Design and Implementation of DuckDB Internals

⑧

Query Rewriting and Optimization

April 7, 2026

Torsten Grust
Universität Tübingen, Germany

© 2025–2026 Torsten Grust, U Tübingen, Germany

1 ┆ SQL *Needs* a Query Optimizer

The **query optimizer** is essential in delivering SQL's promise to function as a *declarative* query language:

- SQL queries describe *what* is to be computed, …
- … but do not—in fact: cannot—detail the *how*.

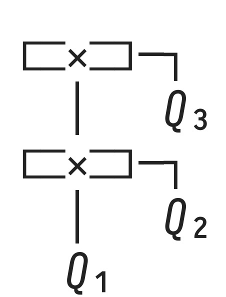
To this end, DuckDB implements a SQL processing regime that the DB community has engineered and refined since the late 1970s:

1. Translate the SQL query text into a **canonical plan** (valid, yet its execution time may well render it unfeasible).
2. Apply equivalence-preserving **plan rewrites** →⁺, each aiming to reduce execution time and/or resource usage.
3. Map **optimized plan**[1] into **physical plan.** Execute.

[1] With `PRAGMA explain_output = 'optimized_only';`, DuckDB's `EXPLAIN` displays that final optimized plan. Additionally use `PRAGMA disable_optimizer;` (see below) to inspect the canonical plan.

Canonical Plans

SQL syntax systematically translates into **canonical plans** in which *plan fragments directly correspond with SQL query clauses*:



- A clause **FROM** $Q_1,\ldots,Q_n$ on n tables/subqueries translates into a left-deep tree of n−1 CROSS_PRODUCT (×) operators.
- The front of this tree exactly mirrors the sequence of the Q_i in **FROM** (no reordering).

- **WHERE** yields FILTER operators that sit *above* the ×-tree.
    - FILTER processes inputs of potentially enormous cardinality.

- **SELECT** translates into a PROJECTION close to the plan root.
    - Unneeded columns are only removed late during execution.

#031

2 ┆ Optimization Passes

DuckDB's query optimizer proceeds in **passes**, each of which consume and produce a valid plan ⁂→⁂.

- Each pass is a “specialist” in one particular plan aspect (e.g., Boolean expressions, join ordering, CTE inlining, …).
- Passes run *once in a pre-determined order*.
    - Predictable optimization effort 👍.
        - Avoid spending dozen of ms to optimize a sub-ms query.
        - Typical pass duration: 10–50 µs, passes overall: 1 ms.
    - Optimization potential may be missed 👎.
        - Passes are *not* iterated (e.g., until fixpoint reached).
        - Pass ordering problem (e.g., statistics lookup may yield EMPTY_RESULT too late for propagation into upstream plan).

Controlling the DuckDB Query Optimizer

```
D FROM duckdb_optimizers();
┌─────────────────────┐
│        name         │
├─────────────────────┤
│ expression_rewriter │   -- DuckDB v1.5 implements 30+ passes
│ filter_pullup       │   -- (listed here in arbitrary order)
│         ⋮           │
│ join_elimination    │
│ window_self_join    │
└─────────────────────┘
```

- Disable and re-enable the entire query optimizer:

```
D PRAGMA disable_optimizer;   --  ⚠ execute canonical plans
D PRAGMA enable_optimizer;
```

- Disable and re-enable all selected optimization passes:

```
D SET disabled_optimizers = 'filter_pullup, join_order';
D SET disabled_optimizers = '';
```

Order of Optimization Passes in DuckDB (version 1.5)

#	Pass	Specializes in…
1	expression_rewriter	simplifying/evaluating scalar expressions
2	cte_inlining	choosing between inlining or materializing CTEs
3	sum_rewriter	rewriting aggregates of the form **sum**(*e* + *const*)
4+5	filter_pullup+pushdown	moving FILTER operators downstream/upstream
6	cte_filter_pusher	moving FILTERs into materialized CTEs
7	regex_range	turning reg.exp. matches into range predicates
8	in_clause	turning **IN** (…) into regular filter or join
9	deliminator	removing redundant DELIM_JOINs/DELIM_GETs
11	empty_result_pullup	simplifying plans in presence of empty results
12	window_self_join	turning window computations into self-joins
13	join_order	turning × into joins, decide join order
14	join_elimination	detecting and removing redundant (semi-)joins
16	unused_columns	identifying unused columns to make scans narrower
17	duplicate_groups	removing redundant grouping criteria
18	common_subexpressions	evaluating common scalar subexpressions only once
19	column_lifetime	introducing PROJECTIONs to remove unused columns
20	build_side_probe_side	placing join inputs on lhs (build)/rhs (probe)
21	common_subplan	converting common subplans into materialized CTEs
22	limit_pushdown	moving LIMIT below PROJECTIONs (limit early)
23	row_group_pruner	identifying row groups that need not be scanned
25	top_n	merging ORDER_BY and LIMIT into TOP_N
26	late_materialization	narrowing inputs, joining in columns late in plans
27	statistics_propagation	using column statistics to simplify plans
30	reorder_filter	evaluating cheap clauses first in con/disjunctions
31	join_filter_pushdown	filtering probe side based on build side values

💡 Some **scalar expressions** e (e.g., in **SELECT**/**WHERE** clauses) can be **statically evaluated**—maybe partially only—*without* base data access.

- Savings are minimal for any individual evaluation of e, but e may be evaluated millions of times during plan execution.
- Simplifcation may reveal opportunity for further plan rewrites.
- Implemented in DuckDB:
  - **Arithmetic/Boolean simplification:** #033
    - *const* - *const*, e + 0, e // 1, e * NULL.
    - *const* ∧ *const*, e ∨ false, NOT ($e_1 < e_2$), distributivity.
  - **CASE simplifcation:** constant guards, branch pruning.
  - **LIKE/regular expresssion match simplification:**
    - Rewrite **LIKE** into range or suffix/contains predicates.
    - Trade regexp_matches for the computationally lighter **LIKE**.

DuckDB evaluates conjunctions[2] e_1 AND e_2 AND … AND e_n left to right using Boolean shortcut.

Reorder clauses e_i from cheap to costly.

- Heuristic expression cost (read $\prec$ as "*is cheaper to evaluate*"):
  - Ops: + $\prec$ * $\prec$ / $\prec$ round $\prec\prec$ ~~. Types: int $\prec$ double $\prec$ text.
  - $cost(e_1 \circledast e_2) = cost(e_1) + cost(e_2) + cost(\circledast)$.

$$cost(e :: \tau) = \begin{cases} cost(e) & \text{if } e \text{ has type } \tau \\ cost(e) + 200 & \text{if } e :: \text{text or if } \tau = \text{text} \\ cost(e) + 5 & \text{otherwise} \end{cases}$$

- Do not reorder if any e_i can possibly fail (e.g., cast to int): moving e_i rightwards may hide an observable side effect.

[2] This also applies to disjunctions (OR).

#034

DuckDB maintains basic **statistics for base table columns.**

💡 Propagate statistics through plan operators to derive min/max value ranges and constraints *without* access to the base data.

