READ MODE · ANSWER FIRST

How Apache Pinot's multi-stage query engine works

See how Pinot plans SQL, dispatches stages, scans segments, shuffles rows through mailboxes, joins data, and returns results.

CHEAT SHEET · 8 ESSENTIAL IDEAS

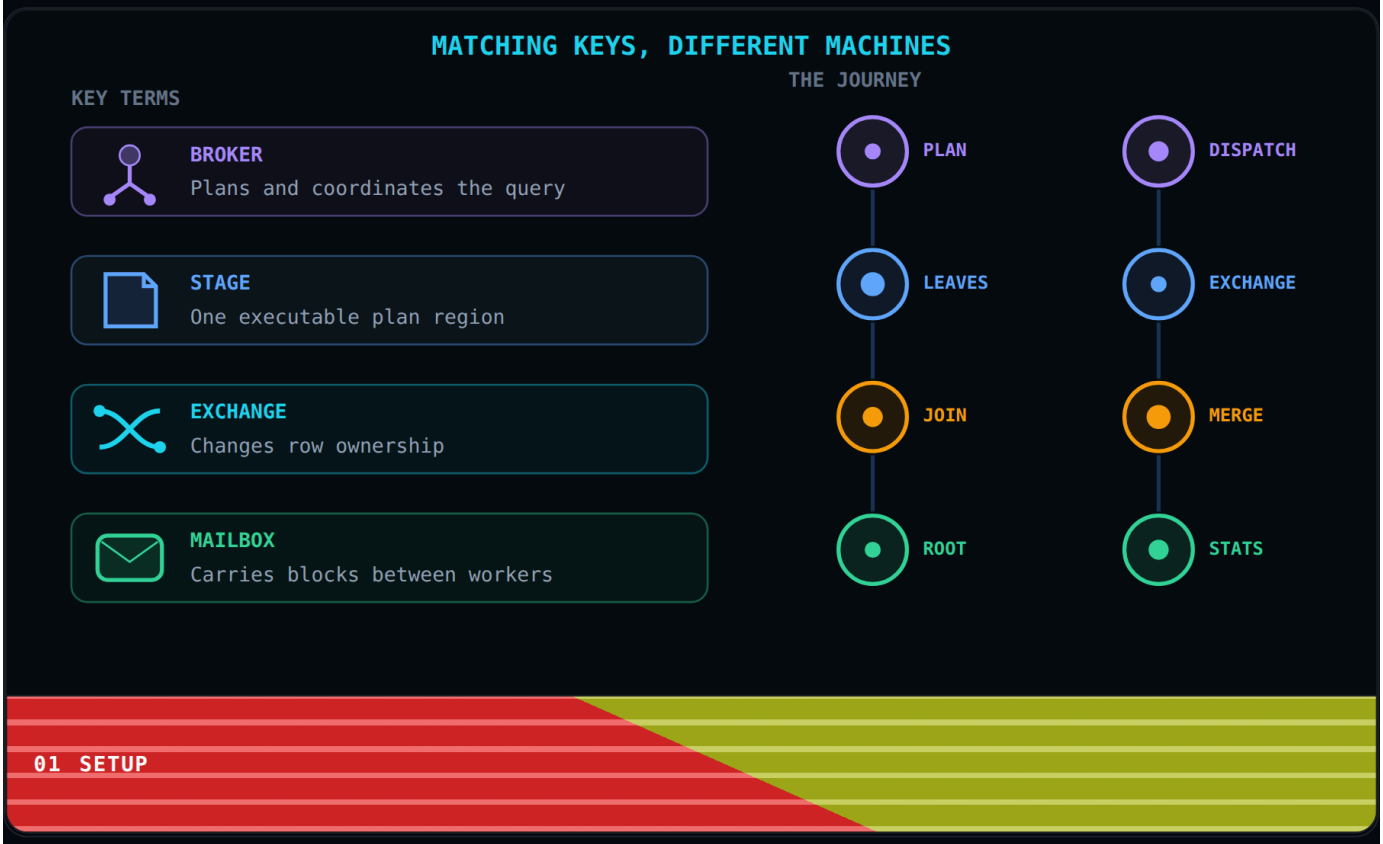
The whole story in 8 lines

Pinot turns complex SQL into parallel stages whose mailboxes place the right data at each worker before the broker returns one answer.

1. The broker translates SQL into an optimized operator tree before any table data moves.
2. Exchange boundaries divide one logical tree into independently assigned stage fragments.
3. Leaf stages reuse Pinot segment execution so filters and indexes reduce rows near storage.
4. Mailboxes repartition or copy row blocks so downstream workers receive the data they need.
5. Each join worker builds the right input first, then probes it with its left input.
6. Parallel partial aggregation reduces data before the final groups are merged.
7. The root gathers final rows, applies ordering and limits, then returns one result table.
8. Per-operator stage statistics show whether scanning, transfer, joining, or output dominated the query.

STAGE 1 OF 10

Setup



Two tables contain the same customer key, yet their rows begin on different Pinot servers. No single server can complete this join from local data alone.

The broker receives SQL and builds the distributed plan. Think of it as the coordinator that decides which work belongs on which machines.

A stage is one connected portion of that plan. Many workers can run separate instances of the same stage against different data partitions.

