EDB POSTGRES AI

Values in PostgreSQL

Internal data value representation in PostgreSQL

Amul Sul

A Little About Me

Amul Sul

Database Developer

EnterpriseDB

Pune, India

Office Location

13 Years

Expertise in PostgreSQL internal development

EDB POSTGRES AI

Agenda

1. **Values & Data Types**
   Type system, typlen, varlena vs. cstring.

2. **Values in Memory**
   Datum container, conversion macros, pass-by-{value,ref}.

3. **Values on Disk**
   HeapTuple layout, padding, heap_form_tuple().

4. **Variable Length**
   varlena headers, TOAST out-of-line storage.

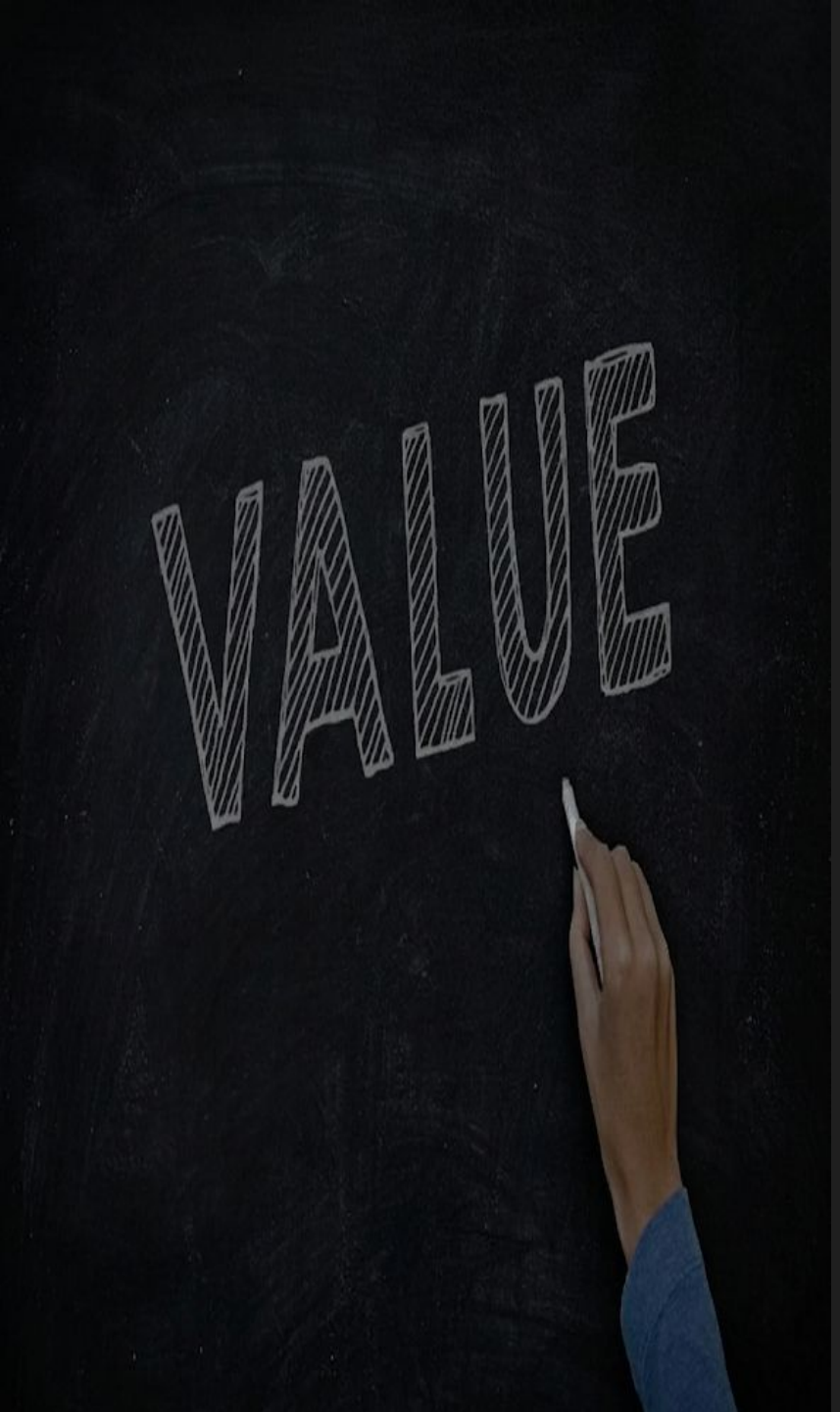
5. **Alignment & Padding**
   typalign, MAXALIGN, column ordering.

EDB POSTGRES AI

PART 01

Values and Data Types

What a value really is, and the type that gives it shape.



What is a value?

