

CloudNativePG

リリース *1.20.1*

The CloudNativePG Contributors

2023 年 06 月 30 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　主な機能　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 用語 . . . 9
4.1.2 PostgreSQL 用語 . . . 11
4.1.3 クラウド用語 . . . 11
4.1.4 次にすること . . . 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 外部のアプリケーション . . . 13
4.3 アーキテクチャ . . . 13
4.3.1 状態の同期 . . . 13
4.3.2 Kubernetes アーキテクチャ . . . 15
4.3.3 PostgreSQL アーキテクチャ . . . 18
4.3.4 Kubernetes クラスター間のデプロイメント . . . 20
4.4 インストールとアップグレード . . . 22
4.4.1 Kubernetes へのインストール . . . 22
4.4.2 展開の詳細 . . . 23
4.4.3 アップグレード . . . 24
4.5 クイックスタート . . . 27
4.5.1 パート 1：ローカル Kubernetes プレイグラウンドをセットアップする . . . 28
4.5.2 パート 2：CloudNativePG をインストールする . . . 29
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 でクラスターを監視する . . . 31
4.6 ブートストラップ . . . 35
4.6.1 `bootstrap` セクション . . . 35
4.6.2 `externalClusters` セクション . . . 36
4.6.3 空のクラスターをブートストラップする（`initdb`） . . . 37
4.6.4 別のクラスターからのブートストラップ . . . 41
4.7 Postgres データベースのインポート . . . 55
4.7.1 仕組み . . . 56
4.7.2 `microservice` タイプ . . . 57
4.7.3 `monolith` タイプ . . . 59
4.8 セキュリティ . . . 61
4.8.1 コード . . . 61

4.8.2 コンテナ . . . 61
4.8.3 クラスター . . . 62
4.9 Postgres インスタンスマネージャー . . . 67
4.9.1 Liveness と Readiness プローブ . . . 67
4.9.2 シャットダウン制御 . . . 68
4.9.3 フェイルオーバー . . . 69
4.10 スケジューリング . . . 69
4.10.1 Pod アフィニティとアンチアフィニティ . . . 69
4.10.2 nodeSelector によるノード選択 . . . 71
4.10.3 容認 . . . 71
4.11 リソース管理 . . . 71
4.12 故障モード . . . 73
4.12.1 ストレージスペースの使用量 . . . 74
4.12.2 故障モード . . . 74
4.12.3 自己治癒 . . . 76
4.12.4 手動介入 . . . 76
4.13 ローリングアップデート . . . 77
4.13.1 自動更新（unsupervised） . . . 78
4.13.2 手動更新（supervised） . . . 78
4.14 レプリケーション . . . 79
4.14.1 アプリケーションレベルのレプリケーション . . . 79
4.14.2 非常に成熟したテクノロジー . . . 79
4.14.3 ストリーミングレプリケーションのサポート . . . 80
4.14.4 同期レプリケーション . . . 80
4.14.5 高可用性のためのレプリケーションスロット . . . 82
4.15 バックアップとリカバリ . . . 84
4.15.1 クラウドプロバイダーのサポート . . . 85
4.15.2 オンデマンドバックアップ . . . 93
4.15.3 スケジュールされたバックアップ . . . 95
4.15.4 WAL アーカイブ . . . 96
4.15.5 スタンバイからのバックアップ . . . 97
4.15.6 回復 . . . 98
4.15.7 保持ポリシー . . . 100
4.15.8 圧縮アルゴリズム . . . 101
4.15.9 バックアップオブジェクトのタグ付け . . . 101
4.16 PostgreSQL 設定 . . . 102
4.16.1 postgresql セクション . . . 103
4.16.2 pg_hba セクション . . . 109
4.16.3 構成の変更 . . . 110
4.16.4 動的共有メモリの設定 . . . 111
4.16.5 固定パラメーター . . . 112
4.17 データベースロール管理 . . . 114
4.17.1 パスワード管理 . . . 116
4.17.2 実現不可能なロール構成 . . . 118
4.17.3 管理対象ロールのステータス . . . 118
4.18 オペレーター設定 . . . 119
4.18.1 利用可能なオプション . . . 120

4.18.2 オペレーター構成マップの定義 . . . 120
4.18.3 オペレーターシークレットの定義 . . . 121
4.18.4 オペレーターを再起動して構成をリロードする . . . 121
4.18.5 PPROF HTTP サーバー . . . 122
4.19 インスタンスポッド構成 . . . 123
4.19.1 予想されるボリューム . . . 123
4.19.2 環境変数 . . . 125
4.20 ストレージ . . . 126
4.20.1 CloudNativePG のベンチマーク . . . 127
4.20.2 保存時の暗号化 . . . 127
4.20.3 永続ボリュームクレーム . . . 127
4.20.4 ストレージクラスを介した構成 . . . 128
4.20.5 PVC テンプレートによる設定 . . . 128
4.20.6 WAL のボリューム . . . 129
4.20.7 ボリューム拡張 . . . 130
4.20.8 永続ボリュームの静的プロビジョニング . . . 134
4.21 ラベルとアノテーション . . . 134
4.21.1 前提条件 . . . 135
4.21.2 クラスターのメタデータの定義 . . . 135
4.21.3 現在の制限 . . . 136
4.22 モニタリング . . . 136
4.22.1 モニタリングインスタンス . . . 136
4.22.2 オペレーターの監視 . . . 153
4.22.3 エクスポートされたメトリックを検査する方法 . . . 154
4.22.4 補助リソース . . . 155
4.23 ロギング . . . 156
4.23.1 オペレーターログ . . . 156
4.23.2 PostgreSQL ログ . . . 156
4.23.3 PGAudit ログ . . . 157
4.23.4 その他のログ . . . 159
4.24 証明書 . . . 160
4.24.1 オペレーター管理モード . . . 160
4.24.2 ユーザー提供の証明書モード . . . 162
4.25 クライアント TLS / SSL 接続 . . . 172
4.25.1 新しい証明書の発行 . . . 172
4.25.2 TLS 証明書を介した接続のテスト . . . 173
4.26 アプリから接続する . . . 175
4.26.1 DNS 解決 . . . 176
4.26.2 環境変数 . . . 176
4.26.3 秘密 . . . 177
4.27 コネクションプーリング . . . 177
4.27.1 アーキテクチャ . . . 177
4.27.2 クイックスタート . . . 177
4.27.3 プーラーリソースのライフサイクル . . . 179
4.27.4 セキュリティ . . . 180
4.27.5 認証 . . . 180
4.27.6 ポッドテンプレート . . . 181

4.27.7 高可用性（HA） . . . 183
4.27.8 PgBouncer 構成オプション . . . 183
4.27.9 モニタリング . . . 185
4.27.10 ロギング . . . 192
4.27.11 接続の一時停止 . . . 192
4.27.12 制限事項 . . . 193
4.28 レプリカクラスター . . . 193
4.28.1 基本概念 . . . 194
4.28.2 レプリカクラスターのセットアップ . . . 195
4.28.3 レプリカクラスター内の指定されたプライマリの昇格 . . . 197
4.29 Kubernetes アップグレード . . . 198
4.29.1 `reusePVC` が `false` に設定されたシングルインスタンスクラスター . . . 199
4.30 Postgres サービスの公開 . . . 200
4.30.1 Minikube でのテスト . . . 203
4.31 CloudNativePG プラグイン . . . 203
4.31.1 インストールする . . . 203
4.31.2 使用する . . . 206
4.32 自動フェイルオーバー . . . 221
4.32.1 RTO と RPO への影響 . . . 222
4.32.2 遅延フェールオーバー . . . 223
4.33 トラブルシューティング . . . 223
4.33.1 始める前に . . . 223
4.33.2 緊急バックアップ . . . 224
4.33.3 オペレーター情報 . . . 226
4.33.4 クラスター情報 . . . 227
4.33.5 ポッド情報 . . . 228
4.33.6 バックアップ情報 . . . 231
4.33.7 ストレージ情報 . . . 231
4.33.8 ノード情報 . . . 231
4.33.9 条件 . . . 232
4.33.10 ネットワーキング . . . 233
4.33.11 いくつかの一般的な問題 . . . 233
4.34 フェンシング . . . 236
4.34.1 インスタンスをフェンスする方法 . . . 236
4.34.2 フェンシングを持ち上げる方法 . . . 237
4.34.3 フェンシングの仕組み . . . 237
4.35 宣言的ハイバーネーション . . . 238
4.35.1 ハイバネーション . . . 238
4.35.2 水分補給 . . . 239
4.36 PostGIS . . . 239
4.36.1 PostGIS クラスターに関する基本概念 . . . 240
4.36.2 PostGIS で新しい PostgreSQL クラスターを作成する . . . 240
4.37 エンドツーエンドのテスト . . . 243
4.38 コンテナイメージの要件 . . . 245
4.38.1 イメージタグの要件 . . . 245
4.39 オペレーターの能力レベル . . . 246
4.39.1 レベル 1：基本インストール . . . 247

4.39.2 レベル 2：シームレスなアップグレード . . . 252
4.39.3 レベル 3：フルライフサイクル . . . 253
4.39.4 レベル 4：深い洞察 . . . 257
4.39.5 レベル 5：オートパイロット . . . 258
4.40 カスタムポッドコントローラー . . . 259
4.40.1 PVC リサイズ . . . 259
4.40.2 プライマリインスタンスとレプリカ . . . 259
4.40.3 PVC のコヒーレンス . . . 260
4.40.4 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 . . . 260
4.41 例 . . . 261
4.42 ネットワーキング . . . 276
4.42.1 オペレーターのクロスネームスペースネットワークポリシー . . . 276
4.42.2 クロスクラスターネットワーキング . . . 277
4.43 ベンチマーク . . . 277
4.43.1 pgbench . . . 278
4.43.2 fio . . . 279
4.44 商用サポート . . . 282
4.45 よくある質問（FAQ） . . . 282
4.45.1 Kubernetes で PostgreSQL を実行する . . . 282
4.45.2 高可用性 . . . 285
4.46 API リファレンス . . . 285
4.46.1 AffinityConfiguration . . . 288
4.46.2 AzureCredentials . . . 289
4.46.3 バックアップ . . . 290
4.46.4 BackupConfiguration . . . 290
4.46.5 バックアップリスト . . . 291
4.46.6 バックアップソース . . . 291
4.46.7 BackupSpec . . . 291
4.46.8 BackupStatus . . . 292
4.46.9 BarmanCredentials . . . 293
4.46.10 BarmanObjectStoreConfiguration . . . 293
4.46.11 BootstrapConfiguration . . . 294
4.46.12 BootstrapInitDB . . . 294
4.46.13 BootstrapPgBaseBackup . . . 296
4.46.14 BootstrapRecovery . . . 296
4.46.15 CertificatesConfiguration . . . 297
4.46.16 CertificatesStatus . . . 299
4.46.17 クラスター . . . 299
4.46.18 ClusterList . . . 299
4.46.19 ClusterSpec . . . 300
4.46.20 ClusterStatus . . . 303
4.46.21 ConfigMapKeySelector . . . 304
4.46.22 ConfigMapResourceVersion . . . 305
4.46.23 DataBackupConfiguration . . . 305
4.46.24 データソース . . . 306
4.46.25 EmbeddedObjectMetadata . . . 306
4.46.26 ExternalCluster . . . 306

4.46.27 Google 資格情報 . . . 307
4.46.28 インポート . . . 307
4.46.29 ImportSource . . . 307
4.46.30 インスタンス ID . . . 308
4.46.31 InstanceReportedState . . . 308
4.46.32 LDAPBindAsAuth . . . 308
4.46.33 LDAPBindSearchAuth . . . 309
4.46.34 LDAPConfig . . . 309
4.46.35 LocalObjectReference . . . 309
4.46.36 ManagedConfiguration . . . 310
4.46.37 管理対象ロール . . . 310
4.46.38 メタデータ . . . 310
4.46.39 MonitoringConfiguration . . . 311
4.46.40 NodeMaintenanceWindow . . . 311
4.46.41 PasswordState . . . 311
4.46.42 PgBouncerIntegrationStatus . . . 312
4.46.43 PgBouncerSecrets . . . 312
4.46.44 PgBouncerSpec . . . 312
4.46.45 PodTemplateSpec . . . 313
4.46.46 プーラー . . . 313
4.46.47 PoolerIntegrations . . . 314
4.46.48 PoolerList . . . 314
4.46.49 PoolerMonitoringConfiguration . . . 314
4.46.50 PoolerSecrets . . . 314
4.46.51 PoolerSpec . . . 315
4.46.52 PoolerStatus . . . 315
4.46.53 PostInitApplicationSQLRefs . . . 315
4.46.54 PostgresConfiguration . . . 316
4.46.55 RecoveryTarget . . . 316
4.46.56 ReplicaClusterConfiguration . . . 317
4.46.57 ReplicationSlotsConfiguration . . . 317
4.46.58 ReplicationSlotsHAConfiguration . . . 318
4.46.59 RoleConfiguration . . . 318
4.46.60 RollingUpdateStatus . . . 320
4.46.61 S3 資格情報 . . . 320
4.46.62 スケジュールされたバックアップ . . . 321
4.46.63 ScheduledBackupList . . . 321
4.46.64 ScheduledBackupSpec . . . 321
4.46.65 ScheduledBackupStatus . . . 322
4.46.66 SecretKeySelector . . . 323
4.46.67 SecretVersion . . . 323
4.46.68 SecretsResourceVersion . . . 323
4.46.69 ServiceAccountTemplate . . . 324
4.46.70 StorageConfiguration . . . 324
4.46.71 SyncReplicaElectionConstraints . . . 325
4.46.72 トポロジー . . . 325
4.46.73 WalBackupConfiguration . . . 326

4.47 サポートされているリリース . . . 326
4.47.1 サポートポリシー . . . 327
4.47.2 ネーミングスキーム . . . 328
4.47.3 CloudNativePG リリースのサポート状況 . . . 328
4.47.4 今後のリリース . . . 329
4.47.5 古いリリース . . . 329
4.47.6 サポートとは . . . 329
4.48 リリースノート . . . 330
4.48.1 CloudNativePG 1.20 のリリースノート . . . 331
4.48.2 CloudNativePG 1.19 のリリースノート . . . 332
4.48.3 CloudNativePG 1.18 のリリースノート . . . 336
4.48.4 CloudNativePG 1.17 のリリースノート . . . 342
4.48.5 CloudNativePG 1.16 のリリースノート . . . 347
4.48.6 CloudNativePG 1.15 のリリースノート . . . 352
4.48.7 1.14.0 以前のリリースノート . . . 356

CloudNativePG はオープンソースの **オペレーター管理モード** です

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 *Kubernetes* **アーキテクチャ** クラスターで *PostgreSQL* **アーキテクチャ** ワークロードを管理するように設計されています。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、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 `Cluster` と呼ばれる新しい Kubernetes リソースを定義します。

同じ Kubernetes クラスター内にあるアプリケーションは、フェー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を心配することなく、オペレーターのみが管理す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。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 に基づいたネイティブの接続プーラーを利用できます。

CloudNativePG はもともと **スケジュールされたバックアップ** によってビルドされ、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 に提出しました。 source code repository is in Github 。

注釈: *Operator Capability Levels model* に基づいて、ユーザーは CloudNativePG Operator からの**「レベル V - オートパイロット」**セット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、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** のページをご覧ください。

第2章　コンテナイメージ

> CloudNativePG community は、演算子とオペランドの両方、つまり PostgreSQL のコンテナイメージを維持します。

CloudNativePG オペレーターのコンテナイメージは distroless です

> cloudnative-pg で利用できます。

PostgreSQL オペランドコンテナイメージは、 postgres-containers から直接、マルチプルのアーキテクチャ上のすべての PGDG supported versions of PostgreSQL で利用できます。

さらに、コミュニティは *PostGIS extension* の画像を提供します。

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つ必要があるため、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　主な機能

- 外部ツールを必要とせずに、高可用性のための Kubernetes API サーバーと直接統合
- 自己回復機能、以下：* 最も整列されたレプリカをプロモートす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スイッチオーバー

*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読み取り-書き込みサービスの定義

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` を介した特定の一般的な Postgres 拡張機能を含む、PostgreSQL 構成の宣言的管理：`pgaudit`、`auto_explain`、`pg_stat_statements`、および `pg_failover_slots`

- Postgres のロール、ユーザー、グループの宣言的管理
- 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 ファイル別冊

- 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）

*オブジェクトストア内の既存のバックアップからの完全復旧とポイントインタイム復旧

- 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
- PostgreSQL クラスター全体、または宣言的な方法でのインスタンスのサブセットのフェンシング
- 宣言的な方法での PostgreSQL クラスターの休止
- 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

*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）

- パラレル WAL アーカイブと復元により、データベースが高書き込みシステムでの 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て、オブジェクトストアレイヤーでのオプションの保持管理を有効にするためのレプリカクラスター

- プライベート、パブリック、ハイブリッド、マルチクラウドの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
- PgBouncer による接続プーリング
- `nodeSelector` を介したノードアフィニティのサポート
- `metrics` ポートを介した 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（9187）
- JSON 形式の PostgreSQL エラーメッセージの標準出力ロギング
- Pod 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

*シンプルバインドと検索+バインド LDAP クライアント認証

- マルチアーチフォーマットのコンテナイメージ

注釈: CloudNativePG は、データの永続性を管理するために `StatefulSet` s を使用しません。むしろ、永続的なボリュームクレーム（PVC）を直接管理します。興味がある場合は、 **カスタムポッドコントローラー** を読んで詳細を確認してください。

第4章　このガイドについて

クイックスタート の指示に従って、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 の基本的な用語に精通していない場合は、
始める前に を参照してください。

- Postgres, PostgreSQL and the Slonik Logo

は、カナダ PostgreSQL Community Association の商標または登録商標であり、彼ら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 に固有の用語を確認することが不可欠です。

4.1.1 Kubernetes 用語

NodeMaintenanceWindow

：**ノード**は、仮想または物理の Kubernetes のワーカーマシンであり、ポッドの実行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。

PodTemplateSpec

：**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 つ以上のコンテナで構成されています。

ServiceAccountTemplate

：**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、ロードバランシング、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。

PgBouncerSecrets

：**シークレット**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。

ストレージクラスを介した構成

：**ストレージクラス**により、管理者は、プロビジョナー（AWS EBS など）、再生ポリシー、マウントオプション、ボリューム拡張などを含むクラスター内のストレージのクラスを定義できます。

Pod での永続ボリュームストレージの再利用

：**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**永続的なボリューム要求**を使用してポッドに関連付けられ、そのライフサイクルは、それを使用するポッド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。

永続ボリュームクレーム

：**永続ボリュームクレーム**（PVC）は、ストレージの要求を表します。これには、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方法と同様に、PVC は PV のリソースを消費します。

オペレーターのクロスネームスペースネットワークポリシー

：**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分離された環境を作成できます。

ロールベースのアクセス制御（*RBAC*）

：**ロールベースのアクセス制御**（RBAC）は、**ロールベースのセキュリティ**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、システムのネットワークとリソースへのアクセスを承認されたユーザーのみに制限する方法です。Kubernetes には、名前空間とクラスターレベルでロールを制御し、それらを特定のリソースや個人に関連付けるためのネイティブ API があります。

CRD を介したオペランド画像のオーバーライド

：**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。

オペレーター管理モード

：**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ときに人間のオペレーターによって通常実行されるステップ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監視された状態と常に一致することを確認する際に 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** ページをご覧ください。

4.1.2 PostgreSQL 用語

インスタンス：実行され、「IP アドレス」と「TCP ポート」（通常は 5432）のペアでリッスンする 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。

レプリカ：クラスター内の唯一のプライマリインスタンスから複製する 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続けます。レプリカは、**スタンバイ**または**セカンダリ**サーバーとしても知られています。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。

ホットスタンバイ：**レプリカ**が読み取り専用ワークロードを受け入れることを可能にする PostgreSQL 機能。

クラスター：高可用性（HA）クラスターとして意図されるもの：単一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 PostgreSQL インスタンスのセット。

レプリカクラスター: 選択した PostgreSQL クラスターからの継続的復旧モードである CloudNativePG `Cluster` 。通常は Kubernetes クラスターの外部にあります。これは、プライベート、パブリック、ハイブリッド、マルチクラウドのコンテキストでマルチクラスターデプロイを可能にする機能です。

Designated Primary ：別の PostgreSQL クラスターからの継続的リカバリ中であり、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。

スーパーユーザー ：PostgreSQL では、 **スーパーユーザー**は `LOGIN` および `SUPERUSER` 特権の両方を持つロールです。セキュリティ上の理由から、CloudNativePG は、ローカル Unix ドメインソケットを介した `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。

WalBackupConfiguration

： Write-Ahead Logging（WAL）は、データベース管理システムのデータ整合性を保証するための標準的な方法です。

PVC グループ：CloudNativePG の用語の PVC グループは、同じ PostgreSQL インスタンス、つまり PGDATA（`storage` ）を含むメインボリュームと WAL のボリューム（`walStorage` ）に属する関連する PVC のグループです。|

4.1.3 クラウド用語

リージョン：クラウド内の**リージョン**は、**アベイラビリティーゾーン**で編成された分離され独立した地理的エリアです。リージョン内のゾーンでは、ラウンドトリップネットワークレイテンシーがほとんどありません。

ゾーン：クラウド（**ゾーン**とも呼ばれます）の**アベイラビリティーゾーン**は、リソースをデプロイできるリージョン内のエリアです。通常、アベイラビリティーゾーンは、データセンターまたは同じデータセンターの隔離された建物に対応します。

4.1.4 次にすること

用語に精通したので、選択したクラウド環境にオペレーターを展開する前に *test CloudNativePG on your laptop using a local cluster* することに決定できます。

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 クラスターにあ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化できず、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。

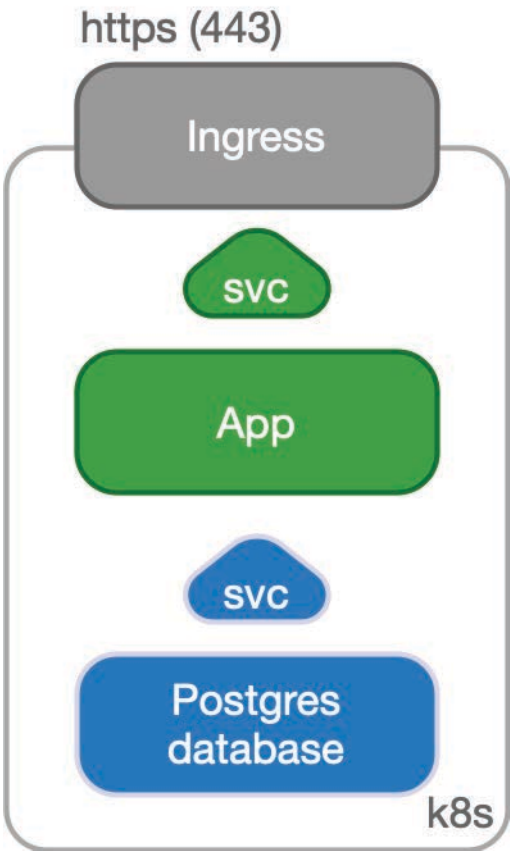


図 1: Application and Database inside Kubernetes

アプリケーションは、通常はステートレスで、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` サービスを介して内部的に公開されます。

このサービスは、Ingress およびプロバイダーのロードバランサー機能、HTTPS を介してエンドユーザーに外部に公開されます。

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、信頼性が高く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取り/書き込みサービスを参照します。Cluster リソースには、単一のプライマリおよびマルチプルスタンバイアーキテクチャのロジックが埋め込まれ、Postgres での高可用性クラスター管理の複雑さが隠されています。

4.2.2 ケース 2：Kubernetes 外部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外部（仮想化環境など）で管理することです。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 接続のメリットを引き続き享受できます。

4.3 アーキテクチャ

このセクションでは、Kubernetes に PostgreSQL をデプロイするときに考慮する必要があるアーキテクチャの主な側面について説明します。

重要: セルフマネージドの 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の計画を開始するときは、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**ステートフルなワークロード**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）とディザスターリカバリー（DR）を達成しますが、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、状態を複数の場所に、好ましくは連続的かつ瞬時に複製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 は、単純な理由で、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ます。PostgreSQL データベース管理システムには、堅牢で信頼性の高いビルトイン**物理レプリケーション**機能が搭載されています。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および同期ストリーミングレプリケーションの両方と、非同期ファイルベースのログシッピング（通常、フォールバックオプションとして、たとえば、WAL ファイ

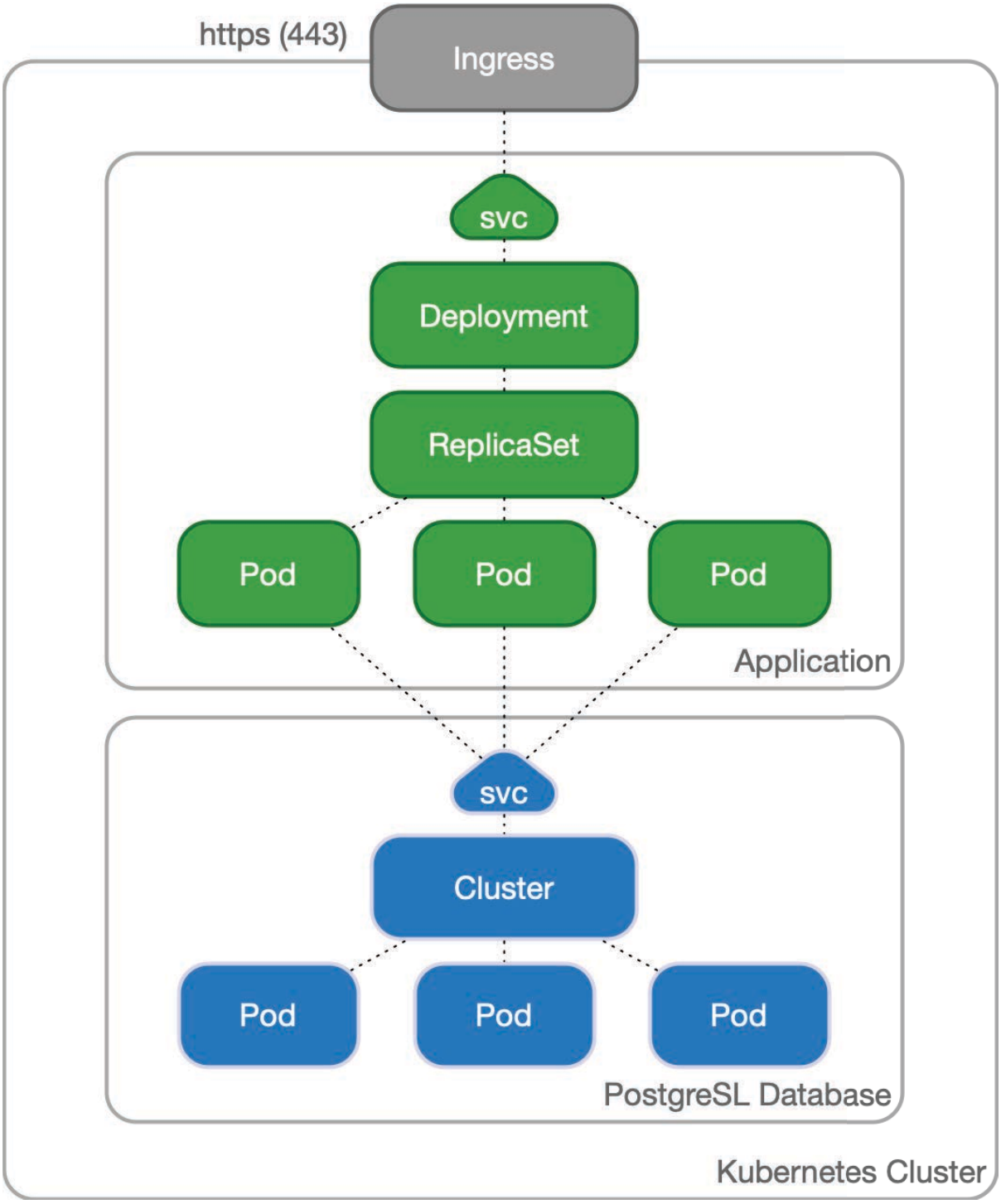


図 2: Close-up view of application and database inside Kubernetes

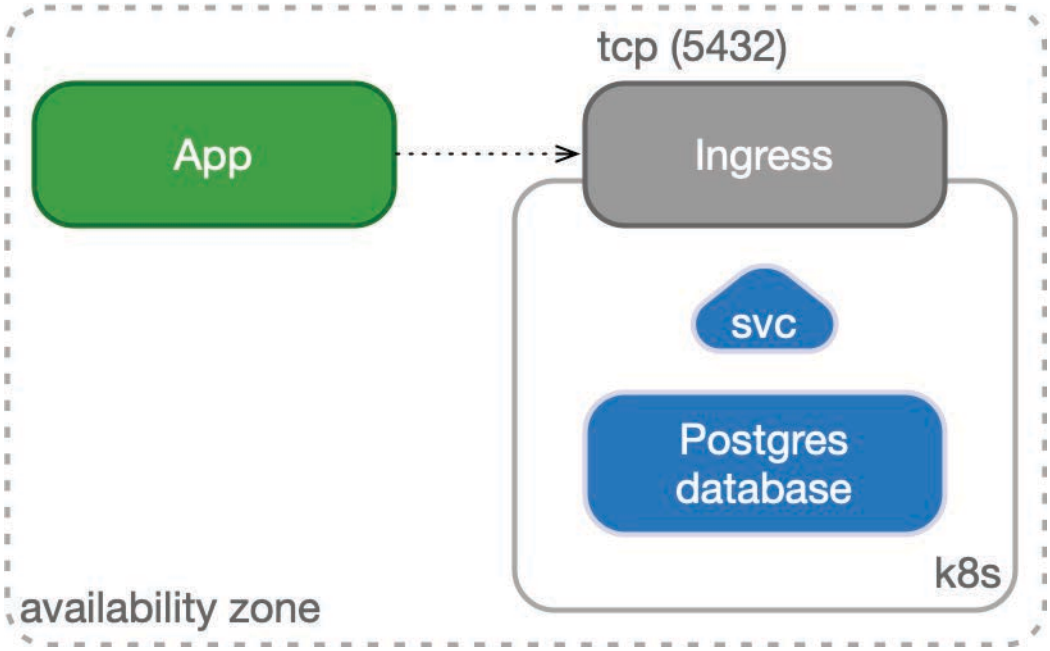


図 3: Application outside Kubernetes

ルをオブジェクトストアに保存するために使用されます）をサポートします。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 Data On Kubernetes, Deploying And Running PostgreSQL And Patterns For Databases In a Kubernetes Cluster と題された講演を参照してください。

このトピックが詳細に説明された場所。

4.3.2 Kubernetes アーキテクチャ

Kubernetes はネイティブに、冗長で低レイテンシーのプライベートネットワーク接続を介して互いに接続された、離れた物理的な場所（データセンター、障害ゾーン、またはより頻繁に**アベイラビリティーゾーン**とも呼ばれます）にまたがる可能性を提供します。

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 を参照ください。これは、

各データセンターがいつでもアクティブであり、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、すでに各リージョンに 3 つ以上のアベイラビリティーゾーンを提供しています。

マルチアベイラビリティーゾーン Kubernetes クラスター

3 つ以上のゾーンを持つマルチアベイラビリティーゾーン Kubernetes アーキテクチャは、PostgreSQL の使用をお勧めします。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。

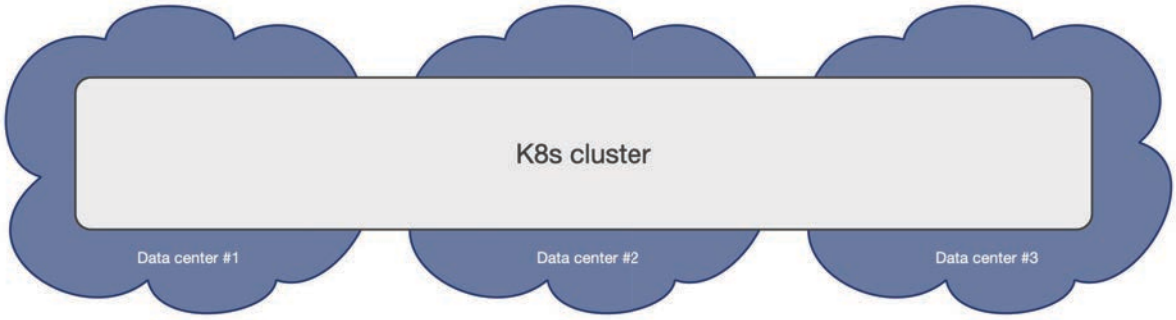


図 4: Kubernetes cluster spanning over 3 independent data centers

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ー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` リソースのライフサイクル全体を制御できます。これには、他のオペレーションの中でも、宣言的な方法で **スケジューリング** ワークロード（ベースアフィニティルール、容認、ノードセレクター）、自動フェイルオーバー、自己修復、および更新。すべては、単一の Kubernetes クラスター内のゾーン全体でシームレスに機能します。

> *PostgreSQL* **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ーゾーンレベルでのシェアードナッシングデプロイメントを介して、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。

さらに、追加の **マルチアベイラビリティーゾーン** *Kubernetes* **クラスター**を利用して、それらを使用して「パッシブ」PostgreSQL レプリカクラスターをホストできます。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が、これは主に DR、読み取り専用オペレーション、またはクロスリージョンの可用性のために使用する必要があります。

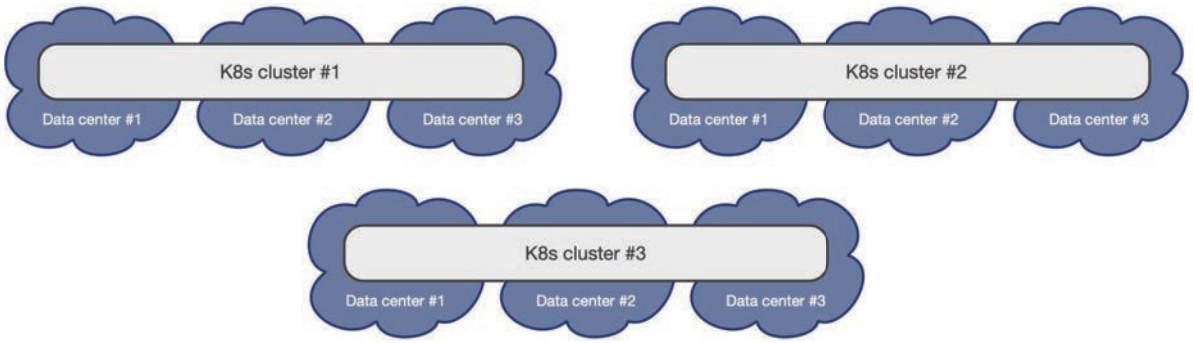


図 5: Example of a multiple Kubernetes cluster architecture distributed over 3 regions each with 3 independent data centers

単一のアベイラビリティーゾーン Kubernetes クラスター

Kubernetes クラスターに 1 つのアベイラビリティーゾーンしかない場合、CloudNativePG は、PostgreSQL データベースの HA と DR の結果を向上させるための多くの機能を引き続き提供し、単一障害点（SPoF）を可能な限りゾーンのレベルにプッシュします-つまり、CloudNativePG クラスターに障害が発生する前に、ゾーンが停止している必要があります。

このシナリオは、1 つのデータセンターのみを使用できるセルフマネージドのオンプレミス Kubernetes クラスターの一般的なシナリオです。

シングルアベイラビリティーゾーン Kubernetes は、残念ながら、低レイテンシー接続（通常は同じ大都市圏）の範囲内で **2 つのデータセンター**のみを利用できる唯一の実行可能なオプションです。2 つのゾーンしかないため、ユーザーはマルチ-アベイラビリティーゾーン Kubernetes クラスター（3 ゾーンの最小数に達していないため）し、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ます。2 番目のクラスターは主にディザスターリカバリーに使用されます。

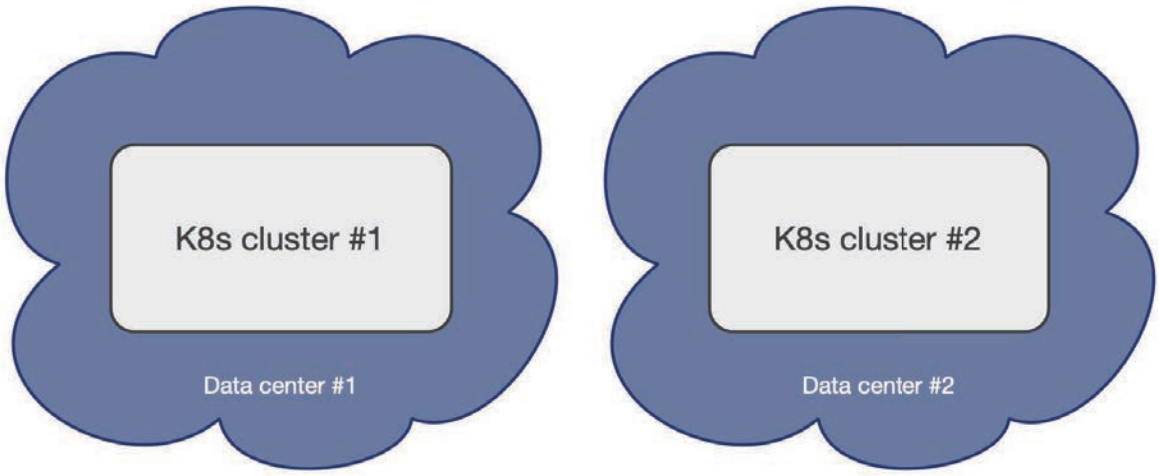


図 6: Example of a Kubernetes architecture with only 2 data centers

---

ヒント:

> Kubernetes ジャーニーの初期段階にいる場合は、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が設計されたときに対処されました。最終的に、他の 2 つに接続された 3 番目の物理的な場所は、組織が検討するための有効なオプションを表す場合があります。

PostgreSQL アーキテクチャ を参照してください

ストレージおよびワーカーノードレベルでのみシェアードナッシングデプロイメントを介して、単一のアベイラビリティーゾーン 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。HA の場合、このようなシナリオでは、PostgreSQL インスタンスを異なるワーカーノードに配置し、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ーゾーン *Kubernetes* クラスター 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、SPoF をシングルゾーンより上にプッシュできます。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 クラスターのプロモーション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 クラスター内でのみ作業できるため、現時点では CloudNativePG を使用した PostgreSQL で Kubernetes クラスター間の自動フェールオーバーを利用できません。

4.3.3 PostgreSQL アーキテクチャ

CloudNativePG は、次の仕様で、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いたクラスターをサポートして、同じ Kubernetes クラスター内の複数のホットスタンバイレプリカを管理します。

- 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 * `-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* `-r` ：アプリケーションは読み取り専用ワークロードのインスタンスのいずれかに接続
- PostgreSQL クラスターの復元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-結果として、インスタンスはストレージを共有せず、ノードに接続されたローカルボリュームを使用することをお勧めしますで実行* PostgreSQL インスタンスは、同じ Kubernetes クラスター/リージョン内の異なるアベイラビリティーゾーンに存在する必要があります

次の図は、3 つの異なるアベイラビリティーゾーンにまたがり、それぞれが PostgreSQL データ用の専用ローカルストレージを備えた別々のノードで実行されている PostgreSQL クラスターに推奨されるシェアードナッシングアーキテクチャの単純なビューを提供します。

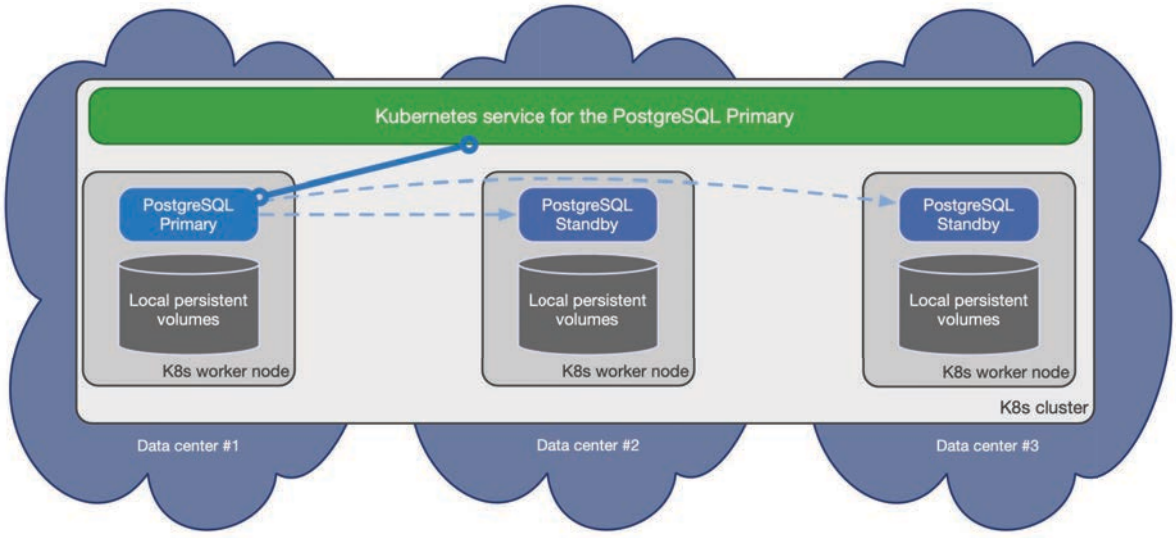


図 7: Bird-eye view of the recommended shared nothing architecture for PostgreSQL in Kubernetes

クラスター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、昇格したプライマリを指すように`-rw` サービスを自動的に更新し、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。

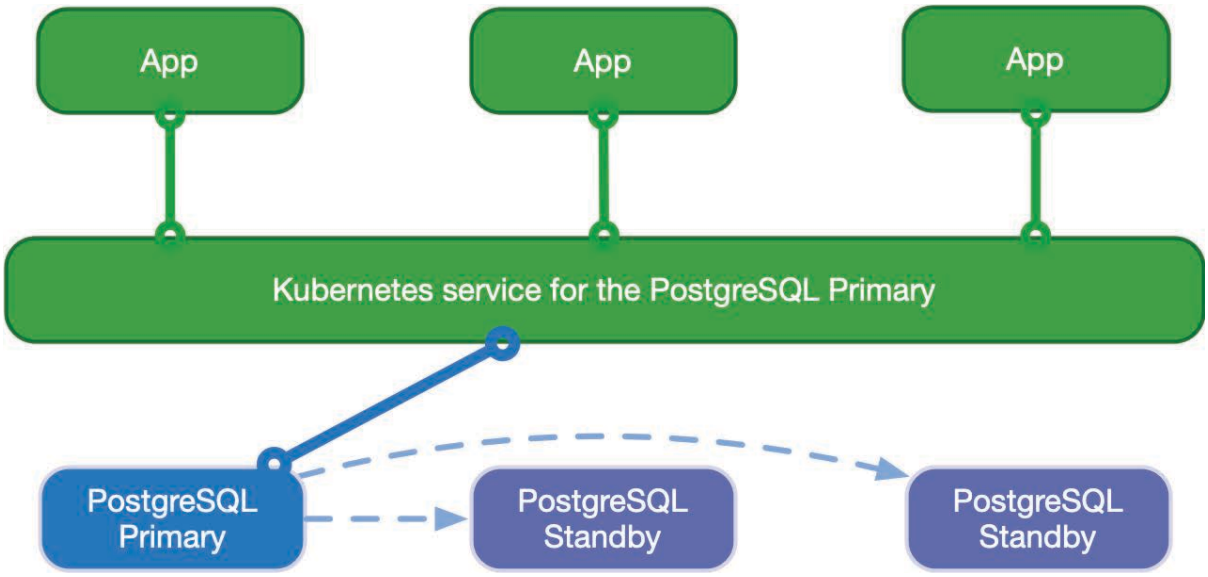


図 8: Applications writing to the single primary

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、高可用性のために 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

