PostgreSQLs Datenbestände sichern

Stefan Schumacher
Stefan.Schumacher@Kaishakunin.com

KAISHAKUNIN.COM
IT-Sicherheitsberatung

Veröffentlicht in der GUUG UpTimes Juni 2007.

Literaturverzeichnis

Schumacher, Stefan (2007). „PostgreSQLs Datenbestände sichern“. Deutsch. In: *UpTimes* 2, Seiten 4–7. ISSN: 1860-7683. URL: http://kaishakunin.com/publ/guug-uptimes-postgresql.pdf (besucht am 07. 11. 2009).

```
@article{Schumacher-ut-pg,
    author = {Stefan Schumacher},
    title = {PostgreSQLs Datenbestände sichern},
    year = {2007},
    month = "Jun",
    pages="4\,--\,7",
    volume="2",
    publisher = {German Unix User Group},
    journaltitle="UpTimes",
    issn="1860-7683",
    language = {german},
    url={http://kaishakunin.com/publ/guug-uptimes-postgresql.pdf},
    urldate="2009-11-07",
    pages="4\,--\,7",
}
```

PostgreSQLs Datenbestände sichern

Stefan Schumacher

Dieser Artikel beschreibt die Möglichkeiten, Datenbestände aus PostgreSQL zu sichern. Hierzu stehen verschiedene Methoden zur Verfügung, die allesamt über Vor- und Nachteile verfügen. Sie werden näher vorgestellt und bewertet.

Sicherung eines PostgreSQL-Servers

Daten vorzuhalten ist die wichtigste Aufgabe eines Datenbankservers. Ebenso wichtig ist die Integrität der Daten sicherzustellen. Hierzu gehören neben verschiedenen Datenbanktechnologien auch die Möglichkeiten die Daten zu sichern. Besonders problematisch ist dabei meist die Komplexität der Daten und Metadaten, so dass verschiedene Ansätze zur Sicherung existieren. Dies wäre zum Beispiel die Sicherung der Datenverzeichnisse eines Servers, sowie ein logisches Backup, in dem einfach nur die Daten und ggf. Metadaten der Datenbank in eine Textdatei geschrieben werden. Weiterhin unterstützt PostgreSQL sogenannte Write-Ahead-Logs, in denen alle Transaktionen mitgeloggt werden, und den Einsatz von Replikationstechniken, die den Datenbestand auf mehrere Systeme verteilen können.

Den Datenbankcluster sichern

PostgreSQL legt alle Daten – auch die Metadaten – im sogenannten Cluster, also einem bestimmten Verzeichnis, ab. Es ist möglich, dieses Verzeichnis wie normale Dateien zu sichern, nach der Wiederherstellung des Servers zurückzuspielen und PostgreSQL dann mit diesem Cluster zu starten. Damit befindet sich das neue System wieder in exakt dem selben Zustand wie das gesicherte, inklusive Rollen-Namen, Passwörter und Zugriffsrechten.

Problematisch bei der Sicherung ist die Konsistenz des Dateisystems, da im Cluster die Schreibaktivität entsprechend hoch ist. Abhilfe schaffen hier Dateisystem-Snapshots (unter NetBSD ist dies fss(4)), die einen konsistenten Zustand des Dateisystems als Pseudogerät einbinden, das anschließend gesichert werden kann. Allerdings wird so nur die Integrität des Dateisystems sichergestellt, nicht jedoch die Integrität der Datenbank. Denn es kann sein, das der Dateisystem-Snapshot zur Laufzeit einer Transaktion erstellt wird. Der so gesicherte Cluster ist inkonsistent und daher nicht brauchbar. Die Datenbanken-Konsistenz kann nur sichergestellt werden, wenn keine Transaktionen laufen, die Datenbank also komplett inaktiv ist – was im Normalfall heisst das sie abgeschaltet ist. Es besteht jedoch die Möglichkeit, einen so gesicherten Cluster mit ebenfalls gesicherten Write-Ahead-Logs wieder brauchbar zu machen. Dies wird im Abschnitt ≫Point-in-Time-Recovery≪ beschrieben.

