Optimising Column Order for Better Performance and Maintainability

Discover how the arrangement of fixed-length and variable-length columns can significantly enhance database efficiency and maintainability.

Amul Sul

EDB

A Little About Me

Amul Sul

Database Developer

EnterpriseDB

Pune, India

Office Location

12 Years

Expertise in PostgreSQL internal development

EDB

Why It Matters

Faster Queries

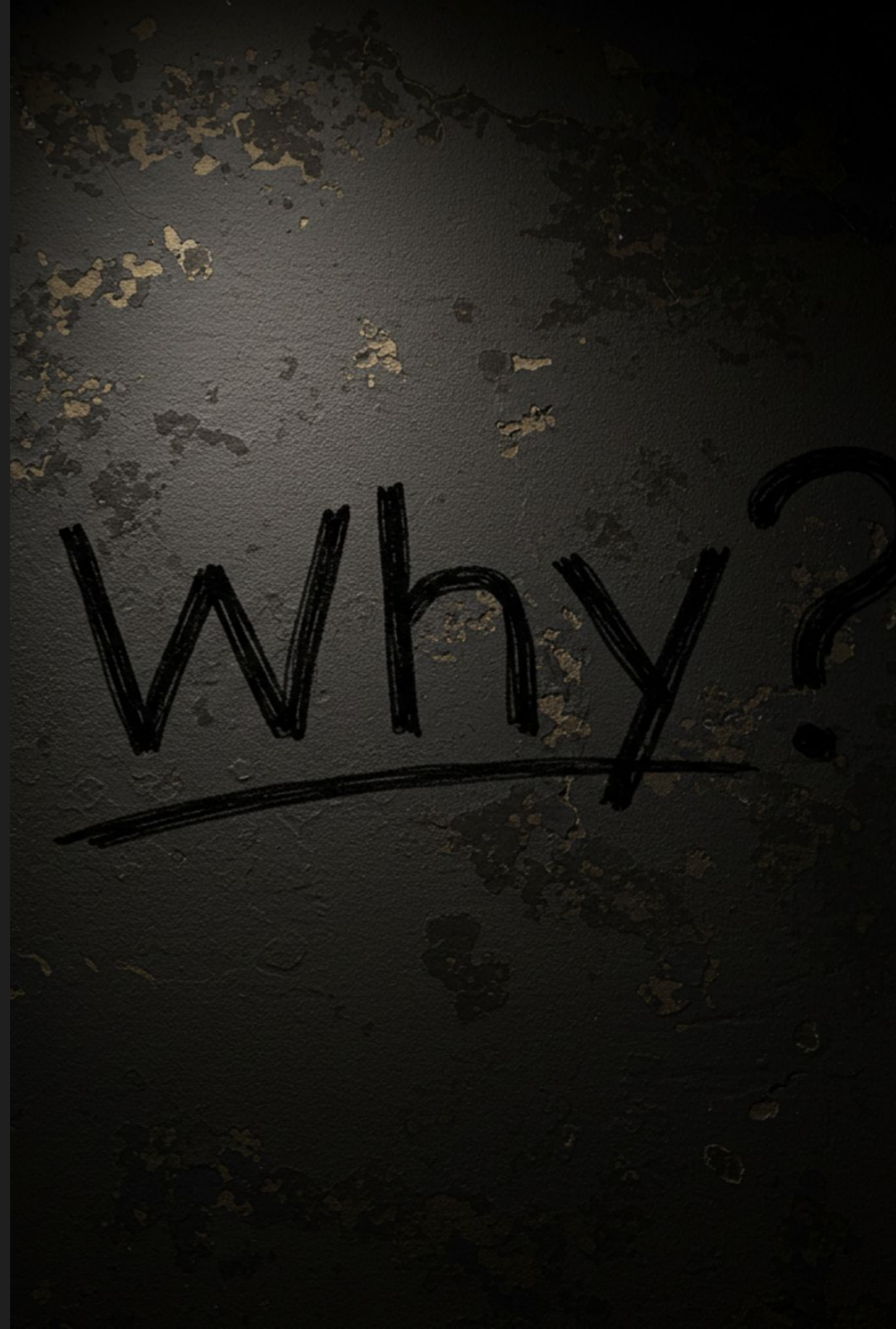
