READ MODE · ANSWER FIRST

How DuckDB Executes a Query

An interactive, visual explanation of How DuckDB Executes a Query, from setup through the complete system.

CHEAT SHEET · 6 ESSENTIAL IDEAS

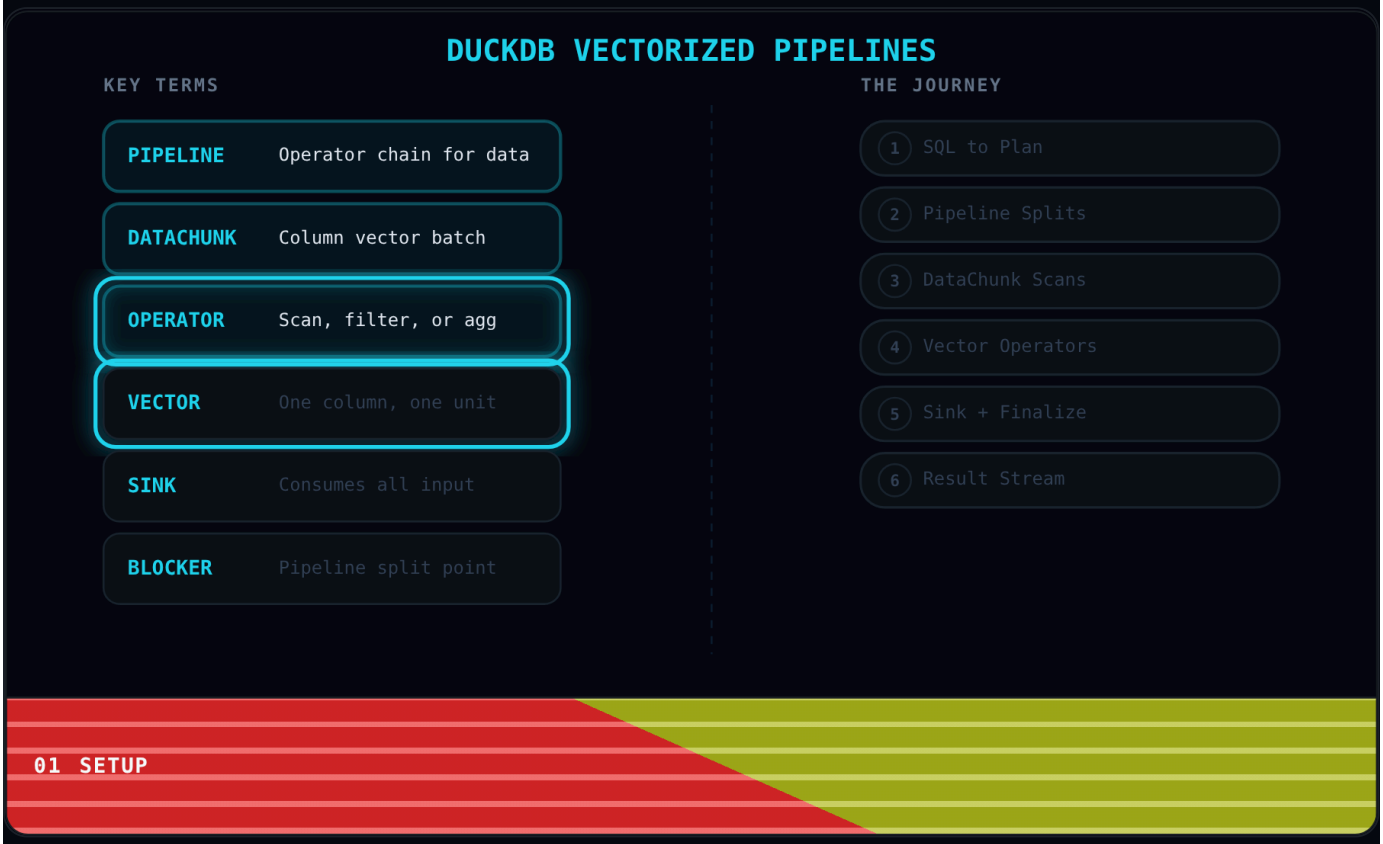
The whole story in 6 lines

How DuckDB Executes a Query becomes easier to reason about when every stage is connected as one system.

