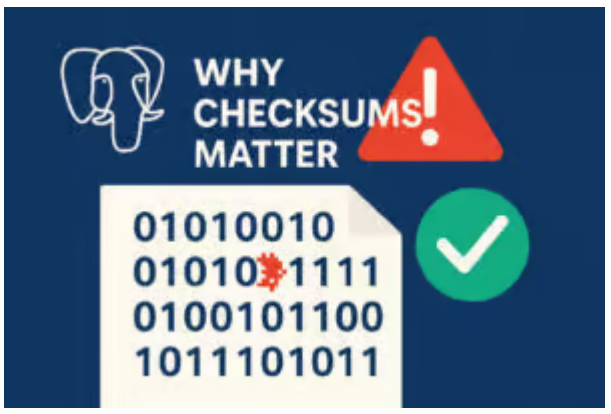
03 NOVEMBER 2025

PostgreSQL 18 enables data-checksums by default

As I explained in my talk on PostgreSQL Conference Europe 2025, data corruption can be silently present in any PostgreSQL database and will remain undetected until we physically read corrupted data. There can be many reasons why some data blocks in tables or other objects can be damaged. Even modern storage hardware is far from being infallible. Binary backups done with pg_basebackup tool – which is very common backup strategy in PostgreSQL environment – leave these problems hidden. Because they do not check data but copy whole data files as they are. With release of PostgreSQL 18, the community decided to turn on data-checksums by default – a major step toward early detection of these failures. This post examines how PostgreSQL implements checksums, how it handles checksum failures, and how we can enable them on existing clusters.

Why checksums matter



A PostgreSQL table or index is stored in 8 KB pages. When a page is written to disk, PostgreSQL computes a 16-bit checksum using every byte of the page (except the checksum field itself) and the page's physical block address. The checksum is stored in the page header. On every read, PostgreSQL recalculates the checksum and compares it against the stored value. Because the block address is part of the calculation, the system detects both bit flips within the page and pages written to the wrong place. Checksums are not maintained while the page sits in shared buffers – they are computed only when the page is flushed from the buffer cache to the operating system page cache. Consequently, an incorrect in-memory page cannot be detected until it is written and read again. PostgreSQL uses a fast FNV-1a hash (with CRC32C on WAL records) that is optimized for performance. On typical hardware the cost of calculating checksum seems to be small. A benchmarking studies found the penalty is usually less than 2 % for normal workloads. PostgreSQL 18's release notes acknowledge that the overhead is non-zero but accept it for the benefit of data integrity.

Changes in PostgreSQL 18

Version 18 enables data-checksums by default. In earlier versions, initdb required the –data-checksums flag. The new release notes explicitly list the change in the incompatibilities section: "Change initdb default to enable data checksums... Checksums can be disabled with the new –no-data-checksums option".
For DBAs this default change has two important consequences:

- **Cluster upgrades must match checksum settings (explicitly mentioned in PostgreSQL 18 release notes). When upgrading via pg_upgrade, the source and target clusters must both have checksums enabled or disabled. If you need to upgrade from an older cluster without checksums, initialise the new cluster with –no-data-checksums or enable checksums on the old cluster first.**
- **Statistics to monitor failures – PostgreSQL already has two columns in pg_stat_database: checksum_failures, counting the number of pages whose checksums failed, and checksum_last_failure, the timestamp of the most recent failure. These metrics allow you to alert on corruption events across all databases in the cluster.**

To see whether our cluster uses data-checksums, we shall inspect the read-only system variable data_checksums using command: "SHOW data_checksums;" A result of "ON" means that data-page checksums are active.

Enabling and disabling checksums with pg_checksums

Checksums are a cluster-wide property and cannot be toggled while the server is running. PostgreSQL ships the pg_checksums utility to check, enable or disable checksums. Key points from the documentation:

- The cluster must be shut down cleanly before running pg_checksums.
- Verifying checksums (–check) scans every file in PGDATA and returns a non-zero exit code if any mismatch is found.
- Enabling checksums (–enable) rewrites each relation block, updating the checksum field on disk. Disabling checksums (–disable) only updates the control file – it does not rewrite pages.
- Options such as –progress display progress, –no-sync skips fsync after modifications, and –filenode restricts verification to a specific relation.
- On large or replicated clusters, enabling checksums can take a long time; all standbys must be stopped or recreated so that all nodes maintain the same checksum state (explicitly mentioned in documentation).

