Reducing Costs and Improving Performance With Data Modeling in Postgres

Charly Batista

PERCONA

Databases run better with Percona

Who am I?

Blame ChatGPT for this introduction!

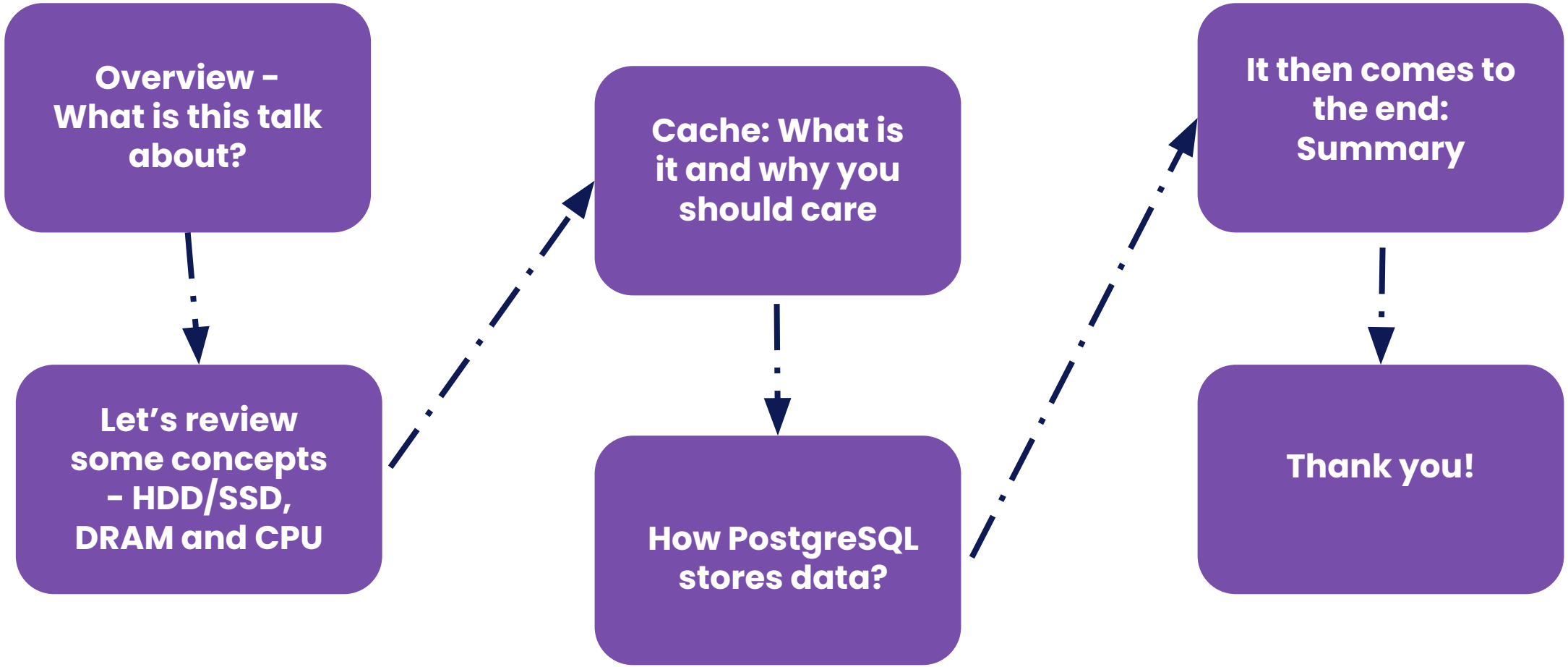
I am Charly Batista

The PostgreSQL Tech Lead at Percona with a knack for turning database queries into poetry! When not crafting SQL magic, you'll find me trading database tips over caipirinhas in Brazil or perfecting their chopstick skills in China. With a love for both the binary world and the great outdoors, I'm equally at home crunching numbers and scaling mountains. Buckle up for a database adventure like no other – this DB guy is ready to merge cultures and conquer queries with a dash of humor!

You can find me at https://www.linkedin.com/in/charlybatista

Agenda

This is what we'll cover today



What is this talk about?

This talk is about how computer stores and work with data and how PostgreSQL does it and the relationship to our data model

What is this talk about?

This is what we'll discuss here today:

- How a computer stores data
- How Postgres stores data
- What is cache and why it matters
- How the different data types impact in the data size and caching

What is this talk about?

We will then relate all the above to understand:

- How the bad design can hurt performance
- How the bad design can hurt your wallet
- Techniques to improve the design
- Examples