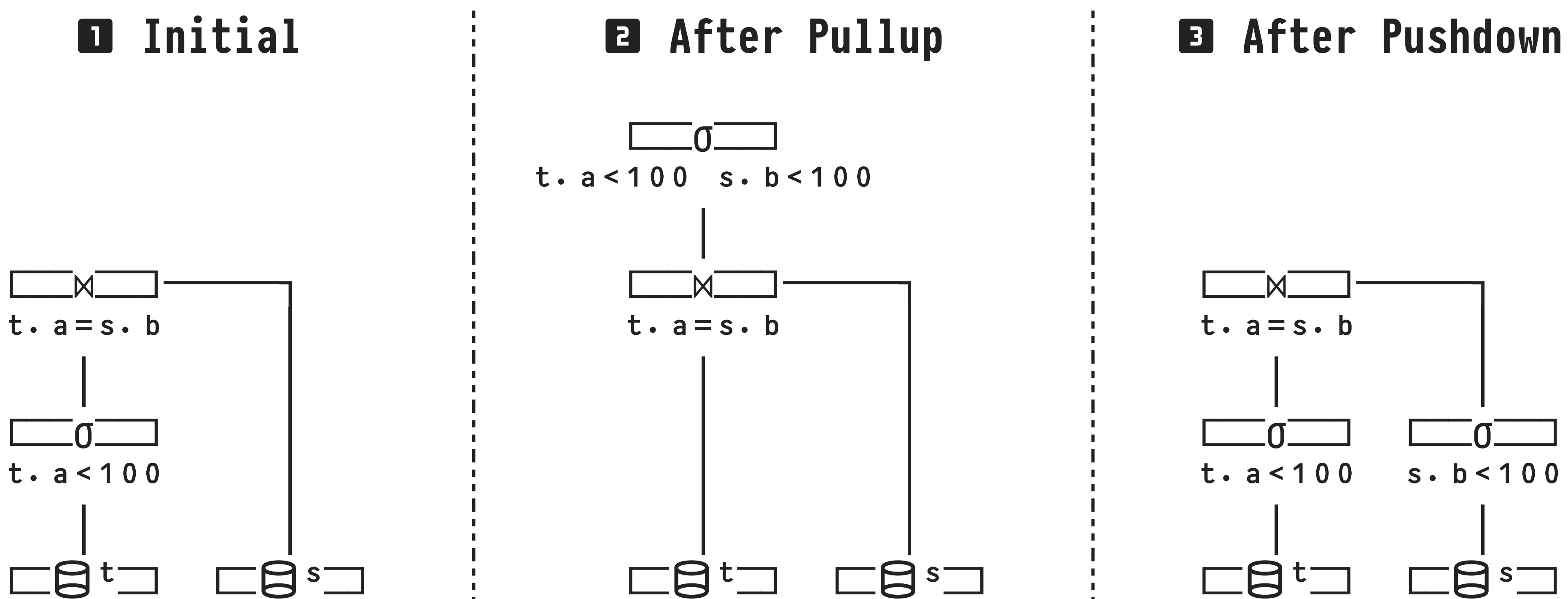
	Propagated statistics (excerpt)
intermediate tables	• estimated cardinality • maximum cardinality (helps allocation)
numeric columns	• min/max value (may be unknown or uninteresting if entire domain of column type is covered) • is constant? • contains NULL?
text columns	• min/max value (first 8 characters) • maximum string length • contains Unicode (non-ASCII) characters?

- Propagated statistics help to identify FILTER predicates that will be *always true* (remove) or *always false* (EMPTY_RESULT).

#035

Move FILTERs upstream (**“filter pushdown”** towards/into base table scans) to reduce intermediate table cardinalities early.

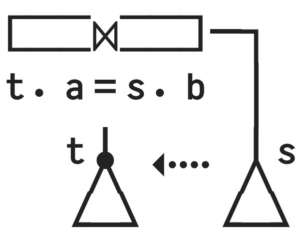
- DuckDB additionally implements **filter pullup** (propagate restrictions through equality predicates):[3]



[3] Here and in the sequel let us adopt common operator abbreviations: ▭⛁▭: SCAN, ▭σ▭: FILTER, ▭⋈▭: JOIN.

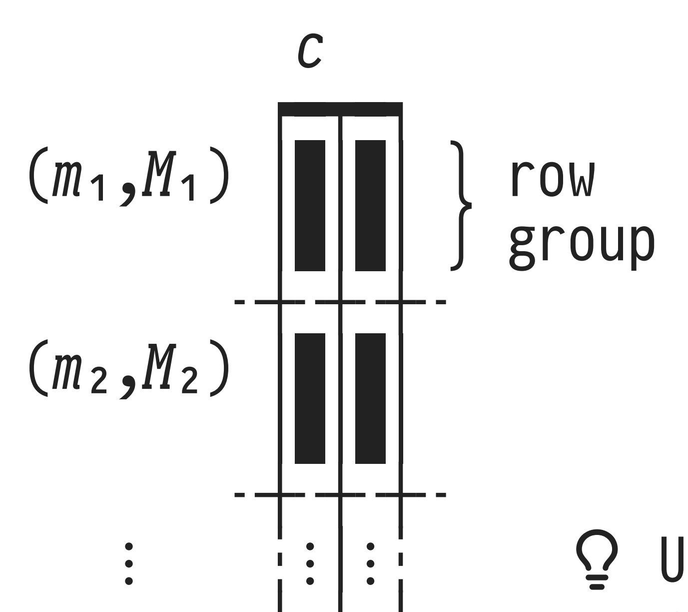
Join execution proceeds in two (alternating) phases:

1. Read rows from build/outer input s.
   💡 Record min/max values m^b/M^b in column $s.b$.
2. While reading rows from the probe/inner input t,
   💡 use m^b/M^b on column $t.a$ to pre-filter rows.



- This **join filter pushdown** ("*sideways information passing*" ⇠) is a *dynamic* optimization performed *during* plan execution: only then are m^b/M^b known.[4]
  - During query optimization, pass join_filter_pushdown only identifies the scan • which will apply the dynamic predicate.
  - ⚠ Only EXPLAIN **ANALYZE** will reveal this optimization.

[4] If the distint values in $s.b$ are $\{x_1,\ldots,x_n\}$ (with $n \leqslant$ dynamic_or_filter_threshold, default 50), the dynamic predicate will be IN $(x_1,\ldots,x_n)$ to filter t precisely.



- DuckDB can **scan row groups in an order** defined by the zonemap entries (m_i,M_i):
  - Scan row groups by ascending m_i (min).
  - Scan row groups by descending M_i (max).

Use these scan orders to limit the row groups read to process TOP_N (**ORDER BY** *c* + **LIMIT** *N*).

```
FROM t ORDER BY c DESC LIMIT N -- rows with N largest c
```

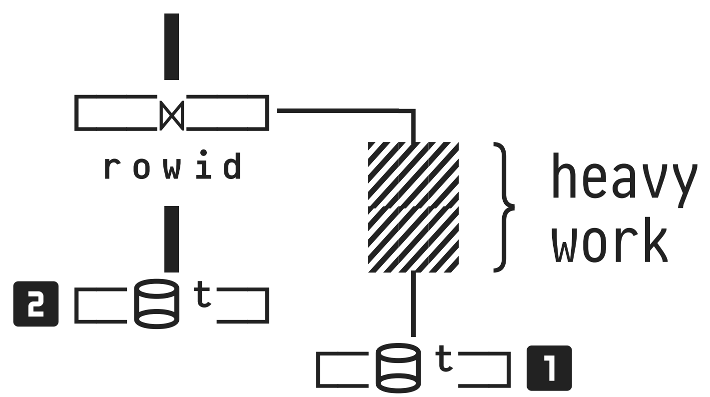
Scan row groups in descending M_i (max) order. **Stop criterion:**

$(m_i,M_i) \cap (m_j,M_j) = \emptyset$?
(no overlap of *c* values between row groups?)

- yes, stop scanning row groups once $R \geq N$ rows have been read
- no, stop after at most N row groups

Execution benefits if rows are kept *narrow* as long as possible:

1. **Push down projections** to scan relevant columns (say, for filtering, grouping, sorting, see ▨ below) only.
2. 💡 *Late in the plan* use **row IDs to join in payload columns** to build (a potentially wide) query result.



- TABLE_SCAN 1 performs projections, leaving only columns relevant in ▨.
- TABLE_SCAN 2 receives the list of relevant row IDs (“rowid pushdown”) to access only few yet wide rows.

rows: | narrow, ▌ wide

- ▭⋉▭ is a left semi-join on row IDs.

