CYBERTEC

POSTGRESQL SERVICES & SUPPORT

Wie erzeuge ich Datenkorruption (und wie repariere ich sie wieder)

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

KUNDEN BRANCHEN

- ICT
- Universitäten
- Regierungen
- Automotive
- Industrie
- Handel
- Finanzwesen
- uvm.

Einleitung

Was meine ich mit „Datenkorruption”?

- Daten, die einen Fehler der Klasse `XX` auslösen
  - `XX000` (`internal_error`)
  - `XX001` (`data_corrupted`)
  - `XX002` (`index_corrupted`)
- Daten, die PostgreSQL crashen lassen
- inkonsistente Daten: Constraintverletzung, nicht indexierte Zeilen, . . .
- falsche Werte, die *nicht* von Benutzer verursacht sind

`DROP TABLE` und `DELETE` können Daten zerstören, aber nicht korrumpieren.

Ursachen für Datenkorruption

- schlechte Hardware (fehlerhafte Disk/RAM)
- schlechte Software (Bugs im Filesystem oder in PostgreSQL)
- schlechte Administratoren (tun verbotene Dinge)
  PostgreSQL macht es schwer, die Daten zu zerstören, aber es ist nicht perfekt.

Dieser Vortrag beschäftigt sich mit dem letzten Punkt (aber der Umgang mit Datenkorruption is für alle gleich).

Datenkorruption durch `fsync = off`

Was ist `fsync`?

- PostgreSQL verwendet (noch) „Buffered I/O”
  - schreibt Daten nicht direkt auf die Disk; der Kernel hält sie im Cache
- Daten müssen beim Checkpoint, beim `COMMIT` und zu anderen Gelegenheiten auf die Disk gezwunden werden
- notwendig, um Datenverlust und Inkonsistenz nach Crash zu vermeiden (das „Write-Ahead Log” muss vor den Daten geschrieben werden)
- PostgreSQL verwendet dafür `fsync` oder verwandte System Calls (die konkrete Methode wird mit `wal_sync_method` eingestellt)

Was passiert, wenn `fsync = off`?

- weniger I/O-Requests, schnellere Datenänderungen, bessere Performance
  (ein einfacher Versuch mit `pgbench` ergibt 2,5-mal mehr TPS)
- solange das Betriebssystem nicht abstürzt, ist alles in Ordnung
- nach einem Absturz sind die Daten wahrscheinlich korrupt

`fsync` niemals deaktivieren! Stattdessen `synchronous_commit = off` setzen. Die Performance ist fast ebensogut, und statt Datenkorruption hat man nur ein paar verlorene Transaktionen.

Datenkorruption durch schlechte Sicherung

Sicherungen auf Filesystemebene verstehen

- eine Kopie aller Dateien des Datendirectorys
- konsistent, wenn PostgreSQL heruntergefahren ist
- Problem bei der Online-Sicherung: Dateien verändern sich beim Kopieren
  $\Rightarrow$ die Sicherung ist inkonsistent
- nach dem Rücksichern müssen alle Änderungen **vom Beginn der Sicherung an** nachgespielt werden
- die Datei `backup_label` enthält die Information über den Checkpoint, von dem an WAL nachgespielt werden muss
- `backup_label` **ist der einzige Unterschied zwischen einer Online-Dateisicherung und einem abgestürzten Datenbankserver**

Datenbank mit schlechter Sicherung zerstören

CYBERTEC
POSTGRESQL SERVICES & SUPPORT

- natürlich nur dann, wenn die Sicherung wiederhergestellt wird (darum testet man die Wiederherstellung)
- wenn `backup_label` fehlt, spielt PostgreSQL WAL vom Checkpoint im Control File nach, aber das ist meistens ein späterer Checkpoint
  $\Rightarrow$ Wiederherstellung scheitert oder repariert nicht alle Daten
- in der Folge sind die Dateien inkonsistent:
  - Index und Tabelle sind nich synchron
  - Fremdschlüssel können verletzt sein
  - kommittierte Transaktionen fehlen (noch nicht im Commit-Log)

