credativ

Database in Distress

Testing and Repairing Different Types of Database Corruption

Josef Machytka <josef.machytka@credativ.de>

2025-09-12 - PgDay Lowlands Rotterdam

credativ GmbH

- Founded 1999 in Jülich, Germany
- Close ties to Open-Source Community
- More than 40 Open-Source experts
- Consulting, development, training, support (3rd-level / 24x7)
- Open-Source infrastructure with Linux, Kubernetes and Proxmox
- Open-Source databases with PostgreSQL
- DevSecOps with Ansible, Puppet, Terraform and others
- Since 2025 independent owner-managed company again

Josef Machytka

credativ

- Professional Service Consultant - PostgreSQL specialist at credativ GmbH
- 30+ years of experience with different databases
- PostgreSQL (13y), BigQuery (7y), Oracle (15y), MySQL (12y), Elasticsearch (5y), MS SQL (5y)
- 10+ years of experience with Data Ingestion pipelines, Data Analysis, Data Lake and Data Warehouse
- 2+ years of practical experience with different LLMs / AI including their architecture and principles
- From Czechia, living now 11 years in Berlin

  - LinkedIn: linkedin.com/in/josef-machytka
  - ResearchGate: researchgate.net/profile/Josef-Machytka
  - Academia.edu: academia.edu/JosefMachytka
  - Medium: medium.com/@josef.machytka
  - sessionize: sessionize.com/josefmachytka

We Must Keep Asking Questions

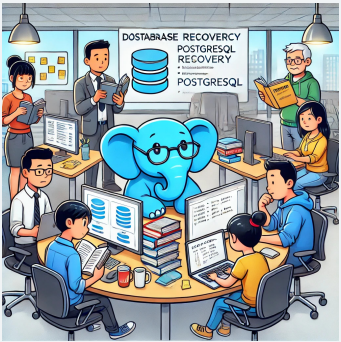
credativ

- Simon Riggs: The Next 20 Years (PGConf.EU 2023)

```
I encourage everybody not to get too confident
that because PostgreSQL is number one,
therefore we are doing everything right...

We still need to ask stupid questions.
And wonderful new users are full of them.

Why don't you do it this way?
Why don't you do that?
Why doesn't it work like this?

They're good questions.
```

Useful Resources for Repairing PostgreSQL Database Corruption

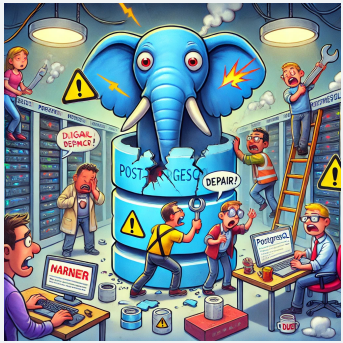
- PostgreSQL talks:
  - PostgreSQL Database Corruption (YouTube Playlist)
  - How to corrupt your database (and how to deal with data corruption) (PGConf.EU 2023)
  - Data Corruption Bugs: Diagnosis and Lessons (PGConf.dev 2024)
- Articles:
  - PostgreSQL Wiki: Corruption
  - Garrett's Blog: Salvaging a Corrupted Table from PostgreSQL
  - StackOverflow: Repair corrupt database PostgreSQL
  - Salvaging a Corrupted Table from PostgreSQL
  - Cybertec: How to corrupt your PostgreSQL database
- This talk dives deeper into different types of data corruption
- Demonstrates usage of data checksums and extensions for diagnostics



AI images created by the author using ChatGPT DALL-E

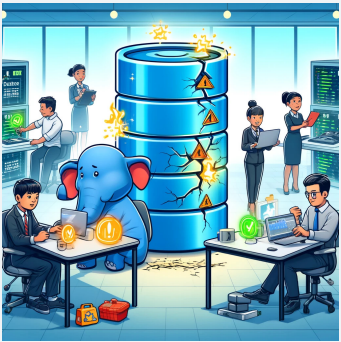
PostgreSQL is actually remarkably stable

- PostgreSQL on Linux is highly stable under typical circumstances
- All damaged databases I had to repair were on Windows
- In one case a crashed kubernetes cluster damaged data
- If you have functional backup you are (mostly) safe
- Except for types of backups which can mask corruption
- In all cases I encountered there was NO useful backup



Data Corruption can be hidden

- PostgreSQL can use data block checksums to detect corruption
- In PG 17 and older not enabled due to performance concerns
- Without checksums corruption may remain undetected
- Backup tools like pg_basebackup copy files without data checks
- Corruption is revealed only when data is read
- Often as errors during pg_dump backup