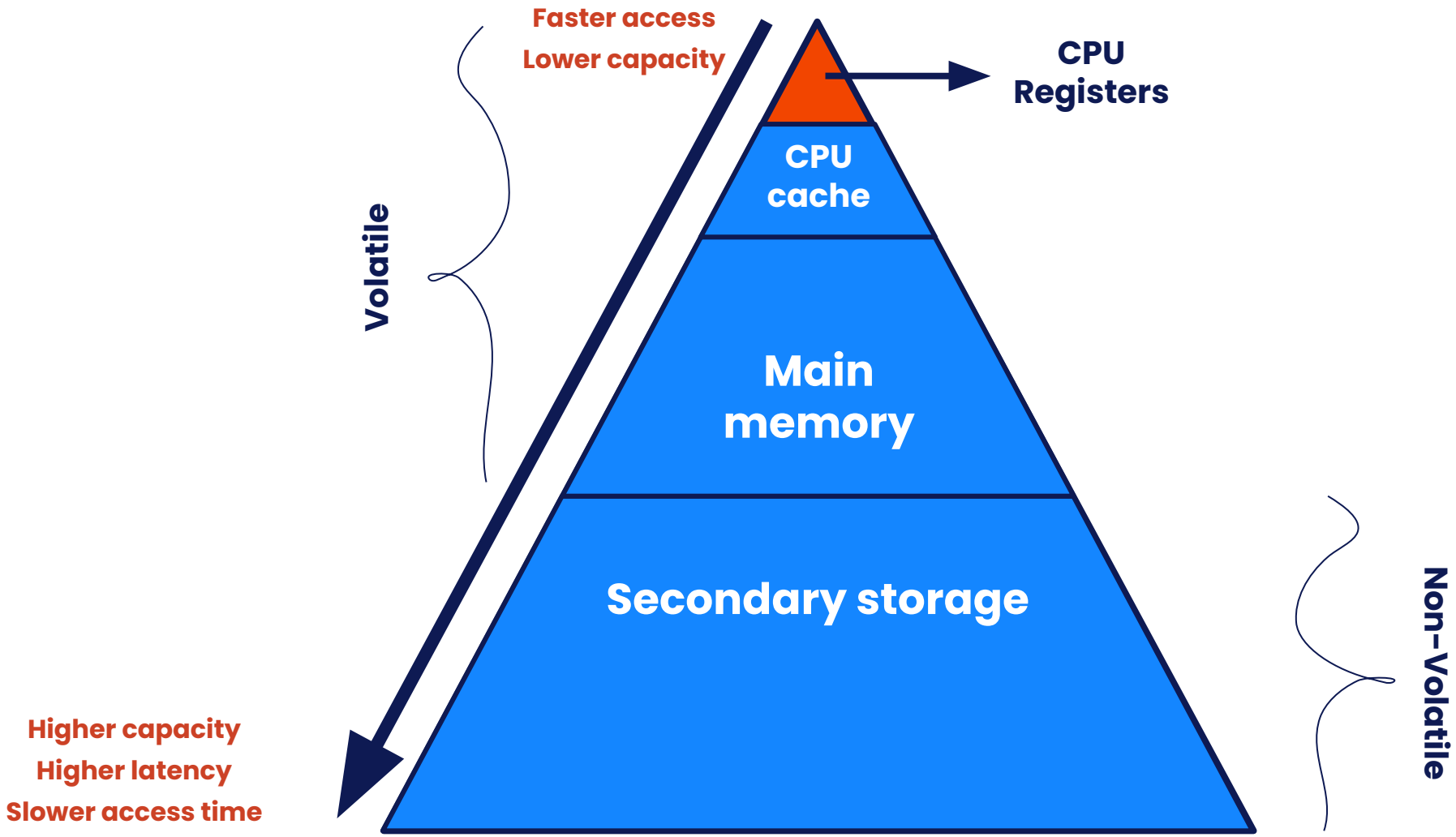
Note that we will use use expressions “block” and “page”

intertengely with the same meaning during this talk

Let’s review some concepts

HDD/SSD, DRAM and CPU

Memory architecture



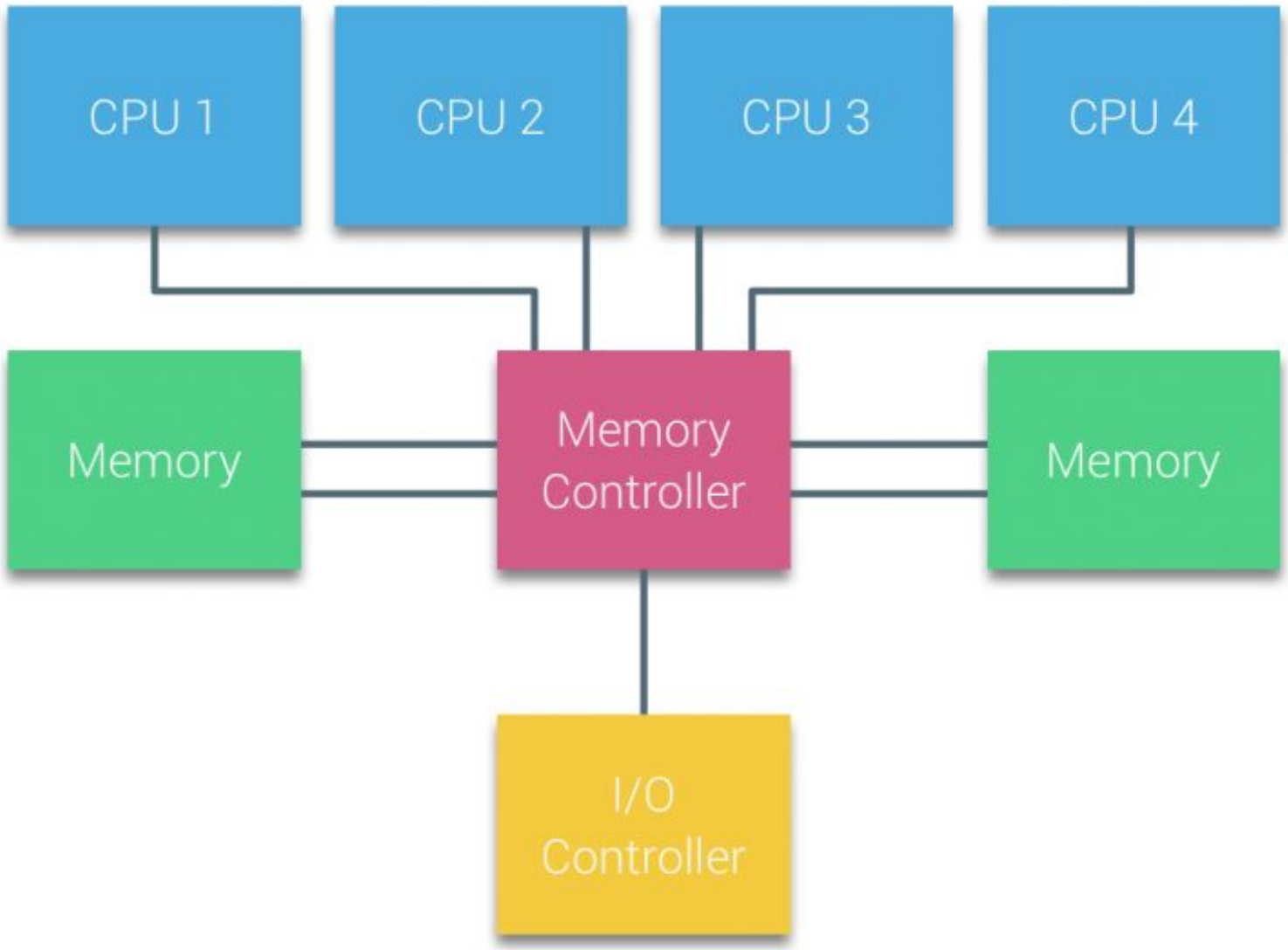
Memory architecture

- Memory is either volatile or non-volatile
- **Primary** memory is **fast** but volatile, **small** and **expensive**
- **Secondary** memory is **cheaper** and **larger** but non-volatile and **slow**
- **CPU** has **no** direct access to secondary memory