Improved user experience

Reduced Costs

Lower infrastructure needs

Competitive Edge

Outperform competitors

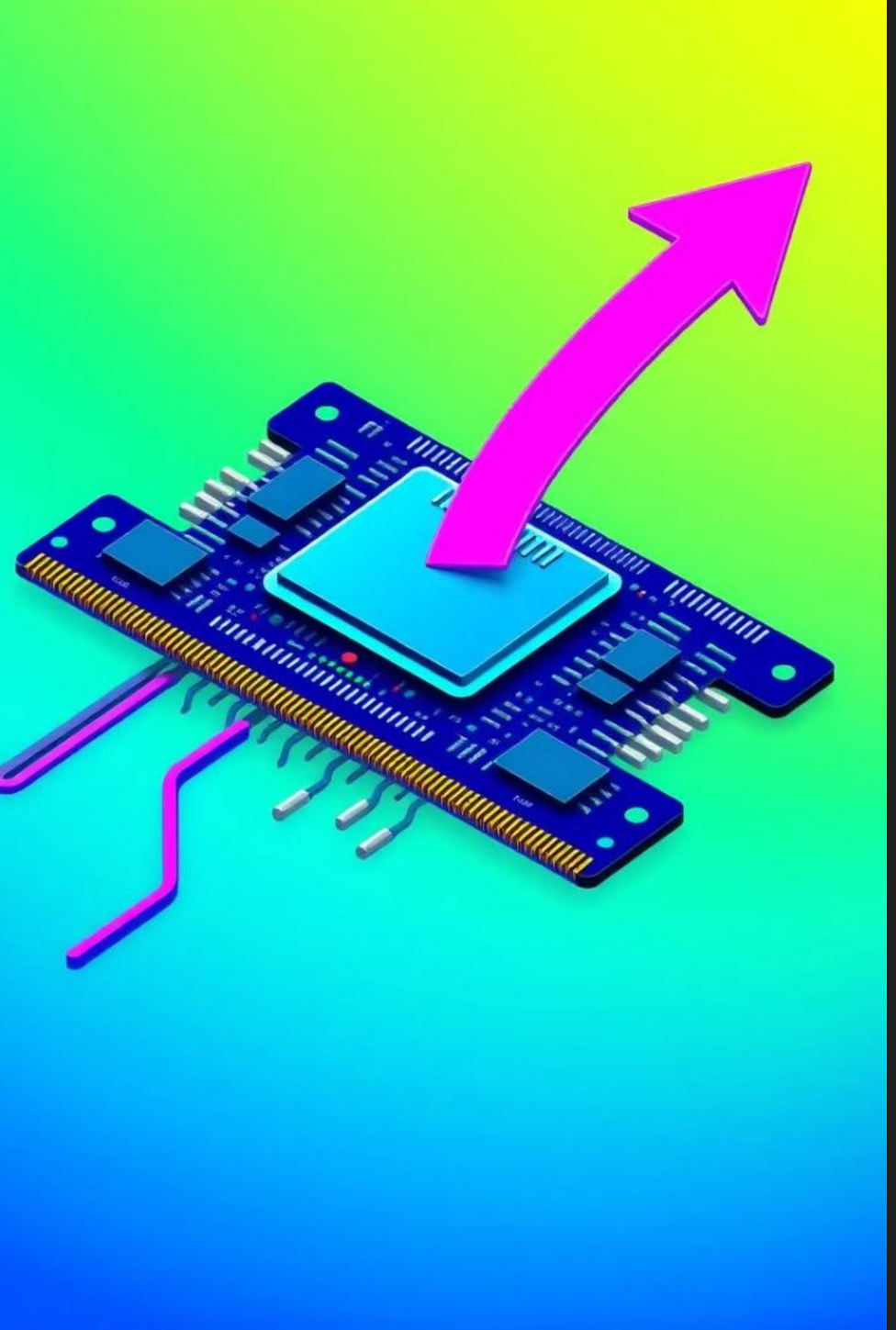


EDB

Agenda

1. Access Latencies
   Efficiency impact
2. Values and Types in PostgreSQL
   Understanding of basic blocks
3. Tuple construction
   Storage layout of columns
4. Alignment & Padding
   Impact on performance
5. Advantages & Limitations
   Reducing IO
6. Consideration
   Optimization Need & Modern Hardware