Ein gesicherter PostgreSQL-Cluster kann allerdings nur mit genau derselben PostgreSQL-Version gelesen werden, mit der er erzeugt wurde. Dies bedeutet, dass nach einem Totalausfall des Systems wieder die alte PostgreSQL-Version installiert werden muss.

Es empfiehlt sich, den PostgreSQL-Cluster auf eigenen Partitionen, oder besser noch, eigenen Festplatten abzulegen. Dies verringert den Administrationsaufwand und kann den Datendurchsatz des Servers erhöhen.

Seit Version 8.0 unterstützt PostgreSQL sog. *Table-Spaces*, d.h. dass komplette Datenbanken, aber auch einzelne Tabellen, Indizes oder Schemas auf beliebige Verzeichnisse verteilt werden können. Dies ermöglicht den Einsatz mehrerer Festplatten, was Kapazität und Durchsatz erhöhen kann. Allerdings sollte beim Sichern des PostgreSQL-Clusters dann auch bedacht werden, alle eingesetzten Table-Spaces mitzusichern.

Abbildung 1 zeigt, wie man PostgreSQLs Cluster sichert und wieder zurückspielt. Dabei wird mit fssconfig(8) ein Dateisystem-Snapshot erstellt und mit dump(8) gesichert. Der zweite Teil zeigt die Rücksicherung mit restore(8) und die Konfiguration des neuen Clusters, sowie den Start des Servers.

Point-in-Time-Recovery

Der Einsatz von Point-in-Time-Recovery-Mechanismen (PiTR) ermöglicht das Zurücksetzen der Datenbank in einen konsistenten Zustand, der an einem beliebigen Zeitpunkt vorgelegen hat. Dazu werden in PostgreSQL ab Version 8.0 die sogenannten ≫*Write Ahead Logs*≪ (WAL) gesichert, und somit jede Änderung am Datenbestand mitgeschnitten. Muss das System zurückgesetzt werden, können die Änderungen seit dem letzten Checkpoint abgespielt werden, um einen konsistenten Zustand zu erreichen. Befindet sich ein wie im Abschnitt ≫Den Datenbankcluster absichern≪ gesicherter Cluster in einem inkonsistenten Zustand, wird durch Einspielen der WALs ein konsistenter Zustand hergestellt.

Die WALs werden ähnlich einem Journal fortlaufend weitergeschrieben.

[1] PostgreSQL behandelt *jede* DML-Aktion als ACID-konforme Transaktion.

```
# fssconfing -x -c fss0 /pgcluster0
# dump -0a -f /mnt/nfs/back/pcluster0.0 /dev/fss0
# fssconfig -u fss0

# restore -r -f /mnt/nfs/back/pcluster0.0
# chown pgsql.pgsql /pgcluster0
# su -m pgsql -c 'pg_ctl -D /pgcluster0 -l /pgcluster0/logfile start&'
```

Abbildung 1: PostgreSQL-Cluster mit dump sichern

So werden alle Transaktionen[1] protokolliert und können bei Bedarf neu eingespielt werden. Innerhalb einer bestimmten Zeit werden die Transaktionen dann auf den eigentlichen Datenbestand im Cluster übernommen. Dies erhöht sogar den Datendurchsatz, da es einfacher ist, Daten ersteinmal einfach an eine Datei anzuhängen, als ständig im Datenbestand der Datenbank hin- und herzuspringen. Datenbestand und Logdateien werden dann ressourcenschonend synchronisiert.

Da alle Transaktionen im Log sequentiell fortgeschrieben werden, würden die Protokolle unaufhörlich wachsen und irgendwann das Dateisystem überlaufen lassen. Daher werden die Logs einfach rotiert, d. h. man definiert[2] einfach wieviele Dateien es geben soll, und wenn diese Zahl erreicht ist wird wieder in die erste Datei geschrieben. Um die Logdateien zu sichern, muss man daher die Logs vor der Rotation auf einen anderen Datenträger kopieren. Dies geschieht, indem man in der postgresql.conf die Option `archive_command` wie in Abbildung 2 beschrieben setzt, so das die Logdateien vor dem Überschreiben in ein anderes Verzeichnis kopiert werden. Dabei sollte darauf geachtet werden, dass der Befehl auch ausgeführt wird, da ihn PostgreSQL sonst bis zum Gelingen wiederholt. So kann es unter Umständen vorkommen, dass das Dateisystem mit WAL-Dateien volläuft, da die Rotation der Logs nicht durchgeführt werden kann. Weiterhin ist es empfehlenswert, zumindest bei ausgelasteten Servern, die Größe und Anzahl der Logdateien zu benchmarken. Je mehr Transaktionen auftreten, die geloggt werden müssen, desto mehr werden die Logdateien rotiert, was wiederum zu Leistungseinbußen führen kann.

