CloudNativePG 1.26 - PostgreSQL In-Place Major Upgrades

CloudNativePG 1.26 bringt eines der am meisten ersehnten Features: **deklarative In-Place-Major-Upgrades** für PostgreSQL unter Verwendung von `pg_upgrade`. Dieser neue Ansatz ermöglicht es, PostgreSQL-Cluster durch das Ändern des `imageName` in ihrer Konfiguration zu aktualisieren—genauso wie bei einem Minor-Version-Update. Obwohl eine kurze Downtime erforderlich ist, wird der betriebliche Aufwand erheblich gesenkt, was es zur idealen Lösung für die Verwaltung `großer Flotten von PostgreSQL-Datenbanken` in Kubernetes macht. In diesem Artikel werde ich die Funktionsweise untersuchen, die Vorteile und Einschränkungen darstellen und das Upgrade einer 2,2-TB-Datenbank erläutern.

CloudNativePG 1.26, welches Ende März released wurde, bringt eines der am meisten herbeigesehnten Features in der Geschichte des Projekts: In-Place-Upgrades der Hauptversion von PostgreSQL mit `pg_upgrade`.

Im Unterschied zu kleineren Upgrades, die hauptsächlich Patches einspielen, erfordern Hauptversion-Upgrades das Anpassen an Änderungen im internen Speicherformat, die durch die neue PostgreSQL-Version eingeführt wurden.

Dieses Feature steht nun in der Version 1.26.0 zur Erprobung bereit.

Eine Übersicht über die bestehenden Methoden

CloudNativePG bietet nun drei deklarative (ja, deklarative!) Upgrade-Methoden für Hauptversionen an. Zwei dieser Methoden erfordern die Einrichtung eines neuen Clusters und sind als **Blue/Green-Deployment-Strategien** bekannt.

Der erste Ansatz verwendet die `Importfunktion` von `pg_dump` und `pg_restore`. Dies ist für kleine Datenbanken praktisch und eignet sich gut zum Testen neuer Versionen, erfordert jedoch für den endgültigen Wechsel eine Ausfallzeit, weshalb es sich um ein Offline-Upgrade handelt.

Die zweite Methode nutzt die native logische Replikation von PostgreSQL, um Upgrades ohne Ausfallzeiten zu ermöglichen—also ein Online-Upgrade—unabhängig von der Größe der Datenbank. Diese Methode ist meine bevorzugte Vorgehensweise für Upgrades geschäftskritischer PostgreSQL-

Datenbanken. Sie kann auch für die Migration aus externen Umgebungen nach Kubernetes eingesetzt werden (z.B. von Amazon RDS zu CloudNativePG).

Die dritte Methode, die im Fokus dieses Artikels steht, sind Offline-In-Place-Upgrades unter Verwendung von `pg_upgrade`, dem offiziellen Werkzeug von PostgreSQL für solche Aufgaben.

Der Anwendungsfall für In-Place-Hauptversion-Upgrades

Der Hauptgrund für die Implementierung dieser Funktion in Kubernetes besteht darin, den betrieblichen Unterschied zwischen kleinen und großen PostgreSQL-Upgrades für GitOps-Nutzer zu eliminieren. Mit diesem Ansatz erfordert das Upgrade lediglich das Anpassen der Clusterkonfiguration und das Aktualisieren des Images für alle Clusterkomponenten (primäre und Standby-Server). Das ist besonders auf großer Ebene nützlich—wenn du dutzende oder sogar hunderte PostgreSQL-Cluster in einem einzigen Kubernetes-Cluster verwaltest—wo Blue/Green-Upgrades betriebliche Herausforderungen mit sich bringen können.

Bevor du startest

In-Place-Hauptversion-Upgrades stehen in CloudNativePG 1.26.0 aktuell zur Anwendung bereit. Diese Funktion kann in jedem Kubernetes-Cluster getestet werden.