#038

💡 Use associativity and commutativity of + to save sum aggregate evaluation from the repeated addition of constant values:

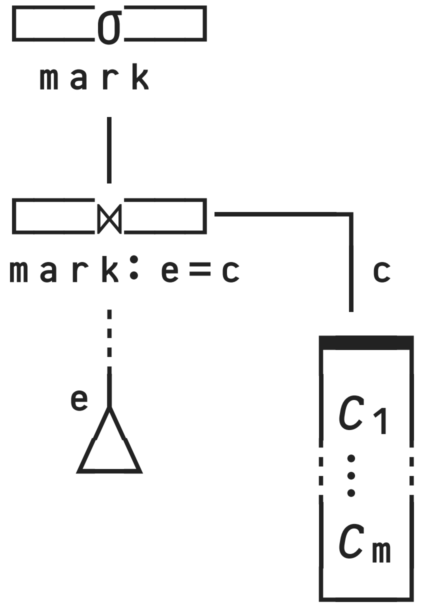
$$\textbf{sum}(e + const) \;\rightarrow\; \textbf{sum}(e) + const \times \textbf{count}(e)$$

- **Notes:**
  - In IEEE754 floating point arithmetic, + is not associative. DuckDB applies pass sum_rewriter only if *e* and *const* are integer-typed.
  - Both sides are equivalent also if *e* may be NULL. (In SQL, aggregates sum and count ignore NULL values.)
  - Expression *e* is only evaluated once (by a dedicated PROJECTION operator) and then re-used in both aggregates.

#039

Naively evaluating predicates like e IN $(c_1,\dots,c_m)$ may be expensive: $O(n \times m)$ if the predicate is tested for n rows.

- 💡 If m is large, rewrite predicate into an $O(n + m)$ HASH_JOIN:
  - Build side: single-column table of m rows carrying the c_i.
  - Probe side: provides n bindings for e.



COLUMN_DATA_SCAN

#040

- Optimizer creates a COLUMN_DATA_SCAN pipeline source that delivers the c_i.
- Since e and the c_i are irrelevant once the predicate is evaluated, ⋈ is a **MARK** join:
  - Generate a Boolean *mark* that holds the result of the equality comparison.
  - Process *mark* as needed (here: σ).

SQL **windows** establish frames of related rows[5] in a table *t* that a function *f* can then process jointly. 💡 For simple frame specs and *f*, the same operation can be expressed by a self-join of *t*:

```
❶ SELECT agg(e) OVER (
              PARTITION BY c)
   FROM t
```

```
❷ WITH frame(part,agg) AS (
     SELECT c AS part, agg(e)
     FROM t
     GROUP BY c
   )
   SELECT agg
   FROM t SEMI JOIN frame
     ON (c IS NOT DISTINCT FROM part)
```

- Pass window_self_join rewrites ❶ into ❷ on the plan level for
  1. window functions *f* that are SQL aggregates and
  2. frames that are exclusively specified by PARTITION BY.

[5] Rows relate based on position in an ordering (as in ORDER BY … ROWS BETWEEN … AND …) and/or inside partitions (PARTITION BY …).

12 ┊ Inlining or Materializing CTEs? ┊ cte_inlining

Common table expressions (CTEs) may be referred multiple times.
Shall the CTE be **materialized once or inlined** at every reference?

DuckDB uses a decision tree to answer that question:


```
              #refs
            1/ 0|  \1+
 MATERIALIZED?  ×   MATERIALIZED?
  n/  ∅|   \y         n/  ∅|   \y
inl   inl   mat      inl   |    mat
                       volatile?
                        n/   \y
                      agg?    mat
                     n/   \y
                 empty?   mat
                 n/   \y
                mat   inl
```

```
    #refs: # of references to CTE
        ×: CTE removed from plan
  MAT...?: CTE def annotated by
           [NOT] MATERIALIZED?
volatile?: does CTE use functions
           like random()?
     agg?: will the CTE result
           likely be small?
   empty?: will the CTE result be
           empty?
  inl/mat: inline/materialize CTE
```

SQL queries may exhibit query fragments that are syntactically similar (albeit not identical). Can evaluation be shared?

1. 💡 Identify **identical plan—not query—fragments** (*subplans*),
2. evaluate subplan once and materialize its result,
3. replace each fragment by a scan of the materialized result.

- Pass common_subplan builds on DuckDB's CTE infrastructure:
  - Use plan operator CTE to evaluate and materialize the subplan (under fabricated CTE name __common_subplan_‹*n*›).
  - Use plan operator CTE_SCAN (multiple times) to read the materialized result.
  - Do not share subplans that call on volatile functions.

#043

Join (operator ▭⋈▭, algebraic: $S \bowtie_p T$) is fundamental in SQL and query optimization. Remains a hot topic in database research.

DuckDB's join optimizer considers a whole range of aspects:

1. Join predicates p come in many forms. When building the physical plan, **pick an implementation** for ▭⋈▭ that fits the given predicate(s).
2. Join can reduce or potentially *multiply* the **cardinalities** of its input tables: $0 \leqslant |S \bowtie T| \leqslant |S| \times |T|$.
3. Join operates on two tables. In an n-table query, it is essential to select an efficient **order of the** n-1 **joins**.
4. $S \bowtie T$ is commutative. In most join *implementations*, however, S and T assume **assymetric** roles. Choose $S \bowtie T$ or $T \bowtie S$ wisely.

Join Predicates Determine Implementations of ⋈ in DuckDB

Inspect join predicate *p* to select a fitting join implementation:

```
                pred?
              n/   \y
   CROSS_PRODUCT    =?
                  n/  \y
              complex?  HASH
  ----------- n/ --┐ \y
       ≤5 rows?  ┆  NESTED_LOOP
       n/    \y  └---------
  ≥1K rows?  NESTED_LOOP
   n/    \y
MERGE  2×<=?
       n/   \y
   MERGE    IE
```

```
   pred?: any predicate at all?
      =?: any equality? (equi-join)
complex?: complex predicate? (~~)
≤5 rows?: nested_loop_join_threshold
≥1K rows?: merge_join_threshold
   2×=?: two or more inequalities
```

- Subtree below ---- exlusively concerned with inequalities.
- Equi-joins are predominant in typical SQL workloads ⇒ HASH_JOIN is DuckDB's workhorse.

Join Implementations for $S \bowtie_p T$ in DuckDB

NESTED_LOOP_JOIN

```
T_m ← materialize(T)
for s ∈ S                         S: outer
│ for t ∈ T_m                     T: inner
│ │ if p(s,t)
└ └ └ emit s⧺t  ⧺: output col.s
```

- **BLOCKWISE_NL_JOIN:** process chunks of S/T at a time.

HASH_JOIN

#044

```
H ← table(hash<p>,T)       T: build
for s ∈ S                  S: probe
│ h ← hash<p>(s)
│ for t ∈ H(h)   scan bucket in H
│ │ if p(s,t)
└ └ └ emit s⧺t
```

PIECEWISE_MERGE_JOIN

```
T_s ← sort<p>(T)
for chunk C ∈ S
│ C_s ← sort<p>(C)
│ for (s,t) ∈ merge(T_s,C_s)
│ │ if p(s,t)
└ └ └ emit s⧺t  scan T_s,C_s once
                top-down
```

IE_JOIN ($p \equiv S_1.aS_2.a \wedge S_1.b > S_2.b$)

```
L_1 ← sort<a>(S)         <: ascending
L_2 ← sort<b>(S)         >: ascending
for j ∈ 1…n
│ s ← L_2[j]      i/j: pos of row s
│ i ← s @ L_1              in L_1/L_2
│ for t ∈ L_1[1…i-1] ∩ L_2[j+1…n]
└ └ emit s⧺t
```

Join Tree Shapes

A SQL join query with n+1 tables requires n binary joins. ⋈ is *associative*: the number of possible **join trees** grows quickly.

k joins ⋈ n-k-1 joins

\# of join tree shapes:

