PERCONA LIVE ONLINE MAY 12 - 13th 2021

Reducing Costs and Improving Performance With Data Modeling in Postgres

Hello!

I am Charly Batista

A Brazilian Percona engineer who lives in China and is passionate about databases

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

Agenda

- Overview - What is this talk about?
- Let’s review some concepts - HDD, SSD, DRAM and CPU
- Heap files - How PostgreSQL stores data?
- The Free Space Map
- It comes to the end, the Summary
- Thank you!

What is this talk about?

This talk is about how PostgreSQL stores the data and how it relates to our data model, understanding the impacts the types of data and its order in the database size have in the storage cost.

During this talk we will understand:

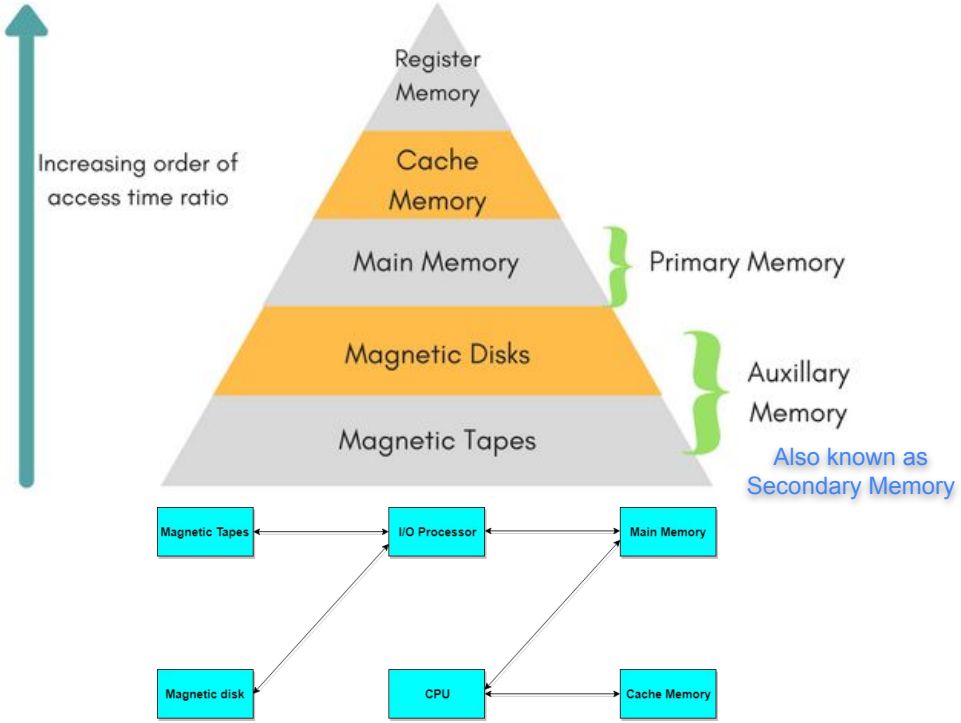
- How Postgres stores data internally
- How the different data types impact in the data size
- How the free space is organized and how vacuum finds blocks to clean up

We will then relate all the above to understand how our data design impacts costs and performance.

Note that we will use use expressions “block” and “page” intertengely with the same meaning during this talk