Gründe für ein fehlendes `backup_label`

- statt einer korrekten Sicherung werden einfach im Betrieb die Dateien gesichert
- `backup_label` wurde vorsätzlich entfernt
- das „Non-Exclusive Low-Level Backup API” wurde verwendet, aber kein `backup_label` wurde aus dem Ergebnis von `pg_backup_stop()` erstellt

Das „Exclusive Low-Level Backup API” (das `backup_label` automatisch erstellt) hatte Probleme und wurde in PostgreSQL v15 entfernt. Also ist es leichter geworden, mit dieser Methode eine korrupte Sicherung zu erstellen.

Um dieses Problem zu vermeiden, sollte man `pg_basebackup` oder eine spezialisierte Backup-Software verwenden (pgBackRest, Barman, pg_probackup, . . .).

Datenkorruption durch `pg_resetwal`

Was ist `pg_resetwal`?

- `pg_resetwal` ist ein Werkzeug für Experten, um Datenkorruption zu behandeln
- es macht einen kaputten Server startfähig **und verursacht normalerweise Datenverlust oder Datenkorruption**
- sicher **nur** nach sauberem Herunterfahren, um mit `--wal-segsize` die WAL-File-Größe zu verändern
- wenn man diese Meldung sieht:

```
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.
```

tu's nicht, außer du weißt genau, was du tust

Eine Datenbank mit `pg_resetwal` kaputtmachen

- das ist einfach: `pg_resetwal -f` auf einen nicht sauber heruntergefahrenen Cluster anwenden
- das ist besonders leicht, wenn man vor dem Gebrauch nicht die Dokumentation liest
- Leute tun das immer wieder gerne, wenn sie beim Starten Probleme haben
- ich habe auch schon Fälle gesehen, wo es bei Problemen mit WAL-Archivierung oder Replikation angewendet wurde – immer, wenn WAL im Spiel ist
- wahrscheinlich ist die Versuchung groß, wenn man von Datenbanksystemen kommt, wo Werkzeuge zur Behebung von Datenkorruption routinemäßig angewendet werden (MySQL)

Datenkorruption durch `pg_upgrade --link`

Zwei Server im selben Datendirectory starten

- das führt sofort zu Datenkorruption
- PostgreSQL hat Sicherungen, um das zu verhindern:
  - `postmaster.pid` enthält die Prozessnummer des Postmasters; der Server startet nicht, wenn es einen Prozess mit dieser Nummer gibt
  - ein kleines Shared-Memory-Segment dient als zusätzliche Sperre
- wir können `postmaster.pid` und das Shared-Memory-Segment entfernen und einen neuen Server starten
- aber wir müssen dabei schnell sein, denn der Postmaster überprüft regelmäßig `postmaster.pid` und stirbt mit

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

Datenkorruption durch `pg_upgrade --link`

- Dateien werden nicht kopiert, sondern stehen mittels „Hard Link” in beiden Verzeichnissen: die Datei (inode) existiert nur einmal
- das macht Upgrade sehr schnell
- **nach `pg_upgrade` muss man den alten Cluster entfernen**
- wenn man den alten und den neuen Cluster startet (gleichzeitig oder nacheinander), korrumpiert man die Datenbank
- PostgreSQL versucht das zu verhindern, indem es das Control File des alten Clusters in `pg_control.old` umbenennt
- einfach zurück in `pg_control` umbenennen und beide Server starten!