INSIDE postgres, A LITERAL CAN BE…

4	4.0	‘four’
an integer?	*a number?*	*a string?*

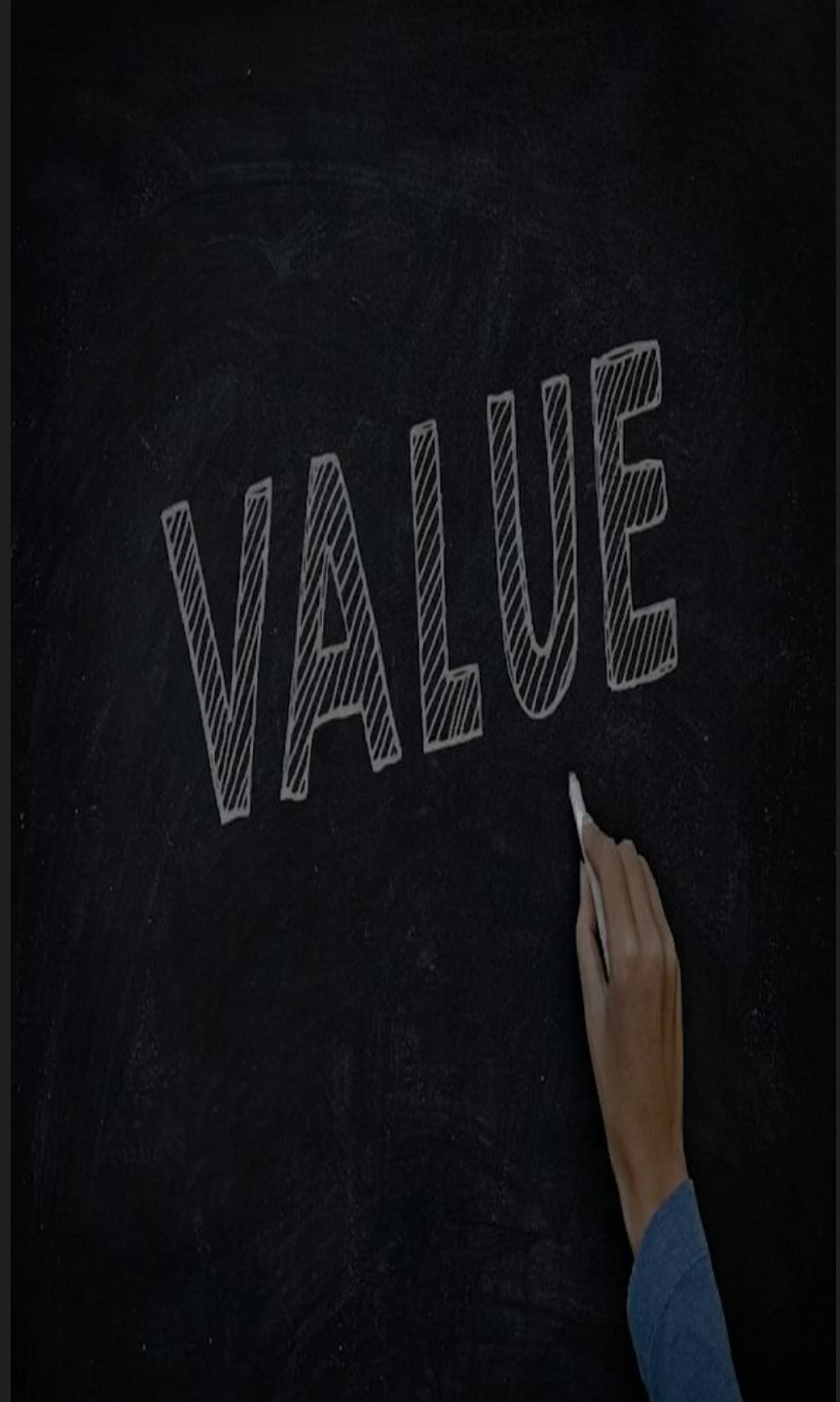
> *Without a data type, a literal has no meaning to the executor — same bytes can mean very different things.*

What is a Value? (More Accurate Version)

WITH AN EXPLICIT CAST, AMBIGUITY DISAPPEARS

4	4.0	‘four’
::pg_catalog.int4 – 4B	::pg_catalog.numeric – varlena	::pg_catalog.text – varlena
::pg_catalog.int8 – 8B	::pg_catalog.float8 – 8B	::pg_catalog.varchar(N) – varlena
::pg_catalog.int2 – 2B		

The data type — encoded in pg_type — is what tells PostgreSQL how the bytes are laid out, compared, and processed.



Fixed vs. Variable Length

TWO STORAGE FAMILIES

Fixed-length

```
typlen: 1 .. 16
```

EXAMPLES

`int2` · `int4` · `int8` · `bool` · `float4` · `float8` · `oid` · `timestamptz`

Always exactly N bytes. typlen > 0 in pg_type. Trivial to copy, compare, and place at a known offset.

