Design and Implementation of DuckDB Internals

①

Welcome & Setup

April 7, 2026

Torsten Grust
Universität Tübingen, Germany

© 2025–2026 Torsten Grust, U Tübingen, Germany

1 ┆ Welcome!

Welcome to this course which is all about digging deep into the internals of **tabular database management systems** (DBMSs).

Our tour through the DBMS kernel will touch on

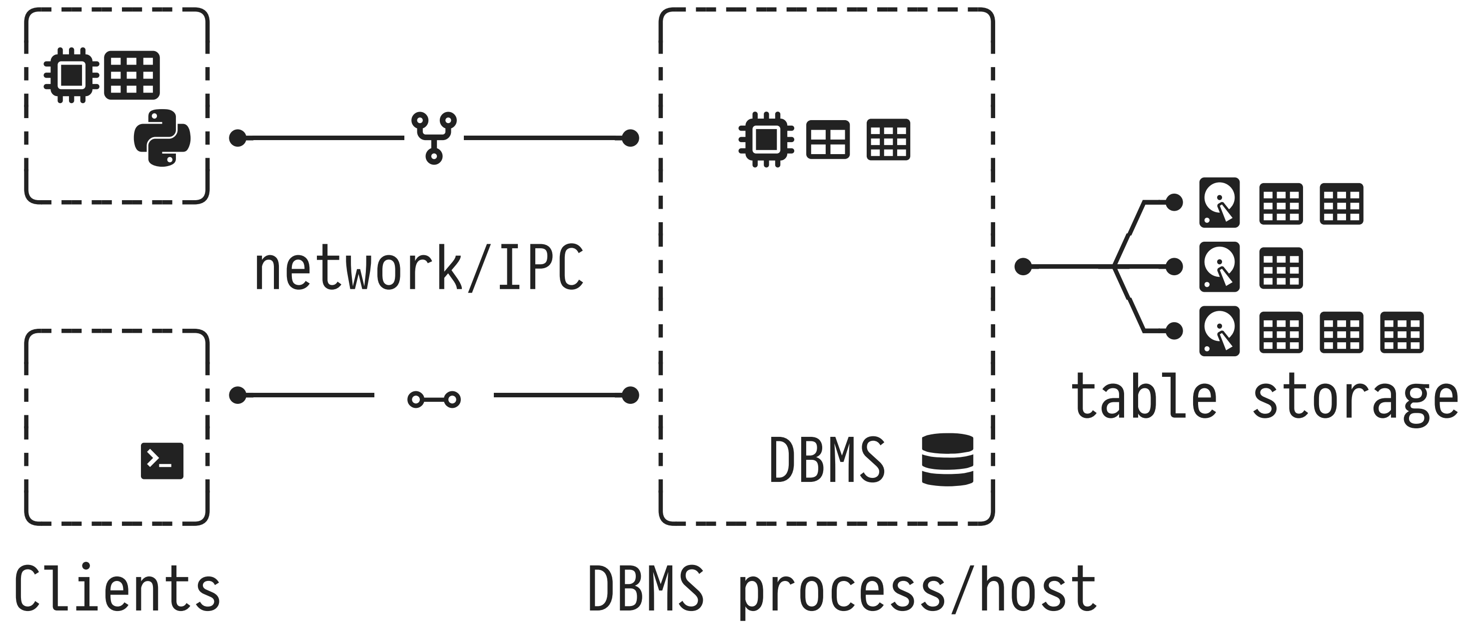
- the efficient representation of data on **secondary** (SSD/HDD) and **primary storage** (RAM),
- turning declarative SQL queries into efficient **flows of data**,
- a variety of interesting **data structures for sizable volumes** and associated **algorithms**,
- **modern CPUs** and how looping/branching code executes,
- various forms of **parallelism** on different levels (from single CPU instructions to threads), or
- ensuring **data integrity** under concurrent access or even if the host machine fails.

Dissecting the Duck's Innards

This course will focus on **DuckDB**, a contemporary tabular DBMS built for high-speed SQL-based data analytics.

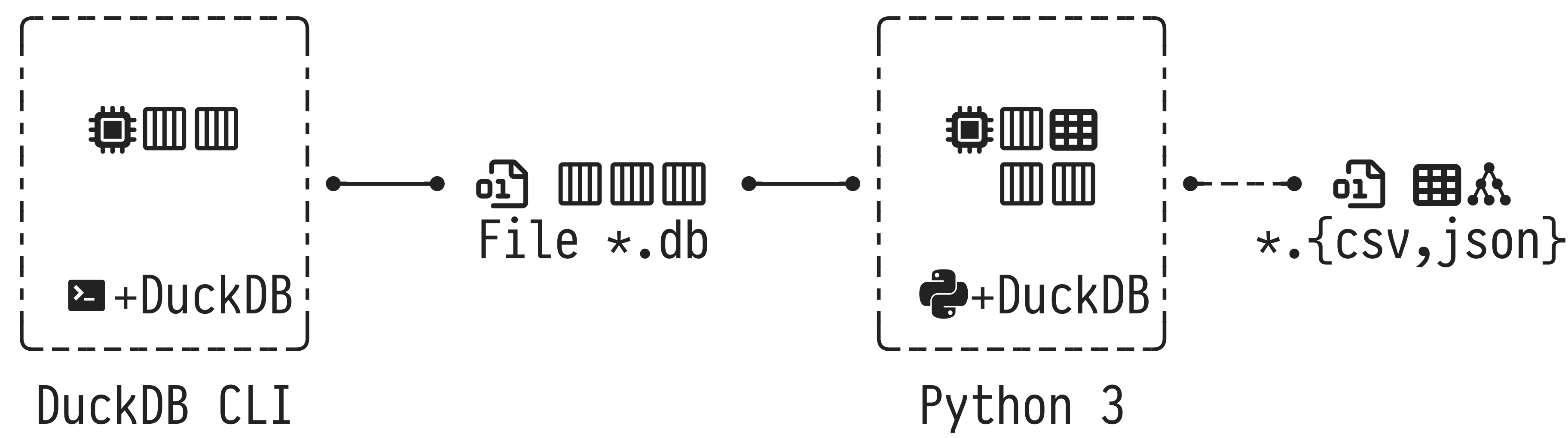
- DuckDB is young, first released in June 2019 (while most DBMSs—like PostgreSQL—originate from the 1980s).
- DuckDB development is moving fast. The system both adopts newest research results and builds on established DB wisdom.
- DuckDB is developed in the open (, MIT license). All code is available for inspection and tinkering.
- DuckDB provides a range of hooks to observe its internals.
- DuckDB comes with a CLI and programming language APIs.
- DuckDB is easy to install and requires no maintenance.

Not DuckDB: Dedicated DBMS Process/Host Isolated From Users



- DBMS controls family of disks holding table data, maps relevant table fragments into RAM buffer.
  - On-disk data organized in directories or in raw blocks.
- Client processes and DBMS process isolated, connected via network or inter-process communication.
  - Data needs to be de-/serialized after/before wire transfer.
- Data structures in clients are inaccessible by the DBMS.
- DBMS archetypes: PostgreSQL, MySQL, SQL Server™, Oracle®.

DuckDB: A Tabular DBMS Inside Your Own Process



- DuckDB kernel and client share a **single process**.
    - Python: import duckdb, C/C++: link with libduckdb.
- Table data resides in a **single database file** *.db, native DuckDB data formats in file and in RAM are similar.
- DuckDB sees in-process client data and can **directly read/write client data structures** using SQL ("**zero copy**", replacement scans).
- DuckDB allocates sizable RAM buffers (but can use disk for temporary storage if required).

2 ┊ This Course (*Dissecting the Duck's Innards*, short: *DiDi*)

- We will focus on DuckDB as a **tabular** SQL-based DBMS ▦. There are other kinds of DBMSs (for graphs, key/value pairs, vectors, …), but we will not discuss those here.

- We will get our hands dirty using **DuckDB** and its extensive **SQL** dialect. Lots of SQL will be read and written.

- Whenever possible we try to observe DuckDB under load or use hooks to inspect its operation while SQL queries are processed.

- We thus assume basic familiarity with the tabular data model and SQL, e.g., as discussed in *Tabular Database Systems* (*TaDa*).

- We will draw data and queries from a variety of sources and have **fun** along the way! 😎

Torsten Grust?

Time Frame	Affiliation/Position
1989–1994	Diploma in Computer Science, TU Clausthal
1994–1999	Promotion (PhD), U Konstanz
2000	*Visiting Researcher*, IBM (USA)
2000–2004	Habilitation, U Konstanz
2004–2005	Professor Database Systems, TU Clausthal
2005–2008	Professor Database Systems, TU München
since 2008	Professor Database Systems, U Tübingen

- Web: https://db.cs.uni-tuebingen.de/grust
- Bluesky 🦋: @teggy.org
- E-mail: torsten.grust@uni-tuebingen.de

- Feel free to reach out with criticism, bug reports, suggestions for improvement, pats on the back, or simply to say “*Hi!*” 👋

Slides and Further Lecture Material

These **slides** (PDF), **code fragments** (SQL, Python, C), and **sample data** will be uploaded to a GitHub repository:

github.com/DBatUTuebingen/DiDi

- Slides point to relevant code files or extra material using tags like #001:
  - Refers to a file named 001-* on the GitHub repository (e.g., 001-sum-quantity.awk).

- **NB.** Code and extra material provide essential content (e.g., details on SQL-based experiments or sample data).
  - + = : Only slides + code provide a complete picture.

Material

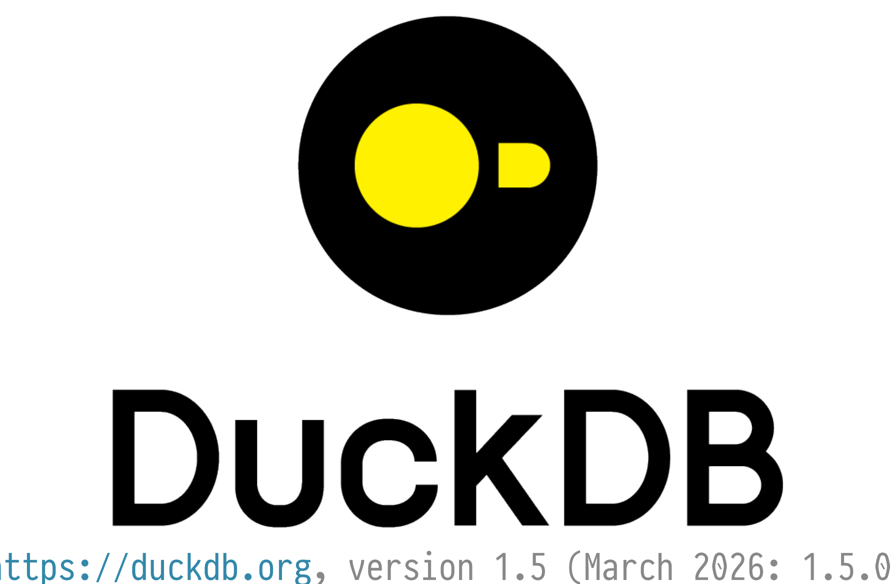
This course is *not* based on a single textbook. Rather, we build on

- a variety of scientific papers,
- textbook excerpts (few),
- the DuckDB documentation at https://duckdb.org/docs/,
- Python/C/C++ code snippets (our own and from inside the 🦆),
- blog posts from a range of authors,
- SQL references/standards,
- experience, and best practices.

There is a plethora of books on tabular DBMSs (both usage and internals), sample SQL snippets (experiments, benchmarks, and idioms), or performance tweaks. If we will use such sources, we will provide pointers.

Get Your Hands Dirty: Install DuckDB!

The tabular DBMS **DuckDB** will be the primary tool in this course:



https://duckdb.org, version 1.5 (March 2026: 1.5.0)

- Implements an extensive SQL dialect, is highly performant, open to contributions, and generally awesome.
- Straightforward to install and use on macOS, Windows, Linux (x86 + ARM).

  No DuckDB CLI (SQL prompt/REPL) on iOS or Android.[2]

[2] Run the DuckDB CLI in the web browser: https://shell.duckdb.org. Suffices for quick SQL experiments.

3 ⁞ *Duck*DB? 🦆?

In case you were wondering:

DuckDB has been named after *Wilbur, the Duck*, which has been living as a pet with Hannes Mühleisen[3]—co-inventor of DuckDB with Mark Raasveldt—on Hannes' houseboat in Amsterdam.

Hannes (CEO) and Mark (CTO) run DuckDB Labs, a company that provides support and consultancy services around DuckDB. The labs are located in Amsterdam, The Netherlands.

Hannes and Wilbur (© Hannes Mühleisen)

[3] Hannes originally is from the Stuttgart area. Back then his car had the license plate S QL 1337.

Design and Implementation of DuckDB Internals

②

The Query Performance Spectrum

April 7, 2026

Torsten Grust
Universität Tübingen, Germany

© 2025–2026 Torsten Grust, U Tübingen, Germany

1 ¦ DBMSs Exploit Modern Computer Architecture[1]

The internals of DBMSs are carefully engineered to exploit the performance features of modern computer architecture:

- **CPUs** (and their multi-threading capabilities),
- **main memory** (DRAM) and its hierarchy of caches, and
- **secondary memory** (mass storage on SSDs or rotating disks).

Since database queries typically process millions of rows, the effect of even the tiniest performance tweaks/tricks played in the innermost loops of DBMS routines multiply.

Goal: Understand the performance spectrum for a simple "query."

quick one-liner shell script hand-written C program

[1] This chapter adapts and expands on a discussion found in Thomas Neumann's lecture "Foundations in Data Engineering" (TUM).

A Simple Benchmark Query

1. Read the CSV file for TPC-H table lineitem (scale factor SF = 1: 6+ million rows × 16 columns ≈ 720 MB of data) and
2. sum the quantity integer values in the 5th column:

lineitem.csv

```
1|155190|7706|1|17|21168.23|0.04|0.02|N|O|1996-03-13|…␤
1|67310|7311|2|36|45983.16|0.09|0.06|N|O|1996-04-12|…␤
1|63700|3701|3|8|13309.60|0.10|0.02|N|O|1996-01-29|…␤
 ⋮
 [6+ million more rows]
```

- Real TPC-H benchmark data and queries are more complex but this suffices to demonstrate the effect of code optimizations.
- We will implement the query in awk, Python, C, and SQL.

2 ┆ Performance Limits

What is the fastest query time we can hope for in principle?

- Torsten's current computer (MacBook Pro M2 Max, 2023):

Memory (❶ Primary/❷ Secondary)	Read Bandwidth	⏱ Query Time
(Ethernet)	2.5 GB/s	0.28s
❷ External USB-C SSD (2 TB)	800 MB/s	0.90s
❷ NVMe SSD (2 TB)	5 GB/s	0.14s
❶ DRAM (64 GB)	21 GB/s	0.03s

- **NB.**
    - Column **Query Time** based on I/O speed, ignores CPU cost (less significant for secondary mem, very significant for DRAM).
    - ⇒ We will *not* reach these limits. Let us try to get close.

- Understand how DuckDB achieves 0.002s for our query. 🦆

3 ⁝ Sum of Quantities ❶ — awk

- awk: interpreted text processing language popular on UNIX™.
  - Read input line by line, match each line against (regular) patterns in order, on a match invoke action {…} on line.

```
BEGIN  { FS = "|"                      ┆   delimiter in CSV is |          #001
         sum = 0 }                     ┆   match first line, reset sum
       { sum = sum + $5 }              ┆   match any line, sum 5th column
END    { print sum }                   ┆   match last line, output sum
```

- Invoke the awk script, measure elapsed wall-clock time (s) ⏱:

```
$ time ./001-sum-quantity.awk lineitem.csv
153078795
        1.58 real         1.43 user         0.14 sys
```

Sum of Quantities ❶ — awk

- ⏱ Query time on Torsten's computer: ≈1.6s:

Output of time	Measurement
real	elapsed wall-clock time ⏱ (≈ user+sys+Δ)
user	time spent in application/library code
sys	time used by OS (system calls)

- The interpreted awk script cannot even keep up with secondary memory (SSD) read bandwidth:[2]
  - awk processes the CSV file with a throughput of 471 MB/s.
  - awk is **CPU-bound** for this query.
  - ⇒ The OS file system cache in DRAM does not help.

[2] Execution speed of awk variants vary greatly. We are using GNU awk (gawk) here. macOS awk is about 10 times slower for our benchmark query.

4 ┆ Sum of Quantities ❷ — Python

- Python: established scripting/programming language, mainly follows an imperative paradigm.
  - Translates to bytecode, then interprets.

```
sum = 0                                         ┆ reset sum              #002
                                                ┆
with open(sys.argv[1], 'r') as file:            ┆ open file (reading)
  for line in file:                             ┆ read line by line
    sum = sum + int(line.split('|', 5)[4])      ┆ | extract 5th col,
                                                ┆ ⌊ cast to int, add
  print(sum)                                    ┆ output
```

- ⏱ Query time on Torsten's computer: ≈2.75s.
  - Python processes the CSV file with a throughput of 275 MB/s.
  - Python is **CPU-bound** for this query.

5 ┆ Sum of Quantities ❸ — C

- Switch to compiled programming language C. Start out with a direct transcription of the Python code:
  - Read CSV file line by line using getline(3).[3]
  - Use strchr(3) to search for delimiter '|' in line (4×).
  - Convert string (up to next '|') into integer using atoi(3).

```
while (getline(&line, &linecap, file) > 0) {
  char* delim = line;

  for (column = 1; column < 5; column++) {
    delim = strchr(delim, '|');
    delim++;
  }

  sum = sum + atoi(delim);
}
```
#003

[3] The (3) suffix in getline(3) refers to Section 3 of the UNIX™ manual which describes the function in the C Standard Library.

Sum of Quantities ❸ — C

- ⏱ Query time on Torsten's computer: ≈0.5s.
  - C processes the CSV file with a throughput of 1.5 GB/s.
  - Yet, C still is **CPU-bound** for this query. Getting closer to SSD read bandwidth, though.

- But where does time go?
  - **Profile** the running program, identify code portions consuming the most CPU time (UNIX™: perf, macOS: Instruments).

C library routine	% of CPU time
(all other C code)	(16%)
getdelim (≡ getline)	58%
atoi	14%
memchr (≡ strchr)	12%

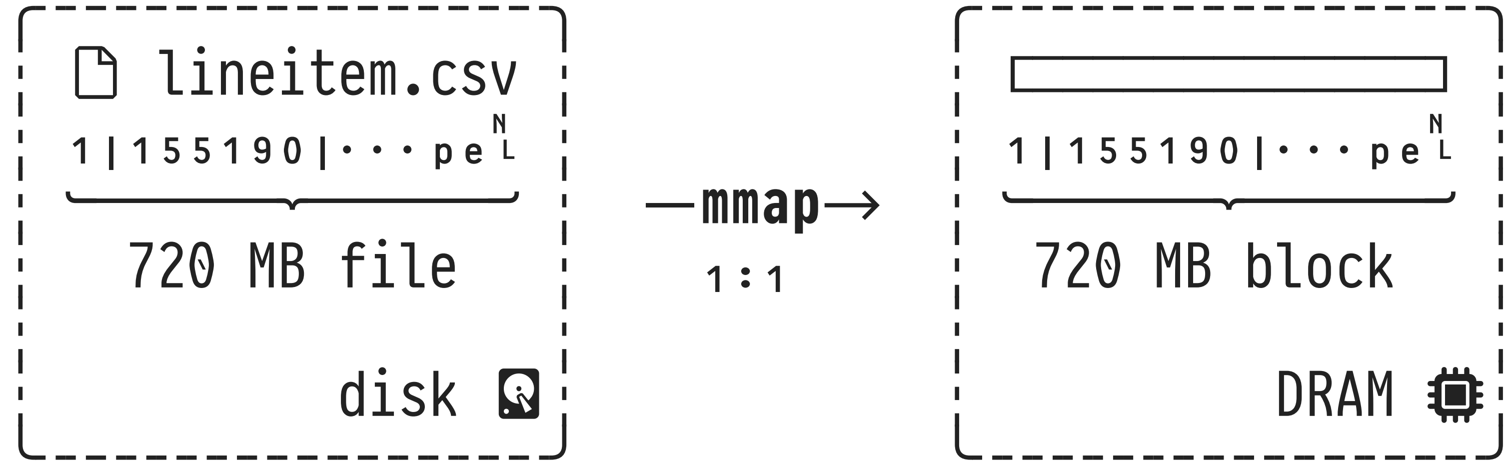
- Reading the CSV file line by line is too slow.
  - 💡 Read entire file at once, impose line structure ourselves.

6 ┆ Sum of Quantities ❹ — C with mmap(2)

Aim to *read the CSV file* into DRAM using a *single OS system call*:

- mmap(2) returns a pointer data to a contiguous block of memory that holds the contents of an entire disk file:

```
data = (char*)mmap(NULL, size, PROT_READ, MAP_SHARED, file, 0);
```



  - If file size exceeds available DRAM, OS pages in file contents on demand.

Sum of Quantities ❹ — C with mmap(2)

- Cannot use `strchr()` to find `'|'` (next column) ❶.
- No `getline()`: need to locate ␤ (`'\n'`) ❷ on our own now:

```
column = 1;                              start in column 1                    #004

while (data < end) {                     scan memory block, byte by byte
  switch (*data) {
    case '|' : column++; break;          ❶ found |: next column begins
    case '\n': ...                       error: line has too few columns
  }
  data++;                                proceed through memory block

  if (column < 5)                        reached column 5 already?
    continue;                              no, keep scanning

  sum = sum + atoi(data);                convert to int (up to |,␤), add

  column = 1;                            next line starts with column 1
  while (*data++ != '\n');               ❷ skip rest of line until ␤
}
```

Sum of Quantities ❹ — C with mmap(2)

- ⏱ Query time on Torsten's computer:
    - Once the OS caches the file in DRAM, mmap() directly maps the file system cache into the program's address space.

OS file system cache	Query time ⏱	Throughput
cold	1.6s	471 MB/s
warm	0.42s	1.8 GB/s

- **NB.** The C program's profile has changed:

C code fragment/function	% of CPU time
(all other C code)	*(25.5%)*
atoi	21.4%
while (*data++ ≠ '\n')	👎 53.1%

- Search for ␤ ❷ dominates. 💡 Use strchr(data, '\n') instead:
    - ⏱ Query time: 0.27s (throughput 2.8 GB/s). 👍
    - How can strchr() be so efficient?

7 ┆ Sum of Quantities ❺ — C with mmap(2) + Block-Wise ␤ Search

Avoid byte-wise search for ␤. Modern CPUs operate on 64-bit words.

- 💡 Load **8 bytes (64 bits) at a time,** search for ␤ (`'\n'` = 0x0a) in this block. Advance pointer `data` in strides of 8 bytes.

```
/* HAS_NL(x): find ␤ in 64 bit-wide character block x */
#define HAS_ZERO(x) (((x) - 0x0101010101010101) &
                     ~(x) & 0x8080808080808080)
#define HAS_NL(x)   (HAS_ZERO(x ^ 0x0a0a0a0a0a0a0a0a))

    HAS_NL(0x0a42440a6b637544)  ≡  "Duck␤DB␤" reversed ⚠
  =        0x8000008000000000      (ARM CPUs: little endian)
             ↑     ↑
 high bit set: found ␤ at offsets 4 and 7
```

- How do C macros `HAS_ZERO()` and `HAS_NL()` work?[4]

#005

[4] See the Stanford Bit Twiddling Hacks (section "Determine if a word has a byte equal to n") for a discussion of these C macros.

Sum of Quantities ❺ — C with mmap(2) + Block-Wise ␤ Search

```
while (data < end) {                                                  #006
  ⋮
  sum = sum + atoi(data);

  column = 1;

  block = (uint64_t*)data;
  while (!(nl = HAS_NL(*block)))
    block++;  /* advance by one 8-byte-block (64 bits) */

  data = (char*)block;
  if (nl & 0x0000000000000080ULL) { data += 1; continue; }
  if (nl & 0x0000000000008000ULL) { data += 2; continue; }
  if (nl & 0x0000000000800000ULL) { data += 3; continue; }
  if (nl & 0x0000000080000000ULL) { data += 4; continue; }
  if (nl & 0x0000008000000000ULL) { data += 5; continue; }
  if (nl & 0x0000800000000000ULL) { data += 6; continue; }
  if (nl & 0x0080000000000000ULL) { data += 7; continue; }
  data += 8;
}
```

Sum of Quantities ❺ — C with mmap(2) + Block-Wise ␤ Search

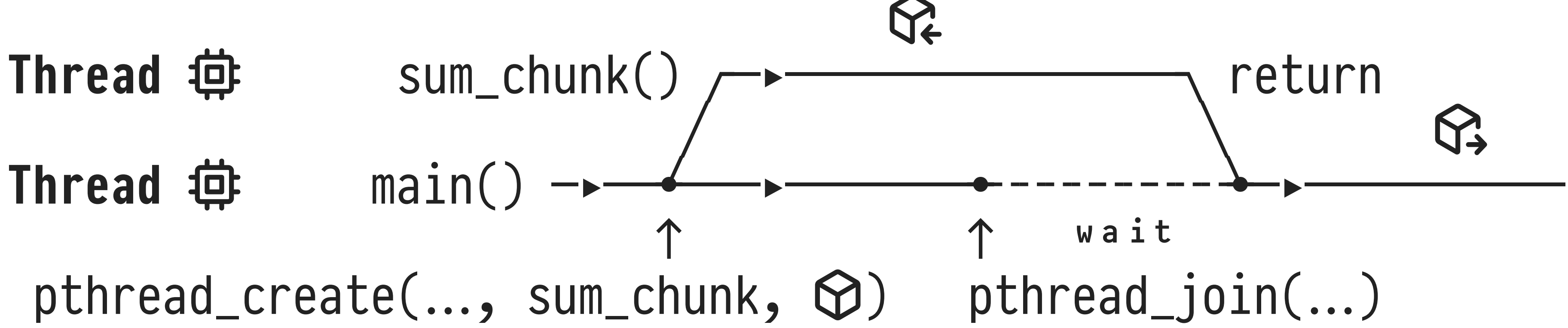
- ⏱ Query time on Torsten's computer (warm cache): ≈0.28s.
    - C with mmap() and block-wise search for ␤ processes the CSV file with a throughput of 2.8 GB/s.
    - We match the performance of the built-in strchr() function.

- Our code definitely got more complex and fiddly.
    - Slowly getting an impression of how much careful performance engineering is required to build a DBMS kernel.

8 ┆ Sum of Quantities ❻ — C with mmap(2) + Threads

CPUs feature **multiple cores** that can execute code in parallel. The CPU (M2 Max) in Torsten's computer features T = 12 such cores.

- Split CSV file at ␤ boundaries into T **partitions** (**chunks**).
- Spawn T **parallel threads**, each summing column 5 in one partition. Add thread-local partial sums to obtain overall sum.



- Threads use shared memory area to exchange data (*e.g.*, thread ID, pointers to chunk start/end, thread-local sum).

Sum of Quantities ❻ — C with mmap(2) + Threads

- C declaration of shared memory area 📦:

```
struct chunk {
  pthread_t thread;  /* thread ID */
  char      *data;   /* pointers to chunk start/end */
  char      *end;
  int       sum;     /* sum of partition */
};
typedef struct chunk chunk_t;
```

- Code for a thread (sums a chunk): #007

```
void *sum_chunk(void *arg)
{
  chunk_t *chunk = (chunk_t*) arg;  /* this is the 📦 */

  char *data = chunk->data;  /* extract relevant arguments */
  char *end  = chunk->end;

  ⋮        /* sum chunk, just like sum-quantity-mmap.c did */

  chunk->sum = sum;            /* put return value into 📦 */
  return NULL;                 /* NULL ≡ 👍 */
}
```

Sum of Quantities ❻ — C with mmap(2) + Threads

- ⏱ Query time on Torsten's computer (T = 12 threads spawned, warm cache): ≈0.04s.
  - Jointly, the threads process the CSV file with a throughput of 18.8 GB/s. This approaches DRAM read bandwidth.

- Profile shows that each `sum_chunk()` + `main()` use about the same chunk summing time. 👍
  - Creating chunk sizes (and thus thread-local work) of roughly equal size has worked well.
  - Wait time after `pthread_join(…)` expected to be small.

Sum of Quantities — Summary so Far

Query Implementation	Query Time ⏱	Throughput
awk	1.60s	471 MB/s
Python	2.75s	275 MB/s
C with getline	0.50s	1.5 GB/s
C with mmap	0.27s	2.8 GB/s
C with mmap + block-wise scan	0.28s	2.8 GB/s
C with mmap + 12 threads	0.04s	18.8 GB/s

- Implementation language and techniques matter **a lot.**
  - 50+ years after the inception of the relational model, database query optimization is a lively field of research.
- Even your laptop can read and process multiple GB/s.
  - Here we saturate everything (but DRAM).
  - Do we always need "big iron" or server clusters? (🦆: "*No!*")

9 ┆ Sum of Quantities ❼ — SQL

Aggregate function sum() is all we need to formulate a SQL variant of the benchmark query over the CSV file:[5]

```
#008
D .timer on
D SELECT sum(l_quantity)
  FROM   read_csv('../../databases/lineitem.csv',
                  header = false,
                  names = [ ..., 'l_quantity', ... ])
┌─────────────────┐
│ sum(l_quantity) │
├─────────────────┤
│    153078795    │
└─────────────────┘
                 ⏱      used CPU time
              ↙              ↓
Run Time (s): real 0.448 user 3.322090 sys 0.133819
```

- ⏱ Query time on Torsten's computer: ~0.45s.
  - Overall CPU time is >3s: DuckDB uses **parallel processing.**

[5] Command .timer on instructs the DuckDB CLI to report query times for all subsequent SQL commands. The DuckDB documentation contains a complete list of such commands.

