Migration von Firebird zu PostgreSQL. Was kann schiefgehen?

(c) Alexandr Shaposhnikov, bearbeitet und angepasst von Alexey Kovyazin. Korrektur der deutschen Übersetzung durch Martin Köditz.

Table of Contents

Vorwort . . . 2
1. Unterschiede in der MVCC-Implementierung . . . 3
  1.1. Test 1 . . . 5
  1.2. Test 2 . . . 6
2. SUSPEND und RETURN NEXT . . . 7
3. Verwendung autonomer Transaktionen . . . 8
4. Eingeschränkte Funktionalität anonymer PSQL-Blöcke in PostgreSQL . . . 10
5. Prozesse und Threads . . . 10
6. Transaktionszähler . . . 11
7. Arbeiten mit Speichersystemen . . . 12
8. Backup-Unterschiede . . . 13
  8.1. Gbak & pg_dump . . . 13
  8.2. Nbackup & pg_basebackup . . . 14
9. Code-Migrationsprobleme mit temporären Tabellen . . . 15
10. Verhalten des Abfrageoptimierers . . . 16
11. Datengrößen . . . 16
12. Konsequenzen hängender Transaktionen. VACUUM FULL und GFIX -sweep . . . 17
13. Eine Transaktion pro Verbindung . . . 17
14. Keine Paket-Implementierung . . . 18
15. Tabellen ändern . . . 18
16. Überprüfung von Abhängigkeiten beim Kompilieren von Prozeduren . . . 18
17. SQL-Code-Migration . . . 19
18. Typinkompatibilität . . . 19
19. Unterschiedliche Trigger-Implementierungen . . . 20
20. Datenbank-Tools . . . 20
21. Komplexere Administration . . . 21
22. Gewohnheit ist zweite Natur . . . 21

23. Fazit 21
23.1. Wichtige Fragen, die Sie sich vor der Migration stellen sollten 21
24. Kontakt 22

Vorwort

Es ist kein Geheimnis, dass in den letzten Jahren verschiedene Unternehmen sehr häufig beschlossen haben, ein funktionierendes Informationssystem von Firebird zu PostgreSQL zu migrieren.

Eine typische Situation sieht folgendermaßen aus:

Das Projekt läuft seit mehreren Jahren. Der Kunde glaubt, dass die Probleme des Projekts (jede Software hat einige Probleme) durch das DBMS verursacht werden. **Firebird ist ein "schlechtes" DBMS.**

Anstatt

- externe Unternehmen als Berater zu engagieren
- die eigenen Mitarbeiter zu schulen und zu zertifizieren
- deren Qualifikationsniveau zu verbessern

ist es viel einfacher, sich selbst davon zu überzeugen, dass die Grundursache der Probleme Firebird ist, und zu beschließen, zu einem anderen DBMS zu migrieren.

Dieses Problem ist nicht auf ein bestimmtes DBMS bezogen und ist rein managementbedingt.

Glauben Sie mir nicht? Die hervorgehobene Zeile ist ein wörtliches Zitat aus einem der Vorträge über PostgreSQL DBMS, mit einer Änderung - im Original stand PostgreSQL anstelle von Firebird.

Die Situation wird dadurch verschlimmert, dass Entscheidungsträger oft die Probleme und Schwierigkeiten dieses Prozesses nicht verstehen.

Diese Entscheidung wird ernsthafte Probleme für das IT-System des Unternehmens verursachen. Normalerweise folgen Menschen der einfachen Regel "wenn etwas gut funktioniert, lass es in Ruhe." Diese Regel hätte solche großen Änderungen verhindert. Stattdessen glaubten die Menschen an die falsche Idee einer "Wunderwaffe" - eine perfekte Lösung, die alles auf magische Weise beheben kann.

Wer sollte diesen Artikel lesen

Dieses Dokument richtet sich an technische Manager und CTOs, die für Datenbank-Migrationsprojekte verantwortlich sind. Migrationsprojekte bergen erhebliche Risiken für technische Führungskräfte, da sie oft die Schuld bekommen, wenn die Kosten deutlich höher ausfallen als geplant (manchmal zwei- bis dreimal so teuer), wenn Projekte viel länger dauern als erwartet und wenn die Endergebnisse nicht so gut funktionieren wie versprochen. Technische Führungskräfte sollten wissen, dass Migrationsprojekte häufig scheitern und Projektleiter meist zur Verantwortung gezogen werden, wenn Budgets zu groß werden, Zeitpläne nicht eingehalten

werden und das neue System nicht so leistungsfähig ist wie das alte.

Beachten Sie, dass die Schwierigkeit der Migration davon abhängt, wie komplex Ihre Datenbank ist und wie stark Sie spezielle Firebird-Funktionen nutzen. Der Zeit- und Kostenaufwand für die Migration hängt von mehreren Faktoren ab: wie viel Code Ihr Projekt enthält und wie gut dieser Code ist, wie groß Ihre Datenbank ist, ob Sie das System beschleunigen müssen und welche weiteren Probleme während der Arbeit auftreten.

Dieser Artikel macht PostgreSQL keine Vorwürfe! Es geht um Migration und ihre Konsequenzen

PostgreSQL ist ein großartiges Datenbanksystem mit vielen nützlichen Funktionen, aber es ist anders, und das Verhalten Ihres Systems wird sich direkt nach der Migration ändern – und zunächst wird es sich verschlechtern.

Wenn Sie von Firebird zu PostgreSQL wechseln, profitieren Sie während der Migration nicht von den Funktionen, die PostgreSQL bietet und Firebird nicht hat, aber Sie werden definitiv Probleme durch fehlende Firebird-Funktionen oder Funktionen haben, die in PostgreSQL anders arbeiten.

Abfragen, die mit denselben Daten in Firebird gut funktionieren, werden in PostgreSQL nicht gut funktionieren. Das Gegenteil ist ebenfalls wahr, aber beim Wechsel von Firebird zu PostgreSQL werden Sie zuerst die Probleme aus dem ersten Teil dieser Aussage erleben.

Schauen wir uns kurz an, mit welchen Problemen Sie bei dieser Migration konfrontiert werden können.

1. Unterschiede in der MVCC-Implementierung

Wahrscheinlich das unangenehmste und unerwartetste für diejenigen, die Firebird gewohnt sind – genauer gesagt, die Art und Weise, wie Firebird die Versionierung von Daten implementiert.

Eigentlich sollte man über die spezifische Implementierung von MVCC in PostgreSQL sprechen: Hier unterscheidet sich der Mechanismus zur Versionskontrolle grundlegend von Firebird, MS SQL und ORACLE.

Über die Unterschiede können Sie in diesem Artikel lesen.

Kurz gesagt, der klassische Ansatz zur Implementierung von MVCC ist ein “optimistischer” Algorithmus, der davon ausgeht, dass Transaktionen erfolgreich abgeschlossen werden und alte Versionen nicht benötigt werden. Daher wird die neue Version des Datensatzes über die alte geschrieben, und entweder das Delta oder der gesamte alte Datensatz wird im Undo-Log gespeichert, sodass die alte Version bei Bedarf wiederhergestellt werden kann – was selten vorkommen sollte. Die konventionelle klassische MVCC-Implementierung speichert also Unterschiede und stellt die alte Version der Daten durch Subtraktion der Diffs von der aktuellen/letzten Version wieder her. So funktioniert es in Firebird.

Die MVCC-Implementierung in PostgreSQL ist nicht schlecht – sie ist einfach grundlegend anders.

Die Idee ist hier „Copy-on-Write“, wobei sehr schnell und in großer Zahl Versionen von Datensätzen (vollständige Kopien mit Systeminformationen) erstellt werden.

Die Implementierung von MVCC in PostgreSQL sieht überhaupt keine Aktualisierung eines Datensatzes vor; wenn eine Aktualisierung notwendig ist, werden Löschung und Einfügung durchgeführt.

Jede Version eines Datensatzes in PostgreSQL (Tupel) ist eine vollständige Kopie des Datensatzes mit einigen Servicefeldern.