Data Corruption can be hidden

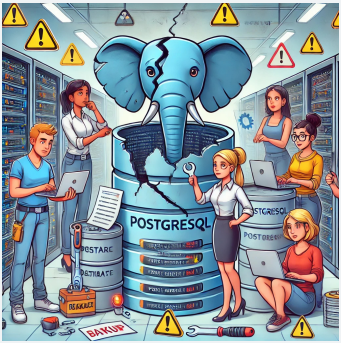
credativ

- Typical data corruption related errors during pg_dump backup:

```
-- invalid page in a table:
pg_dump: error: query failed: ERROR:  invalid page in block 0 of relation base/16384/66427
pg_dump: error: query was: SELECT last_value, is_called FROM public.test_table_bytea_id_seq


-- damaged system columns in a tuple:
pg_dump: error: Dumping the contents of table "test_table_bytea" failed: PQgetResult() failed.
pg_dump: error: Error message from server: ERROR:  could not access status of transaction 3353862211
DETAIL:  Could not open file "pg_xact/0C7E": No such file or directory.
pg_dump: error: The command was: COPY public.test_table_bytea (id, id2, id3, description, data) TO stdout;


-- damaged sequence:
pg_dump: error: query to get data of sequence "test_table_bytea_id2_seq" returned 0 rows (expected 1)


-- memory segmentation fault during pg_dump:
pg_dump: error: Dumping the contents of table "test_table_bytea" failed: PQgetCopyData() failed.
pg_dump: error: Error message from server: server closed the connection unexpectedly
        This probably means the server terminated abnormally
        before or while processing the request.
pg_dump: error: The command was: COPY public.test_table_bytea (id, id2, id3, description, data) TO stdout;
```

Corruption is often hidden

- Old or rarely accessed records can hide corruption
- History and audit tables are prone to unnoticed damage
- Unused indexes may contain silent corruption
- Autovacuum may fail due to corruption, but we might not notice
- Frozen tuples can be damaged and escape detection



pg_dump can show all corrupted objects

- We can use pg_dump to identify all corrupted objects
- Example script for Windows in PowerShell

```
$dburl="postgresql://postgres@localhost:5434/"
$date="20250301_1200"

$databases=& "C:\Programme\PostgreSQL\12\bin\psql.exe" -t -A -c "select datname from pg_database" $dburl"↩
     postgres"

foreach ($db in $databases) {

    $tables=& "C:\Programme\PostgreSQL\12\bin\psql.exe" -t -A -c "select schemaname||'.'||relname from ↩
     pg_stat_user_tables" $dburl$db

        foreach ($table in $tables) {
        & "C:\Programme\PostgreSQL\12\bin\pg_dump.exe" -Fc -E UTF-8 -t $table -f "F:\\credativ\dumps\$date-$db-↩
     $table.dump" $dburl$db   2>"F:\\credativ\logs\$date-$db-$table.log"
    }
}
```

Any PostgreSQL Object Can Be Corrupted

- PostgreSQL allows different access methods, ”heap” is the default
- Older databases all use heap tables - this talk focuses on them
- Main tables, TOAST tables, and sequences are all heap relations
- They can all have similar issues
- I created ”Corruption Simulator” to test various corruption scenarios
- Code surgically damages selected parts of heap data block for testing

What about Checksums?

- PostgreSQL offers data block checksums to detect storage corruption
- Disabled by default in PostgreSQL 17 and earlier due to performance concerns
- Enabling checksums later requires downtime and data integrity checks
- pg_checksums tool can activate checksums on existing clusters
- PostgreSQL 18 enables checksums by default during initdb
- The ”ignore_checksum_failures” parameter controls error handling

```
initdb: Change default to using data checksums.
author  Peter Eisentraut <peter@eisentraut.org>
        Wed, 16 Oct 2024 06:45:09 +0000 (08:45 +0200)
committer   Peter Eisentraut <peter@eisentraut.org>
               Wed, 16 Oct 2024 06:48:10 +0000 (08:48 +0200)

Checksums are now on by default.  They can be disabled by the previously added option --no-data-checksums.

Author: Greg Sabino Mullane <greg@turnstep.com>
Reviewed-by: Nathan Bossart <nathandbossart@gmail.com>
Reviewed-by: Peter Eisentraut <peter@eisentraut.org>
Reviewed-by: Daniel Gustafsson <daniel@yesql.se>
Discussion: https://www.postgresql.org/message-id/flat/CAKAnmmKwiMHik5AHmBEdf5vqzbOBbcwEPHo4-PioWeAbzwcTOQ@mail.gmail.com
```

Detection of Corruption with Checksums

- Checksums are calculated for each data block, stored in the header
- On read, PostgreSQL recalculates and verifies the checksum for each block
- Mismatched checksums indicate corruption and trigger an error
- Errors can be skipped setting ”ignore_checksum_failures=ON”
- Corrupted pages remain unchanged; only error handling behavior is modified

```
-- ignore_checksum_failure="off";  --- Query stops on the first checksum error
test=# select * from pg_toast.pg_toast_17453;
WARNING: page verification failed, calculated checksum 19601 but expected 152
ERROR: invalid page in block 0 of relation base/16384/16402


-- ignore_checksum_failure="on";  --- Query skips checksum errors, exits on other errors
test=# select * from pg_toast.pg_toast_17453;
WARNING:  page verification failed, calculated checksum 29668 but expected 57724
WARNING:  page verification failed, calculated checksum 63113 but expected 3172
WARNING:  page verification failed, calculated checksum 59128 but expected 3155
ERROR:  could not access status of transaction 3088756928
DETAIL:  Could not open file "pg_xact/0B81": No such file or directory.
```