Interlude: SQL EXPLAIN

SQL DBMSs typically implement an **EXPLAIN facility**[6] that exposes details of the system's **query evaluation plans:**

- Supports query and performance debugging (profiling).
- **Shows order of query operations** explicitly, making effects of query optimization visible (e.g., projection pushdown).
- **Profiles query behavior during execution**, annotates plan with:
  - breakdown of how query operations use time,
  - # of rows processed,
  - size of intermediate results (in bytes), …

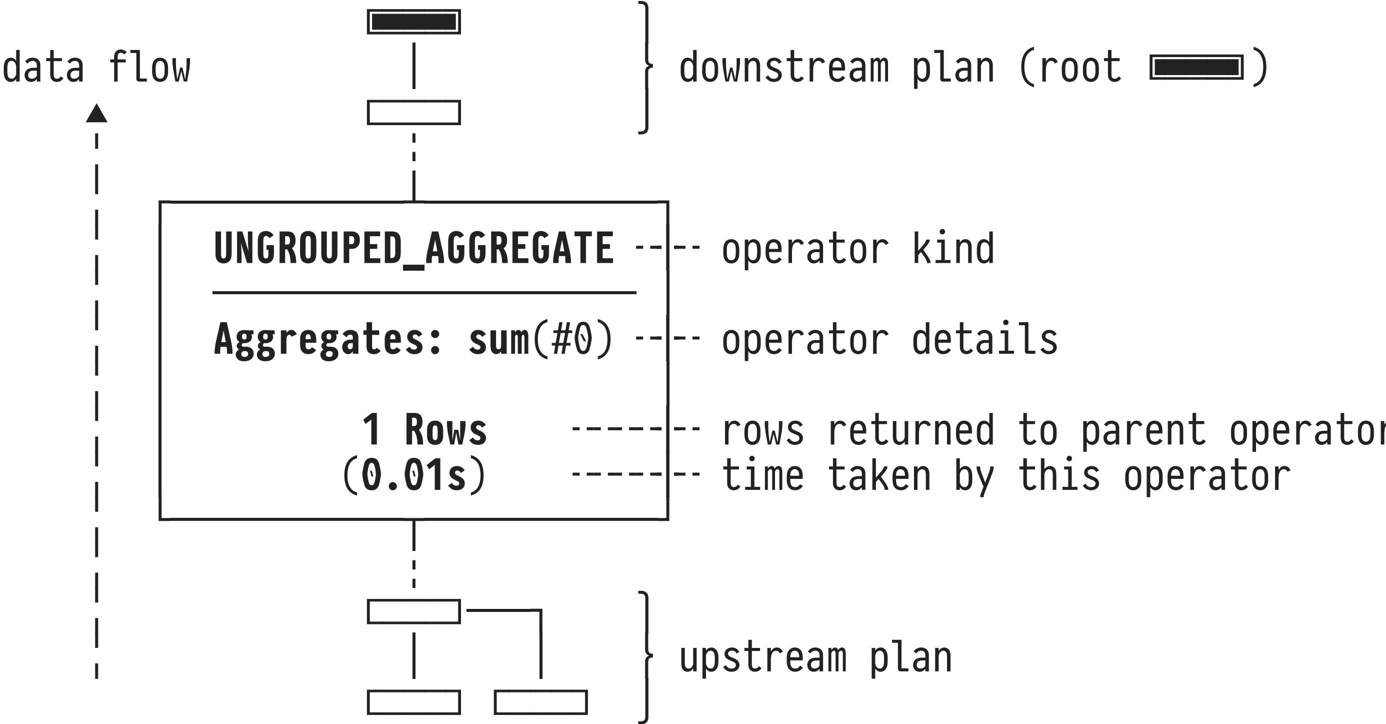
EXPLAIN *query*;	**EXPLAIN ANALYZE** *query*;
Do *not* run query. Show query plan and estimated row count.	*Actually run* query. Measure times/rows, annotate plan.

[6] DuckDB's EXPLAIN facililty is extensive. Can see even more plan details via EXPLAIN (FORMAT json) *query*.

Interlude: SQL EXPLAIN

Query evaluation plans visualize bottom-up **data flow:**[7]

- Data sources (e.g., TABLE_SCAN) reside at the leaves.
- Query result is produced by the root (top-most operator).



[7] U Tübingen has developed the DuckDB Execution Plan Visualizer which can render and inspect plans in the web browser (use EXPLAIN (ANALYZE, FORMAT json) *query* to produce plans that the visualizer can render).

Sum of Quantities ⑦ — SQL

Copy the CSV file into DuckDB's **native database storage format:**

```
D COPY lineitem FROM '../../databases/lineitem.csv';
D SELECT sum(l_quantity)
  FROM   lineitem;
┌─────────────────┐
│ sum(l_quantity) │
├─────────────────┤
│    153078795    │
└─────────────────┘
Run Time (s): real 0.002 user 0.007288 sys 0.000294
```

#008

A ⏱ query time of 0.002s for the benchmark indicates that

- the DBMS uses multiple cores (threads) to evaluate SQL queries,
- the query plan does *not* scan—or skip over—all 16 columns of table lineitem (projection pushdown focuses on l_quantity),
- column values are *not* stored as text and thus do *not* have to be parsed again and again, and that
- table lineitem does reside in DRAM (not in secondary storage).

Design and Implementation of DuckDB Internals

③

Managing Memory + Grouped Aggregation

April 7, 2026

Torsten Grust
Universität Tübingen, Germany

© 2025–2026 Torsten Grust, U Tübingen, Germany

1 ┆ Memory: Fast, But Tiny vs. Slow, But Large

- There are enormous **latency gaps** between accesses to the (L2) CPU cache, RAM, and secondary storage (SSD/HDD):

Memory	Actual Latency	Human Scale	Typical Size
CPU L2 cache	2.8 ns	1 s	$\frac{1}{4}$–16 MB
RAM	≈ 100 ns	36 s	4–256 GB
SSD	50–150 µs	5–15 h	$\frac{1}{2}$–4 TB
HDD	1-10 ms	4-40 days	1–16 TB

- Fact: faster memory is significantly smaller. We will *not* be able to build cache-only DBMSs. Thus:
  - Keep **persistent database data** on secondary memory (disk).
  - During **query processing**, try to keep all of the currently relevant (or: “hot”) data in RAM.
  - General: attempt to process all queries at the upper levels of the memory hierarchy.

Memory Management in DuckDB

Most DuckDB internals are optimized for **in-memory operation.** By default, 80% of the host RAM[1] is used to process data and queries.

Memory is a precious resource. DuckDB is built to use all RAM available but gradually moves to disk-based processing if required (*out-of-core processing*):

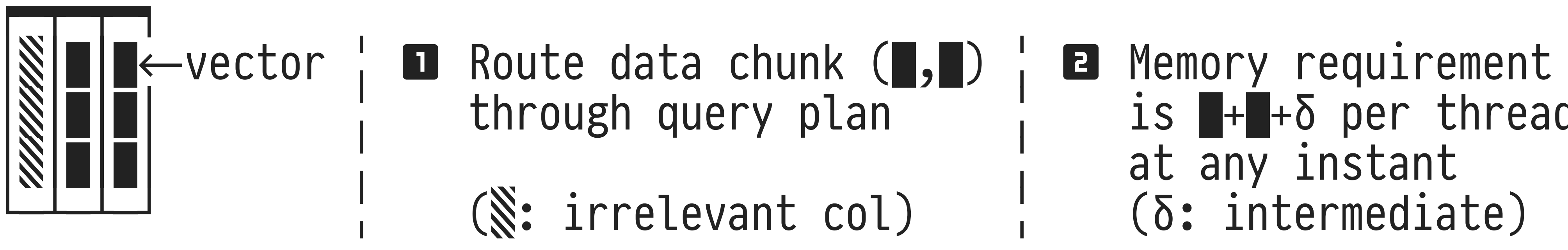
- If queries permit, **stream small data chunks** (packs of *vectors*) through the system. Avoid to materialize entire tables in RAM.
- Try to hold **intermediate data structures** (e.g., hash tables) in memory. **Spill to disk** if a query produces huge intermediates.
- In the remaining memory space (if any), cache disk-resident table data to speed up future accesses.
  DB lingo: **buffering.**

#009

[1] Compare these 80% to PostgreSQL's default memory buffer size of 128 MB (shared_buffers). DuckDB's configuration parameter memory_limit controls the maximum RAM size available to the DBMS.

Streaming Query Execution (Pipelining)

Many SQL queries admit **streaming execution** in which **vectors** of only 2048 elements each flow from data source (tables) to result:



- Simple filter (WHERE) + project (SELECT) operations.
- Ungrouped aggregations or grouped aggregations (GROUP BY) for which the number of groups is small.
- Returning *N* rows only (where *N* is small, LIMIT *N*).
- Reading data from one file and writing to another (e.g., reading from CSV and writing to Parquet, COPY).

Such queries can be evaluated over larger-than-memory tables even under very constraining memory_limit settings.

#010

Spilling

Intermediate data structures larger than available RAM are common when processing complex SQL queries:

- Aggregation in which the grouping criteria create *many* groups (GROUP BY: hash table).
- Counting the *distinct* values in a column *C* with many distinct values (count(DISTINCT *C*): hash table).
- Joining two tables that *both* are larger than memory (join builds hash table for smaller table).
- Sorting larger-than-memory input data (ORDER BY).
- Evaluating a complex window function over larger-than-memory input (… OVER …‹*frame*› with a sweeping frame).

DuckDB temporarily uses **out-of-core** memory on disk to hold (parts of) large intermediates (DB lingo: **spilling**).

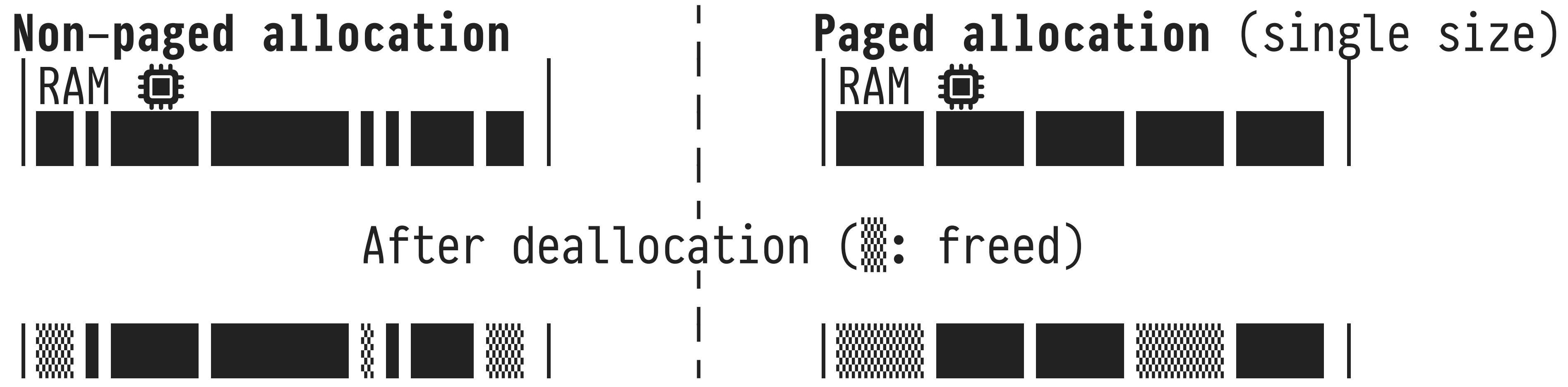
#011

DuckDB: Unified Memory Management

Use the *same memory area* of maximum size memory_limit to perform

1. **non-paged allocations:** fragmented (typically small) bits of data, e.g., as used in pointer-based data structures, and
2. **paged allocations:** blocks of contiguous data of various sizes, typically holding (parts of) tabular data structures. Typical block sizes range from 32KB to 256KB (default).

- DuckDB prefers paged allocation to avoid memory fragmentation:



Column-Wise Buffering of Base Table Data (🖴→⚙)

As long as memory usage permits, DuckDB **buffers** data read from persistent database file *.db file in RAM:

- File I/O is performed in 256KB blocks (as opposed to byte-by-byte) which matches the memory manager's default page size.

- Only buffer table columns relevant for query processing (recall Projections: … in operator SEQ_SCAN).

- If a page is no longer needed, add it to a LRU queue (DB lingo: **unpin**) as a candidate for replacement (DB lingo: **eviction**).
  - Buffered pages can be useful across queries: only evict once memory indeed is tight and needs to be reclaimed.
  - Buffered pages cache on-disk data (that could be re-read): *no need* to write evicted pages back to disk (⚙↛🖴).

Unified Memory Management in DuckDB

DuckDB flexibly assigns the available primary memory to all three kinds of allocations and thus is able to adapt to workloads/query specifics:

	Non-Paged Intermediate	Paged Intermediate	Buffered Base Table Data
Size	flexible	typical: 32…256KB (max: 2^{62} bytes)	256KB
Lifetime	query	query or session	across queries (database session)
Spilling	👎	👍	(not needed)

Unified Memory Management in DuckDB

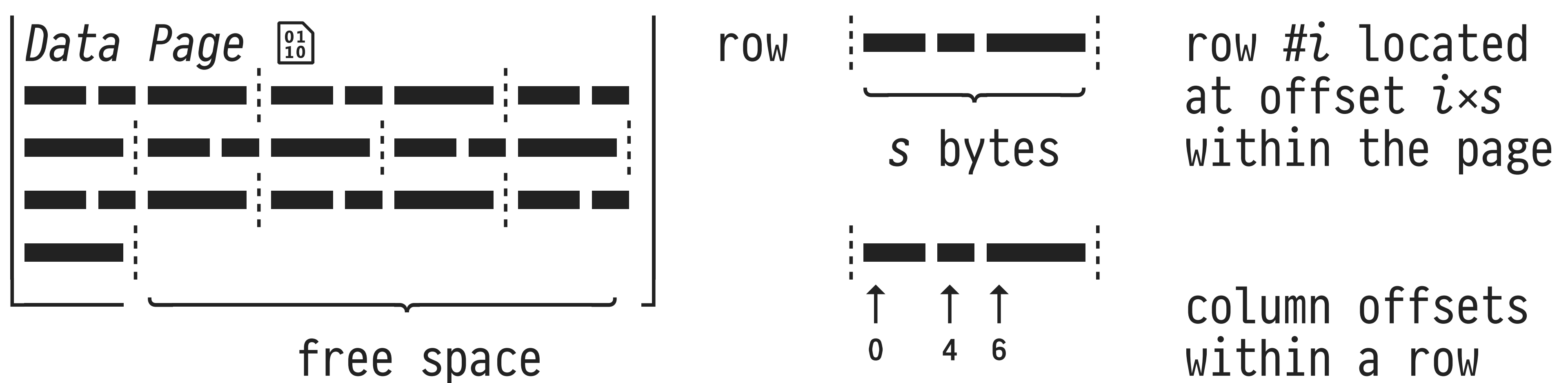
- In contrast to many DBMSs that require a pre-configuration[2] of buffer size (# of pages) or memory size for intermediates.

[2] Since DuckDB shares the address space with its host process, it is important that memory consumption is low when the DBMS is idle. DuckDB thus does use not a fixed-size pre-configured buffer.

2 ┊ Layout of Paged Intermediate Data Structures

- DuckDB uses **column-based** storage ▥ for base tables, but relies on a **row-major layout** ▤ for paged intermediate data:
    - Co-locating the columns of a row helps row comparison during sorting, hashing, or joins (column access locality).

- **Fixed-size rows** with **columns of known width** admit efficient row/column access via offset calucations:

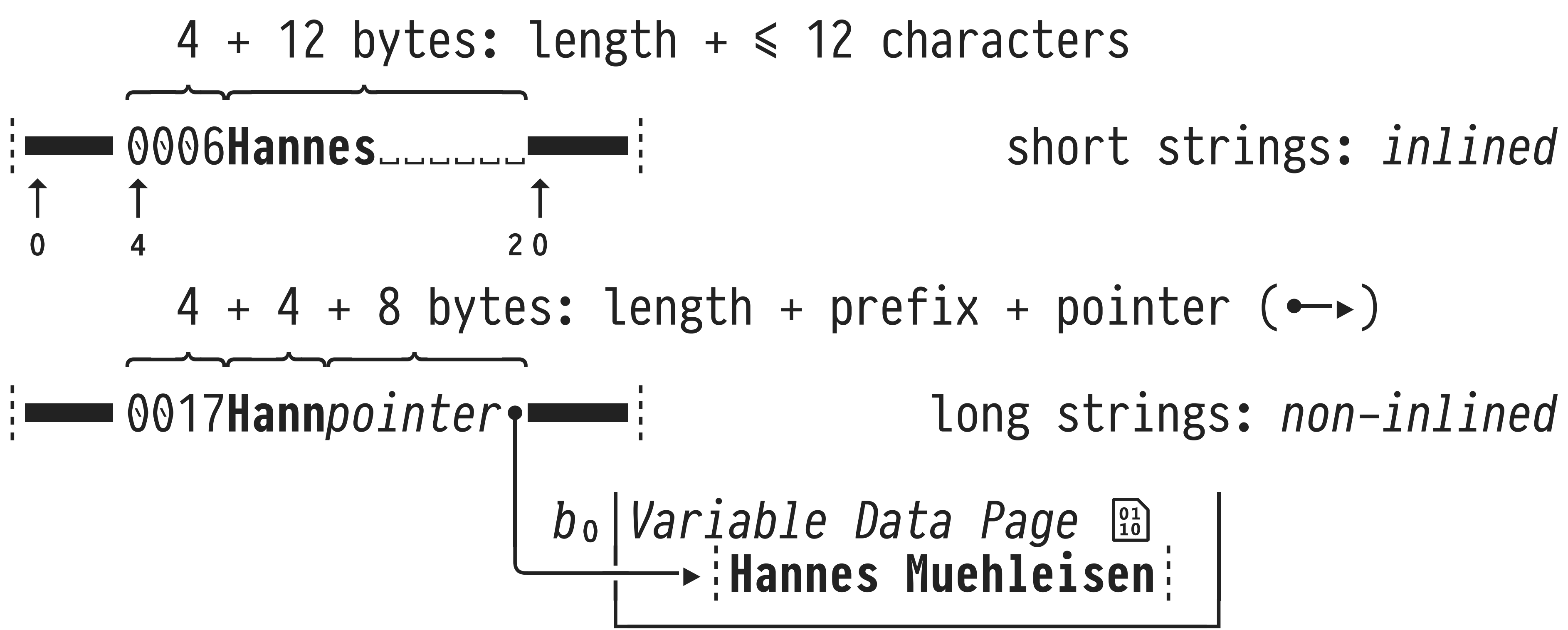


- Row size *s* and column offsets are known at query plan generation time: store these only once per *Data Page*.

Variable-Sized Columns in Fixed-Sized Rows?

How to cope with **variable-sized columns** (of types int[] or text)?

- DuckDB: Store variable-sized data *outside* of fixed-sized rows on **separate pages.** Use pointers to refer to variable data:[3]

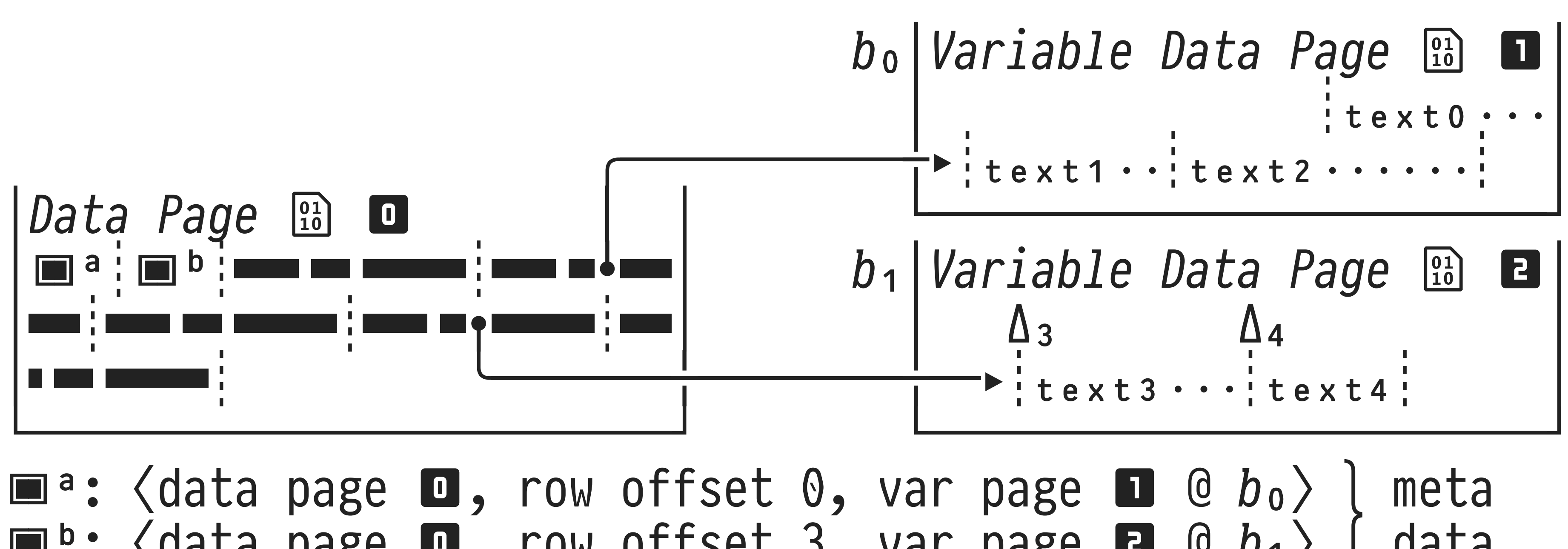


[3] This particular layout for string data is also known as "*German Strings*" since it has been popularized by the tabular DBMS *Umbra* developed at TU Munich (TUM). Now also used by *Apache Arrow* or *Polars*, for example.

Spilling Pages that Hold Pointer Targets

When a *Variable Data Page* is spilled and later loaded back, its base address b_0 may change and pointers ⟶ become invalid.

1. A string on a *Var Page* with base address b lives at $b+\Delta$.
2. Store b as meta data at the start of the *Data Page*.
3. If the *Var Page* is loaded back at base address B, on the next dereference (lazily) replace pointers to $b+\Delta$ by $B+\Delta$.



a: ⟨data page 0, row offset 0, var page 1 @ b_0⟩ } meta
b: ⟨data page 0, row offset 3, var page 2 @ b_1⟩ } data

3 ¦ Hash-Based Grouped Aggregation

Grouping and aggregation are core operations in OLAP workloads:

```
SELECT l_orderkey, count(*)
FROM   lineitem                -- TPC-H (sf = 100): 600,000,000 rows
GROUP BY l_orderkey            -- creates 150,000,000(!) groups
```

- DuckDB implements grouped aggregation in terms of plan operator HASH_GROUP_BY, i.e., through **hashing:**[4]
  1. Row with l_orderkey value o hashes to key k.
     Update entry hash_table[k]: (o,$count$) → (o,$count$+1).
  2. Row with l_orderkey value $o' \neq o$ also hashes to key k.[5]
     Collides with entry for o at hash_table[k]. DuckDB uses linear probing to find entry (o',$count$)—hopefully nearby.

[5] If DuckDB statically knows that no collisions will occur (e.g., when the grouping key has few distinct values like lineitem.l_returnflag), the efficient alternative operator PERFECT_HASH_GROUP_BY is used.

[4] Other tabular DBMSs also implement grouped aggregation through sorting (here: on column l_orderkey). Profitable if the query also contains an ORDER BY l_orderkey clause (DB lingo: *interesting order*).

DuckDB: Robust External Grouped Aggregation

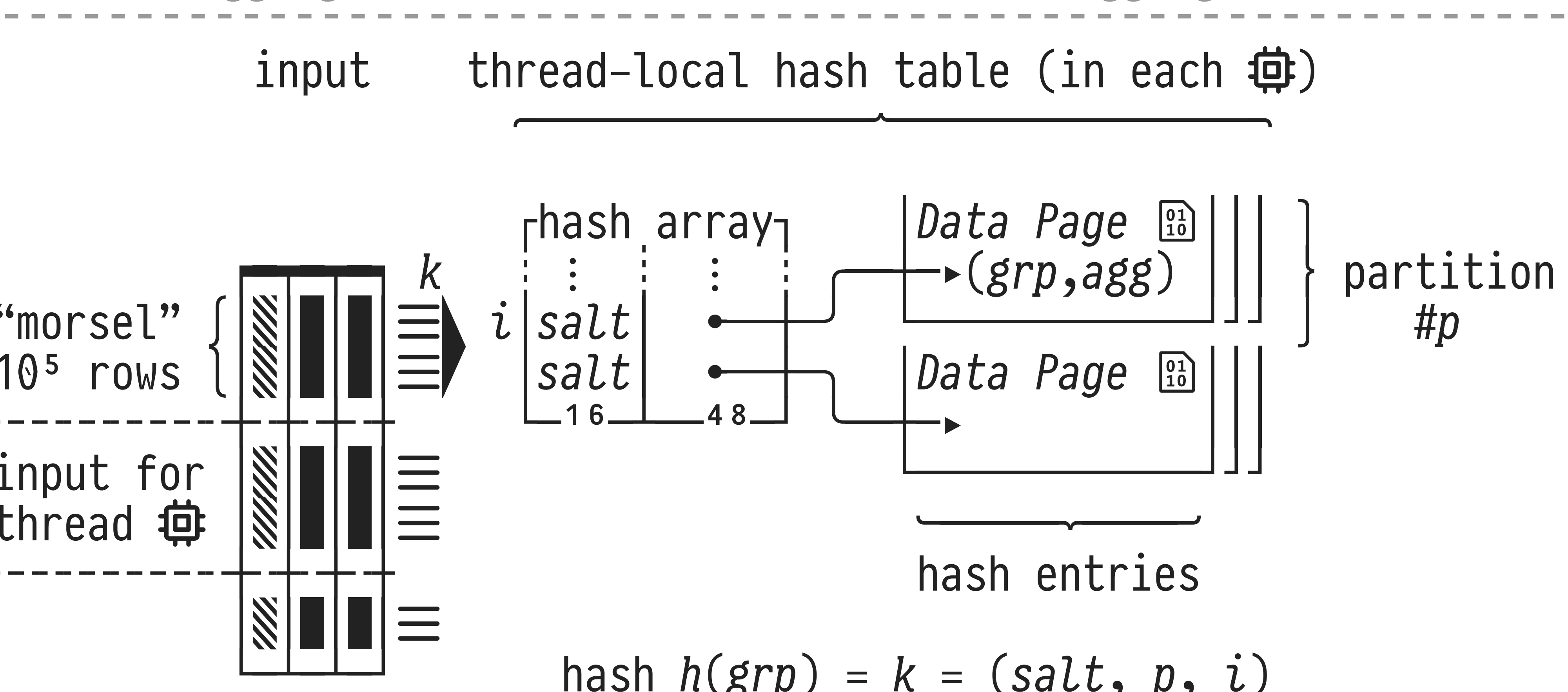
Designing the implementation of HASH_GROUP_BY:

- Aim to use all cores of modern **multi-threaded** CPUs.
- If we create **many groups**, the hash table entries may exceed available memory and thus need to be **spilled** to disk.
  - Maintain hash table entries as a *paged intermediate data structure*, let the memory manager handle the spilling.

DuckDB performs external grouped aggregation in two phases:

- Phase ❶: Thread-local pre-aggregation (one per morsel).
- Phase ❷: Partition-wise aggregation (one per partition).

External Aggregation (Phase ❶): Thread-Local Aggregation



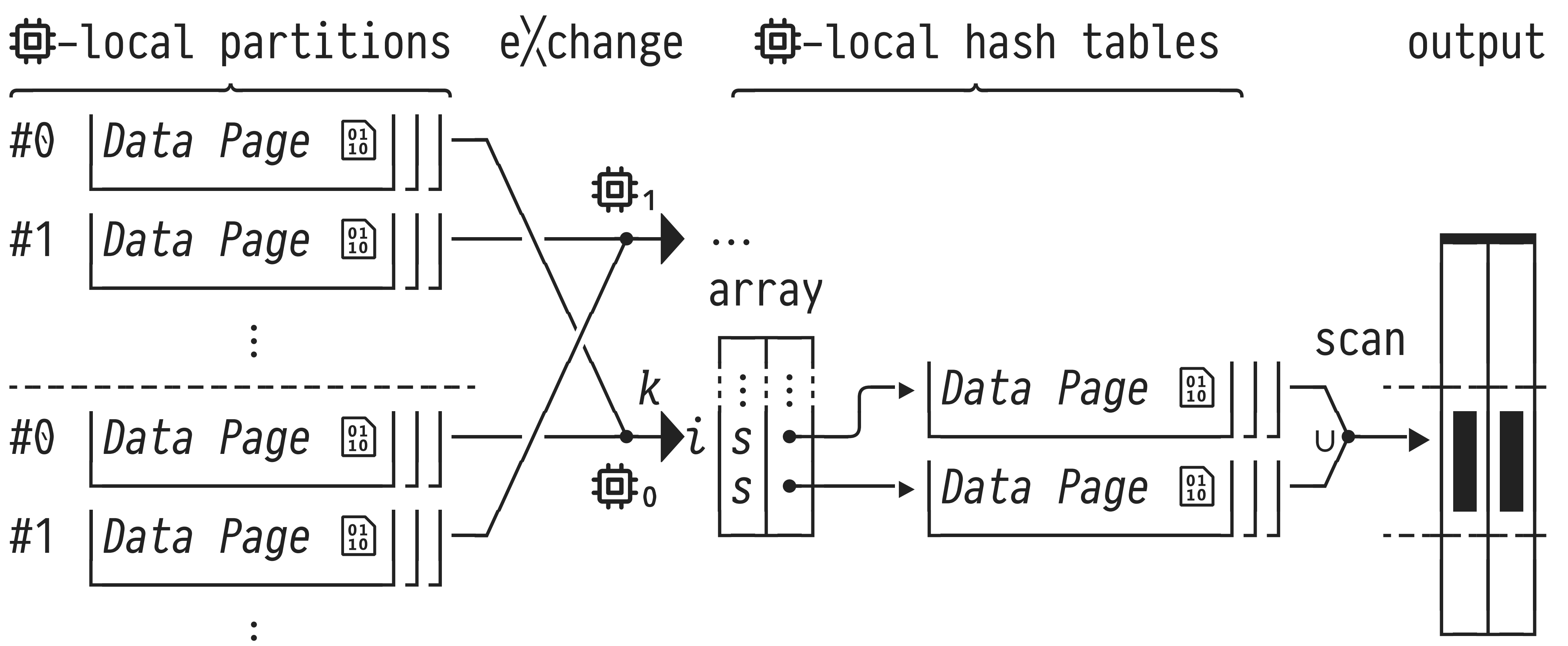
- Relevant columns of input table (grouping keys *grp* + arguments of aggregate *agg*) are split into **morsels** of ≈100,000 rows each.
- Each thread ⚙ reads a morsel. Typical: more morsels than ⚙s.