1. The physical plan is the contract execution follows. Controls change the shape of that contract before pipelines exist.
2. Pipeline breakers are where execution changes shape: a sink finishes, state materializes, and a new source pipeline starts from that state.
3. The engine stays columnar even during execution. A chunk is a small batch of column vectors plus metadata such as cardinality and vector...
4. Execution stays vectorized inside the pipeline: one operator call works over many rows at once, often with selection vectors instead of...
5. This is the pipeline break: sink, combine, and finalize are why a new pipeline has to start from materialized state instead of streaming...
6. Execution is still vectorized even on the way out. The downstream pipeline reads grouped state in chunks, applies any final blocking...

STAGE 1 OF 8

Setup



Welcome. DuckDB is an in-process database that runs SQL queries really fast. Think of it like a tiny power plant inside your application that crunches data without needing a separate server.

A pipeline is a chain of operators that data flows through without stopping. Picture an assembly line in a factory where each station does one job and passes the piece forward.

A DataChunk is a small batch of column values that travel together. Instead of moving one row at a time, DuckDB bundles many rows into a chunk, like shipping a crate of items instead of mailing them one by one.

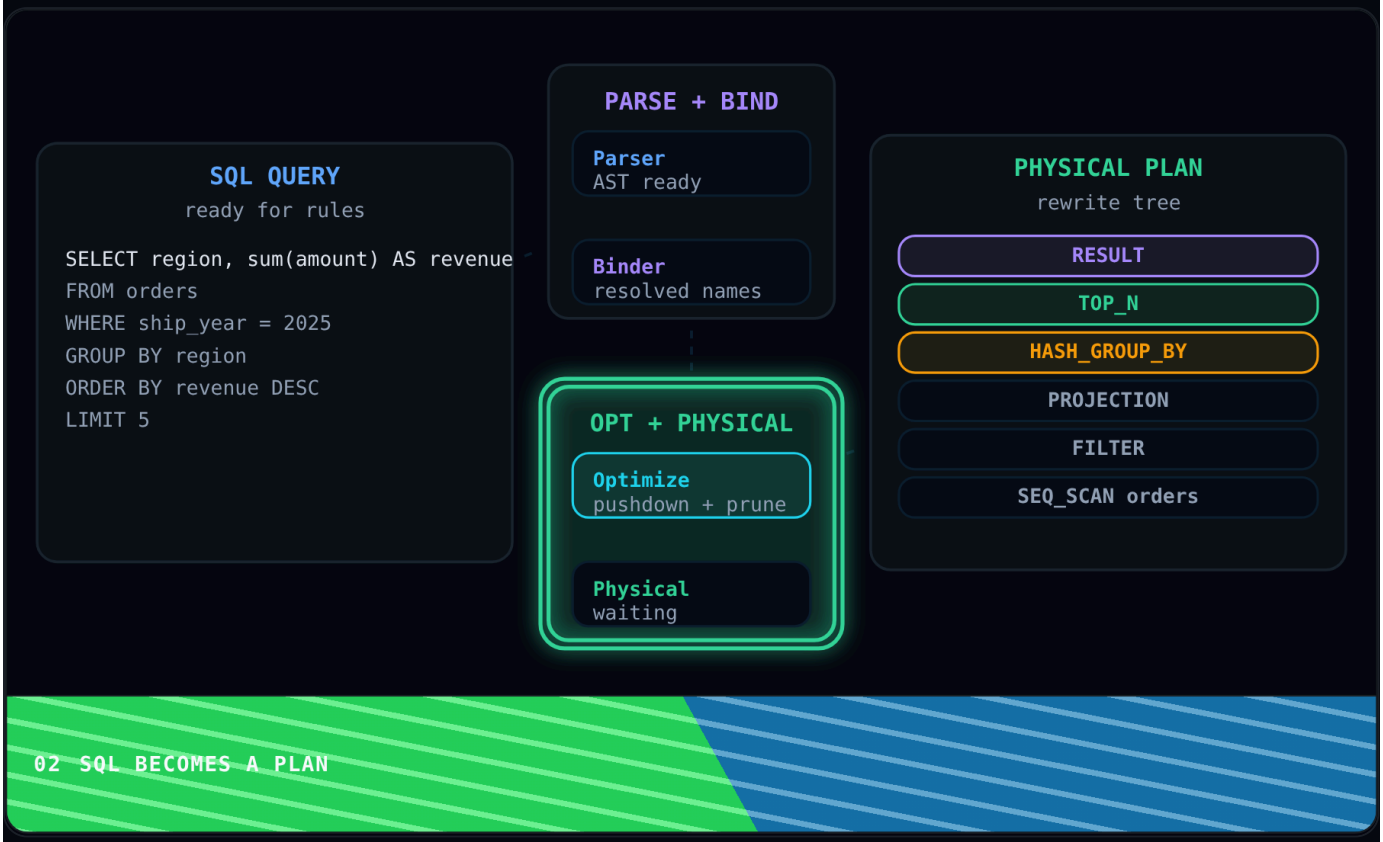
An operator is a single step such as scanning a table, filtering rows, or computing a sum. A vector is one column inside a DataChunk, processed as a single unit for speed.

