Tabular Database Systems

①

Tabular Data and Database Systems

March 17, 2026

Torsten Grust
Universität Tübingen, Germany

1 Welcome!

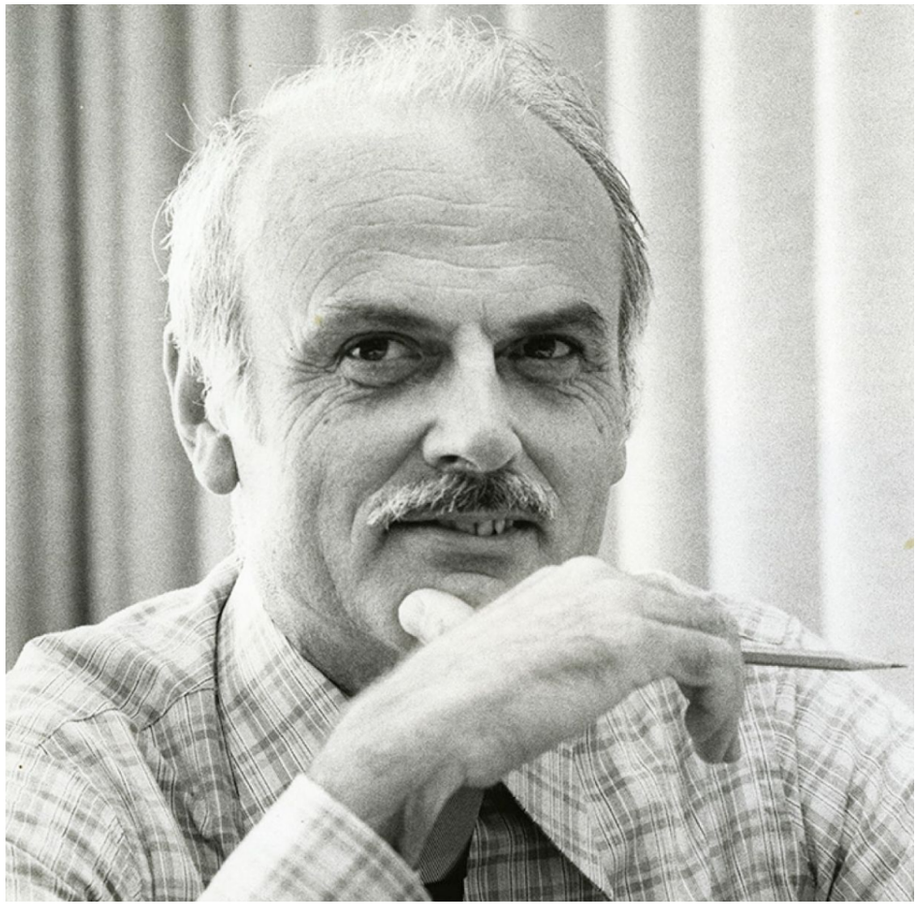
Welcome to the rectangular world of **tabular database systems.**

Reshaping all kinds of data to fit into rows + columns may appear restrictive and arcane (it certainly did to me).

This course will investigate how **tables**...

- ... are a **versatile and flexible data representation** (also for non-tabular data),
- lead to **compact data storage** (in main memory as well as secondary memory), and
- admit **super-efficient data processing.**

Tables are Versatile

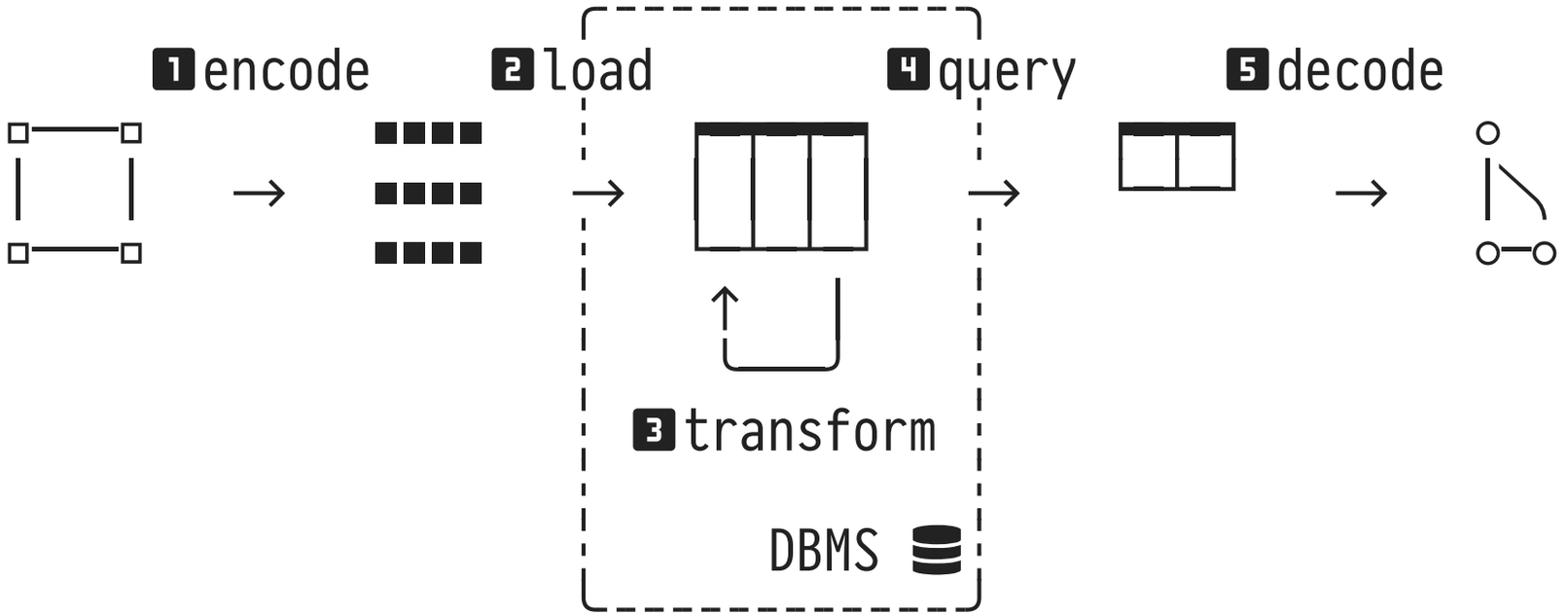


Edgar F. Codd (© IBM)

“Twelve distinct ways to represent data at the logical level are eleven too many.”

— Edgar F. Codd, ACM Turing Laureate

Processing Data Using Tabular Database Systems



- Step 1 reshapes data structures to fit rows + columns ▪▪▪▪.
- Data format ▪▪▪▪ still is external to the **database management system** (DBMS ⬚🛢). Use the DBMS's *load/copy* facilities 2 to turn this external format into in-database tables ◫.
- At 3 and 4 use **SQL** as a language to process tabular data.
- Users or apps may expect the original data structure again 5.

Tabular Database Systems: The Payoff is Substantial

Q: “But are 1️⃣ (encode), 2️⃣ (load), and 5️⃣ (decode) worth it?”
A: “*You bet!*”

- The DBMS-internal data format is **compact** (compression).
- **SQL is a declarative language** whose primary data type are tables. SQL queries 3️⃣+4️⃣ tend to be concise, often elegant.
- DBMS internals rely on regular table structures: **SQL is very efficient,** typically *way* faster than handcrafted programs.
- DBMS **coordinates concurrent access** to its tables—many users may operate on the same tables using well-defined semantics.
- DBMS **safely persists data** under its control, preventing data loss, e.g., through bugs in apps or system outages.

2 ¦ This Course (Tabular Database Systems, short: TaDa or 🎉)

- We will focus on **table-centric data representation and processing.** There are other kinds of DBMSs (e.g., graph-based DBMS: ⚂→¦⚂↻ ⚃¦→⚃), but we will ignore those.
- Whenever we can approach the theory or **pragmatics** of a concept, we typically choose the latter. (Yet, tabular database systems are rooted in a rich and elegant mathematical foundation.)
- We will get our hands dirty using the tabular DBMS **DuckDB** 🦆 and its extensive **SQL** dialect.
- We will explore selected aspects of **DuckDB's internals** (implementation techniques used inside the ⬚).
- We will draw from a variety of data 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/TaDa

- 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-create-table.sql`).

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

Material

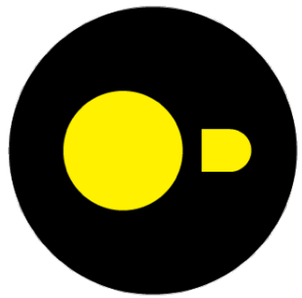
This course is *not* based on a single textbook. Rather, we build on

- 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 (quizzes, puzzles, 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:



DuckDB

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.[1]

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

The **tabular data model** arranges all information in a rigorous grid-like fashion. Consider table vehicles below:

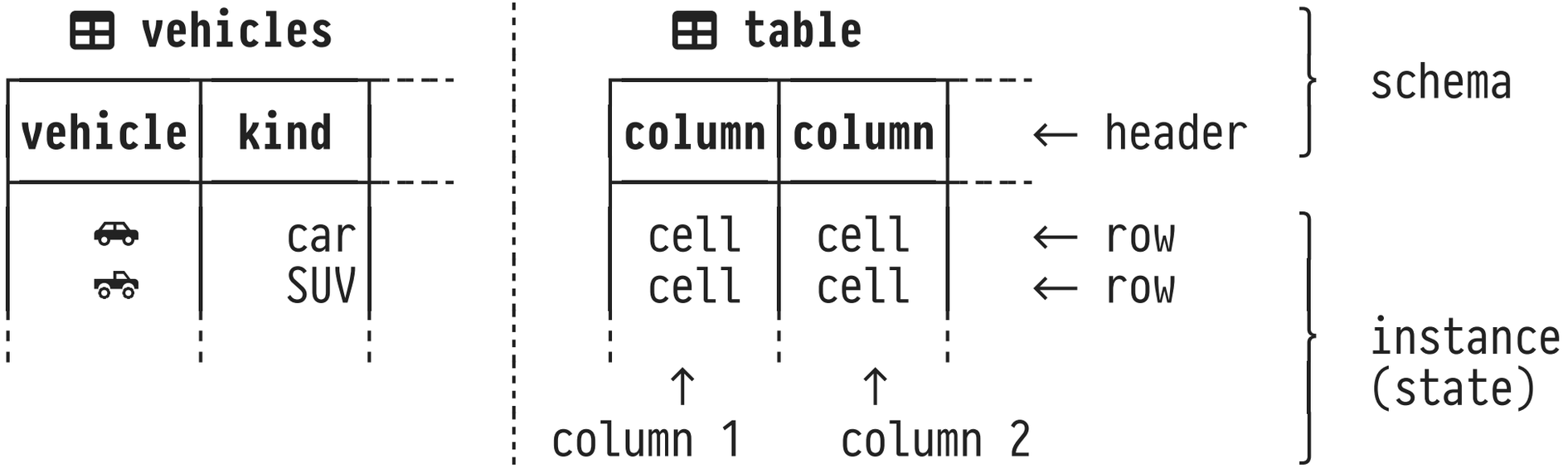
vehicles

vehicle	kind	seats	wheels?
🚗	car	5	true
🚙	SUV	3	true
🚌	bus	42	true
🚐	bus	7	true
🚲	bike	1	true
🚜	tank	▢	false
🏎	cabrio	2	true

- This **table** has four **columns**, seven **rows**.
- Table and columns are **named**, rows are not.
- Each column holds **cell** values of a single **type**.
- Cell value **NULL** (▢) signals absence of information.

- Tables are vertical: feature **few columns** ($\leqslant 20$ is typical), but may contain **large numbers of rows** (10–10^6 rows are typical).

Tabular Data Model: Terminology



- Each **column** $i \in \{1,2,...\}$ is assigned a **name** c_i and its **type** τ_i.
- **Types** τ_i are simple (text, int, boolean, float, int[], ...): **first normal form** or **1NF** (columns may *not* hold nested tables).
  - *All* cell values in column i are of type τ_i (NULL if allowed).
- The **schema** determines the structure of a table (say t):
  $t(c_1\ \tau_1,...,\ c_n\ \tau_n)$, in short: $t(c_1,...,c_n)$. Typically *fixed*.
  Ex: vehicles(vehicle text, kind text, seats int, "wheels?" boolean).
- The **instance** is the bag of rows held in the table. *Dynamic*.

SQL: Creating Tables

- The SQL statement CREATE TABLE $t(c_1\ \tau_1,\ldots,c_n\ \tau_n)$ creates
  - a table with given **schema** $t(c_1\ \tau_1,\ldots,\ c_n\ \tau_n)$ and
  - an **empty instance** (no rows yet).

- In the DuckDB CLI:

optional — table name

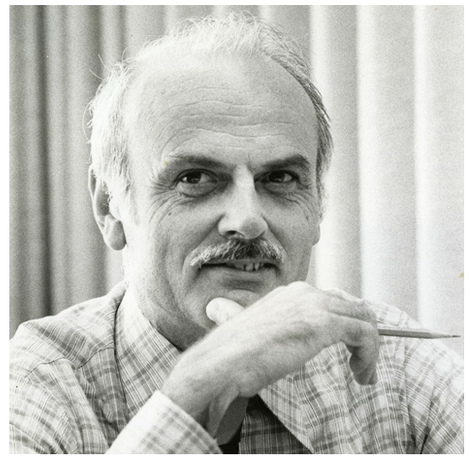
```
D CREATE OR REPLACE TABLE vehicles (
    vehicle   text     NOT NULL,
    kind      text     NOT NULL,
    seats     int                ,
    "wheels?" boolean  NOT NULL
  );
```

columns — types — constraints [optional]

refers to code/extra material in a file named 001- →* #001

Tables vs. Relations ❶

Origin and foundation of the tabular data model is the **relational model** (Edgar F. Codd, 1970).



Edgar F. Codd (© IBM)

Tables and **relations** (subsets of the Cartesian product of domains) are closely related.

- Relation $<$ (*less than*) on integers:
  $< \equiv \{ (x,y) \mid x \in \mathbb{Z}, y \in \mathbb{Z}, \exists n \in \mathbb{N}, n \neq 0 : x+n = y \} \subseteq \mathbb{Z} \times \mathbb{Z}$
  - Notation: $(x,y) \in <$ or $\underline{<}(\underline{x},\underline{y})$ or $x < y$, e.g., $<(0,42)$

- Can define relations **intensionally** (as above) or **extensionally.**
  Game *Rock*, *Paper*, *Scissors* (finite domain $\mathbb{RPS} = \{✊,✋,✌\}$, with extensional "*beats*" relation: $x < y \Leftrightarrow y$ *beats* x):

$<$ ≡

x	*y*
✌	✊
✋	✌
✊	✋

extent of $<$ (a subset of $\mathbb{RPS} \times \mathbb{RPS}$):
lists the set of all ordered pairs included in $<$

Tables vs. Relations ❷

relation <

{ (✌,👎),
(✋,✌), ← tuple (pair)
(👎,✋) }

set of tuples
(⊆ ℝℙ𝕊 × ℝℙ𝕊)

- ✌ ∈ ℝℙ𝕊, 👎 ∈ ℝℙ𝕊, ✋ ∈ ℝℙ𝕊
- tuple components ordered
- no order among tuples
- no duplicate tuples

beats

lose	win
✌	👎
✋	✌
👎	✋

← row

bag (multiset) of rows

- **lose** :: text, **win** :: text
- columns ordered
- no order among rows
- possibly duplicate rows

- Read symbol :: as “*has type*”.

#002

Identifying Rows in Tables: Keys 🔑

Tables are unordered: cannot refer to rows by position (~~1st/2nd/… row~~). Instead, use **cell values that uniquely identify its row.**

vehicles

vehicle	kind	seats	wheels?
🚗	car	5	true
🚙	SUV	3	true
🚌	bus	42	true
🚐	bus	7	true
🚲	bike	1	true
🚜	tank	▢	false
🏎	cabrio	2	true
↑ unique	↑ ~~unique~~	↑ ⁇	↑ ~~unique~~

- A **key** is a column (combination) whose values identify rows:

  *"Return the row with a **vehicle** value of 🚐."*

- **vehicle** is a key for **vehicles.** Columns **kind** or **wheels**? are not.

