DB 2

04 — Row Internals

Summer 2018

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
```

Recreate and populate table ternary. Set some c values to NULL, though:

```
DROP TABLE IF EXISTS ternary;
CREATE TABLE ternary (a int NOT NULL, b text NOT NULL, c float);

INSERT INTO ternary(a, b, c)
  SELECT i,
         md5(i::text),
         CASE WHEN i % 10 = 0 THEN NULL ELSE log(i) END
  FROM   generate_series(1, 1000, 1) AS i;

-- Q3: Retrieve all rows (in arbirary oder) and selected columns
SELECT t.a, t.c
FROM   unary AS t;

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

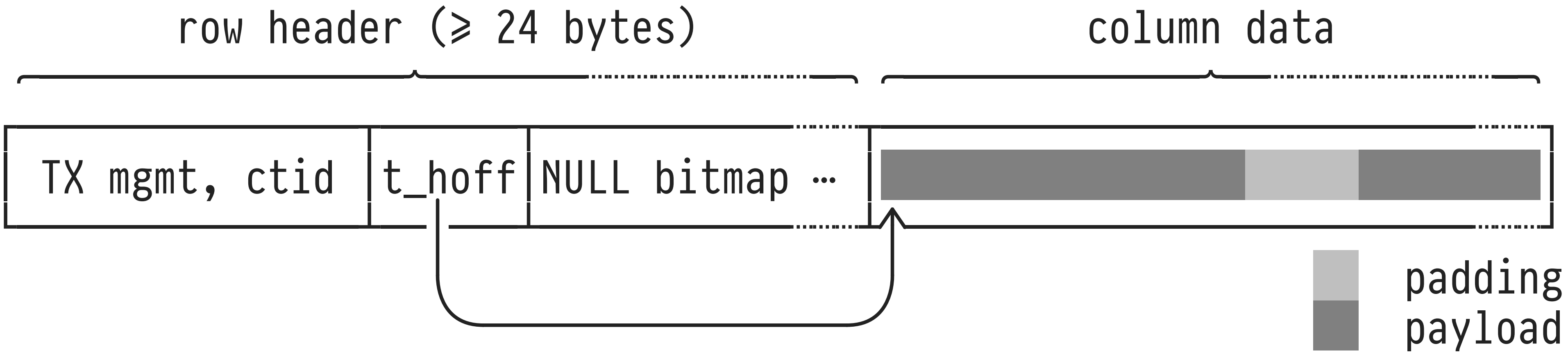
Using EXPLAIN on Q_3

```
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 ◀                                ▲                     ▲
```

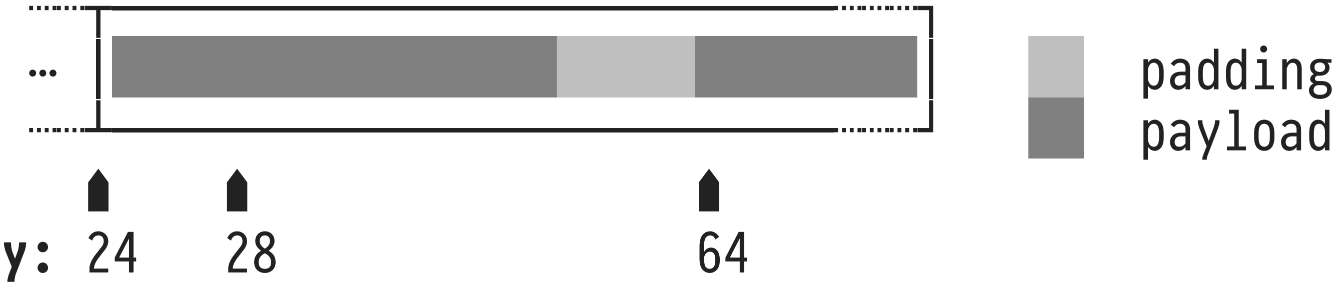
- For each row t, only columns a and c are extracted.
- **Seq Scan** emits narrower rows now, *average* width: 12 bytes = 4 (int) + 8 (float) bytes.
- Estimated cost of 20.00 unchanged from Q_2: Q_3 does *not* scan fewer data pages (→ row storage).

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* mod *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.

