CYBERTEC
POSTGRESQL SERVICES & SUPPORT

# How to corrupt your database (and how to deal with data corruption)

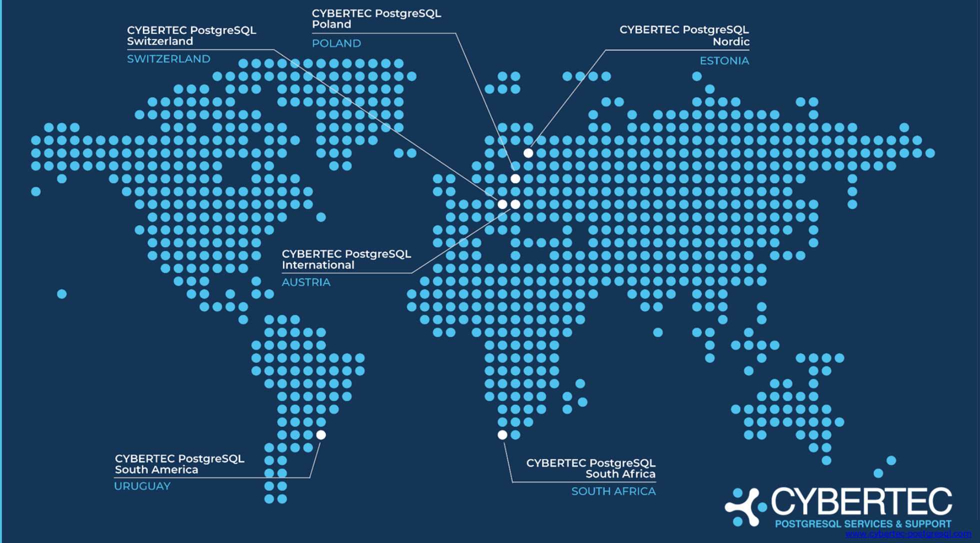
Laurenz Albe

www.cybertec-postgresql.com

## Senior Consultant
## Laurenz Albe

| | |
|---|---|
| MAIL | laurenz.albe@cybertec.at |
| PHONE | +43 670 605 6265 |
| WEB | www.cybertec-postgresql.com |

CYBERTEC
POSTGRESQL SERVICES & SUPPORT


CYBERTEC PostgreSQL Switzerland
SWITZERLAND
CYBERTEC PostgreSQL Poland
POLAND
CYBERTEC PostgreSQL Nordic
ESTONIA
CYBERTEC PostgreSQL International
AUSTRIA
CYBERTEC PostgreSQL South America
URUGUAY
CYBERTEC PostgreSQL South Africa
SOUTH AFRICA
CYBERTEC
POSTGRESQL SERVICES & SUPPORT
www.cybertec-postgresql.com

CYBERTEC POSTGRESQL SERVICES & SUPPORT www.cybertec-postgresql.com

## CUSTOMER BUSINESS SECTORS

- ICT
- Universities
- Government
- Automotive
- Industrial
- Trade
- Financial Services
- etc.

## Introduction

## What I mean by “data corruption”

- data that cause an error of class `XX`
  - `XX000` (`internal_error`)
  - `XX001` (`data_corrupted`)
  - `XX002` (`index_corrupted`)
- data that cause PostgreSQL to crash
- inconsistent data: constraints are violated, rows are not indexed, . . .
- bad data values that were *not* caused by the user

`DROP TABLE` or `DELETE` may wreck your database, but won’t corrupt it.

## Causes of data corruption

- bad hardware (faulty disk, bad memory, storage that lies)
- bad software (unreliable file system, PostgreSQL bugs)
- bad administrators (e.g., taking forbidden shortcuts)

  PostgreSQL makes it as hard as possible to break the database, but it is not perfect.

This talk is mostly about the last of these options (but the techniques for dealing with corruption apply to all of them).

## Data corruption caused by `fsync = off`

## What is `fsync`?

- PostgreSQL uses “buffered I/O”
  - writes don’t go to disk right away, but are buffered by the kernel
- data have to be forced out to disk before the end of a checkpoint, during a `COMMIT` and at other times
- necessary to prevent data loss and inconsistency after crash (the Write Ahead Log must be written before the data)
- PostgreSQL uses `fsync` or related system calls at the appropriate time (the exact method can be configured with `wal_sync_method`)

## What happens if `fsync` is `off`?

- fewer I/O requests, faster data modifications, better performance (2.5 times more transactions per second in a simple `pgbench` run)
- as long as the operating system doesn’t crash, everything is fine
- if there is a crash, your data will probably be corrupted

Don’t disable `fsync`. Instead, set `synchronous_commit = off`. The performance gain will be almost the same, and there will be no data corruption, only some lost transactions.

