Dissecting the Duck's Innards

⑥

Query Execution Plans and Pipelining

Winter 2025/26

Torsten Grust
Universität Tübingen, Germany

1 ¦ Query Execution ≡ Plan Execution

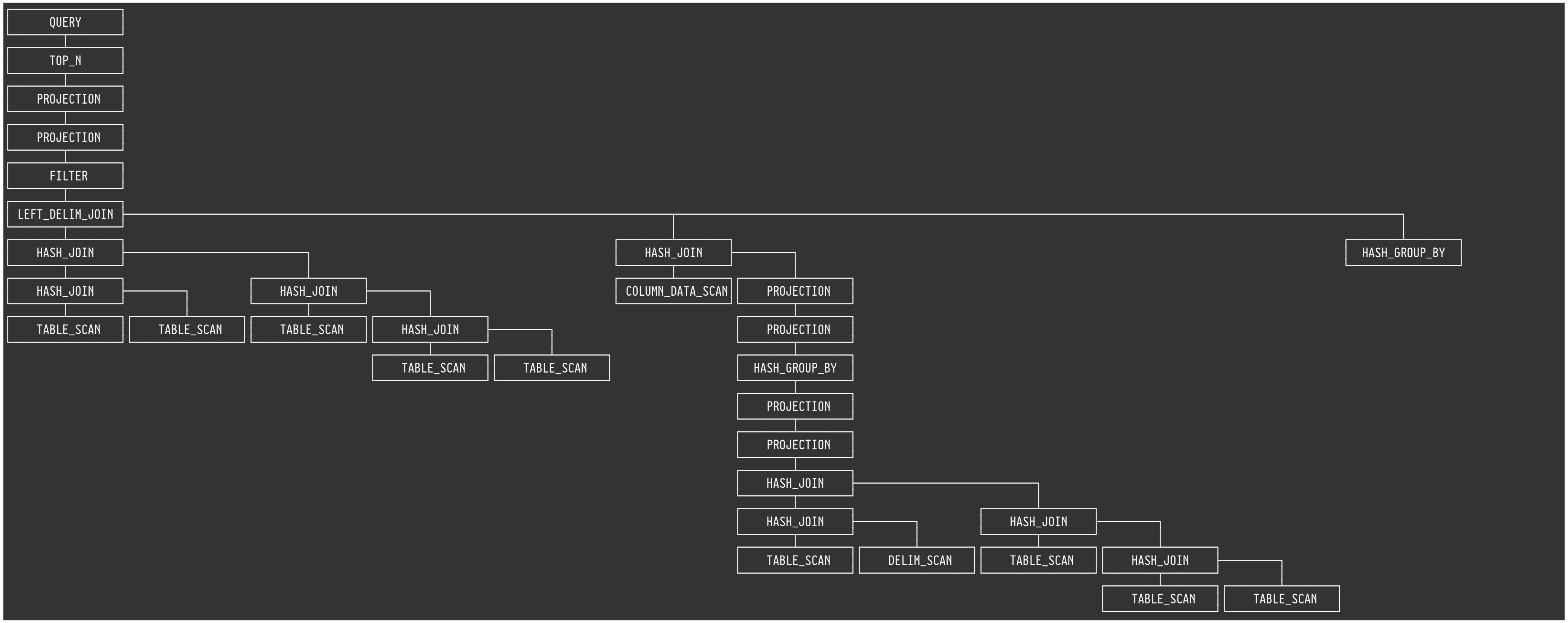
DuckDB translates SQL text into an **execution plan,** a tree-shaped[1] data flow network:

- Nodes ▭ in this network are (physical) **operators,** each of which implement one computation step. Example operators:
  - TABLE_SCAN: reads rows from a table (leaf/source operator),
  - PROJECTION, FILTER: eval (Boolean) expressions, discard rows,
  - HASH_GROUP_BY: hash-based grouped aggregation.
- Edges are directed upwards/leftwards (▭┐ ≡ ▭◂┐) and route rows from child operator(s) to parent operator.
  - DuckDB: **data chunks** of 2048 rows flow jointly along.
- Rows returned by the root operator represent the query result.

[1] This is mostly true. Particular SQL features—e.g., correlated subqueries or common table expressions (CTEs)—can lead to DAG-shaped execution plans.

