



EDB

CloudNativePG

リリース *1.15.0*

EDB

2022 年 05 月 16 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	メインな機能	7
第 4 章	このガイドについて	9
4.1	Before You Start	9
4.1.1	Kubernetes の用語	9
4.1.2	PostgreSQL の用語	11
4.1.3	クラウドの用語	12
4.1.4	次にすること	12
4.2	Use cases	12
4.2.1	ケース 1: Kubernetes 内のアプリケーション	12
4.2.2	ケース 2: Kubernetes 外のアプリケーション	15
4.3	Architecture	15
4.3.1	読み取り/書き込みワークロード	16
4.3.2	読み取り専用のワークロード	16
4.3.3	マルチクラスター展開	17
4.4	Installation and upgrades	18
4.4.1	Kubernetes でのインストール	18
4.4.2	展開に関する詳細	19
4.4.3	アップグレード	19
4.5	Quickstart	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	22
4.5.2	パート 2-CloudNativePG のインストール	23
4.5.3	パート 3- PostgreSQL クラスターをデプロイする	23
4.6	Bootstrap	24
4.6.1	bootstrap セクション	25
4.6.2	externalClusters セクション	25
4.6.3	空のクラスターのブートストラップ (initdb)	26
4.6.4	別のクラスターからのブートストラップ	29
4.7	Security	39
4.7.1	コード	39
4.7.2	コンテナ	40
4.7.3	クラスター	40
4.8	Postgres instance manager	44
4.8.1	活性と準備のプロープ	44
4.8.2	シャットダウン制御	44
4.8.3	フェイルオーバー	45

4.9	Scheduling	45
4.9.1	ポッドアフィニティとアンチアフィニティ	46
4.9.2	<code>nodeSelector</code> によるノードの選択	47
4.9.3	許容値	48
4.10	Resource management	48
4.11	Failure Modes	50
4.11.1	ストレージスペースの使用	50
4.11.2	失敗モード	50
4.11.3	自己修復	52
4.11.4	手動介入	52
4.12	Rolling Updates	53
4.12.1	自動更新 (<code>unsupervised</code>)	53
4.12.2	手動更新 (<code>supervised</code>)	54
4.13	Replication	54
4.13.1	アプリケーションレベルのレプリケーション	55
4.13.2	非常に成熟した技術	55
4.13.3	PostgreSQL クラスター内でのレプリケーション	55
4.13.4	外部 PostgreSQL クラスターからのレプリケーション	57
4.14	Backup and Recovery	60
4.14.1	クラウドプロバイダーのサポート	61
4.14.2	オンデマンドバックアップ	68
4.14.3	スケジュールされたバックアップ	69
4.14.4	WAL アーカイビング	70
4.14.5	回復	71
4.14.6	保持ポリシー	72
4.14.7	圧縮アルゴリズム	73
4.14.8	バックアップオブジェクトのタグ付け	73
4.15	PostgreSQL Configuration	74
4.15.1	<code>postgresql</code> セクション	74
4.15.2	<code>pg_hba</code> セクション	79
4.15.3	設定の変更	81
4.15.4	動的共有メモリ設定	81
4.15.5	固定パラメーター	82
4.16	Operator configuration	84
4.16.1	利用可能オプション	85
4.16.2	演算子構成マップの定義	86
4.16.3	演算子シークレットの定義	86
4.16.4	構成をリロードするために演算子を再起動する	87
4.16.5	PPROF HTTP サーバー	87
4.17	Storage	88
4.17.1	CloudNativePG のベンチマーク	89
4.17.2	永続的なボリューム要求	89
4.17.3	ストレージクラスを介した設定	89
4.17.4	PVC テンプレートを介した設定	90
4.17.5	ボリューム拡張	91
4.18	Labels and annotations	94
4.18.1	前提条件	94

4.18.2	クラスターのメタデータの定義	95
4.18.3	現在の制限	95
4.19	Monitoring	95
4.19.1	インスタンスの監視	95
4.19.2	演算子の監視	107
4.20	Logging	107
4.20.1	オペレーターログ	107
4.20.2	PostgreSQL ログ	108
4.20.3	PGAudit ログ	109
4.20.4	その他のログ	110
4.21	Certificates	111
4.21.1	オペレーター管理モード	111
4.21.2	ユーザー提供の証明書モード	113
4.22	Client TLS/SSL Connections	117
4.22.1	新しい証明書の発行	117
4.22.2	TLS 証明書を介した接続のテスト	118
4.23	Connecting from an application	120
4.23.1	DNS 解決	121
4.23.2	環境変数	121
4.23.3	秘密	121
4.24	Connection Pooling	122
4.24.1	アーキテクチャ	122
4.24.2	クイックスタート	123
4.24.3	プーラーリソースのライフサイクル	124
4.24.4	セキュリティ	124
4.24.5	認証	125
4.24.6	PodTemplates	127
4.24.7	高可用性 (HA)	131
4.24.8	PgBouncer 設定オプション	132
4.24.9	監視	133
4.24.10	ロギング	134
4.24.11	接続の一時停止	134
4.24.12	制限	135
4.25	Kubernetes Upgrade	135
4.25.1	reusePVC が false に設定された単一インスタンスクラスター	137
4.26	Exposing Postgres Services	137
4.26.1	Minikube でのテスト	139
4.27	CloudNativePG Plugin	140
4.27.1	インストール	140
4.27.2	使用	141
4.28	Automated failover	151
4.28.1	RTO および RPO のインパクト	152
4.29	Troubleshooting	152
4.29.1	スタート前に	152
4.29.2	オペレーター情報	154
4.29.3	クラスター情報	155
4.29.4	ポッド情報	156

4.29.5	バックアップ情報	159
4.29.6	ストレージ情報	159
4.29.7	ノード情報	159
4.29.8	条件	160
4.29.9	いくつかの一般的な問題	161
4.30	Fencing	162
4.30.1	インスタンスをフェンスする方法	162
4.30.2	フェンシングを解除する方法	163
4.30.3	フェンシングの仕組み	163
4.31	End-to-End Tests	164
4.32	Container Image Requirements	165
4.32.1	イメージタグの要件	166
4.33	Operator Capability Levels	166
4.33.1	レベル 1-基本インストール	167
4.33.2	レベル 2-シームレスなアップグレード	171
4.33.3	レベル 3-完全なライフサイクル	172
4.33.4	レベル 4-深い洞察	175
4.33.5	レベル 5-オートパイロット	176
4.34	Configuration Samples	177
4.35	Commercial support	177
4.36	AffinityConfiguration	178
4.37	AzureCredentials	179
4.38	バックアップ	179
4.39	BackupConfiguration	180
4.40	BackupList	180
4.41	BackupSource	181
4.42	BackupSpec	181
4.43	BackupStatus	181
4.44	BarmanObjectStoreConfiguration	183
4.45	BootstrapConfiguration	184
4.46	BootstrapInitDB	184
4.47	BootstrapPgBaseBackup	185
4.48	BootstrapRecovery	185
4.49	証明書の構成	186
4.50	証明書ステータス	187
4.51	クラスター	187
4.52	ClusterCondition	188
4.53	ClusterList	188
4.54	ClusterSpec	189
4.55	ClusterStatus	191
4.56	ConfigMapKeySelector	192
4.57	ConfigMapResourceVersion	193
4.58	DataBackupConfiguration	193
4.59	EmbeddedObjectMetadata	194
4.60	ExternalCluster	194
4.61	GoogleCredentials	195
4.62	InstanceID	195

4.63	LDAPBindAsAuth	195
4.64	LDAPBindSearchAuth	195
4.65	LDAPConfig	196
4.66	LocalObjectReference	196
4.67	MonitoringConfiguration	197
4.68	NodeMaintenanceWindow	197
4.69	PgBouncerIntegrationStatus	197
4.70	PgBouncerSecrets	198
4.71	PgBouncerSpec	198
4.72	PodMeta	199
4.73	PodTemplateSpec	199
4.74	プーラー	200
4.75	PoolerIntegrations	200
4.76	PoolerList	200
4.77	PoolerSecrets	201
4.78	PoolerSpec	201
4.79	PoolerStatus	201
4.80	PostgresConfiguration	202
4.81	RecoveryTarget	202
4.82	ReplicaClusterConfiguration	203
4.83	RollingUpdateStatus	203
4.84	S3Credentials	203
4.85	ScheduledBackup	204
4.86	ScheduledBackupList	204
4.87	ScheduledBackupSpec	205
4.88	ScheduledBackupStatus	205
4.89	SecretKeySelector	206
4.90	SecretVersion	206
4.91	SecretsResourceVersion	206
4.92	StorageConfiguration	207
4.93	WalBackupConfiguration	207
4.94	Supported releases	208
4.94.1	サポートポリシー	209
4.94.2	命名スキーム	209
4.94.3	CloudNativePG リリースのサポート状況	210
4.94.4	今後のリリース	210
4.95	Release notes	210
4.95.1	バージョン 1.15.0	210
4.95.2	バージョン 1.14.0	211
4.95.3	バージョン 1.13.0	212
4.95.4	バージョン 1.12.0	213
4.95.5	バージョン 1.11.0	214
4.95.6	バージョン 1.10.0	216
4.95.7	バージョン 1.9.2	217
4.95.8	バージョン 1.9.1	217
4.95.9	バージョン 1.9.0	218
4.95.10	バージョン 1.8.0	218

4.95.11	バージョン 1.7.1	219
4.95.12	バージョン 1.7.0	220
4.95.13	バージョン 1.6.0	221
4.95.14	バージョン 1.5.1	221
4.95.15	バージョン 1.5.0	221
4.95.16	バージョン 1.4.0	223
4.95.17	バージョン 1.3.0	224
4.95.18	バージョン 1.2.1	224
4.95.19	バージョン 1.2.0	224
4.95.20	バージョン 1.1.0	225
4.95.21	バージョン 1.0.0	225
4.96	Frequently Asked Questions (FAQ)	226
4.96.1	Kubernetes で PostgreSQL を実行する	226
4.96.2	高可用性	228
4.96.3	バックアップと復元	228
4.96.4	その他	229
4.96.5	アップデート	229
4.96.6	データベース管理	230

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* **内で実行する** クラスター上の *PostgreSQL* の用語 ワークロードを管理するため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。

Cluster という新しい Kubernetes リソースを定義します。これは、高可用性と読み取り専用クエリのオフロードのために、選択した Kubernetes 名前空間と共存する単一のプライマリとオプションの数のレプリカで構成される PostgreSQLcluster を表します。

同じ Kubernetes クラスターにあるアプリケーションは、演算子のみが管理するサービスを使用して、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、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 または Web アプリケーションを介して Postgres を公開するために Service または Ingress オブジェクトを構成する必要があります、PgBouncer に基づくネイティブコネクションプーラを利用できます。

CloudNativePG は元々 *ScheduledBackup* によって構築され、Apache ライセンス 2.0 の下でオープンソースをリリースし、2022 年 4 月に CNCF Sandbox に提出されました。 [source code repository is in Github](#) 。

注釈: Operator Capability Levels model に基づいて、ユーザーは CloudNativePG オペレーターから**「レベル V-自動パイロット」**機能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 1.19 以降が必要です。詳細については、[Supported releases](#) 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（PostgreSQL）の両方のコンテナイメージを保持します。

CloudNativePG 演算子コンテナイメージは `distroless` であり、`cloudnative-pg` で使用できます。

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

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

第3章 メインな機能

高可用性のための *Kubernetes API* サーバーとの直接統合、外部ツールを必要とせずに自己修復機能：最も整合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フェイルオーバーレプリカの自動作成選択したレプリカを昇格させることにより、プライマリインスタンスの計画的な切り替えスケールアップ/スケールダウン機能任意の数のインスタンスの定義（最小 1-1 つのプライマリーサーバ）アプリケーションをクラスターの唯一のプライマリーサーバに接続するための*読み書きサービスの定義読み取り専用サービスの定義。ワークロードを読み取るためにアプリケーションを任意のインスタンスに接続します。特定の一般的なものを含む、*PostgreSQL* 構成の宣言的管理 クラスター `spec` を介した *Postgres* 拡張：`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ポッド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更新演算子のアップグレードのためのインプレース更新またはローリング更新 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 (AWS S3 および S3 互換、Azure Blob Storage、Google Cloud Storage) への継続的なバックアップバックアップ保存ポリシー（リカバリウィンドウに基づく）オブジェクトストア内の既存のバックアップからの完全リカバリおよびポイントインタイムリカバリデータベースが WAL に対応できるようにする並列 WAL アーカイビングおよび復元高書き込みシステムでの生成オプションを有効にするために、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オブジェクトストアレイヤーでの保持管理マルチプル *Kubernetes* にわたる *PostgreSQL* の展開クラスター、プライベート、パブリック、ハイブリッド、およびマルチクラウドアーキテクチャを実現同期レプリカのサポート*PgBouncerを使用した接続プーリング*nodeSelector によるノードアフィニティのサポート*metrics ポートを介した Prometheus のユーザー定義メトリックのネイティブのカスタマイズ可能なエクスポーター (9187) *JSON フォーマットの *PostgreSQL* エラーメッセージの標準出力ロギングポッドの `readOnlyRootFilesystem` セキュリティコンテキストを自動的に設定 *kubect1* の *cnpg* プラグイン**PostgreSQL* クラスター全体、またはインスタンスのサブセットのフェンシングシンプルなバインドおよび検索+バインド LDAP クライアント認証マルチアーチ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 Kubernetes クラスターで CloudNativePG をテストします。

Kubernetes および PostgreSQL の基本的な用語になじみがない場合は、*Before You Start* を参照してください。

Postgres, PostgreSQL and the Slonik Logo は、カナダ PostgreSQL Community

Association の商標または登録商標であり、パーミッションを得て使用しています。*

4.1 Before You Start

始める前に、Kubernetes および PostgreSQL に固有の用語を検討することが不可欠です。

4.1.1 Kubernetes の用語

Resource	Description
[Node](https://kubernetes.io/docs/concepts/architecture/nodes/)	*ノード*は Kubernetes の仮想マシンまたは物理的マシンのワーカーマシンで、ポッドの実行に必要なすべてのサービスはコントロールプレーンノードによって管理されます。
[Pod](https://kubernetes.io/docs/concepts/workloads/pods/pod/)	pod *は、Kubernetes クラスターに展開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。
[Service](https://kubernetes.io/docs/concepts/services-networking/service/)	*サービス*は、ポッドのグループで実行されるアプリケーションをネットワークサービスとして公開し、アプリケーション全体のサービス検出、負荷分散、フェイルオーバーなどの重要な機能を標準化する抽象化です。
[Secret](https://kubernetes.io/docs/concepts/configuration/secret/)	*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。

次のページに続く

表 1 – 前のページからの続き

Resource	Description
[Storage Class](https://kubernetes.io/docs/concepts/storage/storage-classes/)	*ストレージクラス*を使用すると、管理者は、プロビジョニング担当者 (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。
[Persistent Volume](https://kubernetes.io/docs/concepts/storage/persistent-volumes/)	永続ボリューム* (PV) は、管理者が手動でプロビジョニングしたストレージ、または*ストレージクラス*コントローラが動的にプロビジョニングし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とは無関係です。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的なボリュームです。
[Persistent Volume Claim](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#persistentvolumeclaims)	*永続ボリューム要求* (PVC) 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を含むストレージの要求を表します。ポッドがノードリソースを消費する方法と同様に、PVC は PV のリソースを消費します。
[Namespace](https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/)	*名前空間*は、 <i>Kubernetes</i> クラスターのロジカルかつ分離されたサブセットであり、より広い物理的クラスター内の*仮想クラスター*として見るができます。名前空間を使用すると、管理者はプロジェクト、部門、チームなどに基づいて個別の環境を作成できます。
[RBAC](https://kubernetes.io/docs/reference/access-authn-authz/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、ネットワークおよびシステムの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するためのコンピュータシステムセキュリティで 사용되는メソッドです。 <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、特定のリソースや個人に関連付けるためのネイティブ API があります。
[CRD](https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/)	*カスタムリソース定義* (CRD) は <i>Kubernetes</i> API の拡張であり、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。
[Operator](https://kubernetes.io/docs/concepts/extend-kubernetes/operator/)	*演算子*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演算子によって通常実行されるステップを自動化するカスタムリソースです。演算子は <i>Kubernetes</i> を支援して、リソースの定義された状態が常に観察された状態と一致するようにします。

次のページに続く

表 1 – 前のページからの続き

Resource	Description
[kubectl](https://kubernetes.io/docs/reference/kubectl/overview/)	`kubectl` は、Kubernetes クラスターの管理に使用されるコマンドラインツール。

CloudNativePG には Kubernetes 1.19 以降が必要です。

4.1.2 PostgreSQL の用語

Resource	Description
Instance	「IP アドレス」と「TCP ポート」のペア（通常 5432）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。
Replica	クラスター内の唯一のプライマリインスタンスからレプリケートする PostgreSQL インスタンスは、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で更新され続けます。レプリカは、 *スタンバイ* または *二次的な* サーバーとも呼ばれます。PostgreSQL は、物理的ストリーミングレプリケーション（async / sync）とファイルベースのログ SHIPPING（async）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ることができる PostgreSQL 機能。
Cluster	高可用性（HA）クラスターとしての使用：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	通常は Kubernetes クラスターの外部にある、選択した PostgreSQL クラスターからの連続リカバリモードの CloudNativePG <i>Cluster</i> 。これは、プライベート、パブリック、ハイブリッド、およびマルチクラウドコンテキストでのマルチクラスター展開を可能にする機能です。
Designated Primary	別の PostgreSQL クラスターから継続的にリカバリされ、レプリカクラスターがプライマリになる場合にプライマリになるように指定されているレプリカクラスター内の PostgreSQL スタンバイインスタンス。
Superuser	PostgreSQL では、 *スーパーユーザ* は、 <code>LOGIN</code> と <code>SUPERUSER</code> の両方の特権を持つ任意のロールです。セキュリティ上の理由から、CloudNativePG は、ローカル Unix ドメインソケットを介して <code>peer</code> 認証を介して <code>postgres</code> ユーザとして <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 クラウドの用語

Resource	Description
Region	クラウド内の*リージョン*は、*可用性ゾーン*で編成された独立した独立した地理的領域です。リージョン内のゾーンには、往復のネットワークレイテンシがほとんどありません。
Zone	クラウド内の*可用性ゾーン* (*ゾーン*とも呼ばれる) は、リソースを展開できる領域内の領域です。通常、可用性ゾーンはデータセンターまたは同じデータセンターの隔離された建物に対応します。

4.1.4 次にすること

用語を理解したので、選択したクラウド環境に演算子をデプロイする前に `test CloudNativePG on your laptop using a local cluster` を決定することができます。

4.2 Use cases

CloudNativePG は、完全なクラウドネイティブ体験のために、同じ Kubernetes クラスターに存在するアプリケーション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

典型的なシチュエーションでは、アプリケーションとデータベースは Kubernetes クラスター内の同じ名前空間で実行されます。

通常、ステートレスなアプリケーションは、マルチプル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ている標準の Deployment として管理されます。

サービスは、Ingress およびプロバイダーのロード機能を介して、HTTPS 経由でエンドユーザ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 PostgreSQL データベースを使用して、信頼できる永続的な方法で状態を追跡します。アプリケーションは、TLS 接続を介して現在のプライマリインスタンスを指す CloudNativePG によって定義された Cluster リソースによって公開される読み取り/書き込みサービスを参照します。Cluster リソースには、単一のプライマリアーキテクチャと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ているため、Postgres で高可用性クラスターを管理する複雑さが隠されています。

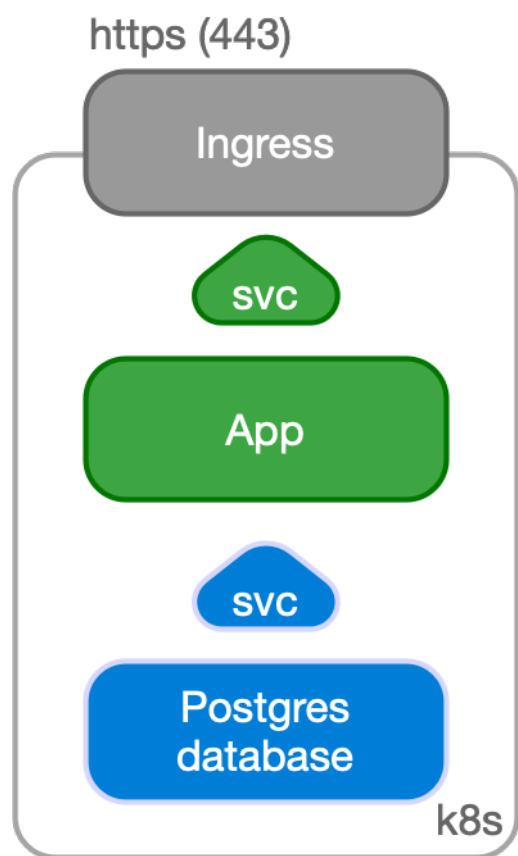


図 1: Application and Database inside Kubernetes

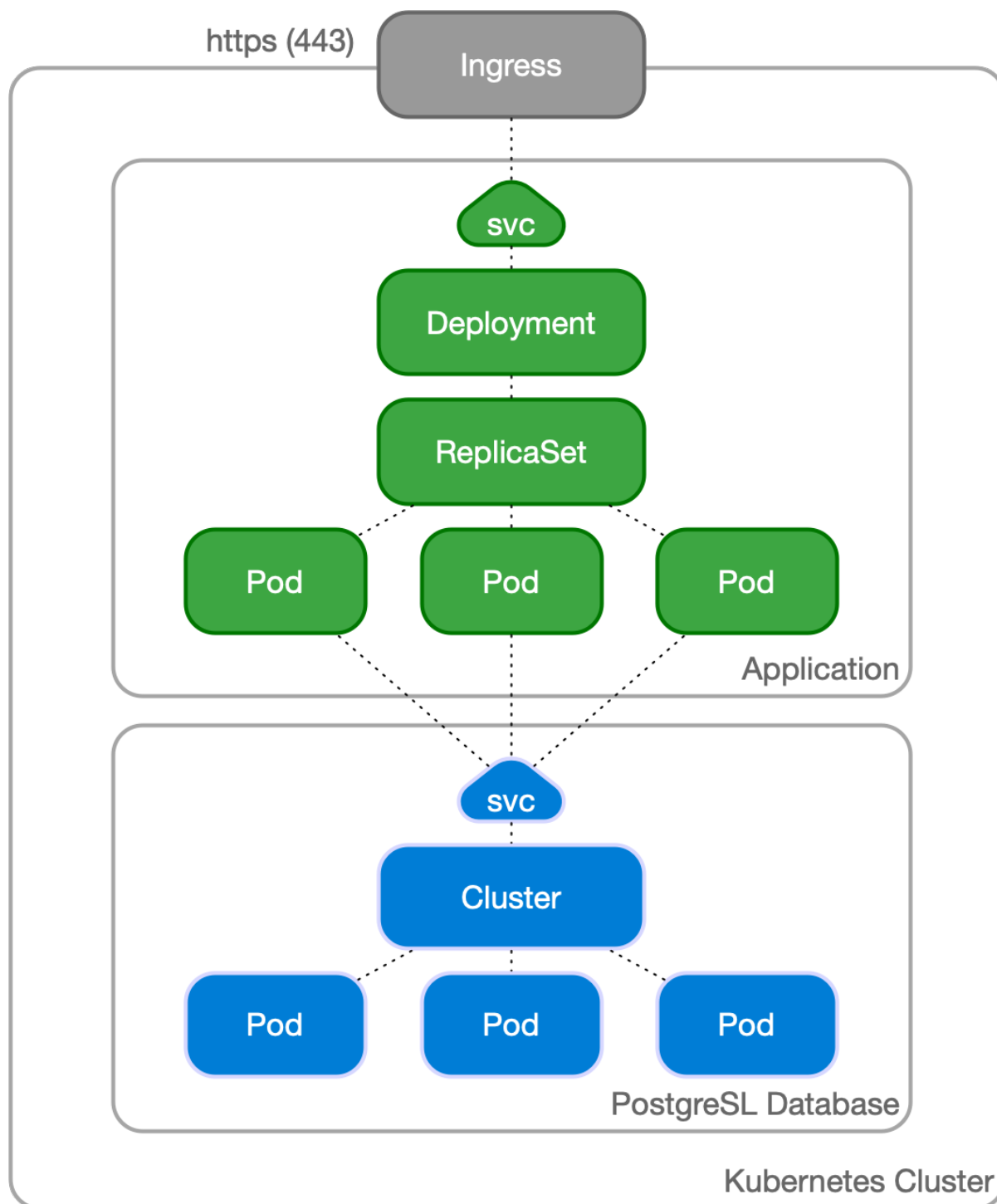


図 2: Close-up view of application and database inside Kubernetes

4.2.2 ケース 2 : Kubernetes 外のアプリケーション

もう 1 つの可能なユースケースは、Kubernetes 内で PostgreSQL データベースを管理し、外部にアプリケーションを配置することです（例、仮想化環境）。この場合、PostgreSQL は Kubernetes で定義された Ingressresource に対応する IP アドレス（またはホスト名前）と TCP ポートで表されます。

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

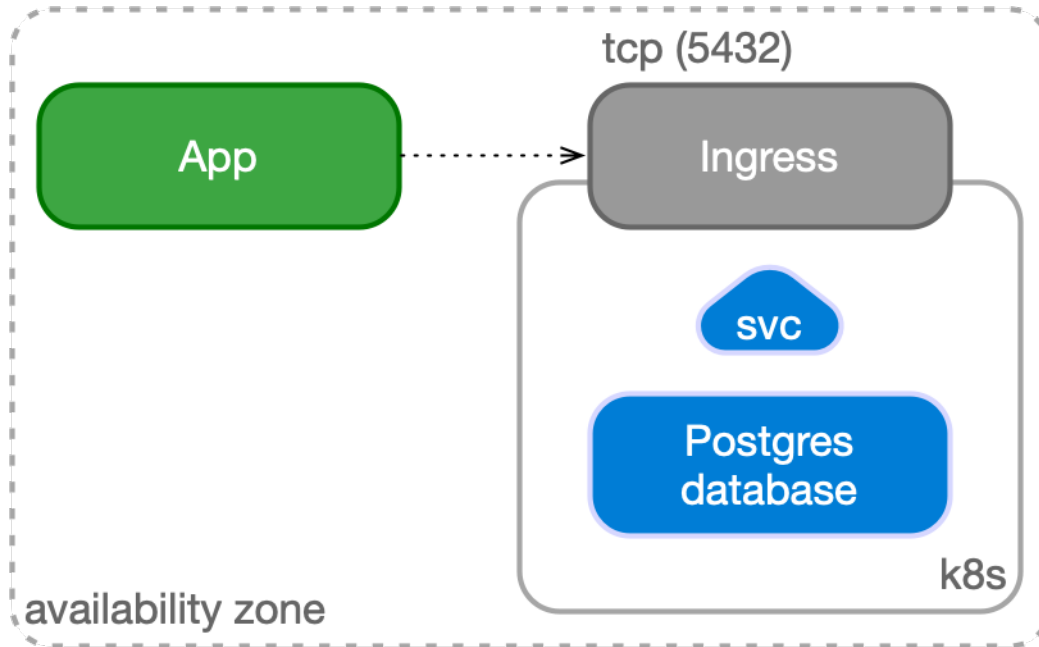


図 3: Application outside Kubernetes

4.3 Architecture

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、管理者にビルトイン物理的レプリケーション **WriteAheadLog (WAL)** シッピングに基づく機能を提供します。

PostgreSQL は、ネットワークの非同期および同期的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、例、WAL ファイルをオブジェクトストアに格納するためのフォールバックオプションとして使用されます）をサポートしています。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、非同期および同期ストリーミングレプリケーションに基づくクラスターをサポートし、同じ Kubernetes クラスター内のマルチプルのホットスタンバイレプリカを管理します。以下の仕様があります。

- 1 つのプライマリ、高可用性のためのオプションのマルチプル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します
- **-ro** : アプリケーションは読み取り専用ワークロードの唯一のホットスタンバイレプリカに接続します

- `-r` : アプリケーションは、読み取り専用ワークロードのインスタンスのいずれかに接続します
- PostgreSQL クラスターの復元力を高めるために、シェアードナッシングアーキテクチャを推奨:
- PostgreSQL インスタンスは異なる Kubernetes ワーカーノードに存在する必要がありますネットワークのみを共有します
- PostgreSQL インスタンスは別の場所に存在できます同じ地域のアベイラビリティゾーン
- PostgreSQL クラスターのすべてのノードは同じリージョンに存在する必要があります

4.3.1 読み取り/書き込みワークロード

アプリケーションは、次の図に示すように、Kubernetes 演算子によって* current 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。

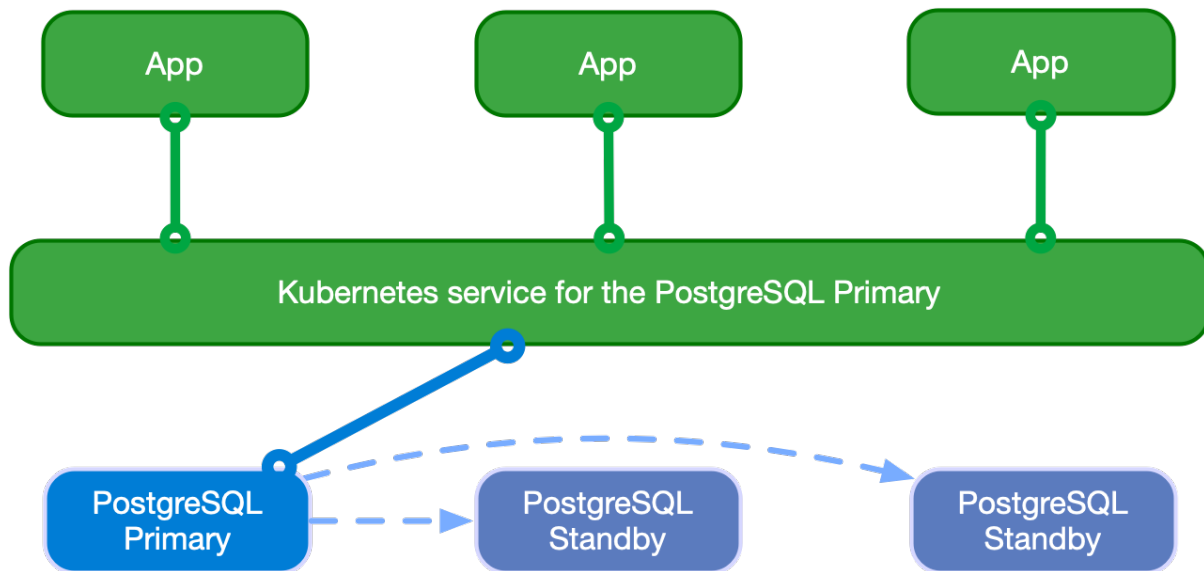


図 4: Applications writing to the single primary

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利用できない場合、Kubernetes は、高可用性を目的として、`-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 読み取り専用のワークロード

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 の動作方法に精通している必要があります。

アプリケーションは、演算子が使用可能にした `-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。

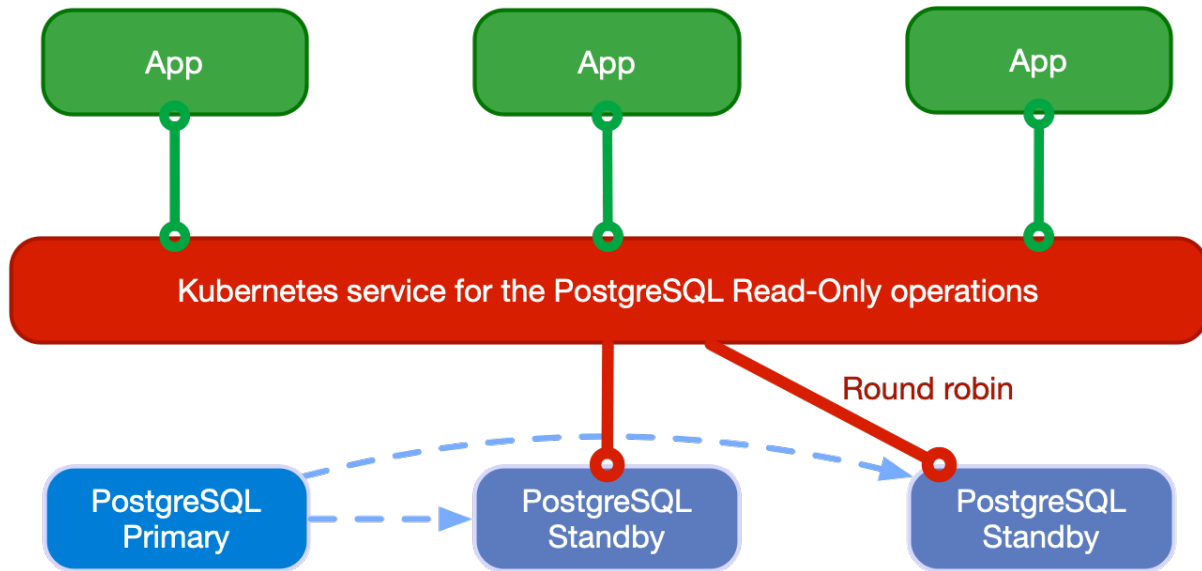


図 5: Applications reading from hot standby replicas in round robin

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単一の PostgreSQL インスタンスのみが存在できます。これは、アプリケーションがいつでも単一の Kubernetes クラスター内にしか書き込みできないことを意味します。

ちなみに: 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、[BDR \(Bi-Directional Replication\) by EDB](#) をご覧ください。Kubernetes の場合、BDR には独自のオペレーターがあり、2022 年後半に予定されています。

ただし、ビジネス継続性の目標では、次のことが基本です。

PostgreSQL バックアップデータを保存することにより、グローバルリカバリポイント目標 (RPO) を削減。マルチプルの場合、地域、場合によっては異なるプロバイダーを使用 (ディザスタリカバリー) PostgreSQL を活用して、グローバルなリカバリ時間目標 (RTO) を削減。プライマリ Kubernetes クラスターを超えるレプリケーション (高可用性)

上記の懸念に対処オーダーに、CloudNativePG は* PostgreSQL Replica Cluster *の概念を導入しています。レプリカクラスターは、プライベート、パブリック、ハイブリッド、およびマルチクラウドコンテキストでのマルチクラスター展開を可能にする CloudNativePG の方法です。

レプリカクラスターは、個別の Cluster リソースです。

1. 定義済みの外部ソースクラスター 2 からの `pg_basebackup` またはフル `recovery` を `bootstrap` オプションとして持つ。 `replica.enabled` オプションをに設定します。通常、Kubernetes クラスターの外

部にある `replica.source` で識別される定義済みの外部クラスターから複製します 4。(PostgreSQL の `restore_command` パラメータを使用して) リカバリオブジェクトストアから受信した WAL 情報を再生する、(PostgreSQL の `primary_conninfo` パラメータを使用して) ストリーミングレプリケーションを介して、または 2 つのいずれか (外部クラスターで `barmanObjectStore` と `connectionParameters` の両方が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:

別のクラスター (`externalClusters` セクションで定義) から PostgreSQL クラスターを複製する方法の詳細については、 [BootstrapConfiguration](#) を参照してください。

次の図は、2 つの異なる Kubernetes クラスターにまたがる PostgreSQL クラスターを示しています。プライマリクラスターは最初の Kubernetes クラスターにあり、レプリカクラスターは 2 番目にあります。2 番目の Kubernetes クラスターは、企業の災害リカバリクラスターとして機能し、災害および最初のクラスターが利用できなくなった場合にアクティブ化する準備ができています。

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) 内の任意の数のカスケードスタンバイサーバを持つスタンバイサーバです。

指定されたプライマリはいつでも昇格させることができ、レプリカ clustera プライマリクラスターが書き込み接続を受け入れることができますようにします。

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。このようなオペレーションは、手動で実行するか、マルチクラスター/統合クラスター対応の権限に委任する必要があります。PostgreSQL クラスターはそれぞれ独立しています。