⁇ Column **seats** is not:
1. column contains **NULL** and
2. will values remain unique once more vehicles are added?

- A table *t* may contain multiple candidate key columns. Choose one of these candidates to be **the PRIMARY KEY for** *t*.

#003

Identifying Rows in Tables: Good Keys

Choosing good primary keys is vital in **database schema design:**

1. Key values identify rows uniquely in **any table state**
   (no matter how many rows will be added in the future).
   - Key values should be immutable during row lifetime.

2. Key columns will be used in `WHERE` filter predicates:
   - Aim for **narrow keys** (ideally: single-column).
   - Prefer key columns whose values can be **compared efficiently**
     (e.g., prefer type `int` over `text`, `date`, or arrays).

3. Consider **introducing additional/artificial key columns** if the
   application domain does offer natural/narrow/efficient keys.
   - Examples: book ISBNs, product EANs, social security IDs.

Foreign Keys: Identifying (or: Pointing to) Rows in *Other* Tables

Splitting data between tables helps

- to **avoid redundancy** (and thus data integrity issues), and
- to design tables focused on specific **domain concepts.**

vehicles

vid	vehicle	kind	seats	wheels?	driver
v_1		car	5	true	p_4
v_2		SUV	3	true	p_4
v_3		bus	42	true	▢
v_4		bus	7	true	▢
v_5		bike	1	true	p_2
v_6		tank	▢	false	p_3
v_7		cabrio	2	true	p_4

peeps

pid	pic	name	born
p_1		Cleo	2013
p_2		Bert	1968
p_3		Drew	▢
p_4		Alex	2002

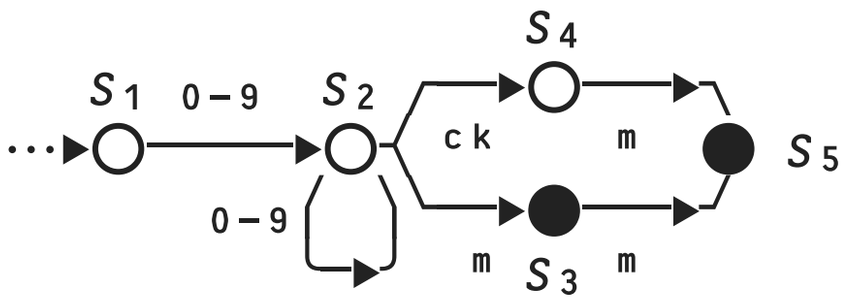
↑

foreign key (references rows in **peeps**, ⊆ peeps(pid))

- Notation: vehicles(vid, vehicle, ..., driver → peeps).

#005

Example: Two Tables Represent a Finite State Machine (FSM)



- FSM accepts metric lengths: '2mm', '135km', '3cm', '42m'.
- Character labels define deterministic transitions.

states

state	start?	final?
s_1	true	false
s_2	false	false
s_3	false	true
s_4	false	false
s_5	false	true

transitions

from	to	labels
s_1	s_2	[0,...,9]
s_2	s_2	[0,...,9]
s_2	s_3	[m]
s_2	s_4	[c,k]
s_3	s_5	[m]
s_4	s_5	[m]

Q: How would you define the **keys**?

Q: Identify **foreign keys** (if any).

Q: Provide SQL **CREATE TABLE** statements for both tables.

4 ┆ *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[2]—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)

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

Tabular Database Systems

②

Tabular Data in CSV Files

March 17, 2026

Torsten Grust
Universität Tübingen, Germany

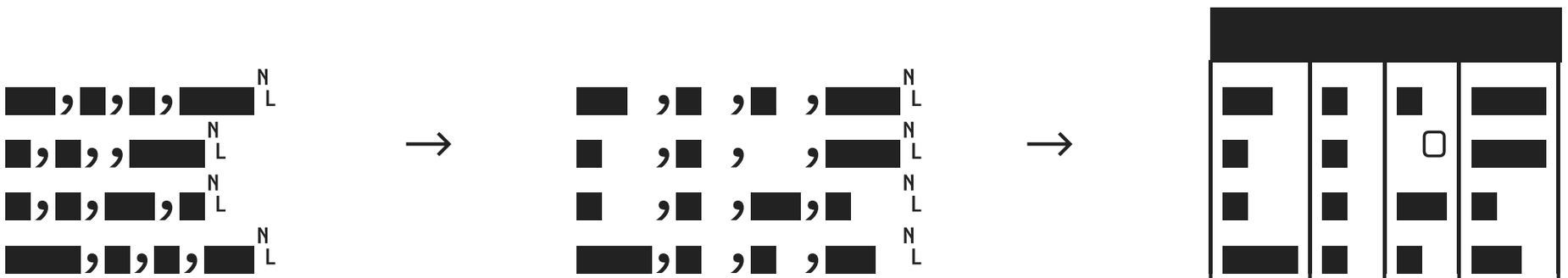
1 Most Data Lives Outside DBMSs

Reality check: The vast majority of data out there does *not* live in files and formats directly accessible for DBMSs:

- Applications read/write their specific data formats which were probably devised before DBMS *X* became available.
- Users process their data using a variety of systems. Why lock-in to the propietary database format of DBMS *X* (or *Y* or 🦆)?
  - Rather build on a “lowest common denominator” format that allows **data interchange between systems.**
- Simplicity wins! DBMS-internal data formats are intricate, heavily optimized for access speed and space efficiency.
  - Rather use formats that admit **simple read/write routines.**
  - Let **humans read/write the data format,** ideally using no tools but their favorite **plain text** or spreadsheet editor.

2 ┆ Tabular Data in Comma-Separated Value (CSV) Files ❶

CSV files (*.csv) provide one such lowest common denominator DBMS-external data format. CSV encodes tabular data in **line-structured plain text** files:



CSV file (plain text) → CSV file (separators aligned) → intended database table

1. Newline (NL) separates lines ☰, a **line** encodes one table **row**.
2. Delimiters (often comma ,) split lines into **columns**.
   - We expect each line to hold the same number of columns, leading to a rectangular grid of cells ▦.

Tabular Data in Comma-Separated Value (CSV) Files ❷

CSV file 007-vehicles.csv to define the column names and contents of table vehicles (see Chapter ①):

#007

```
vehicle, kind, seats, wheels?
🚗, car, 5, true
🚙, SUV, 3, true
🚌, bus, 42, true
🚐, bus, 7, true
🚲, bike, 1, true
🚜, tank,  , false
🏎, cabrio, 2, true
```

- **Notes:**
  - Column names held in first CSV line “by convention.”
    - Otherwise, **header and data rows are identical.**
  - Columns are **untyped** (wheels? “looks like” type boolean).
  - NULL represented by adjacent delimiters(, ,).

Reading CSV Files Like Database Tables

- In DuckDB, read a CSV file much like a database-internal table:

```
D FROM '007-vehicles.csv';
```

vehicles	kind	seats	wheels?

- This is a shorthand for an explicit call to built-in function `read_csv()`:

```
D FROM read_csv('007-vehicles.csv', configuration parameters);
```

vehicles	kind	seats	wheels?

  - Non-default *configuration parameters* may be required to disambiguate CSV file contents regarding presence of headers, choice of delimiters, column types, …)

#008

3 ¦ CSV Enables Data Exchange

A variety of systems export data in CSV format. Paired with the DBMS's CSV import, this enables **CSV-based data exchange.**

- Example: Spreadsheets. On Google Sheets, use command File ▸ ⭳ Download ▸ Comma-Separated Values (.csv):

	A	B	C	D
1	vehicle	kind	seats	wheels?
2	🚗	car	5	☑
3	🚋	trolley	40	☑
4	🛒	trolley		☑
5	🚚	truck	3	☑
6	🚁	helicopter	6	☐
7	🚌	bus	42	☑
8	🚜	tractor	1	☑
9	🚢	boat	4000	☐
10	🛵	scooter	2	☑
11	🚲	bike	1	☑

Access this spreadsheet on Google Sheets

  - **NB.** On export, **data validation rules** (e.g., number ranges, value constraints ☑☐), formulæ, or cell links **are lost.**

#009

4 Importing Data Into Tables, Exporting Query Results Into Files

1. Save repeated imports, use COPY to **copy a file into a table:**

```
COPY table
FROM path_to_file (configuration parameters)
```

   - Can now query *table* like a regular database table.

2. **Copy SQL query results into a file** external to the DBMS:

```
COPY (query)
TO   path_to_file (configuration parameters)
```

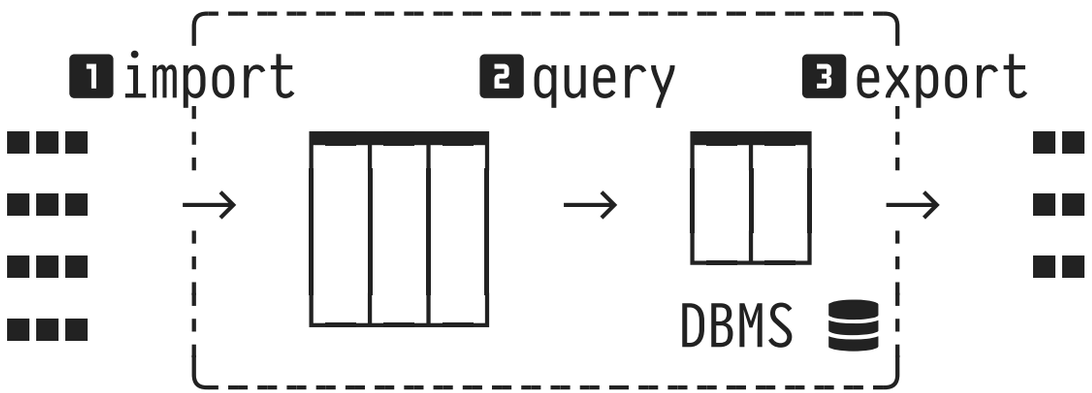
The *configuration parameters* determine file format (e.g., CSV), delimiters, header output, compression, overwrite/append mode, …

Using DuckDB as a CSV Processor

DBMS may act as a processing engine for database-external data:

+ SQL is versatile, expressive, and executed efficiently.
− Need to pay for data import and export.

#010



❶

```
D COPY (SELECT   ...
        FROM     input_file
        WHERE    ...
        ORDER BY ...)
  TO   output_file;
```

❷

```
D COPY table
  FROM input_file;
D COPY (SELECT   ...
        FROM     table
        WHERE    ...
        ORDER BY ...)
  TO   output_file;
```

5 ¦ Is CSV Too Simple? Ubiquitous, But Fragile

The CSV format is documented in RFC 4180. Still, CSV files in daily practice deviate in a variety of ways[1], including

- choice of **delimiters** (frequent: `,` `;` `|` (pipe) `␉` (tab)),
- **quoting** (enclose columns with significant spaces `␣` in `"` or `'`),
- character **escape** sequences (what if `,` `"` `␊` are in columns?),
- absence/presence of **headers or preamble text** above data rows,
- choice of representation of **missing values** (e.g., `…,,…`), or
- a **mixture of data types** in a column.

DuckDBs aims to auto-adapt to these CSV dialects using a "multi-hypothesis sniffing" algorithm[2].

[1] DuckDB's CSV reader comes with 25+ configuration options. To get an impression of the variety of CSV dialects out there, see the paper Characteristics of Open Data CSV Files (2016).

[2] See the blog post DuckDB's CSV Sniffer: Automatic Detection of Types and Dialects and the paper Multi-Hypothesis CSV Parsing (2017) by Hannes Mühleisen *et al.*

DuckDB's CSV "Sniffer"

1. **Sample 20480 lines** from all over the file (unless we need to read top to bottom, e.g., if input is stdin or compressed).
2. Try **CSV dialects** based on 24 selected combinations of the options below. Choose the CSV dialect options that lead to a *consistent and maximum number of columns* per row:

parameter	options
delim	, | ⭾ ;
quote	" ' (empty)
escape	" ' \ (empty)

3. **Detect types.** Try to cast values to candidate types *in order*: [(NULL,) boolean, int, double, time, date, timestamp, text].
4. **Header detection:** Do columns in first row match detected types (*is data*, thus generate own header) or not (*is header*)?

DuckDB's CSV “Sniffer”: Dialect and Type Detection

Dialect detection: #011

flights.csv

```
FlightDate|Carrier|Origin|Destination␤
1988-01-01|AA|New York, NY|Los Angeles, CA␤
1988-01-02|AA|New York, NY|Los Angeles, CA␤
1988-01-03|AA|New York, NY|Los Angeles, CA␤
```

# columns	,	␉	;
1	4	1	1
3	4	1	1
3	4	1	1
3	4	1	1

Type detection (file reading order ⋯▶):

band.csv

```
Name, Age
,
Mike Lindup, 65
Mark King, 66.2
```

	candidate types for column 0	candidate types for column 1
▼	(skip)	(skip)
	[NULL,boolean,...,text]	[NULL,boolean,...,text]
	[text]	[int,double,...,text]
	[text]	[double,...,text]
	↑	↑

Tabular Database Systems

③

Reading Data at the Speed of ~~Light~~Memory

March 17, 2026

Torsten Grust
Universität 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 🔲 / 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.”

[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 its 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 (2 Secondary/1 Primary)	Read Bandwidth	⏱ Query Time
(Ethernet)	2.5 GB/s	0.28s
2 External USB-C SSD (2 TB)	800 MB/s	0.90s
2 NVMe SSD (2 TB)	5 GB/s	0.14s
1 DRAM (64 GB)	21 GB/s	0.03s

- **NB.**
  - **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 |
         sum = 0 }                 ┆  match first line, reset sum
       { sum = sum + $5 }          ┆  match any line, sum 5th column
END    { print sum }               ┆  match last line, output sum
```

01 #012

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

```
$ time ./012-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

with open(sys.argv[1], 'r') as file:
    for line in file:
        sum = sum + int(line.split('|')[4])

print(sum)
```

```
reset sum                    #013

open file (reading)
read line by line
| extract 5th col,
| cast to int, add
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).

#014

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

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

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

[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%

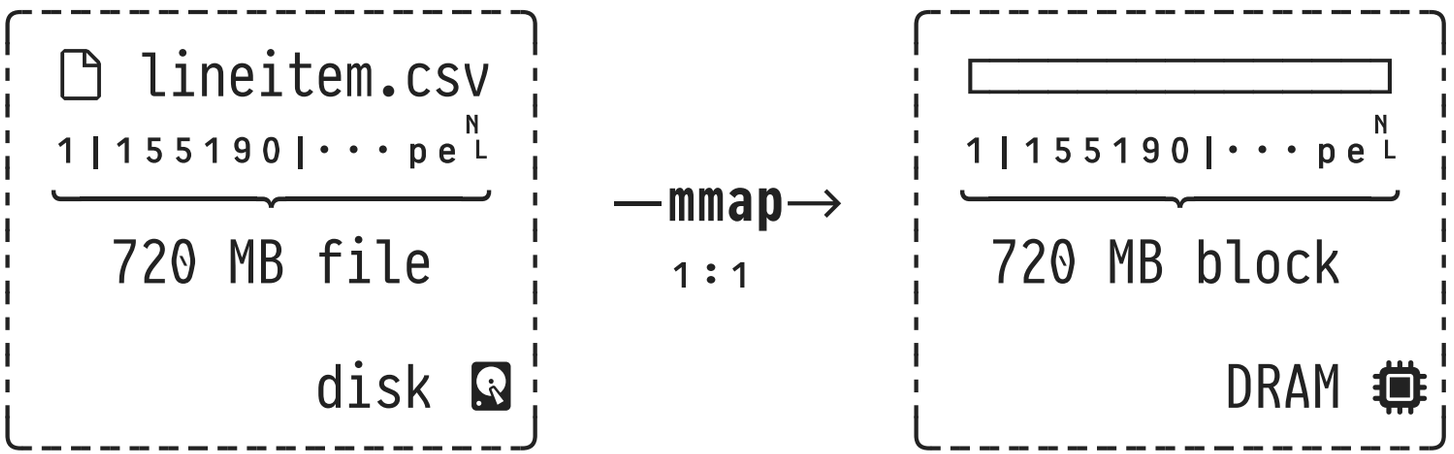
- Reading the CSV file line by line is too slow.
  - 💡 Read entire file at once, impose line structure ourselves.