これらのワークロードを処理する際の PostgreSQL の動作を提示し、精通しています。

アプリケーションは、オペレーターが利用可能にした`-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。

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。

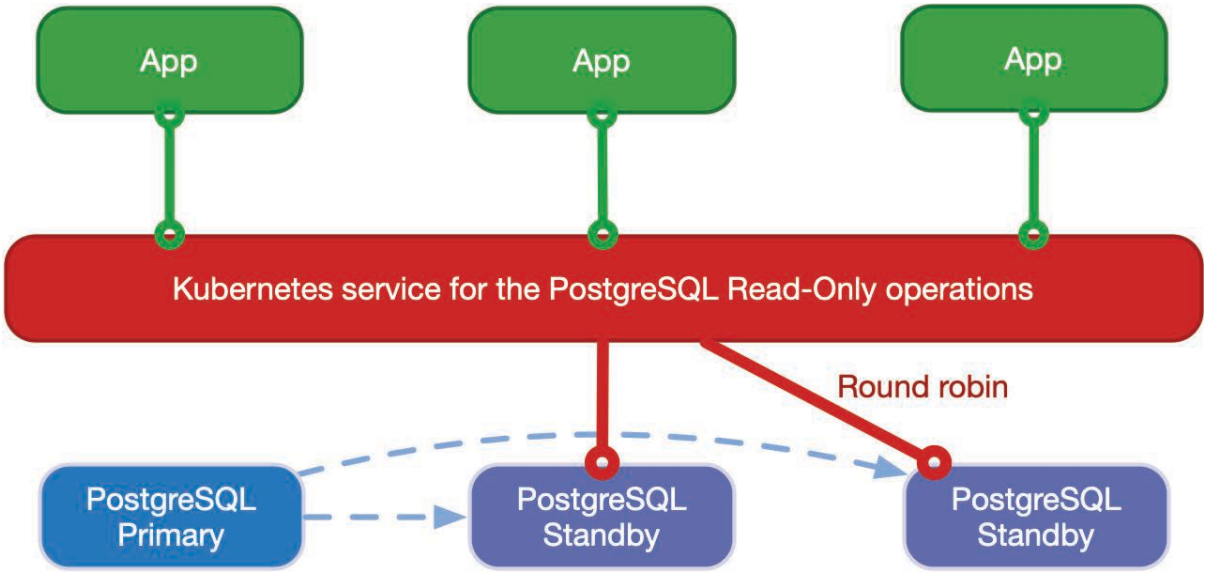


図 9: Applications reading from hot standby replicas in round robin

4.3.4 Kubernetes クラスター間のデプロイメント

注釈: CloudNativePG は、このセクションで説明する**レプリカクラスター**と呼ばれる機能を介して、複数の Kubernetes クラスターにわたる PostgreSQL のデプロイをサポートしています。

分散 PostgreSQL クラスターでは、常にプライマリとして機能する 1 つの PostgreSQL インスタンスのみが存在できます。これは、アプリケーションがいつでも単一の Kubernetes クラスター内にのみ書き込むことができることを意味します。

ただし、事業継続性の目標には、次のことが基本です。

- 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により、グローバルな**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 クラスターを超えて PostgreSQL レプリケーションを活用することにより、グローバルな**目標復旧時間**（RTO）を削減（高可用性）

上記の懸念に対処するために、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、マルチクラウドのコンテキストでマルチクラスターデプロイを有効にする 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ーからの `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` で識別される定義された外部クラスターからの複製、通常は Kubernetes クラスターの外部にあります

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` パラメーターを使用）、またはストリーミングレプリケーション（PostgreSQL の `primary_conninfo` パラメーターを使用）、または 2 つのいずれか（`barmanObjectStore` と `connectionParameters` の両方が外部クラスター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、読み取り接続のみを受け入れる

参考:

別の PostgreSQL クラスター（ *externalClusters* セクションで定義）からのクローン作成の詳細については、*BootstrapConfiguration* を参照してください。

次の図は、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 Kubernetes クラスターが利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。

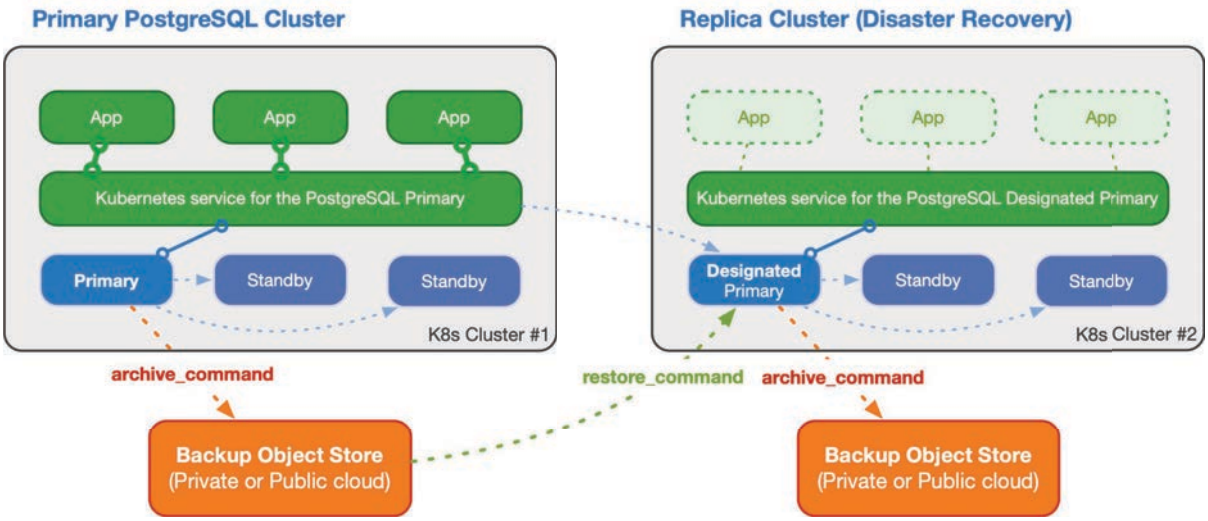


図 10: An example of multi-cluster deployment with a primary and a replica cluster

レプリカクラスターは、プライマリクラスター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、レプリカクラスターを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ーにします。

警告: CloudNativePG は、現時点ではクラスター間のスイッチオーバーまたはフェイルオーバーを実行しません。このような操作は、手動で実行するか、マルチクラスター/フェデレーテッド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、互いに独立しています。

上記の例で指定されたプライマリは、WAL ストリーミング（`primary_conninfo` ）を介して供給され、`restore_command` および `barman-cloud-wal-restore` を介したファイルベースの WAL シッピングのフォールバックオプションがあります。

CloudNativePG では、複数のレプリカクラスターを定義できます。より少ない数のレプリカで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

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 内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とおりです。

```
kubectl apply -f \
  https://raw.githubusercontent.com/cloudnative-pg/cloudnative-pg/release-1.20/
↪releases/cnpg-1.20.1.yaml
```

次の方法でそれを確認できます。

```
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デフォルト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、ウォッチ名前空間を特定の名前空間のみになるように変更するには、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>` ドキュメントを参照してください。

> **警告:** GKE に CloudNativePG をデプロイしていてエラー（*... failed to call webhook...* ）が発生した場合、デフォルトで、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、公式の docs で説明されているように、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

そしてこれによって いくつかの一般的な問題 。 Webhook サービスの `targetPort` を許可されたものに変更するか、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したい場合は、
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）と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、次のコマンドでオペレーターの最新のスナップショットをインストールできます。

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

代わりに、この特定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探している場合は、次を実行できます。

```
curl -sSfL \
  https://raw.githubusercontent.com/cloudnative-pg/artifacts/release-1.20/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対象としていません。

Helm チャートの使用

オペレーターは、提供された *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 展開の詳細

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。このデプロイメントの名前は、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されると、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 を使用すると、 *"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ています。CloudNativePG オペレーターのデフォルト構成には、単一のレプリカの Deployment が付属しており、ほとんどのインストールに適しています。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、オペレーターがリーダー選択をサポートしていれば、デプロイメント構成で複数のレプリカを指定できます。また、テイントと耐性を利用して、実際の PostgreSQL クラスターが実行されているの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ができます（これには、自己管理の 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: リリースノート をよくお読みください

一部のバージョンで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。

> **警告:** バージョン 1.20 にアップグレードする場合は、 dedicated section below を注意深くお読みください。

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します

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

リリースノートで特に明記されていない限り、最初のステップは通常、プレーン Kubernetes インストールの新しいバージョンのマニフェストを適用するか、使用す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います（上記のセクションの指示に従ってください）。

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、展開されたすべての PostgreSQL インスタンスのローリング更新をトリガーして、新しいインスタンスマネージャーを使用します。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 に設定されている場合、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させることにより、ローリング更新を完了する必要があります。

重要: *primaryUpdateStrategy* がデフォルト値の `unsupervised` に設定されている場合、オペレーターのアップグレードは PostgreSQL クラスターのスイッチオーバーをトリガーし、（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は、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。後者では PostgreSQL インスタンスを再起動する必要はなく、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。デフォルトで無効になっているこの動作については、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 Pod に追加されます。

ただし、この動作は構成を介して変更して、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、コンテナーを存続させる PID 1 プロセスです。

内部的には、CloudNativePG バージョン 1.10 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したら、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インジェクションと、すべての内部プロセスのグレースフル終了をサポートしています。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、既に実行中の *postmaster* を採用します。

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なくなります。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を破ることです。

オペレーター設定 `ENABLE_INSTANCE_MANAGER_INPLACE_UPDATES` 環境変数を `'true'` に設定することにより、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、ポッド内の init コンテナイメージは変更されません。したがって、Pod 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、以前のリリース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、少なくとも毎月新しいバージョンのオペレーターを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、各バージョンを順番にアップグレードして、バージョンをスキップせずに定期的に最新の状態にすることをお勧めします。

> リリースノート ページには、CloudNativePG のリリースごとに導入された変更の詳細なリストが含まれており、ソフトウェアの新しいバージョン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 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、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、オペレーター自分自身にのみ影響します。

以前のマイナーバージョンから 1.20 へのアップグレード

CloudNativePG 1.20 では、構成よりも規約を通じて、すぐに使用できる Postgres クラスターの復元力と使いやすさを向上させることを目的として、いくつかの機能のデフォルト動作に以前のバージョンのオペレーターからいくつかの変更が導入されています。

重要: これらの変更はすべて、少なくとも 1 つのレプリカが存在する場合を含み、**新しい `Cluster` リソースにのみ影響します**。

スタンバイからのバックアップ

スタンバイからのバックアップ

は CloudNativePG 1.19 で導入されましたが、デフォルトでは無効になっています。つまり、ターゲットがスタンバイを優先するように明示的に設定されていない限り、ベースバックアップはプライマリから取得されます。

バージョン 1.20 以降、1 つ以上のレプリカが利用可能な場合、オペレーターは完全なベースバックアップを取得するために最も整列されたスタンバイを優先します。

CloudNativePG デプロイメントを 1.20 にアップグレードしており、この機能が作成する新しい `Cluster` リソースの本番環境に影響を与える可能性があることが懸念される場合、次の行をすべての `Cluster` リソースに追加することにより、ターゲットを明示的にプライマリに設定できます。

```
spec:
  ...
  backup:
    target: "primary"
```

ローリングアップデート後のプライマリの再起動

Automated rolling updates

CloudNativePG で常に利用可能であり、デフォルトでは、最も整列されたレプリカへのスイッチオーバーを実行した後にプライマリを更新します。

バージョン 1.20 から、ほとんどの場合、これが最も高速で安全な方法であるため、プライマリのデフォルトの更新方法をスイッチオーバーからリスタートに変更しています。

CloudNativePG デプロイメントを 1.20 にアップグレードしており、この機能が作成する新しい `Cluster` リソースの本番環境に影響を与えることが懸念される場合、次の行をすべての `Cluster` に追加することにより、プライマリの更新メソッドをスイッチオーバーに明示的に設定できますリソース：

```
spec:
   ...
   primaryUpdateMethod: switchover
```

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

CloudNativePG バージョン 1.18 で導入されましたが、デフォルトで無効になっています。

バージョン 1.20 では、レプリケーションスロットにより高可用性クラスターの復元力と堅牢性が強化されるため、バージョン 1.21 からデフォルトでこの機能を有効にする準備をしています。

将来の互換性、環境にレプリケーションスロットが必要ないことがすでにわかっている場合は、次の行を `Cluster` リソースに追加して、管理を明示的に無効にすることをお勧めします。

```
spec:
   ...
   replicationSlots:
     highAvailability:
       enabled: false
```

4.5 クイックスタート

このセクションでは、種類 または *Minikube* でのテストのローカル 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たものであり、実稼働環境で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。

このページの指示に従うと、ローカルの Kubernetes インストールで PostgreSQL クラスターを起動し、実験できるはずです。

重要: Kubernetes クラスターに接続するには、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 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。システムについてしばらく読んで、どちらを続行するかを決定してください。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* でクラスターを監視する** で、ローカルテスト/評価のために Prometheus と Grafana を使用したモニタリングをセットアップする手順も提供します

ミニクベ

Minikube は、Kubernetes をローカルで簡単に実行できるツールです。Minikube は、ノートパソコンの仮想マシン（VM）内でシングルノード Kubernetes クラスターを実行し、Kubernetes を試したり、日常的に開発したりしようとしているユーザー向けです。通常、VirtualBox と組み合わせて使用されます。

詳細については、ローカルの個人環境にある公式の 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。

種類

仮想マシン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「ノード」を使用してローカル 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」の略です）。

> **クイックスタート** の指示に従って `kind`

を環境にインストールし、以下を使用して 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 をインストールして実行したら、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 での他のデプロイメントと同様に、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。

は

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

  storage:
    size: 1Gi
```

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ディスク領域を割り当てる単純な `Cluster` を定義します。

```
#  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
```

(次のページに続く)

(前のページからの続き)

```
  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。Kubernetes インストールでクラスターを他のワークロードから分離するには、クラスターをデプロイする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：

```
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、将来削除されます。 *cngp.io/cluster* をご利用ください。

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。`Cluster` 定義の `spec` セクションで `imageName` キーを設定することにより、これを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：

```
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、常に特定のバージョンのコンテナイメージをポイントする必要があります。 *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、実稼働環境では使用しないでください。厳密な決定論的で反復可能なデプロイメントの場合、 *<image>:<tag>@sha256:<digestValue>* 形式でイメー

ジ名にダイジェストを追加できます。

4.5.4 パート 4：Prometheus と Grafana で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を超えています。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 クラスターをモニタリングする Grafana ダッシュボード、およびアラート条件を定義する Prometheus ルールのセットを作成する方法を示します。

> 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
Prometheus **オペレーターの例** 、および **パート** *4* **：** *Prometheus* **と** *Grafana* **で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 チャートをデプロイするため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` ヘルムチャートリポジトリを追加し、提供す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。

次のコマンドでこれを実現できます。

```
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、および 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 インストールから、Prometheus Operator に **any** PodMonitor を監視させます (*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 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
prometheus-community-kube-operator        ClusterIP   443/TCP
prometheus-community-kube-prometheus      ClusterIP   9090/TCP
```

Prometheus で表示する

この時点で、Monitoring をアクティブ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:
```

(次のページに続く)

(前のページからの続き)

```
    size: 1Gi

  monitoring:
    enablePodMonitor: true
EOF
```

Prometheus にアクセスするには、Prometheus サービスをポートフォワードします。

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

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

モニタリングスタックが正常にデプロイされ、`enablePodMonitor: true` を使用する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。繰り返しますが、詳細については **monitoring section** を参照してください。

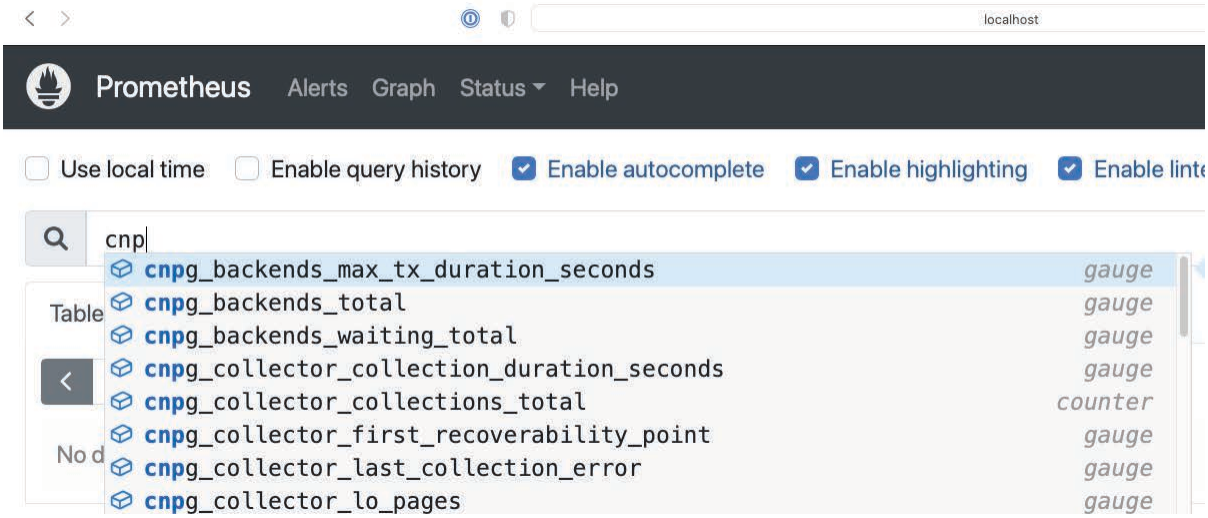


図 11: local prometheus

`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ます。

```
% 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、Grafana は事前定義されたダッシュボードなしでデプロイされます。

ポートフォワードできます。

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

そして、 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提供し、`prom-operator` をパスワードとして提供します (`kube-stack-config.yaml` で定義)。

これで、サンプルの Grafana ダッシュボードをインストールできます。

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

数秒で Grafana ページにピックアップされます。 `CloudNativePG` ダッシュボードが表示されます。

警告: 以前のダッシュボードの一部のグラフは、 *kubelet_volume_stats_available_bytes* や `kubelet_volume_stats_capacity_bytes` のように、これが作成された時点でアルファ段階にあるメトリックを利用しています。*No data* を表示するいくつかのグラフを生成します。

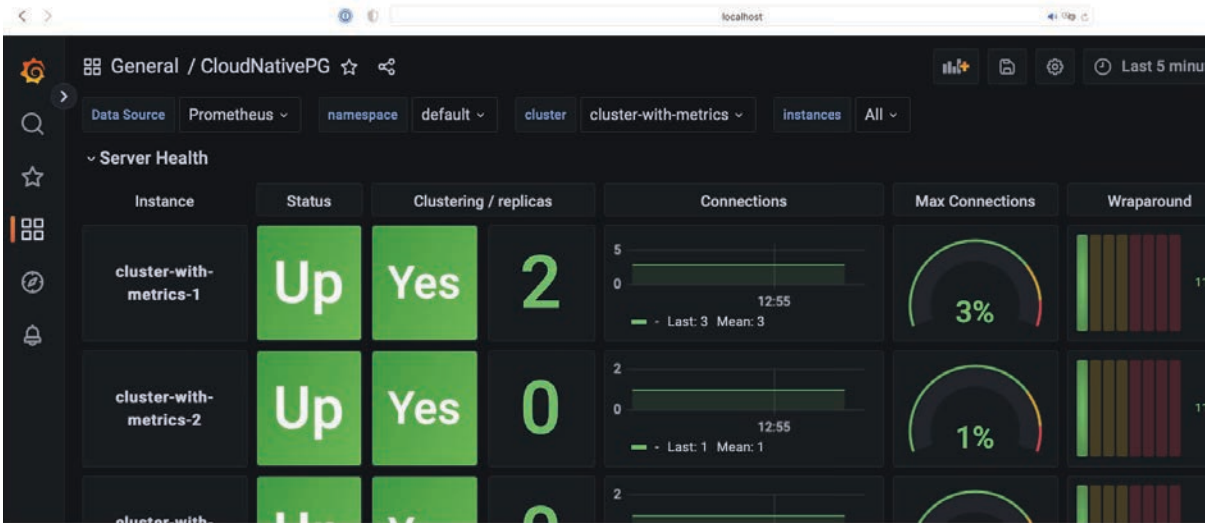


図 12: local grafana

このセットアップ例では、Prometheus と Grafana の両方が、Monitoring がアクティブ化され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、それらの背後にある設計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 2 つの方法があります。

- ゼロから（`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、Kubernetes の外部であっても、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し、Postgres の異なるメジャーバージョンを使用する可能性も提供します。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ブートストラップすると、**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、継続的リカバリーにあり、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。クラスターで管理タスクを実行するには、ローカル Unix ドメインソケットを使用して、 *postgres* ユーザーとして `peer` 認証を介して `postgres` データベースに接続する必要があります。 *postgres* ユーザーまたは `postgres` データベースを**削除しないでください**

注釈: CloudNativePG は *Kubernetes' native `VolumeSnapshot* API <https://github.com/cloudnative-pg/cloudnative-pg/issues/2081>`__ のサポートを徐々に導入しています

バックアップおよびリカバリ操作における増分コピーと差分コピーの両方用 - 基になるストレージクラスでサポートされている場合。

> Recovery from Volume Snapshot objects をご覧ください

詳細については。

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` ：既存のクラスターのベースバックアップから復元し、使用可能なすべての 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を介して pg_basebackup を使用して、同じメジャーバージョンの既存のものをクローニングすることにより、PostgreSQL クラスターを作成します-データベースを CloudNativePG に、Kubernetes の外部からでも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 はどちらも別のクラスター（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スピンアップに使用できます。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何らかの関連がある 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る予定ですが、現在は、物理レプリケーションに基づいて、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クロスリージョン PostgreSQL クラスターを定義することを目的としています。

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 メソッドのいずれかのソース 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、オリジンクラスターを識別する名前
- 次の少なくとも 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、ソースクラスターのバックアップファイル、つまり WAL アーカイブとベースバックアップを含む Barman Cloud 互換オブジェクトストアです。

注釈: リカバリオブジェクトストアは、通常、AWS S3、または 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、ソースを 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に適格です。これは、クラスターが、ストリーミング、PostgreSQL の restore_command を介した WAL シッピング、またはその 2 つのいずれかを介して、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を使用して、新しい PostgreSQL クラスターをゼロから作成します。特に指定がない限り、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ことを行います。

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 シークレットのものを使用して、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。

構成よりも規約のおかげで、オペレーターに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）を選択させ、スーパーユーザーとスーパー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。

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

指定されたシークレットは、 kubernetes.io/basic-auth の仕様に準拠している必要があります。その結果、シークレットの username は、owner （アプリケーションシークレットの場合）およびスーパーユーザーの postgres のいずれか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保存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かもしれません。

---

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ための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続行し、app データベースを作成し、**デフォルトの** *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、デフォルトでデータベース名を使用します。

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ステータスに調整します。

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を介して作成されます。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 を変更するか、データチェックサムを追加する場合)、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に役立ちます - そうでなければサイレント（デフォルト：false ）。

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、テンプレートデータベースのエンコーディングを選択します（デフォルト：UTF8 ）。

localeCollate ： localeCollate が値に設定されている場合、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 のドキュメントから （デフォルト：C）。

localeCType ： localeCType が値に設定されている場合、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 のドキュメントから （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：設定しない-PostgreSQL によって 16 メガバイトとして定義されます）。

注釈: *initdb* ブートストラップ中に CloudNativePG が実装する 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、演算子の動作を壊す可能性のあるオプション（--auth または-d など）があることを考えると、この手法は非推奨であり、API の将来のバージョンから削除されます。

データベースが作成および構成された直後に 1 回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:
```

(次のページに続く)

(前のページからの続き)

```
  name: cluster-example-initdb
spec:
  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、クラスター全体が混乱す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 Secrets および/または ConfigMaps 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
```

(次のページに続く)

(前のページからの続き)

```
        key: configmap.sql
  storage:
    size: 1Gi
```

注釈: *secretRefs* で参照される SQL スクリプトは、 *configMapRefs* で参照される SQL スクリプトよりも前に実行されます。両方のセクションで、 SQL スクリプトはリスト内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、コメントを含めることができますが、 *psql* のような内部コマンドはできません。

警告: *postInitApplicationSQLRefs* で指定された ConfigMap または Secrets 内にエントリーが存在することを確認してください。そうでない場合、ブートストラップは失敗します。これらの SQL ファイルのいずれかにエラーがあると、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始め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、既存の物理**ベースバックアップ**（`recovery` ）を介して間接的に接続することにより発生します。

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（オリジンクラスターと同じ `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の`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、既存の物理ベースバックアップから新しいクラスターを作成し、アーカイブから REDO ログを含む WAL ファイルを再適用できます。ベースバックアップと WAL ファイルの両方が**リカバリオブジェクトストア**からプルされます。

リカバリオブジェクトストアからのリカバリは、2 つの方法で実現できます。

- リカバリオブジェクトストアを使用します。 これは、 Barman Cloud によって作成され、`externalClusters` セクションの `barmanObjectStore` オプションを介して定義された別のクラスターのバックアップです（**推奨**）

- 同じ名前空間内の既存の Backup オブジェクトを使用する（これは、バージョン 1.8.0 より前に利用可能な唯一のオプションでした）。

どちらの回復方法も、完全な回復（利用可能な最後の WAL まで）または最大
ポイントインタイムリカバリ（*PITR*）

のいずれかを有効にします。完全復旧を実行する場合、クラスターをレプリカモードで起動することもできます。また、回復したクラスターの 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 実行中のクラスターのバックアップとリカバリの詳細については、 を見つけることができます。

CloudNativePG は、Kubernetes のボリュームスナップショットのサポートも導入しています。CloudNativePG の現在のバージョンでは、次のことができます。

- `kubectl cnpg snapshot` コマンドを使用して、スタンバイから Postgres クラスターの一貫したコールドバックアップを取得します-必要な `VolumeSnapshot` オブジェクト（現在、別のボリュームに WAL がある場合は 1 つまたは 2 つ）を作成します
- :ref:`VolumeSnapshot` オブジェクトからの回復<`VolumeSnapshot` オブジェクトからの回復>` で説明されているように、 ``.spec.bootstrap.recovery`

  スタンザの `volumeSnapshots` オプションを介して、上記の *VolumeSnapshot* オブジェクトから回復します

以下

オブジェクトストアからの回復

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回復できます。`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 コンテナーに復旧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:
```

(次のページに続く)

(前のページからの続き)

```
    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の`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、最大 8 つのジョブを専用にして、アーカイブから必要な WAL ファイルを同時にフェッチします。この機能により、回復時間を大幅に短縮できます。このシナリオを事前に計画し、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む **いつ** (そうでない場合)、必要になります。

Backup オブジェクトからの回復

クラスターを作成する必要がある名前空間で Backup リソースが既に利用可能な場合、次の例のように、`.spec.bootstrap.recovery.backup.name` を介して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:
```

(次のページに続く)

(前のページからの続き)

```
    name: superuser-secret

  bootstrap:
    recovery:
      backup:
        name: backup-example

  storage:
    size: 1Gi
```

このブートストラップ方法では、復元が必要なバックアップへの参照のみを指定できます。

VolumeSnapshot オブジェクトからの回復

CloudNativePG は、`kubectl cnpg snapshot` で取得された既存の `Cluster` の PVC の `VolumeSnapshot` から新しいクラスターを作成できます。次の例のように、スナップショットの名前を指定する必要があります。

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

bootstrap:
    recovery:
      volumeSnapshots:
        storage:
          name: <snapshot name>
          kind: VolumeSnapshot
          apiGroup: snapshot.storage.k8s.io
```

警告: Kubernetes の `VolumeSnapshot` API の宣言的サポートの開発が進むにつれて、ポイントインタイムリカバリオペレーションまたはレプリカクラスターの WAL アーカイブと組み合わせてこの手法を使用できるようになります。

バックアップされたクラスターが別の PVC を使用して WAL ファイルを保存していた場合、リカバリにはそれも含める必要があります。

```
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: cluster-restore
spec:
  [...]

bootstrap:
    recovery:
      volumeSnapshots:
        storage:
          name: <snapshot name>
          kind: VolumeSnapshot
          apiGroup: snapshot.storage.k8s.io

        walStorage:
          name: <snapshot name>
          kind: VolumeSnapshot
          apiGroup: snapshot.storage.k8s.io
```

kubectl cnpg snapshot コマンドは、ボリュームの物理コピーを作成する前にスタンバイをフェンシングすることにより、**コールドバックアップ**と呼ばれる手法を使用して、レプリカの一貫したスナップショットを作成できます。詳細は

> *Postgres* **クラスターのスナップショット** を参照ください。

その他の考慮事項

リカバリオブジェクトストアまたは既存の Backup リソースのどちらからリカバリする場合でも、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、復元されるバックアップから保持されます。これは Kubernetes クラスター自分自身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で新しいものが自動的に生成されます。その後、シークレットは、クラスターの postgres ユーザーのパスワードをリセットするために使用されます。
- デフォルトでは、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 までリカバリが続行されます（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）。

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により、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 WAL まですべての WAL をリプレイする代わりに、ベースバックアップを抽出した後、任意の時点で WAL のリプレイを停止するように PostgreSQL に依頼できます。PostgreSQL はこのテクニックを使用して、**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および`Backup`オブジェクトから入手できます。

オペレーターは、Azure に格納されている復旧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、復旧ターゲットが指定された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

  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、リカバリを開始するベースバックアップを指定することを心配することなく、タイムスタンプの形式で `targetTime` を指定するだけでよ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。

それに値を（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リカバリ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、オペレーターは次のようにリカバリ用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 を使用する場合、オペレーター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、オペレーターは利用可能な最新のバックアップを時系列で選択します。

使用できるリカバリターゲットの基準は次のとおりです。

targetTime ：
RFC 3339 形式で表される、回復が進行するタイムスタンプ（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開始時に連続して割り当てられますが、トランザクションは異なる番号順に完了す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の前にコミットされたトランザクションです（オプションで含めます）

targetName ：リカバリが進む名前付き復元ポイント（pg_create_restore_point() で作成）

targetLSN ：リカバリが進む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 ：リカバリは、一貫性のある状態に到達次第、つまりできるだけ早く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取得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た場合、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、カタログ内の利用可能な最後のバックアップで問題ない場合を除き、*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
```

(次のページに続く)

(前のページからの続き)

```
      recoveryTarget:
        backupID: 20220616T142236
        targetName: restore_point_1
[...]
```

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

さらに、特定のタイムラインへの targetTLI 強制リカバリを指定できます。

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、リカバリー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、復旧ターゲットの直後に停止し、exclusive パラメーターを false に設定する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:
    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

回復したクラスターの場合、追加の構成で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、それらを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、オペレーターに、ランダムに安全なパスワードを使用して使用するシークレットを生成させることもできます。

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、セクションをご覧ください。

次の例では、所有者 app 、指定されたシークレット app-secret でアプリケーションデータベース app を構成します。

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

上記の構成では、リカバリが完了した後に次のことが起こります。

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 の値とシークレットの場合、アプリケーションデータベースのパスワードは password の値にシークレットに変更されます。

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、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。これらは元のクラスターからリカバリされます。

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、プライマリまたはスタンバイ PostgreSQL サーバーのいずれかです。

このメソッドの主なユースケースは、Kubernetes の外部または Kubernetes 内（別のオペレーターなど）から CloudNativePG への**移行** によって表されます。

> **警告:** 現在の実装では、クローニングプロセスが終了し、作成されたクラスターをすぐに起動すると、オリジン PostgreSQL インスタンスの*スナップショット*を作成します。詳細については、以下の **現在の制限** 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、オペレーターは最初のインスタンスからターゲットクラスターの所有権を取得します。これには、CloudNativePG で必要とされるいくつか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で自分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 によって管理されるソースに接続する場合、または TLS 認証用に構成されている場合、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 外部の従来の環境にある場合、最も簡単な方法かもしれません。両方の場合について以下に説明します。

要件

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

- 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である必要があります
- ターゲットとソースは同じメジャー PostgreSQL バージョン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** 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 回限りのオペレーションの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
- ソースは `REPLICATION LOGIN` 特権を持つロールを持っていなければならず、`pg_hba.conf` のこのロールのターゲットインスタンスから、できれば TLS を介した接続を受け入れる必要があります（以下の **レプリケーションユーザーについて** を参照）
- ターゲットは、`REPLICATION LOGIN` 特権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

と High Availability, Load Balancing, and Replication

PostgreSQL のドキュメント。

レプリケーションユーザーについて

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` 、またはできれば `REPLICATION` 特権だけ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、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、これは `streaming_replica` に許可される唯一の接続方法です）。

Kubernetes の外部を含む他のすべてのケースについては、 `REPLICATION` 特権を持つユーザーが既にいることを確認するか、以下の指示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
```

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、プロンプトでパスワードを入力し、後で使用するために保存します。

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適用する限り、必要に応じて自由に変更してください。

ユーザー名/パスワード認証

pg_basebackup ブートストラップを使用して CloudNativePG が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マッチングに基づいています。

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、 target-db と呼ばれる新しい PostgreSQL 15.3 クラスターを作成します。pg_basebackup ブートストラップメソッドを使用して、 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。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。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: target-db
spec:
  instances: 3
  imageName: ghcr.io/cloudnative-pg/postgresql:15.3

  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
```

クローンオペレーションが機能するには、同じ PostgreSQL バージョン（この場合は 15.3）を含む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 がサポートする 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 クラスター内に既存の PostgreSQL クラスター（cluster-example ）のクローンを作成します。

注釈: この例は、Kubernetes クラスターの外部にあるインスタンスをカバーするように簡単に適合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 15.3 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:15.3

  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:
```

(次のページに続く)

(前のページからの続き)

```
      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` ブートストラップ方法の場合と同様に、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）、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、指定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:
  bootstrap:
    pg_basebackup:
      database: app
      owner: app
      secret:
        name: app-secret
      source: cluster-example
```

上記の構成では、リカバリが完了した後に次のことが起こります。

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 の値とシークレットの場合、アプリケーションデータベースのパスワードは password の値にシークレットに変更されます。

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、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。これらは元のクラスターからリカバリされます。

現在の制限

テーブルスペース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、テーブルスペースの完全な宣言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、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扱う場合、PostgreSQL 自分自身は、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、この場合は CloudNativePG が管理する Kubernetes のポッドにも存在する必要があります。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（最初にソースからインスタンスを削除する必要があります。組織でこの機能が必要な場合は、EDB に連絡して優先順位を付けます)。

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)。

バックアップが完了すると、新しいインスタンスは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 で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、プロシージャ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には、必要なだけ移行手順を実行することが基本です。 EDB に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、オリジンホストへのネットワーク接続を介した `pg_dump` 、および `pg_restore` に依存しています。ネイティブのマルチバージョン同時実行制御（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、ネットワークを介して同時に一貫したバックアップを作成できます。

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できる手法でもあります。

その結果、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

- 任意の PostgreSQL バージョンから同じまたは新しいバージョンにデータベースをインポートし、PostgreSQL の**メジャーアップグレード**を有効にします（バージョン 11.x からバージョン 15.x など）

> **警告:** PostgreSQL のメジャーアップグレードを実行するときは、アプリケーションが新しいバージョンと互換性があること、および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、操作は元のデータベースの一貫した **スナップショット**で実行されます。

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。 「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 仕組み

概念的には、インポートでは *BootstrapInitDB* を使用して新しいクラスター（**宛先クラスター**）を最初から作成し（**宛先クラスター**）、`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 の推奨事項に従って、**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**のメジャーバージョン以上であることをお勧めします。

CloudNativePG 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ための 2 つの主な方法を提供します。

- **マイクロサービスアプローチ**：宛先クラスターは、CloudNativePG プロジェクトの推奨に従って、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異なる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` タイプを介して利用でき、後者は `monolith` タイプを介して利用できます。

> **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ための十分な特権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保証するのはあなたの責任です。 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では、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 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 を使用した選択したデータベース（initdb.import.databases 内）のエクスポート
- pg_restore --no-acl --no-owner を使用したデータベースの、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 パラメーター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

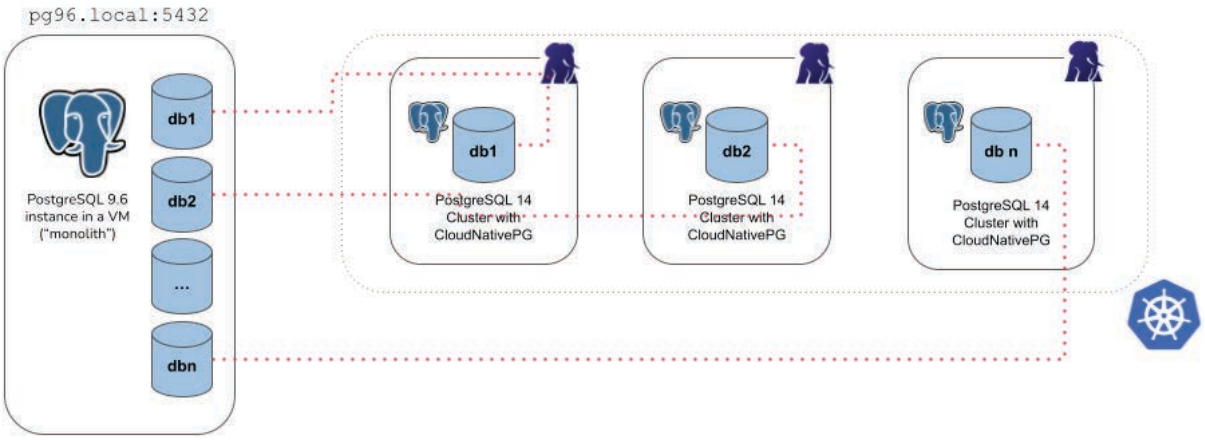


図 13: Example of microservice import type

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 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 ユーザーを使用するデータベース。

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

  bootstrap:
    initdb:
      import:
        type: microservice
```

(次のページに続く)

(前のページからの続き)

```
      databases:
        - angus
      source:
        externalCluster: cluster-pg96
      #postImportApplicationSQL:
      #- |
      #  INSERT YOUR SQL QUERIES HERE
  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
      password:
        name: cluster-pg96-superuser
        key: password
```

警告: 上記の例では、コミュニティ、したがって CloudNativePG でサポートされなくなった PostgreSQL のバージョン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います。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、ソースと等しいかそれ以降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 Postgres の古いサポートされていないバージョンでも機能するはずであり、レガシーデータを Kubernetes 内のより良いシステムに移動する機会を提供します。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、ご意見をお聞かせください。

`microservice` タイプを使用する際には、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）
- オペレーション中に 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
- `pg_dump` を実行し、ロール情報を読み取る必要がある指定されたユーザーでソースデータベースへの接続を許可する必要があります（**スーパーユーザー**は OK です）
- 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、復元されたデータとインデックスを一時的に格納するための十分なスペースが必要です。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 内で指定できません

4.7.3 monolith タイプ

モノリスアプローチでは、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ロール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
- 選択したロール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
- 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

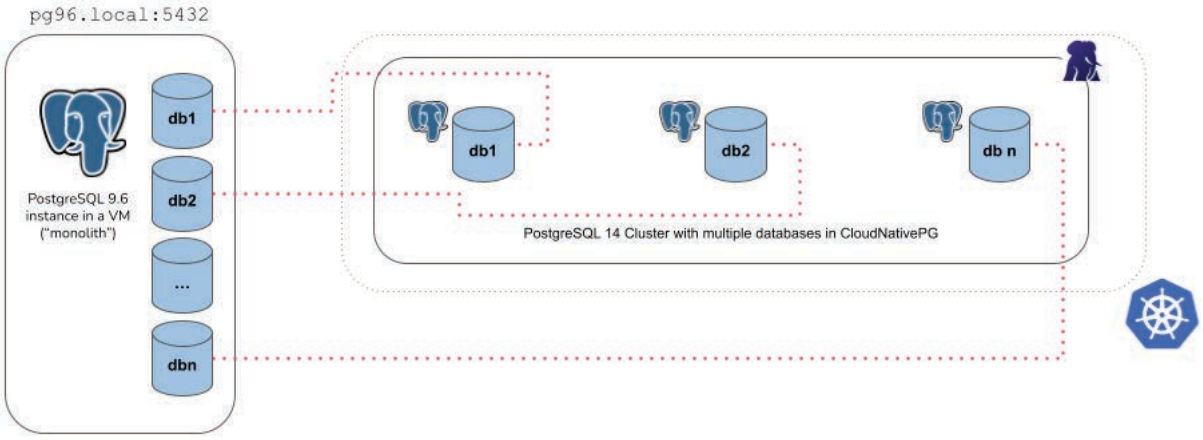


図 14: Example of monolith import type

たとえば、以下の YAML は、accountant および bank_user ロール、ならびに accounting 、banking 、resort データベースをインポートする cluster-monolith という新しい 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。クラスター（サポートされていない PostgreSQL 9.6 を使用）、 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します。

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

(次のページに続く)

