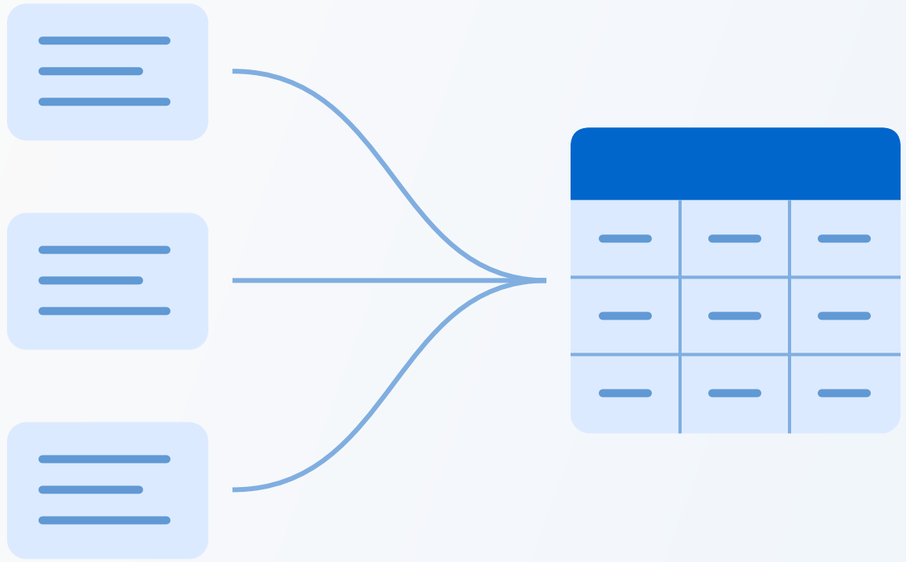
Hive

From stored files to checked SQL results



CONTENTS

Topic overview

Follow movie-rating records from stored files to SQL results whose meaning and row counts can be checked.

Topic	Content
1. Understand Hive	SQL, the metastore, execution and storage
2. Register the data	Files, schemas, table locations and lifecycle
3. Write the query	Filters, counts, averages, grouping and ranking
4. Join the tables	Missing matches, duplicate keys and row counts
5. Select the input	Nested data, partitions, pruning and plan evidence
6. Apply and review	Lab 10, preserving results and a worked final case

Movie ratings

A rating row records one user's score for one movie at a time. We want the most-rated movies and their average scores.

We use six invented ratings to trace the queries; Lab 10 uses the larger MovieLens dataset.

User	Movie	Rating
1	10	5
2	10	3
3	20	4
4	20	5
5	20	3
6	30	5

File-to-result workflow

Hive exposes stored data through named tables and HiveQL, its SQL query language. A table definition tells the engine how to interpret the files.

The result depends on both correct metadata and correct query logic.

Stored file

ratings.tsv

movie	rating
10	5
10	3
20	4
20	5
20	3
30	5

Table definition

ratings

movie_id: INT

rating: INT

Tab-separated fields

Location in HDFS

Query

Group by movie

Count ratings

Average rating

Sort: count descending, then movie ID

Take the first 2

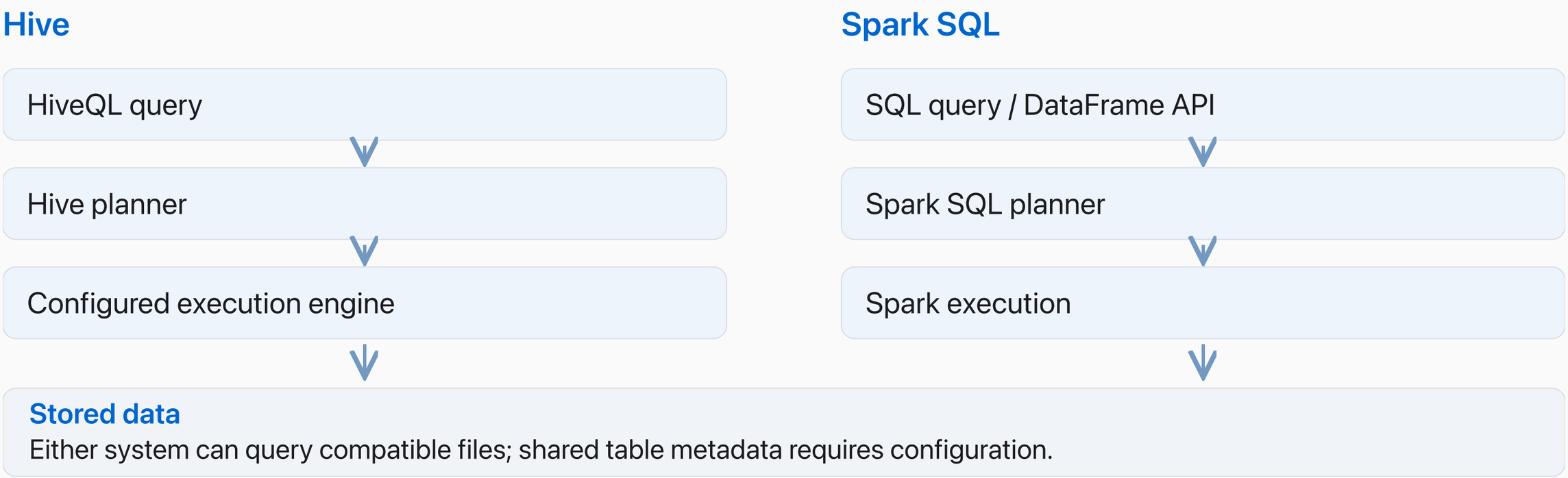
Result

Movie	Count	Avg
20	3	4.0
10	2	4.0

Hive and Spark SQL

Hive and Spark SQL can both express analytical queries over distributed data. Their execution and catalog configuration may differ.

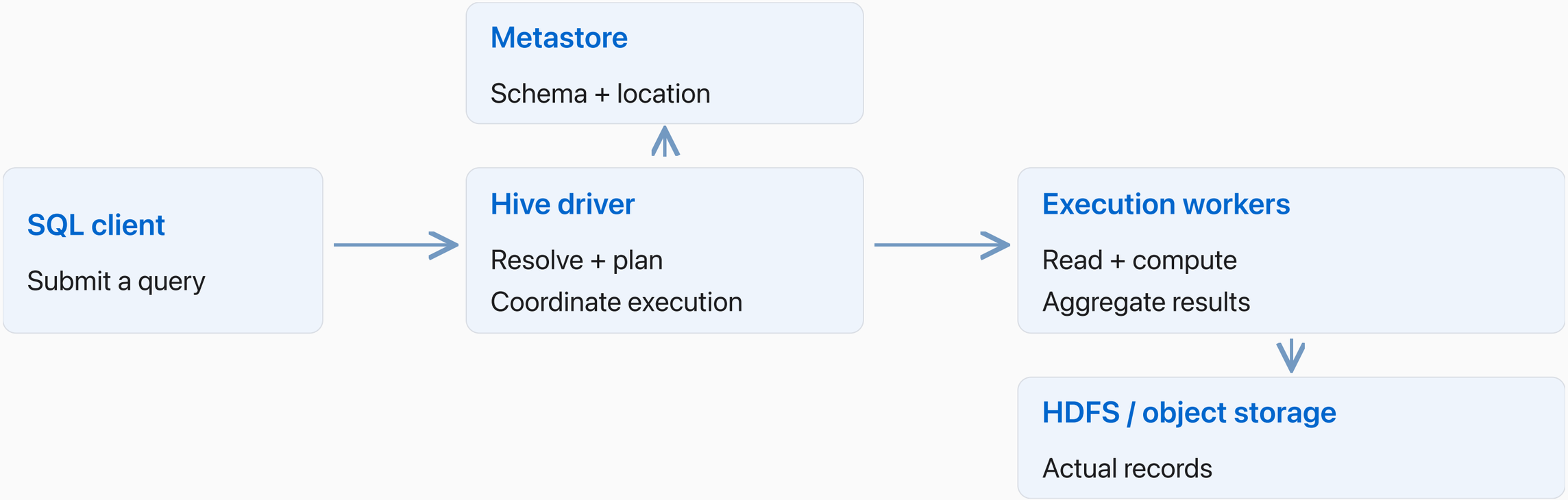
Choose by the environment and workflow; a SQL expression does not imply that Hive runs on Spark.



