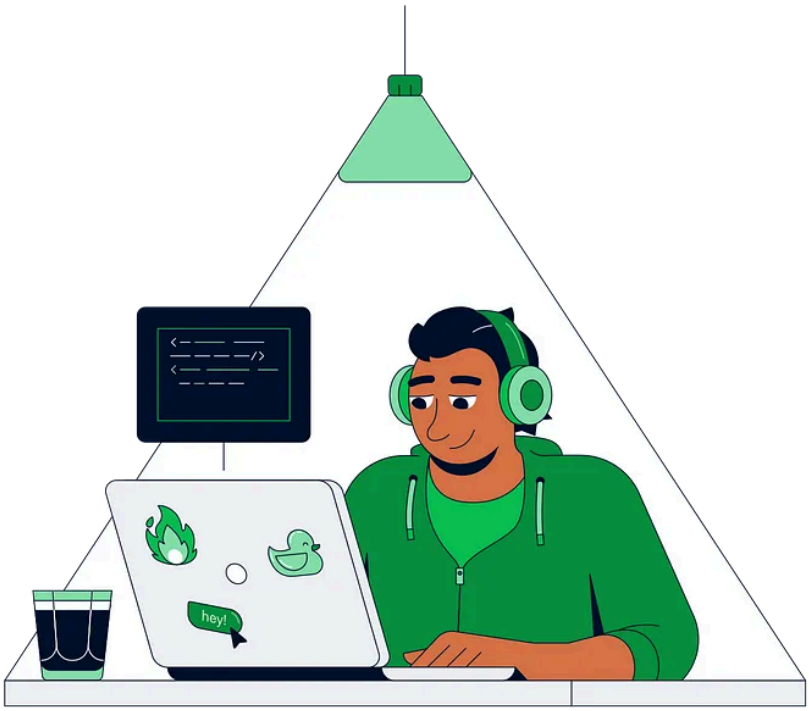
Adyen Tech · Follow publication

Database corruption in PostgreSQL: our journey to improving our upgrade process

by Cagri Biroglu & Derk van Veen, Database Engineers, Adyen

Adyen · Follow

Published in Adyen Tech
19 min read · Jan 25, 2025



PostgreSQL is a core component of Adyen's payment platform, enabling high-throughput, low-latency transaction processing on a global scale. Its stability, scalability, and extensibility make it a critical part of our infrastructure, supporting the reliability required for financial operations.

A few years ago, we upgraded our PostgreSQL fleet from version 9.6 to 13. Upgrades are always long and tedious, but we prepared well for them, and the upgrade on even the hardest clusters with an extremely high number of TPS(transactions per

issue but didn't have much time, as transaction wraparound is always just around the corner in this environment. We had some quiet weeks, where no error was seen, and we thought that whatever it was, it was now behind us. We were wrong.

TOAST storage

The errors became a frequent nuisance, and we had no other option but to figure out what was happening. As it turned out, we had an issue with corrupted *TOAST* data entered in a table in the two weeks before we did the upgrade from 9.6 to 13 on this table's cluster. After the first analysis, we realized:

- This corruption affected millions of rows.
- We couldn't restore this table from a backup before the upgrade.
- If possible, we want to regain access to our data.
- We only knew about this corruption because of a clean-up job. It was unclear how many other tables were affected by the same issue.
- TOAST storage is not something we have understood and mastered yet.
- Transaction logic on TOAST is even more complicated than transaction logic on main tables.

This is the start of a journey that will last almost a year.

Background on TOAST storage

Let's start with some background on TOAST data, as the corruptions are in the externally stored *TOAST*-able data. . Feel free to skip this section if you already know all about TOAST.

TOAST stands for: **T**he **O**versized-**A**ttribute **S**torage **T**echnique. From the documentation:

> *"PostgreSQL uses a fixed page size (commonly 8 kB), and does not allow tuples to span multiple pages. Therefore, it is not possible to store very large field values directly. To overcome this limitation, large field values are compressed and/or broken up into multiple physical rows. This happens transparently to the user, with only a small impact on most of the backend code. The technique is affectionately known as TOAST (or "the best thing since sliced bread")."*

If you would like to understand more about TOAST, and how to optimize TOAST performance, we recommend this blog post — PostgreSQL: TOAST compression and toast_tuple_target by Adrien Nayrat. For this article, we'll use a small example table and explain how TOAST data is referenced internally; this is necessary for understanding the data corruption, and to be honest, that is already complicated enough.

External TOAST data is stored in a special table, referenced by the `reltoastrelid` column in `pg_class`. Data in the TOAST table is split into multiple chunks when required. To retrieve all data from the TOAST table, you need a `chunk_id` to retrieve all the parts that make up the total amount of data. Let's create an easy example to show how this works. First, we create a table with four rows of different lengths. To make it easy, we'll start each row with a specification of its length.

```
create table test_toast(stuff varchar);
insert into test_toast values ('short string');

insert into test_toast (
SELECT '800 length string: ' ||
array_to_string(array(select
substr('ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'
,((random()*(37-1)+1)::integer),1)
from generate_series(1,800 - 19)),''));

insert into test_toast (
SELECT '2500 length string: ' ||
array_to_string(array(select
substr('ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'
,((random()*(37-1)+1)::integer),1)
from generate_series(1,2500 - 20)),''));

insert into test_toast (
SELECT '5000 length string: ' ||
array_to_string(array(select
substr('ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'
```

```
,((random()*(37-1)+1)::integer),1)
from generate_series(1,5000 - 20)),''));
```

```
select ctid, octet_length(stuff) from test_toast;

 ctid  | octet_length
-------+--------------
 (0,1) |           12
 (0,2) |          800
 (0,3) |         2500
 (0,4) |         5000
```

