PostgreSQL Replikation: 15 Tücken und Lösungen

Julian Markwort

pgconf.de 2024

Vorstellung

Julian Markwort
Senior Database Consultant

- PostgreSQL Consulting
- PostgreSQL Support
- PostgreSQL Remote DBA
- and more . . .

Motivation

- wir beraten, installieren und administrieren sehr viele PostgreSQL Cluster
  - mit binärer Replikation
  - fast immer mit Patroni
  - manchmal mit logischer Replikation
- wir sind dabei regelmäßig mit Replikationsproblemen konfrontiert

PostgreSQL Replikationsprobleme

PostgreSQL Replikation ist nicht schlecht

- meistens sind es menschliche Fehler, oder fehlerhafte Automatisierung
- oft sind es falsche Annahmen und Missverständnisse
- manchmal ist die Dokumentation nicht eindeutig oder fehlt
- selten ist es ein Bug in PostgreSQL

Einleitung

Wir haben 15 häufige Tücken ausgewählt

- wir werden diese Tücken einzeln besprechen
- wir werden Konzepte, die für das Verständnis erforderlich sind, vorstellen
- wir werden Lösungen für alle Tücken aufzeigen

Dieser Vortrag soll primär das Bewusstsein für diese Probleme schärfen, aber sie nicht zerlegen bis alle gelangweilt sind

Folien

Dieser Foliensatz ist schon hochgeladen!

Tücken mit WAL

1 - WAL Recycling - Umriss

- PostgreSQL schreibt *Write Ahead Log* um Crash Recovery zu garantieren
- Crash Recovery beginnt immer beim letzten `CHECKPOINT`
- also darf PostgreSQL *älteres* WAL *recyclen*

1 - WAL Recycling - Tücke

- binäre Replikation ist fortlaufende Recovery
- (fortlaufende) Recovery funktioniert nur, wenn es keine *Lücken* im WAL gibt
- Wenn die Primary WAL recyclet können die Replikas es nicht mehr zum Aufholen nutzen

1 - WAL Recycling - Lösung

- setze `wal_keep_size`
- verwende *Replication Slots*
- verwende archiving

2 - Archiving - Umriss

- WAL ist in 16MB *Segment* Dateien aufgeteilt
- sobald PostgreSQL ein Segment abgeschlossen hat
  - wechselt es zu einem neuen Segment
  - ruft es das `archive_command` für das alte auf (falls `archive_mode` aktiviert ist)
  - wenn dies Erfolg meldet, darf PostgreSQL das alte Segment recyclen
- `archive_command` kopiert jeweils ein Segment in ein zentrales *Archiv*
- `restore_command` kann von Replikas verwendet werden, um Segmente aus dem Archiv abzurufen, die von der Primary schon recyclet wurden

2 - Archiving - Tücke

- `archive_command` kann fehlschlagen
- `archive_command` kann zu langsam sein
- dann kann PostgreSQL diese Datei nicht recyclen, und alle nachfolgenden auch nicht
- das führt schnell zu *Platte voll*

2 - Archiving - Lösung

- Archivieren überwachen (`SELECT * FROM pg_stat_archiver;`)
- mache `pg_wal` groß genug
- übe, `pg_wal` auf die Schnelle zu vergrößern

3 - Replication Slots - Umriss

- replication slots können manuell angelegt werden, oder automatisch durch patroni, pg_basebackup etc.
- replication slots verfolgen den Fortschritt der Replikation
  - die Replika *schiebt* eine `restart_lsn` im Slot *voran*
  - diese LSN (*Logical Sequence Number*) beschreibt den Punkt, ab dem die Replika (nochmal) WAL abrufen könnte, z.B. nach Crash oder Netzwerkunterbrechung

3 - Replication Slots - Tücke

- die Primary muss alles WAL ab dieser `restart_lsn` aufheben
  - selbst wenn keine Replika, die das voranschieben könnte, verbunden ist
- das führt schnell zu *Platte voll*

