partition sizing

Partitions determine work units. Available slots, data size and task costs determine useful parallelism.

Too few partitions leave slots idle; too many tiny partitions add scheduling and file overhead.

## Too few: 1 task




Idle slots; large task

Limited parallelism

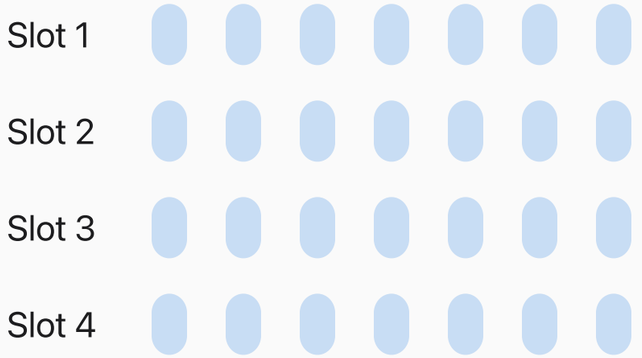
## Enough useful work: 4 tasks




Useful task size

Manageable overhead

## Too many: many tiny tasks




Gaps = launch overhead

Small tasks add up

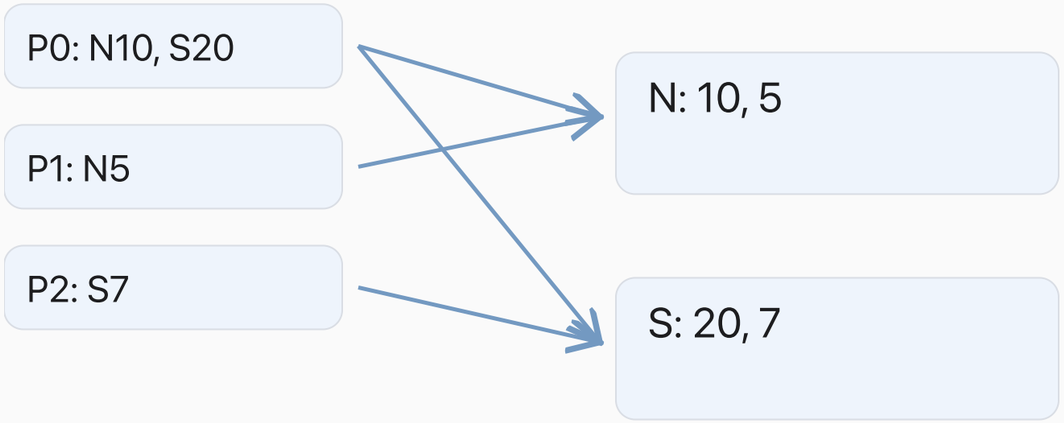
# repartition and coalesce

repartition redistributes rows and can increase or decrease partition count. It may improve balance, but it costs a shuffle.

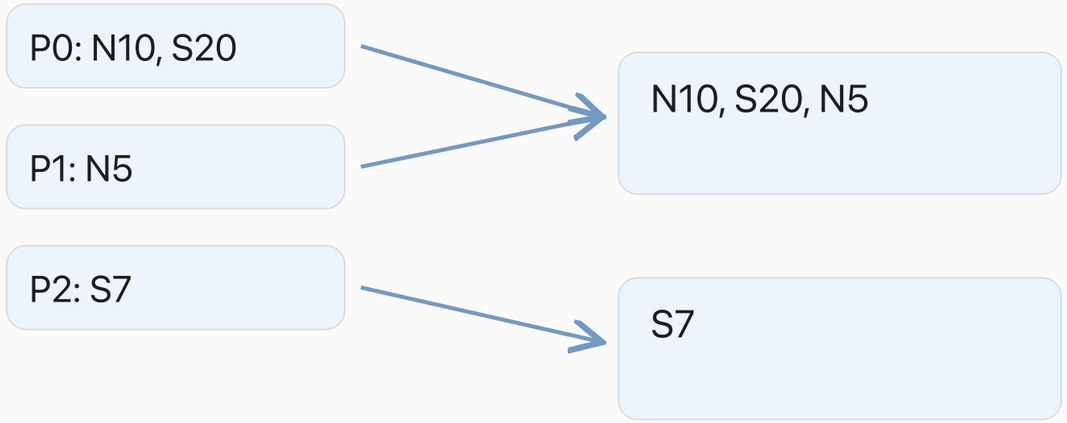
coalesce can reduce partitions without a full rebalance. Fewer tasks can reduce overhead, but a single output partition can serialize a large write.

The extra South 7 row illustrates routing only; the original sales answer remains North 15 and South 20.