(前のページからの続き)

```
  initdb:
    import:
      type: monolith
      databases:
        - accounting
        - banking
        - resort
      roles:
        - accountant
        - bank_user
      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は、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）
- オペレーション中に 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
- pg_dump を実行し、ロール情報を取得する必要がある指定されたユーザーでソースデータベースへの接続を許可する必要があります（**スーパーユーザー**は OK です）
- 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、復元されたデータとインデックスを一時的に格納するための十分なスペースが必要です。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ロールは、以下の制限付きで initdb.import.roles 内で指定する必要があります。-次のロールが存在する場合、インポートされません：postgres 、streaming_replica 、cnp_pooler_pgbouncer

- ワイルドカード"*" は、databases および/または roles 配列内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。データベースと一致する場合、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それらのデータベースを無視します
- clone プロシージャが完了した後、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 セキュリティ

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、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 クラスターで通常の InfoSec の職務を実行することから免責されるものであってはなりません。 *Overview of Cloud Native Security* に慣れてください

Kubernetes ドキュメントのページ。

EDB が CloudNativePG のセキュリティで取ったアプローチの理解とコンテキストをより良く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**されます。GolangCI-Lint は、同じソースコードで複数の**リンター**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。、整数オーバーフロー、 SQL インジェクション - いくつか例を挙げると。

重要: CI / CD パイプラインの静的コード分析フェーズでの障害は、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 で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 コンテナ

CloudNativePG の一部であるすべてのコンテナイメージは、コミットごとに CI / CD パイプラインを介して自動的にビルドされます。このようなイメージには、演算子だけでなく、オペランド、特にサポートされているすべての PostgreSQL バージョンも含まれます。パイプライン内で、画像は次のものでスキャンされます。

- Dockle ：コンテナ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、ベースイメージおよびパッケージレベルでセキュリティ更新が行われた場合にパイプラインによって 1 日 1 回自動的に再構築され、EDB が配布するコンテナイメージに**パッチレベルの更新**を提供します。

コンテナレベルのセキュリティのために、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）が開発した *CIS Benchmark for Docker*

EDB が CloudNativePG のコンテナレベルでセキュリティに取ったアプローチの詳細については、ブログ記事。

4.8.3 クラスター

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、および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。

ロールベースのアクセス制御（RBAC）

オペレーターは、`cnpg-manager` と呼ばれる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ます。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。

重要: 上記の権限は、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だけに予約されています。 *Cluster* 、*Pooler* 、*Backup* 、および `ScheduledBackup` リソースとのみ対話するオペレーターのユーザーは、直接アクセスできません。

以下にいくつかの例を示しますが、最も重要なことに、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、ジョブを処理して、さまざまな `Cluster` のフェーズを管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、選択されたストレージクラスと対話して、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 証明書をセルフプロビジョニングし、それらを秘密に保存することにより、「構成よりも規約」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、自動化された方法でフェールオーバー/スイッチオーバー操作を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* をご覧ください。

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、どのノードでポッドをスケジュールできるかを判断し、レプリカが同じノードにならないようにできます。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ポッド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表示するには、kubectl describe clusterrole cnpg-manager を実行できます。

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、Kubernetes API サーバーを呼び出して、一部のリソースのステータスが正しく更新され、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 サーバーを介してリソースの特定の制限された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。

透明性のために、サービスアカウントに関連付けられた権限は roles.go

ファイル。たとえば、 myns 名前空間にある汎用 mypg クラスターの権限を取得するには、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: **ロールは特定の名前空間に制限される**ことに注意してください。

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た権限の簡単な概要を示します。

`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、すなわち：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

`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

代わりに、ここでは、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` リソース専用の読み取り専用権限、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` に自分自身の `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には、 `get` および `list` 権限が必要です。さらに、オブジェクトストアに対応するものがない `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。通常は保持ポリシーのため

`backups/status` ：インスタンスマネージャーは、 `update` および `patch` に、名前空間内の `Backup` リソースのステータスを必要とします

ポッドセキュリティポリシー

Pod Security Policy

は、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、コンテナの実行に**特権**モードを必要としません。PostgreSQL コンテナは、`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一切ありません。

同様に、ボリュームアクセスには**特権**モードまたは `root` 特権も必要ありません。適切な権限は、Kubernetes プラットフォームおよび/または管理者が適切に割り当てる必要があります。PostgreSQL コンテナは、読み取り専用のルートファイルシステム（つまり、書き込み可能なレイヤーなし）で実行されます。

オペレーターは、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 を使用した Pod アクセスの制限

container.apparmor.security.beta.kubernetes.io アノテーションを介して、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 を使用した Pod アクセスの制限* 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、クラスターが動作を停止する可能性があります。この場合、注釈を `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、基に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、注釈が Cluster 作成時に追加された場合、ポッドは作成されません。一方、Cluster が作成された後にアノテーションを追加すると、クラスター内のポッドを開始できず、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 管理者に連絡し、使用する適切な AppArmor プロファイルを依頼してください。

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 で制御できます

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のネットワークアクセスを有効/無効にします。詳細については、
: *networking document* を参照してください。

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はこのことに留意してください。より細かい制御のために CloudNativePG が使用す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を超えています。
: 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

CloudNativePG は、以下の表に示すように、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 |ポート番号 |露出 |名前 |証明書 |認証 :———— | :——– | :———— | :———— | :———— | :———— – 演算子| 9443 |ウェブフックサーバー | webhook-server | TLS |はい演算子 | 8080 |メトリックス | metrics | TLS なし|インスタンスマネージャーなし | 9187 |メトリックス | 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 ファイルを自動的に作成します。

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。PostgreSQL 14 以降、 password_encryption はデフォルトで scram-sha-256 に設定されますが、以前のバージョンでは md5 に設定されます。

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 に設定することにより、シークレットを介した postgres ユーザーパスワードの管理を無効にできます。

注釈: オペレーターは、 *enableSuperuserAccess* オプションのトグル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（オペレーターが以前に生成した場合）、 *postgres* ユーザーのパスワードを `NULL` に設定します（事実上、パスワード認証を介したリモートアクセスを無効にします）。

詳細については、 *PgBouncerSecrets* 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 と呼ばれる特別なユーザーを使用して、現在のプライマリインスタンスと**物理非同期ストリーミングレプリケーション**で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、管理者が postgresql 構成の pg_hba セクションの一部としてマニフェストに pg_hba.conf 行を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。

pg_hba.conf がオペレーターによって管理される方法の詳細については、ドキュメントの *PostgreSQL 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。

ストレージ

CloudNativePG は、保存時の暗号化を基になるストレージクラスに委任します。運用環境でのデータ保護には、保存時の暗号化をサポートするストレージクラスを選択することを強くお勧めします。

4.9 Postgres インスタンスマネージャー

CloudNativePG は、フェールオーバー管理のために外部ツールに依存しません。単に 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しています。

インスタンスマネージャーは、PostgreSQL の主要なプロセス（postmaster としても知られています）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、メインコンテナの親プロセス（PID 1）としてインスタンスマネージャーを起動し、次に PostgreSQL インスタンスを実行します。Pod の存続期間中、インスタンスマネージャーはバックエンドとして機能し、liveness and readiness probes.

4.9.1 Liveness と Readiness プローブ

liveness プローブは pg_isready に依存していますが、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 プローブはポジティブです。 liveness probe 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各チェックの間に 10 秒の間隔で 3 回失敗した場合、2 つのプローブは失敗を報告します。

今のところ、スタートアッププローブは Kubernetes 1.17 でのみ導入されているため、オペレーターは Pod で startupProbe を構成しません。

liveness probe は、PostgreSQL インスタンスが壊れた状態であり、再起動が必要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延させるために使用され、起動時間が長いインスタンスが再起動されないようにする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` パラメーターで表され、デフォルトは 30 秒です。クラスターの正しい値は、PostgreSQL の起動に必要な時間に関連しています。

`.spec.startDelay` が低すぎる場合、liveness プローブは PostgreSQL の起動前に動作を開始し、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 が手動またはノードのドレイン操作に従って Kubernetes によって削除されると、kubelet は終了シグナルをインスタンスマネージャーに送信し、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、シャットダウンするために PostgreSQL に与えられる時間です。値のデフォルトは 30 秒です。

シャットダウン手順は 2 つのステップで構成されています。

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 への新しい接続を許可しません。このステップは、`.spec.stopDelay` で設定された時間の半分続きます。

2. PostgreSQL がまだ起動している場合、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、既存の接続を終了し、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 クラスター内のデータ損失を回避するには、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、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可能であることを保証するために、選択した新しいプライマリを昇格させる前に、元のプライマリが**高速** シャットダウンを発行することを要求します。

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、すべての WAL ファイルをアーカイブするために与える時間を制御します。この時間枠中、プライマリインスタンスは接続を受け入れません。値のデフォルトは 1 年よりも大きく、無限の遅延をシミュレートするのに十分な大きさであり、したがってデータの耐久性を維持します。

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 が優先される場合がありますが、クラスターレベルおよび/またはバックアップレベルでデータが失われます。逆に、これを高い値に設定すると、データ損失のリスクが排除され、スイッチオーバー中にアクティブなプライマリなしでクラスターが長時間離れなくなります。

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、クラスターはフェールオーバーモードになります。詳細は 自動フェイルオーバー を参照。

4.10 スケジューリング

Kubernetes で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責任を負うプロセスです。

利用可能なすべてのポリシーを含む、スケジュールの詳細については。このページでは、アフィニティ、アンチアフィニティ、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。

> *AffinityConfiguration* を介して 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定義のセクション。

- ポッドアフィニティ/アンチアフィニティ
- ノードセレクター
- 寛容

注釈: CloudNativePG は、ワークロードのスケジューリングをより細かく制御するためのポッドテンプレートをサポートしていません。それらは最初のコンセプトの一部でしたが、開発チームは API の新しいバージョン（おそらく CNPG の v2）での導入を延期することを決定しました。

4.10.1 Pod アフィニティとアンチアフィニティ

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（**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。

CloudNativePG はデフォルトで、クラスターのインスタンスをできれば異なるノードで構成し、次の `affinity` 定義になります。

```
affinity:
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
      - podAffinityTerm:
          labelSelector:
            matchExpressions:
              - key: postgresql
                operator: In
```

(次のページに続く)

(前のページからの続き)

```
              values:
                - cluster-example
        topologyKey: kubernetes.io/hostname
      weight: 100
```

次のクラスター仕様の結果：

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  imageName: ghcr.io/cloudnative-pg/postgresql:15.3

  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 3 つの異なるノード上で 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-リソースが許す限り。

上記の設定を調整することにより、前述のデフォルトの動作を変更できます。

`podAntiAffinityType` は `required` に設定できます。`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条件です）。

クラウド環境での `topologyKey` の別の可能な値は `topology.kubernetes.io/zone` です。これにより、ポッドがノードだけでなくアベイラビリティーゾーン全体に分散されるようになります。

> Well-Known Labels, Annotations and Taints を参照してください

。

`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ーポリシーを無効にできます。

さらに、より きめ細かな制御が必要な場合は、 `additionalPodAffinity` および `additionalPodAntiAffinity` 構成属性を介して、カスタムポッドアフィニティまたはアンチ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: *additionalPodAntiAffinity* または `additionalPodAffinity` に、Pod 仕様で予想される `podAntiAffinity` または `podAffinity` のコンテンツ全体を渡す必要があります（どの PostgreSQL に関係なく、すべてのワーカーノードで PostgreSQL のインスタンスを 1 つだけ実行する例として、次の YAML をご覧ください彼らが属するクラスター）。

```
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（キーと値のペアとして定義）のリストを提供して、ポッドを実行できるノードを選択できます。具体的には、ノードには、スケジュールされて実行されるポッドのラベルとして、指定された各キーと値のペアが必要です。

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実行するように PostgreSQL クラスターをリクエストできます。

4.10.3 容認

Kubernetes では、ノードが `taints` を明示的に（`tolerations` を介して）容認していないすべてのポッドを撃退するかどうかを（`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により、Kubernetes スケジューラーは汚染されたノードを考慮し、ワークロードをスケジュールするノードを決定します。`.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容認を構成できます。これは、容認の通常の Kubernetes 構文を受け入れます。

4.11 リソース管理

一般的な 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` セクションを使用して、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使用量を制御および管理できます。

- `requests` ：初期要件

- `limits` ：最大使用量、リソースニーズが動的に増加する場合

たとえば、次のように 32MiB（128MiB にスケーラブル）の初期量の RAM と 50m の CPU（100m にスケーラブル）を要求できます。

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

メモリの要求と制限はコンテナに関連付けられていますが、ポッドにはメモリの要求と制限があると考えると便利です。ポッドのメモリ要求は、ポッド内のすべてのコンテナに対するメモリ要求の合計です。

ポッド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。ポッドは、ポッドのメモリ要求を満たすのに十分な使用可能なメモリがノードにある場合にのみ、ノードで実行されるようにスケジュールされます。

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

- **保証**
- **壊れやすい**
- **ベストエフォート**

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」QoS を設定することをお勧めします。

Kubernetes のリソース関連の問題を回避するには、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 resources 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 Out Of Memory を表します）および `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」QoS クラスに割り当てられるには、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。
- ポッドリソースと一貫して、必要な PostgreSQL メモリパラメーターを指定します（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で *“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。

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  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 パラメーターを指定しました-つまり、データをキャッシュするために PostgreSQL サーバー専用に割り当てられるメモリの量（このパラメーターのデフォルト値は、定義されていない場合、128MB ）。

shared_buffers の妥当な開始値は、システムのメモリの 25%です。例えば：shared_buffers が 256 MB の場合、コンテナ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、コンテナが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 故障モード

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提供します。

重要: 発生している障害シナリオがこのセクションで説明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 によって実装される liveness プローブと readiness プローブを参照してください。

4.12.1 ストレージスペースの使用量

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 番目の 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 つの場合に再利用のために設定されます。

- 対応する Pod がユーザーによって削除されたとき（そして新しい 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

オペレータが特定の PVC を再利用できないようにしたい場合は、ポッドを削除する前に 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、このプロセス中に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：

```
$ 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。

- ポッドがユーザーによって明示的に削除された。
- postgres コンテナの readiness プローブが失敗します。
- postgres コンテナの活性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 ワーカーノードに障害が発生します。

これらの障害はそれぞれ、 Cluster とオペレーターが管理するサービスにさまざまな影響を与えます。

ユーザーによって削除されたポッド

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、既存の PVC が利用可能な場合は再利用して、それ以外の場合は**プライマリ**の物理バックアップから自動的に作成されます。

重要: Pod を意図的に削除する場合、 *PodDisruptionBudget* ポリシーは強制されません。

自己修復は、 *apiserver* 通知されるとすぐに行われます。

Readiness 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。ポッドは引き続き `Cluster` の一部であり、新しいポッドは作成されません。

失敗の原因を修正できない場合は、ポッドを手動で削除できます。それ以外の場合、ポッドは、障害が解決したときに前のロールを再開します。

プローブが 3 回失敗すると、自己修復が発生します。

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できます。

プローブが 3 回失敗すると、自己修復が発生します。

ワーカーノードがドレインされました

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**プライマリ**の物理バックアップとは異なる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少なくとも別のポッドがある場合、ポッドがエビクトされるのを妨げる場合があります。

注釈: *reusePVC* が `false` に設定されている場合、シングルインスタンスクラスターは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。

自己修復は、 *apiserver* 通知されるとすぐに行われます。

ワーカーノードの障害

ノードに障害が発生したため、 *kubelet* は liveness プローブと readiness プローブを実行しません。ポッドは、その特定の障害の原因に対して Kubernetes クラスター管理者が構成した容認秒後に、削除のマークが付けられます。Kubernetes クラスターの構成方法によっては、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。

プライマリの物理バックアップとは異なるワーカーノードに新しいポッドが作成されます。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 自己治癒

障害が発生したポッドがスタンバイの場合、ポッドは `-r` サービスおよび`-ro` サービスから削除されます。次に、ポッドは、使用可能な場合はその PVC を使用して再起動されます。それ以外の場合、新しいポッドは現在のプライマリのバックアップから作成されます。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。

障害が発生したポッドがプライマリの場合、オペレーターはアクティブなポッドを準備完了ステータスと最小のレプリケーションラグで昇格させ、`-rw` サービスをポイントします。障害が発生したポッドは、 `-r` サービスおよび`-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の複製を開始します。元のプライマリは、`pg_rewind` を使用して、PVC が利用可能な場合、新しいプライマリと同期します。そうしないと、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 以降、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、異常/緊急操作の唯一の期間中に使用する必要があります。

警告: このアノテーションは限られた期間のみ使用し、緊急事態が終了したら削除してください。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ールオーバーなどの自己修復オペレーションを発行できなくなります。

4.13 ローリングアップデート

オペレーターは、アプリケーションの実行中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ことを保証するため（**
  in-place updates are enabled を除く）。

オペレーターは、一度に 1 つの Pod ですべてのレプリカのアップグレードを開始し、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。

ローリングアップデートは構成可能で、完全に自動化（`unsupervised` ）することも、人間の介入を必要とする（`supervised` ）こともできます。

アップグレードでは、データを再クローンせずに、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に、各サービスエンドポイントが移動してクラスターのステータスを反映するため、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、ローリングアップデートプロセスは 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、次のいずれかの値を受け入れます。

- 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、再起動要求は無視され、スイッチオーバーが発行されます。これがデフォルトの動作です。
- switchover ：スイッチオーバー操作が自動的に実行され、最もアラインさ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し、以前のプライマリポッドをシャットダウンします。

更新方法に万能な構成はありません。データベースの実際のワークロード、RPO と RTO の要件、PostgreSQL アーキテクチャが共有されているかどうかなど、いくつかの要因に依存します。オン。

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、更新プロセスは必然的にアプリケーションのダウンタイムを生成します。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 メソッドは、プライマリポッドがシャットダウンした後に、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。データベースにとって、ローリング更新手順の一部として 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あなた次第です。

4.13.2 手動更新（supervised ）

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

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のいずれかでのみ完了できます。イメージのアップグレードはインプレースリスタートでは適用できないため、そのような場合にはスイッチオーバーが必要であることに注意してください。

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]
```

次のコマンド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では、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です。

重要: このセクションは、同じ Kubernetes クラスターで管理される同じ `Cluster` リソース内のレプリケーションについてです。異なる Kubernetes クラスター間でも、別の Postgres *Cluster* リソースで複製する方法については、 レプリカクラスター

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

PostgreSQL のレプリケーション機能に長年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ーの上に 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複製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ます。WAL（Write Ahead Log）に継続的に保存されるトランザクションの変更の概念に基づいて構築されました。

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、物理レプリケーションは PostgreSQL 8.2（2006）で初めて導入され、継続的なリカバリのプライマリからウォームスタンバイへの WAL シッピングを介して導入されました。

PostgreSQL 9.0（2010）は WAL ストリーミングと**ホットスタンバイ**を介した読み取り専用レプリカでそれを強化しましたが、9.1（2011）はトランザクションレベルでの同期レプリケーションを導入しました（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礎は PostgreSQL 9.4 に置かれましたが、バージョン 10（2017）では、オリジンから宛先にデータをレプリケート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、宣言的な方法でクラスター内の物理ストリーミングレプリカを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の直後、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、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のストリーミングレプリケーションを自動的に設定し、streaming_replica ユーザーに TLS クライアント証明書認証を強制します- pg_hba.conf から取得した次の抜粋で強調表示されているように。

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

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

ドキュメントに。

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、フェールオーバーまたはスイッチオーバー後でも、各スタンバイに必要な WAL ファイルがプライマリのストレージに保持されるようにします。

以下。

継続的バックアップ統合

クラスターで継続的バックアップが構成されている場合、CloudNativePG は、継続的リカバリ時に 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を介した WAL のプルに失敗したときはいつでも、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、いつでも使用可能な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。自己修復の目的で、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、クォーラムを判断します。

重要: デフォルトでは、同期レプリケーションは使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` および `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, ...)
```

そこで：

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

警告:

自己修復機能を提供するために、オペレーターは`minSyncReplicas` 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、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 で述べられているように、 *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自己修復を選択した場合でも、3 つ以上のインスタンスを持つクラスターでのみ、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する

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いて、アンチアフィニティルールを介して、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 PostgreSQL インスタンスを選択できます。

この機能のユースケースの例：単一の同期レプリカを持つクラスターでは、通常 `topology.kubernetes.io/zone` label on a node で識別されるプライマリインスタンスとは異なるアベイラビリティーゾーンに同期レプリカがあることを保証できます。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ー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ーの考え方は、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、アベイラビリティーゾーンラベル）とプライマリが現在あるノードの値が異なるノードで実行されているポッ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の対象となります。

重要: 自己修復の強制は、同期レプリカ選択の追加の制約を定義しているときにも適用されます（同期レプリケーションによるゼロデータ損失クラスター を参照）。

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、評価する必要があるノードラベルを指定して、現在のプライマリと、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、アベイラビリティーゾーンは一例にすぎませんが、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

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 セグメントを削除せず、プライマリが行を削除しないことを保証する自動方法を提供します。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 はスタンバイサーバーに複製しません。その結果、フェー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、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により、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ピング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にのみ存在します。
- **スタンバイ HA スロット**：その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、プライマリの pg_replication_slots ビューの内容に基づいて管理され、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、デフォルトで有効になりました。構成で無効にできます。詳細は

ReplicationSlotsConfiguration

を参照ください。以下に、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、機能は有効になっています（`true` がデフォルトです-1.20 以降）

`.spec.replicationSlots.highAvailability.slotPrefix`：この機能のオペレーターが管理す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秒単位で表される現在のプライマリ上の位置と同期する頻度（デフォルト：30）

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

レプリケーションスロットの位置を直接操作します。

> **警告:** PostgreSQL 11 では、最初に無効になっている場合にレプリケーションスロットを有効にしたり、逆に最初に有効になっている場合に無効にしたりすると、クラスターのローリング更新が必要になります（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）。

お勧めはしませんが、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を有効にせずに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
  # Disable replication slots for HA in the cluster
  replicationSlots:
    highAvailability:
      enabled: false

  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` ビューを照会し、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
```

(次のページに続く)

(前のページからの続き)

```
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 メトリックに、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キー情報を提供します。

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いる（ダウンタイムは必要ない）こと、および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リカバリ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 ツールに基づいた継続的バックアップインフラストラクチャを調整できます。オペレーターは、多くの PostgreSQL インスタンスをバックアップする Barman サーバーで古典的なアーキテクチャを使用する代わりに、

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 *tarball* 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 が含まれた画像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: Barman cloud で導入された改善を利用するには、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ください（およびクラスターのセキュリティ側面を改善します）。

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（

レプリカクラスター 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）、または スタンバイからのバックアップ。

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。つまり：

- AWS S3
- *Microsoft Azure Blob Storage*
- *Google Cloud Storage*

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、次のセクションで説明します。

S3

2 つの方法で S3 バケットにバックアップを保存する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 を使用することをお勧めします
- または、 `ACCESS_KEY_ID` および `ACCESS_SECRET_KEY` 資格情報を使用できます

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シークレット部分
- `ACCESS_SESSION_TOKEN` ：必要な場合のオプションのセッショントークン

使用されるアクセスキーには、バケットにファイルをアップロードする権限が必要です。そのため、資格情報で 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。

そのシークレットを作成したら、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
[...]
```

(次のページに続く)

(前のページからの続き)

```
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 互換オブジェクトストレージプロバイダー

MinIO や **LinodeObjectStorage** 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、デフォルトの S3 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、領域 `us-east1` の **Linode** の `bucket` を使用します。

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

(次のページに続く)

(前のページからの続き)

```
    barmanObjectStore:
      destinationPath: "<destination path here>"
      endpointURL: "https://bucket.us-east1.linodeobjects.com"
      s3Credentials:
        [...]
```

DigitalOceanSpaces を使用している場合、Path スタイルの構文を使用する必要があります。この例では、リージョン SFO3 の **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"
      s3Credentials:
        [...]
```

重要: HTTPS 経由で MinIO を使用する場合など、プライベート CA で署名された証明書を使用する Object Storage 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: ConfigMap と Secrets をインスタンスによって**自動的に**リロードしたい場合、キー`cnpg.io/reload`を持つラベルを 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 Gateway 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、以前に作成し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 MinIO ゲートウェイを直接ポイントできます。

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、同じ名前空間でそれらを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>
```

注釈: Cloud Object Storage の資格情報は、この場合 MinIO Gateway でのみ使用されます。

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：

```
apiVersion: v1
kind: Service
metadata:
  name: minio-gateway-service
spec:
  type: ClusterIP
  ports:
    - port: 9000
      targetPort: 9000
      protocol: TCP
  selector:
    app: minio
```

警告: このドキュメントを書いている時点で、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 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リレーします。

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

```
apiVersion: apps/v1
kind: Deployment
[...]
spec:
  containers:
  - name: minio
    image: minio/minio:RELEASE.2020-06-03T22-13-49Z
    args:
```

(次のページに続く)

(前のページからの続き)

```
    - 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:
          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 Gateway サービスを構成し、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
```

(次のページに続く)

(前のページからの続き)

```
[...]
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 Blob ストレージ

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

- ***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。

AzureADWorkloadIdentity を使用すると、資格情報を Kubernetes Secret に保存せずに、クラスター構成に次のように `inheritFromAzureAD` を追加できます。

```
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**の両方を使用して、資格情報を Kubernetes Secret 内に保存し、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。

以前のシークレットがあれば、提供された資格情報をクラスター構成内に注入できます。

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

<resource-path> は<container>/<blob> です。アカウント名はストレージアカウント名とも呼ばれ、使用す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 Storage Emulator 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 Cloud Storage

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 番目のものは環境変数 GOOGLE_APPLICATION_CREDENTIALS を活用します

Google Kubernetes Engine 内で実行

Google Kubernetes Engine 内で実行する場合、資格情報を設定せずに、単に
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 に設定します
- serviceAccountTemplate スタンザに iam.gke.io/gcp-service-account アノテーションを設定します

次の例を参考にしてください。

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

  serviceAccountTemplate:
    metadata:
      annotations:
```

(次のページに続く)

(前のページからの続き)

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

認証の使用

> instruction from Google に続いて

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 ファイルの内容は、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、次のように yaml ファイルで使用されます。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      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 に対して認証を行います。

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取得されます。

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要求するには、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
```

(次のページに続く)

(前のページからの続き)

```
spec:
  cluster:
    name: pg-backup
```

オペレーターは、クラスターのオーケストレーションを開始し、barman-cloud-backup を使用して必要なバックアップを取得します。プレーンな kubectl describe backup <name> コマンドを使用して、バックアップの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
Status:
  Phase:       running
  Started At:  2020-10-26T13:57:40Z
Events:        <none>
```

バックアップが完了すると、次の例のように、フェーズは 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
```

(次のページに続く)

(前のページからの続き)

```
  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 と呼ばれる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。

```yaml
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リカバリ操作を必要とする災害後のリカバリ時間オブジェクト（RTO）に影響を与える可能性があります。私たちのアドバイスは、バックアップを回復して定期的にバックアップをテストし、最初から回復するのにかかる時間を測定して、RTO の予測可能性を向上させることです。リカバリ時間は、ベースバックアップのサイズと、アーカイブからフェッチ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ことを忘れないでください！）。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、バックアップを 1 日に 1 回以上スケジュールすることは非常にまれです。

ScheduledBackups は、必要に応じて `.spec.suspend: true` を設定することにより一時停止できます。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します。

ScheduledBackup リソースが作成されたら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*：スケジュールされたバックアップオブジェクトをバックアップの所有者として設定します- *cluster:* クラスターをバックアップの所有者として設定します

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および/または暗号化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 はシーケンシャルアーカイブスキームを実装しており、アーカイブする WAL セグメントごとに `archive_command` が順次実行されます。

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 、私たちの経験から、オペレーターが設定する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で複数の 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ーは、PostgreSQL が要求したものを含む最大 8 つの準備ができた 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

ベースバックアップを行うには、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り、データベースの実際のワークロードと I/O 競合が発生する可能性があります。

このため、CloudNativePG を使用すると、PostgreSQL で直接利用できる機能を利用できます。**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` の最もアライメントされたレプリカで実行されます。利用可能なレプリカがない場合、バックアップはプライマリインスタンスで実行されます。

注釈: スタンバイは常にプライマリと最新の状態であるとは限りませんが、最初に利用可能なバックアップから最後にアーカイブされた WAL までの連続した時間では、これは通常無関係です。ベースバックアップは、実際に、PITR を含むリカバリ操作を開始する開始点を表します。 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* ）>` で起こることと同様に 、スタンバイからバックアップするとき、プライマリで WAL の切り替えを強制しません。これにより、書き込みアクティビティが少ないデプロイメントで短期間（ *archive_timeout* がキックインする前）に予期しない結果が生じる場合があります。

常にプライマリでバックアップを実行したい場合は、以下の例に概要を示すように、バックアップターゲットを `primary` に設定できます。

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  [...]
spec:
  backup:
    target: "primary"
```

バックアップターゲットが prefer-standby に設定されている場合、このようなポリシーにより、バックアップは最新の利用可能なセカンダリインスタンスで実行されるか、他のインスタンスが利用できない場合はプライマリインスタンスで実行されます。

デフォルトでは、特に指定がない場合、ターゲットはスタンバイからバックアップを取得す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。

```yaml
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る場合でも、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「インプレース」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、以前に作成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）および barman-cloud-wal-restore ツール（WAL ファイル用、要求に応じてパラレルサポートを含む）を使用して、回復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

動作するため、.spec.postgresql.parameters に関しては、リカバリ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。

フードの下では、オペレーターは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 コンテナーはオブジェクトストレージからバックアップの回復を開始します。

重要: 新しい PVC 内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、ならびに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リカバリプロセスが完了すると、オペレーターは 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、ライブネスプローブによると、ポッドは正常です。`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、このフェーズをスピードアップできます）。

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、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。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、オペレーター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が通常どおりに起動され、残りのインスタンスが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とって透過的であり、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管理されます。

バックアップセクションを備えたクラスターへの復元

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、新しいクラスターが、回復後に WAL のアーカイブとバックアップの作成を開始することを意味します（そのように構成されている場合）。

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る可能性があり、`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。回復されたクラスターのベースバックアップと WAL が保存されます。

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

(次のページに続く)

(前のページからの続き)

```
      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ことを保証するための安全性チェックを含めます。既存のストレージを上書きするクラスターは、Pod が Error 状態で `Setting up primary` 状態のままになります。ポッド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いた **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、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 日間の保持ポリシーを使用して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

CloudNativePG は、デフォルトで、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

*サクサク

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 *DataBackupConfiguration* および *WalBackupConfiguration* セクションを参照してください。

アーカイブ時間、復元時間、およびサイズはアルゴリズム間で異なることに注意することが重要です。そのため、ユースケース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の評価を行いました。次の表は、ローカル MinIO 展開でバックアップが実行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 ratio
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、タグ付けのサポートが導入されました。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 を使用すると、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、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` 定義では、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と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履歴ファイルに追加されるキーと値のペアタグ

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 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ための次の 2 つのファイルの存在を認識しています。

- `postgresql.conf` ：PostgreSQL のメイン実行時構成ファイル
- `pg_hba.conf` ：クライアント認証ファイル

PostgreSQL コンテナの宣言的構成と不変性の概念により、ユーザーはこれらのファイルに直接触れ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できます。

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命令的な方法で変更しないでください。通常オペレーターによって制御されるオプションのいくつかを変更すると、実際にクラスターの予測不能/回復不能な状態につなが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のリファレンスは、サンプルに含まれています。を参照してください。

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

  # Parameters and pg_hba configuration will be append
  # to the default ones to make the cluster work
  postgresql:
```

(次のページに続く)

(前のページからの続き)

```
    parameters:
      max_worker_processes: "60"
    pg_hba:
      # To access through TCP/IP you will need to get username
      # and password from the secret cluster-example-custom-app
      - host all all all md5

  # 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 per instance using default storage class
  storage:
    size: 1Gi
```

。

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` ファイルで起動します。このファイルには、次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` セクションのユーザー定義設定が含まれます。

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

`custom.conf` の内容は、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ー
- 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ー
- ユーザー指定のパラメーター
- 固定パラメーター

グローバルデフォルトパラメーター は次のとおりです。

```
dynamic_shared_memory_type = posix
logging_collector = on
log_destination = csvlog
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、サーバーバージョンに応じて、 *wal_keep_size* または `wal_keep_segments` を適切に構成することはあなたの義務です。

または、ストリーミングレプリケーションクライアントのみ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、レプリケーションスロット機能を利用できます。これにより、クラスターレベルでレプリケーションスロットのサポートが追加されます。`replicationSlots.highAvailability` オプションでこの機能を有効にできます（詳細については、*ReplicationSlotsConfiguration* を参照してください）。

レプリケーションスロットも連続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、スタンバイの同期が失われるのを防ぐ唯一の方法です。スタンバイが同期しなくなると、次のようなエラーメッセージが生成されます。`"could not receive data from WAL stream: ERROR: requested WAL segment **** **** **** **** **** **** has already been removed"` 。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。

次のパラメーターは **固定**され、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
```

(次のページに続く)

(前のページからの続き)

```
ssl = on
ssl_ca_file = /controller/certificates/client-ca.crt
ssl_cert_file = /controller/certificates/server.crt
ssl_key_file = /controller/certificates/server.key
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ーは、正しい WAL アーカイブとレプリケーションに必要です。

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 に CSV 形式でログを出力することを要求し、インスタンスマネージャーは自動的に解析して 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プリロードする 1 つ以上の共有ライブラリをコンマ区切りのリストの形式で指定するために存在します。通常、PostgreSQL で使用され、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できる必要がある拡張機能をロードします（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` のコンテンツをオーバーライドできますが、専門の Postgres ユーザーのみがこのオプションを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 は自己修復の試行を妨げ、手動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、 *shared_preload_libraries* の拡張機能と設定の両方を常にテストしてください。

CloudNativePG は、最も使用されている PostgreSQL 拡張機能のいくつかの shared_preload_libraries オプションのコンテンツ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マネージドライブラリのいずれかを必要とすることに気付くとすぐに、必要なライブラリ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いとすぐに、ライブラリを削除します。

重要: *shared_preload_libraries* からライブラリを削除するには、有効にするにはクラスター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常に注意してください。

.spec.postgresql.shared_preload_libraries を介して、文字列のリストとして追加の 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されたように、CloudNativePG は、いくつかのよく知られてサポートされている拡張機能の shared_preload_libraries のコンテンツを自動的に管理します。現在のリストには以下が含まれます。

- auto_explain
- pg_stat_statements
- pgaudit
- pg_failover_slots

これらのライブラリの一部は、使用する前にデータベース内の追加オブジェクトを必要とします。通常、データベースで実行する CREATE EXTENSION コマンドで管理されるビューおよび/または関数（DROP EXTENSION コマンドは通常、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の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* のようなテンプレートデータベースも含まれています。

---

auto_explain を有効にする

:ref:``auto_explain` を有効にする<*auto_explain* を有効にする>`

拡張機能は、 EXPLAIN （最適化されていないクエリを追跡するのに役立ちます）を手動で実行せずに、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。

auto_explain を有効にするには、次の例の抜粋（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）のように、auto_explain. で始まるパラメーターを構成に追加します。

```
# ...
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 を有効にする

:ref:``pg_stat_statements` を有効にする<*pg_stat_statements* を有効にする>`

拡張機能は、クエリのリアルタイムモニタリングのために PostgreSQL で利用できる最も重要な機能の 1 つです。

次の抜粋例のように、 pg_stat_statements. で始まるパラメーターを構成に追加することにより、pg_stat_statements を有効にできます。

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

前述のように、オペレーターは pg_stat_statements を shared_preload_libraries に自動的に追加し、各データベースで CREATE EXTENSION IF NOT EXISTS pg_stat_statements を実行し、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 を有効にする

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監査ロギングを提供します。

CloudNativePG は、PostgreSQL クラスターでの *PGAudit ログ* を透過的かつネイティブにサポートしています。詳細については、
PGAudit ログ を参照してください

次の抜粋例のように、 pgaudit. で始まるパラメーターを構成に追加することにより、pgaudit を有効にできます。

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

pg_failover_slots を有効にする

:ref:``pg_failover_slots` を有効にする<*pg_failover_slots* を有効にする>`

EDB による拡張により、論理レプリケーションスロットがフェールオーバーシナリオに耐えられることが保証されます。フェールオーバーは通常、CloudNativePG の場合のように、物理ストリーミングレプリケーションを使用して実装されます。

pg_failover_slots. で始まるパラメーターを構成に追加することにより、pg_failover_slots を有効にできます。上記で説明したように、オペレーターはこれに応じて shared_preload_libraries オプションの pg_failover_slots エントリーを透過的に管理します。

状態の同期 を参照してください

この拡張機能の詳細については。

さらに、pg_failover_slots で使用する予定のデータベースごとに、各レプリカがプライマリに接続できるようにするエントリーを pg_hba セクションに追加する必要があります。たとえば、pg_failover_slots で app データベースを使用する場合は、 pg_hba セクションにこのエントリを追加する必要があります。

```
postgresql:
  pg_hba:
    - hostssl app streaming_replica all cert
```

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 の作成に使用される PostgreSQL ホストベースの認証ルールのリストです。

最初の一致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、4 つのセクションで構成されていると見ることができます。

1. 固定ルール

2. ユーザー定義ルール

3. オプションの LDAP セクション

4. デフォルトルール

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 から、password_encryption データベースパラメーターのデフォルト値は scram-sha-256 に設定されます。そのため、この PostgreSQL バージョンからデフォルトの認証方法は 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>

  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。
```

クラスターマニフェスト内で、次の抜粋に示すように、 pg_hba 行が spec.postgresql.pg_hba のリスト項目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、 app ユーザーが 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、安全なチャネル（hostssl ）を介して app データベースへのアクセスを有効にしています。

LDAP 構成

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可能なオプションの ldap セクションがあります。

これは 2 つのモードをサポートします。LDAP セクションで server 、 prefix 、および suffix を指定する必要がある simple bind モードと、 server 、 baseDN 、 binDN 、およびシークレットである ldap パスワードを含む bindPassword を指定する必要がある search+bind モード。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 が指定されない場合、 uid のデフォルトが使用されます。

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 で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 セクションを編集して、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リロード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 を介した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自分自身を制限することをお勧めします。

- `posix` ： `shm_open` を使用して割り当てられた POSIX 共有メモリに依存します（デフォルト設定）
- `sysv` ：`shmget` を介して割り当てられた 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とって特に重要です。詳細はこちら

*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

オペレーターが/dev/shm の下の `shm` と呼ばれる**メモリバインドされた** ``*EmptyDir*`` **ボリューム**を自動的にマウントすることを考慮すると、ほとんどの場合、`posix` のデフォルト設定で十分です。次の方法で、実行中の Postgres コンテナ内のそのようなボリュームのサイズを確認できます。

```
mount | grep shm
```

次の出力のようなもの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` および `SHMALL` パラメーターの値が十分に高い場合、以下を設定することもできます。

```
dynamic_shared_memory_type: "sysv"
```

次を実行して、PostgreSQL コンテナ内から `SHMMAX` /`SHMALL` を確認できます。

```
ipcs -lm
```

例：

```
- ----- 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、非常に多数の max total shared memory は、dynamic_shared_memory_type を sysv に設定することを推奨しています。

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 固定パラメーター

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 データベースロール管理

CloudNativePG は当初から、PostgreSQL インスタンスで必要な特定のロールの作成を管理してきました。

- postgres スーパーユーザー、streaming_replica 、cnpg_pooler_pgbouncer などの一部の予約ユーザー（PgBouncer Pooler を使用する場合）
- アプリケーションデータベースの低権限所有者として設定されたアプリケーションユーザー

このプロセスについては、 *BootstrapConfiguration* セクションで説明されています。

クラスター仕様の managed スタンザにより、CloudNativePG は、.spec.managed.roles で指定されたロールの完全なライフサイクル管理を提供するようになりました。

この機能により、既存のロールの宣言的管理と、新しいロールがデータベースにまだ存在しない場合は作成できます。ロールの作成は、データベースのブートストラップが完了した**後に**発生します。

宣言的なロール管理を備えたクラスターのマニフェストの例は、次のファイルにあります。

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

(次のページに続く)

(前のページからの続き)

```
  storage:
    size: 1Gi

  managed:
    roles:
    - name: app
      createdb: true
    - name: dante
      ensure: present
      comment: my database-side comment
      login: true
      superuser: false
      createdb: true
      createrole: false
      inherit: false
      replication: false
      bypassrls: false
      connectionLimit: 4
      validUntil: "2053-04-12T15:04:05Z"
      inRoles:
        - pg_monitor
        - pg_signal_backend
      passwordSecret:
        name: cluster-example-dante
- --
apiVersion: v1
data:
  username: ZGFudGU=
  password: ZGFudGU=
kind: Secret
metadata:
  name: cluster-example-dante
type: kubernetes.io/basic-auth
```

。

これはそのファイルからの抜粋です。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
spec:
  managed:
    roles:
    - name: dante
      ensure: present
```

(次のページに続く)

(前のページからの続き)

```
    comment: Dante Alighieri
    login: true
    superuser: false
    inRoles:
      - pg_monitor
      - pg_signal_backend
```

`spec.managed.roles` のロール仕様は PostgreSQL structure and naming conventions に従います。各ロールに定義できる属性の完全なリストについては、

> *API リファレンス* を参照してください。

いくつかの点に注意してください。

1. `ensure` 属性は、PostgreSQL の **一部ではありません**。これにより、宣言的なロール管理でロールを作成および削除できます。2 つの可能な値は、`present` （デフォルト）と `absent` です。
2. `inherit` 属性は、PostgreSQL の規則に従って、デフォルトで true です。
3. PostgreSQL の規則に従って、 `connectionLimit` 属性のデフォルトは-1 です。
4. `inRoles` を持つロールメンバーシップは、デフォルトでメンバーシップなしになります。

宣言的なロール管理により、PostgreSQL インスタンスが仕様に準拠していることを保証します。ユーザーがデータベースでロール属性を直接変更した場合、CloudNativePG オペレーターは次の調整サイクル中にそれらの変更を元に戻します。

4.17.1 パスワード管理

宣言的なロール管理機能には、ロールパスワードの調整が含まれます。パスワードは、Kubernetes の世界と PostgreSQL では根本的に異なる方法で管理されるため、いくつか注意すべき点があります。

管理対象ロールの構成では、オプションで、ユーザー名とパスワードが保存される**シークレット**の名前を指定できます（Kubernetes での通常のように Base64 でエンコードされます)。例：

```
managed:
  roles:
  - name: dante
    ensure: present
    [··· snipped ···]
    passwordSecret:
      name: cluster-example-dante
```

これは、ユーザー名とパスワードを含む `cluster-example-dante` と呼ばれる Secret の存在を前提としています。ユーザー名は、パスワードを設定しているロールと一致する必要があります。例：

```
apiVersion: v1
data:
```

(次のページに続く)

(前のページからの続き)

```
  username: ZGFudGU=
  password: ZGFudGU=
kind: Secret
metadata:
  name: cluster-example-dante
  labels:
    cnpg.io/reload: "true"
type: kubernetes.io/basic-auth
```

ロールに passwordSecret が指定されていない場合、インスタンスマネージャーはパスワードを使用してロールを作成/変更しようとしません。これは PostgreSQL の慣習に従っており、WITH PASSWORD で指示されない限り ALTER はパスワードを更新しません。

ロールが最初にパスワードで作成され、PostgreSQL でパスワードを NULL に設定したい場合、これは CloudNativePG のユーザーの側で明示的にする必要があります。「仕様でパスワードが提供されていない」と「パスワードを NULL に設定」を区別するには、フィールド DisablePassword を使用する必要があります。

データベースの dante ロールにパスワードを設定しないと決めたとします。そのような場合、以下を指定します。

```
managed:
  roles:
  - name: dante
    ensure: present
    [··· snipped ···]
    disablePassword: true
```

注：特定のロールに passwordSecret と disablePassword の両方を設定するとエラーと見なされます。この構成は、検証 Webhook によって拒否されます。

警告: 宣言的なロール管理機能は、初期バージョン（1.20.0）から動作が変更されました。 1.20.0 では、*passwordSecret* のないロールは、PostgreSQL でパスワードを NULL に設定することになりました。実際には 1.20.0 とほとんど違いはありません。 *passwordSecret* なしで作成された新しいロールのパスワードは NULL になります。関連する変更は、管理対象ロールを使用して、以前に作成されたロールを管理する場合です。 1.20.0 では、これを行うと、誤って既存のパスワードが NULL に設定される場合があります。

4.17.2 実現不可能なロール構成

PostgreSQL では、場合によっては、データベースがコマンドを受け入れることができず、拒否されます。
PostgreSQL documentation on error codes を参照してください

詳細については。

ロール操作は、このような基本的なエラーを生成する可能性があります。2 つの例：

- PostgreSQL にロール（グループ）poets のメンバーとしてロール petrarca を作成するように依頼しますが、poets は存在しません。
- PostgreSQL にロール dante を削除するように依頼しますが、ロール dante はデータベース inferno の所有者です。

これらの基本的なエラーは、人間の管理者からの明確化なしに、データベースも CloudNativePG オペレーターも修正できません。上記の 2 つの例は、ロール poets を作成するか、データベース inferno をそれぞれ削除することで修正できますが、それらは人為的エラーが原因である可能性があり、そのような場合、提案された「修正」は間違ったことである可能性があります。

CloudNativePG は、このような基本的なエラーが発生したときに記録し、クラスターのステータスに表示します。続くのは…

4.17.3 管理対象ロールのステータス

以下に示すように、CRD ステータスには、管理対象ロールのステータスのセクションが含まれます。

```
status:
  [···snipped···]
  managedRolesStatus:
    byStatus:
      not-managed:
      - app
      pending-reconciliation:
      - dante
      - petrarca
      reconciled:
      - ariosto
      reserved:
      - postgres
      - streaming_replica
    cannotReconcile:
      dante:
      - could not perform DELETE on role dante: owner of database inferno
      petrarca:
      - could not perform UPDATE_MEMBERSHIPS on role petrarca: role "poets" does not␣
↪exist
```

データベース（および CloudNativePG）が尊重できず、人間の介入が必要なオペレーションの特別なサブセクション `cannotReconcile` に注意してください。

このセクションでは、オペレーターの使用のために予約されているロールと、宣言管理下にないロールについて説明し、データベースインスタンス内のロールの包括的なビューを提供します。

重要: 下位互換性の観点から、宣言型ロール管理は、データベースに存在するが仕様に含まれていないロールを無視するように設計されています。これらのロールのライフサイクルは引き続き PostgreSQL 内で管理されるため、CloudNativePG ユーザーは都合の良いときにこの機能を採用できます。

4.18 オペレーター設定

CloudNativePG のオペレーターは、標準の展開マニフェストからインストールされ、構成パラダイムよりも規約に従います。ほとんどの場合、これで問題ありませんが、次のようなデフォルトの動作を変更するシナリオがいくつかあります。

- オペレーターが作成し、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よって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-system` 名前空間に `cnpg-controller-manager` と呼ばれる Kubernetes `Deployment` としてインストールされます。

注釈: 以下の例では、オペレーター展開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、オペレーターデプロイメントと同じ名前空間にあり、名前として `cnpg-controller-manager-config` を使用する `ConfigMap` /`Secret` を介してカスタマイズできます。

重要: 構成の `ConfigMap` /*Secret* に対する変更は、オペレーターによって自動的に検出されません。そのため、リロードする必要があります（以下を参照）。さらに、変更は、構成がリ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 の値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 内のものが使用されます。