Now, let's see whether there is any data in the TOAST table. First, we need to find the toast table itself.

```
select
reltoastrelid::regclass::text
from pg_class
where relname = 'test_toast';

       reltoastrelid
---------------------------
 pg_toast.pg_toast_1216051

# And then we can check the content of this table
```

And then we can check the content of this table:

```
select
chunk_id,
chunk_seq,
octet_length(chunk_data)
from pg_toast.pg_toast_1216051;

 chunk_id | chunk_seq | octet_length
----------+-----------+--------------
  1216062 |         0 |         1996
  1216062 |         1 |          504
  1216063 |         0 |         1996
```

```
1216063 |          1 |          1996
1216063 |          2 |          1008
```

The TOAST table contains data for two rows of the main table. The total length of the stored data is 2500 characters for the first row and 5000 for the second. As this is a simple example, it is easy to know where this TOAST data belongs. It's more complex in reality, though, so we need to connect the dots.

A `chunk_id` references TOAST data. To find the TOAST data for a row, we only have to get the `chunk_id` and check the TOAST table. This was our first major complication: how does one get the chunk_id when it is not one of the hidden system columns? The `chunk_id` is stored in the raw tuple data. To get the `chunk_id` for a row, you must get the help of the `page_inspect` extension and understand its internal structure. Only then will you be rewarded with the `chunk_id` . Let's dive in!

```
select
lp,
t_attrs
from heap_page_item_attrs(
get_raw_page('public.test_toast', 0), 'public.test_toast') ;

 lp |  t_attrs
————+————————————————————————————————————————
  1 | {"\\x1b73686f727420737472696e67"}
  2 | {"\\x900c0000383030206c656e67746820735….."}
  3 | {"\\x0112c8090000c40900003e8e1200368e1200"}
  4 | {"\\x01128c130000881300003f8e1200368e1200"}
```

For this article, we will only look at the final row; we know that it contains the externally stored data. If you are interested in the full explanation of tuple descriptors, refer to this blog post by Bertrand Drouvot.

Our tuple header is "\\x01128c130000881300003f8e1200368e1200", which contains the following information:
01: is a magical constant (used only for toast pointers)
12: tag (hex value for decimal 18 which is VARTAG_ONDISK)
8c130000: original length
88130000: stored length

3f8e1200: chunk_id
368e1200: toast_relid

Use the following query to convert the `chunk_id` from hex to a big int, remember to revert the hex order (interpret the bytes as little endian):

```
SELECT ('x' || lpad(hex, 16, '0'))::bit(64)::bigint AS chunk_id
FROM  (
   VALUES
      ('00128e3f')
    ) t(hex);

 chunk_id
----------
  1216063
```

The same query can be applied to the other hex values to convert them into integers. The `toast_relid` will translate into the `oid` of 1216054, which can be used to check the TOAST table's name.

Now that we know what part of the tuple header we need, we can extract all the `chunk_ids` for all rows in our main table.

```
select
lp,
(‘x‘||regexp_replace(substr(
substr(page_item_attrs.t_attrs[1],
octet_length(page_item_attrs.t_attrs[1])-7,4)::text
,3),
’(\w\w)(\w\w)(\w\w)(\w\w)’,‘\4\3\2\1‘)
)::bit(32)::bigint as chunk_id
from
heap_page_item_attrs(
get_raw_page('public.test_toast',0),
'public.test_toast') as page_item_attrs ;
```

```
 lp |  chunk_id
----+------------
  1 | 1953702004
  2 |  928403784
```

```
  3 |    1216062
  4 |    1216063
```

Even though these are all valid integer numbers, not all data was stored outside the main table; hence, not every row contains a pointer. As you probably remember, the TOAST table only contains two unique `chunk_ids`.

The next step is distinguishing between the random characters converted to an integer and the real `chunk_ids`.

This information, again, is stored in the raw table data. Here is one way to do string matching with a quartet of (\w\w) to find the bytes.

```
select
      lp,
      get_bit(t_attrs[1], 0) as short,
      get_byte(t_attrs[1], 0) as header byte
from heap_page_item_attrs(
get_raw_page('test_toast', 0), 'test_toast'::regclass);
 lp | short | header_byte
----+-------+-------------
  1 |     1 |          27
  2 |     0 |         144
  3 |     1 |           1
  4 |     1 |           1
```

Only when `short` and `header_byte` are “1” is the data stored in an external TOAST table.

When we do some bi shift operations instead of a regex search and replace, we get the following query to extract the `chunk_ids` for only the rows where data is stored in the external TOAST table:

```
select
lp,
length(t_attrs[1]),
     substr(t_attrs[1], 1, 1),
     case
when get_bit(t_attrs[1], 0) = 1 then
```

```
    'short'
    else
    'normal-size'
    end,
         case
    when get_byte(t_attrs[1], 0) = 1 then
            get_byte(t_attrs[1], 10) +
              (get_byte(t_attrs[1], 11) << 8) +
               (get_byte(t_attrs[1], 12) << 16) +
               (get_byte(t_attrs[1], 13) << 24)
            else
    0
    end as "toast chunk ID"
    from
    heap_page_item_attrs(
    get_raw_page('test_toast', 0),'test_toast'::regclass);

     lp | length | varlena type | toast chunk ID
    ----+--------+--------------+----------------
      1 |     13 | short        |              0
      2 |    804 | normal-size  |              0
      3 |     18 | short        |        1216062
      4 |     18 | short        |        1216063
```

With this information we can now query the external toast toast table for the toast data, for example for the last row in the table.

```
    select
    chunk_id,
    chunk_seq,
    octet_length(chunk_data)
    from pg_toast.pg_toast_1216051
    where chunk_id = 1216063;

     chunk_id | chunk_seq | octet_length
    ----------+-----------+--------------
      1216063 |         0 |         1996
      1216063 |         1 |         1996
      1216063 |         2 |         1008
```

The data is stored in three chunks, totalling 5000 bytes, which fortunately matches the string size we put in there.

