



EDB

CloudNativePG 1.16

リリース *1.16.0*

EnterpriseDB Corporation

2022 年 08 月 12 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	10
4.1.3	クラウド用語	11
4.1.4	次に行うこと	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	読み書きワークロード	15
4.3.2	読み取り専用ワークロード	16
4.3.3	マルチクラスター展開	16
4.4	インストールとアップグレード	18
4.4.1	Kubernetes へのインストール	18
4.4.2	展開に関する詳細	18
4.4.3	アップグレード	19
4.5	クイックスタート	21
4.5.1	パート 1-ローカル Kubernetes プレイグラウンドのセットアップ	21
4.5.2	パート 2-CloudNativePG のインストール	22
4.5.3	パート 3-PostgreSQL クラスターをデプロイする	22
4.6	ブートストラップ	23
4.6.1	bootstrap セクション	24
4.6.2	externalClusters セクション	24
4.6.3	空のクラスターをブートストラップする (initdb)	25
4.6.4	別のクラスターからのブートストラップ	29
4.7	Postgres データベースのインポート	42
4.7.1	仕組み	43
4.7.2	microservice タイプ	43
4.7.3	monolith タイプ	46
4.8	セキュリティ	48
4.8.1	コード	48
4.8.2	コンテナ	48
4.8.3	クラスター	49

4.9	Postgres インスタスマネージャー	52
4.9.1	Liveness と readiness プロブ	53
4.9.2	シャットダウン制御	53
4.9.3	フェールオーバー	54
4.10	スケジューリング	54
4.10.1	Pod のアフィニティとアンチアフィニティ	54
4.10.2	nodeSelector によるノード選択	56
4.10.3	寛容	56
4.11	リソース管理	57
4.12	障害モード	58
4.12.1	ストレージ使用量	59
4.12.2	障害モード	59
4.12.3	自己修復	61
4.12.4	手動介入	61
4.13	ローリングアップデート	62
4.13.1	自動更新 (unsupervised)	62
4.13.2	手動更新 (supervised)	63
4.14	レプリケーション	63
4.14.1	アプリケーションレベルのレプリケーション	64
4.14.2	非常に成熟したテクノロジー	64
4.14.3	ストリーミングレプリケーションのサポート	64
4.14.4	同期レプリケーション	65
4.15	バックアップとリカバリー	67
4.15.1	クラウドプロバイダーのサポート	67
4.15.2	オンデマンドバックアップ	75
4.15.3	スケジュールされたバックアップ	76
4.15.4	WAL アーカイブ	77
4.15.5	リカバリー	78
4.15.6	保持ポリシー	80
4.15.7	圧縮アルゴリズム	81
4.15.8	バックアップオブジェクトのタグ付け	81
4.16	PostgreSQL 設定	82
4.16.1	postgresql セクション	82
4.16.2	pg_hba セクション	87
4.16.3	構成の変更	89
4.16.4	動的共有メモリの設定	89
4.16.5	固定パラメーター	90
4.17	オペレーター設定	93
4.17.1	利用可能なオプション	93
4.17.2	オペレーター構成マップの定義	94
4.17.3	オペレーターシークレットの定義	94
4.17.4	オペレーターを再起動して構成をリロードする	95
4.17.5	PPROF HTTP サーバー	95
4.18	ストレージ	96
4.18.1	CloudNativePG のベンチマーク	97
4.18.2	永続ボリュームの要求	97
4.18.3	ストレージクラスを介した設定	98

4.18.4	PVC テンプレートを介した設定	98
4.18.5	ボリューム拡張	99
4.19	ラベルとアノテーション	102
4.19.1	前提条件	103
4.19.2	クラスターのメタデータの定義	103
4.19.3	現在の制限	104
4.20	モニタリング	104
4.20.1	インスタンスのモニタリング	104
4.20.2	オペレーターの監視	115
4.21	ロギング	116
4.21.1	オペレーターログ	116
4.21.2	PostgreSQL ログ	117
4.21.3	PGAudit ログ	118
4.21.4	その他のログ	119
4.22	証明書	120
4.22.1	オペレーター管理モード	121
4.22.2	ユーザー提供の証明書モード	122
4.23	クライアント TLS / SSL 接続	126
4.23.1	新しい証明書の発行	127
4.23.2	TLS 証明書を介した接続のテスト	127
4.24	アプリケーションから接続する	130
4.24.1	DNS 解決	130
4.24.2	環境変数	131
4.24.3	秘密	131
4.25	接続プーリング	131
4.25.1	アーキテクチャ	132
4.25.2	クイックスタート	132
4.25.3	プーラーリソースのライフサイクル	133
4.25.4	セキュリティ	134
4.25.5	認証	134
4.25.6	ポッドテンプレート	135
4.25.7	高可用性 (HA)	137
4.25.8	PgBouncer の構成オプション	137
4.25.9	モニタリング	139
4.25.10	ログ	144
4.25.11	接続の一時停止	144
4.25.12	制限	144
4.26	レプリカクラスター	145
4.26.1	基本概念	145
4.26.2	レプリカクラスターのセットアップ	146
4.26.3	レプリカクラスターで指定されたプライマリを昇格	148
4.27	Kubernetes のアップグレード	149
4.27.1	reusePVC が false に設定されたシングルインスタンスクラスター	150
4.28	Postgres サービスの公開	151
4.28.1	Minikube でのテスト	153
4.29	CloudNativePG プラグイン	153
4.29.1	インストールする	154

4.29.2	使用する	154
4.30	自動フェイルオーバー	164
4.30.1	RTO と RPO への影響	165
4.31	トラブルシューティング	166
4.31.1	始める前に	166
4.31.2	オペレーター情報	167
4.31.3	クラスター情報	169
4.31.4	ポッド情報	170
4.31.5	バックアップ情報	172
4.31.6	ストレージ情報	172
4.31.7	ノード情報	173
4.31.8	条件	173
4.31.9	いくつかの一般的な問題	174
4.32	フェンシング	175
4.32.1	インスタンスをフェンスする方法	176
4.32.2	フェンシングを持ち上げる方法	177
4.32.3	フェンシングの仕組み	177
4.33	エンドツーエンドのテスト	178
4.34	コンテナイメージの要件	179
4.34.1	イメージタグの要件	180
4.35	オペレーターの能力レベル	180
4.35.1	レベル 1 - 基本インストール	181
4.35.2	レベル 2 - シームレスなアップグレード	186
4.35.3	レベル 3 - フルライフサイクル	187
4.35.4	レベル 4 - 深い洞察	190
4.35.5	レベル 5 - オートパイロット	192
4.36	例	192
4.37	商用サポート	204
4.38	よくある質問 (FAQ)	205
4.38.1	Kubernetes で PostgreSQL を実行する	205
4.38.2	データベース管理	207
4.39	API リファレンス	209
4.39.1	AffinityConfiguration	211
4.39.2	AzureCredentials	212
4.39.3	バックアップ	213
4.39.4	バックアップ構成	213
4.39.5	バックアップリスト	213
4.39.6	バックアップソース	214
4.39.7	BackupSpec	214
4.39.8	バックアップステータス	214
4.39.9	BarmanCredentials	215
4.39.10	BarmanObjectStoreConfiguration	215
4.39.11	Bootstrap 構成	216
4.39.12	BootstrapInitDB	217
4.39.13	BootstrapPgBaseBackup	218
4.39.14	BootstrapRecovery	218
4.39.15	証明書構成	219

4.39.16	CertificatesStatus	220
4.39.17	クラスター	220
4.39.18	ClusterList	221
4.39.19	ClusterSpec	221
4.39.20	ClusterStatus	223
4.39.21	ConfigMapKeySelector	225
4.39.22	ConfigMapResourceVersion	225
4.39.23	DataBackupConfiguration	225
4.39.24	EmbeddedObject メタデータ	226
4.39.25	外部クラスター	226
4.39.26	Google 資格情報	226
4.39.27	インポート	227
4.39.28	ImportSource	227
4.39.29	インスタンス ID	227
4.39.30	InstanceReportedState	227
4.39.31	LDAPBindAsAuth	228
4.39.32	LDAPBindSearchAuth	228
4.39.33	LDAPConfig	228
4.39.34	LocalObjectReference	229
4.39.35	MonitoringConfiguration	229
4.39.36	NodeMaintenanceWindow	229
4.39.37	PgBouncerIntegrationStatus	230
4.39.38	PgBouncer の秘密	230
4.39.39	PgBouncerSpec	230
4.39.40	ポッドメタ	231
4.39.41	PodTemplateSpec	231
4.39.42	プーラー	232
4.39.43	プーラー統合	232
4.39.44	PoolerList	232
4.39.45	PoolerSecrets	232
4.39.46	PoolerSpec	233
4.39.47	PoolerStatus	233
4.39.48	PostInitApplicationSQLRefs	233
4.39.49	PostgresConfiguration	234
4.39.50	リカバリーターゲット	234
4.39.51	ReplicaClusterConfiguration	235
4.39.52	RollingUpdateStatus	235
4.39.53	S3 資格情報	235
4.39.54	スケジュールされたバックアップ	236
4.39.55	ScheduledBackupList	236
4.39.56	ScheduledBackupSpec	236
4.39.57	ScheduledBackupStatus	237
4.39.58	SecretKey セレクター	237
4.39.59	シークレットバージョン	238
4.39.60	SecretsResource バージョン	238
4.39.61	StorageConfiguration	239
4.39.62	SyncReplicaElectionConstraints	239

4.39.63	トポロジー	240
4.39.64	WalBackup 構成	240
4.40	サポートされているリリース	241
4.40.1	サポートポリシー	241
4.40.2	ネーミングスキーム	242
4.40.3	CloudNativePG リリースのサポート状況	243
4.40.4	今後のリリース	243
4.40.5	サポートとは	243
4.41	リリースノート	244
4.41.1	CloudNativePG 1.15 のリリースノート	244
4.41.2	CloudNativePG 1.16 のリリースノート	248
4.41.3	1.14.0 以前のリリースノート	250

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

これは、高可用性と読み取り専用クエリのオフロードのために、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、TCP を介して Postgres を公開するように Service または Ingress オブジェクトを構成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、次に 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 - 自動パイロット」**セットを期待できます。

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 には、Kubernetes 1.22 以降が必要です。詳細については、[サポートされているリリース ページ](#)を参照してください。

第2章 コンテナイメージ

CloudNativePG community は、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 オペレーターコンテナイメージは **distroless** であり、`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ーで利用できます。

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
- 自己修復機能：* 最も調整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
- 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

アプリケーションを読み取りワークロードのインスタンスに接続するための読み取り専用*サービスの定義

- クラスター spec 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：pg_audit、auto_explain、および pg_stat_statements
- PVC テンプレートを使用したローカル永続ボリュームのサポート
- Pod での永続ボリュームストレージの再利用
- PostgreSQL マイナーバージョンのローリングアップデート

*オペレーターアップグレード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）
- バックアップ保持ポリシー（リカバリーウィンドウに基づく）

*オブジェクトストア内の既存のバックアップからの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
- 並列 WAL アーカイブと復元により、データベースが書き込み量の多いシステムで WAL 生成についていくことができます
- 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します

- 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
- 同期レプリカのサポート
- 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
- PostgreSQL クラスター全体、またはインスタンスのサブセットのフェンシング

*シンプルバインドと検索+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

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 コミュニティ協会の商標または登録商標であり、彼らの許可を得て使用しています。

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

Resource	Description
<i>NodeMaintenanceWindow</i>	*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。
ポッドメタ	*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ます。
:ref:`microservice` タイプ <microservice タイプ>	*サービス*は、ポッドのグループで実行され、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アプリケーションをネットワークサービスとして公開する抽象化です。
<i>PgBouncer</i> の秘密	*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設定	*ストレージクラス*を使用すると、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。

[次のページに続く](#)

表 1 – 前のページからの続き

Resource	Description
ポッド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 <i>Kubernetes</i> 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)` < https://kubernetes.io/docs/concepts/storage/volumes/#local >`__`は、それを使用するポッドが実行されている特定のノードにのみ存在する永続ボリュームです。
永続ボリュームの要求	永続ボリューム要求 (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ます。ポッドがノードの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。
Namespace	*名前空間*は、 <i>Kubernetes</i> 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許可されたユーザーのみに制限します。 <i>Kubernetes</i> には、名前空間とクラスターレベルでロールを制御し、特定のリソースと個人に関連付けるためのネイティブ API があります。
CRD 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 <i>Kubernetes</i> 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が通常実行す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ように <i>Kubernetes</i> を支援します。
kubectl	`kubectl` は、 <i>Kubernetes</i> 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、*Kubernetes* 1.19 以降が必要です。

4.1.2 PostgreSQL 用語

Resource	Description
Instance	「IP アドレス」と「TCP ポート」（通常は 5432）のペアで実行およびリスンする Postgres サーバプロセス。
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。

次のページに続く

表 2 – 前のページからの続き

Resource	Description
Replica	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し、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る PostgreSQL インスタンス。レプリカは、 *スタンバイ* または *セカンダリ* サーバーとも呼ばれます。PostgreSQL は、物理ストリーミングレプリケーション (非同期/同期) とファイルベースのログ SHIPPING (非同期) に依存しています。
Hot Standby	*レプリカ* が読み取り専用ワークロードを受け入れられるようにする PostgreSQL の機能。
Cluster	高可用性 (HA) クラスターとして使用：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。
Replica Cluster	選択した PostgreSQL クラスターからの継続的リカバリモードである CloudNativePG <i>Cluster</i> 。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有効にする機能です。
Designated Primary	別の PostgreSQL クラスターからの継続的復旧中であり、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。
Superuser	PostgreSQL では、 *スーパーユーザー* は <code>LOGIN</code> と <code>'SUPERUSER'</code> の両方の特権を持つロールです。セキュリティ上の理由から、CloudNativePG は、ローカルの Unix ドメインソケットを介した <code>'peer'</code> 認証を介して <code>'postgres'</code> ユーザーとして <code>'postgres'</code> 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。

4.1.3 クラウド用語

Resource	Description
Region	クラウドの *リージョン* は、 *アベイラビリティゾーン* で編成された分離された独立した地理的エリアです。リージョン内のゾーンでは、ラウンドトリップのネットワークレイテンシーはほとんどありません。
Zone	クラウド (*ゾーン* とも呼ばれます) の *アベイラビリティゾーン* は、リソースをデプロイできるリージョンです。通常、アベイラビリティゾーンは、データセンターまたは同じデータセンターの分離された建物に対応します。

4.1.4 次に行うこと

用語に慣れたので、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` することができます。

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で動作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。

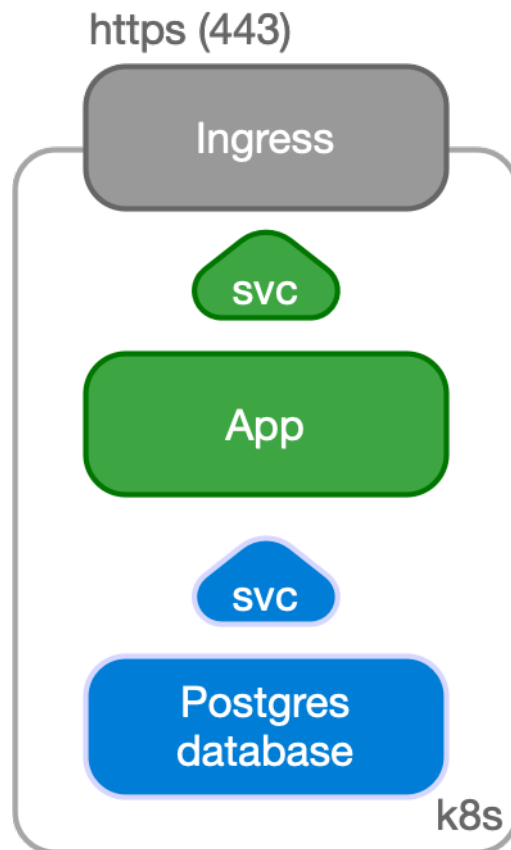


図 1: Application and Database inside Kubernetes

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。

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を介して 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、信頼できる永続的な方法で状態を追跡します。アプリケーションは、TLS 接続を介して、現在のプライマリインスタンスをポイントする、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のスタンバイアーキテクチャのロジックが組み込まれ、Postgres での高可用性クラスターの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

別のユースケースは、アプリケーションを外部（仮想化環境など）にしながら、PostgreSQL データベースを Kubernetes 内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 接続を引き続き活用できます。

4.3 アーキテクチャ

高可用性とスケーラビリティの目標のために、PostgreSQL データベース管理システムは、管理者に **Write-AheadLog (WAL) シッピング** に基づいた組み込みの **物理レプリケーション** 機能を提供します。

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

CloudNativePG は、次の仕様で、同じ Kubernetes クラスター内で複数のホットスタンバイレプリカを管理するための非同期および同期ストリーミングレプリケーションに基づくクラスターをサポートしています。

- 1 つのプライマリ、オプションで高可用性のため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唯一のプライマリインスタンスに接続
 - * **-ro** : アプリケーションは読み取り専用ワークロードの唯一のホットスタンバイレプリカに接続
 - * **-r** : アプリケーションは読み取り専用のいずれかのインスタンスに接続ワークロード
- 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
 - * PostgreSQL インスタンスは同じリージョン内の異なるアベイラビリティゾーンに存在できる同じ地域に住んでいる

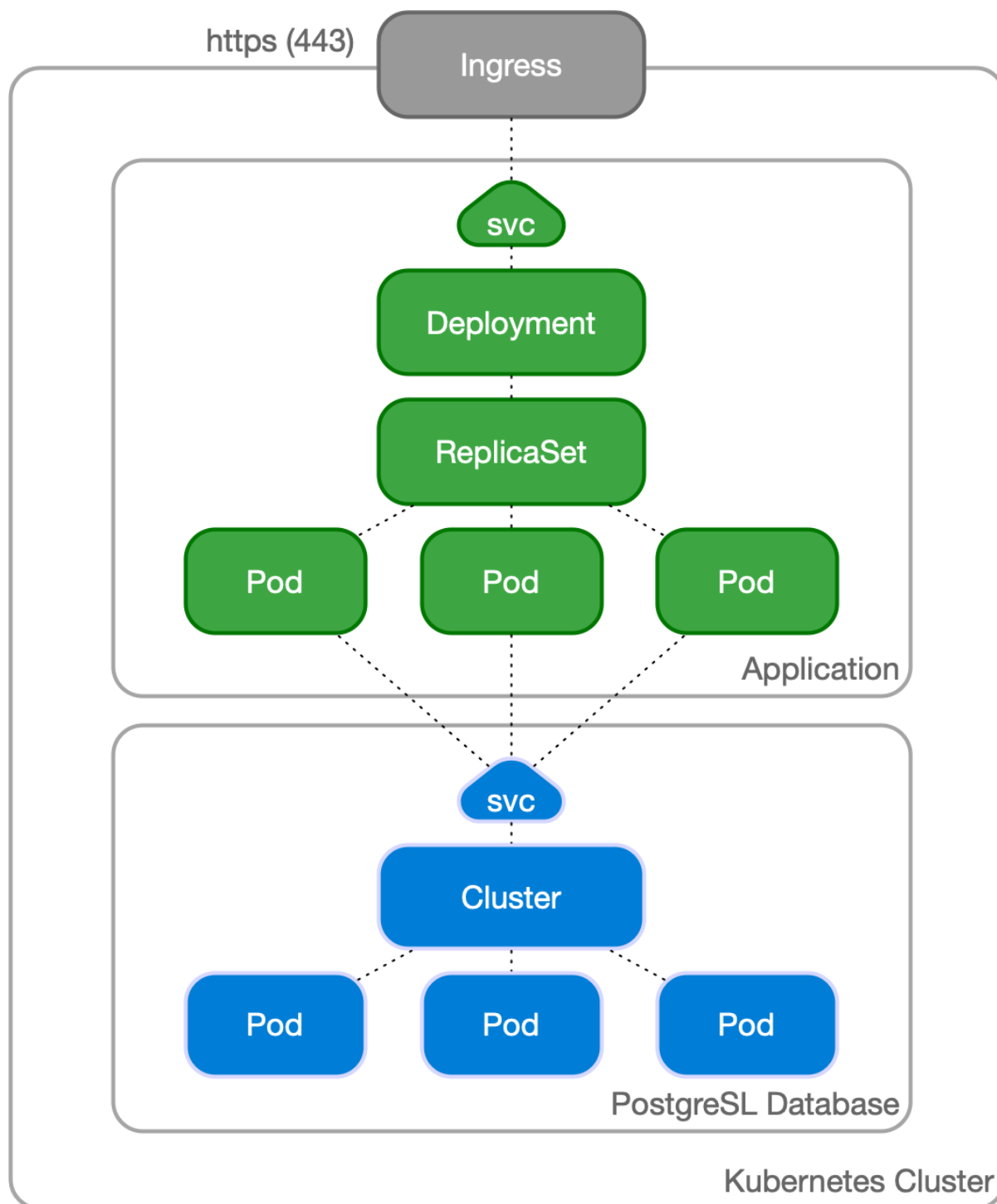


図 2: Close-up view of application and database inside Kubernetes

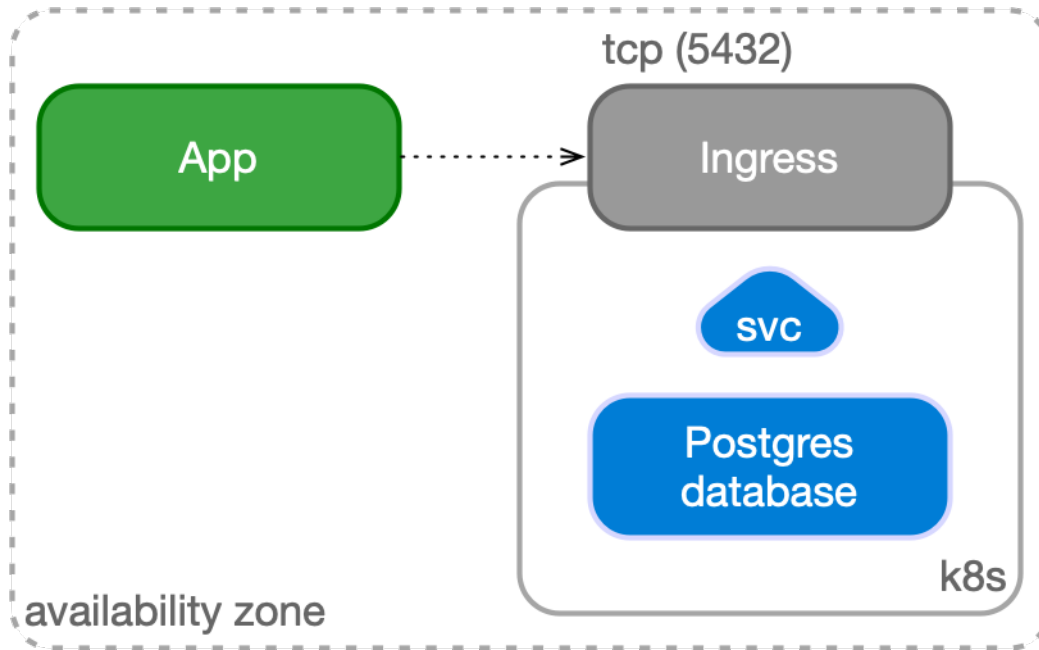


図 3: Application outside Kubernetes

4.3.1 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。

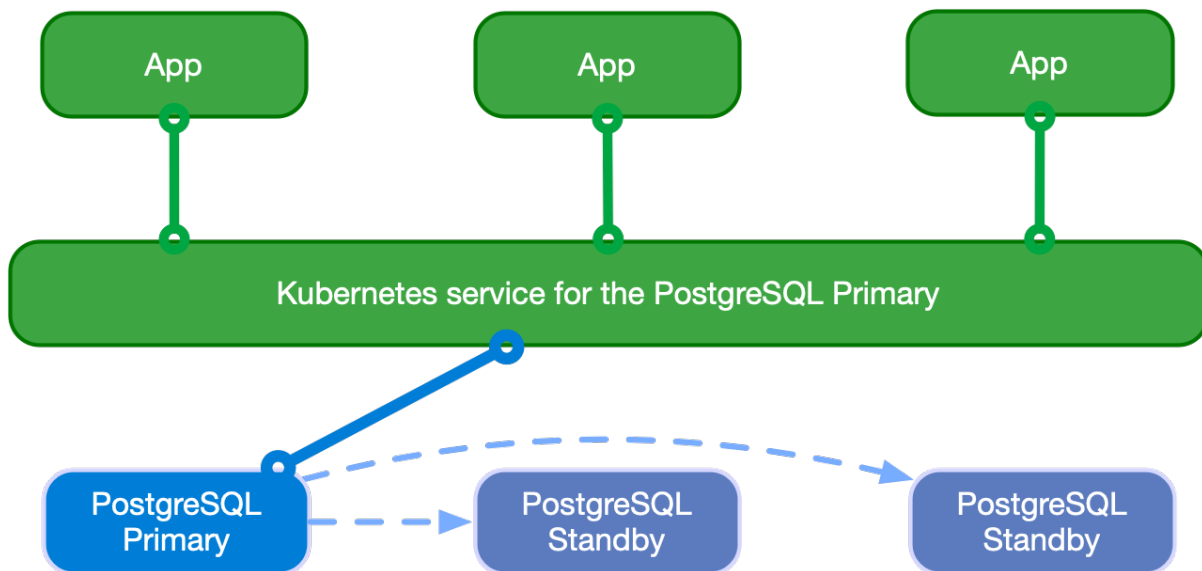


図 4: Applications writing to the single primary

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使用できない場合、Kubernetes は高可用性のために`-rw` サービスをクラスターの別のインスタンスに移動します。

4.3.2 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](#) が提示する制限を認識し、これらのワークロードを処理する際の PostgreSQL の動作に精通する必要があります。

アプリケーションは、オペレーターが使用可能にした `-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。