External Aggregation (Phase ❶): Thread-Local Aggregation

```
In each thread ⚙:
└ For each row in morsel:
  ├ Compute hash key k = h(grp). Split the bits of k:
  │ ├ Lower    17 bits i:    index in hash array (131,072 slots)
  │ ├ Middle ≤ 12 bits p:    partition # for hash entry
  │ └ Upper    16 bits salt: used to optimize collision handling
  └ Hash array slot i occupied?
    ├y─ Are salt values equal?
    │    ├y┬ Follow •─▸ to entry (grp′,agg) on data page 🗎.
    │    │ └ Is grp′ = grp?
    │    │    ├y─ No collision. Update value of agg. Done.
    │    │    └n─ Collision. Linear probing to find proper slot.
    │    └n─ Collision. Linear probing.
    └n┬ Add entry e = (grp,agg₀) to data page 🗎 in partition #p.
      └ Place (salt,•─▸ to e) in array slot i.
```

- When hash slots are $\frac{2}{3}$ full and collisions become frequent:
  1. Unpin data pages 🗎—the memory manager may evict these.
  2. Empty hash array, continue (reset OK because of Phase ❷).

External Aggregation (Phase ❷): Partition-Wise Aggregation



- Run one thread per partition #. Choose bits for *p* such that the hash table for a fully aggregated partition fits memory (create more but smaller partitions).
- When a ends, immediately scan its output data pages. These become a morsel to be fed to the downstream plan.

Design and Implementation of DuckDB Internals

④

Sorting Large Tables

April 7, 2026

Torsten Grust
Universität Tübingen, Germany

© 2025–2026 Torsten Grust, U Tübingen, Germany

1 ┆ Sorting is a Core Operation

The tabular model is based on unordered bags of rows, but the **sorting** of (intermediate) results of SQL queries is material:

1. **ORDER BY** clause,[1]
2. ordered aggregates (list(e_1 **ORDER BY** e_2)),
3. window functions (row_number() **OVER** (**ORDER BY** e)),
4. joins over time series data (**ASOF JOIN**), or
5. to ensure deterministic **LIMIT** and **OFFSET** behavior.

```
SELECT *                  -- wide payload (here: reorder all columns)
FROM   lineitem
ORDER BY l_shipdate DESC NULLS FIRST  ,  l_quantity;
--                  ⏟⏟⏟⏟   ⏟⏟⏟⏟⏟⏟⏟⏟⏟⏟⏟  ↑
--         defaults: ASC  NULLS LAST   lexicographic ordering
```

[1] In DuckDB, can adapt ordering defaults via SET default_order = {ASC,DESC} and SET default_null_order = {NULLS_FIRST,NULLS_LAST,NULLS_FIRST_ON_ASC_LAST_ON_DESC,NULLS_LAST_ON_ASC_FIRST_ON_DESC}.

Sorting (Large) Tables in DuckDB

To this day,[2] DuckDB's internal sorting routines are on the workbench 🛠 and still undergo active development.

Key aspects:

- **Efficient row comparison** that respects column types, directions (↑↓), handles NULLs, allows wide keys (lexicographic ordering).
- Adaptation to **pre-sorted input data.**
- Sorting strategy can process **tables larger than main memory,** spilling to disk 💾 if required.
    - Effectively, sorting needs to materialize its entire input table (last input row could be the first in sort order).
- Balanced **use of all available CPU cores** during sorting.

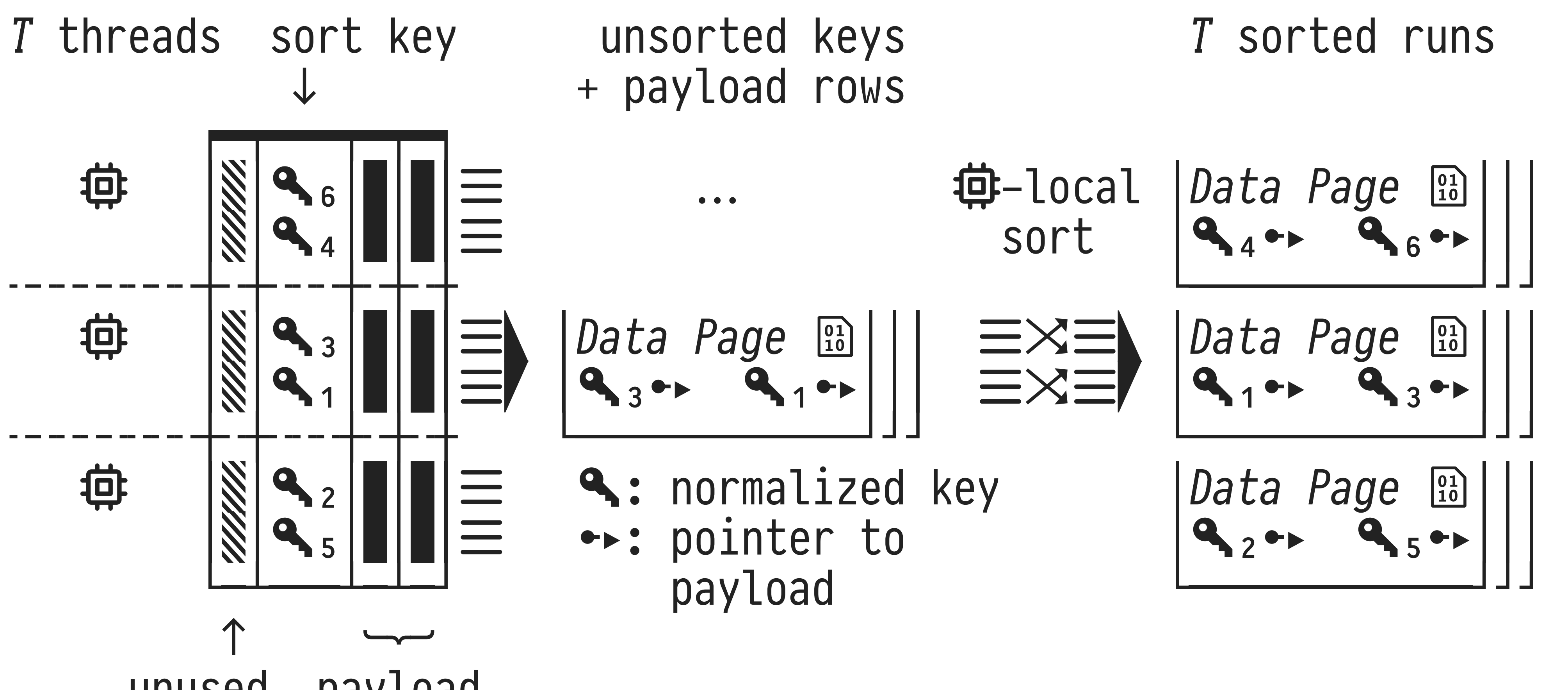
[2] The last major rewrite of DuckDB's sorting code has shipped with version 1.4.0 in September 2025.

DuckDB implements a variant of a **two-phase merge sort** strategy that utilizes all available CPU cores (say T) in both phases:

- **Phase ❶ (thread-local sorting):**
  - Each : read $1/T$ fraction of input, gather relevant columns to form rows on data pages, **normalize sort keys**,
  - then generate a **sorted run** of keys.

- **Phase ❷ (T-way merge of the sorted runs):**
  - Precompute boundaries of T non-overlapping **segments** in each run that can be **merged independently** by the s.
  - Each immediately outputs its merged run segments for consumption by the downstream query plan.

Phase ❶: Thread-Local Sorting



Phase ❶: 🔧 Key Normalization

- Sorting compares keys to decide row order ⤨. These comparison are frequent, but can be complex and costly:

```
⋮
ORDER BY c₁ ASC NULLS LAST, c₂ DESC NULLS FIRST
```

  - Dispatch on type of c_i to select proper < operator, respect sort order (↑↓), implement lexciographic ordering (column c_1 dominates), proper NULL treatment.

- 💡 **Key normalization:** Map[3] values in columns c_i to a byte sequence such that a *single < comparison* on two sequences can decide row order.

#014

[3] DuckDB's key normalization is available at the user level in terms of scalar SQL function create_sort_key().

Phase ➊: Key Normalization for Fixed-Size Keys

CPUs compare fixed-size types (e.g., 32/64-bit ints) faster than variable-length byte sequences.

- 💡 If sequence length is known to be n bytes, store it in groups of $\lceil n/8 \rceil$ 64-bit ints. Directly compare these integers:[4]

```
struct FixedSortKey {
  uint64_t       part0;   // assume 8 < n ⩽ 16: group size = 2
  uint64_t       part1;
  sort_key_ptr_t payload; // (only if query has payload)
};

bool LessThan(const FixedSortKey &k1, const FixedSortKey &k2) {
  return k1.part0 < k2.part0 ||
         (k1.part0 == k2.part0 && k1.part1 < k2.part1);
}
```

[4] If SQL queries have no payload (SELECT c_1,c_2 FROM … ORDER BY c_1,c_2), DuckDB only keeps the normalized keys in fields part0, …: normalization can be inverted before output is generated.

Phase ❶: Sorting Algorithms

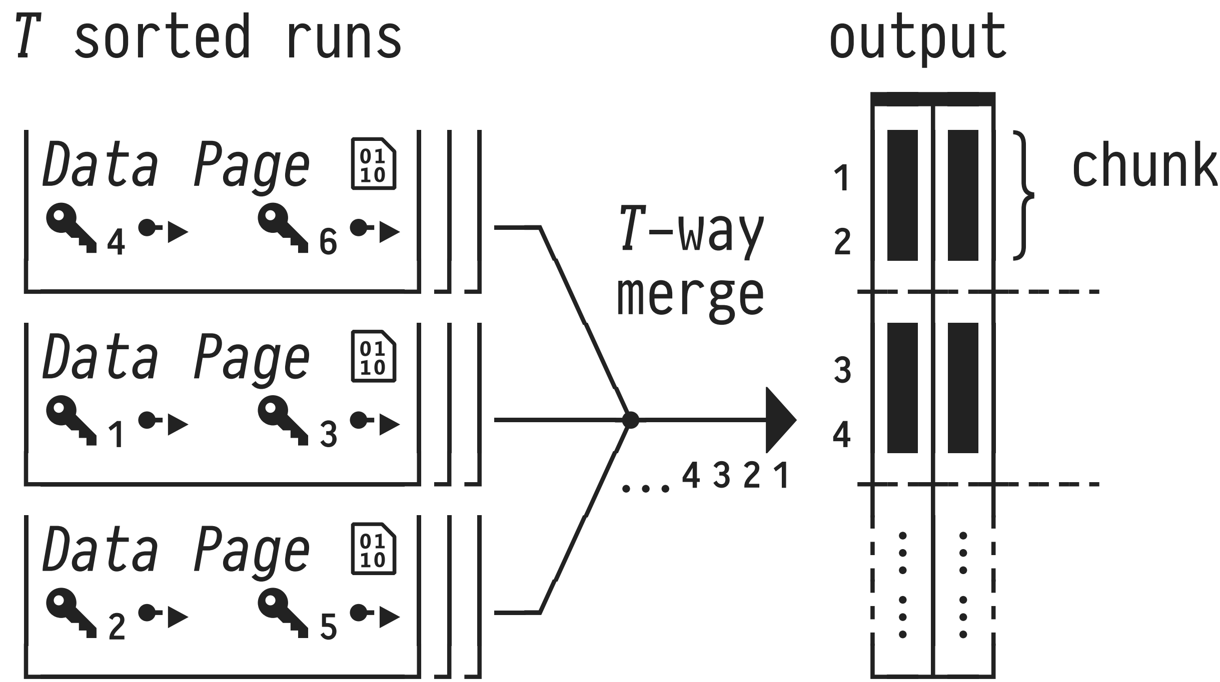
Typed keys (that can be compared via <) fit off-the-shelf sort implementations that assume the C++ `std::iterator` interface.

- DuckDB implements `std::iterator` for chains of 256KB data pages: 🗄→🗄→⋯→🗄, maps element indices to (page#, offset) pairs.
  - (A single 256KB data page would only be able to hold about 10,000 24-byte `FixedSortKey` structs. DuckDB aims to generate runs much longer than that.)

- DuckDB thus is able to adapt and combine three existing sorting algorithms:
  1. *Vergesort* (detects runs of already sorted data).
  2. *Ska Sort* (radix sort on the first 64 bits of the key).
  3. *Pattern-defeating QuickSort* (fallback, if the first 64 bits do not already order the data).

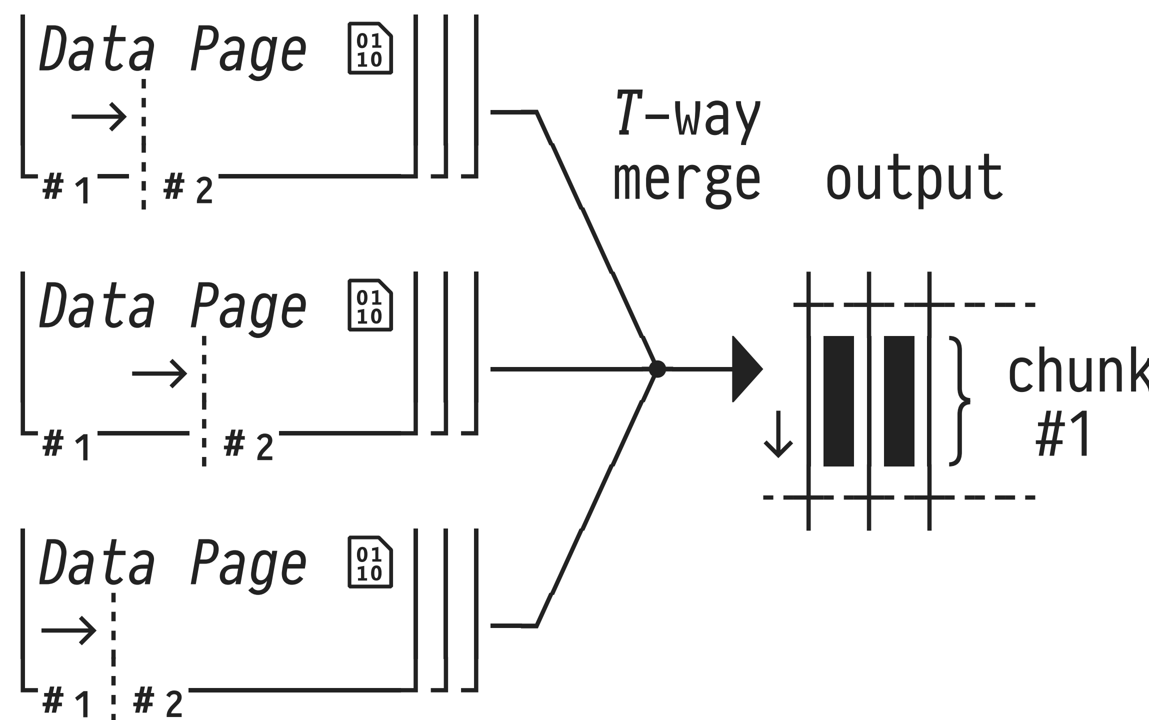
Phase ❷: *T*-Way Merge

- Consume the *T* sorted runs produced by Phase ❶:
  - **Merge** pairs (🔑$_k$, •▸) using < on key 🔑$_k$,
  - dereference pointers •▸ to access payload,
  - emit sorted output columns chunk-by-chunk:



- How can all CPU cores contribute equally during this merge?

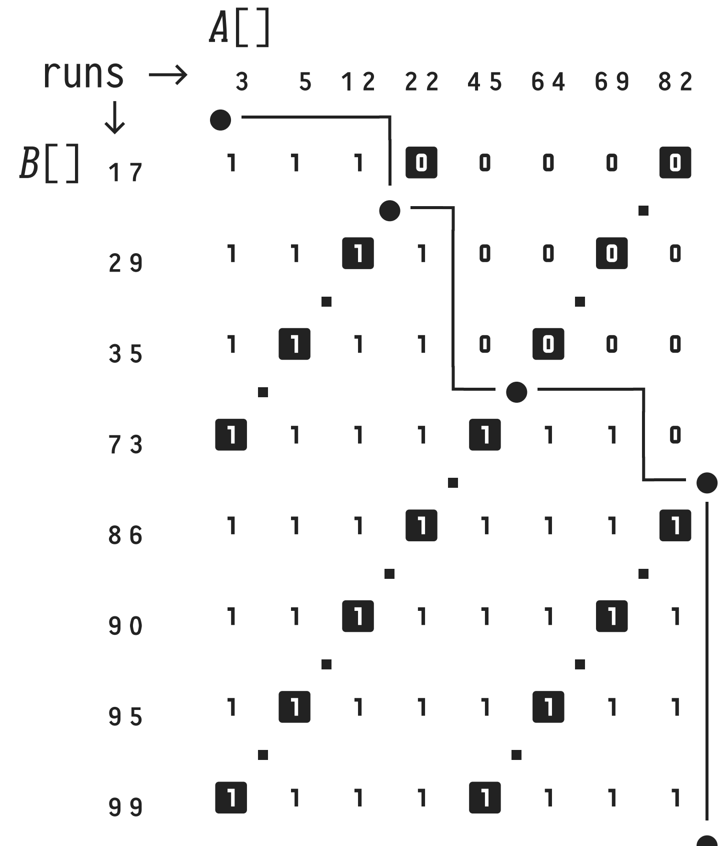
Phase ❷: Parallelizing *T*-Way Merge



- Runs contribute rows in **segment** #1 (up to ┆) to output chunk #1.
- Run segments #*i* hold the rows for chunk #*i*. No overlap between segments.
- Precompute segment boundaries ┆. Threads work on one segment.

- Once run segments #*i* are merged, the thread *immediately* emits chunk #*i*. The downstream plan can assemble the sorted output based on indices #*i* (DuckDB lingo: **batch indices**).

Phase ❷: Precomputing Segment Boundaries (*Merge Path*)



- Merge matrix $M[i,j]$: **1** if $A[i] < B[j]$, **0** otherwise.
- Merge path ╰──╮ marks the **1**/**0** boundary.
- The nth point of the merge path lies on the nth cross diagonal ▪▪▪ of M.
- Partitioning the path into k segments of equal size can be done by considering $k-1$ evenly-spaced diagonals.
- Point -●- marks the only transition **1** → **0** on the diagonals. Can be found by binary seach on A and B. (Merge matrix and merge path *need not* be constructed.)
- Points -●- define segments in A and B (┆ ≡ boundaries):

	#1	#2	#3	#4
A:	3 5 12	22 45	64 69 82	
B:	17	29 35	73	86 90 95 99

- DuckDB generalizes the *Merge Path* idea from two runs to *T* runs. (Profiling shows that boundary computation uses ⩽ 2% of the overall execution time.)

Design and Implementation of DuckDB Internals

⑤

The ART of Indexing

April 7, 2026

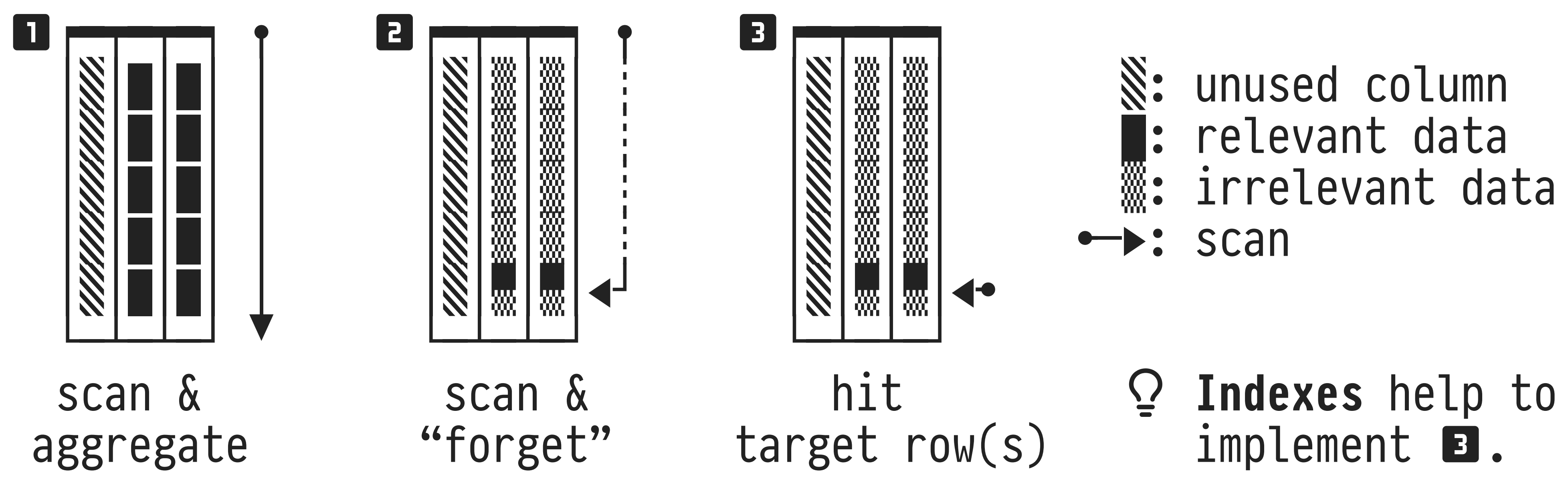
Torsten Grust
Universität Tübingen, Germany

© 2025–2026 Torsten Grust, U Tübingen, Germany

1 ┆ Scanning All Rows vs. Narrowing In On Few Rows

The internals of DuckDB have been designed to efficiently support analytical SQL queries: “***o**nline **a**nalytical **p**rocessing*” (**OLAP**).

- Typically, these queries read (and aggregate) *all rows* of the input tables **1**.
- Yet, scanning all rows wastes I/O and memory bandwidth **2** if a query focuses on *small row subsets (or even a single row)*:



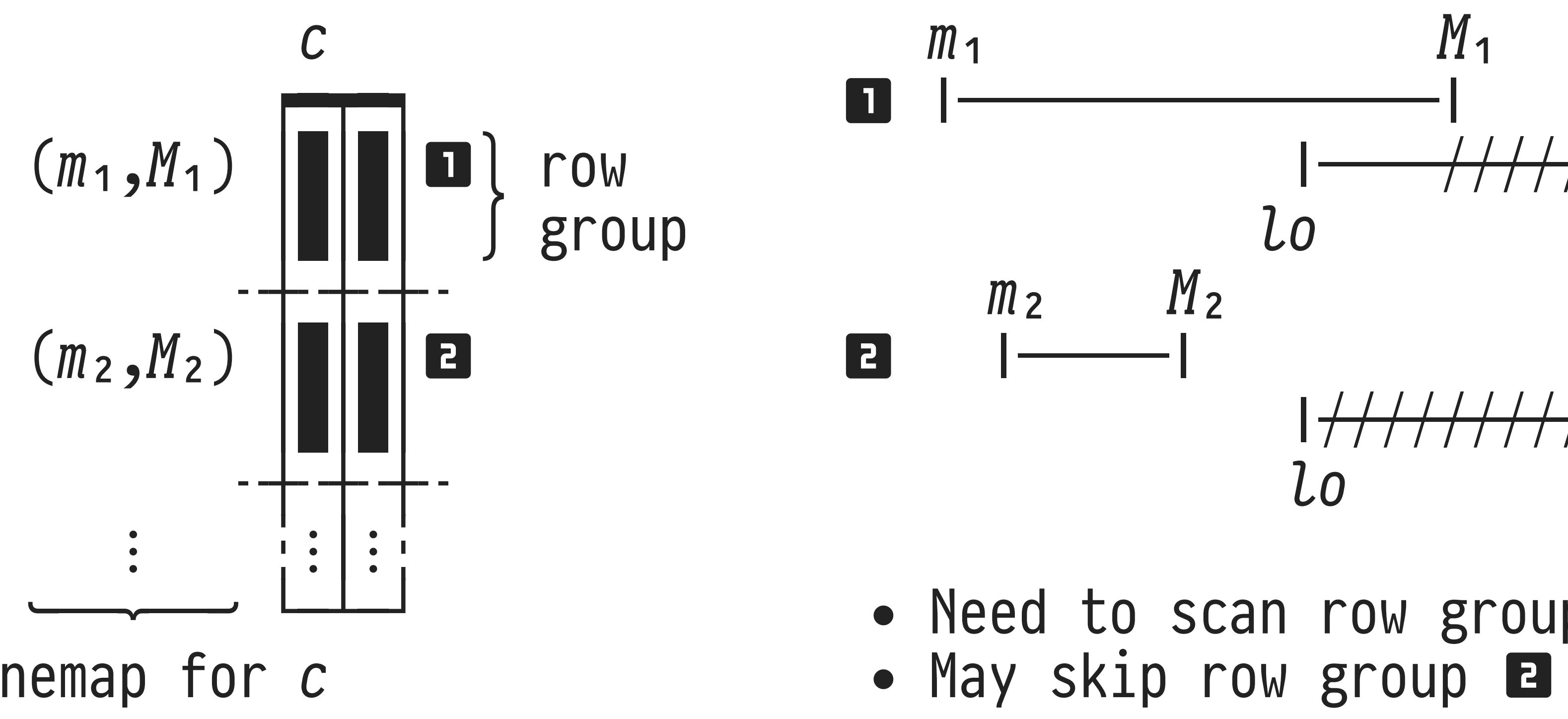
Indexes in DuckDB

DuckDB implements two kinds of **indexes:**

1. **Zonemaps** (also: *min-max indexes*) are integral part of the columnar table storage ▥.
   - Automatically created, always present for all columns.
   - Used when predicates are pushed down into Sequential Scan.
2. **Adaptive radix trees** (**ART**) are tree-shaped data structures maintained outside/in addition to table storage ▥+⁂.
   - Consume working memory space.
   - Require maintenance on table updates.
   - Created either
     - manually via SQL's DDL statement CREATE INDEX or
     - automatically when constraints UNIQUE, PRIMARY KEY, and FOREIGN KEY are declared for a table.

Zonemaps are baked into DuckDB's columnar table storage format:

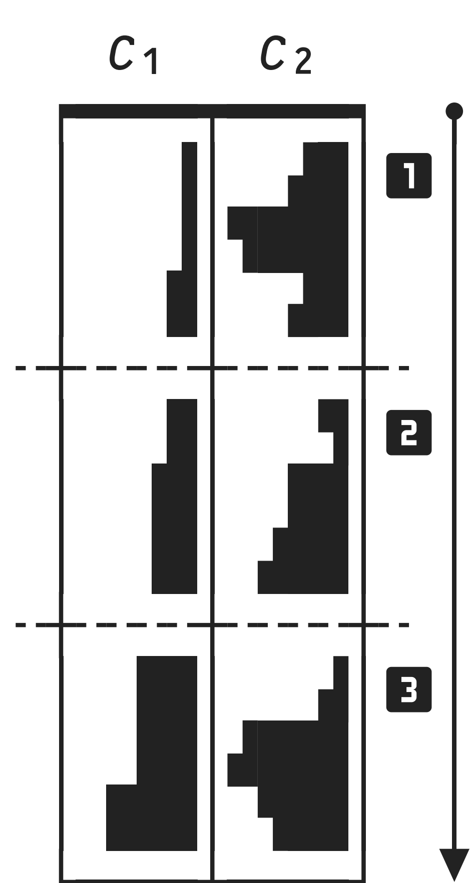
- Each column *c* is divided into *row groups* of 120K (122880) rows.
- Each row group holds a (*min*,*Max*) entry that—quite roughly—characterizes the contained values.
- Operator Sequential Scan can safely **skip row groups** for which a predicate $lo \leqslant c \leqslant hi$ will always fail (⧻):



- Need to scan row group 1 👎
- May skip row group 2 👍

Zonemaps and Column Ordering

Column ordering affects the effectiveness of zonemaps.



c_1: In each row group, the zonemap entries (*min*,*Max*) for c_1 have a small *span* |—|.

⇒ A large number of row groups will never satisfy $lo \leqslant c_1 \leqslant hi$. Can skip. 👍

c_2: All spans in unordered column c_2 are wide |———————| and cover the active domain of c_2.

⇒ Most (all) row groups will potentially contain hits for $lo \leqslant c_2 \leqslant hi$. No skipping. 👎

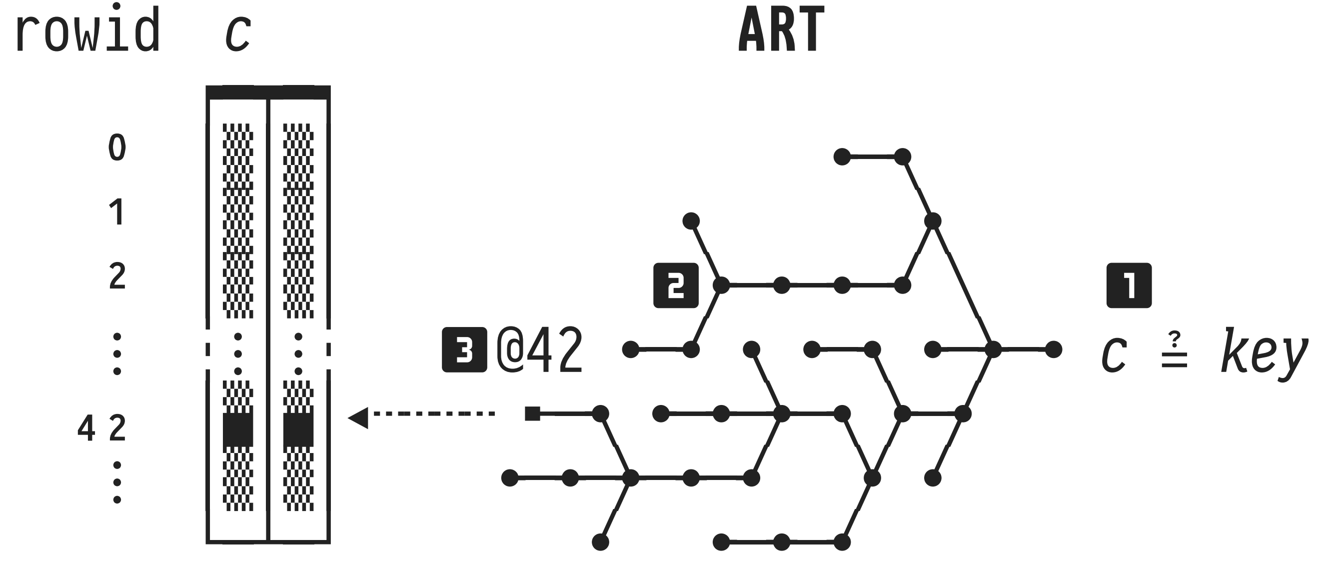
💡 Keeping base tables ordered may help predicate evaluation.