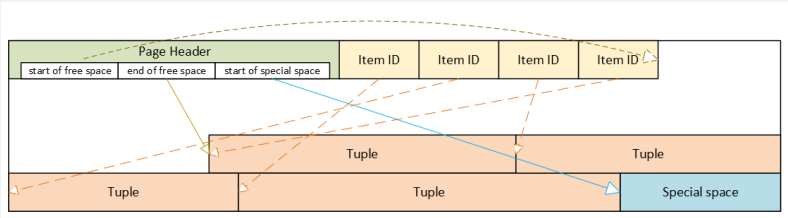
Detection of Corruption with Checksums

- Damaged blocks can be ”zeroed out” by setting ”zero_damaged_pages=ON”
- Corrupted pages are replaced in memory with zeroed pages during query execution
- Zeroed pages are automatically written to disk, permanently overwriting data
- Setting ”zero_damaged_pages=OFF” does not restore lost data or undo changes
- Only scanned blocks are zeroed; unscanned corrupted pages may still cause errors

```
-- zero_damaged_pages=OFF --> Query stops on the first checksum error
test=# select * from test_table_bytea;
WARNING: page verification failed, calculated checksum 19601 but expected 152
ERROR: invalid page in block 0 of relation base/16384/16402


-- zero_damaged_pages=ON --> Query skips damaged blocks and continues
test=# select * from pg_toast.pg_toast_17453;
WARNING: page verification failed, calculated checksum 29668 but expected 57724
WARNING: invalid page in block 204 of relation base/16384/17464; zeroing out page
WARNING: page verification failed, calculated checksum 63113 but expected 3172
WARNING: invalid page in block 222 of relation base/16384/17464; zeroing out page
WARNING: page verification failed, calculated checksum 59128 but expected 3155
```

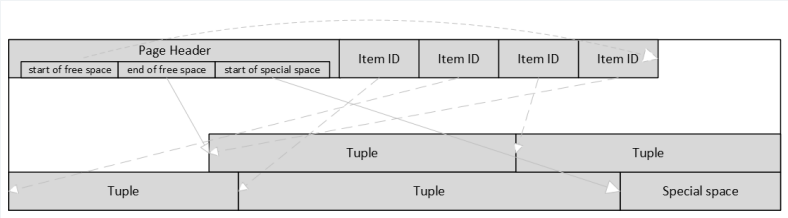
Dissecting Data Corruption - Anatomy of Data Block

- PostgreSQL stores all data in heap tables using 8 KB data blocks
- Block consists of: Header, ItemIDs array, Free space, Tuples, Special space
- The Header contains metadata for block management and integrity
- ItemIDs array points to individual tuples within the block
- Tuples hold actual row data, each with its own header
- Special space is reserved for index-specific or relation-specific data



Different Types of Data Corruption

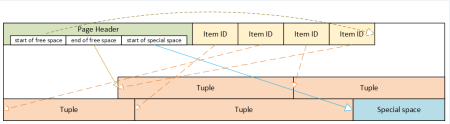
- Error type depends on which part of the block is damaged
- Corrupted header: entire block becomes inaccessible
- Corrupted ItemIDs array: tuples cannot be read
- Corrupted tuples: row data and system columns are invalid
- Corrupted special space: index or relation-specific errors



Corrupted Header

- Block header occupies the first 24 bytes of each data block
- Corruption of the header makes the entire block inaccessible
- ERROR: invalid page in block 285 of relation base/16384/29724
- Only this error can be skipped by setting ”zero_damaged_pages = on”
- Blocks with corrupted header are ”zeroed” in memory and skipped
- VACUUM can remove these blocks
- Nothing to diagnose here, all selects fail on these blocks

Field	Type	Length	Description
pd_lsn	PageXLogRecPtr	8 bytes	LSN: next byte after last byte of WAL record for last change to this page
pd_checksum	uint16	2 bytes	Page checksum
pd_flags	uint16	2 bytes	Flag bits
pd_lower	LocationIndex	2 bytes	Offset to start of free space
pd_upper	LocationIndex	2 bytes	Offset to end of free space
pd_special	LocationIndex	2 bytes	Offset to start of special space
pd_pagesize_version	uint16	2 bytes	Page size and layout version number information
pd_prune_xid	TransactionId	4 bytes	Oldest unpruned XMAX on page, or zero if none



Corrupted Header - look into code

- Looking at the branch ”REL_17_STABLE” / ”REL_18_STABLE”
- Error message ”invalid page in block xx of relation ...”
- From src/backend/catalog/storage.c - ”RelationCopyStorage”
- If ”PageIsVerifiedExtended” (”PageIsVerified” in 18) (src/backend/storage/page/bufpage.c) returns ”false”