Variable-length

```
typlen: -1 (varlena) | -2 (cstring)
```

EXAMPLES

`text` · `bytea` · `varchar` · `jsonb` · `numeric` · `arrays` · `range types`

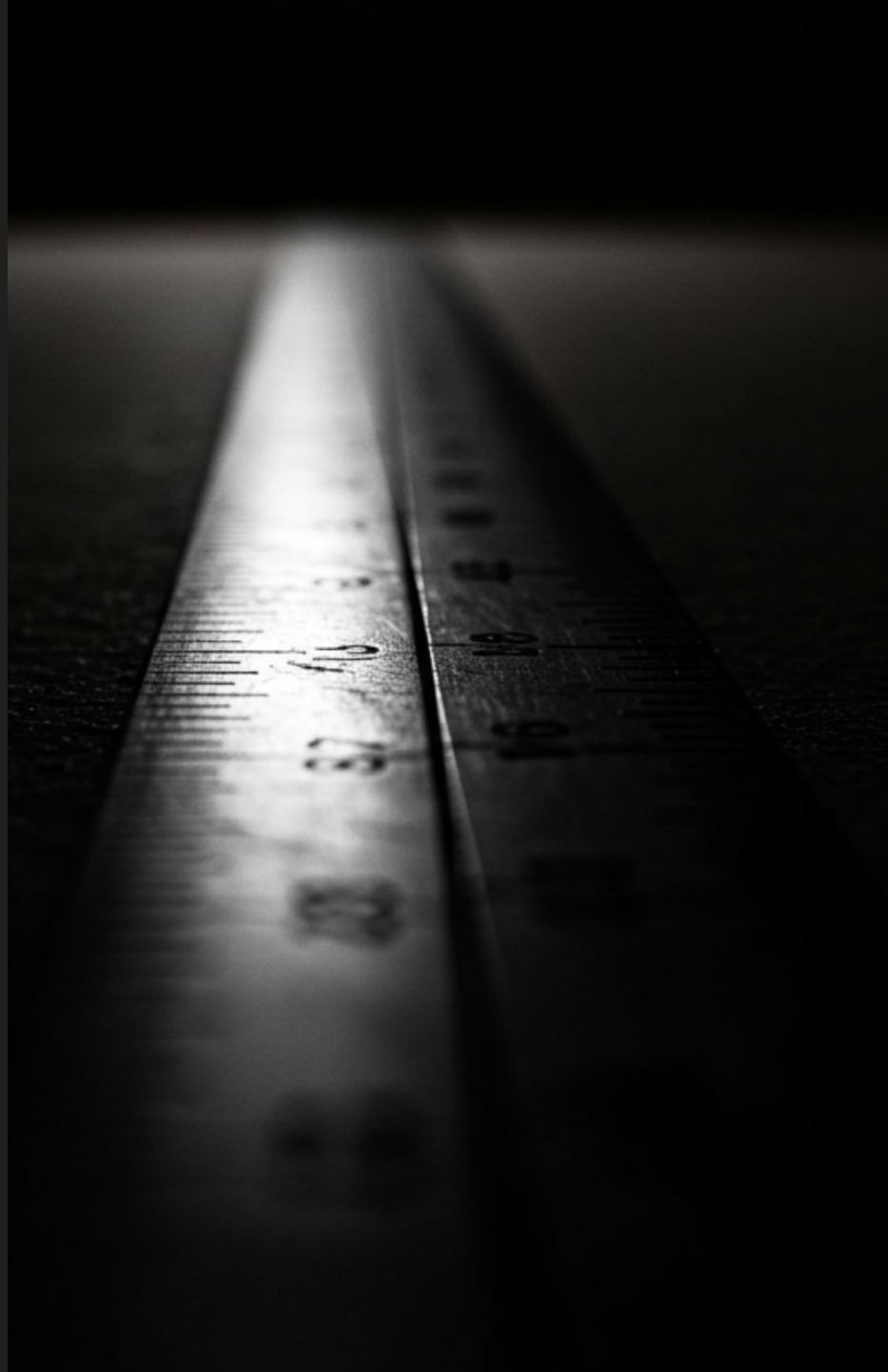
Length encoded in a header. Up to 1 GB − 1. typlen = -1 (varlena) or -2 (cstring).

typlen — three possible values

FROM pg_type — TELLS THE EXECUTOR WHAT SHAPE TO EXPECT

```
psql
SELECT typname, typlen FROM pg_type WHERE typname IN ('int4', 'text', 'cstring');
```

> 0	−1	−2
Fixed-length	**varlena**	**cstring**
int4 → 4 int8 → 8 bool → 1	text · bytea varchar · jsonb numeric · arrays	internal only — never stored on disk (used by I/O funcs)