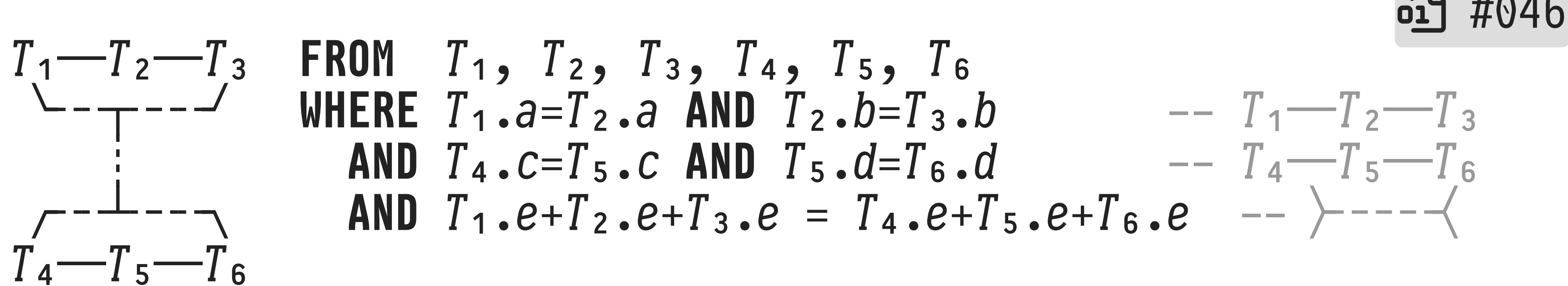
$$C_n = \sum_{k=0}^{n-1} C_k \times C_{n-k-1}$$

n (# of ⋈)	C_n (# of join tree shapes)[6]	
0	1	single-table query
1	1	two tables, single join
2	$C_0 \times C_1 + C_1 \times C_0 = 1 + 1 = 2$	or
3	5	
4	14	
5	42	
10	16,796	

It is not reasonable to fully explore this join tree search space.

[6] C_n is the nth Catalan number. Closed form: $C_n = (2n)! / ((n+1)! \times n!)$.

1. From the SQL join query, build a **join hypergraph** in which n tables are connected by join predicates:



2. Use dynamic programming[7] to iteratively build join trees of minimal *cost*() with 2,3,…,n tables. Build join trees of size k from memoized trees of size i and k-i.
   - Disregard non-connected tables to avoid cross products.
   - $cost(S \bowtie T) \stackrel{\text{def}}{=} |S| + |T| + |S \bowtie T|$. (all estimated)

[7] DuckDB's join_order optimizer is modelled after the hypergraph-aware dynamic programming algorithm *DPhyp* described in G. Moerkotte, T. Neumann, *"Dynamic Programming Strikes Back"*, SIGMOD, Vancouver, Canada, 2008.

Choose $S \bowtie T$ vs $T \bowtie S$ ┆ build_side_probe_side

$S \bowtie T$ is commutative but the chosen operators ⊏⋈⊐ do *not* treat the inputs symmetrically.

lhs — **probe** outer

- Larger inputs
- Simple subplans (scans)
- Inputs that pass many columns/row IDs downstream

rhs — **build** inner

- Smaller inputs S
- Complex subplans (other joins)

- Heuristic S (smaller inputs on the *rhs* of ⊏⋈⊐):
  - HASH_JOIN: Build and materialize small(er) hash table H.
  - NESTED_LOOP_JOIN: Use less space to materialize inner T_m.
  - PIECEWISE_MERGE_JOIN: Use less space to hold sorted T_s.

Wherever a SQL query expects a scalar or tabular value, a **nested subquery** (enclosed in (...)) may be used to compute that value:

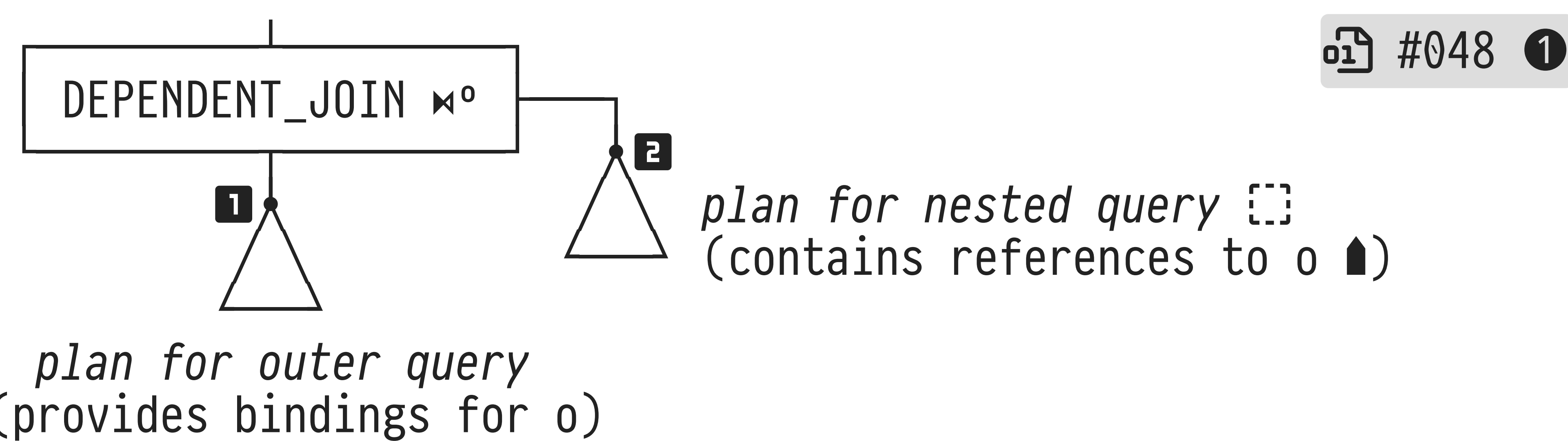
```
SELECT o.o_orderkey, o.o_orderdate, l.l_linenumber
FROM   orders AS o, lineitem AS l
WHERE  o.o_orderkey = l.l_orderkey
AND    l.l_extendedprice = (
                            SELECT min(i.l_extendedprice)
                            FROM   lineitem AS i  ▼
                            WHERE  i.l_orderkey = o.o_orderkey );
```

Q_1: All TPC-H orders along with (the linenumber of) their cheapest lineitem

- Subquery ⬚ is **correlated** (row variable o occurs free at ▼).
- SQL semantics: Re-evaluate query in ⬚ for each o,l pair.
- For sizeable input tables, a naive nested-loop evaluation strategy will lead to unacceptable query run times. 👎

Correlation Leads to Dependent Joins

In initial/non-optimized query plans, correlation is represented by the DEPENDENT_JOIN operator (symbol: ⋈ᵒ):



- Nested loop semantics: Evaluate 2 for each row produced by 1.
- **Query decorrelation:** Remove *all* occurrences of DEPENDENT_JOIN,[8] rewrite plans 1 and 2 such that both are evaluated only once.

[8] In fact, DuckDB does not implement DEPENDENT_JOIN as a *physical* plan operator. Query decorrelation thus is an absolute must in DuckDB.

A correlated subquery acts like a (table-valued) **function** whose parameters are the free row variable references—in Q_1: ⬚(o). Do not compute that function for the same parameters more than once.