Memory architecture

- Memory can basically be accessed using:
  - Random Access Method
  - Sequential Access Method
  - Direct Access Method

Disk Data access



https://frankdenneman.nl/2016/07/07/numa-deep-dive-part-1-uma-numa/

Disk Data access

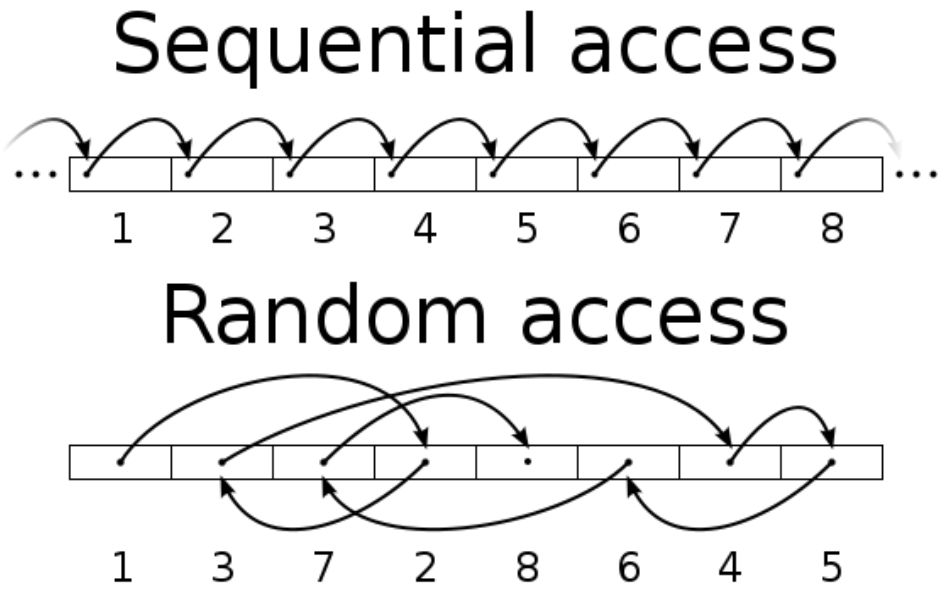
- CPU doesn't have physical access to secondary storage
- Data in disk can only be read/written in blocks
- Most of the systems have 4kB block size
- Slow data manipulation

Disk Data access

- HDD:
  - has moving parts
  - slower by orders of magnitude
  - **random I/O is terrible slow**
- SSD:
  - doesn’t have moving parts
  - random I/O isn’t as bad but still slow

Disk Data access

- Enforcing Sequential I/O will improve performance
  - At Operating System level: less I/O to process
  - At the Storage level: less seek/queueing



Cache

What is it and why you should care?

Cache

- A cache is a hardware or software component that stores data so that future requests for that data can be served faster [1].
- cache hit
- cache miss
- hit rate/ratio

1: https://en.wikipedia.org/wiki/Cache_(computing)

Cache

- Writing policies
  - Write-through
  - Write-back
- Prefetch

1: https://en.wikipedia.org/wiki/Cache_(computing)

Cache Lines and Cache Size

- The **chunks** of memory handled by the cache are the cache lines
- Common cache line sizes are 32, 64 and 128 **bytes**
  - modern x86 CPU usually has **64 bytes** cache line
- A cache can only hold a limited number of lines

Cache Lines and Cache Size

- A 64 kilobyte cache with 64-byte lines has 1024 cache lines
- Accessing L1 cache typically costs 3-5 CPU clock cycles
- Accessing main memory has ~90ns, or ~250 clock cycles latency
- Unaligned data has a higher cost to be processed

Cache Lines and Cache Size

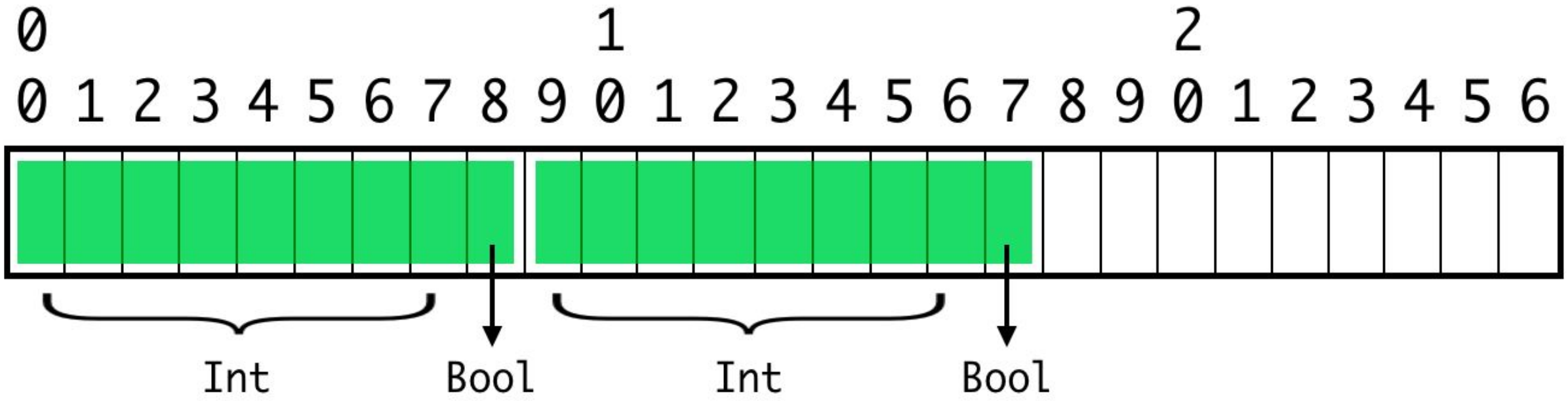
- Let's say we have the below data structure or table:

```
{
        int id,
        bool enable,
        int parent,
        bool valid
}
```

Can you find the issue here?