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

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

4.4 Installation and upgrades

4.4.1 Kubernetes でのインストール

演算子マニフェストを直接使用する

演算子は、`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-
  →1.15.0.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
```

ヘルムチャートの使用

演算子は、提供されている [ヘルムチャートの使用](#) 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、演算子はデフォルトで `cnpg-controller-manager` という Kubernetes Deployment として `cnpg-system` 名前空間にインストールされます。以下を実行することにより、詳細情報を取得できます。

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

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

演算子レベルで高可用性が必要な場合は、オペレーターがリーダー選出をサポートしているため、展開構成でマルチプルのレプリカを指定することができます。また、汚染と許容を `make` して、実際の PostgreSQL クラスターが実行されているのと同じノードで演算子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 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: 一部のバージョンでは追加の手順が必要になる場合があるため、アップグレードを実行する前に [Release notes](#) を注意深くお読みください。

CloudNativePG 演算子のアップグレードは 2 段階のプロセスです。

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

リリースノートで特に明記されていない限り、最初のステップは通常、プレーンな Kubernetes インストールに新しいバージョンのマニフェストを適用するか、使用されているディストリビューションのネイティブパッケージマネージャを使用して行われます（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、コントローラを更新した後に自動的に実行されます。デフォルトでは、デプロイされたすべての PostgreSQL インスタンスのローリング更新をトリガーして、新しいインスタスマネージャを使用します。ローリング更新プロシージャは、スイッチオーバーで終了します。スイッチオーバーは、デフォルトで `unsupervised` に設定された `primaryUpdateStrategy` オプションによって制御されます。`supervised` に設定されている場合、ユーザーは `kubectl` の `cnpg plugin` を使用して新しいインスタンスを手動でプロモートすることにより、ローリング更新を完了する必要があります。

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

バージョン 1.10.0 以降、ローリング更新の動作は、インスタスマネージャのインプレース更新に置き換えることができます。後者は、PostgreSQL インスタンスのリスタートを必要とせず、その結果、クラスターでのスイッチオーバーが必要です。この動作は、デフォルトでは無効になっていますが、以下で説明します。

インスタスマネージャのインプレース更新

デフォルトでは、CloudNativePG は、演算子が更新されるたびにクラスターのローリング更新を発行します。オペレーターとともに出荷される新しいインスタスマネージャは、`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、インスタスマネージャのインプレース更新を有効にするように構成によって変更できます。インスタスマネージャは、コンテナを存続させる PID 1 プロセスです。

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

その結果、PostgreSQL プロセスは更新による影響を受けず、スイッチオーバーを実行する必要がありません。コインのもう一方の側面は、ポッドがスタート後に変更され、不変性の純粋な概念を破ることです。

この機能を有効にするには、*Operator configuration* で `ENABLE_INSTANCE_MANAGER_INPLACE_UPDATES` environment 変数を `'true'` に設定します。

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

重要: この機能を使用するには、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/architecture（例、すべて `linux/amd64`）で実行されている必要があります。

バージョン間の互換性

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

新しい機能に加えて、演算子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 (stable) Postgres 環境を維持するオーダーに各バージョンがリリースされるため**、演算子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、毎月少なくとも演算子の新しいバージョンをリリースしています。各バージョンが使用可能になったときに更新を適用できない場合は、バージョンをバージョンにアップグレード処理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不可能な環境で CloudNativePG の LTS リリースを計画しています。

[Release notes](#) ページには、CloudNativePG のすべてのリリースバージョンで導入された変更の詳細なリストが含まれており、ソフトウェアの新しいバージョンにアップグレード処理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、単純な Kubernetes インストールに `newermanifest` を適用するか、選択したディストリビューションのネイティブパッケージマネージャーを使用するだけで十分です。

バージョンを直接アップグレードできない場合は、新しいバージョンをインストールする前に古いバージョンを削除するニーズがあります。これはユーザーデータには影響しませんが、演算子自身自身にのみ影響します。

4.5 Quickstart

このセクションでは、[種類](#) または [Minikube でのテスト](#) のローカル Kubernetes クラスターで CloudNativePG を使用して、ラップトップ/コンピューターで PostgreSQL クラスターをテストする方法について説明します。

警告: このセクションに含まれる手順は、デモンストレーション、テスト、および実践のみを目的としたものであり、稼動環境では使用しないでください。

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

このページの指示に従うことで、ローカル Kubernetes インストールで PostgreSQLcluster をスタートし、それを試すことができます。

重要: Kubernetes クラスターに接続オーダーには、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 の文書に従ってください。

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

最初のパートは、Minikube または Kind のインストールについてです。システムについて時間をかけて読んで、どちらのシステムに進むかを決めてください。いずれかのシステムをセットアップしたら、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。Minikube は、Kubernetes を毎日試したり開発したりするために、ラップトップ上の仮想マシン（VM）内で単一ノード Kubernetes クラスターとして実行されます。通常、VirtualBox と組み合わせて使用します。

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

```
minikube start
```

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

```
kubectl get nodes
```

minikube という 1 つのノードが表示されます。

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（Kind は「Kubernetes IN Docker」の略です）。

クイックスタート の指示に従って、環境に kind

をインストールしてから、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ているので、CloudNativePG のインストールを続行できます。

Installation and upgrades セクションを参照して、

PostgreSQL クラスターの展開を進めてください。

4.5.3 パート 3- PostgreSQL クラスターをデプロイする

Kubernetes の他の展開と同様に、PostgreSQL クラスターを展開するには、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スペースを割り当てるシンプル

Cluster を定義しています。

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

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

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

3 ノードの PostgreSQL クラスターを作成するには、オーダー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、演算子は、リリースされたときに PostgreSQL の最新のメジャーバージョンの利用可能な最新のマイナーバージョンをインストールし演算子。これは、Cluster 定義の spec セクションで `imageName` キーを設定することでオーバーライドできます。例、PostgreSQL 13.6 をインストールするには：

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

重要： 不変のインフラストラクチャパラダイムでは、コンテナイメージの特定のバージョンを常にポイントする必要があります。実稼動環境では、*latest* や *13* などのタグを使用しないでください。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が発生する可能性があります。厳密で確定的で反復可能なデプロイメントの場合、`<image>:<tag>@sha256:<digestValue>` フォーマットを使用して、イメージ名前にダイジェストを追加できます。

4.6 Bootstrap

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、その背後にあるデザイン原理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`）

重要： 既存のクラスターからのブートストラップにより、**レプリカクラスター**、つまり継続的にリカバリし、ソースと同期し、読み取り専用接続を受け入れる独立した PostgreSQL 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` ユーザとデータベースの両方が常に存在する必要があります。ローカル Unix ドメインソケットを使用して、クラスターで管理タスクを実行オーダーには、`postgres` ユーザとして `peer` 認証を介して `postgres` データベースに接続するニーズがあります。** `postgres` ユーザまたは `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

- ブートストラップ*メソッドは、clusterspecification.CloudNativePG の `bootstrap` セクションで定義できます。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` : 空の PostgreSQL クラスターを初期化します（デフォルト）
- `recovery` : 既存のクラスターから復元して PostgreSQL クラスターを作成しますバックアップオブジェクトストア経由で、利用可能なすべての WAL ファイルまたは最大で指定された**特定の時点**
- `pg_basebackup` : の既存のものを複製して PostgreSQL クラスターを作成しますストリーミングレプリケーションプロトコル経由で `pg_basebackup` を使用する同じメジャーバージョン- データベースを CloudNativePG に移行する場合に便利です。Kubernetes の外部から。

`initdb` メソッドとは異なり、`recovery` と `pg_basebackup` の両方が別のクラスター（オフラインまたはオンライン）に基づいて新しいクラスターを作成し、レプリカクラスターの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` セクションでは、現在のクラスターに何らかの形で関連する 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はより複雑なシナリオを可能にしますが、現在は物理的レプリケーションに基づいてクロスリージョン PostgreSQL クラスターを定義し、異なる Kubernetes クラスターまたは従来の VM /ベアメタル環境にまで及ぶことを意図しています。

ブートストラップに関する限り、`externalClusters` を使用して、`pg_basebackup method` または `recovery` のいずれかのソース PostgreSQL クラスターを定義できます。外部クラスターには次のものがニーズです。

- を介してリファレンスとして使用される、オリジンクラスターを識別する名前 `source` オプション
- 次の少なくとも 1 つ：
- ストリーミング接続に関する情報- リカバリオブジェクトストアに関する情報。これは、ソースクラスターのバックアップファイル（つまり、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す。

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

ストリーミング接続のみが定義されている場合、`pg_basebackup` メソッドにソースを使用できます。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、`recovery` メソッドにソースを使用できます。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 point in time` の場合、クラスターはレプリカクラスターモードに適格です。これは、ストリーミング、WAL シッピング、PostgreSQL の `restore_command`、または 2 つのいずれかを介して、ソースからクラスターが継続的に供給されることを意味します。

4.6.3 空のクラスターのブートストラップ (initdb)

initdb ブートストラップメソッドは、新しい PostgreSQL クラスターをゼロから作成するために使用されます。別に指定しない限り、これはデフォルト。

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とおりです。

1. PostgreSQL のネイティブ initdb コマンドを使用して、新しい PGDATA フォルダを作成します 2. 名前付けのシークレットから postgres **スーパーユーザ**のパスワードを設定します。名前付けの**非特権**ユーザを作成します。 app-secret シークレットのパスワードを使用して、後者 (app) のパスワードを設定します (username が owner と同じ名前に一致 make ことを確認してください) 5. app ユーザが所有する app というデータベースを作成します。

*設定パラダイムよりも規約のおかげで、オペレータにデフォルトデータベース名前 (app) とデフォルトのアプリケーションユーザー名前 (データベース名前と同じ) を選択させ、スーパーユーザとアプリケーションユーザの両方に安全なパスワードをランダムに生成させることができます PostgreSQL で。

あるいは、上記の例で説明したように、パスワードを生成し、秘密として保存し、PostgreSQL クラスターで使用できます。

提供されるシークレットは kubernetes.io/basic-auth の仕様に準拠する必要があります。その結果、シークレットの username は、owner (アプリケーションシークレットの場合) およびスーパーユーザの postgres のいずれかとマッチ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、`applicationdata` を格納するために使用する必要があるデータベースです。アプリケーションは、アプリケーションデータベースを所有するユーザでクラスターに接続する必要があります。

重要: 演算子の将来の実装により、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可能性があります。

`postgres` スーパーユーザと `postgres` データベースは、クラスターを構成するために演算子ののみが使用することになっています。

データベース名前を指定しない場合、演算子は慣例に従い、`app` データベースを作成し、`* defaulting webhook *`を使用してクラスター定義に追加します。データベースを所有するユーザは、代わりにデータベース名前になります。

アプリケーションユーザは、演算子によって内部的に使用されません。オペレーターは、代わりにスーパーユーザに依存して、クラスターを目的のステータスに調整します。

重要: 現時点では、スーパーユーザシークレットの名前の変更はクラスターに適用されません。

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

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

エンコーディング : `encoding` が値に設定されると、CNPG はそれを `initdb` の `--encoding` オプションに渡します。このオプションは、テンプレートデータベースのエンコーディングを選択します（デフォルト : UTF8）。

localeCollate : `localeCollate` が値に設定されると、CNPG はそれを `initdb` の `--lc-collate` オプションに渡します。このオプションは、PostgreSQL 文書の [Locale Support](#) で定義されているように、照合順序オーダー（`LC_COLLATE` サブカテゴリ）を制御します（デフォルト : C）。

localeCTYPE : `localeCTYPE` が値に設定されると、CNPG はそれを `initdb` の `--lc-ctype` オプションに渡します。このオプションは、PostgreSQL 文書の [Locale Support](#) で定義されているように、照合順序オーダー（`LC_CTYPE` サブカテゴリ）を制御します（デフォルト : C）。

`walSegmentSize` : `walSegmentSize` が値に設定されると、CNPG はそれを `initdb` の `--wal-segsize` オプションに渡します（デフォルト：未設定 PostgreSQL により 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は、`LC_COLLATE` および `LC_TYPE` サブカテゴリを参照します。残りのロケールサブカテゴリは、`lc_messages`、`lc_monetary`、`lc_numeric`、および `lc_time` パラメーターを使用して、PostgreSQL 構成で直接構成できます。

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

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

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

CloudNativePG は、`options` サブセクションを使用して、`initdb` 呼び出しの動作をカスタマイズする別の方法をサポートしています。ただし、演算子の動作を壊す可能性のあるオプション（`--auth` または `-d` など）があるため、この手法は廃止され、API の将来のバージョンから削除されます。

データベースを作成して構成した直後に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**（`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
```

(次のページに続く)

(前のページからの続き)

```

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になります。

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バージョンから開始するクラスターのブートストラップを有効にします。このオペレーションは、ストリーミングレプリケーション (*pg_basebackup*) を介してソースクラスターに直接接続するか、リカバリオブジェクトストア (*recovery*) を介して間接的に接続することで実行できます。

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 (リカバリ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 でこの結果を達成するには、2つの方法があります。

- 別のクラスターのバックアップであるリカバリオブジェクトストアを使用する Barman Cloud によって作成され、*barmanObjectStore* オプションを介して定義されます **externalClusters** セクション (推奨)
- 同じ名前空間で既存の Backup オブジェクトを使用する (これはバージョン 1.8.0 より前のオプションのみが利用可能です)。

どちらのリカバリ方法も、完全リカバリ（最後に使用可能な WAL まで）または **ポイントインタイムリカバリ (PITR)** までのいずれかを有効にします。完全リカバリを実行する場合、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: *Backup and Recovery* で実行中のクラスターのバックアップとリカバリに関する詳細情報を見つけることができます。

オブジェクトストアからの回復

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 セクションで必要なすべての構成を含む外部クラスターを定義したら、.spec.recovery.source オプションでそれを参照する必要があります。次の例では、Azure の BLOB コンテナにリカバリオブジェクトストアを定義しています。

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

  superuserSecret:
    name: superuser-secret

  bootstrap:
    recovery:
      source: clusterBackup

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

重要: デフォルトでは、*recovery* メソッドは、*externalClusters* セクションのクラスターの *name* を厳密

に使用して、通常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` プロパティで別のプロパティを指定できます（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、アーカイブから必要な WAL ファイルを同時にフェッチするために最大 8 つのジョブを割り当てています。この機能により、リカバリ時間を大幅に短縮できます。このシナリオを事前に計画し、環境に合わせてこのパラメータの値を正しく調整してください。確かに違いが生じるのは、必要な場合（そうでない場合）です。

Backup オブジェクトからの回復

クラスターを作成する名前空間で Backup リソースが既に利用可能な場合、次の例のように、`ed.spec.bootstrap.recovery.backup.name` で名前を指定できます。

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

  superuserSecret:
    name: superuser-secret

  bootstrap:
    recovery:
      backup:
        name: backup-example

  storage:
    size: 1Gi
```

このブートストラップメソッドでは、復元ニーズがあるバックアップへのリファレンスのみを指定できます。

追加の考慮事項

リカバリオブジェクトストアから回復する場合でも、既存の Backup resource から回復する場合でも、以下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前とアプリケーションデータベースユーザは保持されます復元されるバックアップから。演算子は現在試行していませんこれは通常のメンテナンスのパートであるため、基礎となる秘密をバックアップします Kubernetes クラスタ自分自身のアクティビティ。

- `superuserSecret` を指定しない場合、新しいものが自動的に追加されます安全でランダムパスワードで生成されます。秘密はその後に使用されますクラスターの `postgres` ユーザのパスワードをリセットします。
- デフォルトでは、リカバリは最新まで続行されますデフォルトのターゲットタイムラインで利用可能な WAL (PostgreSQL の場合は `current` まで 11、バージョン 12 以降の場合は `latest`)。オプションで `recoveryTarget` を指定して、特定の時点を実行できますリカバリ ([ポイントインタイムリカバリ \(PITR\)](#) を参照)。

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

ポイントインタイムリカバリ (PITR)

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

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

演算子は、Azure に保存されているリカバリオブジェクトを使用する次の例のように、リカバリターゲットが指定されている場合にこの機能が動作するために必要な構成パラメーターを生成します。

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

  storage:
    size: 5Gi

  bootstrap:
    recovery:
      source: clusterBackup
      recoveryTarget:
        targetTime: "2020-11-26 15:22:00.000000+00"

  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

```

targetTime の他に、次の基準を使用してリカバリを停止できます。

- targetXID は、リカバリを続行するトランザクション ID を指定します
- targetName はリストアポイントを指定します (pg_create_restore_point で作成されます) リカバリの続行先)
- targetLSN は、ログ先行書き込みの場所の LSN を指定しますリカバリが進みます

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

さらに、targetTLI を指定して、特定のタイムラインに強制的にリカバリできます。

デフォルトでは、以前のパラメータ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 コンテナに依存する次の例のように、exclusive パラメータを false に設定して、リカバリターゲットの直後で停止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:
        targetName: "maintenance-activity"
        exclusive: false

  externalClusters:
    - name: clusterBackup

```

(次のページに続く)

```

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

```

ライブクラスタ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ソースインスタンス（ソース）の正確な物理的コピーとして、新しいクラスター（ターゲット）を作成できます。プライマリまたはスタンバイ PostgreSQL サーバー。

このメソッドのプライマリ使用例は、Kubernetes の外部または Kubernetes 内（別の演算子など）から CloudNativePG への移行で表されます。

警告: 現在の実装は、クローン作成プロセスが終了し、作成されたクラスターをすぐに開始するときに、オリジン PostgreSQL インスタンスの*snapshot*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、演算子は最初のインスタンスからターゲットクラスターの所有権を取得します。これには、CloudNativePG で必要とされるいくつかの構成パラメーターのオーバーライド、スーパーユーザパスワードのリセット、streaming_replica ユーザの作成、レプリカの管理などが含まれます。結果のクラスタは、ソースインスタンスから完全に独立します。

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

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

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

CloudNativePG によって管理されるソースまたは TLS 認証用に構成されたソースに接続する場合、後者が推奨されます。ただし、最初のオプションは一般的な PostgreSQL サーバーへの認証の最も一般的なフォー

ムであり、sourceinstance がオンの場合に最も簡単な方法です Kubernetes の外部の従来の環境。両方のケースについて以下で説明します。

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には同じハードウェアアーキテクチャが必要です
- ターゲットとソースには同じメジャー PostgreSQL バージョンが必要です
- ソースにはテーブルスペースを定義してはいけません（下記の [現在の制限](#) を参照）
- ソースは、許可するのに十分な `max_wal_senders` で構成する必要があります少なくとも 1 つを提供することによる、この 1 回限りのオペレーションのためのターゲットからのアクセスバックアップ用に 1 つ* プラス *と WAL ストリーミング用に 1 つ
- ターゲットを有効にするには、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インスタンスの PostgreSQL ポートに接続するインスタンス
- ソースには `REPLICATION LOGIN` 特権を持つロールが必要で、受け入れる必要があります `pg_hba.conf` のこのロールのターゲットインスタンスからの接続、できれば TLS 経由（下記の [レプリケーションユーザについて](#) を参照）
- ターゲットは、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 `REPLICATION LOGIN` 特権を持つロールを使用する

参考:

詳細については、PostgreSQL 文書の [Planning](#)、ライブラスタからのブートストラップ（`\pg_basebackup`）<ライブラスタ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について

要件のセクションで説明したように、ソースインスタンスの `SUPERUSER` またはできれば `REPLICATION privilege` のいずれかを持つユーザが必要です。

ソースデータベースが CloudNativePG で作成されている場合、`streaming_replica` ユーザを再利用し、`clientTLS` 証明書認証（デフォルトでは `streaming_replica` の唯一の許可された接続メソッド）を利用できます。

Kubernetes の外部を含む他のすべての場合、`REPLICATION` 権限を持つユーザが既に存在することを確認するか、以下の手順に従って新しい特権を作成してください。

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

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

パスワードをターゲットインスタンスのシークレットに追加する必要があるため、プロンプトでパスワードを入力し、後で使用するために保存します。

注釈: 名前は重要ではありませんが、簡単にするために `streaming_replica` を使用します。次のセクションの手順を適用する場合は、フリーに変更できます。

ユーザー名/パスワード認証

`pg_basebackup` ブートストラップで CloudNativePG がサポートする最初の認証メソッドは、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。

プロシージャをスタートする前に、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 アドレスで識別されるソースインスタンスの場所および TCP ポート
- 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` file に次のような行を追加する必要がある場合があります。

```
# A more restrictive rule for TLS and IP of origin is recommended
host replication streaming_replica all md5
```

次のマニフェストは、`pg_basebackup` ブートストラップメソッドを使用して、`source-db` (`externalClusters` 配列内) として定義された外部 PostgreSQL クラスターを複製する、`target-db` という新しい PostgreSQL 14.2 クラスターを作成します。ご覧のとおり、`source-db` definition は `source-db.foo.com` ホストをポイントし、`streaming_replica` ユーザとして接続します。このユーザーは、`source-db-replica-user` シークレットの `password` キーにパスワードが保存されています。

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: target-db
spec:
  instances: 3
  imageName: ghcr.io/cloudnative-pg/postgresql:14.2

  bootstrap:
    pg_basebackup:
      source: source-db

  storage:
    size: 1Gi

  externalClusters:
```

(次のページに続く)

(前のページからの続き)

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

同じ PostgreSQL バージョン（この場合は 14.2）を含む、クローンオペレーションが機能するためには、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` ブートストラップで CloudNativePG がサポートする 2 番目の認証メソッドは、TLS クライアント証明書に基づいています。これは、セキュリティの観点から推奨されるアプローチです。

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

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

マニフェストは、`cluster-clone-tls` という新しい PostgreSQL 14.2 クラスターを定義します。これは、`cluster-example` external クラスターから `pg_basebackup` メソッドを使用してブートストラップされます。ホストは、同じクラスター内の読み取り/書き込みサービスによって識別されますが、`streaming_replica` ユーザは、提供されたキー、証明書、および認証機関情報のおかげで（それぞれ `cluster-example-replication` および `cluster-example-ca` シークレットで）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:14.2

  bootstrap:
    pg_basebackup:
      source: cluster-example

  storage:
    size: 1Gi
```

(次のページに続く)

```
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
```

現在の制限

テーブルスペースのサポートがありません

CloudNativePG には現在、PostgreSQL グローバルオブジェクト、つまりロール、データベース、テーブルスペースの完全な宣言的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 が欠落している機能が必要です：追加の永続性の定義と管理ボリューム。ベースバックアップとテーブルスペースを扱う場合、PostgreSQL 自体は、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（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 で直接移行することはできません（最初にテーブルスペースをソースから削除するか、組織がこの機能を必要とする場合は、EDB に連絡して優先順位を付けてください）。

スナップショットコピー

`pg_basebackup` メソッドは、PostgreSQL ベースバックアップのフォームでソースインスタンスの snapshot を取得します。バックアップのスタートからバックアップの正しい終了まで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 のターゲットデータベースに移行する前に、ソースデータベースへのすべての書き込み操作を停止することをお勧めします。

重要: 移行を試みる前に、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、稼働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ために、必要な回数だけ移行プロシージャ

を実行することが基本です。フリーにお問い合わせ EDB。

CloudNativePG の将来のバージョンでは、ユーザーが別の 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、先書きログ (WAL) を介して PostgreSQL の継続的なリカバリメカニズムを制御できるようになります。

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

4.7 Security

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

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

4.7.1 コード

CloudNativePG のソースコードは、CI/CD パイプラインで直接 [GolangCI-Lint](#) と呼ばれる Go の一般的なオープンソースリンターを使用して、[セキュリティ問題](#)を含む静的分析目的で体系的にスキャンされます。GolangCI-Lint はいくつかのリンターを実行できます同じソースコードで。

これらの 1 つは [Golang Security Checker](#)、または単に `gosec`、ソースのアブストラクト構文ツリーをスキャンして、ハードコード資格情報、整数などのコードに隠されている既知の脆弱性、脅威、および脆弱性の発見を目的とした一連のルールに対してスキャンしますオーバーフローと SQL インジェクション-いくつか例を挙げ名前。

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

4.7.2 コンテナ

CloudNativePG のパートであるすべてのコンテナイメージは、コミットごとに CI / CD パイプラインを介して自動的に構築されます。そのようなイメージには、オペレーターだけでなくオペランドも含まれます。具体的には、サポートされるすべての PostgreSQL バージョンです。

- [Docker](#) : 用語のベストプラクティス コンテナ構築プロセスの

重要: ベースイメージおよびパッケージレベルのセキュリティ更新の場合、すべてのオペランドイメージはパイプラインによって 1 日に 1 回自動的に再構築され、EDB が配布するコンテナイメージの**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では、次のガイドラインとフレームワークがアカウントされています。

- [Container Image Creation and Deployment Guide](#)、米国国防総省 (DoD) の防衛情報システム局 (DISA) が開発
- [CIS Benchmark for Docker](#)、インターネットセキュリティセンター (CIS) が開発

4.7.3 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) がアカウントされます。

役割ベースのアクセス制御 (RBAC)

演算子は、`cnpg-manager` という専用のサービスアカウントを使用して Kubernetes API サーバーと対話します。Kubernetes では、これはデフォルトで `cnpg-system` 名前空間にインストールされ、このサービスアカウントと、演算子に付与されたルール/リソース/動詞のセットを定義する `cnpg-manager cluster` ロールとの間のクラスターロールバインドを使用します。

重要: 上記のアクセス許可は、オペレーターのサービスアカウントが Kubernetes API サーバーと対話するために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演算子のユーザーは、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。最も重要なのは、CloudNativePG が標準 Kubernetesnamespaced リソースの完全または部分的な管理を必要とする理由です。

configmaps : 演算子は、Prometheus エクスポーターモニタリングメトリックのデフォルトの構成マップを作成および管理するニーズがあります。

deployments : 演算子は、標準の Kubernetes Deployment リソースを使用して PgBouncer コネクションプーラを管理するニーズがあります。

jobs : 演算子は、異なる Cluster のフェーズを管理するジョブをハンドル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。演算子は、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ニーズがあります。

Pods : 演算子は Cluster のインスタンスを管理するニーズがあります。

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、演算子はランダムに生成されたパスワードと TLS 証明書を自己プロビジョニングし、それらを秘密に保存することにより、「設定より規約」のパラダイムを採用します。

serviceaccounts : 演算子は、インスタンスマネージャ（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、信頼性の高い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ニーズがあります Cluster の。

ニーズ : 演算子は、アプリケーションから PostgreSQL クラスタ（またはコネクションプーラ）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イルオーバー/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（例、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）。

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ニーズのあるセキュリティルールと仕様を定義する Kubernetes の方法です。InfoSec の理由により、すべての Kubernetes プラットフォームはそれらを実装する必要があります。

CloudNativePG 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` システムユーザとして実行されます。 `root` として実行する必要のコンポーネントは一切ありません。

同様に、ボリュームへのアクセスには `* privileges *`モードや `root` 特権も必要ありませんレイヤープラットフォームや管理者によって適切なアクセス許可が適切に割り当てられている必要があります PostgreSQL コンテナは読み取り専用ルートファイルシステムで実行されます

演算子は、必要なセキュリティコンテキストを明示的に設定します。

AppArmor を使用して Pod アクセスを制限する

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

警告: この種類の注釈を使用すると、クラスターの動作が停止する可能性があります。この場合、注釈は *Cluster* から安全に削除できます。

AppArmor 構成は Kubernetes ノードレベルである必要があります。つまり、基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。

これがシチュエーションではなく、注釈が **Cluster** の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** の作成後に注釈を追加すると、クラスター内のポッドはスタートできなくなり、次のようなエラーが表示されます。

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

そのような場合は、Kubernetes 管理者に問い合わせ、使用する適切な AppArmor プロファイルを要求してください。

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: 演算子は、PostgreSQL サーバーのステータスに関する情報を取得するために、TCP ポート 8000 で各インスタンスと通信するニーズがあります。ネットワークポリシーを追加する場合はこれを念頭に置いてください `make` で詳細な制御に使用されるポートのリストについては、以下の「公開ポート」セクションを参照してください。

ネットワークポリシーはこのドキュメントの範囲外です。詳細については、Kubernetes 文書の **ネットワークポリシー** セクションを参照してください。

露出ポート

CloudNativePG は、以下の表にリストされているように、演算子、インスタンス マネージャ、および `operandlevels` でポートを公開します。

System	Port number	Exposing	Name	Certificates	Authentication
operator	9443	webhook server	<i>webhook-server</i>	TLS	Yes
operator	8080	metrics	<i>metrics</i>	no TLS	No
instance manager	9187	metrics	<i>metrics</i>	no TLS	No
instance manager	8000	status	<i>status</i>	no TLS	No
operand	5432	PostgreSQL instance	<i>postgresql</i>	optional TLS	Yes

PostgreSQL

CloudNativePG の現在の実装では、`postgres` スーパーユーザ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、PostgreSQL はデフォルトで `scram-sha-256` に設定されますが、以前のバージョンでは `md5` に設定されます。

重要: 詳細については、PostgreSQL 文書の [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` に設定すると、シークレット経由で `postgres` ユーザパスワードの管理を無効にできます。

注釈: 演算子は、`enableSuperuserAccess` オプションの切り替えをサポートしています。実行中のクラスターでそれを無効にすると、演算子はシークレットのコンテンツを無視し、それを削除し（演算子によって以前に生成された場合）、`postgres` ユーザのパスワードを `NULL` に設定します（事実上のパスワード認証リモートアクセスの無効化）。

詳細については、[PgBouncerSecrets](#) を参照してください。

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

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

現在、演算子は、管理者が `postgresql` 構成の `pg_hba` セクションの `manifests` パートに `pg_hba.conf` 行を直接追加することを許可しています。それらの `anifest` で定義された行は、デフォルトの `pg_hba.conf` に追加されます。

演算子による `pg_hba.conf` の管理方法の詳細については、文書の [PostgreSQL Configuration](#) を参照してください。

重要: の例では、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ることを想定しています。

4.8 Postgres instance manager

CloudNativePG は、フェイルオーバー管理のために外部ツールに依存しません API サーバーと **Postgres インスタンスマネージャ**と呼ばれるネイティブキーコンポーネントに単純に依存しています。

インスタンスマネージャは、PostgreSQL の主要なプロセス (**postmaster** と呼ばれます) のライフサイクル全体を処理します。

新しいクラスターを作成すると、演算子はインスタンスごとにポッドを作成します。フィールド `.spec.instances` は、作成するインスタンスの数を指定します。

各 Pod は、メインコンテナの親プロセス (PID 1) としてインスタンスマネージャをスタートし、メインコンテナは PostgreSQL インスタンスを実行します。Pod の有効期間中、インスタンスマネージャは `liveness and readiness probes` をハンドルするバックエンドとして機能します。

4.8.1 活性と準備のプロープ

`liveness` プロープはプロープに依存していますが、`readiness` プロープはデータベースがリスタートしており、スーパーユーザー資格情報を使用して接続を受け入れることができるかどうかを確認します。

プローブコマンドが各チェック間に 10 秒インターバルで 3 回失敗すると、2 つのプロープは失敗をレポートします。

現時点では、Kubernetes 1.17 でのみ起動プローブが導入されているため、演算子は Pod で `startupProbe` を設定しません。

`liveness` プロープは、PostgreSQL インスタンスが中断状態にあり、再起動ニーズがあるかどうかを検出するために使用されます。`startDelay` の値は、プローブの実行を遅らせるために使用されます。これは、スタートアップ時間が長いインスタンスの再起動を防ぐために使用されます。

ポッドが起動してから `livenessprobe` が動作を開始するまでの秒数は、`.spec.startDelay` パラメータで表され、デフォルトは 30 秒です。クラスターの正しい値は、PostgreSQL のスタートに必要な時間に関連しています。

`.spec.startDelay` が低すぎる場合、PostgreSQL のスタートアップ前に活性プローブが動作をスタートし、Pod が適切に再起動される可能性があります。

4.8.2 シャットダウン制御

Postgres を実行しているポッドが、手動またはノードドレインオペレーションに続いて Kubernetes によって削除されると、`kubelet` はインスタンスマネージャに終了シグナルを送信し、インスタンスマネージャは適切な方法で PostgreSQL をシャットダウンします。は、PostgreSQL がシャットダウンするまでの時間です。値のデフォルトは 30 秒です。

シャットダウンプロシージャは、2 つのステップで構成されます。

1. インスタンスマネージャは **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の間続きます。

2. PostgreSQL がまだ起動している場合、インスタスマネージャは **高速** 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、プロセスは残りの半分まで待機しますオペレーションを完了してから強制的にシャットダウンするために `.spec.stopDelay` に設定された時間

重要: データベースの RPO に影響する Postgres クラスターのデータロス回避オーダーに、プライマリインスタンスが実行されているポッドを削除しないでください。この場合、最初に別のインスタンスへのスイッチオーバーを実行します。

スイッチオーバー中のプライマリのシャットダウン

スイッチオーバー中のシャットダウンプロシージャは、一般的な場合とわずかに異なります。実際、演算子は、すべてのデータが新しいプライマリで利用できることを保証オーダーに、選択された新しいプライマリをプロモートする前に、前のプライマリが **高速** シャットダウンを発行することを要求します。

このため、秒単位で表される `.spec.switchoverDelay` は、元のプライマリに適切にシャットダウンしてすべての WAL ファイルをアーカイブするために与えられる時間を制御します。この時間フレームでは、プライマリインスタンスは接続を受け入れません。値のデフォルトは 1 年より大きい数秒で、無限の遅延をシミュレートし、データの耐久性を維持するのに十分な大きさ。

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。これを低い値に設定すると、RPO よりも RTO が優先されますが、クラスターレベルやバックアップレベルでデータロスされる可能性があります。それどころか、高い値に設定すると、スイッチオーバー中により長い時間アクティブなプライマリなしでクラスターを残しながら、データロスのリスクを取り除くことができます。

4.8.3 フェイルオーバー

プライマリポッドに障害が発生した場合、クラスターはフェイルオーバーモードになります。詳細については、[Automated failover](#) を参照してください。

4.9 Scheduling

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

CloudNativePG クラスターのインスタンスのスケジューリング方法は、クラスターの定義の [AffinityConfiguration](#) セクションで制御できます。

- 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

4.9.1 ポッドアフィニティとアンチアフィニティ

Kubernetes では、こ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リングする（アフィニティ）またはスケジューリングしない（アンチアフィニティ）ノードを制御できます。-アフィニティ**。

CloudNativePG はデフォルトで、クラスターのインスタンスを異なるノードで設定することが好ましく、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
```

次のクラスター仕様の結果：

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  imageName: ghcr.io/cloudnative-pg/postgresql:14.2

  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、リソースが許す限り、3つの異なるノード上で3ノードの PostgreSQL クラスターをスケジューリングすることを優先します。

上記のデフォルトの動作は、上記の設定を調整することで変更できます。

`podAntiAffinityType` は `required` に設定できます。 `preferredDuringSchedulingIgnoredDuringExecution` の代わりに `requiredDuringSchedulingIgnoredDuringExecution` が使用されます。リソースが使用できない場合、このような強い要件により保留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（[Cluster Autoscaler](#) <!-- マスタ : ルール = master-> Kubernetes クラスターの自動水平スケリングの使用時に予想される条件です）。

クラウド環境での `topologyKey` の別の可能な値は `topology.kubernetes.io/zone` で、ポッドがノードだけでなくアベイラビリティゾーン全体に広がるようにします。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` に設定すると、オペレーターが生成した非アフィニティポリシーを無効にできます。

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非アフィニティルールのリストを指定できます。これらのルールは、有効になっている場合は演算子によって生成されたルールに追加され、そうでない場合は透過的に渡されます。

注釈: `additionalPodAntiAffinity` または `additionalPodAffinity` に、Pod 仕様で予期される `podAntiAffinity` または `podAffinity` のコンテンツ全体を渡す必要があります（どの PostgreSQL でも、すべてのワーカーノードで実行される PostgreSQL のインスタンスが 1 つだけの例として、次の YAML を見てください）それらが属するクラスター）。

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

4.9.2 nodeSelector によるノードの選択

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、ポッドがスケジュールおよび実行されるためのラベルとして、示された各キーと値のペアが必要です。

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

4.9.3 許容値

Kubernetes では、ノードが `taints` を明示的に許容しない (`tolerations` を介して) ポッドを撃退するかどうかを (`taints` を介して) 指定できます。

そのため、特定のノードの `taints` にマッチング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 スケジューラは、汚染ノードを考慮し、ワークロードをスケジュールするノードを決定します。セクションは、許容のための通常の Kubernetes 構文を受け入れます。

4.10 Resource management

典型的な Kubernetes クラスターでは、ポッドは無制限のリソースで実行されます。デフォルトでは、必要なだけの CPU と RAM を使用できます。

CloudNativePG を使用すると、管理者は 2 つのノブを使用して、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を制御および管理できます。

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

例、次のように、32MiB (128MiB にスケーラビリティ) の RAM と 50m の CPU (100m にスケーラビリティ) の初期 RAM を要求できます。

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

メモリリクエストと制限はコンテナに関連付けられていますが、ポッドにメモリリクエストと制限があると考えると便利です。ポッドのメモリリクエストは、ポッド内のすべてのコンテナのメモリリクエストの総和です。

ポッド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。ポッドは、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、ノードで実行するようにスケジュールされます。

リソースごとに、コンテナを優先度の高いオーダーに 3 つの Quality of Service (QoS) クラスに分割します。

- 保証
- バースタブル
- ベストエフォート

詳細については、Kubernetes 文書の [Configure Quality of Service for Pods](#) セクションを参照してください。

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

Kubernetes のリソース関連の問題を回避するには、クラスターの作成中に「リソース不足」を処理するためのベストプラクティスを参照できます。

- マニフェストファイルのリソースセクションで、メモリと CPU に必要な値を指定します。このようにして、OOM Killed（「OOM」は Out Of Memory を表します）および CPU throttle またはその他を回避できます。インスタンスの実行に関するリソース関連の問題。
- クラスターのポッドを「保証」QoS クラスに割り当てるには、制限と要求を設定する必要があります。メモリと CPU の両方を同じ値にします。
- 必要な PostgreSQL メモリパラメータをポッドリソースと一貫して指定します（あなたがするように）VM または物理的マシンのシナリオ-以下を参照）。
- nodeSelector を使用して、専用ノードでデータベースサーバポッドをセットアップします。「nodeSelector」および「tolerations」フィールドを参照してください API マニュアルページの “affinityconfiguration” リソース。

次の例マニフェストを参照できます。

