

CloudNativePG

リリース *1.19.1*

The CloudNativePG Contributors.

2023 年 03 月 25 日

目 次

第 1 章　サポートされている Kubernetes ディストリビューション **3**

第 2 章　コンテナ イメージ **5**

第 3 章　主な特徴 **7**

第 4 章　このガイドについて **9**

4.1　始める前に . . . 9
4.1.1　Kubernetes の用語 . . . 9
4.1.2　PostgreSQL 用語 . . . 11
4.1.3　クラウド用語 . . . 11
4.1.4　次はどうする . . . 12
4.2　ユースケース . . . 12
4.2.1　ケース 1: Kubernetes 内のアプリケーション . . . 12
4.2.2　ケース 2: Kubernetes 外のアプリケーション . . . 15
4.3　建築 . . . 15
4.3.1　状態の同期 . . . 16
4.3.2　Kubernetes アーキテクチャ . . . 16
4.3.3　PostgreSQL のアーキテクチャ . . . 19
4.3.4　Kubernetes クラスター全体へのデプロイ . . . 21
4.4　インストールとアップグレード . . . 23
4.4.1　Kubernetes へのインストール . . . 23
4.4.2　導入の詳細 . . . 24
4.4.3　アップグレード . . . 25
4.5　クイックスタート . . . 27
4.5.1　パート 1 - ローカルの Kubernetes プレイグラウンドをセットアップする . . . 27
4.5.2　パート 2 - CloudNativePG をインストールする . . . 28
4.5.3　パート 3 - PostgreSQL クラスターをデプロイする . . . 29
4.5.4　パート 4 - Prometheus と Grafana を使用してクラスターを監視する . . . 30
4.6　ブートストラップ . . . 34
4.6.1　`bootstrap` セクション . . . 35
4.6.2　`externalClusters` セクション . . . 35
4.6.3　空のクラスターをブートストラップする (`initdb`) . . . 36
4.6.4　別のクラスターからのブートストラップ . . . 40
4.7　Postgres データベースのインポート . . . 53
4.7.1　使い方 . . . 54
4.7.2　`microservice` タイプ . . . 54
4.7.3　`monolith` タイプ . . . 56
4.8　安全 . . . 59
4.8.1　コード . . . 59

4.8.2 容器 . . . 59
4.8.3 集まる . . . 60
4.9 Postgre インスタンス マネージャー . . . 65
4.9.1 liveness と readiness プローブ . . . 65
4.9.2 シャットダウン制御 . . . 66
4.9.3 フェイルオーバー . . . 67
4.10 スケジューリング . . . 67
4.10.1 Pod の親和性と非親和性 . . . 67
4.10.2 nodeSelector によるノード選択 . . . 69
4.10.3 許容範囲 . . . 69
4.11 資源管理 . . . 69
4.12 失敗モード . . . 71
4.12.1 ストレージ容量の使用 . . . 72
4.12.2 故障モード . . . 72
4.12.3 自己修復 . . . 74
4.12.4 手動介入 . . . 74
4.13 ローリング更新 . . . 75
4.13.1 自動更新 (unsupervised) . . . 75
4.13.2 手動更新 (supervised) . . . 76
4.14 レプリケーション . . . 76
4.14.1 アプリケーションレベルのレプリケーション . . . 77
4.14.2 非常に成熟したテクノロジー . . . 77
4.14.3 ストリーミング レプリケーションのサポート . . . 77
4.14.4 同期レプリケーション . . . 78
4.14.5 高可用性のためのレプリケーション スロット . . . 80
4.15 バックアップとリカバリ . . . 82
4.15.1 クラウド プロバイダーのサポート . . . 82
4.15.2 オンデマンド バックアップ . . . 91
4.15.3 スケジュールされたバックアップ . . . 93
4.15.4 WAL アーカイブ . . . 94
4.15.5 スタンバイからのバックアップ . . . 95
4.15.6 回復 . . . 96
4.15.7 保持ポリシー . . . 98
4.15.8 圧縮アルゴリズム . . . 98
4.15.9 バックアップ オブジェクトのタグ付け . . . 99
4.16 PostgreSQL の設定 . . . 99
4.16.1 postgresql セクション . . . 100
4.16.2 pg_hba セクション . . . 105
4.16.3 構成の変更 . . . 107
4.16.4 動的共有メモリの設定 . . . 107
4.16.5 固定パラメータ . . . 108
4.17 オペレーター構成 . . . 110
4.17.1 利用可能なオプション . . . 111
4.17.2 オペレーター構成マップの定義 . . . 112
4.17.3 オペレーター シークレットの定義 . . . 112
4.17.4 オペレータを再起動して設定をリロードします . . . 112
4.17.5 PPROF HTTP サーバー . . . 113

4.18 インスタンス Pod 構成 . . . 114
4.18.1 予測ボリューム . . . 114
4.18.2 環境変数 . . . 115
4.19 保管所 . . . 116
4.19.1 CloudNativePG のベンチマーク . . . 117
4.19.2 永続ボリュームの要求 . . . 117
4.19.3 ストレージ クラスによる構成 . . . 118
4.19.4 PVC テンプレートによる設定 . . . 118
4.19.5 WAL のボリューム . . . 119
4.19.6 ボリューム拡張 . . . 120
4.20 ラベルと注釈 . . . 123
4.20.1 前提条件 . . . 124
4.20.2 クラスターのメタデータの定義 . . . 124
4.20.3 現在の制限事項 . . . 125
4.21 モニタリング . . . 125
4.21.1 インスタンスのモニタリング . . . 125
4.21.2 オペレーターの監視 . . . 139
4.21.3 エクスポートされたメトリクスを検査する方法 . . . 140
4.21.4 補助リソース . . . 141
4.22 ロギング . . . 142
4.22.1 オペレーターログ . . . 142
4.22.2 PostgreSQL ログ . . . 142
4.22.3 PGAudit ログ . . . 143
4.22.4 その他のログ . . . 145
4.23 証明書 . . . 146
4.23.1 オペレーター管理モード . . . 146
4.23.2 ユーザー提供の証明書モード . . . 148
4.24 クライアント TLS/SSL 接続 . . . 152
4.24.1 新しい証明書の発行 . . . 153
4.24.2 TLS 証明書による接続のテスト . . . 153
4.25 アプリケーションからの接続 . . . 156
4.25.1 DNS 解決 . . . 156
4.25.2 環境変数 . . . 157
4.25.3 秘密 . . . 157
4.26 接続プーリング . . . 157
4.26.1 建築 . . . 158
4.26.2 クイックスタート . . . 158
4.26.3 Pooler リソースのライフサイクル . . . 159
4.26.4 安全 . . . 160
4.26.5 認証 . . . 160
4.26.6 ポッドテンプレート . . . 161
4.26.7 高可用性 (HA) . . . 163
4.26.8 PgBouncer 設定オプション . . . 163
4.26.9 モニタリング . . . 165
4.26.10 ロギング . . . 172
4.26.11 接続の一時停止 . . . 172
4.26.12 制限事項 . . . 173

4.27 レプリカ クラスタ . . . 173
4.27.1 基本概念 . . . 173
4.27.2 レプリカ クラスタのセットアップ . . . 175
4.27.3 レプリカ クラスタ内の指定されたプライマリの昇格 . . . 177
4.28 Kubernetes のアップグレード . . . 177
4.28.1 reusePVC が false に設定された単一インスタンス クラスタ . . . 179
4.29 Postgres サービスの公開 . . . 179
4.29.1 Minikube でのテスト . . . 181
4.30 CloudNativePG プラグイン . . . 182
4.30.1 インストール . . . 182
4.30.2 使用 . . . 185
4.31 自動フェイルオーバー . . . 199
4.31.1 RTO と RPO への影響 . . . 199
4.31.2 フェイルオーバーの遅延 . . . 200
4.32 トラブルシューティング . . . 200
4.32.1 始める前に . . . 201
4.32.2 緊急バックアップ . . . 201
4.32.3 オペレーター情報 . . . 203
4.32.4 クラスタ情報 . . . 205
4.32.5 ポッド情報 . . . 206
4.32.6 バックアップ情報 . . . 208
4.32.7 保管情報 . . . 208
4.32.8 ノード情報 . . . 208
4.32.9 条件 . . . 209
4.32.10 よくある問題 . . . 210
4.33 フェンシング . . . 211
4.33.1 インスタンスをフェンシングする方法 . . . 212
4.33.2 フェンシングの持ち上げ方 . . . 213
4.33.3 フェンシングの仕組み . . . 213
4.34 PostGIS . . . 214
4.34.1 PostGIS クラスターに関する基本概念 . . . 214
4.34.2 PostGIS で新しい PostgreSQL クラスターを作成する . . . 215
4.35 エンドツーエンドのテスト . . . 217
4.36 コンテナ イメージの要件 . . . 218
4.36.1 画像タグの要件 . . . 219
4.37 オペレーターの能力レベル . . . 219
4.37.1 レベル 1 - 基本インストール . . . 220
4.37.2 レベル 2 - シームレスなアップグレード . . . 225
4.37.3 レベル 3 - フル ライフサイクル . . . 226
4.37.4 レベル 4 - 深い洞察 . . . 230
4.37.5 レベル 5 - 自動操縦 . . . 231
4.38 カスタム Pod コントローラー . . . 232
4.38.1 PVC のサイズ変更 . . . 232
4.38.2 プライマリ インスタンスとレプリカ . . . 232
4.38.3 PVC の一貫性 . . . 233
4.38.4 ローカル ストレージ、リモート ストレージ、およびデータベース サイズ . . . 233
4.39 例 . . . 234

4.40 ベンチマーク . . . 235
4.40.1 pgbench . . . 236
4.40.2 フィオ . . . 237
4.41 商用サポート . . . 239
4.42 よくある質問 (FAQ) . . . 239
4.42.1 Kubernetes で PostgreSQL を実行する . . . 239
4.42.2 高可用性 . . . 241
4.43 API リファレンス . . . 242
4.43.1 アフィニティ構成 . . . 244
4.43.2 AzureCredentials . . . 245
4.43.3 バックアップ . . . 246
4.43.4 バックアップ構成 . . . 246
4.43.5 バックアップリスト . . . 247
4.43.6 バックアップソース . . . 247
4.43.7 バックアップ仕様 . . . 248
4.43.8 バックアップステータス . . . 248
4.43.9 BarmanCredentials . . . 249
4.43.10 BarmanObjectStoreConfiguration . . . 249
4.43.11 ブートストラップ構成 . . . 250
4.43.12 BootstrapInitDB . . . 251
4.43.13 BootstrapPgBaseBackup . . . 252
4.43.14 BootstrapRecovery . . . 253
4.43.15 証明書の構成 . . . 253
4.43.16 証明書ステータス . . . 255
4.43.17 集まる . . . 255
4.43.18 ClusterList . . . 255
4.43.19 ClusterSpec . . . 256
4.43.20 ClusterStatus . . . 258
4.43.21 ConfigMapKeySelector . . . 260
4.43.22 ConfigMapResourceVersion . . . 260
4.43.23 データバックアップ構成 . . . 261
4.43.24 EmbeddedObjectMetadata . . . 261
4.43.25 外部クラスター . . . 262
4.43.26 Google 資格情報 . . . 262
4.43.27 輸入 . . . 263
4.43.28 インポート元 . . . 263
4.43.29 インスタンス ID . . . 263
4.43.30 InstanceReportedState . . . 264
4.43.31 LDAPBindAsAuth . . . 264
4.43.32 LDAPBindSearchAuth . . . 264
4.43.33 LDAPConfig . . . 265
4.43.34 LocalObjectReference . . . 265
4.43.35 メタデータ . . . 265
4.43.36 監視構成 . . . 266
4.43.37 NodeMaintenanceWindow . . . 266
4.43.38 PgBouncerIntegrationStatus . . . 267
4.43.39 PgBouncerSecrets . . . 267

4.43.40 PgBouncerSpec . . . 267
4.43.41 PodTemplateSpec . . . 268
4.43.42 プーラー . . . 268
4.43.43 プーラー統合 . . . 269
4.43.44 プーラーリスト . . . 269
4.43.45 PoolerSecrets . . . 269
4.43.46 プーラースペック . . . 269
4.43.47 プーラーのステータス . . . 270
4.43.48 PostInitApplicationSQLRefs . . . 270
4.43.49 Postgres 構成 . . . 270
4.43.50 回復対象 . . . 271
4.43.51 ReplicaClusterConfiguration . . . 272
4.43.52 ReplicationSlotsConfiguration . . . 272
4.43.53 ReplicationSlotsHAConfiguration . . . 272
4.43.54 ローリングアップデートステータス . . . 273
4.43.55 S3Credentials . . . 273
4.43.56 スケジュールされたバックアップ . . . 274
4.43.57 スケジュールされたバックアップ リスト . . . 274
4.43.58 スケジュールされたバックアップ仕様 . . . 275
4.43.59 スケジュールされたバックアップの状態 . . . 275
4.43.60 シークレットキーセレクター . . . 276
4.43.61 シークレットバージョン . . . 276
4.43.62 SecretsResourceVersion . . . 276
4.43.63 ServiceAccountTemplate . . . 277
4.43.64 ストレージ構成 . . . 277
4.43.65 SyncReplicaElectionConstraints . . . 278
4.43.66 トポロジー . . . 278
4.43.67 WalBackupConfiguration . . . 278
4.44 サポートされているリリース . . . 279
4.44.1 サポートポリシー . . . 280
4.44.2 命名スキーム . . . 280
4.44.3 CloudNativePG リリースのサポート状況 . . . 281
4.44.4 今後のリリース . . . 281
4.44.5 古いリリース . . . 282
4.44.6 サポートとは . . . 282
4.45 リリースノート . . . 283
4.46 CloudNativePG 1.19 のリリース ノート . . . 283
4.46.1 バージョン 1.19.1 . . . 283
4.46.2 バージョン 1.19.0 . . . 284
4.47 CloudNativePG 1.18 のリリース ノート . . . 286
4.47.1 バージョン 1.18.3 . . . 286
4.47.2 バージョン 1.18.2 . . . 287
4.47.3 バージョン 1.18.1 . . . 288
4.47.4 バージョン 1.18.0 . . . 289
4.48 CloudNativePG 1.17 のリリース ノート . . . 290
4.48.1 バージョン 1.17.5 . . . 290
4.48.2 バージョン 1.17.4 . . . 291

4.48.3 バージョン 1.17.3 . . . 292
4.48.4 バージョン 1.17.2 . . . 293
4.48.5 バージョン 1.17.1 . . . 293
4.48.6 バージョン 1.17.0 . . . 294
4.49 CloudNativePG 1.16 のリリース ノート . . . 295
4.49.1 バージョン 1.16.5 . . . 295
4.49.2 バージョン 1.16.4 . . . 296
4.49.3 バージョン 1.16.3 . . . 296
4.49.4 バージョン 1.16.2 . . . 297
4.49.5 バージョン 1.16.1 . . . 298
4.49.6 バージョン 1.16.0 . . . 299
4.50 CloudNativePG 1.15 のリリース ノート . . . 300
4.50.1 バージョン 1.15.5 . . . 300
4.50.2 バージョン 1.15.4 . . . 301
4.50.3 バージョン 1.15.3 . . . 301
4.50.4 バージョン 1.15.2 . . . 302
4.50.5 バージョン 1.15.1 . . . 303
4.50.6 バージョン 1.15.0 . . . 304
4.51 1.14.0 以前のリリース ノート . . . 305
4.51.1 バージョン 1.14.0 . . . 305
4.51.2 バージョン 1.13.0 . . . 306
4.51.3 バージョン 1.12.0 . . . 307
4.51.4 バージョン 1.11.0 . . . 308
4.51.5 バージョン 1.10.0 . . . 310
4.51.6 バージョン 1.9.2 . . . 311
4.51.7 バージョン 1.9.1 . . . 311
4.51.8 バージョン 1.9.0 . . . 312
4.51.9 バージョン 1.8.0 . . . 312
4.51.10 バージョン 1.7.1 . . . 313
4.51.11 バージョン 1.7.0 . . . 314
4.51.12 バージョン 1.6.0 . . . 315
4.51.13 バージョン 1.5.1 . . . 315
4.51.14 バージョン 1.5.0 . . . 315
4.51.15 バージョン 1.4.0 . . . 316
4.51.16 バージョン 1.3.0 . . . 317
4.51.17 バージョン 1.2.1 . . . 318
4.51.18 バージョン 1.2.0 . . . 318
4.51.19 バージョン 1.1.0 . . . 318
4.51.20 バージョン 1.0.0 . . . 319

CloudNativePG は、プライベート、パブリック、ハイブリッド、またはマルチクラウド環境で実行されている、サポートされている任意の Kubernetes クラスターで PostgreSQL ワークロードを管理するように設計されたオープン ソース オペレーターです。 CloudNativePG は、宣言的な構成や不変のインフラストラクチャなど、DevOps の原則と概念に準拠しています。

これは、高可用性と読み取り専用クエリのオフロードのために、選択された Kubernetes 名前空間に共存する単一のプライマリとオプションの数のレプリカで構成される PostgreSQL クラスターを表すクラスターと呼ばれる新しい Kubernetes リソースを定義します。

同じ Kubernetes クラスターに存在するアプリケーションは、オペレーターによってのみ管理されるサービスを使用して PostgreSQL データベースにアクセスできます。フェイルオーバーまたはスイッチオーバー後のプライマリ ロールの変更について心配する必要はありません。 Kubernetes クラスターの外部にあるアプリケーションは、Service または Ingress オブジェクトを構成して、TCP 経由で Postgres を公開する必要があります。 Web アプリケーションは、PgBouncer に基づくネイティブ接続プーラーを利用できます。

CloudNativePG はもともと EDB によって構築され、その後 Apache License 2.0 の下でオープン ソースをリリースし、2022 年 4 月に CNCF Sandbox に提出されました。ソース コード リポジトリは Github にあります。

注釈: Operator Capability Levels model に基づいて、ユーザーは CloudNativePG Operator から **"Level V - Auto Pilot"** の一連の機能を期待できます。

第1章　サポートされている Kubernetes ディストリビューション

CloudNativePG の各マイナー リリースは、さまざまな Kubernetes バージョンで動作するように設計されています。通常は、マイナーバージョンが最初にリリースされた時点で CNCF によってサポートされていたバージョンです。

詳細は **サポートされているリリース** ページを参照してください。

第2章　コンテナ イメージ

CloudNativePG community は、オペレーターとオペランド、つまり PostgreSQL の両方のコンテナー イメージを維持します。

CloudNativePG オペレーター コンテナー イメージは distroless です。

cloudnative-pg で利用できます。

PostgreSQL オペランド コンテナーイメージは、複数のアーキテクチャ上のすべての PGDG supported versions of PostgreSQL で、

postgres-containers から直接利用できます。

さらに、コミュニティは PostGIS extension のイメージを提供します。

警告： CloudNativePG では、Kubernetes クラスター内のすべてのノードが同じ CPU アーキテクチャを持っている必要があるため、ハイブリッド CPU アーキテクチャの Kubernetes クラスターはサポートされていません。

第3章　主な特徴

- 外部ツールを必要としない、高可用性のための Kubernetes API サーバーとの直接統合
- セルフヒーリング機能: * 最も整列されたレプリカを昇格させることによるプライマリインスタンスのフェイルオーバー * レプリカの自動再作成
- 選択したレプリカを昇格させることによる、プライマリインスタンスの計画的な切り替え
- スケールアップ/ダウン機能
- 任意の数のインスタンスの定義 (最小 1 - 1 つのプライマリ サーバー)
- アプリケーションをクラスターの唯一のプライマリサーバーに接続するための *read-write* サービスの定義
- ワークロードを読み取るためにアプリケーションを任意のインスタンスに接続するための**読み取り専用** サービスの定義
- クラスター `spec` : `pg_audit` 、`auto_explain` 、および `pg_stat_statements` による特定の一般的な Postgres 拡張機能を含む、PostgreSQL 構成の宣言型管理
- PVC テンプレートを使用したローカル永続ボリュームのサポート
- Pod での永続ボリューム ストレージの再利用
- WAL ファイル用の別ボリューム
- PostgreSQL マイナー バージョンのローリング アップデート
- オペレーターのアップグレードのためのインプレースまたはローリングアップデート
- TLS 接続とクライアント証明書認証
- カスタム TLS 証明書のサポート (cert-manager との統合を含む)
- オブジェクト ストアへの継続的なバックアップ (AWS S3 および S3 互換、Azure Blob Storage、および Google Cloud Storage)

*スタンバイからのバックアップ

- バックアップ保持ポリシー (リカバリ ウィンドウに基づく)
- オブジェクト ストア内の既存のバックアップからのフルリカバリおよびポイント イン タイム リカバリ
- PostgreSQL のメジャー アップグレードを含む、既存の PostgreSQL データベースのオフライン インポート

- 並列 WAL アーカイブと復元により、書き込みの多いシステムでデータベースが WAL 生成に追いつくことができます
- オブジェクト ストアにアップロードされたバックアップファイルのタグ付けをサポートし、オブジェクト ストアレイヤーでオプションの保持管理を有効にします。
- プライベート、パブリック、ハイブリッド、およびマルチクラウドアーキテクチャを有効にする、複数の Kubernetes クラスターにわたる PostgreSQL の展開
- 同期レプリカのサポート
- クラスター レベルでの HA 物理レプリケーション スロットのサポート
- PgBouncer による接続プーリング
- `nodeSelector` によるノード アフィニティのサポート
- `metrics` ポート (9187) を介した Prometheus のユーザー定義メトリックのネイティブでカスタマイズ可能なエクスポーター
- JSON 形式の PostgreSQL エラー メッセージの標準出力ロギング
- ポッドの `readOnlyRootFilesystem` セキュリティコンテキストを自動的に設定
- `kubectl` 用の `cnpg` プラグイン
- PostgreSQL クラスター全体、またはインスタンスのサブセットのフェンシング
- シンプルなバインドおよび検索+バインド LDAP クライアント認証
- マルチアーチ形式のコンテナ イメージ
- Postgres クラスターのハイバネーション

第4章　このガイドについて

クイックスタート の手順に従って、Kind または Minikube

を使用して、ローカルの Kubernetes クラスターで CloudNativePG をテストします。

Kubernetes と PostgreSQL の基本的な用語に慣れていない場合は、
始める前に を参照してください。

- Postgres, PostgreSQL and the Slonik Logo

カナダの PostgreSQL Community Association の商標または登録商標であり、その許可を得て使用されています。

4.1 始める前に

始める前に、Kubernetes と PostgreSQL に固有の用語について確認することが不可欠です。

4.1.1 Kubernetes の用語

Resource	Description
:ref:``nodeSelector` によるノード選択<*nodeSelector によるノード選択*>`	**ノード** は、仮想または物理のいずれかの Kubernetes のワーカーマシンであり、ポッドを実行するために必要なすべてのサービスがコントロール プレーン ノードによって管理されます。
ポッド セキュリティ ポリシー	**ポッド** は、Kubernetes クラスターにデプロイできる最小のコンピューティング ユニットであり、ネットワークとストレージを共有する 1 つ以上のコンテナーで構成されます。
:ref:``microservice` タイプ<*microservice タイプ*>`	**サービス** は、ポッドのグループで実行されるアプリケーションをネットワーク サービスとして公開し、アプリケーション間のサービス検出、負荷分散、フェイルオーバーなどの重要な機能を標準化する抽象化です。
オペレーター シークレットの定義	*secret* は、パスワード、アクセス キー、トークンなどの少量の機密データを保存し、ポッドで使用するように設計されたオブジェクトです。
ストレージ クラスによる構成	**ストレージ クラス** を使用すると、管理者はクラスター内のストレージのクラスを定義できます。これには、プロビジョナー (AWS EBS など)、再利用ポリシー、マウント オプション、ボリューム拡張などが含まれます。

次のページに続く

表 1 – 前のページからの続き

Resource	Description
永続ボリュームの要求	**永続ボリューム** (PV) は、管理者によって手動でプロビジョニングされた、または **ストレージ クラス** コントローラーによって動的にプロビジョニングされたストレージを表す Kubernetes クラスター内のリソースです。 PV は *persistent volume claim* を使用して Pod に関連付けられ、そのライフサイクルはそれを使用する Pod から独立しています。通常、PV は、特にパブリック クラウドではネットワーク ボリュームです。 *local persistent volume* (LPV) は、それを使用するポッドが実行されている特定のノードにのみ存在する永続ボリュームです。
永続ボリュームの要求	**永続ボリューム クレーム** (PVC) は、ストレージの要求を表し、サイズ、アクセス モード、または特定のストレージ クラスが含まれる場合があります。ポッドがノード リソースを消費するのと同様に、PVC は PV のリソースを消費します。
Namespace	**名前空間** は、Kubernetes クラスターの論理的で分離されたサブセットであり、より広い物理クラスター内の **仮想クラスター** と見なすことができます。名前空間を使用すると、管理者はプロジェクト、部門、チームなどに基づいて個別の環境を作成できます。
役割ベースのアクセス制御 *(RBAC)*	*Role Based Access Control* (RBAC) は、*role-based security* とも呼ばれ、コンピューター システムのセキュリティで使用される方法で、システムのネットワークとリソースへのアクセスを許可されたユーザーのみに制限します。Kubernetes には、名前空間とクラスター レベルで役割を制御し、それらを特定のリソースと個人に関連付けるためのネイティブ API があります。
CRD によるオペランドイメージのオーバーライド	**カスタム リソース定義** (CRD) は Kubernetes API の拡張機能であり、開発者は **カスタム リソースと呼ばれる** 新しいデータ型とオブジェクトを作成できます。
オペレーター マニフェストを直接使用する	**オペレーター** は、1 つ以上のアプリケーションまたは特定のサービスを管理する際に人間のオペレーターによって通常実行される手順を自動化するカスタム リソースです。オペレーターは、リソースの定義された状態が常に観察された状態と一致するように Kubernetes を支援します。
:ref:``kubectl` に *cnpg* プラグインを使用する <*kubectl* に *cnpg* プラグインを使用する>`	*kubectl* は、Kubernetes クラスターの管理に使用されるコマンドライン ツールです。

CloudNativePG には、コミュニティでサポートされている Kubernetes バージョンが必要です。詳細は サポートされているリリース ページを参照してください。

4.1.2 PostgreSQL 用語

Resource	Description
Instance	「IP アドレス」と「TCP ポート」のペア (通常は 5432) で実行およびリッスンする Postgres サーバー プロセス。
Primary	読み取り操作と書き込み操作の両方を受け入れることができる PostgreSQL インスタンス。
Replica	クラスター内の唯一のプライマリ インスタンスからレプリケートする PostgreSQL インスタンスであり、先行書き込みログ (WAL) レコードのストリームを読み取ることによって更新されます。レプリカは、**スタンバイ** または **セカンダリ** サーバーとも呼ばれます。 PostgreSQL は、物理ストリーミングレプリケーション (非同期/同期) とファイルベースのログ シッピング (非同期) に依存しています。
Hot Standby	*レプリカ*が読み取り専用のワークロードを受け入れることを可能にする PostgreSQL の機能。
Cluster	高可用性 (HA) クラスターとして意図されている: 1 つのプライマリとオプションの任意の数のレプリカで構成される一連の PostgreSQL インスタンス。
Replica Cluster	選択した PostgreSQL クラスターからの継続的な復旧モードにある CloudNativePG *Cluster*。通常は Kubernetes クラスターの外部に存在します。これは、プライベート、パブリック、ハイブリッド、およびマルチクラウドのコンテキストでマルチクラスターの展開を可能にする機能です。
Designated Primary	別の PostgreSQL クラスターから継続的に復旧され、レプリカ クラスターがプライマリになる場合にプライマリになるように指定されている、レプリカ クラスター内の PostgreSQL スタンバイ インスタンス。
Superuser	PostgreSQL では、**スーパーユーザー** は *LOGIN* と *SUPERUSER* の両方の特権を持つロールです。セキュリティ上の理由から、CloudNativePG は、ローカルの Unix ドメイン ソケットを介して *peer* 認証を介して *postgres* ユーザーとして *postgres* データベースに接続することにより、管理タスクを実行します。
WAL アーカイブ	Write-Ahead Logging (WAL) は、データベース管理システムでデータの整合性を確保するための標準的な方法です。
PVC group	CloudNativePG の用語における PVC グループは、同じ PostgreSQL インスタンスに属する関連する PVC のグループです。つまり、PGDATA を含むメイン ボリューム (*storage*) と WAL 用のボリューム (*walStorage*) です。

4.1.3 クラウド用語

Resource	Description
Region	クラウドの **リージョン** は、**アベイラビリティ ゾーン** で編成された、隔離された独立した地理的エリアです。リージョン内のゾーンでは、往復のネットワーク レイテンシがほとんどありません。

次のページに続く

表 3 – 前のページからの続き

Resource	Description
Zone	クラウドの **アベイラビリティ ゾーン** (**ゾーン** とも呼ばれます) は、リソースを展開できるリージョン内の領域です。通常、アベイラビリティ ゾーンは、データ センターまたは同じデータ センターの独立した建物に対応します。

4.1.4 次はどうする

用語に慣れてきたので、選択したクラウド環境にオペレーターをデプロイする前に、 test CloudNativePG on your laptop using a local cluster を決定できます。

4.2 ユースケース

CloudNativePG は、完全なクラウド ネイティブエクスペリエンスのために、同じ Kubernetes クラスターに存在するアプリケーションと連携するように設計されています。

ただし、データベースは Kubernetes クラスター内でホストできますが、アプリケーションを同時にコンテナー化することはできず、VM などの **従来の環境** で実行する必要がある場合があります。

4.2.1 ケース 1: Kubernetes 内のアプリケーション

一般的な状況では、アプリケーションとデータベースは Kubernetes クラスター内の同じ名前空間で実行されます。

通常はステートレスなアプリケーションは、標準の `Deployment` として管理され、複数のレプリカが異なる Kubernetes ノードに分散され、`ClusterIP` サービスを通じて内部的に公開されます。

このサービスは、`Ingress` とプロバイダのロードバランサ機能を介して、HTTPS 経由でエンドユーザーに外部的に公開されます。

アプリケーションは、バックエンドの PostgreSQL データベースを使用して、信頼できる永続的な方法で状態を追跡します。アプリケーションは、TLS 接続を介して、現在のプライマリ インスタンスを指す CloudNativePG によって定義された `Cluster` リソースによって公開された読み書きサービスを参照します。`Cluster` リソースには、単一のプライマリ アーキテクチャと複数のスタンバイアーキテクチャのロジックが組み込まれており、Postgres で高可用性クラスターを管理する複雑さを隠しています。

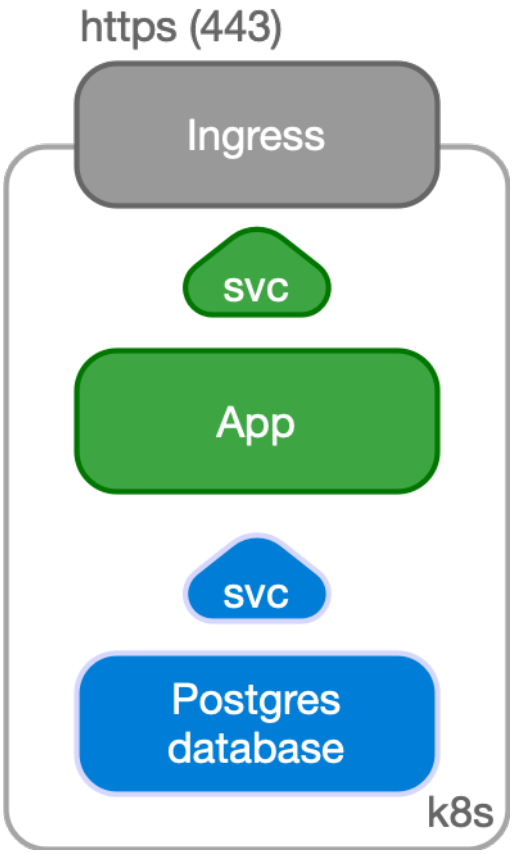


図 1: Application and Database inside Kubernetes

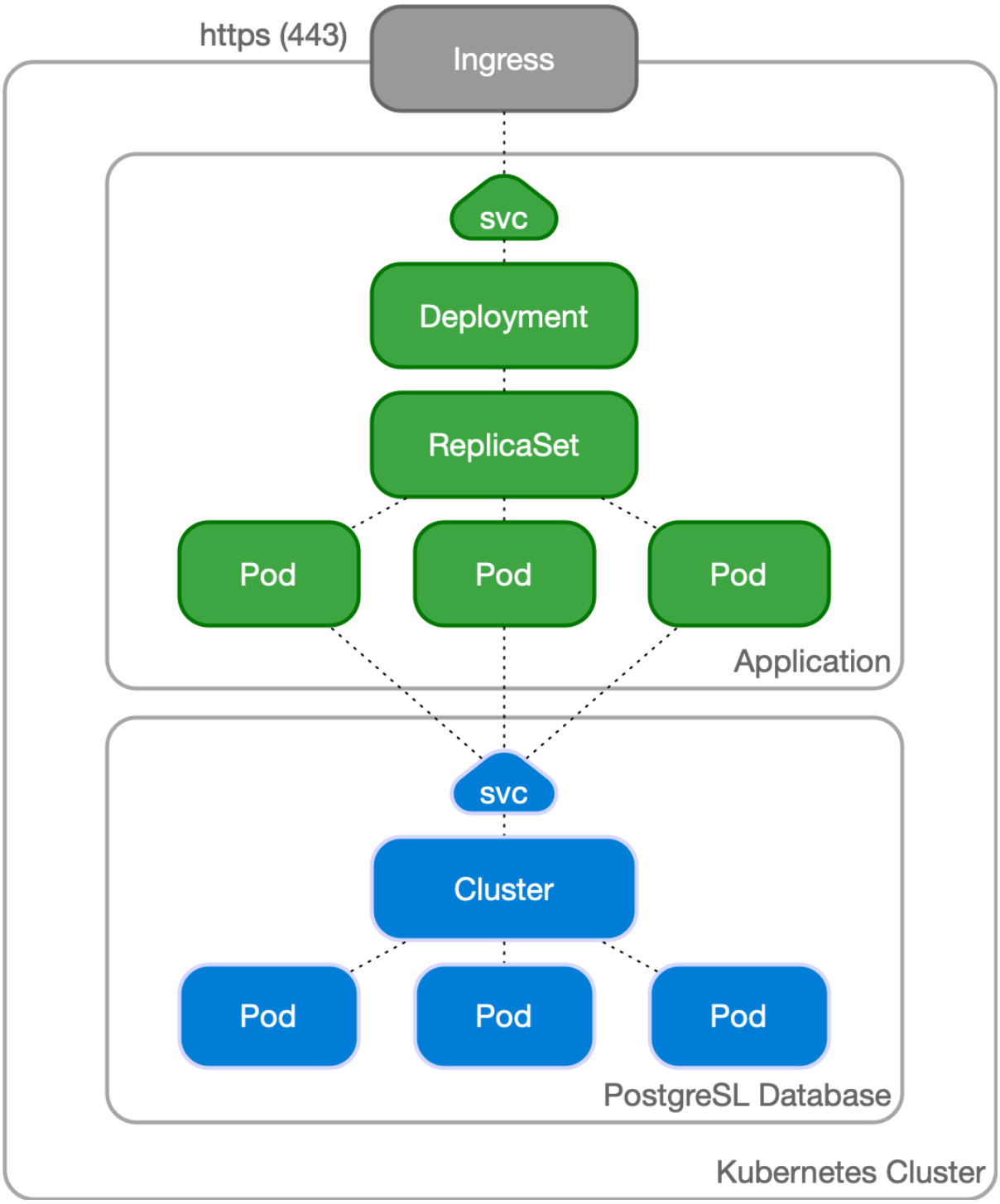


図 2: Close-up view of application and database inside Kubernetes

4.2.2 ケース 2: Kubernetes 外のアプリケーション

もう 1 つの考えられるユース ケースは、Kubernetes 内で PostgreSQL データベースを管理し、アプリケーションを Kubernetes の外部 (仮想化環境など) に置くことです。この場合、PostgreSQL は、Kubernetes で定義された Ingress リソースに対応する IP アドレス (またはホスト名) と TCP ポートで表されます。

アプリケーションは、PostgreSQL への TLS 接続の恩恵を受けることができます。

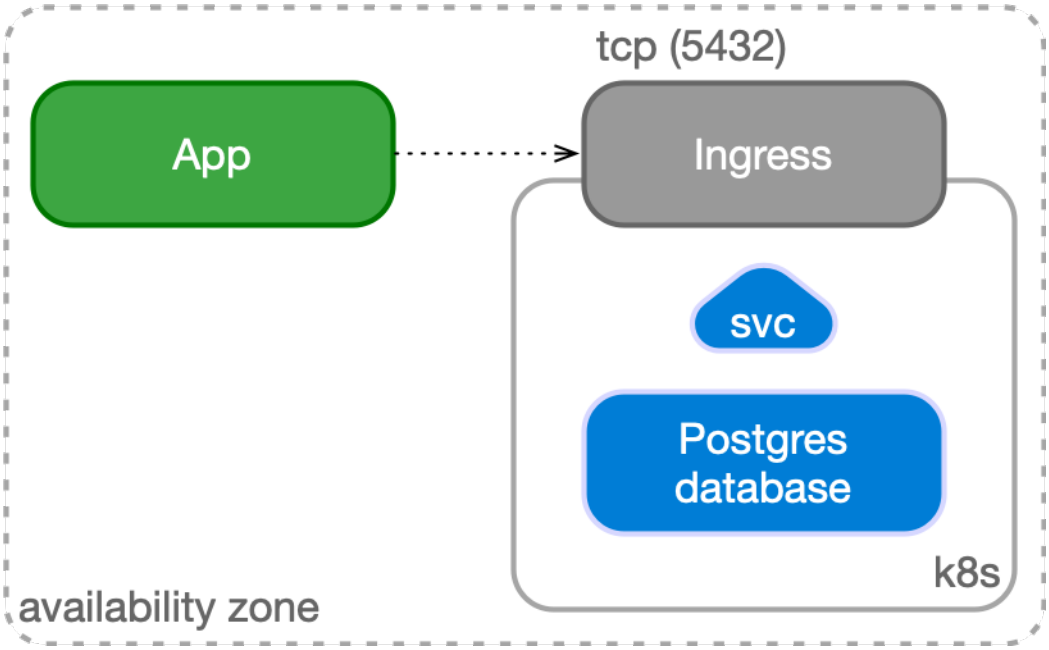


図 3: Application outside Kubernetes

4.3 建築

このセクションでは、PostgreSQL を Kubernetes にデプロイする際に考慮する必要がある主なアーキテクチャの側面について説明します。

重要: 自己管理型の Kubernetes 環境に PostgreSQL をデプロイする場合は、必ず *Kubernetes アーキテクチャ* をお読みください。

クラウドネイティブの世界への旅を計画し始めるときは、以下をご覧ください。

4.3.1 状態の同期

PostgreSQL はデータベース管理システムであるため、Kubernetes では**ステートフルワークロード** として扱う必要があります。ステートレスアプリケーションは主にトラフィック リダイレクトを使用して高可用性 (HA) と災害復旧 (DR) を実現しますが、データベースの場合、次のいずれかを採用することにより、できれば継続的かつ瞬時に状態を複数の場所にレプリケートする必要があります。2 つの戦略:

- **ストレージ レベルのレプリケーション**、通常は永続的なボリューム
- **アプリケーションレベルのレプリケーション**、この特定のケースでは PostgreSQL

CloudNativePG がアプリケーションレベルのレプリケーションに依存している理由は単純です。PostgreSQL データベース管理システムには、 **WriteAheadLog(WAL) 出荷**に基づく堅牢で信頼性の高い組み込みの **物理レプリケーション**機能が付属しています。 10 年以上にわたり、世界中の何百万人ものユーザーが本番環境で使用しています。

PostgreSQL は、ネットワークを介した非同期と同期の両方のストリーミングレプリケーションと、非同期のファイルベースのログシッピング (通常は、WAL ファイルをオブジェクトストアに格納するためのフォールバック オプションとして使用されます) をサポートしています。レプリカは通常、**スタンバイサーバー**と呼ばれ、**ホットスタンバイ**機能のおかげで、読み取り専用のワークロードにも使用できます。

重要: PostgreSQL を使用したストレージ レベルのレプリケーションはお勧めしません。ただし、CloudNativePG ではその戦略を採用できます。詳細については、KubeCon NA 2022 での Chris Milsted と Gabriele Bartolini による講演 Data On Kubernetes, Deploying And Running PostgreSQL And Patterns For Databases In a Kubernetes Cluster を参照してください。

このトピックについて詳しく説明されている場所。

4.3.2 Kubernetes アーキテクチャ

Kubernetes は、個別の物理的な場所 (データセンター、障害ゾーン、またはより頻繁には **可用性ゾーン** とも呼ばれる) にまたがる可能性をネイティブに提供し、冗長で低遅延のプライベートネットワーク接続を介して相互に接続します。

分散システムであるため、単一ゾーンの障害に対するコントロールプレーンの回復力を高めるために、Kubernetes クラスターの推奨される可用性ゾーンの最小数は 3 です。詳細は

Running in multiple zones を参照してください。これは、

各データセンターが常にアクティブで、ワークロードを同時に実行できることを意味します。

注釈: パブリック クラウド プロバイダーのマネージド Kubernetes サービスのほとんどは、各リージョンに 3 つ以上のアベイラビリティ ゾーンをすでに提供しています。

マルチアベイラビリティ ゾーンの Kubernetes クラスタ

3 つ以上のゾーンを持つマルチアベイラビリティ ゾーン Kubernetes アーキテクチャは、PostgreSQL の使用に推奨されるものです。このシナリオは、クラウドプロバイダーによって管理される Kubernetes サービスの典型です。

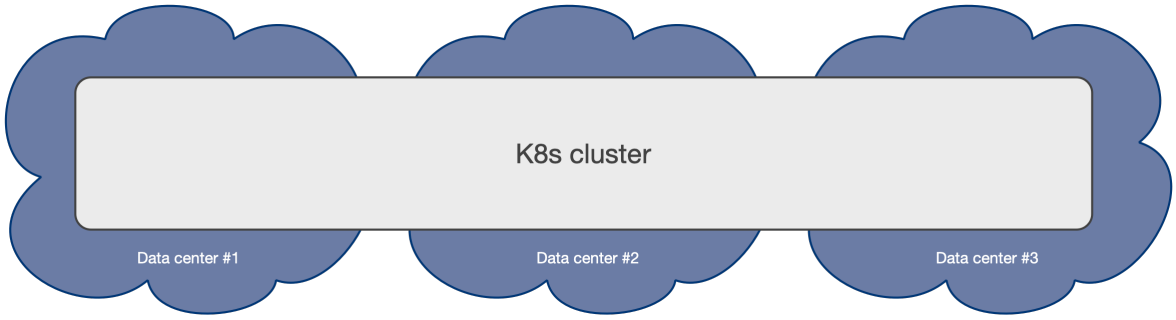


図 4: Kubernetes cluster spanning over 3 independent data centers

このようなアーキテクチャにより、CloudNativePG オペレーターは、すべてのアベイラビリティゾーンをアクティブとして扱うことにより、単一の Kubernetes クラスター内のゾーン全体で `Cluster` リソースの完全なライフサイクルを制御できます。アフィニティルール、容認、およびノードセレクターで)、自動フェイルオーバー、自己修復、および更新を行います。すべてが単一の Kubernetes クラスター内のゾーン間でシームレスに機能します。

> *PostgreSQL のアーキテクチャ* を参照してください。

ストレージ、ワーカーノード、およびアベイラビリティ ゾーンレベルでシェアード ナッシング デプロイを使用して、同じ Kubernetes クラスター内で PostgreSQL クラスターを設計する方法の詳細については、以下のセクションを参照してください。

さらに、追加の **マルチアベイラビリティ ゾーンの *Kubernetes* クラスタ** を利用して、「パッシブ」PostgreSQL レプリカクラスターをホストすることができます。これは主に、DR、読み取り専用操作、またはリージョン間の可用性のために使用する必要がありますが、この場合のフェイルオーバーとプロモーションは手動で行う必要があります。

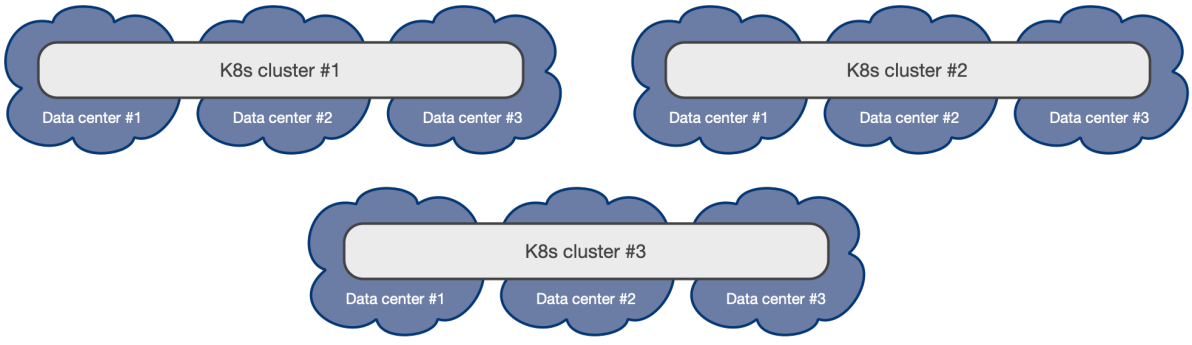


図 5: Example of a multiple Kubernetes cluster architecture distributed over 3 regions each with 3 independent data centers

単一の可用性ゾーンの Kubernetes クラスター

Kubernetes クラスターにアベイラビリティ ゾーンが 1 つしかない場合でも、CloudNativePG は PostgreSQL データベースの HA と DR の結果を改善する多くの機能を提供し、単一障害点 (SPoF) を可能な限りゾーンのレベルに押し上げます -つまり、CloudNativePG クラスターに障害が発生する前に、ゾーンを停止する必要があります。

このシナリオは、1 つのデータセンターしか使用できない自己管理型オンプレミス Kubernetes クラスターの典型です。

シングル アベイラビリティ ゾーン Kubernetes は、残念ながら、低遅延接続 (通常は同じ大都市圏) の範囲内で **2 つのデータセンター**しか利用できない場合に実行可能な唯一のオプションです。 -可用性ゾーン Kubernetes クラスター (3 つのゾーンの最小数に達していないため) をアクティブ/パッシブ構成で 2 つの異なる Kubernetes クラスターを作成するように強制し、2 番目のクラスターは主に災害復旧に使用されます。

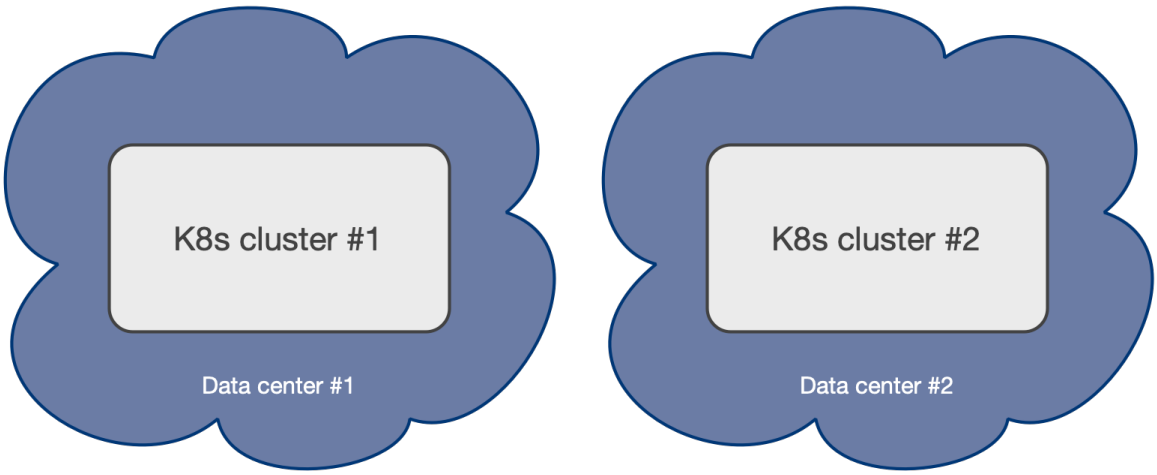


図 6: Example of a Kubernetes architecture with only 2 data centers

ヒント:

Kubernetes ジャーニーの初期段階にある場合は、このドキュメントをインフラストラクチャ チームと共有してください。 2 つのデータ センターのセットアップは、従来のベアメタルまたは VM ベースのインフラストラクチャから Kubernetes への「リフト アンド シフト」移行の結果にすぎない可能性があり、3 つ以上のゾーンのシナリオで Kubernetes が提供する利点は知られていない可能性があります。 、またはインフラストラクチャ アーキテクチャの設計時に対処されました。最終的に、他の 2 つの場所に接続された 3 番目の物理的な場所は、組織にとって検討すべき有効なオプションになる可能性があります。これは、日々の複雑さをアプリケーション レベルから物理インフラストラクチャ レベルに移行することで、インフラストラクチャの全体的なコストを削減するためです。

PostgreSQL のアーキテクチャ を参照してください。

ストレージ レベルとワーカーノード レベルのみでシェアード ナッシングデプロイを使用して、単一の可用性ゾーンの Kubernetes クラスター内で PostgreSQL クラスターを設計する方法の詳細については、以下

のセクションを参照してください。HA の場合、このようなシナリオでは、PostgreSQL インスタンスが異なるワーカーノードに配置され、同じストレージを共有しないことがさらに重要になります。

DR の場合、追加の **マルチアベイラビリティ ゾーンの *Kubernetes* クラスタ** を使用して「パッシブ」PostgreSQL レプリカ クラスターをホストすることにより、単一ゾーンの上に SPoF をプッシュできます。このシナリオの他の Kubernetes ワークロードと同様に、プライマリとしての Kubernetes クラスターの昇格は手動で行う必要があります。以下で説明するように、オペレーターは単一の Kubernetes クラスター内でしか動作できないため、現時点では、CloudNativePG を使用した PostgreSQL で使用できる Kubernetes クラスター間の自動フェイルオーバーはありません。

4.3.3 PostgreSQL のアーキテクチャ

CloudNativePG は、非同期および同期ストリーミングレプリケーションに基づくクラスターをサポートし、同じ Kubernetes クラスター内で複数のホット スタンバイレプリカを管理します。仕様は次のとおりです。

- 1 つのプライマリ、オプションで HA 用の複数のホット スタンバイレプリカ
- アプリケーションで利用可能なサービス: * `-rw` : アプリケーションはクラスターのプライマリ インスタンスにのみ接続します* `-ro` : アプリケーションは読み取り専用ワークロードのホットスタンバイ レプリカにのみ接続します * `-r` : アプリケーションは読み取り専用ワークロードのいずれかのインスタンスに接続します
- PostgreSQL クラスターの回復力を高めるために推奨されるシェアードナッシング アーキテクチャ: * PostgreSQL インスタンスは異なる Kubernetes ワーカーノードに存在し、ネットワークのみを共有する必要があります。* PostgreSQL インスタンスは、同じ Kubernetes クラスタ / リージョン内の異なるアベイラビリティ ゾーンに存在する必要があります

以下の図は、3 つの異なるアベイラビリティー ゾーンにまたがる PostgreSQL クラスターの推奨されるシェアード ナッシングアーキテクチャを単純化して示したもので、それぞれが PostgreSQL データ専用のローカル ストレージを備えた個別のノードで実行されます。

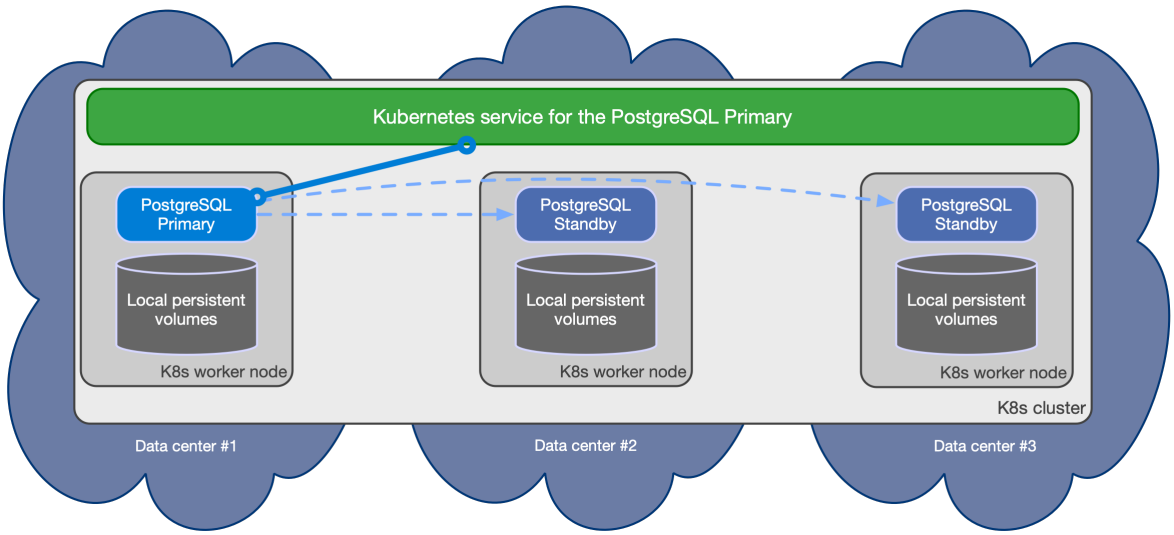


図 7: Bird-eye view of the recommended shared nothing architecture for PostgreSQL in Kubernetes

クラスタのトポロジが変更された場合、CloudNativePG は上記のサービスを自動的に更新します。たとえば、フェイルオーバーの場合、-rw サービスを自動的に更新して、昇格したプライマリを指すようにし、アプリケーションからのトラフィックがシームレスにリダイレクトされるようにします。

読み書きワークロード

次の図に示すように、アプリケーションは、Kubernetes オペレーターによって「現在のプライマリ」として選択された PostgreSQL インスタンスに接続することを決定できます。

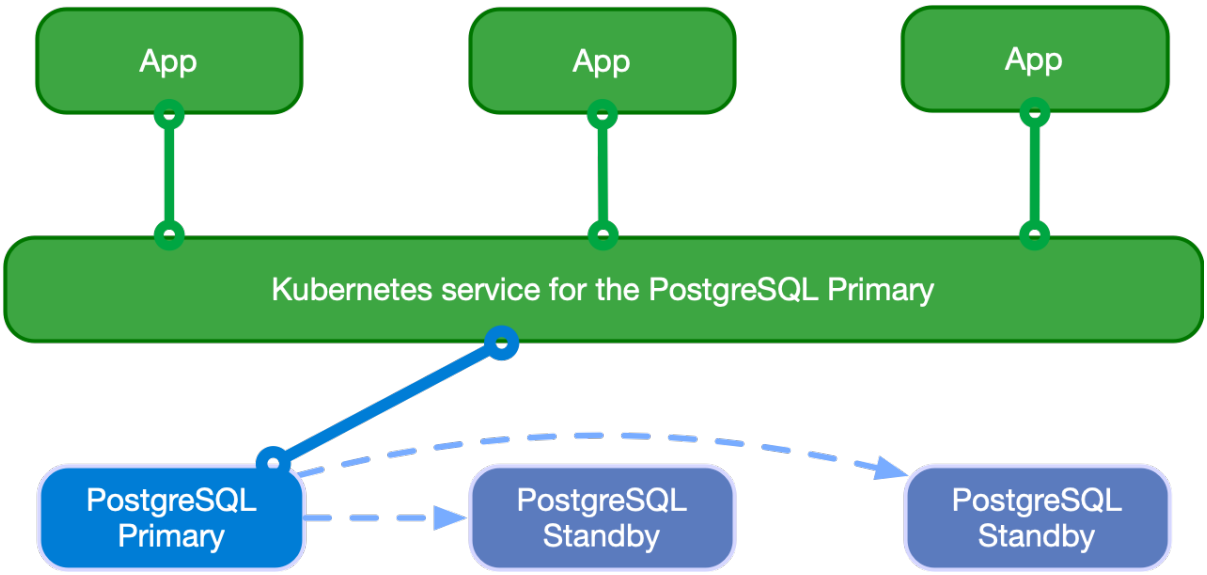


図 8: Applications writing to the single primary

アプリケーションは -rw サフィックス サービスを使用できます。

プライマリが一時的または永続的に利用できない場合、高可用性の目的で CloudNativePG はフェイルオーバーをトリガーし、-rw サービスをクラスターの別のインスタンスにポイントします。

読み取り専用ワークロード

重要: アプリケーションは、 Hot Standby の制限を認識している必要があります。

これらのワークロードを処理する際の PostgreSQL の動作方法を紹介し、熟知しています。

アプリケーションは、オペレーターが提供する -ro サービスを介してホット スタンバイレプリカにアクセスできます。このサービスにより、アプリケーションはプライマリノードから読み取り専用クエリをオフロードできます。

次の図は、アーキテクチャを示しています。

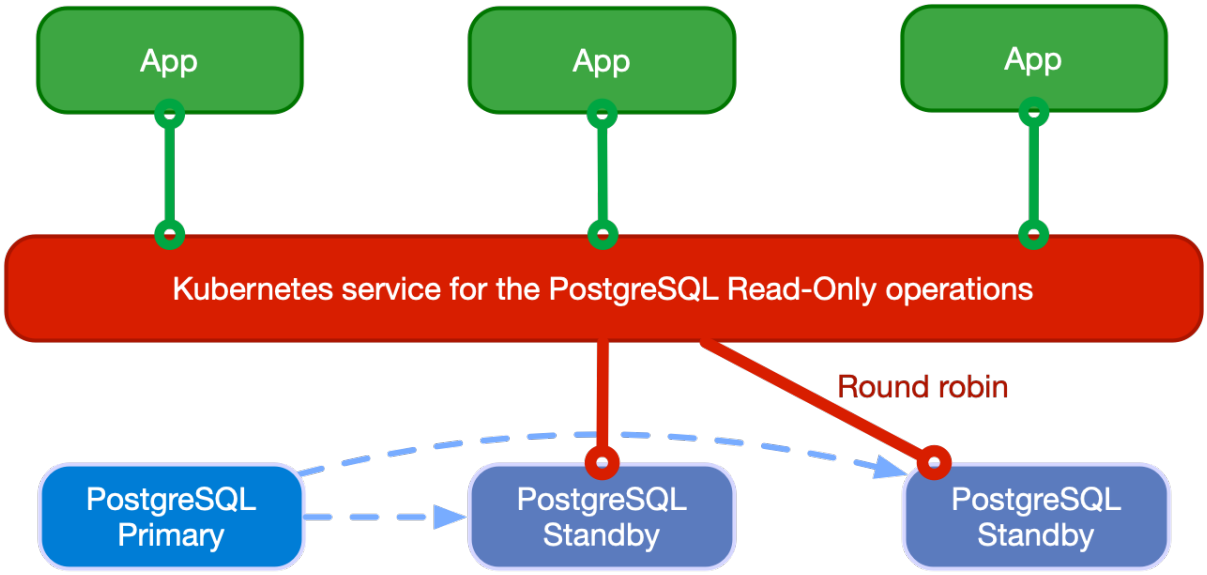


図 9: Applications reading from hot standby replicas in round robin

アプリケーションは、-r サービスを介して任意の PostgreSQL インスタンスにアクセスすることもできます。

4.3.4 Kubernetes クラスター全体へのデプロイ

分散 PostgreSQL クラスタでは、常にプライマリとして機能する 1 つの PostgreSQL インスタンスしか存在できません。これは、アプリケーションがいつでも 1 つの Kubernetes クラスター内でしか書き込みできないことを意味します。

ただし、ビジネス継続性の目的では、次のことが基本となります。

- PostgreSQL バックアップデータを複数の場所、リージョンに保存し、場合によっては異なるプロバイダーを使用することで、グローバル**目標復旧時点** (RPO) を削減します (ディザスター リカバリー)
- プライマリ Kubernetes クラスターを超えた PostgreSQL レプリケーションを利用することで、グローバル **目標復旧時間** (RTO) を短縮します (高可用性)

上記の問題に対処するために、CloudNativePG は *PostgreSQL* **レプリカクラスター** の概念を導入しています。レプリカクラスターは、プライベート、パブリック、ハイブリッド、およびマルチクラウドのコンテキストでマルチクラスター展開を可能にする CloudNativePG の方法です。

レプリカ クラスターは別の `Cluster` リソースです。

1. 定義された外部ソース クラスタからの `bootstrap` オプションとして `pg_basebackup` または完全な `recovery` を持つ
2. `replica.enabled` オプションを `true` に設定する
3. `replica.source` で識別される定義済みの外部クラスター (通常は Kubernetes クラスターの外部にある) から複製する
4. リカバリ オブジェクト ストアから受信した WAL 情報の再生 (PostgreSQL の `restore_command` パラメータを使用)、ストリーミングレプリケーション経由 (PostgreSQL の `primary_conninfo` パラメー

タを使用)、または 2 つのいずれか (`barmanObjectStore` と `connectionParameters` の両方が外部クラスタで定義されている場合)

5. PostgreSQL のホットスタンバイでサポートされているように、読み取り接続のみを受け入れる

参考:

PostgreSQL クラスターを別のクラスターから複製する方法の詳細については、 ブートストラップ を参照してください (*externalClusters* セクションで定義)。

以下の図は、2 つの異なる Kubernetes クラスターにまたがる PostgreSQL クラスターを示しています。プライマリ クラスターは最初の Kubernetes クラスターにあり、レプリカ クラスターは 2 番目の Kubernetes クラスターにあります。 2 番目の Kubernetes クラスターは、会社のディザスター リカバリークラスターとして機能し、災害が発生して最初のクラスターが使用できなくなった場合にアクティブ化する準備ができています。

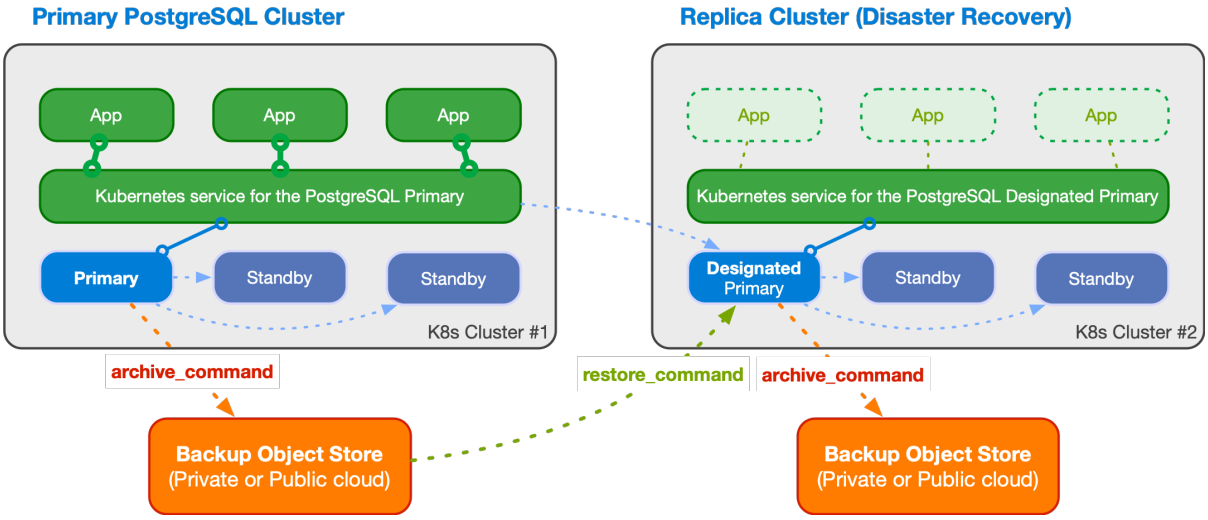


図 10: An example of multi-cluster deployment with a primary and a replica cluster

レプリカ クラスタは、プライマリクラスタと同じアーキテクチャを持つことができます。プライマリインスタンスの代わりに、レプリカ クラスターには **指定されたプライマリ**インスタンスがあります。これは、ストリーミングレプリケーション (対称アーキテクチャ) で任意の数のカスケード スタンバイ サーバーを持つスタンバイサーバーです。

指定されたプライマリはいつでも昇格でき、レプリカクラスタは書き込み接続を受け入れることができるプライマリクラスタになります。

警告: CloudNativePG は現在、クラスタ間の切り替えやフェイルオーバーを実行していません。このような操作は、手動で実行するか、マルチクラスター/フェデレーション クラスター対応機関に委任する必要があります。各 PostgreSQL クラスターは、他から独立しています。

上記の例で指定されたプライマリは、`restore_command` および `barman-cloud-wal-restore` を介してファイルベースの WAL 出荷のフォールバック オプションを使用して、WAL ストリーミング (`primary_conninfo`) を介して供給されます。

CloudNativePG では、複数のレプリカクラスターを定義できます。レプリカの数を減らしてレプリカクラスターを定義し、クラスターがプライマリに昇格したときにこの数を増やすこともできます。

4.4 インストールとアップグレード

4.4.1 Kubernetes へのインストール

オペレーター マニフェストを直接使用する

このオペレーターは、kubectl を介して適用される YAML マニフェストを介して、Kubernetes の他のリソースと同様にインストールできます。

> latest operator manifest をインストールできます

このマイナー リリースでは、次のようになります。

```
kubectl apply -f \
  https://raw.githubusercontent.com/cloudnative-pg/cloudnative-pg/release-1.19/
↪releases/cnpg-1.19.1.yaml
```

次の方法で確認できます。

```
kubectl get deploy -n cnpg-system cnpg-controller-manager
```

kubectl に cnpg プラグインを使用する

cnpg プラグインを使用して、静的マニフェストにある既定の構成オプションをオーバーライドできます。

たとえば、デフォルトの最新のマニフェストを生成し、ウォッチの名前空間を特定の名前空間のみに変更するには、次を実行できます。

```
kubectl cnpg install generate \
  --watch-namespaces "specific-namespace" \
  > cnpg_for_specific_namespace.yaml
```

より包括的な例については、 :ref:``kubectl` に *cnpg* プラグインを使用する<*kubectl* に *cnpg* プラグインを使用する>` のドキュメントを参照してください。

> **警告:** GKE に CloudNativePG をデプロイしてエラー (*... failed to call webhook...*) が発生した場合は、公式の docs で説明されているように、デフォルトでは、ワーカーノードとコントロール プレーン間のトラフィックがいくつかの特定のポートを除いてファイアウォールによってブロックされることに注意してください。

そして、この よくある問題 によって。 Webhook サービスの targetPort を許可されたサービスの 1 つに変更するか、ファイアウォールで Webhook のポート (9443) を開く必要があります。

最新の開発スナップショットのテスト

次の公式パッチ リリースの前に CloudNativePG の最新の開発スナップショットをテストまたは評価する場合は、 cloudnative-pg/artifacts からマニフェストをダウンロードできます。

これにより、現在のトランク (メイン) およびサポートされている各リリースに簡単にアクセスできます。

たとえば、このマイナーリリースのオペレーターの最新のスナップショットを次のようにインストールできます。

```
curl -sSfL \
  https://raw.githubusercontent.com/cloudnative-pg/artifacts/release-1.19/manifests/
↪operator-manifest.yaml | \
  kubectl apply -f -
```

重要: スナップショットは CloudNativePG でサポートされておらず、本番環境での使用を意図していません。

Helm チャートの使用

オペレーターは、提供されている *Helm チャートの使用*を使用してインストールできます。

4.4.2 導入の詳細

Kubernetes では、オペレーターはデフォルトで `cnpg-system` 名前空間に Kubernetes `Deployment` としてインストールされます。この配置の名前は、インストール方法によって異なります。マニフェストまたは `cnpg` プラグインを介してインストールすると、デフォルトで `cnpg-controller-manager` と呼ばれます。Helm 経由でインストールした場合、デフォルト名は `cnpg-cloudnative-pg` です。

注釈: Helm では、 *"values.yaml"* file の *fullnameOverride* フィールドを介してデプロイメントの名前をカスタマイズできます。

`kubectl` で `describe` コマンドを使用して、詳細情報を取得できます。

```
$ kubectl get deployments -n cnpg-system
NAME                READY   UP-TO-DATE   AVAILABLE   AGE
<deployment-name>   1/1     1            1           18m
```

```
kubectl describe deploy \
  -n cnpg-system \
  <deployment-name>
```

他の Deployment と同様に、ReplicaSet の上にあり、ローリングアップグレードをサポートします。 CloudNativePG オペレーターのデフォルト構成には、ほとんどのインストールに適した単一レプリカのデプロイメントが付属しています。ポッドが実行されているノードに到達できなくなった場合、ポッドは別のノードで再スケジュールされます。

オペレーターレベルで高可用性が必要な場合は、オペレーターがリーダーの選択をサポートしている場合、Deployment 構成で複数のレプリカを指定できます。また、taints と tolerations を利用して、実際の PostgreSQL クラスターが実行されているノードと同じノードでオペレーターが実行されないようにすることもできます (これには、セルフマネージド Kubernetes インストールのコントロールプレーンが含まれる場合もあります)。

4.4.3 アップグレード

重要: *CloudNativePG 1.19 のリリース ノート* をよくお読みください

バージョンによっては追加の手順が必要になる場合があるため、アップグレードを実行する前に確認してください。

CloudNativePG オペレーターのアップグレードは、2 段階のプロセスです。

1. コントローラーと関連する Kubernetes リソースをアップグレードする
2. すべての PostgreSQL ポッドで実行されているインスタンスマネージャーをアップグレードします。

リリース ノートに別段の記載がない限り、最初のステップは通常、プレーンな Kubernetes インストールに新しいバージョンのマニフェストを適用するか、使用されているディストリビューションのネイティブ パッケージ マネージャーを使用して行われます (上記のセクションの手順に従ってください)。

2 番目のステップは、コントローラーを更新した後に自動的に実行されます。デフォルトでは、デプロイされたすべての PostgreSQL インスタンスのローリング更新がトリガーされ、新しいインスタンスマネージャーが使用されます。ローリング更新手順は、デフォルトで `unsupervised` に設定されている `primaryUpdateStrategy` オプションによって制御されるスイッチオーバーで最高潮に達します。`supervised` に設定した場合、ユーザーは `kubectl` の `cnpg` プラグインを介して新しいインスタンスを手動で昇格することにより、ローリングアップデートを完了する必要があります。

重要: *primaryUpdateStrategy* が *unsupervised* のデフォルト値に設定されている場合、オペレーターのアップグレードにより、PostgreSQL クラスターでスイッチオーバーがトリガーされ、(通常は無視できる) ダウンタイムが発生します。

バージョン 1.10.0 以降、ローリング更新の動作をインスタンスマネージャーのインプレース更新に置き換えることができます。後者は、PostgreSQL インスタンスの再起動を必要とせず、その結果、クラスターでのスイッチオーバーも必要ありません。デフォルトでは無効になっているこの動作について、以下で説明します。

インスタンスマネージャーのインプレース更新

デフォルトでは、CloudNativePG はオペレーターが更新されるたびにクラスターのローリング更新を発行します。オペレーターに同梱されている新しいインスタンスマネージャーは、init コンテナーを介して各 PostgreSQL ポッドに追加されます。

ただし、この動作は構成を介して変更し、インスタンスマネージャーのインプレース更新を有効にすることができます。これは、コンテナーを維持する PID 1 プロセスです。

内部的には、CloudNativePG のバージョン 1.10 のすべてのインスタンスマネージャーは、整合性検証フェーズが完了すると、既存の実行可能ファイルを置き換える新しい実行可能ファイルの挿入と、すべての内部プロセスの正常な終了をサポートします。新しいインスタンスマネージャーが新しいバイナリを使用して再起動すると、既に実行されている *postmaster* が採用されます。

その結果、PostgreSQL プロセスは更新の影響を受けず、スイッチオーバーを実行する必要がありません。コインの反対側は、ポッドが開始後に変更され、不変性の純粋な概念が壊れていることです。

オペレーター構成 で `ENABLE_INSTANCE_MANAGER_INPLACE_UPDATES` 環境変数を

`'true'` に設定することで、この機能を有効にできます。

インプレース アップグレード プロセスでは、Pod 内の init コンテナーイメージは変更されません。したがって、Pod の定義はオペレーターの現在のバージョンを反映しません。

重要: この機能では、すべてのポッド (演算子とオペランド) が同じプラットフォーム/アーキテクチャ (たとえば、すべての *linux/amd64*) で実行される必要があります。

バージョン間の互換性

CloudNativePG はセマンティック バージョニングに従います。同じ API バージョン内のオペレーターのすべてのリリースは、以前のものと互換性があります。現在の API バージョンは v1 で、オペレーターのバージョン 1.x.y に対応します。

新しい機能に加えて、オペレーターの新しいバージョンには、バグ修正と安定性の強化が含まれています。このため、最も安全で安定した Postgres 環境を維持するために各バージョンがリリースされるため、**ユーザーにはオペレーターの最新バージョンにアップグレードすることを強くお勧めします**。

CloudNativePG は現在、オペレーターの新しいバージョンを少なくとも毎月リリースしています。各バージョンが利用可能になったときに更新を適用できない場合は、バージョンをスキップせずに、各バージョンを順番にアップグレードして定期的に最新の状態にすることをお勧めします。

重要:

2022 年に、EDB は、頻繁なオンライン更新が不可能な環境で CloudNativePG の LTS リリースを計画しています。

CloudNativePG 1.19 のリリース ノート ページには、CloudNativePG のすべてのリリース

バージョンで導入された変更の詳細なリストが含まれており、新しいバージョンのソフトウェアにアップグレードする前に読む必要があります。

ほとんどのバージョンは直接アップグレード可能であり、その場合、プレーンな Kubernetes インストールに新しいマニフェストを適用するか、選択したディストリビューションのネイティブパッケージ マネージャーを使用するだけで十分です。

バージョンを直接アップグレードできない場合は、新しいバージョンをインストールする前に古いバージョンを削除する必要があります。これはユーザーデータには影響しませんが、オペレーター自体にのみ影響します。

4.5 クイックスタート

このセクションでは、 **親切** または **ミニキューブ** のローカル Kubernetes クラスターで CloudNativePG を使用して、ラップトップ/コンピューターで PostgreSQL クラスターをテストする方法について説明します。

> **警告:** このセクションに含まれる手順は、デモンストレーション、テスト、および実践のみを目的としており、本番環境では使用しないでください。

他の Kubernetes アプリケーションと同様に、CloudNativePG は YAML で記述された通常のマニフェストを使用してデプロイされます。

このページの手順に従うことで、ローカルの Kubernetes インストールで PostgreSQL クラスターを開始し、それを試すことができるはずです。

重要: Kubernetes クラスターに接続するために、マシンに *kubectl* がインストールされていることを確認してください。 *how to install `kubectl* <https://kubernetes.io/docs/tasks/tools/install-kubectl/>`__ の Kubernetes ドキュメントに従ってください。

4.5.1 パート 1 - ローカルの Kubernetes プレイグラウンドをセットアップする

最初の部分は、Minikube または Kind のインストールについてです。時間をかけてシステムについて読んで、どちらを先に進めるかを決めてください。いずれかを設定したら、パート 2 に進んでください。

> **パート *4 - Prometheus* と *Grafana* を使用してクラスターを監視する** では、ローカルでのテスト/評価のために Prometheus と

Grafana を使用して監視をセットアップする手順も提供しています。

ミニキューブ

Minikube は、Kubernetes をローカルで簡単に実行できるようにするツールです。 Minikube は、ラップトップの仮想マシン (VM) 内で単一ノードの Kubernetes クラスターを実行し、Kubernetes を試したり、日常的に Kubernetes を使用して開発したりしようとしているユーザーを対象としています。通常、VirtualBox と組み合わせて使用します。

詳細については、ローカルの個人環境の公式 Kubernetes documentation on how to install Minikube を参照してください。インストールしたら、次のコマンドを実行して minikube クラスターを作成します。

```
minikube start
```

これにより、Kubernetes クラスターが作成され、使用する準備が整います。次のコマンドで動作することを確認します。

```
kubectl get nodes
```

`minikube` という 1 つのノードが表示されます。

親切

仮想マシンのハイパーバイザーを使用したくない場合、Kind は Docker コンテナーの「ノード」を使用してローカルの Kubernetes クラスターを実行するためのツールです (Kind は実際には「Kubernetes IN Docker」を表します)。

> クイックスタート の手順に従って、環境に `kind`

をインストールし、次を使用して Kubernetes クラスターを作成します。

```
kind create cluster --name pg
```

4.5.2 パート 2 - CloudNativePG をインストールする

ラップトップで Kubernetes がインストールされて実行されているので、CloudNativePG のインストールに進むことができます。

> インストールとアップグレード セクションを参照してから、PostgreSQL

クラスターのデプロイに進んでください。

4.5.3 パート 3 - PostgreSQL クラスターをデプロイする

Kubernetes での他のデプロイと同様に、PostgreSQL クラスターをデプロイするには、目的の Cluster を定義する構成ファイルを適用する必要があります。

cluster-example.yaml サンプル ファイルは、デフォルトのストレージ

クラスを使用して単純な Cluster を定義し、ディスクスペースを割り当てます。

```
#  Example of PostgreSQL cluster

apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3

  # Example of rolling update strategy:
  # - unsupervised: automated update of the primary once all
  #                 replicas have been upgraded (default)
  # - supervised: requires manual supervision to perform
  #               the switchover of the primary
  primaryUpdateStrategy: unsupervised

  # Require 1Gi of space
  storage:
    size: 1Gi
```