Check table pg_attribute for alignment requirements of the column type of table ternary:

```
SELECT a.attnum, a.attname, a.attlen, a.attstorage, a.attalign
FROM   pg_attribute AS a
WHERE  a.attrelid = 'ternary'::regclass AND a.attnum > 0
ORDER  BY a.attnum;
```

```
┌────────┬─────────┬────────┬────────────┬──────────┐
│ attnum │ attname │ attlen │ attstorage │ attalign │
├────────┼─────────┼────────┼────────────┼──────────┤
│      1 │ a       │      4 │ p          │ i        │
│      2 │ b       │     -1 │ x          │ i        │ ◀ align like an integer (word) ≡ 4 byte
│      3 │ c       │      8 │ p          │ d        │ ◀ align like a double-word     ≡ 8 byte
└────────┴─────────┴────────┴────────────┴──────────┘
                          ▲   ▲
column width (-1: variable)   p: stored plain
                              x: inline (may be compressed) or external (TOAST)
```

“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).

Demonstrate effect of “column tetris”:

```
DROP TABLE IF EXISTS padded;
DROP TABLE IF EXISTS packed;
CREATE TABLE padded (d int2, a int8, e int2, b int8, f int2, c int8);
CREATE TABLE packed (a int8, b int8, c int8, d int2, e int2, f int2);

INSERT INTO padded(d,a,e,b,f,c) SELECT 0,i,0,i,0,i FROM generate_series(1,1000000) AS i;
INSERT INTO packed(a,b,c,d,e,f) SELECT i,i,i,0,0,0 FROM generate_series(1,1000000) AS i;


SELECT COUNT(*) FROM pg_freespace('padded');
```

count
9346

```
SELECT COUNT(*) FROM pg_freespace('packed');
```

count
7353

⬅ **table** 'packed' uses fewer pages (79%)

```
SELECT lp, lp_off, lp_len, t_hoff, t_ctid FROM heap_page_items(get_raw_page('padded',0));
```

```
┌─────┬────────┬────────┬────────┬─────────┐
│ lp  │ lp_off │ lp_len │ t_hoff │ t_ctid  │
├─────┼────────┼────────┼────────┼─────────┤
│   1 │   8120 │     72 │     24 │ (0,1)   │
│   2 │   8048 │     72 │     24 │ (0,2)   │
[...]
│ 106 │    560 │     72 │     24 │ (0,106) │
│ 107 │    488 │     72 │     24 │ (0,107) │  ⬅ 107 rows per page, each row has 24 + 48 = 72 bytes
└─────┴────────┴────────┴────────┴─────────┘
```

```
SELECT lp, lp_off, lp_len, t_hoff, t_ctid FROM heap_page_items(get_raw_page('packed',0));
```

```
┌─────┬────────┬────────┬────────┬─────────┐
│ lp  │ lp_off │ lp_len │ t_hoff │ t_ctid  │
├─────┼────────┼────────┼────────┼─────────┤
│   1 │   8136 │     54 │     24 │ (0,1)   │
│   2 │   8080 │     54 │     24 │ (0,2)   │
[...]
│ 135 │    632 │     54 │     24 │ (0,135) │
│ 136 │    576 │     54 │     24 │ (0,136) │  ⬅ 136 rows per page, each row uses 24 + 30 = 54 bytes
└─────┴────────┴────────┴────────┴─────────┘     (MAXALIGN: a row starts every 56 bytes on the page)
```

```
EXPLAIN VERBOSE SELECT p.* FROM padded AS p;
```

```
                            QUERY PLAN
 Seq Scan on public.padded p  (cost=0.00..19346.00 rows=1000000 width=30)
   Output: d, a, e, b, f, c
```