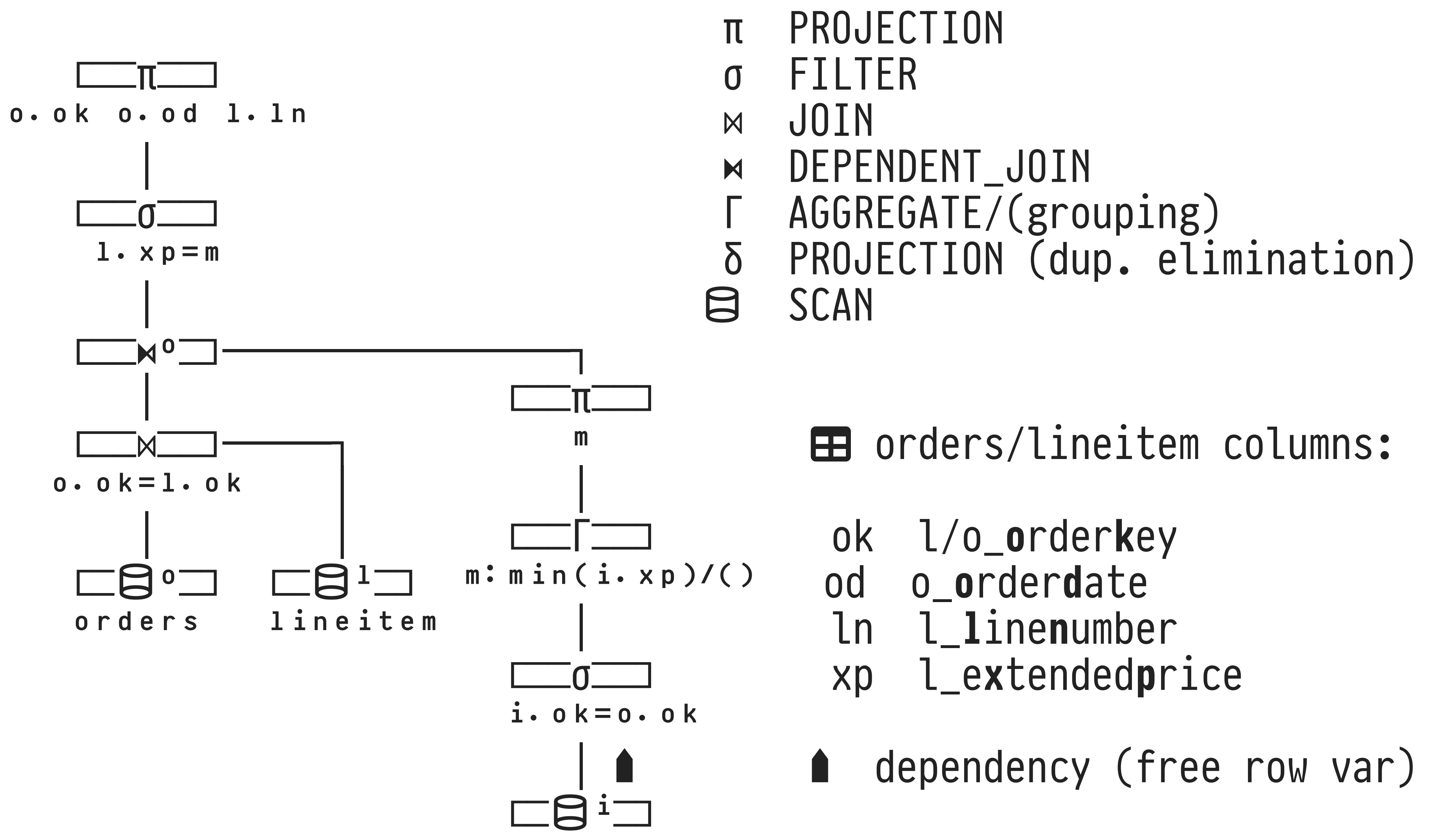
Query decorrelation:

1. Compute the duplicate-free set D of distinct parameters.
2. Evaluate the subquery ⬚ for *all parmeters* $d \in D$ *in one go*, yielding a lookup table T of (d, ⬚(d)) pairs.
3. Evaluate the outer query by joining with lookup table T.

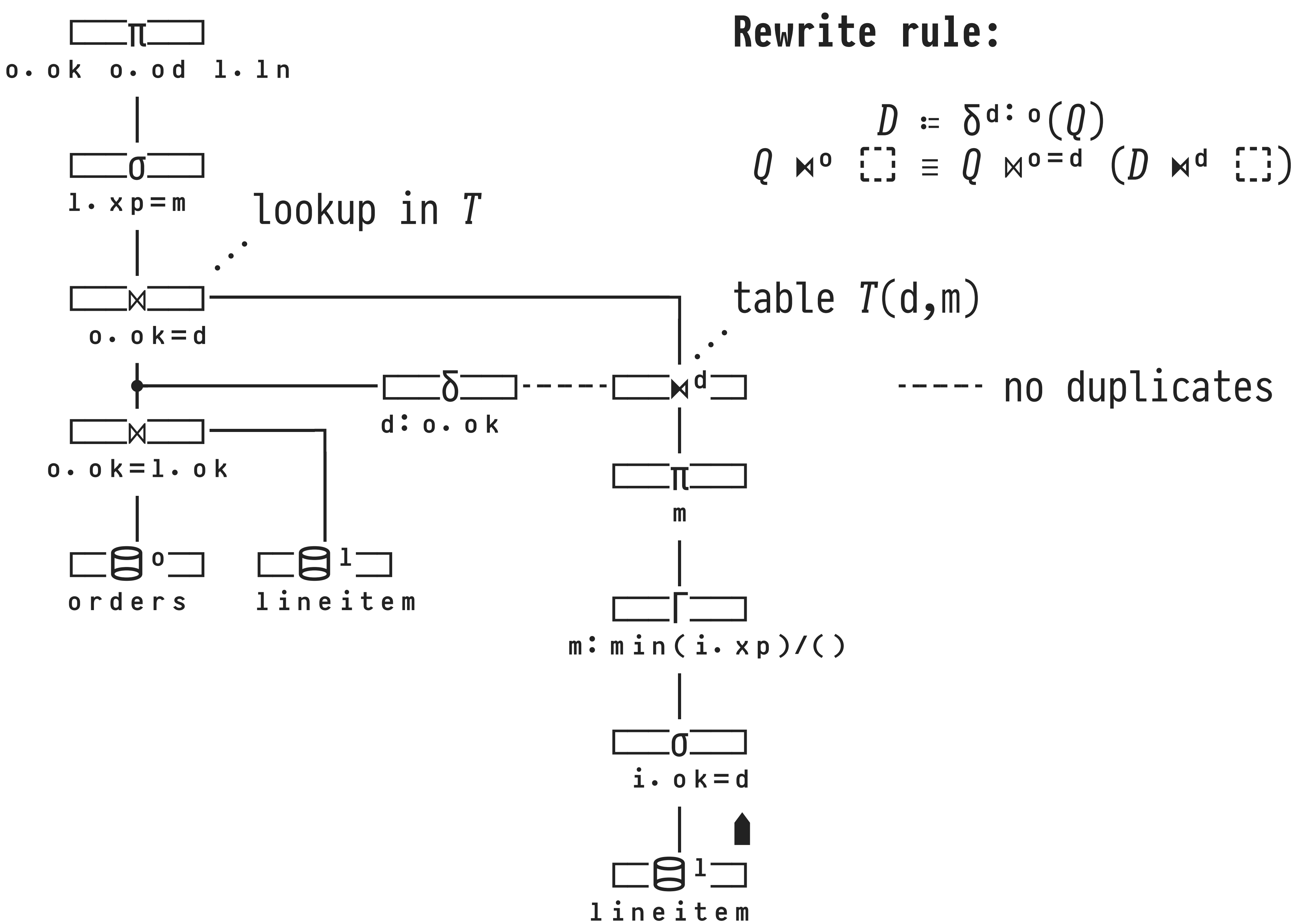
Thomas Neumann and Alfons Kemper of TU Munich[9] implement this idea via systematic rewrites that push DEPENDENT_JOIN ⧓ down into a query plan until it can be trivially replaced by regular JOIN ⋈.

[9] T. Neumann, A. Kemper, "*Unnesting Arbitrary Queries*", Proc. of the BTW Conference, Hamburg, Germany, 2015.