```
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、つまりデータをキャッシュするために PostgreSQL サーバー専用のメモリ量を指定しています（定義されていない場合、このパラメータのデフォルト値は 128MB です）。

shared_buffers の適切な開始値は、システムのメモリの 25 % です。たとえば、shared_buffers が 256 メガバイトの場合、コンテナメモリサイズの推奨値は 1 ギガバイトです。つまり、ポッド内ではすべてのコンテナが Kubernetes が常に保存する合計 1 ギガバイトのメモリ。コンテナを期待どおりに動作させることができます。詳細については、PostgreSQL 文書の [Resource Consumption](#) セクションを参照してください。

4.11 Failure Modes

このセクションでは、PostgreSQL が存続期間中に Kubernetes クラスターで直面する可能性があるメジャー障害シナリオの概要を説明します。

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

4.11.1 ストレージスペースの使用

演算子は、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 コンテンツを格納します。

このようなストレージスペースは、次の 2 つの場合に再利用できるように設定されています。

- 対応するポッドがユーザによって削除された場合（および新しいポッドが再作成されます）
- 対応するポッドが別のノードで削除およびスケジュールされる場合

演算子が特定の PVC を再利用できないようにするには、Pod 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
```

4.11.2 失敗モード

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。

ポッドはユーザーによって明示的に削除されユーザ。 postgres コンテナのレディネスプローブが失敗します。 * postgres コンテナの活性プローブが失敗します。 * Kubernetes ワーカーノードが排出されました。 *ポッドがスケジュールされている Kubernetes ワーカーノードは失敗します。

これらの各障害は、Cluster および演算子が管理するサービスに異なる影響を及ぼします。

ユーザによって削除されたポッド

演算子に削除が通知されます。the `Cluster` に属する新しいポッドは、存在する場合は既存の PVC を再利用するか、そうでない場合は*プライマリ*の物理的バックアップから開始して自動的に作成されます。

重要: ポッドを意図的に削除する場合、`PodDisruptionBudget` ポリシーは適用されません。

- `apiserver` *に通知されるとすぐに、自己修復が行われます。

準備プロブの失敗

3 回失敗すると、ポッドは「準備ができていません」と見なされます。ポッドは引き続き `Cluster` のパートであり、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することができます。そうでない場合、障害が解決されると、ポッドは以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。

活性プロブの失敗

3 回失敗すると、`postgres` コンテナは失敗したと見なされます。ポッドは依然として `Cluster` のパートであり、*`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することができます。

プローブが 3 回失敗すると、自己修復が発生します。

ドレインされたワーカーノード

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（デフォルト：メンテナンス期間中は `on`、それ以外の場合は `off`）、**プライマリ**の物理的バックアップとは別のワーカーノードに新しいポッドが作成されます。

`PodDisruptionBudget` は、準備ができていない別のポッドが少なくとも 1 つある場合、ポッドが削除されないようにします。

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ドレインを防ぎます。Kubernetes Upgrade section を参照してください。

- `apiserver` *に通知されるとすぐに、自己修復が行われます。

ワーカーノードの障害

ノードに障害が発生したため、* kubelet *は活性プローブと準備プローブを実行しません。その特定の障害原因に対して Kubernetes クラスター管理者が設定した許容秒数後に、ポッドは削除対象としてマークされます。Kubernetes クラスターの構成方法に基づいて、ポッドは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**の物理的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。Kubernetescluster のそのパラメータのデフォルト値は 5 分です。

tolerationSeconds の後に自己修復が行われます。

4.11.3 自己修復

失敗したポッドがスタンバイの場合、ポッドは `-r` サービスおよび `-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。そうでない場合、現在のプライマリのバックアップから `newpod` が作成されます。ポッドは、準備ができたなら `-r` サービスと `-ro` サービスに再び追加されます。

障害が発生したポッドがプライマリの場合、演算子はステータスが準備完了でレプリケーションの遅延が最小のアクティブなポッドをプロモートさせ、`-rw` サービスをポイントします。失敗したポッドは `-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複製をスタートします。前のプライマリは、その PVC が使用可能な場合、`pg_rewind` を使用して自分自身を新しいものと同期します。そうでない場合、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。

4.11.4 手動介入

文書化されていない障害の場合、介入して問題を手動で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サポートと支援なしにマニュアルオペレーションを実行しないでください。

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

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

`cnpg.io/reconciliationLoop` は、特別な操作や緊急オペレーションの期間中は細心の注意を払って使用する必要があります。

警告: 限られた期間のみこのアノテーションを使用し、緊急事態が終了したら削除 make ようにしてください。この注釈をクラスターに残すと、演算子がフェイ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。

4.12 Rolling Updates

演算子は、アプリケーションがクラスターに対して実行されている間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。

- ユーザがクラスター仕様の `imageName` 属性を変更します。

PostgreSQL の設定を変更するには、リスタートが必要です 適用された;

- `Cluster .spec.resources` 値の変更
- AKS での永続ボリューム要求のサイズの変更
- 演算子が更新された後、Pod が最新のインスタンスを実行保証にしますマネージャ（`in-place updates are enabled` を除く）。

演算子は、すべてのレプリカのアップグレード処理を開始します（一度に 1 つのポッド）。最高のシリアル番号を持つレプリカから開始します。

プライマリは、アップグレードされる最後のノードです。

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

アップグレードにより、データを再複製せずに CloudNativePG ID が保持されます。必要に応じて、ポッドが削除され、同じ PVC と新しいイメージで再度作成されます。

ローリング更新プロシージャ中、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。

4.12.1 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、完全に自動化されています。レプリカがアップグレードされると、選択した `primaryUpdateMethod` オペレーションがプライマリで開始されます。これがデフォルトの動作です。

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

- **switchover** : スイッチオーバーオペレーションが自動的に実行され、新しいターゲットプライマリとして最も整列されたレプリカ、および前者をシャットダウンプライマリポッド（デフォルト）。

- **restart** : 可能であれば、ポッドの自動リスタートを実行します。プライマリインスタンスが実行されています。それ以外の場合、リスタート要求は無視され、スイッチオーバーが発行されました。

データベースの実際のワークロード、RPO および RTO に関する要件、PostgreSQL アーキテクチャが共有されているか共有されていないかなどのいくつかの要因に依存するため、更新メソッドには万能の構成はありません。

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発生させます。コンテキストで考慮すべき重要な側面の 1 つは、Kubernetes クラスターの設定と仕様に依存するため、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。**switchover** メソッドは、プロモートされたインスタンスが既にコンテナのターゲットイメージバージョンを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された後、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。データベースに対して、ローリング更新プロセスのパートとして **restart** または **switchover** を使用するのが最善かどうかを判断するのはユーザー次第です。

4.12.2 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、マニュアル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]
```

次の方法でリスタート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。

4.13 Replication

物理的レプリケーションは PostgreSQL の強みの 1 つであり、世界最大の組織のいくつかがビジネス継続高可用性コンテキストでデータの管理にそれを選択した理由の 1 つです。-唯一のワークロードとプライマリからの一部の作業のオフロード。

4.13.1 アプリケーションレベルのレプリケーション

PostgreSQL のレプリケーション機能に長年にわたって貢献してきたので、ネイティブの物理的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、Kubernetes API に直接統合することになりました。

Kubernetes の用語では、これはアプリケーションレベルのレプリケーションと呼ばれ、ストレージレベルのレプリケーションとは対照的です。

4.13.2 非常に成熟した技術

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

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開始された物理的レプリケーションは、PostgreSQL 8.2 (2006) でプライマリからウォームスタンバイの連続リカバリへの WAL シッピングを通じて初めて導入されました。

PostgreSQL 9.0 (2010) は、WAL ストリーミングとホットスタンバイ*を介した読み取り専用レプリカでそれを強化し、9.1 (2011) はトランザクションレベルでの同期レプリケーションを導入しました (RPO = 0 クラスターの場合)。カスケードレプリケーションは、PostgreSQL 9.2 (2012) でリリースされました。ロジカル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) は、データをオリジンからデスティネーションに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を導入しました。

4.13.3 PostgreSQL クラスター内でのレプリケーション

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 で提供される 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方法でネイティブかつ透過的に管理します。

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

クラスターの初期化直後に、演算子は次のように `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` の要件により、PostgreSQL 10 では `streaming_replica` ユーザーが `SUPERUSER` 特権で作成されます。

すぐに使用できるように、演算子は暗号化されたチャンネルを介してクラスター内でストリーミングレプリケーションを自動的にセットアップし、`streaming_replica` ユーザに対して TLS クライアント証明書認証を実施します- `pg_hba.conf` からの次の抜粋で強調されています：

```
# 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統合

クラスターで連続バックアップが構成されている場合、CloudNativePG はレプリカを透過的に構成して、連続リカバリに `restore_command` を利用します。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 のプルが失敗するたびに、WAL アーカイブをフォールバックオプションとして使用できます。

同期レプリケーション

演算子は、スタンバイおよびレプリカと呼ばれる 2 つの構成オプションを介した **クォーラムベースの同期ストリーミングレプリケーション** の構成をサポートします。クォーラムを決定するオーダーに使用可能なレプリカの数。

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

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

どこで：

警告:

自己修復機能を提供するために、演算子は、`minSyncReplicas` が現在利用可能なレプリカの数よりも大きい場合、`minSyncReplicas` を無視する権限を持っています。 `readyReplicas` が 0 の場合、同期レプリケーションは自動的に無効になります。

[PostgreSQL documentation](#) で述べたように、*メソッド ANY

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

重要: 演算子が同期レプリケーション設定の強制よりも自己修復を選択している場合でも、3+インスタンスのクラスター、またはより一般的には $\text{maxSyncReplicas} < (\text{instances} - 1)$ のクラスターで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。

4.13.4 外部 PostgreSQL クラスターからのレプリケーション

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** 機能によって同期が維持される場合でも、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連続リカバリレベルの両方で、レプリケーションに関して利用可能なオプションは次のとおりです。

- レプリカクラスターとソース間でストリーミングレプリケーションを使用する（これには確かにいくつかの管理およびセキュリティ関連が必要になります間のネットワーク接続を確認する `make` に行われる作業 2 つのクラスターが正しくセットアップされています）`Barman Cloud` オブジェクトストアを使用して、ベースバックアップのリカバリとソースからオブジェクトに定期的に出荷される WAL ファイルレプリカクラスターに `barman-cloud-wal-restore` によって格納およびプルされます
- 2 つのいずれか

必要なのは、実際に外部クラスターを定義することだけです。`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、[BootstrapConfiguration](#) 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：

- 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るようになります `recovery` セクションを使用する
- CloudNativePG は `restore_command` を自動的に設定します指定されたプライマリインスタンス

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

- 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るようになります `pg_basebackup` セクションを使用する
- CloudNativePG は `primary_conninfo` を自動的に設定します指定されたプライマリインスタンスのオプション、WAL レシーバソースクラスターに接続してデータを受信するプロセスが開始されます

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したオブジェクトストアでバックアップを実行でき、分散型の対称アーキテクチャを可能にします。

以下を選択することにより、PostgreSQL データベースのお気に入りの分散アーキテクチャを自由に決定できます。

- 異なるデータのマルチプルの Kubernetes クラスターにまたがるプライベートクラウドセンター
- 異なるマルチプルの Kubernetes クラスターにまたがるパブリッククラウド地域
- 前の 2 つの組み合わせ（ハイブリッド）
- 異なるマルチプルの Kubernetes クラスターにまたがるパブリッククラウド地域および異なるクラウドサービスプロバイダー

レプリカクラスターのセットアップ

ソースクラスタからレプリカクラスタをセットアップするには、クラスタ `yamfile` 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` セクションを定義する
- レプリカクラスターのブートストラップパートを定義します。経由でブートストラップできますストリーミングセクションを使用したストリーミング、またはオブジェクトストアからのブートストラップ `recovery` セクションを使用する
- レプリカクラスターで連続リカバリパート（`spec.replica`）を定義します。全部 必要なのは、オプション `spec.replica.enabled` を使用してレプリカモードを有効にすることです。オプション `spec.replica.source` に `externalClusters` 名前を設定します

この **最初の例**では、ブートストラップと連続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/` サブディレクトリで sample YAML を確認できます。

ソースクラスタを指す `bootstrap` および `replica` セクションに注意してください。

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

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

`externalClusters` セクションで、`connectionParameters` サブセクションのホストに正しい名前空間を使用することを忘れないでください。レプリカクラスタが別の名前空間にある場合は、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:
```

(次のページに続く)

(前のページからの続き)

```

name: <MAIN-CLUSTER>-ca
key: ca.crt

```

2 番目の例は、`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するレプリカクラスターと、`streamingreplication` と指定されたオブジェクトストアの両方を使用した連続リカバリを定義します。ストリーミングレプリケーションの場合、`replicacluster` 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す `bootstrap` および `replica` セクションに注意してください。

```

bootstrap:
  recovery:
    source: cluster-example

replica:
  enabled: true
  source: cluster-example

```

`externalClusters` セクションで、`endpointURL` および `connectionParameters.host` で正しい名前空間を使用するように注意します。必要に保証が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ます。

```

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 `make` 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。

レプリカクラスターで指定されたプライマリを昇格させる

指定されたプライマリ を **プライマリ** にプロモートさせるには、`option spec.replica.enabled` を使用してレプリカクラスターのレプリカモードを無効にするだけです。

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

レプリカモードが無効になると、レプリカクラスターとソースクラスターは2つの別個のクラスターになり、`replicaclasses` の **指定プライマリ** はそのクラスターの **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、`rolechange` 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**オペレーションです。レプリケーションが無効になり、**指定されたプライマリ**が**プライマリ**に昇格されると、レプリカクラスターとソースクラスターは最終的に2つの独立したクラスターになります。

4.14 Backup and Recovery

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

このためには、`barman-cli-cloud` がインストールされたイメージが必要です。このスコープには、コミュニティ PostgreSQL イメージと latest `barman-cli-cloud` パッケージで構成されるため、イメージ `ghcr.io/cloudnative-pg/postgresql` を使用できます。

重要: システムで最新バージョンのオペランドを実行していることを常に保証し、Barman クラウドで導入された改善を活用する（およびクラスターのセキュリティ面を改善する）。

バックアップは `a Cluster` のプライマリインスタンス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）。

4.14.1 クラウドプロバイダーのサポート

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

- [AWS S3](#)
- [Microsoft Azure Blob Storage](#)
- [Google Cloud Storage](#)

サポートされているサービスの互換性のある実装を使用することもできます。

必要なセットアップは、選択したストレージプロバイダーによって異なります。以下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になります。

- `ACCESS_KEY_ID`：使用されるアクセスキーの ID S3 でファイルをアップロードするには
- `ACCESS_SECRET_KEY`：前のアクセスキーの秘密のパート
- `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パーミッションが必要です。そのため、資格情報を使用して k8s シークレットを作成する必要があり、次のコマンドを使用して作成できます。

```
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、インストール時に暗号化が保存されている場合は暗号化されます。

その秘密を考えると、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:
```

(次のページに続く)

```
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すべての URL (eg `s3://BUCKET_NAME/path/to/folder` など) にすることができます。

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、region `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: ConfigMaps と Secrets をインスタンスによって**自動的にリロードする場合、キー `cnpg.io/reload` の label 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。

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リレーする共通インタフェースとして MinIO Gateway を使用できます。詳細については、[MinIO official documentation](#) を参照してください。

具体的には、CloudNativePG クラスタは、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 localMinIO ゲートウェイを直接指すことができます。

MinIO シークレットは PostgreSQL クラスターと MinIO インスタンスの両方で使用されるため、同じネームスペースで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。

オーダー PostgreSQL が MinIO Gateway に到達できるようにするには、MinIO Gateway インスタンスにバインドされたポート 9000 に *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: この文書の執筆時点では、Kubernetes の公式 [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認証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。

AWS S3 をクラウドオブジェクトストレージとして使用する例を次に示します。

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

(次のページに続く)

```

- s3
env:
# MinIO access key and secret key
- name: MINIO_ACCESS_KEY
  valueFrom:
    secretKeyRef:
      name: minio-creds
      key: MINIO_ACCESS_KEY
- name: MINIO_SECRET_KEY
  valueFrom:
    secretKeyRef:
      name: minio-creds
      key: MINIO_SECRET_KEY
# AWS credentials
- name: AWS_ACCESS_KEY_ID
  valueFrom:
    secretKeyRef:
      name: aws-creds
      key: ACCESS_KEY_ID
- name: AWS_SECRET_ACCESS_KEY
  valueFrom:
    secretKeyRef:
      name: aws-creds
      key: ACCESS_SECRET_KEY
# Uncomment the below section if session token is required
# - name: AWS_SESSION_TOKEN
#   valueFrom:
#     secretKeyRef:
#       name: aws-creds
#       key: ACCESS_SESSION_TOKEN
ports:
- containerPort: 9000

```

Cluster definition で endpointURL として MinIO Gateway サービスを構成し、次にバケット名を選択して BUCKET_NAME を置き換えます。

```

apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      destinationPath: s3://BUCKET_NAME/
      endpointURL: http://minio-gateway-service:9000
      s3Credentials:

```

(前のページからの続き)

```

accessKeyId:
  name: minio-creds
  key: MINIO_ACCESS_KEY
secretAccessKey:
  name: minio-creds
  key: MINIO_SECRET_KEY
[...]
```

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

Azure Blob Storage

ストレージアカウントにアクセスオーダーには、次の認証情報の組み合わせのいずれかが必要です。

- **Connection String**
- ストレージアカウント名前 および **Storage account access key**
- ストレージアカウント名前 および **Storage account SAS Token**

資格情報は Kubernetes Secret 内に保存する必要があり、必要な場合にのみデータエントリを追加します。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>
```

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

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

```

apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      destinationPath: "<destination path here>"
      azureCredentials:
        connectionString:
          name: azure-creds
          key: AZURE_CONNECTION_STRING
        storageAccount:
          name: azure-creds
```

(次のページに続く)

```

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

```

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

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

ここで、`<resource-path>` は `<container>/<blob>` です。**ストレージ名前** は、使用されるホスト名前に含まれアカウント。

その他の Azure Blob Storage 互換プロバイダー

Azure Blob Storage API の異なる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 StorageEmulator 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、演算子は 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 クラスター内で実行されていると想定し、もう 1 つは環境変数 `GOOGLE_APPLICATION_CREDENTIALS` を利用します。

Google Kubernetes Engine 内で実行する

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

```

これは、クラスターが Google KubernetesEngine 内で実行されていること、つまりファイルをアップロードするために資格情報が必要ないことを演算子に伝えます。

重要: このメソッドでは、クラスター管理者が定義する必要があるクラスターとポッドのアクセス許可を慎重に定義する必要があります。

認証を使用する

[instruction from Google](#) に続いて、認証に必要なすべての情報を含む JSON ファイルを取得します。

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

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

これにより、backup-creds という名前の Secret が作成され、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、演算子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: この認証方法は、Google Cloud Storage バケットにアクセスするために必要なすべての情報を含む JSON ファイルをコンテナ内に作成します。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与えられます。

4.14.2 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
```

演算子は、`barman-cloud-backup` を使用して必要なバックアップを取るためにクラスターの調整をスタートします。プレーンな `kubectl describe backup <name>` コマンドを使用して、バックアップステータスを確認できます。

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

バックアップが完了すると、次の例のフェーズは `completed like` 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>

```

!!!重要この機能は、スーパーユーザとアプリケーションユーザの秘密をバックアップしません。シークレットは、Kubernetes クラスターの標準バックアップ手順のパートとしてバックアップされることになっています。

4.14.3 スケジュールされたバックアップ

ScheduledBackup 名前付け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 a Backup と似ていますが、`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、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： バックアップ頻度は、フルまたはポイントインタイムリカバリオペレーションを必要とする災害

後のリカバリ時間オブジェクト（RTO）にインパクト可能性があります。私たちのアドバイスは、定期的にバックアップをリカバリしてテストし、最初からリカバリするのにかかる時間を測定して、RTO の予測可能性を改善できるようにすることです。リカバリ時間は、ベースバックアップのサイズと、アーカイブからフェッチしてリカバリ中に再生する必要がある WAL ファイルの量に影響されます（WAL アーカイビングが PostgreSQL での連続バックアップを可能にすることを思い出してください！）。経験に基づいて、ほとんどの場合、週単位のベースバックアップで十分です-1 日に 1 回よりも頻繁にバックアップをスケジュールすることは非常にまれです。

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

`ScheduledBackup` リソースの作成後すぐにバックアップを発行する場合は、`.spec.immediate: true` を設定できます。

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

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

4.14.4 WAL アーカイビング

宛先パスを選択し、クラウド認証情報を設定するとすぐに、WAL アーカイビングが有効になります。

必要に応じて、WAL ファイルをアップロードしたらすぐに圧縮したり、暗号化したりできます：

```
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設定できます。クラスタ設定で暗号化を上書きしない限り、オペレータは暗号化を使用します。

PostgreSQL はシーケンシャルアーカイビングスキームを実装します。このスキームでは、アーカイブするすべての WAL セグメントに対して `the archive_command` がシーケンシャルに実行されます。

重要: デフォルトでは、CloudNativePG は `archive_timeout` を `5min` に設定します。これにより、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、アーカイブされます。 `archive_timeout` の値を変更しても、経験により、演算子によって設定されたデフォルト値はほとんどのユースケースに適していることが示唆されています。

PostgreSQL インスタンスとオブジェクトストア間の帯域幅が複数の WAL ファイルの並列アーカイピングを許可する場合、次の例のようにインスタスマネージャーの並列 WAL アーカイピング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、インスタスマネージャは、PostgreSQL が要求するものを含め、最大 8 つの readyWAL を並行してアーカイピングすることにより、WALarchiving プロセスを最適化します。

PostgreSQL が、最適化としてインスタスマネージャによって既にアーカイブされている WAL のアーカイピングを要求する場合、そのアーカイブ要求は肯定的なステータスで破棄されます。

4.14.5 回復

クラスターの復元は、既存のクラスターでは「インプレース」では実行されません。オブジェクトストレージにアップロードされたデータを使用して、以前に取得したバックアップからブートストラップ新しいクラスターを作成できます。演算子は、リカバリプロセスツール（ベースバックアップ）および the `barman-cloud-wal-restore` ツール（要求に応じて、パラレルサポートを含む WAL ファイル用）。

`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。

フードの下で、演算子は新しいクラスターの最初のインスタンスに `init` コンテナを注入し、`init` コンテナはオブジェクトストレージからバックアップのリカバリをスタートします。

重要: 新しい PVC のベースバックアップコピーの期間は、バックアップのサイズ、およびネットワークとストレージの両方の速度に依存します。

ベースバックアップリカバリプロセスが完了すると、演算子は `Postgres` インスタンスをリカバリモードで開始します。このフェーズでは、接続を受け入れることができず、ポッドは健全であるにもかかわらず、PostgreSQL はプローブしています。 `restore_command` を介して、PostgreSQL はアーカイブから WAL ファ

イルの取得を開始します (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)。

PostgreSQL がターゲット (Point-In-Time-Recovery の場合は WAL の終わりまたは必要なターゲット) に到達すると、このフェーズは終了します。実際、オプションで `recoveryTarget` を指定して、Point-in-Time 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 (PostgreSQL の場合は `current`、バージョン 12 以上の場合は `latest`) まで継続します。

リカバリが完了すると、演算子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は通常どおり起動し、残りのインスタンスはレプリカとしてクラスターに結合します。

このプロセスはユーザに対して透過的であり、ポッドで実行されている `instancemanager` によって管理されます。

4.14.6 保持ポリシー

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

内部的に、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。

例、次のように、30 日間の保持ポリシーを使用してバックアップを定義できます。

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

4.14.7 圧縮アルゴリズム

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

- `bzip2`
- `gzip` *スナッピー

バックアップと WAL の圧縮設定は独立しています。API リファレンスの [DataBackupConfiguration](#) および [WalBackupConfiguration](#) セクションを参照してください。

アーカイブ時間、復元時間、およびアルゴリズム間のサイズの変更にノートすることが重要です。そのため、ユースケースに応じて圧縮アルゴリズムを選択する必要があります。

Barman チームは、Barman Cloud の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、ローカルの MinIO 展開でバックアップが作成されるシナリオをまとめたものです。Barman GitHub プロジェクトには [deeper analysis](#) が含まれています。

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

4.14.8 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バックアップのサポートが導入されています。その結果、PostgreSQL コンテナイメージにバージョン 2.18 以降の Barman が含まれている場合、CloudNativePG では、バックアップオブジェクト、つまりベースバックアップ、WAL ファイル、履歴ファイルのキーと値のペアとしてタグを指定できます。

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

- `tags` : バックアップオブジェクトとアーカイブされた WAL に追加されるキーと値のペアのタグバックアップオブジェクトストア内のファイル
- `historyTags` : 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バックアップオブジェクトストア

以下の YAML マニフェストの抜粋は、この機能の使用例を示しています。

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

(次のページに続く)

```
barmanObjectStore:
  [...]
  tags:
    backupRetentionPolicy: "expire"
  historyTags:
    backupRetentionPolicy: "keep"
```

4.15 PostgreSQL Configuration

PostgreSQL に精通しているユーザーは、インスタンスを設定するための次の 2 つのファイルの存在を認識しています。

- `postgresql.conf` : PostgreSQL のメインの実行時設定ファイル
- `pg_hba.conf` : クライアント認証ファイル

宣言的な設定と PostgreSQL コンテナの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` セクションで構成が可能です。

これらの設定は、すべてのインスタンスで同じです。

警告: `ALTER SYSTEM` クエリーを使用して、PostgreSQL インスタンスの設定を強制的に変更しないでください。演算子によって通常制御されるオプションの一部を変更すると、実際にクラスターの予測不能/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。

サンプルには、カスタム設定の使用リファレンスが含まれています。cluster-example-custom.yaml を参照してください。

4.15.1 postgresql セクション

ポッドの PostgreSQL インスタンスは、これらの設定が自動的に追加されるデフォルトの `postgresql.conf` ファイルで始まります：

```
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オーダーで適用することにより、オペレーターによって自動的に生成および維持されます。

- グローバルデフォルトパラメータ PostgreSQL メジャーバージョンに依存するデフォルトのパラメーター
- 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 は次のとおりです。

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

警告: PostgreSQL クラスターでの WAL セグメントの保持を計画し、予想および観測されたワークロードに基づいて、サーババージョンに応じて `wal_keep_size` または `wal_keep_segments` を適切に構成することがあなたの義務です。 CloudNativePG がレプリケーションスロットをサポートするまで、および継続的なバックアップを用意していない場合、これがスタンバイが同期から外れて *"could not receive data from WAL stream: ERROR: requested WAL segment ***** has already been removed"* のようなエラーメッセージを返す場合を防ぐ唯一の方法です。これには、ストリーミングレプリケーションのために古い WAL セグメントを保持するために、`PGDATA` のパートを専用にする必要があります。

以下のパラメーターは **固定**であり、演算子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 = /var/run/postgresql
wal_level = logical
wal_log_hints = on
```

固定パラメータは最後に追加されるため、YAML 設定を介してユーザーが上書きすることはできません。これらのパラメータは、正しい WALarchiving 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` パラメータは、クラスター内のインスタンスの状態に応じて演算子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

演算子は PostgreSQL にログを CSV フォーマットで出力するよう要求し、インスタンスマネージャは自動的に解析して JSON フォーマットで出力します。このため、PostgreSQL のすべてのログ設定は固定されており、変更できません。

詳細については、[ロギング](#) を参照してください。

共有プリロードライブラリ

PostgreSQL の `shared_preload_libraries` オプションは、comma-separated リストのフォームで、サーバースタートにプリロードされる 1 つ以上の共有ライブラリを指定するために存在します。通常、PostgreSQL では、システム全体のほとんどのデータベースセッション（`pg_stat_statements` など）で使用する必要がある拡張機能をロードするために使用されます。

CloudNativePG では、デフォルトで `shared_preload_libraries` オプションは空です。`shared_preload_libraries` のコンテンツをオーバーライドできますが、このオプションを利用できるのはエキスパートの Postgres ユーザーのみに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、サーバーはスタートに失敗し、CloudNativePG が自己修復を試みなくなり、マニュアル介入が必要になります。そのコンテンツを直接管理する場合は、`shared_preload_libraries` の拡張子と設定の両方を常にテスト `make` してください。

CloudNativePG は、最もよく使用されるいくつかの PostgreSQL 拡張機能の `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 [マネージ拡張](#) セクションを参照してください）。

具体的には、構成パラメーターが管理対象ライブラリの 1 つを必要とすることに演算子が気づくとすぐに、必要なライブラリが自動的に追加されます。演算子は、実際のパラメーターが必要としないライブラリをすぐに削除します。

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リスタートして有効にオーダー必要があることに常に留意してください。

`.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。オペレータは、それらを自動的に管理する文字列とマージします。

マネージ拡張

前のセクションで予想したように、CloudNativePG は、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に追加のオブジェクトを必要とするものもあります。通常は、`CREATE EXTENSION` コマンドで管理されるビューや関数をデータベースで実行します（`DROP EXTENSION` コマンドは通常これらのオブジェクトを削除します）。

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

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

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

auto_explain 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`

を手動で実行することなく、遅いステートメントの実行計画を自動的にロギングする手段を提供します（最適化されていないクエリの追跡に役立ちます）。

次の抜粋の例のように、`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（完了まで 10 秒以上かかるクエリの実行プランを自動的に記録します）。

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

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

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

次の例の抜粋のように、`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
# ...
```

前に説明したように、演算子は自動的に `pg_stat_statements` を `shared_preload_libraries` に追加し、各データベースで `CREATE EXTENSION IFNOT EXISTS pg_stat_statements` を実行し、`pg_stat_statements` ビューに対してクエリを実行できるようにします。

pgaudit を有効にする

pgaudit 拡張機能は、標準の PostgreSQL ログイン機能を介して、詳細なセッションおよび/またはオブジェクト監査ログを提供します。

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

次の例の抜粋のように、`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"
#
```

4.15.2 pg_hba セクション

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

最初のマッチングルールが認証に使用されるため、演算子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なすことができます。

1. ルール 2 を修正しました。ユーザー定義のルール 3。オプションの LDAP セクション 4。デフォルトのルール

固定ルール：

```
local all all peer

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

デフォルトのルール：

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

PostgreSQL 14 以降、`password_encryption database` パラメータのデフォルト値は `scram-sha-256` に設定されています。そのため、デフォルトの認証メソッドはこの PostgreSQL バージョンの `scram-sha-256` です。

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

結果の `pg_hba.conf` は次のようになります。

```
local all all peer
```

```
hostssl postgres streaming_replica all cert
```

```
hostssl replication streaming_replica all cert
```

```
<user defined rules>
```

```
<user defined LDAP>
```

```
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 の文書を参照してください。

LDAP 設定

クラスター仕様の `postgres` セクションの下には、`pg_hba.conf` ファイルに追加されたルールに変換される LDAP 構成を定義するためのオプションの `ldap` セクションがあります。

これは、LDAP セクションで `server`、`prefix`、`suffix` を指定する必要がある `simple bind` モードと、`server`、`baseDN`、`bindDN`、および `ldap` を含むパスワードである `bindPassword` を指定する `search+bind` モードをサポートします。さらに、`search+bind` モードでは、`a searchFilter` または `searchAttribute` を指定するオプションがあります。`searchAttribute` が指定されていない場合、デフォルトの `uid` が使用されます。

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

検索用に記入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
```

4.15.3 設定の変更

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。変更にリスタートが必要なパラメータが含まれる場合、演算子はローリングアップグレードを実行します。

4.15.4 動的共有メモリ設定

PostgreSQL は、`dynamic_shared_memory_type` 構成オプションを使用した動的共有メモリ管理の実装をいくつか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アロケーションにとって特に重要です。詳細については、この *thread from the `pgsql-general` mailing list* <https://www.postgresql.org/message-id/CA%2BhUKGJOj7qzDLxeFPVvto8YEWop6FSQoTYPO9Z6Ee%3Di-nPS_Q%40mail.gmail.com>`\_\_` を参照してください。

POSIX 共有メモリ

ほとんどの場合、`posix` のデフォルト設定で十分です。これは、演算子が `shm` under `/dev/shm` と呼ばれる `*memory-bound EmptyDir` ボリューム*を自動的にマウントすることを考慮しています。`runningPostgres` コンテナ内のそのようなボリュームのサイズは、次の方法で確認できます。

```
mount | grep shm
```

次のような出力が得られるはずです。

```
shm on /dev/shm type tmpfs (rw,nosuid,nodev,noexec,relatime,size= **** **)
```

System V 共有メモリ

Kubernetes クラスターの `SHMMAX` および `SHMALL` パラメーターに十分な値がある場合は、以下を設定することもできます。

```
dynamic_shared_memory_type: "sysv"
```

以下を実行することにより、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、`max total shared memory` の数が非常に多いため、`dynamic_shared_memory_type` を `sysv` に設定することをお勧めします。

別のメソッドは次を実行することです：

```
cat /proc/sys/kernel/shmall  
cat /proc/sys/kernel/shmmax
```

4.15.5 固定パラメーター

一部の PostgreSQL 構成パラメーターは、オペレーターのみが管理する必要があります。演算子は、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`
- `huge_pages`
- `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`

4.16 Operator configuration

CloudNativePG の演算子は、標準の展開マニフェストからインストールされ、構成パラダイムよりも慣習に従います。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したいシナリオがいくつかあります。

- 作成されたすべてのリソースに継承されるアノテーションとラベルの定義演算子によって、クラスタリソースに設定されている PostgreSQL 用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、演算子は `cnpg-controller-manager` という Kubernetes Deployment として `cnpg-system` namespace にインストールされます。

注釈: 以下の例では、演算子展開のデフォルトの名前と名前空間を想定しています。

演算子のデプロイメントと同じ名前空間にあり、名前として `with cnpg-controller-manager-config` にある `ConfigMap / Secret` 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、演算子によって自動的に検出されないため、再読み込みするニーズがあります（以下を参照）。さらに、変更は構成が再ロードされた後に作成されたリソースにのみ適用されます。

重要: 演算子は最初に `ConfigMap` 値を処理し、次にシークレットの値をこのオーダーで処理します。その結果、両方の場所でパラメータが定義されている場合、シークレットのパラメーターが使用されます。

