



Trusted Postgres Architect

リリース *23.40.0*

EnterpriseDB Corporation

2025 年 10 月 06 日

目次

第 1 章 Introduction	3
1.1 Introduction	3
1.1.1 はじめに	3
1.1.2 TPA では何ができますか?	3
1.1.3 どのように使用すればよいですか?	3
1.1.4 それはただの Postgres です	6
1.1.5 TPA でのバージョンニング	6
1.1.6 はじめに	7
1.2 Trusted Postgres Architect release notes	7
1.2.1 Trusted Postgres Architect 23.40.0 release notes	8
1.2.2 Trusted Postgres Architect 23.39.0 release notes	9
1.2.3 Trusted Postgres Architect 23.38.1 release notes	13
1.2.4 Trusted Postgres Architect 23.38.0 release notes	14
1.2.5 Trusted Postgres Architect 23.37.0 release notes	18
1.2.6 Trusted Postgres Architect 23.36.0 release notes	20
1.2.7 Trusted Postgres Architect 23.35.0 release notes	23
1.2.8 Trusted Postgres Architect 23.34.1 release notes	28
1.2.9 Trusted Postgres Architect 23.34 release notes	28
1.2.10 Trusted Postgres Architect 23.33 release notes	30
1.2.11 Trusted Postgres Architect 23.32 release notes	32
1.2.12 Trusted Postgres Architect 23.31 release notes	33
1.2.13 Trusted Postgres Architect 23.30 release notes	34
1.2.14 Trusted Postgres Architect 23.29 release notes	35
1.2.15 Trusted Postgres Architect 23.28 release notes	35
1.2.16 Trusted Postgres Architect 23.27 release notes	37
1.2.17 Trusted Postgres Architect 23.26 release notes	37
1.2.18 Trusted Postgres Architect 23.25 release notes	38
1.2.19 Trusted Postgres Architect 23.24 release notes	39
1.2.20 Trusted Postgres Architect 23.23 release notes	40
1.2.21 Trusted Postgres Architect 23.22 release notes	41
1.2.22 Trusted Postgres Architect 23.21 release notes	41
1.2.23 Trusted Postgres Architect 23.20 release notes	41
1.2.24 Trusted Postgres Architect 23.19 release notes	42
1.2.25 Trusted Postgres Architect 23.18 release notes	43
1.2.26 Trusted Postgres Architect 23.17 release notes	44
1.2.27 Trusted Postgres Architect 23.16 release notes	44
1.2.28 Trusted Postgres Architect 23.15 release notes	45
1.2.29 Trusted Postgres Architect 23.14 release notes	45
1.2.30 Trusted Postgres Architect 23.13 release notes	46

1.2.31	Trusted Postgres Architect 23.12 release notes	46
1.2.32	Trusted Postgres Architect 23.1 to 23.11 release notes	47
1.3	TPA installation	55
1.3.1	クイックスタート	55
1.3.2	TPA をインストールする場所	56
1.3.3	TPA パッケージのインストール	56
1.3.4	TPA Python 環境のセットアップ	56
1.3.5	インターネットまたはネットワークアクセスエアギャップなしで TPA をインストールする	57
1.3.6	TPA インストールの検証	58
1.3.7	TPA のアップグレード	58
1.3.8	Ansible バージョン	58
1.4	Open source TPA	58
1.4.1	Trusted Postgres ArchitectTPA とは何ですか?	58
1.4.2	次のステップ	58
1.4.3	TPA オープンソースのよくある質問	59
1.5	Installing TPA from source	59
1.5.1	クイックスタート	59
1.5.2	クローンとセットアップ	60
1.5.3	依存関係	60
第 2 章	Creating a cluster	63
2.1	A First Cluster Deployment	63
2.1.1	TPA のインストール	63
2.1.2	Docker のインストール	63
2.1.3	TPA を使用した構成の作成	64
2.2	Cluster configuration	67
2.2.1	クイックスタート	67
2.2.2	構成オプション	67
2.2.3	ソフトウェアの選択	70
2.2.4	オーバーライド	73
2.2.5	Ansible Tower	74
2.2.6	ビーコンエージェント	74
2.2.7	Git リポジトリ	74
2.2.8	ボールドパスワードのキーリングバックエンド	74
2.2.9	セキュリティ基準へのコンプライアンス	75
2.2.10	例	75
2.3	tpaexec deploy	77
2.3.1	前提条件	78
2.3.2	クイックスタート	78
2.3.3	選択的な展開	79
2.3.4	deploy.yml	79
2.4	tpaexec provision	79
2.4.1	前提条件	79
2.4.2	クイックスタート	80
2.4.3	オプション	80
2.4.4	インスタンスへのアクセス	81

2.4.5	生成されたファイル	81
2.5	tpaexec test	83
2.5.1	クイックスタート	83
第 3 章	Architectures	85
3.1	PGD-Always-ON	85
3.1.1	クラスター構成	85
3.2	PGD Lightweight	87
3.2.1	クラスター構成	88
3.3	BDR-Always-ON	89
3.3.1	クラスター構成	90
3.4	M1	91
3.4.1	フェイルオーバー管理	91
3.4.2	クラスター構成	92
3.5	PGD-S	94
3.5.1	クラスター構成	94
3.6	PGD-X	96
3.6.1	クラスター構成	96
第 4 章	Platforms	99
4.1	aws	99
4.1.1	API アクセス設定	99
4.1.2	はじめに	101
4.1.3	構成	102
4.2	bare(-metal servers)	106
4.2.1	SSH アクセス	107
4.2.2	ディストリビューションのサポート	107
4.2.3	IP アドレス	107
4.2.4	新しく始める	108
4.3	Docker	108
4.3.1	概要	108
4.3.2	オペレーティングシステムの選択	108
4.3.3	Docker のインストール	109
4.3.4	Docker コンテナ管理	111
第 5 章	Customizations	113
5.1	Cluster configuration	113
5.1.1	config.yml	113
5.1.2	変数	114
5.1.3	クラスター変数	114
5.1.4	インスタンス変数	115
5.1.5	instance_defaults	116
5.1.6	ロケーション	117
5.2	Instance configuration	117
5.2.1	システムレベルの構成	118
5.2.2	Postgres	119
5.2.3	その他のコンポーネント	120
5.2.4	Postgres を起動した後	120

5.3	Building from source	121
5.3.1	クイックスタート	121
5.3.2	構成	122
5.3.3	ローカルソースディレクトリ	122
5.3.4	リビルド	123
5.4	TPA hooks	124
5.4.1	概要	124
5.4.2	汎用フック	124
5.4.3	PGD フック	126
5.4.4	その他のフック	127
5.4.5	新しいフック	127
第 6 章	Cluster Management	129
6.1	Upgrading your cluster	129
6.1.1	はじめに	129
6.1.2	構成	130
6.1.3	BDR-Always-ON から PGD-Always-ON へのアップグレード	130
6.1.4	PGD-Always-ON から PGD-X へのアップグレード	131
6.1.5	PGD-Always-ON	134
6.1.6	BDR-Always-ON	134
6.1.7	M1	134
6.1.8	アップグレードプロセスの制御	135
6.1.9	ノードのサブセットのアップグレード	135
6.1.10	パッケージバージョンの選択	135
6.2	tpaexec switchover	136
6.2.1	例	136
6.2.2	アーキテクチャオプション	136
6.2.3	Repmgr リダイレクト pgbouncer	137
6.3	BDR/HAProxy server pool management	137
6.3.1	例	137
6.4	tpaexec rehydrate	138
6.4.1	前提条件	138
6.4.2	例	138
6.4.3	段階的に Rehydrate する	140
6.4.4	付録	140
第 7 章	Miscellaneous	141
7.1	TPA and Ansible Tower/Ansible Automation Platform	141
7.1.1	AAP の初期セットアップ	141
7.1.2	クラスターのセットアップ	142
7.1.3	複数のインベントリに 1 つのプロジェクトを使用する	144
7.1.4	AAP で TPA を更新	144
7.1.5	TPA の EE をビルドする	145
7.2	TPA, Ansible, and sudo	146
7.2.1	Ansible sudo 呼び出し	146
7.2.2	おすすめ	147
7.2.3	SSH および sudo パスワード	147

7.2.4	sudo オプション	148
7.2.5	ロギング	148
7.2.6	ローカル権限	148
7.3	TPA - PuTTY Configuration guide	148
7.3.1	キー変換	149
7.3.2	PuTTY の構成	149
7.4	Troubleshooting	150
7.4.1	Python 仮想環境を再作成します	150
7.4.2	資格情報に関する奇妙な AWS エラー	150
7.4.3	ロギング	151
7.4.4	クラスターテスト	151
7.4.5	TPA サーバーテスト	151
7.4.6	特定のタスクを含めるまたは除外する	151
7.5	Reference	151
7.5.1	Installation	151
7.5.2	Administration	156
7.5.3	Services	160
7.5.4	Configuration	192
7.5.5	Postgres Configuration	214
7.5.6	Commands	224
7.6	Selective task execution	243
7.6.1	タスクセクターを使用する	243
7.6.2	例	243
7.6.3	tpaexec deploy でサポートされているセクター	244
7.6.4	tpaexec test でサポートされているセクター	248

© Copyright EnterpriseDB UK Limited 2015-2025 - 無断複写・転載を禁じます。

第1章 Introduction

1.1 Introduction

1.1.1 はじめに

TPA は、Ansible を使用して EDB の推奨事項に従って Postgres クラスターを展開するオーケストレーションツールです。

TPA は、Postgres の展開とサポートに関する長年の苦勞して得た経験に基づいて、EDB が追従するベストプラクティスを具体化しています。これらの推奨事項は、運用環境と同様に簡単なテストベッドのセットアップに適用されます。

1.1.2 TPA では何ができますか？

TPA は、トポロジから構成の最小の詳細まで Postgres クラスターを記述するために使用できる宣言的構成メカニズムを中心に構築されています。

`tpaexec configure` を実行して、いくつかの高レベルの選択たとえば、インストールする Postgres のバージョンなどに基づいて、初期クラスター構成を生成することから始めます。デフォルト構成はそのまま使用できますが、ニーズに合わせて編集できます。生成される構成は単なるテキストファイル `config.yml` です。

この構成を使用して、TPA は次のことを行うことができます。

1. サーバー AWS EC2 インスタンスまたは Docker コンテナなど、クラスターをホストするために必要なその他のリソースをプロビジョニングしますまたは、接続の詳細を指定するだけで既存のサーバーまたは VM に展開できます。
2. オペレーティングシステムを構成しますカーネル設定の調整、ユーザーと SSH キーの作成、パッケージのインストール、systemd サービスの定義、ログローテーションの設定など。
3. Postgres および関連コンポーネントたとえば、PGD、Barman、pgbouncer、repmgr、およびさまざまな Postgres 拡張機能などをインストールおよび構成します。
4. 展開後にクラスターで自動テストを実行します。
5. 構成に将来の変更を展開します Postgres 設定の変更、パッケージのインストールとアップグレード、新しいサーバーの追加など。

1.1.3 どのように使用すればよいですか？

TPA を使用するには、TPA をインストールし、`tpaexec setup` コマンドを実行する必要があります。ご使用のプラットフォームの [TPA installation](#) に従ってください。

TPA は 4 つの異なるステージで動作して、Postgres クラスターを起動します。

- クラスターの生成 [クラスター構成](#)
- クラスターをホストする [デプロイメントのプロビジョニング](#) サーバー VM、コンテナ
- プロビジョニングされたインスタンスへの *A First Cluster Deployment* ソフトウェア
- [テスト](#) デプロイされたクラスター

```
# 1. Configuration: decide what kind of cluster you want

[tpa]$ tpaexec configure clustername --architecture M1 --platform aws \
    --postgresql 14 \
    --failover-manager repmgr

# 2. Provisioning: create the servers needed to host the cluster

[tpa]$ tpaexec provision clustername

# 3. Deployment: install and configure the necessary software

[tpa]$ tpaexec deploy clustername

# 4. Testing: make sure everything is working as expected

[tpa]$ tpaexec test clustername
```

TPA は、ラップトップ、EC2 インスタンス、またはネットワーク経由でクラスターのサーバーに到達できるマシンから実行できます。

これは *TPA capabilities and supported software* です。

構成

クラスター構成

コマンドは、選択したオプションに基づいて、クラスターを記述する単純な YAML 構成ファイルを生成します。この構成はすぐに使用できますが、ニーズに合わせて変更できます。構成ファイルの編集は、[クラスター構成](#) の作成前と作成後の両方での通常の方法です。

この段階で、クラスターのアーキテクチャとプラットフォームを選択する必要があります。**アーキテクチャ**は、特定の目的に Postgres をセットアップするためのサーバーとソフトウェアの推奨レイアウトです。例には、「M1」プライマリおよびストリーミングレプリカを備えた Postgres および「PGD-Always-ON」Always On 構成の EDB Postgres 分散 5 が含まれます。**プラットフォーム**は、AWS、Docker、ベアメタルサーバーなどのアーキテクチャを展開するサーバーをホストする手段です。

プロビジョニング

tpaexec provision

コマンドは、クラスターに必要なインスタンスとその他のリソースを作成します。プロセスの詳細は、クラスターの構成中に選択したアーキテクチャ M1 などとプラットフォーム AWS などによって異なります。

たとえば、必要な特権で AWS アクセスが与えられると、TPA は EC2 インスタンス、VPC、サブネット、ルーティングテーブル、インターネットゲートウェイ、セキュリティグループ、EBS ボリューム、Elastic IPなどをプロビジョニングします。

「ベア」プラットフォームを選択し、接続の詳細を提供することにより、既存のサーバーを「プロビジョニング」することもできます。これらはベアメタルサーバーであるか、クラウドプラットフォームで別途プロビジョニングされたサーバーであるかにかかわらず、TPA によって作成されたかのように使用できます。

単一のプラットフォームに制限されません。必要に応じて、クラスターをいくつかの AWS インスタンスマルチプルのリージョンと一部のオンプレミスサーバー、または他のデータセンターのサーバーに分散できます。

プロビジョニングステージの最後に、基本的なオペレーティングシステムがインストールされた必要な数のインスタンスができます。TPA は、SSH 経由でルートに `sudo` を使用してアクセスできます。

導入

tpaexec deploy

コマンドは、プロビジョニングされたサーバーに Postgres およびその他のソフトウェアをインストールおよび構成します。TPA によって作成された場合とされていない場合があります。ただし、SSH と `sudo` アクセスが利用できる限り誰が作成したかは関係ありません。これには、レプリケーション、バックアップなどの設定が含まれます。

展開段階の最後に、Postgres が起動して実行されます。

テスト

tpaexec test コマンドは、展開されたクラスターに対してさまざまなアーキテクチャとプラットフォーム固有のテストを実行して、予想通りに動作していることを確認します。

テスト段階の最後には、完全に機能するクラスターが完成します。

インクリメンタルな変更

TPA は、プロビジョニング、展開、およびテストがべき等であるように注意深く設計されています。それらを実行し、`config.yml` を変更し、プロセスを再度実行して変更を展開できます。構成またはインスタンスで何も変更されない場合、プロセス全体を再実行しても何も変更されません。

クラスター管理

クラスターが起動して実行されると、TPA は、構成変更、スイッチオーバー、ゼロダウンタイムのマイナーバージョンのアップグレードなどの便利なクラスター管理機能を提供します。これらの機能により、手動で変更を行うよりもクラスターの管理が簡単かつ安全になります。

Ansible を介して拡張可能

TPA は *Instance configuration* をサポートしているため、`config.yml` を編集し、プロビジョニング/展開/テストを再実行するだけで、多くのことができます。TPA が既にサポートしているものを超える必要がある場合は、次のように書くことができます

- *TPA custom commands*、クラスターで実行するプレイブックの作成を簡単にします。クラスターディレクトリに `commands/xyz.yml` を作成し、`tpaexec xyz /path/to/cluster` を使用して呼び出すだけです。自動化する必要がある管理タスクまたはプロセスに最適です。
- *TPA custom tests*、環境とアプリケーションに固有の詳細な検証でビルトインテストを強化します。`tpaexec test` を使用してすべてのテストを均一で反復可能な方法で実行すると、危機に対処するときも、日常的なクラスター管理中にも、重要な要素を見逃すことはありません。
- *TPA hooks*、展開のさまざまな段階で呼び出されます。たとえば、`hooks/pre-deploy.yml` のタスクは、メイン展開の前に実行されます。`post-deploy` を含む他にも多くのフックがあります。これにより、Ansible の全機能を自由に利用できます。

1.1.4 それはただの Postgres です

TPA は、多くの機能が構成された複雑なクラスターを作成できますが、結果は単なる Postgres です。インストールは、生活を簡素化するように設計されたいくつかの規則に従いますが、あなたとデータベースの間には、隠された魔法も何も存在しません。他の Postgres インストールで実行できることはすべて TPA クラスターで実行できます。

1.1.5 TPA でのバージョニング

TPA は、以前にメジャーバージョンが年から派生する日付ベースのバージョン管理スキームを使用していました。バージョン 23 から TPA はセマンティックバージョニングに移行し、最初は 2 部の `major-minor` パターンを使用して、バージョン 23.34.1 で完全な 3 部セマンティックバージョニングを採用しました。このスキームでは、メジャーバージョンは、以下の下位互換性の原則に準拠するために必要な場合にのみインクリメントされます。

下位互換性

TPA の重要な開発原則は、下位互換性を維持することであるため、ユーザーが TPA の最新バージョン以外のものが必要になる理由はありません。下位互換性を次のように定義します。

- TPA X.a で作成された `config.yml` は、TPA X.b で有効です。ここで、 $b \geq a$
- その `config.yml` から作成されたクラスターは、TPA X.b で保守および再展開が可能です。

したがって、新しいメジャーバージョンは、下位互換性の中断を意味します。そのため、私たちはメジャーバージョンのリリースを回避することを目指しており、例外的な状況でのみそうします。

1.1.6 はじめに

システムの *TPA installation* に従って、次に **クラスター構成** に従います。

1.2 Trusted Postgres Architect release notes

Trusted Postgres Architect ドキュメントでは、Trusted Postgres Architect 23 の最新バージョンについて説明しています。

Trusted Postgres Architect version	Release Date
<i>Configuring lvm-direct for production</i>	02 Oct 2025
<i>Trusted Postgres Architect 23.39.0 release notes</i>	22 Aug 2025
<i>Trusted Postgres Architect 23.38.1 release notes</i>	25 Jun 2025
<i>Trusted Postgres Architect 23.38.0 release notes</i>	09 Jun 2025
<i>Trusted Postgres Architect 23.37.0 release notes</i>	24 Mar 2025
<i>Trusted Postgres Architect 23.36.0 release notes</i>	19 Feb 2025
<i>Trusted Postgres Architect 23.35.0 release notes</i>	25 Nov 2024
<i>Trusted Postgres Architect 23.34.1 release notes</i>	09 Sep 2024
<i>Trusted Postgres Architect 23.34 release notes</i>	22 Aug 2024
<i>Trusted Postgres Architect 23.33 release notes</i>	24 Jun 2024
<i>Trusted Postgres Architect 23.32 release notes</i>	15 May 2024
<i>Trusted Postgres Architect 23.31 release notes</i>	19 Mar 2024
<i>Trusted Postgres Architect 23.30 release notes</i>	19 Mar 2024
<i>Trusted Postgres Architect 23.29 release notes</i>	15 Feb 2024
<i>Trusted Postgres Architect 23.28 release notes</i>	23 Jan 2024
<i>Trusted Postgres Architect 23.27 release notes</i>	19 Dec 2023
<i>Trusted Postgres Architect 23.26 release notes</i>	30 Nov 2023
<i>Trusted Postgres Architect 23.25 release notes</i>	14 Nov 2023
<i>Trusted Postgres Architect 23.24 release notes</i>	17 Oct 2023
<i>Trusted Postgres Architect 23.23 release notes</i>	21 Sep 2023
<i>Trusted Postgres Architect 23.22 release notes</i>	06 Sep 2023
<i>Trusted Postgres Architect 23.21 release notes</i>	05 Sep 2023
<i>Trusted Postgres Architect 23.20 release notes</i>	01 Aug 2023
<i>Trusted Postgres Architect 23.19 release notes</i>	12 Jul 2023
<i>Trusted Postgres Architect 23.18 release notes</i>	23 May 2023
<i>Trusted Postgres Architect 23.17 release notes</i>	10 May 2023
<i>Trusted Postgres Architect 23.16 release notes</i>	21 Mar 2023
<i>Trusted Postgres Architect 23.15 release notes</i>	15 Mar 2023
<i>Trusted Postgres Architect 23.14 release notes</i>	23 Feb 2023
<i>Trusted Postgres Architect 23.13 release notes</i>	22 Feb 2023
<i>Trusted Postgres Architect 23.12 release notes</i>	21 Feb 2023
<i>Trusted Postgres Architect 23.1 to 23.11 release notes</i>	21 Jun 2023 to 31 Jan 2023

1.2.1 Trusted Postgres Architect 23.40.0 release notes

リリース日 2025 年 10 月 2 日

Trusted Postgres Architect 23.40.0 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

機能強化

Description	Addresses
EDB Postgres 分散 PGD v5 クラスターを v6 にアップグレードするためのサポートを追加しました。	
PGD-S クラスターの構成の拡張サポート。	
<code>`pgd-proxy-config`</code> フックが導入されました。	49911
EFM 5.1 以降に 3 つの <code>`release.vip.*`</code> プロパティを追加しました。	
TPA では、証明書に RSA キーサイズを設定できるようになりました。	
TPA が Postgres サーバー TLS 証明書とキーを生成する方法と、これをオーバーライドする方法を文書化しました。	52049

の変更

Description	Addresses
TPA は、フェールオーバーマネージャーとして repmgr を使用した EPAS クラスターをサポートするようになりました。	

バグ修正

Description	Addresses
PGD 6 クラスターでバックアップが構成されていない問題を修正しました。	
PGD-Always-ON での <code>`update_host`</code> 変数のサポートを修正しました。	52558, 45181
HAProxy ヘルスチェックは、SSL 対応の Patroni クラスターで HTTPS を正しく使用するようになりました。	
<code>edbpgc</code> Postgres で使用した場合の <code>`pgaudit`</code> 拡張機能の問題を修正しました。	
グローバルプロキシルーティングが PGD 5.9 で正しく設定されなかった問題を修正しました。	
TPA は、PEM エージェント Barman ノードで <code>`postgres/pgpass`</code> タスクを正しくスキップするようになりました	

1.2.2 Trusted Postgres Architect 23.39.0 release notes

リリース日 2025 年 8 月 22 日

Trusted Postgres Architect 23.39.0 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

ハイライト

- s390x アーキテクチャのサポート
- edb_stat_monitor 拡張機能のサポート
- auto.rewind、jdbc.properties、jdbc.loglevel、および check.vip.timeout の EFM プロパティのサポート

機能強化

Description	Addresses
Ubuntu 24.04 での PEM のサポートを追加しました。このディストリビューションには、展開エラーを発生させた特定のパッケージが定義されていませんでした。さらに、これにより、mod_wsgi または edb の mod_wsgi モジュールが適用される場所でデフォルトで有効になります。 認識される拡張機能に `edb_stat_monitor` を追加しました。ユーザーが `postgres_extensions` または `postgres_databases` の下で名前付けられた拡張機能のリストのエントリーとして `edb_stat_monitor` を指定すると、TPA は正しいパッケージのインストール、拡張機能の作成、および `shared_preload_libraries` に含めることを処理します。	
s390x 別名 IBM Z および LinuxONE のサポートを追加します。TPA が s390x CPU アーキテクチャをサポートするようになりました。サポートされている s390x オペレーティングシステムは、RHEL 8、9、および SLES 15 です。TPA はこれらのシステムで実行でき、クラスター展開のターゲットホストとして使用できます。s390x バイナリはすべての PyPI パッケージで利用できるわけではないため、セットアップ中に PyPI からのインストールに依存するのではなく、EDB から `tpaexec-deps` パッケージをインストールすることを強くお勧めします。同様に、s390x の PGDG RPM リポジトリがないため、TPA の EDB Repos にアクセスして、パッケージから Postgres およびその他のクラスターコンポーネントをインストールする必要があります。	
EFM 5.1 以降に `auto.rewind` プロパティを追加しました。EFM 5.1 以降、障害が発生したプライマリ DB サーバーを再構築する新機能の一部である新しいプロパティ `auto.rewind` があります。	
EFM `jdbc.properties` プロパティのサポートを追加します。これにより、EFM 5.0 以降での `jdbc.properties` プロパティのサポートが追加されます。	

次のページに続く

表 5 – 前のページからの続き

Description	Addresses
EFM 5.1 以降に `jdbc.loglevel` プロパティを追加します。EFM 5.1 から、JDBC ドライバーからログに記録される情報を増やすために使用される新しいプロパティ `jdbc.loglevel` があります。このプロパティは、接続に問題がある場合に詳細情報を取得するために使用できます。データベース接続に ssl を使用する場合。	
EFM 5.1 以降に `check.vip.timeout property` を追加します。EFM 5.1 以降、スタンバイを昇格する前に VIP を使用する場合に到達可能かどうかを EFM がチェックし続ける時間を制御するために使用される新しいプロパティ `check.vip.timeout` があります。	

の変更

Description	Addresses
Ubuntu 20.04Focal は、古いディストリビューションになりました。Ubuntu 20.04 はアップストリームのサポートが終了し、TPA でも完全にサポートされなくなりました。	
古い EDB/2q リポジトリのサポートを削除しました。古い 2q リポジトリと EDB Repos 1.0 が利用できなくなったため、TPA はそれらを使用しようとせず、それらに関連する構成をチェックしません。これらのリポジトリで以前に利用できたすべてのパッケージは、EDB Repos 2.0 リポジトリで利用できるようになりました。	
TPA は、SLES で EDB リポジトリを使用する場合、PGDG リポジトリを抑制するようになりました。SLES システムで `tpaexec configure` を実行する場合、他のオペレーティングシステムの動作と一致する EDB リポジトリまたは PGDG リポジトリの両方ではなく、いずれかを使用します。これにより、両方の場所で利用可能な barman などのパッケージが、他のパッケージと同じリポジトリから一貫してソースされることが保証されます。	
TPA は、デフォルトの PGD プロキシバージョンとして `bdr_package_version` を使用するようになりました。`pgd_proxy_package_version` および `pgd-cli_package_version` は、config.yml で明示的に定義されている場合があります。明示的に定義されていない場合、`bdr_package_version` の値がデフォルトで設定されている場合。それらが何も設定されていない場合、パッケージの最新の利用可能なバージョンがインストールされます。これらのパッケージバージョンを別の値に設定することはサポートされておらず、テストにのみ役立ちます。	49150
`vacuum_cost_delay` のデフォルトを 2 ミリ秒に削減します。最近の Postgres バージョンでは、`vacuum_cost_delay` の推奨デフォルトが 2 ミリ秒に削減されました。TPA はこれに一致するようになりました。	

[次のページに続く](#)

表 6 – 前のページからの続き

Description	Addresses
ノードのプロモーションロジックと EFM の <code>`efm-not-promotable`</code> ロールを文書化しました。 <code>`efm.md`</code> ドキュメントに新しいセクションを追加し、フェールオーバー中にノードがプロモーションの対象かどうかを TPA が判断する方法を説明します。この更新では、監視ノード、カスケードスタンバイ、 <code>`efm-not-promotable`</code> ロールで明示的にマークされたノードのロールを含むプロモート性のルールを明確にします。この機能強化は、ユーザーがフェールオーバー動作を理解および制御するのに役立ち、EFM 管理クラスターでの意図しないプロモーションのリスクを削減します。	
テンプレート変数の <code>`include_vars`</code> 動作を文書化します。 <code>`include_vars`</code> モジュールは式をすぐに解析して評価するため、ネストされた変数は <code>`config.yml`</code> からロードされた時点では存在せず、したがって、テンプレート式で評価されるときに未定義です。これは例を使用して文書化されたため、ユーザーがわかりやすくなります。	45722
Rocky Linux 組織の Docker イメージを使用します。TPA は、Docker Hub の「公式」イメージではなく、Rocky Linux 組織の docker イメージを使用するようになりました。これらはより頻繁に更新されるため、新しいパッケージで依存関係の問題が発生する可能性が低くなります。	
PGD 6 アーキテクチャのアーキテクチャ情報を更新しました。PGD6 アーキテクチャに正しいメタデータがあるため、 <code>`tpaexec info architectures`</code> の出力に予想通りに表示されます。	

バグ修正

Description	Addresses
<code>replication`</code> ユーザーがプライマリノードの <code>.pgpass`</code> から欠落している問題を修正しました。 <code>replication_user`</code> エントリは、 <code>failover_manager`</code> として <code>`efm`</code> または <code>`patroni`</code> のいずれかによって管理される <code>`replica`</code> および <code>`primary`</code> ノードの両方の <code>.pgpass`</code> ファイルに必要です。以前は、 <code>`replica`</code> ノードの <code>.pgpass`</code> ファイルにのみ追加されていたため、スイッチオーバー後にプライマリノードへの接続問題が発生しました。	48224
TPA が PGD6 クラスターの加入者専用グループのルーティングを有効にしようとする問題を修正しました。以前は、TPA は、PGD6 展開中に誤って加入者専用グループのルーティングを有効にしようとしていました。このプロパティは、PGD6 のサブスライバ専用グループの定義のコア部分を形成するため、編集できません。	49673

次のページに続く

表 7 – 前のページからの続き

Description	Addresses
PEM 10.1.1 で PEM 展開が失敗する問題を修正しました。TPA タスク <code>`Register PEM backend database server for monitoring and configuration`</code> は、<code>pem.setup</code> SQL ファンクションを明示的に呼び出します。これは PEM のパブリック API の一部とはみなされず、シグネチャが 10.1.0 と 10.1.1 の間で変更されたため、TPA のこのファンクションへの呼び出しが失敗します。この修正では、インストールされている PEM サーバーバージョンに応じてファンクションコールを調整することにより問題を解決します。この変更に対応するために、インストールされた PEM バージョンに関するファクトを収集する新しい <code>`pem/server/facts`</code> Ansible ロールを導入しました。これは、TPA が Postgres Expert が存在しない PEM 10 で <code>`postgresexpert.sql`</code> を実行しようとしなくても意味します。	
複数のフェイルオーバーマネージャーを使用する場合に <code>`harp_dcs_user`</code> が作成されない問題を修正しました	49150
<code>`bdr_node_groups`</code> をオーバーライドできない問題を修正しました。<code>`tpaexec configure`</code> の <code>--overrides-from</code> 引数で提供されたファイルが <code>`cluster_vars`</code> で <code>`bdr_node_groups`</code> を設定する場合、このコンテンツは自動的に作成されるノードグループに追加されます。	50550
TPA が PGD6 ノードの DSN で <code>`pgdproxy`</code> ユーザーを使用する問題を修正しました。PGD バージョン 6 には、PGD プロキシに代わるビルトイン接続マネージャーがあるため、<code>pgdproxy</code> Postgres ユーザーは DSN で使用しないでください。Connection Manager への接続文字列を定義するために、新しい変数 <code>`bdr_connection_manager_route_dsn`</code> が導入されました。ユーザーは <code>`config.yml`</code> で <code>`bdr_connection_manager_dsn_attributes`</code> を追加定義して、DSN で追加の接続パラメーターを指定できます。	
複数の Barman サーバーを使用する Patroni クラスターの <code>`ignore_slots`</code> の問題を修正しました。Patroni クラスターの場合、TPA は、クラスター内の Barman ノードに対応する <code>ignore_slots</code> 設定を動的に生成するようになりました。この変更により、Patroni は、各 Barman ノード用に作成された物理スロットを削除しません。これは、複数のバックアップサーバーがある Patroni クラスターにとって特に重要です。例、共有 Barman ノードを含む。	50748
TPA Docker イメージの使いやすさを修正し、信頼性を向上させました。TPA の Python 3.12 最小要件を満たし、最新の依存関係との互換性を保証するためにベースイメージを Debian Trixie にアップグレードしました。 <code>--use-community-ansible`</code> フラグはサポートされなくなったため、削除しました。 <code>- `openssh-client`</code> パッケージを含めて、<code>`ssh-keygen`</code> の欠落による <code>`tpaexec`</code> の障害を防ぎ、すぐに使える機能を強化しました。 <code>- ビルドタイムバージョンニングの導入`</code> ARG 変数を追加して、ビルド中にホストから <code>`git describe`</code> の出力を渡し、バージョン追跡を向上させるために <code>/opt/EDB/TPA/VERSION`</code> として保存します。 <code>- 最新のセットアップ手順とベストプラクティスを反映するように`</code> <code>docker/README.md`</code> および <code>`docs/src/INSTALL-docker.md`</code> を修正しました。	
Barman が <code>`switch-wal`</code> を実行できない問題を修正しました。Postgres 15+ の場合、Barman ユーザーが <code>`pg_checkpoint`</code> ロールで作成されるようになりました。これにより、Barman はスーパーユーザーでなくても CHECKPOINT を実行できます。<code>`barman switch-wal --force`</code> は失敗しません。	

1.2.3 Trusted Postgres Architect 23.38.1 release notes

リリース日 2025 年 6 月 25 日

Trusted Postgres Architect 23.38.1 は、次の問題を解決するバグ修正リリースです。

バグ修正

Description	Addresses
ベアプラットフォームを使用する PGD6 の configure コマンドの問題を修正しました。PGD6 の新しくリリースされたアーキテクチャ PGD-S および PGD-X で tpaexecconfigure コマンドのバグを修正しました。`--platform bare`を使用してクラスターを生成しようとする、不明なプラットフォームエラーが発生します。この修正により、configure コマンドはベアプラットフォームの PGD6 構成ファイルを正常に生成します。 pgpass_users が pgpass ファイルに正しく追加されない問題を修正しました。以前は、<code>generate_password: true</code> で指定され、<code>pgpass_users</code> リストに含まれる <code>postgres_users</code> リスト内のユーザーは、初期展開で <code>~postgres/pgpass</code> ファイルに追加されません。これは、pgpass タスクが実行されたときにユーザーのパスワードがまだ存在せず、したがって、ユーザーがスキップされたためです。これは、すべての Postgres ユーザーが作成された後、pgpass タスクを再度呼び出すことにより修正されます。repmgr および replication ユーザーは、以前は <code>default_pgpass_users</code> リストに含まれていました。これらは pgpass_users に追加され、使用中のレプリケーションマネージャーの必要に応じて pgpass ファイルに追加されます。postgres_user デフォルトでは postgres または enterprisedb は、default_pgpass_users リストの一部です。これが postgres_user を含まない config.yml の pgpass_users リストによってオーバーライドされる場合、PEM 対応クラスターは、暗号化された postgres_user パスワードを必要とするため、エージェントの登録に失敗します。これは、PEM エージェントタスクの一部として postgres_user を ~postgres/pgpass ファイルに追加することにより修正されます。 無効な cluster_vars 辞書により、展開の失敗または予期しない構成が発生する問題を修正しました。以前は、無効な辞書が config.yml の cluster_vars 辞書として設定されている場合他の cluster_vars 変数を参照する cluster_vars 変数など、TPA は空の辞書のデフォルト値にフォールバックすることにより Ansible エラーを飲み込んでいた。これにより、すべての cluster_vars 変数が未定義になるため、それぞれが TPA デフォルト値に設定されました。結果のクラスターは、ユーザーが config.yml ファイルで指定したものとは完全に異なります。これは、構成ファイルのロード時に cluster_vars ディクショナリが定義され空でないことをアサートすることにより修正されます。また、最終的な救済として、cluster_vars 変数がデフォルトで空の辞書に設定されなくなりました。これにより、Ansible は group_vars の作成時にエラーをスローして終了できます。	49673

1.2.4 Trusted Postgres Architect 23.38.0 release notes

リリース日 2025 年 6 月 9 日

Trusted Postgres Architect 23.38.0 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

ハイライト

- Expanded と Essential アーキテクチャの両方での EDB Postgres 分散バージョン 6 の展開のサポート
- CVE-2025-2291 の緩和を含む PgBouncer 実装の改善
- PEM 10.1 のサポート

機能強化

Description	Addresses
PGD6 アーキテクチャのサポート。TPA は、PGD6 に基づいて PGD-X および PGD-S アーキテクチャを使用してクラスターを構成および展開できるようになりました PGD-S アーキテクチャは PGD Essential を実装し、PGD-X アーキテクチャは PGD Expanded を実装します。これらのアーキテクチャには、合理的なデフォルト構成があり、さまざまな構成オプションを受け入れて動作をカスタマイズします。PGD 6 展開には pgd-proxy が含まれません。代わりに、PGD のビルトイン接続マネージャーが構成されます。新しいアーキテクチャのテストサポートが追加されます。	
クラスター内に両方が存在する場合、Barman を監視するように PEM を構成します。クラスターが PEM を有効にして <code>--enable-pem</code> オプションを使用して構成され、Barman ノードを含む場合、次のアクションが自動的に実行されるようになります。- <code>enable_pg_backup_api</code> は <code>config.yml</code> で <code>true</code> に設定されます - <code>pem-agent</code> ロールは Barman ノードに割り当てられます - Barman エンドポイントローカル PEM エージェントに登録されています。これらの変更により、セットアップが簡素化され、PEM と Barman 間のシームレスな統合が保証されます。	
AWS での Rocky Linux 9 のサポートを追加しました。TPA は、 <code>aws</code> プラットフォームで Rocky Linux 9.5 を使用したクラスターの構成をサポートするようになりました。バージョンが指定されない場合、これは AWS の Rocky Linux のデフォルトバージョンになりました。	
SLES 15 で <code>pg_backup_api</code> のサポートを追加しました。PEM モニタリングが有効で、Barman ノードがクラスター内に存在する場合、TPA は <code>pg_backup_api</code> が SUSE Linux Enterprise Server 15 SLES 15 で構成されます。	

の変更

Description	Addresses
PEM 10.1 以降では、<code>PEM_DB_PORT</code> を文字列として扱います。PEM 10.1 では、Web アプリケーションからバックエンドサーバーまでのマルチホスト接続文字列のサポートを追加します。この変更をサポートするために、PEM の <code>config_setup.py</code> ファイルの <code>PEM_DB_PORT</code> パラメーターが整数ではなく文字列になりました。TPA は HA PEM 構成の展開をまだサポートしていませんが、PEM バージョンが 10.1 以上の場合、TPA はこのパラメーターを文字列として正しく設定するようになりました。 <code>repo</code> がタスクから除外される場合、TPA はリポジトリチェックをスキップするようになりました。<code>repo</code> タグは除外できますが、以前は <code>sys/repositories</code> ロールの下の子タスクのみをスキップしていました。また、どのリポジトリを使用するかをチェックする初期化タスクをスキップし、それらにアクセスするための資格情報が提供されていることを確認します。 <code>postgres_package_version</code> の動作を改善しました。<code>postgres_package_version</code> 設定により、TPA は EDB Postgres Advanced Server または EDB Postgres Extended Server をインストールしている Debian または Ubuntu システムに、さまざまな postgres 関連コンポーネントの選択したバージョンをインストールします。これにより、新しいパッケージバージョンがリポジトリで表示される場合の依存関係解決の問題を回避します。	
新しいタスクセクター <code>create_postgres_system_user</code> および <code>create_pgproxy_system_user</code> を追加しました。 <code>postgres_user</code> および <code>pgproxy_user</code> オペレーティングシステムユーザーをスキップできる新しいタスクセクターを追加しました。これにより、クラスターは、NIS などの集中ユーザー管理によって作成されたりモートユーザーを使用できます。これは <code>config.yml</code> で設定できます。 <code>cluster_vars:</code> <code>excluded_tasks:</code> <code>- create_postgres_system_user - create_pgproxy_system_user</code>	48601,44388
TPA は、スイッチオーバー中に PgBouncer を M1 repmgr クラスターの新しいプライマリにリダイレクトします。TPA は、<code>repmgr_redirect_pgbouncer</code> が true に設定されている repmgr + PgBouncer クラスターで <code>switchover</code> コマンドを使用した後、PgBouncer インスタンスが新しいプライマリノードにリダイレクトされるようになりました。<code>tpaexec switchover</code> コマンドは、PgBouncer インスタンスが新しいプライマリノードに接続するようになりました。最初のスイッチオーバーが実行されて最初のプライマリノードに戻った後、新しい <code>revert_redirect</code> 変数を追加変数として設定することもできます。 カスタムリポジトリ定義で <code>key_id</code> / <code>gpgkey</code> をオプションにしました。<code>key_id</code> および <code>gpgkey</code> パラメーター apt および yum カスタムリポジトリ定義の場合、基になるモジュールでは必要ありません。インストール前にこれを提供するのが簡単ではないユースケースがあります。この変更により、TPA は、カスタムリポジトリ定義でそれらを提供することを必須にしません。	

次のページに続く

表 10 – 前のページからの続き

Description	Addresses
PGD CLI 5.7.0+の`tpaexec test`の`verify-settings`チェックを修正しました。PGD CLI コマンド`pgd verify-settings`の出力は、PGD 5.7.0 で変更されました。PGD CLI の 5.7.0 より上のバージョンを使用する場合、TPA は出力を正しく解析するようになりました。`verify-settings`は、将来の PGD リリースで他のコマンドとともに非推奨になることに注意してください。これらのコマンドは、非推奨が発生するまで、新しいコマンドを呼び出すラッパーになります。 `barman_role`に権限を付与するためのロジックが改善されました。TPA は`postgresql_privs`モジュールを使用して`barman_role`に付与を適用するように、必要な場合にのみ変更が適用されます。さらに、PGD クラスターでは、TPA は 2 番目の展開で`bdr_database`を使用するため、DDL は PGD によってクラスター全体でレプリケートされます。	
config.yml の`cluster_vars`変数テンプレートを文書化しました。`config.yml`で不適切にテンプレート化された場合に定義されていないインベントリ変数に関連する予期しない動作による混乱を避けるために、`cluster_vars`で定義された変数の正しいテンプレート手順を作業例で説明するドキュメントを追加しました。 TPA は、configure 中に無効な CIDR が`--network`フラグに渡された場合、ArchitectureError を送出するようになりました。デフォルトでは、Python 標準ライブラリ`ipaddress`パッケージは、CIDR の'厳密な'解釈を強制します。これにより、使用される IP は範囲のネットワークアドレスである必要があります。以前は、ホストビットを含む`--network`フラグに渡される IP は、ValueError が発生したためスタックトレースをダンプしていました。その例外がキャッチされ、ArchitectureError が発生して、`--network`パラメーターに関する明確なメッセージをユーザーに表示します。 EFM application.name プロパティのデフォルト値を追加しました。ノードに EFM `application.name`プロパティが設定されていない場合、TPA はデフォルトとして Postgres `cluster_name`プロパティを使用します。EFM は、スイッチオーバーを実行するとき、または新しいスタンバイデータベースを構築するときにこの値を使用します。	48797
TPA は、SUSE で EDB リポジトリセットアップスクリプトを使用するようになりました。以前は、TPA は繰り返しの展開で機能しないため、SUSE で EDB リポジトリセットアップスクリプトを使用しませんでした。スクリプトがインストールしようとしているリポジトリが既に存在し、一意の名前が必要なため、Zypper は発生します。リポジトリが既にインストールされている場合、リポジトリのセットアップスクリプトタスクがスキップされるようになったため、この問題は発生しません。	

バグ修正

Description	Addresses
TPA は、専用のデータベースに <code>`pgbouncer_get_auth()`</code> ファンクションを作成します。 <code>`pgbouncer_get_auth()`</code> ファンクションは <code>`pg_catalog`</code> スキーマで作成され、 <code>`pgbouncer_auth_user`</code> に実行許可されています。このファンクションはすべてのデータベースで作成されましたが、これは PgBouncer には必要ありませんでした。このファンクションが <code>`pg_catalog`</code> スキーマで作成された場合、 <code>`pg_dump`</code> によって作成されたダンプに含まれないため、 <code>`pgd node upgrade`</code> プロセス中に障害が発生する場合があります。後のタスクは、このファンクションで <code>`GRANT`</code> を実行しようとしませんが、ファンクションは最初にダンプされていないため復元されないため、失敗します。現在、このファンクションは単一のデータベースでのみ作成され、 <code>`config.yml`</code> の <code>`pgbouncer_auth_database`</code> 変数の下で名前付けられます。これが含まれない場合はデフォルトの <code>`pgbouncer_auth_database`</code> になります。 <code>`pgbouncer`</code> ロールを持つ少なくとも 1 つのインスタンスがクラスターに含まれる場合にのみ作成されます。将来の TPA リリースはそのスキーマからファンクションを削除する可能性があるため、データベースが <code>`pg_catalog`</code> スキーマでこのファンクションを定義している場合、展開およびアップグレード中に警告も発行されます。PgBouncer <code>`auth_query`</code> が使用する <code>`pgbouncer_get_auth()`</code> ファンクション自分自身が更新されて、CVE-2025-2291 に対処しました。この脆弱性により、有効期限が切れたパスワードを使用した認証が可能になりましたが、 <code>a_query</code> メカニズムは、ユーザーパスワードの PostgreSQL で設定された <code>`VALID UNTIL`</code> 属性を考慮していませんでした。 <code>`--check`</code> オプションを使用すると、一部のタスクが誤ってスキップされる問題を修正しました。HARP のない PGD クラスターでは、 <code>`Read current configuration file if exists`</code> タスクをチェックモードで実行して、次の HARP チェックタスクを正しくスキップするために利用できる情報があることを確認する必要があります。ただし、デフォルトでは、Ansible はチェックモード中に <code>`shell`</code> モジュールを使用してタスクをスキップします。これは、このタスクが実行されず、後続のタスクで擬似障害が発生することを意味します。このタスクを実行する必要があることを Ansible に通知します。	42911, 45068
<code>`cluster_vars[patroni_conf_settings][bootstrap][dcs]`</code> を介して <code>`ignore_slots`</code> に追加された設定が最終的な構成にマージされないバグを修正しました。 レプリカロールを持つ BDR ノードの RAFT チェックがアップグレード中にスキップされない問題を修正しました。 <code>`subscriber-only`</code> ノードの物理レプリケーションは、フェールオーバーマネージャーとして <code>`repmgr`</code> をインストールし、 <code>`subscriber-only`</code> ノードを <code>`primary`</code> として指定し、別の BDR データノードを <code>`backup`</code> としてリストすることにより、 <code>`PGD`</code> クラスターで実現できます。このバックアップノードには、 <code>`replica`</code> ロールが付与されます。TPA は BDR データノードで RAFT が有効になっていることを想定しているため、この構成では PGD アップグレードプロセスが失敗しますが、物理レプリカ BDR データノード <code>`replica`</code> と <code>`bdr`</code> ロールの両方はデザインによってそうではありません。修正として、アップグレードプロセスの特定の BDR 固有のタスクは、 <code>`replica`</code> ロールを持つノードをスキップし、アップグレードを成功させます。	46186

1.2.5 Trusted Postgres Architect 23.37.0 release notes

リリース日 2025 年 3 月 24 日

Trusted Postgres Architect 23.37.0 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

ハイライト

- `upgrade` コマンドは、Patroni および EFM クラスターでの Postgres のマイナーバージョンのアップグレードをサポートするようになりました
- `repmgr` クラスターのアップグレードも改善されました

機能強化

Description	Addresses
M1 + Patroni クラスターのマイナーバージョンの Postgres のアップグレード。TPA は、フェールオーバーマネージャーとしてパトロニを使用する M1 クラスターで Postgres を最新のマイナーバージョンにアップグレードできるようになりました。アップグレードプロセスは、クラスター内の Barman サーバーで Barman を停止し、クラスター内のレプリカをアップグレードします。次に、一時的なプライマリとして 1 つのレプリカに切り替え、元のプライマリで Postgres をアップグレードし、元のプライマリに戻します。Patroni のクラスターの処理は、プロセス中に一時停止され、その後再開されます。次に、Barman が再起動され、クラスターヘルスチェックが実行されます。 M1 + repmgr クラスターのマイナーバージョンの Postgres アップグレードを改善しました。Witness ノードは、以前はアップグレードから省略されましたが、レプリカとともにアップグレードされるようになりました。Postgres サービスの再起動の信頼性が向上し、ノードでのパッケージのアップグレードが完了した直後に常に実行されます。	102662
M1 + EFM クラスターのマイナーバージョンの Postgres のアップグレード。TPA は、フェールオーバーマネージャーとして EFM を使用する M1 クラスターで Postgres を最新のマイナーバージョンにアップグレードできるようになりました。アップグレードプロセスは、クラスター内の Barman サーバーで Barman を停止し、クラスター内のレプリカをアップグレードします。次に、一時的なプライマリとして 1 つのレプリカに切り替え、元のプライマリで Postgres をアップグレードし、元のプライマリに戻します。EFM エージェントは、さまざまなサーバーで適切なタイミングで起動および停止されます。次に、Barman が再起動され、クラスターヘルスチェックが実行されます。 出力内の変更されたタスクから未変更のタスクを分離します。TPA のデフォルトの出力プラグインでは、「ok」を返すが変更のないタスクが、変更を報告したタスクから分離され、黄色で強調表示されます。 Docker インスタンスのプロビジョニング解除の速度が向上しました。Docker インスタンスのプロビジョニングを解除する場合、TPA はコンテナを停止する代わりに kill し、すべてのインスタンスに並列して実行します。	103595

次のページに続く

表 12 – 前のページからの続き

Description	Addresses
EFM 5.x の `check.num.sync.period` プロパティを追加しました。 EFM 5.0 以降、プライマリエージェントがプライマリデータベースで num_sync を変更する必要があるかどうかをチェックする頻度を定義する新しいプロパティ `check.num.sync.period` があります。 `config.yml` で指定できるようになりました。	

の変更

Description	Addresses
TPA は、構成が変更されない場合でも、EFM 構成ファイルが削除される場合、コピーするようになりました。 <code>efm.nodes</code> または <code>efm.properties</code> 構成ファイルがトップレベルの EFM ディレクトリに存在しない場合、構成変更がなくても、 <code>efm upgrade-conf</code> コマンドは <code>/raw</code> ディレクトリからそれらをコピーします。これにより、 <code>upgrade-conf</code> コマンドを実行してファイルをコピーする前に構成の変更が必要だった以前の動作が修正されます。	
TPA は、EDB リポジトリ設定の URI にアクセスできることを確認します。 EDB リポジトリは、EDB Repos 2.0 ドキュメントに従って、 <code>setup</code> スクリプトを使用してセットアップされます。 cURL 出力を <code>bash</code> にパイピングして実行します。ただし、ユーザーが存在しない <code>EDB_SUBSCRIPTION_TOKEN</code> またはリポジトリを cURL に渡すと、 <code>bash</code> は空の入力を実行するため、終了コードは静かに飲み込まれ、0 に置き換えられます。このバージョンでは、セットアップスクリプトのダウンロードを続行する前に、要求がリポジトリの GPG キーエンドポイントにディスパッチされ、 <code>404</code> 応答が返されないようにします。さらに、リポジトリが既に設定されている場合、EDB リポジトリのセットアップに関連するタスクがスキップされるようになりました。	
BDR-Always-ON が BDR バージョン 5 で構成されている場合、TPA は <code>ArchitectureError</code> を発生するようになりました。 <code>-a BDR-Always-ON</code> で <code>--bdr-version 5</code> が渡された場合、 <code>tpaexec configure</code> 中にアーキテクチャエラーが発生し、BDR バージョン 5 を <code>PGD-Always-ON</code> で使用する必要があることをユーザーに警告します。	

バグ修正

Description	Addresses
デプロイ中の <code>shared_preload_libraries</code> 計算を修正しました。 構成を変更しない 2 番目の展開シナリオでも、 <code>shared_preload_libraries</code> の不要で制御されない書き換えをトリガーし、その後 Postgres サービスの再起動が必要になるリストの順序付けの Ansible の処理の制限を修正しました。	

次のページに続く

表 14 – 前のページからの続き

Description	Addresses
Patroni の構成時に不要なレプリケーションスロットが作成される問題を修正しました。 Patroni クラスターをセットアップするときに、etcd 専用ノードごとにレプリケーションスロットが作成されました。未使用のスロットにより WAL が蓄積されるため、これにより問題が発生しました。スロットは DB サーバー用によりのみ作成されるようになりました。	
<code>patronictl switchover</code> コマンドの使用法を修正しました。 TPA は、 <code>patronictl switchover</code> コマンドを使用する場合、非推奨の <code>--master</code> パラメーターの代わりに <code>--leader</code> を正しく使用するようになりました。	
PEM を含む BDR-Always-ON クラスターのインスタンスの COUNT に関する問題を修正しました。特定の状況で、TPA が PEM サーバーを使用した BDR-Always-ON クラスターのインスタンス数を誤って計算し、「tpaexec configuration」が「StopIteration」で失敗するバグを修正しました。	
PEM サーバーが EFM クラスターのクラスタープライマリとして誤って選択される問題を修正しました。クラスターが pem サーバーで作成された場合、そのバックエンドは EFM によって監視されないため、クラスター全体の Postgres プライマリを検出するときにそのノードを含めないでください。この動作は、TPA によって正しく実装されるようになりました。	45279
.pgpass ファイルの重複行を修正しました。 'tpaexecdeploy' を再実行すると、同じユーザーの.pgpass に余分な行が追加されるバグを修正しました。	

1.2.6 Trusted Postgres Architect 23.36.0 release notes

リリース日 2025 年 2 月 19 日

Trusted Postgres Architect 23.36.0 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

ハイライト

- Ubuntu 24.04 および SLES 15 SP6 のサポート
- すべてのクラスターコンポーネントのバージョン指定子
- EDB Failover ManagerEFM 5 のサポートの向上

機能強化

Description	Addresses
すべてのクラスターコンポーネントのパッケージバージョン指定子のサポートを追加しました。次のソフトウェアパッケージは、 <code>tpaexec configure`コマンドへの--xxx-package-version`オプションを受け入れるようになりました。これは、生成された`config.yml` - barman - pgbouncer - beacon-agent - etcd - patroni - pem-server - pem-agent - pg_backup_api - pgd_proxy - pgdcli - repmgr に`xxx_package_version`を設定します。</code>	
Ubuntu Noble 24.04 のサポートを追加しました。TPA は Ubuntu 24.04 で実行され、この OS をターゲットホストとしてサポートします。 Ubuntu 24.04 の EDB パッケージは、2025 年 3 月末に一般提供されます。その前に、EDB パッケージを必要とする展開は失敗する可能性があります。	
EFM 5 の「auto resume」プロパティのサポートを追加しました。EFM 5.0 以降、 <code>auto.resume.period`プロパティは2つのプロパティに分割されています。1つは起動ケース用、もう1つはdb障害ケース用です。この変更により、使用されている`efm_version`に基づいて、正しいプロパティが追加されます。</code>	
EFM 5 'backup.wal' プロパティのサポートを追加しました。EFM 5.0 から、新しいプロパティ backup.wal が追加されました。この変更により、EFM のバージョンが5以上の場合、新しいプロパティが追加されます。	
SLES 15 SP6 のサポートを追加しました。構成時に SLES が要求されると、TPA は SLES 15 SP6 をインストールします。Docker および EC2 イメージは SP6 になり、systemd-sysvcompat パッケージは SLES にインストールされるため、ローカルブート時スクリプトは引き続き機能します。	

の変更

Description	Addresses
RHEL 同様のシステムで PGDG の最新の Barman を使用します。TPA は、壊れたパッケージの回避策として、RHEL 同様のシステムに PGDG からインストールする場合、以前はデフォルトの Barman 3.9 を使用していました。より新しい barman パッケージは OK であるため、yum に最新のパッケージをインストールさせます。	
RedHat および Debian のセットアップスクリプトを介して EDB リポジトリをセットアップします。この変更により、TPA ノードでの EDB リポジトリのセットアップが簡素化され、dnf、yum、および`apt`パッケージマネージャーの単一のタスクになります。これにより、EDB リポジトリをターゲットホストに追加する際の予期しない HTTP エラーの発生が減少します。	
<code>efm_conf_settings`のドキュメントを修正しました。以前、ドキュメントには*「TPA で認識されるかどうかにかかわらず、efm_conf_settings を使用してパラメーターを設定できます。必要に応じて、efm.properties に表示されるとおりに値を正確に引用する必要があります。ただし、`efm.properties.j2`テンプレートは次の値を使用します`efm_conf_settings`からAnsible 辞書として変換するため、エントリは`key: value`フォームで記述する必要があります。`yaml cluster_vars: efm_conf_settings: notification.level: WARNING ping.server.ip: <well known address in the network></code>	

次のページに続く

表 16 – 前のページからの続き

Description	Addresses
Python の依存関係をバージョン 3.12 にバンプします。TPA では Python 3.12 が必要になりました。EDB Repos 2.0 から TPA をインストールすると、適切な Python インタープリターが自動的にインストールされます。	
PGD のアップグレード中にプロトコルバージョンの更新を待ちます。PGD3 から PGD5 へのアップグレード中に、プロトコルバージョンの更新に時間がかかる場合があります。PGD 5 特定の構成変更は、プロトコルバージョンが変更される前に試行されると失敗します。このような障害を回避するために待機を追加しました。	
新しいノードが既存の EFM クラスターに追加されるときに、 <code><cluster-name>.nodes</code> を更新します。TPA の以前のバージョンでは、新しい EFM ノードが <code>config.yml</code> に追加されると、クラスター内の既存の EFM ノードの <code>Allowed node host list</code> にリストされませんでした。 <code>efm upgrade-conf</code> を実行し、 <code>/raw/<clustername>.properties</code> および <code>/raw/<clustername>.nodes</code> から変更を伝播するタスクは、これらのファイルのいずれかが変更されたときに実行されるようになりました。これにより、新しい EFM ノードが <code><clustername>.nodes</code> ファイルに書き込まれ、 <code>efm cluster-status</code> が <code>Allowed node host list</code> に含めます。	
postgres ユーザーの <code>.bashrc</code> ファイルに <code>PGPORT</code> を含めます。PGPORT 環境変数は、postgres ユーザーの <code>.bashrc</code> ファイルの一部としてエクスポートされます。選択した <code>postgres_flavour</code> 、または構成ファイルで指定されている場合は <code>postgres_port</code> が使用するポート値がデフォルトになります。	