4.18.1 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た場合、ポッドを含む生成されたすべてのリソースによって継承される注釈名のリスト INHERITED_LABELS | Cluster メタデータで定義された場合、ポッドを含む生成されたすべての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 ENABLE_AZURE_PVC_UPDATES | PVC がサイズ変更条件でスタックしている場合、Postgres ポッドを削除できます。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の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 ）作成されたすべてのクラスター MONITORING_QUERIES_SECRET | オペレーターの名前空間内の ConfigMap の名前とデフォルトのクエリのセット（キー queries で指定）（キー queries で指定）を使用します。作成されたすべてのクラスター CREATE_ANY_SERVICE | オペレーターの名前空間内の Secret の名前とデフォルトのクエリのセット（キー queries の下で指定されます） true に設定すると、クラスターの-any サービスが作成されます。デフォルトは false です

INHERITED_ANNOTATIONS および 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。そのシークレットの名前は<cluster-name>-pull 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られない場合、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、動作が以前の説明と一致するように変更されました。オペレーターの以前のバージョン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。

4.18.2 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るリソースによって継承されるラベル/注釈名を定義し、in-place updates for the instance manager を有効にすることにより、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:
```

(次のページに続く)

(前のページからの続き)

```
  INHERITED_ANNOTATIONS: categories
  INHERITED_LABELS: environment, workload, app
  ENABLE_INSTANCE_MANAGER_INPLACE_UPDATES: true
```

4.18.3 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るリソースによって継承されるラベル/注釈名を定義し、in-place updates for the instance manager を有効にすることにより、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
```

4.18.4 オペレーターを再起動して構成をリロードする

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、次を発行してそれを行うことができます。

```
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たら、次のコマンドで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 アノテーションと 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。

4.18.5 PPROF HTTP サーバー

オペレーターは、localhost:6060 で次のエンドポイントを使用して 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 args を下にスクロールし、--pprof-server=true を追加します。例：

```
     containers:
     - args:
       - controller
       - --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、デプロイメント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内で exec できます。
```

kubectl exec -ti -n cnpg-system – bash

curl localhost:6060/debug/pprof/ ```

4.19 インスタンスポッド構成

4.19.1 予想されるボリューム

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 予想されるボリューム テンプレートであり、これにより、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。

これは、既存の tls Secret（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に関する簡単な例です。sample-secret の秘密鍵 `tls.crt` および `tls.key` の値は、Postgres ポッドのパス`/projected/certificate/tls.crt` および`/projected/certificate/tls.key` にファイルとしてマウントされます。

```
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
            - key: tls.key
              path: certificate/tls.key
  storage:
    size: 1Gi
```

プロジェクションボリューム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、

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

(次のページに続く)

(前のページからの続き)

```
    sources:
      - secret:
          name: sample-secret
          items:
            - key: tls.crt
              path: certificate/tls.crt
            - key: tls.key
              path: certificate/tls.key
      - configMap:
          name: sample-configmap
          items:
            - key: key1
              path: config/key1
            - key: key2
              path: config/key2
  storage:
    size: 1Gi

- --
apiVersion: v1
data:
  tls.crt:␣
↪LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNqekNDQWVXZ0F3SUJBZ0lRSHVDY2lKcDVkZis0dHZBcDBrTk9rekFLQmdr
  tls.key:␣
↪LS0tLS1CRUdJTiBFQyBQUklWQVRFIEtFWS0tLS0tCk1IY0NBUUVFSUtucWxVQjFqU2sxWW5VZzAyb0tGbXlRdDJLUEZwaFc0K1lm
kind: Secret
metadata:
  name: sample-secret
type: kubernetes.io/tls
- --
apiVersion: v1
data:
  key1: value1
  key2: value2
  key3: value3
kind: ConfigMap
metadata:
  name: sample-configmap
```

配置マニフェスト。

4.19.2 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。一例は 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。別の例は TZ 環境変数で、PostgreSQL コンテナが使用するタイムゾーンを表します。

CloudNativePG を使用すると、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、デフォルトのクラスターレベルのタイムゾーンとして Australia/Sydney タイムゾーンを使用する 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、ConfigMap または Secret を参照して、そ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:
  - configMapRef:
      name: config-map-name
  - secretRef:
      name: secret-name

  storage:
    size: 1Gi
```

オペレーターは、次の環境変数を設定すること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 セクションを変更すると、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 Secret または ConfigMap を参照している場合、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 と同じ動作を使用し、ユーザーは Pod のロールアウトを手動でトリガーすることになっています。

4.20 ストレージ

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あり、拡張性があり、パフォーマンスが高く、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コンテキストでも有効です。

重要: Kubernetes には、動的にプロビジョニングされるストレージに関して、独自の特異性があります。これらには、**ストレージクラス**、**永続ボリューム**、および*永続ボリューム要求*が含まれます。 VM と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。

ストレージへのアクセスには、主に 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（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ます（これには、Kubernetes のベアメタルインストールに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には、従来の環境で発生するのと同じスループットとレイテンシーの問題があります。これらは、複数のアプリケーションとの I / O 競合がパフォーマンス結果の変動性を高め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は、シェアードナッシングアーキテクチャを有効にします。これは、より高い予測可能なパフォーマンスを保証するため、高トランザクションおよび Very Large Database（VLDB）ワークロードにより適しています。

> **警告:** CloudNativePG で 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。私たちのアドバイスは、最初に *fio* 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* 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です。

注釈: CloudNativePG は、データの永続性を管理するために `StatefulSet` s を使用しません。むしろ、永続的なボリュームクレーム（PVC）を直接管理します。もっと詳しく知りたい場合は、 **カスタムポッドコン**

トローラー ドキュメントをお読みください。

4.20.1 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 をベンチマーク評価するためのオープンソースのガイドラインと Helm チャートのオープンソースセットである
cnp-bench を維持します。

簡単に言えば、 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、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 とともに配布されるデフォルトのベンチマークツールを使用したデータベースのパフォーマンスの測定：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、データベースが実稼働に入る前に**行う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（容量計画など）だけでなく、生産ライフサイクル、特に緊急事態（この種のテストを実行する余裕がなくなった場合）でも非常に貴重です。データベースは実際に時間の経過とともに変化および進化します。データの分散もそうであり、パフォーマンスに影響を与える可能性があります。シーケンシャル読み取りまたはシーケンシャル書き込みの理論上の最大スループットを知ることは、これらの状況で非常に役立ちます。特に、外部ワークロードの影響によって結果が変わらないシェアードナッシングコンテキストでは。 **システムを知り、ベンチマークします。**

4.20.2 保存時の暗号化

CloudNativePG を使用すると、保存時の暗号化が可能です。オペレーターは、それを基になるストレージクラスに委任します。この重要なセキュリティ機能については、ストレージクラスを参照してください。

4.20.3 永続ボリュームクレーム

オペレーターは、 PGDATA を保存することを目的として、PostgreSQL インスタンスごとに永続的ボリュームクレーム（PVC）を作成し、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も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。

4.20.4 ストレージクラスを介した構成

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、本番環境にデプロイする前に、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。

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
```

前の構成を使用して、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。ターゲット 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、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
```

4.20.5 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
```

4.20.6 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 として知られています）であり、データベースで発生したトランザクションの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。

注釈: 通常、各セグメントのサイズは 16MB ですが、 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* ）>` で説明されているように、サイズはクラスター初期化時に適用される *walSegmentSize* オプションを介して構成できます。

ほとんどの場合、 PGDATA が存在する同じボリュームに pg_wal を置くことは問題ありませんが、WAL を別のボリュームに保存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**：WAL ファイルを PGDATA とは異なるストレージに保存することにより、PostgreSQL は WAL 操作（通常はシーケンシャル書き込み）およびデータファイル（テーブルとインデックスなど）に並列 I / O を活用できるため、垂直方向が向上しますスケーラビリティ
- **より高い信頼性**：WAL ファイル専用のディスク領域を予約することにより、 PGDATA ボリュームの領域の枯渇が WAL の書き込みを妨げることはなく、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保証できます。
- **より細かい制御** ：PGDATA と pg_wal の両方専用のスペース量を定義し、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 モニタリング** ： PGDATA と pg_wal の両方で負荷とディスク使用量を常に監視し、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、公式の PostgreSQL ドキュメントから。

.spec.walStorage オプションを介して WAL 用の別個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されているのと同じルールに従い、専用の PVC をプロビジョニングします。例：

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: separate-pgwal-volume
spec:
  instances: 3
  storage:
    size: 1Gi
  walStorage:
    size: 1Gi
```

重要: *walStorage* の削除はサポートされていません。追加すると、WAL 用の別個のボリュームを既存の Postgres クラスターから削除できません。

4.20.7 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、オペレーターはすべての PVC に変更を適用します。

`StorageClass` が *online volume resizing* をサポートしている場合

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、ポッドを削除してサイズ変更をトリガーする必要があります。

続行するための最良の方法は、レプリカから始めて、各ポッドがバックアップされるのを待って、一度に 1 つのポッドを削除することです。

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、 **オペレーター設定** の `ENABLE_AZURE_PVC_UPDATES` 環境変数を使用してこの制限を克服しています。 `'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、以下の回避策に従って、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 上の PVC のサイズを手動でサイズ変更できます。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

`docs <https://github.com/kubernetes-sigs/azuredisk-csi-driver/blob/master/docs/known-
↪issues/sizegrow.md>`__ で説明されているように、Azure ディスクは「非接続」状態でのみ拡張できます。
```

つまり、PostgreSQL クラスターが使用するディスクのサイズを変更するには、手動ロールアウトを実行し、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遮断する必要があります。これにより、オペレーターはバックグラウンドのディスクサイズ変更が完了する前に Pod を再作成し、すぐに PVC に再接続できなくなります。

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、最初にレプリカを処理できます。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、ノードのコルドンを解除できます。

```
kubectl uncordon <node of cluster-example-3>
```

Pod が正しく再作成され、 Running and Ready 状態になるのを待ちます。

```
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
```

次のコマンドを実行して、PVC 拡張を確認します。設定どおりに「2Gi」が返されます。

```
kubectl get pvc cluster-example-3 -o=jsonpath={.status.capacity.storage}
```

そのため、残りのポッド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* を使用して、新しいサイズ変更された Pod をスイッチオーバーでプロモートした後、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のままにしてください（例、 *cluster-example-3* をプライマリにプロモートする `kubectl cnpg promote cluster-example 3` ）。

ストレージの再作成

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て新しい PVC を割り当て、そこにデータベースを移動することにより、別の PVC でクラスターを再生成できます。この操作は、クラスターに複数のノードが含まれる場合にのみ実行できます。

それをしている間、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
```

(次のページに続く)

(前のページからの続き)

```
spec:
  instances: 3

  storage:
    storageClass: standard
    size: 1Gi
    resizeInUseVolumes: False
```

クラスター全体を別のストレージ領域に移動するには、すべての PVC とすべての Pod を再作成する必要があります。次の例のような 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
```

異なる 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` を無効にしてから、すべてのインスタンスを別の PVC に再作成します。

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、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。これは、 *.spec.walStorage* セクションに対しても `resizeInUseVolumes` を無効にすることで実行できます。

例えば（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
```

それが終わったら、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
```

4.20.8 永続ボリュームの静的プロビジョニング

CloudNativePG は動的ボリュームプロビジョニングと連携するように設計されています。これにより、ユーザーが要求したときに、前述のストレージクラスと永続ボリュームクレームテンプレートを介して、ストレージボリュームをオンデマンドで自動的に作成できます。

ただし、場合によっては、Kubernetes 管理者は、新しいストレージボリュームを**手動で**作成し、Kubernetes クラスター内での表現に関連する `PersistentVolume` オブジェクトを作成します。これは、ボリュームの**事前プロビジョニング** とも呼ばれます。

重要: オペレーターの高可用性と自己修復機能に影響を与え、CloudNativePG が構築されている完全な宣言モデルを壊すため、ボリュームの事前プロビジョニングを避けることをお勧めします。

次の手順に従って、CloudNativePG で事前にプロビジョニングされたボリュームを使用できます。

1.Kubernetes の外部にボリュームを手動で作成する

2. 実際の CSI ドライバー（つまり、 `volumeHandle` 、`fsType` 、`storageClassName` など）で必要とされる正しいパラメーターを使用して、上記のボリュームと一致する `PersistentVolume` オブジェクトを作成します

3. ストレージセクションごとに、コヒーレントな *pvcTemplate* を使用して Postgres `Cluster` を作成します

Kubernetes が上記の `PersistentVolume` と一致し、CloudNativePG が必要な `PersistentVolumeClaim` を作成できるようにするセクション

警告: 静的プロビジョニングでは、クラスターのアフィニティルールに基づいて、事前にプロビジョニングされたボリュームが存在する Kubernetes によって Postgres ポッドを正しくスケジュールできることを保証するのはあなたの責任です。クラスターをデプロイした後、ポッドが `Pending` でスタックしていないかどうかを確認し、状態が解決しない場合は、なぜこれが起こっているのかを調査してください。

4.21 ラベルとアノテーション

Kubernetes のリソースはフラット構造で編成され、それらの間に階層情報や関係はありません。ただし、そのようなリソースとオブジェクトは、**ラベル** と **注釈** を介してリンクし、関係付けることができます。

注釈: 詳細については、 ラベルとアノテーション および ラベルとアノテーション に関する Kubernetes ドキュメントを参照してください。

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ことを目的として、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

- ラベルは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照会するために使用されます

CloudNativePG デプロイメントで使用する 1 つ以上のラベルおよび/またはアノテーションを選択できます。次に、クラスターのメタデータでこれらのラベルおよび/または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、Pod テンプレートなどの代替アプローチの代わりに CloudNativePG が採用した手法です。

4.21.1 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継承されません。ラベル/アノテーションの継承を有効にするには、
オペレーター設定 セクションの指示に従う必要があります。

以下では、その例を続行し、以下に制限します。

- 注釈：`categories`
- ラベル：`app` 、`environment` 、および `workload`

注釈: 注釈とラベルの両方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。命名にワイルドカードを使用し、 *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できることに注意してください。

4.21.2 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。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
  annotations:
    categories: database
  labels:
    environment: production
    workload: database
    app: sso
```

(次のページに続く)

(前のページからの続き)

```
spec:
    # ... <snip>
```

クラスターがデプロイされたら、たとえば、次のコマンドを使用して、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
```

4.21.3 現在の制限

現在、CloudNativePG はラベルまたはアノテーションの削除を自動的に伝播しません。したがって、以前に基になるポッドに伝播された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ると、オペレーターは関連付けられたリソースから自動的に削除しません。

4.22 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を超えています。それらがシステムに正しくインストールされていると仮定します。ただし、実験のために、 *Part 4 of the Quickstart* で手順を提供します。

4.22.1 モニタリングインスタンス

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 を介して *Prometheus* **オペレーターの例** 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** と、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ための高度に構成および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 **ユーザー定義のメトリック** を参照）。

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。

注釈: **エクスポートされたメトリックを検査する方法** の指示に従って、エクスポートされたメトリックを検査できます

以下のセクション。

PostgreSQL で実行されるすべてのモニタリング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「デフォルトのロール」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 メソッドで定義された**メインデータベース**に対して実行されます。

- initdb を使用：クエリはデフォルトで initdb.database 、または指定されていない場合 app の指定されたデータベースに対して実行されます
- recovery を使用：クエリはデフォルトで recovery.database 、または指定されていない場合 postgres の指定されたデータベースに対して実行されます
- pg_basebackup を使用：クエリはデフォルトで pg_basebackup.database 、または指定されていない場合 postgres の指定されたデータベースに対して実行されます

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、デフォルトのデータベースを特定のユーザー定義メトリックに対していつでもオーバーライドできます。

Prometheus 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、クラスターリソース自分自身で.spec.monitoring.enablePodMonitor を true に設定することにより、オペレーターが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。カスタマイズが必要な場合は、以下に説明するように行うことができます。

特定のクラスターに 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
```

(次のページに続く)

(前のページからの続き)

```
  podMetricsEndpoints:
  - port: metrics
```

重要: 一意の名前、正しいクラスターの名前空間とラベルを使用して、上記の例を変更してください（*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たラベル`postgresql` は非推奨であり、将来的に削除されます。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。これは、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。
- 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、ならびに予想および実際に観測された値- インスタンスを収容する個別のノードの数-最後に失敗したバックアップと使用可能な最後のバックアップ、およびクラスターの最初の回復可能性を示すタイムスタンプ- レプリカクラスターモードが有効または無効かどうかを示すフラグ - 手動スイッチオーバーが必要かどうかを示すフラグ - フェンシングが有効か無効かを示すフラグ
- go_* で始まる Go ランタイム関連のメトリック

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 フォーマットは自己文書化です。

```
#  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_fencing_on 1 if the instance is fenced, 0 otherwise
```

(次のページに続く)

(前のページからの続き)

```
#  TYPE cnpg_collector_fencing_on gauge

cnpg_collector_fencing_on 0

#  HELP cnpg_collector_nodes_used NodesUsed represents the count of distinct nodes␣
↪accommodating the instances. A value of -1 suggests that the metric is not␣
↪available. A value of 1 suggests that all instances are hosted on a single node,␣
↪implying the absence of High Availability (HA). Ideally this value should match the␣
↪number of instances in the cluster.

#  TYPE cnpg_collector_nodes_used gauge

cnpg_collector_nodes_used 3

#  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

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)
```

(次のページに続く)

(前のページからの続き)

```
#  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="15.3"} 15.3

#  HELP cnpg_collector_last_failed_backup_timestamp The last failed backup as a unix␣
↪timestamp

#  TYPE cnpg_collector_last_failed_backup_timestamp gauge

cnpg_collector_last_failed_backup_timestamp 0

#  HELP cnpg_collector_last_available_backup_timestamp The last available backup as a␣
↪unix timestamp

#  TYPE cnpg_collector_last_available_backup_timestamp gauge
```

(次のページに続く)

(前のページからの続き)

```
cnpg_collector_last_available_backup_timestamp 1.63238406e+09

#  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 cnpg_collector_wal_buffers_full Number of times WAL data was written to disk␣
↪because WAL buffers became full. Only available on PG 14+

#  TYPE cnpg_collector_wal_buffers_full gauge

cnpg_collector_wal_buffers_full{stats_reset="2023-06-19T10:51:27.473259Z"} 6472

#  HELP cnpg_collector_wal_bytes Total amount of WAL generated in bytes. Only␣
↪available on PG 14+

#  TYPE cnpg_collector_wal_bytes gauge

cnpg_collector_wal_bytes{stats_reset="2023-06-19T10:51:27.473259Z"} 1.0035147e+07

#  HELP cnpg_collector_wal_fpi Total number of WAL full page images generated. Only␣
↪available on PG 14+

#  TYPE cnpg_collector_wal_fpi gauge

cnpg_collector_wal_fpi{stats_reset="2023-06-19T10:51:27.473259Z"} 1474

#  HELP cnpg_collector_wal_records Total number of WAL records generated. Only␣
↪available on PG 14+

#  TYPE cnpg_collector_wal_records gauge

cnpg_collector_wal_records{stats_reset="2023-06-19T10:51:27.473259Z"} 26178
```

(次のページに続く)

(前のページからの続き)

```
#  HELP cnpg_collector_wal_sync Number of times WAL files were synced to disk via␣
↪issue_xlog_fsync request (if fsync is on and wal_sync_method is either fdatasync,␣
↪fsync or fsync_writethrough, otherwise zero). Only available on PG 14+

#  TYPE cnpg_collector_wal_sync gauge

cnpg_collector_wal_sync{stats_reset="2023-06-19T10:51:27.473259Z"} 37

#  HELP cnpg_collector_wal_sync_time Total amount of time spent syncing WAL files to␣
↪disk via issue_xlog_fsync request, in milliseconds (if track_wal_io_timing is␣
↪enabled, fsync is on, and wal_sync_method is either fdatasync, fsync or fsync_
↪writethrough, otherwise zero). Only available on PG 14+

#  TYPE cnpg_collector_wal_sync_time gauge

cnpg_collector_wal_sync_time{stats_reset="2023-06-19T10:51:27.473259Z"} 0

#  HELP cnpg_collector_wal_write Number of times WAL buffers were written out to disk␣
↪via XLogWrite request. Only available on PG 14+

#  TYPE cnpg_collector_wal_write gauge

cnpg_collector_wal_write{stats_reset="2023-06-19T10:51:27.473259Z"} 7243

#  HELP cnpg_collector_wal_write_time Total amount of time spent writing WAL buffers␣
↪to disk via XLogWrite request, in milliseconds (if track_wal_io_timing is enabled,␣
↪otherwise zero). This includes the sync time when wal_sync_method is either open_
↪datasync or open_sync. Only available on PG 14+

#  TYPE cnpg_collector_wal_write_time gauge

cnpg_collector_wal_write_time{stats_reset="2023-06-19T10:51:27.473259Z"} 0

#  HELP cnpg_last_error 1 if the last collection ended with error, 0 otherwise.

#  TYPE cnpg_last_error gauge

cnpg_last_error 0

#  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
```

(次のページに続く)

(前のページからの続き)

```
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="go1.20.5"} 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

#  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␣
```

(次のページに続く)

(前のページからの続き)

```
↪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

#  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.
```

(次のページに続く)

(前のページからの続き)

```
#  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.

#  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.
```

(次のページに続く)

(前のページからの続き)

```
#  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␣
↪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* という名前のラベルフィール

ドの 1 つの中にあります。

注釈: *cnpg_collector_first_recoverability_point* および `cnpg_collector_last_available_backup_timestamp` は、オブジェクトストアへの最初のバックアップまでゼロです。これは WAL アーカイブとは別のものです。

ユーザー定義のメトリック

この機能は現在**ベータ**状態であり、形式は PostgreSQL Prometheus Exporter の queries.yaml file に触発されています。

ユーザーは、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` を参照することにより、カスタムメトリックを定義できます。

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

  storage:
    size: 1Gi

  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: ConfigMap とシークレットをインスタンスによって**自動的に**リロードしたい場合は、キー *cnpg.io/reload* を持つラベルを追加できます。そうでない場合、 *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を含む json 警告ログを出力します。*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* 。

ユーザー定義メトリックの例

ここでは、上記の Cluster の例で参照される、単一のカスタムクエリを含む ConfigMap の例を見ることができます。

```
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:
        - 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"
```

基本的なモニタリングクエリのリストは、 *default-monitoring.yaml*

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 オプションに複数のデータベースがリストされている場合、メトリックはそれぞれから収集されます。

target_databases のリストに**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展開します。

注釈: * 文字には yaml に special meaning があるため、そのようなパターンが含まれる場合は`target_databases`値をクォート（"*"）する必要があります。

たとえば、次の例のように current_database() ファンクションを使用して、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"
        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
```

これは、template1 データベースでも実行される自動検出を有効にしたクエリの例です（それ以外の場合、前述のクエリで返されません）。

```
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>"
```

利用可能なすべてのフィールドの簡単な説明は次のとおりです。

- `<MetricName>` ：プロメテウスメトリックの名前 - `query` ：メトリックを生成するためにターゲットデータベースで実行する 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 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。提供されている場合、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` 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 2: :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する
COUNTER	この列をカウンターとして使用する
GAUGE	この列をゲージとして使用する
MAPPEDMETRIC	この列を指定されたテキスト値のマッピングで使用します
DURATION	この列をテキスト期間として使用します（ミリ秒単位）
HISTOGRAM	この列をヒストグラムとして使用する
Metric Types をご覧ください	

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、次の形式で Prometheus エクスポーターエンドポイント（`:9187/metrics` ）によって返されます。

```
cnpg_<MetricName>_<ColumnName>{<LabelColumnName>=<LabelColumnValue> ... }
↪<ColumnValue>
```

注釈: *LabelColumnName* は、 *usage* が `LABEL` に設定されたメトリックとその `Value` 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
```

(次のページに続く)

(前のページからの続き)

```
#  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 Secret で定義された一連のモニタリングクエリをクラスターに自動的に注入するように構成できます。
オペレーター設定 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。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、それを使用する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。

オペレーターインストールマニフェストには、すべてのクラスターで使用される cnpg-default-monitoring と呼ばれる事前定義された ConfigMap が付属しています。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。
- 特定のクラスターで無効にします。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* を介して指定された ConfigMap または Secret は、常に固定名前 `cnpg-default-monitoring` で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ックを使用する場合は、クラスターの名前空間にこの名前の 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。

4.22.2 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 を介して *Prometheus* **オペレーターの例** メトリックを内部的に公開します。

注釈: **エクスポートされたメトリックを検査する方法** の指示に従って、エクスポートされたメトリックを検査できます

以下のセクション。

現在、オペレーターはデフォルトの `kubebuilder` メトリックを公開しています。詳細については、

> kubebuilder documentation を参照してください。

Prometheus オペレーターの例

オペレーターの展開は、次の PodMonitor を定義することにより、
Prometheus **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
```

4.22.3 エクスポートされたメトリックを検査する方法

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ックを検査する方法に関する基本的な手順を提供します。

注釈: 以下の例では、PostgreSQL クラスターと一緒にデフォルトの名前空間で作業していると仮定します。Kubernetes の基本的な知識を適用して、この例を自分の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、実行する汎用コマンドです（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、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ック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、 curl pod を削除してください。

```
kubectl delete -f curl.yaml
```

4.22.4 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、実稼働システムに対する推奨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。これにより、Prometheus がすべての PodMonitor リソースをリッスンします。
- `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 を見つけることが保証されます。

さらに、ご参考までに、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` の構成では、`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。

4.23 ロギング

オペレーターは、PostgreSQL ログを含む標準出力に JSON 形式で直接記録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` ：実際のメッセージ、またはメッセージが JSON 形式で解析される場合はキーワード `record`
- `record` ：実レコード（`logger` の種類によって構造が異なる）
- `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

ドキュメント。

注釈: *level* および `ts` フィールド名は、ログインジェストシステムで必要な場合、オペレーターコントローラーの `log-field-level` および `log-field-timestamp` フラグを介して名前を変更できます。 *cloudnative-pg* オペレーターの `Deployment` 定義を編集するだけです。

4.23.1 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。クラスターが起動された後にクラスター仕様で値が変更された場合、これは新しいポッドでのみ有効になり、古いポッドでは有効になりません。

4.23.2 PostgreSQL ログ

PostgreSQL ログの各エントリは、`logger` キーが `postgres` に設定され、次の例で説明する構造を持つ JSON オブジェクトです。

```
{
  "level": "info",
  "ts": 1619781249.7188137,
  "logger": "postgres",
```

(次のページに続く)

(前のページからの続き)


```
  "msg": "record",
  "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、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。

4.23.3 PGAudit ログ

CloudNativePG は、PostgreSQL クラスターでの *PGAudit ログ* を透過的かつネイティブにサポートしています。

あなたがする必要があるのは、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ことだけです。

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および管理します。

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で拡張機能の作成と削除も行います。

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。

次に、PostgreSQL 13 `Cluster` デプロイメントの例を示します。その結果、要求された構成で `pgaudit` が有効になります。

```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
```

PGAudit によって返された監査 CSV ログエントリは、残りのすべてのログと同様に解析され、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` を除きます

以下の例を参照してください。

```yaml
{
  "level": "info",
  "ts": 1627394507.8814096,
  "logger": "pgaudit",
  "msg": "record",
  "record": {
```

(次のページに続く)

(前のページからの続き)


```
    "log_time": "2021-07-27 14:01:47.881 UTC",
    "user_name": "postgres",
    "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。

4.23.4 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、`logger` はそれらを生成したプロセスに応じて設定されます。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（msg は record とは異なります）
- 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 を除き、他のすべての可能な値では、ログに記録されるエスケープされたメッセージに msg が設定されます。

4.24 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には、オペレーターに以下が必要です。

- 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: クラスターによって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見つけることができます。

CloudNativePG は、TLS 証明書に関して非常に柔軟性があり、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 は自分自身を 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 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。

4.24.1 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、クライアントとサーバーの両方の証明書に使用します。これらの証明書は、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保存します。

- `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。タイプ `kubernetes.io/tls` の Secret に保存され、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、クライアントが自分の ID を確認して安全に接続できるようにします。

サーバーの代替 DNS 名

デフォルトの名前に加えて、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

デフォルトでは、サーバー CA と同じ自己署名 CA が使用されます。public 部分は、署名したクライアント証明書を検証できるように、すべてのインスタンスに `ssl_ca_file` として渡されます。プライベートキー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、`kubernetes.io/tls` 型の Secret に保存します。この証明書は、プライマリインスタンスに安全に接続できるように、レプリカの接続文字列で `sslcert` および `sslkey` として渡されます。

4.24.2 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個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` キーと `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 Secret の名前。

注釈: オペレーターは、クライアント証明書に関連する 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: ConfigMap とシークレットをインスタンスによって**自動的に**リロードしたい場合は、キー *cnpg.io/reload* を持つラベルをそれに追加できます。そうでない場合、 *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 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 を設定し、必要な TLS サーバー証明書を生成する方法に関する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は 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、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、

```
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: selfsigned-issuer
spec:
  selfSigned: {}
- --
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: server-ca
```

(次のページに続く)

(前のページからの続き)

```
spec:
  isCA: true
  commonName: my-selfsigned-server-ca
  secretName: server-ca-key-pair
  privateKey:
    algorithm: ECDSA
    size: 256
  issuerRef:
    name: selfsigned-issuer
    kind: Issuer
    group: cert-manager.io
- --
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: server-ca-issuer
spec:
  ca:
    secretName: server-ca-key-pair
- --
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:
    - 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
```

(次のページに続く)

(前のページからの続き)

```
    - cluster-example-ro.default.svc
  issuerRef:
    name: server-ca-issuer
    kind: Issuer
    group: cert-manager.io
- --
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: client-ca
spec:
  isCA: true
  commonName: my-selfsigned-client-ca
  secretName: client-ca-key-pair
  privateKey:
    algorithm: ECDSA
    size: 256
  issuerRef:
    name: selfsigned-issuer
    kind: Issuer
    group: cert-manager.io
- --
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: client-ca-issuer
spec:
  ca:
    secretName: client-ca-key-pair
- --
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: client-ca-issuer
    kind: Issuer
    group: cert-manager.io
- --
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
    clientCASecret: my-postgres-client-cert
    replicationTLSSecret: my-postgres-client-cert
  storage:
    size: 1Gi
```

配置マニフェスト。

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、 証明書マネージャーの例 や
hashicorp vault

などの別個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` キーと `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必要がある CA の `ca.crt` キーを含む Secret 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 秘密鍵を制御できないため、 *kubectl cnpg certificate* を使用してクライアント証明書を生成できなくなりました。

注釈: ConfigMap とシークレットをインスタンスによって**自動的に**リロードしたい場合は、キー *cnpg.io/reload* を持つラベルを追加できます。そうでない場合、 *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

ここでは、

証明書マネージャーの例 を使用して自己署名 CA を設定し、必要な TLS サーバー証明書を生成する方法に関する簡単な例を示します。

```
- --
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
  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は 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、次のようにクラスターから参照できます。

```yaml
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、

```yaml
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: selfsigned-issuer
spec:
  selfSigned: {}
- --
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: server-ca
spec:
  isCA: true
  commonName: my-selfsigned-server-ca
  secretName: server-ca-key-pair
  privateKey:
    algorithm: ECDSA
    size: 256
  issuerRef:
    name: selfsigned-issuer
    kind: Issuer
    group: cert-manager.io
- --
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: server-ca-issuer
spec:
  ca:
    secretName: server-ca-key-pair
- --
```

(次のページに続く)

(前のページからの続き)

```
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:
    - 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: server-ca-issuer
    kind: Issuer
    group: cert-manager.io
- --
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: client-ca
spec:
  isCA: true
  commonName: my-selfsigned-client-ca
  secretName: client-ca-key-pair
  privateKey:
    algorithm: ECDSA
    size: 256
  issuerRef:
    name: selfsigned-issuer
    kind: Issuer
```

(次のページに続く)

(前のページからの続き)

```
    group: cert-manager.io
- --
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: client-ca-issuer
spec:
  ca:
    secretName: client-ca-key-pair
- --
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
  commonName: streaming_replica
  issuerRef:
    name: client-ca-issuer
    kind: Issuer
    group: cert-manager.io
- --
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
    clientCASecret: my-postgres-client-cert
    replicationTLSSecret: my-postgres-client-cert
  storage:
    size: 1Gi
```

配置マニフェスト。

4.25 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。

CloudNativePG オペレーターは、サーバー側とクライアント側での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および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。

TLS / SSL 証明書を介して認証するには、次の手順を参照してください。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  postgresql:
    pg_hba:
      - hostssl app all all cert

  storage:
    size: 1Gi
```

マニフェスト。構成パラダイムの規則によると、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 構成を介して変更できます)。

4.25.1 新しい証明書の発行

kubectl 用プラグインの使い方の詳細ページ。

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
```

これで、次の方法で証明書を検証できます。

```
kubectl get secret cluster-app \
  -o jsonpath="{.data[tls\.crt]}" \
```

(次のページに続く)

(前のページからの続き)

```
  | 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、デフォルトで TLS クライアント証明書は 90 日間の有効性と、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` 認証方法を活用するために必要です。

4.25.2 TLS 証明書を介した接続のテスト

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。

`cert-test.yaml` という次のマニフェストは、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使用してデモポッドを作成します。

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cert-test
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webtest
  template:
    metadata:
      labels:
        app: webtest
    spec:
      containers:
```

(次のページに続く)

(前のページからの続き)

```
    - 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
        defaultMode: 0600
```

この Pod は、CloudNativePG オペレーターが管理す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` ：TLS 認証局証明書、インスタンスの ID を検証するために使用されるサーバー上の証明書に署名

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の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` ディレクトリ内で証明書を検索しますが、sslkey、

sslcert、sslrootcert オプションを使用して、libpq が暗号化素材の実際の場所を指すように指定できます。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、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到達できるときにアプリケーションの準備ができていることを確認するように、readiness 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 を実行することにより、接続が機能することを確認できます。PostgreSQL サーバーは、読み書き可能な Kubernetes サービスを介して公開されます。psql コマンドをポイントして、このサービスに接続します。

```
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 15.3 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.3.1 20191121 (Red Hat
8.3.1-5), 64-bit
(1 row)
```

4.26 アプリから接続する

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で動作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、
ServiceAccountTemplate で説明されている形式の仮想 IP 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 インスタンスへの直接接続を避ける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ります。

次の方法で、アプリケーションでこれらのサービスを使用できます。

- 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。

4.26.1 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、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、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。

4.26.2 環境変数

PostgreSQL クラスターが含まれる同じ名前空間にアプリケーションをデプロイする場合、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` であるとします。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**プライマリ**インスタンスを指すサービスの IP アドレス

4.26.3 秘密

PostgreSQL オペレーターは、デプロイする PostgreSQL クラスターごとに 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、および作業用の .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**。

-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で使用する必要がある資格情報です。

-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。

4.27 コネクションプーリング

CloudNativePG は、 Pooler CRD を介して、PostgreSQL 用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

PgBouncerIntegrationStatus 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。

簡単に言えば、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、別個のスケーラブルで構成可能な高可用性**データベースアクセスレイヤー** を作成します。

4.27.1 アーキテクチャ

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り、スイスアーミーナイフの追加のブレードのように、CloudNativePG のアーキテクチャがどのように変わるかを強調しています。PostgreSQL プライマリサービスに直接接続する代わりに、アプリケーションは PgBouncer の同等のサービスに接続できるようになり、既存の接続を再利用して、PostgreSQL 側でのパフォーマンスの高速化とリソース管理の向上を実現できます。

4.27.2 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、例です。

```
apiVersion: postgresql.cnpg.io/v1
kind: Pooler
metadata:
  name: pooler-example-rw
spec:
  cluster:
```

(次のページに続く)

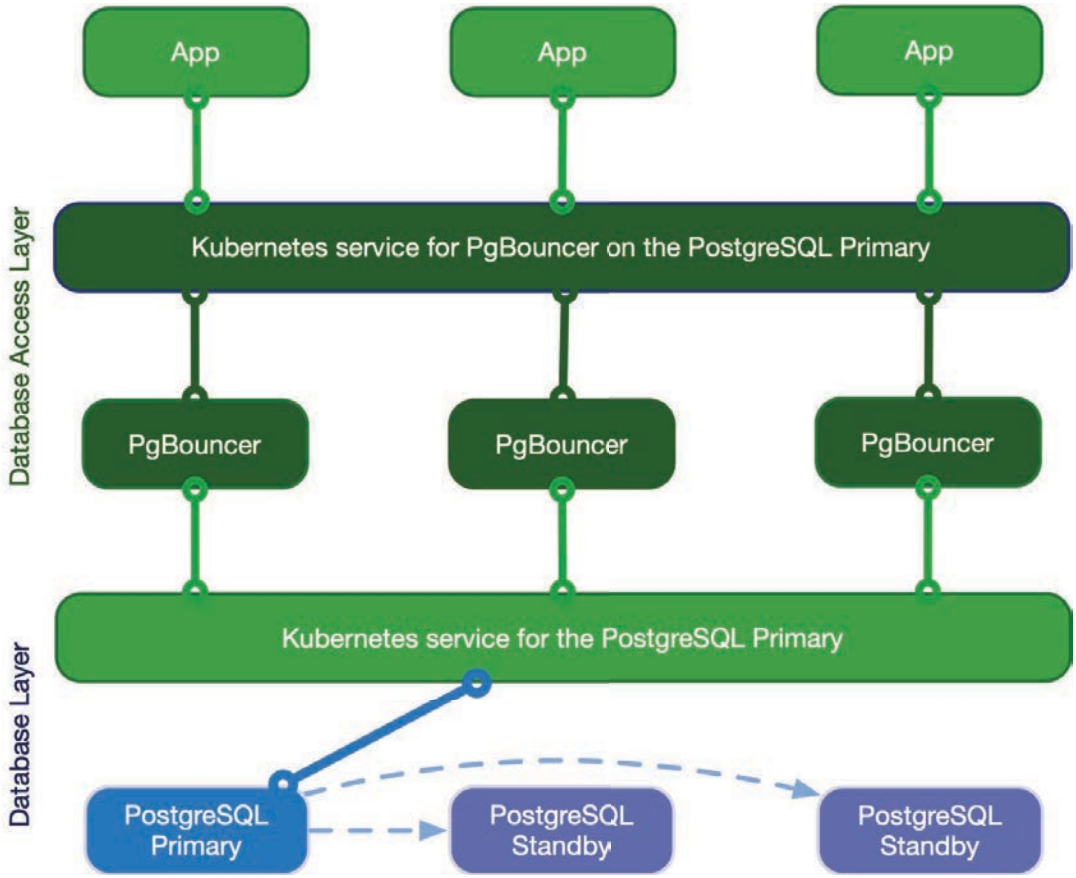


図 15: Applications writing to the single primary via PgBouncer

(前のページからの続き)

```
  name: cluster-example

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。

`Pooler` は、Postgres クラスターと同じ名前空間に存在する必要があります。

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 個の接続を受け入れます-PostgreSQL に対するデフォルトのプールサイズは 10 ユーザー/データベースペアです。

重要: *Pooler* は PgBouncer に `*` 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 サーバーにリレーされます（ ["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。

4.27.3 プーラーリソースのライフサイクル

`Pooler` リソースは `Cluster` 管理のリソースではありません。プーラーは、必要に応じて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。

`Cluster` および `Pooler` リソースのライフサイクルは現在独立していることに注意することが重要です。`Cluster` の削除は、 `Pooler` の自動削除を意味するものではなく、その逆も同様です。

重要: *Pooler* がどのように機能するかを知ったので、可能なアーキテクチャに関して完全に自由があります。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を持つクラス

ター（つまり、アプリケーションごとに 1 つ）を持つことができます。

4.27.4 セキュリティ

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**を介した転送中の暗号化の CloudNativePG サポート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 認証の使用 を参照）。

コンテナは `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` 認証を介したローカル接続を介してのみ許可されます。

証明書

デフォルトでは、PgBouncer プーラーはクラスター自分自身が使用する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

不透明な場合、使用する必要がある特定のキーを探します。そ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、そこから始めることができます。

4.27.5 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、私たちの実装は 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、オペレーターは次のことを行います。

- 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 PostgreSQL サーバーに対して cnpg_pooler_pgbouncer を認証するように 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を検出すると、上記すべてを削除します

重要: 独自のシークレットを指定した場合、オペレーターはプーラーを自動的に統合しません。

独自のシークレットを指定した場合にプーラーを手動で統合するには、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 に接続する権限を付与します。

```
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;
```

4.27.6 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仕様を利用できます。詳細は 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の微調整など、ポッドを好きなように構成できます。デフォルトでは、コンテナは ghcr.io/cloudnative-pg/pgbouncer のイメージを使用します。

ここに PodAntiAffinity を指定するプーラーの例：

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

(次のページに続く)

(前のページからの続き)

```
  name: pooler-example-rw
spec:
  cluster:
    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、変更し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"*

ここでは、リソースを設定し、使用される画像を変更する例を示します。

```
apiVersion: postgresql.cnpg.io/v1
kind: Pooler
metadata:
  name: pooler-example-rw
spec:
  cluster:
    name: cluster-example
  instances: 3
  type: rw

  template:
    metadata:
```

(次のページに続く)

(前のページからの続き)

```
    labels:
      app: pooler
  spec:
    containers:
      - name: pgbouncer
        image: my-pgbouncer:latest
        resources:
          requests:
            cpu: "0.1"
            memory: 100Mi
          limits:
            cpu: "0.5"
            memory: 500Mi
```

4.27.7 高可用性（HA）

Kubernetes の展開のおかげで、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により、クライアントが PgBouncer を実行している利用可能なポッドにランダムに分散されます。。

> **警告:** インフラストラクチャが複数のアベイラビリティーゾーンにまたがり、レイテンシーが高い場合は、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、ゾーン 3 で実行されている PgBouncer に接続し、ゾーン 1 の PostgreSQL プライマリをポイントしている場合を考えます。

4.27.8 PgBouncer 構成オプション

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:** オペレーターは各オプションの値を検証しないため、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、カスタマイズが許可されている PgBouncer オプションのリストを示します。それぞれには、その特定のパラメーターの PgBouncer ドキュメントへのリンクが含まれています。ここで特に断りのない限り、デフォルト値は PgBouncer によって直接設定された値です。

- 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` に追加されます -

  CNP が必要
- log_connections
- log_disconnections
- log_pooler_errors
- log_stats ：デフォルトでは無効（`0`

  ）、統計は以下の *MonitoringConfiguration* 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既に収集されている場合
- 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` マップに宣言的に書き込まれます。

オペレーターはプーラー仕様の変更に反応し、すべての PgBouncer インスタンスはサービスを中断することなく更新された構成をリロードします。

警告: すべての PgBouncer Pod 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**の操作性が混乱す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。**

4.27.9 モニタリング

Pooler の PgBouncer 実装には、デフォルトの Prometheus エクスポーターが付属しています。これは、次を実行して、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します。

- 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（プレフィックス go_* ）。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 メトリックの出力例：

```
#  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
```

(次のページに続く)

(前のページからの続き)

```
cnpg_pgbouncer_collections_total 1

#  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
```

(次のページに続く)

(前のページからの続き)

```
#  HELP cnpg_pgbouncer_lists_free_servers Count of free servers.