Hive architecture

The driver coordinates query compilation and execution. The metastore supplies table metadata; execution workers read and process the actual data.

The metastore describes the files; it does not contain all their rating rows.



Metastore

The metastore is a catalog of table definitions, schemas, storage locations and partition metadata.

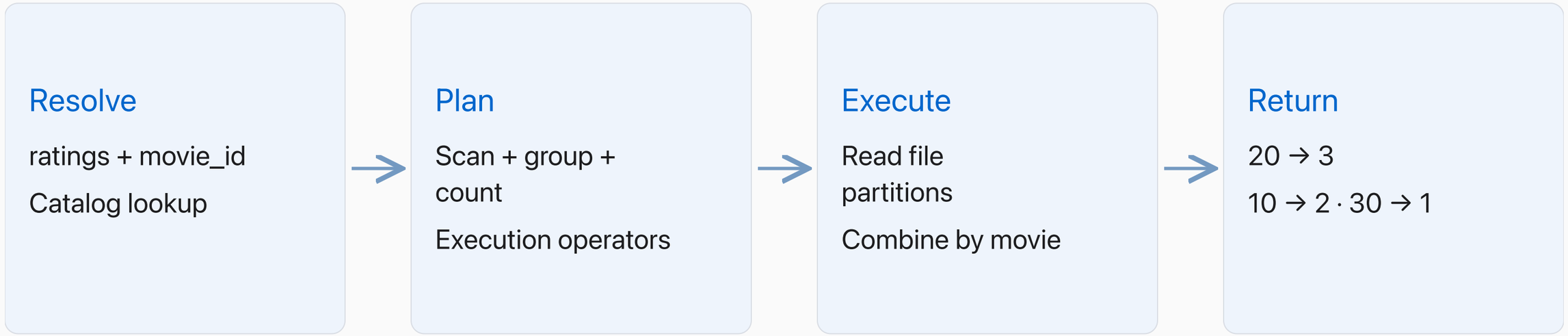
A table name is useful only when its metadata agrees with the stored records.

Metadata field	Ratings example	Purpose
Table name	movielens.ratings	Resolve the SQL name
Columns	user_id, movie_id, rating, rating_ts	Interpret fields
Format	Tab-delimited text	Decode each record
Location	/user/hive/data/movielens/	Find the dataset
Partitions	None in this first table	Describe optional subdivisions

Query execution

Hive resolves the query using metadata, builds an execution plan and asks the configured engine to run it.

Workers read the relevant files, compute partial results and combine them as the plan requires.



Execution engine

Hive’s execution engine depends on its release and configuration. Amazon EMR has used Tez as the Hive default since release 5.x.

Read the actual setting rather than assuming that the presence of Spark determines how Hive executes.

```
SET hive.execution.engine;
```

Record with the result

Hive version

EMR release

Reported engine setting

Schema-on-read

Schema-on-read applies a declared structure when existing data is read. Registering a table does not clean or rewrite those records.

Correct column names alone cannot fix a wrong delimiter, invalid value or wrong file location.

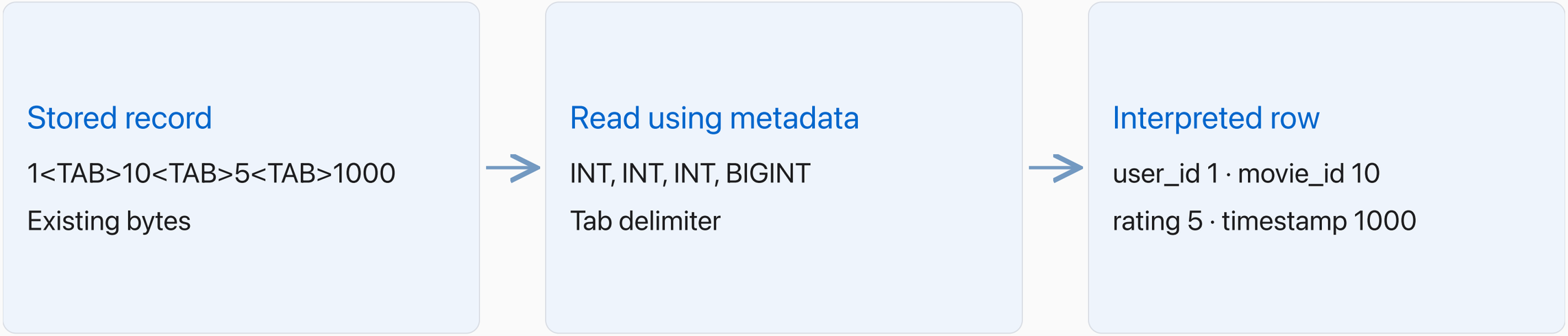


Table registration

A queryable table needs compatible files, a matching schema and format, and a catalog entry pointing to the right location.

Validate the interpreted rows before trusting an aggregate.

Stage

Place dataset in HDFS

Check files

Declare

Columns + delimiter

Table location

Inspect

Metadata + sample rows

Counts + ranges

Query

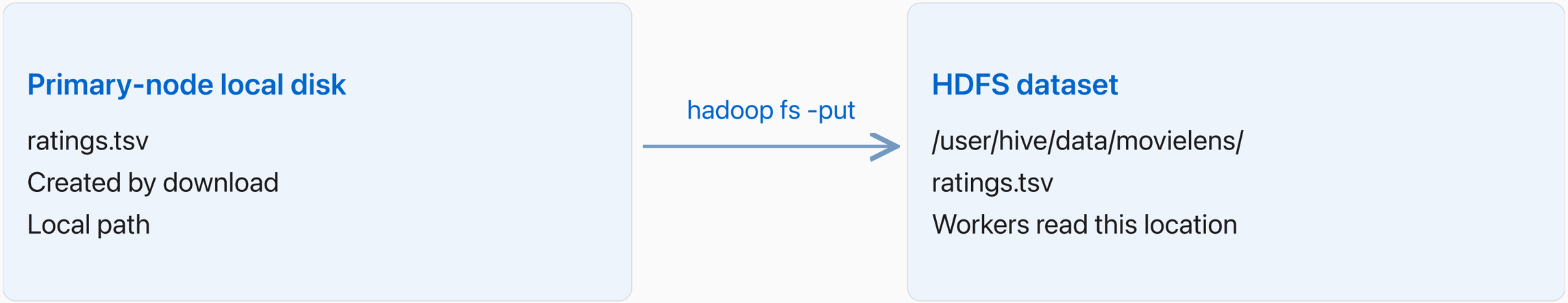
Verified interpretation

Meaningful aggregation

Local files and HDFS

The primary node’s local filesystem and HDFS are different storage locations. Downloading a file locally does not place it in HDFS.