Zum Beispiel: das XMIN-Feld – es enthält die ID der Transaktion, die den Datensatz erstellt hat, was es ermöglicht zu verstehen, welche Transaktionen diesen Datensatz sehen sollen. Oder das XMAX-Feld – es enthält die ID der Transaktion, die den Datensatz gelöscht hat.

Was beim Aktualisieren eines Datensatzes passiert:

Zuerst füllen wir das XMAX-Feld in der aktuellen Version des Datensatzes aus – dort schreiben wir die ID der aktualisierenden Transaktion, dann erstellen wir eine neue Zeile, in der wir die ID der aktualisierenden Transaktion in XMIN schreiben. Und ein Link zur neuen Zeile wird zur alten Version des Datensatzes hinzugefügt.

Das heißt, in der Implementierung – buchstäblich – DELETE UND INSERT.

Wenn Sie eine Spalte einer Zeile aktualisieren, wird die gesamte Zeile in die neue Version kopiert, wahrscheinlich auf eine neue Seite. (PostgreSQL versucht, die neue Version auf derselben Seite wie die alte zu platzieren, aber das ist längst nicht immer möglich), und die alte Zeile wird ebenfalls mit einem Zeiger auf die neue Version geändert. Indexeinträge folgen demselben Muster: Da es eine komplett neue Kopie gibt, müssen alle Indizes aktualisiert werden, um auf den neuen Seitenort zu zeigen. **Alle Indizes** – auch diejenigen, die nicht mit der geänderten Spalte zu tun haben, werden aktualisiert, weil die gesamte Zeile verschoben wird.

Später ist eine Bereinigung der alten Tupel (VACUUM) erforderlich. Eine weitere Konsequenz dieses Ansatzes ist ein großes Volumen an WAL-Generierung (Redo-Log), da viele Blöcke betroffen sind, wenn ein Tupel an einen anderen Ort verschoben wird.

Die Folge ist eine höhere Festplattenbelastung bei Updates, höhere Intensität bei der Replikation.

Vielleicht liegt der Grund für den Unterschied in der MVCC-Implementierung darin, dass PostgreSQL als akademisches Projekt begann und sich entwickelte, sodass die MVCC-Implementierung wahrlich akademisch ist. In anderen Systemen war die Implementierung zum Zeitpunkt der Entstehung auf Effizienz ausgerichtet, zumal die Leistung der Computer damals deutlich geringer war als heute.

Hier muss man umdenken. Nach der Arbeit mit Firebird erwartet man, dass das Aktualisieren eines Datensatzes viel weniger kostet als „delete and insert“, aber denken Sie an das alte Sprichwort: „Die Realität zerschlägt Erwartungen“, und PostgreSQL folgt seinen eigenen Regeln, nicht Ihren.

Um das oben Gesagte zu veranschaulichen, werden wir mit Tests eine Tabelle mit folgender Struktur erstellen:

DAT-Testtabelle (Syntax zur besseren Lesbarkeit gekürzt)

```
-- Felder
CREATE TABLE DAT (
    ID BIGINT NOT NULL,
    I1..I8 BIGINT,
    N1..N8 DOUBLE PRECISION,
    D1..D8 TIMESTAMP,
    S1..S8 VARCHAR(100),
    T1..T8 VARCHAR(1000)
);

PK$DAT PRIMARY KEY (ID);

-- Indizes
X_I1 ON DAT (I1); X_I23 ON DAT (I2, I3); X_I456 ON DAT (I4, I5, I6);
X_N1 ON DAT (N1); X_N23 ON DAT (N2, N3); X_N456 ON DAT (N4, N5, N6);
X_D1 ON DAT (D1); X_D23 ON DAT (D2, D3); X_D456 ON DAT (D4, D5, D6);
X_S1 ON DAT (S1); X_S23 ON DAT (S2, S3); X_S456 ON DAT (S4, S5, S6);
X_F1 ON DAT (F1); X_F23 ON DAT (F2, F3); X_F456 ON DAT (F4, F5, F6);
X_T1 ON DAT (T1);
```

Die Tabelle enthält 10 Millionen Datensätze, das erste Feld der Gruppe ist immer gefüllt, die verbleibenden Felder der Gruppe (Nr. 2-Nr. 8) sind mit 50%iger Wahrscheinlichkeit mit `NULL` gefüllt.

Die Datentabelle ist in Firebird 5.0.2 und PostgreSQL 17.4 Datenbanken identisch.

Die Datenbankserver laufen auf Linux Mint 22.1 (SSD, 32Gb RAM).

Eine grundlegende Konfiguration der Datenbankserver wurde durchgeführt.

1.1. Test 1

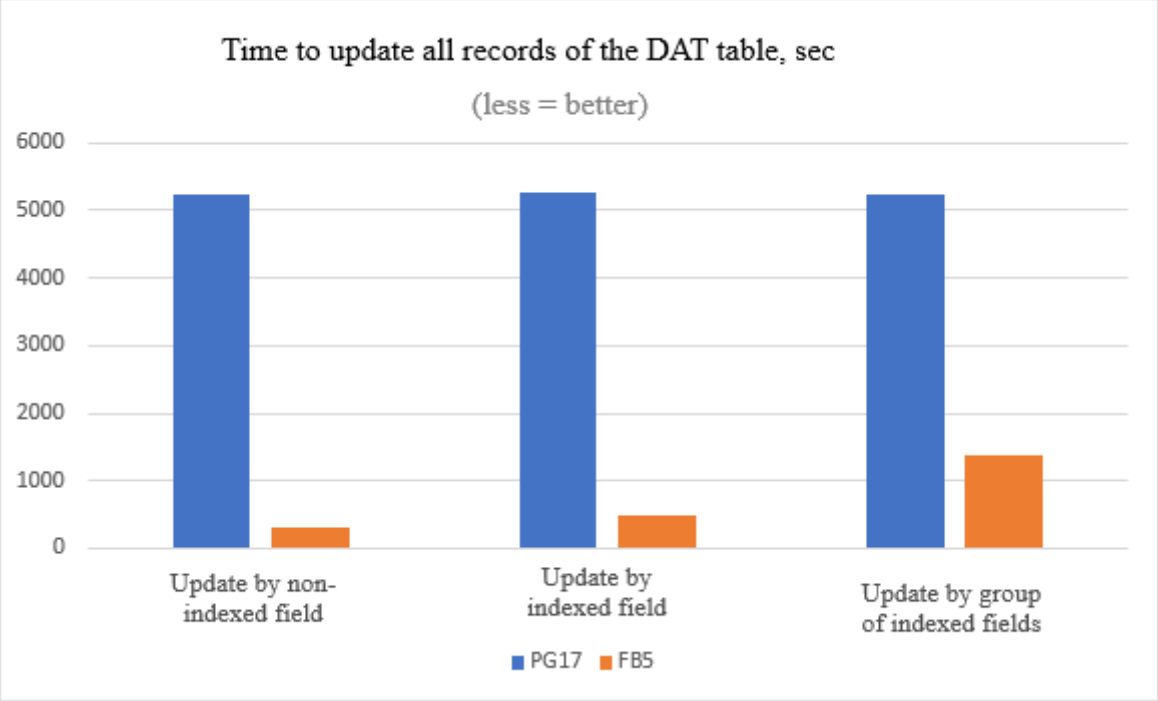
Beim Aktualisieren eines indizierten Tabellenfeldes in Firebird sollten nur die Indizes neu erstellt werden, die auf diesem Feld basieren; beim Aktualisieren in PostgreSQL sollten alle Indizes neu erstellt werden, unabhängig davon, ob die `UPDATE`-Abfrage die Felder aktualisiert, auf denen diese Indizes basieren oder nicht.