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) 1.
- No `getline()`: need to locate ␤ (`'\n'`) 2 on our own now:

```
column = 1;

while (data < end) {
  switch (*data) {
    case '|' : column++; break;
    case '\n': ...
  }
  data++;

  if (column < 5)
    continue;

  sum = sum + atoi(data);

  column = 1;
  while (*data++ != '\n');
}
```

```
start in column 1                    #015

scan memory block, byte by byte

1 found |: next column begins
error: line has too few columns

proceed through memory block

reached column 5 already?
  no, keep scanning

convert to int (up to |,␤), add

next line starts with column 1
2 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 🔤 2️⃣ dominates. 💡 Use strchr(data, '\n') instead.
  - ⏱ Query time: 0.27s (throughput 2.8 GB/s).
  - How can strchr() be so efficient?

01 #015

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]

#016

[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 '\n' Search

#017

```
while (data < end) {
  ⋮
  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 '\n' Search

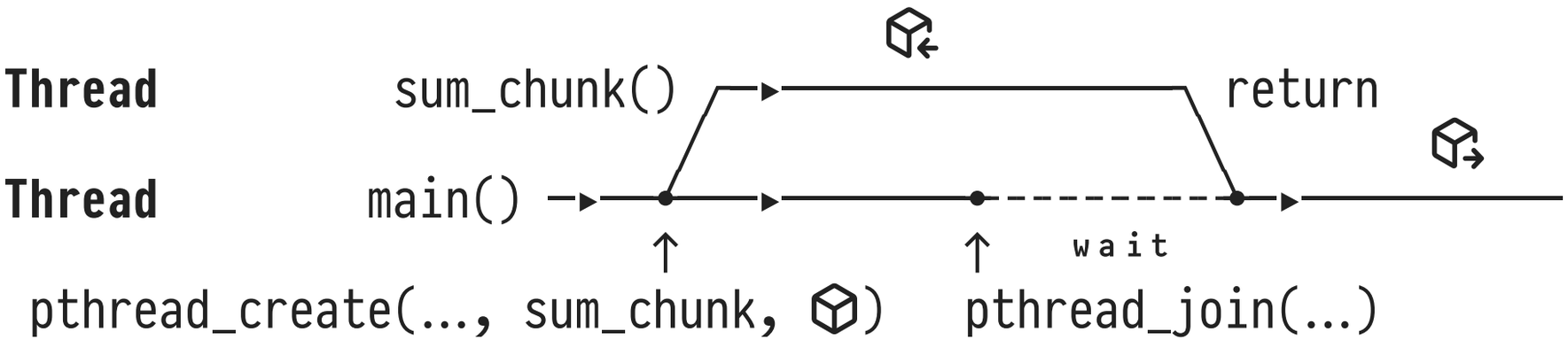
- ⏱ Query time on Torsten's computer (warm cache): ≈0.28s.
  - C with mmap() and block-wise search for '\n' processes the CSV file with a throughput of 2.8 GB/s.
  - We match the performance of the built-in strchr() function.

- 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 `\n` 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): 💾 #018

```
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 Interlude: SQL Aggregate Functions

SQL aggregates[5] condense a bag of rows into a *single* value:

vehicles

vehicle	kind	seats	wheels?
🚗	car	5	true
🚙	SUV	3	true
🚌	bus	42	true
🚐	bus	7	true
🏍	bike	1	true
🚜	tank	▢	false
🏎	cabrio	2	true

- Most aggregates ignore (= skip) NULL (▢) values.
  - An optional **FILTER** clause can control value inclusion.
- Order-aware aggregates use clause **ORDER BY** to be deterministic:

  list(vehicle **ORDER BY** seats)

bool_and("wheels?") ≡ false
max(seats) ≡ 42
arg_max(kind, seats) ≡ 'bus'
count(vehicle) ≡ 7

[5] DuckDB's SQL dialect features an extensive list of built-in aggregate functions.

Interlude: SQL Aggregate Functions

- Let us use annotations to explain SQL constructs (here: query clause **row cardinality, 1 2 3:** clause processing order[6]):

```
SELECT expr,...,expr      ┐ 3 m
FROM   t                  ├ 1 n (= |t|, cardinality of t)
WHERE  p                  ├ 2 m (⩽ n)
                          └── # rows returned by query clause
```

- SQL aggregates reduce row cardinality to 1:

```
SELECT agg,...,agg        ┐ 3 1
FROM   t                  ├ 1 n (= |t|)
WHERE  p                  ├ 2 m (⩽ n)
                          └── # rows
```

  - ⇒ **SELECT** clause *cannot mix* scalar *expr* and aggregates *agg*.

#019

[6] SQL clause processing order ≠ SQL syntactic order. DuckDB implements a "friendly SQL dialect" to partly rectify this (e.g., allowing FROM-SELECT queries). We'll use such DuckDB-specific features sparingly.

10 ┆ Sum of Quantities ❼ — SQL

Aggregate function sum() is what we need to formulate a SQL variant of the benchmark query over the CSV file:[7]

```
D .timer on
D SELECT sum(l_quantity)
  FROM   read_csv('lineitem.csv',
                  header = false,
                  names = [ ..., 'l_quantity', ... ])
```

#020

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.**

[7] 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**[8] that exposes details of the system's **query evaluation plans:**

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

```
D EXPLAIN
  query;
```

Do *not* run query. Show query plan and estimated row count.

```
D EXPLAIN ANALYZE
  query;
```

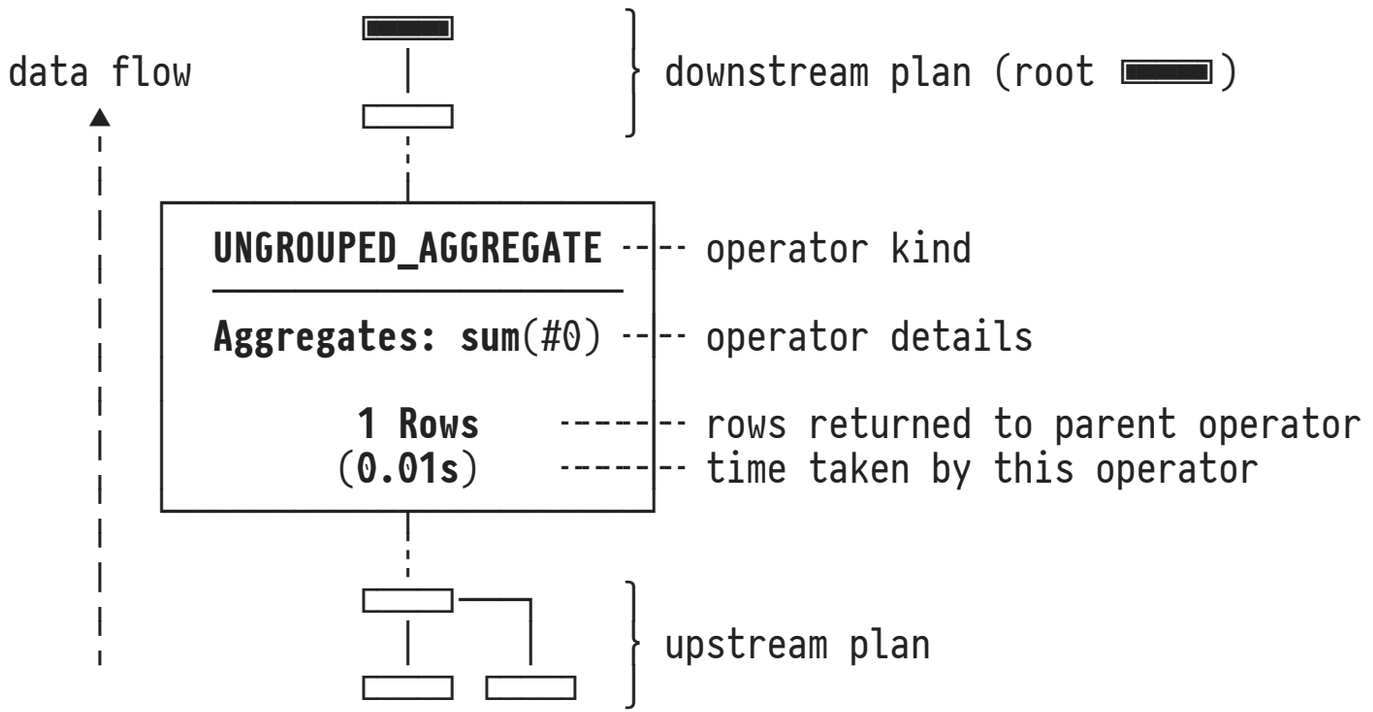
Actually run query. Measure times/rows, annotate plan.

[8] 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:**[9]

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



[9] 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 ❼ — SQL

```
D SELECT SUM(l_quantity)                                    #020
  FROM   lineitem;
┌──────────────────┐
│ sum(l_quantity)  │
├──────────────────┤
│    153078795     │
└──────────────────┘
Run Time (s): real 0.002 user 0.007288 sys 0.000294
```

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
- the data for table `lineitem` does reside in DRAM (not in secondary storage).

Tabular Database Systems

④

Columnar Table Storage

March 17, 2026

Torsten Grust
Universität Tübingen, Germany

1 Transient vs. Persistent Databases

- **Transient in-memory databases**

  By default, tables are created and stored in volatile **DRAM** 🖳. When the DuckDB process ends, all table data is lost.