3 - Replication Slots - Lösung

- replications slots überwachen (`SELECT * FROM pg_replication_slots;`)
- setze `max_slot_wal_keep_size`
- mache `pg_wal` groß genug

4 - Replikation von Replication Slots - Umriss

- replication slots werden nicht repliziert

4 - Replikation von Replication Slots - Tücke

- Wenn die Primary abstürtzt oder abschaltet und dann eine Replika befördert wird, so weiß diese neue Replika nichts von den Slots, die auf der alten Primary waren

4 - Replikation von Replication Slots - Lösung

- verwende das `permanent replication slot` Feature in **Patroni**
- verwende **pg_failover_slots** extension
- wird vielleicht in PG 17 gelöst

5 - Tanz der Parameter - Umriss

- es gibt einige PostgreSQL Parameter, die zur Allokation des shared memory gebraucht werden
  - z.B. zum verfolgen des Status von Transaktionen

```
max_connections
max_locks_per_transactions
max_worker_processes
max_wal_senders
max_prepared_transactions
wal_level
track_commit_timestamp
```


5 - Tanz der Parameter - Tücke

- Solche Parameter müssen auf der Replika gleich (oder größer) sein, sonst kann diese die Transaktionen nicht rekonstruieren
- eine Replika kann unter solchen Umständen nicht gestartet werden
  - Start endet mit einer Fehlermeldung

5 - Tanz der Parameter - Lösung

- Wenn man solche Werte erhöht, muss man damit bei den Replikas beginnen
- Wenn man solche Werte verringert, muss man bei der Primary anfangen

Tücken beim Switchover

6 - Split Brain

- es sollte immer nur eine Primary im Cluster existieren
- Transaktionen, die auf verschiedenen Primaries akzeptiert werden, können nicht vereinbart werden.
- stelle sicher, dass die alte Primary wirklich abgeschaltet ist, bevor eine Replika befördert wird
- analysiere, wie die eingesetzte HA Lösung das vermeidet (es sollte eine Art Lock verwenden)

7 - Timeline-Wechsel - Umriss

- Knoten A ist Primary, Knoten B ist Replika, beide sind auf Timeline (*TL*) 1

```
Knoten A  |  TL 1  |  1  2  3  4  5  6  7  *Knoten A stürzt ab*
Knoten B  |  TL 1  |  1  2  3  4  *Verbindung zu Knoten A verloren*
```

- **befördere** Knoten B

```
Knoten B  |  TL 2  |              5  6  7  8  9
```

- Knoten A startet erneut

```
Knoten A  |  TL 1  |           ..  5  6  7
```

- man muss alle konfliktierenden Daten und Transaktionen auf Knoten A verwerfen (TL 1, records 5-7)

7 - Timeline-Wechsel - Tücke

- PostgreSQL schreibt nur REDO Transaktionslog
- wir können nur *vorwärst* recovern
- man kann - mit WAL alleine - keine Änderungen ungeschehen machen

7 - Timeline-Wechsel - Lösung

- hole eine neue Kopie des *Data Verzeichnisses* von der Primary
  - einfach, idiotensicher, aber teuer (IO, Bandbreite)
- verwende pg_rewind
  - sucht den *point of divergence*
  - synchronisiert die Replika mit der Primary indem das WAL zwischen den beiden verglichen wird
  - synchronisiert nur betroffene Teile der Datendateien
  - nach Abschluss kann die Replika am *point of divergence* die Recovery wieder aufnehmen und den Timeline-Wechsel nachvollziehen

8 - Switchover-Implikationen für Autovacuum - Umriss

- manches wird aus Performanzgründen nicht repliziert
- das betrifft vor allem den *statistics collector* (`pg_stat_user_tables` etc.)