Datenkorruption durch Herumfummeln im Datendirectory

`pg_wal` verändern

- die beliebteste Methode ist das Löschen von Dateien aus `pg_wal`
  (das geschah früher noch öfter, als es noch `pg_xlog` hieß)
- Leute machen das gerne, wenn die PostgreSQL wegen voller Disk abgestürzt ist
- `pg_wal` läuft gerne voll, wenn
  - der Archivier-Prozess Probleme hat
  - ein Standby-Server mit Replikation Slot heruntergefahren ist
- händisch WAL-Files eines abgestürzten Servers entfernen ist ein sicherer Weg, PostgreSQL zu zerstören (damit verunmöglicht man die Wiederherstellung)

Datenkorruption durch Herumfummeln im Katalog

Beispiele für Katalogmanipulationen

- Katalogtabellen ändern ist immer ein guter Weg, die Datenbank zu zerstören
- Beispiel: Spalte ohne `ACCESS EXCLUSIVE`-Sperre löschen

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

- Beispiel: Datentyp ändern, ohne die Tabelle neu zu schreiben

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

Datenkorruption behandeln

Grundlegende Hinweise

- mit der beschädigten Datenbank nicht weiterarbeiten (Korruption kann um sich greifen, neue Daten können verloren gehen)
- wenn man ein gutes Backup hat, sollte man das einspielen
- PostgreSQL herunterfahren und das Filesystem sichern (Datenkorruption behandeln zerstört oft Daten)
- nach dem Beheben des Problems, unbedingt einen Dump ziehen und in einen neuen Cluster importieren (sonst kann unsichtbare Korruption zurückbleiben)
- Ursache erforschen und das zugrundeliegende Problem beheben

Datensicherung gegen Datenkorruption

Schütz dich mit Datensicherungen

- das tut ohnehin jeder, oder?
- Datensicherung ohne Überwachung ist keine Datensicherung (z.B. ein `pg_dump`, das wegen Datenkorruption wiederholt fehlschlägt)
- `pg_dump` ist besser als eine Filesystemsicherung (wenn der Dump eingespielt werden kann, ist die Datenkorruption automatisch behoben)
- mit Filesystemsicherung kann Point-In-Time-Recovery den Datenverlust minimieren (aber man kann Datenkorruption meist nicht erkennen)
- am besten ist eine Kombination: regelmäßige Filesystemsicherung und gelegentlich ein `pg_dump`

Behebung durch Einspielen der Sicherung

- einspielen der letzten guten Sicherung heißt die späteren Daten verlorengeben, aber ist wenig Aufwand
- kein Expertenwissen erforderlich
- wenn man keine andere Lösung findet ist das der einzige Ausweg (oder Rückweg?)
- beim Erwägen anderer Möglichkeiten, so vorgehen:
  - vorher ein Limit an Zeit und Aufwand für Wiederherstellungsversuche festlegen
  - Expertenschätzung einholen (wird nie exakt sein)
  - aufhören, wenn das Limit erreicht ist (sonst wird's noch teurer)

Indexkorruption

Indexkorruption: Ursachen und Symptome

- Index inkonsistent, oder Daten in Tabelle und Index passen nicht zusammen
- oft durch Dinge verursacht, die unveränderlich sein sollten, es aber nicht waren (Funktionen, Sortierreihenfolgen nach dem Upgrade, . . .)
- unterschiedliche Ergebnisse bei Table Scan und Index Scan (experimentieren mit `enable_indexscan` und anderen)
- Indexe mit der Extension `amcheck` untersuchen:
  - `bt_index_check` prüft interne Konsistenz (mit `heapallindexed => TRUE` wird getestet, ob alle Zeilen indexiert sind)
  - `bt_index_parent_check` testet gründlicher, benötigt aber eine `SHARE`-Sperre (keine gleichzeitigen Datenänderungen)

Indexkorruption behandeln

- das ist normalerweise einfach, weil Indexdaten redundant sind:

  ```
  REINDEX INDEX CONCURRENTLY kaputter_index;
  ```

- wenn das fehlschlägt, ist auch die Tabelle korrupt (vielleicht durch die Indexkorruption verursacht)
- wenn ein Index auf einer Katalogtabelle betroffen ist:
  - Server mit `-P` starten (Systemindexe ignorieren)
  - als Superuser Index neu aufbauen
  - Server ohne `-P` neu starten