The table location must point to the distributed dataset that workers can read.



Staging ratings

These commands create the HDFS directory, upload ratings.tsv and list the destination. The local file must already exist.

Check for an existing upload before repeating the operation so the dataset is not duplicated.

```
hadoop fs -mkdir -p /user/hive/data/movielens
hadoop fs -put ratings.tsv /user/hive/data/movielens/
hadoop fs -ls /user/hive/data/movielens/
```

Expected destination

HDFS directory: movielens

Contains: ratings.tsv

Tab-separated records; no header

Database and table names

A Hive database provides a namespace for tables. USE selects the default namespace for unqualified names. movielens.ratings identifies the table explicitly, even if the current database changes.

```
CREATE DATABASE IF NOT EXISTS movielens;
USE movielens;
SHOW TABLES;
```

Name resolution

ratings → current database

movielens.ratings → explicit database

Database name ≠ one data file

External table definition

CREATE EXTERNAL TABLE registers the ratings fields, tab delimiter and storage directory. The four fields follow the file’s column order.

BIGINT provides room for the timestamp value; the table reads compatible records from its directory.

```
CREATE EXTERNAL TABLE ratings (
  user_id INT, movie_id INT,
  rating INT, rating_ts BIGINT
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE
LOCATION '/user/hive/data/movielens/';
```

Four fields

user → movie → rating → time

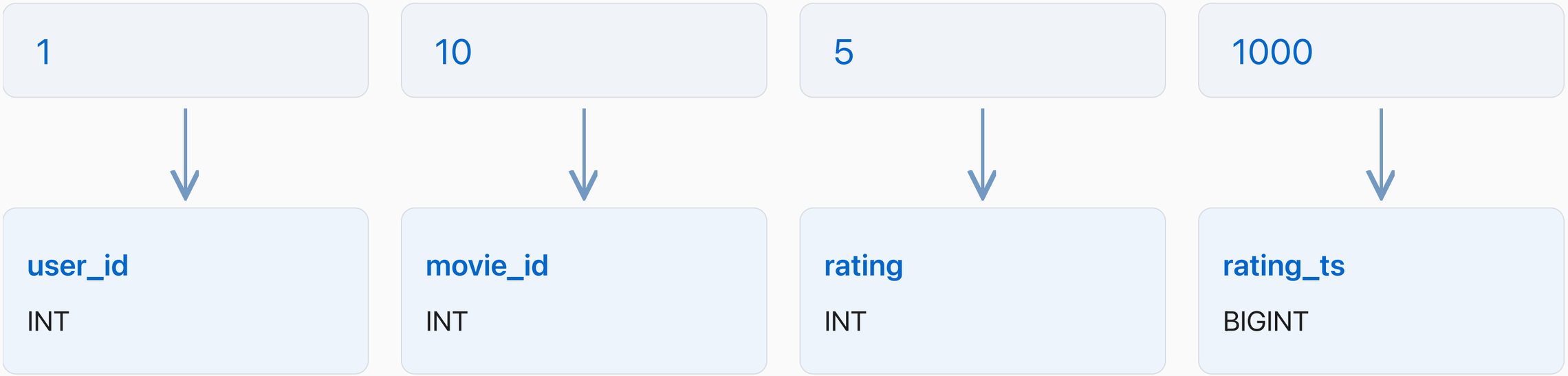
No header row

Tab-separated text

Column order and types

For these delimited records, the first field becomes user_id, the second movie_id, the third rating and the fourth rating_ts.

A swapped declaration can produce plausible numbers with the wrong meaning.



File format

The reader needs a format and delimiter that match the bytes. Tab-separated values are not comma-separated values.

A mismatched delimiter may turn several intended fields into one field and leave the remaining fields invalid or null.

Actual TSV record

1<TAB>10<TAB>5<TAB>1000

Four tab-separated fields

Wrong comma reader

No comma boundaries found

Fields are misinterpreted

Correct tab reader

1 / 10 / 5 / 1000

Four declared values

Dataset location

A table location identifies a dataset directory that may contain multiple compatible files.

Keep unrelated files and duplicate uploads out of that directory; the catalog does not know which copies you intended to count.

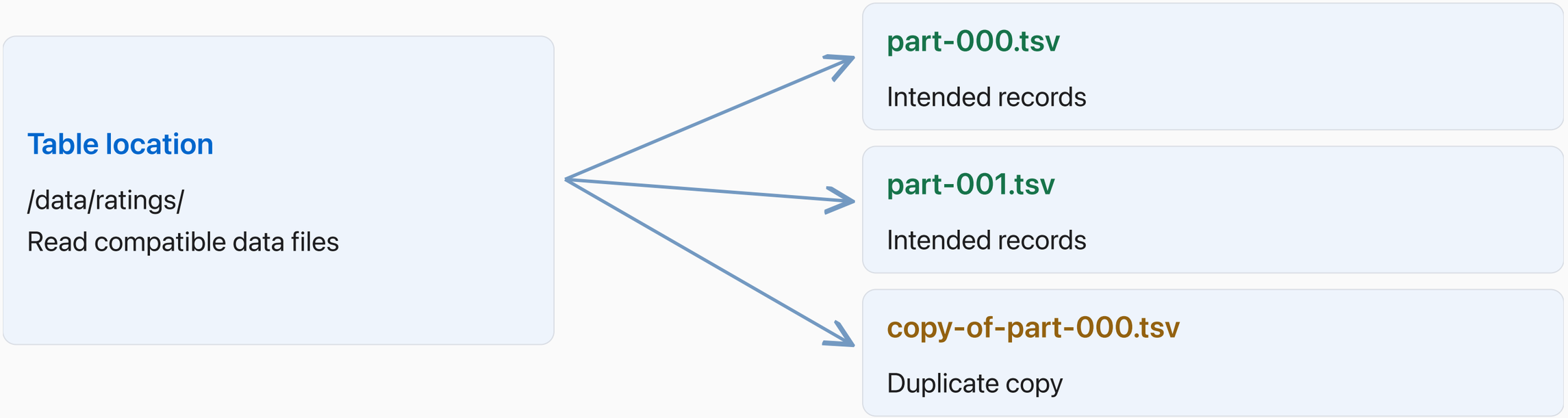


Table validation

Validate both the table definition and the interpreted values. A successful CREATE TABLE only confirms registration.