## Data corruption caused by a bad backup

## Understanding file system backups

- a copy of all the files in the data directory
- safe if the database is shut down
- problem with online backups: the files are modified while being copied
  $\Rightarrow$ backup is inconsistent
- after restore, recovery must be used to replay modifications **from the start of the backup**
- the `backup_label` file contains information about the checkpoint from which WAL has to be replayed
- **`backup_label` is the only way to tell an online file system backup from a crashed database server**

## How a bad backup can corrupt your database

- obviously, only if you restore the backup (that’s why you test restore)
- if `backup_label` is missing, PostgreSQL will recover from the checkpoint in the control file, which is probably a later checkpoint
  $\Rightarrow$ recovery will either fail or replay too little WAL
- as a consequence, the files will not be consistent:
  - index and table may not be synchronized
  - foreign keys may be violated
  - committed transactions are undone (not yet in the commit log)

## Reasons for a missing `backup_label`

- you don’t bother about correct backup and just backup all the files
- you deliberately remove `backup_label` from a backup
- you use the “non-exclusive low-level backup API” and don’t create `backup_label` from the result of `pg_backup_stop()`

The “exclusive low-level backup API” (which created `backup_label` automatically) has its own problems and was removed in PostgreSQL v15, so it has become easier to create a corrupted backup with the third method.

If you want to avoid this problem, use `pg_basebackup` or some third-party backup software (pgBackRest, Barman, pg_probackup, . . . ).

## Data corruption caused by `pg_resetwal`

## What is `pg_resetwal`?

- `pg_resetwal` is a tool for experts to deal with data corruption
- it is a last-ditch effort to get a server to start **and will typically cause data loss or data corruption**
- safe to use **only** after a clean shutdown and with `--wal-segsize` to change the WAL segment size
- if you get this message:

```
The database server was not shut down cleanly.
Resetting the write-ahead log might cause data to be lost.
If you want to proceed anyway, use -f to force reset.
```

**don't do it unless you know what you are doing**

## How to corrupt your database with `pg_resetwal`

- pretty simple: just run it with `-f` on a cluster that was not shut down properly
- easier if you don’t read the documentation before using `pg_resetwal`
- people tend to do that if there are problems during startup
- I have also seen people do it if there are problems with WAL archiving or replication – anything that has to do with WAL
- perhaps this also appeals to people who are used to databases where tools to fix data corruption are routinely used (MySQL)

## Data corruption from `pg_upgrade --link`

## Starting two servers on the same data directory

- this would lead to data corruption
- PostgreSQL has safeguards against that
    - `postmaster.pid` contains the running postmaster process ID; the server will refuse to start if there is already a process with that ID running
    - A small shared memory segment serves as an additional lock
- we can remove both `postmaster.pid` and the shared memory segment and start a new server
- but we have to be quick, because the postmaster regularly checks `postmaster.pid` and will die with the message

```
performing immediate shutdown because
data directory lock file is invalid
```

## Data corruption with `pg_upgrade --link`

- data files are not copied, but shared via a “hard link”: the file (inode) exists only once, but is listed in both data directories
- this makes upgrades very fast
- **you must remove the old cluster** after `pg_upgrade`
- if you start both the old and the new server (one after the other or at the same time), you end up with data corruption
- PostgreSQL tries to prevent that by renaming the control file on the old cluster to `pg_control.old`
- rename the file back to `pg_control` and start both servers!