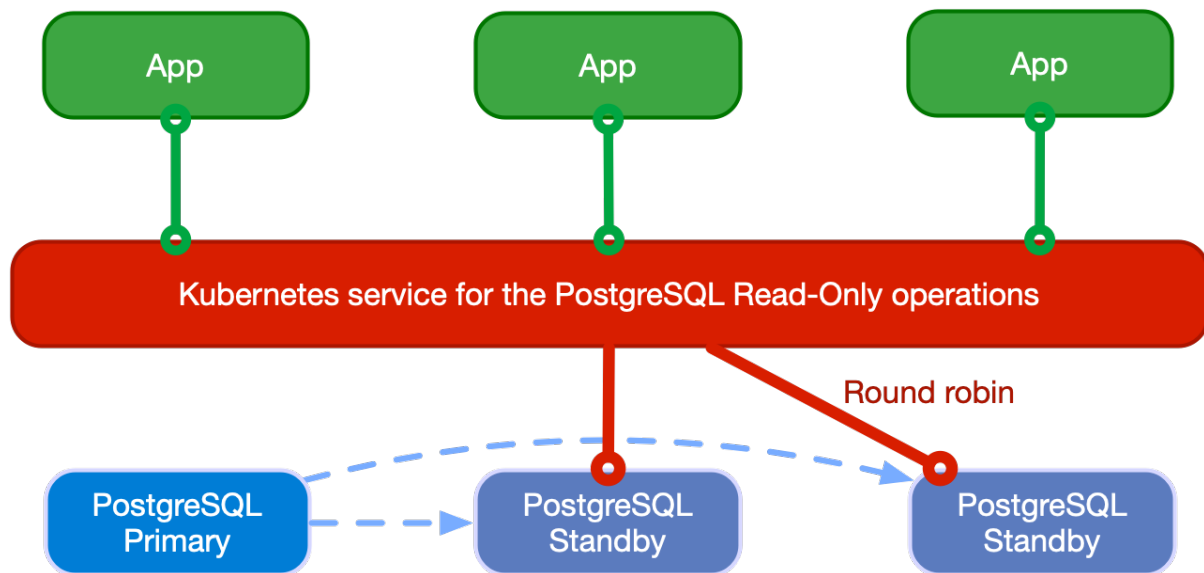


図 5: Applications reading from hot standby replicas in round robin

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

4.3.3 マルチクラスター展開

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します。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。

Tip: すべてのインスタンスが書き込みを受け入れる PostgreSQL アーキテクチャに興味がある場合は、[EDB Distributed Postgres by EDB](#) をご覧ください。Kubernetes の場合、EDB Distributed Postgres には独自の Operator があり、2022 年後半に予定されています。

ただし、ビジネス継続性目標には、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利用することにより、グローバルな目標復旧時間（RTO）を削減します（高可用性）

上記の懸念に対処するために、CloudNativePG では *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. 通常 Kubernetes クラスターの外部にある、`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
4. 復旧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、[Bootstrap 構成](#) 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、会社のディザスタリカバリークラスターとして機能し、災害が発生して最初のクラスターが使用できなくなった場合にアクティブ化できます。

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るため、レプリカクラスターを書き込み接続を受け入れられ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/フェデレーションクラスター対応の当局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、`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を使用して、WAL ストリーミング

(`primary_conninfo`) を介して供給されます。

CloudNativePG では、複数のレプリカクラスターを定義できます。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 の他のリソースと同様にインストールできます。

`latest operator manifest` は次のようにインストールできます。

```
kubectl apply -f \
  https://raw.githubusercontent.com/cloudnative-pg/cloudnative-pg/main/releases/cnpg-
  ↪1.16.0.yaml
```

`kubectl` コマンドを実行すると、CloudNativePG が Kubernetes クラスターにインストールされます。

次の方法で確認できます。

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

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 展開に関する詳細

Kubernetes では、オペレーターはデフォルトで `cnpg-controller-manager` と呼ばれる Kubernetes Deployment として `cnpg-system` 名前空間にインストールされます。次を実行して、詳細を取得できます。

```
kubectl describe deploy \
  -n cnpg-system \
  cnpg-controller-manager
```

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、ほとんどのインストールに適した単一のレプリカの Deployment が付属しています。ポッドが実行されているノードに到達できなくなった場合、ポッドは別のノードで再スケジュールされます。

オペレーターレベルで高可用性が必要な場合、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、ティントと容認を利用して、実際の PostgreSQL クラスタ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: バージョンによっては追加の手順が必要になる場合があるため、アップグレードを実行する前に [リリースノート](#) 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、次の 2 ステップのプロセスです。

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

リリースノートに特に記載されていない限り、最初のステップは通常、プレーンな Kubernetes インストール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ます。`supervised` に設定した場合、ユーザーは `kubect1` 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て、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により PostgreSQL クラスタのスイッチオーバーがトリガーされ、（通常は無視できる）ダウンタイムが発生します。

バージョン 1.10.0 以降、ローリング更新の動作をインスタ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がなく、その結果、クラスタ内でスイッチオーバーが必要です。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スマネージャーは、`init` コンテナを介して各 PostgreSQL ポッドに追加されます。

ただし、この動作は構成を介して変更して、インスタ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、コンテナを維持する PID 1 プロセスです。

内部的には、CloudNativePG のバージョン 1.10 のインスタスマネージャーは、すべての内部プロセスの正常な終了だけでなく、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

ンをサポートしています。新しいインスタスマネージャーが新しいバイナリを使用して再起動すると、既に実行されている *postmaster* 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ありません。コインの反対側は、ポッドが開始後に変更され、不変性の純粋な概念を破ることです。

オペレーター設定 で `ENABLE_INSTANCE_MANAGER_INPLACE_UPDATES` 環境変数を

'true' に設定すると、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ンニングに従います。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、バージョン `1.x.y` に対応しています。

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、少なくとも月次でオペレーターの新しいバージョン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、バージョンをスキップせずに、各バージョンを順番にアップグレードして、定期的に最新の状態にすることをお勧めします。

重要:

2022 年、EDB は頻繁なオンライン更新ができない環境での CloudNativePG の LTS リリースを計画しています。

リリースノート ページには、CloudNativePG のすべてのリリースバージョンで導入された変更の詳細なリストが含まれています。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、プレーンな Kubernetes インストールに新しいマニフェストを適用するか、選択したディストリビューションのネイティブパッケージマネージャーを使用するだけで十分です。

バージョンが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。

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、および練習のみを目的としており、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て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 プレイグラウンドのセットアップ

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。

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、日常的に Kubernetes を試したり、開発したりしようとしているユーザーのために、ラップトップ上の仮想マシン (VM) 内でシングルノードの Kubernetes クラスターを実行します。通常、VirtualBox と組み合わせて使用します。

詳細については、ローカルの個人環境にある公式の [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 というノードが 1 つ表示されます。

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、次を使用して Kubernetes クラスターを作成します。

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

4.5.2 パート 2-CloudNativePG のインストール

ラップトップで Kubernetes をインストールして実行したので、CloudNativePG のインストールを続行できます。

インストールとアップグレード セクションを参照して、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。

`cluster-example.yaml` サンプルファイルは、デフォルトのストレージクラスを使用して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
  primaryUpdateStrategy: unsupervised

  # Require 1Gi of space
```

(次のページに続く)

(前のページからの続き)

```
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
```

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ーパラダイムでは、常にコンテナイメージの特定のバージョンをポイントする必要があります。クラスター内の更新ポリシーとバージョンの一貫性に関して予測できないシナリオにつながる可能性があるため、運用環境では *latest* や *13* などのタグを使用しないでください。厳密な決定論的で反復可能な展開の場合、`<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。

4.6 ブートストラップ

このセクションでは、新しい PostgreSQL クラスターを作成するために必要なオプションとその背後にある設計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。

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

initdb ブートストラップは、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、別のメジャーバージョンの Postgres 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、継続的リカバリーにあり、読み取り専用接続を受け入れる独立した PostgreSQL クラスターである**レプリカクラスター**を作成する可能性が開かれます。

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、ローカルの Unix ドメインソケットを使用して、*peer* 認証を介して `\postgres` ユーザーとして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**

4.6.1 bootstrap セクション

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

- **initdb** : 新しい PostgreSQL クラスターを初期化します (デフォルト)
- **recovery** : バックアップオブジェクトストアを介して既存のクラスターから復元し、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クラスターを複製して、PostgreSQL クラスターを作成します-Kubernetes の外部からでも、データベースを CloudNativePG に移行する場合に役立ちます。

initdb メソッドとは異なり、*recovery* と *pg_basebackup* 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。

4.6.2 externalClusters セクション

externalClusters セクションを使用すると、現在のクラスターに関連する 1 つ以上の PostgreSQL クラスターを定義できます。将来的には、このセクションではより複雑なシナリオが可能になりますが、現在、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 - ソースクラスターのバックアップファイル、つまりベースバックアップと WAL アーカイブを含む Barman Cloud 互換のオブジェクトストアである **リカバリオブジェクトストア**に関する情報。

注釈: 復旧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、ソースは `recovery` メソッドに使用できます。両方が定義されている場合、2つのブートストラップ方法のいずれかを選択できます。

さらに、`pg_basebackup` または完全な `recovery` ポイントインタイム)、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つのいずれかから継続的に供給されることを意味します。

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。特に指定し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` と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 の仕様に準拠する必要があります。その結果、シークレットの `username` は、スーパーユーザーの `owner` (アプリケーションシークレットの場合) および `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ります。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: この演算子の将来の実装では、宣言的な構成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 `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。代わりに、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: 現在、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ていません。

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を検出します（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（デフォルト : **UTF8**）のエンコーディングを選択します。

localeCollate : **localeCollate** に値が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、PostgreSQL ドキュメントの [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（デフォルト : **C**）。

localeCTYPE : **localeCTYPE** に値が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、PostgreSQL ドキュメント（デフォルト : **C**）の [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。

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

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、オペレーターの動作を壊す可能性のあるオプション（**--auth** や **-d** など）があることを考えると、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:
  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: クエリはスーパーユーザーとして実行され、クラスター全体を混乱させる可能性があるため、*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。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:
```

(次のページに続く)

(前のページからの続き)

```

- name: my-secret
  key: secret.sql
configMapRefs:
- name: my-configmap
  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の順序に従って実行されます。SQL スクリプト内では、各 SQL ステートメントは [PostgreSQL semantics](#) に従ってサーバー上の単一の `exec` で実行されますが、コメントを含めることができます *psql*。

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、またはリカバリーオブジェクトストア (`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。 `backupObjectStore.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、既存のバックアップ、つまりリカバリーオブジェクトストアから新しいクラスターを作成できます。

CloudNativePG でこの結果を得るには 2 つの方法があります。

- リカバリー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)
- 同じ名前空間にある既存の Backup オブジェクトを使用します (これはバージョン 1.8.0 より前に使用可能な唯一のオプションでした)。

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

が有効になります。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。

注釈: 実行中のクラスターのバックアップとリカバリーの詳細は、**バックアップとリカバリー** で見つけることができます。

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構成をすべて含めて、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 コンテナに回復オブジェクトストアを定義します。

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

  superuserSecret:
    name: superuser-secret

  bootstrap:
    recovery:
      source: clusterBackup

  externalClusters:
    - name: clusterBackup
      barmanObjectStore:
```

(次のページに続く)

(前のページからの続き)

```

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。 `backupObjectStore.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、回復時間が大幅に短縮されます。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****（必要になるか）。

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バックアップ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:
    name: superuser-secret

  bootstrap:
    recovery:
      backup:

```

(次のページに続く)

```
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。

その他の考慮事項

回復オブジェクトストアまたは既存の **Backup** リソースのいずれから回復する場合でも、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、復元されているバックアップから保持されます。これは Kubernetes クラスター自身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をバックアップしようとしません。
- `superuserSecret` を指定しない場合、安全な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。このシークレットは、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（最大 11 の PostgreSQL の場合は `current`、バージョン 12 以降の場合）でリカバリが実行されます。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ー \(PITR\)](#) 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ー (PITR)

最新の WAL までのすべての 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 はこの手法を使用して、**ポイントインタイムリカバリー (PITR)** を実現します。

注釈: PITR は、リカバリー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 15:22:00.000000+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なく、タイムスタンプの形式で `targetTime` を指定するだけであることに気付いたかもしれません。

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

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

重要: このようなバックアップが存在し、アクセスでき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ーのベースバックアップを自動的に検出します。

- `targetTime` または `targetLSN` を使用すると、オペレータ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のタイムスタンプ（正確な停止ポイントは

exclusive オプションの影響も受けます）

targetXID : リカバリが実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の開始時にシーケンシャルに割り当てられますが、トランザクションは異なる番号順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（オプションで含めて）コミットされたトランザクションです。

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あります）

targetImmediate : リカバリは、一貫した状態に到達次第、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ポイントを意味します

重要: *targetTime* または `targetLSN` が指定されている場合、オペレーターは最も近いバックアップを自動的に取得できますが、残りのターゲットである `targetName`、*targetXID*、および `targetImmediate` はできません。このような場合、カタログ内の使用可能な最新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
    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の名前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、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シークレットを生成させることもできます。シークレットの詳細については、[空のクラスターをブートストラップする（initdb）](#) <空のクラスターをブートストラップする（initdb）> セクションを参照してください。

次の例では、所有者が `app`、指定されたシークレットが `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 `secret` の `owner` の値と一致する場合、アプリケーションデータベースのパスワードが `secret` の `password` の値に変更されます。

重要: レプリカモードが有効になっているレプリカクラスターの場合、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回復されます。

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の PostgreSQL サーバーです。

このメソッドの主なユースケースは、Kubernetes 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、クローン作成プロセスが終了してすぐに作成されたクラスターを起動すると、オリジンの 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 で必要な一部の構成パラメーターのオーバーライド、スーパーユーザーのパスワードのリセット、`streaming_replica`

ユーザーの作成、レプリカの管理が含まれます。結果のクラスターは、ソースインスタンスから完全に独立しています。

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

`pg_basebackup` によって透過的に管理されるターゲットインスタンス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 で管理されているソースまたは TLS 認証用に構成されたソースに接続する場合は、後者が推奨されます。ただし、最初のオプションは、一般的に PostgreSQL サーバーに対する認証の最も一般的な形式であり、ソースインスタンスが Kubernetes 外部の従来の環境にある場合に最も簡単な方法です。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
- ソースには表領域を定義しないでください（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` を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る必要があり、`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` 権限を持つロールを使用してソース PostgreSQL インスタンスに正常に接続できる必要があります

参考:

詳細については、PostgreSQL ドキュメントの [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`)、および [High Availability, Load Balancing, and Replication](#) を参照してください。

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、これが `streaming_replica` に許可されている接続方法です）。

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て保存します。

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れば、必要に応じて自由に変更できます。

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 でサポートされてい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` という新しい PostgreSQL 14.4 クラスターを作成し、`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 クラスターを複製します。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。そのパスワードは `source-db-replica-user` シークレットの `password` キーに保存されます。

```

apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: target-db
spec:
  instances: 3
  imageName: ghcr.io/cloudnative-pg/postgresql:14.4

  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

```

クローン操作を機能させるには、同じ PostgreSQL バージョン（この場合は 14.4）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 でサポートされている 2 番目の認証方法は、TLS クライアント証明書に基づいています。これは、セキュリティの観点から推奨されるアプローチです。

次の例では、同じ Kubernetes クラスター内に既存の 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応できます。

マニフェストは、`cluster-clone-tls` と呼ばれる新しい PostgreSQL 14.4 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の `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、`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）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:14.4

  bootstrap:
    pg_basebackup:
      source: cluster-example

  storage:
    size: 1Gi

  externalClusters:
    - name: cluster-example
      connectionParameters:
        host: cluster-example-rw.default.svc
        user: streaming_replica
        sslmode: verify-full
      sslKey:
        name: cluster-example-replication
        key: tls.key
      sslCert:
        name: cluster-example-replication
        key: tls.crt
      sslRootCert:
        name: cluster-example-ca
        key: ca.crt

```

アプリケーションデータベースを構成する

initdb および recovery ブートストラップ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場合）、アプリケーションデータベースの構成はスキップされます。

次の例では、ライブクラスターからのブートストラップ後に、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。

```

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

```

(次のページに続く)

(前のページからの続き)

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

上記の構成では、リカバリ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 `secret` の `owner` の値と一致する場合、アプリケーションデータベースのパスワードが `secret` の `password` の値に変更されます。

重要: レプリカモードが有効になっているレプリカクラスターの場合、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回復されます。

現在の制限

表領域のサポートがありません

CloudNativePG には現在、PostgreSQL グローバルオブジェクト、つまりロール、データベース、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、つまり追加の永続ボリュームの定義と管理が必要です。ベースバックアップとテーブルスペースを扱う場合、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が新しいタイムラインで開始され、ソースから分岐します。このため、Kubernetes 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 までお気軽にお問い合わせください。

CloudNativePG の将来のバージョンでは、別の PostgreSQL インスタンスのレプリカである新しいクラスターを作成することにより、ユーザーがログ先行書き込み (WAL) シッピングを介して PostgreSQL の継続的復旧メカニズムを制御できるようになります。これにより、2 つの主要なユースケースが開かれます。

- CloudNativePG の異なる Kubernetes クラスターにわたるレプリケーション
- - 0 カットオーバー時間* `pg_basebackup` ブートストラップメソッドを使用した CloudNativePG への移行

4.7 Postgres データベースのインポート

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

インポート操作は、PostgreSQL のオンライン論理バックアップの概念に基づいており、元のホストへのネットワーク接続を介した `pg_dump`、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 では、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一貫したバックアップを同時に取得できます。

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 10.x からバージョン 14.x など)

警告: PostgreSQL のメジャーアップグレードを実行する場合、アプリケーションが新しいバージョンと互換性があること、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、バックアップの開始後にソースデータベースに加えられた変更は宛先クラスターにないため、*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）を作成し、`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にす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* タイプを介して使用でき、後者は *monolith* タイプを介して使用できます。

警告: あなたの責任で、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権限を持つユーザーでソースクラスターにアクセスできます。詳細は [PostgreSQL documentation on "SQL Dump"](#) を参照してください。

4.7.2 microservice タイプ

マイクロサービスのアプローチでは、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

- `initdb.owner` ユーザーが所有する `initdb.database` (アプリケーションデータベース) への `pg_restore --no-acl --no-owner` を使用したデータベースのインポート
- データベースダンプファイルのクリーンアップ
- `postImportApplicationSQL` パラメータ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

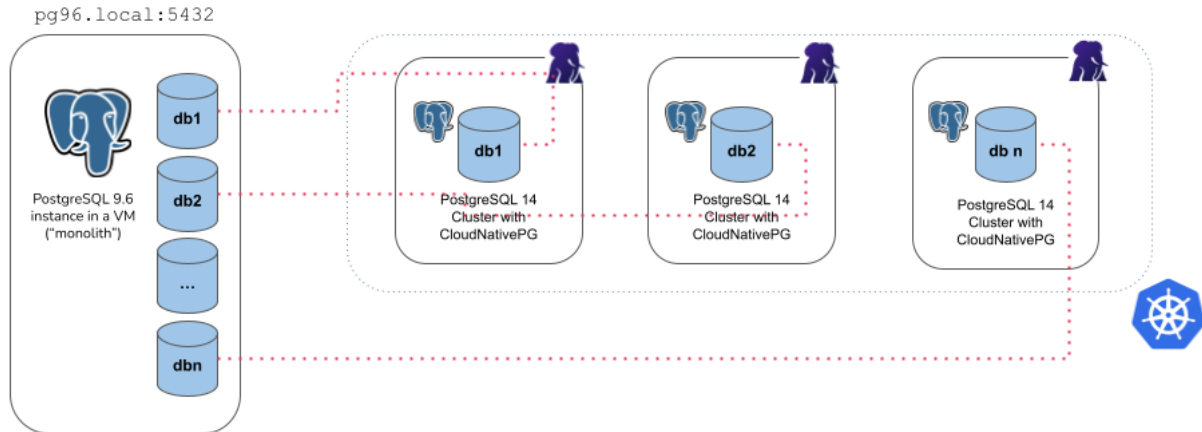


図 6: Example of microservice import type

たとえば、以下の YAML は、`postgres` に接続することにより、`cluster-pg96` クラスター (サポートされていない PostgreSQL 9.6 を使用) から `angus` データベースをインポートする `cluster-microservice` という新しい 3 インスタンスの PostgreSQL クラスターを作成します。`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
      databases:
        - angus
      source:
        externalCluster: cluster-pg96
        #postImportApplicationSQL:
        #- |
        # INSERT YOUR SQL QUERIES HERE
  storage:
```

(次のページに続く)

(前のページからの続き)

```

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バージョンである必要があります。私たちの経験によれば、データをエクスポートする方法は PostgreSQL の古いバージョンやサポートされていないバージョンでも機能し、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、ご連絡をお待ちしております。

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)
- 操作中に Kubernetes クラスターと `externalCluster` 間の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問題ありません)
- 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含めるための十分な領域があるはずです。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ないため、`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

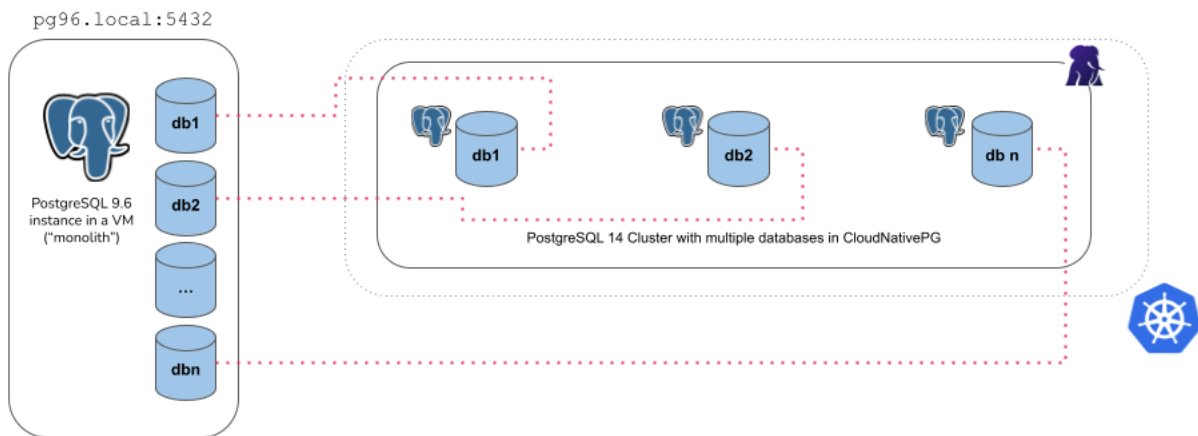


図 7: Example of monolith import type

たとえば、以下の YAML は、`accountant` と `bank_user` のロール、および `cluster-pg96` から `accounting`、`banking`、`resort` データベースをインポートする `cluster-microservice` という新しい 3 インスタンスの PostgreSQL クラスター（オペレーターがリリースされた時点で利用可能な最新のメジャーバージョン）を作成します。クラスター（サポートされていない PostgreSQL 9.6 を使用）、`postgres` ユーザーを使用して、`cluster-pg96-superuser` シークレットに保存されているパスワードを使用して `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:
    initdb:
      import:
        type: monolith
        databases:
```

(次のページに続く)

(前のページからの続き)

```

    - 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)
- 操作中に Kubernetes クラスターと **externalCluster** 間のトラフィックを許可する必要があります
- SSL なしで PostgreSQL インスタンスに接続する必要がある場合は、**connectionParameters** セクションで `sslmode: disable` を指定する必要があります
- **pg_dump** を実行してロール情報を取得する必要がある指定されたユーザー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問題ありません)
- 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含めるための十分な領域があるはずです。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。-次のロールがある場合、インポートされません:`postgres`、`streaming_replica`、`cnp_pooler_pgouncer`
- ワイルドカード "*" は、その種類のすべてのオブジェクトをインポートする **databases** および/または **roles** 配列の唯一の要素として使用できます。データベースを一致させる場合、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ていないデータベースを無視します

- `clone` 手順が完了した後、すべてのデータベースに対して `ANALYZE VERBOSE` が実行されます。
- `postImportApplicationSQL` フィールドはサポートされていません

4.8 セキュリティ

このセクションには、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る CloudNativePG のセキュリティに関する情報が含まれています。

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を免責してはなりません。Kubernetes ドキュメントの [Overview of Cloud Native Security](#) ページを理解してください。

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 - いくつかの例を挙げます。

重要: CI / CD パイプラインの静的コード分析フェーズでの障害は、CloudNativePG の配信全体のブロッカーです。

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、すべてのコミットに続いて CI / CD パイプラインを介して自動的にビルドされます。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、画像は次のようにスキャンされます。

- [Docker](#) : 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 日に 1 回自動的にリビルドされ、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

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、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がバインドされます。

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にのみ予約されています。 `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、さまざまな Cluster のフェーズを管理するためにジョブを処理する必要があります。

persistentvolumeclaims : PGDATA が存在するボリュームは、PostgreSQL Cluster リソースの中心的な要素です。オペレーターは、定義されたスケジューリングポリシーに基づいて、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、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な両方の Webhook 構成に自己署名 Webhook CA を挿入します。詳細については、[Kubernetes documentation](#) をご覧ください。

nodes : オペレーターは、アフィニティとアンチアフィニティ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ゾーンにある場合、ポッドをスケジュールしてレプリカが同じノードにならないように決定できます。この権限は、ノードがスケジュールされているかどうかを判断するためにも使用され、まったく作成できない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ための Kubernetes の方法です。

InfoSec の理由から、すべての Kubernetes プラットフォームで実装する必要があります。

CloudNativePG では、コンテナの実行に**特権**モードは必要ありません。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、クラスターが動作しなくなる場合があります。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起動できず、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 管理者に連絡して、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を Kubernetes

ネットワークポリシー で制御して、IP および TCP レベルで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/無効にすることができます。

重要: オペレーターは、TCP ポート 8000 で各インスタンスと通信して、PostgreSQL サーバーのステータスに関する情報を取得する必要があります。ネットワークポリシーを追加する場合はこのことに注意してください。CloudNativePG がより詳細に制御するために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。詳細については、Kubernetes ドキュメントの **ネットワークポリシー** セクションを参照してください。

公開されたポート

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
オペレーター	9443	はい	webhook-server	TLS	はい
メトリックス	8080	はい	metrics	TLS	なし
インスタスマネージャー	9187	なし	metrics	TLS	なし
インスタスマネージャー	8000	なし	status	TLS	なし
オペランド	5432	なし	PostgreSQL	なし	なし
PostgreSQL	5432	はい	PostgreSQL	なし	なし

PostgreSQL

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

パスワードの暗号化に関する限り、CloudNativePG は PostgreSQL のデフォルトの動作に従います。PostgreSQL 14 以降、`password_encryption` はデフォルトで `scram-sha-256` に設定されていますが、以前のバージョンでは `md5` に設定されています。

重要: 詳細については、PostgreSQL ドキュメントの **ユーザー名/パスワード認証** 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（パスワード認証を介したりモートアクセスを事実上無効にします）。

詳細については、[PgBouncer の秘密](#) を参照してください。

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

デフォルトでは、すべてのレプリカは、`streaming_replica` と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に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直接 `pg_hba.conf` 行を追加することを許可します。マニフェストで定義された行がデフォルトの `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。