3 ノードの PostgreSQL クラスターを作成するには、次のコマンドを実行する必要があります。

```
kubectl apply -f cluster-example.yaml
```

get pods コマンドを使用して、ポッドが作成されていることを確認できます。

```
kubectl get pods
```

これにより、デフォルトの名前空間でポッドが検索されます。クラスターを Kubernetes インストール上の他のワークロードから分離するために、クラスターをデプロイするための新しい名前空間をいつでも作成できます。または、ラベルを使用することもできます。オペレーターは、特定のクラスターに関連するすべてのオブジェクトに cnpg.io/cluster ラベルを適用します。例えば：

```
kubectl get pods -l cnpg.io/cluster=<CLUSTER>
```

重要: *cnpg.io/cluster* をラベルとして使用していることに注意してください。過去に *postgresql* を見たり使用したりしたことがあるかもしれません。このラベルは非推奨であり、将来的に削除される予定です。

cngp.io/cluster を使用してください。

デフォルトでは、オペレーターは、オペレーターがリリースされたときに、PostgreSQL の最新のメジャーバージョンの利用可能な最新のマイナーバージョンをインストールします。これは、`Cluster` 定義の `spec` セクションに `imageName` キーを設定することでオーバーライドできます。たとえば、PostgreSQL 13.6 をインストールするには、次のようにします。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
   # [...]
spec:
   # [...]
   imageName: ghcr.io/cloudnative-pg/postgresql:13.6
   #[...]
```

重要: 不変のインフラストラクチャ パラダイムでは、コンテナー イメージの特定のバージョンを常に指す必要があります。 *latest* や *13* などのタグは、クラスター内の更新ポリシーとバージョンの一貫性に関して予測できないシナリオにつながる可能性があるため、運用環境では使用しないでください。厳密に決定論的で反復可能な展開の場合、*<image>:<tag>@sha256:<digestValue>* 形式を使用してダイジェストをイメージ名に追加できます。

4.5.4 パート 4 - Prometheus と Grafana を使用してクラスターを監視する

重要: Prometheus と Grafana のインストールは、このプロジェクトの範囲外です。このセクションの手順は、実験と説明のみを目的として提供されています。

このセクションでは、可観測性のために Prometheus と Grafana をデプロイする方法、CloudNativePG クラスターを監視する Grafana ダッシュボードを作成する方法、およびアラート条件を定義する一連の Prometheus ルールを示します。

> Prometheus Community によって維持される Kube-Prometheus stack 、Helm

チャートを活用します。追加のドキュメントと背景については、プロジェクトの Web サイトを参照してください。

Kube-Prometheus-stack Helm チャートは、 Alert Manager を含む
プロメテウス オペレーターの例 と **パート** *4 - Prometheus* と *Grafana* **を使用してクラスターを監視する** デプロイメントをインストールします。

CloudNativePG クラスターの可観測性に役立つ初期設定を提供する、この Helm チャートのデプロイ用の構成ファイルが含まれています。

インストール

Helm チャートの使用 をまだインストールしていない場合は、 instructions

に従ってシステムにインストールしてください。

prometheus-community Helm チャートリポジトリを追加し、提供されているサンプル構成を使用して *Kube Prometheus* スタック をインストールする必要があります。

これは、次のコマンドで実現できます。

```
helm repo add prometheus-community \
  https://prometheus-community.github.io/helm-charts

helm upgrade --install \
  -f https://raw.githubusercontent.com/cloudnative-pg/cloudnative-pg/main/docs/src/
↪samples/monitoring/kube-stack-config.yaml \
  prometheus-community \
  prometheus-community/kube-prometheus-stack
```

完了すると、Prometheus、Grafana、Alert Manager が kube-stack-config.yaml ファイルの値でインストールされます。

- Prometheus のインストールから、Prometheus Operator が **any** PodMonitor を監視します (*monitoring* を参照)。
- Grafana インストールは、Grafana ダッシュボード ConfigMap を監視します。

参考:

上記のコマンドの詳細については、 helm install のドキュメントを参照してください。

いくつかのカスタム リソースが作成されていることがわかります。

```
% kubectl get crds
NAME                                        CREATED AT
...
alertmanagers.monitoring.coreos.com         <timestamp>
...
prometheuses.monitoring.coreos.com          <timestamp>
prometheusrules.monitoring.coreos.com       <timestamp>
...
```

および一連のサービス:

```
% kubectl get svc
NAME                                      TYPE        PORT(S)
...                                        ...          ...
prometheus-community-grafana              ClusterIP   80/TCP
prometheus-community-kube-alertmanager    ClusterIP   9093/TCP
```

(次のページに続く)

(前のページからの続き)

```
prometheus-community-kube-operator        ClusterIP   443/TCP
prometheus-community-kube-prometheus      ClusterIP   9090/TCP
```

Prometheus で見る

この時点で、モニタリングを有効にしてデプロイされた CloudNativePG クラスターは、Prometheus を介して監視可能になります。

たとえば、`PodMonitor` を有効にして単純なクラスターをデプロイできます。

```
kubectl apply -f - <<EOF
- --
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-with-metrics
spec:
  instances: 3

  storage:
    size: 1Gi

  monitoring:
    enablePodMonitor: true
EOF
```

Prometheus にアクセスするには、Prometheus サービスをポート転送します。

```
kubectl port-forward svc/prometheus-community-kube-prometheus 9090
```

次に、Prometheus コンソールにローカルでアクセスします: http://localhost:9090/

監視スタックが正常にデプロイされ、`enablePodMonitor: true` を含むクラスターがあると仮定すると、CloudNativePG クラスターに関連する一連のメトリックが見つかるはずです。詳細については、*monitoring section* を参照してください。

`prometheusRule` を作成することで、いくつかのアラートを定義できるようになりました。

```
kubectl apply -f \
  https://raw.githubusercontent.com/cloudnative-pg/cloudnative-pg/main/docs/src/
↪samples/monitoring/cnpg-prometheusrule.yaml
```

デフォルトのアラートが表示されるはずです。

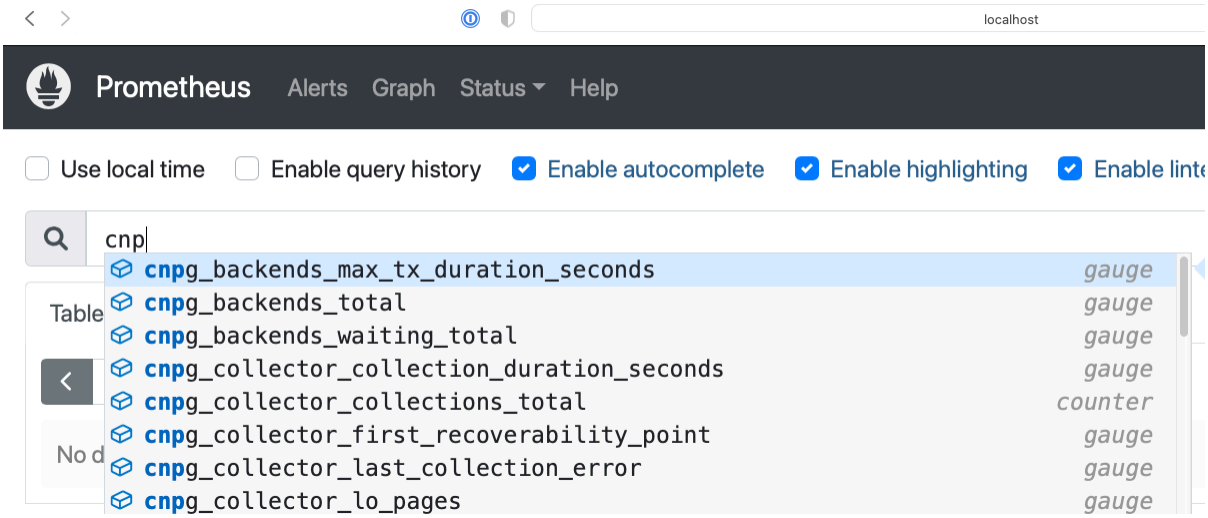


図 11: local prometheus

```
% kubectl get prometheusrules
NAME                                                AGE
cnpg-default-alerts                                 3m27s
```

Prometheus コンソールで、*Alerts* メニューをクリックして、インストールしたばかりのアラートを表示できます。

Grafana ダッシュボード

「プレーン」インストールでは、事前定義されたダッシュボードなしで Grafana がデプロイされます。

ポート転送できます：

```
  kubectl port-forward svc/prometheus-community-grafana 3000:80

`http://localhost:3000/ <http://localhost:3000/>`__  でローカルに Grafana にアクセスします。
```

資格情報 `admin` をユーザー名として、`prom-operator` をパスワードとして提供します (`kube-stack-config.yaml` で定義)。

サンプルの Grafana ダッシュボードをインストールできるようになりました。

```
kubectl apply -f \
  https://raw.githubusercontent.com/cloudnative-pg/cloudnative-pg/main/docs/src/
↪samples/monitoring/grafana-configmap.yaml
```

これは、Grafana ページによって数秒で取得されます。 `CloudNativePG` ダッシュボードが表示されます。

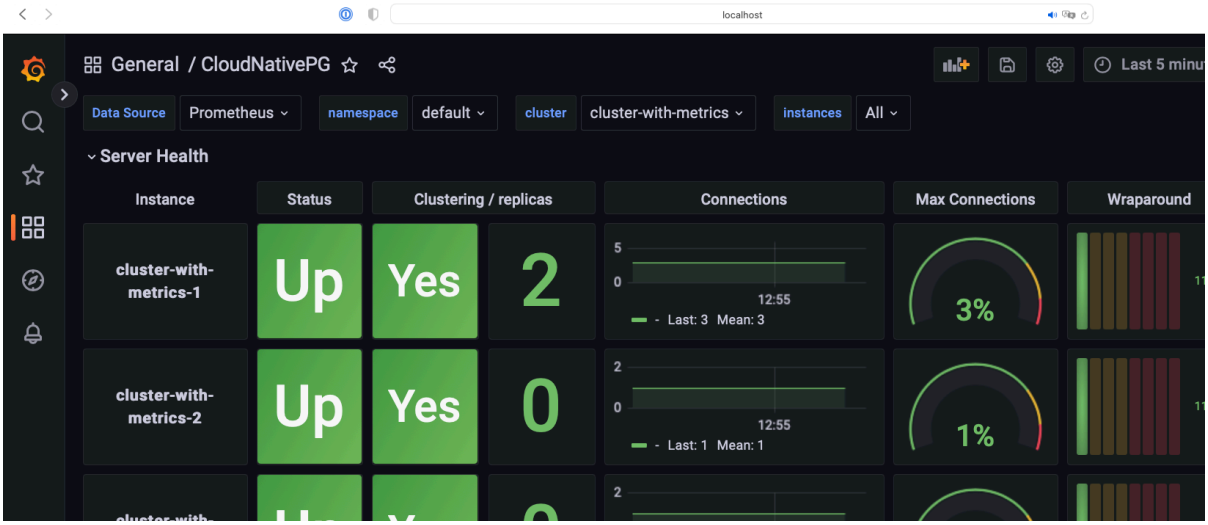


図 12: local grafana

このセットアップ例では、Prometheus と Grafana の両方が、モニタリングを有効にしてデプロイされた他の CloudNativePG クラスターをピックアップすることに注意してください。

4.6 ブートストラップ

このセクションでは、新しい PostgreSQL クラスターを作成するために必要なオプションと、その背後にある設計の根拠について説明します。新しいクラスターをブートストラップするには、主に次の 2 つの方法があります。

- 最初から (`initdb`)
- 既存の PostgreSQL クラスターから直接 (`pg_basebackup`) または間接的に (`recovery`)

`initdb` ブートストラップは、既存の Postgres クラスターから 1 つ以上のデータベースをインポートする可能性も提供します。これは、Kubernetes の外部であっても、Postgres のメジャーバージョンが異なります。この機能の詳細については、 *Postgres* **データベースのインポート**セクションを参照してください。

重要: 既存のクラスターからのブートストラップにより、**レプリカ クラスター**を作成する可能性が開かれます。これは、継続的に復旧され、ソースと同期され、読み取り専用接続を受け入れる独立した PostgreSQL クラスターです。

警告: CloudNativePG では、*postgres* ユーザーとデータベースの両方が常に存在する必要があります。ローカル Unix ドメイン ソケットを使用して、クラスターで管理タスクを実行するには、*peer* 認証を介して *postgres* ユーザーとして *postgres* データベースに接続する必要があります。 **削除しないでください** *postgres* ユーザーまたは *postgres* データベース!!!

4.6.1 bootstrap セクション

bootstrap メソッドは、クラスター仕様の bootstrap セクションで定義できます。 CloudNativePG は現在、次のブートストラップメソッドをサポートしています。

- initdb : 新しい PostgreSQL クラスターを初期化します (デフォルト)
- recovery : バックアップ オブジェクトストアを介して既存のクラスターから復元し、すべての利用可能な WAL ファイルまたは特定の時点までを再生することにより、PostgreSQL クラスターを作成します。
- pg_basebackup : ストリーミングレプリケーションプロトコル経由で pg_basebackup を使用して、同じメジャーバージョンの既存のクラスターを複製することにより、PostgreSQL クラスターを作成します。Kubernetes の外部からでも、データベースを CloudNativePG に移行する場合に役立ちます。

initdb メソッドとは異なり、recovery と pg_basebackup の両方が別のクラスター (オフラインまたはオンライン) に基づいて新しいクラスターを作成し、レプリカクラスターの起動に使用できます。どちらも外部クラスターの定義に依存しています。

詳細については。

4.6.2 externalClusters セクション

externalClusters セクションでは、現在のクラスターに関連する 1 つ以上の PostgreSQL クラスターを定義できます。将来的には、このセクションではより複雑なシナリオが可能になりますが、現時点では、物理レプリケーションに基づいてクロスリージョン PostgreSQL クラスターを定義し、さまざまな Kubernetes クラスターまたは従来の VM/ベアメタル環境にまたがることを意図しています。

ブートストラップに関する限り、externalClusters を使用して、pg_basebackup メソッドまたは recovery メソッドのソース PostgreSQL クラスターを定義できます。外部クラスターには以下が必要です。

- source オプションを介して参照として使用される元のクラスターを識別する名前
- 以下の少なくとも 1 つ:
- ストリーミング接続に関する情報 - **リカバリオブジェクトストア**に関する情報。これは、ソース クラスタのバックアップファイル、つまりベース バックアップと WAL アーカイブを含む Barman Cloud 互換オブジェクト ストアです。

注釈: 復旧オブジェクト ストアは通常、AWS S3、Azure Blob Storage、または Barman Cloud によって管理される Google Cloud Storage ソースです。

ストリーミング接続のみが定義されている場合、ソースは pg_basebackup メソッドに使用できます。リカバリ オブジェクトストアのみが定義されている場合、ソースは recovery メソッドに使用できます。両方が定義されている場合、2 つのブートストラップ方法のいずれかを選択できます。

さらに、pg_basebackup または完全な recovery 時点) の場合、クラスターはレプリカ クラスターモードの対象となります。これは、ストリーミング、PostgreSQL の restore_command を介した WAL シッピング、または 2 つのいずれかを介して、クラスターがソースから継続的に供給されることを意味します。

詳細については。

4.6.3 空のクラスターをブートストラップする (initdb)

initdb ブートストラップ メソッドは、新しい PostgreSQL クラスターを最初から作成するために使用されます。特に指定がない限り、これがデフォルトです。

次の例には、initdb 構成の完全な構造が含まれています。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-initdb
spec:
  instances: 3

  superuserSecret:
    name: superuser-secret

  bootstrap:
    initdb:
      database: app
      owner: app
      secret:
        name: app-secret

  storage:
    size: 1Gi
```

上記のブートストラップの例は次のようになります。

1. PostgreSQL のネイティブ initdb コマンドを使用して、新しい PGDATA フォルダーを作成します。
2. superuser-secret という名前のシークレットから postgres **スーパーユーザー** のパスワードを設定します
3. app という名前の**非特権**ユーザーを作成します
4. app-secret シークレットのパスワードを使用して、後者 (app) のパスワードを設定します (username が owner の同じ名前と一致することを確認してください)。
5. app ユーザーが所有する app というデータベースを作成します。

構成パラダイム上の慣習のおかげで、オペレーターはデフォルトのデータベース名 (app) とデフォルトのアプリケーション ユーザー名 (データベース名と同じ) を選択できるほか、スーパーユーザーとユーザーの両方の安全なパスワードをランダムに生成できます。PostgreSQL のアプリケーション ユーザー。

または、上記の例で説明したように、パスワードを生成してシークレットとして保存し、PostgreSQL クラスターで使用することもできます。

提供されるシークレットは、kubernetes.io/basic-auth の仕様に準拠する必要があります。その結果、シークレットの username は、スーパーユーザー シークレットの owner (アプリケーションシークレット用) および postgres の 1 つと一致する必要があります。

以下は、basic-auth シークレットの例です。

```
apiVersion: v1
data:
  username: YXBw
  password: cGFzc3dvcmQ=
kind: Secret
metadata:
  name: app-secret
type: kubernetes.io/basic-auth
```

アプリケーション データベースは、アプリケーションデータの格納に使用する必要があるデータベースです。アプリケーションは、アプリケーションデータベースを所有するユーザーでクラスターに接続する必要があります。

重要: オペレーターの将来の実装では、宣言的な構成方法で追加のユーザーを作成できるようになる可能性があります。

---

postgres スーパーユーザーと postgres データベースは、クラスターを構成するオペレーターのみが使用することになっています。

データベース名を指定しない場合、オペレーターは慣例に従って処理を進め、app データベースを作成し、それを *defaulting webhook* を使用してクラスター定義に追加します。データベースを所有するユーザーは、代わりにデータベース名にデフォルト設定されます。

アプリケーションユーザーは、オペレーターによって内部的に使用されることはありません。代わりに、スーパーユーザーに依存してクラスターを目的の状態に調整します。

重要: 今のところ、スーパーユーザー シークレットの名前の変更はクラスターに適用されません。

実際の PostgreSQL データ ディレクトリは、initdb PostgreSQL コマンドの呼び出しによって作成されます。そのコマンドにカスタムオプションを追加する必要がある場合 (つまり、テンプレートデータベースに使用される locale を変更したり、データチェックサムを追加したりする場合)、次のパラメーターを使用できます。

dataChecksums : dataChecksums が true に設定されている場合、CNPG は initdb の -k オプションを呼び出して、データページのチェックサムを有効にし、I/O システムによる破損の検出を支援します。

encoding : encoding が値に設定されると、CNPG はそれを initdb の--encoding オプションに渡します。これにより、テンプレートデータベースのエンコーディングが選択されます (デフォルト: UTF8)。

localeCollate : localeCollate が値に設定されると、CNPG はそれを initdb の --lc-collate オプションに渡します。このオプションは、 Locale Support で定義されているように、照合順序 (LC_COLLATE サブカテゴリ) を制御します。

PostgreSQL ドキュメントから (デフォルト: C)。

localeCType : localeCType が値に設定されると、CNPG はそれを initdb の --lc-ctype オプションに渡します。このオプションは、Locale Support で定義されているように、照合順序 (LC_CTYPE サブカテゴリ) を制御します。

PostgreSQL ドキュメントから (デフォルト: C)。

walSegmentSize : walSegmentSize が値に設定されると、CNPG はそれを initdb の --wal-segsize オプションに渡します (デフォルト: 未設定- PostgreSQL によって 16 メガバイトとして定義されています)。

注釈: CloudNativePG が *initdb* ブートストラップ中に実装する 2 つのロケール オプションのみが、*LC_COLLATE* および *LC_TYPE* サブカテゴリを参照します。残りのロケール サブカテゴリは、*lc_messages* 、*lc_monetary* 、*lc_numeric* 、および *lc_time* パラメーターを使用して、PostgreSQL 構成で直接構成できます。

次の例では、データチェックサムを有効にし、デフォルトのエンコーディングを LATIN1 に設定します。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-initdb
spec:
  instances: 3

  bootstrap:
    initdb:
      database: app
      owner: app
      dataChecksums: true
      encoding: LATIN1
  storage:
    size: 1Gi
```

CloudNativePG は、options サブセクションを使用して、initdb 呼び出しの動作をカスタマイズする別の方法をサポートしています。ただし、オペレーターの動作を壊す可能性のあるオプション (--auth や -d など) がある場合、この手法は推奨されておらず、API の将来のバージョンから削除される予定です。

データベースが作成および構成された直後に一度だけ実行されるクエリのカスタムリストを指定することもできます。これらのクエリは、postgres データベースに接続された **スーパーユーザー** (postgres) として実行されます。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-initdb
spec:
```

(次のページに続く)

(前のページからの続き)

```
  instances: 3

  bootstrap:
    initdb:
      database: app
      owner: app
      dataChecksums: true
      localeCollate: en_US
      localeCType: en_US
      postInitSQL:
        - CREATE ROLE angus
        - CREATE ROLE malcolm
  storage:
    size: 1Gi
```

警告: *postInitSQL* 、*postInitApplicationSQL* 、および *postInitTemplateSQL* オプションは細心の注意を払って使用してください。クエリはスーパーユーザーとして実行され、クラスター全体を混乱させる可能性があります。これらのクエリのいずれかでエラーが発生すると、ブートストラップ フェーズが中断され、クラスターが不完全なままになります。

さらに、データベースの作成および構成後に実行される SQL スクリプトを含む Secret および/または ConfigMap のリストを指定できます。これらの SQL スクリプトは、`initdb` セクションで指定されたデータベースに接続された **superuser** ロール (`postgres`) を使用して実行されます。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-initdb
spec:
  instances: 3

  bootstrap:
    initdb:
      database: app
      owner: app
      postInitApplicationSQLRefs:
        secretRefs:
        - name: my-secret
          key: secret.sql
        configMapRefs:
        - name: my-configmap
          key: configmap.sql
  storage:
```

(次のページに続く)

(前のページからの続き)

```
    size: 1Gi
```

注釈: *secretRefs* で参照されている SQL スクリプトは、*configMapRefs* で参照されているものの前に実行されます。どちらのセクションでも、 SQL スクリプトはリストの順序に従って実行されます。 SQL スクリプト内では、各 SQL ステートメントは PostgreSQL semantics に従ってサーバー上の単一の exec で実行されます。コメントを含めることはできますが、*psql* のような内部コマンドは含めることができません。

警告: *postInitApplicationSQLRefs* で指定された ConfigMaps または Secrets 内にエントリが存在することを確認してください。存在しない場合、ブートストラップは失敗します。これらの SQL ファイルにエラーがあると、ブートストラップ フェーズが正常に完了しません。

4.6.4 別のクラスターからのブートストラップ

CloudNativePG は、同じメジャーバージョンの別のクラスターから開始するクラスターのブートストラップを有効にします。この操作は、ストリーミングレプリケーション (`pg_basebackup`) を介してソース クラスターに直接接続するか、**回復オブジェクト ストア** (`recovery`) を介して間接的に接続することで発生します。

ソース クラスターは、`name` で識別される `externalClusters` セクションで定義する必要があります (元のクラスターと同じ `name` を使用することをお勧めします)。

重要: デフォルトでは、*recovery* メソッドは *externalClusters* セクションのクラスターの *name* を厳密に使用して、通常はサーバーの名前用に予約されているオブジェクト ストア内のバックアップ データのメイン フォルダーを見つけます。 *barmanObjectStore.serverName* プロパティで別のものを指定できます (デフォルトでは、外部クラスター定義で *name* の値に割り当てられます)。

バックアップからのブートストラップ (recovery)

`recovery` ブートストラップモードでは、既存のバックアップから新しいクラスターを作成できます。つまり、**回復オブジェクトストア**です。

CloudNativePG でこの結果を達成するには、次の 2 つの方法があります。

- Barman Cloud によって作成され、`externalClusters` セクションの `barmanObjectStore` オプションで定義された別のクラスターのバックアップである回復オブジェクトストアを使用する (**推奨**)
- 同じ名前空間で既存の `Backup` オブジェクトを使用する (バージョン 1.8.0 より前では、これが唯一のオプションでした)。

どちらの回復方法でも、完全な回復 (利用可能な最後の WAL まで) または
ポイントインタイムリカバリ *(PITR)* までのいずれかが可能です。フル

リカバリを実行する場合、クラスタをレプリカモードで起動することもできます。また、復元されたクラスターの PostgreSQL 構成 (`.spec.postgresql.parameters`) が、物理レプリケーションの観点から、元の構成と互換性があることを確認してください。

注釈: バックアップとリカバリ で、実行中のクラスターのバックアップと復元に関する詳細情報を見つけることができます。

オブジェクト ストアからの復元

Barman Cloud によって作成され、サポートされているオブジェクトストレージに保存されているバックアップから復元できます。`barmanObjectStore` セクションに必要なすべての構成を含め、外部クラスターを定義したら、`.spec.recovery.source` オプションでそれを参照する必要があります。次の例では、Azure の BLOB コンテナーに回復オブジェクト ストアを定義します。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-restore
spec:
  [...]

  superuserSecret:
    name: superuser-secret

  bootstrap:
    recovery:
      source: clusterBackup

  externalClusters:
    - name: clusterBackup
      barmanObjectStore:
        destinationPath: https://STORAGEACCOUNTNAME.blob.core.windows.net/
↪CONTAINERNAME/
        azureCredentials:
          storageAccount:
            name: recovery-object-store-secret
            key: storage_account_name
          storageKey:
            name: recovery-object-store-secret
            key: storage_account_key
        wal:
          maxParallel: 8
```

重要: デフォルトでは、*recovery* メソッドは *externalClusters* セクションのクラスターの *name* を厳密に使用して、通常はサーバーの名前用に予約されているオブジェクト ストア内のバックアップ データのメインフォルダーを見つけます。 *barmanObjectStore.serverName* プロパティで別のものを指定できます (デフォルトでは、外部クラスター定義で *name* の値に割り当てられます)。

注釈: 上記の例では、並列 WAL 復元機能を利用して、必要な WAL ファイルをアーカイブから同時に取得するために最大 8 つのジョブを専用にしています。この機能により、回復時間を大幅に短縮できます。このシナリオを前もって計画し、環境に合わせてこのパラメーターの値を正しく調整してください。それは確かに違いを生みます **いつ** (場合ではなく) あなたがそれを必要とするでしょう.

Backup オブジェクトからの復元

クラスターを作成する名前空間でバックアップリソースが既に利用可能な場合は、次の例のように、`.spec.bootstrap.recovery.backup.name` でその名前を指定できます。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-initdb
spec:
  instances: 3

  superuserSecret:
    name: superuser-secret

  bootstrap:
    recovery:
      backup:
        name: backup-example

  storage:
    size: 1Gi
```

このブートストラップ方式では、復元する必要があるバックアップへの参照のみを指定できます。

その他の考慮事項

回復オブジェクト ストアまたは既存の Backup リソースから回復するかどうかにかかわらず、次の考慮事項が適用されます。

- アプリケーション データベース名とアプリケーション データベースユーザーは、復元中のバックアップから保持されます。これは Kubernetes クラスター自体の通常のメンテナンスアクティビティの一部であるため、オペレーターは現在、基になるシークレットのバックアップを試行していません。
- superuserSecret を指定しない場合、安全でランダムなパスワードを使用して新しいものが自動的に生成されます。次に、シークレットを使用して、クラスターの postgres ユーザーのパスワードをリセットします。
- デフォルトでは、リカバリはデフォルトのターゲットタイムラインで利用可能な最新の WAL まで続行されます (11 までの PostgreSQL の場合は current 、バージョン 12 以降の場合は latest)。オプションで recoveryTarget を指定して、ポイントイン タイム リカバリを実行できます (**ポイントインタイムリカバリ** *(PITR)* を参照してください)。

重要: *barmanObjectStore.wal.maxParallel* オプションを使用して、リカバリ オブジェクト ストアからトランザクション ログを同時にダウンロードすることで、アーカイブからの WAL フェッチを高速化することを検討してください。

ポイントインタイムリカバリ (PITR)

最新のものまですべての WAL を再生する代わりに、基本バックアップを抽出した後、任意の時点で WAL の再生を停止するよう PostgreSQL に要求できます。 PostgreSQL はこの手法を使用して、**ポイントインタイム** リカバリ (PITR) を実現します。

注釈: PITR は、リカバリ オブジェクト ストアと *Backup* オブジェクトから利用できます。

オペレーターは、Azure に格納された回復オブジェクトとタイムスタンプベースの目標を使用する次の例のように、回復ターゲットが指定されている場合に、この機能が動作するために必要な構成パラメーターを生成します。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-restore-pitr
spec:
  instances: 3

  storage:
    size: 5Gi

```

(次のページに続く)

(前のページからの続き)

```
  bootstrap:
    recovery:
      source: clusterBackup
      recoveryTarget:
        targetTime: "2020-11-26 15:22:00.00000+00"

  externalClusters:
    - name: clusterBackup
      barmanObjectStore:
        destinationPath: https://STORAGEACCOUNTNAME.blob.core.windows.net/
↪CONTAINERNAME/
        azureCredentials:
          storageAccount:
            name: recovery-object-store-secret
            key: storage_account_name
          storageKey:
            name: recovery-object-store-secret
            key: storage_account_key
        wal:
          maxParallel: 8
```

上記の例では、タイムスタンプの形式で `targetTime` を指定するだけで済み、復元を開始するベースバックアップの指定について心配する必要がないことに気付いたかもしれません。

`backupID` オプションは、リカバリ プロセスを開始するベースバックアップを指定できるオプションです。デフォルトでは、この値は空です。

値を (Barman バックアップ ID の形式で) 割り当てると、オペレーターはそのバックアップを復旧のベースとして使用します。

重要: そのようなバックアップが存在し、アクセス可能であることを確認する必要があります。

バックアップ ID が指定されていない場合、オペレータは次のようにリカバリのベースバックアップを自動的に検出します。

- `targetTime` または `targetLSN` を使用すると、オペレータはそのターゲットの前に完了した最も近いバックアップを選択します
- それ以外の場合、オペレータは利用可能な最後のバックアップを時系列で選択します。

使用できる回復目標基準は次のとおりです。

targetTime : RFC 3339 形式で表される、リカバリが進行するまでのタイム スタンプ (正確な停止ポイントは、`exclusive` オプションの影響も受けます)

targetXID : リカバリが続行されるまでのトランザクション ID (正確な停止ポイントは、`exclusive` オプションの影響も受けます); トランザクション ID はトランザクションの開始時に順番に割り当てられますが、トランザクションは異なる番号順で完了する可能性があることに注意してください。回復されるトランザク

ションは、指定されたトランザクションよりも前にコミットされたトランザクションです (オプションで指定されたトランザクションも含まれます)。

targetName : 復旧が進む名前付きの復元ポイント (pg_create_restore_point() で作成)

targetLSN : リカバリが続行されるログ先行書き込みの場所の LSN (正確な停止ポイントは exclusive オプションの影響も受けます)

targetImmediate : 一貫した状態に達したらすぐに、つまりできるだけ早く復旧を終了する必要があります。オンラインバックアップから復元する場合、これはバックアップの作成が終了した時点を意味します。

重要: *targetTime* または *targetLSN* のいずれかが指定されている場合、オペレーターは最も近いバックアップを自動的に取得できますが、これは残りのターゲット (*targetName* 、*targetXID* 、および *targetImmediate*) では不可能です。このような場合、カタログで利用可能な最後のバックアップで問題がない場合を除き、*backupID* を指定することが重要です。

以下の例では、targetName ベースのリカバリターゲットを使用しています。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
  bootstrap:
    recovery:
      source: clusterBackup
      recoveryTarget:
        backupID: 20220616T142236
        targetName: restore_point_1
[...]
```

各 recoveryTarget 構成で、上記のターゲットから 1 つだけ選択できます。

さらに、targetTLI 強制復旧を特定のタイムラインに指定できます。

デフォルトでは、以前のパラメーターは排他的であると見なされ、回復ターゲットの直前で停止します。Azure の BLOB コンテナーに依存する次の例のように、回復ターゲットの直後に停止し、exclusive パラメーターを false に設定して、包括的な動作を要求できます。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-restore-pitr
spec:
  instances: 3

  storage:
    size: 5Gi

  bootstrap:
```

(次のページに続く)

(前のページからの続き)

```
    recovery:
      source: clusterBackup
      recoveryTarget:
        backupID: 20220616T142236
        targetName: "maintenance-activity"
        exclusive: false

  externalClusters:
    - name: clusterBackup
      barmanObjectStore:
        destinationPath: https://STORAGEACCOUNTNAME.blob.core.windows.net/
↪CONTAINERNAME/
        azureCredentials:
          storageAccount:
            name: recovery-object-store-secret
            key: storage_account_name
          storageKey:
            name: recovery-object-store-secret
            key: storage_account_key
        wal:
          maxParallel: 8
```

アプリケーション データベースを構成する

復元されたクラスターの場合、追加の構成を使用して、アプリケーションデータベース名と資格情報を構成できます。アプリケーションデータベースの資格情報を更新するには、独自のパスワードを生成してシークレットとして保存し、シークレットを使用してデータベースを更新します。または、オペレーターがランダムに安全なパスワードを使用してシークレットを生成できるようにすることもできます。

> 空のクラスターをブートストラップする (`initdb )<空のクラスターをブートストラップする (*initdb* )>` を参照してください

シークレットの詳細については、セクションを参照してください。

次の例では、アプリケーション データベース app を所有者 app で構成し、シークレット app-secret を指定します。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  bootstrap:
    recovery:
      database: app
      owner: app
```

(次のページに続く)

(前のページからの続き)

```
    secret:
      name: app-secret
    [...]
```

上記の構成では、リカバリの完了後に次のことが行われます。

1. データベース app が存在しない場合、新しいデータベース app が作成されます。
2. ユーザー app が存在しない場合、新しいユーザー app が作成されます。
3. ユーザー app がデータベースの所有者でない場合、ユーザー app はデータベース app の所有者として付与されます。
4. username の値が owner の値と秘密裏に一致する場合、アプリケーションデータベースのパスワードは秘密裏に password の値に変更されます。

重要: レプリカ モードが有効になっているレプリカ クラスターの場合、元のクラスターから復元されるため、オペレーターは PostgreSQL インスタンスにデータベースまたはユーザーを作成しません。

ライブ クラスターからのブートストラップ (pg_basebackup)

pg_basebackup ブートストラップ モードでは、有効な**ストリーミングレプリケーション** 接続を介して、既存の **バイナリ互換** PostgreSQL インスタンス (**ソース**) の正確な物理コピーとして、新しいクラスター (**ターゲット**) を作成できます。ソース インスタンスは、プライマリ PostgreSQL サーバーまたはスタンバイ PostgreSQL サーバーのいずれかです。

この方法の主な使用例は、Kubernetes の外部または Kubernetes 内 (別のオペレーターなど) からの CloudNativePG への **移行** です。

> **警告:** 現在の実装では、クローン作成プロセスが終了すると、元の PostgreSQL インスタンスの **スナップショット** が作成され、作成されたクラスターがすぐに開始されます。詳細については、以下の 現在の制限事項 を参照してください。

recovery ブートストラップメソッドの場合と同様に、クローン操作が完了すると、オペレーターはターゲットクラスターの所有権を最初のインスタンスから取得します。これには、CloudNativePG で必要な一部の構成パラメーターのオーバーライド、スーパーユーザーパスワードのリセット、streaming_replica ユーザーの作成、レプリカの管理などが含まれます。結果のクラスターは、ソースインスタンスから完全に独立しています。

重要: ターゲット インスタンスとソース インスタンス間のネットワークの構成は、実際のコンテキストと環境に依存するため、CloudNativePG ドキュメントの範囲を超えています。

---

pg_basebackup によって透過的に管理されるターゲットインスタンス上のストリーミングレプリケーションクライアントは、次のいずれかの方法でソースインスタンス上で自身を認証できます。

1. ユーザー名/パスワード認証 経由
2. TLS client certificate 経由

CloudNativePG によって管理されているか、TLS 認証用に構成されているソースに接続する場合は、後者が推奨されます。ただし、最初のオプションは、一般的に PostgreSQL サーバーに対する認証の最も一般的な形式であり、ソースインスタンスが Kubernetes 以外の従来の環境にある場合は、最も簡単な方法かもしれません。両方のケースについて以下で説明します。

要件

次の要件が pg_basebackup ブートストラップ メソッドに適用されます。

- ターゲットとソースは同じハードウェアアーキテクチャを持つ必要があります
- ターゲットとソースの PostgreSQL のメジャーバージョンが同じである必要があります
- **ソースにはテーブルスペースが定義されていてはなりません (以下の 現在の制限事項 を参照)**
- ソースは、バックアップ用に少なくとも 1 つの *walsender* と WAL ストリーミング用に 1 つを提供することにより、この 1 回限りの操作のためにターゲットからのアクセスを許可するのに十分な max_wal_senders で構成する必要があります。
- ターゲット インスタンスがソース インスタンスの PostgreSQL ポートに接続できるように、ソースとターゲット間のネットワークを構成する必要があります。
- ソースには REPLICATION LOGIN 特権を持つ役割が必要であり、できれば TLS 経由で、pg_hba.conf でこの役割のターゲットインスタンスからの接続を受け入れる必要があります (以下の 複製ユーザーについて を参照)。
- ターゲットは、REPLICATION LOGIN 権限を持つロールを使用してソース PostgreSQL インスタンスに正常に接続できる必要があります

参考:

詳細については、 Planning 、 ライブ クラスターからのブートストラップ (`pg_basebackup )<ライブ クラスターからのブートストラップ (*pg_basebackup* )>` を参照してください。

および High Availability, Load Balancing, and Replication

PostgreSQL のドキュメントに記載されています。

複製ユーザーについて

要件セクションで説明したように、ソース インスタンスで SUPERUSER 権限、またはできれば REPLICATION 権限のみを持つユーザーが必要です。

ソース データベースが CloudNativePG で作成されている場合、streaming_replica ユーザーを再利用して、クライアント TLS 証明書認証を利用できます (デフォルトでは、streaming_replica で許可されている唯一の接続方法です)。

Kubernetes 以外の場合を含め、他のすべてのケースでは、REPLICATION 権限を持つユーザーが既に存在することを確認するか、以下の手順に従って新しいユーザーを作成してください。

ソース システムの postgres ユーザーとして、次を実行してください。

```
createuser -P --replication streaming_replica
```

プロンプトでパスワードを入力し、後で使用できるように保存します。これは、ターゲットインスタンスのシークレットに追加する必要があるためです。

注釈: 名前は重要ではありませんが、簡単にするために *streaming_replica* を使用します。次のセクションの手順を変更する場合は、自由に変更してください。

ユーザー名/パスワード認証

CloudNativePG が pg_basebackup ブートストラップでサポートする最初の認証方法は、ユーザー名とパスワードの照合に基づいています。

手順を開始する前に、次の情報があることを確認してください。

- ホスト名または IP アドレスと TCP ポートで識別されるソースインスタンスの場所
- レプリケーション ユーザー名 (簡単にするために streaming_replica)
- パスワード

ソース PostgreSQL インスタンスの pg_hba.conf ファイルに、次のような行を追加する必要がある場合があります。

```
#  A more restrictive rule for TLS and IP of origin is recommended

host replication streaming_replica all md5
```

次のマニフェストは、pg_basebackup ブートストラップメソッドを使用して target-db という名前の新しい PostgreSQL 15.2 クラスターを作成し、source-db (externalClusters 配列内) として定義された外部 PostgreSQL クラスターのクローンを作成します。ご覧のとおり、source-db 定義は source-db.foo.com ホストを指し、streaming_replica ユーザーとして接続します。このユーザーのパスワードは source-db-replica-user シークレットの password キーに保存されます。

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: target-db
spec:
  instances: 3
  imageName: ghcr.io/cloudnative-pg/postgresql:15.2

  bootstrap:
    pg_basebackup:
      source: source-db

  storage:
    size: 1Gi

  externalClusters:
  - name: source-db
    connectionParameters:
      host: source-db.foo.com
      user: streaming_replica
    password:
      name: source-db-replica-user
      key: password
```

複製操作が機能するには、同じ PostgreSQL バージョン (この場合は 15.2) を含め、すべての要件を満たす必要があります。

TLS 証明書認証

pg_basebackup ブートストラップを使用して CloudNativePG でサポートされている 2 番目の認証方法は、TLS クライアント証明書に基づいています。これは、セキュリティの観点から推奨されるアプローチです。

次の例では、同じ Kubernetes クラスター内の既存の PostgreSQL クラスター (cluster-example) を複製します。

注釈: この例は、Kubernetes クラスターの外部に存在するインスタンスをカバーするように簡単に適応させることができます。

マニフェストは、cluster-clone-tls と呼ばれる新しい PostgreSQL 15.2 クラスターを定義します。これは、cluster-example 外部クラスターから pg_basebackup メソッドを使用してブートストラップされます。ホストは同じクラスター内の読み取り/書き込みサービスによって識別されますが、streaming_replica ユーザーは、提供されたキー、証明書、および認証局情報 (それぞれ cluster-example-replication および cluster-example-ca シークレット内) のおかげで認証されます。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-clone-tls
spec:
  instances: 3
  imageName: ghcr.io/cloudnative-pg/postgresql:15.2

  bootstrap:
    pg_basebackup:
      source: cluster-example

  storage:
    size: 1Gi

  externalClusters:
  - name: cluster-example
    connectionParameters:
      host: cluster-example-rw.default.svc
      user: streaming_replica
      sslmode: verify-full
    sslKey:
      name: cluster-example-replication
      key: tls.key
    sslCert:
      name: cluster-example-replication
      key: tls.crt
    sslRootCert:
      name: cluster-example-ca
      key: ca.crt
``` #### アプリケーション データベースを構成する

`initdb` および `recovery` ブートストラップ メソッドの場合と同様に、ライブ クラスターからブートストラップするクラスターのアプリケーション データベースの構成もサポートしています。新しいクラスターがレプリカ クラスターとして作成された場合 (レプリカ モードが有効になっている場合)、アプリケーション データベースの構成はスキップされます。

次の例では、ライブ クラスターからのブートストラップ後に、提供されたシークレット `app-secret` のパスワードを使用してアプリケーション データベース `app` を構成します。

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
```

(次のページに続く)

(前のページからの続き)

```
bootstrap:
  pg_basebackup:
    database: app
    owner: app
    secret:
      name: app-secret
    source: cluster-example
```

上記の構成では、リカバリの完了後に次のことが行われます。

1. データベース app が存在しない場合、新しいデータベース app が作成されます。
2. ユーザー app が存在しない場合、新しいユーザー app が作成されます。
3. ユーザー app がデータベースの所有者でない場合、ユーザー app はデータベース app の所有者として付与されます。
4. username の値が owner の値と秘密裏に一致する場合、アプリケーションデータベースのパスワードは秘密裏に password の値に変更されます。

重要: レプリカ モードが有効になっているレプリカ クラスターの場合、元のクラスターから復元されるため、オペレーターは PostgreSQL インスタンスにデータベースまたはユーザーを作成しません。

現在の制限事項

表領域のサポートがありません

CloudNativePG には現在、PostgreSQL グローバルオブジェクト、つまりロール、データベース、およびテーブルスペースの完全な宣言型管理は含まれていません。ロールとデータベースはソースインスタンスからターゲットクラスターにコピーされますが、テーブルスペースには、このバージョンの CloudNativePG にはない機能、つまり追加の永続ボリュームの定義と管理が必要です。ベースバックアップとテーブルスペースを処理する場合、PostgreSQL 自体では、ソース インスタンスの正確なマウント ポイントがターゲット インスタンスにも存在する必要があります。この場合、CloudNativePG が管理する Kubernetes のポッドです。このため、テーブルスペースを利用する PostgreSQL インスタンスを CloudNativePG に直接移行することはできません (最初にテーブルスペースをソースから削除するか、組織がこの機能を必要とする場合は EDB に連絡して優先順位を付けてください)。

スナップショット コピー

pg_basebackup メソッドは、ソース インスタンスのスナップショットを PostgreSQL ベースバックアップの形式で取得します。バックアップの開始からバックアップの正しい終了までに書き込まれたすべてのトランザクションは、2 番目の接続を使用してターゲット インスタンスにストリーミングされます (pg_basebackup の --wal-method=stream オプションを参照してください)。

バックアップが完了すると、新しいインスタンスが新しいタイムラインで開始され、ソースから分岐します。このため、Kubernetes のターゲット データベースに移行する前に、ソースデータベースへのすべての書き込み操作を停止することをお勧めします。

重要: 移行を試みる前に、手順とアプリケーションの両方をテストする必要があります。特に、本番環境でのアプリケーションのダウンタイムを体系的に測定するために、必要に応じて何度でも移行手順を実行することが基本です。サポートが必要な場合は、EDB までお気軽にお問い合わせください。

4.7 Postgres データベースのインポート

このセクションでは、新しい CloudNativePG クラスター内に 1 つ以上の既存の PostgreSQL データベースをインポートする方法について説明します。

インポート操作は、PostgreSQL のオンライン論理バックアップの概念に基づいており、元のホストへのネットワーク接続を介した pg_dump と pg_restore に依存しています。ネイティブの Multi-Version Concurrency Control (MVCC) とスナップショットのおかげで、PostgreSQL は、書き込みアクティビティを停止することなく、ネットワーク経由で一貫したバックアップを同時に行うことができます。

論理バックアップは、PostgreSQL バージョンのメジャーアップグレードを実行するための最も一般的で、柔軟で信頼性の高い手法でもあります。

そのため、このセクションの手順は次の両方に適しています。

- Kubernetes の外部であっても、既存の PostgreSQL インスタンスから 1 つ以上のデータベースをインポートする
- データベースを任意の PostgreSQL バージョンから同じか新しいバージョンにインポートし、PostgreSQL の**メジャー アップグレード** を有効にします (たとえば、バージョン 11.x からバージョン 15.x へ)。

> **警告:** PostgreSQL のメジャー アップグレードを実行する場合、アプリケーションが新しいバージョンと互換性があり、データベースに含まれるオブジェクト (拡張機能を含む) のアップグレード パスが実行可能であることを確認する責任があります。

どちらの場合も、元のデータベースの一貫性のある **スナップショット**に対して操作が実行されます。

重要: このため、*Cluster* リソースでの最終インポートの前に、ソースでの書き込み操作を停止することをお勧めします。バックアップの開始後にソース データベースに加えられた変更は宛先クラスターに含まれ

ないため、この機能が参照される理由です。「オフライン インポート」または「オフライン メジャー アップグレード」として。

4.7.1 使い方

概念的には、インポートでは、空のクラスターをブートストラップする (`initdb )<空のクラスターをブートストラップする (*initdb* )>` を使用してゼロから新しいクラスター (**宛先クラスター**) を作成し、次に `initdb.import` サブセクションを完了して、既存の Postgres クラスター (**ソース クラスター**) からオブジェクトをインポートする必要があります。PostgreSQL の推奨に従って、**宛先クラスター** の PostgreSQL メジャーバージョンは、**ソース クラスター** のもの以上であることをお勧めします。

CloudNativePG には、ソースクラスターから宛先クラスターにオブジェクトをインポートする主な方法が 2 つあります。

- **マイクロサービスアプローチ**：CloudNativePG プロジェクトで推奨されているように、宛先クラスターは、指定されたアプリケーションユーザーが所有する単一のアプリケーションデータベースをホストするように設計されています。
- **モノリスアプローチ**：宛先クラスターは、ソースクラスターからインポートされた複数のデータベースとさまざまなユーザーをホストするように設計されています。

最初のインポート方法は `microservice` タイプで利用でき、後者は `monolith` タイプで利用できます。

> **警告:** スーパーユーザーまたは *pg_dump* で論理バックアップを作成するのに十分な権限を持つユーザーを使用して、宛先クラスターがソース クラスターにアクセスできることを確認するのは、ユーザーの責任です。 PostgreSQL documentation on "SQL Dump" を参照してください。

詳細については。

4.7.2 `microservice` タイプ

マイクロサービス アプローチを使用すると、ソースクラスターから宛先クラスターにインポートする単一のデータベースを指定できます。操作は次の 4 つのステップで実行されます。

- 新しいクラスターの `initdb` ブートストラップ
- `pg_dump -Fc` を使用して、選択したデータベース (`initdb.import.databases` 内) をエクスポートします。
- `initdb.owner` ユーザーが所有する `initdb.database` (アプリケーション データベース) への `pg_restore --no-acl --no-owner` を使用したデータベースのインポート
- データベース ダンプ ファイルのクリーンアップ
- `postImportApplicationSQL` パラメータを介したアプリケーションデータベースでのユーザー定義 SQL クエリのオプションの実行

- インポートされたデータベースでの ANALYZE VERBOSE の実行

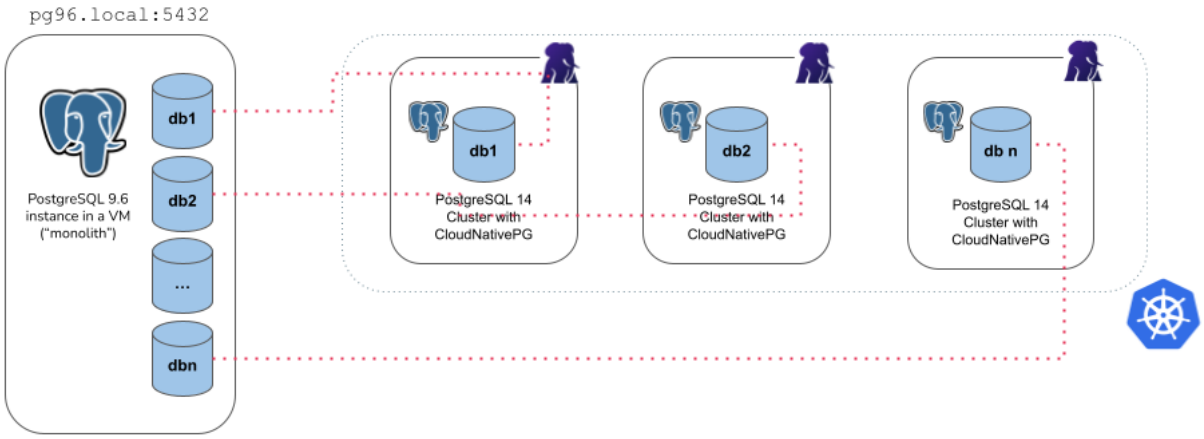


図 13: Example of microservice import type

たとえば、以下の YAML は、postgres に接続することにより、cluster-pg96 クラスター (サポートされていない PostgreSQL 9.6 を使用) から angus データベースをインポートする cluster-microservice という新しい 3 インスタンスの PostgreSQL クラスター (オペレーターがリリースされた時点で利用可能な最新のメジャー バージョン) を作成します。 cluster-pg96-superuser シークレットに保存されているパスワードを介して、postgres ユーザーを使用してデータベース。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-microservice
spec:
  instances: 3

  bootstrap:
    initdb:
      import:
        type: microservice
        databases:
          - angus
        source:
          externalCluster: cluster-pg96
        #postImportApplicationSQL:
        #- |
        #  INSERT YOUR SQL QUERIES HERE
  storage:
    size: 1Gi
  externalClusters:
    - name: cluster-pg96
      connectionParameters:
```

(次のページに続く)

(前のページからの続き)

```
    # Use the correct IP or host name for the source database
    host: pg96.local
    user: postgres
    dbname: postgres
  password:
    name: cluster-pg96-superuser
    key: password
```

警告: 上記の例では、コミュニティによってサポートされなくなったバージョンの PostgreSQL を実行しているソース データベースを意図的に使用しており、結果として CloudNativePG によってサポートされていません。ソース インスタンスからのデータ エクスポートは、宛先クラスター内の *pg_dump* のバージョンを使用して実行されます。これは、サポートされているものであり、ソース クラスター以上である必要があります。私たちの経験に基づいて、データをエクスポートするこの方法は、サポートされていない古いバージョンの Postgres でも機能するはずであり、Kubernetes 内のより良いシステムにレガシー データを移動する機会を与えてくれます。これが、このセクションの例で 9.6 を使用した主な理由です。この分野で問題が発生した場合は、ご連絡をお待ちしております。

`microservice` タイプを使用する際に注意する必要があることがいくつかあります。

- インポートするデータを含む既存の PostgreSQL インスタンスを指す `externalCluster` が必要です (詳細については、 :ref:``externalClusters` セクション<*externalClusters* セクション>` を参照してください)。
- 操作中、Kubernetes クラスターと `externalCluster` の間でトラフィックを許可する必要があります。
- ソース データベースへの接続は、`pg_dump` を実行してロール情報を読み取る必要がある指定されたユーザーに許可する必要があります (**スーパーユーザー** は OK)。
- 現在、`pg_dump -Fc` の結果は `PGDATA` ボリュームの `dumps` フォルダー内に一時的に保存されているため、割り当てられたノードのダンプ結果と復元されたデータとインデックスを一時的に格納するのに十分な空き容量が必要です。インポート操作が完了すると、このフォルダーはオペレーターによって自動的に削除されます。
- `initdb.import.databases` 配列内に指定できるデータベースは 1 つだけです
- ロールはインポートされません - そのため、`initdb.import.roles` 内で指定することはできません

4.7.3 monolith タイプ

モノリスアプローチでは、移行元クラスターから移行先クラスターにインポートする役割とデータベースのセットを指定できます。操作は次の手順で実行されます。

- 新しいクラスターの `initdb` ブートストラップ
- 選択した役割のエクスポートとインポート

- pg_dump -Fc を使用して、選択したデータベース (initdb.import.databases 内) を一度に 1 つずつエクスポートします。
- 選択した各データベースを作成し、pg_restore を使用してデータをインポートします
- インポートされた各データベースで ANALYZE を実行します
- データベース ダンプ ファイルのクリーンアップ

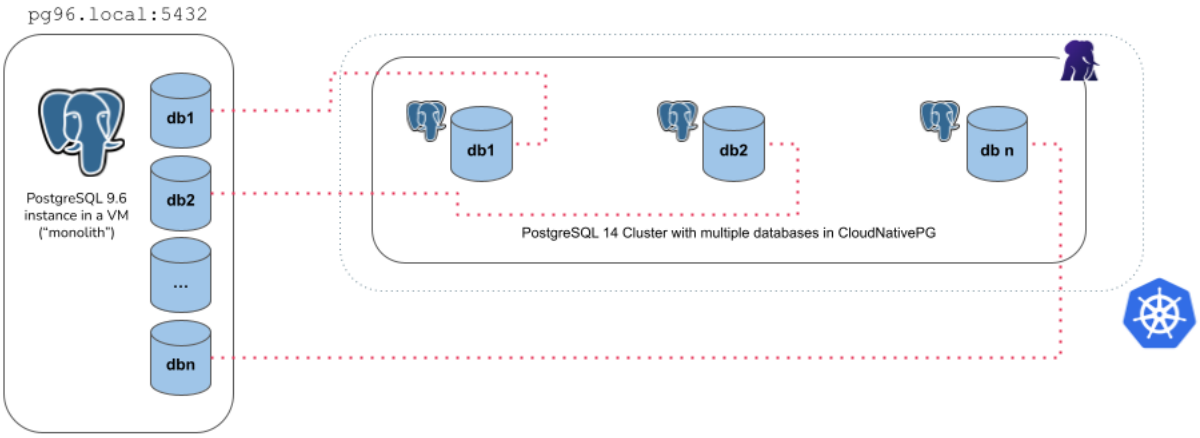


図 14: Example of monolith import type

たとえば、以下の YAML は、cluster-pg96 から accounting 、banking 、resort データベースだけでなく、accountant および bank_user ロールをインポートする cluster-monolith と呼ばれる新しい 3 インスタンス PostgreSQL クラスター (オペレーターがリリースされた時点で利用可能な最新のメジャー バージョン) を作成します。 cluster-pg96-superuser シークレットに保存されているパスワードを介して、postgres ユーザーを使用して postgres データベースに接続することにより、クラスター (サポートされていない PostgreSQL 9.6 を使用)。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-monolith
spec:
  instances: 3
  bootstrap:
    initdb:
      import:
        type: monolith
        databases:
          - accounting
          - banking
          - resort
        roles:
          - accountant
          - bank_user
```

(次のページに続く)

(前のページからの続き)

```
      source:
        externalCluster: cluster-pg96
  storage:
    size: 1Gi
  externalClusters:
    - name: cluster-pg96
      connectionParameters:
        # Use the correct IP or host name for the source database
        host: pg96.local
        user: postgres
        dbname: postgres
        sslmode: require
      password:
        name: cluster-pg96-superuser
        key: password
```

monolith タイプを使用する際に注意する必要があることがいくつかあります。

- インポートするデータを含む既存の PostgreSQL インスタンスを指す externalCluster が必要です (詳細については、 :ref:``externalClusters` セクション<*externalClusters* セクション>` を参照してください)。
- 操作中、Kubernetes クラスターと externalCluster の間でトラフィックを許可する必要があります。
- ソース データベースへの接続は、pg_dump を実行してロール情報を取得する必要がある指定されたユーザーに許可する必要があります (**スーパーユーザー** は OK)。
- 現在、pg_dump -Fc の結果は PGDATA ボリュームの dumps フォルダー内に一時的に保存されているため、割り当てられたノードのダンプ結果と復元されたデータとインデックスを一時的に格納するのに十分な空き容量が必要です。インポート操作が完了すると、このフォルダーはオペレーターによって自動的に削除されます。
- initdb.import.databases 配列で指定される少なくとも 1 つのデータベース
- インポートされたデータベースに必要な役割は、以下の制限付きで initdb.import.roles 内で指定する必要があります。
- ワイルドカード "*" を databases および/または roles 配列の唯一の要素として使用して、その種類のすべてのオブジェクトをインポートできます。データベースに一致する場合、ワイルドカードは postgres データベース、テンプレートデータベース、および接続を許可しないデータベースを無視します。
- クローン手順が完了した後、すべてのデータベースに対して ANALYZE VERBOSE が実行されます。
- postImportApplicationSQL フィールドはサポートされていません

4.8 安全

このセクションには、CloudNativePG のセキュリティに関する情報が含まれており、コード、コンテナー、およびクラスターの 3 つの異なるレイヤーで分析されます。

> **警告:** このページに含まれる情報は、Kubernetes クラスターでの通常の InfoSec 業務の実行を免除するものであってはなりません。 Overview of Cloud Native Security についてよく理解してください。

Kubernetes ドキュメントのページ。

EDB が CloudNativePG のセキュリティで採用したアプローチの理解と背景を理解するためのブログ記事。

4.8.1 コード

CloudNativePG のソース コードは、 **セキュリティの問題**を含む静的分析の目的で **体系的にスキャン** され、CI/CD パイプラインで直接

> GolangCI-Lint と呼ばれる Go 用の一般的なオープンソース

リンターを使用します。 GolangCI-Lint は、同じソース コードで複数の *linter* を実行できます。

これらの 1 つは Golang Security Checker 、または単に `gosec` です。これは、ハードコードされた資格情報などのコードに隠されている既知の脆弱性、脅威、および弱点の発見を目的とした一連のルールに対してソースの抽象構文ツリーをスキャンするリンターです。、整数オーバーフロー、 SQL インジェクションなどがあります。

重要: CI/CD パイプラインの静的コード分析フェーズでの失敗は、CloudNativePG の配信全体のブロッカーです。つまり、各コミットは GolangCI-Lint によって定義されたすべてのリンターに対して検証されます。

4.8.2 容器

CloudNativePG の一部であるすべてのコンテナーイメージは、すべてのコミットに続いて CI/CD パイプラインを介して自動的に構築されます。このようなイメージには、演算子だけでなくオペランドも含まれます。具体的には、サポートされているすべての PostgreSQL バージョンです。パイプライン内で、画像は次の方法でスキャンされます。

- Dockle : コンテナのビルド プロセスに関するベスト プラクティス

重要: すべてのオペランド イメージは、ベース イメージとパッケージ レベルでのセキュリティ更新の場合にパイプラインによって 1 日に 1 回自動的に再構築され、EDB が配布するコンテナー イメージに **パッチ レベルの更新**を提供します。

コンテナーレベルのセキュリティでは、次のガイドラインとフレームワークが考慮されています。

- **米国国防総省 (DoD) の国防情報システム局 (DISA) によって開発された**
  Container Image Creation and Deployment Guide 。
- Center for Internet Security (CIS) によって開発された CIS Benchmark for Docker

EDB が CloudNativePG のコンテナーレベルでセキュリティに対して採用したアプローチの詳細については、ブログ記事を参照してください。

4.8.3 集まる

クラスター レベルのセキュリティでは、コントロールプレーンとノードの両方を形成するすべての Kubernetes コンポーネントと、クラスターで実行されるアプリケーション (PostgreSQL を含む) が考慮されます。

役割ベースのアクセス制御 (RBAC)

オペレーターは、`cnpg-manager` という専用のサービスアカウントを使用して Kubernetes API サーバーと対話します。 Kubernetes では、これはデフォルトで `cnpg-system` 名前空間にインストールされ、このサービスアカウントとオペレーターに付与されるルール/リソース/動詞のセットを定義する `cnpg-manager` クラスター ロールとの間のクラスター ロールバインディングを使用します。

重要: 上記のアクセス許可は、オペレーターのサービス アカウントが Kubernetes API サーバーと対話するために排他的に予約されています。 *Cluster* 、*Pooler* 、*Backup* 、および *ScheduledBackup* リソースのみと対話するオペレーターのユーザーは、それらに直接アクセスできません。

以下に、いくつかの例と、最も重要なこととして、CloudNativePG が標準の Kubernetes 名前空間リソースの完全または部分的な管理を必要とする理由を示します。

`configmaps` : オペレーターは、Prometheus エクスポーター モニタリングメトリックのデフォルトの構成マップを作成および管理する必要があります。

`deployments` : オペレーターは、標準の Kubernetes `Deployment` リソースを使用して PgBouncer 接続プーラーを管理する必要があります。

`jobs` : オペレーターは、さまざまな `Cluster` のフェーズを管理するためにジョブを処理する必要があります。

`persistentvolumeclaims` : `PGDATA` が存在するボリュームは、PostgreSQL `Cluster` リソースの中心的な要素です。オペレーターは、定義されたスケジューリングポリシーに基づいて、要求されたボリュームを動的にプロビジョニングするために、選択したストレージクラスと対話する必要があります。

`pods` : オペレーターは `Cluster` のインスタンスを管理する必要があります。

`secrets` : `Cluster` オブジェクトに証明書とパスワードを提供しない限り、オペレーターは、ランダムに生成されたパスワードと TLS 証明書を自己プロビジョニングし、それらをシークレットに格納することにより、「構成よりも規則」パラダイムを採用します。

`serviceaccounts` : オペレーターは、インスタンス マネージャー (PostgreSQL サーバーを制御するコンテナーの *PID 1* プロセス) が Kubernetes API サーバーと安全に通信してアクションを調整し、信頼できるステータスを継続的に提供できるようにするサービスアカウントを作成する必要があります。 `Cluster` の。

`services` : オペレーターは、アプリケーションから PostgreSQL クラスター (または接続プーラー) へのネットワークアクセスを制御し、フェイルオーバー/スイッチオーバー操作を自動化された方法で適切に管理する必要があります (たとえば、サービスの正しいエンドポイントを適切なプライマリ PostgreSQL インスタンス)。

`validatingwebhookconfigurations` および `mutatingwebhookconfigurations` : オペレーターは自己署名 Webhook CA を両方の Webhook 構成に挿入します。これは、管理するすべてのリソースを検証および変更するために必要です。詳細については、 Kubernetes documentation を参照してください。

`nodes` : オペレーターは Affinity と AntiAffinity のラベルを取得する必要があるため、特にノードが異なるアベイラビリティーゾーンにある場合に、レプリカが同じノードに配置されるのを防ぐためにポッドをスケジュールできるノードを決定できます。この権限は、ノードがスケジュールされているかどうかを判断するためにも使用され、まったく作成できない Pod の作成を回避します。

オペレーターが必要とするすべての権限を確認するには、`kubectl describe clusterrole cnpg-manager` を実行します。

インスタンス マネージャーによる API サーバーへの呼び出し

オペランド コンテナーのエントリ ポイントであるインスタンスマネージャーは、一部のリソースの状態が正しく更新されていることを確認し、その Postgres クラスターに関連付けられている構成マップとシークレットにアクセスするために、Kubernetes API サーバーに対していくつかの呼び出しを行う必要があります。. このような呼び出しは、同じ PostgreSQL `Cluster` リソース名を共有するオペレーターによって作成された専用の `ServiceAccount` を介して実行されます。

重要: オペランドは、API サーバーを介して特定の制限されたリソースのサブセットにのみアクセスできます。サービス アカウントは recommended way to access the API server from within a Pod です。

透過性のために、サービス アカウントに関連付けられたアクセス許可は
roles.go で定義されます。

ファイル。たとえば、`myns` 名前空間の汎用 `mypg` クラスターのアクセス許可を取得するには、次のコマンドを入力できます。

```
kubectl get role -n myns mypg -o yaml
```

次に、ロールがサービス アカウントにバインドされていることを確認します。

```
kubectl get rolebinding -n myns mypg -o yaml
```

重要: 役割は特定の名前空間に限定されていることを思い出してください。

以下に、汎用 Kubernetes リソースのサービスアカウントに関連付けられているアクセス許可の概要を示します。

`configmaps` : インスタンス マネージャーは、カスタム モニタリングクエリなど、同じクラスターに関連する構成マップのみを読み取ることができます

`secrets` : インスタンスマネージャーは、同じクラスターに関連するシークレットのみを読み取ることができます。つまり、ストリーミングレプリケーションユーザー、アプリケーションユーザー、スーパー ユーザー、LDAP 認証ユーザー、クライアント CA、サーバー CA、サーバー証明書、バックアップ資格情報、カスタムモニタリング クエリです。

`events` : インスタンスマネージャーはクラスターのイベントを作成し、PostgreSQL インスタンスのライフサイクルの特定の側面について API サーバーに通知できます。

ここでは代わりに、CloudNativePG に固有のリソースについて同じ概要を提供します。

`clusters` : インスタンス マネージャーには、独自の `Cluster` リソースに対してのみ、読み取り専用のアクセス許可、つまり `get` 、`list` 、および `watch` が必要です。

`clusters/status` : インスタンス マネージャーは、`update` および `patch` に対して、独自の `Cluster` リソースのステータスのみを必要とします。

`backups` : インスタンス マネージャーは、名前空間内の `Backup` リソースを読み取るために `get` および `list` 権限を必要とします。さらに、オブジェクトストアに対応するオブジェクトがない `Backup` オブジェクトを削除して Kubernetes クラスターをクリーンアップするには、`delete` 権限が必要です。これは通常、保持ポリシーのためです。

`backups/status` : インスタンス マネージャーは、名前空間内の `Backup` リソースのステータスを `update` および `patch` に要求します。

ポッド セキュリティ ポリシー

Pod Security Policy

ポッドがクラスターで実行するために満たす必要があるセキュリティルールと仕様を定義する Kubernetes の方法です。 InfoSec の理由から、すべての Kubernetes プラットフォームはそれらを実装する必要があります。

CloudNativePG は、コンテナーの実行に **特権** モードを必要としません。PostgreSQL コンテナーは `postgres` システムユーザーとして実行されます。 `root` として実行する必要があるコンポーネントはありません。

同様に、ボリュームへのアクセスには *privileges* モードや `root` 権限も必要ありません。適切なアクセス許可は、Kubernetes プラットフォームおよび/または管理者によって適切に割り当てられる必要があります。PostgreSQL コンテナは、読み取り専用のルートファイルシステムで実行されます (つまり、書き込み可能なレイヤーはありません)。

オペレータは、必要なセキュリティ コンテキストを明示的に設定します。

AppArmor を使用して Pod アクセスを制限する

container.apparmor.security.beta.kubernetes.io アノテーションを使用して、すべての Cluster ポッド内の postgres 、initdb 、join 、full-recovery 、および bootstrap-controller コンテナーに *AppArmor を使用して Pod アクセスを制限する*プロファイルを割り当てることができます。

> **警告:** この種のアノテーションを使用すると、クラスターが機能しなくなる可能性があります。この場合、注釈は *Cluster* から安全に削除できます。

AppArmor 構成は Kubernetes ノードレベルである必要があります。つまり、基盤となるオペレーティングシステムでこのオプションを有効にし、適切に構成する必要があります。

これが状況ではなく、Cluster の作成時に注釈が追加された場合、ポッドは作成されません。一方、Cluster が作成された後に注釈を追加すると、クラスター内のポッドを開始できなくなり、次のようなエラーが発生します。

```
metadata.annotations[container.apparmor.security.beta.kubernetes.io/postgres]:␣
↪Forbidden: may not add AppArmor annotations]
```

このような場合は、Kubernetes 管理者に問い合わせて、適切な AppArmor プロファイルを使用するよう依頼してください。

ネットワーク ポリシー

Cluster リソースによって作成されたポッドは、Kubernetes
ネットワーク ポリシー によって制御できます。

IP および TCP レベルでのインバウンドおよびアウトバウンド ネットワークアクセスを有効/無効にします。

重要: オペレーターは、TCP ポート 8000 で各インスタンスと通信して、PostgreSQL サーバーのステータスに関する情報を取得する必要があります。ネットワーク ポリシーを追加する場合に備えて、この点に留意してください。詳細な制御のために CloudNativePG で使用されるポートのリストについては、以下の「公開ポート」セクションを参照してください。

ネットワーク ポリシーは、このドキュメントの範囲外です。 ネットワーク ポリシーを参照してください。

詳細については、Kubernetes ドキュメントのセクションを参照してください。

公開ポート

以下の表に示すように、CloudNativePG は、オペレーター、インスタンスマネージャー、およびオペランド レベルでポートを公開します。

システム | システム |ポート番号 |公開 |名前 |証明書 |認証: ———— | :———— | :———————— – | :———————— – | :———— | :———— – 演算子 | 9443 | Webhook サーバー | webhook-server | TLS |はいオペレーター| 8080 |メトリック | metrics | TLS なし |インスタンスマネージャーなし | 9187 |メトリック | metrics | TLS なし|インスタンス マネージャーなし | 8000 |ステータス | status | TLS なし |オペランドなし | 5432 | PostgreSQL インスタンス | postgresql |オプションの TLS |はい

PostgreSQL

CloudNativePG の現在の実装では、postgres スーパーユーザーとデータベース所有者のパスワードと .pgpass ファイルが自動的に作成されます。

パスワードの暗号化に関する限り、CloudNativePG は PostgreSQL のデフォルトの動作に従います。

重要: ユーザー名/パスワード認証 を参照してください

詳細については、PostgreSQL ドキュメントのセクションを参照してください。

enableSuperuserAccess を false に設定することで、シークレットによる postgres ユーザーパスワードの管理を無効にすることができます。

注釈: オペレーターは、*enableSuperuserAccess* オプションの切り替えをサポートしています。実行中のクラスターで無効にすると、オペレーターはシークレットの内容を無視し、それを削除し (オペレーターによって以前に生成された場合)、*postgres* ユーザーのパスワードを *NULL* に設定します (パスワード認証によるリモート アクセスを事実上無効にします)。

詳細については、 秘密 を参照してください。

これらのファイルを使用して、データベースへのアプリケーションアクセスを構成できます。

デフォルトでは、すべてのレプリカは、streaming_replica という特別なユーザーを使用して、**物理非同期ストリーミングレプリケーション** で現在のプライマリインスタンスに接続するように自動的に構成されます。ノード間の接続は**暗号化** され、認証は **TLS クライアント証明書** を介して行われます (詳細については、 クライアント *TLS/SSL* 接続 ページを参照してください)。

現在、オペレーターは、postgresql 構成の pg_hba セクションの一部として、管理者が pg_hba.conf 行をマニフェストに直接追加することを許可しています。マニフェストで定義された行は、デフォルトの pg_hba.conf に追加されます。

オペレーターによる pg_hba.conf の管理方法の詳細については、ドキュメントの *PostgreSQL* の設定を参照してください。

重要: 例では、Kubernetes クラスターがプライベートで安全なネットワークで実行されていることを前提としています。

4.9 Postgre インスタンス マネージャー

CloudNativePG は、フェイルオーバー管理のために外部ツールに依存しません。Kubernetes API サーバーと、 **Postgres インスタンスマネージャー**と呼ばれるネイティブ キー コンポーネントに依存するだけです。

インスタンス マネージャーは、PostgreSQL の先行プロセス (`postmaster` とも呼ばれます) のライフサイクル全体を処理します。

新しいクラスターを作成すると、オペレーターはインスタンスごとに Pod を作成します。フィールド `.spec.instances` は、作成するインスタンスの数を指定します。

各 Pod は、インスタンス マネージャーをメイン コンテナーの親プロセス (PID 1) として起動し、次に PostgreSQL インスタンスを実行します。 Pod の有効期間中、インスタンス マネージャーは liveness and readiness probes を処理するバックエンドとして機能します。

4.9.1 liveness と readiness プローブ

liveness プローブは `pg_isready` に依存しますが、readiness プローブは、データベースが稼働しており、スーパーユーザーの資格情報を使用して接続を受け入れることができるかどうかを確認します。Pod がトラフィックを受け入れる準備ができている場合、readiness プローブは肯定的です。 liveness プローブは、コンテナーを再起動するタイミングを制御します。

> プローブ コマンドが 10 秒間隔で 3 回失敗すると、2 つのプローブは失敗を報告します。

スタートアップ プローブは Kubernetes 1.17 でのみ導入されたため、現時点では、オペレーターは Pod で `startupProbe` を構成しません。

liveness プローブは、PostgreSQL インスタンスが壊れた状態にあり、再起動する必要があるかどうかを検出するために使用されます。`startDelay` の値は、プローブの実行を遅らせるために使用されます。これは、起動時間が長いインスタンスが再起動されるのを防ぐために使用されます。

Pod が起動してから liveness プローブが動作を開始するまでの秒数は、`.spec.startDelay` パラメータで表されます。デフォルトは 30 秒です。クラスタの正しい値は、PostgreSQL の起動に必要な時間に関連しています。

`.spec.startDelay` が低すぎると、PostgreSQL の起動前に liveness プローブが動作を開始し、Pod が不適切に再起動される可能性があります。

4.9.2 シャットダウン制御

Postgres を実行している Pod が削除されると、ノードのドレイン操作に続いて、手動または Kubernetes によって削除されます。kubelet はインスタンスマネージャーに終了信号を送信し、インスタンス マネージャーは適切な方法で PostgreSQL をシャットダウンします。秒単位で表される `.spec.stopDelay` は、PostgreSQL がシャットダウンするために与えられた時間です。値のデフォルトは 30 秒です。

シャットダウン手順は、次の 2 つの手順で構成されます。

1. インスタンス マネージャーは **スマート**シャットダウンを要求し、PostgreSQL への新しい接続を禁止します。このステップは、`.spec.stopDelay` で設定された時間の半分の間続きます。
2. PostgreSQL がまだ稼働している場合、インスタンス マネージャーは**高速**シャットダウンを要求し、既存の接続をすべて終了してすぐに終了します。インスタンスが WAL ファイルをアーカイブおよび/またはストリーミングしている場合、プロセスは`.spec.stopDelay` で設定された時間の残りの半分まで待機して操作を完了し、強制的にシャットダウンします。

重要: データベースの RPO に影響を与える Postgres クラスターでのデータ損失を回避するために、プライマリ インスタンスが実行されている Pod を削除しないでください。この場合、最初に別のインスタンスへのスイッチオーバーを実行します。

スイッチオーバー中のプライマリのシャットダウン

スイッチオーバー中のシャットダウン手順は、一般的な場合とは少し異なります。実際、オペレーターは、すべてのデータが新しいプライマリで利用できるようにするために、選択した新しいプライマリを昇格させる前に、元のプライマリに**高速** シャットダウンを発行するよう要求します。

このため、秒単位で表される `.spec.switchoverDelay` は、元のプライマリが正常にシャットダウンしてすべての WAL ファイルをアーカイブするために与えられる時間を制御します。この時間枠の間、プライマリインスタンスは接続を受け入れません。デフォルトの値は 1 年を超える秒数であり、無限の遅延をシミュレートしてデータの耐久性を維持するのに十分な大きさです。

警告: *.spec.switchoverDelay* オプションは、PostgreSQL データベースの RPO と RTO に影響します。低い値に設定すると、RPO よりも RTO が優先される可能性がありますが、クラスター レベルやバックアップ レベルでデータが失われる可能性があります。反対に、高い値に設定すると、スイッチオーバー中にクラスターがアクティブなプライマリなしで長時間放置されたままになるため、データ損失のリスクがなくなる可能性があります。

4.9.3 フェイルオーバー

プライマリ ポッドに障害が発生した場合、クラスターはフェイルオーバーモードになります。詳細は フェイルオーバー を参照してください。

4.10 スケジューリング

Kubernetes のスケジューリングは、いくつかの基準に基づいて、可能な限り最適なノードに新しいポッドを配置するプロセスです。

利用可能なすべてのポリシーを含むスケジューリングの詳細については、を参照してください。このページでは、アフィニティ、非アフィニティ、ノードセレクターなどの概念に精通していることを前提としています。

> *Pod* の親和性と非親和性 を使用して、CloudNativePG

クラスターのインスタンスをスケジュールする方法を制御できます。

以下をサポートするクラスターの定義のセクション。

- ポッドの親和性/反親和性
- ノードセレクター
- 寛容

4.10.1 Pod の親和性と非親和性

Kubernetes を使用すると、それらのノードですでに実行されている実際のワークロードに基づいて、ポッドをスケジュールする必要がある (**アフィニティ**) またはスケジュールしない (**アンチアフィニティ**) ノードを制御できます。これは技術的に**ポッド間アフィニティ/アンチアフィニティ** として知られています。

デフォルトでは、CloudNativePG はクラスターのインスタンスを異なるノード上に構成することが望ましいため、次の `affinity` 定義が生成されます。

```
affinity:
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
      - podAffinityTerm:
          labelSelector:
            matchExpressions:
              - key: postgresql
                operator: In
                values:
                  - cluster-example
          topologyKey: kubernetes.io/hostname
        weight: 100
```