バグ修正

Description	Addresses
データベースが構成どおりではなくデフォルトのパラメーターで作成される問題を修正しました。TPA の以前のバージョンでは、それを必要としない拡張機能別名「モジュール」の <code>CREATE EXTENSION</code> ステートメントを抑制するために、 <code>postgres_databases</code> ハッシュ全体が変更されました。これは、データベース <code>name</code> と <code>extensions</code> のリストのみを含む新しいハッシュが作成され、データベースの他のすべての構成設定を無視するためバグが発生しました <code>owner</code> 、 <code>template</code> 、 <code>encoding</code> など。これにより、データベースは構成どおりではなくデフォルトのパラメーターで作成されることになりました。これを修正するには、拡張機能のリストからモジュールが削除され、結果のリストが <code>CREATE EXTENSION</code> を実行するタスクに渡されます。	44539
Patroni クラスターでの <code>shared_preload_libraries</code> の問題を修正しました。 <code>shared_preload</code> ライブラリのエントリが Patroni によって正しく処理されるようになりました。これにより、 <code>config.yml</code> を介して pglogical 拡張機能を Patroni クラスターに追加すると失敗するバグが修正されます。	

次のページに続く

表 17 – 前のページからの続き

Description	Addresses
拡張機能用の追加のリポジトリセットアップコマンドがないディストリビューションの展開が失敗するバグを修正しました。TPA の以前のバージョンでは、 <code>`postgres`</code> が <code>`postgres_databases`</code> のデータベースの <code>`extra_postgres_extensions`</code> または <code>`extensions`</code> リストに追加されると、 <code>`repository_setup_commands`</code> のリストが空であったため、Debian、SLES、および Ubuntu での展開は失敗していました。RHEL のみが <code>run`crb enable`</code> への追加コマンドを持っています。この空のリストは <code>`command`</code> モジュールに渡され、 <code>`no command given`</code> で失敗し、展開が失敗しました。ディストリビューションに実行する追加コマンドがない場合、 <code>`Automatically run additional repository setup commands for recognized extensions`</code> タスクがスキップされるようになりました。	
<code>`repmgr_use_slot`</code> が 0 に設定されている場合でも、TPA がレプリケーションスロットを作成しようとした問題を修正しました	41776
<code>`pgd_proxy_dsn_attributes`</code> および <code>`pgd_cli_dsn_attributes`</code> のデフォルトとして <code>`bdr_client_dsn_attributes`</code> を設定します。pgd-proxy および pgd-cli は Go で書かれ、Go ドライバーを使用するため、libpq がサポートするパラメーターキーワードの完全なセットをサポートしていません。クラスターが pgd-proxy および/または pgd-cli をインストールし、Go ドライバーがサポートしていないパラメーターで <code>`bdr_client_dsn_attributes`</code> を構成している場合、2つの新しい構成変数 <code>`pgd_proxy_dsn_attributes`</code> および <code>`pgd_cli_dsn_attributes`</code> を含める必要があります。Go ドライバーがサポートしています。逆に、pgd-proxy および pgd-cli がインストールされ、 <code>`bdr_client_dsn_attributes`</code> が Go 非互換のパラメーターを含まない場合、これらのツールの接続文字列は <code>`bdr_client_dsn_attributes`</code> の属性で構成されます。これにより、ユーザーが <code>`bdr_client_dsn_attributes`</code> のパラメーターが使用されることを想定していても、 <code>`pgd_proxy_dsn_attributes`</code> および <code>`pgd_cli_dsn_attributes`</code> が定義されていない場合に空の文字列がデフォルトであった予期せぬ動作が修正されました。	44819
etcd のホームディレクトリの標準形式を使用します。etcd ユーザーを作成するときに、TPA は後続のスラッシュなしでホームディレクトリを参照するようになりました、 <code>`useradd`</code> のような他のツールの使用法と一致します。	
<code>cluster_ssh_port`</code> 設定が Barman によって尊重されない問題を修正しました。Barman 構成は、 <code>`config.yml`</code> の <code>`cluster_ssh_port`</code> を介して設定されたカスタム ssh ポートを使用できるようになりました。設定されていない場合はデフォルトの 22 です。 <code>`-p / --port`</code> フラグは、それぞれ <code>`barman.d.conf`</code> および <code>`barman-wal-restore`</code> / <code>`barman-wal-archive`</code> コマンドの <code>`ssh`</code> コマンドに含まれるようになりました。	

1.2.7 Trusted Postgres Architect 23.35.0 release notes

リリース日 2024 年 11 月 25 日

Trusted Postgres Architect 23.35.0 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

ハイライト

- STIG / CIS コンプライアンスのためのオプション。
- PGD 軽量アーキテクチャのサポート
- Postgis は認識される拡張機能になりました。
- Docker configure は、静的 IP アドレスを使用して名前付けネットワークを作成します。
- ARM アーキテクチャ用の RedHat Enterprise Linux 9 のサポート。
- PostgreSQL、EDB Postgres Extended、および EDB Postgres Advanced Server 17 のサポート。

機能強化

Description	Addresses
STIG / CIS コンプライアンスをサポートします。TPA は、STIG および CIS セキュリティ標準の多くの要件に準拠するように構成されたクラスターを作成するコマンドラインオプションをサポートするようになりました。これらのオプションにより、TPA は、関連する標準で定義されているように postgresql.conf 設定を設定し、必要な拡張機能をインストールし、ファイルシステムの権限やユーザー接続制限などのシステム動作の他の側面を構成し、FIPS 暗号標準などの他の要件をチェックします。TPA は直接課すことはできません。このように生成されたクラスターは、標準に適合することは TPA によって認定されていませんが、適合するクラスターを作成する基礎作業の多くが自動化されています。 PGD Lightweight アーキテクチャのサポートを追加します。TPA は、物理レプリケーションからの移行を容易にするように設計された 2 つの場所の 3 つのノードプライマリに 2 ノード、ディザスターリカバリーに 1 つで構成される PGD Lightweight アーキテクチャを生成できるようになりました。ユーザーは `tpaexec configure lw -a Lightweight --postgresql 15` を実行できるようになりました。 `configure` に Docker でユーザー定義のネットワークを作成してもらいます。Docker が選択されたプラットフォームである場合、configure コマンドは、名前付けネットワークと静的 IP アドレスを config.yml に自動的に追加します。ネットワーク名はクラスター名と同じであり、アドレス範囲は --network オプションの既存のセマンティクスに従いますが、場所ごとに 1 つではなくクラスター全体で 1 つのサブネットのみが使用されることを除きます。ユーザーがサブネットプレフィックスを指定しない場合、TPA はプレフィックスを選択しようとします。その結果、クラスター全体に適合する十分な大きさのサブネットが得られます。config.yml で名前付けネットワークが指定されている限り、キー `ip_address` を使用して Docker インスタンスの静的 IP を指定できるようになりました。	

次のページに続く

表 18 – 前のページからの続き

Description	Addresses
新しいクラスターのバックアップノードとして既存の Barman ノードを使用するための実験的なサポートを追加しました。既存の Barman ノードを新しいクラスターのバックアップノードとして使用する場合、ユーザーはプラットフォームを `bare` に設定し、ベアインスタンスに通常どおり提供されるその他の情報を使用して、Barman インスタンスの vars に `barman_shared: true` を設定できます。Barman ユーザーは既に以前の展開からプロセスを実行しているため、この変更により、TPA は usermod の問題が原因で失敗するいくつかの構成手順をスキップできます。共有 Barman インスタンスはベアインスタンスとして扱われるため、ターゲット PostgreSQL インスタンスへの Barman ユーザーのアクセスを含む必要なアクセスが既に設定されている必要があります。Barman ユーザーのキーを元のクラスターから新しいクラスターにコピーすることは、これを実現するために使用できます。詳細については、TPA ドキュメントの Barman セクションを参照してください。	
`postgis` を認識される拡張機能のリストに追加します。PostGIS パッケージは、ユーザーが `postgres_extensions` または `postgres_databases` の下で名前付けた拡張機能のリストのエントリとして `postgis` を指定すると、自動的に追加されます。また、RHEL 互換ディストリビューションの CRB Code Ready Builder リポジトリを有効にして、PostGIS 依存関係をインストールできるようにします。	
PEM エージェントが EFM ノードに登録されているときに EFM プロローブを有効にします。`--efm-install-path` および `--efm-cluster-name` フラグは、PEM サーバーが EFM ノードに登録されるときに設定されます。`Streaming Replication`、`Failover Manager Node Status`、および `Failover Manager Cluster Info` プロローブは、PEM エージェントが EFM ノードに登録されているときに有効になります。	
ARM アーキテクチャ向けの RedHat Enterprise Linux 9 をサポートします。パッケージは RHEL 9 ARM64 をターゲットとして公開され、TPA はこのアーキテクチャと OS を使用した展開をサポートしています。また、サポートされる AWS イメージのリストを更新して、Amazon が提供する RedHat 9 ARM64 AMI を含めます。ARM64 EC2 インスタンスのデフォルトの `instance_type` が `a1` から `t4g` に更新されました。これは、バースト可能な汎用ワークロードで使用可能な現行世代のプロセッサです。	
PostgreSQL、EDB Postgres Extended、および EDB Postgres Advanced Server 17 をサポート。PostgreSQL、EDB Postgres Extended、および EDB Postgres Advanced Server バージョン 17 を使用するようにクラスターを構成できます。Barman は、postgres サーバーパッケージをインストールして使用するときに `pg_receivewal` バイナリを取得する必要がなくなりました。EDB Postgres Advanced Server 17 または EDB Postgres Extended 17。バイナリがこれらのバージョンのクライアントパッケージに追加されているため。TPA は、クラスターが failover_manager として `repmgr` で構成されている場合、Postgres 17 で使用できないため、アーキテクチャエラーが発生します。サポートされているバージョンを反映するようにドキュメントを更新しました。	

次のページに続く

表 18 – 前のページからの続き

Description	Addresses
<i>efm</i> Postgres の <code>`password_encryption`</code> アルゴリズムをユーザー設定可能にします。構成可能な <code>`efm_user_password_encryption`</code> 変数を公開します。これは、ユーザー要件に応じて <code>`md5`</code> または <code>`scram-sha-256`</code> のいずれかに設定する必要があります。これは、<code>`pg_hba.conf`</code> の <i>efm</i> Postgres ユーザーの <code>`auth-method`</code> と、その暗号化されたパスワードを生成するために使用されるアルゴリズムを制御します。<i>stig`</i> に構成された <code>`compliance`</code> で展開されたクラスターでは、<i>FIPS</i> 対応オペレーティングシステムでは <code>`md5`</code> の使用が許可されていないため、<code>`pg_hba.conf`</code> の <i>efm`</i> Postgres ユーザーの <code>`auth-method`</code> は <code>`scram-sha-256`</code> に設定されます。 ホスト名とともに複数のアドレスを指定できるようにします。<i>tpaexec configure`</i> に <code>--hostnames-from`</code> オプションを使用する場合、各行に 2 つの IP アドレスを含めることができるようになりました。これらは、生成された <code>config.yml</code> に <code>public_ip</code> および <code>private_ip</code> として含まれます。	

の変更

Description	Addresses
postgres.service.j2 テンプレートの非推奨の <code>`PermissionStartOnly`</code> を削除します。<code>`PermissionsStartOnly`</code> は非推奨であり、<code>`ExecStartPost=+/bin/bash...`</code> 構文を介して実現されます <i>barman</i> Postgres ユーザーはスーパーユーザーではなくなりました。特定の必要な特権が Postgres ロール <code>`barman_role`</code> に付与され、これが <code>`barman`</code> Postgres ユーザーに付与されます。これにより、スーパーユーザーとして <code>`barman`</code> ユーザーを作成する必要がなくなります。このロールは、<code>`postgres/createuser`</code> を使用して <code>`granted_roles`</code> リストに追加することにより、他の Postgres ユーザーに付与することもできます。<code>`barman_role`</code> は、Barman タスクの一部として作成されます。Barman を使用しない場合、このロールは作成されません。したがって、このロールに特権を付与するタスクは、<code>`barman_role`</code> ユーザー名が作成される Postgres ユーザーのリストにある場合にのみ実行されます。<code>`barman`</code> ユーザーは、ロール属性として明示的に指定された <code>`NOSUPERUSER`</code> を持つようになりました。クラスターが以前の TPA バージョンスーパーユーザーとして作成されたバーマンユーザーで展開された場合、このバージョンで展開すると <code>`barman`</code> ユーザーから <code>`superuser`</code> ロール属性を削除します。 HARP で <code>etcd</code> を使用する場合、新しいオプション <code>`harp_local_etcd_only`</code> を追加します。HARP で <code>etcd</code> を使用する場合に使用可能な新しいオプションの <code>var`</code> <code>`harp_local_etcd_only`</code> を追加します。このオプションは、HARP マネージャーにローカル <code>etcd</code> ノードに接続するように指示します。この推奨事項は、コンセンサスプロトコルとして <code>`bdr`</code> が使用されるときに同じことを行うことにより学習されたベストプラクティスに従います。複数のエンドポイントを追加するデフォルトモードでは、場合によってパフォーマンスの問題が発生する可能性があります。このオプションは、ユーザーにさらなる制御を与えるために追加されます。	

次のページに続く

表 19 – 前のページからの続き

Description	Addresses
postgres-monitor スクリプトを改善します。 postgres-monitor スクリプトを改善して回復可能なエラーをより適切に管理し、ネットワークエラーの再試行を追加して、postgres サービスが完全に開始するための十分な時間を許可できなかった場合に障害を返さないようにします。	
`efm` ロールを持つノードのみをクラスター `efm.nodes` ファイルに追加します。以前は、`pemserver` および `barman` ノードは、EFM 機能に関連しない場合、EFM の `Allowed node host list` に追加されました。ロールのリストに `efm` があるノードのみを含めるように `efm.node` 構成を書き込むタスクをリファクタリングしました。	

バグ修正

Description	Addresses
pgd-proxy 構成検証の tpaexec テストを修正します。現在の pgd-proxy 構成が予想される構成と一致することを確認するテストが、バージョン<5.5.0 では失敗するバグを修正しました。この修正により、TPA はバージョン 5.5.0 で追加された構成キーを照会しようとしません。	
EFM 互換性のために追加された `primary_slot_name` が `bdr_init_physical` と干渉する場合を修正します。 `primary_slot_name` は、プライマリノードで構成されて、古いプライマリが EFM スイッチオーバー中にレプリケーションに物理スロットを使用するようにします。ただし、`bdr_init_physical` はノードの初期化にそれを使用しようとしませんが、スロットが PGD インストールに存在しないため、無期限にハングします。 `failover_manager` が EFM の場合、この `primary_slot_name` は条件付きで明示的に設定されるようになり、不必要な設定を回避します。	
正しい `bash-completion` パッケージバージョンをダウンロードします。 `config.yml` で `pgdcli_package_version` が指定されている場合、 `packages_for` フィルタは誤って `pgdcli_package_version` をパッケージ名に追加するため、 `bash-completion` パッケージの名前は誤って命名されます。これにより、存在しないパッケージをダウンロードしようとしています。 `bash-completion` パッケージのバージョンは `pgdcli_package_version` から独立しているため、 `packages_for` フィルタの後のリストに追加されます。	
場合によっては、タスクが成功した後でもエラーメッセージが繰り返される問題を修正します。 TPA は、各タスクの後にエラーメッセージスタックをクリアして、メッセージが誤って繰り返されないように保証	
Patroni クラスターへのレプリカの追加を妨げた問題を修正します。 Patroni クラスターの新しいレプリカがレプリケーションスロットに関連するエラーで失敗する問題を修正しました。	

次のページに続く

表 20 – 前のページからの続き

Description	Addresses
M1 アーキテクチャの場合、barman ノードに `pem-agent` ロールを最大 1 回追加します。`--enable-pem` および `--enable-pg-backup-api` が `tpaexec configure` に渡される場合、`witness` でもある場合、`pem-agent` は `barman` ノードに 2 回追加されます。両方の `if` ステートメントを統合して、条件を 1 回のみ評価することにより修正しました。	
`pem_python_executable` を `pkg` ロールの外部に設定します。ユーザーが `pkg` セレクターを除外すると、`pem_python_executable` ファクトが設定されていないために後の PEM 関連タスクが失敗するバグを修正しました。	

1.2.8 Trusted Postgres Architect 23.34.1 release notes

リリース日 2024 年 9 月 9 日

Trusted Postgres Architect 23.34.1 は、次の問題を解決するバグ修正リリースです。

Type	Description
Bug Fix	`efm-witness` ロールを持つノードのスイッチオーバーが失敗した後に deploy を実行する問題を修正しました。EFM ノードの `upstream-primary` は、Postgres から収集されたファクトを使用して決定されます。これは、以前は Postgres がないため、`efm-witness` ロールを持つノードでは失敗していました。上流プライマリを決定するタスクは、`primary` または `replica` ロールを持つノードでのみ実行されます。

1.2.9 Trusted Postgres Architect 23.34 release notes

リリース日 2024 年 8 月 22 日

Trusted Postgres Architect 23.34 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
Enhancement	TPA には、efm がインストールされ、構成ディレクトリとユーザーが作成された後、efm が構成される前に実行される `efm-pre-config` フックが追加されました。カスタム efm ヘルパースクリプトのインストールに使用できます。
Enhancement	TPA には、タスクごとに 1 行の情報を表示する独自の出力プラグインがあり、1 行でも情報のないタスクを省略します。線は、TPA の制御フローを可視できるようにインデントされており、成功、スキップ、および無視されたホストの色分けされた数が含まれています。より完全な出力は、ご使用の環境で TPA_USE_DEFAULT_OUTPUT=true に設定するか、コマンドラインに -v スイッチを追加することにより、オンにできます。

次のページに続く

表 22 – 前のページからの続き

Type	Description
Enhancement	TPA では、PEM エージェントを登録するときに、 <code>cluster_vars</code> の <code>pem-agent-registration_opts</code> の下にリストすることにより追加オプションを指定できるようになりました。
Enhancement	<code>cluster_vars</code> または <code>pem-server</code> インスタンス vars <code>pem_server_ssl_certificate</code> および <code>pem_server_ssl_key</code> の下の <code>config.yml</code> に PEM サーバーで使用する証明書とキーペアの名前を含めることにより、PEM サーバーで使用する独自の Web サーバー証明書を提供できるようになりました。TPA は、クラスターディレクトリの <code>ssl/pemserver</code> ディレクトリから PEM サーバーにそれらをコピーし、それに応じて Apache/httpd を構成します。
Enhancement	TPA は、新しいクラスター展開で EFM <code>upgrade-conf</code> コマンドを実行して、 <code><cluster_name>.properties</code> ファイルと <code><cluster_name>.nodes</code> ファイルの両方に追加されるコメントとインラインドキュメントを活用するようになりました。
Enhancement	TPA は、 <code>bind.address</code> 値の IP アドレスの代わりにホスト名解決を使用して EFM クラスターをセットアップできるようになりました。これは、 <code>configure</code> コマンドの <code>--efm-bind-by-hostname</code> オプションまたは <code>config.yml</code> の <code>cluster_vars</code> セクションの <code>efm_bind_by_hostname: true/false</code> で呼び出すことができます。
Enhancement	TPA は、EFM 4.9 で追加された EFM プロパティ <code>enable.stop.cluster: boolean, default true</code> 、 <code>priority.standbys: default "</code> 、 <code>detach.on.agent.failure: boolean, default false</code> 、 <code>pid.dir: default "</code> の設定をサポートするようになりました。
Enhancement	TPA は、pgBouncer から Postgres への接続に <code>cert</code> 認証を使用するように pgBouncer を構成できるようになりました。これは、 <code>cluster_vars</code> で <code>pg-bouncer_use_cert_authentication</code> を <code>true</code> に設定することにより有効にします。有効にすると、pgBouncer に接続するユーザーの認証方法も <code>md5</code> から <code>scram-sha-256</code> に変更されます。
Change	TPA は、AWS での RAID 作成をサポートしなくなりました。
Change	<code>upstream_primary</code> を解決するための EFM 依存関係を削除しました。以前は、スイッチオーバー後の展開で、現在のプライマリの EFM が照会されていました。EFM が実行されていない場合、これは失敗します。Postgres を介して収集された <code>cluster_facts</code> は、スイッチオーバー後の現在のプライマリを決定するために使用され、EFM への依存関係を削除します。
Change	EFM クラスターでは、スイッチオーバー後に <code>upstream_primary</code> が正しく更新されるようになりました。その結果、レプリカに正しい <code>auto.reconfigure</code> 設定が設定されます。スタンバイが新しいプライマリに従うようになりました。
Bug Fix	Patroni API にアクセスするときに TPA がプロキシ設定を誤って適用する問題を修正しました。Ansible のデフォルトでは、プロキシが定義されている場合、それを使用します。これは、プロキシも Patroni API への呼び出しをインターセプトおよび中断するため、インターネットからパッケージをダウンロードするためにプロキシを必要とするエアギャップ環境のかなり一般的な場合では機能しません。
Bug Fix	TPA がすべてのインスタンスで <code>pemserver</code> インスタンスにのみ適切な PEM エージェントパラメータを設定する問題を修正しました。

次のページに続く

表 22 – 前のページからの続き

Type	Description
Bug Fix	クラスターノードのターゲットとして SLES オペレーティングシステムを使用する場合、pgd-proxy および pgdcli のデフォルトパッケージ名に欠落していたエントリを追加しました。
Bug Fix	TPA が既存のノードで postgres をリロード/再起動して構成変更を再読み取りすることに失敗し、したがって、新しいノードがクラスターに接続できないという問題を修正します。
Bug Fix	レプリカからバックアップを取得する場合、WAL ファイルの待機がタイムアウトになることにより、初期バックアップの取得時にバーマンが失敗する場 合がある問題を修正しました。これは、WAL セグメントのスイッチを強制する前に、バーマンがベースバックアップを完了するのを待機することにより修正されます。
Bug Fix	上流_プライマリ postgres にデフォルト以外の postgres ポートを使用する場合でも、`repmgr witness register` コマンドが正しい postgres_port 値で使 用されていることを確認します。
Bug Fix	PGD クラスター内のサブスクライバー専用ノードとそのレプリカのインスタ ンスレベルで設定されている場合、`repmgr` への failover_manager オーバ ライドが正しく機能しない問題を修正しました。
Bug Fix	誤った cgroup 検出の 2 つのケースを修正しました。MacOSX では、 `/proc/mounts` を読み取ろうとしなくなりました。`/sys/fs/cgroup` が `ro` であるが、その下にマウントが `rw` であるシステムでは、TPA がこれを正しく検 出するようになりました。
Bug Fix	pgd-proxy に新しく追加された `read_listen_port` オプションの pgd-proxy ノ ードで実際の構成セットを検証できることを確認します。
Bug Fix	PEM 9.7.0 での展開を妨げる問題を修正しました。PEM 9.7.0 はパッケー ジレベルで Apache に依存しないため、Web サーバーとして Apache を使用する には、パッケージを明示的にインストールします。

1.2.10 Trusted Postgres Architect 23.33 release notes

リリース日 2024 年 6 月 24 日

Trusted Postgres Architect 23.33 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
Enhancement	TPA は、ARM64 CPU アーキテクチャで Debian 12 Bookworm をサポートする ようになりました。

[次のページに続く](#)

表 23 – 前のページからの続き

Type	Description
Enhancement	PGD バージョン 5.5 では、プロキシノードを読み取りエンドポイントとして構成でき、読み取り専用クエリをシャドウノードに転送します。TPA は、 <code>config.yml</code> の <code>default_pgd_proxy_options</code> および <code>pgd_proxy_options</code> の下の <code>read_listen_port</code> パラメーターを設定することにより、この構成オプションをサポートします。このパラメーターは、PGD バージョンが 5.5 以上の場合、デフォルトで含まれます。 <code>--proxy-listen-port</code> および <code>proxy-read-listen-port</code> 引数を <code>tpaexec configure</code> コマンドに渡すことにより、ポート番号を指定することもできます。
Enhancement	TPA は、ロール <code>beacon-agent</code> を割り当てるか、 <code>configure</code> とともに <code>--enable-beacon-agent</code> オプションを使用することにより、Postgres ノードでのビーコンエージェントの展開と構成をサポートするようになりました。
Enhancement	Patroni 展開での <code>postgres_wal_dir</code> のサポートを追加しました。TPA 構成でカスタム <code>postgres_wal_dir</code> が指定されている場合、TPA はそのオプションを Patroni 構成ファイルの対応する設定に中継します。このようにして、Patroni が TPA から独自にスタンバイを再構築する必要がある場合、スタンバイはカスタム WAL ディレクトリで適切にセットアップされます。
Enhancement	Patroni クラスターに PgBouncer ノードを追加すると、TPA は <code>on_role_change</code> コールバックで Patroni を構成するようになりました。そのコールバックは、フェールオーバーおよびスイッチオーバーイベントに応答して、PgBouncer ノードのプライマリ接続情報を更新します。
Enhancement	EDB は、PGDG から <code>patroni</code> パッケージをリビルドする代わりに、独自の <code>edb-patroni</code> パッケージを生成するようになりました。TPA では、ユーザーが <code>patroni</code> パッケージと <code>edb-patroni</code> パッケージを選択できるようになりました。選択は、新しい TPA 設定 <code>patroni_package_flavour</code> を介して行われます。
Change	PGDG リポジトリの壊れた Barman 3.10 パッケージをワークアラウンドために、RHEL ファミリーシステムで PGDG リポジトリを使用している場合、TPA は Barman のバージョン 3.9 をインストールするようになりました。この動作は、 <code>config.yml</code> で <code>barman_package_version</code> を明示的に設定することによりオーバーライドできます。
Change	Patroni がフェールオーバーマネージャーの場合、 <code>haproxy_bind_address</code> は <code>0.0.0.0</code> に設定されます。これは、Postgres ノードと HA プロキシノード間の通信を妨げる <code>127.0.0.1</code> の一般的なデフォルトに関する問題を解決します。ユーザーはこの値を、クラスターネットワークに適したより制限的なものに変更する必要があります。
Change	タスクセクターが、展開の最終段階で一貫して適用されるようになりました。テストのタスクセクターの一貫性が向上し、ドキュメントのタスクセクターの例が正しくなりました。すべての展開時フックに、対応するタスクセクターがあります。
Change	<code>barman_package_version</code> が設定されている場合、TPA は <code>barman-cli</code> パッケージおよび Barman 自分自身を検索するときにそれを参照するようになりました。これにより、 <code>barman_package_version</code> が使用されたときにダウンローダーを使用するクラスターで障害が発生した不一致が解決されます。

次のページに続く

表 23 – 前のページからの続き

Type	Description
Bug Fix	BDR データベース内の機能に必要な権限が、監視ノードの HARP DCS ユーザーに付与されていない問題を修正しました。
Bug Fix	「読み取り専用ファイルシステム」で Docker プロビジョニングが失敗する問題を修正しました。最近の OS イメージを実行している docker コンテナで cgroup1 を実行しているホストシステムでは、`tpaexec provision` はコンテナのプロビジョニングに失敗し、「mkdir /sys/fs/cgroup/tpa.scope: read-only file system」のようなエラーメッセージを表示する場合があります。TPA はこのケースを検出して回避します。
Bug Fix	TPA は、プロビジョニング前にユーザーが `tpaexec cmd` または `tpaexec ping` を実行した場合、明確なエラーメッセージを提供するようになりました。

1.2.11 Trusted Postgres Architect 23.32 release notes

リリース日 2024 年 5 月 15 日

Trusted Postgres Architect 23.32 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
Enhancement	M1 アーキテクチャは、 <i>tpaexec configure</i> `への追加引数` <code>--location-names</code> 、 <code>--primary-location</code> 、 <code>--data-nodes-per-location</code> 、 <code>--witness-only-location</code> 、および <code>--single-node-location</code> をサポートするようになりました。これらの引数を組み合わせることで、 <code>config.yml</code> を編集することなく、ほとんどの一般的なレイアウトを指定できます。
Enhancement	TPA は、展開中に chrony をインストールし、Amazon Time Sync サービスをポイントする AWS を除き、すべてでデフォルト構成を維持します。
Enhancement	TPA は、IBM PowerPPC64le で RHEL 8 および 9 をサポートするようになりました。
Enhancement	<i>tpaexec relink</i> `に` <code>--force</code> ` オプションを追加しました。デフォルトでは、再リンクはターゲットファイルが既に存在する場合を変更しません。` <code>--force</code> ` では、再リンクは既存のすべてのターゲットファイルを削除し、それらを再作成します。
Enhancement	TPA が Debian 12 x86 をサポートするようになりました。
Enhancement	<code>pg_failover_slots</code> が認識される拡張機能になりました
Enhancement	<code>sql_profiler</code> 、 <code>edb_wait_states</code> 、および <code>query_advisor</code> 拡張機能が <code>pem-agent</code> ノードに自動的に含まれるようになりました。pem-agent ノードのデフォルトの拡張機能のリストは、 <code>config.yml</code> に <code>pemagent_extensions</code> のリストを含めることによりオーバーライドできます。このリストが空の場合、拡張機能は自動的に含まれません。

次のページに続く

表 24 – 前のページからの続き

Type	Description
Change	TPA は、RHEL 7 を除くすべてのシステムの cgroups 2 を実行しているホストに Docker クラスタをプロビジョニングできるようになりました。新しいシステム RHEL 9 または Ubuntu 22.04 では、TPA は cgroups 2 スコープを使用してホストとコンテナ間の分離を追加します。
Change	AWS AMI バージョンを最新バージョンに更新しました。
Bug Fix	AWS で Debian 10 への展開がメッセージ `The repository 'http://cdn-aws.deb.debian.org/debian buster-backports Release' does not have a Release file` で失敗する問題を修正しました。 debian 10 Buster のバックポートリポジトリは、deb.debian.org で利用できなくなりましたが、標準の AWS AMI はまだそれを参照しているため、apt 操作を試行する前に/etc/apt/sources.list をそれに応じて変更します。
Bug Fix	`assign_public_ip:no` が設定されている場合、AWS での展開が失敗する問題を修正しました。
Bug Fix	barman と postgres のカスタムユーザーを使用しようとしたときに混合エラーを発生させ、展開の失敗を引き起こしたさまざまなロールの問題を修正しました。
Bug Fix	キー `ip_address` を使用して IP アドレスを定義した場合、最初の展開以降の展開が `Unrecognised host=... in primary_conninfo` のようなエラーで失敗する可能性がある問題を修正しました。
Bug Fix	`tpaexec upgrade` がエラーを発生させ、`tpaexec relink` の役に立たない使用メッセージを表示する方法でリリンクスクリプトを呼び出すことができるエラーを修正しました。
Bug Fix	Postgres をリロードするタスクがスキップされ、restore_command オーバーライドが `postgresql.auto.conf` から削除されない問題を修正しました
Bug Fix	ソースから BDR をコンパイルしようとする前に、TPA がソースディレクトリに変更しなかった問題を修正しました。
Bug Fix	指定されたクラスタに必要でない場合でも、TPA が有効な 2ndQuadrant トークンを必要とする問題を修正しました。
Documentation	tower/AAP のドキュメントを更新して、AAP 実行環境の作成に関する手順を含めます。

1.2.12 Trusted Postgres Architect 23.31 release notes

リリース日 2024 年 3 月 19 日

Trusted Postgres Architect 23.31 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
Bug Fix	構文エラーのために展開が失敗する可能性がある重大なバグを修正しました。

1.2.13 Trusted Postgres Architect 23.30 release notes

リリース日 2024 年 3 月 19 日

Trusted Postgres Architect 23.30 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
New Feature	TPA は、Ansible Automation Platform 2.4+ コントローラバージョン 4+ で使用されるカスタムの 'ExecutionEnvironment' イメージを提供するようになりました。このイメージには、AAP を介して展開を実行するために必要なすべてが含まれています。このイメージは、ansible-builder と python-alpine の軽量ベースイメージを使用してビルドされます。
Enhancement	TPA は、拡張機能のサブセットのパッケージ名と共有プリロードライブラリエントリを自動的に追加するようになりました。これらの特定の拡張機能の場合、 <code>`postgres_databases`</code> のデータベースエントリの <code>`extra_postgres_extensions`</code> リストまたは <code>`extensions`</code> リストでは拡張機能名前のみが必要です。
Enhancement	EDB Advanced Storage Pack パッケージと共有プリロードライブラリエントリは、ユーザーが拡張機能として指定し、 <code>`postgres_version`</code> が 15 以上の場合、 <code>`bluefin`</code> に自動的に追加されます。
Enhancement	インスタンスに新しい <code>`provision_only`</code> オプションを追加しました。インスタンスの <code>config.yml</code> に <code>`provision_only: true`</code> がある場合、通常どおりプロビジョニングされますが、 <code>`tpaexec deploy`</code> によって認識されるインベントリには追加されません。
Change	TPA の以前のバージョンでは、ソースノードのデータベース構造を監視ノードに同期していました。これは必要ではなく、同期されたスキーマは使用または更新されません。これが発生するのを防ぐため、TPA は、監視ノードの <code>`bdr.join_node_group()`</code> を呼び出すときに「 <code>synchronize_struction`</code> 」を「 <code>none`</code> 」に明示的に設定するようになりました。
Change	Ansible タグではなくカスタムセクターを使用して、タスクの選択的な実行がサポートされるようになりました。特定のセクター <code>`tpaexec deploy . --included_tasks=barman`</code> に一致するタスクのみを実行するには。特定のセクターに一致するタスクをスキップするには <code>tpaexec deploy . --excluded_tasks=ssh`</code> タスクセクターは、 <code>config.yml</code> で <code>`excluded_tasks`</code> または <code>`included_tasks`</code> 変数を指定して使用できます。
Change	Ansible 2.9 は、コミュニティディストリビューションも 2ndQuadrant フォークもサポートされなくなりました。 <code>tpaexec deploy`</code> に <code>--skip-tags`</code> オプションを使用しているユーザーは、新しい <code>`--excluded_tasks`</code> オプションに移動する必要があります。

1.2.14 Trusted Postgres Architect 23.29 release notes

リリース日 2024 年 2 月 15 日

Trusted Postgres Architect 23.29 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
Enhancement	クラスターボルトのパスワードをシステムキーリングに保存するためのサポートを追加しました。これは、 <code>keyring_backend</code> が <code>system</code> 新しいクラスターのデフォルトに設定されている場合、Python キーリングモジュールを活用して、サポートされているシステムキーリングにボルトパスワードを保存します。この変更は、既存のクラスター、または <code>config.yml</code> で <code>keyring_backend</code> から <code>legacy</code> を設定するクラスターには影響しません。
Enhancement	<code>tpaexec setup --ansible-version</code> 引数は、有効な ansible バージョンとして <code>8</code> または <code>9</code> 、および既存の <code>2q</code> または <code>community</code> 、どちらも ansible 2.9 を暗黙的に受け入れます。デフォルトは <code>8</code> になりました。ansible 9 のサポートは実験的であり、Python 3.10 以降が必要です。
Bug Fix	<code>config.yml</code> で既に定義された <code>edb_repositories</code> が、再構成中に保持されない問題を修正しました。エアギャップ環境での <code>bdr4</code> から <code>pgd5</code> へのアップグレードシナリオを修正します。
Bug Fix	TPA の <code>postgres-monitor</code> は、「データベースシステムはまだ接続を受け入れていません」というメッセージを回復可能なエラーとして認識するようになりました。
Bug Fix	TPA は、アップグレード時にレプリカの <code>postgres/config/final</code> ロールを正しくスキップするようになりました。
Bug Fix	Debian および Ubuntu システムでパッケージダウンローダーを使用する場合、パッケージ名のワイルドカードが尊重されない問題を修正しました。
Bug Fix	ダウンローダーは、Debian および Ubuntu システムでパッケージを取得する前に <code>apt-get update</code> を実行するようになりました。
Bug Fix	PGD クラスターで CAMO が有効になっている場合、TPA はトランザクションストリーミングを無効にするようになりました。
Bug Fix	TPA は、 <code>selinux</code> が有効になっている Barman サーバーを正しく構成するようになりました。

1.2.15 Trusted Postgres Architect 23.28 release notes

リリース日 2024 年 1 月 23 日

Trusted Postgres Architect 23.28 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
Enhancement	新しいオプション <code>`postgres_log_file`</code> を追加しました。このオプションは、Postgres ログファイルを設定します、stderr または syslog を介してロギングするかどうか。デフォルトは <code>'/var/log/postgres/postgres.log'</code> であり、以前にハードコードされた値です。
Enhancement	新しいフック <code>`barman-pre-config`</code> を追加しました。このフックは、Barman がインストールされ、そのユーザーがセットアップされた後、構成される前に呼び出されます。これは、証明書ファイルまたは barman ユーザーの存在を必要とするその他のタスクのインストールに使用できますが、Barman が開始される前に行う必要があります。
Enhancement	<code>`config.yml`</code> の AWS インスタンスのキー <code>`elastic_ip`</code> は、このインスタンスに割り当てるために、既に割り当てられた Elastic IP アドレスに設定できます。
Change	Patroni クラスターでは、TPA は、プライマリのみを設定して Patroni にレプリカをセットアップさせるのではなく、クラスターの制御を Patroni に引き渡す前にレプリカをセットアップするようになりました。
Change	新しいクラスターの場合、TPA は <code>`bdr_superuser`</code> ロールに属する <code>`harp_manager_user`</code> デフォルトでは <code>`harpmanager`</code> の設定で指定されたユーザーを作成し、 <code>`postgres`</code> スーパーユーザーの代わりにこのユーザーとして動作するように HARP Manager を設定します。これは、ユーザが <code>`config.yml`</code> で <code>`harp_manager_user`</code> を明示的に設定することによりこの動作をオーバーライドしない限り、TPA が HARP Manager ユーザーとして <code>`postgres`</code> を使用し続ける既存のクラスターに影響しません。
Bug Fix	TPA が誤って EFM クラスターに repmgr をインストールしようとする問題を修正しました。
Bug Fix	デプロイが成功したにもかかわらず、2q リポジトリに関する警告が表示されると、TPA がゼロ以外の終了コードを返す問題を修正しました。
Bug Fix	TPA は、Debian ファミリーシステムでオフライン使用のためのパッケージをダウンロードするときにワイルドカードを正しく解釈するようになりました。
Bug Fix	PGDG リポジトリからインストールするときに、TPA が repmgr の誤ったパッケージ名を使用しようとする問題を修正しました。
Bug Fix	selinux とカスタム barman ホームディレクトリを使用する場合の barman 接続障害を修正しました。
Bug Fix	TPA は、ディレクトリ名と異なる場合、 <code>`show-password`</code> および <code>`store-password`</code> コマンドで正しいクラスター名を使用するようになりました。
Bug Fix	使用できない 2ndQuadrant リポジトリキーが必要な場合、TPA はきれいにエラーアウトするようになりました。
Bug Fix	<code>--cluster-prefixed-hostnames</code> オプションが使用される場合、TPA はホスト名を正しくサニタイズするようになりました。
Bug Fix	TPA は、ローカルリポジトリを使用してクラスターをアップグレードするときに、パッケージがリモートホストに正しくコピーされるようになりました。

1.2.16 Trusted Postgres Architect 23.27 release notes

リリース日 2023 年 12 月 19 日

Trusted Postgres Architect 23.27 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
Enhancement	TPA は、Docker で Oracle Linux 7、8、および 9 をサポートするようになりました。
Change	TPA は Python 3.9-3.11 を必要とし、適切なインタープリターを提供するように `edb-python-39` パッケージに依存しています。
Change	TPA は、デフォルトで 2ndQuadrant リポジトリを構成せず、代わりに EDB Repos 2.0 から適切なリポジトリを選択します。
Change	TPA は、新しい `--replace-2q-repositories` 引数を `tpaexec reconfigure` に提供するようになりました。これは、既存の config.yml から 2q リポジトリを削除し、クラスターの postgres フレーバーと BDR バージョンに適切な EDB リポジトリを追加します。
Change	TPA は、より多くのオブジェクトにファイルシステムのアクセス許可を明示的に設定するようになりました。
Change	config.yml で新しい変数 `disable_repository_checks` を true に設定して、PGD-Always-ON アーキテクチャを展開するときに EDB リポジトリの通常のチェックをバイパスできます。
Change	TPA は、スイッチオーバーの場合に使用されるプライマリノードでも primary_slot_name を生成して、スイッチされたプライマリが新しいプライマリに物理スロットを持つようにします。
Change	TPA は、メジャー PGD アップグレードの準備をするために tpaexec reconfigure コマンドを実行するときに、古い BDR バージョンの既存の構成オプションを使用して、CAMO 対応パートナーの commit_scope が生成されるようになりました。また、より良いデフォルトも選択します。
Bug fix	postgres 変数が検証ルールによりパトローニによって拒否される問題を修正しました。
Bug fix	ユーザーが `sslmode=verify-full` で単一の `barman_client_dsn_attributes` を設定できない問題を修正しました。
Bug Fix	TPA は、メモリ不足エラーを回避するために、より低いデフォルトの `maintenance_work_mem` を割り当てます。

1.2.17 Trusted Postgres Architect 23.26 release notes

リリース日 2023 年 11 月 30 日

Trusted Postgres Architect 23.26 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
Enhancement	TPA は、Docker プラットフォームで Oracle Linux 9 をサポートするようになりました。
Enhancement	`--cluster-prefixed-hostnames` オプションを `tpaexec configure` に追加しました。これにより、複数の Docker クラスターをホストするマシンでのホスト名の衝突を簡単に回避できます。
Change	Mac OS X で Docker ビルドを有効にするパッケージを追加しました。
Change	クラスター内に複数の PEM サーバーが存在する場合、PEM サーバーで実行されているエージェントはローカルサーバーに登録します。
Change	CAMO を使用した PGD 5 クラスターの場合。TPA は、デフォルトでタイムアウトを 60 秒に設定し、`require_write_lead` を true に設定します。
Bug Fix	PGD 3 クラスターを PGD 5 にアップグレードするときに CAMO 構成が正しくセットアップされない問題を修正しました。アップグレードが CAMO クラスターで完全にサポートされるようになりました。
Bug Fix	HARP ノードをフェンシングまたはアンフェンシングするときに、`bdr_node_name` ではなく `hostname` が使用される問題を修正しました。
Bug Fix	デプロイを抑制するオプションで `deploy` が呼び出されたときに、`provision` が自動的に実行される問題を修正しました。

1.2.18 Trusted Postgres Architect 23.25 release notes

リリース日 2023 年 11 月 14 日

Trusted Postgres Architect 23.25 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
Enhancement	TPA は、PGD 3.7 から PGD 5.3 以上への自動アップグレードをサポートするようになりました。CAMO を使用したクラスターのアップグレードはまだサポートされていないことに注意してください。
Enhancement	TPA が EDB Advanced Server 16 および EDB Extended Server 16 をサポートするようになりました。
Change	PGD 4 から PGD 5 へのアップグレードで導入されたアップグレードプロセスのさまざまな改善が、BDR-Always-ON アップグレードにバックポートされました。
Change	TPA は、SLES への PEM のインストールをサポートするようになりました。
Change	TPA は、一部のファイルシステムオブジェクトを作成するときにアクセス許可を明示的に設定するようになりました。これは、将来のリリースではすべてのファイルシステムオブジェクトに拡張される予定です。
Change	TPA は v1 の pgd-cli 構成ファイルにシンボリックリンクを追加するため、`-f` スイッチを介してパスを指定せずに実行できるようになりました。
Change	TPA は `alter_node_kind` PGD ファンクションを呼び出して、BDR バージョン 4.3 以降を使用する BDR -Always-ON クラスターのノード種類が正しく設定されていることを確認します。

[次のページに続く](#)

表 31 – 前のページからの続き

Type	Description
Change	今後のデフォルトのクラスター構成は、SLES 15 が要求されたときに SLES 15 SP5 を選択するようになりました以前の SP4。
Bug Fix	tpaexec-deps ユーザーの `tpaexec setup` コマンド中にチェックサムエラーが発生する問題を修正しました。
Bug Fix	pemworker が呼び出されたときに `pem_server_group` が正しく適用されず、サーバーが PEM で予想通りにグループ化されなかった問題を修正しました。
Bug Fix	`sar` が Barman インスタンス以外のインスタンスで実行されるようにスケジューリングされていない sys / sysstat ロールの問題を修正しました。

1.2.19 Trusted Postgres Architect 23.24 release notes

リリース日 2023 年 10 月 17 日

2ndQuadrant Ansible フォークの使用のサポートは、2024 年 4 月に TPA から削除され、GitHub リポジトリはアーカイブされます。

Community Ansible に切り替える必要があります。これがデフォルトになりました。大部分のユーザーにとって、この変更は透過的です。

2ndQuadrant Ansible で `--skip-tags` を使用している場合、これは TPA およびコミュニティ Ansible ではサポートされていないことに注意してください。2ndQuadrant Ansible が削除される前に、Community Ansible と互換性のある `--skip-tags` の代替を提供する予定です。

Type	Description
Change	<code>tpaexec setup</code> は、2ndQuadrant Ansible ではなくコミュニティ Ansible を使用することがデフォルトになりました。オプション <code>--use-2q-ansible</code> は、2ndQuadrant ansible の使用を強制するために使用できます。これは現在非推奨であり、将来のリリースで削除されます。 <code>--skip-tags</code> を使用している場合、 :ref: `Configuring lvm-direct for production <TPA installation> を参照してください。
Change	config.yml の <code>edb_repositories</code> からリポジトリが削除されている場合、 <code>tpaexec deploy</code> はノードから削除するようになりました。
Change	TPA は、harp-proxy と harp-manager が同じノードで実行されている場合を検出し、harp-proxy に別の構成ファイルを使用します。
Change	<code>upgrade</code> コマンドは、ターゲットインスタンスのローカルリポジトリを更新するようになりました。
Bug Fix	TPA が pg_basebackup 呼び出しの <code>postgres_wal_dir</code> を尊重しない問題を修正しました
Bug Fix	TPA は、PGD クラスター内のサブスクライバー専用ノードのフェールオーバーマネージャーとして repmgr を受け入れ、このようなノードの物理レプリケーションを許可します。
Bug Fix	TPA による Ubuntu 22.04 Docker イメージのビルドを妨げたタイプミスを修正しました。

次のページに続く

表 32 – 前のページからの続き

Type	Description
Bug Fix	TPA は、構成時に BDR-Always-ON アーキテクチャ、EDB Postgres Extended フレーバー、および PEM のサポートされていない組み合わせを拒否するようになりました。

1.2.20 Trusted Postgres Architect 23.23 release notes

リリース日 2023 年 9 月 21 日

Trusted Postgres Architect 23.23 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
Enhancement	TPA は PostgreSQL 16 をサポートするようになりました。PostgreSQL 16 パッケージは、サポートされているすべてのリポジトリでまだ利用できないことに注意してください。したがって、これに該当するまで、すべての構成が機能するわけではありません。
Change	Postgres 16 以降が選択されている場合、TPA はデフォルトで 2ndQuadrant リポジトリを追加しません。この場合、TPA は <code>`tpa_2q_repositories: []`</code> を明示的に設定します。
Change	EFM は、利用可能なプラットフォームでデフォルトで JDK 11 を使用するよう構成されました。
Change	EDB リポジトリが使用されていない場合、TPA は PGDG からパッケージを除外しません。以前は Barman と psycopg2 が除外されていました。
Change	SLES へのインストールをサポートするために etcd および Patroni のパッケージ名を追加しました。
Bug Fix	PEM サーバーの Apache HTTPD サービスがブート時に開始されない問題を修正しました。
Bug Fix	不正な権限で <code>pg_backup_api</code> テストが実行され、失敗する問題を修正しました。
Bug Fix	<code>pg_backup_api</code> の Apache HTTPD サービスがブート時に開始されない問題を修正しました。
Bug Fix	<code>`bdr.standby_slot_names`</code> および <code>`bdr.standby_slots_min_confirmed`</code> チェックが bdr3 クラスターで誤ったスキーマを使用した問題を修正しました。
Bug Fix	拡張機能の構成キーが間違った形式でパトローニに渡され、「警告 予期しないパラメーターの削除」が発生する問題を修正しました。
Bug Fix	<code>`docker_images`</code> の中間ベースイメージオプションを使用した場合、結果のイメージ名が正しくない問題を修正しました。

1.2.21 Trusted Postgres Architect 23.22 release notes

リリース日 2023 年 9 月 6 日

Trusted Postgres Architect 23.22 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
Change	TPA はオープンソースプロジェクトになりました。 GitHub から GPLv3 ライセンスに基づいてソースのクローンを作成できます。

1.2.22 Trusted Postgres Architect 23.21 release notes

リリース日 2023 年 9 月 5 日

Trusted Postgres Architect 23.21 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
Change	デフォルトの M1 構成は、EDB ソフトウェアが選択されている場合は EDB Repos 2.0 を使用し、それ以外の場合は PGDG が使用されます。これは、新しいクラスターにのみ影響します。
Change	M1 アーキテクチャで `tpaexec configure` を実行する場合、フェールオーバーマネージャーを明示的に選択する必要があります。
Bug fix	TPA が誤って上流インスタンスではなくレプリカに 'subscriber-only' を設定することを必要とした、PGD サブスクライバ専用ノードの作成に関する問題を修正しました。
Bug fix	TPA は、'bare' プラットフォームオプションを使用している場合でも、コンテナに展開するときに適用できないタスクをスキップするようになりました。以前は、これらは 'docker' が選択された場合にのみスキップされました。
Bug fix	`/etc/edb` の権限に関する問題を修正しました。展開された PGD5 クラスターのデータノードに pgd-proxy ロールを追加すると、pgd-proxy-config.yml を開く権限がないため pgd-proxy の起動に失敗します。
Bug fix	厳密な umask が設定されている場合、`/var/log/postgres` が不適切な権限 0600 になる可能性がある問題を修正しました
Bug fix	PEM に正しく登録された Barman インスタンスで `tpaexec deploy` を繰り返すと、PEM エージェントの Barman 構成が失われる問題を修正しました。

1.2.23 Trusted Postgres Architect 23.20 release notes

リリース日 2023 年 8 月 1 日

Trusted Postgres Architect 23.20 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
New feature	TPA は、新しいコマンド <code>tpaexec reconfigure</code> を実行して変更された <code>config.yml</code> を生成し、 <code>tpaexec upgrade</code> を実行してアップグレードを実行することにより、PGD 4 から PGD 5 へのアップグレードをサポートするようになりました。
Enhancement	TPA インストールでチェックサムを実行し、パッケージで配布されたものと一致することを確認する新しいサブコマンド <code>tpaexec info validate</code> を追加しました。
Change	<code>update-postgres</code> コマンドは、より一般的な <code>upgrade</code> コマンドに置き換えられました。
Change	RHEL 8 または 9 に Failover Manager の OpenJDK をインストールするときに、TPA は <code>tzdata-java</code> を明示的に追加するようになりました。これは this OpenJDK bug の回避策です。
Change	TPA は、AWS で利用可能な最新の Debian AMI このリリースの時点で最新を使用するようになりました。
Change	<code>config.yml</code> が変更された場合、TPA は <code>tpaexec deploy</code> または <code>tpaexec upgrade</code> の一部として <code>tpaexec provision</code> を自動的に実行するようになりました。
Bug fix	pgd-proxy ノードで pgd-cli を実行しているときに TPA が存在しないユーザーを使用しようとするバグを修正しました。
Bug fix	<code>tpaexec relink</code> による変更が Git リポジトリに正しくコミットされないバグを修正しました。

1.2.24 Trusted Postgres Architect 23.19 release notes

リリース日 2023 年 7 月 12 日

Trusted Postgres Architect 23.19 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
New feature	TPA では、サブスクリバ専用 PGD ノードの物理レプリカを作成できるようになりました。
New feature	TPA は、HTTP(S) HARP および PGD プロキシ正常性プローブの構成をサポートするようになりました。
New feature	TPA では、M1 アーキテクチャを使用したフェールオーバーマネージャーとして Patroni を選択できるようになりました。このサポートは実験的なものであり、運用環境での使用はまだ推奨されていません。
Enhancement	TPA では、常に最新バージョンを使用するのではなく、 <code>edb-pgd-proxy</code> および <code>edb-bdr</code> -ユーティリティの特定のバージョンを設定できるようになりました。
Change	Debian に似たシステムでは、パッケージ選択コードは、該当する場合、特定のパッケージに <code>-dbgsym</code> ではなく <code>-dbg</code> を使用するようになりました。

次のページに続く

表 37 – 前のページからの続き

Type	Description
Change	レプリケーションスロットを構成するときに、TPA は有効な文字のみが <code>primary_slot_name</code> で使用されるようになりました。以前は、TPA はデフォルトとして <code>inventory_hostname</code> を使用していましたが、これにはハイフンを含めることができます。これらはアンダースコアに置き換えられます。
Change	デフォルトのフェールオーバーマネージャーのバージョンは 4.7 になりました。
Bug fix	TPA 23.18 で PGD 3.7 から 4 へのアップグレードが失敗する問題を修正しました。
Bug fix	PEM 展開時に TPA が TLS 証明書のコモンネームにアンダースコアを含める問題を修正しました。これは無効であり、一部のプラットフォームでは障害が発生します。
Bug fix	誤った <code>etcd</code> サービス名が Debian 系プラットフォームで使用され、TPA が <code>etcd</code> を起動できない問題を修正しました。
Bug fix	TPA が RHEL 8 に <code>etcd</code> パッケージをインストールできない問題を修正しました。
Bug fix	<code>configuration</code> 中にメッセージ <code>`Failed to commit files to git: b`</code> が表示される問題を修正しました。
Bug fix	<code>`generate_password: false`</code> のときに TPA が Postgres ユーザーパスワードを誤って生成および上書きする問題を修正しました。
Bug fix	AWS でのボリュームマップの作成がリージョンを考慮していないため、 <code>eu-west-1</code> 以外のリージョンを使用すると障害が発生する問題を修正しました。

1.2.25 Trusted Postgres Architect 23.18 release notes

リリース日 2023 年 5 月 23 日

Trusted Postgres Architect 23.18 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
Enhancement	TPA は、初期レプリカの作成に <code>repmgr</code> の代わりに <code>pg_basebackup</code> を使用するようになりました。
Enhancement	TPA は、エアギャップ展開のためのローカルリポジトリの作成を除きて、SLES 15 をサポートするようになりました。
Enhancement	TPA は、PGD5 のマイナーバージョンのアップグレードをサポートするようになりました。
Enhancement	TPA は、 <code>`tpaexec test`</code> が実行されるときに改善されたテストを実行するようになりました。
Bug fix	TPA がサポートされていないディストリビューションで古い 2ndQuadrant リポジトリを使用しようとした問題を修正しました。
Bug fix	TPA が <code>pg_receivewal</code> が必要な Barman インスタンスに <code>pg_receivewal</code> をインストールしない問題を修正しました。

次のページに続く

表 38 – 前のページからの続き

Type	Description
Bug fix	TPA がプロビジョニング中に AWS ホストでデバイスをブロックするシンボリックリンクの作成に断続的に失敗し、展開が失敗する問題を修正しました。

1.2.26 Trusted Postgres Architect 23.17 release notes

リリース日 2023 年 5 月 10 日

Trusted Postgres Architect 23.17 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
Enhancement	新しい `--pgd-proxy-routing` パラメーターを <code>configure</code> コマンドに追加しました。これは、`global` または `local` に設定できます。ローカルルーティングは、すべての PGD-プロキシを独自の場所内の書き込みリーダーにルートさせます。グローバルルーティングは、すべてのプロキシを単一の書き込みリーダーにルートさせます。この書き込みリーダーは、すべての場所で使用可能なすべてのデータノードの間で選択されます。
Change	<code>configure</code> コマンドから `--active-locations` パラメーターを削除しました。
Enhancement	TPA が Ubuntu 22.04 をサポートするようになりました
Change	RHEL 7 および 8 で使用される AWS AMI を更新しました。
Bug fix	TPA が既存の Postgres ユーザーからグループを誤って削除する問題を修正しました。
Bug fix	git コミットが失敗したときに TPA が役に立たないエラーメッセージを出力する問題を修正しました。
Bug fix	グループ名が誤ってサニタイズされ、大文字が小文字ではなくアンダースコアに変換される問題を修正しました。
Bug fix	CAMO 構成後に必要なときに Postgres が再起動されない問題を修正しました。
Bug fix	etcd の変更に関する問題を修正し、べき等になったことを確認し、後続の展開での etcd の不必要な再起動を回避します。

1.2.27 Trusted Postgres Architect 23.16 release notes

リリース日 2023 年 3 月 21 日

Trusted Postgres Architect 23.16 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
Change	デフォルトの PGD-Always-ON クラスタは、2 つのデータノードと 1 つの監視ノードを含む関連サブグループを持つ 1 つの場所になりました。