Execution Plans can be Complex

The execution plans of complex queries are intricate (TPC-H Q2):



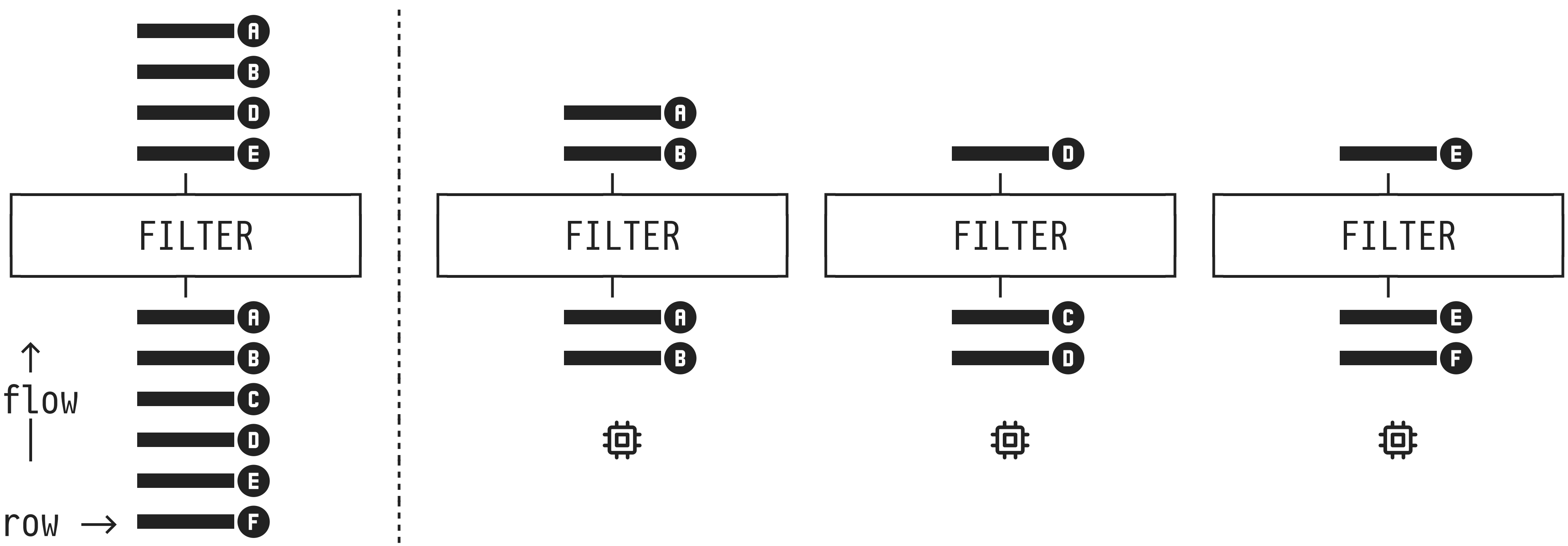
#020

- **Q:** How to **orchestrate the data flow** such that DuckDB
  - does *not need to materialize intermediate results* and
  - and that all *CPU cores* *can equally contribute in parallel*?

2 ¦ Trivially Parallel Operators 1

Operators like FILTER (discard rows based on predicate) or PROJECTION (evaluate scalar expressions) are inherently parallel:

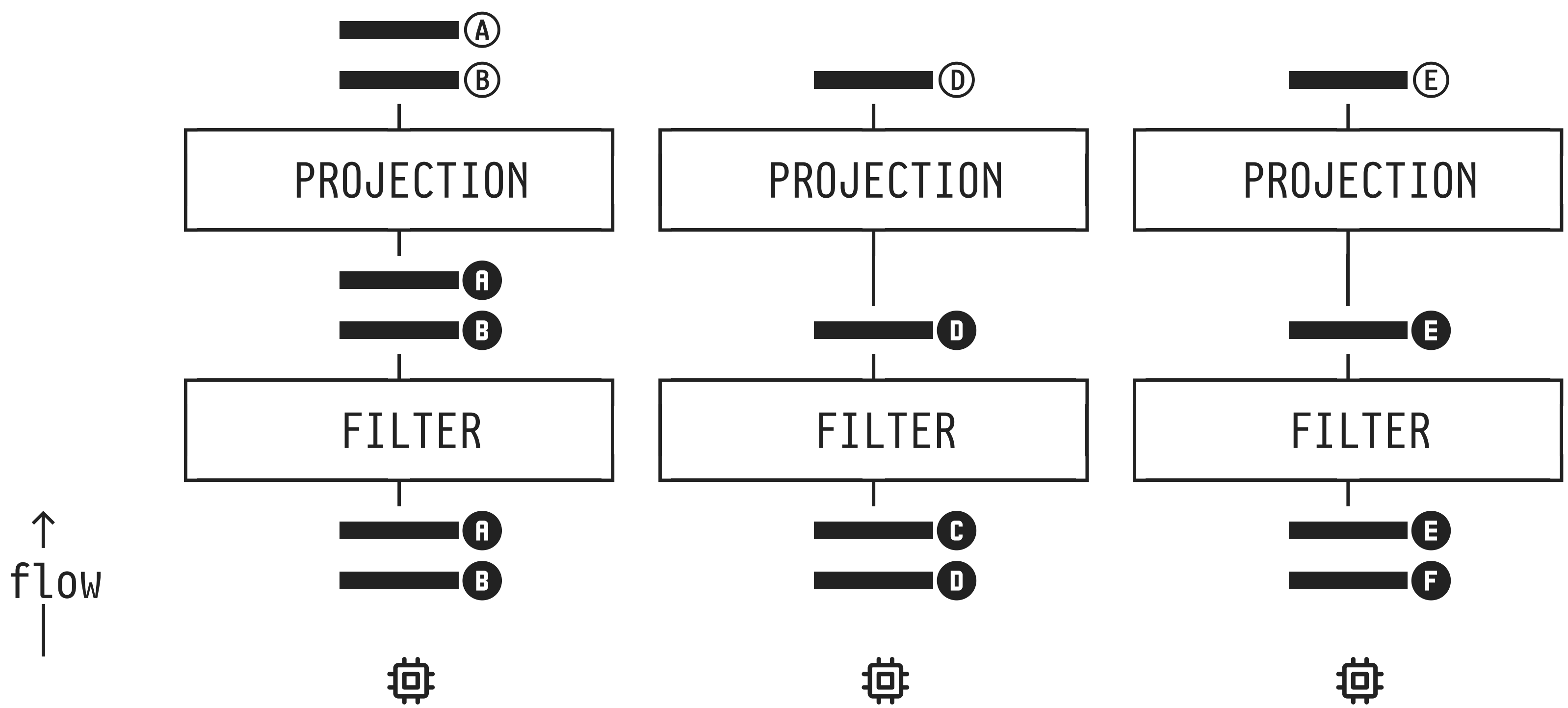
- Each thread operates on a partition of rows. Overall result is the (disjoint) union of all -local results.
- No inter-thread dependencies, operator implementation itself does need *not* to be aware of //ism.



Trivially Parallel Operators 2

Sequences of “PROJECTION-like” operators are assembled into **pipelines:**

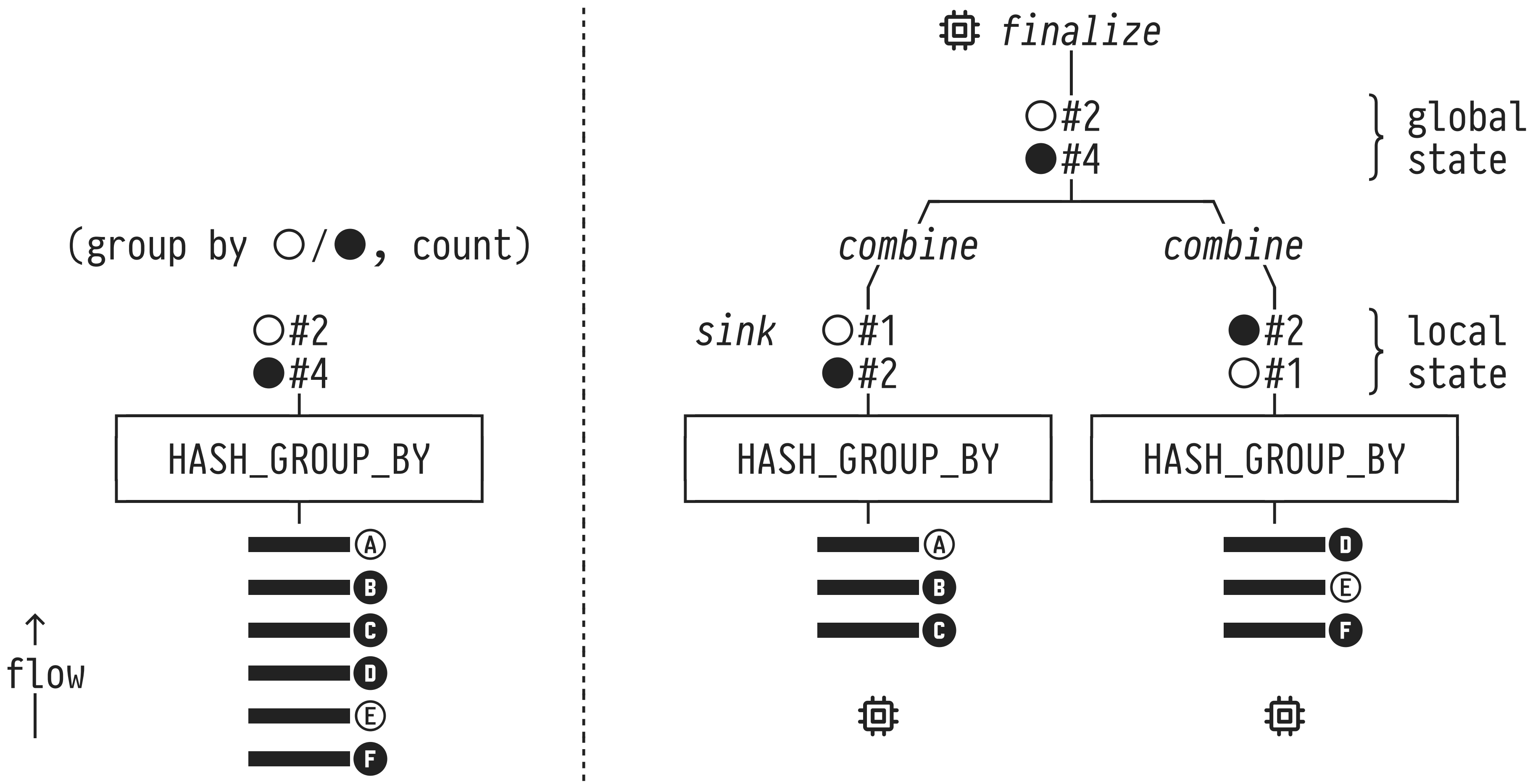
- Each thread/core runs its own instance of the pipeline.
- Data is passed between operators in **chunks** (2048 rows).[2]