EDB



Access Time from CPU to DISK

1. CPU
   Fastest access
2. RAM
   Slower access
3. Disk
   Slowest access

EDB

Latency Numbers Every Programmer Should Know

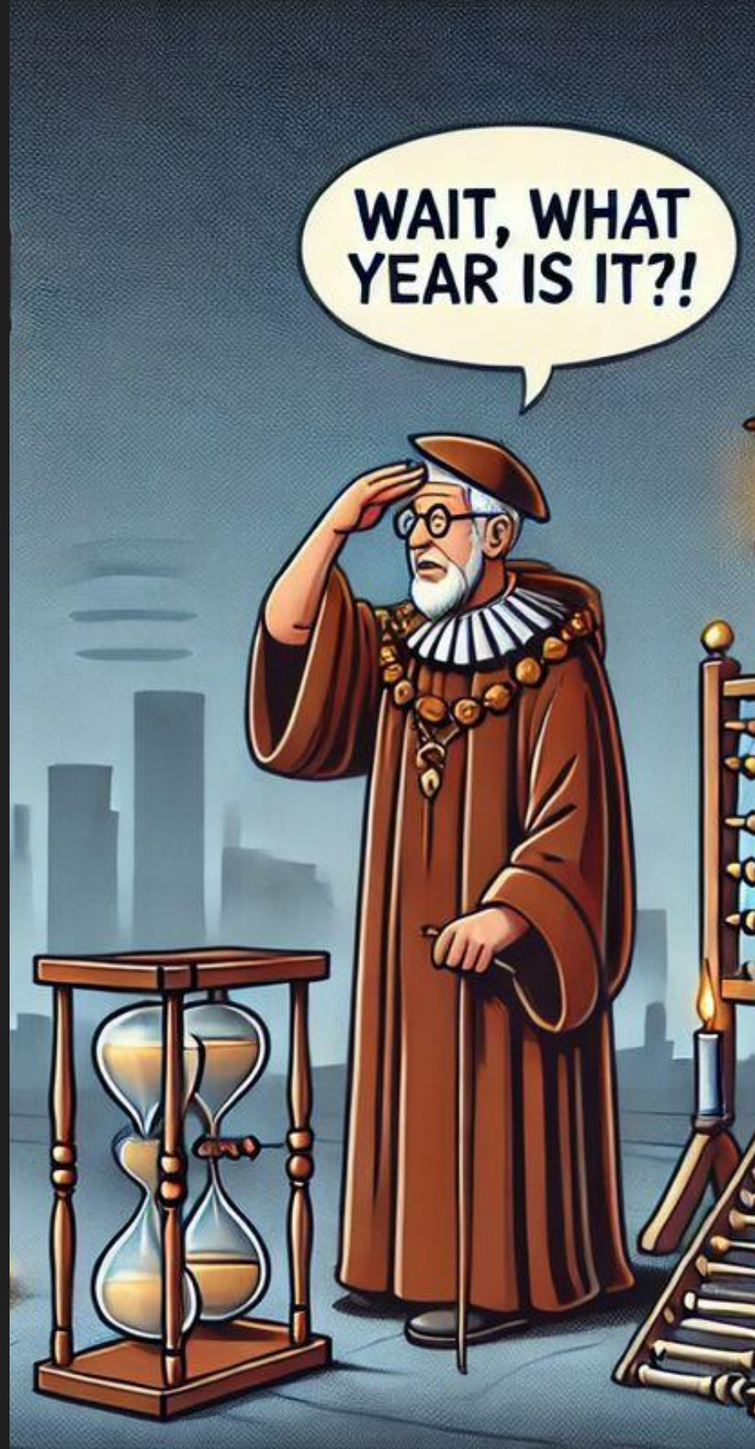
Operation	Latency
L1 cache reference	1 ns
L2 cache reference	4 ns
Main memory reference	100 ns
Read 1 MB sequentially from memory	3,000 ns
SSD random read	16,000 ns
Read 1 MB sequentially from SSD	49,000 ns (49 µs)
Disk seek	2,000,000 ns (2 ms)
Read 1 MB sequentially from disk	825,000 ns (825 µs)

EDB

Source: Colin Scott

Latency Numbers Every Programmer Should Know

Operation	Latency	1 ns = 1 sec
L1 cache reference	1 ns	1 sec
L2 cache reference	4 ns	4 sec
Main memory reference	100 ns	100 sec
Read 1 MB sequentially from memory	3,000 ns	3,000 sec (50 min)
SSD random read	16,000 ns	16,000 sec (4.4 hr)
Read 1 MB sequentially from SSD	49,000 ns (49 µs)	49,000 sec (13.6 hr)
Disk seek	2,000,000 ns (2 ms)	2,000,000 sec (23 days)
Read 1 MB sequentially from disk	825,000 ns (825 µs)	825,000 sec (9.5 days)



EDB

What is a Value?

- **4**
- **4.0**
- **’four’**

EDB

Source: EDB internal talk by Robert Haas

What is a Value? (More Accurate Version)

- **4**::pg_catalog.int4
- **4**::pg_catalog.int8
- **4**::pg_catalog.int2

- **4.0**::pg_catalog.numeric
- **4.0**::pg_catalog.float8

- **‘four’**::pg_catalog.text
- **‘four’**::pg_catalog.varchar

EDB

Source: EDB internal talk by Robert Haas

Fixed Length vs. Variable Length

Fixed-Length

Predefined size: BOOL, CHAR, INT, FLOAT, etc

E.g. pg_catalog.int4 means a 4-byte integer, it should be stored using a fixed amount of storage, namely 4 bytes.

Variable-Length

Dynamic size: TEXT, BYTEA, VARCHAR, etc.

Can store as little as 0 bytes - that is, an empty string - and as much as 5 bytes less than 1GB, it should be stored using a variable amount of storage.

EDB

Understanding Type “Lengths”

1. **pg_type.typlen**

   Indicates the length of a datatype.

2. **Positive values**

   Indicates that the data type is fixed-length.