Check row count and valid rating range; investigate nulls and unexpected identifiers.

```
DESCRIBE FORMATTED ratings;
SELECT * FROM ratings LIMIT 5;
SELECT COUNT(*) FROM ratings;
SELECT MIN(rating), MAX(rating) FROM ratings;
SELECT COUNT(*) FROM ratings WHERE rating IS NULL;
```

For the six-row example

Count = 6

Rating range = 3 to 5

Null rating count = 0

Managed and external tables

A managed table normally includes data in Hive’s lifecycle. An external table normally separates catalog registration from the existing files.

DROP behaviour also depends on release and properties; external does not mean that data is protected from every deletion path.

Operation	Managed table, normally	External table, normally
CREATE	Hive manages table data lifecycle	Register an existing location
DROP TABLE	Remove metadata and table data	Remove metadata; retain files
Important exception	Purge and storage settings matter	External purge can remove files
Protection	Use appropriate access and recovery controls	Use appropriate access and recovery controls

Deletion behaviour

An external table points to an important dataset. Its release supports external.table.purge=true, and that property is enabled.

Dropping this table can delete its data too; inspect the actual metadata and release before changing it.

```
DESCRIBE FORMATTED ratings;
SHOW TBLPROPERTIES ratings;
```

Inspect before a destructive action

Table type

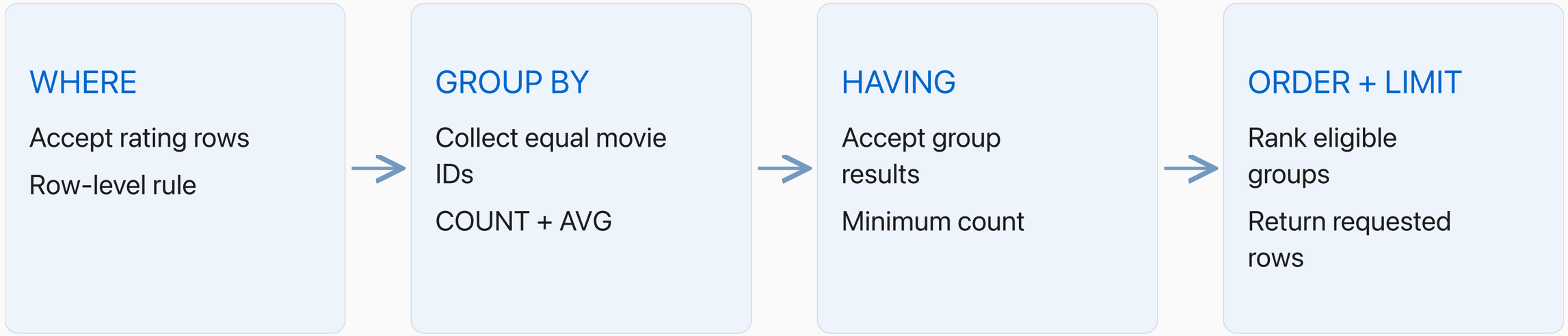
Storage location

Purge properties + release support

Query workflow

SQL describes the required relation. For this analysis, rows are filtered, grouped by movie, aggregated, filtered by group result, then ordered.

This is the logical meaning of the query; the physical engine may rearrange equivalent work.



Projection and filtering

SELECT chooses output expressions; WHERE keeps rows for which its condition is true.

For rating >= 4, the six-row example keeps four ratings: 5, 4, 5 and 5.

```
SELECT user_id, movie_id, rating
FROM ratings
WHERE rating >= 4;
```

Rows that survive

User 1 · movie 10 · rating 5

User 3 · movie 20 · rating 4

User 4 · movie 20 · rating 5

User 6 · movie 30 · rating 5

COUNT and NULL

COUNT(*) counts rows. COUNT(column) counts rows where that column is non-null.

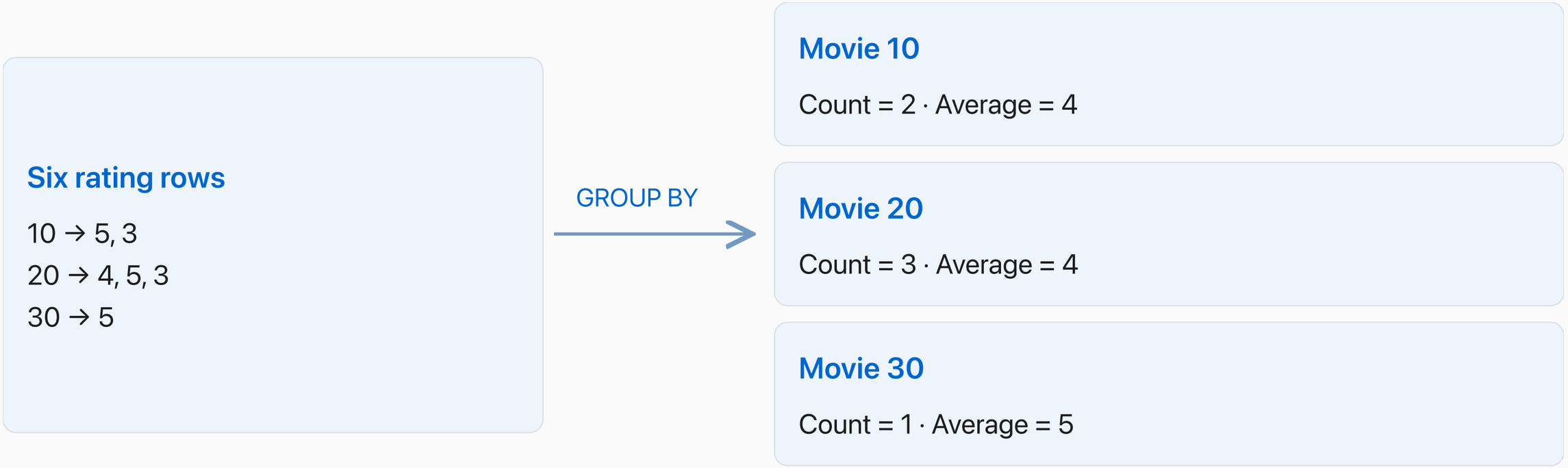
In a variation with ratings 5, NULL and 3, the two counts are 3 and 2 respectively.

Input rating	COUNT(*) contribution	COUNT(rating) contribution
5	1	1
NULL	1	0
3	1	1
Total	3	2

Grouped counts

GROUP BY movie_id forms one group for each distinct movie identifier. COUNT(*) gives the number of rating rows in each group.

The six input rows become three groups with counts 2, 3 and 1.



Average ratings

AVG(rating) computes the mean of non-null rating values within each group.

Movie 10 averages 4, movie 20 averages 4, and movie 30 averages 5; a higher average can come from fewer ratings.

Movie	Ratings	Count and average
10	5, 3	2 ratings; average 4
20	4, 5, 3	3 ratings; average 4
30	5	1 rating; average 5

Most-rated movies

“Most rated” means largest rating count. It does not mean highest mean score.