4.16.1 利用可能オプション

演算子は、`ConfigMap / Secret` で定義される以下の環境変数を探します。

Name	Description
<code>INHERITED_ANNOTATIONS</code>	<code>`Cluster`</code> メタデータで定義されている場合、ポッドを含むすべての生成されたリソースに継承される注釈名のリスト
<code>INHERITED_LABELS</code>	<code>`Cluster`</code> メタデータで定義されている場合、ポッドを含むすべての生成されたリソースに継承される <code>label</code> 名のリスト
<code>PULL_SECRET_NAME</code>	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
<code>ENABLE_AZURE_PVC_UPDATE</code>	PVC がサイズ変更状態でスタックしている場合、Postgres ポッドを削除できない。この機能は主に Azure 環境用です（デフォルトは <code>`false`</code> ）
<code>ENABLE_INSTANCE_MANAGER_UPDATE</code>	<code>true</code> に設定すると、演算子の更新後にインスタスマネージャのインプレースメント更新を有効にし、Kubernetes のローリング更新を回避します（デフォルトの <code>false`</code> ）
<code>MONITORING_QUERIES_CONFIGMAP</code>	作成されたすべてのクラスターに適用される一連のデフォルトクエリ（キー <code>`queries`</code> で指定される）を持つオペレーターの名前スペース内の <code>ConfigMap</code> の名前
<code>MONITORING_QUERIES_SECRET</code>	作成されたすべてのクラスターに適用される、一連のデフォルトクエリ（キー <code>`queries`</code> で指定される）を持つオペレーターの名前スペース内のシークレットの名前

`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。例、値 `example.com/*` は値 `example.com/one` と `example.com/two` の両方に一致します。

`PULL_SECRET_NAME` パラメータを使用して追加のプルシークレット名前を指定すると、演算子はそのシークレットを使用して、作成された PostgreSQLcluster ごとにプルシークレットを作成します。その秘密の名前付けは `<cluster-name>-pull` になります。

演算子が `PULL_SECRET_NAME` シークレットを探す名前空間は、演算子をインストールした場所です。演算子はその秘密を見つけることができない場合、設定パラメータを無視します。

警告: 以前のバージョンの演算子は、PostgreSQL クラスターをデプロイするネームスペース内に *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、以前の説明とマッチするように動作が変更されました。演算子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ていません。

4.16.2 演算子構成マップの定義

以下の例は、label/注釈名を定義して、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継承されるようにし、[in-place updates for the instancemanager](#) を有効にすることによって、演算子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
```

4.16.3 演算子シークレットの定義

以下の例は、label/注釈名を定義して、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継承されるようにし、[in-place updates for the instancemanager](#) を有効にすることによって、演算子の動作をカスタマイズします。

```
apiVersion: v1
kind: Secret
metadata:
  name: cnpg-controller-manager-config
  namespace: cnpg-system
type: Opaque
stringData:
  INHERITED_ANNOTATIONS: categories
  INHERITED_LABELS: environment, workload, app
  ENABLE_INSTANCE_MANAGER_INPLACE_UPDATES: true
```

4.16.4 構成をリロードするために演算子を再起動する

変更を有効にするには、演算子ポッドを再作成して構成マップを再ロードする必要があります。マニフェストを使用して Kubernetes に演算子をインストールした場合は、次を発行することで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、特定の名前空間を指定すると、次のコマンドを使用して演算子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、演算子デプロイメントのリロード後に作成された *Cluster* リソースにのみ適用されます。

上記の例に従って、Cluster 定義に `categories` 注釈と `environment`、`workload`、または `app` ラベルのいずれかが含まれている場合、これらはデプロイメントによって生成されるすべてのリソースに継承されます。

4.16.5 PPROF HTTP サーバー

演算子は、ローカルホスト:6060 上の以下のエンドポイントで PPROF HTTP サーバーを公開できます。

```
- /debug/pprof/. Responds to a request for "/debug/pprof/" with an HTML page,
  ↳listing the available profiles
- /debug/pprof/cmdline. Responds with the running programs command line, with
  ↳arguments separated by NUL bytes.
- /debug/pprof/profile. Responds with the pprof-formatted cpu profile. Profiling
  ↳lasts for duration specified in seconds GET parameter, or for 30 seconds if not
  ↳specified.
- /debug/pprof/symbol. Looks up the program counters listed in the request,
  ↳responding with a table mapping program counters to function names.
- /debug/pprof/trace. Responds with the execution trace in binary form. Tracing
  ↳lasts for duration specified in seconds GET parameter, or for 1 second if not
  ↳specified.
```

演算子を有効にするには、演算子のデプロイメントを編集して、`--pprof-server=true` フラグを追加する必要があります。

これを行うには、次のコマンドを実行します: `kubectl edit deployment -n cnpg-system cnpg-controller-manager`

次に、編集ページでコンテナ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、展開でロールアウトが実行され、新しいポッドで PPROF サーバーが有効になります。

ポッドが実行されたら、次のようにしてコンテナ内で実行できます：`kubectl exec -ti -n cnpg-system <pod name> -- bash` 一度実行に入ると：

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

4.17 Storage

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。ストレージは常に利用可能で、スケーリングし、適切に機能し、一貫性と耐久性を保証する必要があります。仮想マシンやベアメタルなどの従来の環境に適用されるのと同じ期待と要件は、Kubernetes によって管理されるコンテナコンテキストでも有効です。

重要： Kubernetes には、動的にプロビジョニングされたストレージに関して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 および物理的サーバー上のデータベースワークロードのストレージに関して長年にわたって構築してきた貴重な知識に加えて、これらの概念を所有する必要があります。

ストレージへのアクセスには、プライマリに2つの方法があります。

- **ネットワーク：**直接または間接（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：**Pod が実行されているノードに直接接続されます（これには、Kubernetes のベアメタルインストールに直接接続されたディスクも含まれます）

Kubernetes で最も一般的な使用パターンであるネットワークストレージは、従来の環境で経験できるのと同じスループットとレイテンシの問題を示しています。これらは、いくつかのアプリケーションとの I/O 競合によりパフォーマンス結果のばらつきが増加する共有環境で強調できます。

ローカルストレージは、シェアードナッシングアーキテクチャを可能にします。これは、より高い予測可能なパフォーマンスを保証するため、高トランザクションおよび超大規模データベース（VLDB）ワークロードにより適しています。

警告： CloudNativePG で PostgreSQL クラスターをデプロイする前に、使用しているストレージがデータベースワークロードに推奨されていることを保証してください。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。

4.17.1 CloudNativePG のベンチマーク

EDB は、`cnp-bench` を管理します。`cnp-bench` は、運用環境でデータベースを展開する前に、管理された 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 チャート稼動。

簡単に言うと、`cnp-bench` は 2 つのレベルで動作するように設計されています。

- 関連する `fio` を使用して、基礎となるストレージのパフォーマンスを測定するシーケンシャル読み取り、シーケンシャルのスループットなどのデータベースワークロードのメトリック書き込み、ランダム読み取り、ランダム書き込み
- 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 PostgreSQL と一緒に配布：`pgbench`

重要: ストレージとデータベースのパフォーマンスの両方を測定することは、データベースが稼動する**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（キャパシティプランニングなど）だけでなく、特に緊急事態（この種のテストを実行する余裕がない場合）の稼動ライフサイクルでも非常に貴重です。データベースは実際に時間とともに変化し進化するため、データのディストリビューションも変化し、パフォーマンスに影響を与える可能性があります。シーケンシャル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、こうした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り結果が変わらないシェアードナッシングコンテキストでは、**システムを把握し、ベンチマークを行います。

4.17.2 永続的なボリューム要求

演算子は、PGDATA を保存することを目的として、PostgreSQL インスタンスごとに永続ボリューム要求 (PVC) を作成し、それを各ポッドにマウントします。

4.17.3 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設定する簡単な方法は、次の例のように、特定のサイズのストレージを要求することです。

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

前の構成を使用すると、生成された PVC はデフォルトの storageclass で満たされます。ターゲット Kubernetes クラスターにデフォルトのストレージクラスがない場合、または PVC を既知のストレージクラスで満たす必要がある場合でも、カスタムリソースに設定できます。

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

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、稼動環境に到達する前に、制御された環境でストレージクラスを適切にベンチマークすることをお勧めします。

4.17.4 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できます。

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

  storage:
    pvcTemplate:
      accessModes:
        - ReadWriteOnce
      resources:
        requests:
          storage: 1Gi
      storageClassName: standard
      volumeMode: Filesystem
```

4.17.5 ボリューム拡張

Kubernetes は、[expanding PVCs](#) を許可する API を公開します。これはデフォルトで有効になっていますが、基礎となる StorageClass でサポートするニーズがあります。

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 機能を使用する

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。演算子はすべての PVC に変更を適用します。

StorageClass が [online volume resizing](#) をサポートしている場合、変更はすぐにポッドに適用されます。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、サイズ変更をトリガーするにはポッドを削除する必要があります。

続行する最善の方法は、レプリカから開始し、各 Pod がバックアップされるのを待機して、一度に 1 つの Pod 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、*Operator configuration* の `ENABLE_AZURE_PVC_UPDATES` 環境変数によっ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 を手動でサイズ変更できます。例、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「接続されていない」状態のときにのみ展開できます。

<！ _ _ _ _ _。これにより、オペレータはポッドを再作成し、バックグラウンドディスクのサイズ変更が完了する前にすぐに PVC に再接続できなくなります。

最初のステップは、新しいサイズを適用するクラスター定義を編集することです。たとえば、「2Gi」と次のようになります。

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

  storage:
    storageClass: default
    size: 2Gi
```

cluster-example-1 Pod がクラスターのプライマリであると仮定すると、最初にレプリカに進むことができます。例、cluster-example-3 Pod をホストする kubernetes ノードのコード化からスタートします。

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

次に、cluster-example-3 ポッドを削除します。

```
$ 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、ノードのコードを解除できます。

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

ポッドが正しく再作成され、実行中および準備完了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
```

ここで、次のコマンドを実行して PVC の拡張を確認します。これにより、構成された「2Gi」が結果れます。

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

そのため、残りのポッドに対してこれらの手順を繰り返すことができます。

重要: `kubectl cnpg promote`（例: `kubectl cnpg promote cluster-example 3` をプライマリにプロモートさせる `kubectl cnpg promote cluster-example 3`）を使用して、スイッチオーバーで新しいサイズ変更された Pod を昇格した後、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ます。

ストレージの再作成

ストレージクラスがボリュームの拡張をサポートしていない場合でも、ストレージを増やした新しい PVC を割り当てて、そこにデータベースを移動することで、異なる PVC でクラスターを再生成できます。このオペレーションは、クラスターに複数のノードが含まれる場合にのみ実行可能です。

それを行う間、次の例のように、`resizeInUseVolumes` フラグを無効にして、演算子が既存の 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格納領域に移動オーダーには、すべての PVC とすべての Pod を再作成する必要があります。次の例のような 3 つのレプリカを持つクラスターがあるとします。

```
$ kubectl get pods
NAME                READY   STATUS    RESTARTS   AGE
cluster-example-1   1/1     Running   0           2m37s
cluster-example-2   1/1     Running   0           2m22s
cluster-example-3   1/1     Running   0           2m10s
```

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

例として、`cluster-example-3` のストレージを再作成するには、次のことができます。

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

それを行った後、演算子は、サイズ変更された PVC を使用して別のレプリカの作成を調整します。

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
cluster-example-1	1/1	Running	0	5m58s
cluster-example-2	1/1	Running	0	5m43s
cluster-example-4-join-v2bfg	0/1	Completed	0	17s
cluster-example-4	1/1	Running	0	10s

4.18 Labels and annotations

Kubernetes のリソースはフラットな構造で編成されており、階層情報やリソース間のリレーションはありません。ただし、このようなリソースとオブジェクトをリンクして、**ラベル** および **注釈** を通じてリレーションを構築できます。

要するに：

- 注釈は、追加の非識別情報を割り当てるために使用されます外部ツールとの統合を促進することを目的としたリソース
- オブジェクトをグループ化し、Kubernetes のネイティブを介してクエリーするために label が使用されますセクター機能

CloudNativePG デプロイメントで使用する 1 つ以上のラベルや注釈を選択できます。次に、クラスターのメタデータでこれらのラベルや注釈を定義するときに、作成されたすべてのリソース（ポッドを含む）によって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: ラベルと注釈の継承は、ポッドテンプレートなどの代替アプローチの代わりに CloudNativePG で採用されている手法です。

4.18.1 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 label または注釈は、関連するリソースによって継承されません。label/注釈の継承を有効にオーダーには、[Operator configuration](#) セクションで提供される指示に従う必要があります。

以下では、その例を続けて、以下に限定します。

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

注釈: 注釈とラベルの両方のコンテキストに最も適した名前をフリーに選択してください。また、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に `mycompany/*` などの戦略を採用して継承できることを忘れないでください。

4.18.2 クラスターのメタデータの定義

クラスターを定義するとき、リソースがデプロイされる前に、次のようにメタデータを適切に設定できます。

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

クラスターがデプロイされると、例、次のようにしてラベルがポッドに正しく設定されていることを確認できます。

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

4.18.3 現在の制限

現在、CloudNativePG はラベルまたは注釈の削除を自動的に伝播しません。したがって、アノテーションまたは label がクラスターから削除された場合、以前にポッドに伝達されていた場合、オペレーターは関連リソースからそれを自動的に削除しません。

4.19 Monitoring

4.19.1 インスタンスの監視

PostgreSQL インスタンスごとに、演算子は、演算子名前付けのポート 9187 で、HTTP をシステムで [プロメテウス演算子の例](#) のメトリックのエクスポートを提供しリソース。(詳細については、以下の [ユーザー定義のメトリック](#) を参照してください)。

重要: バージョン 1.11 以降、CloudNativePG は *default-monitoring* と呼ばれる *ConfigMap* に by default a set of predefined metrics をすでに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 で実行されるモニタリングクエリはすべて次のとおりです。

- トランザクションアトミック（クエリーごとに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 `postgres` として実行

pg_monitor ロールの詳細については、PostgreSQL [documentation](#) の「デフォルトロール」セクションを参照してください。

デフォルトでは、次のロジックに従って、Cluster リソースの指定された `bootstrap` メソッドによって定義されているように、クエリは `* main データベース*` に対して実行されます。

デフォルトデータベースは、`target_databases` オプションで1つ以上のデータベースのリストを指定することにより、特定のユーザー定義メトリックに対して常にオーバーライドできます。

プロメテウス演算子の例

特定の PostgreSQL クラスターは、Prometheus Operator's リソース `PodMonitor` リソースを使用してモニターできます。クラスターを正しく指す `PodMonitor` は、クラスターリソース自分自身で `.spec.monitoring.enablePodMonitor` を `true` に設定することにより、演算子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上書きされます。カスタマイズする必要がある場合は、以下で説明するように行うことができます。

特定のクラスターの `PodMonitor` を手動でデプロイするには、次のように定義し、必要に応じて変更します：`edb_notranlate_1`

重要: 必ず上記の例を、一意の名前と正しいクラスターの名前空間とラベルで変更してください（*cluster-example* を使用しています）。

定義済みのメトリックのセット

すべての PostgreSQL インスタンスエクスポートは、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。これらは、2つのメジャー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-同期的レプリカの要求された最小数と最大数、および期待値と実際の観測値-レプリカクラスターモードが有効かどうかを示すフラグまたは無効-マニュアル切り替え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の実行

以下は、インスタンスの localhost:9187/metrics endpoint 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されています。

```
# HELP cnpg_collector_collection_duration_seconds Collection time duration in seconds
# TYPE cnpg_collector_collection_duration_seconds gauge
cnpg_collector_collection_duration_seconds{collector="Collect.up"} 0.0031393

# HELP cnpg_collector_collections_total Total number of times PostgreSQL was
→accessed for metrics.
# TYPE cnpg_collector_collections_total counter
cnpg_collector_collections_total 2

# HELP cnpg_collector_last_collection_error 1 if the last collection ended with
→error, 0 otherwise.
# TYPE cnpg_collector_last_collection_error gauge
cnpg_collector_last_collection_error 0

# HELP cnpg_collector_manual_switchover_required 1 if a manual switchover is
→required, 0 otherwise
# TYPE cnpg_collector_manual_switchover_required gauge
cnpg_collector_manual_switchover_required 0

# HELP cnpg_collector_pg_wal Total size in bytes of WAL segments in the /var/lib/
→postgresql/data/pgdata/pg_wal directory computed as (wal_segment_size * count)
# TYPE cnpg_collector_pg_wal gauge
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
```

(次のページに続く)

```

# 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
# 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

```

(前のページからの続き)

```

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

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

# HELP go_memstats_gc_cpu_fraction The fraction of this programs available CPU time_
↳used by the GC since the program started.
# TYPE go_memstats_gc_cpu_fraction gauge
go_memstats_gc_cpu_fraction 1.0749468700447189e-05

# HELP go_memstats_gc_sys_bytes Number of bytes used for garbage collection system_
↳metadata.
# TYPE go_memstats_gc_sys_bytes gauge
go_memstats_gc_sys_bytes 5.530048e+06

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

# HELP go_memstats_heap_idle_bytes Number of heap bytes waiting to be used.
# TYPE go_memstats_heap_idle_bytes gauge
go_memstats_heap_idle_bytes 5.8236928e+07

# 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.

```

(次のページに続く)

```
# TYPE go_memstats_last_gc_time_seconds gauge
go_memstats_last_gc_time_seconds 1.6311727586032727e+09

# HELP go_memstats_lookups_total Total number of pointer lookups.
# TYPE go_memstats_lookups_total counter
go_memstats_lookups_total 0

# HELP go_memstats_mallocs_total Total number of mallocs.
# TYPE go_memstats_mallocs_total counter
go_memstats_mallocs_total 198424

# HELP go_memstats_mcache_inuse_bytes Number of bytes in use by mcache structures.
# TYPE go_memstats_mcache_inuse_bytes gauge
go_memstats_mcache_inuse_bytes 14400

# HELP go_memstats_mcache_sys_bytes Number of bytes used for mcache structures.
↳obtained from system.
# TYPE go_memstats_mcache_sys_bytes gauge
go_memstats_mcache_sys_bytes 16384

# HELP go_memstats_mspan_inuse_bytes Number of bytes in use by mspan structures.
# TYPE go_memstats_mspan_inuse_bytes gauge
go_memstats_mspan_inuse_bytes 191896

# HELP go_memstats_mspan_sys_bytes Number of bytes used for mspan structures.
↳obtained from system.
# 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
```

(次のページに続く)

(前のページからの続き)

```

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* 名前付けの label フィールドの 1 つの中にあります。

ユーザー定義のメトリック

この機能は現在 ** beta ** 状態にあり、フォーマットは PostgreSQL Prometheus Exporter の `queries.yaml` file <!-- マスタ : ルール = master->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、ユーザーが定義できます。

```

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で作成する必要があります。

注釈: ConfigMaps と Secrets をインスタンスによって**自動的にリロードする場合、 `cnpg.io/reload` キーを持つ label を追加できます。そうでない場合は、 `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は、メッセージ: *Query with the same name already found. Overwriting the existing one.* および上書きされたクエリー名前を含むキー `queryName` を含む json ワーニングログを出力します。

ユーザ定義メトリックの例

ここで、上記の Cluster の例で被参照される単一のカスタムクエリーを含む 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"
```

(次のページに続く)

(前のページからの続き)

```
- streaming_replicas:
  usage: "GAUGE"
  description: "Number of streaming replicas connected to the instance"
```

基本的なモニタリングクエリのリストは、`cnpg-basic-monitoring.yaml file <./samples/cnpg-basic-monitoring.yaml>`__にあります。

マルチプルのデータベースで実行されるユーザ定義メトリックの例

`target_databases` オプションが複数のデータベースをリストする場合、メトリックはそれらのそれぞれから収集されます。

特定のクエリーに対して、`target_databases` のリストで `* shell-like パターン *` (つまり、`*`、`?` または `[]` を含む) を指定することにより、データベースの自動検出を有効にできます。指定された場合、演算子は、`SELECT datname FROM pg_database WHERE datallowconn AND NOT datistemplate` の実行によって返されたすべてのデータベースを追加し、`path.Match()` ルールに従ってパターンを照合することにより、ターゲットデータベースのリストを展開します。

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

返されるラベルにデータベースの名前を常を含めることをお勧めします。例、次の例のように `current_database()` 関数を使用します。

```
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
```

次に、`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ーの例を示します（それ以外の場合、前述のクエリーでは返されません）。

```
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
```

ユーザ定義メトリックの構造

すべてのカスタムクエリーの基本構造は次のとおりです。

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

使用可能なすべてのフィールドの簡単な説明を次に示します。

- `<MetricName>` : プロメテウス指標の名前

- **query** : ターゲットデータベースで実行してメトリックを生成する SQL クエリー
- **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<!-- マスタ: ルール = master-->
- **master** : **primary** と同じ (Prometheus PostgreSQL エクスポートの構文との互換性-非推奨) <!-- マスタ : ルール = master-->
- **runonserver** : クエリーを実行する PostgreSQL のバージョンを制限するセマンティックバージョンレンジ (例: ">=10.0.0" または ">=12.0.0 <=14.2.0")
- **target_databases** : **query** を実行するデータベースのリスト、または shell-like pattern 自動検出を有効にします。指定されている場合、デフォルト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>** : クエリーによって返された列の名前
- **usage** : 下記の値のいずれか
- **description** : メトリックの説明
- **metrics_mapping** : **usage** が **MAPPEDMETRIC** に設定されている場合のオプションの列マッピング

usage の可能な値は次のとおりです。

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

詳細については、Prometheus の文書の [Metric Types](#) をご覧ください。

ユーザ定義のメトリックの出力

カスタム定義のメトリックは、次のフォーマットで Prometheus エクスポートエンドポイント (:9187/metrics) によって返されます。

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

注釈: *LabelColumnName* は、*usage* が *LABEL* および *Value* に設定されたメトリックです

上記の `pg_replication` の例を考慮すると、エクスポートのエンドポイントは、呼び出されたときに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

演算子は、キーまたはシークレットで定義された一連のモニタリングクエリをクラスターのオペレーターの名前スペース内に自動的に注入するように設定できます。シークレット。演算子は `queries` キーのコンテンツを使用します。

`queries` コンテンツへの変更は、それを使用してデプロイされたすべてのクラスターにすぐに反映されます。

演算子インストールマニフェストには、`cnpg-default-monitoring` と呼ばれる定義済みの ConfigMap が付属しており、すべての Clusters で使用されます。MONITORING_QUERIES_CONFIGMAP は、演算子構成でデフォルトで `cnpg-default-monitoring` に設定されます。

デフォルトのメトリックキーを無効にする場合は、次の演算子を演算子でレベル。演算子 ConfigMap を変更するには、演算子をリスタートする必要があります。

重要: `MONITORING_QUERIES_CONFIGMAP` / `MONITORING_QUERIES_SECRET` で指定された ConfigMap または Secret は、常に固定名前 `cnpg-default-monitoring` でクラスターのネームスペースにコピーされます。そのため、デフォルトのメトリックを使用する場合は、クラスターのネームスペースにこの名前の ConfigMap を作成しないでください。

Prometheus Postgres エクスポーターとの違い

CloudNativePG は PostgreSQL Prometheus Exporter に触発されていますが、いくつかの違いがあります。特に、`cache_seconds` フィールドは CloudNativePG のエクスポーターに実装されていません。

4.19.2 演算子の監視

演算子は、`metrics` 名前付けのポート 8080 で HTTP を介して [プロメテウス演算子の例](#) 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、演算子はデフォルトの `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。

プロメテウス演算子の例

次の `PodMonitor` リソースを定義することにより、[プロメテウス演算子の例](#) を使用して演算子の展開を監視できます。

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

4.20 Logging

演算子は、PostgreSQL ログを含む標準出力に JSON フォーマットで直接ログ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。

4.20.1 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、ランタイムには変更できません。クラスターの開始後にクラスター仕様で値が変更された場合、これは新しいポッドでのみ有効になり、古いポッドでは有効になりません。

4.20.2 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、演算子は PostgreSQL CSV ログフォーマットに依存しています。CSV logformat の詳細については、PostgreSQL の文書を参照してください。

4.20.3 PGAudit ログ

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

必要なのは、必要な `pgaudit` パラメーターをクラスターの構成の `postgresql` セクションに追加することだけです。

重要: PGAudit ライブラリを `shared_preload_libraries` に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。演算子は、`shared_preload_libraries` からのライブラリの追加と削除を検出および管理します。

また、演算子は、クラスター内の使用可能なすべてのデータベースから拡張機能を作成および削除します。

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。

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を次に示します。

```
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 によって返される監査 CSV ログエントリは、残りのすべてのログと同様に、JSON フォーマットで解析され、stdout にルーティングされます。

以下の例を参照してください。

```
{
  "level": "info",
  "ts": 1627394507.8814096,
  "logger": "pgaudit",
```

(次のページに続く)

```

"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",
}

```

レコード内の各フィールドの詳細については、 [PGAudit documentation](#) < ! - マスタ : ルール = master->を参照してください。

4.20.4 その他のログ

演算子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、考えられるすべての `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` : 実行中の `initdb` から
- `pg_basebackup` : 実行中の `pg_basebackup`
- `pg_controldata` : 実行中の `pg_controldata` から

- `pg_ctl` : `pg_ctl` サブコマンドの実行から
- `pg_rewind` : 実行中の `pg_rewind` 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の `wal-archive` サブコマンドから
- `wal-restore` : インスタンスマネージャの `wal-restore` サブコマンドから

前述の構造を持つ `postgres` を除き、他のすべての可能な値は、記録されたエスケープメッセージに `msg` が設定されているだけです。

4.21 Certificates

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。オーダーをセットアップするには、演算子は以下を必要とします。

- サーバー認証局 (CA) 証明書
- サーバー認証局によって署名されたサーバー TLS 証明書
- クライアント認証局証明書
- クライアント認証局によって生成されたストリーミングレプリケーションクライアント証明書

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

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

1. ****operator managed**** : 証明書は完全に自動化された方法で演算子によって内部的に管理され、CloudNativePG2 によって作成された CA を使用して署名されます。****user provided**** : 証明書は演算子の外部で生成され、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 `cert-manager` と統合し自分自身 (以下の例を参照)

また、証明書のパートのみが 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。

4.21.1 オペレーター管理モード

デフォルトでは、演算子は単一の認証局を生成し、それをクライアント・サーバ証明書の両方に使用します。その後、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

演算子は自己署名 CA を生成し、次のキーを含む一般的な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

演算子は、生成された自己署名 CA を使用して、サーバー TLS 証明書に署名し、タイプ **kubernetes.io/tls** の Secret に格納され、インスタンスが **ssl_cert_file** および **ssl_key_file** として使用されるように構成します。

サーバーの代替 DNS 名

デフォルトの名前に加えて、生成されたサーバー TLS シークレットのパートとなる DNS サーバー代替名を指定できます。

クライアント証明書

クライアント CA シークレット

サーバー CA と同じ自己署名 CA がデフォルトで使用されます。公開部分は、署名したクライアント証明書を検証できるオーダーに、**ssl_ca_file** としてすべてのインスタンスに渡されます。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の署名に使用されます。

クライアント **streaming_replica** 証明書

演算子は、生成された自己署名 CA を使用して、ユーザ **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、レプリカの接続文字列で **sslcert** および **sslkey** として渡され、プライマリインスタンスに安全に接続できます。

4.21.2 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、*Cert-manager の例*などの個別のコンポーネントを使用してそれらを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。

注釈: 演算子は、引き続きクライアント証明書に関連する2つのシークレットを作成および管理します。

注釈: ConfigMaps と Secrets をインスタンスによって**自動的にリロードする場合、キー *cnpg.io/reload* の label 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て：

- **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 証明書のプライベートキー。

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 証明書で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
```

(次のページに続く)

(前のページからの続き)

```

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、TLS 接続に提供されたサーバ証明書を使用します。

Cert-manager の例

Cert-manager の例 を使用して自己署名 CA をセットアップし、必要な TLS サーバ証明書を生成する方法に関するシンプル例を次に示します。

```

- --
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: selfsigned-issuer
spec:
  selfSigned: {}
- --
apiVersion: v1
kind: Secret
metadata:
  name: my-postgres-server-cert
  labels:
    cnpg.io/reload: ""
- --
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: my-postgres-server-cert
spec:
  secretName: my-postgres-server-cert
  usages:
    - server auth
  dnsNames:
    - cluster-example-lb.internal.mydomain.net
    - cluster-example-rw
    - cluster-example-rw.default
    - cluster-example-rw.default.svc
    - cluster-example-r

```

(次のページに続く)

(前のページからの続き)

```

- 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 名前付けのシークレットが cert-manager によって作成され、必要なすべてのファイルが含まれ、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 cluster-example-cert-manager.yaml 展開マニフェストで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、[Cert-manager の例](#) や [hashicorp vault](#) などの個別のコンポーネントを使用してそれらを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オーダーには、次のパラメーターを指定する必要があります。

- **replicationTLSSecret** : タイプ `kubernetes.io/tls` のシークレットの名前、 ユーザ `streaming_replica` のクライアント証明書が含まれています。含まれている必要があります標準の `tls.crt` キーと `tls.key` キーの両方。
- **clientCASecret** : CA の `ca.crt` キーを含むシークレットの名前クライアント証明書を検証するために使用する必要があります。

注釈: 演算子は、サーバ証明書に関連する 2つの秘密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 シークレットキーを制御していないため、`kubectl cnpg certificate` を使用してクライアント証明書を生成できなくなりました。

注釈: ConfigMaps と Secrets をインスタンスによって**自動的にリロードする場合、`cnpg.io/reload` キーを持つ label を追加できます。そうでない場合は、`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。

Cert-manager の例

ここで、*Cert-manager の例* を使用して自己署名 CA をセットアップし、必要な TLS サーバー証明書を生成する方法に関するシンプル例を示します。

```
- --
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: selfsigned-issuer
spec:
  selfSigned: {}
- --
apiVersion: v1
kind: Secret
metadata:
  name: my-postgres-client-cert
  labels:
    cnpg.io/reload: ""
- --
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: my-postgres-client-cert
spec:
  secretName: my-postgres-client-cert
  usages:
    - client auth
  commonName: streaming_replica
  issuerRef:
    name: selfsigned-issuer
    kind: Issuer
    group: cert-manager.io
```

my-postgres-client-cert 名前付けのシークレットが cert-manager によって作成され、必要なすべてのファイルが含まれ、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、cluster-example-cert-manager.yaml 展開マニフェストで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を見つけることができます。

4.22 Client TLS/SSL Connections

CloudNativePG 演算子は、サーバー側とクライアント側で、転送中の暗号化と認証の両方で TLS / SSL と連携するように設計されています。CNPG 演算子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 CertificationAuthority (CA) が付属しています。kubectl の cnpg プラグインを使用して、パスワードを使用する代わりにユーザ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 証明書によるユーザについては、ファイル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ていることを前提として、次の手順を参照してください。bootstrap セクションの initdb 構成を介してこの規則を変更できます)。

4.22.1 新しい証明書の発行

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

```

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

```

これで証明書を検証できます：

```

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 日間有効で作成され、PostgreSQL のユーザー名に対応するシンプル CN で作成されます。これは、`pg_hba.conf` の `hostssl` entries に対して `cert` 認証メソッドを活用するために必要です。

4.22.2 TLS 証明書を介した接続のテスト

次に、CloudNativePGcluster 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` と呼ばれる次のマニフェストは、データベースクラスタが実行されているのと同じネームスペースにテストアプリケーションを含むデモ Pod を作成します。

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: cert-test
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webtest
  template:
    metadata:
      labels:
        app: webtest
    spec:
      containers:
        - image: quay.io/leonardoce/webtest:1.3.0
          name: cert-test
          volumeMounts:
            - name: secret-volume-root-ca
              mountPath: /etc/secrets/ca
            - name: secret-volume-app

```

(次のページに続く)

(前のページからの続き)

```

    mountPath: /etc/secrets/app
  ports:
    - containerPort: 8080
  env:
    - name: DATABASE_URL
      value: >
        sslkey=/etc/secrets/app/tls.key
        sslcert=/etc/secrets/app/tls.crt
        sslrootcert=/etc/secrets/ca/ca.crt
        host=cluster-example-rw.default.svc
        dbname=app
        user=app
        sslmode=verify-full
    - name: SQL_QUERY
      value: SELECT 1
  readinessProbe:
    httpGet:
      port: 8080
      path: /tx
  volumes:
    - name: secret-volume-root-ca
      secret:
        secretName: cluster-example-ca
        defaultMode: 0600
    - name: secret-volume-app
      secret:
        secretName: cluster-app
        defaultMode: 0600

```

このポッドは、CloudNativePG 演算子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プライベートキー
- `sslrootcert` : 証明書に署名した TLS 証明機関証明書インスタンスの身元を確認するために使用されるサーバー

それらは、Postgres データベースへの TLS 暗号化接続を確立するために `psql`（および `pgbench` などの他の `libpq` ベースのアプリケーション）が必要とするデフォルトのリソースを作成するために使用されます。

デフォルトでは、`psql` は現在のユーザの `~/.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` に暗号化素材の実際の場所を指定できます。上記のファイルのコンテンツは、以前に `cnpg` プラグイン `forkubectl` を使用して作成されました。

次に、アプリケーションをデプロイします。

```
kubect1 create -f cert-test.yaml
```

次に、作成した Pod を PostgreSQL クライアントとして使用して、作成した 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保証に、準備プロンプトが構成されています。