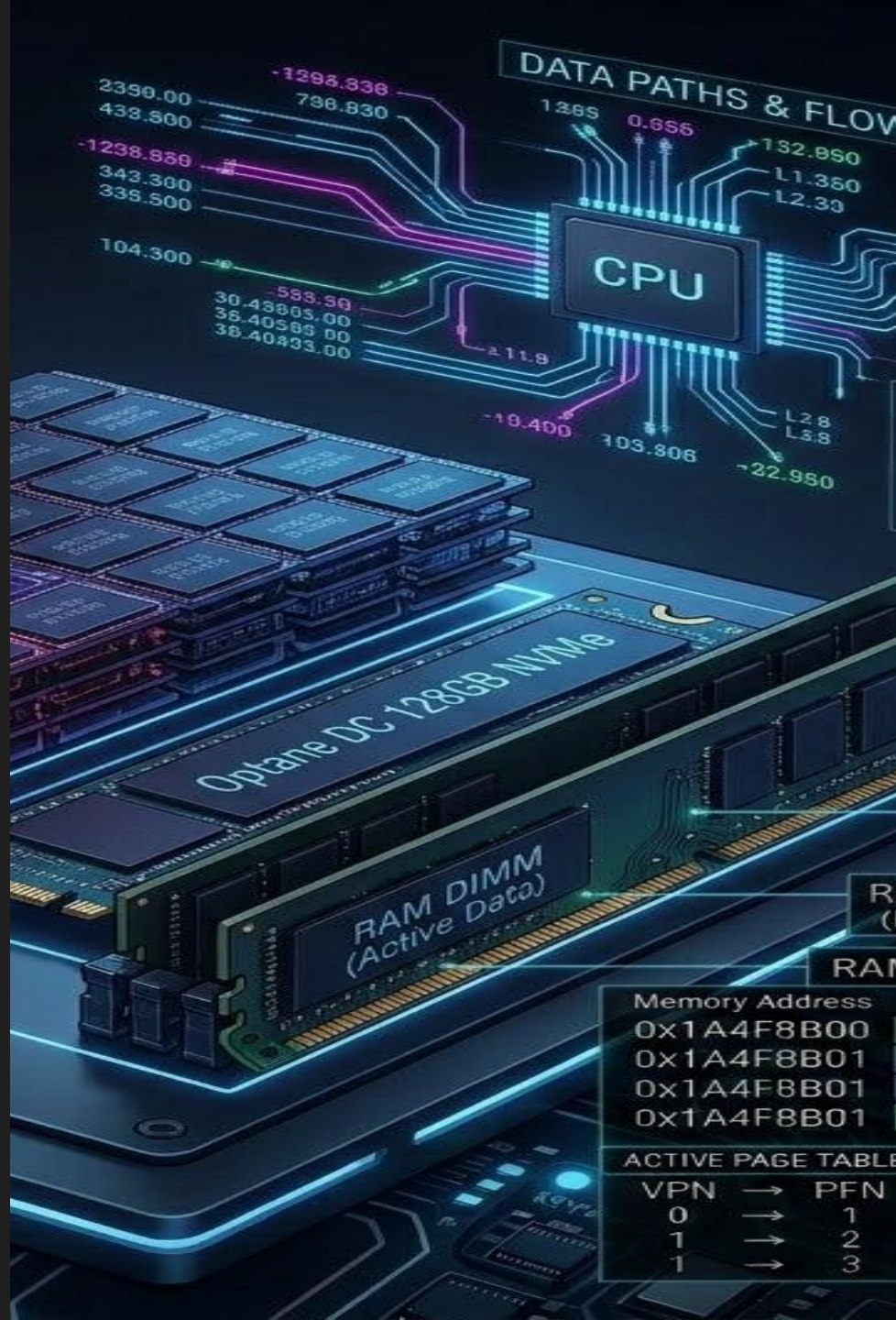


EDB POSTGRES AI

PART 02

Values in Memory

Datums, pointers, and the universal value container.



Datum — universal value container

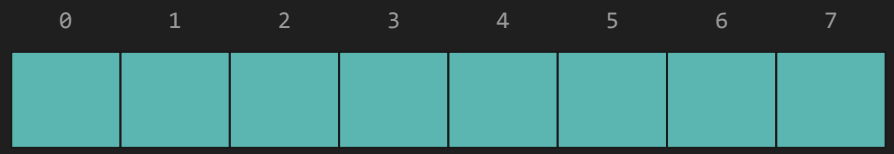
EVERY VALUE IN THE EXECUTOR IS WRAPPED IN ONE

src/include/postgres.h

```
typedef uint64_t Datum;

#define SIZEOF_DATUM 8
```

Datum is just a machine word — 8 bytes on a 64-bit system.