Cache Lines and Cache Size

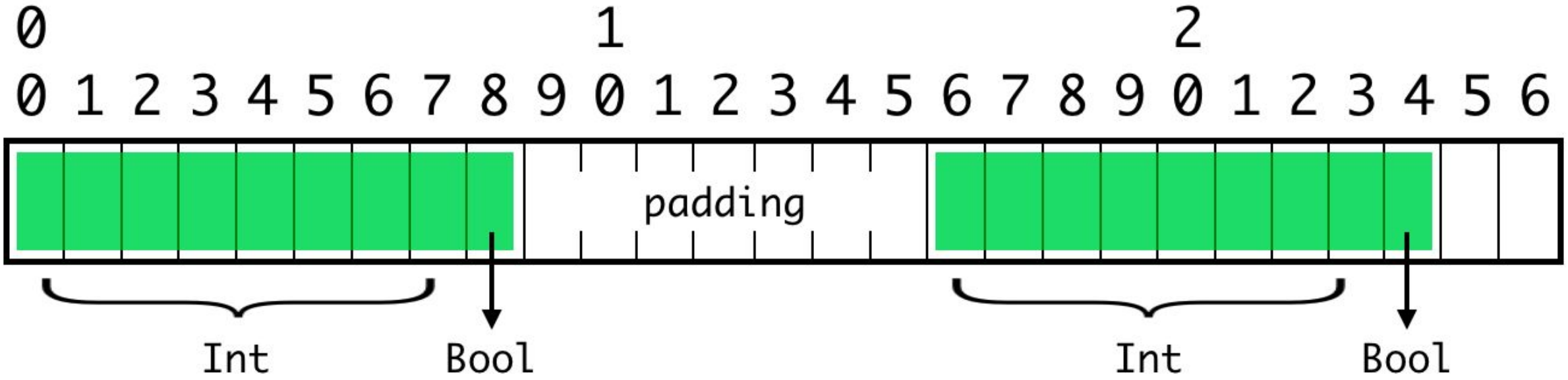
Another chance?



https://swiftunboxed.com/internals/size-stride-alignment/

Cache Lines and Cache Size

- Unaligned data can cause padding
- Padding adds to the cost for the data to be processed



https://swiftunboxed.com/internals/size-stride-alignment/

How PostgreSQL stores data?

Heap files

File Organization

There are many ways to organize files and most common are:

- B+ Tree File Organization
- Clustered File Organization
- Hash File Organization
- Heap File Organization
- ISAM (Indexed Sequential Access Method)
- Sequential File Organization

PostgreSQL uses Heap File Organization

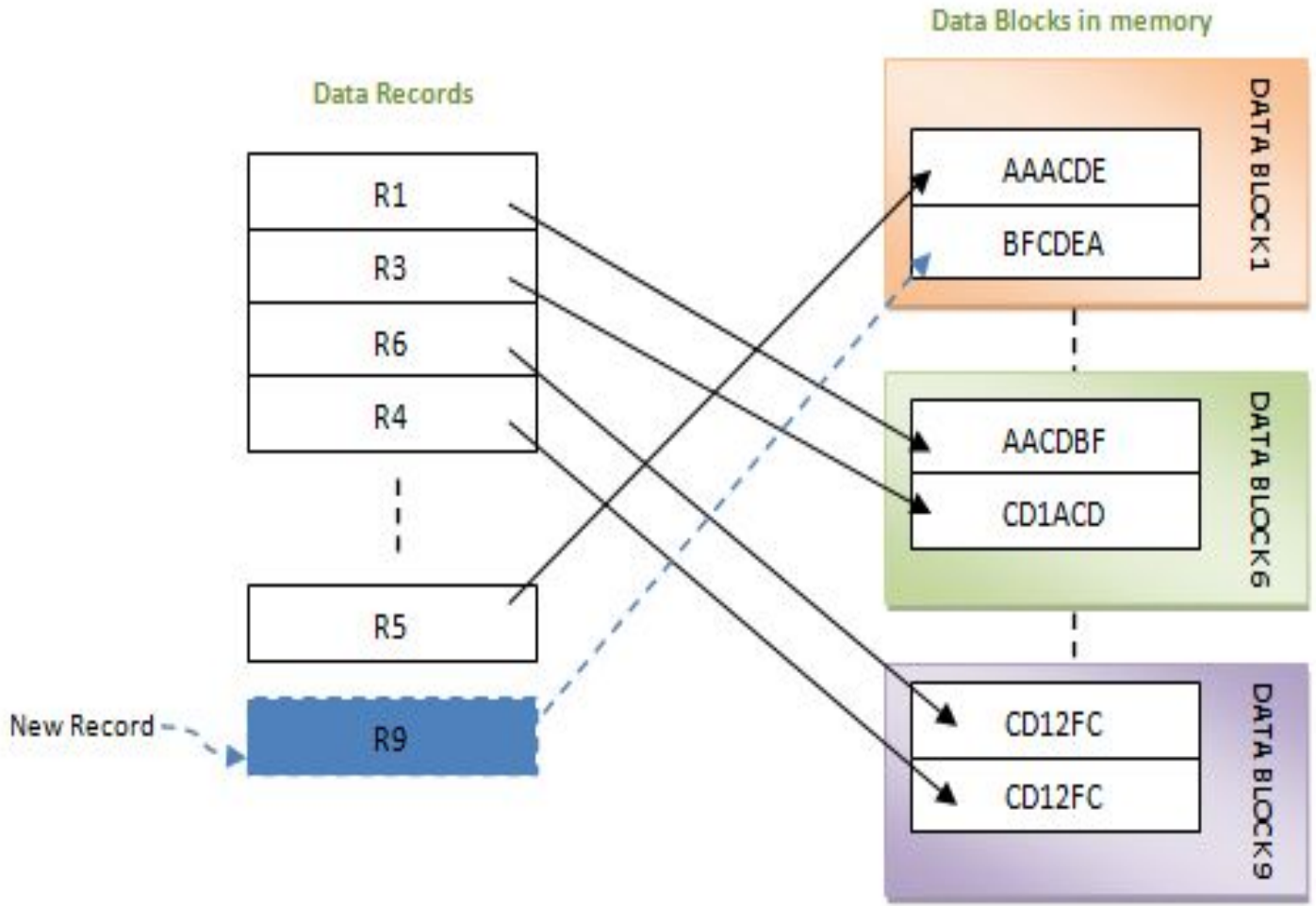
Heap File

- One of the simplest form of file organization
- Unordered set of records stored on pages
- Insert efficient
  - New records are inserted at the end of the file
- No sorting or ordering of the records can be expected