Um eine Sicherung des Datenbankclusters durchzuführen, muss man die Datenbank vor- und nachbereiten. Dazwischen kann der Datenbankcluster gesichert werden. Dies geschieht mit den SQL-Befehlen in Abbildung 3. Dort wird der Datenbankcluster zur Sicherung vorbereitet und gesichert, anschließend wird der Backup-Checkpoint entsperrt und es können die neuen WALs rotiert werden. Hat man einen derart gesicherten Datenbankcluster, reicht es aus, die WALs, die danach erzeugt werden, zu sichern. Ist die Datenbankaktivität – und damit das Wachstum der Logs – recht hoch, empfiehlt es sich u. U. häufiger den Cluster zu sichern und somit die Zahl der zu archivierenden Logdateien zu minimieren.

Um einen derartig gesicherten Datenbestand zurückzuspielen, sind folgende Schritte notwendig:

1. PostgreSQL neu installieren oder anhalten
2. Sicherung des Datenbankclusters zurückspielen
3. recovery.conf erstellen und
4. `restore_command` und `recovery_target_time` auf gewünschten Zeitpunkt setzen
5. PostgreSQL starten

Punkt 1 wird erledigt, indem man den Dump aus Abbildung 3 mit `restore -r -f /usr/backuppgcluster.0` zurückspielt und ggf. die Dateirechte anpasst.

Punkt 2 und 3 setzen die entsprechenden Befehle, die der Postmaster ausführen soll. `restore_command` ist der Befehl, um die WALs zurückzukopieren, also etwas derart: `restore_command=cp /usr/backup/postgres-wal/%f %p`, `recovery_target_time` erwartet einen Zeitstempel, bis zu dem rückgesichert werden soll.

pg_dump, pg_dumpall und pg_restore

PostgreSQL verfügt auch über ein Werkzeug, um logische Backups zu erzeugen. Dabei werden SQL-Befehle zum Erstellen der Datenbank(en) in eine einfache ASCII-Textdatei oder ein Archiv geschrieben. Die entstandene ASCII-Datei kann danach mit Sicherungsmechanismen für Dateisysteme gesichert werden. Da PostgreSQL zur Transaktionsverwaltung die »Multi Version Concurrency Control« (MVCC) einsetzt, können pg_dump und pg_dumpall problemlos im laufenden Betrieb eingesetzt werden. Es wird immer der aktuell gültige konsistente Datenbankbestand gesichert.

Um eine Datenbank zu sichern, benutzt man pg_dump(1). Um alle Datenbanken in eine Datei zu sichern existiert pg_dumpall(1).

Der erste Befehl in Abbildung 4 bereinigt die Datenbank »meineDB« als PostgreSQL-Benutzer »pgoperator« mit vacuumdb(1). Dabei werden die Datensätze reorganisiert und leere Zeilen entfernt, um den Platzbedarf zu verringern. Der zweite Befehl schreibt die Datenbank mit komprimierten »Large Objects« in eine Sicherungsdatei. Der letzte Befehl verwendet pg_dumpall(1) um alle Datenbanken in eine Datei zu sichern.

Zum Rückspielen einer mit pg_dump(1) erzeugten Sicherung verwendet man pg_restore(1), wie in Abb. 5 gezeigt. Dazu muss die rückzusichernde Datenbank vorher allerdings mit createdb(1) erstellt worden sein. Anders verhält es sich mit pg_dumpall(1), denn dessen Sicherung enthält alle Befehle um die gesicherten Datenbanken neu anzulegen, so dass man sich mit einer beliebigen Datenbank verbinden kann.