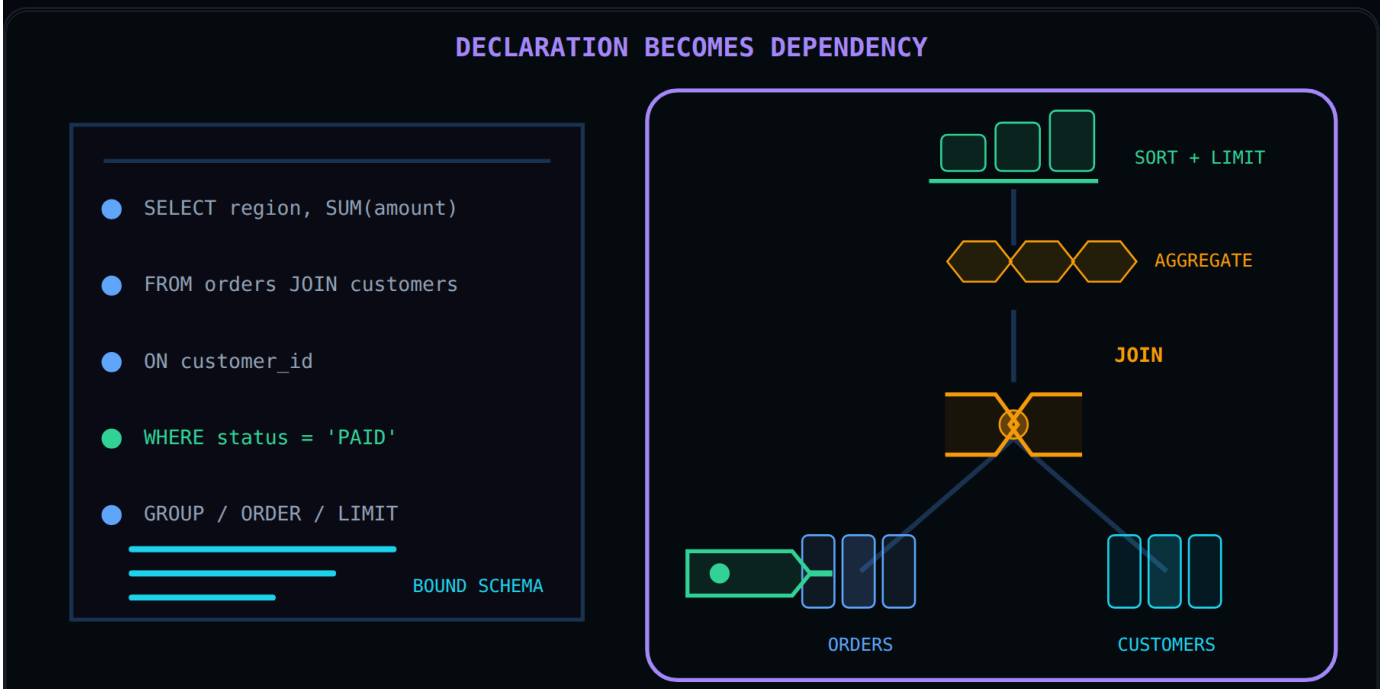
An exchange marks where rows must change ownership between stages. It can partition rows by a key or copy a smaller input more broadly.

A mailbox carries row blocks between workers across an exchange. Those transfers turn rows from unrelated servers into useful inputs for the next operation.

Our journey follows planning, dispatch, leaf scans, exchange, join, aggregation, root output, and diagnosis. Let us begin with the broker turning SQL into a plan.

STAGE 2 OF 10

SQL Becomes a Plan



02 SQL BECOMES A PLAN

We now understand why rows need a distributed handoff. The broker first reads our SQL as a request for regions ranked by paid order revenue. The desired answer is still independent of machines and segments.

Parsing turns clauses into a syntax tree, then binding resolves table names, columns, and data types. The query now has precise meaning.

The optimizer pushes the paid order filter toward the orders scan. Early filtering matters because discarded rows should never consume mailbox bandwidth later.

The broker still needs a concrete dependency order before it can assign work. Which operation must consume both table branches before aggregation can begin?

PAUSE AND PREDICT

Which operation must consume both table branches before aggregation can begin?