3 ┆ Adaptive Radix Tree (ART) Indexes

ART indexes are **ordered search tree structures** built to evaluate simple **equality and range predicates** (assume an ART on column *c*):

c = *key* ┆ c IN (key_1, key_2, …) ┆ c < *key* ┆ c > *key*

- ARTs and tables are separate. These extra data structures
  - use **row IDs** to refer to rows in their associated table,
  - must be created, then maintained under table updates, and
  - occupy space in working memory.



Index lookup:

1. Enter at root
2. Navigate ART to leaf node ─▪
3. Use rowid @42 to access row

Creating ART Indexes in DuckDB

1. **Manual:**

```
CREATE [UNIQUE] INDEX index ON table (column [, column, …]);
DROP INDEX [IF EXISTS] index;
```

   - **NB.** More indexes aid query evaluation but incur maintenance and space[1] overhead. A tradeoff in physical database design.

2. **Implicit** (supports constraint enforcement). DDL statement ❶ creates on a unique ART index on *t*(*c*) behind the scenes:

```
❶ CREATE TABLE t (…, c … PRIMARY KEY, …);  -- also: UNIQUE
❷ CREATE TABLE s (…, f … REFERENCES t(c), …);
```

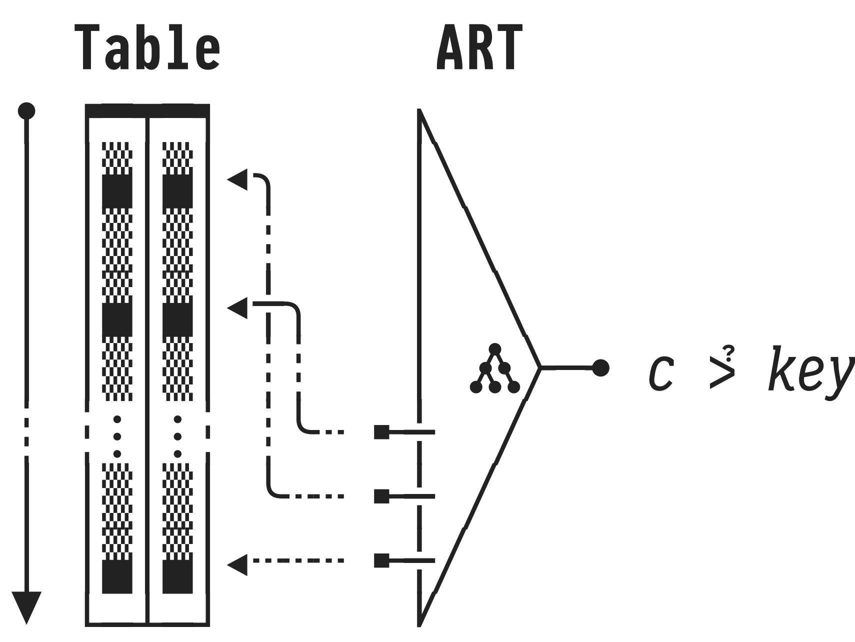
   - DDL statements like INSERT INTO *s* VALUES (…, *key*, …) or UPDATE *t* SET *c* = *key* lead to lookups $c \stackrel{?}{=} key$ in that index.

[1] As of DuckDB 1.4, memory occupied by ART indexes is under control of the buffer manager but cannot be evicted. Work to rectify this is underway as I type this.

Use the Index? Ignore the Index?

An index lookup predicate p may yield *multiple rows* in table t:

- $p \equiv c \stackrel{?}{=} key$: If c is not unqiue in t: a single leaf node ─▪ holds multiple rowids ···▸ ···▸ ···▸.
- $p \equiv c \stackrel{?}{>} key$: ❶ perform lookup $c \stackrel{?}{=} key$, ❷ access rows using the rowids in all leaves ▪ ▪ ▪ following the found leaf:



- Lookup finds one leaf (or adjacent leaves ▪ ▪ ▪).

#016

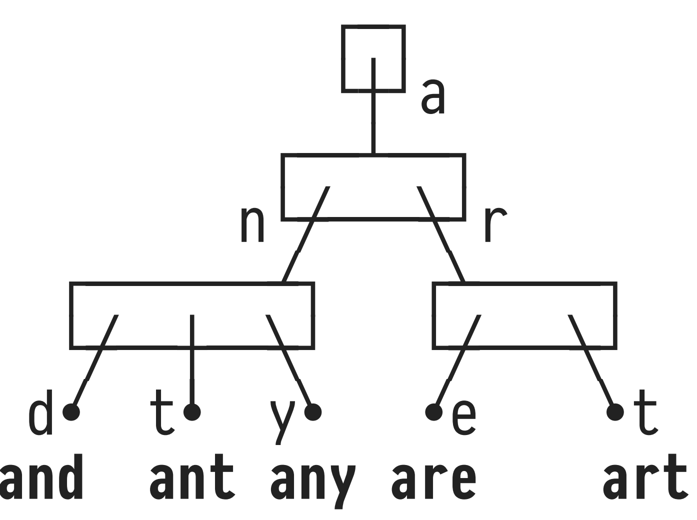
- Yet the rowids ···▸ may point all over table t. May need to "jump around" to collect rows. Violates memory locality.

Use index only if **selectivity** $sel(p)$ of predicate p is low:

$$0 \leqslant sel(p) \stackrel{\text{def}}{=} (\textbf{SELECT}\ \text{count}(*)\ \textbf{FILTER}\ (p)\ /\ \text{count}(*)\ \textbf{FROM}\ t) \leqslant 1$$

An ART on $t(c)$ uses the **bit-wise representation of values** in column c to organize itself. Values are *not* hashed or compared.

- Divide values into groups of s bits each (s is the *span*).
- During tree traversal, the next s bits of key determine the child node to descend into.



- Example: c :: text, span s = 8 bits.
- Values have k = 24 bits.

s bits   **ant** < **any**

ant $=_2$ 01000001 01001110 01001110
any $=_2$ 01000001 01001110 01011001

common prefix ≡ shared path in ART

ART Internals: Mapping Values to Bit Sequences

Map values to bit sequences whose **lexicographic order** properly reflects value sort order:

- **Unsigned ints:** use standard binary representation (little-endian machines: reverse byte order so that MSB comes first).
- **Signed ints** (in two's complement): flip the sign bit, then treat like unsigned ints.
- **IEEE 754 floats:** ❶ always flip the sign bit, ❷ if the sign bit was orignally set, now flip all bits. #017
- **Text strings:** map UTF-8 byte sequence to binary[2], end strings with NUL (values must not be prefixes of other values).
- **Composite values** $(v_1,\ldots,v_n)$: map the individual fields v_i, then *concatenate* the resulting bit sequences.
- **NULL**: increase bit length k (*e.g.*, by one bit/one byte) to accomodate the additional value.

[2] DuckDB uses function ucol_getSortKey() of the C/C++ library ICU to perform this mapping.

ART Internals: What is a Good Span?

- The height of ARTs depend on the value bit length k (*not* the number n of entries).
  - An ART for k-bit values has $\lceil k/s \rceil$ levels of inner nodes.
  - ART lookup and insert operations have complexity $O(k)$.

- ARTs *vs.* binary search trees (BSTs):
  - Height: if $n > 2^{k/s}$, ARTs have smaller height than BSTs ($\log_2(n)$).
  - Lookup: for values of k bits (k large), comparison ($<$) is $O(k)$. Complexity of BST lookup thus is $O(k \cdot \log_2(n))$.

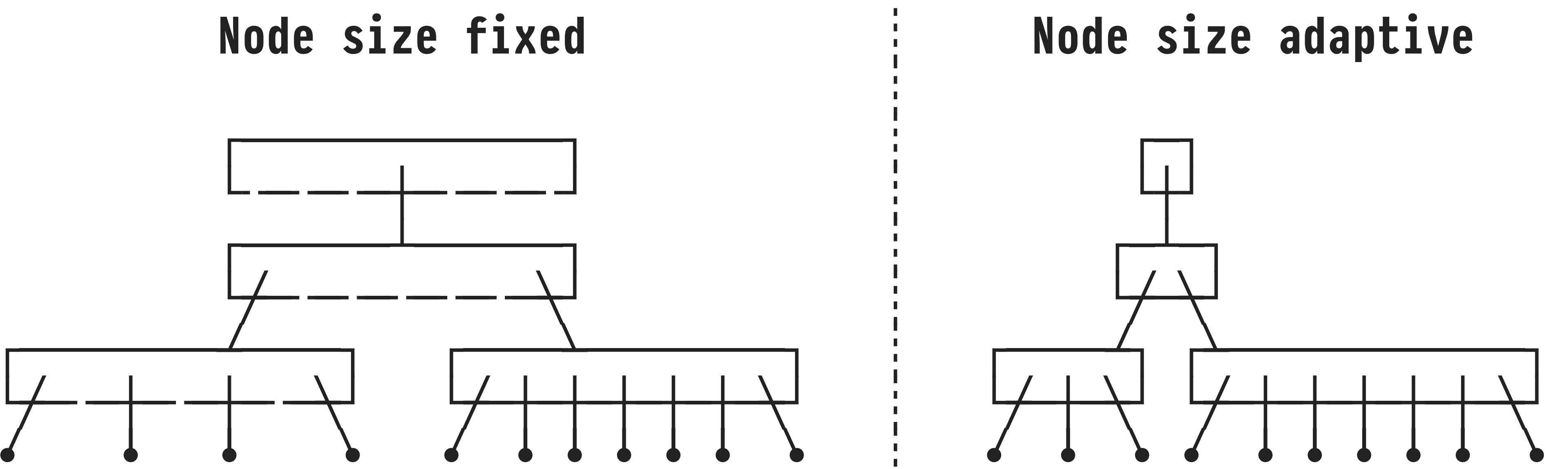
💡 Optimize ART **performance:** Work with a large span s! 👍

😱 But how about the **space usage** of such ARTs? 👎

ART Internals: Designing Inner Nodes

- A simple representation of inner ART nodes:
  - Array of 2^s child pointers. (Nodes would have **fixed size.**)
  - Use *s*-bit chunk of *key* to index array + access child node.

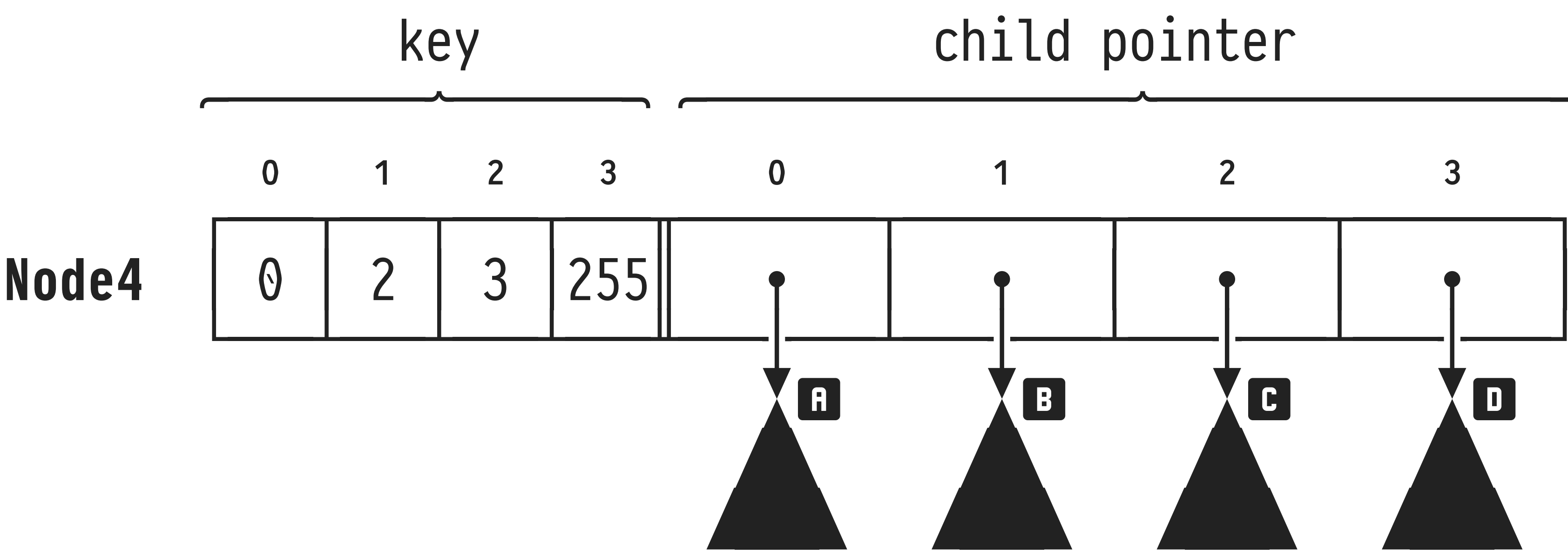
  - But: Most child pointers will be NULL (└──), space usage will be excessive if *s* is large (grows exponentially):



💡 Use a fixed large *s* but **adapt node size** (use variable fanout).

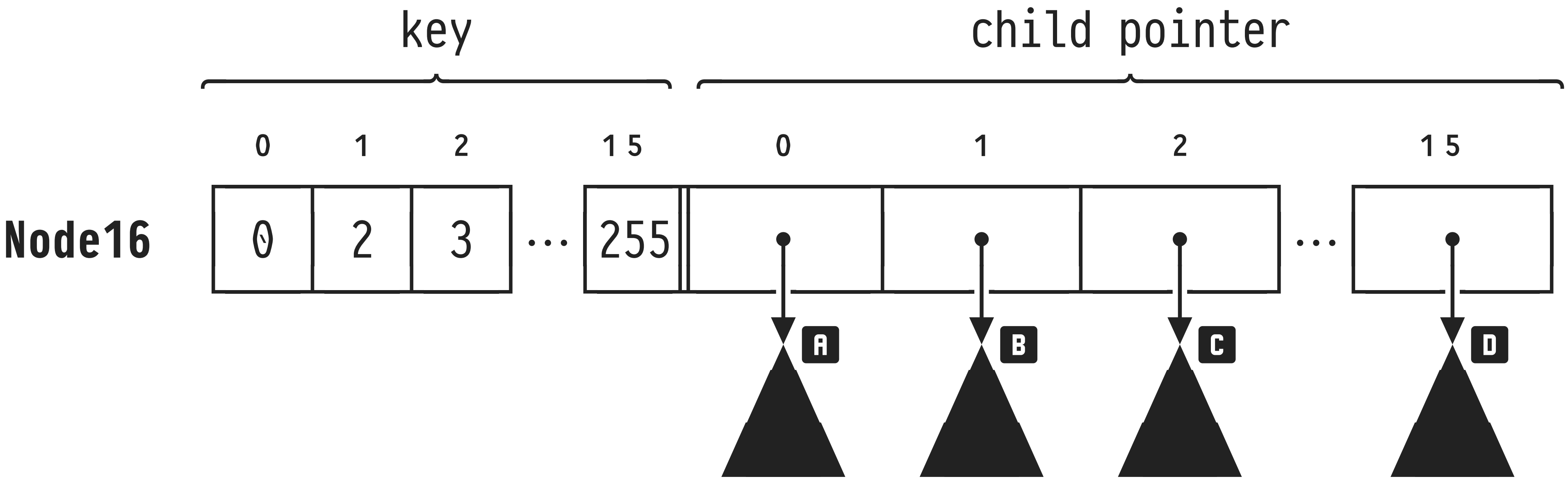
ART Internals: Four Inner Node Types ➊

- DuckDB: s = 8 (can cut values into bytes, admits large fanout).
- Choose inner node type based on the number of non-NULL child pointers: 4, 16, 48, or 256 (= 2^s).
  - On value insertion/deletion, switch to larger/smaller node type when node overflows/underflows.



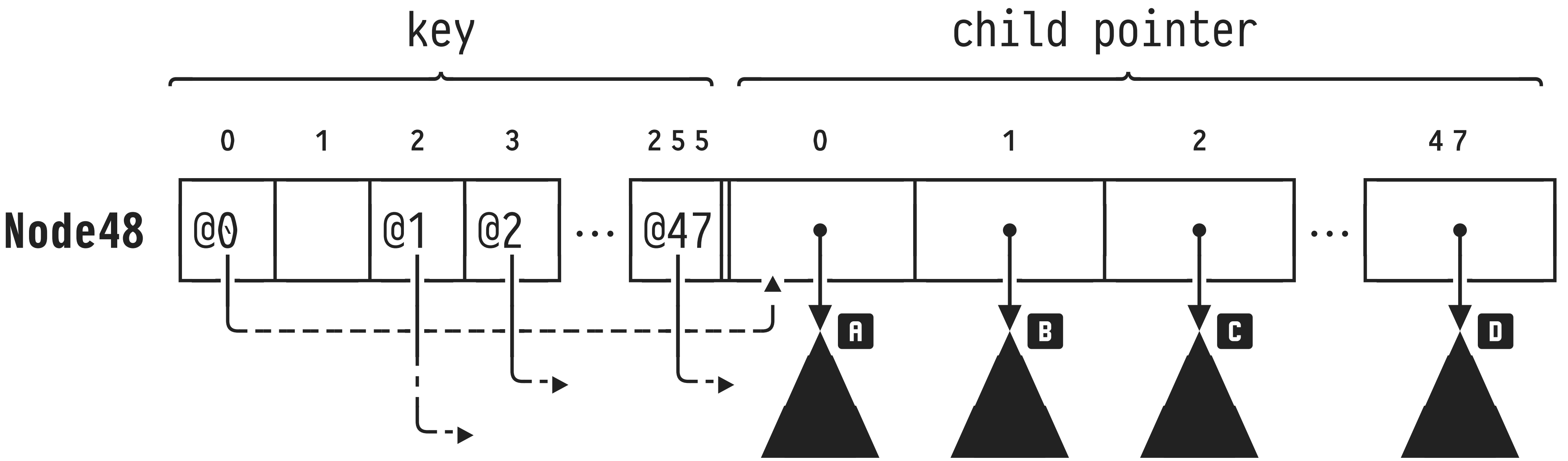
- **Node4:** Array of 4 keys is ordered (search left to right).
- Value/pointer pairs stored at corresponding indices 0…3.

ART Internals: Four Inner Node Types ❷



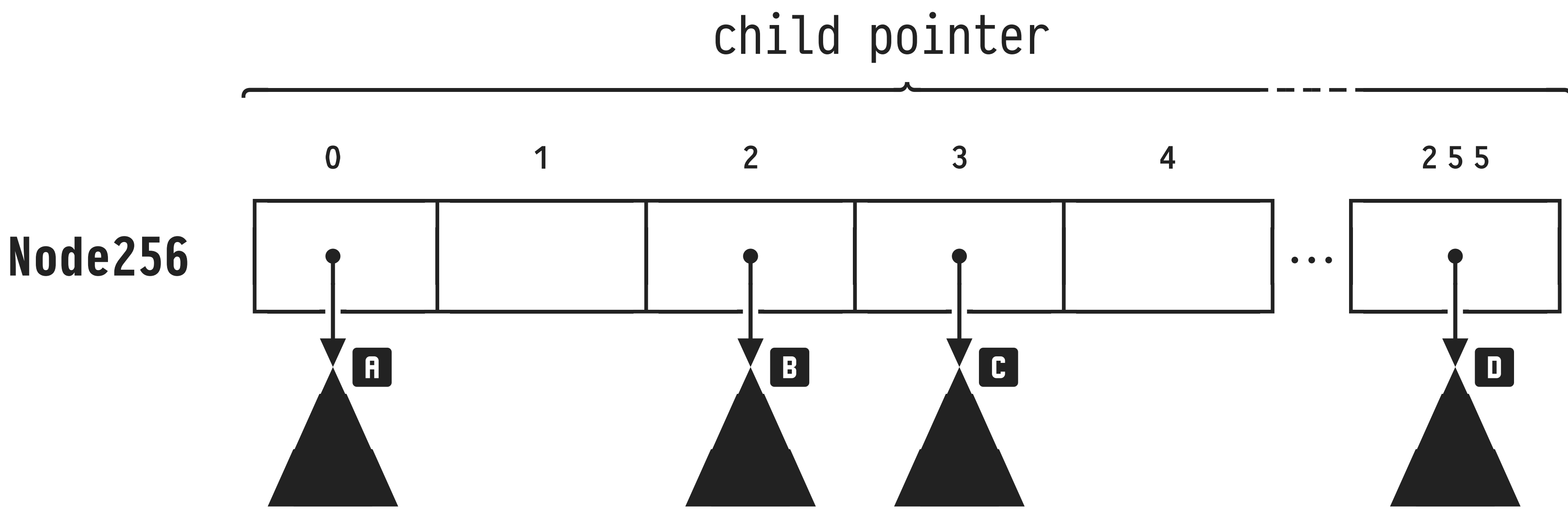
- **Node16:** stores 5…16 child/pointer pairs.
- Locate current *key* byte by binary search or parallel SIMD comparisons.

ART Internals: Four Inner Node Types ❸



- **Node48:** With 17…48 values in a node, searching the value array becomes expensive. Thus: do *not* store values. Instead:
  - Use current byte of *key* as index into array of 256 indices.
  - key[‹*key* byte›] holds the index @*i* ∈ {0…47} of the corresponding child pointer. Access pointer[@*i*].
- Saves space compared to an array of 256 8-byte pointers: 640B = 256B + 48 × 8B < 256 × 8B = 2048B.

ART Internals: Four Inner Node Types ❸

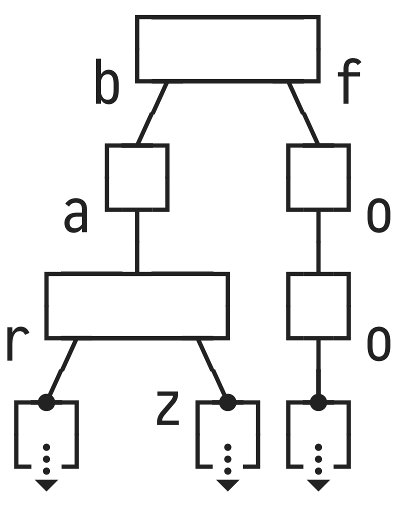


- **Node256:** If an inner node needs to hold 49…256 entries, invest 256 × 8B = 2kB to hold an array of 256 child pointers.
  - Simply return pointer[‹*key* byte›].

ART Internals: Leaf Nodes

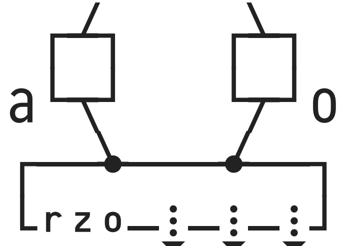
In an ART for *t(c)*, **leaf nodes** hold rowids that point to the rows in table *t* in which column *c* holds the search value *key*.

- Possible ART leaf designs:



(This is an ART for values **bar**, **baz**, **foo**.)

Single-entry leaves, each hold rowids (⋮) for one value *val*.



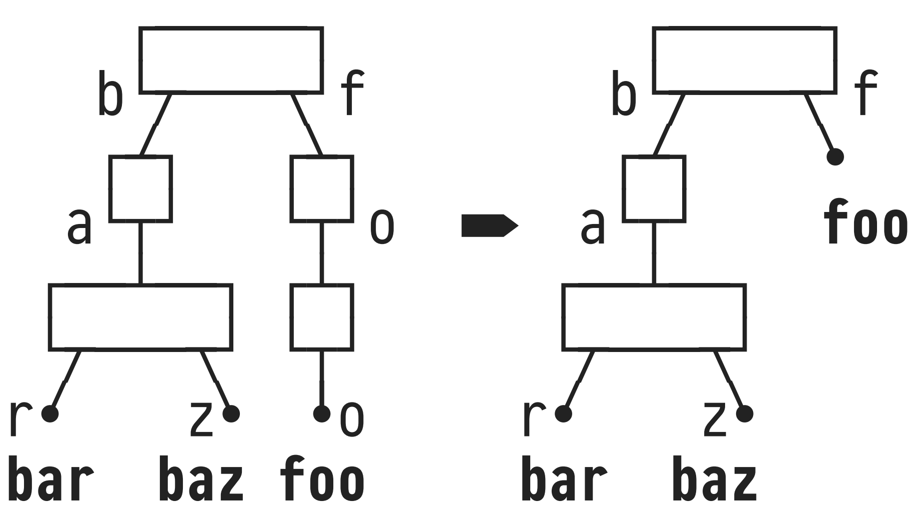
Multi-entry leaf, holds rowids for multiple values. DuckDB has Leaf7/Leaf15/Leaf256 much like Node4…256 (hold rowids, not pointers).

ART Optimizations: Lazy Expansion + Path Compression

To efficiently use space and save tree traversal effort during lookups, try to remove inner ART nodes—and thus **reduce tree height**—if possible.

Most effective if values are long (bit length k is large).

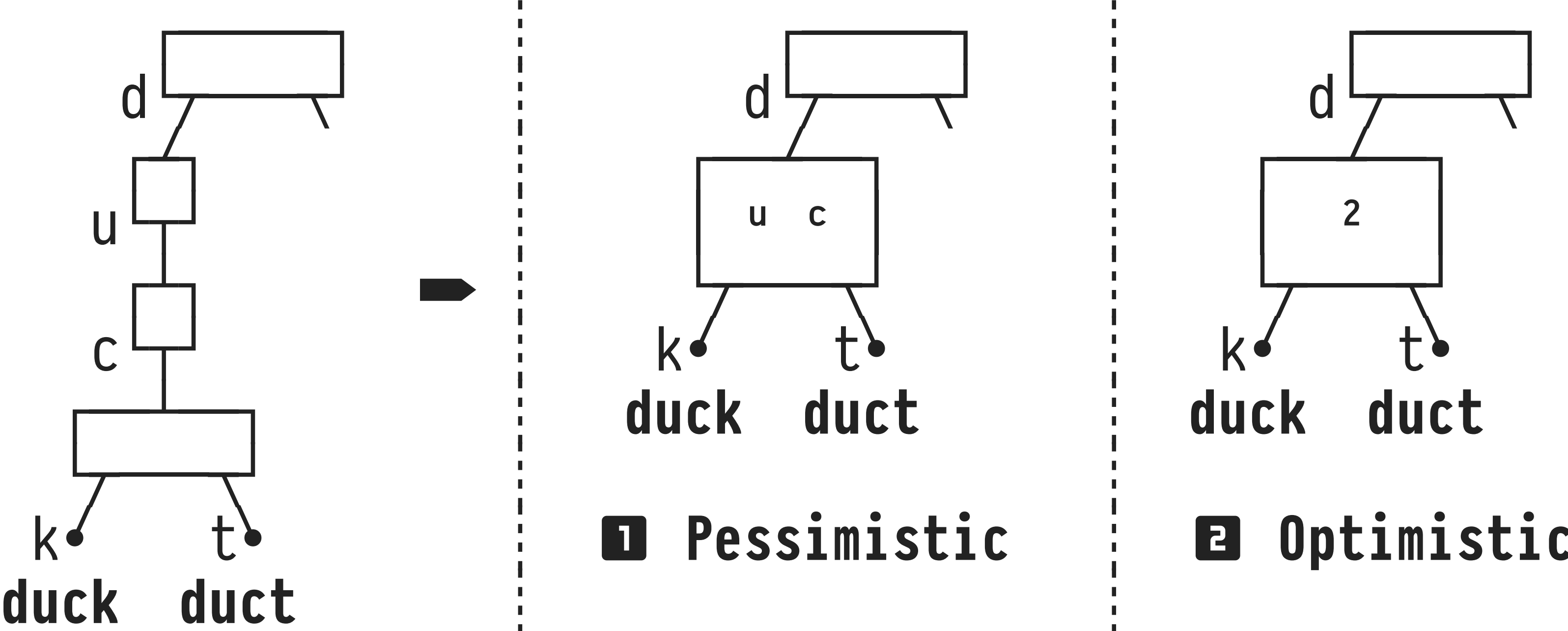
1. **Lazy expansion:** create inner nodes only if they are required to distinguish at least two leaf nodes.



- Saves two inner nodes on the path to leaf **foo**.
- Expand path if another leaf with prefix **f** is added.
- Path does not spell **foo**: store at leaf or in table.

ART Optimizations: Lazy Expansion + Path Compression

2. **Path compression:** remove inner nodes that have a single child.



- ❶ Inner node holds byte sequence (here: uc) of preceding single-child nodes. Compare this sequence against *key*.
- ❷ Inner holds count (2) of removed single-child nodes. Skip that many bytes in *key*. At a leaf, compare its value to *key* to ensure that traversal indeed reached the proper leaf.

ART: Index Lookup (Lazy Expansion + Pessimistic Path Compression)

```
search(node, key, depth):
 1 if node = NULL                                   search failed (findChild()
 2 ⌊ return NULL                                      returned NULL)
 3 if isLeaf(node)                                  reached leaf level?
 4 │ if leafMatches(node, key)                      reached the proper leaf?
 5 │ ⌊ return node                                    yes, success
 6 ⌊ return NULL                                      no, failure
 7 s ← byteSequence(node)                           byte sequence in inner node
 8 if s ≠ key[depth...depth+len(s)]                 does partial key match
 9 ⌊ return NULL                                      byte sequence?
10 depth ← depth + len(s)                           skip to next key byte
11 child ← findChild(node, key[depth])              key byte selects child
12 return search(child, key, depth+1)               descend into subtree
```

- Invoke with **search**(*node*: ‹*ART root*›, *key*: 'duck', *depth*: 0). (character positions 0 1 2 3 printed above 'duck')
- Returns leaf node that holds rowid(s) or NULL.

DuckDB is an OLAP DBMS, optimized to scan entire tables. Basic index support is present—but there certainly is unused potential.