Order by count and add movie_id as a tie-breaker so the requested top rows are well defined.

```
SELECT movie_id, COUNT(*) AS rating_count,
       AVG(rating) AS average_rating
FROM ratings
GROUP BY movie_id
ORDER BY rating_count DESC, movie_id
LIMIT 2;
```

Top two in the example

20 · count 3 · average 4

10 · count 2 · average 4

HAVING

HAVING filters aggregated groups. WHERE filters source rows before aggregation.

A minimum review count is a group condition; putting the same idea in a row predicate cannot test the completed group.

```
SELECT movie_id, COUNT(*) AS n, AVG(rating) AS avg_r
FROM ratings
WHERE rating BETWEEN 1 AND 5
GROUP BY movie_id
HAVING COUNT(*) >= 2 AND AVG(rating) >= 4
ORDER BY avg_r DESC, movie_id;
```

Eligible groups

10 · n = 2 · avg = 4

20 · n = 3 · avg = 4

30 excluded: only one rating

Rows and groups

A row-level rating threshold changes the values available for each average. An average threshold tests the group after its values have been combined.

Filtering out low ratings before calculating an average can inflate that average and answer a different question.

WHERE rating >= 4

Movie 10 keeps only 5

Average becomes 5

Lower rating discarded

HAVING AVG(rating) >= 4

Movie 10 uses 5 and 3

Average remains 4

Whole group qualifies

ORDER BY and LIMIT

ORDER BY establishes the requested result order. LIMIT restricts the number of returned rows.

LIMIT alone does not define the “top” movies; the ordering expression and tie-breaker define that choice.

LIMIT 2 alone

Two returned rows

No promised ranking

Cannot claim most rated

ORDER BY n DESC, movie_id

Largest counts first

Ties use movie ID

LIMIT 2 selects that order

Query exercise

Using the six original ratings, return movies with at least two ratings and an average of at least 4.

Write the GROUP BY and HAVING clauses, predict the result, and explain why movie 30 is excluded.

Movie	All ratings	Condition to evaluate
10	5, 3	Count ≥ 2 and average ≥ 4
20	4, 5, 3	Count ≥ 2 and average ≥ 4
30	5	Count ≥ 2 and average ≥ 4

Eligible movies

Movies 10 and 20 both have an average of 4 and enough ratings. Movie 30 has only one rating.

The HAVING conditions apply to the completed groups, so every valid rating contributes to the average.

```
SELECT movie_id, COUNT(*) AS n, AVG(rating) AS avg_r
FROM ratings
GROUP BY movie_id
HAVING COUNT(*) >= 2 AND AVG(rating) >= 4
ORDER BY movie_id;
```

Answer

10 · 2 · 4.0

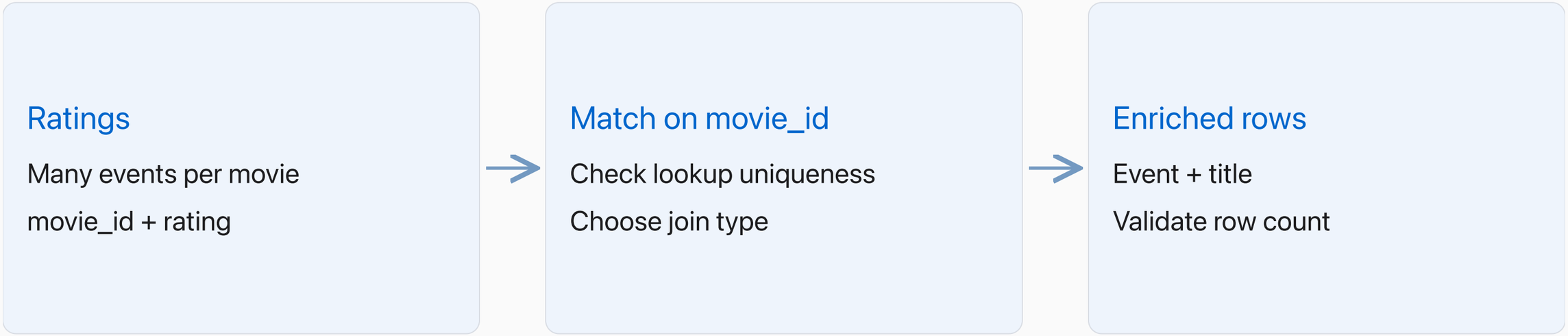
20 · 3 · 4.0

30 fails COUNT(*) >= 2

Ratings and movie titles

A join matches rows from two relations using a condition. Here ratings.movie_id links events to movie metadata.

The number of matching metadata rows determines whether rating rows are preserved, removed or multiplied.



Join inputs

Use the original six ratings and a movie lookup containing (10, Alpha) and (20, Beta). Movie 30 has no lookup row.

A unique lookup has at most one match per rating; a missing key still needs an explicit join policy.

Ratings

Movie 10: two ratings

Movie 20: three ratings

Movie 30: one rating

Movie lookup

10 → Alpha

20 → Beta

30 → no matching row

Inner join

An inner join returns matching row pairs. A left rating without a movie match does not appear in the result.

With the current lookup, the six ratings become five joined rows; movie 30 is lost.

```
SELECT r.movie_id, m.title, r.rating
FROM ratings r
JOIN movies m ON r.movie_id = m.movie_id;
```

Result structure

10 / Alpha: 2 rows

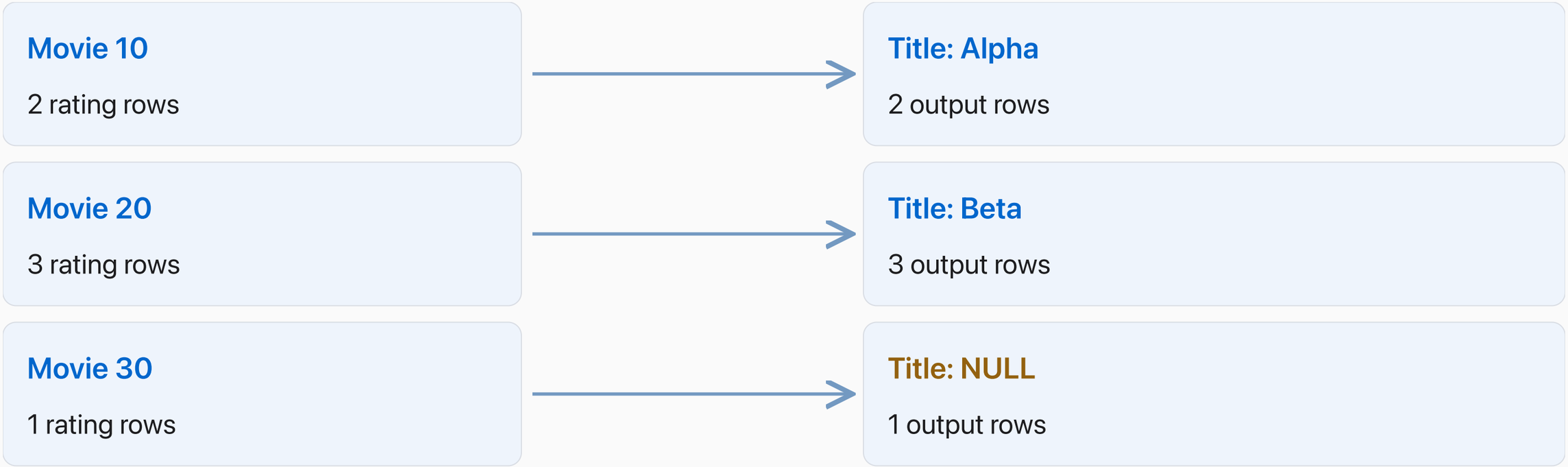
20 / Beta: 3 rows

30: no output row

Left join

A left join retains every left row, adding null right-side fields when no match exists.