次のページに続く

表 40 – 前のページからの続き

Type	Description
Change	TPA は、デフォルトですべてのデータノードに pgd-proxy を展開するようになりました。
Enhancement	別のプロキシインスタンスを作成する新しいオプション <code>--add-proxy-nodes-per-location N</code> を追加しました
Enhancement	<code>--data_nodes_per_location</code> が偶数の場合、TPA は監視ノードを自動的に追加し、2つの場所のみのクラスターを指定した場合に警告を出力するようになりました
Change	ロケーションを追加するのではなく、既に名前付けられた <code>--location-names</code> でのロケーションを監視専用として指定するため、パラメーター <code>--add-witness-only-location</code> の名前が <code>--witness-only-location</code> に変更されました。
Change	<code>tpaexec configure</code> 時に Postgres のフレーバーとバージョンを明示的に指定する必要があります
Enhancement	Postgres フレーバーとバージョンの新しい CLI 略語を追加しましたたとえば <code>--postgresql 14</code> または <code>--edbpg 15</code>
Enhancement	さまざまなサポートされている EDB ソフトウェアリポジトリの処理とドキュメントの向上
Change	TPA には、PGD-Always-ON クラスターのデフォルトで PGDG リポジトリが含まれません
Bug fix	EDB Failover Manager がデフォルトで EPAS のフェールオーバーマネージャーとして選択されない問題を修正しました
Bug fix	pglogical が M1 アーキテクチャに不必要にインストールされる問題を修正しました

1.2.28 Trusted Postgres Architect 23.15 release notes

リリース日 2023 年 3 月 15 日

Trusted Postgres Architect 23.15 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
Minor change	依存関係マッピングの変更。

1.2.29 Trusted Postgres Architect 23.14 release notes

リリース日 2023 年 2 月 23 日

Trusted Postgres Architect 23.14 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
Bug fix	PGD 3 および 4 構成のパッケージリストが正しく入力されなかったエラーを修正します。 TPA-365
Change	HARP の config.yaml で複数行の BDR DCS 構成を使用する TPA-360、RT90034

1.2.30 Trusted Postgres Architect 23.13 release notes

リリース日 2023 年 2 月 22 日

Trusted Postgres Architect 23.13 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
Bug fix	PGD-Always-ON および `--epas` を使用して古い EDB リポジトリを有効にしないでください。
Bug fix	PGD-Always-ON および `--postgres-version 15` のエラーを修正します。
	NA

1.2.31 Trusted Postgres Architect 23.12 release notes

リリース日 2023 年 2 月 21 日

Trusted Postgres Architect 23.12 の新機能、機能強化、バグ修正、およびその他の変更には次のものが含まれます。

Type	Description
Feature	コミットスコープに基づいた Commit At Most OnceCAMO 構成サポートを含む、 EDB Postgres Distributed 5 の完全なサポートを導入します。
Feature	EDB Postgres 拡張リポジトリとパッケージのサポートを導入します。
Enhancement	マルチリージョン AWS クラスターの構成の予備サポート。 マルチリージョンクラスターには、VPC と VPC の手動セットアップが必要です。
Enhancement	`--active-locations` のプロキシルーティングおよびそのためサブグループ RAFT を自動的に有効にします。 configure オプションを削除して、サブグループ RAFT をグローバルに有効にします。
Bug fix	EDB_SUBSCRIPTION_TOKEN がログに記録されていないことを確認します。
Bug fix	ユーザーが tpa_2q_repositories への products/default/release リポジトリの追加を抑制できるようにします。
Bug fix	利用可能な場合、ノードが bdr_child_group をサブスクリाइブしていることを確認します。 複数のサブグループを持つクラスターでは、TPA はインスタンスがトップグループとサブグループの両方のレプリケーションセットにサブスクリाइブされることを想定していなかったため、ノードのサブスクリाइブしたレプリケーションセットから後者を誤って削除します。

次のページに続く

表 44 – 前のページからの続き

Type	Description
Bug fix	Postgres が起動しない場合、有用なエラーで確実に失敗します。 Ansible のバグが原因で、Postgres が一部のインスタンスで起動しないが、他のインスタンスで起動した場合、デプロイメントは失敗しませんたとえば、構成の違いのため。展開を続行すると、後で障害が発生したホストの cluster_facts にアクセスしようとする、エラーが発生しました。
Bug fix	BDR 4.3 以降の監視では `bdr.alter_node_replication_sets()` を呼び出さないでください。 これは、監視ノードがカスタムレプリケーションセット構成で処理されない BDR バージョンの新しい制限に調整します。
Bug fix	ハードコードされた「barman」参照を置き換えて、barman_{user,group} 設定を使用して barman ユーザーとホームディレクトリをカスタマイズできるようにします。
Bug fix	必要に応じて、 postgres_databases[*].extensions の下で記載されている拡張機能の shared_preload_libraries エントリを追加します。
Bug fix	`pgaudit` が shared_preload_libraries の `bdr` の前に表示されないことを確認します既知のクラッシュを回避するため。
Bug fix	pgd-cli 構成ファイルの構文エラー DSN 引用符を修正します。
Bug fix	pgd-proxy 構成のエンドポイントを並べ替えて、ファイルの書き換えを回避します。 これには、次の展開で pgd-proxy の再起動が必要になる可能性があります、その後の展開での不要な将来の書き換え/再起動が回避されます。
Bug fix	RH システムにローカルリポジトリから rsync をインストール中のエラーを修正します。
Bug fix	Debian システムでの PEM 9 の Apache WSGI モジュール構成のエラーを修正します。
Bug fix	bdr 拡張子が意図的に作成された場合、未使用であっても削除しないでください。
	NA

1.2.32 Trusted Postgres Architect 23.1 to 23.11 release notes

TPA 23.11

リリース日 2023-01-31

重要な変更

- TPA-180 PGD-Always-ON アーキテクチャ今年後半にリリースの実験的なサポートを導入します。PGD-Always-ON アーキテクチャは、次の BDR バージョン 5 を使用します。初期サポートは内部目的に追加され、今後のリリースで改善されます。

マイナー変更

- TPA-349 依存関係バージョンのバンプ暗号化バージョン 38.0.4 から 39.0.0 のバンプ jq バージョン 1.3.0 から 1.4.0 のバンプ
- TPA-345 ドキュメント内の TPAexec 参照を TPA に変更します。製品を参照するときは、'TPAexec' の代わりに 'TPA' を使用するようにドキュメントを更新します。

TPA 23.10

リリース日 2023-01-04

マイナー変更

- TPA-161 harp_manager_restart_on_failure 設定デフォルト false を導入して、harp-manager systemd サービスの障害時のプロセスの再起動を有効にします

バグ修正

- TPA-281 AWS クラスターのプロビジョニングを解除するときに FMS セキュリティグループを削除する削除されていない依存関係が原因でクラスターの VPC のプロビジョニング解除の失敗を修正します。
- TPA-305 Tower の事前生成されたシークレットに enterprisedb_password を追加します
- TPA-306 PEM_PYTHON_EXECUTABLE が存在する場合は、/usr/bin/python3 に優先します PEM 9.0 での展開中の Python モジュールのインポートエラーを修正します。
- TPA-219 BDR インスタンスで pem-agent にデフォルトで bdr_database を監視させます

TPA 23.9

リリース日 2022-12-12

バグ修正

- TPA-301 Tower クラスターの cluster_dir の自動検出を修正 Tower プロジェクトディレクトリに基づいて cluster_dir を設定する場合、デプロイ先のインスタンスではなく、コントローラ上のディレクトリの存在を正しく確認するようになりました。
- TPA-283 Ansible Tower に必要な psutil への依存関係を追加します。
- TPA-278 rsyslog 構成から「umask 0」指示子を削除します。これにより、以前は rsyslogd.pid などの誰でも読み取り可能なファイルが作成されます。
- TPA-291 Postgres サーバーパッケージをインストールして Barman インスタンスで pg_receivewal を取得するときは、postgres_package_version 設定を尊重します。

TPA 23.8

リリース日 2022-11-30

重要な変更

- TPA-18 サポート Ansible Tower 3.8 このリリースでは、Ansible Tower 3.8 を介して bare クラスターつまり既存のサーバーでの `deploy.yml` のみの実行をサポートしています。Tower サーバーに TPAexec をインストールし、`tpaexec setup` を実行して、Tower テンプレートで利用できる仮想環境を作成して TPAexec プレイブックを実行します。--use-ansible-tower および--tower-git-repository 構成オプションを使用して、Tower 互換のクラスター構成を生成します。詳細については、[Ansible Tower](#) を参照してください。

マイナー変更

- TPA-238 クラスターディレクトリを git リポジトリとして初期化する TPAexec を実行するシステムで git が利用可能な場合、`tpaexec configure` はデフォルトでクラスターディレクトリ内の git リポジトリを初期化します。git が利用できない場合、以前と同様に続行されます。リポジトリの作成を回避するにはたとえば、既存のリポジトリ内にクラスターディレクトリを保存する場合、--no-git オプションを使用します。

TPA 23.7

リリース日 2022-11-09

重要な変更

- TPA-234 Ansible 2.9 のコミュニティリリースをサポート TPAexec は、Ansible 2.9 の 2ndQuadrant/ansible フォークを必要としました。このリリースでは、代わりに Ansible のコミュニティリリースを `tpaexec setup --use-community-ansible` で使用することを選択できます。今のところ、デフォルトは引き続き 2ndQuadrant/ansible を使用することです。これは将来のリリースで変更される予定です。2ndQuadrant / ansible のサポートがドロップされ、Ansible が新しいデフォルトになります。

マイナー変更

- TPA-209 Postgres 15 のリリースに続いて、有効な `tpaexec configure` オプションとして--postgres-version 15 を受け入れます
- TPA-226 --hostnames-from ファイルの IP アドレスの受け入れ以前は、`tpaexec configure` に渡されるファイルには、行ごとに 1 つのホスト名が含まれることが想定されていました。各ホスト名の後にオプションの IP アドレスを含めることもできるようになりました。存在する場合、このアドレスは config.yml の対応するインスタンスの `ip_address` として設定されます。独自の--hostnames-from ファイルを指定すると、ホスト名がデフォルトでランダム化されなくなります。
- TPA-231 新しい bdr-pre-group-join フックを追加しますこのフックは、各ノードが BDR ノードグループに参加する前に実行されます。TPAexec が提供するデフォルトのレプリケーションセット構成を変更するために使用される場合があります。

- TPA-130 Community.postgresql の postgresql_user モジュールを使用します SCRAM パスワードを使用するときにタスクステータスを正しく報告するには、Community.postgresql コレクションの更新されたモジュールが必要です。デフォルトモジュールは常に **changed** を報告します。
- TPA-250 さまざまな Python 依存関係の最新バージョンへのアップグレード

バグ修正

- TPA-220 .bashrc の LD_LIBRARY_PATH が「:」で始まらないことを確認します
- TPA-82 BDR の削除を回避する -internal \${group_name}_ext レプリケーションセット
- TPA-247 AWS での「'str object' には属性 'node_dsn'」エラーを修正します。コードは hostvars[hostname] を中間変数に割り当てず、後で通常の dict と同様に動作することを想定しています時々のみ動作します。これにより、PEM が有効になっている AWS クラスターで報告された 23.6 の回帰が修正されますが、コードベース全体の他の同様のエラーも修正されます。
- TPA-232 インベントリで生成されたシークレットへのシンボリックリンクを作成する際の競合状態を排除し、「Error while linking: [Errno 17] File attempts」エラーを発生させました
- TPA-252 すべての BDR ノードを監視専用レプリケーションセットに公開する復元コードこのコードブロックは、TPA-193 に対して行われたリファクタリング作業の一部として、v23.6 リリースで誤って削除されました。

TPA 23.6

リリース日 2022-09-28

重要な変更

- TPA-21 AWS 展開には、メンテナンスされていない boto2 の代わりに boto3 AWS クライアントライブラリを使用します。これにより、SSO ログインやその他の便利な機能が有効になります。
- TPA-202 harp-config フックを追加します。この展開時フックは、HARP がインストールおよび構成された後、HARP がインストールされているすべてのノードで開始される前に実行されます。

バグ修正

- TPA-181 RHEL 7 でデフォルトの Python バージョンを 2 に設定します。以前は、tpaexec は RHEL 7 と Python 3 のサポートされていない組み合わせで config.yml を生成できました。
- TPA-210 既存のセキュリティグループを使用して aws 展開を修正します。このような展開は、プロビジョニング時に失敗していましたが、予想通りに動作するようになりました。
- TPA-189 プロビジョニング解除時に group_vars ディレクトリを削除します。これにより、ダングリングのシンボリックリンクが原因で後続のプロビジョニングが失敗する原因となった問題が修正されます。
- TPA-175 共有メモリセグメントをそのままにするように systemd を正しく構成します。これはソースビルドにのみ影響します。
- TPA-160 haproxy および PEM のバージョン設定を許可します。これは、特定のバージョンが指定された場合でもパッケージの最新バージョンがインストールされるバグを修正します。

- TPA-172 正しいホストセットに EFM をインストールします。EFM は、PEM サーバーなどの他の理由で postgres がインストールされているサーバーではなく、クラスターのメンバーである postgres サーバーにのみインストールする必要があります。
- TPA-113 PEM エージェント登録をシリアル化します。これにより、複数のホストが pemworker-register-agent を同時に実行しようとした場合の競合状態が回避されます。

TPA 23.5

リリース日 2022-08-23

重要な変更

- TPA-81 Ubuntu 22.04 Jammy の tpaexec および tpaexec-deps パッケージを公開する
- TPA-26 単一ノードでの harp-proxy および harp-manager のインストールをサポートします。harp-proxy と harp-manager サービスの両方をクラスター内の同じターゲットノードで実行できるようになりました。

TPA 23.4

リリース日 2022-08-03

バグ修正

- TPA-152 は、AWS ホストでの Debian インスタンスの最初のブート中のロケール検出に関する問題を修正します。ホストは最初のブートを完了できず、SSH キーネゴシエーションの問題と、展開中に見つからないディスクのエラーとして現れます。この問題は 23.3 で導入され、TPA-38 に関連しています

TPA 23.3

リリース日 2022-08-03

重要な変更

- TPA-118 harp-manager 構成の 2 つの新しいオプションを公開しました。最初の設定は dcs request_timeout と同様に HARP harp_db_request_timeout を設定しますが、データベース接続用であり、2 番目の harp_ssl_password_command は、SSL 対応データベース接続の ssl キーの復号化に使用される ssl パスワードの難読化を解除するために使用されるコマンドを指定します

マイナー変更

- TPA-117 tpaexec config.yml の package_version オプションでのワイルドカードの使用に関するドキュメントの更新を追加します。これにより、deploy 操作中に予期しないパッケージのアップグレードが発生する可能性があるという警告が導入されます。詳細については、tpaexec-configure.md のドキュメントを参照してください

- TPA-38 Debian のすべてのバージョン、および RHEL 8 以降のロケールファイルを追加します。Barman などの一部の EDB ソフトウェアでは、ユーザーロケールを `en_US.UTF-8` に設定する必要があります。一部のユーザーは、ロケール、文字セット、または言語をローカル地域に変更したい場合があります。この変更により、libc が提供する OS ファイルが、ユーザーデータスクリプトを使用したファーストブート中に AWS インスタンスにインストールされます。デフォルトロケールは `en_US.UTF-8` です。詳細については、`platform_aws.md` ドキュメントを参照してください
- TPA-23 クラスターサービス Barman、HARP、repmgr、PgBouncer、および EFM の syslog のログ構成を追加します。指定されたログサーバーは、これらのサービスの `/var/log/hosts` ディレクトリで受信したログファイルを保存します
- TPA-109 BDR の特定のバージョンに基づいて、ロックタイムアウト構文の選択を中心とした `pgbench` ロールのコードのマイナーリファクタリング

バグ修正

- TPA-147 ソースインストール方法を使用するクラスターでは、Debian および Rocky Linux の一部の欠落パッケージが観察されました。Debian は、`krb5` および `lz4` のライブラリヘッダーを受信します。RedHat 派生 OS では、「Development Tools」パッケージグループの必須パッケージと `libcurl` ヘッダーが追加されました
- TPA-146 Postgres 9.6 をインストールするクラスターのパッケージ選択方法の小さな修正
- TPA-138 `local-repoconfigure` オプションを有効にする「ベア」プラットフォームを使用するクラスターでの警告メッセージに対処します。OS はベアプラットフォームの `TPAexec` によって管理されないため、`local-repo` 構造を作成するようにユーザーに通知する必要があります。これにより、以前は未処理のエラーが発生し、構成の進行状況が停止しました
- TPA-135 “docker” プラットフォームおよび Rocky Linux イメージで `--use-local-repo-only` を使用する場合、ノード上の既存の `yum` リポジトリ構成の初期削除は、コマンド `find` および `xargs` が欠落しているため失敗します。この変更により、`findutils` パッケージがソースリポジトリに存在する場合、最初にインストールされます
- TPA-111 `--enable-pem` オプションが指定されている場合、PEM サーバーにエージェントを登録するために使用されるコマンドの冗長な追加引数を削除します。以前は、最初の引数現在削除された引数が 2 番目の引数によってオーバーライドされるため、問題は発生しませんでした。
- TPA-108 Postgres がソースからインストールされている場合、ポストマスターシンボリックリンクの SELinux ファイルコンテキストを復元します。以前は、`systemd` デーモンが Postgres データ `bin` ディレクトリに保存されているシンボリックリンクを読み取ることができないため、ソースから `postgres` をインストールしている SELinux 対応 OS を使用しているクラスターは、Postgres の再起動に失敗していました。これは、AWS で最近採用された Rocky Linux イメージを使用したテストで発見されました。デフォルトで SELinux が有効になり、強制モードになっています

TPA 23.2

リリース日 2022-07-13

重要な変更

- Barman および PEM で使用する Postgres Backup API のサポートを追加します。
--enable-pg-backup-api オプションを介してアクセスできます。
- SSL 証明書は、Postgres Backup API プロキシサービスのサーバー証明書など、サービスごとに作成できるようになりました。証明書は/etc/tpa/<service>/<hostname>.cert に配置されます。これらの証明書は、クラスター用に生成された CA 証明書で署名することもできます。
- BDR の Etcd の配置 - Always-ON アーキテクチャ ' harp_consensus_protocol: etcd' を使用する場合、次の各インスタンスのロールに ' etcd' を明示的に追加します。
- BDR プライマリ ' bdr' ロール
- BDR 論理スタンバイ ' bdr' + ' readonly' roles)
- Bronze レイアウトの場合のみ BDR Witness ' bdr' + ' witness' ロール)
- Gold レイアウトのみ Barman ' barman' ロール クレジット: Gianni Ciolli gianni.ciolli@enterprisedb.com

マイナー変更

- 製品ブランドの変更を反映するように、configure 引数--2q を--pgextended に置き換えます。既存の構成は、予想される動作を保持します。
- Debian の古い安定版に付属している docker のバージョン 18 を使用する場合の Docker プラットフォームの互換性チェックのエラーレポートを改善しました。
- 不足しているコマンドを CLI ヘルプドキュメントに追加します。
- configure コマンドのエラーレポートが改善されました。
- ソースから BDR 5 をビルドするための初期サポートを追加します。クレジット: Florin Irion florin.irion@enterprisedb.com
- Postgres の古いバージョンから EDB 製品への移行の継続的な互換性を確保するための変更。

バグ修正

- download-packages コマンドを使用してローカルリポジトリにデータを入力するときに、etcd のパッケージが欠落することを意味する問題を修正しました。
- efm フェールオーバーマネージャーの使用とそのパッケージの依存関係の選択に影響を与える問題を修正しました

TPA 23.1

リリース日 2022-06-21

このリリースでは、アップグレード後に tpaexec setup を実行する必要があります、それ以外の場合はエラーで失敗します。

パッケージのインストール動作の変更

以前のバージョンでは、正確なバージョンが明示的に指定されていない限り、`tpaexec deploy` を実行すると、インストールされたパッケージをアップグレードする可能性があります `postgres_package_version` 設定など。ただし、これは決して安全、サポート、または推奨されるアップグレード方法ではありません。特に、展開中のパッケージのアップグレード後に、サービスが安全かつ正しく再起動されていない可能性があります。

このリリース以降、`tpaexec deploy` は、インストールされたパッケージをアップグレードすることはありません。最初の展開は、必要なパッケージをすべてインストールします特定のバージョンが設定されている場合、または利用可能な最新のパッケージをインストールします。後続の実行では、パッケージがインストールされたことを確認し、それ以上は何も行いません。これは、予測可能で安全な新しいデフォルト動作です。

コンポーネントを更新する必要がある場合は、`tpaexec update-postgres` を使用します。このリリースでは、コマンドは Postgres および BDR または `pglogical` などの Postgres 関連パッケージ、および HARP、`pgbouncer`、`etcd` などの特定の他のコンポーネントを更新できます特定のクラスターに該当する場合。将来のリリースでは、より多くのコンポーネントのアップグレードを安全にサポートする予定です。

重要な変更

- HARP ブートストラップ構成が変更された場合にのみ「`harpctl apply`」を実行します 警告これにより、`tpaexec deploy` を実行したときに既存のクラスターで単一の `harp` サービスがリスタートします。これは、リストが一貫した順序になるように `config.yml` が変更され、意図しない変更が回避されるためです将来の展開
- `tpaexec download-packages` コマンドを追加して、クラスターに必要なすべてのパッケージを `local-repo` ディレクトリにダウンロードして、エアギャップ/切断された環境のクラスターインスタンスにコピーできるようにします。詳細については、`air-gapped.md` および `local-repo.md` を参照してください。
- 新しい BDR の `--harp-consensus-protocol <etcd|bdr>` 構成オプションが必要です-Always-ON クラスターコンセンサスプロトコルの選択がデータセンター/場所間のネットワークレイテンシーによっては、フェイルオーバーパフォーマンスに悪影響を与える可能性があるため、TPAexec はここでデフォルト値を提供しなくなり、ユーザーの状態が向上します特定のクラスターに最適なプロトコルを選択します。これは、新しく生成されたクラスターの構成に影響しますが、`harp_consensus_protocol` を明示的に設定せずに以前のデフォルトの `etcd` を使用する既存のクラスターには影響しません。

マイナー変更

- 古いバージョンが利用できないディストリビューションに EFM 用の `openjdk-8` の代わりに `openjdk-11` をインストールする
- `harp_log_level` 設定たとえば `cluster_vars` の下を受け入れて、デフォルトの `harp-manager` および `harp-proxy` ログレベル `info` をオーバーライドします
- 復元力を向上させるために、個々のエンドポイント DSN のリストの代わりに単一のマルチホスト BDR DCS エンドポイント DSN を使用するように `harp-proxy` を構成します
- `harp-manager` の BDR DCS のローカル Unix ソケット DSN から余分な接続属性たとえば `ssl*` を省略します

バグ修正

- harp-manager と harp-proxy の構成が変更された場合に、harp-manager と harp-proxy が再起動されていることを確認します
- 読み取り専用の harp_dcs_user に必要な追加の新しい権限を付与することにより、harp プロキシエラーを修正します
- CAMO が有効になっているときに BDR4 トランザクションストリーミングを無効にする bdr.enable_camo が設定されている場合、bdr.default_streaming_mode を無効にする必要があります。これは、BDR4 の CAMO で保護されたトランザクションと互換性がありません。これにより、CAMO 対応 BDR4 クラスターいずれにせよストリーミングが有効になった状態では動作しませんでした。サーバーが再起動します。

1.3 TPA installation

TPA を使用するには、パッケージまたはソースからインストールし、`tpaexec setup` コマンドを実行する必要があります。このドキュメントでは、TPA パッケージをインストールする方法について説明します。EDB サブスクリプションプランを契約しており、EDB リポジトリにアクセスできる場合、これらの手順に従う必要があります。ソースから TPA をインストールするには、[Installing TPA from source](#) を参照してください。

サポートされているプラットフォームについては、[Distribution support](#) を参照してください。

注釈

システムの日付と時刻が正しいことを完全に確認してください。そうしないと、さまざまな機能が失敗します。ネットワーク時間、例 `sudo ntpdate pool.ntp.org` を使用することをお勧めします

1.3.1 クイックスタート

まず、[subscribe to an EDB repository](#)。

TPA のインストール

```
sudo <your-package-manager> install tpaexec
```

追加の依存関係をインストールする

```
sudo /opt/EDB/TPA/bin/tpaexec setup
```

インストールの検証一般のユーザーとして実行

```
/opt/EDB/TPA/bin/tpaexec selftest
```

各ステップのより詳細な説明を以下に示します。

1.3.2 TPA をインストールする場所

サポートされているプラットフォームを使用している限り、TPA はワークステーションからインストールして実行できます。これは、学習、ローカルテスト、またはデモンストレーションの目的に適しています。TPA は [Docker のインストール](#) をサポート

独自のワークステーションで完全な展開を実行したい場合。

運用環境で使用する場合、専用の永続的な仮想マシンで TPA を実行することをお勧めします。これにより、クラスターディレクトリが保持され、将来のクラスター管理または更新にチームが利用できるためです。また、TPA のコピーを 1 つだけ更新する必要がある、単一の TPA ホストからターゲットインスタンスへのネットワークアクセスを提供するだけであることも意味します。

1.3.3 TPA パッケージのインストール

TPA をインストールするには、最初に [subscribe to an EDB repository](#) を行う必要があります。TPA は、すべての EDB リポジトリで使用できます。

次のとおりに TPA をインストールします。

Debian または Ubuntu にインストールする

```
sudo apt-get install tpaexec
```

RHEL、Rocky、AlmaLinux、または Oracle Linux にインストールする

```
sudo dnf install tpaexec
```

SLES

```
sudo zypper install tpaexec
```

これにより、TPA が `/opt/EDB/TPA` にインストールされます。また、他の必要なパッケージ Python 3.12 以降がインストールされていることも確認します。

ここで `sudo` について言及しているのは、どのコマンドが root 権限を必要とするかを示すためです。他の手段を使用して、root としてコマンドを実行できます。

1.3.4 TPA Python 環境のセットアップ

次に、`tpaexec setup` を実行して、分離された Python 環境を作成し、必要なすべてのモジュールの正しいバージョンをインストールします。

注釈

20.04 より前の Ubuntu バージョンでは、`sudo -H tpaexec setup` を使用して、`tpaexec configure` 中の後続の権限エラーを回避してください

```
sudo /opt/EDB/TPA/bin/tpaexec setup
```

これは/opt/EDB/TPA に書き込むため、root として実行する必要がありますが、プロセスは、インストールしているシステム全体の Python モジュール Ansible を含むには影響しません。

通常 tpaexec コマンドを実行するユーザーの PATH に/opt/EDB/TPA/bin を追加します。たとえば、これを.bashrc または同等のシェル構成ファイルに追加できます。

```
export PATH=$PATH:/opt/EDB/TPA/bin
```

1.3.5 インターネットまたはネットワークアクセスエアギャップなしで TPA をインストールする

このセクションでは、EDB リポジトリ、Python パッケージインデックス、またはその両方にアクセスできないサーバーに TPA をインストールする方法について説明します。このような環境で TPA を使用方法については、*Managing clusters in a disconnected or air-gapped environment* を参照してください。

TPA パッケージのダウンロード

TPA をインストールする必要があるサーバーから EDB リポジトリに直接アクセスできない場合、インターネットに接続されたマシンからパッケージをダウンロードして転送できます。これを行うには、いくつかの方法があります。

インターネットに接続されたマシンがターゲットと同じオペレーティングシステムを使用している場合、yumdownloader RHEL ライクまたは apt download Debian ライクを使用してパッケージをダウンロードすることをお勧めします。

または、[EDB Repos](#) にアクセスし、[Download EDB software Packages from your Browser] という見出しの下で [Enterprise]、[Standard] または [Community 360] のいずれかを選択することにより、ブラウザーから任意のプラットフォームのパッケージをダウンロードできます。TPA をインストールするには、次のパッケージが必要です。

- tpaexec
- tpaexec-deps
- edb-python312

ダウンロードしたパッケージをターゲットサーバーに転送したら、プラットフォームに適切なツールを使用してインストールする必要があります。

Python パッケージインデックスへのアクセスを使用しないインストール

tpaexec setup を実行すると、通常は Python パッケージインデックスから Python パッケージをダウンロードします。環境が別のインデックスを提供しない限り、デフォルトは公式の PyPI です。使用可能なパッケージインデックスがない場合は、インストールした tpaexec と同じ方法で tpaexec-deps パッケージをインストールする必要があります。tpaexec-deps パッケージ tpaexec と同じリポジトリから入手できますは、ダウンロードされるすべてをバンドルしているため、ネットワークアクセスなしでインストールできます。tpaexec setup を実行する前にパッケージをインストールするだけで、バンドルされたコピーが自動的に使用されます。

1.3.6 TPA インストールの検証

上記の手順がすべて完了したら、次のコマンドを実行してローカルインストールを確認します。

```
tpaexec selftest
```

そのコマンドがエラーなしで完了した場合、TPA インストールは使用できる状態です。

1.3.7 TPA のアップグレード

TPA の新しいリリースにアップグレードするには、次のことを行う必要があります。

1. 最新の `tpaexec` パッケージをインストールする
2. 最新の `tpaexec-deps` パッケージ必要な場合、上記を参照してください。
3. `tpaexec setup` を再度実行します

上記のように TPA パッケージリポジトリをサブスクライブしている場合、`apt-get update && apt-get upgrade` または `yum update` を実行すると、これらのパッケージの利用可能な最新バージョンをインストールする必要があります。それ以外の場合は、利用可能な任意の手段でパッケージをインストールできます。

`tpaexec` の新しいバージョンがインストールされるときは、`tpaexec setup` を再度実行することをお勧めします。一部の新しいリリースはこれを厳密に必要としない場合がありますが、他のリリースはそれなしでは動作しません。

1.3.8 Ansible バージョン

TPA は、デフォルトで `ansible-core 2.16` を使用します。

1.4 Open source TPA

1.4.1 Trusted Postgres ArchitectTPA とは何ですか？

TPA は、[EnterpriseDB \(EDB\)](#) によって開発されたオーケストレーションツールで、Ansible を使用して EDB の推奨事項に従って Postgres クラスターを展開します。

TPA は、Postgres の展開とサポートに関する長年の苦労して得た経験に基づいて、EDB が追従するベストプラクティスを具体化しています。これらの推奨事項は、運用環境と同様に簡単なテストベッドのセットアップに適用されます。

1.4.2 次のステップ

- [Installing TPA from source](#)
- [A First Cluster Deployment](#)
- [TPA's full documentation online](#)

1.4.3 TPA オープンソースのよくある質問

EDB の顧客でない場合、これを使用できますか？

はい、TPA は、GPLv3 ライセンスの下のオープンソースプロジェクトです。オープンソースソフトウェア、EDB 独自の製品、またはその組み合わせで構成されるクラスターの展開をサポートします。

問題を報告できますか？

はい、EDB をご利用の場合は、サポートにご連絡ください。それ以外の場合は、GitHub Issue を開きます。

貢献できますか？

もちろん、ご意見をお待ちしておりますが、コーディングを開始する前にイシューをオープンしてください。私たちは TPA ができること/すべきことをかなり選択しているため、新機能よりもバグ修正が受け入れられる可能性が高いです。

1.5 Installing TPA from source

このドキュメントでは、ソースコードリポジトリのコピーから TPA を使用方法について説明します。

注釈

EDB 顧客がソフトウェアの EDB サポートを受けるには、*TPA installation* 必要があります。

ソースから TPA を実行するには、パッケージが処理するすべての依存関係たとえば Python 3.12+をインストールするか、ソースと *Running TPA in a Docker container* をダウンロードする必要があります。(どちらの方法も Linux および macOS では正常に動作します。)

1.5.1 クイックスタート

最初に、さまざまな依存関係である Python 3、Python venv、git、openvpn、および patch をインストールする必要があります。EDB リポジトリからインストールすると、これらは TPA パッケージとともに自動的にインストールされます。

TPA をインストールする前に、必要なパッケージをインストールする必要があります。

- **Debian/Ubuntu** `sudo apt-get install python3 python3-pip python3-venv git openvpn patch`
- **Redhat, Rocky または AlmaLinux(RHEL7)** `sudo yum install python3 python3-pip epel-release git openvpn patch`
- **Redhat, Rocky または AlmaLinux(RHEL8)** `sudo yum install python36 python3-pip epel-release git openvpn patch`

1.5.2 クローンとセットアップ

前提条件がインストールされると、リポジトリのクローンを作成できます。

```
git clone https://github.com/enterprisedb/tpa.git ~/tpa
```

これにより、ホームディレクトリに `tpa` ディレクトリが作成されます。

ssh を使用してチェックアウトする場合は、次を使用します。

```
git clone ssh://git@github.com/EnterpriseDB/tpa.git ~/tpa
```

新しく作成したクローン内にある `bin` ディレクトリを、次のパスに追加します。

```
export PATH=$PATH:$HOME/tpa/bin
```

この行を `.bashrc` ファイルまたは使用するシェルの他のプロファイルファイルに追加します。

次を実行して、作業用の `tpa` 環境を作成できるようになりました。

```
tpaexec setup
```

これにより、TPA が将来的に使用する Python 仮想環境が作成されます。必要なパッケージはすべてこの環境にインストールされます。これが正しく構成されたことをテストするには、次を実行します。

```
tpaexec selftest
```

これで、`tpaexec` がインストールされました。

1.5.3 依存関係

Python 3.12+

TPA には、ほとんどの最新のディストリビューションで利用可能な Python 3.12 以降が必要です。お持ちでない場合は、`pyenv` を使用して、システムパッケージに影響を与えることなく、希望のバージョンの Python をインストールできます。

```
# First, install pyenv and activate it in ~/.bashrc

# See https://github.com/pyenv/pyenv#installation

# (e.g., `brew install pyenv` on MacOS X)

$ pyenv install 3.12.0
Downloading Python-3.12.0.tar.xz...
- > https://www.python.org/ftp/python/3.12.0/Python-3.12.0.tar.xz
Installing Python-3.12.0...
Installed Python-3.12.0 to /home/ams/.pyenv/versions/3.12.0

$ pyenv local 3.12.0
$ pyenv version
```

(次のページに続く)

(前のページからの続き)

```
3.12.0 (set by /home/ams/pyenv/.python-version)

$ pyenv which python3
/home/ams/.pyenv/versions/3.12.0/bin/python3
$ python3 --version
3.12.0
```

pyenv をまだ使用していない場合は、[pyenv documentation](#) の説明に従って、pyenv を .bashrc の PATH に追加し、`eval "$(pyenv init -)"` を呼び出してください。

仮想環境オプション

デフォルトでは、`tpaexec setup` はビルトイン `Python 3 -m venv` を使用して `$TPA_DIR/tpa-venv` の下に `venv` を作成し、`tpaexec` が呼び出されるたびに自動的にアクティブにします。

`tpaexec setup --venv /other/location` を実行して、新しい `venv` の別の場所を指定できます。

デフォルトの `venv` の場所を使用することを強くお勧めします。別の場所を使用する場合は、環境変数 `TPA_VENV` をその場所に設定する必要があります。たとえば、次の行を .bashrc または他のシェル起動スクリプトに追加します。

```
export TPA_VENV="/other/location"
```


第2章 Creating a cluster

2.1 A First Cluster Deployment

この短いチュートリアルでは、単純な *MI* デプロイメントをローカル Docker インストールに展開する手順を説明します。最後までに、4つのコンテナ、1つのプライマリデータベース、2つのレプリカ、およびバックアップノードが構成され、探索できる準備が整います。

この例では、Ubuntu システムで TPA を実行しますが、ほとんどの Linux システムで考慮事項は同様です。

2.1.1 TPA のインストール

あなたが EDB 顧客の場合、EDB のリポジトリから TPA パッケージを直接インストールする *TPA installation* に従ってください。

TPA のオープンソースユーザーであれば、Github.com からダウンロードできる *Installing TPA from source* があります。

これらのガイドに従って、ここに戻ります。

2.1.2 Docker のインストール

前述のように、展開例を Docker に展開します。まだ Docker をインストールしていない限り、それをセットアップする必要があります。

Debian または Ubuntu では、次のコマンドを実行して Docker をインストールします。

```
sudo apt update
sudo apt install docker.io
```

他の Linux ディストリビューションについては、[Docker Engine Install page](#) を参照してください。

次を使用してユーザーを docker グループに追加します。

```
sudo usermod -aG docker <yourusername>
newgrp docker
```

警告

ユーザーに Docker デーモンと対話できる機能を付与すると、Docker ホストでの root を簡単に取得できます。信頼できるユーザーのみが Docker デーモンにアクセスする必要があります。

2.1.3 TPA を使用した構成の作成

このプロセスの次のステップは、構成を作成することです。TPA は、`configure` コマンドを介して作業のほとんどを実行します。あなたがしなければならないのは、コマンドラインフラグとオプションを指定して、展開するものを大まかに選択することです。`tpaexec configure` コマンドは次のとおりです。

```
tpaexec configure demo --architecture M1 --platform docker --postgresql 15 --enable-
→repmgr --no-git
```

これにより、`M1` を含む `demo` という構成が作成されます。したがって、プライマリ、レプリカ、バックアップノードがあります。

`--platform docker` は、この構成をローカルの Docker インスタンスで作成する必要があることを TPA に指示します。すべてのコンテナと OS 要件をプロビジョニングします。他のプラットフォームには、アマゾンウェブサービスと同じことを行う `awscli` を使用してボリューム属性を変更すると `bare(-metal servers)` が含まれます。オペレーティングシステムのプロビジョニングをスキップし、既に構成された Linux ホストへのソフトウェアのインストールに直接進みます。

`--postgresql 15` では、Community Postgres バージョン 15 を使用するように TPA に指示します。ソフトウェアの選択に関していくつかのオプションがありますが、これはオープンソースユーザーにとって最も簡単なデフォルトです。

`--enable-repmgr` を追加すると、TPA に *Replication Manager* を使用してレプリケーションとフェイルオーバーをハンドリングするように展開を構成するように指示します。

最後に、`--no-git` は、`git` を介して構成をリビジョン制御できる TPA の機能をオフにします。

このコマンドを実行すると、どうやらコマンドラインでは何も起こりません。ただし、`demo` というディレクトリが作成され、新しい展開のブループリントである `config.yml` ファイルを含むいくつかのファイルが含まれています。

デプロイメントのプロビジョニング

これで、新しい展開を実行するコンテナまたは仮想マシンを作成する準備が整いました。これは、`provision` コマンドで実現できます。実行します

```
tpaexec provision demo
```

構成の展開の準備に必要なさまざまな操作を介した TPA の動作が表示されます。

Deploying

プロビジョニングが完了したら、展開に進むことができます。これにより、必要に応じて、オペレーティングシステムとシステムパッケージがインストールされます。次に、要求された Postgres アーキテクチャをインストールし、必要な構成をすべて実行します。

```
tpaexec deploy demo
```

構成を展開するために必要なさまざまな操作を介した TPA の動作が表示されます。

テスト

新しいデータベースで `pgbench` を実行する `tpaexec test` コマンドを使用して、新しく展開された構成をすばやくテストできます。

```
tpaexec test demo
```

接続しています

`psql` プロンプトを表示するには、最も簡単なルートは、`docker` または `SSH` を使用して、コンテナまたは構成に応じて VM またはホストのいずれかにログインすることです。走る

```
tpaexec ping demo
```

展開内のすべての接続可能なホストに `ping` を実行すると、次のような出力が得られます。

```
$ tpaexec ping demo
unfair | SUCCESS => {
  "changed": false,
  "ping": "pong"
}
uptake | SUCCESS => {
  "changed": false,
  "ping": "pong"
}
quondam | SUCCESS => {
  "changed": false,
  "ping": "pong"
}
uptight | SUCCESS => {
  "changed": false,
  "ping": "pong"
}
```

`SUCCESS` で応答したノードの 1 つを選択します。この例では、`uptake` を使用します。

`docker` のみを使用する予定がある場合は、`docker exec -it uptake /bin/bash` コマンドを使用して、適切なホスト名に置き換えます。

すべての種類の TPA 展開で機能する別のオプションは、`SSH` を使用することです。これを行うには、最初に現在のディレクトリを作成した構成ディレクトリに変更します。

たとえば、構成の名前付けは `demo` なので、そのディレクトリに移動します。そこで、`ssh -F ssh_config ourhostname` を実行して接続します。

```
cd demo
ssh -F ssh_config uptake
Last login: Wed Sep  6 10:08:01 2023 from 172.17.0.1
[root@uptake ~]#
```

どちらの場合も、コンテナに root ユーザーとしてログインします。

`sudo -iu postgres` を使用して、ユーザーを postgres ユーザーに変更できるようになりました。postgres として、`psql` を実行できます。TPA はそのユーザーを .pgpass ファイルで既に構成しているため、パスワードを提示する必要はありません。

```
[root@uptake ~]#
postgres@uptake:~ $ psql
psql (15.4)
Type "help" for help.

postgres=#
```

そして、データベースに接続しています。

SSH を使用せずにホストシステムからコンテナに接続できます。ssh_config ファイルから接続するホストの IP アドレスを取得します。

```
$ grep "^ *Host" demo/ssh_config
Host *
Host uptight
    HostName 172.17.0.9
Host unfair
    HostName 172.17.0.4
Host quondam
    HostName 172.17.0.10
Host uptake
    HostName 172.17.0.11
```

Uptake に接続するので、IP アドレスは 172.17.0.11 です。

また、postgres ユーザーのパスワードも取得する必要があります。tpaexec show-password demo postgres を実行して、システムから保存されたパスワードを取得します。

```
tpaexec show-password demo postgres
a9LmI1X^uM0pPoEnLuRdL%L$oRQak3om
```

Postgresql クライアントがインストールされていると仮定すると、次のことを実行できます。

```
psql --host 172.17.0.11 -U postgres
Password for user postgres:
```

前に取得したパスワードを入力します。

```
psql (14.9 (Ubuntu 14.9-0ubuntu0.22.04.1), server 15.4)
WARNING: psql major version 14, server major version 15.
         Some psql features might not work.
SSL connection (protocol: TLSv1.3, cipher: TLS_AES_256_GCM_SHA384, bits: 256,
compression: off)
```

(次のページに続く)

(前のページからの続き)

```
Type "help" for help.
```

```
postgres=#
```

これで、Docker ホストから、TPA 展開された Docker コンテナのいずれかで実行されている Postgres に接続されました。

2.2 Cluster configuration

`tpaexec configure` コマンドは、プロビジョニング/展開/テストサイクルの後続のステージに必要な YAML クラスター構成ファイルを生成します。

2.2.1 クイックスタート

```
[tpa]$ tpaexec configure ~/clusters/speedy --architecture M1 \
      --postgresql 14 \
      --failover-manager repmgr
```

このコマンドは、`~/clusters/speedy` という名前のディレクトリを作成し、`M1` という名前のアーキテクチャのレイアウトに従う `config.yml` という名前の構成ファイルを生成します。シングルプライマリ、N レプリカ。新しいディレクトリに git リポジトリを作成し、生成された `config.yml` を含む初期コミットを作成します。

また、コマンドは、構成を変更するためのさまざまなオプション選択したアーキテクチャまたはプラットフォームに固有の一部を受け入れますが、デフォルトは賢明で、すぐに使用できるように設計されています。生成された `config.yml` を読み、ニーズに合わせて構成を微調整することをお勧めします。(これが *Instance configuration* の概要です。)

`config.yml` を完全に手動で書くことも可能ですが、生成されたファイルを編集するほうが簡単です。

2.2.2 構成オプション

最初の引数は、クラスターディレクトリ、たとえば、`speedy` または `~/clusters/speedy` である必要があります。クラスターの名前は、どちらの場合も `speedy` になります。すべてのクラスターを共通のディレクトリたとえば、上記の例では `~/clusters` に保持することをお勧めします。

アーキテクチャ、たとえば `M1`、`BDR-Always-ON`、または `PGD-X` を選択するには、次の引数が `--architecture <name>` である必要があります。アーキテクチャの完全なリストについては、`tpaexec info architectures` を実行してください。

次に、インストールする *Postgres のフレーバーとバージョン* を指定する必要があります。

上記の引数は常に必須です。上記の例のように、ここで説明するオプションの残りの部分は安全に省略できます。デフォルトでは使用可能な結果が得られます。

共通オプションのリストについては、`tpaexec help configure-options` を実行します。

アーキテクチャ固有のオプション

選択したアーキテクチャにより、他のどのオプションが受け入れられるかが決まります。通常、各アーキテクチャは、いくつかの一意のオプションと以下で説明する汎用オプションを受け入れます。

たとえば、M1 では `--location-names l1 l2` を使用して、2つの名前付けの場所にノードを含むクラスターを作成できます。アーキテクチャが受け入れるオプションのリストについては、アーキテクチャのドキュメントを参照してください。または、場合によっては必要となります。

プラットフォームオプション

次に、`--platform <name>` を使用してプラットフォーム、たとえば *awscli* を使用してボリューム属性を変更する または *bare(-metal servers)* を選択できます。

アーキテクチャは特定のプラットフォームをサポートする場合と、サポートしない場合があります。そうしない場合、クラスターの構成に失敗します。

プラットフォームの選択は、特定のオプションの解釈に影響を与えます。たとえば、aws を選択した場合、`--region <region>` および `--instance-type <type>` への引数は有効な *AWS region name* である必要があります

と *EC2 instance type*

それぞれ。詳細については、プラットフォームのドキュメントを参照してください。

プラットフォームを明示的に選択しない場合、現在デフォルトは aws です。

注 TPA は、さまざまなプラットフォーム上のインスタンスを使用したクラスターの作成を完全にサポートしていますが、`tpaexec configure` は現在このような構成を生成できません。`config.yml` を編集して、複数のプラットフォームを指定する必要があります。

所有者

`--owner <name>` を指定して、クラスターを何らかのプラットフォーム固有の手段 AWS タグなどで関連付けるには、クラスターに責任を負う人の名前を関連付けます。これは、クラウドプラットフォームにとって特に重要です。デフォルトでは、所有者は `tpaexec provision` を実行しているユーザーのログイン名に設定されます。

あなたのイニシャル、または「名、姓」、またはあなたを一意に識別するその他のものを使用できます。

リージョン

`--region <region>` を指定してリージョンを選択します。

このオプションは、クラウドプラットフォームの場合にのみ意味があります。AWS のデフォルトは eu-west-1 です。

注 TPA は、マルチプルのリージョンにまたがるクラスターの作成を完全にサポートしていますが、`tpaexec configure` は現在このような構成を生成できません。`config.yml` を編集して、複数のリージョンを指定する必要があります。

ネットワーク構成

注釈

これらのオプションは、TPA が既存のサーバーのネットワーク構成を変更しない「ベア」プラットフォームでは意味がありません。

`--network 192.168.0.0/16` を指定して、別のネットワークからサブネットを割り当てます。AWS クラスターでは、これは VPC CIDR に対応します。詳細については、[awscli を使用してボリューム属性を変更する](#) ドキュメントを参照してください。

`--subnet-prefix 26` を指定して、別のサイズのサブネット、この場合は /28 ではなく /26 のサブネットを割り当てます。

注釈

「docker」プラットフォームが選択されている場合、TPA はアーキテクチャに関係なく、常にクラスター全体を単一のサブネットに配置します。このサブネットは、ここで説明するロジックに従って生成されますが、`subnet-prefix` が指定されない場合、TPA は `config.yaml` のインスタンス数を収容するのに十分な大きさのサブネットサイズを自動的に選択します。

`--exclude-subnets-from <directory>` を指定して、既存のクラスター `config.yaml` ファイルで既に使用されているサブネットを除外します。この引数は、ディレクトリごとに複数回指定できます。

インスタンスタイプ

`--instance-type <type>` を指定してインスタンスタイプを選択します。

このオプションは、クラウドプラットフォームの場合にのみ意味があります。AWS のデフォルトは `t3.micro` です。

ディスク領域

`--root-volume-size 64` を指定して、ルートボリュームのサイズを GB 単位で設定します。プラットフォームによっては、ルートボリュームに必要な最小サイズが存在する場合があります。

`--postgres-volume-size <size>` および `--barman-volume-size <size>` オプションは、Postgres と Barman の個別のボリュームをサポートするアーキテクチャとプラットフォームで、Postgres および Barman ボリュームのサイズを設定するために使用できます。

これらのオプションは、TPA がボリュームサイズを制御できない「ベア」プラットフォームでは意味がありません。

ホスト名

デフォルトでは、`tpaexec configure` は、数十の名前の事前承認リストから必要な数のホスト名をランダムに選択します。ほとんどのクラスターではこれで十分です。

`--hostnames-from <filename>` を指定して、1 行に 1 つの名前を含むファイルからホスト名を選択します。ファイルには、クラスター内のインスタンスと同じ数以上の有効なホスト名が含まれている必要があります。各行には、名前の後にオプションの IP アドレスを含めることができます。存在する場合、このアドレスは `config.yml` の対応するインスタンスの `ip_address` として設定されます。2 つの IP アドレスが存在する場合、1 つ目は `public_ip` として設定され、2 つ目は `private_ip` として設定されます。

`--hostnames-pattern '...pattern...'` を使用して、選択を `egrep` パターンに一致する行に制限します。

`--hostnames-sorted-by="--dictionary-order"` を使用して、`--random-sort` デフォルト以外の `sort(1)` オプションを選択します。

`--hostnames-unsorted` を使用して、ホスト名をまったくソートしません。この場合、ホスト名ファイルで見つかった順序で割り当てられます。これは、ホスト名ファイルが明示的に指定された場合のデフォルトです。

`--cluster-prefixed-hostnames` を使用して、各ホスト名がクラスターの名前で始まるようにします。これは、同じホストで複数の Docker クラスターを実行している場合、ホスト名の衝突を回避するのに役立ちます。

ホスト名には、文字 `a~z`、数字 `0~9`、および `-` のみを含めることができます。選択したプラットフォームによっては、FQDN である場合があります。ホスト名は小文字である必要があります。大文字は内部で小文字に変換され、`config.yml` たとえば `upstream: hostname` でのこれらのホスト名への参照は小文字バージョンを使用する必要があります。

2.2.3 ソフトウェアの選択

ディストリビューション

`--distribution <name>` を指定してディストリビューションを選択します。

選択したプラットフォームにより、どのディストリビューションが利用可能か、およびデフォルトで使用されるディストリビューションが決まります。

一般に、「Debian」、「RedHat」、「Ubuntu」、および「SLES」を使用して、適切なイメージを選択する必要があります。

このオプションは、TPA がどのディストリビューションをインストールするかを制御できない「ベア」プラットフォームでは意味がありません。

EDB リポジトリ

TPA は、サブスクリプションを介してアクセスしている EDB ソフトウェアリポジトリを有効にできます。デフォルトでは、TPA はアーキテクチャが必要とする製品リポジトリをインストールします。

TPA が EDB リポジトリを使用する方法の詳細な説明は、[Configuring EDB Repos 2.0 repositories](#) および各アーキテクチャのページで利用できます。

`--edb-repositories repository` ...を指定して、各インスタンスにインストールする EDB リポジトリの完全なリストを指定します。

このオプションは注意して使用してください。TPA は、組み合わせが適切であることを確認せずに名前付けリポジトリを構成します。

このオプションを使用するには、TPA を実行する前に `export EDB_SUBSCRIPTION_TOKEN=xxx` を行う必要があります。EDB トークンは、enterprisedb.com/repos から取得できます。

ローカルリポジトリのサポート

`--enable-local-repo` を使用して、パッケージをターゲットインスタンスにシップするローカルパッケージリポジトリを作成します。

ネットワークアクセスが制限されている環境では、代わりに `--use-local-repo-only` を使用してローカルリポジトリを作成し、ターゲットインスタンスで他のすべてのパッケージリポジトリを無効にして、パッケージがローカルリポジトリからのみインストールされるようにできます。

[Creating and using a local repository](#) に関するページに詳細が記載されています。

ソフトウェアバージョン

Postgres のフレーバーとバージョン

TPA は、PostgreSQL、EDB Postgres Extended、および EDB Postgres Advanced Server バージョン 11～17 をサポートしています。

インストールする Postgres のフレーバーまたはディストリビューションとメジャーバージョンの両方を指定する必要があります。たとえば、

- `--postgresql 14` は PostgreSQL 14 をインストールします
- `--edb-postgres-extended 15` は EDB Postgres Extended 15 をインストールします
- `--edb-postgres-advanced 15 --redwood` は、EPAS 15 を「Redwood」モードでインストールします
- `--edb-postgres-advanced 15 --no-redwood` は、非 Redwood モードで EPAS 15 をインストールします

EPAS をインストールする場合、`--redwood` または `--no-redwood` モードで動作するかどうか、つまり Oracle 互換性機能を有効にするか無効にするかを指定する必要があります。

EDB Postgres Extended または Postgres Advanced Server のインストールには、有効な [Configuring EDB Repos 2.0 repositories](#) が必要です。

パッケージバージョン

デフォルトでは、常にすべてのパッケージの最新バージョンをインストールします。これは通常、望ましい動作ですが、一部のテストシナリオでは、次のオプションのいずれかを使用して特定のパッケージバージョンを選択する必要がある場合があります。

1. `--postgres-package-version 10.4-2.pgdg90+1`
2. `--repmgr-package-version 4.0.5-1.pgdg90+1`

3. `--barman-package-version 2.4-1.pgdb90+1`
4. `--pglogical-package-version '2.2.0*'`
5. `--bdr-package-version '3.0.2*'`
6. `--pgbouncer-package-version '1.8*'`
7. `--beacon-agent-package-version 1.56.2-1`
8. `--etcd-package-version 9.8.0-1.el8`
9. `--patroni-package-version 4.0.0-1PGDG.rhel8` 10. `--pem-server-package-version 9.7.0-1.el9` 11. `--pem-agent-package-version 9.7.0-1.el9` 12. `--pg-backup-api-package-version 2.0.0-1.el8` 13. `--pgd-proxy-package-version 5.0.0-1` 14. `--pgdcli-package-version 5.6.1`

`apt` または `yum` が受け入れるバージョン指定子を使用できます。

バージョンが一致しない場合は、* ワイルドカードを追加してみてください。これは、パッケージバージョンに `2:...` のようなエポック修飾子がある場合に必要になることがよくあります。

`--extra-packages p1 p2 ...` または `--extra-postgres-packages p1 p2 ...` を指定して、追加パッケージをインストールすることもできます。前者はシステムパッケージとともにインストールするパッケージをリストし、後者は `postgres` パッケージとともに後でインストールするパッケージをリストします。前者のリストで `Postgres` に依存するパッケージを記載すると、`Postgres` がまだインストールされていないため、インストールは失敗します。引数は変更せずにインストールのためにパッケージマネージャーに渡されます。

`--extra-optional-packages p1 p2 ...` オプションは `--extra-packages` と同様に動作しますが、名前付けたパッケージをインストールできない場合、これはエラーではありません。

ワイルドカードの使用に関する既知の問題

`config.yml` に永続的に追加する場合、* `_package_version` でワイルドカードを使用すると、RHEL 8 以降のノードまたは Rocky Linux などの `dnf` を使用する派生 OS で `tpaexec deploy` の実行中に、インストールされたソフトウェアの予期しない更新が発生する可能性があることに注意してください。既にパッケージがインストールされている既存のクラスターで `deploy` が実行されている場合、`ansible` は完全なパッケージバージョンと一致しない場合があります。たとえば、`bdr_package_version` の値が `3.6*` に設定されている場合、`ansible` はこれを PGD のインストールされたバージョンと照合できず、パッケージがインストールされていないと仮定し、次のパッケージの利用可能な最新バージョンをインストールしようとします。構成済みのリポジトリ内の同じ名前、例 3.7.