- The following predicates *could* be evaluated by Index Scan:[3]

Predicate	Index Constellation	Index used by 🦆?
$c_1 \circledcirc val_1$	$t(c_1)$	👍 (if selective)
	$t(c_1,c_2)$ (composite)	👎
$c_1 \circledcirc val_1$ **AND** $c_2 \circledcirc val_2$	$t(c_1,c_2)$ (composite)	👎
	$t(c_1)$ and/or $t(c_2)$	👎
$c_1 \circledcirc val_1$ **OR** $c_2 \circledcirc val_2$	$t(c_1)$ and $t(c_2)$	👎
c_1 **LIKE** '*patt_rn%*'	$t(c_1)$	👎

Index support in DuckDB (as of version 1.4)

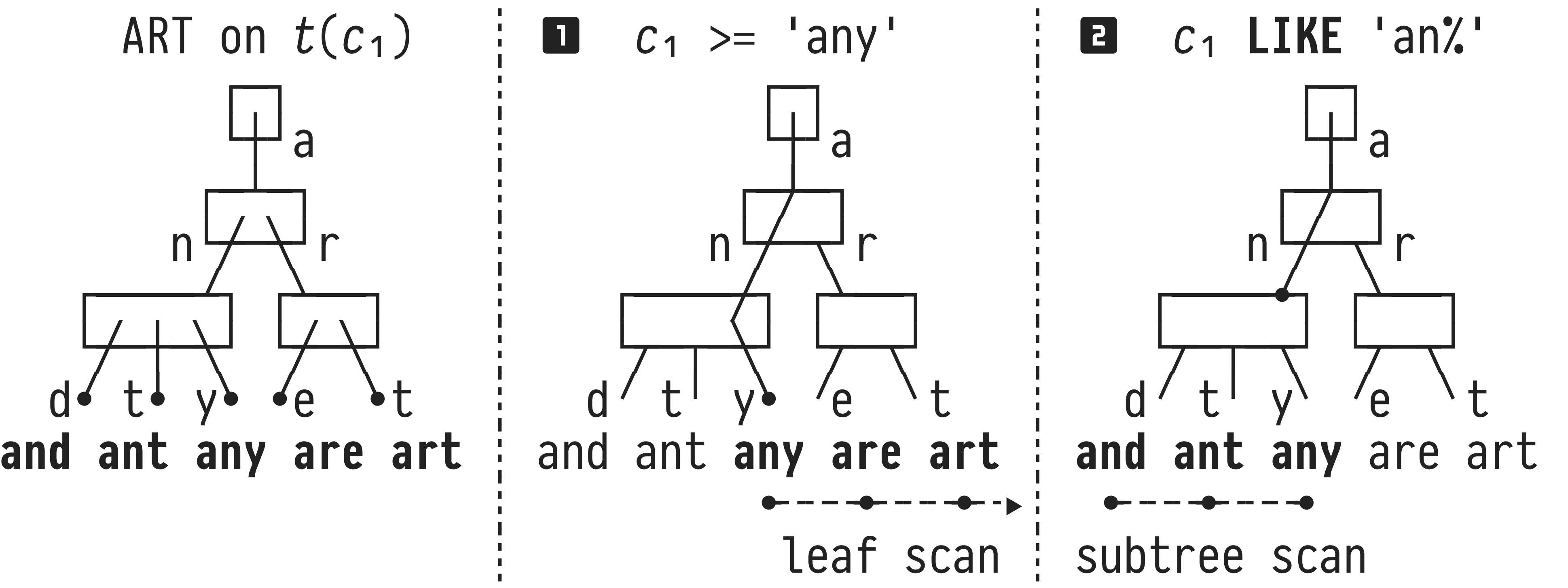
- (PostgreSQL🐘 is optimized to operate as an OLTP DBMS and can use indexes in all of these scenarios.)

[3] In this table, $\circledcirc$ represents a SQL comparison operator: <, <=, =, >=, >.

Index Traversals

Index traversals (beyond equality-based index lookups for a key *val*) can support **range predicates:** #019

- c_1 >= val_1 or c_1 >= val_1 **AND** c_1 <= val_2: perform lookup ─• for val_1, then scan leaves left to right •--▸ (until > val_2). ❶
- c_1 **LIKE** '*prefix%*': perform lookup ─• for *prefix*, hit inner node. Scan leaves in subtree •--•. ❷



Design and Implementation of DuckDB Internals

⑥

Query Execution Plans and Pipelining

April 7, 2026

Torsten Grust
Universität Tübingen, Germany

© 2025–2026 Torsten Grust, U Tübingen, Germany

1 ┆ Query Execution ≡ Plan Execution

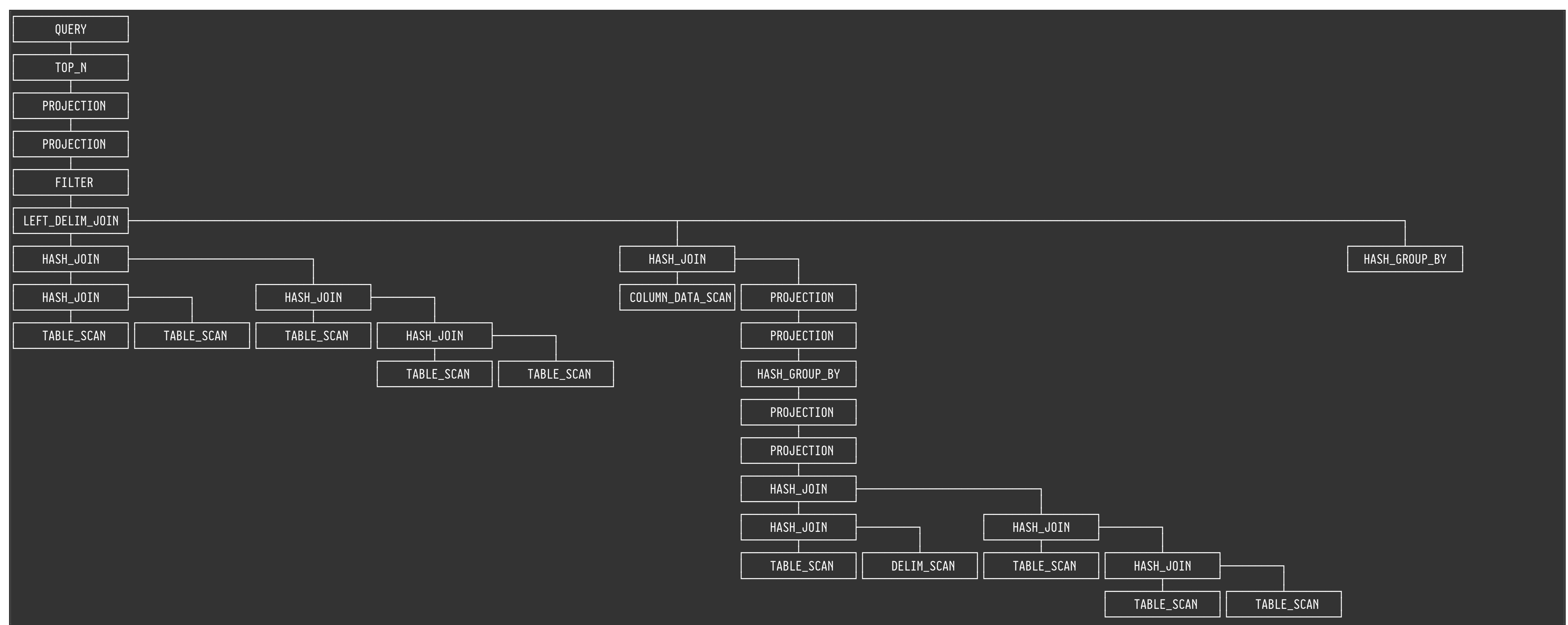
DuckDB translates SQL text into an **execution plan,** a tree-shaped[1] data flow network:

- 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):



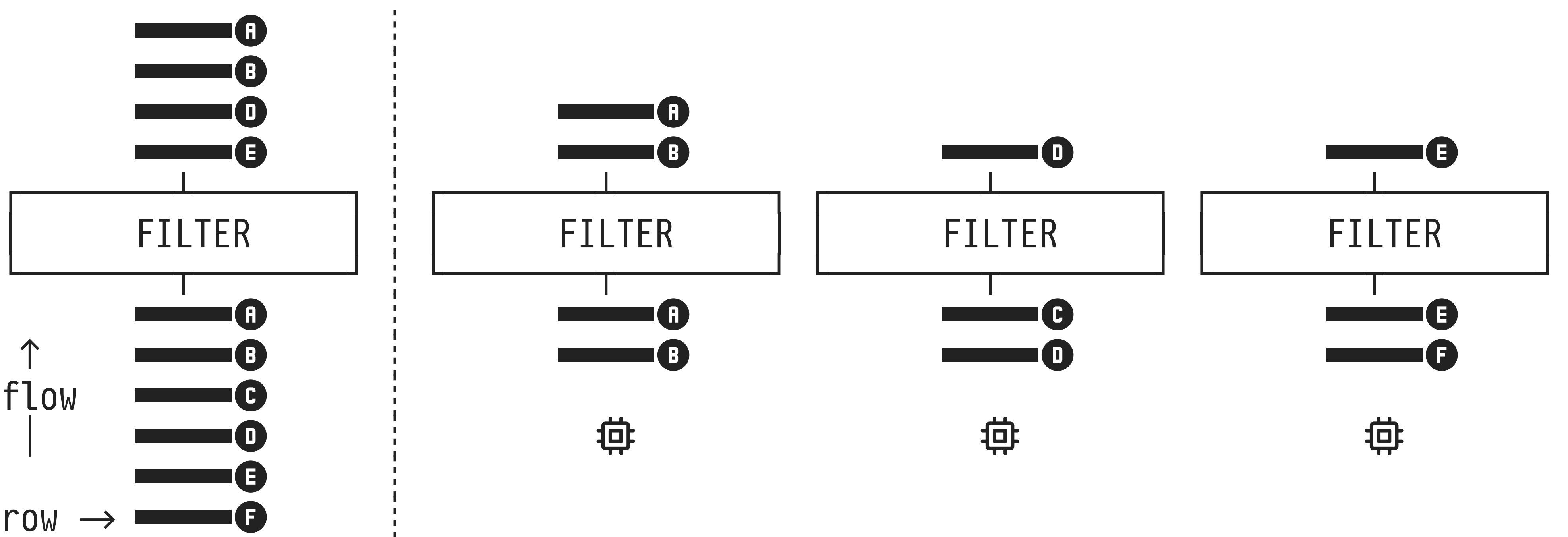
#020

- **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:

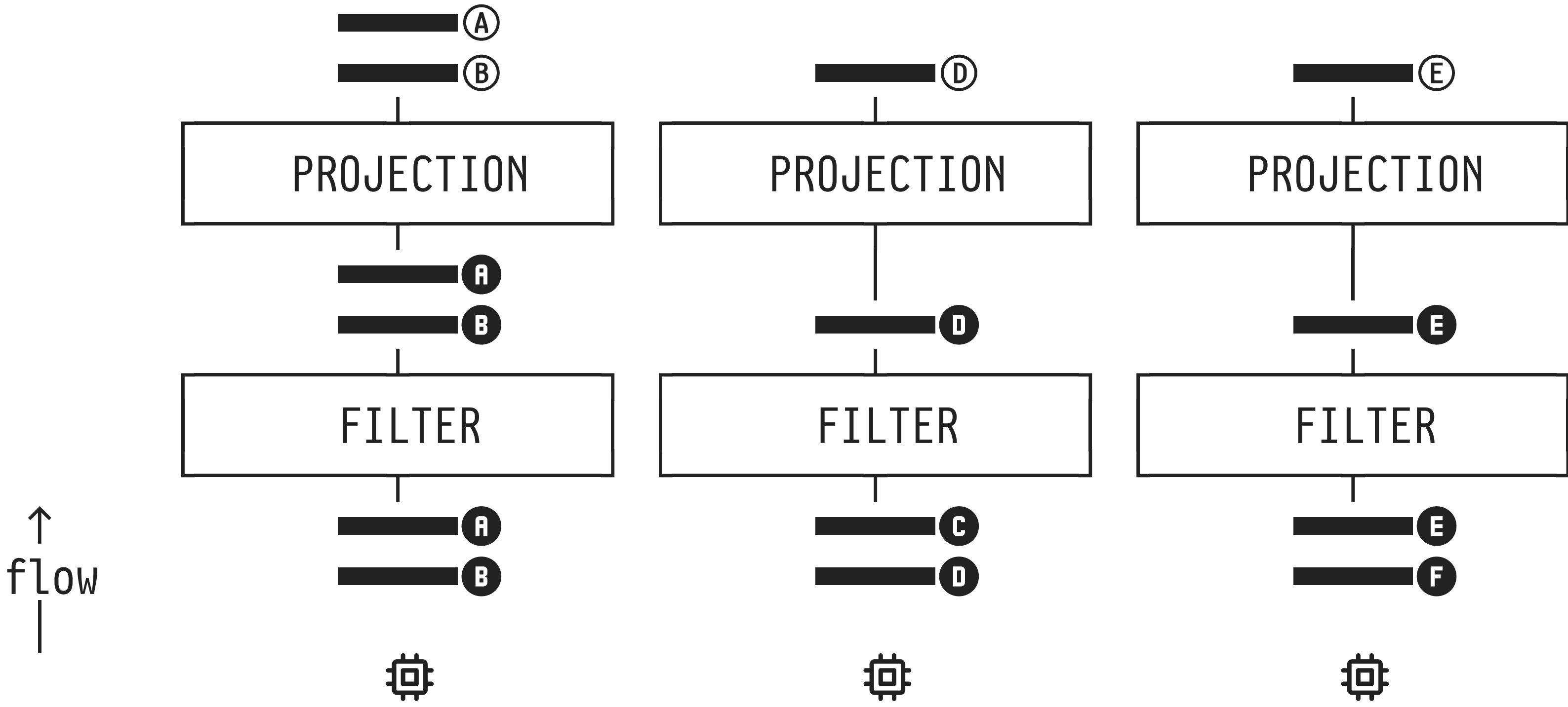
- Each thread operates on a partition of rows. Overall result is the (disjoint) union of all -local results.
- No inter-thread dependencies, operator implementation itself does need *not* to be aware of //ism.



Trivially Parallel Operators 2

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

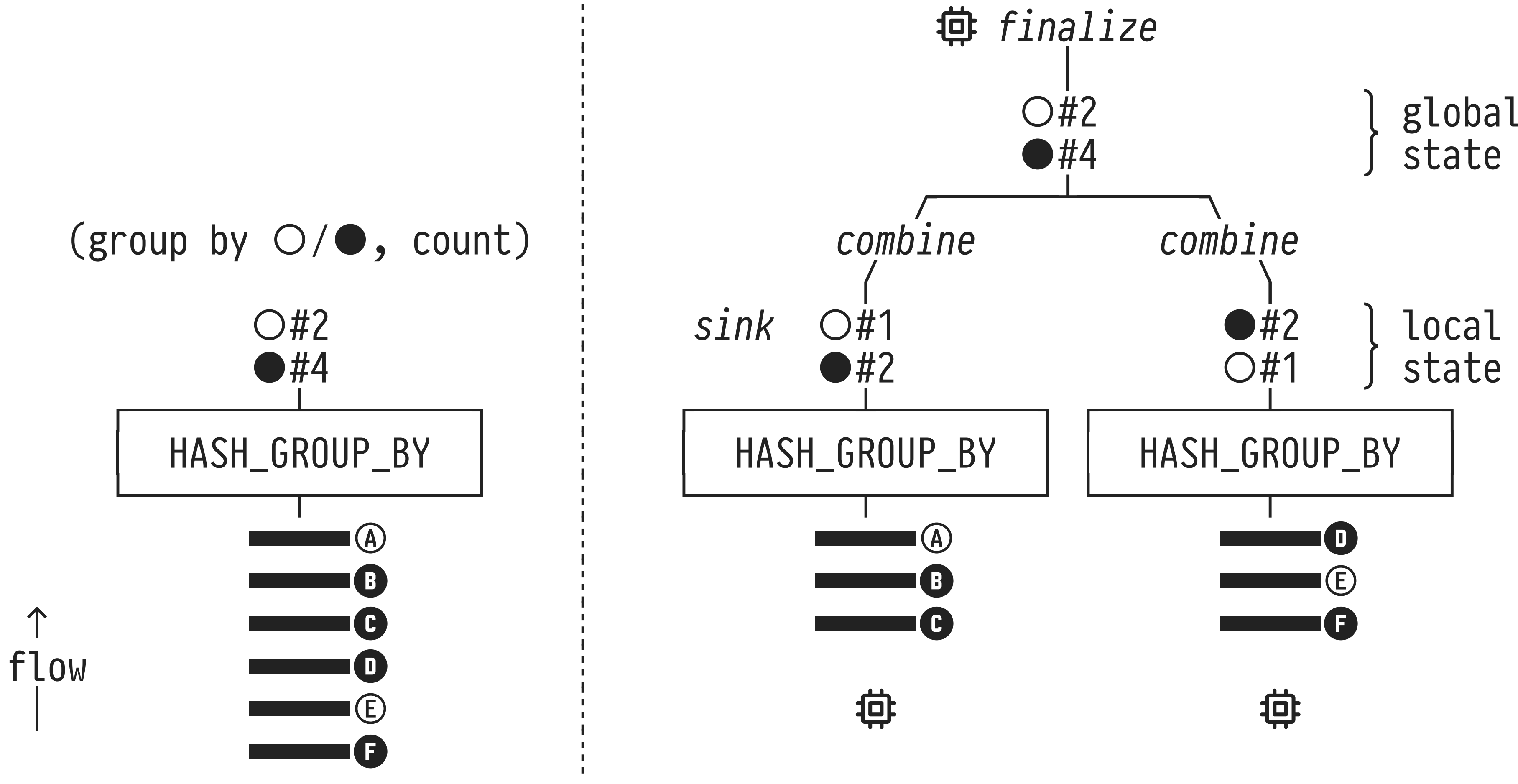
- Each thread/core runs its own instance of the pipeline.
- Data is passed between operators in **chunks** (2048 rows).[2]



[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**).

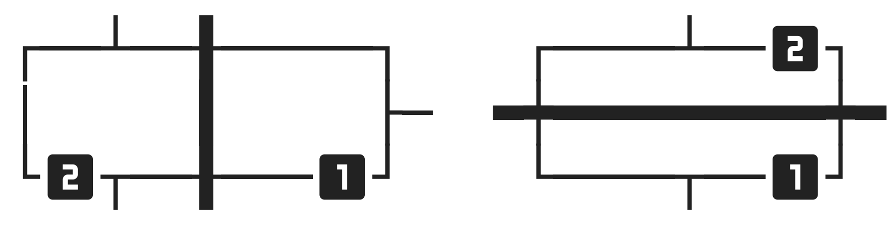
- Examples of sinks in DuckDB:
  - HASH_GROUP_BY (builds aggregate hash table)
  - Build side of HASH_JOIN (hash table for rhs join input)
  - ORDER_BY (local state: sorted run)
  - UNGROUPED_AGGREGATE (local state: partial aggregate)

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.

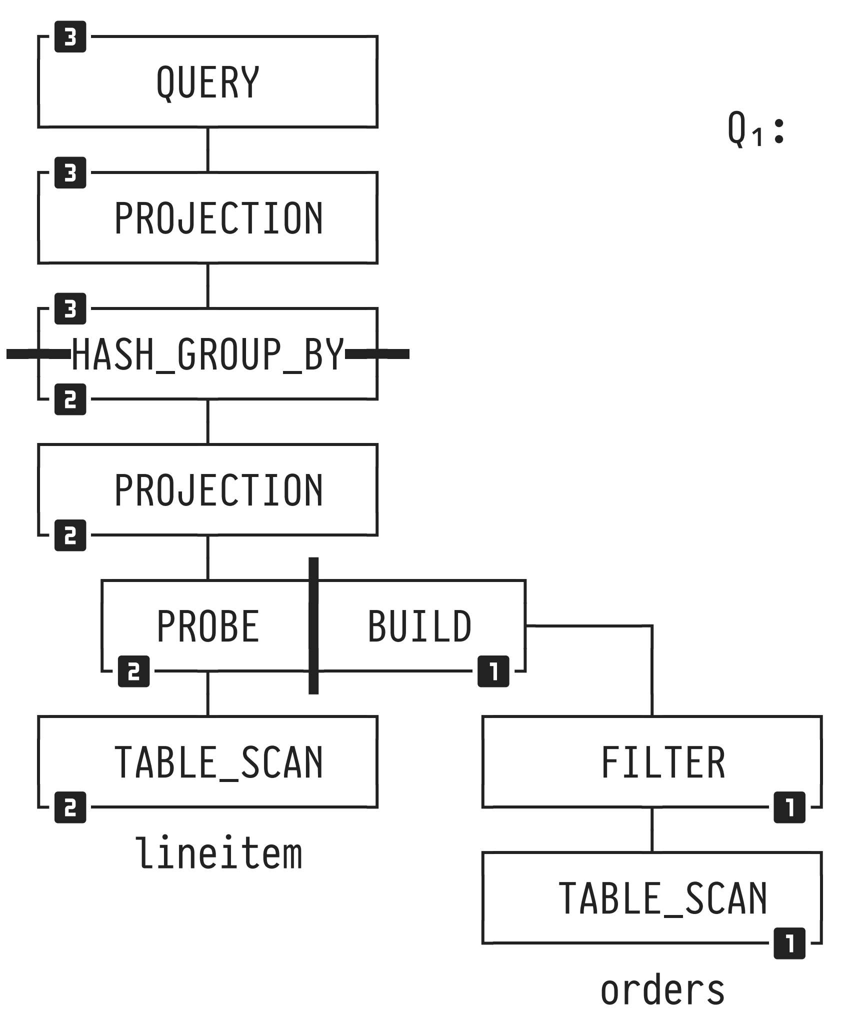


- Such operators introduce **pipeline dependencies:** ❶ < ❷ (pipeline ❶ must finalize before ❷ can read its output).

- 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]



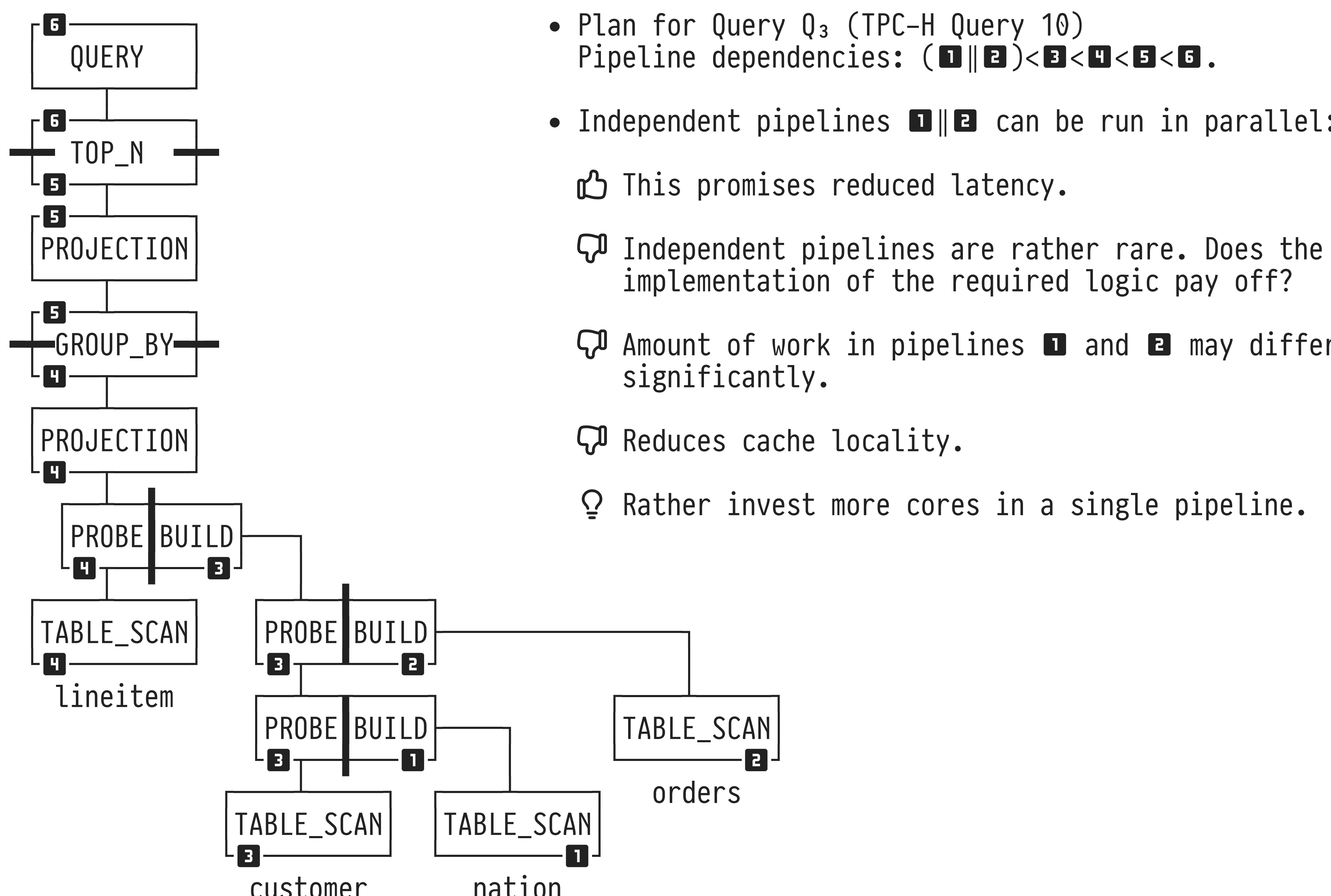
Q_1:

```
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: (❶‖❷)<❸<❹<❺<❻.

- Independent pipelines ❶‖❷ 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 ❶ and ❷ may differ significantly.

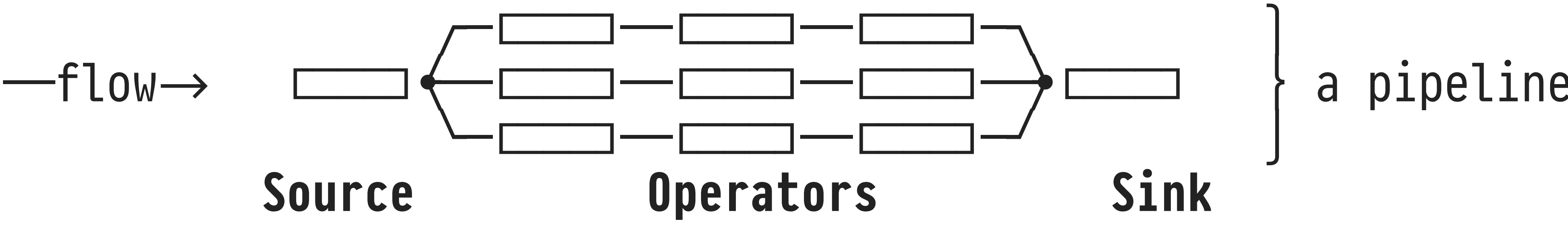
  👎 Reduces cache locality.

  💡 Rather invest more cores in a single pipeline.

5 ┆ Parallelism in a Pipeline

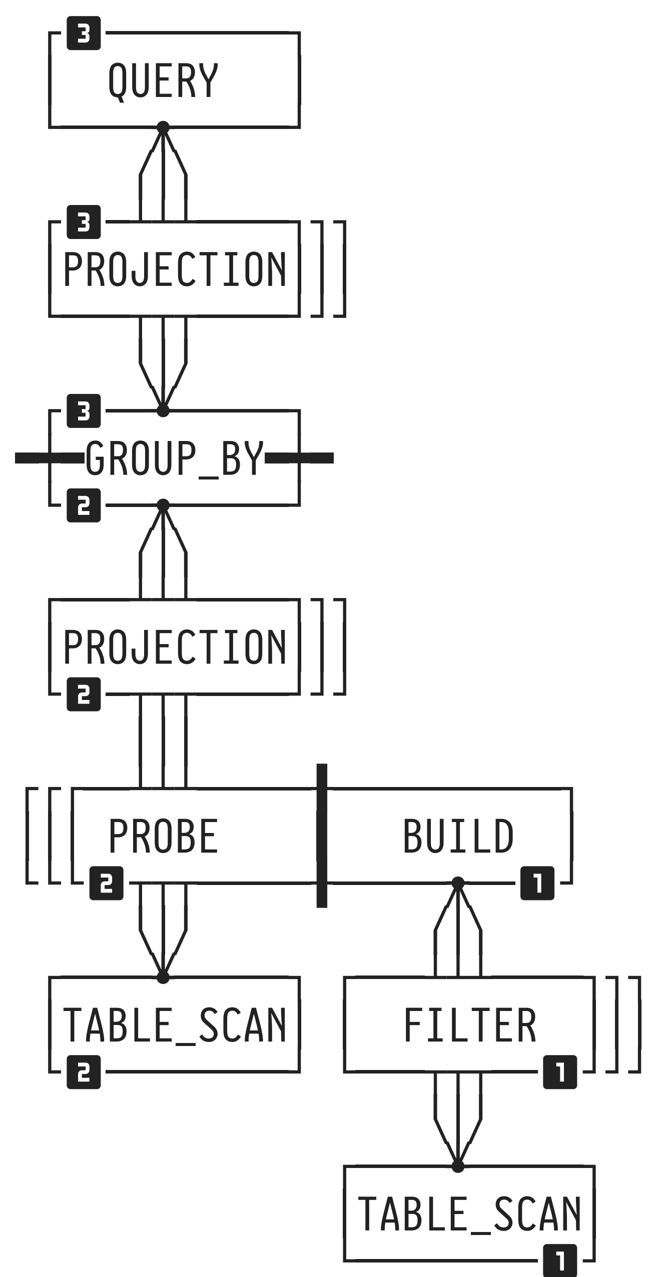
A pipeline in a DuckDB execution consists of three segments.

- **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



- 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 ❷ of HASH_GROUP_BY

- Pipeline **sinks** ⟩:
  1: BUILD phase of HASH_JOIN
  2: Phase ❶ 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)
```

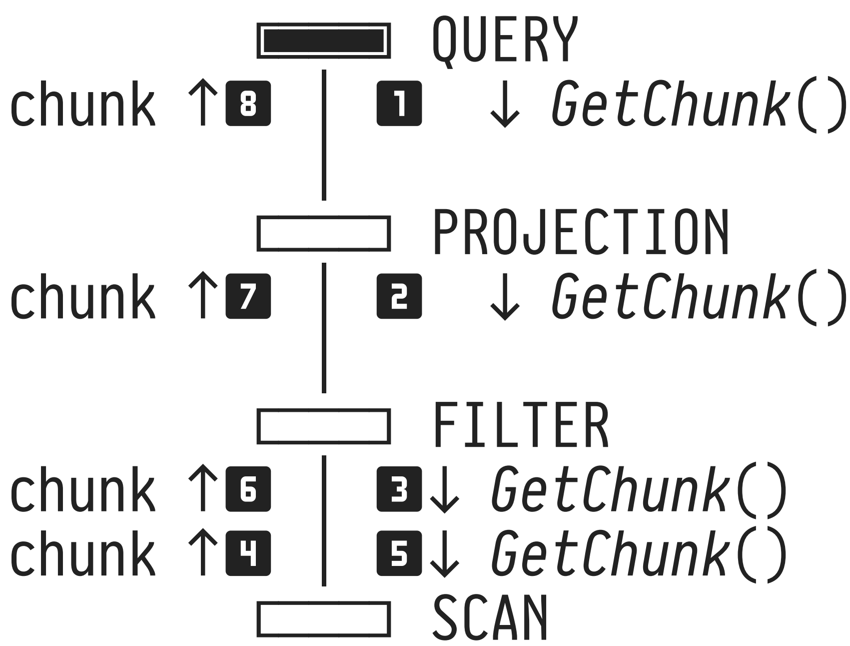
   The *Combine* phase of HASH_JOIN merges the ⚙-local hash tables into a global hash table that can be probed against.