Um CloudNativePG 1.26.0 zu installieren:

```
kubectl apply --server-side -f
https://raw.githubusercontent.com/cloudnative-pg/cloudnative-
pg/main/releases/cnpg-1.26.0.yaml
```

Wie es funktioniert

CloudNativePG erlaubt dir, das PostgreSQL-Operand-Image auf zwei Arten anzugeben:

- mit der Option `.spec.imageName`
- mit Image-Katalogen (`ImageCatalog`- und `ClusterImageCatalog`-Ressourcen)

Dieser Artikel konzentriert sich auf die `imageName`-Methode, obwohl dieselben Prinzipien auch bei der Image-Katalog-Methode gelten.

Nehmen wir an, du hast ein PostgreSQL-Cluster, mit:

```
imageName: ghcr.io/cloudnative-pg/postgresql:13.20-minimal-bullseye
```

Dies bedeutet, dass dein Cluster das neueste verfügbare Container-Image für PostgreSQL 13 (Nebenversion 20) verwendet. Da PostgreSQL 13 im November dieses Jahres das Lebensende erreicht, entscheidest du dich für ein Upgrade auf PostgreSQL 17 mit dem Image `ghcr.io/cloudnative-pg/postgresql:17.4-minimal-bullseye`.

Durch die Aktualisierung des `imageName`-Feldes in der Cluster-Konfiguration initiiert CloudNativePG automatisch ein Upgrade auf eine neue Hauptversion.

Der Upgrade-Prozess

Der erste Schritt besteht darin, das PostgreSQL-Cluster sicher herunterzufahren, um vor dem Upgrade die Datenkonsistenz zu gewährleisten. Dies ist ein Offline-Vorgang, der einen Ausfall bedeutet, aber es ermöglicht, statische Datendateien mit voller Integrität zu modifizieren.

CloudNativePG aktualisiert dann den Status der Cluster-Ressource (kind: cluster), um das derzeit laufende Image vor dem Beginn des Upgrades zu protokollieren. Dies ist wesentlich für einen Rollback im Falle eines Fehlers (wird später im Artikel noch intensiver behandelt).

Danach startet CloudNativePG einen Kubernetes-Job, der für die Vorbereitung der PostgreSQL-Datendateien auf den Persistent Volume Claims (PVC) für die neue Hauptversion mit `pg_upgrade` zuständig ist:

- Der Job erstellt eine temporäre Kopie der alten PostgreSQL-Binärdateien.
- Er initialisiert ein neues PGDATA-Verzeichnis mit initdb für die Zielversion von PostgreSQL.
- Er überprüft das Upgrade-Erfordernis, indem er die auf der Festplatte gespeicherten PostgreSQL-Versionen vergleicht, um ungewollte Upgrades basierend auf Image-Tags zu verhindern.
- Er ordnet die WAL- und Tablespace-Volumes bei Bedarf automatisch neu zu.

An diesem Punkt wird der tatsächliche Upgrade-Prozess mit `pg_upgrade` und der `–link`-Option ausgeführt, um Hardlinks zu nutzen, was die Datenmigration erheblich beschleunigt und den Speicherplatzbedarf sowie die Festplatten-E/A minimiert.

Wenn das Upgrade erfolgreich abgeschlossen wird, ersetzt CloudNativePG die ursprünglichen PostgreSQL-Datenverzeichnisse durch die aktualisierten Versionen, zerstört die Persistent Volume Claims der Replikate und startet das Cluster neu.

Tritt jedoch ein Fehler bei pg_upgrade auf, musst du manuell zur vorherigen Hauptversion von PostgreSQL zurückkehren, indem du die Cluster-Spezifikation aktualisierst und den Upgrade-Job löschst. Wie bei jedem In-place-Upgrade besteht immer das Risiko eines Scheiterns. Um dies zu minimieren, ist es entscheidend, kontinuierliche Basissicherungen zu pflegen. Falls deine StorageClass Volumes-Snapshots unterstützt, solltest du erwägen einen solchen vor dem Upgrade zu erstellen, es ist eine einfache Vorsichtsmaßnahme die dich vor unerwarteten Problemen bewahren könnte.

Insgesamt verbessert dieser effiziente Ansatz die Effizienz und Zuverlässigkeit von In-place-Upgrades auf eine neue Hauptversion und macht PostgreSQL-Versionsübergänge in Kubernetes-Umgebungen handhabbarer.