私たちはこの制限を `ansible dnf` モジュールのバグとして認識しており、TPA の将来のリリースでこれに対処したいと考えています。

ソースからのビルドとインストール

`--install-from-source postgres` を指定すると、`Postgres` はパッケージからインストールされるのではなく、`git` リポジトリからビルドおよびインストールされます。デフォルトでは、これは適切な `REL_nnn_STABLE` ブランチをビルドします。

`--install-from-source postgres bdr5` を使用してソースから両方のコンポーネントをビルドおよびインストールするか、`--install-from-source bdr5` を使用して `Postgres` のパッケージを使用できますが、

ソースから PGD v5 をビルドおよびインストールします。デフォルトでは、これは PGD の main ブランチをビルドします。

別のブランチをビルドするには、対応する引数に:branchname を追加します。例、`--install-from-source 2ndqpostgres:dev/xxx`、または `pglogical:bug/nnnn`。

代わりにソースインストールに置き換えることを選択したパッケージに依存するパッケージをインストールできない場合があります。たとえば、PGD v3 パッケージは pglogical v3 パッケージに依存しているため、ソースリポジトリから pglogical をインストールし、パッケージから PGD をインストールすることはできません。同様に、ソースから Postgres をインストールし、パッケージから pglogical をインストールすることはできません。

2.2.4 オーバーライド

オプションで `--overrides-from a.yml ...` を指定して、生成された `config.yml` にマージする設定を含む 1 つ以上の YAML ファイルをロードできます。

ここで指定されたファイルは、最初に Jinja2 テンプレートとして展開され、結果は YAML データ構造としてロードされ、`config.yml` アーキテクチャとプラットフォームのデフォルトとコマンドラインからの引数で構成される引数に再帰的にマージされます。このプロセスは、指定された追加オーバーライドファイルごとに繰り返されます。これは、1 つのファイルで定義された設定が後続のファイルに可視されることを意味します。

たとえば、オーバーライドファイルには次のものが含まれる場合があります。

```
cluster_tags:
  some_tag: "{{ lookup(env, SOME_ENV_VAR) }}"

cluster_vars:
  synchronous_commit: remote_write
  postgres_conf_settings:
    effective_cache_size: 4GB
```

これらの設定は、`config.yml` にある `cluster_tags` および `cluster_vars` を強化します。設定は再帰的にマージされるため、`cluster_tags` には、デフォルトの Owner タグと `some_tag` の両方が含まれることになります。同様に、`effective_cache_size` 設定はその変数をオーバーライドし、他の `postgres_conf_settings` 存在する場合は影響を受けません。つまり、`config.yml` で特定のサブキーを設定またはオーバーライドすることはできませんが、`cluster_tags` またはその他のハッシュを完全に空にしたり置き換えたりすることはできません。

マージはハッシュ構造にのみ適用されるため、このメカニズムを使用して `config.yml` 内の `instances` のリストを変更することはできません。環境によくある設定で `cluster_vars` および `instance_defaults` を強化するのが最も役立ちます。

とはいえ、このメカニズムはいかなる制限も強制するものではないため、十分な注意を払ってください。オーバーライドを含む場合とない場合の 2 つの構成を生成し、2 つの `config.yml` ファイルを比較して、すべてのオーバーライドの効果を理解することをお勧めします。

2.2.5 Ansible Tower

`--use-ansible-tower` および `--tower-git-repository` オプションを使用して、Ansible Tower での展開に適したクラスターを作成します。詳細については、*TPA and Ansible Tower/Ansible Automation Platform* を参照してください。

2.2.6 ビーコンエージェント

`--enable-beacon-agent` および `--beacon-agent-project-id` オプションを使用してビーコンエージェントをインストールします。これにより、EDB Postgres AI コンソールでクラスターを表示できます。詳細は、*Configuring the beacon agent* を参照してください。

2.2.7 Git リポジトリ

デフォルトでは、クラスターにちなんで名前付けられた初期ブランチで git リポジトリが作成され、コミットメッセージで使った `configure` オプションを使用して単一のコミットが作成されます。`$PATH` に `git` がない場合、`tpaexec` はエラーを発生させませんが、リポジトリは作成されません。git リポジトリの作成を抑制するには、`--no-git` オプションを使用します。Ansible Tower クラスターでは、git リポジトリが必要で、`tpaexec provision` によってまだ存在しない場合は後で作成されることに注意してください。

2.2.8 ボールトパスワードのキーリングバックエンド

TPA は、クラスター固有の `ansible vault` パスワードを生成します。このパスワードは、クラスター用に生成された他の機密変数、postgres ユーザーパスワード、barman ユーザーパスワードなどを暗号化するために使用されます。

キーリングバックエンド `system` は、`gnome-keyring` や `secret-tool` を含む Python キーリングモジュールでサポートされているバックエンドのリストから、システムで最適なキーリングバックエンドを活用します。

デフォルトでは、新しいクラスターの `system` キーリングを使用してボールトパスワードを保存します。`provision` の前、`config.yml` ファイルの `keyring_backend: system` を削除すると、以前のデフォルトに戻してボールトのパスワードをプレーンテキストファイルに保存します。

`keyring_backend: system` を使用すると、ボールトパスワードの一意のストレージ名を保存するために使用される `config.yml` に `vault_name` エントリーも生成されます。TPA はデフォルトで UUID を生成しますが、命名スキーム要件はありません。

注 同じ `cluster_name` を使用する複数のクラスターで `keyring_backend: system` と同じベースの `config.yml` ファイルを使用する場合、構成ファイルを別の場所にコピーすることにより、値ペア `vault_name`、`cluster_name` がクラスターコピーごとに一意であることを確認します。

注 `keyring_backend: system` を使用し、既にプロビジョニングされたクラスターフォルダーを別の tpa ホストに移動する場合、新しいマシンのシステムキーリングに関連するボールトパスワードをエクスポートしていることを確認してください。vault パスワードは `tpaexec show-vault <cluster_dir>` を介して表示できます。

2.2.9 セキュリティ基準へのコンプライアンス

--compliance stig または --compliance cis オプションを使用して、STIG または CIS 標準に準拠するのに適した構成でクラスターを生成します。詳細は、[セキュリティ基準へのコンプライアンス](#) を参照してください。これらのオプションは、クラスターが関連する標準を満たすことを保証するものではないことに注意してください。これらは、TPA によって制御可能な標準のこれらの側面に準拠するように設計された構成を TPA に生成するだけです。

2.2.10 例

次のコマンドを実行するとどうなるか見てみましょう。

```
[tpa]$ tpaexec configure ~/clusters/speedy --architecture M1 \
--distribution Debian \
--platform aws --region us-east-1 --network 10.33.0.0/16 \
--instance-type t2.medium --root-volume-size 32 \
--postgres-volume-size 64 --barman-volume-size 128 \
--postgresql 14 \
--failover-manager repmgr
[tpa]$
```

出力はないため、エラーはありません。クラスターディレクトリが作成され、データが設定されました。

```
$ ls -lh ~/clusters/speedy/
total 8.0K
drwxrwxr-x 2 haroon haroon 4.0K Aug 17 02:33 commands
-rw-rw-r-- 1 haroon haroon 1.5K Aug 17 02:33 config.yml
lrwxrwxrwx 1 haroon haroon 53 Aug 17 02:33 deploy.yml -> /home/haroon/tpa/
↪architectures/M1/deploy.yml
```

クラスター構成は config.yml にあり、その近傍は直接対話する必要のないアーキテクチャ固有のサポートファイルへのリンクです。構成は次のとおりです。

```
- --
architecture: M1
cluster_name: speedy
cluster_tags: {}

keyring_backend: system
vault_name: cfae3da3-ec00-46cd-ab05-e153f1c788db

cluster_rules:

- cidr_ip: 0.0.0.0/0
  from_port: 22
  proto: tcp
  to_port: 22
```

(次のページに続く)

(前のページからの続き)

```
- cidr_ip: 10.33.120.80/28
  from_port: 0
  proto: tcp
  to_port: 65535
ec2_ami:
  Name: debian-11-amd64-20240104-1616
  Owner: 136693071363
ec2_instance_reachability: public
ec2_vpc:
  us-east-1:
    Name: Test
    cidr: 10.33.0.0/16

cluster_vars:
  edb_repositories: []
  failover_manager: repmgr
  postgres_flavour: postgresql
  postgres_version: 14
  preferred_python_version: python3

locations:

- Name: main
  az: us-east-1a
  region: us-east-1
  subnet: 10.33.120.80/28

instance_defaults:
  default_volumes:
    - device_name: root
      encrypted: true
      volume_size: 32
      volume_type: gp2
    - device_name: /dev/sdf
      encrypted: true
      vars:
        volume_for: postgres_data
        volume_size: 64
        volume_type: gp2
  platform: aws
  type: t2.medium
  vars:
    ansible_user: admin
```

(次のページに続く)

(前のページからの続き)

```
instances:

- Name: uproar
  backup: kinsman
  location: main
  node: 1
  role:
    - primary

- Name: unravel
  location: main
  node: 2
  role:
    - replica
  upstream: uproar

- Name: kinsman
  location: main
  node: 3
  role:
    - barman
    - log-server
    - witness
  upstream: uproar
  volumes:
    - device_name: /dev/sdf
      encrypted: true
      vars:
        volume_for: barman_data
      volume_size: 128
      volume_type: gp2
```

次のステップは、*tpaexec provision* を実行することです

または、**クラスター構成** の構成をカスタマイズする方法または *Instance configuration* の構成方法の詳細を参照してください。

2.3 tpaexec deploy

展開は、クラスターのサーバーに Postgres およびその他のソフトウェアをインストールおよび構成するプロセスです。これには、レプリケーション、バックアップなどの設定が含まれます。

展開ステージの最後に、Postgres は、repmgr、Barman、pgbouncer などの他のコンポーネントとともに実行されます選択したアーキテクチャに応じて。

2.3.1 前提条件

`tpaexec deploy` を実行する前に、**クラスター構成** を実行してクラスター構成を生成し、`tpaexec provision` でサーバーをプロビジョニングしておく必要があります。

Configuring EDB Repos 2.0 repositories を使用している場合は、展開の前に `export EDB_SUBSCRIPTION_TOKEN=xxx` を行う必要があります。これを忘れた場合、エラーメッセージがすぐに思い出させます。

2.3.2 クイックスタート

```
[tpa]$ tpaexec deploy ~/clusters/speedy -v
Using /opt/EDB/TPA/ansible/ansible.cfg as config file

PLAY [Basic initialisation and fact discovery] *****
-> *****
...

PLAY [Set up TPA cluster nodes] *****
*****

...

PLAY RECAP *****
-> *****

zealot           : ok=281  changed=116  unreachable=0    failed=0
keeper           : ok=284  changed=96   unreachable=0    failed=0
quaver           : ok=260  changed=89   unreachable=0    failed=0
quavery          : ok=260  changed=88   unreachable=0    failed=0
quirk            : ok=262  changed=100  unreachable=0    failed=0

real    7m1.907s
user    3m2.492s
sys     1m5.318s
```

このコマンドは、大量の出力を生成し、時間がかかる場合があります主に、`tpaexec` を実行しているホストとクラスター内のホスト間の遅延、およびインスタンスがインストールする必要があるパッケージをダウンロードするのにかかる時間によって異なります。。デプロイメント中に少なくとも 1 つの `-v` を使用することをお勧めします。出力は、クラスターディレクトリの `ansible.log` にも記録されます。

ホスト、タスク、および変更されたタスクの正確な数は、もちろん異なる場合があります。

`apply` コマンドは自分自身にオプションを取りません。クラスター名の後に指定したオプションは、変更せずに `Ansible -v` などに渡されます。

Ansible に慣れている人は、時折赤い「failed」タスク出力がスクロールするのを心配するかもしれませんが。プロセスがその後すぐに停止しない場合、エラーは重要ではなく、コードは自動的に回復しますのでご安心ください。

デプロイが完了したら、`tpaexec test` を実行してインストールを確認できます。

2.3.3 選択的な展開

`deploy_hosts` をインスタンス名のコンマ区切りリストに設定することにより、展開をホストのサブセットに制限できます。

```
[tpa]$ tpaexec deploy ~/clusters/speedy -v -e deploy_hosts=keeper,quaver
```

これにより、指定されたインスタンスで展開が実行されますが、最初に他のホストでいくつかのタスクを実行して、クラスターの状態に関する情報を収集します。

`deploy_hosts` 設定は、TPA がサポートしていない Ansible の `--limit` オプションを使用することに代わる推奨される代替品です。

2.3.4 deploy.yml

展開プロセスはアーキテクチャ固有です。次に、さまざまな *Instance configuration* の概要を示します。Ansible プレイブックに精通している場合は、`tpaexec` がクラスターのインスタンスにさまざまなロールを適用するのを理解することができます。

`config.yml` とは異なり、`deploy.yml` は編集するように設計されておらず、通常はアーキテクチャディレクトリへのリンクです。独自の Ansible タスクを実行するようにデプロイメントプロセスを拡張する場合でも、*TPA hooks* によってこれを行う必要があります。これにより、特定のアーキテクチャ内の将来の実装の変更から保護されます。

2.4 tpaexec provision

プロビジョニングは、クラスターに必要なインスタンスとその他のリソースを作成します。

このプロセスの正確な詳細は、クラスターの構成中に選択したアーキテクチャたとえば *MI* とプラットフォームたとえば *awscli* を使用してボリューム属性を変更するの両方に依存します。

プロビジョニングステージの最後に、基本的なオペレーティングシステムがインストールされた必要な数のインスタンスができます。TPA は、ssh 経由でアクセスできますルートに `sudo` を使用します。

2.4.1 前提条件

クラスターをプロビジョニングする前に、*クラスター構成* を使用してクラスター構成を生成する必要があります

そして、必要に応じて `config.yml` を編集して構成を微調整します。

プラットフォーム依存の追加の手順が必要になる場合があります。たとえば、AWS API アクセスキーペアを取得して EC2 インスタンスをプロビジョニングするか、LXD または Docker をセットアップしてコンテナをプロビジョニングする必要があります。詳細については、プラットフォームのドキュメントを参照してください。

2.4.2 クイックスタート

```
[tpa]$ tpaexec provision ~/clusters/speedy

PLAY [Provision cluster] *****
-> *****
...

TASK [Set up EC2 instances] *****
-> *****
changed: [localhost] => (item=us-east-1:uproar)
changed: [localhost] => (item=us-east-1:unravel)
changed: [localhost] => (item=us-east-1:kinsman)
...

TASK [Generate ssh_config file for the cluster] *****
-> *****
changed: [localhost]

PLAY RECAP *****
-> *****
localhost : ok=163 changed=35 unreachable=0 failed=0
->skipped=44 rescued=0 ignored=2

real    4m42.726s
user    0m39.101s
sys     0m15.687s
```

このコマンドは、大量の出力を生成しますさらに詳細な出力が必要な場合は、コマンドに `-v`、`-vv` などを追加します。出力は、クラスターディレクトリの `ansible.log` にも記録されます。これは、環境変数 `ANSIBLE_LOG_PATH` を目的のログファイルのパスと名前に設定することによりオーバーライドできます。

エラーなしで完了した場合は、`tpaexec deploy` の実行に進んでソフトウェアのインストールと構成を行うことができます。

2.4.3 オプション

クラウドインスタンスをプロビジョニングするときは、インスタンスが、それらを担当する人間まで直接追跡可能であることを確認することが特に重要です。デフォルトでは、TPA は `tpaexec provision` を実行しているユーザーのログイン名が所有するものとして EC2 インスタンスにタグ付けします。

名前を変更するには `--owner <name>` を指定しますたとえば、ユーザー名が `postgres` や `ec2-user` のような汎用的なものである場合。イニシャル、「名、姓」、またはその他を一意に識別するために使用できます。

指定した他のオプションは Ansible に渡されます。

2.4.4 インスタンスへのアクセス

プロビジョニングが完了すると、インスタンスに SSH 接続できるようになります。インスタンスが起動して SSH ホストキーをインストールできるようにするために少し遅延した後。上記の出力に示されているように、tpaexec は使用できる ssh_config ファイルを生成します。

```
[tpa]$ cd ~/clusters/speedy
[tpa]$ cat ssh_config
Host *
    Port 22
    IdentitiesOnly yes
    IdentityFile "id_speedy"
    UserKnownHostsFile known_hosts tpa_known_hosts
    ServerAliveInterval 60

Host uproar
    User admin
    HostName 3.88.255.205

Host unravel
    User admin
    HostName 54.80.99.142

Host kinsman
    User admin
    HostName 54.165.229.179
```

ホストにログインするには、コマンド `ssh -F ssh_config` の後にホスト名を使用します。例 `ssh -F ssh_config uproar`

プロビジョニングの直後に `tpaexec deploy` を実行できます。インスタンスが起動するまで、必要なだけ待機します。インスタンスが起動するのを待ったり、展開を開始する前に ssh 接続したりする必要はありません。

2.4.5 生成されたファイル

プロビジョニングプロセス中に、多数の新しいファイルがクラスターディレクトリに作成されます。

```
[tpa]$ ls ~/clusters/speedy
total 240
-rw-r--r-- 1 ams ams 193098 Aug  4 17:59 ansible.log
drwxr-xr-x 2 ams ams  4096 Aug  4 17:38 commands
-rw-r--r-- 1 ams ams  1442 Aug  4 17:54 config.yml
lrwxrwxrwx 1 ams ams    51 Aug  4 17:38 deploy.yml ->
/opt/EDB/TPA/architectures/M1/deploy.yml
drwxr-xr-x 2 ams ams  4096 Aug  4 17:38 hostkeys
-rw----- 1 ams ams  1675 Aug  4 17:38 id_speedy
-rw----- 1 ams ams  1438 Aug  4 17:38 id_speedy.ppk
-rw-r--r-- 1 ams ams   393 Aug  4 17:38 id_speedy.pub
```

(次のページに続く)

(前のページからの続き)

```
drwxr-xr-x 4 ams ams 4096 Aug 4 17:50 inventory
- rw-r--r-- 1 ams ams 2928 Aug 4 17:50 tpa_known_hosts
- rw-r--r-- 1 ams ams 410 Aug 4 17:50 ssh_config
- rw-r--r-- 1 ams ams 3395 Aug 4 17:59 vars.json
drwxr-xr-x 2 ams ams 4096 Aug 4 17:38 vault
```

私たちはすでに `ssh config` ファイルを研究しました。これは、`id *files` クラスター用に生成された SSH キーペアと `tpa_known_hosts` インスタンスにインストールされた `hostkeys/` の署名を参照します。

`vars.json` ファイルは、`--cached` の後続の呼び出しで `tpaexec provision` によって使用される場合があります。

`inventory/` ディレクトリには、静的および動的インベントリーファイルと、`config.yml` のグループおよびホスト変数定義が含まれています。

```
[tpa]$ cat inventory/00-speedy
[tag_Cluster_speedy]
uproar ansible_host=3.88.255.205 node=1 platform=aws
unravel ansible_host=54.80.99.142 node=2 platform=aws
kinsman ansible_host=54.165.229.179 node=3 platform=aws

[tpa]$ cat inventory/group_vars/tag_Cluster_speedy/01-speedy.yml
cluster_name: speedy
cluster_tag: tag_Cluster_speedy
edb_repositories: []
failover_manager: repmgr
keyring_backend: system
postgres_flavour: postgresql
postgres_version: 14
preferred_python_version: python3
ssh_key_file: id_speedy
tpa_version: v23.33-24-g4c0909d1

[tpa]$ cat inventory/host_vars/kinsman/01-instance_vars.yml
ansible_user: admin
location: main
region: us-east-1
role:

- barman

- log-server

- witness
upstream: uproar
volumes:
```

(次のページに続く)

(前のページからの続き)

```
- device: /dev/xvda  
  
- device: /dev/sdf  
  volume_for: barman_data
```

config.yml の変数を変更し、プロビジョニングを再実行すると、これらのファイルが更新されます。構成を変更しない場合、何も起こりません。config.yml に新しいインスタンスを追加して再実行すると、既存のインスタンスに影響を与えることなく新しいインスタンスが表示されます。

2.5 tpaexec test

次に、展開されたクラスターに対してアーキテクチャ固有のテストを実行して、インストールを確認します。このステージの最後に、完全に機能するクラスターが完成します。

tpaexec test を実行する前に、tpaexec configure、tpaexec provision、および tpaexec deploy を正常に実行している必要があります。

2.5.1 クイックスタート

```
[tpa]$ tpaexec test ~/clusters/speedy -v
```

出力は、クラスターディレクトリの ansible.log に再びログに記録されます。

このコマンドが成功した場合、クラスターは機能します。

おめでとうございます。

第3章 Architectures

3.1 PGD-Always-ON

注釈

このアーキテクチャは、Postgres Distributed 5 専用です。PGD 4 または 3.7 が必要な場合は、[BDR-Always-ON](#) を使用してください。

Always-ON 構成の EDB Postgres Distributed 5 は、テストおよび運用環境での使用に適しています。

このアーキテクチャには、EDB サブスクリプションが必要です。すべてのソフトウェアは [Configuring EDB Repos 2.0 repositories](#) からソースされます。

3.1.1 クラスター構成

構成オプションの概要

このアーキテクチャの `tpaexec configure` の呼び出し例を以下に示します。

```
tpaexec configure ~/clusters/pgd-ao \
  --architecture PGD-Always-ON \
  --edb-postgres-extended 15 \
  --platform aws --instance-type t3.micro \
  --distribution Debian \
  --pgd-proxy-routing global \
  --location-names dc1 dc2 dc3 \
  --witness-only-location dc3 \
  --data-nodes-per-location 2
```

`help` コマンドを使用して、使用可能なすべてのオプションをリストできます。

```
tpaexec configure --architecture PGD-Always-ON --help
```

以下の表では、PGD-Always-ON の必須オプションと追加の重要なオプションについて説明しています。オプションの詳細については、次のセクションで説明します。

必須オプション

Options	Description
<code>--architecture (-a)</code>	<code>`PGD-Always-ON`</code> に設定する必要があります
Postgres flavour and version (e.g. <code>--postgresql 15</code>)	有効な <i>Postgres</i> のフレーバーとバージョン。
<code>--pgd-proxy-routing</code>	<code>`global`</code> または <code>`local`</code> のいずれかである必要があります。

追加オプション

Options	Description	Behaviour if omitted
<code>--platform</code>	<code>aws</code> 、 <code>docker</code> 、 <code>`bare`</code> のいずれか。	Defaults to <code>aws</code> .
<code>--location-names</code>	場所名のスペース区切りのリスト。場所の数は、指定された名前数と同じです。	TPA will configure a single location with three data nodes.
<code>--witness-only-location</code>	ロケーション名は、 <code>`location-names`</code> のメンバーである必要があります。	No witness-only location is added.
<code>--data-nodes-per-location</code>	各場所のデータノードの数は、少なくとも 2 である必要があります。	Defaults to 3.
<code>--add-proxy-nodes-per-location</code>	各場所のプロキシノードの数。	PGD-proxy will be installed on each data node.
<code>--enable-camo</code>	各ロケーションに 2 つのデータノードを CAMO パートナーとして設定します。	CAMO will not be enabled.
<code>--bdr-database</code>	レプリケーションに使用されるデータベースの名前。	Defaults to <code>bdrdb</code> .
<code>--enable-pgd-probes</code>	<code>`health/is-ready`</code> などの pgd-プロキシの http(s) API エンドポイントを有効にして、プロキシの状態をプローブできるようにします。	Disabled by default.
<code>--proxy-listen-port</code>	プロキシノードがトラフィックを書き込みリーダーにルーティングするポート。	Defaults to 6432
<code>--proxy-read-only-port</code>	プロキシノードが読み取り専用トラフィックをシャドウノードにルーティングするポート。	Defaults to 6433

PGD-Always-ON 構成の詳細

PGD-Always-ON クラスターは、多くの場所、好ましくは奇数で構成され、それぞれが同じ数のデータノード、やはり好ましくは奇数です。 `--location-names` を指定しない場合、デフォルトでは、3 つのデータノードを持つ単一の場所を使用します。

クラスターの場所名は `--location-names dc1 dc2 ...` として指定されます。ロケーションは、この定義がユースケースにとって意味のある方法で、レベルの冗長性を提供する独立したデータセンターを表します。たとえば、AWS リージョン、独自のデータセンター、またはサーバーがホストされている場所を識別するためのその他の指定。

`--data-nodes-per-location N` を使用して、各場所のデータノードの数を指定します。最小数は 2 で、デフォルトは 3 です。

ロケーションごとに偶数のデータノードを指定すると、TPA は各ロケーションに追加の監視ノードを自動的に追加します。これにより、コストを削減しながら、信頼できるコンセンサスを確立する機能が維持されます。監視のハードウェア要件は、データノードと比較して最小限です。

2つのロケーションのみからなるクラスターは、いずれかのロケーションに障害が発生した場合、グローバルコンセンサスを確立する能力を完全に失います。3 番目の監視専用の場所を追加することをお勧めします。これにはデータノードは含まず、監視ノードのみが含まれ、これもコンセンサスを確実に確立するために使用されます。`--witness-only-location loc` を使用して、いずれかの場所を監視者として指定します。

デフォルトでは、すべてのデータノードすべての場所で、接続ルーティングのために PGD-プロキシも実行されます。代わりに別の PGD-プロキシインスタンスを作成するには、`--add-proxy-nodes-per-location 3` または追加する数のプロキシを使用します。

ユースケースによっては、`--pgd-proxy-routing local` または `global` を指定して、PGD-Proxy が書き込みリーダーに接続をルーティングする方法を構成する必要があります。ローカルルーティングは、すべての PGD-プロキシを独自の場所内の書き込みリーダーにルートさせますジオシャーディングアプリケーションに適しています。グローバルルーティングは、すべてのプロキシを単一の書き込みリーダーにルートさせます。この書き込みリーダーは、すべての場所で使用可能なすべてのデータノードの間で選択されます。

オプションで `--bdr-database dbname` を指定して、BDR が有効になったデータベースの名前を設定できますデフォルト `bdrdb`。

オプションで `--enable-camo` を指定して、各リージョンの 2 つのデータノードを CAMO パートナーとして設定できます。

オプションで `--enable-pgd-probes [{http, https}]` を指定して、プロキシの状態を簡単にプローブできる http API エンドポイントを有効にすることができます。

クラスター構成 で説明されるオプションを指定することもできます。

3.2 PGD Lightweight

注釈

このアーキテクチャは Postgres Distributed 5 専用です。PGD 4 または 3.7 が必要な場合は、*BDR-Always-ON* を使用してください。

軽量構成の EDB Postgres Distributed 5 は、テストおよび実稼働での使用に適しています。

このアーキテクチャには、EDB サブスクリプションが必要です。すべてのソフトウェアは *Configuring EDB Repos 2.0 repositories* からソースされます。

3.2.1 クラスター構成

構成オプションの概要

このアーキテクチャの `tpaexec configure` の呼び出し例を以下に示します。

```
tpaexec configure ~/clusters/pgd-lw \
  --architecture Lightweight \
  --edb-postgres-extended 15 \
  --platform aws --instance-type t3.micro \
  --distribution Debian \
  --location-names main dr \
```

`help` コマンドを使用して、使用可能なすべてのオプションをリストできます。

```
tpaexec configure --architecture Lightweight --help
```

以下の表では、PGD-Always-ON の必須オプションと追加の重要なオプションについて説明しています。オプションの詳細については、次のセクションで説明します。

必須オプション

Options	Description
<code>--architecture (-a)</code>	<code>`Lightweight`</code> に設定する必要があります
Postgres flavour and version (e.g. <code>--postgresql 15</code>)	有効な <i>Postgres</i> のフレーバーとバージョン。

追加オプション

Options	Description	Behaviour if omitted
<code>--platform</code>	<code>aws</code> 、 <code>docker</code> 、 <code>`bare`</code> のいずれか。	Defaults to <code>aws</code> .
<code>--location-names</code>	場所名のスペース区切りのリスト。場所の数は、指定された名前の数と同じです。	TPA will configure a single location with three data nodes.
<code>--add-proxy-nodes-per-location</code>	各場所のプロキシノードの数。	PGD-proxy will be installed on each data node.
<code>--bdr-database</code>	レプリケーションに使用されるデータベースの名前。	Defaults to <code>bdrdb</code> .
<code>--enable-pgd-probes</code>	<code>`health/is-ready`</code> などの pgd-プロキシの http(s) API エンドポイントを有効にして、プロキシの状態をプローブできるようにします。	Disabled by default.
<code>--proxy-listen-port</code>	プロキシノードがトラフィックを書き込みリターにルーティングするポート。	Defaults to 6432

次のページに続く

表 4 – 前のページからの続き

Options	Description	Behaviour if omitted
<code>--proxy-read-only-port</code>	プロキシノードが読み取り専用トラフィックを シャドウノードにルーティングするポート。	Defaults to 6433

軽量構成の詳細

PGD Lightweight クラスタは2つのロケーションで構成され、2つのノードを含むプライマリアクティブロケーションと、単一のノードを含むディザスタリカバリー dr ロケーション。

クラスタの場所名は`--location-names primary dr`として指定されます。ロケーションは、この定義がユースケースにとって意味のある方法で、レベルの冗長性を提供する独立したデータセンターを表します。たとえば、AWS リージョン、独自のデータセンター、またはサーバーがホストされている場所を識別するためのその他の指定。

デフォルトでは、すべてのデータノードすべての場所で、接続ルーティングのためにPGD-プロキシも実行されます。代わりに別のPGD-プロキシインスタンスを作成するには、`--add-proxy-nodes-per-location 3`または追加する数のプロキシを使用します。

グローバルルーティングは、すべてのプロキシを単一の書き込みリーダーにルートさせます。この書き込みリーダーは、すべての場所で使用可能なすべてのデータノードの間で選択されます。

オプションで`--bdr-database dbname`を指定して、BDR が有効になったデータベースの名前を設定できますデフォルト `bdrdb`。

オプションで`--enable-pgd-probes [{http, https}]`を指定して、プロキシの状態を簡単にプローブできる http API エンドポイントを有効にすることができます。

クラスタ構成 で説明されるオプションを指定することもできます。

3.3 BDR-Always-ON

Always-ON 構成の EDB Postgres 分散 3.7 または 4。テストおよびプロダクションでの使用に適しています。

このアーキテクチャには、EDB サブスクリプションが必要です。すべてのソフトウェアは [Configuring EDB Repos 2.0 repositories](#) からソースされます。

BDR-Always-ON アーキテクチャには、`--layout configure` オプションで選択できる 4 つのバリエーションがあります。

1. ブロンズ 2 × bdr+プライマリ、bdr+witness、barman、2 × harp-プロキシ
2. シルバー: ブロンズ、bdr + Witness が bdr +プライマリに昇格し、バーテンダーが別の場所に移動した
3. gold それぞれ 2 × bdr+プライマリ、2 × harp-プロキシ、および barman を備えた 2 つの対称的な場所。プラス 3 番目の場所に bdr + Witness
4. プラチナ: ゴールド、ただし 1 つの bdr+読み取り専用ロジカルスタンバイが各メインの場所に追加されます

EDB の Postgres DistributedAlways On アーキテクチャ [whitepaper](#) を確認できます。

詳細なレイアウト図については、

このアーキテクチャは、PGD バージョン 3.7 および 4 で使用することを対象としています。

3.3.1 クラスター構成

構成オプションの概要

このアーキテクチャの `tpaexec configure` の呼び出し例を以下に示します。

```
tpaexec configure ~/clusters/bdr \
  --architecture BDR-Always-ON \
  --platform aws --region eu-west-1 --instance-type t3.micro \
  --distribution Debian \
  --edb-postgres-advanced 14 --redwood
  --layout gold \
  --harp-consensus-protocol bdr
```

`help` コマンドを使用して、使用可能なすべてのオプションをリストできます。

```
tpaexec configure --architecture BDR-Always-ON --help
```

以下の表では、BDR-Always-ON の必須オプションと追加の重要なオプションについて説明します。オプションの詳細については、次のセクションで説明します。

必須オプション

Option	Description
<code>--architecture (-a)</code>	<code>`BDR-Always-ON`</code> に設定する必要があります。
Postgres flavour and version (e.g. <code>--postgresql 14</code>)	有効な Postgres のフレーバーとバージョン 。
<code>--layout</code>	<code>bronze</code> 、 <code>silver</code> 、 <code>gold</code> 、 <code>`platinum`</code> のいずれか。
<code>--harp-consensus-protocol</code>	<code>bdr</code> 、 <code>`etcd`</code> のいずれか。

追加オプション

Option	Description	Behaviour if omitted
<code>--platform</code>	<code>aws</code> 、 <code>docker</code> 、 <code>`bare`</code> のいずれか。	Defaults to <code>aws</code> .
<code>--enable-camo</code>	各ロケーションに 2 つのデータノードを CAMO パートナーとして設定します。	CAMO will not be enabled.

次のページに続く

表 6 – 前のページからの続き

Option	Description	Behaviour if omitted
<code>--bdr-database</code>	レプリケーションに使用されるデータベースの名前。	Defaults to <i>bdrdb</i> .
<code>--enable-harp-probes</code>	<code>health/is-ready`などの harp の http API エンドポイントを有効にして、harp の状態をプローブできるようにします。</code>	Disabled by default.

BDR-Always-ON 構成の詳細

`--layout layoutname` を指定して、サポートされている BDR ユースケースバリエーションのいずれかを設定する必要があります。許可される引数は、ブロンズ、シルバー、ゴールド、およびプラチナです。ブロンズ、ゴールド、およびプラチナレイアウトには、Raft コンセンサスマジョリティの奇数のノードを保証する PGD 監視ノードがあります。Witness ノードはデータレプリケーションに参加しません。

`--harp-consensus-protocol protocolname` を指定する必要があります。サポートされているプロトコルは *bdr* および *etcd* です。詳細は、[Configuring HARP](#) を参照してください。

オプションで `--bdr-database dbname` を指定して、PGD が有効になっているデータベースの名前を設定できますデフォルト *bdrdb*。

オプションで `--enable-camo` を指定して、各リージョンの PGD プライマリインスタンスのペアを互いの CAMO パートナーに設定できます。

オプションで `--enable-harp-probes [{http, https}]` を指定して、harp の状態を簡単にプローブできる http API エンドポイントを有効にすることができます。

BDR-Always-ON アーキテクチャでは、デフォルトで HARP2 が有効になっていることに注意してください。

[クラスター構成](#) で説明されるオプションを指定することもできます。

3.4 M1

単一のプライマリノードと、バックアップとフェールオーバー管理を含む多数のスタンバイノードへの物理レプリケーションを備えた Postgres クラスター。

このアーキテクチャは、運用に適していますが、そのシンプルさと独自のコンポーネントなしで構成できる機能により、テスト、デモンストレーション、学習にも適しています。

このアーキテクチャでサブスクリプション専用の EDB ソフトウェアを選択した場合、EDB Repos 2.0 からソースされ、[Configuring EDB Repos 2.0 repositories](#) が必要です。

3.4.1 フェイルオーバー管理

M1 アーキテクチャには、常にフェールオーバーマネージャーが含まれます。サポートされているオプションは *repmgr*、EDB Failover ManagerEFM、および *Patroni* です。すべての場合、フェールオーバーマネージャーはデフォルトで構成され、プライマリが使用できなくなった場合にレプリカがプライマリに代わって昇格します。

アプリケーションのフェイルオーバー

M1 アーキテクチャは、通常、アプリケーショントラフィックをプライマリに再ルーティングする自動機能を提供しません。この機能をクラスターに追加するには、いくつかの方法があります。

TPA で

- フェールオーバーマネージャーとして `repmgr` を選択し、PgBouncer を有効にすると、`config.yml` の `cluster_vars` の下に `repmgr_redirect_pgouncer: true` ハッシュを含めることができます。これにより、`repmgr` は PgBouncer を自動的に再構成して、フェールオーバー時にトラフィックを新しいプライマリにルーティングします。
- フェールオーバーマネージャーとして `Patroni` を選択し、PgBouncer を有効にすると、`Patroni` はフェールオーバー時にトラフィックを新しいプライマリにルーティングするように PgBouncer を自動的に再構成します。
- フェールオーバーマネージャーとして EFM を選択した場合、`config.yml` の `cluster_vars` の下の `efm_conf_settings` ハッシュを使用できます `config.yml` から *configure EFM to use a virtual IP address (VIP)*。これは、常にプライマリノードにルーティングされる追加の IP アドレスです。
- クラスターとアプリケーションの間に適切なプロキシまたはロードバランサーを配置し、*TPA hooks* を使用して、選択したフェールオーバーマネージャーをフェールオーバー時に新しいプライマリへのルートで更新するように構成します。
- マルチホスト接続文字列を使用するなど、アプリケーション自身自身でフェールオーバーを処理します。

バックアップフェイルオーバー

TPA は、いかなる種類の「バックアップフェイルオーバー」も構成しません。バックアップしている Postgres ノードがダウンしている場合、ノードがオンラインに戻るまでバックアップは単に停止します。バックアップを新しいプライマリに手動で接続するには、`config.yml` を編集して `backup` ハッシュを新しいプライマリインスタンスに追加し、`tpaexec deploy` を再実行します。

3.4.2 クラスター構成

構成オプションの概要

このアーキテクチャの `tpaexec configure` の呼び出し例を以下に示します。

```
tpaexec configure ~/clusters/m1 \
  --architecture M1 \
  --platform aws --region eu-west-1 --instance-type t3.micro \
  --distribution Debian \
  --postgresql 14 \
  --failover-manager repmgr \
  --data-nodes-per-location 3
```

`help` コマンドを使用して、使用可能なすべてのオプションをリストできます。

```
tpaexec configure --architecture M1 --help
```

以下の表では、M1 の必須オプションと追加の重要なオプションについて説明します。オプションの詳細については、次のセクションで説明します。

必須オプション

Parameter	Description
<code>--architecture (-a)</code>	`M1` に設定する必要があります。
Postgres flavour and version (e.g. <code>--postgresql 15</code>)	有効な Postgres のフレーバーとバージョン 。
One of: <code>--failover-manager {efm, repmgr, patroni}</code> <code>--enable-efm` `--enable-repmgr` `--enable-patroni`</code>	Configuring EFM 、 Repmgr リダイレクト pgbouncer 、および Patroni cluster management commands からフェイルオーバーマネージャーを選択します。

追加オプション

Parameter	Description	Behaviour if omitted
<code>--platform</code>	<code>aws</code> 、 <code>docker</code> 、 <code>`bare`</code> のいずれか。	Defaults to <code>aws</code> .
<code>--location-names</code>	場所名のスペース区切りのリスト。場所の数は、指定された名前の数と同じです。	A single location called "main" is used.
<code>--primary-location</code>	プライマリサーバーが置かれる場所。`location-names` のメンバーである必要があります。	The first listed location is used.
<code>--data-nodes-per-location</code>	1 以上の数字。各場所では、1 つのノードがクラスターのプライマリノードから直接ストリーミングするように構成され、他のノードが存在する場合はそのノードからストリーミングします。	Defaults to 2.
<code>--witness-only-location</code>	ロケーション名は、`location-names` のメンバーである必要があります。この場所には、単一の監視ノードのみが設定されます。	No witness-only location is added.
<code>--single-node-location</code>	ロケーション名は、`location-names` のメンバーである必要があります。この場所には、単一のデータノードのみが設定されます。	No single-node location is added.
<code>--enable-haproxy</code>	2 つの追加ノードがロードバランサーレイヤーとして追加されます。フェールオーバーマネージャーとして Patroni でのみサポートされています。	HAproxy nodes will not be added to the cluster.

次のページに続く

表 8 – 前のページからの続き

Parameter	Description	Behaviour if omitted
<code>--enable-pgbouncer</code>	PgBouncer は、プライマリの接続をプールするように Postgres ノードで構成されます。	PgBouncer will not be configured in the cluster.
<code>--patroni-dcs</code>	patroni の分散構成ストアバックエンドを選択します。現時点では、オプションは `etcd` のみです。フェールオーバーマネージャーとして Patroni のみサポートされています。	Defaults to <i>etcd</i> .
<code>--efm-bind-by-hostname</code>	efm が IP アドレスの代わりにホスト名を使用してクラスター `bind.address` を構成するようにします。	Defaults to use IP addresses

M1 構成の詳細

クラスター構成 で説明されるオプションを指定することもできます。

3.5 PGD-S

!!!注

このアーキテクチャは、Postgres Distributed 6 専用です。PGD 5 が必要な場合は、*PGD-Always-ON* を使用してください。

このアーキテクチャには、EDB サブスクリプションが必要です。すべてのソフトウェアは *Configuring EDB Repos 2.0 repositories* から供給されています。

3.5.1 クラスター構成

構成オプションの概要

このアーキテクチャの `tpaexec configure` の呼び出し例を以下に示します。

```
tpaexec configure ~/clusters/pgd-s \
  --architecture PGD-S
  --edb-postgres-extended 15 \
  --platform aws --instance-type t3.micro \
  --distribution Debian \
```

`help` コマンドを使用して、使用可能なすべてのオプションをリストできます。

```
tpaexec configure --architecture PGD-S --help
```

以下の表では、PGD-S の必須オプションと追加の重要なオプションについて説明しています。オプションの詳細については、次のセクションで説明します。

必須オプション

Options	Description
<code>--architecture (-a)</code>	<code>`PGD-S`</code> に設定する必要があります
Postgres flavour and version (e.g. <code>--postgresql 15</code>)	有効な <i>Postgres</i> のフレーバーとバージョン。

追加オプション

Options	Description	Behaviour if omitted
<code>--platform</code>	<i>aws</i> 、 <i>docker</i> 、 <code>`bare`</code> のいずれか。	Defaults to <i>aws</i> .
<code>--bdr-database</code>	レプリケーションに使用されるデータベースの名前。	Defaults to <i>bdrdb</i> .
<code>--layout</code>	<i>standard</i> または <i>near-far</i>	Defaults to <i>standard</i>
<code>--add-subscriber-only-nodes</code>	追加する加入者専用ノードの数最大 10。	Defaults to 0
<code>--read-write-port</code>	接続マネージャーが読み取り/書き込み接続をリッスンするポート。	Left empty in config.yml, allowing default of the postgres port + 1000
<code>--read-only-port</code>	接続マネージャーが読み取り専用接続をリッスンするポート。	Left empty in config.yml, allowing default of the read-write port + 1
<code>--http-port</code>	接続マネージャーが http API 接続をリッスンするポート。	Left empty in config.yml, allowing default of the read-write port + 2
<code>--use https</code>	Connection Manager の http API の https を有効にする	https is not enabled

PGD-S 構成の詳細

PGD-S クラスターには 3 つのデータノードがあります。 *standard* レイアウトでは、ノードはすべて同じ場所にあります。 *near-far* レイアウトでは、2 つのノードがプライマリロケーションにあり、もう 1 つがセカンダリロケーションにあります。 [PGDdocumentation](#) を参照してください。

2 つのレイアウトの詳細については、

また、クラスターには、1 つのバーマンノードと最大 10 のサブスクライバー専用ノードが含まれ、`--add-subscriber-only-nodes` パラメーターによって制御されます。これらは常にプライマリの場所にあります。

オプションで `--bdr-database dbname` を指定して、BDR が有効になったデータベースの名前を設定できますデフォルト *bdrdb*。

クラスター構成 で説明されるオプションを指定することもできます。

3.6 PGD-X

!!!注

このアーキテクチャは、Postgres Distributed 6 専用です。PGD 5 が必要な場合は、*PGD-Always-ON* を使用してください。

このアーキテクチャには、EDB サブスクリプションが必要です。すべてのソフトウェアは *Configuring EDB Repos 2.0 repositories* から供給されています。

3.6.1 クラスター構成

構成オプションの概要

このアーキテクチャの `tpaexec configure` の呼び出し例を以下に示します。

```
tpaexec configure ~/clusters/pgd-x \
  --architecture PGD-X \
  --edb-postgres-extended 15 \
  --platform aws --instance-type t3.micro \
  --distribution Debian \
  --pgd-routing global \
  --location-names dc1 dc2 dc3 \
  --witness-only-location dc3 \
  --data-nodes-per-location 2
```

`help` コマンドを使用して、使用可能なすべてのオプションをリストできます。

```
tpaexec configure --architecture PGD-X --help
```

以下の表では、PGD-X の必須オプションと追加の重要なオプションについて説明しています。オプションの詳細については、次のセクションで説明します。

必須オプション

Options	Description
<code>--architecture (-a)</code>	<code>`PGD-X`</code> に設定する必要があります
Postgres flavour and version (e.g. <code>--postgresql 15</code>)	有効な <i>Postgres</i> のフレーバーとバージョン。
<code>--pgd-routing</code>	<code>`global`</code> または <code>`local`</code> のいずれかである必要があります。

追加オプション

Options	Description	Behaviour if omitted
<code>--platform</code>	<code>aws</code> 、 <code>docker</code> 、 <code>`bare`</code> のいずれか。	Defaults to <code>aws</code> .
<code>--location-names</code>	場所名のスペース区切りのリスト。場所の数は、指定された名前数と同じです。	TPA will configure a single location with three data nodes.
<code>--witness-only-location</code>	ロケーション名は、 <code>`location-names`</code> のメンバーである必要があります。	No witness-only location is added.
<code>--data-nodes-per-location</code>	各場所のデータノードの数は、少なくとも 2 である必要があります。	Defaults to 3.
<code>--enable-camo</code>	各ロケーションに 2 つのデータノードを CAMO パートナーとして設定します。	CAMO will not be enabled.
<code>--bdr-database</code>	レプリケーションに使用されるデータベースの名前。	Defaults to <code>bdrdb</code> .
<code>--enable-pgd-probes</code>	<code>`health/is-ready`</code> などの pgd-プロキシの http(s) API エンドポイントを有効にして、プロキシの状態をプローブできるようにします。	Disabled by default.
<code>--proxy-listen-port</code>	プロキシノードがトラフィックを書き込みリーダーにルーティングするポート。	Defaults to 6432
<code>--proxy-read-only-port</code>	プロキシノードが読み取り専用トラフィックをシャドウノードにルーティングするポート。	Defaults to 6433

PGD-X 構成の詳細

PGD-X クラスターは、多くの場所、好ましくは奇数で構成され、それぞれが同じ数のデータノード、やはり好ましくは奇数です。`--location-names` を指定しない場合、デフォルトでは、3 つのデータノードを持つ単一の場所を使用します。

クラスターの場所名は `--location-names dc1 dc2 …` として指定されます。ロケーションは、この定義がユースケースにとって意味のある方法で、レベルの冗長性を提供する独立したデータセンターを表します。たとえば、AWS リージョン、独自のデータセンター、またはサーバーがホストされている場所を識別するためのその他の指定。

`--data-nodes-per-location N` を使用して、各場所のデータノードの数を指定します。最小数は 2 で、デフォルトは 3 です。

ロケーションごとに偶数のデータノードを指定すると、TPA は各ロケーションに追加の監視ノードを自動的に追加します。これにより、コストを削減しながら、信頼できるコンセンサスを確立する機能が維持されます。監視のハードウェア要件は、データノードと比較して最小限です。

2 つのロケーションのみからなるクラスターは、いずれかのロケーションに障害が発生した場合、グローバルコンセンサスを確立する能力を完全に失います。3 番目の監視専用の場所を追加することをお勧めします。これにはデータノードは含まず、監視ノードのみが含まれ、これもコンセンサスを確実に確立する

ために使用されます。--witness-only-location loc を使用して、いずれかの場所を監視者として指定します。

ユースケースによっては、--pgd-routing local または global を指定して、Connection Manager が書き込みリーダーに接続をルーティングする方法を構成する必要があります。ローカルルーティングは、すべての接続マネージャーを独自の場所内の書き込みリーダーにルートさせますジオシャーディングアプリケーションに適しています。グローバルルーティングは、すべての接続マネージャーを、すべての場所で使用可能なすべてのデータノードの中から選択される単一の書き込みリーダーにルートします。

オプションで--bdr-database dbname を指定して、BDR が有効になったデータベースの名前を設定できますデフォルト bdrdb。

オプションで--enable-camo を指定して、各リージョンの2つのデータノードを CAMO パートナーとして設定できます。

オプションで--enable-pgd-probes [{http, https}] を指定して、プロキシの状態を簡単にプローブできる http API エンドポイントを有効にすることができます。

クラスター構成 で説明されるオプションを指定することもできます。

第4章 Platforms

4.1 aws

TPA は、AWS EC2 での運用クラスターのプロビジョニングを完全にサポートしています。

4.1.1 API アクセス設定

AWS API を使用するには、次のことを行う必要があります。

- Obtain an access keypair
- Add it to your configuration

例

```
[tpa]$ cat > ~/.aws/credentials
[default]
aws_access_key_id = AKIAIOSFODNN7EXAMPLE
aws_secret_access_key = wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

tpaexec がそれを使用して ec2 リソースをプロビジョニングできるように、IAM ユーザーは少なくとも次の権限セットを持っている必要があります。

```
ec2:AssociateRouteTable
ec2:AttachInternetGateway
ec2:AuthorizeSecurityGroupIngress
ec2:CreateInternetGateway
ec2:CreateRoute
ec2:CreateRouteTable
ec2:CreateSecurityGroup
ec2:CreateSubnet
ec2:CreateTags
ec2:CreateVpc
ec2>DeleteKeyPair
ec2>DeleteRouteTable
ec2>DeleteSecurityGroup
ec2>DeleteSubnet
ec2>DeleteVpc
ec2:DescribeImages
ec2:DescribeInstanceStatus
ec2:DescribeInstances
```

(次のページに続く)

(前のページからの続き)

```
ec2:DescribeInternetGateways
ec2:DescribeKeyPairs
ec2:DescribeRouteTables
ec2:DescribeSecurityGroups
ec2:DescribeSubnets
ec2:DescribeTags
ec2:DescribeVolumes
ec2:DescribeVpcAttribute
ec2:DescribeVpcClassicLink
ec2:DescribeVpcClassicLinkDnsSupport
ec2:DescribeVpcs
ec2:DisassociateRouteTable
ec2:ImportKeyPair
ec2:ModifyVpcAttribute
ec2:RevokeSecurityGroupIngress
ec2:RunInstances
ec2:TerminateInstances
iam:AddRoleToInstanceProfile
iam:CreateInstanceProfile
iam:CreateRole
iam>DeleteInstanceProfile
iam>DeleteRole
iam>DeleteRolePolicy
iam:GetInstanceProfile
iam:GetRole
iam:GetRolePolicy
iam:ListAttachedRolePolicies
iam:ListGroups
iam:ListInstanceProfiles
iam:ListInstanceProfilesForRole
iam:ListRolePolicies
iam:ListRoles
iam:ListUsers
iam:PassRole
iam:PutRolePolicy
iam:RemoveRoleFromInstanceProfile
kms:CreateGrant
kms:GenerateDataKeyWithoutPlaintext
s3:CreateBucket
s3:GetBucketVersioning
s3:GetObject
s3:GetObjectTagging
s3:ListAllMyBuckets
s3:ListBucket
s3:ListBucketVersions
```

(次のページに続く)

(前のページからの続き)

```
s3:PutBucketOwnershipControls
s3:PutObject
s3:PutObjectAcl
```

4.1.2 はじめに

サービスは物理的に **regions and availability zones** に分割されます。アベイラビリティゾーンは、リージョンコードに続いて 1 文字の **eu-west-1a** など表されますが、その名前は、異なる AWS アカウントの異なる場所を参照する場合があります、アカウント間の解釈を調整する方法はありません。

AWS リージョンは互いに完全に分離され、リソースを共有しません。リージョン内のアベイラビリティゾーンは物理的に分離され、論理的にほとんど分離されていますが、低遅延リンクで接続され、特定のネットワークリソースを共有できます。

ネットワーキング

AWS のすべてのネットワーク構成は、**Virtual Private Cloud** のコンテキストで発生します

リージョン内で。VPC 内で、**subnets** を作成できます

インターネットゲートウェイ、ルーティングテーブルなどとともに、特定のアベイラビリティゾーンに関連付けられています。

Security Groups は、いくつでも作成できます

インスタンスに許可されるインバウンドおよびアウトバウンドトラフィックのルール、プロトコル、宛先ポートレンジ、およびソースまたは宛先 IP アドレスレンジの観点から。

インスタンス

AWS EC2 は、さまざまな **instance types** を提供します

異なる価格/パフォーマンスポイントで異なるハードウェア構成を使用します。特定のアベイラビリティゾーンのサブネット内に、**EC2 instances** を作成できます。

AMI と呼ばれるディストリビューションイメージに基づいて、1 つ以上の **EBS volumes** をアタッチします

インスタンスに永続ストレージを提供します。**SSH public key** を登録することにより、インスタンスに SSH 接続できます。

インスタンスには、サブネット内のプライベート IP アドレスが常に割り当てられます。サブネット構成によっては、**ephemeral public IP address** が割り当てられる場合があります。

これは、インスタンスがシャットダウンすると失われ、再起動されたときに別のエフェメラル IP が割り当てられます。代わりに、**Elastic IP** と呼ばれる静的リージョン固有のルーティング可能 IP アドレスを割り当てることができます。

任意のインスタンスに。

外部からインスタンスに到達できるようにするには、ルーティング可能な IP アドレスを持つだけでなく、VPC のネットワーク構成インターネットゲートウェイ、ルーティングテーブル、セキュリティグループもアクセスを許可するように調整する必要があります。

4.1.3 構成

ここでは、`tpaexec configure` を介して指定できる、または `config.yml` で直接定義できる AWS 固有の設定の簡単な説明を示します。

リージョン

`--region` または `--regions` で使用するクラスターの 1 つ以上のリージョンを指定できます。TPA は、それぞれに関連する必要な `vpc` エントリを生成し、可能な場合は異なるアベイラビリティゾーンを使用してロケーションをこれらのリージョンに均等に分散します。

`regions` は `locations` とは異なり、各ロケーションはリージョンおよびこのリージョン内のアベイラビリティゾーンに属します。`regions` は AWS 固有のオブジェクト、`locations` はクラスターオブジェクトです。

注複数のリージョンを指定する場合、ネットワーク構成を手動で編集する必要があります。

- `ec2_vpc` エントリに AWS `vpc` ピアリングを使用できるように、オーバーラップしない `cidr` ネットワークが必要です。デフォルトでは、TPA はすべての `cidr` を `10.33.0.0/16` に設定します。詳細については、[VPC 必須](#) を参照してください。
- 各 `location` は、それらが属している `ec2_vpc cidr` と一致する `subnet` で更新する必要があります。詳細については、[サブネットオプション](#) を参照してください。
- TPA は `cluster_rules` の下で基本的なルールを使用してセキュリティグループを作成します。これらは、各 `subnet cidr` の `ec2_vpc cidr` と一致するように更新する必要があります。詳細については、[セキュリティグループオプション](#) を参照してください。
- `tpaexec deploy` の前に VPC ピアリングを手動で設定する必要があります。`tpaexec configure` を実行し、`config.yml` で `vpc-id` を使用する前に、VPC と必要な VPC ピアリングを作成することをお勧めします。詳細については、[VPC 必須](#) を参照してください。

VPC 必須

使用する VPC を指定する必要があります。

```
ec2_vpc:
  Name: Test
  cidr: 10.33.0.0/16
```

これはデフォルト構成であり、指定された CIDR で Test という名前の VPC が存在しない場合は作成し、それ以外の場合は既存の VPC を使用します。

VPC を作成するには、名前と `cidr` の両方を指定する必要があります。VPC 名のみを指定した場合、一致する VPC が存在しない場合、TPA は失敗します。

TPA が VPC を作成すると、`tpaexec deprovision` はそれを削除しようとしませんが、既存の VPC はそのままにしておきます。AWS にはアカウントごとの VPC 数に 1 桁のデフォルト制限があるため、新しい VPC を作成する前によく考えてください。

より詳細なマッチングが必要な場合、またはさまざまなリージョンでさまざまな VPC を指定する場合、展開されたフォームを使用できます。

```
ec2_vpc:
  eu-west-1:
    Name: Test
    cidr: 172.16.0.0/16
  us-east-1:
    filters:
      vpc-id: vpc-nnn
  us-east-2:
    Name: Example
    filters:
      [filter expressions]
```

AMI 必須

使用する AMI を指定する必要があります。

```
ec2_ami:
  Name: xxx
  Owner: self
```

より正確なマッチングのためにフィルタ仕様を追加できます。

```
ec2_ami:
  Name: xxx
  Owner: self
  filters:
    architecture: x86_64
    [more key/value filters]
```

デフォルトでは、`tpaexec configure` は `--distribution` 引数に基づいて、適切な `ec2_ami` を選択します。

サブネットオプション

すべてのインスタンスは、そのサブネットを CIDR 形式で、または `subnet-xxx` ID として指定する必要があります。オプションで、作成する各サブネットの名前とアベイラビリティゾーンを指定できます。

```
ec2_vpc_subnets:
  us-east-1:
    192.0.2.0/27:
      az: us-east-1b
```

(次のページに続く)

(前のページからの続き)

```
Name: example1
192.0.2.100/27:
  az: us-east-1b
Name: example2
```

セキュリティグループオプション

デフォルトでは、クラスターのセキュリティグループを作成します。1つ以上の既存のセキュリティグループを使用するには、次のように設定します。

```
ec2_groups:
  us-east-1:
    group-name:
      - foo
```

デフォルトのセキュリティグループのルールをカスタマイズする場合は、`cluster_rules` を設定します。

```
cluster_rules:

- cidr_ip: 0.0.0.0/0
  from_port: 22
  proto: tcp
  to_port: 22

- cidr_ip: 192.0.2.0/27
  from_port: 0
  proto: tcp
  to_port: 65535

- cidr_ip: 192.0.2.100/27
  from_port: 0
  proto: tcp
  to_port: 65535
```

この例では、任意のアドレスからの ssh ポート 22、および特定の IP レンジからの任意のポートでの TCP 接続を許可します。注 `from_port` および `to_port` は、ソースと宛先ではなく、ポートの数値範囲を定義します。

カスタムルールを設定するか、既存のセキュリティグループを使用する場合、クラスター内のインスタスが必要に応じて相互に通信を許可されていることを確認する必要がありますたとえば、Postgres の TCP/5432 を許可します。

インターネットゲートウェイオプション

デフォルトでは、次を設定しない限り、すべての VPC のインターネットゲートウェイを作成します。

```
ec2_instance_reachability: private
```

より詳細な制御のために、次を設定できます。

```
ec2_vpc_igw:
  eu-west-1: yes
  eu-central-1: yes
  us-east-1: no
```

SSH キーオプション

```
# Set this to change the name under which we register our SSH key.