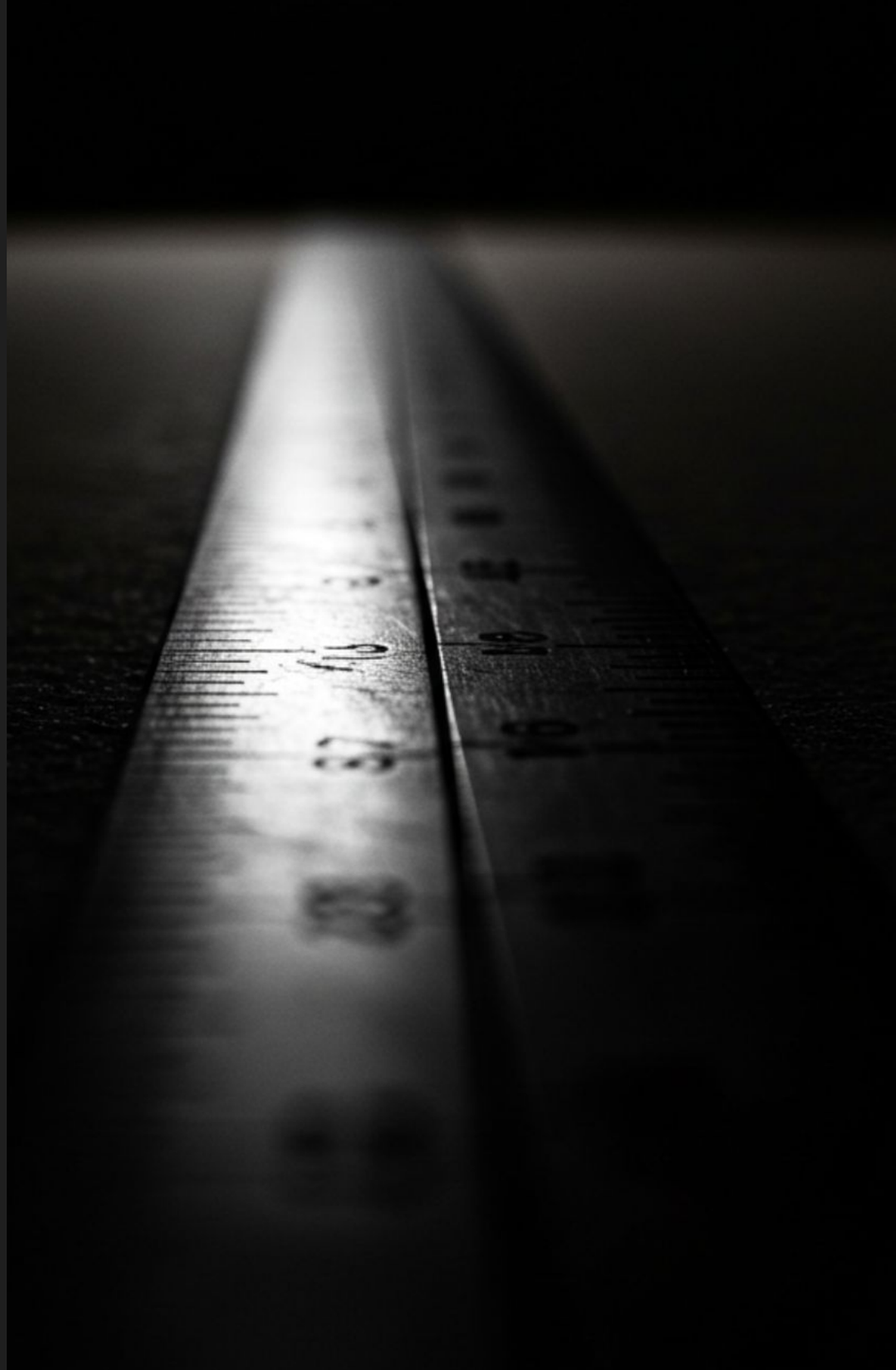
3. **Negative value**

   Indicates that the data type is variable-length.

   -1: Indicating that a value is stored as a varlena (most of).

   -2: Indicating that a value is stored as a cstring (very few).

4. **There are no other possibilities (currently).**



EDB

Tuple Construction Strategy

- 23-byte header.
- Null bitmap, if the tuple contains any null columns, with 1 bit per column.
- Pad out to an 8 byte boundary.
- Column values, possibly separated by more alignment padding.
  - Column alignment is defined by **pg_type.typalign** – '**c**' = 1, '**s**' = 2, '**i**' = 4, '**d**' = 8.
  - **A column must start at an offset which is a multiple of the required alignment.**

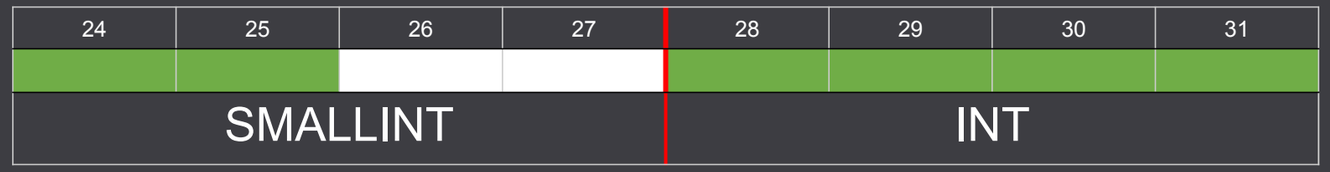
EDB

Tuple Construction Example

CREATE TABLE foo (a int2 not null, b int4 not null);