lie! diregards padding

```
EXPLAIN VERBOSE SELECT p.* FROM packed AS p;
```

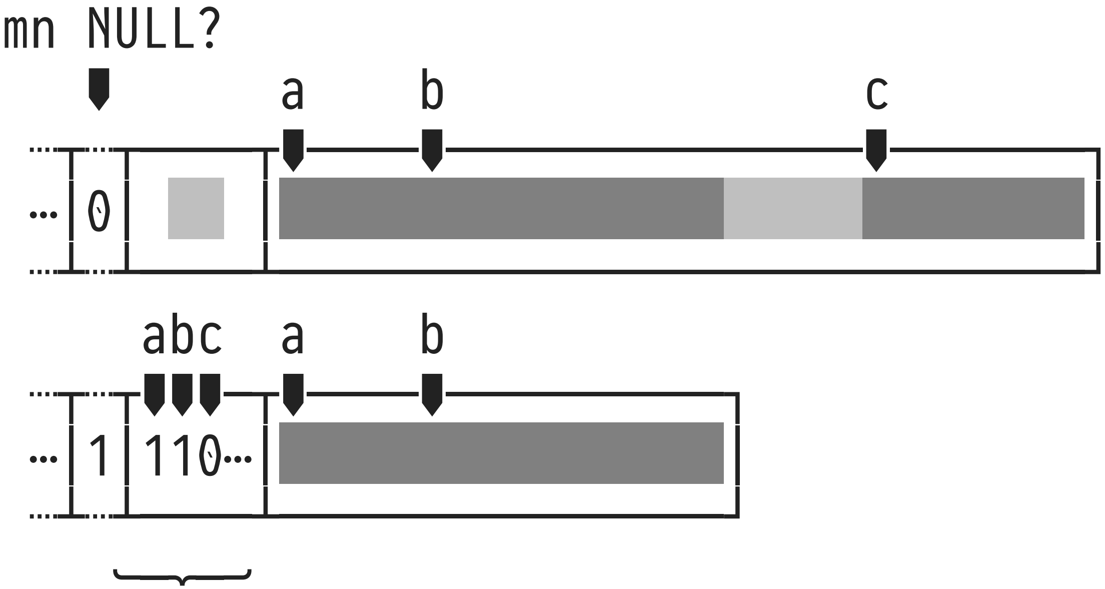
```
                            QUERY PLAN
 Seq Scan on public.packed p  (cost=0.00..17353.00 rows=1000000 width=30)
   Output: a, b, c, d, e, f
```

lie! diregards padding

DBMS expects to **perform less work on table** `'packed'`

NULL (Non-)Storage

a	b	c
1	abc	0.1
2	def	NULL



NULL bitmap (⌈table width / 8⌉ bytes)

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

Use extension pageinspect to check NULL-related row details:

```
SELECT lp, lp_off, lp_len, t_hoff, t_ctid, t_infomask::bit(1) AS "any NULL?", t_bits
FROM heap_page_items(get_raw_page('ternary',0));
```

```
+-----+--------+--------+--------+---------+-----------+----------+
| lp  | lp_off | lp_len | t_hoff | t_ctid  | any NULL? | t_bits   |
|-----+--------+--------+--------+---------+-----------+----------|
|    1|    8120|      72|      24| (0,1)   | 0         | □        |
|    2|    8048|      72|      24| (0,2)   | 0         | □        |
|    3|    7976|      72|      24| (0,3)   | 0         | □        |
|    4|    7904|      72|      24| (0,4)   | 0         | □        |
|    5|    7832|      72|      24| (0,5)   | 0         | □        |
|    6|    7760|      72|      24| (0,6)   | 0         | □        |
|    7|    7688|      72|      24| (0,7)   | 0         | □        |
|    8|    7616|      72|      24| (0,8)   | 0         | □        |
|    9|    7544|      72|      24| (0,9)   | 0         | □        |
|   10|    7480|      61|      24| (0,10)  | 1         | 11000000 |
[...]                 ▲        ▲                         ▲▲▲
                      |  column a                        abc (a,b: non-NULL, c: NULL)
                      |  starts at offset 24 in row
                      |   (NULL bitmap = MAXALIGN padding = 1 byte)
                      |
                      61 = 24 + 4 + 33 (no intra-row padding)
                            ▲   ▲    ▲
                       header   a    b
                               int  text (md5 hash is 32 chars long, 1 byte: string length)
```

Use table padded to insert and then inspect an all-NULL row (what will length/NULL bitmap look like)?

```
INSERT INTO padded(a,b,c,d,e,f) VALUES (NULL, NULL, NULL, NULL, NULL, NULL);

SELECT p.ctid FROM padded AS p WHERE (a,b,c,d,e,f) IS NULL; -- ≡ a IS NULL AND b IS NULL AND ...
```

ctid
(9345,86)

```
SELECT lp, lp_off, lp_len, t_hoff, t_ctid, t_infomask::bit(1) AS "any NULL?", t_bits
FROM   heap_page_items(get_raw_page('padded',9345))
WHERE  lp = 86;
```

lp	lp_off	lp_len	t_hoff	t_ctid	any NULL?	t_bits
86	2048	24	24	(9345,86)	1	00000000

```
                       ▲                            ▲▲▲▲▲▲
    only row header + NULL bitmap                   abcdef
```

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.)