次のクラスター仕様の結果として:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  imageName: ghcr.io/cloudnative-pg/postgresql:15.2

  affinity:
    enablePodAntiAffinity: true #default value
    topologyKey: kubernetes.io/hostname #defaul value
    podAntiAffinityType: preferred #default value

  storage:
    size: 1Gi
```

したがって、Kubernetes は、リソースが許す限り、3 つの異なるノードよりも 3 ノードの PostgreSQL クラスターをスケジュールすることを**優先**します。

前述のデフォルトの動作は、上記の設定を微調整することで変更できます。

`podAntiAffinityType` は `required` に設定できます。その結果、`preferredDuringSchedulingIgnoredDuringExecution` の代わりに `requiredDuringSchedulingIgnoredDuringExecution` が使用されます。このような強い要件により、リソースが利用できない場合に保留中のインスタンスが発生する可能性があることに注意してください (これは、Kubernetes クラスターの自動水平スケーリングに Cluster Autoscaler を使用する場合に予想される状態です)。 .

クラウド環境での `topologyKey` の別の可能な値は `topology.kubernetes.io/zone` であり、ポッドがノードだけでなくアベイラビリティゾーン全体に確実に分散されるようにします。 Well-Known Labels, Annotations and Taints を参照してください

より多くのオプションについて。

`enablePodAntiAffinity` を false に設定することで、オペレーターが生成した非アフィニティポリシーを無効にすることができます。

さらに、より細かい制御が必要な場合は、`additionalPodAffinity` および `additionalPodAntiAffinity` 構成属性を使用して、カスタム ポッドアフィニティまたは非アフィニティルールのリストを指定できます。これらのルールは、有効になっている場合はオペレーターによって生成されたルールに追加され、それ以外の場合は透過的に渡されます。

注釈: Pod 仕様で期待される *podAntiAffinity* または *podAffinity* のコンテンツ全体を *additionalPodAntiAffinity* または *additionalPodAffinity* に渡す必要があります (次の YAML を、すべてのワーカーノードで実行されている PostgreSQL のインスタンスを 1 つだけ持つ例として見てください。属しているクラスター)。

```
additionalPodAntiAffinity:
  requiredDuringSchedulingIgnoredDuringExecution:
  - labelSelector:
      matchExpressions:
      - key: postgresql
        operator: Exists
        values: []
    topologyKey: "kubernetes.io/hostname"
```

4.10.2 nodeSelector によるノード選択

Kubernetes では、nodeSelector がラベルのリスト (キーと値のペアとして定義) を提供して、ポッドを実行できるノードを選択できます。具体的には、ノードは、スケジュールされて実行されるポッドのラベルとして、示された各キーと値のペアを持っている必要があります。

同様に、CloudNativePG は、affinity セクションで nodeSelector を定義することに同意します。これにより、これらのラベルを持つノードでのみ PostgreSQL クラスターを実行するように要求できます。

4.10.3 許容範囲

Kubernetes では、(tolerations を介して) taints を明示的に許容していないすべてのポッドをノードが拒否するかどうかを (taints を介して) 指定できます。

そのため、特定のノードの taints に一致するワークロードに tolerations の適切なセットを設定することで、Kubernetes スケジューラは、ワークロードをスケジュールするノードを決定する際に、汚染されたノードを考慮に入れるようになります。許容は、許容のための通常の Kubernetes 構文を受け入れる .spec.affinity.tolerations セクションを介して、クラスターのすべてのポッドに対して構成できます。

4.11 資源管理

典型的な Kubernetes クラスターでは、ポッドは無制限のリソースで実行されます。デフォルトでは、必要なだけの CPU と RAM の使用が許可される場合があります。

CloudNativePG を使用すると、管理者はマニフェストの resources セクションを介して、次の 2 つのノブを使用して、クラスターのポッドによるリソースの使用を制御および管理できます。

- requests : 初期要件
- limits : リソースのニーズが動的に増加した場合の最大使用量

たとえば、次のように 32MiB の RAM (128MiB まで拡張可能) と 50m の CPU (100m まで拡張可能) の初期量を要求できます。

```
resources:
  requests:
    memory: "32Mi"
    cpu: "50m"
  limits:
    memory: "128Mi"
    cpu: "100m"
```

メモリーの要求と制限はコンテナーに関連付けられていますが、ポッドにはメモリーの要求と制限があると考えると便利です。ポッドのメモリリクエストは、ポッド内のすべてのコンテナのメモリ リクエストの合計です。

Pod のスケジューリングは、制限ではなくリクエストに基づいています。 Pod は、Pod のメモリー要求を満たすのに十分な使用可能なメモリーが Node にある場合にのみ、Node 上で実行するようにスケジュールされます。

リソースごとに、コンテナーを優先度の高い順に 3 つのサービス品質 (QoS) クラスに分けます。

- **保証済み**
- **バースト可能**
- **最大限の努力**

詳細については、 Configure Quality of Service for Pods を参照してください。

Kubernetes ドキュメントのセクション。

PostgreSQL ワークロードの場合、「保証された」QoS を設定することをお勧めします。

Kubernetes でのリソース関連の問題を回避するために、クラスター作成時の「リソース不足」処理のベストプラクティスを参照できます。

- マニフェスト ファイルのリソース セクションで、メモリと CPU に必要な値を指定します。このようにして、`OOM Killed` (「OOM」はメモリ不足を表します) や `CPU throttle` 、または実行中のインスタンスでのその他のリソース関連の問題を回避できます。
- クラスターのポッドが「保証された」QoS クラスに割り当てられるようにするには、メモリと CPU の両方の制限と要求を同じ値に設定する必要があります。
- 必要な PostgreSQL メモリ パラメーターをポッドリソースと一貫して指定します (VM または物理マシンのシナリオで行うように - 以下を参照)。
- nodeSelector を使用して専用ノードにデータベース サーバーポッドをセットアップします。 API リファレンス ページの “affinityconfiguration" リソースの「nodeSelector」および「tolerations」フィールドを参照してください。

次のマニフェストの例を参照できます。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
```

(次のページに続く)

(前のページからの続き)

```
  name: postgresql-resources
spec:

  instances: 3

  postgresql:
    parameters:
      shared_buffers: "256MB"

  resources:
    requests:
      memory: "1024Mi"
      cpu: 1
    limits:
      memory: "1024Mi"
      cpu: 1

  storage:
    size: 1Gi
```

上記の例では、256MB の値で shared_buffers パラメータを指定しました。つまり、データをキャッシュするために PostgreSQL サーバーに割り当てられるメモリの量です (このパラメータが定義されていない場合、このパラメータのデフォルト値は 128MB です)。

shared_buffers の適切な開始値は、システムのメモリの 25% です。例: shared_buffers が 256 MB の場合、コンテナーのメモリ サイズの推奨値は 1 GB です。これは、ポッド内のすべてのコンテナーに、Kubernetes が常に保持する合計 1 GB のメモリがあることを意味します。期待どおりに動作します。詳細については、Resource Consumption を参照してください。

PostgreSQL ドキュメントのセクション。

Kubernetes ドキュメントのページ。

4.12 失敗モード

このセクションでは、PostgreSQL がその存続期間中に Kubernetes クラスターで直面する可能性のある主要な障害シナリオの概要を説明します。

重要: 発生している障害シナリオがこのセクションでカバーされていない場合は、すぐに EDB に連絡してサポートと支援を受けてください。

詳細については、CloudNativePG によって実装された liveness および readiness プローブを参照してください。

4.12.1 ストレージ容量の使用

オペレーターは、PostgreSQL インスタンスごとに 1 つの PVC をインスタンス化して、PGDATA コンテンツを保存します。クラスターの初期化中に .spec.walStorage が指定された場合、WAL ストレージ専用の 2 つ目の PVC がプロビジョニングされます。

このようなストレージ スペースは、次の 2 つのケースで再利用できるように設定されています。

- 対応する Pod がユーザーによって削除されたとき (新しい Pod が再作成されます)
- 対応する Pod が削除され、別のノードでスケジュールされたとき

オペレーターが特定の PVC を再利用できないようにする場合は、Pod を削除する前に PVC を削除する必要があります。この目的のために、次のコマンドを使用できます。

```
kubectl delete -n [namespace] pvc/[cluster-name]-[serial] pod/[cluster-name]-[serial]
```

注釈: 専用の WAL ボリュームを指定した場合は、このプロセス中にそれも削除する必要があります。

```
kubectl delete -n [namespace] pvc/[cluster-name]-[serial] pvc/[cluster-name]-[serial]-
↪wal pod/[cluster-name]-[serial]
```

例えば：

```
$ kubectl delete -n default pvc/cluster-example-1 pvc/cluster-example-1-wal pod/
↪cluster-example-1
persistentvolumeclaim "cluster-example-1" deleted
persistentvolumeclaim "cluster-example-1-wal" deleted
pod "cluster-example-1" deleted
```

4.12.2 故障モード

Cluster に属するポッドは、次の方法で失敗する可能性があります。

- Pod はユーザーによって明示的に削除されます。
- postgres コンテナーの準備プローブが失敗します。
- postgres コンテナの liveness プローブが失敗します。
- Kubernetes ワーカーノードがドレインされます。
- ポッドがスケジュールされている Kubernetes ワーカーノードが失敗します。

これらの障害のそれぞれが、Cluster およびオペレーターによって管理されるサービスに異なる影響を与えます。

ユーザーによって削除された Pod

オペレーターに削除が通知されます。 `Cluster` に属する新しいポッドは、利用可能な場合は既存の PVC を再利用して自動的に作成されます。そうでない場合は *primary* の物理バックアップから開始します。

重要: Pod を意図的に削除した場合、*PodDisruptionBudget* ポリシーは適用されません。

---

apiserver に通知されるとすぐに自己修復が行われます。

準備プローブの失敗

3 回失敗すると、ポッドは **準備ができていない** と見なされます。 Pod は引き続き `Cluster` の一部であり、新しい Pod は作成されません。

失敗の原因を修正できない場合は、ポッドを手動で削除することができます。それ以外の場合、ポッドは障害が解決されたときに以前の役割を再開します。

プローブが 3 回失敗すると、自己修復が行われます。

liveness プローブの失敗

3 回失敗すると、`postgres` コンテナーは失敗したと見なされます。ポッドは引き続き `Cluster` の一部であり、*kubelet* はコンテナーを再起動しようとします。失敗の原因を修正できない場合は、ポッドを手動で削除することができます。

プローブが 3 回失敗すると、自己修復が行われます。

ワーカーノードが排出されました

ポッドはワーカーノードから削除され、サービスから削除されます。`nodeMaintenanceWindow` パラメーターの `reusePVC` オプションが `off` に設定されている場合、新しいポッドは *primary* の物理バックアップから別のワーカーノードに作成されます (デフォルト: メンテナンス ウィンドウ中は `on` 、それ以外の場合は `off`)。

`PodDisruptionBudget` は、準備ができていないポッドが少なくとも 1 つある場合、ポッドが削除されないようにすることができます。

注釈: 単一インスタンス クラスタは、*reusePVC* が *false* に設定されている場合、ノードのドレインを防ぎます。 Kubernetes Upgrade section を参照してください。

---

apiserver に通知されるとすぐに自己修復が行われます。

ワーカー ノードの障害

ノードに障害が発生しているため、*kubelet* は liveness プローブと readiness プローブを実行しません。ポッドは、その特定の障害原因に対して Kubernetes クラスター管理者によって構成された許容秒数が経過した後、削除対象としてマークされます。Kubernetes クラスターの構成方法に基づいて、ポッドが以前にサービスから削除される場合があります。

新しいポッドは、**プライマリ**の物理バックアップから別のワーカーノードに作成されます。 Kubernetes クラスターでのそのパラメーターのデフォルト値は 5 分です。

`tolerationSeconds` の後に自己修復が行われます。

4.12.3 自己修復

失敗したポッドがスタンバイの場合、ポッドは `-r` サービスと `-ro` サービスから削除されます。次に、利用可能な場合は PVC を使用して Pod が再起動されます。それ以外の場合は、現在のプライマリのバックアップから新しいポッドが作成されます。ポッドは、準備ができたら`-r` サービスと `-ro` サービスに再度追加されます。

失敗したポッドがプライマリである場合、オペレーターはステータスが準備完了でレプリケーションラグが最小のアクティブなポッドを昇格させ、`-rw` サービスをそれに向けます。失敗したポッドは、`-r` サービスと `-rw` サービスから削除されます。他のスタンバイは、新しいプライマリから複製を開始します。PVC が使用可能な場合、元のプライマリは `pg_rewind` を使用して新しいプライマリと同期します。それ以外の場合は、現在のプライマリのバックアップから新しいスタンバイが作成されます。

4.12.4 手動介入

文書化されていない障害の場合、手動で問題を解決するために介入する必要がある場合があります。

重要: このような場合、EDB エンジニアリング チームのサポートと支援なしに手動操作を実行しないでください。

オペレーターのバージョン 1.11.0 から、次のように、`cnpg.io/reconciliationLoop` アノテーションを使用して、選択した PostgreSQL クラスターで調整ループを一時的に無効にすることができます。

```
metadata:
  name: cluster-example-no-reconcile
  annotations:
    cnpg.io/reconciliationLoop: "disabled"
spec:
  # ...
```

`cnpg.io/reconciliationLoop` は、細心の注意を払って使用する必要があり、異常/緊急操作の期間中のみ使用する必要があります。

警告: この注釈は限られた期間のみ使用し、緊急事態が終了したら削除するようにしてください。このアノテーションをクラスターに残すと、オペレーターはフェイルオーバーなどの自己修復操作を実行できなくなります。

4.13 ローリング更新

オペレーターを使用すると、アプリケーションがクラスターに対して実行されているときに、クラスターで使用される PostgreSQL バージョンを変更できます。

重要: PostgreSQL マイナー リリースのアップグレードのみがサポートされています。

ローリング アップグレードは、次の場合に開始されます。

- ユーザーがクラスター仕様の `imageName` 属性を変更します。
- PostgreSQL 構成の変更には、再起動を適用する必要があります。
- `Cluster .spec.resources` 値の変更
- AKS での永続ボリューム要求のサイズの変更
- オペレーターが更新された後、Pod が最新のインスタンスマネージャーを実行するようにします (in-place updates are enabled を除く)。

オペレーターは、すべてのレプリカのアップグレードを一度に 1 つの Pod で開始し、シリアルが最も高いものから開始します。

プライマリは、アップグレードする最後のノードです。

ローリング更新は構成可能で、完全に自動化することも (`unsupervised`)、人間の介入を必要とすることもできます (`supervised`)。

アップグレードでは、データを再複製することなく、CloudNativePG の ID が保持されます。 Pod は削除され、必要に応じて同じ PVC と新しいイメージで再度作成されます。

ローリング更新手順中、各サービスエンドポイントはクラスタのステータスを反映するように移動するため、アプリケーションは更新中のノードを無視できます。

4.13.1 自動更新 (unsupervised)

`primaryUpdateStrategy` が `unsupervised` に設定されている場合、ローリング更新プロセスは Kubernetes によって管理され、完全に自動化されます。レプリカがアップグレードされると、選択した `primaryUpdateMethod` 操作がプライマリで開始されます。これがデフォルトの動作です。

`primaryUpdateMethod` オプションは、次の値のいずれかを受け入れます。

- `switchover` : スイッチオーバー操作が自動的に実行され、最も整列されたレプリカが新しいターゲットプライマリとして設定され、以前のプライマリポッドがシャットダウンされます (デフォルト)。

- `restart` : 可能であれば、プライマリインスタンスが実行されているポッドの自動再起動を実行します。それ以外の場合、再始動要求は無視され、スイッチオーバーが発行されます。

データベースの実際のワークロード、RPO と RTO に関する要件、PostgreSQL アーキテクチャが共有されているか何も共有されていないかなどのいくつかの要因に依存するため、更新方法に万能の構成はありません。の上。

実際、PostgreSQL はプライマリ/スタンバイアーキテクチャのデータベース管理システムであるため、更新プロセスでは必然的にアプリケーションのダウンタイムが発生します。コンテキストで考慮すべき重要な側面の 1 つは、ポッドが新しい PostgreSQL コンテナーイメージをダウンロードするのにかかる時間です。これは、Kubernetes クラスターの設定と仕様によって異なります。 `switchover` メソッドは、昇格したインスタンスがコンテナーのターゲット イメージバージョンを既に実行していることを確認します。代わりに、`restart` メソッドでは、プライマリ ポッドがシャットダウンされた後に、オリジンレジストリからイメージをダウンロードする必要がある場合があります。ローリング更新手順の一部として、データベースに対して `restart` または `switchover` を使用するのが最適かどうかを判断するのは、ユーザー次第です。

4.13.2 手動更新 (supervised)

`primaryUpdateStrategy` が `supervised` に設定されている場合、すべてのレプリカがアップグレードされた直後にローリング更新プロセスが中断されます。

このフェーズは、手動スイッチオーバーまたはインプレース再起動でのみ完了できます。

次の方法でスイッチオーバーをトリガーできます。

```
kubectl cnpg promote [cluster] [new_primary]
```

次の方法で再起動をトリガーできます。

```
kubectl cnpg restart [cluster] [current_primary]
```

詳細については、 :ref:``kubectl` に *cnpg* プラグインを使用する<*kubectl* に *cnpg* プラグインを使用する>` を参照してください。

4.14 レプリケーション

物理レプリケーションは PostgreSQL の強みの 1 つであり、世界最大の組織の一部がビジネス継続性のコンテキストでのデータ管理に PostgreSQL を選択した理由の 1 つです。主に高可用性を実現するために使用される物理レプリケーションは、読み取り専用ワークロードのスケールアウトと、プライマリからの一部の作業のオフロードも可能にします。

重要: このセクションでは、同じ Kubernetes クラスターで管理されている同じ *Cluster* リソース内のレプリケーションについて説明します。異なる Kubernetes クラスター間であっても、別の Postgres *Cluster* リソースで複製する方法については、 レプリカ クラスタ を参照してください。

セクション。

4.14.1 アプリケーションレベルのレプリケーション

何年にもわたって PostgreSQL のレプリケーション機能に貢献してきた私たちは、ネイティブの物理レプリケーションテクノロジの上に CloudNativePG で高可用性を構築し、それを Kubernetes API に直接統合することにしました。

Kubernetes の用語では、これは **ストレージ レベルのレプリケーション**とは対照的に、 **アプリケーション レベルのレプリケーション**と呼ばれます。

4.14.2 非常に成熟したテクノロジー

PostgreSQL には、プライマリ インスタンスから 1 つ以上のレプリカにデータをレプリケートするための非常に堅牢で成熟したネイティブフレームワークがあり、WAL (Write Ahead Log) に継続的に保存されるトランザクション変更の概念に基づいて構築されています。

物理レプリケーションは、クラッシュ リカバリとポイント イン タイムリカバリテクノロジーの進化として始まりました。物理レプリケーションは、プライマリから連続リカバリのウォームスタンバイへの WAL シッピングを通じて、PostgreSQL 8.2 (2006) で最初に導入されました。

PostgreSQL 9.0 (2010) では、**ホット スタンバイ** を介して WAL ストリーミングと読み取り専用レプリカで強化され、9.1 (2011) ではトランザクション レベルで同期レプリケーションが導入されました (RPO=0 クラスターの場合)。カスケード レプリケーションは、PostgreSQL 9.2 (2012) でリリースされました。論理レプリケーションの基盤は PostgreSQL 9.4 で構築されましたが、バージョン 10 (2017) では、発信元から宛先にデータを複製するパブリッシャー/サブスクライバーパターンのネイティブ サポートが導入されました。

4.14.3 ストリーミング レプリケーションのサポート

現時点では、CloudNativePG は、spec で提供された instances の数に基づいて、クラスター内の物理ストリーミングレプリカを宣言的な方法でネイティブかつ透過的に管理します。

```
replicas = instances - 1 (where  instances > 0)
```

クラスターの初期化直後に、オペレーターは次のように streaming_replica というユーザーを作成します。

```
CREATE USER streaming_replica WITH REPLICATION;
   -- NOSUPERUSER INHERIT NOCREATEROLE NOCREATEDB NOBYPASSRLS
```

すぐに使用できるオペレーターは、暗号化されたチャネルを介してクラスター内でストリーミングレプリケーションを自動的にセットアップし、streaming_replica ユーザーに TLS クライアント証明書認証を適用します。

```
#  Require client certificate authentication for the streaming_replica user

hostssl postgres streaming_replica all cert
hostssl replication streaming_replica all cert
```

ドキュメントで。

構成されている場合、オペレーターは HA クラスター内のすべてのレプリカのレプリケーションスロットを管理し、フェイルオーバーまたはスイッチオーバーの後でも、各スタンバイに必要な WAL ファイルがプライマリーのストレージに保持されるようにします。

下。

継続的なバックアップ統合

クラスターで継続的なバックアップが構成されている場合、CloudNativePG は、継続的な復旧時に `restore_command` を利用するようにレプリカを透過的に構成します。その結果、PostgreSQL は、ストリーミングレプリケーションによる WAL のプルが失敗するたびに、フォールバック オプションとして WAL アーカイブを使用できます。

4.14.4 同期レプリケーション

CloudNativePG は、`minSyncReplicas` および `maxSyncReplicas` と呼ばれる 2 つの構成オプションを介して**クォーラムベースの同期ストリーミングレプリケーション**の構成をサポートします。これらは、いつでも使用できると予想される同期スタンバイレプリカの最小数と最大数です。自己回復の目的で、オペレーターは常にこれら 2 つの値を使用可能なレプリカの数と比較して、クォーラムを決定します。

重要: デフォルトでは、同期レプリケーションは利用可能なすべてのレプリカの中から区別なく選択します。次のセクションで説明するように、*syncReplicaElectionConstraint* オプションを使用してノード ラベルを操作することにより、同期レプリカをスケジュールできるノードを制限できます。

同期レプリケーションはデフォルトで無効になっています (`minSyncReplicas` と `maxSyncReplicas` は定義されていません)。`minSyncReplicas` と `maxSyncReplicas` の両方が設定されている場合、CloudNativePG は PostgreSQL の `synchronous_standby_names` オプションを次の値に自動的に更新します。

```
ANY q (pod1, pod2, ...)
```

どこ：

- `q` は、オペレーターによって自動的に計算される整数です: `1 <= minSyncReplicas <= q <= maxSyncReplicas <= readyReplicas`
- `pod1, pod2, ...` は、クラスター内のすべての PostgreSQL ポッドのリストです。

警告:

自己修復機能を提供するために、そのような値が現在利用可能なレプリカの数よりも大きい場合、オペレーターは *minSyncReplicas* を無視できます。 *readyReplicas* が *0* の場合、同期レプリケーションは自動的に無効になります。

PostgreSQL documentation で述べたように、*メソッド `ANY`

はクォーラムベースの同期レプリケーションを指定し、WAL レコードが少なくともリスト内の要求された数の同期スタンバイにレプリケートされるまでトランザクションのコミットを待機させます*。

重要: オペレーターが同期レプリケーション設定の強制よりもセルフヒーリングを選択しても、3 つ以上のインスタンスを持つクラスター、またはより一般的には *maxSyncReplicas < (instances - 1)* の場合にのみ、同期レプリケーションを計画することをお勧めします。

同期レプリケーションのノードを選択

CloudNativePG を使用すると、PGDATA を保持する PVC と Postgres ポッドがあるノード ラベルに基づく非アフィニティ ルールを通じて、どの PostgreSQL インスタンスがクォーラムベースの同期レプリケーションセットに参加する資格があるかを選択できます。

この機能の使用例: 単一の同期レプリカを持つクラスターでは、同期レプリカがプライマリインスタンスとは異なるアベイラビリティゾーンにあることを確認できます。通常は `topology.kubernetes.io/zone`

label on a node で識別されます。これにより、特に目標復旧時点 (RPO)

の観点から、単一のアベイラビリティゾーンで障害が発生した場合のクラスターの堅牢性が向上します。

アンチアフィニティの考え方は、クォーラムに参加する同期レプリカが、選択されたラベル (この場合はアベイラビリティ ゾーン ラベル) に対して異なる値を持つノードで実行されているポッドから選択され、次にプライマリが現在あるノードから選択されるようにすることです。実行中。このような基準に一致するノードがない場合、レプリカは同期レプリケーションに適格です。

重要: 同期レプリカ選出のための追加の制約を定義している間も、自己修復の強制が適用されます (同期レプリケーション を参照してください)。

以下の例は、`spec.postgresql` 内の `syncReplicaElectionConstraint` セクションを介してこれを行う方法を示しています。`nodeLabelsAntiAffinity` を使用すると、評価が必要なノードラベルを指定して、現在のプライマリと、アベイラビリティ ゾーンラベルの値が異なるアベイラビリティ ゾーンラベルの値を持つノード上にあるレプリカとの間で、オペレータによって同期レプリケーションが動的に構成されることを確認できます。プライマリがあるノード:

```
spec:
  instances: 3
  postgresql:
    syncReplicaElectionConstraint:
      enabled: true
      nodeLabelsAntiAffinity:
      - topology.kubernetes.io/zone
```

ご想像のとおり、アベイラビリティゾーンは単なる例ですが、ストレージ、CPU、メモリなど、ノードを説明する他のラベルに基づいてこの動作をカスタマイズできます。

4.14.5 高可用性のためのレプリケーション スロット

高可用性のためのレプリケーション スロット

は、9.4 で導入されたネイティブの PostgreSQL 機能であり、接続されているすべてのストリーミングレプリケーションクライアントが WAL セグメントを受信するまでプライマリが WAL セグメントを削除しないようにする自動化された方法を提供し、プライマリが行を削除しないことを保証します。スタンバイは (一時的に) 切断されます。

レプリケーションスロットは、それを作成したインスタンスにのみ存在し、PostgreSQL はそれをスタンバイサーバーにレプリケートしません。その結果、フェイルオーバーまたはスイッチオーバーの後、新しいプライマリには古いプライマリのレプリケーションスロットが含まれません。これにより、古いプライマリに接続されていてスロットを失ったストリーミングレプリケーションクライアントに問題が発生する可能性があります。

CloudNativePG は、高可用性クラスターから始めて、クラスター管理のレプリケーションスロットの概念を導入することで、このギャップを埋めます。この機能は、プライマリとスタンバイの両方で、高可用性クラスター内の各ホットスタンバイ レプリカの物理レプリケーション スロットを自動的に管理します。

CloudNativePG では、次の用語を使用します。

- **プライマリ HA スロット**: ライフサイクルがクラスターの現在のプライマリによって完全に管理され、ストリーミングレプリケーションで特定のスタンバイにマップすることを目的とする物理レプリケーションスロット。このようなスロットは、プライマリのみに存在します。
- **スタンバイ HA スロット**: プライマリの `pg_replication_slots` ビューの内容に基づいてライフサイクルがクラスター内の別のスタンバイによって完全に管理され、`pg_replication_slot_advance()` を使用して定期的に更新されるスタンバイの物理レプリケーションスロット。

CloudNativePG 1.18 で導入されたこの機能は、構成によって有効にすることができます。詳細は

ReplicationSlotsConfiguration

を参照してください。次に、主なオプションの簡単な説明を示します。

`.spec.replicationSlots.highAvailability.enabled`: true の場合、機能が有効になります (`false` がデフォルトです)

`.spec.replicationSlots.highAvailability.slotPrefix`: この機能のオペレーターによって管理されるレプリケーションスロットを識別するプレフィックス (デフォルト: `_cnpg_`)

`.spec.replicationSlots.updateInterval`: スタンバイがレプリケーションスロットのローカル コピーの位置を現在のプライマリの位置と同期する頻度 (秒単位) (デフォルト: 30)

重要: この機能には pg_replication_slot_advance() に依存しているため、PostgreSQL 11 以降が必要です。

複製スロットの位置を直接操作します。

警告: PostgreSQL 11 では、レプリケーション スロットを最初に無効にした場合に有効にするか、逆に最初に有効にした場合に無効にすると、クラスターのローリング アップデートが必要になります (起動時にのみ読み取られる *recovery.conf* ファイルが存在するため)。

お勧めしませんが、別の動作が必要な場合は、上記のオプションをカスタマイズできます。

たとえば、次のマニフェストは、レプリケーションスロットが有効なクラスターを作成します。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  # Enable replication slots for HA in the cluster
  replicationSlots:
    highAvailability:
      enabled: true

  storage:
    size: 1Gi
```

次の例のように、スタンバイがプライマリの pg_replication_slots ビューをクエリし、レプリケーション スロットのローカルコピーを更新する頻度を制御することもできます。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  # Reduce the frequency of standby HA slots updates to once every 5 minutes
  replicationSlots:
    highAvailability:
      enabled: true
    updateInterval: 300

  storage:
    size: 1Gi
```

レプリケーションスロットは、インフラストラクチャで注意深く監視する必要があります。デフォルトでは、Prometheus エクスポーターの pg_replication_slots メトリクスに、スロットの名前、タイプ、アクティブかどうか、プライマリからのラグなどの重要な情報を提供します。

4.15 バックアップとリカバリ

CloudNativePG は、継続的な物理バックアップと WAL アーカイブを通じて、PostgreSQL クラスターの**オンライン/ホットバックアップ**をネイティブにサポートします。これは、データベースが常に稼働しており (ダウンタイムが不要)、システムで最初に使用可能なベースバックアップからいつでも復旧できることを意味します。後者は通常、「ポイントイン タイム リカバリ」(PITR) と呼ばれます。

オペレーターは、*BarmanCredentials* ツールに基づく継続的なバックアップインフラストラクチャを調整できます。多くの PostgreSQL インスタンスをバックアップする Barman サーバーで従来のアーキテクチャを使用する代わりに、オペレーターは `barman-cloud-wal-archive` 、`barman-cloud-check-wal-archive` 、`barman-cloud-backup` 、`barman-cloud-backup-list` 、および `barman-cloud-backup-delete` ツールに依存しています。その結果、ベースバックアップは *tarballs* になります。ベース バックアップと WAL ファイルの両方を圧縮および暗号化できます。

このためには、`barman-cli-cloud` が含まれているイメージを使用する必要があります。コミュニティ PostgreSQL イメージと最新の `barman-cli-cloud` パッケージで構成されているため、このスコープにはイメージ `ghcr.io/cloudnative-pg/postgresql` を使用できます。

重要: システムでオペランドの最新バージョンを実行していることを常に確認して、 Barman クラウドで導入された改善を活用してください (また、クラスターのセキュリティ面を改善します)。

バックアップは、`Cluster` のプライマリまたは指定されたプライマリインスタンスから実行されます (**レプリカ クラスタ** を参照してください)。

指定されたプライマリ インスタンスの詳細については、 **スタンバイからのバックアップ**を参照してください)。

4.15.1 クラウド プロバイダーのサポート

Barman Cloud インフラストラクチャでサポートされている任意のサービスでバックアップファイルをアーカイブできます。あれは：

- AWS S3
- Microsoft Azure Blob Storage
- *Google* **クラウド ストレージ**

サポートされているサービスの互換性のある実装を使用することもできます。

必要なセットアップは、選択したストレージプロバイダーによって異なります。これについては、次のセクションで説明します。

S3

次の 2 つの方法で、S3 バケットにバックアップを保存するためのアクセス許可を定義できます。

- CloudNativePG が EKS で実行されている場合。 IRSA authentication method を使用することもできます
- または、ACCESS_KEY_ID および ACCESS_SECRET_KEY 資格情報を使用することもできます

AWS アクセス キー

環境に関する次の情報が必要になります。

- ACCESS_KEY_ID : S3 にファイルをアップロードするために使用されるアクセスキーの ID
- ACCESS_SECRET_KEY : 上記のアクセス キーの秘密部分
- ACCESS_SESSION_TOKEN : 必要な場合のオプションのセッショントークン

使用するアクセスキーには、ファイルをバケットにアップロードする権限が必要です。そのため、資格情報を使用して Kubernetes シークレットを作成する必要があり、次のコマンドでそれを行うことができます。

```
kubectl create secret generic aws-creds \
  --from-literal=ACCESS_KEY_ID=<access key here> \
  --from-literal=ACCESS_SECRET_KEY=<secret key here>
#  --from-literal=ACCESS_SESSION_TOKEN=<session token here> # if required
```

資格情報は Kubernetes 内に保存され、インストールで保存時の暗号化が構成されている場合は暗号化されます。

そのシークレットが作成されたら、次の例のようにクラスターを構成できます。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      destinationPath: "<destination path here>"
      s3Credentials:
        accessKeyId:
          name: aws-creds
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: aws-creds
          key: ACCESS_SECRET_KEY
```

宛先パスは、インスタンスが WAL ファイルをアップロードできるフォルダーを指す任意の URL にすることができます。 s3://BUCKET_NAME/path/to/folder .

サービス アカウントの IAM ロール (IRSA)

IRSA を使用するには、Postgres クラスターの ServiceAccount に annotation を設定する必要があります。

serviceAccountTemplate スタンザを使用してそれらを注入するように CloudNativePG を構成できます。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
[...]
spec:
  serviceAccountTemplate:
    metadata:
      annotations:
        eks.amazonaws.com/role-arn: arn:[...]
        [...]
```

その他の S3 互換 Object Storage プロバイダー

MinIO や **Linode オブジェクトストレージ** などの S3 互換オブジェクトストレージを使用している場合は、デフォルトの S3 のものを使用する代わりに、エンドポイントを指定できます。

この例では、リージョン us-east1 で **Linode** の bucket を使用します。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      destinationPath: "<destination path here>"
      endpointURL: "https://bucket.us-east1.linodeobjects.com"
      s3Credentials:
        [...]
```

デジタルオーシャンスペース を使用している場合は、パススタイルの構文を使用する必要があります。この例では、領域 SFO3 で **DigitalOceanSpaces** の bucket を使用します。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      destinationPath: "s3://[your-bucket-name]/[your-backup-folder]/"
      endpointURL: "https://sfo3.digitaloceanspaces.com"
```

(次のページに続く)

(前のページからの続き)

```
    s3Credentials:
      [...]
```

重要: HTTPS 経由で MinIO を使用する場合のように、プライベート CA で署名された証明書を使用するオブジェクト ストレージ プロバイダーを構成するとします。その場合、 Barman が証明書を正しく検証できるように、CA バンドルを含むシークレットを参照するオプション *endpointCA* を設定する必要があります。

注釈: インスタンスによって ConfigMap とシークレットを **自動的に** リロードする場合は、キー *cnpg.io/reload* のラベルを Secrets/ConfigMaps に追加できます。それ以外の場合は、*kubectl cnpg reload* サブコマンドを使用してインスタンスをリロードする必要があります。

MinIO ゲートウェイ

必要に応じて、バックアップ オブジェクトを S3 や GCS などの他のクラウドストレージ ソリューションに中継する共通インターフェイスとして、MinIO ゲートウェイを使用できます。詳細については、 MinIO official documentation を参照してください。

具体的には、CloudNativePG クラスターは、以前に作成された資格情報とサービスを使用して、エンドポイントとしてローカルの MinIO ゲートウェイを直接指すことができます。

MinIO シークレットは、PostgreSQL クラスターと MinIO インスタンスの両方で使用されます。したがって、それらを同じ名前空間に作成する必要があります。

```
kubectl create secret generic minio-creds \
  --from-literal=MINIO_ACCESS_KEY=<minio access key here> \
  --from-literal=MINIO_SECRET_KEY=<minio secret key here>
```

この場合、.. Note::

クラウド オブジェクト ストレージ資格情報は、MinIO ゲートウェイによってのみ使用されます。

重要: PostgreSQL が MinIO ゲートウェイに到達できるようにするには、MinIO ゲートウェイ インスタンスにバインドされたポート *9000* で *ClusterIP* サービスを作成する必要があります。

例えば：

```
apiVersion: v1
kind: Service
metadata:
  name: minio-gateway-service
spec:
```

(次のページに続く)

(前のページからの続き)

```
  type: ClusterIP
  ports:
    - port: 9000
      targetPort: 9000
      protocol: TCP
  selector:
    app: minio
```

警告: このドキュメントの執筆時点では、公式の MinIO Operator

for Kubernetes はゲートウェイ機能をサポートしていません。そのため、代わりに deployment を使用します。

MinIO の展開では、クラウドストレージの資格情報を使用してオブジェクトをリモートバケットにアップロードし、バックアップ ファイルを別の場所に中継します。

AWS S3 を Cloud Object Storage として使用する例を次に示します。

```
apiVersion: apps/v1
kind: Deployment
[...]
spec:
  containers:
  - name: minio
    image: minio/minio:RELEASE.2020-06-03T22-13-49Z
    args:
    - gateway
    - s3
    env:
    # MinIO access key and secret key
    - name: MINIO_ACCESS_KEY
      valueFrom:
        secretKeyRef:
          name: minio-creds
          key: MINIO_ACCESS_KEY
    - name: MINIO_SECRET_KEY
      valueFrom:
        secretKeyRef:
          name: minio-creds
          key: MINIO_SECRET_KEY
    # AWS credentials
    - name: AWS_ACCESS_KEY_ID
      valueFrom:
        secretKeyRef:
```

(次のページに続く)

(前のページからの続き)

```
          name: aws-creds
          key: ACCESS_KEY_ID
    - name: AWS_SECRET_ACCESS_KEY
      valueFrom:
        secretKeyRef:
          name: aws-creds
          key: ACCESS_SECRET_KEY
#  Uncomment the below section if session token is required

#    - name: AWS_SESSION_TOKEN

#      valueFrom:

#        secretKeyRef:

#          name: aws-creds

#          key: ACCESS_SESSION_TOKEN

        ports:
        - containerPort: 9000
```

Cluster 定義で endpointURL として MinIO ゲートウェイサービスを構成して続行し、BUCKET_NAME を置き換えるバケット名を選択します。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      destinationPath: s3://BUCKET_NAME/
      endpointURL: http://minio-gateway-service:9000
      s3Credentials:
        accessKeyId:
          name: minio-creds
          key: MINIO_ACCESS_KEY
        secretAccessKey:
          name: minio-creds
          key: MINIO_SECRET_KEY
    [...]
```

バックアップを続行する前に、s3://BUCKET_NAME/ でアーカイブされた WAL ファイルの存在を確認します。

Azure ブロブ ストレージ

ストレージアカウントにアクセスするには、次の資格情報の組み合わせのいずれかが必要です。

- **Connection String**
- **ストレージアカウント名** と **Storage account access key**
- **ストレージアカウント名** と **Storage account SAS Token**
- **ストレージアカウント名** と **Azure AD Workload Identity**

適切に構成されています。

AzureAD ワークロード ID を使用すると、資格情報を Kubernetes シークレットに保存することを回避し、次のように `inheritFromAzureAD` を追加するクラスター構成を持つことができます。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      destinationPath: "<destination path here>"
      azureCredentials:
        inheritFromAzureAD: true
```

一方、 **ストレージアカウントアクセスキー** または**ストレージアカウント SAS トークン** の両方を使用する場合、資格情報を Kubernetes シークレット内に格納し、必要な場合にのみデータエントリを追加する必要があります。次のコマンドはそれを実行します。

```
kubectl create secret generic azure-creds \
  --from-literal=AZURE_STORAGE_ACCOUNT=<storage account name> \
  --from-literal=AZURE_STORAGE_KEY=<storage account key> \
  --from-literal=AZURE_STORAGE_SAS_TOKEN=<SAS token> \
  --from-literal=AZURE_STORAGE_CONNECTION_STRING=<connection string>
```

使用されている Kubernetes クラスターでこの機能が有効になっている場合、資格情報は保存時に暗号化されます。

前のシークレットを指定すると、提供された資格情報をクラスター構成内に挿入できます。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      destinationPath: "<destination path here>"
      azureCredentials:
```

(次のページに続く)

(前のページからの続き)

```
      connectionString:
        name: azure-creds
        key: AZURE_CONNECTION_STRING
      storageAccount:
        name: azure-creds
        key: AZURE_STORAGE_ACCOUNT
      storageKey:
        name: azure-creds
        key: AZURE_STORAGE_KEY
      storageSasToken:
        name: azure-creds
        key: AZURE_STORAGE_SAS_TOKEN
```

Azure Blob Storage を使用する場合、destinationPath は次の構造を満たします。

```
<http|https>://<account-name>.<service-name>.core.windows.net/<resource-path>
```

ここで、<resource-path> は <container>/<blob> です。**ストレージアカウント名** とも呼ばれる **アカウント名**は、使用されるホスト名に含まれます。

その他の Azure Blob Storage 互換プロバイダー

Azure Blob Storage API の別の実装を使用している場合、destinationPath は次の構造になります。

```
<http|https>://<local-machine-address>:<port>/<account-name>/<resource-path>
```

その場合、<account-name> はパスの最初のコンポーネントです。

これは、Azure ストレージ エミュレーターまたは Azurite を介して Azure サポートをテストする場合に必要です。

Google クラウド ストレージ

現在、オペレーターは Google Cloud Storage の 2 つの認証方法をサポートしています。

- 最初のものは、ポッドが Google Kubernetes Engine クラスタ内で実行されていることを前提としています
- 2 つ目は、環境変数 GOOGLE_APPLICATION_CREDENTIALS を利用します。

Google Kubernetes Engine 内での実行

Google Kubernetes Engine 内で実行している場合、資格情報を設定する必要なく、単純に Workload Identity に依存するようにバックアップを構成できます。特に、次のことを行う必要があります。

- `.spec.backup.barmanObjectStore.googleCredentials.gkeEnvironment` を `true` に設定
- `serviceAccountTemplate` スタンザに `iam.gke.io/gcp-service-account` 注釈を設定します

次の例を参考にしてください。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  [...]
  backup:
    barmanObjectStore:
      destinationPath: "gs://<destination path here>"
      googleCredentials:
        gkeEnvironment: true

  serviceAccountTemplate:
    metadata:
      annotations:
        iam.gke.io/gcp-service-account:  [...].iam.gserviceaccount.com
        [...]
```

認証の使用

> instruction from Google に続く

認証に必要なすべての情報を含む JSON ファイルを取得します。

JSON ファイルのコンテンツは、次のコマンドで作成できる `Secret` を使用して提供する必要があります。

```
kubectl create secret generic backup-creds --from-file=gcsCredentials=gcs_credentials_
↪file.json
```

これにより、`backup-creds` という名前の `Secret` が作成され、yaml ファイルで次のように使用されます。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
```

(次のページに続く)

(前のページからの続き)

```
    destinationPath: "gs://<destination path here>"
    googleCredentials:
      applicationCredentials:
        name: backup-creds
        key: gcsCredentials
```

これで、オペレーターは資格情報を使用して Google Cloud Storage に対して認証します。

重要: この認証方法では、Google Cloud Storage バケットにアクセスするために必要なすべての情報を含む JSON ファイルがコンテナ内に作成されます。つまり、誰かがポッドにアクセスすると、バケットへの書き込み権限も付与されます。

4.15.2 オンデマンド バックアップ

新しいバックアップをリクエストするには、次のような新しいバックアップリソースを作成する必要があります。

```
apiVersion: postgresql.cnpg.io/v1
kind: Backup
metadata:
  name: backup-example
spec:
  cluster:
    name: pg-backup
```

オペレーターは、barman-cloud-backup を使用して必要なバックアップを取得するためにクラスターの調整を開始します。プレーンな kubectl describe backup <name> コマンドを使用して、バックアップステータスを確認できます。

```
Name:         backup-example
Namespace:    default
Labels:       <none>
Annotations:  API Version:  postgresql.cnpg.io/v1
Kind:         Backup
Metadata:
  Creation Timestamp:  2020-10-26T13:57:40Z
  Self Link:         /apis/postgresql.cnpg.io/v1/namespaces/default/backups/backup-
↪example
  UID:               ad5f855c-2ffd-454a-a157-900d5f1f6584
Spec:
  Cluster:
    Name:  pg-backup
```

(次のページに続く)

(前のページからの続き)

```
Status:
  Phase:       running
  Started At:  2020-10-26T13:57:40Z
Events:        <none>
```

バックアップが完了すると、フェーズは次の例のように completed になります。

```
Name:         backup-example
Namespace:    default
Labels:       <none>
Annotations:  API Version:  postgresql.cnpg.io/v1
Kind:         Backup
Metadata:
  Creation Timestamp:  2020-10-26T13:57:40Z
  Self Link:         /apis/postgresql.cnpg.io/v1/namespaces/default/backups/backup-
↪example
  UID:               ad5f855c-2ffd-454a-a157-900d5f1f6584
Spec:
  Cluster:
    Name:  pg-backup
Status:
  Backup Id:         20201026T135740
  Destination Path:  s3://backups/
  Endpoint URL:      http://minio:9000
  Phase:             completed
  s3Credentials:
    Access Key Id:
      Key:   ACCESS_KEY_ID
      Name:  minio
    Secret Access Key:
      Key:      ACCESS_SECRET_KEY
      Name:     minio
  Server Name:  pg-backup
  Started At:   2020-10-26T13:57:40Z
  Stopped At:   2020-10-26T13:57:44Z
Events:         <none>
```

!!!重要 この機能は、スーパーユーザーとアプリケーションユーザーのシークレットをバックアップしません。シークレットは、Kubernetes クラスターの標準的なバックアップ手順の一部としてバックアップされることになっています。

4.15.3 スケジュールされたバックアップ

ScheduledBackup という名前のリソースを作成して、バックアップを定期的にスケジュールすることもできます。後者は Backup に似ていますが、schedule というフィールドが追加されています。

このフィールドは *cron* **スケジュール** 仕様であり、同じ format used in Kubernetes CronJobs に従います。

これはスケジュールされたバックアップの例です:

```
apiVersion: postgresql.cnpg.io/v1
kind: ScheduledBackup
metadata:
  name: backup-example
spec:
  schedule: "0 0 0 * * *"
  backupOwnerReference: self
  cluster:
    name: pg-backup
```

上記の例では、毎日午前 0 時にバックアップをスケジュールします。

ヒント: バックアップの頻度は、完全またはポイントインタイムの復旧操作を必要とする災害の後の復旧時間オブジェクト (RTO) に影響を与える可能性があります。バックアップを復元して定期的にテストし、最初から復元するのにかかる時間を測定して、RTO の予測可能性を改善することをお勧めします。リカバリ時間は、ベース バックアップのサイズと、アーカイブから取得してリカバリ中に再生する必要がある WAL ファイルの量に影響されます (WAL アーカイブは、PostgreSQL での継続的なバックアップを可能にするものであることを思い出してください!)。私たちの経験に基づくと、ほとんどの場合、毎週のベース バックアップで十分ですが、1 日 1 回以上の頻度でバックアップをスケジュールすることは非常にまれです。

.spec.suspend: true を設定することにより、必要に応じて ScheduledBackups を一時停止できます。これにより、オプションが false に設定されている限り、スケジュールされる新しいバックアップが停止されます。

ScheduledBackup リソースが作成されるとすぐにバックアップを発行する場合は、.spec.immediate: true を設定できます。

注釈: *.spec.backupOwnerReference* は、作成されたバックアップ リソース内にどの ownerReference を配置する必要があるかを示します。

- *none:* 作成されたバックアップ オブジェクトの所有者参照なし (フィールドが導入される前と同じ動作) - *self:* バックアップの所有者としてスケジュールされたバックアップオブジェクトを設定します - *cluster:* バックアップの所有者としてクラスターを設定します

4.15.4 WAL アーカイブ

宛先パスを選択し、クラウド資格情報を構成するとすぐに、WAL アーカイブが有効になります。

必要に応じて、WAL ファイルがアップロードされたらすぐに圧縮したり、暗号化したりすることを選択できます。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      [...]
      wal:
        compression: gzip
        encryption: AES256
```

バケットで暗号化を直接構成できます。クラスター構成でオーバーライドしない限り、オペレーターはそれを使用します。

PostgreSQL は、アーカイブされるすべての WAL セグメントに対して archive_command が順次実行される順次アーカイブスキームを実装します。

重要: デフォルトでは、CloudNativePG は *archive_timeout* を *5min* に設定し、ワークロードが低い場合でも、WAL ファイルが少なくとも 5 分ごとに閉じられてアーカイブされるようにし、目標復旧時点 (RPO) の決定論的な時間ベースの値を提供します。 archive_timeout の値を変更しても、オペレーターによって設定されたデフォルト値がほとんどのユースケースに適していることが、私たちの経験から示唆されています。

PostgreSQL インスタンスとオブジェクト ストア間の帯域幅により、複数の WAL ファイルを並行してアーカイブできる場合、次の例のように、インスタンスマネージャーの並行 WAL アーカイブ機能を使用できます。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      [...]
      wal:
        compression: gzip
        maxParallel: 8
        encryption: AES256
```

前の例では、インスタンス マネージャは、PostgreSQL によって要求されたものを含め、最大 8 つの準備が整った WAL を並行してアーカイブすることにより、WAL アーカイブプロセスを最適化します。

PostgreSQL が最適化としてインスタンスマネージャーによって既にアーカイブされている WAL のアーカイブを要求する場合、そのアーカイブ要求は肯定的なステータスで却下されます。

4.15.5 スタンバイからのバックアップ

デフォルトでは、バックアップは Cluster のプライマリインスタンスで実行されます。

ベース バックアップを作成するには、ディスク上の PostgreSQL インスタンスのデータコンテンツ全体をスクレイピングする必要があるため、データベースの実際のワークロードとの I/O 競合が発生する可能性があります。

このため、CloudNativePG では、PostgreSQL で直接利用できる機能、**スタンバイからのバックアップ** を利用できます。

以下の例に示すように、バックアップ ターゲットを prefer-standby に設定できます。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  [...]
spec:
  backup:
    target: "prefer-standby"
```

prefer-standby ポリシーは、バックアップが最新の使用可能なセカンダリインスタンスで実行されることを保証し、他のインスタンスが使用できない場合はプライマリインスタンスにフォールバックします。

デフォルトでは、指定されていない場合、ターゲットはプライマリからバックアップを取るように自動的に設定されます。

Cluster で指定されたバックアップターゲットは、次の例のように、Backup および ScheduledBackup タイプでオーバーライドできます。

```
apiVersion: postgresql.cnpg.io/v1
kind: Backup
metadata:
  [...]
spec:
  cluster:
    name: [...]
  target: "primary"
```

前の例では、Cluster がレプリカを優先するように設定されていても、CloudNativePG は常にプライマリインスタンスを選択します。

4.15.6 回復

クラスターの復元は、既存のクラスターでは“インプレース”で実行されません。オブジェクトストレージにアップロードされたデータを使用して、以前に取得したバックアップから新しいクラスターを**ブートストラップ** できます。オペレーターは、`barman-cloud-restore` ツール (ベース バックアップ用) と `barman-cloud-wal-restore` ツール (WAL ファイル用、必要に応じて並列サポートを含む) を使用して、復旧プロセスを調整します。

`recovery` ブートストラップメソッドの詳細と手順については、 バックアップからのブートストラップ (`recovery )<バックアップからのブートストラップ (*recovery* )>` を参照してください。

重要: PostgreSQL PITR の方法に慣れていない場合

`.spec.postgresql.parameters` に関しては、回復クラスターを元のクラスターとして構成することをお勧めします。新しいクラスターが復元されたら、必要に応じて設定を変更できます。

内部では、オペレーターが新しいクラスターの最初のインスタンスに init コンテナーを挿入し、init コンテナーがオブジェクトストレージからのバックアップの回復を開始します。

重要: 新しい PVC のベース バックアップ コピーの期間は、バックアップのサイズと、ネットワークとストレージの両方の速度によって異なります。

ベース バックアップのリカバリ プロセスが完了すると、オペレータは Postgres インスタンスをリカバリ モードで起動します。このフェーズでは、PostgreSQL は稼働していますが、接続を受け入れることはできず、ポッドは liveness プローブに従って正常です。 `restore_command` を介して、PostgreSQL はアーカイブから WAL ファイルのフェッチを開始します (`maxParallel` オプションを設定し、並列 WAL 復元機能を有効にすることで、このフェーズを高速化できます)。

このフェーズは、PostgreSQL がターゲット (WAL の終わり、またはポイントインタイム リカバリの場合は必要なターゲット) に到達すると終了します。実際、必要に応じて `recoveryTarget` を指定して、ポイント イン タイムリカバリを実行できます。指定しない場合、リカバリはデフォルトのターゲットタイムラインで利用可能な最新の WAL まで続行されます (11 までの PostgreSQL の場合は `current` 、バージョン 12 以降の場合は `latest`)。

リカバリが完了すると、オペレータは必要なスーパーユーザーパスワードをインスタンスに設定します。新しいプライマリインスタンスは通常どおり起動し、残りのインスタンスはレプリカとしてクラスターに参加します。

このプロセスはユーザーに対して透過的であり、Pod で実行されているインスタンス マネージャーによって管理されます。

バックアップ セクションを使用してクラスターに復元する

クラスター復元のマニフェストには、backup セクションが含まれる場合があります。これは、リカバリ後に新しいクラスタが WAL のアーカイブを開始し、そのように構成されている場合はバックアップを取得することを意味します。

たとえば、以下のセクションは、クラスター cluster-example-backup からのクラスターブートストラップのマニフェストの一部であり、復元されたクラスターのベースバックアップと WAL が保存される recoveredCluster という名前のストレージ バケットに新しいフォルダーを作成します。

```
backup:
  barmanObjectStore:
    destinationPath: s3://backups/
    endpointURL: http://minio:9000
    serverName: "recoveredCluster"
    s3Credentials:
      accessKeyId:
        name: minio
        key: ACCESS_KEY_ID
      secretAccessKey:
        name: minio
        key: ACCESS_SECRET_KEY
  retentionPolicy: "30d"

externalClusters:
- name: cluster-example-backup
  barmanObjectStore:
    destinationPath: s3://backups/
    endpointURL: http://minio:9000
    s3Credentials:
```

異なるクラスターにまったく同じ barmanObjectStore 構成を再利用しないでください。ストレージバケット内の既存の情報が新しいクラスターによって上書きされる場合があります。

> **警告:** オペレーターには、クラスターが情報を含むストレージ バケットを上書きしないようにするための安全チェックが含まれています。既存のストレージを上書きするクラスターは、状態 *Setting up primary* のままになり、Pod はエラー状態になります。ポッドのログには次のように表示されます: *ERROR: WAL archive check failed for server recoveredCluster: Expected empty archive*

4.15.7 保持ポリシー

CloudNativePG は、リカバリ ウィンドウに基づく **保持ポリシー**を使用して、バックアップ オブジェクトストアからのバックアップファイルの自動削除を管理できます。

内部的には、保持ポリシー機能は barman-cloud-backup-delete と--retention-policy “RECOVERY WINDOW OF {{ retention policy value }} {{ retention policy unit }}”を使用します。

たとえば、次のように 30 日の保持ポリシーでバックアップを定義できます。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      destinationPath: "<destination path here>"
      s3Credentials:
        accessKeyId:
          name: aws-creds
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: aws-creds
          key: ACCESS_SECRET_KEY
    retentionPolicy: "30d"
```

4.15.8 圧縮アルゴリズム

デフォルトでは、CloudNativePG はバックアップと WAL ファイルを非圧縮形式でアーカイブします。ただし、barman-cloud-backup (バックアップ用) および barman-cloud-wal-archive (WAL ファイル用) を介して次の圧縮アルゴリズムもサポートしています。

- bzip2
- gzip

*きびきび

バックアップと WAL の圧縮設定は独立しています。 API リファレンスの
データバックアップ構成 および *WalBackupConfiguration* セクションを参照してください。

アルゴリズム間でアーカイブ時間、復元時間、およびサイズが異なることに注意することが重要であるため、使用事例に応じて圧縮アルゴリズムを選択する必要があります。

Barman チームは、Barman Cloud でサポートされているアルゴリズムのパフォーマンスを評価しました。次の表は、ローカルの MinIO 展開でバックアップが作成されるシナリオをまとめたものです。 Barman GitHub プロジェクトには deeper analysis が含まれています。

Compression	Backup Time (ms)	Restore Time (ms)	Uncompressed size (MB)	Compressed size (MB)	Approx
None	10927	7553	395	395	1:1
bzip2	25404	13886	395	67	5.9:1
gzip	116281	3077	395	91	4.3:1
snappy	8134	8341	395	166	2.4:1

4.15.9 バックアップ オブジェクトのタグ付け

Barman 2.18 では、`barman-cloud-backup` および `barman-cloud-wal-archive` を介してオブジェクト ストアにバックアップリソースを保存するときに、バックアップリソースにタグを付けるためのサポートが導入されています。その結果、PostgreSQL コンテナ イメージにバージョン 2.18 以降の Barman が含まれている場合、CloudNativePG では、バックアップオブジェクト、つまりベース バックアップ、WAL ファイル、および履歴ファイルのキーと値のペアとしてタグを指定できます。

`.spec.backup.barmanObjectStore` 定義で 2 つのプロパティを使用できます。

- `tags` : バックアップ オブジェクトおよびバックアップ オブジェクトストア内のアーカイブされた WAL ファイルに追加されるキーと値のペアのタグ
- `historyTags` : バックアップ オブジェクトストア内のアーカイブされた履歴ファイルに追加されるキーと値のペアのタグ

以下の YAML マニフェストの抜粋は、この機能の使用例を示しています。

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      [...]
      tags:
        backupRetentionPolicy: "expire"
      historyTags:
        backupRetentionPolicy: "keep"
```

4.16 PostgreSQL の設定

PostgreSQL に精通しているユーザーは、インスタンスを構成する次の 2 つのファイルの存在を認識しています。

- `postgresql.conf` : PostgreSQL のメインのランタイム構成ファイル
- `pg_hba.conf` : クライアント認証ファイル

宣言型構成の概念と PostgreSQL コンテナーの不変性により、ユーザーはこれらのファイルに直接アクセスすることはできません。parameters および pg_hba キーを介してカスタム postgresql.conf および pg_hba.conf 設定を定義することにより、Cluster リソース定義の postgresql セクションを介して構成が可能です。

これらの設定は、すべてのインスタンスで同じです。

警告: *ALTER SYSTEM* クエリを使用して、PostgreSQL インスタンスの構成を強制的に変更しないでください。通常はオペレーターが制御するオプションの一部を変更すると、クラスターが予測不能/回復不能な状態になる可能性があります。さらに、*ALTER SYSTEM* の変更はクラスター全体に複製されません。

カスタム設定の使用法に関するリファレンスはサンプルに含まれています。

cluster-example-custom.yaml を参照してください。

4.16.1 postgresql セクション

ポッド内の PostgreSQL インスタンスは、デフォルトの postgresql.conf ファイルで開始され、これらの設定が自動的に追加されます。

```
listen_addresses = *
include custom.conf
```

custom.conf ファイルには、次の例のように、postgresql セクションにユーザー定義の設定が含まれます。

```
# ...
postgresql:
  parameters:
    shared_buffers: "1GB"
# ...
```

custom.conf のコンテンツは、次のセクションをこの順序で適用することにより、オペレーターによって自動的に生成および維持されます。

- グローバルデフォルトパラメータ
- PostgreSQL メジャー バージョンに依存するデフォルト パラメータ
- ユーザー提供のパラメーター
- 固定パラメータ

グローバルデフォルトパラメータ は次のとおりです。

```
dynamic_shared_memory_type = posix
logging_collector = on
log_destination = csvlog
```

(次のページに続く)

(前のページからの続き)

```
log_directory = /controller/log
log_filename = postgres
log_rotation_age = 0
log_rotation_size = 0
log_truncate_on_rotation = false
max_parallel_workers = 32
max_replication_slots = 32
max_worker_processes = 32
shared_memory_type = mmap # for PostgreSQL >= 12 only
wal_keep_size = 512MB # for PostgreSQL >= 13 only
wal_keep_segments = 32 # for PostgreSQL <= 12 only
wal_sender_timeout = 5s
wal_receiver_timeout = 5s
```

警告: PostgreSQL クラスターでの WAL セグメントの保持を計画し、予想および観察されたワークロードに基づいて、サーバーのバージョンに応じて *wal_keep_size* または *wal_keep_segments* のいずれかを適切に構成することは、ユーザーの義務です。

または、唯一のストリーミングレプリケーションクライアントが高可用性クラスターで実行されているレプリカインスタンスである場合は、クラスター レベルでのレプリケーションスロットのサポートを追加するレプリケーションスロット機能を利用できます。 `replicationSlots.highAvailability` オプションを使用してこの機能を有効にすることができます (詳細については、

> 複製ユーザーについて を参照してください)。

レプリケーションスロットも継続的なバックアップも配置されていない場合、`wal_keep_size` または `wal_keep_segments` を構成することが、スタンバイが同期から外れるのを防ぐ唯一の方法です。スタンバイが同期から外れると、`"could not receive data from WAL stream: ERROR: requested WAL segment **** **** **** **** **** **** has already been removed"` のようなエラーメッセージが生成されます。これには、ストリーミングレプリケーションの目的で古い WAL セグメントを保持するために、`PGDATA` の一部、または WAL ファイルの保存専用のボリュームを専用にする必要があります。

次のパラメータは **固定**であり、オペレーターによって排他的に制御されます。

```
archive_command = /controller/manager wal-archive %p
archive_mode = on
full_page_writes = on
hot_standby = true
listen_addresses = *
port = 5432
restart_after_crash = false
ssl = on
ssl_ca_file = /controller/certificates/client-ca.crt
ssl_cert_file = /controller/certificates/server.crt
ssl_key_file = /controller/certificates/server.key
```

(次のページに続く)

(前のページからの続き)

```
unix_socket_directories = /controller/run
wal_level = logical
wal_log_hints = on
```

固定パラメーターは最後に追加されるため、YAML 構成を介してユーザーがオーバーライドすることはできません。これらのパラメータは、WAL のアーカイブと複製を正しく行うために必要です。

レプリケーション設定

`primary_conninfo` 、`restore_command` 、および `recovery_target_timeline` パラメーターは、クラスター内のインスタンスの状態に応じてオペレーターによって自動的に管理されます。

```
primary_conninfo = host=cluster-example-rw user=postgres dbname=postgres
recovery_target_timeline = latest
```

ログ制御設定

オペレーターは PostgreSQL がそのログを CSV 形式で出力することを要求し、インスタンスマネージャーはそれを自動的に解析して JSON 形式で出力します。このため、PostgreSQL のすべてのログ設定は固定されており、変更できません。

詳細については、 ロギング を参照してください。

共有プリロード ライブラリ

PostgreSQL の `shared_preload_libraries` オプションは、サーバーの起動時に事前に読み込まれる 1 つ以上の共有ライブラリをコンマ区切りリストの形式で指定するために存在します。通常、システム全体のほとんどのデータベースセッションで利用可能にする必要がある拡張機能をロードするために PostgreSQL で使用されます (例: `pg_stat_statements`)。

CloudNativePG では、`shared_preload_libraries` オプションはデフォルトで空です。`shared_preload_libraries` の内容を上書きすることはできますが、このオプションは Postgres のエキスパート ユーザーのみが利用することをお勧めします。

重要: 指定されたライブラリが見つからない場合、サーバーは起動に失敗し、CloudNativePG の自己修復の試みが妨げられ、手動の介入が必要になります。 *shared_preload_libraries* のコンテンツを直接管理する予定がある場合は、必ず拡張機能と *shared_preload_libraries* の設定の両方をテストしてください。

CloudNativePG は、最もよく使用される一部の PostgreSQL 拡張機能の `shared_preload_libraries` オプションの内容を自動的に管理できます (詳細については、以下の マネージド拡張機能 セクションを参照してください)。

具体的には、オペレーターは、構成パラメーターが管理ライブラリーの 1 つを必要とすることに気付くとすぐに、必要なライブラリーを自動的に追加します。オペレーターは、実際のパラメーターがそれを必要としなくなるとすぐに、ライブラリーも削除します。

重要: *shared_preload_libraries* からライブラリを削除するには、クラスタ内のすべてのインスタンスを再起動する必要があることに注意してください。

文字列のリストとして `.spec.postgresql.shared_preload_libraries` を介して追加の `shared_preload_libraries` を提供できます。オペレーターはそれらを自動的に管理するものとマージします。

マネージド拡張機能

前のセクションで予想したように、CloudNativePG は、一部の既知のサポートされている拡張機能について、`shared_preload_libraries` のコンテンツを自動的に管理します。現在のリストには次のものが含まれます。

- `auto_explain`
- `pg_stat_statements`
- `pgaudit`

これらのライブラリのいくつかは、それらを使用する前にデータベースに追加のオブジェクトを必要とし、通常は `CREATE EXTENSION` コマンドを介して管理されるビューや関数をデータベースで実行します (通常、`DROP EXTENSION` コマンドはこれらのオブジェクトを削除します)。

このようなライブラリの場合、CloudNativePG は、次のクエリで識別される、クラスター内の接続を受け入れるすべてのデータベースで拡張機能の作成と削除を自動的に処理します。

```
SELECT datname FROM pg_database WHERE datallowconn
```

注釈: 上記のクエリには、*template1* などのテンプレート データベースも含まれます。

---

auto_explain の有効化

:ref:``auto_explain` の有効化<*auto_explain* の有効化>`

拡張機能は、`EXPLAIN` を手動で実行することなく、遅いステートメントの実行計画を自動的にログに記録する手段を提供します (最適化されていないクエリを追跡するのに役立ちます)。

次の例の抜粋のように、`auto_explain.` で始まるパラメーターを構成に追加することで、`auto_explain` を有効にすることができます (完了するまでに 10 秒以上かかるクエリの実行計画を自動的にログに記録します)。

```
# ...
postgresql:
  parameters:
    auto_explain.log_min_duration: "10s"
# ...
```

注釈: auto_explain を有効にすると、パフォーマンスの問題が発生する可能性があります。 the auto explain documentation を参照してください

---

pg_stat_statements の有効化

:ref:``pg_stat_statements` の有効化<*pg_stat_statements* の有効化>`

拡張機能は、クエリをリアルタイムで監視するために PostgreSQL で利用できる最も重要な機能の 1 つです。

次の例の抜粋のように、`pg_stat_statements.` で始まるパラメーターを構成に追加することで、`pg_stat_statements` を有効にすることができます。

```
# ...
postgresql:
  parameters:
    pg_stat_statements.max: "10000"
    pg_stat_statements.track: all
# ...
```

前に説明したように、オペレーターは自動的に `pg_stat_statements` を `shared_preload_libraries` に追加し、各データベースで `CREATE EXTENSION IF NOT EXISTS pg_stat_statements` を実行して、`pg_stat_statements` ビューに対してクエリを実行できるようにします。

pgaudit の有効化

`pgaudit` 拡張機能は、標準の PostgreSQL ロギング機能を介して、詳細なセッションおよび/またはオブジェクトの監査ロギングを提供します。

CloudNativePG は、PostgreSQL クラスターで :ref:``pgaudit` の有効化<*pgaudit* の有効化>` を透過的かつネイティブにサポートします。詳細については、 :ref:``pgaudit` の有効化<*pgaudit* の有効化>` を参照してください。

次の例の抜粋のように、`pgaudit.` で始まるパラメーターを構成に追加することで、`pgaudit` を有効にすることができます。

```
#
postgresql:
  parameters:
```

(次のページに続く)

(前のページからの続き)

```
    pgaudit.log: "all, -misc"
    pgaudit.log_catalog: "off"
    pgaudit.log_parameter: "on"
    pgaudit.log_relation: "on"
#
```

4.16.2 pg_hba セクション

pg_hba は、ポッドで使用される pg_hba.conf を作成するために使用される PostgreSQL ホストベース認証ルールのリストです。

最初に一致したルールが認証に使用されるため、オペレーターによって生成された pg_hba.conf ファイルは、次の 4 つのセクションで構成されていることがわかります。

1. 決まったルール
2. ユーザー定義のルール
3. オプションの LDAP セクション
4. デフォルトのルール

固定ルール:

```
local all all peer

hostssl postgres streaming_replica all cert
hostssl replication streaming_replica all cert
```

デフォルトのルール:

```
host all all all <default-authentication-method>
```

PostgreSQL 14 以降、password_encryption データベースパラメータのデフォルト値は scram-sha-256 に設定されています。そのため、デフォルトの認証方法は、この PostgreSQL バージョンからの scram-sha-256 です。

PostgreSQL 13 以前では、デフォルトの認証方法として md5 が使用されます。

結果の pg_hba.conf は次のようになります。

```
  local all all peer

  hostssl postgres streaming_replica all cert
  hostssl replication streaming_replica all cert

  <user defined rules>
  <user defined LDAP>
```

(次のページに続く)

(前のページからの続き)

```
  host all all all scram-sha-256 # (or md5 for PostgreSQL version <= 13)

`more information on `pg_hba.conf` <https://www.postgresql.org/docs/current/auth-pg-
↪hba-conf.html>`__  については、PostgreSQL のドキュメントを参照してください。
```

次の抜粋のように、クラスター マニフェスト内で、pg_hba 行が spec.postgresql.pg_hba のリスト アイテムとして追加されます。

```
postgresql:
  pg_hba:
    - hostssl app app 10.244.0.0/16 md5
```

上記の例では、安全なチャネル (hostssl) 経由で MD5 パスワード認証 (必要に応じて scram-sha-256 を使用できます) を使用して、app ユーザーが app データベースにアクセスできるようにしています。

LDAP 設定

クラスター仕様の postgres セクションの下には、pg_hba.conf ファイルに追加されるルールに変換される LDAP 構成を定義するために使用できるオプションの ldap セクションがあります。

これは、LDAP セクションで server 、prefix 、suffix を指定する必要がある simple bind モードと、server 、baseDN 、binDN 、および LDAP パスワードを含むシークレットである bindPassword を指定する必要がある search+bind モードの 2 つのモードをサポートします。さらに、search+bind モードでは、searchFilter または searchAttribute を指定するオプションがあります。 searchAttribute が指定されていない場合、デフォルトの uid が使用されます。

さらに、どちらのモードでも、ldapscheme の scheme と port を指定できます。ただし、スキームもポートも必要ありません。

search+bind 用に入力されたこのセクションは、次のようになります。

```
postgresql:
  ldap:
    server: openldap.default.svc.cluster.local
    bindSearchAuth:
      baseDN: ou=org,dc=example,dc=com
      bindDN: cn=admin,dc=example,dc=com
      bindPassword:
        name: ldapBindPassword
        key: data
      searchAttribute: uid
```

4.16.3 構成の変更

Cluster リソースの postgresql セクションを編集することで、構成の変更を適用できます。

変更後、クラスターインスタンスはすぐに構成を再読み込みして変更を適用します。変更に再起動が必要なパラメーターが含まれる場合、オペレーターはローリングアップグレードを実行します。

4.16.4 動的共有メモリの設定

PostgreSQL は、 dynamic_shared_memory_type を通じて動的共有メモリ管理のいくつかの実装をサポートしています。

構成オプション。 CloudNativePG では、次の 2 つの値のいずれかに制限することをお勧めします。

- posix : shm_open (デフォルト設定) を使用して割り当てられた POSIX 共有メモリに依存します
- sysv : shmget 経由で割り当てられた System V 共有メモリに基づいています

PostgreSQL では、この設定は並列クエリでのメモリ割り当てに特に重要です。詳細はこちらの

> *thread from the `pgsql-general* mailing list <https://www.postgresql.org/message-id/CA%2BhUKGJOj7qzDLxeFPVvto8YEWop6FSQoTYPO9Z6Ee%3Di-nPS_Q%40mail.gmail.com>`__
> を参照してください。

POSIX 共有メモリ

ほとんどの場合、posix のデフォルト設定で十分です。これは、オペレーターが /dev/shm の下に shm という **メモリ バウンド** ``*EmptyDir*`` **ボリューム**を自動的にマウントすることを考慮した場合です。実行中の Postgres コンテナー内のそのようなボリュームのサイズは、次の方法で確認できます。

```
mount | grep shm
```

次のような出力が得られるはずです。

```
shm on /dev/shm type tmpfs (rw,nosuid,nodev,noexec,relatime,size= **** **)
```

System V 共有メモリ

Kubernetes クラスターの SHMMAX パラメーターと SHMALL パラメーターの値が十分に高い場合は、次のように設定することもできます。

```
dynamic_shared_memory_type: "sysv"
```

次のコマンドを実行して、PostgreSQL コンテナー内から SHMMAX /SHMALL を確認できます。

```
ipcs -lm
```

例えば：

```
- ----- Shared Memory Limits --------
max number of segments = 4096
max seg size (kbytes) = 18014398509465599
max total shared memory (kbytes) = 18014398509481980
min seg size (bytes) = 1
```

ご覧のとおり、max total shared memory の数が非常に多いため、dynamic_shared_memory_type を sysv に設定することをお勧めします。

別の方法は、次を実行することです。

```
cat /proc/sys/kernel/shmall
cat /proc/sys/kernel/shmmax
```

4.16.5 固定パラメータ

一部の PostgreSQL 構成パラメーターは、オペレーターが排他的に管理する必要があります。オペレーターは、ユーザーが Webhook を使用してそれらを設定できないようにします。

ユーザーは、postgresql セクションで次の構成パラメーターを設定することはできません。

- allow_system_table_mods
- archive_cleanup_command
- archive_command
- archive_mode
- bonjour
- bonjour_name
- cluster_name
- config_file
- data_directory
- data_sync_retry
- event_source
- external_pid_file
- full_page_writes
- hba_file
- hot_standby
- ident_file

- jit_provider
- listen_addresses
- log_destination
- log_directory
- log_file_mode
- log_filename
- log_rotation_age
- log_rotation_size
- log_truncate_on_rotation
- logging_collector
- port
- primary_conninfo
- primary_slot_name
- promote_trigger_file
- recovery_end_command
- recovery_min_apply_delay
- recovery_target
- recovery_target_action
- recovery_target_inclusive
- recovery_target_lsn
- recovery_target_name
- recovery_target_time
- recovery_target_timeline
- recovery_target_xid
- restart_after_crash
- restore_command
- shared_preload_libraries
- ssl
- ssl_ca_file
- ssl_cert_file
- ssl_ciphers

- ssl_crl_file
- ssl_dh_params_file
- ssl_ecdh_curve
- ssl_key_file
- ssl_max_protocol_version
- ssl_min_protocol_version
- ssl_passphrase_command
- ssl_passphrase_command_supports_reload
- ssl_prefer_server_ciphers
- stats_temp_directory
- synchronous_standby_names
- syslog_facility
- syslog_ident
- syslog_sequence_numbers
- syslog_split_messages
- unix_socket_directories
- unix_socket_group
- unix_socket_permissions
- wal_level
- wal_log_hints

4.17 オペレーター構成

CloudNativePG のオペレーターは、標準のデプロイマニフェストからインストールされ、構成パラダイムに関する規則に従います。ほとんどの場合はこれで問題ありませんが、次のようにデフォルトの動作を変更したいシナリオがいくつかあります。

- オペレーターによって作成され、クラスターリソースに設定されているすべてのリソースに継承されるアノテーションとラベルを定義する
- PostgreSQL の別のデフォルト イメージまたは追加のプルシークレットを定義する

デフォルトでは、オペレーターは cnpg-controller-manager という Kubernetes Deployment として cnpg-system 名前空間にインストールされます。

注釈: 以下の例では、オペレーターのデプロイメントのデフォルトの名前と名前空間を想定しています。

オペレーターの動作は、オペレーターのデプロイと同じ名前空間にあり、名前が cnpg-controller-manager-config の ConfigMap /Secret を介してカスタマイズできます。

重要: config の *ConfigMap /Secret* への変更は、オペレーターによって自動的に検出されないため、リロードする必要があります (以下を参照)。さらに、変更は、構成が再ロードされた後に作成されたリソースにのみ適用されます。

重要: オペレーターは、最初に ConfigMap 値を処理し、次に Secret をこの順序で処理します。その結果、パラメーターが両方の場所で定義されている場合、Secret のパラメーターが使用されます。

4.17.1 利用可能なオプション

オペレーターは、ConfigMap /Secret で定義される次の環境変数を探します。

名前 |説明 —- | ——— – INHERITED_ANNOTATIONS | Cluster メタデータで定義されると、ポッドを含むすべての生成されたリソースによって継承されるアノテーション名のリスト INHERITED_LABELS | Cluster メタデータで定義されると、ポッドを含むすべての生成されたリソースによって継承されるラベル名のリスト PULL_SECRET_NAME |オペレーターの名前空間で定義され、イメージのダウンロードに使用される追加のプルシークレットの名前 PVC がサイズ変更状態でスタックしている場合、Postgres ポッドを削除できるようにします。この機能は主に Azure 環境向けです (デフォルト false) ENABLE_INSTANCE_MANAGER_INPLACE_UPDATES | true に設定すると、オペレーターの更新後にインスタンスマネージャーのインプレース更新が有効になり、クラスターのローリング更新が回避されます (デフォルト false) MONITORING_QUERIES_CONFIGMAP |作成されたすべてのクラスターに適用される一連のデフォルト クエリ (キー queries で指定) を含む、オペレーターの名前空間内の ConfigMap の名前。作成されたすべてのクラスターに適用される一連のデフォルト クエリ (キー queries で指定) を含む、オペレーターの名前空間内のシークレットの名前

INHERITED_ANNOTATIONS と INHERITED_LABELS の値は、パスのようなワイルドカードをサポートします。たとえば、値 example.com/* は、値 example.com/one と example.com/two の両方に一致します。

PULL_SECRET_NAME パラメーターを使用して追加のプルシークレット名を指定すると、オペレーターはそのシークレットを使用して、作成されたすべての PostgreSQL クラスターのプル シークレットを作成します。そのシークレットは<cluster-name>-pull という名前になります。

オペレーターが PULL_SECRET_NAME シークレットを探す名前空間は、オペレーターをインストールした場所です。オペレーターがそのシークレットを見つけることができない場合、構成パラメーターは無視されます。

警告: 以前のバージョンのオペレーターは、PostgreSQL クラスターをデプロイする名前空間内に *PULL_SECRET_NAME* シークレットをコピーしていました。バージョン「1.11.0」から、以前の説明と一致するように動作が変更されました。以前のバージョンのオペレーターによって作成されたプルシークレットは使用されません。

4.17.2 オペレーター構成マップの定義

以下の例では、後でデプロイされる Cluster オブジェクトによって作成されたリソースによって継承されるラベル/アノテーション名を定義し、in-place updates for the instance manager を有効にすることによって、オペレーターの動作をカスタマイズします。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cnpg-controller-manager-config
  namespace: cnpg-system
data:
  INHERITED_ANNOTATIONS: categories
  INHERITED_LABELS: environment, workload, app
  ENABLE_INSTANCE_MANAGER_INPLACE_UPDATES: true
```

4.17.3 オペレーター シークレットの定義

以下の例では、後でデプロイされる Cluster オブジェクトによって作成されたリソースによって継承されるラベル/アノテーション名を定義し、in-place updates for the instance manager を有効にすることによって、オペレーターの動作をカスタマイズします。

```
apiVersion: v1
kind: Secret
metadata:
  name: cnpg-controller-manager-config
  namespace: cnpg-system
type: Opaque
stringData:
  INHERITED_ANNOTATIONS: categories
  INHERITED_LABELS: environment, workload, app
  ENABLE_INSTANCE_MANAGER_INPLACE_UPDATES: true
```

4.17.4 オペレータを再起動して設定をリロードします

変更を有効にするには、オペレーターポッドを再作成して構成マップをリロードする必要があります。マニフェストを使用して Kubernetes にオペレーターをインストールした場合は、以下を発行して実行できます。

```
kubectl rollout restart deployment \
    -n cnpg-system \
    cnpg-controller-manager
```

一般に、特定の名前空間が指定されている場合、次のコマンドを使用してオペレーターポッドを削除できます。

```
kubectl delete pods -n [NAMESPACE_NAME_HERE] \
  -l app.kubernetes.io/name=cloudnative-pg
```

警告: カスタマイズは、オペレーター デプロイメントのリロード後に作成された *Cluster* リソースにのみ適用されます。

上記の例に従って、Cluster 定義に categories 注釈と environment 、workload 、または app ラベルのいずれかが含まれている場合、これらはデプロイメントによって生成されたすべてのリソースによって継承されます。

4.17.5 PPROF HTTP サーバー

オペレーターは、localhost:6060 上の次のエンドポイントで PPROF HTTP サーバーを公開できます。

```
- `/debug/pprof/`. Responds to a request for "/debug/pprof/" with an HTML page␣
↪listing the available profiles

- `/debug/pprof/cmdline`. Responds with the running programs command line, with␣
↪arguments separated by NUL bytes.