```
$ duckdb
  ⋮
Connected to a transient in-memory database.
D
```

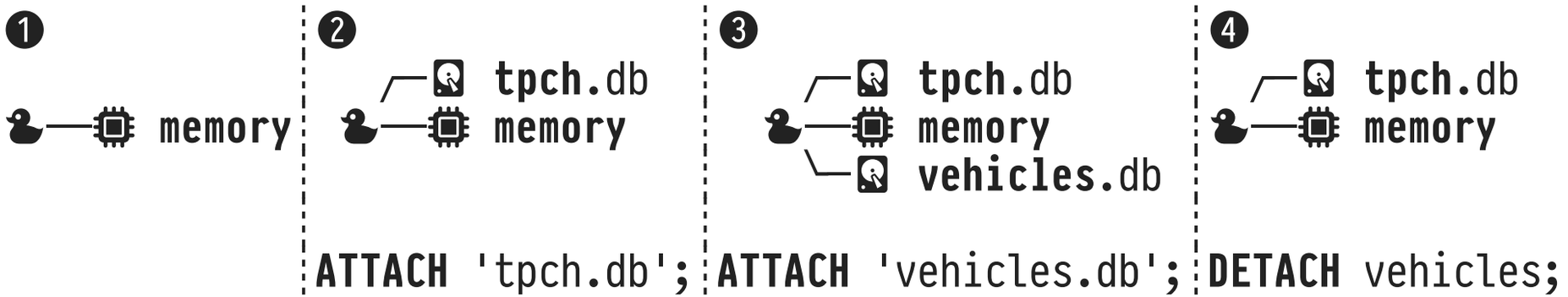
- **Persistent on-disk databases**

  Table data is held in a **file on secondary storage** 🖴[1]. When the DuckDB process ends and later restarts, the current state of the database **persists** (= is preserved between DBMS sessions).

[1] DuckDB stores the instances of all tables and associated data in a self-contained *single regular file*, conventionally named ‹*database*›.db, ready to be posted/shipped. Other DBMSs (like PostgreSQL) store databases in subdirectory structures or directly in disk blocks (“raw” storage, no filesystem).

One DBMS, Multiple Databases

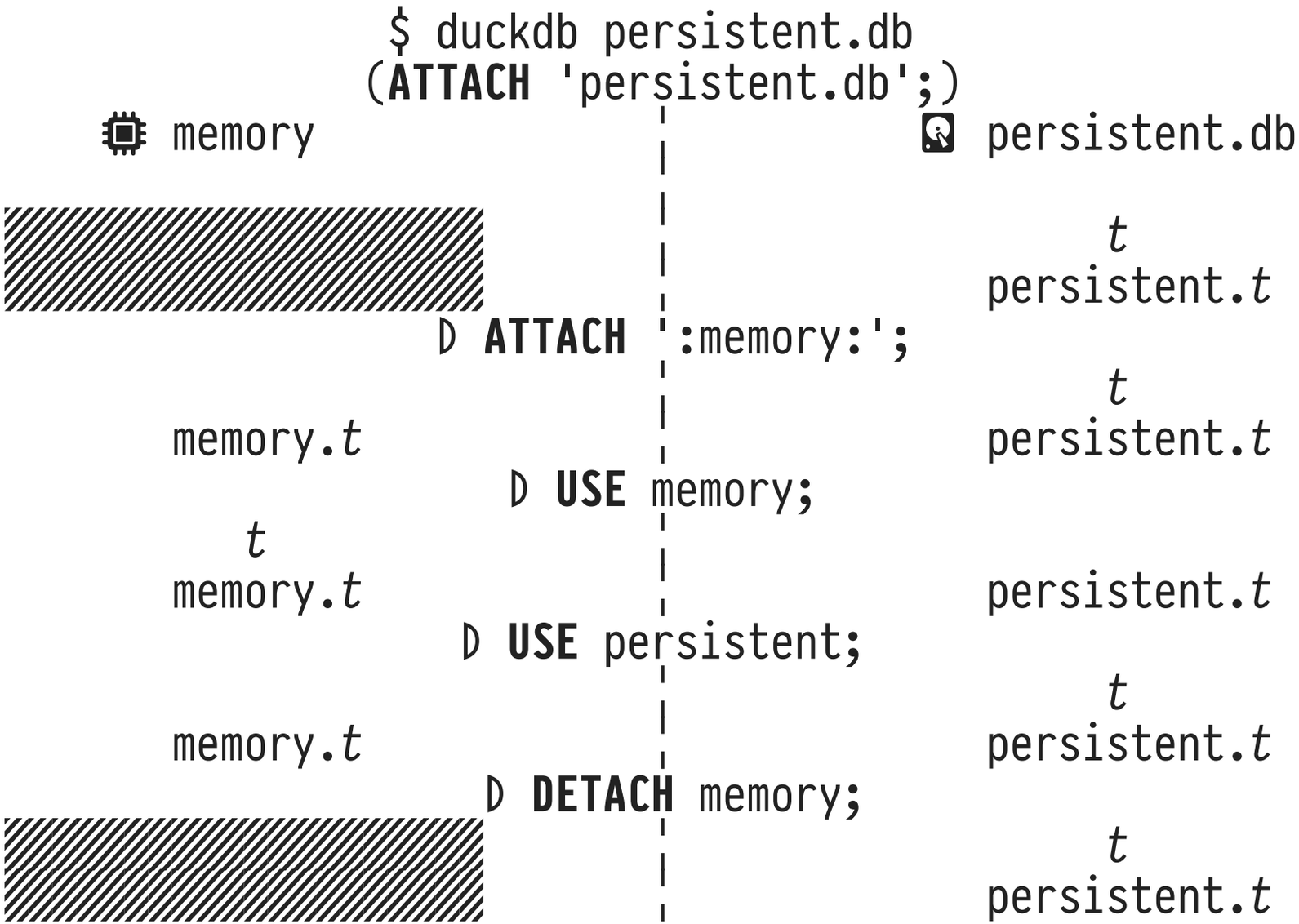
- **A DuckDB process** can *attach* to **multiple databases** at one time.
  - A **database** comprises
    - schema and state (i.e., bag of rows) of all tables,
    - other user-defined objects (types, views, macros), and
    - installed DBMS extensions.



- Objects (e.g,. tables) live in database-owned **namespaces:**
  - Name *t* refers to table *t* in the default database (see USE).
  - Name tpch.*t* refers to table *t* in database tpch.

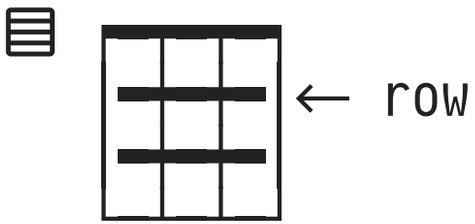
DuckDB: Attaching to/Detaching From Databases, Access Table *t*

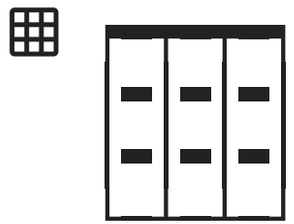
#021



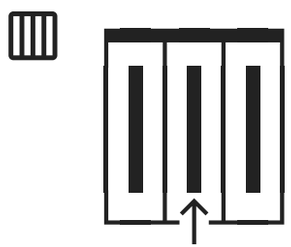
2 ┆ Table: A Collection of Rows or Columns?

- The tabular data model appears to have its focus on **rows:**
  - ▤**:** Instances of tables are bags of **rows** ▬**.**
  - ▦**:** Each **row** is split into individual cell values ▪ ▪ ▪**.**
  - SQL clause `FROM` *t* iterates over the **rows** of *t***,** evaluates `WHERE`/`SELECT` clause per **row,** `ORDER BY`/`LIMIT` sort/count **rows.**





- Yet, DBMSs like ⛁ organize the **storage** of tables by **column:**
  - ▥**:** Values in a **column** appear consecutive in memory/on disk.



  - *n* (= 3) columns jointly form a table.
  - All columns have the **same number** of entries.
  - All entries in a column have the **same type.**

Queries That Focus on Few Columns (But Read All Rows)

- Observation: a large class of SQL queries touch **few columns** but (almost) **all rows** of a table. These are known as **OLAP**[2] **queries.**
  - The benchmark query over table lineitem from Chapter 03.
  - Aggregation queries scan all entries of their source column:

vehicles

vehicle	kind	seats	wheels?
○	car ○	5 ✓	true ○
○	SUV ○	3 ✓	true ○
○	bus ○	42 ✓	true ○
○	bus ○	7 ✓	true ○
○	bike ○	1 ✓	true ○
○	tank ○	▢ ✓	false ○
○	cabrio ○	2 ✓	true ○

Q: **SELECT** max(seats)
FROM vehicles

✓ relevant cells (25%)
○ no contribution to result

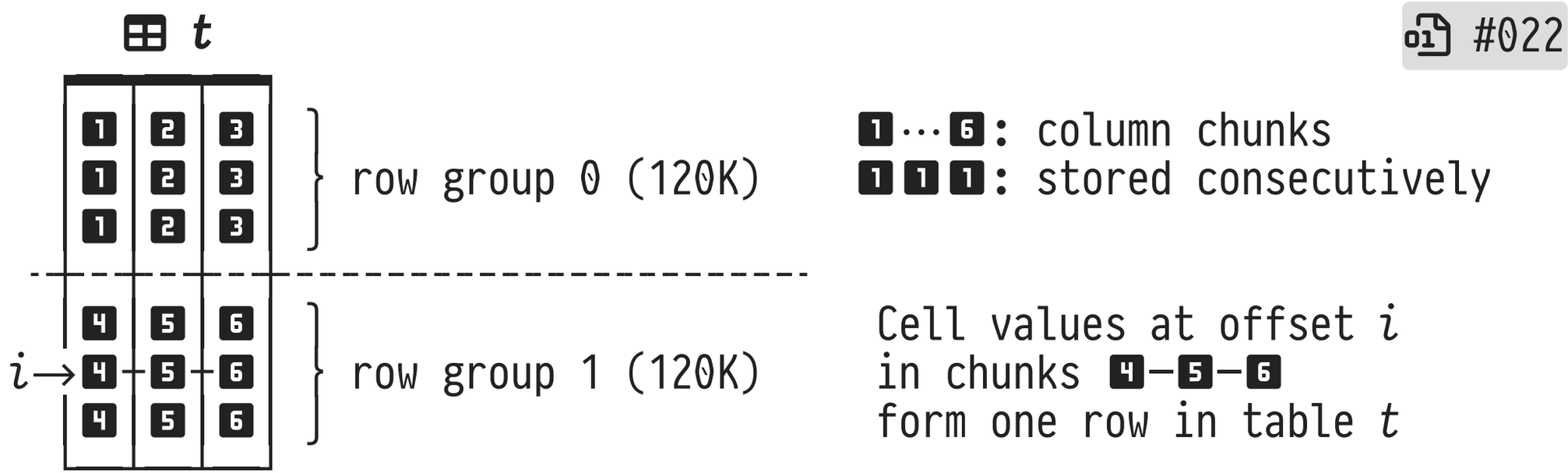
- Only read column **seats** in TABLE_SCAN in plan for query **Q.**
- Row-wise table storage would read all cells (but ignore 75%).

[2] **O**nline **A**nalytical **P**rocessing, as opposed to OLTP (Online Transactional Processing) queries which touch few (even single) rows but read all those rows' columns.

3 DuckDB: Columnar Table Storage

DuckDB storage format for table t with r rows and c colums:

1. **Row groups:** Partition table t horizontally into groups of 120K (120 × 1024 = 122880) rows $\Rightarrow$ t has $\lceil r \,/\, 120K \rceil$ row groups.
2. **Column chunks:** Inside each row group, store a chunk (of 120K cell values) for each of the c columns.



4 ¦ Column Compression

Tables may exhibit repeating cell values or redundancies across rows that make instances worthwhile targets for **compression.**

- Compression (and decompression) **adds CPU effort** but reduces the storage footprint and may thus **save disk I/O bandwidth.**
  Overall, performance may improve.
  - DuckDB applies compression to on-disk databases only.

- **Columnar table storage works well with compression:**
  - Within a column (chunk) 1️⃣1️⃣1️⃣, values are homogenously typed and potentially similar. This aids compression algorithms.
  - (In row-wise storage, values of different types are interleaved 1️⃣2️⃣3️⃣, leading to lower compression rates.)

- For some tables, the effect of compression can be drastic. 👍

Compression: General vs. Lightweight

1. **General-purpose compression** algorithms (e.g., *gzip*, *zstd*):
   - Detect and exploit **patterns in arbitrary bit sequences:**
     - 11111111: predictable, compressible as 8×1 (run length).
     - 10110010: noisy, random, not compressible.
   - High compression rates, but (de)compression is **costly.**
   - Work best on **large chunks of data** (>> 256kB)—decompressing sizable chunks renders accessing individual rows expensive.

2. DuckDB builds on a family of **lightweight compression** schemes:
   - Detect **patterns in typed data** in a column chunk:
     - 10 12 9 10 8 11: compressible as 8 ⊕ 2 4 1 2 0 3 (values close to reference 8).
   - (De)compression is **cheap** and incurs light CPU load only.
   - Effective on small data: DuckDB scans column chunks to select best compression algorithm **per row group** (120K rows).

DuckDB: Lightweight Compression Schemes ❶

- **Constant Encoding:**
  - Applies if **every value** in column chunk is the **same.**
  - Example: ranges of NULLs or rarely changing values (e.g., year in the timestamp of log entries).

Uncompressed	Constant Encoding
2025 2025 2025 2025 ⋯	2025

- **Run-Length Encoding** (RLE):
  - Compress **groups of repeated values** using (*count*,*value*) pairs.
  - Example: sorted or partitioned data.

Uncompressed	Run-Length Encoding
a, a, a, a, b, b, c, c, c	(4,a), (2,b), (3,c)

DuckDB: Lightweight Compression Schemes ❷

- **Bit Packing:**
  - Exploit that values do **not span full domain of their type.**
  - Max value determines bit width, then elide leading 0 bits.

Uncompressed (type smallint, int2: 16 bits)	Bit Packing (6 bits)
0000000000101010, 0000000000010011, 0000000000000100 42 19 4	101010, 010011, 000100

- **Frame of Reference** (FOR):
  - Store **Δs from a reference** (**minimum**), not absolute values.
  - Example: dates close to one point in time. Absolute values are days since 1970-01-01, Δs to min date will be smaller.

Uncompressed	FOR Encoding
1968-08-26, 1968-08-24, 1968-08-27	1968-08-24 ⊕ 2 0 3 reference Δs

DuckDB: Lightweight Compression Schemes ❸

- **Dictionary Encoding:**
  - Place **frequent values in dictionary,** only store # of dictionary entry. Effective if values are wide (strings).

Uncompressed	Dictionary Encoding
Zelda, Mario, Mario, Zelda, Zelda	[$_0$Zelda,$_1$Mario] (dictionary) 0, 1, 1, 0, 0 (entry #s)

- **Fast Static Symbol Table Encoding** (FSST):
  - Extends dictionary encoding to capture **frequent substrings.**
  - Example: URLs or e-mail address strings.

Uncompressed	FSST Encoding
www.archive.org, www.duckdb.org	[$_0$www.,$_1$.org,$_2$archive,$_3$duckdb] (symbol table) (0 2 1), (0 3 1) (entry #s)

– **Compression of IEEE 754 Floating-Point Values** (ALP[3]).

#023

[3] XORing similar doubles typically leads to a high number of leading/trailing 0 bits. See the paper ALP: Adaptive Lossless Floating-Point Compression (2023). ALP is implemented in DuckDB.

Tabular Database Systems

⑤

Database-External Data in Parquet Files

March 17, 2026

Torsten Grust
Universität Tübingen, Germany

1 ¦ Columnar Compressed Data Storage Outside the DBMS: Parquet

Fragility, space requirements, and parsing effort render CSV files problematic for applications that read/write GBs or TBs of data.

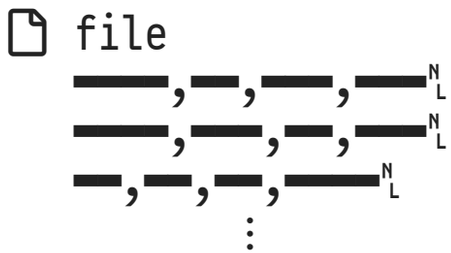
Alternative **database-external file formats** have been developed to address these issues. Among these, **Parquet**[1] is widely used.

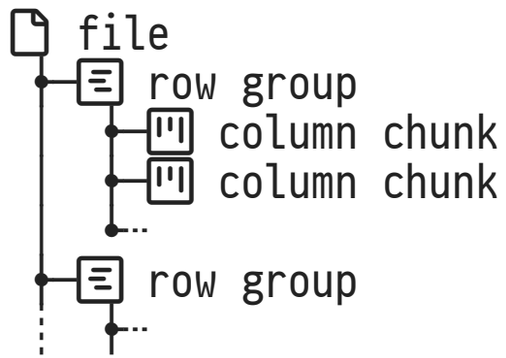
- Developed in the open since 2013 (initiated by Twitter *et al*).
- Stores **columns of typed data.**
- Built-in **compression** (on file and column levels).
- Incorporates **rich metadata** that supports projection and selection pushdown.
- Supported by libraries for a wide range of programming languages. Directly readable/writable by DuckDB.

[1] Apache Parquet (sponsored by the Apache Software Foundation): open source, column-oriented data file format designed for efficient data storage and retrieval.

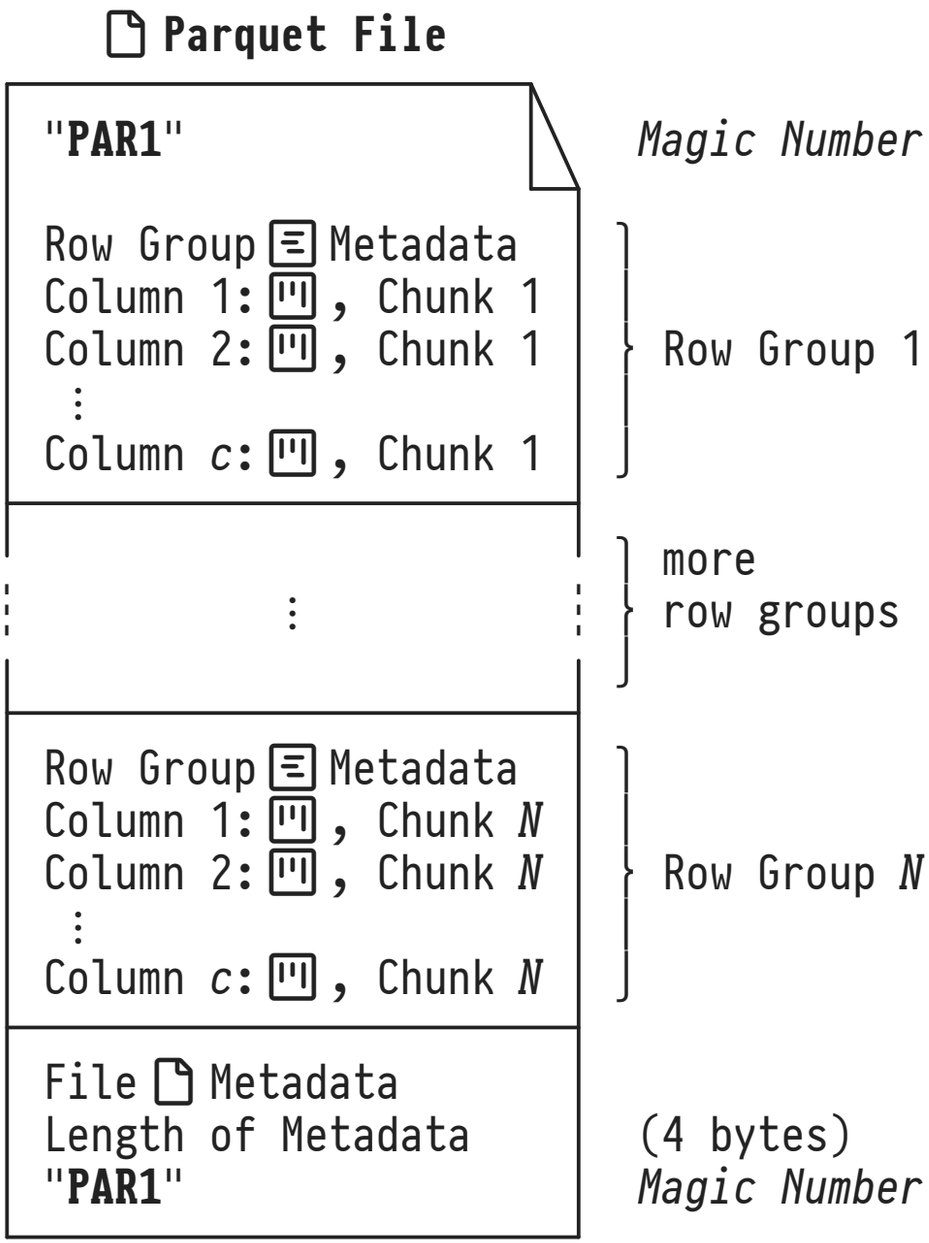
CSV vs. Parquet

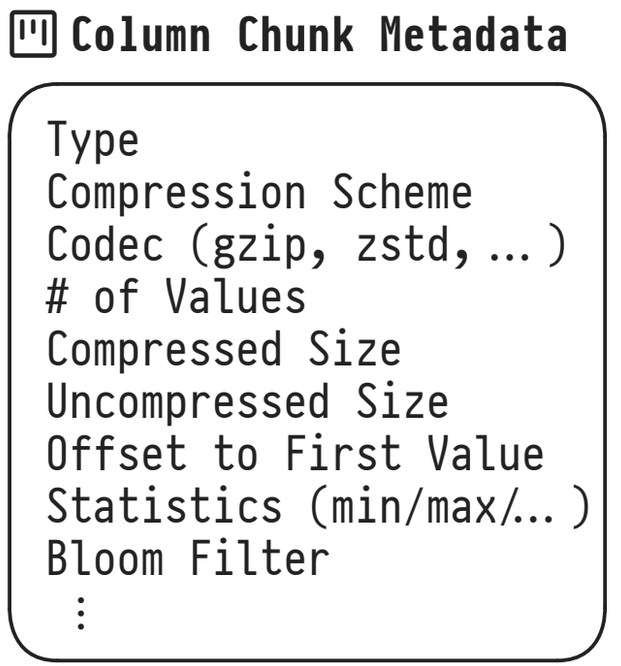
CSV	Parquet
monolithic	split into horizontal row groups
row-oriented (lines separated by ␤)	column-based (chunk-by-chunk)
plain text, uncompressed	supports compression: • file-level (gzip, zstd, …) • column-level (dictionary, RLE, …)
untyped (requires parsing)	typed: • scalar (INT32/64, FLOAT, BYTE_ARRAY, …) [• nested records]
no metadata (optional header row)	metadata for file/row groups/columns: • file format version • min/max/count statistics • cardinality, (un)compressed byte sizes

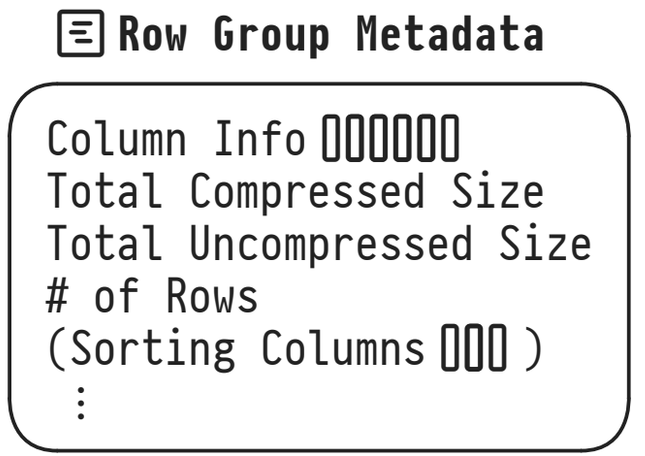




A Sketch of Parquet's Storage Format



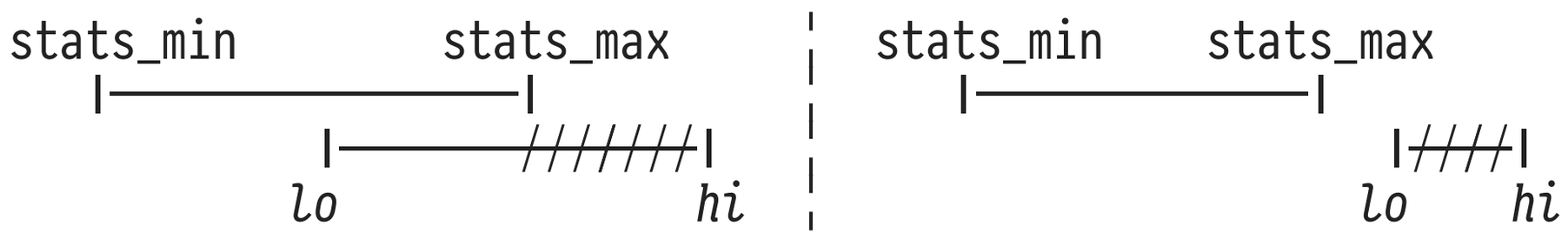




2 Pushing Projection and Filtering Down into Parquet Reading

Parquet's file structure + metadata enables readers (like DuckDB's PARQUET_SCAN operator) to **only access relevant data subsets.**

- **Projection pushdown:** Exploit **column-based layout.** In each row group, use data_page_offset to navigate the file to only read required column chunk(s).
- **Filter pushdown:** Exploit **statistics metadata.** Entirely skip row group if stats_min/stats_max[2] for column *c* indicate that filter predicates like $lo \leqslant c \leqslant hi$ will always fail (~~//~~).

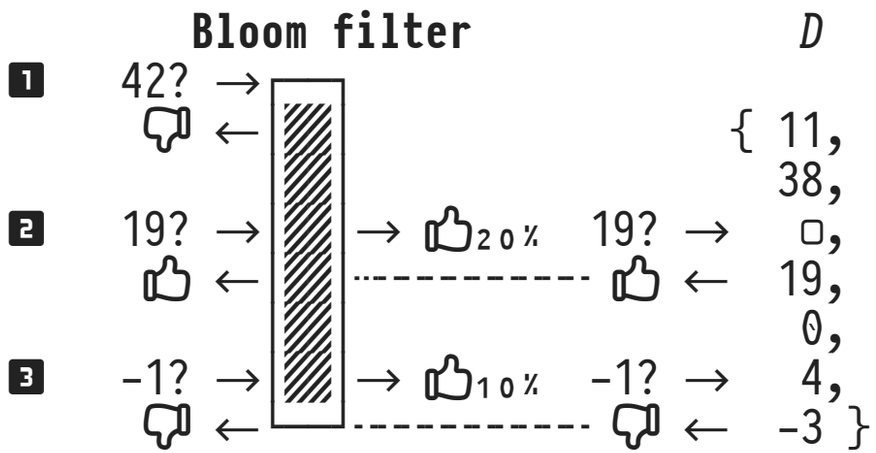


[2] In the DB research literature, the (stats_min,stats_max) pairs in all row groups are sometimes collectively referred to as **zone map** (for column *c*).

3 Optional in Parquet Files: Bloom Filters

If column values are randomly shuffled, all values may occur in all row groups and the effectiveness of zone maps is largely lost.

- **Bloom filters** are *compact* data structures that *over-approximate* the set D of distinct values[3] in a row group.
- Given the question $x \in D?$, Bloom filters either respond with
  - *definitely no* (👎) or
  - *probably yes* ($👍_{p\,\%}$, where p is a false positive rate).

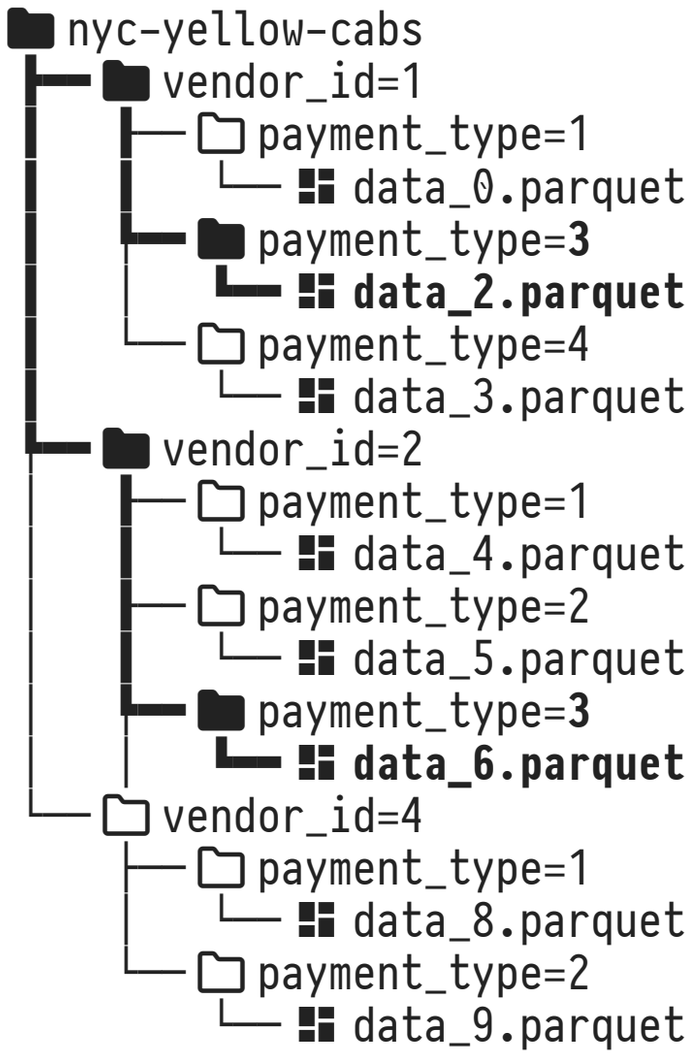


#026

[3] Set D is the ***active* domain** of the row group. Example: if the values in a column have type int32 with domain $[-2^{31},2^{31})$, we typically have $D \Subset [-2^{31},2^{31})$ (read $\Subset$ as "significantly smaller").

4 ¦ Partitioned Files and Filter Pushdown (Hive Partitioning)

If data volumes get truly large, it may make sense to **partition rows** into a family of files. Which partition will a row belong to?



- Choose $n \geqslant 1$ columns $c_1,\ldots,c_n$ as partition criteria (left: n=2, vendor_id + payment_type).
- Arrange partitions in **tree hierarchy:**
  - Depth of tree: n+1.
  - Width of tree $\leqslant$ product of the size of active domains of columns c_i.
- Inner nodes: 🗀 OS directories, leaf nodes: ▪ data (Parquet or CSV).
- Filter query: traverse relevant subtrees/read relevant data files only (left: ┣━ 🖿 ≡ payment_type = 3).
- Particularly useful when I/O is slow.

Tabular Database Systems

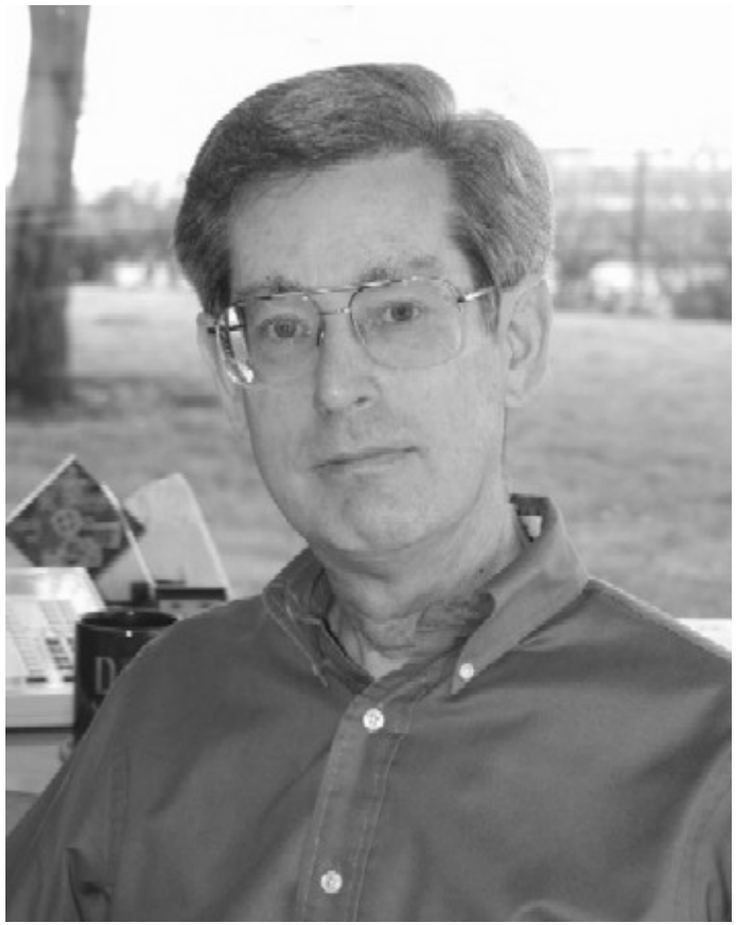
⑥

The Structured Query Language (SQL)

March 17, 2026

Torsten Grust
Universität Tübingen, Germany

1 The Origins of SQL



Don Chamberlin

Ray Boyce (†1974)

Don Chamberlin and **Ray Boyce,** co-inventors of SQL, back in 1972/73 both members of IBM's research project *System R*.

The Origins of SQL

- Development of the language started in 1972, first as **SQUARE**, from 1973 on as **SEQUEL** (*Structured English Query Language*). In 1977, SEQUEL became **SQL** because of a trademark dispute. (Thus, both "*S–Q–L*" /ˌɛskjuːˈɛl/ and "*sequel*" /ˈsiːkwəl/ are okay pronounciations.)

- First commercial implementations in the late 1970s/early 1980s. By 1986, the ANSI/ISO standardization process begins.

- Since then, SQL has been in active development and remains the ***"Intergalactic Dataspeak"***[1] to this day.

  Current SQL standard (as of April 2025): SQL:2023.

[1] Due to Mike Stonebraker, inventor of Ingres (1972, precursor of Postgres, PostgreSQL)

2 ┆ SQL: Row Variables

Like most regular programming languages, SQL features a construct to **bind variables to values.**

- `FROM` is the *only* SQL clause that can introduce variables:

```
 ⋮
FROM t AS v -- bind variable v to the rows in table t
 ⋮
```

  - If $|t| = m$, v iterates over all m rows of t in some order. v is thus known as **row variable** (sometimes: table alias 👎).
  - If table t has schema $t(c_1\ \tau_1,\ldots,c_n\ \tau_n)$, then v has type $\text{row}(c_1\ \tau_1,\ldots,c_n\ \tau_n)$.[2]
  - Can access column c_i of v using **dot notation:** $v.c_i$ $(::\ \tau_i)$.

[2] In DuckDB, $\text{row}(c_1\ \tau_1,\ldots,c_n\ \tau_n)$ is a synonym for type $\text{struct}(c_1\ \tau_1,\ldots,c_n\ \tau_n)$: think of a table row like a record/struct with n fields.

FROM generates Cross Products

- If a SQL query needs to read data from **multiple tables** $t_1,\ldots,t_m$, list all tables in the FROM clause:

```
  ⋮
FROM t₁ AS v₁, …, tₘ AS vₘ -- row variable names vᵢ unique
  ⋮
```

  - **Cross product** semantics: This generates **all** $|t_1| \times \cdots \times |t_m|$ **combinations** of bindings for row variables v_i in some order.
  - Row variable names v_i are unique, tables t_j may repeat (FROM t AS v_1, t AS v_2 allows to combine each row of t with its peers).

- ⚠️ Yes, FROM clauses may generate *MANY* bindings.
  Typical queries will use predicates on the v_i to only focus on the combinations that make sense.

#029

3 ¦ SQL: Joining Tables

Cross products between tables are (too?) general: it is common to draw rows from tables based on *conditions* that identify (un)wanted row combinations. SQL supports several such **table joins.**

- **Inner Join:**[3]

```
  ⋮
FROM t₁ AS v₁ [INNER] JOIN t₂ AS v₂ ON (p)
  ⋮
```

  - Draw all combinations of rows from tables t_1, t_2.
  - Only keep those bindings of v_1, v_2 that satisfy **predicate** p (v_1, v_2 typically do occur free in p).

[3] Inner join is often referrered to as simply *join* or *0-join* (initially, DB literature used the fancy greek theta θ instead of p to denote the join predicate).

Inner Join Follows Foreign Keys

vehicles

vid	vehicle	kind	seats	wheels?	pid
v_1	🚗	car	5	true	p_4
v_2	🚙	SUV	3	true	p_4
v_3	🚌	bus	42	true	▢
v_4	🚐	bus	7	true	▢
v_5	🚲	bike	1	true	p_2
v_6	🚜	tank	▢	false	p_3
v_7	🏎	cabrio	2	true	p_4

peeps

pid	pic	name	born
p_1	👩	Cleo	2013
p_2	👴	Bert	1968
p_3	🥷	Drew	▢
p_4	👨	Alex	2002