PostgreSQL source, file src/backend/access/common/heaptuple.c:

```
/*
 * slot_getattr
 *    This function fetches an attribute of the slot's current tuple.
 *    It is functionally equivalent to heap_getattr, but fetches of
 *    multiple attributes of the same tuple will be optimized better,
 *    because we avoid O(N^2) behavior from multiple calls of
 *    nocachegetattr(), even when attcacheoff isn't usable.
 *
 *    A difference from raw heap_getattr is that attnums beyond the
 *    slot's tupdesc's last attribute will be considered NULL even
 *    when the physical tuple is longer than the tupdesc.
 */
Datum
slot_getattr(TupleTableSlot *slot, int attnum, bool *isnull)
{
  HeapTuple tuple = slot->tts_tuple;
  TupleDesc tupleDesc = slot->tts_tupleDescriptor;
  HeapTupleHeader tup;

  /*
   * system attributes are handled by heap_getsysattr
   */
  if (attnum <= 0)
  {
    if (tuple == NULL)    /* internal error */
      elog(ERROR, "cannot extract system attribute from virtual tuple");
    if (tuple == &(slot->tts_minhdr)) /* internal error */
      elog(ERROR, "cannot extract system attribute from minimal tuple");
    return heap_getsysattr(tuple, attnum, tupleDesc, isnull);
  }

  /*
   * fast path if desired attribute already cached
   */
  if (attnum <= slot->tts_nvalid)
  {
    *isnull = slot->tts_isnull[attnum - 1];
    return slot->tts_values[attnum - 1];
  }

  /*
   * return NULL if attnum is out of range according to the tupdesc
   */
  if (attnum > tupleDesc->natts)
  {
    *isnull = true;
    return (Datum) 0;
  }

  /*
   * otherwise we had better have a physical tuple (tts_nvalid should equal
   * natts in all virtual-tuple cases)
   */
  if (tuple == NULL)      /* internal error */
    elog(ERROR, "cannot extract attribute from empty tuple slot");

  /*
   * return NULL if attnum is out of range according to the tuple
   *
   * (We have to check this separately because of various inheritance and
   * table-alteration scenarios: the tuple could be either longer or shorter
   * than the tupdesc.)
   */
  tup = tuple->t_data;
  if (attnum > HeapTupleHeaderGetNatts(tup))
  {
    *isnull = true;
    return (Datum) 0;
  }

  /*
   * check if target attribute is null: no point in groveling through tuple
   */
  if (HeapTupleHasNulls(tuple) && att_isnull(attnum - 1, tup->t_bits))
  {
    *isnull = true;
    return (Datum) 0;
  }

  /*
   * If the attribute's column has been dropped, we force a NULL result.
   * This case should not happen in normal use, but it could happen if we
   * are executing a plan cached before the column was dropped.
   */
  if (TupleDescAttr(tupleDesc, attnum - 1)->attisdropped)
  {
    *isnull = true;
    return (Datum) 0;
  }

  /*
   * Extract the attribute, along with any preceding attributes.
   */
  slot_deform_tuple(slot, attnum);

  /*
   * The result is acquired from tts_values array.
   */
  *isnull = slot->tts_isnull[attnum - 1];
  return slot->tts_values[attnum - 1];
}
```

Experiment: What's the contribution of slot_deform_tuple() to the overall PostgreSQL runtime, when the DBMS evaluates a query?