4.9 Postgres インスタスマネージャー

CloudNativePG は、フェイルオーバー管理を外部ツールに依存しません。これは、Kubernetes API サーバーと、**Postgres インスタスマネージャー**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 の有効期間中、インスタ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 プロブは `pg_isready` に依存していますが、readiness プロブはデータベースが稼働しており、スーパーユーザーの資格情報を使用して接続を受け入れることができるかどうかを確認します。Pod がトラフィックを受け入れる準備ができている場合、readiness プロブはポジティブです。liveness probe は、コンテナ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 回失敗すると、2 つのプローブが失敗を報告します。

現時点では、起動プローブは Kubernetes 1.17 でのみ導入されているため、オペレーターはポッドで `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します。

Pod が起動してから liveness プロ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ます。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` が低すぎると、liveness probe 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削除されると、kubelet はインスタスマネージャーに終了シグナルを送信し、インスタスマネージャーは適切な方法で PostgreSQL をシャットダウンします。秒単位で表される `.spec.stopDelay` は、PostgreSQL がシャットダウンする時間です。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、インスタスマネージャーは **高速** のシャットダウンを要求し、既存の接続を終了してすぐに終了します。インスタンスが WAL ファイルをアーカイブおよび/またはストリーミングしている場合、プロセスは `.spec.stopDelay` で設定された時間の残りの半分まで待機してから、強制的にシャットダウンします。

重要: データベースの RPO に影響を与える Postgres クラスターでのデータの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切り替えを実行します。

スイッチオーバー中のプライマリのシャットダウン

スイッチオーバー中のシャットダウン手順は、一般的な場合とは少し異なります。実際、オペレーターは、新しいプライマリですべてのデータを利用できるように、選択した新しいプライマリを昇格させる前に、以前のプライマリが**高速** シャットダウンを発行する必要があります。

このため、秒単位で表される `.spec.switchoverDelay` は、以前のプライマリが正常にシャットダウンしてすべての WAL ファイルをアーカイブする時間を制御します。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。デフォルト値は 1 年 (秒) より大きく、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、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、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を長時間放置するときに、データ損失のリスクが削除される場合があります。

4.9.3 フェールオーバー

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 [自動フェールオーバー](#) を参照してください。

4.10 スケジューリング

Kubernetes 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プロセスです。

クラスターの定義の [AffinityConfiguration](#) セクションを使用して、CloudNativePG クラスターのインスタンスをスケジュールする方法を制御できます。

- ポッドアフィニティ/アンチアフィニティ
- ノードセクター
- 寛容

4.10.1 Pod のアフィニティとアンチアフィニティ

Kubernetes では、それらのノードで既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**）かしない（**アンチアフィニティ**）かを制御できます。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好ましくは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
                values:
                  - cluster-example
          topologyKey: kubernetes.io/hostname
        weight: 100

```

次のクラスタ仕様の結果：

```

apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  imageName: ghcr.io/cloudnative-pg/postgresql:14.4

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

  storage:
    size: 1Gi

```

したがって、Kubernetes は3つの異なるノードで3ノードの PostgreSQL クラスタをスケジュールすることを優先します。

上記の設定を調整することで、前述のデフォルトの動作を変更できます。

`podAntiAffinityType` を `required` に設定できます。その結果、`preferredDuringSchedulingIgnoredDuringExecution` の代わりに `requiredDuringSchedulingIgnoredDuringExecution` が使用されます。このような強力な要件により、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 クラスタの自動水平スケールに

[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、ポッドがノードだけでなくアベイラビリティゾーンに分散されるように、`topology.kubernetes.io/zone` です。その他のオプションについては、

[Well-Known Labels, Annotations and Taints](#) を参照してください。

`enablePodAntiAffinity` を `false` に設定することにより、オペレーターの生成され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: Pod 仕様で予想される `podAntiAffinity` または `podAffinity` のコンテンツ全体をワーカーまたは `additionalPodAffinity` に渡す必要があります (どの PostgreSQL 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、これらのラベルがある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 寛容

Kubernetes では、ノードが `taints` を明示的に許容していない (`tolerations` を介して) 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。

したがって、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に入れ、ワークロードをスケジュールするノードを決定します。容認は、容認の通常の Kubernetes 構文を受け入れる `.spec.affinity.tolerations` セクションを介して、クラスターのすべてのポッドに対して構成できます。

4.11 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、クラスターのポッドによるリソース使用量を 2 つのノブで制御および管理できます。

- **requests** : 初期要件
- **limits** : リソースニーズが動的に増加する場合の最大使用量

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ますが、ポッドにはメモリの要求と制限があると考ええると便利です。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、制限ではなく要求に基づいています。ポッドのメモリ要求を満たすのに十分なメモリがノードにある場合にのみ、ポッドは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、Kubernetes ドキュメントの [Configure Quality of Service for Pods](#) セクションを参照してください。

PostgreSQL のワークロードの場合、「保証」の 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してください。

- 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) と 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を「保証」QoS クラスに割り当てるには、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の “`affinityconfiguration`” リソースの「`nodeSelector`」および「`tolerations`」フィールドを参照してください。

次のマニフェストの例を参照できます。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  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 サーバー専用割り当てられるメモリーの量（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）。

`shared_buffers` の妥当な開始値は、システムのメモリーの 25% です。例： `shared_buffers` が 256 MB の場合、コンテナのメモリーサイズの推奨値は 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 のメモリーがあることを意味します。期待どおりに動作します。詳細については、PostgreSQL ドキュメントの [Resource Consumption](#) セクションを参照してください。

4.12 障害モード

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、サポートと支援についてはすぐに EDB に連絡してください。

4.12.1 ストレージ使用量

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。

このようなストレージ領域は、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場合（および新しい 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 を再利用しないようにするには、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]
```

例：

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

4.12.2 障害モード

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 プロブが失敗します。
- postgres コンテナの活性プロブが失敗します。
- 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する。

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異なる影響を与えます。

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 がある場合は再利用して、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 を意図的に削除した場合、*PodDisruptionBudget* ポリシーは適用されません。

自己修復は、*apiserver* に通知されるとすぐに発生します。

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** コンテナは失敗したと見なされます。Pod 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別のポッドがある場合、ポッドのエビクト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* に通知されるとすぐに発生します。

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許容秒後に削除用にマークされます。**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。

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

tolerationSeconds の後に自己修復が行われます。

4.12.3 自己修復

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。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、オペレーターはステータスが `ready` でレプリケーションラグが最小のアクティブなポッドを昇格させ、`-rw` サービスをポイントします。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、PVC が利用可能な場合、`pg_rewrite` を使用して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支援と支援なしに手動操作を実行しないでください。

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

```
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、臨時/緊急操作の唯一の期間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に残すと、オペレーターが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` の値の変更
- 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、ポッドが最新のインスタスマネージャーを実行するようにします (in-place updates are enabled を除く)。

オペレーターは、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。

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

アップグレードでは、データを再複製せずに 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はクラスターのステータスを反映するように移動するため、アプリケーションは更新中のノードを無視できます。

4.13.1 自動更新 (`unsupervised`)

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

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

- `switchover` : スイッチオーバー操作が自動的に実行され、最も調整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 (デフォルト)。
- `restart` : 可能であれば、プライマリインスタンスが実行されているポッドの自動再起動を実行します。それ以外の場合、再起動要求は無視され、スイッチオーバーが発行されます。

更新方法には、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するため、万能の構成はありません。に。

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、プロモートされたインスタンスがコンテナ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です。

4.13.2 手動更新 (supervised)

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

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

次の方法でスイッチオーバーをトリガーできます。

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

次の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、`cnpg` 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データを管理するために 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できます。

重要: このセクションでは、同じ Kubernetes クラスターで管理される同じ `Cluster` リソース内のレプリケーションについて説明します。異なる Kubernetes クラスター間でも、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](#) セクションを参照してください。

4.14.1 アプリケーションレベルのレプリケーション

長年にわたって PostgreSQL のレプリケーション機能に貢献してきた私たちは、ネイティブの物理レプリケーションテクノロジーの上に 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。

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) で連続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強化されましたが、9.1 (2011) ではトランザクションレベル (RPO = 0 クラスタ) での同期レプリケーションが導入されました。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 に置かれていましたが、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ためのパブリッシャー/サブスクライバーパターンのネイティブサポートが導入されました。

4.14.3 ストリーミングレプリケーションのサポート

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

```
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 NOCREATOROLE NOCREATEDB NOBYPASSRLS
```

注釈: pg_rewind 要件のため、PostgreSQL 10 では `streaming\_replica` ユーザーが `SUPERUSER` 権限で作成されます。

オペレーターはすぐに、暗号化されたチャンネルを介してクラスター内にストリーミングレプリケーションを設定し、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ます。

```
# Require client certificate authentication for the streaming_replica user
hostssl postgres streaming_replica all cert
hostssl replication streaming_replica all cert
```

継続的なバックアップの統合

クラスターで継続的なバックアップが構成されている場合、CloudNativePG は継続的なリカバリ時に `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 のプルが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` と `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ています。これらは、常に使用可能な予想される同期スタンバイレプリカの数です。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を無差別に選択します。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、オペレーターが自動的に計算した整数です。 $1 \leq \text{minSyncReplicas} \leq q \leq \text{maxSyncReplicas} \leq \text{readyReplicas}$
- `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` の値が現在使用可能なレプリカの数より大きい場合、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複製されるまでトランザクションのコミットを待機させます*。

重要: オペレーターは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ますが、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ゾーンにあることを保証できます。これにより、特に目標復旧ポイント（RPO）の観点から、単一のアベイラビリティゾーンで障害が発生した場合のクラスターの堅牢性が向上します。

アンチアフィニティの考え方は、クォーラムに参加する同期レプリカが、選択されたラベル（この場合、アベイラビリティゾーンラベル）の値が異なるノードで実行されており、プライマリが現在あるノードから選択されるようにすることです実行中。このような基準に一致するノードがない場合、レプリカは同期レプリケーションの対象となります。

重要: 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、自己修復の強制が適用されます（[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、アベイラビリティゾーンは単なる例ですが、ストレージ、CPU、メモリなど、ノードを説明する他のラベルに基づいてこの動作をカスタマイズできます。

4.15 バックアップとリカバリー

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して暗号化できます。

このためには、`barman-cli-cloud` を含む画像を使用する必要があります。このスコープにはイメージ `ghcr.io/cloudnative-pg/postgresql` を使用できます。これは、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ます。

重要: Barman cloud で導入された機能強化を利用するには、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（クラスターのセキュリティ面を向上させます）。

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（指定されたプライマリインスタンスの詳細については、

[レプリカクラスター](#) を参照してください）。

4.15.1 クラウドプロバイダーのサポート

Barman Cloud インフラストラクチャでサポートされている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。

必要な設定は、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

環境に関する次の情報が必要です。

- `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

使用す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ます。次のコマンドで作成できます。

```
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
[...]
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 です。
`s3://BUCKET_NAME/path/to/folder`。

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 Linode Object Storage 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、リージョン us-east1 の Linode の bucket バケットを使用します。

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

オプションで、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バックアップオブジェクトを中継する共通インターフェイスとして MinIO ゲートウェイを使用できます。詳細については、[MinIO official documentation](#) を参照してください。

具体的には、CloudNativePG クラスターは、以前に作成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、同じ名前空間に作成する必要があります。

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

注釈: Cloud Object Storage の資格情報は、この場合 MinIO ゲートウェイ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は、Kubernetes の公式の [MinIO Operator](#) 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。

以下は、AWS S3 をクラウドオブジェクトストレージとして使用した例です。

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

(次のページに続く)

(前のページからの続き)

```

    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
[...]
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
      [...]

```

バックアップを進める前に、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** が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、次のようにクラスター構成に `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。

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
```

(次のページに続く)

(前のページからの続き)

```

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

```

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

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

ここで、`<resource-path>` は`<container>/<blob>` です。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。1 つはポッドが Google Kubernetes Engine クラスタ内で実行されていることを前提とし、もう 1 つは環境変数 `GOOGLE_APPLICATION_CREDENTIALS` を利用します。

Google Kubernetes Engine 内で実行

これは、バックアップを作成する最も簡単な方法の 1 つであり、次の構成のみが必要です。

```

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: "gs://<destination path here>"
  googleCredentials:
    gkeEnvironment: true

```

これにより、クラスターが Google Kubernetes Engine 内で実行されていることをオペレーターに伝えます。つまり、ファイルのアップロードに資格情報は必要ありません

重要: この方法では、クラスターとポッドに注意深く定義された権限が必要です。これらはクラスター管理者が定義する必要があります。

認証の使用

`instruction from Google` に従うと、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 という名前で 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 バケットへのアクセスに必要なすべての情報を含む JSON ファイルがコンテナ内に作成されます。つまり、誰かがポッドにアクセスすると、バケットへの書き込み権限も付与されます。

4.15.2 オンデマンドバックアップ

新しいバックアップを要求するには、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
spec:
  cluster:
    name: pg-backup
```

オペレーターは、`barman-cloud-backup` を使用して必要なバックアップを取得するようにクラスターのオーケストレーションを開始します。プレーンな `kubectl describe backup <name>` コマンドを使用して、バックアップ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:
```

(次のページに続く)

(前のページからの続き)

```

Cluster:
  Name: pg-backup
Status:
  Backup Id:      20201026T135740
  Destination Path: s3://backups/
  Endpoint URL:   http://minio:9000
  Phase:          completed
  s3Credentials:
    Access Key Id:
      Key:  ACCESS_KEY_ID
      Name: minio
    Secret Access Key:
      Key:  ACCESS_SECRET_KEY
      Name: minio
  Server Name: pg-backup
  Started At:  2020-10-26T13:57:40Z
  Stopped At:  2020-10-26T13:57:44Z
Events:       <none>

```

!!!重要この機能では、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はバックアップされません。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。

```

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

```

上記の例では、毎日午前 0 時にバックアップをスケジュールします。

ヒント: バックアップの頻度は、完全またはポイントインタイムの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）に影響を与える場合があります。私たちのアドバイスは、バックアップを復元して定期的にテストし、最初から復元するのにかかる時間を測定して、RTO の予測可能性を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、アーカイブから取得してリカバリ中に再生する必要がある WAL ファイルの量に影響されます（WAL アーカイブは PostgreSQL での連続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いて、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` を設定することで一時停止できます。これにより、オプションが `false` に設定されている限り、スケジュールされた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` を設定します。

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

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

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、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、アーカイブするすべての WAL セグメントに対して `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、少なくとも 5 分ごとに WAL ファイルが閉じられてアーカイブされ、目標復旧時点（RPO）の確定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、オペレーターによって設定されたデフォルト値がほとんどのユースケースに適しています。

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 ファイルを並列に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列 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 を並列に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 リカバリー

クラスターの復元は、既存のクラスターでは「インプレース」で実行されません。オブジェクトストレージにアップロードされたデータを使用して、以前に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（要求に応じて並列サポートを含む WAL ファイル用）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。

内部では、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` コンテナを挿入し、`init` コンテナはオブジェクトストレージからのバックアップの回復を開始します。

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

ベースバックアップリカバリープロセスが完了すると、オペレーターは PostgreSQL インスタンスをリカバリー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、このフェーズを高速化できます)。

このフェーズは、PostgreSQL がターゲット (WAL の終わりまたはポイントインタイムリカバリの場合は必要なターゲット) に到達すると終了します。実際、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 `latest`、バージョン 12 以降の場合)。

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の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対して透過的であり、ポッドで実行されているインスタスマネージャーによって管理されます。

バックアップセクションを使用し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。

警告: リカバリーのために、`externalClusters` セクションと同じ ``barmanObjectStore`` オブジェクトを使用することは許可されていません。場合によっては、これは機能*できますが、ソースクラスターと同じバックアップオブジェクトを使用している場合、オペレーターは新しいクラスターへのリカバリを防ぎます。これは、ソース WAL が上書きされる可能性のあるエッジケースから保護するためです。リカバリとバックアップに再利用された同じ ``barmanObjectStore`` を持つクラスターは、検証エラーを生成します: `spec.backup.barmanObjectStore: Invalid value: ...`

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、回復されたクラスターのベースバックアップと WAL が保存される `recoveredCluster` という名前のストレージバケットに新しいフォルダーを作成します。

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

(次のページに続く)

```
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: 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安全性チェックがオペレーターに含まれるようになりました。既存のストレージを上書きするクラスターは、ポッドが Error 状態である *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"

4.15.6 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"
```

4.15.7 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 [WalBackup 構成](#) 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。圧縮アルゴリズムは、ユースケースに応じて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実行しました。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
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

4.15.8 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際のタグ付けのサポートが導入されています。その結果、PostgreSQL コンテナイメージにバージョン 2.18 以降の Barman が含まれている場合、CloudNativePG 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のタグ
- **historyTags** : バックアップオブジェクトストア内のアーカイブされた履歴ファイルに追加されるキーと値のペアのタグ

以下の YAML マニフェストの抜粋は、この機能の使用例を提供しています。

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

(次のページに続く)

```
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強制的に変更しないでください。通常はオペレーターによって制御されるオプションの一部を変更すると、実際にクラスターが予測不能/回復不能な状態になる可能性があります。さらに、`ALTER SYSTEM` の変更はクラスター全体に複製されません。

カスタム設定の使用法のリファレンスはサンプルに含まれています。

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

4.16.1 postgresql セクション

ポッド内の PostgreSQL インスタンスは、デフォルトの `postgresql.conf` ファイルで起動します。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` セクションにユーザー定義の設定が含まれます。

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

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

- グローバルデフォルトパラメーター
- PostgreSQL の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
- ユーザー提供のパラメーター
- 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、サーバーバージョンに応じて、予想および観察されたワークロードに基づいて、`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。CloudNativePG がレプリケーションスロットをサポートするまで、そして継続的なバックアップを実施していない場合、これが現時点でスタンバイが同期しなくなって次のようなエラーメッセージを返す場合から保護する唯一の方法です。これには、`PGDATA` の一部を専用にして、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必要があります。

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

```
archive_command = /controller/manager wal-archive %p
archive_mode = on
full_page_writes = on
hot_standby = true
listen_addresses = *
port = 5432
restart_after_crash = false
ssl = on
ssl_ca_file = /controller/certificates/client-ca.crt
ssl_cert_file = /controller/certificates/server.crt
ssl_key_file = /controller/certificates/server.key
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、ユーザーが YAML 構成を介して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従って、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が自動的に解析して 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使用できる必要がある拡張機能（たとえば、`pg_stat_statements`）をロードします。

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、専門家の PostgreSQL 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 は自己修復を試行できず、手動介入が必要です。 `shared_preload_libraries` のコンテンツを直接管理する予定がある場合は、常に拡張機能と設定の両方を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必要としないとき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、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` を介して追加の `shared_preload_libraries` を文字列のリストとして提供できます。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いる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。現在のリストには次のものが含まれます。

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

これらのライブラリの中には、使用する前にデータベース内に追加のオブジェクトが必要です。通常、データベースで実行するには `CREATE EXTENSION` コマンドで管理されるビューおよび/または関数です（通常、`DROP EXTENSION` コマンドはこれらのオブジェクトを削除します）。

このようなライブラリの場合、CloudNativePG は、次のクエリで識別されるクラスター内の接続を受け入れるすべてのデータベースで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 dataallowconn
```

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

を手動で実行せずに、遅いステートメントの実行プランを自動的にログに記録する手段を提供します（最適化されていないクエリを追跡するのに役立ちます）。

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

```
# ...
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
のリアルタイム監視のために PostgreSQL で使用できる最も重要な機能の 1 つです。

次の例の抜粋のように、`pg_stat_statements.` で始まるパラメーターを構成に追加することにより、`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` を実行するため、`pg_stat_statements` ビューに対してクエリを実行できます。

pgaudit を有効にする

pgaudit 拡張機能は、標準の PostgreSQL ログ機能を通じて詳細なセッションおよび/またはオブジェクト監査ログを提供します。

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](#)

次の例の抜粋のように、pgaudit. で始まるパラメーターを構成に追加することにより、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"
#
```

4.16.2 pg_hba セクション

pg_hba は、ポッドで使用する 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` に設定されています。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は、次の抜粋に示すように、`spec.postgresql.pg_hba` のリスト項目として `pg_hba` 行が追加されます。

```
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、安全なチャンネル（`hostssl`）を介してアクセスできるようにしています。

LDAP 構成

クラスター仕様の `postgres` セクションには、`pg_hba.conf` ファイルに追加されるルールに変換する LDAP 構成を定義するために使用できるオプションの `ldap` セクションがあります。

これは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`、`bindDN`、およびシークレットである `ldaptran_9` を指定する必要がある `search+bind` モード。さらに、`search+bind` モードには、`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、デフォルトの `uid` が使用されます。

さらに、どちらのモードでも、`ldapscheme` の `scheme` と `port` を指定できます。ただし、スキームもポートも必要ありません。

`search+bind` で入力されたこのセクションは次のようになります。

```
postgresql:
  parameters:
    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` 構成オプション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。

CloudNativePG では、次の 2 つの値のいずれかに制限することをお勧めします。

- `posix` : `shm_open` 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。詳細はこちらの

thread from the `pgsql-general` mailing list <https://www.postgresql.org/message-id/CA%2BhUKGJOj7qzDLxeFPVvto8YEWop6FSQoTYPO9Z6Ee%3Di-nPS_Q%40mail.gmail.com>`\_\_` を参照してください。

POSIX 共有メモリ

オペレーターが `/dev/shm` の下に `shm` と呼ばれるメモリバインドの ``EmptyDir`` ボリュームを自動的にマウントすることを考えると、ほとんどの場合、`posix` のデフォルト設定で十分です。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
```

次の出力が表示されます。

```
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、次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、非常に多くの 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 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、構成パラダイムの規則に従います。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-controller-manager` と呼ばれる Kubernetes Deployment として `cnpg-system` 名前空間にインストールされます。

注釈: 以下の例では、オペレーター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` を使用してカスタマイズできます。

重要: 構成の `ConfigMap` / `Secret` への変更は、オペレーターによって自動的に検出されません。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。

4.17.1 利用可能なオプション

オペレーターは、`ConfigMap` / `Secret` で定義される次の環境変数を探します。

名前 | 説明 --- | --- - `INHERITED_ANNOTATIONS` | Cluster メタデータで定義されると、ポッド `INHERITED_LABELS` | を含むすべての生成されたリソースに継承されるアノテーション名のリスト Cluster メタデータで定義されると、ポッド `PULL_SECRET_NAME` | を含むすべての生成されたリソースによって継承されるラベル名のリストオペレーターの名前空間で定義され、イメージのダウンロードに使用される追加のプルシークレットの名前 PVC がサイズ変更状態でスタックしている場合、Postgres Pod を削除できます。この機能は主に Azure 環境向けです（デフォルトは `false`） `true` に設定すると、オペレーターの更新後のインスタ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 `MONITORING_QUERIES_CONFIGMAP`）作成されたすべてのクラスターに適用されるデフォルトのクエリー（キー `queries` で指定）を含むオペレーターの名前空間内の `ConfigMap` の名前作成されたすべてのクラスターに適用されるデフォルトのクエリ（キー `queries` で指定）を持つオペレーターの名前空間内の `Secret` の名前

`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ています。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 です。

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ました。バージョン「1.11.0」から、動作が以前の説明に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で作成されたプルシークレットは使用されません。

4.17.2 オペレーター構成マップの定義

以下の例では、後で展開される 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/アノテーション名を定義し、`in-place updates for the instance manager` を有効にすることにより、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:
  INHERITED_ANNOTATIONS: categories
  INHERITED_LABELS: environment, workload, app
  ENABLE_INSTANCE_MANAGER_INPLACE_UPDATES: true
```

4.17.3 オペレーターシークレットの定義

以下の例では、後で展開される 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/アノテーション名を定義し、`in-place updates for the instance manager` を有効にすることにより、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
```

(次のページに続く)

(前のページからの続き)

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

4.17.4 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継承されます。

4.17.5 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.
```

(次のページに続く)

```
- `./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。

次に、編集ページでコンテナ引数を下にスクロールし、`--pprof-server=true` を追加します。例：`edb_notranlate_6` 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、次を実行してコンテナ内で実行できます。

```
kubectl exec -ti -n cnpg-system <pod name> -- bash
```

内部で実行：

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

4.18 ストレージ

ストレージは、データベースの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、一貫性と耐久性が保証されている必要があります。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のコンテキストでも有効です。

重要： 動的にプロビジョニングされるストレージに関しては、Kubernetes には独自の特性があります。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。

ストレージにアクセスするには、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ています（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問題が発生します。これらは、複数のアプリケーションとの I/O 競合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、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) のワークロードにより適したシェアードナッシングアーキテクチャを有効にします。

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。

4.18.1 CloudNativePG のベンチマーク

EDB は、データベースを本番環境にデプロイする前に、制御された Kubernetes 環境で CloudNativePG をベンチマークするためのオープンソースのガイドラインと Helm チャートのセットである `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、基に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、ランダム書き込みなどのデータベースワークロードに関連するメトリック
- 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てデータベースのパフォーマンスを測定する：`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する前に**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、データの分散も同様にパフォーマンスに影響を与える可能性があります。シーケンシャルの読み取りまたは書き込みの理論上の最大スループットを知ることは、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のコンテキストでは、**システムを知り、ベンチマークします。**

4.18.2 永続ボリュームの要求

オペレーターは、PGDATA を保存することを目的として、各 PostgreSQL インスタンスの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。

4.18.3 ストレージクラスを介した設定

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。ターゲットの Kubernetes クラスターにデフォルトのストレージクラスがない場合、または PVC を既知のストレージクラスで満たす必要がある場合でも、カスタムリソースに設定できます。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgresql-storage-class
spec:
  instances: 3
  storage:
    storageClass: standard
    size: 1Gi
```

重要: CloudNativePG は、ストレージクラスに依存しないように設計されています。いつものように、本稼働に移す前に、制御された環境でストレージクラスを適切にベンチマークすることをお勧めします。

4.18.4 PVC テンプレートを介した設定

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します。

```
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

```

4.18.5 ボリューム拡張

Kubernetes は、デフォルトで有効になっている `expanding PVCs` を許可する API を公開しますが、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ります。