[2] Rows that pass FILTER are buffered before they are pushed down the pipeline (do not pass “single-row chunks”—DuckDB aims to pass more than 64 rows). If *all* rows pass, simply pass on the input rows.

3 ¦ Pipeline-Breaking Operators 1

Operators like HASH_GROUP_BY produce a result only once the *entire row input* has been consumed:



Pipeline-Breaking Operators (Sinks) 2

"HASH_GROUP_BY-like" (DB lingo: **sinks**) operator phases:

1. ***Sink***: Each thread ⚙⚙ "sinks" all incoming data into its *local state* (e.g., HASH_GROUP_BY Phase ❶, recall Chapter 03).
2. ***Combine***: Once a thread has read its input, combine local state with the operator's *global state* (HASH_GROUP_BY, Phase ❷).
3. ***Finalize***: *Combine* done, a single thread ⚙ post-processes the global state which is then pushed to the next pipeline.

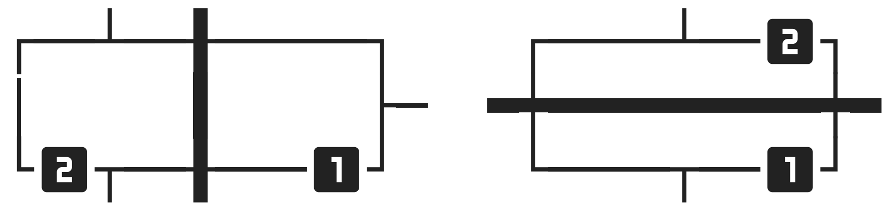
Sinks may only be placed at pipeline ends (**pipeline breakers**).

- Examples of sinks in DuckDB:
    - HASH_GROUP_BY (builds aggregate hash table)
    - Build side of HASH_JOIN (hash table for rhs join input)
    - ORDER_BY (local state: sorted run)
    - UNGROUPED_AGGREGATE (local state: partial aggregate)

4 ┆ Assembling Execution Plans from Pipelines

Execution plans for non-trivial SQL queries are assembled by **stitching multiple pipelines together.**

- In the plans below, operators ▭ are tagged by ❶,❷,… to assign them to a pipeline. Pipeline breakers are marked using ┃.
- Operators like those on the right act like a sink for pipeline ❶. Their output can then be read by the subsequent pipeline ❷.