## Data corruption from messing with the data directory

## Messing with `pg_wal`

- the most popular way is to remove files from `pg_wal`
  (this happened more often back when it was called `pg_xlog`)
- people tend to do that when the disk is full and PostgreSQL crashed
- `pg_wal` is likely to fill up if
  - the archiver has problems
  - a standby server using a replication slot is down
- manually deleting the WAL segments in a crashed server is a sure way to break PostgreSQL (it cannot recover from the crash)

## Data corruption by messing with the catalogs

## Messing with the catalogs

- this is always a good idea for corruption
- example: drop a column without `ACCESS EXCLUSIVE` lock

```
DELETE FROM pg_attribute
WHERE attrelid = 'pgbench_accounts'::regclass
  AND attname = 'bid';
```

- example: change the data type without rewriting the table

```
UPDATE pg_attribute
SET atttypid = 'bigint'::regtype
WHERE attrelid = 'pgbench_accounts'::regclass
  AND attname = 'bid';
```

## Dealing with data corruption

## Fundamental advice for data corruption

- don’t continue working with the corrupted database
  (corruption can become worse, and any new work may be lost)
- if you have a good backup that you can restore, do that
- shut down PostgreSQL and take a cold file system backup
  (dealing with corruption often destroys data)
- after fixing corruption, always dump/restore the data to a fresh cluster
  (otherwise, invisible corruption may be left behind)
- investigate the cause and avoid it in the future

## Backups against data corruption

## Protect yourself with backups

- you are doing that anyway, right?
- a backup that isn’t monitored is no backup (e.g., `pg_dump` failing repeatedly because of data corruption)
- `pg_dump` is better than a file system backup (if the dump restores, all data corruption is automatically fixed)
- a file system level backup can be used for point-in-time-recovery to minimize data loss (but you usually won’t notice if the backup contains corruption)
- best to do both backups: a regular file system backup and an occasional `pg_dump`

## Recovering with a backup

- restoring the last good backup means to cut your losses
- requires no expert knowledge
- if you cannot find other ways to deal with the data corruption, that is your only way forward (or backward?)
- when considering other techniques, proceed as follows
  - decide on a time and expense limit for efforts to repair corruption
  - get an assessment by an expert (will involve guesswork)
  - stop your efforts when the agreed limit is reached (don’t throw good money after bad)

## Index corruption

## Index corruption: causes and symptoms

- index is inconsistent or data in table and index are out of sync
- often caused by things that should be immutable, but aren't (functions, collations after an operating system update, ...)
- results are different, depending on whether an index scan or a sequential scan are used (play with `enable_indexscan` and friends)
- use `amcheck` contrib extension to check indexes:
  - `bt_index_check` checks internal consistency (with `heapallindexed => TRUE`, also checks if all rows are indexed)
  - `bt_index_parent_check` performs a thorough check, but requires a `SHARE` lock (no concurrent data modifications)

## How to deal with index corruption

- this is usually simple, since index data are redundant:

  ```
  REINDEX INDEX CONCURRENTLY broken_index;
  ```

- if that fails, there is also data corruption (perhaps caused by the index corruption)
- If a catalog index is broken:
  - start the server with `-P` (ignore system indexes)
  - rebuild the index as superuser
  - restart the server without `-P`

## Data corruption that causes no errors

## Symptoms of corruption without errors

- bad values
- missing data, can be caused by
  - rows that become invisible
  - blocks that are all-zero (are treated as empty)
  - files that have been truncated
- foreign keys that are violated (perhaps somebody disabled the constraint in the past)
- duplicate data that violate uniqueness constraints (often caused by index corruption)

## Dealing with error-less data corruption

- this is fairly easy
- `pg_dump` the database, restore to a new cluster
- if there are errors during the restore (constraint cannot be created), delete/add data manually until they are consistent
- we will try to reduce all of the more difficult cases to this one by making the errors go away
  - remember that you should not continue working with a database that had data corruption

## Data corruption that causes errors but no crash

## Possible causes of errors

- checksum failure in data blocks
  - only if cluster was created with checksums (additional cost, but you can detect **storage-induced** corruption early on)
- block header is corrupted, for example:

```
invalid page in block 4711 of relation 183200
```

- table row is corrupted, for example:

```
found xmin 16804535 from before relfrozenxid 90126924
invalid memory alloc request size 18446744073709551613
could not access status of transaction 808464919
```

- TOAST data are corrupted, for example:

```
missing chunk number 0 for toast value 171568 in pg_toast_80762
```

## Dealing with corrupted blocks

- you can set `ignore_checksum_failure = on` to ignore checksum failures (but garbage in the block will still cause errors)
- you can set `zero_damaged_pages = on` to have PostgreSQL consider pages with a corrupted header as empty
  - won’t help if the header is ok, but rows are damaged
- those settings won’t change the data on disk, they are only useful to avoid errors so that you can run `pg_dump`
- we’ll look at salvaging data from corrupted blocks later