```

$ kubectl get storageclass -o jsonpath={$.allowVolumeExpansion} premium-storage
true

```

ボリューム拡張 Kubernetes 機能の使用

ストレージクラスがボリューム拡張をサポートしている場合、`Cluster` のサイズ要件を変更できます。オペレーターは変更をすべての `PVC` に適用します。

`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。

続行するための最善の方法は、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。

これは、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、手動でロールアウトを実行する必要があることを意味します。最初に、ディスクにバインドされた PVC を使用して Pod をホストするノードを遮断します。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 に再接続できません。

最初のステップは、次のように新しいサイズ、たとえば「2Gi」を適用する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、最初にレプリカを続行できます。たとえば、cluster-example-3 Pod をホストする kubernetes ノードの暗号化から始めます。

```
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 および Ready 状態になるまで待ちます。

```
kubectl get pods -w cluster-example-3
cluster-example-3  0/1      Init:0/1    0          12m
cluster-example-3  1/1      Running    0          12m
```

次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}
```

したがって、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してプロモートした後、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ための ``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 を割り当ててから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。

その際、次の例のように、`resizeInUseVolumes` フラグを無効にして、オペレーターが既存の PVC を変更しないようにする必要があります。

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

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

クラスタ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 とすべてのポッドを再作成する必要があります。次の例のように、3 つのレプリカを持つクラスターがあるとします。

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
cluster-example-1	1/1	Running	0	2m37s
cluster-example-2	1/1	Running	0	2m22s
cluster-example-3	1/1	Running	0	2m10s

別の PVC を使用してクラスターを再作成するには、クラスター定義を編集して `resizeInUseVolumes` を無効にしてから、別の PVC にすべてのインスタンスを再作成します。

例として、`cluster-example-3` のストレージを再作成するには：

```
$ kubectl delete pvc/cluster-example-3 pod/cluster-example-3
```

それが終わったら、オペレーターはサイズを変更した 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-v2bfg	0/1	Completed	0	17s
cluster-example-4	1/1	Running	0	10s

4.19 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、このようなリソースとオブジェクトは、**ラベル** と **アノテーション** を介してリンクして関係付けることができます。

要するに：

- アノテーションは、外部ツールとの統合を促進することを目的として、追加の非識別情報を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 のネイティブセクター機能を介して照会するために使用されます

CloudNativePG 展開で使用する 1 つ以上のラベル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やアノテーションを定義すると、それ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 が採用する手法です。

4.19.1 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する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に、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/** などの戦略を採用できることに注意してください。

4.19.2 クラスターのメタデータの定義

クラスターを定義する場合、リソースをデプロイする前に、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
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、ポッドにラベルが正しく設定されていることを確認できます。

```
kubectl get pods --show-labels
```

4.19.3 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ても、オペレーターは関連するリソースから自動的に削除しません。

4.20 モニタリング

4.20.1 インスタンスのモニタリング

オペレーターは、PostgreSQL インスタンスごとに、ポート 9187 で `metrics` という 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トを提供します。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 は既に ``default-monitoring`` と呼ばれる ``ConfigMap`` に by default a set of predefined metrics をインストールしています。

メトリックには次のようにアクセスできます。

```
curl http://<pod_ip>:9187/metrics
```

PostgreSQL で実行されるすべての監視クエリは次のとおりです。

- トランザクション的にアトミック（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

pg_monitor ロールの詳細については、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。

デフォルトでは、クエリは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 `initdb.database`、または指定されていない場合は `app` に対してデフォルトで実行されます
- `recovery` を使用：クエリは `recovery.database`、または指定されていない場合は `postgres` に対してデフォルトで実行されます
- `pg_basebackup` を使用：クエリは `pg_basebackup.database`、または指定されていない場合は `postgres` に対してデフォルトで実行されます

`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、特定のユーザー定義メトリックのデフォルトのデータベースを常に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身自身で `.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、以下で説明します。

特定のクラスターの PodMonitor を手動でデプロイするには、次のように定義し、必要に応じて変更します。

重要: 一意の名前と、正しいクラスターの名前空間とラベルを使用して、上の例を変更してください（`cluster-example` を使用しています）。

事前定義されたメトリックセット

すべての PostgreSQL インスタンスエクスポートは、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数、予想および実際に観測された値-レプリカクラスターモードが有効かどうかを示すフラグまたは `disabled` - 手動スイッチオーバーが必要かどうかを示すフラグ
- `go_*` で始まる、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
```

(次のページに続く)

```

# 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"} 7
cnpg_collector_pg_wal{value="size"} 1.17440512e+08

# HELP cnpg_collector_pg_wal_archive_status Number of WAL segments in the /var/lib/
↳postgresql/data/pgdata/pg_wal/archive_status directory (ready, done)
# TYPE cnpg_collector_pg_wal_archive_status gauge
cnpg_collector_pg_wal_archive_status{value="done"} 6
cnpg_collector_pg_wal_archive_status{value="ready"} 0

# HELP cnpg_collector_replica_mode 1 if the cluster is in replica mode, 0 otherwise
# TYPE cnpg_collector_replica_mode gauge
cnpg_collector_replica_mode 0

# HELP cnpg_collector_sync_replicas Number of requested synchronous replicas_
↳(synchronous_standby_names)
# TYPE cnpg_collector_sync_replicas gauge
cnpg_collector_sync_replicas{value="expected"} 0
cnpg_collector_sync_replicas{value="max"} 0
cnpg_collector_sync_replicas{value="min"} 0
cnpg_collector_sync_replicas{value="observed"} 0

# HELP cnpg_collector_up 1 if PostgreSQL is up, 0 otherwise.
# TYPE cnpg_collector_up gauge
cnpg_collector_up{cluster="cluster-example"} 1

# HELP cnpg_collector_postgres_version Postgres version
# TYPE cnpg_collector_postgres_version gauge
cnpg_collector_postgres_version{cluster="cluster-example",full="13.4.0"} 13.4

# HELP cnpg_collector_first_recoverability_point The first point of recoverability_
↳for the cluster as a unix timestamp

```

(前のページからの続き)

```

# TYPE cnpg_collector_first_recoverability_point gauge
cnpg_collector_first_recoverability_point 1.63238406e+09

# HELP cnpg_collector_lo_pages Estimated number of pages in the pg_largeobject table
# TYPE cnpg_collector_lo_pages gauge
cnpg_collector_lo_pages{datname="app"} 0
cnpg_collector_lo_pages{datname="postgres"} 78

# HELP go_gc_duration_seconds A summary of the pause duration of garbage collection.
→cycles.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 5.01e-05
go_gc_duration_seconds{quantile="0.25"} 7.27e-05
go_gc_duration_seconds{quantile="0.5"} 0.0001748
go_gc_duration_seconds{quantile="0.75"} 0.0002959
go_gc_duration_seconds{quantile="1"} 0.0012776
go_gc_duration_seconds_sum 0.0035741
go_gc_duration_seconds_count 13

# HELP go_goroutines Number of goroutines that currently exist.
# TYPE go_goroutines gauge
go_goroutines 25

# HELP go_info Information about the Go environment.
# TYPE go_info gauge
go_info{version="go1.17.1"} 1

# HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.
# TYPE go_memstats_alloc_bytes gauge
go_memstats_alloc_bytes 4.493744e+06

# HELP go_memstats_alloc_bytes_total Total number of bytes allocated, even if freed.
# TYPE go_memstats_alloc_bytes_total counter
go_memstats_alloc_bytes_total 2.1698216e+07

# HELP go_memstats_buck_hash_sys_bytes Number of bytes used by the profiling bucket.
→hash table.
# TYPE go_memstats_buck_hash_sys_bytes gauge
go_memstats_buck_hash_sys_bytes 1.456234e+06

# HELP go_memstats_frees_total Total number of frees.
# TYPE go_memstats_frees_total counter
go_memstats_frees_total 172118

# HELP go_memstats_gc_cpu_fraction The fraction of this programs available CPU time.
→used by the GC since the program started.

```

(次のページに続く)

```
# 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.
# 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
```

(前のページからの続き)

```
# 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.
# 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中にあります。

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 エクスポートの `queries.yaml file` に触発されています。

カスタムメトリックは、次の例のように、`.spec.monitoring.customQueriesConfigMap` または `customQueriesSecret` セクションの下に `Cluster` 定義で作成された `Configmap /Secret` を参照することにより、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ているキーを指定する `ConfigMap /Secret` 参照のリストが含まれています。参照されるリソースは、クラスターリソースと同じ名前空間に作成する必要があることに注意してください。

注釈: `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` でラベルを追加できます。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。

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。

ユーザー定義のメトリックの例

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"
```

基本的な監視クエリのリストは、``cnpg-basic-monitoring.yaml`` file `<./samples/cnpg-basic-monitoring.yaml>`_``.

複数のデータベースで実行されるユーザー定義のメトリックの例

`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 dataallowconn 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マッチングすることにより、ターゲットデータベースのリストを展開します。

注釈: * のキャラクターは 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`` の値をクォート ("*") 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、前述のクエリでは返されません）。

```

some_query:
  query: |
    SELECT
      current_database() as datname,
      count(*) as rows
    FROM some_table
  metrics:
    - datname:
        usage: "LABEL"
        description: "Name of current database"
    - rows:
        usage: "GAUGE"
        description: "number of rows"
  target_databases:
    - "*"
    - "template1"

```

上記の例では、次のメトリックが生成されます（データベースが存在する場合）。

```

cnpg_some_query_rows{datname="albert"} 2
cnpg_some_query_rows{datname="bb"} 5
cnpg_some_query_rows{datname="freddie"} 10
cnpg_some_query_rows{datname="template1"} 7
cnpg_some_query_rows{datname="postgres"} 42

```

ユーザー定義メトリックの構造

すべてのカスタムクエリには、次の基本構造があります。

```

<MetricName>:
  query: "<SQLQuery>"
  metrics:
    - <ColumnName>:
        usage: "<MetricType>"
        description: "<MetricDescription>"

```

使用可能なすべてのフィールドの簡単な説明は次のとおりです。

- **<MetricName>** : Prometheus 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 クエリー - **primary** : プライマリインスタンスでのみクエリーを実行するかどうか - **master** : **primary** と同じ (Prometheus PostgreSQL エクスポート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 "**>=10.0.0**" または "**>=12.0.0 <=14.4.0**") - **target_databases** : **query** を実行するデータベースのリスト、または自動検出を有効にする shell-like pattern。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。

Column Usage Label	Description
<code>DISCARD</code>	この列は無視する必要があります
<code>LABEL</code>	この列をラベルとして使用する
<code>COUNTER</code>	この列をカウンターとして使用する
<code>GAUGE</code>	この列をゲージとして使用する
<code>MAPPEDMETRIC</code>	この列を指定されたテキスト値のマッピングで使します
<code>DURATION</code>	この列をテキスト期間として使します (ミリ秒単位)
<code>HISTOGRAM</code>	この列をヒストグラムとして使用する

詳細については、Prometheus のドキュメントの [Metric Types](#) にアクセスしてください。

ユーザー定義のメトリックの出力

カスタム定義のメトリックは、Prometheus エクスポーターエンドポイント (`:9187/metrics`) によって次の形式で返されます。

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

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

上記の `pg_replication` の例を考慮すると、エクスポーターのエンドポイントは呼び出されると次の出力を返します。

```
# HELP cnpg_pg_replication_in_recovery Whether the instance is in recovery
# TYPE cnpg_pg_replication_in_recovery gauge
cnpg_pg_replication_in_recovery 0
# HELP cnpg_pg_replication_lag Replication lag behind primary in seconds
# TYPE cnpg_pg_replication_lag gauge
cnpg_pg_replication_lag 0
# HELP cnpg_pg_replication_streaming_replicas Number of streaming replicas connected_
→to the instance
# TYPE cnpg_pg_replication_streaming_replicas gauge
cnpg_pg_replication_streaming_replicas 2
# HELP cnpg_pg_replication_is_wal_receiver_up Whether the instance wal_receiver is up
# TYPE cnpg_pg_replication_is_wal_receiver_up gauge
cnpg_pg_replication_is_wal_receiver_up 0
```

メトリックのデフォルトセット

オペレーターは、オペレーターの名前空間内の `ConfigMap` または `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、デフォルトでオペレーター構成で `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、次のようにします。

- 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。オペレーターの `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 `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で `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。

4.20.2 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。

メトリックには次のようにアクセスできます。

```
curl http://<pod_ip>:8080/metrics
```

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

オペレーターのデプロイメントは、次の **PodMonitor** リソース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。

```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: cnpg-controller-manager
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: cloudnative-pg
  podMetricsEndpoints:
    - port: metrics
```

4.21 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** : 実録 (logger の種類によって構造が異なります)
- **logging_podName** : ログが生成されたポッド

警告: ログの長期保存と管理はオペレーターの権限の範囲外であり、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](#) のドキュメントを参照してください。

4.21.1 オペレーターログ

ログレベルは、オプション **logLevel** を使用してクラスター仕様で指定でき、**error**、**warning**、**info** (デフォルト)、**debug** または **trace** のいずれかに設定できます。

現時点では、ログレベルはインスタンスの起動時にのみ設定でき、実行時に変更することはできません。クラスターの起動後にクラスター仕様で値が変更された場合、これは古いポッドではなく新しいポッドで有効になります。

4.21.2 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",
  "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。

4.21.3 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。

必要なことは、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して管理します。

オペレーターは、クラスター内の使用可能なすべてのデータベース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。

これは、要求された構成で `pgaudit` が有効になる PostgreSQL 13 Cluster デプロイメントの例です。

```
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` に設定されています
- `.msg` は `record` に設定されています
- `.record` には、`logging_collector` ログと同様に、解析されたレコード全体が含まれています。

以下の例を参照してください。

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

4.21.4 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。

4.22 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` を設定するには、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: クラスターで使用されるすべてのシークレットとその有効期限は、クラスターのステータスで確認できます。

CloudNativePG は TLS 証明書に関して非常に柔軟で、主に 2 つのモードで動作します。

1. ****operator managed****

: 証明書は完全に自動化された方法でオペレーターによって内部的に管理され、CloudNativePG によって作成された CA を使用して署名されます

2. ****user provided****

: 証明書はオペレーターの外部で生成され、クラスター定義にシークレットとして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 の外部で生成されるハイブリッドアプローチを選択することもできます。

4.22.1 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、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ているため、クライアントが ID を検証して安全に接続できます。

サーバーの代替 DNS 名

デフォルトの名前に加えて、生成されたサーバー TLS シークレットの一部になる 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。パブリック部分は、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、タイプ `kubernetes.io/tls` 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。

4.22.2 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 `kubernetes.io/tls` タイプの 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` でラベルを追加できます。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。

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 証明書の秘密キー。

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:
```

(次のページに続く)

```

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: 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、必要なすべてのファイルを含む cert-manager によって作成され、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 cluster-example-cert-manager.yaml 配置マニフェストで管理する完全な例を見つけることができます。

クライアント証明書

必要に応じて、2つのクライアント証明書を提供し、[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 `kubernetes.io/tls` タイプの 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される 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ません。

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` でラベルを追加できます。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。

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: ""
```

(次のページに続く)

```

- -
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、必要なすべてのファイルを含む cert-manager によって作成され、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:
    clientCASecret: my-postgres-client-cert
    replicationTLSSecret: my-postgres-client-cert
  storage:
    size: 1Gi

```

cert-manager を使用してサーバーとクライアントの両方の CA と証明書を管理する完全な例は、cluster-example-cert-manager.yaml 配置マニフェストにあります。

4.23 クライアント TLS / SSL 接続

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 を使用するように設計されています。CNPG オペレーターを使用して作成されたクラスターには、TLS クライアント証明書を作成して署名するための認証局 (CA) が付属しています。kubectl の cnpg プラグインを介して、パスワードを使用する代わりにユーザーの認証に使用できる新しい TLS クライアント証明書を発行できます。

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。これは、

```
cluster-example.yaml
```

デプロイメントマニフェストを使用してクラスターをインストールしたことを前提としています。構成パラダイムの規約によれば、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（この規約は `bootstrap` セクションの `initdb` 構成で変更できます）。

4.23.1 新しい証明書の発行

次のように、`cluster-example` PostgreSQL クラスターで `app` ユーザーの証明書を作成できます。

```
kubectl cnpg certificate cluster-app \
  --cnpg-cluster cluster-example \
  --cnpg-user app
```

次を使用して証明書を確認できます。

```
kubectl get secret cluster-app \
  -o jsonpath="{.data[tls\.crt]}" \
  | base64 -d | openssl x509 -text -noout \
  | head -n 11
```

出力：

```
Certificate:
Data:
  Version: 3 (0x2)
  Serial Number:
    5d:e1:72:8a:39:9f:ce:51:19:9d:21:ff:1e:4b:24:5d
  Signature Algorithm: ecdsa-with-SHA256
  Issuer: OU = default, CN = cluster-example
  Validity
    Not Before: Mar 22 10:22:14 2021 GMT
    Not After : Mar 22 10:22:14 2022 GMT
  Subject: CN = app
```

ご覧のとおり、TLS クライアント証明書はデフォルトで 90 日間の有効性と、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。

4.23.2 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:
        - 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

```

(次のページに続く)

(前のページからの続き)

```
- 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`: 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した TLS 認証局証明書

これらは、`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 PostgreSQL データベースへの TLS 暗号化接続を確立するために必要な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` を暗号化素材の実際の場所にポイントできます。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 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、準備ができています。

Pod のコンテナ内で対話型の `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 14.4 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.3.1 20191121 (Red Hat
8.3.1-5), 64-bit
(1 row)
```

4.24 アプリケーションから接続する

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、

`:ref: microservice` タイプ<microservice タイプ>`` 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 インスタンスに直接接続しないことを強くお勧めします。後者はクラスターの有効期間中に変更される可能性があります。

これらのサービスは、次の方法でアプリケーションで使用できます。

- DNS 解決

*環境変数

PostgreSQL に接続するための資格情報には、オペレーターによって生成されたシークレットを使用できます。

警告: オペレーターは、`[cluster name]-any` という名前の別のサービスを作成します。そのサービスは、PostgreSQL インスタンスの検出を管理するために内部で使用されます。アプリケーションで直接使用することは想定されていません。

4.24.1 DNS 解決

Kubernetes DNS サービスを使用して、特定のサーバーをポイントできます。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、推奨される検出方法です。

4.24.2 環境変数

PostgreSQL クラスターが含まれるのと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ポイントするサービスの 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

4.24.3 秘密

PostgreSQL オペレーターは、デプロイするすべての PostgreSQL クラスターに対して 2 つの `basic-auth` タイプのシークレットを生成します。

- `[cluster name]-superuser`
- `[cluster name]-app`

シークレットには、それぞれ `postgres` ユーザーとデータベースの所有者のユーザー名、パスワード、および作業中の `.pgpass file` が含まれています。

`-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ります。

`-superuser` は管理目的でのみ使用されることになっています。

4.25 接続プーリング

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をネイティブでサポートしています。

簡単に言うと、CloudNativePG の `Pooler` は、アプリケーションと PostgreSQL サービス（`rw` サービスなど）の間にある `PgBouncer` ポッドのデプロイであり、別個のスケラブルで構成可能な可用性の高いデータベースアクセスレイヤーを作成します。

4.25.1 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。アプリケーションは、PostgreSQL プライマリサービスに直接接続する代わりに、PgBouncer の同等のサービスに接続できるようになり、既存の接続を再利用して、PostgreSQL 側のパフォーマンスとリソース管理を向上させます。

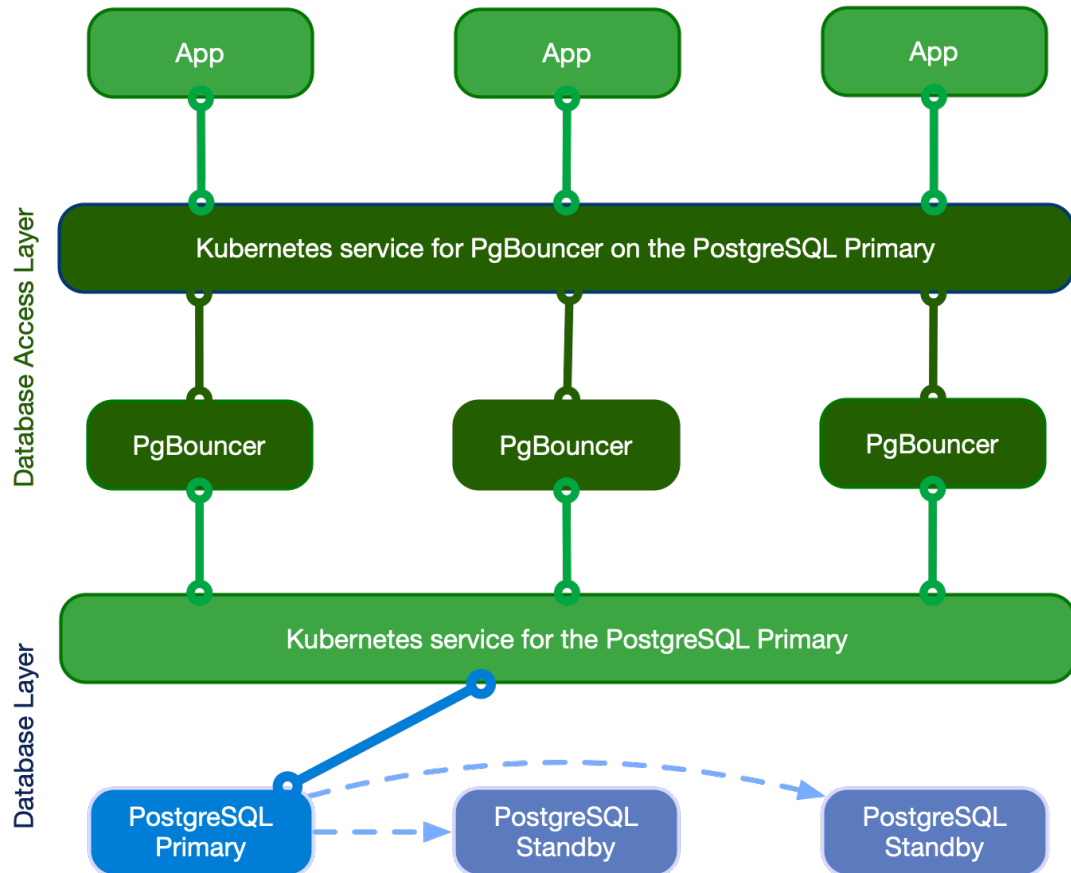


図 8: Applications writing to the single primary via PgBouncer

4.25.2 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:
    name: cluster-example
```

(次のページに続く)

(前のページからの続き)

```
instances: 3
type: rw
pgbouncer:
  poolMode: session
  parameters:
    max_client_conn: "1000"
    default_pool_size: "10"
```

重要: プーラー名は、同じ名前空間内のクラスター名と一致しないでください。

これは、`cluster-example` と呼ばれる Postgres Cluster リソースに厳密に関連付けられ、読み取り/書き込みサービス (`rw`、したがって `cluster-example-rw`) を指す `pooler-example-rw` と呼ばれる新しい Pooler リソースを作成します。

Pooler は、Postgres クラスターと同じ名前空間にある必要があります。これは、

[latest stable image of PgBouncer](#) を実行し、`session`

で構成され、それぞれ最大 1000 個の接続を受け入れます。デフォルトのプールサイズは PostgreSQL 向けです。

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は PostgreSQL サーバーに中継されます（["Section [databases]" in PgBouncer's documentation](<https://www.pgbouncer.org/config.html#section-databases>) を参照してください）。

さらに、CloudNativePG は、PgBouncer で使用される構成ファイルを含むプーラーと同じ名前のシークレットを自動的に作成します。

4.25.3 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展開できます。

注意すべき重要なことは、Cluster と Pooler リソースのライフサイクルは現在独立していることです。Cluster の削除は、Pooler の自動削除を意味せず、その逆も同様です。

重要: *Pooler* がどのように機能するかを理解したので、可能なアーキテクチャの点で完全に自由です。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。

4.25.4 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身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。

4.25.5 認証

パスワードベースの認証は、CloudNativePG の PgBouncer のクライアントでサポートされている唯一の方法です。

内部的には、実装は PgBouncer の `auth_user` および `auth_query` オプションに依存しています。具体的には、オペレーター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、上記をすべて削除します

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。

最初に、ロールを作成する必要があります。

```
CREATE ROLE cnpg_pooler_pgouncer WITH LOGIN;
```

次に、アプリケーションデータベースごとに、cnpg_pooler_pgouncer に接続する権限を付与します。

```
GRANT CONNECT ON DATABASE { database name here } TO cnpg_pooler_pgouncer;
```

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ファンクションを作成します。

```
CREATE OR REPLACE FUNCTION user_search(uname TEXT)
  RETURNS TABLE (username name, passwd text)
  LANGUAGE sql SECURITY DEFINER AS
  SELECT username, passwd FROM pg_shadow WHERE username=$1;;

REVOKE ALL ON FUNCTION user_search(text)
  FROM public;

GRANT EXECUTE ON FUNCTION user_search(text)
  TO cnpg_pooler_pgouncer;
```

4.25.6 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ルールとアンチアフィニティルールの微調整など、ポッドを自由に構成できます。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。

ここでは、PodAntiAffinity を指定するプーラーの例：

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

(次のページに続く)

```

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したときに `kubernetes api-server` は次のエラーを返します。

ここでは、リソースを設定し、使用す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:
      labels:
        app: pooler
    spec:
      containers:
        - name: pgbouncer

```

(前のページからの続き)

```

image: my-pgbouncer:latest
resources:
  requests:
    cpu: 0.1
    memory: 100Mi
  limits:
    cpu: 0.5
    memory: 500Mi

```

4.25.7 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、PgBouncer を実行している利用可能なポッドにクライアントがランダムに分散されます。

警告: インフラストラクチャが高遅延の複数のアベイラビリティゾーンにまたがる場合は、ネットワークホップに注意してください。たとえば、ゾーン 2 で実行されているアプリケーションが、ゾーン 3 で実行されている PgBouncer に接続し、ゾーン 1 の PostgreSQL プライマリをポイントしている場合を考えます。

4.25.8 PgBouncer の構成オプション

オペレーターは [configuration options for PgBouncer](#) のほとんどを管理するため、それらのサブセットのみを変更できます。

警告: オペレーターは各オプションを検証しないため、各オプションの値を正しく設定する必要があります。

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 で必要
- `log_connections`
- `log_disconnections`
- `log_pooler_errors`
- `log_stats` : デフォルトでは無効 (0)
)、以下の *MonitoringConfiguration* セクションで説明するように Prometheus エクスポートによって統計が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`
- `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応じ、すべての PgBouncer インスタンスはサービスを中断せずに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。

4.25.9 モニタリング

Pooler の PgBouncer 実装には、デフォルトの Prometheus エクスポートが付属しています。

- SHOW LISTS (プレフィックス: `cnpg_pgbouncer_lists`)
- SHOW POOLS (プレフィックス: `cnpg_pgbouncer_pools`)
- SHOW STATS (プレフィックス: `cnpg_pgbouncer_stats`)