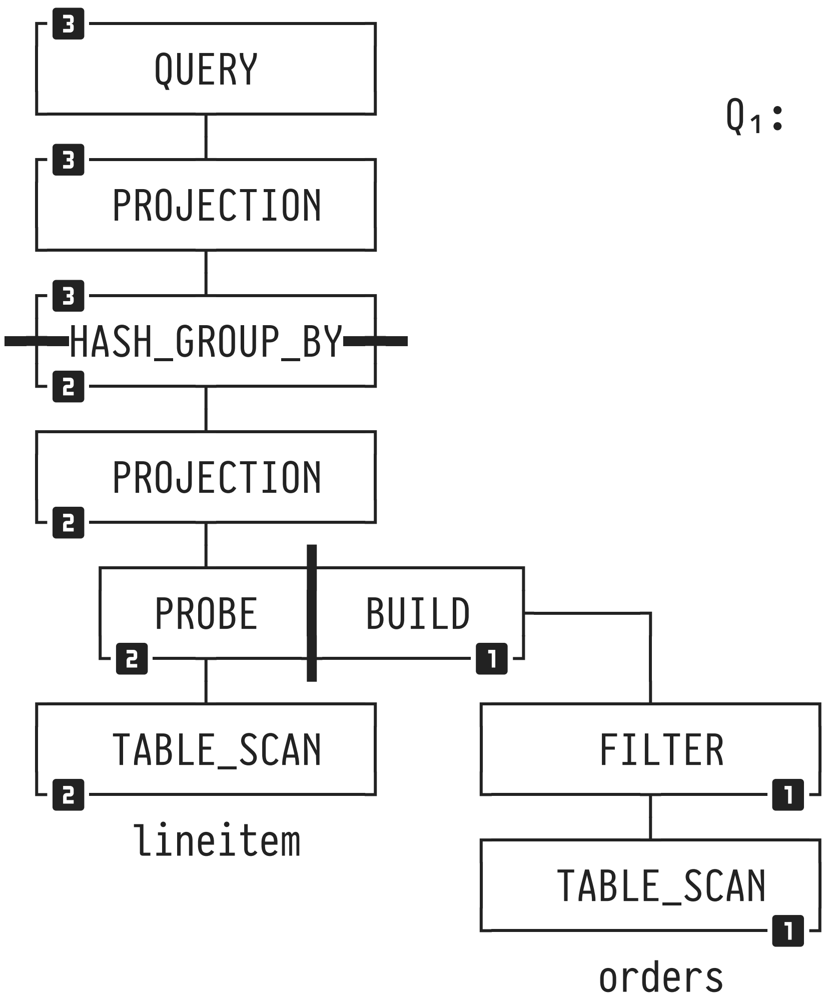


- Such operators introduce **pipeline dependencies:** ❶ < ❷ (pipeline ❶ must finalize before ❷ can read its output).
- Root operator QUERY acts as a sink that
  - collects all incoming rows and
  - constructs the final query result (to be shipped to the caller or displayed by the CLI).

Assembling Execution Plans from Pipelines

Sample query Q_1 over TPC-H data of medium complexity:[3]

#021

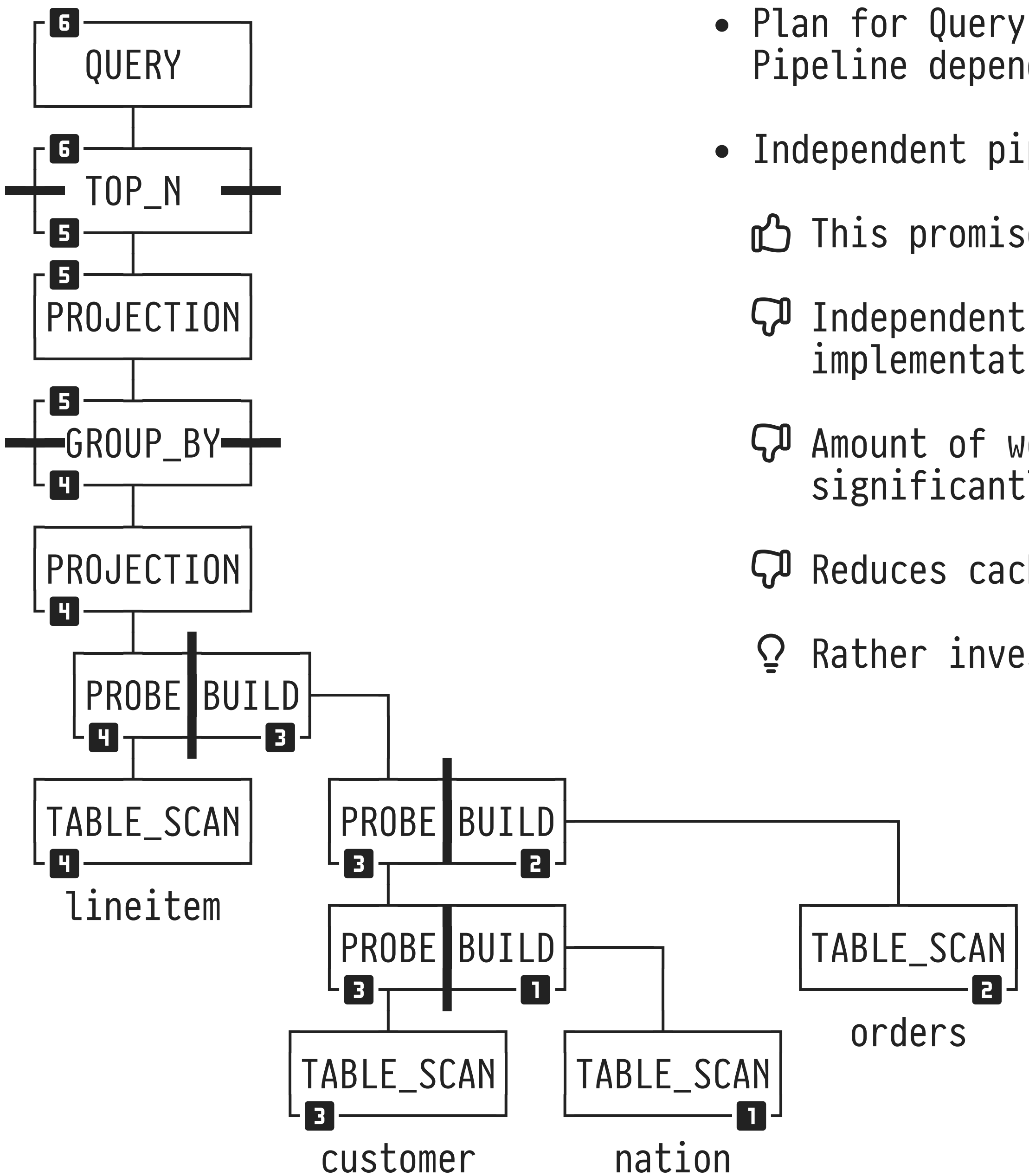



```
Q₁:   SELECT DISTINCT o.o_orderkey AS violation
      FROM   lineitem AS l, orders AS o
      WHERE  l.l_orderkey = o.o_orderkey
      AND    o.o_orderstatus IN ('O', 'F')
      AND    l.l_linestatus <> o.o_orderstatus;
```