Prepare a large variant of the ternary table:

```
DROP TABLE IF EXISTS ternary_10M;
CREATE TABLE ternary_10M(a int NOT NULL, b text NOT NULL, c float);

INSERT INTO ternary_10M(a, b, c)
  SELECT i,
         md5(i::text),
         CASE WHEN i % 10 = 0 THEN NULL ELSE log(i) END
  FROM   generate_series(1, 10000000, 1) AS i;
```

Use Activity Monitor to find out the PID of the postgres process that is used to perform work when a query is submitted (query takes about 3 seconds):

```
EXPLAIN (VERBOSE, ANALYZE) SELECT t.b, t.c FROM ternary_10M AS t;
```

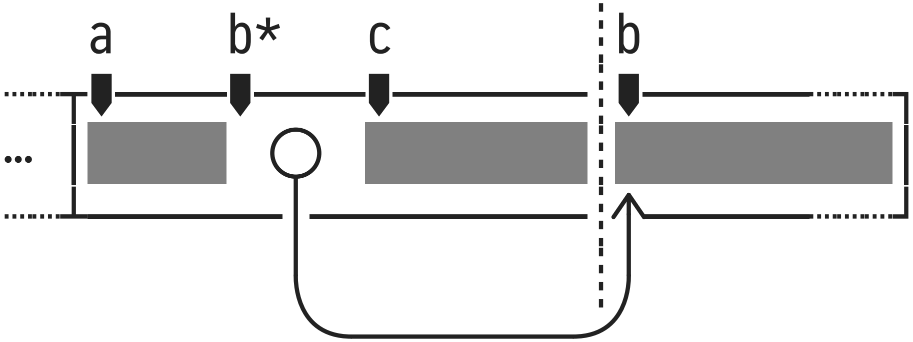
Use macOS' sample utility to capture the call stack of postgres during a 5 second interval while the above query is executing:

```
(in other terminal) $ sample postgres 5
You have access to multiple processes named postgres:
    a) 56806 /Applications/Postgres.app/Contents/Versions/10/bin/postgres
    [...]
    h) 57167 /Applications/Postgres.app/Contents/Versions/10/bin/postgres
Which process? (letter or PID) <h>

(in psql -d db2)
 EXPLAIN (VERBOSE, ANALYZE) SELECT t.b, t.c FROM ternary_10M AS t;
```

On other terminal/shell find output like this:

```
Sort by top of stack, same collapsed (when >= 5):
        poll  (in libsystem_kernel.dylib)        5894
        slot_deform_tuple  (in postgres)        394         ◀━ ⚠ top contributor
        __commpage_gettimeofday  (in libsystem_kernel.dylib)        370
        mach_absolute_time  (in libsystem_kernel.dylib)        350
        read  (in libsystem_kernel.dylib)        232
        ExecInterpExpr  (in postgres)        156
        heapgettup_pagemode  (in postgres)        118
        slot_getsomeattrs  (in postgres)        83
        HeapTupleSatisfiesMVCC  (in postgres)        73
        ExecProject  (in postgres)        68
[...]
```

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.

Representation of NULL?

- Presence of NULL left of ┊ may spoil static precalculation of column offsets. Option: represent NULL using special bit pattern that is as wide as a regular value (if column access performance more important than space savings).

Row navigation and column access is costly. Experiment:

```
DROP TABLE IF EXISTS ternary_10M;
CREATE TABLE ternary_10M (a int NOT NULL, b text NOT NULL, c float);

INSERT INTO ternary_10M(a, b, c)
  SELECT i,
         md5(i::text),
         CASE WHEN i % 10 = 0 THEN NULL ELSE log(i) END
  FROM   generate_series(1, 10000000, 1) AS i;
```

Perform two queries that retrieve and deliver the same column contents:

- Query ❶ accesses columns c, b, a individually.
- Query ❷ reproduces rows as is (fast-path, no actual access to individual columns).

```
-- ❶
EXPLAIN (VERBOSE, ANALYZE)
  SELECT t.c, t.b, t.a
  FROM ternary_10M AS t;
```

```
                                             QUERY PLAN
Seq Scan on public.ternary_10m t  (cost=0.00..192593.44 rows=10000044 width=45) (actual time=0.030..2306.229 rows=10000000 loops=1)
  Output: b, c, a                                                                                         ▲
Planning time: 0.033 ms
Execution time: 2827.920 ms ◀
```

```
-- ❷
EXPLAIN (VERBOSE, ANALYZE)
  SELECT t.*                -- also OK: t.a, t.b, t.c ≡ t.*
  FROM ternary_10M AS t;
```

```
                                             QUERY PLAN
Seq Scan on public.ternary_10m t  (cost=0.00..192593.44 rows=10000044 width=45) (actual time=0.066..1328.628 rows=10000000 loops=1)
  Output: a, b, c                                                                                         ▲
Planning time: 0.037 ms
Execution time: 1839.703 ms ◀
```

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.