Lassen Sie uns das überprüfen:

10 Millionen aktualisierte Datensätze	Firebird 5	PostgreSQL 17
nach nicht-indiziertem Feld `update dat set n8 = 0`	5 Minuten 3 Sekunden	1 Stunde 27 Minuten 23 Sekunden
nach indiziertem Feld `update dat set i4 = -i4`	8 Minuten 16 Sekunden	1 Stunde 27 Minuten 42 Sekunden

10 Millionen aktualisierte Datensätze	Firebird 5	PostgreSQL 17

nach Gruppe indizierter Felder
`update dat`
`set i1 = 0, f2 = true, n4 = 4,`
`d6 = '1974-04-01', s3 = 'FC Bayern',`
`t1 = '1{...}Z'`
	23 Minuten 16 Sekunden	1 Stunde 26 Minuten 29 Sekunden



Wir sehen, dass die praktischen Ergebnisse vollständig mit dem theoretischen Teil korrelieren: Die Update-Zeit in PostgreSQL hängt nicht davon ab, wie viele Felder wir aktualisieren und ob sie indiziert sind, in Firebird schon.

Der Unterschied in der Ausführungszeit solcher Massen-Updates in Firebird und PostgreSQL ist sehr signifikant.

1.2. Test 2

Lassen Sie uns auch die Geschwindigkeit des Einfügens und Löschens von Datensätzen vergleichen:

Löschen von 1 Million Datensätzen aus 10 Millionen

```
delete from dat where id between 3000000 and 3999999
```

Firebird 5.0.2	PostgreSQL 17.4
2,4 Sekunden	2,2 Sekunden

Die Durchschnittsdaten für eine Serie von 50 Messungen werden präsentiert; der Unterschied ist erwartungsgemäß unbedeutend.

INSERT 1 Million Datensätze

```
insert into dat (id, i1, i2, i3, i4, i5, i6, i7, i8, f1, f2, f3, f4, f5, f6, f7, f8, n1, n2,
n3, n4, n5, n6, n7, n8, d1, d2, d3, d4, d5, d6, d7, d8, s1, s2, s3, s4, s5, s6, s7, s8, t1, t2,
t3, t4, t5, t6, t7, t8)
select id+10000000,  i1, i2, i3, i4, i5, i6, i7, i8, f1, f2, f3, f4, f5, f6, f7, f8, n1, n2,
n3, n4, n5, n6, n7, n8, d1, d2, d3, d4, d5, d6, d7, d8, s1, s2, s3, s4, s5, s6, s7, s8, t1, t2,
t3, t4, t5, t6, t7, t8 from dat where id between 1000000 and 1999999
```

Firebird 5.0.2	PostgreSQL 17.4
10 Minuten 36 Sekunden	4 Minuten 38 Sekunden

Hier ist das Ergebnis deutlich besser für PostgreSQL.

Fazit: Sie müssen Ihr Informationssystem analysieren, um Datenverarbeitungsmuster mit ähnlicher Funktionalität vor der Migration zu identifizieren, entscheiden, ob sie benötigt werden, ob sie irgendwie ersetzt oder vollständig aufgegeben werden können.

Zusätzlich müssen Sie berücksichtigen, dass Sie in PostgreSQL nach einem solchen Update und vor der Bereinigung (Garbage Collection) vorübergehend doppelt so viel Speicherplatz benötigen, während dieser Effekt in Firebird deutlich geringer ausfällt – denn der Speicherbedarf für das Erstellen eines Deltas beim Aktualisieren eines bestimmten Feldes ist wesentlich geringer als für das Erstellen einer vollständigen Kopie eines Datensatzes.

2. SUSPEND und RETURN NEXT

Bei der Arbeit mit Firebird sind wir daran gewöhnt, dass bei der Ausführung einer Prozedur, die eine sehr große Anzahl von Zeilen an eine Client-Anwendung zurückgibt, der Firebird-Server dieser Client-Anwendung einen Teil dieser Daten gibt und stoppt, bis er einen Befehl von der Client-Anwendung über die Notwendigkeit erhält, einen neuen Teil auszugeben. Unterstützung für solches Verhalten ist explizit im Code der Client- und Serverteile von Firebird implementiert. In gespeicherten Prozeduren wird dieses Verhalten mit dem `SUSPEND`-Operator unterstützt.

In PostgreSQL ist das nicht der Fall

Aus der Dokumentation:

> Die aktuelle Implementierung von `RETURN NEXT` und `RETURN QUERY` speichert das gesamte Ergebnisset, bevor die Funktion zurückkehrt, wie oben beschrieben. Das bedeutet, dass die Leistung einer PL/pgSQL-Funktion, die eine sehr große Ergebnismenge erzeugt, schlecht sein kann: Die Daten werden auf die Festplatte geschrieben, um einen Speicherüberlauf zu vermeiden, aber die Funktion selbst gibt erst dann ein Ergebnis zurück, wenn das gesamte Ergebnisset erzeugt wurde. In einer zukünftigen Version von PL/pgSQL könnten Benutzer Set-Returning-Funktionen definieren, die diese Einschränkung nicht haben. Der Punkt, an dem Daten auf die

> Festplatte geschrieben werden, wird derzeit durch die Konfigurationsvariable work_mem gesteuert. Administratoren, die über ausreichend Speicher verfügen, um größere Ergebnismengen im Speicher zu halten, sollten diesen Parameter erhöhen.

Seien Sie darauf vorbereitet, dass, wenn Ihr System Prozeduren enthält, deren Ergebnismengen gemäß der Geschäftslogik der Anwendung bisher nicht vollständig abgerufen wurden, diese nach der Migration zu PostgreSQL vollständig abgerufen werden. Dies kann zu einer Verschlechterung der Systemleistung führen.

3. Verwendung autonomer Transaktionen

Der Hauptanwendungsbereich (aber nicht der einzige) autonomer Transaktionen ist die Auditierung, die nicht rückgängig gemacht werden kann.

Hier ist ein Beispiel für die Verwendung einer autonomen Transaktion in Firebird in einem Trigger für ein Datenbankverbindungsereignis, um alle Verbindungsversuche, einschließlich erfolgloser, zu protokollieren (entnommen aus der Quelle Firebird 5.0 SQL Language Reference).