## repartition(2, "region") · shuffle




## coalesce(2) · merge partitions



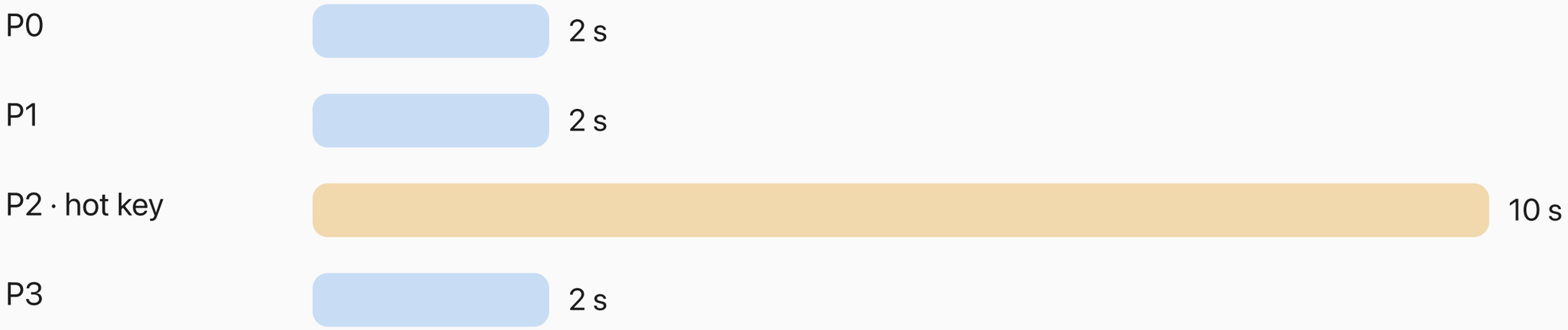
# Skew

Skew means one task receives much more data or work than its peers. A frequent key can dominate one destination partition.

The stage waits for its slowest task. More workers or hash partitions may leave that hot key together; inspect key sizes and task times.

Stage completion waits for the 10-second task. More workers alone may not split its key.

Illustrative task durations in one stage

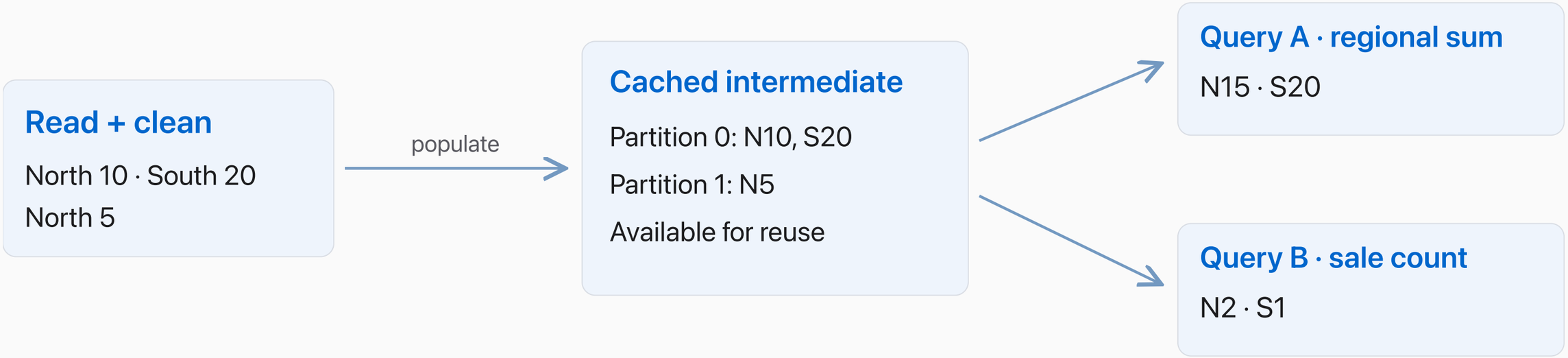


# Caching

Caching asks Spark to retain computed partitions for later actions. It is useful when the cost of repeated work exceeds cache population and storage costs.

Populate only when an action runs, reuse when valuable, and release with unpersist(). Cached data is not durable output.

Register cache → materialise through an action → reuse → unpersist when finished.



# Cache reuse in code

Here clean is reused by a row count and a regional aggregation.

Benchmark cold work and reused work separately; report whether cache population is included.