Pass-by-value

Small types (≤ SIZEOF_DATUM) packed directly into the slot. int4, bool, oid, float8 …

Pass-by-reference

Larger types stored elsewhere; the Datum holds a pointer. text, jsonb, numeric, arrays …

Datum conversions — encode / decode

src/include/fmgr.h — TYPE-SAFE BRIDGES BETWEEN C TYPES AND Datum

encoding (C → Datum)

```
Datum d1 = Int32GetDatum(42);

Datum d2 = Int64GetDatum(1L);

Datum d3 = BoolGetDatum(true);

Datum d4 = PointerGetDatum(text_p);

Datum d5 = CStringGetDatum("hi");
```

decoding (Datum → C)

```
int32 v1 = DatumGetInt32(d1);

int64 v2 = DatumGetInt64(d2);

bool  v3 = DatumGetBool(d3);

Pointer p4 = DatumGetPointer(d4);

char *s5 = DatumGetCString(d5);
```

WHY MACROS, NOT CASTS?

These macros facilitate seamless data flow between the executor and the internal functions.

Macros centralize the truncation, sign-extension, and ABI handling for each type — and let one codebase work on 32-bit and 64-bit platforms without changing call sites.

EDB POSTGRES AI

PART 03

Values on Disk

From Datum to HeapTuple — bytes laid out on the page.

What's different on disk?

DISK STORAGE DROPS ABSTRACTIONS MEMORY TAKES FOR GRANTED

No pointers

Disk is content-addressed; pointer / by-value distinction is erased. Every value is materialized into the page.

Tuple-only

Individual Datums are never stored alone. The unit of storage is the heap tuple — header + null bitmap + columns.

8 kB page limit

A tuple must fit inside a single 8 kB block. Large rows trigger TOAST (compress / move out-of-line).

No cstring

cstring (typlen = -2) is in-memory only. The on-disk representation must be varlena, fixed, or a TOAST pointer.

EDB POSTGRES AI

heap_form_tuple / heap_deform_tuple

TWO PRIMITIVES MOVE VALUES BETWEEN Datum[] AND HeapTuple

src/backend/access/common/heaptuple.c

```c
/* build a HeapTuple from an array of Datum values */
HeapTuple heap_form_tuple(TupleDesc tupdesc, Datum *values, bool *isnull);

/* extract Datum array from a HeapTuple */
void heap_deform_tuple(HeapTuple tuple, TupleDesc tupdesc, Datum *values, bool *isnull);
```

Datum[] + isnull[]

in-memory representation

→

heap_form_tuple()

writes the on-disk shape

→

HeapTuple

ready for the page

Use deform_tuple when reading multiple columns; for a single-column read, prefer heap_getattr — it skips earlier columns with the offset cache.

EDB POSTGRES AI

Tuple construction strategy

WHAT heap_form_tuple WRITES, IN ORDER

1. **23-byte HeapTupleHeader**
   t_xmin, t_xmax, t_cid, t_ctid, t_infomask, t_natts, t_hoff …
2. **Null bitmap (optional)**
   1 bit per column. Present only if any column is NULL.
3. **Pad to 8-byte boundary**
   MAXALIGN — first column must start on a multiple of MAXIMUM_ALIGNOF.
4. **Column values + alignment padding**
   Per-column padding from pg_type.typalign before each value.
5. **(repeat for every column)**
   Fixed types written inline; varlenas may be short, full, or TOAST pointer.

pg_type.typalign

Per-column alignment

c	**1 byte** `char, bool`
s	**2 bytes** `int2`
i	**4 bytes** `int4 · varlena`
d	**8 bytes** `int8 · float8 · timestamptz`

EDB POSTGRES AI

Example — int2 + int4 = 32-byte tuple

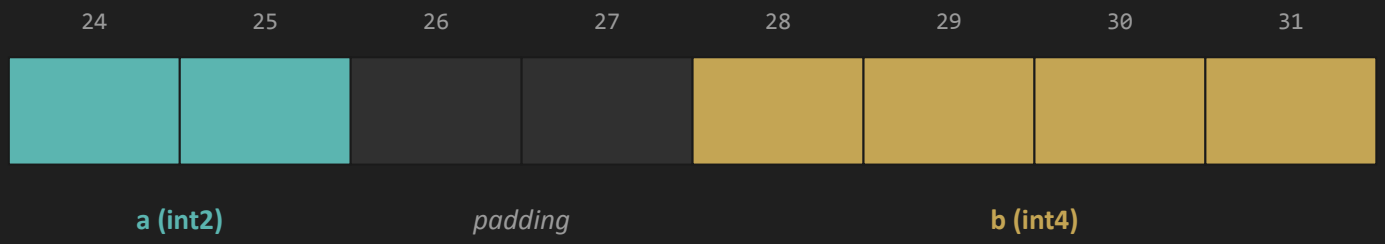
WALK THE BYTES OF heap_form_tuple FOR A TWO-COLUMN ROW

ddl

```
CREATE TABLE foo (a int2 NOT NULL, b int4 NOT NULL);
```

- Bytes 0–22 — HeapTupleHeader
- Byte 23 — pad to MAXALIGN (8-byte) boundary
- Bytes 24–25 — column a (int2)
- Bytes 26–27 — pad to int alignment for b
- Bytes 28–31 — column b (int4)

tuple, last 8 bytes (24 + 8 ⇒ 32 total):