CloudNativePG インスタンスと同様に、エクスポートは 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ト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
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
```

(次のページに続く)

(前のページからの続き)

```

cnpg_pgrounder_lists_dns_pending 0

# HELP cnpg_pgrounder_lists_dns_queries Count of in-flight DNS queries.
# TYPE cnpg_pgrounder_lists_dns_queries gauge
cnpg_pgrounder_lists_dns_queries 0

# HELP cnpg_pgrounder_lists_dns_zones Count of DNS zones in the cache.
# TYPE cnpg_pgrounder_lists_dns_zones gauge
cnpg_pgrounder_lists_dns_zones 0

# HELP cnpg_pgrounder_lists_free_clients Count of free clients.
# TYPE cnpg_pgrounder_lists_free_clients gauge
cnpg_pgrounder_lists_free_clients 49

# HELP cnpg_pgrounder_lists_free_servers Count of free servers.
# TYPE cnpg_pgrounder_lists_free_servers gauge
cnpg_pgrounder_lists_free_servers 0

# HELP cnpg_pgrounder_lists_login_clients Count of clients in login state.
# TYPE cnpg_pgrounder_lists_login_clients gauge
cnpg_pgrounder_lists_login_clients 0

# HELP cnpg_pgrounder_lists_pools Count of pools.
# TYPE cnpg_pgrounder_lists_pools gauge
cnpg_pgrounder_lists_pools 1

# HELP cnpg_pgrounder_lists_used_clients Count of used clients.
# TYPE cnpg_pgrounder_lists_used_clients gauge
cnpg_pgrounder_lists_used_clients 1

# HELP cnpg_pgrounder_lists_used_servers Count of used servers.
# TYPE cnpg_pgrounder_lists_used_servers gauge
cnpg_pgrounder_lists_used_servers 0

# HELP cnpg_pgrounder_lists_users Count of users.
# TYPE cnpg_pgrounder_lists_users gauge
cnpg_pgrounder_lists_users 2

# HELP cnpg_pgrounder_pools_cl_active Client connections that are linked to server.
→connection and can process queries.
# TYPE cnpg_pgrounder_pools_cl_active gauge
cnpg_pgrounder_pools_cl_active{database="pgrounder",user="pgrounder"} 1

# HELP cnpg_pgrounder_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_pgrounder_pools_cl_cancel_req gauge
cnpg_pgrounder_pools_cl_cancel_req{database="pgrounder",user="pgrounder"} 0

# HELP cnpg_pgrounder_pools_cl_waiting Client connections that have sent queries but
→have not yet got a server connection.
# TYPE cnpg_pgrounder_pools_cl_waiting gauge
cnpg_pgrounder_pools_cl_waiting{database="pgrounder",user="pgrounder"} 0

# HELP cnpg_pgrounder_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_pgrounder_pools_maxwait gauge
cnpg_pgrounder_pools_maxwait{database="pgrounder",user="pgrounder"} 0

# HELP cnpg_pgrounder_pools_maxwait_us Microsecond part of the maximum waiting time.
# TYPE cnpg_pgrounder_pools_maxwait_us gauge
cnpg_pgrounder_pools_maxwait_us{database="pgrounder",user="pgrounder"} 0

# HELP cnpg_pgrounder_pools_pool_mode The pooling mode in use. 1 for session, 2 for
→transaction, 3 for statement, -1 if unknown
# TYPE cnpg_pgrounder_pools_pool_mode gauge
cnpg_pgrounder_pools_pool_mode{database="pgrounder",user="pgrounder"} 3

# HELP cnpg_pgrounder_pools_sv_active Server connections that are linked to a client.
# TYPE cnpg_pgrounder_pools_sv_active gauge
cnpg_pgrounder_pools_sv_active{database="pgrounder",user="pgrounder"} 0

# HELP cnpg_pgrounder_pools_sv_idle Server connections that are unused and
→immediately usable for client queries.
# TYPE cnpg_pgrounder_pools_sv_idle gauge
cnpg_pgrounder_pools_sv_idle{database="pgrounder",user="pgrounder"} 0

# HELP cnpg_pgrounder_pools_sv_login Server connections currently in the process of
→logging in.
# TYPE cnpg_pgrounder_pools_sv_login gauge
cnpg_pgrounder_pools_sv_login{database="pgrounder",user="pgrounder"} 0

# HELP cnpg_pgrounder_pools_sv_tested Server connections that are currently running
→either server_reset_query or server_check_query.
# TYPE cnpg_pgrounder_pools_sv_tested gauge
cnpg_pgrounder_pools_sv_tested{database="pgrounder",user="pgrounder"} 0

# HELP cnpg_pgrounder_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.

```

(次のページに続く)

```

# TYPE cnpg_pgouncer_pools_sv_used gauge
cnpg_pgouncer_pools_sv_used{database="pgbouncer",user="pgbouncer"} 0

# HELP cnpg_pgouncer_stats_avg_query_count Average queries per second in last stat_
→period.
# TYPE cnpg_pgouncer_stats_avg_query_count gauge
cnpg_pgouncer_stats_avg_query_count{database="pgbouncer"} 1

# HELP cnpg_pgouncer_stats_avg_query_time Average query duration, in microseconds.
# TYPE cnpg_pgouncer_stats_avg_query_time gauge
cnpg_pgouncer_stats_avg_query_time{database="pgbouncer"} 0

# HELP cnpg_pgouncer_stats_avg_recv Average received (from clients) bytes per_
→second.
# TYPE cnpg_pgouncer_stats_avg_recv gauge
cnpg_pgouncer_stats_avg_recv{database="pgbouncer"} 0

# HELP cnpg_pgouncer_stats_avg_sent Average sent (to clients) bytes per second.
# TYPE cnpg_pgouncer_stats_avg_sent gauge
cnpg_pgouncer_stats_avg_sent{database="pgbouncer"} 0

# HELP cnpg_pgouncer_stats_avg_wait_time Time spent by clients waiting for a server,
in microseconds (average per second).
# TYPE cnpg_pgouncer_stats_avg_wait_time gauge
cnpg_pgouncer_stats_avg_wait_time{database="pgbouncer"} 0

# HELP cnpg_pgouncer_stats_avg_xact_count Average transactions per second in last_
→stat period.
# TYPE cnpg_pgouncer_stats_avg_xact_count gauge
cnpg_pgouncer_stats_avg_xact_count{database="pgbouncer"} 1

# HELP cnpg_pgouncer_stats_avg_xact_time Average transaction duration, in_
→microseconds.
# TYPE cnpg_pgouncer_stats_avg_xact_time gauge
cnpg_pgouncer_stats_avg_xact_time{database="pgbouncer"} 0

# HELP cnpg_pgouncer_stats_total_query_count Total number of SQL queries pooled by_
→pgbouncer.
# TYPE cnpg_pgouncer_stats_total_query_count gauge
cnpg_pgouncer_stats_total_query_count{database="pgbouncer"} 3

# HELP cnpg_pgouncer_stats_total_query_time Total number of microseconds spent by_
→pgbouncer when actively connected to PostgreSQL, executing queries.
# TYPE cnpg_pgouncer_stats_total_query_time gauge
cnpg_pgouncer_stats_total_query_time{database="pgbouncer"} 0

```

(前のページからの続き)

```
# HELP cnpg_pgrounder_stats_total_received Total volume in bytes of network traffic
→received by pgrounder.
# TYPE cnpg_pgrounder_stats_total_received gauge
cnpg_pgrounder_stats_total_received{database="pgrounder"} 0

# HELP cnpg_pgrounder_stats_total_sent Total volume in bytes of network traffic sent
→by pgrounder.
# TYPE cnpg_pgrounder_stats_total_sent gauge
cnpg_pgrounder_stats_total_sent{database="pgrounder"} 0

# HELP cnpg_pgrounder_stats_total_wait_time Time spent by clients waiting for a
→server, in microseconds.
# TYPE cnpg_pgrounder_stats_total_wait_time gauge
cnpg_pgrounder_stats_total_wait_time{database="pgrounder"} 0

# HELP cnpg_pgrounder_stats_total_xact_count Total number of SQL transactions pooled
→by pgrounder.
# TYPE cnpg_pgrounder_stats_total_xact_count gauge
cnpg_pgrounder_stats_total_xact_count{database="pgrounder"} 3

# HELP cnpg_pgrounder_stats_total_xact_time Total number of microseconds spent by
→pgrounder when connected to PostgreSQL in a transaction, either idle in transaction
→or executing queries.
# TYPE cnpg_pgrounder_stats_total_xact_time gauge
cnpg_pgrounder_stats_total_xact_time{database="pgrounder"} 0
```

Clusters のように、*Prometheus* オペレーターの例 を使用している場合は、次の

PodMonitor

を定義することにより、特定のプーラーをスクレイピングするように構成できます。

```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: <POOLER_NAME>
spec:
  selector:
    matchLabels:
      cnpg.io/poolerName: <POOLER_NAME>
  podMetricsEndpoints:
    - port: metrics
```

4.25.10 ログ

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"
  }
}
```

4.25.11 接続の一時停止

Pooler 仕様では、デフォルトで `false` に設定されている `paused` オプションを介して、宣言的構成のみを使用して、PgBouncer の `PAUSE` および `RESUME` コマンドを利用できます。 `true` に設定すると、オペレーターは PgBouncer で `PAUSE` コマンドを内部的に呼び出します。

1. クエリが完了するのを待った後、PostgreSQL サーバーへのすべてのアクティブな接続を閉じます
2. クライアントからの新しい接続を一時停止します

`paused` オプションを `false` に戻すと、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が認識す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、`cnpg` 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、同じ結果を得ることができます。

4.25.12 制限

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は大部分のユースケースに対応していますが、より高度でカスタマイズされたシナリオで PgBouncer が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ます。

4.26 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ことが主な特徴であり、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。

次の図-この機能に関する詳細を含む “[Architecture](#)” section から取得-は、レプリカクラスターで実装できるアーキテクチャの例を示しています。

4.26.1 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）です。

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設定されていることを確認するために、管理およびセキュリティ関連の作業が必要です）
- 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の `barman-cloud-wal-restore` によってプルされる WAL ファイルの回復に Barman Cloud オブジェクトストアを使用します
- 2つのいずれか

実際に外部クラスターを定義するだけです。 `pg_basebackup`（ストリーミング）または `recovery`（オブジェクトストア）を使用して PostgreSQL サーバーのクローンを作成する方法については、

Bootstrap 構成 を参照してください。

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

- `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ソースクラスターに接続してデータを受信するようにします

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るため、分散型の対称アーキテクチャが有効になります。

次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完全に柔軟かつ自由に決定できます。

- 異なるデータセンターにある複数の Kubernetes クラスターにまたがるプライベートクラウド
- 異なるリージョンの複数の Kubernetes クラスターにまたがるパブリッククラウド
- 前の2つのミックス（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

4.26.2 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。

`samples/` サブディレクトリの sample YAML を確認できます。

ソースクラスターを指している `bootstrap` セクションと `replica` セクションに注意してください。

```
bootstrap:
  pg_basebackup:
    source: cluster-example

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

`externalClusters` セクションでは、`connectionParameters` サブセクションのホストに正しい名前空間を使用してください。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。

`samples/` サブディレクトリで sample YAML を確認できます。

ソースクラスターを指している `bootstrap` セクションと `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。

4.26.3 レプリカクラスターで指定されたプライマリを昇格

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つの別個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、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して、ロールの変更を確認できます。

```
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。

4.27 Kubernetes のアップグレード

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整理し、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、クラスターでのメンテナンス操作は、一度に 1 つのノードで実行されます。

1. 更新するノードからワークロードを追い出す (drain)
2. 実際の操作 (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 の自己修復機能の面で予想されるケースですが、一時的に低下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度稼働するまで待機する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**ローカル**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。ポッドによって現在使用され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をスキップできます）。

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。**有効にすると**、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した。**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を使用して、新しい PVC を使用して別のノードにポッドを強制的に再作成し、ポッドと一緒に古い 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ン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場合、`--delete-local-data` オプションを追加する必要があります。恐れる必要はありません。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。

4.27.1 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 が **false** に設定された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、ユーザーがそのようなインスタンスが実行されているノードをドレインしないようにします。

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、プライマリを切り替えます

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 が使用可能になるとすぐに（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期間です。

考えられるアプローチは次のとおりです。

1. 現在のインスタンスが実行されているノードをコーディングします。
2. クラスターを 2 つのインスタンスにスケールアップします。データベースのサイズによっては時間がかかる場合があります。
3. 新しいインスタンスが実行されるとすぐに、現在のプライマリが遮断されたノードで実行されていると、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で、古いプライマリのノードを正常にドレインできます。

4.28 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作成されているはずです。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するとします。Kubernetes インフラストラクチャに移行する場合の一般的なユースケースは、実際に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**です。賢明な回避策は、仮想マシンまたは物理サーバーにある可能性が高いアプリケーションに、同じネットワーク内の Kubernetes クラスター内の PostgreSQL データベースへのアクセスを許可することです。

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できます (包括的なリストについては [Kubernetes documentation](#) を参照)。

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、正常に動作しています
- クラスターに `LoadBalancer` タイプのサービスを作成できます

重要: Ingress は、HTTP および HTTPS トラフィックを公開するためにのみ必要です。NGINX Ingress コントローラーはできますが、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を含む `tcp-services` ConfigMap を作成することです。

```
apiVersion: v1
kind: ConfigMap
```

(次のページに続く)

(前のページからの続き)

```

metadata:
  name: tcp-services
  namespace: ingress-nginx
data:
  5432: default/cluster-example-rw:5432

```

次に、ドキュメントで提案されているように NGINX Ingress Controller をインストールした場合、ingress-nginx サービスが必要です。5432 ポートを公開するには、ingress-nginx サービスに追加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

```

:ref:`cluster-expose-service.yaml` <samples/cluster-expose-service.yaml>` を使用し、
 ➡ ``kubectl`` を使用して適用できます。

警告: このファイルを直接適用すると、Ingress の `ConfigMap` 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。

4.28.1 Minikube でのテスト

Minikube では、次を実行してイングレスコントローラーをセットアップできます。

```
minikube addons enable ingress
```

次に、tcp-service ConfigMap にパッチを適用して、Ingress のポート 5432 の接続をプライマリにリダイレクトします。

```
kubectl patch configmap tcp-services -n kube-system \
  --patch {"data":{"5432":"default/cluster-example-rw:5432"}}
```

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: nginx-ingress-controller
          ports:
            - containerPort: 5432
              hostPort: 5432
```

それを nginx-ingress-controller deployment に適用します。

```
kubectl patch deployment nginx-ingress-controller --patch "$(cat patch.yaml)" -n kube-
→system
```

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
```

4.29 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。

4.29.1 インストールする

次の方法でシステムにプラグインをインストールできます。

```
curl -sSfL \
  https://github.com/cloudnative-pg/cloudnative-pg/raw/main/hack/install-cnpg-plugin.
↪sh | \
  sudo sh -s -- -b /usr/local/bin
```

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

4.29.2 使用する

プラグインをインストールして展開したら、次のように使用を開始できます。

```
kubectl cnpg <command> <args...>
```

ステータス

`status` コマンドは、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、WAL での現在のタイムラインと位置
- **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する各 Postgres インスタンスに関する情報。スタンバイの場合、**Current LSN** フィールドは、リカバリ中にリプレイされた最新のログ先行書き込みの場所（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値が返されます。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` フィールドとは異なる場合があります。

```
kubectl cnpg status sandbox
```

Cluster **in** healthy state

```
Name:                sandbox
Namespace:           default
System ID:           7039966298120953877
PostgreSQL Image:    ghcr.io/cloudnative-pg/postgresql:14.4
Primary instance:    sandbox-2
Instances:           3
Ready instances:     3
Current Write LSN:    3AF/EAFA6168 (Timeline: 8 - WAL File: 00000008000003AF000000075)
```

Continuous Backup status

```
First Point of Recoverability: Not Available
Working WAL archiving:         OK
Last Archived WAL:            00000008000003AE000000079 @ 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
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						

(次のページに続く)

(前のページからの続き)

```

-----
->-----
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 を追加して、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:14.4
Primary instance: sandbox-2
Instances:     3
Ready instances: 3
Current Write LSN: 3B1/61DE3158 (Timeline: 8 - WAL File: 000000080000003B1000000030)

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
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

```

(次のページに続く)

(前のページからの続き)

```

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 = 3cfa683e23fe513afae7c97b50ce0628e0cc634bca8b096517538a9a4428efc

```

PostgreSQL HBA Rules

```

# Grant local access
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_pgouncer 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: 00000008000003B000000001D @ 2021-12-14T10:20:42.
→272815Z
Last Failed WAL: -

```

Streaming Replication status

(次のページに続く)

(前のページからの続き)

Name	Sent LSN	Write LSN	Flush LSN	Replay LSN	Write Lag	
→ Flush Lag		Replay Lag	State	Sync State	Sync Priority	
-----	-----	-----	-----	-----	-----	-
→-----	-----	-----	-----	-----	-----	
sandbox-1	3B1/61E26448	3B1/61DF82F0	3B1/61DF82F0	3B1/61DF82F0	00:00:00.000333	└
→00:00:00.000333	00:00:00.005484	streaming	quorum	1		
sandbox-3	3B1/61E26448	3B1/61E26448	3B1/61DF82F0	3B1/61DF82F0	00:00:00.000756	└
→00:00:00.000756	00:00:00.000756	streaming	quorum	1		
Instances status						
Name	Database Size	Current LSN	Replication role	Status	QoS	Manager└
→Version						
-----	-----	-----	-----	-----	-----	-----
→-----						
sandbox-1		3B1/610204B8	Standby (sync)	OK	Guaranteed	1.11.0
sandbox-2		3B1/61DE3158	Primary	OK	Guaranteed	1.11.0
sandbox-3		3B1/62618470	Standby (sync)	OK	Guaranteed	1.11.0

このコマンドは、yaml および json 形式の出力もサポートしています。

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** できるため、クラスターでメンテナンス作業を開始したり、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ら、**kubectl** を使用して取得できます

```
kubectl get secret cluster-cert
```

そして、次のコマンドを使用して、プレーンテキストで同じ内容を表示します。

```
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の再起動を調整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 などの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]

# 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、キー `cnpg.io/reload` でラベルを追加できます。

リロード

`kubectl cnpg reload` コマンドは、特定のクラスターの調整ループをトリガーするようにオペレーターに要求します。これは、カスタムモニタリングクエリを含む ConfigMap などのクラスター依存オブジェクトに構成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 `true` に、`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、このアクションがリスト内のすべてのクラスターに適用されます。

Kubernetes クラスター内のすべての PostgreSQL をメンテナンスに設定する場合は、次のコマンドを記述するだけです。

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

そして、更新するすべてのクラスターのリストがあります

```
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 ファイルにまとめます。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。

レポート演算子

`operator` サブコマンドは、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オペレーターに要求します。

重要: Secrets と ConfigMaps のすべての機密情報は 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て値を表示します。ご自身の責任でのみ使用してください。これは個人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の収集を有効にできます

- **デプロイメント情報** : オペレーターデプロイメントとオペレーターポッド
- **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
- **ウェブフックサービス** : 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` フラグを使用しない場合、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は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`

```
kubectl cnpg report operator -n <namespace>
```

の結果

```
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
```

(次のページに続く)

(前のページからの続き)

```

inflating: report_operator_<TIMESTAMP>/manifests/validating-webhook-configuration.
→yaml
inflating: report_operator_<TIMESTAMP>/manifests/mutating-webhook-configuration.yaml
inflating: report_operator_<TIMESTAMP>/manifests/webhook-service.yaml
inflating: report_operator_<TIMESTAMP>/manifests/cnpg-ca-secret.yaml
inflating: report_operator_<TIMESTAMP>/manifests/cnpg-webhook-cert.yaml

```

機密情報が 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) オプションを有効にすると、シークレットが表示されます。

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

機密情報を表示しようとしている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、次を収集します。

- **クラスターリソース**：クラスター情報、`kubectl get cluster -o yaml` と同じ
- **クラスターポッド**：クラスター名前に一致するクラスター名前空間のポッド
- **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
- **podlogs**：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**：ジョブによって作成された Pod のログ（オプション、デフォルトでオフ）

`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` フラグを使用しない場合、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` は無効になっています。

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、クラスターがデフォルトにない限り、クラスター名、およびほとんどの場合名前空間を指定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
```

4.30 自動フェイルオーバー

プライマリで予期しないエラーが発生した場合、クラスターはフェー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る
- 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

フェールオーバーのシナリオでは、プライマリが正常に動作していると想定できません。

上記のようなケースの後、プライマリポッドの `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検出されます。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、レプリカの WAL レシーバーが停止します。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昇格を開始し、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、以前のプライマリポッドは再起動し、プライマリではなくなったことを検出し、レプリカノードになります。

重要: この 2 フェーズの手順は、WAL レシーバーを正常に停止でき、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します。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** を試行します。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ています。
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。

4.30.1 RTO と RPO への影響

フェールオーバーにより、サービスが影響を受けたり、データが失われたりする可能性があります。

1. プライマリに障害が発生し始めた間、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、サービスは影響を受けますが、データは失われません。
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、データは失われません。

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ことは、RPO の観点からは安全ですが、通常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： スイッチオーバープロセスを説明するときに **インスタスマネージャーのインプレース更新** で既に述べたように、`.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、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を長時間放置するときに、データ損失のリスクが削除される場合があります。

4.31 トラブルシューティング

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

ヒント: Kubernetes 管理者は、[kubect1](#) ページをブックマークしておく必要があります。

4.31.1 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）

便利なユーティリティ

トラブルシューティングには、必須の `kubect1` ユーティリティに加えて、システムで次のプラグイン/ユーティリティを使用することをお勧めします。

- `kubect1` の `cnpg`
- `jq`、軽量で柔軟なコマンドライン JSON プロセッサ
- `grep`

、指定されたパターンに一致する行を 1 つ以上の入力ファイルで検索します。ほとんどの `*nix` ディストリビューションで既に利用可能です。Windows OS を使用している場合は、`grep` の代わりに `findstr` を使用するか、

`wsl` を直接使用して、好みの*

`nix` ディストリビューションをインストールして上記のツールを使用できます。

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、直接 **JSON format** にログ記録します。その結果、特定のリソースからログを取得するには `kubectl logs` コマンドに依存する必要があります。

詳細については、次のように入力します。

```
kubectl logs --help
```

ヒント: JSON ログは機械で読み取るのに最適ですが、人間が読み取るのは困難です。使いやすさを向上させるには、`jq` コマンドを使用することをお勧めします。たとえば、`|jq -C` で ``kubectl logs`` コマンドを*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。

4.31.2 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（詳細については、

[展開に関する詳細](#) を参照）。

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: 通常の状態では、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-` で始まる名前でも識別されます。高可用性を実現するよう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。

次を使用して、`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の場合）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:14.4-3
Primary instance: cluster-example-1
Instances:     3
Ready instances: 3
Current Write LSN: 0/5000000 (Timeline: 1 - WAL File: 00000001000000000000000004)

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		

(次のページに続く)

(前のページからの続き)

Instances status						
Name	Database Size	Current LSN	Replication role	Status	QoS	
→ Manager Version						
-----	-----	-----	-----	-----	---	
→ -----						
cluster-example-1	33 MB	0/50000000	Primary	OK	BestEffort	
→ 1.12.0						
cluster-example-2	33 MB	0/50000000	Standby (async)	OK	BestEffort	
→ 1.12.0						
cluster-example-3	33 MB	0/50000060	Standby (async)	OK	BestEffort	
→ 1.12.0						

4.31.3 クラスター情報

NAMESPACE 名前空間の<CLUSTER> クラスターのステータスは、次のように確認できます。

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

出力：

NAME	AGE	INSTANCES	READY	STATUS	PRIMARY
<CLUSTER>	10d4h3m	3	3	Cluster in healthy state	<CLUSTER>-1

上記の例では、3つのインスタンスからなる正常な PostgreSQL クラスターが報告され、すべてが *ready* 状態で、<CLUSTER>-1 がプライマリーです。

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と、さらに発見できます。

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

収集するもう1つの重要なコマンドは、cnpg プラグインによって提供される `status` コマンドです。

```
kubectl cnpg status -n <NAMESPACE> <CLUSTER>
```

Tip: `--verbose` オプションを追加すると、より多くの情報を出力できます。

注釈: クラスターの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の変更を適用します。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、cnpg 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:14.4-3
```

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。

4.31.4 ポッド情報

次を使用して、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を追加します。

```
kubectl logs -n <NAMESPACE> <CLUSTER>--<N> | \
jq -r select(.logger=="postgres") | [(ts|strflocaltime("%Y-%m-%dT%H:%M:%S %Z")), .
→record.message] | @csv
```

PostgreSQL ポッドに関する追加情報を収集してフィルタリングする

クラッシュした特定のポッドからログを確認します。

```
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",
  "session_start_time": "2021-11-08 14:07:44 UTC",
  "virtual_transaction_id": "3/75",
  "transaction_id": "0",
  "error_severity": "FATAL",
  "sql_state_code": "28000",
  "message": "role \"streaming_replica\" does not exist",
  "backend_type": "walsender"
}
```

特定のポッドのログで PostgreSQL DB エラーメッセージをフィルタリングします。

```
kubectl logs -n <NAMESPACE> <CLUSTER>--<N> | jq -r .err | select(. != null)
```

出力：

```
dial unix /controller/run/.s.PGSQL.5432: connect: no such file or directory
```

特定のポッドから **err** ワード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。

```
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
```

4.31.5 バックアップ情報

次を使用して、名前付け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。

4.31.6 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。

4.31.7 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が戦略的に重要です。

次を使用して、Kubernetes クラスター内のノードのリストを取得できます。

```
# look at the worker nodes and their status
kubectl get nodes -o wide
```

さらに、特定のクラスターのポッドが実行されているノードのリストを収集できます。

```
kubectl get pod -l cnpg.io/clusterName=<CLUSTER> \
-L role -n <NAMESPACE> -o wide
```

後者は、ポッドが配布されている場所を理解することが重要です。

affinity/anti-affinity rules and/or tolerations を使用している場合に非常に役立ちます。

4.31.8 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。現在利用可能な条件は次のとおりです。

- LastBackupSucceeded
- 連続アーカイブ
- 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が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が正常に終了されました。それ以外の場合は、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まで待機できます。

特定の条件を待機する方法

- バックアップ: `bash $ kubectl wait --for=condition=LastBackupSucceeded cluster/<CLUSTER-NAME> -n <NAMESPACE>`
- ContinuousArchiving: `bash $ 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_37` 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