HDD, SSD, DRAM and CPU

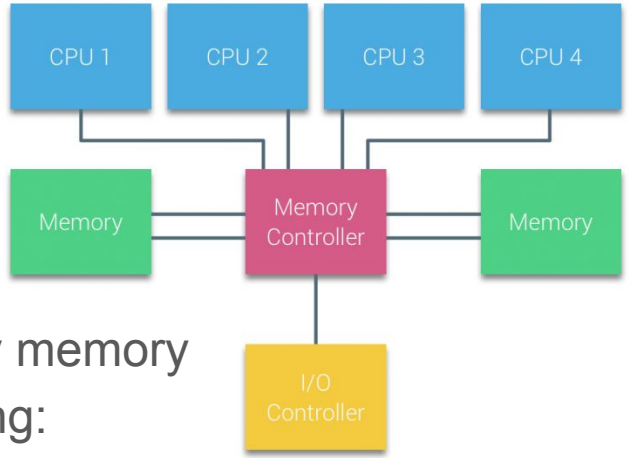
Memory architecture/hierarchy



https://www.studytonight.com/computer-architecture/memory-organization

Memory architecture

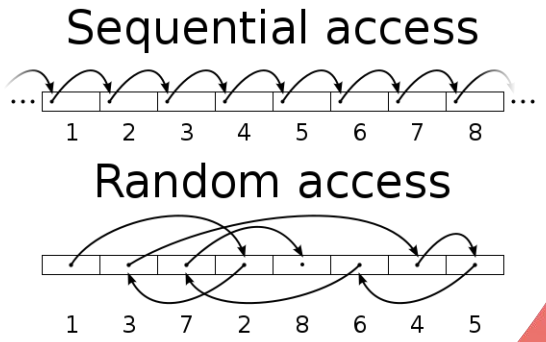
- Memory is either volatile or non-volatile
- **Primary** memory is **volatile**, small, fast and **expensive**
- **Secondary** memory is **non-volatile**, larger, **cheap** and **slow**
- CPU has no direct access to secondary memory
- Memory can basically be accessed using:
  - Random Access Method
  - Sequential Access Method
  - Direct Access Method



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

Disk Data access

- Data in disk can only be read/written in blocks
- Most of the systems have 4kB block size
- **Random access is usually slow by orders of magnitude**
  - HDDs is worse because the extra disk head movement
  - SSDs do not have moving parts but random is still slow here
- Enforcing Sequential I/Os will improve performance
  - At Operating System level: less I/Os to process
  - At the Storage level: less seek/queueing



https://kb.sandisk.com/app/answers/detail/a_id/8150/~/difference-between-sequential-and-random-access-operations

HDD Performance

- Access time is the sum of:
  - seek time + rotational time + transfer time
- Access time:
  - Random access is slower due to seek time
  - Sequential access is faster, almost no seek time

HDD layout	6 platters, 12 r/w reads
Bytes per sector	4096 (4k)
Default sectors per track	63
I/O data-transfer rate (max)	600MB/s
Track-to-track seek time	1ms
Average seek time (6TB and 8TB models)	4ms

https://www.seagate.com/files/www-content/product-content/nas-fam/nas-hdd/en-us/docs/100724684k.pdf

- Random access time for a 8k block (suppose 15k RPM disk):
  - Average seek time(t_S): 4ms
  - Average rotation delay (t_R): 0.5 * (1/15000 min^{-1}) = 2ms
  - Average transfer time(t_T): 8kB / (600 MB/s) = 0.013ms
  - $t_{total8k} = t_S + t_R + t_T$ = 4ms + 2ms + 0.1ms = 6.1ms
- Random access time for 1k blocks: 1000 * $t_{total8k}$ = **6.1s**
- Sequential access time for 1k blocks:
  - 252kB per track will span 32 tracks to store 1k blocks of 8kb;
  - $t_{total} = t_S + t_R + (1000 * t_T) + (32 *$ track-track seek time)
  - t_{total} = 6.1ms + 2ms + (1000 * 0.013)ms + (32 * 1)ms = **53.1ms**