```
SELECT v.vehicle, p.name AS driver, p.pic
FROM   vehicles AS v INNER JOIN peeps AS p ON (v.pid = p.pid)
                                              "equi-join" ↗
```

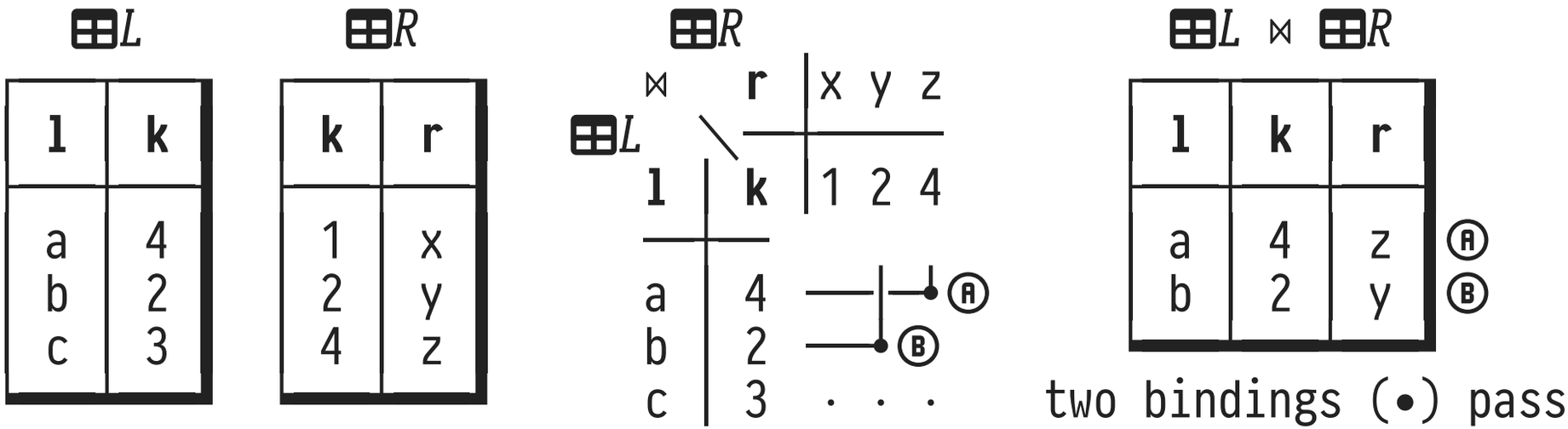
- Some rows in table peeps find multiple *join partners* in vehicles (like p_4 👨), some find none (p_1 👩).
- General: A join between t_1 and t_2 may yield 0…$|t_1|\times|t_2|$ rows.

More Join Operations ❶

Inner joins (in particular: equi-joins) are frequent. Daily SQL practice calls for a whole **family of join variants,** however.

To introduce the join family, let us use a sketch of their semantics (see below). We assume join predicate $p \equiv L.\mathtt{k} = R.\mathtt{k}$.

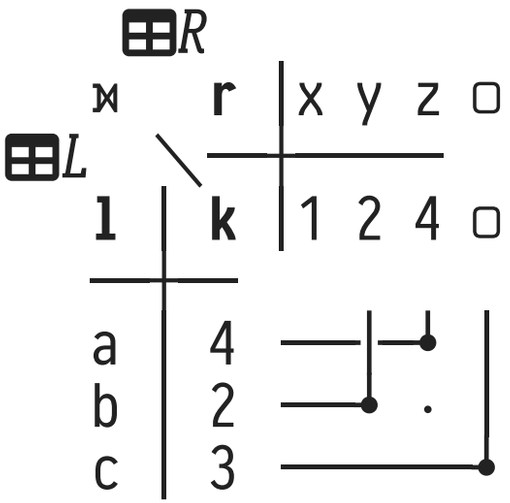
- **INNER JOIN** (⋈, `FROM L INNER JOIN R ON (L.k = R.k)`):



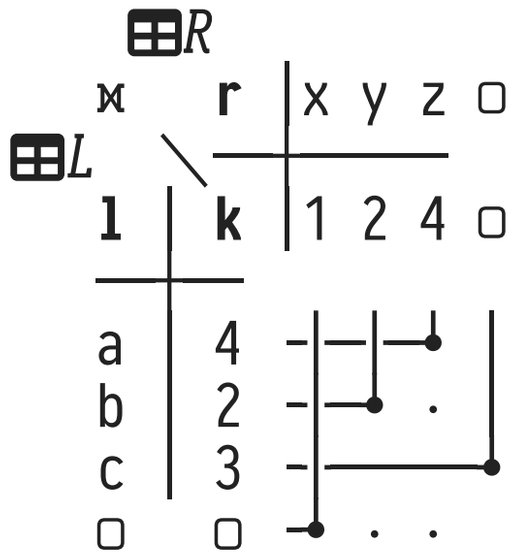
More Join Operations ❷: Outer Joins

- **LEFT/RIGHT/FULL OUTER JOIN** (⟕, ⟖, ⟗)

LEFT OUTER JOIN



FULL OUTER JOIN



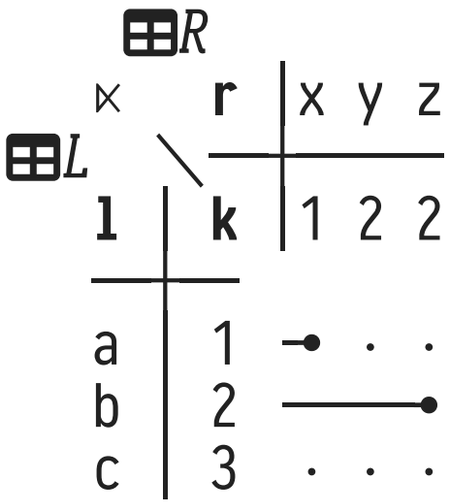
- Left outer join: *All rows* of the left table 𝌆*L* are kept. Rows from 𝌆*L* that find no join partner (like (c,3)) are paired with an artificial all-NULL (▢) row.

01 #031

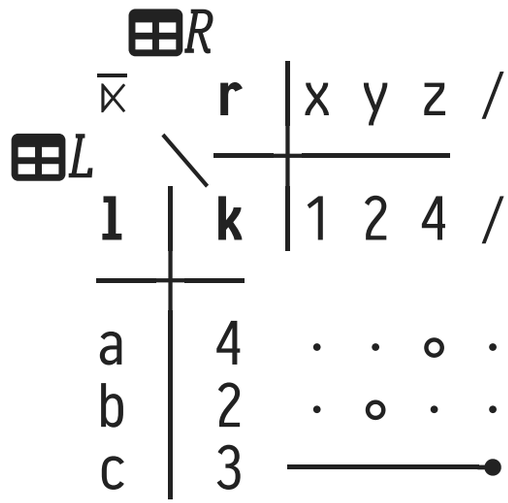
More Join Operations ❸: Semi/Anti/Cross Joins

- **SEMI/ANTI/CROSS JOIN** ($\ltimes$, $\overline{\ltimes}$, $\times$)

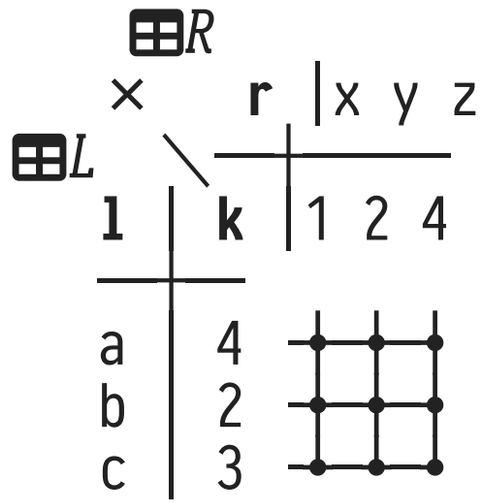
SEMI JOIN



ANTI JOIN



CROSS JOIN



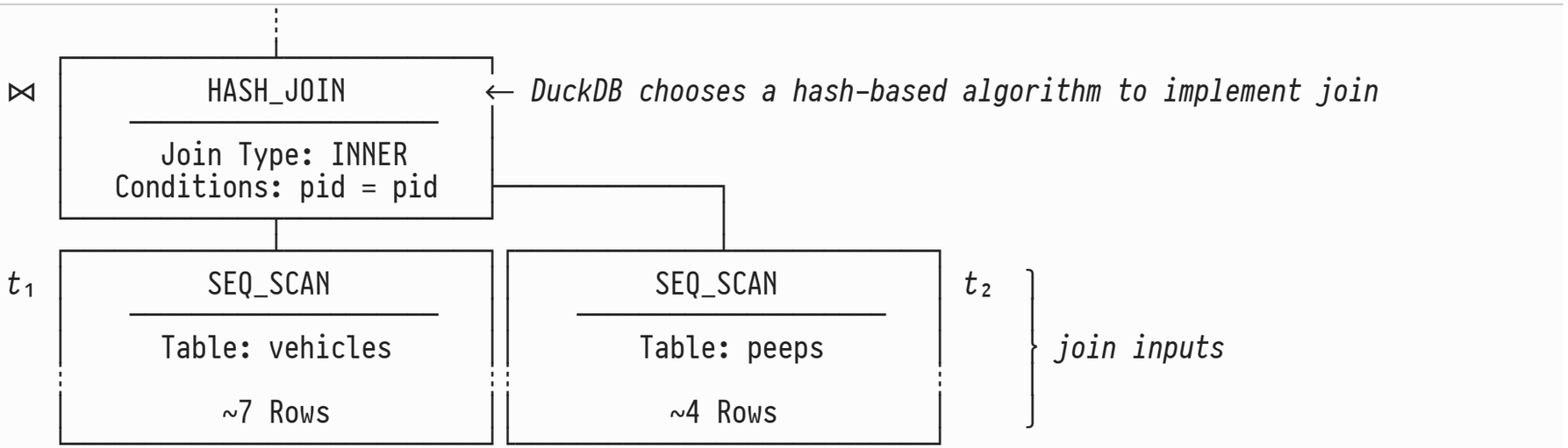
  - Semi join: Keep rows from L with *at least one join partner*.
  - Anti join: Keep rows from L that have *no join partner*.

  - Semi/anti join return a subset of rows from L: join predicate p aside, row variables bound to R are inaccessible in the SQL query (thus above: ⊸ instead of ⊣⊸). 01 #032

Joins in Query Plans

Joins—in textbooks often: $\bowtie$ (or $\rtimes$ $\ltimes$ $\overline{\rtimes}$ $\overline{\ltimes}$ ⟗ ⟕...)—operate on two tables. A join between n tables requires n-1 binary $\bowtie$ operations).

- **FROM** vehicles **NATURAL JOIN** peeps in EXPLAIN output (excerpt):



- Inner join is *commutative* ($t_1 \bowtie t_2 \equiv t_2 \bowtie t_1$) and *associative* ($(t_1 \bowtie t_2) \bowtie t_3 \equiv t_1 \bowtie (t_2 \bowtie t_3)$).
  Exploited by the DBMS's **query optimizer** to rearrange joins and inputs to find an efficient query plan alternative. #033

4 ¦ SQL: Bag Algebra

A table instance is a **bag** of rows (no order, duplicate rows may occur). SQL features the binary operations of the bag algebra:

[Likewise: **INTERSECT ALL** $\cap_+$, **EXCEPT ALL** $\setminus_+$]

bag of rows

$query_1$ **UNION ALL** $query_2$ -- $query_1 \uplus query_2$

two bags of rows

- Rows returned by $query_1$, $query_2$ (and the result) conform:[4]
  - Number of columns are equal.
  - Types in the same column position match (or are castable).
  - $query_1$ determines the column names of the result.

#034

[4] Alternatively, DuckDB can match columns in a bag union operation by name (not position): **UNION ALL BY NAME**. This is non-standard. See the DuckDB documentation on bag (and set) operations.

SQL's Bag Algebra Respects Row Multiplicities

Operations of the bag algebra respect *multiplicities*. So does SQL:

```
{🅐🅐🅐🅑🅒} ⊎  {🅐🅑🅑🅓} = {🅐🅐🅐🅐🅑🅑🅑🅒🅓}  (add)
{🅐🅐🅐🅑🅒} ⋂+ {🅐🅑🅑🅓} = {🅐🅑}              (minimum)
{🅐🅐🅐🅑🅒} \+  {🅐🅑🅑🅓} = {🅐🅐🅒}            (subtract, ⩾ 0)
```

🅐🅑🅒🅓 denote rows, {…} denotes a bag (table) of rows.

- To ignore multiplicities, omit the **ALL** modifier. Bag operations then discard row duplicates (δ({🅐🅐🅐🅑🅒}) ≝ {🅐🅑🅒}):

$$query_1 \ \textbf{\textit{OP}} \ query_2 \quad \equiv \quad \delta(\ \delta(query_1) \ \textbf{\textit{OP}}\ \textbf{ALL}\ \delta(query_2)\ )$$

- **Duplicate elimination** (δ) can be generally useful in queries. SQL thus supports the **DISTINCT** modifier in the **SELECT** clause:

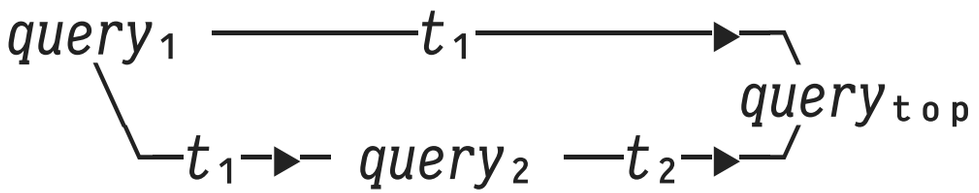
$$\begin{array}{l}\textbf{SELECT DISTINCT} \cdots \\ \textbf{FROM} \quad \cdots \\ \ \vdots\end{array} \quad \equiv \quad \delta\left(\begin{array}{l}\textbf{SELECT} \cdots \\ \textbf{FROM} \quad \cdots \\ \ \vdots\end{array}\right)$$

5 ¦ SQL: Assembling Queries From Pieces (Common Table Expressions)

If a complex SQL query is best understood in terms of *intermediate result tables*, name these intermediates and refer to them later:

```
WITH                          Common Table Expression (CTE)
t₁ AS (                         • locally define name tᵢ as result of queryᵢ
  query₁ ),
t₂(c₁,...,cₙ) AS (              • may name result columns of intermediates
  query₂ )
query_top                       • top-level query (= result of CTE)
```

- In such a CTE, the dependencies between queries and visibility (or: scopes) of the t_i form a DAG. No cycles:



- Decision: *materialize* t_i or *inline* $query_i$ at its use site(s)?

6 A *Star Wars* Database

kaggle is an extensive online repository of open data sources, many of those in CSV and/or Parquet formats.

- The "Star Wars Dataverse" collects bits of information on Disney's *Star Wars* franchise in 15 Parquet files.
- We created a native database file `'035-starwars.db'` from these Parquet sources and cleaned the data:
  - Normalized in-universe dates[5], map to type `int` (`'3 BBY'` → `-3`, `'2 ABY'` → `2`).
  - Turned comma-separated `text` values into `text[]` arrays (`'Luke, Leia'` → `['Luke', 'Leia']`).
  - Normalized film titles across tables. (`'Episode IV: A New Hope'` → `'A New Hope'`)

Load the *Star Wars* database to exercise your SQL skills. #035

[5] BBY/ABY: Years *Before/After the Battle of Yavin* in which the Rebel Alliance destroyed the Death Star.

DuckDB UI: Browsing a Database

Load the *Star Wars* database and use the browser-based **DuckDB UI**[6] to get familiar with the schemata/instances of the 15 tables:

1. Install and load the DuckDB UI extension (first use only):

```
D INSTALL motherduck;
D LOAD motherduck;
```

2. Load *Star Wars* database, invoke the UI (starts browser):

```
D ATTACH '035-starwars.db' AS starwars;
D CALL start_ui();
┌──────────────────────────────────────┐
│ UI started at http://localhost:4213/ │
└──────────────────────────────────────┘
```

[6] Once the UI extension is installed, may start DuckDB UI from the shell via command `duckdb -ui <database>.db`.

SQL Training (Star Wars Database)

Use SQL to formulate these queries against the *Star Wars* database. If needed, use CTEs (`WITH ...`) to cope with query complexity.

1. Which characters have got "*a bad feeling about this*" in what film(s)? [★]
2. Which films (title, year) make up the prequels (released after *Return of the Jedi*, before *The Force Awakens*)? [★★]
3. Which films (title) were created by the congenial duo of director George Lucas and composer John Williams? [★★]
4. Which droids (name) appear in *any* film directed by George Lucas? [★★]
5. Which droids (name) appear in *all* films directed by George Lucas? [★★★★]

Tabular Database Systems

⑦

More SQL (Subqueries + Embedded SQL)

March 17, 2026

Torsten Grust
Universität Tübingen, Germany

1 ¦ Compositionality (in Programming Languages)

“The meaning of a complex program is determined by the meanings of its constituent programs.”

—Principle of **Compositionality**

The following two (Python) programs are equivalent:

1. Uses literal 21:

```
print(2 * 21)
```

2. Computes value 21:[1]