For a one-off query this extra materialisation may be wasteful.

```
clean = sales.filter("amount > 0").cache()
clean.count()  # populate for this example

clean.groupBy("region").sum("amount").show()
clean.groupBy("region").count().show()

clean.unpersist()
```

## What each step means

cache(): register persistence

count(): request all rows for this example

Two queries reuse available cached data

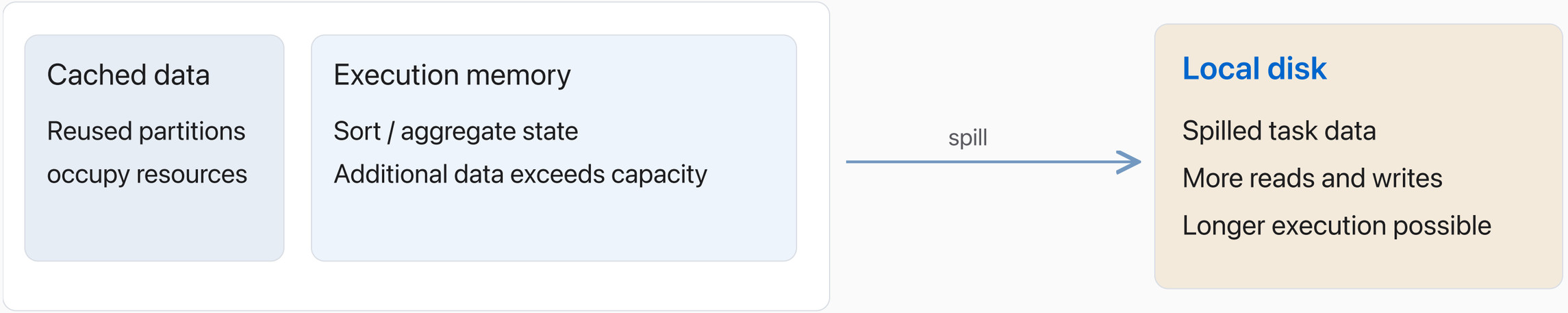
unpersist(): release cached blocks

# Spill

Spill occurs when execution work such as sorting or aggregation cannot remain entirely in executor memory and uses local disk. Cached partitions can compete for resources.

Use spill metrics and task sizes to test whether smaller tasks or releasing cache helps. Driver memory pressure from collect() is a separate problem.

## Executor memory · finite capacity



# Fair benchmarks

A fair performance comparison holds the query meaning, input and output checks fixed.

State the change, verify equal results, and explain the measured effect. Do not infer speed from shorter code.

| Keep / record | Why it matters |
| --- | --- |
| Same input, query meaning and output checks | Otherwise you may be timing different work. |
| An action that completes the intended computation | Timing lazy plan construction misses execution. |
| Same cluster resources; record relevant settings | Capacity differences change elapsed time. |
| Cold versus warm cache; repeated runs | Reuse and run-to-run variation affect results. |
| Elapsed time, shuffle, task spread and spill | Evidence connects the bottleneck to the change. |

# Benchmark evidence

Illustrative repeated writes use the same input, result, cluster and cold-cache condition.

Compare repeated completed runs and connect the change to a measured reduction in work.

| Evidence | Baseline | After reducing unused data |
| --- | --- | --- |
| Completed write times | 30, 32, 31 seconds | 22, 24, 23 seconds |
| Median | 31 seconds | 23 seconds |
| Shuffle bytes | 120 MB | 60 MB |
| Read-back checks | Schema, count, totals match | Same checks match |

# Storage and output

Skip unnecessary input, then save and verify the regional totals.

# Parquet

CSV stores row-oriented text; Parquet stores typed values in column chunks with metadata.

The query needs region and amount but not date, so Parquet can skip the date chunks. Actual I/O and time savings depend on the file layout and workload.

## CSV · row-oriented text

```
region,amount,date
North,10,2026-10-01
South,20,2026-10-01
North,5,2026-10-02
```

## Parquet · typed column chunks

| region | amount | date |
|---|---|---|
| North | 10 | Oct 01 |
| South | 20 | Oct 01 |
| North | 5 | Oct 02 |

# Pruning and filtering

Directory pruning skips directories; supported statistics can skip row groups; row filtering tests the surviving records.

A pushed predicate does not guarantee that all nonmatching bytes are skipped. The format, metadata and reader determine the savings.

## Query: date = '2026-10-01' AND amount > 15

### Directory pruning

date=Oct 02

Skip this directory.

date=Oct 01

Inspect its files.

### Row-group statistics

RG1 amount: min 5, max 10

Skip: cannot exceed 15.

RG2 amount: min 10, max 20

Read: may contain matches.

### Row filtering

RG2 values: 10, 20

Test amount > 15.