Heap File

- Once the page is full, next record is stored in a new page
- The new page is logically the next closer page
- The new page can be physically located anywhere in the disk
- Deletion is accomplished by marking records as "deleted"
- Update is done by: “delete” the old record and insert the new one

Heap File



https://www.tutorialcup.com/dbms/heap-file-organization.htm

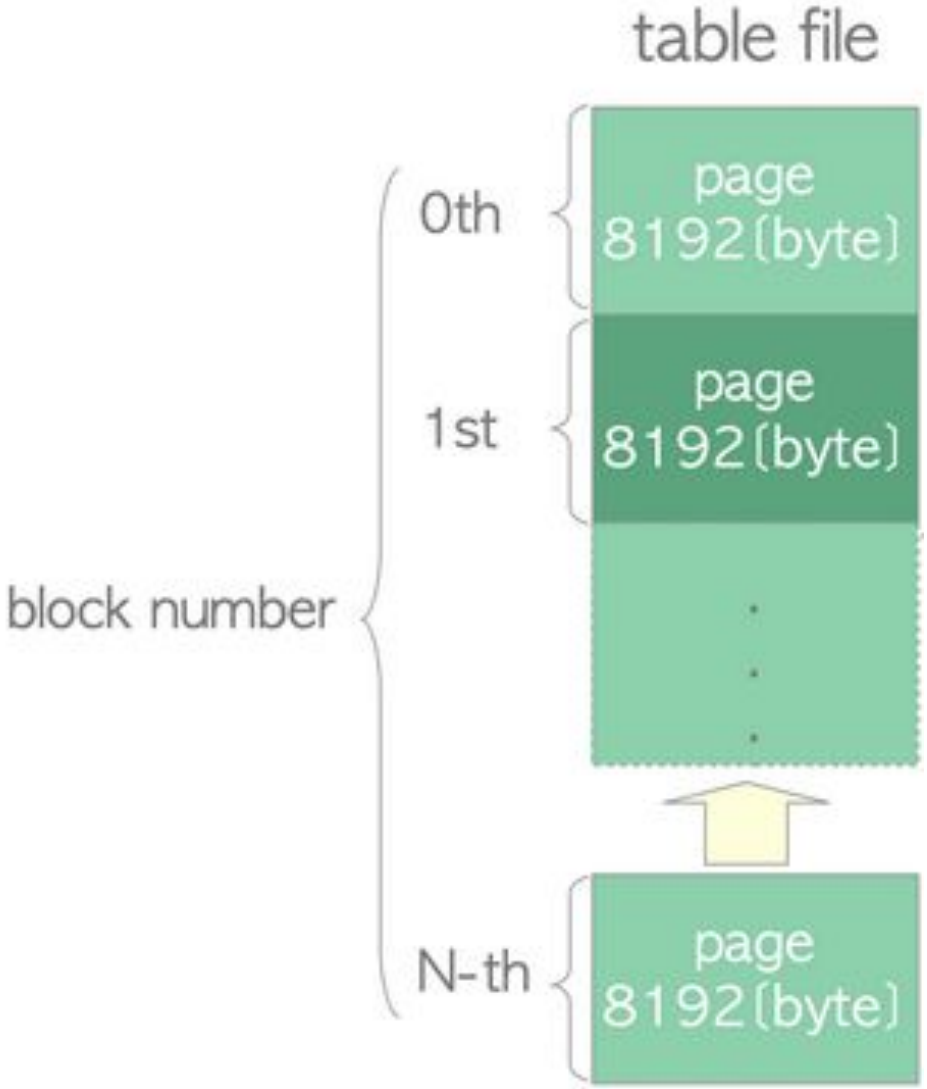
Heap File in Postgres

- The tables are heap files
- Each heap file has a limit of 1GB
- Meaning that:
  - Each table has a primary heap disk file
  - When growing more than 1GB other files are created

Heap File in Postgres

- It’s divided into pages (or blocks) of fixed length
- The default page size is 8 KB
  - It can only be changed at compilation time
- In a table, all the pages are logically equivalent
- A row can be stored in any page

