DB 2

04 – Row Internals

Summer 2020

Torsten Grust
Universität Tübingen, Germany

1 ┆ Q_3 — Projecting on Columns

SQL probe Q_3 projects on selected columns only (column b of table ternary is “projected away”):

```
SELECT t.a, t.c              -- access some columns of row t
FROM   ternary AS t
```

Retrieve all rows. Unpack/navigate the row and **extract selected columns.** Recall table ternary:

```
CREATE TABLE ternary (a int  NOT NULL,
                      b text NOT NULL, -- variable width
                      c float);        -- may be NULL
```

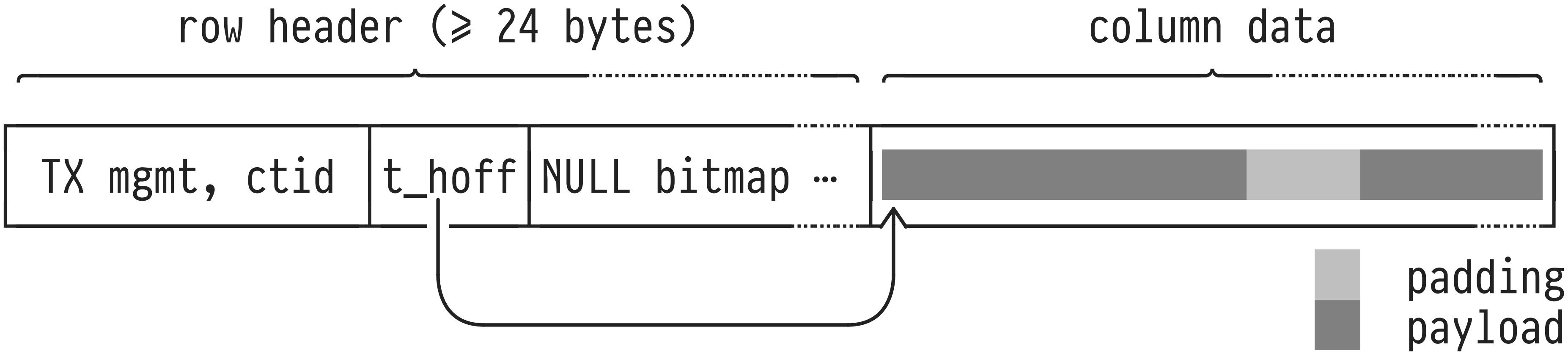
Using EXPLAIN on Q_3

```
EXPLAIN VERBOSE
  SELECT t.a, t.c
  FROM   ternary AS t;
```

QUERY PLAN
Seq Scan on public.ternary t (cost=0.00..20.00 rows=1000 width=12)   Output: a, c

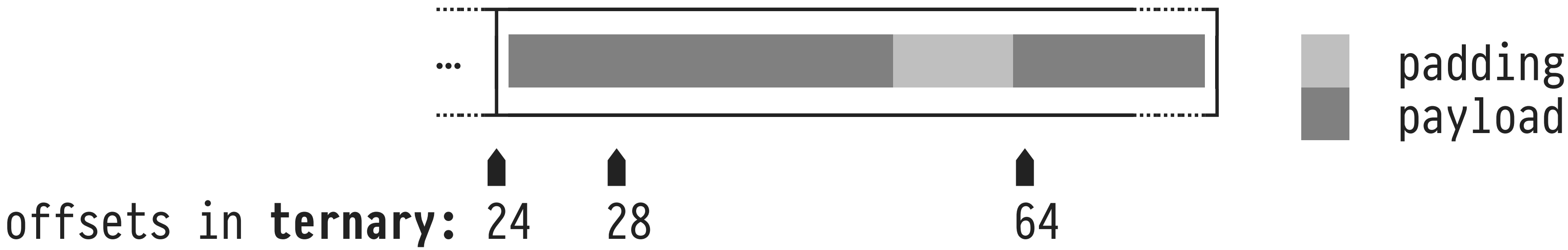
- For each row t, only columns a and c are extracted.
- **Seq Scan** emits narrower rows now, *average* width: 12 bytes = 4 (int) + 8 (float) bytes.
- Estimated cost of 20.00 unchanged from Q_2: Q_3 does *not* scan fewer data pages (→ row storage).