The first part is done. We now understand when and how data is stored in the TOAST table and how to retrieve it.

The Problem

The trouble started when we upgraded PostgreSQL from version 9.6 to 13. Because our database is massive (over 200TB), one practical choice was to use `pg_upgrade` with hard links. After stopping all writes, we promoted one of the standbys and proceeded with `pg_upgrade` . Afterwards, we used **rsync** with the `--size-only` option, as recommended in the PostgreSQL documentation.

Before diving into the specifics, it's important to provide a management-level overview of the upgrade process. We first upgraded the primary node using an Ansible playbook and then upgraded the first replica. During this step, we stopped both the primary and the first replica. To maintain availability, we temporarily pointed read-only connections to another available node. Once we upgraded both the primary and the first replica, we brought the services back online, performed additional tests, and continued upgrading the remaining replicas similarly.

With that context, the critical part was that we stopped the promoted standby (now the new primary) during the `pg_upgrade` process, resulting in a new cluster ID and version for the database. Once we performed `rsync` to sync the remaining nodes, we returned the new primary online and started replication to minimize downtime. We thought there was no database activity since all writes had been stopped, but we missed one crucial detail: *`autovacuum` , which prevents wraparound.*

How does Autovacuum work in PostgreSQL?

We dug into PostgreSQL's source code to understand how *`autovacuum`* kicks in to aggressively prevent transaction *`ID wraparound`* :

Entry Point: `do_autovacuum()` (located in `/src/backend/postmaster/autovacuum.c`)

- The `do_autovacuum()` function starts the autovacuum worker, which scans through tables to determine if they need **vacuuming** or **analyzing**.
  The function gathers tables into a list called `table_oids` based on conditions like the risk of transaction wraparound.

Identifying Tables at Risk (handled by `relation_needs_vacanalyze()` in `/src/backend/postmaster/autovacuum.c`)

- The `relation_needs_vacanalyze()` function, called from `do_autovacuum()`, determines if a table needs **vacuuming** or **analyzing**.
  If the `wraparound` flag is set to **true**, indicating that `relfrozenxid` is getting close to the `xidForceLimit`, an **aggressive vacuum** is triggered to **freeze tuples** thoroughly.

- The `xidForceLimit` is calculated by subtracting `autovacuum_freeze_max_age` from the **current transaction ID (** `recentXid` **)**. If a table's `relfrozenxid` is older than `xidForceLimit`, it is marked for an aggressive vacuum.

```
/* Force vacuum if table is at risk of wraparound */
xidForceLimit = recentXid - autovacuum_freeze_max_age;
if (xidForceLimit < FirstNormalTransactionId)
    xidForceLimit -= FirstNormalTransactionId;
force_vacuum = (TransactionIdIsNormal(classForm->relfrozenxid) &&
                TransactionIdPrecedes(classForm->relfrozenxid, xidForceLimit));
```

Rechecking Vacuum Necessity: `table_recheck_autovac()` (found in `/src/backend/postmaster/autovacuum.c`)

- The `table_recheck_autovac()` function double-checks if the table still needs **vacuuming** before proceeding.

- If wraparound is still true, the `autovac_table` structure (**tab**) is set up with parameters for an aggressive vacuum.

Vacuum Parameters for Wraparound Prevention

- The `autovac_table` structure (tab) includes parameters for starting the vacuum, such as the `is_wraparound` flag.

- If **wraparound** is true, the vacuum becomes **aggressive** by avoiding any skipped pages. (`/src/backend/commands/vacuumlazy.c`)

Correlation Between Parameter and Trigger Event

- We looked into how `autovacuum_freeze_max_age` (set to **1.2 billion for us**) impacts when an aggressive autovacuum is triggered.

- The `xidForceLimit` is derived from the **current recentXid** minus `autovacuum_freeze_max_age`. Once the **oldest** `xmin` reaches this `xidForceLimit`, PostgreSQL initiates an aggressive vacuum to prevent wraparound.

Example:

Suppose recentXid is at **2.4 billion**, and `autovacuum_freeze_max_age` is set to **1.2 billion**. The `xidForceLimit` would be calculated as:

```
xidForceLimit = 2.4 billion − 1.2 billion = 1.2 billion
```

Once the **oldest xmin** in any table reaches **1.2 billion** (the calculated `xidForceLimit`), PostgreSQL initiates an aggressive vacuum. This vacuum aims to freeze tuples and prevent wraparound, ensuring the integrity of the database. In our scenario, when the `sweepexecution` table reached `xidForceLimit`, the autovacuum worker initiated an aggressive vacuum to freeze the tuples.

This sequence of events is crucial to avoid the disastrous effects of transaction **ID wraparound**. In our case, missing the importance of autovacuum for preventing wraparound led to unexpected complications only 2 years after the upgrade 😳.

Our famous cleanup job mentioned in the introduction tried to update some data by updating some rows; during this update operation, the database ran into trouble:

```
ERROR: could not access status of transaction 3147388289
DETAIL:  Could not open file "pg_xact/0A17": No such file or directory.
```

Since we used `rsync--size-only`, it's possible that some updates to the visibility map on the primary did not make it to the replicas. Specifically, the visibility map bits for these pages were incorrectly set to "all visible." As a result, vacuum skipped these pages, leaving them in an inconsistent state once the associated transaction information (clog) had been cleared. This ultimately led to corruption in the standby servers. But as mentioned, this problem was hidden; we just had it clear after update operation, which means by that time, we had promoted one of those corrupted standbys and moved on until the historical data was being updated.