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
```

Design and Implementation of DuckDB Internals

Vectorized Query Execution

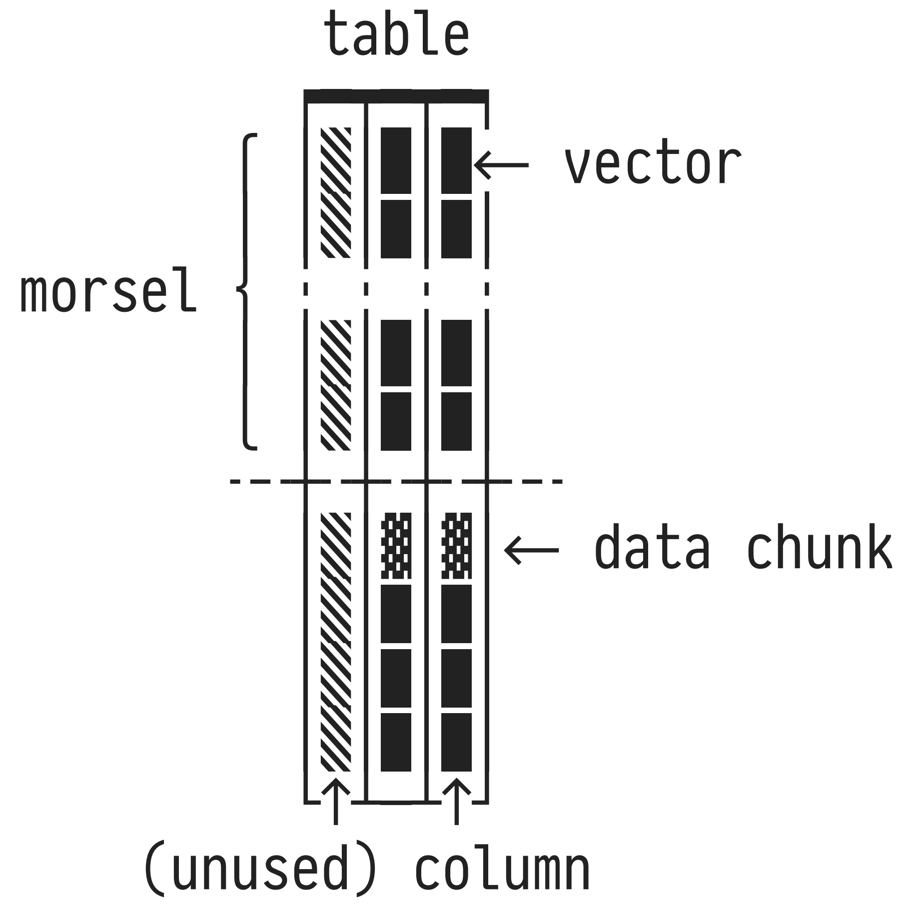
April 7, 2026

Torsten Grust
Universität Tübingen, Germany

© 2025–2026 Torsten Grust, U Tübingen, Germany

1 DuckDB Terminology Recap

Before we proceed, let us make sure that we agree on the data items processed by DuckDB's query engine:



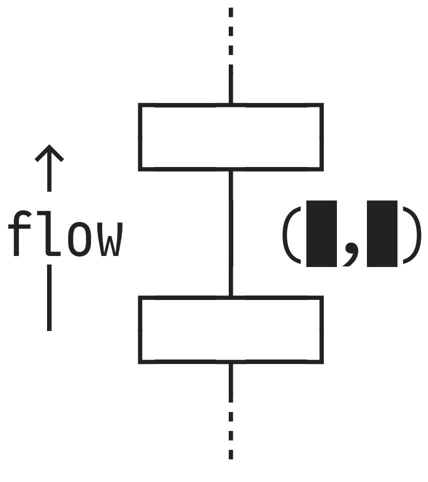
- **Tables** are organized in **columns** some of which may be unused in any given plan operator.
- Columns are split into **vectors**, each containing 2048 values.[1]
- A **data chunk** groups vectors of multiple columns, representing a horizontal slice of a table.
- Threads consume input data **morsel**-by-morsel, each consisting of 60 data chunks (= 122,880 rows).

[1] DuckDB's vector size can be configured at compile time (STANDARD_VECTOR_SIZE).

2 ┊ Passing Data Chunks Between Operators

During plan execution, operators process **one data chunk at a time**, then pass that chunk downstream:

Memory Location	Latency ⌛	Size (M2 Max)	… Can Hold
CPU Registers		$n \times 64$ bits	cell value(s)
L1 Cache	< 1 ns	(per ⚙) 192 KB	vector
L2 Cache	2.8 ns	(shared) 32 MB	data chunk
RAM ⚙	≈ 100 ns	16–96 GB	column(s)



- The passed **data chunks** will fit in the L2 cache. Operators can access intermediate results with low latency. 👍
- Passing **entire columns:** many intermediates will only fit into RAM. High latency. 👎
- Passing **single rows:** frequent context switches between operators. CPU overhead. 👎

3 ¦ DuckDB Vectors: Logical vs. Physical

To form a horizontal table slice, data chunks (█,…,█) contain a **vector** for each relevant column.

- Vector █ (**logical**): sequence of 2048 values of a *single type*.
- DuckDB exploits regularity in value sequences to implement a family of compressed **physical vector representations:**

	FLAT	CONSTANT	DICTIONARY	SEQUENCE
phys	[1 9 0 4 2]	[42] *const* [5] *len*	0 [DE], 1 [NL] *dict* [0 1 0 0 1] *sel*	[0] *base* [10] *inc*
log	[1 9 0 4 2]	[42 42 42 42 42]	[DE NL DE DE NL]	[0 10 20 30 40]

DuckDB Vectors: Selection Vectors and Validity Masks

Any vector is associated with a **selection vector** and **validity mask** used to permute/filter its value sequence and encode NULLs (□):

phys **vector** | ❶ **selection vector** | ❷ **validity mask** | log **vector**

	phys vector	sel		val	log vector
0	8	1	1	1	1
1	1	1	1	1	1
2	5	4	3	1	3
3	9	0	8	1	8
4	3	2	5	0	□

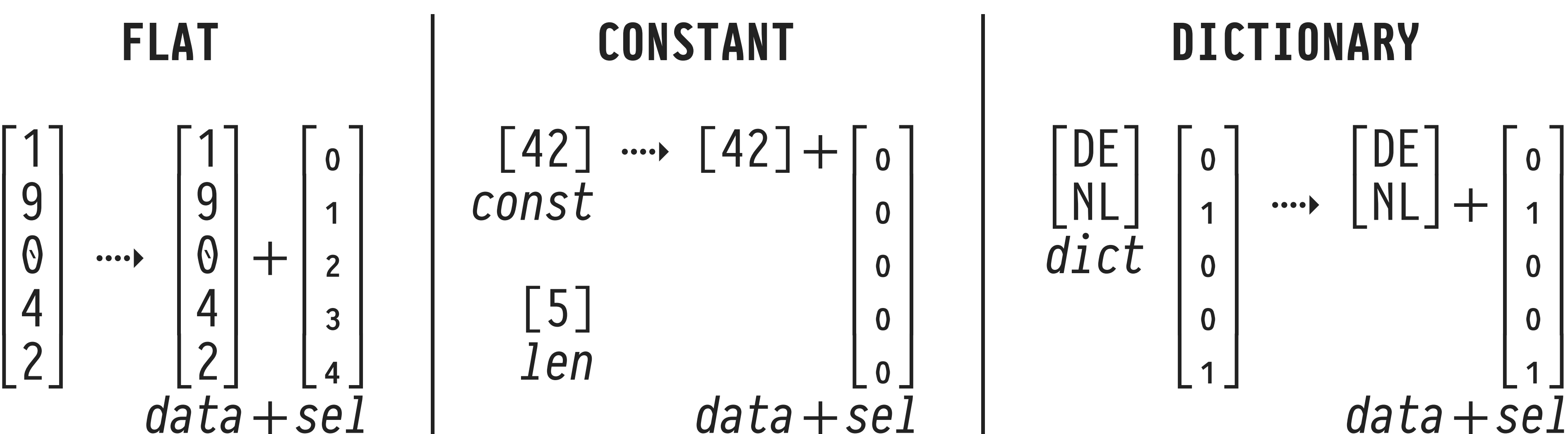
- Selection vectors contain vector indices ∈ {0,…,2047}. Can help to avoid copying—potentially sizable—values after a sorting or filtering operation.
- Validity masks contain Booleans (0/1).

DuckDB Vectors: Unified Representation

DuckDB aims to push vectors in compressed physical representation from operator to operator. However, *handling all representations everywhere* would lead to combinatorial code explosion.

- Convert to (····▸) and then process a **unified representation** comprising a *data vector*+*selection vector* pair.

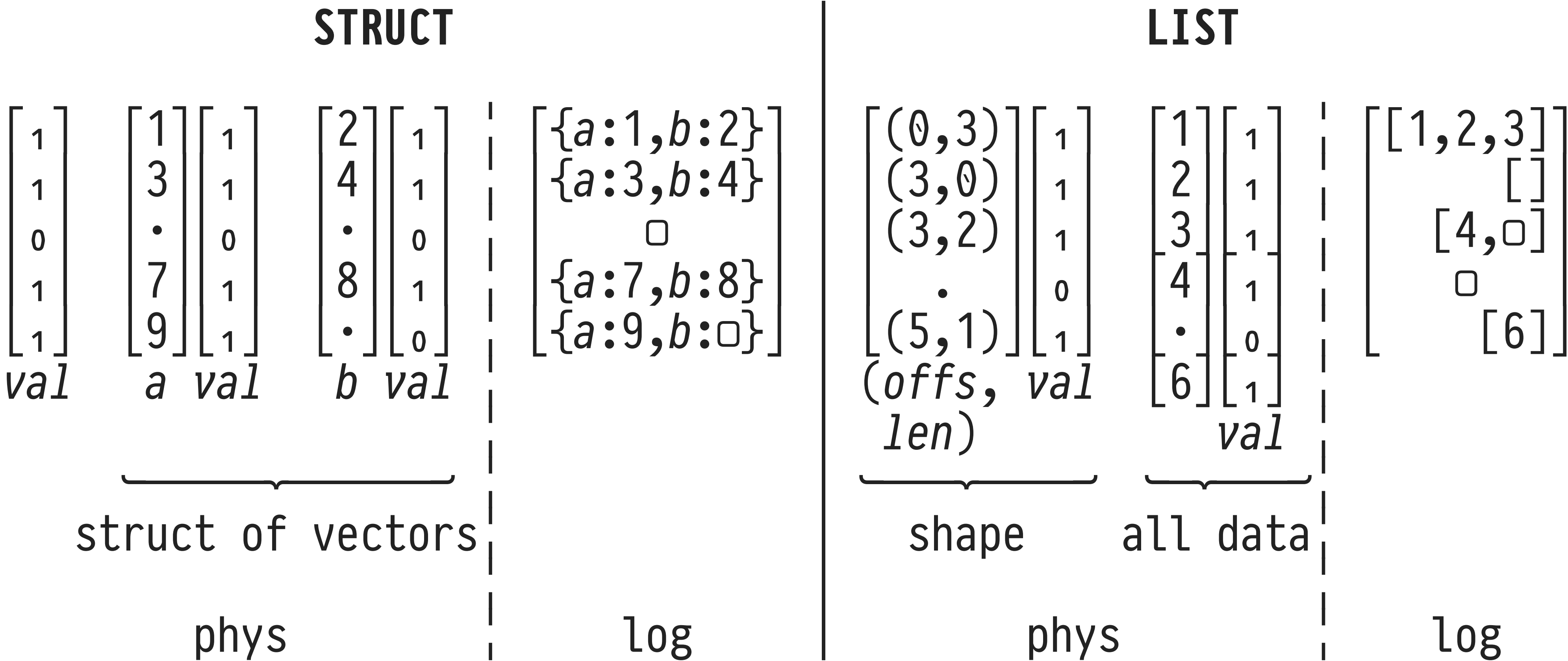
  NB. This conversion does *not* involve value copying:[2]



[2] SEQUENCE vectors are first expanded into FLAT vectors, then converted into the unified representation.

DuckDB Vectors of Complex Values (Structs {…}, Lists […])

- **Vectors of structs** are represented as structs of vectors.
- **Vectors of lists** store list shapes and data values separately, combine data of all lists into a single data vector.



- Nested types: apply representation schemes recursively.

#025

4 ┆ Super-Specific Vector Code

Q: How does DuckDB perform a binary operation on vectors `l` and `r`?

That depends on

- ❶ the kind of operation (e.g., arithmetics: `+`, `*`, …),
- ❷ value types (`int`, `float`, …),
- vector representations of ❸ `l` and ❹ `r` (`FLAT`, `CONST`, …).

💡 Branch on ❶…❹ *before* we enter the tight loop that scans the values in `l` and `r`:

- Avoids repeated tests (with identical outcome),
- leads to *tight inner loops* that feature less branches.

Super-Specific Vector Code

```
switch (❶ op)
  case +: switch (❷ type)
  | case int: switch (❸ l.repr)
  | | case FLAT:     switch (❹ r.repr)
  | | | case FLAT:     res.repr = FLAT;
  | | | |              for (i = 0; i < count; i++)
  | | | |              [ res.data[i] = l.data[i] + r.data[i];
  | | | [              break;
  | | | case CONSTANT: res.repr = FLAT;
  | | | |              for (i = 0; i < count; i++)
  | | | |              [ res.data[i] = l.data[i] + r.const;
  | | | [              break;
  | | [ ⋮
  | | case CONSTANT: switch (❹ r.repr)
  | | | case FLAT:     res.repr = FLAT;
  | | | |              for (i = 0; i < count; i++)
  | | | |              [ res.data[i] = l.const + r.data[i];
  | | | [              break;
  | | | case CONSTANT: res.repr = CONSTANT;
  | | | |              res.const = l.const + r.const;
  | | | [              break;
  | | [ ⋮
  | [ ⋮
  | case float: ...
  [ ⋮
  case *: ...
  ⋮
```


Super-specific code section: **Add** the **constant** vector l of **integers** to **flat** vector r

Super-Specific Vector Code

Previous slide does not handle **sel**ection vector/**val**idity mask (each leaf of the decision tree requires code much like this):

```
for (i = 0; i < count; i++)      /* code for leaf ❶: +, ❷: int, ❸: FLAT, ❹: FLAT */
  lindex = l.sel[i];
  rindex = r.sel[i];
  if (l.val[lindex] && r.val[rindex])
    res.data[i] = l.data[lindex] + r.data[rindex];
    res.val[i]  = 1;
  else
    res.val[i]  = 0;
```

DuckDB aims to control the resulting C++ code volume:

1. Convert vectors l, r to **unified representation**, process these by generic loops (cuts lower levels of).
2. Use **C++ templates** to generate code at *DBMS compile time*.[3]

[3] A different breed of DBMS—for example Umbra —generates and compiles C++/IR/machine code at *query translation time*. These query-specific code fragments are then linked with the DBMS and invoked at runtime.

Deep Dive Into DuckDB C++ Code: Vector Operations ❶

```
SELECT i = 42              -- i: SEQUENCE (base=1, inc=1), 42: CONST
FROM   generate_series(1, 100) AS _(i)
```

Trace how DuckDB vectorizes the evaluation of expression i = 42:

```
ExpressionExecutor::Execute( =(i, 42) , …, count, result)
```

```
void ExpressionExecutor::Execute(
  const BoundComparisonExpression &expr, …, idx_t count, Vector &result) {
  ⋮
  Execute(*expr.left,  …, count, left);   ⇝ left  ≡ [1|1]ˢ    (SEQUENCE)
  Execute(*expr.right, …, count, right);  ⇝ right ≡ [42]ᶜ     (CONSTANT)

  switch (expr.GetExpressionType() ❶) {
  case ExpressionType::COMPARE_EQUAL:
    VectorOperations::Equals(left, right, result, count);
    break;
  case ExpressionType::COMPARE_NOTEQUAL:
    VectorOperations::NotEquals(left, right, result, count);
    break;
  ⋮
  }
}
```

DuckDB 1.4 source: src/execution/expression_executor/execute_comparison.cpp

Deep Dive Into DuckDB C++ Code: Vector Operations ❷

VectorOperations::Equals([1|1]ˢ, [42]ᶜ, result, count)

```
void VectorOperations::Equals(
  Vector &left, Vector &right, Vector &result, idx_t count) {
    ComparisonExecutor::Execute<duckdb::Equals>(left, right, result, count); ⓐ
}
```

```
struct ComparisonExecutor {
private:
  template <class T, class OP>
  static inline void TemplatedExecute(
    Vector &left, Vector &right, Vector &result, idx_t count) {
      BinaryExecutor::ExecuteSwitch<T, T, bool, OP>(left, right, result, count); ⓒ
  }
public:
  template <class OP>
  static inline void Execute(
    Vector &left, Vector &right, Vector &result, idx_t count) {
    ⋮
    switch (left.GetType().InternalType() ❷) {
    ⋮
    case PhysicalType::INT64:
      TemplatedExecute<int64_t, OP>(left, right, result, count); ⓑ
      break;
    ⋮
    }
  }
}
```

DuckDB 1.4 source: src/common/vector_operations/comparison_operators.cpp

Deep Dive Into DuckDB C++ Code: Vector Operations ❸

```
BinaryExecutor::ExecuteSwitch<int64_t, int64_t, bool, …, duckdb::Equals, …> (
  [1|1]ˢ, [42]ᶜ, result, count, …)
```

```
struct BinaryExecutor {
⋮
  template <class LEFT_TYPE, class RIGHT_TYPE, class RESULT_TYPE, …, class OP, …>
  static void ExecuteSwitch(
    Vector &left, Vector &right, Vector &result, idx_t count, …) {
      auto left_vector_type = left.GetVectorType();              ⇝ SEQUENCE_VECTOR
      auto right_vector_type = right.GetVectorType();            ⇝ CONSTANT_VECTOR
      if (left_vector_type  == VectorType::CONSTANT_VECTOR ❸ &&
          right_vector_type == VectorType::CONSTANT_VECTOR ❹) {
        ExecuteConstant<LEFT_TYPE, RIGHT_TYPE, RESULT_TYPE, …, OP, …>(
          left, right, result, …);
      }
      ⋮
      else {
        ExecuteGeneric<LEFT_TYPE, RIGHT_TYPE, RESULT_TYPE, …, OP, …>(
          left, right, result, count, …);
      }
  }
⋮
}
```

DuckDB 1.4 source: src/include/duckdb/common/vector_operations/binary_executor.hpp

Deep Dive Into DuckDB C++ Code: Vector Operations ❹

No specific code section for vector operation SEQUENCE = CONSTANT: first transform left and right into **unified representation.**

```
BinaryExecutor::ExeuteGeneric‹int64_t, int64_t, bool, …, duckdb::Equals, …› (
  [1|1]ˢ, [42]ᶜ, result, count, …)
```

```
struct BinaryExecutor {
⋮
  template ‹class LEFT_TYPE, class RIGHT_TYPE, class RESULT_TYPE, …, class OP, …›
  static void ExecuteGeneric(
    Vector &left, Vector &right, Vector &result, idx_t count, …) {
      UnifiedVectorFormat ldata, rdata;

      left.ToUnifiedFormat(count, ldata);   ⇝ ldata ≡ [1,2,3,…,100]ᵘ (UnifiedVector)
      right.ToUnifiedFormat(count, rdata);  ⇝ rdata ≡ [42,42,…,42]ᵘ  (UnifiedVector)

      result.SetVectorType(VectorType::FLAT_VECTOR);
      auto result_data = FlatVector::GetData‹RESULT_TYPE›(result);
      ExecuteGenericLoop‹LEFT_TYPE, RIGHT_TYPE, RESULT_TYPE, …, OP, …› (
        UnifiedVectorFormat::GetData‹LEFT_TYPE›(ldata),
        UnifiedVectorFormat::GetData‹RIGHT_TYPE›(rdata),
        result_data, …, count, …);
  }
⋮
}
```

DuckDB 1.4 source: src/include/duckdb/common/vector_operations/binary_executor.hpp

Deep Dive Into DuckDB C++ Code: Vector Operations ❺

Both argument vectors [...]ᵘ are now in unified representation:

```
BinaryExecutor::ExeuteGenericLoop<int64_t, int64_t, bool, ..., duckdb::Equals, ...> (
  [1,2,3,...,100]ᵘ, [42,42,...,42]ᵘ, result, ..., count, ...)
```

```
struct BinaryExecutor {
⋮
  template <class LEFT_TYPE, class RIGHT_TYPE, class RESULT_TYPE, ..., class OP, ...>
  static void ExecuteGenericLoop(
    const LEFT_TYPE *__restrict ldata, const RIGHT_TYPE *__restrict rdata,
    RESULT_TYPE *__restrict result_data, ..., idx_t count, ...) {
      ⋮
      for (idx_t i = 0; i < count; i++) {
        auto lentry = ldata[i];
        auto rentry = rdata[i];
        result_data[i] = Operation<..., OP, LEFT_TYPE, RIGHT_TYPE, RESULT_TYPE> (
            ..., lentry, rentry, ...);
      }
  }
⋮
}
```

DuckDB 1.4 source: src/include/duckdb/common/vector_operations/binary_executor.hpp

Deep Dive Into DuckDB C++ Code: Vector Operations ❻

```
BinaryExecutor::ExeuteGenericLoop<int64_t, int64_t, bool, …, duckdb::Equals, …> (
  [1,2,3,…,100]ᵘ, [42,42,…,42]ᵘ, result, …, count, …)
```

```
struct BinaryExecutor {
⋮
  template <class LEFT_TYPE, class RIGHT_TYPE, class RESULT_TYPE, …, class OP, …>
  static void ExecuteGenericLoop(
    const LEFT_TYPE *__restrict ldata, const RIGHT_TYPE *__restrict rdata,
    RESULT_TYPE *__restrict result_data, …, idx_t count, …) {
      ⋮
      for (idx_t i = 0; i < count; i++) {
        auto lentry = ldata[i];
        auto rentry = rdata[i];
        result_data[i] = lentry == rentry;
      }
  }
⋮
}
```

DuckDB 1.4 source: src/include/duckdb/common/vector_operations/binary_executor.hpp

- ldata/rdata: plain C++ arrays of LEFT_TYPE/RIGHT_TYPE elements.
- const … *__restrict: *Hey C++, we never write to these arrays and they do not overlap in memory. Go on, optimize!*

Tight loops over vectors form the core of DuckDB's query engine.

- Routine to subtract two flat integer vectors (⚠ simplified):

```
#define STANDARD_VECTOR_SIZE 2048

void PROJECT_sub_int_col_int_col(
  int* col1, int* col2, int *res, int* sel)
{
    int i;

    if (sel) {
        // skip discussion of selection vector for now
    }

    for (i = 0; i < STANDARD_VECTOR_SIZE; i += 1) {
        res[i] = col1[i] - col2[i];
    }
}
```

#023

Assembly Code for Tight Loops

Use clang (options -O2 -fno-vectorize -fno-unroll-loops) to generate code for x86-64 CPUs.[4]

- Register assignment:
  col1: %rdi, col2: %rsi, res: %rdx, i: %rax

```
PROJECT_sub_int_col_int_col:
  xorl %eax, %eax              # %rax ←32 0
loop:
│ movl (%rdi,%rax,4), %ecx     # %ecx ←32 mem[%rdi + %rax × 4]
│ subl (%rsi,%rax,4), %ecx     # %ecx ←32 %ecx −32 mem[%rsi + %rax × 4]
│ movl %ecx, (%rdx,%rax,4)     # mem[%rdx + %rax × 4] ←32 %ecx
│ incq %rax
│ cmpq $2048, %rax             # 2048 loop iterations
└ jne  loop                    # exit if %rax = 2048
  retq
```

- **NB.** One loop exit test per vector element computed.

[4] Use Matt Godbolt's Compiler Explorer to inspect assembly code generated by contemporary compilers.

Explicit Loop Unrolling

- Manually perform **loop unrolling** to
  1. improve the ratio $^{(\textit{useful work})}/_{(\textit{loop exit test})}$,
  2. expose independent work that may be executed in ∥:

```
void PROJECT_sub_int_col_int_col(int* col1, int* col2, int* res)
{
  int i;

  for (i = 0; i + 3 < STANDARD_VECTOR_SIZE; i += 4) {
      res[i  ] = col1[i  ] - col2[i  ];
      res[i+1] = col1[i+1] - col2[i+1];    independent, execute in
      res[i+2] = col1[i+2] - col2[i+2];    any order or even in ∥
      res[i+3] = col1[i+3] - col2[i+3];
  }
}
```

#026

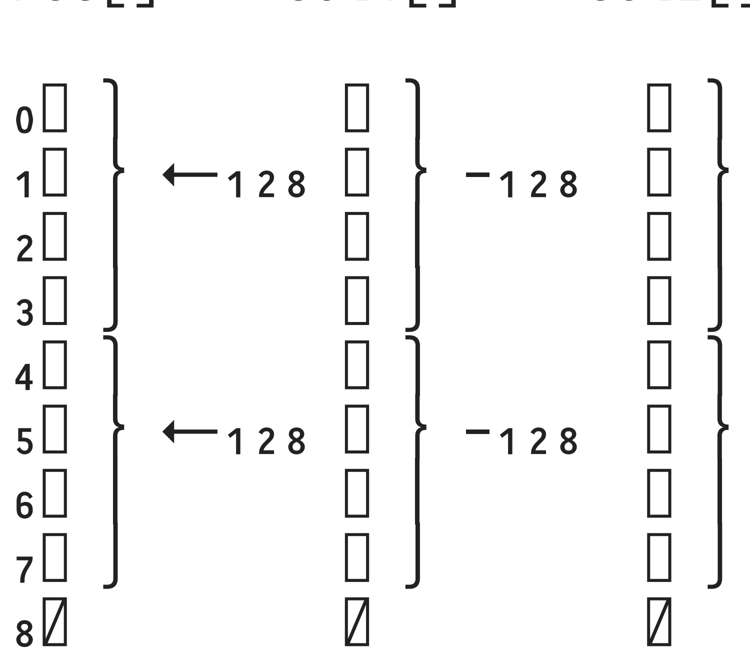
- **NB.** This code does not handle the case of STANDARD_VECTOR_SIZE mod 4 ≠ 0.

Loop Unrolling by the Compiler

Compiler clang (options -O2 -fno-vectorize -funroll-loops) unrolls four loop iterations (easy for CPU to ∥ize):

```
PROJECT_sub_int_col_int_col:
  xorl %eax, %eax               # i ←32 0
loop:
  movl (%rdi,%rax,4), %ecx      # %ecx ←32 col1[i]                }  no control or
  subl (%rsi,%rax,4), %ecx      # %ecx ←32 %ecx -32 col2[i]       }  data dependencies
  movl %ecx, (%rdx,%rax,4)      # res[i] ←32 %ecx                 }  between these
  movl 4(%rdi,%rax,4), %ecx     # %ecx ←32 col1[i+1]              }  code blocks
  subl 4(%rsi,%rax,4), %ecx     # %ecx ←32 %ecx -32 col2[i+1]     }
  movl %ecx, 4(%rdx,%rax,4)     # res[i+1] ←32 %ecx
  movl 8(%rdi,%rax,4), %ecx     # ⋮
  subl 8(%rsi,%rax,4), %ecx
  movl %ecx, 8(%rdx,%rax,4)
  movl 12(%rdi,%rax,4), %ecx
  subl 12(%rsi,%rax,4), %ecx
  movl %ecx, 12(%rdx,%rax,4)
  addq $4, %rax                 # } i ←32 i+4
  cmpq $2048, %rax              # } 2048 / 4 = 512 loop iterations
  jne  loop                     # exit if %rax = 2048
  retq
```

Data-Parallelism Through SIMD[5] Instructions



- Read/compute/write 4 vector elements (of width 4 × 32 bits = 128 bits) at a time in **data-parallel** fashion.
- Relies on CPU's SIMD support (e.g., Intel® SSE registers %xmm$_i$ and instruction **mov**e **d**ouble **q**uad word).

- **NB.** Requires care if
  - vectors res[] and col1[]/col2[] overlap in memory,
  - residual vector elements (see ▯) are to be processed.

[5] SIMD: **S**ingle-**I**nstruction, **M**ultiple **D**ata. Support is vendor-specific, rapidly evolving in contemporary CPUs. Intel's *AVX2* registers: 512 bits, ARM *NEON* registers: 128 bits. Compilers implement CPU architecture flags.

Data-Parallelism Through SIMD Instructions (Vector Overlap)

C compiler clang (options -O2 -fvectorize) uses SIMD registers and instruction set.

- Extra prelude code checks for **vector overlap** on function entry. If so, jumps to non-vectorized (yet unrolled) version of code.
- Declare function arguments with modifier __restrict__ to inform C compiler that vectors do not overlap (⚠ omits all checks):

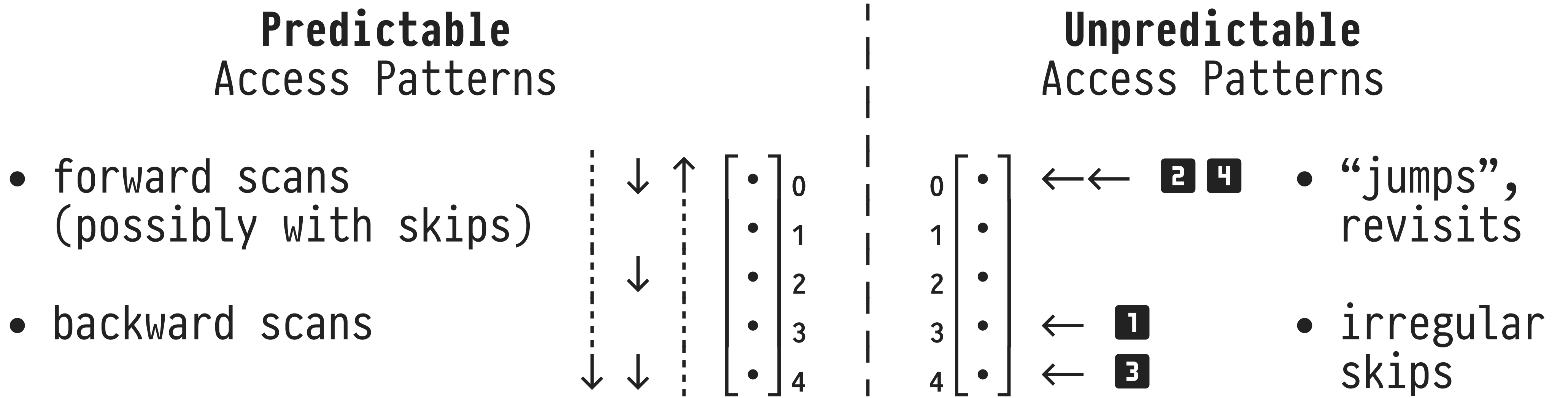
```
void PROJECT_sub_int_col_int_col(
  int *__restrict__ col1, int *__restrict__ col2, int *__restrict__ res)
{
  int i;
  ⋮
}
```

Data-Parallelism Through SIMD Instructions (Core Loop)

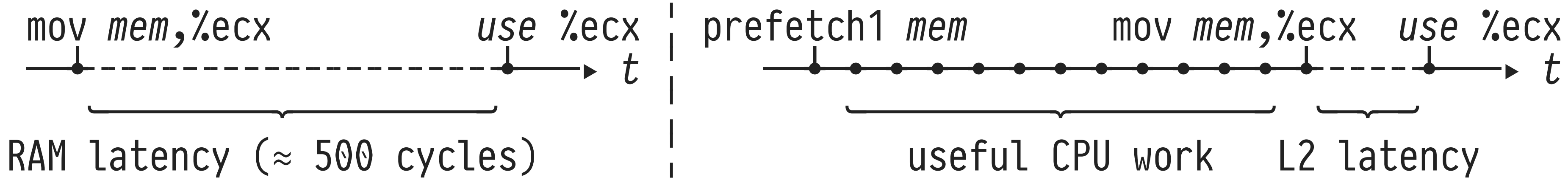
Process 16 vector elements per iteration (SIMD + 2 loops unrolled; assumes no overlap of vectors res[] and col1[]/col2[]):