Create and populate table ternary on MonetDB:

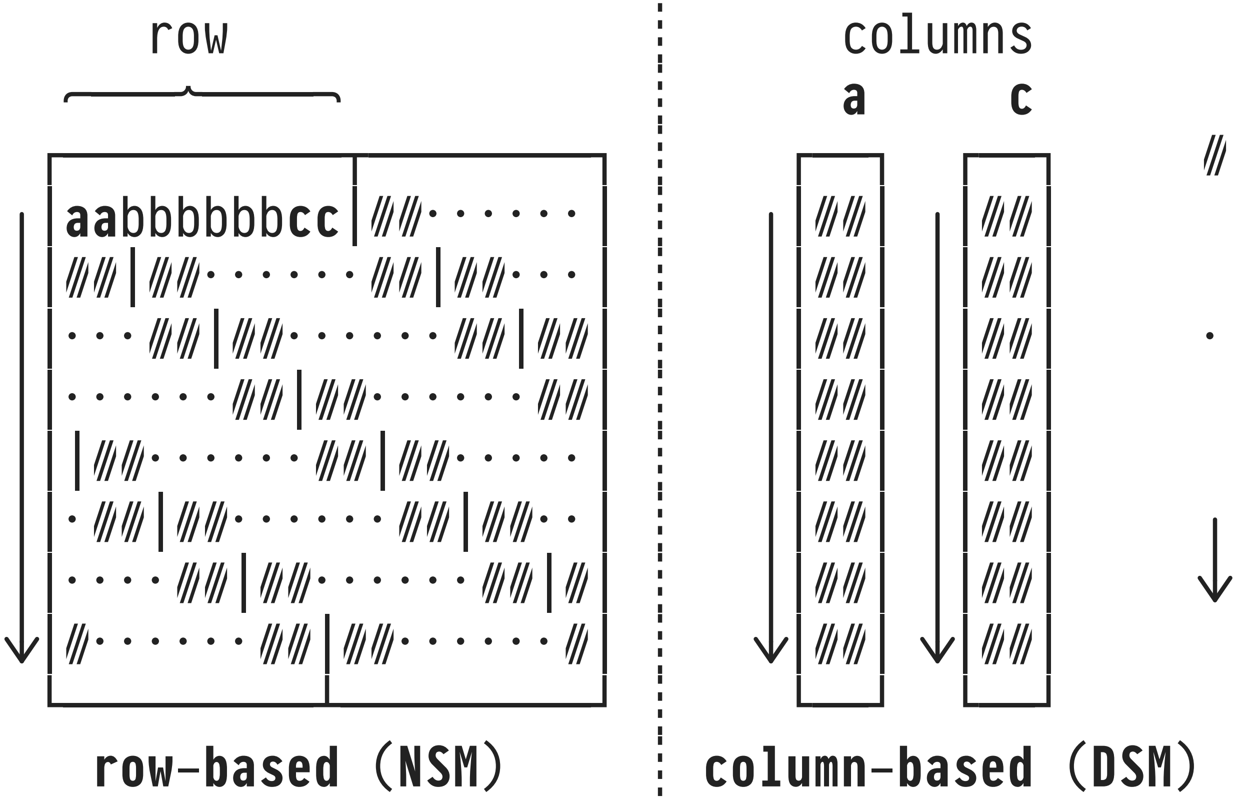
```
$ mclient -d scratch
Welcome to mclient, the MonetDB/SQL interactive terminal (Jul2017-SP4)
Database: MonetDB v11.27.13 (Jul2017-SP4), 'scratch'
Type \q to quit, \? for a list of available commands
auto commit mode: on

 DROP TABLE IF EXISTS ternary;
 CREATE TABLE ternary (a int NOT NULL, b text NOT NULL, c double);

 INSERT INTO ternary(a, b, c)
   SELECT value, md5(value), log(value)
   FROM   generate_series(1, 1001);
```

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!



⫽ page contents relevant to Q_3

· loaded but irrelevant to Q_3

↓ sequential scan

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