EDB POSTGRES AI

PART 04

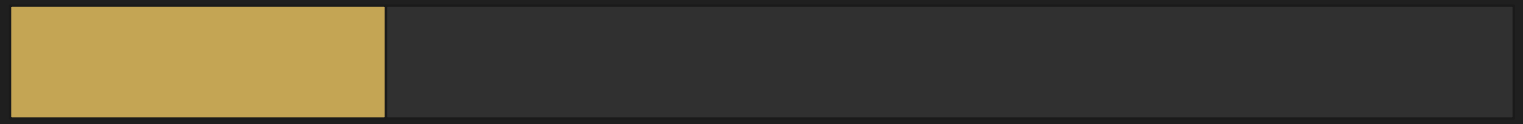
Variable Length — varlena

Length-prefixed bytes, with a clever 1-byte short header optimization.

varlena — two header formats

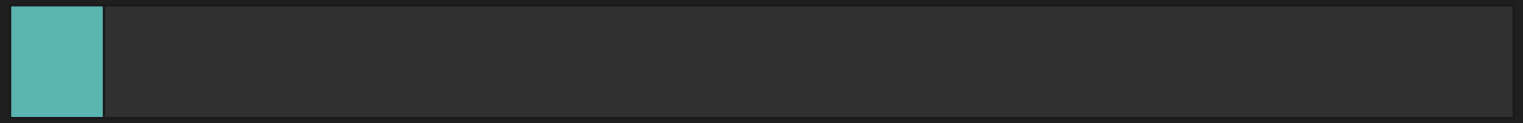
POSTGRES 8.3 ADDED A 1-BYTE SHORT HEADER FOR SMALL VALUES

4-byte header (length 4 .. 1 GB − 1)



len (30 bits + 2 flag bits) payload bytes …

1-byte header (length 1 .. 127, payload 0 .. 126)



len payload …

```
DECODE RULE: Bit layout:
  xxxxxx00    →   4-byte header, aligned, uncompressed (length up to 1 GB)
  xxxxxx10    →   4-byte header, aligned, COMPRESSED   (length up to 1 GB)
  xxxxxxx1    →   1-byte header, unaligned, short data (length up to 126 bytes)
  00000001    →   1-byte external – TOAST pointer follows
```

EDB POSTGRES AI

TOAST — flow

THE OVERSIZED-ATTRIBUTE STORAGE TECHNIQUE — KEEPS EVERY ROW UNDER 8 kB

1 → **Tuple too big**
row exceeds ~ 2 kB target

2 → **Compress**
LZ4 / pglz on widest column

3 → **Move out-of-line**
split into ~ 2 kB chunks → toast table

4 → **Replace w/ pointer**
row keeps an 18-byte ref

PER-COLUMN STORAGE STRATEGY (ALTER TABLE … SET STORAGE)

`PLAIN`	no compress, no out-of-line (fixed-len only)
`EXTERNAL`	out-of-line, no compress
`EXTENDED`	compress + out-of-line (default for varlena)
`MAIN`	compress in place; out-of-line only if needed

Demo — pg_column_size

WATCH varlena AND TOAST IN ACTION

setup

```
CREATE TABLE foo (a int2, b text, c text);
INSERT INTO foo VALUES (42, 'hello', repeat('test or something', 1000000));
```

Q1 – sizes of literals before insert

```
SELECT pg_column_size(42::int2),
       pg_column_size('hello'::text),
       pg_column_size(repeat('test or something'::text, 1000000));
```

Q2 – sizes after insert

```
SELECT pg_column_size(a), pg_column_size(b), pg_column_size(c) FROM foo;
```

Q3 – *operations re-materialize varlena*

```
SELECT pg_column_size(a + 0), pg_column_size(b || ‘’), pg_column_size(c || ‘’) FROM foo;
```

Demo — pg_column_size - result

WATCH varlena AND TOAST IN ACTION

setup

```
CREATE TABLE foo (a int2, b text, c text);
INSERT INTO foo VALUES (42, 'hello', repeat('test or something', 1000000));
```

Q1 — sizes of literals before insert

```
SELECT pg_column_size(42::int2),
       pg_column_size('hello'::text),
       pg_column_size(repeat('test or something'::text, 1000000));
```

=> 2, 9, 17000004

int2 = 2. text 'hello' = 4-byte hdr + 5 = 9. Big text = 4-byte hdr + payload.