- `/debug/pprof/profile`. Responds with the pprof-formatted cpu profile. Profiling␣
↪lasts for duration specified in seconds GET parameter, or for 30 seconds if not␣
↪specified.

- `/debug/pprof/symbol`. Looks up the program counters listed in the request,␣
↪responding with a table mapping program counters to function names.

- `/debug/pprof/trace`. Responds with the execution trace in binary form.  Tracing␣
↪lasts for duration specified in seconds GET parameter, or for 1 second if not␣
↪specified.
```

オペレーターを有効にするには、オペレーターのデプロイメントを編集してフラグ--pprof-server=true を追加する必要があります。

これを行うには、次のコマンドを実行します。

```
kubectl edit deployment -n cnpg-system cnpg-controller-manager
```

次に、編集ページでコンテナー引数を下にスクロールし、--pprof-server=true を追加します。例:

```
      containers:
      - args:
        - controller
```

(次のページに続く)

(前のページからの続き)

```
        - --enable-leader-election
        - --config-map-name=cnpg-controller-manager-config
        - --secret-name=cnpg-controller-manager-config
        - --log-level=info
        - --pprof-server=true # relevant line
        command:
        - /manager
```

``` 変更を保存すると、デプロイでロールアウトが実行され、新しいポッドで PPROF サーバーが有効になります。

ポッドが実行されたら、次のようにしてコンテナー内で実行できます。

kubectl exec -ti -n cnpg-system – bash

curl localhost:6060/debug/pprof/ ```

## 4.18 インスタンス Pod 構成

### 4.18.1 予測ボリューム

CloudNativePG は、`.spec.projectedVolumeTemplate` を介して Postgres ポッド内にカスタムファイルをマウントすることをサポートしています。これは、追加のデータファイルを必要とするいくつかの Postgres 機能と拡張機能に役立ちます。CloudNativePG では、`.spec.projectedVolumeTemplate` フィールドは kubernetes の **予測ボリューム** テンプレートであり、ユーザーは Postgres ポッドの `/projected` フォルダーの下に任意のデータをマウントできます。

既存の tls シークレット (sample-secret という名前) をファイルとして Postgres ポッドにマウントする方法の簡単な例を次に示します。sample-secret のシークレット キー `tls.crt` と `tls.key` の値は、Postgres ポッドのパス `/projected/certificate/tls.crt` と`/projected/certificate/tls.key` にファイルとしてマウントされます。

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-projected-volumes
spec:
  instances: 3
  projectedVolumeTemplate:
    sources:
      - secret:
          name: sample-secret
          items:
            - key: tls.crt
              path: certificate/tls.crt
```

(次のページに続く)
```

(前のページからの続き)

```
          - key: tls.key
            path: certificate/tls.key
  storage:
    size: 1Gi
```

Projected Volume テンプレートを使用して Secret と Configmap をマウントする完全な例は、cluster-example-projected-volume.yaml 配置マニフェストにあります。

4.18.2 環境変数

一部のシステム動作は、環境変数を使用してカスタマイズできます。 1 つの例は LDAPCONF 変数で、カスタム LDAP 構成ファイルを指す場合があります。もう 1 つの例は、PostgreSQL コンテナーで使用されるタイムゾーンを表す TZ 環境変数です。

CloudNativePG では、ユーザーはクラスター仕様の env および envFrom スタンザを介してカスタム環境変数を設定できます。

以下は、Australia/Sydney タイムゾーンをデフォルトのクラスターレベルのタイムゾーンとして使用する PostgreSQL クラスターの定義です。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3

  env:
  - name: TZ
    value: Australia/Sydney

  storage:
    size: 1Gi
```

envFrom スタンザは、ConfigMaps または Secrets を参照して、それらのコンテンツを環境変数として使用できます。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3

  envFrom:
```

(次のページに続く)

(前のページからの続き)

```
  - configMapRef:
      name: config-map-name
  - secretRef:
      name: secret-name

  storage:
    size: 1Gi
```

オペレーターは、次の環境変数の設定を許可していません:

- POD_NAME
- NAMESPACE
- 名前が PG で始まる環境変数。

env または envFrom セクションが変更されると、PostgreSQL Pod のローリング更新がトリガーされます。

env または envFrom セクションがシークレットまたは ConfigMap を参照する場合、オペレーターはそれらの変更を検出せず、ロールアウトをトリガーしません。Kubelet は Pod で同じ動作を使用し、ユーザーは Pod のロールアウトを手動でトリガーする必要があります。

4.19 保管所

ストレージは、データベースワークロードで最も重要なコンポーネントです。ストレージは常に利用可能で、拡張性があり、適切に機能し、一貫性と耐久性を保証する必要があります。仮想マシンやベアメタルなどの従来の環境に適用されるのと同じ期待と要件は、Kubernetes によって管理されるコンテナー コンテキストでも有効です。

重要: 動的にプロビジョニングされたストレージに関しては、Kubernetes には独自の特徴があります。これらには、**ストレージ クラス**、**永続ボリューム**、*永続ボリューム クレーム*が含まれます。 VM および物理サーバー上のデータベース ワークロードのストレージに関して長年にわたって構築してきたすべての貴重な知識に加えて、これらの概念を所有する必要があります。

ストレージにアクセスするには、主に次の 2 つの方法があります。

- **ネットワーク** : 直接的または間接的に (Kubernetes を実行しているホストにローカルにマウントされた NFS ボリュームを考えてください)
- **ローカル** : ポッドが実行されているノードに直接接続されています (これには、Kubernetes のベア メタルインストール上の直接接続されたディスクも含まれます)

Kubernetes で最も一般的な使用パターンであるネットワークストレージでは、従来の環境で発生する可能性があるのと同じスループットと待機時間の問題が発生します。これらは、複数のアプリケーションとの I/O 競合によってパフォーマンス結果の変動性が高まる共有環境で強調される可能性があります。

ローカル ストレージを使用すると、シェアード ナッシングアーキテクチャが可能になります。これは、より高い予測可能なパフォーマンスが保証されるため、高いトランザクションおよび超大規模データベース (VLDB) のワークロードにより適しています。

警告: CloudNativePG を使用して PostgreSQL クラスターをデプロイする前に、使用しているストレージがデータベース ワークロードに推奨されていることを確認してください。最初に fio によるストレージのベンチマーク などのツールを使用してストレージをベンチマークし、次に pgbench によるデータベースのベンチマーク を使用してデータベースをベンチマークすることにより、パフォーマンスの期待値を明確に設定することをお勧めします。

4.19.1 CloudNativePG のベンチマーク

EDB は、データベースを本番環境にデプロイする前に、制御された Kubernetes 環境で CloudNativePG をベンチマークするためのガイドラインと Helm チャートのオープン ソース セットである cnp-bench を維持しています。

簡単に言うと、cnp-bench は 2 つのレベルで動作するように設計されています。

- `fio` を使用して基礎となるストレージのパフォーマンスを測定し、シーケンシャル読み取り、シーケンシャル書き込み、ランダム読み取り、ランダム書き込みのスループットなどのデータベースワークロードに関連するメトリックを使用します。
- PostgreSQL とともに配布されているデフォルトのベンチマークツールを使用して、データベースのパフォーマンスを測定します: pgbench

---

重要: ストレージとデータベースの両方のパフォーマンスを測定することは、**データベースが運用に入る前に**実行する必要があるアクティビティです。ただし、このような結果は、計画段階 (キャパシティ プランニングなど) だけでなく、特に緊急事態 (この種のテストを実行する余裕がない場合) の生産ライフサイクルでも非常に価値があります。データベースは確かに時間の経過とともに変化し、進化し、データの分散も変化し、パフォーマンスに影響を与える可能性があります。シーケンシャルな読み取りまたは書き込みの理論上の最大スループットを知ることは、そのような状況で非常に役立つことがわかります. 特にシェアード ナッシング コンテキストでは、外部ワークロードの影響によって結果が変化することはありません。 **あなたのシステムを知り、ベンチマークしましょう。

4.19.2 永続ボリュームの要求

オペレーターは、`PGDATA` を格納することを目的として、PostgreSQL インスタンスごとに永続ボリューム要求 (PVC) を作成し、それを各 Pod にマウントします。

さらに、以下の *WAL のボリューム* で説明されているように、PostgreSQL Write-Ahead Log (WAL) を保存する別の PVC を使用したクラスターの作成をサポートしています。

CloudNativePG では、単一の PostgreSQL インスタンスにアタッチされたボリュームは **PVC グループ**として定義されます。

4.19.3 ストレージ クラスによる構成

PostgreSQL クラスのストレージを構成する簡単な方法は、次の例のように、特定のサイズのストレージを要求することです。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgresql-storage-class
spec:
  instances: 3
  storage:
    size: 1Gi
```

前の構成を使用すると、生成された PVC はデフォルトのストレージクラスで満たされます。ターゲットの Kubernetes クラスターにデフォルトのストレージ クラスがない場合、または PVC を既知のストレージ クラスで満たす必要がある場合でも、それをカスタムリソースに設定できます。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgresql-storage-class
spec:
  instances: 3
  storage:
    storageClass: standard
    size: 1Gi
```

重要: CloudNativePG は、ストレージ クラスに依存しないように設計されています。いつものように、運用に入る前に、制御された環境でストレージ クラスを適切にベンチマークすることをお勧めします。

4.19.4 PVC テンプレートによる設定

生成された PVC をさらにカスタマイズするには、次の例のように、カスタムリソース内に PVC テンプレートを提供できます。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgresql-pvc-template
spec:
  instances: 3
```

(次のページに続く)

(前のページからの続き)

```
  storage:
    pvcTemplate:
      accessModes:
        - ReadWriteOnce
      resources:
        requests:
          storage: 1Gi
      storageClassName: standard
      volumeMode: Filesystem
```

4.19.5 WAL のボリューム

デフォルトでは、PostgreSQL はすべてのデータをいわゆる PGDATA (ディレクトリ) に保存します。 PGDATA 内のコア ディレクトリの 1 つは pg_wal (歴史的に PostgreSQL では pg_xlog として知られている) で、データベースで発生したトランザクション変更のログがセグメントファイルの形式で含まれています。

ほとんどの場合、PGDATA が存在する同じボリュームに pg_wal を配置しても問題ありませんが、WAL を別のボリュームに格納することにはいくつかの利点があります。

- **I/O パフォーマンス**：PGDATA とは異なるストレージに WAL ファイルを格納することにより、PostgreSQL は WAL 操作 (通常はシーケンシャル書き込み) とデータ ファイル (テーブルやインデックスなど) に並列 I/O を利用できるため、垂直方向のパフォーマンスが向上します。スケーラビリティ
- **信頼性の向上**：WAL ファイル専用のディスクスペースを予約することで、PGDATA ボリュームのスペースが枯渇しても WAL 書き込みが妨げられず、PostgreSQL プライマリが正しくシャットダウンされることを常に確認できます。
- **より細かい制御**：PGDATA と pg_wal の両方に専用のスペースの量を定義し、WAL configuration とチェックポイントを微調整し、コスト最適化のために別のストレージクラスを使用することもできます
- **より良い I/O 監視**：PGDATA と pg_wal の両方で負荷とディスクの使用状況を常に監視し、たとえば PGDATA のサイズ変更が必要な場合に通知する適切なアラートを設定できます。

詳細については、PostgreSQL の公式ドキュメントを参照してください。

.spec.walStorage オプションを使用して、WAL 用に個別のボリュームを追加できます。これは、storage フィールドについて説明したのと同じ規則に従い、専用の PVC をプロビジョニングします。例えば：

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: separate-pgwal-volume
spec:
  instances: 3
  storage:
    size: 1Gi
```

(次のページに続く)

(前のページからの続き)

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

重要: *walStorage* の削除はサポートされていません。一度追加すると、WAL 用の別のボリュームを既存の Postgres クラスターから削除することはできません。

4.19.6 ボリューム拡張

Kubernetes は expanding PVCs を許可する API を公開します

これはデフォルトで有効になっていますが、基になる `StorageClass` でサポートする必要があります。

特定の `StorageClass` がボリューム拡張をサポートしているかどうかを確認するには、ストレージクラスの `allowVolumeExpansion` フィールドを読み取ることができます。

```
$ kubectl get storageclass -o jsonpath={$.allowVolumeExpansion} premium-storage
true
```

ボリューム拡張 Kubernetes 機能の使用

ストレージ クラスがボリューム拡張をサポートしている場合、`Cluster` のサイズ要件を変更できます。オペレーターはその変更をすべての PVC に適用します。

`StorageClass` が online volume resizing をサポートする場合

変更はすぐに Pod に適用されます。基になるストレージクラスがそれをサポートしていない場合は、Pod を削除してサイズ変更をトリガーする必要があります。

続行する最善の方法は、一度に 1 つの Pod を削除することです。レプリカから始めて、各 Pod がバックアップされるのを待ちます。

AKS での PVC ボリュームの拡張

現時点では、 Azure is not able to resize the PVC's volume without restarting the pod . CloudNativePG は、 **オペレーター構成** の `ENABLE_AZURE_PVC_UPDATES` 環境変数によってこの制限を克服しました。`'true'` に設定すると、CloudNativePG は Postgres クラスターのローリング更新をトリガーします。

または、次の回避策に従って、AKS でボリュームのサイズを手動で変更することもできます。

AKS でのボリューム拡張の回避策

次の手順に従って、AKS で PVC のサイズを手動で変更できます。例として、3 つのレプリカを持つクラスターがあるとします。

```
$ kubectl get pods
NAME                READY   STATUS    RESTARTS   AGE
cluster-example-1   1/1     Running   0          2m37s
cluster-example-2   1/1     Running   0          2m22s
cluster-example-3   1/1     Running   0          2m10s
```

Azure ディスクは、 docs で説明されているように、「接続されていない」状態でのみ拡張できます。つまり、PostgreSQL クラスターで使用されるディスクのサイズを変更するには、手動でロールアウトを実行する必要があります。最初に、ディスクにバインドされた PVC を使用して Pod をホストするノードを封鎖します。. これにより、Operator が Pod を再作成し、バックグラウンドディスクのサイズ変更が完了する前にすぐにその PVC に再接続することが防止されます。

最初のステップは、次のように、新しいサイズ (たとえば “2Gi”) を適用してクラスター定義を編集することです。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3

  storage:
    storageClass: default
    size: 2Gi
```

`cluster-example-1` Pod がクラスターのプライマリであると仮定すると、最初にレプリカに進むことができます。たとえば、`cluster-example-3` Pod をホストする kubernetes ノードを封鎖することから始めます。

```
kubectl cordon <node of cluster-example-3>
```

次に、`cluster-example-3` Pod を削除します。

```
$ kubectl delete pod/cluster-example-3
```

次のコマンドを実行します。

```
kubectl get pvc -w -o=jsonpath={.status.conditions[].message} cluster-example-3
```

次の出力が表示されるまで待ちます。

```
Waiting for user to (re-)start a Pod to finish file system resize of volume on node.
```

次に、ノードを uncordon できます。

```
kubectl uncordon <node of cluster-example-3>
```

Pod が正しく再作成され、Running および Ready 状態になるまで待ちます。

```
kubectl get pods -w cluster-example-3
cluster-example-3   0/1     Init:0/1   0          12m
cluster-example-3   1/1     Running    0          12m
```

次に、次のコマンドを実行して PVC 拡張を確認します。構成どおりに「2Gi」が返されます。

```
kubectl get pvc cluster-example-3 -o=jsonpath={.status.capacity.storage}
```

したがって、残りの Pod に対してこれらの手順を繰り返すことができます。

重要: *kubectl cnpg promote* (たとえば、*cluster-example-3* をプライマリに昇格させるための *kubectl cnpg promote cluster-example 3*) を使用して、スイッチオーバーによって新しいサイズ変更された Pod を昇格させた後、プライマリ インスタンスに関連付けられたディスクのサイズ変更を最後のディスクとして残してください。

ストレージを再作成しています

ストレージクラスがボリューム拡張をサポートしていない場合でも、ストレージを増やした新しい PVC を割り当ててからデータベースをそこに移動することにより、異なる PVC でクラスターを再生成できます。この操作は、クラスタに複数のノードが含まれている場合にのみ実行できます。

その際、次の例のように、`resizeInUseVolumes` フラグを無効にして、オペレーターが既存の PVC を変更できないようにする必要があります。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgresql-pvc-template
spec:
  instances: 3

  storage:
    storageClass: standard
    size: 1Gi
    resizeInUseVolumes: False
```

クラスタ全体を別のストレージ エリアに移動するには、すべての PVC とすべての Pod を再作成する必要があります。次の例のように、3 つのレプリカを持つクラスターがあるとします。

```
$ kubectl get pods
NAME                READY   STATUS    RESTARTS   AGE
cluster-example-1   1/1     Running   0          2m37s
cluster-example-2   1/1     Running   0          2m22s
cluster-example-3   1/1     Running   0          2m10s
```

別の PVC を使用してクラスターを再作成するには、クラスター定義を編集して `resizeInUseVolumes` を無効にし、別の PVC ですべてのインスタンスを再作成します。

例として、`cluster-example-3` のストレージを再作成するには、次のことができます。

```
$ kubectl delete pvc/cluster-example-3 pod/cluster-example-3
```

重要: 専用の WAL ボリュームを作成した場合、このプロセス中に両方の PVC を削除する必要があります。さらに、*.spec.walStorage* セクションの *resizeInUseVolumes* も無効にすることで、WAL ボリューム PVC を再生成する場合にも同じ手順が適用されます。

例 (WAL ストレージ専用の PVC が存在する場合):

```
$ kubectl delete pvc/cluster-example-3 pvc/cluster-example-3-wal pod/cluster-example-3
```

それが完了すると、オペレーターはサイズ変更された PVC を使用して別のレプリカの作成を調整します。

```
$ kubectl get pods
NAME                          READY   STATUS      RESTARTS   AGE
cluster-example-1             1/1     Running     0          5m58s
cluster-example-2             1/1     Running     0          5m43s
cluster-example-4-join-v2     0/1     Completed   0          17s
cluster-example-4             1/1     Running     0          10s
```

4.20 ラベルと注釈

Kubernetes のリソースはフラットな構造で編成されており、階層的な情報やリソース間の関係はありません。ただし、このようなリソースとオブジェクトは、**ラベル** と **注釈**を介して相互にリンクし、関連付けることができます。

要するに：

- アノテーションは、外部ツールとの統合を促進する目的で、追加の非識別情報をリソースに割り当てるために使用されます
- ラベルを使用してオブジェクトをグループ化し、Kubernetes のネイティブセレクター機能を介してそれらをクエリします

CloudNativePG のデプロイで使用する 1 つ以上のラベルや注釈を選択できます。次に、クラスターのメタデータでこれらのラベルやアノテーションを定義すると、オペレーターによって作成されたすべてのリソース (ポッドを含む) によってそれらが自動的に継承されるように、オペレーターを構成する必要があります。

注釈: ラベルとアノテーションの継承は、ポッド テンプレートなどの代替アプローチの代わりに CloudNativePG で採用されている手法です。

4.20.1 前提条件

デフォルトでは、クラスターのメタデータで定義されたラベルまたはアノテーションは、関連付けられたリソースによって継承されません。ラベル/注釈の継承を有効にするには、**オペレーター構成**セクションに記載されている手順に従う必要があります。

以下では、その例を続けて、以下に限定します。

- 注釈: `categories`
- ラベル: `app` 、`environment` 、`workload`

注釈: 注釈とラベルのコンテキストに最も適した名前を自由に選択してください。名前付けにワイルドカードを使用したり、*mycompany/* で始まるすべてのラベルまたは注釈を継承するために *mycompany/** のような戦略を採用したりできることを覚えておいてください。

4.20.2 クラスターのメタデータの定義

クラスターを定義するとき、リソースがデプロイされる **前**に、次のようにメタデータを適切に設定できます。

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
  annotations:
    categories: database
  labels:
    environment: production
    workload: database
    app: sso
spec:
    # ... <snip>
```

クラスターがデプロイされたら、たとえば、ラベルがポッドに正しく設定されていることを確認できます。

```
kubectl get pods --show-labels
```

4.20.3 現在の制限事項

現在、CloudNativePG は、ラベルまたは注釈の削除を自動的に伝達しません。したがって、アノテーションまたはラベルがクラスターから削除された場合、以前に基礎となる Pod に伝搬されていた場合、オペレーターは関連するリソースでそれを自動的に削除しません。

4.21 モニタリング

重要: Prometheus と Grafana のインストールは、このプロジェクトの範囲外です。システムに正しくインストールされていることを前提としています。ただし、実験のために、 Part 4 of the Quickstart で指示を提供します。

4.21.1 インスタンスのモニタリング

PostgreSQL インスタンスごとに、オペレーターは `metrics` という名前のポート 9187 で、HTTP 経由で **パート *4 - Prometheus* と *Grafana* を使用してクラスターを監視する**のメトリックのエクスポーターを提供します。オペレーターには **事前定義されたメトリックのセット**が付属しており、1 つ以上の `ConfigMap` または `Secret` リソースを介して追加のクエリを定義する高度に構成可能でカスタマイズ可能なシステムが付属しています (詳細については、以下の **ユーザー定義の指標** を参照してください)。

重要: バージョン 1.11 以降、CloudNativePG は *default-monitoring* と呼ばれる *ConfigMap* に by default a set of predefined metrics を既にインストールしています。

以下のセクション。

PostgreSQL で実行されるすべての監視クエリは次のとおりです。

- アトミック (クエリごとに 1 つのトランザクション)
- `pg_monitor` ロールで実行
- `application_name` を `cnpg_metrics_exporter` に設定して実行
- ユーザー `postgres` として実行

PostgreSQL documentation の「デフォルトロール」セクションを参照してください。

`pg_monitor` ロールの詳細については。

クエリは、デフォルトで、次のロジックに従って、`Cluster` リソースの指定された `bootstrap` メソッドによって定義されるように、**メインデータベース**に対して実行されます。

- initdb を使用: クエリは、指定されていない場合は initdb.database または app で指定されたデータベースに対してデフォルトで実行されます。
- recovery を使用: クエリは、指定されていない場合は recovery.database または postgres で指定されたデータベースに対してデフォルトで実行されます。
- pg_basebackup を使用: クエリは、指定されていない場合は pg_basebackup.database または postgres で指定されたデータベースに対してデフォルトで実行されます。

デフォルトのデータベースは、target_databases オプションで 1 つ以上のデータベースのリストを指定することにより、特定のユーザー定義メトリックに対して常にオーバーライドできます。

プロメテウス オペレーターの例

Prometheus Operator's リソース PodMonitor を使用して、特定の PostgreSQL

クラスターを監視できます。クラスターを正しく指す PodMonitor は、オペレーターがクラスター リソース自体で.spec.monitoring.enablePodMonitor を true に設定することで自動的に作成できます (デフォルト: false)。

重要: 自動的に作成された *PodMonitor* への変更は、次の調整サイクルでオペレーターによってオーバーライドされます。カスタマイズする必要がある場合は、以下で説明するように行うことができます。

特定のクラスターの PodMonitor を手動でデプロイするには、次のように定義し、必要に応じて変更します。

```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: cluster-example
spec:
  selector:
    matchLabels:
      "cnpg.io/cluster": cluster-example
  podMetricsEndpoints:
  - port: metrics
```

重要: 上記の例を、一意の名前と正しいクラスターの名前空間とラベルで変更してください (ここでは *cluster-example* を使用しています)。

重要: このドキュメントの以前のバージョンで使用されていたラベル *postgresql* は非推奨であり、将来削除される予定です。インスタンスを選択するには、代わりにラベル *cnpg.io/cluster* を使用してください。

事前定義されたメトリックのセット

すべての PostgreSQL インスタンスエクスポーターは、事前定義された一連のメトリックを自動的に公開します。これらのメトリックは、次の 2 つの主要なカテゴリに分類できます。

- cnpg_collector_* で始まる PostgreSQL 関連のメトリック。以下が含まれます。
- ディスク上の WAL ファイルの数と合計サイズ - アーカイブ ステータスフォルダー内の .ready および .done ファイルの数 - 要求された同期レプリカの最小数と最大数、および期待値と実際に観測された値- レプリカ クラスター モードが有効かどうかを示すフラグまたは無効 - 手動切り替えが必要かどうかを示すフラグ
- go_* から始まるランタイム関連のメトリックに移動します

以下は、インスタンスの localhost:9187/metrics エンドポイントによって返されるメトリックのサンプルです。ご覧のとおり、Prometheus 形式は自己文書化されています。

```
#  HELP cnpg_collector_collection_duration_seconds Collection time duration in seconds

#  TYPE cnpg_collector_collection_duration_seconds gauge

cnpg_collector_collection_duration_seconds{collector="Collect.up"} 0.0031393

#  HELP cnpg_collector_collections_total Total number of times PostgreSQL was␣
↪accessed for metrics.

#  TYPE cnpg_collector_collections_total counter

cnpg_collector_collections_total 2

#  HELP cnpg_collector_last_collection_error 1 if the last collection ended with␣
↪error, 0 otherwise.

#  TYPE cnpg_collector_last_collection_error gauge

cnpg_collector_last_collection_error 0

#  HELP cnpg_collector_manual_switchover_required 1 if a manual switchover is␣
↪required, 0 otherwise

#  TYPE cnpg_collector_manual_switchover_required gauge

cnpg_collector_manual_switchover_required 0

#  HELP cnpg_collector_pg_wal Total size in bytes of WAL segments in the /var/lib/
↪postgresql/data/pgdata/pg_wal directory  computed as (wal_segment_size * count)

#  TYPE cnpg_collector_pg_wal gauge
```

(次のページに続く)

(前のページからの続き)

```
cnpg_collector_pg_wal{value="count"} 9
cnpg_collector_pg_wal{value="slots_max"} NaN
cnpg_collector_pg_wal{value="keep"} 32
cnpg_collector_pg_wal{value="max"} 64
cnpg_collector_pg_wal{value="min"} 5
cnpg_collector_pg_wal{value="size"} 1.50994944e+08
cnpg_collector_pg_wal{value="volume_max"} 128
cnpg_collector_pg_wal{value="volume_size"} 2.147483648e+09

#  HELP cnpg_collector_pg_wal_archive_status Number of WAL segments in the /var/lib/
↪postgresql/data/pgdata/pg_wal/archive_status directory (ready, done)

#  TYPE cnpg_collector_pg_wal_archive_status gauge

cnpg_collector_pg_wal_archive_status{value="done"} 6
cnpg_collector_pg_wal_archive_status{value="ready"} 0

#  HELP cnpg_collector_replica_mode 1 if the cluster is in replica mode, 0 otherwise

#  TYPE cnpg_collector_replica_mode gauge

cnpg_collector_replica_mode 0

#  HELP cnpg_collector_sync_replicas Number of requested synchronous replicas␣
↪(synchronous_standby_names)

#  TYPE cnpg_collector_sync_replicas gauge

cnpg_collector_sync_replicas{value="expected"} 0
cnpg_collector_sync_replicas{value="max"} 0
cnpg_collector_sync_replicas{value="min"} 0
cnpg_collector_sync_replicas{value="observed"} 0

#  HELP cnpg_collector_up 1 if PostgreSQL is up, 0 otherwise.

#  TYPE cnpg_collector_up gauge

cnpg_collector_up{cluster="cluster-example"} 1

#  HELP cnpg_collector_postgres_version Postgres version

#  TYPE cnpg_collector_postgres_version gauge

cnpg_collector_postgres_version{cluster="cluster-example",full="13.4.0"} 13.4
```

(次のページに続く)

(前のページからの続き)

```
# HELP cnpg_collector_first_recoverability_point The first point of recoverability␣
↪for the cluster as a unix timestamp

# TYPE cnpg_collector_first_recoverability_point gauge

cnpg_collector_first_recoverability_point 1.63238406e+09

# HELP cnpg_collector_lo_pages Estimated number of pages in the pg_largeobject table

# TYPE cnpg_collector_lo_pages gauge

cnpg_collector_lo_pages{datname="app"} 0
cnpg_collector_lo_pages{datname="postgres"} 78

# HELP go_gc_duration_seconds A summary of the pause duration of garbage collection␣
↪cycles.

# TYPE go_gc_duration_seconds summary

go_gc_duration_seconds{quantile="0"} 5.01e-05
go_gc_duration_seconds{quantile="0.25"} 7.27e-05
go_gc_duration_seconds{quantile="0.5"} 0.0001748
go_gc_duration_seconds{quantile="0.75"} 0.0002959
go_gc_duration_seconds{quantile="1"} 0.0012776
go_gc_duration_seconds_sum 0.0035741
go_gc_duration_seconds_count 13

# HELP go_goroutines Number of goroutines that currently exist.

# TYPE go_goroutines gauge

go_goroutines 25

# HELP go_info Information about the Go environment.

# TYPE go_info gauge

go_info{version="go1.17.1"} 1

# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.

# TYPE go_memstats_alloc_bytes gauge

go_memstats_alloc_bytes 4.493744e+06
```

(次のページに続く)

(前のページからの続き)

```
# HELP go_memstats_alloc_bytes_total Total number of bytes allocated, even if freed.
# TYPE go_memstats_alloc_bytes_total counter
go_memstats_alloc_bytes_total 2.1698216e+07
# HELP go_memstats_buck_hash_sys_bytes Number of bytes used by the profiling bucket␣
↪hash table.
# TYPE go_memstats_buck_hash_sys_bytes gauge
go_memstats_buck_hash_sys_bytes 1.456234e+06
# HELP go_memstats_frees_total Total number of frees.
# TYPE go_memstats_frees_total counter
go_memstats_frees_total 172118
# HELP go_memstats_gc_cpu_fraction The fraction of this programs available CPU time␣
↪used by the GC since the program started.
# TYPE go_memstats_gc_cpu_fraction gauge
go_memstats_gc_cpu_fraction 1.0749468700447189e-05
# HELP go_memstats_gc_sys_bytes Number of bytes used for garbage collection system␣
↪metadata.
# TYPE go_memstats_gc_sys_bytes gauge
go_memstats_gc_sys_bytes 5.530048e+06
# HELP go_memstats_heap_alloc_bytes Number of heap bytes allocated and still in use.
# TYPE go_memstats_heap_alloc_bytes gauge
go_memstats_heap_alloc_bytes 4.493744e+06
# HELP go_memstats_heap_idle_bytes Number of heap bytes waiting to be used.
# TYPE go_memstats_heap_idle_bytes gauge
go_memstats_heap_idle_bytes 5.8236928e+07
```

(次のページに続く)

(前のページからの続き)

```
# HELP go_memstats_heap_inuse_bytes Number of heap bytes that are in use.
# TYPE go_memstats_heap_inuse_bytes gauge
go_memstats_heap_inuse_bytes 7.528448e+06
# HELP go_memstats_heap_objects Number of allocated objects.
# TYPE go_memstats_heap_objects gauge
go_memstats_heap_objects 26306
# HELP go_memstats_heap_released_bytes Number of heap bytes released to OS.
# TYPE go_memstats_heap_released_bytes gauge
go_memstats_heap_released_bytes 5.7401344e+07
# HELP go_memstats_heap_sys_bytes Number of heap bytes obtained from system.
# TYPE go_memstats_heap_sys_bytes gauge
go_memstats_heap_sys_bytes 6.5765376e+07
# HELP go_memstats_last_gc_time_seconds Number of seconds since 1970 of last garbage␣
↪collection.
# TYPE go_memstats_last_gc_time_seconds gauge
go_memstats_last_gc_time_seconds 1.6311727586032727e+09
# HELP go_memstats_lookups_total Total number of pointer lookups.
# TYPE go_memstats_lookups_total counter
go_memstats_lookups_total 0
# HELP go_memstats_mallocs_total Total number of mallocs.
# TYPE go_memstats_mallocs_total counter
go_memstats_mallocs_total 198424
# HELP go_memstats_mcache_inuse_bytes Number of bytes in use by mcache structures.
```

(次のページに続く)

(前のページからの続き)

```
#  TYPE go_memstats_mcache_inuse_bytes gauge

go_memstats_mcache_inuse_bytes 14400

#  HELP go_memstats_mcache_sys_bytes Number of bytes used for mcache structures␣
↪obtained from system.

#  TYPE go_memstats_mcache_sys_bytes gauge

go_memstats_mcache_sys_bytes 16384

#  HELP go_memstats_mspan_inuse_bytes Number of bytes in use by mspan structures.

#  TYPE go_memstats_mspan_inuse_bytes gauge

go_memstats_mspan_inuse_bytes 191896

#  HELP go_memstats_mspan_sys_bytes Number of bytes used for mspan structures␣
↪obtained from system.

#  TYPE go_memstats_mspan_sys_bytes gauge

go_memstats_mspan_sys_bytes 212992

#  HELP go_memstats_next_gc_bytes Number of heap bytes when next garbage collection␣
↪will take place.

#  TYPE go_memstats_next_gc_bytes gauge

go_memstats_next_gc_bytes 8.689632e+06

#  HELP go_memstats_other_sys_bytes Number of bytes used for other system allocations.

#  TYPE go_memstats_other_sys_bytes gauge

go_memstats_other_sys_bytes 2.566622e+06

#  HELP go_memstats_stack_inuse_bytes Number of bytes in use by the stack allocator.

#  TYPE go_memstats_stack_inuse_bytes gauge

go_memstats_stack_inuse_bytes 1.343488e+06

#  HELP go_memstats_stack_sys_bytes Number of bytes obtained from system for stack␣
```

(次のページに続く)

(前のページからの続き)

```
↪allocator.

#  TYPE go_memstats_stack_sys_bytes gauge

go_memstats_stack_sys_bytes 1.343488e+06

#  HELP go_memstats_sys_bytes Number of bytes obtained from system.

#  TYPE go_memstats_sys_bytes gauge

go_memstats_sys_bytes 7.6891144e+07

#  HELP go_threads Number of OS threads created.

#  TYPE go_threads gauge

go_threads 18
```

注釈: *cnpg_collector_postgres_version* は、PostgreSQL の *Major.Minor* バージョンを含む GaugeVec メトリックです。完全なセマンティック バージョン *Major.Minor.Patch* は、*full* というラベル フィールドの 1 つの中にあります。

ユーザー定義の指標

この機能は現在 **ベータ** 状態であり、フォーマットは PostgreSQL Prometheus Exporter の queries.yaml file に触発されています。

カスタムメトリックは、次の例のように、`.spec.monitoring.customQueriesConfigMap` または `customQueriesSecret` セクションの下の `Cluster` 定義で作成された `Configmap`/`Secret` を参照することで、ユーザーが定義できます。

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
  namespace: test
spec:
  instances: 3

  storage:
    size: 1Gi
```

(次のページに続く)

(前のページからの続き)

```
  monitoring:
    customQueriesConfigMap:
      - name: example-monitoring
        key: custom-queries
```

customQueriesConfigMap /customQueriesSecret セクションには、カスタム クエリが定義されているキーを指定する ConfigMap /Secret 参照のリストが含まれています。参照されるリソースは、**クラスター**リソースと同じ名前空間に作成する必要があることに注意してください。

注釈: インスタンスによって ConfigMap とシークレットを **自動的に** リロードする場合は、キー *cnpg.io/reload* のラベルを追加できます。そうしないと、*kubectl cnpg reload* サブコマンドを使用してインスタンスをリロードする必要があります。

重要: ユーザー定義のメトリックが既存のメトリックを上書きすると、インスタンス マネージャーはメッセージ *Query with the same name already found. Overwriting the existing one.* と上書きされたクエリ名を含むキー *queryName* を含む json 警告ログを出力します。

ユーザー定義指標の例

上記の Cluster の例で参照されている、単一のカスタム クエリを含む ConfigMap の例を次に示します。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-monitoring
  namespace: test
  labels:
    cnpg.io/reload: ""
data:
  custom-queries: |
    pg_replication:
      query: "SELECT CASE WHEN NOT pg_is_in_recovery()
              THEN 0
              ELSE GREATEST (0,
                EXTRACT(EPOCH FROM (now() - pg_last_xact_replay_timestamp())))
              END AS lag,
              pg_is_in_recovery() AS in_recovery,
              EXISTS (TABLE pg_stat_wal_receiver) AS is_wal_receiver_up,
              (SELECT count(*) FROM pg_stat_replication) AS streaming_replicas"

      metrics:
```

(次のページに続く)

(前のページからの続き)

```
      - lag:
          usage: "GAUGE"
          description: "Replication lag behind primary in seconds"
      - in_recovery:
          usage: "GAUGE"
          description: "Whether the instance is in recovery"
      - is_wal_receiver_up:
          usage: "GAUGE"
          description: "Whether the instance wal_receiver is up"
      - streaming_replicas:
          usage: "GAUGE"
          description: "Number of streaming replicas connected to the instance"
```

基本的な監視クエリのリストは、 default-monitoring.yaml にあります。

CloudNativePG デプロイメントに既にインストールされています (
メトリックのデフォルト セット を参照)。

複数のデータベースで実行されるユーザー定義メトリックの例

`target_databases` オプションが複数のデータベースをリストする場合、メトリックはそれらのそれぞれから収集されます。

`target_databases` のリストで **シェルのようなパターン** (つまり、`*`、`?`、または `[]` を含む) を指定することにより、特定のクエリに対してデータベースの自動検出を有効にすることができます。提供されている場合、オペレーターは、`SELECT datname FROM pg_database WHERE datallowconn AND NOT datistemplate` の実行によって返されたすべてのデータベースを追加し、 path.Match() ルールに従ってパターンを照合することにより、ターゲットデータベースのリストを拡張します。

注釈: * 文字には yaml の special meaning があるため、このようなパターンが含まれる場合は *target_databases* 値を引用 (`"*"`) する必要があります。

次の例のように `current_database()` 関数を使用するなど、返されるラベルにデータベースの名前を常に含めることをお勧めします。

```
some_query:
  query: |
    SELECT
     current_database() as datname,
     count(*) as rows
    FROM some_table
  metrics:
    - datname:
        usage: "LABEL"
```

(次のページに続く)

(前のページからの続き)

```
        description: "Name of current database"
    - rows:
        usage: "GAUGE"
        description: "number of rows"
  target_databases:
    - albert
    - bb
    - freddie
```

これにより、次のメトリックが公開されます。

```
cnpg_some_query_rows{datname="albert"} 2
cnpg_some_query_rows{datname="bb"} 5
cnpg_some_query_rows{datname="freddie"} 10
```

template1 データベースでも実行される、自動検出が有効になっているクエリの例を次に示します (それ以外の場合、前述のクエリでは返されません)。

```
some_query:
  query: |
    SELECT
     current_database() as datname,
     count(*) as rows
    FROM some_table
  metrics:
    - datname:
        usage: "LABEL"
        description: "Name of current database"
    - rows:
        usage: "GAUGE"
        description: "number of rows"
  target_databases:
    - "*"
    - "template1"
```

上記の例では、次のメトリックが生成されます (データベースが存在する場合)。

```
cnpg_some_query_rows{datname="albert"} 2
cnpg_some_query_rows{datname="bb"} 5
cnpg_some_query_rows{datname="freddie"} 10
cnpg_some_query_rows{datname="template1"} 7
cnpg_some_query_rows{datname="postgres"} 42
```

ユーザー定義メトリックの構造

すべてのカスタム クエリには、次の基本構造があります。

```
<MetricName>:
      query: "<SQLQuery>"
      metrics:
        - <ColumnName>:
            usage: "<MetricType>"
            description: "<MetricDescription>"
```

使用可能なすべてのフィールドの簡単な説明を次に示します。

- `<MetricName>` : Prometheus メトリックの名前 - `query` : ターゲット データベースで実行してメトリックを生成する SQL クエリ - `primary` : プライマリ インスタンスでのみクエリを実行するかどうか - `master` : `primary` と同じ (Prometheus PostgreSQL エクスポーターの構文との互換性のため - 非推奨) - `runonserver` : クエリを実行する必要がある PostgreSQL のバージョンを制限するセマンティック バージョン範囲 (例: `">=11.0.0"` または `">=12.0.0 <=15.0.0"`) - `target_databases` : `query` を実行するデータベースのリスト、または shell-like pattern

自動検出を有効にします。指定されている場合、デフォルトのデータベースを上書きします。- `metrics` : エクスポートされたすべての列のリストを含むセクション。次のように定義されます: - `<ColumnName>` : クエリによって返される列の名前 - `usage` : 以下で説明する値の 1 つ - `description` : メトリックの説明 - `metrics_mapping` : オプションの列マッピング `usage` が `MAPPEDMETRIC` に設定されている場合

`usage` の可能な値は次のとおりです。

表 5: :header: Column Usage Label,Description :widths: 10,30 :align: left :class: longtable

DISCARD	この列は無視する必要があります
LABEL	この列をラベルとして使用
COUNTER	この列をカウンターとして使用します
GAUGE	この列をゲージとして使用する
MAPPEDMETRIC	提供されたテキスト値のマッピングでこの列を使用します
DURATION	この列をテキストの長さ (ミリ秒単位) として使用します
HISTOGRAM	この列をヒストグラムとして使用します
Metric Types にアクセスしてください。	

詳細については、Prometheus のドキュメントを参照してください。

ユーザー定義メトリックの出力

カスタム定義のメトリックは、Prometheus エクスポーター エンドポイント (:9187/metrics) によって次の形式で返されます。

```
cnpg_<MetricName>_<ColumnName>{<LabelColumnName>=<LabelColumnValue> ... }
↪<ColumnValue>
```

注釈: *LabelColumnName* は、*usage* が *LABEL* に設定され、*Value* が設定されたメトリックです。

上記の pg_replication の例を考慮すると、エクスポーターのエンドポイントは、呼び出されると次の出力を返します。

```
#  HELP cnpg_pg_replication_in_recovery Whether the instance is in recovery

#  TYPE cnpg_pg_replication_in_recovery gauge

cnpg_pg_replication_in_recovery 0
#  HELP cnpg_pg_replication_lag Replication lag behind primary in seconds

#  TYPE cnpg_pg_replication_lag gauge

cnpg_pg_replication_lag 0
#  HELP cnpg_pg_replication_streaming_replicas Number of streaming replicas connected␣
↪to the instance

#  TYPE cnpg_pg_replication_streaming_replicas gauge

cnpg_pg_replication_streaming_replicas 2
#  HELP cnpg_pg_replication_is_wal_receiver_up Whether the instance wal_receiver is up

#  TYPE cnpg_pg_replication_is_wal_receiver_up gauge

cnpg_pg_replication_is_wal_receiver_up 0
```

メトリックのデフォルト セット

オペレーターは、オペレーターの名前空間内の ConfigMap またはシークレットで定義された一連の監視クエリをクラスターに自動的に挿入するように構成できます。

> **オペレーター構成** の MONITORING_QUERIES_CONFIGMAP または

MONITORING_QUERIES_SECRET キーを、それぞれ ConfigMap または Secret の名前に設定する必要があります。その後、オペレーターは queries キーの内容を使用します。

queries コンテンツへの変更は、それを使用してデプロイされたすべてのクラスターにすぐに反映されます。

オペレーターのインストールマニフェストには、cnpg-default-monitoring と呼ばれる定義済みの ConfigMap が付属しており、すべてのクラスターで使用されます。MONITORING_QUERIES_CONFIGMAP は、オペレーター構成でデフォルトで cnpg-default-monitoring に設定されています。

デフォルトのメトリック セットを無効にする場合は、次のことができます。

- オペレーター レベルで無効にします。オペレーター ConfigMap で、MONITORING_QUERIES_CONFIGMAP /MONITORING_QUERIES_SECRET キーを "" (空の文字列) に設定します。オペレーター ConfigMap を変更するには、オペレーターの再起動が必要です。
- 特定のクラスターに対して無効にする: クラスターで.spec.monitoring.disableDefaultQueries を true に設定します。

重要: *MONITORING_QUERIES_CONFIGMAP /MONITORING_QUERIES_SECRET* で指定された ConfigMap または Secret は、固定名 *cnpg-default-monitoring* で常にクラスターの名前空間にコピーされます。そのため、デフォルトのメトリクスを使用する場合は、クラスターの名前空間でこの名前の ConfigMap を作成しないでください。

Prometheus Postgres エクスポーターとの違い

CloudNativePG は PostgreSQL Prometheus Exporter に触発されていますが、いくつかの違いがあります。特に、cache_seconds フィールドは CloudNativePG のエクスポーターに実装されていません。

4.21.2 オペレーターの監視

オペレーターは、metrics という名前のポート 8080 で、HTTP 経由で
パート *4 - Prometheus と Grafana* を使用してクラスターを監視する メトリックを内部的に公開します。

以下のセクション。

現在、オペレーターはデフォルトの kubebuilder メトリクスを公開しています。詳細については、kubebuilder documentation を参照してください。

プロメテウス オペレーターの例

オペレーターのデプロイは、次の PodMonitor を定義することにより、 プロメテウス オペレーターの例 を使用して監視できます。

リソース：

```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: cnpg-controller-manager
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: cloudnative-pg
  podMetricsEndpoints:
    - port: metrics
```

4.21.3 エクスポートされたメトリクスを検査する方法

このセクションでは、同じ名前空間で curl を実行している一時ポッドを使用して、特定の PostgreSQL インスタンスマネージャー (プライマリまたはレプリカ) またはオペレーターによってエクスポートされたメトリクスを検査する方法に関する基本的な手順を説明します。

注釈: 以下の例では、PostgreSQL クラスターと共に、デフォルトの名前空間で作業していると想定しています。基本的な Kubernetes の知識を適用して、この例をユースケースに自由に適応させてください。

次の内容で curl.yaml ファイルを作成します。

```
apiVersion: v1
kind: Pod
metadata:
  name: curl
spec:
  containers:
  - name: curl
    image: curlimages/curl:7.84.0
    command: [sleep, 3600]
```

次に、ポッドを作成します。

```
kubectl apply -f curl.yaml
```

インスタンスによってエクスポートされたメトリックを検査する場合は、ターゲットポッドのポート 9187 に接続する必要があります。これは、実行する一般的なコマンドです (ポッドに正しい IP を使用していることを確認してください)。

```
kubectl exec -ti curl -- curl -s <pod_ip>:9187/metrics
```

たとえば、PostgreSQL クラスターの名前が cluster-example で、クラスター内の最初のポッドのエクスポートされたメトリックを取得する場合は、次のコマンドを実行して、そのポッドの IP をプログラムで取

得できます。

```
POD_IP=$(kubectl get pod cluster-example-1 --template {{.status.podIP}})
```

そして実行します：

```
kubectl exec -ti curl -- curl -s ${POD_IP}:9187/metrics
```

オペレーターのメトリクスにアクセスする場合は、オペレーターが実行されているポッドをポイントし、TCP ポート 8080 をターゲットとして使用する必要があります。

検査の最後に、必ず `curl` ポッドを削除してください。

```
kubectl delete -f curl.yaml
```

4.21.4 補助リソース

重要:

> これらのリソースは説明と実験のために提供されており、本番システムに対する推奨事項を表すものではありません

doc/src/samples/monitoring/ で

ディレクトリには、可観測性のための一連のサンプル ファイルがあります。

Part 4 of the quickstart を参照してください

コンテキストのセクション:

- `kube-stack-config.yaml` : kube-stack ヘルム チャートインストール用の構成ファイル。 Prometheus がすべての PodMonitor リソースをリッスンするようにします。
- `cnpg-prometheusrule.yaml` : CloudNativePG のアラートを含む `PrometheusRule` 。注: これには、通知サービスとの相互運用は含まれません。 Prometheus documentation を参照してください。
- `grafana-configmap.yaml` : サンプル CloudNativePG ダッシュボードの定義を含む ConfigMap。定義内のラベルに注意してください。これにより、Grafana デプロイメントが ConfigMap を確実に見つけることができます。

さらに、参照用に、Grafana ダッシュボードと Prometheus アラートルールの「生の」ソースを提供します。

- `alerts.yaml` : アラート付きの Prometheus ルール
- `grafana-dashboard.json` : ネイティブ Grafana JSON としての CloudNativePG ダッシュボード。

`kube-prometheus-stack` の構成については、`kube-stack-config.yaml` で提供するものよりも他のフィールドと設定を使用できることに注意してください。

`helm show values prometheus-community/kube-prometheus-stack` を実行して表示できます。詳細については、 kube-prometheus-stack を参照してください。

ページ。

4.22 ロギング

オペレーターは、PostgreSQL ログを含む標準出力に JSON 形式で直接ログインするように設計されています。

各ログ エントリには次のフィールドがあります。

- `level` : ログ レベル (`info` 、`notice` 、…)
- `ts` : タイムスタンプ (マイクロ秒のエポック)
- `logger` : レコードのタイプ (例: `postgres` または `pg_controldata`)
- `msg` : メッセージが JSON 形式で解析される場合の実際のメッセージまたはキーワード `record`
- `record` : 実際のレコード (`logger` タイプによって構造が異なります)
- `logging_podName` : ログが生成されたポッド

警告: ログの長期保存と管理はオペレーターの権限外であり、Kubernetes インストールのレベルで提供する必要があります。 Kubernetes Logging Architecture を参照してください。

ドキュメンテーション。

4.22.1 オペレーターログ

ログ レベルは、オプション `logLevel` を使用してクラスター仕様で指定でき、`error` 、`warning` 、`info` (デフォルト)、`debug` 、または `trace` のいずれかに設定できます。

現時点では、ログレベルはインスタンスの起動時にのみ設定でき、実行時に変更することはできません。クラスターの開始後にクラスター仕様で値が変更された場合、これは新しい Pod でのみ有効になり、古い Pod では有効になりません。

4.22.2 PostgreSQL ログ

PostgreSQL ログの各エントリは、`postgres` に設定された `logger` キーと、次の例で説明されている構造を持つ JSON オブジェクトです。

```
{
  "level": "info",
  "ts": 1619781249.7188137,
  "logger": "postgres",
  "msg": "record",
```

(次のページに続く)

(前のページからの続き)


```
  "record": {
    "log_time": "2021-04-30 11:14:09.718 UTC",
    "user_name": "",
    "database_name": "",
    "process_id": "25",
    "connection_from": "",
    "session_id": "608be681.19",
    "session_line_num": "1",
    "command_tag": "",
    "session_start_time": "2021-04-30 11:14:09 UTC",
    "virtual_transaction_id": "",
    "transaction_id": "0",
    "error_severity": "LOG",
    "sql_state_code": "00000",
    "message": "database system was interrupted; last known up at 2021-04-30 11:14:07␣
↪UTC",
    "detail": "",
    "hint": "",
    "internal_query": "",
    "internal_query_pos": "",
    "context": "",
    "query": "",
    "query_pos": "",
    "location": "",
    "application_name": "",
    "backend_type": "startup"
  },
  "logging_pod": "cluster-example-1",
}
```


内部的には、オペレーターは PostgreSQL CSV ログ形式に依存しています。CSV log format の詳細については、PostgreSQL のドキュメントを参照してください。

4.22.3 PGAudit ログ

CloudNativePG は、PostgreSQL クラスターで :ref:``pgaudit` の有効化<*pgaudit* の有効化>` を透過的かつネイティブにサポートします。

必要な `pgaudit` パラメーターをクラスターの構成の `postgresql` セクションに追加するだけです。

重要: *shared_preload_libraries* に PGAudit ライブラリを追加する必要はありません。ライブラリは、postgresql 構成内の *pgaudit.** パラメーターの存在に基づいて、CloudNativePG によって自動的に追加されます。オペレーターは、*shared_preload_libraries* からのライブラリの追加と削除を検出して管理します。

オペレーターは、クラスター内の使用可能なすべてのデータベースから拡張機能を作成および削除することも行います。

重要: CloudNativePG は、接続を受け入れるクラスター内のすべてのデータベースで *CREATE EXTENSION* および *DROP EXTENSION* コマンドを実行します。

以下は、要求された構成で `pgaudit` が有効になる PostgreSQL 13 `Cluster` 展開の例です。

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  imageName: ghcr.io/cloudnative-pg/postgresql:13

  postgresql:
    parameters:
      "pgaudit.log": "all, -misc"
      "pgaudit.log_catalog": "off"
      "pgaudit.log_parameter": "on"
      "pgaudit.log_relation": "on"

  storage:
    size: 1Gi
```

次に、PGAudit によって返された監査 CSV ログエントリが解析され、残りのすべてのログと同様に、JSON 形式で stdout にルーティングされます。

- `.logger` は `pgaudit` に設定されます
- `.msg` は `record` に設定されます
- `.record` には、`logging_collector` ログと同様に、解析されたレコード全体が JSON オブジェクトとして含まれます - JSON オブジェクトとしてフォーマットされた PGAudit CSV メッセージを含む`.record.audit` を除いて

以下の例を参照してください。

```yaml
{
  "level": "info",
  "ts": 1627394507.8814096,
  "logger": "pgaudit",
  "msg": "record",
  "record": {
    "log_time": "2021-07-27 14:01:47.881 UTC",
    "user_name": "postgres",
```

(次のページに続く)

(前のページからの続き)


```
    "database_name": "postgres",
    "process_id": "203",
    "connection_from": "[local]",
    "session_id": "610011cb.cb",
    "session_line_num": "1",
    "command_tag": "SELECT",
    "session_start_time": "2021-07-27 14:01:47 UTC",
    "virtual_transaction_id": "3/336",
    "transaction_id": "0",
    "error_severity": "LOG",
    "sql_state_code": "00000",
    "backend_type": "client backend",
    "audit": {
      "audit_type": "SESSION",
      "statement_id": "1",
      "substatement_id": "1",
      "class": "READ",
      "command": "SELECT FOR KEY SHARE",
      "statement": "SELECT pg_current_wal_lsn()",
      "parameter": "<none>"
    }
  },
  "logging_pod": "cluster-example-1",
}
```


レコードの各フィールドの詳細については、 PGAudit documentation を参照してください。

4.22.4 その他のログ

オペレーターとそのインスタンスによって生成されるすべてのログは JSON 形式であり、それらを生成したプロセスに応じて `logger` が設定されます。したがって、可能なすべての `logger` 値は次のとおりです。

- `barman-cloud-wal-archive` : `barman-cloud-wal-archive` から直接
- `barman-cloud-wal-restore` : `barman-cloud-wal-restore` から直接
- `initdb` : `initdb` の実行から
- `pg_basebackup` : `pg_basebackup` の実行から
- `pg_controldata` : `pg_controldata` の実行から
- `pg_ctl` : `pg_ctl` サブコマンドの実行から
- `pg_rewind` : `pg_rewind` の実行から
- `pgaudit` : PGAudit 拡張機能から
- `postgres` : `postgres` インスタンスから (`record` とは異なる `msg` を持つ)

- wal-archive : インスタンス マネージャーの wal-archive サブコマンドから
- wal-restore : インスタンス マネージャーの wal-restore サブコマンドから

前述の構造を持つ postgres を除いて、他のすべての可能な値には、ログに記録されるエスケープされたメッセージに設定された msg があります。

4.23 証明書

CloudNativePG は、TLS 証明書をネイティブにサポートするように設計されています。 Cluster をセットアップするために、オペレーターは以下を必要とします。

- サーバー認証局 (CA) 証明書
- サーバー認証局によって署名されたサーバー TLS 証明書
- クライアント認証局証明書
- クライアント認証局によって生成されたストリーミングレプリケーションクライアント証明書

注釈: クラスターで使用されるすべてのシークレットとその有効期限は、クラスターのステータスで確認できます。

CloudNativePG は TLS 証明書に関して非常に柔軟で、主に 2 つのモードで動作します。

1. **operator managed** :

   証明書は完全に自動化された方法でオペレーターによって内部的に管理され、CloudNativePG によって作成された CA を使用して署名されます。

2. **user provided** :

   証明書はオペレーターの外部で生成され、シークレットとしてクラスター定義にインポートされます- CloudNativePG はそれ自体を cert-manager と統合します (以下の例を参照)

証明書の一部のみが CNPG の外部で生成されるハイブリッドアプローチを選択することもできます。

4.23.1 オペレーター管理モード

デフォルトでは、オペレーターは単一の認証局を生成し、それをクライアントとサーバーの両方の証明書に使用します。その後、認証局は自動的に管理および更新されます。

サーバー証明書

サーバー CA シークレット

オペレーターは自己署名 CA を生成し、次のキーを含む汎用シークレットに格納します。

- `ca.crt` : サーバー証明書の検証に使用される CA 証明書。クライアントの接続文字列で `sslrootcert` として使用されます。
- `ca.key` : サーバー SSL 証明書に自動的に署名するために使用されるキー

サーバー TLS シークレット

オペレーターは、生成された自己署名 CA を使用してサーバー TLS 証明書に署名します。この証明書は、タイプ `kubernetes.io/tls` の Secret に格納され、インスタンスによって `ssl_cert_file` および `ssl_key_file` として使用されるように構成されているため、クライアントは ID を確認して安全に接続できます。

サーバーの代替 DNS 名

デフォルトのものに加えて、生成されたサーバー TLS シークレットの一部となる DNS サーバーの代替名を指定できます。

クライアント証明書

クライアント CA シークレット

サーバー CA と同じ自己署名 CA がデフォルトで使用されます。パブリック部分は、署名されたクライアント証明書を検証できるようにするために、すべてのインスタンスに `ssl_ca_file` として渡されます。秘密鍵は同じシークレットに保存され、`kubectl cnpg` プラグインによって生成されたクライアント証明書に署名するために使用されます。

クライアント streaming_replica 証明書

オペレーターは、生成された自己署名 CA を使用して、ユーザー `streaming_replica` のクライアント証明書に署名し、タイプ `kubernetes.io/tls` のシークレットに格納します。この証明書は、レプリカの接続文字列で `sslcert` および `sslkey` として渡され、プライマリインスタンスに安全に接続できるようになります。

4.23.2 ユーザー提供の証明書モード

サーバー証明書

必要に応じて、2 つのサーバー証明書を提供し、証明書マネージャーの例などの別のコンポーネントを使用して生成することもできます。クラスターにカスタムサーバー TLS 証明書を使用するには、次のパラメーターを指定する必要があります。

- `serverTLSSecret` : サーバー TLS 証明書を含むタイプ `kubernetes.io/tls` の Secret の名前。標準の `tls.crt` と `tls.key` キーの両方が含まれている必要があります。
- `serverCASecret` : `ca.crt` キーを含むシークレットの名前。

注釈: オペレーターは引き続き、クライアント証明書に関連する 2 つのシークレットを作成および管理します。

注釈: インスタンスによって ConfigMap とシークレットを **自動的に** リロードする場合は、キー *cnpg.io/reload* のラベルを追加できます。そうしないと、*kubectl cnpg reload* サブコマンドを使用してインスタンスをリロードする必要があります。

完全な例については、以下を参照してください。

例

次のファイルがあるとします。

- `server-ca.crt` : サーバーの TLS 証明書に署名した CA の証明書。
- `server.crt` : サーバー TLS 証明書の証明書。
- `server.key` : サーバー TLS 証明書の秘密鍵。

CA 証明書を含むシークレットを作成します。

```
kubectl create secret generic my-postgresql-server-ca \
  --from-file=ca.crt=./server-ca.crt
```

TLS 証明書を使用してシークレットを作成します。

```
kubectl create secret tls my-postgresql-server \
  --cert=./server.crt --key=./server.key
```

これらのシークレットを参照する `Cluster` を作成します。

```
kubectl apply -f - <<EOF
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  certificates:
    serverCASecret: my-postgresql-server-ca
    serverTLSSecret: my-postgresql-server
  storage:
    storageClass: standard
    size: 1Gi
EOF
```

新しいクラスターは、提供されたサーバー証明書を TLS 接続に使用します。

証明書マネージャーの例

証明書マネージャーの例 を使用して自己署名 CA をセットアップし、必要な TLS サーバー証明書を生成する方法の簡単な例を次に示します。

```
- --
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: selfsigned-issuer
spec:
  selfSigned: {}
- --
apiVersion: v1
kind: Secret
metadata:
  name: my-postgres-server-cert
  labels:
    cnpg.io/reload: ""
- --
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: my-postgres-server-cert
spec:
  secretName: my-postgres-server-cert
  usages:
```

(次のページに続く)

(前のページからの続き)

```
    - server auth
  dnsNames:
    - cluster-example-lb.internal.mydomain.net
    - cluster-example-rw
    - cluster-example-rw.default
    - cluster-example-rw.default.svc
    - cluster-example-r
    - cluster-example-r.default
    - cluster-example-r.default.svc
    - cluster-example-ro
    - cluster-example-ro.default
    - cluster-example-ro.default.svc
  issuerRef:
    name: selfsigned-issuer
    kind: Issuer
    group: cert-manager.io
```

my-postgres-server-cert という名前のシークレットが cert-manager によって作成され、必要なすべてのファイルが含まれており、次のようにクラスターから参照できます。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  certificates:
    serverTLSSecret: my-postgres-server-cert
    serverCASecret: my-postgres-server-cert
  storage:
    size: 1Gi
```

cert-manager を使用してサーバーとクライアントの両方の CA と証明書を管理する完全な例は、 cluster-example-cert-manager.yaml 配置マニフェストにあります。

クライアント証明書

必要に応じて、2 つのクライアント証明書を提供して、 証明書マネージャーの例 や hashicorp vault

などの別のコンポーネントを使用して生成することもできます。カスタム CA を使用してクラスターのクライアント証明書を検証するには、次のパラメーターを指定する必要があります。

- replicationTLSSecret : ユーザー streaming_replica のクライアント証明書を含むタイプ kubernetes.io/tls の Secret の名前。標準の tls.crt と tls.key キーの両方が含まれている必要があります。

- clientCASecret : クライアント証明書の検証に使用する CA の ca.crt キーを含むシークレットの名前。

注釈: オペレーターは、サーバー証明書に関連する 2 つのシークレットを引き続き作成および管理します。

注釈: クラスターはクライアント CA 秘密鍵を管理していないため、*kubectl cnpg certificate* を使用してクライアント証明書を生成することはできなくなりました。

注釈: インスタンスによって ConfigMap とシークレットを **自動的に** リロードする場合は、キー *cnpg.io/reload* のラベルを追加できます。そうしないと、*kubectl cnpg reload* サブコマンドを使用してインスタンスをリロードする必要があります。

証明書マネージャーの例

証明書マネージャーの例 を使用して自己署名 CA をセットアップし、必要な TLS

サーバー証明書を生成する方法の簡単な例を次に示します。

```
- --
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: selfsigned-issuer
spec:
  selfSigned: {}
- --
apiVersion: v1
kind: Secret
metadata:
  name: my-postgres-client-cert
  labels:
    cnpg.io/reload: ""
- --
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: my-postgres-client-cert
spec:
  secretName: my-postgres-client-cert
  usages:
    - client auth
```

(次のページに続く)

(前のページからの続き)

```
  commonName: streaming_replica
  issuerRef:
    name: selfsigned-issuer
    kind: Issuer
    group: cert-manager.io
```

my-postgres-client-cert という名前のシークレットが cert-manager によって作成され、必要なすべてのファイルが含まれており、次のようにクラスターから参照できます。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  certificates:
    clientCASecret: my-postgres-client-cert
    replicationTLSSecret: my-postgres-client-cert
  storage:
    size: 1Gi
```

cert-manager を使用してサーバーとクライアントの両方の CA と証明書を管理する完全な例は、 cluster-example-cert-manager.yaml 配置マニフェストにあります。

4.24 クライアント TLS/SSL 接続

CloudNativePG が TLS 証明書をサポートする方法の詳細については、ページを参照してください。

CloudNativePG オペレーターは、サーバー側とクライアント側で、転送中の暗号化と認証の両方で TLS/SSL と連携するように設計されています。 CNPG オペレーターを使用して作成されたクラスターには、TLS クライアント証明書を作成して署名するための認証局 (CA) が付属しています。kubectl の cnpg プラグインを介して、パスワードを使用する代わりにユーザーの認証に使用できる新しい TLS クライアント証明書を発行できます。

cluster-example-pg-hba.yaml .

マニフェスト。構成パラダイムの規則に従って、そのファイルは app というユーザーが所有する app データベースを自動的に作成します (この規則は、bootstrap セクションの initdb 構成で変更できます)。

4.24.1 新しい証明書の発行

kubectl のプラグインの使用方法の詳細については、ページを参照してください。

次のように、cluster-example PostgreSQL クラスターで app ユーザーの証明書を作成できます。

```
kubectl cnpg certificate cluster-app \
  --cnpg-cluster cluster-example \
  --cnpg-user app
```

次の方法で証明書を確認できるようになりました。

```
kubectl get secret cluster-app \
  -o jsonpath="{.data[tls\.crt]}" \
  | base64 -d | openssl x509 -text -noout \
  | head -n 11
```

出力：

```
Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number:
      5d:e1:72:8a:39:9f:ce:51:19:9d:21:ff:1e:4b:24:5d
    Signature Algorithm: ecdsa-with-SHA256
    Issuer: OU = default, CN = cluster-example
    Validity
      Not Before: Mar 22 10:22:14 2021 GMT
      Not After : Mar 22 10:22:14 2022 GMT
    Subject: CN = app
```

ご覧のとおり、TLS クライアント証明書はデフォルトで 90 日間有効で、PostgreSQL のユーザー名に対応する単純な CN で作成されます。これは、pg_hba.conf の hostssl エントリに cert 認証方法を利用するために必要です。

4.24.2 TLS 証明書による接続のテスト

次に、CloudNativePG クラスターに接続するデモ クライアントアプリケーションを構成して、このクライアント証明書をテストします。

次の cert-test.yaml というマニフェストは、データベースクラスターが実行されているのと同じ名前空間にテストアプリケーションを含むデモ Pod を作成します。

```
apiVersion: apps/v1
kind: Deployment
metadata:
```

(次のページに続く)

(前のページからの続き)

```
  name: cert-test
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webtest
  template:
    metadata:
      labels:
        app: webtest
    spec:
      containers:
        - image: ghcr.io/cloudnative-pg/webtest:1.6.0
          name: cert-test
          volumeMounts:
            - name: secret-volume-root-ca
              mountPath: /etc/secrets/ca
            - name: secret-volume-app
              mountPath: /etc/secrets/app
          ports:
            - containerPort: 8080
          env:
            - name: DATABASE_URL
              value: >
                sslkey=/etc/secrets/app/tls.key
                sslcert=/etc/secrets/app/tls.crt
                sslrootcert=/etc/secrets/ca/ca.crt
                host=cluster-example-rw.default.svc
                dbname=app
                user=app
                sslmode=verify-full
            - name: SQL_QUERY
              value: SELECT 1
          readinessProbe:
            httpGet:
              port: 8080
              path: /tx
      volumes:
        - name: secret-volume-root-ca
          secret:
            secretName: cluster-example-ca
            defaultMode: 0600
        - name: secret-volume-app
          secret:
            secretName: cluster-app
```

(次のページに続く)

(前のページからの続き)

```
            defaultMode: 0600
```

この Pod は、次のような CloudNativePG オペレーターによって管理されるシークレットをマウントします。

- sslcert : TLS クライアントの公開証明書
- sslkey : TLS クライアント証明書の秘密鍵
- sslrootcert : インスタンスの身元を確認するために使用されるサーバー上の証明書に署名した TLS 認証局証明書

それらは、psql (および pgbench のような他の libpq ベースのアプリケーション) が Postgres データベースへの TLS 暗号化接続を確立するために必要とするデフォルトリソースを作成するために使用されます。

デフォルトでは、psql は現在のユーザーの ~/.postgresql ディレクトリ内の証明書を検索しますが、sslkey、sslcert、sslrootcert オプションを使用して、libpq が暗号化マテリアルの実際の場所を指すようにすることができます。上記のファイルの内容は、kubectl の cnpg プラグインを使用して以前に作成されたシークレットから収集されます。

アプリケーションをデプロイします。

```
kubectl create -f cert-test.yaml
```

次に、作成した Pod を PostgreSQL クライアントとして使用して、作成したばかりの TLS 証明書を使用して SSL 接続と認証を検証します。

データベースサーバーにアクセスできるときにアプリケーションの準備が整っていることを確認する準備プローブが構成されています。

Pod のコンテナ内でインタラクティブな bash を実行し、必要なオプションを使用して psql を実行することで、接続が機能していることを確認できます。 PostgreSQL サーバーは、読み取り/書き込み Kubernetes サービスを通じて公開されます。このサービスに接続するように psql コマンドを指定します。

```
kubectl exec -it cert-test -- bash -c "psql
sslkey=/etc/secrets/app/tls.key sslcert=/etc/secrets/app/tls.crt
sslrootcert=/etc/secrets/ca/ca.crt host=cluster-example-rw.default.svc dbname=app
user=app sslmode=verify-full -c select version();"
```

出力:

```
                                        version
- ------------------------------------------------------------------------------
↪-
- -----------------
PostgreSQL 15.2 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.3.1 20191121 (Red Hat
8.3.1-5), 64-bit
(1 row)
```

4.25 アプリケーションからの接続

アプリケーションは、同じ Kubernetes クラスター内の CloudNativePG によって作成されたサービスと連携することになっています。

*[cluster name]-rw

*[cluster name]-ro

*[cluster name]-r

これらのサービスは Kubernetes クラスターによって完全に管理され、:ref:``microservice` タイプ<*microservice* タイプ>` で説明されているように仮想 IP の形式を実装します。

ヒント: アプリケーションでこれらのサービスを使用し、特定の PostgreSQL インスタンスに直接接続しないことを強くお勧めします。後者はクラスターの有効期間中に変更される可能性があるためです。

これらのサービスは、次の方法でアプリケーションで使用できます。

- DNS 解決
- 環境変数

PostgreSQL に接続するための資格情報については、オペレーターによって生成されたシークレットを使用できます。

警告: オペレーターは、*[cluster name]-any* という名前の別のサービスを作成します。このサービスは、PostgreSQL インスタンスの検出を管理するために内部で使用されます。アプリケーションによって直接使用されることは想定されていません。

4.25.1 DNS 解決

Kubernetes DNS サービスを使用して、特定のサーバーを指すことができます。オペレーターは Kubernetes DNS サービスを必要とします。アプリケーションが PostgreSQL クラスターと同じ名前空間にデプロイされている場合は、サービスの名前を使用してこれを行うことができます。PostgreSQL クラスターが別の名前空間にある場合は、完全修飾子 `service-name.namespace-name` を使用できます。

DNS は、優先および推奨される検出方法です。

4.25.2 環境変数

アプリケーションを PostgreSQL クラスターを含む同じ名前空間にデプロイする場合、環境変数を使用してデータベースに接続することもできます。

たとえば、PostgreSQL クラスターの名前が `pg-database` であるとすると、アプリケーションで次の環境変数を使用できます。

- `PG_DATABASE_R_SERVICE_HOST` : 読み取り専用ワークロードのすべての PostgreSQL インスタンスを指すサービスの IP アドレス
- `PG_DATABASE_RO_SERVICE_HOST` : クラスターのすべてのホットスタンバイ レプリカを指すサービスの IP アドレス
- `PG_DATABASE_RW_SERVICE_HOST` : クラスターの *primary* インスタンスを指すサービスの IP アドレス

4.25.3 秘密

PostgreSQL オペレーターは、デプロイするすべての PostgreSQL クラスターに対して 2 つの `basic-auth` タイプのシークレットを生成します。

*`[cluster name]-superuser`

*`[cluster name]-app`

シークレットには、ユーザー名、パスワード、および作業中の .pgpass file が含まれています

それぞれ `postgres` ユーザーとデータベースの **所有者** です。

`-app` 資格情報は、PostgreSQL クラスターに接続するアプリケーションが使用する資格情報です。

`-superuser` のものは、管理目的でのみ使用されることになっています。

4.26 接続プーリング

CloudNativePG は、`Pooler` CRD を通じて、PostgreSQL の最も一般的なオープン ソース接続プーラーの 1 つである *PgBouncer* **設定オプション**による接続プーリングのネイティブ サポートを提供します。

簡単に言うと、CloudNativePG の `Pooler` は、アプリケーションと PostgreSQL サービス (`rw` サービスなど) の間にある PgBouncer ポッドのデプロイであり、スケーラブルで構成可能で可用性の高い個別の**データベースアクセスレイヤー** を作成します。

4.26.1 建築

次の図は、PgBouncer に基づくデータベース アクセス レイヤーの導入によって CloudNativePG のアーキテクチャがどのように変化するかを示しています。アプリケーションは、PostgreSQL プライマリ サービスに直接接続する代わりに、PgBouncer の同等のサービスに接続できるようになりました。これにより、既存の接続を再利用して、PostgreSQL 側のパフォーマンスを高速化し、リソース管理を改善できます。

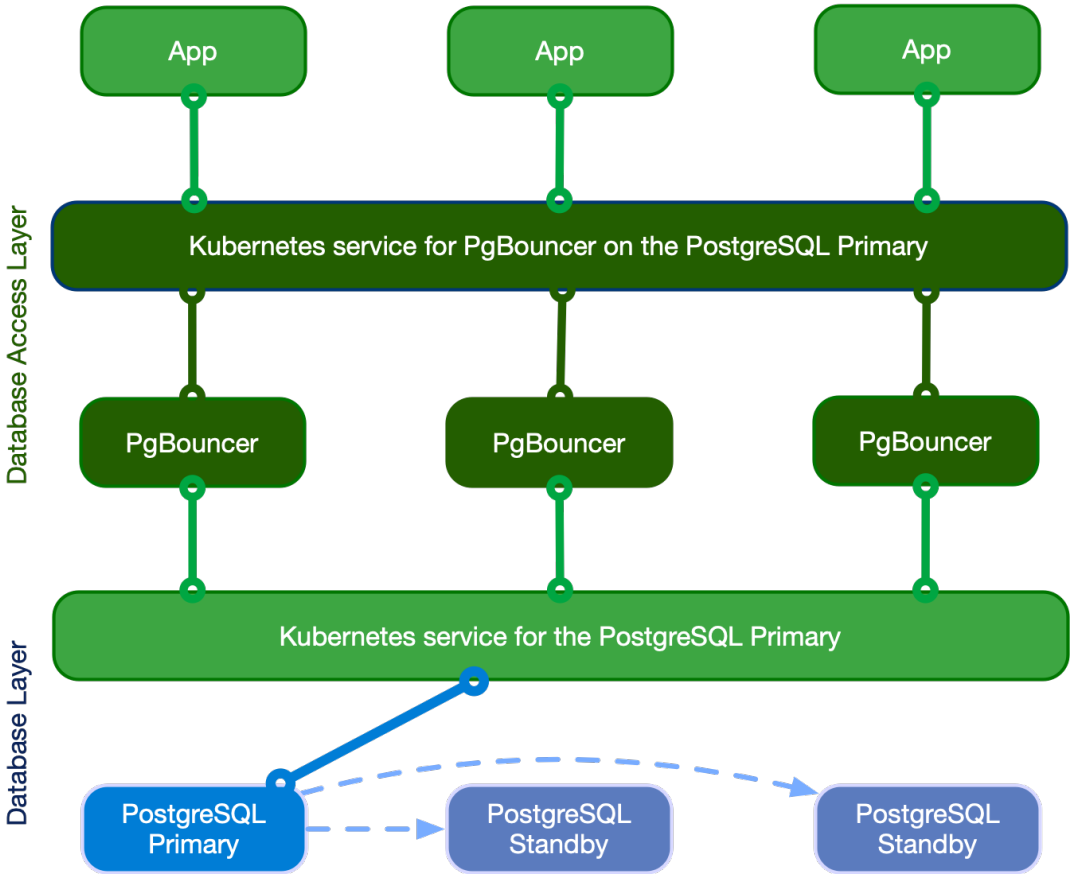


図 15: Applications writing to the single primary via PgBouncer

4.26.2 クイックスタート

CloudNativePG が PgBouncer プーラーを実装する方法を説明する最も簡単な方法は、例を使用することです。

```
apiVersion: postgresql.cnpg.io/v1
kind: Pooler
metadata:
  name: pooler-example-rw
spec:
  cluster:
    name: cluster-example
```

(次のページに続く)

(前のページからの続き)

```
  instances: 3
  type: rw
  pgbouncer:
    poolMode: session
    parameters:
      max_client_conn: "1000"
      default_pool_size: "10"
```

重要: プーラー名は、同じ名前空間内のクラスター名と決して一致してはなりません。

これにより、`pooler-example-rw` (名前は任意) と呼ばれる新しい `Pooler` リソースが作成されます。これは、`cluster-example` と呼ばれる Postgres `Cluster` リソースに厳密に関連付けられ、読み取り/書き込みサービス (`rw` 、したがって `cluster-example-rw`) によって識別されるプライマリを指します。

`Pooler` は、Postgres クラスターの同じ名前空間に存在する必要があります。これは、 session で構成された latest stable image of PgBouncer を実行する 3 つのポッドの Kubernetes デプロイメントで構成されています

それぞれ最大 1000 の接続を受け入れます。デフォルトのプールサイズは、PostgreSQL に対して 10 のユーザー/データベース ペアです。

重要: *Pooler* は、PgBouncer で * フォールバック データベースのみを設定します。つまり、クライアントから渡された接続文字列のすべてのパラメーターが PostgreSQL サーバーに中継されます (["Section [databases]" in PgBouncer's documentation](https://www.pgbouncer.org/config.html#section-databases) を参照してください)。

さらに、CloudNativePG は、PgBouncer で使用される構成ファイルを含むプーラーと同じ名前のシークレットを自動的に作成します。

API リファレンスで。

4.26.3 Pooler リソースのライフサイクル

`Pooler` リソースは、`Cluster` が管理するリソースではありません。必要に応じて、プーラーを手動で作成することになっています。さらに、PostgreSQL クラスターごとに複数のプーラーをデプロイできます。

注意すべき重要なことは、`Cluster` リソースと `Pooler` リソースのライフサイクルが現在独立していることです。

重要: *Pooler* がどのように機能するかがわかったので、考えられるアーキテクチャに関して完全な自由が得られます。プーラーのないクラスター、単一のプーラーを持つクラスター、または複数のプーラー (つまり、アプリケーションごとに 1 つ) を持つクラスターを持つことができます。

4.26.4 安全

すべての PgBouncer プーラーは、プールのクライアント (アプリケーション) 側とサーバー (PostgreSQL) 側の両方で、 **TLS 接続**を介して転送中の暗号化をサポートする CloudNativePG と透過的に統合されます。

具体的には、PgBouncer は PostgreSQL サーバーの証明書を自動的に再利用します。さらに、TLS クライアント証明書認証を使用して PostgreSQL サーバーに接続し、クライアントのパスワード認証のために `auth_query` を実行します (以下の **ユーザー名/パスワード認証** を参照)。

コンテナーは `pgbouncer` システムユーザーとして実行され、`pgbouncer` データベースへのアクセスは、`peer` 認証によるローカル接続経由でのみ許可されます。

証明書

デフォルトでは、PgBouncer プーラーはクラスター自体が使用するものと同じ証明書を使用しますが、ユーザーがこれらの証明書を提供すると、プーラーは次の形式のシークレットを受け入れます。

1. 基本認証

2.TLS

3. 不透明

Opaque の場合、使用する必要がある特定のキーを探します。これらのキーは次のとおりです。

- tls.crt
- tls.key

したがって、このシークレットを TLS シークレットとして扱い、そこから開始できます。

4.26.5 認証

パスワードベースの認証 は、CloudNativePG の PgBouncer のクライアントでサポートされている唯一の方法です。

内部的には、実装は PgBouncer の `auth_user` および `auth_query` オプションに依存しています。具体的には、オペレーター:

- PostgreSQL サーバーに `cnpg_pooler_pgbouncer` という標準ユーザーを作成します。
- `postgres` データベースにルックアップ関数を作成し、`cnpg_pooler_pgbouncer` ユーザーに実行権限を付与します (PoLA)
- このユーザーの TLS 証明書を発行します
- `cnpg_pooler_pgbouncer` を `auth_user` として設定します
- TLS 証明書を使用して `cnpg_pooler_pgbouncer` を PostgreSQL サーバーに対して認証するように PgBouncer を構成します。
- クラスターにプーラーが関連付けられていないことが検出されると、上記のすべてが削除されます

重要: 独自のシークレットを指定すると、オペレーターはプーラーを自動的に統合しません。

プーラーを手動で統合するには、独自のシークレットを指定した場合、クラスター内から次のクエリを実行する必要があります。

まず、ロールを作成する必要があります。

```
CREATE ROLE cnpg_pooler_pgbouncer WITH LOGIN;
```

次に、アプリケーション データベースごとに、cnpg_pooler_pgbouncer に接続するためのアクセス許可を付与します。

```
GRANT CONNECT ON DATABASE { database name here } TO cnpg_pooler_pgbouncer;
```

最後に、各アプリケーション データベースに接続し、各アプリケーションデータベース内に認証関数を作成します。

```
CREATE OR REPLACE FUNCTION user_search(uname TEXT)
  RETURNS TABLE (usename name, passwd text)
  LANGUAGE sql SECURITY DEFINER AS
  SELECT usename, passwd FROM pg_shadow WHERE usename=$1;;

REVOKE ALL ON FUNCTION user_search(text)
  FROM public;

GRANT EXECUTE ON FUNCTION user_search(text)
  TO cnpg_pooler_pgbouncer;
```

4.26.6 ポッドテンプレート

Pooler リソースの template セクションでポッドテンプレートの仕様を利用できます。詳細については、API リファレンスの `PoolerSpec section <api_reference.md#PoolerSpec>`__ を参照してください。

テンプレートを使用して、ポッドとノードのアフィニティおよび非アフィニティルールを細かく制御するなど、好きなようにポッドを構成できます。デフォルトでは、コンテナーは ghcr.io/cloudnative-pg/pgbouncer のイメージを使用します。

PodAntiAffinity を指定する Pooler の例を次に示します。

```
apiVersion: postgresql.cnpg.io/v1
kind: Pooler
metadata:
  name: pooler-example-rw
spec:
  cluster:
```

(次のページに続く)

(前のページからの続き)

```
    name: cluster-example
  instances: 3
  type: rw

  template:
    metadata:
      labels:
        app: pooler
    spec:
      containers: []
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - pooler
            topologyKey: "kubernetes.io/hostname"
```

注釈: *.spec.template.spec.containers* は、*PodSpec* の必須フィールドであるため、変更されていない場合は明示的に *[]* に設定する必要があります。 *.spec.template.spec.containers* が設定されていない場合、マニフェストを適用しようとすると、kubernetes api-server は次のエラーを返します: *error validating "pooler.yaml": error validating data: ValidationError(Pooler.spec.template.spec): missing required field "containers"*

リソースを設定し、使用する画像を変更する例を次に示します。

```
apiVersion: postgresql.cnpg.io/v1
kind: Pooler
metadata:
  name: pooler-example-rw
spec:
  cluster:
    name: cluster-example
  instances: 3
  type: rw

  template:
    metadata:
      labels:
        app: pooler
    spec:
```

(次のページに続く)

(前のページからの続き)

```
    containers:
      - name: pgbouncer
        image: my-pgbouncer:latest
        resources:
          requests:
            cpu: "0.1"
            memory: 100Mi
          limits:
            cpu: "0.5"
            memory: 500Mi
```

4.26.7 高可用性 (HA)

Kubernetes のデプロイメントのおかげで、単一のインスタンスまたは複数のポッドで実行するようにプーラーを構成できます。公開されたサービスは、PgBouncer を実行している利用可能なポッドにクライアントがランダムに分散されるようにします。これにより、基になるサーバー (`rw` サービスを使用している場合) またはサーバー (複数のレプリカで `ro` サービスを使用している場合) への接続が自動的に管理および再利用されます。 .