# ec2_key_name: tpa_cluster_name

#
# Set this to use an already-registered key.

# ec2_instance_key: xxx
```

S3 バケットオプション

TPA では、AWS クラスターをプロビジョニングするために S3 バケットへのアクセスが必要です。このバケットは、SSH ホストキーなどのファイルを一時的に保存するために使用されますが、他のクラスターデータバックアップなどに使用される場合があります。

デフォルトでは、TPA はプロビジョニングするクラスターに `edb-tpa-<aws-account-user-id>` という名前の S3 バケットを使用します。バケットが存在しない場合、TPA にバケットを作成するかどうかを確認するメッセージが表示されます。

代わりに既存の S3 バケットを使用するには、次のように設定します

```
cluster_bucket: name-of-bucket
```

確認プロンプトを表示せずにデフォルトのバケット名を受け入れるように `cluster_bucket: auto` を設定することもできます。

クラスターのプロビジョニングを解除するときに、TPA は S3 バケットを削除しません。バケットを自分で削除するには、次を実行します。

```
aws s3 rb s3://<bucket> --force
```

インスタンスのプロビジョニングに使用している IAM ユーザーには、このバケットへの読み取りおよび書き込みアクセスが必要です。プロビジョニング中に、`tpaexec` は、インスタンスプロファイルを介して `cluster_bucket` への読み取り専用アクセスをインスタンスに提供します。

Elastic IP アドレス

Elastic IP アドレスを使用するには、`config.yml` で `assign_elastic_ip` を `true` に設定します。クラスター内のすべてのインスタンスに影響を与えるように `instance_defaults` で、または必要に応じて別のインスタンスに個別に。デフォルトでは、これにより、新しい Elastic IP アドレスが割り当てられ、新しいインスタンスに割り当てられます。割り当て済みだがまだ割り当てられていない Elastic IP アドレスを使用するには、`elastic_ip: 34.252.55.252` を使用して、割り当てられたアドレスに置き換えます。

インスタンスプロファイルオプション

```
# Set this to change the name of the instance profile role we create.

# cluster_profile: cluster_name_profile

#

# Set this to use an existing instance profile (which must have all the

# required permissions assigned to it).

# instance_profile_name: xxx
```

4.2 bare(-metal servers)

`config.yml` に `platform: bare` を設定します

このプラットフォームは、ベアメタルサーバーやクラウドプラットフォーム AWS を含む既にプロビジョニングされたサーバーを含む、SSH を介してアクセス可能なサーバーをサポートすることを目的としています。

各ターゲットサーバーの IP アドレスとユーザー名を定義する必要があります。

```
instances:
- node: 1
  Name: igor
  platform: bare
  public_ip: 192.0.2.1
  private_ip: 192.0.2.222
  vars:
    ansible_user: xyzzy
```

次のことを確認する必要があります

1. TPA は `ansible_user` としてインスタンスに `ssh` できます
2. `ansible_user` には、インスタンスで `sudo` アクセスがあります

4.2.1 SSH アクセス

上記の例では、TPA は `xyzz@192.0.2.1` に ssh 接続してインスタンスにアクセスします。

デフォルトでは、TPA は `ssh-keygen` を実行して、クラスターディレクトリに新しい SSH キーペアを生成します。プライベートキーの名前は `id_cluster_name` であり、公開キーは `id_cluster_name.pub` に保存されます。

インスタンスが受け入れる別のキーを使用するように `ssh_key_file: /path/to/id_keyname` を設定するか、生成されたキーからのアクセスを許可するようにインスタンスを構成する必要がありますたとえば、`id_cluster_name.pub` のコンテンツを `~xyzz/.ssh/authorized_keys` に追加する `ssh-copy-id` を使用します。

また、ssh がインスタンスのホストキーを検証できることを確認する必要があります。クラスターディレクトリの `known_hosts` ファイルにエントリを追加するか、TPA 生成ホストキーをクラスターディレクトリの `hostkeys/ssh_host_ *_key*` からインスタンスの `/etc/ssh` にインストールできます。生成された `tpa_known_hosts` ファイルには、これらのキーのエントリが含まれています。

たとえば、生成されたユーザーキーを使用して ssh 接続しますが、既存のホストキーを保持するには、次のことができます。

```
$ cd ~/clusters/speedy
$ ssh-copy-id -i id_speedy xyzz@192.0.2.1
$ ssh-keyscan -H 192.0.2.1 >> tpa_known_hosts
```

`tpaexec ping ~/clusters/speedy` を実行して、動作しているかどうかを確認します。まだない場合は、コマンドに `-vvv` を追加して、完全な ssh コマンドラインを確認します。（注 Ansible は ssh を呼び出して、インスタンスで `bash -c 'python3 && sleep 0'` のようなコマンドを実行します。デバッグ中に ssh コマンドを手動で実行する場合、これを 'id' などの出力を生成して終了するコマンドに置き換えます。）

詳細については

- [ssh_key_file](#)
- [Managing SSH host keys](#)

4.2.2 ディストリビューションのサポート

TPA は、ターゲットインスタンスで実行されているディストリビューションを検出しようとしませんが、サポートされていない場合は失敗します。TPA は現在 Debian 10/11/12; または buster/bulseye/bookworm、Ubuntu 16.04/18.04/20.04/22.04; または xenial/bionic/focal/jammy)、および RHEL/CentOS/Rocky/AlmaLinux 7. x/8.x)bare インスタンスで。

4.2.3 IP アドレス

任意のインスタンスに `public_ip`、`private_ip`、またはその両方を指定できます。

TPA は、これらの IP アドレスを 2 つの方法で使用します。1 つ目は、インスタンスに ssh して、展開中にコマンドを実行します。次に、クラスター内の通信を設定するためなど、`/etc/hosts` または Postgres の `primary_conninfo` を設定します。

`public_ip` を指定すると、デプロイ中にインスタンスに `ssh` 接続するために使用されます。`private_ip` を指定すると、クラスター内の通信を設定するために使用されます。両方を指定した場合、`public_ip` は展開中に使用され、`private_ip` はクラスター通信に使用されます。

いずれかのみを指定した場合、アドレスは両方の目的に使用されます。たとえば、別のネットワーク上のサーバーに `public_ip` のみを設定したり、閉じたネットワーク内で TPA を実行している場合は `private_ip` のみを設定したりできます。(パブリック/プライベートを使用する代わりに、1 つの IP アドレスのみを指定する必要がある場合は、`ip_address` を設定できます。)

4.2.4 新しく始める

`bare` プラットフォームのクラスターで新しくスタートするには、適切な外部ツールを使用してサーバーを再インストール、再イメージ、または再プロビジョニングし、このドキュメントで説明しているプロセスを繰り返します。新しいサーバーの IP アドレスが異なる場合、または複雑な `ssh` セットアップがある場合は、`tpaexec deprovision` を実行してローカルに作成されたすべてのファイルを削除してから `tpaexec provision` それらを再作成し、上記と同様にこのドキュメントのプロセスを繰り返します。

4.3 Docker

TPA は Docker コンテナを作成し、それらにクラスターを展開できます。現時点では、通常の VM であるかのように `systemd` およびその他のサービスを実行するようにコンテナをセットアップします。

Docker コンテナへの展開は、さまざまなクラスター構成をテストする簡単な方法です。実稼働使用を目的としたものではありません。

4.3.1 概要

構成時にプラットフォームを選択するだけです。

```
[tpa]$ tpaexec configure clustername --platform docker [...]
[tpa]$ tpaexec provision clustername
[tpa]$ tpaexec deploy clustername
```

4.3.2 オペレーティングシステムの選択

標準の `--os Debian/Ubuntu/RedHat/SLES` `configure` オプションを使用して、コンテナに使用するディストリビューションを選択します。TPA は、このディストリビューション用に独自の `systemd` 対応イメージをビルドします。これらのイメージには、`tpa/redhat:8` など、`tpa/` 接頭辞が付いた名前が付けられます。

代わりに `--os-image some/image:name` を使用して、既存の `systemd` 対応イメージを指定します。例、`centos/systemd`

イメージ CentOS 7 ベースは、この方法で使用できます。

TPA は、Docker コンテナで Debian 8 `jessie` または Ubuntu 16.04 `xenial` をサポートしていません。これらのディストリビューションに同梱されている `systemd` の古いバージョンのバグのため。

4.3.3 Docker のインストール

最新の安定した Docker-CE パッケージを使用して TPA をテストします。

このドキュメントでは、動作する Docker インストールがあり、イメージのプルやコンテナの作成などの基本的な操作に精通していることを前提としています。

[install Docker](#) および [get started](#) のヘルプが必要な場合は、[Docker documentation](#) を参照してください。

MacOS X では、次のことができます [install "Docker Desktop forMac"](#)

そして、アプリケーションメニューから Docker を起動します。

Cgroup

TPA は、cgroups バージョン 1 または 2 を実行しているホストで Docker コンテナをサポートします。cgroups2 を実行しているホストでは、RHEL 7 を実行しているインスタンスはサポートされていません。

RHEL 7 インスタンスを使用する必要があるが、ホストが cgroups バージョン 2 を実行している場合、次のように cgroups バージョン 1 に切り替えることができます。

Debian ファミリーの Linux ディストリビューションの場合

```
$ echo GRUB_CMDLINE_LINUX=systemd.unified_cgroup_hierarchy=false > \
/etc/default/grub.d/cgroup.cfg
$ update-grub
$ reboot
```

RedHat ファミリーの Linux ディストリビューションの場合

```
$ grubby --args=systemd.unified_cgroup_hierarchy=false --update-kernel=ALL
$ reboot
```

MacOS の場合

1. ~/ライブラリー/Group\ Containers/group.com.docker/settings.json を編集し、次の置き換えを行います
"deprecatedCgroupv1": false → "deprecatedCgroupv1": true
2. Docker デスクトップアプリを再起動します

アクセス許可

TPA は、それを実行しているユーザーが Docker デーモンへのアクセス権限を持っていることを想定しています。通常は、/var/run/docker.sock を所有する docker グループのメンバーであること。次のようなコマンドを実行して、アクセスがあるかどうかを確認します。

```
[tpa]$ docker version --format {{.Server.Version}}
19.03.12
```

!!!警告

ユーザーに Docker デーモンと対話できる機能を提供すると、Docker ホストでの root を簡単に取得できます。信頼できるユーザーのみが Docker デーモンにアクセスできる必要があります。

特権コンテナ

デフォルトでは、TPA は Linux 機能フラグを追加せずに、非特権モードで Docker コンテナをプロビジョニングします。このようなコンテナは、ホストファイアウォールルール、ファイルシステム、ブロックデバイス、またはホストで真の root 権限を必要とするその他のほとんどのタスクを管理できません。

コンテナを特権モードで実行する必要がある場合は、それを必要とするインスタンスまたは `instance_defaults` でグローバルに `privileged` ブール変数を設定します。例

```
instance_defaults:
  privileged: true
```

!!!警告

特権モードでコンテナを実行すると、root ユーザーまたは root を取得できるプロセスは、カーネルモジュールのロード、ホストファイアウォールルールを変更、コンテナ名前空間をエスケープ、またはその他の方法で実際のホストの「root」ユーザーと同じように動作できます。本当に必要な場合を除き、特権モードでコンテナを実行しないでください。

security_opts および no-new-privileges フラグ

`tpaexec` は、プロセスが特権を増加できない制限モードで Docker コンテナを起動できます。setuid バイナリは制限されているなど。 `instance_defaults` またはコンテナごとの変数 `docker_security_opts` を使用して `tpaexec` でこれを有効にします。

```
instance_defaults:
  docker_security_opts:
    - no-new-privileges
```

`docker run` の `--security-opts` への他の引数も受け入れられます。SELinux のユーザーとロール。

Linux 機能フラグ

`tpaexec` は、`docker_cap_add` リスト変数を使用して Linux 機能フラグを Docker の制御を公開します。このリスト変数は、コンテナごとまたは `instance_defaults` で設定できます。機能フラグの詳細については、`man capabilities`、`docker run` ドキュメント、および Ansible `docker_containers` モジュールのドキュメントを参照してください。

Docker の `--cap-drop` は、`docker_cap_drop` リストを介してサポートされています。

たとえば、コンテナを非特権として実行しながら、システムクロックを変更する機能を付与するには、次のように書くことができます。

```
instance_defaults:
  privileged: false
  docker_cap_add:
    - sys_time
  docker_cap_drop:
    - all
```

Docker ストレージ構成

多くのホスト上のデフォルトの Docker 構成は `lvm-loop` ブロックストレージを使用し、運用展開には適していません。 `docker info` を実行して、使用しているストレージドライバーを確認します。ループバックスキームを使用している場合、次のようなものが表示されます。

```
Storage Driver: devicemapper
...
Data file: /dev/loop0
```

ストレージ構成の詳細については、Docker ドキュメントを参照してください。

- [Storage Drivers](#)
- [Configuring lvm-direct for production](#)

Docker MTU 設定

デフォルトでは、Docker ネットワークの最大転送ユニット MTU は 1500 バイトです。これがホストシステムのネットワークインターフェイスの MTU より大きい場合、そのインターフェイスを介して Docker コンテナに接続をルーティングする際に問題が発生する可能性があります。 `ipconfig | grep mtu`、`ip | grep mtu` などのコマンドを使用して、ネットワークインターフェイスの MTU を確認できます。以下に示すように、`config.yml` のネットワークに適切なドライバーオプションを追加することにより、TPA によってプロビジョニングされた Docker ネットワークの MTU を変更できます。

```
docker_networks:
- ipam_config:
  - subnet: 10.33.214.192/28
  name: tpa-docker
  driver_options:
    com.docker.network.driver.mtu: 1400
```

!!!警告

MTU は、Docker ネットワークが最初にプロビジョニングされるときにのみ設定できます。 `config.yml` の後続の変更は効果ありません。

4.3.4 Docker コンテナ管理

クラスター内のすべての Docker コンテナは、 `start-containers` および `stop-containers` コマンドを使用して、一緒に起動および停止できます。

```
[tpa]$ tpaexec start-containers clustername
[tpa]$ tpaexec stop-containers clustername
```

これらのコマンドは、コンテナをプロビジョニングまたはプロビジョニング解除し、さらにはコンテナに接続しません。これらは、将来の使用に備えて維持する必要がある Docker クラスターを一時的に使用しない場合に、リソースを節約することを目的としています。

クラスター内にプロビジョニングされた Docker コンテナの概要を表示するには、起動または停止にかかわらず、`list-containers` コマンドを使用します。

```
[tpa]$ tpaexec list-containers clustername
```

第5章 Customizations

5.1 Cluster configuration

TPA では、クラスターの構成を変更する方法は、`config.yml` を編集し、プロビジョニング/展開/テストサイクルを実行することです。このプロセスは、べき等であるように、構成の変更またはインスタンスの変更に応じてのみ変更を行うように注意深く設計されています。

クラスター構成 コマンドは、賢明な `config.yml` ファイルを生成しますが、最も一般的なトポロジと構成オプションのみをカバーしています。デフォルト以外のものが必要な場合、またはクラスターをプロビジョニングした後に変更を行う必要がある場合は、とにかく `config.yml` を編集する必要があります。

このページは、使用可能な構成メカニズムの概要を示しています。特定の *Instance configuration* の詳細を含む別のページがあります。

5.1.1 config.yml

`config.yml` ファイルは、目的のクラスター構成のすべての側面を表す **YAML format** テキストファイルです。次に、2つのインスタンスを持つクラスターの最小限の例を示します。

```
cluster_name: speedy

cluster_vars:
  postgres_version: 14

instances:

- node: 1
  Name: one
  role: primary
  platform: docker
  vars:
    ansible_user: root
    x: 42

- node: 2
  Name: two
  role: replica
  platform: docker
  upstream: one
```

(次のページに続く)

```
vars:
  ansible_user: root
  x: 53
```

これら 3 つの定義は、クラスター構成の中心となります。ファイルには、他の多くの定義プラットフォーム固有の詳細を含む場合がありますが、1 つのインスタンスまたはクラスター全体に設定された `instances` と `vars` のリストは、すべての TPA 構成の基本ビルディングブロックです。

すべて クラスター構成

オプションは、何らかの方法で `config.yml` 変数に変換されます。1 つのオプションは複数の変数に影響を与える場合がありますたとえば、`--bdr-version` は `postgres_version`、`edb_repositories`、`extra_postgres_extensions` などを設定できますが、コマンドを実行してできることはエディターでいつでも実行できます。

YAML 構文の観点では、`config.yml` 全体が `cluster_vars` や `instances` などのキーを含むハッシュを表します。各キーが 1 度だけ定義されていることを確認する必要があります。誤って `cluster_vars` を繰り返した場合、2 番目の定義が前者を完全にオーバーライドし、次の展開はシャドウされた変数が欠落しているために意図しない変更を行う可能性があります。

TPA は、クラスタートポロジ全体の一貫性をチェックしますたとえば、ロール「レプリカ」でインスタンスを宣言する場合、その上流インスタンスの名前も宣言し、そのインスタンスが存在する必要があります。ただし、それは妨げるものではありませんインスタンスに好きな変数を設定します。十分な注意を払い、変更を運用環境にロールアウトする前にテスト環境で変更を試す必要があります。

5.1.2 変数

Ansible の用語では、ほとんどの構成設定は「インベントリー変数」です。TPA は、プロビジョニング中にインベントリーで `cluster_vars` を `group_vars` クラスター全体に適用し、各インスタンスの `vars` を `host_vars` に変換し、デプロイメントはインベントリー値を使用します。`config.yml` を変更した後、`tpaexec deploy` の前に `tpaexec provision` を実行することを忘れないでください。

クラスター全体、個々のホスト、またはその両方に変数を設定できます。ホスト変数はグループ変数をオーバーライドします。実際には、`cluster_vars` で `x: 42` を設定することは、すべてのホストの `vars` で設定することと同じです。デプロイメント中に `x` を必要とするホストには、いずれの場合も値 42 が表示されます。ホストは常に最も具体的な値を参照するため、グループにいくつかのデフォルト値を設定し、必要に応じて特定のインスタンスにオーバーライドすると便利です。

可能な限り、`cluster_vars` で変数を定義し、特定のインスタンスでそれらをオーバーライドすると、確認と変更が簡単な簡潔な構成が得られます。繰り返しが少なくなります。それを超えて、特定の設定がグループ変数またはホスト変数としてより意味があるかどうかは、あなたが決定する必要があります。

5.1.3 クラスター変数

`cluster_vars` の下のキーは、有効な YAML タイプにマップでき、Ansible インベントリのグループ変数に直接変換されます。

```
cluster_vars:
  postgres_version: 14
  edb_repositories:
    - enterprise
    - postgres_distributed
  postgres_conf_settings:
    bdr.trace_replay: true
```

!!!警告

テンプレートで使われる変数は、`config.yml` のトップレベル `cluster_name` 変数と同じレベルで定義する必要があります

以下の例を参照してください。

この場合、`tpaexec provision` は 3 つの変数文字列、リスト、およびハッシュを `group_vars/tag_Cluster_name/01-cluster_name.yml` のインベントリに書き込みます。

5.1.4 インスタンス変数

このドキュメントでは、`config.yml` の特定のインスタンスに対して定義された変数を参照するために「インスタンス変数」という用語を使用しています。例、これは一般的なインスタンス定義です。

```
instances:
- Name: unwind
  node: 1
  backup: unkempt
  location: a
  role:
    - primary
    - bdr
  volumes:
    - device_name: root
      encrypted: true
      volume_size: 16
      volume_type: gp2
    - device_name: /dev/xvdf
      encrypted: true
      vars:
        volume_for: postgres_data
        volume_size: 64
        volume_type: gp2
  platform: aws
  type: t3.micro
  vars:
```

(次のページに続く)

```

ansible_user: ec2-user
postgres_conf_directory: /opt/postgres/conf

```

このインスタンスの **vars** で定義された変数はすべてインベントリーのホスト変数になりますが、インベントリー内のすべてのホスト変数は **vars** のみから取得されるわけではありません。 **platform**、**location**、**volumes**、**role** などの他の一部のインスタンス設定もホスト変数としてインベントリーにコピーされますが、代わりに **vars** または **cluster_vars** でこれらの設定を定義することはできません。

vars の外部の設定は、インスタンスのプロパティたとえば **Name** および **node** またはクラスターのトポロジ内のその場所たとえば **role**、**backup** またはそれらはプラットフォーム固有の属性たとえばインスタンス **type** および **volumes** などを記述する場合があります。。 **vars** では定義できないことを除き、これらのインスタンスの「設定」とインスタンス「変数」を区別する必要があることはほとんどありません。

この場合、**tpaexec provision** は多くのホスト変数を **host_vars/unwind/01-instance_vars.yml** のインベントリーに書き込みます。

5.1.5 instance_defaults

これは、**config.yml** の繰り返しをさらに削減するメカニズムです。 **cluster_vars** として定義できないインスタンス設定に最も役立ちます。例、次のように書くことができます。

```

instance_defaults:
  platform: aws
  type: t3.micro
  tags:
    AWS_ENVIRONMENT_SPECIFIC_TAG_KEY: some_mandated_value

instances:

- node: 1
  Name: one

- node: 2
  Name: two

- ...

```

instance_defaults の下で指定したものは、**instances** のすべてのエントリーのデフォルトとして機能します。この例では、各インスタンスの **platform** および **type** のスペルアウトを保存し、すべてのインスタンスを別のタイプに変更するのが簡単になります。インスタンスが別の値を指定している場合、もちろんデフォルトより優先されます。

instance_defaults は、**instances** を定義する際に使用するマクロ機能であると考えると役立つでしょう。最終的にインベントリーに書き込まれるものは、**instances** の展開された定義のみから来ます。 **cluster_vars** または **instance_defaults** に何かを入れるかどうかを決定しようとしている場合、それはおそらく前者に属します変数たとえば、**platform** または **type** などとして定義できません。これは、多くのプラット

フォーム固有のプロパティに当てはまります AWS リソースタグなど) は、プロビジョニングでのみ使用され、展開中は使用されません。

`instance_defaults` メカニズムは、インスタンスの `vars` を埋めるために使用することを妨げるものではありません。デフォルトのハッシュ値は、`instances` エントリーで指定されたハッシュとマージされます。ただし、`cluster_vars` で同じデフォルトを設定し、必要に応じてインスタンスにオーバーライドするよりもこれを行うことに特に利点ははありません。よくわからない場合は、`cluster_vars` を使用してください。

5.1.6 ロケーション

`config.yml` で `locations` のリストを指定することもできます。

```
locations:

- Name: first
  az: eu-west-1a
  region: eu-west-1
  subnet: 10.33.110.128/28

- Name: second
  az: us-east-1b
  region: us-east-1
  subnet: 10.33.75.0/24

instances:

- node: 1
  Name: one
  location: first
...
```

インスタンスが `location: first` または `location: 0` を指定する場合、その場所の設定はそのインスタンスのデフォルトとして機能します。ここでも、`instance_defaults` と同様に、インスタンスはその場所から継承するデフォルトをオーバーライドできます。また、この機能を使用して、インスタンスの `vars` を入力できます。これは、インスタンスの半分にのみ適用されるデフォルトと、残りの半分に異なる値がある場合に役立ちます。上記の例のプラットフォーム固有の設定の場合。

ロケーションは、インスタンスが「オプトイン」できる設定のコレクションを表します。それらを使用して、さまざまなデータセンター、AWS リージョン、Docker ホスト、または完全に他のものを表すことができます。TPA は、特定の解釈を想定または強制しません。

5.2 Instance configuration

このページでは、クラスターインスタンスでの展開プロセスをカスタマイズするために TPA が提供するさまざまな制御の概要を、より詳細なドキュメントへのリンクとともに紹介します。

デプロイメントの詳細に入る前に、[クラスター構成](#) を読んで、`config.yml` のクラスター変数、インスタンス変数、およびその他のメカニズムがどのように連携して、簡潔でレビューが簡単な構成を書くことができるようにする方法を理解すると役立ちます。

5.2.1 システムレベルの構成

TPA が最初に行うことは、Python がブートストラップされ、Ansible モジュールディストリビューション固有のプロセスを実行する準備ができていることを確認することです。次に、以下の *Postgres* の [フレーバーとバージョン](#) に進む前に、さまざまなシステムレベルの構成タスクを完了します。

- *Distribution support*
- *TPA Python 環境のセットアップ* `preferred_python_version`
- *Environment variables* 例 `https_proxy`

パッケージリポジトリ

[展開前](#) を使用できます

パッケージリポジトリが構成される前にタスクを実行します。

- *Configuring YUM repositories*

RHEL、Rocky、および AlmaLinux の場合

- *Configuring APT repositories*

Debian および Ubuntu の場合

- *Configuring EDB Repos 2.0 repositories*

任意のシステムで

- *Creating and using a local repository*

パッケージをターゲットインスタンスにシップするため

[ポストリポジトリ](#) を使用できます

パッケージリポジトリが構成された後にタスクを実行するにはたとえば、パッケージをインストールする前にリポジトリ構成の問題を修正します。

パッケージのインストール

リポジトリが構成されると、パッケージは展開のさまざまな段階でインストールされます。システムパッケージのバッチから始めます。

- *Installing packages*

例、`acl`、`openssl`、`sysstat`

Postgres およびその他のコンポーネント Barman、`repmgr`、`pgbouncer` などは、クラスター構成に応じて個別にインストールされます。これらは、以下の独自のセクションに文書化されています。

その他のシステムレベルのタスク

- *Filesystem configuration* RAID、LUKS セットアップを含む
- *Uploading artifacts* ファイル、ディレクトリ、tar アーカイブ
- *Setting systemctl values*
- *Configuring /etc/hosts*
- *Managing SSH host keys*
- *Locale*

デプロイメントを完全にスキップする

TPA がインスタンスの展開プロセスの一部を実行しないようにするには、つまり、TPA にインスタンスをプロビジョニングして、そのままにしておく場合、インスタンスの `provision_only` 設定を `config.yml` で `true` に設定します。この設定により、TPA は、`tpaexec deploy` が認識するインベントリからインスタンスを完全に省略します。

5.2.2 Postgres

Postgres 構成は、`repmgr` や `pgbouncer` などの他のコンポーネントのセットアップと連携する拡張プロセスです。それは Postgres 自分自身をインストールすることから始まります。

バージョンの選択

`ソフトウェアバージョン` を使用して Postgres フレーバーとバージョンを選択するか、`config.yml` で `postgres_version` を設定して、インストールする Postgres メジャーバージョンを指定します。

作業用クラスターをセットアップするために実際に必要な作業はこれだけです。このページのその他の部分はすべてオプション。必要に応じて、展開のすべての側面を制御できますが、デフォルトは注意深く調整されて、開始点として合理的なクラスターを提供します。

インストール

デフォルトの `postgres_installation_method` では、アーキテクチャのニーズに応じて、選択した Postgres のバージョンのパッケージをさまざまな拡張機能とともにインストールします。

- *Installing Postgres-related packages*

例、`pglogical`、`BDR` など

- *Postgres source installation*

開発およびテスト用

どのインストール方法を選択するかにかかわらず、TPA は最小限の労力で同じクラスター構成を提供できます。

構成

- *The postgres Unix user*
- *Running initdb*
- *pg_hba.conf*
- *pg_ident.conf*
- *postgresql.conf*

postgres-config を使用できます

Postgres 構成ファイルがインストールされた後にタスクを実行するには追加の構成ファイルをインストールします。

Postgres の構成が配置されると、TPA は、アーキテクチャの詳細に従って、Barman、repmgr、pgbouncer、haproxy などの他のコンポーネントのインストールと構成を開始します。

5.2.3 その他のコンポーネント

- *barman-pre-config*
- *Repmgr* リダイレクト *pgbouncer*
- *Configuring haproxy*
- *Configuring HARP*
- *Configuring EFM*

サービスの構成と開始

TPA は、各サービスの systemd サービスユニットファイルをインストールします。Postgres のサービスの名前は `postgres.service` で、`systemctl start postgres` で開始または停止できます。

最初の展開では、Postgres サービスが開始されます。 `tpaexec deploy` を再度実行している場合、行った構成変更によっては、サービスがリロードまたは再起動される場合があります。もちろん、サービスが既に行われており、変更がない場合は、放置されます。

いずれの場合も、この手順の最後に Postgres は実行されます。

5.2.4 Postgres を起動した後

- *Creating Postgres users*
- *Creating Postgres tablespaces*
- *Creating Postgres databases*
- *pglogical*
- *Configuring .pgpass*

`postgres-config-final` を使用できます

起動後の Postgres 構成が完了した後にタスクを実行するため SQL クエリを実行してオブジェクトを作成またはデータをロードするためなど。

- *EDB Postgres Distributed configuration*

`展開後` を使用できます

展開プロセスが完了した後にタスクを実行します。

5.3 Building from source

!!!警告

このオプションは、開発者および上級ユーザーを対象としています。EDB によって構築およびテストされたソフトウェアのみが EDB によってサポートされています。 [Self-Managed Supported Open Source Software](#) を参照してください。

特定のコンポーネントのパッケージインストールを他のコンポーネントのソースビルドと組み合わせることもできます。たとえば、パッケージから Postgres をインストールし、ソースから pglogical と PGD をコンパイルできますが、パッケージの依存関係により、ソースから pglogical をインストールし、パッケージから PGD をインストールできません。

ソースビルドは、開発、テスト、およびサポート操作で使用することを目的としています。

5.3.1 クイックスタート

2ndQPostgres、pglogical3、および bdr はすべて安定したブランチからビルドされたものを使用してクラスターをスピンアップします。

```
$ tpaexec configure ~/clusters/speedy -a BDR-Always-ON \
  --layout bronze \
  --harp-consensus-protocol etcd \
  --install-from-source \
  2ndqpostgres:2QREL_13_STABLE_dev \
  pglogical3:REL3_7_STABLE \
  bdr3:REL3_7_STABLE
```

上記と同様ですが、公式 git リポジトリから 2ndQPostgres ソースコードをビルドし、指定されたローカルワークツリーを使用して pglogical および BDR をビルドするクラスターをセットアップします。この機能は Docker に固有です。

```
$ tpaexec configure ~/clusters/speedy \
  --architecture BDR-Always-ON --layout bronze \
  --harp-consensus-protocol etcd \
  --platform docker \
  --install-from-source 2ndqpostgres:2QREL_13_STABLE_dev \
  pglogical3 bdr3
```

(次のページに続く)

(前のページからの続き)

```
--local-source-directories \
pglogical3:~/src/pglogical bdr3:~/src/bdr
```

クラスターを展開した後、後続の実行で `tpaexec deploy ... --skip-tags build-clean` を使用して、ビルドディレクトリを再利用できます。それ以外の場合、ビルドディレクトリはビルドを開始する前に空にされます。

カスタムの場所とビルドパラメーターを使用して Postgres、pglogical、BDR、およびその他のコンポーネントをビルドする方法の詳細な説明を読んでください。

5.3.2 構成

ソースビルドの構成には2つの側面があります。

特定のソースの組み合わせを実行するクラスターのみが必要な場合は、`tpaexec configure` を実行して、選択したコンポーネントをダウンロード、コンパイル、およびインストールするための合理的なデフォルトで構成を生成します。上記の例に示すように、Postgres または Postgres Extended、pglogical、および BDR をビルドし、ビルドするブランチ名を指定できます。

基になるメカニズムは、コマンドラインオプションで許可される以上の機能を備えています。`config.yml` を編集することにより、さまざまなソースリポジトリのクローンを作成したり、ビルドの場所を変更したり、別の `configure` または `build` パラメーターを指定したり、`build` コマンドを完全に再定義したりできます。したがって、Postgres、pglogical、および BDR 以外のものをビルドできます。

使用可能なオプションはここに文書化されています。

- [Postgres source installation](#)
- [ソースからのビルドとインストール](#)

5.3.3 ローカルソースディレクトリ

TPA を使用して、Git リポジトリからではなくローカルソースディレクトリから Postgres および/または拡張機能をビルドする Docker コンテナをプロビジョニングできます。

`--install-from-source` を使用して、ビルドするものを宣言しているとします。

```
$ tpaexec configure ~/clusters/speedy \
--architecture BDR-Always-ON --layout bronze \
--harp-consensus-protocol etcd \
--platform docker \
--install-from-source 2ndqpostgres:2QREL_13_STABLE_dev \
pglogical3:REL3_7_STABLE bdr3:REL3_7_STABLE \
...
```

デフォルトでは、これは Postgres Extended、pglogical3、および bdr3 の既知のリポジトリのクローンを作成し、指定されたブランチをチェックアウトして、ビルドします。ただし、`--local-source-directories` を追加して、代わりにホストマシンからソースを直接取得するように指定できます。

```
$ tpaexec configure ~/clusters/speedy \
  --architecture BDR-Always-ON --layout bronze \
  --harp-consensus-protocol etcd \
  --platform docker \
  --install-from-source 2ndqpostgres:2QREL_13_STABLE_dev \
  pglogical3 bdr3 \
  --local-source-directories \
  pglogical3:~/src/pglogical bdr3:~/src/bdr \
  ...
```

この構成でも Postgres Extended はリポジトリからインストールされますが、ホスト上の指定されたディレクトリから pglogical3 および bdr3 ソースを取得します。これらのディレクトリは、git リポジトリがクローン作成されるのと同じ場所にある Docker コンテナに読み取り専用でバインドマウントされ、デフォルトのソリ外ビルドが通常のように続行されます。

コンポーネントのローカルソースディレクトリを指定する場合、ビルドするブランチを指定することはできません。上記の例の `--install-from-source` の `pglogical3:REL3_7_STABLE` と `pglogical3` を参照してください。ソースディレクトリはコンテナに読み取り専用でマウントされるため、TPA はそれを変更することはできません。git pull も git checkout も。ローカルにチェックアウトしたブランチ、コミットされていない変更などを取得します。

`--local-source-directories` を使用すると、`config.yml` に Docker ボリューム定義のリストが含まれます。

local_source_directories:

- /home/ams/src/pglogical:/opt/postgres/src/pglogical:ro
- /home/ams/src/bdr:/opt/postgres/src/bdr:ro
- ccache-bdr_src_36-20200828200021:/root/.ccache:rw

ccache

TPA は、すべての種類のソースビルドにデフォルトで ccache をインストールします。ローカルソースディレクトリで Docker クラスターを使用している場合、デフォルトでは、新しい Docker ボリュームがクラスターのコンテナに接続され、共有 ccache ディレクトリとして機能します。このボリュームはホストから完全に分離され、クラスターがプロビジョニング解除されるときに削除されます。

`--shared-ccache /path/to/host/ccache` オプションを使用して、有効期間の長い共有 ccache ディレクトリを指定します。このディレクトリは、コンテナに `r/w` バインドマウントされ、そのコンテンツはホストとコンテナの間で共有されます。

デザインとして、ホストでコンパイルされたバイナリをコンテナに直接インストールする方法はありません。

5.3.4 リビルド

ソースからビルドされたコンポーネントを使用してクラスターを展開した後、`tpaexec rebuild-sources` コマンドを使用して `tpaexec deploy` を再実行せずに、これらのコンポーネントをすばやくリビルドできます。これにより、コンテナ上の git リポジトリからビルドされたコンポーネントに対して `git pull` が実行され、すべてのコンポーネントがリビルドされます。

5.4 TPA hooks

TPA は、ユーザーの介入なしで完全な機能のクラスターをセットアップでき、カスタムリポジトリとパッケージ、カスタム Postgres 構成 `pg_hba.conf` と `postgresql.conf` の両方を含むさまざまな *Instance configuration* を既に提供しています。

フックスクリプトを作成して、使用可能な構成設定では満たされない特定のニーズに対応できます。フックを使用すると、展開中に任意の Ansible タスクを実行できます。

フックは TPA の究極の拡張メカニズムであり、それらを使用してできることに制限はありません。それらは注意して使用し、追加のメンテナンス負担が発生することに注意してください。TPA 開発者はフックコードについての洞察がなく、予想される段階でフックを呼び出すことを超えて、リリース間の互換性を保証できません。

5.4.1 概要

クラスターディレクトリの `hooks` サブディレクトリの下に特定の名前のファイルを作成すると、TPA は以下で説明するように、展開プロセスのさまざまな段階でそれら呼び出します。

```
$ mkdir ~/clusters/speedy/hooks
$ cat > ~/clusters/speedy/hooks/pre-deploy.yml
- --
-  debug: msg="hello world!"
```

フックスクリプトは `include_tasks` で呼び出されるため、Ansible タスクのリストを含む YAML ファイルであることが想定されます。プレイのリストを含むプレイブックではありません。以下に特に記載されていない限り、フックは展開内のすべてのホストに対して無条件に実行されます。

5.4.2 汎用フック

展開前

TPA は、Python をブートストラップした後すぐに `hooks/pre-deploy.yml` を呼び出しますが、リポジトリの構成やパッケージのインストールなど、他のことを行う前に。これは、独自のコードを実行できる最初の段階です。

このフックを使用して、*Configuring APT repositories* または *Configuring YUM repositories* でできることを超えて、カスタムリポジトリ構成をセットアップできます。

ポストリポジトリ

TPA は、パッケージリポジトリの構成後に `hooks/post-repo.yml` を呼び出します。これを使用して、パッケージのインストールを開始する前にリポジトリ構成を修正できます。

pre-initdb

TPA は、*Running initdb* が存在しない場合、*Running initdb* を決定する前に `hooks/pre-initdb.yml` を呼び出します。通常、このフックを使用する必要はありませんが、これを使用して PGDATA を自分自身で作成すると、TPA は `initdb` をスキップします。

postgres-config

TPA は、`pg_hba.conf` および `conf.d` のファイルを含む Postgres 構成ファイルを生成した後、サーバーが起動する前に `hooks/postgres-config.yml` を呼び出します。

このフックを使用して、たとえば、`conf.d` の下に追加の構成ファイルを作成できます。

postgres-config-final

TPA は、Postgres を起動し、ユーザー、データベース、および拡張機能を作成した後、`hooks/postgres-config-final.yml` を呼び出します。このフックを使用して、SQL コマンドを実行できます。たとえば、カスタム拡張機能の構成を実行したり、データベースオブジェクトを作成したりできます。

barman-pre-config

TPA は、Barman をインストールして Barman ユーザーを設定した後、Barman 構成を生成する前に `hooks/barman-pre-config.yml` を呼び出します。

このフックを使用して、たとえば、Barman 証明書ファイルまたはマウントポイントに関連するタスクを実行できます。

efm-pre-config

TPA は、efm のインストール、構成ディレクトリの作成、および efm ユーザーのセットアップの後、efm 構成を生成する前に `hooks/efm-pre-config.yml` を呼び出します。

このフックの使用例は、efm ヘルパースクリプトをインストールすることです。

harp-config

TPA は、HARP 構成ファイルを生成した後、HARP サービスが開始される前に `hooks/harp-config.yml` を呼び出します。

このフックを使用して、たとえば、TPA のビルトインインターフェイスによって提供されない HARP プロキシのカスタマイズを実行できます。

このフックは、PGD ノードを含む HARP パッケージをインストールするノードで実行されることに注意してください。

pgd-proxy-config

TPA は、PGD PROXY 構成ファイルを生成した後、PGD PROXY サービスが開始される前に、`hooks/pgd-proxy-config.yml` を呼び出します。

このフックを使用して、TPA のビルトインインターフェイスでは提供されていない PGD プロキシのカスタマイズを実行できます。

このフックは、PGD PROXY パッケージをインストールするノードで実行されることに注意してください。

展開後

TPA は、展開の最後に `hooks/post-deploy.yml` を呼び出します。

この段階の後には、好きなことを続けることができます。

このフックを使用して、TPA 展開中に生成または変更された構成ファイルを変更すると、次の `tpaexec deploy` が変更を上書きするリスクがあります TPA はフックが何をしたのかを知りません。

5.4.3 PGD フック

これらのフックは、PGD 展開に固有です。

bdr-pre-node-creation

TPA は、インスタンスに初めて PGD ノードを作成する前に、すべてのインスタンスで `hooks/bdr-pre-node-creation.yml` を呼び出します。必要なすべての PGD ノードが既に存在する場合、フックは呼び出されません。

bdr-post-group-creation

TPA は、`first_bdr_primary` インスタンスで PGD ノードグループを作成した後、すべてのインスタンスで `hooks/bdr-post-group-creation.yml` を呼び出します。必要な PGD グループが既に存在する場合、フックは呼び出されません。

bdr-pre-group-join

TPA は、レプリケーションセットを作成、変更、または削除し、必要なサブスクリプションを構成した後、ノードに参加する前に、すべてのインスタンスで `hooks/bdr-pre-group-join.yml` を呼び出します。

このフックを使用して、SQL コマンドを実行し、ノード参加が開始する前に必要になるレプリケーションセットの構成とサブスクリプションに対してその他の調整を実行できます。

たとえば、PGD 監視レプリケーションセットを調整して、新しいテーブルを自動的に追加し、一般的に DDL フィルタを作成できます。

5.4.4 その他のフック

postgres-pre-update、postgres-post-update

Upgrading your cluster コマンドは、パッケージをインストールする前に特定のインスタンスで `hooks/postgres-pre-update.yml` を呼び出し、パッケージのインストールが完了した後に `hooks/postgres-post-update.yml` を呼び出します。どちらのフックも、更新されるインスタンスでのみ呼び出されます。

これらのフックを使用して、環境に合わせて更新プロセスをカスタマイズできますたとえば、他のパッケージをインストールし、TPA が管理しないサービスを停止および再起動します。

5.4.5 新しいフック

EDB は、必要に応じて TPA に新しいフックを追加します。ユースケースが既存のフックでカバーされていない場合は、私たちに連絡して問題について話し合ってください。

第6章 Cluster Management

6.1 Upgrading your cluster

`tpaexec upgrade` コマンドは、TPA クラスターで実行されているソフトウェアをアップグレードするために使用されます。`tpaexec deploy` はアップグレードを実行しません。

このコマンドは、以前の `tpaexec update-postgres` コマンドを置き換えます。

6.1.1 はじめに

`config.yml` に変更を加えた場合、それらの変更を適用する方法は、`tpaexec provision` に続いて `tpaexec deploy` を実行することです。

この規則の例外は、`tpaexec deploy` が既にインストールされているパッケージの別のバージョンのインストールを拒否することです。代わりに、`tpaexec upgrade` を使用してソフトウェアアップグレードを実行する必要があります。

TPA がアップグレードできるものはアーキテクチャによって異なります。

- M1 アーキテクチャと、M1 に該当するすべてのフェールオーバーマネージャー、`upgrade` は、Postgres のマイナーバージョンのアップグレードのみを実行します。
- PGD アーキテクチャでは、`upgrade` は Postgres と BDR 拡張機能のマイナーバージョンのアップグレードを実行します。
- PGD アーキテクチャでは、`reconfigure` コマンドとの組み合わせでのみ、`upgrade` は BDR 拡張機能のメジャーバージョンのアップグレードを実行できます。

他のクラスターコンポーネントのアップグレードのサポートは、将来のリリースで計画されています。

アップグレードするときは、アップグレードを開始する前に常に `barman` を使用してバックアップをとり、アップグレードのために確保されている時間中に実行されるスケジュールされたバックアップを無効にする必要があります。

一般に、TPA はインスタンスごとに続行し、影響を受けるサービスの停止、新しいパッケージのインストール、必要に応じて構成の更新、サービスを再起動し、ランタイム構成の変更を実行してから、次のインスタンスで同じことを行う前に。プロセス中の任意の時点で、クラスターのノードの 1 つだけが使用不可になります。

クラスターを PGD-Always-ON にアップグレードする場合、または既存の PGD-Always-ON クラスターをアップグレードする場合、`tpaexec upgrade` コマンドラインにオプション `-e enable_proxy_monitoring=true` を追加することにより、アップグレード中にプロキシノードのステータスのモニタリングを有効にできます。有効にすると、bdr データベースに追加のテーブルが作成され、アップグレードの実行中に監視データ

がそこに書き込まれます。モニタリングを有効にすることによるパフォーマンスへの影響は非常に小さいため、有効にすることをお勧めします。

6.1.2 構成

多くの場合、マイナーバージョンのアップグレードでは、`config.yml` を変更する必要はありません。 `tpaexec upgrade` を実行するだけで、インストールされたパッケージの利用可能な最新バージョンに優雅な方法でアップグレードします。それが正確に何を意味するかは、クラスターの詳細によって異なります。

場合によっては、アップグレードには、新しいパッケージのインストールとサービスの再起動を超える追加の手順が含まれる場合があります。たとえば、BDR4 から PGD5 にアップグレードするには、新しいパッケージリポジトリをセットアップし、プロセス中に BDR ノードとグループ構成に特定の変更を加える必要があります。

このような場合、ソフトウェアのアップグレードを有効にするプロセスの一部として必要な複雑な手順がある場合、`tpaexec upgrade` はこれらの手順を実行します。たとえば、上記のシナリオでは、新しい PGD5 パッケージリポジトリを構成します通常、これは `deploy` でも行います。

ただし、アップグレードプロセス自身自身に直接必要な変更のみを行います。たとえば、`config.yml` を編集して新しい Postgres ユーザーまたはデータベースを追加する場合、これらの変更はアップグレード中に行われません。混乱を避けるために、ソフトウェアのアップグレードプロセスを開始する前に、無関係の保留中の変更を `tpaexec deploy` することをお勧めします。

6.1.3 BDR-Always-ON から PGD-Always-ON へのアップグレード

BDR-Always-ON から PGD-Always-ON つまり BDR3/4 から PGD5 にアップグレードするには、最初に `tpaexec reconfigure` を実行します。

```
$ tpaexec reconfigure ~/clusters/speedy\  
--architecture PGD-Always-ON\  
--pgd-proxy-routing local
```

このコマンドは、`config.yml` を読み取り、クラスターをアップグレードするために必要な変更を行い、新しい `config.yml` を書き込みます。呼び出しの詳細については、[tpaexec reconfigure](#) を参照してください。変更を確認した後、`tpaexec upgrade` を実行してアップグレードを実行します。

```
$ tpaexec upgrade ~/clusters/speedy\  

```

または、プロキシモニタリングを有効にしてアップグレードを実行するには、

```
$ tpaexec upgrade ~/clusters/speedy\  
-e enable_proxy_monitoring=true
```

`tpaexec upgrade` は `tpaexec provision` を自動的に実行して、Ansible インベントリを更新します。アップグレードプロセスでは、次のことを行います。

1. クラスターをアップグレードするためのすべての前提条件が満たされていることを確認します。
2. クラスター内のインスタンスごとに、正しいリポジトリが構成されていること、および必要な postgres パッケージがそれらの中で利用可能なことを確認します。

3. クラスター内の BDR ノードごとに、一度に 1 つずつ
 - harp-プロキシがそれに接続を送信しないことを保証にノードをフェンスします。
 - BDR4 を PGD5 に置き換えるなど、postgres を停止、更新、および再起動します。
 - ノードのフェンスを解除して、接続を再度受信できるようにします。
 - このノードに適用される pgbouncer および pgd-cli を更新します。
4. クラスター内のインスタンスごとに、BDR v5 専用の BDR 構成を更新します
5. クラスター内のプロキシノードごとに、一度に 1 つずつ
 - pgd-proxy.
 - をセットアップします harp-proxy.
 - を停止します pgd-proxy を起動します。
6. harp-proxy とそのサポートファイルを削除します。

6.1.4 PGD-Always-ON から PGD-X へのアップグレード

PGD-Always-ON クラスターの PGD-X へのアップグレードは、**重要なアーキテクチャの進化**であり、単純な **ソフトウェアの更新**を超えた変更を含みます。これは**注意深く調整されたマルチステージプロセス**であり、最終的なソフトウェアアップグレードを実行する前に、個別のフェーズでクラスターを再構成する必要があります。このプロシージャは、最初に **pgd-proxy** をビルトイン **Connection Manager** に置き換えることにより、PGD 5 クラスターの接続処理を最新化します。このステップは、現在ライブクラスターでの手動操作を必要としますが、将来の TPA リリースで自動化が計画されています。次に、クラスターを新しい PGD-X に移行します。アーキテクチャ。

アップグレードプロセスは、クラスターを 3 つの異なる状態を介して移行します。

1. **開始** PGD 5.9+ (PGD-Always-ON) PGD-Proxy を使用
2. **中級** PGD 5.9+ (PGD-Always-ON) ビルトイン **Connection Manager** を使用するようになりました
3. **最終** PGD 6 (PGD-X Architecture)

前提条件

開始する前に、次の要件を満たしていることを確認してください。

- **クラスターバージョン** クラスターは PGD バージョン 5.9 以降を実行している必要があります。以前の 5.x バージョンを使用している場合は、**tpaexec upgrade** を使用して最初に最新のマイナーバージョンにアップグレードします。PGD-Always-ON クラスターのマイナーバージョンのアップグレードの詳細については、セクション#pgd-always-on を参照してください。
- **バックアップ** クラスターの現在のテストされたバックアップがあります。
- **オーバーライドの確認** インスタンスレベルのプロキシオーバーライドたとえば **pgd_proxy_options** などについて **config.yml** を確認しました。これらは自動的に移行できないため、手動での介入が必要です。

- 共同ホストプロキシ PGD 5 クラスターは、共同ホストプロキシを使用して構成する必要があります。ここで pgd-proxy ロールは bdr ロールと同じインスタンスにあります。スタンドアロンプロキシインスタンスは、このアップグレードパスではサポートされていません。

ステージ 1 ビルトイン接続マネージャーへの移行

最初のステージは、PGD 5.9+ クラスターを再構成して、外部 pgd-proxy の使用から最新のビルトイン Connection Manager に切り替えることです。

!!!移行状態のみに注意してください

このプロセスは、PGD 6 にアップグレードする前の中間ステップとしてのみを目的とした移行的な PGD 5.9+ クラスター状態を作成します。TPA は現在、この特定の Connection Manager 構成での tpaexec upgrade の使用をサポートしていません。将来の TPA リリースは、Connection Manager を使用した PGD 5 のライフサイクル管理を完全にサポートします。

この段階では、構成の変更を適用するために、ライブクラスターに大幅な手動介入が含まれます。これらの手順の実行に慣れていない場合は、このプロセスを完全に自動化する将来の TPA リリースを待つことをお勧めします。

次のコマンドを実行して、config.yml ファイルを更新します。これにより、ビルトイン Connection Manager を有効にするために必要な設定が追加されます。

このアクションは構成ファイルのみを変更します。データベースクラスターの実行状態はまだ変更されません。

新しいバージョンを書き込む前に、reconfigure は、現在のファイルたとえば config.yml.~1~ のバックアップを自動的に保存し、安全な復元ポイントを提供します。

呼び出しの詳細については、[tpaexec reconfigure](#) を参照してください。

```
$ tpaexec reconfigure ~/clusters/speedy --enable-connection-manager
```

ステップ 1.2 構成を適用し、接続マネージャーをアクティブにする

構成の変更をライブクラスターに適用します。これは **手動** 操作タスクであり、Postgres 構成パラメーターの追加、pgd-proxy サービス、および PostgreSQL ノードをローリング方法で再起動して Connection Manager をアクティブにすることが含まれます。

このプロセスの詳細な手順については、公式 [Connection Manager MigrationGuide](#) に従ってください。

ステージ 1 完了

このステージの最後に、ビルトイン Connection Manager で実行される PGD クラスターが作成されます。これは中間状態であり、ステージ 2 に直接進む必要があります。マイナーバージョンアップグレードの tpaexec upgrade は、この中間状態では **サポートされていません** が、PGD 6 へのアップグレードが完了するまで tpaexec deploy を実行することをお勧めします。

ステージ 2 アーキテクチャの PGD-X へのアップグレード

クラスターが **Connection Manager** で実行されたら、最後の構成手順に進んで、PGD 6 アップグレードの準備をできます。

注釈

Stage 1 が正常に完了し、ビルトイン **Connection Manager** で実行されているクラスターからこのプロセスを開始する必要があります。

次のコマンドを実行して、新しいアーキテクチャに合わせて `config.yml` を更新します。これにより、クラスターアーキテクチャタイプが変更され、BDR バージョンが 6 に設定され、古い古い設定が削除されます。

このアクションは構成ファイルのみを変更します。データベースクラスターの実行状態はまだ変更されません。

```
$ tpaexec reconfigure ~/clusters/speedy --architecture PGD-X
```

ステップ 2.2 ソフトウェアのアップグレードを実行する

`config.yml` の最終的な変更を確認した後、標準の `tpaexec upgrade` コマンドを実行できます。これにより、すべてのノードでソフトウェアのアップグレードが実行され、クラスターが PGD 6 になります。

```
$ tpaexec upgrade ~/clusters/speedy
```

または、プロキシモニタリングを有効にしてアップグレードを実行するには、

```
$ tpaexec upgrade ~/clusters/speedy\  
-e enable_proxy_monitoring=true
```

`tpaexec upgrade` は `tpaexec provision` を自動的に実行して、Ansible インベントリを更新します。アップグレードプロセスでは、次のことを行います。

1. クラスターをアップグレードするためのすべての前提条件が満たされていることを確認します。
2. クラスター内のインスタンスごとに、正しいリポジトリが構成されていること、および必要な postgres パッケージがそれらの中で利用可能なことを確認します。
3. クラスター内の BDR ノードごとに、一度に 1 つずつ
 - ノードをフェンスオフして、接続がないようにします。
 - PGD5 を PGD6 に置き換えるなど、postgres を停止、更新、および再起動します。
 - ノードのフェンスを解除して、
 - このノードに適用される pgbouncer および pgd-cli を更新します。
4. BDR v6 専用の BDR 構成を適用します

アップグレードの完了

クラスターは PGD-X アーキテクチャで PGD 6 を実行し、通常のように `tpaexec deploy` と `tpaexec upgrade` の両方で完全に管理可能です。

6.1.5 PGD-Always-ON

既存の PGD-Always-ON PGD5 クラスターを最新の利用可能なソフトウェアバージョンにアップグレードする場合、アップグレードプロセスは次を実行します。

1. クラスターをアップグレードするためのすべての前提条件が満たされていることを確認します。
2. クラスター内のインスタンスごとに、正しいリポジトリが構成されていること、および必要な postgres パッケージがそれらの中で利用可能なことを確認します。
3. クラスター内の BDR ノードごとに、一度に 1 つずつ
 - `pgd-proxy` が接続を送信しないようにノードをフェンスオフします。
 - postgres を停止、更新、および再起動します。
 - ノードのフェンスを解除しますので、
 - このノードに適用される `pgbouncer`、`pgd-proxy`、および `pgd-cli` を更新します。

6.1.6 BDR-Always-ON

BDR-Always-ON クラスターの場合、アップグレードプロセスはクラスターインスタンスを 1 つずつ実行し、次のことを実行します。

1. サーバーがメンテナンス中であることを `haproxy` に伝えます。
2. インスタンスがアクティブサーバーの場合、`pgbouncer` に再接続し、アクティブなセッションが閉じられるまで待機するように要求します。
3. Postgres を停止し、パッケージを更新し、Postgres を再起動します。
4. 最後に、`haproxy` を介して要求を受信するためにサーバーを再度「準備ができている」としてマークします。

PGD 論理スタンバイまたは物理レプリカインスタンスは、`haproxy` または `pgbouncer` の相互作用なしで更新されます。クラスター内の非 Postgres インスタンスは放置されます。

6.1.7 M1

注釈

M1 アーキテクチャは、Postgres のマイナーバージョンのアップグレードのみをサポートしています。M1 に該当するすべてのフェールオーバーマネージャーは、Postgres のマイナーバージョンのアップグレードを実行できます。

他のソフトウェアコンポーネントのマイナーアップグレードは、将来のリリースに追加される予定です。

プライマリからアップグレードされたレプリカのいずれかに移行し、プライマリを更新し、最初のプライマリノードにスイッチオーバーします。

6.1.8 アップグレードプロセスの制御

`update_hosts` 変数を定義することにより、クラスターのインスタンスがアップグレードされる順序を制御できます。

```
$ tpaexec upgrade ~/clusters/speedy \
-e update_hosts=quirk,keeper,quaver
```

これは、シャドウサーバーが最初にアップグレードされるように、アクティブな PGD プライマリインスタンスを最後にリストすることにより、アップグレード中のリード/シャドウのスイッチオーバーを最小限に抑えるのに役立つ場合があります。

環境で追加のアクションが必要な場合、*TPA hooks*

パッケージのインストール手順の前後にカスタム Ansible タスクを実行できます。

6.1.9 ノードのサブセットのアップグレード

`update_hosts` 変数を設定することにより、インスタンスのサブセットでローリングアップグレードを実行できます。ただし、この機能のサポートはアーキテクチャによって異なります。

- **M1** アーキテクチャの場合、この機能はそのすべてのアップグレードシナリオでサポートされています。
- **PGD-Always-ON/ BDR-Always-ON** の場合、これはマイナーバージョンのアップグレード中に **のみ** サポートされています。

PGD-Always-ON/ BDR-Always-ON のベストプラクティス

PGD ノードのサブセットでマイナーアップグレードを実行する場合、**RAFT リーダーノードを最後に更新することを強くお勧めします。この戦略により、クラスターが BDR の混合バージョンを実行している間のアップグレード後チェックに関する潜在的な問題を回避します。

6.1.10 パッケージバージョンの選択

デフォルトでは、クラスターを作成したときにパッケージバージョン Postgres、PGD、または pglogicalなどを明示的に指定しなかった場合、`tpaexec upgrade` はインストールされたパッケージの最新の利用可能なバージョンに更新します。

`--xxx-package-version` オプション postgres、bdr、pglogicalなどを **クラスター構成** に使用するか、`config.yml` で `xxx_package_version` 変数を定義することにより、特定のバージョンを選択した場合、インストールされたパッケージが既に要件を満たしているため、アップグレードは何も行いません。要求されたバージョン。

この場合、config.yml を編集し、バージョン設定を削除し、`tpaexec provision` を再実行する必要があります。更新により、最新の利用可能なパッケージがインストールされます。以下に示すように、コマンドラインでバージョンを指定することにより、特定のバージョンに更新できます。

```
$ tpaexec upgrade ~/clusters/speedy -vv \
-e postgres_package_version="2:11.6r2ndq1.6.13*" \
-e pglogical_package_version="2:3.6.11*" \
-e bdr_package_version="2:3.6.11"
```

ここでのバージョン構文は、OS ディストリビューションとパッケージマネージャーによって異なることに注意してください。特に、yum は `*xyz*` ワイルドカードを受け入れますが、apt は `xyz*` のみを理解します。上記の例のように。

注 [ワイルドカードの使用に関する既知の問題](#) の `package_version` でのワイルドカードの使用の制限を参照してください。

要求した Postgres、PGD、および pglogical パッケージバージョンの組み合わせが賢明であることを確認するのはあなたの責任です。つまり、これらは連携して動作し、インストールしたものから新しいバージョンへのアップグレードパスが存在する必要があります。

PGD クラスターの場合、パッケージマネージャーの依存関係の解決に依存して正しい依存関係を選択するのではなく、3つのコンポーネントすべて Postgres、PGD、pglogical の正確なバージョンを明示的に指定することをお勧めします。

運用環境で実行する前に、QA 環境でアップグレードをテストすることを強くお勧めします。

6.2 tpaexec switchover

`tpaexec switchover` コマンドは、*MI* のプライマリとレプリカの間で制御されたスイッチオーバーを実行します。このコマンドを実行すると、選択したレプリカが新しいプライマリに昇格し、元のプライマリは新しいレプリカになり、クラスター内の他のレプリカは新しいプライマリに従うように再構成されます。

このコマンドは、ロールを切り替える前にクラスターが正常であることを確認し、事前に `repmgr` サービスをシャットダウンせずに実行されるように設計されています。

これは、`--siblings-follow` オプションを指定して `repmgr standby switchover` を実行することと同じです。

6.2.1 例

このコマンドは、`replicaname` を `~/clusters/speedy` の新しいプライマリにします。

```
tpaexec switchover ~/clusters/speedy replicaname
```

6.2.2 アーキテクチャオプション

このコマンドは *MI* にのみ適用されます

1つの書き込み可能なプライマリインスタンスと1つ以上の読み取り専用レプリカがあります。

BDR-Always-ON クラスターの場合、*BDR/HAProxy server pool management* を使用して PGD インスタンスのメンテナンスを実行します。

6.2.3 Repmgr リダイレクト pgbouncer

repmgr をフェールオーバーマネージャーとして、pgbouncer を接続プーラーとして使用し、`repmgr_redirect_pgbouncer: true` を設定している場合、`switchover` コマンドは、pgbouncer インスタンスが新しいプライマリノードにリダイレクトされます。

```
# switchover to a replica
tpaexec switchover <cluster_name> <replica_name>
# revert to initial primary
tpaexec switchover <cluster_name> <initial_primary_name> -e"revert_redirect=true"
```

6.3 BDR/HAProxy server pool management

`tpaexec pool-disable-server` および `pool-enable-server` コマンドを使用すると、*BDR-Always-ON* の PGD インスタンスを、メンテナンスのために HAProxy アクティブサーバープールから一時的に削除し、その後に追加できます。

これらのコマンドは、デフォルトで *Upgrading your cluster* と同じプロセスに従うため、`pool-disable-server` は、PGD インスタンスに対するアクティブなトランザクションが完了し、pgbouncer が完了する前に新しい接続を別のインスタンスに転送するのを待機します。アクティブなセッションが終了するまで待機したくない場合は、`--nowait` オプションを使用します。

インスタンスで `pool-disable-server` の直後に `pool-enable-server` を実行すると、すべてのアクティブなトラフィックが別のインスタンス本質的にスイッチオーバーに移動する効果があります。これにより、インスタンスの可用性を維持しながら、長時間実行される `VACUUM` コマンドなどのオンラインメンテナンスタスクを実行できます。

`haproxy_backend_servers` の同じセットで構成された複数の HAProxy サーバーが存在する場合、このコマンドは、指定されたサーバーに関連するすべてのプロキシのプールに並行して削除または追加します。

6.3.1 例

```
$ tpaexec pool-disable-server ~/clusters/clockwork orange # --nowait

# HAProxy will no longer direct any traffic to the PGD instance named
# orange, so you can perform maintenance on it (e.g., run `tpaexec
# rehydrate`).