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

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

4.31.9 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許容が正しく構成されていません
- 使用可能なノードがまったくありません。これは、`cluster-autoscaler` がいくつかの制限に到達したか、一時的な問題が発生した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にレプリカが同期しない

メンテナンスのため、レプリカが少しオフになる場合があります（Kubernetes ノードがドレインされるときを考えてください）。クラスターで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>`
```

で説明したように、WAL 管理ポリシーに従って既にリサイクルされています）同期の。

同様に、`pg_rewind` が以前のプライマリに存在しない 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、ポッドは準備ができていないため、PVC を削除して、オペレーターにレプリカをリビルド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を使用しており、PV 自分自身を削除することに自信がある場合は、次のようにできます。

```
PODNAME=<POD>
VOLNAME=$(kubectl get pv -o json | \
jq -r .items[] | select(.spec.claimRef.name=="$PODNAME")|.metadata.name)

kubectl delete pod/$PODNAME pvc/$PODNAME pv/$VOLNAME
```

4.32 フェンシング

CloudNativePG のフェンシングは、PostgreSQL クラスターの 1 つ以上、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、ポッドの実行中に 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ます。

4.32.1 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、インスタンス名の JSON 形式のリストを必要とする `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。アノテーションが `["*"]`、ワイルドカードを使用した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。注釈が空の JSON リストに設定されている場合、オペレーターは注釈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。

アノテーションは、`kubectl annotate` コマンドなどを使用して、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
metadata:
  annotations:
    cnpg.io/fencedInstances: ["cluster-example-1"]
[...]
```

4.32.2 フェンシングを持ち上げる方法

フェンシングは、アノテーションをクリアするか、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` を使用して手動で実行するか、次のように `kubectl cnpg fencing` サブコマンドを使用して実行できます。

```
# 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 ""
```

4.32.3 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、続いて必要な場合の高速シャットダウンで構成されます。次に：

- Pod は維持されます
- 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。
- 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ないため、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` の場合に非常に役立ちます。

4.33 エンドツーエンドのテスト

CloudNativePG オペレーターは、一連の **エンドツーエンド (E2E)** テストを介して、各コミット後に自動的にテストされます。これにより、オペレーターは PostgreSQL クラスターを正しくデプロイして管理できます。

さらに、次の Kubernetes バージョンがコミットごとにテストされ、開発プロセスの初期段階で障害とバグが検出されます。

- 1.24
- 1.23
- 1.22
- 1.21
- 1.20
- 1.19

次の PostgreSQL バージョンがテストされています。

- PostgreSQL 15 ベータ (**実稼働向けではありません!!!**)
- PostgreSQL14
- PostgreSQL13
- PostgreSQL 12
- PostgreSQL11
- PostgreSQL 10

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。

*オペレーターのインストール;

- **Cluster** 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。

*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。

- **Cluster** のスケールアップとスケールダウン;
- フェールオーバー;

*スイッチオーバー;

- 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。

*バックアップと ScheduledBackups の実行。

- Barman Cloud on Azure blob ストレージを使用したバックアップと ScheduledBackups の実行。
- 同期複製。

*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。
- NodeSelector を使用した Pod アフィニティ。
- メトリックの収集。
- JSON ログ形式;
- ConfigMap を介したオペレーター設定。
- オペレーターポッドの削除;
- オペレーターポッドのエビクション。
- 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。

*ノードドレイン;

- 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。
- 物理レプリカクラスター。

*ストレージ拡張;

*データの破損。

4.34 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 10 以降の実行可能ファイル： - 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-wal-archive - barman-cloud-wal-restore - barman-cloud-backup - barman-cloud-restore - barman-cloud-backup-list - barman-cloud-check-wal-archive
- 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）
- 賢明なロケール設定

CloudNativePG はインスタスマネージャーで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、**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**でのみ CloudNativePG によって使用されます。

EDB は、CloudNativePG のパブリックコンテナイメージを提供およびサポートし、ghcr.io で公開します。

4.34.1 イメージタグの要件

イメージ名には Docker で有効な名前を付けることができますが、CloudNativePG オペレーターは**イメージタグ**を使用して、イメージによって実行される 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 (9.6 または 12 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。

プレフィックスの後には、Docker タグで有効で受け入れられる有効な文字の組み合わせを、前にドット、アンダースコア、またはマイナス記号を続けることができます。

使用できる画像タグの例:

- 9.6.19-alpine
- 12.4
- 11_1
- 13
- 12.3.2.1-1

警告: *latest* はイメージの有効なタグとは見なされません。

4.35 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](#) フレームワーク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。

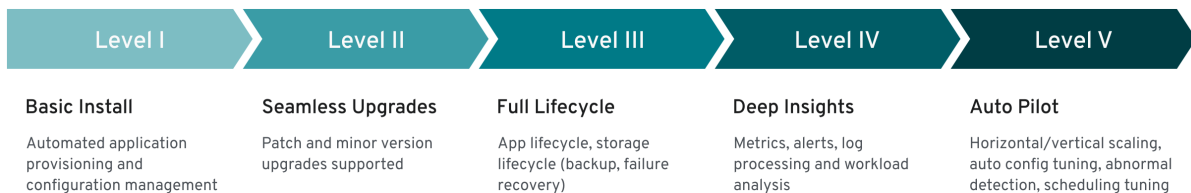


図 9: Operator Capability Levels

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」**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考えています。

4.35.1 レベル 1 - 基本インストール

機能レベル 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機能強化が含まれます。

重要: このレベルの**情報セキュリティ**の部分を検討しています。

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、**Cluster**、**Backup**、および **ScheduledBackup** を定義する Kubernetes マニフェストを使用して、宣言的な方法でインストールされます。

宣言構成による PostgreSQL クラスターの展開

PostgreSQL クラスター（オペランド）は、完全に宣言的な方法で **Cluster** カスタムリソースを使用して定義されます。PostgreSQL のバージョンは、要求されたレジストリから自動的に取得される 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まります。オペランドを展開すると、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 が内部にある任意の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レジストリにアクセスするための `imagePullSecrets` と、コンテナイメージの不変性をより細かく制御するためのタグもサポートしています。

ラベルとアノテーション

オペレーターは、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スマネージャー

オペレーターは、Patroni や Stolon などの外部ツールを使用して 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、各ポッド内の `/controller/manager` という名前のファイルにオペレーター実行可能ファイルを挿入します。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身と調整するために使用されます。インスタスマネージャーは、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た Unix シグナルはインスタ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。インスタスマネージャーは Go で作成されており、外部依存関係はありません。

ストレージ構成

ストレージは、データベースワークロードの重要なコンポーネントです。オペレーターは、ストレージの観点から Kubernetes のネイティブ機能とリソースを利用するため、基盤となる Kubernetes 環境が提供できるものに基づいて、ワークロードの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。

[cnp-bench](#) オープンソースプロジェクトを使用して、実稼働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使用して、クラスター内のレプリカを自動的に検出します。1 に設定されている場合、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。

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 サービスを使用して、データベースへのアクセスを透過的に管理します。構成よりも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。クラスターを実行するための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。

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適用して、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ています。

- ポッドアフィニティとアンチアフィニティ
- ノードセクター
- テイントと寛容

コマンドラインインターフェース

CloudNativePG には、独自のコマンドラインインターフェースがありません。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** に依存しています。

クラスターの現在のステータス

オペレーターは、監視されたクラスターのステータスで 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：その pod -1 はプライマリであり、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、詳細を提供するために **kubectl** の **cnpg** プラグインによって使用されます。

オペレーター認証局

オペレーターは自分用の認証局を自動的に作成します。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て、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 クラスターの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 証明書の発行と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。証明書は、**kubectl** の **cnpg** プラグインを使用して発行できます。

TLS 接続

オペレーターは、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する TLS / SSL 接続を透過的かつネイティブにサポートします。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** リソースの 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` での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のメジャーアップグレードもカバーしています。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て、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 からであり、操作のすべてのデータベース部分、および要求された場合はロールに対して実行される新しいクラスタープライマリからの `pg_dump` および `pg_restore` に依存しています。

PostGIS クラスター

CloudNativePG は、PostgreSQL の最も一般的な拡張機能の 1 つである地理データベース用の **PostGIS クラスター** オープンソース拡張機能を使用したクラスターのインストールをサポートしています。

PostgreSQL の基本的な LDAP 認証

オペレーターを使用すると、[PostgreSQL documentation: LDAP authentication](#) で説明されているように、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

Operator は、`kubectl apply` を介して Kubernetes マニフェストをインストールし、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、構成よりも規約のパラダイムをサポートし、標準のデフォルト値を決定しながら、オーバーライドや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。

4.35.2 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 サーバーに関するものです。これには、**PostgreSQL のマイナーリリースの更新**（通常の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

オペレーターは、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、オペレーターのアップグレード後に **インスタスマネージャーのインプレース更新** もサポートします。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れに続く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な構成アプローチを使用してアップグレードできます。オペレーターは、メジャーバージョン内のマイナーな PostgreSQL リリースの観点から両方向に進むことを可能にしながら、PostgreSQL のメジャーアップグレードを防ぎます（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`）を発行するのを待ちます。実際のデータベースの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発生す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）。

4.35.3 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。 **障害復旧** は、データベースのバックアップと復旧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、事業継続性のもう 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。

- 同期レプリケーション、（カスケード）レプリケーションクラスターなどの PostgreSQL 物理レプリケーションの制御。
- 接続プーリング、pgBouncer を使用した接続プーリングレイヤーを介してパフォーマンスと制御を向上させます。

PostgreSQL のバックアップ

オペレーターは、物理ベースのバックアップと継続的な WAL アーカイブに基づいた PostgreSQL のネイティブ継続的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 と 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 を指定することにより（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するために）クラスターレベルで定義されます。汎用エンドポイント URL。継続的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さ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 に似た構文を使用して、ScheduledBackup カスタム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にバックアップを中継します。

`barman-cloud-wal-restore` と `barman-cloud-backup` は両方とも、GNU GPL 3 条件に基づいてアプリケーションコンテナイメージで配布されます。

バックアップからの完全な復元

オペレーターを使用すると、`barman-cloud-backup` を使用して作成された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 ファイルを再生し、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、オペレーターは要求された数のスタンバイインスタンスをプライマリから複製します。CloudNativePG は、アーカイブからの並列 WAL フェッチをサポートしています。

バックアップからのポイントインタイムリカバリ (PITR)

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。この機能は完全な復元の上に構築されており、

[PostgreSQL for PITR](#) で使用可能なすべてのオプションをサポートしています。

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、ローカルの高可用性 CloudNativePG クラスターで**ゼロデータ損失** (RPO=0) を達成します。オペレーターは、常に使用可能な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`) の次の式を介して、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 クラスタトポロジを定義します。`replica` オプションを使用すると、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するように独立したクラスターをセットアップできます。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 Kubernetes の外部でも、物理環境または仮想環境で実行できます。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 SHIPPING 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 フェデレーションのネイティブ機能を待っている間は、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 プロブと readiness プロブを定義し、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 probe 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（サーバーは正常に接続を受け入れます）および 1（サーバーは起動中などに接続を拒否しています）。readiness プロブは、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に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、指定され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オ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使用可能になるまで待機します。

フェンシング

フェンシングとは、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場合に、そのデータを保護するプロセスです。インスタンスがフェンスされると、ポッドの実行中に 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ます。

ポッドでの永続ボリュームストレージの再利用

オペレーターは、ユーザーによって削除されたポッド、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、使用可能な場合は `PersistentVolumeClaim` を再利用し、プライマリからデータを再クローンする必要を回避します。

CPU とメモリの要求と制限

オペレーターを使用すると、管理者はマニフェストの `resources` セクションを介して、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。特に、`requests` と `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、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。アプリケーションは、PostgreSQL サービスに直接接続する代わりに、PgBouncer サービスに接続して、既存の接続の再利用を開始できるようになりました。

4.35.4 レベル 4 - 深い洞察

機能レベル 4 は **可観測性**、特にモニタリング、アラート、傾向分析、ログ処理に関するものです。これには、Prometheus、Grafana、Fluent Bit などの外部ツールのほか、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が受け入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ための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。[cnp-sandbox プロジェクト] は、いくつかの基本的なメトリックとダッシュボードの例を提供することにより、CloudNativePG を Prometheus および Grafana と統合する方法を示すオープンソースの Helm チャートです。

JSON 形式の PostgreSQL エラーメッセージの標準出力ログ

すべてのログメッセージは、タイムスタンプの第 1 レベルの定義、ログレベル、およびログエントリのタイプ（正規の PostgreSQL エラーメッセージチャンネルの `postgres` など）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。

リアルタイムの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と実行の統計を追跡できます。
 - `: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）。

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。

4.35.5 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た、**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。前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_rewrite` を実行し、スタンバイとして再起動します。

スタンバイの自動再作成

スタンバイをホストしているポッドが削除された場合、オペレーターはスタンバイサーバーを再作成する手順を開始します。

4.36 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示します。

重要: これらはデモンストレーションと実験を目的としたもので、**クイックスタート** で説明されているように、Minikube または Kind を使用して個人の Kubernetes クラスターで実行できます。

基本クラスター：cluster-example.yaml クラスターの基本的な例。

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

(次のページに続く)

(前のページからの続き)

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

カスタムクラスター : cluster-example-custom.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:
    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 : cluster-storage-class.yaml :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgresql-storage-class
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
```

standard の指定されたストレージクラスを使用する基本的なクラスター。

カスタマイズされたストレージクラスとバックアップ: **前提条件**: バケットストレージが利用可能である必要があります。サンプル構成は AWS 用です。セットアップに合わせて変更してください: cluster-storage-class-with-backup.yaml

```
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:
```

(次のページに続く)

(前のページからの続き)

```

    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 : cluster-storage-class-with-backup.yaml

```

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

```

適用され、Healthy : backup-example.yaml : 前のサンプルに対して実行されるバックアップの例

```

apiVersion: postgresql.cnpg.io/v1
kind: Backup

```

(次のページに続く)

(前のページからの続き)

```

metadata:
  name: pg-backup-example
spec:
  cluster:
    name: pg-backup

```

PVC（永続ボリューム要求）が構成されたクラスター：cluster-pvc-template.yaml：明示的な永続ボリューム要求テンプレートを使用する基本クラスター。

```

apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgresql-pvc-template
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:
    size: 1Gi
    pvcTemplate:
      accessModes:
        - ReadWriteOnce
      resources:
        requests:
          storage: 1Gi
        storageClassName: standard
      volumeMode: Filesystem

```

完全な例：cluster-example-full.yaml：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。

```

# 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:

```

```

  VHHWZVE0bk44M1NTaV1Ib3N3cU9VUlp2UURhTDRLcE5FbHNDRUVlOWJ3RHhNZDczS2NrSWVYeIM1Y1U2TG1DMg==

```

(次のページに続く)

(前のページからの続き)

```

    username: YXBw
kind: Secret
metadata:
  name: cluster-example-app-user
type: kubernetes.io/basic-auth
- -
apiVersion: v1
data:
  password:
    ↪dU4zaTFIaDBiWWJDYzRUeVZBYWNCaG1TemdxdHppeG1PVmpBbjBRSUNoc0pyU2110VBZMmZ3MnE4RUtLTHBaOQ==
  username: cG9zdGdyZXM=
kind: Secret
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: c2Vjc mV0X2tleQ==
- -
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-full
spec:
  description: "Example of cluster"
  imageName: ghcr.io/cloudnative-pg/postgresql:14.4
  # 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

```

(次のページに続く)

```
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:
    #   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"
```

(前のページからの続き)

```

    cpu: "1"
  limits:
    memory: "1Gi"
    cpu: "2"

  affinity:
    enablePodAntiAffinity: true
    topologyKey: failure-domain.beta.kubernetes.io/zone

  nodeMaintenanceWindow:
    inProgress: false
    reusePVC: false

```

PostGIS の例 : `postgis-example.yaml` : 「PostGIS クラスター」の例 (詳細は PostGIS section を参照)

```

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

```

ストリーミングを介したレプリカクラスター: 前提条件: `cluster-example.yaml` が適用され、正常

```

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

  storage:
    size: 1Gi

```

: cluster-example-replica-streaming.yaml: ストリーミングレプリケーションを使用した cluster-example に続くレプリカクラスター。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-replica-example
spec:
  instances: 1

  bootstrap:
    pg_basebackup:
      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
```

バックアップが構成された単純なクラスター: **前提条件** : 構成では、minio が実行され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。 cluster-example-with-backup.yaml バックアップが構成された基本クラスター。

```

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

  # Persistent storage configuration
  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

```

バックアップを介したレプリカクラスター：前提条件：cluster-storage-class-with-backup.yaml が適用され、正常。

```

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

```

(次のページに続く)

(前のページからの続き)

```

# 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

```

そして、バックアップ cluster-example-trigger-backup.yaml を適用して完了しました。

```

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

```

: cluster-example-replica-from-backup-simple.yaml : バックアップが構成されたクラスターに続くレプリカクラスター。

```

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

```

(次のページに続く)

(前のページからの続き)

```

    source: cluster-example-backup

storage:
  size: 1Gi

externalClusters:
- name: cluster-example-backup
  barmanObjectStore:
    destinationPath: s3://backups/
    endpointURL: http://minio:9000
    s3Credentials:
      accessKeyId:
        name: minio
        key: ACCESS_KEY_ID
      secretAccessKey:
        name: minio
        key: ACCESS_SECRET_KEY

```

SQL ファイルを含むブートストラップクラスター: cluster-example-initdb-sql-refs.yaml: データベースが作成された直後に Secret と ConfigMap で定義されたクエリのセットを実行するクラスターの例。

```

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

```

(次のページに続く)

```
bootstrap:
  initdb:
    database: appdb
    owner: appuser
    postInitSQL:
      - create table numbers (i integer)
      - 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
```

使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。

4.37 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

4.38 よくある質問 (FAQ)

4.38.1 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。なぜ逆なのですか？

2021 年 9 月の [independent research survey commissioned by the Data on Kubernetes Community](#) では、回答者の半分がほとんどの本番ワークロードを Kubernetes で実行していることが明らかになりました。90% は Kubernetes はステートフルワークロードに対応できると考えており、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急な学習曲線があります）や Kubernetes オペレーターの品質など、重要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の熱狂的な人にとって、本当のゲームチェンジャーは

[*Kubernetes 1.14 in April 2019*](#) でのローカル永続ボリュームのサポートの導入です。

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 の用語では、アプリケーションはワークロードと呼ばれます。ワークロードは、Web アプリケーションサーバーのようにステートレスにすることも、データベースのようにステートフルにすることもできます。この概念を PostgreSQL にマッピングすると、不変のアプリケーションコンテナは、実行され、不変の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、展開がより安全になり、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？

クラウドネイティブコンピューティング財団は、[*Cloud Native*](#) “という用語を定義しています。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 つの主な概念として解釈しています。

1. 人々、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。より安全で効率的で魅力的な方法でビジネスに価値を

2. イミュータブルアプリケーションコンテナ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、クラウドネイティブアプリケーションの展開、管理、スケーラビリティを自動化する Kubernetes です。

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 Ibram と Huß によって定義されたものです。

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使用し、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、別の Kubernetes ノードに移動します。

最も洗練されたアプローチは、演算子を使用して PostgreSQL を実行することです。オペレーターは Kubernetes コントローラーの拡張であり、複雑なアプリケーションがビジネス継続性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マティック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展開する（最初のステップ）だけでなく、予期しないイベントに適切に対応する必要があります。典型的な例はフェイルオーバーです。

オペレーターは、自己修復、スケーラビリティ、レプリケーション、高可用性、バックアップ、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、ログ管理および監視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なります。

CloudNativePG では、宣言的な構成を介して 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、調整ループが自己修復手順をトリガーします。そこで、CloudNativePG のようなオペレーターが登場します。

CloudNativePG は完全に宣言型のオペレーターだとおっしゃいます。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をクローニングし、3 番目のインスタンス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 などのソース制御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。

クラウドネイティブ PostgreSQL を最大限に活用して、CNCF 認定プログラムから「Certified Kubernetes Administrator (CKA)」ステータスを取得したいと考えているすべての人へのアドバイスです。

4.38.2 データベース管理

なぜ PostgreSQL を使用する必要があるのですか？

PostgreSQL は、Linux がオペレーティングシステム空間で表すものとデータベース領域で同等であると考えています。Postgres の現在の最新メジャーバージョンはバージョン 14 で、すぐに出荷されます。

- ネイティブストリーミングレプリケーション、物理的および論理的
- 継続的なホットバックアップとポイントインタイムリカバリ
- 単一インスタンスの垂直スケーラビリティを向上させるためのデータベース分野で非常によく知られている技術である、水平方向のテーブルパーティショニングの宣言的パーティショニング
- 拡張性、地理データベース用の *PostGIS クラスタ* のような拡張機能
- 垂直スケーラビリティのための並列クエリ
- JSON サポート、標準 SQL を介して照会される構造化データと非構造化データの両方のマルチモデルハイブリッドデータベースを解き放ちます

などなど…

単一の PostgreSQL インスタンスでホストするデータベースはいくつですか？

単一のマイクロサービスアプリケーションで完全に管理される単一のデータベース専用、単一の PostgreSQL クラスタ（プライマリサーバーとマルチプルのスタンバイサーバー）を使用することをお勧めします。ただし、「postgres」スーパーユーザーを活用することにより、必要な数のユーザーとデータベースを作成できます（使用可能なリソースによります）。

この推奨の理由は、マイクロサービスに基づくクラウドネイティブの概念にあります。純粋なマイクロサービスアーキテクチャでは、マイクロサービス自身自身が管理するデータを排他的に所有する必要があります。これらは、フラットファイル、キュー、キーバリューストア、またはこの場合は構造化データと非構造化データの両方を含む PostgreSQL リレーショナルデータベースです。一般的な考え方は、スキーマ管理と移行を含め、マイクロサービスのみがデータベースにアクセスできるということです。

CloudNativePG は、このようにすぐに動作するように設計されており、デフォルトでは、アプリケーションユーザーと、前述のアプリケーションユーザーが所有するアプリケーションデータベースを作成します。

PostgreSQL インスタンスを単一のマイクロサービス所有データベースに予約すると、以下が強化されます。

- リソース管理：PostgreSQL では、CPU、およびメモリの制約のあるリソースは一般的にデータベースレベルではなくインスタンスレベルで処理されるため、プードレベルでの Kubernetes リソース管理ポリシーとの統合が簡単になります
- 物理的連続バックアップとポイントインタイムリカバリ（PITR）：PostgreSQL がインスタンスレベルで継続的なバックアップとリカバリを処理することを考えると、インスタンスごとに 1 つのデータベースを持つことで PITR 操作が簡素化され、保持ポリシー管理が差別化され、バックアップのデータ保護が向上します
- アプリケーションの更新：各アプリケーションは、異なるアプリケーションが所有する他のデータベースに影響を与えることなく、更新ポリシーを決定できます

- データベースの更新: 各アプリケーションは、使用する PostgreSQL のバージョンを独立して、PostgreSQL の別のメジャーバージョンにアップグレードするタイミングと条件（カットオーバー時間など）を決定できます

Kubernetes を考慮しない場合のデータベースサイズに上限はありますか?

いいえ、これに関する限り、Kubernetes は仮想マシンやベアメタルと同じです。ただし、実際には、Kubernetes クラスターで使用可能なリソースによって異なります。非常に大規模なデータベース（ワーカー）でのアドバイスは、専用ストレージを備えた単一の Postgres インスタンス専用のシェアードナッシングアーキテクチャを検討することです。 [running PostgreSQL on bare metal Kubernetes with local persistent volumes](#) をベンチマークしたときに、この極端なアーキテクチャパターンが機能することが証明されました。将来のリリースで克服される CloudNativePG の現在の制限は、水平パーティション化を簡単に実装できるようにテーブルスペースがサポートされていないことです。

PostgreSQL クラスターでタイムゾーンを指定するにはどうすればよいですか?

公式ドキュメントで説明されているように、PostgreSQL はタイムゾーンを幅広くサポートしています。

- [Date time data types](#)
- [Client connections config options](#)

タイムゾーンはセッション、トランザクション、さらには PostgreSQL のクエリの一部としても使用できますが、非常に一般的な方法はそれらをグローバルに設定することです。CloudNativePG を使用すると、次の例のように、`spec.postgresql.parameters` セクションでクラスターレベルのタイムゾーンを構成できます。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: pg-italy
spec:
  instances: 1

  postgresql:
    parameters:
      timezone: "Europe/Rome"

  storage:
    size: 1Gi
```

タイムゾーンは次で確認できます。

```
$ kubectl exec -ti pg-italy-1 -c postgres -- psql -x -c "SHOW timezone"
- [ RECORD 1 ]-----
TimeZone | Europe/Rome
```

4.39 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
- バックアップ構成
- バックアップリスト
- バックアップソース
- *BackupSpec*
- バックアップステータス
- *BarmanCredentials*
- *BarmanObjectStoreConfiguration*
- *Bootstrap* 構成
- *BootstrapInitDB*
- *BootstrapPgBaseBackup*
- *BootstrapRecovery*
- 証明書構成
- *CertificatesStatus*
- クラスター
- *ClusterList*
- *ClusterSpec*
- *ClusterStatus*
- *ConfigMapKeySelector*
- *ConfigMapResourceVersion*

- *DataBackupConfiguration*
- *EmbeddedObject* メタデータ
- 外部クラスター
- *Google* 資格情報
- インポート
- *ImportSource*
- インスタンス ID
- *InstanceReportedState*
- *LDAPBindAsAuth*
- *LDAPBindSearchAuth*
- *LDAPConfig*
- *LocalObjectReference*
- *MonitoringConfiguration*
- *NodeMaintenanceWindow*
- *PgBouncerIntegrationStatus*
- *PgBouncer* の秘密
- *PgBouncerSpec*
- ポッドメタ
- *PodTemplateSpec*
- プーラー
- プーラー統合
- *PoolerList*
- *PoolerSecrets*
- *PoolerSpec*
- *PoolerStatus*
- *PostInitApplicationSQLRefs*
- *PostgresConfiguration*
- リカバリーターゲット
- *ReplicaClusterConfiguration*
- *RollingUpdateStatus*
- *S3* 資格情報

- スケジュールされたバックアップ
- *ScheduledBackupList*
- *ScheduledBackupSpec*
- *ScheduledBackupStatus*
- *SecretKey* セレクター
- シークレットバージョン
- *SecretsResource* バージョン
- *StorageConfiguration*
- *SyncReplicaElectionConstraints*
- トポロジー
- *WalBackup* 構成