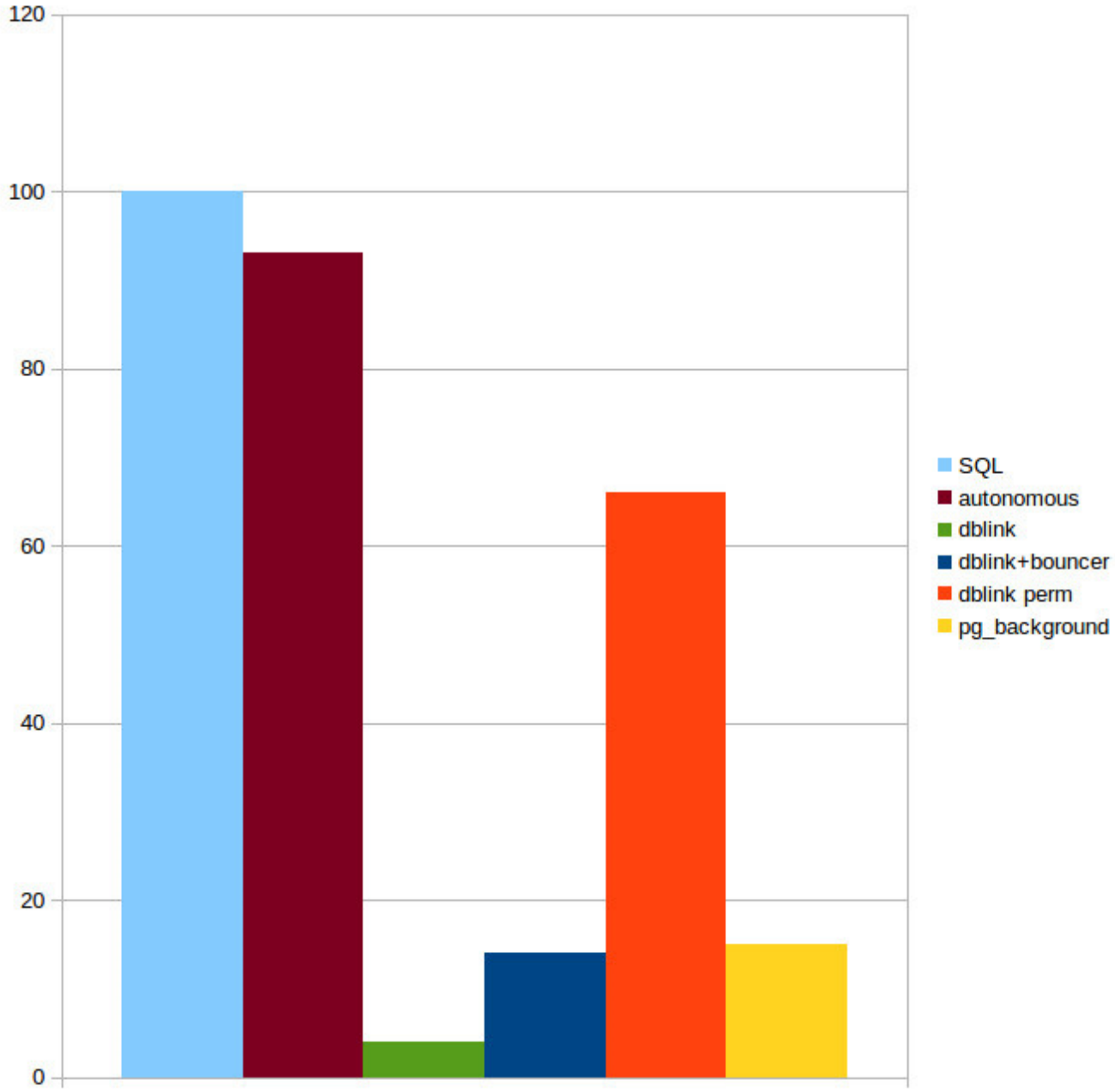
```
CREATE TRIGGER TR_CONNECT ON CONNECT AS
BEGIN
  -- Alle Versuche, sich mit der Datenbank zu verbinden, werden im Log gespeichert
  IN AUTONOMOUS TRANSACTION DO
    INSERT INTO LOG(MSG) VALUES ('USER ' || CURRENT_USER || ' CONNECTS.');
  IF (CURRENT_USER IN (SELECT USERNAME FROM BLOCKED_USERS)) THEN
  BEGIN
    -- im Log speichern, dass der Versuch, sich mit der Datenbank zu verbinden, erfolglos war
    IN AUTONOMOUS TRANSACTION DO
    BEGIN
      INSERT INTO LOG(MSG) VALUES ('USER ' || CURRENT_USER || ' REFUSED.');
      POST_EVENT 'CONNECTION ATTEMPT' || ' BY BLOCKED USER!';
    END
    -- jetzt eine Exception werfen
    EXCEPTION EX_BADUSER;
  END
END
```

Vanilla PostgreSQL hat überhaupt keine autonomen Transaktionen. Sie können durch das Starten einer neuen Verbindung mit `dblink` oder `pg_background` simuliert werden, aber das führt zu Overhead, beeinflusst die Leistung und ist einfach unbequem, aber es ist machbar. Hier ist ein Beispiel für die Implementierung einer Logging-Funktion mit der `dblink`-Erweiterung:

```
CREATE FUNCTION log_dblink(msg text) RETURNS void LANGUAGE sql
AS $function$
  select dblink('host=/var/run/postgresql port=5432 user=postgres dbname=postgres',
                format('insert into log select %L, %L', msg, clock_timestamp()::text))
$function$
```

Nur Postgres Pro (kommerzielle Version von PostgreSQL), Enterprise Edition (nicht einmal Standard!) hat volle Unterstützung für autonome Transaktionen mit Ausführung einer autonomen Transaktion innerhalb desselben Serverprozesses.

Nur die letzte Lösung vermeidet Leistungsverschlechterung, weil es der einzige Ansatz ist, der eine autonome Transaktion innerhalb desselben Serverprozesses ausführt.



Das Diagramm zeigt die Leistung einer regulären SQL-Abfrage (entspricht 100% im Diagramm), einer Abfrage in einer autonomen Transaktion der Postgres Pro Enterprise-Version und unter Verwendung verschiedener Erweiterungen (aus den Postgres Pro-Materialien).

Wenn Sie autonome Transaktionen in Ihrem Firebird-Projekt aktiv für die Auditierung verwenden, berücksichtigen Sie dies.

4. Eingeschränkte Funktionalität anonymer PSQL-Blöcke in PostgreSQL

> Ein anonymer Codeblock nimmt keine Argumente und jeder Wert, den er möglicherweise zurückgibt, wird verworfen. Ansonsten funktioniert er wie Funktionscode.
>
> — PostgreSQL-Dokumentation [(44.4. Anonymous Code Blocks)]

Natürlich kann das Problem mit Eingabeparametern irgendwie durch Makros gelöst werden, indem diese vor der Ausführung der Funktion durch Werte ersetzt werden; das resultierende Set kann in temporäre oder unprotokollierte Tabellen eingefügt werden. Aber erstens ist das umständlich und zweitens erfordert es erhebliche Änderungen am Code und an der Logik der Funktion.

`EXECUTE BLOCK` aus Firebird hat solche Einschränkungen nicht. Hier ein Beispiel für diese Syntaxkonstruktion, die das geometrische Mittel zweier Zahlen berechnet und an den Benutzer zurückgibt:

```
EXECUTE BLOCK(x DOUBLE PRECISION=?, y DOUBLE PRECISION=?)
RETURNS (gmean DOUBLE PRECISION)
AS
BEGIN
  gmean = sqrt(x*y);
  SUSPEND;
END
```

Wie Sie sehen können, kann die gespeicherte Prozedur mit dieser Konstruktion mit minimalen Änderungen am Quellcode ausgeführt werden. Wenn die Syntaxkonstruktion `EXECUTE BLOCK` in Ihrem System häufig verwendet wird, kann die Überarbeitung all dieses Codes während der Migration zu einem Problem werden.

5. Prozesse und Threads

PostgreSQL erstellt für jede Verbindung einen neuen Prozess (keinen Thread). Ohne entsprechende Konfiguration und geeigneten Server können ungeplante Nutzungsspitzen die Datenbank schnell überlasten.

PostgreSQL verwendet das Prozessmodell seit Beginn des Projekts aufgrund seiner Einfachheit, und es gibt laufende Diskussionen über einen Wechsel zu Threads.

Das aktuelle Modell hat einige Nachteile: statisch zugewiesener Shared Memory erlaubt keine dynamische Größenänderung von Strukturen wie dem Buffer Cache; parallele Algorithmen sind schwer zu implementieren und weniger effizient als möglich; Sitzungen sind eng an Prozesse gebunden. Die Verwendung von Threads erscheint vielversprechend, ist jedoch mit Schwierigkeiten bei Isolation, Kompatibilität mit Betriebssystemen und Ressourcenmanagement verbunden. Ganz

zu schweigen davon, dass der Wechsel radikale Änderungen am Code und jahrelange Arbeit erfordern würde. Bisher setzt sich die konservative Sichtweise durch, und Änderungen sind in naher Zukunft nicht zu erwarten.

Die Linux-Implementierung des Multitasking begünstigt dieses Verhalten deutlich und ist wahrscheinlich der Hauptgrund, warum PostgreSQL-Experten ~~empfehlen,~~ ~~dringend empfehlen,~~ **zwingen** PostgreSQL unter Linux zu verwenden. Postgres Pro (kommerzielle Distribution) hat beispielsweise in den letzten Versionen die Windows-Unterstützung vollständig aufgegeben.

Firebird ist nicht besonders empfindlich gegenüber einer großen Anzahl inaktiver Verbindungen. Basierend auf meiner langjährigen Erfahrung mit Informationssystemen unter Firebird (und in der Classic Server-Architektur, die der PostgreSQL-Architektur am nächsten kommt), kann ich sagen, dass weder 500, noch 1000, noch 1500 offene Verbindungen zu spürbaren Leistungsproblemen geführt haben. Die Classic-Architektur ist demnach “Prozess”. Die Architekturen, die die Arbeit mit Threads in Firebird ermöglichen, sind SuperClassic und SuperServer (diese Architektur wird in modernen Firebird-Versionen empfohlen).

