Reducing Costs and Improving Performance With Data Modeling in Postgres

Charly Batista
Who am I?

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

Blame ChatGPT for this introduction!
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

- CPU Registers
- CPU cache
- Main memory
- Secondary storage

Volatile:
- Faster access
- Lower capacity

Non-Volatile:
- Higher capacity
- Higher latency
- Slower access time
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:

```c
{
    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

[Diagram showing the layout of a page with sections labeled Page Header, Item, Tuple, and Special, with pointers indicating upper and lower connections.]

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

row header (≥ 24 bytes)

TX mgmt, ctid  t_hoff  NULL bitmap ...

column data

padding

payload
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 to 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
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,  
);  
```

Note how the fields are organized...

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;  
```

The size difference is over 25% in this example!!
What are the implications?

Sometimes even tiny changes can make a huge impact.
What are the implications?

```sql
fosdem=# \d+ public.order_line10

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Collation</th>
<th>Nullable</th>
<th>Default</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ol_o_id</td>
<td>integer</td>
<td></td>
<td>not null</td>
<td></td>
<td>plain</td>
</tr>
<tr>
<td>ol_d_id</td>
<td>smallint</td>
<td></td>
<td>not null</td>
<td></td>
<td>plain</td>
</tr>
<tr>
<td>ol_w_id</td>
<td>smallint</td>
<td></td>
<td>not null</td>
<td></td>
<td>plain</td>
</tr>
<tr>
<td>ol_number</td>
<td>smallint</td>
<td></td>
<td>not null</td>
<td></td>
<td>plain</td>
</tr>
<tr>
<td>ol_i_id</td>
<td>integer</td>
<td></td>
<td></td>
<td></td>
<td>plain</td>
</tr>
<tr>
<td>ol_supply_w_id</td>
<td>smallint</td>
<td></td>
<td></td>
<td></td>
<td>plain</td>
</tr>
<tr>
<td>ol_delivery_d</td>
<td>timestamp without time zone</td>
<td></td>
<td></td>
<td></td>
<td>plain</td>
</tr>
<tr>
<td>ol_quantity</td>
<td>smallint</td>
<td></td>
<td></td>
<td></td>
<td>plain</td>
</tr>
<tr>
<td>ol_amount</td>
<td>numeric(6,2)</td>
<td></td>
<td></td>
<td></td>
<td>main</td>
</tr>
<tr>
<td>ol_dist_info</td>
<td>character(24)</td>
<td></td>
<td></td>
<td></td>
<td>extended</td>
</tr>
</tbody>
</table>

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?

```plaintext
fosdem=# \d+ new.order_line10

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Collation</th>
<th>Nullable</th>
<th>Default</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ol_o_id</td>
<td>integer</td>
<td></td>
<td>not null</td>
<td></td>
<td>plain</td>
</tr>
<tr>
<td>ol_i_id</td>
<td>integer</td>
<td></td>
<td>not null</td>
<td></td>
<td>plain</td>
</tr>
<tr>
<td>ol_d_id</td>
<td>smallint</td>
<td></td>
<td>not null</td>
<td></td>
<td>plain</td>
</tr>
<tr>
<td>ol_w_id</td>
<td>smallint</td>
<td></td>
<td>not null</td>
<td></td>
<td>plain</td>
</tr>
<tr>
<td>ol_number</td>
<td>smallint</td>
<td></td>
<td>not null</td>
<td></td>
<td>plain</td>
</tr>
<tr>
<td>ol_supply_w_id</td>
<td>smallint</td>
<td></td>
<td></td>
<td></td>
<td>plain</td>
</tr>
<tr>
<td>ol_delivery_d</td>
<td>timestamp without time zone</td>
<td></td>
<td></td>
<td></td>
<td>plain</td>
</tr>
<tr>
<td>ol_quantity</td>
<td>smallint</td>
<td></td>
<td></td>
<td></td>
<td>plain</td>
</tr>
<tr>
<td>ol_amount</td>
<td>numeric(6,2)</td>
<td></td>
<td></td>
<td></td>
<td>main</td>
</tr>
<tr>
<td>ol_dist_info</td>
<td>character(24)</td>
<td></td>
<td></td>
<td></td>
<td>extended</td>
</tr>
</tbody>
</table>

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?