#  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
```

(次のページに続く)

(前のページからの続き)

```
cnpg_pgbouncer_pools_sv_idle{database="pgbouncer",user="pgbouncer"} 0

#  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
```

(次のページに続く)

(前のページからの続き)

```
cnpg_pgbouncer_stats_total_received{database="pgbouncer"} 0

#  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、特定の Pooler は、 Prometheus Operator's リソース
PodMonitor を使用して監視できます。 Pooler

を正しく指す PodMonitor は、オペレーターが Pooler リソース自分自身で.spec.monitoring.enablePodMonitor を true に設定することにより、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。カスタマイズが必要な場合は、以下に説明するように行うことができます。

特定のプーラーの 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
```

(次のページに続く)

(前のページからの続き)

```
metadata:
  name: <POOLER_NAME>
spec:
  selector:
    matchLabels:
      cnpg.io/poolerName: <POOLER_NAME>
  podMetricsEndpoints:
  - port: metrics
```

4.27.10 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"
  }
}
```

4.27.11 接続の一時停止

Pooler 仕様により、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 に設定されている場合、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 に設定されると、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、スイッチオーバー操作は 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知覚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>` を介してスイッチオーバーコマンドを発行し、最後に *paused* 属性を `false` に復元することにより、同じ結果を達成できます。

4.27.12 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構成できず、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ユースケースを考慮すると、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: 採用されたソリューションがユースケースの大部分に対処する一方で、PgBouncer が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の別のオペレーターを将来的に実装する余地を残していると信じる理由があります。

4.28 レプリカクラスター

レプリカクラスターは独立した CloudNativePG `Cluster` リソースで、別の Postgres インスタンスのレプリカにあるという主な特徴を持ち、理想的には CloudNativePG でも管理されます。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、以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。

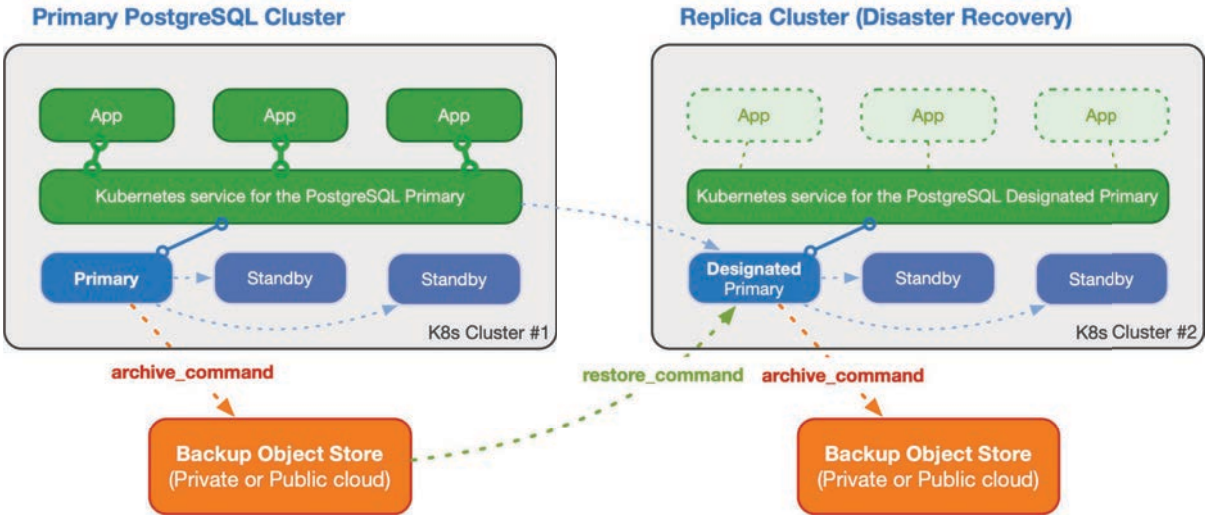


図 16: An example of multi-cluster deployment with a primary and a replica cluster

4.28.1 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、
レプリカクラスター 機能を介して同期されたままになっている場合でも、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使用可能なオプションは次のとおりです。

- レプリカクラスターとソース間の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 つのクラスター間のネットワーク接続が正しくセットアップ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が確かに必要になります）
- ベースバックアップと、ソースからオブジェクトストアに定期的に出荷され、レプリカクラスター内の `barman-cloud-wal-restore` によってプルされる 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します
- 2 つのいずれか

あなたがしなければならないのは、実際に外部クラスターを定義することだけです。
BootstrapConfiguration 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）のいずれかを使用して PostgreSQL サーバーのクローンを作成する方法について。

外部クラスターに `barmanObjectStore` セクションが含まれる場合：

- `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で primary_conninfo オプションを自動的に設定し、WAL レシーバープロセスが開始されてソースクラスターに接続し、データを受信します。

作成されたレプリカクラスターは、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により、PostgreSQL データベースのお気に入り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あります。

- 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
- 異なるリージョンにある複数の Kubernetes クラスターにまたがるパブリッククラウド
- 前の 2 つのミックス（ハイブリッド）
- 異なるリージョンおよび異なるクラウドサービスプロバイダー上の複数の Kubernetes クラスターにまたがるパブリッククラウド

4.28.2 レプリカクラスターのセットアップ

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
- レプリカクラスター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できます

・レプリカクラスターに継続的リカバリ部（spec.replica ）を定義します。私たちがする必要があるのは、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 名前を設定することです

この **最初の例**では、ブートストラップと継続的リカバリの両方でストリーミングレプリケーションを使用して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/ サブディレクトリ内。

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:
```

(次のページに続く)

(前のページからの続き)

```
  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。レプリカクラスターが別の名前空間にある場合、必要に応じて、-replication および-ca シークレットをコピーする必要があります。

```
externalClusters:
- name: <MAIN-CLUSTER>
  connectionParameters:
    host: <MAIN-CLUSTER>-rw.<NAMESPACE>.svc
    user: streaming_replica
    sslmode: verify-full
    dbname: postgres
  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リカバリーを定義する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、レプリカクラスターは基本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:
  recovery:
    source: cluster-example

replica:
  enabled: true
  source: cluster-example
```

externalClusters セクションでは、 endpointURL および 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、必要に応じて必要なシークレットがコピーされ、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。

4.28.3 レプリカクラスター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
```

レプリカモードを無効にすると、レプリカクラスターとソースクラスターが 2 つの別々のクラスターになり、レプリカクラスターの**指定されたプライマリ** がそのクラスターの **プライマリ**に昇格します。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**不可逆的な**操作です。レプリケーションを無効にして、**指定されたプライマリ**を**プライマリ**に昇格させると、レプリカクラスターとソースクラスターは決定的に 2 つの独立したクラスターになります。

4.29 Kubernetes アップグレード

Kubernetes クラスターは最新の状態に保つ必要があります。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、メンテナンスのためにクラスターからノードを一時的に削除する制御されたダウンタイムが Kubernetes インフラストラクチャに導入されているにもかかわらず、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（推奨される読書： Embracing Risk

Site Reliability Engineering の本から）。

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た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 ノードずつ実行されます。

1. 更新するノードからワークロードを追い出す （`drain` ）

2. 実際の運用（システムアップデートなど）の実行

3. ノードをクラスターに再参加する（`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 の自己修復機能の観点から予想されるケースですが、一時的に劣化したクラスターで動作し、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が推奨される場合があります。

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、それは* 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*）は、パフォーマンスを向上させるために使用されます。

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上にある場合、メンテナンスウィンドウを定義する必要はない場合があります。ポッドが現在使用しているボリュームを、ドレイン後に別のノードで実行されているポッドで再利用できる場合、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（このセクションの残りの部分はスキップできます）。

PostgreSQL のローカルストレージを使用する場合、たとえば、物理ノードまたはノード自分自身を更新します。

> **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 の予想される動作の一部が無効になるか、自己修復、ローリングアップデート、ポッド中断バジェットなど、いくつかの制限付きで実行されます。

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。メンテナンスウィンドウ中に、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**の場合、Kubernetes はノードが再び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**の場合、Kubernetes は PostgreSQL の物理ストリーミングレプリケーションに依存することにより、新しい PVC を使用して別のノードで Pod の再作成を強制し、Pod と一緒に古い PVC を破棄します。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ニングにかかる時間が待機するよりも短い場合を除き、通常はお勧めできません。この動作は、インスタンスが 1 つだけで、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ときは、*--delete-local-data* オプションを追加する必要があります。恐れないでください。PostgreSQL データディレクトリではなく、オペレーターが内部的に使用する別のボリュームを参照しています。

4.29.1 reusePVC が false に設定されたシングルインスタンスクラスター

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でクラスターを作成することをお勧めします。

---

reusePVC を false に設定して、シングル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、ユーザーがインスタンスが実行されている可能性のあるノードを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、2 つのオプションがあります。

1. ダウンタイムを受け入れて reusePVC を有効にします

2. 別のノードにインスタンスをレプリケートし、プライマリに切り替えます

データベースサービスのダウンタイムが環境で許容できる限り、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（ノードのローカルストレージを使用して、ノードがバックアップされるとすぐに）、インスタンスを削除して再作成できます。

それ以外の場合は、クラスターをスケールアップし、別のノードで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継続時間です。

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コーディングします。

2. クラスターを 2 インスタンスにスケールアップします。データベースサイズによっては、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隔離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. クラスターを単一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. 新しいノードで新しいプライマリを実行したまま、古いプライマリのノードを正常にドレインできます。

4.30 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 に従った場合、 `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にアクセスできるデータベースがすでにあるはずです。どちらのサービスもポート `5432` を使用します。

ポート `5432` で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しょう。Kubernetes インフラストラクチャに移行するときの一般的なユースケースは、実際、簡単または持続的に「コンテナ化」できない**レガシーアプリケーション**によって表されるユースケースです。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。他にも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)。

次のことを前提としています。

- NGINX Ingress コントローラーがデプロイされ、正常に動作します
- クラスターに `LoadBalancer` タイプのサービスを作成できます

重要: イングレスは、HTTP および HTTPS トラフィックを公開する場合にのみ必要です。 NGINX Ingress コントローラーはできますが、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、データフィールドに外部に公開されたポートと、内部でポイントする名前空間、サービス、ポートに関する情報が含まれる `tcp-services` `ConfigMap` を作成することです。

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: tcp-services
  namespace: ingress-nginx
data:
  5432: default/cluster-example-rw:5432
```

次に、ドキュメントの提案に従って NGINX Ingress Controller をインストールした場合、 ingress-nginx サービスが必要です。 5432 ポートを ingress-nginx サービスに追加して公開する必要があります。イングレスは、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
  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
```

使用できます

```
- --
apiVersion: v1
kind: ConfigMap
metadata:
  name: tcp-services
```

(次のページに続く)

(前のページからの続き)

```
  namespace: ingress-nginx
data:
  5432: default/cluster-example-rw:5432

- --
apiVersion: v1
kind: Service
metadata:
  name: ingress-nginx
  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
```

kubectl を使用して適用します。

警告: このファイルを直接適用すると、Ingress の `ConfigMap` および `Service` の以前の変更を上書きします

これで、Kubernetes クラスターの外部から PostgreSQL クラスターに到達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` を構成してください。

4.30.1 Minikube でのテスト

Minikube では、実行中の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、展開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 というファイルを作成します。

```
spec:
  template:
    spec:
      containers:
      - name: controller
        ports:
         - containerPort: 5432
           hostPort: 5432
```

そしてそれを ingress-nginx-controller デプロイメントに適用します。

```
kubectl patch deployment ingress-nginx-controller --patch "$(cat patch.yaml)" -n␣
↪ingress-nginx
```

次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
```

4.31 CloudNativePG プラグイン

CloudNativePG は、Kubernetes でクラスターを管理するための kubectl 用プラグインを提供します。

4.31.1 インストールする

さまざまな方法を使用して cnpg プラグインを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たインストールが良いオプションかもしれません。

インストールスクリプトを介して

```
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、目的のリリースに移動でき（CloudNativePG オペレーターと同じまたは新しいリリースを選択します）、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、標準的な慣行と指示に従って、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 ビットサーバーに、プラグインの 1.18.1 リリースをインストールしましょう。まず、適切な.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 で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
```

Krew の使用

Krew 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
```

プラグインの新しいバージョンがリリースされたら、次のコマンドで既存のインストールを更新できます。

```
kubectl krew update
kubectl krew upgrade 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向けにビルドされています。

- Linux * amd64 * arm 5/6/7 * arm64 * s390x * ppc64le
- macOS * amd64 * arm64
- Windows * 386 * amd64 * arm 5/6/7 * arm64

4.31.2 使用する

プラグインをインストールしてデプロイしたら、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。このオプションは通常、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、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。

詳細と使用可能なオプションについては、次を実行します。

```
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 ：デプロイメント内のレプリカの数
- --version ： 1.17 など、インストールされ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 ：ウォッチする名前空間を含むコ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の意味は次のとおりです。

- -n king は、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 マイナーバージョン 1.17 の最新パッチバージョンをインストールします
- --replicas 3 は 3 つのレプリカでオペレーターをインストールします
- --watch-namespaces "albert, bb, freddie" は、オペレーターが albert 、bb 、および freddie 名前空間の変更のみを監視します

ステータス

status コマンドは、以下を含む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と WAL 内の位置
- **backup** ：回復可能性ポイント、およびプライマリから 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ー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が直接取得した各 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（リプレイ LSN）に対応します。

重要: 上記のステータス情報はさまざまな時間にさまざまな場所で取得され、結果としてわずかに一貫性のない戻り値が得られます。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:15.3
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:15.3
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:              000000080000003B00000001D   @   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 形式での出力もサポートしています。

宣伝する

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に promote することです。したがって、メンテナンス作業から開始したり、クラスターのスイッチオーバー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、資格情報を保存する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
```

シークレットが作成された後、 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表示します。

```
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 つの場合に使用できます。

- オペレーターに、特定のクラスターのロールアウトリスタートをオーケストレーションするように要求します。これは、カスタムモニタリングクエリを含む ConfigMap など、クラスター依存オブジェクトに構成の変更を適用する場合に役立ちます。
- 単一のインスタンスの再起動を要求します。インスタンスがクラスターのプライマリの場合はその場で、ポッドがレプリカの場合はポッドを削除して再作成します。

```
#  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変更を適用できない場合、スイッチオーバーはインプレース再起動よりも優先されます。こ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: ConfigMap と Secrets をインスタンスによって**自動的に**リロードしたい場合、キー *cnpg.io/reload* を持つラベルを追加できます。

リロード

`kubectl cnpg reload` コマンドは、オペレーターに特定のクラスターの調整ループをトリガーするように要求します。これは、カスタムモニタリングクエリを含む ConfigMap など、クラスター依存オブジェクトに構成の変更を適用する場合に役立ちます。

次のコマンドは、特定のクラスターのすべての構成をリロードします。

```
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。次のフィールドを変更します。

- .spec.nodeMaintenanceWindow.inProgress
- .spec.nodeMaintenanceWindow.reusePVC

これを使用して `set` および `unset` を引数として受け入れ、`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とその新しい値のリストを使用して確認を求めます。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、次のコマンドを書くだけです。

```
kubectl cnpg maintenance set --all-namespaces
```

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
```

レポート

kubectl cnpg report コマンドは、さまざまな情報を ZIP ファイルにバンドルします。本番環境の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ー

operator サブコマンドは、オペレーターにオペレーターの展開、構成、およびイベントに関する情報を提供するように要求します。

重要: Secrets と ConfigMaps のすべての機密情報は REDACTED 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* はリダクションを無効にし、値を表示します。あなた自身の責任でのみ使用してください、これはプライベートデータを共有します。

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* フラグでオペレーターログ収集を有効にできます

- **デプロイメント情報** ：operator Deployment と operator Pod
- **構成** ：オペレーター名前空間の Secrets と ConfigMaps
- **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
- **webhookservice** ：ウェブフックサービス
- **logs** ：JSON 行形式のオペレーター Pod（オプション、デフォルトでオフ）のログ

このコマンドは、さまざまなマニフェストを YAML 形式で含む ZIP ファイルを生成します（デフォルトでは、-o フラグで 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 フラグが使用されない場合、デフォルトのタイムスタンプ付きのファイル名が zip ファイルに作成されます。

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る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インストールされません。例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

```
Successfully written report to "report_operator_<TIMESTAMP>.zip" (format: "yaml")
```

-f フラグを設定します。

```
kubectl cnpg report operator -n <namespace> -f reportRedacted.zip
```

ファイルを解凍すると、ディレクトリを整理するためのタイムスタンプ付きの最上位フォルダーが生成されます。

```
unzip reportRedacted.zip
```

結果：

```
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
```

--logs オプションをアクティブにすると、追加のサブディレクトリが表示されます。

```
Archive:  report_operator_<TIMESTAMP>.zip
  <snipped …>
  creating: report_operator_<TIMESTAMP>/operator-logs/
  inflating: report_operator_<TIMESTAMP>/operator-logs/cnpg-controller-manager-
↪66fb98dbc5-pxkmh-logs.jsonl
```

注釈: プラグインは、PREVIOUS オペレーターのログの取得を試みます。これは、再起動されたオペレーターを調査するときに役立ちます。すべての場合において、 CURRENT オペレーターログの取得も試みま

す。現在および以前のログが利用可能な場合、それらの両方が表示されます。

```
====== Begin of Previous Log =====
2023-03-28T12:56:41.251711811Z {"level":"info","ts":"2023-03-28T12:56:41Z","logger":
↪"setup","msg":"Starting CloudNativePG Operator","version":"1.19.1","build":{"Version
↪":"1.19.0+dev107","Commit":"cc9bab17","Date":"2023-03-28"}}
2023-03-28T12:56:41.251851909Z {"level":"info","ts":"2023-03-28T12:56:41Z","logger":
↪"setup","msg":"Starting pprof HTTP server","addr":"0.0.0.0:6060"}
  <snipped …>

====== End of Previous Log =====
2023-03-28T12:57:09.854306024Z {"level":"info","ts":"2023-03-28T12:57:09Z","logger":
↪"setup","msg":"Starting CloudNativePG Operator","version":"1.19.1","build":{"Version
↪":"1.19.0+dev107","Commit":"cc9bab17","Date":"2023-03-28"}}
2023-03-28T12:57:09.854363943Z {"level":"info","ts":"2023-03-28T12:57:09Z","logger":
↪"setup","msg":"Starting pprof HTTP server","addr":"0.0.0.0:6060"}
```

オペレーターが再起動されていない場合、`====== Begin …`および`====== End …`ガードは、内部にコンテンツなしで引き続き表示されます。

機密情報がデフォルトで REDACTED 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
    fieldsType: FieldsV1
    fieldsV1:
```

`-S`（`--stopRedaction`）オプションをアクティブにすると、シークレットが表示されます。

```
kubectl cnpg report operator -n <namespace> -f reportNonRedacted.zip -S
```

機密情報を閲覧しようとしているという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** ：クラスター名前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**：クラスター名前と一致するクラスター名前空間にジョブがある場合
- **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、デフォルトでオフ）のログ
- **ジョブログ**：JSON 行形式のジョブ（オプション、デフォルトでオフ）によって作成された Pod のログ

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 フラグが使用されない場合、デフォルトのタイムスタンプ付きの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* フラグでクラスターログ収集を有効にできます

使用法：

```
kubectl cnpg report cluster <clusterName> [flags]
```

operator サブコマンドとは異なり、 cluster サブコマンドでは、クラスター名、およびクラスターがデフォルトにない限り、ほとんどの場合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
```

そして：

```
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、ポッドとジョブのログを 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：

```
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
``` ### 破壊する

`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` フラグが指定されている場合、指定すると、PVC を保持しながら、インスタンスによって設定されたすべての `metadata.ownerReferences` を削除できます。さらに、PVC の `cnpg.io/pvcStatus` ラベルが `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` フラグを付けずにコマンドを再度実行すると、デタッチされた PVC が削除されます。

(次のページに続く)

(前のページからの続き)

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

クラスターハイバネーション

場合によっては、データを保持しながら CloudNativePG `Cluster` の実行を一時停止し、後でアクティビティを再開することができます。この機能を **クラスターハイバネーション** 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 クラスターを休止すると、PostgreSQL プライマリインスタンスに属する PVC を除き、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。

次の方法でクラスターを休止できます。

kubectl cnpg hibernate on

これにより：

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 を切断し、最新のデータベースステータスと最新のクラスター構成で注釈を付けます

3. 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-前述の PVC を除く

休止状態になると、CloudNativePG クラスターは PVC のグループのみで表され、 `PGDATA` を含むものには、 `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ます。

.. Warning::
フェンシングも休止状態手順の一部であるため、フェンスされたインスタンスを持つクラスターは休止できません。

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はできません。次のコマンドで操作を強制できます。

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

クラスターが休止すると、シャットダウン後の PostgreSQL の最後の構成とステータスを表示できます。これは次のように実行できます。

kubectl cnpg hibernate status

pgbench を使用した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 クラスターに対して実行できます。

kubectl cnpg pgbench – –time 30 –client 1 –jobs 1

詳細については、 :ref:`Benchmarking pgbench section <ベンチマーク>` を参照してください。

fio を使用したストレージのベンチマーク

fio は、次のコマンドを使用して、既存のストレージクラスで実行できます。

kubectl cnpg fio -n

詳細については、 :ref:`Benchmarking fio section <ベンチマーク>` を参照してください。

新しいベースバックアップの要求

`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には、要求時間の名前が付けられます。

edb_notranlate_53 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、実稼働環境では、このメソッドは細心の注意を払って、許可された担当者のみが使用する必要があります。

```
kubectl cnpg psql cluster-example

psql (15.3 (Debian 15.3-1.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 (15.3 (Debian 15.3-1.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 を起動します。正しく動作するには、PATH 変数で kubectl 実行可能ファイルに到達できる必要があります。

Postgres クラスターのスナップショット

kubectl cnpg snapshot は、次の方法で Postgres Cluster の一貫したスナップショットを作成します。

1. 作業するレプリカ Pod の選択

2. レプリカのフェンシング

3. スナップショットを撮る

4. レプリカのフェンシングを解除する

警告: フェンスで囲まれたインスタンスが既にあるクラスターはスナップショットを作成できません。

現時点では、このコマンドは少なくとも 1 つのレプリカを持つクラスターでのみ使用できます。そのレプリカはフェンシング手順によってシャットダウンされ、スナップショットの一貫性が保証されます（コールドバックアップ）。Kubernetes の VolumeSnapshot API の宣言的サポートの開発が継続されると、この制限が削除され、ビジネス継続性が必要な場合にオンラインバックアップを取得できるようになります。

重要: プロシージャがレプリカをシャットダウンする場合でも、プライマリ Pod は関与しません。

kubectl cnpg snapshot コマンドにはクラスター名が必要です。

```
kubectl cnpg snapshot cluster-example

waiting for cluster-example-3 to be fenced
waiting for VolumeSnapshot cluster-example-3-1682539624 to be ready to use
unfencing pod cluster-example-3
```

VolumeSnapshot リソースは、空の VolumeSnapshotClass 参照で作成されます。そのリソースは、デフォルトとして構成された VolumeSnapshotClass によって使用されることを目的としています。

-c オプションを使用して、特定の VolumeSnapshotClass を要求できます。

```
kubectl cnpg snapshot cluster-example -c longhorn
```

4.32 自動フェイルオーバー

spec.failoverDelay（デフォルトでは 0 秒）より長くプライマリで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ります

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想定することはできません。

上記のようなケースの後、プライマリポッドの readiness 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。コントローラーは、次の 2 つの手順でフェールオーバープロセスを開始します。

1. 最初に、TargetPrimary を pending としてマークします。この状態の変化により、プライマリポッドが強制的にシャットダウンされ、レプリカ上の WAL レシーバーが確実に停止します。クラスターは、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、リーダー選挙が行われ、新しいプライマリが命名されます。選択されたインスタンスはプライマリへのプロモーションを開始し、これが完了すると、クラスターは通常

のオペレーションを再開します。一方、元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 フェーズプロシージャは、WAL レシーバーを正常に停止できること、および障害が発生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を保証するのに役立ちます。これらの安全対策により、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。

障害が発生したプライマリがシャットダウンされている間：

1. 最初に、タイムアウトとして`.spec.switchoverDelay`秒で PostgreSQL の**高速シャットダウン**を試みます。このグレースフルシャットダウンは、保留中の WAL をアーカイブしようとします。

2. 高速シャットダウンが失敗するか、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。

注釈: 「高速」モードは、PostgreSQL クライアントが切断するのを待たず、進行中のオンラインバックアップを終了します。アクティブなすべてのトランザクションがロールバックされ、クライアントが強制的に切断され、サーバーがシャットダウンされます。 「即時」モードは、クリーンシャットダウンせずに、すべての PostgreSQL サーバープロセスをすぐに中止します。

4.32.1 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場合があります。

1. プライマリに障害が発生し始めたとき、およびコントローラ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場合があります。

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、サービスに影響はありますが、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は WAL 書き込みを含む保留中のプロセスを停止します。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起動していない間、クラスター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: 高速シャットダウンを制御するタイムアウトは、スイッチオーバーの場合と同様に、*.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と、RPO の観点からは安全ですが、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 ですでに述べたように

スイッチオーバープロセスを説明すると、`.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 が優先される場合がありますが、クラスターレベルおよび/またはバックアップレベルでデータが失われます。逆に、これを高い値に設定すると、データ損失のリスクが排除され、スイッチオーバー中にアクティブなプライマリなしでクラスターが長時間離れなくなります。

4.32.2 遅延フェールオーバー

上記で予想されたように、`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てから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、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、複数の層が影響を受けるネットワークの中断（つまり、ダウンストリームの論理サブスクライバー）、またはフェールオーバーを実行する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延さ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対する時期尚早のフェールオーバーを防ぐメカニズムが提供されます。

4.33 トラブルシューティング

このページでは、Kubernetes クラスターデプロイで CloudNativePG をトラブルシューティングする方法に関する基本情報を見つけることができます。

ヒント: Kubernetes 管理者は、 :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>` ページをブックマークしておく必要があります。

4.33.1 始める前に

Kubernetes 環境

トラブルシューティングアクティビティで違いを生むのは、基になる 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 ディストリビューションとバージョン
- PostgreSQL が動作しているノードの仕様
- **実稼働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む、実際の** *StorageConfiguration* についてできる限り。

- 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）
- 継続的バックアップの状況、特にそれが所定の位置にあり、正しく動作している場合：そうでない場合は、必ず **緊急バックアップ**

混乱を招く可能性のある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との一致を含む行の 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すでに利用可能です。Windows OS を使用している場合、`grep` の代わりに findstr を使用するか、 wsl を直接使用できます

そして、お好みの* 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。

4.33.2 緊急バックアップ

緊急事態によっては、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: 以下に示す手順は、緊急事態においてのみ実行し、一時バックアップファイルは組織で有効なデータ保護ポリシーの下で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ます。そのため、すべての保護が整っており、バックアップファイルを保存するための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` pod 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作成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により、上記のコマンドを簡単に調整してクラスターをバックアップできます。

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（したがって、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）に接続することにより、new-cluster-example Postgres クラスターの app データベースに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、推奨事項に従って、ダンプおよび復元する他のグローバルオブジェクト（データベースとロール）がないことを前提としています。複数のロールがある場合は、 *pg_dumpall -g* を使用してバックアップを作成し、新しいクラスターに手動で復元してください。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ながら、一度に 1 つのデータベースずつ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、専門のサポート会社の指導の下でこれらの重要な操作を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 で期待されるように、標準出力にログを記録し、JSON format に直接記録します。その結果、kubectl logs コマンドに依存して、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、次を入力してください。

```
kubectl logs --help
```

ヒント: JSON ログは機械の読み取りには最適ですが、人間が読み取るのは困難です。 *jq* コマンドを使用して、使いやすさを向上させ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に関するいくつかの例を示します。

4.33.3 オペレーター情報

デフォルトでは、CloudNativePG オペレーターは、Kubernetes の cnpg-system 名前空間に Deployment としてインストールされます（ 展開の詳細

詳細については）。

次を実行して、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、オペレーターが実行されている 1 つのポッドがあり、 *cnpg-controller-manager-* で始まる名前で識別されます。高可用性のためにオペレーターを設定している場合は、より多くのエントリが必要です。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、 pod <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

次を実行して、CloudNativePG オペレーターデプロイ（マルチオペレーターデプロイがある場合）のすべてのポッドからログを取得します。

```
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、ログをリアルタイムで追跡できます。

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:15.3-3
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
```

4.33.4 クラスター情報

次のコマンドを使用して、 NAMESPACE 名前空間の<CLUSTER> クラスターのステータスを確認できます。

```
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 つのインスタンスの正常な PostgreSQL クラスターが報告され、すべてが**準備完了**状態で、<CLUSTER>-1 がプライマリです。

異常な状態の場合、 Cluster リソースのマニフェストを取得することで、さらに発見できます。

```
kubectl get cluster -o yaml -n <NAMESPACE> <CLUSTER>
```

収集する別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、より多くの情報を印刷できます。

注釈: クラスターステータスを知ることに加えて、 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。レプリカをプロモートします。
証明書を管理します。
ロールアウトリスタートクラスターを作成して構成変更を適用します。
ループを使用してリロードし、構成の変更を適用します。
 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>` ドキュメントを参照してください。

PostgreSQL コンテナ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:15.3-3
```

注釈: *kubectl-cnpg status -n <NAMESPACE> <CLUSTER_NAME>* を使用して同じ情報を取得することもできます。

4.33.5 ポッド情報

次のコマンドを使用して、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できます。

```
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
```

次を実行して、ポッドが失敗しているかどうかを確認できます。

```
kubectl get pod -n <NAMESPACE> -o yaml <CLUSTER>-<N>
```

次のコマンドで、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、使いやすい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およびフィルタリングします

クラッシュした特定のポッドからログを確認します。

```
kubectl logs -n <NAMESPACE> --previous <CLUSTER>-<N>
```

特定の PostgreSQL Pod 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",
```

(次のページに続く)

(前のページからの続き)

```
  "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 pod から err word 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".
```

値を持つフィールドでフィルタリングされたポッドログを取得し、 | で区切って結合します 実行：

```
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
```

4.33.6 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リストできます。

```
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。したがって、そのバージョン以降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。

4.33.7 ストレージ情報

次のように、クラスターが使用する StorageClass をダブルチェックして、調査またはトラブルシューティング中により多くのコンテキストを確認すると役立つ場合があります。

```
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 を取得します。

4.33.8 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だけ多くを知ることは戦略的に重要です。

次のコマンドで、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、ポッドが配布される場所を理解する上で重要です。 *affinity/anti-affinity rules and/or tolerations* を使用している場合は非常に役立ちます。

4.33.9 条件

多くのネイティブ kubernetes オブジェクト like here と同様に、クラスターは status.conditions も公開します。これにより、クラスター全体のヘルス状態に依存する代わりに、特定のイベントが発生するのを「待機」できます。現時点で利用可能な条件は次のとおりです。

- LastBackupSucceeded
- ContinuousArchiving
- レディ

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は正しく取得されています、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は正しく終了されました、そうでない場合は False に設定されます。

Ready は、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できている場合、True です。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機できます。

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>
```

- 連続アーカイブ：

```
$ kubectl wait --for=condition=ContinuousArchiving cluster/<CLUSTER-NAME> -n
↪<NAMESPACE>
```

- Ready（クラスターの準備ができているかどうか）：

edb_notranlate_39 以下は、失敗した条件を含む 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
```

(次のページに続く)

(前のページからの続き)

```
    - message: exit status 2
      reason: LastBackupFailed
      status: "False"
      type: LastBackupSucceeded

    - message: Cluster Is Not Ready
      reason: ClusterIsNotReady
      status: "False"
      type: Ready
```

4.33.10 ネットワーキング

CloudNativePG には、基本的なネットワーキングと接続が整っている必要があります。

ネットワーキング セクションで詳細情報を見つけることができます。

CloudNativePG を既存の環境にインストールする場合、ネットワークポリシーが設定されているか、クラスター専用に作成されたその他のネットワーク構成が存在する可能性があり、オペレーターとクラスターポッド間および/またはポッド間で必要な接続に影響を与える可能性があります。

次のコマンドを使用して、既存のネットワークポリシーを探すことができます。

```
kubectl get networkpolicies
```

Kubernetes ネットワーク管理者によって設定されたネットワークポリシーがいくつかある場合があります。

```
$ kubectl get networkpolicies
NAME                   POD-SELECTOR                        AGE
allow-prometheus       cnpg.io/cluster=cluster-example   47m
default-deny-ingress   <none>                              57m
```

4.33.11 いくつかの一般的な問題

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、ログからディスクがいっぱいになっていることが原因である可能性があることが示されている場合は、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 をご覧ください。

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 セクションを確認して、この背後にある理由を理解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 に一致するノードはありません
- ノードのテイントと一致するように容認が正しく構成されていません
- ノードがまったく利用できません。これは、cluster-autoscaler がいくつかの制限に到達するか、一時的な問題があることに関連している可能性もあります

この場合、名前空間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、レプリカが少しの間オフにされる場合があります（Kubernetes ノードがドレインされる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、レプリカが回復したときに、プライマリに存在しない WAL ファイルが必要になる場合があります（
: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すで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い WAL ファイルを必要とする場合、pg_rewind: error: could not open file を報告します。

これらの場合、ポッドはこれ以上準備ができていないため、PVC を削除し、オペレーターにレプリカを再構築させる必要があります。

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 自分自身を削除することに自信がある場合は、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
```

Creating new replica でスタックするクラスター

クラスターは「新しいレプリカの作成」でスタックしていますが、ポッドログには関連する問題が表示されません。これは、次の問題
on connectivity

に関連していることが判明しています。リリース 1.20.1、1.19.3、および 1.18.5 から、ネットワークの問題は、次のようにステータス列により明確に反映されます。

```
Instance Status Extraction Error: HTTP communication issue
```

インストールされたネットワークポリシーによりネットワーキングが損なわれています

> *networking section* で指摘されているように、ローカルネットワークポリシーは、必要な接続の一部を妨げる可能性があります。

接続が損なわれていることを示す兆候は、次のようなメッセージのオペレーターログに存在することです。

```
"Cannot extract Pod status", [···snipped···] "Get \"http://<pod IP>:8000/pg/status\":␣
↪dial tcp <pod IP>:8000: i/o timeout"
```

ネットワークポリシーをリストし、接続を制限するポリシーを探す必要があります。

```
$ kubectl get networkpolicies
NAME                   POD-SELECTOR                       AGE
allow-prometheus       cnpg.io/cluster=cluster-example    47m
default-deny-ingress   <none>                             57m
```

たとえば、上記のリストでは、 default-deny-ingress が犯人である可能性が高いようです。あなたはそれにドリルすることができます：

```
  $ kubectl get networkpolicies default-deny-ingress -o yaml
  <···snipped···>
  spec:
    podSelector: {}
    policyTypes:
    - Ingress

:ref:`networking page <ネットワーキング>` には、 ``NetworkPolicy``
```

を明示的に作成してオペレーターがクロスネームスペースをクラスターポッドに接続できるようにカスタマイズできるネットワークポリシーファイルがあります。

4.34 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が誤動作していると思われる場合に、それらのインスタンス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スされると、ポッドが実行されたままで、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、ポッド上のデータが PostgreSQL によって変更されず、デバッグと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。

4.34.1 インスタンスをフェンスする方法

CloudNativePG では、次のことをフェンスできます。

- 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` アノテーションのコンテンツを介して制御されます。これは、インスタンス名の JSON 形式のリストを必要とします。注釈が`'["*"]'`、ワイルドカード付きの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スされます。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：

- `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スします。

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 "*"
```

これは、以前にフェンスされたインスタンスを持つ `Cluster` の例です。

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

(次のページに続く)

(前のページからの続き)

```
metadata:
    annotations:
      cnpg.io/fencedInstances: ["cluster-example-1"]
[...]
```

4.34.2 フェンシングを持ち上げる方法

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 "*"
```

4.34.3 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、続いて必要に応じて高速シャットダウンで構成されます。次に：

- Pod は生き続けます
- Pod は**準備完了**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 - 証明書およびすべての暗号化マテリアル
- 1 に設定される cnpg_collector_fencing_on を除き、メトリックは収集されません

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 postmaster 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スすることをお勧めします。

fenced インスタンスが削除されると、Pod は正常に再作成されますが、postmaster は起動されません。これは、インスタンスが `Crashlooping` である場合に非常に役立ちます。

4.35 宣言的ハイバーネーション

CloudNativePG は、PostgreSQL クラスターを稼働させ、いつでも利用できるように設計されています。

ワークロードがアクティブな場合にのみデータベースを稼働させる必要がある種類のワークロードがあります。バッチ駆動型ソリューションはそのようなケースの 1 つです。

バッチ駆動型ソリューションでは、データベースはバッチプロセスの実行中にのみ稼働する必要があります。

宣言的ハイバネーション機能を使用すると、データベースの PVC を維持しながらデータベースの Pod を削除することにより、CPU の電力を節約できます。

注釈: 宣言的ハイバネーションは、 imperative hibernation via the `cnpg plugin <CloudNativePG プラグイン>` の既存の実装とは異なります。強制的な休止は、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* および WAL を含むプライマリの PVC の静的コピーを保持します。プラグインは、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により、休止状態フェーズを終了できるようにします（存在する場合）。

4.35.1 ハイバネーション

クラスターを休止状態にするには、 `cnpg.io/hibernation=on` アノテーションを設定します。

```
$ kubectl annotate cluster <cluster-name> --overwrite cnpg.io/hibernation=on
```

休止状態のクラスターには実行中のポッドがありませんが、後でクラスターをリハイドレートできるように PVC は保持されます。プライマリの PVC に加えて、レプリカ PVC が保持されます。

ハイバネーション手順は、プライマリ Pod、次にレプリカ Pod を削除し、スイッチオーバーを回避して、レプリカの同期が維持されるようにします。

`cnpg.io/hibernation` の状態を探すことで、休止状態を監視できます。

```
$ kubectl get cluster <cluster-name> -o "jsonpath={.status.conditions[?(.type==\"cnpg.
↪io/hibernation\")]}"

{
        "lastTransitionTime":"2023-03-05T16:43:35Z",
        "message":"Cluster has been hibernated",
        "reason":"Hibernated",
        "status":"True",
```

(次のページに続く)

(前のページからの続き)

```
        "type":"cnpg.io/hibernation"
}
```

休止状態は、 kubectl の cnpg プラグインの status サブコマンドで読み取ることもできます。

```
$ kubectl cnpg status <cluster-name>
Cluster Summary
Name:               cluster-example
Namespace:          default
PostgreSQL Image:   ghcr.io/cloudnative-pg/postgresql:15.3
Primary instance:   cluster-example-2
Status:             Cluster in healthy state
Instances:          3
Ready instances:    0

Hibernation
Status   Hibernated
Message  Cluster has been hibernated
Time     2023-03-05 16:43:35 +0000 UTC
[..]
```

4.35.2 水分補給

クラスターをリハイドレートするには、 cnpg.io/hibernation アノテーションを off に設定します。

```
$ kubectl annotate cluster <cluster-name> --overwrite cnpg.io/hibernation=off
```

または、完全に設定を解除します。

```
$ kubectl annotate cluster <cluster-name> cnpg.io/hibernation-
```

Pod が再作成され、クラスターがオペレーションを再開します。

4.36 PostGIS

PostGIS **クラスター** は、PostgreSQL 用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）オブジェクトを保存し、

SQL を介して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、読者が PostGIS に精通していることを前提としており、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* 上に構築されたコンテナイメージを維持しています。詳細については、次をご覧ください。

- *postgis-containers*
- *postgis-containers*

4.36.1 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 クラスターと同様です。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の存在
- PostGIS エクステンションのデータベース内の存在

CloudNativePG はイミュータブルアプリケーションコンテナ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、これを達成する方法に関する詳細な手順を提供します。より簡単には、以下の例のように、コミュニティの PostGIS コンテナイメージを使用できます。

2 番目のステップでは、PostgreSQL データベースに拡張機能をインストールします。これは 2 つの方法で行うことができます。

- 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かつおそらく唯一のデータベースであるアプリケーションデータベースにインストールする、または
- 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、クラスター内で最終的に作成するすべてのデータベースで利用できるように `template1` データベースにインストールします

注釈: データベース内のマイクロサービスとモノリスアーキテクチャの詳細については、*How many databases should be hosted in a single PostgreSQL instance?* を参照してください

または *Database import* 。

4.36.2 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 クラスターを作成したいとしましょう。

最初のステップは、オペランドに正しい PostGIS コンテナイメージを使用し、`Cluster` リソースに`.spec.imageName` オプションを適切に設定することです。

は

```
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
```

以下は、PostGIS クラスターの作成を行う方法に関するいくつかのガイダンスを提供します。

警告: CloudNativePG には構成よりも規約が適用されますが、実稼働用にシステムの構成とチューニング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、PostgreSQL 14 で利用可能な最新のイメージを意図的に指し示しています。

```
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 と呼ばれます）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。

注釈: 時間を取って、 *postInitApplicationSQL* など、 *API リファレンス* の *.spec.bootstrap.initdb* で使用可能なオプションを調べます。

---

app データベースに接続することにより、コンテナ内にある PostGIS の利用可能な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 (15.3 (Debian 15.3-1.pgdg110+1))
Type "help" for help.

app=# SELECT * FROM pg_available_extensions WHERE name ~ ^postgis ORDER BY 1;
           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
```

(次のページに続く)

(前のページからの続き)

```
                                    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)
```

最後に：

```
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)
```

4.37 エンドツーエンドのテスト

CloudNativePG は、コミットごとに一連の **エンドツーエンド（E2E）テスト**を介して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します。

さらに、次の Kubernetes バージョンがコミットごとにテストされるため、開発プロセスの初期段階でバグを検出できます。

- 1.27
- 1.26
- 1.25
- 1.24
- 1.23

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 のテストされたバージョンごとに、 種類を使用して Kubernetes クラスターが作成され、そのクラスターで次の E2E テストスイートが実行されます。

- **基本： オペレーターのインストール。** クラスターの作成; *データストレージのため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を介した接続。**ユーザーが提供したサーバーおよび/またはクライアント証明書を介した接続。* PgBouncer;
- **自己修復： フェールオーバー。** スイッチオーバー; * 10 秒未満で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 秒以内にスイッチオーバーした場合のプライマリエンドポイントのスイッチ。* 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 上の Barman Cloud を使用したバックアップと ScheduledBackups の実行。* Azure Blob Storage 上の Barman Cloud を使用したバックアップと ScheduledBackups 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 BLOB ストレージを使用してバックアップから復元します。*ウォルリストア;
- **オペレーター： オペレーター展開。**ConfigMap を介したオペレーター構成。 **オペレーターポッドの削除**; オペレーターポッドのエビクション。 **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 メトリック; * JSON ログ形式。
- **レプリケーション： 物理レプリカクラスター。**レプリケーションスロット。 **同期レプリケーション。**クラスターのスケールアップとスケールダウン。
- **Plugin：** * CNPG プラグインを使用したクラスターハイバネーション。**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ときのローリング更新。
- **ポッドスケジューリング：** * Tolerations と taint。 * `NodeSelector` を使用した Pod 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アノテーションの *ConfigMap*。**オブジェクトメタデータ;
- **リカバリ： データ破損。** 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。

4.38 コンテナイメージの要件

Kubernetes 用の 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で動作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`
- PGAudit 拡張機能がインストールされました（オプション-展開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 はインスタンスマネージャーでそれをオーバーライドするため、イメージ定義にエントリー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イメージは、CloudNativePG によって**マルチ/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持つプライマリ**のみで使用されます。

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 で公開します。

4.38.1 イメージタグの要件

イメージ名は Docker で有効な任意の名前にできますが、CloudNativePG オペレーターは**イメージタグ**を使用して、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、オプションでドットとパッチレベルが続く必要があります。

この後には、Docker タグで有効で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、前にドット、アンダースコア、またはマイナス記号を付けます。

使用できる画像タグの例:

- `11.1`
- `12.3.2.1-1`
- `12.4`

- 13
- 14.5-10
- 15.0

警告: *latest* はイメージの有効なタグとは見なされません。

4.39 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 によって実装される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 からの**「レベル V - 自動パイロット」**機能セットを期待できます。

各機能レベルは、オペレーターが提供する特定の管理機能セット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. フルライフサイクル

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、オペレーターでの将来の作業と実装のガイドと見なされます。

4.39.1 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、オペレーターや PostgreSQL クラスター構成との対話方法の改善など、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: このレベルの**情報セキュリティ**部分を検討します。

宣言型構成によるオペレーターの展開

オペレーターは、4 つの主要な `CustomResourceDefinition` オブジェクトを定義する 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。`Cluster`、`Pooler`、`Backup`、および `ScheduledBackup`。

宣言構成による PostgreSQL クラスターの展開

PostgreSQL クラスター（オペランド）は、`Cluster` カスタムリソースを完全に宣言的な方法で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イメージによって決定され、要求されたレジストリから自動的にフェッチされます。オペランドを展開するとき、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 を介したオペランド画像のオーバーライド