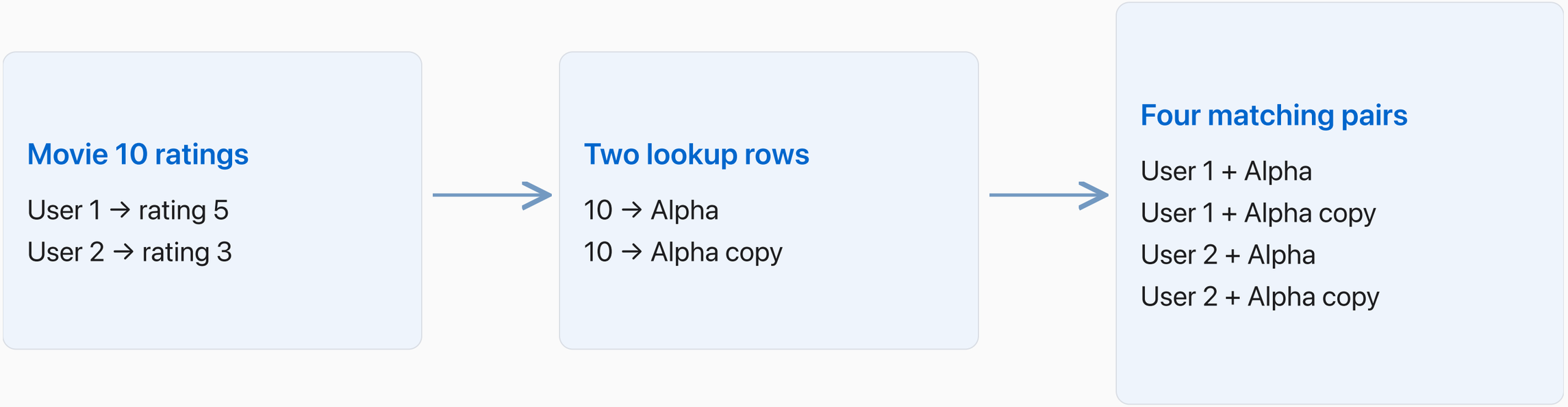
The unmatched movie-30 rating remains, with a null title; the six-row input still has six rows in this unique-lookup case.



Duplicate lookup keys

If movie 10 appears twice in the lookup, each of its two ratings matches both entries.

The left join now has 4 + 3 + 1 = 8 rows, so later counts and sums can be inflated.



Filtering after a left join

WHERE is applied to the joined result. A condition on a right-side column can reject the null fields of unmatched rows.

WHERE m.title = 'Alpha' removes movie 30 as well as other titles, despite the left join.

```
SELECT r.movie_id, m.title, r.rating
FROM ratings r
LEFT JOIN movies m ON r.movie_id = m.movie_id
WHERE m.title = 'Alpha';
```

With the unique lookup

Movie 10: two rows remain

Movie 20: title is not Alpha

Movie 30: null does not pass

Conditions in ON

An ON condition controls which right rows qualify as matches. A left join still retains left rows that have no qualifying match.

Moving a right-side condition from WHERE to ON changes the result when unmatched left rows must remain.

```
SELECT r.movie_id, m.title, r.rating
FROM ratings r
LEFT JOIN movies m
  ON r.movie_id = m.movie_id AND m.title = 'Alpha';
```

With the unique lookup

Movie 10: title Alpha

Movie 20: null title, rows retained

Movie 30: null title, row retained

Join validation

Check lookup-key uniqueness, unmatched keys and before/after row counts before aggregating joined data.

Group by stable identifiers as well as titles when different movies could share the same title.

```
SELECT movie_id, COUNT(*) AS n
FROM movies
GROUP BY movie_id
HAVING COUNT(*) > 1;
```

Evidence to compare

Duplicate lookup keys

Unmatched rating count

Input and joined row counts

Nested data

Nested records contain fields within fields. Hive complex types, such as STRUCT and ARRAY, describe those relationships.

A matching serializer/deserializer, or SerDe, tells Hive how records are decoded.



STRUCT and JSON SerDe

A STRUCT holds named fields within a column. This example reads an author object containing name and followers.

The JSON SerDe class and expected record layout must be supported by the installed Hive release.

```
CREATE EXTERNAL TABLE posts (
  id BIGINT, text STRING,
  author STRUCT<name:STRING, followers:INT>
)
ROW FORMAT SERDE
  'org.apache.hive.hcatalog.data.JsonSerDe'
LOCATION '/data/teaching/posts/';
```

Example JSON record

id: 1

text: "Hello"

author: {name: "Ana", followers: 1200000}

Nested field access

Dot notation selects a field inside a STRUCT. A nested input can still produce an ordinary flat SQL result.

For Ana with 1,200,000 followers and Ben with 500, the predicate retains only Ana’s post.

```
SELECT text, author.name, author.followers
FROM posts
WHERE author.followers > 1000000;
```