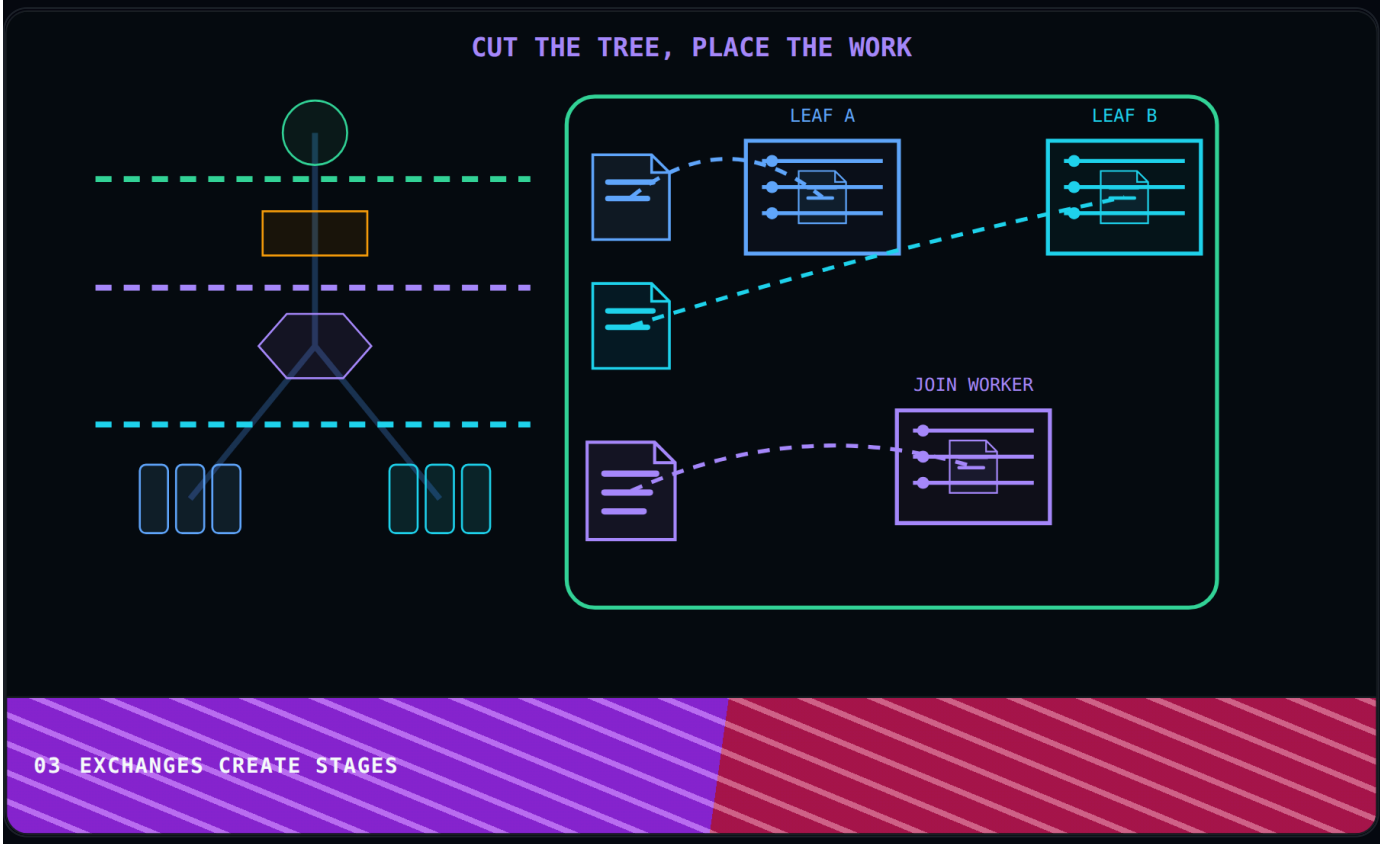
- The final sort
- The customer join
- The row limit

Skip this check

The optimized tree now has two scans feeding a join, followed by aggregation, sorting, and a limit. This structure preserves SQL meaning while exposing dependencies.

The tree explains what must happen, but not yet where each operation runs. Next, we will cut this tree into stage fragments and assign workers.

Exchanges Create Stages



03 EXCHANGES CREATE STAGES

Now that the operator tree is ready, the broker must decide where execution changes hands. Exchange nodes mark those ownership boundaries inside the tree. The tree still represents logic, not a running task.

The two scan branches become leaf stages because they read Pinot tables. Their workers are chosen from servers that own the required segments.

The join and aggregation occupy intermediate stages where workers process rows from earlier stages. These workers can be different from the original segment owners.

The broker assigns parallelism and worker locations to each stage. One server may execute several stage instances when the plan and cluster placement require it.

Compact plan fragments now leave the broker for their assigned workers. The same stage logic fans out, while each worker receives its own input partition.

Planning has become executable placement, but the leaf workers still need data. Next, we will see how they read segments and reduce rows locally.

Leaf Stages Read Segments

We just assigned the leaf fragments to servers that own table segments. Each leaf stage now bridges the distributed plan into Pinot’s segment execution engine. The leaf wrapper preserves that local engine’s useful optimizations.

The orders branch checks segment metadata and indexes for the paid status predicate. Rows that cannot match are rejected before their other values are materialized.

The customer branch reads only customer ID and region. Projection matters because every unused column would increase processing and transfer without helping the join.

Both leaves work at the same time on their own segments. The orders branch removes one pending order while the customer branch emits five compact rows.

The leaf results become transferable row blocks instead of complete query answers. Seven paid orders and five customer rows are ready for the exchange.