A sink is an operator that must consume all of its input before anything downstream can start. A blocker is the point where one pipeline ends and a new one begins, right at a sink.

Over the next six stages we will follow a query from raw SQL text all the way to result rows. Let us start with how DuckDB turns your SQL into an executable plan.

STAGE 2 OF 8

SQL Becomes a Plan



Now that you know the key vocabulary, let us see what happens when DuckDB receives your SQL. The query text on the left is just declarative words. Nothing can execute yet.

The parser reads the SQL text and builds a syntax tree with SELECT, FROM, WHERE, and ORDER BY clauses. This syntax tree moves to the binder for name resolution.

The binder looks up table names, column names, and expression types in the catalog. Try toggling the Filter control on and off to see the WHERE clause appear or disappear from the plan.

The optimizer applies rewrite rules that push predicates down and trim unnecessary columns. These rewrites happen before any data moves.

DuckDB picks concrete physical operators like SEQ_SCAN, FILTER, HASH_GROUP_BY, and TOP_N. Toggle the Aggregate and Order controls and watch the physical plan panel gain or lose operator rows.

The physical operator tree is now complete. This tree is the contract that execution will follow. Next, we will see how DuckDB slices this tree into pipelines at blocking operators.

STAGE 3 OF 8

Pipelines Form at Blockers

03 PIPELINES FORM AT BLOCKERS

Now that we have a physical plan, DuckDB needs to decide which operators can run together in a single pass. The plan tree on the left is the starting point.

Non-blocking operators like SEQ_SCAN, FILTER, and PROJECTION fuse together into Pipeline 1. Data can flow through all of them without stopping.

A blocking operator ends that pipeline because it must sink every input chunk before anything downstream can start. Use the Blocker control to switch between Hash Agg and Window and watch the cut point change.

Once the blocker finishes, its materialized state becomes the source of a fresh Pipeline 2. This is the pipeline break we introduced in Setup.

Toggle the Order control off and you will see the third pipeline disappear. When ORDER BY is present, TOP_N adds one more short result pipeline after the blocker.

The scheduler fills worker slots with tasks from the currently ready pipeline. Increase the Workers stepper to see more slots light up. Next, we will look at how a scan operator inside a pipeline actually reads data.

STAGE 4 OF 8

Scans Emit DataChunks

04 SCANS EMIT DATACHUNKS

Now that we understand how pipelines form, let us zoom into the first pipeline and see where data actually comes from. DuckDB stores table data in columns, not rows.

Worker threads claim morsels from column storage so that multiple scan tasks run at the same time. Increase the Threads stepper and watch more morsel rows switch from "queued" to "claimed."

The scan operator reads only the columns this query needs. It pulls one morsel at a time and passes the raw column segments forward.