4.39.1 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
<code>`enablePodAntiAffinity`</code>	ポッドのアンチアフィニティをアクティブにします。このフィールドが明示的に <code>false</code> に設定されていない限り、オペレーターはポッドのアンチアフィニティを定義します	<code>*bool</code>
<code>`topologyKey`</code>	アンチアフィニティ構成に使用する <code>TopologyKey</code> 。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	<code>string</code>
<code>`nodeSelector`</code>	<code>NodeSelector</code> 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/	<code>map[string]string</code>
<code>`tolerations`</code>	<code>Tolerations</code> 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 <code>Tolerations</code> のリストです。詳細： https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/	<code>[]corev1.Toleration</code>

次のページに続く

表 7 – 前のページからの続き

Name	Description	Type
<code>podAntiAffinityType`</code>	PodAntiAffinityType を使用すると、ユーザーはクラスターインスタンス間のポッドのアンチアフィニティをスケジューリング中に強力な要件と見なす必要があるかどうかを決定できます。使用できる値は「preferred」（空の場合のデフォルト）または「required」です。「required」に設定すると、既存のノードがすべて必要なポッドの反アフィニティルールに一致しない場合、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
<code>additionalPodAntiAffinity</code>	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ものに追加するか、false に設定されている場合のみ使用されます。	*corev1.PodAntiAffinity
<code>additionalPodAffinity`</code>	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

4.39.2 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は、ストレージアカウント名と、次のいずれか（および 1 つだけ）が必要です。

- storageKey - storageSasToken
- inheritFromAzureAD を true に設定してポッド環境から資格情報を継承

Name	Description	Type
<code>connectionString`</code>	使用する接続文字列	*SecretKeySelector
<code>storageAccount`</code>	データをアップロードするストレージ アカウント	*SecretKeySelector
<code>storageKey`</code>	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	*SecretKeySelector
<code>storageSasToken`</code>	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	*SecretKeySelector
<code>inheritFromAzureAD</code>	キーを明示的に指定せずに Azure AD ベースの認証を使用します。 - 必須	bool

4.39.3 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
<i>metadata</i>	該当なし	<i>metav1.ObjectMeta</i>
<code>`spec`</code>	バックアップの目的の動作の指定。詳細： https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status	<i>BackupSpec</i>
<code>`status`</code>	バックアップの最後に監視されたステータス。このデータは最新でない可能性があります。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	バックアップステータス

4.39.4 バックアップ構成

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
<i>barmanObjectStore</i>	barman-cloud ツールスイートの構成	* <i>BarmanObjectStoreConfiguration</i>
<code>`retentionPolicy`</code>	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 <i>XXu</i> の形式で表されます。 <code>`XX`</code> は正の整数であり、 <code>`u`</code> は <code>`[dwm]`</code> 日、週、月です。	string

4.39.5 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
<i>metadata</i>	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	<i>metav1.ListMeta</i>
<code>`items`</code>	バックアップのリスト - 必須	<i>[]Backup</i>

4.39.6 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、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
<i>endpointCA</i>	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、証明書の発行者と barman-cloud-wal-archive でのエラーを回避する場合に役立ちます。	*SecretKeySelector

4.39.7 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
<i>cluster</i>	バックアップするクラスター	<i>LocalObjectReference</i>

4.39.8 バックアップステータス

BackupStatus は、バックアップの監視状態を定義します

Name	Description	Type
<i>endpointCA</i>	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、証明書の発行者と barman-cloud-wal-archive でのエラーを回避する場合に役立ちます。	*SecretKeySelector
<i>endpointURL</i>	自動エンドポイント検出をオーバーライドする、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
<i>destinationPath</i>	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	string
<i>serverName</i>	S3 上のサーバー名、このパラメーターを省略した場合はクラスター名が使用されます	string
<i>encryption</i>	S3 API に必要な暗号化方法	string
<i>backupId</i>	Barman のバックアップの ID	string
<i>phase</i>	前回のバックアップの状態	BackupPhase
<i>startedAt</i>	バックアップが開始された日時	*metav1.Time
<i>stoppedAt</i>	バックアップが終了したとき	*metav1.Time

次のページに続く

表 14 – 前のページからの続き

Name	Description	Type
<code>`beginWal`</code>	開始 WAL	string
<code>`endWal`</code>	エンディング WAL	string
<code>`beginLSN`</code>	開始の xlog	string
<code>`endLSN`</code>	エンディングの xlog	string
<code>`error`</code>	検出されたエラー	string
<code>`commandOutput`</code>	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
<code>`commandError`</code>	エラーが発生した場合のバックアップコマンド出力	string
<code>`instanceID`</code>	バックアップの取得元のインスタンスを特定する情報	*InstanceID

4.39.9 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
<code>googleCredentials</code>	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	*GoogleCredentials
<code>`s3Credentials`</code>	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	*S3Credentials
<code>`azureCredentials`</code>	Azure Blob Storage にデータをアップロードするために使用する資格情報	*AzureCredentials

4.39.10 BarmanObjectStoreConfiguration

BarmanObjectStoreConfiguration には、S3 互換のオブジェクトストレージに対して Barman を使用したバックアップ構成が含まれています

Name	Description	Type
<code>`endpointURL`</code>	自動エンドポイント検出をオーバーライドする、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
<code>`endpointCA`</code>	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、証明書の発行者と barman-cloud-wal-archive のエラーを回避する場合に役立ちます	*SecretKeySelector

次のページに続く

表 16 – 前のページからの続き

Name	Description	Type
<i>destinationPath</i>	バックアップを保存するパス（つまり <code>s3://bucket/path/to/folder</code> ）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	string
<code>`serverName`</code>	S3 上のサーバー名、このパラメーターを省略した場合はクラスター名が使用されます	string
<code>`wal`</code>	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 ファイルは非圧縮で保存され、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。	*WalBackupConfiguration
<code>`data`</code>	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る場合があります。	*DataBackupConfiguration
<code>`tags`</code>	Tags は、Barman <code>--tags</code> 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
<code>`historyTags`</code>	HistoryTags は、Barman <code>--history-tags</code> 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

4.39.11 Bootstrap 構成

BootstrapConfiguration には、PostgreSQL クラスターの作成方法に関する情報が含まれています。サポートされているブートストラップメソッドは 1 つだけ定義できます。指定しない場合、`initdb` がブートストラップ方法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
<code>`initdb`</code>	<code>initdb</code> を介してクラスターをブートストラップ	*BootstrapInitDB
<code>`recovery`</code>	バックアップからクラスターをブートストラップする	*BootstrapRecovery
<i>pg_basebackup</i>	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する	*BootstrapPgBaseBackup

4.39.12 BootstrapInitDB

BootstrapInitDB は、initdb を使用する場合のブートストラッププロセスの構成です。詳細については、ドキュメントの Bootstrap ページを参照してください。

Name	Description	Type
<code>`database`</code>	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: <i>app</i> 。 - 必須	string
<code>`owner`</code>	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 <i>database</i> キーの値です。 - 必須	string
<code>`secret`</code>	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	*LocalObjectReference
<code>`options`</code>	クラスターの作成時に initdb に渡す必要があるオプションのリスト。非推奨: これにより、構成に一貫性がなくなる可能性があります。代わりに、明示的に提供されたパラメータ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
<code>`dataChecksums`</code>	-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: <i>false</i>)	*bool
<code>`encoding`</code>	initdb のオプション <code>--encoding`</code> として渡される値 (デフォルト: <i>UTF8</i>)	string
<code>`localeCollate`</code>	initdb のオプション <code>--lc-collate`</code> として渡される値 (デフォルト: <i>C</i>)	string
<code>`localeCTYPE`</code>	initdb のオプション <code>--lc-ctype`</code> として渡される値 (デフォルト: <i>C</i>)	string
<code>`walSegmentSize`</code>	initdb の <code>--wal-segsize`</code> 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、結果として PostgreSQL のデフォルト: 16MB)	int
<code>`postInitSQL`</code>	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します (デフォルトでは空)	[]string
<code>`postInitApplication-SQL`</code>	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します (デフォルトでは空)	[]string
<code>`postInitTemplateSQL`</code>	クラスターが作成された後に <code>`template1`</code> 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)	[]string

次のページに続く

表 18 – 前のページからの続き

Name	Description	Type
<code>`import`</code>	論理バックアップ (<code>pg_dump</code> および <code>pg_restore</code>) を使用して既存の PostgreSQL インスタンスからデータをインポートすることにより、新しいクラスターをブートストラップします	*Import
<code>postInitApplication-SQLRefs</code>	PostInitApplicationSQLRefs は、SQL ファイルを含む ConfigMap または Secret への参照をポイントします。これらの参照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）	*PostInitApplicationSQL-Refs

4.39.13 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
<code>`source`</code>	物理バックアップを作成する必要があるサーバーの名前 - 必須	string
<code>database</code>	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: <code>app</code> 。 - 必須	string
<code>`owner`</code>	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 <code>database</code> キーの値です。 - 必須	string
<code>`secret`</code>	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	*LocalObjectReference

4.39.14 BootstrapRecovery

BootstrapRecovery には、指定された名前でバックアップを復元するために必要な構成が含まれており、スーパーユーザー用に選択されたパスワードでパスワードを変更した後、それを使用して、復元されたプライマリからすべてのインスタンスを複製します。詳細については、ドキュメントの Bootstrap ページを参照してください。

Name	Description	Type
<code>`backup`</code>	復元する必要があるバックアップ	*BackupSource

次のページに続く

表 20 – 前のページからの続き

Name	Description	Type
<code>`source`</code>	バックアップを復元する外部クラスター。これは、バックアップが保存されるフォルダーの名前としても使用されるため、ソースクラスターの名前に設定する必要があります	string
<code>recoveryTarget</code>	デフォルトでは、回復プロセスはアーカイブ内の使用可能なすべての 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、 <code>`RecoveryTarget`</code> 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	*RecoveryTarget
<code>`database`</code>	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： <code>app</code> 。 - 必須	string
<code>`owner`</code>	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 <code>database</code> キーの値です。 - 必須	string
<code>`secret`</code>	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	*LocalObjectReference

4.39.15 証明書構成

CertificatesConfiguration には、サーバー証明書を処理するために必要な構成が含まれています。

Name	Description	Type
<code>`serverCASecret`</code>	サーバー CA 証明書を含むシークレット。定義されていない場合、自己署名 CA で新しいシークレットが作成され、TLS 証明書 <code>ServerTLSSecret</code> の生成に使用されます。サーバー証明書の検証に使用する必要があります。クライアント接続文字列で <code>`sslrootcert`</code> として使用されます。 - <code>ca.key</code> : サーバー SSL 証明書の生成に使用されるキー。 <code>ServerTLSSecret</code> が提供されている場合、これは省略できます。 	string
<code>`serverTLSSecret`</code>	クライアントが postgres に安全に接続できるように <code>`ssl_cert_file`</code> および <code>`ssl_key_file`</code> として設定されるサーバー TLS 証明書とキーを含むタイプ <code>kubernetes.io/tls</code> 。定義されていない場合、 <code>ServerCASecret</code> は <code>`ca.key`</code> 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

次のページに続く

表 21 – 前のページからの続き

Name	Description	Type
<code>replicationTLSSecret</code>	<code>`streaming_replica`</code> ユーザーとして認証するクライアント証明書を含むタイプ <code>kubernetes.io/tls</code> のシークレット。定義されていない場合、 <code>ClientCASecret</code> は <code>`ca.key`</code> も提供する必要がある、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
<code>`clientCASecret`</code>	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。すべてのインスタンスの <code>`ssl_ca_file`</code> 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。 - <code>ca.key</code> : クライアント証明書の生成に使用されるキー。 <code>ReplicationTLSSecret</code> が提供されている場合、これは省略できます。 	string
<code>`serverAltDNSNames`</code>	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

4.39.16 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
<code>expirations</code>	すべての証明書の有効期限。	map[string]string

4.39.17 クラスター

Cluster は PostgreSQL API のスキーマです

Name	Description	Type
<code>metadata</code>	該当なし	<code>metav1.ObjectMeta</code>
<code>`spec`</code>	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	<code>ClusterSpec</code>

次のページに続く

表 23 – 前のページからの続き

Name	Description	Type
<code>`status`</code>	クラスターの最後に監視されたステータス。このデータは最新でない可能性があります。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	<i>ClusterStatus</i>

4.39.18 ClusterList

ClusterList には、クラスターのリストが含まれています

Name	Description	Type
<i>metadata</i>	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	<i>metav1.ListMeta</i>
<code>`items`</code>	クラスターのリスト - 必須	[]Cluster

4.39.19 ClusterSpec

ClusterSpec はクラスターの望ましい状態を定義します

Name	Description	Type
<code>`description`</code>	この PostgreSQL クラスターの説明	string
<code>`inheritedMetadata`</code>	クラスターに関連するすべてのオブジェクトに継承されるメタデータ	*EmbeddedObjectMetadata
<code>`imageName`</code>	コンテナイメージの名前であり、タグ (<image>:<tag>) と決定論的で反復可能な展開のダイジェスト (<image>:<tag>@sha256:<digestValue>) の両方をサポート	string
<code>`imagePullPolicy`</code>	イメージのプルポリシー。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
<code>`postgresUID`</code>	イメージ内の `postgres` ユーザーの UID、デフォルトは `26`	int64
<code>`postgresGID`</code>	イメージ内の `postgres` ユーザーの GID、デフォルトは `26`	int64
<code>`instances`</code>	クラスターに必要なインスタンスの数 - 必須	int

次のページに続く

表 25 – 前のページからの続き

Name	Description	Type
<code>minSyncReplicas`</code>	プライマリとの同期レプリケーションに必要なインスタンスの最小数。未定義または 0 を指定すると、スタンバイが利用できないときに書き込みを完了できます。	int
<code>maxSyncReplicas`</code>	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る同期レプリケーションクォーラムのターゲット値。未定義または 0 は同期レプリケーションを無効にします。	int
<code>postgresql`</code>	PostgreSQL サーバーの構成	<i>PostgresConfiguration</i>
<code>bootstrap`</code>	このクラスターをブートストラップする手順	<i>*BootstrapConfiguration</i>
<code>replica`</code>	レプリカクラスター構成	<i>*ReplicaClusterConfiguration</i>
<code>superuserSecret`</code>	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	<i>*LocalObjectReference</i>
<code>enableSuperuserAccess</code>	このオプションを有効にすると、オペレーターは <code>SuperuserSecret</code> を使用して <code>postgres`</code> ユーザーパスワードを更新します（シークレットが存在しない場合、オペレーターは自動的に作成します）。このオプションを無効にすると、オペレーターは <code>SuperuserSecret</code> のコンテンツを無視し、自動作成時に削除し、 <code>postgres`</code> ユーザーのパスワードを <code>NULL</code> に設定します。デフォルトで有効になっています。	<i>*bool</i>
<code>certificates`</code>	CA および関連する証明書書の構成	<i>*CertificatesConfiguration</i>
<code>imagePullSecrets`</code>	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	<i>[]LocalObjectReference</i>
<code>storage`</code>	インスタンスのストレージの構成	<i>StorageConfiguration</i>
<code>startDelay`</code>	PostgreSQL インスタンスが正常に起動するまでに許可される時間（デフォルト 30）	int32
<code>stopDelay`</code>	PostgreSQL インスタンスが正常にシャットダウンできる時間（デフォルト 30）	int32
<code>switchoverDelay`</code>	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
<code>affinity`</code>	Pod のアフィニティ/アンチアフィニティルール	<i>AffinityConfiguration</i>
<code>resources`</code>	生成されたすべてのポッドのリソース要件。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。	<i>corev1.ResourceRequirements</i>

次のページに続く

表 25 – 前のページからの続き

Name	Description	Type
<code>primaryUpdateStrategy</code>	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戦略: 自動 (<i>unsupervised</i> - デフォルト) または手動 (<i>supervised</i>)	PrimaryUpdateStrategy
<code>primaryUpdateMethod</code>	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方法: スイッチオーバー (<i>switchover</i> - デフォルト) またはインプレース (<i>restart</i>)	PrimaryUpdateMethod
<code>backup</code>	バックアップに使用される構成	*BackupConfiguration
<code>nodeMaintenanceWindow</code>	Kubernetes ノードのメンテナンスウィンドウを定義する	*NodeMaintenanceWindow
<code>monitoring</code>	このクラスターの監視インフラストラクチャの構成	*MonitoringConfiguration
<code>externalClusters</code>	構成で使用される外部クラスターのリスト	[]ExternalCluster
<code>logLevel</code>	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

4.39.20 ClusterStatus

ClusterStatus は、クラスターの監視状態を定義します

Name	Description	Type
<code>instances</code>	クラスター内のインスタンスの合計数	int
<code>readyInstances</code>	クラスター内の準備ができているインスタンスの合計数	int
<code>instancesStatus</code>	InstancesStatus は、インスタンスがどのステータスかを示します	map[utils.PodStatus][]string
<code>instancesReportedState</code>	最後の調整ループ中に報告されたインスタンスの状態	map[PodName]InstanceReportedState
<code>timelineID</code>	Postgres クラスターのタイムライン	int
<code>topology</code>	インスタンストポロジー。	トポロジー
<code>latestGeneratedNode</code>	最後に生成されたノードの ID (ノード名の衝突を回避するために使用)	int
<code>currentPrimary</code>	現在のプライマリインスタンス	string
<code>targetPrimary</code>	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
<code>pvcCount</code>	このクラスターによって作成された PVC の数	int32
<code>jobCount</code>	このクラスターによって作成されたジョブの数	int32

次のページに続く

表 26 – 前のページからの続き

Name	Description	Type
<code>`danglingPVC`</code>	このクラスターによって作成され、まだ使用可能な、ポッドに接続されていないすべての PVC のリスト	<code>[]string</code>
<code>`resizingPVC`</code>	ResizingPVC 条件を持つすべての PVC のリスト。	<code>[]string</code>
<code>`initializingPVC`</code>	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	<code>[]string</code>
<code>`healthyPVC`</code>	ダングリングも初期化もしていないすべての PVC のリスト	<code>[]string</code>
<code>`writeService`</code>	現在の書き込みポッド	<code>string</code>
<code>`readService`</code>	読み取りポッドの現在のリスト	<code>string</code>
<code>`phase`</code>	クラスターの現在のフェーズ	<code>string</code>
<code>`phaseReason`</code>	現在のフェーズの理由	<code>string</code>
<code>`secretsResourceVersion`</code>	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スマネージャーのために行われます	<i>SecretsResource バージョン</i>
<code>`configMapResourceVersion`</code>	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スマネージャーのために行われます	<i>ConfigMapResourceVersion</i>
<code>`certificates`</code>	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。	<i>CertificatesStatus</i>
<code>`firstRecoverabilityPoint`</code>	RFC3339 形式で日付として保存された最初の回復可能ポイント	<code>string</code>
<code>`cloudNativePGCommitHash`</code>	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	<code>string</code>
<code>`currentPrimaryTimestamp`</code>	プライマリへの実際の昇格が発生したときのタイムスタンプ	<code>string</code>
<code>`targetPrimaryTimestamp`</code>	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	<code>string</code>
<code>`poolerIntegrations`</code>	クラスターを参照するプーラーに必要な統合	<code>*PoolerIntegrations</code>
<code>cloudNativePGOperatorHash</code>	演算子のバイナリのハッシュ	<code>string</code>
<code>`onlineUpdateEnabled`</code>	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	<code>bool</code>
<code>`azurePVCUpdateEnabled`</code>	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	<code>bool</code>
<code>`conditions`</code>	クラスターオブジェクトの条件	<code>[]metav1.Condition</code>

4.39.21 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
<i>key</i>	選択するキー - 必須	string

4.39.22 ConfigMapResourceVersion

ConfigMapResourceVersion は、オペレーターによって管理されるシークレットのリソースバージョンです

Name	Description	Type
<i>metrics</i>	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	map[string]string

4.39.23 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
<code>`compression`</code>	オブジェクトストアにストリーミング中に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）、 <i>gzip</i> 、 <i>bzip2`</i> または <i>`snappy`</i> です。	CompressionType
<code>`encryption`</code>	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）、 <i>AES256`</i> および <i>`aws:kms`</i>	EncryptionType
<i>immediateCheckpoint</i>	PostgreSQL サーバーの <code>`checkpoint_completion_target`</code> 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 <code>`false`</code> 。	bool
<code>`jobs`</code>	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	<i>*int32</i>

4.39.24 EmbeddedObject メタデータ

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
<code>`labels`</code>	該当なし	map[string]string
<code>annotations</code>	該当なし	map[string]string

4.39.25 外部クラスター

ExternalCluster は、構成の他のセクションで使用される外部クラスターへの接続パラメーターを表します

Name	Description	Type
<code>`name`</code>	サーバー名、必須- 必須	string
<code>connectionParameters</code>	dbname、host、username などの接続パラメーターのリスト	map[string]string
<code>`sslCert`</code>	このインスタンスへの接続に使用される SSL 証明書への参照	*corev1.SecretKeySelector
<code>`sslKey`</code>	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	*corev1.SecretKeySelector
<code>`sslRootCert`</code>	このインスタンスへの接続に使用される SSL CA 公開キーへの参照	*corev1.SecretKeySelector
<code>`password`</code>	サーバーへの接続に使用されるパスワードへの参照	*corev1.SecretKeySelector
<code>`barmanObjectStore`</code>	barman-cloud ツールスイートの構成	*BarmanObjectStoreConfiguration

4.39.26 Google 資格情報

GoogleCredentials は、Google Cloud Storage 資格情報のタイプです。これは、GKE 環境内で実行する場合でも指定する必要があります。

Name	Description	Type
<code>`gkeEnvironment`</code>	true に設定すると、GKE 環境内で実行されると想定されます。デフォルトは false です。 - 必須	bool
<code>applicationCredentials</code>	認証情報を含む Google Cloud Storage JSON ファイルを含むシークレット	*SecretKeySelector

4.39.27 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
<code>`source`</code>	インポートのソース - 必須	<i>ImportSource</i>
<code>`type`</code>	インポートタイプ。`microservice`または`mono-lith`です。 - 必須	SnapshotType
<code>`databases`</code>	インポートするデータベース - 必須	[]string
<code>`roles`</code>	インポートするロール	[]string
<i>postImportApplicationSQL</i>	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

4.39.28 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
<i>externalCluster</i>	インポートに使用される externalCluster の名前 - 必須	string

4.39.29 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
<code>`podName`</code>	ポッド名	string
<i>ContainerID</i>	コンテナ ID	string

4.39.30 InstanceReportedState

InstanceReportedState は、調整ループ中に報告されたインスタンスの状態を示します

Name	Description	Type
<code>`isPrimary`</code>	インスタンスがプライマリかどうかを示します - 必須	bool

次のページに続く

表 36 – 前のページからの続き

Name	Description	Type
<i>timeLineID</i>	インスタンスがどの TimelineId にあるかを示します	int

4.39.31 LDAPBindAsAuth

LDAPBindAsAuth は、LDAP のバインド認証を使用するために必要なフィールドを提供します

Name	Description	Type
<i>prefix</i>	bind 認証オプションのプレフィックス	string
<i>suffix</i>	bind 認証オプションのサフィックス	string

4.39.32 LDAPBindSearchAuth

LDAPBindSearchAuth は、バインド+検索 LDAP 認証プロセスを使用するために必要なフィールドを提供します

Name	Description	Type
<i>`baseDN`</i>	ユーザー検索を開始するルート DN	string
<i>`bindDN`</i>	ディレクトリにバインドするユーザーの DN	string
<i>`bindPassword`</i>	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	*corev1.SecretKeySelector
<i>searchAttribute</i>	ユーザー名と照合する属性	string
<i>`searchFilter`</i>	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

4.39.33 LDAPConfig

LDAPConfig には、LDAP 認証に必要なパラメーターが含まれています

Name	Description	Type
<i>`server`</i>	LDAP ホスト名または IP アドレス	string
<i>`port`</i>	LDAP サーバーポート	int
<i>`scheme`</i>	使用する LDAP スキーマ。使用可能なオプションは <code>`ldap`</code> および <code>`ldaps`</code>	LDAPScheme
<i>`tls`</i>	1 に設定して LDAP over TLS を有効にします	bool
<i>`bindAsAuth`</i>	認証構成としてバインド	*LDAPBindAsAuth
<i>bindSearchAuth</i>	バインド+検索認証構成	*LDAPBindSearchAuth

4.39.34 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
<i>name</i>	レファレントの名前。 - 必須	string

4.39.35 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
<code>`disableDefault-Queries`</code>	デフォルトのクエリを挿入するかどうか。クラスターにデフォルトのクエリを挿入しない場合は、 <i>true</i> に設定します。デフォルト： <i>false</i> 。	<i>*bool</i>
<i>customQueriesConfigMap</i>	カスタムクエリを含む構成マップのリスト	[]ConfigMapKeySelector
<code>`customQueriesSecret`</code>	カスタムクエリを含むシークレットのリスト	[]SecretKeySelector
<code>`enablePodMonitor`</code>	<code>`PodMonitor`</code> を有効または無効にします	bool

4.39.36 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ポッド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
<i>inProgress</i>	進行中のノードのメンテナンスアクティビティはありますか? - 必須	bool
<code>`reusePVC`</code>	既存の PVC を再利用する（ノードが再度起動するのを待つ）か、しない（他の場所で再作成する - <i>instances >1</i> の場合） - 必須	<i>*bool</i>

4.39.37 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
<i>secrets</i>	該当なし	[]string

4.39.38 PgBouncer の秘密

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
<i>authQuery</i>	認証クエリのシークレットバージョン	シークレットバージョン

4.39.39 PgBouncerSpec

PgBouncerSpec は PgBouncer の構成方法を定義します

Name	Description	Type
<code>poolMode`</code>	プールモード - 必須	PgBouncerPoolMode
<i>authQuerySecret</i>	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 username, passwd FROM pg_shadow WHERE username=\$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。	*LocalObjectReference
<code>authQuery`</code>	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 username, 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
<code>parameters`</code>	PgBouncer に渡す追加パラメーター - 構成可能な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

次のページに続く

表 45 – 前のページからの続き

Name	Description	Type
<code>paused</code>	<code>true</code> に設定すると、PgBouncer は PostgreSQL サーバーから切断し、最初にすべてのクエリが完了するのを待機し、この値が <code>false</code> （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 <code>PAUSE</code> および <code>RESUME</code> コマンドを呼び出します。	<code>*bool</code>