- **Bytes 0-22**: Tuple header.
- **Byte 23**: Padding byte, so that first column starts on a **multiple of 8**.
- **Bytes 24-25**: Column **a**.
- **Bytes 26-27**: Padding bytes, so that second column starts on a **multiple of 4**.
- **Bytes 28-31**: Column **b**.

tuple (24 + 8 => 32 bytes):



EDB

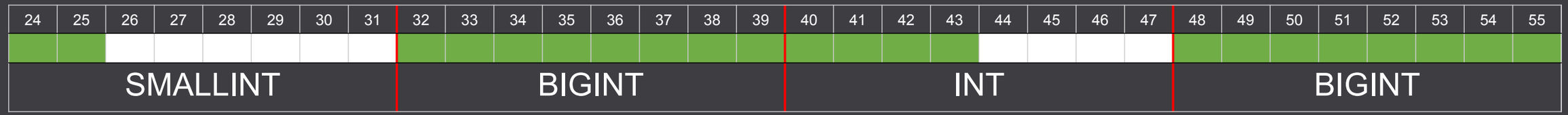
Example 2

```
CREATE TABLE example (
    c1    SMALLINT,
    c2    BIGINT,
    c3    INT,
    c4    BIGINT);
=> (1, 1, 1, 1)
```

```
=> SELECT pg_column_size(t.*) - 24  AS row_size FROM example t;

 row_size
----------
       32
(1 row)
```

tuple (24 + 32 => 56 bytes):



EDB

Source: Percona blogs

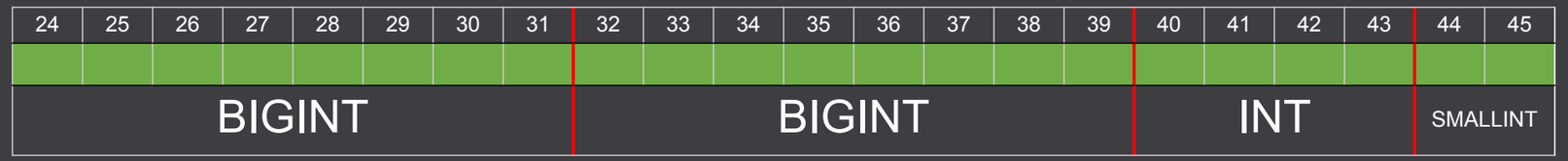
Example 2

```
CREATE TABLE optimized_example (
    c1    BIGINT,
    c2    BIGINT,
    c3    INT,
    c4    SMALLINT);
=> (1, 1, 1, 1)
```

```
=> SELECT pg_column_size(t.*) - 24  AS row_size FROM optimized_example t;

row_size
----------
        22
(1 row)
```

tuple (24 + 22 => 46 bytes):



EDB

Source: Percona blogs

Variable Length – varlena

1. Simple varlena - 4 byte length word
   - The simplest form of varlena is a 4-byte length word where the length value can range from 4 to 1 byte less than 1GB.
   - The length word is followed by the payload bytes.
   - This form of varlena requires 4-byte alignment.
2. Short verlena - 1 byte length word
   - The varlena length is 1-127 bytes, and so the payload is 0-126 bytes.
   - If non-zero, it’s the beginning of a varlena with a 1-byte header. If it’s zero, skip to the next 4 byte boundary; a 4-byte varlena header begins there.
   - If we are starting on a 4-byte boundary, read 1 byte initially. The value we read will tell us whether it’s the first byte of a 4-byte header, or the only byte of a 1-byte header.

EDB

Variable Length – TOASTing

1. The Oversized-Attribute Storage Technique - TOAST
   - PostgreSQL uses TOAST to store large TEXT values.
   - By default, large variable length values are compressed and sometimes stored out-of-line.
2. Why do we have TOAST?
   - Limiting the size of a tuple to what can fit into a single 8kB page would be unacceptable.
   - We need a way to take a tuple that might be quite large and turn it into one or more tuples each of which can fit into an 8kB page.
3. How does TOAST work?
   - Replace larger varlenas with smaller ones.
   - Repeat until the tuple is as small as you need or want it to be, or until there’s nothing else that can be done.
   - Fixed-size data types are not affected by TOAST.