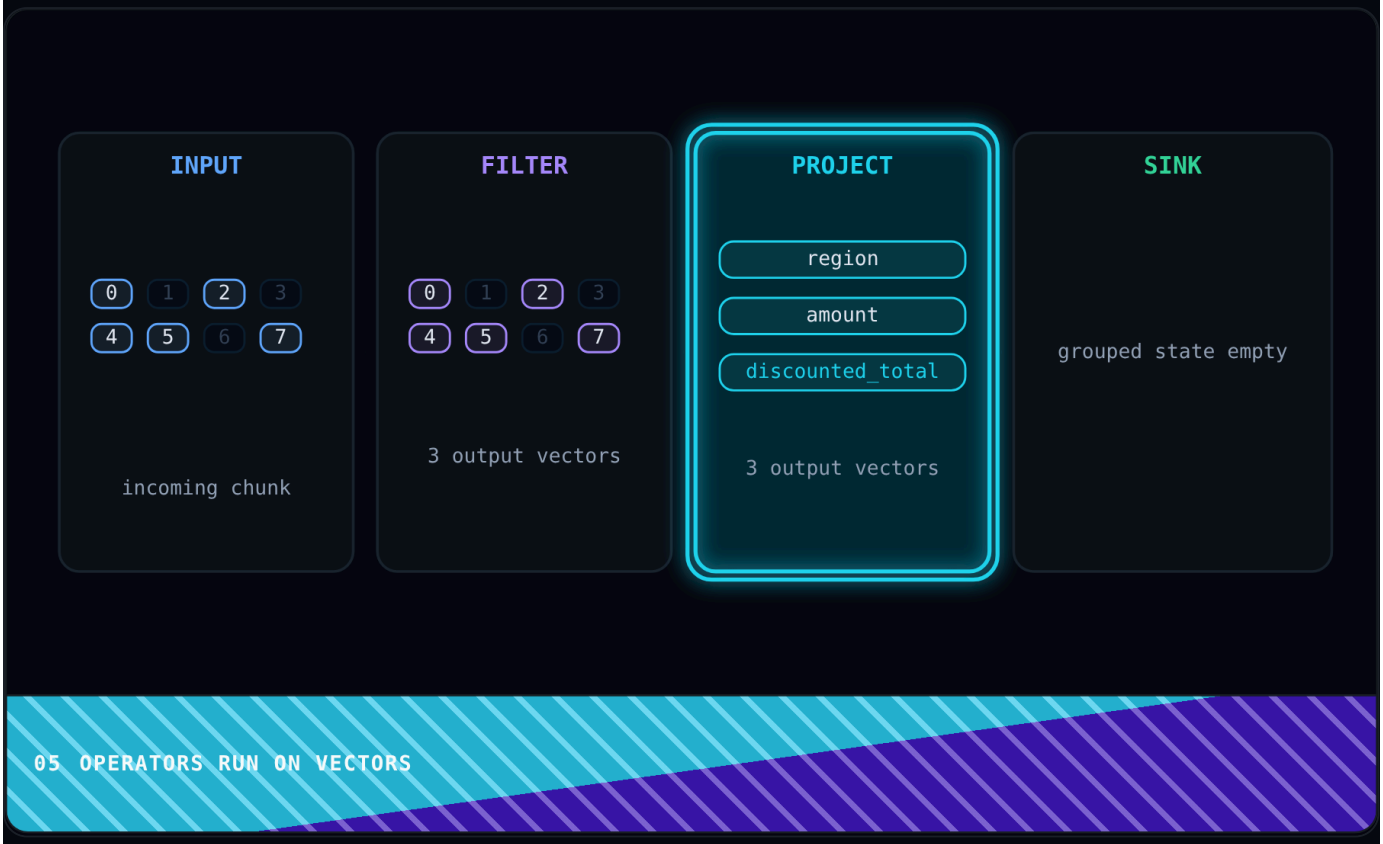
Those column segments are assembled into a DataChunk with a fixed logical row count. Use the Chunk Rows control to change the batch size and see the label update.

Each vector inside the DataChunk carries its own format. Switch the Format control between Flat, Dict, and Const to see how the ship_year vector encoding changes.

The finished DataChunk is now ready to travel through the rest of the pipeline. Next, we will see how operators process these vectors without touching one row at a time.

STAGE 5 OF 8

Operators Run on Vectors



05 OPERATORS RUN ON VECTORS

Now that we have a DataChunk, let us watch it flow through operators. The input panel shows all eight lanes active. Every lane holds one row of data.

The filter evaluates the predicate over the whole vector batch at once, not row by row. Toggle the Filter control off and you will see all eight lanes survive untouched.

Surviving lanes are tracked with a compact selection vector. Switch the Selectivity control between Dense, Medium, and Sparse and notice how the lane grid lights up differently.

Projection operators compute new output vectors only for active lanes. Increase the Cols stepper to add more output columns and watch additional rows appear in the project panel.

The sink operator at the end of this pipeline consumes the active lanes as grouped keys and values. You can see the group counters update in the sink panel.

DuckDB repeats this same vectorized loop for every DataChunk in the pipeline. Next, we will look at what happens inside that sink when it must block and accumulate all the input.

STAGE 6 OF 8

Sink, Combine, Finalize

06 SINK, COMBINE, FINALIZE

Now that we have seen operators run on vectors, let us look at what happens at the pipeline boundary. Chunks arrive at the sink operator and cannot pass through.

Each worker thread sinks its chunks into a private, thread-local hash table. Increase the Threads stepper and you will see more local state panels appear.

These local hash tables grow independently while input is still flowing. Switch the Groups control between Few and Many to see how the number of group keys changes the table contents.