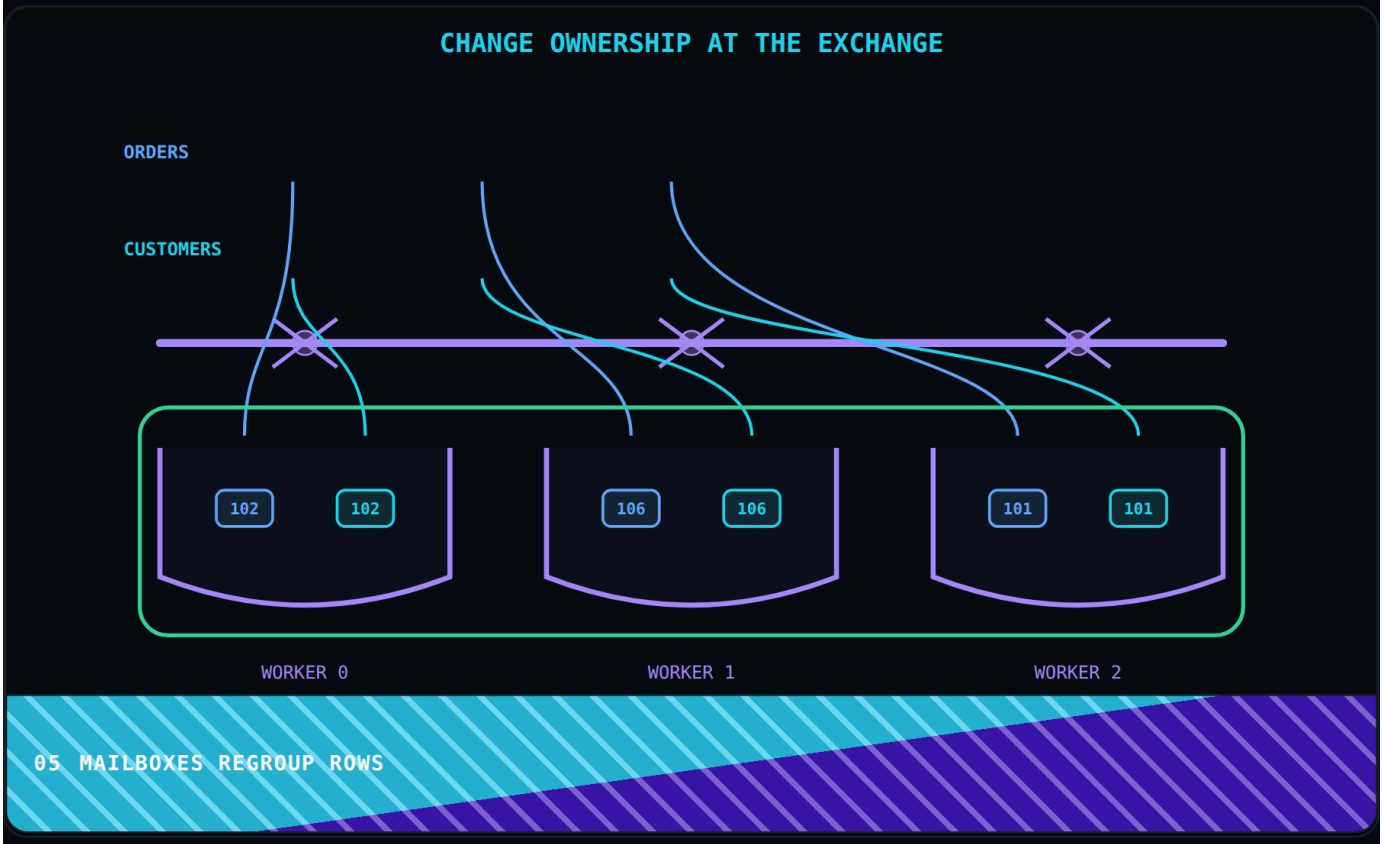
Local pruning has reduced the input, but matching keys still live on unrelated servers. Next, the mailbox exchange will give every row a destination.

STAGE 5 OF 10

Mailboxes Regroup Rows

★ If you remember one thing · A hash exchange makes a distributed join local by sending equal keys to the same worker.

Try it in Watch mode. Make every worker receive the complete customer side.



05 MAILBOXES REGROUP ROWS

The leaves produced useful blocks, but their row order still reflects segment ownership. Matching customer keys can begin on opposite sides of the cluster.

Mailbox send operators open logical routes toward the next stage. Across servers these routes normally use gRPC, while same-server workers can exchange on heap.

The join workers must stop depending on where segments happened to live. Where should rows with the same customer key arrive?

PAUSE AND PREDICT

Where should rows with the same customer key arrive?

- At the same worker
- At their original servers
- At random workers

```
Skip this check
```

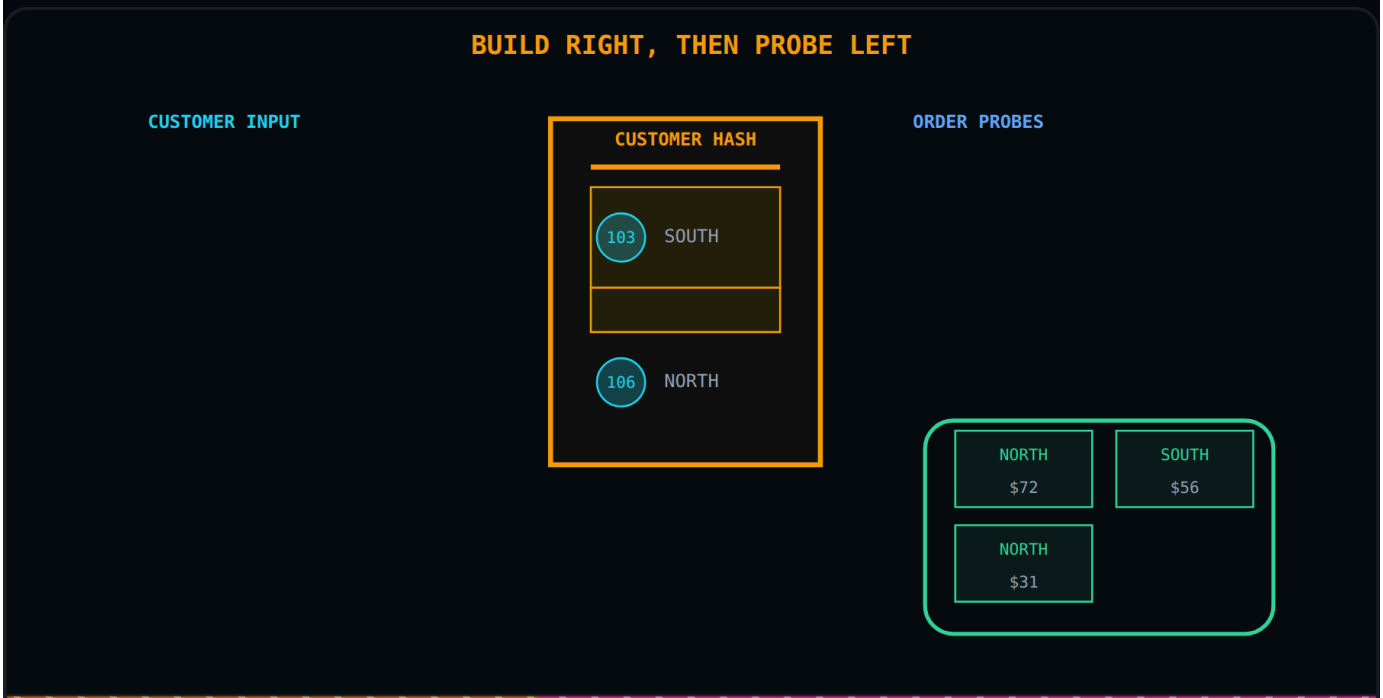
Hash partitioning now crosses the source lanes and regroups equal keys. Each destination worker receives matching order and customer rows for its own key partition.

