Tabular Database Systems

④

Columnar Table Storage

Summer 2025

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 █
```

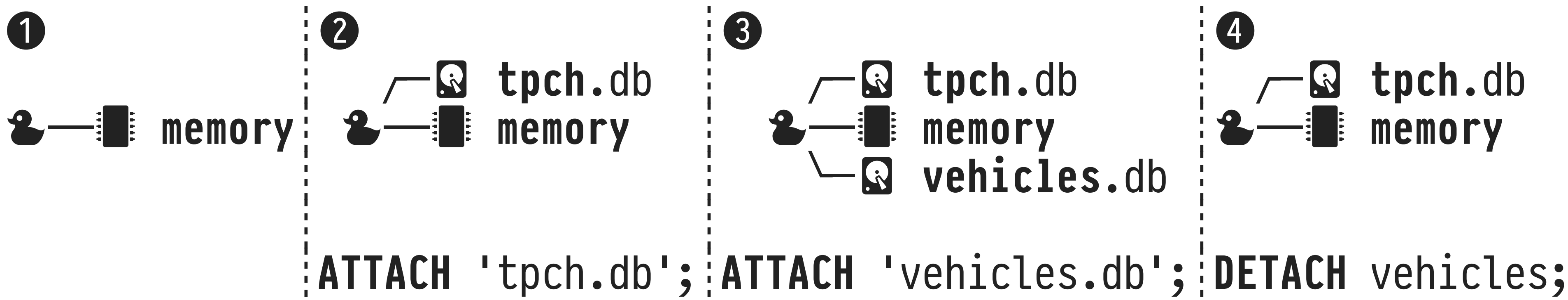
- **Persistent on-disk databases**

  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

$ duckdb persistent.db
(**ATTACH** 'persistent.db';)

memory | persistent.db

t
persistent.*t*

▷ **ATTACH** ':memory:';

memory.*t*

t
persistent.*t*

▷ **USE** memory;

t
memory.*t*

persistent.*t*

▷ **USE** persistent;

memory.*t*

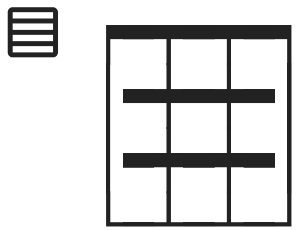
t
persistent.*t*

▷ **DETACH** memory;

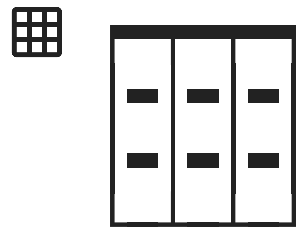
t
persistent.*t*

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**.

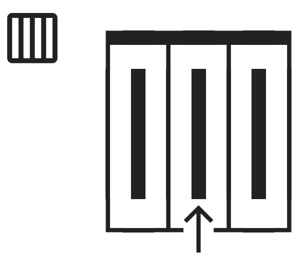


← row



← row split into cells
of a variety of types

- 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%).

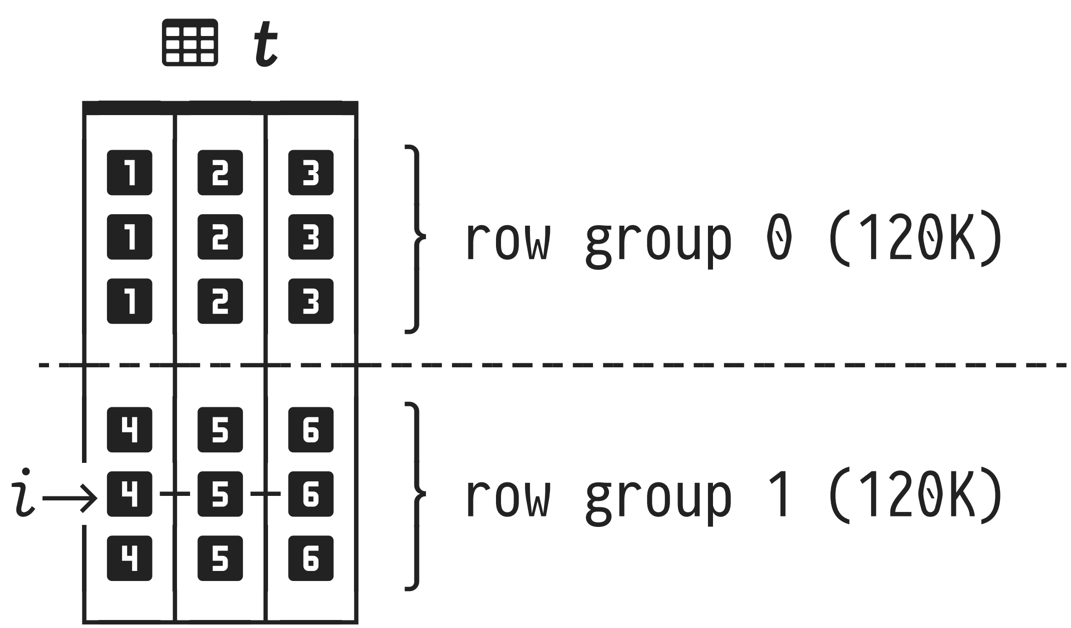
[2] **On**line **A**nalytical **P**rocessing, as opposed to OLTP (Online Transactional Processing) queries which touch few (even single) rows but read all those rows' columns.

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 ($120 \times 1024 = 122880$) rows $\Rightarrow$ t has $\lceil r \,/\, 120\text{K} \rceil$ row groups.
2. **Column chunks:** Inside each row group, store a chunk (of 120K cell values) for each of the c columns.

#022



1 … 6: column chunks
1 1 1: stored consecutively

Cell values at offset i
in chunks 4 – 5 – 6
form one row in table t

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)
$\underbrace{0000000000101010}_{42}$, $\underbrace{0000000000010011}_{19}$, $\underbrace{0000000000000100}_{4}$	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	$\underbrace{\text{1968-08-24}}_{\text{reference}} \oplus \underbrace{2\ 0\ 3}_{\Delta s}$

DuckDB: Lightweight Compression Schemes ③

- **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.