Once all input is consumed, DuckDB combines those local tables so that each group key has exactly one merged state. This is the combine step.

Finalize seals the combined result into a readable source structure. The final state panel now shows the grouped rows.

Only now can the next pipeline start. The sink, combine, and finalize cycle is the reason DuckDB needs separate pipelines. Next, we will see how that downstream result pipeline streams rows out to the client.

STAGE 7 OF 8

Result Pipeline Streams Out

07 RESULT PIPELINE STREAMS OUT

Now that the sink has finalized, a brand new pipeline starts from the grouped state. This is the same pipeline break we saw in Stage 2, but now we are on the downstream side of it.

A source operator reads the grouped rows back into column vectors. The data is still columnar, just as it was when the scan first produced DataChunks.

If ORDER BY is present, a Top-N operator keeps only the best K rows. Use the Top K stepper to raise or lower the cutoff and watch the row count in the TOP_N panel change. Toggle the Order control off to skip this step entirely.

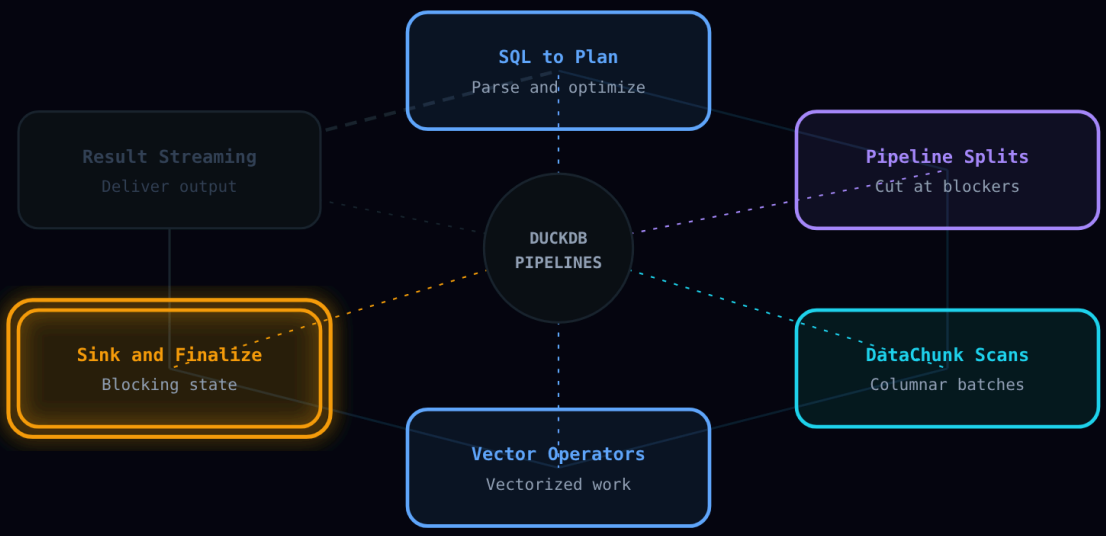
The result collector formats the final rows into output chunks for the client API. Increase the Batches stepper and you will see more batch cards appear in the result panel.

The client begins receiving result batches while the last pipeline drains. Each batch is still a columnar DataChunk, keeping the same format from start to finish.

The query is complete after the final output chunk is handed off. You have now seen the full journey from SQL text to result rows. Let us recap everything in the next stage.

STAGE 8 OF 8

Recap



08 RECAP

First, DuckDB parsed your SQL, resolved names and types, ran optimizer rules, and chose physical operators. The query became a concrete execution plan before any data moved.

Next, the engine sliced that operator tree into pipelines. Every time it hit a blocking operator like hash aggregation, it cut the flow and started a new pipeline on the other side.

Inside each pipeline, scan operators read column data from storage and assembled it into DataChunks. Multiple threads claimed morsels so scans could run in parallel.

Operators then processed those DataChunks as vectors, not row by row. Filters used selection vectors, projections computed new columns, and the whole batch stayed columnar for speed.

At pipeline boundaries, sink operators absorbed all incoming chunks into thread-local state. Combine merged those states, and finalize sealed the result so the next pipeline could read it.

Finally, a downstream result pipeline read the finalized state back into chunks, applied any remaining steps like Top-N ordering, and streamed output batches to the client.

That is the full loop. From SQL text to physical plan, through vectorized pipelines, past blocking sinks, and out as result batches. DuckDB keeps data columnar and batched at every step for maximum throughput.