Q2 — sizes after insert

```
SELECT pg_column_size(a), pg_column_size(b), pg_column_size(c) FROM foo;
```

=> 2, 6, 194620

int2 = 2. 'hello' shrank — short 1-byte header. Big text was compressed + TOAST-ed out-of-line.

Q3 — *operations re-materialize varlena*

```
SELECT pg_column_size(a + 0), pg_column_size(b || ''), pg_column_size(c || '') FROM foo;
```

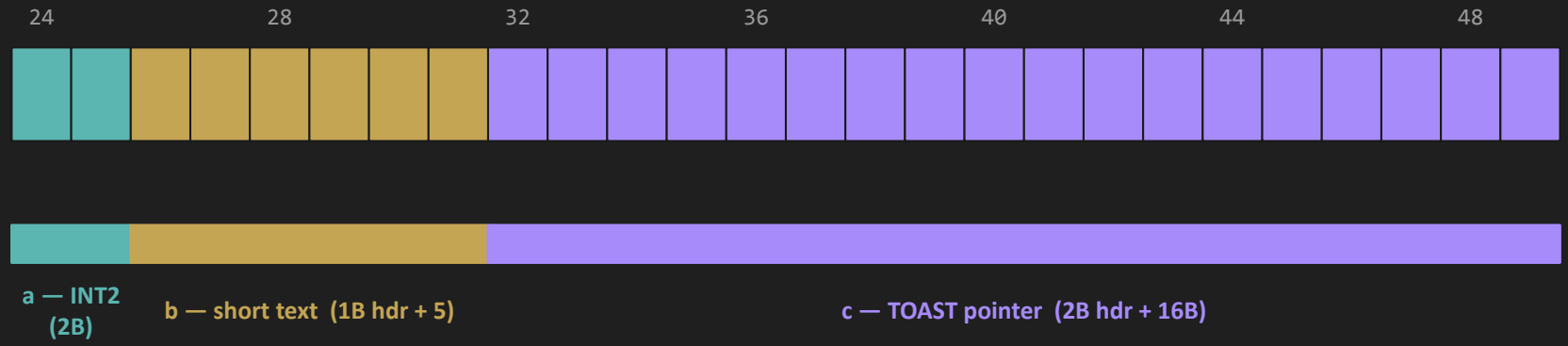
=> 4, 9, 17000004

(a + 0) widens to int4. (b || '') rebuilds with a 4-byte header. (c || '') de-TOASTs.

Stored row — byte-by-byte

foo (a int2, b text, c text) → pg_column_size = 2, 6, 194620

tuple body, last 26 bytes (24 + 26 ⇒ 50 bytes total):



a — INT2 (2B)

b — short text (1B hdr + 5)

c — TOAST pointer (2B hdr + 16B)

Note: column c — a 17 MB string — is replaced in the row by an 18-byte TOAST pointer.

TOAST pointer — the 18 bytes

WHAT THE STUB IN THE MAIN ROW ACTUALLY HOLDS

2-byte header + 16-byte varatt_external struct:

rawsize
4 bytes
original size incl. varlena header

extsize
4 bytes
stored size on disk excl. header

valueid
4 bytes
chunk_id in the toast table

toastrelid
4 bytes
OID of the toast table

Tag = 0x01 marks a toasted attribute; 0x12 (= VARTAG_ONDISK = 18) means "on-disk pointer".

PART 05

Alignment and Padding

Why the bytes between columns matter.

Alignment — the rule

EVERY FIELD STARTS AT AN OFFSET DIVISIBLE BY ITS REQUIRED ALIGNMENT

> *An int8 at byte 6 isn't an int8 — it's a slow read on x86 and a SIGBUS on strict-aligned ISAs.*

psql

```
SELECT typname, typalign FROM pg_type WHERE typname IN ('int4', 'text', 'cstring');
```

typalign	bytes	C type	examples
c	1	char	char · bool
s	2	short	int2
i	4	int	int4 · float4 · oid · varlena (header)
d	8	double	int8 · float8 · timestamptz · interval

Why alignment matters

FOUR REASONS POSTGRES BAKES ALIGNMENT INTO HEAP TUPLES

Single-fetch reads

Aligned 8-byte values fit in one CPU word fetch. Misalignment doubles memory traffic.

Cache-line locality

Fields stay within 64-byte cache lines — fewer L1/L2 misses.

Atomicity

Aligned loads / stores are atomic on most ISAs. Misaligned reads can split across writes.