When we tried to `VACUUM FREEZE`, we saw the problematic case because `relfrozenxid` is supposed to indicate that **everything before it has been frozen** (i.e. made safe from wraparound). Finding an **xmin** value from before `relfrozenxid` means that there is a tuple in the table that was not correctly frozen, which violates the expectation that all older XIDs should have been frozen.

Mitigation

As we mentioned earlier, this issue came from how we used `rsync` with the `--size-only` flag during the upgrade from primary to replicas. As described in Scenario 1, we first identified this issue with the vacuum not scanning all the blocks and finding `xmin` lower than `relfrozenxid`, which is not sensible. That tuple had to be frozen. This led us to dig deeper and find a connection between the corrupted rows and specific transactions. By examining the `creationDate` of these transactions, we could correlate the corruptions to the time of our upgrade process, indicating that the upgrade itself was likely the source of the issue.

To address it afterwards, we identified which member was the primary at the switch-over and promoted that as the new primary. We then rebuilt all replicas using snapshots from this healthier member, applying fewer fixes compared to the others. What we are sharing here is also an exercise we could have done if we had lost the good replicas and were left to deal with the messy ones.

> *All these scenarios, including solutions, might not fit your case exactly; all the steps we mention as solutions here might be extremely dangerous for your environment. Our experiments here were to dive deeper and study more about Postgresql internals. Using the* ***pg_surgery*** *module provides various functions to perform surgery on a damaged relation. These functions are unsafe by design, and using them may corrupt (or further corrupt) your database. For example, these functions can easily be used to make a table inconsistent with its indexes, to cause UNIQUE or FOREIGN KEY constraint violations, or even to make tuples visible, which, when read, will cause a database server crash. They should be used with great caution and only as a last resort.*

Scenario 1: `xmin` Before `relfrozenxid`

During a **vacuum** operation on a table called `pg_toast11111`, we observed the following:

```
DB=# vacuum (verbose) public.sampledata;
INFO:  vacuuming "public.sampledata"
INFO:  scanned index "sampledata" to remove 2222 row versions
DETAIL:  CPU: user: 82.99 s, system: 17.29 s, elapsed: 221.39 s
INFO:  "sampledata": removed 2222 row versions in 1849 pages
DETAIL:  CPU: user: 0.11 s, system: 0.01 s, elapsed: 0.17 s
INFO:  index "sampledata" now contains 1488209499 row versions in 4369480 pages
DETAIL:  1577 index row versions were removed.
0 index pages have been deleted, 0 are currently reusable.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
INFO:  "sampledata": found 123 removable, 65878 nonremovable row versions in 87
DETAIL:  0 dead row versions cannot be removed yet, oldest xmin: 41139771
There were 229 unused item identifiers.
Skipped 0 pages due to buffer pins, 293689831 frozen pages.
0 pages are entirely empty.
CPU: user: 88.30 s, system: 18.17 s, elapsed: 232.46 s.
INFO:  aggressively vacuuming "pg_toast.pg_toast11111"
ERROR:  found xmin 2730507531 from before relfrozenxid 2860506521
CONTEXT:  while scanning block 34 of relation "pg_toast.pg_toast11111"
```

The error — “`found xmin 2730507531 from before relfrozenxid 2860506521`” — indicates that the `xmin` value of a tuple was older than the table’s `relfrozenxid`, which is a violation of expected behavior. This error tells us that this tuple needs to be frozen.

Fix for Scenario 1:

We used `pg_surgery` to resolve this issue by forcing a freeze on the problematic tuple. The reason, as mentioned previously, is that when we ran `VACUUM FREEZE`, the status of these tuples should have been frozen but it was not, so in order to prevent wraparound we did the freeze. To find the CTID, you need to scan the whole toast table. You can use PL/SQL procedure power to speed this up and even run in threads, but frankly, something like this would work:

```
BEGIN
  EXECUTE 'SELECT * FROM pg_toast.pg_toast11111 WHERE ctid = ''' || generated_t
  tmp_text := tmp::text;
     EXCEPTION WHEN OTHERS THEN
 BEGIN
      INSERT INTO corrupted_rows_toast VALUES(schemaname, tablename, generated_
```

```
 END;
 END;
```

```
DB=# create extension pg_surgery;
DB=# SELECT heap_force_freeze('pg_toast.sampledata', '{"(34,1)"}')
```

After applying the freeze, there are no more invalid `xmin` values. After that, we verify the visibility, as well as a successful VACUUM as follows:

```
with pageimages as (select 34 pagenum, 7 lps, get_raw_page('pg_toast.pg_toast11
from pageimages, heap_page_items(page) hpi ;

 pagenum | lp |   t_xmin   | t_xmax | to_hex
---------+----+------------+--------+--------
      34 |  1 |          2 |      0 | b02
      34 |  2 |            |        |
      34 |  3 |            |        |
      34 |  4 |            |        |
      34 |  5 | 1898817094 |      0 | b02
      34 |  6 | 4216776851 |      0 | b02
      34 |  7 | 4216776851 |      0 | b02
      34 |  8 | 4216776851 |      0 | b02
      34 |  9 | 4216776851 |      0 | b02
(9 rows)

DB=# SELECT * FROM pg_visibility_map('pg_toast.pg_toast_11111', 34);

 all_visible | all_frozen
-------------+------------
 t           | t
(1 row)
```

Scenario 2: Invalid `xmax` values In another instance, we had a table with invalid `xmax` values, which caused inconsistencies. You can find more `pg_visibility` information at https://www.postgresql.org/docs/current/pgvisibility.html and https://wiki.postgresql.org/wiki/Visibility_Map_Problems.