この演算子は、PostgreSQL を内部に含むオペランドコンテナ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 コミュニティでサポートされ、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` と、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

オペレーターは、Kubernetes インフラストラクチャでの CloudNativePG デプロイメントの組織を改善することを目的として、クラスターのメタデータで定義された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。

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注入します。このアプリケーションは、基になる 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ポッドのステータスをインスタンス自分自身と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、プローブの `kubelet` によって呼び出される Web サーバーも起動します。 `kubelet` によって呼び出される 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、外部イベントへの高速で制御された反応のために `postgres` プロセスに転送されます。インスタンスマネージャーは Go で書かれており、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。オペレーターは、ストレージの観点から Kubernetes ネイティブの機能とリソースを活用することにより、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 テンプレートを介して生成された 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、クラスターのログ先行書き込み（WAL、`pg_wal` としても知られる）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を選択できます。

> cnp-bench オープンソースプロジェクトを使用して、本番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` と呼ばれる単一のパラメーターを介して、クラスター内のレプリカを自動的に検出します。1 に設定されている場合、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 より大きい場合、オペレーターは `instances -1` レプリカを管理します。これには、自動フェイルオーバーによる高可用性（HA）とスイッチオーバー操作によるローリング更新が含まれます。CloudNativePG は、 Failover slots と呼ばれる PostgreSQL 用に以前に提案された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

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、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 サービスを介して、データベースへのアクセスを透過的に管理します。構成に対する規約のアプローチを使用して、オペレーターは `app` というデータベースを作成します。デフォルトでは、同じ名前の通常の Postgres ユーザーが所有します。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。クラスターの実行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 ホストベースの認証ルールの両方をカスタマイズできます。

Postgres のロール、ユーザー、およびグループの構成

CloudNativePG は *management of PostgreSQL roles, users, and groups through declarative configuration* をサポート

`.spec.managed.roles` スタンザを使用します。

ポッドセキュリティポリシー

InfoSec 要件については、オペレーターはコンテナーの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強制して、オペレーターとオペランドポッドの両方に対して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。

アフィニティ

クラスターの `affinity` セクションを使用すると、ポッドと、永続ボリュームなどの関連リソースを Kubernetes クラスターのノード全体で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。

- ポッドアフィニティとアンチアフィニティ
- ノードセレクター
- テイントと寛容

トポロジースプレッドの制約

クラスターの `topologySpreadConstraints` セクションを使用すると、トポロジ全体でポッドのスケジュールを追加制御できるため、アフィニティとアンチアフィニティが提供できるものが強化されます。

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ースがありません。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により、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。

クラスターの現在のステータス

オペレーターは、クラスターの監視されたステータスで CR のステータス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します（例：クラスターステータスがレポートその pod -1 がプライマリであり、pod -1 は自分自身を昇格させる必要がありますが、他の Pod は pod -1

に従う必要があります)。同じステータスは、 kubectl の cnpg プラグインによって使用され、詳細を提供します。

オペレーターの認証局

オペレータは、自分用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分自身の間の安全な通信を保証するために、Webhook サーバーが使用するリーフ証明書を作成してオペレーター認証局と署名します。

クラスターの認証局

オペレーターは、すべての PostgreSQL クラスターに認証局を自動的に作成します。これは、ストリーミングレプリケーションスタンバイサーバー（パスワードの代わり）を含む、クライアントの認証用の TLS 証明書の発行および更新に使用されます。クライアント証明書のカスタム認証局のサポートは、シークレットを介して利用できます。これには、cert-manager との統合も含まれます。kubectl の cnpg プラグインを使用して、証明書を発行できます。

TLS 接続

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、クラスターの認証局を使用してセキュリティを強化するためにクライアント/サーバー通信を暗号化します。カスタムサーバー証明書のサポートは、シークレットを介して利用できます。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リロードまたは再起動されます。**現在の制限：** ALTER SYSTEM による変更は検出されません。つまり、クラスターの状態は強制されません。

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もカバーします。オフラインは、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、別の PostgreSQL データベースで利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バージョンの Postgres からのもので、`pg_dump` および `pg_restore` に依存して、オペレーション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の新しいクラスタープライマリから実行されます。

PostGIS クラスター

CloudNativePG は、

PostGIS クラスター 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

地理データベースのオープンソース拡張機能であり、PostgreSQL の最も一般的な拡張機能の 1 つです。

PostgreSQL の基本 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、**シンプルバインド**または**検索+バインド**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

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し、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 Chart も利用できます。

構成よりも規約

オペレーターは、構成パラダイムに対する規約をサポートし、標準のデフォルト値を決定しながら、オーバーライドおよびカスタマイズを許可します。数行の 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。

4.39.2 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に関するものです。これには、**PostgreSQL マイナーリリース更新** （通常はセキュリティとバグの修正）と**メジャー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

オペレーターを新しい展開としてシームレスにアップグレードできます。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、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。演算子は、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とその後のスイッチオーバーは必要ありません。

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部として宣言的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ますが、メジャーバージョン内の PostgreSQL のマイナー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で新しいポッドを作成することにより、レプリカから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` ）を更新する前にスイッチオーバーを続行するか、ユーザーが `kubectl` の `cnpg` プラグインを介してスイッチオーバープロシージャ（`supervised` ）を手動で発行するのを待ちます。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に数秒から数分のダウンタイムが生成される可能性があるため、使用する設定は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、クラスターの高可用性ステータスを伝えます（例、`Setting up primary` 、`Creating a new replica` 、`Cluster in healthy state` 、`Switchover in progress` 、`Failing over` 、`Upgrading cluster` など）。

4.39.3 レベル 3：フルライフサイクル

機能レベル 3 では、オペレーターは **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。**ディザスターリカバリー**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コンポーネントです。開始点としての目標は RPO < 5 分を達成することですが、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 つの重要なコンポーネントです。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を介して、オペレーターはフェールオーバーとスイッチオーバーの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ます。

- PostgreSQL 物理レプリケーションの制御、たとえば、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。
- 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

オペレーターは、物理ベースバックアップと継続的な WAL アーカイブに基づいた PostgreSQL のネイティブ継続的ホットバックアップテクノロジーを使用して、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）で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、クラスターレベルで宣言的に定義され、クラスター定義の `backup` パラメーターを介して、S3 プロトコルの宛先 URL を指定し（たとえば、AWS S3 バケット内の特定のフォルダーをポイントします）、オプションで、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、ユーザーからのさらなる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 ファイルを出荷するために `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。

ベースバックアップは、オンデマンド（`Backup` カスタムリソース定義を介して）またはスケジュール（cron のような構文を使用して、`ScheduledBackup` カスタマーリソース定義を介して）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イメージの一部として配布）を `barman-cloud-backup` に依存して、WAL ファイルとともに同じエンドポイントでバックアップを中継します。

`barman-cloud-wal-restore` と `barman-cloud-backup` の両方は、GNU GPL 3 条件の下でアプリケーションコンテナ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、標準の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` を使用して取得されたアクセス可能な既存のバックアップから開始して、新しいクラスター（その設定）を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、オペレーターはインスタンスをリカバリモードで起動し、指定されたアーカイブから使用可能なすべての WAL ファイルをリプレイし、リカバリを終了してプライマリとして起動します。その後、オペレーターは、プライマリから要求された数のスタンバイインスタンスを複製します。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）

オペレーターを使用すると、既存のバックアップをタイムスタンプ、ラベル、またはトランザクション ID で定義された特定のポイントインタイムに復元することにより、新しい PostgreSQL クラスターを作成できます。この機能は、完全復元の上に構築され、
PostgreSQL for PITR で使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ゼロデータ損失クラスター

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、いつでも使用可能な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`）の次の式を使用して、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
```

レプリカクラスター

PostgreSQL の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 ソースからデータを継続的に複製できます。このようなソースは、2 つのエンドポイントからの TLS 経由の直接ストリーミング接続が許可されている限り、どこにあってもかまいません。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 の外部にあってもかまいません。レプリカクラスターは、回復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ています。レプリカクラスターは、マルチプル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を

可能にする、Kubernetes 内の PostgreSQL データベースのビジネス継続性態勢を劇的に向上させます（現在、Kubernetes フェデレーションネイティブ機能を待っている間、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、Postgres コンテナの liveness プローブと readiness プローブを定義し、kubelet によって呼び出されます。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、ポッドは正常であると見なされ、終了コード 0（サーバーは通常に接続を受け入れます）および 1（サーバーは、たとえば起動中にサーバーが接続を拒否します）で終了します。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。

ローリングデプロイメント

オペレーターは、ダウンタイムを最小限に抑えるためのローリングデプロイをサポートしています。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、使用可能なポッドにのみ読み取り専用トラフィックを負荷分散します。

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の数をスケールアップまたは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「scale」サブリソースを宣言します。

Kubernetes ノードのメンテナンスウィンドウと 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、メンテナンスオペレーションでクラスター内のすべてのポッドが削除されるのを防ぎ、指定した数のインスタンスを作成できます。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、すべてのワーカーノードでストレージが共有される 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または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場合があります。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、基になるノードのエビクションへの反応を構成できます。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の別の 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機します。

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が誤動作していると思われる場合に、それらのインスタンス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スされると、ポッドが実行されたままで、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、ポッド上のデータが PostgreSQL によって変更されず、デバッグと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（宣言的）

CloudNativePG は *hibernation of a running PostgreSQL cluster* をサポート

`cnpg.io/hibernation` アノテーションを介して、宣言的に。ハイバネーションにより、データベースの PVC を維持しながらデータベースの Pod を削除することにより、CPU の電力を節約できます。この機能は、0 インスタンスへのスケーリングをシミュレートします。

ハイバネーション（必須）

CloudNativePG は *hibernation of a running PostgreSQL cluster* をサポート

`cnpg` プラグインを介して。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 グループ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により、休止状態フェーズを終了できます。

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 メンテナンスオペレーションによってエビクトされたポッドを作成する必要がある場合、`PersistentVolumeClaim` 利用可能な場合は再利用し、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 とメモリのリクエストと制限

オペレーターは、管理者がマニフェストの `resources` セクションを介して、クラスターのポッドによるリソース使用量を制御および管理できるようにします。特に、`requests` および `limits` の値は、CPU と RAM の両方に設定できます。

PgBouncer を使用した接続プーリング

CloudNativePG は、PostgreSQL 用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
PgBouncerIntegrationStatus

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は、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、インスタンスへのクエリフローを最適化し、基になる PostgreSQL リソースの使用をより効率的にします。PostgreSQL サービスに直接接続する代わりに、アプリケーションは PgBouncer サービスに接続し、既存の接続の再利用を開始できるようになりました。

4.39.4 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。特に、モニタリング、アラート、傾向分析、ログ処理です。これには、Prometheus、Grafana、Fluent Bit などの外部ツールや、エラーログを JSON 形式で直接出力するための PostgreSQL エンジンの拡張機能の使用が含まれる場合があります。

CloudNativePG は、柔軟なモニタリングとロギングのための業界標準およびコミュニティで受け入れられた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。

設定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、

> *Prometheus* **オペレーターの例** モニタリング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モニタリングクエリをサポートします。CloudNativePG は、コンテキストに統合して適応できる PostgreSQL の基本的なモニタリングクエリのセットを提供します。

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、タイムスタンプ、ログレベル、および正規の PostgreSQL エラーメッセージチャネルの `postgres` などのログエントリのタイプの第 1 レベルの定義とともに、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 を、ソースデータタイプとして JSON をサポートするダウンストリームのログ処理スタックと簡単かつ直接統合できます。

リアルタイムクエリモニタリング

CloudNativePG は、以下を透過的かつネイティブにサポートします。

- 重要な :ref:``pg_stat_statements` を有効にする<*pg_stat_statements* を有効にする>` 。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および実行統計の追跡を可能にします。
- :ref:`auto_explain` を有効にする<`auto_explain` を有効にする>` 、 ``EXPLAIN

  を手動で実行せずに、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）。
- :ref:``pg_failover_slots` を有効にする<*pg_failover_slots* を有効にする>`

  、論理レプリケーションスロットを物理フェイルオーバー全体で使用可能にし、PostgreSQL のネイティブ論理レプリケーションに基づいた変更データキャプチャ（CDC）コンテキストでの回復力を保証します。

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、監査人、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 ログ内を直接流れ、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 で予想され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` コマンドを使用して表示できます。

4.39.5 レベル 5：オートパイロット

機能レベル 5 は、オブザーバビリティレイヤーから浮上した異常と洞察の発見を通じて、**自動スケーリング** 、 **修復** 、および **チューニング**に焦点を当てています。

自己修復のための自動フェールオーバー

プライマリで障害が検出された場合、オペレーターは、最もアラインさ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により、クラスターのステータスを変更します。その結果、各生きている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、クラスターの要求されたステータスに自分自身を調整するために必要な手順を開始します。元のプライマリが復旧した場合、同じメカニズムにより、アプリケーションがプライマリに到達できないようにし、サーバーで `pg_rewind` を実行し、スタンバイとして再起動することにより、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

スタンバイをホスティング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。

4.40 カスタムポッドコントローラー

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*

コントローラ。同じ仕様から構築された一連のポッドを作成および調整し、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。実装をより複雑にしますが、この設計の選択により、オペレーターはクラスターの管理方法についてより柔軟に対応できるようになり、PostgreSQL クラスターのトポロジは透過的になります。

デザイン領域の多くの選択肢と同様に、さまざまな選択肢が他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り、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。

4.40.1 PVC リサイズ

これは `StatefulSet` のよく知られた制限です。PVC のサイズ変更は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を変更するには、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活用して基になる 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。

4.40.2 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` を使用すると仮定すると、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、特定の操作で実行する正しい展開戦略を決定するときに関係します。

一部の操作は、最初にレプリカで実行してから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した後にのみ実行します。たとえば、別の PostgreSQL イメージバージョンを適用する場合、または `max_connections`（これは treated specially by PostgreSQL because CloudNativePG uses hot standby replicas です）のような構成パラメーターを増やす場合です。

それをしている間、CloudNativePG は PostgreSQL インスタンスのロールを考慮します-シリアル番号だけでなく。

場合によっては、オペレーターは反対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。例えば、`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

`StatefulSet` コントローラーは、アプリケーションに依存しないため、PostgreSQL のネイティブレプリケーションテクノロジーに固有のこの動作を組み込むことはできません。

4.40.3 PVC のコヒーレンス

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。これが、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` が保存されている 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 をドロップした後、 `StatefulSet` はそれを再作成するだけで、PostgreSQL インスタンスの破損につながりました。

CloudNativePG は代わりに、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新しい PVC ペアの作成を開始します。

4.40.4 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

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る場合があります。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稼働するか、新しいノードに置き換えられる可能性があります。

利用できないノードが PostgreSQL インスタンスをホストしていると仮定すると、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できます。

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、その Pod をスケジュールします

2. ポッドをドロップし、別のノードでポッドをスケジュールし、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するのを待ちます。

最初の解決策は、データベースのサイズが許せば実用的であり、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現可能であり、別のホストに PVC を再マウントすることが妥当な時間内に可能です（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 コントローラーはこれらすべての戦略を実装するため、ユーザーはクラスターレベルで好ましい動作を選択できます（詳細については、
Kubernetes アップグレード セクションをお読みください）。

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。

4.41 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見つけることができます。

重要: これらはデモンストレーションおよび実験目的のためのものであり、 クイックスタート で説明されているように、Minikube または Kind を使用して個人用 Kubernetes クラスターで実行できます。

基本クラスター ：

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

  storage:
    size: 1Gi
```

クラスターの基本的な例。

カスタムクラスター：

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

  # Parameters and pg_hba configuration will be append
```

(次のページに続く)

(前のページからの続き)

```
  # to the default ones to make the cluster work
  postgresql:
    parameters:
      max_worker_processes: "60"
    pg_hba:
      # To access through TCP/IP you will need to get username
      # and password from the secret cluster-example-custom-app
      - host all all all md5

  # 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 per instance using default storage class
  storage:
    size: 1Gi
```

postgresql.conf および pg_hba.conf ファイルのデフォルト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：

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgresql-storage-class
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

  # Persistent storage configuration
  storage:
    storageClass: standard
    size: 1Gi
```

: 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。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: pg-backup
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

  # Persistent storage configuration
  storage:
    storageClass: standard
    size: 1Gi

  # Backup properties
  backup:
    barmanObjectStore:
      destinationPath: s3://BUCKET_NAME/path/to/folder
      s3Credentials:
        accessKeyId:
          name: aws-creds
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: aws-creds
          key: ACCESS_SECRET_KEY
      wal:
        compression: gzip
```

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** ：

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

  # Example of rolling update strategy:
```

(次のページに続く)

(前のページからの続き)

```
  # - 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

  # Persistent storage configuration
  storage:
    storageClass: standard
    size: 1Gi

  # Backup properties
  backup:
    barmanObjectStore:
      destinationPath: s3://BUCKET_NAME/path/to/folder
      s3Credentials:
        accessKeyId:
          name: aws-creds
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: aws-creds
          key: ACCESS_SECRET_KEY
      wal:
        compression: gzip
```

適用され、正常：

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

：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：

```
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)

```
  # 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

  # Persistent storage configuration
  storage:
    size: 1Gi
    pvcTemplate:
      accessModes:
        - ReadWriteOnce
      resources:
        requests:
          storage: 1Gi
      storageClassName: standard
      volumeMode: Filesystem
```

：明示的な永続ボリュームクレームテンプレートを使用する基本クラスター。

完全な例：

```
#  Example of definition of a test cluster using all the elements available
#  in the CRD. Please change values appropriately for your environment.
#  Remember that you can take advantage of convention over configuration
#  and normally you dont need to use all these definitions.

apiVersion: v1
data:
  password:␣
↪VHhWZVE0bk44MlNTaVlIb3N3cU9VUlp2UURhTDRLcE5FbHNDRUVlOWJ3RHhNZDczS2NrSWVYelM1Y1U2TGlDMg==
  username: YXBw
kind: Secret
metadata:
  name: cluster-example-app-user
type: kubernetes.io/basic-auth
- --
apiVersion: v1
data:
  password:␣
↪dU4zaTFIaDBiWWJDYzRUeVZBYWNCaG1TemdxdHpxeG1PVmpBbjBRSUNoc0pyU211OVBZMmZ3MnE4RUtLTHBaOQ==
  username: cG9zdGdyZXM=
kind: Secret
```

(次のページに続く)

(前のページからの続き)

```
metadata:
  name: cluster-example-superuser
type: kubernetes.io/basic-auth
- --
apiVersion: v1
kind: Secret
metadata:
  name: backup-creds
data:
  ACCESS_KEY_ID: a2V5X2lk
  ACCESS_SECRET_KEY: c2VjcmV0X2tleQ==
- --
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-full
spec:
  description: "Example of cluster"
  imageName: ghcr.io/cloudnative-pg/postgresql:15.3
  # imagePullSecret is only required if the images are located in a private registry
  # imagePullSecrets:
  #   - name: private_registry_access
  instances: 3
  startDelay: 300
  stopDelay: 300
  primaryUpdateStrategy: unsupervised

  postgresql:
    parameters:
      shared_buffers: 256MB
      pg_stat_statements.max: 10000
      pg_stat_statements.track: all
      auto_explain.log_min_duration: 10s
    pg_hba:
      - host all all 10.244.0.0/16 md5

  bootstrap:
    initdb:
      database: app
      owner: app
      secret:
        name: cluster-example-app-user
    # Alternative bootstrap method: start from a backup
    #recovery:
    #  backup:
```

(次のページに続く)

(前のページからの続き)

```
  #    name: backup-example

superuserSecret:
  name: cluster-example-superuser

storage:
  storageClass: standard
  size: 1Gi

backup:
  barmanObjectStore:
    destinationPath: s3://cluster-example-full-backup/
    endpointURL: http://custom-endpoint:1234
    s3Credentials:
      accessKeyId:
        name: backup-creds
        key: ACCESS_KEY_ID
      secretAccessKey:
        name: backup-creds
        key: ACCESS_SECRET_KEY
    wal:
      compression: gzip
      encryption: AES256
    data:
      compression: gzip
      encryption: AES256
      immediateCheckpoint: false
      jobs: 2
  retentionPolicy: "30d"

resources:
  requests:
    memory: "512Mi"
    cpu: "1"
  limits:
    memory: "1Gi"
    cpu: "2"

affinity:
  enablePodAntiAffinity: true
  topologyKey: failure-domain.beta.kubernetes.io/zone

nodeMaintenanceWindow:
  inProgress: false
  reusePVC: false
```

: 使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：

```
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
```

：「PostGIS クラスタ」の例（詳細は *PostGIS section* を参照）

ストリーミングを介したレプリカクラスター： **前提条件** ：

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

  storage:
    size: 1Gi
```

適用され、正常：

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-replica-example
spec:
  instances: 1

  bootstrap:
    pg_basebackup:
```

(次のページに続く)

(前のページからの続き)

```
    source: cluster-example

  replica:
    enabled: false
    source: cluster-example

  storage:
    size: 1Gi
  # note the namespace default in the host name `cluster-example-rw.default.svc`
  # remember to change accordingly with the namespace of the main cluster
  externalClusters:
  - name: cluster-example
    connectionParameters:
      host: cluster-example-rw.default.svc
      user: streaming_replica
      sslmode: verify-full
      dbname: postgres
    # NOTE: if this cluster is created in a different namespace than the main cluster
    # remember to create the `-replication` and `-ca` secrets in the follower namespace
    # before creating the follower cluster
    sslKey:
      name: cluster-example-replication
      key: tls.key
    sslCert:
      name: cluster-example-replication
      key: tls.crt
    sslRootCert:
      name: cluster-example-ca
      key: ca.crt
```

：ストリーミングレプリケーションを持つ cluster-example に続くレプリカクラスター。

バックアップが構成されたシンプルなクラスター：**前提条件** ：構成では、minio が実行されて動作していることを前提としています。backup.barmanObjectStore を minio パラメーターまたはクラウドソリューションで更新してください。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-with-backup
spec:
  instances: 3
  primaryUpdateStrategy: unsupervised

  # Persistent storage configuration
```

(次のページに続く)

(前のページからの続き)

```
  storage:
    storageClass: standard
    size: 1Gi

  # Backup properties
  # This assumes a local minio setup
  backup:
    barmanObjectStore:
      destinationPath: s3://backups/
      endpointURL: http://minio:9000
      s3Credentials:
        accessKeyId:
          name: minio
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: minio
          key: ACCESS_SECRET_KEY
      wal:
        compression: gzip
```

バックアップが構成された基本的なクラスター。

バックアップを介したレプリカクラスター： **前提条件** ：

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: pg-backup
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

  # Persistent storage configuration
  storage:
    storageClass: standard
    size: 1Gi

  # Backup properties
  backup:
```

(次のページに続く)

(前のページからの続き)

```
    barmanObjectStore:
      destinationPath: s3://BUCKET_NAME/path/to/folder
      s3Credentials:
        accessKeyId:
          name: aws-creds
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: aws-creds
          key: ACCESS_SECRET_KEY
      wal:
        compression: gzip
```

適用され、正常。そして、バックアップ

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

適用され、完了しました。 ：

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-replica-from-backup-simple
spec:
  instances: 1

  bootstrap:
    recovery:
      source: cluster-example-backup

  replica:
    enabled: true
    source: cluster-example-backup

  storage:
    size: 1Gi

  externalClusters:
  - name: cluster-example-backup
    barmanObjectStore:
```

(次のページに続く)

(前のページからの続き)

```
      destinationPath: s3://backups/
      endpointURL: http://minio:9000
      s3Credentials:
        accessKeyId:
          name: minio
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: minio
          key: ACCESS_SECRET_KEY
```

：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 ファイルを使用したブートストラップクラスター：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: post-init-sql-configmap
data:
  configmap.sql: |
    create table configmaps (i integer);
    insert into configmaps (select generate_series(1,10000));
- --
apiVersion: v1
kind: Secret
metadata:
  name: post-init-sql-secret
stringData:
  secret.sql: |
    create table secrets (i integer);
    insert into secrets (select generate_series(1,10000));
- --
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-initdb
spec:
  instances: 3

  bootstrap:
    initdb:
      database: appdb
      owner: appuser
      postInitSQL:
        - create table numbers (i integer)
```

(次のページに続く)

(前のページからの続き)

```
      - insert into numbers (select generate_series(1,10000))
    postInitTemplateSQL:
      - create extension intarray
    postInitApplicationSQL:
      - create table application_numbers (i integer)
      - insert into application_numbers (select generate_series(1,10000))
    postInitApplicationSQLRefs:
      configMapRefs:
      - name: post-init-sql-configmap
        key: configmap.sql
      secretRefs:
      - name: post-init-sql-secret
        key: secret.sql

  storage:
    size: 1Gi
```

：データベースが作成された直後に Secret と ConfigMap で定義されたクエリのセットを実行するクラスターの例。

pg_hba 構成がカスタマイズされたサンプルクラスター：

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  postgresql:
    pg_hba:
      - hostssl app all all cert

  storage:
    size: 1Gi
```

：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。

プロジェクトボリュームテンプレートを使用してマウントされた Secret と Configmap を含むサンプルクラスター：

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

(次のページに続く)

(前のページからの続き)

```
  projectedVolumeTemplate:
    sources:
      - secret:
          name: sample-secret
          items:
            - key: tls.crt
              path: certificate/tls.crt
            - key: tls.key
              path: certificate/tls.key
      - configMap:
          name: sample-configmap
          items:
            - key: key1
              path: config/key1
            - key: key2
              path: config/key2
  storage:
    size: 1Gi

- --
apiVersion: v1
data:
  tls.crt:␣
↪LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNQekNDQWVXZ0F3SUJBZ0lRSHVDY2lKcDVkZis0dHZBcDBrTk9rekFLQmdr
  tls.key:␣
↪LS0tLS1CRUdJTiBFQyBQUklWQVRFIEtFWS0tLS0tCk1IY0NBUUVFSUtucWxVQjFqU2sxWW5VZzAyb0tGbXlRdDJLUEZwaFc0K1lm
kind: Secret
metadata:
  name: sample-secret
type: kubernetes.io/tls
- --
apiVersion: v1
data:
  key1: value1
  key2: value2
  key3: value3
kind: ConfigMap
metadata:
  name: sample-configmap
```

プロジェクションボリュームマウントを使用して、既存の Secret と Configmap を Postgres ポッドにマウントした基本クラスター。

宣言的なロール管理を備えたクラスター：

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-with-roles
spec:
  instances: 3
  storage:
    size: 1Gi

  managed:
    roles:
    - name: app
      createdb: true
    - name: dante
      ensure: present
      comment: my database-side comment
      login: true
      superuser: false
      createdb: true
      createrole: false
      inherit: false
      replication: false
      bypassrls: false
      connectionLimit: 4
      validUntil: "2053-04-12T15:04:05Z"
      inRoles:
        - pg_monitor
        - pg_signal_backend
      passwordSecret:
        name: cluster-example-dante
- --
apiVersion: v1
data:
  username: ZGFudGU=
  password: ZGFudGU=
kind: Secret
metadata:
  name: cluster-example-dante
type: kubernetes.io/basic-auth
```

：managed スタンザでロールを宣言し、kubernetes secrets によるパスワード管理を含みます

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。

4.42 ネットワーキング

CloudNativePG は、基になる Kubernetes クラスターに必要な接続が既に設定されていることを前提としています。Kubernetes でのネットワーキングは、重要かつ拡張されたトピックです。詳細については、*Kubernetes documentation* を参照してください。

クイックスタートガイドに従って、ローカル KinD または K3d クラスターに CloudNativePG をインストールしている場合、どちらのプラットフォームもデフォルトでネットワーク制限を追加しないため、ネットワークの問題は発生しません。

ただし、既存のインフラストラクチャに CloudNativePG をデプロイする場合、ネットワークの制限が適用され、オペレーターと PostgreSQL クラスターとの通信が損なわれる可能性があります。具体的には、既存の **ネットワークポリシー**

特定の種類のトラフィックを制限する場合があります。

または、セキュリティを強化するために、環境にネットワークポリシーを追加することに関心がある場合があります。

security document で述べたように、オペレーターがポート 8000 および 5432 のすべてのクラスターポッドに到達でき、ポッドが相互に接続できることを確認してください。

4.42.1 オペレーターのクロスネームスペースネットワークポリシー

クイックスタートガイドに従うか、展開に helm チャートを使用すると、オペレーターは専用の名前空間（デフォルトで `cnpg-system` ）にインストールされます。別の名前空間にクラスターを作成することをお勧めします。

オペレーターはクラスターポッドに接続できる**必要があります**。クロスネームスペーストラフィックを制限する `NetworkPolicy` がある場合、これは除外される可能性があります。

たとえば、 *kubernetes guide on network policies*

には、デフォルトですべての入力トラフィックを拒否するポリシーの例が含まれています。

ローカルの kubernetes セットアップにこの種の制限的なネットワークポリシーがある場合、`NetworkPolicy` を作成して、オペレーター名前空間とポッドからクラスター名前空間とポッドへの接続を明示的に許可する必要があります。で例を見つけることができます

```
#  This network policy allows ingress from the CNPG operator

#  installed in namespace cnpg-system into the pods corresponding

#  to a cluster named cluster-example

apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-operator
spec:
```

(次のページに続く)

(前のページからの続き)

```
  podSelector:
    matchLabels:
      cnpg.io/cluster: cluster-example # The label value must be the cluster name
  ingress:
    - from:
        - namespaceSelector:
            matchLabels:
              kubernetes.io/metadata.name: cnpg-system # Namespace where the operator␣
↪is deployed
          podSelector:
            matchLabels:
              app.kubernetes.io/name: cloudnative-pg # Matches the Operator pod
      ports:
        - port: 8000
        - port: 5432
```

このリポジトリ内のファイル。クラスター名とクラスター名前空間を調整して特定のセットアップに合わせ、オペレーター名前空間がデフォルトの名前空間でない場合は調整する必要があることに注意してください。

4.42.2 クロスクラスターネットワーキング

別のクラスターから *bootstrapping* している間、または `externalClusters` セクションを使用している場合、関連するすべてのクラスター、オブジェクトストア、および名前空間間の接続を確認します。

繰り返しますが、 *Kubernetes documentation* を参照してください

セットアップ情報について。

4.43 ベンチマーク

CNPG kubectl プラグインは、CloudNativePG を使用して Kubernetes で PostgreSQL デプロイメントをベンチマークする簡単な方法を提供します。

ベンチマークは 2 つの側面に焦点を当てています。

- *pgbench* に依存することによる **データベース**
- *fio* に依存することによる **ストレージ**

重要: *pgbench* および `fio` は、ステージングまたは本番前環境で実行する必要があります。データベースや、同じ共有環境で実行される他のワークロード/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、これらのプラグインを実稼働環境で使用しないでください。

4.43.1 pgbench

kubectl CNPG プラグインコマンド pgbench は、既存の Postgres クラスターに対してユーザー定義の pgbench ジョブを実行します。

--dry-run フラグを介して、後で変更/実行するためにジョブのマニフェストを生成できます。

pgbench を使用した一般的なコマンド構造は次のとおりです。

```
kubectl cnpg pgbench \
  -n <namespace> <cluster-name> \
  --job-name <pgbench-job> \
  --db-name <db-name> \
  -- <pgbench options>
```

重要: *pgbench* を参照してください

ジョブで使用する特定のオプションについては。

この例では、pgbench OLTP のような目的で、1000 のスケール係数を使用して cluster-example という名前の Cluster 内の app データベースを初期化する pgbench-init というジョブを作成します。

```
kubectl cnpg pgbench \
  --job-name pgbench-init \
  cluster-example \
  -- --initialize --scale 1000
```

注釈: これにより、100000000 レコードのデータベースが生成され、ディスク上に約 13GB の領域が必要になります。

ジョブの進行状況は次のコマンドで確認できます。

```
shell    kubectl logs jobs/pgbench-run
```

次の例では、単一の接続を使用して、以前に初期化されたデータベースに対して pgbench を 30 秒間実行する pgbench-run というジョブを作成します。

```
kubectl cnpg pgbench \
  --job-name pgbench-run \
  cluster-example \
  -- --time 30 --client 1 --jobs 1
```

次の例では、 --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に対して pgbench を実行します。

```
kubectl cnpg pgbench \
  --db-name pgbench \
  --job-name pgbench-job \
  cluster-example \
  -- --time 30 --client 1 --jobs 1
```

特定のワーカーノードで pgbench ジョブを実行する場合は、--node-selector オプションを使用できます。 workload=pgbench ラベルを持つノードで以前の初期化ジョブを実行すると仮定すると、次を実行できます。

```
kubectl cnpg pgbench \
  --db-name pgbench \
  --job-name pgbench-init \
  --node-selector workload=pgbench \
  cluster-example \
  -- --initialize --scale 1000
```

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
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>
```

4.43.2 fio

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。--dry-run フラグを使用して、後で変更/実行するためにジョブのマニフェストを生成できます。

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メントを作成します。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。

デフォルトの使用例：

```
kubectl cnpg fio <fio-name>
```カスタム値の例：

```shell
```

(次のページに続く)

(前のページからの続き)

```
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
```

展開ステータスは、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。

それはします：

1.PVC を作成する

1.fio ジョブの構成を表す ConfigMap を作成します

1. 単一の Pod で構成される fio デプロイメントを作成します。PVC で fio を実行し、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>`__* 画像を使用します。