> **警告:** インフラストラクチャが複数のアベイラビリティ ゾーンにまたがり、それらの間でレイテンシが高い場合は、ネットワーク ホップに注意してください。たとえば、ゾーン 2 で実行されているアプリケーションがゾーン 3 で実行されている PgBouncer に接続し、ゾーン 1 の PostgreSQL プライマリを指している場合を考えてみましょう。

4.26.8 PgBouncer 設定オプション

オペレーターは configuration options for PgBouncer のほとんどを管理し、それらのサブセットのみを変更できるようにします。

> **警告:** オペレーターはそれらを検証しないため、各オプションの値を正しく設定する責任があります。

以下に、カスタマイズできる PgBouncer オプションのリストを示します。それぞれに、その特定のパラメーターに関する PgBouncer ドキュメントへのリンクが含まれています。ここで別段の記載がない限り、デフォルト値は PgBouncer によって直接設定されたものです。

- application_name_add_host
- autodb_idle_timeout
- client_idle_timeout
- client_login_timeout

- default_pool_size
- disable_pqexec
- idle_transaction_timeout
- ignore_startup_parameters : `extra_float_digits,options` に追加する - CNP

  で必要
- log_connections
- log_disconnections
- log_pooler_errors
- log_stats : 以下の **モニタリング**

  セクションで説明されているように、Prometheus エクスポートによって統計が既に収集されている場合、デフォルトで無効 (`0`) です。
- max_client_conn
- max_db_connections
- max_user_connections
- min_pool_size
- query_timeout
- query_wait_timeout
- reserve_pool_size
- reserve_pool_timeout
- server_check_delay
- server_check_query
- server_connect_timeout
- server_fast_close
- server_idle_timeout
- server_lifetime
- server_login_retry
- server_reset_query
- server_reset_query_always
- server_round_robin
- stats_period
- tcp_keepalive
- tcp_keepcnt

- tcp_keepidle
- tcp_keepintvl
- tcp_user_timeout
- verbose

PgBouncer 構成のカスタマイズは、`.spec.pgbouncer.parameters` マップで宣言的に記述されます。

オペレーターは Pooler 仕様の変更に対応し、サービスを中断することなく、すべての PgBouncer インスタンスが更新された構成をリロードします。

警告: すべての PgBouncer ポッドは、仕様のパラメーターに合わせて同じ構成になります。これらのパラメーターに誤りがあると、**プーラー全体**の操作性が損なわれる可能性があります。演算子は、オプションの値を検証しません。**

4.26.9 モニタリング

Pooler の PgBouncer 実装には、次を実行することにより、cnpg_pgbouncer_ プレフィックスを持ついくつかのメトリックを自動的に利用可能にするデフォルトの Prometheus エクスポーターが付属しています。

- SHOW LISTS (プレフィックス: cnpg_pgbouncer_lists)
- SHOW POOLS (プレフィックス: cnpg_pgbouncer_pools)
- SHOW STATS (プレフィックス: cnpg_pgbouncer_stats)

CloudNativePG インスタンスと同様に、エクスポーターは PgBouncer を実行している各ポッドのポート 9127 で実行され、Go ランタイムに関連するメトリックも提供します (接頭辞 go_* 付き)。次のコマンドを使用して、PgBouncer を実行しているポッドでエクスポーターをデバッグできます。

```
kubectl exec -ti <PGBOUNCER_POD> -- curl 127.0.0.1:9127/metrics
```

cnpg_pgbouncer メトリクスの出力例:

```
#  HELP cnpg_pgbouncer_collection_duration_seconds Collection time duration in seconds

#  TYPE cnpg_pgbouncer_collection_duration_seconds gauge

cnpg_pgbouncer_collection_duration_seconds{collector="Collect.up"} 0.002443168

#  HELP cnpg_pgbouncer_collections_total Total number of times PostgreSQL was␣
↪accessed for metrics.

#  TYPE cnpg_pgbouncer_collections_total counter

cnpg_pgbouncer_collections_total 1
```

(次のページに続く)

(前のページからの続き)

```
# HELP cnpg_pgbouncer_last_collection_error 1 if the last collection ended with␣
↪error, 0 otherwise.

# TYPE cnpg_pgbouncer_last_collection_error gauge

cnpg_pgbouncer_last_collection_error 0

# HELP cnpg_pgbouncer_lists_databases Count of databases.

# TYPE cnpg_pgbouncer_lists_databases gauge

cnpg_pgbouncer_lists_databases 1

# HELP cnpg_pgbouncer_lists_dns_names Count of DNS names in the cache.

# TYPE cnpg_pgbouncer_lists_dns_names gauge

cnpg_pgbouncer_lists_dns_names 0

# HELP cnpg_pgbouncer_lists_dns_pending Not used.

# TYPE cnpg_pgbouncer_lists_dns_pending gauge

cnpg_pgbouncer_lists_dns_pending 0

# HELP cnpg_pgbouncer_lists_dns_queries Count of in-flight DNS queries.

# TYPE cnpg_pgbouncer_lists_dns_queries gauge

cnpg_pgbouncer_lists_dns_queries 0

# HELP cnpg_pgbouncer_lists_dns_zones Count of DNS zones in the cache.

# TYPE cnpg_pgbouncer_lists_dns_zones gauge

cnpg_pgbouncer_lists_dns_zones 0

# HELP cnpg_pgbouncer_lists_free_clients Count of free clients.

# TYPE cnpg_pgbouncer_lists_free_clients gauge

cnpg_pgbouncer_lists_free_clients 49

# HELP cnpg_pgbouncer_lists_free_servers Count of free servers.
```

(次のページに続く)

(前のページからの続き)

```
#  TYPE cnpg_pgbouncer_lists_free_servers gauge

cnpg_pgbouncer_lists_free_servers 0

#  HELP cnpg_pgbouncer_lists_login_clients Count of clients in login state.

#  TYPE cnpg_pgbouncer_lists_login_clients gauge

cnpg_pgbouncer_lists_login_clients 0

#  HELP cnpg_pgbouncer_lists_pools Count of pools.

#  TYPE cnpg_pgbouncer_lists_pools gauge

cnpg_pgbouncer_lists_pools 1

#  HELP cnpg_pgbouncer_lists_used_clients Count of used clients.

#  TYPE cnpg_pgbouncer_lists_used_clients gauge

cnpg_pgbouncer_lists_used_clients 1

#  HELP cnpg_pgbouncer_lists_used_servers Count of used servers.

#  TYPE cnpg_pgbouncer_lists_used_servers gauge

cnpg_pgbouncer_lists_used_servers 0

#  HELP cnpg_pgbouncer_lists_users Count of users.

#  TYPE cnpg_pgbouncer_lists_users gauge

cnpg_pgbouncer_lists_users 2

#  HELP cnpg_pgbouncer_pools_cl_active Client connections that are linked to server␣
↪connection and can process queries.

#  TYPE cnpg_pgbouncer_pools_cl_active gauge

cnpg_pgbouncer_pools_cl_active{database="pgbouncer",user="pgbouncer"} 1

#  HELP cnpg_pgbouncer_pools_cl_cancel_req Client connections that have not forwarded␣
↪query cancellations to the server yet.
```

(次のページに続く)

(前のページからの続き)

```
#  TYPE cnpg_pgbouncer_pools_cl_cancel_req gauge

cnpg_pgbouncer_pools_cl_cancel_req{database="pgbouncer",user="pgbouncer"} 0

#  HELP cnpg_pgbouncer_pools_cl_waiting Client connections that have sent queries but␣
↪have not yet got a server connection.

#  TYPE cnpg_pgbouncer_pools_cl_waiting gauge

cnpg_pgbouncer_pools_cl_waiting{database="pgbouncer",user="pgbouncer"} 0

#  HELP cnpg_pgbouncer_pools_maxwait How long the first (oldest) client in the queue␣
↪has waited, in seconds. If this starts increasing, then the current pool of servers␣
↪does not handle requests quickly enough. The reason may be either an overloaded␣
↪server or just too small of a pool_size setting.

#  TYPE cnpg_pgbouncer_pools_maxwait gauge

cnpg_pgbouncer_pools_maxwait{database="pgbouncer",user="pgbouncer"} 0

#  HELP cnpg_pgbouncer_pools_maxwait_us Microsecond part of the maximum waiting time.

#  TYPE cnpg_pgbouncer_pools_maxwait_us gauge

cnpg_pgbouncer_pools_maxwait_us{database="pgbouncer",user="pgbouncer"} 0

#  HELP cnpg_pgbouncer_pools_pool_mode The pooling mode in use. 1 for session, 2 for␣
↪transaction, 3 for statement, -1 if unknown

#  TYPE cnpg_pgbouncer_pools_pool_mode gauge

cnpg_pgbouncer_pools_pool_mode{database="pgbouncer",user="pgbouncer"} 3

#  HELP cnpg_pgbouncer_pools_sv_active Server connections that are linked to a client.

#  TYPE cnpg_pgbouncer_pools_sv_active gauge

cnpg_pgbouncer_pools_sv_active{database="pgbouncer",user="pgbouncer"} 0

#  HELP cnpg_pgbouncer_pools_sv_idle Server connections that are unused and␣
↪immediately usable for client queries.

#  TYPE cnpg_pgbouncer_pools_sv_idle gauge

cnpg_pgbouncer_pools_sv_idle{database="pgbouncer",user="pgbouncer"} 0
```

(次のページに続く)

(前のページからの続き)

```
# HELP cnpg_pgbouncer_pools_sv_login Server connections currently in the process of␣
↪logging in.

# TYPE cnpg_pgbouncer_pools_sv_login gauge

cnpg_pgbouncer_pools_sv_login{database="pgbouncer",user="pgbouncer"} 0

# HELP cnpg_pgbouncer_pools_sv_tested Server connections that are currently running␣
↪either server_reset_query or server_check_query.

# TYPE cnpg_pgbouncer_pools_sv_tested gauge

cnpg_pgbouncer_pools_sv_tested{database="pgbouncer",user="pgbouncer"} 0

# HELP cnpg_pgbouncer_pools_sv_used Server connections that have been idle for more␣
↪than server_check_delay, so they need server_check_query to run on them before they␣
↪can be used again.

# TYPE cnpg_pgbouncer_pools_sv_used gauge

cnpg_pgbouncer_pools_sv_used{database="pgbouncer",user="pgbouncer"} 0

# HELP cnpg_pgbouncer_stats_avg_query_count Average queries per second in last stat␣
↪period.

# TYPE cnpg_pgbouncer_stats_avg_query_count gauge

cnpg_pgbouncer_stats_avg_query_count{database="pgbouncer"} 1

# HELP cnpg_pgbouncer_stats_avg_query_time Average query duration, in microseconds.

# TYPE cnpg_pgbouncer_stats_avg_query_time gauge

cnpg_pgbouncer_stats_avg_query_time{database="pgbouncer"} 0

# HELP cnpg_pgbouncer_stats_avg_recv Average received (from clients) bytes per␣
↪second.

# TYPE cnpg_pgbouncer_stats_avg_recv gauge

cnpg_pgbouncer_stats_avg_recv{database="pgbouncer"} 0

# HELP cnpg_pgbouncer_stats_avg_sent Average sent (to clients) bytes per second.
```

(次のページに続く)

(前のページからの続き)

```
#  TYPE cnpg_pgbouncer_stats_avg_sent gauge

cnpg_pgbouncer_stats_avg_sent{database="pgbouncer"} 0

#  HELP cnpg_pgbouncer_stats_avg_wait_time Time spent by clients waiting for a server,
 in microseconds (average per second).

#  TYPE cnpg_pgbouncer_stats_avg_wait_time gauge

cnpg_pgbouncer_stats_avg_wait_time{database="pgbouncer"} 0

#  HELP cnpg_pgbouncer_stats_avg_xact_count Average transactions per second in last␣
↪stat period.

#  TYPE cnpg_pgbouncer_stats_avg_xact_count gauge

cnpg_pgbouncer_stats_avg_xact_count{database="pgbouncer"} 1

#  HELP cnpg_pgbouncer_stats_avg_xact_time Average transaction duration, in␣
↪microseconds.

#  TYPE cnpg_pgbouncer_stats_avg_xact_time gauge

cnpg_pgbouncer_stats_avg_xact_time{database="pgbouncer"} 0

#  HELP cnpg_pgbouncer_stats_total_query_count Total number of SQL queries pooled by␣
↪pgbouncer.

#  TYPE cnpg_pgbouncer_stats_total_query_count gauge

cnpg_pgbouncer_stats_total_query_count{database="pgbouncer"} 3

#  HELP cnpg_pgbouncer_stats_total_query_time Total number of microseconds spent by␣
↪pgbouncer when actively connected to PostgreSQL, executing queries.

#  TYPE cnpg_pgbouncer_stats_total_query_time gauge

cnpg_pgbouncer_stats_total_query_time{database="pgbouncer"} 0

#  HELP cnpg_pgbouncer_stats_total_received Total volume in bytes of network traffic␣
↪received by pgbouncer.

#  TYPE cnpg_pgbouncer_stats_total_received gauge

cnpg_pgbouncer_stats_total_received{database="pgbouncer"} 0
```

(次のページに続く)

(前のページからの続き)

```
# HELP cnpg_pgbouncer_stats_total_sent Total volume in bytes of network traffic sent␣
↪by pgbouncer.

# TYPE cnpg_pgbouncer_stats_total_sent gauge

cnpg_pgbouncer_stats_total_sent{database="pgbouncer"} 0

# HELP cnpg_pgbouncer_stats_total_wait_time Time spent by clients waiting for a␣
↪server, in microseconds.

# TYPE cnpg_pgbouncer_stats_total_wait_time gauge

cnpg_pgbouncer_stats_total_wait_time{database="pgbouncer"} 0

# HELP cnpg_pgbouncer_stats_total_xact_count Total number of SQL transactions pooled␣
↪by pgbouncer.

# TYPE cnpg_pgbouncer_stats_total_xact_count gauge

cnpg_pgbouncer_stats_total_xact_count{database="pgbouncer"} 3

# HELP cnpg_pgbouncer_stats_total_xact_time Total number of microseconds spent by␣
↪pgbouncer when connected to PostgreSQL in a transaction, either idle in transaction␣
↪or executing queries.

# TYPE cnpg_pgbouncer_stats_total_xact_time gauge

cnpg_pgbouncer_stats_total_xact_time{database="pgbouncer"} 0
```

Clusters と同様に、 プロメテウス オペレーターの例 を使用している場合

次の PodMonitor を定義することで、特定のプーラーをスクレイピングするように構成できます。

```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: <POOLER_NAME>
spec:
  selector:
    matchLabels:
      cnpg.io/poolerName: <POOLER_NAME>
  podMetricsEndpoints:
  - port: metrics
```

4.26.10 ロギング

ログは、次の例のように、JSON 形式で標準出力に直接送信されます。

```
{
  "level": "info",
  "ts": SECONDS.MICROSECONDS,
  "msg": "record",
  "pipe": "stderr",
  "record": {
    "timestamp": "YYYY-MM-DD HH:MM:SS.MS UTC",
    "pid": "<PID>",
    "level": "LOG",
    "msg": "kernel file descriptor limit: 1048576 (hard: 1048576); max_client_conn:␣
↪100, max expected fd use: 112"
  }
}
```

4.26.11 接続の一時停止

Pooler 仕様では、デフォルトで false に設定されている paused オプションを介して、宣言的な構成のみを使用して、PgBouncer の PAUSE および RESUME コマンドを利用できます。 true に設定すると、オペレーターは PgBouncer で PAUSE コマンドを内部的に呼び出します。

1. クエリが完了するのを待った後、PostgreSQL サーバーへのすべてのアクティブな接続を閉じます。
2. クライアントからの新しい接続を一時停止します

paused オプションが false に戻されると、オペレーターは PgBouncer で RESUME コマンドを呼び出し、Pooler で定義された PostgreSQL サービスへのタップを再度開きます。

重要: 将来のバージョンでは、切り替え操作は PgBouncer プーラーと完全に統合され、*PAUSE /RESUME* 機能を利用して、クライアント アプリケーションによって認識されるダウンタイムを削減します。現時点では、*paused* 属性を *true* に設定し、 :ref:``kubectl` に *cnpg* プラグインを使用する<*kubectl* に *cnpg* プラグインを使用する>` を介して切り替えコマンドを発行し、最後に *paused* 属性を *false* に復元することで、同じ結果を得ることができます。

4.26.12 制限事項

単一の PostgreSQL クラスター

プーラーの現在の実装は、特定の CloudNativePG クラスター (正確にはサービス) の一部として機能するように設計されています。現時点では、複数のクラスターにまたがるプーラーを作成することはできません。

制御された構成可能性

CloudNativePG は、PgBouncer レイヤーが PostgreSQL と通信するために使用されるいくつかの構成オプションを透過的に管理します。このようなオプションは外部から設定できず、TLS 証明書、認証設定、`databases` セクション、および `users` セクションが含まれます。また、単一の PostgreSQL クラスターの特定の使用例を考慮して、採用された基準は、ユーザーが構成できるオプションを明示的にリストすることです。

注釈: より高度でカスタマイズされたシナリオでガンマを完了するために、PgBouncer の別のオペレーターの将来の実装の余地を残しながら、採用されたソリューションがほとんどのユース ケースに対処すると信じる理由があります。

4.27 レプリカ クラスタ

レプリカ クラスターは独立した CloudNativePG `Cluster` リソースであり、理想的には CloudNativePG によっても管理され、別の Postgres インスタンスからレプリカになるという主な特徴があります。通常、レプリカクラスターは別のリージョンの別の Kubernetes クラスターにあります。レプリカクラスターもカスケードすることができ、さらに下で説明するように、ソースからのデータのレプリケーションをオブジェクトストアのみに依存することができます。

以下の図 (この機能に関する詳細情報を含む “Architecture” section から取得) は、レプリカクラスターで実装できるアーキテクチャの例を示しています。

4.27.1 基本概念

CloudNativePG は、PostgreSQL クラスターが既存のクラスター (ソース) から作成され、 **レプリカ クラスタ**機能によって同期が維持されている場合でも、PostgreSQL レプリケーションフレームワークの基盤に依存しています。ソースは、プライマリクラスターまたは別のレプリカ クラスター (カスケード レプリカ クラスター) にすることができます。

ブートストラップ レベルと継続的リカバリレベルの両方で、レプリケーションに関して利用可能なオプションは次のとおりです。

- レプリカクラスタとソース間でストリーミングレプリケーションを使用します (これには、2 つのクラスタ間のネットワーク接続が正しく設定されていることを確認するために、管理およびセキュリティ関連の作業を行う必要があります)。

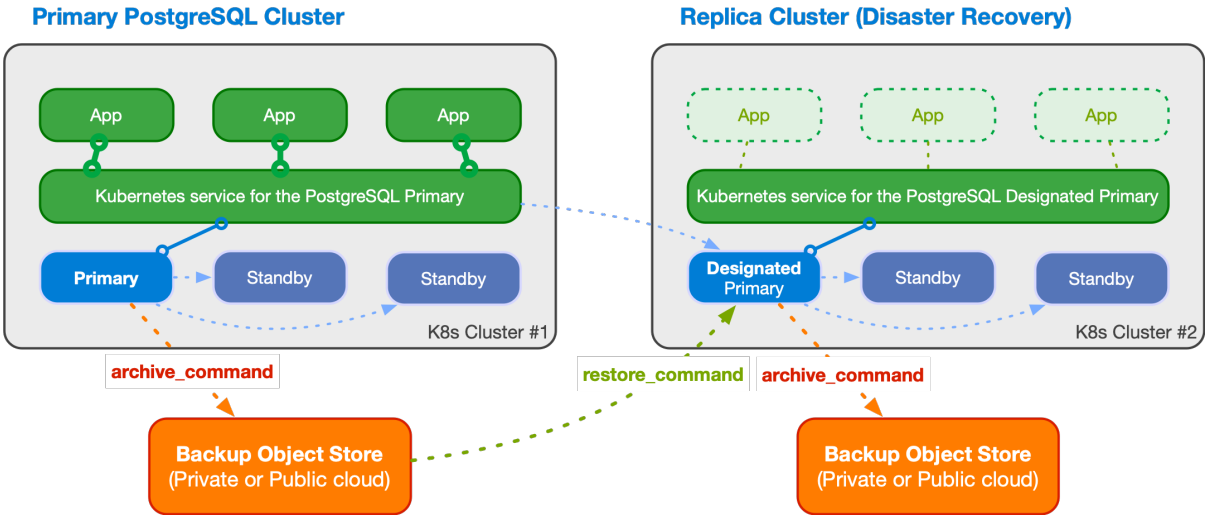


図 16: An example of multi-cluster deployment with a primary and a replica cluster

- ソースからオブジェクト ストアに定期的に送信され、レプリカ クラスタの `barman-cloud-wal-restore` によってプルされるベース バックアップと WAL ファイルのリカバリに Barman Cloud オブジェクト ストアを使用する
- 2 つのいずれか

必要なのは、実際に外部クラスターを定義することだけです。 ブートストラップを参照してください。

`pg_basebackup` (ストリーミング) または `recovery` (オブジェクトストア) を使用して PostgreSQL サーバーのクローンを作成する方法については、 を参照してください。

外部クラスターに `barmanObjectStore` セクションが含まれている場合:

- `recovery` セクションを使用して、オブジェクト ストアからレプリカクラスターをブートストラップできます。
- CloudNativePG は、指定されたプライマリ インスタンスに `restore_command` を自動的に設定します。

外部クラスターに `connectionParameters` セクションが含まれている場合:

- `pg_basebackup` セクションを使用して、ストリーミングレプリケーションを介してレプリカクラスターをブートストラップできます。
- CloudNativePG は、指定されたプライマリ インスタンスに `primary_conninfo` オプションを自動的に設定し、ソースクラスタに接続してデータを受信する WAL レシーバプロセスが開始されるようにします。

作成されたレプリカクラスタは、指定されたプライマリから予約済みオブジェクトストアでバックアップを実行でき、分散方式で対称アーキテクチャを有効にします。

以下を選択することで、PostgreSQL データベースの好みの分散アーキテクチャを決定する完全な柔軟性と自由が得られます。

- 異なるデータ センター内の複数の Kubernetes クラスターにまたがるプライベート クラウド
- 異なる地域の複数の Kubernetes クラスターにまたがるパブリック クラウド

- 前の 2 つの混合 (ハイブリッド)
- 異なる地域と異なるクラウド サービス プロバイダーの複数の Kubernetes クラスターにまたがるパブリック クラウド

4.27.2 レプリカ クラスタのセットアップ

ソース クラスターからレプリカクラスターをセットアップするには、クラスター YAML ファイルを作成し、それに応じて次の部分を定義する必要があります。

- レプリカ クラスターで externalClusters セクションを定義します。
- レプリカ クラスタのブートストラップ部分を定義します。pg_basebackup セクションを使用してストリーミング経由でブートストラップするか、recovery セクションを使用してオブジェクトストアからブートストラップすることができます。
- レプリカ クラスターで継続的な復旧部分 (spec.replica) を定義します。オプション spec.replica.enabled でレプリカモードを有効にし、オプション spec.replica.source で externalClusters 名を設定するだけです。

この **最初の例**は、ブートストラップと継続的な復旧の両方でストリーミングレプリケーションを使用するレプリカクラスターを定義します。レプリカ クラスターは、TLS 認証を使用してソースクラスターに接続します。

> sample YAML を確認できます

samples/ サブディレクトリにあります。

ソース クラスターを指している bootstrap および replica セクションに注意してください。

```
bootstrap:
  pg_basebackup:
    source: cluster-example

replica:
  enabled: true
  source: cluster-example
```

externalClusters セクションでは、connectionParameters サブセクションのホストに正しい名前空間を使用することを忘れないでください。-replication および -ca シークレットは、レプリカクラスターが別の名前空間にある場合に備えて、必要に応じてコピーされている必要があります。

```
externalClusters:
- name: <MAIN-CLUSTER>
  connectionParameters:
    host: <MAIN-CLUSTER>-rw.<NAMESPACE>.svc
    user: streaming_replica
    sslmode: verify-full
    dbname: postgres
```

(次のページに続く)

(前のページからの続き)

```
  sslKey:
    name: <MAIN-CLUSTER>-replication
    key: tls.key
  sslCert:
    name: <MAIN-CLUSTER>-replication
    key: tls.crt
  sslRootCert:
    name: <MAIN-CLUSTER>-ca
    key: ca.crt
```

2 番目の例 では、recovery セクションを使用してオブジェクトストアからブートストラップし、ストリーミングレプリケーションと指定されたオブジェクトストアの両方を使用して継続的な回復を行うレプリカクラスターを定義します。ストリーミングレプリケーションの場合、レプリカクラスタは基本認証を使用してソース クラスタに接続します。

sample YAML を確認できます

samples/ サブディレクトリにあります。

ソース クラスターを指している bootstrap および replica セクションに注意してください。

```
bootstrap:
  recovery:
    source: cluster-example

replica:
  enabled: true
  source: cluster-example
```

externalClusters セクションでは、endpointURL と connectionParameters.host で正しい名前空間を使用するように注意してください。また、必要なシークレットが必要に応じてコピーされていること、およびソースクラスターのバックアップが既に作成されていることを確認してください。

```
externalClusters:
- name: <MAIN-CLUSTER>
  barmanObjectStore:
    destinationPath: s3://backups/
    endpointURL: http://minio:9000
    s3Credentials:
      ...
  connectionParameters:
    host: <MAIN-CLUSTER>-rw.default.svc
    user: postgres
    dbname: postgres
  password:
    name: <MAIN-CLUSTER>-superuser
    key: password
```

注釈: ソース クラスターとレプリカ クラスター間でストリーミングレプリケーションを使用するには、2つのクラスター間にネットワーク接続があること、およびパスワードまたは証明書を保持する必要なすべてのシークレットが事前に適切に作成されていることを確認する必要があります。

4.27.3 レプリカ クラスタ内の指定されたプライマリの昇格

指定されたプライマリ を **プライマリ** に昇格させるには、オプション `spec.replica.enabled` を使用してレプリカ クラスターのレプリカモードを無効にするだけです。

```
replica:
  enabled: false
  source: cluster-example
```

レプリカ モードが無効になると、レプリカ クラスターとソース クラスターは 2 つの別個のクラスターになり、レプリカ クラスター内の**指定されたプライマリ** は、そのクラスターの **プライマリ**に昇格されます。cnpg プラグインを使用してロールの変更を確認し、以前はレプリカであったクラスターのステータスを確認できます。

```
kubectl cnpg -n <cluster-name-space> status cluster-replica-example
```

注釈: レプリケーションの無効化は **不可逆** 操作です: レプリケーションが無効になり、**指定されたプライマリ** が **プライマリ** に昇格すると、レプリカ クラスターとソース クラスターは確実に 2 つの独立したクラスターになります。

4.28 Kubernetes のアップグレード

Kubernetes クラスターは常に最新の状態にしておく必要があります。特に**ベアメタル** で Kubernetes クラスターを自己管理している場合、これはさらに重要になります。

定期的な更新を計画して実行することは、組織が技術的負債をクリーンアップし、ビジネスリスクを軽減する方法です。Kubernetes インフラストラクチャに、メンテナンスの理由でクラスターからノードを一時的に取り出す制御されたダウンタイムが導入されているにもかかわらず (推奨される読書: Embracing Risk

Site Reliability Engineering ブックから)。

たとえば、Kubernetes がインストールされている Linux サーバーにセキュリティ更新プログラムを適用したり、RAM、CPU、RAID コントローラーなどの誤動作しているハードウェアコンポーネントを交換したり、クラスターを最新バージョンの Kubernetes にアップグレードしたりする必要がある場合があります。

通常、クラスター内のメンテナンス操作は、次の方法で一度に 1 つのノードで実行されます。

1. 更新するノードからワークロードを削除する (`drain`)
2. 実際の操作の実行 (システムの更新など)

3. ノードをクラスターに再参加させる (uncordon)

上記のプロセスでは、アップグレードの全期間にわたってワークロードを停止するか、別のノードに移行する必要があります。

最新のケースは、サービスの信頼性と Kubernetes の自己修復機能の点で予想されるものですが、一時的に機能が低下したクラスターで運用し、アップグレードされたノードが再び稼働するまで待つことをお勧めする状況が発生する可能性があります。

特に、PostgreSQL クラスターが **ノードローカルストレージ** - に依存している場合、これは *PostgreSQL* **データベースが実行されている** *Kubernetes* **ワーカーノードに対してローカルなストレージです**。ノードローカル ストレージ (または単に **ローカル ストレージ**) を使用して、パフォーマンスを向上させます。

注釈: データベース ファイルがネットワーク上の共有ストレージにある場合、メンテナンス ウィンドウを定義する必要はないかもしれません。 Pod によって現在使用されているボリュームが、ドレイン後に別のノードで実行されている Pod によって再利用できる場合、オペレーターのデフォルトの自己修復動作は正常に機能します (その後、このセクションの残りをスキップできます)。

PostgreSQL にローカルストレージを使用する場合は、`nodeMaintenanceWindow` オプションを使用してクラスターを一時的に **メンテナンスモード**にして、標準の自己修復手順が開始されないようにすることをお勧めします。たとえば、物理ノードのパーティションを拡大したり、ノード自体を更新します。

> **警告:** メンテナンス ウィンドウの期間を可能な限り短い時間に制限します。このフェーズでは、Kubernetes の予期される動作の一部が無効になっているか、いくつかの制限付きで実行されています。これには、自己修復、ローリング アップデート、Pod 中断バジェットが含まれます。

クラスターの `nodeMaintenanceWindow` オプションには、さらに 2 つの設定があります。

`inProgress` : ノードのメンテナンスウィンドウが現在進行中かどうかを示すブール値。デフォルトでは、`off` に設定されています。メンテナンス期間中に、以下の `reusePVC` オプションがオペレーターによって評価されます。

`reusePVC` : メンテナンス操作中に既存の PVC を再利用するかどうかを定義するブール値。デフォルトでは、on に設定されています。 **有効** にすると、Kubernetes はノードが再起動するのを待ってから、既存の PVC を再利用します。`PodDisruptionBudget` ポリシーは一時的に削除されます。 **無効化**すると、Kubernetes は、PostgreSQL の物理ストリーミングレプリケーションに依存して、新しい PVC を使用して別のノードに Pod を強制的に再作成し、古い PVC を Pod と一緒に破棄します。このシナリオは、データベースのサイズが小さく、新しい PostgreSQL インスタンスの再クローンを作成する方が待機するよりも短時間で済む場合を除き、通常はお勧めできません。この動作は、インスタンスが 1 つしかなく、reusePVC が無効になっているクラスターには**適用されません** 。以下のセクションを参照してください。

注釈: *kubectl drain* コマンドを実行する場合、*--delete-local-data* オプションを追加する必要があります。恐れる必要はありません。これは、PostgreSQL データ ディレクトリではなく、オペレータが内部的に使用する別のボリュームを参照しています。

4.28.1 reusePVC が false に設定された単一インスタンス クラスタ

重要: 高可用性を保証するために、常に複数のインスタンスを持つクラスターを作成することをお勧めします。

reusePVC が false に設定されている単一インスタンスクラスター内の唯一の PostgreSQL インスタンスを削除すると、すべてのデータが失われることを意味するため、メンテナンスモードであっても、そのようなインスタンスが実行されている可能性のあるノードをユーザーがドレインできないようにします。

ただし、そのようなノードのメンテナンスが必要な場合は、次の 2 つのオプションがあります。

1. reusePVC を有効にして、ダウンタイムを受け入れます
2. インスタンスを別のノードにレプリケートし、プライマリを切り替えます

環境でデータベースサービスのダウンタイムが許容される限り、ノードのドレインは nodeMaintenanceWindow を inProgress: true および reusePVC: true に設定するのと同じくらい簡単です。これにより、元の PVC が利用可能になるとすぐにインスタンスを削除して再作成できます (ノードがバックアップされるとすぐに、ノード ローカルストレージを使用する場合など)。

それ以外の場合は、クラスタをスケールアップして、別のノードに新しいインスタンスを作成し、新しいインスタンスをプライマリに昇格させて、メンテナンス中のノードで元のインスタンスをシャットダウンする必要があります。この場合の唯一のダウンタイムは、スイッチオーバーの期間です。

考えられるアプローチは次のとおりです。

1. 現在のインスタンスが実行されているノードを閉鎖します。
2. クラスターを 2 つのインスタンスにスケールアップします。データベースのサイズによっては、時間がかかる場合があります。
3. 新しいインスタンスが実行されるとすぐに、現在のプライマリが閉鎖されたノードで実行されている場合、オペレーターは自動的にスイッチオーバーを実行します。
4. クラスターを 1 つのインスタンスにスケールダウンします。これにより、古いインスタンスが削除されます。
5. 新しいプライマリを新しいノードで実行したまま、古いプライマリのノードを正常にドレインできるようになりました。

4.29 Postgres サービスの公開

このセクションでは、PostgreSQL サービスを外部に公開し、NGINX Ingress Controller を使用して **Kubernetes クラスターの外部から** PostgreSQL データベースにアクセスできるようにする方法について説明します。

> クイックスタート に従っている場合は、default 名前空間の

cluster-example-rw (プライマリ) および cluster-example-r (読み取り専用) サービスを介してクラスター内でアクセスできるデータベースが必要です。どちらのサービスもポート 5432 を使用します。

ポート 5432 の外部アクセスからプライマリインスタンスにアクセスできるようにしたいとします。 Kubernetes インフラストラクチャに移行する場合の典型的なユースケースは、実際には、簡単にまたは持続的に「コンテナー化」できない**レガシーアプリケーション**に代表されるものです。賢明な回避策は、仮想マシンまたは物理サーバーに存在する可能性が最も高いアプリケーションが、同じネットワーク内の Kubernetes クラスター内の PostgreSQL データベースにアクセスできるようにすることです。

> **警告:** パブリック ネットワークからデータベースへのアクセスを許可すると、データベースが悪意のあるユーザーからの潜在的な攻撃にさらされる可能性があります。外部アクセスを許可する前にデータベースを保護するか、Kubernetes クラスターがプライベート ネットワークからのみ到達可能であることを確認してください。

この例では、 NGINX Ingress Controller を使用します。これは、Kubernetes プロジェクトによって直接維持され、すべての Kubernetes クラスターでセットアップできるためです。他の多くのコントローラーが利用可能です (Kubernetes documentation を参照)

包括的なリストについては)。

次のように仮定します。

- NGINX Ingress コントローラーがデプロイされ、正常に動作します
- クラスター内にタイプ `LoadBalancer` のサービスを作成することが可能です

重要: Ingress は、HTTP および HTTPS トラフィックを公開するためにのみ必要です。 NGINX Ingress コントローラーは可能ですが、すべての Ingress オブジェクトが任意のポートまたはプロトコルを公開できるわけではありません。

最初のステップは、`tcp-services` `ConfigMap` を作成することです。このデータフィールドには、外部に公開されたポートと名前空間、サービス、および内部で指すポートに関する情報が含まれています。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: tcp-services
  namespace: ingress-nginx
data:
  5432: default/cluster-example-rw:5432
```

次に、ドキュメントで提案されているように NGINX Ingress Controller をインストールした場合は、`ingress-nginx` サービスが必要です。公開するには、5432 ポートを `ingress-nginx` サービスに追加する必要があります。 Ingress は、ポート 5432 の着信接続をデータベースにリダイレクトします。

```
  apiVersion: v1
  kind: Service
  metadata:
    name: ingress-nginx
```

(次のページに続く)

(前のページからの続き)

```
    namespace: ingress-nginx
    labels:
      app.kubernetes.io/name: ingress-nginx
      app.kubernetes.io/part-of: ingress-nginx
  spec:
    type: LoadBalancer
    ports:
      - name: http
        port: 80
        targetPort: 80
        protocol: TCP
      - name: https
        port: 443
        targetPort: 443
        protocol: TCP
      - name: postgres
        port: 5432
        targetPort: 5432
        protocol: TCP
    selector:
      app.kubernetes.io/name: ingress-nginx
      app.kubernetes.io/part-of: ingress-nginx

:ref:`cluster-expose-service.yaml <samples/cluster-expose-service.yaml>`  を使用して、\
 ``kubectl`` を使用して適用できます。
```

> **警告:** このファイルを直接適用すると、Ingress の *ConfigMap* と *Service* の以前の変更が上書きされます

これで、Kubernetes クラスターの外部から PostgreSQL クラスターにアクセスできるようになります。

重要: Ingress からの接続を許可するように *pg_hba* を設定してください。

4.29.1 Minikube でのテスト

Minikube では、以下を実行するイングレスコントローラーをセットアップできます。

```
minikube addons enable ingress
```

次に、`tcp-service` ConfigMap にパッチを適用して、Ingress のポート 5432 の接続をプライマリにリダイレクトします。

```
kubectl patch configmap tcp-services -n kube-system \
  --patch {"data":{"5432":"default/cluster-example-rw:5432"}}
```

その後、デプロイメントにパッチを適用して、ポート 5432 でのアクセスを許可できます。次の内容で patch.yaml という名前のファイルを作成します。

```
spec:
  template:
    spec:
      containers:
      - name: nginx-ingress-controller
        ports:
         - containerPort: 5432
           hostPort: 5432
```

それを nginx-ingress-controller deployment に適用します。

```
kubectl patch deployment nginx-ingress-controller --patch "$(cat patch.yaml)" -n kube-
↪system
```

以下を実行しているマシンからプライマリにアクセスできます。

```
psql -h $(minikube ip) -p 5432 -U postgres
```

4.30 CloudNativePG プラグイン

CloudNativePG は、kubectl が Kubernetes でクラスターを管理するためのプラグインを提供します。

4.30.1 インストール

cnpg プラグインは、さまざまな方法でインストールできます。

注釈: エアギャップ システムの場合、以前にダウンロードしたファイルを使用してパッケージ マネージャーを介してインストールするのが適切なオプションです。

インストールスクリプト経由

```
curl -sSfL \
  https://github.com/cloudnative-pg/cloudnative-pg/raw/main/hack/install-cnpg-plugin.
↪sh | \
  sudo sh -s -- -b /usr/local/bin
```

Debian または RedHat パッケージの使用

> releases section of the GitHub repository では、関心のある任意のリリースに移動でき (CloudNativePG

オペレーターと同じか新しいリリースを選択)、その中に **Assets** セクションがあります。そのセクションには、さまざまなシステム用のビルド済みパッケージがあります。その結果、標準的な方法と指示に従ってシステムにインストールできます。

Debian パッケージ

たとえば、Intel ベースの 64 ビット サーバー用に、プラグインの 1.18.1 リリースをインストールしてみましょう。まず、正しい .deb ファイルをダウンロードします。

```
$ wget https://github.com/cloudnative-pg/cloudnative-pg/releases/download/v1.18.1/
↪kubectl-cnpg_1.18.1_linux_x86_64.deb
```

次に、dpkg を使用してローカル ファイルからインストールします。

```
$ dpkg -i kubectl-cnpg_1.18.1_linux_x86_64.deb
(Reading database ... 16102 files and directories currently installed.)
Preparing to unpack kubectl-cnpg_1.18.1_linux_x86_64.deb ...
Unpacking cnpg (1.18.1) over (1.18.1) ...
Setting up cnpg (1.18.1) ...
```

RPM パッケージ

.deb パッケージの例のように、Intel 64 ビット マシン用の 1.18.1 リリースをインストールしましょう。ファイル名を提供する --output フラグに注意してください。

```
curl -L https://github.com/cloudnative-pg/cloudnative-pg/releases/download/v1.18.1/
↪kubectl-cnpg_1.18.1_linux_x86_64.rpm \
  --output kube-plugin.rpm
```

次に、yum をインストールすると、使用する準備が整います。

```
$ yum --disablerepo=* localinstall kube-plugin.rpm
yum --disablerepo=* localinstall kube-plugin.rpm
Failed to set locale, defaulting to C.UTF-8
Dependencies resolved.
================================================================================
 Package            Architecture         Version                  Repository
         Size
================================================================================
Installing:
 cnpg               x86_64               1.18.1-1                 @commandline
         14 M

Transaction Summary
================================================================================
Install  1 Package

Total size: 14 M
Installed size: 43 M
Is this ok [y/N]: y
```

クルーの使用

クルーの使用 が既にインストールされている場合は、次を実行するだけです。

```
kubectl krew install cnpg
```

サポートされているアーキテクチャ

CloudNativePG プラグインは現在、次のオペレーティングシステムとアーキテクチャ用に構築されています。

- Linux * amd64 * arm 5/6/7 * arm64 * s390x * ppc64le
- macOS * amd64 * arm64
- Windows * 386 * amd64 * アーム 5/6/7 * arm64

4.30.2 使用

プラグインをインストールして展開したら、次のように使用を開始できます。

```
kubectl cnpg <command> <args...>
```

インストール マニフェストの生成

cnpg プラグインを使用して、オペレーターのインストール用の YAML マニフェストを生成できます。通常、このオプションは、レプリカの数、インストール名前空間、監視する名前空間などの一部のデフォルト構成をオーバーライドする場合に使用されます。

詳細と利用可能なオプションについては、次を実行してください。

```
kubectl cnpg install generate --help
```

主なオプションは次のとおりです。

- -n : オペレーターをインストールする名前空間 (デフォルト: cnpg-system)
- --replicas : デプロイ内のレプリカの数
- --version : 1.17 など、インストールするオペレーターのマイナーバージョン。マイナーバージョンが指定されている場合、プラグインはそのマイナーバージョンの最新のパッチバージョンをインストールします。バージョンが指定されていない場合、プラグインはオペレーターの最新の MAJOR.MINOR.PATCH バージョンをインストールします。
- --watch-namespace : 監視する名前空間を含むカンマ区切りの文字列 (デフォルトではすべての名前空間)

オペレーターをインストールする YAML マニフェストを生成する generate コマンドの例は次のとおりです。

```
kubectl cnpg install generate \
  -n king \
  --version 1.17 \
  --replicas 3 \
  --watch-namespace "albert, bb, freddie" \
  > operator.yaml
```

上記のコマンドのフラグには、次の意味があります。

- -n king CNPG オペレーターを king 名前空間にインストールします
- --version 1.17 は、マイナー バージョン 1.17 の最新のパッチバージョンをインストールします。
- --replicas 3 は、3 つのレプリカを持つオペレーターをインストールします
- --watch-namespaces "albert, bb, freddie" では、albert 、bb 、および freddie 名前空間のみの変更を監視するオペレーターがいます。

スターテス

status コマンドは、次のようなクラスターの現在のステータスの概要を提供します。

- **一般情報**：クラスターの名前、PostgreSQL のシステム ID、インスタンス数、現在のタイムライン、および WAL 内の位置
- **バックアップ**：回復可能ポイント、およびプライマリからの pg_stat_archiver ビューによって返される WAL アーカイブステータス、またはレプリカ クラスタの場合は指定されたプライマリ
- **ストリーミングレプリケーション**：プライマリ インスタンスの pg_stat_replication ビューから直接取得した情報
- **インスタンス**：各インスタンスマネージャーによって直接取得される、各 Postgres インスタンスに関する情報。スタンバイの場合、Current LSN フィールドは、リカバリ中にリプレイされた最新のログ先行書き込みの場所に対応します (リプレイ LSN)。

重要: 上記のステータス情報は、さまざまな時点でさまざまな場所で取得されているため、返される値に若干の一貫性がありません。たとえば、メイン ヘッダーの *Current Write LSN* の場所は、2 つの異なる時間間隔で取得されるため、インスタンス ステータスの *Current LSN* フィールドとは異なる場合があります。

```
kubectl cnpg status sandbox
```

```
Cluster in healthy state
Name:               sandbox
Namespace:          default
System ID:          7039966298120953877
PostgreSQL Image:   ghcr.io/cloudnative-pg/postgresql:15.2
Primary instance:   sandbox-2
Instances:          3
Ready instances:    3
Current Write LSN:  3AF/EAFA6168 (Timeline: 8 - WAL File: 000000080000003AF00000075)

Continuous Backup status
First Point of Recoverability:  Not Available
Working WAL archiving:          OK
Last Archived WAL:              000000080000003AE00000079   @   2021-12-14T10:16:29.
↪340047Z
Last Failed WAL: -

Certificates Status
Certificate Name             Expiration Date                Days Left Until Expiration
- ---------------             ---------------                --------------------------
↪-
cluster-example-ca           2022-05-05 15:02:42 +0000 UTC  87.23
cluster-example-replication  2022-05-05 15:02:42 +0000 UTC  87.23
```

(次のページに続く)

(前のページからの続き)

```
cluster-example-server       2022-05-05 15:02:42 +0000 UTC  87.23

Streaming Replication status
Name       Sent LSN      Write LSN     Flush LSN     Replay LSN    Write Lag       ␣
↪Flush Lag        Replay Lag       State      Sync State  Sync Priority
- ---       --------      ---------     ---------     ----------    ---------        -
↪--------        ----------       -----      ----------  -------------
sandbox-1  3AF/EB0524F0  3AF/EB011760  3AF/EAFEDE50  3AF/EAFEDE50  00:00:00.004461 ␣
↪00:00:00.007901  00:00:00.007901  streaming  quorum      1
sandbox-3  3AF/EB0524F0  3AF/EB030B00  3AF/EB030B00  3AF/EB011760  00:00:00.000977 ␣
↪00:00:00.004194  00:00:00.008252  streaming  quorum      1

Instances status
Name       Database Size  Current LSN   Replication role  Status  QoS         Manager␣
↪Version
- ---       -------------  -----------   ----------------  ------  ---         -------
↪--------
sandbox-1  302 GB         3AF/E9FFFFE0  Standby (sync)    OK      Guaranteed  1.11.0
sandbox-2  302 GB         3AF/EAFA6168  Primary           OK      Guaranteed  1.11.0
sandbox-3  302 GB         3AF/EBAD5D18  Standby (sync)    OK      Guaranteed  1.11.0
```

--verbose または単に -v を追加することで、ステータスのより詳細なバージョンを取得することもできます

```
kubectl cnpg status sandbox --verbose
```

```
Cluster in healthy state
Name:               sandbox
Namespace:          default
System ID:          7039966298120953877
PostgreSQL Image:   ghcr.io/cloudnative-pg/postgresql:15.2
Primary instance:   sandbox-2
Instances:          3
Ready instances:    3
Current Write LSN:  3B1/61DE3158 (Timeline: 8 - WAL File: 00000008000003B100000030)

PostgreSQL Configuration
archive_command = /controller/manager wal-archive --log-destination /controller/log/
↪postgres.json %p
archive_mode = on
archive_timeout = 5min
checkpoint_completion_target = 0.9
checkpoint_timeout = 900s
cluster_name = sandbox
```

(次のページに続く)

(前のページからの続き)

```
dynamic_shared_memory_type = sysv
full_page_writes = on
hot_standby = true
jit = on
listen_addresses = *
log_autovacuum_min_duration = 1s
log_checkpoints = on
log_destination = csvlog
log_directory = /controller/log
log_filename = postgres
log_lock_waits = on
log_min_duration_statement = 1000
log_rotation_age = 0
log_rotation_size = 0
log_statement = ddl
log_temp_files = 1024
log_truncate_on_rotation = false
logging_collector = on
maintenance_work_mem = 2GB
max_connections = 1000
max_parallel_workers = 32
max_replication_slots = 32
max_wal_size = 15GB
max_worker_processes = 32
pg_stat_statements.max = 10000
pg_stat_statements.track = all
port = 5432
shared_buffers = 16GB
shared_memory_type = sysv
shared_preload_libraries = pg_stat_statements
ssl = on
ssl_ca_file = /controller/certificates/client-ca.crt
ssl_cert_file = /controller/certificates/server.crt
ssl_key_file = /controller/certificates/server.key
synchronous_standby_names = ANY 1 ("sandbox-1","sandbox-3")
unix_socket_directories = /controller/run
wal_keep_size = 512MB
wal_level = logical
wal_log_hints = on
cnpg.config_sha256 = 3cfa683e23fe513afaee7c97b50ce0628e0cc634bca8b096517538a9a4428efc

PostgreSQL HBA Rules

#  Grant local access
```

(次のページに続く)

(前のページからの続き)

```
local all all peer map=local

#  Require client certificate authentication for the streaming_replica user

hostssl postgres streaming_replica all cert
hostssl replication streaming_replica all cert
hostssl all cnpg_pooler_pgbouncer all cert

#  Otherwise use the default authentication method

host all all all scram-sha-256

Continuous Backup status
First Point of Recoverability:  Not Available
Working WAL archiving:          OK
Last Archived WAL:              00000008000003B00000001D   @   2021-12-14T10:20:42.
↪272815Z
Last Failed WAL: -

Streaming Replication status
Name       Sent LSN      Write LSN     Flush LSN     Replay LSN    Write Lag       ␣
↪Flush Lag        Replay Lag       State      Sync State  Sync Priority
- ---       --------      ---------     ---------     ----------    ---------        -
↪--------        ----------       -----      ----------  --------------
sandbox-1  3B1/61E26448  3B1/61DF82F0  3B1/61DF82F0  3B1/61DF82F0  00:00:00.000333 ␣
↪00:00:00.000333  00:00:00.005484  streaming  quorum      1
sandbox-3  3B1/61E26448  3B1/61E26448  3B1/61DF82F0  3B1/61DF82F0  00:00:00.000756 ␣
↪00:00:00.000756  00:00:00.000756  streaming  quorum      1

Instances status
Name       Database Size  Current LSN   Replication role  Status  QoS         Manager␣
↪Version
- ---       -------------  -----------   ----------------  ------  ---         -------
↪--------
sandbox-1                 3B1/610204B8  Standby (sync)    OK      Guaranteed  1.11.0
sandbox-2                 3B1/61DE3158  Primary           OK      Guaranteed  1.11.0
sandbox-3                 3B1/62618470  Standby (sync)    OK      Guaranteed  1.11.0
```

このコマンドは、yaml および json 形式の出力もサポートしています。

促進

このコマンドの意味は、クラスター内のポッドをプライマリーに promote することです。そのため、メンテナンス作業を開始したり、クラスターで切り替え状況をテストしたりできます。

```
kubectl cnpg promote cluster-example cluster-example-2
```

または、インスタンス ノード番号を使用して昇格することもできます

```
kubectl cnpg promote cluster-example 2
```

証明書

CloudNativePG オペレーターを使用して作成されたクラスターは、CA と連携して TLS 認証証明書に署名します。

証明書を取得するには、資格情報を格納するためのシークレットの名前、クラスター名、およびこの証明書のユーザーを指定する必要があります。

```
kubectl cnpg certificate cluster-cert --cnpg-cluster cluster-example --cnpg-user␣
↪appuser
```

secrete が作成されたら、kubectl を使用して取得できます。

```
kubectl get secret cluster-cert
```

そして、次のコマンドを使用してプレーンテキストで同じ内容を取得します。

```
kubectl get secret cluster-cert -o json | jq -r .data | map(@base64d) | .[]
```

再起動

kubectl cnpg restart コマンドは、次の 2 つのケースで使用できます。

- 特定のクラスターのロールアウトの再起動を調整するようにオペレーターに要求する。これは、カスタム監視クエリを含む ConfigMap などのクラスター依存オブジェクトに構成変更を適用するのに役立ちます。
- インスタンスがクラスターのプライマリの場合はインプレースで、レプリカの場合はポッドを削除して再作成する、単一インスタンスの再起動を要求します。

```
#  this command will restart a whole cluster in a rollout fashion

kubectl cnpg restart [clusterName]
```

(次のページに続く)

(前のページからの続き)

```
#  this command will restart a single instance, according to the policy above

kubectl cnpg restart [clusterName] [pod]
```

インプレース再起動が要求されたが、スイッチオーバーなしでは変更を適用できない場合、インプレース再起動よりもスイッチオーバーが優先されます。これの一般的なケースは、PostgreSQL イメージのマイナーアップグレードです。

注釈: インスタンスによって ConfigMap とシークレットを **自動的に** リロードする場合は、キー *cnpg.io/reload* のラベルを追加できます。

リロード

`kubectl cnpg reload` コマンドは、特定のクラスターの調整ループをトリガーするようにオペレーターに要求します。これは、カスタム監視クエリを含む ConfigMap などのクラスター依存オブジェクトに構成変更を適用するのに役立ちます。

次のコマンドは、特定のクラスターのすべての構成を再読み込みします。

```
kubectl cnpg reload [cluster_name]
```

メンテナンス

`kubectl cnpg maintenance` コマンドは、名前空間全体で 1 つ以上のクラスターを変更し、メンテナンスウィンドウの値を設定するのに役立ちます。次のフィールドが変更されます。

- .spec.nodeMaintenanceWindow.inProgress
- .spec.nodeMaintenanceWindow.reusePVC

引数 `set` および `unset` として受け入れます。これを使用して、`set` の場合は `inProgress` を `true` に設定し、`unset` の場合は `false` に設定します。

デフォルトでは、`--reusePVC` フラグが渡されない限り、`reusePVC` は常に `false` に設定されます。

プラグインは、変更するクラスターのリストとその新しい値の確認を求めます。これが受け入れられると、このアクションはリスト内のすべてのクラスターに適用されます。

Kubernetes クラスター内のすべての PostgreSQL をメンテナンスに設定する場合は、次のコマンドを記述するだけです。

```
kubectl cnpg maintenance set --all-namespaces
```

そして、更新するすべてのクラスターのリストが表示されます

```
The following are the new values for the clusters
Namespace  Cluster Name     Maintenance  reusePVC
- --------  ------------     -----------  --------
default    cluster-example  true         false
default    pg-backup        true         false
test       cluster-example  true         false
Do you want to proceed? [y/n]: y
```

報告

`kubectl cnpg report` コマンドは、さまざまな情報を ZIP ファイルにまとめます。本番環境でクラスターの問題をデバッグするために必要なコンテキストを提供することを目的としています。

`operator` と `cluster` の 2 つのサブコマンドがあります。

レポート オペレータ

`operator` サブコマンドは、オペレーターのデプロイ、構成、およびイベントに関する情報を提供するようオペレーターに要求します。

重要: Secrets と ConfigMaps のすべての機密情報は編集済みです。データ マップには **キー** が表示されますが、値は空になります。フラグ *-S* / *--stopRedaction* は編集を無効にし、値を表示します。自己責任でのみ使用してください。これにより、個人データが共有されます。

注釈: デフォルトでは、オペレーター ログは収集されませんが、*--logs* フラグを使用してオペレーター ログの収集を有効にすることができます

- **デプロイメント情報** : operator Deployment および operator Pod
- **構成** : オペレーター名前空間の Secret と ConfigMap
- **events** : オペレーター名前空間のイベント
- **webhook 構成** : Webhook 構成の変更と検証
- **webhook サービス** : Webhook サービス
- **logs** : JSON 行形式のオペレーター Pod (オプション、デフォルトではオフ) のログ

このコマンドは、YAML 形式のさまざまなマニフェストを含む ZIP ファイルを生成します (既定では、`-o` フラグを使用して JSON に設定できます)。 `-f` フラグを使用して、結果ファイルに明示的に名前を付けます。 `-f` フラグが使用されていない場合、デフォルトのタイムスタンプ付きファイル名が zip ファイル用に作成されます。

注釈: レポート プラグインは *kubectl* 規則に従い、名前空間によって制約されたオブジェクトを探します。CNPG Operator は通常、クラスターと同じ名前空間にはインストールされません。例えば。デフォルトのインストール名前空間は cnpg-system です

```
kubectl cnpg report operator -n <namespace>
```

結果は

```
Successfully written report to "report_operator_<TIMESTAMP>.zip" (format: "yaml")
```

-f フラグが設定されている場合:

```
kubectl cnpg report operator -n <namespace> -f reportRedacted.zip
```

ファイルを解凍すると、タイムスタンプ付きの最上位フォルダーが作成され、ディレクトリが整理されます。

```
unzip reportRedacted.zip
```

結果は次のとおりです。

```
Archive:  reportRedacted.zip
   creating: report_operator_<TIMESTAMP>/
   creating: report_operator_<TIMESTAMP>/manifests/
  inflating: report_operator_<TIMESTAMP>/manifests/deployment.yaml
  inflating: report_operator_<TIMESTAMP>/manifests/operator-pod.yaml
  inflating: report_operator_<TIMESTAMP>/manifests/events.yaml
  inflating: report_operator_<TIMESTAMP>/manifests/validating-webhook-configuration.
↪yaml
  inflating: report_operator_<TIMESTAMP>/manifests/mutating-webhook-configuration.yaml
  inflating: report_operator_<TIMESTAMP>/manifests/webhook-service.yaml
  inflating: report_operator_<TIMESTAMP>/manifests/cnpg-ca-secret.yaml
  inflating: report_operator_<TIMESTAMP>/manifests/cnpg-webhook-cert.yaml
```

機密情報が編集済みであることを確認できます。

```
cd report_operator_<TIMESTAMP>/manifests/
head cnpg-ca-secret.yaml
```

```
data:
  ca.crt: ""
  ca.key: ""
metadata:
  creationTimestamp: "2022-03-22T10:42:28Z"
  managedFields:
  - apiVersion: v1
```

(次のページに続く)

(前のページからの続き)

```
    fieldsType: FieldsV1
    fieldsV1:
```

-S (--stopRedaction) オプションを有効にすると、シークレットが表示されます。

```
kubectl cnpg report operator -n <namespace> -f reportNonRedacted.zip -S
```

機密情報を表示しようとしていることを知らせるリマインダーが表示されます。

```
WARNING: secret Redaction is OFF. Use it with caution
Successfully written report to "reportNonRedacted.zip" (format: "yaml")
```

```
unzip reportNonRedacted.zip
head cnpg-ca-secret.yaml
```

```
data:
  ca.crt: LS0tLS1CRUdJTiBD···
  ca.key: LS0tLS1CRUdJTiBF···
metadata:
  creationTimestamp: "2022-03-22T10:42:28Z"
  managedFields:
  - apiVersion: v1
    fieldsType: FieldsV1
```

レポート クラスター

cluster サブコマンドは、以下を収集します。

- **クラスターリソース** : クラスター情報、kubectl get cluster -o yaml と同じ
- **clusterpods** : クラスター名と一致するクラスター名前空間内のポッド
- **クラスタージョブ** : クラスター名と一致するクラスター名前空間内のジョブ (存在する場合)
- **events** : クラスター名前空間のイベント
- **ポッドログ** : JSON 行形式のクラスター ポッド (オプション、デフォルトではオフ) のログ
- **ジョブログ** : ジョブによって作成された Pod のログ (オプション、デフォルトではオフ)、JSON 行形式

cluster サブコマンドは、operator と同様に、-f および-o フラグを受け入れます。 -f フラグが使用されていない場合、デフォルトのタイムスタンプ付きレポート名が使用されます。クラスター情報には構成 Secrets / ConfigMaps が含まれていないため、-S は無効になっていることに注意してください。

注釈: デフォルトでは、クラスター ログは収集されませんが、*--logs* フラグを使用してクラスター ログ収集を有効にすることができます

使用法：

```
kubectl cnpg report cluster <clusterName> [flags]
```

operator サブコマンドとは異なり、cluster サブコマンドの場合、クラスターがデフォルトのものでない限り、クラスター名とおそらく名前空間を提供する必要があることに注意してください。

```
kubectl cnpg report cluster example -f report.zip -n example_namespace
```

その後：

```
unzip report.zip
```

```
Archive:  report.zip
   creating: report_cluster_example_<TIMESTAMP>/
   creating: report_cluster_example_<TIMESTAMP>/manifests/
  inflating: report_cluster_example_<TIMESTAMP>/manifests/cluster.yaml
  inflating: report_cluster_example_<TIMESTAMP>/manifests/cluster-pods.yaml
  inflating: report_cluster_example_<TIMESTAMP>/manifests/cluster-jobs.yaml
  inflating: report_cluster_example_<TIMESTAMP>/manifests/events.yaml
```

--logs フラグを使用して、ポッドとジョブ ログを ZIP に追加できることに注意してください。

```
kubectl cnpg report cluster example -n example_namespace --logs
```

結果は次のとおりです。

```
Successfully written report to "report_cluster_example_<TIMESTAMP>.zip" (format: "yaml
↪")
```

```
unzip report_cluster_<TIMESTAMP>.zip
```

```
Archive:  report_cluster_example_<TIMESTAMP>.zip
   creating: report_cluster_example_<TIMESTAMP>/
   creating: report_cluster_example_<TIMESTAMP>/manifests/
  inflating: report_cluster_example_<TIMESTAMP>/manifests/cluster.yaml
  inflating: report_cluster_example_<TIMESTAMP>/manifests/cluster-pods.yaml
  inflating: report_cluster_example_<TIMESTAMP>/manifests/cluster-jobs.yaml
  inflating: report_cluster_example_<TIMESTAMP>/manifests/events.yaml
   creating: report_cluster_example_<TIMESTAMP>/logs/
  inflating: report_cluster_example_<TIMESTAMP>/logs/cluster-example-full-1.jsonl
   creating: report_cluster_example_<TIMESTAMP>/job-logs/
  inflating: report_cluster_example_<TIMESTAMP>/job-logs/cluster-example-full-1-
↪initdb-qnnvw.jsonl
  inflating: report_cluster_example_<TIMESTAMP>/job-logs/cluster-example-full-2-join-
↪tvj8r.jsonl
```

(次のページに続く)

(前のページからの続き)

```
``` ### 破棄

`kubectl cnpg destroy` コマンドは、インスタンスと関連するすべての PVC を Kubernetes クラスターから削除するのに役立ちます。

オプションの `--keep-pvc` フラグを指定すると、インスタンスによって設定されたすべての␣
↪`metadata.ownerReferences` を削除しながら、PVC を保持できます。さらに、PVC の `cnpg.io/
↪pvcStatus` ラベルは `ready` から `detached` に変更され、使用されなくなったことを示します。

`--keep-pvc` フラグを指定せずにコマンドを再度実行すると、切り離された PVC が削除されます。

使用法：
```

kubectl cnpg destroy [CLUSTER_NAME] [INSTANCE_ID]

```
次の例では、`cluster-example-2` ポッドと関連する PVC を削除します。
```

kubectl cnpg destroy cluster-example 2

```
### クラスターの休止状態

データを保持しながら CloudNativePG `Cluster` の実行を一時停止し、後でそのアクティビティを再開したい場合があります。この機能を **クラスター休止状態** と呼んでいます。

ハイバネーションは、`kubectl cnpg hibernate [on|off]` コマンドを介してのみ利用できます。

CloudNativePG クラスターを休止状態にすることは、PostgreSQL プライマリ インスタンスに属する␣
↪PVC を除いて、クラスターによって生成されたすべてのリソースを破棄することを意味します。

以下を使用してクラスターを休止状態にすることができます。
```

kubectl cnpg hibernate on

```
この意志：

1. すべての PostgreSQL インスタンスをシャットダウンします

2. プライマリ インスタンスのデータを含む PVC を切り離し、最新のデータベース ステータスと最新のクラスタ構成で注釈を付けます。

3. 前述の PVC を除く、生成されたすべてのリソースを含む `Cluster` リソースを削除します。

休止状態の場合、CloudNativePG クラスターは、`PGDATA` を含むものに `pg_controldata` のコンテンツを含む最新の利用可能なステータスで注釈が付けられた PVC のグループだけで表されます。
```

(次のページに続く)
```

(前のページからの続き)

```
.. Warning::
フェンシングも休止手順の一部であるため、フェンシングされたインスタンスを持つクラスターは休止状態にできません。

エラーが発生した場合、オペレーターは手順を元に戻すことができません。次のコマンドを使用して操作を強制することもできます。
```

kubectl cnpg hibernate on cluster-example –force

```
休止状態のクラスターは、次の方法で再開できます。
```

kubectl cnpg hibernate off

```
クラスターが休止状態になると、最後の構成とシャットダウン後の PostgreSQL の状態を表示できます。それは次のように行うことができます:
```

kubectl cnpg hibernate status

```
### pgbench によるデータベースのベンチマーク

Pgbench は、次のコマンドを使用して既存の PostgreSQL クラスターで実行できます。
```

kubectl cnpg pgbench – –time 30 –client 1 –jobs 1

```
詳細については、 :ref:`Benchmarking pgbench section <benchmarking.md#pgbench>` を参照してください。

### fio によるストレージのベンチマーク

次のコマンドを使用して、既存のストレージ クラスで fio を実行できます。
```

kubectl cnpg fio -n

```
詳細については、 :ref:`Benchmarking fio section <benchmarking.md#fio>` を参照してください。

### 新しい基本バックアップのリクエスト

`kubectl cnpg backup` コマンドは、新しい `Backup` リソースを作成することにより、既存の␣
↪Postgres クラスターの新しい物理ベース バックアップを要求します。

次の例では、特定のクラスターのオンデマンド バックアップを要求します。

```shell
kubectl cnpg backup [cluster_name]
```
```

作成されたバックアップは、リクエスト時間に基づいて名前が付けられます。

edb_notranlate_50 デフォルトでは、新しく作成されたバックアップは、クラスターで定義されたバックアップターゲット ポリシーを使用して、実行するインスタンスを選択します。--backup-target オプションを使用して、このポリシーをオーバーライドすることもできます。

> バックアップとリカバリ を参照してください

バックアップ ターゲットの詳細については、

psql の起動

kubectl cnpg psql コマンドは、実際のポッドから実行しているかのように、既存の Postgres クラスターに接続された新しい PostgreSQL インタラクティブ フロントエンドプロセス (psql) を開始します。これは、postgres ユーザーを使用することを意味します。

重要: *postgres* ユーザーとして接続するため、本番環境では、このメソッドは許可された担当者のみが細心の注意を払って使用する必要があります。

```
kubectl cnpg psql cluster-example

psql (15.2 (Debian 15.2-1.pgdg110+1))
Type "help" for help.

postgres=#
```

デフォルトでは、コマンドはプライマリインスタンスに接続します。ユーザーは、--replica オプションを使用して、レプリカに対して作業することを選択できます。

```
kubectl cnpg psql --replica cluster-example
psql (15.2 (Debian 15.2-1.pgdg110+1))

Type "help" for help.

postgres=# select pg_is_in_recovery();
 pg_is_in_recovery
- ------------------
 t
(1 row)

postgres=# \q
```

このコマンドは kubectl exec を開始します。正しく機能するには、PATH 変数で kubectl 実行可能ファイルにアクセスできる必要があります。

4.31 自動フェイルオーバー

プライマリで spec.failoverDelay (デフォルトでは 0 秒) より長く予期しないエラーが発生した場合、クラスターは**フェイルオーバーモード**になります。これは、たとえば次の場合に発生する可能性があります。

- プライマリ ポッドにディスク障害があります
- プライマリ ポッドが削除されます
- プライマリの postgres コンテナに何らかの持続的な障害がある

フェールオーバーシナリオでは、プライマリが正常に動作しているとは想定できません。

上記のようなケースの後、プライマリポッドの準備プローブが失敗し始めます。これは、コントローラーの調整ループで取得されます。コントローラは、次の 2 つの手順でフェイルオーバー プロセスを開始します。

1. まず、TargetPrimary を pending としてマークします。この状態の変更により、プライマリポッドが強制的にシャットダウンされ、レプリカの WAL レシーバーが確実に停止します。クラスタはフェールオーバー フェーズ (「フェールオーバー」) でマークされます。
2. すべての WAL レシーバーが停止すると、リーダーの選挙が行われ、新しいプライマリが指定されます。選択したインスタンスがプライマリへの昇格を開始し、これが完了すると、クラスターは通常の操作を再開します。一方、以前のプライマリポッドは再起動し、プライマリではなくなったことを検出し、レプリカノードになります。

重要: 2 段階の手順により、WAL レシーバーが正常に停止し、失敗したプライマリが再起動時に WAL のストリーミングを再開しないことが保証されます。これらのセーフガードにより、新しいプライマリとレプリカの間のタイムラインの不一致が防止されます。

障害のあるプライマリがシャットダウンされている間:

1. 最初に .spec.switchoverDelay 秒のタイムアウトで PostgreSQL の**高速シャットダウン**を試みます。この正常なシャットダウンにより、保留中の WAL のアーカイブが試行されます。
2. 高速シャットダウンが失敗した場合、またはタイムアウトを超えた場合、PostgreSQL の **即時シャットダウン** が開始されます。

4.31.1 RTO と RPO への影響

フェールオーバーにより、サービスが影響を受けたり、データが失われたりする可能性があります。

1. プライマリに障害が発生し始め、コントローラーがフェイルオーバー手順を開始する前に、転送中のクエリ、WAL 書き込み、チェックポイント、および同様の操作が失敗する可能性があります。
2. 高速シャットダウンコマンドが発行されると、クラスターは接続を受け入れなくなるため、サービスは影響を受けますが、データが失われることはありません。
3. 高速シャットダウンが失敗した場合、即時シャットダウンにより、保留中のプロセス (WAL 書き込みを含む) が停止されます。データが失われる可能性があります。

4. プライマリがシャットダウンし、新しいプライマリがまだ開始されていない間、クラスタはプライマリなしで動作するため、障害が発生しますが、データが失われることはありません。

注釈: スイッチオーバーの場合と同様に、高速シャットダウンを制御するタイムアウトは *.spec.switchoverDelay* によって設定されます。高速シャットダウンの時間を長くすることは、RPO の観点からはより安全ですが、通常の操作への復帰が遅れる可能性があり、RTO に悪影響を及ぼします。

警告: インスタンスマネージャーのインプレース更新 で既に述べたように

スイッチオーバー プロセスを説明するとき、`.spec.switchoverDelay` オプションは PostgreSQL データベースの RPO と RTO に影響します。低い値に設定すると、RPO よりも RTO が優先される可能性がありますが、クラスター レベルやバックアップレベルでデータが失われる可能性があります。反対に、高い値に設定すると、スイッチオーバー中にクラスターがアクティブなプライマリなしで長時間放置されたままになるため、データ損失のリスクがなくなる可能性があります。

4.31.2 フェイルオーバーの遅延

上記で予想したように、`spec.failoverDelay` オプションを使用すると、プライマリが異常であることが検出された後、フェイルオーバー手順の開始を数秒遅らせることができます。デフォルトでは、この設定は `0` に設定されており、フェイルオーバー手順がすぐにトリガーされます。

新しいプライマリへのフェイルオーバーは、プライマリがオンラインに戻るのを待つよりも混乱を招く場合があります。これは、複数の層 (つまり、ダウンストリームの論理サブスクライバー) が影響を受けるネットワークの中断、またはフェイルオーバーの実行時間が予想される停止よりも長い場合に特に当てはまります。

フェールオーバーを遅らせる新しい構成オプションを有効にすると、短期間のネットワークまたはノードの不安定性による早すぎるフェールオーバーを防ぐメカニズムが提供されます。

4.32 トラブルシューティング

このページでは、Kubernetes クラスターのデプロイで CloudNativePG をトラブルシューティングする方法に関する基本的な情報を見つけることができます。

ヒント: Kubernetes 管理者として、 :ref:``kubectl` に *cnpg* プラグインを使用する<*kubectl* に *cnpg* プラグインを使用する>` ページをブックマークする必要があります。

4.32.1 始める前に

Kubernetes 環境

トラブルシューティング活動に違いをもたらすことができるのは、基盤となる Kubernetes システムに関する明確な情報を提供することです。

次のことを確認してください。

- 使用している Kubernetes のディストリビューションとバージョン

・PostgreSQL が稼働しているノードの仕様

- 実際の ストレージ容量の使用についてできる限り、本番環境に入る前に行ったストレージクラスとベンチマークを含みます。
- クラスターで使用している関連する Kubernetes アプリケーション (つまり、Prometheus、Grafana、Istio、Certmanager など)
- 継続的なバックアップの状況、特に適切に機能しているかどうか: 正しく機能していない場合は、必ず 緊急バックアップ を実行してください。

潜在的な中断操作を実行する前に

便利なユーティリティ

必須の `kubectl` ユーティリティに加えて、トラブルシューティングのために、システムで次のプラグイン/ユーティリティを使用できるようにすることをお勧めします。

- `kubectl` の :ref:``kubectl` に *cnpg* プラグインを使用する<*kubectl* に *cnpg* プラグインを使用する>`
- jq 、軽量で柔軟なコマンドライン JSON プロセッサ
- grep 、指定されたパターンに一致する行を含む 1

  つ以上の入力ファイルを検索します。ほとんどの *nix ディストリビューションで既に利用可能です。 Windows OS を使用している場合は、`grep` の代わりに findstr を使用するか、 wsl を直接使用できます。

好みの *nix ディストリビューションをインストールし、上記のツールを使用します。

4.32.2 緊急バックアップ

緊急事態では、メインの `app` データベースの緊急論理バックアップを作成する必要がある場合があります。

重要: 以下の手順は、緊急時にのみ実行し、組織で有効なデータ保護ポリシーの下で一時バックアップ ファイルを保持する必要があります。ダンプ ファイルは実際に *kubectl* コマンドを実行するクライアント マシンに保存されるため、すべての保護が実施され、バックアップ ファイルを保存するのに十分なスペースがあることを確認してください。

次の例は、cluster-example-1 ポッドから、cluster-example Postgres クラスター内の app データベースの論理バックアップを取得する方法を示しています。