Result

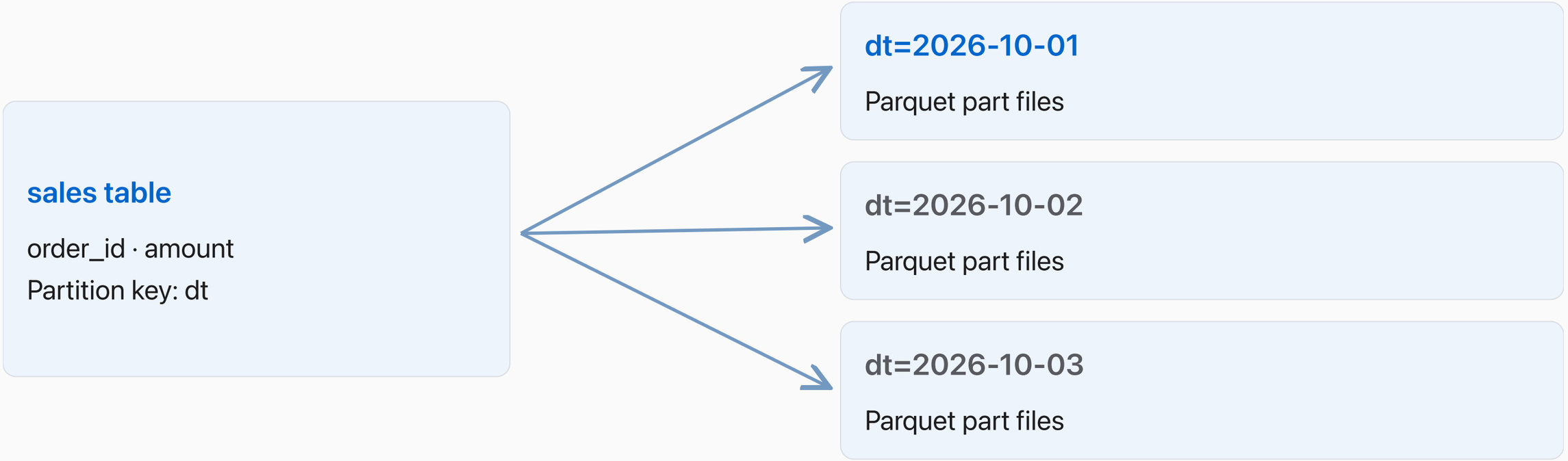
Hello · Ana · 1200000

Ben’s record does not qualify

Partitioned tables

A Hive partition maps a partition-key value to a storage location. Date partitioning lets date predicates exclude other locations.

A storage partition is different from a Spark task partition or a SQL grouping key.



Partition definition

PARTITIONED BY declares the directory-level keys separately from the fields stored inside each record.

Here the Parquet rows contain order_id and amount; dt is supplied by the registered partition value.

```
CREATE EXTERNAL TABLE sales (
  order_id STRING, amount DOUBLE
)
PARTITIONED BY (dt STRING)
STORED AS PARQUET
LOCATION '/data/teaching/sales/';
```

Stored row vs partition

Row: o1, 10.0

Partition: dt=2026-10-01

Query row includes both

Partition registration

An existing directory needs a corresponding catalog partition entry before Hive can reliably query it as a table partition.

Creating the table does not automatically register every pre-existing directory.

```
ALTER TABLE sales ADD IF NOT EXISTS
PARTITION (dt='2026-10-01')
LOCATION '/data/teaching/sales/dt=2026-10-01/';
SHOW PARTITIONS sales;
```

Check both sides

Directory exists and contains data

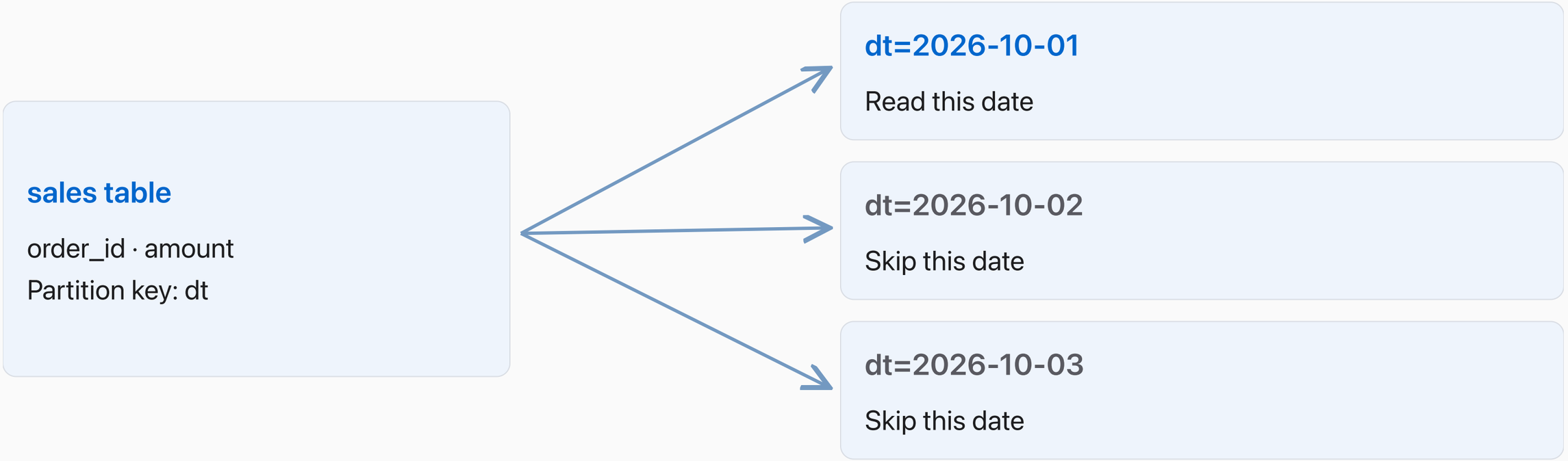
Catalog entry has correct dt

Registered location matches files

Partition pruning

Partition pruning excludes table partitions that cannot match a predicate on the partition key.

A query for October 1 can skip October 2 and 3; predicates on amount still need data-level evaluation.



Plan evidence

EXPLAIN shows the planned operators and selected input structure. Runtime evidence is needed to confirm actual work and cost.

Check that the date predicate selects the intended partition, then inspect measured input and execution behaviour.

```
EXPLAIN
SELECT SUM(amount)
FROM sales
WHERE dt = '2026-10-01';
```

Inspect

Partition predicate

Selected paths or partitions

Scan and aggregation operators

Layout trade-offs

Partition keys should support frequent selective filters. Very many tiny partitions can add metadata, listing and small-file overhead.

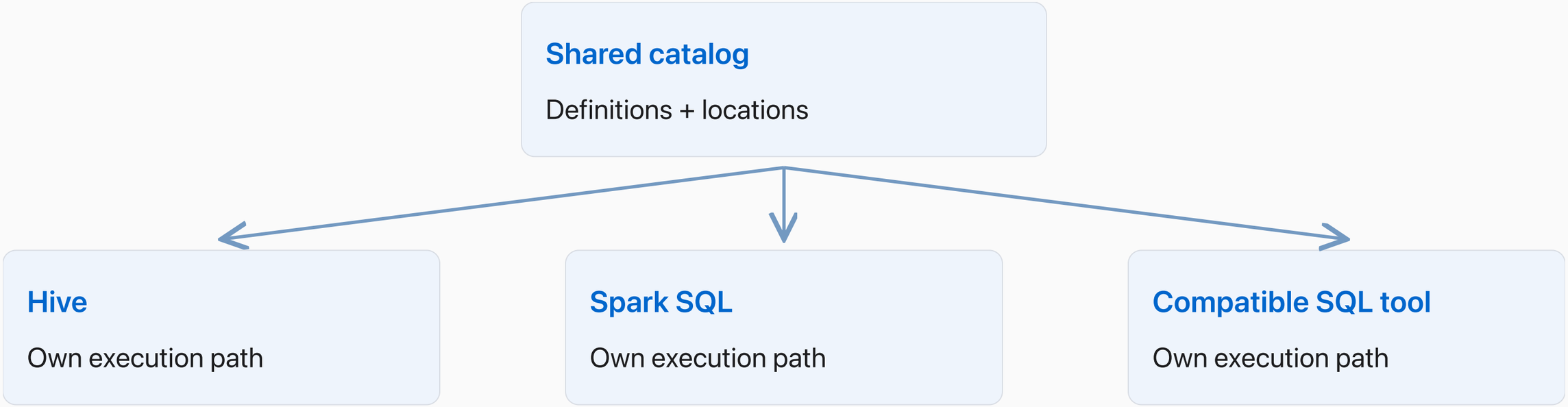
Parquet column pruning and row-group statistics complement directory pruning; none changes the required query result.