Field	Type	Length	Description
pd_lsn	PageXLogRecPtr	8 bytes	LSN: next byte after last byte of WAL record for last change to this page
pd_checksum	uint16	2 bytes	Page checksum
pd_flags	uint16	2 bytes	Flag bits
pd_lower	LocationIndex	2 bytes	Offset to start of free space
pd_upper	LocationIndex	2 bytes	Offset to end of free space
pd_special	LocationIndex	2 bytes	Offset to start of special space
pd_pagesize_version	uint16	2 bytes	Page size and layout version number information
pd_prune_xid	TransactionId	4 bytes	Oldest unpruned XMAX on page, or zero if none

```
        /*
         * The following checks don't prove the header is correct, only that
         * it looks sane enough to allow into the buffer pool. Later usage of
         * the block can still reveal problems, which is why we offer the
         * checksum option.
         */
        if ((p->pd_flags & ~PD_VALID_FLAG_BITS) == 0 &&
            p->pd_lower <= p->pd_upper &&
            p->pd_upper <= p->pd_special &&
            p->pd_special <= BLCKSZ &&
            p->pd_special == MAXALIGN(p->pd_special))
            header_sane = true;

        if (header_sane && !checksum_failure)
            return true;
```

Corrupted Header - look into code

- Let's look at PD_VALID_FLAG_BITS
- Defined in src/include/storage/bufpage.h

Field	Type	Length	Description
pd_lsn	PageXLogRecPtr	8 bytes	LSN: next byte after last byte of WAL record for last change to this page
pd_checksum	uint16	2 bytes	Page checksum
pd_flags	uint16	2 bytes	Flag bits
pd_lower	LocationIndex	2 bytes	Offset to start of free space
pd_upper	LocationIndex	2 bytes	Offset to end of free space
pd_special	LocationIndex	2 bytes	Offset to start of special space
pd_pagesize_version	uint16	2 bytes	Page size and layout version number information
pd_prune_xid	TransactionId	4 bytes	Oldest unpruned XMAX on page, or zero if none

```
/*
 * pd_flags contains the following flag bits.  Undefined bits are initialized
 * to zero and may be used in the future.
 *
 * PD_HAS_FREE_LINES is set if there are any LP_UNUSED line pointers before
 * pd_lower.  This should be considered a hint rather than the truth, since
 * changes to it are not WAL-logged.
 *
 * PD_PAGE_FULL is set if an UPDATE doesn't find enough free space in the
 * page for its new tuple version; this suggests that a prune is needed.
 * Again, this is just a hint.
 */
#define PD_HAS_FREE_LINES   0x0001  /* are there any unused line pointers? */
#define PD_PAGE_FULL        0x0002  /* not enough free space for new tuple? */
#define PD_ALL_VISIBLE      0x0004  /* all tuples on page are visible to
                                     * everyone */

#define PD_VALID_FLAG_BITS  0x0007  /* OR of all valid pd_flags bits */
```

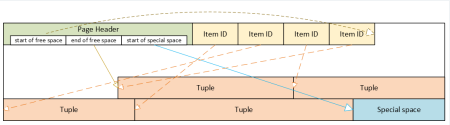
Corrupted Header - Diagnostics with Extensions

- Let's try different extensions if we can diagnose something in this case

```
-- Healthy page header:
SELECT * FROM page_header(get_raw_page('pg_toast.pg_toast_32840', 100));
    lsn     | checksum | flags | lower | upper | special | pagesize | version | prune_xid
------------+----------+-------+-------+-------+---------+----------+---------+-----------
 0/2B2FCD68 |        0 |     4 |    40 |    64 |    8192 |     8192 |       4 |         0
(1 row)


--> zero_damaged_pages=OFF
-- pageinspect
SELECT * from page_header(get_raw_page('pg_toast.pg_toast_28740', 578))
ERROR: XX001-invalid page in block 578 of relation base/16384/28751

-- amcheck
SELECT * FROM verify_heapam('pg_toast.pg_toast_28740', FALSE, TRUE, 'none', 578, 578)
ERROR: XX001-invalid page in block 578 of relation base/16384/28751


--> zero_damaged_pages=ON
-- pageinspect
SELECT * from page_header(get_raw_page('pg_toast.pg_toast_28740', 578));
WARNING:  invalid page in block 578 of relation base/16384/28751; zeroing out page
 lsn | checksum | flags | lower | upper | special | pagesize | version | prune_xid
-----+----------+-------+-------+-------+---------+----------+---------+-----------
 0/0 |        0 |     0 |     0 |     0 |       0 |        0 |       0 |         0
(1 row)

-- amcheck
SELECT * FROM verify_heapam('pg_toast.pg_toast_28740', FALSE, TRUE, 'none', 578, 578);
 blkno | offnum | attnum | msg
-------+--------+--------+-----
(0 rows)
```