Switch the Exchange control through both choices. Compare equal-key partitioning with copying the complete customer side to every worker.

Mailbox receive operators stream available blocks fairly from their inputs. The next stage can begin consuming without waiting for unrelated workers to finish everything.

The exchange has made each join partition self-contained. Next, we will open one worker and watch its right input build before the left input probes.

Workers Perform the Join



06 WORKERS PERFORM THE JOIN

Now that every partition has matching keys, we can look inside one intermediate worker. Its operator chain receives two mailbox inputs for the join.

The customer relation is the right input and arrives first. Pinot consumes it into a hash table keyed by customer ID before producing joined rows.

Each customer key occupies a distinct bucket with its region value. This in-memory state is why keeping the smaller relation on the right usually helps.

The order input now streams through the worker and probes one bucket per key. What happens when an order finds customer 106 in the local hash table?

PAUSE AND PREDICT

What happens when an order finds customer 106 in the local hash table?

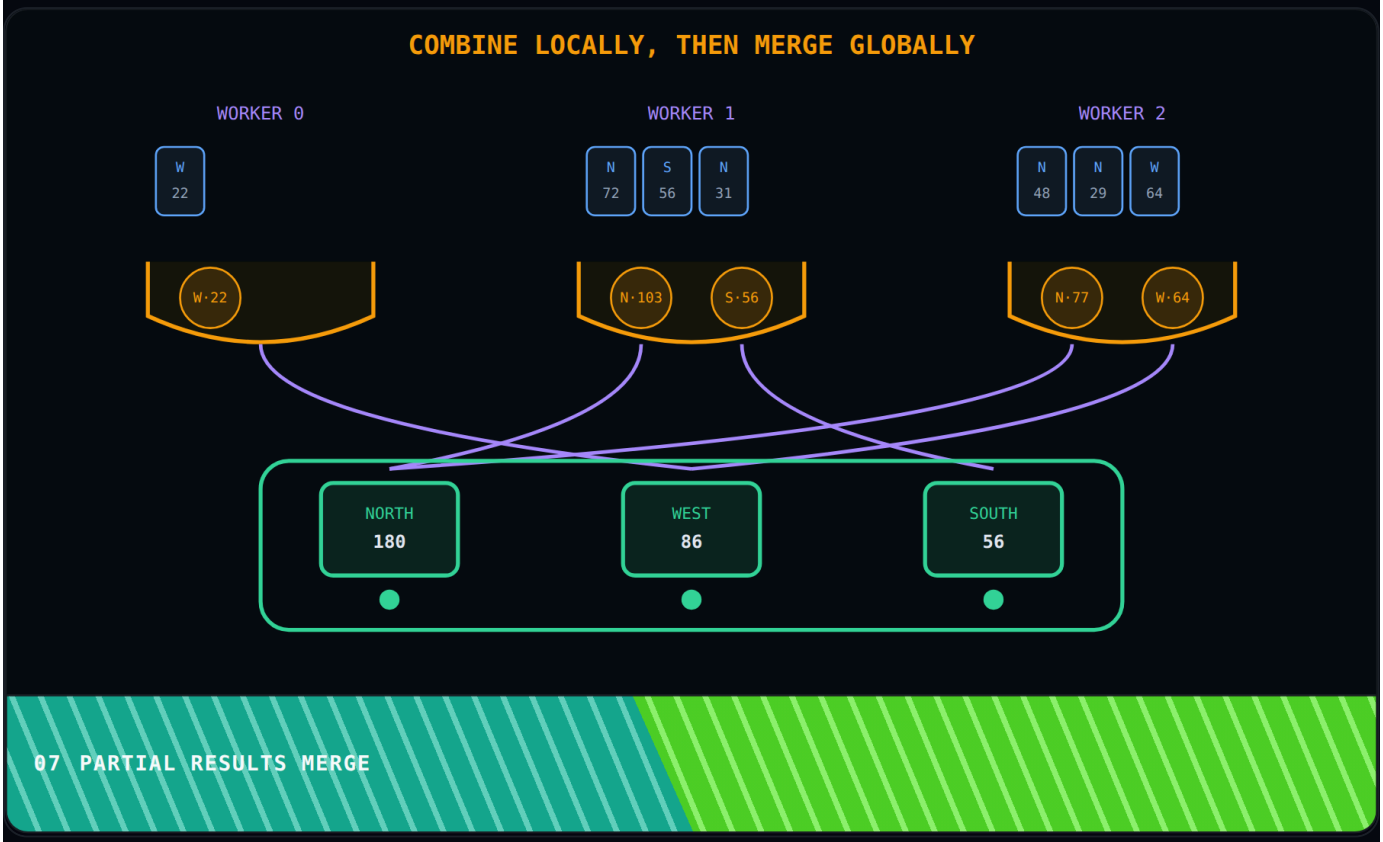
- **A joined row is emitted**
- **Both rows are reshuffled again**
- **The order is discarded**