EDB

Example 3

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

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

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

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

EDB

Source: EDB internal talk by Robert Haas

Example 3

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

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

=> **2, 9, 17000004**

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

=> **2, 6, 194620**

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

=> **4, 9, 17000004**

EDB

Source: EDB internal talk by Robert Haas

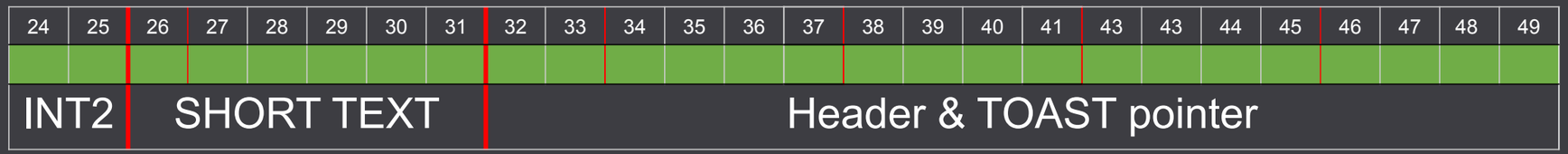
Example 3

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

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

=> **2, 6, 194620**

tuple (24 + 26 => 50 bytes):



EDB

Example 3

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

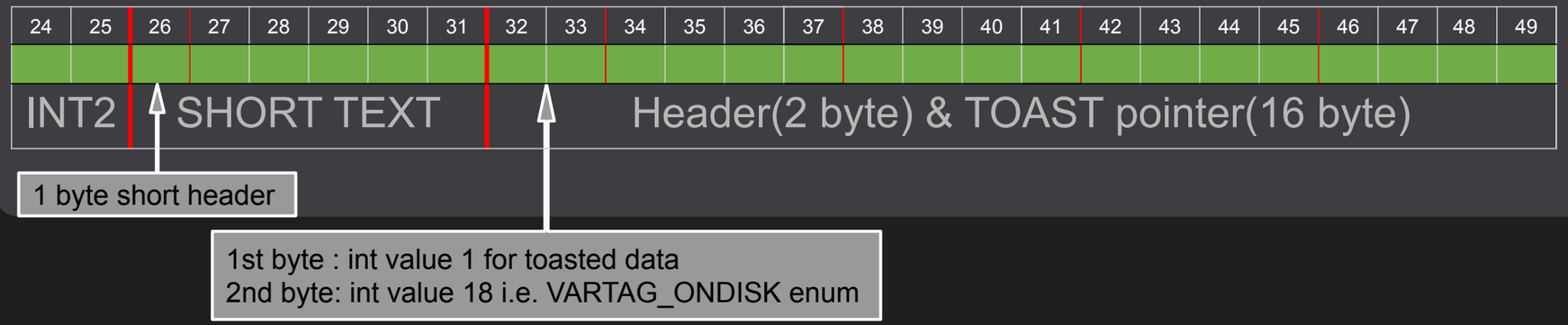
```
SELECT pg_column_size(a), pg_column_size(b), pg_column_size(c) from foo;
```

=> **2, 6, 194620**

Rest of 16 byte is TOAST pointers stores:

- Original data size including header (4 byte)
- External saved size excluding header(4 byte),
- Unique ID of value within TOAST table (4 byte)
- RelID of TOAST table containing it (4 byte)

tuple (24 + 26 => 50 bytes):



EDB

Variable Length - Optimization

1. STORAGE PLAIN **for frequently accessed small values**

   If most of your TEXT values are small (<2 KB) and frequently accessed, storing them with PLAIN storage avoids unnecessary TOAST compression overhead.

2. STORAGE EXTERNAL **for large values**

   If TEXT values are large (>2 KB) and infrequently accessed, storing them in TOAST reduces main table size, improving query performance.

3. STORAGE EXTENDED **(default) if compression helps**

   If compression significantly reduces storage.

4. Configuration

   Some of this behavior is configurable using ALTER TABLE .. SET STORAGE or by setting the toast_tuple_target relation option.

EDB

Variable Length - Placement

1. **Reduces Row Size and Improves Cache Efficiency**

   Placing variable-length columns at the end minimizes padding waste, reduces row size, and improves cache locality by allowing more rows per page.

2. **Optimizes TOAST Usage & Minimizes Data Reads**

   Placing variable-length columns at the end allows PostgreSQL to access fixed-length data first, avoiding unnecessary TOAST retrieval, as large values (>2KB) are stored out-of-line.

3. **Improves Index Efficiency**

   PostgreSQL indexes store row references (ctid) without large column values, so placing variable-length data at the end ensures compact row storage, optimizing indexed queries like WHERE id = 123.

EDB

Alignment and Padding

- Memory alignment and padding are fundamental concepts in low-level programming, particularly in C, C++, and system-level programming.
- Alignment is arranging data in memory at addresses that are multiples of the data type's size.
- PostgreSQL defaults to 8-byte and for the data type it uses **pg_type.typalign** alignment

EDB

Why Alignment and Padding

PostgreSQL is a performance-sensitive system that handles large amounts of structured data. Proper memory alignment ensures

Efficient CPU access to structured data in shared buffers and tuples.

Reduced cache misses and better CPU cache utilization.

Avoidance of unaligned memory access penalties, especially on architectures like ARM.

Optimization of struct layouts to minimize wasted memory while ensuring correctness.