Heap File in Postgres



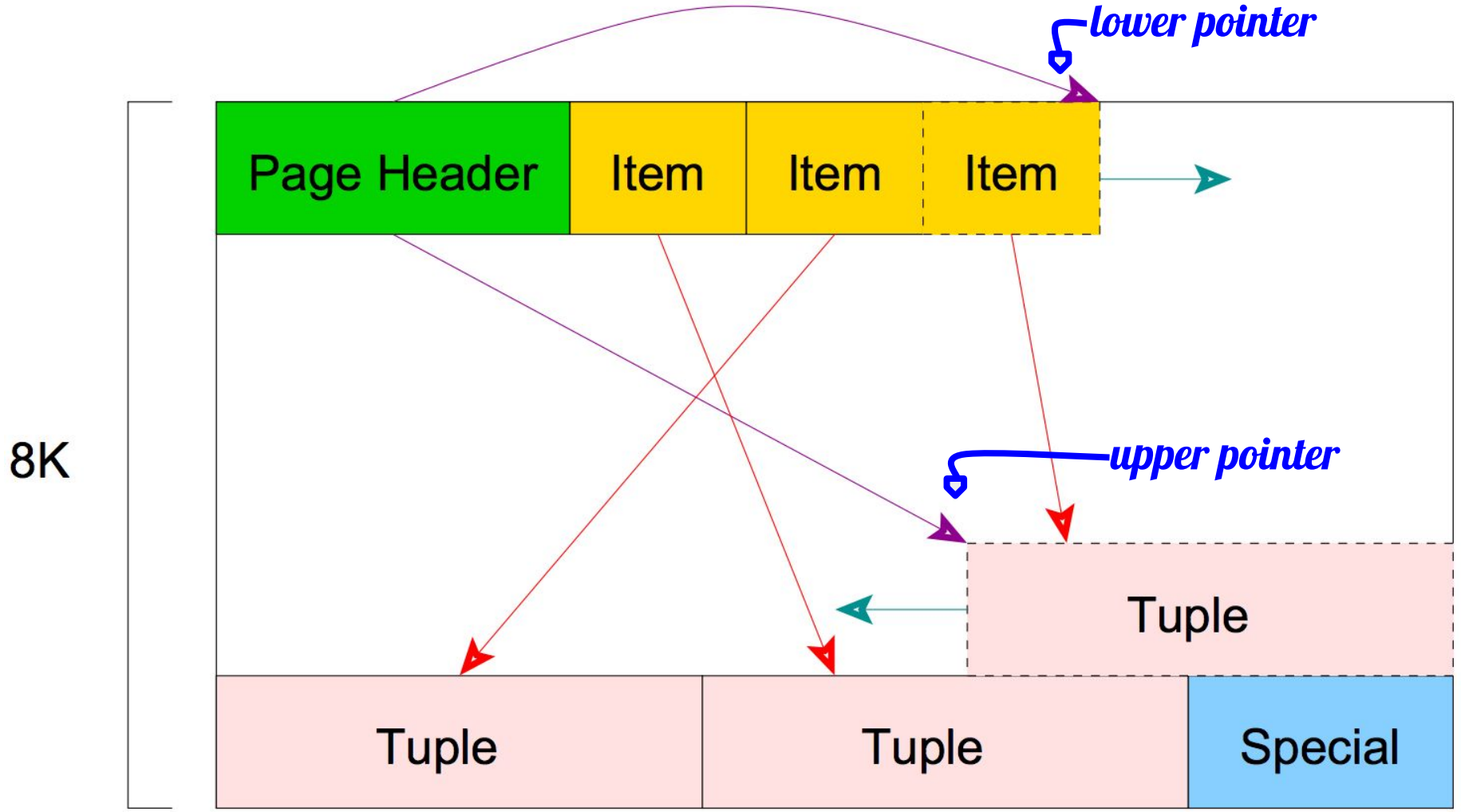
Page Layout

- A page is divided into:
  - PageHeaderData: The first 24 bytes of each page is a page header
  - ItemIdData: Array of item identifiers (line pointer) pointing to the actual items

Page Layout

- Free space: The unallocated space used for new ItemIdData and new Items
- Items or heap tuple: The actual items (rows) themselves
- Special space: Holds index access method specific data.
  - Empty in ordinary tables

Page Layout



https://stackoverflow.com/questions/59861645/postgres-and-tables-internal-organization

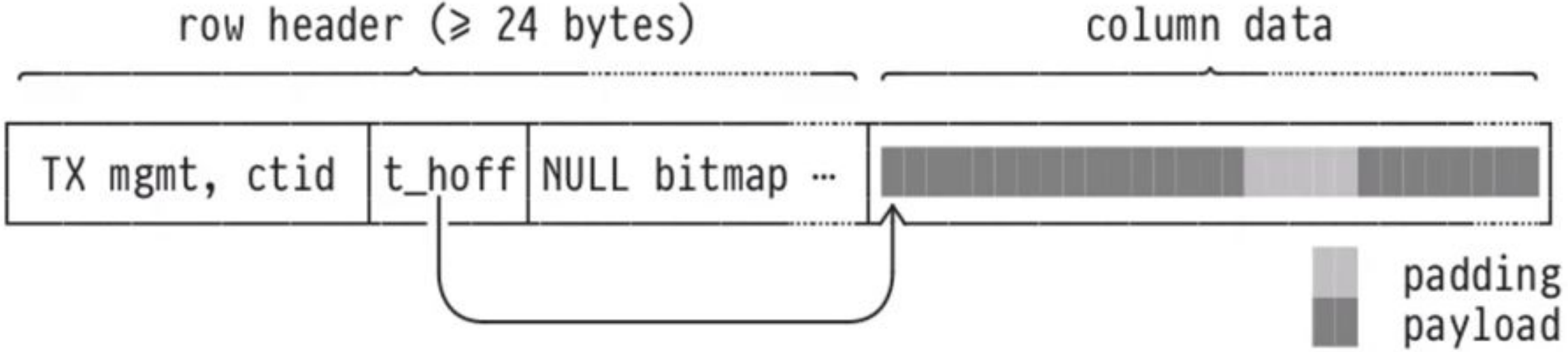
Table Row Layout

- All table rows are structured in the same way:
  - Fixed-size header (23 bytes on most machines), followed by an optional null bitmap
  - An optional object ID field
  - The user data
  - The user data begins at the offset indicated by t_hoff field in the header

Table Row Layout

- The value of t_hoff must always be a multiple of the MAXALIGN distance for the platform
- The field t_infomask in the header defines if the null bitmap is present
- If the null bitmap is present it begins just after the fixed header
- The null bitmap occupies enough bytes to have one bit per data column
- When the bitmap is not present, all columns are assumed not-null

Table Row Layout



Torsten Grust
Universitat Tubingen, Germany

TOAST

The Oversized-Attribute Storage Technique

- PostgreSQL uses a fixed page size (commonly 8 kB)
- PostgreSQL does not allow tuples to span multiple pages
- Large field values are stored outside of the heap table in separated files
- They are compressed and broken up into multiple physical rows outside

TOAST

- The technique is affectionately known as TOAST
- Not all data types support TOAST
- Each table that is created has its own associated (unique)

  TOAST table

TOAST

- How does it work?
  - When a row is "too large" (> 2KB by default), the TOAST mechanism attempts to compress any wide field values;
  - If that isn't enough to get the row under 2KB, it breaks up the wide field values into chunks that get stored in the associated TOAST table;
  - Each original field value is replaced by a small pointer that shows where to find this "out of line" data in the TOAST table;

Data Alignment and Padding

- To efficiently performs read/write to memory, the CPU needs aligned data
- Postgres is designed to have an internal natural alignment of 8 bytes
- Every data type in PostgreSQL has a specific alignment requirement

Data Alignment and Padding

- The typalign attribute in pg_type describes the required alignments:
  - c = char alignment, i.e., no alignment needed
  - s = short alignment (2 bytes on most machines)
  - i = int alignment (4 bytes on most machines)
  - d = double alignment (8 bytes on many machines, but by no means all)

Data Alignment and Padding

- Consecutive fixed-length columns of differing size may need be padded with empty bytes
- **It is possible to define table columns in an order that minimizes padding**

Reference: https://www.postgresql.org/docs/current/catalog-pg-type.html

Data Alignment and Padding

Say we have a table with the below structure

```
CREATE TABLE t_queue_item_bad (
   item_type int2,
   q_id int8 not null,
   is_active boolean,
   q_item_id int8,
   q_item_value numeric,
   q_item_parent int8
);
```