SSD Performance

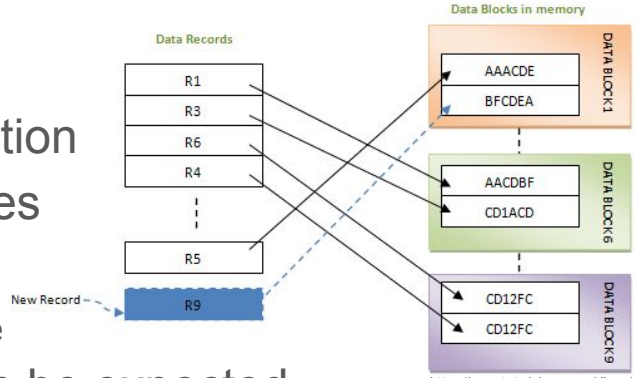
- Random access time for a 8k block:
  - Max performance: 4k block * 480000 IOPS = 1920000 kB/s
  - Transfer time: 8kB / 1920000 kB/s = 0.00417ms
- Random access time for 1k blocks: **4.17ms**
- Sequential access time for 1k blocks:
  - Sequential Read: (8kB * 1000) / 3.4 MB/s = 0.00235ms
- Sequential access time for 1k blocks: **2.35ms**

Interface	PCIe Gen3 8 Gb/s, up to 4 lanes
Sequential Read MB/s up to (Q=32, T=1)	3.4
Sequential Write MB/s up to (Q=32, T=1)	2.9
Random Read 4KB IOPS up to (Q=32, T=8)	480,000
Random Write 4KB IOPS up to (Q=32, T=8)	550,000

https://documents.westerndigital.com/content/dam/doc-library/en_us/assets/public/western-digital/product/internal-drives/wd-black-ssd/data-sheet-wd-black-sn-750-nvme-ssd-heatsink-idk.pdf

Heap files - How PostgreSQL stores data?

Heap File

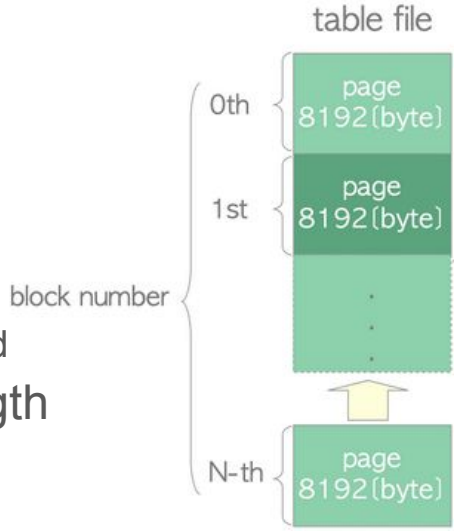


https://www.tutorialcup.com/dbms/heap-file-organization.htm

- 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
- 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

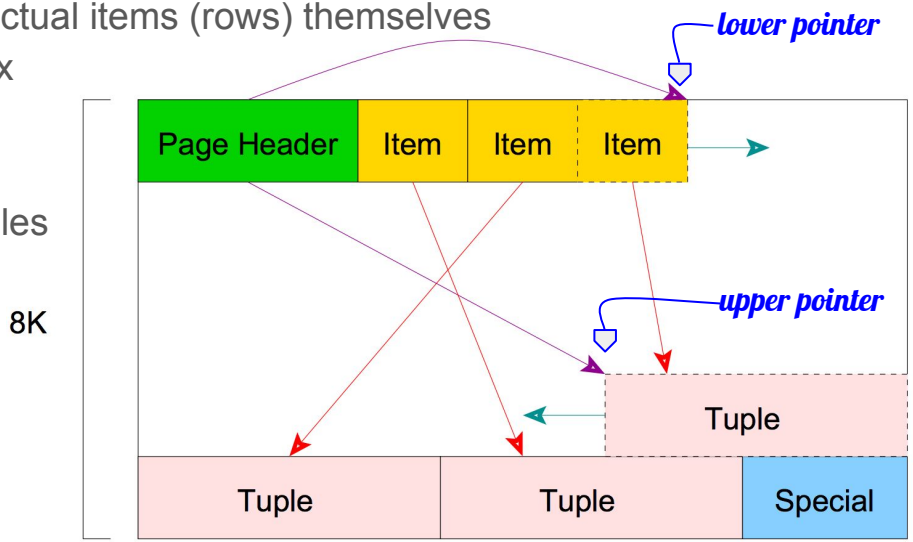
Heap File in Postgres

- 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
- 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