```
kubectl exec cluster-example-1 -c postgres \
  -- pg_dump -Fc -d app > app.dump
```

注釈: 環境で使用したオブジェクトの名前を指定することで、上記のコマンドをクラスターのバックアップに簡単に適用できます。

上記のコマンドは、カスタム形式で pg_dump コマンドを発行します。これは、logical backups in PostgreSQL を取得する最も用途の広い方法です。

次のステップは、データベースを復元することです。初期化されたばかりの新しい PostgreSQL クラスターで操作していると仮定します (つまり、app データベースは空です)。

次の例は、プライマリ (new-cluster-example-1 ポッド) に接続して、new-cluster-example Postgres クラスターの app データベースで上記の論理バックアップを復元する方法を示しています。

```
kubectl exec -i new-cluster-example-1 -c postgres \
  -- pg_restore --no-owner --role=app -d app --verbose < app.dump
```

重要: このセクションの例では、推奨に従って、ダンプおよび復元するグローバル オブジェクト (データベースとロール) が他にないことを前提としています。複数の役割がある場合は、*pg_dumpall -g* を使用してバックアップを作成し、それらを新しいクラスターに手動で復元していることを確認してください。複数のデータベースがある場合は、正しい所有権を割り当てていることを確認して、一度に 1 つのデータベースで上記の操作を繰り返す必要があります。 PostgreSQL に慣れていない場合は、これらの重要な操作を専門のサポート会社の指導の下で行うことをお勧めします。

上記の手順は、将来のある段階で cnpg プラグインに統合される可能性があります。

ログ

CloudNativePG によって作成および制御されるすべてのリソースは、Kubernetes によって期待されるように標準出力に記録され、JSON format に直接記録されます。そのため、kubectl logs コマンドを使用して特定のリソースからログを取得する必要があります。

詳細については、次のように入力してください。

```
kubectl logs --help
```

ヒント: JSON ログは機械読み取りには適していますが、人間には読み取りが困難です。使いやすさを向上させるために、*jq* コマンドを使用することをお勧めします。たとえば、*kubectl logs* コマンドを | *jq -C* で **パイプ** できます。

注釈: 以下のセクションでは、CloudNativePG のトラブルシューティングに関して、さまざまなリソースに関するログを取得する方法の例をいくつか示します。

4.32.3 オペレーター情報

デフォルトでは、CloudNativePG オペレーターは Deployment として Kubernetes の cnpg-system 名前空間にインストールされます (導入の詳細 を参照)。

詳細については)。

次のコマンドを実行して、オペレーター ポッドのリストを取得できます。

```
kubectl get pods -n cnpg-system
```

注釈: 通常の状況では、*cnpg-controller-manager-* で始まる名前で識別される、オペレーターが実行されているポッドが 1 つ必要です。高可用性のためにオペレーターをセットアップした場合は、より多くのエントリーが必要です。これらのポッドは、*cnpg-controller-manager* という名前のデプロイメントによって管理されます。

以下を使用して、ポッド <POD> で実行されているオペレーターに関する関連情報を収集します。

```
kubectl describe pod -n cnpg-system <POD>
```

次に、次のコマンドを実行して、同じポッドからログを取得します。

```
kubectl logs -n cnpg-system <POD>
```

オペレーターに関する詳細情報を収集する

以下を実行して、CloudNativePG オペレーターデプロイのすべてのポッドからログを取得します (マルチ オペレーターデプロイがある場合)。

```
kubectl logs -n cnpg-system \
  deployment/cnpg-controller-manager --all-containers=true
```

Tip: 上記のコマンドに *-f* フラグを追加して、リアルタイムでログを追跡できます。

次を実行して、ログを JSON ファイルに保存します。

```
kubectl logs -n cnpg-system \
  deployment/cnpg-controller-manager --all-containers=true | \
  jq -r . > cnpg_logs.json
```

kubectl-cnpg プラグインを使用して CloudNativePG オペレーターのバージョンを取得します。

```
kubectl-cnpg status <CLUSTER>
```

出力：

```
Cluster in healthy state
Name:               cluster-example
Namespace:          default
System ID:          7044925089871458324
PostgreSQL Image:   ghcr.io/cloudnative-pg/postgresql:15.2-3
Primary instance:   cluster-example-1
Instances:          3
Ready instances:    3
Current Write LSN:  0/5000000 (Timeline: 1 - WAL File: 000000010000000000000004)

Continuous Backup status
Not configured

Streaming Replication status
Name               Sent LSN   Write LSN  Flush LSN  Replay LSN  Write Lag       Flush␣
↪Lag       Replay Lag      State      Sync State  Sync Priority
- ---               --------   ---------  ---------  ----------  ---------       -----
↪----       ----------      -----      ----------  -------------
cluster-example-2  0/5000000  0/5000000  0/5000000  0/5000000   00:00:00       ␣
↪00:00:00        00:00:00        streaming  async       0
cluster-example-3  0/5000000  0/5000000  0/5000000  0/5000000   00:00:00.10033 ␣
↪00:00:00.10033  00:00:00.10033  streaming  async       0

Instances status
Name               Database Size  Current LSN  Replication role  Status  QoS        ␣
↪Manager Version
- ---               -------------  -----------  ----------------  ------  ---        ␣
↪---------------
cluster-example-1  33 MB          0/5000000    Primary           OK      BestEffort ␣
↪1.12.0
cluster-example-2  33 MB          0/5000000    Standby (async)   OK      BestEffort ␣
↪1.12.0
cluster-example-3  33 MB          0/5000060    Standby (async)   OK      BestEffort ␣
↪1.12.0
```

4.32.4 クラスタ情報

NAMESPACE 名前空間の <CLUSTER> クラスターのステータスは、次のコマンドで確認できます。

```
kubectl get cluster -n <NAMESPACE> <CLUSTER>
```

出力：

```
NAME        AGE        INSTANCES   READY   STATUS                     PRIMARY
<CLUSTER>   10d4h3m    3           3       Cluster in healthy state   <CLUSTER>-1
```

上記の例では、3 つのインスタンスからなる正常な PostgreSQL クラスターが報告されています。これらはすべて *ready* 状態であり、<CLUSTER>-1 がプライマリです。

異常な状態の場合は、Cluster リソースのマニフェストを取得することで、さらに多くのことを発見できます。

```
kubectl get cluster -o yaml -n <NAMESPACE> <CLUSTER>
```

収集するもう 1 つの重要なコマンドは、cnpg プラグインによって提供される status コマンドです。

```
kubectl cnpg status -n <NAMESPACE> <CLUSTER>
```

Tip: *--verbose* オプションを追加すると、より多くの情報を出力できます。

注釈: クラスターの状態を知るだけでなく、cnpg プラグインを使用して次のことを行うこともできます。構成の変更を再ロードして適用するためのループ。
 詳細については、 :ref:``kubectl` に *cnpg* プラグインを使用する<*kubectl* に *cnpg* プラグインを使用する>` のドキュメントを参照してください。

PostgreSQL コンテナー イメージのバージョンを取得します。

```
kubectl describe cluster <CLUSTER_NAME> -n <NAMESPACE> | grep "Image Name"
```

出力：

```
Image Name:    ghcr.io/cloudnative-pg/postgresql:15.2-3
```

注釈: また、*kubectl-cnpg status -n <NAMESPACE> <CLUSTER_NAME>* を使用して同じ情報を取得できます。

4.32.5 ポッド情報

特定の PostgreSQL クラスターに属するインスタンスのリストを取得するには、次のコマンドを使用します。

```
kubectl get pod -l cnpg.io/cluster=<CLUSTER> -L role -n <NAMESPACE>
```

出力：

```
NAME          READY   STATUS    RESTARTS   AGE       ROLE
<CLUSTER>-1   1/1     Running   0          10d4h5m   primary
<CLUSTER>-2   1/1     Running   0          10d4h4m   replica
<CLUSTER>-3   1/1     Running   0          10d4h4m   replica
```

以下を実行することで、ポッドが失敗しているかどうか、またどのように失敗しているかを確認できます。

```
kubectl get pod -n <NAMESPACE> -o yaml <CLUSTER>-<N>
```

次のコマンドを使用して、特定の PostgreSQL インスタンスのすべてのログを取得できます。

```
kubectl logs -n <NAMESPACE> <CLUSTER>-<N>
```

検索を PostgreSQL プロセスのみに制限する場合は、次を実行できます。

```
kubectl logs -n <NAMESPACE> <CLUSTER>-<N> | \
  jq select(.logger=="postgres") | .record.message
```

次の例では、ユーザー フレンドリな形式でタイムスタンプも追加します。

```
kubectl logs -n <NAMESPACE> <CLUSTER>-<N> | \
  jq -r select(.logger=="postgres") | [(.ts|strflocaltime("%Y-%m-%dT%H:%M:%S %Z")), .
↪record.message] | @csv
```

PostgreSQL ポッドに関する追加情報を収集してフィルタリングする

クラッシュした特定のポッドからのログを確認します。

```
kubectl logs -n <NAMESPACE> --previous <CLUSTER>-<N>
```

特定の PostgreSQL ポッドから FATAL エラーを取得します。

```
kubectl logs -n <NAMESPACE> <CLUSTER>-<N> | \
  jq -r .record | select(.error_severity == "FATAL")
```

出力：

```
{
  "log_time": "2021-11-08 14:07:44.520 UTC",
  "user_name": "streaming_replica",
  "process_id": "68",
  "connection_from": "10.244.0.10:60616",
  "session_id": "61892f30.44",
  "session_line_num": "1",
  "command_tag": "startup",
  "session_start_time": "2021-11-08 14:07:44 UTC",
  "virtual_transaction_id": "3/75",
  "transaction_id": "0",
  "error_severity": "FATAL",
  "sql_state_code": "28000",
  "message": "role \"streaming_replica\" does not exist",
  "backend_type": "walsender"
}
```

特定のポッドのログで PostgreSQL DB エラーメッセージをフィルタリングします。

```
kubectl logs -n <NAMESPACE> <CLUSTER>-<N> | jq -r .err | select(. != null)
```

出力：

```
dial unix /controller/run/.s.PGSQL.5432: connect: no such file or directory
```

特定のポッドから err 単語に一致するメッセージを取得します。

```
kubectl logs -n <NAMESPACE> <CLUSTER>-<N> | jq -r .msg | grep "err"
```

出力：

```
2021-11-08 14:07:39.610 UTC [15] LOG:  ending log output to stderr
```

特定のポッドの PostgreSQL プロセスからすべてのログを取得します。

```
kubectl logs -n <NAMESPACE> <CLUSTER>-<N> | \
  jq -r . | select(.logger == "postgres") | select(.msg != "record") | .msg
``` 出力:

```shell
2021-11-08 14:07:52.591 UTC [16] LOG:  redirecting log output to logging collector␣
↪process
2021-11-08 14:07:52.591 UTC [16] HINT:  Future log output will appear in directory "/
↪controller/log".
2021-11-08 14:07:52.591 UTC [16] LOG:  ending log output to stderr
2021-11-08 14:07:52.591 UTC [16] HINT:  Future log output will go to log destination
↪"csvlog".
```

値を含むフィールドでフィルター処理されたポッド ログを取得し、実行中の| で区切られたそれらを結合します。

```
kubectl logs -n <NAMESPACE> <CLUSTER>-<N> | \
  jq -r [.level, .ts, .logger, .msg] | join(" | ")
```

出力：

```
info | 1636380469.5728037 | wal-archive | Backup not configured, skip WAL archiving
info | 1636383566.0664876 | postgres | record
```

4.32.6 バックアップ情報

次のコマンドを使用して、名前付きクラスター用に作成されたバックアップを一覧表示できます。

```
kubectl get backup -l cnpg.io/cluster=<CLUSTER>
```

重要: バックアップのラベル付けは、CloudNativePG のバージョン 1.10.0 で導入されました。そのため、そのバージョンまたはそれ以降のバージョンで作成されたリソースのみに、そのようなラベルが含まれます。

4.32.7 保管情報

次のように、クラスターが使用する StorageClass を再確認して、調査またはトラブルシューティング中にさらにコンテキストを取得すると便利な場合があります。

```
STORAGECLASS=$(kubectl get pvc <POD> -o jsonpath={.spec.storageClassName})
kubectl get storageclasses $STORAGECLASS -o yaml
```

多くの場合、クラスターはデフォルトの StorageClass を使用して作成されるため、ここではクラスター ポッドの 1 つから StorageClass を取得しています。

4.32.8 ノード情報

Kubernetes ノードは、最終的に PostgreSQL ポッドが実行される場所です。それらについてできる限り多くのことを知ることは、戦略的に重要です。

以下を使用して、Kubernetes クラスター内のノードのリストを取得できます。

```
#  look at the worker nodes and their status

kubectl get nodes -o wide
```

さらに、特定のクラスターのポッドが実行されているノードのリストを収集できます。

```
kubectl get pod -l cnpg.io/cluster=<CLUSTER> \
  -L role -n <NAMESPACE> -o wide
```

後者は、ポッドが配布されている場所を理解するために重要です。

affinity/anti-affinity rules and/or tolerations を使用している場合は非常に便利です。

4.32.9 条件

多くのネイティブ kubernetes オブジェクト like here と同様に、Cluster は `status.conditions` も公開します。これにより、クラスター全体の正常性状態に頼るのではなく、特定のイベントが発生するのを「待つ」ことができます。現在利用可能な条件は次のとおりです。

- LastBackupSucceeded
- 継続的なアーカイブ
- 準備

`LastBackupSucceeded` は最新のバックアップのステータスを報告しています。 `True` に設定されている場合、最後のバックアップが正しく取得され、それ以外の場合は `False` に設定されます。

`ContinuousArchiving` は WAL アーカイブのステータスを報告しています。`True` に設定されている場合、最後の WAL アーカイブプロセスが正常に終了し、それ以外の場合は `False` に設定されます。

クラスターにユーザーが指定した数のインスタンスがあり、プライマリインスタンスの準備が整っている場合、`Ready` は `True` になります。この条件をスクリプトで使用して、クラスターが作成されるのを待つことができます。

特定の条件を待つ方法

- バックアップ:

```
$ kubectl wait --for=condition=LastBackupSucceeded cluster/<CLUSTER-NAME> -n
↪<NAMESPACE>
```

- 継続的なアーカイブ:

```
$ kubectl wait --for=condition=ContinuousArchiving cluster/<CLUSTER-NAME> -n
↪<NAMESPACE>
```

- 準備完了 (クラスターの準備ができているかどうか):

edb_notranlate_39 以下は、失敗した状態を含む `cluster.status` のスニペットです。

```
$ kubectl get cluster/<cluster-name> -o yaml
.
.
.
  status:
    conditions:
    - message: unexpected failure invoking barman-cloud-wal-archive: exit status
        2
      reason: ContinuousArchivingFailing
      status: "False"
      type: ContinuousArchiving

    - message: exit status 2
      reason: LastBackupFailed
      status: "False"
      type: LastBackupSucceeded

    - message: Cluster Is Not Ready
      reason: ClusterIsNotReady
      status: "False"
      type: Ready
```

4.32.10 よくある問題

ストレージがいっぱいです

クラスター内の 1 つ以上のポッドが CrashloopBackoff にあり、これがディスクがいっぱいであることが原因である可能性があることをログが示唆している場合、おそらくインスタンスの PersistentVolumeClaim のサイズを増やす必要があります。ドキュメントの

> ボリューム拡張 を見てください。

Pod が Pending 状態でスタックする

クラスターのインスタンスが Pending フェーズでスタックしている場合は、ポッドの Events セクションをチェックして、この背後にある理由を把握する必要があります。

```
kubectl describe pod -n <NAMESPACE> <POD>
```

これには、次のような原因が考えられます。

- nodeSelector に一致するノードがありません
- 容認がノードのテイントに一致するように正しく構成されていない

- ノードがまったく利用できない: これは、cluster-autoscaler が制限に達したり、一時的な問題が発生したりすることにも関連している可能性があります

この場合、名前空間内のイベントを確認することも役立ちます。

```
kubectl get events -n <NAMESPACE>
#  list events in chronological order

kubectl get events -n <NAMESPACE> --sort-by=.metadata.creationTimestamp
```

バックアップが構成されていない場合、レプリカは同期されません

メンテナンス上の理由により、レプリカがしばらくオフになることがあります (Kubernetes ノードがドレインされたときを考えてください)。クラスターにバックアップが構成されていない場合、レプリカが復旧したときに、プライマリにもう存在しない WAL ファイルが必要になる可能性があります (:ref:``postgresql`セクション<*postgresql* セクション>`` で説明されているように、WAL 管理ポリシーに従って既にリサイクルされています)。同期の。

同様に、pg_rewind が以前のプライマリに存在しなくなった WAL ファイルを必要とする可能性がある場合、pg_rewind: error: could not open file を報告します。

このような場合、Pod の準備ができなくなり、PVC を削除してオペレーターにレプリカの再構築を依頼する必要があります。

動的にプロビジョニングされた永続ボリュームに依存していて、PV 自体を削除する自信がある場合は、次の方法で削除できます。

```
PODNAME=<POD>
VOLNAME=$(kubectl get pv -o json | \
  jq -r .items[]|select(.spec.claimRef.name==\"$PODNAME\")|.metadata.name)

kubectl delete pod/$PODNAME pvc/$PODNAME pvc/$PODNAME-wal pv/$VOLNAME
```

4.33 フェンシング

CloudNativePG のフェンシングは、PostgreSQL クラスターの 1 つ、複数、またはすべてのインスタンスのデータが誤動作しているように見える場合に、そのデータを保護する究極のプロセスです。インスタンスがフェンシングされると、ポッドが実行され続けている間、PostgreSQL サーバー プロセス (postmaster) がシャットダウンされることが保証されます。これにより、フェンスが解除されるまで、Pod 上のデータが PostgreSQL によって変更されず、デバッグやトラブルシューティングの目的でファイルシステムを調査できることが保証されます。

4.33.1 インスタンスをフェンシングする方法

CloudNativePG では、以下をフェンシングできます。

- 特定のインスタンス
- インスタンスのリスト
- Postgres Cluster 全体

フェンシングは、cnpg.io/fencedInstances アノテーションの内容によって制御されます。これは、インスタンス名の JSON 形式のリストを想定しています。注釈が '["*"]' (ワイルドカードを含むシングルトン リスト) に設定されている場合、クラスター全体がフェンシングされます。アノテーションが空の JSON リストに設定されている場合、オペレーターはアノテーションが設定されていないかのように動作します。

例えば：

- cnpg.io/fencedInstances: '["cluster-example-1"]' は cluster-example-1 インスタンスのみをフェンスします
- cnpg.io/fencedInstances: '["cluster-example-1","cluster-example-2"]' は cluster-example-1 および cluster-example-2 インスタンスをフェンスします
- cnpg.io/fencedInstances: '["*"]' は、クラスター内のすべてのインスタンスをフェンシングします。

注釈は、たとえば kubectl annotate コマンドを使用して、または kubectl cnpg fencing on サブコマンドを使用して透過的な方法で、Kubernetes オブジェクトに手動で設定できます。

```
#  to fence only one instance

kubectl cnpg fencing on cluster-example 1

#  to fence all the instances in a Cluster

kubectl cnpg fencing on cluster-example "*"
```

以前にフェンシングされたインスタンスを持つ Cluster の例を次に示します。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
    annotations:
      cnpg.io/fencedInstances: ["cluster-example-1"]
[...]
```

4.33.2 フェンシングの持ち上げ方

フェンシングは、注釈をクリアするか、別の値に設定することで解除できます。

フェンシングに関しては、これは kubectl annotate を使用して手動で行うか、次のように kubectl cnpg fencing サブコマンドを使用して行うことができます。

```
#  to lift the fencing only for one instance

#  N.B.: at the moment this wont work if the whole cluster was fenced previously,

#        in that case you will have to manually set the annotation as explained above

kubectl cnpg fencing off cluster-example 1

#  to lift the fencing for all the instances in a Cluster

kubectl cnpg fencing off cluster-example "*"
```

4.33.3 フェンシングの仕組み

インスタンスがフェンシング用に設定されると、postmaster プロセスをシャットダウンする手順が開始されます。これは、タイムアウトが.spec.stopDelay に設定された最初のスマートシャットダウンと、その後の必要に応じた高速シャットダウンで構成されます。それから：

- Pod は存続します
- Pod は *Ready* としてマークされません
- Postgres インスタンスの起動を必要としないすべての変更が調整されます。これには以下が含まれます。- 構成ファイル - 証明書とすべての暗号化マテリアル
- 1 に設定される cnpg_collector_fencing_on を除いて、メトリクスは収集されません

警告: **プライマリ インスタンス**がフェンスされている場合、そのポストマスター プロセスはシャットダウンされますが、フェイルオーバーは実行されず、アプリケーションの操作性が中断されます。フェンスが解除されると、フェイルオーバーを実行せずにプライマリ インスタンスが再起動されます。

そのため、厳密に必要な場合にのみプライマリインスタンスをフェンシングすることをお勧めします。

フェンシングされたインスタンスが削除された場合、ポッドは正常に再作成されますが、postmaster は開始されません。これは、インスタンスが Crashlooping の場合に非常に役立ちます。

4.34 PostGIS

PostGIS は PostgreSQL の非常に人気のあるオープン

ソース拡張機能であり、データベースに GIS (地理情報システム) オブジェクトを格納し、SQL 経由でクエリを実行するためのサポートを導入します。

重要: このセクションでは、PostGIS に精通していることを前提として、CloudNativePG を介して Kubernetes で PostGIS データベースを使用して新しい PostgreSQL クラスターを作成する方法に関する基本的な情報を提供します。

CloudNativePG コミュニティは、公式の PostGIS images hosted on DockerHub の上に構築されたコンテナーイメージを維持しています。詳細については、以下をご覧ください。

- postgis-containers
- postgis-containers

4.34.1 PostGIS クラスターに関する基本概念

概念的には、PostGIS ベースの PostgreSQL クラスター (または単に PostGIS クラスター) は、他の PostgreSQL クラスターと似ています。唯一の違いは次のとおりです。

- PostGIS および関連ライブラリのシステム内での存在
- データベース内の PostGIS エクステンションの存在

CloudNativePG は Immutable Application Containers に基づいているため、PostGIS をプロビジョニングする唯一の方法は、オペランドに使用するコンテナーイメージに追加することです。 **コンテナ イメージの要件**は、これがどのように達成されるかについての詳細な手順を提供します。もっと簡単に言うと、以下の例のように、コミュニティから PostGIS コンテナー イメージを使用するだけです。

2 番目のステップは、拡張機能を PostgreSQL データベースにインストールすることです。これは、次の 2 つの方法で行うことができます。

- アプリケーションデータベースにインストールします。これは、マイクロサービスアーキテクチャに従ってクラスターでホストするメインでおそらく唯一のデータベースです。または
- `template1` データベースにインストールして、インスタンスが複数のデータベースで共有されるモノリスアーキテクチャを採用する場合に、クラスター内に作成することになるすべてのデータベースで使用できるようにします。

または Database import 。

4.34.2 PostGIS で新しい PostgreSQL クラスターを作成する

PostGIS 3.2 で新しい PostgreSQL 14 クラスターを作成するとします。

最初のステップは、オペランドに正しい PostGIS コンテナーイメージを使用し、`Cluster` リソースで `.spec.imageName` オプションを適切に設定することです。

以下の postgis-example.yaml は、PostGIS クラスターの作成方法に関するガイダンスを提供します。

> **警告:** CloudNativePG では設定よりも規則が適用されますが、本番用にシステムの設定と調整に時間を費やす必要があることを考慮してください。また、以下の例の *imageName* は、意図的に PostgreSQL 14 の最新の利用可能なイメージを指しています。真の不変性のために、特定のイメージ名を使用するか、できれば SHA256 ダイジェストを使用する必要があります。

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgis-example
spec:
  instances: 3
  imageName: ghcr.io/cloudnative-pg/postgis:14
  bootstrap:
    initdb:
      postInitTemplateSQL:
        - CREATE EXTENSION postgis;
        - CREATE EXTENSION postgis_topology;
        - CREATE EXTENSION fuzzystrmatch;
        - CREATE EXTENSION postgis_tiger_geocoder;

  storage:
    size: 1Gi
```

この例は、アプリケーション データベース (app と呼ばれる) を実際に作成する前に、`template1` データベースに対してクエリのリストを実行する `postInitTemplateSQL` オプションに依存しています。つまり、マニフェストを適用してクラスターが起動すると、上記の拡張機能がテンプレートデータベースとアプリケーションデータベースの両方にインストールされ、使用できるようになります。

app データベースに接続することで、コンテナー内にある利用可能な PostGIS のバージョンを簡単に確認できます (このドキュメントの値とは異なる値を取得する場合があります)。

```
$ kubectl exec -ti postgis-example-1 -- psql app
Defaulted container "postgres" out of: postgres, bootstrap-controller (init)
psql (15.2 (Debian 15.2-1.pgdg110+1))
Type "help" for help.

app=# SELECT * FROM pg_available_extensions WHERE name ~ ^postgis ORDER BY 1;
```

(次のページに続く)

(前のページからの続き)

```
           name           | default_version | installed_version |
     comment
- ------------------------+-----------------+-------------------+--------------------
↪---------------------------------------
 postgis                  | 3.2.2           | 3.2.2             | PostGIS geometry␣
↪and geography spatial types and functions
 postgis-3                | 3.2.2           |                   | PostGIS geometry␣
↪and geography spatial types and functions
 postgis_raster           | 3.2.2           |                   | PostGIS raster␣
↪types and functions
 postgis_raster-3         | 3.2.2           |                   | PostGIS raster␣
↪types and functions
 postgis_sfcgal           | 3.2.2           |                   | PostGIS SFCGAL␣
↪functions
 postgis_sfcgal-3         | 3.2.2           |                   | PostGIS SFCGAL␣
↪functions
 postgis_tiger_geocoder   | 3.2.2           | 3.2.2             | PostGIS tiger␣
↪geocoder and reverse geocoder
 postgis_tiger_geocoder-3 | 3.2.2           |                   | PostGIS tiger␣
↪geocoder and reverse geocoder
 postgis_topology         | 3.2.2           | 3.2.2             | PostGIS topology␣
↪spatial types and functions
 postgis_topology-3       | 3.2.2           |                   | PostGIS topology␣
↪spatial types and functions
(10 rows)
```

次のステップは、postInitTemplateSQL セクションにリストされている拡張機能が app データベースに正しくインストールされていることを確認することです。

```
app=# \dx
                                    List of installed extensions
          Name          | Version |   Schema   |                          Description
- ----------------------+---------+------------+--------------------------------------
↪-----------------------
 fuzzystrmatch          | 1.1     | public     | determine similarities and distance␣
↪between strings
 plpgsql                | 1.0     | pg_catalog | PL/pgSQL procedural language
 postgis                | 3.2.2   | public     | PostGIS geometry and geography␣
↪spatial types and functions
 postgis_tiger_geocoder | 3.2.2   | tiger      | PostGIS tiger geocoder and reverse␣
↪geocoder
 postgis_topology       | 3.2.2   | topology   | PostGIS topology spatial types and␣
↪functions
(5 rows)
```

ついに：

```
app=# SELECT postgis_full_version();
                                                                                postgis_
↪full_version
- ------------------------------------------------------------------------------------------
↪------------------------------------------------------------------------------------------
↪---
 POSTGIS="3.2.2 628da50" [EXTENSION] PGSQL="140" GEOS="3.9.0-CAPI-1.16.2" PROJ="7.2.1"
 LIBXML="2.9.10" LIBJSON="0.15" LIBPROTOBUF="1.3.3" WAGYU="0.5.0 (Internal)" TOPOLOGY
(1 row)
```

4.35 エンドツーエンドのテスト

CloudNativePG は、一連の **エンドツーエンド (E2E) テスト**を介して各コミット後に自動的にテストされます。これにより、オペレーターは PostgreSQL クラスターを正しくデプロイおよび管理できます。

さらに、次の Kubernetes バージョンはコミットごとにテストされ、開発プロセスの初期段階でバグを検出するのに役立ちます。

- 1.26
- 1.25
- 1.24
- 1.23
- 1.22

次の PostgreSQL バージョンがテストされています。

- PostgreSQL 15
- PostgreSQL 14
- PostgreSQL 13
- PostgreSQL 12
- PostgreSQL 11

Kubernetes と PostgreSQL のテスト済みバージョンごとに、 親切を使用して Kubernetes クラスターが作成され、そのクラスターで次の一連の E2E テストが実行されます。

- **基本:** * オペレーターのインストール。 * クラスターの作成; * データ ストレージ用の永続ボリュームの使用。
- **サービス接続:** * 読み取り専用を含むサービス経由の接続。 * ユーザー提供のサーバーおよび/またはクライアント証明書による接続。 * PgBouncer;
- **自己修復:** * フェールオーバー。 * 切り替える; * 10 秒以内にフェールオーバーが発生した場合のプライマリ エンドポイントスイッチ。 * 20 秒未満での切り替えの場合のプライマリエンドポイントの切り替え。 * 60 秒以内に劣化状態から回復します。 * PVC 削除;

- **バックアップと復元:** * S3 で Barman Cloud を使用したバックアップとスケジュールされたバックアップの実行。 * Barman Cloud on Azure ブロブストレージを使用したバックアップとスケジュール バックアップの実行。* S3 で Barman Cloud を使用してバックアップから復元します。 * Barman Cloud on Azure ブロブストレージを使用してバックアップから復元します。 * ウォール復元;
- **オペレーター:** * オペレーターの展開。 * ConfigMap によるオペレーター設定。 * オペレーター ポッドの削除。 * オペレーターポッドの立ち退き; * オペレーターのアップグレード; *オペレーターの高可用性。
- **可観測性:** * メトリック コレクション。 * PgBouncer メトリクス; * JSON ログ形式;
- **レプリケーション:** * 物理レプリカ クラスター。 * 複製スロット; * 同期複製; * クラスターのスケールアップとスケールダウン。
- **プラグイン:** * CNPG プラグインを使用したクラスター休止状態。**フェンシング**; 接続証明書の作成;
- **Postgres 構成:** * PostgreSQL 構成の変更を管理します。 * PostgreSQL イメージを変更する際のローリング更新。
- **Pod スケジューリング:** * 耐性と汚染。 * `NodeSelector` を使用したポッド アフィニティ。
- **クラスターメタデータ:** * クラスター ラベルと注釈の ConfigMap。 * オブジェクトのメタデータ。
- **回復:** * データの破損。 * pg_basebackup;
- **データベースのインポート:** * マイクロサービス アプローチ。*モノリスアプローチ;
- **ストレージ:** * ストレージの拡張。
- **セキュリティ:** * AppArmor アノテーションの伝播。 Azure 環境でのみ実行されます。
- **メンテナンス:** * ノードドレイン;

4.36 コンテナ イメージの要件

Kubernetes の CloudNativePG オペレーターは、次の要件に準拠する PostgreSQL の互換性のあるコンテナーイメージと連携するように設計されています。

- パスにある必要がある PostgreSQL 実行可能ファイル: - `initdb` - `postgres` - `pg_ctl` - `pg_controldata` - `pg_basebackup`
- パスにある必要がある Barman Cloud 実行可能ファイル: - `barman-cloud-backup` - `barman-cloud-backup-delete` - `barman-cloud-backup-list` - `barman-cloud-check-wal-archive` - `barman-cloud-restore` - `barman-cloud-wal-archive` - `barman-cloud-wal-restore`
- インストールされた PGAudit 拡張機能 (オプション - デプロイされたクラスターで PGAudit が必要な場合のみ)
- 適切なロケール設定

重要: PostgreSQL versions supported by the PGDG のみが許可されます。

CloudNativePG がインスタンスマネージャーで上書きするため、イメージ定義にエントリポイントやコマンドは必要ありません。

> **警告:** アプリケーション コンテナー イメージは、**複数/オプションのホット スタンバイ サーバー アーキテクチャを備えたプライマリ** で CloudNativePG によって使用されます。

CloudNativePG コミュニティは public PostgreSQL container images を提供およびサポートしています

CloudNativePG で動作し、 ghcr.io で公開しています。

4.36.1 画像タグの要件

イメージ名は Docker で有効なものであれば何でもかまいませんが、CloudNativePG オペレーターは *image タグ* に依存して、イメージに含まれる Postgres メジャーバージョンを検出します。

イメージ タグは、有効な PostgreSQL メジャー バージョン番号 (14.5 または 15 など) で始まり、必要に応じてドットとパッチ レベルが続きます。

この後に、ドット、アンダースコア、またはマイナス記号を前に付けて、Docker タグで有効かつ受け入れられる任意の文字の組み合わせを続けることができます。

使用できるイメージ タグの例:

- `11.1`
- `12.3.2.1-1`
- `12.4`
- `13`
- `14.5-10`
- `15.0`

> **警告:** *latest* は、画像の有効なタグとは見なされません。

4.37 オペレーターの能力レベル

このセクションでは、 Operator SDK definition of Capability Levels を使用して分類された、CloudNativePG によって実装された機能の概要を提供します。

フレームワーク。

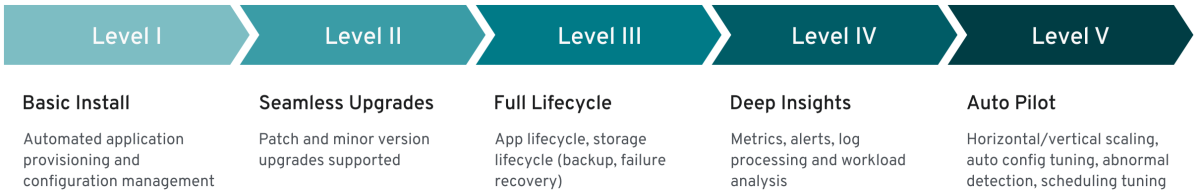


図 17: Operator Capability Levels

重要: Operator Capability Levels model に基づいて、CloudNativePG Operator から **"Level V - Auto Pilot"** 一連の機能を期待できます。

各機能レベルは、オペレーターが提供する特定の一連の管理機能に関連付けられています。

1. 基本インストール

2. シームレスなアップグレード

3. 完全なライフサイクル

4. 深い洞察

5. オートパイロット

注釈: このフレームワークは、オペレーターの将来の作業と実装のガイドと見なされます。

4.37.1 レベル 1 - 基本インストール

能力レベル 1 には、オペレーターの **インストール** と **構成**が含まれます。このカテゴリには、オペレーターとの対話方法の改善や PostgreSQL クラスター構成など、使いやすさとユーザーエクスペリエンスの強化が含まれます。

重要: **情報セキュリティ**はこのレベルの一部と考えています。

宣言型構成によるオペレーターのデプロイ

オペレーターは、4 つの主要な `CustomResourceDefinition` オブジェクト (`Cluster` 、`Pooler` 、`Backup` 、および `ScheduledBackup`) を定義する Kubernetes マニフェストを使用して、宣言的な方法でインストールされます。

宣言型構成による PostgreSQL クラスターの展開

PostgreSQL クラスター (オペランド) は、完全に宣言的な方法で `Cluster` カスタム リソースを使用して定義されます。 PostgreSQL のバージョンは、CR で定義されたオペランド コンテナーイメージによって決定されます。これは、要求されたレジストリから自動的にフェッチされます。オペランドをデプロイするとき、オペレーターは次のリソースも自動的に作成します: `Pod` 、`Service` 、`Secret` 、`ConfigMap` 、`PersistentVolumeClaim` 、`PodDisruptionBudget` 、`ServiceAccount` 、`RoleBinding` 、`Role` 。

CRD によるオペランド イメージのオーバーライド

オペレーターは、PostgreSQL を内部に持つオペランド コンテナーイメージをサポートするように設計されています。デフォルトでは、オペレーターは、PostgreSQL コミュニティーによってサポートされ、ghcr.io で公開されている最新の安定したメジャーバージョンの最新の利用可能なマイナー バージョンを使用します。 CR で `imageName` 属性を設定することにより、プライマリ/スタンバイアーキテクチャを直接サポートする PostgreSQL の互換性のあるイメージを使用できます。オペレーターは、プライベートコンテナー レジストリにアクセスするための `imagePullSecrets` もサポートしています。また、コンテナーイメージの不変性をより細かく制御するためのタグに加えて、ダイジェストもサポートしています。

ラベルと注釈

Operator は、クラスターのメタデータで定義されているラベルとアノテーションの継承をサポートするように構成できます。これは、Kubernetes インフラストラクチャーでの CloudNativePG デプロイメントの組織を改善することを目的としています。

自己完結型のインスタンス マネージャー

Patroni や Stolon などの外部ツールに依存して Kubernetes クラスターポッド内の PostgreSQL インスタンスを調整する代わりに、オペレーターは各ポッド内の`/controller/manager` という名前のファイルにオペレーター実行可能ファイルを挿入します。アプリケーションは、基盤となる PostgreSQL インスタンスを制御し、PostgreSQL クラスタートポロジに基づいてインスタンス自体とポッドの状態を調整するために使用されます。インスタンスマネージャーは、プローブ用に `kubelet` によって呼び出される Web サーバーも開始します。 `kubelet` によって呼び出された Unix シグナルは、インスタンスマネージャーによってフィルター処理され、必要に応じて `postgres` プロセスに転送され、外部イベントに対する迅速かつ制御された反応が行われます。インスタンスマネージャーは Go で記述されており、外部依存関係はありません。

ストレージ構成

ストレージは、データベースワークロードの重要なコンポーネントです。ストレージに関して Kubernetes ネイティブの機能とリソースを利用するオペレーターは、基盤となる Kubernetes 環境が提供できるものに基づいて、ワークロード要件に適したストレージを選択するのに十分な柔軟性を提供します。これは、パブリッククラウド環境で特定のストレージ クラスを選択するか、CR の `storage` パラメーターの PVC テンプレートを使用して生成された PVC を微調整することを意味します。パフォーマンスを向上させ、より細かく制御するために、クラスターの先行書き込みログ (WAL、`pg_wal` とも呼ばれる) を別のボリューム (できれば別のストレージ) でホストすることもできます。 cnp-bench オープン ソースプロジェクトを使用して、運用前にストレージとデータベースの両方をベンチマークできます。

レプリカ構成

オペレーターは、`instances` という単一のパラメーターを使用して、クラスター内のレプリカを自動的に検出します。1 に設定すると、クラスターはレプリカのない単一のプライマリ PostgreSQL インスタンスで構成されます。 1 より高い場合、オペレーターは、自動フェイルオーバーによる高可用性 (HA) と切り替え操作によるローリング更新を含む、`instances -1` レプリカを管理します。 CloudNativePG は、以前に提案された Failover slots と呼ばれる PostgreSQL 用のパッチに触発された実装を使用して、HA クラスター内のすべてのレプリカのレプリケーションスロットを自動的に管理します。

データベース構成

このオペレーターは、単一のデータベースで PostgreSQL クラスターを管理するように設計されています。オペレーターは、読み取り/書き込み、読み取り、および読み取り専用のワークロード用に自動的にプロビジョニングおよび管理される 3 つの Kubernetes サービスを通じて、データベースへのアクセスを透過的に管理します。構成よりも規則的なアプローチを使用して、オペレーターは `app` という名前のデータベースを作成します。これはデフォルトで、同じ名前の通常の Postgres ユーザーによって所有されます。必要に応じて、データベース名とユーザー名の両方を指定できます。クラスターを実行するために構成は必要ありませんが、CR の `postgresql` セクションで PostgreSQL ランタイム構成と PostgreSQL ホストベース認証ルールの両方をカスタマイズできます。

ポッド セキュリティ ポリシー

InfoSec 要件の場合、オペレーターはコンテナーに対して特権モードを必要とせず、読み取り専用のルートファイルシステムを適用して、オペレーターとオペランドポッドの両方のコンテナーの不変性を保証します。また、必要なセキュリティコンテキストを明示的に設定します。

親和性

クラスターの affinity セクションを使用すると、Kubernetes クラスターのノード全体で、ポッドおよび永続ボリュームなどの関連リソースをスケジュールする方法を微調整できます。特に、オペレーターは以下をサポートします。

- ポッドの親和性と非親和性
- ノードセレクター
- 汚染と寛容

コマンドラインインターフェース

CloudNativePG には、独自のコマンド ライン インターフェイスはありません。PostgreSQL クラスター管理エクスペリエンスを強化および簡素化する cnpg というプラグインを提供することで、Kubernetes に最適なコマンド ラインインターフェースである kubectl に依存するだけです。

クラスタの現在のステータス

オペレーターは、クラスターの観察された状況で CR の状況セクションを継続的に更新します。 PostgreSQL クラスター全体のステータスは、各ポッドで実行されているインスタンスマネージャーによって継続的に監視されます。インスタンスマネージャーは、制御された PostgreSQL インスタンスに必要な変更を適用して、クラスターの必要なステータスに収束する責任があります (たとえば、クラスター ステータスがレポートする場合)。ポッド -1 がプライマリであり、ポッド -1 はそれ自体を昇格させる必要があり、他のポッドはポッド -1 に従う必要があります)。詳細を提供するために、kubectl の cnpg プラグインによって同じステータスが使用されます。

事業者の認証局

オペレーターは、自身の認証局を自動的に作成します。 Kubernetes API サーバーとオペレーター自体の間の安全な通信を確保するために、webhook サーバーで使用されるリーフ証明書を作成し、オペレーター認証局で署名します。

クラスターの認証局

オペレーターは、すべての PostgreSQL クラスターに対して認証局を自動的に作成します。これは、(パスワードの代わりに) ストリーミングレプリケーションスタンバイサーバーを含む、クライアントの認証用の TLS 証明書を発行および更新するために使用されます。クライアント証明書のカスタム証明機関のサポートは、シークレットを通じて利用できます。これには、cert-manager との統合も含まれます。証明書は、kubectl の cnpg プラグインで発行できます。

TLS 接続

このオペレーターは、TLS/SSL 接続を透過的かつネイティブにサポートし、クライアント/サーバー通信を暗号化して、クラスターの認証機関を使用してセキュリティを強化します。カスタムサーバー証明書のサポートは、シークレットを通じて利用できます。これには、cert-manager との統合も含まれます。

ストリーミングレプリケーションの証明書認証

オペレーターは、パスワード (したがってシークレット) に依存する代わりに、TLS クライアント証明書認証に依存して、スタンバイサーバーからのストリーミングレプリケーション接続を承認します。

継続的な構成管理

オペレーターを使用すると、PostgreSQL 構成の `Cluster` リソース YAML セクションに変更を適用でき、構成オプションに応じて、すべてのインスタンスが適切に再ロードまたは再起動されるようになります。**現在の制限**: `ALTER SYSTEM` の変更は検出されません。つまり、クラスターの状態は適用されません。

既存の PostgreSQL データベースのインポート

バージョン 1.16 以降、オペレーターはオフライン移行を使用して、既存の Postgres データベースを Kubernetes の新しい CloudNativePG `Cluster` にインポートする宣言的な方法を提供します。同じ機能は、PostgreSQL データベースのオフライン メジャーアップグレードにも対応しています。オフラインとは、データベースがインポートされるまで、アプリケーションがソースで書き込み操作を停止する必要があることを意味します。この機能は、`initdb` ブートストラップ メソッドを拡張して、別の PostgreSQL データベースで使用可能なデータの論理スナップショットを使用して新しい PostgreSQL クラスターを作成します。これは、スーパーユーザー接続を介してネットワーク経由でアクセスできます。インポートは、サポートされているすべてのバージョンの Postgres から行われ、`pg_dump` および `pg_restore` に依存して、操作のすべてのデータベース部分、および要求された場合はロールに対して、新しいクラスタープライマリから実行されます。

PostGIS クラスタ

CloudNativePG は、 *PostGIS* を使用したクラスターのインストールをサポートしています

PostgreSQL の最も一般的な拡張機能の 1 つである、地理データベース用のオープン ソース拡張機能。

PostgreSQL の基本的な LDAP 認証

オペレーターを使用すると、 PostgreSQL documentation: LDAP authentication で説明されているように、*simple bind* または *search+bind* モードを使用して、PostgreSQL クライアントの LDAP 認証を構成できます。

複数のインストール方法

このオペレーターは、`kubectl apply` を介して Kubernetes マニフェストを介してインストールでき、パブリックおよびプライベートクラウド環境の従来の Kubernetes インストールで使用できます。さらに、オペレーター用の Helm チャートも利用できます。

設定より規約

オペレーターは、構成パラダイムよりも規則をサポートし、標準のデフォルト値を決定しながら、それらをオーバーライドしてカスタマイズできるようにします。いくつかの YAML コード行で `Cluster` CRD を使用して、PostgreSQL クラスターのデプロイを指定できます。

4.37.2 レベル 2 - シームレスなアップグレード

機能レベル 2 は、 **オペレーターと実際のワークロードの更新** (この場合は PostgreSQL サーバー) を有効にすることです。これには、**PostgreSQL のマイナーリリース更新** (通常はセキュリティとバグ修正) と**主要なオンラインアップグレード** が含まれます。

オペレーターのアップグレード

新しい展開としてオペレーターをシームレスにアップグレードできます。演算子を変更しても、オペランドを変更する必要はありません- インスタンスマネージャーのインジェクションのおかげです。オペレーターは、古いバージョンのオペランドを管理できます。

CloudNativePG は インスタンスマネージャーのインプレース更新 もサポートしています

オペレーターのアップグレード後: インプレース更新では、クラスターのローリング更新 (およびその後の切り替え) は必要ありません。

管理対象ワークロードのアップグレード

オペランドは、CR、特に `imageName` パラメーターの変更の一環として、宣言的な構成アプローチを使用してアップグレードできます。オペレーターは、PostgreSQL のメジャー アップグレードを防ぎながら、メジャー バージョン内のマイナー PostgreSQL リリースに関して両方向に進むことを可能にします (更新とロールバックを有効にします)。

スタンバイサーバーが存在する場合、オペレーターは、既存のポッドを削除し、基になるストレージを再利用する新しい要求されたオペランドイメージを使用して新しいポッドを作成することにより、レプリカ

から開始するローリング更新を実行します。primaryUpdateStrategy の値に応じて、オペレーターは以前のプライマリ (unsupervised) を更新する前に切り替えを続行するか、ユーザーが kubectl の cnpg プラグインを介して切り替え手順 (supervised) を手動で発行するのを待ちます。どの設定を使用するかは、実際のデータベースワークロードに基づいて、操作によってアプリケーションのダウンタイムが数秒から数分発生する可能性があるため、ビジネス要件によって異なります。

アップグレード中にクラスターの可用性ステータスを表示する

いつでも、Setting up primary、Creating a new replica、Cluster in healthy state、Switchover in progress、Failing over、Upgrading cluster など、クラスターの高可用性ステータスを伝えます。

4.37.3 レベル 3 - フル ライフサイクル

能力レベル 3 では、事業者は **ビジネス継続性** と **スケーラビリティ** の側面を管理する必要があります。 **ディザスタリカバリ**は、データベースのバックアップとリカバリの両方が正しく機能することを必要とするビジネス継続性のコンポーネントです。開始点としての目標は RPO を 5 分未満にすることですが、長期的な目標は RPO=0 のバックアップソリューションを実装することです。 **高可用性** は、ビジネス継続性のもう 1 つの重要な要素であり、PostgreSQL ネイティブの物理レプリケーションとホット スタンバイレプリカにより、オペレーターはフェイルオーバーとスイッチオーバー操作を実行できます。この領域には、次の機能強化が含まれています。

- 同期レプリケーション、(カスケード) レプリケーション クラスターなどの PostgreSQL 物理レプリケーションの制御。
- 接続プーリング。pgBouncer を使用した接続プーリングレイヤーを介してパフォーマンスと制御を向上させます。

PostgreSQL ホット バックアップ

このオペレーターは、物理ベースのバックアップと継続的な WAL アーカイブに基づく PostgreSQL のネイティブの継続的なホット バックアップテクノロジを使用して、アプリケーションレベルのバックアップを提供するように設計されています。具体的には、オペレーターは現在、オブジェクトストア (AWS S3 および S3 互換、Azure Blob Storage、Google Cloud Storage、および MinIO などのゲートウェイ) のバックアップのみをサポートしています。

WAL アーカイブとベース バックアップは、S3 プロトコルの宛先 URL (たとえば、AWS S3 バケット内の特定のフォルダーを指すように) を指定することによって、クラスター定義の backup パラメーターを使用して、宣言的にクラスターレベルで定義されます。汎用エンドポイント URL。継続的なバックアップの前提条件である WAL アーカイブは、ユーザーによる追加のアクションを必要としません。オペレーターは、定義されたエンドポイントに WAL ファイルを送信するために barman-cloud-wal-archive に依存するように archive_command を自動的かつ透過的に設定します。ユーザーは、圧縮アルゴリズムと、アーカイブ内の WAL ファイルを同時にアップロードする並列ジョブの数を決定できます。それに加えて、Instance Manager は、WAL ファイルの最初のセットの出荷を開始する前に barman-cloud-check-wal-archive コマンドを実行することにより、アーカイブ先の正確性を自動的にチェックします。

基本バックアップは、オンデマンド (Backup カスタムリソース定義による) またはスケジュール (ScheduledBackup 顧客リソース定義による、cron のような構文を使用) の 2 つの方法で定義できます。どちらも、ジョブ (アプリケーション コンテナーイメージの一部として配布される) を barman-cloud-backup に依存して、WAL ファイルと一緒に同じエンドポイントでバックアップをリレーします。

barman-cloud-wal-restore と barman-cloud-backup の両方が、GNU GPL 3 条件の下でアプリケーション コンテナー イメージで配布されます。

スタンバイからのバックアップ

オペレーターは、データベースの RPO に影響を与えることなく、スタンバイへのベースバックアップのオフロードをサポートします。これにより、標準的なデータベース操作のために、プライマリのリソース、特に I/O を保持できます。

バックアップからの完全復元

オペレーターを使用すると、barman-cloud-backup を使用して作成された既存のアクセス可能なバックアップから開始して、新しいクラスターを (その設定と共に) ブートストラップできます。ブートストラッププロセスが完了すると、オペレータはインスタンスをリカバリモードで開始し、指定されたアーカイブから利用可能なすべての WAL ファイルを再生し、リカバリを終了してプライマリとして開始します。その後、オペレーターは要求された数のスタンバイインスタンスをプライマリから複製します。 CloudNativePG は、アーカイブからの並列 WAL フェッチをサポートしています。

バックアップからのポイントインタイム リカバリ (PITR)

Operator を使用すると、タイムスタンプ、ラベル、またはトランザクション ID で定義された特定の時点まで既存のバックアップを復元することで、新しい PostgreSQL クラスターを作成できます。この機能は完全復元機能の上に構築され、

> PostgreSQL for PITR で利用可能なすべてのオプションをサポートします。

同期レプリケーションによるデータ損失ゼロのクラスター

クォーラム ベースの同期レプリケーションサポートにより、ローカルの高可用性 CloudNativePG クラスターで **ゼロデータ損失** (RPO=0) を達成します。オペレーターは、いつでも使用できると予想される同期スタンバイレプリカの最小数と最大数を制御する 2 つの構成オプションを提供します。オペレーターは、クォーラム (q) の次の式を通じて、クラスター内の使用可能で準備ができている PostgreSQL インスタンスの数に基づいて、それに応じて反応します。

```
1 <= minSyncReplicas <= q <= maxSyncReplicas <= readyReplicas
```

レプリカ クラスタ

PostgreSQL ネイティブ ストリーミングとカスケードレプリケーションを利用して、PostgreSQL クラスターのクロス Kubernetes クラスター トポロジを定義します。 `replica` オプションを使用すると、独立したクラスターをセットアップして、同じメジャーバージョンの別の PostgreSQL ソースから継続的にデータをレプリケートできます。このようなソースは、2 つのエンドポイントから TLS を介した直接ストリーミング接続が許可されている限り、どこにでも配置できます。さらに、ソースは、物理環境または仮想環境で実行されている Kubernetes の外部であってもかまいません。レプリカ クラスタは、リカバリオブジェクトストア (Barman Cloud 形式のバックアップ) から、または `pg_basebackup` を介したストリーミングを介して作成できます。 WAL ファイル シッピングと WAL ストリーミングの両方が許可されます。レプリカクラスターは、複数のデータセンターにまたがり、ハイブリッドおよびマルチクラウドのセットアップに対応できるようにすることで、Kubernetes 内の PostgreSQL データベースのビジネス継続性を劇的に改善します (現在、Kubernetes フェデレーションのネイティブ機能を待つ間、データセンター間の手動スイッチオーバーが必要です)。

liveness と readiness プローブ

オペレーターは、kubelet によって呼び出される Postgres コンテナーの liveness および readiness プローブを定義します。これらは、インスタンスマネージャーによって直接管理される Web サーバーの `/healthz` および`/readyz` エンドポイントにそれぞれマップされます。 liveness プローブは `pg_isready` 実行可能ファイルに基づいており、ポッドは終了コード 0 (サーバーが正常に接続を受け入れている) および 1 (サーバーが起動中などに接続を拒否している) で正常であると見なされます。readiness プローブは、単純なクエリ (`;`) を発行して、サーバーが接続を受け入れる準備ができていることを確認します。

ローリング展開

オペレーターは、ダウンタイムを最小限に抑えるためにローリングデプロイをサポートします。PostgreSQL クラスターが公開されている場合、サービスは、初期化または更新中に、読み取り専用トラフィックを使用可能なポッドにのみ負荷分散します。

レプリカのスケールアップとスケールダウン

オペレーターを使用すると、PostgreSQL クラスター内のインスタンス数をスケールアップおよびスケールダウンできます。新しいレプリカはプライマリサーバーから自動的に起動され、クラスターの HA インフラストラクチャに参加します。 CRD は、ユーザーが `kubectl scale` コマンドを使用できるようにする「スケール」サブリソースを宣言します。

Kubernetes ノードのメンテナンス期間と PodDisruptionBudget

オペレーターは PodDisruptionBudget リソースを作成して、同時中断の数を 1 つのプライマリインスタンスに制限します。この構成により、メンテナンス操作によってクラスター内のすべてのポッドが削除されるのを防ぎ、指定された数のインスタンスを作成できるようになります。PodDisruptionBudget は、ノードのドレイン操作中に適用され、クラスターサービスの中断を防ぎます。

この戦略は、ストレージがすべてのワーカーノード間で共有される Kubernetes クラスターには適していますが、ローカルストレージを使用するクラスターや、プライベートクラウドにインストールされたクラスターには最適なソリューションではない可能性があります。オペレーターを使用すると、メンテナンスウィンドウを指定し、基になるノードの削除に対する対応を構成できます。メンテナンスウィンドウ セクションの ReusePVC オプションを使用すると、使用する戦略を指定できます。削除されたインスタンスに別の PVC に新しいストレージを割り当てるか、基になるノードが再び利用可能になるまで待ちます。

フェンシング

フェンシングは、PostgreSQL クラスターの 1 つ、複数、またはすべてのインスタンスのデータが誤動作しているように見える場合に、そのデータを保護するプロセスです。インスタンスがフェンシングされると、ポッドが実行され続けている間、PostgreSQL サーバープロセスがシャットダウンされることが保証されます。これにより、フェンスが解除されるまで、Pod 上のデータが PostgreSQL によって変更されず、デバッグやトラブルシューティングの目的でファイルシステムを調査できることが保証されます。

ハイバネーション

CloudNativePG は hibernation of a running PostgreSQL cluster をサポートします

cnpg プラグイン経由。ハイバネーションは、高可用性クラスター内のすべての Postgres インスタンスをシャットダウンし、PGDATA と WAL を含むプライマリの PVC の静的コピーを保持します。プラグインを使用すると、プライマリを再開してからすべてのレプリカを再作成することで、ハイバネーションフェーズを終了できます (レプリカが存在する場合)。

Pod での永続ボリューム ストレージの再利用

オペレーターが、ユーザーによって削除された、または Kubernetes メンテナンス操作によって削除されたポッドを作成する必要がある場合、利用可能な場合は PersistentVolumeClaim を再利用し、プライマリからデータを再クローンする必要を回避します。

CPU とメモリの要求と制限

オペレーターは、管理者がマニフェストの `resources` セクションを通じて、クラスターのポッドによるリソースの使用を制御および管理できるようにします。特に、`requests` と `limits` の値は、CPU と RAM の両方に設定できます。

PgBouncer による接続プーリング

CloudNativePG は、PostgreSQL の最も一般的なオープン ソース接続プーラーの 1 つである *PgBouncer* **設定オプション** を使用した接続プーリングのネイティブサポートを提供します。アーキテクチャの観点から、PgBouncer 接続プーラーのネイティブ実装は、データベースにアクセスするための新しいレイヤーを導入します。これにより、インスタンスへのクエリフローが最適化され、基盤となる PostgreSQL リソースの使用がより効率的になります。アプリケーションは、PostgreSQL サービスに直接接続する代わりに、PgBouncer サービスに接続して既存の接続を再利用できるようになりました。

4.37.4 レベル 4 - 深い洞察

機能レベル 4 は、 **可観測性**に関するものです。具体的には、監視、アラート、傾向分析、ログ処理です。これには、Prometheus、Grafana、Fluent Bit などの外部ツールの使用や、JSON 形式でエラー ログを直接出力するための PostgreSQL エンジンの拡張機能が含まれる場合があります。

CloudNativePG は、柔軟な監視とロギングのために、業界標準およびコミュニティで受け入れられているツールと簡単に統合するために必要なすべてを提供するように設計されています。

構成可能なクエリを備えた Prometheus エクスポーター

インスタンスマネージャーはプラグ可能なフレームワークを提供し、`metrics` ポート (9187) でリッスンする独自の Web サーバーを介して、 **パート *4 - Prometheus* と *Grafana* を使用してクラスターを監視する**監視およびアラートツールのメトリックをエクスポートするエンドポイントを公開します。オペレーターは、

> postgres_exporter と互換性のある構文を使用して、 `ConfigMap`

および/または `Secret` オブジェクトとして定義されたカスタム監視クエリをサポートします。CloudNativePG は、PostgreSQL 用の一連の基本的な監視クエリを提供します。これらは、統合してコンテキストに適応させることができます。

JSON 形式の PostgreSQL エラー メッセージの標準出力ロギング

すべてのログ メッセージは、標準的な PostgreSQL エラー メッセージチャネルの `postgres` など、タイムスタンプの最初のレベルの定義、ログレベル、およびログ エントリのタイプとともに、JSON 形式で標準出力に配信されます。その結果、CloudNativePG によって管理されるすべての Pod は、JSON をソース データタイプとしてサポートする任意のダウンストリームログ処理スタックと簡単かつ直接統合できます。

リアルタイムのクエリ監視

CloudNativePG は透過的かつネイティブに以下をサポートします。

- 重要な :ref:``pg_stat_statements` の有効化<*pg_stat_statements* の有効化>` 。これにより、PostgreSQL サーバーによって実行されるすべての SQL ステートメントの計画と実行の統計を追跡できます。
- `EXPLAIN` を手動で実行することなく、遅いステートメントの実行計画を自動的にログに記録する手段を提供する

  :ref:``auto_explain` の有効化<*auto_explain* の有効化>` (最適化されていないクエリを追跡するのに役立ちます)。

監査

CloudNativePG を使用すると、データベースとセキュリティの管理者、監査者、およびオペレーターは、PGAudit (PostgreSQL 用) を使用してデータベースアクティビティを追跡および分析できます。このようなアクティビティは JSON ログに直接流れ、Fluentd などの一般的なログブローカーを使用して適切なダウンストリームターゲットに適切にルーティングできます。

Kubernetes イベント

リソースの作成、ノードの削除、アップグレードなど、Kubernetes API が期待する主要なイベントを記録します。イベントは、`kubectl describe` および `kubectl get events` コマンドで表示できます。

4.37.5 レベル 5 - 自動操縦

機能レベル 5 は、 **自動スケーリング** 、 **修復** 、 **チューニング**に重点を置いています。これは、オブザーバビリティレイヤーから明らかになった異常と洞察の発見を通じて行われます。

自己修復のための自動フェイルオーバー

プライマリで障害が検出された場合、オペレーターは、最も整合性のとれたレプリカを新しいターゲットプライマリとして設定することで、クラスターのステータスを変更します。その結果、稼働中の各ポッドのインスタンスマネージャーは、新しいプライマリになるか、それに従うことにより、必要な手順を開始して、クラスターの要求されたステータスに合わせます。以前のプライマリが復旧した場合、同じメカニズムでアプリケーションがプライマリに到達するのを防ぎ、サーバーで `pg_rewind` を実行してスタンバイとして再起動することで、スプリットブレインを回避します。

スタンバイの自動再作成

スタンバイをホストしているポッドが削除された場合、オペレーターはスタンバイサーバーを再作成する手順を開始します。

4.38 カスタム Pod コントローラー

Kubernetes は Controller pattern を使用します

現在のクラスターの状態を目的の状態に合わせます。

ステートフル アプリケーションは通常、 StatefulSet で管理されます。

同じ仕様から構築された一連の Pod を作成して調整し、スティッキー ID を割り当てます。

CloudNativePG は、`StatefulSet` コントローラーに依存する代わりに、独自のカスタムコントローラーを実装して PostgreSQL インスタンスを管理します。この設計の選択により、実装がより複雑になりますが、PostgreSQL クラスターのトポロジーを透過的にしながら、オペレーターはクラスターを管理する方法についてより柔軟に対応できます。

デザイン領域の多くの選択肢と同様に、選択肢が異なれば別の妥協につながります。以下のセクションでは、この設計の選択によって CloudNativePG の実装の信頼性が向上し、理解しやすくなったと思われるいくつかの点について説明します。

4.38.1 PVC のサイズ変更

これは `StatefulSet` のよく知られた制限です: PVC のサイズ変更をサポートしていません。これはデータベースにとって不便です。ボリュームのサイズ変更には、複雑な回避策が必要です。

対照的に、CloudNativePG は構成されたストレージクラスを利用して基盤となる PVC を直接管理し、ストレージクラスがサポートしている場合は PVC のサイズ変更を処理できます。

4.38.2 プライマリ インスタンスとレプリカ

`StatefulSet` コントローラーは、1 つのテンプレートから Pod のセットを作成するように設計されています。 PostgreSQL インスタンスごとに 1 つの `Pod` を使用する場合、2 種類の Pod があります。

1. プライマリ インスタンス (1 つだけ)
2. レプリカ (複数、オプション)

この違いは、特定の操作に対して実行する正しい展開戦略を決定する際に関係します。

一部の操作は、最初にレプリカで実行し、次にプライマリで実行する必要がありますが、更新されたレプリカが新しいプライマリとして昇格された後にのみ実行する必要があります。たとえば、別の PostgreSQL イメージ バージョンを適用する場合や、`max_connections` (treated specially by PostgreSQL because CloudNativePG uses hot standby replicas) などの構成パラメーターを増やす場合です。

その際、CloudNativePG は、シリアル番号だけでなく、PostgreSQL インスタンスの役割を考慮します。

場合によっては、オペレーターは逆のプロセスに従う必要があります。最初にプライマリで作業し、次にレプリカで作業します。たとえば、max_connections を下げる場合。その場合、CloudNativePG は次のことを行います。

- 新しい設定をプライマリ インスタンスに適用します
- 再起動します
- レプリカに新しい設定を適用します

アプリケーションに依存しない StatefulSet コントローラーは、PostgreSQL のネイティブ レプリケーション テクノロジに固有のこの動作を組み込むことができません。

4.38.3 PVC の一貫性

PostgreSQL インスタンスは、複数の PVC で動作するように構成できます。これは、WAL ストレージを PGDATA から分離する方法です。

2 つのデータ ストアは、同時に使用されるため、PostgreSQL の観点から一貫している必要があります。インスタンスの WAL ストレージに対応する PVC を削除すると、PGDATA が格納されている PVC は使用できなくなります。

この動作は PostgreSQL に固有のものであり、StatefulSet コントローラーには実装されていません。後者はアプリケーション固有ではありません。

ユーザーが PVC を削除した後、StatefulSet はそれを再作成するだけで、PostgreSQL インスタンスが破損していました。

代わりに、CloudNativePG は残りの PVC を使用不可として分類し、別のインスタンスがクラスターに正しく参加できるように、PVC の新しいペアの作成を開始します。

4.38.4 ローカル ストレージ、リモート ストレージ、およびデータベース サイズ

場合によっては、アップグレードを行うために Kubernetes ノードを停止する必要があります。アップグレード後、アップグレード戦略に応じて、更新されたノードが再び起動するか、新しいノードに置き換わる可能性があります。

使用できないノードが PostgreSQL インスタンスをホストしていたと仮定すると、データベースのサイズとクラウドインフラストラクチャに応じて、次のいずれかのアクションを選択することをお勧めします。

1. ダウンしたノードにある PVC と Pod をドロップします。別の PVC からデータを複製する新しい PVC を作成します。その後、ポッドをスケジュールします
2. Pod をドロップし、別のノードで Pod をスケジュールし、そこから PVC をマウントします。
3. Pod と PVC をそのままにして、ノードがバックアップされるのを待ちます。

最初のソリューションは、データベースのサイズが許す場合に実用的であり、必要な数のレプリカをすぐに戻すことができます。

2 番目の解決策は、ローカルノードのストレージを使用していない場合にのみ実行可能であり、妥当な時間内に別のホストに PVC を再マウントできます (これは、あなたとあなたの組織だけが知っています)。

3 番目のソリューションは、データベースが大きく、パフォーマンスとデータの耐久性を最大化するためにローカルノード ストレージを使用する場合に適しています。

CloudNativePG コントローラーは、ユーザーがクラスターレベルで優先動作を選択できるように、これらすべての戦略を実装します (詳細については、*Kubernetes* **のアップグレード** セクションを参照してください)。

汎用であるため、`StatefulSet` はこのレベルのカスタマイズを許可しません。

4.39 例

このセクションでは、PostgreSQL クラスターをセットアップするための構成ファイルの例をいくつか紹介します。

重要: これらはデモンストレーションと実験の目的でここにあり、**クイックスタート** で説明されているように、Minikube または Kind を使用して個人の Kubernetes クラスターで実行できます。

基本クラスタ: cluster-example.yaml

クラスターの基本的な例。

カスタム クラスター: cluster-example-custom.yaml

`postgresql.conf` および `pg_hba.conf` ファイルの既定のストレージクラスとカスタム パラメーターを使用する基本的なクラスター。

カスタマイズされたストレージ クラス : cluster-storage-class.yaml : `standard` の指定されたストレージ クラスを使用する基本的なクラスター。

カスタマイズされたストレージ クラスとバックアップ: **前提条件** : バケット ストレージが利用可能である必要があります。サンプル構成は AWS 用です。セットアップに合わせて変更してください: cluster-storage-class-with-backup.yaml バックアップが構成されたクラスター

バックアップ: **前提条件** : cluster-storage-class-with-backup.yaml

applied and Healthy : backup-example.yaml : 前のサンプルに対して実行されるバックアップの例

PVC (永続ボリューム要求) が構成されたクラスター: cluster-pvc-template.yaml : 明示的な永続ボリューム要求テンプレートを使用する基本クラスター。

完全な例: cluster-example-full.yaml : 利用可能なオプションのほとんどを設定するクラスター。

PostGIS の例 : postgis-example.yaml : 「PostGIS クラスター」の例 (詳細については、 PostGIS section を参照してください。)

ストリーミングによるレプリカ クラスタ: **前提条件** : cluster-example.yaml

適用済みで正常: cluster-example-replica-streaming.yaml : `cluster-example` に続くストリーミングレプリケーションを使用するレプリカ クラスター。

バックアップが構成された単純なクラスター: **前提条件** : 構成は、minio が実行中で動作していることを前提としています。minio パラメータまたはクラウド ソリューションで backup.barmanObjectStore を更新してください: cluster-example-with-backup.yaml

バックアップが構成された基本的なクラスター。

バックアップによるレプリカ クラスタ: **前提条件** : cluster-storage-class-with-backup.yaml が適用され、正常であること。そしてバックアップ cluster-example-trigger-backup.yaml

適用され、完了しました。 : cluster-example-replica-from-backup-simple.yaml : バックアップが構成されたクラスターに続くレプリカ クラスター。

SQL ファイルを含むブートストラップ クラスター: cluster-example-initdb-sql-refs.yaml : データベースが作成された直後に Secret と ConfigMap で定義された一連のクエリを実行するクラスターの例。

カスタマイズされた pg_hba 構成のサンプル クラスター: cluster-example-pg-hba.yaml : ユーザー app が証明書を使用して認証できるようにする基本クラスター。

プロジェクション ボリューム テンプレートを使用してシークレットと Configmap をマウントしたサンプル クラスター: cluster-example-projected-volume.yaml

Projected Volume Mount を使用して、既存の Secret と Configmap が Postgres Pod にマウントされた基本クラスター。

利用可能なオプションのリストについては、 *API* リファレンスを参照してください。

4.40 ベンチマーク

CNPG kubectl プラグインは、CloudNativePG を使用して Kubernetes での PostgreSQL デプロイのベンチマークを簡単に行う方法を提供します。

ベンチマークは、次の 2 つの側面に重点を置いています。

- pgbench によるデータベースのベンチマーク に依存する **データベース**
- fio によるストレージのベンチマーク に依存する **ストレージ**

重要: *pgbench* および *fio* は、ステージングまたは実稼働前の環境で実行する必要があります。これらのプラグインを本番環境で使用しないでください。同じ共有環境で実行されるデータベースや他のワークロード/アプリケーションに壊滅的な結果をもたらす可能性があるためです。

4.40.1 pgbench

kubectl CNPG プラグイン コマンド pgbench は、既存の Postgres クラスターでユーザー定義の pgbench ジョブを実行します。このコマンドは--dry-run フラグも受け入れます。これにより、適用せずにジョブマニフェストが出力されます。

使用例:

```
kubectl cnpg pgbench \
  -n <namespace> <cluster-name> \
  --pgbench-job-name <pgbench-job> \
  --db-name <db-name> \
  -- --time 30 --client 1 --jobs 1
```

pgbench 名前空間で cluster-example という名前の Cluster に対して pgbench を実行する方法の例:

```
kubectl cnpg pgbench \
   -n pgbench cluster-example \
   --pgbench-job-name pgbench-job \
   -- --time 30 --client 1 --jobs 1
```

--db-name フラグと pgbench 名前空間を使用して、既存のデータベースで pgbench を実行する方法の例:

```
kubectl cnpg pgbench \
  -n pgbench cluster-example \
  --db-name pgbench \
  --pgbench-job-name pgbench-job \
  -- --time 30 --client 1 --jobs 1
```

ジョブのステータスは、次を実行して取得できます。

```
kubectl get job/pgbench-job -n <namespace>

NAME               COMPLETIONS   DURATION   AGE
pgbench-job-name   1/1           15s        41s
```

ジョブが完了したら、次を実行して結果を収集できます。

```
kubectl logs job/pgbench-job -n <namespace>
```

4.40.2 フィオ

kubectl CNPG プラグイン コマンド `fio` は、デフォルト値と読み取り操作で fio ジョブを実行します。このコマンドは`--dry-run` フラグも受け入れます。これにより、適用せずに fio マニフェストが出力されます。

注釈: kubectl プラグイン コマンド *fio* は、ConfigMap を使用して事前定義された fio ジョブ値でデプロイを作成します。カスタム ジョブ値を提供する場合は、*--dry-run* フラグを使用してマニフェストを生成し、生成された ConfigMap でカスタム ジョブ値を提供することをお勧めします。

使用例:

デフォルト

```
kubectl cnpg fio <fio-name>
``` と値

```shell
kubectl cnpg fio <fio-name> \
  -n <namespace>  \
  --storageClass <name> \
  --pvcSize <size>
```

`fio` 名前空間の `standard` および `pvcSize:  2Gi` という名前の `StorageClass` に対して `fio` コマンドを実行する方法の例:

```
kubectl cnpg fio fio-job \
  -n fio  \
  --storageClass standard \
  --pvcSize 2Gi
```

デプロイのステータスは、次を実行して取得できます。

```
kubectl get deployment/fio-job -n fio

NAME          READY   UP-TO-DATE   AVAILABLE   AGE
fio-job        1/1     1            1           14s
```

kubectl プラグイン コマンド `fio` を実行した後。

そうなる：

1. PVC を作成する
2. fio ジョブの構成を表す ConfigMap を作成する
3. PVC で fio を実行する単一の Pod で構成される fio デプロイメントを作成し、ベンチマークの完了後にグラフを作成し、生成されたファイルの Web サーバーでの提供を開始します。そのために *fio-tools <https://github.com/wallnerryan/fio-tools>`__* イメージを使用します。

結果の提供を開始すると、デプロイによって作成された Pod の準備が整います。デプロイによって作成されたポッドのポートを転送できます