Ein Wechsel des Betriebssystems, unter dem Ihr DBMS läuft, bedeutet einen Austausch oder eine Umschulung Ihres Personals. Wenn Ihr Firebird-Server derzeit unter Windows läuft, ist es wichtig, dies im Voraus zu planen.

Wenn Sie Hersteller von verteilter Software sind, müssen Ihre Kunden einen ähnlichen Wechsel vollziehen, und sie werden darüber vermutlich wenig begeistert sein.

Beachten Sie außerdem, dass PostgreSQL unabhängig vom Betriebssystem eine große Anzahl gleichzeitig geöffneter Verbindungen "nicht mag". Daher gibt es das separate Paket pgbouncer (dringend empfohlen) zur Verwaltung des Verbindungspools und den Standardwert des Parameters `max_connections=100`.

Wenn jedoch *pgbouncer* oder ähnliche Connection-Pooler verwendet werden, funktionieren kontextbezogene Variablen auf Verbindungsebene nicht mehr korrekt, was größere Änderungen an Ihren Lösungen erfordern kann, die auf dieser Funktion basieren. Beispielsweise können Sie in Firebird eine Verbindungsvariable mit der aktuellen Benutzer-ID setzen, sodass Sie den Benutzerparameter nicht an jede Prozedur übergeben müssen. In PostgreSQL müssen Sie dafür einen anderen Ansatz wählen.

Ebenso wird es unmöglich, temporäre Tabellen auf Verbindungsebene zu verwenden.

6. Transaktionszähler

Der PostgreSQL-Kernel verfügt über einen 32-Bit-Transaktionszähler, was bedeutet, dass er nicht mehr als 4 Milliarden zählen kann. Dies führt zu Problemen, die durch das „Einfrieren“ gelöst werden – eine spezielle routinemäßige Wartungsprozedur namens `VACUUM FREEZE`.

Dabei handelt es sich um ein aggressives „Einfrieren“ von Tupeln. Es sollte von Zeit zu Zeit für alle Datensätze durchgeführt werden, die weit in der Vergangenheit liegen, und ein Flag setzen, dass dieser Datensatz „eingefroren“ ist, also für alle Transaktionen sichtbar und eine spezielle `FrozenTransactionId` (minus unendlich) besitzt.

Das System erkennt, dass diese Zeile vor langer Zeit erstellt wurde und die Transaktionsnummer, die sie erstellt hat, keine Rolle mehr spielt. Das bedeutet, dass diese Transaktionsnummer wiederverwendet werden kann. Eingefrorene Transaktionsnummern können wiederverwendet werden.

Wenn der Zähler jedoch zu oft überläuft oder eingefrorene Transaktionen das Einfrieren rechtzeitig verhindern, sind die Kosten für diese Prozedur sehr hoch und können sogar dazu führen, dass nichts mehr in die Datenbank geschrieben werden kann.

Wenn Sie es sich jedoch leisten können, Ihren DB-Cluster jederzeit zu stoppen, ihn im Einzelbenutzermodus zu starten, um den Befehl `VACUUM FREEZE` auszuführen und einige Stunden zu warten, betrifft Sie dieses Problem nicht.

64-Bit-Transaktionszähler sind eine grundlegende Überarbeitung des DBMS-Kernels und werden nur für stark ausgelastete Systeme benötigt, für diese sind sie jedoch nicht nur wünschenswert, sondern notwendig.

Deshalb implementieren jetzt ALLE kommerziellen Versionen von PostgreSQL einen 64-Bit-Transaktionszähler.

Die bekannteste und erfolgreichste Implementierung stammt von Postgres Pro, die als Erste weltweit eine Lösung angeboten haben.

Es gibt vier Gründe, warum diese Lösung nicht in der Vanilla-Version enthalten ist:

1. Refactoring einer großen Menge Code – der gesamte andere PostgreSQL-Code erwartet, dass der Zähler eine 32-Bit-Zahl ist
2. Jede Zeilenversion hat einen Header. Wenn der Zähler 64 Bit ist, gibt es zu viele Serviceinformationen für jede Zeilenversion.
3. Eine solche Änderung erfordert die Überarbeitung vieler PostgreSQL-Erweiterungen, die verstehen müssen, dass die Transaktionsnummer jetzt 64 Bit ist
4. „Trägheit“ der Community. Die Lösung von Postgres Professional wurde von der Community wegen ihrer Größe und Komplexität abgelehnt. Der aktuelle Status ist mysteriös… „Returned with feedback“.

Was Firebird betrifft: Wenn der Transaktionszähler „erschöpft“ ist, ist eine vollständige DB-Wartung erforderlich, aber die Transaktionszähler in Firebird sind seit Version 3.0 (veröffentlicht 2016) 48-Bit und das beschriebene Problem ist eher theoretisch als praxisrelevant.

7. Arbeiten mit Speichersystemen

Der Markt für Datenspeichersysteme (auch SAN genannt) zeigt stetiges Wachstum, wobei die Hauptinvestitionen aus dem Unternehmensbereich beobachtet werden. Programme zur digitalen Transformation und neue Online-Services erfordern ständige Infrastruktur-Upgrades, was die Nachfrage nach Hochleistungssystemen erhöht.

Wenn Ihr System Abfragen hat, die sehr schnell ausgeführt werden müssen (zehn Millisekunden), aber sehr oft, dann werden Sie nach der Migration einer spürbaren Erhöhung der

Gesamtausführungszeit solcher Abfragen gegenüberstehen, wenn sich die Datenbank auf dem SDS befindet.

Der Grund ist, dass die PostgreSQL-Datenbank Hunderttausende von Dateien enthalten kann, und für jede Datei entstehen beim Übertragen von Informationen zwischen dem Server und dem SAN zusätzliche Kosten für die Übertragung von Daten über den Dateinamen, Serviceinformationen usw.

Sehr grob kann dies mit der Situation verglichen werden, wenn Sie eine 10-GB-Datei über das Netzwerk kopieren oder 1000 Dateien von 10 MB. Ich hoffe, jeder kennt den Zeitunterschied zwischen diesen Operationen.

Sie können mehr über diese Situation hier lesen.

8. Backup-Unterschiede

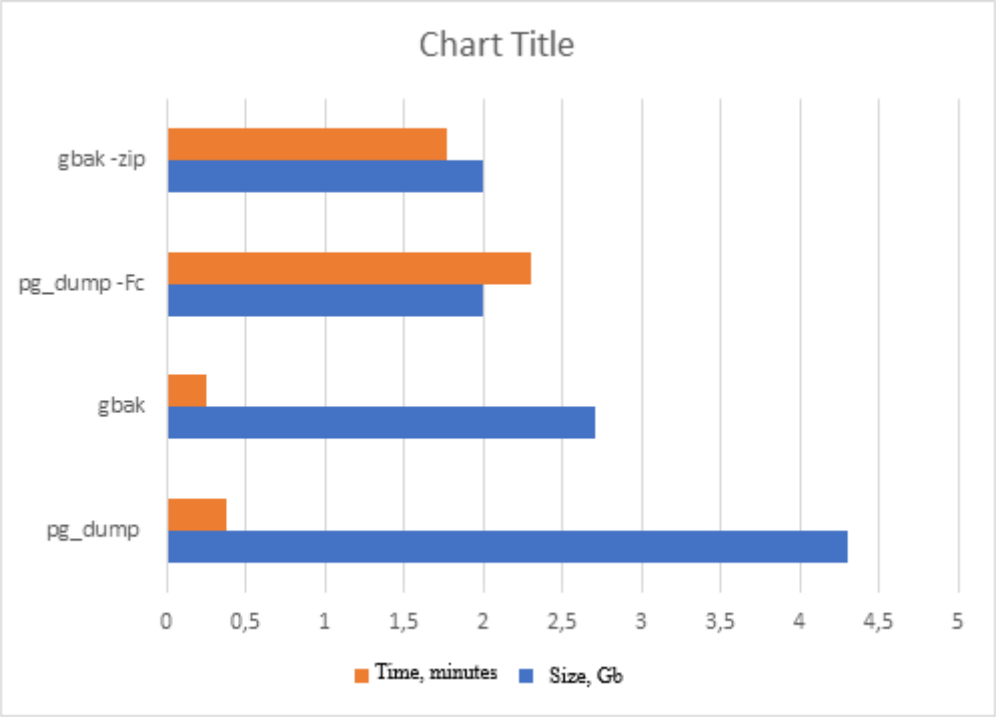
In PostgreSQL funktionieren Backups völlig anders als in anderen Datenbanken - so anders, dass es einen kompletten mentalen Wechsel erfordert und das Aufgeben von allem, was Sie über Backups aus der Arbeit mit anderen Datenbanksystemen gelernt haben.