```
PROJECT_sub_int_col_int_col:
  ⋮
  overlap check omitted
  ⋮
  xorl %eax, %eax                          4 × 32 bits = 128 bits wide
loop:
  movdqu (%rdi,%rax,4), %xmm0       # %xmm0 ←128 col1[i+0...i+3]
  movdqu 16(%rdi,%rax,4), %xmm1     # %xmm1 ←128 col1[i+4...i+7]
  movdqu (%rsi,%rax,4), %xmm2       # %xmm2 ←128 col2[i+0...i+3]
  psubd %xmm2, %xmm0                # %xmm0 ←128 %xmm0 -128 %xmm2
  movdqu 16(%rsi,%rax,4), %xmm2     # %xmm2 ←128 col2[i+4...i+7]
  psubd %xmm2, %xmm1                # %xmm1 ←128 %xmm1 -128 %xmm2
  movdqu %xmm0, (%rdx,%rax,4)       # res[i+0...i+3] ←128 %xmm0            loop #n
  movdqu %xmm1, 16(%rdx,%rax,4)     # res[i+4...i+7] ←128 %xmm1  ..........................
  movdqu 32(%rdi,%rax,4), %xmm0     # %xmm0 ←128 col1[i+8 ...i+11]
  movdqu 48(%rdi,%rax,4), %xmm1     # %xmm1 ←128 col1[i+12...i+15]         loop #n+1
  movdqu 32(%rsi,%rax,4), %xmm2     # %xmm2 ←128 col2[i+8...i+11]
  psubd %xmm2, %xmm0                # %xmm0 ←128 %xmm0 -128 %xmm2
  movdqu 48(%rsi,%rax,4), %xmm2     # %xmm2 ←128 col2[i+12...i+15]
  psubd %xmm2, %xmm1                # %xmm1 ←128 %xmm1 -128 %xmm2
  movdqu %xmm0, 32(%rdx,%rax,4)     # res[i+8 ...i+11] ←128 %xmm0
  movdqu %xmm1, 48(%rdx,%rax,4)     # res[i+12...i+15] ←128 %xmm1
  addq $16, %rax                    # }
  cmpq $2048, %rax                  # } 2048 / 16 = 128 iterations
  jne  loop                         # exit if %rax = 2048
  ⋮                                 # (non-vectorized code not shown)
```

6 ¦ Predictable Memory Access and Prefetching



- **Predictable:** CPU automatically issues **asynchronous memory prefetch** operations to preload caches and hide memory latency.
- **Unpredictable:** DBMS code adds explicit **software prefetch**[6] **instructions** for memory addresses needed in the future:



[6] No-ops with side effect on the data cache. Example: prefetcht1 loads into the L2 cache on Intel® Core i7.

7 ┆ Implementing Selection in Tight Loops

The core of the FILTER operator is a **conditional statement** that is evaluated for each input vector element (⚠ simplified again):

#023

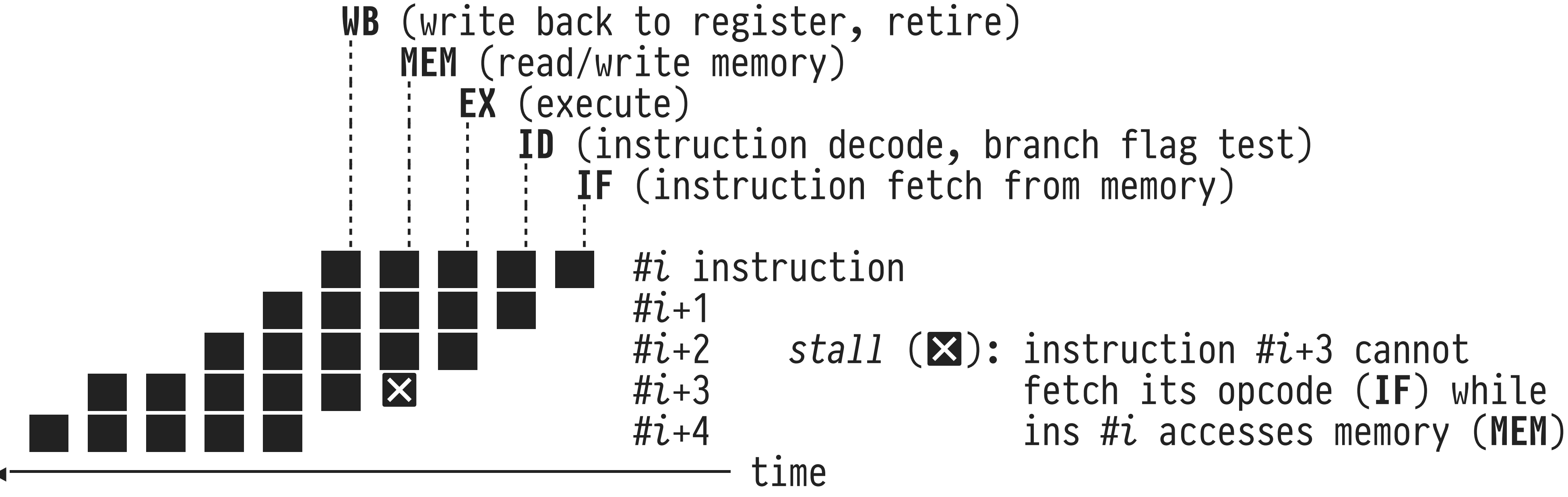
```
int FILTER_lt_date_col_date_val(int* res, int* col, int val)
{
  int i, o = 0;

  for (i = 0; i < STANDARD_VECTOR_SIZE; i += 1) {
     if (col[i] < val) {                         // test filter condition (col < val)
        res[o] = i;                              // build selection vector
        o += 1;
     }
  }

  return o;                                      // output cardinality
}
```

- Conditionals (and loops) lead to **branches** in generated machine code that disrupt the CPU's processing of straight-line instruction sequences. 👎

Instruction Pipelining in Modern CPUs

Contemporary CPUs process instructions in a pipeline of execution stages that compete for CPU resources (*e.g.*, the arithmetic logic unit or the memory bus).

- A simple five-stage pipeline:



Branch Taken? Yes, Flush Pipeline

Control flow branches (for, but particularly if) are a challenge for modern pipelining CPUs.

The simple pipeline decides the outcome of branch #*i* (at end of **ID**) only *after* instruction #*i*+1 has already been fetched (**IF**).

- If the branch is taken, **flush** instruction #*i*+1 from the pipeline 👎, instead fetch instruction #*j* at jump target:

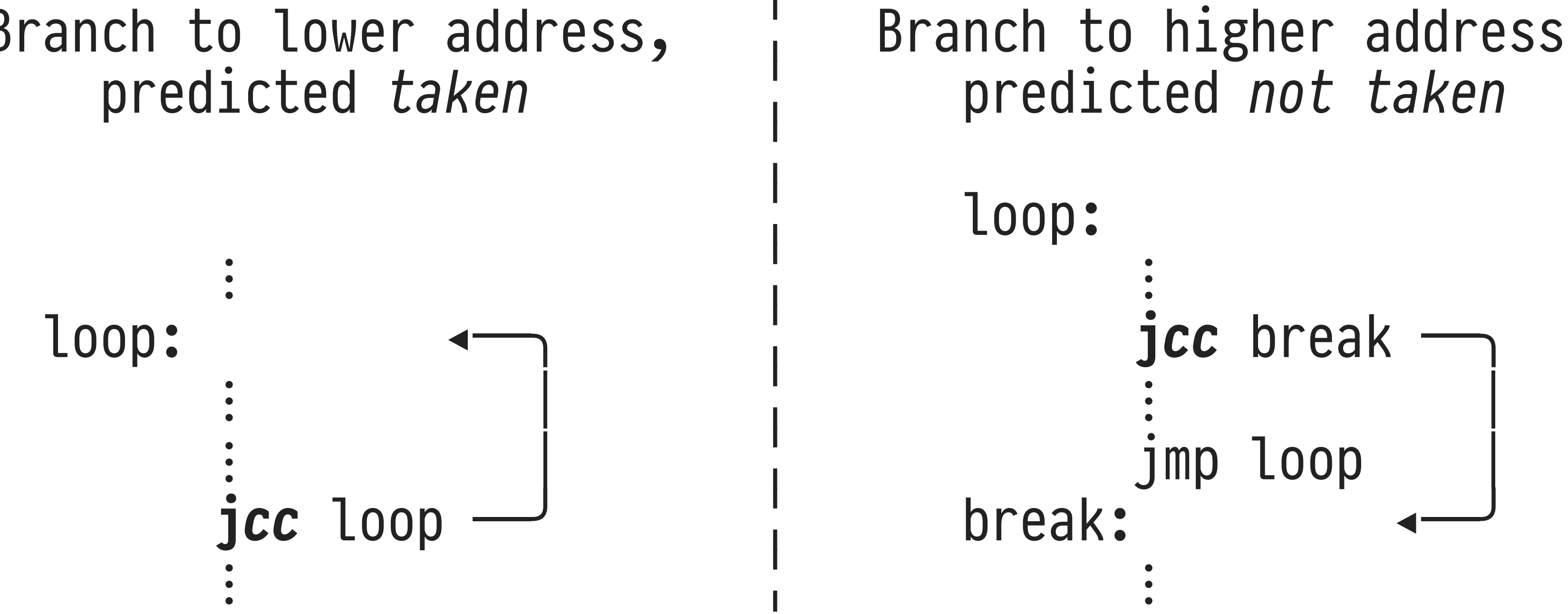
ID (instruction decode, branch flag test)
IF (instruction fetch from memory)
#*i* instruction (conditional branch, *e.g.* **jne**)
#*i*+1
#*j*
time
flush (☒): remove instruction #*i*+1 from pipeline

Branch Predictions: History and Heuristics

CPUs thus try to **predict the outcome of a branch** *#i* based on **earlier recorded outcomes** of the same branch:

Branch prediction	Fetch instruction
taken	*#j*
not taken	*#i*+1

- Also: **heuristics** based on typical control flow patterns (instruction **jcc**: conditonal branch on **c**ondition **c**ode):



Avoiding Branch Mispredictions

A **mispredicted branch** leads to

1. pipeline flushes—effectively a stall ❎—and
2. (possibly) CPU instruction cache misses.

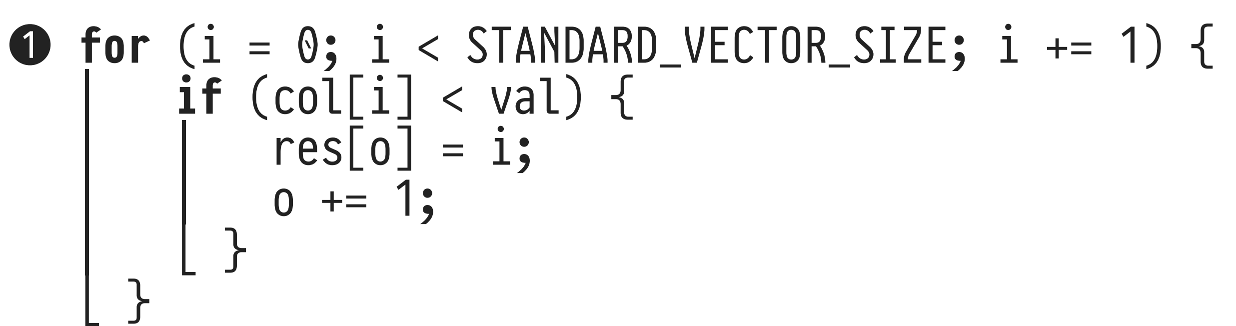
The resulting runtime penalty is significant (≈15 cycles).

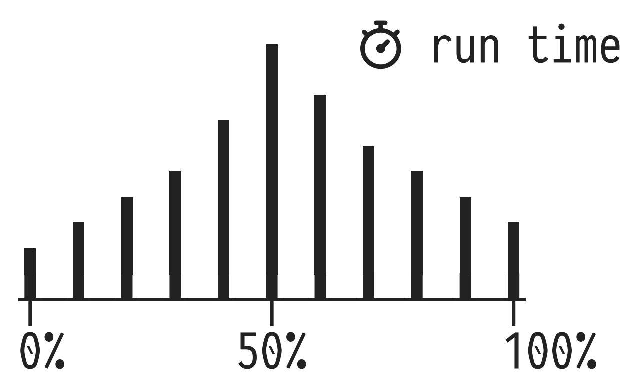
💡 DBMS code aims to avoid branch mispredictions in tight loops:

- Prefer **branch-less** implementations of operator logic, or at least
- reduce the number of random/hard-to-predict branches.

Branch-Less Filtering

Measure wall-clock run time of core FILTER loop for different **selectivities** of predicate col < val:

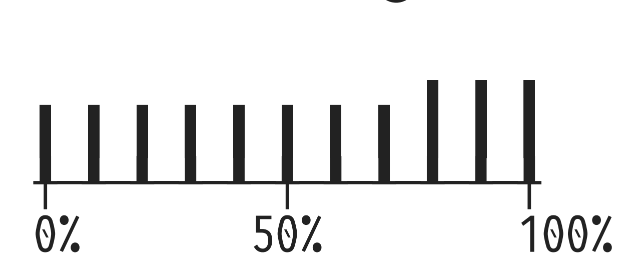




run time

```
❷ for (i = 0; i < STANDARD_VECTOR_SIZE; i += 1) {
      res[o] = i;
      o += (col[i] < v);
  }
```

≡ 1 if predicate satisfied, else 0



❷: Only well-predictable loop control flow (**for**) remains. 👍

Mixed-Mode Selection

There is an entire space of possibilities to implement composite predicates (e.g., the conjunction p_1 AND p_2):

- Use branch-less selection via `o += `p_1` & `p_2 (NB. uses of C's bit-wise *and* operator `&`).
- Identify the *more selective*[7] (and thus more predictable) conjunct p_1, say, then use

```
if (p₁) {
    res[o] = i;
    o += (p₂);
}
```

#030

(This approach particularly fits DBMSs that generate code from SQL queries—which DuckDB does *not* do.)

[7] ⚠ **This is important.** If p_2 is unpredictable then the mixed-mode selection `if (`p_2`) { … o += (`p_1`) }` will suffer from branch misprediction.

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*.

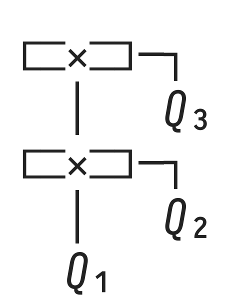
To this end, DuckDB implements a SQL processing regime that the DB community has engineered and refined since the late 1970s:

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** $\rightarrow^{+}$, 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.

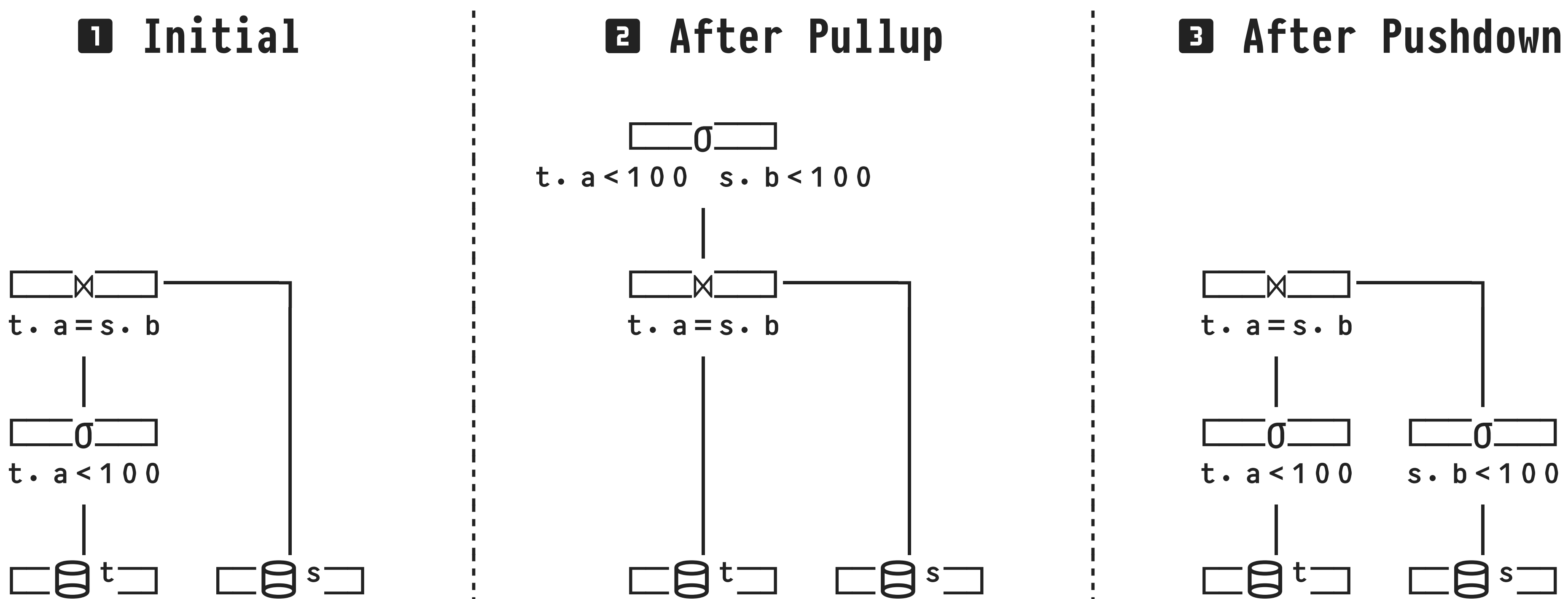
	Propagated statistics (excerpt)
intermediate tables	• estimated cardinality • maximum cardinality (helps allocation)
numeric columns	• min/max value (may be unknown or uninteresting if entire domain of column type is covered) • is constant? • contains NULL?
text columns	• min/max value (first 8 characters)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

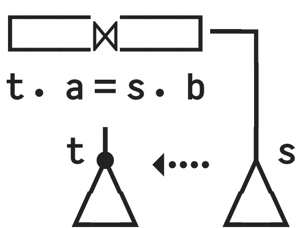
- DuckDB additionally implements **filter pullup** (propagate restrictions through equality predicates):[3]



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

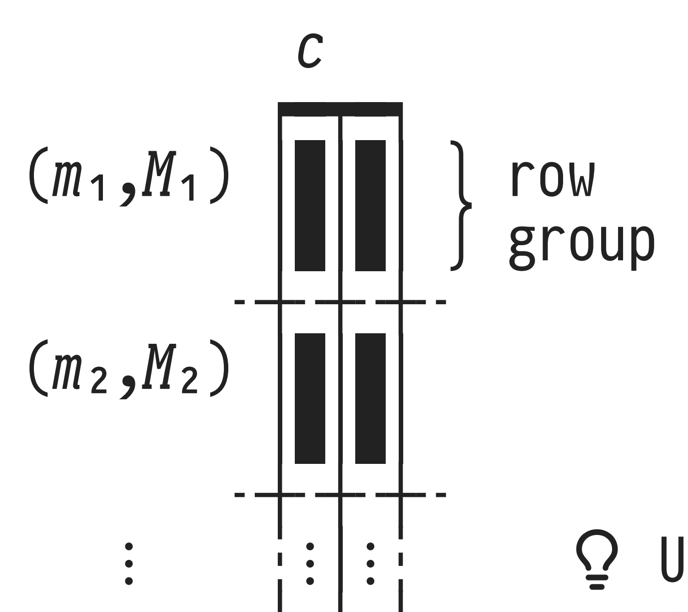
Join execution proceeds in two (alternating) phases:

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 $\{x_1,\dots,x_n\}$ (with $n \leqslant$ dynamic_or_filter_threshold, default 50), the dynamic predicate will be IN $(x_1,\dots,x_n)$ 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
```

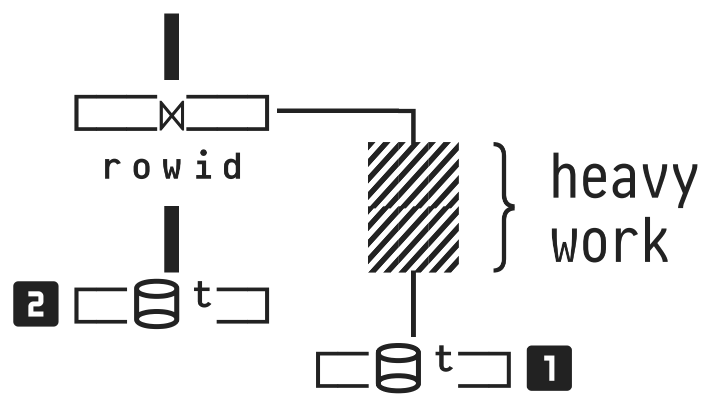
Scan row groups in descending M_i (max) order. **Stop criterion:**

$(m_i,M_i) \cap (m_j,M_j) = \emptyset$? (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.
- ⋉ is a left semi-join on row IDs.

rows: | narrow, ▌ wide

#038

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

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

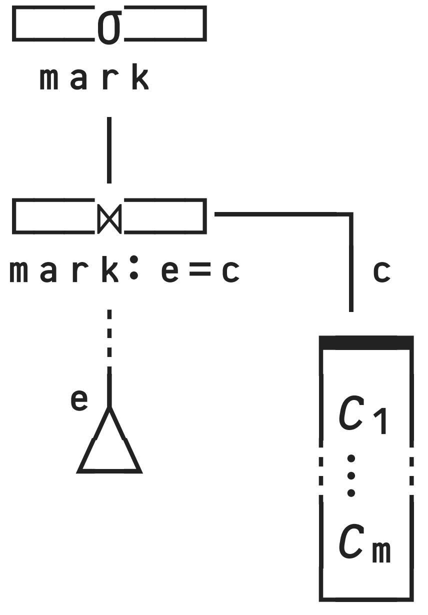
- **Notes:**
  - In IEEE754 floating point arithmetic, + is not associative. DuckDB applies pass sum_rewriter only if *e* and *const* are integer-typed.
  - Both sides are equivalent also if *e* may be NULL. (In SQL, aggregates sum and count ignore NULL values.)
  - Expression *e* is only evaluated once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

#039

Naively evaluating predicates like e IN (c_1,…,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.

#040



COLUMN_DATA_SCAN

- 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: #042

```
               #refs                     #refs: # of references to CTE
           1/  0|  \1+                        ×: CTE removed from plan
 MATERIALIZED?  ×   MATERIALIZED?       MAT...?: CTE def annotated by
  n/  ∅|   \y         n/  ∅|    \y              [NOT] MATERIALIZED?
inl   inl   mat     inl    |     mat
                       volatile?     volatile?: does CTE use functions
                        n/   \y                 like random()?
                     agg?    mat            agg?: will the CTE result
                    n/  \y                        likely be small?
                empty?   mat              empty?: will the CTE result be
               n/   \y                            empty?
              mat   inl                  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

#044

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