In this scenario, you will see that CTID (5102438,5) has an invalid value set.

```
DB=# select relname, relfrozenxid from pg_class where relname = 'pg_toast_11111

    relname     | relfrozenxid
----------------+--------------
 pg_toast_11111 |    456080953
(1 row)

DB=# select * from pg_visibility('pg_toast.pg_toast_11111',5102438);

 all_visible | all_frozen | pd_all_visible
-------------+------------+----------------
 t           | t          | f
(1 row)

DB=# select * from page_header(get_raw_page('pg_toast.pg_toast_11111', 5102438)

      lsn       | checksum | flags | lower | upper | special | pagesize | versi
----------------+----------+-------+-------+-------+---------+----------+------
 743D0/537946E8 |        1 |     1 |    48 |   184 |    8192 |     8192 |
(1 row)

DB=# select hpi.t_ctid,lp, t_xmin, t_xmax, t_infomask, t_infomask2 , hti.*
DB-#       from heap_page_items( get_raw_page('pg_toast.pg_toast_11111', 510243

   t_ctid    | lp |   t_xmin   |   t_xmax   | t_infomask | t_infomask2 |
-------------+----+------------+------------+------------+-------------+------
 (5102438,1) |  1 | 2392155061 |          0 |       2818 |           3 | {HEAP_
 (5102438,2) |  2 | 2392155061 |          0 |       2818 |           3 | {HEAP_
 (5102438,3) |  3 | 2392155061 |          0 |       2818 |           3 | {HEAP_
 (5102438,4) |  4 |          2 |          0 |       2818 |           3 | {HEAP_
 (5102438,5) |  5 |          2 | 3540933740 |        258 |        8195 | {HEAP_
 (5102438,6) |  6 |          2 |          0 |       2818 |           3 | {HEAP_
(6 rows)
```

Solution

What we know:

- Corruptions in the TOAST table
- Related to unclear transaction status
- The main table does contain the correct transaction status

The process towards fixing the problem:

1. We need to build a mapping between the toast tables and the main table.

2. Use the mapping to compare the transaction status on the main and TOAST table.

- The **chunk_id** from TOAST must also be obtained using **page_inspect**. The transaction status is unclear, so we can't query directly.
- Use the query from the toast section to get the **chunk_id**s from the main table

3. Freeze/kill rows in the TOAST table based on the transaction status.

We collected all the corrupted CTIDs by scanning the database using the `pg_check_frozen` function of the `pg_visibility` extension. Since we had all the corrupted pointers, the remaining part was to do the mapping in place and read and understand the transaction status. Many thanks to Alvaro Herrera from EDB for helping us.

```
select
  lp,
  hti.raw_flags,
  combined_flags
from
  heap_page_items(
get_raw_page('pg_toast.pg_toast_111111', 0) ) hpi,
LATERAL heap_tuple_infomask_flags(t_infomask, t_infomask2) hti ;

 lp |                                raw_flags
 ---+-------------------------------------------------------------------------
  1 | {HEAP_HASVARWIDTH,HEAP_XMIN_COMMITTED,HEAP_XMAX_INVALID}
  2 | {HEAP_HASVARWIDTH,HEAP_XMAX_INVALID}
  3 | {HEAP_HASVARWIDTH,HEAP_XMAX_INVALID}
  4 | {HEAP_HASVARWIDTH,HEAP_XMIN_COMMITTED,HEAP_XMIN_INVALID,HEAP_XMAX_INVALID
```

At the end of our investigation, we needed to understand the status of the tuples in our TOAST tables. Specifically, we focused on the **combined flags** in `t_infomask`, which guide the necessary actions for each tuple. Using `pg_surgery`, we cleaned up the corrupted data as follows:

```
('{HEAP_HASVARWIDTH, HEAP_XMIN_COMMITTED, HEAP_XMAX_INVALID}', 'heap_force_free
('{HEAP_HASVARWIDTH, HEAP_XMIN_COMMITTED, HEAP_KEYS_UPDATED}', 'heap_force_free
```

```
    ('{HEAP_HASVARWIDTH, HEAP_XMIN_INVALID, HEAP_XMAX_INVALID}', 'heap_force_kill')
    ('{HEAP_HASVARWIDTH, HEAP_XMIN_COMMITTED, HEAP_XMAX_COMMITTED, HEAP_KEYS_UPDATE
    ('{HEAP_HASVARWIDTH, HEAP_XMAX_INVALID}', 'heap_force_freeze')
```

To explain our decisions here, it helps to look at `t_infomask` definitions from `src/include/access/htup_details.h`:

```
/*
 * information stored in t_infomask:
 */
#define HEAP_HASNULL    0x0001 /* has null attribute(s) */
#define HEAP_HASVARWIDTH  0x0002 /* has variable-width attribute(s) */
#define HEAP_XMIN_COMMITTED  0x0100 /* t_xmin committed */
#define HEAP_XMIN_INVALID  0x0200 /* t_xmin invalid/aborted */
#define HEAP_XMIN_FROZEN  (HEAP_XMIN_COMMITTED|HEAP_XMIN_INVALID)
#define HEAP_XMAX_COMMITTED  0x0400 /* t_xmax committed */
#define HEAP_XMAX_INVALID  0x0800 /* t_xmax invalid/aborted */
#define HEAP_KEYS_UPDATED  0x2000 /* tuple was updated and key cols modified, c
#define HEAP_XMIN_FROZEN  (HEAP_XMIN_COMMITTED|HEAP_XMIN_INVALID)
```

Combined Flags and Tuple Actions

1. `{HEAP_HASVARWIDTH, HEAP_XMIN_COMMITTED, HEAP_XMAX_INVALID}`

- These tuples were **correctly written at some point and never updated**, but the `XMIN_FREEZE` bit was not set, making them **unreadable.**
- **Action:** Use `heap_force_freeze` to set `XMIN_FREEZE`, making these rows readable without ambiguity.