## Identifying TOAST corruption

- oversized column values are stored “out of line” in the TOAST table
- the actual table row contains a “TOAST pointer”
- the typical symptom is an error message with “`toast`” in it
- `SELECT` of the row works as long as you don’t select the column that points to the broken TOAST record

## Fixing TOAST corruption

- fixing is easy: `DELETE` the row or `UPDATE` the broken column to a different value
- identify the broken rows with something like

```
DO $$
DECLARE t tid;
        x text;
BEGIN
   FOR t IN SELECT ctid FROM badtable LOOP
      BEGIN
         SELECT badcol INTO x FROM badtable WHERE ctid = t;
      EXCEPTION WHEN OTHERS THEN
         RAISE NOTICE 'ctid = %', t;
      END;
   END LOOP;
END;$$;
```

## Dealing with corrupted rows

- similar to TOAST, corruption, but we cannot find the primary key, so we have to try all possible keys

```
DO $$
DECLARE i bigint; max_id bigint;
BEGIN
   /* if that fails, guess a constant */
   SELECT max(id) INTO max_id FROM badtable;
   FOR i IN 1..max_id LOOP
      BEGIN
         /* make PostgreSQL read all data in the row */
         PERFORM badtable::text INTO r FROM badtable WHERE id = i;
      EXCEPTION WHEN OTHERS THEN
         RAISE NOTICE 'id = %', i;
      END;
   END LOOP;
END;$$;
```

- if `DELETE` doesn't work, `SELECT` everything except the bad rows into a new table

## Using `pg_surgery`

- new extension in PostgreSQL v14
- useful for dealing with errors where the row cannot be accessed, like

  ```
  found xmin 16804535 from before relfrozenxid 90126924
  ```
- function `heap_force_kill` to delete rows by `ctid`
- function `heap_force_freeze` to make rows visible by `ctid`
- another great tool to corrupt your database – handle with care

## Data corruption that causes a crash

## Why can data corruption cause crashes?

- for example, a bogus offset is added, which may lead to huge memory allocations or bad pointers
- normally, such a crash is a software bug (lack of defensive programming)
- however, checking everything all the time slows down processing
- if you build PostgreSQL with `--enable-cassert`, many more checks are enabled
- assert-enabled builds will still cause crashes, but provide more meaningful log messages
- on the other hand, data corruption may trigger assertions where a normal build would just work $\Rightarrow$ try both

## Dealing with data corruption that causes crashes

- the idea is again to identify the bad rows and copy everything else
- because of the crash, PL/pgSQL code won’t do, and we have to write client code
- the program has to re-establish the connection after a crash and continue
- more development effort, but still fairly straightforward

## Salvaging data from broken blocks or rows

- do that only if it is worth the effort
- try the `pageinspect` extension
  - `get_raw_page` to read a block
  - `heap_page_item_attrs` can extract data, but will usually fail in the face of corruption
- `pg_filedump`
  `https://git.postgresql.org/gitweb/?p=pg_filedump.git`
  - also likely to be confused by data corruption
- the last resort is "`od -t x1`" or a hex editor and knowledge about PostgreSQL internals

## Missing or empty files

## Causes for missing or empty files

- pilot error (administrator deleted files)
  - this is frequent with WAL segments (see above)
- disks that lie (PostgreSQL synced the file to disk, but it is not there after a crash)
- file system check after a hardware problem
- misconfigured anti-virus software
  - never let it run on the PostgreSQL data directory

## Dealing with missing files

- tables, indexes etc. are easy: `DROP` the objects
- missing WAL segments ⇒ `pg_resetwal`
    - read the documentation about guessing good values!
- if other files are missing, try faking them
- for example, to create a missing commit log file with 12 blocks will all transactions committed (so data are visible):

```
for (( i=0; i<8192*12; i++ )); do
    echo -e -n '\x55' >> pg_xact/0130
done
```

## Conclusion

## Key messages from the talk

- stay on the safe side, don't play with guns
- have good backups, ideally both physical and with `pg_dump`
- only fix corruption if you really have to (backup is better)
- take a cold file system backup before dealing with corruption
- use built-in tools: `pg_resetwal`, `pg_surgery`, `pageinspect`
- don't work with the repaired database: dump and restore

## Questions