$ tpaexec pool-enable-server ~/clusters/clockwork orange
```

サーバープールからインスタンスを削除すると、プール内の他のインスタンスに障害が発生しても、HAProxy はトラフィックをそれに転送しません。メンテナンスアクティビティが終了したら、サーバーをアクティブなプールに追加することを忘れないでください。

6.4 tpaexec rehydrate

`tpaexec rehydrate` コマンドは、更新されたマシンイメージ AMI で AWS EC2 インスタンスを再構築し、TPA によって管理されるクラスターにセキュリティパッチと OS アップグレードを迅速に展開できます。

必要な変更をすべて備えた新しい AMI が与えられると、このコマンドはインスタンスを終了し、新しいイメージを使用する新しくプロビジョニングされたインスタンスに置き換え、サーバーの構成を正確に再作成する前に古いインスタンスからデータボリュームを接続します (`config.yml`)。

最新のイメージを公開し、定期的なスケジュールでサーバーをスクラッチから再構築することは、サーバーフリートに個々のセキュリティ更新を自分自身でダウンロードしてインストールできるようにするための代替手段です。これにより、各サーバーの状態を一目で追跡することが簡単になり、個々のサーバーへの手動変更を防ぎますインスタンスの置換中に一掃されます。

TPA を使用すると、リハイドレーション中にクラスター全体の中断を最小限に抑えることが簡単になりますが、プロセスには、個々のサーバーが終了および置き換えられるときに必然的にダウンタイムが含まれる必要があります。*M1* では、最初にレプリカを `rehydrate` から、*tpaexec switchover* を使用できます

リハイドレートする前にプライマリをレプリカに変換します。*BDR-Always-ON* では、リハイドレートする前に *BDR/HAProxy server pool management* を実行し、後で追加して戻すことができます。

Postgres および関連コンポーネントのマイナーバージョンの更新をインストールするだけの場合は、代わりに *Upgrading your cluster* コマンドを使用できます。

6.4.1 前提条件

インスタンスを `rehydrate` できるようにするには、`config.yml` のデータボリュームごとに `delete_on_termination: no` および `attach_existing: yes` を指定する必要があります。新しいインスタンスには、必ず新しい EBS ルートボリュームがあります。

デフォルトでは、EC2 インスタンスを終了すると、それにアタッチされている EBS ボリュームも終了します。この場合、新しいインスタンスに再アタッチしたいため、`delete_on_termination` を無効にする必要があります。`attach_existing` 設定により、TPA は、新しいインスタンスをプロビジョニングするときに古いボリュームを検索し、見つかった場合は、実行後にインスタンスにアタッチします。

古いインスタンスを手動で停止または終了しないでください。`tpaexec rehydrate` コマンドは、インスタンスを安全にリハイドレートできることを確認した後にこれを実行します。

6.4.2 例

`~/clusters/night` に AWS クラスター構成があると仮定します。

構成を変更する

最初に、`config.yml` を編集し、新しい AMI を指定する必要があります。例

```
ec2_ami:
  Name: RHEL-8.3_HVM-20210209-x86_64-0-Hourly2-GP2
  Owner: 309956199498
```

各データボリュームで `delete_on_termination` が無効になっていることを確認します。パラメーターが存在しない場合、AWS EC2 管理コンソールからその値を確認できます。’Instances’ をクリックし、インスタンスを選択してから、[Description] タブを開き、[Block devices] までスクロールし、EBS ボリュームをクリックします。“Delete on termination” フラグが `true` に設定されている場合、[付録](#) ことができます。また、`attach_existing` を確認し、まだ設定されていない場合は `yes` に設定します。

次に、両方の属性が正しく設定された例を示します。

```
instances:

- node: 1
  Name: vlad
  subnet: 10.33.14.0/24
  role: primary
  volumes:
  - device_name: /dev/xvdf
    volume_type: gp2
    volume_size: 16
    attach_existing: yes
    delete_on_termination: false
  vars:
    volume_for: postgres_data
    mountpoint: /var/lib/pgsql
```

ボリュームパラメーターは `instance_defaults` および特定のインスタンスで設定できることに注意してください。`volumes:` を検索し、関連するすべてのボリュームにこれらの2つの属性が設定されていることを確認します。

Rehydrate を開始します

次に、`rehydrate` コマンドの構文を示します。

```
$ tpaexec rehydrate ~/clusters/night instancename
```

単一のインスタンス名またはインスタンス名のコンマ区切りリストを指定できますが、クラスター内のすべてのインスタンスを一度に `rehydrate` ことはできません。

このコマンドは、リハイドレートされるインスタンスに接続されているすべての非ルート EBS ボリュームの `delete_on_termination` フラグが `false` に設定されていることを確認します。これに当てはまらない場合、インスタンスが終了する前にエラーで停止します。

ボリューム属性が正しく設定されている場合、コマンドは最初に各インスタンスを終了し、次にプロビジョニングと展開を実行して、新しい AMI を使用する新しいインスタンスに置き換えます。

6.4.3 段階的に Rehydrate する

クラスターの継続性を維持するために、クラスターを段階的にリハイドレートすることをお勧めします。

たとえば、プライマリインスタンス、2つのレプリカ、および Barman バックアップサーバーを備えた *MI* では、最初に Barman インスタンスと 1 つのレプリカを rehydrate、次に別のレプリカをリハイドレートし、次に *tpaexec switchover* プライマリからリハイドレートしたレプリカのいずれかに、前者を rehydrate ますプライマリ、およびオプションで元のプライマリにスイッチオーバーします。このシーケンスにより、1 つのプライマリと 1 つのレプリカが常に利用できるようになります。

6.4.4 付録

awscli を使用してボリューム属性を変更する

最初に、AWS 管理コンソールでインスタンスと EBS ボリュームを見つけます。'Instances' をクリックしてインスタンスを選択し、[Description] タブを開いて [Block devices] までスクロールし、EBS ボリュームを選択します。delete_on_termination を無効にするには、--region、--instance-id、およびブロックデバイス名を正しい値に置き換えた後、次のコマンドを実行します。

```
$ aws ec2 modify-instance-attribute \  
  --region eu-west-1 --instance-id i-XXXXXXXXXXXXXXXXX \  
  --block-device-mappings \  
    [{"DeviceName": "/dev/xvdf", "Ebs": {"DeleteOnTermination": false}}]
```

インスタンスのデータボリュームごとにこれを行います。少し遅延すると、管理コンソールに変更が表示されます。tpaexec rehydrate は、インスタンスを安全にリハイドレートできることも検出します。

第7章 Miscellaneous

7.1 TPA and Ansible Tower/Ansible Automation Platform

TPA は、RedHat Ansible Automation Platform(AAP) を介した展開をサポートしています。このサポートは、以下で詳細に説明するように、AAP で **deploy** および **upgrade** ステップを実行できるようにすることで機能します。デプロイまたはアップグレードを実行する前に、**tpa** パッケージがインストールされている別のスタンドアロンマシンで構成 **configure** コマンドとプロビジョニング **provision** コマンドを実行する必要があります。このスタンドアロンマシンで適切なオプションを使用して **configure** および **provision** を実行すると、結果のクラスターディレクトリを AAP にインポートできます。サポートはベアメタルプラットフォームに制限されています。

7.1.1 AAP の初期セットアップ

TPA が AAP を使用してクラスターを展開する前に、この初期設定を実行する必要があります。

TPA 実行環境イメージの追加 管理者

バージョン 2.4 以降、AAP はコンテナイメージを使用して Ansible プレイブックを実行します。これらのコンテナは実行環境 EE と呼ばれ、プレイブックが正常に実行するために必要な依存関係をバンドルしています。その結果、これは、必要な TPA 依存関係をすべて解決して使用するには、TPA を含む EE が必要で、展開とアップグレードの実行時に AAP が使用できることを意味します。

EDB の顧客は、EDB サポートに連絡して、EE に関するサポートを求めることができます。

AAP 管理者として、TPA 対応 EE イメージを指す使用可能な EE リストにエントリを作成します。

EDB_SUBSCRIPTION_TOKEN 資格情報タイプを作成します admin

AAP 管理者として、カスタム資格情報タイプ EDB_SUBSCRIPTION_TOKEN を作成して、EDB サブスクリプションアクセストークンを保持します。

1. AAP UI の [Credentials Type] ページに移動します。
2. **Name** フィールドを EDB_SUBSCRIPTION_TOKEN に設定します。
3. 次を **入力構成** フィールドに貼り付けます。

```
fields:
- id: tpa_edb_sub_token
  type: string
  label: EDB_SUBSCRIPTION_TOKEN
```

(次のページに続く)

(前のページからの続き)

```
secret: true
required:
- tpa_edb_sub_token
```

4. 次に **Injector Configuration** フィールドに貼り付けます。

```
env:
EDB_SUBSCRIPTION_TOKEN: {{ tpa_edb_sub_token }}
```

5. 変更を保存します。

6. 新しく追加されたタイプ EDB_SUBSCRIPTION_TOKEN を使用して資格情報を作成します。

7.1.2 クラスターのセットアップ

tpaexec パッケージがインストールされているワークステーションで初期手順を実行します。

TPA ワークステーションで

構成

次のオプションを含む `tpaexec configure` コマンドを実行します。--platform bare 、 --use-ansible-tower、 --tower-git-repository

```
[tpa]$ tpaexec configure <clustername> \
--platform bare \
--use-ansible-tower https://aac.example.com \
--tower-git-repository ssh://git@git.example.com/example \
--hostnames-from <hostnamefile> \
--architecture PGD-Always-ON \
--pgd-proxy-routing local \
--postgresql 16
```

--use-ansible-tower は、その時点で使用されない場合でも、パラメーターとして AAP アドレスを想定しています。--tower-git-repository は、クラスターデータを AAP にインポートするために使用されます。TPA は、ブランチ名として cluster_name を使用して独自のブランチを作成します。これにより、すべてのクラスターに同じリポジトリを使用できます。[クラスター構成](#) で説明しているように、tpaexec configure の他のすべてのオプションは引き続き有効です。

config.yml の変更

config.yml には、トップレベル辞書の ansible_tower が含まれ、これにより、tpaexec provision はクラスターを AAP 対応クラスターとして扱います。

config.yml を編集して、ノードごとに ansible_host および {private,public}_ip が定義され、ansible_host が AAP が解決可能な値に設定されていることを確認します。必要に応じて、さらに変更または追加を加えます。詳細は、[クラスター構成](#) を参照してください。

インベントリおよびその他の関連ファイルを生成するには、`tpaexec provision` を実行します。

AAP UI の場合

プロジェクト

git リポジトリをソースとして使用して、AAP にプロジェクトを追加します。TPA EE イメージを使用するようにプロジェクトのデフォルト EE を設定します。

単一のプロジェクトで複数のインベントリを使用しようとする場合、**Allow Branch Override** が必要です。

インベントリ

空のインベントリを追加します。プロジェクトを外部ソースとして使用し、インベントリファイルとして `inventory/00-cluster_name` を使用してデータを入力します。

AAP で TPA の制御の外部にある追加変数を設定しない場合、**リモートインベントリソースからローカル変数を上書きする**を使用することをお勧めします。

資格情報

`vault` 資格情報を作成します。TPA ワークステーションで `tpaexec show-vault <cluster_dir>` を使用して、ボールドのパスワードを取得できます。

展開中に SSH を介してインベントリノードに接続するには、マシンの資格情報が AAP で利用できることを確認してください。

テンプレートの作成

テンプレートを作成するには

1. プロジェクトとインベントリを使用するテンプレートを作成します。
2. 次の必要な資格情報を含めます。
 - Vault credential
 - EDB_SUBSCRIPTION_TOKEN credential
 - マシン資格情報
3. 2つの追加変数を設定します。

```
tpa_dir: /opt/EDB/TPA
cluster_dir: /runner/project
```

4. プレイブックとして `deploy.yml` を選択します。
5. クラスタを展開するには、新しいテンプレートに基づいてジョブを実行します。

7.1.3 複数のインベントリに1つのプロジェクトを使用する

TPA は、関連する git リポジトリのクラスターごとに異なるブランチ名を使用します。このアプローチでは、複数のクラスターで単一のプロジェクトを使用できます。

Allow ブランチオーバーライドオプションを設定します

AAP プロジェクトで、**Allow Branch override** オプションを有効にします。

複数のインベントリを定義する

TPA は、git リポジトリ内のクラスターごとに異なるブランチ名を使用します。インベントリごとにブランチをオーバーライドすることにより、同じプロジェクトをソースと使用して複数のインベントリを生成できます。

インベントリごとに資格情報を定義する

ボルトのパスワードは、TPA クラスターごとに異なるため、インベントリごとにそれに応じて設定されていることを確認します。

7.1.4 AAP で TPA を更新

AAP で TPA の更新には、いくつかの追加手順が含まれます。

TPA ワークステーションパッケージの更新

OS に応じて、OS パッケージと同様に TPA ワークステーションパッケージを更新します。*TPA installation* を参照してください。

同じバージョンタグで EE イメージを使用する

AAP の EE イメージを変更して、使用されるワークステーションパッケージバージョンと同じバージョンタグを使用します。

クラスターディレクトリで `tpaexec relink` を実行します

`tpaexec relink <cluster_dir> --force` を実行して、AAP を使用するクラスターが最新の状態であることを確認します。これを行う必要がある場合の例は、TPA インストールを新しいバージョンにアップグレードした後です。`relink` コマンドによってコミットされた変更は必ずプッシュしてください。

```
$ git status
On branch cluster_name
Your branch is ahead of tower/cluster_name by 1 commit.
  (use "git push" to publish your local commits)
...
$ git push tower
```

プロジェクトとインベントリの同期

ジョブの起動時にリビジョンの更新 と 起動時の更新を使用するように設定されていない場合は、それぞれ AAP UI と関連インベントリでプロジェクトを同期します。

7.1.5 TPA の EE をビルドする

前提条件

独自の EE イメージをビルドするには、`ansible-builder` を使用することをお勧めします。

必要なものは次のとおりです。

1. `docker` または `podman`
2. `ansible-builder` および `ansible-navigator python` ツールキット
3. `TPA repo` からタグ `vA.B.C` でチェックアウトした TPA ソースコード `vA.B.C` は、使用する TPA バージョンです。

環境ファイル

`ansible-builder` は、環境ファイルを使用して、作業用 EE イメージを生成します。

次に、このような TPA の環境ファイルのテンプレート例を示します。

`execution-environment.yml`

```
version: 3
images:
  base_image:
    name: registry.redhat.io/ansible-automation-platform-24/ee-minimal-rhel9:latest
dependencies:
  python: << TPA_REPO_CLONE_FOLDER >>/requirements-aap.txt
  galaxy: << TPA_REPO_CLONE_FOLDER >>/collections/requirements.yml
options:
  package_manager_path: /usr/bin/microdnf

additional_build_steps:
  append_final:
    - RUN mkdir -p /opt/EDB/TPA
    - COPY << TPA_REPO_CLONE_FOLDER >> /opt/EDB/TPA
    - ENV PYTHONPATH="${PYTHONPATH:+${PYTHONPATH}:}/opt/EDB/TPA/lib"
```

異なるベースイメージには、より多くの `additional_build_steps` が必要になる場合があります。 `ansible-builder` を参照してください

高度な使用のために。

EE build コマンド

次のコマンドは、EE イメージをビルドします。

```
ansible-builder build \
  --file=execution-environment.yml \
  --container-runtime=<docker/podman> \
  --tag=<your-registry>/<your-namespace>/tpa-ee:vA.B.C \
  --verbosity 2
```

7.2 TPA, Ansible, and sudo

TPA は sudo で Ansible を使用して、ターゲットインスタンスで昇格した特権でタスクを実行します。このページでは、Ansible が sudo を使用方法決して TPA 固有ではありません、および TPA で管理されているシステムへの影響について説明します。

TPA には root 権限が必要です。

- パッケージをインストールするためオペレーティングシステムのネイティブパッケージマネージャーを使用した必須パッケージ、および pip を使用したオプションパッケージ
- サービスを停止、リロード、および再起動するため Postgres、repmgr、efm、etcd、haproxy、pgbouncer など
- 他のさまざまなタスクを実行するためたとえば、クラスターファクトの収集、スイッチオーバーの実行、クラスターノードのセットアップ

TPA は sudo を使用できることも必要です。 `ansible_user: root` を設定することにより、root として直接 ssh インすることができますが、引き続き sudo を使用して他のユーザー postgres などとしてタスクを実行します。

7.2.1 Ansible sudo 呼び出し

Ansible が sudo を使用してタスクを実行すると、ターゲットインスタンスに次のようなプロセスが表示されます。

```
/bin/bash -c sudo -H -S -n -u root /bin/bash -c \
  ""echo BECOME-SUCCESS-kfoodiiprztstyerrriqbjquhhbemejgpc ; \
  /usr/bin/python2"" && sleep 0
```

`sudo yum install -y xyzpkg` のようなものを期待していた人は、これに驚くことがよくあります。一般に、Ansible のほとんどのタスクは、認識可能なシェルコマンドを実行するのではなく、Python インタープリターを呼び出して Python コードを実行します。Playbook は raw シェルコマンドを実行できますが、TPA はそのようなタスクを Python インタープリターをブートストラップするためにのみ使用します。

Ansible モジュールには、さまざまな複雑さの Python コードが含まれており、Ansible プレイブックは YAML 形式で書かれた単なるシェルスクリプトではありません。任意の Ansible プレイブックを実行するのと同じことを行うシェルコマンドを「抽出」する方法はありません。

Ansible が `sudo privilegeescalation must be general` を使用方法の重要な結果が 1 つあります。つまり、一部の管理者が慣れているように、`sudoers.conf` で `sudo` 呼び出しを特定のコマンドに制限することはできません。ほとんどのタスクは Python を呼び出すだけです。すべてのコマンドにランダムな文字列がなければ、Python への `sudo` アクセスを制限できますが、Python が `root` として実行されると、とにかくできることに対する事実上の制限がありません。

ターゲットホストで Python モジュールを実行するのは、Ansible の動作です。これはどれも TPA に固有ではなく、これらの考慮事項は他の Ansible プレイブックに等しく適用されます。

7.2.2 おすすめ

- SSH 公開キーベースの認証を使用して、ターゲットインスタンスにアクセスします。
- SSH ユーザーがパスワードなしで `sudo` コマンドを実行できるようにします。
- コマンドではなく時間でアクセスを制限します。

TPA は、クラスターを最初にセットアップするとき、またはメンテナンスウィンドウなどに構成を変更するために `tpaexec deploy` を再度実行するときのみにアクセスが必要です。それまでは、そのアクセスを完全に無効にすることができます `ssh` と `sudo` の両方を 1 行の変更で。

デプロイメント中に、Ansible が行うことはすべて、プレイブックの動作と指定したパラメーターに基づいて予測可能であり、各アクションは、ターゲットインスタンスのシステムログ、および `tpaexec` 自分自身が実行されるマシンの Ansible ログに表示されます。

Ansible の焦点は、実行できるアクションにきめの細かい制限を課することよりも、実行時に何をするのかの可視性を提供することであるため、昇格した特権はスコープではなく時間でより適切に割り当ておよび管理されます。

7.2.3 SSH および `sudo` パスワード

パスワードなしの SSH キー認証とパスワードなしの `sudo` アクセスを設定することを強くお勧めしますが、パスワードを使用することも可能です。

`tpaexec` を実行する前に環境で `ANSIBLE_ASK_PASS=yes` および `ANSIBLE_BECOME_ASK_PASS=yes` を設定すると、Ansible はリモートサーバーのログインパスワードと `sudo` パスワードを入力するように求めます。次に、リモートサーバーで `login/sudo` パスワードプロンプトをネゴシエートし、指定したパスワードを送信しますこれにより、Playbook の実行時間が著しく長くなります。

パスワードの組み合わせを使用してアクセスを制限するよりも、必要がないときに特定のアカウントを介したアクセスを完全に無効にする方が効果的なセキュリティ制御であると考えられるため、この操作モードはお勧めしません。`ssh` の公開キー認証を使用すると、サーバーにアクセスできる人を効果的に制御できます。許可されたユーザーごとに 1 つのプライベートキーを保護することは、1 つまたは複数の共有パスワードを保護するよりも簡単です。また、`ssh / sudo` レベルでのアクセスを必要な場合に制限すると、メンテナンス期間にパスワードは追加のセキュリティを追加しません。

7.2.4 sudo オプション

sudo で Ansible を使用するには、sudoers.conf で `requiretty` を設定しないでください。

必要に応じて、ansible.cfg の [privilege_escalation] セクションで `become_flags`、または環境の `ANSIBLE_BECOME_FLAGS`、またはインベントリの `ansible_become_flags` を設定することにより、Ansible が使用する sudo オプション `-H -S -n` を変更できます。3つの方法はすべて同等ですが、特定の必要がある場合にのみ sudo オプションを変更してください。デフォルトが選択されたのには十分な理由があります。たとえば、パスワードなしの sudo が誤って構成されている場合、`-S -n` を削除するとタスクがタイムアウトします。

7.2.5 ロギング

プレイブックの実行の場合、sudo ログは主に Python の呼び出しを表示します誰かが `sudo -i` を使用するときに `bash` の呼び出しのみを表示するのと同様に。

詳細については、syslog には、ターゲットインスタンスでの各モジュール呼び出しの正確な引数が表示されます。そのモジュールが呼び出された理由の高レベルのビューについては、コントローラの `ansible.log` に、そのタスクがしようとしたことと結果が表示されます。

さらに詳細な監査データの独立したソースが必要な場合は、サーバーで `auditd` を実行し、SELinux ログファイルを使用できます。bpfftrace/bcc からさらに詳細なシステムコールレベルの情報を取得できます。たとえば、opensnoop はシステムで開かれているすべてのファイルを表示し、execsnoop はシステムで実行されたすべてのプロセスを表示します。ニーズに応じて、これらのことのいずれかまたはすべてを実行できますが、ロギングの増加に伴うオーバーヘッドの増加という明らかな注意事項。

7.2.6 ローカル権限

TPA installation

sudo は、「run これらのコマンドを root attempts で実行する」の短縮形としてのみ言及しています。TPA がインストールされ、`tpaexec setup` を実行すると、TPA 自分自身はローカルマシンでの昇格した特権を必要としません。ただし、Docker を使用する場合、Docker デモンへの接続が許可されたグループに属するユーザーとして `tpaexec` を実行する必要があります。

7.3 TPA - PuTTY Configuration guide

Windows で PuTTY を使用して、TPA ユーティリティ **tpaexec** **プロビジョニング***によって作成された AWS インスタンスに ssh を介して接続するには、Amazon が生成したプライベートキー形式.pem からキーを変換する必要があります EC2 を PuTTY 形式.ppk に変換します。

```
# Provision the cluster

[tpa]$ tpaexec provision <clustername>
```

PuTTY には、**PuTTYgen** という名前のツールがあり、キーを必要な形式に変換できます。

- **

7.3.1 キー変換

プライベートキーを見つけます

クラスターディレクトリ<clustername> でプライベートキーを見つけます。名前付けは、**id_** が接頭辞を付けた **config.yml** で設定された **cluster_name** 変数に従って名前付けられます。例、**cluster_name** が **testenv1** に設定されている場合、プライベートキーの名前は **id_testenv1** になります。

キーを.pem として保存

このファイルを Windows ファイルシステムにコピーし、.pem ファイルとして保存します。この例では **id_testenv1.pem** - テキストファイルにカットアンドペーストすると、これにうまく機能します。

キー変換

PuTTYgen を起動し、[パラメーター] の下で、生成するキーの適切なタイプを選択します。

PuTTYgen の古いバージョンの場合、**SSH-2 RSA** を選択します。最近のバージョンの場合 select **RSA**

- SSH-1 RSA は選択しないでください

次に、**Load** を選択します。 **PuTTY Private Key Files (*.ppk)** と記載されたボックスで、 **All Files (*.*)** を選択する必要があります。

.pem ファイルを選択し、 **Open** を選択して、 **OK** をクリックします。

Save private key を選択し、 **Yes** をクリックして、パスフレーズなしでキーを保存することに関する警告を無視します。ファイルのサフィックスが.ppkであることを確認し、.pem ファイルと同じ名前を選択します。この例では、ファイル名は **id_testenv1.ppk** になります。

- **

7.3.2 PuTTY の構成

PuTTY を起動し、 **Category** ウィンドウから **Session** を選択します。 **Host Name** パネルに<user>@<IP address> と入力し、ポートパネルに **22** と入力します。

<user> および<IP address> は、 ``tpaexec provision`` ユーティリティによって作成される<clustername>/ssh_config ファイルにあります。

Putty **Category** ウィンドウで、 **Connection** を選択し、 **SSH** を展開して **Auth** を選択します

認証用の秘密キーファイル *とマークされたパネルで、 **Browse** をクリックし、上記で保存した.ppk ファイルを選択し、 **Open** を選択します

Putty **Category** ウィンドウで、 **Session** を再度選択し、 **Saved Sessions** にセッション名を入力して、 **Save** この保存されたセッションを選択することにより、PuTTY を介して AWS ホストに接続できるようになります。

- **

7.4 Troubleshooting

7.4.1 Python 仮想環境を再作成します

時々、Python venv は一貫性のない状態になる場合があります。その場合、最も簡単な解決策は削除して再作成することです。壊れた venv の症状には、次のようなプロビジョニング中のエラーが含まれる場合があります。

```
TASK [Write Vagrantfile and firstboot.sh] *****
→ *****
→ *****
failed: [localhost] (item=Vagrantfile) => {"changed": false, "checksum":
→ "bf1403a17d897b68fa8137784d298d4da36fb7f9", "item": "Vagrantfile", "msg": "Aborting,
target uses selinux but python bindings (libselinux-python) arent installed!"}
```

新しい仮想環境を作成するには tpaexec がデフォルトの場所にインストールされていると仮定します。

```
[tpa]$ sudo rm -rf /opt/EDB/TPA/tpa-venv
[tpa]$ sudo /opt/EDB/TPA/bin/tpaexec setup
```

7.4.2 資格情報に関する奇妙な AWS エラー

TPA サーバーの時刻と日付が正しくない場合、プロビジョニング中にこれと同様の AWS エラーを取得する場合があります。

```
TASK [Register key tpa_cluster in each region] *****
→ *****
An exception occurred during task execution. To see the full traceback, use -vvv. The
→ error was: ClientError: An error occurred (AuthFailure) when calling the
→ DescribeKeyPairs operation: AWS was not able to validate the provided access
→ credentials
failed: [localhost] (item=eu-central-1) => {"boto3_version": "1.8.8", "botocore_
→ version": "1.11.8", "changed": false, "error": {"code": "AuthFailure", "message":
→ "AWS was not able to validate the provided access credentials"}, "item": "eu-
→ central-1", "msg": "error finding keypair: An error occurred (AuthFailure) when
→ calling the DescribeKeyPairs operation: AWS was not able to validate the provided
→ access credentials", "response_metadata": {"http_headers": {"date": "Thu, 27 Sep
→ 2018 12:49:41 GMT", "server": "AmazonEC2", "transfer-encoding": "chunked"}, "http_
→ status_code": 401, "request_id": "a0d905ba-188f-48fe-8e5a-c8d8799e3232", "retry_
→ attempts": 0}}
```

解決策-時刻と日付を正しく設定します。

```
[tpa]$ sudo ntpdate pool.ntp.org
```

7.4.3 ロギング

デフォルトでは、すべての tpaexec ログはログファイル<clusterdir>/ansible.log に保存されます
ログファイルの場所を変更するには、環境変数 `ANSIBLE_LOG_PATH` を目的の場所に設定します。たとえば、

```
export ANSIBLE_LOG_PATH=~/.ansible.log
```

ロギングの詳細度を高めるには、`-v` / `-vv` / `-vvv` / `-vvvv` / `-vvvvv` を tpaexec コマンドラインに追加するだけです。

```
[tpa]$ tpaexec deploy <clustername> -v

- v      shows the results of modules
- vv     shows the files from which tasks come
- vvv    shows what commands are being executed on the target machines
- vvvv   enables connection debugging, what callbacks have been loaded
- vvvvv  shows some additional ssh configuration, filepath information
```

7.4.4 クラスターテスト

既存のクラスターをスモークテストする簡単な方法は、次を実行することです。

```
[tpa]$ tpaexec test <clustername>
```

これにより、クラスターコンポーネントの機能テストが実行され、続いて pgbench を使用してクラスターのパフォーマンステストが実行されます。pgbench は完了までに時間がかかるため、次のコマンドを実行してベンチマークを省略できます。

```
[tpa]$ tpaexec test <clustername> --excluded_tasks=pgbench
```

7.4.5 TPA サーバーテスト

TPA サーバー自分自身のインストールを確認するには、次のコマンドを実行します。

```
[tpa]$ tpaexec selftest
```

7.4.6 特定のタスクを含めるまたは除外する

障害が発生した後、またはテストを実行しているときに tpaexec のプロビジョニングまたは展開を再実行する場合、TPA の *Selective task execution* を使用してタスクを見逃すことが役立つ場合があります。

7.5 Reference

7.5.1 Installation

Managing clusters in a disconnected or air-gapped environment

インターネットへの直接接続が許可されないセキュリティ制御された環境では、展開を完了するために TPA が必要とするすべてのパッケージを提供する必要があります。これは、クラスター内の各ノードのローカルリポジトリを介して実行できます。TPA は、*Creating and using a local repository* を介して各ノードでのカスタムリポジトリの追加をサポートし、必要なパッケージは `tpaexec download-packages` コマンドを使用してダウンロードできます。

準備

TPA をインストールできるインターネットに接続されたマシンを選択し、以下の手順に従って、既存のクラスター構成をコピーするか、新しいクラスターを作成します。

注釈

エアギャップサーバーに TPA がまだインストールされていない場合、次の指示に従ってください *TPA をインストールする場所*

インストールします。

切断された環境に既存のクラスターがある場合、インターネットに接続されたホストで必要なのは `config.yml` だけです。ディレクトリを作成し、そのファイルをコピーします。次に、そのディレクトリで `tpaexec relink` を実行して、通常は `tpaexec configure` によって作成される残りのファイルを生成します。

または、ターゲットインスタンスがネットワークアクセスを持たない環境の新しい構成を作成するには、このオプションを使用して新しいクラスターを構成します。

```
tpaexec configure --use-local-repo-only ...
```

これは、`--enable-local-repo` が行うことをすべて行い、他のすべてのパッケージリポジトリの構成を無効にします。RedHat インスタンスでは、これには、サブスクリプションベースのサービスへのアクセスの無効化も含まれます。

既存のクラスターでは、`config.yml` で `use_local_repo_only: yes` を設定できます。

```
cluster_vars:
  use_local_repo_only: yes
```

注インターネットに接続された環境と切断された環境で個別のクラスター構成は必要ないことに注意してください、以下のオプションは両方で機能します。

Creating and using a local repository の詳細

パッケージのダウンロード

インターネットに接続されたマシンで、*Docker のインストール* があることを確認し、実行します。

```
tpaexec download-packages cluster-dir --os <OS> --os-version <version>
```

tpaexec download-packages の詳細な説明を参照してください。

ターゲット環境へのパッケージのコピー

生成されるリポジトリは、`cluster-dir/local-repo` ディレクトリに含まれます。これは、ターゲット OS の完全なパッケージリポジトリです。このディレクトリを、接続されたコントローラから、クラスターの展開に使用される切断されたコントローラにコピーします。ディレクトリをクラスターディレクトリの下と同じ場所に配置します。TPA は、`deploy` の実行時にパッケージをインスタンスに自動的にコピーします。

切断された環境での展開

クラスター `config.yml` が **準備**で上記のように構成されていることを確認します。通常どおりに `tpaexec provision` および `deploy` を実行します。

切断された環境での更新

Upgrading your cluster コマンドを使用して、エアギャップ環境で更新を実行できます。このコマンドを実行する前に、接続されたコントローラで `download-packages` を実行し、更新されたりポジトリを切断されたコントローラにコピーする必要があります。

Distribution support

TPA は、各ターゲットインスタンスで実行されているディストリビューションを検出して適応します。このページには、アクティブにサポートされているプラットフォームと、以前にサポートされていた「古いディストリビューション」がリストされています。古いプラットフォームへの展開は、必要なパッケージにアクセスできる限り機能する可能性があります。これは TPA のサポートされている使用とは見なされず、運用使用には適していません。

完全にサポートされているプラットフォームは、TPA を実行するホストシステムとして、および TPA が Postgres クラスターを展開するターゲットシステムの両方としてサポートされています。

Debian ARM64

- Debian 12/bookworm は完全にサポートされています

Debian x86

- Debian 12/bookworm は完全にサポートされています
- Debian 11/bulseye は完全にサポートされています
- Debian 10/buster は古いディストリビューションです
- Debian 9/stretch は古いディストリビューションです
- Debian 8/jessie は古いディストリビューションです

Ubuntu x86

- Ubuntu 24.04/noble が完全にサポートされています
- Ubuntu 22.04/jammy は完全にサポートされています
- Ubuntu 20.04/focal は古いディストリビューションです

- Ubuntu 18.04/bionic は古いディストリビューションです
- Ubuntu 16.04/xenial は古いディストリビューションです

Oracle Linux x86

- Oracle Linux 9.x が完全にサポートされています docker のみ
- Oracle Linux 8.x が完全にサポートされています docker のみ
- Oracle Linux 7.x は古いディストリビューション Docker のみ

RedHat x86

- RHEL / Rocky / AlmaLinux / Oracle Linux 9.x が完全にサポートされています python3 のみ
- RHEL / CentOS / Rocky / AlmaLinux 8.x が完全にサポートされています python3 のみ
- RHEL / CentOS 7.x は古いディストリビューション python2 のみ

RedHat ppc64le

- RHEL / Rocky / AlmaLinux 9.x が完全にサポートされています python3 のみ
- RHEL / AlmaLinux 8.x が完全にサポートされています python3 のみ

SLES x86

- SLES 15.x が完全にサポートされています

プラットフォーム固有の考慮事項

一部のプラットフォームは、ここで記載されている古いディストリビューションで動作しない場合があります。たとえば、Debian 8 および Ubuntu 16.04 は [Docker のインストール](#) では利用できません。

Running TPA in a Docker container

利用可能な *TPA installation* がなく、*Installing TPA from source* の後に TPA を実行することが難しい場合たとえば、動作する Python 3.12 以降のインタープリターを入手するのは簡単ではないため、最後の手段は Docker イメージをビルドすることですそして、Docker コンテナ内で TPA を実行します。

[Docker のインストール](#) を行うために Docker コンテナで TPA を実行する必要はないことに注意してください。可能であれば、MacOS X でも TPA を直接実行することをお勧めします。

クイックスタート

システムに Docker をインストールして動作している必要があります。

次のコマンドを実行して、Github から tpaexec ソースリポジトリのクローンを作成し、tpa/tpaexec という名前の新しい Docker イメージをビルドします。

```
$ git clone ssh://git@github.com/EnterpriseDB/tpa.git
$ cd tpa
$ docker build -f docker/Dockerfile --build-arg TPA_VER=$(git describe) -t
→tpaexec:latest .
```

作成したイメージをダブルチェックします。

```
$ docker image ls tpaexec
REPOSITORY    TAG          IMAGE ID      CREATED        SIZE
tpaexec       latest      3943dec4d660  20 minutes ago 658MB

$ docker run --rm tpaexec info

# TPAexec v23.38.0-38-g4dc030dc1

tpaexec=/usr/local/bin/tpaexec
TPA_DIR=/opt/EDB/TPA
PYTHON=/opt/EDB/TPA/tpa-venv/bin/python3 (v3.13.5, venv)
TPA_VENV=/opt/EDB/TPA/tpa-venv
ANSIBLE=/opt/EDB/TPA/tpa-venv/bin/ansible (v2.16.14)
```

次に、実行しているシェルセッションで `tpaexec` のエイリアスを設定する必要があります。

```
alias tpaexec="docker run --rm -v $PWD:/work -v /var/run/docker.sock:/var/run/docker.
→sock tpaexec"
```

これで、次のようなコマンドを実行できます。

```
$ tpaexec configure cluster -a M1 --postgresql 15 --failover-manager patroni --
→platform docker
$ tpaexec deploy cluster
```

Docker のインストール

install Docker および `get started` のヘルプが必要な場合は、[Docker documentation](#) を参照してください。

MacOS X では、次のことができます `install "Docker Desktop forMac"`

そして、アプリケーションメニューから Docker を起動します。

TPA capabilities and supported software

- *TPA Python 環境のセットアップ*
- *Distribution support*

サポートされているソフトウェア

TPA は、次のメジャーコンポーネントをインストールおよび構成できます。

- Postgres
- EPAS EDB Postgres Advanced Server
- 5、4、3.7 ポンド
- pglogical 3、2 オープンソース
- pgd-cli および pgd-proxy
- HARP 2
- repmgr
- Barman
- pgbouncer
- haproxy PGD 3.7 でのみサポートされています
- フェールオーバーマネージャー EFM
- Postgres Enterprise ManagerPEM

7.5.2 Administration

Reconciling changes made outside of TPA

TPA 構成を変更して実行されない TPA 作成クラスターへの変更は、`config.yml` に保存されません。これは、クラスターに TPA 構成で再作成できない変更が含まれることを意味します。

このページでは、TPA で構成が管理される方法と、構成を変更するための推奨される方法を示します。次に、作成する戦略を確認し、クラスターに手動変更を加えた結果を調整します。

手動で構成を変更する必要があるのはなぜですか？

TPA の外部で構成を変更する必要がある最も一般的なシナリオは、実行している操作が TPA でサポートされていない場合です。このような操作で最も一般的な 2 つは、ノードの削除などの破壊的な変更と、Postgres のメジャーバージョンのアップグレードです。

破壊的な変更

一般に、TPA は、クラスターの以前に展開された要素を削除しません。これらは `config.yml` から削除される場合でも。厳密な宣言型システムは、宣言と一致するようにデプロイされたアーティファクトを常に変更する必要があるため、これは人々を驚かせる場合があります。ただし、実稼働データベースに破壊的な変更を行うと、重大な結果を招く可能性があるため、サポートしないことを選択しました。

メジャーバージョンの Postgres のアップグレード

TPA は、Postgres のメジャーバージョンのアップグレードを実行するための自動メカニズムをまだ提供していません。したがって、既存のクラスターでインプレースアップグレードを実行する必要がある場合、これは `pg_upgrade` や `pgd node upgrade` などの他のツールを使用して実行する必要があります。

変更がリコンサイルされていない場合、何が起こりますか？

未リコンサイルの変更に関する一般的な問題は、既存の `config.yml` を使用して新しいクラスターを展開する場合、または問題を再現するために `config.yml` を EDB サポートに提供する場合、元のクラスターと一致しないことです。さらに、将来的に TPA を使用してそのクラスターを管理する場合、操作上の問題が発生する可能性があります。

未調整の変更の運用への影響は、変更の性質によって異なります。特に変更が破壊的かどうか、および変更がエラーを発生させるか `config.yml` のデータを無効にすることにより TPA の実行をブロックするかどうか。

非破壊的、非ブロッキングな変更

多くの場合、追加的な変更はすぐに運用上の問題なく対応されます。ユーザーを手動で追加することを検討します。新しいユーザーは存在し続けますが、TPA に関する問題はまったく発生しません。TPA を介してユーザーを管理する場合、`config.yml` で宣言できますが、手動追加ユーザーの存在は操作上の問題を発生させません。

一部の手動追加は、より微妙な効果を持つ場合があります。手動で追加された拡張機能の例を考えます。TPA は破壊的な変更を行わないため、次回 `tpaexec deploy` が実行されたときに拡張機能は削除されません。ただし、新しい拡張機能に対応するように Postgres 構成を変更した場合、TPA でサポートされているメカニズム以下を参照していない場合、これらは上書きされる可能性があります。

さらに、TPA は手動変更を反映するように `config.yml` ファイルを変更しようとせず、新しい拡張機能は `tpaexec upgrade` から省略されるため、クラスターに互換性のないソフトウェアバージョンが存在する可能性があります。

破壊的なノンブロッキングの変更

簡単に検出され、TPA のオペレーションをブロックしない破壊的な変更は、次回 `tpaexec deploy` が実行されたときに単に元に戻されます。拡張機能を手動で削除することを検討します。TPA の観点から、この状況は、ユーザーが `config.yml` ファイルに拡張機能を追加し、`deploy` を実行することと区別できません。そのため、TPA は、クラスターと `config.yml` が調整されるように拡張機能を追加しますが、ユーザーが意図したものと逆の方法です。

同様に、構成パラメーターに手動で加えられた変更は、次の場合を除き、元に戻されます。

1. 手動編集用に予約された `conf.d/9999-override.conf` ファイルで作成されます。
2. `ALTER SYSTEM SQL` を使用して作成されました。または
3. `postgres_conf_settings` を追加して、`postgres_conf_settings` を作成しました。

オプション 3 が自己文書でポータブルであるという事実を除き、方法 1 または 2 によって行われた変更を調整するための差し迫った運用上の理由はありません。

破壊的な変更をブロックする

`config.yml` 間のより根本的な不一致を作成する変更は、TPA がオペレーションの実行をブロックする可能性があります。たとえば、ベアメタルクラスター内のノードを物理的に削除すると、TPA によるそのノードへの接続は失敗します。つまり、ほとんどの TPA 操作はエラーで終了し、この違いを調整するまで TPA でクラスターを管理できません。

構成変更を調整する方法

通常、リコンシリエーションプロセスには、クラスターの現在の状態を記述するように `config.yml` を変更し、`tpaexec deploy` を実行することが含まれます。

例 PGD ノードの分割

ベアアーキテクチャと次のような `configure` コマンドを使用して、最小限の PGD クラスターを展開します。

```
tpaexec configure mycluster \
- a PGD-Always-ON \
- -platform bare \
- -edbpge 15 \
- -location-names a \
- -pgd-proxy-routing local
```

次の SQL を使用してノードを分割します。これは、任意のノードから実行できます。

```
select * from bdr.part_node( 'node-2' );
```

`deploy` を再実行します。エラーは発生していませんが、ノードはまだ分離されたままであることを注意してください。これは、任意のノードでコマンド `pgd show-nodes` を使用して確認できます。これは、TPA が PGD にノードが分離されたことを伝えるメタデータを上書きしないためです。

注釈

TPA、そして実際に PGD 自分自身には、'parted' 状態でノードを初期化するメカニズムがないため、`config.yml` をこのクラスター状態と調整することはできません。原則として、TPA を使用してこの分割されたクラスターを継続できますが、これはお勧めできません。ほとんどの場合、ノードの完全な削除と `config.yml` の調整を続行することができます。

例 PGD ノードを完全に削除する

前の例では、PGD クラスターからノードを分離しましたが、ノード自分自身は無傷のままで、実行可能だが調整不能な状態で TPA によって管理されています。

ノードを完全に停止するには、`node-2` に対応するサーバーの電源をオフにするだけで安全です。この段階で `deploy` を実行しようとすると、サーバーに到達できず早期に失敗します。

`config.yml` のこの変更を調整するには、`node-2` に対応する `instances` の下のエントリーを削除するだけです。これは次のようになります。

```
- Name: node-2
  public_ip: 44.201.93.236
  private_ip: 172.31.71.186
  location: a
  node: 2
  role:
    - bdr
    - pgd-proxy
  vars:
    bdr_child_group: a_subgroup
    bdr_node_options:
      route_priority: 100
```

これで、TPA を使用してこのノードを通常のように管理できるようになりました。元のクラスターには、状態が **PARTED** であるノードとして **node-2** を参照するメタデータがまだあります。これは、クラスター機能に影響を与えないため、デフォルトでは削除されません。

注釈

`config.yml` から削除した後、元の **node-2** をクラスターに参加させたい場合は、`config.yml` の削除された行を復元し、Postgres を停止し、そのノードの PGDATA ディレクトリを削除して、`tpaexec deploy` を繰り返すことにより実行できます。上記のように、TPA は、対応するエントリが `config.yml` から削除されても、既存のデータベースを削除しないため、このアクションを手動で実行する必要があります。

例 スーパーユーザーパスワードの変更

TPA は、スーパーユーザーのパスワードを自動的に生成します。これは、`tpaexec show-password <cluster> <superuser-name>` を使用して表示できます。パスワードを手動で変更する場合たとえば、`psql` の `/password` コマンドを使用すると、次に `tpaexec deploy` が実行された後、パスワードが TPA によって設定されたものに戻っていることがわかります。TPA を介して変更を行い、`tpaexec deploy` の実行全体で持続させるには、`tpaexec store-password <cluster> <superuser-name>` コマンドを使用してパスワードを指定し、`tpaexec deploy` を実行する必要があります。これは、TPA を通じて作成された他のユーザーにも適用されます。

例 拡張機能の追加または削除

単純なシングルノードクラスターは、次の `config.yml` を使用して展開できます。

```
- --
architecture: M1
cluster_name: singlenode

cluster_vars:
  postgres_flavour: postgresql
  postgres_version: 15
  preferred_python_version: python3
```

(次のページに続く)

```
instance_defaults:
  image: tpa/debian:11
  platform: docker
  vars:
    ansible_user: root

instances:

- Name: nodeone
  node: 1
  role:
    - primary
```

ノードに接続して `apt install postgresql-15-pgvector` を実行して、次の SQL コマンド `CREATE EXTENSION vector;` を実行することにより、pgvector 拡張機能を手動で追加できます。これは、`config.yml` が以前のようにクラスターを完全に説明しないという事実を超えて、操作上の問題は発生しません。ただし、次のクラスター変数を追加することにより、`config.yml` を調整するか、実際に単に TPA を使用して拡張機能を追加することをお勧めします。

```
cluster_vars:
  ...
  extra_postgres_packages:
    common:
      - postgresql-15-pgvector
  extra_postgres_extensions:
    - vector
```

この構成を追加した後、SQL コマンド `DROP EXTENSION vector;`、次に `apt remove postgresql-15-pgvector` を実行して、拡張機能を手動で削除できます。ただし、`config.yml` を調整せずに `tpaexec deploy` を再度実行すると、拡張機能は再インストールされます。`config.yml` を調整するには、前に追加した行を削除するだけです。

注釈

前述のように、TPA は破壊的な変更を受け入れません。したがって、`config.yml` から行を削除するだけでは拡張機能は削除されません。この操作を手動で実行し、変更を調整する必要があります。

7.5.3 Services

EDB Postgres Distributed configuration

TPA は、EDB Postgres DistributedPGD、旧名 BDR 双方向レプリケーションバージョン 3.7、4.x、および 5.x をインストールおよび構成できます。

PGD パッケージへのアクセスは、EDB のパッケージリポジトリを介してのみです。パッケージをダウンロードするには、有効な EDB サブスクリプショントークンが必要です。

このドキュメントでは、PGD 構成のいくつかの側面について触れていますが、詳細の信頼できる説明については [PGDdocumentation](#) を参照してください。

はじめに

TPA は、Postgres 自分自身とともに PGD と依存関係をすべての PGD インスタンスにインストールします。

基本的な Postgres のセットアップを完了し、Postgres を起動した後、TPA は `bdr_database` を作成し、以下で説明するさまざまな手順を介して PGD クラスターのセットアップを続行します。

インストール

TPA は、使用中の Postgres のバージョンとフレーバー Postgres、Postgres Extended、または EPAS などに応じて、正しい PGD パッケージをインストールします。

`bdr_version` を設定して、PGD のどのメジャーバージョン 3、4、5 をインストールするかを決定します。
`bdr_package_version` を設定して、インストールする正確なパッケージを決定しますたとえば、最新の 5.0.x をインストールする場合は '5.0*'。

クラスターセットアップの概要

必要なパッケージをインストールし、PGD をロードするように Postgres を構成し、サーバーを起動した後、TPA は PGD ノード、グループ、レプリケーションセット、およびその他のリソースのセットアップを続行します。

次に、TPA が実行する手順の概要を示します。

- 参加するインスタンスごとに PGD ノードを作成します `bdr.create_node()`
- `bdr_node_groups` に応じて、1 つ以上の PGD ノードグループを作成します `bdr.create_node_group()` を使用
- 必要に応じて、レプリケーションセットを作成して、どの変更をレプリケートするかを正確に制御しますノードグループタイプとメンバーシップによって異なります。たとえば、サブスクライバ専用ノードと監視ノードは特別な処理が必要な場合があります。
- 個々のインスタンスで関連するノードグループに参加する
- サブグループ RAFT またはプロキシルーティングを有効にするなど、追加構成を実行します。

このプロセスには、クエリの複雑なシーケンスの実行が含まれます。一部は各インスタンスで順番に、他はパラレルで。手順を簡単に行うために、TPA は任意の PGD プライマリインスタンスをクラスターの「`first_bdr_primary`」として指定し、これを使用しますインスタンスを使用して、これらのクエリのほとんどを実行します。インスタンスはそれ以外の点では特別ではなく、その ID は PGD 構成自分自身にとって重要ではありません。

インスタンスロール

role に `bdr` を含むすべてのインスタンスは PGD インスタンスであり、暗黙的に `postgres` サーバーインスタンスでもあります。

ロールに `readonly` が含まれる PGD インスタンスは、ロジカルスタンバイノード `pause_in_standby` 設定で PGD ノードグループに参加します。昇格の対象です。

ロールに `subscriber-only` を持つ PGD インスタンスはサブスクライバー専用ノードであり、レプリケートされた変更を受信しますが、公開しません。

ロールに `witness` が含まれる PGD インスタンスは、監視ノードです。

上記のすべての PGD インスタンスは、暗黙的に `primary` インスタンスでもあります。例外は、ロールに `replica` を持つインスタンスです。これは、上流の PGD インスタンスの物理的ストリーミングレプリカを示します。このようなインスタンスは、推奨される PGD アーキテクチャには含まれておらず、現在 TPA ではサポートされていません。

構成設定

以下で説明する設定は、クラスター内のすべての PGD インスタンスに均一に設定されるように、通常は `cluster_vars` で設定する必要があります。一部の場合 `bdr_database` など、さまざまなインスタンスに異なる値を設定できますが、他の場合、結果は未定義ですたとえば、すべてのインスタンスが完全に同じ値の `bdr_node_groups` を持つ必要があります。

`cluster_vars` の下でクラスター全体の均一な値を設定することにより、PGD 構成を定義することを強くお勧めします。

`bdr_database`

`bdr_database` デフォルト `bdrdb` は PGD で初期化されます。

`bdr_client_dsn_attributes`

`bdr_client_dsn_attributes` には、追加の `parameter keywords supported by libpq` を含めることができます。

`host`、`port`、`dbname`、および `user` は含めないでください。これらは既に接続文字列に含まれています。

注釈

`pgd-proxy` および `pgd-cli` がインストールされている場合 `pgd-proxy` および `pgd-cli` は Go で書かれているため、Postgres への接続に Go ドライバーを使用します。これらのドライバは、`libpqC` ライブラリが提供する DSN 属性の完全なセットをサポートしていません。

`pgd-proxy` および/または `pgd-cli` がインストールされ、`bdr_client_dsn_attributes` に Go ドライバーでサポートされていないパラメータータイムアウトなどが含まれている場合、2つの新しい変数をクラスター構成に含める必要があります。

- `pgd_proxy_dsn_attributes` `pgd-proxy-conf` で接続文字列を作成するために使用されます
- `pgd_cli_dsn_attributes` `pgd-cli-conf` で接続文字列を作成するために使用されます

これらの2つの文字列には、Go ドライバーと互換性のあるパラメーターキーワードのみが含まれている必要があります。

`bdr_client_dsn_attributes` がサポートされていないパラメーターを含まない場合、これは無視でき、`bdr_client_dsn_attributes` は `pgd-proxy-conf` および `pgd-cli-conf` の接続文字列に含まれます。

`bdr_node_group` の設定デフォルトクラスター名に基づいて、インスタンスがどの PGD クラスターに参加するかを指定します。外部コンポーネントたとえば、`pgd-proxy` または `harp-proxy` などの特定のクラスターを識別するためにも使用されます。

bdr_node_groups

これは、グループ参加ステージ前に作成する必要がある PGD ノードグループのリストです。クラスターに追加のサブグループが必要な場合。

通常、`tpaexec configure` は、選択したアーキテクチャに基づいて適切な値を生成します。

```
cluster_vars:
  bdr_node_groups:
    - name: topgroup
    - name: abc_subgroup
      node_group_type: data
      parent_group_name: topgroup
      options:
        location: abc
    ...
```

最初のエントリーは、クラスターの `bdr_node_group` 用である必要があります。

リストの後続の各エントリーは `parent_group_name` を指定する必要がある、`node_group_type` オptional を指定する場合があります。

各エントリーは、グループオプションのオプションのキー/値のマッピングを持つ場合があります。使用可能なオプションは PGD バージョンによって異なります。

bdr_child_group

インスタンスに `bdr_node_groups` で記載されているグループの名前に `bdr_child_group` が設定されている場合、`bdr_node_group` の代わりにそのグループに参加します。

bdr_commit_scopes

これは `commit scopes` のオプションリストです

PGD データベース PGD 4.1 以降で利用可能に存在する必要があります。

```
cluster_vars:
  bdr_commit_scopes:
    - name: somescope
      origin: somegroup
```

(次のページに続く)

(前のページからの続き)

```

    rule: ALL (somegroup) ON received ...`
- name: otherscope
  origin: othergroup
  rule: ...
...

```

各エントリは、コミットスコープの **name**、**origin** グループの名前、およびコミットスコープ **rule** を指定する必要があります。グループは **bdr_node_groups** のエントリに対応する必要があります。

bdr_commit_scopes を明示的に設定すると、TPA は必要に応じてコミットスコープを作成、変更、またはドロップして、データベースが構成と一致することを確認します。設定しない場合、既存のコミットスコープはそのままになります。

その他のノート

フック

TPA は、PGD クラスターセットアッププロセス中に **bdr-node-pre-creation**、**bdr-post-group-creation**、および **bdr-pre-group-join** *TPA hooks* を呼び出します。

データベースの照合順序

TPA は、クラスター内のすべてのインスタンスの PGD データベースが同じ照合順序 **LC_COLLATE** 設定があることを確認します。同じ PGD クラスター内のデータベースに異なる照合順序があると、データ損失のリスクが発生します。

PGD の古いバージョン

TPA は、BDR v1 Postgres 9.4 のパッチ適用バージョンを使用、v2 Postgres 9.6、または v3.7 より前の PGD バージョンの展開を積極的にサポートまたはテストしなくなりました。

Barman

インスタンスに **config.yml** で **barman** ロールが付与されている場合、TPA は **Barman** サーバーとして構成して、**backup** 設定で名前を付けている他のインスタンスのバックアップを取得します。

```

instances:

- Name: one
  backup: two
  ...