8.1. Gbak & pg_dump

Es gibt ein Analogon von gbak für vollständige Backups in PostgreSQL: pg_dump/pg_restore, und hier scheint alles ungefähr gleich zu sein:

- Backup einer Datenbank
- Sie können die erforderlichen Tabellen auswählen
- Multi-threaded (in PostgreSQL — mit Einschränkungen beim Format der Ausgabedatei)
- kein Inkrement
- Möglichkeit, eine komprimierte Kopie zu erhalten

Erstellen einer Backup-Kopie am Beispiel einer Datenbank mit einer `DAT`-Tabelle:



Aber `pg_dump` ist nicht geeignet für die Erstellung von Echtzeit-Backups großer Datenbanken, da es erhebliche Ressourcen und Ausführungszeit erfordern kann, was sich negativ auf die Systemleistung auswirken kann. Mit `gbak` wurde ein solches Problem seit Firebird 2.5 nicht beobachtet.

8.2. Nbackup & pg_basebackup

Um Echtzeit-Backups zu erstellen, gibt es in Firebird `nbackup` und in PostgreSQL `pg_basebackup`.

Und hier ist wirklich alles anders:

`pg_basebackup` sichert immer den gesamten Cluster. Wenn Sie nur eine Datenbank wiederherstellen möchten, wird Ihr gesamter PostgreSQL-Cluster wiederhergestellt, der dann nur eine Datenbank enthält. Die Dateien aller anderen Datenbanken sind dann leer (Größe null).

Nachteile: eingeschränkte Möglichkeiten im Multi-Threading, eingeschränkte Erstellung von inkrementellen Backups, weniger Flexibilität als `pg_dump` bei der Auswahl einzelner Objekte für das Backup, geringe Kopiergeschwindigkeit.

`pg_probackup` von Postgres Professional bietet viele Vorteile gegenüber `pg_basebackup`, aber die meisten davon sind nur in der kommerziellen Version verfügbar.

Das ist eigentlich typisch – wer käme auf die Idee, eine kommerzielle Version von Firebird nbackup zu schreiben? Es funktioniert einfach perfekt!

Um schnelle und zuverlässige Backups in PostgreSQL zu erstellen, müssen Sie das Wichtigste verstehen: Der Begriff “Datenbank” im PostgreSQL-Backup ist irreführend. Nennen Sie es wie Sie wollen, aber arbeiten Sie immer nur mit Clustern. Teilen Sie große Datenbanken in separate Cluster auf anderen Ports auf. Glauben Sie mir, das ist das kleinere Übel (wenn man den Speicherverbrauch berücksichtigt), als alle “Datenbanken” in einem Cluster zu halten.

Selbst ein Cold Backup/Restore ist nicht so einfach, und ein Online Backup/Restore über `pg_basebackup` ist noch schwieriger, da es im Grunde das Hochziehen einer Replik ist.

`Nbackup` in Firebird ist einfacher, verständlicher und vor allem – bequemer.

9. Code-Migrationsprobleme mit temporären Tabellen

Es gibt ein Problem in PostgreSQL, das wir ziemlich häufig antreffen. Leider weist die Implementierung der Logik für temporäre Tabellen eine Reihe von Mängeln auf, die sich negativ auf die Systemleistung auswirken.

Diese Tatsache ist allgemein bekannt, ebenso wie die Frage “Wir haben temporäre Tabellen in <IRGENDEINER> SQL-Datenbank verwendet und hatten keine Probleme. Warum passiert das in PostgreSQL?”, die in Foren sehr häufig gestellt wird.

- Das aktive Erstellen, Ändern und Löschen von temporären Tabellen (einschließlich Arbeiten mit Indizes) kann zu erheblichen Leistungseinbußen führen
- AUTOVACUUM sieht temporäre Tabellen nicht – die entsprechenden Bereinigungs- und Analyseoperationen müssen manuell durchgeführt werden
- Ein typisches Muster beim Arbeiten mit temporären Tabellen führt zu einer Überfüllung des OS-Caches mit Datenmüll
- pg_class (und andere Systemtabellen) wachsen während des Betriebs
- Schreibzugriffe auf temporäre Tabellen sind in ReadOnly-Replikas verboten (z. B. beim Erstellen von Berichten)

Angesichts der weitverbreiteten Nutzung temporärer Tabellen in modernen Lösungen kann dies zu Problemen führen.

Mögliche Lösungen:

- Platzieren temporärer Tabellen auf einem RAM-Disk
- Verwendung von nicht protokollierten (UNLOGGED) Tabellen anstelle von temporären Tabellen
- Einsatz kommerzieller Versionen von PostgreSQL, in denen der Mechanismus für temporäre Tabellen deutlich überarbeitet wurde
- Begrenzung/Vermeidung der Nutzung temporärer Tabellen in Ihren Anwendungen

In Firebird gilt:

- Temporäre Tabellen werden außerhalb der Datenbank in temporären Dateien erstellt
- Forced Writes ist für sie immer AUS
- Sie werden sofort bereinigt

Es gibt keine Probleme bei der Arbeit mit temporären Tabellen, was bedeutet, dass während der Systemmigration Schwierigkeiten auftreten können, die durch die oben genannten Besonderheiten

von PostgreSQL verursacht werden. Seien Sie vorsichtig!

Wir weisen nochmals darauf hin, dass bei Verwendung von `pg_bouncer` der Code, der zuvor verbindungsbezogene GTT genutzt hat, nicht mehr funktionieren kann.

10. Verhalten des Abfrageoptimierers

PostgreSQL verfügt über einen relativ einfachen, aber schnellen Abfrageoptimierer. Allerdings hat dieser Algorithmus ein großes Problem: Er erstellt einen Ausführungsplan und hält sich daran, selbst wenn sich dieser als falsch herausstellt. Im schlimmsten Fall kann es passieren, dass PostgreSQL in Zwischenberechnungen mit 1–2 Datensätzen rechnet, tatsächlich aber tausendfach mehr vorhanden sind. Die Ausführung eines Nested Loop führt dann zu enormer Komplexität und kann zu einem Prozess-Hänger mit hoher CPU-Auslastung führen. Leider bietet die Standardversion von PostgreSQL keine Möglichkeit, wie in anderen Datenbanksystemen sogenannte Hints zu setzen.

Der Vollständigkeit halber sei erwähnt, dass es spezielle Erweiterungen zur Lösung dieses Problems gibt – zum Beispiel pg_hint_plan.

Das Argument lautet oft: „Man muss korrekte Abfragen schreiben, nicht sie nachträglich korrigieren.“ Sie werden solche problematischen Abfragen also während der Migration so umschreiben müssen, dass sie für den PostgreSQL-Optimierer „korrekt“ sind.

Wer schon einmal von einer Firebird-Version auf eine andere migriert hat, weiß, dass sich das Verhalten des Abfrageoptimierers ändern kann – ganz zu schweigen von den Änderungen beim Wechsel des Datenbanksystems.

11. Datengrößen

PostgreSQL verfügt nicht über eine Komprimierung auf Seitenebene. Es werden nur TOAST-Daten komprimiert. Wenn die Datenbank viele Datensätze mit relativ kleinen Textfeldern enthält, könnte eine Komprimierung die Größe der Datenbank um ein Vielfaches reduzieren, was nicht nur Speicherplatz spart, sondern auch die Leistung des DBMS erhöht. Besonders analytische Abfragen, die viele Daten von der Festplatte lesen und diese nicht zu oft ändern, können durch die Reduzierung der Ein-/Ausgabeoperationen deutlich beschleunigt werden.