Upgrade strategy

If you we upgrading a pre-18 cluster without checksums, we have two options:

- Disable checksums on the new cluster: run initdb with –no-data-checksums so that pg_upgrade allows the migration. After the upgrade you can enable checksums offline using pg_checksums –enable.
- Enable checksums on the old cluster first: shut down the old server, run pg_checksums –enable -D $PGDATA (on every node if using streaming replication), then start the server and verify the new SHOW data_checksums value. When you initialise PostgreSQL 18, it will inherit the enabled state.

Handling checksum failures

When PostgreSQL detects a checksum mismatch, it issues a warning and raises an error. Two developer-only GUCs control what happens next. They should never be enabled in normal operation, but DBAs may use them for data recovery:

- *ignore_checksum_failure* – When off (default), the server aborts the current transaction on the first checksum error. Setting it to on logs a warning and continues processing, allowing queries to skip over corrupted blocks. This option may hide corruption, cause crashes or return incorrect data; only superusers can change it.
- *zero_damaged_pages* – When a damaged page header or checksum is detected, setting this parameter to on causes PostgreSQL to replace the entire 8 KB page in memory with zeroes and then continue processing. The zeroed page is later written to disk, destroying all tuples on that page. Use this only when you have exhausted backup or standby options. Turning zero_damaged_pages off does not restore data and only affects how future corrupt pages are handled.

The following simplified examples illustrate these settings:

```
-- With ignore_checksum_failure=off the query stops on the first 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

-- With ignore_checksum_failure=on, the server logs warnings and continues scanning until it find good data:
test=# SET ignore_checksum_failure = ON;
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
```

With zero_damaged_pages=on, invalid pages are zeroed out rather than causing an error. The query continues, but the data on those pages is lost:

```
test=# SET zero_damaged_pages = ON;
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
```

Internally the buffer manager performs this zeroing by calling memset() on the 8 KB page when the verification fails and the READ_BUFFERS_ZERO_ON_ERROR flag is set. If the flag is not set, the buffer is marked invalid and an error is thrown. We must of course understand, that checksums and ignore_checksum_failure and zero_damaged_pages settings cannot repair damages data blocks. These options are last resorts for salvaging remaining rows. Their usage will always lead to data loses. Once page is zeroed out in the memory, its previous corrupted content cannot be restored, even if we set zero_damaged_pages back to OFF. To get original good data back we must restore them from a known good backup or standby.

Autovacuum interaction

Vacuum processes may encounter corrupted pages while scanning tables. Because automatically zeroing pages could silently destroy data, the autovacuum launcher forcibly disables zero_damaged_pages for its workers. The source code calls SetConfigOption with “zero_damaged_pages”, “false” with a comment explaining that this dangerous option should never be applied non-interactively. This way corrupted pages will be zeroed out only when we directly work with them.

Why we shall embrace checksums

Data corruption on the database which does not use checksums can lead to much more problematic situations. Without checksums only pages with clearly damaged page header can be detected and zeroed out. Below we can see test in the PostgreSQL code, which shows that even this detection is not easy without checksums – see the comment:

```
/*
* 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;
```

Generally this code tests if important values in the page header fit into expected relationships of their values. Healthy data page is shown here:

```
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)
```

So, only page header with clearly damaged flag bits, lower, upper, special and/or pagesize can be safely detected as corrupted. In that case we will get an error message:

```
ERROR: XX001-invalid page in block 578 of relation base/16384/28751
```

And only these pages can be zeroed out. But if header is intact (or at least passes the test above), we can get many different errors, which are caused either by damaged Item IDs array or damaged system columns in tuples.
Damaged Item IDs array will contain wrong offsets to the beginning of tuple and wrong length of tuple. These corrupted numbers can cause invalid memory allocation request or even crash of the session reading data:

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

If Item IDs array values are intact, but tuples are corrupted, we usually see different errors signalizing that system columns xmin and xmax, which are crucial for check of visibility in multiversion concurrency control system, contain useless values:

```
58P01 - could not access status of transaction 3047172894
XX000 - MultiXactId 1074710815 has not been created yet -- apparent wraparound
WARNING:  Concurrent insert in progress within table "test_table_bytea"
```

With these errors, we can face difficult and time consuming manual repairs and data salvage if we do not have reliable backup which we could use for restoring data. These descriptions clearly show that enabling data checksums is a very important change for PostgreSQL community.