8 - Switchover-Implikationen für Autovacuum - Tücke

- das sind z.B. Verwendungszähler, diese dauerhaft zu speichern und zu replizieren wäre langsam und ist für die Datenkonsistenz nicht nötig
- `pg_stat_user_tables` und ähnliche Views werden von **autovacuum** genutzt um zu entscheiden, wo es aufräumen muss
- das Problem besteht auch auf alleinstehenden Instanzen nach einer Crash Recovery

8 - Switchover-Implikationen für Autovacuum - Lösung

- lasse `ANALYZE` nach einem Switchover laufen
  - zumindest auf den Tabellen, die der Autovakuum nicht von selbst schnell genug aufgreift
  - also primär auf sehr großen Tabellen, die ein vergleichsweise niedriges Volumen an Inserts/Updates/Deletes erfahren
- überwache autovacuum und table bloat

9 - Transaktionsverlust nach Failover - Tücke

- standardmäßig wartet PotgreSQL nicht auf *replication feedback* der Replikas
- es passiert leicht, dass man eine Replika befördert, die nicht alle Transaktionen erhalten hat

9 - Transaktionsverlust nach Failover - Lösung

- stelle sicher, dass nur aktuelle Replikas befördert werden
  - `pg_ctl stop` (smart mode oder fast mode) wartet bis die Replikas alles bekommen haben
  - achte auf Timeouts
- verwende ggf. *synchrone Replikation*

10 - Synchrone Replikation - Umriss

du schaltest logische Replikation an

- `COMMIT` Latenzen steigen
  - nimm's hin!
- ewiges Warten beim `COMMIT` wenn keine Replika da ist
  - füge eine weitere Replika hinzu!
- du machst einen failover und es wird trotzdem ein pg_rewing oder neues Backup benötigt
  - hast du jetzt Transaktionen verloren?

10 - Synchrone Replikation - Tücke

Das *synchronous commit* Feature wartet schlicht, dass die Replikas den `COMMIT` Transaktionslogeintrag bestätigen

- bei allen anderen Einträge wird nicht gewartet
  - man kann also immer noch Änderungen verlieren, die man aber auch verlöre, wenn eine Instanz abstürtzt, bevor man den `COMMIT` anstößt
  - betrifft z.B. auch Änderungen durch Autovacuum

Tücken bei Read-Only-Replikas

11 - Konsistenz bei Abfragen auf Replikas - Tücke

- Konsistenz über Instanzen hinweg ist manchmal komisch
- im asynchronen Modus:
  - auf einer Replika kann man ggf. Änderungen nicht sehen, die auf der Primary schon committed sind
- im synchronen Modus:
  - kann man ggf. Änderungen auf einer Replika sehen, während die Primary noch auf Feedback von anderen Replikas wartet

11 - Konsistenz bei Abfragen auf Replikas - Lösung

- überwache *replication lag* (`SELECT pg_wal_lsn_diff(pg_current_wal_insert_lsn(), replay_lsn) from pg_stat_activity`)
- behandle Replikas als wären sie nicht einsatzbereit wenn sie Lag haben
- Applikationen sollten für mehrere Abfragen nacheinander die gleiche Instanz nutzen

12 - Vacuum und Replikationskonflikte - Umriss

- replay kann durch offene Transaktionen auf Replikas blockiert werden
- es gibt keine schreibenden Queries auf Replikas, wie kann es dann Konflikte geben?
  - jede Transaktion hat einen Snapshot, der dafür sorgt, dass sie die gleichen Versionen von Zeilen sieht, selbst wenn es nebenläufige Updates gibt
  - Transaktionen auf Replikas benutzen natürlich auch Snapshots

12 - Vacuum und Replikationskonflikte - Tücke

- die Primary lässt durch Autovacuum regelmäßig alte Daten aufräumen
- Autovacuum möchte Zeilenversionen entfernen, die auf der Primary von keiner Transaktion mehr gesehen werden können
- Autovacuum schreibt diese Änderungen natürlich ins WAL
- replay dieser Änderungen auf der Replika konfliktiert mit Transaktionen, die noch einen älteren Snapshot haben und diese alten Versionen noch sehen wollen *könnten*