EDB

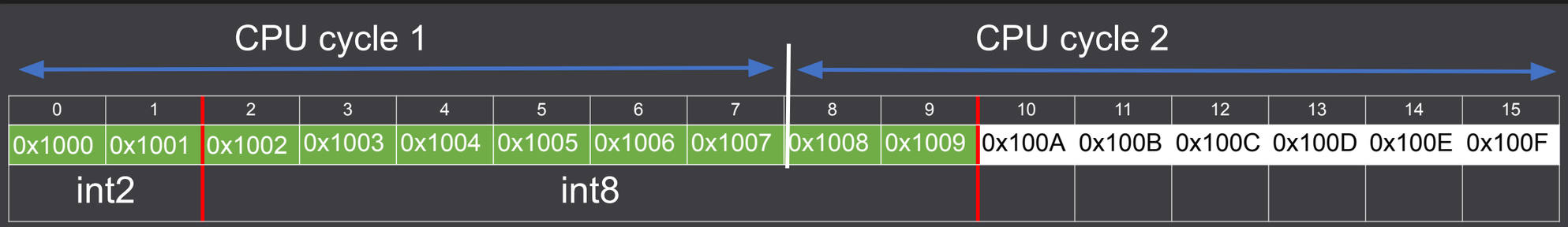
Efficient CPU access

Modern CPUs read and write memory in fixed-size chunks known as cache lines or words, rather than byte-by-byte. These chunks are typically 4, 8, or 16 bytes, with 8 bytes (64-bit word) being the most common in modern architectures.

1. **Efficiency in Memory Fetching:**
   CPUs fetch data from RAM in blocks to reduce the number of memory accesses.
   Instead of fetching a single byte at a time (which is slow), CPUs read multiple bytes in one go.
2. **Alignment with CPU Registers:**
   The word size typically matches the CPU's register size (e.g., 4 bytes for 32-bit, 8 bytes for 64-bit), ensuring efficient aligned memory access and avoiding extra operations caused by misalignment.
3. **Cache Optimization:**
   CPUs use L1/L2/L3 caches to store frequently accessed data, optimizing performance by fetching memory in 8-byte or larger blocks.

Example: CPUs Reads

- Suppose a CPU needs to read an BIGINT (8-byte integer) stored at memory address 0x1002 (unaligned).
- The CPU, working in 8-byte chunks, will need to read both 0x1000–0x1007 and 0x1008–0x100F, causing an extra fetch.
- If the data was aligned at 0x1000 or 0x1008, only one memory fetch would be needed—improving efficiency.



EDB

Alignment In PostgreSQL

1. **Tuple Storage**
   Ensures proper field alignment for efficient access.
2. **Shared Buffers**
   Memory pages must be aligned to avoid performance penalties.
3. **Index Structures**
   Proper alignment improves lookup speed.
4. **WAL Logging**
   WAL records are aligned for efficient sequential writes.
5. **Dynamic Memory Allocation**
   Ensures all allocated memory follows alignment constraints.

EDB

Advantages of Column Order Optimization

Pros
Optimised storage
Reduced I/O

EDB

Storage efficiency

```
=> CREATE TABLE example (c1 SMALLINT, c2 BIGINT, c3 INT, c4 BIGINT);
=> INSERT INTO example SELECT 1, 1, 1, 1 FROM generate_series(1, 10000000);
=> SELECT pg_size_pretty(pg_total_relation_size('example'));
 pg_size_pretty
----------------
 575 MB
```

```
=> CREATE TABLE optimized_example (c1 BIGINT,  c2 BIGINT, c3 INT,  c4 SMALLINT);
=> INSERT INTO optimized_example SELECT 1, 1, 1, 1 FROM generate_series(1,10000000);
=> SELECT pg_size_pretty(pg_total_relation_size('optimized_example'));
 pg_size_pretty
----------------
 498 MB                    <-------- 77 MB less
```

EDB

Reduced I/O

1

Faster Sequential Scans

Improve data locality for faster retrieval.