```
kubectl port-forward -n <namespace> deployment/<fio-name> 8000
```

次に、ブラウザーを使用して `http://localhost:8000/` に接続し、データを取得します。

PostgreSQL ワークロードをエミュレートするために、デフォルトの 8k ブロック サイズが選択されています。使用可能な IOPS の量を制限するディスクは、このパラメーターを変更すると、非常に異なるスループット値を示す可能性があります。

以下は、専用の Kubernetes ノードにマウントされたローカルディスクへの順次書き込みの例の図です (1 時間のベンチマーク)。

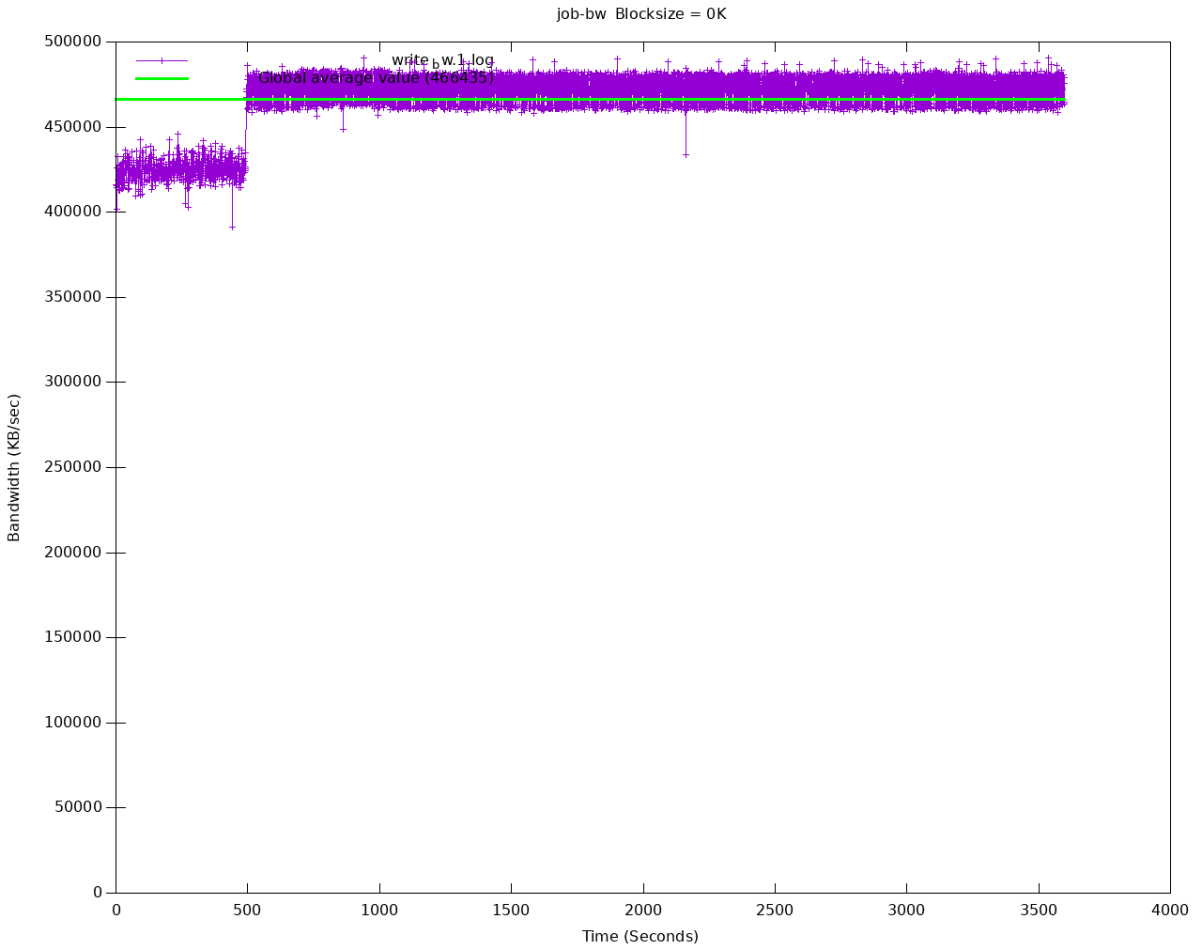


図 18: Sequential writes bandwidth

すべてのテストが完了したら、次の方法で fio のデプロイとリソースを削除できます。

`shell kubectl cnpg fio <fio-job-name> --dry-run | kubectl delete -f -` fio デプロイメントの作成に使用したものと同じ名前を使用し、該当する場合は名前空間を追加してください。

4.41 商用サポート

CloudNativePG は独立したオープン ソースプロジェクトであり、どの企業も支持していません。

これは、CloudNativePG に関連する製品またはサービスを提供するサードパーティの企業および個人のリストです。

CloudNativePG の商用サポートを提供している場合は、このページを編集して、自分自身または組織をリストに追加してください。

リストはアルファベット順に記載されています。

Company	URL	Regions/Countries
スケジュールされたバックアップ	enterprisedb.com	Global

4.42 よくある質問 (FAQ)

4.42.1 Kubernetes で PostgreSQL を実行する

誰もが、PostgreSQL のようなステートフルワークロードを Kubernetes で実行できないことを知っています。なぜ反対のことを言うのですか?

2021 年 9 月の independent research survey commissioned by the Data on Kubernetes Community では、回答者の半数が本番ワークロードのほとんどを Kubernetes で実行していることが明らかになりました。彼らの 90% は、Kubernetes がステートフルなワークロードに対応できると考えており、70% が本番環境でデータベースを実行しています。 Postgres のようなデータベース。ただし、彼らによると、知識のギャップ (一般に、Kubernetes とクラウド ネイティブは学習曲線が急峻です) や Kubernetes オペレーターの質など、重大な課題が残っています。後者は、CloudNativePG のようなオペレーターがプロジェクトの成功に大きく貢献すると私たちが考える理由です。

私たちのようなデータベース狂信者にとって、真のゲームチェンジャーは

Kubernetes 1.14 in April 2019 でのローカル永続ボリュームのサポートの導入でした。

CloudNativePG は不変のアプリケーションコンテナ上に構築されています。どういう意味ですか？

マイクロサービス アーキテクチャパターンに従って、コンテナーは単一のアプリケーションまたはプロセスを実行するように設計されています。その結果、このようなコンテナーイメージは、単一のエントリ ポイント (いわゆる PID 1 プロセス) としてメイン アプリケーションを実行するように構築されます。

Kubernetes の用語では、アプリケーションはワークロードと呼ばれます。ワークロードは、Web アプリケーションサーバーのようにステートレスにすることも、データベースのようにステートフルにすることもできます。この概念を PostgreSQL に当てはめると、不変アプリケーションコンテナーは、単一の特定のバージョン (不変コンテナーイメージ内のバージョン) に関連付けられて実行される単一の “postgres” プロセスです。

SSH、systemd、syslog などの他のプロセスは許可されません。

不変アプリケーションコンテナーは、コンテナーを解釈して使用するための非常に一般的な方法である可変システムコンテナーとは対照的です。

不変とは、コンテナーがその存続期間中に変更されないことを意味します。更新、パッチ、構成の変更はありません。アプリケーションコードを更新するか、パッチを適用する必要がある場合は、新しいイメージをビルドして再デプロイします。不変性により、デプロイがより安全になり、再現性が向上します。

詳細については、 *"Why EDB chose immutable application containers"* を参照してください。

クラウドネイティブとはどういう意味ですか?

Cloud Native Computing Foundation は、「 *Cloud Native* 」という用語を定義しています。ただし、第 2 象限でクラウド ネイティブ PostgreSQL/CloudNativePG オペレーターが開始されて以来、開発チームはクラウド ネイティブを次の 3 つの主要な概念として解釈してきました。

1. チームと組織 (チームのチームとして) が絶え間なく変化し、成果の提供を革新し加速することを可能にする、原則とプロセスだけでなく、人々に基づいた、既存の、健全で、真正で、繁栄した DevOps 文化。より安全で効率的、かつ魅力的な方法でビジネスに価値をもたらす
2. Immutable Application Containers に基づくマイクロサービスアーキテクチャ
3. Kubernetes などのこれらのコンテナーを管理および調整する方法

現在、コンテナ オーケストレーションの事実上の標準は Kubernetes であり、クラウド ネイティブアプリケーションの展開、管理、およびスケーラビリティを自動化します。

私たちの心に響くクラウド ネイティブの別の定義は、*"Kubernetes Patterns", published by O'Reilly* で Ibryam と Huß によって定義されたものです。

> コンテナー化されたマイクロサービスを大規模に自動化するための原則、パターン、ツール

なぜ、PostgreSQL をコンテナーとして実行する代わりにオペレーターを使用する必要があるのですか?

Kubernetes で PostgreSQL を実行するための最も基本的なアプローチは、Kubernetes での展開の最小単位であるポッドを使用して、レプリカなしで Postgres コンテナーを実行することです。 Postgres データディレクトリをホストするボリュームはポッドにマウントされ、通常はネットワークストレージに存在します。この場合、Kubernetes は、問題が発生した場合にポッドを再起動するか、別の Kubernetes ノードに移動します。

最も洗練されたアプローチは、演算子を使用して PostgreSQL を実行することです。オペレーターは Kubernetes コントローラーの拡張であり、複雑なアプリケーションがビジネス継続性のコンテキストでどのように機能するかを定義します。operator パターンは、現在、この目的のための Kubernetes の最先端です。オペレーターは、自動化されたプログラムによる方法で人間のオペレーターの作業をシミュレートします。

Postgres は複雑なアプリケーションであり、オペレーターはクラスターをデプロイするだけでなく (最初のステップ)、予期しないイベントの後に適切に対応する必要があります。典型的な例はフェイルオーバーです。

オペレーターは、自己修復、スケーラビリティ、レプリケーション、高可用性、バックアップ、回復、更新、アクセス、リソース制御、ストレージ管理などの機能を Kubernetes に依存しています。また、ログ管理および監視インフラストラクチャでの PostgreSQL クラスターの統合も容易になります。

CloudNativePG を使用すると、宣言型構成を介して PostgreSQL クラスターの望ましい状態を定義できます。Kubernetes は、Kubernetes コントローラーによって開始された調整ループを通じて、インフラストラクチャの現在の状態が目的の状態と一致することを継続的に確認します。望ましい状態と実際の状態が一致しな

い場合、調整ループが自己修復手順をトリガーします。そこで CloudNativePG のようなオペレーターの出番です。

あなたは、CloudNativePG は完全に宣言型の演算子であると言います。それはどういう意味ですか?

最も簡単な方法は、命令型構成との違いを強調する例を通して宣言型構成を説明することです。命令的なコンテキストでは、状態は、順番に実行される一連のタスクとして定義されます。したがって、最初のインスタンスを作成し、レプリケーションを構成し、2 番目のインスタンスを複製し、3 番目のインスタンスを複製することで、3 ノードの PostgreSQL クラスターを取得できます。

宣言型のアプローチでは、システムの状態は構成を使用して定義されます。つまり、2 つのレプリカを持つ PostgreSQL 13 クラスターがあります。このアプローチにより、変更管理操作が大幅に簡素化され、これらが Git などのソース管理システムに格納されると、Infrastructure as Code 機能が有効になります。そして Kubernetes は、私たちの要求がいつでも確実に満たされるようにするため、デプロイよりも先に進みます。

Kubernetes で PostgreSQL を実行するために必要なスキルは何ですか?

Kubernetes で PostgreSQL を実行するには、DevOps チームに PostgreSQL と Kubernetes の両方のスキルが必要です。最高のエクスペリエンスは、データベース管理者が Kubernetes のコア コンセプトに精通し、Kubernetes 管理者とやり取りできるようになったときです。

クラウド ネイティブ PostgreSQL を最大限に活用して、CNCF 認定プログラムから「Certified Kubernetes Administrator (CKA)」ステータスを取得したいすべての人にアドバイスします。

CloudNativePG が StatefulSets を使用しないのはなぜですか?

CloudNativePG は `StatefulSet` リソースに依存せず、動的プロビジョニング用に選択されたストレージクラスを活用することで、基礎となる PVC を直接管理します。この決定の背後にある詳細と理由については、 **カスタム *Pod* コントローラー**セクションを参照してください。

4.42.2 高可用性

OperatorPod が停止した場合、または一定時間使用できなくなった場合、PostgreSQL クラスターはどうなりますか?

CloudNativePG オペレーターは、とりわけ、自己修復機能を担当します。そのため、オペレーターの停止中は使用できない場合があります。

ただし、停止が PostgreSQL クラスターが実行されているノードに影響を及ぼさないと仮定すると、データベースは関連する Kubernetes サービスを通じて通常の操作を提供し続けます。さらに、各 PostgreSQL ポッド内で実行される **インスタンスマネージャーのインプレース更新**は引き続き機能し、ロギング、メトリックのエクスポート、WAL ファイルの継続的なアーカイブなどのアクセサリ サービスを含むデータベースサーバーが稼働していることを確認します。

要約する：

オペレーターの停止は、必ずしも PostgreSQL データベースの停止を意味するわけではありません。これは、DBA やシステム管理者なしでデータベースを実行するようなものです。

4.43 API リファレンス

CloudNativePG は、次のカスタム リソースを定義する Kubernetes API を拡張します。

- バックアップからのブートストラップ (`recovery )<バックアップからのブートストラップ (*recovery* )>`
- **マルチアベイラビリティ ゾーンの *Kubernetes* クラスタ**
- ***Pooler* リソースのライフサイクル**
- **スケジュールされたバックアップ**

すべてのリソースは `postgresql.cnpg.io/v1` API で定義されています。

使用例については、ドキュメントの Configuration Samples “を参照してください。

以下に、定義されたリソースの説明を示します。

- **アフィニティ構成**
- *AzureCredentials*
- バックアップからのブートストラップ (`recovery )<バックアップからのブートストラップ (*recovery* )>`
- **バックアップ構成**
- **バックアップリスト**
- **バックアップソース**
- **バックアップ仕様**
- **バックアップステータス**
- *BarmanCredentials*
- *BarmanObjectStoreConfiguration*
- **ブートストラップ構成**
- *BootstrapInitDB*
- *BootstrapPgBaseBackup*
- *BootstrapRecovery*
- **証明書の構成**
- **証明書ステータス**
- **マルチアベイラビリティ ゾーンの *Kubernetes* クラスタ**
- *ClusterList*
- *ClusterSpec*
- *ClusterStatus*

- *ConfigMapKeySelector*
- *ConfigMapResourceVersion*
- データバックアップ構成
- *EmbeddedObjectMetadata*
- :ref:``externalClusters` セクション<*externalClusters* セクション>`
- *Google* 資格情報
- *Postgres* データベースのインポート
- インポート元
- インスタンス *ID*
- *InstanceReportedState*
- *LDAPBindAsAuth*
- *LDAPBindSearchAuth*
- *LDAPConfig*
- *LocalObjectReference*
- クラスターのメタデータの定義
- 監視構成
- *NodeMaintenanceWindow*
- *PgBouncerIntegrationStatus*
- *PgBouncerSecrets*
- *PgBouncerSpec*
- *PodTemplateSpec*
- *Pooler* リソースのライフサイクル
- プーラー統合
- プーラーリスト
- *PoolerSecrets*
- プーラースペック
- プーラーのステータス
- *PostInitApplicationSQLRefs*
- *Postgres* 構成
- 回復対象
- *ReplicaClusterConfiguration*

- *ReplicationSlotsConfiguration*
- *ReplicationSlotsHAConfiguration*
- ローリングアップデートステータス
- *S3Credentials*
- スケジュールされたバックアップ
- スケジュールされたバックアップ リスト
- スケジュールされたバックアップ仕様
- スケジュールされたバックアップの状態
- シークレットキーセレクター
- シークレットバージョン
- *SecretsResourceVersion*
- *ServiceAccountTemplate*
- ストレージ構成
- *SyncReplicaElectionConstraints*
- トポロジー
- *WalBackupConfiguration*

4.43.1 アフィニティ構成

AffinityConfiguration には、Pod のアフィニティルールを作成するために必要な情報が含まれています

Name	Description	Type
`enablePodAntiAffinity `	ポッドの非親和性をアクティブにします。このフィールドが明示的に false に設定されていない限り、オペレーターは Pod のアンチアフィニティーを定義します。	*bool
`topologyKey `	非アフィニティ構成に使用する TopologyKey。詳細については、k8s のドキュメントを参照してください - **必須**	string
`nodeSelector `	NodeSelector は、ポッドを実行できるノードを定義するために使用されるキーと値のペアのマップです。詳細: https://kubernetes.io/docs/concepts/configuration/assign-pod-node/	map[string]string

次のページに続く

表 7 – 前のページからの続き

Name	Description	Type
`nodeAffinity`	NodeAffinity は、ポッドのノード アフィニティ スケジューリング ルールを記述します。詳細: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#node-affinity	*corev1.NodeAffinity
`tolerations`	Tolerations は、汚染されたノードで Pod を実行できるようにするために、すべての Pod に設定する必要がある Tolerations のリストです。詳細: https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/	[]corev1.Toleration
`podAntiAffinityType`	PodAntiAffinityType を使用すると、ユーザーは、クラスター インスタンス間のポッドのアンチアフィニティをスケジューリング中に強い要件と見なす必要があるかどうかを決定できます。使用できる値は次のとおりです: "preferred" (空の場合のデフォルト) または "required"。 「必須」に設定すると、既存のすべてのノードが必要なポッドの非アフィニティ ルールと一致しない場合、新しい kubernetes ノードが追加されるまでインスタンスが保留状態になる可能性があります。詳細: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#inter-pod-affinity-and-anti-affinity	string
additionalPodAntiAffinity	AdditionalPodAntiAffinity を使用すると、EnablePodAntiAffinity が true (デフォルト) に設定されている場合、または false に設定されている場合に排他的に使用される場合に、オペレーターによって生成された用語に追加されるポッドのアンチアフィニティ用語を指定できます。	*corev1.PodAntiAffinity
`additionalPodAffinity`	AdditionalPodAffinity を使用すると、クラスターのすべてのポッドに渡されるポッド アフィニティ条件を指定できます。	*corev1.PodAffinity

4.43.2 AzureCredentials

AzureCredentials は、ファイルを Azure Blob Storage にアップロードするために使用される資格情報の種類です。接続文字列には、必要なすべての情報が含まれています。接続文字列が指定されていない場合は、ストレージアカウント名と、次のいずれか (および 1 つだけ) が必要になります。

- storageKey - storageSasToken
- inheritFromAzureAD を true に設定して、ポッド環境から資格情報を継承する

Name	Description	Type
`connectionString`	使用する接続文字列	*SecretKeySelector
`storageAccount`	データをアップロードするストレージ アカウント	*SecretKeySelector
`storageKey`	ストレージ アカウント名と組み合わせて使用するストレージ アカウント キー	*SecretKeySelector
`storageSasToken`	ストレージ アカウント名と組み合わせて使用する共有アクセス署名	*SecretKeySelector
inheritFromAzureAD	キーを明示的に提供せずに、Azure AD ベースの認証を使用します。 - **必須**	bool

4.43.3 バックアップ

Backup はバックアップ API のスキーマです

Name	Description	Type
metadata	NA	metav1.ObjectMeta
`spec`	バックアップの望ましい動作の仕様。詳細: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status	バックアップ仕様
`status`	バックアップの最近観察されたステータス。このデータは最新ではない可能性があります。システムによって設定されます。読み取り専用。詳細: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status	バックアップステータス

4.43.4 バックアップ構成

BackupConfiguration は、クラスターのバックアップの取得方法を定義します。現在サポートされている唯一のバックアップ方法は、barmanObjectStore です。詳細と例については、ドキュメントのバックアップとリカバリのセクションを参照してください。

Name	Description	Type
barmanObjectStore	barman-cloud ツール スイートの構成	*BarmanObjectStoreConfiguration

次のページに続く

表 10 – 前のページからの続き

Name	Description	Type
`retentionPolicy `	RetentionPolicy は、バックアップと WAL に使用される保持ポリシーです (つまり、「60d」)。保持ポリシーは *XXu* の形式で表されます。ここで、*XX* は正の整数で、*u* は *[dwm]* (日、週、月) です。	string
`target `	どのインスタンスがバックアップを実行するかを決定するポリシー。利用可能なオプションは空の文字列で、デフォルトで *primary* ポリシーに設定されます。*primary* はバックアップを常にプライマリ インスタンスで実行し、*prefer-standby* は可能な場合は最新のスタンバイでバックアップを実行することを推奨します。	BackupTarget

4.43.5 バックアップリスト

BackupList には、バックアップのリストが含まれています。

Name	Description	Type
metadata	標準リスト メタデータ。詳細: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#types-kinds	metav1.ListMeta
`items `	バックアップのリスト - **必須**	[]Backup

4.43.6 バックアップソース

BackupSource には、復元する必要があるバックアップと、それを正しく復元するために必要な情報が含まれています。

Name	Description	Type
endpointCA	EndpointCA は、barman エンドポイントの CA バンドルを格納します。証明書発行者と barman-cloud-wal-archive でのエラーを回避するために自己署名証明書を使用する場合に役立ちます。	*SecretKeySelector

4.43.7 バックアップ仕様

BackupSpec は、バックアップの望ましい状態を定義します

Name	Description	Type
cluster	バックアップするクラスター	*LocalObjectReference*
`target `	このバックアップを実行するインスタンスを決定するポリシー。空の場合、デフォルトで *cluster.spec.backup.target* になります。利用可能なオプションは空の文字列で、デフォルトで *primary* ポリシーに設定されます。*primary* は常にプライマリ インスタンスでバックアップを実行し、*prefer-standby* は可能な場合は最新のスタンバイでバックアップを実行することを推奨します。	BackupTarget

4.43.8 バックアップステータス

BackupStatus は、バックアップの監視状態を定義します

Name	Description	Type
`endpointCA `	EndpointCA は、barman エンドポイントの CA バンドルを格納します。証明書発行者と barman-cloud-wal-archive でのエラーを回避するために自己署名証明書を使用する場合に役立ちます。	*SecretKeySelector
`endpointURL `	自動エンドポイント検出をオーバーライドして、データをクラウドにアップロードするために使用されるエンドポイント	string
destinationPath	バックアップを保存するパス (つまり、s3://bucket/path/to/folder) このパスは、異なる宛先フォルダーで、WAL とデータに使用されます。エラーが発生した場合、これは入力されない場合があります。	string
`serverName `	S3 上のサーバー名。このパラメーターを省略した場合はクラスター名が使用されます。	string
`encryption `	S3 API に必要な暗号化方式	string
`backupId `	Barman バックアップの ID	string
`backupName `	Barman バックアップの名前	string
`phase `	最後のバックアップ ステータス	BackupPhase
`startedAt `	バックアップが開始されたとき	*metav1.Time

次のページに続く

表 14 – 前のページからの続き

Name	Description	Type
`stoppedAt`	バックアップが終了したとき	*metav1.Time
`beginWal`	開始 WAL	string
`endWal`	エンディングの WAL	string
`beginLSN`	開始の xlog	string
`endLSN`	エンディングの xlog	string
`error`	検出されたエラー	string
`commandOutput`	未使用。古いバージョンとの互換性のために保持されています。	string
`commandError`	エラーが発生した場合のバックアップ コマンドの出力	string
`instanceID`	バックアップが取得されたインスタンスを識別するための情報	*InstanceID

4.43.9 BarmanCredentials

BarmanCredentials 各クラウドプロバイダーの潜在的な資格情報を含むオブジェクト

Name	Description	Type
googleCredentials	データを Google Cloud Storage にアップロードするために使用する認証情報	*GoogleCredentials
`s3Credentials`	データを S3 にアップロードするために使用する認証情報	*S3Credentials
`azureCredentials`	Azure Blob Storage にデータをアップロードするために使用する資格情報	*AzureCredentials

4.43.10 BarmanObjectStoreConfiguration

BarmanObjectStoreConfiguration には、S3 互換オブジェクトストレージに対して Barman を使用したバックアップ構成が含まれています。

Name	Description	Type
`endpointURL`	自動エンドポイント検出をオーバーライドして、データをクラウドにアップロードするために使用されるエンドポイント	string
`endpointCA`	EndpointCA は、barman エンドポイントの CA バンドルを格納します。証明書発行者と barman-cloud-wal-archive でのエラーを回避するために自己署名証明書を使用する場合に役立ちます	*SecretKeySelector

次のページに続く

表 16 – 前のページからの続き

Name	Description	Type
destinationPath	バックアップを保存するパス (例: s3://bucket/path/to/folder) このパスは、異なる宛先フォルダーと共に、WAL とデータに使用されます - **必須**	string
`serverName`	S3 上のサーバー名。このパラメーターを省略した場合はクラスター名が使用されます。	string
`wal`	WAL ストリームのバックアップの構成。定義されていない場合、バケットのデフォルト ポリシーに従って、WAL ファイルは圧縮されずに保存され、オブジェクト ストアで暗号化されない場合があります。	*WalBackupConfiguration
`data`	データ ファイルのバックアップに使用される構成 定義されていない場合、ベース バックアップ ファイルは圧縮されずに保存され、バケットのデフォルト ポリシーに従って、オブジェクト ストアで暗号化されない場合があります。	*DataBackupConfiguration
`tags`	Tags は、 Barman --tags オプションに渡されるキーと値のペアのリストです。	map[string]string
`historyTags`	HistoryTags は、 Barman --history-tags オプションに渡されるキーと値のペアのリストです。	map[string]string

4.43.11 ブートストラップ構成

BootstrapConfiguration には、PostgreSQL クラスターの作成方法に関する情報が含まれています。サポートされているブートストラップメソッドの中で定義できるのは、1 つのブートストラップメソッドだけです。指定しない場合、`initdb` がブートストラップメソッドとして使用されます。詳細については、ドキュメントの Bootstrap ページを参照してください。

Name	Description	Type
`initdb`	initdb を介してクラスターをブートストラップする	*BootstrapInitDB
`recovery`	バックアップからクラスターをブートストラップする	*BootstrapRecovery
pg_basebackup	別の互換性のある PostgreSQL インスタンスの物理バックアップを取得してクラスターをブートストラップする	*BootstrapPgBaseBackup

4.43.12 BootstrapInitDB

BootstrapInitDB は、initdb を使用する場合のブートストラッププロセスの構成です。詳細については、ドキュメントの Bootstrap ページを参照してください。

Name	Description	Type
`database `	アプリケーションが使用するデータベースの名前。デフォルト: *app*。 - **必須**	string
`owner `	アプリケーションによって使用されるインスタンス内のデータベースの所有者の名前。デフォルトは *database* キーの値です。 - **必須**	string
`secret `	ユーザー データベースの所有者の初期認証情報を含むシークレットの名前。空の場合、新しいシークレットが最初から作成されます	*LocalObjectReference
`options `	クラスターの作成時に initdb に渡す必要があるオプションのリスト。非推奨: これにより、構成に一貫性がなくなる可能性があります。代わりに、明示的に指定されたパラメーターを使用してください。定義されている場合、明示的な値は無視されます。	[]string
`dataChecksums `	*-k* オプションを initdb に渡して、データ ページのチェックサムを有効にするかどうか (デフォルト: *false*)	*bool
`encoding `	initdb のオプション *--encoding* として渡される値 (デフォルト: *UTF8*)	string
`localeCollate `	initdb のオプション *--lc-collate* として渡される値 (デフォルト: *C*)	string
`localeCType `	initdb のオプション *--lc-ctype* として渡される値 (デフォルト: *C*)	string
`walSegmentSize `	initdb の *--wal-segsize* オプションに渡されるメガバイト単位 (1 から 1024) の値 (デフォルト: 空、結果として PostgreSQL のデフォルト: 16MB)	int
`postInitSQL `	クラスターが作成された直後にスーパーユーザーとして実行される SQL クエリのリスト - 細心の注意を払って使用されます (デフォルトでは空)	[]string
`postInitApplication-SQL `	作成直後にアプリケーションデータベースでスーパーユーザーとして実行される SQL クエリのリスト - 細心の注意を払って使用する (デフォルトでは空)	[]string

次のページに続く

表 18 – 前のページからの続き

Name	Description	Type
`postInitTemplateSQL `	クラスターが作成された後に *template1* でスーパーユーザーとして実行される SQL クエリのリスト - 細心の注意を払って使用されます (デフォルトでは空)	[]string
`import `	論理バックアップ (*pg_dump* および *pg_restore*) を使用して既存の PostgreSQL インスタンスからデータをインポートすることにより、新しいクラスターをブートストラップします。	*Import
postInitApplicationSQLRefs	PostInitApplicationSQLRefs は、SQL ファイルを含む ConfigMap または Secret への参照をポイントします。これらの参照の一般的な実装順序は、すべての Secret からすべての ConfigMap への順序であり、Secret または ConfigMap 内では、実装順序は各配列の順序と同じです (デフォルトでは空)。	*PostInitApplicationSQLRefs

4.43.13 BootstrapPgBaseBackup

BootstrapPgBaseBackup には、既存の PostgreSQL クラスターの物理バックアップを作成するために必要な構成が含まれています。

Name	Description	Type
`source `	物理バックアップを取得する必要があるサーバーの名前 - **必須**	string
database	アプリケーションが使用するデータベースの名前。デフォルト: *app*。 - **必須**	string
`owner `	アプリケーションによって使用されるインスタンス内のデータベースの所有者の名前。デフォルトは *database* キーの値です。 - **必須**	string
`secret `	ユーザー データベースの所有者の初期認証情報を含むシークレットの名前。空の場合、新しいシークレットが最初から作成されます	*LocalObjectReference

4.43.14 BootstrapRecovery

BootstrapRecovery には、指定された名前でバックアップを復元するために必要な構成が含まれており、スーパーユーザー用に選択されたパスワードでパスワードを変更した後、それを使用して、復元されたプライマリからすべてのインスタンスを複製する完全なクラスターをブートストラップします。詳細については、ドキュメントの Bootstrap ページを参照してください。

Name	Description	Type
`backup`	復元する必要があるバックアップ	*BackupSource
`source`	バックアップを復元する外部クラスター。これは、バックアップが保存されるフォルダーの名前としても使用されるため、ソース クラスターの名前に設定する必要があります。	string
recoveryTarget	デフォルトでは、リカバリ プロセスはアーカイブ内のすべての利用可能な WAL ファイルを適用します (完全リカバリ)。ただし、一貫性のある状態に達したらすぐにリカバリを終了するか、PostgreSQL で期待されるように *RecoveryTarget* オブジェクトを指定してポイント イン タイム (PITR) にリカバリすることもできます (つまり、タイムスタンプ、トランザクション ID、LSN など)。 .)。詳細: https://www.postgresql.org/docs/current/runtime-config-wal.html#RUNTIME-CONFIG-WAL-RECOVERY-TARGET	*RecoveryTarget
`database`	アプリケーションが使用するデータベースの名前。デフォルト: *app*。 - **必須**	string
`owner`	アプリケーションによって使用されるインスタンス内のデータベースの所有者の名前。デフォルトは *database* キーの値です。 - **必須**	string
`secret`	ユーザー データベースの所有者の初期認証情報を含むシークレットの名前。空の場合、新しいシークレットが最初から作成されます	*LocalObjectReference

4.43.15 証明書の構成

CertificatesConfiguration には、サーバー証明書を処理するために必要な構成が含まれています。

Name	Description	Type
`serverCASecret`	サーバー CA 証明書を含むシークレット。定義されていない場合、自己署名 CA を使用して新しいシークレットが作成され、TLS 証明書 ServerTLSSecret の生成に使用されます。 内容: - *ca.crt*: CA サーバー証明書の検証に使用する必要があり、クライアント接続文字列で *sslrootcert* として使用されます。 - *ca.key*: サーバー SSL 証明書の生成に使用されるキー。ServerTLSSecret が提供されている場合、これは省略できます。 	string
`serverTLSSecret`	クライアントが postgres に安全に接続できるように、サーバー TLS 証明書と *ssl_cert_file* および *ssl_key_file* として設定されるキーを含むタイプ kubernetes.io/tls のシークレット。定義されていない場合、ServerCASecret は *ca.key* も提供する必要があり、提供された CA を使用して新しいシークレットが作成されます。	string
replicationTLSSecret	*streaming_replica* ユーザーとして認証するためのクライアント証明書を含むタイプ kubernetes.io/tls のシークレット。定義されていない場合、ClientCASecret は *ca.key* も提供する必要があり、提供された CA を使用して新しいシークレットが作成されます。	string
`clientCASecret`	クライアント CA 証明書を含むシークレット。定義されていない場合、自己署名 CA を使用して新しいシークレットが作成され、すべてのクライアント証明書の生成に使用されます。 内容: - *ca.crt*: CA クライアント証明書の検証に使用する必要があり、すべてのインスタンスの *ssl_ca_file* として使用されます。 - *ca.key*: クライアント証明書の生成に使用されるキー。ReplicationTLSSecret が提供されている場合、これは省略できます。 	string
`serverAltDNSNames`	必要に応じて、生成されたサーバー TLS 証明書に追加されるサーバーの代替 DNS 名のリスト。	[]string

4.43.16 証明書ステータス

CertificatesStatus には、構成証明書と関連する有効期限が含まれています。

Name	Description	Type
expirations	すべての証明書の有効期限。	map[string]string

4.43.17 集まる

クラスタは PostgreSQL API のスキーマです

Name	Description	Type
metadata	NA	metav1.ObjectMeta
`spec `	クラスターの望ましい動作の仕様。詳細: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status	*ClusterSpec*
`status `	クラスタの最近観測されたステータス。このデータは最新ではない可能性があります。システムによって設定されます。読み取り専用。詳細: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status	*ClusterStatus*

4.43.18 ClusterList

ClusterList には Cluster のリストが含まれています

Name	Description	Type
metadata	標準リスト メタデータ。詳細: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#types-kinds	metav1.ListMeta
`items `	クラスターのリスト - **必須**	[]Cluster

4.43.19 ClusterSpec

ClusterSpec は Cluster の望ましい状態を定義します

Name	Description	Type
`description`	この PostgreSQL クラスターの説明	string
`inheritedMetadata`	クラスタに関連するすべてのオブジェクトに継承されるメタデータ	*EmbeddedObjectMetadata
`imageName`	タグ (*<image>:<tag>*) と決定論的で反復可能なデプロイのダイジェスト (*<image>:<tag>@sha256:<digestValue>*) の両方をサポートするコンテナ イメージの名前	string
`imagePullPolicy`	イメージ プル ポリシー。*Always*、*Never*、または *IfNotPresent* のいずれか。定義されていない場合、デフォルトで *IfNotPresent* になります。更新できません。詳細: https://kubernetes.io/docs/concepts/containers/images#updating-images	corev1.PullPolicy
`postgresUID`	イメージ内の *postgres* ユーザーの UID、デフォルトは *26*	int64
`postgresGID`	イメージ内の *postgres* ユーザーの GID、デフォルトは *26*	int64
`instances`	クラスターに必要なインスタンスの数 - **必須**	int
`minSyncReplicas`	プライマリとの同期レプリケーションに必要なインスタンスの最小数。未定義または 0 を指定すると、使用可能なスタンバイがない場合に書き込みを完了できます。	int
`maxSyncReplicas`	準備ができているスタンバイの数がこれよりも少ない場合は、減らすことができる同期レプリケーション クォーラムのターゲット値。未定義または 0 は、同期レプリケーションを無効にします。	int
`postgresql`	PostgreSQL サーバーの構成	*Postgres* **構成**
`replicationSlots`	レプリケーション スロット管理構成	*ReplicationSlotsConfiguration
`bootstrap`	このクラスターをブートストラップする手順	*BootstrapConfiguration
`replica`	レプリカ クラスタの構成	*ReplicaClusterConfiguration
`superuserSecret`	スーパーユーザーのパスワードを含むシークレット。定義されていない場合、ランダムに生成されたパスワードで新しいシークレットが作成されます	*LocalObjectReference

次のページに続く

表 25 – 前のページからの続き

Name	Description	Type
`enableSuperuserAccess `	このオプションを有効にすると、オペレーターは *SuperuserSecret* を使用して *postgres* ユーザーパスワードを更新します (シークレットが存在しない場合、オペレーターは自動的に作成します)。このオプションを無効にすると、オペレーターは *SuperuserSecret* コンテンツを無視し、自動的に作成されたときに削除し、*postgres* ユーザーのパスワードを *NULL* に設定して空白にします。デフォルトで有効になっています。	*bool
`certificates `	CA および関連する証明書の構成	*CertificatesConfiguration
`imagePullSecrets `	イメージのプルに使用されるプル シークレットのリスト	[]LocalObjectReference
`storage `	インスタンスのストレージの構成	ストレージ構成
`serviceAccountTemplate `	サービス アカウントの生成を構成する	*ServiceAccountTemplate
`walStorage `	PostgreSQL WAL (Write-Ahead Log) のストレージの構成	*StorageConfiguration
`startDelay `	PostgreSQL インスタンスが正常に起動するまでの許容時間 (秒単位) (デフォルトは 30)	int32
`stopDelay `	PostgreSQL インスタンスが正常にシャットダウンするまでの許容時間 (秒単位) (デフォルトは 30)	int32
`switchoverDelay `	スイッチオーバー中にプライマリ PostgreSQL インスタンスが正常にシャットダウンするために許容される秒単位の時間。デフォルト値は 40000000 で、秒単位で 1 年を超え、無限の遅延をシミュレートするのに十分な大きさです	int32
`failoverDelay `	クラスター内のプライマリ PostgreSQL インスタンスが異常であることが検出された後、フェイルオーバーをトリガーするまでの待機時間 (秒単位)	int32
`affinity `	Pod のアフィニティ/アンチアフィニティ ルール	アフィニティ構成
`resources `	生成されたすべての Pod のリソース要件。詳細については、https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/ を参照してください。	corev1.ResourceRequirements
`primaryUpdateStrategy `	すべてのレプリカが正常に更新された後、ローリング更新手順中にプライマリ サーバーをアップグレードするために従う戦略: 自動化 (*unsupervised* - デフォルト) または手動 (*supervised*) のいずれかです。	PrimaryUpdateStrategy

次のページに続く

表 25 – 前のページからの続き

Name	Description	Type
`primaryUpdateMethod`	すべてのレプリカが正常に更新された後、ローリング更新手順中にプライマリ サーバーをアップグレードするために従う方法: スイッチオーバー (*switchover* - デフォルト) またはインプレース (*restart*) を使用できます。	PrimaryUpdateMethod
`backup`	バックアップに使用する構成	*BackupConfiguration
`nodeMaintenanceWindow`	Kubernetes ノードのメンテナンス ウィンドウを定義する	*NodeMaintenanceWindow
`monitoring`	このクラスタの監視インフラストラクチャの構成	*MonitoringConfiguration
`externalClusters`	構成で使用される外部クラスターのリスト	[]ExternalCluster
`logLevel`	インスタンスのログ レベル。次のいずれかの値: error、warning、info (デフォルト)、debug、trace	string
projectedVolumeTemplate	投影されたボリュームを定義するために使用されるテンプレート。投影されたボリュームは */projected* ベース フォルダーの下にマウントされます	*corev1.ProjectedVolumeSource
`env`	Env は Env 形式に従って、クラスターで作成されたポッドに環境変数を渡します	[]corev1.EnvVar
`envFrom`	EnvFrom は EnvFrom 形式に従って、環境変数ソースを Pod に渡し、Env によって使用されます。	[]corev1.EnvFromSource

4.43.20 ClusterStatus

ClusterStatus は、クラスタの監視状態を定義します

Name	Description	Type
`instances`	クラスタで検出された PVC グループの総数。既存のインスタンス ポッドの数とは異なる場合があります。	int
`readyInstances`	クラスター内の準備ができているインスタンスの総数。これは、準備ができているインスタンス ポッドの数と同じです。	int
`instancesStatus`	InstancesStatus は、インスタンスがどのステータスにあるかを示します	map[utils.PodStatus][]string
`instancesReportedState`	最後の調整ループ中に報告されたインスタンスの状態	map[PodName]InstanceReportedState
`timelineID`	Postgres クラスターのタイムライン	int
`topology`	インスタンス トポロジ。	トポロジー
`latestGeneratedNode`	最後に生成されたノードの ID (ノード名の衝突を避けるために使用)	int

次のページに続く

表 26 – 前のページからの続き

Name	Description	Type
`currentPrimary`	現在のプライマリ インスタンス	string
`targetPrimary`	ターゲット プライマリ インスタンス。これは、スイッチオーバーまたはフェイルオーバー中の以前のものとは異なります	string
`pvcCount`	このクラスタによって作成された PVC の数	int32
`jobCount`	このクラスターによって作成されたジョブの数	int32
`danglingPVC`	このクラスタによって作成され、Pod に接続されていないまだ使用可能なすべての PVC のリスト	[]string
`resizingPVC`	ResizingPVC 条件を持つすべての PVC のリスト。	[]string
`initializingPVC`	このクラスタによって初期化されているすべての PVC のリスト	[]string
`healthyPVC`	ダングリングも初期化もしていないすべての PVC のリスト	[]string
`unusablePVC`	別の PVC がないために使用できないすべての PVC のリスト	[]string
`writeService`	現在の書き込みポッド	string
`readService`	読み取りポッドの現在のリスト	string
`phase`	クラスターの現在のフェーズ	string
`phaseReason`	現在のフェーズの理由	string
`secretsResourceVersion`	オペレーターが管理するシークレットのリソース バージョンのリスト。ここでのすべての変更は、シークレット データを更新するインスタンス マネージャーのために行われます。	*SecretsResourceVersion*
`configMapResourceVersion`	オペレーターによって管理される configmap のリソース バージョンのリスト。ここでのすべての変更は、configmap データを更新するインスタンス マネージャーのために行われます。	*ConfigMapResourceVersion*
`certificates`	デフォルトで初期化された、CA および関連する証明書の構成。	証明書ステータス
`firstRecoverabilityPoint`	RFC3339 形式の日付として保存される最初の回復可能ポイント	string
`lastSuccessfulBackup`	RFC3339 形式の日付として保存	string
`lastFailedBackup`	RFC3339 形式の日付として保存	string
`cloudNativePGCommitHash`	このオペレーターが実行しているコミットハッシュ番号	string
`currentPrimaryTimestamp`	プライマリへの最後の実際の昇格が発生したときのタイムスタンプ	string
currentPrimaryFailingSinceTimestamp	プライマリが異常であることが検出されたときのタイムスタンプ。このフィールドは、spec.failoverDelay が入力されたとき、またはオンライン アップグレード中に報告されます。	string

次のページに続く

表 26 – 前のページからの続き

Name	Description	Type
`targetPrimaryTimestamp`	新しいプライマリの最後のリクエストが発生したときのタイムスタンプ	string
`poolerIntegrations`	クラスターを参照するプーラーが必要とする統合	*PoolerIntegrations
`cloudNativePGOperatorHash`	演算子のバイナリのハッシュ	string
`onlineUpdateEnabled`	OnlineUpdateEnabled は、クラスター内でオンライン アップグレードが有効になっているかどうかを示します	bool
`azurePVCUpdateEnabled`	AzurePVCUpdateEnabled は、このクラスターで PVC オンライン アップグレードが有効になっているかどうかを示します	bool
`conditions`	クラスタ オブジェクトの条件	[]metav1.Condition
`instanceNames`	クラスター内のインスタンス名のリスト	[]string

4.43.21 ConfigMapKeySelector

ConfigMapKeySelector には、ConfigMap のキーを見つけるのに十分な情報が含まれています

Name	Description	Type
key	選択するキー - **必須**	string

4.43.22 ConfigMapResourceVersion

ConfigMapResourceVersion は、オペレーターが管理するシークレットのリソース バージョンです。

Name	Description	Type
metrics	メトリックを渡すために使用されるすべての構成マップのバージョンを含むマップ。マップ キーは構成マップ名、マップ値はバージョンです。	map[string]string

4.43.23 データバックアップ構成

DataBackupConfiguration は、データディレクトリのバックアップの構成です。

Name	Description	Type
`compression `	オブジェクト ストアにストリーミングしながら、バックアップ ファイル (テーブルスペースごとに tar ファイル) を圧縮します。利用可能なオプションは、空の文字列 (圧縮なし、デフォルト)、*gzip*、*bzip2*、または *snappy* です。	CompressionType
`encryption `	ファイルの暗号化を強制する場合 (バケットがまだそのように構成されていない場合)。許可されているオプションは、空の文字列 (バケット ポリシーを使用、デフォルト)、*AES256* および *aws:kms* です。	EncryptionType
immediateCheckpoint	PostgreSQL サーバーの *checkpoint_completion_target* 設定に従って、バックアップの初期チェックポイントの I/O ワークロードを制限するかどうかを制御します。 true に設定すると、即時チェックポイントが使用されます。つまり、PostgreSQL はできるだけ早くチェックポイントを完了します。デフォルトでは `false`。	bool
`jobs `	バックアップのアップロードに使用される並列ジョブの数。デフォルトは 2 です。	*int32

4.43.24 EmbeddedObjectMetadata

EmbeddedObjectMetadata には、クラスタに関連するすべてのリソースによって継承されるメタデータが含まれています

Name	Description	Type
`labels `	NA	map[string]string
annotations	NA	map[string]string

4.43.25 外部クラスター

ExternalCluster は、構成の他のセクションで使用される外部クラスターへの接続パラメーターを表します。

Name	Description	Type
`name `	サーバー名、必須 - **必須**	string
connectionParameters	データベース名、ホスト、ユーザー名などの接続パラメータのリスト	map[string]string
`sslCert `	このインスタンスへの接続に使用される SSL 証明書への参照	*corev1.SecretKeySelector
`sslKey `	このインスタンスへの接続に使用される SSL 秘密鍵への参照	*corev1.SecretKeySelector
`sslRootCert `	このインスタンスへの接続に使用される SSL CA 公開鍵への参照	*corev1.SecretKeySelector
`password `	サーバーへの接続に使用するパスワードへの参照	*corev1.SecretKeySelector
`barmanObjectStore `	barman-cloud ツール スイートの構成	*BarmanObjectStoreConfiguration

4.43.26 Google 資格情報

GoogleCredentials は、Google Cloud Storage 資格情報のタイプです。これは、GKE 環境内で実行する場合でも指定する必要があります。

Name	Description	Type
`gkeEnvironment `	true に設定すると、GKE 環境内で実行されていると見なされます。デフォルトは false です。 - **必須**	bool
applicationCredentials	認証情報を含む Google Cloud Storage JSON ファイルを含むシークレット	*SecretKeySelector

4.43.27 輸入

インポートには、externalCluster のロジックスナップショットからデータベースを初期化するための構成が含まれています

Name	Description	Type
`source `	インポート元 - **必須**	インポート元
`type `	インポートの種類。 *microservice* または *monolith* にすることができます。 - **必須**	SnapshotType
`databases `	インポートするデータベース - **必須**	[]string
`roles `	インポートする役割	[]string
postImportApplicationSQL	インポート直後にアプリケーション データベースでスーパーユーザーとして実行される SQL クエリのリスト - 細心の注意を払って使用します (デフォルトでは空)。マイクロサービス タイプでのみ使用できます。	[]string

4.43.28 インポート元

ImportSource は、論理スナップショットのソースを記述します

Name	Description	Type
externalCluster	インポートに使用される externalCluster の名前 - **必須**	string

4.43.29 インスタンス ID

InstanceID には、インスタンスを識別するための情報が含まれています

Name	Description	Type
`podName `	ポッド名	string
ContainerID	コンテナー ID	string

4.43.30 InstanceReportedState

InstanceReportedState は、調整ループ中に最後に報告されたインスタンスの状態を示します

Name	Description	Type
`isPrimary`	インスタンスがプライマリかどうかを示します - **必須**	bool
timeLineID	インスタンスがどの TimelineId にあるかを示します	int

4.43.31 LDAPBindAsAuth

LDAPBindAsAuth は、LDAP のバインド認証を使用するために必要なフィールドを提供します

Name	Description	Type
prefix	バインド認証オプションのプレフィックス	string
suffix	バインド認証オプションのサフィックス	string

4.43.32 LDAPBindSearchAuth

LDAPBindSearchAuth は、バインド + 検索 LDAP 認証プロセスを使用するために必要なフィールドを提供します。

Name	Description	Type
`baseDN`	ユーザー検索を開始するためのルート DN	string
`bindDN`	ディレクトリにバインドするユーザーの DN	string
`bindPassword`	ユーザーがディレクトリにバインドするためのパスワードを含むシークレット	*corev1.SecretKeySelector
searchAttribute	ユーザー名と照合する属性	string
`searchFilter`	検索+バインド認証を行うときに使用する検索フィルター	string

4.43.33 LDAPConfig

LDAPConfig には、LDAP 認証に必要なパラメーターが含まれています

Name	Description	Type
`server`	LDAP ホスト名または IP アドレス	string
`port`	LDAP サーバー ポート	int
`scheme`	使用する LDAP スキーマ。可能なオプションは *ldap* と *ldaps* です	LDAPScheme
`tls`	LDAP over TLS を有効にするには、「true」に設定します。 「false」がデフォルトです	bool
`bindAsAuth`	認証構成としてバインド	*LDAPBindAsAuth
bindSearchAuth	Bind+Search 認証構成	*LDAPBindSearchAuth

4.43.34 LocalObjectReference

LocalObjectReference には、同じ名前空間内の既知の型を持つローカルオブジェクトを見つけるのに十分な情報が含まれています

Name	Description	Type
name	参照対象の名前。 - **必須**	string

4.43.35 メタデータ

メタデータは metav1.ObjectMeta に似た構造ですが、controller-gen で解析して、ユーザーに適した CRD を作成できます。 PodTemplateSpec のコメントには、コア データ型を使用しない理由が説明されています。

Name	Description	Type
`labels`	オブジェクトの整理と分類 (スコープと選択) に使用できる文字列キーと値のマップ。レプリケーション コントローラーおよびサービスのセレクターと一致する場合があります。詳細: http://kubernetes.io/docs/user-guide/labels	map[string]string

次のページに続く

表 41 – 前のページからの続き

Name	Description	Type
annotations	Annotations は、任意のメタデータを保存および取得するために外部ツールによって設定される可能性がある、リソースと共に保存される構造化されていないキー値マップです。これらはクエリ可能ではなく、オブジェクトを変更するときに保持する必要があります。詳細: http://kubernetes.io/docs/user-guide/annotations	map[string]string

4.43.36 監視構成

MonitoringConfiguration は、特定のクラスターのすべてのモニタリング構成を含むタイプです

Name	Description	Type
`disableDefaultQueries `	デフォルトのクエリを挿入するかどうか。デフォルトのクエリをクラスターに挿入したくない場合は、*true* に設定します。デフォルト: false。	*bool
customQueriesConfigMap	カスタム クエリを含む構成マップのリスト	[]ConfigMapKeySelector
`customQueriesSecret `	カスタム クエリを含むシークレットのリスト	[]SecretKeySelector
`enablePodMonitor `	*PodMonitor* を有効または無効にする	bool

4.43.37 NodeMaintenanceWindow

NodeMaintenanceWindow には、基礎となるノードのアップグレード中にオペレーターが使用する情報が含まれています。

このオプションは、選択したストレージによって Pod がノード間で自由に移動できない場合にのみ役立ちます。

Name	Description	Type
inProgress	進行中のノード メンテナンス アクティビティはありますか? - **必須**	bool
`reusePVC `	既存の PVC を再利用する (ノードが再起動するのを待つ) またはしない (別の場所で再作成する - *instances* >1 の場合) - **必須**	*bool

4.43.38 PgBouncerIntegrationStatus

PgBouncerIntegrationStatus は、クラスターを参照する pgbouncer プーラーに必要な統合をカプセル化します

Name	Description	Type
secrets	NA	[]string

4.43.39 PgBouncerSecrets

PgBouncerSecrets には、pgbouncer が使用するシークレットのバージョンが含まれています。

Name	Description	Type
authQuery	認証クエリのシークレット バージョン	シークレットバージョン

4.43.40 PgBouncerSpec

PgBouncerSpec は PgBouncer の設定方法を定義します

Name	Description	Type
`poolMode `	プールモード - **必須**	PgBouncerPoolMode
authQuerySecret	認証クエリに使用する必要があるユーザーの資格情報。指定されている場合は、AuthQuery (例: "SELECT usename, passwd FROM pg_shadow WHERE usename=$1") も指定する必要があり、CNPG クラスターの自動統合はトリガーされません。	*LocalObjectReference
`authQuery `	特定のユーザーのパスワードのハッシュをダウンロードするために使用されるクエリ。デフォルト: "SELECT ユーザー名、パスワード FROM user_search($1)"。指定されている場合は、AuthQuerySecret も指定する必要があり、CNPG クラスターの自動統合はトリガーされません。	string
`parameters `	PgBouncer に渡される追加のパラメーター - 構成できるオプションのリストについては、CNPG のドキュメントを確認してください。	map[string]string
`pg_hba `	PostgreSQL ホスト ベース認証ルール (pg_hba.conf ファイルに追加する行)	[]string

次のページに続く

表 46 – 前のページからの続き

Name	Description	Type
`paused `	*true* に設定すると、PgBouncer は PostgreSQL サーバーから切断し、最初にすべてのクエリが完了するのを待ち、この値が *false* (デフォルト) に設定されるまですべての新しいクライアント接続を一時停止します。内部的に、オペレーターは PgBouncer の *PAUSE* および *RESUME* コマンドを呼び出します。	*bool

4.43.41 PodTemplateSpec

PodTemplateSpec は、ユーザーが Pod 生成用のテンプレートを設定できるようにする構造です。

残念ながら、生成された CRD にはメタデータセクションのフィールドがないため、corev1.PodTemplateSpec タイプは使用できません。

参考文献: https://github.com/kubernetes-sigs/controller-tools/issues/385 https://github.com/kubernetes-sigs/controller-tools/issues/448 https://github.com/prometheus-オペレーター/プロメテウス-オペレーター/問題/3041

Name	Description	Type
metadata	標準オブジェクトのメタデータ。詳細: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#metadata	クラスターのメタデータの定義
`spec `	Pod の望ましい動作の仕様。詳細: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status	corev1.PodSpec

4.43.42 プーラー

Pooler はプーラー API のスキーマです

Name	Description	Type
metadata	NA	metav1.ObjectMeta
`spec `	NA	プーラースペック
`status `	NA	プーラーのステータス

4.43.43 プーラー統合

PoolerIntegrations は、クラスターを参照するプーラーに必要な統合をカプセル化します

Name	Description	Type
pgBouncerIntegration	NA	*PgBouncerIntegrationStatus*

4.43.44 プーラーリスト

PoolerList には Pooler のリストが含まれています

Name	Description	Type
metadata	NA	metav1.ListMeta
`items `	• **必須**	[]Pooler

4.43.45 PoolerSecrets

PoolerSecrets には、使用されるすべてのシークレットのバージョンが含まれています

Name	Description	Type
`serverTLS `	サーバーの TLS シークレット バージョン	シークレットバージョン
`serverCA `	サーバー CA シークレット バージョン	シークレットバージョン
`clientCA `	クライアント CA シークレット バージョン	シークレットバージョン
pgBouncerSecrets	PgBouncer が使用するシークレットのバージョン	*PgBouncerSecrets

4.43.46 プーラースペック

PoolerSpec は、プーラーの望ましい状態を定義します

Name	Description	Type
`cluster`	これは、プーラーが機能するクラスター参照です。プーラー名は、同じ名前空間内のどのクラスター名とも一致しないはずです。 - **必須**	*LocalObjectReference*
`type`	どのインスタンスにトラフィックを転送する必要がありますか? - **必須**	PoolerType
instances	必要なレプリカの数 - **必須**	int32
`template`	作成する Pod のテンプレート	*PodTemplateSpec
pgbouncer	PgBouncer 構成 - **必須**	*PgBouncerSpec

4.43.47 プーラーのステータス

PoolerStatus は、監視された Pooler の状態を定義します

Name	Description	Type
`secrets`	構成オブジェクトのリソース バージョン	*PoolerSecrets
instances	スケジュールしようとしているポッドの数	int32

4.43.48 PostInitApplicationSQLRefs

PostInitApplicationSQLRefs は、 SQL ファイルを含む ConfigMap または Secret への参照をポイントします。これらの参照の一般的な実装順序は、すべての Secret からすべての ConfigMap への順序であり、Secret または ConfigMap 内では、実装順序は各配列の順序と同じです。

Name	Description	Type
`secretRefs`	SecretRefs は、シークレットへの参照のリストを保持します	[]SecretKeySelector
configMapRefs	ConfigMapRefs は、ConfigMaps への参照のリストを保持します	[]ConfigMapKeySelector

4.43.49 Postgres 構成

PostgresConfiguration は PostgreSQL 構成を定義します

Name	Description	Type
`parameters`	PostgreSQL 構成オプション (postgresql.conf)	map[string]string
`pg_hba`	PostgreSQL ホスト ベース認証ルール (pg_hba.conf ファイルに追加する行)	[]string

次のページに続く

表 55 – 前のページからの続き

Name	Description	Type
syncReplicaElection-Constraint	同期レプリカが満たすべき要件。これは、「synchronous_standby_names」パラメーターの設定方法に影響します。	*SyncReplicaElectionConstraints*
`promotionTimeout `	インスタンスをプライマリに昇格するときに待機する最大秒数を指定します。デフォルト値は 40000000 で、秒単位で 1 年を超え、無限のタイムアウトをシミュレートするのに十分な大きさです	int32
`shared_preload_librarie `	デフォルトのものに追加する共有プリロード ライブラリのリスト	[]string
`ldap `	LDAP 構成を指定するオプション	*LDAPConfig

4.43.50 回復対象

RecoveryTarget を使用すると、回復プロセスが停止する瞬間を構成できます。TargetTLI を除くすべてのターゲット オプションは相互に排他的です。

Name	Description	Type
`backupID `	リカバリ プロセスを開始するバックアップの ID。空の場合 (デフォルト)、オペレーターは targetTime または targetLSN (指定されている場合) に基づいてバックアップを自動的に検出します。それ以外の場合は、利用可能な最新のバックアップを時系列で使用します。	string
`targetTLI `	ターゲットのタイムライン (「最新」または正の整数)	string
`targetXID `	対象取引 ID	string
`targetName `	ターゲット名 (事前に *pg_create_restore_point* で作成)	string
`targetLSN `	ターゲット LSN (ログ シーケンス番号)	string
`targetTime `	RFC3339 標準のタイムスタンプとしてのターゲット時刻	string
targetImmediate	一貫した状態に達したらすぐに回復を終了する	*bool
`exclusive `	ターゲットを排他的に設定します (デフォルトは true)。	*bool

4.43.51 ReplicaClusterConfiguration

ReplicaClusterConfiguration は、レプリカクラスターの構成をカプセル化します。

Name	Description	Type
enabled	レプリカ モードが有効になっている場合、このクラスターは既存のクラスターのレプリカになります。レプリカ クラスタは、リカバリ オブジェクト ストアから、または pg_basebackup を介したストリーミングを介して作成できます。詳細については、ドキュメントのレプリケーションページを参照してください。 - **必須**	bool
`source `	レプリケーション元である外部クラスターの名前 - **必須**	string

4.43.52 ReplicationSlotsConfiguration

ReplicationSlotsConfiguration は、レプリケーションスロットの構成をカプセル化します

Name	Description	Type
highAvailability	高可用性構成のレプリケーション スロット	*ReplicationSlotsHAConfiguration
`updateInterval `	スタンバイは、ローカル レプリケーション スロットのステータスを *updateInterval* 秒 (デフォルト 30) ごとに更新します。	int

4.43.53 ReplicationSlotsHAConfiguration

ReplicationSlotsHAConfiguration は、オペレーターによって自動的に管理されるレプリケーションスロットの構成をカプセル化し、高可用性 (HA) の目的でスタンバイインスタンスとのストリーミングレプリケーション接続を制御します。レプリケーションスロットは、ストリーミング クライアント (この場合は HA クラスターの一部であるレプリカ) が切断されたときに、PostgreSQL が WAL ファイルをプライマリに自動的に保持するようにする PostgreSQL の機能です。

Name	Description	Type
`enabled`	有効にすると、オペレータはプライマリ インスタンスのレプリケーション スロットを自動的に管理し、HA クラスタの一部であるすべてのスタンバイ インスタンスとのストリーミングレプリケーション接続でそれらを使用します。無効 (デフォルト) の場合、オペレータはレプリカとのストリーミング接続でレプリケーション スロットを利用しません。この機能は、指定されたプライマリからそのカスケード レプリカまで、レプリカ クラスター内のレプリケーション スロットも制御します。これは、作成時にのみ設定できます。 - **必須**	bool
slotPrefix	HA のオペレーターによって管理されるレプリケーション スロットのプレフィックス。小文字、数字、およびアンダースコア文字のみを含めることができます。これは、作成時にのみ設定できます。デフォルトでは *_cnpg_* に設定されています。	string

4.43.54 ローリングアップデートステータス

RollingUpdateStatus には、更新中のインスタンスに関する情報が含まれています

Name	Description	Type
imageName	Pod に挿入した画像 - **必須**	string
startedAt	アップデートが開始されたとき	metav1.Time

4.43.55 S3Credentials

S3Credentials は、ファイルを S3 にアップロードするために使用される資格情報のタイプです。次の 2 つの方法で提供できます。

- accessKeyId と secretAccessKey を明示的に渡す
- inheritFromIAMRole を true に設定して、ポッド環境からロールを継承する

Name	Description	Type
`accessKeyId`	アクセスキー ID への参照	*SecretKeySelector
`secretAccessKey`	シークレット アクセス キーへの参照	*SecretKeySelector
`region`	リージョン名を含むシークレットへの参照	*SecretKeySelector

次のページに続く

表 61 – 前のページからの続き

Name	Description	Type
`sessionToken`	セッションキーへの参照	*SecretKeySelector
inheritFromIAMRole	キーを明示的に提供せずに、役割ベースの認証を使用します。 - **必須**	bool

4.43.56 スケジュールされたバックアップ

ScheduledBackup は、scheduledbackups API のスキーマです。

Name	Description	Type
metadata	NA	metav1.ObjectMeta
`spec`	ScheduledBackup の望ましい動作の仕様。詳細: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status	スケジュールされたバックアップ仕様
`status`	ScheduledBackup の最近観察されたステータス。このデータは最新ではない可能性があります。システムによって設定されます。読み取り専用。詳細: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status	スケジュールされたバックアップの状態

4.43.57 スケジュールされたバックアップ リスト

ScheduledBackupList には、ScheduledBackup のリストが含まれています

Name	Description	Type
metadata	標準リスト メタデータ。詳細: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#types-kinds	metav1.ListMeta
`items`	クラスターのリスト - **必須**	[]ScheduledBackup

4.43.58 スケジュールされたバックアップ仕様

ScheduledBackupSpec は、ScheduledBackup の望ましい状態を定義します

Name	Description	Type
`suspend `	このバックアップが中断されているかどうか	*bool
`immediate `	最初のバックアップを作成後すぐに開始する必要があるかどうか	*bool
`schedule `	スケジュールは、追加の秒指定子が含まれているため、Kubernetes CronJobs で使用されるのと同じ形式には従いません。https://pkg.go.dev/github.com/robfig/cron#hdr-CRON_Expression_Format - **必須** を参照してください。	string
`cluster `	バックアップするクラスター	*LocalObjectReference*
backupOwnerReference	作成されたバックアップ リソース内にどの ownerReference を配置する必要があるかを示します。 - none: 作成されたバックアップ オブジェクトの所有者参照はありません (フィールドが導入される前と同じ動作) - self: スケジュールされたバックアップ オブジェクトを所有者として設定します。バックアップの - クラスター: クラスターをバックアップの所有者として設定します 	string
`target `	このバックアップを実行するインスタンスを決定するポリシー。空の場合、デフォルトで *cluster.spec.backup.target* になります。利用可能なオプションは空の文字列で、デフォルトで *primary* ポリシーに設定されます。*primary* は常にプライマリ インスタンスでバックアップを実行し、*prefer-standby* は可能な場合は最新のスタンバイでバックアップを実行することを推奨します。	BackupTarget

4.43.59 スケジュールされたバックアップの状態

ScheduledBackupStatus は、ScheduledBackup の監視状態を定義します

Name	Description	Type
`lastCheckTime `	スケジュールの最終時刻	*metav1.Time

次のページに続く

表 65 – 前のページからの続き

Name	Description	Type
lastScheduleTime	バックアップが最後に正常にスケジュールされたときの情報。	*metav1.Time
nextScheduleTime	次回はバックアップを実行します	*metav1.Time

4.43.60 シークレットキーセレクター

SecretKeySelector には、シークレットのキーを見つけるのに十分な情報が含まれています

Name	Description	Type
key	選択するキー - **必須**	string

4.43.61 シークレットバージョン

SecretVersion にはシークレット名とその ResourceVersion が含まれます

Name	Description	Type
`name `	秘密の名前	string
version	シークレットの ResourceVersion	string

4.43.62 SecretsResourceVersion

SecretsResourceVersion は、オペレーターが管理するシークレットのリソースバージョンです。

Name	Description	Type
`superuserSecretVersion `	「postgres」ユーザー シークレットのリソース バージョン	string
replicationSecretVersion	「streaming_replica」ユーザー シークレットのリソース バージョン	string
applicationSecretVersion	「アプリ」ユーザー シークレットのリソース バージョン	string
`caSecretVersion `	未使用。古いバージョンとの互換性のために保持されています。	string
`clientCaSecretVersion `	PostgreSQL クライアント側 CA シークレット バージョンのリソース バージョン	string
`serverCaSecretVersion `	PostgreSQL サーバー側 CA シークレット バージョンのリソース バージョン	string

次のページに続く

表 68 – 前のページからの続き

Name	Description	Type
`serverSecretVersion `	PostgreSQL サーバー側シークレット バージョンのリソース バージョン	string
`barmanEndpointCA `	Barman Endpoint CA のリソース バージョン (提供されている場合)	string
`metrics `	メトリックを渡すために使用されるすべてのシークレットのバージョンを含むマップ。マップ キーはシークレット名、マップ値はバージョンです	map[string]string

4.43.63 ServiceAccountTemplate

ServiceAccountTemplate には、サービスアカウントの生成に必要なテンプレートが含まれています

Name	Description	Type
metadata	メタデータは、生成されたサービス アカウントに使用されるメタデータです - **必須**	クラスターのメタデータの定義

4.43.64 ストレージ構成

StorageConfiguration は、PostgreSQL インスタンスのストレージの構成です

Name	Description	Type
`storageClass `	データベース データ (*PGDATA*) に使用する StorageClass。利用可能な場合、PVC テンプレートを評価した後に適用されます。指定しない場合、生成された PVC はデフォルトのストレージ クラスによって満たされます。	*string
`size `	ストレージのサイズ。 PVC テンプレートでまだ指定されていない場合は必須です。このフィールドへの変更は、作成された PVC に自動的に再適用されます。サイズダウン不可。	string
resizeInUseVolumes	既存の PVC のサイズを変更します。デフォルトは true です	*bool
`pvcTemplate `	Persistent Volume Claim の生成に使用するテンプレート	*corev1.PersistentVolumeClaimSpec

4.43.65 SyncReplicaElectionConstraints

SyncReplicaElectionConstraints には、同期レプリカの選択に関する制約が含まれています。

アンチアフィニティ パラメーターの場合、すべてのラベル値が一致する場合、2 つのインスタンスが同じ場所にあると見なされます。

将来的には、名前による同期レプリカの選択制限がサポートされる予定です。

Name	Description	Type
`enabled `	このフラグは、同期レプリカの制約を有効にします - **必須**	bool
nodeLabelsAntiAffinity	Pod が同じトポロジに存在するかどうかを評価するために抽出および比較するノード ラベル値のリスト	[]string

4.43.66 トポロジー

トポロジにはクラスタ トポロジが含まれます

Name	Description	Type
successfullyExtracted	SuccessfullyExtracted は、トポロジ データが抽出されたかどうかを示します。障害が発生した場合に、同期レプリカの選択でフォールバック動作を制定すると便利です	bool
`instances `	インスタンスには、インスタンスのポッド トポロジが含まれています	map[PodName]PodTopologyLabels

4.43.67 WalBackupConfiguration

WalBackupConfiguration は、WAL ストリームのバックアップの構成です

Name	Description	Type
compression	オブジェクトストアに送信する前に WAL ファイルを圧縮します。利用可能なオプションは、空の文字列 (圧縮なし、デフォルト)、*gzip*、*bzip2*、または *snappy* です。	CompressionType

次のページに続く

表 73 – 前のページからの続き

Name	Description	Type
`encryption `	ファイルの暗号化を強制する場合 (バケットがまだそのように構成されていない場合)。許可されているオプションは、空の文字列 (バケット ポリシーを使用、デフォルト)、*AES256* および *aws:kms* です。	EncryptionType
maxParallel	並行してアーカイブされる (PostgreSQL インスタンスがバックアップ オブジェクト ストアにアーカイブしている場合)、または並行して復元される (PostgreSQL スタンバイが復旧オブジェクト ストアから WAL ファイルをフェッチしている場合) WAL ファイルの数。指定しない場合、WAL ファイルは一度に 1 つずつ処理されます。値として正の整数を受け入れます。受け入れられる最小値は 1 です。	int

4.44 サポートされているリリース

- このページには、現在サポートされている CloudNativePG* のリリースのステータス、タイムライン、およびポリシーが一覧表示されています。

最新の 2 つのマイナー リリースをサポートしており、2 か月ごとに新しいマイナーリリースを作成することを目指しています。各リリースは、 table below で宣言された「サポート終了」日までサポートされます。これは、アップグレード計画を可能にする追加の 1 か月のサポートを考慮したものです。

サポートされている CloudNativePG のリリースには、アクティブなメンテナンスウィンドウにあり、セキュリティとバグ修正のためにパッチが適用されているリリースが含まれます。

マイナー リリースの後続のパッチリリースには、下位互換性のない変更は含まれていません。

- **サポートポリシー**
- **命名スキーム**
- *CloudNativePG* **リリースのサポート状況**
- **サポートとは**

4.44.1 サポートポリシー

コミットごとに CloudNativePG の新しいビルドを生成します。

ほぼ 2 か月ごとに、マイナーリリースをビルドし、いくつかの追加テストとリリース認定を実行します。マイナーリリースで見つかった問題のパッチ バージョンをリリースします。

さまざまなタイプのリリースは、異なる製品品質レベルと CloudNativePG コミュニティからの支援レベルを表しています。コミュニティが提供するサポートの詳細については、以下の

> サポートとは を参照してください。

表 74: :header: Type,Support Level,Quality and Recommended Use :widths: 12,25,15 :align: left :class: longtable

Development Build	No support	Dangerous, may not be fully reliable. Useful to experiment with.
Minor Release	Support provided until 1 month after the N+2 minor release (ex. 1.15 supported until 1 month after 1.17.0 is released)	
Patch	Same as the corresponding Minor release	Users are encouraged to adopt patch releases as soon as they are available for a given release.
Security Patch	Same as a Patch, however, it will not contain any additional code other than the security fix from the previous patch	Given the nature of security fixes, users are **strongly** encouraged to adopt security patches after release.
releases page で利用可能なリリースを見つけることができます。		
CloudNativePG 1.19 のリリース ノート で、各マイナー リリースおよびパッチ		

リリースに関する詳細な情報を見つけることができます。

4.44.2 命名スキーム

私たちの命名スキームは Semantic Versioning 2.0.0 に基づいています

次のように：