Layout choice	Potential benefit	Trade-off
Date partition	Skip unrelated dates	Too fine a layout makes many files
Parquet columns	Read needed columns	Depends on reader and layout
Row-group statistics	Skip impossible matches	Some groups still need reading
Compacted files	Reduce small-file overhead	Requires controlled rewrite work

Shared catalog

A shared catalog lets compatible engines resolve the same table definitions and storage locations.

Shared metadata does not guarantee identical SQL semantics, permissions, format support or execution plans.



Shared metadata + shared storage · verify reader and semantic compatibility

Data lake

A data lake stores datasets in shared storage, often in several formats. Catalog metadata makes selected datasets discoverable and interpretable.

External tables provide structure over files; data quality, access controls and lifecycle policies remain separate responsibilities.

Storage

Raw and curated datasets

Independent durable location

Catalog

Names, schemas and locations

Discoverable definitions

Use and governance

Compatible query engines

Quality, access and retention

Lab 10

Lab 10 stages MovieLens ratings, registers an external table, and runs counts, averages and grouped filters.

Use the explicit task goal: at least 100 ratings and average above 4.0, despite the “few reviews” heading.

Lab evidence	Interpretation
COUNT(*) = 100000	Expected for one intact MovieLens 100K file
Top 10 most rated	Rank by count, include average if requested
At least 100 and avg > 4	HAVING conditions on groups
Counts by rating	GROUP BY rating; order by rating
Reflection and cleanup	Explain path from SQL to files; terminate cluster

Preserve the result

Cluster HDFS is tied to the cluster lifecycle. An external table does not make those files survive cluster termination.

Save required query evidence and data to the approved independent location, verify them, then terminate the cluster.

Complete queries

Check rows and aggregates

Capture required evidence

Preserve

Independent storage

Verify saved copy

Terminate

End cluster in console

Confirm final state

Files, metadata and results

A reliable Hive result requires correct files, matching metadata and a query that expresses the intended population and aggregation.

Trace each result back through the query, table definition and stored records.

Files

Correct dataset

No duplicate upload

Metadata

Schema + format

Location + partitions

Query

Rows + groups + joins

Required ranking

Evidence

Checked result

Saved independently

Component roles

The catalog supplies definitions, the driver coordinates the query, the engine executes work, and storage contains the data.

A table name is metadata, a query plan is computation, and the record bytes remain in storage.

Component	Role	Example
Metastore	Describe tables	ratings schema and location
Driver	Resolve and coordinate query	Plan the grouped count
Execution engine	Run distributed work	Scan and combine movie counts
Storage	Hold record bytes	HDFS ratings.tsv

Schema and lifecycle

Schema-on-read interprets existing records. Managed and external table types govern lifecycle defaults, with release and property exceptions.

Validate format and values, and inspect deletion settings before changing a valuable table.

Interpretation

Column order + types

Delimiter or SerDe

Location + compatible files

Lifecycle

Managed vs external defaults

Release + purge properties

Independent storage protection

Query meanings

WHERE selects rows; GROUP BY and aggregates form group results; HAVING selects groups; ORDER BY defines ranking.

The six ratings yield counts 2, 3 and 1, averages 4, 4 and 5, and a most-rated winner of movie 20.

Business quantity	SQL idea	Example answer
Number of rating events	COUNT(*)	6
Number of movie groups	GROUP BY movie_id	3
Most-rated movie	Order group counts descending	20, with 3 ratings
Highest mean score	Order group averages descending	30, with average 5

Join consequences

An inner join can remove unmatched ratings. A left join preserves them with null right fields, but duplicate keys can multiply them.

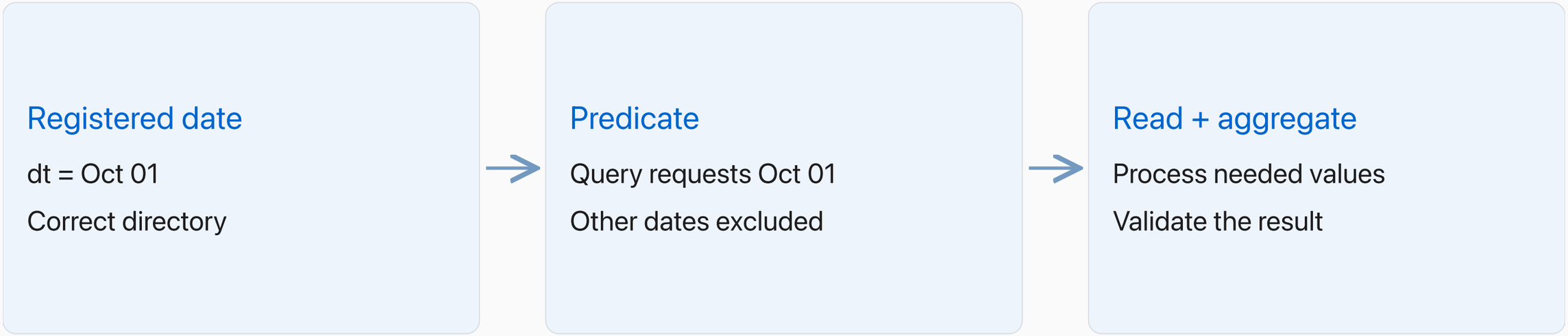
A right-side WHERE predicate can remove unmatched rows after a left join; an ON predicate changes which rows match.

Case with original six ratings	Output effect
Unique lookup for 10 and 20; inner join	5 rows; movie 30 removed
Same lookup; left join	6 rows; movie 30 title is null
Duplicate movie 10 lookup; left join	8 rows; movie 10 events multiply
WHERE title = Alpha after left join	Only qualifying matched rows survive

Selective input

A partition entry links a key value to a location. A matching predicate can exclude other locations before their rows are read.

The catalog and directory layout must agree; plan inspection and runtime evidence support performance claims.



Final SQL case

Use the six ratings and the unique lookup for movies 10 and 20. Return every movie’s count and average, including missing titles.

Choose the join, grouping keys and order for the two most-rated movies; then state what must be preserved before cluster shutdown.

Required population

All six rating events

Missing movie titles allowed

Required result

One row per movie

Count + average

Top two by rating count

Checked analytical result

A left join with the unique lookup retains all six ratings. Grouping by movie ID and title produces three groups; ranking counts returns movies 20 and 10.

Verify row multiplicity and the saved result before ending the cluster.

```
SELECT r.movie_id, m.title, COUNT(*) AS n,
       AVG(r.rating) AS avg_r
FROM ratings r LEFT JOIN movies m
  ON r.movie_id = m.movie_id
GROUP BY r.movie_id, m.title
ORDER BY n DESC, r.movie_id
LIMIT 2;
```

Top two

20 · Beta · 3 · 4.0

10 · Alpha · 2 · 4.0

Preserve query + checked output