- PROBE | BUILD ≡ HASH_JOIN
- For HASH_JOIN, DuckDB chooses the smaller input to be on the BUILD side.
- Pipeline dependencies: 1 < 2 < 3

[3] Checks for violations of a TPC-H constraint: o_orderstatus is set to 'O' ('F') iff all lineitems of this order have l_linestatus set to 'O' ('F'). See the TPC-H benchmark specification, §4.2.3.

Assembling Execution Plans from Pipelines



- Plan for Query Q_3 (TPC-H Query 10)
  Pipeline dependencies: (1‖2)<3<4<5<6.

- Independent pipelines 1‖2 can be run in parallel:

  This promises reduced latency.

  Independent pipelines are rather rare. Does the implementation of the required logic pay off?

  Amount of work in pipelines 1 and 2 may differ significantly.

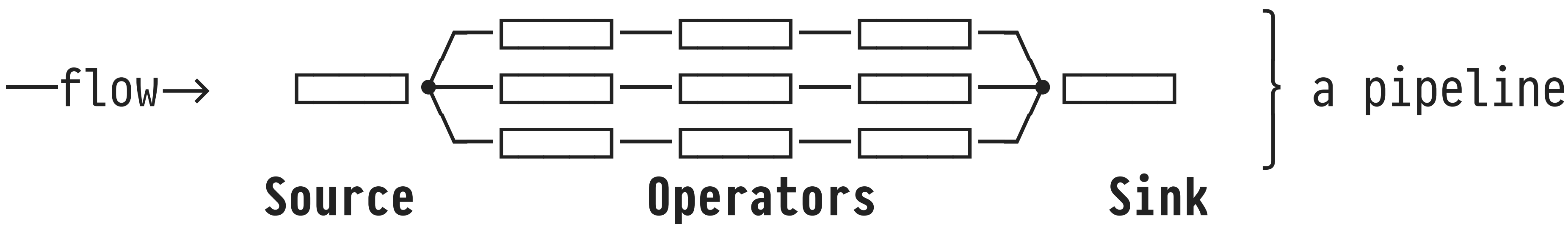
  Reduces cache locality.

  Rather invest more cores in a single pipeline.