2 ¦ Internal Layout of a PostgreSQL Row



- NULL bitmap is of variable length (1 bit per column), offset t_hoff points to first byte of row payload data.
- **NB:** EXPLAIN's width=*w* reports payload bytes only.

Padding and Alignment



- CPU and memory subsystem require **alignment:** value of width n bytes is stored at address a with $a \bmod n = 0$.[1]
- ⇒ Pad payload such that each column starts at properly aligned offset (PostgreSQL: see table pg_attribute).

[1] Non-aligned data access incur performance penalties (multiple accesses) or even exceptions.

“Column Tetris”

Padding may lead to substantial space overhead. If viable, reorder columns to tightly pack rows and avoid padding:

```
CREATE TABLE padded (
  d int2,
  a int8,
  e int2,
  b int8,
  f int2,
  c int8)
```

48

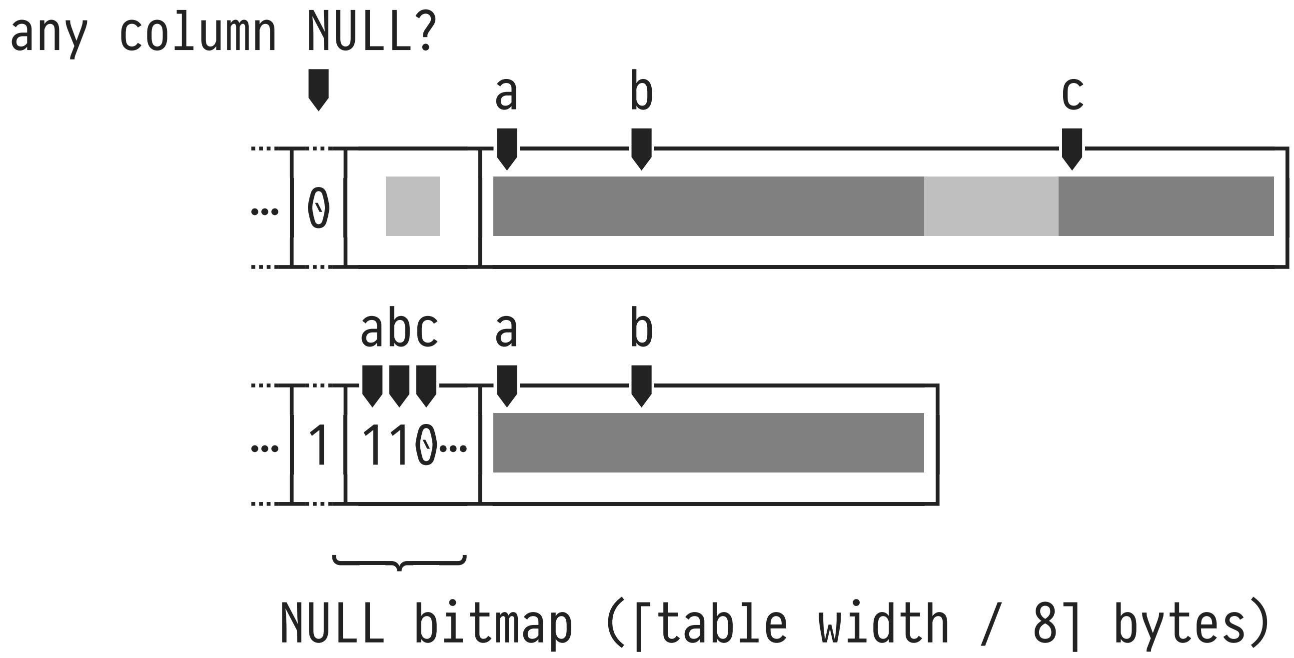
```
CREATE TABLE packed (
  a int8   -- int8: 8-byte aligned
  b int8
  c int8
  d int2   -- int2: 2-byte aligned
  e int2
  f int2)
```

30 (+2)   column data width

- +2: Rows start at MAXALIGN offsets (≡ 8 on 64-bit CPUs).