Datenkorruption, die keine Fehler verursacht

Datenkorruption ohne Fehler: Symptome

- falsche Ergebnisse
- fehlende Daten, verursacht durch
  - Zeilen, die unsichtbar werden
  - Blöcke, die ganz aus Nullen bestehen (gelten als leer)
  - abgeschnittene oder geleerte Dateien
- verletzte Fremdschlüssel (vielleicht hat jemand in der Vergangenheit die Bedingung deaktiviert)
- doppelte Daten, die Eindeutigkeitsbedingungen verletzen (oft durch Indexkorruption verursacht)

Datenkorruption ohne Fehler behandeln

- das ist eine leichte Übung
- Datenbank mit `pg_dump` extrakieren und in neuen Cluster einspielen
- wenn es beim Einspielen Fehler gibt (Constraint kann nicht erzeugt werden), händisch Daten löschen oder hinzufügen
- wir werden versuchen, alle schwierigeren Fälle auf diesen Fall zurückzuführen, indem wir Fehler zum Schweigen bringen
  - niemals vergessen, dass man mit einer Datenbank, die einmal korrupt war, nicht weiterarbeiten soll

Datenkorruption verursacht Fehler ohne Absturz

Mögliche Gründe für Fehler

- Checksumme im Datenblock stimmt nicht
  - nur wenn Cluster mit Checksummen erstellt wurde (kostet zusätzlich, zeigt aber **von der Platte erzeugte** Datenkurruption frühzeitig an)
- Block-Header ist korrupt, zum Beispiel:

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

- Tabellenzeile ist korrupt, zum Beispiel:

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

- TOAST-Daten sind korrupt, zum Beispiel:

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

Umgang mit korrupten Blöcken

- `ignore_checksum_failure = on` ignoriert falsche Checksummen (aber Mist im Datenblock kann trotzdem zu Fehlern führen)
- mit `zero_damaged_pages = on` behandelt PostgreSQL Blöcke mit korruptem Header als leere Blöcke
  - hilft nicht, wenn der Header in Ordnung ist, aber der Inhalt kaputt
- diese Parameter ändern die Blöcke auf der Platte nicht, können aber Fehler vermeiden, damit `pg_dump` laufen kann
- wir sehen später, wie man Daten aus korrupten Blöcken extrahieren kann

TOAST-Korruption erkennen

- überlange Werte werden „extern” in der TOAST-Tabelle gespeichert
- die eigentliche Tabellenzeile enthält nur einen Verweis („TOAST-Pointer”)
- das typische Symptom ist ein Fehler, der das Wort „`toast`” enthält
- `SELECT` funktioniert, solange man nicht die Spalte anzeigt, die auf den kaputten TOAST-Eintrag verweist

TOAST-Korruption beheben

- an sich einfach: die betoffene Zeile mit `DELETE` löschen oder den kaputten Wert mit `UPDATE` überschreiben
- finde die betroffenen Zeilen mit etwas wie

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

Fehlerhafte Tabellenzeilen behandeln

- ähnlich wie bei TOAST, aber wir können den Primärschlüssel nicht selektieren, also müssen wir alle möglichen Werte durchprobieren

```
DO $$
DECLARE i bigint; max_id bigint;
BEGIN
   /* wenn das nicht geht, muss man eine Obergrenze raten */
   SELECT max(id) INTO max_id FROM badtable;
   FOR i IN 1..max_id LOOP
      BEGIN
         /* zwingt PostgreSQL, alle Spalten zu lesen */
         PERFORM badtable::text INTO r FROM badtable WHERE id = i;
      EXCEPTION WHEN OTHERS THEN
         RAISE NOTICE 'id = %', i;
      END;
   END LOOP;
END;$$;
```

- wenn `DELETE` fehlschlägt, mit `SELECT` alle Zeilen außer den schadhaften extrahieren.