4.39.40 ポッドメタ

PodMeta 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 を作成します。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
<code>labels</code>	オブジェクトの編成と分類（スコープと選択）に使用できる文字列のキーと値のマッピング。レプリケーションコントローラーとサービスのセクターと一致する場合があります。詳細： http://kubernetes.io/docs/user-guide/labels	<code>map[string]string</code>
<code>annotations</code>	アノテーションは、任意のメタデータを保存および取得する外部ツールによって設定されるリソースとともに保存される非構造化キー値マッピングです。これらは照会可能ではないため、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	<code>map[string]string</code>

4.39.41 PodTemplateSpec

PodTemplateSpec は、ユーザが Pod 生成のテンプレートを設定できる構造です。

残念ながら、生成された CRD にはメタデータセクションのフィールドがないため、`corev1.PodTemplateSpec` タイプを使用できません。

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
<code>metadata</code>	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	ポッドメタ

次のページに続く

表 47 – 前のページからの続き

Name	Description	Type
<code>spec`</code>	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	<code>corev1.PodSpec</code>

4.39.42 プーラー

Pooler は pooler API のスキーマです

Name	Description	Type
<code>metadata</code>	該当なし	<code>metav1.ObjectMeta</code>
<code>spec`</code>	該当なし	<code>PoolerSpec</code>
<code>status`</code>	該当なし	<code>PoolerStatus</code>

4.39.43 プーラー統合

PoolerIntegrations は、クラスターを参照するプーラーに必要な統合をカプセル化します

Name	Description	Type
<code>pgBouncerIntegration</code>	該当なし	<code>PgBouncerIntegrationStatus</code>

4.39.44 PoolerList

PoolerList にはプーラーのリストが含まれています

Name	Description	Type
<code>metadata</code>	該当なし	<code>metav1.ListMeta</code>
<code>items`</code>	必須	<code>[]Pooler</code>

4.39.45 PoolerSecrets

PoolerSecrets には、使用されるすべてのシークレットのバージョンが含まれています

Name	Description	Type
<code>serverTLS`</code>	サーバーの TLS シークレットバージョン	<code>シークレットバージョン</code>

次のページに続く

表 51 – 前のページからの続き

Name	Description	Type
<code>`serverCA`</code>	サーバー CA シークレットバージョン	シークレットバージョン
<code>`clientCA`</code>	クライアント CA シークレットバージョン	シークレットバージョン
<code>pgBouncerSecrets</code>	PgBouncer が使用するシークレットのバージョン	*PgBouncerSecrets

4.39.46 PoolerSpec

PoolerSpec はプーラーの望ましい状態を定義します

Name	Description	Type
<code>`cluster`</code>	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	<i>LocalObjectReference</i>
<code>`type`</code>	トラフィックを転送する必要があるインスタンスは? - 必須	PoolerType
<code>instances</code>	必要なレプリカの数 - 必須	int32
<code>`template`</code>	作成する Pod のテンプレート	*PodTemplateSpec
<code>pgbouncer</code>	PgBouncer 設定 - 必須	*PgBouncerSpec

4.39.47 PoolerStatus

PoolerStatus は、プーラーの監視状態を定義します

Name	Description	Type
<code>`secrets`</code>	構成オブジェクトのリソースバージョン	*PoolerSecrets
<code>instances</code>	スケジュールしようとしているポッドの数	int32

4.39.48 PostInitApplicationSQLRefs

PostInitApplicationSQLRefs は、SQL ファイルを含む ConfigMap または Secret への参照をポイントします。これらの参照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
<code>`secretRefs`</code>	SecretRefs は Secrets への参照のリストを保持します	[]SecretKeySelector
<code>configMapRefs</code>	ConfigMapRefs は ConfigMaps への参照のリストを保持しています	[]ConfigMapKeySelector

4.39.49 PostgresConfiguration

PostgresConfiguration は PostgreSQL 構成を定義します

Name	Description	Type
<code>`parameters`</code>	PostgreSQL 構成オプション (postgresql.conf)	map[string]string
<code>`pg_hba`</code>	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
<code>syncReplicaElectionConstraint</code>	同期レプリカが満たす要件。これは、「synchronous_standby_names」パラメーターの設定方法に影響します。	SyncReplicaElectionConstraints
<code>`promotionTimeout`</code>	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。デフォルト値は 40000000 で、1 年を超える秒単位で、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
<code>`shared_preload_libraries`</code>	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
<code>`ldap`</code>	LDAP 構成を指定するオプション	*LDAPConfig

4.39.50 リカバリーターゲット

RecoveryTarget では、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
<code>`backupID`</code>	リカバリー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
<code>`targetTLI`</code>	ターゲットタイムライン (「最新」または正の整数)	string
<code>`targetXID`</code>	対象トランザクション ID	string
<code>`targetName`</code>	ターゲット名 (以前に <code>`pg_create_restore_point`</code> で作成)	string
<code>`targetLSN`</code>	対象の LSN (Log Sequence Number)	string
<code>`targetTime`</code>	RFC3339 標準のタイムスタンプとしてのターゲット時刻	string
<code>targetImmediate</code>	一貫した状態に達したらすぐにリカバリを終了する	*bool

次のページに続く

表 56 – 前のページからの続き

Name	Description	Type
<code>exclusive`</code>	ターゲットを排他的に設定します（デフォルトは true）	*bool

4.39.51 ReplicaClusterConfiguration

ReplicaClusterConfiguration はレプリカクラスターの構成をカプセル化します

Name	Description	Type
<code>enabled</code>	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、回復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
<code>source`</code>	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

4.39.52 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
<code>imageName</code>	Pod に入れる画像 - 必須	string
<code>startedAt</code>	アップデートが開始されたら	metav1.Time

4.39.53 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。これは、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

Name	Description	Type
<code>accessKeyId`</code>	アクセスキー ID への参照	*SecretKeySelector
<code>secretAccessKey`</code>	シークレットアクセスキーへの参照	*SecretKeySelector
<code>region`</code>	リージョン名を含むシークレットへの参照	*SecretKeySelector

次のページに続く

表 59 – 前のページからの続き

Name	Description	Type
<code>`sessionToken`</code>	セッションキーへの参照	<code>*SecretKeySelector</code>
<code>inheritFromIAMRole</code>	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	<code>bool</code>

4.39.54 スケジュールされたバックアップ

ScheduledBackup は `cheduledbackups` API のスキーマです

Name	Description	Type
<code>metadata</code>	該当なし	<code>metav1.ObjectMeta</code>
<code>`spec`</code>	ScheduledBackup の目的の動作の指定。詳細： https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status	<code>ScheduledBackupSpec</code>
<code>`status`</code>	ScheduledBackup の最後に監視されたステータス。このデータは最新でない可能性があります。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	<code>ScheduledBackupStatus</code>

4.39.55 ScheduledBackupList

ScheduledBackupList には ScheduledBackup のリストが含まれています

Name	Description	Type
<code>metadata</code>	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	<code>metav1.ListMeta</code>
<code>`items`</code>	クラスターのリスト - 必須	<code>[]ScheduledBackup</code>

4.39.56 ScheduledBackupSpec

ScheduledBackupSpec は ScheduledBackup の望ましい状態を定義します

Name	Description	Type
<code>`suspend`</code>	このバックアップが中断されているかどうか	<code>*bool</code>

次のページに続く

表 62 – 前のページからの続き

Name	Description	Type
<code>`immediate`</code>	作成後に最初のバックアップをすぐに開始する必要があるかどうか	<code>*bool</code>
<code>`schedule`</code>	スケジュールは、Kubernetes CronJob で使用されるのと同じフォーマットに従います。 https://pkg.go.dev/github.com/robfig/cron#hdr-CRON_Expression_Format - 必須	string
<code>`cluster`</code>	バックアップするクラスター	<i>LocalObjectReference</i>
<code>backupOwnerReference</code>	作成されたバックアップリソース内に配置する ownerReference を示します。 - none：作成されたバックアップオブジェクトの所有者参照はありません（フィールドが導入される前と同じ動作） - self：スケジュールされたバックアップオブジェクトを所有者として設定します。バックアップの - クラスター：クラスターをバックアップの所有者として設定します 	string

4.39.57 ScheduledBackupStatus

ScheduledBackupStatus は、ScheduledBackup の監視状態を定義します

Name	Description	Type
<code>`lastCheckTime`</code>	最新の時刻	<code>*metav1.Time</code>
<code>lastScheduleTime</code>	バックアップが最後にスケジュールされたときの情報。	<code>*metav1.Time</code>
<code>nextScheduleTime</code>	次回はバックアップを実行します	<code>*metav1.Time</code>

4.39.58 SecretKey セレクター

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
<code>key</code>	選択するキー - 必須	string

4.39.59 シークレットバージョン

SecretVersion には、シークレット名とその ResourceVersion が含まれています

Name	Description	Type
<code>`name`</code>	秘密の名前	string
<code>version</code>	シークレットの ResourceVersion	string

4.39.60 SecretsResource バージョン

SecretsResourceVersion は、オペレーターによって管理されるシークレットのリソースバージョンです

Name	Description	Type
<code>`superuserSecretVersion`</code>	「postgres」ユーザーシークレットのリソースバージョン	string
<code>replicationSecretVersion</code>	「streaming_replica」ユーザーシークレットのリソースバージョン	string
<code>applicationSecretVersion</code>	「アプリ」ユーザーシークレットのリソースバージョン	string
<code>`caSecretVersion`</code>	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
<code>`clientCaSecretVersion`</code>	PostgreSQL クライアントサイド CA シークレットバージョンのリソースバージョン	string
<code>`serverCaSecretVersion`</code>	PostgreSQL サーバー側 CA シークレットバージョンのリソースバージョン	string
<code>`serverSecretVersion`</code>	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
<code>`barmanEndpointCA`</code>	提供されている場合、Barman Endpoint CA のリソースバージョン	string
<code>`metrics`</code>	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	map[string]string

4.39.61 StorageConfiguration

StorageConfiguration は、PostgreSQL インスタンスのストレージの構成です

Name	Description	Type
<code>storageClass</code>	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	<code>*string</code>
<code>size</code>	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。サイズダウンはできません。 - 必須	<code>string</code>
<code>resizeInUseVolumes</code>	既存の PVC のサイズを変更します。デフォルトは true	<code>*bool</code>
<code>pvcTemplate</code>	永続ボリュームの要求の生成に使用されるテンプレート	<code>*corev1.PersistentVolumeClaimSpec</code>

4.39.62 SyncReplicaElectionConstraints

SyncReplicaElectionConstraints には、同期レプリカの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ます

Name	Description	Type
<code>enabled</code>	このフラグにより、同期レプリカの制約が有効になりました-必須	<code>bool</code>
<code>nodeLabelsAntiAffinity</code>	ポッドが同じトポロジにあ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	<code>[]string</code>

4.39.63 トポロジー

トポロジーにはクラスター・トポロジーが含まれます

Name	Description	Type
<code>successfullyExtracted</code>	SuccessfullyExtracted は、トポロジー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役立ちます	bool
<code>instances`</code>	インスタンスにはインスタンスのポッドトポロジーが含まれています	map[PodName]PodTopologyLabels

4.39.64 WalBackup 構成

WalBackupConfiguration は、WAL ストリームのバックアップの構成です

Name	Description	Type
<code>compression</code>	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）、 <code>gzip</code> 、 <code>bzip2`</code> または <code>snappy`</code> です。	CompressionType
<code>encryption`</code>	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）、 <code>AES256`および`aws:kms</code>	EncryptionType
<code>maxParallel</code>	並行してアーカイブされる（PostgreSQL インスタンスがバックアップオブジェクトストアにアーカイブしている場合）または並行して復元される（PostgreSQL スタンバイがリカバリオブジェクトストアから WAL ファイルを取得している場合）、WAL ファイルの数。指定しない場合、WAL ファイルは一度に 1 つずつ処理されます。値として正の整数を受け入れます。1 は受け入れられる最小値です。	int

4.40 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。

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](#)

4.40.1 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ています。

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。コミュニティによるサポートの詳細は、以下の [サポートとは](#) を参照してください。

表 71: :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情報をを見つけることができます。		

4.40.2 ネーミングスキーム

私たちの命名スキームは、次のように [Semantic Versioning 2.0.0](#) に基づいています。

```
<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 タグの先頭に v が追加されます。

(前のページからの続き)

\	RELEASES
\ v1.15.0	= the two
\ Apr 21, 2022	last
+-----+-----> release-1.15	releases
	v

次の2種類のサポートを提供しています。

テクニカルサポート：テクニカルサポートは、サポートされているリリースに対してのみベストエフォートで提供されます。

CloudNativePG Slack（[#general](#) チャンネル）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の修正：現在サポートされているすべてのリリースに、セキュリティの修正を含む重要なバグの修正を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。「既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。

4.41 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 1.16 のリリースノート](#)
- [CloudNativePG 1.15 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。

また、CloudNativePG の前身であるすべての release notes from 1.0.0 to 1.14.0 of Cloud Native PostgreSQL by EDB の記録を保持します。

4.41.1 CloudNativePG 1.15 のリリースノート

CloudNativePG の 1.15 マイナーリリースでの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、GitHub のリリースブランチの [commits](#) を参照してください。

バージョン 1.15.3

リリース日：2022 年 8 月 11 日

機能強化：

- PostgreSQL の `huge_pages` オプションの設定を有効にする (#456)
- 昇格の要求から実際の書き込みが可能になるまでの経過時間を出力することにより、フェイルオーバーまたはスイッチオーバー後に、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` と `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ます (#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`) とスイッチオーバー (`switchoverDelay`) に正しい遅延を使用してください。以前に誤ってスワップされました。`switchoverDelay` が設定されておらず、デフォルト値の 40000000 秒を使用している場合、無期限にリスタートをブロックす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すのを防ぎます (#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 (#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)

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witchオーバーとフェイルオーバー中のインスタ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 14.4 に設定

修正：

- PostgreSQL サーバーの CSV ログパイプを開始する `WithActiveInstance` ファンクション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グを確保します-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に特に役立ちます（このパッチの前に、このようなログ標準出力チャンネルに送信されず、完全に失われました）
- ホットスタンバイセンシティブパラメーターを減らし、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
- バックアップとリカバリ： - 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た、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。`recoveryTarget` セクションに `backupID` という新しいフィールドを提供することによる「イミディエイト」ターゲット
- `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、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あった PostgreSQL の `archive_timeout` 設定の構成を有効にします（デフォルトでは 5 分に設定されています）
- `scheduledBackup` リソースに `backupOwnerReference` という新しいフィールドを導入して、作成されたバックアップリソースに所有権の参照を設定します。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 エクスポーターに 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 14.4 に設定

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る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てます
- 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` ブートストラップ方法では、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は、調整ループを再度キューに入れます
- バックアップとクラスター名プロパティの間違った指定の問題を修正

- 複数のレプリカを使用して、高可用性でオペレーターを展開する場合、オペレーターポッドが常に最新の証明書を持つようにします
- `cnpg report operator` コマンドを修正して、複数のレプリカを使用して、高可用性でオペレーターを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` ラベルとアノテーション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、オペレーターの展開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/フェー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、検査と緊急時にポッドは実行され続けます（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 シークレットの使用を有効にします
- **switchover**（デフォルト）および `restart` の値を受け入れ、教師なし `primaryUpdateStrategy` の場合に使用される `primaryUpdateMethod` オプションの導入。このメソッドは、ローリング更新手順中にプライマリインスタンスがどうなるかを制御します
- 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 cnpg` 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` のデフォルトのイメージ名

修正：

- `targetTime` を介したポイントインタイムリカバリのベースバックアップ検出が正しく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が向上しました

- 無効な証明書エラーをスローするのではなく、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収集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、ダウンストリームの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増える場合があります）

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。

4.41.2 CloudNativePG 1.16 のリリースノート

CloudNativePG の 1.16 マイナーリリースでの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、GitHub のリリースブランチの [commits](#) を参照してください。

バージョン 1.16.1

リリース日：2022 年 8 月 11 日

機能強化：

- PostgreSQL の `huge_pages` オプションの設定を有効にする (#456)
- 昇格の要求から実際の書き込みが可能になるまでの経過時間を出力することにより、フェイルオーバーまたはスイッチオーバー後に、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` と `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ます (#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`) とスイッチオーバー (`switchoverDelay`) に正しい遅延を使用してください。以前に誤ってスワップされました。`switchoverDelay` が設定されておらず、デフォルト値の 40000000 秒を使用している場合、無期限にリスタートをブロックす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すのを防ぎます (#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 (#491)
- クラスタ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` ユーザーを使用する (#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)

バージョン 1.16.0

リリース日： 2022 年 7 月 7 日（マイナーリリース）

特徴：

- **PostgreSQL のオフラインデータインポートとメジャーアップグレード**：`bootstrap.initdb.import` セクションを導入して、PostgreSQL 論理バックアップの概念を使用して、既存の PostgreSQL インスタンス（Kubernetes の外部でも）から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、プライマリが実行されているノードとは異なる特性を持つノード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
- 不必要に `DROP EXTENSION IF EXISTS` を実行する代わりに、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
- スイッチオーバーとフェイルオーバー中のインスタ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複製するときに、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、これまでに `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 を有効にしますオブジェクトストア
- Kubernetes 1.24 のサポート

変更点：

- デフォルトのオペランドイメージを PostgreSQL 14.4 に設定
- 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` ファンクション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グを確保します-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に特に役立ちます（このパッチの前に、このようなログ標準出力チャンネルに送信されず、完全に失われました）
- ホットスタンバイセンシティブパラメーターを減らし、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
- バックアップとリカバリ： - 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た、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。`recoveryTarget` セクションに `backupID` という新しいフィールドを提供することによる「イミディエイト」ターゲット
- `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、出力のばらつきを削除します

4.41.3 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 の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、コミットに加えて製品の履歴を示すための情報提供のみを目的としています。ここにリストされているバージョンは **CloudNativePG** には存在しません。

バージョン 1.14.0

リリース日：2022 年 3 月 25 日

特徴：

- Barman Cloud 2.19 で導入された機能を利用して、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資格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：

- インスタンスマネージャーが元の `pg_control` ファイルのコピーを保持することにより、中断された `pg_rewind` を再試行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
- 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するために、`primary_conninfo` のキーによるソートを強制します
- 調整時に `ServiceAccount` の変更（ラベル、注釈など）を保持
- `imagePullPolicy` のデフォルト値の適用を無効にします
- 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

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
- `status.conditions` によるクラスターのステータスの拡張。条件 `ContinuousArchiving` は、クラスターが WAL ファイルのアーカイブを開始したことを示します

- 追加情報を使用して `kubectl` の `cnpg` プラグインの `status` コマンドを改善します。クラスターのステータスを表示する `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を有効期限とともに含む `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監視クエリを追加するためのサポートを追加します。
- ユーザーが AppArmor を使用してコンテナの権限を制限できるようにします (AppArmor サポートで展開された Kubernetes クラスター上)
- `kubectl` の `cnpg` プラグインに Windows プラットフォームのサポートを追加しました。プラグインは Windows x86 および ARM で使用できます
- Kubernetes 1.18 および非推奨の API バージョンのサポートを削除

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シークレット内のユーザー名フィールドの一貫性チェックを追加します。以前は、悪意のあるユーザーがシークレットを使用して PostgreSQL ユーザーのパスワードを変更できました。

修正：

- Postgres Pod 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ぎます。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 を起動できません
- バックアップと 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
- まれにクラッシュする可能性のある 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依存する PostgreSQL 構成パラメーターを正しく処理する
- クラスター作成中の 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、ネットワークに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、フェイルオーバー操作の堅牢性と予測可能性が強化されました。
- `.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減らす
- PgBouncer の readiness プローブを追加します
- Azure プロトコル使用時のバーマン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

修正：

- フェイルオーバー中に、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迅速に通知するようにします
- レプリカクラスター内の PostgreSQL インスタンスの「`archive_mode`」を「`always`」に設定して、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、WAL のアップロードを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する
- デフォルトの Web フック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
- リソースの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、すべての調整サイクルで PodMonitor の削除のログを停止します
- ポッドに到達できない場合に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

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**：`backupObjectStore.maxParallel` オプションを導入して、WAL アーカイブ（PostgreSQL の `archive_command` による）と WAL 復元（`restore_command` による）。並列復元オプションを使用すると、必要な WAL ファイルを取得して順次ではなく並列に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、デフォルトでは、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変更するか、`.spec.monitoring.disableDefaultQueries` オプション（デフォルトでは `false` に設定）を介してクラスターレベルで変更できます
- 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により、PostgreSQL のシャットダウン手順を改善します
- 前のプライマリが正常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する時間を制御する `switchoverDelay` オプションを追加します（デフォルトでは、CloudNativePG は権限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して、PostgreSQL Cluster のリソース要求と制限の変更を処理します
- `kubectl` の `cnpg` プラグインの `status` コマンドを改善します：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、多くの並列ワーカーを含むワークロードのサポートを強化
- 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
- 特定の Postgres クラスターで `disabled` に設定すると、調整ループが実行されないようにする `cnpg.io/reconciliationLoop` アノテーションを導入
- `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て、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、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る再起動されたスタンバイサーバーに影響しますが、liveness プロブのタイムアウトが作動する前にそれを行うことができず、ポッドが `CrashLoopBackOff` に残りますステータス。
- 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に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 Pod 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 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ぎます
- 有効期限が切れた Webhook TLS シークレットを更新するときは、正しい公開キーを使用します。
- 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 デプロイメントを管理します。この機能には、TLS クライアント/サーバー接続、パスワード認証、高可用性、ポッドテンプレートのサポート、主要な PgBouncer パラメーターの構成、PAUSE /RESUME、JSON 形式でのログイン、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 日間のリカバリウィンドウを保証)
- **オペレーターのインプレース更新**：インスタスマネージャーのインプレースオンライン更新を導入すると、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て、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。この機能を使用すると、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のオブジェクトを含めることができます
- インスタンス Prometheus エクスポーターに追加された新しいデフォルトメトリック：バックアップカタログに従った Postgres バージョン、クラスター名、および最初のリカバリーポイント
- 失敗した後にバックアップを取ってください
- 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
- 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 `cnp.io/cluster` ラベルにクラスター名を設定します
- Cluster、Backup、および ScheduledBackup 種類での非推奨の API バージョン `postgresql.cnp.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認証局（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 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：

- スタンバイ昇格中に 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、特定のイメージ名を定義せずに新しい Cluster が作成されたときのデフォルトのメジャーバージョンにすることにより、

PostgreSQL 14 の発売を祝うものです。

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、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 の後続のリリースが遅れるため、OpenShift では利用できません。

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 ファイルを取得するために、`pg_rewind` の `--restore-target-wal` オプションのサポートを導入します (PostgreSQL 13 以降でのみ利用可能)
- 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
- ローカルのインメモリ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する - 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 を定義するための以前の要件を排除します
- スケジュールされたバックアップに `immediate` オプションを導入して、最初に実行された PostgreSQL インスタンスの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、バックアップオブジェクトストアのコンテンツに基づいてリカバリを要求する最も古い一貫したポイントを報告します
- 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 エクスポーターを強化します。

1. ディスク上の WAL ファイルの数と計算された合計サイズ 2. アーカイブステータスフォルダー内の `.ready` および `.done` ファイルの数 3. レプリカモードのフラグ 4. 要求された最小/最大の同期レプリカ、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して、メトリックの収集を一連の PostgreSQL バージョンに制限するときに `runonserver` オプションのサポートを追加
- Barman 2.13 for Barman Cloud で導入された機能を利用して、バックアップと回復のために 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 フォーマットに対応
- 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する `.spec.imagePullPolicy` を導入
- 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さない場合に備えて、オペレーターは一時的に `minSyncReplicas` および `maxSyncReplicas` 設定を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て、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する 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イルオーバーできるようにします (1.7.0 で導入されたバグ)
- LOCAL 同期コミットレベルを使用して管理クエリを実行する
- PGAudit の複数行のログエントリを正しく解析

バージョン 1.7.0

リリース日： 2021 年 7 月 28 日

特徴：

- JSON 出力で直接利用できる `pgaudit` と呼ばれる新しいタイプの `logger` を使用して PGAudit にネイティブサポートを追加します
- 以下を通じてモニタリングと可観測性機能を強化します。
- `pg_stat_statements` および `auto_explain` 拡張機能のネイティブサポート - 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 エクスポートの `target_databases` オプション - `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 `cnpg` 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向上させます

- 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、PostgreSQL サーバーのカスタムクライアント認証局と証明書を定義し、クライアント証明書を認証し、PostgreSQL ノード間の通信を保護します
- 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する `.spec.backup.barmanObjectStore.endpointCA` オプションを追加します

修正：

- Prometheus エクスポーターでヒストグラムを正しく解析
- クラスター用にオペレーターが作成したサービスを調整する

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）に設定すると preferredDuringSchedulingIgnoredDuringExecution が使用されるか、required に設定して requiredDuringSchedulingIgnoredDuringExecution になります。

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 と 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 のオペレーター展開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ように 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ます
- クラスター名（サービス名の継承元）に DNS-1035 検証スキームを採用
- スタンバイインスタンスをクローニングするとき、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
- フィジカルベースのバックアップに関して `Backup` リソースを `beginWal`、`endWal`、`beginLSN`、`endLSN`、`startedAt`、`stoppedAt` に統合
- 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
- 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

- 以下を含むマイナーなバグ修正 - 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ていない - `barmanObjectStore` バックアップの `immediateCheckpoint` および `jobs` パラメーターの処理を修正しました

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
- `kubect1` の `cnf` プラグインに `restart` コマンドを追加して、ロールアウト形式で特定の PostgreSQL クラスターのポッドを再起動します

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ている場合は、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ト接続に `application_name` として `cnf_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：- `pg_basebackup` を呼び出すときに `-R` を使用しない

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用に予約されました）
- 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` との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なサポートを導入
- Operator Lifecycle Manager (OLM) の展開をサポートし、その後に OperatorHub.io に存在する
- 米国国防総省 (DoD) の国防情報システム局 (DISA) と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 との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 での E2E テストを改善
- 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 cnpg 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 cnpg 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` よりも低い値にロールバックする
- 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` の `Cluster`、`Backup`、および `ScheduledBackup` を実装しています。これらのリソースを使用して、次の主な機能を備えた Kubernetes 内に 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、高可用性を実現するための Kubernetes API サーバーとの直接統合
- 以下を介した自己修復機能 - 最も調整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）
 - 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
 - ポッドでの永続ボリューム ストレージの再利用
 - PostgreSQL のマイナーバージョンとオペレーターのアップグレードのローリングアップデート
 - TLS 接続とクライアント証明書認証
 - S3 互換オブジェクトストアへの継続的なバックアップ
 - S3 互換オブジェクトストアバックアップからの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 のエラーメッセージの標準出力ロギング