接続が機能することを確認するには、Pod のコンテナ内でインタラクティブ `bash` を実行し、必要なオプションを使用して `psql` を実行します。PostgreSQL サーバーは、読み書き 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/appuser/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
-----
--
(1 row)

```

PostgreSQL 14.2 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.3.1 20191121 (Red Hat 8.3.1-5), 64-bit

4.23 Connecting from an application

アプリケーションは、同じ Kubernetes クラスターで CloudNativePG によって作成されたサービスと連携することになっています。

- `[cluster name]-rw`
- `[cluster name]-ro`
- `[cluster name]-r`

これらのサービスは Kubernetes クラスターによって完全に管理され、*Exposing Postgres Services* で説明されているように仮想 IP のフォームを実装します。

ヒント： アプリケーションでこれらのサービスを使用し、特定の PostgreSQL インスタンスへの直接接続を避けることを強くお勧めします。後者はクラスターの存続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、演算子が生成したシークレットを使用できます。

警告: 演算子は、`[cluster name]-any` 名前付けの別のサービスを作成します。このサービスは、PostgreSQL インスタンスの検出を管理するために内部的に使用されます。アプリケーションが直接使用することは想定されていません。

4.23.1 DNS 解決

Kubernetes DNS サービスを使用して、特定のサーバーを指すことができます。KubernetesDNS サービスは、演算子が必要です。アプリケーションが PostgreSQL クラスターと同じネームスペースにデプロイされている場合は、サービスの名前を使用してこれを行うことができます。PostgreSQL クラスターが別のネームスペースにある場合は、完全修飾子 `service-name.namespace-name` を使用できます。

DNS は、推奨される発見メソッドです。

4.23.2 環境変数

PostgreSQL クラスターを含む同じネームスペース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。

例、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` : サービスの IP アドレス読み取り専用ワークロード用のすべての PostgreSQL インスタンスを指す
- `PG_DATABASE_RO_SERVICE_HOST` : の IP アドレスクラスターのすべてのホットスタンバイレプリカを指すサービス
- `PG_DATABASE_RW_SERVICE_HOST` : の IP アドレス クラスターの*プライマリ*インスタンスを指すサービス

4.23.3 秘密

PostgreSQL 演算子は、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、データベースの `postgres` ユーザと `* owner *` のユーザー名、パスワード、および作業中の `.pgpass file` がそれぞれ含まれています。

`-app` クレデンシャルは、PostgreSQL クラスターに接続するアプリケーションが使用する必要があるクレデンシャルです。

`-superuser` のものは、管理目的でのみ使用されることになっています。

4.24 Connection Pooling

CloudNativePG は、Pooler CRD を介して、PostgreSQL で最も人気のあるオープンソース接続プーラーの 1 つである *PgBouncerIntegrationStatus* とのコネクションプーリングのネイティブサポートを提供します。

一言で言えば、CloudNativePG の Pooler は、アプリケーションと PostgreSQL サービス（例 *rw* サービス）の間にある PgBouncer ポッドのデプロイであり、個別のスケラビリティ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。

4.24.1 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序文が、スイスアーミーナイフの追加ブレードのように、CloudNativePG のアーキテクチャをどのように変えるかを強調しています。PostgreSQL プライマリサービスに直接接続する代わりに、アプリケーションは PgBouncer の同等のサービスに接続できるようになり、既存の接続を再利用して、PostgreSQL 側のパフォーマンスとリソース管理を改善できます。

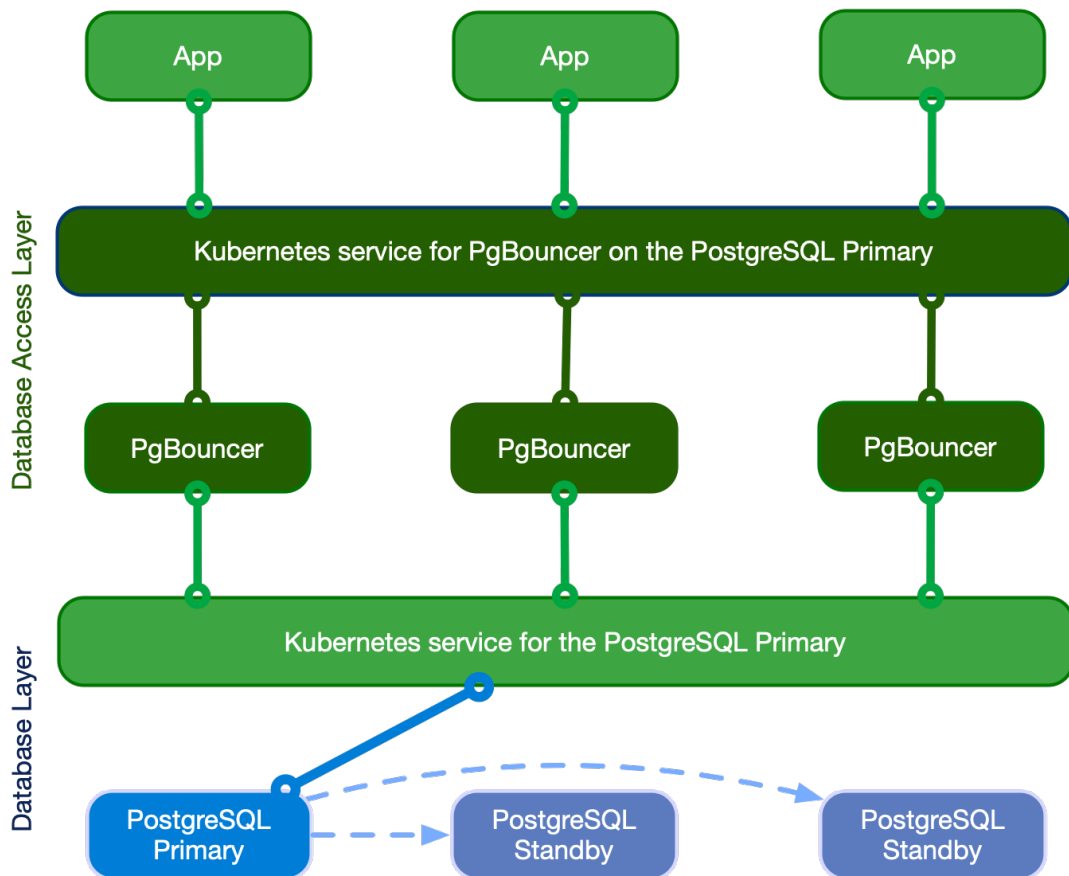


図 6: Applications writing to the single primary via PgBouncer

4.24.2 クイックスタート

CloudNativePG が PgBouncerpooler を実装する方法を説明する最も簡単な方法は、例を使用することです。

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

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

重要: プーラー名前は、同じネームスペース内のクラスター名前と決してマッチしてはなりません。

これにより、`cluster-example` という Postgres Cluster リソースと厳密に関連付けられ、`read / writeservice` (`rw`、したがって `cluster-example-rw`) によって識別されるプライマリを指す、`pooler-example-rw` (`isarbitrary` という名前) という新しい `Pooler` リソースが作成されます。

`Pooler` は、Postgres クラスターの同じネームスペースに存在する必要がありデフォルトプールサイズで実行され、それぞれ最大 1000 接続を受け入れる PostgreSQL を実行する 3 つのポッドの Kubernetes デプロイメントで構成されデータベース。。

重要: `Pooler` は PgBouncer の * フォールバックデータベースのみを設定します。つまり、クライアントから渡された接続文字列のすべてのパラメーターが PostgreSQL サーバーに中継されます ([`"Section [databases]"` in PgBouncer's documentation](<https://www.pgbouncer.org/config.html#section-databases>) を参照してください)。

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

4.24.3 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、必要なときに手動で作成することになっています。さらに、PostgreSQL クラスターごとに複数のプーラーをデプロイできます。

ノート重要なことは、Cluster および the 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 の仕組みがわかったので、可能なアーキテクチャに関して完全に自由になりました：プーラーのないクラスター、単一のプーラーのクラスター、または複数のプーラーのクラスター（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。

4.24.4 セキュリティ

PgBouncer プーラーは CloudNativePG サポートと透過的に統合され、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** による転送中の暗号化をサポートしプール。

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 [認証を使用する](#) を参照）。

コンテナは `pgbouncer` システムユーザとして実行され、`pgbouncer` データベースへのアクセスは、ローカル接続を介して、`peer` 認証を通じてのみ許可されます。

証明書

デフォルトでは、PgBouncer プーラーはクラスター自分自身が使用するものと同じ証明書を使用しますが、ユーザがそれらの証明書を提供する場合、プーラーは次のフォーマットのシークレットを受け入れます。

1. 基本認証 2. TLS3. 不透明

Opaque の場合、使用するニーズがある特定のキーを探します。これらのキーは次のとおりです。

- `tls.crt`
- `tls. キー`

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、そこからスタートことができます。

4.24.5 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メソッドです。

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、演算子：

PostgreSQL サーバーに `cnpg_pooler_pgouncer` という標準ユーザを作成します - `postgres` データベースにルックアップファンクションを作成し、実行を許可します `cnpg_pooler_pgouncer` ユーザ (PoLA) に対する特権 - このユーザの TLS 証明書を発行します - `cnpg_pooler_pgouncer` を `auth_user` として設定します - 認証に TLS 証明書を使用するように PgBouncer を構成します PostgreSQL サーバーに対する `cnpg_pooler_pgouncer` - クラスターにないことを検出すると、上記のすべてを削除します関連付けられているプーラー

重要: 独自のシークレットを指定した場合、演算子はプーラーを自動的に統合しません。

プーラーを手動で統合するには、独自のシークレットを指定した場合、クラスター内から次のクエリを実行する必要があります。

1. ロールを作成します。

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

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

2. アプリケーションデータベースごとに、`cnpg_pooler_pgouncer` に接続するパーミッションを付与します。

```
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: []
      affinity:
        podAntiAffinity:
```

(次のページに続く)

```

requiredDuringSchedulingIgnoredDuringExecution:
- labelSelector:
    matchExpressions:
    - key: app
      operator: In
      values:
      - pooler
    topologyKey: "kubernetes.io/hostname"

```

```

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: []
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - pooler
              topologyKey: "kubernetes.io/hostname"

```

3. 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。

```

edb_notranlate_5 sql CREATE OR REPLACE 関数 user_search (uname TEXT) RETURNS TABLE (username
名前、passwd text) as ‘SELECT username、passwd FROM pg_shadow WHERE username = $ 1;’ LANGUAGE
SQL SECURITY DEFINER;

```

すべての関数の取り消し user_search (text) FROM public;

```

cnpg_pooler_pgrounder への関数 user_search (text) に対する EXECUTE の付与。 console kubectl exec
-ti <PGBOUNCER_POD> -- curl 127.0.0.1:9127/metrics

```

4.24.6 PodTemplates

Pooler リソースの `template` セクションでポッドテンプレートの仕様を利用できます。詳細については、API リファレンスの `PoolerSpec` section `<api_reference.md#PoolerSpec>` を参照してください。

テンプレートを使用して、ポッドとノードのアフィニティおよびアンチアフィニティルールの詳細な制御を含め、必要に応じてポッドを設定できます。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。

PodAntiAffinity を指定する Pooler の例：

```
# 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
cnpg_pgbouncer_lists_dns_pending 0

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

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

# HELP cnpg_pgbouncer_lists_free_clients Count of free clients.
# TYPE cnpg_pgbouncer_lists_free_clients gauge
```

(次のページに続く)

```
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.
# 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.
```

(前のページからの続き)

```

# 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.
# TYPE cnpg_pgrounder_pools_sv_used gauge
cnpg_pgrounder_pools_sv_used{database="pgrounder",user="pgrounder"} 0

# HELP cnpg_pgrounder_stats_avg_query_count Average queries per second in last stat
↳period.
# TYPE cnpg_pgrounder_stats_avg_query_count gauge
cnpg_pgrounder_stats_avg_query_count{database="pgrounder"} 1

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

```

(次のページに続く)

```
# 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

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

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

(前のページからの続き)

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

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

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

注釈: `.spec.template.spec.containers` は、`PodSpec` の必須フィールドであるため、変更しない場合は明示的に `[]` に設定する必要があります。 `.spec.template.spec.containers` が設定されていない場合、マニフェストを適用しようとすると kubernetes API サーバーは次のエラーを結果ます: `error validating "pooler.yaml": error validating data: ValidationError(Pooler.spec.template.spec): missing required field "containers"`

ここで、リソースを設定し、使用するイメージを変更する例:

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

4.24.7 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マルチプルのポッドで実行するようにプーラーを構成できます。公開されたサービスにより、クライアントは PgBouncer を実行している利用可能なポッドにランダムに分散されます-これにより、基盤となるサーバー (rw サービスを使用する場合) またはサーバー (マルチプルのレプリカで ro サービスを使用する場合) への接続を自動的に管理および再利用します。

警告: インフラストラクチャがマルチプルのアベイラビリティゾーンにまたがっており、それらのレイテンシが高い場合は、ネットワークホップに注意してください。例、ゾーン 2 で実行されているアプリケーションが、ゾーン 3 で実行されている PgBouncer に接続し、ゾーン 1 の PostgreSQL プライマリを指している場合を考えます。

4.24.8 PgBouncer 設定オプション

演算子は `configuration options for PgBouncer` のほとんどを管理し、それらのサブセットのみを変更できます。

警告: 演算子が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、カスタマイズできる PgBouncer オプションのリストがあります。それらのそれぞれには、その特定のパラメータの PgBouncer 文書へのリンクが含まれています。ここで特に明記しない限り、デフォルト値は PgBouncer によって直接設定されたものです：

- `application_name_add_host`
- `autodb_idle_timeout`
- `client_idle_timeout`
- `client_login_timeout`
- `default_pool_size`
- `disable_pqexec`
- `idle_transaction_timeout`
- `ignore_startup_parameters` : `extra_float_digits,options`

に追加される -CNP で必要

- `log_connections`
- `log_disconnections`
- `log_pooler_errors`
- `log_stats` : デフォルトで

統計が Prometheus によって既に収集されている場合、無効 (0) 以下の *MonitoringConfiguration* セクションで説明されているエクスポート

- `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` マップに宣言的に記述されます。

演算子は Pooler 仕様の変更に反応し、すべての PgBouncer インスタンスがサービスを中断することなく更新された構成をリロードします。

警告: すべての PgBouncer ポッドは、仕様のパラメーターに合わせて同じ構成になります。これらのパラメーターに誤りがあると、全体 Pooler **の操作性が混乱する可能性があります。演算子は、**オプションの値を検証しません。

4.24.9 監視

Pooler の PgBouncer 実装には defaultPrometheus エクスポートが付属しており、次を実行することにより、`cnpg_pgbouncer_` プレフィックスを持つ複数のメトリックを自動的に使用可能にします。

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

CloudNativePG インスタンスと同様に、エクスポートは PgBouncer を実行している各ポッドの port 9127 で実行され、Go ランタイムに関連するメトリック (プレフィックス `go_*` を含む) も提供します。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。

```
{
  "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"
  }
}
```

cnpg_pgrounder メトリックの出力の例：

edb_notranlate_10

Clusters と同様に、[プロメテウス演算子の例](#)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。

edb_notranlate_11

4.24.10 ロギング

ログは、次の例のように、JSON フォーマットで標準出力に直接送信されます。

edb_notranlate_12

4.24.11 接続の一時停止

Pooler 仕様では、宣言構成のみを使用して、デフォルトで `false` に設定された `paused` オプションを介して、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、演算子は PgBouncer で `PAUSE` コマンドを内部的に呼び出します。

1. クエリの完了を待った後、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、演算子は PgBouncer で `RESUME` コマンドを呼び出し、Pooler で定義された PostgreSQLservice に向けてタップを再度開きます。

重要： 将来のバージョンでは、スイッチオーバーオペレーションは 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感知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、最終的に `paused` 属性を `false` に復元することで、同じ結果を得ることができます。

4.24.12 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）のパートとして機能するように設計されています。現時点では、マルチプル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。そ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、採用される基準は、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、ユースケースの大部分に対応しているが、PgBouncer の別の演算子を将来実装して、より高度でカスタマイズされたシナリオでガンマを完成させる余地があると信じる理由があります。

4.25 Kubernetes Upgrade

Kubernetes クラスターは常に更新する必要があります。Kubernetes クラスター、特に **ベアメタル** を自己管理している場合、これはさらに重要になります。

メンテナンスのために一時的にクラスターからノードを削除する管理されたダウンタイムの Kubernetes インフラストラクチャへの序文にもかかわらず、定期的な更新の計画と実行は、組織が技術的負債をクリーンアップし、ビジネスリスクを軽減する方法です（推奨読書：サイト信頼性から [Embracing Risk](#) エンジニアリングブック）。

例、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を適用したり、RAM、CPU、RAID コントローラなどの誤動作しているハードウェアコンポーネントを交換したり、クラスターを Kubernetes の最新バージョンにアップグレードしたりする必要がある場合があります。

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

1. 更新するノードからワークロードを削除します（`drain`） 2. 実際のオペレーション（システムの更新例）の実行 3. ノードをクラスターに再度参加させる（`uncordon`）

上記のプロセスでは、ワークロードをアップグレードの全期間停止するか、別のノードに移行する必要があります。

最新のケースは、Kubernetes のサービスの信頼性と自己修復機能の観点から予想されるケースですが、一時的に劣化したクラスターで動作し、アップグレードされたノードが再び起動するのを待つことが推奨される場合があります。

特に、PostgreSQL クラスターが **ノードローカルストレージ** に依存している場合、* PostgreSQL データベースが実行されている Kubernetes ワーカーノードに対してローカルストレージです。**ノードローカルストレージ（または単にローカルストレージ*）** 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ありません。ポッドで現在使用されているボリュームが、ドレイン後に異なるノードで実行されているポッドで再利用できる場合、演算子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をスキップできます）。

PostgreSQL にローカルストレージを使用する場合は、`nodeMaintenanceWindow` オプションを使用してクラスターを**メンテナンスモード**で一時的に配置して、例、物理的ノードのパーティションを拡大したり、ノードを更新したりする標準的な自己修復手順を開始することをお勧めします自分自身。

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っているか、自己修復、ローリング更新、Pod 中断予算などの制限付きで実行されています。

クラスターの `nodeMaintenanceWindow` オプションには、さらに 2 つの設定があります。

inProgress : ノードのメンテナンスウィンドウが現在進行中かどうかを示すブール値。デフォルトでは、**off** に設定されています。メンテナンス期間中、以下の `reusePVC` オプションは演算子によって評価されます。

reusePVC : メンテナンスオペレーション中に既存の PVC を再利用するかどうかを定義するブール値。デフォルトでは、**on** に設定されています。 **有効** の場合、Kubernetes はノードが再び起動するのを待ってから、既存の PVC を再利用します。PodDisruptionBudget policy は一時的に削除されます。 **無効** の場合、Kubernetes は、PostgreSQL の物理的ストリーミングレプリケーションに依存することにより、新しい PVC を使用して別のノードで thePod を強制的に再作成します。通常、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作成に待機するよりも時間がかかる場合を除き、お勧めできません。この動作は、インスタンスが 1 つしかなく、reusePVC が無効になっているクラスターには適用されません。以下のセクションを参照してください。

注釈: `kubectl drain` コマンドを実行する場合、`--delete-local-data` オプションを追加する必要があります。恐れてはいけません：PostgreSQL データディレクトリではなく、演算子によって内部的に使用される別のボリュームを参照します。

4.25.1 reusePVC が false に設定された単一インスタンスクラスター

重要: 高可用性を保証オーダーに、常に複数のインスタンスを持つクラスターを作成することをお勧めします。

reusePVC が false に設定された単一インスタンスクラスターで唯一の 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も、ユーザーがそのようなインスタンスを実行しているノードをドレインできないようにします。

ただし、このようなノードでメンテナンスが必要な場合、次の 2 つのオプションがあります。

1. reusePVC を有効にして、ダウンタイム 2 を受け入れます。別のノードでインスタンスを複製し、プライマリをスイッチします

データベースサービスのダウンタイムが環境で許容される限り、ノードのドレインは、nodeMaintenanceWindow を inProgress: true および reusePVC: true に設定するのと同じくらいシンプルです。これにより、オリジナルの PVC が使用可能になったらすぐにインスタンスを削除および再作成できます（たとえば、ノードがバックアップされるとすぐに、ノードローカルストレージを使用）。

そうでない場合は、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して、メンテナンス中のノードでオリジナルのインスタンスをシャットダウンオーダー必要があります。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。

可能な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保護します。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。データベースサイズによっては時間がかかる場合があります。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コード化されたノードで実行されている場合、演算子は自動的にスイッチオーバーを実行します。
4. クラスターを 1 つのインスタンスにスケールダウンすると、古いインスタンスが削除されます
5. これで、新しいプライマリで新しいプライマリを実行したまま、古いプライマリのノードを正常に排出できノード。

4.26 Exposing Postgres Services

このセクションでは、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、PostgreSQL サービスを外部に公開する方法について説明します。

クイックスタート を実行した場合、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でアクセスできるデータベースが必要になります。両方のサービスはポート 5432 を使用します。

ポート 5432 の外部アクセスからプライマリインスタンスにアクセスできる make にしたいとします。Kubernetes インフラストラクチャに移行する場合の典型的なユースケースは、実際には、**レガシーアプリケーション**で表されるものであり、簡単または持続的に「コンテナ化」することはできません。賢明な回避策は、仮想マシンまたは物理的サーバーにある可能性が高いアプリケーションが、同じネットワーク内の Kubernetes クラスター内の PostgreSQL データベースにアクセスできるようにすることです。

警告: パブリックネットワークからデータベースへのアクセスを許可すると、データベースが悪意のあるユーザーからの潜在的な攻撃にさらされる可能性があります。外部アクセスを許可する前にデータベースを保護するか、Kubernetes クラスターがプライベートネットワークからのみ到達可能であることを確認してください。

この例では、[NGINX Ingress Controller](#) を使用します。これは、Kubernetes プロジェクトによって直接維持され、すべての Kubernetes クラスターで設定できるためです。他の多くのコントローラーが使用可能です (包括的なリストについては、[Kubernetes documentation](#) を参照してください)。

次のことを想定しています。

- NGINX Ingress コントローラーが展開され、正常に動作する *クラスターに LoadBalancer タイプのサービスを作成することが可能です

重要: Ingress は、HTTP および HTTPS トラフィックを公開するためにのみ必要です。NGINX Ingress コントローラーは可能ですが、すべての Ingress オブジェクトが任意のポートまたはプロトコルを公開できるわけではありません。

最初のステップは、`tcp-services ConfigMap` を作成することです。このデータフィールドには、外部に公開されたポートと、内部的に指す名前空間、サービス、ポートに関する情報が含まれます。

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

次に、ドキュメントで提案されているように NGINX Ingress Controller をインストールした場合、`ingress-nginx` サービスが必要です。5432 ポートを `ingress-nginx` サービスに追加して公開する必要があります。イングレスはポート 5432 の着信接続をデータベースにリダイレクトします。

```
apiVersion: v1
kind: Service
metadata:
  name: ingress-nginx
  namespace: ingress-nginx
  labels:
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/part-of: ingress-nginx
spec:
  type: LoadBalancer
  ports:
    - name: http
```

(次のページに続く)

(前のページからの続き)

```

    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、イングレスの *ConfigMap* および *Service* の以前の変更を上書きします

これで、Kubernetes クラスターの外部から PostgreSQL クラスターに到達できるようになります。

重要: イングレスからの接続を許可するように *pg_hba* を設定してください。

4.26.1 Minikube でのテスト

Minikube では、実行中のイングレスコントローラ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、イングレスのポート 5432 のプライマリ theconnections 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、デプロイメントにパッチを適用して、ポート 5432 でのアクセスを許可します。次のコンテンツで 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
```

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
```

4.27 CloudNativePG Plugin

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

4.27.1 インストール

以下を使用して、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構築されています。

- Linux
- amd64 *アーム 5/6/7
- arm64
- s390x
- ppc64le

- macOS
- amd64
- arm64
- Windows
- 386
- amd64 *アーム 5/6/7
- arm64

4.27.2 使用

プラグインをインストールしてデプロイしたら、次のように使用をスタートできます。

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

ステータス

`status` コマンドは、以下を含むクラスターの現在のステータスの概要を提供します。

- **一般情報** : クラスターの名前、PostgreSQL のシステム ID、インスタンス、現在のタイムラインおよび WAL 内の位置
- **バックアップ** : リカバリ可能ポイント、およびによって返される WAL アーカイビングステータスプライマリからのプライマリビュー-またはレプリカクラスターの場合
- **ストリーミングレプリケーション** : `pg_stat_replication` から直接取得した情報 プライマリインスタンスのビュー
- **instances** : 各 Postgres インスタンスに関する情報。インスタンスマネージャ; スタンバイの場合、`Current LSN` フィールドは対応しますリカバリ中に再生された最新のログ先行書き込みの場所 (リプレイ LSN)。

重要: 上記のステータス情報は、異なる時間および異なる場所で取得されるため、戻り値がわずかに矛盾します。例、メインヘッダーの *Current Write LSN* の場所は、2つの異なる時間間隔で取得されるため、インスタンスステータスの *Current LSN* フィールドと異なる場合があります。

```
kubectl cnpg status sandbox
```

```
Cluster in healthy state
Name:          sandbox
Namespace:     default
System ID:     7039966298120953877
```

(次のページに続く)

(前のページからの続き)

```

PostgreSQL Image:  ghcr.io/cloudnative-pg/postgresql:14.2
Primary instance:   sandbox-2
Instances:          3
Ready instances:    3
Current Write LSN:  3AF/EAFA6168 (Timeline: 8 - WAL File: 000000080000003AF000000075)

Continuous Backup status
First Point of Recoverability:  Not Available
Working WAL archiving:         OK
Last Archived WAL:             000000080000003AE000000079 @ 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
- - - - -
➔-----
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

```

また、`--verbose` または単に `-v` を追加することで、より詳細なバージョンのステータス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:14.2
Primary instance:    sandbox-2
Instances:           3
Ready instances:     3
Current Write LSN:    3B1/61DE3158 (Timeline: 8 - WAL File: 0000000080000003B1000000030)

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
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

```

(次のページに続く)

```

shared_buffers = 16GB
shared_memory_type = sysv
shared_preload_libraries = pg_stat_statements
ssl = on
ssl_ca_file = /controller/certificates/client-ca.crt
ssl_cert_file = /controller/certificates/server.crt
ssl_key_file = /controller/certificates/server.key
synchronous_standby_names = ANY 1 ("sandbox-1","sandbox-3")
unix_socket_directories = /controller/run
wal_keep_size = 512MB
wal_level = logical
wal_log_hints = on
cnpg.config_sha256 = 3cfa683e23fe513afaee7c97b50ce0628e0cc634bca8b096517538a9a4428efc

```

PostgreSQL HBA Rules

```

# Grant local access
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: 000000080000003B000000001D @ 2021-12-14T10:20:42.
→272815Z
Last Failed WAL: -

```

Streaming Replication status

Name	Sent LSN	Write LSN	Flush LSN	Replay LSN	Write Lag
→Flush Lag	Replay Lag	State	Sync State	Sync Priority	
sandbox-1	3B1/61E26448	3B1/61DF82F0	3B1/61DF82F0	3B1/61DF82F0	00:00:00.000333
→00:00:00.000333	00:00:00.005484	streaming	quorum	1	
sandbox-3	3B1/61E26448	3B1/61E26448	3B1/61DF82F0	3B1/61DF82F0	00:00:00.000756
→00:00:00.000756	00:00:00.000756	streaming	quorum	1	

Instances status

(前のページからの続き)

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 フォーマットの出力もサポートしています。

プロモート

このコマンドの意味は、クラスター内のポッドをプライマリに **promote** することです。したがって、メンテナンス作業をスタートしたり、クラスターのスイッチオーバーシチュエーション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プロモートさせることができます

```
kubectl cnpg promote cluster-example 2
```

証明書

CloudNativePG 演算子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、クラスター名前、およびこの証明書のユーザを指定する必要があります

```
kubectl cnpg certificate cluster-cert --cnpg-cluster cluster-example --cnpg-user-↪
↪appuser
```

Secret が作成されたら、**kubectl** を使用して取得できます

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

また、次のコマンドを使用したプレーンテキストでの同じコンテンツ：

```
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。

- 演算子にロールアウトのリスタートを調整するよう要求する特定のクラスターに対して。これは適用するのに便利です `ConfigMap` などのクラスター依存オブジェクトの構成変更カスタムモニタリングクエリを含む。
- 単一インスタンスのリスタートを要求します。インスタンスがクラスターのプライマリまたはポッドの削除と再作成 それはレプリカです。

```
# 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]
```

インプレースリスタートが要求されているが、スイッチオーバーなしでは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リスタートよりも優先されます。これの一般的なケースは、PostgreSQL イメージのマイナーアップグレードです。

注釈: `ConfigMaps` と `Secrets` をインスタンスによって**自動的に**リロードする場合、`cnpg.io/reload` キーを持つ `label` 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演算子に要求します。これは、カスタムモニタリングクエリを含む `ConfigMap` など、クラスターに依存するオブジェクトに構成の変更を適用するのに役立ちます。

次のコマンドは、特定のクラスターのすべての構成をリロードします。

```
kubectl cnpg reload [cluster_name]
```

メンテナンス

`kubectl cnpg maintenance` コマンドは、ネームスペース全体で 1 つ以上のクラスターを変更し、メンテナンスウィンドウの値を設定するのに役立ちます。次のフィールドが変更されます。

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

これを引数 `set` および `unset` として受け入れ、`set` の場合は `inProgress` を `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のリストとそれらの新しい値の確認を求めます。これが受け入れられると、このアクションはリスト内のすべてのクラスターに適用されます。

Kubernetes クラスター内のすべての PostgreSQL をメンテナンスで設定する場合は、次のコマンドを記述する必要があります。

```
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 ファイルにバンドルします。稼働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、演算子のデプロイメント、構成、およびイベントに関する情報を提供するように演算子に要求します。

重要: Secrets および ConfigMaps のすべての機密情報は編集済みです。データマップには **** keys **** が表示されますが、値は空になります。フラグ `-S / --stopRedaction` は、リダクションを無効にし、値を表示します。ご自身のリスクでのみ使用してください。これはプライベートデータを共有します。

注釈: デフォルトでは、演算子ログは収集されませんが、`--logs` フラグを使用して演算子ログ収集を有効にできます

- **デプロイメント情報** : 演算子のデプロイメントと演算子のポッド
- **構成** : 演算子名前空間の Secrets および ConfigMaps
- **イベント** : 演算子名前空間のイベント
- **webhook 設定** : webhook 設定の変更と検証

- **webhook サービス** : webhook サービス
- **logs** : 演算子 Pod (オプション、デフォルトでは off) の JSON 行フォーマットのログ

このコマンドは、YAML フォーマットのさまざまなマニフェストを含む ZIP ファイルを生成します (デフォルトでは、ただし `-o` フラグを使用して JSON に設定可能)。 `-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ファイル名が zip ファイル用に作成されます。

```
kubectl cnpg report operator
```

結果として

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

`-f` フラグセットしている場合:

```
kubectl cnpg report operator -f reportRedacted.zip
```

ファイルを解凍すると、タイムスタンプ付きの最上位フォルダーが作成され、ディレクトリが整理されます。

```
unzip reportRedacted.zip
```

結果として:

```
Archive:  reportRedacted.zip
  creating: report_operator_<TIMESTAMP>/
  creating: report_operator_<TIMESTAMP>/manifests/
 inflating: report_operator_<TIMESTAMP>/manifests/deployment.yaml
 inflating: report_operator_<TIMESTAMP>/manifests/operator-pod.yaml
 inflating: report_operator_<TIMESTAMP>/manifests/events.yaml
 inflating: report_operator_<TIMESTAMP>/manifests/validating-webhook-configuration.
→yaml
 inflating: report_operator_<TIMESTAMP>/manifests/mutating-webhook-configuration.yaml
 inflating: report_operator_<TIMESTAMP>/manifests/webhook-service.yaml
 inflating: report_operator_<TIMESTAMP>/manifests/cnpg-ca-secret.yaml
 inflating: report_operator_<TIMESTAMP>/manifests/cnpg-webhook-cert.yaml
```

機密情報が編集されてい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"
```

(次のページに続く)

(前のページからの続き)

```
managedFields:
- apiVersion: v1
  fieldsType: FieldsV1
  fieldsV1:
```

-S (`--stopRedaction`) オプションをアクティブにすると、秘密が表示されます：

```
kubectl cnpg report operator -f reportNonRedacted.zip -S
```

機密情報をビューしようとしていることを通知します。

```
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： `kubectl get cluster -o yaml` と同じクラスター情報
- クラスターポッド：クラスター名にマッチングするクラスター名前空間のポッド
- クラスタージョブ：存在する場合、クラスター名にマッチングするクラスター名前空間にジョブ
- イベント：クラスター名前空間のイベント
- `podlogs`：JSON-lines フォーマットのクラスターポッド（オプション、デフォルトではオフ）のログ
- `ジョブブログ`：ジョブによって作成された Pod のログ（オプション、デフォルトでは off）JSON 行フォーマット

`cluster` サブコマンドは、`operator` が行うように、`-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レポート名前が使用されます。無効です。

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` フラグを使用してクラスターログの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の名前にない限り、クラスター名前と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
```

結果として:

```
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/
```

(次のページに続く)

(前のページからの続き)

```

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

```

4.28 Automated failover

プライマリで予期しないエラーが発生した場合、クラスターは**フェイルオーバーモード**に入ります。これは、例次の場合に発生する可能性があります。

- プライマリポッドにディスク障害があります
- プライマリポッドが削除されます
- プライマリの `postgres` コンテナには、あらゆる種類の持続的な障害があります

フェイルオーバーシナリオでは、プライマリが正常に動作していると想定することはできません。

上記のようなケースの後、プライマリポッドのレディネスプローブは失敗し始めます。これは、コントローラーの調整ループで取得されます。コントローラーは、2つのステップでフェイ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。この状態の変更により、プライマリポッドが強制的にシャットダウンされ、レプリカ上の WAL レシーバーが保証に停止します。クラスターはフェールオーバーフェーズ（「フェイルオーバー」）でマークされます。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名前付けます。選択し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操作を再開します。一方、以前のプライマリポッドはリスタートし、それが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。

保証 2 フェーズプロシージャは、WAL レシーバーが正常に停止できるようにし、障害が発生したプライマリが再スタート時に WAL のストリーミングをリスタートしないようにします。これらのセーフガードは、新しいプライマリとレプリカの間のタイムラインの不一致を防ぎます。

障害のあるプライマリがシャットダウンされている間：

1. まず、タイムアウトとして `.spec.switchoverDelay` 秒で PostgreSQL の**高速シャットダウン**を試みます。この正常なシャットダウンは、保留中の WAL をアーカイブしようとします。高速シャットダウンが失敗するか、タイムアウトが超過すると、PostgreSQL の**即時シャットダウン**が開始されます。

4.28.1 RTO および RPO のインパクト

フェイルオーバーにより、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. プライマリが失敗し始め、コントローラがフェイルオーバー手順を開始する前に、送信中のクエリ、WAL 書き込み、チェックポイントなどの操作が失敗する場合があります。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。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すべて停止します。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され、新しいプライマリがまだ開始されていない間、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、データロスしません。

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 のビューからは安全ですが、通常のオペレーションへの結果を遅らせる可能性があります、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の説明で [インスタスマネージャ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。これを低い値に設定すると、RPO よりも RTO が優先されますが、クラスターレベルやバックアップレベルでデータロスする可能性があります。反対に、高い値に設定すると、スイッチオーバー中により長い時間アクティブなプライマリがないクラスターを残しながら、データロスのリスクを取り除くことができます。

4.29 Troubleshooting

このページでは、Kubernetes クラスター展開で CloudNativePG のトラブルシューティングを行う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 管理者として、[kubectrl](#) ページをブックマークする必要があります！

4.29.1 スタート前に

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 `make` ことができるのは、基礎となる Kubernetes システムに関する明確な情報を提供することです。

必ず確認してください：

- 使用している Kubernetes のディストリビューションとバージョン PostgreSQL が実行されているノードの仕様

- 可能な限り、ストレージを含む実際の *StorageConfiguration* について稼動に入る前に行ったクラスとベンチマーク。
- クラスタで使用している関連 Kubernetes アプリケーション（つまり、プロメテウス、グラファナ、イスティオ、証明書マネージャー、…）

便利なユーティリティ

必須の `kubectl` ユーティリティに加えて、トラブルシューティングのために、次のプラグイン/ユーティリティをシステムで使用可能にすることをお勧めします。

- `kubectl` の `cnpg`
- `jq`、軽量で柔軟なコマンドライン JSON プロセッサ
- `grep`、1 つ以上の入力ファイルを検索します

指定されたパターンへのマッチを含む行の場合。ほとんどの *nix ディストリビューションですでに利用可能です。Windows OS を使用している場合は、`grep` の代わりに `findstr` を使用するか、直接 `ws` を使用できます。お好みの *nix ディストリビューションをインストールし、上記のツールを使用します。

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 が期待するように標準出力にログを記録し、*JSONformat* に直接記録します。そのため、`kubectl logs` コマンドを使用して、特定のリソースからログを取得する必要があります。

詳細については、次を入力してください。

```
kubectl logs --help
```

ヒント: JSON ログは機械読み取りには最適ですが、人間にとっては読み取りがハードです。`jq` コマンドを使用して使いやすさを改善することをお勧めします。例、`|jq -C` で `kubectl logs` コマンドを*パイプ*することができます。

注釈: 以下のセクションでは、CloudNativePG のトラブルシューティングに関して、さまざまなリソースに関するログを取得する方法の例を示します。

4.29.2 オペレーター情報

デフォルトでは、CloudNativePG 演算子は 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）。

以下を実行することにより、演算子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、演算子が実行しているポッドが 1 つあり、`cnpg-controller-manager` で始まる名前で識別されます。演算子を高可用性用にセットアップした場合、より多くのエントリーが必要です。これらのポッドは、`cnpg-controller-manager` 名前付けのデプロイメントによって管理されます。

以下を使用して、`pod <POD>` で実行されている演算子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>
```

演算子に関する詳細情報を収集する

次を実行して、CloudNativePG 演算子の展開（マルチ演算子の展開がある場合）のすべてのポッドからログを取得します。

```
kubectl logs -n cnpg-system \
  deployment/cnpg-controller-manager --all-containers=true
```

ちなみに: 上記のコマンドに `-f` フラグを追加して、リアルタイムでログを追跡できます。

以下を実行して、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 演算子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:14.2-3
Primary instance: cluster-example-1
Instances:     3
Ready instances: 3
Current Write LSN: 0/50000000 (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 0/50000000 0/50000000 0/50000000 0/50000000 00:00:00    ✓
→00:00:00      00:00:00      streaming async      0
cluster-example-3 0/50000000 0/50000000 0/50000000 0/50000000 00:00:00.10033 ✓
→00:00:00.10033 00:00:00.10033 streaming async      0

Instances status
Name          Database Size  Current LSN  Replication role  Status  QoS
→Manager Version
-----
→-----
cluster-example-1 33 MB          0/50000000  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
```

4.29.3 クラスター情報

NAMESPACE namespace の <CLUSTER> クラスターのステータスは、次の方法で確認できます。

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

出力：

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

上記の例は、すべてが `* ready *` 状態で、`<CLUSTER>-1` がプライマリである 3 つのインスタンスの健全な PostgreSQL クラスターを報告します。

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

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

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

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

ちなみに: `--verbose` オプションを追加すると、より多くの情報をプリントできます。

リスタート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実行することもできます。レプリカを昇格します。
証明書を管理します。
ループをリロードロードして設定変更を適用します。
詳細については、`cnpg` の文書を参照してください。

PostgreSQL コンテナイメージ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:14.2-3
```

注釈: また、`kubectl-cnpg status -n <NAMESPACE> <CLUSTER_NAME>` を使用して同じ情報を取得できます。

4.29.4 ポッド情報

以下を使用して、特定の PostgreSQLcluster 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