`pg_surgery` verwenden

- neue Extension in PostgreSQL v14
- nützlich für den Umgang mit Fehlern, die Zugriff auf die Zeile verhindern, wie

  ```
  found xmin 16804535 from before relfrozenxid 90126924
  ```

- Funktion `heap_force_kill` löscht Zeilen mit bestimmten `ctid`s
- Function `heap_force_freeze` macht Zeilen mit bestimmten `ctid`s sichtbar
- ein weiteres gutes Werkzeug, um die Datenbank zu zerstören – nur für Experten

Datenkorruption, die zum Absturz führt

Wie verursacht Datenkorruption Abstürze?

- z.B. durch Lesen einer falschen Länge, was zu riesiger Speicherallokation oder zur Schutzverletzung führen kann
- solche Abstürze sind normalerweise ein Programmfehler (fehlende Überprüfung)
- aber alles immer testen macht die Verarbeitung zu langsam
- wenn man PostgreSQL mit `--enable-cassert` kompiliert, wird viel mehr überprüft
- mit Asserts gibt es immer noch einen Absturz, aber die Fehlermeldung sagt mehr aus
- andererseits kann Datenkorruption Asserts auslösen, die bei normaler Kompilation nicht auftreten $\Rightarrow$ beides probieren

Umgang mit Datenkorruption mit Abstürzen

- wieder ist die Idee, die kaputten Zeilen herauszufinden und den Rest wegzukopieren
- bei Abstürzen genügt PL/pgSQL nicht, und wir müssen Client-Code schreiben
- das Programm muss sich nach einem Absturz neu verbinden und weitermachen
- mehr Entwicklungsaufwand, aber sonst wie gehabt

Daten aus kaputten Blöcken oder Zeilen retten

- nur versuchen, wenn es die Mühe wert ist
- Versuch mit der Extension `pageinspect`
  - `get_raw_page` liest einen Block
  - `heap_page_item_attrs` kann Daten extrahieren, versagt aber normalerweise bei Korruption
- `pg_filedump`
  `https://git.postgresql.org/gitweb/?p=pg_filedump.git`
  - auch das wird meist durch Datenkorruption verwirrt
- die letzte Zuflucht ist „`od -t x1`" oder ein Hex-Editor und Wissen über die Datenstrukturen von PostgreSQL

Fehlende oder leere Dateien

Ursachen für fehlende oder leere Dateien

- Benutzerfehler (gelöscht vom Administrator)
  - häufig bei WAL-Segmenten (siehe oben)
- lügende Platte (PostgreSQL, hat die Daten geschrieben, aber sie fehlen nach dem Absturz)
- Dateisystem-Check nach einem Hardwareproblem
- falsche eingestellter Virenschutz
  - niemals auf das PostgreSQL-Datenverzeichnis loslassen

Umgang mit fehlenden Dateien

- Tabellen, Indexe etc. sind einfach: `DROP` auf das Objekt
- fehlende WAL-Segmente $\Rightarrow$ `pg_resetwal`
  - die Dokumentation sagt, wie man gute Parameter findet
- wenn andere Dateien fehlen, kann man sie manchmal „ersetzen”
- Beispiel: Erstellen eines fehlenden Commit-Logs aus 12 Blöcken mit allen Transaktionen kommittiert (Daten also sichtbar)

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

Zusammenfassung

Das wichtigste in Kürze

- brav bleiben, nicht mit dem Feuer spielen
- gute Datensicherung, idealerweise logisch *und* physisch
- Korruption nur reparieren, wenn es anders nicht geht (Datensicherung einspielen ist besser)
- Dateisystem wegsichern, bevor man beginnt, Korruption zu behandeln
- Bordmittel verwenden: `pg_resetwal`, `pg_surgery`, `pageinspect`
- mit der reparierten Datenbank nicht weiterarbeiten: Export und Import

Fragen