Skip this check

Matching probes now produce composite rows containing order amount and customer region. Three workers perform this same operation independently over different key partitions.

The global join is complete because exchange ownership turned it into several local hash joins. Next, those joined rows will shrink into partial region totals.

Partial Results Merge



07 PARTIAL RESULTS MERGE

We just produced joined rows on three workers. The query asks for revenue by region, so keeping every individual order would waste later transfer and memory. This early reduction keeps every later input smaller.

Each worker accumulates its own rows into region buckets. Amount chips disappear into a smaller set of NORTH, WEST, and SOUTH partial states.

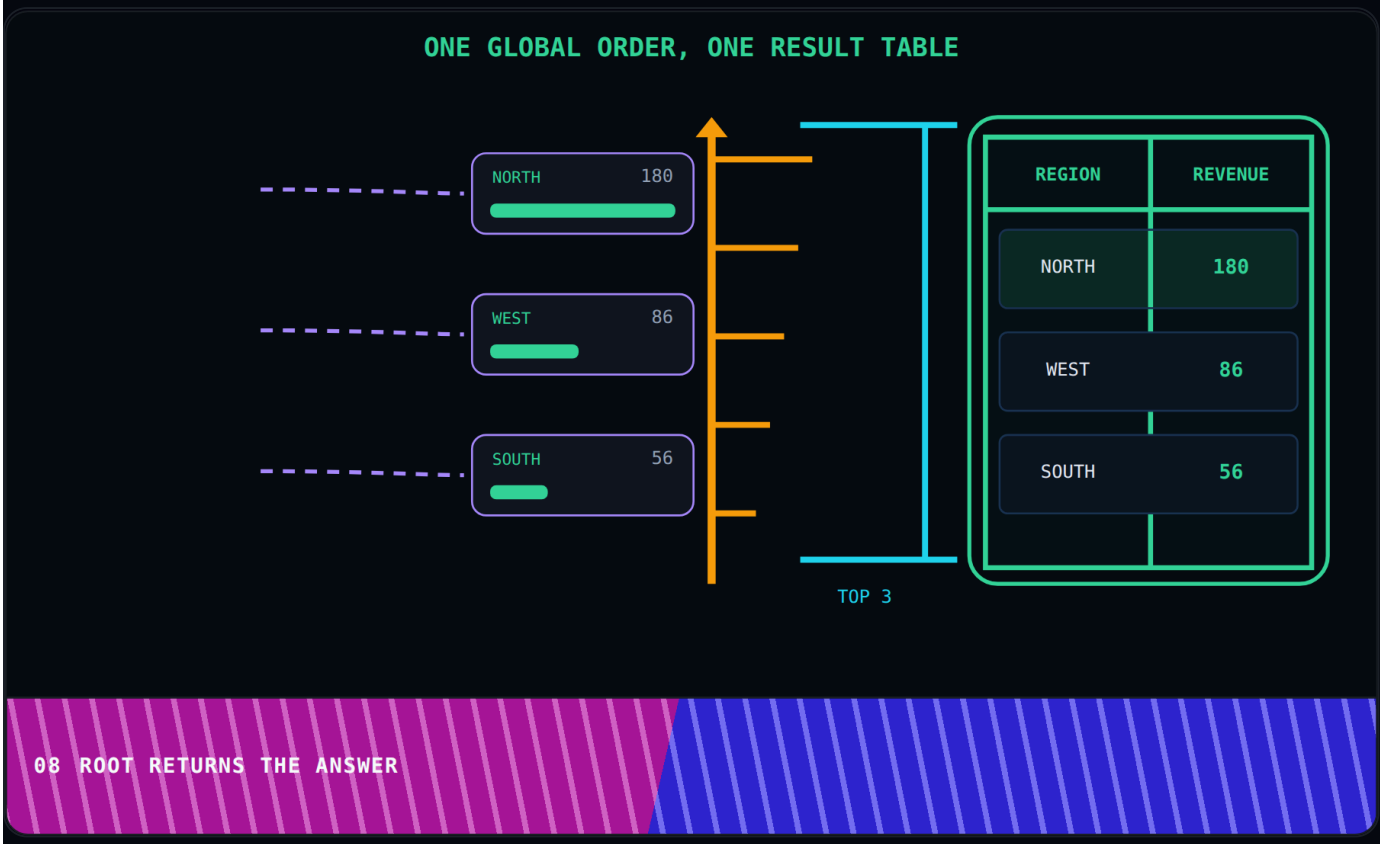
Worker zero produces a NORTH partial, while other workers may produce NORTH or WEST independently. These are not final because one region can span partitions.