12 - Vacuum und Replikationskonflikte - Lösung

- es gibt einen Spagat zwischen
  - Abarbeiten von Transaktionen auf der Replika
  - mti dem WAL replay fortfahren
- dieser Spagat kann mit `max_standby_streaming_delay` beeinflusst werden
  - so lange darf das replay - ab dem Zeitpunkt des Empfangs des WAL-Eintrags - warten
  - Standard sind 30 Sekunden
  - bezieht sich *nicht* auf die Dauer der konfliktierenden Transaktion

12 - Vacuum und Replikationskonflikte - Lösung

- Replikas, die das replay lange verzögern dürfen, sind keine guten Kandidaten für Switchover
  - sie müssten, wenn sie befördert werden, noch alle Transaktionen nachholen

12 - Vacuum und Replikationskonflikte - Lösung

- die Replika kann die Primary informieren, welche Snapshots sie noch braucht: `hot_standby_feedback`
  - das bremst aber ggf. den Fortschritt des Autovacuum auf der Primary

13 - Prepared Transactions und Recovery

- prepared transactions (*2 phase commit*) werden ins WAL geschrieben
  - überleben crash recovery und somit auch Switchover
  - stelle sicher, dass der Transaktionsmanager der Applikation damit umgehen kann
  - überwache “verwaiste” prepared transactions (`SELECT gid FROM pg_prepared_xacts WHERE prepared<now()-'2 hours'::interval`)
- es *gab* einen Bug bei der Recovery von prepared transactions in PG 13 und 14

14 - Hot Standby funktioniert nicht - Umriss

Eine Replika im *hot standby* Modus muss wissen, welche Transaktionen auf der Primary aktuell *in flight* sind, um zu wissen, welche Daten sie den Queries geben darf

- die Primary schreibt regelmäßig einen `XLOG_RUNNING_XACTS` Eintrag ins WAL
- Replikas können erst dann Queries bedienen, wenn sie seit ihrer *Minimum recovery ending location* (`pg_controldata`) einen solchen Eintrag gesehen haben

14 - Hot Standby funktioniert nicht - Tücke

- alle Instanzen im Cluster stürzen ab
- du startest alle als Replikas neu, wartest bis sie Verbindungen erlauben
  - sie könnten keinen `XLOG_RUNNING_XACTS` Eintrag in ihrem WAL nach der *Minimum recovery ending location* haben
  - dann können sie keine lesenden Verbindungen erlauben und unbegrenzt auf so einen Eintrag warten

14 - Hot Standby funktioniert nicht - Lösung

Du musst eine Replika auswählen (am besten die mit den meisten Transaktionen, z.B. per Analyse mit pg_waldump) und manuell befördern

15 - Hot Standby funktioniert nicht - Bonus Tücke

ein `XLOG_RUNNING_XACTS` Eintrag hat nur begrenzt viel Platz für Subtransaktionen

- wenn auf der Primary zu viele Subtransaktionen laufen kann dieser Eintrag nicht alle erfassen
- dann wird im `XLOG_RUNNING_XACTS` Eintrag das `suboverflowed` Flag gesetzt
- wir können nicht den hot standby Modus wechseln wenn wir nicht alle laufenden (Sub-)Transaktionen kennen

15 - Hot Standby funktioniert nicht - Bonus Lösung

- benutze gar keine Subtransaktionen (es gibt bekannte Performanceprobleme)
- benutze nicht zu viele Subtransaktionen
  - benutze `SAVEPOINT` nicht wie Freibier
  - benutze `PL/pgSQL` exception blocks nicht wie Freibier
- erlaube keine lang laufenden Transaktionen

Zusammenfassung

Rezept für guten Schlaf

- Speicherpplatz überwachen!
- replication slots nutzen und überwachen!
- archiving nutzen und überwachen!

Rezept für gute Administration

- alles doppelt prüfen
- sicherstellen, dass kein split brain auftreten kann
- sicherstellen dass keine Transaktionen verloren gehen

mehr Tücken und Lösungen

PostgreSQL Replication: 20 Pitfalls and Solutions

- Vortrag auf der PGConf.EU 2023 in Prag
- Folien hochgeladen
- Video auf Youtube

Danke!