```
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
```

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:

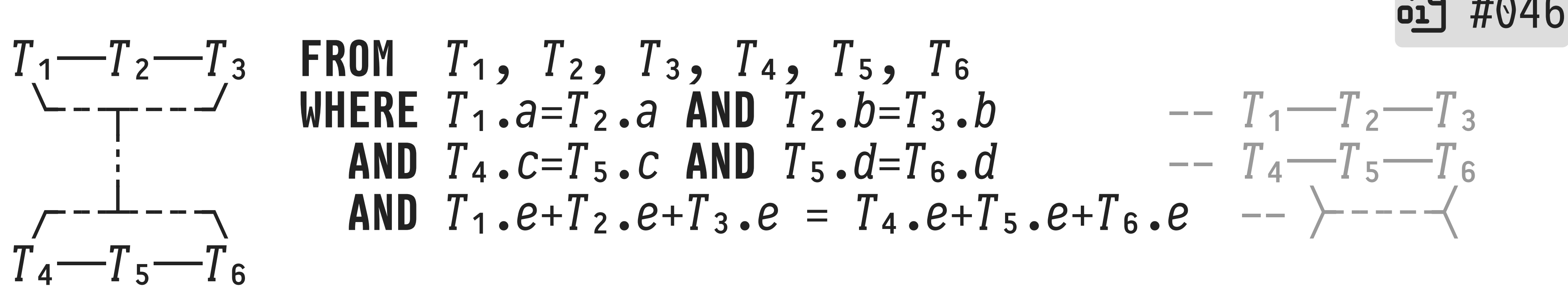
$$C_n = \sum_{k=0}^{n-1} C_k \times C_{n-k-1}$$

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.

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

lhs

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

probe
outer

build
inner

rhs

- 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:

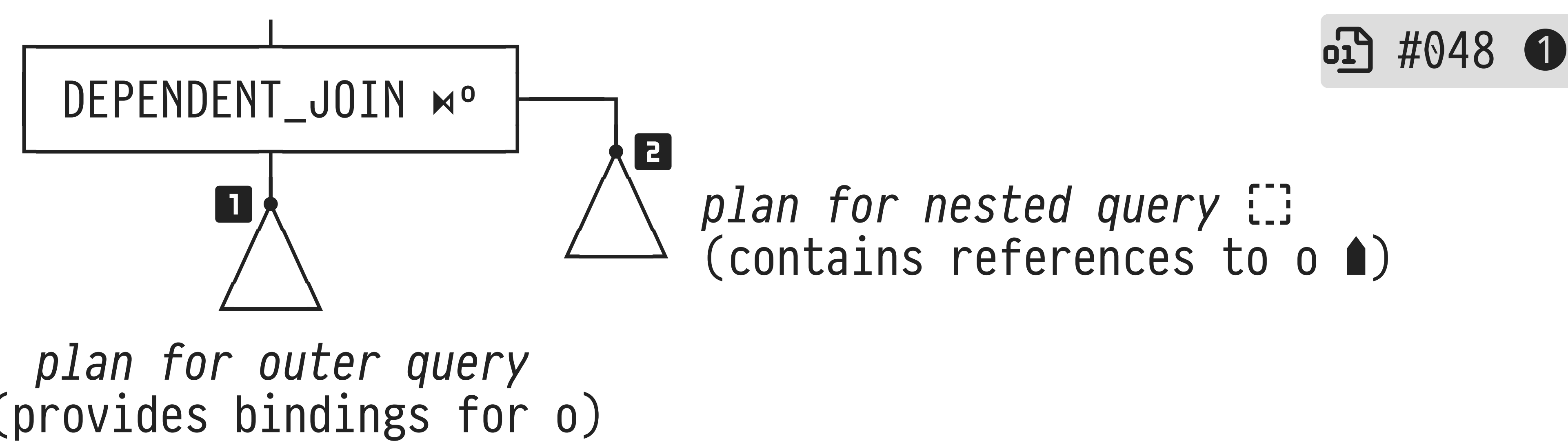
```
SELECT o.o_orderkey, o.o_orderdate, l.l_linenumber
FROM   orders AS o, lineitem AS l
WHERE  o.o_orderkey = l.l_orderkey
AND    l.l_extendedprice = (SELECT min(i.l_extendedprice)
                            FROM   lineitem AS i  ⛊
                            WHERE  i.l_orderkey = o.o_orderkey);
```

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: ⋈°):



- 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.

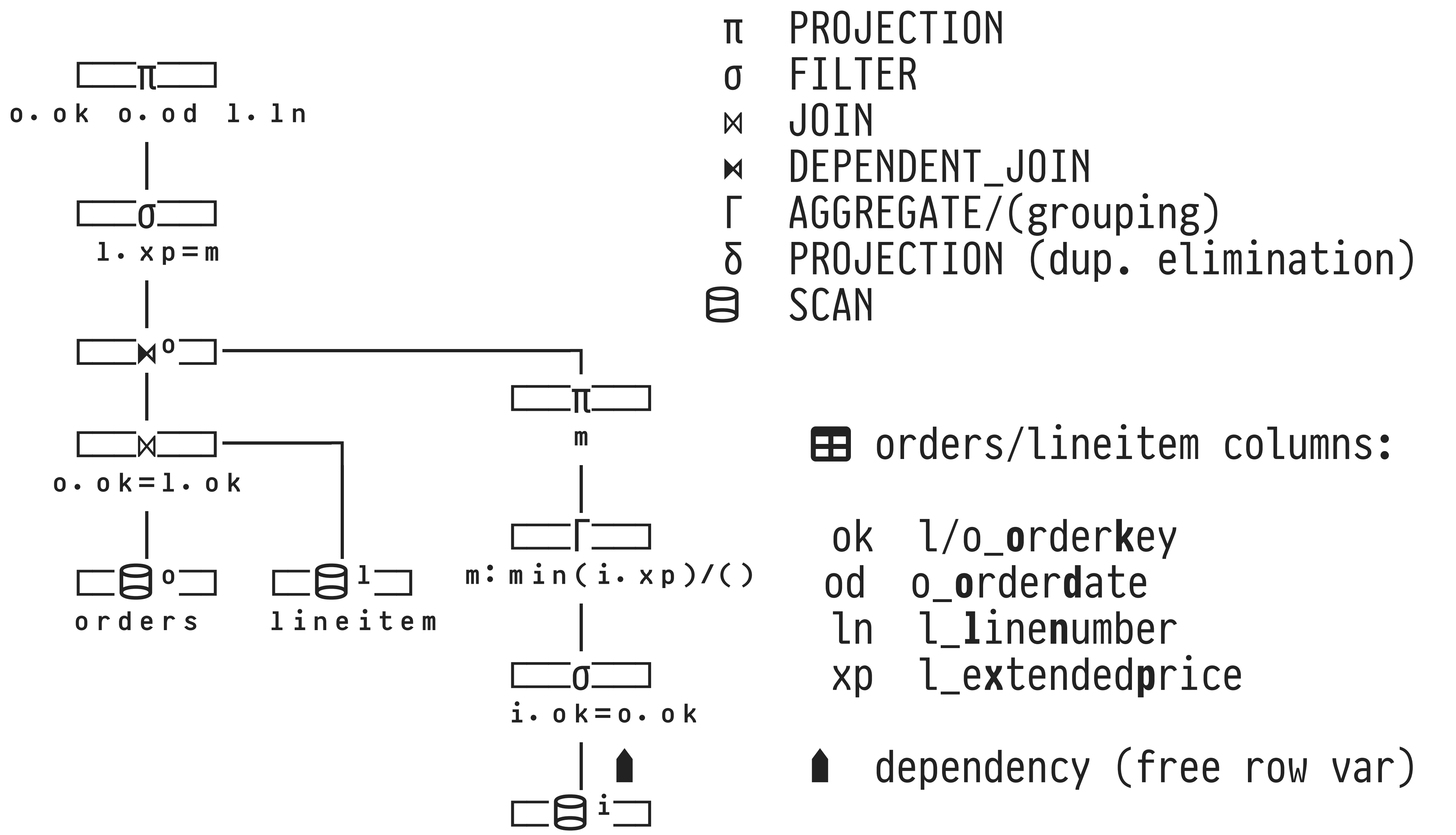
Query decorrelation:

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.

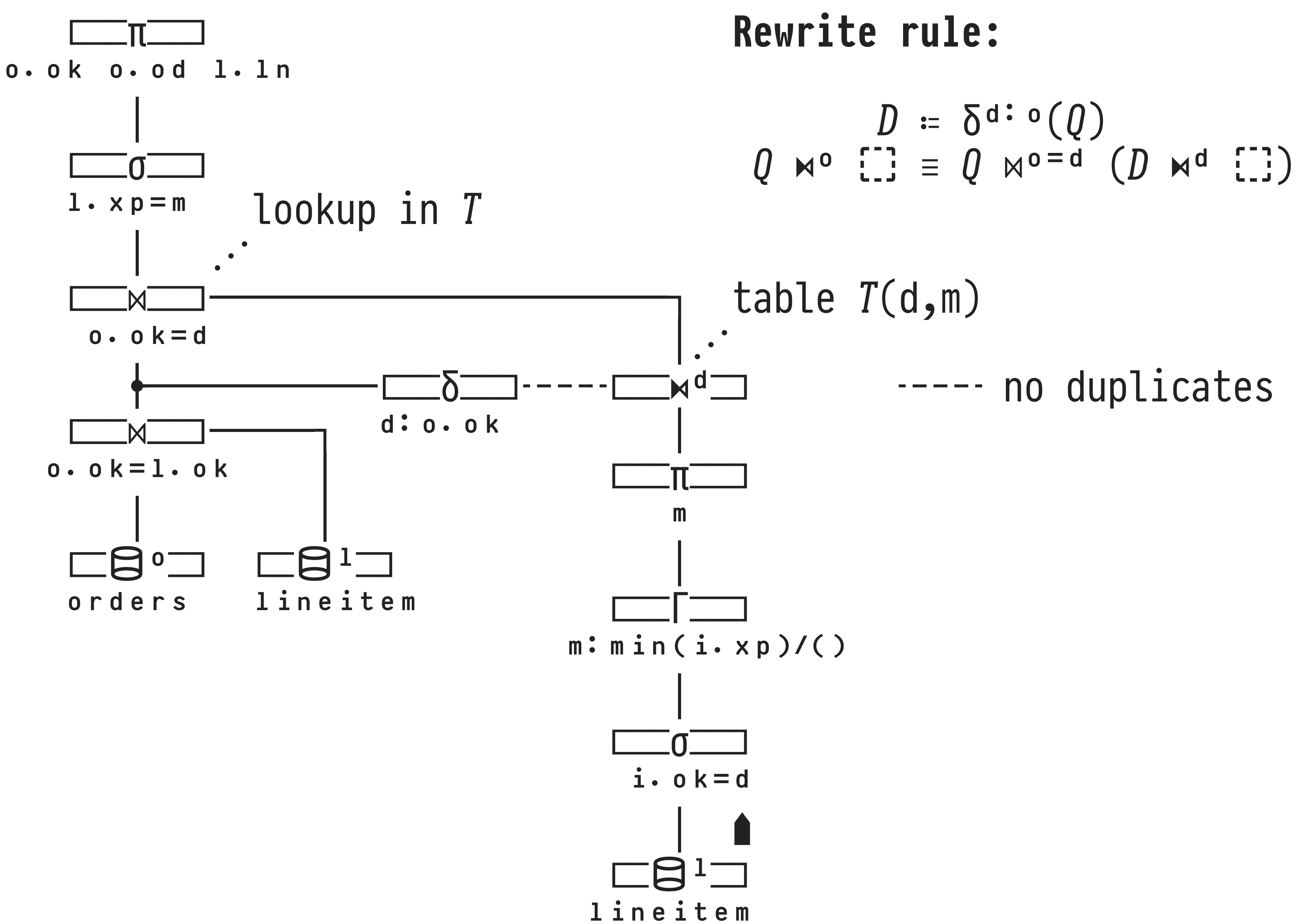
Thomas Neumann and Alfons Kemper of TU Munich[9] implement this idea via systematic rewrites that push DEPENDENT_JOIN ⧓ down into a query plan until it can be trivially replaced by regular JOIN ⋈.

[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)

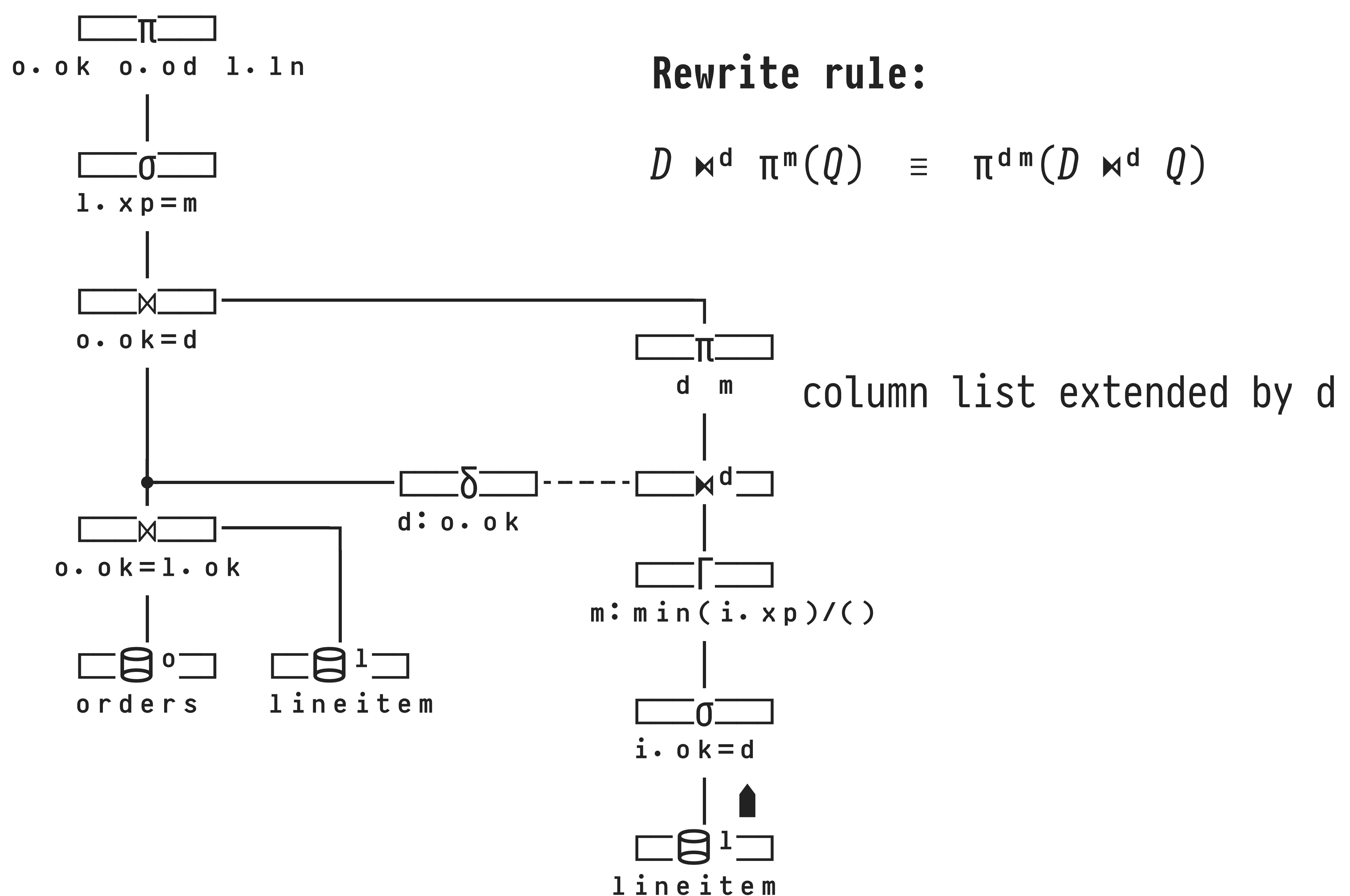


Rewrite rule:

$$D \coloneqq \delta^{d:o}(Q)$$

$$Q \ltimes^{o} \square \equiv Q \bowtie^{o=d} (D \ltimes^{d} \square)$$

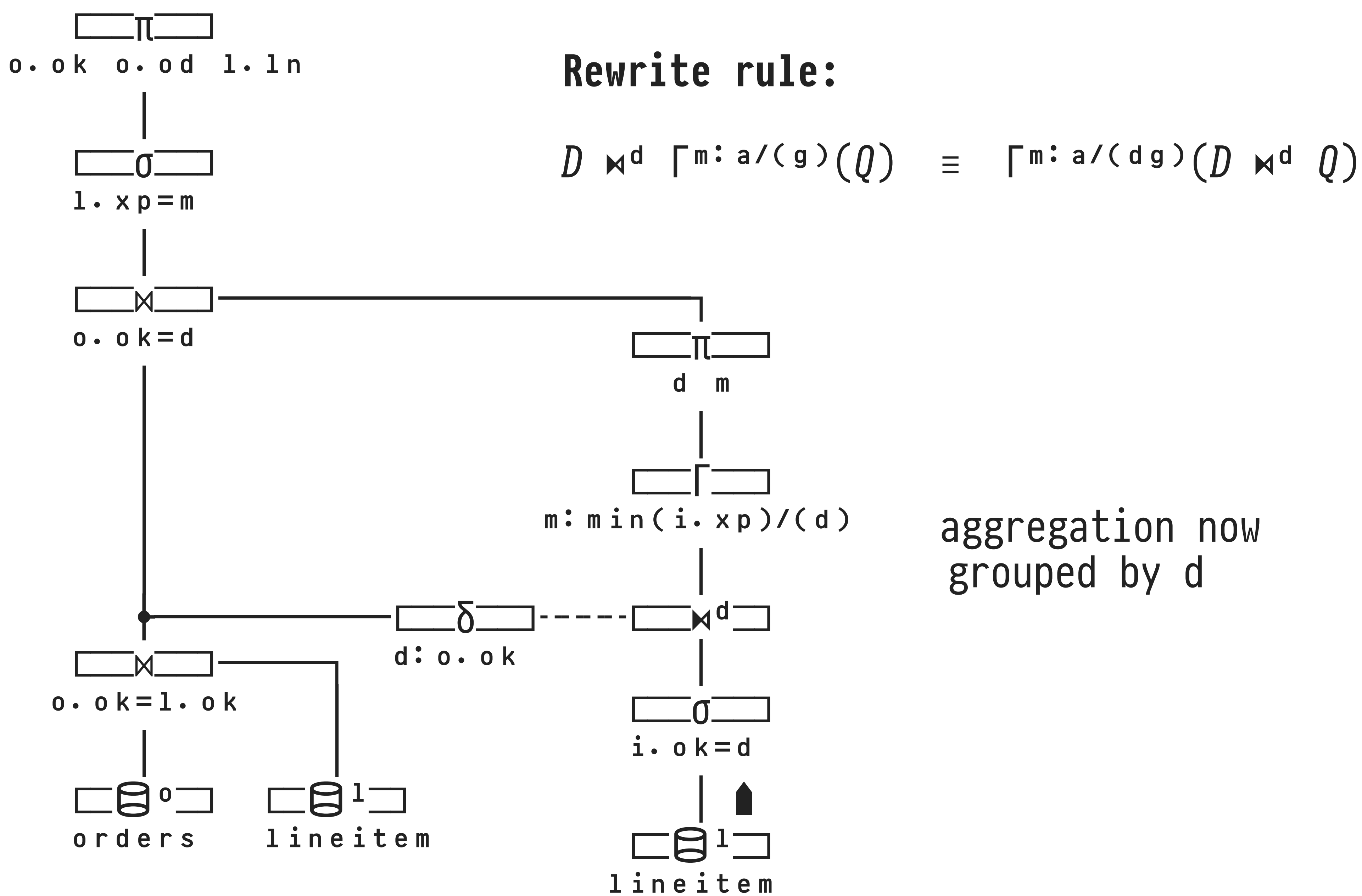
Query Decorrelation 2 (Push ⋉ Down Through π)



Rewrite rule:

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

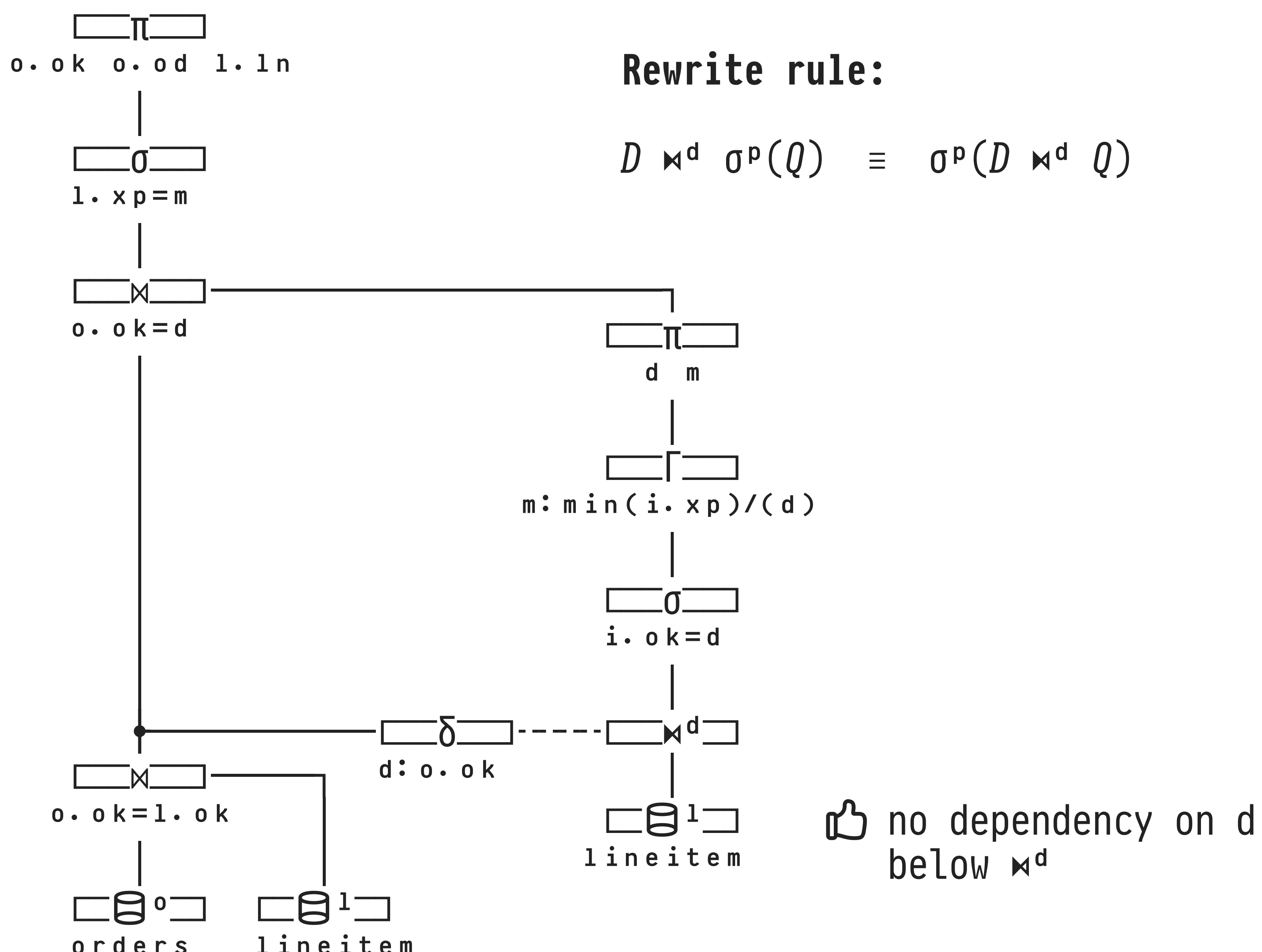
Query Decorrelation 3 (Push ⋉ Down Through Γ)



Rewrite rule:

$$D \ltimes^{d} \Gamma^{m:a/(g)}(Q) \equiv \Gamma^{m:a/(dg)}(D \ltimes^{d} Q)$$

Query Decorrelation 4 (Push ⋈ Down Through σ)

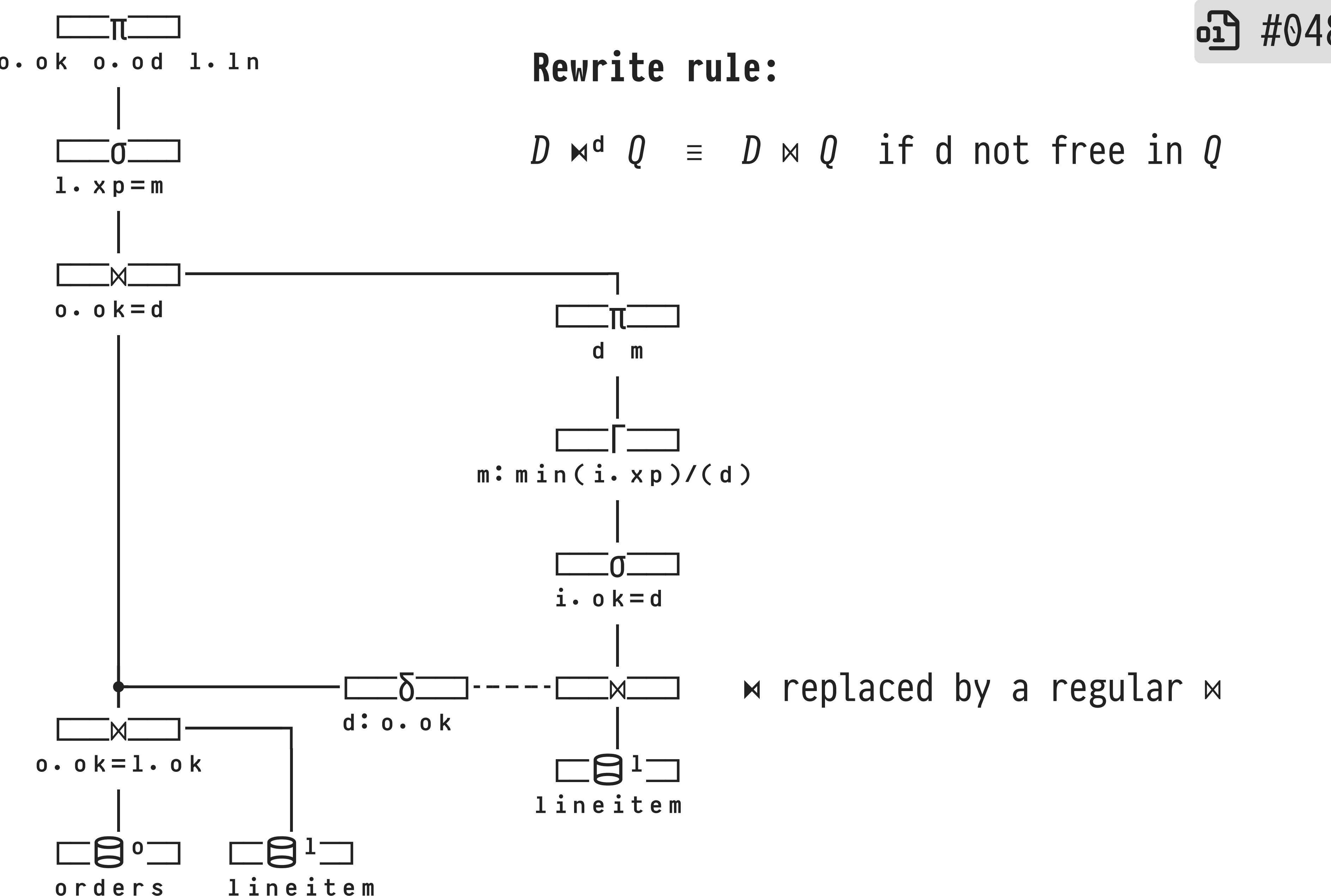


Rewrite rule:

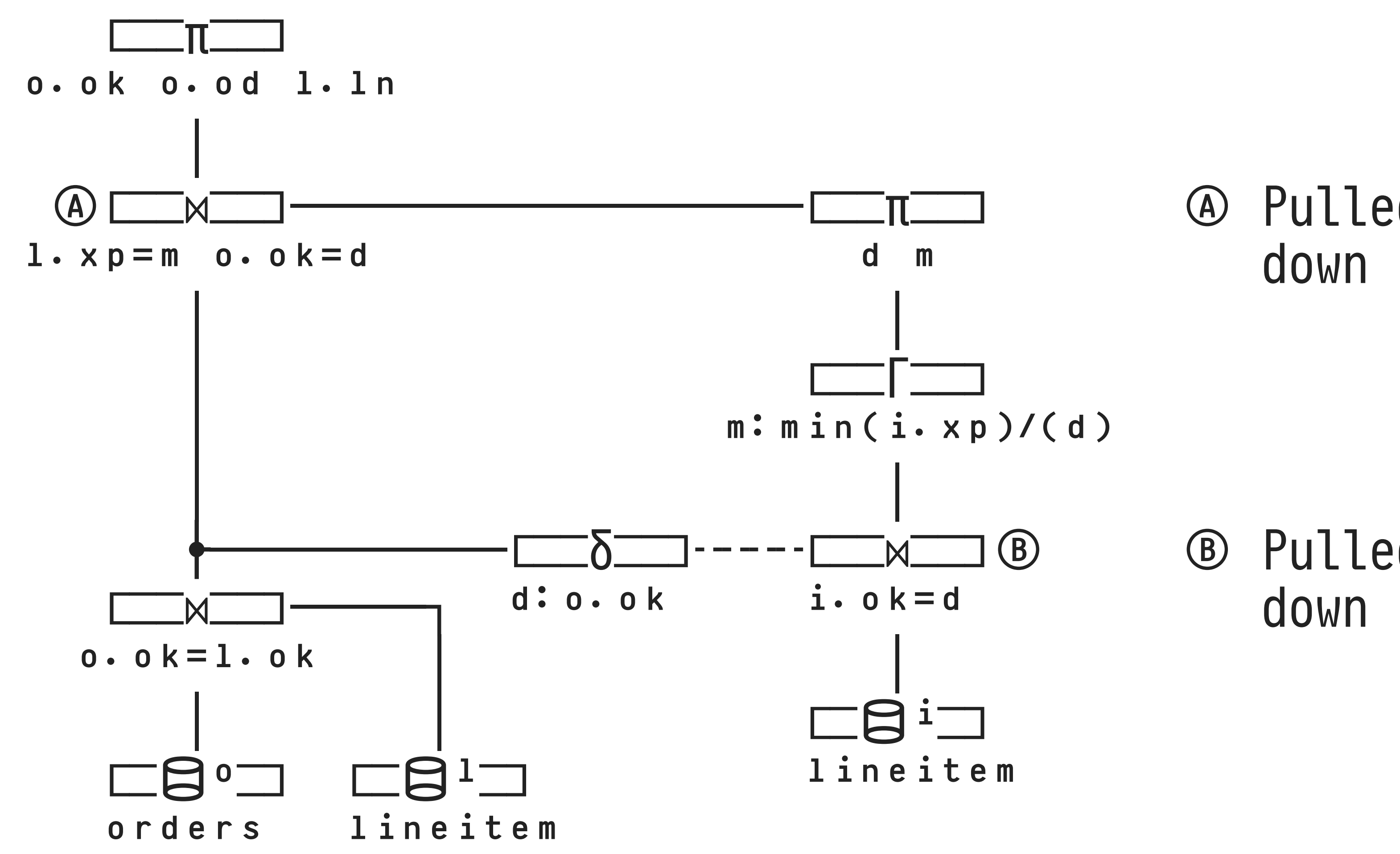
$$D \ltimes^d \sigma^p(Q) \equiv \sigma^p(D \ltimes^d Q)$$

no dependency on d below $\ltimes^d$

Query Decorrelation 5 (Eliminate ⋈)



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

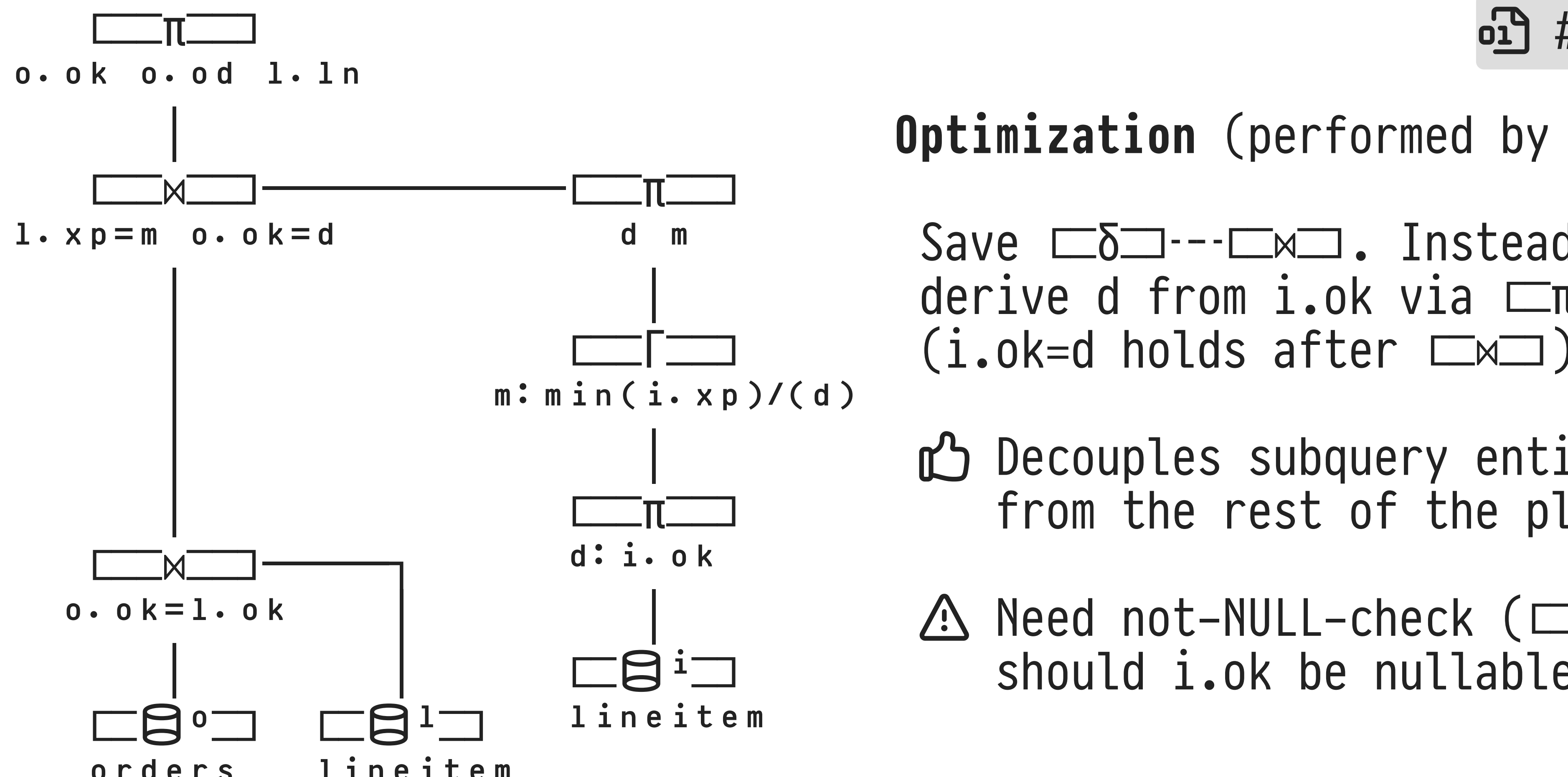


Ⓐ Pulled $\sigma^{l.xp=m}$ down into ⋈.

Ⓑ 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