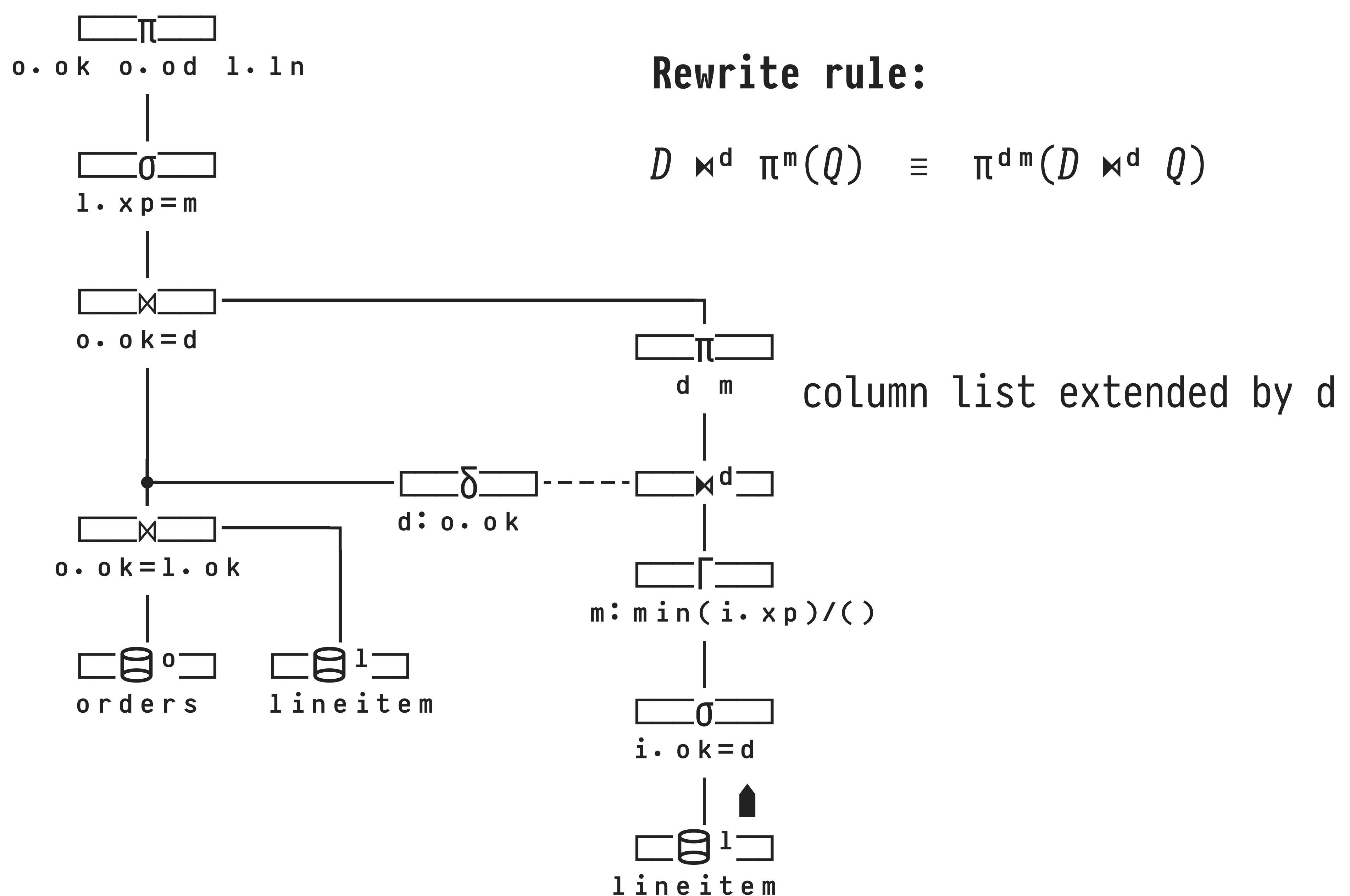
Query Decorrelation 0 (Initial Plan)



Query Decorrelation 1 (Compute Parameter Set *D*)



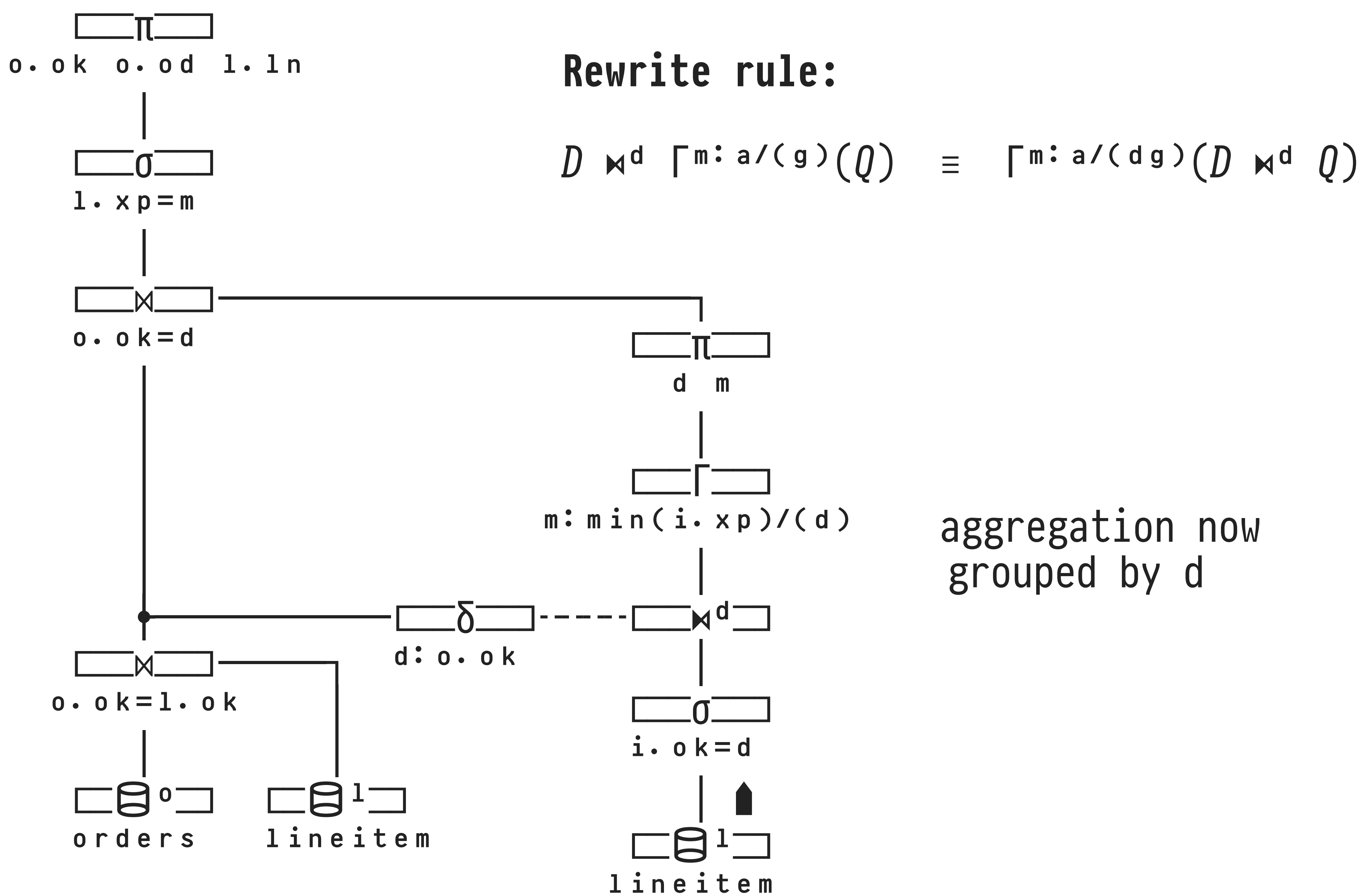
Query Decorrelation ❷ (Push ⋉ Down Through π)