NULL (Non-)Storage

a	b	c
1	abc	0.1
⋮	⋮	⋮
2	def	NULL
⋮	⋮	⋮



- NULL values are represented by 0 bits in a **NULL bitmap** (bitmap is present only if the row indeed contains a NULL).

Column Access (Projection)

- If t denotes a row, **column access** — denoted using dot notation t.a — is the most common operation in SQL query expressions.
    - A typical SQL query will perform multiple column accesses per row (in SELECT, WHERE, GROUP BY, ... clauses), potentially millions of times during evaluation of a single query.

- Even tiny savings in processing effort (here: CPU time) will add up and can lead to substantial benefits.[2]

[2] This is a recurring theme in DBMS implementation. The larger the table cardinalities, the more worthwhile "micro optimizations" become.

Column Access (Projection)

- PostgreSQL: access ith column of a row using C routine `slot_getattr(`i`)`:
  1. Has value for column i been cached? If so, immediately return value.
  2. Check bit for ith column in NULL bitmap (if present): if `0`, immediately return `NULL`.
  3. Scan row payload data **from left to right** for all columns $k \leqslant i$:
     - Use type of column k to decode payload bytes.
     - Skip over contents if column k has variable width.
     - **Cache decoded value** for column k for subsequent `slot_getattr(`k`)` calls.

Column Access: PostgreSQL's slot_getattr()

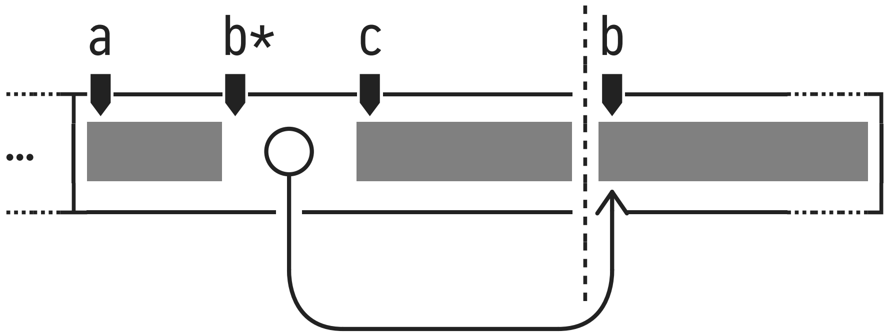
See PostgreSQL source code (a prime example of readable, consistent, well-documented C code—go read it!):

- File src/backend/access/common/heaptuple.c:

```
Datum
slot_getattr(TupleTableSlot *slot, int attnum, bool *isnull)
{
  /* step 1. check cache for column attnum (≡ i) */
  /* step 2. check NULL bitmap */
  ⋮
  /*
   * Extract the attribute, along with any preceding attributes.
   */
  slot_deform_tuple(slot, attnum);
  ⋮
}
```

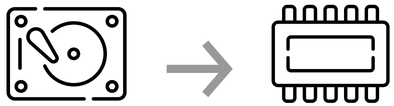
- slot_deform_tuple() does the hard decoding work (step 3.)

Alternative Layout of Row Payload: Fixed-Width First



- **Separate fixed- from variable-width** payload data at ┆:
  - ┆ : fixed-width columns a, c (types int, double) + fixed-width pointers b* to variable-width columns
  - ┆ : variable-width value for column b (type text)
- ⇒ Can calculate offsets of fixed-width columns **at query compile time**, no left-to-right scanning at run time.

3 ┆ Q_3 — Projecting on Columns



Column b of table ternary(a,b,c) is irrelevant for the projection query Q_3:

```
SELECT t.a, t.c          -- access some columns of row t
FROM   ternary AS t
```