Page Layout

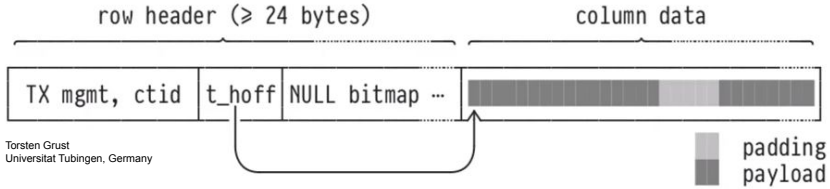
- A page is divided into:
  - PageHeaderData: The first 24 bytes of each page is a page header
  - ItemIdData: Array of item identifiers (line pointer) pointing to the actual items
  - Free space: The unallocated space used for new ItemIdData and new Items
  - Items or heap tuple: The actual items (rows) themselves
  - Special space: Holds index
    access method
    specific data.
    Empty in ordinary tables



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
  - 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



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
- 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
- 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
- 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)*
- 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)
```

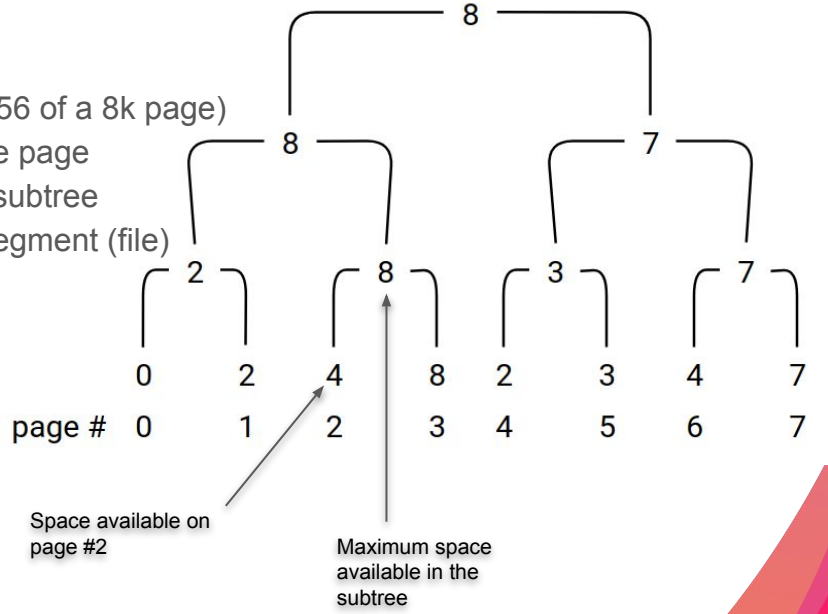
The Free Space Map

Free Space Map

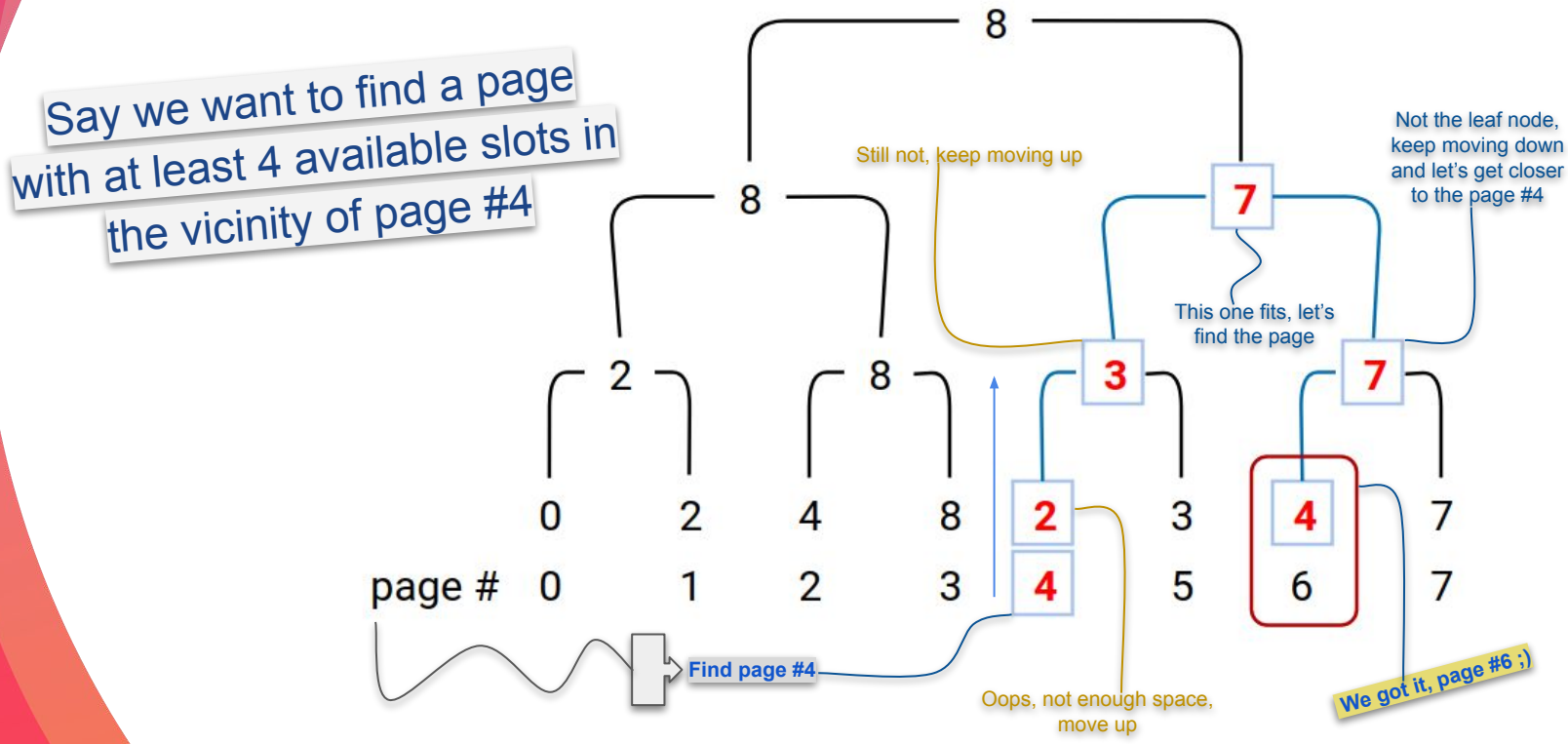
- Row delete or update may lead to holes in the heap file
- “New” records could fill those holes
- Postgres maintains a map to keep track of the available free space
- Used by INSERTs and UPDATEs
- Each heap and index relation has a Free Space Map (FSM)
- Hash index is an exception, it doesn’t have a FSM

Free Space Map Tree

- The FSM file is organized as a tree of FSM pages
- The bottom pages store the free space available on each heap page
- The upper levels aggregate information from the lower levels
- Summary is:
  - One byte per node
  - Space measured in 32 byte units (1/256 of a 8k page)
  - Leaf node: space available in heap file page
  - Inner nodes: max space found in the subtree
  - Root node: max space found in this segment (file)



Free Space Map Tree



Summary

Summary

- Postgres stores its data in heap files
- The file is divided in blocks of 8kB each
- The data has no order
- Postgres controls the free space of each block in a map file (FSM)
- It uses a bitmap file to exposure block visibility
- 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
- It uses a natural alignment of 8 bytes internally
- Every data type has its alignment requirement and can cause padding
- You can find this slides and the scripts on my github at the end

Thanks!

Any questions?

You can find me at:

- https://www.linkedin.com/in/charlybatista
- charly.batista@percona.com
- https://github.com/elchinoo/pl21

THANK YOU !

PERCONA
LIVEONLINE
MAY 12 - 13th
2021