Beispiel

Der beste Weg, dieses Feature zu verstehen, ist, es in der Praxis zu testen. Beginnen wir mit einem einfachen PostgreSQL-13-Cluster namens pg, definiert in der folgenden `pg.yaml`:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: pg
spec:
  imageName: ghcr.io/cloudnative-pg/postgresql:13.20-minimal-bullseye
  instances: 3

  storage:
    size: 1Gi
  walStorage:
    size: 1Gi
```

Nach dem Erstellen des Clusters überprüfe seinen Status mit:

```
kubectl cnpg status pg
```

Du kannst die Version auch mit `psql` kontrollieren:

```
kubectl cnpg psql pg -- -qAt -c 'SELECT version()'
```

Es sollte eine ähnliche Ausgabe erscheinen:

```
PostgreSQL 13.20 (Debian 13.20-1.pgdg110+1) on x86_64-pc-linux-gnu, compiled
by gcc (Debian 10.2.1-6) 10.2.1 20210110, 64-bit
```

Lassen wir uns nun von PostgreSQL 13, das bald sein Lebensende erreicht, auf die neueste Minor-Version der aktuellsten Hauptversion upgraden. Dafür brauchst du lediglich das Feld `imageName` in deiner Konfiguration zu aktualisieren:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: pg
spec:
  imageName: ghcr.io/cloudnative-pg/postgresql:17.4-minimal-bullseye
  instances: 3

  storage:
    size: 1Gi
  walStorage:
    size: 1Gi
```

Wende die Änderungen an, um den Major-Upgrade-Prozess zu starten:

```
kubectl apply -f pg.yaml
```

Sobald der Prozess abgeschlossen ist, überprüfe das Upgrade, indem du den Clusterstatus erneut kontrollierst. Deine Datenbank sollte nun unter PostgreSQL 17 laufen.

Wenn du die Version erneut prüfst, sollte eine ähnliche Ausgabe erscheinen:

```
PostgreSQL 17.4 (Debian 17.4-1.pgdg110+2) on x86_64-pc-linux-gnu, compiled
by gcc (Debian 10.2.1-6) 10.2.1 20210110, 64-bit
```

Wenn du jetzt `kubectl get pods` eingibst, wirst du feststellen, dass die Pods und PVCs mit den Namen `pg-2` und `pg-3` nicht mehr vorhanden sind, da sie durch Sequenznummern 4 und 5 ersetzt wurden.

```
NAME   READY   STATUS    RESTARTS   AGE
pg-1   1/1     Running   0          62s
pg-4   1/1     Running   0          36s
pg-5   1/1     Running   0          15s
```

Einschränkungen und Vorbehalte

Wie du gerade gelernt hast, erfordert diese Implementierung, dass `Replikate` (replicas) neu erstellt werden, was derzeit nur mit `pg_basebackup` unterstützt wird, dies beeinträchtigt jedoch den Datenbankzugriff nicht. Wenn der primäre Knoten ausfällt, musst du bis zur Verfügbarkeit eines neuen Replikats aus dem neuesten Backup wiederherstellen. In den meisten Fällen stammt dieses Backup von der vorherigen PostgreSQL-Version, was bedeutet, dass du den Major-Upgrade-Prozess wiederholen musst.

Obwohl dieses Szenario unwahrscheinlich ist, ist es wichtig, das Risiko zu erkennen. In der Regel wird die Replikation innerhalb von Minuten abgeschlossen, je nach Komplexität der Datenbank (insbesondere der Anzahl der Tabellen).

Bei deutlich größeren Datenbanken sei darauf hingewiesen, dass der Cluster bis zur vollständigen Wiederherstellung der Replikation in einem eingeschränkten Zustand für hohe Verfügbarkeit verbleibt. Um das Risiko zu minimieren, empfehle ich dringend, so bald wie möglich nach Abschluss des Upgrades ein physisches Backup zu erstellen.