2. `{HEAP_HASVARWIDTH, HEAP_XMIN_COMMITTED, HEAP_KEYS_UPDATED}`

- These tuples have `XMIN_COMMITTED` and `HEAP_KEYS_UPDATED`. The `xmax` values range from **3517981142 to 3542253656.** The details of the `xmax` values suggest these transactions might have been committed **or aborted.**
- Since **vacuum** had scanned these pages to the point of marking them **all-frozen**, it's inferred that if these tuples remained, the **transactions likely aborted**, and

the tuples should be visible.

- **Action: heap_force_freeze** the rows.

3. Null Data in Combined Flags

- These tuples had no data, meaning they were **empty line pointers**.
- **Action: heap_force_kill**

4. `{HEAP_HASVARWIDTH, HEAP_XMIN_COMMITTED, HEAP_XMAX_COMMITTED, HEAP_KEYS_UPDATED}`

- These tuples have **XMAX_COMMITTED** and **HEAP_KEYS_UPDATED**, which means they were **deleted**.
- **Action: heap_force_kill** the rows, as they no longer hold valid data.

5. `{HEAP_HASVARWIDTH, HEAP_XMAX_INVALID}`

- These tuples have `XMAX_INVALID` and a valid `xmin` value, suggesting the **inserting transaction was committed** and no subsequent actions were taken.
- Since there was no `XMIN_COMMITTED` set, it's possible that no transaction validated these tuples for removal or freezing. To err on the side of caution and avoid possible data loss, we freeze these rows to prevent ambiguity in their status.
- **Action: Freeze the row** to avoid data loss.

In this cleanup process, we opted for caution, prioritizing data integrity over prematurely removing rows. We risked losing data if we incorrectly guessed that a tuple was dead and removed it. By freezing and keeping these tuples, the risk was minimal — lingering TOAST tuples do little harm compared to losing data permanently.

Extracting and Processing Corrupted Tuples with SQL

As we dived deeper into analyzing and fixing the corrupted tuples in our PostgreSQL TOAST data, we decided to use SQL scripts to extract detailed information on the status of each tuple. Using `pg_surgery` functions, we identified problematic tuples and took actions such as freezing or killing tuples. A small prerequisite is that assuming you have run `pg_check_frozen()` from pg_visibility in db and already have

all the ctids stored in some table. In our SQL, you will see that table as `pg_frozen_check_results`.

Step 1: Extracting Tuple Information

First, we extracted the tuple information from `pg_toast` to see the detailed status of each tuple:

`{Blkno,offs}` is actually `{CTID}`, `get_raw_page()` and `heap_page_items()`: are the functions that allowed us to read the raw page content and extract information like `t_infomask` and match **tuple identifiers** (`t_ctid`).

```
create table addresses_pg_toast_11111 as
with addresses as (
    select (t_ctid::text::point)[0]::integer as blkno,
           array_agg((t_ctid::text::point)[1]::smallint) as offs
    from pg_frozen_check_results
    where table_name='pg_toast_36822'
    group by 1
)
select *
from (
    with pages as (
        select blkno, get_raw_page('pg_toast.pg_toast_36822', blkno) as rawpage
        from addresses
    )
    select blkno, lp, t_xmin, t_xmax, t_field3, t_ctid,
           (heap_tuple_infomask_flags(t_infomask, t_infomask2)).raw_flags::text
    from pages, heap_page_items(rawpage) hpi
) inn
where (blkno, lp) in (select blkno, unnest(offs) from addresses);
```

Step 2: Defining Actions for Each Tuple Group

After collecting the detailed information, we grouped tuples based on their `t_infomask` and defined appropriate actions using `pg_surgery` functions:

```
with groups (mask, action) as (
    values
    ('{HEAP_HASVARWIDTH,HEAP_XMIN_COMMITTED,HEAP_XMAX_INVALID}', 'freeze'),
    ('{HEAP_HASVARWIDTH,HEAP_XMIN_COMMITTED,HEAP_KEYS_UPDATED}', 'freeze'),
```

```
        ('{HEAP_HASVARWIDTH,HEAP_XMIN_INVALID,HEAP_XMAX_INVALID}', 'kill'),
        ('{HEAP_HASVARWIDTH,HEAP_XMIN_COMMITTED,HEAP_XMAX_COMMITTED,HEAP_KEYS_UPDAT
        ('{HEAP_HASVARWIDTH,HEAP_XMAX_INVALID}', 'freeze')
    )
    select case
                when action = 'freeze' then heap_force_freeze('pg_toast.pg_toast_11
                                                              array_agg(format('(%s
                when action = 'kill' then heap_force_kill('pg_toast.pg_toast_11111'
                                                          array_agg(format('(%s,%s)
           end
    from addresses_pg_toast_11111
    join groups on (mask = raw_flags)
    group by blkno, action
    order by blkno;
```

Lessons learned

We know now that these corruptions were likely caused by automatic transaction wraparound vacuuming during our upgrade process. After upgrading the first and second instances, we restarted the primary. At that point, autovacuum began updating the infomasks by freezing rows — which turned out to be invalid in our case, leading to the corruption.

The core issue was that some transactions on certain pages were not properly finalized. Normally, the vacuum process in PostgreSQL cleans up heap pages, including updating transaction-related columns and setting hint bits. However, in our case, these heap pages were incorrectly marked as “frozen” in the visibility map, even though they hadn’t been fully cleaned. As a result, the database incorrectly treated these pages as if they required no further attention. This error remained unnoticed because those pages weren’t accessed for a long time, and the problem only became apparent when we began cleaning up old data and revisiting those pages, revealing inconsistencies due to transactions still marked as incomplete.