Corrupted Item IDs Array

- Contains 4 bytes pointers to the tuples (offset and length)
- Corrupted item ids array means no access to tuples
- Offset and Length contain random, often bigger than 8192



- ERROR: invalid memory alloc request size 18446744073709551594
- DEBUG: server process (PID 76) was terminated by signal 11: Segmentation fault

```
SELECT lp, lp_off, lp_flags, lp_len, t_xmin, t_xmax, t_field3, t_ctid, t_infomask2, t_infomask, t_hoff, t_bits, t_oid, substr(t_data::text,1,50) as t_data
FROM heap_page_items(get_raw_page('public.test_table_bytea', 7));
```

lp	lp_off	lp_flags	lp_len	t_xmin	t_xmax	t_field3	t_ctid	t_infomask2	t_infomask	t_hoff	t_bits	t_oid	t_data
1	7936	1	252	29475	0	0	(7,1)	5	2310	24			\x01010000010100000101000018030000486f742073656520
2	7696	1	236	29476	0	0	(7,2)	5	2310	24			\x02010000020100000201000d802000043756c747572616c
3	7504	1	189	29477	0	0	(7,3)	5	2310	24			\x03010000030100000301000001c020000446f6f7220726563
4	7368	1	132	29478	0	0	(7,4)	5	2310	24			\x04010000040100000401000009d4d6f76656d656e74207374
5	7128	1	238	29479	0	0	(7,5)	5	2310	24			\x05010000050100000501000e0020000426f617264207065
6	6872	1	249	29480	0	0	(7,6)	5	2310	24			\x06010000060100000601000000c030000576866f6c6520616c
7	6648	1	219	29481	0	0	(7,7)	5	2310	24			\x07010000070100000701000094020000416756e6379206d

lp	lp_off	lp_flags	lp_len	t_xmin	t_xmax	t_field3	t_ctid	t_infomask2	t_infomask	t_hoff	t_bits	t_oid	t_data
1	6577	1	28310										
2	20113	3	13097										
3	1273	1	28308										
4	17972	0	15077										
5	11161	2	28274										

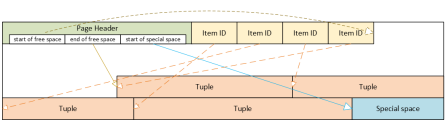
Diagnostics with amcheck

- amcheck's "verify_heapam" function used on heap page with corrupted ItemIDs
- Can target specific blocks and provides detailed diagnostics
- But output messages often cryptic and difficult to interpret
- Corrupted values in ItemIDs array are reported as-is, not validated

```
-- pageinspect
SELECT lp, lp_off, lp_flags, lp_len, t_xmin, t_xmax, t_field3, t_ctid, t_infomask2, t_infomask, t_hoff, t_bits, t_oid, t_data
FROM heap_page_items(get_raw_page('pg_toast.pg_toast_32840', 563));

 lp | lp_off | lp_flags | lp_len | t_xmin | t_xmax | t_field3 | t_ctid | t_infomask2 | t_infomask | t_hoff | t_bits | t_oid | t_data
----+--------+----------+--------+--------+--------+----------+--------+-------------+------------+--------+--------+-------+--------
  1 |  19543 |        1 |  16226 |        |        |          |        |             |            |        |        |       |
  2 |   5585 |        2 |   3798 |        |        |          |        |             |            |        |        |       |
  3 |  25664 |        3 |  15332 |        |        |          |        |             |            |        |        |       |
  4 |  10285 |        2 |  17420 |        |        |          |        |             |            |        |        |       |
(4 rows)


-- amcheck
SELECT * FROM verify_heapam('pg_toast.pg_toast_32840', FALSE, FALSE, 'none', 563, 563);

 blkno | offnum | attnum |                                msg
-------+--------+--------+-------------------------------------------------------------
   563 |      1 |        | line pointer to page offset 19543 is not maximally aligned
   563 |      2 |        | line pointer redirection to item at offset 5585 exceeds maximum offset 4
   563 |      4 |        | line pointer redirection to item at offset 10285 exceeds maximum offset 4
(3 rows)
```

Corrupted Tuples

- Corrupted tuples mean random values in the columns
- Each tuple has header with system columns - 27 bytes
- Especially corrupted xmin, xmax and ctid are critical



Field	Type	Length	Description
t_xmin	TransactionId	4 bytes	insert XID stamp
t_xmax	TransactionId	4 bytes	delete XID stamp
t_cid	CommandId	4 bytes	insert and/or delete CID stamp (overlays with t_xvac)
t_xvac	TransactionId	4 bytes	XID for VACUUM operation moving a row version
t_ctid	ItemPointerData	6 bytes	current TID of this or newer row version
t_infomask2	uint16	2 bytes	number of attributes, plus various flag bits
t_infomask	uint16	2 bytes	various flag bits
t_hoff	uint8	1 byte	offset to user data