Darüber hinaus dürfen wir nicht vergessen, dass bei großen Produktionsdatenbanken das Transaktionsarchiv für den Tag die Datenbankgröße um ein Vielfaches überschreiten kann.

Die PostgreSQL-Community empfiehlt zur Komprimierung die Verwendung von Dateisystemen mit Komprimierungsunterstützung. Das ist jedoch nicht immer praktisch und möglich. Bereiten Sie sich erneut darauf vor, mehrere Tausend USD pro Kern für kommerzielle Versionen mit Seitenkomprimierung zu zahlen!

Um diese These zu testen, wurde eine Tabelle kombinierter Nachschlagewerke aus einer Firebird-5-Datenbank eines bestimmten Transport- und Logistikunternehmens verwendet. Sie enthielt etwa 700.000 Zeilen und viele Felder. Diese Tabelle wurde in eine PostgreSQL-17-Datenbank konvertiert. Die Tabelle enthielt folgende Datentypen: Zeichenketten, Daten, Flags, Ganzzahlen und reelle

Zahlen, UUIDs. Die Zeichenkodierung ist utf8.

Die Größe der Tabellendaten wurde vor und nach der Konvertierung gemessen.

Ergebnis: Nach der Konvertierung stieg der von den Daten dieser Tabelle belegte Speicherplatz in der Datenbank exakt um das Anderthalbfache.

Beachten Sie, dass vor der Veröffentlichung von Firebird 5 das Packen von Textdaten mit utf8-Kodierung in Firebird deutlich weniger effizient war als heute.

Es sollte auch bedacht werden, dass aufgrund der MVCC-Implementierung beim vollständigen Aktualisieren eines Feldes in der gesamten Tabelle PostgreSQL für einige Zeit Festplattenspeicher in Höhe der gesamten Tabellengröße benötigt, um die Operation auszuführen.

Werfen Sie einen Blick auf diesen Artikel mit einer detaillierten Studie und einem Vergleich: https://firebirdsql.org/img/articles/fb_vs_pg_datafill/fb_vs_pg_datafill.html

12. Konsequenzen hängender Transaktionen. VACUUM FULL und GFIX -sweep

Einige Probleme in PostgreSQL, die durch hängende Transaktionen entstehen (Tabellen- und Index-Bloat), werden nur durch `VACUUM FULL` behoben, und dieser:

- benötigt deutlich mehr Zeit als ein normales `VACUUM`
- fordert ein exklusives Sperren der Tabelle
- benötigt zusätzlichen Speicherplatz (schreibt eine neue Kopie der Tabelle und gibt die alte erst nach Abschluss der Operation frei)

Der Firebird-Befehl `gfix -sweep` behebt ähnliche Probleme, die typisch für die Multi-Version-Architektur sind, im Online-Modus und ist eine schnelle, mehrthreadige Lösung.

Erwähnenswert ist die Erweiterung `pg_repack` aus dem kommerziellen Postgres Pro, die eine Tabellenreorganisation ohne exklusives Sperren ermöglicht.

13. Eine Transaktion pro Verbindung

Firebird unterstützt mehrere Transaktionen pro Verbindung. Falls Ihr Informationssystem unter Firebird diese Funktionalität nutzt, muss es umgeschrieben werden – PostgreSQL bietet diese Möglichkeit nicht.

Damit können mehrere Transaktionen mit unterschiedlichen Isolationsstufen innerhalb einer Verbindung ausgeführt werden.

Die häufigste Anwendung ist das Lesen von Daten in einer langen, nur-lesenden Transaktion und das Ändern dieser Daten in kurzen Schreib-Transaktionen. Dieses Verhalten wurde immer von beliebten Firebird-Zugriffskomponenten unterstützt, sowohl von veralteten (FibPlus) als auch von modernen (FireDac, UniDac). Vergessen Sie diesen Punkt nicht bei der Migration Ihrer Anwendung

von Firebird zu PostgreSQL.

14. Keine Paket-Implementierung

Pakete… Wie praktisch sie während der Entwicklung sind. Die Möglichkeit, Prozeduren und Funktionen nach Geschäftslogik oder Typ zu gruppieren, die Möglichkeit, das gesamte Paket zu aktualisieren, statt eine Menge einzelner Prozeduren und Funktionen… Zum Beispiel gibt es in unserem System das Dokument “Bestellung an Lieferanten”. Alle Funktionen und Prozeduren, die mit diesem Dokument arbeiten, werden im Paket implementiert.

Nach Änderungen ist das Update des Systems einfach – wir aktualisieren das gesamte Paket in der Ziel-Datenbank und nicht eine Vielzahl von Prozeduren und Funktionen, die wir bei der neuen Version geändert haben.

Nach der Migration müssen Sie auf Pakete verzichten – sie sind in der Standardversion von PostgreSQL nicht verfügbar, in kommerziellen Versionen werden sie über Schemas umgesetzt und sind nicht so komfortabel. Falls Sie keine Pakete in Firebird nutzen – schade, denn sie sind wirklich sehr praktisch. Falls Sie sie nutzen und zu PostgreSQL migrieren möchten – ebenfalls schade, denn Sie müssen nicht nur alles umschreiben, sondern auch auf Pakete verzichten.

15. Tabellen ändern

Wenn Sie nach der Migration ein Feld zu einer Tabelle hinzufügen möchten, ist das kein Problem! Aber nur am Ende der Tabelle. Wenn Sie die Felder in einer bestimmten Reihenfolge in der Tabelle haben möchten (z. B. ein Feld irgendwo in der Mitte einfügen), kann dies nur durch das Neuerstellen und vollständige Wiederhochladen der Tabelle erreicht werden. Tatsächlich werden auch in Firebird Felder auf Format-Ebene am Ende der Tabelle hinzugefügt. Es gibt jedoch in den Systemtabellen ein zusätzliches Feld, das für die Reihenfolge der Feldausgabe in der IDE oder bei der Ausführung einer `SELECT`-Abfrage mit * verantwortlich ist. Das ist eine kleine Verbesserung für das DBMS. PostgreSQL hat dies nicht.

16. Überprüfung von Abhängigkeiten beim Kompilieren von Prozeduren

Wenn Sie es gewohnt sind, beim Kompilieren von gespeicherten Prozeduren und Triggern Abhängigkeiten und Metadaten zu prüfen, gewöhnen Sie sich das ab. In PostgreSQL erfahren Sie, dass Sie einen Fehler im Tabellennamen im Prozedurcode gemacht haben, erst wenn dieser ausgeführt wird. Das ist in der Produktion nicht sehr angenehm.

Sie müssen diese Realität einfach akzeptieren, wenn Sie zu PostgreSQL wechseln. Wenn Abfragen zu bestimmten Tabellen nur unter bestimmten Geschäftsbedingungen ausgeführt werden, gilt: Je komplexer Ihr Prozedur- oder Funktionscode wird, desto höher ist die Wahrscheinlichkeit, dass er einen Fehler im Namen einer Tabelle oder Ansicht enthält. Sie entdecken diesen Fehler erst, wenn genau dieser (wahrscheinlich sehr seltene) Codepfad ausgeführt wird.

17. SQL-Code-Migration

Auch wenn Sie nicht viele gespeicherte Prozeduren, UDFs oder UDRs umschreiben müssen, werden Sie auf Inkompatibilitäten der SQL-Syntax zwischen Firebird und PostgreSQL stoßen. Außerdem werden Sie in der Migrationsphase nicht von syntaktischem Zucker profitieren, den PostgreSQL bietet, Firebird aber nicht, sondern auf Schwierigkeiten stoßen, weil PostgreSQL einige Firebird-Features nicht hat oder weil es Firebird-spezifische Syntaxkonstrukte in Abfragen gibt (zum Beispiel `SELECT FIRST … SKIP`).

Was ich am meisten vermisse, ist die sehr praktische Firebird-Syntax `UPDATE OR INSERT`, deren Umsetzung in PostgreSQL über `ON CONFLICT…` meiner Meinung nach weit weniger komfortabel ist.