[2] Standardmäßig existieren in pg_xlog/ drei WAL Dateien mit je 16MB Größe. Diese Größe kann beim Kompilieren des Servers angegeben werden, die Anzahl in postgresql.conf.

```
archive_command = 'cp %p /usr/backup/postgres-wal/%f'
```

Abbildung 2: postgresql.conf-Konfig zur Sicherung der Logs

```
$ echo "select pg_start_backup('Label');" | psql -U pgsql template1
# fssconfig -x -c fss0 /pgcluster /
# dump -0 -f /usr/backuppgcluster.0 /dev/fss0
# fssconfig -u fss0
$ echo "select pg_stop_backup();" | psql -U pgsql template1
```

Abbildung 3: PostgreSQL-Datenbankcluster und WALs sichern

```
$ /usr/pkg/bin/vacuumdb -Upgoperator -f -z meineDB

$ /usr/pkg/bin/pg_dump -Fc -Upgoperator -meineDB >
               /home/pgsql/meineDB_‘date +%y%m%d‘

$ /usr/pkg/bin/pg_dumpall > pg_‘date +%y%m%d‘
```

Abbildung 4: PostgreSQL-Datenbanken sichern

```
$ createdb meineDB
$ pg_restore -d meineDB -f meineDB_051206

$ psql -f pg_051206 template1
```

Abbildung 5: Rückspielen von PostgreSQL-Sicherungen

Replikation

Replikationssysteme synchronisieren den Datenbestand von mehreren Servern. Dies kann auf verschiedene Arten geschehen, beispielsweise synchron/asynchron oder voll/teilweise.

Somit ist es möglich, ein Ersatzsystem[3] bereitzuhalten, das per Replikation auf dem aktuellen Stand des Originals gehalten wird und bei Ausfall des Originals dessen Funktion übernehmen kann.

Asynchrone Replikation verteilt die Daten erst nach einem erfolgreichen ≫`BEGIN ...COMMIT`≪ -Block auf die angeschlossenen Replikanten. Dies schränkt das System aber etwas ein:

- nur ein einziger Master ist sinnvoll, da sonst Konsistenzprobleme drohen (OIDs, Primärschlüssel)
- Lastverteilung ist nicht vorgebbar

Synchrone Replikation hingegen verteilt die Daten sofort bei Beginn der Transaktion, so dass diese auf allen Rechnern ausgeführt wird. Hierbei ist allerdings die Konsistenz der Rechner untereinander ein Problem, denn es muss auf allen Maschinen ein erfolgreicher `COMMIT` garantiert werden. Die Slaves informieren nach einem erfolgten Schreibzugriff den Master von der Transaktion, so das weitere Schreibtransaktionen ausgeführt werden können. Dieses Verfahren wird Zwei-Phasen-Commit genannt, da der Master jedesmal auf die Slaves warten muss, bevor eine Transaktion endgültig für den gesamten Rechnerverbund als commited markiert wird. So wird zwar eine sofortige Replikation des Datenbestandes erreicht, allerdings auf Kosten des Durchsatzes, da diese Transaktion eben auf jedem Rechner erfolgen muss und der Erfolg an den Master zurückgemeldet werden muss.

Synchrone Replikation mit Pgpool

Pgpool fungiert als sogenannter ≫*Connection Pool*≪, d.h. er klinkt sich zwischen den eigentlichen PostgreSQL-Server und die Anwendungen. Dies funktioniert im Prinzip wie bei einem transparenten Proxy. Pgpool cached Verbindungen zum Datenbankserver, um so Lasten durch Verbindungsauf- und -abbau zu reduzieren. Weiterhin kann sich Pgpool mit zwei PostgreSQL-Servern verbinden und so im Falle eines Ausfalls auf den anderen, noch funktionierenden Server umschalten.

Zwischen den beiden PostgreSQL-Servern kann Pgpool ausserdem als synchroner Replikationsserver fungieren, d.h. alle Datenbankenanfragen werden an beide Postmaster geschickt. Dies ermöglicht eine einfache Replikation des Datenbestandes auf zwei verschiedene Rechner, die lediglich durch das Netzwerk miteinander verbunden sein müssen. Diese Methode schließt allerdings einige Operationen aus, die bspw. abhängig vom Server (Abfrage der OID, eines Zeitstempels oder ähnliches) oder eben nicht deterministisch (z.B. Zufallsgenerator) sind.