Ein weiterer wesentlicher Aspekt sind die `Erweiterungen` (extensions). Sie spielen eine entscheidende Rolle beim Upgrade-Prozess. Achte darauf, dass alle erforderlichen Erweiterunge, und deren jeweilige Versionen, im Operand-Image der Ziel PostgreSQL Version verfügbar sind. Sind Erweiterungen nicht vorhanden, wird das Upgrade fehlschlagen. Überprüfe deshalb immer die Kompatibilität der Erweiterungen im Vorfeld.

Test eines großen Datenbank-Upgrades

Um zu evaluieren, wie ein Major-Upgrade von PostgreSQL mit einer großen Datenbank umgeht, habe ich ins Rahmen meiner Tests eine `2,2` TB große PostgreSQL 16-Datenbank mit `pgbench` und einem Skalierungsfaktor von `150.000` erstellt. Hier ein Auszug aus dem `cnpg status` Befehl:

```
Cluster Summary
Name                 default/pg
System ID:           7487705689911701534
```

```
PostgreSQL Image:    ghcr.io/cloudnative-pg/postgresql:16
Primary instance:    pg-1
Primary start time:  2025-03-30 20:42:26 +0000 UTC (uptime 72h32m31s)
Status:              Cluster in healthy state
Instances:           1
Ready instances:     1
Size:                2.2T
Current Write LSN:   1D0/8000000 (Timeline: 1 - WAL File:
00000001000001D000000001)
<snip>
```

Ich habe dann ein Upgrade auf **PostgreSQL 17** angestoßen, welches in lediglich **33 Sekunden** abgeschlossen war, wobei der Cluster in weniger als einer Minute komplett einsatzbereit war. Hier die aktualisierte `cnpg status` Ausgabe:

```
Cluster Summary
Name                 default/pg
System ID:           7488830276033003555
PostgreSQL Image:    ghcr.io/cloudnative-pg/postgresql:17
Primary instance:    pg-1
Primary start time:  2025-03-30 20:42:26 +0000 UTC (uptime 72h44m45s)
Status:              Cluster in healthy state
Instances:           1
Ready instances:     1
Size:                2.2T
Current Write LSN:   1D0/F404F9E0 (Timeline: 1 - WAL File:
00000001000001D00000003D)
```

Da CloudNativePG die `–link` Option von PostgreSQL verwendet (die auf Hardlinks basiert), **hängt die Upgrade-Dauer vor allem von der Anzahl der Tabellen und nicht von der Größe der Datenbank ab**.

Schlussfolgerungen

In-place Major-Upgrades mit `pg_upgrade` bringen den herkömmlichen Upgrade-Pfad von PostgreSQL in Kubernetes, wodurch Nutzern eine deklarative Methode geboten wird, um mit minimalem Betriebsaufwand zwischen Hauptversionen zu wechseln. Diese Methode beinhaltet zwar Ausfallzeiten, eliminiert aber die Notwendigkeit von Blue/Green-Clustern und eignet sich somit besonders für Umgebungen, **die eine große Vielzahl kleiner bis mittelgroßer PostgreSQL-Instanzen verwalten**.

Bei einem erfolgreichen Upgrade erhältst du einen voll funktionsfähigen PostgreSQL-Cluster, wie wenn du `pg_upgrade` auf einer traditionellen VM oder einem physischen Server ausgeführt hättest. Bei einem Fehlschlag stehen Rollback-Optionen zur Verfügung—inklusive der Rückkehr zum ursprünglichen Manifest und Löschen des Upgrade-Jobs. Kontinuierliche Backups bieten ein zusätzliches Sicherheitsnetz.

Auch wenn In-place-Upgrades möglicherweise nicht meine bevorzugte Methode für geschäftskritische

Datenbanken sind, stellen sie eine bedeutende Option für Teams dar, **die Betriebseinfachheit und Skalierbarkeit** über Null-Ausfallzeit-Upgrades priorisieren. Wie in den Tests aufgezeigt, wird die Upgrade-Dauer in erster Linie durch die Anzahl der Tabellen und nicht durch das Datenbankvolumen bestimmt, was diesen Ansatz auch für große Datensätze effizient macht.