Keep: 20

Reject: 10

# What can be skipped?

Query: date = Oct 01 AND amount > 15. Dates are directory partitions.

Skip only data that cannot match; test rows in any surviving row group.

| Stored data | Evidence | Decision |
| --- | --- | --- |
| Oct 02 directory | Wrong partition date | Skip directory |
| Oct 01, group A: 5, 10 | Maximum 10 ≤ 15 | Skip row group |
| Oct 01, group B: 10, 20 | Maximum 20 > 15 | Read and test both rows |
| Final matching row | 20 > 15 | Keep amount 20 |

# Saved datasets

A write creates a dataset at a path, often with several part-files. Read that path back to verify the answer actually stored.

Read the whole dataset directory, then check schema, two regional rows and total 35. Save to independent storage before cluster shutdown.

A relative path uses the environment’s filesystem. Choose independent durable storage before cluster shutdown.

```
out = "results/region_totals"  # new path
totals.write.mode("errorifexists").parquet(out)

check = spark.read.parquet(out)
check.printSchema()
check.show()
check.count()
```

## Verify the dataset

Schema: region and numeric total

2 rows: North 15; South 20

Grand total: 35

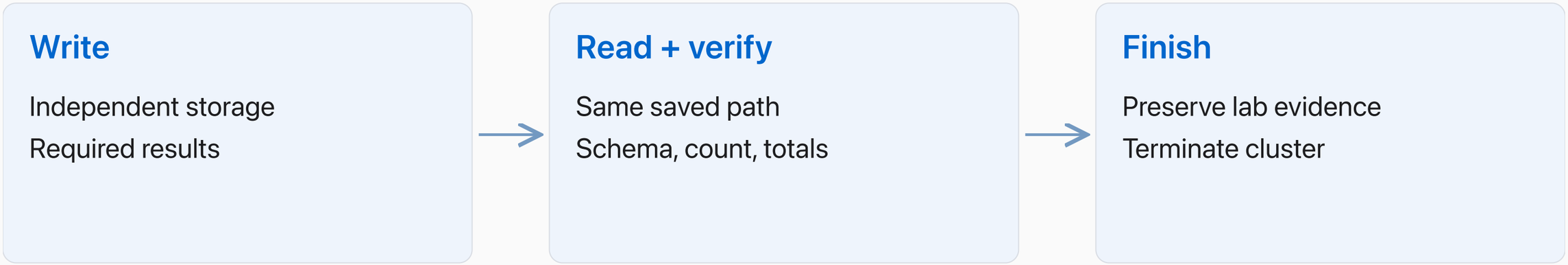
Several part-files may form one dataset.

# Durable output

Durable output is stored independently of a cluster. Cluster-local files can disappear when the cluster is terminated.

Write to independent durable storage, read that copy back, then end the cluster.

Example independent destination: an approved S3 location outside cluster-local storage.



# Spark in Labs 07 and 08

Lab 07 counts words using DataFrame split, explode and groupBy. Lab 08 reads typed CSV, compares SQL with DataFrame code and saves Parquet.

The same principles recur in both labs: define data, request work, check the result, preserve the saved copy and terminate the cluster.

The notebook submits the query; Spark executors do distributed work.

### Lab 07

DataFrame word count

split → explode → groupBy

### Lab 08

Typed CSV + SQL

Validate → save Parquet

### Before shutdown

Save results + evidence

Read back → terminate

# From query rules to a reliable Spark result

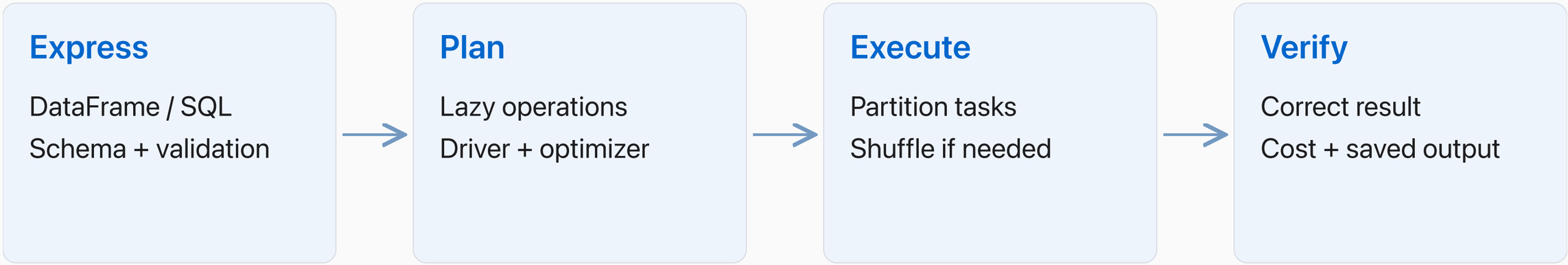
Connect query rules, execution, performance and verified output.