Rewrite rule:

$$D \ltimes^{d} \pi^{m}(Q) \equiv \pi^{dm}(D \ltimes^{d} Q)$$

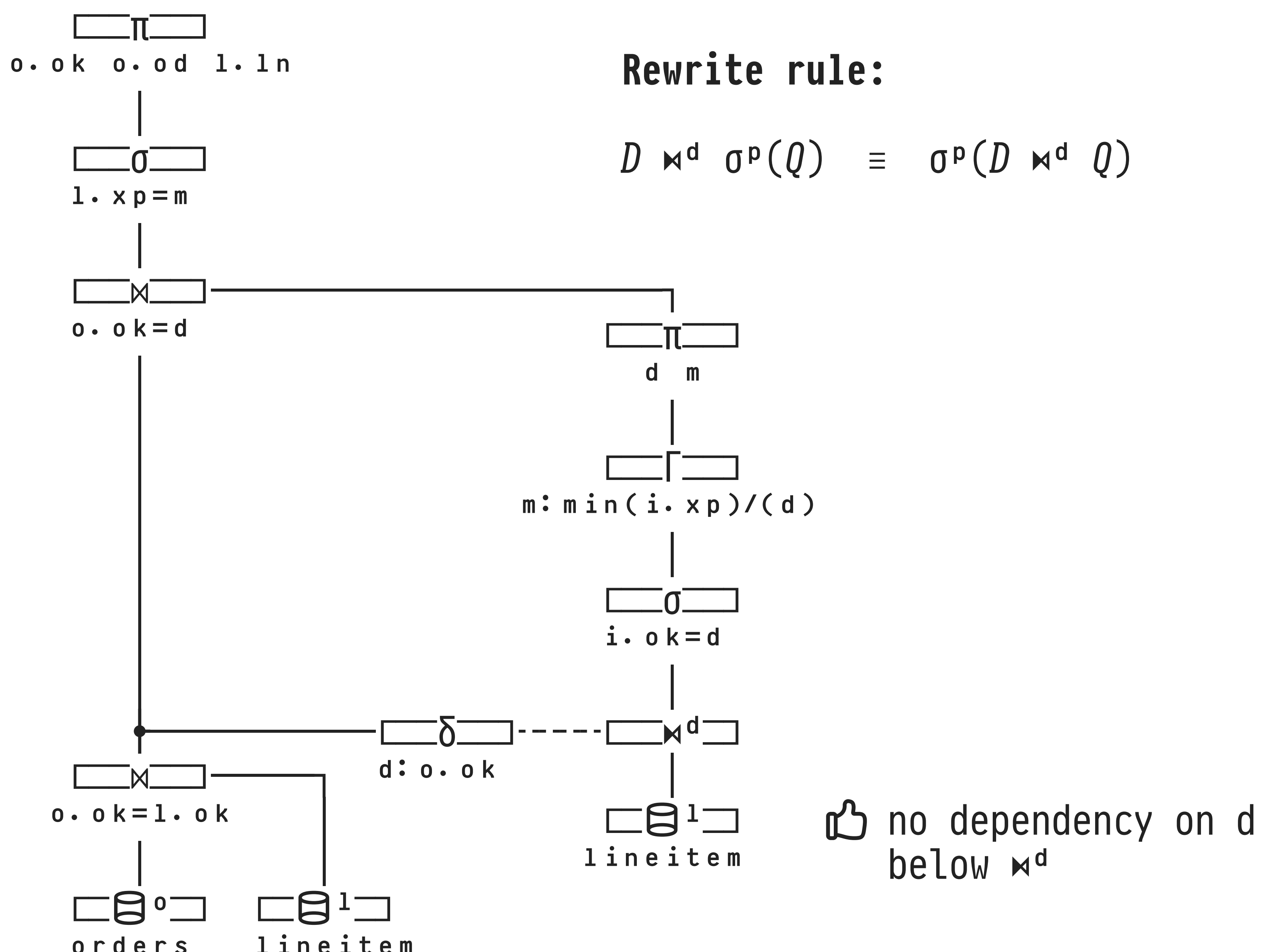
Query Decorrelation 3 (Push ⋉ Down Through Γ)



Query Decorrelation 4 (Push ⋈ Down Through σ)

Rewrite rule:

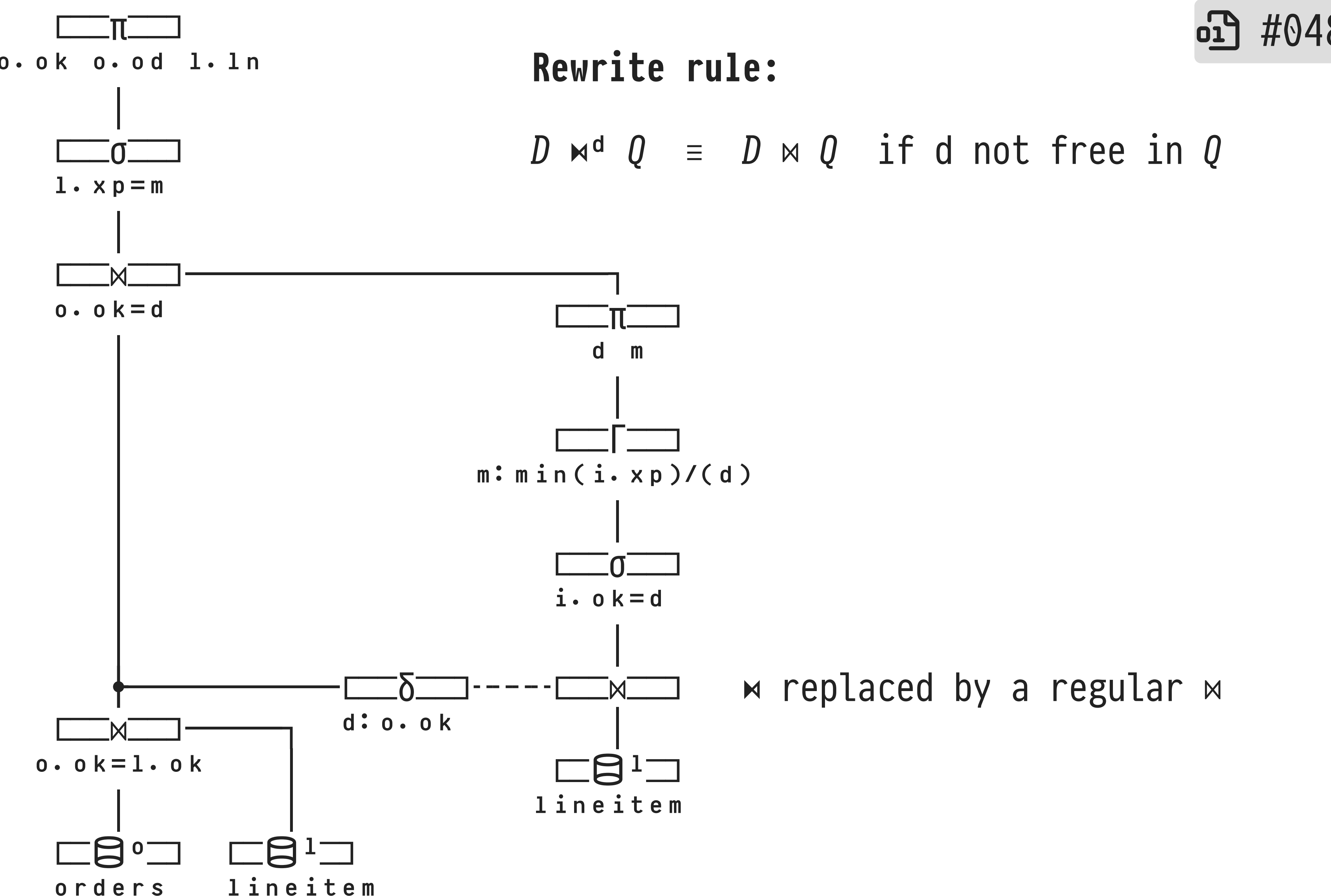
$$D \ltimes^{d} \sigma^{p}(Q) \equiv \sigma^{p}(D \ltimes^{d} Q)$$