Corrupted Tuples

- 58P01 - could not access status of transaction 3047172894
- XX000 - MultiXactId 1074710815 has not been created yet – apparent wraparound
- ERROR: invalid memory alloc request size 18446744073709551594
- WARNING: Concurrent insert in progress within table ”test_table_bytea”

```
SELECT lp, lp_off, lp_flags, lp_len, t_xmin, t_xmax, t_field3, t_ctid, t_infomask2, t_infomask, t_hoff, t_bits, t_oid, substr(t_data::text,1,50) as t_data
FROM heap_page_items(get_raw_page('pg_toast.pg_toast_64234', 1655));

 lp | lp_off | lp_flags | lp_len | t_xmin | t_xmax | t_field3 |  t_ctid  | t_infomask2 | t_infomask | t_hoff | t_bits | t_oid |                    t_data
----+--------+----------+--------+--------+--------+----------+----------+-------------+------------+--------+--------+-------+----------------------------------------------
  1 |   6160 |        1 |   2032 |  29377 |      0 |        0 | (1655,1) |           3 |       2050 |     24 |        |       | \x9cfb00000c000000401f0000bf5bcb49edc7eab0ecaebc11
  2 |   4128 |        1 |   2032 |  29377 |      0 |        0 | (1655,2) |           3 |       2050 |     24 |        |       | \x9cfb00000d000000401f0000bd871bee922f63de0a19e7f1
  3 |   2096 |        1 |   2032 |  29377 |      0 |        0 | (1655,3) |           3 |       2050 |     24 |        |       | \x9cfb00000e000000401f000074d1fb7bde41c51d18322d85
  4 |     64 |        1 |   2032 |  29377 |      0 |        0 | (1655,4) |           3 |       2050 |     24 |        |       | \x9cfb00000f000000401f0000f913a066a35506e5c5293fee
(4 rows)

SELECT lp, lp_off, lp_flags, lp_len, t_xmin, t_xmax, t_field3, t_ctid, t_infomask2, t_infomask, t_hoff, t_bits, t_oid, substr(t_data::text,1,50) as t_data
FROM heap_page_items(get_raw_page('pg_toast.pg_toast_64234', 1654));

 lp | lp_off | lp_flags | lp_len |   t_xmin   |   t_xmax   |  t_field3   |       t_ctid       | t_infomask2 | t_infomask | t_hoff | t_bits | t_oid |                    t_data
----+--------+----------+--------+------------+------------+-------------+--------------------+-------------+------------+--------+--------+-------+----------------------------------------------
  1 |   6160 |        1 |   2032 |  534805867 |  873710472 | -1291348379 | (2395715003,28289) |       19277 |      32958 |     53 |        |       | \xfa80109a61a85e45de4ab580a26
  2 |   4128 |        1 |   2032 | 1026820500 | 1776295209 |  2142400425 | (1412156050,62855) |       11710 |      65140 |    253 |        |       | \xb9ebb3e852ca77a5195bc768469
  3 |   2096 |        1 |   2032 | 1509174647 | 1612155862 |   -79064812 | (3682366356,39443) |        2403 |      63137 |     97 |        |       | \x7e18ac3b8504440b7938764232
  4 |     64 |        1 |   2032 |      29377 |          0 |           0 | (1654,4)           |           3 |       2050 |     24 |        |       | \x9cfb00000b000000401f000085
(4 rows)
```

Corrupted Tuples of Toast Table

credativ

- Damaged TOAST table causes additional errors in selects from the main table

- XX000 - unexpected chunk number 19 (expected 16) for toast value 29685 in pg_toast_29580
- XX000 - unexpected chunk number 1873032786 (expected 4) for toast value 29595 in pg_toast_29580
- XX000 - unexpected chunk number -556107646 (expected 20) for toast value 29611 in pg_toast_29580
- XX000 - found toasted toast chunk for toast value 29707 in pg_toast_29580