Umsetzung von UPSERT in PostgreSQL:

```
INSERT INTO director
  (id, name, fd1, fd2, fd3, fd4, fd5)
  VALUES (1, 'director Name',1,2,3,4,5)
ON CONFLICT (id)
DO UPDATE
SET name = EXCLUDED.name
  , fd1 = EXCLUDED.fd1
  , fd2 = EXCLUDED.fd2
  , fd3 = EXCLUDED.fd3
  , fd4 = EXCLUDED.fd4
  , fd5 = EXCLUDED.fd5;
```

Vergleichen Sie dies mit der Firebird-Implementierung:

```
update or insert
into director (id, name, fd1, fd2, fd3, fd4, fd5)
VALUES (1, 'director Name',1,2,3,4,5)
matching (Id)
```

Durch Hinzufügen der Klausel `returning old.id, new.id into :old_id, :new_id` zu der angegebenen Konstruktion in der Prozedur können wir feststellen, ob die nächste Zeile eingefügt oder aktualisiert wurde.

18. Typinkompatibilität

Postgres verfügt über deutlich mehr Datentypen, was bei der Migration jedoch wenig hilft. Die Existenz von Firebird-Typen, die in Postgres anders deklariert werden müssen (BLOB, DECIMAL, INT128, DECFLOAT …), kann zu einigen Schwierigkeiten führen.

Besonders erwähnenswert ist der DECFLOAT-Typ, ein numerischer Typ aus dem SQL:2016-Standard, der Fließkommazahlen exakt speichert. Im Gegensatz zu DECFLOAT liefern die Typen FLOAT oder DOUBLE PRECISION nur eine binäre Annäherung an die erwartete Genauigkeit. Soweit ich weiß, wird dieser Typ nur von Firebird und DB2 unterstützt.

19. Unterschiedliche Trigger-Implementierungen

Ein Trigger in PostgreSQL besteht aus zwei Komponenten: einer Bedingung, die bestimmt, wann der Trigger ausgelöst wird, und einer Aktion (Triggerfunktion), die ausgeführt wird, wenn der Trigger ausgelöst wird. In Firebird ist dies eine einzelne Komponentenfunktion. Die Implementierung von Triggern in PostgreSQL ist umfangreicher und funktionaler, erfordert bei der Migration jedoch eine Überarbeitung des Codes. Wenn Ihr System viele Trigger enthält, kann deren Anpassung an die Besonderheiten von PostgreSQL einige Zeit in Anspruch nehmen.

```
CREATE TABLE orders (
  order_id bigint, name VARCHAR(100),
  quantity INT, updated_at TIMESTAMP
);
```

Trigger in Firebird:

```
CREATE OR ALTER TRIGGER ORDERS_TBI FOR ORDERS
ACTIVE BEFORE INSERT POSITION 0 AS
begin
  new.updated_at = localTimestamp;
end
```

Trigger in PostgreSQL:

```
CREATE OR REPLACE FUNCTION set_updated_at()
RETURNS TRIGGER AS $$
BEGIN
    NEW.updated_at := NOW();
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER before_update_orders
BEFORE UPDATE ON orders
FOR EACH ROW
  EXECUTE FUNCTION set_updated_at();
```

20. Datenbank-Tools

Wenn Sie einen Manager für die Arbeit mit PostgreSQL benötigen, der sich auf die Entwicklung und das Debuggen von gespeicherten Prozeduren konzentriert, dann sind Ihre Optionen: “EMS SQL Manager for Postgres SQL”, “Postgres SQL Maestro” oder “Navicat Premium”. Alle sind den Tools, die wir bei der Arbeit mit Firebird gewohnt sind, deutlich unterlegen.

21. Komplexere Administration

Seien Sie darauf vorbereitet, dass Sie nach der Migration zusätzliche personelle Ressourcen für die Administration und den Support Ihrer Datenbanken benötigen. PostgreSQL ist schwieriger zu administrieren als Firebird und wächst sehr schnell mit verschiedenen Dienstprogrammen und Add-ons wie `pgbouncer`, `pgpool`, `patroni` und Dutzenden anderen, die ebenfalls “betreut” werden müssen.

Dadurch kann das monatliche Budget für die Entwicklung und Wartung des Systems deutlich steigen.

22. Gewohnheit ist zweite Natur

Anders als bei Firebird werden in PostgreSQL-Prozeduren und -Funktionen Ein- und Ausgabewerte nicht mit ":" gekennzeichnet. Bei Mehrdeutigkeiten führt dies zum Fehler "column is ambiguous" und verursacht anfangs leichte Verwirrung und Unbehagen – wie geht man damit um?

Wie Sie dieses Unbehagen überwinden können, erfahren Sie in diesem Artikel.

Sich daran zu gewöhnen, dass jedes IF ein END IF benötigt, dauert ebenfalls eine Weile. Anfangs vergisst man es ständig, und die Tatsache, dass dieser Fehler überall erkannt wird, nur nicht dort, wo man das `END IF` vergessen hat, hilft wenig.

23. Fazit

Wichtiger Hinweis: Dieser Artikel soll PostgreSQL nicht kritisieren. PostgreSQL ist ein ausgezeichnetes Datenbankmanagementsystem mit vielen fortschrittlichen Funktionen, die Firebird nicht hat (genauso wie Firebird sehr gut ist und viele Funktionen bietet, die PostgreSQL nicht hat).

Wenn Ihr Informationssystem in der Firebird-Umgebung entworfen und entwickelt wurde, ist es Teil dieses Ökosystems. Stellen Sie sich das wie eine Pflanze vor, die in einem bestimmten Klima wächst. Wenn Sie diese Pflanze plötzlich in ein völlig anderes Klima versetzen, wird sie ernsthafte Schwierigkeiten haben zu überleben.

Dasselbe passiert bei einer Datenbankmigration. Ihr System hat sich an die spezifischen Funktionen, Verhaltensweisen und Eigenschaften von Firebird angepasst. Ein plötzlicher Wechsel zu PostgreSQL (oder einem anderen DBMS) wird definitiv Probleme und Komplikationen verursachen.

23.1. Wichtige Fragen, die Sie sich vor der Migration stellen sollten

Bevor Sie ein Migrationsprojekt starten, sollten Sie klare und triftige Gründe haben. Stellen Sie sich diese wichtigen Fragen:

1. Was ist das eigentliche Problem?

1. Stehen Sie vor tatsächlichen technischen Einschränkungen mit Firebird? Erstellen Sie eine Liste der Einschränkungen und prüfen Sie diese anhand der Dokumentation. Finden Sie heraus, ob andere Unternehmen oder Entwickler mit denselben Einschränkungen konfrontiert waren und wie sie diese gelöst haben.
2. Erfüllt Ihre aktuelle Datenbank die Geschäftsanforderungen nicht? Bestimmen Sie genau, was nicht funktioniert.
3. Haben Sie Leistungsprobleme, die in Firebird nicht gelöst werden können? Stellen Sie sicher, dass Sie die Probleme auf Firebird eingegrenzt haben und Hardware-, VM- und Betriebssystemeinstellungen ausgeschlossen sind.

2. Was sind Ihre erwarteten Vorteile?

1. Wird die Migration Ihre aktuellen Probleme lösen?
2. Welche neuen Möglichkeiten werden Sie gewinnen?
3. Wie wird dies Ihre Geschäftsabläufe verbessern?
4. Können Sie den erwarteten Return on Investment der Migration messen?

3. Haben Sie alle Alternativen geprüft?

1. Können Sie Ihre aktuelle Firebird-Version aktualisieren?
2. Gibt es Firebird-basierte Lösungen für Ihre Probleme?
3. Haben Sie Ihr aktuelles Datenbankdesign optimiert?
4. Könnten Drittanbieter-Tools helfen, ohne zu migrieren?
5. Haben Sie professionelle Hilfe angefragt?

24. Kontakt

Bitte zögern Sie nicht, Alexey Kovyazin bei allen Fragen zu kontaktieren: ak@firebirdsql.org.

Wenn Sie eine Datenbankmigration von Firebird zu PostgreSQL oder einer anderen Datenbank in Betracht ziehen, teilen Sie uns bitte Ihre Bedenken und Fragen mit.