The partial states cross another exchange keyed by region. Values with the same region now converge on the same final aggregation owner.

Final aggregation adds the partial values and produces NORTH 180, WEST 86, and SOUTH 56. Seven joined rows have become three grouped results.

Parallel work reduced the data before one final merge completed each group. Next, the root will order these totals and shape the client response.

Root Returns the Answer



08 ROOT RETURNS THE ANSWER

The distributed computation has produced three final region totals. The root now consumes the last exchange and owns the remaining global work. No client row exists until this global work finishes.

All totals enter one comparison space because ordering must consider every remaining region. Separate workers cannot independently decide the global ranking.

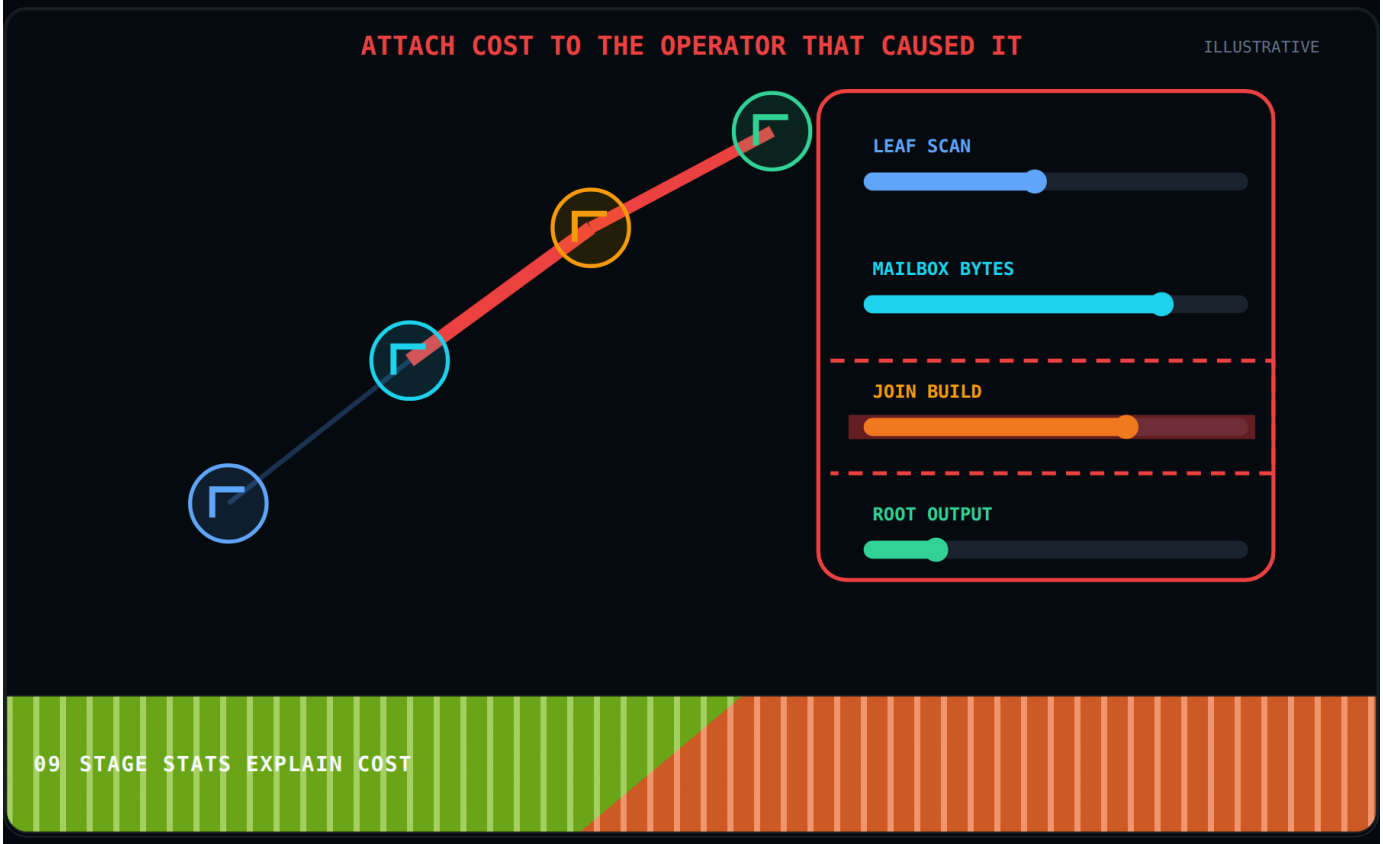
The sort comb arranges revenue from largest to smallest. NORTH leads with 180, followed by WEST at 86 and SOUTH at 56.

The limit keeps the requested top three rows, which happens to retain every region in this small example. Larger results would be trimmed here.

The ordered values lock into a result table with region and revenue columns. The broker can now return one coherent response to the client.

The answer is complete, but production work also requires explaining its cost. Those statistics travel beside the result without changing its rows. Next, we will read the operator statistics attached to this execution.

Stage Stats Explain Cost



The result arrived, yet latency alone cannot identify the expensive operation. Pinot attaches per-operator stage statistics to every multi-stage response by default. The answer and its execution evidence arrive together.

Leaf counters describe segments, documents, and filter scanning. They tell us whether storage work remained large before any distributed transfer began.