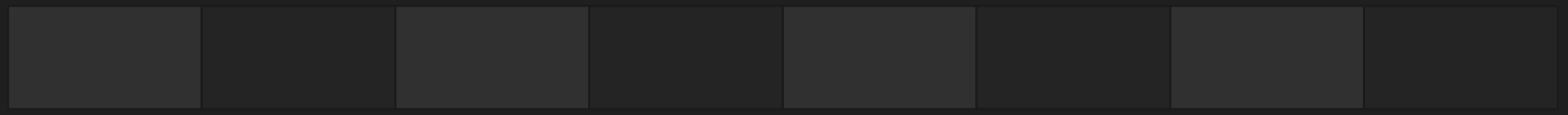
Portability

ARM, SPARC, RISC-V either trap on misalignment or emulate it slowly.

CPU access — cache lines & words

MODERN CPUs READ MEMORY IN 64-BYTE CACHE LINES, IN 8-BYTE WORDS

One 64-byte cache line = 8 × 8-byte words:



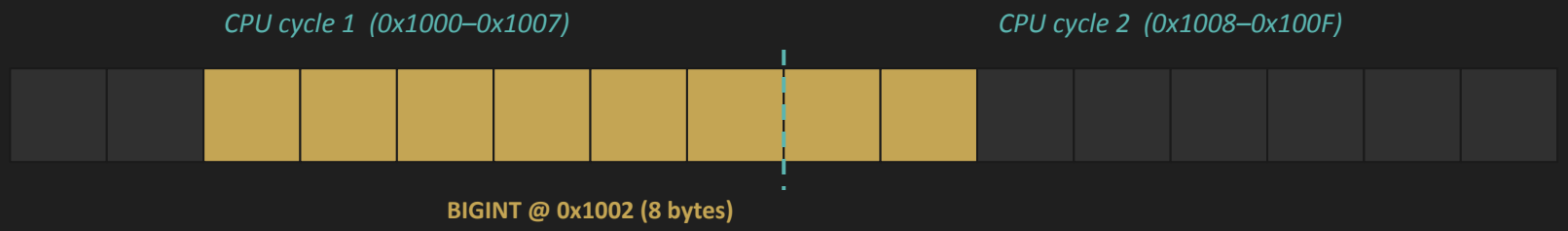
1. **Aligned 8 Byte reads → one word fetch**
   An int8 at offset 0, 8, 16 … is a single load.

2. **Misaligned 8 Byte reads → two fetches**
   An int8 at offset 6 spans words 0 and 1. Splits into two loads + a merge.

3. **Word size matches register size**
   8 bytes on 64-bit, 4 on 32-bit. Aligned data flows directly into registers.

Example: Aligned vs. misaligned reads

AN int8 (BIGINT) AT ADDRESS 0x1002 STRADDLES TWO CPU WORDS



1 fetch

aligned at 0x1000 (or 0x1008)

2 fetches

misaligned — split read + merge

Where alignment shows up in PostgreSQL

FIVE PLACES THE PADDING BYTES PAY OFF

1

Tuple storage

Per-column alignment from typalign in the heap.

2

Shared buffers

8 kB pages aligned to BLCKSZ in BufferTag.

3

Index pages

Same 8 kB rule; aligned ItemPointers.

4

WAL records

MAXALIGN padded so XLogRecord reads are atomic.

5

palloc / MemoryContext

All allocations are MAXALIGN rounded — every Datum is safe.

Key Distinctions

Anatomy of a Data Type —— fixed vs. variable, varlena vs. cstring, 4-byte vs. 1-byte.

FIXED-LENGTH

Data type

- **Pass-by-value**
  `fits in a Datum slot`
- **Pass-by-reference**
  `Datum is a pointer`

VARIABLE-LENGTH *always pass-by-reference*

- **cstring**
- **varlena**
  `len + payload`
  - **4-byte header**
    - **Compressed**
    - **Uncompressed**
  - **1-byte header**
    - **Short data**
    - **TOAST pointer**

ALIGNMENT

fields padded to start at their typalign boundary

c	1 byte	`char, bool`
s	2 bytes	`int2`
i	4 bytes	`int4, varlena`
d	8 bytes	`int8, float8`

EDB POSTGRES AI

EDB POSTGRES AI

Thank You!

I look forward to discussing your thoughts and contributions.

EMAIL ME

amul.sul@enterprisedb.com

CONNECT

https://www.linkedin.com/in/sulamul/

References

SOURCES

1

EDB PG Internal Knowledge Sharing

Talks by Robert Haas — internal series, EnterpriseDB

Source for several examples and the structure of this talk

2

PostgreSQL Column Alignment and Padding

Percona Blog — how to improve performance with smarter table design

https://www.percona.com/blog/postgresql-column-alignment-and-padding-how-to-improve-performance-with-smarter-table-design/

3

PostgreSQL source code

github.com/postgres/postgres — heaptuple.c, postgres.h, varatt.h

https://github.com/postgres/postgres