While we may not have all the specifics of why it happened, we’ve identified some improvements to our upgrade process that should prevent similar issues in our environment and yours.

Upgrade improvements

- Rsync from your newly promoted standby and keep it shut while rsync finishes.
- If not possible, make sure no vacuum is running while starting the upgrade.

- Disable auto-vacuum.
- Use **rsync –size-only** for data heap files if you have to, but skip the **size-only** flag for the visibility map and free space map (.vm and .fsm)

Advice to the reader

If you encounter similar issues during a PostgreSQL upgrade, where tables appear corrupted due to invalid transaction statuses or inconsistent visibility maps, consider the following steps:

1. **Understand the Visibility Map and Infomasks:** Visibility maps are critical for tracking which pages can be skipped during vacuum operations. If a page is incorrectly marked as frozen, autovacuum might overlook it, leading to inconsistencies. Be vigilant about understanding how autovacuum interacts with these markers, especially during major upgrades.
2. **Use Tools Like** `pg_visibility` **or** `amcheck` **:** For visibility map-related issues, extensions like `pg_surgery` can be invaluable for addressing specific tuples that need attention for operation (be careful!). Additionally, `amcheck` can validate indexes and detect corruption early before it cascades into larger issues.
3. **Pause Autovacuum Strategically:** During in-place upgrades and syncing from the master method, consider pausing **autovacuum** or monitoring it very closely to prevent it from updating infomasks prematurely. This is especially important if you're running `pg_upgrade` and using techniques like **rsync** to sync multiple instances. The **safest method** is to use disk snapshots and restores for replicas while everything is shut.
4. **Be Proactive with Snapshots:** Take snapshots of the upgraded instance before bringing it online. This can help you avoid cascading corruption in replicas and provide a cleaner fallback if things go wrong.

Visibility map corruption can be tricky because it silently impacts how PostgreSQL vacuums your tables. By staying proactive and using the right tools, you can mitigate the risk and respond quickly if something goes wrong.

Acknowledgements

During this long journey, we would like to take a moment to acknowledge the engineers who contributed to this effort from the Adyen DBA team: Cagri Biroglu,

Derk Van Veen and Milen Blagojevic, as well as Alvaro Herrera, Mark Dilger and Artur Nascimento from EDB.

Postgresql Scale Database Corruption Adyen Toast Storage

Follow

Published in Adyen Tech

1.1K Followers · Last published Feb 5, 2025

Insights from the team building the world's payments infrastructure.

Follow

Written by Adyen

1.3K Followers · 2 Following

Development and design stories from the company building the world's payments infrastructure.
https://www.adyen.com/careers/

Responses (1)

Gvadakte

What are your thoughts?

Mohammed Brückner
Jan 26

One could enhance checksum usage by not only checking at the table level but also inspecting individual table pages. This granular analysis might surface subtle corruption patterns. In addition to PostgreSQL server logs, an analysis of the... more