2

Faster Maintenance Activity

Minimized total amount of read

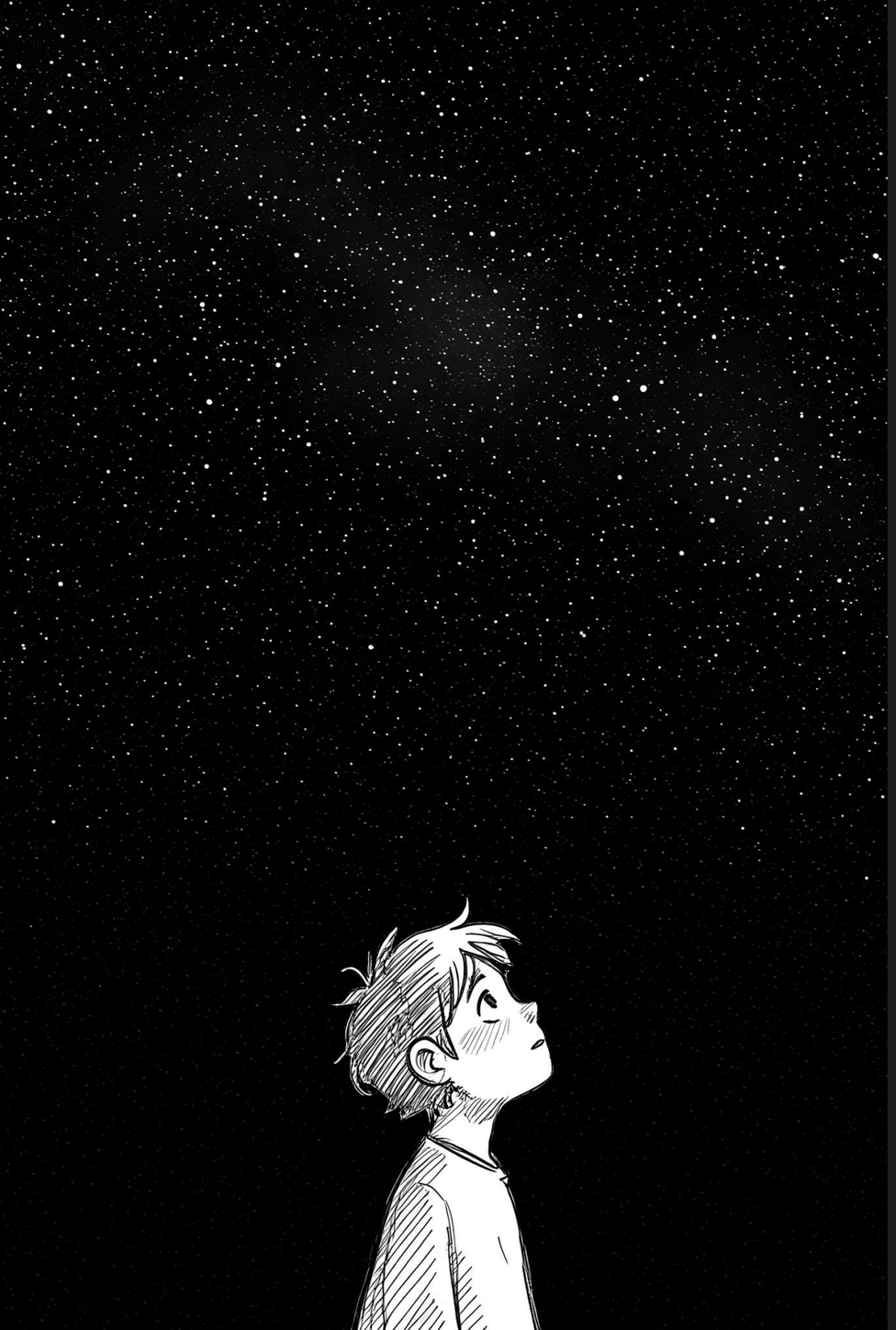
3

Index Optimization

Optimize the index structure to enhance lookup speed.

(E.g index only scans).

EDB



Limitations of Column Order Optimization

Cons
Increased query complexity.
Reduced maintainability.

EDB

Maintainability Considerations

Complex Queries

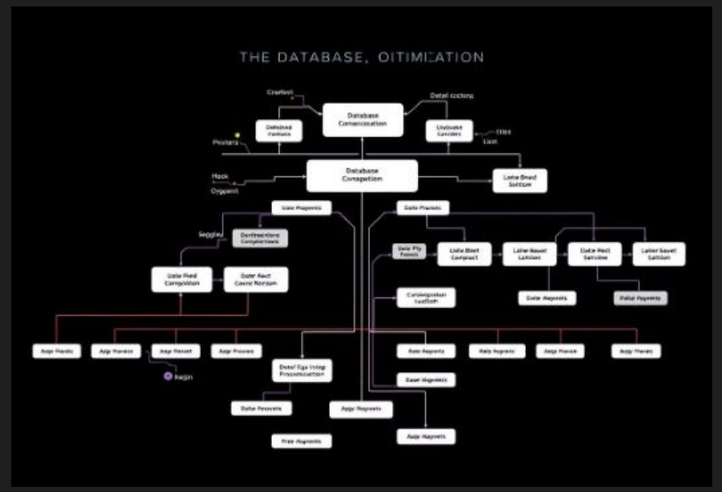
Balance performance with maintainability.

Long-Term Costs

Consider long-term costs of optimizations.

EDB

When to Optimize Column Order



High cardinality columns

For efficient filtering.

Frequently accessed columns

Reduce I/O operations.

Fixed-width columns

Optimize storage usage.

EDB

Modern Hardware Advancements

1

SSD Adoption

Increased adoption of solid-state drives (SSDs).

2

Memory Capacity

Larger memory capacities.

EDB

Key Takeaways

1. **Strategic order**
   Optimize performance.
2. **Regular analysis**
   Adapt to changes.

EDB