- Name: two
  role:
    - barman
  ...

```

複数の postgres インスタンスは、**backup** と同じ名前の Barman サーバーを持つことができます。同様に、1つの postgres インスタンスは、その **backup** として名前付けた Barman サーバーのリストを持つことができ、バックアップはすべての名前付けサーバーに取得されます。

デフォルトの Barman 構成は、pg_receivewal を使用して PostgreSQL に接続し、WAL の継続的なバックアップを取得し、週に 2 回 ssh での rsync を使用してインスタンスの完全なバックアップを取得します。完全なバックアップと WAL は、過去 4 週間の任意のポイントにリカバリーできるように十分な期間保持されます。

Barman パッケージバージョン

デフォルトでは、TPA は Barman の最新バージョンをインストールします。

config.yml ファイルの cluster_vars セクションの下に **barman_package_version:** xxx を含めることにより、インストールされる Barman パッケージのバージョンを指定できます。

```
cluster_vars:
  ...
  barman_package_version: 1.56.2-1
  ...
```

apt または yum が受け入れるバージョン指定子を使用できます。

バージョンが一致しない場合は、* ワイルドカードを追加してみてください。これは、パッケージバージョンに 2:... のようなエポック修飾子がある場合に必要になることがよくあります。

Barman 構成

Barman サーバーの Barman ホームディレクトリは、クラスター変数 **barman_home** を使用して設定できます。デフォルト値は **/var/lib/barman** です。

各 Barman サーバーでは、グローバル構成ファイルが **/etc/barman.conf** として作成されます。このファイルには、多くの Barman 構成変数のデフォルト値が含まれています。バックアップされる Postgres サーバーごとに、追加の Barman 構成ファイルが作成されます。たとえば、サーバー **one** をバックアップするには、ファイルは **/etc/barman.d/one.conf** で、バックアップは Barman ホームディレクトリのサブディレクトリ **one** に保存されます。構成ファイルとディレクトリ名は、バックアップされたインスタンスの **backup_name** 設定から取得されます。この設定のデフォルトはインスタンス名です。

次の変数はバックアップインスタンスで設定でき、接頭辞 **barman_** が削除された状態で Barman の構成に渡されます。

variable	default
barman_archiver	false
barman_log_file	/var/log/barman.log
barman_backup_method	rsync
barman_compression	pigz
barman_reuse_backup	link
barman_parallel_jobs	1

次のページに続く

表 1 – 前のページからの続き

variable	default
barman_backup_options	concurrent_backup
barman_immediate_checkpoint	false
barman_network_compression	false
barman_basebackup_retry_times	3
barman_basebackup_retry_sleep	30
barman_minimum_redundancy	3
barman_retention_policy	RECOVERY WINDOW OF 4 WEEKS
barman_last_backup_maximum_age	1 WEEK
barman_pre_archive_retry_script	
barman_post_backup_retry_script	
barman_post_backup_script	
barman_streaming_wals_directory	

バックアップ スケジュール

TPA は、`/etc/cron.d/barman` に cron ジョブをインストールします。これは 1 分ごとに実行され、`barman cron` を呼び出してメンテナンスタスクを実行します。

バックアップされるインスタンスごとに、そのインスタンスのバックアップを取得する別の cron ジョブを `/etc/cron.d/<backup_name>` にインストールします。このジョブは、インスタンスの `barman_backup_interval` 変数で決定されたとおり実行され、デフォルトでは、毎週水曜日と土曜日の 04:00 にバックアップを取得します。

SSH キー

TPA は、`postgres` および `barman` ユーザーの ssh キーペアを生成し、それぞれの `~/.ssh` ディレクトリにインストールし、お互いの `authorized_keys` ファイルに追加します。`postgres` ユーザーは、WAL セグメント構成されている場合をアーカイブするために `barman` サーバーに ssh 接続できる必要があります、`barman` ユーザーは `Postgres` インスタンスに ssh 接続して、バックアップを取得または復元できる必要があります。

barman および barman_role Postgres ユーザー

TPA は、2 人の `Postgres` ユーザー `barman` および `barman_role` を作成します。

TPA バージョン <23.35 は、`barman` `Postgres` ユーザーを `superuser` として作成しました。

23.35 から `barman` ユーザーは `NOSUPERUSER` で作成されるため、既存のクラスターに再展開すると、`barman` `Postgres` ユーザーから `superuser` 属性が削除されます。代わりに、`barman_role` には必要な特権セットが付与され、`barman` ユーザーには `barman_role` メンバーシップが付与されます。

これにより、*Barman Manual* で提供される特権セットを使用して、`barman` ユーザーに `superuser` 属性を付与することを回避します。

共有 Barman サーバー

i 注釈

23.35 より前の TPA バージョンを使用して作成されたクラスターで共有 Barman 機能を使用するには、以下を行う必要があります。a) 共有 Barman インスタンスの作成をサポートする TPA のバージョンにアップグレードします。b) TPA をアップグレードした後、\$first-cluster で deploy を実行して、TPA が後続のクラスターが共有 Barman ノードに対してスムーズに実行されるように必要な構成変更を加えられるように。

一部の展開では、複数のクラスターで単一の Barman サーバーを共有する必要がある場合があります。tpaexec 内の共有 Barman サーバーの展開は、既存の Barman サーバーを使用する予定がある特定のクラスター構成の Barman サーバーインスタンスの下で **vars:** を介して設定できる **barman_shared** 設定を介してサポートされています。**barman_shared** はブール変数であるため、可能な値は **true** および **false** デフォルトです。共有シナリオで Barman 構成を変更する場合、複数のクラスターにわたる構成が同期したままであることを確認して、1 つのクラスターが特定の構成を追加し、2 番目のクラスターがそれをオーバーライドするシナリオを回避する必要があります。

複数のクラスターで共有 Barman サーバーを使用するための一般的なワークフローを以下に説明します。

1. **barman** ロールを持つインスタンスを使用して TPA クラスターを作成しますこの例では、'first-cluster' と呼びます。
2. 2 番目のクラスター **second-cluster** 例では、この特定の Barman インスタンスを共有 Barman サーバーインスタンスとして \$clusters/first-cluster から参照し、プラットフォームとして **bare** を使用するため、実行時に新しい Barman インスタンスを作成しようとしません条項。また、このクラスターがアクセスするために使用できる Barman インスタンスの IP アドレスを指定します。

```
- Name: myBarman
  node: 5
  role:
    - barman
  platform: bare
  ip_address: x.x.x.x
  vars:
    barman_shared: true
```

3. 2 番目のクラスターがプロビジョニングされたら、展開を実行する前に、ssh を介して Barman サーバーインスタンスにアクセスできることを確認します。ssh-copy-id を介して 2 番目のクラスターの公開キーを Barman サーバーインスタンスにコピーすることにより、このアクセスを許可でき、ssh を実行して、パスワードを指定せずにログインできることを確認します。

```
# add first-clusters key to the ssh-agent
$ cd $clusters/first-cluster
$ ssh-add id_first-clutser
$ cd $clusters/second-cluster
$ ssh-keyscan -t rsa,ecdsa -4 $barman-server-ip >> tpa_known_hosts
```

(次のページに続く)

(前のページからの続き)

```
$ ssh-copy-id -i id_second-cluster.pub -o UserKnownHostsFile=tpa_known_hosts $user@
→$barman-server-ip
$ ssh -F ssh_config $barman-server
```

4. Barman ユーザーのキーを最初のクラスターから 2 番目のクラスターにコピーします

```
$ mkdir $clusters/second-cluster/keys
$ cp $clusters/first-cluster/keys/id_barman* clusters/second-cluster/keys
```

5. `tpaexec deploy $clusters/second-cluster` を実行します

注釈

混合プラットフォームのクラスター共有 Barman インスタンスを `platform: bare` として宣言することにより、クラスターを混合プラットフォームクラスターに変更した可能性があります。これには、この変更に対応するように `config.yml` の他の部分を調整する必要がある場合があります。具体的には、`instance_defaults`` セクションには、クラスター内のすべてのインスタンスに適用される設定のみを含める必要があります。たとえば、``instance_defaults`` に、`platform: aws` でのみ有効な `type` などの設定が含まれている場合、その設定を `instance_defaults` から AWS プラットフォームを使用するインスタンスにのみ移動する必要があります。

共有 Barman サーバーの特別な考慮事項

Barman サーバーインスタンスを共有するクラスターをセットアップするときは、注意する必要があります。このようなセットアップを試行する前に、考慮する必要がある重要な側面が多数あります。

1. Barman サーバーを共有するクラスター内の 2 つのインスタンスが同じ名前を使用していないことを確認します。`tpaexec configure` の `--cluster-prefixed-hostnames` オプションは、この点で役立つ場合があります。
2. Barman の構成と設定は、共通の Barman サーバーを使用するすべてのクラスターで同期したままにして、1 つのクラスターが特定の構成をセットアップし、構成が欠落しているか別の値を使用しているために他のクラスターがセットアップしないシナリオを回避する必要があります。
3. 異なるクラスターにわたってバックアップされるインスタンスの Postgres のバージョンは、同じである必要があります。
4. 共通の Barman サーバーを使用する異なるクラスターは、デフォルトをオーバーライドしようとするときに Barman パッケージの異なるバージョンを指定できません。

共有 Barman サーバーのサポートの向上を続けているため、これらの一部は将来のリリースで対応される可能性があります。

!!!警告

共通の Barman ノードを共有するクラスターのプロビジョニングを解除するときは、特に注意してください。特に、Barman を展開した最初のクラスターが非ベアプラットフォームを使用する場合。Barman は最初のクラスターによって既にプロビジョニング解除され、もう存在しないため、最初に Barman をプロビ

ジョニングおよび展開した最初のクラスターをプロビジョニング解除すると、Barman ノードを共有する他のクラスターが事実上一貫性のない状態のままになります。!!!

Configuring EFM

failover_manager が efm に設定されている場合、TPA は EFM をインストールおよび構成します。

EFM は EDB のパッケージリポジトリを介してのみ利用でき、有効なサブスクリプションが必要であることに注意してください。

EFM 構成

TPA は、適切なインスタンス固有の設定で `efm.nodes` および `efm.properties` を生成し、残りの設定はそれぞれのデフォルト値に設定されます。TPA はまた、`efm.notification.sh` スクリプトも配置します。これは、基本的にデフォルトでは何も含まず、ユーザーが必要に応じて入力するかに任されています。TPA は、`auto.allow.hosts` および `stable.nodes.file` のデフォルト設定をオーバーライドして、クラスターへのエージェントの追加を簡素化します。

[EFM documentation](#) を参照してください

EFM 構成の詳細については、こちらをご覧ください。

efm_user_password_encryption

scram-sha-256 または md5 のいずれかである必要があります

`efm_user_password_encryption` を設定して、`pg_hba.conf` の `efm` Postgres ユーザーの `auth-method` の `auth-method` と、暗号化されたパスワードを生成するときに使用されるアルゴリズムを制御します。

```
efm_user_password_encryption: scram-sha-256 # or can be set to `md5`
```

efm_conf_settings

`efm_conf_settings` を使用して、特定のパラメーターを設定できます。これらは、Ansible 辞書のエントリーとして `key: value` フォームで書き込む必要があります

documentation on the `efm.properties` file <Configuring EFM> を参照してください

構成できる設定の詳細については、[こちら](#)を参照してください。

```
cluster_vars:
  efm_conf_settings:
    notification.level: WARNING
    ping.server.ip: <well known address in network>
```

`efm_conf_settings` の下の値を変更すると、TPA は常に EFM を再起動して変更をアクティブにします。

EFM 証人

TPA は、role に `efm-witness` が含まれるインスタンスに監視として EFM をインストールおよび構成します。

Repmgr

EFM はフェールオーバーマネージャーとして機能するため、TPA は引き続き `repmgr` をインストールして、postgres バージョン 11 以下で `postgresql` レプリカをセットアップします。`repmgrd` つまり、この場合、`repmgr` のデーモンは無効なままで、`repmgr` の唯一のジョブは、レプリケーションセットアップ機能を提供することです。

postgres バージョン 12 以降の場合、EFM を使用するクラスターは、`pg_basebackup` を使用してスタンバイノードを作成し、どの形式でも `repmgr` を使用しません。

ノードのプロモーション

TPA は、ノードのロールとレプリケーショントポロジに基づいて、フェールオーバー中にノードが EFM によるプロモーションの対象かどうかを判断します。EFM 構成を生成するときに次のルールが適用されます。

- **Witness ノード** `witness` ロールはプロモーションできません。
- ```efm-not-promotable``` ロールを持つノードは昇格の資格がありません。これは、DR または報告ノードなどの特定のスタンバイがプライマリに昇格しないようにするために使用できます。
- **カスケードスタンバイ**プライマリから直接複製していないノードもプロモーション可能ではありません。
- 他のすべてのノードは、デフォルトでプロモーション可能であると考えられます。

スタンバイが昇格することを明示的に防ぐには、クラスター構成のノードの `roles` リストに `efm-not-promotable` を追加します。これにより、EFM は、フェールオーバーイベント中にこのノードを昇格させようとしないことが保証されます。

Configuring haproxy

TPA は、role に `haproxy` が含まれるインスタンスに `haproxy` をインストールおよび構成します。

デフォルトでは、`haproxy` は `127.0.0.1:5432` で、同じインスタンスで実行されている [Repmgr リダイレクト `pgbouncer`](#) によって転送された要求をリッスンします。要求を転送する `haproxy_backend_servers` のリストを指定する必要があります。

TPA は、デフォルトで `haproxy` の最新バージョンをインストールします。`haproxy_package_version: 1.9.15*` たとえば、を設定することにより、特定のバージョンをインストールできます。

注 [`tpaexec-configure`](#) の `package_version` でのワイルドカードの使用の制限を参照してください。

Haproxy パッケージは、アーキテクチャのタイプに応じて選択されます。EDB 管理対象 `haproxy` パッケージを使用できますが、サブスクリプションが必要です。必要に応じて、PGDG エクストラリポジトリからのパッケージをインストールできます。

`haproxy` インスタンスで次の変数を設定できます。

Variable	Default value	Description
<i>haproxy_bind_address</i>	127.0.0.1*	<code>failover_manager</code> がパトローニの場合、haproxy が 0.0.0.0 をバインドするアドレス。 * ユーザーはこの値を、クラスターネットワーキングに適したより制限的なものに変更する必要があります**
<i>haproxy_port</i>	5432 (5444 for EPAS)	TCP ポート HA プロキシがリッスンする必要がある
<i>haproxy_read_only_port</i>	5433 (5445 for EPAS)	読み取り専用ロードバランサーの TCP ポート
<i>haproxy_backend_servers</i>	None	Postgres インスタンス名のリスト
<i>haproxy_maxconn</i>	<code>max_connections</code> × 0.9	バックエンドサーバーごとに許可される接続の最大数。デフォルトは、バックエンドの <code>max_connections</code> 設定から取得されます
<i>haproxy_peer_enabled</i>	True*	既知のハプロキシホストを <code>peer</code> リストとして追加します。 *False <code>failover_manager</code> が harp または patroni の場合。

読み取り専用ロードバランサー

データベースへの読み取り専用アクセスを追加ポートでリッスンするように Haproxy を構成できます。現時点では、これは Patroni フェールオーバーマネージャーでのみサポートされています。バックエンドヘルスチェックは、どの postgres インスタンスが現在レプリカとして機能し、ラウンドロビンロードバランシングアルゴリズムを使用してトラフィックを送信するかを判断します。

読み取り専用ロードバランサーはデフォルトで無効になっていますが、`cluster_vars` 変数 `haproxy_read_only_load_balancer_enabled` を使用して有効にできます。

サーバーオプション

TPA は、`default-server` 行とバックエンドサーバーごとに 1 行があるバックエンドで `/etc/haproxy/haproxy.cfg` を生成します。最初のサーバーを除くすべては「バックアップ」サーバーとしてマークされます。

`haproxy_default_server_extra_options` を haproxy インスタンスのオプションのリストに設定して、`default-server` 行にオプションを追加します。そして、`haproxy_server_options` をバックエンドサーバーのオプションのリストに設定して、各バックエンドの個々のサーバー行にオプションデフォルトをオーバーライドします。

`ssl-dh-param-file` に使用される RSA キーサイズは、`cluster_vars` セクションに変数 `ha_proxy_dhparams_key_size` を追加することにより変更できます。

例

```
cluster_vars:
  ha_proxy_dhparams_key_size: 4096
instances:

- Name: one
  vars:
    haproxy_server_options:
      - maxconn 33

- Name: two
...

- Name: proxy
  role:
    - haproxy
  vars:
    haproxy_backend_servers:
      - one
      - two
    haproxy_default_server_extra_options:
      - on-error mark-down
      - on-marked-down shutdown-sessions
```

Configuring HARP

failover_manager が harp に設定されている場合、TPA は HARP をインストールおよび構成します。これは、BDR-Always-ON クラスターのデフォルトです。

HARP のインストール

harp-manager および harp-proxy パッケージを提供する必要があります。これらのパッケージへのアクセスを取得するには、EDB に連絡してください。

HARP 構成の変数

HARP documentation を参照してください

HARP 構成の詳細については、

Variable	Default value	Description
cluster_name	``	クラスターの名前。
harp_consensus_protocol	``	使用するコンセンサスレイヤー <i>etcd</i> または <i>bdr</i>

次のページに続く

表 3 – 前のページからの続き

Variable	Default value	Description
<i>harp_location</i>	<i>location</i>	このインスタンスの場所 デフォルトは <code>`location`</code> パラメーター
<i>harp_ready_status_duration</i>	<i>10</i>	ノードの準備ができていない時間秒更新されない場合、ステータスは持続します。
<i>harp_leader_lease_duration</i>	<i>6</i>	リードマスターの秒単位の時間リースは、更新されない場合は持続します。
<i>harp_lease_refresh_interval</i>	<i>2000</i>	ミリ秒単位の時間リードマスターリースの更新。
<i>harp_dcs_reconnect_interval</i>	<i>1000</i>	ミリ秒単位で測定される間隔、切断されたノードが DCS への再接続を試行する間隔。
<i>harp_dcs_priority</i>	<i>500</i>	2つのノードが等しい量の遅延とリードマスターリースを取得するその他の修飾基準がある場合、これは追加のランキング値として機能し、1つのノードを別のノードより優先します。
<i>harp_stop_database_when_fenced</i>	<i>false</i>	考えられるすべてのルーティングからノードを単に削除するのではなく、ノードがフェンスされたときにデータベースを停止します。
<i>harp_fenced_node_on_dcs_failure</i>	<i>false</i>	HARP が DCS に到達できない場合は、ノードをフェンスします。
<i>harp_maximum_lag</i>	<i>1048576</i>	リードマスターロックを取られる前に、以前のリードマスターとこのノードの LSN を記録した間の最大許容分散バイト単位。
<i>harp_maximum_camo_lag</i>	<i>1048576</i>	このノードとその CAMO パートナー間で最後に受信した LSN と適用された LSN 間の最大許容分散バイト単位。
<i>harp_camo_enforcement</i>	<i>lag_only</i>	CAMO キュー状態を厳密に適用するかどうか。
<i>harp_use_unix_sock</i>	<i>false</i>	マネージャーデータベースアクセスには UNIX ドメインソケットを使用します。
<i>harp_request_timeout</i>	<i>250</i>	ミリ秒単位の時間 DCS へのクエリーが成功するまでの時間。
<i>harp_watch_poll_interval</i>	<i>500</i>	ポーリング DCS 間のスリープするミリ秒。 <code>`harp_consensus_protocol`</code> が <code>`bdr`</code> の場合にのみ適用されます。
<i>harp_proxy_timeout</i>	<i>1</i>	Lead Master へのビルトインプロキシ接続タイムアウト秒単位。
<i>harp_proxy_keepalive</i>	<i>5</i>	ビルトインプロキシがキープアライブ ping を送信する前に、リードマスターへのアイドル接続で待機する時間。
<i>harp_proxy_max_client_conn</i>	<i>75</i>	harp-プロキシによって受け入れられるクライアントの最大数 <i>max_client_conn</i>

次のページに続く

表 3 – 前のページからの続き

Variable	Default value	Description
<i>harp_ssl_password_command</i>	None	難読化された ssl パスワードを stdin で受信し、処理された ssl パスワードを stdout 経由で提供するカスタムコマンド。
<i>harp_db_request_timeout</i>	10s	dcs -> request_timeout に似ていますが、データベース自分自身への接続用です。
<i>harp_local_etcd_only</i>	None	すべての etcd ノードではなくローカル etcd ノードのみを含むように harp manager エンドポイントリストを制限する

harp-config hook を使用できます

HARP 構成ファイルがインストールされた後にタスクを実行するには追加の構成ファイルをインストールします。

コンセンサスレイヤー

`tpaexec configure` への `--harp-consensus-protocol` 引数は、BDR-Always-ON アーキテクチャでは必須です。

etcd

`--harp-consensus-protocol etcd` オプションが `tpaexec configure` に指定されている場合、TPA は `config.yml` で `harp_consensus_protocol` を `etcd` に設定し、選択したレイアウトに応じてインスタンスの適切なサブセットに `etcd` ロールを付与します。

HARP v2 には、`etcd v3.5.0` 以降が必要です。これは、EDB が提供する `products/harp/release` パッケージリポジトリで入手できます。

`etcd` の次のパラメーターを構成できます。

Variable	Default value	Description
<code>etcd_peer_port</code>	2380	<code>etcd</code> がピア通信に使用するポート
<code>etcd_client_port</code>	2379	クライアントが <code>etcd</code> に接続するために使用するポート

bdr

`tpaexec configure` に `--harp-consensus-protocol bdr` オプションが指定されている場合、TPA は `config.yml` で `harp_consensus_protocol` から `bdr` を設定します。この場合、既存の PGD インスタンスはコンセンサスに使用され、追加の構成は必要ありません。

harp プロキシの別のユーザーの構成

harp プロキシで別の読み取り専用ユーザーを使用する場合は、`cluster_vars` の下で `harp_dcs_user: username` を設定することにより、それを指定できます。TPA は `harp_dcs_user` 設定を使用して読み取り専用ユーザーを作成し、DCS 構成で設定します。

harp manager 用の別のユーザーの設定

harp manager で別のユーザーを使用する場合は、`cluster_vars` の下で `harp_manager_user: username` を設定することにより、それを指定できます。TPAexec はその設定を使用して、新しいユーザーを作成し、`bdr_superuser` ロールを付与します。

カスタム SSL パスワードコマンド

`harp_ssl_password_command` が提供するコマンドは、HARP が使用して、接続文字列で指定された `sslpassword` の難読化を解除します。`sslpassword` が存在しない場合、`harp_ssl_password_command` は無視されます。`sslpassword` が難読化されていない場合、`harp_ssl_password_command` は必要なく、指定しないでください。

harp サービスの構成

harp サービスの次のパラメーターを構成できます。

Variable	Default value	Description
<code>harp_manager_restart_on_failure</code>	<code>false</code>	<code>`true`</code> の場合、 <code>`harp-manager`</code> サービスはオーバーライドされるため、失敗時に再起動されます。デフォルトは <code>`false`</code> で、 <code>`harp-manager`</code> パッケージによってインストールされるサービスに準拠します。

harp http(s) ヘルスプローブの構成

サービスの状態を監視するための API エンドポイントを提供する harp の http(s) サービスを有効にして構成できます。

Variable	Default value	Description
<i>harp_http_options</i>	<pre>
`enable: false`
`se- cure:      false`
`host:    <inven- tory_hostname>`
`port:    8080`
`probes:<br      />           `timeout: 10s`<br />`endpoint: "host=<proxy_name> port=<6432> db- name=<bdrdb> user=<username>" </pre>	http(s) API 設定を定義する harp config.yml の http セクションを構成します。

変数には、次のキーを含めることができます。

```

enable: false
secure: false
cert_file: "/etc/tpa/harp_proxy/harp_proxy.crt"
key_file: "/etc/tpa/harp_proxy/harp_proxy.key"
host: <inventory_hostname>
port: 8080
probes:
  timeout: 10s
endpoint: "<valid dsn>"

```

cert_file および key_file キーは、secure: true を使用し、独自の証明書とキーを使用する場合に必要です。

deploy を実行する前に、証明書とキーの両方がターゲットノードの指定された場所で利用できることを確認する必要があります。

クラスター固有の CA 証明書を使用して TPA に証明書とキーを生成する場合、cert_file および key_file の両方を空のままにします。TPA CA 証明書は「ウェルノウン」になりません。この証明書を、エンドポイントをプローブする各マシンの信頼ストアに追加する必要があります。CA 証明書は、TPA ノードのクラスターディレクトリで deploy の後の<cluster_dir>/ssl/CA.crt にあります。

使用可能な API エンドポイントの詳細については、harp ドキュメントを参照してください。

Configuring Postgres Enterprise Manager (PEM)

TPA は、--enable-pem コマンドラインオプションを使用して tpaexec configure コマンドが実行されると、PEM のインストールと構成を行います。

--enable-pem のデフォルト動作は、クラスター内のすべての postgres インスタンスの pem-agent ロールを有効にすることです。pem-agent ロールは、--enable-pg-backup-api コマンドラインオプションが--enable-pem とともに使用される場合、バーマンノードにも追加されます。

`pemserver` という名前の専用インスタンスもクラスターに追加されます。

PEM サーバーは postgres バックエンドを使用するため、`pemserver` インスタンスは、`postgres` ロールを暗黙的に使用するため、`pemserver` が PEM バックエンドとして使用するよう構成された有効な postgres クラスターを取得します。通常の postgres インスタンスで使用可能なすべての構成オプションは、PEM のバックエンド postgres インスタンスでも有効です。詳細については、以下を参照してください。

- [`pg\_hba.conf`](#)
- [`postgresql.conf`](#)

PEM は EDB のパッケージリポジトリを介してのみ利用可能であるため、有効なサブスクリプションが必要であることに注意してください。

サポートされているアーキテクチャ

PEM は、`--enable-pem` 構成コマンドラインオプションを介してすべてのアーキテクチャでサポートされています。ただし、EDB Postgres Extended で使用する場合は BDR -Always-ON アーキテクチャを除きます。オプションで、生成されたクラスター構成 `config.yml` を編集し、クラスター内の postgres インスタンスから `pem-agent` ロールを割り当てまたは削除して、そこで PEM を有効または無効にすることができます。

PEM コンポーネントパッケージバージョン

デフォルトでは、TPA は PEM エージェントと PEM サーバーの最新の利用可能なバージョンをインストールします。

`config.yml` ファイルの `cluster_vars` セクションの下に `pem_agent_package_version: xxx` および `pem_server_package_version: xxx` を含めることにより、インストールされる PEM エージェントと PEM サーバーパッケージのバージョンを指定できます。

```
cluster_vars:
  ...
  pem_agent_package_version: 9.7.0-1.el9
  pem_server_package_version: 9.7.0-1.el9
  ...
```

`apt` または `yum` が受け入れるバージョン指定子を使用できます。

バージョンが一致しない場合は、* ワイルドカードを追加してみてください。これは、パッケージバージョンに `2:...` のようなエポック修飾子がある場合に必要になることがよくあります。

PEM 構成

TPA は、適切なインスタンス固有の設定で `pem` エージェントと `pem` サーバーを構成し、残りの設定はそれぞれのデフォルト値に設定されます。構成オプションの一部は、将来のある時点でユーザー構成用に公開される可能性があります。

PEM サーバーの Web インターフェイスは、`https` で実行されるように構成され、同じに 443 ポートを使用します。PEM のウェブサーバー構成は自己署名証明書を使用します。

PEM サーバー Web インターフェイスのデフォルトのログイン資格情報は、postgres バックエンドデータベースユーザーを使用します。これは、デフォルトで `postgresql` の場合は `postgres`、EPAS クラスターの

場合は `enterprisedb` に設定されます。 `tpaexec show-password $clusterdir $user` を実行して、Web インターフェイスのログインパスワードを取得できます。

PEM エージェントの登録時に追加オプションを渡す

TPA は、 `pemworker` ユーティリティの `--register agent` コマンドを使用して、クラスター内の各 PEM エージェントを登録します。

クラスター構成に `pemagent_registration_opts` を含めることにより、追加登録オプションのリストを渡すことができます。

例

```
pemagent_registration_opts:
- --enable-smtp true
- --enable-heartbeat-connection
- --allow-batch-probes true
- -l DEBUG1
```

PEM documentation

に、登録オプションの詳細をリストします。

pem エージェントを使用するノードの便利な拡張機能

デフォルトでは、TPA は `sql_profiler`、`edb_wait_states`、および `query_advisor` 拡張機能を `pem-agent` ロールを持つインスタンスに追加します。

`pem-agent` ノードのデフォルトの拡張機能のこのリストは、`config.yml` で `pemagent_extensions` を設定することによりオーバーライドできます。

このリストが空の場合、拡張機能は自動的に含まれません。

PEM サーバー SSL 認証の外部証明書の提供

デフォルトでは、PEM サーバーは自己署名証明書ペア `server-pem.crt` および `server-pem.key` を作成し、HTTPS アクセスにそれらを使用するようにウェブサーバーを構成します。

`server-pem.key` のサイズは、変数 `pem_rsa_key_size` を `cluster_vars` セクションに追加して変更できます。

```
(...)
cluster_vars:
  pem_rsa_key_size: 4096
```

PEM データベースによって促進される CA 証明書のサイズは、変数 `pem_db_ca_certificate_key_size` を `cluster_vars` セクションに追加して変更することもできます。

```
(...)
cluster_vars:
  pem_db_ca_certificate_key_size: 4096
```

独自の証明書ペアを提供するには、`ssl/pemserver` という名前のクラスターディレクトリのルートの下にディレクトリを作成し、その中に証明書ペアを配置します。

```
cluster directory
├── ssl
│   └── pemserver
│       ├── externally-provided.crt
│       └── externally-provided.key
```

次に、`pem` サーバーインスタンスの下の変数: またはクラスター構成ファイルの `cluster_vars` の値として、各ファイル名を使用して変数 `pem_server_ssl_certificate` および `pem_server_ssl_key` を設定します。

TPA は、これらのファイルの `pem` サーバーインスタンスへのコピーを処理し、それに応じてウェブサーバーを構成します。

```
- Name: pemserver
  location: main
  node: 4
  role:
  - pem-server
  vars:
    pem_server_ssl_certificate: externally-provided.crt
    pem_server_ssl_key: externally-provided.key
```

共有 PEM サーバー

一部の展開では、組織内の複数のクラスターの監視と管理に単一の PEM サーバーを使用する場合があります。tpaexec 内の共有 pem サーバーの展開は、既存の pem サーバーを使用する予定がある特定のクラスター構成の pem サーバーインスタンスの下で `vars:` を介して設定できる `pem_shared` 変数を介してサポートされています。`pem_shared` はブール変数であるため、可能な値は `true` および `false` デフォルトです。`pemserver` インスタンスを共有として宣言する場合、`pemserver` インスタンスが実際には、最初に pem サーバーをプロビジョニングおよび展開した別のクラスター構成によって管理されていることを指定されたクラスター構成に伝えます。したがって、pem の postgres バックエンドを含む pem サーバーインスタンスに加えたい変更は、`pemserver` インスタンスが共有 pem インスタンスとして宣言されていないクラスターによって管理されます。

複数のクラスターで共有 pem サーバーを使用するための一般的なワークフローは次のとおりです。

1. `pem-server` ロールを持つ単一のインスタンスを使用して `tpaexec` クラスターを作成しますこの例では、`'pem-cluster'` と呼びます。`pem` サーバーとして実行される単一のインスタンスではなく、大規模なクラスターの一部としてプロビジョニングされるシナリオでも、同じワークフローを簡単に使用できますが、例として使用の方が簡単であるため、おそらくシングルノードクラスターを使用しますメンテナンスも簡単です。
2. 他のクラスターたとえば `pg-cluster` では、共有 pem サーバーインスタンスとして `$clusters/pem-cluster` からこの特定の pem サーバーを参照し、プラットフォームとして `bare` を使用するため、新しい `pemserver` インスタンスを作成しようとはしていません。また、このクラスターが `pemserver` インスタンスにアクセスするために使用できる `pemserver` の IP アドレスを指定します。

```
- Name: pemserver
  node: 5
  role:
  - pem-server
  platform: bare
  public_ip: 13.213.53.205
  private_ip: 10.33.15.102
  vars:
    pem_shared: true
```

3. postgres クラスターで deploy を実行する前に、pg-cluster が ssh を介して pem サーバーインスタンスにアクセスできることを確認してください。ssh-copy-id を介して pg-cluster の公開キーを pem サーバーインスタンスにコピーすることにより、このアクセスを許可でき、ssh を実行して、パスワードを指定せずにログインできることを確認します。

```
# add pem-clusters key to the ssh-agent (handy for `aws` platform)
$ cd $clusters/pem-cluster
$ ssh-add id_pem-clutser
$ cd $clusters/pg-cluster
$ ssh-keyscan -4 $pem-server-ip >> known_hosts
$ ssh-copy-id -i id_pg-cluster.pub -o UserKnownHostsFile=tpa_known_hosts $user@$pem-
→server-ip
$ ssh -F ssh_config pemserver
```

4. pem サーバーノードの postgresql 構成を更新して、新しい pg-cluster からの接続を許可します。pem-cluster の config.yml の vars: の下の pem_postgres_extra_hba_settings に新しいエントリを追加することにより、pem サーバー上の既存の pg_hba.conf を変更できます。例

```
instances:
- Name: pemserver
  location: main
  node: 1
  role:
  - pem-server
  vars:
    pem_postgres_extra_hba_settings:
      - "# Allow pem connections from pg-cluster1.quire"
      - hostssl pem +pem_agent 10.33.15.108/32 cert
      - "# Allow pem connections from pg-cluster1.upside"
      - hostssl pem +pem_agent 10.33.15.104/32 cert
      - "# Allow pem connections from pg-cluster2.zippy"
      - hostssl pem +pem_agent 10.33.15.110/32 cert
      - "# Allow pem connections from pg-cluster2.utopic"
      - hostssl pem +pem_agent 10.33.15.109/32 cert
```

次に、 tpaexec provision \$clusters/pem-cluster に続いて tpaexec deploy \$clusters/pem-cluster を実行します。完了すると、新しい pg-クラスターのノードは pem サーバーバックエ

ンドと通信できるようになります。

5. pg-cluster 内のノードの pem エージェントが pem サーバーバックエンドに接続して登録できることを確認するには、tpaexec deploy を実行する前に最初に `export EDB_PEM_CREDENTIALS_FILE=/path/to/pem/credentials/file` を行う必要があります。資格情報ファイルは、pem サーバーのバックエンド postgres インスタンスへのアクセス資格情報を `username:password` 形式で含むテキストファイルです。

```
$ cat pem_creds
postgres:f1I%fw!QmWevdzw#EL#$Ulu1cWhg7&RT
```

バックエンドパスワードがわからない場合は、`show-password tpaexec` コマンドを使用して取得できます。

```
tpaexec show-password $pem-clusterdir $user
```

6. `tpaexec deploy $clusters/pg-cluster` を実行して、共有 pem サーバーインスタンスを使用しながら pem が新しい pg クラスターに展開されます。

注釈

混合プラットフォームのクラスター共有 PEM インスタンスを `platform: bare``` として宣言することにより、クラスターを混合プラットフォームクラスターに変更した可能性があります。これには、この変更に対応するように、`config.yml` の他の部分を調整する必要がある場合があります。具体的には、`instance_defaults` セクションには、クラスター内のすべてのインスタンスに適用される設定のみを含める必要があります。たとえば、`instance_defaults` に、`platform: aws` でのみ有効な `type` などの設定が含まれている場合、その設定を `instance_defaults` から AWS プラットフォームを使用するインスタンスにのみ移動する必要があります。

PEM UI への接続

PEM UI は https インターフェイスで実行されるため、`https://\protect\T1\textdollarpem-server-ip/pem` を介して PEM サーバーの実行中のインスタンスに接続できます。PEM UI のログイン資格情報は、`postgresql` および `epas` フレーバーのそれぞれに `postgres` または `enterprisedb` を使用する `postgres` バックエンドユーザーに設定されます。tpaexec の `show-password` コマンドは、バックエンドユーザーのパスワードを表示します。例

```
tpaexec show-password $clusterdir $user
```

PEM documentation を参照

PEM の構成と使用法の詳細については、こちらをご覧ください。

PgBouncer

PgBouncer パッケージバージョン

デフォルトでは、TPA は PgBouncer の最新バージョンをインストールします。

config.yml ファイルの cluster_vars セクションの下の pgbouncer_package_version: xxx を含むることにより、インストールされる PgBouncer パッケージのバージョンを指定できます。

```
cluster_vars:
  ...
  pgbouncer_package_version: 1.8*
  ...
```

apt または yum が受け入れるバージョン指定子を使用できます。

バージョンが一致しない場合は、* ワイルドカードを追加してみてください。これは、パッケージバージョンに 2:... のようなエポック修飾子がある場合に必要になることがよくあります。

PgBouncer の構成

TPA は、role に pgbouncer が含まれるインスタンスに PgBouncer をインストールおよび構成します。

デフォルトでは、PgBouncer はポート 6432 で接続を監視し、pgbouncer_backend が指定されない場合、接続を 127.0.0.1:5432 アーキテクチャに応じて Postgres または *Configuring haproxy* のいずれかに転送します。

注釈

PgBouncer を使用してトラフィックをプライマリにルーティングする repmgr で M1 アーキテクチャを使用している場合、cluster_vars の下で repmgr_redirect_pgbouncer: true ハッシュを設定して、PgBouncer 接続をプライマリに向けることができます。PgBouncer は、フェールオーバー時に自動的に更新され、新しいプライマリにルーティングされます。pgbouncer_backend をプライマリインスタンス名に設定することとこのオプションを組み合わせることで、プライマリにルーティングするように構成された PgBouncer でクラスターが最初に展開されるようにする必要があります。

Variable	Default value	Description
pgbouncer_port	6432	pgbouncer がリッスンする TCP ポート
pgbouncer_backend	127.0.0.1	接続する Postgres サーバー
pgbouncer_backend_port	5432	`pgbouncer_backend` がリッスンするポート
pgbouncer_max_client_conn	`max_connections` × 0.9	許可される接続の最大数。デフォルトは、可能であればバックエンドの `max_connections` 設定から取得されます
pgbouncer_auth_user	pgbouncer_auth_user	認証に使用する Postgres ユーザー

データベース

デフォルトでは、TPA は [databases] の下に単一のワイルドカード* エントリーを使用して/etc/pgbouncer/pgbouncer.databases.ini を生成し、すべての接続をバックエンドサーバーに転送します。次の例に示すように pgbouncer_databases を設定して、データベース構成を変更できます。

認証

PgBouncer は `pgbouncer_auth_user` として Postgres に接続し、構成済みの `auth_query` を実行してユーザーを認証します。

PgBouncer によって `auth_query` として使用される `pgbouncer_get_auth()` ファンクションは、単一のデータベース `pgbouncer_auth_database` に作成されます。このファンクションの実行権限は `pgbouncer_auth_user` に付与されます。

例

```
instances:

- Name: one
  vars:
    max_connections: 300

- Name: two

- Name: proxy
  role:
    - pgbouncer
  vars:
    pgbouncer_backend: one
    pgbouncer_databases:
      - name: dbname
        options:
          pool_mode: transaction
          dbname: otherdb
      - name: bdrdb
        options:
          host: two
          port: 6543
```

pgd-proxy

pgd-proxy パッケージバージョン

デフォルトでは、TPA は `pgd-proxy` の利用可能な最新バージョンをインストールします。

`config.yml` ファイルの `cluster_vars` セクションの下に `pgd_proxy_package_version: xxx` を含めることにより、インストールされる `pgd-proxy` パッケージのバージョンを指定できます。

```
cluster_vars:
  ...
  pgd_proxy_package_version: 5.0.0-1
  ...
```

apt または yum が受け入れるバージョン指定子を使用できます。

バージョンが一致しない場合は、* ワイルドカードを追加してみてください。これは、パッケージバージョンに 2:... のようなエポック修飾子がある場合に必要になることがよくあります。

pgd-プロキシの構成

TPA は、role に pgd-proxy を含むインスタンスに、PGD 5 を使用した PGD-Always-ON アーキテクチャの pgd-proxy をインストールおよび構成します。

デフォルトでは、*PGD-Always-ON*

は、すべての場所のすべてのデータノードで pgd-proxy を実行しますが、代わりに `--add-proxy-nodes-per-location 3` を使用して任意の数の追加プロキシインスタンスを作成できます。

構成

pgd-proxy は、SQL ファンクションを介して PGD レベルで構成されます。

Hash	Function	Description
<i>pgd_proxy_options</i>	<i>bdr.alter_proxy_option()</i>	pgd-proxy 構成、例ポート
<i>bdr_node_groups</i>	<i>bdr.alter_node_group_option()</i>	プロキシのノードグループの構成、例 <code>`enable_proxy_routing`</code>
<i>bdr_node_options</i>	<i>bdr.alter_node_option()</i>	個々の PGD ノードのルーティング構成

詳細については、PGD ドキュメントを参照してください。

pgd-proxy-config hook を使用できます

PGD PROXY 構成ファイルがインストールされた後にタスクを実行するには追加の構成ファイルをインストールします。

bdr_node_groups

pgd-proxy に関連するグループレベルのオプションは、他のノードグループオプションとともに `bdr_node_groups` の下で設定できます。

```
cluster_vars:
  bdr_node_groups:
    - name: group1
      options:
        enable_proxy_routing: true
```

グループで pgd-プロキシを有効にするには、`enable_proxy_routing` を明示的に `true` に設定する必要があることに注意してください。

bdr_node_options

pgd-proxy に関連するノードレベルのオプションは、PGD インスタンスの `bdr_node_options` の下で設定できます。

```
instances:

- Name: first
  vars:
    bdr_node_options:
      route_priority: 42
```

pgd_proxy_options

pgd-proxy インスタンスが接続されているグループまたはノードではなく、pgd-proxy インスタンス自身のオプションは、`default_pgd_proxy_options` の下、`cluster_vars` すべてのプロキシに適用されます、または pgd-proxy インスタンスの `pgd_proxy_options` の下で設定できます。

```
cluster_vars:
  default_pgd_proxy_options:
    listen_port: 6432
    read_listen_port: 6433

instances:

- Name: someproxy
  vars:
    pgd_proxy_options:
      listen_port: 9000
      read_listen_port: 9001
```

この場合、他のインスタンスは `cluster_vars` から `listen_port` 設定を取得しますが、`someproxy` はそのデフォルト設定をオーバーライドし、インスタンスの `vars` セクションで独自の `listen_port` を構成します。

PGD プロキシ http(s) ヘルスプローブ

プロキシの状態をモニタするための API エンドポイントを提供する PGD プロキシの `http(s)` サービスを有効にして構成できます。

`cluster_vars` またはインスタンス `vars` の下の `pgd_http_options` は、`pgd-proxy-config.yml` の `proxy` トップセクションの `http` サブセクションの下にある `http API` を定義するすべての設定を保存します。

変数には、次のキーを含めることができます。

```
enable: false
secure: false
cert_file: "/etc/tpa/harp_proxy/harp_proxy.crt"
```

(次のページに続く)

(前のページからの続き)

```
key_file: "/etc/tpa/harp_proxy/harp_proxy.key"
host: <inventory_hostname>
port: 8080
probes:
  timeout: 10s
endpoint: "<valid dsn>"
```

cert_file および key_file キーは、secure: true を使用し、独自の証明書とキーを使用する場合に必要です。

deploy を実行する前に、証明書とキーの両方がターゲットノードの指定された場所で利用できることを確認する必要があります。

クラスター固有の CA 証明書を使用して TPA に証明書とキーを生成する場合、cert_file および key_file の両方を空のままにします。TPA CA 証明書は「ウェルノウン」になりません。この証明書を、エンドポイントをプローブする各マシンの信頼ストアに追加する必要があります。CA 証明書は、TPA ノードのクラスターディレクトリで deploy の後の<cluster_dir>/ssl/CA.crt にあります。

使用可能な API エンドポイントの詳細については、pgd-proxy ドキュメントを参照してください。

pglogical

pglogical パッケージバージョン

デフォルトでは、TPA は pglogical の利用可能な最新バージョンをインストールします。

config.yml ファイルの cluster_vars セクションの下に pglogical_package_version: xxx を含めることにより、インストールされる pglogical パッケージのバージョンを指定できます。

```
cluster_vars:
  ...
  pglogical_package_version: 2.2.0*
  ...
```

apt または yum が受け入れるバージョン指定子を使用できます。

バージョンが一致しない場合は、* ワイルドカードを追加してみてください。これは、パッケージバージョンに 2:... のようなエポック修飾子がある場合に必要になることがよくあります。

pg 論理構成

TPA は、pglogical v2 および pglogical v3 を使用して pglogical レプリケーションセットパブリケーションとサブスクリプションを構成できます。

```
instances:
- node: 1
  Name: kazoo
```

(次のページに続く)

(前のページからの続き)

```

...
vars:
  publications:
    - type: pglogical
      database: example
      name: some_publication_name
      replication_sets:
        - name: custom_replication_set
        ...
- node: 2
  Name: keeper
  vars:
    subscriptions:
      - type: pglogical
        database: example
        name: some_subscription_name
        publication:
          name: some_publication_name
        replication_sets:
          - default
          - default_insert_only
          - custom_replication_set
        ...

```

`type: pglogical` でパブリケーションまたはサブスクリプションを定義すると、pglogical 拡張機能がデフォルトで作成されますが、どのバージョンをインストールするかは自分自身ですたとえば、pglogical3 の `products/pglogical3/release` リポジトリをサブスクライブします。

はじめに

TPA は、pglogical を使用してインスタンス間で変更をレプリケートするために必要なすべてを構成でき、`config.yml` の変更に基づいてレプリケーションセットアップを変更することもできます。

変更を公開するには、`publications` に `type: pglogical` を使用してエントリーを定義します。これらの変更をサブスクライブするには、上記のように、`subscriptions` に `type: pglogical` を使用してエントリーを定義します。

Pglogical には、名前付けパブリケーションエンティティがありませんビルトイン論理レプリケーションが `CREATE PUBLICATION` がある意味で、`config.yml` のパブリケーションは、レプリケーションセットのコレクションに名前を割り当てただけであり、サブスクリプションはこの名前を使用して、目的のプロバイダーを参照できます。

pg 論理レプリケーションを使用するには、パブリッシャーとサブスクライバーの両方が名前付けローカル pg 論理ノードが必要です。TPA はこのノードが存在しない場合、`pglogical.create_node()` で作成します。パブリケーションの場合、パブリケーション名が pg 論理ノード名として使用されます。特定のデータ

ベースには1つの pg 論理ノードのみが存在できるため、データベースごとに `publications` のエントリーは1つのみです。

ただし、pglogical サブスクリプション `do` には、独自の名前があります。TPA は、指定された `name` でサブスクリプションを作成し、インスタンスの名前とサブスクリプションが作成されるデータベースの名前に基づいて、pg 論理ノード名のデフォルト値を使用します。必要に応じて別の `node_name` を指定できます。たとえば、同じデータベース内のパブリケーションを構成している場合、そのデータベース内のすべてのサブスクリプションが同じ pg 論理ノードを共有する必要があります。

TPA は、構成の基本的な検証を行います。`replication_sets` を `replciation_sets` と書くか、定義されていないパブリケーションをサブスクライブしようとする、エラーを指摘しますが、意味のあるパブリケーションとサブスクリプションのセットを指定するのはあなたの責任です。

TPA は、ユーザー、拡張機能、データベースを作成した後、PGD 構成の前に pglogical を構成します。[Creating Postgres users](#) および [Creating Postgres databases](#) を設定してレプリケーション用のデータベースを作成し、`postgres-config-final` を使用できます。

pglogical が構成する前にデータベースにデータを設定するフック。

出版物

`publications` のエントリーは、`name` および `database` を指定する必要があり、オプションの属性を備えた名前付け `replication_sets` のリスト、およびテーブルまたはシーケンス名のリストを指定できます。

```
publications:

- type: pglogical
  database: example
  name: some_publication_name
  replication_sets:
  - name: default
    replicate_insert: true
    replicate_update: true
    replicate_delete: true
    replicate_truncate: true
    autoadd_tables: false
    autoadd_sequences: false
    autoadd_existing: true
  - name: custom_replication_set
    tables:
    - name: sometable
    - name: "some-schema".othertable
      columns: [a, b, c]
      row_filter: a > 42
      synchronize_data: true
    sequences:
    - name: someseq
      synchronize_data: true
    - name: "some-schema".otherseq
```

各レプリケーションセットは、`replicate_insert`、`autoadd_existing`などのオプションの属性を指定できます。指定した場合、それらは`pglogical.create_replication_set()`呼び出しに名前付けパラメーターとして含まれます。それ以外の場合、それらは省略され、代わりに`pglogical`のデフォルトでレプリケーションセットが作成されます。

`pglogical3`の`autoadd_*`パラメーターを使用してレプリケーションセットに属するリレーションのリストを操作することとは別に、テーブルまたはシーケンスのリストを明示的に指定することもできます。各リレーションの名前はスキーマ修飾されており、非修飾名は`public`にあるものと想定されており、エントリには、上に示しているように、`row_filter`テーブルのみ、`synchronize_data`などのオプションの属性が含まれる場合があります。

サブスクリプション

`subscriptions`のエントリは、`name`および`database`を指定し、サブスクライブするパブリケーションを定義する必要があり、サブスクリプションの他のオプション属性を指定する場合があります。

```
subscriptions:

- type: pglogical
  database: example
  name: some_subscription_name
  node_name: optional_pglogical_node_name
  publication:
    name: some_publication_name
  # Optional attributes:
  synchronize_structure: true
  synchronize_data: true
  forward_origins: [all]
  strip_origins: false
  apply_delay: 1 second
  writer: heap
  writer_options:
    - magic
    - key=value
    - just-a-string
  # Optional attributes that can be changed for an existing
  # subscription:
  replication_sets:
    - default
    - default_insert_only
    - custom_replication_set
  enabled: true
```

サブスクリプションは`publication.name`を設定して、サブスクライブするパブリケーションを定義できます。その名前のクラスター全体に複数のパブリケーションがある場合、明確にするインスタンスの名前を指定できます。パブリケーションを名前参照したい場合は、同じインスタンスに同じ名前でも複数のパブリケーションを作成しないでください。

```
- type: pglogical
...
publication:
  name: some_publication_name
  instance: kazoo

# OR

provider_dsn: "host=... dbname=..."
```

パブリケーションを名前で参照する代わりに、代わりに `provider_dsn` を明示的に指定できます。この場合、指定された DSN は `pglogical.create_subscription()` に直接渡され、`publication` は無視されます。このメカニズムを使用して、TPA クラスターの外部のインスタンスをサブスクライブできます。

上記の例の他の属性はオプション。定義されている場合、それらは `pglogical.create_subscription()` 呼び出しに名前付けパラメーターとして含まれます。それ以外の場合、それらは除外されます。表示される一部の属性は `pglogical3` に固有です。

構成の変更

パブリケーションの場合、レプリケーションセットの追加または削除、レプリケーションセットの属性の変更、またはそのメンバーシップそれに含まれるテーブルとシーケンスを変更できます。

`replicate_*` または `autoadd_*` を変更すると、TPA はそれに応じて `pglogical.alter_replication_set()` を呼び出しますが、既存のレプリケーションセットの `autoadd_existing` は変更できず、`autoadd_*` パラメーターはすべて `pglogical3` 固有であることに注意してください。

レプリケーションセットの `tables` または `sequences` のリストを変更すると、TPA は必要に応じて `pglogical.alter_replication_set_{add,remove}_{table,sequence}()` を呼び出すことによりこれらの変更を調整します。

ただし、レプリケーションセットのメンバーであるリレーションテーブルまたはシーケンスの `synchronize_data` または他の属性を変更した場合、TPA は変更を伝播しませんたとえば、テーブルをドロップし、別の構成で再追加します。

サブスクリプションの場合、`replication_sets` のリストを変更し、サブスクリプション `enabled: false` を有効または無効にすることのみができます。

どちらの場合も、存在するが構成に記載されていないレプリケーションセットは、サブスクライバーで `pglogical.alter_subscription_remove_replication_set()`、またはパブリッシャーで `pglogical.drop_replication_set()` を使用して削除されます。ただし、`default`、`default_insert_only`、および `ddl_sql` という名前のデフォルトのレプリケーションセットはドロップされません。

`config.yml` を編集する場合は、`tpaexec deploy` を実行する前に `tpaexec provision` を実行してください

PGD との相互作用

注意すれば、同じデータベースで PGD と pglogical を一緒に使用することが可能です。

PGD v3 は内部で pglogical3 を使用し、pglogical ノードが存在しない場合は作成します。データベースごとに pg 論理ノードは 1 つだけであるため、bdr_database で pg 論理パブリケーションを構成する場合、インスタンスの bdr_node_name はパブリケーションの name と同じである必要があります。それ以外の場合、最初にパブリケーション用にノードが作成され、後で bdr.create_node() はノード名の競合に関するエラーで失敗します。bdr_database の subscriptions は、同じ node_name も使用する必要があります。

制限事項

- 現在 pglogical.replication_set_{add,remove}_ddl() のサポートはありません
- 現在 pglogical.replication_set_add_all_{tables,sequences}() のサポートはありません
- 現在、pglogical.alter_subscription_{interface,writer_options}() または pglogical.alter_subscription_{add,remove}_log() のサポートはありません
- pglogical v1 のサポートは現在テストされていません。

repmgr

TPA は、failover_manager インスタンス変数が repmgr に設定されているすべての postgres インスタンスに repmgr をインストールします。これはデフォルト設定です。

repmgr 構成ファイルのディレクトリは、デフォルトで/etc/repmgr/<version> になります。ここで、<version> は、このインスタンスにインストールされる postgres のメジャーバージョンですが、インスタンスの repmgr_conf_dir 変数を設定することにより変更できます。構成ファイル自身自身は常に repmgr.conf と呼ばれます。

デフォルトの repmgr 構成は、ロール primary とロール replica で構成されたインスタンス間の自動フェイルオーバーをセットアップします。

repmgr パッケージバージョン

デフォルトでは、TPA は repmgr の最新バージョンをインストールします。

インストールされる repmgr パッケージのバージョンは、config.yml ファイルの cluster_vars セクションの下で repmgr_package_version: xxx を含めることにより指定できます。

```
cluster_vars:
  ...
  repmgr_package_version: 4.0.5-1.pgdg90+1
  ...
```

apt または yum が受け入れるバージョン指定子を使用できます。

バージョンが一致しない場合は、* ワイルドカードを追加してみてください。これは、パッケージバージョンに 2:... のようなエポック修飾子がある場合に必要になることがよくあります。

repmgr 構成

次のインスタンス変数を設定できます。

`repmgr_priority` : sets priority in the config file
`repmgr_location` : sets location in the config file
`repmgr_reconnect_attempts` : sets reconnect_attempts in the config file, default 6
`repmgr_reconnect_interval` : sets reconnect_interval in the config file, default 10
`repmgr_use_slots` : sets use_replication_slots in the config file, default 1
`repmgr_failover` 構成ファイルで failover を設定します、デフォルト automatic

`repmgr_conf_settings` の追加設定も `repmgr` 構成ファイルに渡されます。

PGD インスタンスの## `repmgr`

PGD インスタンスでは、`repmgr_failover` はデフォルトで `manual` に設定されます。

7.5.4 Configuration

Configuring EDB Repos 2.0 repositories

TPA は、EDB Repos 2.0 から EDB ソフトウェアをソースします。EDB Repos 2.0 を使用するには、`tpaexec` を実行する前に `export EDB_SUBSCRIPTION_TOKEN=xxx` を行う必要があります。サブスクリプショントークンは [the web interface](#) から取得できます。