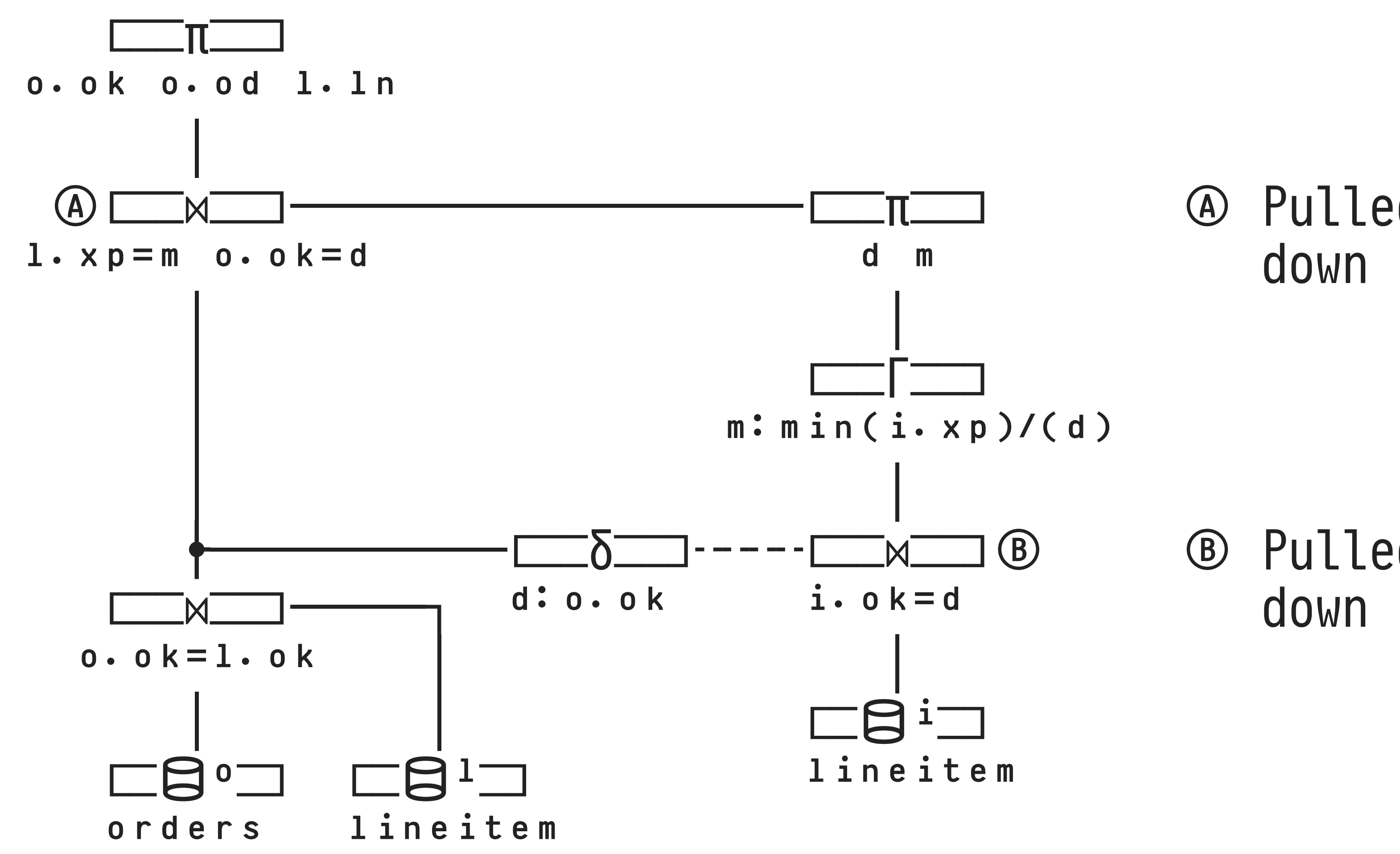
Query Decorrelation 5 (Eliminate ⋈)

Rewrite rule:

$D \bowtie^{d} Q \;\equiv\; D \bowtie Q$ if d not free in Q



17 ┊ Post-Optimizations 6 (Push σ Down Into ⋈)

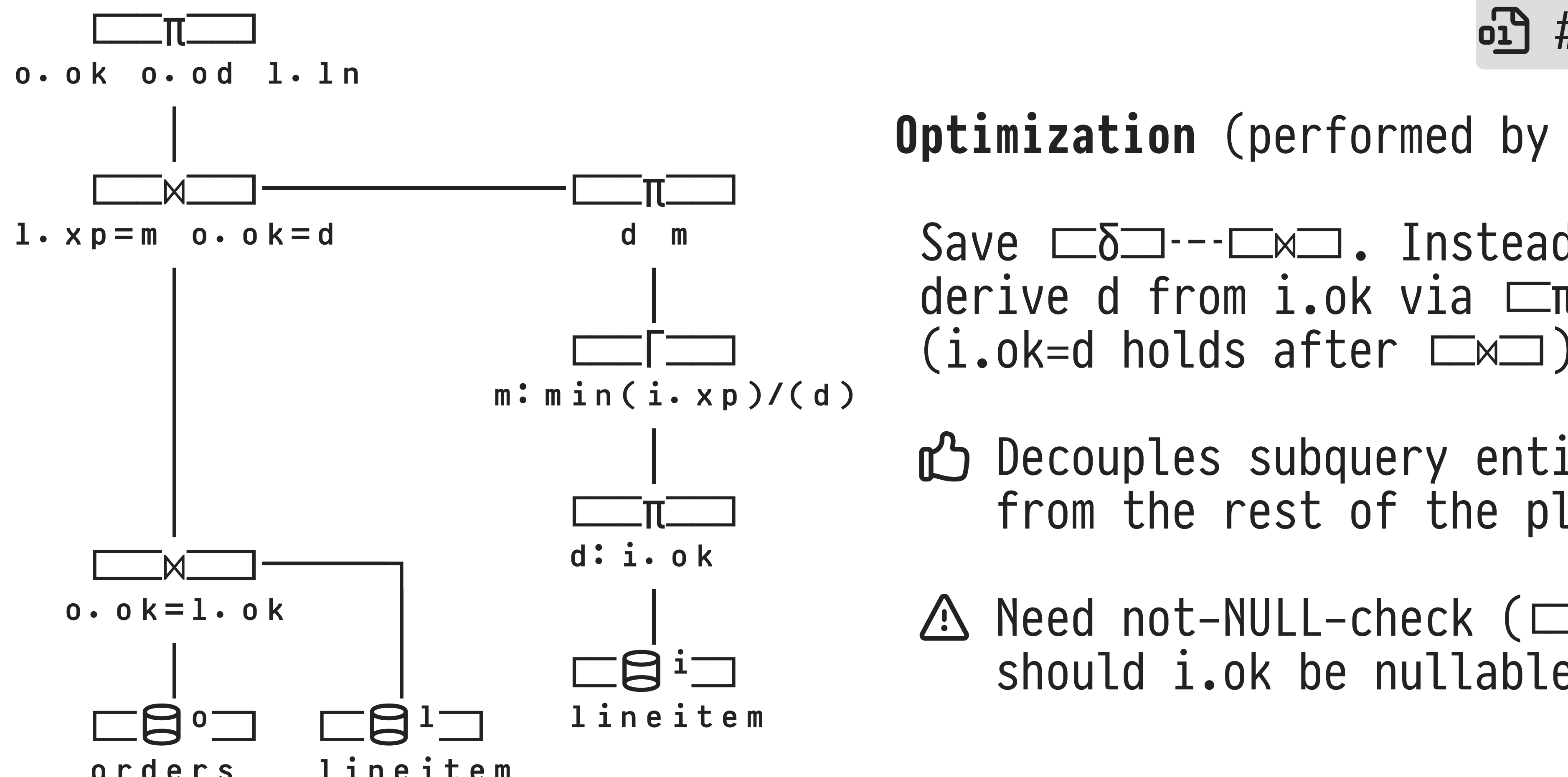


Ⓐ Pulled $\sigma^{l.xp=m}$ down into ⋈.

Ⓑ Pulled $\sigma^{i.ok=d}$ down into into ⋈.

18 ┊ Post-optimization: decoupling

#048 ❸



Optimization (performed by 🦆):

Save δ---⋈. Instead, derive d from i.ok via π (i.ok=d holds after ⋈).

👍 Decouples subquery entirely from the rest of the plan.

⚠ Need not-NULL-check (σ) should i.ok be nullable.

The End.

Since you've got this far... A DuckDB-based companion course on the fundamentals of the relational data model and SQL is available at

https://github.com/DBatUTuebingen/TaDa.[10]

[10] TaDa: **Ta**bular **Da**tabase Systems