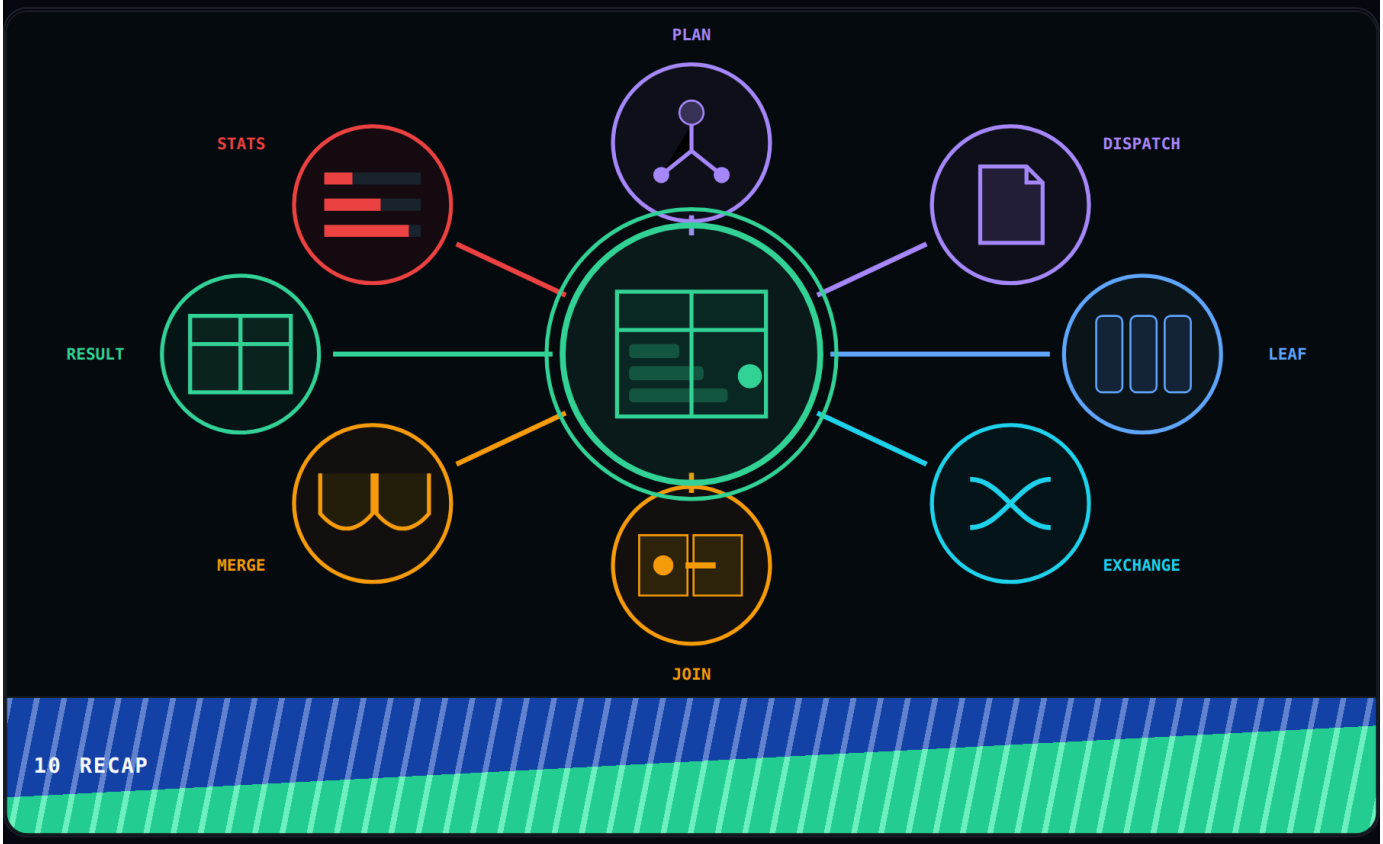
Mailbox counters separate on-heap messages from serialized network messages. Deserialized bytes and upstream wait time expose expensive movement or a slow producer.

Join counters show emitted rows and hash table build time. A large right input can make the in-memory build dominate both memory and latency.

The completed graph makes the bottleneck comparable across operators. In this illustrative trace, mailbox transfer and join building outweigh final result formatting.

MSE keeps intermediate work in memory, so very large shuffles deserve careful limits and plan review. That boundary shapes which workloads belong in this engine. Now let us connect the complete engine together.

Recap



We started with SQL becoming an optimized operator tree. That plan established the two scans, join, aggregation, ordering, and dependency direction.

Then exchange boundaries divided the tree into stage fragments. The broker assigned parallel workers and dispatched each fragment to its execution location.

Leaf stages reused Pinot segment execution to prune and filter near storage. Compact row blocks crossed into the distributed part of the engine.

Mailboxes changed row ownership across stage boundaries. Hash partitioning put equal keys together, while broadcast offered another plan for a small input.

Each join worker built a customer hash table before probing with paid orders. Partitioning made every match local inside one intermediate worker.

Local region totals reduced seven joined rows before final aggregation merged matching groups. Parallel work shrank the data before convergence.

The root gathered completed totals, applied global ordering and the limit, then formatted one result table for the client.

Stage statistics attached scan, transfer, join, and output costs to the operator graph. Those measurements turn a slow query into a diagnosable system.

Together, the stages separate planning, placement, local reads, data movement, intermediate computation, and final output. That separation is the core of Pinot MSE.