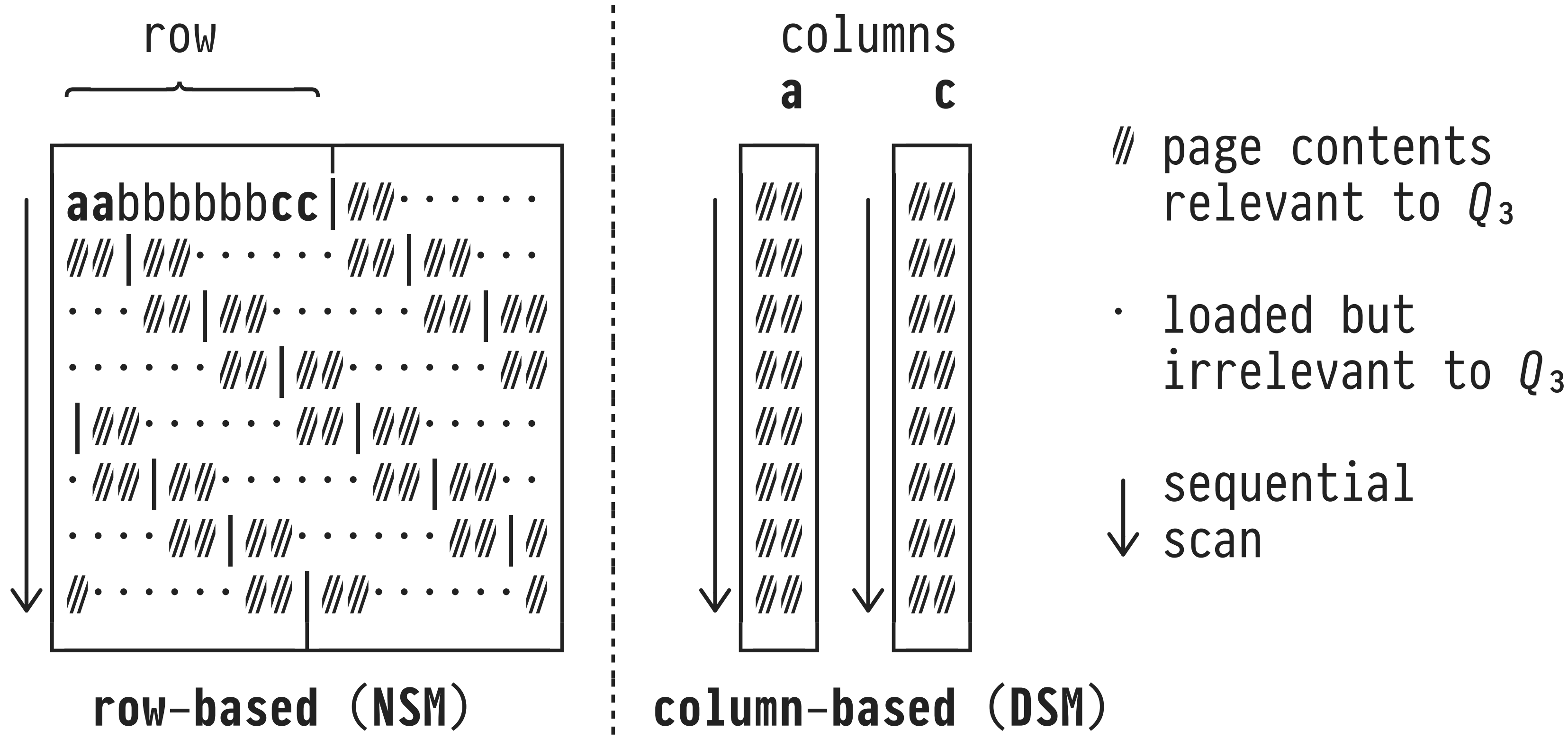
We expect the column-oriented DBMS to **exclusively touch the relevant columns.** The wider the input table (and the less columns are accessed), the higher the expected benefit over the row-based DBMS.

Using **EXPLAIN** on Q_3

MAL program for Q_3, shortened and formatted (compare with the MAL program for Q_2):

```
 ⋮
X_4              := sql.mvc();
C_5 :bat[:oid] := sql.tid(X_4, "sys", "ternary");
X_18:bat[:dbl] := sql.bind(X_4, "sys", "ternary", "c", …);
X_24:bat[dbl]  := algebra.projection(C_5, X_18);
X_8 :bat[:int] := sql.bind(X_4, "sys", "ternary", "a", …);
X_17:bat[:int] := algebra.projection(C_5, X_8);
 ⋮
 ‹create schema of result table›
 ⋮
sql.resultSet(…, X_17, X_24);
```

Don't Need it? Don't Load it!



- **100%** of the data loaded by the column-based DBMS is useful for query evaluation.