```
def twenty_one():
  return (12 + 12*12 + 20) // (2*4) - 1
print(2 * twenty_one())
```

[1] Based on a variation of a popular limerick: “*A dozen, a gross, and a score / divided by two times the four / decreased just by one / it gives twenty-one / (which is three times seven, no more).*”

2 ┆ SQL: Subqueries (Queries Inside Queries)

SQL queries may contain nested **subqueries** enclosed in (⋯).

1. **Scalar subquery:** Where a query accepts a scalar x, x may be replaced by a subquery (q) that returns a single-cell table:

   #036

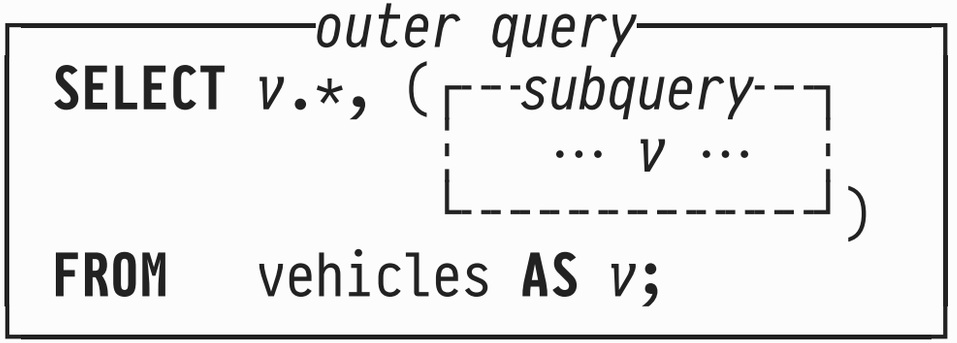
   q evaluates to $\boxed{\begin{array}{|c|}\hline\\\hline x\\\hline\end{array}} \equiv x$

   NB. A scalar subquery q needs to return…

   - … a **single-column** table (column name irrelevant). Otherwise: error at query *compile time*. 👎
   - … **at most one row.**
     0 rows ≡ NULL, ⩾ 2 rows: error at query *run time*. 👎 👎

Subqueries Can Relate to Outer Queries: Correlation

- Row variables bound inside a subquery do not “escape”: their scope is local to the subquery (see var v1 in #036).
- **But:** Subqueries may relate to row variables bound in the enclosing/outer query:



The subquery can access the current row bound to *v* ✓

  - The subquery in ⬚ *cannot* be evaluated in isolation: depends on outer query ☐ to provide a binding for row variable *v*.
  - DB jargon: “The subquery is **correlated** (since it uses *v*).”[2]

[2] PL jargon: “Variable *v occurs free* in the subquery (but bound in the outer query).”

Correlated Subqueries ≡ Nested Loops?

NB. Due to correlation, the subquery *q* in ⬚ below acts like a function of type int → text (*q*(pid) maps vehicle to driver name):

```
-- Query Q: Pair vehicles with their driver (if any)
                  ┌--------- q ----------┐
SELECT v.*, (     ¦ SELECT p.name        ¦
                  ¦ FROM   peeps AS p    ¦
                  ¦ WHERE  p.pid = v.pid ¦ ) AS driver
                  └----------------------┘
FROM   vehicles AS v;
```

- A "*nested loops*" evaluation strategy for *Q*:

```
for each v ∈ vehicles                                  ⎫
| driver ← q(v.pid)                                    ⎪  evaluated
⌊ result ← result ⊎ (v.*, driver)                      ⎭  |vehicles| times
```

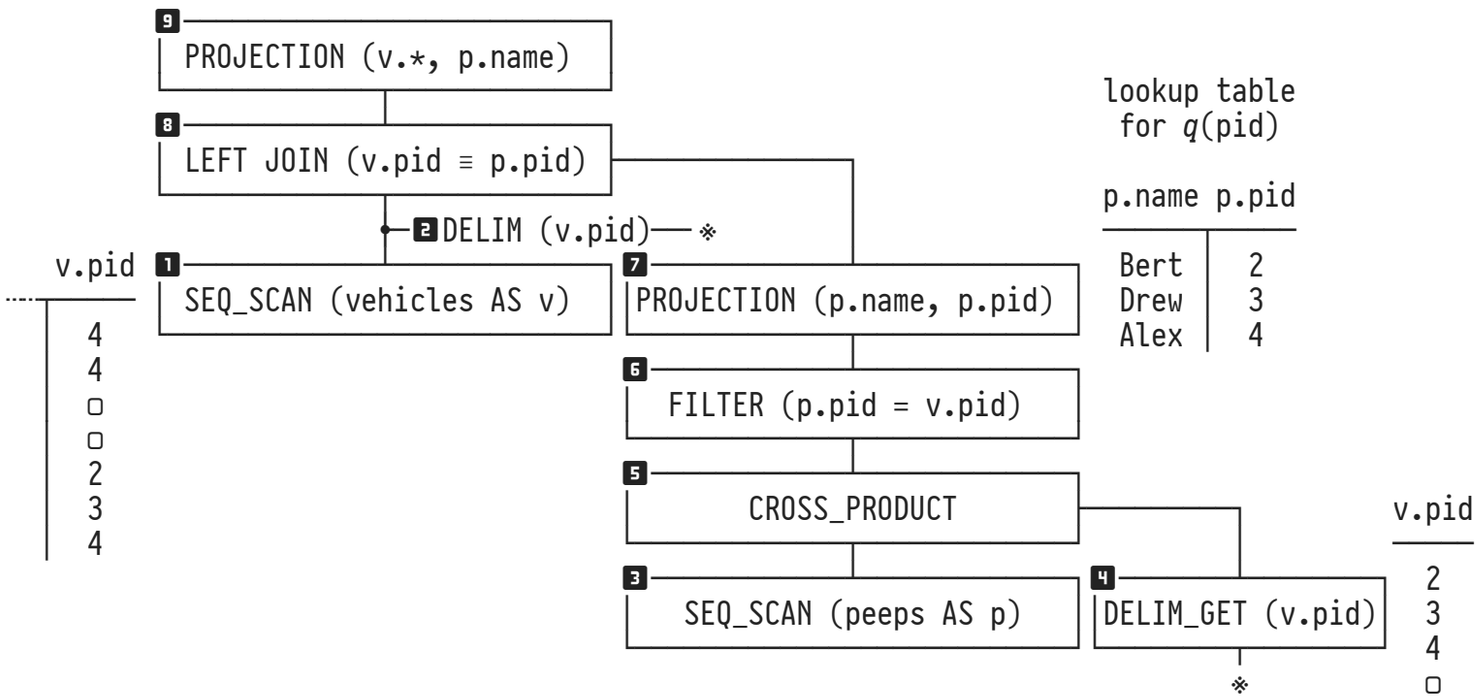
  - If vehicles.pid is not unique, this will evaluate subquery *q*(pid) repeatedly with identical arguments. 👎

DuckDB's Query Optimizer Removes Correlation

Query optimization **decorrelates** the subquery (plan for *Q*):[3]

1. 2,4: Compute the set of *distinct arguments* for subquery *q*.
2. 3–7: Build a lookup table for "function" *q*(pid).
3. 8: For each vehicle v, perform a lookup to evaluate *q*(v.pid).

#038

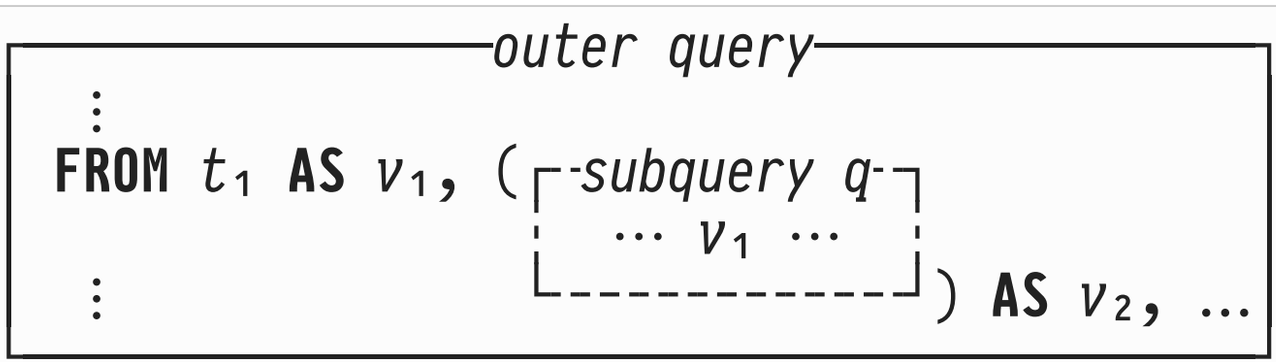


[3] Plan has been simplified. Use `PRAGMA explain_output = 'all';` to see the above Unoptimized Logical Plan. In these plans, the LEFT (OUTER) JOIN/DELIM pair is represented as DELIM_JOIN.

Scalar vs. Table-Valued Subqueries

SQL interprets subqueries (q) based on their *usage context*:

1. **Scalar:** q returns single-cell table ▢ that holds one scalar.
2. **Table-valued** (e.g., in **FROM** clause): q returns any table ▦:



#039

Subquery q in table-valued context

- Correlation: row variable v_1 may occur free[4] in subquery q. In this case, ⬚ acts like a table-valued function $q(v_1)$. (DBMS will decorrelate to avoid $|t_1|$ evaluations of q).
- If q is uncorrelated: ⬚ acts like a (computed) table.

[4] Some SQL implementations require the keyword **LATERAL** (“sideways”) to allow q to refer to v_1 (and thus depend on the evaluation of t_1): **FROM** t_1 AS v_1, **LATERAL** (q) AS v_2, …. (DuckDB infers whether **LATERAL** is required.)

3 ¦ SQL: Existential and Universal Quantification

SQL uses table-valued subqueries (*q*) to compactly formulate **existentially or universally quantified comparisons:**[5]

```
    EXISTS (q)  does q return ⩾ 1 rows  (is q non-empty)?
NOT EXISTS (q)  does q return no row at all (is q empty)?
```

```
expr = ANY (q)  does expr equal any value in ▨?
expr = ALL (q)  does expr equal all values in ▨?
```

also: <> < <= >= >

q evaluates to a single-column table:

[5] The SQL keywords **ANY** and **SOME** are synonyms. Syntactic sugar: *expr* = **ANY** (*q*) is equivalent to *expr* **IN** (*q*) (think of ∈ or "*is element* ***in***").

#040 + #041

4 Embedding SQL in Python Programs

- The DuckDB CLI enables interactive experimentation and the execution of ad-hoc/one-short querying. Definitely valuable.

- **Database-supported applications** embed SQL statements directly in the program source instead:
  - Programs can connect to/disconnect from selected databases.
  - Program flow controls which/how often SQL queries execute.
    - Queries may be constructed/parameterized on the fly.
  - Query results may be consumed by the program:
    - Map SQL data types to programming language's type system.
    - Receive *all* rows at once? *Iterate* over result row-by-row?

  Q: Which parts of the app logic are performed by the DBMS?
  Which parts are implemented by program code? ¯_(ツ)_/¯

Here ↔ **:** Use DuckDB's API to **embed SQL queries into Python.**

Embdedding SQL in Python: General Setup

Application code mixes SQL query strings 🛢 with program code 🐍:

#042

```
import duckdb   # requires: pip install duckdb
 ⋮
 # 1 connect to database (in 🔲 or on 💽)
 with duckdb.connect(database) as con:
    # 2 construct SQL query, submit to DuckDB
    rel = con.sql("""
      ┌-----------🛢   ┐  • embed SQL query as literal multi-line
      ¦  SQL query  ¦   }    string between """..."""
      └-------------┘   ┘  • sent to DuckDB (not executed yet)
      """)
    # 3 DuckDB executes query, retrieve all result rows
    result = rel.fetchall()
    # 4 iterate over list of rows
    for row in result:
      ┌-----------🐍   ┐  • Python code operates on a result row
      ¦    code     ¦   }    represented as a tuple (...,...,...)
      └-------------┘   ┘  • DuckDB not involved
```

DuckDB's Python DB API (Overview[6])

DuckDB Python API Call	Python Result
❶ `con = duckdb.connect(":memory:")`	DuckDB connection object
`con = duckdb.connect(`*`database file`*`)`	
❷ `rel = con.sql(`*`sql`*`)`	effect on DB or DuckDB relation object[7]
`rel = con.sql(`*`sql`*`, params = [...])`	*sql* may contain parameters `$1`, `$2`, ...
❸ `rel.fetchall()`	list of all result tuples in table `rel`
`rel.fetchmany(`*`n`*`)`	list of next *n* result tuples
`rel.fetchone()`	next result tuple or `None`
`rel.columns`	list of column names for table `rel`
`rel.types`	list of column types
`rel.show()`	`None` + printed table output (⊞)

[6] DuckDB's Python DB API (documentation)

[7] If *sql* is a **SELECT** query, returns a DuckDB relation object `rel`. Otherwise, applies the effect of the DDL or DML statement to the database represented by DuckDB connection object `con`.

How SQL Data Maps to Python Values/Objects

The type systems of SQL and Python—indeed most PLs—differ.[8] SQL data (tables, rows, cell values) needs to be **mapped** to Python values and objects:

a	b	c
a_1	b_1	c_1
a_2	b_2	c_2
a_3	b_3	c_3

SQL	Python
table	≡ list
row	≡ tuple
cell a_i	≡ val/obj A_1

```
[
  (A₁, B₁, C₁),
  (A₂, B₂, C₂),
  (A₃, B₃, C₃)
]
```

Data and Type Mapping in DuckDB's Python DB API

- DuckDB implements a best-effort mapping from cell values a_i (of types int, text, ...) to Python's value and objects A_i. #043

[8] This tension—not only regarding types—between SQL as a query language and programming languages (PLs) is known as the **impedance mismatch**.

Constructing SQL Queries at Program Run Time

Embedded SQL queries are regular strings: **programs can construct queries at run time** by interpolation 👍 or concatenation 👎.

1. **Interpolation** (Python values replace parameterized SQL values $1, $2, … in a template query):

   "**SELECT** $1 || p.name **FROM** peeps **AS** p **AS WHERE** p.born < $2"

   🐍 [1:"Driver: ", 2:2000]

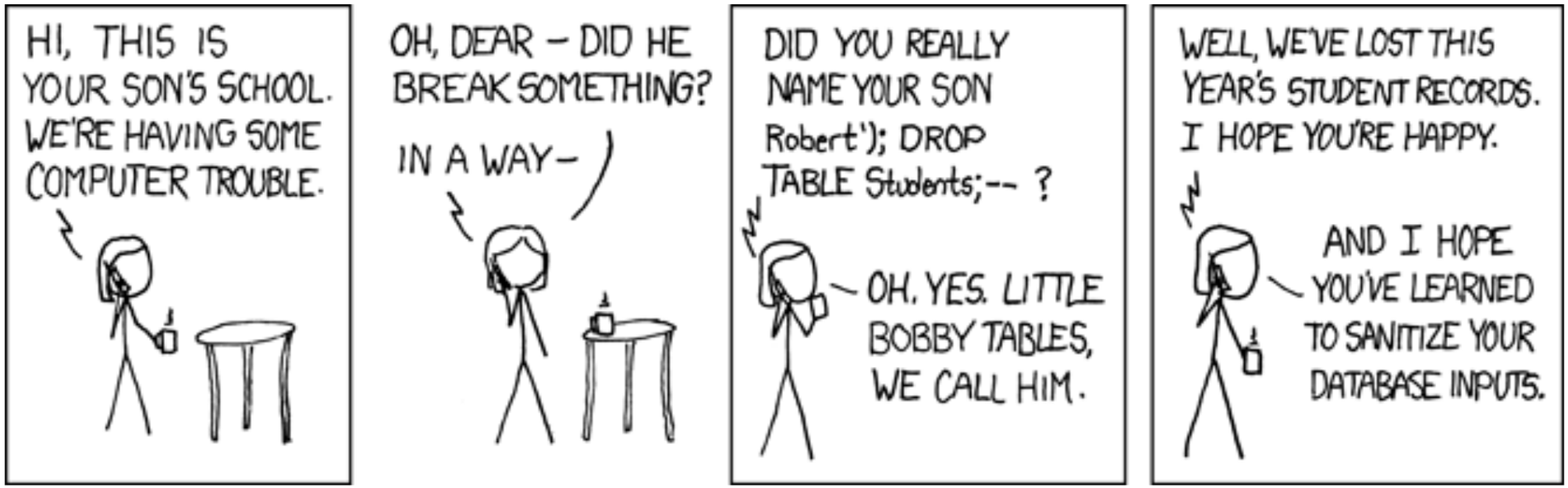
2. String concatenation (⚠️ Risk of SQL injection):

   "**FROM** peeps **AS** p **WHERE** p.name = '" + *N* + "' **AND** p.born < 2000"

   🐍 string concatentation

   If an attacker controls the value of Python string variable *N*, the DBMS may be tricked into executing arbitrary queries.

Parameterized Queries Protect Against Little Bobby's Mom



"Exploits of a Mom" , © xkcd

#044 + #045

Move Your Computation Close to the Data!

A rule of thumb 👍: *If you can, express data-related computation using SQL.*[9] *Do not demote the DBMS to a dumb table storage.*

- Filter/aggregate tables to **reduce result set sizes** before you transport data from the DBMS to the program.
- Common anti-pattern (the "***n+1 query problem***"):