# Sales to saved totals

For regional sales, query rules determine the answer; partition tasks and any shuffle produce it; a read-back check confirms it.

The data, query, execution and saved checks must agree on North 15 and South 20.

Storage provides data; the cluster manager provides resources; executors do the partition work.



# Execution roles

The driver coordinates; executors run tasks over partitions; slots limit how many tasks run at once.

Four partitions and two slots mean four tasks in two waves under the stated equal-duration assumptions.

| Term | Meaning | Do not confuse it with... |
| --- | --- | --- |
| Driver / cluster manager | Plan and schedule / allocate resources | Executors that compute partition tasks |
| Application / job | Application processes / action-triggered work | One line of Python code |
| Stage / task | Parallel work phase / work on one partition | One task per row |
| Partition / slot | Data work unit / concurrent task capacity | A fixed machine or an HDFS block |

# Query correctness

Types, accepted rows, grouping keys and aggregation must match the sales question.

Trace actual rows and values. A successful run does not prove that the chosen filter, join or aggregation answers the question.

| Concept | Rule to retain |
| --- | --- |
| RDD / DataFrame / SQL | Records / schema-based table / query language for the same engine |
| Transformation / action | Build a new plan / request a result or output |
| show / count / collect / write | Sample / row count / all driver rows / stored dataset |
| groupBy(...).count() / df.count() | Lazy grouped aggregation / action returning row count |
| Join type + key multiplicity | Unmatched rows can disappear; repeated matches multiply rows |

# Cross-partition totals

Original input: P0 has North 10 and South 20; P1 has North 5.

A shuffle can separate stages and add network/disk cost. Explain shows the plan; runtime evidence shows what it cost.

Same-key contributions meet; multiple keys may share a shuffle partition.



# Performance choices

Idle slots, uneven tasks, repeated work and spill point to different possible changes.

Reduce unnecessary data, diagnose the bottleneck, change one thing, and compare correct completed results.

| Observed evidence | Candidate response | Trade-off / check |
| --- | --- | --- |
| Idle slots; few large tasks | More useful partitions | Extra shuffle or scheduling cost |
| Many tiny tasks or files | Suitable coalesce / output layout | Too few tasks restrict parallelism |
| One very slow, large task | Investigate skew and hot keys | Extra workers alone may not help |
| Repeated expensive intermediate | Cache, then unpersist | Population cost and memory use |
| Large join + small lookup | Consider broadcast | Lookup must fit; verify chosen plan |

# Reuse and durability

Pruning skips input; cache reuses computed data; lineage can rebuild it; durable storage preserves output.

A cached DataFrame is not a backup. Before shutdown, verify the independent saved copy and required evidence.

| Mechanism | What it provides | Important limit |
| --- | --- | --- |
| Parquet + column pruning | Read required column chunks | Benefit depends on query and source |
| Directory / row-group pruning | Skip data that cannot match | Still test surviving rows |
| Cache / lineage | Reuse / recompute partitions | Needs resources / available inputs |
| Durable write + read-back | Preserved, checked output | Cluster-local data may disappear |
| Schema + count + totals | Evidence of saved result meaning | A sample alone is incomplete |

# Predict the saved regional totals

P0: North 10, South 20. P1: North 5, South −4. Keep amount > 0, then sum by region.

Name the rule and explain its consequence. Show enough arithmetic to justify the answer.

Assume out is a new path; the filter deliberately excludes the return. Two input partitions do not fix output-file count.

```
q = (sales.filter("amount > 0")
      .groupBy("region").sum("amount"))
q.write.mode("errorifexists").parquet(out)
```

## Explain four things

1. What result is written?
2. Which line requests execution?
3. Why might a shuffle be needed?
4. What must be checked before shutdown?

# Saved totals

The positive-sales rule keeps 10, 20 and 5 and excludes the South return of −4.

The write triggers execution; North contributions meet; a read-back check confirms two rows and total 35.

| Question | Reasoned answer |
| --- | --- |
| Written result | North = 15; South = 20; 2 rows; grand total = 35. |
| Execution trigger | write requests output; filter and aggregation build the plan. |
| Movement | North contributions cross input partitions and must meet. |
| Before shutdown | Read saved data; check schema, count and totals; preserve an independent copy. |