We then insert 1M rows:

```
INSERT INTO t_queue_item_bad
   SELECT
      (random() * 125)::int,          -- item_type
      (random() * 99999)::int,        -- q_id
      ((random() * 999)::int % 2 = 0), -- is_active
      i,                              -- q_item_id
      (random() * 999)::int,          -- q_item_value
      (random() * 999)::int           -- q_item_parent
   FROM generate_series(1, 1000000) AS i;
```

We then create another table, same structure, different column order

```
CREATE TABLE t_queue_item_good AS
   SELECT q_id,
          q_item_id,
          q_item_parent,
          item_type,
          is_active,
          q_item_value
   FROM t_queue_item_bad;
```

Note how the fields are organized...

```
test=# SELECT a.attname, t.typname, t.typalign, t.typlen
test-#   FROM pg_class c
test-#   JOIN pg_attribute a ON (a.attrelid = c.oid)
test-#   JOIN pg_type t ON (t.oid = a.atttypid)
test-#  WHERE c.relname like 't_queue_item_bad'
test-#    AND a.attnum >= 0
test-#  ORDER BY a.attnum;
    attname    | typname | typalign | typlen
---------------+---------+----------+--------
 item_type     | int2    | s        |      2
 q_id          | int8    | d        |      8
 is_active     | bool    | c        |      1
 q_item_id     | int8    | d        |      8
 q_item_value  | numeric | i        |     -1
 q_item_parent | int8    | d        |      8
(6 rows)
```

The size difference is over 25% in this example!!

```
test=# SELECT relname,
test-#     pg_size_pretty(pg_relation_size(relname::TEXT)) AS size
test-# FROM pg_class
test-# WHERE relname LIKE 't_queue_item%';
      relname      | size
-------------------+-------
 t_queue_item_bad  | 73 MB
 t_queue_item_good | 57 MB
(2 rows)
```

What are the implications?

Sometimes even tiny changes can make a huge impact

What are the implications?

```
fosdem=# \dS+ public.order_line10
                                                Table "public.order_line10"
     Column     |            Type             | Collation | Nullable | Default | Storage  |
----------------+-----------------------------+-----------+----------+---------+----------+-
 ol_o_id        | integer                     |           | not null |         | plain    |
 ol_d_id        | smallint                    |           | not null |         | plain    |
 ol_w_id        | smallint                    |           | not null |         | plain    |
 ol_number      | smallint                    |           | not null |         | plain    |
 ol_i_id        | integer                     |           |          |         | plain    |
 ol_supply_w_id | smallint                    |           |          |         | plain    |
 ol_delivery_d  | timestamp without time zone |           |          |         | plain    |
 ol_quantity    | smallint                    |           |          |         | plain    |
 ol_amount      | numeric(6,2)                |           |          |         | main     |
 ol_dist_info   | character(24)               |           |          |         | extended |
Indexes:
    "order_line10_pkey" PRIMARY KEY, btree (ol_w_id, ol_d_id, ol_o_id, ol_number)
    "fkey_order_line_210" btree (ol_supply_w_id, ol_i_id)
Access method: heap
```

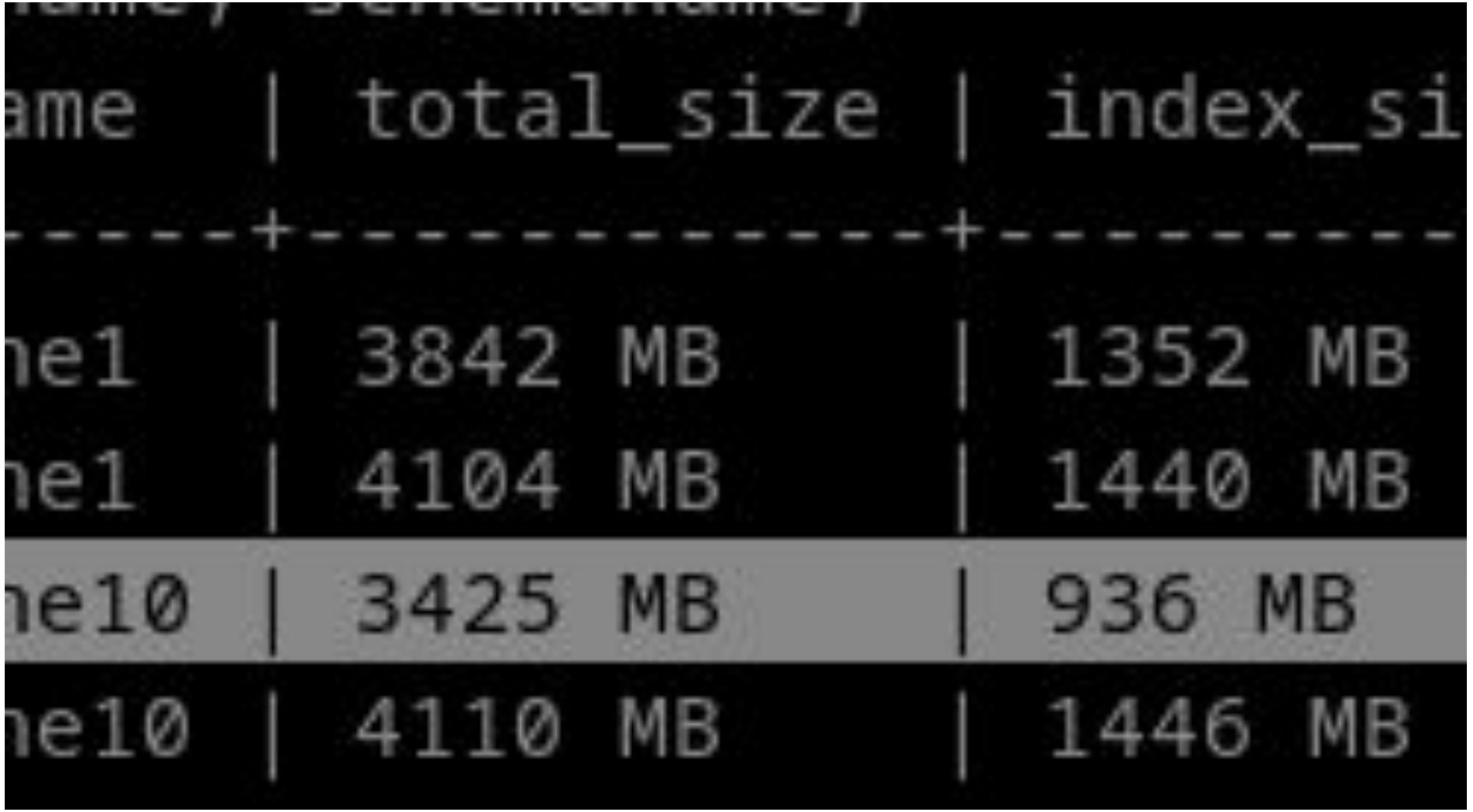
What are the implications?

```
fosdem=# \dS+ new.order_line10
                                                Table "new.order_line10"
    Column     |            Type             | Collation | Nullable | Default | Storage  |
---------------+-----------------------------+-----------+----------+---------+----------+-
 ol_o_id       | integer                     |           | not null |         | plain    |
 ol_i_id       | integer                     |           |          |         | plain    |
 ol_d_id       | smallint                    |           | not null |         | plain    |
 ol_w_id       | smallint                    |           | not null |         | plain    |
 ol_number     | smallint                    |           | not null |         | plain    |
 ol_supply_w_id | smallint                   |           |          |         | plain    |
 ol_delivery_d | timestamp without time zone |           |          |         | plain    |
 ol_quantity   | smallint                    |           |          |         | plain    |
 ol_amount     | numeric(6,2)                |           |          |         | main     |
 ol_dist_info  | character(24)               |           |          |         | extended |
Indexes:
    "order_line10_pkey" PRIMARY KEY, btree (ol_w_id, ol_d_id, ol_o_id, ol_number)
    "new_fkey_order_line_210" btree (ol_supply_w_id, ol_i_id)
Access method: heap
```