```
outer = con.sql("SELECT ...")     # yields n rows
for row in outer.fetchall():
    inner = con.sql("SELECT ... $1 ...", params = [... row ...])
    ⋮
```

#046

  - The above issues *n+1 separate queries* all of which need to be interpolated, parsed, optimized, executed, and fetched. 👎
  - Reformulate using a join or a correlated subquery (which the query optimizer will decorrelate). Issues a *single query*. 👍

[9] Indeed, with SQL:1999 and the introduction of *recursive common table expressions*, the query language has become Turing-complete. We explore the consequences of this jump in expressivness in the course *Advanced SQL*.

Tabular Database Systems

⑧

SQL: Grouping + Aggregation and Functional Dependencies

April 7, 2026

Torsten Grust
Universität Tübingen, Germany

1 ¦ SQL: Grouping (Clause GROUP BY)

- Up to now, SQL queries operate *row-by-row*:
  - **FROM** *t* binds *individual rows* of *t* to row variables,
  - **WHERE** *p* drops row var bindings that do not satisfy *p*,
  - **SELECT** *e*,...,*e* evaluates *e*,...,*e* for each row var binding.

Clause **GROUP BY** e_1,...,e_n identifies **subtables (or: groups) of rows that yield the same value for expressions (criteria)** e_1,...,e_n:

vehicles

vehicle	kind	seats	wheels?	driver	groups
🏍	bike	1	true	p_2	
🚙	cabrio	2	true	p_4	true
🚜	SUV	3	true	p_4	
🚗	car	5	true	p_4	
🚌	bus	42	true	▢	false
🚐	bus	7	true	▢	
🚜	tank	▢	false	p_3	▢

```
SELECT   ···
FROM     vehicles AS v
GROUP BY v.seats < 5
         └──────────┘
       grouping criterion
```

∀ rows v in this group:
v.seats < 5 ≡ false

SQL: Grouping Requires Aggregation

💡 After grouping, use **aggregation** to represent group contents:

1 ⊞ **vehicles** (grouped)

vehicle	⋯	**seats**	⋯	**driver**
🏍		1		p_2
🚗	⋯	2	⋯	p_4
🚙		3		p_4
🚘		5		p_4
🚌	⋯	42	⋯	▢
🚐		7		▢
🚜	⋯	▢	⋯	p_3

2 ⊞ **vehicles** (grouped + aggregated)

seats<5	**count**(*)	**max**(**seats**)	**list**(**driver**)
true	3	3	$[p_2, p_4, p_4]$
false	3	42	$[p_4, ▢, ▢]$
▢	1	▢	$[p_3]$

- Tables are in 1NF. Cells cannot contain groups/subtables. Thus:
  - SQL never provides access to the grouped table **1** itself.
  - **Aggregation after grouping 2 is mandatory.**

01 #047

GROUP BY Reduces Granularity (Row-by-Row to Group-by-Group)

- The **FROM** 1 and **WHERE** 2 clauses operate row-by-row.
- After **GROUP BY** 3, the **SELECT** 4 clause operates group-by-group and yields **one row per group:**

```
SELECT   expr₁,...,exprₙ,
         agg,...,agg
FROM     t
WHERE    p
GROUP BY expr₁,...,exprₙ
```

```
3 } g
4 }
1 n (= |t|)
2 m (⩽ n)
3 g (⩽ m)
  # rows
```

#048

Non-aggregate $expr_i$ in **SELECT** is OK only if (the DBMS can statically deduce that) $expr_i$ is *constant within each group*.

- **Grouping Quiz:** What can you infer if you observe the following?
  Ⓐ $g = 1$. Ⓑ $g = m$. Ⓒ $g = 0$.
  Ⓓ Adding a new $expr_j$ to the **GROUP BY** clause does not change g.

Controlling Granularity: Derived Grouping Criteria

```
SELECT t.c,agg,...,agg                                  #049
FROM   t
GROUP BY t.c -- criterion c leads to too many groups... ¯\_(ツ)_/¯
```

1. **Division.** Group numeric c into buckets of **equal width** $1/N$:
   - If c is integral (int): **GROUP BY** c // N
   - Otherwise: **GROUP BY** round(c / N)
2. **Modulus.** Put integral c into one of N buckets (**order lost**):
   - **GROUP BY** c % N
   - If $N = 2^n$: **GROUP BY** c & $(1 << n) - 1$ (based on n low bits)
3. **Grading.**[1] Place ordered c into buckets with **given borders** b_i:

[1] *Grading* is derived from APL's ⍋ ("grade up"), list_grade_up() in DuckDB: list_grade_up([30,10,20]) = [2,3,1] are the list indices that would rearrange [30,10,20] in ascending order.

A Grouping Playground: The New York City Taxi Database

The *Yellow Cab* database[2] published by the NYC Taxi & Limousine Commission (TLC) provides a fun playground for **GROUP BY** queries.

A DuckDB database file holds the dataset for year 2024: #051

rides	main table, one row per taxi trip
zones	division of New York City into 265 zones (“boroughs”), rides start/stop here
central_park_weather	temperature/wind/precipitation data for Central Park, one row per day

Find plain + embedded SQL queries over this data in #050 #052

[2] Published monthly on the TLC Trip Record Data page, (formerly in CSV, since 2022 in Parquet format).

2 ¦ Group Preservation

```sql
-- How many peeps can each driver give a lift?
SELECT p.pid AS driver, ▨▨▨▨▨▨▨▨,
       max(v.seats) - 1 AS "can lift"
FROM   vehicles AS v NATURAL JOIN peeps AS p
GROUP BY p.pid, ▨▨▨▨▨▨▨▨;
```

#053

vehicles ⋈ peeps

vehicle	kind	seats	wheels?	pid	pic	name	born
	bike	1	true	2		Bert	1968
	tank	▢	false	3		Drew	▢
	car	5	true	4		Alex	2002
	SUV	3	true	4		Alex	2002
	cabrio	2	true	4		Alex	2002

- Adding p.name (pic, born) to GROUP BY ▨ **preserves groups:** rows that agree on pid, also agree on name: pid → name.

Functional Dependencies (FDs)

Given table $t(c_1,..,x,..,y,..,c_n)$, the **functional dependency (FD)** $x \to y$ ("*column x determines column y*") holds in t iff

$$\forall \text{ rows } r_1,r_2 \in t\text{: } r_1.x = r_2.x \implies r_1.y = r_2.y.$$

- **Notes on FDs:**
  - FD $x \to y$ indicates that t **embeds a lookup table** $t^f(\underline{x},y)$ defining a function $f(x) = y$.[3] Embedded table t^f has key x.
  - Generalization: $x_1\ x_2\ ..\ x_n \to y$ defines an n-ary function.
  - If $x \to y$ and $y \to z$, then $x \to z$ (transitivity, like $g \circ f$).
  - Shorthand notation: $x \to y$ and $x \to z \equiv x \to y\ z$.
  - A key 🔑 k for table $t(\underline{k},c_1,..,c_n)$ is like a special, powerful FD: $k \to c_1\ ..\ c_n$ (key k determines all columns).
  - Trivial FD (holds for any column x in any table t): $x \to x$.
  - $x \to y$ does *not* generally imply $y \to x$.

01 #053

[3] Example: For the FD pid → pic in vehicles ⋈ peeps, we have $f(2)$ = 😳, $f(3)$ = 😎, $f(4)$ = 🥵 (3×).

SQL: HAVING (and a Test for Functional Dependencies)

- SQL clause **HAVING** is evaluated *after* groups have been formed. Predicate q thus may use aggregates to **filter entire groups:**

```
SELECT   expr,...,expr,          ┐ 3 ┐
         agg,...,agg             │ 4 ┘ h
FROM     t                       │ 1   n (= |t|)
WHERE    p                       │ 2   m (⩽ n)
GROUP BY expr,...,expr           │ 3   g (⩽ m)
HAVING   q                       │ 4   h (⩽ g)
                                 └──   # rows
```

- A **HAVING**-based test: Does FD $x \to y$ hold in table t? #054

```
SELECT DISTINCT 'x ↛ y' AS "FD violated?"  ┐
FROM   t                                   │ empty result
GROUP BY x                                 │ indicates
HAVING count(DISTINCT y) > 1               ┘ that x → y does hold
```

3 ¦ Apps Define FDs …

trips

trip	weekday	driver	vehicle	from	to	dist	via	hop#
t_1	Mon	4	🚙	Alton	Corby	150	Alton	1
t_1	Mon	4	🚙	Alton	Corby	150	Luton	2
t_1	Mon	4	🚙	Alton	Corby	150	Corby	3
t_2	Tue	2	🏍	Derby	Eaton	17	Derby	1
t_2	Tue	2	🏍	Derby	Eaton	17	Eaton	2
t_3	Wed	2	🏍	Derby	Crich	23	Derby	1
t_3	Wed	2	🏍	Derby	Crich	23	Crich	2
t_4	Thu	4	🚙	Alton	Corby	150	Alton	1
t_4	Thu	4	🚙	Alton	Corby	150	Luton	2
t_4	Thu	4	🚙	Alton	Corby	150	Corby	3

Schedule of multi-hop road trips taken (with drivers and vehicles in use).

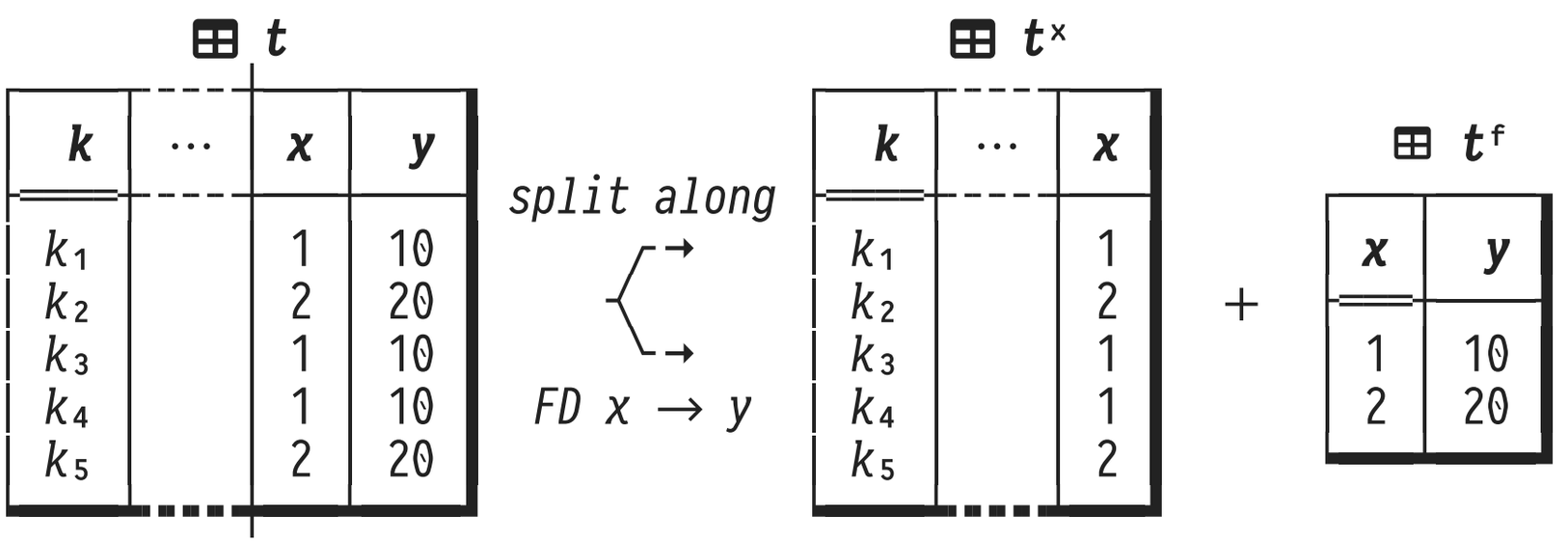
- Here are some FDs that hold in table trips. Verbalize what they mean in a DB application that manages road trips:

  Ⓐ vehicle → driver Ⓑ trip → weekday from to Ⓒ weekday → driver
  Ⓓ from to → dist driver Ⓔ from to hop# → via vehicle

… FDs Suggest Good Table Designs

1. The only FDs actually enforced by DBMSs are primary keys 🔑: recall the PRIMARY KEY constraint.
2. Non-🔑 FDs (embedded lookup tables) indicate data redundancy.

#055



- After the **split** ▦ ⟨ ▦+⊞ along FD $x \to y$:[4]
  - x is key in t^f: the DBMS **can now enforce the FD** $x \to y$. 👍
  - **No redundancy** in t^f. Function $f(x) = y$ is now explicit. 👍

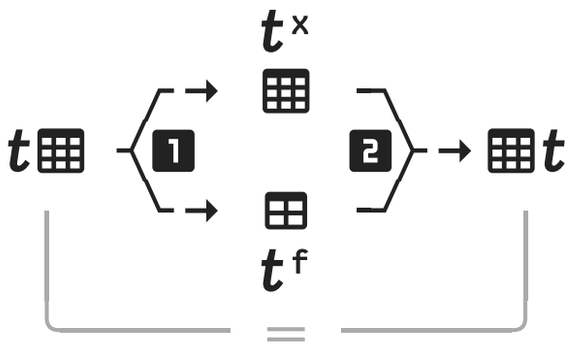
[4] Advise: When splitting along FD $x \to y$, move *all* columns functionally determined by x into t^f (including transitively determined columns). If t^f contains non-🔑 FDs, split t^f again.

Splits Can Be Undone (Virtually)

FD splits improve constraint checking and reduce redundancy. But users or apps may require/expect the original non-splitted table…

- An inner join ⋈ can **faithfully reconstruct** the original t.
  Table state is preserved (rows are neither lost nor added):[5]

FD split 1 2 Inner join ⋈ #056



- A **table view** can render this reconstruction of t transparent.

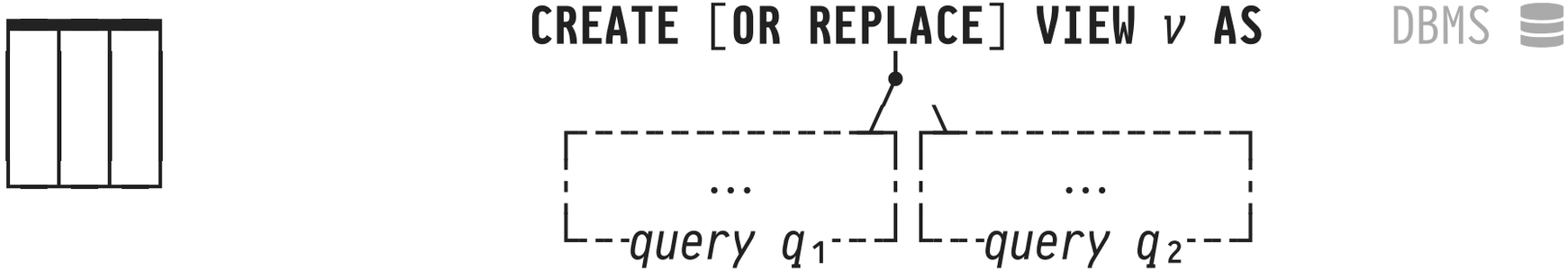
[5] FD splits are **lossless splits.** ⚠️ But: arbitrary vertical table splits may lose information (▦ ≠ 目+目+目).

4 ¦ Views (Logical Data Independence)

A SQL **view** v represents a *virtual* table whose schema and state is **defined by a query** q (instead of extensionally by a bag of rows).

- Views provide **logical data independence.** The actual data sources read by q remain hidden (q may change ↯ ↔ ↯):

Table ⊞ t **View** ⊞ v app



- Whenever v is referenced by a query, the DBMS re-evaluates q.
- Views are read-only and cannot be updated.[6]

[6] In general, it is ambigious how an “update” of v should affect the underlying source tables read by q.

The End.

Since you've got this far... A companion course on the design and implementation of selected DuckDB internals is available at https://github.com/DBatUTuebingen/DiDi.[7]

[7] DiDi: **D**esign and **I**mplementation of **D**uckDB **I**nternals 🩺🦆