以下を実行することにより、ポッドが失敗しているかどうか/どのように失敗しているかを確認できます。

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

以下を使用して、特定の PostgreSQL インスタンスのすべてのログを取得できます。

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

検索を PostgreSQL プロセスのみに制限する場合は、次を実行できます。

```
kubectl logs -n <NAMESPACE> <CLUSTER>--<N> | \
jq select(.logger=="postgres") | .record.message
```

次の例も、ユーザフレンドリフォーマット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のログを確認します。

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

特定の PostgreSQL ポッドから致命的な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` ワードにマッチング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
```

特定のポッドから 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、実行中の `|` 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
```

4.29.5 バックアップ情報

以下を使用して、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: CloudNativePG のバージョン 1.10.0 でバックアップラベルが導入されました。そのため、そのバージョン以上で作成されたリソースのみにこのような label が含まれます。

4.29.6 ストレージ情報

次のように、調査またはトラブルシューティング中にクラスターで使用される StorageClass を再確認してコンテキストを確認すると便利な場合があります。

```
STORAGECLASS=$(kubectl get pvc <POD> -o jsonpath={.spec.storageClassName})
kubectl get storageclasses $STORAGECLASS -o yaml
```

多くの場合、クラスターはデフォルトの StorageClass を使用して作成されるため、ここではクラスターポッドの 1 つから StorageClass を取得しています。

4.29.7 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限り知ることが戦略的に重要です。

Kubernetes クラスター内のノードのリストを取得するには、次を使用します。

```
# 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、ポッドがどこに分散されているかを理解するために重要です- affinity/anti-affinity rules and/or tolerations を使用している場合は非常に便利です。

4.29.8 条件

多くのネイティブ kubernetesobjects [like here](#) と同様に、クラスターは `status.conditions` も公開します。これにより、クラスター全体のヘルス状態に依存する代わりに、特定のイベントが発生するのを「待つ」ことができます。現在利用可能な条件は次のとおりです。

- LastBackupSucceeded
- 継続的なアーカイブ

特定の条件を待つ方法

- バックアップ：

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

- 継続的なアーカイブ：

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

以下は、失敗した状態を含む `cluster.status` のスニペットです。

```
$ kubectl get cluster/<cluster-name> -o yaml
.
.
.
status:
  conditions:
  - message: unexpected failure invoking barman-cloud-wal-archive: exit status
    2
    reason: Continuous Archiving is Failing
    status: "False"
    type: ContinuousArchiving

  - message: exit status 2
    reason: Backup is failed
    status: "False"
    type: LastBackupSucceeded
```

4.29.9 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、これがフルディスクである可能性が `logsuggest` である場合、おそらくインスタンスの `PersistentVolumeClaim` のサイズを増やす必要があります。文書の [ボリューム拡張](#) をご覧ください。

ポッドが `Pending` 状態でスタックしている

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把握する必要があります。

```
kubectl describe pod -n <NAMESPACE> <POD>
```

この考えられる原因のいくつかは次のとおりです：

- `nodeSelector` にマッチングするノードはありません
- 許容値は、ノードの汚染とマッチするように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達している、または一時的な問題がある

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
```

バックアップが設定されていないときにレプリカが同期しない

メンテナンスの理由により、レプリカがしばらくオフになる場合があります（Kubernetes ノードが空になった場合を考えてください）。クラスターにバックアップが設定されていない場合、レプリカがバックアップされるときに、プライマリに存在しない WAL ファイル（`:ref:postgresql` セクション<postgresql セクション>` で説明されているように WAL 管理ポリシーに従って既にリサイクルされている）が必要になり、同期が外れることがあります。

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なくなり、PVC を削除し、演算子にレプリカを再構築させる必要があります。

動的にプロビジョニングされた永続ボリュームに依存していて、PV 自分自身を削除することに自信がある場合は、次の方法で削除できます。

```

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

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

```

4.30 Fencing

CloudNativePG のフェンシングは、誤動作していると思われる場合に、PostgreSQL クラスターの 1 つ、複数、またはすべてのインスタンスのデータを保護する究極のプロセスです。インスタンスがフェンスされると、ポッドが実行され続ける間、PostgreSQL サーバープロセス（`postmaster`）が確実にシャットダウンされます。これにより、フェンスが解除されるまで、ポッド上のデータは PostgreSQL によって変更されず、ファイルシステムデバッグおよびトラブルシューティングの目的で調査できます。

4.30.1 インスタンスをフェンスする方法

CloudNativePG では、以下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

ワイルドカードは、リストのコンテンツを介して制御されリスト。これは、JSON 形式のインスタンス名のリストを必要とし JSON、演算子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: '["*"]'` は、すべてのインスタンスをクラスター。

注釈は、例 `kubectl annotate` コマンドを使用して Kubernetes オブジェクトに手動で設定するか、`kubectl cnpg fencing on` サブコマンドを使用して透過的に設定できます。

```

# 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 "*"

```

以下は、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"]
[...]
```

4.30.2 フェンシングを解除する方法

フェンシングは、注釈をクリアするか、別の値に設定することで解除できます。

フェンシングの場合、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 ""
```

4.30.3 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プロシージャが開始されます。これは、タイムアウトを `.spec.stopDelay` に設定した最初のスマートシャットダウンと、必要に応じて高速シャットダウンで構成されます。次に：

- ポッドは生き続けます
- ポッドは `* Ready *` としてマークされません

Postgres インスタンスを起動する必要のないすべての変更は以下を含む調整済み - 構成ファイル - 証明書とすべての暗号化資料

警告: *Cluster* の少なくとも 1 つのインスタンスがフェンスされると、*Cluster* のステータスを安定 (stable) と見なすことができないため、その *Cluster* のフェイルオーバー/スイッチオーバーはフェンスが解除されるまでブロックされます。

特に、**プライマリインスタンス**がフェンスされる場合、ポストマスタープロセスはシャットダウンされますが、フェイルオーバーは発生せず、アプリケーションの操作性が中断されます。フェンスが解除されると、フェイルオーバーが発生することなく、プライマリインスタンスが再び起動されます。

そのため、可能な場合はレプリカインスタンスのみをフェンスすることをお勧めしユーザ。

警告: プライマリが *Cluster* 内の唯一のフェンスされたインスタンスであり、ポッドが削除された場合、フェイルオーバーが実行されます。古いプライマリのフェンスが解除されると、そのインスタンスはスタンバイ (新しいプライマリのフォロワー) として再起動されます。

フェンスされたインスタンスが削除された場合、ポッドは通常どおり再作成されますが、ポストマスターは起動しません。これは、instances が `Crashlooping` の場合に非常に役立ちます。

4.31 End-to-End Tests

CloudNativePG 演算子は、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、演算子が PostgreSQL クラスターを正しく展開および管理できます。

さらに、各コミットについて次の Kubernetes バージョンがテストされ、開発プロセスの初期段階で障害とバグの検出が保証されます。

- 1.23
- 1.22
- 1.21
- 1.20
- 1.19

以下の PostgreSQL バージョンがテストされています。

- PostgreSQL 14
- PostgreSQL 13
- PostgreSQL 12
- PostgreSQL 11
- PostgreSQL 10

Kubernetes および PostgreSQL の各テストバージョンについて、**種類** を使用して Kubernetescluster が作成され、そのクラスターで次の一連の E2E テストが実行されます。

演算子のインストール。 Cluster の作成。データストレージ用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。ユーザー提供のサーバーおよび/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。フェイルオーバー。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際のローリング更新。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。同期レプリケーション。バックアップからの復元。* Azure BLOB ストレージで Barman Cloud を使用してバックアップから復元します。* **NodeSelector** を使用したポッドアフィニティ。メトリックコレクション。JSON ログフォーマット; * ConfigMap を介したオペレーター構成。オペレータポッドの削除。オペレータポッドの立ち退き。オペレーターのアップグレード。オペレーターの高可用性。ノードドレイン。フェイルオーバーが 10 秒より小さい場合のプライマリエンドポイントスイッチ。* 20 秒より小さいでスイッチ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。* 60 秒より小さいで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。

4.32 Container Image Requirements

Kubernetes の CloudNativePG 演算子は、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。

- パスに存在する必要がある PostgreSQL 10 以降の実行可能ファイル：
 - `initdb`
 - `postgres`
 - `pg_ctl`
 - `pg_controldata`
 - `pg_basebackup`
- パスに含まれている必要がある 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ている（オプション-PGAudit が必要な場合のみデプロイされたクラスター内）
- 賢明なロケール設定

CloudNativePG がインスタスマネージャで上書きするため、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、**マルチプル/オプションのホットスタンバイサーバーアーキテクチャを備えたプライマリ**でのみ 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、それらを ghcr.io で公開します。

4.32.1 イメージタグの要件

イメージ名前は Docker に有効なものであれば何でもかまいませんが、CloudNativePGoperator は* イメージタグ*に依存して、イメージによって実行される Postgres majorversion 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) でスタート、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、ドット、アンダースコア、またはマイナス符号を前に付けて続けることができます。

受け入れられるイメージタグの例：

- 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。

4.33 Operator Capability Levels

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説明します。

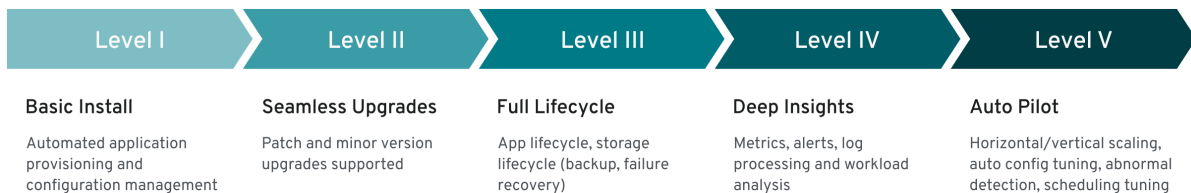


図 7: Operator Capability Levels

重要: Operator Capability Levels model に基づいて、CloudNativePG オペレーターから**「レベル V-自動パイロット」**機能セットが期待できます。

各機能レベルは、演算子が提供する管理機能の特定の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、演算子の将来の作業と実装のガイドと考えています。

4.33.1 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** および **構成**が含まれます。このカテゴリには、操作性や 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エクスペリエンスの向上が含まれます。

重要: このレベルの**情報セキュリティ**パートを考慮します。

宣言的な設定によるオペレーターの展開

演算子は、4つのメジャー CustomResourceDefinition オブジェクト、Cluster、Pooler、Backup、および ScheduledBackup を定義する Kubernetes マニフェストを使用して宣言的にインストールされます。

宣言的な設定による PostgreSQL クラスターの展開

PostgreSQL クラスター（オペランド）は、完全に宣言的な方法で Cluster カスタムリソースを使用して定義されます。PostgreSQL のバージョンは、CR で定義されているオペランドコンテナイメージによって決定され、要求されたレジストリから自動的に取得されます。オペランドをデプロイするとき、演算子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イメージのオーバーライド

演算子は、PostgreSQL が内部にある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、演算子は、PostgreSQL がサポートし、イメージで公開されている最新の安定（stable）メジャーバージョンの最新の利用可能なマイナーバージョンを使用します。/ CR で `imageName` attribute を設定して直接スタンバイアーキテクチャ。演算子は、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に加えて、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` もサポートしています。

ラベルと注釈

演算子は、Kubernetes インフラストラクチャでの CloudNativePG 展開の組織を改善する目的で、クラスターのメタデータで定義されたラベルと注釈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

Patroni や Stolon などの外部ツールに依存して Kubernetes クラスターポッド内の PostgreSQL インスタンスを調整する代わりに、オペレーターは各ポッド内で、`/controller/manager` という名前のファイルで演算子実行可能ファイルを注入します。このアプリケーションは、基盤となる PostgreSQL インスタンスを制御し、PostgreSQL クラスタートポロジに基づいてポッドのステータスをインスタンス自体と調整するために使用されます。インスタスマネージャは、プローブ用に `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によってフィルター処理され、必要に応じて `postgres` プロセスに転送され、外部イベントに対する高速で制御された反応が行われます。インスタスマネージャは Go で記述されており、外部依存関係はありません。

ストレージ構成

ストレージは、データベースワークロードの重要なコンポーネントです。ストレージに関して、Kubernetes のネイティブ機能とリソースを活用することで、オペレーターは、基盤となる Kubernetes 環境が提供できるものに基づいて、ワークロード要件に適したストレージを選択するのに十分な柔軟性を提供します。これは、パブリッククラウド環境で特定のストレージクラスを選択すること、または生成された PVC を CR のストレージパラメータの PVC テンプレートを介して微調整することを意味しデータベース。

レプリカ構成

演算子は、`instances` という単一のパラメータを使用して、クラスター内のレプリカを自動的に検出します。1 に設定すると、クラスターはレプリカなしの単一のプライマリ PostgreSQL インスタンスで構成されます。1 よりも高い場合、演算子は、自動フェイルオーバーによる高可用性やスイッチオーバー操作によるローリング更新など、`instances -1` レプリカを管理します。

データベース設定

演算子は、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。演算子は、読み書き、読み取り、ユーザ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 つのデフォルト Postgres を通じて、データベースへのアクセスを透過的に管理し演算子。同じ名前。必要に応じて、データベース名前とユーザ名前の両方を指定できます。クラスターの実行に構成は必要ありませんが、CR の `postgresql` セクションで PostgreSQL 実行時構成と PostgreSQL Host-Based Authentication ルールの両方をカスタマイズできます。

ポッドセキュリティポリシー

InfoSec 要件の場合、演算子はどのコンテナに対しても特権モードを必要とせず、読み取り専用ルートファイルシステムを適用して、演算子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` セクションでは、ポッドおよび永続的なボリュームなどの関連リソースを aKubernetes 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、演算子は以下をサポートします。

- ポッドアフィニティとアンチアフィニティ
- ノードセクター
- 汚染と許容

コマンドラインインタフェース

CloudNativePG には独自のコマンドラインインタフェースはありません。PostgreSQL クラスター管理エクスペリエンスを強化および簡素化するために、`cnpg` と呼ばれるプラグインを提供することにより、Kubernetes の最適なコマンドラインインタフェース `kubectl` に依存しています。

クラスターの現在のステータス

演算子は、CR の状況セクションをクラスターの監視状況で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によって継続的に監視されます：インスタスマネージャは、制御された PostgreSQL インスタンスに必要な変更を適用して、クラスターの必要なステータスに収束するマネージャがあります（例、クラスターステータスがポッド-1 であると報告した場合プライマリ、ポッド -1 は自分自身をプロモートニーズがあり、他のポッドはポッド -1 をフォローする必要があります。 `kubectl` の `cnpg` プラグインは、同じステータスを使用して詳細を提供します。

オペレーターの認証権限

演算子は、自分自身の証明権限を自動的に作成します API サーバーと演算子自分自身の間の安全な通信を保証に、Webhook サーバーが使用するリーフ証明書を作成し、演算子証明権限と署名します。

クラスターの証明権限

演算子は、すべての PostgreSQLcluster に対して証明権限を自動的に作成します。これは、パスワードの代わりにストリーミングレプリケーションスタンバイサーバを含むクライアントの認証用に TLS 証明権限を発行および更新するために使用されます。cert-manager との統合も含まれます。証明書は、`kubectl` の `cnpg` プラグインで発行できます。

TLS 接続

演算子は、TLS / SSL 接続を透過的かつネイティブにサポートして、クラスターの証明権限を使用してセキュリティを強化するためのクライアント/サーバー通信を暗号化します。カスタムサーバ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。

ストリーミングレプリケーションの証明書認証

演算子は、パスワード（およびそのための秘密）に頼るのではなく、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

演算子を使用すると、PostgreSQL 設定の `Cluster` リソース YAML セクションに変更を適用し、設定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るようにすることができます。
現在の制限：ALTER SYSTEM による変更は検出されません。強制されません。

PostgreSQL の基本的な LDAP 認証

演算子は、[PostgreSQL documentation: LDAP authentication](#) で説明されているように、* シンプル bind または 検索+ bind *モードを使用して、PostgreSQL クライアントの LDAP 認証を構成できます。

複数のインストール方法

演算子は、`kubectlapply` を介して Kubernetes マニフェストからインストールでき、パブリックおよびプライベートクラウド環境での従来の Kubernetes インストールで使用できます。さらに、演算子のヘルムチャートも利用できます。

設定より規約

演算子は、構成パラダイムに関する規則をサポートし、標準のデフォルト値を決定すると同時に、それらをオーバーライドしてカスタマイズすることもできます。いくつかの YAML コード行で **Cluster CRD** を使用して、PostgreSQL クラスターのデプロイメントを指定できます。

4.33.2 レベル 2-シームレスなアップグレード

機能レベル 2 とは、**演算子と実際のワークロード**（この場合は PostgreSQL サーバー）の更新を有効にすることです。これには、**PostgreSQL マイナーリリース更新**（通常はセキュリティおよびバグ修正）および**メジャーオンラインアップグレード**が含まれます。

演算子のアップグレード

新しい展開として演算子をシームレスにアップグレードできます。演算子の変更は、オペランドの変更を必要としません-`instancemanager` のインジェクションのおかげです。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、演算子のアップグレード後に **インスタスマネージャ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（およびその後のスイッチオーバー）は不要です。

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` パラメータの変更の一環として、宣言的な構成アプローチを使用してアップグレードできます。オペレーターは、PostgreSQL のメジャーアップグレードを防止し、メジャーバージョン内のマイナー PostgreSQL リリースの両方の方向に進むことを可能にします（更新とロールバックを有効にします）。

スタンバイサーバが存在する場合、演算子は既存のポッドをドロップし、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、演算子はスイッチオーバーを続行します以前のプライマリ（`unsupervised`）を更新するか、`kubectl` の `cnpg` プラグインを介してユーザがスイッチオーバープロシージャ（`supervised`）を手動で発行するのを待機します。実際のデータベースワークロードに基づきます。

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、例、 `Setting up primary`、`Creating a new replica`、`Cluster in healthy state`、`Switchover in progress`、`Failing over`、`Upgrading cluster` など、クラスターの高可用性ステータスを伝えます。

4.33.3 レベル 3-完全なライフサイクル

機能レベル 3 では、演算子が **ビジネス継続性** および **スケーラビリティ** の側面を管理する必要があります。**災害リカバリ** は、データベースのバックアップとリカバリの両方が正しく機能することを必要とするビジネス継続性コンポーネントです。出発点としての目標は $RPO < 5$ 分を達成することですが、長期的な目標は $RPO = 0$ 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、PostgreSQL のネイティブ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、演算子がフェールオーバーおよびスイッチオーバー操作を実行できるようにするビジネス継続性のもう 1 つの重要コンポーネントです。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などの PostgreSQL 物理的レプリケーションの制御、(カスケード) レプリケーションクラスターなど。
- コネクションプーリング、を介してパフォーマンスと制御を改善します pgBouncer によるコネクションプーリングレイヤー。

PostgreSQL バックアップ

演算子は、物理ベースバックアップと連続 WAL アーカイビングに基づく PostgreSQL のネイティブ連続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、演算子は現在、オブジェクトストア（AWS S3 および S3 互換、Azure Blob Storage、Google Cloud Storage、MinIO などのゲートウェイ）のバックアップのみをサポートしています。

WAL アーカイビングとベースバックアップは、S3 プロトコルの宛先 URL（例、AWS S3 バケット内の特定のフォルダーを指す）を指定することにより、クラスターレベルで宣言的にクラスター定義で `backup` パラメータを介して定義され、オプションで汎用エンドポイント URL。連続バックアップの前提条件である WAL アーカイビングでは、ユーザからのさらなるアクションは不要です。演算子は、`archive_command` を自動的かつ透過的に設定し、`barman-cloud-wal-archive` に依存して WALfiles を定義されたエンドポイントに送信します。ユーザーは、圧縮アルゴリズム、およびアーカイブ内の 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、最初の WAL ファイルセットの出荷を開始する前に `barman-cloud-check-wal-archive` コマンドを実行することにより、アーカイブ先の正確性を自動的にチェックします。

ベースバックアップは、オンデマンド（`Backup custom` リソース定義を使用）またはスケジュール（`cron` のような構文を使用して `ScheduledBackup customer` リソース定義を使用）の 2 つの方法で定義できます。これらは両方とも、ジョブの `on barman-cloud-backup`（`applicationcontainer` イメージのパートとして配布されます）を使用して、WAL ファイルとともに同じエンドポイントでバックアップをリレーします。

`barman-cloud-wal-restore` と `barman-cloud-backup` は両方とも、GNU GPL 3 の条件の下でアプリケーションコンテナイメージに配布されます。

バックアップからの完全な復元

演算子を使用すると、`barman-cloud-backup` を使用して取得された既存のアクセス可能なバックアップから開始して、新しいクラスターを（設定とともに）ブートストラップできます。ブートストラッププロセスが完了すると、演算子はリカバリモードでインスタンスを開始し、指定されたアーカイブから利用可能プライマリすべてのスタンバイファイルを再生し、リカバリを終了してプライマリとして開始します。アーカイブから取得する WAL。

バックアップからのポイントインタイムリカバリ (PITR)

演算子を使用すると、タイムスタンプ、`labeled`、またはトランザクション ID で定義された特定の時点に既存のバックアップを回復することにより、新しい PostgreSQL クラスターを作成できます。この機能は、完全な `restoreone` の上に構築され、`PostgreSQL for PITR` で利用可能なすべてのオプションをサポートします。

同期レプリケーションによる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 `CloudNativePGcluster` で **ゼロデータ損失** (RPO = 0) を達成します。演算子は、いつでも利用可能な予想される同期的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。演算子は、クォーラムの次の式 (q) を使用して、クラスター内の使用可能な 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 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の利点を活用して、PostgreSQL クラスターのクロス Kubernetes クラスタポートロジを定義します。`replica` オプションを使用して、独立したクラスターをセットアップし、同じメジャーバージョンの別の PostgreSQL ソースからのデータを継続的に複製できソース。TLS を介した直接ストリーミング接続が 2 つのエンドポイントから許可されている限り、ソースは Kubernetes の外部でも、物理的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、リカバリオブジェクトストア (BarmanCloud フォーマットのバックアップ) から作成できます。`pg_basebackup` を介したストリーミング経由。レプリカクラスターは、ストリーミングでの PostgreSQL データベースのビジネス継続性を劇的に改善し、マルチプルのファイル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します（現在、Kubernetes を待機している間に、データセンター間のマニュアル切り替えが必要です）フェデレーション機能）。

活性と準備のプロープ

演算子は、PostgresContainers の活性プロープと準備プロープを定義します。これらのプロープは、kubelet によって呼び出されます。これらはそれぞれ、インスタスマネージャによって直接管理されるウェブサーバの `/healthz` および `/readyz` エンドポイントにマッピングされます。liveness プロープは、`pg_isready` 実行可能ファイルに基づいており、ポッドは終了コード 0（サーバーは正常に接続を受け入れています）および 1（サーバーは拒否しています）接続、例スタートアップ中）。準備状況プロープは、シンプルクエリー（`;`）を発行して、サーバーが接続を受け入れる準備ができていることを確認します。

ローリング展開

演算子はローリング展開をサポートしてダウンタイムを最小限に抑え、PostgreSQL クラスターが公開されている場合、サービスは初期化または更新中に読み取り専用トラフィックを利用可能なポッドにのみ負荷分散します。

レプリカのスケールアップとスケールダウン

演算子を使用すると、PostgreSQL クラスター内のインスタンスの数を増減できます。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、ユーザが `kubectl scale` コマンドを使用できるようにする「スケール」サブリソースを宣言します。

Kubernetes ノードのメンテナンスウィンドウと PodDisruptionBudget

演算子は、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なくなり、指定された数のインスタンスを作成できます。ノードのドレインオペレーション中に PodDisruptionBudget が適用され、クラスターサービスの中断を防ぎます。

この戦略は、ストレージがすべてのワーカーノード間で共有される Kubernetes クラスターには正しいものですが、ローカルストレージを使用するクラスターやプライベートクラウドにインストールされたクラスターには最適なソリューションではない場合があります。演算子を使用すると、メンテナンスウィンドウを指定し、基になるノードの排除に対する反応を設定できます。メンテナンスウィンドウセクションの ReusePVC オプションを使用して、使用する戦略を指定できます。削除されたインスタンスに別の PVC に新しいストレージを割り当てるか、基になるノードが再び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 PostgreSQL ように見える場合に、それらのインスタンスのデータを保護するサーバプロセスです。これにより、フェンスが解除されるまで、ポッドのデータは PostgreSQL によって変更されず、ファイルシステムをデバッグおよびトラブルシューティングの目的で調査できるようになります。

ポッドでの永続ボリュームストレージの再利用

演算子がユーザによって削除されたポッドを作成するニーズがある場合、または Kubernetes メンテナンスオペレーションによって削除されたポッドを作成する必要がある場合、可能であれば、`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 とメモリのリクエストと制限

演算子により、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を制御および管理できます。特定の `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 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* を使用したコネクションプーリングのネイティブサポートを提供します。アーキテクチャのビューから、PgBouncer コネクションプーラーの一般的な実装は、インスタンスへのクエリーフローを最適化し、基になる PostgreSQL リソースの使用をより効率的にするデータベースにアクセスするための新しいレイヤーを導入します。PostgreSQL サービスに直接接続する代わりに、アプリケーションこれで PgBouncer サービスに接続し、既存の接続の再利用をスタートできます。

4.33.4 レベル 4-深い洞察

機能レベル 4 は、**監視可能性**：特に、モニタリング、アラート、トレンド、ログ処理です。これには、Prometheus、Grafana、Fluent Bit などの外部ツールの使用、および JSON フォーマットでエラーログを直接出力するための PostgreSQL エンジンの拡張が含まれる場合があります。

CloudNativePG は、柔軟なモニタリングとロギングのための業界標準およびコミュニティで受け入れられているツールと簡単に統合するために必要なすべてを提供するように設計されています。

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はプラグラブルフレームワークを提供し、`metrics` ポート (9187) でリスニングする独自の Web サーバーを介して、モニタリングモニタリングおよびアラートツールのメトリックをエクスポートするエンドポイントを公開します。演算子は、オープンソースと互換性のある構文プロジェクトは、コンテキストに統合および適応できる 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。いくつかの基本的な指標とダッシュボードの例を提供します。

JSON フォーマットの PostgreSQL エラーメッセージの標準出力ロギング

すべてのログメッセージは、標準の PostgreSQL エラーメッセージチャンネルの `postgres` など、タイムスタンプ、ログレベル、ログエントリーのタイプの最初のレベル定義とともに、JSON フォーマットで標準出力に配信されます。その結果、CloudNativePG によって管理されるすべての Pod はソースデータ型として JSON をサポートするダウンストリームログ処理スタックと簡単かつ直接統合できます。

リアルタイムのクエリーモニタリング

CloudNativePG は透過的かつネイティブに以下をサポートします。

- 必須の `:ref:`pg_stat_statements`` を有効にする `<pg_stat_statements を有効にする>`、すべての SQL の計画および実行統計の追跡を可能にします PostgreSQL サーバーによって実行されるステートメント。
- `:ref:`auto_explain`` を有効にする `<auto_explain を有効にする>`、
遅いステートメントの実行計画をロギングする手段を提供します `EXPLAIN` を手動で実行することなく、自動的に（追跡に役立ちます最適化されていないクエリ）。

監査

CloudNativePG を使用すると、データベースおよびセキュリティの管理者、監査者、オペレーターは PGAudit (forPostgreSQL) を使用してデータベースアクティビティを追跡および分析できます。このようなアクティビティは JSON ログに直接フロー、Fluentd などの一般的なログブローカーを使用して適切なダウンストリームターゲットに適切にルーティングできます。

Kubernetes イベント

リソースの作成、ノードの削除、アップグレード処理など、Kubernetes API が期待するメジャー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。

4.33.5 レベル 5-オートパイロット

機能レベル 5 は、観測可能性レイヤーから出現した異常と洞察の発見を通じて、**自動スケーリング**、**修復**、**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イルオーバー

プライマリで障害が検出された場合、演算子は、最も整合し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。結果として、各アライブポッドのインスタスマネージャは、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合わせて必要な手順を開始し自分自身。前のプライマリが復旧した場合、同じメカニズムはアプリケーションからのアクセスを防ぎ、サーバーで `pg_rewind` を実行し、スタンバイとして再起動することにより、`asplit-brain`。

スタンバイの自動作成

スタンバイをホスティングしているポッドが取り外された場合、演算子はスタンバイサーバを再作成するプロシージャを開始します。

4.34 Configuration Samples

このセクションでは、PostgreSQL Cluster をセットアップするための構成ファイルの例を見つけることができます。

利用可能なオプションのリストについては、API Reference を参照してください。

4.35 Commercial support

CloudNativePG は独立したオープンソースプロジェクトであり、どの会社も支持していません。

これは、CloudNativePG に関連する製品またはサービスを提供するサードパーティ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ます。

Company	URL	Regions/Countries
[EDB](https://enterprisedb.com)	enterprisedb.com	Global

API リファレンス

4.36 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>	ポッドの非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、演算子はポッドの非アフィニティを定義します	<code>*bool</code>
<code>topologyKey`</code>	非アフィニティ構成に使用する TopologyKey。詳細については、k8s の文書を参照してください- 必須	<code>string</code>
<code>nodeSelector`</code>	NodeSelector は、ポッドを実行できるノードを定義するために使用されるキーと値のペアのマップです。詳細： https://kubernetes.io/docs/concepts/configuration/assign-pod-node/	<code>map[string]string</code>
<code>tolerations`</code>	許容値は、汚染されたノードで実行できるようにオーダーに、すべてのポッドに設定する必要があります許容値のリストです。詳細： https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/	<code>[]corev1.Toleration</code>
<code>podAntiAffinityType`</code>	PodAntiAffinityType を使用すると、クラスターインスタンス間のポッドの非アフィニティをスケジューリング時に強力な要件と見なす必要があるかどうかをユーザが決定できます。許可される値は「preferred」（空の場合のデフォルト）または「required」です。「必須」に設定すると、既存のノードがすべて必要なポッドの非アフィニティルールにマッチしない場合、新しい kubernetes ノードが追加されるまでインスタンスが保留状態になる可能性があります。詳細： https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#inter-pod-affinity-and-anti-affinity	<code>string</code>

次のページに続く

表 9 – 前のページからの続き

Name	Description	Type
<i>additionalPodAntiAffinity</i>	AdditionalPodAntiAffinity は、EnablePodAntiAffinity が true（デフォルト）に設定されている場合に演算子によって生成されたものに追加されるポッドの非アフィニティー用語を指定するか、false に設定された場合に排他的に使用されます。	*corev1.PodAntiAffinity
<i>additionalPodAffinity</i>	AdditionalPodAffinity では、すべてのクラスターのポッドに渡されるポッドアフィニティー用語を指定できます。	*corev1.PodAffinity

4.37 AzureCredentials

AzureCredentials は、ファイルを Azure Blob Storage にアップロードするために使用される資格情報の種類です。接続文字列には、必要なすべての情報が含まれています。接続文字列が指定されていない場合、ストレージアカウント名前と次のいずれか（1 つのみ）が必要です。

- storageKey-storageSasToken

Name	Description	Type
<i>connectionString</i>	使用する接続文字列	[*SecretKeySelector](#SecretKeySelector)
<i>storageAccount</i>	データをアップロードするストレージアカウント	[*SecretKeySelector](#SecretKeySelector)
<i>storageKey</i>	ストレージアカウント名前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](#SecretKeySelector)
<i>storageSasToken</i>	ストレージアカウント名前と組み合わせて使用される共有アクセス署名	[*SecretKeySelector](#SecretKeySelector)

4.38 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>	NA	[metav1.ObjectMeta](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/#objectmeta-v1-meta)

次のページに続く

表 11 – 前のページからの続き

Name	Description	Type
<code>spec`</code>	バックアップの望ましい動作の仕様。詳細: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status	[BackupSpec](#BackupSpec)
<code>status`</code>	バックアップの最新の状態。このデータは最新ではない可能性があります。システムによって設定されます。読み取り専用。詳細: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status	[BackupStatus](#BackupStatus)

4.39 BackupConfiguration

BackupConfiguration は、クラスターのバックアップの取得方法を定義します。現在、サポートされている唯一のバックアップメソッドは `barmanObjectStore` です。詳細と例については、文書の「バックアップリカバリー」セクションを参照してください

Name	Description	Type
<code>barmanObjectStore</code>	barman-cloud ツールスイートの構成	[*BarmanObjectStoreConfiguration](#BarmanObjectStoreConfiguration)
<code>retentionPolicy`</code>	RetentionPolicy は、バックアップおよび 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 <code>XXu`</code> のフォームで表されます。 <code>XX`</code> は正の整数で、 <code>u`</code> は <code>[dwm]-日、週、月</code> です。	string

4.40 BackupList

BackupList にはバックアップのリストが含まれます

Name	Description	Type
<i>metadata</i>	標準リストメタデータ。詳細: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#types-kinds	[metav1.ListMeta](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/#listmeta-v1-meta)
<i>items</i>	バックアップのリスト-必須	[[]Backup](#Backup)

4.41 BackupSource

BackupSource には、復元元のバックアップと、それを正しく復元するために必要な情報が含まれています。

Name	Description	Type
<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。証明書発行者と barman-cloud-wal-archive のエラーを回避するために、自己署名証明書を使用する場合に役立ちます。	[*SecretKeySelector](#SecretKeySelector)

4.42 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	[LocalObjectReference](#LocalObjectReference)

4.43 BackupStatus

BackupStatus は、観測されたバックアップの状態を定義します

Name	Description	Type
<i>s3Credentials</i>	S3 にデータをアップロードするために使用される資格情報	[*S3Credentials](#S3Credentials)
<i>azureCredentials</i>	Azure Blob Storage にデータをアップロードするために使用される資格情報	[*AzureCredentials](#AzureCredentials)
<i>googleCredentials</i>	Google Cloud 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	[*GoogleCredentials](#GoogleCredentials)

次のページに続く

表 16 – 前のページからの続き

Name	Description	Type
<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。証明書発行者と barman-cloud-wal-archive のエラーを回避するために、自己署名証明書を使用する場合に役立ちます。	[*SecretKeySelector](#SecretKeySelector)
<code>`endpointURL`</code>	データをクラウドにアップロードするために使用されるエンドポイント。自動エンドポイント検出をオーバーライドします	string
<code>`destinationPath`</code>	バックアップを保存するパス (s3:// bucket / パス/to / folder) このパスは、異なる宛先フォルダーで、WAL とデータに使用されます-必須	string
<code>`serverName`</code>	S3 上のサーバー名前。このパラメータを省略するとクラスター名前が使用されます	string
<code>`encryption`</code>	S3 API に必要な暗号化メソッド	string
<code>`backupId`</code>	Barman バックアップの ID	string
<code>`phase`</code>	最後のバックアップステータス	BackupPhase
<code>`startedAt`</code>	バックアップが開始されたとき	[*metav1.Time](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/#time-v1-meta)
<code>`stoppedAt`</code>	バックアップが終了したとき	[*metav1.Time](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/#time-v1-meta)
<code>`beginWal`</code>	開始 WAL	string
<code>`endWal`</code>	エンディング WAL	string
<code>`beginLSN`</code>	開始 xlog	string
<code>`endLSN`</code>	終わりの 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ます。	string
<code>`commandError`</code>	エラーの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識別する情報	[*InstanceID](#InstanceID)

4.44 BarmanObjectStoreConfiguration

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

Name	Description	Type
<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	<code>[*S3Credentials](#S3Credentials)</code>
<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	<code>[*AzureCredentials](#AzureCredentials)</code>
<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資格情報	<code>[*GoogleCredentials](#GoogleCredentials)</code>
<code>`endpointURL`</code>	データをクラウドにアップロードするために使用されるエンドポイント。自動エンドポイント検出をオーバーライドします	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。証明書発行者と barman-cloud-wal-archive のエラーを回避するために自己署名証明書を使用する場合に役立ちます	<code>[*SecretKeySelector](#SecretKeySelector)</code>
<code>`destinationPath`</code>	バックアップを保存するパス (s3:// bucket / パス / to / folder) このパスは、異なる宛先フォルダーで、WAL とデータに使用されます-必須	string
<code>`serverName`</code>	S3 上のサーバー名前。このパラメータを省略するとクラスター名前が使用されます	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 ファイルは圧縮されずに保存され、バケットのデフォルトポリシーに従って、オブジェクトストアで暗号化されていない場合があります。	<code>[*WalBackupConfiguration](#WalBackupConfiguration)</code>
<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格納され、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。	<code>[*DataBackupConfiguration](#DataBackupConfiguration)</code>
<code>`tags`</code>	タグは、Barman の <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 の <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