5 ┆ Parallelism in a Pipeline

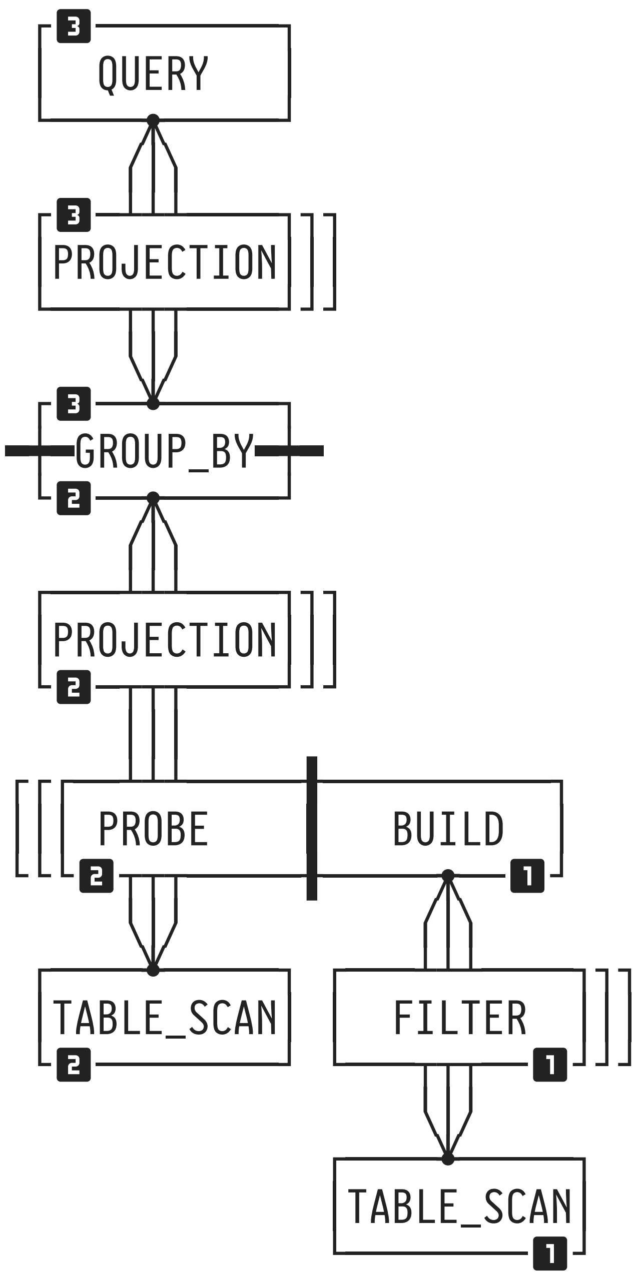
A pipeline in a DuckDB execution consists of three segments.

- **Sources** and **sinks** are parallelism-aware, **operators** need not be:



- Sources know how to partition rows. Examples:
  - COLUMN_DATA_SCAN (literal data), RANGE (row generator)
  - TABLE_SCAN (read base table), CTE_SCAN (read CTE)
  - READ_CSV, PARQUET_SCAN (read external file)
  - Some operators are sinks *and* sources:
    (HASH_GROUP_BY serves rows from its aggregate hash table).

Parallelism in a Pipeline



#022

- Pipelines and parallelism in the plan for Q_1 (TPC-H constraint check, see above).
- Pipeline **sources**:
  1: TABLE_SCAN (orders)
  2: TABLE_SCAN (lineitem)
  3: Phase 2 of HASH_GROUP_BY
- Pipeline **sinks**:
  1: BUILD phase of HASH_JOIN
  2: Phase 1 of HASH_GROUP_BY
  3: QUERY root node
- PROBE side of HASH_JOIN is an operator: multiple threads can peek into the hash table in parallel.

6 ¦ Pipeline Execution

In DuckDB, **pipeline execution** is driven by a core loop (see next slide) that is run by each thread. This loop

- maintains operator state (-local as well as global),
- receives data chunks from operators (or the source) and **pushes** these chunks towards the downstream operator (or the sink),
- contains control flow that detects whether
  1. the source is exhausted or
  2. an operator outputs an empty chunk.

The pipeline driver can

- *cache* rows in case a FILTER or PROBE returns few results (avoid the overhead of passing on tiny or single-row chunks),
- *split* the data flow and pass chunks to multiple consumers (e.g., to share the output of a TABLE_SCAN).

Sketch of DuckDB's Pipeline Driver (Run by each Thread 🖥)[4]

```
var operator[]   pipeline[0…P]           pipeline to execute (operator count P+1)
var state[]      states[0…P]             🖥-local + global operator state
var data_chunk[] intermediates[0…P-1]    rows output by source/operators (intermediate results)

source ← pipeline[0]                     pipeline source and sink
sink   ← pipeline[P]                     (pipeline[1…P-1] are operators)

for i in 0…P:                            initial operator states
⌊ states[i]        ← pipeline[i].GetOperatorState(…)
for i in 0…P-1:                          allocate intermediate chunks
⌊ intermediates[i] ← data_chunk.Initialize(pipeline[i].return_type)

while true:
│ intermediates[0] ← source.GetData(states[0], …)  get next data chunk from 🖥's morsel
│ if (intermediates[0].size = 0)                   no more rows? 1
│ ⌊ return FINISHED                                └ yes, bail out
│
│ reached_sink ← true
│ for i in 1…P-1:                                  execute operators in pipeline order
│ │  intermediates[i] ← pipeline[i].Execute(intermediates[i-1], states[i])
│ │  if (intermediates[i].size = 0)                operator reduced data chunk to size 0? 2
│ │  │ reached_sink ← false                        └ yes, restart at source with next chunk
│ ⌊  ⌊ break
│
│ if reached_sink:                                 did non-empty data chunk reach the sink?
⌊ ⌊ sink.Sink(intermediates[P-1], states[P], …)    └ yes, collect rows at the sink
```

[4] The pseudo code for the pipeline driver is modelled after DuckDB GitHub issue 1583.