Dealing with Corrupted Tuples

- Even a single corrupted tuple can prevent selects from the entire table
- Corruption in xmin or xmax system columns causes query failures
- If table is huge, salvaging data row by row would take extremely long time
- Use the pg_surgery extension to freeze or remove corrupted tuples
- Proper identification of damaged tuples is critical for precise actions

```
-- look at the tuples in the damaged page with pageinspect:
SELECT lp, lp_off, lp_flags, lp_len, t_xmin, t_xmax, t_field3, t_ctid, t_infomask2, t_infomask, t_hoff, t_bits, t_oid
FROM heap_page_items(get_raw_page('pg_toast.pg_toast_32840', 273));

 lp | lp_off | lp_flags | lp_len |   t_xmin   |   t_xmax   |  t_field3  |       t_ctid       | t_infomask2 | t_infomask | t_hoff | t_bits | t_oid
----+--------+----------+--------+------------+------------+------------+--------------------+-------------+------------+--------+--------+-------
  1 |   6160 |        1 |   2032 | 1491852297 |  287039843 |  491133876 | (3637106980,61186) |       50867 |      46441 |    124 |        |
  2 |   4128 |        1 |   2032 | 3846288155 | 3344221045 | 2002219688 | (2496224126,65391) |       34913 |      32266 |     82 |        |
  3 |   2096 |        1 |   2032 | 1209990178 | 1861759146 | 2010821376 | (426538995,32644)  |       23049 |       2764 |    215 |        |
  4 |     64 |        1 |   2032 |       4850 |          0 |          0 | (273,4)            |           3 |       2306 |     24 |        |
(4 rows)


-- corrupted ctid (3637106980,61186) in the damaged page 273 does not work:
-- we cannot use it to identify the tuple + corrupted number can even point to some other existing page!
SELECT * FROM heap_force_freeze('pg_toast.pg_toast_32840'::regclass, ARRAY['(3637106980,61186)']::tid[]);
NOTICE:  skipping block 3637106980 for relation "pg_toast_32840" because the block number is out of range


-- we have to use proper ctid of the tuple in the damaged page 273:
test=# select * from heap_force_freeze('pg_toast.pg_toast_32840'::regclass, ARRAY['(273,1)']::tid[]);
 heap_force_freeze
-------------------

(1 row)
```

Dealing with Corrupted Tuples

- Frozen tuples are visible to all transactions, they do not block selects anymore
- But they still contain corrupted data - queries will return garbage
- Best strategy is to delete them with ”heap_force_kill” function

```
-- let's look at the frozen tuple in the damaged page again:
SELECT lp, lp_off, lp_flags, lp_len, t_xmin, t_xmax, t_field3, t_ctid, t_infomask2, t_infomask, t_hoff, t_bits, t_oid
FROM heap_page_items(get_raw_page('pg_toast.pg_toast_32840', 273));

 lp | lp_off | lp_flags | lp_len |   t_xmin   |   t_xmax   |  t_field3  |       t_ctid       | t_infomask2 | t_infomask | t_hoff | t_bits | t_oid
----+--------+----------+--------+------------+------------+------------+--------------------+-------------+------------+--------+--------+-------
  1 |   6160 |        1 |   2032 |          2 |          0 |          2 | (273,1)            |       34483 |       2825 |    124 |        |
  2 |   4128 |        1 |   2032 | 3846288155 | 3344221045 | 2002219688 | (2496224126,65391) |       34913 |      32266 |     82 |        |
  3 |   2096 |        1 |   2032 | 1209990178 | 1861759146 | 2010821376 | (426538995,32644)  |       23049 |       2764 |    215 |        |
  4 |     64 |        1 |   2032 |       4850 |          0 |          0 | (273,4)            |           3 |       2306 |     24 |        |
(4 rows)


-- let's delete the corrupted tuple with pg_surgery:
SELECT * FROM heap_force_kill('pg_toast.pg_toast_32840'::regclass, ARRAY['(273,1)']::tid[]);
 heap_force_kill
-----------------

(1 row)


-- let's look at the page again:
 lp | lp_off | lp_flags | lp_len |   t_xmin   |   t_xmax   |  t_field3  |       t_ctid       | t_infomask2 | t_infomask | t_hoff | t_bits | t_oid
----+--------+----------+--------+------------+------------+------------+--------------------+-------------+------------+--------+--------+-------
  1 |      0 |        3 |      0 |            |            |            |                    |             |            |        |        |
  2 |   4128 |        1 |   2032 | 3846288155 | 3344221045 | 2002219688 | (2496224126,65391) |       34913 |      32266 |     82 |        |
  3 |   2096 |        1 |   2032 | 1209990178 | 1861759146 | 2010821376 | (426538995,32644)  |       23049 |       2764 |    215 |        |
  4 |     64 |        1 |   2032 |       4850 |          0 |          0 | (273,4)            |           3 |       2306 |     24 |        |
(4 rows)
```

Diagnostics with amcheck extension

- amcheck's "verify_heapam" function used on heap page with corrupted tuples
- Output messages even more cryptic and difficult to interpret
- Corrupted xmin/xmax fields cause errors, not flagged as suspicious
- Invalid tuple offsets and infomask values are reported as-is, not validated

```
-- Example 1:
 blkno | offnum | attnum |                                                  msg
-------+--------+--------+-------------------------------------------------------------------------------------------------------
   273 |      2 |        | multixact should not be marked committed


-- Example 2:
   586 |      2 |        | tuple data should begin at byte 232, but actually begins at byte 12 (1628 attributes, has nulls)
   586 |      3 |        | tuple data should begin at byte 24, but actually begins at byte 86 (2014 attributes, no nulls)


-- Example 3:
  1102 |      1 |        | tuple is heap only, but not the result of an update


-- Example 4 - call fails with an error:
ERROR:  MultiXactId 444928887 has not been created yet -- apparent wraparound
```

Corrupted Sequence

- Sequence is non-transactional table with 1 data block and 1 row
- Row has hidden system columns and 3 normal columns, one of them ”last_value”