What are the implications?

```
fosdem-#     pg_size_pretty(pg_indexes_size(relid)) as index_size
fosdem-# from pg_catalog.pg_statio_user_tables
fosdem-# where relname like 'order_line%'
fosdem-# order by table_name, schemaname;
 table_schema |  table_name  | total_size | data_size | index_size
--------------+--------------+------------+-----------+------------
 new          | order_line1  | 3842 MB    | 2489 MB   | 1352 MB
 public       | order_line1  | 4104 MB    | 2663 MB   | 1440 MB
 new          | order_line10 | 3425 MB    | 2489 MB   | 936 MB
 public       | order_line10 | 4110 MB    | 2663 MB   | 1446 MB
 new          | order_line2  | 3842 MB    | 2490 MB   | 1352 MB
 public       | order_line2  | 4109 MB    | 2664 MB   | 1444 MB
 new          | order_line3  | 3841 MB    | 2489 MB   | 1352 MB
 public       | order_line3  | 4102 MB    | 2663 MB   | 1438 MB
 new          | order_line4  | 3842 MB    | 2489 MB   | 1352 MB
 public       | order_line4  | 4108 MB    | 2663 MB   | 1444 MB
 new          | order_line5  | 3843 MB    | 2489 MB   | 1353 MB
 public       | order_line5  | 4100 MB    | 2663 MB   | 1436 MB
 new          | order_line6  | 3426 MB    | 2489 MB   | 936 MB
 public       | order_line6  | 4102 MB    | 2664 MB   | 1438 MB
 new          | order_line7  | 3424 MB    | 2489 MB   | 934 MB
 public       | order_line7  | 4103 MB    | 2663 MB   | 1439 MB
 new          | order_line8  | 3425 MB    | 2489 MB   | 935 MB
 public       | order_line8  | 4103 MB    | 2663 MB   | 1439 MB
 new          | order_line9  | 3427 MB    | 2490 MB   | 936 MB
 public       | order_line9  | 4109 MB    | 2664 MB   | 1444 MB
(20 rows)
```

What are the implications?



ame	total_size	index_si
he1	3842 MB	1352 MB
he1	4104 MB	1440 MB
he10	3425 MB	936 MB
he10	4110 MB	1446 MB

What are the implications?

```
SQL statistics:
    queries performed:
        read:                            970858
        write:                           1008659
        other:                           149002
        total:                           2128519
    transactions:                        74485  (124.12 per sec.)
    queries:                             2128519 (3547.05 per sec.)
    ignored errors:                      296    (0.49 per sec.)
    reconnects:                          0      (0.00 per sec.)

General statistics:
    total time:                          600.0789s
    total number of events:              74485

Latency (ms):
         min:                                    1.19
         avg:                                  128.89
         max:                                 9529.43
         95th percentile:                      344.08
         sum:                                9600707.92

Threads fairness:
    events (avg/stddev):           4655.3125/89.83
    execution time (avg/stddev):   600.0442/0.02
```

What are the implications?

```
SQL statistics:
    queries performed:
        read:                            1052853
        write:                           1091666
        other:                           162552
        total:                           2307071
    transactions:                        81260  (135.38 per sec.)
    queries:                             2307071 (3843.70 per sec.)
    ignored errors:                      334    (0.56 per sec.)
    reconnects:                          0      (0.00 per sec.)

General statistics:
    total time:                          600.2195s
    total number of events:              81260

Latency (ms):
         min:                                    1.17
         avg:                                  118.16
         max:                                 2114.85
         95th percentile:                      292.60
         sum:                                9601333.56

Threads fairness:
    events (avg/stddev):           5078.7500/93.71
    execution time (avg/stddev):   600.0833/0.08
```

What are the implications?

This is what we found in this sysbench TPC-C like test:

- Average 19% disk space reduction
- Average 8.4% overall performance improvement
  - Write 8.2% in avg
  - Reads 8.5% in avg
- Reduction in latency by an average of 15%

Keep in mind that it was a small dataset and only 5min warm

up queries for 10min test each round!

It then comes to the end

Summary

Summary

- Postgres stores its data in heap files
- The file is divided in blocks of 8kB each
- The data has no order
- Deleting a record doesn’t remove it but mark as removed
- Postgres can insert new record in the end of the file or in any free space
- Updating a row does a “delete”+”insert” operation

Summary

Every data type has its alignment requirement and can cause padding!

Questions?

percona.com

THANK YOU!

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

percona.com