4.45 BootstrapConfiguration

BootstrapConfiguration には、PostgreSQL クラスターの作成方法に関する情報が含まれています。サポートされている方法の中では、1つのブートストラップメソッドのみを定義できます。指定されていない場合、`initdb` はブートストラップメソッドとして使用されます。詳細については、文書のブートストラップページを参照してください。

Name	Description	Type
<code>`initdb`</code>	<code>initdb</code> 経由でクラスターをブートストラップします	<code>[*BootstrapInitDB](#BootstrapInitDB)</code>
<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します	<code>[*BootstrapRecovery](#BootstrapRecovery)</code>
<code>pg_basebackup</code>	互換性のある別の PostgreSQL インスタンスの物理的バックアップを取るクラスターをブートストラップします	<code>[*BootstrapPgBaseBackup](#BootstrapPgBaseBackup)</code>

4.46 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、文書のブートストラップ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: <code>app</code> 。-必須	string
<code>`owner`</code>	アプリケーションで使用されるインスタンス内のデータベースの所有者の名前。デフォルトは <code>`database`</code> キーの値です。-必須	string
<code>`secret`</code>	ユーザ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	<code>[*LocalObjectReference](#LocalObjectReference)</code>
<code>`options`</code>	クラスターの作成時に <code>initdb</code> 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、矛盾した構成が発生する可能性があります。代わりに明示的に提供されたパラメーターを使用してください。定義されている場合、明示的な値は無視されます。	[]string
<code>`dataChecksums`</code>	<code>-k</code> オプションを <code>initdb</code> 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: <code>false</code>)	<code>*bool</code>
<code>`encoding`</code>	<code>initdb</code> のオプション <code>--encoding`</code> として渡される値 (デフォルト: <code>UTF8</code>)	string
<code>`localeCollate`</code>	<code>initdb</code> のオプション <code>--lc-collate`</code> として渡される値 (デフォルト: <code>C</code>)	string

次のページに続く

表 19 – 前のページからの続き

Name	Description	Type
<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: <code>C</code>)	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、PostgreSQL のデフォルト: 16MB)	int
<code>`postInitSQL`</code>	クラスターの作成直後にスーパーユーザとして実行される SQL クエリのリスト-細心の注意を払って使用する (デフォルトでは空)	[]string
<code>postInitApplication-SQL</code>	作成直後にアプリケーションデータベースでスーパーユーザとして実行される SQL クエリのリスト-細心の注意を払って使用する (デフォルトでは空)	[]string
<code>`postInitTemplateSQL`</code>	クラスターの作成後に <code>`template1`</code> でスーパーユーザとして実行される SQL クエリのリスト-細心の注意を払って使用する (デフォルトでは空)	[]string

4.47 BootstrapPgBaseBackup

BootstrapPgBaseBackup には、既存の PostgreSQL クラスターの物理的バックアップを取るために必要な構成が含まれています

Name	Description	Type
<code>source</code>	物理的バックアップを取る必要があるサーバーの名前-必須	string

4.48 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、スーパーユーザ用に選択したパスワードでパスワードを変更した後、復元されたプライマリからすべてのインスタンスを複製する完全なクラスターをブートストラップするためにそれを使用します。詳細については、文書のブートストラップ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	[*Backup-Source](#BackupSource)
<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

次のページに続く

表 21 – 前のページからの続き

Name	Description	Type
<i>recoveryTarget</i>	デフォルトでは、リカバリプロセスはアーカイブ内の利用可能なすべての WAL ファイルを適用します (完全リカバリ)。ただし、PostgreSQL で期待されるようにリカバリオブジェクトを指定することで、一貫性のある状態に到達するか、ポイントインタイム (PITR) に回復する (タイムスタンプ、トランザクション ID、LSN など)。)。詳細： https://www.postgresql.org/docs/current/runtime-config-wal.html#RUNTIME-CONFIG-WAL-RECOVERY-TARGET	[*RecoveryTarget](#RecoveryTarget)

4.49 証明書の構成

CertificatesConfiguration には、サーバ証明書をハンドル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 を使用して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。 次を含む： <code>-ca.crt</code> ：CA クライアント接続文字列で <code>sslrootcert</code> として使用されるサーバー証明書を検証するために使用する必要があります。 <code>-ca.key</code> ：サーバー SSL 証明書の生成に使用されるキー。	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 <code>kubernetes.io/tls</code> タイプの秘密。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
<i>replicationTLSSecret</i>	<code>streaming_replica`</code> ユーザとして認証するクライアント証明書を含む <code>kubernetes.io/tls</code> タイプの秘密。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 22 – 前のページからの続き

Name	Description	Type
<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。 次を含む： -ca.crt：CA 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。 -ca.key：クライアント証明書の生成に使用されるキー。ReplicationTLSSecret が提供されている場合、これは省略可能	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

4.50 証明書ステータス

CertificatesStatus には、構成証明書と関連する有効期限が含まれます。

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

4.51 クラスター

Cluster は PostgreSQL API のスキーマです

Name	Description	Type
<code>metadata</code>	NA	[metav1.ObjectMeta](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/#objectmeta-v1-meta)
<code>spec`</code>	クラスターの望ましい動作の仕様。詳細： https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status	[ClusterSpec](#ClusterSpec)

次のページに続く

表 24 – 前のページからの続き

Name	Description	Type
<code>status</code>	クラスターの最近観察されたステータス。このデータは最新ではない可能性があります。システムによって設定されます。読み取り専用。詳細: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status	[ClusterStatus](#ClusterStatus)

4.52 ClusterCondition

ClusterCondition は、特定の時点でのクラスターオブジェクトの状態を記述します

Name	Description	Type
<code>type</code>	条件のタイプ。	ClusterConditionType
<code>status</code>	条件のステータス。True、False、Unknown のいずれか。	ConditionStatus
<code>lastTransitionTime</code>	条件がある状態から別の状態に最後に遷移した時間。	[*metav1.Time](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/#time-v1-meta)
<code>reason</code>	条件の最後の遷移の理由。	string
<code>message</code>	トランジションに関する詳細を示す人間が読めるメッセージ。	string

4.53 ClusterList

ClusterList にはクラスターのリストが含まれます

Name	Description	Type
<code>metadata</code>	標準リストメタデータ。詳細: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#types-kinds	[metav1.ListMeta](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/#listmeta-v1-meta)
<code>items</code>	クラスターのリスト-必須	[[]Cluster](#Cluster)

4.54 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](#EmbeddedObjectMetadata)
<code>`imageName`</code>	コンテナイメージの名前。タグ (<code><image>:<tag></code>) と確定的で反復可能なデプロイメントのダイジェスト (<code><image>:<tag>@sha256:<digestValue></code>) の両方をサポートします	string
<code>`imagePullPolicy`</code>	イメージプルポリシー。 <i>Always</i> 、 <i>Never</i> 、または <code>`IfNotPresent`</code> のいずれか。定義されていない場合、デフォルトは <code>`IfNotPresent`</code> になります。更新できません。詳細： https ://kubernetes.io/docs/concepts/containers/images#updating-images	corev1.PullPolicy
<code>`postgresUID`</code>	イメージ内の <code>`postgres`</code> ユーザの UID、デフォルトは <code>`26`</code>	int64
<code>`postgresGID`</code>	イメージ内の <code>`postgres`</code> ユーザの GID、デフォルトは <code>`26`</code>	int64
<code>`instances`</code>	クラスターに必要なインスタンスの数-必須	int
<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 の場合、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>	同期レプリケーションクォーラムの目標値。準備完了スタンバイの数がこれより少ない場合は、減らすことができます。未定義または 0 は、同期レプリケーションを無効にします。	int
<code>`postgresql`</code>	PostgreSQL サーバーの構成	[PostgresConfiguration](#PostgresConfiguration)
<code>`bootstrap`</code>	このクラスターをブートストラップする手順	[*BootstrapConfiguration](#BootstrapConfiguration)
<code>`replica`</code>	レプリカクラスタ構成	[*ReplicaClusterConfiguration](#ReplicaClusterConfiguration)
<code>`superuserSecret`</code>	スーパーユーザのパスワードを含むシークレット。定義されていない場合、ランダムに生成されたパスワードで新しいシークレットが作成されます	[*LocalObjectReference](#LocalObjectReference)

次のページに続く

表 27 – 前のページからの続き

Name	Description	Type
<code>enableSuperuserAccess</code>	このオプションを有効にすると、演算子は`SuperuserSecret`を使用して`postgres`ユーザ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、演算子は自動的にパスワードを作成します）。このオプションを無効にすると、演算子は`SuperuserSecret`コンテンツを無視し、自動作成時に削除し、`NULL`に設定して`postgres`ユーザのパスワードをブランクにします。デフォルトで有効。	<code>*bool</code>
<code>`certificates`</code>	CA および関連証明書の構成	<code>[*CertificatesConfiguration](#CertificatesConfiguration)</code>
<code>`imagePullSecrets`</code>	画像のプルに使用されるプルシークレットのリスト	<code>[[]LocalObjectReference](#LocalObjectReference)</code>
<code>`storage`</code>	インスタンスのストレージの構成	<code>[StorageConfiguration](#StorageConfiguration)</code>
<code>`startDelay`</code>	PostgreSQL インスタンスが正常にスタートするために許可される秒単位の時間（デフォルト 30）	<code>int32</code>
<code>`stopDelay`</code>	PostgreSQL インスタンスが正常にシャットダウンできる秒数（デフォルトは 30）	<code>int32</code>
<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するのに許可される秒単位の時間。デフォルトより大きいのは 40000000 で、1 年は秒単位で、無限の遅延をシミュレートするのに十分な大きさです	<code>int32</code>
<code>`affinity`</code>	ポッドのアフィニティ/アンチアフィニティルール	<code>[AffinityConfiguration](#AffinityConfiguration)</code>
<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。	<code>[corev1.ResourceRequirements](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/#resourcerequirements-v1-core)</code>
<code>primaryUpdateStrategy</code>	すべてのレプリカが正常に更新された後、ローリング更新プロシージャ中にプライマリーサーバをアップグレードするために従う戦略：自動化（ <i>unsupervised</i> -デフォルト）またはマニュアル（ <i>supervised</i> ）が可能	<code>PrimaryUpdateStrategy</code>
<code>`primaryUpdateMethod`</code>	すべてのレプリカが正常に更新された後、ローリング更新プロシージャ中にプライマリーサーバ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> ）で可能	<code>PrimaryUpdateMethod</code>
<code>`backup`</code>	バックアップに使用される構成	<code>[*BackupConfiguration](#BackupConfiguration)</code>

次のページに続く

表 27 – 前のページからの続き

Name	Description	Type
<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](#NodeMaintenanceWindow)
<code>monitoring`</code>	このクラスターのモニタリングインフラストラクチャの構成	[*MonitoringConfiguration](#MonitoringConfiguration)
<code>externalClusters`</code>	構成で使用する外部クラスターのリスト	[[ExternalCluster](#ExternalCluster)]
<code>logLevel`</code>	インスタンスのログレベル、次の値のいずれか：エラー、ワーニング、情報（デフォルト）、デバッグ、トレース	string

4.55 ClusterStatus

ClusterStatus は、クラスターの観測状態を定義します

Name	Description	Type
<code>instances`</code>	クラスター内のインスタンスの総数	int
<code>readyInstances`</code>	クラスター内の準備完了インスタンスの総数	int
<code>instancesStatus`</code>	インスタンスのステータス	map[utils.PodStatus][]string
<code>latestGeneratedNode`</code>	生成された最新のノードの ID（ノード名前の衝突を避けるために使用）	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
<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	[]string
<code>resizingPVC`</code>	ResizingPVC 状態のすべての PVC のリスト。	[]string
<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	[]string
<code>healthyPVC`</code>	ぶら下がりも初期化もされていないすべての PVC のリスト	[]string
<code>writeService`</code>	現在の書き込みポッド	string
<code>readService`</code>	読み取りポッドの現在のリスト	string
<code>phase`</code>	クラスターの現在のフェーズ	string
<code>phaseReason`</code>	現在のフェーズの理由	string

次のページに続く

表 28 – 前のページからの続き

Name	Description	Type
<code>`secretsResourceVersion`</code>	演算子によって管理される秘密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、秘密データをリフレッシュするインスタスマネージャの利益のために行われます	<code>[SecretsResourceVersion](#SecretsResourceVersion)</code>
<code>`configMapResourceVersion`</code>	演算子が管理する configmap のリソースバージョンのリスト。ここでのすべての変更は、configmap データをリフレッシュするインスタスマネージャの利益のために行われます。	<code>[ConfigMapResourceVersion](#ConfigMapResourceVersion)</code>
<code>`certificates`</code>	デフォルトで初期化された CA および関連証明書の構成。	<code>[CertificatesStatus](#CertificatesStatus)</code>
<code>`firstRecoverabilityPoint`</code>	RFC3339 フォーマットの日付として保存された最初の回復可能ポイント	string
<code>`cloudNativePGCommitHash`</code>	この演算子が実行しているコミットハッシュ番号	string
<code>`currentPrimaryTimestamp`</code>	プライマリへの最後の実際のプロモーションが発生したときのタイムスタンプ	string
<code>`targetPrimaryTimestamp`</code>	新しいプライマリの最後のリクエストが発生したときのタイムスタンプ	string
<code>`poolerIntegrations`</code>	クラスターを参照するプーラーが必要とする統合	<code>[*PoolerIntegrations](#PoolerIntegrations)</code>
<code>cloudNativePGOperatorHash</code>	演算子のバイナリのハッシュ	string
<code>`onlineUpdateEnabled`</code>	OnlineUpdateEnabled は、クラスター内でオンラインアップグレードが有効になっているかどうかを示します	bool
<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	bool
<code>`conditions`</code>	クラスターオブジェクトの条件	<code>[[]ClusterCondition](#ClusterCondition)</code>

4.56 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
<code>key</code>	選択するキー-必須	string

4.57 ConfigMapResourceVersion

ConfigMapResourceVersion は、演算子が管理するシークレットのリソースバージョンです

Name	Description	Type
<i>metrics</i>	メトリックス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

4.58 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しながら、バックアップファイル（テーブルスペースごとの tar ファイル）を圧縮します。使用可能オプションは、空の文字列（圧縮なし、デフォルト）、 <i>gzip</i> 、 <code>`bzip2`</code> 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（パケットがそのためにまだ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（パケットポリシー、デフォルトを使用）、 <code>`AES256`</code> および <i>aws:kms</i> です。	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 はチェックポイントをできるだけ早く完了します。デフォルトでは <code>`false`</code> です。	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>

4.59 EmbeddedObjectMetadata

EmbeddedObjectMetadata には、クラスターに関連するすべてのリソースに継承されるメタデータが含まれます

Name	Description	Type
<code>labels`</code>	NA	map[string]string
<code>annotations`</code>	NA	map[string]string

4.60 ExternalCluster

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

Name	Description	Type
<code>`name`</code>	サーバー名前、必須-必須	string
<code>connectionParameters`</code>	dbname、ホスト、username などの接続パラメータのリスト	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 証明書へのリファレンス	[*corev1.SecretKeySelector](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/#secretkeyselector-v1-core)
<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リファレンス	[*corev1.SecretKeySelector](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/#secretkeyselector-v1-core)
<code>`sslRootCert`</code>	このインスタンスへの接続に使用される SSL CA 公開キーへのリファレンス	[*corev1.SecretKeySelector](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/#secretkeyselector-v1-core)
<code>`password`</code>	サーバーへの接続に使用されるパスワードへのリファレンス	[*corev1.SecretKeySelector](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/#secretkeyselector-v1-core)
<code>`barmanObjectStore`</code>	barman-cloud ツールスイートの構成	[*BarmanObjectStoreConfiguration](#BarmanObjectStoreConfiguration)

4.61 GoogleCredentials

GoogleCredentials は、Google Cloud Storage の資格情報のタイプです。GKE 環境内で実行する場合でも、これを指定ニーズがあります。

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

4.62 InstanceID

InstanceID には、インスタンスを識別する情報が含まれます

Name	Description	Type
<code>`podName`</code>	ポッド名前	string
<code>ContainerID</code>	コンテナ ID	string

4.63 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
<code>prefix</code>	バインド認証オプションのプレフィックス	string
<code>suffix</code>	バインド認証オプションのサフィックス	string

4.64 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
<code>`baseDN`</code>	ユーザ検索を開始するルート DN	string
<code>`bindDN`</code>	ディレクトリにバインドするユーザの DN	string

[次のページに続く](#)

表 37 – 前のページからの続き

Name	Description	Type
<code>`bindPassword`</code>	ユーザがディレクトリにバインドするためのパスワードの秘密	[*corev1.SecretKeySelector](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/#secretkeyselector-v1-core)
<code>searchAttribute</code>	ユーザー名とマッチする属性	string
<code>`searchFilter`</code>	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

4.65 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
<code>`server`</code>	LDAP ホスト名または IP アドレス	string
<code>`port`</code>	LDAP サーバーのポート	int
<code>`scheme`</code>	使用する LDAP スキーマ、可能なオプションは <code>`ldap`</code> および <code>`ldaps`</code>	LDAPScheme
<code>`tls`</code>	LDAP over TLS を有効にするには 1 に設定します	bool
<code>`bindAsAuth`</code>	認証設定としてバインド	[*LDAP-BindAsAuth](#LDAPBindAsAuth)
<code>bindSearchAuth</code>	バインド+検索認証の構成	[*LDAPBind-SearchAuth](#LDAPBindSearchAuth)

4.66 LocalObjectReference

LocalObjectReference には、同じ名前空間内で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
<code>name</code>	指示対象の名前。 -必須	string

4.67 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。デフォルトのクエリをクラスターに注入したくない場合は、 <code>`true`</code> に設定します。デフォルト：false。	* bool
<code>customQueriesConfigMap</code>	カスタムクエリを含む構成マップのリスト	[[]ConfigMapKeySelector](#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](#SecretKeySelector)
<code>`enablePodMonitor`</code>	<code>`PodMonitor`</code> を有効または無効にします	bool

4.68 NodeMaintenanceWindow

NodeMaintenanceWindow には、基になるノードのアップグレード処理中に演算子を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り、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
<code>inProgress</code>	進行中のノードメンテナンスアクティビティはありますか？ - 必須	bool
<code>`reusePVC`</code>	既存の PVC を再使用する（ノードが再び起動するのを待つ）または使用しない（別の場所で再作成する - <i>instances</i> > 1 の場合） - 必須	* bool

4.69 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
<code>secrets</code>	NA	[]string

4.70 PgBouncerSecrets

PgBouncerSecrets には、pgbouncer によって使用されるシークレットのバージョンが含まれています

Name	Description	Type
<i>authQuery</i>	認証クエリーシークレットバージョン	[SecretVersion](#SecretVersion)

4.71 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>	認証クエリーに使用する必要があるユーザの資格情報。指定する場合は、AuthQuery (「SELECT username, passwd FROM pg_shadow WHERE username = \$ 1」) も指定する必要がある、自動 CNPG クラスター統合はトリガーされません。	[*LocalObjectReference](#LocalObjectReference)
<code>authQuery`</code>	特定のユーザ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ー。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る、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 に渡される追加パラメータ-設定可能なオプションのリストについては CNPG の文書を確認してください	map[string]string
<code>paused`</code>	<code>true`</code> に設定すると、PgBouncer は PostgreSQL サーバーから切断し、まずすべてのクエリが完了するまで待機し、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、演算子は PgBouncer の <code>PAUSE`</code> および <code>RESUME`</code> コマンドを呼び出します。	* bool

4.72 PodMeta

PodMeta は、metav1.ObjectMeta に似た構造ですが、まだユーザに適した CRD を作成するために controller-gen によって解析可能です。PodTemplateSpec のコメントには、コアデータ型を使用しない理由が説明されています。

Name	Description	Type
<code>labels`</code>	オブジェクトの整理と分類（スコープと選択）に使用できる文字列キーと値のマップ。レプリケーションコントローラーおよびサービスのセクターとマッチする場合があります。詳細： http://kubernetes.io/docs/user-guide/labels	map[string]string
<code>annotations`</code>	注釈は、任意のメタデータを保存および取得するために外部ツールで設定できるリソースとともに保存される非構造化キー値マップです。これらはクエリ可能ではなく、オブジェクトを変更ときに保存する必要があります。詳細： http://kubernetes.io/docs/user-guide/annotations	map[string]string

4.73 PodTemplateSpec

PodTemplateSpec は、ユーザがポッド生成用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	[PodMeta](#PodMeta)
<code>spec`</code>	ポッドの望ましい動作の仕様。詳細： https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status	corev1.PodSpec

4.74 プーラー

プーラーは、プーラー API のスキーマです

Name	Description	Type
<i>metadata</i>	NA	[metav1.ObjectMeta](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/#objectmeta-v1-meta)
<code>spec</code>	NA	[PoolerSpec](#PoolerSpec)
<code>status</code>	NA	[PoolerStatus](#PoolerStatus)

4.75 PoolerIntegrations

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

Name	Description	Type
<i>pgBouncerIntegration</i>	NA	[PgBouncerIntegrationStatus](#PgBouncerIntegrationStatus)

4.76 PoolerList

PoolerList には、Pooler のリストが含まれています

Name	Description	Type
<i>metadata</i>	NA	[metav1.ListMeta](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/#listmeta-v1-meta)
<code>items</code>	-必須	[[Pooler](#Pooler)]

4.77 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ます

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

4.78 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リファレンスです。プーラー名前は、同じ名前空間内のどのクラスター名前とも決してマッチしてはなりません。-必須	[LocalObjectReference](#LocalObjectReference)
<code>type`</code>	どのインスタンスにトラフィックを転送する必要がありますか？-必須	PoolerType
<code>instances</code>	必要なレプリカの数-必須	int32
<code>template`</code>	作成するポッドのテンプレート	[*PodTemplateSpec](#PodTemplateSpec)
<code>pgbouncer</code>	PgBouncer 設定-必須	[*PgBouncerSpec](#PgBouncerSpec)

4.79 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	[*Pooler-Secrets](#PoolerSecrets)
<code>instances</code>	スケジュールしようとしているポッドの数	int32

4.80 PostgresConfiguration

PostgresConfiguration は PostgreSQL の構成を定義します

Name	Description	Type
<code>`parameters`</code>	PostgreSQL 設定オプション (postgresql.conf)	map[string]string
<code>`pg_hba`</code>	PostgreSQL ホストベース認証ルール (pg_hba.conf ファイルに追加される行)	[]string
<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。デフォルトより大きいのは 40000000 で、1 年は秒単位で、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>	デフォルトのライブラリ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](#LDAPConfig)

4.81 RecoveryTarget

RecoveryTarget を使用すると、リカバリプロセスが停止する瞬間を構成できます。TargetTLI を除くすべてのターゲットオプションは相互に排他的です。

Name	Description	Type
<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>	ターゲットトランザクション ID	string
<code>`targetName`</code>	ターゲット名前 (<code>`pg_create_restore_point`</code> で以前に作成される)	string
<code>`targetLSN`</code>	ターゲット LSN (ログシーケンス番号)	string
<code>`targetTime`</code>	PostgreSQL で許可されている明確な表現形式でのターゲット時間	string
<code>targetImmediate</code>	一貫した状態に達するとすぐにリカバリを終了します	*bool
<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

4.82 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な場合、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、リカバリオブジェクトストアから、または pg_basebackup を介したストリーミングを介して作成できます。詳細については、文書の複製ページを参照してください。 -必須	bool
<code>source`</code>	レプリケーションのオリジン- 必須	string

4.83 RollingUpdateStatus

RollingUpdateStatus には、更新中のインスタンスに関する情報が含まれます

Name	Description	Type
<code>imageName</code>	ポッドに入れたイメージ- 必須	string
<code>startedAt</code>	更新が開始されたとき	[metav1.Time](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/#time-v1-meta)

4.84 S3Credentials

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。次の2つの方法で提供できます。

- 明示的に accessKeyId と secretAccessKey を渡す
- inheritFromIAMRole を true に設定して、ポッド環境からロールを継承する

Name	Description	Type
<code>`accessKeyId`</code>	アクセスキー ID へのリファレンス	[*SecretKeySelector](#SecretKeySelector)
<code>`secretAccessKey`</code>	シークレットアクセスキーへのリファレンス	[*SecretKeySelector](#SecretKeySelector)
<code>`sessionToken`</code>	セッションキーへの参照	[*SecretKeySelector](#SecretKeySelector)

次のページに続く

表 57 – 前のページからの続き

Name	Description	Type
<i>inheritFromIAMRole</i>	キーを明示的に提供せずに、ロールベースの認証を使用します。-必須	bool

4.85 ScheduledBackup

ScheduledBackup は、scheduledbackups API のスキーマです

Name	Description	Type
<i>metadata</i>	NA	[metav1.ObjectMeta](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/#objectmeta-v1-meta)
<code>spec</code>	ScheduledBackup の望ましい動作の仕様。詳細： https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status	[ScheduledBackupSpec](#ScheduledBackupSpec)
<code>status</code>	ScheduledBackup の最新の監視ステータス。このデータは最新ではない可能性があります。システムによって設定されます。読み取り専用。詳細： https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status	[ScheduledBackupStatus](#ScheduledBackupStatus)

4.86 ScheduledBackupList

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

Name	Description	Type
<i>metadata</i>	標準リストメタデータ。詳細： https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#types-kinds	[metav1.ListMeta](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/#listmeta-v1-meta)
<code>items</code>	クラスターのリスト-必須	[[]ScheduledBackup](#ScheduledBackup)

4.87 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>
<code>immediate`</code>	最初のバックアップを作成後すぐにスタート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 CronJobs で使用されてい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 を参照してください-必須	string
<code>cluster`</code>	バックアップするクラスター	[LocalObjectReference](#LocalObjectReference)
<code>backupOwnerReference</code>	作成されたバックアップリソース内にどの ownerReference を配置するかを示します。 -なし：作成されたバックアップオブジェクトの所有者リファレンスなし（フィールドが導入される前と同じ動作） -self：スケジュールされたバックアップオブジェクトを所有者として設定します。バックアップの -クラスター：クラスターをバックアップの所有者として設定 	string

4.88 ScheduledBackupStatus

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

Name	Description	Type
<code>lastCheckTime`</code>	スケジュールの最新時刻	[*metav1.Time](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/#time-v1-meta)
<code>lastScheduleTime</code>	バックアップが正常にスケジュールされた最後の時間である情報。	[*metav1.Time](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/#time-v1-meta)

次のページに続く

表 61 – 前のページからの続き

Name	Description	Type
<i>nextScheduleTime</i>	次回バックアップを実行します	[*metav1.Time](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/#time-v1-meta)

4.89 SecretKeySelector

SecretKeySelector には、秘密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

4.90 SecretVersion

SecretVersion にはシークレット名前とその ResourceVersion が含まれます

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

4.91 SecretsResourceVersion

SecretsResourceVersion は、演算子が管理するシークレットのリソースバージョンです

Name	Description	Type
<i>superuserSecretVersion</i>	「postgres」ユーザシークレットのリソースバージョン	string
<i>replicationSecretVersion</i>	「streaming_replica」ユーザシークレットのリソースバージョン	string
<i>applicationSecretVersion</i>	「アプリ」ユーザシークレットのリソースバージョン	string
<i>caSecretVersion</i>	未使用。古いバージョンとの互換性のために保持されます。	string
<i>clientCaSecretVersion</i>	PostgreSQL クライアント側 CA シークレットバージョンのリソースバージョン	string

次のページに続く

表 64 – 前のページからの続き

Name	Description	Type
<code>serverCaSecretVersion</code>	PostgreSQL サーバサイドの 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 サーバサイドの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>	Barman エンドポイント CA のリソースバージョン (提供されている場合)	string
<code>metrics</code>	メトリックス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

4.92 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に使用する StorageClass (<i>PG-DATA</i>)。使用可能な場合、PVC テンプレートを評価した後に適用されます。指定しない場合、生成された PVC はデフォルトのストレージクラスによって満たされます。	<i>*string</i>
<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です。このフィールドへの変更は、作成された PVC に自動的に再適用されます。サイズを小さくすることはできません。- 必須	string
<code>resizeInUseVolumes</code>	既存の PVC のサイズを変更、デフォルトは true	<i>*bool</i>
<code>pvcTemplate</code>	永続ボリューム要求の生成に使用されるテンプレート	[<i>*corev1.PersistentVolumeClaimSpec</i>](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/#persistentvolumeclaim-v1-core)

4.93 WalBackupConfiguration

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（バケットがそのためにまだ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、デフォルトを使用）、 <code>AES256</code> および <code>aws:kms</code> です。	EncryptionType
<code>maxParallel</code>	並行してアーカイブする（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ビングしている場合）か、並行して復元する（PostgreSQL スタンバイがリカバリオブジェクトストアから WAL ファイルをフェッチしている場合）WAL ファイルの数。指定しない場合、WAL ファイルは 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 が受け入れられる最小値です。	int

4.94 Supported releases

< ! -<https://raw.githubusercontent.com/istio/istio.io/master/コンテンツ/en/docs/releases/supported-releases/インデックス.md> wokeignore：ルール = master->に触発された

このページには、現在サポートされている CloudNativePG のリリースのステータス、タイムライン、ポリシーがリストされています。

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、セキュリティおよびバグ修正のために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ません。

- [サポートポリシー](#)
- [命名スキーム](#)
- [CloudNativePG リリースのサポート状況](#)

4.94.1 サポートポリシー

コミットごとに CloudNativePG の新しいビルドを作成します。

ほぼ2か月ごとに、マイナーリリースを構築し、リリースの認定だけでなく、いくつかの追加のテストを実行します。マイナーリリースで見つかった問題のパッチバージョンをリリースします。

さまざまなタイプのリリースは、CloudNativePG コミュニティからのさまざまな製品品質レベルと支援レベルを表しています。

このコンテキストでは、**サポート**は、コミュニティが重大な問題のパッチリリースを作成し、技術支援を提供することを意味します。

Type	Support Level	Quality and Recommended Use
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.