Conclusion

PostgreSQL 18's decision to enable data-page checksums reflects experience showing that the performance impact is minimal and the benefits enormous. Checksums detect a wide range of silent corruption events so we can easier diagnose cases when hardware goes awry. They also make salvage of good data much quicker and easier – if for any reason reliable backups are not available.
Read more:

- PostgreSQL checksums & base backup

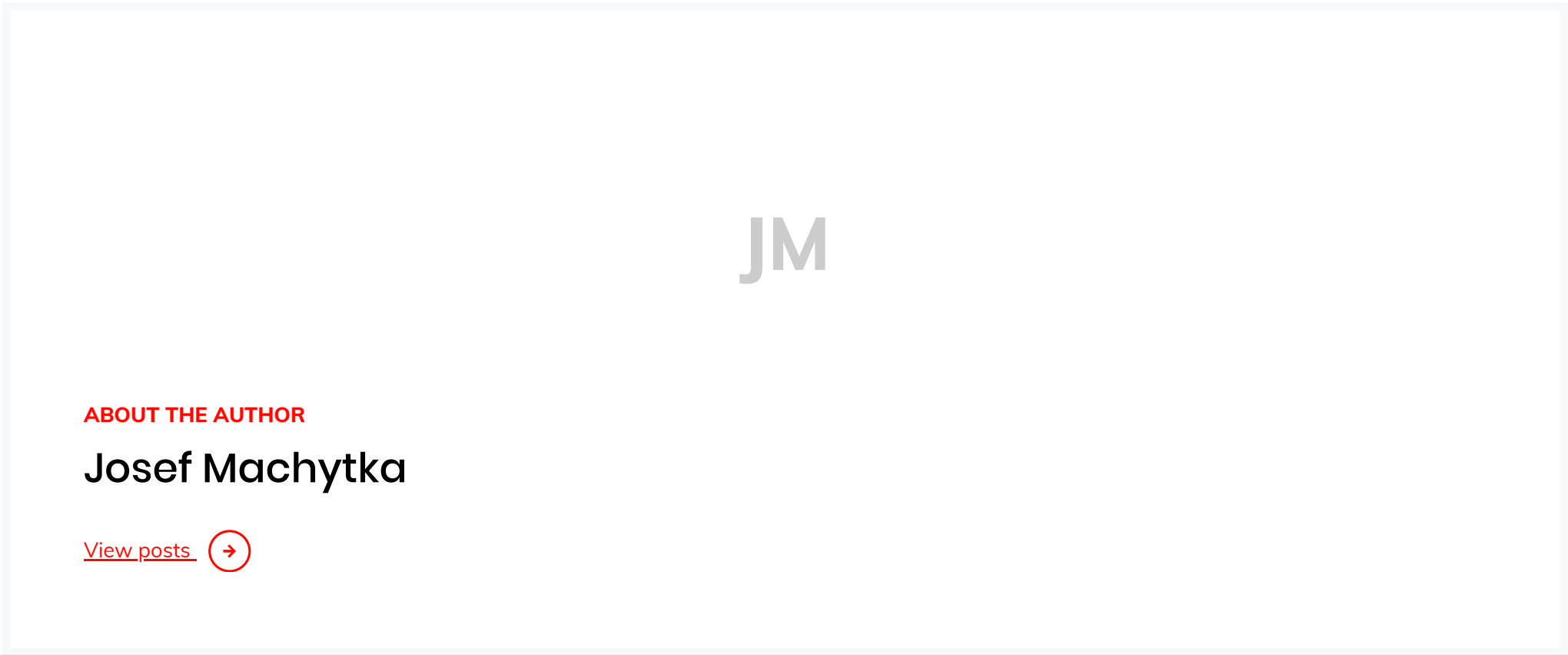
We are happy to help!

Whether it's Ansible, Debian, Proxmox, Kubernetes or **PostgreSQL**, with over 25+ years of development and service experience in the open source space, credativ GmbH can assist you with unparalleled and individually customizable support. We are there to help and assist you in all your open source infrastructure needs.

Do you have any questions about our article or would you like credativ's specialists to take a look at another software of your choice?
Then stop by and get in touch via our contact form or drop us an email at info@credativ.de.

About credativ

The credativ GmbH is a manufacturer-independent consulting and service company located in Moenchengladbach, Germany.



ABOUT THE AUTHOR

Josef Machytka

View posts