Reply

More from Adyen and Adyen Tech

In Adyen Tech by Adyen

Efficiently RePartitioning Large Tables in PostgreSQL

By Cagri Biroglu, Database Engineer

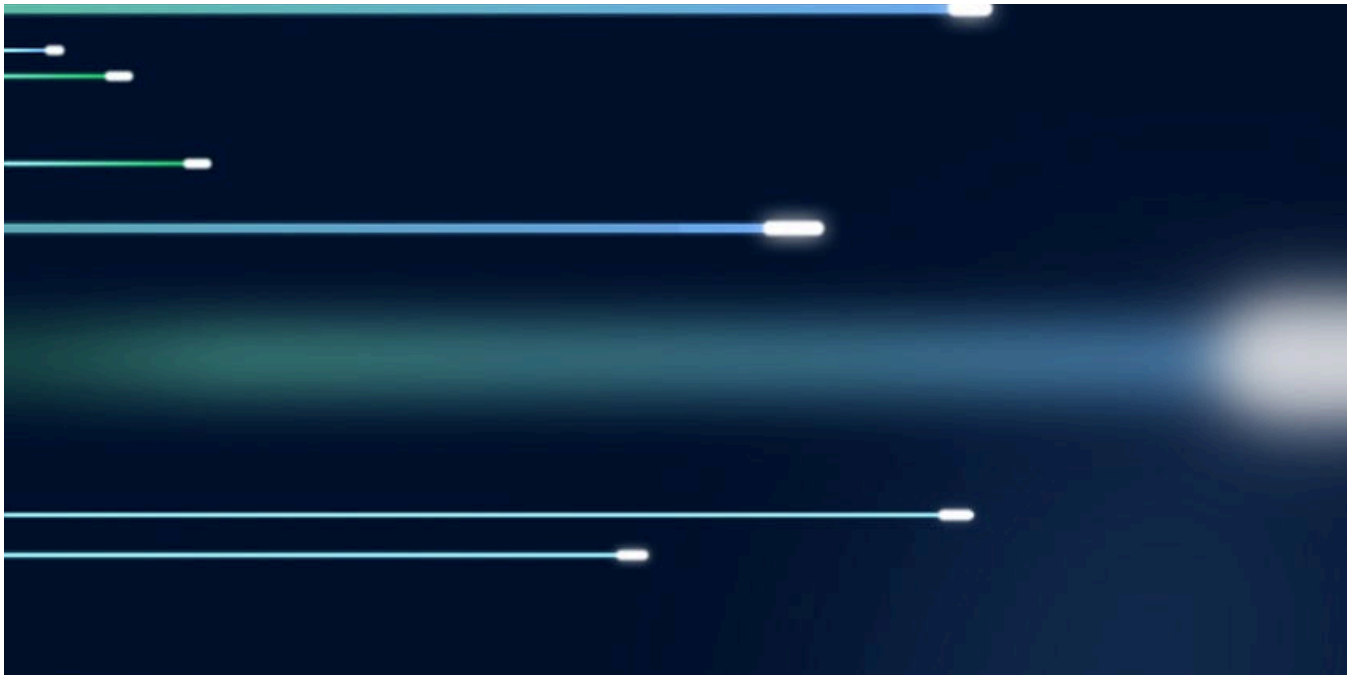
Nov 26, 2024 86

In Adyen Tech by Adyen

A Developer's Guide to the new iDEAL

By Beppe Catanese, Developer Advocate, Adyen

Sep 10, 2024



In Adyen Tech by Adyen

The AI behind Uplift

by Andreu Mora · SVP / Global Head of Engineering Data, Adyen

Jan 17 125

In Adyen Tech by Adyen

LLM inference at scale with TGI

By Martin Iglesias Goyanes, Machine Learning Enigneer, Adyen

Sep 16, 2024 3 1

See all from Adyen

See all from Adyen Tech

Recommended from Medium

In Databases by Sergey Egorenkov

Why Uber Moved from Postgres to MySQL

How PostgreSQL's architecture clashed with Uber's scale—and why MySQL offered a better path forward

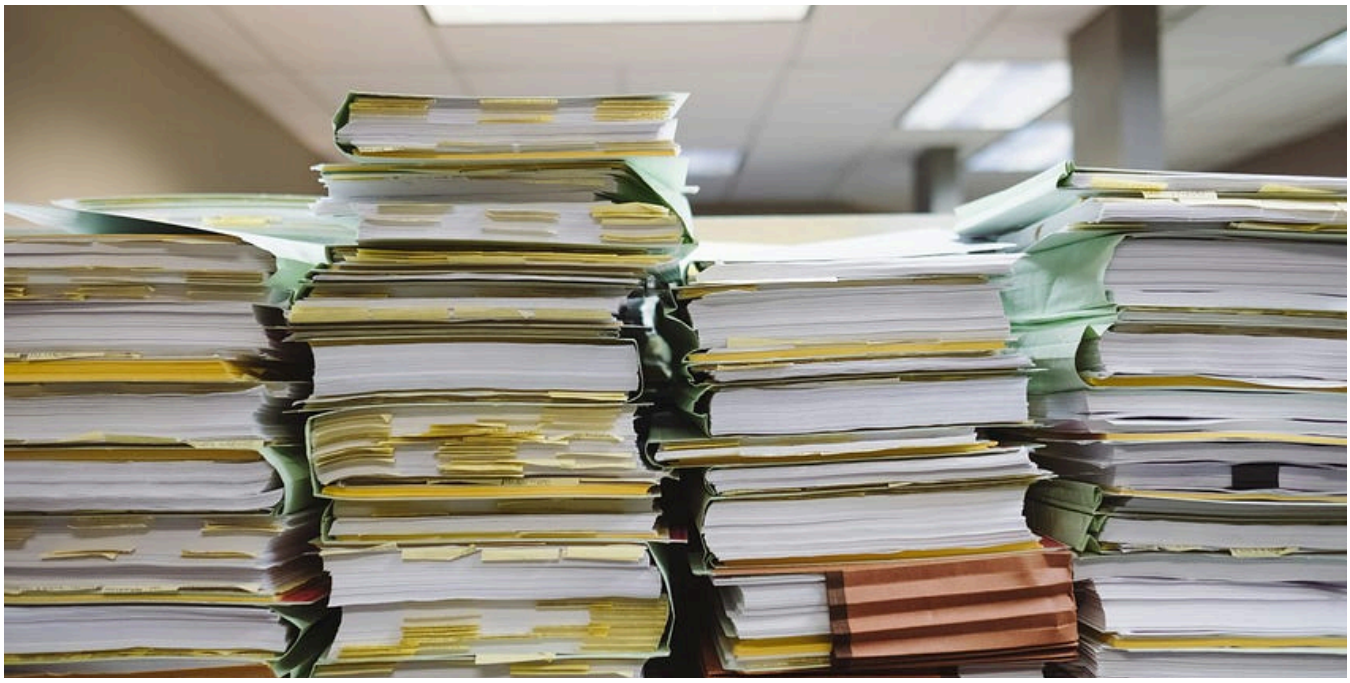
Mar 29 228 7

In Hack the Stack by Coders Stop

9 Database Optimization Tricks SQL Experts Are Hiding From You

Most developers learn enough SQL to get by—SELECT, INSERT, UPDATE, DELETE, and maybe a few JOINs. They might even know how to create...

Mar 27 185 5



Tihomir Manushev

Unstructured Data in PostgreSQL

JSONB and hstore can Replace a NoSQL Database Like MongoDB

Mar 25 22

Dipanshu

Paying for software is stupid ..Open-Source tools to Destroy Your SaaS Expenses

These 40 Open-Source Tools Will Make Your SaaS Subscriptions Look Obsolete

Mar 27 2K 26

In Dev Genius by Doran Gao

Efficient Large Table Cleanup in PostgreSQL

“We can only see a short distance ahead, but we can see plenty there that needs to be done.” —Alan Turing

Oct 31, 2024 1

Ajaymaurya

What Are the Best Indexing Strategies for High-Performance PostgreSQL Queries?

PostgreSQL is a powerful open-source relational database, but ensuring optimal performance requires well-planned indexing strategies. A...

Mar 15 1

See more recommendations