利用可能なリリースは [releases page](#) にあります。

Release notes で各マイナーおよびパッチリリースの高レベルの詳細情報を見つけることができます。

4.94.2 命名スキーム

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>
```

ここで、<minor> はリリースごとに増加し、<patch> は現在の<minor> リリースのパッチの数をカウントします。通常、パッチは<minor> リリースに関連する小さな変更です。

バージョンの Git タグには、**v** が先頭に追加されます。

4.94.3 CloudNativePG リリースのサポート状況

Version	Currently Supported	Release Date	End of Life	Supported Kubernetes Versions	Tested, but not su
main	No, development only				
1.15.0	Yes	April 21, 2022	October 1, 2022	1.21, 1.22, 1.23	1.19, 1.20

上記のリストでサポートされている Kubernetes バージョンのリストは、CloudNativePG のメンテナーがサポートおよびテストするのに合理的であると考えられるものによって異なります。

4.94.4 今後のリリース

Version	Release Date	End of Life	Supported Kubernetes Versions
1.16.0	June 23, 2022	~ November 23, 2022	1.21, 1.22, 1.23
1.17.0	September 1, 2022	~ February 1, 2023	1.21, 1.22, 1.23

注釈: 将来の日付は不確実であり、変更される可能性があります。

4.95 Release notes

CloudNativePG のユーザーに表示される変更の履歴。

4.95.1 バージョン 1.15.0

リリース日：2022 年 4 月 21 日

機能：

- **フェンシング**：クラスターまたは特定のフェンシング機能の紹介 cnpg.io/fencedInstances を介した PostgreSQL インスタンスのセット空でない場合、クラスター内のスイッチオーバー/フェイルオーバーを無効にする注釈。フェンスされたインスタンスはシャットダウンされ、ポッドは実行され続けます（検討中準備ができていない）検査と緊急事態
- **LDAP 認証**：LDAP Simple Bind および Search + Bind 構成を許可 Postgres クラスター仕様に定義される `pg_hba.conf` のオプション宣言的に、Kubernetes シークレットを機密情報としてオプションで使用できるようにします `ldapbindpasswd` などのオプション
- の値を受け入れる `primaryUpdateMethod` オプションの導入 `switchover` (デフォルト) および `restart`、監視なしの場合に使用 `primaryUpdateStrategy`; このメソッドは、プライマリに何が起るかを制御しますローリング更新プロシージャ中のインスタンス

- `kubectl cnp` プラグインの新しい `report` コマンドにより、診断が改善され、演算子と特定の Postgres の両方のより効果的なトラブルシューティングクラスター
- バックアップオブジェクトストアに存在しない `Backup` オブジェクトを整理する
- ポイントインタイムリカバリでのターゲットタイムラインと LSN の指定ブートストラップメソッド
- AWS S3 での `AWS_SESSION_TOKEN` 認証トークンのサポート `sessionToken` オプション
- Pooler ポッドの PgBouncer のデフォルトのイメージ名前は `quay.io/enterprisedb/pgbouncer:1.17.0`

修正：

- `targetTime` による Point-in-Time リカバリのベースバックアップ検出以前は、利用可能な最新のバックアップの前のターゲットが不可能（検出アルゴリズムは、最後の 出発点としてのバックアップ）
- に依存することにより、ホットスタンバイの影響を受けやすいパラメーターの回復力を向上演算子が `pg_controldata` から収集する値
- 無効なエントリを適切に破棄することにより、中間証明書のハンドリングが改善されました。無効な証明書エラーをスローする代わりに
- データベース内の Prometheus エクスポートメトリックコレクションクエリは、ロールバックではなくコミット（これにより、数値が変更される可能性があります下流のダッシュボードから可視ロールバックされたトランザクションの該当する場合）

4.95.2 バージョン 1.14.0

リリース日：2022 年 3 月 25 日

機能：

- 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します。Barman Cloud 2.19 で導入された機能の利点
- の序文によるバックアップの視認性の改善 `Cluster` オブジェクトの `LastBackupSucceeded` 条件
- ホットスタンバイの重要なパラメーターの更新のサポート：`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表示します演算子のオンラインアップグレードが進行中のとき
- 提供する代替方法として AWS IAM ロールを継承する機能 S3 オブジェクトストレージの資格情報
- プーラーの `authQuerySecret` および証明書の不透明なシークレットのサポート
- デフォルトの PostgreSQL バージョンを 14.2 に更新
- `maintenance` 名前付けの `kubectl cnp` プラグインに新しいコマンドを追加して設定します Kubernetes 全体の 1 つまたはすべてのネームスペースのクラスターに対するメンテナンスウィンドウクラスター

コンテナ画像：

- 最新の PostgreSQL コンテナには、Barman Cloud 2.19 が含まれます

セキュリティの強化：

- Operator を使用したネームスペース演算子のインストールに対する RBAC の強化 OpenShift を含む Lifecycle Manager。OpenShift ユーザーの推奨事項このバージョンにアップデートしてください。

修正：

- インスタンスマネージャが、中断された `pg_rewind` を保持してリトライできるようにします。オリジナルの `pg_control` ファイルのコピー
- `pg_rewind` を実行する前に古い PID ファイルをクリーンアップします
- `primary_conninfo` のキーによる並べ替えを強制して、ランダム再起動を回避します 13 より前の PostgreSQL バージョン
- `ServiceAccount` の変更（ラベル、注釈など）を保持します 和解
- `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` バリデーションを改善
- LSN を使用してクラスターを回復する際に `targetLSN` オプションを適切にハンドルの指定された
- 証明書チェーンの両方を許可することにより、カスタム TLS 証明書のバリデーションを修正サーバーおよび CA 証明書

4.95.3 バージョン 1.13.0

リリース日：2022 年 2 月 17 日

機能：

- Snappy 圧縮のサポート。Snappy は、増加するバックアップの高速圧縮オプションですより低い圧縮率を使用したオブジェクトストアへのアップロードの速度 Barman オブジェクトストアにアップロードされたタグファイルのサポート。この機能にはオペランドイメージの Barman2.18。クラスタ削除後のバックアップの
- `status.conditions` を使用したクラスターのステータスの拡張。条件 `ContinuousArchiving` クラスターが WAL ファイルのアーカイブを開始したことを示します
- `kubectl` の `cnpg` プラグインの `status` コマンドを追加情報で改善：クラスターと `Certificates Status` のステータスを示す `Cluster Summary` セクションを追加しますクラスターで使用される証明書のステータスを含むセクションエキスパイアまで
- 空では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新しいクラスターを作成するとき
- `Secret` を使用してデフォルトのモニタリングクエリを追加するためのサポートを追加 `MONITORING_QUERIES_SECRET` 構成変数。
- ユーザが 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（デプロイされた Kubernetes クラスター上）AppArmor サポート付き）

- `kubectl` の `cnpg` プラグインに Windows プラットフォームサポートを追加しました。現在、プラグインが利用可能です Windows x86 および ARM
- Kubernetes 1.18 および非推奨の API バージョンのサポートを終了

コンテナ画像：

PostgreSQL コンテナには Barman 2.18 が含まれます

セキュリティ修正：

- 所有者およびスーパーユーザ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。以前は、悪意のあるユーザがシークレットを使用してパスワードを変更できた可能性があります PostgreSQL ユーザの

修正：

Postgres ポッドからステータスを取得するコードのメモリーリークを修正 - クラッシュ後の PostgreSQL の自己再起動を無効にします。インスタンスコントローラが処理する PostgreSQL インスタンスのライフサイクル - `spec.postgresUID` および `spec.postgresGID` フィールドの変更を防止バリデーション webhook で。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 をスタートできなくなります - バックアップおよび WAL アーカイビングハンドリングコードからのログの冗長性を削減 - 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 稼働レートのクラスター内のスタンバイサーバをストリーミングにスイッチことができます。それらが整列したら - まれにクラッシュにつながる可能性のある PostgreSQL ポッドのスタートアップ中の競合条件を修正 - 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条件を修正 - クラスターをレプリカとして参加させる前に、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 - 復旧時にレプリケーションに敏感な PostgreSQL 構成パラメーターを正しくハンドルするバックアップから - クラスター作成中の PostgreSQL 構成のバリデーションの欠落を修正

4.95.4 バージョン 1.12.0

リリース日：2022 年 1 月 11 日

機能：

- 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、1.17 のエンドツーエンドテストを削除します。2020 年 12 月に Kubernetes プロジェクトによるサポートを終了しました
- ネットワークに問題がある場合のポッドステータスチェックの応答性を改善する 2 秒の接続タイムアウトと通信タイムアウトを追加することにより 30 秒。この変更により、演算子が待機する時間に制限が設定されます失敗として宣言する前にステータスをレポートするポッドフェイルオーバーオペレーションの堅牢性と予測可能性
- ユーザに許可するクラスターに `.spec.inheritedMetadata` フィールドを導入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するクラスターによって
- ステータスの計算時に実行されるクエリの数減らす インスタンスの
- PgBouncer の準備プロンプトを追加します

- バーマンのエンドポイントのカスタム認証局のサポートを追加 Azure プロトコルを使用する場合のバックアップオブジェクトストア

修正：

- フェイルオーバー中、すべての WAL ストリーミングが完了するまで、新しいプライマリが選択されるのを待ちます接続が閉じられます。演算子はデフォルトで `wal_sender_timeout` を設定するようになりました および `wal_receiver_timeout` を 5 秒に `make`、スタンバイノードがプライマリにネットワークの問題がある場合はすぐに気づく
- レプリカクラスターの WAL アーカイビング戦略を変更してローリング更新を修正の PostgreSQL インスタンスに対して「`archive_mode`」を「`always`」に設定することによりレプリカクラスター。次に、WAL のアップロードを制限します現在のターゲットと指定されたプライマリ。WAL は 2 回アップロードされる場合があります切り替え中、これは問題ではありません
- バーマンのエンドポイントのカスタム認証局のサポートを修正レプリカクラスターソースのバックアップオブジェクトストア
- 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前を使用する
- デフォルトの `webhook` が何らかの理由で機能しない場合、演算子は今調整サイクル中にもデフォルトでクラスターを更新します
- リソースリクエストと制限の比較を修正して、まれな問題を修正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
- `webhook` からのログメッセージを改善して、オブジェクトのネームスペースも含める
- すべての調整ループのスタートに「デフォルト」メッセージのロギングを停止します
- 調整サイクルごとに `PodMonitor` 削除のロギングを停止します `enablePodMonitor` が `false` の場合
- ポッドがそうでない場合、アーキテクチャの不一致の可能性について文句を言わないでください到達可能

4.95.5 バージョン 1.11.0

リリース日：2021 年 12 月 15 日

機能：

- **パラレル WAL のアーカイビングと復元**：データベースが WAL に対応できるようにするを導入することにより、高書き込みシステムでの生成並列の最大数を設定する `backupObjectStore.maxParallel` オプション両方の WAL アーカイビング中に実行されるジョブ (PostgreSQL の `archive_command`) および WAL 復元 (`restore_command` による)。パラレルを使用する復元オプションにより、新しく昇格されたスタンバイをより早く準備完了状態にすることができます必要な WAL ファイルをフェッチして、順次ではなく並行してリプレイする
- **モニタリングのためのメトリックのデフォルトセット**：新しい `ConfigMap` と呼ばれる `default-monitoring` は、同じ名前空間に自動的にデプロイされますデフォルトでは、既存の PostgreSQL クラスターに追加され演算子。そのような振る舞い `MONITORING_QUERIES_CONFIGMAP` パラメータを設定してグローバルに変更できますオペレーターの構成、またはクラスターレベルで `spec.monitoring.disableDefaultQueries` オプション (デフォルトでは `false` に設定)

- のモニタリングセクションに `enablePodMonitor` オプションを導入 `PodMonitor` リソースを自動的かつシームレスに管理するクラスタープロメテウスと統合する
- スマートを実行しようとすることにより、PostgreSQL のシャットダウンプロシーダを改善します。目的の `stopDelay` 時間の前半のシャットダウン、および高速 Kubernetes によってポッドが殺される前の残りの半分のシャットダウン
- `switchoverDelay` オプションを追加して、前者に与えられる時間を制御しますプライマリを正常にシャットダウンし、すべての WAL ファイルをアーカイブする前に新しいプライマリをプロモートします（デフォルトでは、CloudNativePG は待機します権限データの永続性まで無期限に） PostgreSQL Cluster のリソース要求と制限の変更を処理する ローリング更新の発行
- `kubectl` の `cnpg` プラグインの `status` コマンドを改善追加情報：ストリーミングレプリケーションステータス、合計サイズデータベース、クラスター内のインスタンスのロール
- 有効にすることにより、多くの並列ワーカーでワークロードのサポートを強化 `dynamic_shared_memory_type` および `shared_memory_type` の構成 PostgreSQL の共有メモリ管理のパラメーター
- クラスターレベルで定義されたラベルと注釈を関連リソース（ポッドを含む）（削除はサポートされていません）
- Kubelet によって削除されたポッドを自動的に削除します
- Azure の永続ボリュームの自動サイズ変更を管理する `ENABLE_AZURE_PVC_UPDATES` 演算子構成オプション。必要に応じてクラスターのローリング更新（デフォルトでは無効）
- `cnpg.io/reconciliationLoop` アノテーションを導入します。特定の Postgres クラスターで `disabled` に設定し、調整を防止します実行中のループ
- `initdb` オプションのパートとして `postInitApplicationSQL` オプションを導入メインで実行される SQL クエリのリストを指定するブートストラップメソッドクラスターが実行された直後のスーパーユーザとしてのアプリケーションデータベース作成された

修正：

- Liveness プローブが PostgreSQL のスタートアッププロセスを正しく処理するようになりましたサーバー。これにより、少数のお客様から報告された問題が修正され、一貫性を保つために WAL ファイルを回復するニーズがある再起動されたスタンバイサーバ状態ですが、活性プロブのタイムアウトになる前にそれを行うことができませんでした起動して、ポッドを `CrashLoopBackOff` ステータスのままにします。
- Liveness プローブは、ニーズ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 `pg_rewind` を使用して、タイムライン後に現在のプライマリと再調整する転用。これにより、新しいスタンバイのポッドが繰り返し殺されるのを修正します Kubernetes による。Postgres ポッドからのクライアント側の調整を削減します（例：「優先度と公平性ではなく、クライアント側の調整による 1.182388649 リクエスト：GET」）
- OpenShift および OLM で公開キーインフラストラクチャ（PKI）の初期化を無効にする提供されたものを使用したインストール
- リスタートが必要な構成パラメータを変更するときは、常に最後のプライマリ

- 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ます。PVC 初期化での競合条件の防止
- 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 のパース時のオーバーフローを修正
- スタートアップに古い PID ファイルを削除する
- Pooler リソースに、定義されている `imagePullSecret` を継承させます。演算子、存在する場合

4.95.6 バージョン 1.10.0

リリース日：2021 年 11 月 11 日

機能：

- **PgBouncer による接続プーリング**：Pooler リソースを導入し、として使用される PgBouncer 展開を自動的に管理するコントローラローカル PostgreSQL Cluster のコネクションプーラ。この機能には TLS が含まれますクライアント/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、キー PgBouncer パラメータの設定、PAUSE / RESUME、JSON フォーマットのロギング、統計、プール、リストの Prometheus エクスポーター
- **バックアップ保存ポリシー**：リカバリウィンドウ保存の定義をサポートバックアップのポリシー（たとえば、30 日間のリカバリ期間を保証の「30d」）
- **演算子のインプレース更新**：のインプレースオンライン更新を導入インスタンスマネージャ。これは、のローリング更新を実行する必要を取り除きます。演算子の更新後のクラスター全体。デフォルトでは、このオプションは無効（を参照してください [documentation for more detailed information](#)）
- **initdb ブートストラップでカスタマイズできるオプションのリストを制限する** `dataChecksums`、`encoding`、`localeCollate`、`localeCTYPE` のメソッド、`walSegmentSize`。これにより、`options` 配列は廃止され、予定されています。v2 API で削除されました
- **initdb ブートストラップのパートとして postInitTemplateSQL オプションを導入** `template1` で実行される SQL クエリのリストを指定するメソッドクラスタが作成された直後のスーパーユーザとしてのデータベース。これこの機能により、すべてのアプリケーションデータベースにデフォルトオブジェクトを含めることができます。クラスターで作成
- **インスタンス Prometheus エクスポーターに追加された新しいデフォルトメトリック**：Postgres バージョン、クラスター名前、およびリカバリ可能性の最初のポイントバックアップカタログ
- **失敗後にバックアップの取得を再試行します** Barman Cloud の機能に関する認識を構築して、最近導入された機能（保持など）の呼び出しからの演算子オペランドに存在しないポリシー、または Azure Blob Container ストレージ）頻繁に更新されない画像
- **cnpg プラグインの status コマンドの出力と情報を統合します**バックアップについて
- **PVC のステータスを報告する新しい注釈を導入します**（初期化済みまたは準備完了）
- **cnpg.io/cluster label にクラスター名前を設定します** Backup オブジェクトを含む Backup オブジェクトで生成されたオブジェクト

- 非推奨の API バージョンのサポートを終了 Cluster、Backup の `postgresql.cnpg.io/v1alpha1`、および `ScheduledBackup` 種類
- 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:

- 演算子コンテナの `allowPrivilegeEscalation` を `false` に設定します `securityContext`

修正:

- 単一インスタンスのメンテナンス中にプライマリ `PodDisruptionBudget` を無効にしますクラスター
- リカバリ中に正しい証明書証明権限 (CA) を使用する 操作
- WAL アーカイビング状態を確認する前に PostgreSQL 接続がリークするのを防ぐバックアップを取る
- 一時的なエラーが発生した後、WAL アーカイブ/リストアを 100 ミリ秒間スリープさせます。それ以外の場合はログをフラッシュする

4.95.7 バージョン 1.9.2

リリース日: 2021 年 10 月 15 日

機能:

- 2 つの新しいロガーで JSON ログを強化します: PostgreSQL の `wal-archive` スタンバイの `archive_command`、および `restore_command` の `wal-restore`

修正:

- スタンバイプロモーション中に WAL アーカイビングを有効にします (`.history` ファイルを防止します) アーカイブされることから
- 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 古いオペランドのエラーを回避するための Barman 2.13 以降
- プライマリのポッドが準備できるのを待ってから、バックアップをトリガーします

4.95.8 バージョン 1.9.1

リリース日: 2021 年 9 月 30 日

このリリースは、*PostgreSQL 14* <<https://www.postgresql.org/about/news/postgresql-14-released-2318/>> `\_\_` の発売を祝うためのものです。新しい ``Cluster`` を作成せずにデフォルトのメジャーバージョンにする特定のイメージ名前を定義します。

修正:

- `Error while getting barman endpoint CA secret` メッセージの原因となる問題を修正プライマリポッドのログに表示され、バックアップが機能しなかった 正しく
- 一時的な通信の場合、新しいバックアップの要求を適切にリトライしますインスタスマネージャの問題

4.95.9 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、*OpenShift* では使用できないため、*OpenShift* では使用できませんバージョン 1.9.1。のリリースプロセスとその後のリリース

機能：

- 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 を追加します。
削除 1.16
- `pg_rewind` の `--restore-target-wal` オプションのサポートを導入必要に応じて、バックアップアーカイブから WAL ファイルをフェッチオーダー（利用可能 PostgreSQL 13 以降のみ）
- Prometheus エクスポートのデフォルトメトリックを公開して、数を推定します各データベースの `pg_catalog.pg_largeobject` テーブル内のページの
- ローカルインメモリを介して、WAL のアーカイビングとフェッチのパフォーマンスを強化します。
キャッシュ

修正：

- `pg_isready` を呼び出すときに `postgres` ユーザを明示的に設定します。 OpenShift の制限された SCC
- ステータスの `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

4.95.10 バージョン 1.8.0

リリース日：2021 年 9 月 13 日

機能：

- 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直接ブートストラップします
外部クラスタセクションで定義されたオブジェクトストアバックアップ CR を定義するための以前の要件
- 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、バックアップを要求します
最初の Postgres インスタンスの実行直後、機能を追加 Point-in-Time Recovery が有効な場合にクラスターの最初に巻き戻す 設定済み
- クラスターステータスに `firstRecoverabilityPoint` を追加して、最も古いものをレポートします
バックアップオブジェクトに基づいてリカバリを要求する一貫した時点店のコンテンツ
- 公開することにより、PostgreSQL インスタンスのデフォルトの Prometheus エクスポートを強化次の新しいメトリック：

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レプリカおよび同期的ファイルの数 3. レプリカモードのフラグもの

- でカスタムメトリックを定義するときに `runonserver` オプションのサポートを追加メトリックの収集を次のレンジに制限するプロメテウスエクスポート PostgreSQL バージョン
- バックアップとリカバリのために Azure Blob Storage をネイティブにサポートします。Barman Cloud 用に Barman 2.13 で導入された機能の利点
- 活性プローブを `pg_isready` に依存
- リカバリターゲット時間のタイムスタンプ仕様の RFC3339 フォーマットをサポート
- `.spec.imagePullPolicy` を導入して、イメージのプルポリシーを制御しますクラスター用に作成されたすべてのポッドとジョブのコンテナ
- OpenShift 4.5 に代わる OpenShift 4.8 のサポートを追加 PostgreSQL 14 (ベータ) のサポート
- 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外部クラスタセクションで定義されたオブジェクトストア。ストリーミング接続 (実験的)
- `logLevel` オプションをクラスターの仕様に導入して、次のレベル: エラー、情報、デバッグまたはトレース

セキュリティの強化:

- `.spec.enableSuperuserAccess` を導入して、ネットワークアクセスを有効/無効にしますパスワード認証 postgres ユーザ

修正:

- クラスターが回復不能な状態になったときにユーザーに適切に通知し、人間の介入が必要

4.95.11 バージョン 1.7.1

リリース日: 2021 年 8 月 11 日

機能:

- 同期的に関しては、構成よりも自己修復を優先するレプリケーション、演算子が一時的にオーバーライドできるようにしますクラスターがそうでない場合の `minSyncReplicas` および `maxSyncReplicas` 設定 自己修復操作中に要件を満たすことができる
- `initdb` ブートストラップメソッドのパートとして `postInitSQL` オプションを導入スーパーユーザとしてすぐに実行される SQL クエリのリストを指定するにはクラスターが作成された後

修正:

- プライマリの準備ができていないときに演算子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 同期的コミットレベルを使用して管理クエリを実行する
- PGAudit で複数行のログエントリを正しく解析

4.95.12 バージョン 1.7.0

リリース日： 2021 年 7 月 28 日

機能：

- `pgaudit` と呼ばれる `logger` の新しいタイプで `PGAudit` にネイティブサポートを追加 JSON 出力で直接利用可能
- 以下を介して、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` 拡張のネイティブサポート-Prometheus エクスポートの `target_databases` オプションは、1 つ以上のデータベース（シェルのようなパターンマッチングによるデータベースの自動検出を含む）でユーザー定義のメトリッククエリーを実行します-`manual_switchover_required` メトリックの公開 `primaryUpdateStrategy` が `supervised` に設定されているクラスターにマニュアルスイッチオーバーが必要かどうかを迅速にレポートする
- `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` label をサポートして、カスタムに使用されるものを含む、構成マップとシークレットの自動リロード Prometheus エクスポートでのメトリックのモニタリング/警告、または証明書の保存
- `reload` コマンドを `kubectl` の `cnpg` プラグインに追加して、インスタンスの調整ループ
- ポッドアフィニティおよび非アフィニティ設定の制御を改善します `additionalPodAffinity` および `additionalPodAntiAffinity`
- プライマリインスタンス用に別の `PodDisruptionBudget` を導入します。少なくとも 1 プライマリインスタンスをいつでも実行する必要がある

セキュリティの強化：

- `.spec.certificates.clientCASecret` を追加し、カスタムクライアントを定義する `spec.certificates.replicationTLSSecret` オプション使用する PostgreSQL サーバーの証明機関と証明書クライアント証明書を認証し、PostgreSQL 間の通信を保護する ノード
- `.spec.backup.barmanObjectStore.endpointCA` オプションを追加して、Barman のバックアップのエンドポイントのカスタム証明機関バンドルオブジェクトストア

修正：

- Prometheus エクスポートでヒストグラムを正しく解析
- 演算子がクラスター用に作成したサービスの調整

4.95.13 バージョン 1.6.0

リリース日：2021 年 7 月 12 日

機能：

- レプリカモード（**EXPERIMENTAL**）：クラスターをレプリカとして作成できるようにしますソースクラスターの。レプリカクラスターには、**指定プライマリ**およびスタンバイの数。
- `.spec.postgresql.promotionTimeout` パラメータを追加して、インスタンスをプライマリに昇格させるときに待機する秒数。デフォルトは 40000000 秒です。
- `.spec.affinity.podAntiAffinityType` パラメータを追加します。に設定できます `preferred`（デフォルト）、結果は 使用されている `preferredDuringSchedulingIgnoredDuringExecution` または `required`、`requiredDuringSchedulingIgnoredDuringExecution` になります。

変更点：

- PVC とポッドを削除する際の競合条件を修正しました。演算子が新しいポッドを作成するのを止めます。
- マネージャが必要性を検出できない競合条件を修正構成の変更時に PostgreSQL がリスタートします。
- 注釈のないクラスターでの `kubectl-cnp` のパニックを修正しました。
- 一部のログメッセージのレベルを `debug` に下げました。
- サーバー CA および TLS インジェクションの E2E テスト。

4.95.14 バージョン 1.5.1

リリース日：2021 年 6 月 17 日

変更：

- Red Hat OpenShift でのオペレーターデプロイメントでの自動更新を妨げる CRD バリデーションのバグを修正
- シークレットを使用してオペレーターの構成を渡すことを許可します。

4.95.15 バージョン 1.5.0

リリース日：2021 年 6 月 11 日

機能：

- `pg_basebackup` ブートストラップメソッドを導入して、新しい PostgreSQL を作成します同じメジャーの既存の PostgreSQL インスタンスのコピーとしてのクラスターバージョン、Kubernetes 外でも
- の `Affinity` セクションに Kubernetes の許容のサポートを追加ユーザーが PostgreSQL インスタンスを配布できるようにする Cluster リソース 必要な汚染のある Kubernetes ノード

- を介して、イメージ名前へのダイジェストの指定を有効にします
`<image>:<tag>@sha256:<digestValue>` フォーマット、より決定的かつ反復可能な展開

セキュリティの強化：

- TLS 証明書をカスタマイズして、PostgreSQL サーバーを認証します。サーバー証明書と関連する認証局の秘密 署名した
- 内部の WAL レシーバプロセスの `sslmode` を上げ、`require` から `verify-ca` への自動管理ストリーミングレプリカ

変更点：

- `kubectl` が受け入れるように `cnpg` プラグインの `promote` サブコマンドを強化 ポッドの名前全体ではなく、ノード番号のみ
- クラスター名に DNS-1035 バリデーションスキームを採用（そこからサービス名継承されます）
- スタンバイインスタンスのクローン作成時にストリーミングレプリケーション接続を強制する、または `pg_basebackup` メソッドを使用してブートストラップする場合
- Backup リソースを `beginWal`、`endWal`、`beginLSN` と統合し、物理的ベースバックアップに関する `endLSN`、`startedAt`、および `stoppedAt`
- ドキュメントの改善：
- 演算子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
- `cnpg-bench` ヘルムチャートとベンチマークのガイドラインを紹介します。データベースワークロード用のストレージと PostgreSQL
- E2E テストの機能強化：
- Kubernetes 1.21 をテストする
- 演算子の高可用性のテストを追加します
- 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` スタートのタイムアウトが短すぎます
- オペレーターは、PVC の直接イベントを監視していません
- `immediateCheckpoint` および `jobs` パラメータのハンドリングを修正 `barmanObjectStore` バックアップ
- バックアップから回復するときに空のログ

4.95.16 バージョン 1.4.0

リリース日： 2021 年 5 月 18 日

機能：

JSON フォーマットの PostgreSQL エラーメッセージの標準出力ロギング - Prometheus エクスポーターに PostgreSQL メトリックの基本セットを提供します - `kubectl` 用の `cnpg` プラグインに `restart` コマンドを追加してリスタートしますロールアウト方式での特定の PostgreSQL クラスターのポッド

セキュリティの強化：

- ポッドの `readOnlyRootFilesystem` セキュリティコンテキストを設定する

変更点：

- **重要：** CloudNativePG を以前にデプロイしたことがある場合 YAML マニフェストを使用する演算子、既存の演算子を削除する必要があります新しいバージョンをインストールする前の展開。これは避けるために必要ですラベルの変更による他の Kubernetes API との競合そして演算子によって直接管理されている label セレクター。参照してくださいアップグレード処理の詳細については、CloudNativePG の文書 1.4.0 へ
- 演算子によって自動的に定義されるラベルを修正し、名前を変更します `control-plane: controller-manager` から `app.kubernetes.io/name: cloudnative-pg`
- `metrics` 名前を Prometheus エクスポーターの TCP ポートに割り当てます
- `cnpg_metrics_exporter` をメトリックエクスポーターの `application_name` として設定します PostgreSQL での接続
- 可能な場合は、モニタリングデータベースを使用して、`postgres` データベースではなく Prometheus エクスポーター
- 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
- 以下を含むマイナーなバグ修正：
 - `pg_basebackup` の呼び出し時に `-R` の使用を避ける
 - ステータスの取得時にエラーログからスタックトレースを削除する

4.95.17 バージョン 1.3.0

リリース日： 2021 年 4 月 23 日

機能：

- ラベルと注釈の継承
- すべてのコンテナにリソース制限を設定する

セキュリティの強化：

- 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ポッドの実行を、UID および SELinux コンテキストが割り当てられたネームスペースに制限します。
- 実行演算子とし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非ルート、非特権、権限エスカレーションなし

変更点：

- ポート 9187 でリッスンしている Prometheus エクスポートエンドポイント（ポート 8000 は現在 API サーバーとのインスタンス調整のために予約）
- ドキュメントの改善
- GKE 環境を含む E2E テストの機能強化
- マイナーなバグ修正

4.95.18 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、バックアップは Scheduled-Backup オブジェクトが削除されても削除されません
- RHSA-2021:1024 を解決するための ubi8-minimal イメージの更新（セキュリティアドバイザリ：重要）

4.95.19 バージョン 1.2.0

リリース日： 2021 年 3 月 31 日

- ConfigMap としてカスタムモニタリングクエリの実験的支持を導入し、Prometheus の `postgres_exporter` と互換性のある構文を使用する秘密オブジェクト
- Operator Lifecycle Manager（OLM）の展開をサポートします。OperatorHub.io 上の存在
- 米国国防総省のガイドラインを適用して、コンテナのセキュリティを強化防衛（DoD）の防衛情報システム局（DISA）およびセンターインターネットセキュリティ（CIS）およびパイプラインで直接検証するドックル
- AKS での E2E テストの改善

- マイナーなバグ修正

4.95.20 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、クラスターのステータスをきれいに出力します。JSON および YAML 出力
- `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、利用可能なホットに接続をルーティングしますスタンバイレプリカのみ。読み取り専用クエリのオフロードを有効にします。クラスターのプライマリインスタンス
- クラスターを `maxSyncReplicas` よりも低い値に縮小するロールバック
- 元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

4.95.21 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（stable）リリースは、API グループ `postgresql.cnpg.io/v1` の `Cluster`、`Backup` および `ScheduledBackup` を実装します。これらのリソースを使用して、以下の主な機能を持つ Kubernetes 内で PostgreSQL クラスターを作成および管理します。

- 高可用性のための Kubernetes API サーバーとの直接統合、なし外部ツールが必要
- 以下を介した自己修復機能：
 - 最も整合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フェイルオーバー
 - レプリカの自動作成
 - 選択したレプリカをプロモートすることにより、プライマリインスタンスの計画的な切り替え
 - スケールアップ/スケールダウン機能
 - 任意の数のインスタンスの定義（最小 1-1 つのプライマリーサーバ）
 - アプリケーションを接続する* 読み書き*サービスの定義クラスターのプライマリーサーバのみ
 - アプリケーションを次のいずれかに接続する* read *サービスの定義ワークロードを読み取るためのインスタンス
 - 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 マイナーバージョンのローリング更新と演算子アップグレード

- 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 互換のオブジェクトストアバックアップからの完全リカバリとポイントインタイムリカバリ
- 同期的レプリカのサポート
- `nodeSelector` プロパティを介したノードアフィニティのサポート PostgreSQL エラーメッセージの標準出力ロギング

4.96 Frequently Asked Questions (FAQ)

4.96.1 Kubernetes で PostgreSQL を実行する

PostgreSQL のようなステートフルワークロードは `inKubernetes` で実行できないことは誰もが知っています。なぜあなたは反対を言うのですか？

2021 年 9 月の [independent research survey commissioned by the Data on KubernetesCommunity](#) は、回答者の半数が Kubernetes で本番ワークロードのほとんどを実行していることを明らかにしました。90 %が Kubernetes がステートフルワークロードに対応していると考えており、70 %が Postgres のような稼動データベースでデータベースを実行しています。しかし、彼らによると、知識のギャップ（一般に Kubernetes と CloudNative は急勾配の学習曲線を持っています）や Kubernetes 演算子の品質など、重要な課題が残っています。後者は、CloudNativePG のような `anoperator` がプロジェクトの成功に大きく貢献すると信じる理由です。

私たちのようなデータベースの狂信者にとって、本当のゲームチェンジャーは [*Kubernetes 1.14 in April 2019*](#) 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。それはどういう意味ですか？

マイクロサービスのアーキテクチャパターン、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트리ポイントとして実行するように構築されます（いわゆる PID 1 プロセス）。

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なステートレスまたはデータベースのようなステートフルです。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、単一の特定のバージョン（`immutablecontainer` イメージにあるもの）に関連付けられて実行される単一の “`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。

不変の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非常に一般的な方法である `Mutable SystemContainer` 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：`noupdates`、`no patch`、`no configuration changes`。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。不変性は、展開をより安全で再現性の高いものにします。

詳細については、[*"Why EDB chose immutable application containers"*](#) を参照してください。

CloudNative の意味

Cloud Native Computing Foundation は **Cloud Native** という用語を定義しています。ただし、2ndQuadrant で Cloud Native PostgreSQL / CloudNativePG オペレーターがスタートされて以来、開発チームは Cloud Nativeas の 3 つの主要な概念を解釈しています。

1. 人々に根ざした既存の健全で本物の繁栄した DevOps 文化、およびチームと組織（チームのチームとして）が継続的に変化し、成果と成果の提供を革新および加速できるようにする原則とプロセスより安全で、より効率的で、より魅力的な方法でのビジネスの価値 2. Immutable Application Containers³ に基づくマイクロサービスアーキテクチャ。Kubernetes など、これらのコンテナを管理および編成する方法

現在、コンテナオーケストレーションの標準デファクトは、クラウドネイティブアプリケーションの展開、管理、およびスケーラビリティを自動化する Kubernetes です。

私たちに共鳴する Cloud Native の別の定義は、**"Kubernetes Patterns", published by O'Reilly** で Ibryam と Huß によって定義されたものです。

大規模なコンテナ化されたマイクロサービスを自動化するための原則、パターン、ツール

<！ - ベアメタル Kubernetes で CloudNativePG を実行できますか？

TODO

ファイルシステムレプリケーションの代わりに PostgreSQL レプリケーションを使用する必要があるのはなぜですか？

TODO>

コンテナとして PostgreSQL を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 を実行する最も基本的な方法は、Apod を使用することです。これは Kubernetes での展開の最小単位であり、レプリカなしで Postgres コンテナを実行します。Postgres データディレクトリをホスティングするボリュームはポッドにマウントされ、通常はネットワークストレージに存在します。この場合、Kubernetes は問題が発生した場合にポッドを再起動するか、別の Kubernetes ノードに移動します。

最も洗練されたアプローチは、演算子を使用して PostgreSQL を実行することです。演算子は Kubernetes コントローラの拡張機能であり、ビジネス継続性コンテキストでの複雑なアプリケーションの動作を定義します。Theoperator パターンは現在、この目的のために Kubernetes の最先端技術です。演算子は、自動化されたプログラマティックな方法で人間の演算子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、演算子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後に適切に対応するニーズもあります。典型的な例は、フェイルオーバーの例です。

演算子は、自己修復、スケーラビリティ、レプリケーション、高可用性、バックアップ、リカバリ、更新、アクセス、リソース制御、ストレージ管理などの機能を Kubernetes に依存しています。また、ログ管理およびモニタリングインフラストラクチャでの PostgreSQL クラスターの統合を促進します。

CloudNativePG を使用すると、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 コントローラによって開始された調整ループを通じて、インフラストラクチャの現在の状態が目的の状態と一致することを継続的に確認します。目的の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 のような演算子が登場します。

CloudNativePG は完全に宣言的な演算子であると言いますが、それはどういう意味ですか？

最も簡単な方法は、命令型構成との違いを強調する例を通して宣言型の構成を説明することです。命令型のコンテキストでは、状態はシーケンスに実行されるシリーズのタスクとして定義されます。したがって、

最初のインスタンスを作成し、レプリケーションを構成し、2 番目のインスタンスを複製し、3 番目のインスタンスを作成することで、3 ノードの PostgreSQL クラスターを取得できます。

宣言的なアプローチでは、構成を使用してシステムの状態を定義します。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。このアプローチは、変更管理操作を非常に単純化し、これらを Git などのソース管理システムに保存すると、Infrastructure as Code 機能を有効にします。また、Kubernetes は、要求よりも確実に要求を確実に満たすため、展開よりもはるかに重要です。

Kubernetes で PostgreSQL を実行するために必要なスキルはどれですか？

Kubernetes で PostgreSQL を実行するには、DevOps チームで PostgreSQL と Kubernetes skills の両方が必要です。最高の体験は、データベース管理者が Kubernetes のコア概念に精通し、Kubernetes 管理者とやり取りできるようになることです。

私たちのアドバイスは、Cloud Native PostgreSQL を完全に活用して CNCF 認定プログラムから「Certified Kubernetes Administrator (CKA)」ステータスを取得したいすべての人に対するものです。

< ! -

4.96.2 高可用性

Patroni、repmgr、Stolon などのフェールオーバー管理ツールに依存しない CloudNativePG の背後にある理由は何ですか？

TODO

フェイルオーバー（計画外）およびスイッチオーバー（計画）時間が年間 99.995 % の SLA 内であることを保証にはどうすればよいですか？

TODO

フェールオーバー後、以前のプライマリを新しいプライマリと手動で再同期する必要がありますか？

TODO

PostgreSQL のすべてのノードに障害が発生するとどうなりますか？

TODO

4.96.3 バックアップと復元

CloudNativePG は pg_dump を使用したロジカルバックアップをサポートしていますか？

TODO

災害復旧に関して最良の結果を得るための推奨設定は何ですか？

TODO

PostgreSQL を実行していた Kubernetes クラスターが永久になくなった場合はどうなりますか？

TODO

4.96.4 その他

CloudNativePG がサポートする Kubernetes ディストリビューションとは何ですか？この決定の根拠は何ですか？

TODO

ラージ環境（>4TB/256 ギガバイト/64 コアなど）のパフォーマンステストまたは値はありますか？

TODO

LDAP/ActiveDirectory サポートに関する質問

TODO

CloudNativePG データベースのプロビジョニングはどのように自動化できますか？

TODO

CloudNativePG のプロビジョニング解除はどのように自動化できますか？

TODO

CloudNativePG は、クラスターが削除されたときに PostgreSQL のデータディレクトリを消去しますか？

TODO

ランタイム中にデータベースインスタンスに対してリソース（ストレージ/CPU/メモリ）の変更はどのように行われますか？

TODO

PostgreSQL クラスターが正常に作成された後、接続に関する詳細をメールで通知するにはどうすればよいですか？

TODO

4.96.5 アップデート

基盤となる Kubernetes ノードで更新を実行するにはどうすればよいですか？

TODO

PostgreSQL のマイナーバージョン更新をダウンタイムなしで実行できますか？

TODO

PostgreSQL のメジャーバージョンアップグレードをダウンタイムなしで実行できますか？

TODO

-

4.96.6 データベース管理

PostgreSQL を使用する理由は？

PostgreSQL は、オペレーティングシステムの分野で Linux が表すデータベーススペースでは PostgreSQL に相当すると考えています。Postgres の現在の最新のメジャーバージョンはバージョン 14 で、そのまま出荷されます。

- 物理的およびロジカルなネイティブストリーミングレプリケーション
- 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
- 水平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垂直性を改善するためのデータベース分野で非常によく知られた手法単一インスタンスでのスケラビリティ
- 拡張性、地理用の PostGIS などの拡張機能 データベース
- 垂直スケラビリティのための並列クエリ JSON サポート、両方のマルチモデルハイブリッドデータベースの解放標準 SQL を介してクエリされる構造化 非構造化データ

などなど…

単一の PostgreSQL インスタンスでいくつかのデータベースをホストする必要がありますか？

単一のマイクロサービスアプリケーションによって完全に管理される単一のデータベースに、単一の PostgreSQL クラスター（プライマリサーバーとマルチプルのスタンバイサーバとして使用）を割り当てることをお勧めします。ただし、「postgres」スーパーユーザ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することができます（利用サブジェクトなりソースに依存します）。

この推奨事項の理由は、マイクロサービスに基づくクラウドネイティブコンセプトにあります。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独占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、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、スキーマ管理や移行を含め、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、デフォルトでアプリケーションユーザと前述のアプリケーションユーザが所有するアプリケーションデータベースを作成することで、この方法ですぐに使用できるように設計されています。

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、次の機能が強化されます。

- リソース管理：PostgreSQL、CPU、およびメモリの制約通常、リソースはインスタンスレベルで処理され、データベースレベル、Kubernetes との統合を容易にする Pood レベルのリソース管理ポリシー
- 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）：指定 PostgreSQL は、インスタンスレベル、インスタンスごとに 1 つのデータベースを持つことにより、PITR が簡素化されます運用、保持ポリシー管理の差別化、およびバックアップのデータ保護を強化
- アプリケーションの更新：各アプリケーションが更新を決定できるようにします異なる所有者が所有する他のデータベースに影響を与えないポリシー 用途
- データベースの更新：各アプリケーションがどの PostgreSQL を決定できるか使用するバージョン、および別のバージョンにアップグレードするタイミング PostgreSQL のメジャーバージョンとその条件（カットオーバーなど） 時間

< ! - 最高のビジネス継続性成果のために推奨されるアーキテクチャは何ですか？

TODO

CloudNativePG によって自動的に作成されるロールやデータベースなどのグローバルオブジェクトは何ですか？

TODO

Q：宣言的な設定による Postgres ロールの管理のサポート

TODO

Q：表領域のサポート

TODO

Q：GUI

TODO

Q：モニタリング

TODO

Q：ロギング

TODO

インスタンスを停止または開始するにはどうすればよいですか？

TODO>