デプロイメントによって作成された Pod は、結果の提供を開始すると準備が整います。デプロイで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
```

ブラウザを使用して http://localhost:8000/ に接続し、データを取得します。

デフォルトの 8k ブロックサイズは、PostgreSQL ワークロードをエミュレートするために選択されています。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場合があります。

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シーケンシャル書き込みの例図です（1 時間のベンチマーク）。

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、必要に応じて名前空間を追加してください。

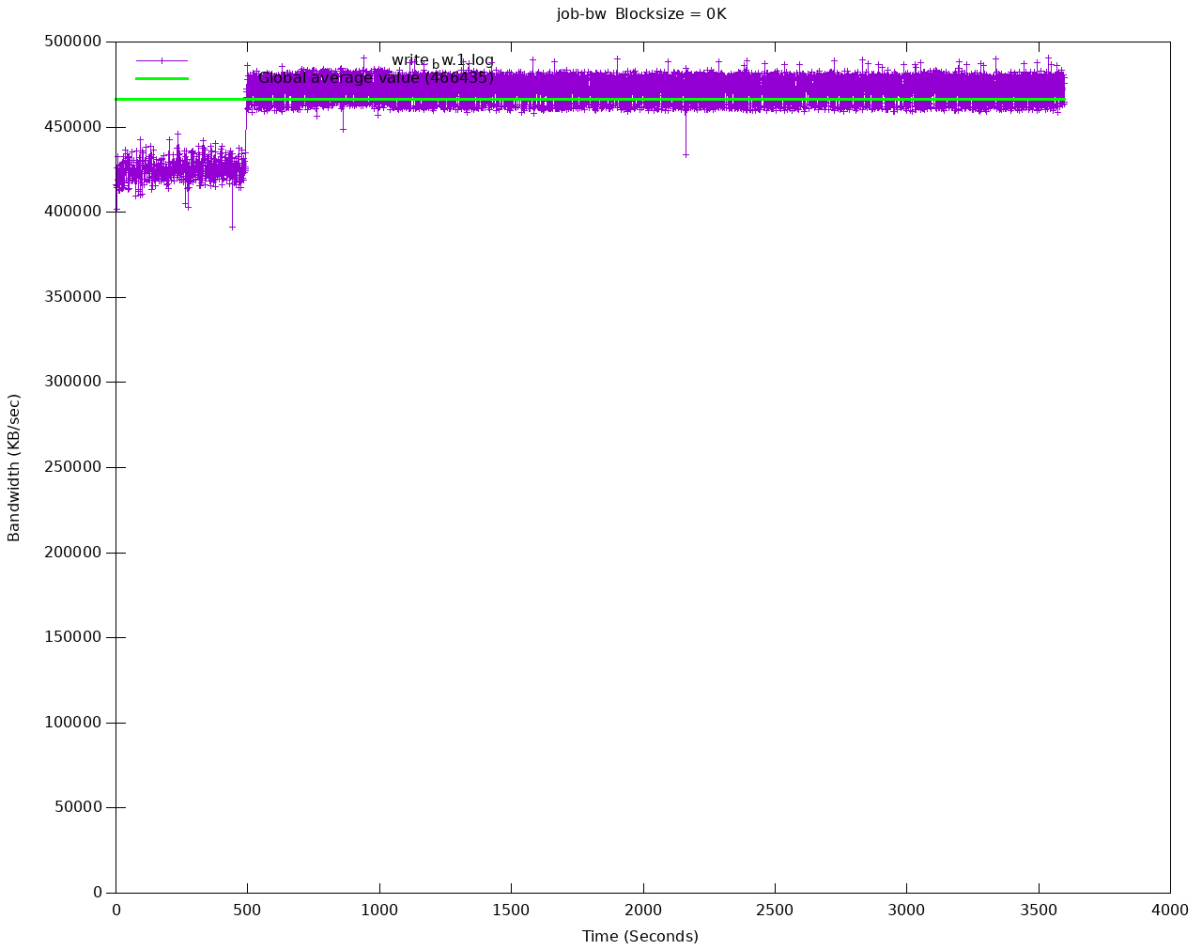


図 18: Sequential writes bandwidth

4.44 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、企業を推奨していません。

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提供されています。

Company	URL	Regions/Countries
スケジュールされたバックアップ	enterprisedb.com	Global

4.45 よくある質問（FAQ）

4.45.1 Kubernetes で PostgreSQL を実行する

PostgreSQL のようなステートフルなワークロードは Kubernetes で実行できないことは誰もが知っています。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 により、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 はステートフルワークロードの準備ができていると信じており、彼らの 70 %は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（一般的に Kubernetes とクラウドネイティブは学習曲線が急勾配です）や Kubernetes オペレーターの品質など、大きな課題が残っています。後者が、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考えている理由です。

私たちのようなデータベースマニアにとって、本当のゲームチェンジャーは、

Kubernetes 1.14 in April 2019 でローカル永続ボリュームのサポートが導入されたことです。

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よると、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イメージは、メインアプリケーションを単一のエントリポイント（いわゆる PID 1 プロセス）として実行するように構築されます。

Kubernetes 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 にマッピングすると、不変アプリケーションコンテナは、実行され、不変コンテナイメージ内の単一の特定のバージョンに関連付けられた単一の「postgres」プロセスです。

SSH や 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は、コンテナ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とは対照的です。

不変とは、コンテナ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、展開がより安全で反復可能になります。

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、用語「 *Cloud Native* 」を定義しています。ただし、2ndQuadrant 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 3 つの主要な概念として解釈してきました。

1. 人、原則とプロセスに基づいた、既存の健全で本物の繁栄した DevOps 文化。より安全で、より効率的で、より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. これらのコンテナ（Kubernetes など）を管理およびオーケストレーションする方法

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。

私たちの共感を呼ぶ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

ベアメタル Kubernetes で CloudNativePG を実行できますか?

はい、間違いなく。ベアメタルで Kubernetes を実行できます。また、ローカルに接続されたストレージを備えた 1 つ以上のフィジカルワーカーノードを PostgreSQL ワークロード専用にして、最大かつ予測可能な I / O パフォーマンスを実現できます。

CloudNativePG が由来する実際のクラウドネイティブ PostgreSQL プロジェクトは、2019 年のパイロットプロジェクトの後に生まれました。同じベアメタルサーバー上のストレージと PostgreSQL を、最初は Linux で直接、次に Kubernetes 内でベンチマークしました。予想通り、実験では、ローカル永続ボリュームを介して Kubernetes で実行されているコンテナによって導入されたパフォーマンスへの影響は無視できる程度であり、クラウドネイティブイニシアチブを続行できます。

ファイルシステムレプリケーションの代わりに PostgreSQL レプリケーションを使用する必要があるのはなぜですか?

> *Architecture: Synchronizing the state* をお読みください

セクション。

PostgreSQL をコンテナとして実行する代わりに演算子を使用する必要があるのはなぜですか?

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用意し、レプリカのない Postgres コンテナ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あります。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、ポッドを別の 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機能であり、複雑なアプリケーションがビジネス継続性コンテキストでどのように機能するかを定義します。オペレーターパターンは、この目的のための Kubernetes の現在最先端です。オペレーターは、自動化されたプログラム的な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（最初のステップ）だけでなく、予期しないイベントの後に適切に反応する必要があります。典型的な例はフェイルオーバーの例です。

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、リカバリ、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、ログ管理およびモニタリングインフラストラクチャへの PostgreSQL クラスターの統合も容易にします。

CloudNativePG では、宣言構成を介して PostgreSQL クラスターの目的の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る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、調整ループは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が活躍します。

あなたは、CloudNativePG は完全に宣言型の演算子だと言います。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使用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、3 番目のインスタンスのクローンを作成することにより、3 ノードの PostgreSQL クラスターを取得できます。

宣言型アプローチでは、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、変更管理操作が非常に簡素化され、これらが Git などのソース管理システムに保存されると、Infrastructure as Code 機能が有効になります。また、Kubernetes は、リクエストがいつでも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 チームで 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 のコア概念に精通し、Kubernetes 管理者と対話できるときです。

Cloud Native 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へのアドバイスです。

CloudNativePG が StatefulSet 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、代わりに、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により、基になる PVC を直接管理します。この決定の詳細と理由については、

> カスタムポッドコントローラー セクションを参照してください。

4.45.2 高可用性

オペレーターポッドが停止した場合、または一定時間利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には利用できない場合があります。

ただし、停止が PostgreSQL クラスターを実行しているノードに影響しないと仮定すると、データベースは関連する Kubernetes サービスを介して、通常のオペレーションを提供し続けます。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起動していることを確認します。

要約すると：

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ものではありません。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。

CloudNativePG が Patroni、repmgr、Stolon などのフェイルオーバー管理ツールに依存しない理由は何ですか?

CloudNativePG を開発するチームの一部は、過去に repmgr に大きく関与していましたが、別のアプローチを採用して Kubernetes コントローラーを直接拡張し、Kubernetes API サーバーに依存して Postgres クラスターのステータスを保持し、使用することにしました。唯一の真実の情報源として：

- 主に自動フェイルオーバーとスイッチオーバーを介して Postgres クラスターの高可用性を制御し、**インスタンスマネージャーのインプレース更新** と自分自身を調整します
- アプリケーションのエントリポイントである Kubernetes サービスを制御します

4.46 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**
- **クラスター**
- **プーラー**
- **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があります。

- *AffinityConfiguration*
- *AzureCredentials*

- バックアップ
- *BackupConfiguration*
- バックアップリスト
- バックアップソース
- *BackupSpec*
- *BackupStatus*
- *BarmanCredentials*
- *BarmanObjectStoreConfiguration*
- *BootstrapConfiguration*
- *BootstrapInitDB*
- *BootstrapPgBaseBackup*
- *BootstrapRecovery*
- *CertificatesConfiguration*
- *CertificatesStatus*
- クラスター
- *ClusterList*
- *ClusterSpec*
- *ClusterStatus*
- *ConfigMapKeySelector*
- *ConfigMapResourceVersion*
- *DataBackupConfiguration*
- データソース
- *EmbeddedObjectMetadata*
- *ExternalCluster*
- *Google* 資格情報
- インポート
- *ImportSource*
- インスタンス *ID*
- *InstanceReportedState*
- *LDAPBindAsAuth*
- *LDAPBindSearchAuth*

- *LDAPConfig*
- *LocalObjectReference*
- *ManagedConfiguration*
- 管理対象ロール
- *EmbeddedObjectMetadata*
- *MonitoringConfiguration*
- *NodeMaintenanceWindow*
- *PasswordState*
- *PgBouncerIntegrationStatus*
- *PgBouncerSecrets*
- *PgBouncerSpec*
- *PodTemplateSpec*
- プーラー
- *PoolerIntegrations*
- *PoolerList*
- *PoolerMonitoringConfiguration*
- *PoolerSecrets*
- *PoolerSpec*
- *PoolerStatus*
- *PostInitApplicationSQLRefs*
- *PostgresConfiguration*
- *RecoveryTarget*
- *ReplicaClusterConfiguration*
- *ReplicationSlotsConfiguration*
- *ReplicationSlotsHAConfiguration*
- *RoleConfiguration*
- *RollingUpdateStatus*
- *S3* 資格情報
- スケジュールされたバックアップ
- *ScheduledBackupList*
- *ScheduledBackupSpec*

- *ScheduledBackupStatus*
- *SecretKeySelector*
- *SecretVersion*
- *SecretsResourceVersion*
- *ServiceAccountTemplate*
- *StorageConfiguration*
- *SyncReplicaElectionConstraints*
- トポロジー
- *WalBackupConfiguration*

4.46.1 AffinityConfiguration

AffinityConfiguration には、ポッド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`	ポッドのアンチアフィニティをアクティブにします。オペレーターは、このフィールドが明示的に false に設定されていない限り、ポッドのアンチアフィニティを定義します	*bool
`topologyKey`	アンチ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
`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説明します。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実行できるようにすべてのポッド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

次のページに続く

表 4 – 前のページからの続き

Name	Description	Type
`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アンチアフィニティーをスケジューリング中に強力な要件と見なす必要があるかどうかを決定できます。許可され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」。「required」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中のまま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、オペレーターによって生成された用語に追加するポッド反アフィニティー用語を指定でき、false に設定されている場合は排他的に使用されます。	*corev1.PodAntiAffinity
`additionalPodAffinity`	AdditionalPodAffinity を使用すると、すべてのクラスターのポッドに渡すポッドアフィニティ用語を指定できます。	*corev1.PodAffinity

4.46.2 AzureCredentials

AzureCredentials は、Azure Blob Storage にファイルを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、ストレージアカウント名と、次の 1 つ（および 1 つだけ）が必要です。

- 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	*networking page*
`storageAccount`	データをアップロードするストレージアカウント	*networking page*
`storageKey`	ストレージアカウント名と組み合わせて使用されるストレージアカウントキー	*networking page*
`storageSasToken`	ストレージアカウント名と組み合わせて使用される共有アクセス署名	*networking page*

次のページに続く

表 5 – 前のページからの続き

Name	Description	Type
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

4.46.3 バックアップ

バックアップはバックアップ API のスキーマです

Name	Description	Type
metadata	該当なし	*networking page*
`spec `	バックアップの目的の動作の仕様。詳細： https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status	*BackupSpec*
`status `	バックアップの最後に観察されたステータス。このデータは最新でない場合があります。システムによって入力されます。読み取り専用。詳細： https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status	*BackupStatus*

4.46.4 BackupConfiguration

BackupConfiguration は、クラスターのバックアップを取得する方法を定義します。現在サポートされているバックアップ方法は barmanObjectStore のみです。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	*networking page*
`retentionPolicy `	RetentionPolicy は、バックアップと WAL（つまり '60d'）に使用される保持ポリシーです。保持ポリシーは`XXu`の形式で表されます。ここで、`XX`は正の整数であり、`u`は`[dwm]`日、週、月です。	string

次のページに続く

表 7 – 前のページからの続き

Name	Description	Type
`target `	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。利用可能なオプションは空の文字列で、デフォルトは`prefer-standby`ポリシーになります。`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ます。`prefer-standby`はできれば最新のスタンバイでバックアップを実行します。	BackupTarget

4.46.5 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	標準リストのメタデータ。詳細： https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#types-kinds	*networking page*
`items `	バックアップのリスト - **必須**	*networking page*

4.46.6 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 バンドルを保存します。自己署名証明書を使用して、証明書発行者と barman-cloud-wal-archive のエラーを回避する場合に役立ちます。	*networking page*

4.46.7 BackupSpec

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*

次のページに続く

表 10 – 前のページからの続き

Name	Description	Type
`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、デフォルトは `cluster.spec.backup.target`です。利用可能なオプションは、空の文字列、`primary`および`prefer-standby`です。 `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ます。`prefer-standby`はできれば最新のスタンバイでバックアップを実行します（可能な場合）。	BackupTarget

4.46.8 BackupStatus

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 バンドルを保存します。自己署名証明書を使用して、証明書発行者と barman-cloud-wal-archive のエラーを回避する場合に役立ちます。	*networking page*
`endpointURL`	自動エンドポイント検出をオーバーライドして、クラウドにデータを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、さまざまな宛先フォルダーを使用して、WAL とデータに使用されます。これは、エラーが発生した場合に設定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すると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 のバックアップの名前	string
`phase`	最後のバックアップの状態	BackupPhase
`startedAt`	バックアップが開始された日時	*networking page*
`stoppedAt`	バックアップが終了したとき	*networking page*
`beginWal`	開始 WAL	string
`endWal`	エンディング WAL	string
`beginLSN`	開始の xlog	string
`endLSN`	エンディング 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

次のページに続く

表 11 – 前のページからの続き

Name	Description	Type
`commandError`	エラーが発生した場合のバックアップコマンド出力	string
`instanceID`	バックアップの取得元のインスタンスを特定する情報	*networking page*

4.46.9 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	Google Cloud Storage にデータをアップロードするために使用する認証情報	*networking page*
`s3Credentials`	S3 にデータをアップロードするために使用する資格情報	*networking page*
`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	*networking page*

4.46.10 BarmanObjectStoreConfiguration

BarmanObjectStoreConfiguration には、S3 互換オブジェクトストレージに対する 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、クラウドにデータを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 バンドルを保存します。自己署名証明書を使用して、証明書発行者と barman-cloud-wal-archive のエラーを回避する場合に役立ちます	*networking page*
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、さまざまな宛先フォルダーがあり、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するとクラスター名が使用されます	string

次のページに続く

表 13 – 前のページからの続き

Name	Description	Type
`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、WAL 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に暗号化されない場合があります。	*networking page*
`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に暗号化されない場合があります。	*networking page*
`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

4.46.11 BootstrapConfiguration

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もののうち、単一のブートストラップメソッドのみを定義できます。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	*networking page*
`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	*networking page*
pg_basebackup	互換性のある別の 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します	*networking page*

4.46.12 BootstrapInitDB

BootstrapInitDB は、initdb が使用され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

次のページに続く

表 15 – 前のページからの続き

Name	Description	Type
`owner`	アプリケーションが使用す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資格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	*networking page*
`options`	クラスターを作成するときに 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、明示的に提供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（1〜1024）の値（デフォルト：空、結果 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すること（デフォルトでは空）	[]string
`postInitApplicationSQL`	が作成された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こと（デフォルトでは空）	[]string
`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すること（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	*networking page*

次のページに続く

表 15 – 前のページからの続き

Name	Description	Type
postInitApplication-SQLRefs	PostInitApplicationSQLRefs は、SQL ファイルを含む ConfigMap または Secrets への参照をポイントします。これらの参照への一般的な実装順序は、すべての Secrets からすべての ConfigMaps へであり、Secrets または ConfigMaps 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）	*networking page*

4.46.13 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 `	アプリケーションが使用す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資格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	*networking page*

4.46.14 BootstrapRecovery

BootstrapRecovery には、外部クラスター、ボリュームスナップショット、またはバックアップオブジェクトの 3 つの方法を使用して既存のクラスターから復元するために必要な構成が含まれています。完全復旧とポイントインタイム復旧がサポートされています。この方法は、連続リカバリー（レプリカクラスター）でクラスターを作成するためにも使用でき、`instances` > 1 の場合にカスケードレプリケーションもサポートします。クラスターがリカバリーを終了すると、スーパーユーザーのパスワードは指定されたシークレットを介して変更されます。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
`backup `	リカバリ手順を開始する物理ベースバックアップを含むバックアップオブジェクト。`source`および `volumeSnapshots`とは相互に排他的です。	*networking page*

次のページに続く

表 17 – 前のページからの続き

Name	Description	Type
`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 `backup`および `volumeSnapshots`と相互に排他的です。	string
volumeSnapshots	リカバリ手順を開始する静的 PVC データソース。現在、CloudNativePG と互換性のある既存の PVC グループをマップし、フェンスで囲まれた Postgres インスタンスでコールドバックアップコピーで取得される `VolumeSnapshot`および `PersistentVolumeClaim`リソースをサポートしています（オンラインバックアップが実装されるときに将来削除される制限）。 `backup`および `source`とは相互に排他的です。	*networking page*
`recoveryTarget `	デフォルトでは、回復プロセスはアーカイブ内の利用可能なすべての WAL ファイルを適用します（完全回復）。ただし、PostgreSQL で予想されるように、 `RecoveryTarget`オブジェクト（つまり、タイムスタンプ、トランザクション ID、LSN、.. 。）。詳細： https://www.postgresql.org/docs/current/runtime-config-wal.html#RUNTIME-CONFIG-WAL-RECOVERY-TARGET	*networking page*
`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 `	アプリケーションが使用す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資格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	*networking page*

4.46.15 CertificatesConfiguration

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、新しいシークレットは自己署名 CA で作成され、TLS 証明書 ServerTLSSecret の生成に使用されます。 内容： - *ca.crt*：その CA クライアント接続文字列で `sslrootcert`として使用される、サーバー証明書の検証に使用する必要があります。 - *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、`ssl_cert_file`および`ssl_key_file`として設定されるサーバー TLS 証明書とキーを含む kubernetes.io/tls タイプのシークレット。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クライアント証明書を含む kubernetes.io/tls タイプのシークレット。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 で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*：その CA すべてのインスタンスの `ssl_ca_file`として使用される、クライアント証明書の検証に使用する必要があります。 - *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

4.46.16 CertificatesStatus

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

4.46.17 クラスター

Cluster は PostgreSQL API のスキーマです

Name	Description	Type
metadata	該当なし	*networking page*
`spec `	クラスターの目的の動作の仕様。詳細：https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status	*ClusterSpec*
`status `	クラスターの最後に観察されたステータス。このデータは最新でない場合があります。システムによって入力されます。読み取り専用。詳細： https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status	*ClusterStatus*

4.46.18 ClusterList

ClusterList には、クラスターのリストが含まれています

Name	Description	Type
metadata	標準リストのメタデータ。詳細： https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#types-kinds	*networking page*
`items `	クラスターのリスト - **必須**	*networking page*

4.46.19 ClusterSpec

ClusterSpec は、クラスター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`	クラスターに関連するすべてのオブジェクトによって継承されるメタデータ	*networking page*
`imageName`	コンテナイメージの名前、タグ（*<image>:<tag>*）と決定論的で反復可能なデプロイメントのダイジェスト（*<image>:<tag>@sha256:<digestValue>*）の両方をサポート	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、デフォルトは`IfNotPresent`です。更新できません。詳細：https://kubernetes.io/docs/concepts/containers/images#updating-images	corev1.PullPolicy
`schedulerName`	指定すると、ポッドは指定された Kubernetes スケジューラによってディスパッチされます。指定しない場合、ポッドはデフォルトのスケジューラーによってディスパッチされます。詳細：https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/	string
`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 は、スタンバイが利用できないときに書き込みを完了することを許可します。	int
`maxSyncReplicas`	同期レプリケーションクォーラムのターゲット値。準備ができたスタンバイの数がこれよりも少ない場合に減らすことができます。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	*PostgresConfiguration*
`replicationSlots`	レプリケーションスロット管理構成	*networking page*
`bootstrap`	このクラスターをブートストラップする手順	*networking page*
`replica`	レプリカクラスター構成	*networking page*

次のページに続く

表 22 – 前のページからの続き

Name	Description	Type
`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	*networking page*
`enableSuperuserAccess `	このオプションが有効になっている場合、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）。このオプションが無効になっている場合、オペレーターは`SuperuserSecret`コンテンツを無視し、自動作成時に削除し、`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	*networking page*
`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	*networking page*
`storage `	インスタンスのストレージの構成	*StorageConfiguration*
`serviceAccountTemplate `	サービスアカウントの世代を構成する	*networking page*
`walStorage `	PostgreSQL WAL（ログ先行書き込み）のストレージの構成	*networking page*
`startDelay `	PostgreSQL インスタンスが正常に起動できる秒単位の時間（デフォルト 30）	int32
`stopDelay `	PostgreSQL インスタンスが正常にシャットダウンできる秒単位の時間（デフォルト 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できる秒単位の時間。デフォルト値は 40000000 で、1 年を超える秒単位で、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、フェールオーバーをトリガーするまでに待機する時間（秒）	int32
`affinity `	Pod のアフィニティ/アンチアフィニティルール	*AffinityConfiguration*
topologySpreadConstraints	TopologySpreadConstraints は、指定されたトポロジー間で一致するポッドを広げる方法を指定します。詳細： https://kubernetes.io/docs/concepts/scheduling-eviction/topology-spread-constraints/	[]corev1.TopologySpreadConstraint

次のページに続く

表 22 – 前のページからの続き

Name	Description	Type
`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。	*networking page*
`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展開戦略：自動（*unsupervised* - デフォルト）または手動（*supervised*）できます	PrimaryUpdateStrategy
`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の方法：スイッチオーバー（*switchover*）またはインプレース（*restart*-デフォルト）を使用できます	PrimaryUpdateMethod
`backup `	バックアップに使用される構成	*networking page*
`nodeMaintenanceWindow `	Kubernetes ノードのメンテナンスウィンドウを定義する	*networking page*
`monitoring `	このクラスターの監視インフラストラクチャの構成	*networking page*
`externalClusters `	構成で使用される外部クラスターのリスト	*networking page*
`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 フォーマットに従って、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 フォーマットに従って、環境変数ソースを Env で使用されるポッドに渡します	[]corev1.EnvFromSource
`managed `	インスタンスマネージャーによって管理される PostgreSQL の部分によって使用される構成	*networking page*
`seccompProfile `	すべての Pod と Container に適用される SeccompProfile。デフォルト：*RuntimeDefault*	*corev1.SeccompProfile

4.46.20 ClusterStatus

ClusterStatus は、クラスターの監視状態を定義します

Name	Description	Type
`instances `	クラスターで検出された 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たインスタンスの合計数。レディ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
`instancesReported-State `	最後の調整ループ中に報告されたインスタンスの状態	*networking page*
`managedRolesStatus `	ManagedRolesStatus は、クラスター内の管理対象ロールの状態を報告します	管理対象ロール
`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
`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インスタンスとは異なります	string
`pvcCount `	このクラスター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 `	このクラスターによって作成され、まだ利用可能な、ポッドに接続されていないすべての PVC のリスト	[]string
`resizingPVC `	ResizingPVC 条件を持つすべての PVC のリスト。	[]string
`initializingPVC `	このクラスターによって初期化されているすべての PVC のリスト	[]string
`healthyPVC `	ぶら下がっておらず、初期化もしていないすべての PVC のリスト	[]string
`unusablePVC `	別の PVC が見つから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
`secretsResourceVer-sion `	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利益のために行われます	*SecretsResourceVersion*

次のページに続く

表 23 – 前のページからの続き

Name	Description	Type
`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、構成マップデータを更新するインスタンスマネージャーの利益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。	*CertificatesStatus*
`firstRecoverabilityPoint `	RFC3339 形式で日付として保存された最初の回復可能性ポイント	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 `	プライマリへの実際のプロモーションが最後に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
`targetPrimaryTimestamp `	新しいプライマリに対する最後の要求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に必要な統合	*networking page*
`cloudNativePGOperatorHash `	演算子のバイナリのハッシュ	string
`onlineUpdateEnabled `	OnlineUpdateEnabled は、オンラインアップグレードがクラスター内で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 `	クラスター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

4.46.21 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

4.46.22 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

4.46.23 DataBackupConfiguration

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の tar ファイル）を圧縮します。使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とき（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*	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 `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	*int32

4.46.24 データソース

DataSource には、既存のストレージから PostgreSQL クラスターをブートストラップするために必要な構成が含まれています

Name	Description	Type
`storage `	インスタンスのストレージの構成 - **必須**	corev1.TypedLocalObjectReference
walStorage	PostgreSQL WAL（ログ先行書き込み）のストレージの構成	*corev1.TypedLocalObjectReference

4.46.25 EmbeddedObjectMetadata

EmbeddedObjectMetadata には、クラスター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 `	該当なし	map[string]string
annotations	該当なし	map[string]string

4.46.26 ExternalCluster

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	dbname、host、username などの接続パラメーターのリスト	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	*networking page*
`sslKey `	このインスタンスへの接続に使用される SSL プライベートキーへの参照	*networking page*
`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	*networking page*
`password `	サーバーへの接続に使用されるパスワードへの参照	*networking page*
`barmanObjectStore `	barman-cloud ツールスイートの構成	*networking page*

4.46.27 Google 資格情報

GoogleCredentials は、Google Cloud Storage 資格情報のタイプです。GKE 環境内で実行する場合でも、これを指定する必要があります。

Name	Description	Type
`gkeEnvironment `	true に設定されている場合、GKE 環境内で実行されていると想定します。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	*networking page*

4.46.28 インポート

Import 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構成が含まれています

Name	Description	Type
`source `	インポートのソース - **必須**	*ImportSource*
`type `	インポートタイプ。 `microservice`または`monolith`になります。 - **必須**	SnapshotType
`databases `	インポートするデータベース - **必須**	[]string
`roles `	インポートするロール	[]string
postImportApplicationSQL	がインポートされた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
`schemaOnly `	true に設定すると、 *pg_restore* `の `*pre-data* `および `*post-data* `セクションのみが呼び出され、データのインポートが回避されます。デフォルト：*`false*。	bool

4.46.29 ImportSource

ImportSource は、論理スナップショットのソースを説明します

Name	Description	Type
externalCluster	インポートに使用される externalCluster の名前- **必須**	string

4.46.30 インスタンス ID

InstanceID には、インスタンスを識別する情報が含まれています

Name	Description	Type
`podName `	ポッド名	string
ContainerID	コンテナ ID	string

4.46.31 InstanceReportedState

InstanceReportedState は、調整ループ中のインスタンスの最後に報告された状態を示します

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

4.46.32 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	bind 認証オプションのプレフィックス	string
suffix	bind 認証オプションのサフィックス	string

4.46.33 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`	ディレクトリにバインドするユーザーのパスワードを持つシークレット	*networking page*
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

4.46.34 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`および `ldaps`です	LDAPScheme
`tls`	「true」に設定して、LDAP over TLS を有効にします。 「false」がデフォルトです	bool
`bindAsAuth`	認証構成としてバインド	*networking page*
bindSearchAuth	Bind+Search 認証構成	*networking page*

4.46.35 LocalObjectReference

LocalObjectReference には、同じ名前空間内の既知のタイプの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	レファレントの名前。 - **必須**	string

4.46.36 ManagedConfiguration

ManagedConfiguration は、インスタンスマネージャーによって管理される PostgreSQL の部分を表します

Name	Description	Type
roles	`Cluster`によって管理されるデータベースロール	*networking page*

4.46.37 管理対象ロール

ManagedRoles は、クラスターの管理対象ロールのステータスを追跡します

Name	Description	Type
`byStatus `	ByStatus は、各状態のロールのリストを提供します	map[RoleStatus][]string
cannotReconcile	CannotReconcile は、PostgreSQL で調整できないロールを、その原因の説明とともにリストします	map[string][]string
`passwordStatus `	PasswordStatus は、管理対象ロールごとに最後のトランザクション ID とパスワードのシークレットバージョンを提供します	*networking page*

4.46.38 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ていない理由の説明があります。

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と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
annotations	アノテーションは、リソースとともに保存される非構造化キーバリューマップであり、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ります。これらはクエリ不可能であり、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

4.46.39 MonitoringConfiguration

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
`disableDefault-Queries `	デフォルトのクエリを注入するかどうか。クラスターにデフォルトのクエリを挿入しない場合は、 *true* に設定します。デフォルト：false。	*bool
customQueriesCon-figMap	カスタムクエリを含む構成マップのリスト	*networking page*
`customQueriesSecret `	カスタムクエリを含むシークレットのリスト	*networking page*
`enablePodMonitor `	`PodMonitor`を有効または無効にします	bool

4.46.40 NodeMaintenanceWindow

NodeMaintenanceWindow には、基に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（ノードが再び立ち上がるのを待つ）か、しない（他の場所で再作成する - *instances* >1 の場合） - **必須**	*bool

4.46.41 PasswordState

PasswordState は、管理対象 RoleConfiguration のパスワードの状態を表します

Name	Description	Type
`transactionID `	PostgreSQL のロール定義に影響を与える最後のトランザクション ID	int64
resourceVersion	パスワードシークレットのリソースバージョン	string

4.46.42 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	該当なし	[]string

4.46.43 PgBouncerSecrets

PgBouncerSecrets には、pgbouncer によって使用され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	*SecretVersion*

4.46.44 PgBouncerSpec

PgBouncerSpec は、PgBouncer の構成方法を定義します

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、AuthQuery（例 "SELECT usename, passwd FROM pg_shadow WHERE usename=$1"）も指定する必要があり、自動 CNPG クラスター統合はトリガーされません。	*networking page*
`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 usename、passwd 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、自動 CNPG クラスター統合はトリガーされません。	string
`parameters`	PgBouncer に渡される追加パラメーター - 構成できるオプションのリストについては、CNPG ドキュメントを確認してください	map[string]string
`pg_hba`	PostgreSQL ホストベースの認証ルール（pg_hba.conf ファイルに追加される行）	[]string

次のページに続く

表 47 – 前のページからの続き

Name	Description	Type
`paused `	`true`に設定されている場合、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機し、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

4.46.45 PodTemplateSpec

PodTemplateSpec は、ユーザーが Pod 生成のテンプレートを設定できる構造体です。

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 タイプを使用することはできません。

参照： https://github.com/kubernetes-sigs/controller-tools/issues/385 https://github.com/kubernetes-sigs/controller-tools/issues/448 https://github.com/prometheus- operator/prometheus-operator/issues/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	*EmbeddedObjectMetadata*
`spec `	ポッドの目的の動作の仕様。詳細：https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status	corev1.PodSpec

4.46.46 プーラー

Pooler は poolers API のスキーマです

Name	Description	Type
metadata	該当なし	*networking page*
`spec `	該当なし	*PoolerSpec*
`status `	該当なし	*PoolerStatus*

4.46.47 PoolerIntegrations

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	該当なし	*PgBouncerIntegrationStatus*

4.46.48 PoolerList

PoolerList には、プーラーのリストが含まれています

Name	Description	Type
metadata	該当なし	*networking page*
`items `	• **必須**	*networking page*

4.46.49 PoolerMonitoringConfiguration

PoolerMonitoringConfiguration は、特定のプーラーのすべての監視構成を含むタイプです。

クラスターの MonitoringConfiguration をミラーリングしますが、今のところカスタムクエリ部分はありません。

Name	Description	Type
enablePodMonitor	`PodMonitor`を有効または無効にします	bool

4.46.50 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 TLS シークレットバージョン	*SecretVersion*
`serverCA `	サーバー CA シークレットバージョン	*SecretVersion*
`clientCA `	クライアント CA シークレットバージョン	*SecretVersion*

次のページに続く

表 53 – 前のページからの続き

Name	Description	Type
pgBouncerSecrets	PgBouncer が使用するシークレットのバージョン	*networking page*

4.46.51 PoolerSpec

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、プーラーが動作するクラスター参照です。プーラー名は、同じ名前空間内のクラスター名と一致しないでください。 - **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	*networking page*
`pgbouncer `	PgBouncer 構成 - **必須**	*networking page*
deploymentStrategy	pgbouncer が既存のポッドを新しいポッドに置き換えるために使用するデプロイ戦略	*appsv1.DeploymentStrategy
`monitoring `	このプーラーの監視インフラストラクチャの構成。	*networking page*

4.46.52 PoolerStatus

PoolerStatus は、プーラーの監視状態を定義します

Name	Description	Type
`secrets `	構成オブジェクトのリソースバージョン	*networking page*
instances	スケジュールしようとしているポッドの数	int32

4.46.53 PostInitApplicationSQLRefs

PostInitApplicationSQLRefs は、SQL ファイルを含む ConfigMap または Secrets への参照をポイントします。これらの参照への一般的な実装順序は、すべての Secrets からすべての ConfigMaps へであり、Secrets または ConfigMaps 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 Secrets への参照のリストを保持します	*networking page*
configMapRefs	ConfigMapRefs は ConfigMaps への参照のリストを保持します	*networking page*

4.46.54 PostgresConfiguration

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 ホストベースの認証ルール（pg_hba.conf ファイルに追加される行）	[]string
syncReplicaElectionConstraint	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、1 年単位の秒より大きく、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_libraries`	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	*networking page*

4.46.55 RecoveryTarget

RecoveryTarget を使用すると、回復プロセスを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`	ターゲットタイムライン（「最新」または正の整数）	string

次のページに続く

表 58 – 前のページからの続き

Name	Description	Type
`targetXID`	対象トランザクション 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（Log Sequence Number）	string
`targetTime`	RFC3339 標準のタイムスタンプとしてのターゲット時間	string
targetImmediate	一貫した状態に達したらすぐにリカバリを終了します	*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

4.46.56 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。レプリカクラスターは、復旧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

4.46.57 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	*networking page*
`updateInterval`	スタンバイは、`updateInterval`秒（デフォルト 30）ごとにローカルレプリケーションスロットのステータスを更新します。	int

4.46.58 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）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 PostgreSQL の機能であり、ストリーミングクライアント（この特定の場合、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します。

Name	Description	Type
`enabled `	有効にすると、オペレーター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 クラスター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。無効にした場合（デフォルト）、オペレーター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 のオペレーターが管理す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、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_`に設定されます。	string

4.46.59 RoleConfiguration

RoleConfiguration は、Kubernetes での、追加フィールドを備えた PostgreSQL ロールの表現ですデータベース内のロールの有無を保証するかどうかを指定する Ensure

CREATE ROLE コマンドのデフォルトが適用されます参考：https://www.postgresql.org/docs/current/sql-createrole.html

Name	Description	Type
`name `	ロールの名前 - **必須**	string
`comment `	役割の説明	string
`ensure `	ロールが`present`または`absent`であることを確認します-デフォルトは「present」	EnsureOption
`passwordSecret `	ロールのパスワードを含むシークレット（存在する場合） null の場合、DisablePassword が設定されていない限り、パスワードは無視されます	*networking page*

次のページに続く

表 62 – 前のページからの続き

Name	Description	Type
disablePassword	DisablePassword は、Postgres でロールのパスワードを NULL に設定する必要があることを示します	bool
`superuser `	ロールがデータベース内のすべてのアクセス制限をオーバーライドできる `superuser`であるかどうか-スーパーユーザーステータスは危険であり、本当に必要な場合にのみ使用する必要があります。新しいスーパーユーザーを作成するには、自分がスーパーユーザーである必要があります。デフォルトは `false`です。	bool
`createdb `	`true`に設定すると、定義されているロールは新しいデータベースを作成できます。 `false`（デフォルト）を指定すると、ロールはデータベースを作成する機能を拒否されます。	bool
`createrole `	ロールが他のロールの作成、変更、ドロップ、コメント、セキュリティラベルの変更、および他のロールのメンバーシップの付与または取り消しを許可されるかどうか。デフォルトは `false`です。	bool
`inherit `	ロールがメンバーであるロールの特権を「継承」するかどうか。デフォルトは `true`です。	*bool
`login `	ロールにログインが許可されているかどうか。 `login`属性を持つロールは、ユーザーと見なすことができます。この属性のないロールは、データベース権限の管理に役立ちますが、通常の意味でのユーザーではありません。デフォルトは `false`です。	bool
`replication `	ロールがレプリケーションロールかどうか。レプリケーションモード（物理または論理レプリケーション）でサーバーに接続し、レプリケーションスロットを作成またはドロップできるようにするには、ロールがこの属性を持っている（またはスーパーユーザーである）必要があります。 `replication`属性を持つロールは非常に高い特権のロールであり、実際にレプリケーションに使用されるロールでのみ使用する必要があります。デフォルトは `false`です。	bool
`bypassrls `	ロールがすべての行レベルセキュリティ（RLS）ポリシーをバイパスするかどうか。デフォルトは `false`です。	bool
connectionLimit	ロールがログインできる場合、これはロールが作成できる同時接続の数を指定します。 *-1* （デフォルト）は制限がないことを意味します。	int64

次のページに続く

表 62 – 前のページからの続き

Name	Description	Type
`validUntil`	ロールのパスワードが無効になる日時。省略すると、パスワードは無期限になります（デフォルト）。	*networking page*
`inRoles`	このロールが新しいメンバーとしてすぐに追加される 1 つ以上の既存のロールのリスト。デフォルトは空です。	[]string

4.46.60 RollingUpdateStatus

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 に入れる画像 - **必須**	string
startedAt	アップデートが開始されたら	*networking page*

4.46.61 S3 資格情報

S3Credentials は、S3 にファイルを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。2 つの代替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	*networking page*
`secretAccessKey`	シークレットアクセスキーへの参照	*networking page*
`region`	リージョン名を含むシークレットへの参照	*networking page*
`sessionToken`	セッションキーへの参照	*networking page*
inheritFromIAMRole	キーを明示的に提供せずにロールベースの認証を使用します。 - **必須**	bool

4.46.62 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	該当なし	*networking page*
`spec `	ScheduledBackup の目的の動作の仕様。詳細： https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status	*ScheduledBackupSpec*
`status `	ScheduledBackup の最後に観察されたステータス。このデータは最新でない場合があります。システムによって入力されます。読み取り専用。詳細： https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status	*ScheduledBackupStatus*

4.46.63 ScheduledBackupList

ScheduledBackupList には ScheduledBackup のリストが含まれています

Name	Description	Type
metadata	標準リストのメタデータ。詳細： https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#types-kinds	*networking page*
`items `	クラスターのリスト - **必須**	*networking page*

4.46.64 ScheduledBackupSpec

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

次のページに続く

表 67 – 前のページからの続き

Name	Description	Type
`schedule `	スケジュールは、追加の秒指定子が含まれるため、Kubernetes CronJobs で使用される同じ形式に従いません。 https://pkg.go.dev/github.com/robfig/cron#hdr-CRON_Expression_Format-**必須**	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 を置く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：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、デフォルトは`cluster.spec.backup.target`です。利用可能なオプションは、空の文字列、`primary`および`prefer-standby`です。 `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ます。 `prefer-standby`はできれば最新のスタンバイでバックアップを実行します（可能な場合）。	BackupTarget

4.46.65 ScheduledBackupStatus

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	*networking page*
lastScheduleTime	バックアップが最後にスケジュールされたときの情報。	*networking page*
nextScheduleTime	次回はバックアップを実行します	*networking page*

4.46.66 SecretKeySelector

SecretKeySelector には、Secret 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

4.46.67 SecretVersion

SecretVersion には、シークレット名とその ResourceVersion が含まれています

Name	Description	Type
`name `	秘密の名前	string
version	シークレットの ResourceVersion	string

4.46.68 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
managedRoleSecretVersion	管理対象ロールシークレットのリソースバージョン	map[string]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 サーバ側 CA シークレットバージョンのリソースバージョン	string
`serverSecretVersion `	PostgreSQL サーバーサイドシークレットバージョンのリソースバージョン	string
`barmanEndpointCA `	提供されている場合、 Barman Endpoint CA のリソースバージョン	string

次のページに続く

表 71 – 前のページからの続き

Name	Description	Type
`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

4.46.69 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	Metadata 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**	*EmbeddedObjectMetadata*

4.46.70 StorageConfiguration

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。 PVC テンプレートを評価した後に適用されます（可能な場合）。指定しない場合、生成される 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。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、デフォルトは true	*bool
`pvcTemplate `	永続ボリューム要求の生成に使用されるテンプレート	*networking page*

4.46.71 SyncReplicaElectionConstraints

SyncReplicaElectionConstraints には、同期レプリカ選択の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 つのインスタンスは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	ポッド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

4.46.72 トポロジー

トポロジーにはクラストートポロジーが含まれます

Name	Description	Type
successfullyExtracted	successlyExtracted 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の同期レプリカ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
`nodesUsed`	NodesUsed は、インスタンスを収容する個別のノードの数を表します。値「1」は、すべてのインスタンスが単一のノードでホストされていることを示し、高可用性（HA）がないことを意味します。理想的には、この値は Postgres HA クラスター内のインスタンス数と同じである必要があり、コンピューティング側で何も共有されないアーキテクチャを意味します。	int32

4.46.73 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。使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とき（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*	EncryptionType
maxParallel	並行してアーカイブ（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（PostgreSQL スタンバイがリカバリ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）される 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。1 は受け入れられる最小値です。	int

4.47 サポートされているリリース

このページには、*CloudNativePG* の現在サポートされているリリースのステータス、タイムライン、ポリシーがリストされています。

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。これには、アップグレード計画を可能にするための追加の 1 か月の支援が考慮されます。

CloudNativePG のサポートされているリリースには、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のパッチが適用されたリリースが含まれます。

マイナーリリースで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
- ネーミングスキーム
- *CloudNativePG* リリースのサポート状況

- サポートとは

4.47.1 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。

約 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、CloudNativePG コミュニティからのさまざまな製品品質レベルと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 77: :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。		
リリースノート で、各マイナーおよびパッチリリースの高レベルの詳細情報を見つけることができます。		

4.47.2 ネーミングスキーム

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>
```

<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。パッチは通常、 <minor> リリースと比較して小さな変更です。

バージョンの Git タグには、先頭に v が追加されます。

4.47.3 CloudNativePG リリースのサポート状況

Version	Currently Supported	Release Date	End of Life	Supported Kubernetes Versions	Tested, b
1.20.x	Yes	April 27, 2023	~ September 28, 2023	1.24, 1.25, 1.26, 1.27	1.23
1.19.x	Yes	February 14, 2023	~ July 15, 2023	1.23, 1.24, 1.25, 1.26	1.27
main	No, development only				

上記の表のサポートされている Kubernetes バージョンのリストは、CloudNativePG メンテナーがサポートとテストが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もテストもしていません。これは将来変更される可能性があり、その場合、CloudNativePG メンテナーが作成した公式ポリシーに反映されます。

サポートされている PostgreSQL バージョン

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、CloudNativePG の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でサポートされていた PostgreSQL バージョンによって異なります。

PostgreSQL Versioning Policy を参照してください

サポートされているバージョンの詳細については、ページをご覧ください。

注釈: 2022 年 11 月 10 日までに、Postgres 10 は最終リリースに達し、サポートされなくなりました。

アップグレードしないことはアップグレードよりもリスクが高いため、定期的に PostgreSQL オペランドイメージを更新し、使用しているメジャーバージョンの最新のマイナーリリースを使用することをお勧めします。その結果、古いマイナーバージョンの PostgreSQL でイシューを開いた場合、お手伝いできない場合があります。

4.47.4 今後のリリース

Version	Release Date	End of Life	Supported Kubernetes Versions
1.21.0	June 27, 2023	•	•
1.22.0	Aug 29, 2023	•	•

注釈: リリースの 1 週間前に機能のフリーズが発生します

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 バージョンにも適用されます。リリーススケジュールの更新と変更は、 Release updates

メイン GitHub リポジトリでのディスカッション。

4.47.5 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
1.18.x	Nov 10, 2022	June 12, 2023	1.23, 1.24, 1.25, 1.26, 1.27

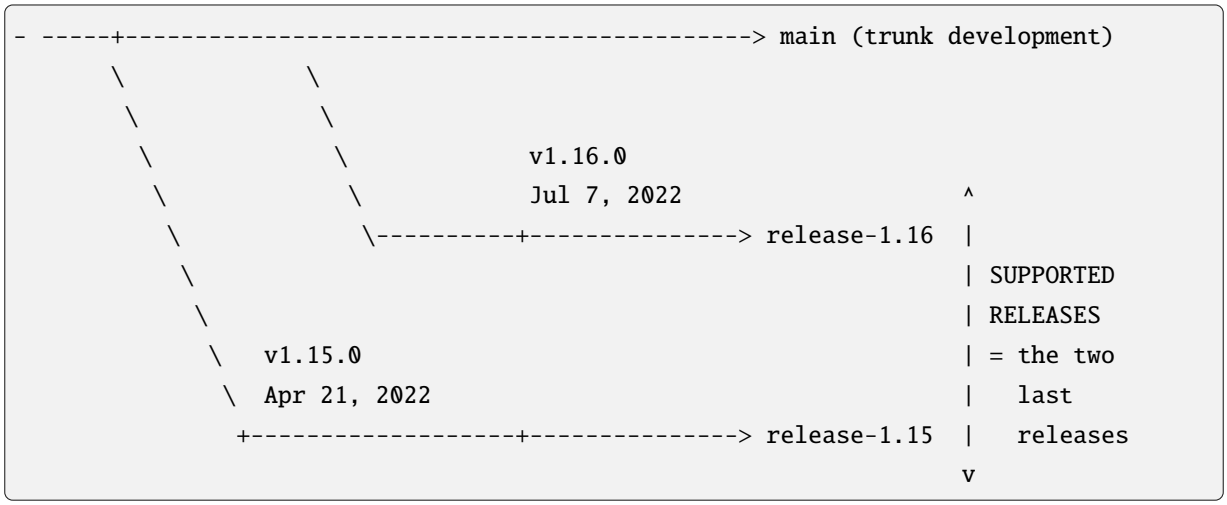
4.47.6 サポートとは

2 か月ごとに新しい最終リリースを作成することを考えると、サポートウィンドウは各リリースブランチ（最新の 2 つのマイナーリリースとさらに 1 か月）の約 5 か月です。

下の図では、 `release-1.16` は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ています。



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、ベストエフォートで提供されます。

> CloudNativePG Slack （ `#general` チャネル）で、または GitHub Discussions

を使用して、コミュニティにサポートを要求できます。

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、次のことを自問します。**「すでにリリースされたバージョンの安定性を本当に重視していることを念頭に置いて、このバックポートは ``*CloudNativePG*`` を改善しますか?」**

専門的なサポートが必要な場合は、 **クラウドプロバイダーのサポート**を参照してください。そこにリスト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契約を提供している場合があります。

4.48 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 1.20* **のリリースノート**
- *CloudNativePG 1.19* **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 1.18* **のリリースノート**
- *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**

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。

4.48.1 CloudNativePG 1.20 のリリースノート

CloudNativePG の 1.20 マイナーリリースにおけるユーザーに見える変更の履歴。

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で。

バージョン 1.20.1

発売日： 2023 年 6 月 12 日

機能強化：

- `snapshot` コマンドを `cnpg` プラグインに追加し、Kubernetes `VolumeSnapshot` 標準リソースを使用してスタンバイからクラスターの一貫したコールドバックアップを作成します (#1960)
- `.spec.bootstrap.recovery.volumeSnapshot` スタンザを介した CSI VolumeSnapshot リソースのセットからのリカバリの最初の実装 (#1960)
- `pg_failover_slots` をマネージドエクステンションに追加 (#2057)
- ドキュメントとクイックスタートガイドの手順を更新して Grafana ダッシュボードを改善 (#1916)
- `import` スタンザに `schemaOnly` オプションを導入し、1 つ以上の既存のデータベースから新しい Postgres クラスターをブートストラップするときにデータのエクスポートとインポートを回避 (#2234)
- インスタンスポッドのスケジューリングを管理する TopologySpreadConstraints のサポートを追加 (#2202)
- PgBouncer の `Pooler` に `PodMonitor` サポートを追加 (#2034)
- デフォルトの Kubernetes スケジューラーをオーバーライドするオプションを追加 (#2013)
- `Pooler` リソースの展開戦略の設定を許可 (#1983)
- デフォルトの PostgreSQL バージョンを 15.3 に更新 (#2022)
- デフォルトで PgBouncer 1.19 を使用する (#2018)

技術的な機能強化：

- k8s の種類テスト済みバージョンを更新 (#2054)
- 宣言的ロールは、設定されていない場合、パスワードを無視するべきであり、以前に存在したロールの管理を容易にする (#2029)
- 別のトランザクションを使用して、ロールの資格情報を調整します。このパッチの前に、オペレーターは、1 つが失敗した場合、すべてのロールの同期を元に戻していました (#2004)
- クラスターの復元中にフェンシングが削除されるように (#1987)

- Pod 削除時のロギングを改善 (#2136)

修正：

- K3 でのセットアップを妨げる可能性のある k3s エンジンでのバインドされていない変数を修正 (#2157)
- メトリックで正しい PG バージョンを報告 (#2126)
- ドキュメントで正しい walStorage キーを使用するように (#2140)
- オペレーターがインスタンスに接続できない場合は調整を停止し、そのような場合には明確な診断を提供します。これは、ネットワークの問題が CloudNativePG の正常な動作を妨げるケースを明確にするのに役立ちます (#2145)、(#2233)、および (#2242)

バージョン 1.20.0

発売日： 2023 年 4 月 27 日

特徴：

- **宣言的なロール管理：** `Cluster` 仕様に `managed.roles` スタンザを導入し、 DDL や `ALTER ROLE` (#1524、#1793、および#1815)
- **PostgreSQL クラスターの宣言的休止：** `cnpg.io/hibernation` と呼ばれる新しいアノテーションを導入して、すべてのポッドを削除し、PVC のみを保持することにより、PostgreSQL クラスターを宣言的に休止します。・この機能は、逆の手順も実装します (#1657)

機能強化：

- `kubectl` の `cnpg` プラグインの `report` コマンドの`--logs` オプションを改善し、利用可能な場合は以前のログも含める (#1811)
- `-any` サービスがデフォルトで無効になりました (#1755)

セキュリティ：

- ローカルファイルを介したオーバーライドを介して `SeccompProfile` のカスタマイズを有効に (#1827)

修正：

- サーバーが初めて起動される前の `initdb` ブートストラップフェーズ中に、ユーザーが提供した PostgreSQL 設定を適用します (#1858)

4.48.2 CloudNativePG 1.19 のリリースノート

CloudNativePG の 1.19 マイナーリリースにおける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で。

バージョン 1.19.3

発売日： 2023 年 6 月 12 日

機能強化：

- `snapshot` コマンドを `cnpg` プラグインに追加し、Kubernetes `VolumeSnapshot` 標準リソースを使用してスタンバイからクラスターの一貫したコールドバックアップを作成します (#1960)
- `.spec.bootstrap.recovery.volumeSnapshot` スタンザを介した CSI VolumeSnapshot リソースのセットからのリカバリの最初の実装 (#1960)
- `pg_failover_slots` をマネージドエクステンションに追加 (#2057)
- ドキュメントとクイックスタートガイドの手順を更新して Grafana ダッシュボードを改善 (#1916)
- `import` スタンザに `schemaOnly` オプションを導入し、1 つ以上の既存のデータベースから新しい Postgres クラスターをブートストラップするときにデータのエクスポートとインポートを回避 (#2234)
- インスタンスポッドのスケジューリングを管理する TopologySpreadConstraints のサポートを追加 (#2202)
- PgBouncer の `Pooler` に `PodMonitor` サポートを追加 (#2034)
- デフォルトの Kubernetes スケジューラーをオーバーライドするオプションを追加 (#2013)
- `Pooler` リソースの展開戦略の設定を許可 (#1983)
- デフォルトの PostgreSQL バージョンを 15.3 に更新 (#2022)
- デフォルトで PgBouncer 1.19 を使用する (#2018)

技術的な機能強化：

- k8s の種類テスト済みバージョンを更新 (#2054)
- 別のトランザクションを使用して、ロールの資格情報を調整します。このパッチの前に、オペレーターは、1 つが失敗した場合、すべてのロールの同期を元に戻していました (#2004)
- クラスターの復元中にフェンシングが削除されるように (#1987)
- Pod 削除時のロギングを改善 (#2136)

修正：

- K3 でのセットアップを妨げる可能性のある k3s エンジンでのバインドされていない変数を修正 (#2157)
- メトリックで正しい PG バージョンを報告 (#2126)
- ドキュメントで正しい walStorage キーを使用するように (#2140)
- オペレーターがインスタンスに接続できない場合は調整を停止し、そのような場合には明確な診断を提供します。これは、ネットワークの問題が CloudNativePG の正常な動作を妨げるケースを明確にするのに役立ちます (#2145)、(#2233)、および (#2242)

バージョン 1.19.2

発売日： 2023 年 4 月 27 日

機能強化：

- kubectl の cnpg プラグインの report コマンドの--logs オプションを改善し、利用可能な場合は以前のログも含める (#1811)
- -any サービスがデフォルトで無効になりました (#1755)