Pgpool lässt sich aus pkgsrc/databases/pgpool installieren. Zur Konfiguration genügt es `/usr/pkg/share/examples/pgpool.conf.sample` nach `/usr/pkg/etc/pgpool.conf` zu kopieren und anzupassen. Die Konfigurationsdatei ist wohldokumentiert und recht einfach zu verstehen. Wichtig sind folgende Optionen:

- **listen_addresses** und **port**: zu benutzende Netzwerkadresse und Port

[3] In Fachkreisen auch *Hot-Backup* genannt.

- **backend_host_name** und **backend_port**: Netzadresse und Port des PostgreSQL-Servers
- **secondary_backend_host_name** und **secondary_backend_port**: Netzadresse und Port des zweiten PostgreSQL-Servers (Slave)
- **replication_mode**: Einsatz als Replikationssystem
- **replication_strict**: Wenn aktiviert, werden Deadlocks vermieden, was auf Kosten des Durchsatzes geht.
- **replication_timeout**: Wenn replication_strict deaktiviert ist könnten Deadlocks auftreten. Dieser Wert gibt den Timeout in μs an, nachdem die blockierte Transaktion abgebrochen wird.
- **replication_stop_on_mismatch**: Der Replikationsmodus soll bei inkonsistenten Daten zwischen Master und Slave abgebrochen werden.

Hat man Pgpool wie in Abbildung (6) konfiguriert, kann man sich an localhost:9999 mit dem Datenbankterminal verbinden. Alle DML-Operationen werden dann auf beiden PostgreSQL-Servern ausgeführt.

Mit `pgpool switch` kann man die Server umschalten, bspw. mit `pgpool -s master switch` die Verbindung zum Master beenden und auf den Secondary Server umschalten.

Fazit

Es existieren verschiedene Möglichkeiten und Strategien um PostgreSQL-Datenbestände zu sichern. Welche Variante am besten funktioniert, hängt vom Profil der Datenbank und den Daten ab. Wenn man einfache Datenstrukturen hat und gewisse Datenverluste bei einem Systemausfall verkraften kann, reicht eine regelmäßige Sicherung der Datenbankinhalte mit pg_dump/pg_dumpall bspw. via Cron aus.

Sind die Daten wichtiger und nur sehr geringe Verluste vertretbar, oder es besteht der Wunsch, Daten in einen vorigen Zustand versetzen zu können, bieten sich Point-in-Time-Recovery-Methoden an. Diese belasten allerdings unter Umständen die Hardware sehr stark und können Leistungsgrenzen ausreizen. Trotzdem kann man mit PiTR-Methoden nahezu alle Zustände der Datenbank sichern und wiederherstellen.

Nachteil der beiden erstgenannten Methoden ist fehlende ≫Hot-Standby-Fähigkeit≪ , d.h. ein Ersatzsystem muss erst mühsam mit den gesicherten Daten versorgt werden. Dies kann vor allem bei PiTR äußerst zeitaufwändig werden. Einzige Möglichkeit eines Hot-Standby-Systems ist die Replikation der Datenbestände auf ein oder mehrere angeschlossene Systeme, da dabei nur maximal die aktuelle offene Transaktion verloren gehen kann. Somit ist es möglich, beliebig viele Ersatzsysteme Gewehr bei Fuß zu halten und im Bedarfsfall sofort einzusetzen. Trotzdem sollte man auch hier regelmäßig den Haupt-Datenbestand extern sichern, da Replikationen auch fehlschlagen können und die Datenbank-Cluster u. U. nicht mehr synchron sind. Dies muss dann vom Administrator per Hand korrigiert werden und lässt sich am einfachsten mit dem Einspielen einer Sicherung erledigen.



Stefan Schumacher beschäftigt sich in seiner Freizeit mit japanischen Kampfkünsten sowie mit NetBSD und PostgreSQL. Als studentische Hilfskraft arbeitet er als Datenbankentwickler im Forschungsprojekt ≫Russisch-Deutsches Wörterbuch≪ an der Otto-von-Guericke-Universität Magdeburg. Seine persönlichen Webseiten sind unter www.net-tex.de bzw. www.cryptomancer.de erreichbar, seine PGP-Schlüssel-ID ist 0xB3FBAE33.