- SELECT nextval('<sequence>');
- Corrupted Header -> ERROR: invalid page in block 0 of relation base/16384/64228
- Corrupted Item IDs Array / Tuples:
  - ERROR: bad magic number in sequence ”<sequence>”: 00000017
  - ERROR: invalid memory alloc request size 18446744073709551477

- SELECT * FROM <sequence>;
- Corrupted Header -> ERROR: invalid page in block 0 of relation base/16384/64228
- Corrupted Item IDs Array / Tuples -> returns 0 rows

Memory Segmentation Fault

- Corrupted Item IDs Array or Tuples can trigger memory allocation errors
- In some cases corruption may cause backend session to crash with segmentation fault
- Crashes are handled by postmaster: ”HandleChildCrash” and ”LogChildExit” functions
- Root cause: invalid offsets or lengths in ItemIDs or tuple headers exceed page size
- Lack of low-level sanity checks allows corrupted data to cause memory errors
- Some new basic sanity checks would be very useful here

```
if (!EXIT_STATUS_0(exitstatus))
    activity = pgstat_get_crashed_backend_activity(pid,
                                                   activity_buffer,
                                                   sizeof(activity_buffer));
...

ereport(lev,
/*------
    translator: %s is a noun phrase describing a child process, such as
    "server process" */
        (errmsg("%s (PID %d) was terminated by signal %d: %s",
                procname, pid, WTERMSIG(exitstatus),
                pg_strsignal(WTERMSIG(exitstatus))),
            activity ? errdetail("Failed process was running: %s", activity) : 0));
```

Summary: Can we cure corrupted data?

- Without a valid backup, original data cannot be restored
- Without checksums, only pages with invalid headers are detected
- ”zero_damaged_pages = ON” zeros out unreadable pages
- Corrupted data blocks contain unpredictable values
- Only good data can be salvaged; corrupted data must be removed
- Corrupted tuples might require precise manual removal
- Row-by-row salvage is slow and labor-intensive

Salvaging Good Data Row by Row using PRIMARY KEY

- Without checksums, salvage of good data must be done row by row
- Script assumes PRIMARY KEY "id" is readable for all rows (damage in TOAST table)
- If session crashes with segmentation fault, we must skip problematic ids

```
do $$
declare
    i record;
    d record;
begin
for i in (select id from public.test_table_bytea) loop
    begin
        select * into d from public.test_table_bytea where id=i.id;
        insert into public.test_table_bytea_new values (...) values (d....);
        commit;
    exception when  others then
        raise notice 'id=%', i.id
        insert into public.test_table_bytea_broken_ids (id) values (i.id);
    end;
end loop;
end;
```

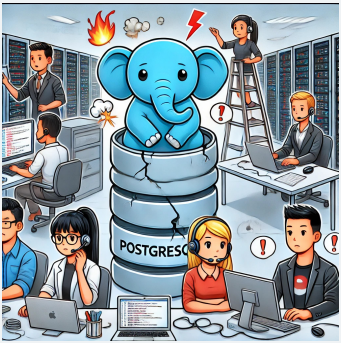
Salvaging Good Data Row by Row using ctid

- If some rows are unreadable in a table, must attempt to read single tuples using ctid
- If session crashes with segmentation fault, we must skip problematic ctids

```
do $$
declare
    r record;
    query text;
begin
    for data_page in 0..MAX_PAGES loop
        for tuple in 0..MAX_TUPLES loop
            begin
                query = 'select * from public.mybigtable where ctid = ''('||data_page||','||tuple||')''';';
                execute query into r;
                if r.id is null then
                    -- if record not found skip to next tuple number
                    -- to uncover all readable tuples on page
                    continue;
                end if;
                insert into public.mybigtable_salvaged values (r.id, r.col_int);
                commit; -- to not lose already salvaged data in case of crash
            exception when others then null; -- ignore errors
            end;
        end loop;
    end loop;
end;
$$ language plpgsql;
```

How to repair other corrupted objects

- If some data are lost, foreign keys might be compromised
- Either orphaned rows in child tables must be removed
- Or dummy rows in parent table must be added
- Damaged indexes can be rebuilt using REINDEX if table is intact
- Corrupted sequences must be dropped and recreated
- Set sequence ”last_value” based on current table data
- System catalog corruption requires new cluster installation



How to Make Repair of Corrupted Data Easier?

- Sanity checks in the code to prevent ”Segmentation Fault” crashes
- Validate ItemIDs: offset and length must fit page size
- Ensure ctid values match expected page and tuple ranges
- Huge xmin/xmax values are suspicious or impossible by definition
- Such values shall not be taken seriously as a valid transaction ID
- Strict checks could be enabled optionally to not impact performance
- Or emergency mode to skip rows with suspicious xmin/xmax/ctid ?
- Queries and pg_dump must be able to skip such problematic records
- Ridiculous content of system columns means tuples are lost anyway

Questions?