セキュリティ：

- ローカルファイルを介したオーバーライドを介して SeccompProfile のカスタマイズを有効に (#1827)

修正：

- サーバーが初めて起動される前の initdb ブートストラップフェーズ中に、ユーザーが提供した PostgreSQL 設定を適用します (#1858)

バージョン 1.19.1

発売日： 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オーバーライドを許可 (#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がクラスターレベルのターゲットポリシーを使用していました
- debug クラスターのログレベルを initdb ジョブに拡張 (#1503)
- PgBouncer プーラーの 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により、バックアップの可観測性を強化 (#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)
- kubectl の cnpg プラグインに psql コマンドを追加（#1668)、ユーザーが pod（デフォルトでプライマリ）と psql セッションを開始できるようにします

技術的な機能強化：

- 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注入 (#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と一致します (#1650) Barman 3.3 で導入された--name オプションを活用して、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ぐ (#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する (#1442)
- シングルインスタンスクラスター内のクラスターに 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 (#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ことがありました。

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。PostgreSQL 15 への移行をできるだけ早く計画してください。

  Postgres **データベースのインポート** 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` に設定されたときに実行されることを受け入れ、最も整列したレプリカから物理ベースバックアップを取得します (#1162)
- 遅延フェイルオーバー：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る `failoverDelay` パラメーターを導入 (#1366)

機能強化：

- Kubernetes の予測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サポートカスタム環境変数を導入 (#1275)
- `kubectl` の cnpg プラグインに `backup` コマンドを導入し、クラスターの新しいベースバックアップを発行します (#1348)
- ユーザーが WAL ファイルを既存の Postgres クラスター上の専用ボリュームに移動できるようにすることにより、別個の 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` と同等の値、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のメトリックを使用して 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 上の Webhook に関するファイアウォールの問題を文書化 (#1364)

- recovery の 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 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 互換スペースの例を追加 (#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)
- Architecture ドキュメントで Kubernetes レイヤーをより詳細にカバーする (#1432)

技術的な機能強化：

- リリースブランチに毎日のエンドツーエンドのスモークテストを追加 (#1235)

修正：

- streaming_replica ユーザーとして CHECKPOINT の実行をスキップ (#1408)
- waitForWalArchiveWorking が接続エラーに強いように (#1399)
- PVC ロールが常に一貫していることを確認してください (#1380)
- pvcTemplate 使用時に walStorage のリサイズを許可 (#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 reconcile を回避 (#1238)
- 必要ないときに PodMonitor を探さないように (#1213)

4.48.3 CloudNativePG 1.18 のリリースノート

CloudNativePG の 1.18 マイナーリリースにおけるユーザーに見える変更の履歴。

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で。

バージョン 1.18.5

発売日： 2023 年 6 月 12 日

> **警告:** バージョン 1.18 はサポート終了（EOL）になりました。バージョン 1.18.5 は、1.18 マイナーバージョンでリリースされる最後です。

機能強化：

- snapshot コマンドを cnpg プラグインに追加し、Kubernetes VolumeSnapshot 標準リソースを使用してスタンバイからクラスターの一貫したコールドバックアップを作成します (#1960)

- `.spec.bootstrap.recovery.volumeSnapshot` スタンザを介した CSI VolumeSnapshot リソースのセットからのリカバリの最初の実装 (#1960)
- `pg_failover_slots` をマネージドエクステンションに追加 (#2057)
- ドキュメントとクイックスタートガイドの手順を更新して Grafana ダッシュボードを改善 (#1916)
- `import` スタンザに `schemaOnly` オプションを導入し、1 つ以上の既存のデータベースから新しい Postgres クラスターをブートストラップするときにデータのエクスポートとインポートを回避 (#2234)
- インスタンスポッドのスケジューリングを管理する TopologySpreadConstraints のサポートを追加 (#2202)
- PgBouncer の `Pooler` に `PodMonitor` サポートを追加 (#2034)
- デフォルトの Kubernetes スケジューラーをオーバーライドするオプションを追加 (#2013)
- `Pooler` リソースの展開戦略の設定を許可 (#1983)
- デフォルトの PostgreSQL バージョンを 15.3 に更新 (#2022)
- デフォルトで PgBouncer 1.19 を使用する (#2018)

技術的な機能強化：

- k8s の種類テスト済みバージョンを更新 (#2054)
- 別のトランザクションを使用して、ロールの資格情報を調整します。このパッチの前に、オペレーターは、1 つが失敗した場合、すべてのロールの同期を元に戻していました (#2004)
- クラスターの復元中にフェンシングが削除されるように (#1987)
- Pod 削除時のロギングを改善 (#2136)

修正：

- K3 でのセットアップを妨げる可能性のある k3s エンジンでのバインドされていない変数を修正 (#2157)
- メトリックで正しい PG バージョンを報告 (#2126)
- ドキュメントで正しい walStorage キーを使用するように (#2140)
- オペレーターがインスタンスに接続できない場合は調整を停止し、そのような場合には明確な診断を提供します。これは、ネットワークの問題が CloudNativePG の正常な動作を妨げるケースを明確にするのに役立ちます (#2145)、(#2233)、および (#2242)

バージョン 1.18.4

発売日： 2023 年 4 月 27 日

> **警告：バージョン 1.18 は 2023 年 5 月 27 日にサポート終了（EOL）を迎えます**。まだアップグレードしていない場合は、できるだけ早くアップグレードの計画を開始してください。

重要: PostgreSQL 10 は 2022 年 11 月にサポート終了（EOL）に達したため、CloudNativePG は PostgreSQL 10 のサポートを終了します。バージョン 11 以降がサポートされています。 PostgreSQL 15 への移行をできるだけ早く計画してください。 *Postgres* **データベースのインポート** 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。

機能強化：

- `kubectl` の `cnpg` プラグインの `report` コマンドの`--logs` オプションを改善し、利用可能な場合は以前のログも含める (#1811)
- `-any` サービスがデフォルトで無効になりました (#1755)

セキュリティ：

- ローカルファイルを介したオーバーライドを介して `SeccompProfile` のカスタマイズを有効に (#1827)

修正：

- サーバーが初めて起動される前の `initdb` ブートストラップフェーズ中に、ユーザーが提供した PostgreSQL 設定を適用します (#1858)

バージョン 1.18.3

発売日： 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 プーラーの 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により、バックアップの可観測性を強化 (#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)
- `kubectl` の `cnpg` プラグインに `psql` コマンドを追加（#1668）、ユーザーが pod（デフォルトでプライマリ）と `psql` セッションを開始できるようにします

技術的な機能強化：

- 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注入 (#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と一致します (#1650) Barman 3.3 で導入された`--name` オプションを活用して、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ぐ (#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する (#1442)
- シングルインスタンスクラスター内のクラスターに 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)

バージョン 1.18.2

リリース日： 2023 年 2 月 14 日

機能強化：

- Kubernetes の予測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サポートカスタム環境変数を導入 (#1275)
- kubectl の cnpg プラグインに backup コマンドを導入し、クラスターの新しいベースバックアップを発行します (#1348)
- ユーザーが WAL ファイルを既存の Postgres クラスター上の専用ボリュームに移動できるようにすることにより、別個の 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 と同等の値、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のメトリックを使用して 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 上の Webhook に関するファイアウォールの問題を文書化 (#1364)
- recovery の 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 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 互換スペースの例を追加 (#1289)

技術的な機能強化：

- リリースブランチに毎日のエンドツーエンドのスモークテストを追加 (#1235)

修正：

- streaming_replica ユーザーとして CHECKPOINT の実行をスキップ (#1408)
- waitForWalArchiveWorking が接続エラーに強いように (#1399)
- PVC ロールが常に一貫していることを確認してください (#1380)
- pvcTemplate 使用時に walStorage のリサイズを許可 (#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 reconcile を回避 (#1238)
- 必要ないときに PodMonitor を探さないように (#1213)

バージョン 1.18.1

発売日： 2022 年 12 月 21 日

重要なお知らせ：

- CNPG クラスターを識別するラベルとしての postgresql の非推奨が差し迫っていることに関するアラート。このラベルを使用したリモートの場合、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）を一貫してインパクトのある貢献により、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 はサポート終了に達します。

機能強化：

- サービスアカウントのラベルとアノテーションをカスタマイズする：たとえば、IRSA を介した GKE または EKS での ID 管理を介した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 との互換性を改善：Istio の quit エンドポイントのサポートを追加して、Istio サイドカーを使用したジョブが無期限に実行されないように (#967)
- JSON ログでフィールドのリマッピングを許可する: level および ts フィールドが既存のロギングと干渉する可能性のあるユースケースに役立ちます (#843)
- kubectl-cnpg プラグインに fio コマンドを追加 (#1097)
- kubectl-cnpg プラグインの 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)
- ドキュメント - CNPG サンドボックスへの参照を削除 (#1120) - CNPG サンドボックスは非推奨になり、クイックスタートドキュメントでのモニタリングに関する指示を優先しました-“Release updates” ディスカッションへのリンク (#1148) リリースのお知らせと議論のためのデフォルトチャネル - メンテナーの名誉ステータスを GOVERNANCE.md に文書化する (#1033) - メンテナーがコントリビュートを継続する準備ができていない場合にどのように進むべきかを説明- 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・バックアップ指示 (#1184)
- Architecture ドキュメントで Kubernetes レイヤーをより詳細にカバーする (#1432)

修正：

- ブートストラップの PGDATA パーミッションが 750 に適切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ときに、WAL を含む PVC が削除されるように (#1135)
- `--dry-run` 使用時に pgbench マニフェストで欠落していた ApiVersion と Kind を修正しました (#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 (#1145)
- 復元時に設定された pg-wal を尊重する (#1216)
- nodeToClusters マップからレプリカを除外 (#1194)

技術的な機能強化：

- テストの概要に `ciclops` を使用します（#1064）：ciclops GitHub アクションに依存して、そのプロジェクトから改善を継承し、E2E スイートの概要を提供します
- バックポートプルリクエストワークフローを追加 (#965) - パッチがそのように注釈されている場合、リリースブランチに自動的にパッチをバックポート
- e2e テストスイートでオペレーターログレベルを設定可能に (#1094)
- ラベルに基づいたテスト実行を有効にする (#951)
- Go バージョンを 1.18 から 1.19 に更新 (#1166)

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 クラスターハイバネーション**：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ハイバネーションを導入します。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 を除き、クラスターによって生成されたすべてのリソースを破壊します (#782)

セキュリティ：

- `SeccomProfile` を Pod と Containers に追加 (#888)

機能強化：

- `pvcTemplate` でサイズ要求がある場合、クラススペック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) - fenced 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)
- ドキュメント： - モニタリングページを改善し、Prometheus と Grafana を使用してローカルシステムで CloudNativePG の可観測性機能を評価する方法に関する指示を提供します (#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)・to-End Test Suite 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、オペレーターのインストールマニフェストを生成します（#944）
- PostgreSQL 15.0 を新しいデフォルトバージョンとして設定 (#821)

修正：

- plpgsql 関数でデータベースをインポートする (#974)
- ポイントインタイムリカバリを行うときに最も近いバックアップを適切に見つけます (#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)
- フェイルオーバーロジックを、古い可能性のある Kubernetes ポッドの準備ではなく、インスタンスマネージャーからの Postgres 情報に基づいてください (#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。・不足により、バックアップが機能しない可能性がある (#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)

4.48.4 CloudNativePG 1.17 のリリースノート

CloudNativePG の 1.17 マイナーリリースにおけるユーザーに見える変更の履歴。

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で。

バージョン 1.17.5

発売日： 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でリリースされる最後です。

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 プーラーの 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)
- kubectl の cnpg プラグインに psql コマンドを追加（#1668）、ユーザーが pod（デフォルトでプライマリ）と psql セッションを開始できるようにします

技術的な機能強化：

- 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ぐ (#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)

バージョン 1.17.4

リリース日： 2023 年 2 月 14 日

特徴：

- Kubernetes の予測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、クラスターの新しいベース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 上の Webhook に関するファイアウォールの問題を文書化 (#1364)
- recovery の 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 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 互換スペースの例を追加 (#1289)

技術的な機能強化：

- リリースブランチに毎日のエンドツーエンドの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てください (#1380)
- pvcTemplate 使用時に walStorage のリサイズを許可 (#1315)
- Prometheus がインストールされていない場合、PodMonitor reconcile を回避 (#1238)
- 必要ないときに PodMonitor を探さないように (#1213)

バージョン 1.17.3

発売日： 2022 年 12 月 21 日

重要なお知らせ：

- Armando Ruocco（@armru）を一貫してインパクトのある貢献により、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 はサポート終了に達します。

機能強化：

- Istio との互換性を改善：Istio の quit エンドポイントのサポートを追加して、Istio サイドカーを使用したジョブが無期限に実行されないように (#967)
- kubectl-cnpg プラグインの 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)
- ドキュメント - CNPG サンドボックスへの参照を削除 (#1120) - CNPG サンドボックスは非推奨になり、クイックスタートドキュメントでのモニタリングに関する指示を優先しました-“Release updates”ディスカッションへのリンク (#1148) リリースのお知らせと議論のためのデフォルトチャネル - メンテナーの名誉ステータスを GOVERNANCE.md に文書化する (#1033) - メンテナーがコントリビュートを継続する準備ができていない場合にどのように進むべきかを説明- 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・バックアップ指示 (#1184)
- Architecture ドキュメントで Kubernetes レイヤーをより詳細にカバーする (#1432)

修正：

- ブートストラップの PGDATA パーミッションが 750 に適切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ときに、WAL を含む PVC が削除されるように (#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 (#1145)
- 復元時に設定された pg-wal を尊重する (#1216)
- nodeToClusters マップからレプリカを除外 (#1194)

技術的な機能強化：

- テストの概要に `ciclops` を使用します（#1064）：ciclops GitHub アクションに依存して、そのプロジェクトから改善を継承し、E2E スイートの概要を提供します
- バックポートプルリクエストワークフローを追加 (#965) - パッチがそのように注釈されている場合、リリースブランチに自動的にパッチをバックポート
- e2e テストスイートでオペレーターログレベルを設定可能に (#1094)
- ラベルに基づいたテスト実行を有効にする (#951)
- Go バージョンを 1.18 から 1.19 に更新 (#1166)

バージョン 1.17.2

リリース日： 2022 年 11 月 10 日

セキュリティ：

- `SeccomProfile` を Pod と Containers に追加 (#888)

機能強化：

- `cnpg` プラグインの `status` コマンド： - fenced 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)
- ドキュメント： - モニタリングページを改善し、Prometheus と Grafana を使用してローカルシステムで CloudNativePG の可観測性機能を評価する方法に関する指示を提供します (#968)
- PostgreSQL 15.0 を新しいデフォルトバージョンとして設定 (#821)

修正：

- `plpgsql` 関数でデータベースをインポートする (#974)
- ポイントインタイムリカバリを行うときに最も近いバックアップを適切に見つけます (#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)
- フェイルオーバーロジックを、古い可能性のある Kubernetes ポッドの準備ではなく、インスタンスマネージャーからの Postgres 情報に基づいてください (#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。・不足により、バックアップが機能しない可能性がある (#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)

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、オペレーター展開でのリーダー選挙の構成を強化 (#759)
- WAL ファイルを初めて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。正常にアーカイブされました

セキュリティ：

- `nonroot:nonroot` ユーザーを使用して、`distroless/static:nonroot` コンテナイメージにコピーされるインスタンスマネージャーバイナリの権限を明示的に設定する (#754)

修正：

- プライマリに昇格した後、スタンバイ上のアクティブな接続をドロップする (#737)

- ネイティブ Prometheus エクスポーターで MAPPEDMETRIC および DURATION メトリックタイプ変換を尊重する (#765)
- PostgreSQL フォーマットを使用してマイクロ秒精度で指定されたタイムスタンプが正しく解析されることを確認します (#741)

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 ファイル用の分離ボリューム：**ライトアヘッドログ（WAL）とデータベースデータファイルを異なる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により、高書き込みシステムでのパフォーマンスの向上につながる可能性があります。このオプションは、オプションの walStorage セクションを導入して、メインおよび必須の storage セクションで定義された PGDATA とは別の専用ボリュームで WAL ファイル（pg_wal ）を分離することで制御されます（# 513）。現在の制限： walStorage はクラスターの作成時にのみ設定でき、クラスターの稼働中に追加または削除することはできません。

機能強化：

- PgBouncer 接続プーラー実装で低レベルネットワーク TCP 設定の構成を有効にする (#584)
- cnpg.io/instanceName および cnpg.io/podRole ラベルがポッドと PVC に常に存在することを確認してください (#632 および#680)
- インスタンスの role ラベルを基になる PVC に伝播 (#634)
- インスタンスと関連するすべての PVC の削除に役立つ kubectl cnpg destroy コマンドを導入 (#643)

修正：

- 新しいクラスターのプライマリをブートストラップするときに shared_preload_libraries を使用する (#642)
- 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することにより、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るのを防ぎます (#655)
- ハードコードされたファイル名を使用して、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保存しないでください。オペレーターのオンラインアップグレード中に発生する可能性のある競合状態を防止します (#660)
- GetAllAccessibleDatabases 呼び出し時にパニックが発生するのを防止 (#641)

4.48.5 CloudNativePG 1.16 のリリースノート

CloudNativePG の 1.16 マイナーリリースにおけるユーザーに見える変更の履歴。

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で。

バージョン 1.16.5

発売日： 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でリリースされる最後です。

重要なお知らせ：

- Armando Ruocco（@armru）を一貫してインパクトのある貢献により、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 はサポート終了に達します。

機能強化：

- kubectl-cnpg プラグインの 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)
- ドキュメント - CNPG サンドボックスへの参照を削除 (#1120) - CNPG サンドボックスは非推奨になり、クイックスタートドキュメントでのモニタリングに関する指示を優先しました-“Release updates” ディスカッションへのリンク (#1148) リリースのお知らせと議論のためのデフォルトチャネル - メンテナーの名誉ステータスを GOVERNANCE.md に文書化する (#1033) - メンテナーがコントリビュートを継続する準備ができていない場合にどのように進むべきかを説明- 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・バックアップ指示 (#1184)

修正：

- ブートストラップの PGDATA パーミッションが 750 に適切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 (#1145)
- 復元時に設定された pg-wal を尊重する (#1216)
- nodeToClusters マップからレプリカを除外 (#1194)

技術的な機能強化：

- テストの概要に `ciclops` を使用します（#1064）：ciclops GitHub アクションに依存して、そのプロジェクトから改善を継承し、E2E スイートの概要を提供します

- バックポートプルリクエストワークフローを追加 (#965) - パッチがそのように注釈されている場合、リリースブランチに自動的にパッチをバックポート
- e2e テストスイートでオペレーターログレベルを設定可能に (#1094)
- ラベルに基づいたテスト実行を有効にする (#951)
- Go バージョンを 1.18 から 1.19 に更新 (#1166)

バージョン 1.16.4

リリース日： 2022 年 11 月 10 日

セキュリティ：

- `SeccomProfile` を Pod と Containers に追加 (#888)

機能強化：

- `cnpg` プラグインの `status` コマンド： - fenced 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)
- ドキュメント： - モニタリングページを改善し、Prometheus と Grafana を使用してローカルシステムで CloudNativePG の可観測性機能を評価する方法に関する指示を提供します (#968)

修正：

- `plpgsql` 関数でデータベースをインポートする (#974)
- ポイントインタイムリカバリを行うときに最も近いバックアップを適切に見つけます (#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)
- フェイルオーバーロジックを、古い可能性のある Kubernetes ポッドの準備ではなく、インスタンスマネージャーからの Postgres 情報に基づいてください (#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。・不足により、バックアップが機能しない可能性がある (#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)

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、オペレーター展開でのリーダー選挙の構成を強化 (#759)
- WAL ファイルを初めて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。正常にアーカイブされました

セキュリティ：

- nonroot:nonroot ユーザーを使用して、distroless/static:nonroot コンテナイメージにコピーされるインスタンスマネージャーバイナリの権限を明示的に設定する (#754)

修正：

- プライマリに昇格した後、スタンバイ上のアクティブな接続をドロップする (#737)
- ネイティブ Prometheus エクスポーターで MAPPEDMETRIC と DURATION のメトリックタイプ変換を尊重する (#765)

バージョン 1.16.2

リリース日： 2022 年 9 月 6 日

機能強化：

- PgBouncer 接続プーラー実装で低レベルネットワーク TCP 設定の構成を有効にする (#584)
- cnpg.io/instanceName および cnpg.io/podRole ラベルがポッドと PVC に常に存在することを確認してください (#632 および#680)
- インスタンスの role ラベルを基になる PVC に伝播 (#634)

修正：

- 新しいクラスターのプライマリをブートストラップするときに shared_preload_libraries を使用する (#642)
- 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することにより、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るのを防ぎます (#655)
- ハードコードされたファイル名を使用して、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保存しないでください。オペレーターのオンラインアップグレード中に発生する可能性のある競合状態を防止します (#660)
- GetAllAccessibleDatabases 呼び出し時にパニックが発生するのを防止 (#641)

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にする (#456)
- フェイルオーバーまたはスイッチオーバー後、昇格の要求と実際の書き込み可用性の間の経過時間を出力することにより、昇格および降格中のログを強化します (#371)
- クラスターステータスに PostgreSQL クラスターのタイムラインを導入 (#462)
- ジョブ、ポッド、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 クラスターを作成する方法の説明を追加 (#570)

セキュリティ：

- `securityContext` を `Pooler` デプロイメントに明示的に割り当てる (#485)
- Slowloris DDoS を防ぐために、内部 Web サーバーに読み取りタイムアウト値を追加 (#437)

修正：

- リスタート（`stopDelay`）およびスイッチオーバー（`switchoverDelay`）に正しい遅延を使用します。以前に誤ってスワップされたため。`switchoverDelay` が設定されておらず、デフォルト値 40000000 秒を使用している場合、リスタート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、メトリックコレクターがパニックを引き起こさないように (#396)
- チェックされていないポインタを参照していた安全でないデバッグメッセージを削除し、ログレベルに関係なくセグメンテーションフォールトを引き起こす場合がある (#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 の更新を避ける (#501)
- CRD に証明書を挿入する際の競合を処理する (#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` ユーザーを使用します (#411)・`connectionParameters` で `dbname` が指定された場合の作業からのインポート (#569)
- バックアップとリカバリ：- Google Cloud でオブジェクトストアの資格情報を正しく渡すように (#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

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、PostgreSQL 論理バックアップの概念を使用して、新しい CloudNativePG クラスター内の既存の PostgreSQL インスタンス（Kubernetes の外部でも）からネットワークを介して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`（1 つのデータベースのみを新しいクラスターにインポート）と `monolith`（選択したデータベースとロールを既存のインスタンスからインポート）の 2 種類のインポートが導入されました。
- 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、可用性ゾーンなど、プライマリが実行されているノードとは異なる特性のノード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スされたとき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確認することにより、自動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回復するとき、または pg_basebackup を介してインスタンスをクローニングするときに、アプリケーションのデータベース、その所有者、および関連するシークレットの名前の再定義を有効にします（これは、これまでのところ initdb ブートストラップでのみ可能でした）
- バックアップとリカバリ： - PostgreSQL 15 の将来のサポートには Barman >= 3.0.0 が必要 - inheritFromAzureAD オプションを介して Barman Cloud バックアップの Azure AD Workload Identity を有効にする - barmanObjectStore.s3Credentials.region を導入して、バックアップとリカバリの両方のために AWS（AWS_DEFAULT_REGION ）のリージョンを定義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、クラスター初期化フェーズで適切なロギングを保証します-これは、バックアップからのリカバリなどのブートストラップ操作が失敗した場合に特に役立ちます（このパッチの前、こ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）
- ホットスタンバイに敏感なパラメーターが減少し、プライマリが既に再起動している場合の不要なスイッチオーバーを回避します
- ALTER ROLE ステートメントでロール名を適切に引用します
- 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、終了ではなくバックアップの開始と比較していました。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り、「イミディエイト」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します-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回復するときに barmanObjectStore.serverName オプションの無視を停止します
- cnpg プラグイン： - ユーザーが指定したときにプラグインが-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、出力のばらつきを削除します

4.48.6 CloudNativePG 1.15 のリリースノート

CloudNativePG の 1.15 マイナーリリースにおけるユーザーに見える変更の履歴。

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で。

> **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でリリースされる最後です。

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、オペレーター展開でのリーダー選挙の構成を強化 (#759)
- WAL ファイルを初めて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。正常にアーカイブされました

セキュリティ：

- `nonroot:nonroot` ユーザーを使用して、`distroless/static:nonroot` コンテナイメージにコピーされるインスタンスマネージャーバイナリの権限を明示的に設定する (#754)

修正：

- クラスターの条件を `metav1.Conditions` struct と互換性を持たせる (#720)
- プライマリに昇格した後、スタンバイ上のアクティブな接続をドロップする (#737)
- ネイティブ Prometheus エクスポーターで `MAPPEDMETRIC` および `DURATION` メトリックタイプ変換を尊重する (#765)

バージョン 1.15.4

リリース日： 2022 年 9 月 6 日

機能強化：

- PgBouncer 接続プーラー実装で低レベルネットワーク TCP 設定の構成を有効にする (#584)
- `cnpg.io/instanceName` および `cnpg.io/podRole` ラベルがポッドと PVC に常に存在することを確認してください (#632 および#680)
- インスタンスの `role` ラベルを基になる PVC に伝播 (#634)

修正：

- 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することにより、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るのを防ぎます (#655)
- ハードコードされたファイル名を使用して、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保存しないでください。オペレーターのオンラインアップグレード中に発生する可能性のある競合状態を防止します (#660)
- `GetAllAccessibleDatabases` 呼び出し時にパニックが発生するのを防止 (#641)

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にする (#456)
- フェイルオーバーまたはスイッチオーバー後、昇格の要求と実際の書き込み可用性の間の経過時間を出力することにより、昇格および降格中のログを強化します (#371)
- ジョブ、ポッド、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 クラスターを作成する方法の説明を追加 (#570)

セキュリティ：

- `securityContext` を `Pooler` デプロイメントに明示的に割り当てる (#485)
- Slowloris DDoS を防ぐために、内部 Web サーバーに読み取りタイムアウト値を追加 (#437)

修正：

- リスタート（`stopDelay`）およびスイッチオーバー（`switchoverDelay`）に正しい遅延を使用します。以前に誤ってスワップされたため。`switchoverDelay` が設定されておらず、デフォルト値 40000000 秒を使用している場合、リスタート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、メトリックコレクターがパニックを引き起こさないように (#396)
- チェックされていないポインタを参照していた安全でないデバッグメッセージを削除し、ログレベルに関係なくセグメンテーションフォールトを引き起こす場合がある (#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 の更新を避ける (#501)
- CRD に証明書を挿入する際の競合を処理する (#547)
- バックアップとリカバリ：- Google Cloud でオブジェクトストアの資格情報を正しく渡すように (#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

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の PostgreSQL 15 の将来のサポートには Barman >= 3.0.0 が必要

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、クラスター初期化フェーズで適切なロギングを保証します-これは、バックアップからのリカバリなどのブートストラップ操作が失敗した場合に特に役立ちます（このパッチの前、こ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）
- ホットスタンバイに敏感なパラメーターが減少し、プライマリが既に再起動している場合の不要なスイッチオーバーを回避します
- `ALTER ROLE` ステートメントでロール名を適切に引用します
- 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、終了ではなくバックアップの開始と比較していました。`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り、「イミディエイト」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します-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回復するときに `barmanObjectStore.serverName` オプションの無視を停止します
- cnpg プラグイン： - ユーザーが指定したときにプラグインが-`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、出力のばらつきを削除します

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した 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に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が所有するオブジェクト）、および `cluster`（Postgres クラスターが所有）オブジェクト）
- ネイティブ Prometheus エクスポーターでの 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することにより、フェンシングを修正
- WithActiveInstance ファンクション内の 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ちます
- 空でないバックアップオブジェクトストアで新しいクラスターがブートストラップされるのを防ぎ、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、リカバリ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、調整ループを再キューイングします
- 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します
- 複数のレプリカを使用して、高可用性でオペレーターを展開する場合、オペレーターポッドが常に最新の証明書を持っていることを保証します
- 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展開する場合を正しく処理します
- 再起動を必要とせずに、クラスターの inheritedMetadata ラベルと注釈のセットの変更をクラスターの関連リソースに適切に伝播します
- 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、オペレーターデプロイメントで定義されたカスタム構成マップ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

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たクラスターまたは特定の PostgreSQL インスタンスのセットのフェンシング機能の導入。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。フェンスで囲まれたインスタンスはシャットダウンされ、ポッドは検査と緊急時に実行され続けます（準備ができていないと見なされます）
- **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の Kubernetes シークレットのオプションの使用を有効にします
- primaryUpdateMethod オプションの導入、switchover（デフォルト）および restart の値を受け入れ、教師なし primaryUpdateStrategy の場合に使用されます。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
- 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のための kubectl cnp プラグインの新しい report コマンド

- バックアップオブジェクトストアになくなった Backup オブジェクトを整理します
- Point-In-Time Recovery ブートストラップ方法でのターゲットタイムラインと LSN の指定
- sessionToken オプションを介した AWS S3 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常に動作するようになりました
- オペレーターが pg_controldata から収集する値に依存することにより、ホットスタンバイに敏感なパラメーターの回復力を向上させました
- 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がロールバックされる代わりにコミットされるようになりました（これにより、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。

4.48.7 1.14.0 以前のリリースノート

CloudNativePG の最初のパブリックリリースはバージョン 1.15.0 です。それ以前は、製品は EDB が完全に所有し、
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、コミットに加えて製品の履歴をデモンストレーションします。**ここにリストされているバージョンはいずれも CloudNativePG には存在しません。**

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活用して、バックアップとリカバリのために 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依存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認証情報を提供する代替方法として AWS IAM ロールを継承する機能
- Pooler の authQuerySecret と証明書の Opaque secrets のサポート
- デフォルトの PostgreSQL バージョンを 14.2 に更新しました
- 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指定のオペレーター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により、インスタンスマネージャーが中断された pg_rewind をリトライできるようにします
- pg_rewind を実行する前に古い PID ファイルをクリーンアップ
- primary_conninfo のキーによるソートを強制して、13 より前の PostgreSQL バージョンでのランダムリスタートを回避
- 調整時に ServiceAccount の変更（ラベル、注釈など）を保持します
- imagePullPolicy のデフォルト値の施行を無効にします
- WAL セグメントサイズの initDB 検証を改善
- LSN を指定したクラスターを回復するときに targetLSN オプションを適切に処理する
- サーバーと 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

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を高速化するバックアップ用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。この機能を使用するには、オペランド画像に Barman 2.18 が必要です。クラスター削除後のバックアップの
- `status.conditions` を使用したクラスターのステータスの拡張。`ContinuousArchiving` 条件は、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 status コマンドを追加情報で改善します。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時間を含む `Certificates Status` セクションを追加します。
- 新しいクラスターを作成するとき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モニタリング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の権限を制限できるようにします（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、プラグインが Windows x86 および ARM で利用できるようになりました
- Kubernetes 1.18 および非推奨の API バージョンのサポートを削除

コンテナイメージ：

- PostgreSQL コンテナには Barman 2.18 が含まれます

セキュリティ修正：

- 所有者とスーパーユーザーの秘密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 PostgreSQL ユーザーのパスワードを変更できました

修正：

- Postgres ポッドからステータスを取得するコードでのメモリリークを修正
- クラッシュ後の PostgreSQL のセルフリスタートを無効にします。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 の `spec.postgresUID` および `spec.postgresGID` フィールドの変更を防止します。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 を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 生産率を持つクラスター内のスタンバイサーバーが、調整されたらストリーミングに切り替えることができます
- まれにクラッシュにつながる可能性のある PostgreSQL Pod 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のある競合状態を修正
- レプリカとしてクラスターに参加する前に、降格したばかりのプライマリポッドの余分なリスタートを削除します

- バックアップから回復するときにレプリケーションに敏感な PostgreSQL 構成パラメーターを正しく処理します
- クラスター作成中の PostgreSQL 構成の欠落している検証を修正

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により、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向上させます。この変更により、オペレーターがポッドが失敗と宣言する前にポッドがそのステータスを報告するのを待機する時間に制限が設定され、フェイルオーバーオペレーションの堅牢性と予測可能性が強化されます
- `.spec.inheritedMetadata` フィールドをクラスターに導入して、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 プローブを追加します
- 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認証局のサポートを追加します

修正：

- フェールオーバー中、すべての WAL ストリーミング接続が閉じられるまで、新しいプライマリを選択するのを待ちます。オペレーターは、デフォルトで `wal_sender_timeout` と `wal_receiver_timeout` を 5 秒に設定して、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
- レプリカクラスター内の PostgreSQL インスタンスの「archive_mode」を「always」に設定することにより、ローリング更新を修正するようにレプリカクラスターの WAL アーカイブ戦略を変更します。次に、現在およびターゲットの指定されたプライマリからのみ WAL のアップロードを制限します。スイッチオーバー中に WAL が 2 回アップロードされる場合がありますが、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認証局のサポートを修正
- クラスター名前空間のデフォルトのモニタリング構成マップに固定名を使用します
- デフォルトの Webhook が何らかの理由で機能していない場合、オペレーターは調整サイクル中にもデフォルトでクラスターを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
- すべての調整ループの開始時に「デフォルト」メッセージのログ記録を停止します
- 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します
- ポッドに到達できない場合、アーキテクチャの不一致の可能性について不平を言わないでください

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）および WAL 復元（restore_command による）。パラレル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シーケンシャルではなくパラレル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
- **モニタリング用のメトリックのデフォルトセット：**default-monitoring と呼ばれる新しい ConfigMap は、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によりグローバルに、または.spec.monitoring.disableDefaultQueries オプションを介してクラスターレベルで変更できます（デフォルトでは false に設定されます）
- クラスターのモニタリングセクションに 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
- ポッドが 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しようとすることにより、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グレースフルにシャットダウンし、新しいプライマリを昇格する前にすべての WAL ファイルをアーカイブします（デフォルトでは、CloudNativePG は特権データの耐久性を無期限に待機します）
- ローリング更新を発行することにより、PostgreSQL Cluster のリソース要求と制限の変更を処理します
- kubectl の cnp プラグインの status コマンドを改善します。追加情報：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ロール
- PostgreSQL の共有メモリ管理用に dynamic_shared_memory_type および shared_memory_type パラメーターの構成を有効にすることにより、多数の並列ワーカーを含む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アノテーションを、ポッドを含む関連するリソースに伝播します（削除はサポートされていません）
- Kubelet によって追い出されたポッド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介して Azure の永続ボリュームの自動サイズ変更を管理します（デフォルトでは無効）
- 特定の Postgres クラスターで disabled に設定されている場合、調整ループの実行を防止する cnpg.io/reconciliationLoop アノテーションを導入します
- 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する SQL クエリのリストを指定します

修正：

- Liveness probe 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、数人のお客様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に影響を与えます WAL ファイルを回復して一貫した状態に到達する必要がありますが、liveness プローブのタイムアウトが始まる前にそれを行うことができず、ポッドが CrashLoopBackOff のままになりますステータス。
- Liveness probe は、タイムラインの変更後に 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が修正されます。
- Postgres ポッドからのクライアント側のスロットルを削減します（例：Waited for 1.182388649s due to client-side throttling, not priority and fairness, request: GET ）
- 提供されたものを使用して、OpenShift および OLM インストールでの Public Key Infrastructure（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ーを変更する場合、常にプライマリを最後のままにします
- 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 初期化での競合状態を防止
- 期限切れの 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ます。
- 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 ファイルを削除します
- Pooler リソースにオペレーターで定義された imagePullSecret を継承させます（存在する場合）

バージョン 1.10.0

リリース日： 2021 年 11 月 11 日

特徴：

- **PgBouncer を使用した接続プーリング** ：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 形式でのロギング、統計、プール、リストの 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（例：30 日のリカバリウィンドウを保証するための「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に続いて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` 配列は廃止され、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デフォルトオブジェクトを含めることができます
- インスタンスに追加された新しいデフォルトのメトリック Prometheus エクスポーター：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従った回復可能性の最初のポイント
- 失敗した後にバックアップを取ることを再試行します
- オペレーターが、頻繁に更新されないオペランドイメージに存在しない、最近導入された機能（保持ポリシー、Azure Blob Container ストレージなど）を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高めます
- `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します
- PVC のステータス（初期化中または準備完了）を報告する新しいアノテーションを導入します
- `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

セキュリティ：

- オペレーターコンテナ securityContext の allowPrivilegeEscalation を `false` に設定します

修正：

- シングルインスタンスクラスターのメンテナンス中にプライマリ PodDisruptionBudget を無効にします
- 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 アーカイブステータスをチェックする際の Postgres 接続リークを防ぎます
- ログをあふれさせる一時的なエラーの後、WAL のアーカイブ/復元を 100 ミリ秒間スリープさせます

バージョン 1.9.2

リリース日： 2021 年 10 月 15 日

特徴：

- 2 つの新しいロガーで JSON ログを強化します。PostgreSQL の `archive_command` の場合は `wal-archive`、スタンバイの `restore_command` の場合は `wal-restore`

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した）
- 古いオペランドでのエラーを回避するために 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できるのを待ちます

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 の発売を祝うためのものです

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、それを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、バックアップが正常に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に問題が発生した場合は、新しいバックアップを要求して適切に再試行します

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* の後続のリリースでの遅延のため、*OpenShift* で利用できません。

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` の`--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介して、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 を呼び出すときに postgres ユーザーを明示的に設定します - OpenShift の制限付き SCC で必要
- status の FirstRecoverabilityPoint を適切に更新します
- バックアップが要求された場合、指定されたプライマリで archive_mode = always を設定します
- マイナーなバグ修正

バージョン 1.8.0

リリース日： 2021 年 9 月 13 日

特徴：

- 外部クラスター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ーをブートストラップし、バックアップ CR を定義する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、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ときにクラスターの先頭に巻き戻す機能を追加します
- クラスターステータスに firstRecoverabilityPoint を追加して、最も古い一貫した時点を報告し、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します
- 次の新しいメトリックを公開することにより、PostgreSQL インスタンスのデフォルトの Prometheus エクスポーターを拡張します。

1. WAL ファイルの数とディスク上の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/最大同期レプリカ、ならびに予想および実際に観察された数もの

- Prometheus エクスポーター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、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 2.13 for Barman Cloud で導入された機能を利用して、バックアップとリカバリのために Azure Blob Storage をネイティブにサポート
- liveness probe は pg_isready に依存します
- リカバリ対象時刻のタイムスタンプ指定に RFC3339 フォーマットをサポート
- .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のプルポリシーを制御します
- OpenShift 4.5 に代わる OpenShift 4.8 のサポートを追加します
- PostgreSQL 14（ベータ版）のサポート

- 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のクロスクラスターレプリケーションで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：

- パスワード認証を介して `postgres` ユーザーとのネットワークアクセスを有効/無効にする `.spec.enableSuperuserAccess` を導入

修正：

- クラスターが回復不能な状態になり、人間の介入が必要な場合にユーザーに適切に通知します

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は構成よりも自己修復を優先し、クラスターが自己修復操作中に要件を満たすことができない場合に備えて、オペレーターは `minSyncReplicas` および `maxSyncReplicas` 設定を一時的にオーバーライドできます
- `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ときにオペレーターがフェー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します
- PGAudit の複数行のログエントリを正しく解析します

バージョン 1.7.0

リリース日： 2021 年 7 月 28 日

特徴：

- JSON 出力で直接利用可能な `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、モニタリングおよび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 - 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するための Prometheus エクスポーターの `target_databases` オプション（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` メトリックの露出 `primaryUpdateStrategy` が `supervised` に設定されているクラスターに手動スイッチオーバーが必要かどうかを迅速に報告する

- pg_audit 、auto_explain 、および 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 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。
- kubectl の cnp プラグインに reload コマンドを追加して、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 を介した Pod アフィニティおよびアンチアフィニティ構成の制御を改善します
- 少なくともプライマリインスタンスをいつでも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個の PodDisruptionBudget を導入します

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、カスタムクライアント認証局と PostgreSQL サーバーの証明書を定義し、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認証局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
- クラスターのオペレーターによって作成されたサービスを調整します

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。レプリカクラスターには、**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 パラメーターを追加して、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 パラメーターを追加します。preferred（デフォルト）に設定して preferredDuringSchedulingIgnoredDuringExecution が使用されるか、required に設定して requiredDuringSchedulingIgnoredDuringExecution に設定できます。

変更点：

- オペレーターが新しいポッドを作成できない、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の変更時に 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。
- アノテーション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。

バージョン 1.5.1

リリース日： 2021 年 6 月 17 日

変更：

- Red Hat OpenShift の Operator Deployments 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。

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持つ Kubernetes ノードに PostgreSQL インスタンスを配布できるようにします
- より決定的で反復可能な展開のために、<image>:<tag>@sha256:<digestValue> 形式を介したイメージ名へのダイジェストの指定を有効にします

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する認証局のシークレットを定義することにより、PostgreSQL サーバーを認証する TLS 証明書をカスタマイズします
- require から verify-ca への内部および自動管理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（サービス名が継承される）に DNS-1035 検証スキームを採用
- 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

- 物理ベースバックアップに関して Backup リソースと beginWal 、endWal 、beginLSN 、endLSN 、startedAt 、stoppedAt を統合します
- ドキュメントの改善： - オペレーターとオペランドコンテナによって公開されるポートのリストを提供する- データベースワークロードのストレージと PostgreSQL のベンチマークのための cnp-bench ヘルムチャートとガイドラインを導入する
- E2E テストの機能強化： - テスト Kubernetes 1.21 - 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
- マイナーなバグ修正、以下を含む： - リカバリ操作中の pg_ctl start へ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ない - barmanObjectStore バックアップの immediateCheckpoint および jobs パラメータの処理を修正 - バックアップから回復するときの空のログ

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 エクスポーターに PostgreSQL メトリックの基本セットを提供します
- 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：

- Pod 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：**YAML マニフェストを使用して CloudNativePG オペレーターを以前に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 との競合を回避するために必要です。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
- オペレーターが自動的に定義するラベルを修正し、名前を control-plane: controller-manager から app.kubernetes.io/name: cloudnative-pg に変更します
- Prometheus エクスポーターの TCP ポートに metrics 名前を割り当てます
- PostgreSQL のメトリックエクスポーター接続への application_name として cnp_metrics_exporter を設定します
- 利用可能な場合、 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の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： - モニタリング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 テストの機能強化

- マイナーなバグ修正、以下を含む：- `pg_basebackup` を呼び出すときに`-R` を使用しないでください

バージョン 1.3.0

リリース日： 2021 年 4 月 23 日

特徴：

- ラベルとアノテーションの継承
- 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：

- Pod 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、Red Hat OpenShift の制限付きセキュリティコンテキスト制約のサポート
- 非ルート、非特権、特権エスカレーションなしで実行するようにオペレーターによって明示的に定義されたポッドセキュリティコンテキスト

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に予約されました）
- 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

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)

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のカスタムモニタリングクエリの実験的サポートを導入
- Operator Lifecycle Manager（OLM）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）のガイドラインを適用し、Doccle を使用したパイプラインで直接検証することにより、コンテナのセキュリティを強化します

- AKS の E2E テストを改善する
- マイナーなバグ修正

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、使用可能なホットスタンバイレプリカのみに接続をルーティングし、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

バージョン 1.0.0

リリース日： 2021 年 2 月 4 日

CloudNativePG の最初のメジャー安定版リリースは、API グループ `postgresql.cnpg.io/v1` で `Cluster`、`Backup`、および `ScheduledBackup` を実装します。これらのリソースを使用して、次の主な機能を備えた Kubernetes 内に 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せずに、高可用性のための Kubernetes API サーバーとの直接統合
- 自己修復機能、以下を介したもの： - 最も調整されたレプリカを昇格す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スイッチオーバー
- 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**読み取り/書き込み**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**読み取り**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 Persistent Volumes ストレージの再利用
- PostgreSQL マイナーバージョンとオペレーター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完全復旧とポイントインタイム復旧
- 同期レプリカのサポート
- `nodeSelector` プロパティを介した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