```sql
select table_schema, table_name, total_size, data_size, index_size
from pg_catalog.pg_statio_user_tables
where relname like 'order_line%'
order by table_name, schemaname;
```

<table>
<thead>
<tr>
<th>table_schema</th>
<th>table_name</th>
<th>total_size</th>
<th>data_size</th>
<th>index_size</th>
</tr>
</thead>
<tbody>
<tr>
<td>new</td>
<td>order_line1</td>
<td>3842 MB</td>
<td>2489 MB</td>
<td>1352 MB</td>
</tr>
<tr>
<td>public</td>
<td>order_line1</td>
<td>4104 MB</td>
<td>2663 MB</td>
<td>1440 MB</td>
</tr>
<tr>
<td>new</td>
<td>order_line10</td>
<td>3425 MB</td>
<td>2489 MB</td>
<td>936 MB</td>
</tr>
<tr>
<td>public</td>
<td>order_line10</td>
<td>4110 MB</td>
<td>2663 MB</td>
<td>1446 MB</td>
</tr>
<tr>
<td>new</td>
<td>order_line2</td>
<td>3842 MB</td>
<td>2490 MB</td>
<td>1352 MB</td>
</tr>
<tr>
<td>public</td>
<td>order_line2</td>
<td>4109 MB</td>
<td>2664 MB</td>
<td>1444 MB</td>
</tr>
<tr>
<td>new</td>
<td>order_line3</td>
<td>3841 MB</td>
<td>2489 MB</td>
<td>1352 MB</td>
</tr>
<tr>
<td>public</td>
<td>order_line3</td>
<td>4102 MB</td>
<td>2663 MB</td>
<td>1438 MB</td>
</tr>
<tr>
<td>new</td>
<td>order_line4</td>
<td>3842 MB</td>
<td>2489 MB</td>
<td>1352 MB</td>
</tr>
<tr>
<td>public</td>
<td>order_line4</td>
<td>4108 MB</td>
<td>2663 MB</td>
<td>1444 MB</td>
</tr>
<tr>
<td>new</td>
<td>order_line5</td>
<td>3843 MB</td>
<td>2489 MB</td>
<td>1353 MB</td>
</tr>
<tr>
<td>public</td>
<td>order_line5</td>
<td>4100 MB</td>
<td>2663 MB</td>
<td>1436 MB</td>
</tr>
<tr>
<td>new</td>
<td>order_line6</td>
<td>3426 MB</td>
<td>2489 MB</td>
<td>936 MB</td>
</tr>
<tr>
<td>public</td>
<td>order_line6</td>
<td>4102 MB</td>
<td>2664 MB</td>
<td>1438 MB</td>
</tr>
<tr>
<td>new</td>
<td>order_line7</td>
<td>3424 MB</td>
<td>2489 MB</td>
<td>934 MB</td>
</tr>
<tr>
<td>public</td>
<td>order_line7</td>
<td>4103 MB</td>
<td>2663 MB</td>
<td>1439 MB</td>
</tr>
<tr>
<td>new</td>
<td>order_line8</td>
<td>3425 MB</td>
<td>2489 MB</td>
<td>935 MB</td>
</tr>
<tr>
<td>public</td>
<td>order_line8</td>
<td>4103 MB</td>
<td>2663 MB</td>
<td>1439 MB</td>
</tr>
<tr>
<td>new</td>
<td>order_line9</td>
<td>3427 MB</td>
<td>2490 MB</td>
<td>936 MB</td>
</tr>
<tr>
<td>public</td>
<td>order_line9</td>
<td>4109 MB</td>
<td>2664 MB</td>
<td>1444 MB</td>
</tr>
</tbody>
</table>

(20 rows)
What are the implications?

<table>
<thead>
<tr>
<th>name</th>
<th>total_size</th>
<th>index_size</th>
</tr>
</thead>
<tbody>
<tr>
<td>me1</td>
<td>3842 MB</td>
<td>1352 MB</td>
</tr>
<tr>
<td>me1</td>
<td>4104 MB</td>
<td>1440 MB</td>
</tr>
<tr>
<td>me10</td>
<td>3425 MB</td>
<td>936 MB</td>
</tr>
<tr>
<td>me10</td>
<td>4110 MB</td>
<td>1446 MB</td>
</tr>
</tbody>
</table>
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 (0.55 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