Pipeline Driver Manages Control Flow, Operators are Simple

Since control flow is handled by the pipeline driver, source/operator/sink **implementations** turn out to be simple:

1. Build phase of HASH_JOIN (sink):

```
HashJoinBuild.Sink(input, state):
  ht ← state.local_hash_table
  ht.BuildHashTable(input)
```

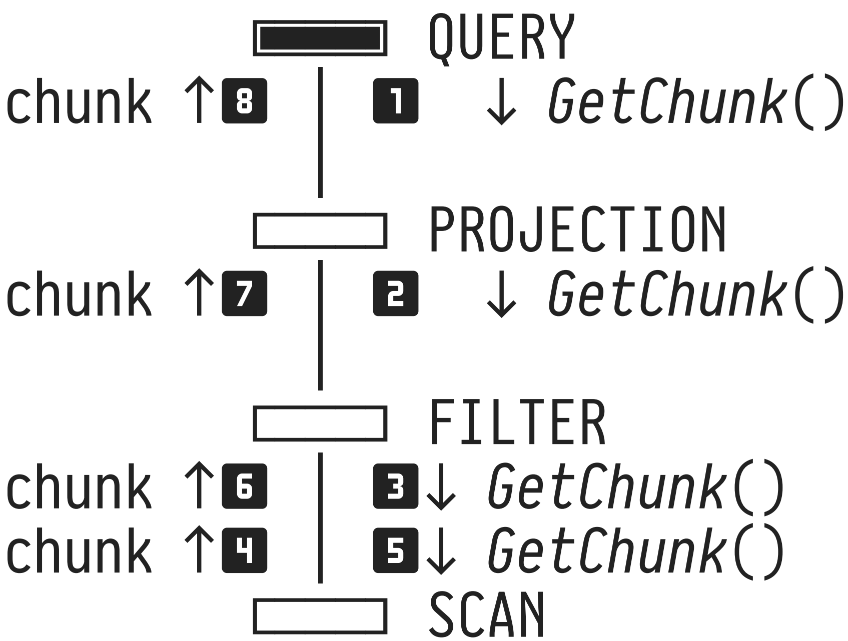
   The *Combine* phase of HASH_JOIN merges the -local hash tables into a global hash table that can be probed against.

2. Probe phase of HASH_JOIN (operator):

```
HashJoinProbe.Execute(input, state):
  ht ← state.global_hash_table
  joined ← ht.ProbeHashTable(input)
  return joined
```

Not DuckDB: Pull-Based Execution (Volcano Iterator Model)

Prior to release 0.3.0 (October 2021), DuckDB implemented **pull-based** operators[5] without a pipeline driver:



1. Start at plan root ▬. Call *GetChunk*() on child operator to request next data chunk.
2. Operators forward *GetChunk*() call down the plan until leaves can return a chunk.
3. Operators process obtained chunk. Call *GetChunk*() again if all of chunk was filtered, otherwise pass chunk upwards.
4. Operators return FINISHED if no more chunks to deliver.

[5] This pull-based plan execution strategy is known as the Volcano iterator model. It is still widely implemented in today's DBMSs. Example: PostgreSQL (operators return a *single row* on each "GetChunk()" call).

Not DuckDB: Pull-Based Execution (Volcano Iterator Model)

Pseudo code for HASH_JOIN in the pull-based Volcano model:

- Control flow handled inside operator.
- Probe + build phases are entangled, hard to parallelize. 👎

```
HashJoin.GetChunk(probe, build):
  state ← this.GetOperatorState(…)
  ht ← state.hash_table

  if not state.build_done:                  are we in the build or probe phase?
    while true:                           ❶ build phase
      chunk ← build.GetChunk(…)              pull next chunk from build side
      if chunk.size = 0:                     no more rows?
        break                                └ yes, hash table complete—now probe
      ht.BuildHashTable(chunk)               insert chunk into hash table
    state.build_done ← true

  do:                                      ❷ probe phase
    chunk ← probe.GetChunk(…)                 pull next chunk from probe side
    if chunk.size = 0:                        no more rows?
      return FINISHED                         └ yes, join complete
    joined ← ht.ProbeHashTable(chunk)         probe chunk against hash table
  while joined.size = 0                       retry if no rows joined
  return joined
```