```
<major>.<minor>.<patch>
```

ここで、<minor> はリリースごとに増加し、<patch> は現在の<minor> リリースのパッチの数をカウントします。通常、パッチは<minor> リリースに対する小さな変更です。

バージョンの Git タグの前に v が追加されます。

4.44.3 CloudNativePG リリースのサポート状況

Version	Currently Supported	Release Date	End of Life	Supported Kubernetes Versions	Tested, but no
1.19.x	Yes	February 14, 2023	~ July 15, 2023	1.23, 1.24, 1.25, 1.26	1.22
1.18.x	Yes	November 10, 2022	~ May 13, 2023	1.23, 1.24, 1.25, 1.26	1.22
main	No, development only				

上記の表でサポートされている Kubernetes バージョンのリストは、CloudNativePG のメンテナーがサポートおよびテストするのに合理的であると考えるものによって異なります。

現時点では、CloudNativePG コミュニティは、Red Hat OpenShift などの追加の Kubernetes ディストリビューションをサポートしていません (これは将来変更される可能性があります。変更された場合、CloudNativePG メンテナーによって作成された公式ポリシーに反映されます)。

上記の表でサポートされている Postgres バージョンのリストは、通常、マイナーバージョンがリリースされた時点でコミュニティによってサポートされていた PostgreSQL のバージョンによって異なります。

4.44.4 今後のリリース

Version	Release Date	End of Life	Supported Kubernetes Versions
1.20.0	April 13, 2023	•	•
1.21.0	June 15, 2023	•	•

注釈: 機能の凍結はリリースの 1 週間前に発生します

重要: 将来の日付は不確実であり、変更される可能性があります. これは Kubernetes バージョンにも当てはまります。リリース スケジュールの更新と変更は、 Release updates で通知されます。

メインの GitHub リポジトリでの議論。

4.44.5 古いリリース

Version	Release Date	End of Life	Compatible Kubernetes Versions
1.15.x	April 21, 2022	October 6, 2022	1.21, 1.22, 1.23
1.16.x	July 7, 2022	December 21, 2022	1.22, 1.23, 1.24
1.17.x	September 6, 2022	March 20, 2023	1.22, 1.23, 1.24

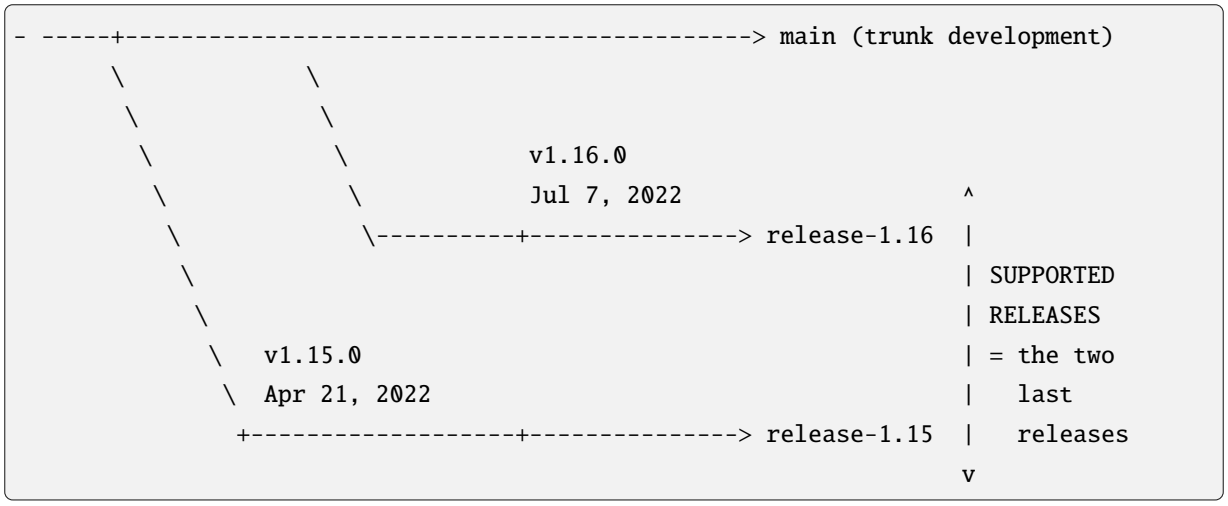
4.44.6 サポートとは

2 か月ごとに新しい最終リリースを作成するため、サポート期間はリリースブランチごとに約 5 か月 (最新の 2 つのマイナー リリースとさらに 1 か月) です。

以下の図では、`release-1.16` がリリース ブランチの例です。

たとえば、最新のリリースが `v1.16.0` であると想像すると、`v1.16.0` と `v1.15.0` の両方のサポートが期待できます。

各ブランチの最後のパッチ リリースのみがサポートされます。



次の 2 種類のサポートを提供しています。

テクニカル サポート : テクニカルサポートは、サポートされているリリースに対してのみベスト エフォートベースで提供されます。 CloudNativePG Slack (`#general` チャネル) または

GitHub Discussions

を使用して、コミュニティからサポートをリクエストできます。

セキュリティとバグ修正 : セキュリティ修正を含む重要なバグ修正を、現在サポートされているすべてのリリースにバックポートします。パッチをバックポートする前に、私たちは次のことを自問します。

専門的なサポートをお探しの場合は、 **表領域のサポートがありません**を参照してください。そこに記載されているベンダーは、サポート期間の延長を含むサービスレベル アグリーメントを提供している場合があります。

4.45 リリースノート

マイナー リリースごとに分類された、CloudNativePG のユーザーに表示される変更の履歴。

- *CloudNativePG 1.19* のリリース ノート
- *CloudNativePG 1.18* のリリース ノート

CloudNativePG のコミュニティ サポート ポリシーについては、**サポートされているリリース**を参照してください。

古いリリース:

- *CloudNativePG 1.17* のリリース ノート
- *CloudNativePG 1.16* のリリース ノート
- *CloudNativePG 1.15* のリリース ノート

また、CloudNativePG の前身であるすべての release notes from 1.0.0 to 1.14.0 of Cloud Native PostgreSQL by EDB の記録も保持します。

4.46 CloudNativePG 1.19 のリリース ノート

CloudNativePG の 1.19 マイナーリリースでユーザーに表示される変更の履歴。

変更の完全なリストについては、 commits を参照してください。

GitHub のリリース ブランチにあります。

4.46.1 バージョン 1.19.1

リリース日: 2023 年 3 月 20 日

機能強化:

- デフォルトのバックアップ ターゲット ポリシーの上書きを許可 (#1602): 以前は、すべてのバックアップとスケジュールされたバックアップでクラスターレベルのターゲット ポリシーが使用されていました。
- `debug` クラスターのログ レベルを `initdb` ジョブに拡張 (#1503)
- PgBouncer プーラーで IPv6 とカスタム `pg_hba` をサポート (#1395)
- 2 つの新しいメトリックとステータスの追加情報を使用して、バックアップの可観測性を強化します (#1428)
- インスタンスマネージャーからのドキュメント API 呼び出し (#1641)
- Helm 経由のデプロイメント名を明確化 (#1505)
- `psql` コマンドを `kubectl` の `cnpg` プラグインに追加 (#1668) して、ユーザーがポッド (デフォルトではプライマリ) で `psql` セッションを開始できるようにします。

技術的な強化:

- 依存関係の追跡/更新に Renovate を採用 (#1367, #1473)
- サポートされているすべてのアーキテクチャのバイナリをオペレーターイメージに挿入 (#1513)
- バックアップ名を使用して、バックアップ オブジェクトストア内のリソースを一致させます (#1650) Barman 3.3 で導入された`--name` オプションを利用して、バックアップとオブジェクトストア間の関連付けをより堅牢にします。

修正:

- プローブのエラー処理でパニックを防止 (#1716)
- HTTP パッケージとコントローラーのランタイム ログが JSON 形式であることを確認します (#1442)
- 単一インスタンス クラスタ内のクラスタに WAL ストレージを追加 (#1570)
- バックアップ コードをより堅牢にするためのさまざまな改善 (#1536、#1564、#1588、#1466、#1647)
- cnpg プラグインの `status` コマンドで WAL アーカイブ情報を適切に表示 (#1666)
- `AdditionalPodAffinity` と `AdditionalPodAntiAffinity` が設定されていない場合でも、`nodeAffinity` が適用されるように (#1663)
- OnlineUpgrading フェーズ中にフェイルオーバー遅延を導入する (#1728) 以前は、オンライン アップグレード プロセスがフェイルオーバーロジックを不必要にトリガーする可能性がありました。

4.46.2 バージョン 1.19.0

リリース日: 2023 年 2 月 14 日

重要なお知らせ:

- PostgreSQL バージョン 10 は EOL に達したため、サポートされなくなりました。バージョン 11 以降がサポートされています。できるだけ早く PostgreSQL 15 への移行を計画してください。 *Postgres データベースのインポート* を参照

PostgreSQL のメジャー オフライン アップグレードの詳細については、

特徴：

- スタンバイからのバックアップ: `.spec.backup.target` オプションを導入し、`prefer-standby` に設定すると、最も整列されたレプリカから物理ベースのバックアップを実行することを受け入れます (#1162)
- フェイルオーバーの遅延: `failoverDelay` パラメーターを導入して、プライマリが異常であることが検出されたらフェイルオーバープロセスを遅延させます (#1366)

機能強化:

- Kubernetes の投影ボリュームのサポートを導入 (#1269)
- PostgreSQL サーバープロセスをより細かく制御するためのカスタム環境変数のサポートを導入 (#1275)

- kubectl の cnpg プラグインに backup コマンドを導入して、クラスターの新しい基本バックアップを発行します (#1348)
- ユーザーが既存の Postgres クラスターの専用ボリュームに WAL ファイルを移動できるようにすることで、個別の WAL ボリューム機能のサポートを改善 (#1066)
- min_wal_size 、max_wal_size 、keep_wal_size 、wal_keep_segments に相当する値、および専用ボリュームに保存できる WAL の最大数を含む、Prometheus エクスポーターの追加メトリックで WAL の可観測性を強化します (#1382)
- streaming_replica ユーザーにデータベース コメントを追加 (#1349)
- GKE の Webhook に関するファイアウォールの問題を文書化 (#1364)
- recovery に postgresql.conf に関するメモを追加 (#1211)
- パッケージを使用してプラグインをインストールする手順を追加 (#1357)
- 各マイナー リリースでサポートされている Postgres のバージョンを指定 (#1355)
- CloudNativePG の PVC グループの意味を明確化 (#1344)
- DigitalOcean S3 互換 Spaces の例を追加 (#1289)
- 新しいクラスター定義のデフォルトの PostgreSQL バージョンを 15.2 に更新 (#1430)
- アーキテクチャ ドキュメントで Kubernetes レイヤーをより詳細にカバー (#1432)

技術的な強化:

- リリース ブランチの毎日のエンド ツー エンド スモーク テストを追加 (#1235)

修正:

- streaming_replica ユーザーとしての CHECKPOINT の実行をスキップ (#1408)
- waitForWalArchiveWorking を接続エラーに対して回復力のあるものにする (#1399)
- PVC ロールが常に一貫していることを確認します (#1380)
- pvcTemplate 使用時に walStorage のサイズ変更を許可 (#1315)
- ExecCommand がタイムアウトに従うように (#1242)
- Prometheus がインストールされていない場合、PodMonitor 調整を回避 (#1238)
- 不要な場合は PodMonitor を探すのを避ける (#1213)

4.47 CloudNativePG 1.18 のリリース ノート

CloudNativePG の 1.18 マイナーリリースでユーザーに表示される変更の履歴。

変更の完全なリストについては、 commits を参照してください。

GitHub のリリース ブランチにあります。

4.47.1 バージョン 1.18.3

リリース日: 2023 年 3 月 20 日

機能強化:

- debug クラスターのログ レベルを `initdb` ジョブに拡張 (#1503)
- PgBouncer プーラーで IPv6 とカスタム `pg_hba` をサポート (#1395)
- 2 つの新しいメトリックとステータスの追加情報を使用して、バックアップの可観測性を強化します (#1428)
- インスタンスマネージャーからのドキュメント API 呼び出し (#1641)
- Helm 経由のデプロイメント名を明確化 (#1505)
- `psql` コマンドを `kubectl` の `cnpg` プラグインに追加 (#1668) して、ユーザーがポッド (デフォルトではプライマリ) で `psql` セッションを開始できるようにします。

技術的な強化:

- 依存関係の追跡/更新に Renovate を採用 (#1367, #1473)
- サポートされているすべてのアーキテクチャのバイナリをオペレーターイメージに挿入 (#1513)
- バックアップ名を使用して、バックアップ オブジェクトストア内のリソースを一致させます (#1650) Barman 3.3 で導入された`--name` オプションを利用して、バックアップとオブジェクトストア間の関連付けをより堅牢にします。

修正:

- プローブのエラー処理でパニックを防止 (#1716)
- HTTP パッケージとコントローラーのランタイム ログが JSON 形式であることを確認します (#1442)
- 単一インスタンス クラスタ内のクラスタに WAL ストレージを追加 (#1570)
- バックアップ コードをより堅牢にするためのさまざまな改善 (#1536、#1564、#1588、#1466、#1647)
- `cnpg` プラグインの `status` コマンドで WAL アーカイブ情報を適切に表示 (#1666)
- `AdditionalPodAffinity` と `AdditionalPodAntiAffinity` が設定されていない場合でも、`nodeAffinity` が適用されるように (#1663)

4.47.2 バージョン 1.18.2

リリース日: 2023 年 2 月 14 日

機能強化:

- Kubernetes の投影ボリュームのサポートを導入 (#1269)
- PostgreSQL サーバープロセスをより細かく制御するためのカスタム環境変数のサポートを導入 (#1275)
- `kubectl` の `cnpg` プラグインに `backup` コマンドを導入して、クラスターの新しい基本バックアップを発行します (#1348)
- ユーザーが既存の Postgres クラスターの専用ボリュームに WAL ファイルを移動できるようにすることで、個別の WAL ボリューム機能のサポートを改善 (#1066)
- `min_wal_size` 、`max_wal_size` 、`keep_wal_size` 、`wal_keep_segments` に相当する値、および専用ボリュームに保存できる WAL の最大数を含む、Prometheus エクスポーターの追加メトリックで WAL の可観測性を強化します (#1382)
- `streaming_replica` ユーザーにデータベース コメントを追加 (#1349)
- GKE の Webhook に関するファイアウォールの問題を文書化 (#1364)
- `recovery` に postgresql.conf に関するメモを追加 (#1211)
- パッケージを使用してプラグインをインストールする手順を追加 (#1357)
- 各マイナー リリースでサポートされている Postgres のバージョンを指定 (#1355)
- CloudNativePG の PVC グループの意味を明確化 (#1344)
- DigitalOcean S3 互換 Spaces の例を追加 (#1289)

技術的な強化:

- リリース ブランチの毎日のエンド ツー エンド スモーク テストを追加 (#1235)

修正:

- `streaming_replica` ユーザーとしての `CHECKPOINT` の実行をスキップ (#1408)
- `waitForWalArchiveWorking` を接続エラーに対して回復力のあるものにする (#1399)
- PVC ロールが常に一貫していることを確認します (#1380)
- `pvcTemplate` 使用時に `walStorage` のサイズ変更を許可 (#1315)
- `ExecCommand` がタイムアウトに従うように (#1242)
- Prometheus がインストールされていない場合、`PodMonitor` 調整を回避 (#1238)
- 不要な場合は `PodMonitor` を探すのを避ける (#1213)

4.47.3 バージョン 1.18.1

リリース日: 2022 年 12 月 21 日

重要なお知らせ:

- CNPG クラスターを識別するためのラベルとしての `postgresql` の差し迫った非推奨について警告します。このラベルを使用したリモートケースでは、代わりに `cnpg.io/cluster` ラベルの使用を開始してください (#1130)
- Armando Ruocco (@armru) を新しい CloudNativePG メンテナーとして表彰 (#1167)
- ARMv7 サポートを削除 (#1092)
- 1.16 の最終パッチ リリース: 1.16.5。リリース 1.16 はサポート終了です。

機能強化:

- サービス アカウントのラベルと注釈をカスタマイズします。たとえば、IRSA を介した GKE または EKS での ID 管理による認証を容易にするために使用できるサービス アカウントテンプレートを追加します (#1105)
- `nodeAffinity` サポートを追加 (#1182) - より豊富なスケジューリングオプションを可能にします
- Istio との互換性の向上: Istio の終了エンドポイントのサポートを追加して、Istio サイドカーを使用するジョブが無期限に実行されないようにします (#967)
- JSON ログでフィールドの再マッピングを許可: `level` および `ts` フィールドが既存のログに干渉する可能性があるユース ケースに役立ちます (#843)
- `fio` コマンドを kubectl-cnpg プラグインに追加 (#1097)
- kubectl-cnpg プラグイン用の rpm/deb パッケージを追加 (#1008)
- 新しいクラスター定義のデフォルトの PostgreSQL バージョンを 15.1 に更新 (#908)
- ドキュメンテーション - CNPG サンドボックスへの参照を削除 (#1120) - CNPG サンドボックスは廃止され、クイックスタートドキュメントの監視に関する指示が優先されました -「リリース更新」ディスカッションへのリンク (#1148) - リリース更新に関するディスカッションは次のようになります。リリースのアナウンスとディスカッションのデフォルトチャネル - GOVERNANCE.md にメンテナの名誉ステータスを文書化 (#1033) - メンテナが貢献を続ける準備ができていない場合にどのように進めるべきかを説明- プル リクエストの作成手順を改善 (#1132) - 緊急時のトラブルシューティング バックアップ手順 (#1184)
- アーキテクチャ ドキュメントで Kubernetes レイヤーをより詳細にカバー (#1432)

修正:

- ブートストラップの PGDATA パーミッションが適切に 750 に設定されていることを確認 (#1164)
- クラスターをスケールダウンする際に、WAL を含む PVC が削除されていることを確認します (#1135)
- `--dry-run` の使用時に pgbench マニフェストに ApiVersion と Kind が欠落している問題を修正 (#1088)
- 見つからない場合にのみシークレットとサービスを作成するようにします (#1145)
- 復元時に設定済みの pg-wal を尊重 (#1216)

- nodeToClusters マップからレプリカを除外 (#1194)

技術的な強化:

- テスト サマリーに ciclops を使用 (#1064): ciclops GitHub アクションに依存して、E2E スイートのサマリーを提供し、そのプロジェクトから改善を継承します。
- バックポート プル リクエスト ワークフローを追加 (#965) - リリースブランチに注釈が付けられている場合、パッチを自動的にバックポートします。
- e2e テスト スイートでオペレータ ログ レベルを設定可能に (#1094)
- ラベルに基づくテスト実行を有効化 (#951)
- Go のバージョンを 1.18 から 1.19 に更新 (#1166)

4.47.4 バージョン 1.18.0

リリース日: 2022 年 11 月 10 日

特徴：

- **高可用性のためのクラスター管理の物理レプリケーションスロット**：プライマリとスタンバイの両方で、高可用性クラスター内の各ホットスタンバイ レプリカの物理レプリケーションスロットを自動的に管理します (#740)
- **Postgres クラスターの休止状態**：新しいサブコマンド `kubectl cnpg hibernate on/off/status <cluster-name>` を使用して、プラグインを介してクラスターの休止状態を導入します。ハイバネーションは、PostgreSQL プライマリ インスタンスに属する PVC を除いて、クラスターによって生成されたすべてのリソースを破棄します (#782)

安全：

- Pod と Containers に `SeccomProfile` を追加 (#888)

機能強化:

- `pvcTemplate` にサイズ要求がある場合、クラスター仕様でストレージサイズを省略できるように (#914)
- cnpg プラグインの `status` コマンド: - レプリケーションスロット情報を追加 (#873) - フェンスされたクラスターの表示を明確化 (#886) - レプリカ クラスターの表示を改善 (#871)
- ドキュメンテーション: - 監視ページを改善し、Prometheus と Grafana を使用してローカル システムで CloudNativePG のオブザーバビリティ機能を評価する方法を説明します (#968) - カスタムコントローラーの設計上の理由に関するページを追加します (#918) - 最後まで更新します- to-End テスト スイート ページ (#945)
- cnpg プラグインの新しいサブコマンド: - `pgbench` はクラスターに対して `pgbench` を実行するジョブ定義を生成します (#958) - `install` はオペレーター用のインストールマニフェストを生成します (#944)
- 新しいデフォルト バージョンとして PostgreSQL 15.0 を設定 (#821)

修正:

- plpgsql 関数でデータベースをインポート (#974)
- ポイントインタイムリカバリを実行する際に、最も近いバックアップを適切に見つける (#949)
- ScheduledBackup フォーマットが Kubernetes CronJob フォーマットに従わないことを明確化 (#883)
- 古い可能性がある Kubernetes ポッドの準備状況ではなく、インスタンスマネージャーからの Postgres 情報に基づいてフェイルオーバーロジックを作成 (#890)
- 新しく作成されたクラスターごとにアーカイブする WAL があることを確認します。欠落により、バックアップが機能しなくなる可能性があります (#897)
- ドキュメントの barmanObjectStore の YAML キー名を修正 (#877)
- krew リリースを修正 (#866)

4.48 CloudNativePG 1.17 のリリース ノート

CloudNativePG の 1.17 マイナーリリースでユーザーに表示される変更の履歴。

変更の完全なリストについては、 commits を参照してください。

GitHub のリリース ブランチにあります。

4.48.1 バージョン 1.17.5

リリース日: 2023 年 3 月 20 日

> **警告:** バージョン 1.17 はサポート終了 (EOL) になりました。バージョン 1.17.5 は、1.17 マイナー バージョン用にリリースされる最後のバージョンです。

機能強化:

- debug クラスターのログ レベルを initdb ジョブに拡張 (#1503)
- PgBouncer プーラーで IPv6 とカスタム pg_hba をサポート (#1395)
- インスタンスマネージャーからのドキュメント API 呼び出し (#1641)
- Helm 経由のデプロイメント名を明確化 (#1505)
- psql コマンドを kubectl の cnpg プラグインに追加 (#1668) して、ユーザーがポッド (デフォルトではプライマリ) で psql セッションを開始できるようにします。

技術的な強化:

- 依存関係の追跡/更新に Renovate を採用 (#1367, #1473)

修正:

- プローブのエラー処理でパニックを防止 (#1716)
- cnpg プラグインの status コマンドで WAL アーカイブ情報を適切に表示 (#1666)

4.48.2 バージョン 1.17.4

リリース日: 2023 年 2 月 14 日

特徴：

- Kubernetes の投影ボリュームのサポート (#1269)
- PostgreSQL サーバープロセスをより細かく制御するためのカスタム環境変数をサポート (#1275)

機能強化:

- kubectl の cnpg プラグインに backup コマンドを導入して、クラスターの新しい基本バックアップを発行します (#1348)
- streaming_replica ユーザーにデータベース コメントを追加 (#1349)
- GKE の Webhook に関するファイアウォールの問題を文書化 (#1364)
- recovery に postgresql.conf に関するメモを追加 (#1211)
- パッケージを使用してプラグインをインストールする手順を追加 (#1357)
- 各マイナー リリースでサポートされている Postgres のバージョンを指定 (#1355)
- CloudNativePG の PVC グループの意味を明確化 (#1344)
- DigitalOcean S3 互換 Spaces の例を追加 (#1289)

技術的な強化:

- リリース ブランチの毎日のエンド ツー エンド スモーク テストを追加 (#1235)

修正:

- PVC ロールが常に一貫していることを確認します (#1380)
- pvcTemplate 使用時に walStorage のサイズ変更を許可 (#1315)
- Prometheus がインストールされていない場合、PodMonitor 調整を回避 (#1238)
- 不要な場合は PodMonitor を探すのを避ける (#1213)

4.48.3 バージョン 1.17.3

リリース日: 2022 年 12 月 21 日

重要なお知らせ:

- Armando Ruocco (@armru) を新しい CloudNativePG メンテナーとして表彰 (#1167)
- ARMv7 サポートを削除 (#1092)
- 1.16 の最終パッチ リリース: 1.16.5。リリース 1.16 はサポート終了です。

機能強化:

- Istio との互換性の向上: Istio の終了エンドポイントのサポートを追加して、Istio サイドカーを使用するジョブが無期限に実行されないようにします (#967)
- kubectl-cnpg プラグイン用の rpm/deb パッケージを追加 (#1008)
- 新しいクラスター定義のデフォルトの PostgreSQL バージョンを 15.1 に更新 (#908)
- ドキュメンテーション - CNPG サンドボックスへの参照を削除 (#1120) - CNPG サンドボックスは廃止され、クイックスタートドキュメントの監視に関する指示が優先されました -「リリース更新」ディスカッションへのリンク (#1148) - リリース更新に関するディスカッションは次のようになります。リリースのアナウンスとディスカッションのデフォルトチャネル - GOVERNANCE.md にメンテナの名誉ステータスを文書化 (#1033) - メンテナが貢献を続ける準備ができていない場合にどのように進めるべきかを説明- プル リクエストの作成手順を改善 (#1132) - 緊急時のトラブルシューティングバックアップ手順 (#1184)
- アーキテクチャ ドキュメントで Kubernetes レイヤーをより詳細にカバー (#1432)

修正:

- ブートストラップの PGDATA パーミッションが適切に 750 に設定されていることを確認 (#1164)
- クラスターをスケールダウンする際に、WAL を含む PVC が削除されていることを確認します (#1135)
- 見つからない場合にのみシークレットとサービスを作成するようにします (#1145)
- 復元時に設定済みの pg-wal を尊重 (#1216)
- nodeToClusters マップからレプリカを除外 (#1194)

技術的な強化:

- テスト サマリーに `ciclops` を使用 (#1064): ciclops GitHub アクションに依存して、E2E スイートのサマリーを提供し、そのプロジェクトから改善を継承します。
- バックポート プル リクエスト ワークフローを追加 (#965) - リリースブランチに注釈が付けられている場合、パッチを自動的にバックポートします。
- e2e テスト スイートでオペレータ ログ レベルを設定可能に (#1094)
- ラベルに基づくテスト実行を有効化 (#951)
- Go のバージョンを 1.18 から 1.19 に更新 (#1166)

4.48.4 バージョン 1.17.2

リリース日: 2022 年 11 月 10 日

安全：

- Pod と Containers に `SeccomProfile` を追加 (#888)

機能強化:

- `cnpg` プラグインの `status` コマンド: - フェンスされたクラスターの表示を明確化 (#886) - レプリカ クラスターの表示を改善 (#871)
- ドキュメンテーション: - 監視ページを改善し、Prometheus と Grafana を使用してローカル システムで CloudNativePG のオブザーバビリティ機能を評価する方法を説明します (#968) - カスタムコントローラーの設計上の理由に関するページを追加します (#918)
- 新しいデフォルト バージョンとして PostgreSQL 15.0 を設定 (#821)

修正:

- `plpgsql` 関数でデータベースをインポート (#974)
- ポイントインタイムリカバリを実行する際に、最も近いバックアップを適切に見つける (#949)
- `ScheduledBackup` フォーマットが Kubernetes `CronJob` フォーマットに従わないことを明確化 (#883)
- 古い可能性がある Kubernetes ポッドの準備状況ではなく、インスタンスマネージャーからの Postgres 情報に基づいてフェイルオーバーロジックを作成 (#890)
- 新しく作成されたクラスターごとにアーカイブする WAL があることを確認します。欠落により、バックアップが機能しなくなる可能性があります (#897)
- ドキュメントの `barmanObjectStore` の YAML キー名を修正 (#877)
- `krew` リリースを修正 (#866)

4.48.5 バージョン 1.17.1

リリース日: 2022 年 10 月 6 日

機能強化:

- コントローラー マネージャーに `leaseDuration` および `renewDeadline` パラメーターを導入して、オペレーターの展開におけるリーダー選出の構成を強化 (#759)
- 初めて WAL ファイルをアーカイブする前に、バックアップ オブジェクトストアが空であることをチェックするメカニズムを改善します。クラスタがブートストラップされた直後に `.check-empty-wal-archive` という新しいファイルが `PGDATA` に配置され、最初の WAL ファイルが実行された後に削除されます。正常にアーカイブされました

安全：

- `nonroot:nonroot` ユーザーを使用して、`distroless/static:nonroot` コンテナーイメージにコピーされたインスタンス マネージャーバイナリのアクセス許可を明示的に設定 (#754)

修正:

- スタンバイがプライマリに昇格した後、アクティブな接続をドロップします (#737)
- ネイティブの Prometheus エクスポーターで MAPPEDMETRIC および DURATION メトリック タイプの変換を尊重する (#765)
- PostgreSQL 形式を使用してマイクロ秒の精度で指定されたタイムスタンプが正しく解析されるように (#741)

4.48.6 バージョン 1.17.0

リリース日: 2022 年 9 月 6 日 (マイナー リリース)

特徴：

- **WAL ファイル用の別のボリューム:** Write Ahead Log (WAL) とデータベース データファイルを別のディスクに分離するためのサポート。データ ディレクトリの I/O 負荷を軽減することで、書き込みの多いシステムでのパフォーマンスが向上する可能性があります。このオプションは、メインおよび必須の storage セクションで定義された PGDATA とは別に、専用ボリューム内の WAL ファイル (pg_wal) を分離するためのオプションの walStorage セクションの導入によって制御されます (#513)。現在の制限: walStorage はクラスターの作成時にのみ設定でき、クラスターが稼働しているときに追加または削除することはできません。

機能強化:

- PgBouncer 接続プーラーの実装で低レベルのネットワーク TCP 設定の構成を有効にします (#584)
- cnpg.io/instanceName および cnpg.io/podRole ラベルがポッドおよび PVC に常に存在することを確認します (#632 および#680)
- インスタンスの role ラベルを基礎となる PVC に伝播 (#634)
- kubectl cnpg destroy コマンドを導入して、インスタンスと関連するすべての PVC を削除できるようにします (#643)

修正:

- 新しいクラスタのプライマリをブートストラップするときに shared_preload_libraries を使用 (#642)
- オペレーターの複数のインプレース アップグレードプロセスが同時に実行されないようにするには、別のプロセスが進行中かどうかをアトミックにチェックします (#655)。
- 新しくアップロードされたインスタンスマネージャーを格納するためにハードコードされたファイル名を使用しないようにし、オペレーターのオンラインアップグレード中に競合状態が発生するのを防ぎます (#660)
- GetAllAccessibleDatabases の呼び出し時にパニックが発生しないように (#641)

4.49 CloudNativePG 1.16 のリリース ノート

CloudNativePG の 1.16 マイナーリリースでユーザーに表示される変更の履歴。

変更の完全なリストについては、 commits を参照してください。

GitHub のリリース ブランチにあります。

4.49.1 バージョン 1.16.5

リリース日: 2022 年 12 月 21 日

> **警告:** バージョン 1.16 はサポート終了 (EOL) になりました。バージョン 1.16.5 は、1.16 マイナー バージョン用にリリースされる最後のバージョンです。

重要なお知らせ:

- Armando Ruocco (@armru) を新しい CloudNativePG メンテナーとして表彰 (#1167)
- ARMv7 サポートを削除 (#1092)
- 1.16 の最終パッチ リリース: 1.16.5。リリース 1.16 はサポート終了です。

機能強化:

- kubectl-cnpg プラグイン用の rpm/deb パッケージを追加 (#1008)
- 新しいクラスター定義のデフォルトの PostgreSQL バージョンを 15.1 に更新 (#908)
- ドキュメンテーション - CNPG サンドボックスへの参照を削除 (#1120) - CNPG サンドボックスは廃止され、クイックスタートドキュメントの監視に関する指示が優先されました -「リリース更新」ディスカッションへのリンク (#1148) - リリース更新に関するディスカッションは次のようになります。リリースのアナウンスとディスカッションのデフォルトチャネル - GOVERNANCE.md にメンテナの名誉ステータスを文書化 (#1033) - メンテナが貢献を続ける準備ができていない場合にどのように進めるべきかを説明- プル リクエストの作成手順を改善 (#1132) - 緊急時のトラブルシューティング バックアップ手順 (#1184)

修正:

- ブートストラップの PGDATA パーミッションが適切に 750 に設定されていることを確認 (#1164)
- 見つからない場合にのみシークレットとサービスを作成するようにします (#1145)
- 復元時に設定済みの pg-wal を尊重 (#1216)
- nodeToClusters マップからレプリカを除外 (#1194)

技術的な強化:

- テスト サマリーに `ciclops` を使用 (#1064): ciclops GitHub アクションに依存して、E2E スイートのサマリーを提供し、そのプロジェクトから改善を継承します。

- バックポート プル リクエスト ワークフローを追加 (#965) - リリースブランチに注釈が付けられている場合、パッチを自動的にバックポートします。
- e2e テスト スイートでオペレータ ログ レベルを設定可能に (#1094)
- ラベルに基づくテスト実行を有効化 (#951)
- Go のバージョンを 1.18 から 1.19 に更新 (#1166)

4.49.2 バージョン 1.16.4

リリース日: 2022 年 11 月 10 日

安全：

- Pod と Containers に `SeccomProfile` を追加 (#888)

機能強化:

- `cnpg` プラグインの `status` コマンド: - フェンスされたクラスターの表示を明確化 (#886) - レプリカクラスターの表示を改善 (#871)
- ドキュメンテーション: - 監視ページを改善し、Prometheus と Grafana を使用してローカル システムで CloudNativePG のオブザーバビリティ機能を評価する方法を説明します (#968) - カスタムコントローラーの設計上の理由に関するページを追加します (#918)

修正:

- `plpgsql` 関数でデータベースをインポート (#974)
- ポイントインタイムリカバリを実行する際に、最も近いバックアップを適切に見つける (#949)
- `ScheduledBackup` フォーマットが Kubernetes `CronJob` フォーマットに従わないことを明確化 (#883)
- 古い可能性がある Kubernetes ポッドの準備状況ではなく、インスタンスマネージャーからの Postgres 情報に基づいてフェイルオーバーロジックを作成 (#890)
- 新しく作成されたクラスターごとにアーカイブする WAL があることを確認します。欠落により、バックアップが機能しなくなる可能性があります (#897)
- ドキュメントの `barmanObjectStore` の YAML キー名を修正 (#877)
- `krew` リリースを修正 (#866)

4.49.3 バージョン 1.16.3

リリース日: 2022 年 10 月 6 日

機能強化:

- コントローラー マネージャーに `leaseDuration` および `renewDeadline` パラメーターを導入して、オペレーターの展開におけるリーダー選出の構成を強化 (#759)

- 初めて WAL ファイルをアーカイブする前に、バックアップ オブジェクトストアが空であることをチェックするメカニズムを改善します。クラスタがブートストラップされた直後に、`check-empty-wal-archive` という新しいファイルが `PGDATA` に配置され、最初の WAL ファイルが実行された後に削除されます。正常にアーカイブされました

安全：

- `nonroot:nonroot` ユーザーを使用して、`distroless/static:nonroot` コンテナーイメージにコピーされたインスタンス マネージャーバイナリのアクセス許可を明示的に設定 (#754)

修正:

- スタンバイがプライマリに昇格した後、アクティブな接続をドロップします (#737)
- ネイティブの Prometheus エクスポーターで MAPPEDMETRIC および DURATION メトリック タイプの変換を尊重する (#765)

4.49.4 バージョン 1.16.2

リリース日: 2022 年 9 月 6 日

機能強化:

- PgBouncer 接続プーラーの実装で低レベルのネットワーク TCP 設定の構成を有効にします (#584)
- `cnpg.io/instanceName` および `cnpg.io/podRole` ラベルがポッドおよび PVC に常に存在することを確認します (#632 および#680)
- インスタンスの `role` ラベルを基礎となる PVC に伝播 (#634)

修正:

- 新しいクラスタのプライマリをブートストラップするときに `shared_preload_libraries` を使用 (#642)
- オペレーターの複数のインプレース アップグレードプロセスが同時に実行されないようにするには、別のプロセスが進行中かどうかをアトミックにチェックします (#655)。
- 新しくアップロードされたインスタンスマネージャーを格納するためにハードコードされたファイル名を使用しないようにし、オペレーターのオンラインアップグレード中に競合状態が発生するのを防ぎます (#660)
- `GetAllAccessibleDatabases` の呼び出し時にパニックが発生しないように (#641)

4.49.5 バージョン 1.16.1

リリース日: 2022 年 8 月 12 日

機能強化:

- PostgreSQL の `huge_pages` オプションの設定を有効化 (#456)
- フェイルオーバーまたはスイッチオーバー後の昇格および降格中のログを強化し、昇格のリクエストから実際に書き込みが可能になるまでの経過時間を出力する (#371)
- クラスタステータスに PostgreSQL クラスタのタイムラインを導入 (#462)
- ジョブ、ポッド、および PVC に `instanceName` および `clusterName` ラベルを追加して、これらのリソースとの相互作用を改善 (#534)
- PostGIS クラスターの作成方法に関する説明を追加 (#570)

安全：

- `securityContext` を `Pooler` デプロイメントに明示的に割り当てる (#485)
- 内部 Web サーバーに読み取りタイムアウト値を追加して、Slowloris DDoS を防止 (#437)

修正:

- 再起動 (`stopDelay`) と切り替え (`switchoverDelay`) に正しい遅延を使用します。これは、以前に誤って交換されたためです。`switchoverDelay` が設定されておらず、デフォルト値の 40000000 秒を使用している場合、再起動が無期限にブロックされる可能性があるため、これは重要な修正です (#531)
- クエリがエラーを返したときにメトリクスコレクターがパニックを引き起こさないようにする (#396)
- 場合によってはログレベルに関係なくセグメンテーション違反につながる、未チェックのポインターを参照していた安全でないデバッグメッセージを削除 (#491)
- クラスターにアノテーションがない場合のフェンシング時のパニックを防止 (#512)
- TLS 証明書が変更されていない場合、CRD の更新を回避 (#501)
- CRD に証明書を挿入する際の競合の処理 (#547)
- データベースのインポート: - データベースのインポートで `pg_restore` を実行中に `postgres` ユーザーを使用 (#411) - オリジン外部サーバーとの SSL 接続を制御するために、モノリスのインポート ケースで `sslmode` を明示的に設定する要件を文書化 (#572) - 妨げていたバグを修正 `connectionParameters` で `dbname` が指定されたときの動作からのインポート (#569)
- バックアップと復元: - Google Cloud でオブジェクトストアの資格情報を正しく渡す (#454)

マイナーな変更：

- デフォルトのオペランド イメージを PostgreSQL 15.0 に設定します。

4.49.6 バージョン 1.16.0

リリース日: 2022 年 7 月 7 日 (マイナー リリース)

特徴：

- **PostgreSQL のオフラインデータインポートとメジャーアップグレード:** `bootstrap.initdb.import` セクションを導入して、PostgreSQL 論理バックアップの概念を使用して、真新しい CloudNativePG クラスター内の既存の PostgreSQL インスタンス (Kubernetes の外部であっても) からネットワーク経由でオブジェクトをインポートする方法を提供します。(`pg_dump` /`pg_restore`)。同じ方法を使用して、新しいクラスターで PostgreSQL のメジャーアップグレードを実行できます。この機能では、`microservice` (新しいクラスターに 1 つのデータベースのみをインポート) と `monolith` (既存のインスタンスから選択したデータベースとロールをインポート) の 2 種類のインポートが導入されています。
- ラベルに基づく同期レプリケーションの非アフィニティ ルール: 同期レプリカが、プライマリが実行されているノードとは異なる特性を持つノード (アベイラビリティ ゾーンなど) で実行されていることを確認します。

機能強化:

- 1 つ以上のインスタンスがフェンシングされている場合にフェイルオーバーを無効にする既存の制限を削除することにより、フェンシングを改善します
- `DROP EXTENSION IF EXISTS` を不必要に実行する代わりに、拡張機能がカタログに存在するかどうかをチェックすることで、自動化された拡張機能管理フレームワークを強化します。
- スイッチオーバーおよびフェイルオーバー中のインスタンスマネージャーのロギングを改善
- オブジェクト ストアから復元するとき、または `pg_basebackup` を介してインスタンスを複製するときに、アプリケーションのデータベースの名前、その所有者、および関連するシークレットの再定義を有効にします (これは、これまで `initdb` ブートストラップでのみ可能でした)。
- バックアップと復旧: - PostgreSQL 15 の将来のサポートのために Barman >= 3.0.0 が必要です - `inheritFromAzureAD` オプションを介して Barman Cloud バックアップ用の Azure AD Workload Identity を有効にします - バックアップと復旧の両方のために AWS (`AWS_DEFAULT_REGION`) でリージョンを定義するために `barmanObjectStore.s3Credentials.region` を導入しますオブジェクトストア
- Kubernetes 1.24 のサポート

変更点:

- デフォルトのオペランド イメージを PostgreSQL 15.0 に設定します。
- バックアップと WAL アーカイブのための独自の実装に依存する代わりに、Kubernetes API の条件を使用します

修正:

- PostgreSQL サーバーの CSV ログ パイプを開始する `WithActiveInstance` 関数内の初期化順序を修正し、クラスターの初期化フェーズで適切なログを確保します- これは、バックアップからの回復などのブートストラップ操作で特に役立ちます (このパッチの前に、そのようなログ標準出力チャネルに送信されず、永久に失われました)

- ホットスタンバイの機密パラメータが減少し、プライマリがすでに再起動されている場合に、不要なスイッチオーバーを回避する
- ALTER ROLE ステートメントでロール名を適切に引用する
- バックアップとリカバリ: - PITR の最も近い Barman バックアップを検出するアルゴリズムを修正します。これは、要求されたリカバリタイムスタンプをバックアップの終了ではなく開始と比較していました。 - トランザクション ID、名前付きの復元ポイント、または recoveryTarget セクションに backupID と呼ばれる新しいフィールドを提供することによる「即時」ターゲット - barman-cloud-wal-archive および barman-cloud-backup コマンドを呼び出す暗号化パラメーターを修正 - 現在のクラスター名と一致しないサーバー名を使用してバックアップオブジェクト ストアから復元するときに barmanObjectStore.serverName オプションを無視しないようにする
- cnpg プラグイン: - ユーザーが指定した場合、プラグインが -n パラメーターに準拠していることを確認します - status コマンドを修正して、結果をソートし、出力のばらつきを取り除きます

4.50 CloudNativePG 1.15 のリリース ノート

CloudNativePG の 1.15 マイナーリリースでユーザーに表示される変更の履歴。

変更の完全なリストについては、 commits を参照してください。

GitHub のリリース ブランチにあります。

> **警告:** バージョン 1.15 はサポート終了 (EOL) になりました。バージョン 1.15.5 は、1.15 マイナー バージョン用にリリースされる最後のバージョンです。

4.50.1 バージョン 1.15.5

リリース日: 2022 年 10 月 6 日

機能強化:

- コントローラー マネージャーに leaseDuration および renewDeadline パラメーターを導入して、オペレーターの展開におけるリーダー選出の構成を強化 (#759)
- 初めて WAL ファイルをアーカイブする前に、バックアップ オブジェクトストアが空であることをチェックするメカニズムを改善します。クラスタがブートストラップされた直後に .check-empty-wal-archive という新しいファイルが PGDATA に配置され、最初の WAL ファイルが実行された後に削除されます。正常にアーカイブされました

安全：

- nonroot:nonroot ユーザーを使用して、distroless/static:nonroot コンテナーイメージにコピーされたインスタンス マネージャーバイナリのアクセス許可を明示的に設定 (#754)

修正:

- クラスターの条件を metav1.Conditions 構造体と互換性を持たせる (#720)

- スタンバイがプライマリに昇格した後、アクティブな接続をドロップします (#737)
- ネイティブの Prometheus エクスポーターで `MAPPEDMETRIC` および `DURATION` メトリック タイプの変換を尊重する (#765)

4.50.2 バージョン 1.15.4

リリース日: 2022 年 9 月 6 日

機能強化:

- PgBouncer 接続プーラーの実装で低レベルのネットワーク TCP 設定の構成を有効にします (#584)
- `cnpg.io/instanceName` および `cnpg.io/podRole` ラベルがポッドおよび PVC に常に存在することを確認します (#632 および#680)
- インスタンスの `role` ラベルを基礎となる PVC に伝播 (#634)

修正:

- オペレーターの複数のインプレース アップグレードプロセスが同時に実行されないようにするには、別のプロセスが進行中かどうかをアトミックにチェックします (#655)。
- 新しくアップロードされたインスタンスマネージャーを格納するためにハードコードされたファイル名を使用しないようにし、オペレーターのオンラインアップグレード中に競合状態が発生するのを防ぎます (#660)
- `GetAllAccessibleDatabases` の呼び出し時にパニックが発生しないように (#641)

4.50.3 バージョン 1.15.3

リリース日: 2022 年 8 月 12 日

機能強化:

- PostgreSQL の `huge_pages` オプションの設定を有効化 (#456)
- フェイルオーバーまたはスイッチオーバー後の昇格および降格中のログを強化し、昇格のリクエストから実際に書き込みが可能になるまでの経過時間を出力する (#371)
- ジョブ、ポッド、および PVC に `instanceName` および `clusterName` ラベルを追加して、これらのリソースとの相互作用を改善 (#534)
- PostGIS クラスターの作成方法に関する説明を追加 (#570)

安全：

- `securityContext` を `Pooler` デプロイメントに明示的に割り当てる (#485)
- 内部 Web サーバーに読み取りタイムアウト値を追加して、Slowloris DDoS を防止 (#437)

修正:

- 再起動 (stopDelay) と切り替え (switchoverDelay) に正しい遅延を使用します。これは、以前に誤って交換されたためです。switchoverDelay が設定されておらず、デフォルト値の 40000000 秒を使用している場合、再起動が無期限にブロックされる可能性があるため、これは重要な修正です (#531)
- クエリがエラーを返したときにメトリクスコレクターがパニックを引き起こさないようにする (#396)
- 場合によってはログレベルに関係なくセグメンテーション違反につながる、未チェックのポインターを参照していた安全でないデバッグメッセージを削除 (#491)
- クラスターにアノテーションがない場合のフェンシング時のパニックを防止 (#512)
- TLS 証明書が変更されていない場合、CRD の更新を回避 (#501)
- CRD に証明書を挿入する際の競合の処理 (#547)
- バックアップと復元: - Google Cloud でオブジェクトストアの資格情報を正しく渡す (#454)

マイナーな変更：

- デフォルトのオペランド イメージを PostgreSQL 15.0 に設定します。

4.50.4 バージョン 1.15.2

リリース日: 2022 年 7 月 7 日 (パッチ リリース)

機能強化:

- スイッチオーバーおよびフェイルオーバー中のインスタンスマネージャーのロギングを改善
- バックアップとリカバリで PostgreSQL 15 を今後サポートするには、Barman >= 3.0.0 が必要です

変更点:

- デフォルトのオペランド イメージを PostgreSQL 15.0 に設定します。

修正:

- PostgreSQL サーバーの CSV ログ パイプを開始する WithActiveInstance 関数内の初期化順序を修正し、クラスターの初期化フェーズで適切なログを確保します- これは、バックアップからの回復などのブートストラップ操作で特に役立ちます (このパッチの前に、そのようなログ標準出力チャネルに送信されず、永久に失われました)
- ホットスタンバイの機密パラメータが減少し、プライマリがすでに再起動されている場合に、不要なスイッチオーバーを回避する
- ALTER ROLE ステートメントでロール名を適切に引用する
- バックアップとリカバリ: - PITR の最も近い Barman バックアップを検出するアルゴリズムを修正します。これは、要求されたリカバリタイムスタンプをバックアップの終了ではなく開始と比較していました。 - トランザクション ID、名前付きの復元ポイント、または recoveryTarget セクションに backupID と呼ばれる新しいフィールドを提供することによる「即時」ターゲット - barman-cloud-wal-archive および barman-cloud-backup コマンドを呼び出す暗号化パラメーターを修正 - 現在のクラスター名と一致しないサーバー名を使用してバックアップオブジェクト ストアから復元するときに barmanObjectStore.serverName オプションを無視しないようにする

- cnpg プラグイン: - ユーザーが指定した場合、プラグインが -n パラメーターに準拠していることを確認します - status コマンドを修正して、結果をソートし、出力のばらつきを取り除きます

4.50.5 バージョン 1.15.1

リリース日: 2022 年 5 月 27 日 (パッチ リリース)

マイナーな変更：

- 以前は固定パラメーターであった PostgreSQL の archive_timeout 設定の構成を有効にします (デフォルトでは 5 分に設定されています)。
- scheduledBackup リソースに backupOwnerReference という新しいフィールドを導入して、作成されたバックアップリソースの所有権参照を設定します。可能な値は、none (デフォルト)、self (スケジュールされたバックアップオブジェクトによって所有されるオブジェクト)、および cluster (Postgres クラスターによって所有される) です。物体）
- ネイティブ Prometheus エクスポーターで PostgreSQL 14 以降の pg_stat_wal メトリックの自動収集を導入
- デフォルトのオペランド イメージを PostgreSQL 15.0 に設定します。

修正:

- postgres に関連する孤立したプロセスを強制終了してフェンシングを修正
- WithActiveInstance 関数内の CSV ログ パイプを有効にして、リカバリブートストラップ ジョブからログを収集し、トラブルシューティングフェーズに役立てます。
- 空でないバックアップ オブジェクトストアを使用して新しいクラスターをブートストラップすることを防止し、既存のバックアップを上書きするリスクを排除します。
- recovery ブートストラップメソッドでは、既存のバックアップを上書きしないように、復旧オブジェクトストアとバックアップ オブジェクトストアが異なることを確認してください。
- バックアップ用の RBAC がまだ作成されていない場合は、調整ループを再度キューに入れます。
- バックアップの問題と、クラスター名プロパティの間違った指定を修正
- 複数のレプリカを使用して高可用性でオペレーターをデプロイする場合、オペレーターポッドが常に最新の証明書を保持するようにします。
- cnpg report operator コマンドを修正して、複数のレプリカを使用して高可用性でオペレーターをデプロイする場合を正しく処理するようにします。
- 再起動を必要とせずに、クラスターの inheritedMetadata セットのラベルと注釈の変更をクラスターの関連リソースに適切に伝達する
- cnpg プラグインを修正して、デフォルト値だけに依存するのではなく、オペレーターのデプロイで定義されたカスタム configmap とシークレット名を正しく解析します。
- mkdocs の minidocks/mkdocs イメージを使用して、ドキュメントのローカル ビルドを修正します。

4.50.6 バージョン 1.15.0

リリース日: 2022 年 4 月 21 日

特徴：

- **フェンシング:** `cnpg.io/fencedInstances` アノテーションを介してクラスターまたは特定の PostgreSQL インスタンスのセットにフェンシング機能を導入します。これは、空でない場合、クラスター内のスイッチオーバー/フェイルオーバーを無効にします。フェンシングされたインスタンスはシャットダウンされ、Pod は検査と緊急事態のために (準備ができていないと見なされている間) 実行され続けます
- **LDAP 認証:** `pg_hba.conf` の LDAP Simple Bind および Search+Bind 構成オプションを Postgres クラスター仕様で宣言的に定義できるようにし、`ldapbindpasswd` などの機密オプションに Kubernetes シークレットをオプションで使用できるようにします。
- 監視されていない `primaryUpdateStrategy` の場合に使用される `switchover` (デフォルト) および `restart` の値を受け入れる `primaryUpdateMethod` オプションの導入。このメソッドは、ローリング更新手順中にプライマリインスタンスに何が起こるかを制御します
- `kubectl cnp` プラグインの新しい `report` コマンドにより、オペレーターと特定の Postgres クラスターの両方のより良い診断とより効果的なトラブルシューティングが可能
- バックアップ オブジェクト ストアに存在しなくなった `Backup` オブジェクトを削除する
- ポイントインタイム リカバリ ブートストラップ方式でのターゲットタイムラインと `LSN` の指定
- `sessionToken` オプションによる AWS S3 での `AWS_SESSION_TOKEN` 認証トークンのサポート
- `quay.io/enterprisedb/pgbouncer:1.17.0` に設定された `Pooler` ポッドの PgBouncer のデフォルトのイメージ名

修正:

- `targetTime` を介したポイントインタイム リカバリのベースバックアップ検出が正しく機能するようになりました。以前は、利用可能な最新のバックアップより前のターゲットは不可能でした (検出アルゴリズムは、開始点として最後のバックアップを選択することで常に間違っていました)。
- オペレーターが `pg_controldata` から収集した値に依存することで、ホットスタンバイの影響を受けやすいパラメーターの回復力を向上
- 無効な証明書エラーをスローする代わりに、無効なエントリを適切に破棄することにより、中間証明書の処理が改善されました
- データベース内の Prometheus エクスポーター メトリック コレクションクエリは、ロールバックではなくコミットされるようになりました (該当する場合、ダウンストリームダッシュボードから表示されるロールバックされたトランザクションの数が変更される可能性があります)。

バージョン 1.15.0 は、CloudNativePG の最初のリリースです。以前は、このソフトウェアは EDB Cloud Native PostgreSQL (現在は EDB Postgres for Kubernetes) と呼ばれていました。以前のリリースに関する情報をお探しの場合は、 EDB documentation を参照してください。

4.51 1.14.0 以前のリリース ノート

CloudNativePG の最初の公開リリースはバージョン 1.15.0 です。それ以前は、この製品は EDB によって完全に所有され、 Cloud Native PostgreSQL という名前で配布されていました。

このページの変更のリストは、情報提供のみを目的としており、コミットに加えて製品の履歴を示しています。**ここにリストされているバージョンは、CloudNativePG には存在しません。**

4.51.1 バージョン 1.14.0

リリース日: 2022 年 3 月 25 日

特徴：

- Barman Cloud 2.19 で導入された機能を利用することで、バックアップとリカバリ用に Google Cloud Storage をネイティブにサポート
- `Cluster` オブジェクトの `LastBackupSucceeded` 条件の導入によるバックアップの可観測性の向上
- ホット スタンバイに影響するパラメーターの更新をサポート: `max_connections` 、`max_prepared_transactions` 、`max_locks_per_transaction` 、`max_wal_senders` 、`max_worker_processes`
- `Cluster` オブジェクトに `Online upgrade in progress` フェーズを追加して、オペレーターのオンラインアップグレードが進行中であることを示します。
- S3 オブジェクト ストレージの資格情報を提供する別の方法として、AWS IAM ロールを継承する機能
- Pooler の authQuerySecret と証明書の不透明な秘密のサポート
- デフォルトの PostgreSQL バージョンを 14.2 に更新
- `maintenance` という名前の `kubectl cnp` プラグインに新しいコマンドを追加して、Kubernetes クラスター全体の 1 つまたはすべての名前空間のクラスターにメンテナンスウィンドウを設定します。

コンテナー イメージ:

- 最新の PostgreSQL コンテナには Barman Cloud 2.19 が含まれています

セキュリティの強化:

- OpenShift を含む、Operator Lifecycle Manager を使用した名前空間の Operator インストールに対する RBAC の強化。 OpenShift ユーザーは、このバージョンに更新することをお勧めします。

修正:

- 元の `pg_control` ファイルのコピーを保持することで、インスタンスマネージャーが中断された `pg_rewind` を再試行できるようにする
- `pg_rewind` を実行する前に古い PID ファイルをクリーンアップする
- 13 より前のバージョンの PostgreSQL でのランダムな再起動を回避するために、`primary_conninfo` のキーによるソートを強制します。
- 調整時に `ServiceAccount` の変更 (ラベル、注釈など) を保持する

- imagePullPolicy デフォルト値の適用を無効にする
- WAL セグメントサイズの initDB 検証を改善
- LSN を指定してクラスターを復旧するときに、`targetLSN` オプションを適切に処理する
- サーバー証明書と CA 証明書の両方で証明書チェーンを許可することにより、カスタム TLS 証明書の検証を修正します。

4.51.2 バージョン 1.13.0

リリース日: 2022 年 2 月 17 日

特徴：

- Snappy 圧縮のサポート。 Snappy は、より低い圧縮率を使用してオブジェクトストアへのアップロード速度を上げるバックアップの高速圧縮オプションです。
- Barman オブジェクトストアにアップロードされたファイルのタグ付けのサポート。この機能には、オペランドイメージに Barman 2.18 が必要です。クラスタ削除後のバックアップ数
- `status.conditions` によるクラスターのステータスの拡張。条件 `ContinuousArchiving` は、クラスターが WAL ファイルのアーカイブを開始したことを示します
- `kubectl` の `cnp` プラグインのステータスコマンドを追加情報で改善: クラスターのステータスを示す `Cluster Summary` セクションと、クラスターで使用される証明書のステータスと有効期限までの残り時間を含む `Certificates Status` セクションを追加
- 新しいクラスターの作成時に空でないバックアップ先を検出する新しい `barman-cloud-check-wal-archive` コマンドをサポート
- `Secret` を使用して、`MONITORING_QUERIES_SECRET` 構成変数を介してデフォルトの監視クエリを追加するためのサポートを追加します。
- ユーザーが AppArmor を使用してコンテナーのアクセス許可を制限できるようにする (AppArmor サポートでデプロイされた Kubernetes クラスター上)
- `kubectl` の `cnp` プラグインに Windows プラットフォームのサポートを追加しました。現在、プラグインは Windows x86 および ARM で使用できます。
- Kubernetes 1.18 および非推奨の API バージョンのサポートを終了

コンテナー イメージ:

- PostgreSQL コンテナには Barman 2.18 が含まれています

セキュリティ修正:

- 所有者とスーパーユーザーのシークレット内のユーザー名フィールドの一貫性チェックを追加します。以前は、悪意のあるユーザーがシークレットを使用して任意の PostgreSQL ユーザーのパスワードを変更できた可能性がありました

修正:

- Postgres ポッドからのコード フェッチ ステータスのメモリ リークを修正

- クラッシュ後の PostgreSQL の自動再起動を無効にします。インスタンスコントローラーは、PostgreSQL インスタンスのライフサイクルを処理します。
- 検証 Webhook で `spec.postgresUID` および `spec.postgresGID` フィールドを変更できないようにします。クラスターの作成後にこれらのフィールドを変更すると、PostgreSQL が起動できなくなります
- バックアップおよび WAL アーカイブ処理コードからログの冗長性を減らします
- まだ初期化されていないときにクラスターが `Healthy` としてマークされるバグを修正
- 非常に高い WAL 生産レートを持つクラスター内のスタンバイサーバーが、整列したらストリーミングに切り替えることができます。
- ほとんどクラッシュしない PostgreSQL ポッドの起動中の競合状態を修正
- クラスター内の最初の PVC の初期化に失敗する可能性がある競合状態を修正します
- レプリカとしてクラスタに参加する前に、降格したばかりのプライマリ Pod の余分な再起動を削除します
- バックアップからの復元時に、レプリケーションに依存する PostgreSQL 構成パラメーターを正しく処理する
- クラスターの作成中に PostgreSQL 構成の検証が欠落していた問題を修正

4.51.3 バージョン 1.12.0

リリース日: 2022 年 1 月 11 日

特徴：

- サポートされている Kubernetes ディストリビューションのリストに Kubernetes 1.23 を追加し、2020 年 12 月に Kubernetes プロジェクトによるサポートを終了した 1.17 のエンドツーエンドテストを削除します。
- 2 秒の接続タイムアウトと 30 秒の通信タイムアウトを追加することで、ネットワークの問題が発生した場合のポッドステータス チェックの応答性を改善します。この変更により、Pod が失敗したと宣言する前に Pod がステータスを報告するのをオペレーターが待機する時間に制限が設定され、フェイルオーバー操作の堅牢性と予測可能性が向上します。
- クラスターに `.spec.inheritedMetadata` フィールドを導入して、ユーザーがクラスターによって生成されたすべてのオブジェクトに適用されるラベルと注釈を指定できるようにします
- インスタンスのステータスを計算するときに実行されるクエリの数を減らす
- PgBouncer の readiness プローブを追加
- Azure プロトコルを使用する場合、Barman のバックアップ オブジェクトストアのエンドポイントのカスタム証明機関のサポートを追加します。

修正:

- フェイルオーバー中は、すべての WAL ストリーミング接続が閉じられるまで、新しいプライマリの選択を待ちます。オペレーターはデフォルトで `wal_sender_timeout` と `wal_receiver_timeout` を 5 秒に設定し、プライマリにネットワークの問題がある場合にスタンバイノードがすぐに気付くようにします。
- レプリカ クラスタ内のすべての PostgreSQL インスタンスに対して「archive_mode」を「always」に設定することにより、レプリカクラスタ内の WAL アーカイブ戦略を変更してローリング更新を修正します。次に、WAL のアップロードを、現在のターゲット指定されたプライマリからのみ制限します。スイッチオーバー中に WAL が 2 回アップロードされることがありますが、これは問題ではありません
- レプリカ クラスター ソース内の Barman のバックアップ オブジェクトストアのエンドポイントのカスタム証明機関のサポートを修正
- クラスタの名前空間でデフォルトのモニタリング構成マップに固定名を使用する
- デフォルトの Webhook が何らかの理由で機能しない場合、オペレーターは調整サイクル中にもクラスターをデフォルトで更新するようになりました
- リソース要求と制限の比較を修正して、すべての調整サイクルですべてのポッドの更新につながるまれな問題を修正します
- Webhook からのログメッセージを改善して、オブジェクトの名前空間も含めるようにします
- すべての調整ループの開始時に「デフォルト」メッセージの記録を停止する
- `enablePodMonitor` が false の場合、調整サイクルごとに PodMonitor 削除のログ記録を停止する
- ポッドに到達できない場合でも、アーキテクチャの不一致の可能性について文句を言わないでください

4.51.4 バージョン 1.11.0

リリース日: 2021 年 12 月 15 日

特徴：

- **並列 WAL アーカイブと復元:** WAL アーカイブ (PostgreSQL の `archive_command` による) と WAL 復元 (`restore_command` による)。並列復元オプションを使用すると、必要な WAL ファイルを取得して順次ではなく並列で再生することにより、新しく昇格したスタンバイをより迅速に準備完了状態にすることができます。
- **監視用のメトリックのデフォルトセット:** `default-monitoring` という名前の新しい `ConfigMap` が、オペレーターの同じ名前空間に自動的にデプロイされ、デフォルトで既存の Postgres クラスターに追加されます。このような動作は、オペレーターの構成で `MONITORING_QUERIES_CONFIGMAP` パラメーターを設定することでグローバルに変更できます。また、`.spec.monitoring.disableDefaultQueries` オプション (デフォルトでは `false` に設定) を使用してクラスターレベルで変更することもできます。
- クラスターの監視セクションに `enablePodMonitor` オプションを導入して、`PodMonitor` リソースを自動的に管理し、Prometheus とシームレスに統合します

- Pod が Kubernetes によって強制終了される前に、目的の `stopDelay` 時間の前半にスマートシャットダウンを実行し、残りの半分に高速シャットダウンを実行することで、PostgreSQL のシャットダウン手順を改善します。
- `switchoverDelay` オプションを追加して、以前のプライマリに与えられた時間を制御して、新しいプライマリを昇格する前に正常にシャットダウンし、すべての WAL ファイルをアーカイブします (デフォルトでは、CloudNativePG はデータの耐久性を優先するために無期限に待機します)。
- ローリング更新を発行して、PostgreSQL `Cluster` のリソース要求と制限の変更を処理する
- `kubectl` の `cnp` プラグインの `status` コマンドを改善し、追加情報を追加: ストリーミングレプリケーションステータス、データベースの合計サイズ、クラスター内のインスタンスの役割
- PostgreSQL の共有メモリ管理用に `dynamic_shared_memory_type` および `shared_memory_type` パラメータの構成を有効にすることで、多数の並列ワーカーを使用するワークロードのサポートを強化します
- クラスターレベルで定義されたラベルと注釈を、ポッドを含む関連リソースに伝達します (削除はサポートされていません)。
- Kubelet によって削除された Pod を自動的に削除します
- 必要に応じてクラスターのローリング更新を発行することにより、`ENABLE_AZURE_PVC_UPDATES` オペレーター構成オプションを使用して、Azure の永続ボリュームの自動サイズ変更を管理します (既定では無効になっています)。
- 特定の Postgres クラスターで `disabled` に設定すると、調整ループが実行されないようにする `cnpg.io/reconciliationLoop` アノテーションを導入する
- `initdb` ブートストラップ メソッドの一部として `postInitApplicationSQL` オプションを導入し、クラスターが作成された直後にメインアプリケーションデータベースでスーパーユーザーとして実行される SQL クエリのリストを指定します。

修正:

- Liveness プローブは、PostgreSQL サーバーの起動プロセスを正しく処理するようになりました。これにより、少数の顧客から報告された問題が修正され、再起動されたスタンバイサーバーが一貫した状態に到達するために WAL ファイルを回復する必要がありますが、ライブネスプローブのタイムアウトが発生する前にそれを行うことができず、ポッドが `CrashLoopBackOff` に残ります。スターテス。
- Liveness プローブは、タイムラインの迂回後に `pg_rewind` を使用して現在のプライマリと再調整する必要がある以前のプライマリのケースを正しく処理するようになりました。これにより、新しいスタンバイのポッドが Kubernetes によって繰り返し強制終了されるのを修正します。
- Postgres ポッド (`Waited for 1.182388649s due to client-side throttling, not priority and fairness, request: GET` など) からのクライアント側のスロットリングを減らします。
- 提供されたものを使用して、OpenShift および OLM インストールで公開鍵基盤 (PKI) の初期化を無効にします。
- 再起動が必要な構成パラメータを変更する場合は、常にプライマリを最後のままにしておきます
- ジョブが正常に完了した後にのみ PVC を準備完了としてマークし、PVC 初期化の競合状態を防ぎます。

- 有効期限が切れた Webhook TLS シークレットを更新するときは、正しい公開キーを使用してください。
- LSN を解析するときのオーバーフローを修正
- 起動時に古い PID ファイルを削除
- `Pooler` リソースが、オペレーターで定義された `imagePullSecret` を継承するようにします (存在する場合)

4.51.5 バージョン 1.10.0

リリース日: 2021 年 11 月 11 日

特徴：

- **PgBouncer による接続プーリング**：`Pooler` リソースとコントローラーを導入して、ローカル PostgreSQL `Cluster` の接続プーラーとして使用される PgBouncer デプロイメントを自動的に管理します。この機能には、TLS クライアント/サーバー接続、パスワード認証、高可用性、ポッドテンプレートのサポート、主要な PgBouncer パラメーターの構成、`PAUSE` /`RESUME` 、JSON 形式でのログ記録、統計、プール、およびリスト用の Prometheus エクスポーターが含まれます。
- **バックアップ保持ポリシー**：バックアップのリカバリウィンドウ保持ポリシーの定義をサポートします (たとえば、30 日のリカバリ ウィンドウを確保するための「30d」)。
- **オペレーターのインプレース更新**：インスタンスマネージャーのインプレースオンライン更新を導入します。これにより、オペレーターの更新後にクラスター全体のローリング更新を実行する必要がなくなります。デフォルトでは、このオプションは無効になっています (documentation for more detailed information を参照してください)。
- `initdb` ブートストラップメソッドでカスタマイズできるオプションのリストを `dataChecksums` 、`encoding` 、`localeCollate` 、`localeCType` 、`walSegmentSize` に制限します。これにより、`options` 配列が廃止され、v2 API で削除される予定です。
- `initdb` ブートストラップ メソッドの一部として `postInitTemplateSQL` オプションを導入し、クラスターが作成された直後にスーパーユーザーとして `template1` データベースで実行される SQL クエリのリストを指定します。この機能により、クラスターで作成されたすべてのアプリケーションデータベースに既定のオブジェクトを含めることができます。
- インスタンス Prometheus エクスポーターに追加された新しいデフォルトメトリック: Postgres バージョン、クラスター名、およびバックアップカタログに基づく回復可能性の最初のポイント
- 失敗後にバックアップを再試行する
- 頻繁に更新されないオペランドイメージには存在しない、最近導入された機能 (保持ポリシーや Azure Blob コンテナー ストレージなど) をオペレーターが呼び出さないようにするために、 Barman Cloud の機能に関する認識を構築します。
- cnp プラグインの `status` コマンドの出力をバックアップに関する情報と統合する
- PVC のステータス (初期化中または準備完了) を報告する新しい注釈を導入します。

- Backup オブジェクトを含む、Cluster で生成されたすべてのオブジェクトの k8s.enterprisedb.io/cluster ラベルにクラスター名を設定します。
- Cluster、Backup、および ScheduledBackup の種類で非推奨の API バージョン postgresql.cnpg.io/v1alpha1 のサポートを削除
- デフォルトのオペランド イメージを PostgreSQL 14.2 に設定

安全：

- オペレーター コンテナー securityContext の allowPrivilegeEscalation を false に設定します。

修正:

- 単一インスタンス クラスタでのメンテナンス中にプライマリ PodDisruptionBudget を無効にする
- 回復操作中に正しい証明書証明機関 (CA) を使用する
- バックアップを取る前に WAL のアーカイブ状態をチェックする際の Postgres 接続のリークを防ぐ
- 一時的なエラーが発生した後、WAL のアーカイブ/復元を 100 ミリ秒スリープ状態にします。

4.51.6 バージョン 1.9.2

リリース日: 2021 年 10 月 15 日

特徴：

- PostgreSQL の archive_command 用の wal-archive とスタンバイの restore_command 用の wal-restore の 2 つの新しいロガーで JSON ログを強化します

修正:

- スタンバイ プロモーション中に WAL アーカイブを有効にします (.history ファイルがアーカイブされないようにします)。
- Barman 2.13 以降を使用する場合にのみ、--cloud-provider オプションを Barman Cloud ツールに渡して、古いオペランドでのエラーを回避します。
- バックアップをトリガーする前に、プライマリのポッドの準備が整うのを待ちます

4.51.7 バージョン 1.9.1

リリース日: 2021 年 9 月 30 日

- このリリースは、 PostgreSQL 14 のリリースを記念するものです

特定のイメージ名を定義せずに、新しい Cluster が作成されたときにデフォルトのメジャー バージョンにすることにより。 *

修正:

- プライマリ ポッドのログに Error while getting barman endpoint CA secret メッセージが表示され、バックアップが正しく機能しない問題を修正

- インスタンスマネージャーとの一時的な通信の問題が発生した場合に備えて、新しいバックアップの要求を適切に再試行します。

4.51.8 バージョン 1.9.0

リリース日: 2021 年 9 月 28 日

- バージョン 1.9.0 は、リリース プロセスとバージョン 1.9.1 のその後のリリースの遅延により、OpenShift では使用できません。*

特徴：

- サポートされている Kubernetes ディストリビューションのリストに Kubernetes 1.22 を追加し、1.16 を削除します。
- 必要に応じて、バックアップ アーカイブから WAL ファイルをフェッチするために、`pg_rewind` で`--restore-target-wal` オプションのサポートを導入します (PostgreSQL 13+ でのみ利用可能)。
- 各データベースの `pg_catalog.pg_largeobject` テーブルのページ数を推定する Prometheus エクスポーターのデフォルトメトリックを公開します。
- ローカルのメモリ内キャッシュを通じて、WAL のアーカイブとフェッチのパフォーマンスを向上させます

修正:

- `pg_isready` の呼び出し時に `postgres` ユーザーを明示的に設定 - OpenShift の制限付き SCC で必要
- ステータスの `FirstRecoverabilityPoint` を適切に更新する
- バックアップが要求された場合、指定されたプライマリに `archive_mode = always` を設定します
- マイナーなバグ修正

4.51.9 バージョン 1.8.0

リリース日: 2021 年 9 月 13 日

特徴：

- 外部クラスタ セクションで定義されたオブジェクトストアから直接、完全リカバリまたはポイントイン タイムリカバリを介して新しいクラスタをブートストラップし、バックアップ CR を定義するという以前の要件を排除します。
- スケジュールされたバックアップに `immediate` オプションを導入して、最初の Postgres インスタンスが実行された直後にバックアップを要求し、ポイントインタイムリカバリが構成されている場合にクラスターの最初に巻き戻す機能を追加します
- クラスタ ステータスに `firstRecoverabilityPoint` を追加して、バックアップ オブジェクトストアのコンテンツに基づいてリカバリを要求する最も古い一貫した時点を報告します

- 次の新しいメトリクスを公開することで、PostgreSQL インスタンスのデフォルトの Prometheus エクスポーターを強化します。

1. ディスク上の WAL ファイルの数と計算された合計サイズ 2. アーカイブステータス フォルダー内の `.ready` および `.done` ファイルの数 3. レプリカ モードのフラグ 4. 要求された最小/最大の同期レプリカの数、および予想および実際に観測された数もの

- Prometheus エクスポータでカスタム メトリックを定義するときに `runonserver` オプションのサポートを追加して、メトリックの収集を PostgreSQL バージョンの範囲に制限します
- Barman Cloud の Barman 2.13 で導入された機能を利用することにより、バックアップと回復のために Azure Blob Storage をネイティブにサポートします。
- liveness プローブを `pg_isready` に依存する
- 復旧対象時刻のタイムスタンプ指定に RFC3339 形式をサポート
- `.spec.imagePullPolicy` を導入して、クラスター用に作成されたすべてのポッドとジョブのイメージコンテナーのプル ポリシーを制御します
- OpenShift 4.5 を置き換える OpenShift 4.8 のサポートを追加
- PostgreSQL 14 (ベータ) をサポート
- ストリーミング接続を必要とせずに、外部クラスターセクションで定義されたオブジェクトストアからクラスター間レプリケーションを使用して、レプリカクラスター機能を強化します (実験的)。
- クラスターの仕様に `logLevel` オプションを導入して、エラー、情報、デバッグ、またはトレースのいずれかのレベルを指定します。

セキュリティの強化:

- `.spec.enableSuperuserAccess` を導入して、パスワード認証を通じて `postgres` ユーザーによるネットワーク アクセスを有効/無効にします。

修正:

- クラスターが回復不能な状態になり、人間の介入が必要になった場合に、ユーザーに適切に通知する

4.51.10 バージョン 1.7.1

リリース日: 2021 年 8 月 11 日

特徴：

- 同期レプリケーションに関して構成よりも自己回復を優先し、クラスターが自己回復操作中に要件を満たすことができない場合にオペレーターが `minSyncReplicas` および `maxSyncReplicas` 設定を一時的にオーバーライドできるようにします。
- `initdb` ブートストラップ メソッドの一部として `postInitSQL` オプションを導入し、クラスターが作成された直後にスーパーユーザーとして実行される SQL クエリのリストを指定します。

修正:

- プライマリの準備ができていない場合にオペレーターがフェイルオーバーできるようにします (1.7.0 で導入されたバグ)。
- LOCAL 同期コミット レベルを使用して管理クエリを実行する
- PGAudit で複数行のログ エントリを正しく解析する

4.51.11 バージョン 1.7.0

リリース日: 2021 年 7 月 28 日

特徴：

- JSON 出力で直接利用できる pgaudit と呼ばれる新しいタイプの logger を使用して、PGAudit にネイティブ サポートを追加します。
- 以下を通じて、監視機能と可観測性機能を強化します。
- pg_stat_statements および auto_explain 拡張機能のネイティブサポート - Prometheus エクスポーターの target_databases オプションは、1 つ以上のデータベースでユーザー定義のメトリッククエリを実行します (シェルのようなパターンマッチングによるデータベースの自動検出を含む) - manual_switchover_required メトリックの公開 primaryUpdateStrategy が supervised に設定されているクラスターで手動切り替えが必要かどうかを迅速に報告する
- pg_audit 、auto_explain 、pg_stat_statements の shared_preload_libraries を透過的に処理
- pg_stat_statements 、pgaudit 、または auto_explain オプションが postgresql パラメーター セクションに追加された場合の PostgreSQL 用の shared_preload_libraries の自動構成
- cnpg.io/reload ラベルをサポートして、構成マップとシークレットの自動リロードを細かく制御します。これには、Prometheus エクスポーターのカスタム モニタリング/アラートメトリックに使用されるものや、証明書を保存するために使用されるものも含まれます。
- reload コマンドを kubectl の cnp プラグインに追加して、インスタンスで調整ループをトリガーします。
- additionalPodAffinity および additionalPodAntiAffinity を介してポッド アフィニティおよび非アフィニティ構成の制御を改善
- 少なくともプライマリインスタンスを常に実行することを要求することにより、プライマリインスタンスに別の PodDisruptionBudget を導入する

セキュリティの強化:

- .spec.certificates.clientCASecret および spec.certificates.replicationTLSSecret オプションを追加して、PostgreSQL サーバーのカスタムクライアント認証局と証明書を定義し、クライアント証明書の認証と PostgreSQL ノード間の安全な通信に使用します。
- .spec.backup.barmanObjectStore.endpointCA オプションを追加して、Barman のバックアップ オブジェクトストアのエンドポイントのカスタム証明機関バンドルを定義します。

修正:

- Prometheus エクスポーターでヒストグラムを正しく解析する

- オペレーターがクラスター用に作成した調整サービス

4.51.12 バージョン 1.6.0

リリース日: 2021 年 7 月 12 日

特徴：

- レプリカ モード (**実験的**): クラスターをソースクラスターのレプリカとして作成できるようにします。レプリカクラスタには、**指定されたプライマリ**と任意の数のスタンバイがあります。
- `.spec.postgresql.promotionTimeout` パラメータを追加して、インスタンスをプライマリに昇格させるときに待機する最大秒数を指定します。デフォルトは 40000000 秒です。
- `.spec.affinity.podAntiAffinityType` パラメータを追加します。`preferred` (デフォルト) に設定すると `preferredDuringSchedulingIgnoredDuringExecution` が使用され、`required` に設定すると `requiredDuringSchedulingIgnoredDuringExecution` になります。

変更点:

- オペレータが新しいポッドを作成するのを妨げていた PVC とポッドを削除するときの競合状態を修正しました。
- マネージャーが構成変更時に PostgreSQL の再起動の必要性を検出できない競合状態を修正しました。
- 注釈のないクラスターでの `kubectl-cnp` のパニックを修正しました。
- 一部のログ メッセージのレベルを `debug` に下げました。
- サーバー CA および TLS インジェクションの E2E テスト。

4.51.13 バージョン 1.5.1

リリース日: 2021 年 6 月 17 日

変化：

- Red Hat OpenShift の Operator Deployment での自動更新を妨げる CRD 検証のバグを修正
- Secret を使用してオペレーターの構成を渡すことを許可します。

4.51.14 バージョン 1.5.0

リリース日: 2021 年 6 月 11 日

特徴：

- `pg_basebackup` ブートストラップ メソッドを導入して、Kubernetes の外部であっても、同じメジャー バージョンの既存の PostgreSQL インスタンスのコピーとして新しい PostgreSQL クラスターを作成します。

- Cluster リソースの Affinity セクションに Kubernetes の容認のサポートを追加し、ユーザーが必要なテイントを使用して Kubernetes ノードに PostgreSQL インスタンスを配布できるようにします
- <image>:<tag>@sha256:<digestValue> 形式を使用して、イメージ名へのダイジェストの指定を有効にし、より決定論的で反復可能な展開を実現します。

セキュリティの強化:

- サーバー証明書とそれに署名した関連認証機関の秘密を定義することにより、TLS 証明書をカスタマイズして PostgreSQL サーバーを認証します。
- require から verify-ca への内部および自動的に管理されるストリーミング レプリカの WAL レシーバー プロセスの sslmode を上げる

変更点:

- kubectl の cnp プラグインの promote サブコマンドを拡張して、ポッドの名前全体ではなくノード番号のみを受け入れるようにします
- クラスター名に DNS-1035 検証スキームを採用 (サービス名はそこから継承されます)
- スタンバイ インスタンスのクローンを作成するとき、または pg_basebackup メソッドを使用してブートストラップするときに、ストリーミングレプリケーション接続を強制します。
- 物理ベースのバックアップに関して、Backup リソースを beginWal 、endWal 、beginLSN 、endLSN 、startedAt 、および stoppedAt と統合します。
- ドキュメンテーションの改善: - オペレーターおよびオペランドコンテナーによって公開されるポートのリストを提供します - データベースワークロード用のストレージと PostgreSQL のベンチマークを行うための cnp-bench Helm チャートとガイドラインを導入します
- E2E テストの機能強化: - Kubernetes 1.21 のテスト - オペレーターの高可用性のテストを追加 - ノード ドレインのテストを追加
- 以下を含むマイナーなバグ修正: - 回復操作中の pg_ctl 開始のタイムアウトが短すぎる - オペレーターが PVC 上の直接イベントを監視していない - barmanObjectStore バックアップの immediateCheckpoint および jobs パラメーターの処理を修正 - バックアップからの回復時にログを空にする

4.51.15 バージョン 1.4.0

リリース日: 2021 年 5 月 18 日

特徴：

- JSON 形式の PostgreSQL エラー メッセージの標準出力ロギング
- Prometheus エクスポータに PostgreSQL メトリクスの基本セットを提供
- restart コマンドを kubectl の cnp プラグインに追加して、特定の PostgreSQL クラスターのポッドをロールアウト方式で再起動します。

セキュリティの強化:

- ポッドの `readOnlyRootFilesystem` セキュリティコンテキストを設定します

変更点:

- **重要:** 以前に YAML マニフェストを使用して CloudNativePG オペレーターをデプロイしたことがある場合は、新しいバージョンをインストールする前に、既存のオペレーターのデプロイを削除する必要があります。これは、オペレーターが直接管理するラベルとラベルセレクターの変更による他の Kubernetes API との競合を避けるために必要です。 1.4.0 へのアップグレードの詳細については、CloudNativePG のドキュメントを参照してください。
- オペレーターによって自動的に定義されるラベルを修正し、名前を `control-plane: controller-manager` から `app.kubernetes.io/name: cloudnative-pg` に変更します
- Prometheus エクスポータの TCP ポートに `metrics` 名を割り当てます。
- `cnp_metrics_exporter` を `application_name` として PostgreSQL のメトリクス エクスポーター接続に設定します。
- 利用可能な場合は、`postgres` データベースの代わりに、Prometheus エクスポーターのクエリを監視するためにアプリケーションデータベースを使用します。
- ドキュメントの改善: - 監視クエリのカスタマイズ - オペレーターのアップグレード手順
- E2E テストの強化
- 以下を含むマイナーなバグ修正: - `pg_basebackup` を呼び出すときに `-R` を使用しないようにする - ステータスを取得するときにエラーログからスタック トレースを削除する

4.51.16 バージョン 1.3.0

リリース日: 2021 年 4 月 23 日

特徴：

- ラベルと注釈の継承
- すべてのコンテナにリソース制限を設定する

セキュリティの強化:

- ポッドの実行を名前空間に割り当てられた UID および SELinux コンテキストに制限するための、Red Hat OpenShift での制限付きセキュリティ コンテキスト制約のサポート
- オペレーターによって明示的に定義された Pod セキュリティコンテキストは、非ルート、非特権、および特権エスカレーションなしで実行されます。

変更点:

- ポート 9187 でリッスンする Prometheus エクスポーター エンドポイント (ポート 8000 は、API サーバーとのインスタンス調整用に予約されています)
- ドキュメントの改善
- GKE 環境を含む E2E テストの機能強化

- マイナーなバグ修正

4.51.17 バージョン 1.2.1

リリース日: 2021 年 4 月 6 日

- ScheduledBackup はバックアップの所有者ではなくなりました。つまり、ScheduledBackup オブジェクトが削除されてもバックアップは削除されません。
- RHSA-2021:1024 を解決するための ubi8-minimal イメージの更新 (セキュリティ アドバイザリ: 重要)

4.51.18 バージョン 1.2.0

リリース日: 2021 年 3 月 31 日

- Prometheus の `postgres_exporter` と互換性のある構文を使用して、ConfigMap および Secret オブジェクトとしてカスタム監視クエリの実験的サポートを導入
- オペレーター ライフサイクル マネージャー (OLM) の展開をサポートし、その後 OperatorHub.io に存在します。
- 米国国防総省 (DoD) の国防情報システム局 (DISA) およびインターネットセキュリティ センター (CIS) からのガイドラインを適用し、Dockle を使用してパイプラインで直接検証することにより、コンテナーのセキュリティを強化します。
- AKS での E2E テストを改善する
- マイナーなバグ修正

4.51.19 バージョン 1.1.0

リリース日: 2021 年 3 月 3 日

- `kubectl cnp status` を追加して、JSON および YAML 出力を含むクラスターのステータスをきれいに印刷します
- `kubectl cnp certificate` を追加して、クライアントアプリケーションの TLS 認証を有効にします
- `-ro` サービスを追加して、接続を利用可能なホット スタンバイレプリカのみにルーティングし、クラスターのプライマリインスタンスからの読み取り専用クエリのオフロードを有効にします。
- クラスターを `maxSyncReplicas` よりも低い値にスケールダウンするロールバック
- 以前のプライマリーを降格する前にチェックポイントを要求する
- `SIGTERM` 上の PostgreSQL プロセスに `SIGINT` シグナル (高速シャットダウン) を送信します。
- マイナーなバグ修正

4.51.20 バージョン 1.0.0

リリース日: 2021 年 2 月 4 日

CloudNativePG の最初の主要な安定版リリースでは、API グループ `postgresql.cnpg.io/v1` に `Cluster`、`Backup`、および `ScheduledBackup` が実装されています。これらのリソースを使用して、次の主な機能を備えた Kubernetes 内で PostgreSQL クラスターを作成および管理します。

- 外部ツールを必要としない、高可用性のための Kubernetes API サーバーとの直接統合
- セルフヒーリング機能: - 最も整合性のとれたレプリカを昇格させることによるプライマリインスタンスのフェイルオーバー - レプリカの自動再作成
- 選択したレプリカの昇格によるプライマリ インスタンスの計画的な切り替え
- スケールアップ/ダウン機能
- 任意の数のインスタンスの定義 (最小 1 - 1 つのプライマリ サーバー)
- アプリケーションをクラスターの唯一のプライマリサーバーに接続するための *read-write* サービスの定義
- アプリケーションを任意のインスタンスに接続してワークロードを読み取るための *read* サービスの定義
- PVC テンプレートを使用したローカル永続ボリュームのサポート
- Pod での永続ボリューム ストレージの再利用
- PostgreSQL のマイナーバージョンとオペレーターのアップグレードのローリング アップデート
- TLS 接続とクライアント証明書認証
- S3 互換オブジェクト ストアへの継続的なバックアップ
- S3 互換オブジェクト ストア バックアップからのフル リカバリとポイントイン タイム リカバリ
- 同期レプリカのサポート
- `nodeSelector` プロパティによるノード アフィニティのサポート
- PostgreSQL エラー メッセージの標準出力ロギング