注釈

`tpaexec configure` コマンドを使用して `config.yml` ファイルを作成する場合、`edb_repositories` キーには、選択した構成に必要なリポジトリが自動的に入力されるため、編集する必要はありません。

各インスタンスにインストールする EDB Repos 2.0 のリポジトリの完全なリストを指定するには、`edb_repositories` を EDB リポジトリ名のリストに設定します。

```
cluster_vars:
  edb_repositories:
    - enterprise
    - postgres_distributed
```

この例では、'enterprise' サブスクリプションリポジトリと、EPAS および PGD 製品へのアクセスを付与する 'postgres_distributed' をインストールします。Debian または Ubuntu システムでは、APT リポジトリを使用し、RedHat または SLES システムでは、それぞれ yum または zypper フロントエンドを介して rpm リポジトリを使用します。

Configuring APT repositories

このページでは、Debian および Ubuntu システムで APT パッケージリポジトリを構成する方法について説明します。

`apt_repositories` で名前付けリポジトリを定義し、`apt_repository_list` で名前をリストすることによりどのリポジトリを使用するかを決定できます。

```
cluster_vars:
  apt_repositories:
    Example:
      key_id: XXXXXXXX
      key_url: https://repo.example.com/path/to/XXXXXXXX.asc
      repo: >-
        deb https://repo.example.com/repos/Example/ xxx-Example main

  apt_repository_list:
    - PGDG
    - Example
```

この構成では、GPG キー `key_url` から取得した ID `key_id` を使用して、PGDG リポジトリデフォルトで既に定義されているものと新しい例リポジトリの指定された `repo` 行複数可を使用して `/etc/apt/sources.list.d` の下に新しいエントリーをインストールします。

追加のリポジトリを構成するとき、PGDG パッケージをインストールする場合は、`apt_repository_list` に PGDG を含めることを忘れないでください。

リポジトリをインストールしないように `apt_repository_list: []` を設定できます。

Configuring YUM repositories

このページでは、RedHat システムで YUM パッケージリポジトリを構成する方法について説明します。

`yum_repositories` で名前付けリポジトリを定義し、`yum_repository_list` で名前をリストすることによりどのリポジトリを使用するかを決定できます。

```
cluster_vars:
  yum_repositories:
    Example:
      rpm_url: >-
        https://repo.example.com/repos/Example/example-repo.rpm

    Other:
      description: "Optional repository description"
      baseurl: https://other.example.com/repos/Other/$basearch
      gpgkey:
        https://other.example.com/repos/Other/gpg.XXXXXXXXXXXXXXXXXX.key

  yum_repository_list:
    - EPEL
    - PGDG
    - Example
    - Other
```

この例では、YUM リポジトリを定義する 2 つの方法を示しています。

リポジトリに「repo RPM」つまり、必要な/etc/yum.repos.d/*.repo ファイルと、リポジトリから署名されたパッケージを検証するために必要な GPG キーを慣例的にインストールするパッケージがある場合、それをポイントするだけで済みます。

それ以外の場合は、説明、baseurl、および gpgkey URL を指定できます。TPA は、この情報に基づいて/etc/yum.repos.d/other.repo ファイルを作成します。

EPEL および PGDG リポジトリはデフォルトで定義されています。正常な操作には EPEL リポジトリが必要であるため、常に EPEL を yum_repository_list に含める必要があります。PGDG パッケージをインストールする場合は、PGDG も含める必要があります。

リポジトリをインストールしないように yum_repository_list: [] を設定できますが、EPEL パッケージの代替ソースがないと壊れます。

リポジトリアクセスを構成する特別な手順を実行する必要がある場合は、[TPA hooks](#) を使用して.repo ファイルを自分自身で作成できます。

```
- name: Define Example repository
  copy:
    dest: /etc/yum.repos.d/example.repo
    owner: root
    group: root
    mode: "0644"
    content: |
      [example]
      name=Example repo
      baseurl=https://repo.example.com/repos/Example/
      enabled=1
      gpgkey=https://repo.example.com/repokey.asc
      gpgcheck=1
```

この場合、yum_repository_list にリポジトリをリストする必要はありません。

Creating and using a local repository

クラスターディレクトリ内にローカルリポジトリを作成すると、TPA はリポジトリ内のパッケージをクラスターインスタンスで利用できるようにします。これは、追加のパッケージをクラスターにシップする簡単な方法です。

オプションで、このリポジトリのみを使用するようにインスタンスを構成するように TPA に指示することもできます。つまり、他のすべてを無効にします。この場合、rsync、Python などの基本的な依存関係から始めて、展開中に必要なすべての*パッケージを提供する必要があります。

ローカルリポジトリを手動で作成するか、TPA にローカルリポジトリを作成してもらうことができます。両方の手順を以下に示します。

注釈

特定の指示は *Managing clusters in a disconnected or air-gapped environment* 環境で使用できます。

TPA を使用したローカルリポジトリの作成

TPA には、このようなローカルリポジトリの作成に役立つツールが含まれています。具体的には、`--enable-local-repo` スイッチを `tpaexec configure` とともに使用して、ローカルリポジトリとして使用する空のディレクトリ構造を作成できます。`tpaexec download-packages` は、その構造に必要なパッケージを設定します。

ディレクトリ構造の作成

ローカルリポジトリでクラスターを構成するには、次のコマンドを実行します。

```
tpaexec configure --enable-local-repo ...
```

これにより、クラスター構成が生成され、`local-repo` ディレクトリと OS 固有のサブディレクトリが作成されます。[ローカルリポジトリのレイアウト](#) については、以下を参照してください。

リポジトリにデータを設定し、メタデータを生成する

`tpaexec download-packages` を実行して、クラスターに必要なすべてのパッケージをローカルリポジトリにダウンロードします。結果のリポジトリには、すべてのパッケージの完全な依存関係ツリーが含まれるため、このリポジトリからクラスター全体をインストールできます。リポジトリのメタデータも自動的に作成され、すぐに使用できます。

ローカルリポジトリの手動作成

ローカルリポジトリのレイアウト

ローカルリポジトリを手動で作成するには、最初に適切なディレクトリ構造を作成する必要があります。`--enable-local-repo` を使用する場合、TPA は、クラスター用に選択した OS に基づいて、`local-repo` ディレクトリとその内に OS 固有のサブディレクトリたとえば `local-repo/Debian/12` を作成します。この構造は、手動で作成したリポジトリにも使用することをお勧めします。

たとえば、RedHat 8 を実行しているクラスターは次のレイアウトになる場合があります。

```
local-repo/
`-- RedHat
    |-- 8.5 -> 8
    `-- 8
        `-- repodata
```

インスタンスごとに、TPA は `local-repo` の次のサブディレクトリを順番に検索し、最初に見つけたものを使用します。

- `<distribution>/<version>`、例 RedHat/8.5
- `<distribution>/<major version>`、例 RedHat/8
- `<distribution>/<release name>`、例 Ubuntu/focal
- `<distribution>`、例 Debian
- `local-repo` ディレクトリ自分自身。

これらのディレクトリが存在しない場合、もちろん、TPA はターゲットインスタンスにローカルリポジトリをセットアップしようとしません。

リポジトリの設定とメタデータの生成

`tpaexec deploy` を実行する前に、以下で詳しく説明する手順を完了する必要があります。

リポジトリにデータを設定するには、含めるパッケージを適切なディレクトリにコピーします。次に、以下で詳しく説明するように、システムに適切なツールを使用してメタデータを生成します。

i 注釈

制御ノードつまり `tpaexec` を実行するマシンでメタデータを生成する必要があります。TPA は、メタデータとパッケージをターゲットインスタンスにコピーします。

i 注釈

インスタンスが使用するサブディレクトリにメタデータを生成する必要があります。つまり、パッケージを `local-repo/Debian/12` にコピーする場合、`local-repo/Debian` ではなくそのディレクトリにメタデータを作成する必要があります。

Debian / Ubuntu リポジトリのメタデータ

Debian ベースのディストリビューションの場合、`dpkg-dev` パッケージをインストールします。

```
$ sudo apt-get update && sudo apt-get install -y dpkg-dev
```

これで、`dpkg-scanpackages` を使用してメタデータを生成できます。

```
$ cd local-repo/Debian/bookworm  
  
# download/copy .deb package files  
  
$ dpkg-scanpackages . | gzip > Packages.gz
```

RedHat / SLES リポジトリのメタデータ

最初に、`createrepo` パッケージをインストールします。

```
$ sudo yum install -y createrepo
```

これで、`createrepo` を使用してメタデータを生成できます。

```
$ cd local-repo/RedHat/8  
  
# download/copy .rpm package files
```

(次のページに続く)

(前のページからの続き)

```
$ createrepo .
```

TPA がローカルリポジトリを使用する方法

リポジトリのコピー

TPA は rsync を使用して、生成されたメタデータを含むリポジトリディレクトリの内容をターゲットインスタンス上のディレクトリにコピーします。

rsync がインスタンスでまだ利用できない場合、TPA はそれをインストールできますつまり `apt-get install rsync` または `yum install rsync`。ただし、`use_local_repo_only` を設定している場合、rsync パッケージはローカルリポジトリに含まれる必要があります。必要に応じて、TPA は scp を使用して rsync パッケージのみをコピーし、残りをコピーする前にインストールします。

リポジトリ構成

ローカルリポジトリのコンテンツをターゲットインスタンスにコピーした後、TPA は宛先ディレクトリをローカルつまり、URL ベースではなくパスベースのリポジトリとして構成します。

たとえば、リポジトリディレクトリに `example.deb` を提供する場合、他のリポジトリのパッケージと同様に、`apt-get install example` を実行するだけでインストールできます。

パッケージのインストール

TPA は、提供されたコンテンツでリポジトリを構成しますが、同じパッケージが別のリポジトリから利用可能な場合、どれをインストールするかはパッケージマネージャーの判断にあります特定のバージョンを指定しない限り、通常は最新です。

ただし、`use_local_repo_only: yes` を設定すると、TPA は他のすべてのパッケージリポジトリを無効にするため、インスタンスは `local-repo` で提供したパッケージのみを使用できます。

Installing from source

`config.yml` で `install_from_source` を設定することにより、Git リポジトリからビルドおよびインストールする拡張機能のリストを定義できます。

```
cluster_vars:
  install_from_source:
    - name: ext
      git_repository_url: https://repo.example.com/ext.git
      git_repository_ref: dev/example

    - name: otherext
      git_repository_url: ssh://repo.example.com/otherext.git
      git_repository_ref: master
      source_directory: /opt/postgres/src/otherext
```

(次のページに続く)

(前のページからの続き)

```

build_directory: /opt/postgres/build/otherext
build_commands:
  - "make -f /opt/postgres/src/otherext/Makefile install"
build_environment:
  VAR: value

```

TPA はリストされた順序で拡張機能を 1 つずつビルドしてインストールするため、正しい順序でメンションすることにより、別の拡張機能 pglogical や BDR などに依存する拡張機能をビルドできます。

ビルドするには、各エントリーで `name`、`git_repository_url`、および `git_repository_ref` デフォルト `master` を指定する必要があります。 *Git credentials* を使用できます

Git リポジトリに対して認証するため。また、上記のように `source_directory`、`build_directory`、`build_environment`、および `build_commands` を設定します。

`tpaexec deploy ... --skip-tags build-clean` を実行して、展開を繰り返すときにビルドディレクトリを再利用します。それ以外の場合、古いビルドディレクトリはビルドを開始する前に空にされます。 **ローカルソースディレクトリ** を構成することもできます。

開発ビルドを高速化します。

ソースビルドを実行するたびに、Postgres が再起動されます。

ビルドの依存関係

ソースからビルドする場合、TPA は、基本的な Postgres ビルドの依存関係がインストールされることを保証します。追加パッケージが必要な場合は、 *Installing packages* に記載してください。例

```

cluster_vars:
  packages:
    common:
      - golang-1.16

```

Git credentials

このページでは、認証が必要な Git リポジトリのクローンを作成する方法について説明します。

これは、`postgres_git_url` を *Postgres source installation* または **ソースからのビルドとインストール** に変更して拡張機能をコンパイルおよびインストールする場合に必要な場合があります。

資格情報をターゲットインスタンスのディスクに書き込まずに認証するには、2つのオプションがあります。

- `ssh://` リポジトリの場合、SSH キーをローカル `ssh-agent` に追加できます。 `--install-from-source` `config.yml` の `forward_ssh_agent: yes` を使用する場合、エージェント転送はデフォルトで有効になります。
- `https://` リポジトリの場合、`tpaexec deploy` を実行する前に環境で `export TPA_GIT_CREDENTIALS=username:token` を実行できます。

i 注釈

この環境ではホストから ssh で Docker コンテナにアクセスできないため、macOS で Docker に展開する場合、`https://` リポジトリ URL のみを使用する必要があります。

SSH キー認証

SSH リポジトリのクローンを作成しており、SSH キーペア `id_example` および `id_example.pub` がある場合、SSH エージェント転送を使用してターゲットインスタンスで認証します。

- `ssh-agent` をローカルで実行する必要があります。デスクトップ環境がこれをまだ設定していない場合は、ほとんどの場合 `pgrep ssh-agent` が実行されているかを確認します `ssh-agent bash` を実行してエージェントを有効にした新しいシェルを一時的に起動し、そのシェルから `tpaexec deploy` を実行します。
- `ssh-add /path/to/id_example` プライベートキーファイルを使用して、必要なキーをエージェントに追加します**
- `tpaexec provision` の前に `config.yml` のトップレベルで `forward_ssh_agent: yes` を設定することにより、SSH エージェント転送を有効にします**。 `--install-from-source` を使用する場合は、これはデフォルトで行われます。

展開中に、エージェントに追加したキーは、転送されたエージェント接続を介してリモートサーバーへの認証に利用できるようになります。

SSH エージェント転送は注意して使用し、できればこの目的のために専用に生成された使い捨てキーペアを使用します。ターゲットサーバー上のエージェントの Unix ドメインソケットにアクセスする権限を持つユーザーは、他のサーバーへの認証中にエージェントを組み込んでなりすましができます。

HTTPS のユーザー名/パスワード認証

ユーザー名と認証トークンまたはパスワードを使用して HTTPS リポジトリのクローンを作成する場合は、`tpaexec deploy` の前に環境で `export TPA_GIT_CREDENTIALS=username:token` だけを作成します。デプロイメント中に、これらの資格情報は `git clone` または `git pull` タスクのみで利用できるようになります。これらはターゲットインスタンスのディスクに書き込まれません。

Environment

`target_environment` を設定して、展開中に TPA がターゲットインスタンスに設定する必要がある環境変数を指定できますたとえば、以下に示すように HTTPS プロキシを指定します。

```
cluster_vars:
  target_environment:
    https_proxy: https://proxy.example:8080
```

TPA は、これらの設定が、展開中、およびその後のクラスター管理コマンドの実行中に、必要な他の設定とともに環境に存在することを確認します。

これらの環境設定は永続的なものではありませんが、代わりに *The postgres Unix user* を使用して postgres ユーザーの環境変数を設定できます。

Python environment

TPA は、ターゲットインスタンスの *Distribution support* に基づいて、どの Python インタープリターを使用するかを決定します。可能な場合は常に Python 3 を使用し、やむを得ない場合にのみ Python 2 にフォールバックします。

`tpaexec configure` コマンドは、ディストリビューションに応じて `preferred_python_version` を設定します。

Distribution	Python 2	Python 3
Debian 12/bookworm	Yes	Yes (3.11)
Debian 11/bullseye	Yes	Yes (3.9)
Debian 10/buster	Yes	Yes (3.7)
Ubuntu 24.04/jammy	No	Yes (3.12)
Ubuntu 22.04/jammy	No	Yes (3.10)
Ubuntu 20.04/focal	No	Yes (3.8)
RHEL 9.x	No	Yes (3.9)
RHEL 8.x	No	Yes (3.6)
RHEL 7.x	Yes	No (3.6)

Ubuntu 20.04、22.04、および RHEL 8.x は、Python 3 でのみ使用できます。

RHEL 7.x には Python 3.6 が同梱されていますが、システム Python 3 の `librpm` バインディングは利用できません。

他のディストリビューションの場合、`python2` または `python3` のどちらを優先するかを決定できますが、新しいクラスターのデフォルトは `python3` です。

下位互換性

既存のクラスターとの互換性のため、`preferred_python_version` のデフォルト値は `python2` ですが、`python2` で既に展開されたシステムでも `python3` を明示的に選択できます。

cluster_vars:

preferred_python_version: python3

TPA は、Python 3 を使用できないディストリビューションではこの設定を無視します。

Configuring /etc/hosts

デフォルトでは、TPA はクラスター内のすべてのインスタンスの IP アドレスとホスト名を使用してターゲットインスタンスの `/etc/hosts` に行を追加して、クラスター内の通信に相互の名前を使用できるようにしますたとえば `primary_conninfo` Postgres の場合)。

`extra_etc_hosts_lines` のリストを指定することもできます。

instances:

(次のページに続く)

(前のページからの続き)

```
- Name: one
...
vars:
  extra_etc_hosts_lines:
    - 192.0.2.1 acid.example.com
    - 192.0.2.2 water.example.com
```

デフォルトエントリがまったく必要ない場合は、代わりにインスタンスの `etc_hosts_lines` の完全なリストを指定できます。これらの行のみが `/etc/hosts` に追加されます。

```
instances:

- Name: one
...
vars:
  etc_hosts_lines:
    - 192.0.2.1 acid.example.com
    - 192.0.2.2 water.example.com
    - 192.0.2.3 base.example.com
```

`/etc/hosts` にクラスター内のインスタンスのデフォルトエントリが含まれていない場合、他の方法で名前を解決できることを確認する必要があります。

Filesystem configuration

TPA では、各インスタンスにアタッチされる `volumes` のリストを定義できます。

このリストは、プロビジョニング中に使用されるプラットフォーム固有の設定と、展開中に使用されるファイルシステムレベルの設定の両方で構成されています。

最初に、`tpaexec provision` は、情報を使用してボリュームを作成し、インスタンスに接続します該当する場合。詳細については、以下のプラットフォーム固有のセクションを参照してください。次に、ボリュームの単純化されたリストプラットフォーム固有ではない設定のみを含むをインスタンスのホスト変数として書き込みます。最後に、`tpaexec deploy` は、単純化されたリストに基づいて動作し、必要に応じてファイルシステムをセットアップおよびマウントします。

次に、AWS クラスターからの適度に複雑な例を示します。

```
instances:

- Name: one
...
volumes:
  - device_name: root
    volume_type: gp2
    volume_size: 32
  - device_name: /dev/xvdf
```

(次のページに続く)

(前のページからの続き)

```

volume_type: io2
volume_size: 64
iops: 5000
vars:
  volume_for: postgres_data
  encryption: luks
- device_name: /dev/xvdb
  ephemeral: ephemeral0
vars:
  mountpoint: /mnt/scratch

```

この例では、EC2 インスタンスは、32GB EBS ルートボリューム、/opt/postgres/data としてマウントされた 64GB io2 ボリューム provisioned-iops EBS ボリューム、およびインスタンスストアによって提供される /tmp/scratch ファイルシステムになります。「エフェメラル」ボリューム。その数とサイズは、インスタンスタイプによって決まります。

詳細は以下の AWS に関するセクションに記載されていますが、`volume_type` や `volume_size` のような設定はプロビジョニング中に使用され、`volume_for` や `mountpoint` などの `vars` の下の設定は、展開中に使用するためにインベントリに書き込まれます。

default_volumes

ボリュームはインスタンスのプロパティです。プラットフォーム固有の設定が含まれるため、`cluster_vars` では設定できません。

instance_defaults

このメカニズムでは、ボリューム定義に特別な考慮事項を作成します。大規模なクラスター内のボリューム定義は非常に反復的な場合があるため、特にクラスター内のインスタンスを可能な限り互いに近づけて構成することをお勧めしますので、ここに示すように `default_volumes` を指定できます。

```

instance_defaults:
  default_volumes:
    - device_name: root
      volume_type: gp2
      volume_size: 32
    - device_name: /dev/xvdf
      volume_size: 100

instances:
- Name: one
  ...

- Name: two
  volumes:
    - device_name: /dev/xvdf

```

(次のページに続く)

(前のページからの続き)

```

    volume_size: 64
  - device_name: /dev/xvdg
    volume_size: 64
    ...

- Name: three
  volumes:
  - device_name: /dev/xvdf
    volume_type: none

- Name: four
  volumes: []

```

ここで、すべてのインスタンスには、デフォルトで 32GB のルートボリュームと 100GB の追加ボリュームがあります他に何も指定しないインスタンス **one** の場合と同様に。インスタンス **two** は同じルートボリュームがありますが、代わりに 64GB になるように `/dev/xvdf` をオーバーライドし、さらに別の 64GB ボリュームがあります。インスタンス **three** には同じルートボリュームがありますが、デフォルトの `/dev/xvdf` に `volume_type: none` を設定するため、追加ボリュームはありません。インスタンス **four** にはボリュームがまったくありません。

インスタンスは `default_volumes` で指定されたもので始まり、その `volumes` エントリは、同じ `device_name` でデフォルトエントリをオーバーライドしたり、`volume_type` を `none` に設定してボリュームを削除したり、別の名前で新しいボリュームを追加したり、デフォルトを完全に拒否したりできます。

2つのリストをマージするこの動作は、`default_volumes` に固有です。`instance_defaults` と `instances` の両方で他のリストを設定すると、後者は前者を完全にオーバーライドします。

プラットフォーム AWS

AWS EC2 インスタンスでは、EBS ボリュームをアタッチできます。

```

instances:

- Name: one
  ...
  volumes:
  - device_name: root
    volume_type: gp2
    volume_size: 32
    encrypted: yes
    ...
  - device_name: /dev/xvdf
    volume_type: io1
    volume_size: 32
    iops: 10000
    delete_on_termination: false

```

(次のページに続く)

(前のページからの続き)

```
...
- device_name: /dev/xvdg
  ephemeral: ephemeral0
...
```

TPA は、インスタンスタイプに基づいて、root の `device_name` を `/dev/sda` または `/dev/xvda` に変換するため、どれを使用するかを覚えたり変更したりする必要はありません。

`volume_type` は、EBS ボリュームタイプ、たとえば `gp2` 「汎用」EBS ボリュームの場合、プロビジョニングされた IOPS ボリュームの `io1 iops: 5000` も設定する必要があります。などを指定します。

`volume_size` は、ボリュームのサイズをギガバイト単位で指定します。

`encrypted: yes` を設定して、保存時の EBS 暗号化を有効にします。(これは AWS 機能であり、新しく生成された TPA 構成でデフォルトで有効になり、以下で説明する [LUKS 暗号化](#) とは異なります。

`delete_on_termination` を `false` に設定して、アタッチされたインスタンスが終了したときにボリュームが破壊されないようにします。これがデフォルトの動作です。

物理的に接続された [instance store volume](#)、以前はエフェメラルボリュームと呼ばれていたを使用するように `ephemeral: ephemeralN` を設定します。使用可能なインスタンスストアボリュームの数、タイプ、およびサイズは、インスタンスタイプによって異なります。すべてのインスタンスにインスタンスストアボリュームがあるわけではありません。インスタンスストアボリュームはテストまたは一時データにのみ使用し、EBS ボリュームは重要なデータに使用します。

EBS ボリュームの場合、既存のスナップショットからボリュームを接続するように `snapshot: snap-xxxxxxx` を設定することもできます。十分なデータが S3 から読み込まれローカルにキャッシュされるまで、スナップショットから復元されたボリュームは非常に遅くなる場合があります。(特に、スナップショットから PGDATA を使用して新しいインスタンスをスピニングアップできますが、完全なロードを処理する準備が整うまでに数時間かかることが予想されます。

ボリュームに `attach_existing: yes` を設定し、名前/タイプ/サイズ/iops と一致する既存の未接続の EBS ボリュームが存在する場合、インスタンスの起動時に新しいボリュームは作成されませんが、代わりに既存のボリュームがインスタンスにアタッチされます初めて始まります。再接続された EBS ボリュームは、スナップショットから作成されたボリュームのパフォーマンス制限の影響を受けません。

プラットフォームベア

TPA は、事前にプロビジョニングされた `bare` インスタンスにどのボリュームを接続できるかを制御できませんが、適切な `device_name` で `volumes` を定義すると、必要に応じてデバイスの `mkfs` および `mount` を処理します。

Platform Docker

Docker コンテナにはボリュームを接続できますが、それらはバインドマウントされたディレクトリであり、通常のブロックデバイスではありません。個別に初期化またはマウントする必要はありません。そのため、構成はかなり異なって見えます。

```
instances:

- Name: one
  platform: docker
  ...
  volumes:
    - /host/path/to/dir:/tmp/container/path:ro
    - named_volume:/mnt/somevol:rw
```

これらのボリューム仕様は、`docker run -v` への引数として認識される場合があります。

ボリュームはコンテナの作成時に接続され、展開中にそれ以上のアクションはありません。

RAID アレイ

AWS EC2 インスタンスでは、RAID 0 のみが Amazon によって推奨されています。同様のコマンドで RAID ボリュームを作成できます。

```
sudo mdadm --create --verbose /dev/md0 --level=0 --name=MY_RAID --raid-devices=number_
→of_volumes device_name1 device_name2
```

この例では、`/dev/md0` という名前のブロックデバイスを接続します。導入中の `volume_for` または `mountpoint` の処理は、他のボリュームの場合と同じように発生します。TPA は、`mkfs` および `mount` を処理します。

```
- Name: one
  ...
  volumes:
    - device_name: /dev/md0
      vars:
        volume_for: postgres_data
```

LUKS 暗号化

TPA は、LUKS で暗号化されたデバイスをセットアップできます。

```
instances:

- Name: one
  ...
  volumes:
    - device_name: /dev/xyz
      vars:
        encryption: luks
        luks_volume: mappedname
        volume_for: ...
```

encryption: luks セットがまだ初期化されていない場合、TPA は他のデバイスと同様にファイルシステムの作成を処理する前に、`cryptsetup` を使用して最初に `luksFormat`、次に `luksOpen` を使用して `/dev/mapper/mappedname` の下にマップします。

データ損失の可能性を回避するため、TPA は、有効なファイルシステムが既に含まれているデバイスでの LUKS 暗号化のセットアップを拒否します。

LUKS で暗号化された `volume_for: postgres_data` を作成する場合、TPA はブート時に Postgres が自動的に起動しないように構成します。`tpaexec start-postgres clustername` を使用してボリュームをマウントし、Postgres を起動できます。および `stop-postgres` を使用して Postgres を停止し、ボリュームのマップを解除します。

LUKS パスフレーズはローカルで生成され、ボールトに保存されます。

ファイルシステムの作成とマウント

`device` に有効なファイルシステムが含まれていない場合、`mkfs` で初期化されます。

```
instances:

- Name: one
  ...
  volumes:
  - device_name: /dev/xyz
    vars:
      volume_for: ...
      fstype: ext4
      fsopts:
        - -cc
        - -m 2
      mountopts: defaults,relatime,nosuid
      readahead: 65536
      owner: root
      group: root
      mode: "0755"
```

`fstype` デフォルト `ext4`、`mkfs` デフォルト `none` に渡される `fsopts`、および `mount` に渡されて `fstab` に書き込まれる `mountopts` を指定できます。以下を参照してください。

TPA は、デバイスの先読みをデフォルトで 16MB に設定し、リブート後も値を持続させますが、上記のようにボリュームに別の値を指定できます。

ボリュームがマウントされている場所を決定するには2つの方法があります。`mountpoint` を明示的に指定するか、`volume_for` を `postgres_data`、`postgres_wal`、`postgres_tablespace` または `barman_data` に設定できます。TPA は設定をシステムの適切なマウントポイントに変換します。

`mountpoint` が決定されると、`device` は指定された `mountopts` デフォルト `defaults,noatime` とともにそこにマウントされます。`/etc/fstab` のファイルシステムのエントリーも作成されます。

オプションでボリュームに `owner`、`group`、または `mode` を指定できます。これらの属性は `mountpoint` に設定されます。デプロイメントのこの非常に初期段階では、`postgres` ユーザーの存在を当てにできない

ことに注意してください。いずれの場合も、TPA は Postgres が必要とするディレクトリに適切な所有権と権限があることを別途に保証するため、自分で行う必要はありません。

Uploading artifacts

artifacts を定義して、ファイルをターゲットインスタンスに作成またはコピーできます。

```
cluster_vars:
  artifacts:
    - type: path
      path: /some/target/path
      state: directory
      owner: root
      group: root
      mode: "0755"
    - type: file
      src: /host/path/to/file
      dest: /target/path/to/file
      owner: root
      group: root
      mode: "0644"
    - type: archive
      src: example.tar.gz
      dest: /some/target/path
    - type: directory
      src: /host/path/a/
      dest: /target/path/b/
```

次のタイプがサポートされています。

- `path` を使用して、ファイルとディレクトリの所有権またはモードを作成または削除、および変更します内部で使用する Ansible の `file` モジュールと同じパラメーターを取得します
- `file` を使用してコントローラからファイルをコピーし、所有権とモードを設定します `copy` を使用
- `archive` を使用して、アーカイブから指定した場所にファイルを抽出します `unarchive` を使用
- `directory` を使用して、コントローラからターゲットインスタンスにディレクトリを `r` 同期します `synchronize` を使用

この例では、上記のアーティファクトタイプごとに 1 つのエントリーを示していますが、これらのパラメーター、または対応する Ansible モジュールが受け入れるその他のパラメーターを使用できます。

ファイルとディレクトリのターゲットインスタンスへのコピーは、十分な一般的なニーズであるため、この機能は *TPA hooks* を書き込む代わりに使用できる便利なショートカットを提供します。

ssh_key_file

デフォルトでは、tpaexec provision は ssh-keygen を使用して、クラスターの新しい SSH キーペアを生成します。クラスターディレクトリ内の `id_cluster_name` および `id_cluster_name.pub` という名前のファイルに。

代わりに既存のキーを使用する場合は、`config.yml` のトップレベルで `ssh_key_file` を SSH プライベートキーファイルの場所に設定できます。対応する公開キーは、同じ場所にある `.pub` の拡張子で利用できる必要があります。

```
ssh_key_file: ~/.ssh/id_rsa
```

このファイルがまだ存在しない場合、プロビジョニング中に `ssh-keygen` によって作成されます。

Managing SSH host keys

TPA は、クラスターのプロビジョニング中に SSH ホストキーのセットを生成します。これらのキーは、`hostkeys` サブディレクトリの下に `cluster` ディレクトリに保存されます。これらのホストキーは、AWS EC2 インスタンスと Docker コンテナの `/etc/ssh` に自動的にインストールされます。

デフォルトでは、これらのホストキーは *bare(-metal servers)* にインストールされませんが、`manage_ssh_hostkeys` を設定して有効にすることができます。

```
instances:

- Name: one
  ...
  platform: bare
  vars:
    manage_ssh_hostkeys: yes
```

bare(-metal servers) のドキュメントの説明に従って、正しいエントリを使用してクラスターディレクトリに `known_hosts` を最初にセットアップする必要があります。TPA は、展開中にホストキーを置き換えます。

`manage_ssh_hostkeys` 設定は、ベアインスタンスの場合にのみ意味があります。生成されたホストキーは、他のすべてのインスタンスにインストールされます。

known_hosts

TPA は、すべてのホストとそのパブリックホストキーのエントリをクラスター内のすべてのインスタンスのグローバル `ssh_known_hosts` ファイルに追加するため、`manage_ssh_hostkeys` 設定があるかどうかに関係なく、ホストキー検証プロンプトなしで相互に ssh できます。

Postgres source installation

`postgres_installation_method` を `src` に設定している場合、TPA はソースから Postgres をコンパイルおよびインストールします。この機能は、開発とテストで使用することを目的としており、同一構成のクラスター内でパッケージビルドとソースビルドを切り替えることができます。

ここでも、デフォルトを変更する必要はありません。これにより、デバッグが有効になった作業用クラスターが得られます。

Git リポジトリ

デフォルト設定では、`postgres_version` に対応する `REL_xx_STABLE` ブランチを使用して、コミュニティ Git リポジトリから Postgres をビルドおよびインストールします。次のように、別のリポジトリまたはブランチ有効な git リファレンスを指定できます。

```
cluster_vars:
  postgres_git_url: git://git.postgresql.org/git/postgresql.git
  postgres_git_ref: REL_12_STABLE
```

デフォルトの `git.postgresql.org` リポジトリは認証を必要としませんが、必要に応じて、*Git credentials* を使用できます

他のリポジトリに対して認証します。

リポジトリは `postgres_src_dir` デフォルト `/opt/postgres/src/postgres` にクローン作成されるか、ディレクトリが既に存在する場合たとえば、再展開する場合は `git pull` で更新されます。

ビルドのカスタマイズ

デフォルトでは、TPA はデバッグを有効にし、`postgres_build_dir` デフォルト `/opt/postgres/build/postgres` の賢明なデフォルトで Postgres を構成およびビルドします。さまざまな設定を変更して、ビルドをカスタマイズできます。

```
cluster_vars:
  postgres_extra_configure_env:
    CFLAGS: "-O3"
  postgres_extra_configure_opts:
    - --with-llvm
    - --disable-tap-tests
```

これは、`postgres_extra_configure_opts` のオプションと環境で定義された `postgres_extra_configure_env` の設定を使用して `./configure` を実行します。一部のオプションはデフォルトで指定されます `--with-debug` ですが、対応する `--disable-xxx` または `--without-xxx` オプションで無効にできます。 `--without-openssl` のビルドはサポートされていません。

必要に応じて、次のデフォルトのビルドコマンドを変更することもできます。

```
cluster_vars:
  postgres_make_command: "make -s"
  postgres_build_targets:
    - "all"
    - "-C contrib all"
  postgres_install_targets:
    - "install"
    - "-C contrib install"
```

`tpaexec deploy ... --skip-tags build-clean` を実行して、展開を繰り返すときにビルドディレクトリを再利用します。それ以外の場合、古いビルドディレクトリはビルドを開始する前に空にされます。[ローカルソースディレクトリ](#) を構成することもできます。

開発ビルドを高速化します。

ソースビルドを実行するたびに、Postgres が再起動されます。

追加コンポーネント

パッケージから Postgres をインストールする場合でも、ソースから拡張機能をコンパイルおよびインストールできます。[ソースからのビルドとインストール](#) の構成方法に関する別のページがあります。

ただし、拡張機能パッケージは通常、インストールされる Postgres パッケージに依存するため、ソースから Postgres をインストールする場合、ソースから拡張機能もインストールする必要があります。

パッケージのインストール

Installing Postgres-related packages に関する別のページがあります

`postgres_installation_method:` `pkg` デフォルト

Installing packages

TPA は、展開の早い段階で非 Postgres 関連パッケージのバッチをインストールし、次にすべての Postgres 関連パッケージと一緒にインストールし、次にオプションコンポーネントのパッケージを個別にインストールします。このページは、Postgres パッケージに依存していない `sysstat` や `strace` のようなパッケージのインストールに関するものです。

`config.yml` の `cluster_vars` または特定のインスタンスの `vars` の下の `packages` にエントリを追加できます。

```
cluster_vars:
  packages:
    common:
      - pkg1
      - pkg2
    Debian:
      - debpkg1
    RedHat:
      - rhpkg1
      - rhpkg2
    Ubuntu:
      - ubpkg1
    SLES:
      - slespkg1
```

上記の例では、TPA は独自の `default_packages` のリストとすべてのインスタンスの `packages.common` の下にリストされているパッケージをインストールし、インスタンスが実行されているディストリビュー

ションに基づいて、残りのディストリビューション固有パッケージをインストールします。これらのパッケージのいずれかが使用できない場合、展開は失敗します。

Postgres に依存するパッケージはリストしないでください。 *Installing Postgres-related packages* を使用代わりに。

オプションパッケージ

インストールする `optional_packages` のリストを指定できます。これらは利用可能な場合はインストールされ、それ以外の場合は無視されます。他の設定と同様に、`common` エントリはすべてのインスタンスに適用されますが、他のリストは、関連するディストリビューションを実行しているインスタンスにのみ適用されます。

`optional_packages:`

`common:`

- pkg1
- pkg2

`Debian:`

- debpkg4

パッケージの削除

インストールされている場合に削除する `unwanted_packages` のリストを指定できます。

`unwanted_packages:`

`common:`

- badpkg1

`Ubuntu:`

- badpkg2

Running initdb

TPA は、最初に `postgres_data_dir` が存在しない場合は作成し、正しい所有権、権限、および SELinux コンテキストがあることを確認します。次に、ディレクトリに `VERSION` ファイルがまだ含まれていない限り、`initdb` を実行して `postgres_data_dir` を初期化します。

pre-initdb hook を使用できます

`postgres_data_dir` が作成され、`initdb` が実行される前にタスクを実行します。フックが `postgres_data_dir` を初期化すると、TPA は `VERSION` ファイルを見つけ、`initdb` 自分自身を実行する必要がないことを認識します。

オプションで `postgres_initdb_opts` を `initdb` に渡すオプションのリストに設定できます。

`cluster_vars:`

`postgres_locale:` de_DE.UTF-8

`postgres_initdb_opts:`

- --data-checksums

常に`--data-checksums` オプションデフォルトで含まれるを含めることをお勧めします。

TPA は、`initdb` の実行時に、環境に `TZ=UTC` を設定し、`LC_ALL` を指定した `postgres_locale` に設定します。

別の構成ディレクトリ

デフォルトでは、`postgres_conf_dir` は `postgres_data_dir` と等しく、Postgres 構成ファイル `postgresql.conf`、`pg_ident.conf`、`pg_hba.conf`、および `conf.d` のインクルードファイルはデータディレクトリ内に作成されます。`postgres_conf_dir` を変更すると、TPA は `initdb` を実行した後、生成された構成ファイルを新しい場所に移動します。

Installing Postgres-related packages

TPA は、展開の早い段階で非 Postgres 関連パッケージのバッチをインストールし、次にすべての Postgres 関連パッケージと一緒にインストールし、次にオプションコンポーネントのパッケージを個別にインストールします。このページは、Postgres 自分自身に依存する `pglogical` のようなパッケージのインストールに関するものです。

PostgresPostgis に依存する追加パッケージをインストールするには、`config.yml` の `cluster_vars` の `extra_postgres_packages` または特定のインスタンスの `vars` の下にリストします。

```
cluster_vars:
  extra_postgres_packages:
    common:
      - postgres-pkg1
      - postgres-pkg2
    Debian:
      - postgres-deb-pkg1
    RedHat:
      - postgres11-rhpkg1
      - postgres11-rhpkg2
    Ubuntu:
      - ubpkg1
    SLES:
      - slespkg1
```

`packages.common` の下にリストされているパッケージは、Postgres パッケージのデフォルトリスト、および指定したディストリビューション固有パッケージとともに、すべてのインスタンスにインストールされます。

[Postgres source installation](#) に関する別のページがあります。

SSL Certificates

`config.yml` で `enable_pg_backup_api: true` を設定するか、`configure` 中に`--enable-pg-backup-api` コマンドラインオプションを使用すると、`barman` ロールを持つインスタンスは `pg-backup-api` をインストールし、クライアント証明書認証用に Apache プロキシを設定します。この Apache プロキシは、クラスター用に生成された SSL CA を使用して、サーバーとクライアントの証明書を生成します。

```
cluster_vars:
  enable_pg_backup_api: true
```

PG Backup API パッケージバージョン

デフォルトでは、TPA は pg-backup-api の利用可能な最新バージョンをインストールします。

config.yml ファイルの cluster_vars セクションの下での pg_backup_api_package_version: xxx を含めることにより、インストールされる pg-backup-api パッケージのバージョンを指定できます。

```
cluster_vars:
  ...
  pg_backup_api_package_version: 2.0.0-1.el8
  ...
```

apt または yum が受け入れるバージョン指定子を使用できます。

バージョンが一致しない場合は、* ワイルドカードを追加してみてください。これは、パッケージバージョンに 2:... のようなエポック修飾子がある場合に必要になることがよくあります。

pg-backup-api は、デフォルトでパッケージを介してインストールされますが、git ブランチまたはローカルディレクトリからインストールすることもできます。詳細については、[Building from source](#) および [ソースからのビルドとインストール](#) を参照してください。

pg-backup-api を実行している barman ノードで pg-backup-api status を実行します。「OK」が返された場合、pg-backup-api サービスが実行されています。

プロキシが動作していることをテストするには、次のコマンドを実行します。

```
curl --cert /etc/tpa/pg-backup-api/pg-backup-user.crt \
      --key /etc/tpa/pg-backup-api/pg-backup-user.key \
      -X GET https://{hostname}/diagnose
```

機能している場合は、大規模な json 出力を取得します。これを barman diagnose の出力と比較できます。それらは正確に一致するはずですが。

ルート証明書は、デフォルトで/etc/tpa/pg-backup-api/ にコピーされます。

クライアント証明書とキー pg-backup-user.crt および pg-backup-user.key は、テストのために生成されます tpaexec test を介してまたは barman ホストからのコマンドライン。tpaexec test を参照してください。

Apache プロキシサーバーの証明書とキー pg-backup-api.crt および pg-backup-api.key も生成されます

API を照会する必要がある各サービスは、独自のクライアント証明書を個別に生成する必要があります。たとえば、PEM エージェントロールは、--enable-pem および --enable-pg-backup-api または config.yml 同等の両方が使用される場合、セットアップ中にクライアント証明書を生成します。

Setting sysctl values

デフォルトでは、TPA はターゲットインスタンスにさまざまな sysctl 値を設定し、それらを `/etc/sysctl.conf` に含めて、リブート後も持続するようにします。

オプションで `sysctl_values` に独自の値を指定できます。

```
cluster_vars:
  sysctl_values:
    kernel.core_pattern: core.%e.%p.%t
    vm.dirty_bytes: 4294967296
    vm.zone_reclaim_mode: 0
```

指定した値は、その変数の TPA のデフォルト値がある場合より優先されます。設定は最初に `sysctl.conf` 行ごとに追加され、最後に `sysctl -p` でロードされます。

Docker および lxd インスタンスは sysctl の設定をサポートしていないため、TPA はこれらのプラットフォームでこの手順を完全にスキップします。

7.5.5 Postgres Configuration

Creating Postgres databases

デプロイ中に Postgres データベースを作成するには、`config.yml` の `cluster_vars` または特定のインスタンスの `vars` の下の `postgres_databases` のリストにエントリを追加します。

```
cluster_vars:
  postgres_databases:
    - name: exampledb

    - name: complexdb
      owner: example
      encoding: UTF8
      lc_collate: de_DE.UTF-8
      lc_ctype: de_DE.UTF-8
      template: template0
      extensions:
        - name: hstore
        - name: dblink
      languages:
        - name: plperl
        - name: plpython
      tablespace: exampl.tablespace
```

上記の例では、2つのデータベースを作成します `bdr_database` など、TPA 自分自身が作成することを決定したデータベースとは別に。

各エントリは、作成するデータベースの `name` を指定する必要があります。他のすべての属性はオプション。

`owner` はデフォルトで `postgres` ですが、任意の有効なユーザー名に設定できます [Creating Postgres users](#) のユーザー

この時点で作成されています)。

`encoding`、`lc_collate`、および `lc_ctype` 値は、デフォルトで `initdb` の実行時に設定された `postgres_locale` になりますデフォルトは、ターゲットシステムの `LC_ALL` または `LANG` 設定を使用します。デフォルト以外のロケール設定でデータベースを作成する場合、`template: template0` も指定する必要があります。

オプションで、データベースのデフォルトの `tablespace` を指定できます。表領域は既に存在する必要があります [Creating Postgres tablespaces](#) を参照してください。

`extensions` および `languages` のオプションのリストを指定して、各データベース内で作成することができますテンプレートデータベースから継承した拡張機能または言語に加えて。必要なパッケージは、たとえば [Installing Postgres-related packages](#) に含めることにより、既にインストールされている必要があります。

TPA は、`postgres_databases` で言及されていない既存のデータベースをドロップせず、必要に応じて追加のデータベースを作成する場合があります BDR など。

Creating Postgres tablespaces

デプロイメント中に Postgres テーブルスペースを作成するには、`config.yml` の `cluster_vars` または特定のインスタンスの `vars` の下の `postgres_tablespaces` に名前と場所を定義します。

`volume_for: postgres_tablespace` セットと `tablespace_name` を定義して [Filesystem configuration](#) を実行すると、それらはデフォルトエントリとして `postgres_tablespaces` に追加されます。

```
cluster_vars:
  postgres_tablespaces:
    explicit:
      location: /some/path

instances:
- Name: example
  ...
  volumes:
  - device_name: /dev/xvdh
    ...
  vars:
    volume_for: postgres_tablespace
    tablespace_name: implicit
```

上記の例では、2つのテーブルスペースを作成します。明示的 `/some/path` と暗黙的 ボリュームの別のマウントポイントを指定しない限り、デフォルトで `/opt/postgres/tablespaces/implicit/tablespace_data` に。

すべての `postgres_tablespace` ボリュームには `tablespace_name` を定義する必要があります。テーブルスペースの場所は、ボリュームのマウントポイントから取得されます。

`postgres_tablespace` のすべてのエントリは、表領域名をキーとしてとその `location` を指定する必要があります。テーブルスペースの場所を明示的に指定する場合は、PGDATA 内にテーブルスペースを置かず、ボリュームマウントポイントをテーブルスペースの場所として直接使用しないでください。lost+found は、一部のツールを混乱させて、ディレクトリが空でないと判断します。

デフォルトでは、テーブルスペース owner は `postgres` ですが、任意の有効なユーザー名に設定できます *Creating Postgres users* のユーザー

(この時点で作成されています)。

ストリーミングレプリカには、上流のインスタンスと同じ `postgres_tablespace` ボリュームと `postgres_tablespace` 設定が必要です

Creating Postgres databases でデータベースのデフォルトの表領域を設定できます。

postgresql.conf

TPA は、その下にさまざまな `.conf` ファイルを含む `conf.d` ディレクトリを作成し、メイン `postgresql.conf` の `include_dir` を使用してこれらの追加構成ファイルを使用します。

Postgres 構成ファイル `postgresql.conf`、`pg_ident.conf`、および `pg_hba.conf` および `conf.d` の下のインクルードファイルは、常に `postgres_conf_dir` に保存されます。これはデフォルトで `postgres_data_dir` と同じですが、構成をデータディレクトリとは別にしておきたい場合は、別の場所に設定できます。

主な構成メカニズムは、変数を直接設定することです。

```
cluster_vars:
  temp_buffers: 16MB
  log_connections: on
  autovacuum_vacuum_cost_limit: -1
  effective_cache_size: 4GB
  max_connections: 300
  max_wal_senders: 32
```

TPA は、構成を複数のファイルに分割します。2 つのメインファイルは、`0000-tpa.conf` および `0001-tpa_restart.conf` です。これらには、それぞれサーバーのリロードまたは再起動を必要とする設定が含まれています。展開中に、TPA は正しいファイルに変更を書き込み、必要に応じて Postgres をリロードまたは再起動します。

TPA は、特定の状況オプションの拡張機能を構成する場合などに他のファイルを使用する場合がありますが、通常、特定のパラメーターが正確にどこに設定されているかを気にする必要はありません。

次回 `tpaexec deploy` を実行したときに変更が上書きされる可能性があるため、`conf.d` の下のファイルは決して編集しないでください。

postgres_conf_settings

TPA は、多くの使用可能な `postgresql.conf` 設定すべてではありませんが、直接設定できる `temp_buffers` や `maintenance_work_mem` のような変数を提供します。

`postgres_conf_settings` を使用して、TPA で認識されるかどうかにかかわらず、パラメーターを設定できます。`postgresql.conf` に表示されるのとまったく同じように値を引用符で囲む必要があります。

```
cluster_vars:
  effective_cache_size: 2GB
  postgres_conf_settings:
    effective_cache_size: 4GB
    authentication_timeout: 1min
    synchronous_standby_names: >-
      any 2 ("first", "second", "third")
    bdr.global_lock_statement_timeout: 60s
```

これは、TPA がネイティブに認識しない設定で最も役立ちますが、任意のパラメーターに使用できますたとえば、`effective_cache_size` は変数として設定できますが、`authentication_timeout` はできません。

これらの設定は `conf.d/9900-role-settings.conf` に書き込まれるため、他の方法で設定された変数より優先されます。

`postgres_conf_settings` の下の値を変更した場合、TPA は、リロードが変更を有効にするのに十分であるかどうか、または再起動が必要かどうかを知る方法がありません。したがって、常にサーバーを再起動して変更をアクティブにします。これが、可能な場合は常に変数を直接使用することをお勧めする理由です。

shared_buffers

デフォルトでは、TPA は `shared_buffers` を使用可能なメモリの 25% に設定しますこれは単なる経験則であり、推奨ではありません。別の比率を使用するように `shared_buffers_ratio: 0.35` を設定するか、`shared_buffers_mb: 796` を特定の MB 数に設定するか、`shared_buffers: "2GB"` などの正確な値を直接指定することにより、このデフォルトをオーバーライドできます。

effect_cache_size

デフォルトでは、TPA は `effective_cache_size` を使用可能なメモリの 50% に設定します。別の比率を使用するように `effective_cache_size_ratio: 0.35` を設定するか、`effective_cache_size_mb: 796` を特定の MB 数に設定するか、`effective_cache_size: "8GB"` などの正確な値を直接指定することにより、このデフォルトをオーバーライドできます。

shared_preload_libraries

TPA は、動作するために `shared_preload_libraries` のエントリを必要とする拡張機能の内部リストを維持しており、このような拡張機能を `postgres_extensions` に含めると、`shared_preload_libraries` が自動的に更新されます。

プリロードが必要な認識されない拡張機能を使用している場合は、`preload_extensions` に追加できます。

```
cluster_vars:
  preload_extensions:
    - myext
    - otherext
```

`myext` を `postgres_extensions` に追加すると、`shared_preload_libraries` には `myext` が含まれます。

デフォルトでは、`conf.d/8888-shared_preload_libraries.conf` に `shared_preload_libraries` が設定されます。

`shared_preload_libraries` を変数として直接設定することはサポートされていません。通常は設定する必要はありませんが、やむを得ない場合は、`postgres_conf_settings` の下に完全に引用符で囲まれた値を設定できます。この場合、値は `conf.d/9900-tpa-postgres-conf-settings.conf` に設定されます。

Postgres ログ

デフォルトのログファイルは `/var/log/postgres/postgres.log` として定義されます。それを変更する必要がある場合は、`config.yml` で `postgres_log_file` を設定できます。

```
cluster_vars:
  [...]
  postgres_log_file: /srv/fantastic_logs/pg_server.log
```

TPA はディレクトリの作成を担当し、必要に応じてログをローテーションします。

SSL 構成

デフォルトでは、TPA はプライベートキーと自己署名 TLS 証明書を生成します。これらは Postgres がそれぞれ `ssl_key_file` および `ssl_cert_file` として使用します。ファイルは TPA クラスター名 `cluster_name.key` および `cluster_name.crt` を使用して名前付けられ、`/etc/tpa` にあります。その結果、`0001-tpa-restart.conf` では次のデフォルト構成になります。

```
ssl_key_file=/etc/tpa/cluster_name.key
ssl_cert_file=/etc/tpa/cluster_name.crt
```

これは、クライアントとサーバー間のトラフィックが転送中に暗号化されることを保証には十分です。

独自の証明書を提供するには、証明書を *Uploading artifacts* としてターゲットノードにアップロードし、次のクラスター変数を指定してパスを設定します。

```
cluster_vars:
  ...
  artifacts:
    - type: file
      dest: /path/to/your_key.key
      src: /local/path/to/your_key.key
      owner: root
      group: root
      mode: "0644"
    - type: file
      dest: /path/to/your_cert.crt
      src: /local/path/to/your_cert.crt
      owner: root
      group: root
      mode: "0600"
  ssl_key_file: /path/to/your_key.key
  ssl_cert_file: /path/to/your_cert.crt
```

または、キーと証明書をデフォルトの場所にアップロードすると、TPA は独自の生成の代わりにそれらを使用し、`ssl_key_file` または `ssl_cert_file` を指定する必要はありません。ただし、`/etc/tpa` は、アーティファクトのアップロード時に存在しないため、明示的に作成する必要があることに注意してください。これらのファイルの権限と所有権は、展開中に `postgres` ユーザーが作成されるときに TPA によって調整されます。

```
cluster_vars:
  ...
  artifacts:
  - type: path
    path: /etc/tpa
    state: directory
    owner: root
    group: root
    mode: "0755"
  - type: file
    dest: /etc/tpa/cluster_name.key
    src: /local/path/to/your_key.key
    owner: root
    group: root
    mode: "0644"
  - type: file
    dest: /etc/tpa/cluster_name.crt
    src: /local/path/to/your_cert.crt
    owner: root
    group: root
    mode: "0600"
```

注釈

その他の SSL 設定 TPA は、デフォルトで `ssl_ca_file` または `ssl_crl_file` を指定しません。これらのファイルを自分自身で提供するには、*Uploading artifacts* を使用し、同じ名前のクラスター変数を指定します。

手動で変更を加える

TPA で生成された構成をオーバーライドするには2つの方法があります。

最初のおよび推奨されるオプションは、`ALTER SYSTEM` を使用することです。これは、常に構成ファイルのすべてより優先されます。

```
# ALTER SYSTEM SET bdr.global_lock_statement_timeout TO 60s;
```

`conf.d/9999-override.conf` を編集することもできます。

```
$ echo "bdr.global_lock_statement_timeout=60s" >> conf.d/9999-override.conf
```

`conf.d` の下の他のすべてのファイルは、展開中に構成が変更された場合に上書きされる可能性があります。TPA は、最初に空のファイルを作成した後、`9999-override.conf` を変更することはありません。

変更する設定によっては、`SELECT pg_reload_conf()` を実行するか、サーバーを再起動して変更を有効にする必要がある場合があります。

postgresql.conf をスクラッチから生成する

デフォルトでは、TPA は `include_dir` を追加する以外に、デフォルトの `initdb` 生成された `postgresql.conf` ファイルをそのまま残します。通常、この動作をオーバーライドする必要はありませんが、そうするように `postgres_conf_template` を設定できます。

```
cluster_vars:
  postgres_conf_template: pgconf.j2
```

これで、クラスターディレクトリの `templates/pgconf.j2` を使用して `postgresql.conf` が生成されます。

pg_hba.conf

Postgres ドキュメントでは、`pg_hba.conf` で使用可能なさまざまなオプションについて説明しています。

デフォルトでは、TPA はクラスター用に適切な `pg_hba.conf` を生成し、インスタンス間のレプリケーション、および認証されたクライアントからの接続を許可します。

`postgres_hba_settings` のリストを提供することにより、デフォルト構成にエントリを追加できます。

```
cluster_vars:
  postgres_hba_settings:
    - "# let authenticated users connect from anywhere"
    - hostssl all all 0.0.0.0/0 scram-sha-256
```

`postgres_hba_local_auth_method: md5` を設定することにより、`pg_hba.conf` のデフォルトの `local all all peer` 行をオーバーライドできます。

デフォルトのエントリが不要な場合は、`postgres_hba_template` を変更できます。

```
cluster_vars:
  postgres_hba_template: pg_hba.lines.j2
  postgres_hba_settings:
    - "# my lines of text"
    - "# and nothing but my lines"
    - "# ...not even any clients!"
    - hostssl all all 0.0.0.0/0 reject
```

クラスターディレクトリに `templates/my_hba.conf.j2` を作成し、次のように設定することもできます。

```
cluster_vars:
  postgres_hba_template: my_hba.conf.j2
```

クラスターディレクトリの `templates` サブディレクトリの外部にテンプレートファイルを配置する場合、ファイルへの絶対パスを指定してください。

```
# in the root of the cluster directory
```

```
cluster_vars:
```

```
  postgres_hba_template: "{{ cluster_dir }}/my_hba.conf.j2"
```

```
# in a subdirectory of the cluster directory that is NOT templates
```

```
cluster_vars:
```

```
  postgres_hba_template: "{{ cluster_dir }}/subdirectory/my_hba.conf.j2"
```

```
# in a directory outside of the cluster directory
```

```
cluster_vars:
```

```
  postgres_hba_template: /path/to/file/outside/cluster_dir/my_hba.conf.j2
```

既存の `pg_hba.conf` だけを残したい場合は、それもできます。

```
cluster_vars:
```

```
  postgres_hba_template:
```

さまざまなインスタンスで異なるように `pg_hba.conf` を構成することは可能ですが、スイッチオーバーやフェイルオーバーなどのトポロジを変更するイベント後のアクセスとレプリケーションの問題を回避するために、一般的に均一な構成を推奨します。

pg_ident.conf

通常は `pg_ident.conf` を変更する必要はありません。デフォルトでは、TPA はそれを変更しません。

`pg_ident.conf` を好きなコンテンツに置き換えるように `postgres_ident_template` を設定できます。

```
cluster_vars:
```

```
  pg_ident_template: ident.j2
```

クラスターディレクトリに `templates/ident.j2` も作成する必要があります。

```
{% for u in [unixuser1, unixuser2] %}
mymap {{ u }} dbusername
{% endfor %}
```

Configuring .pgpass

TPA は、クラスターインスタンス間で使用するために、デフォルトで `~postgres/.pgpass` を作成し、その中の `postgres_user postgres_flavour` に応じてデフォルトで `postgres` または `entrepisedb` のパスワードを使用します。

`pgpass_users` を設定して、ユーザーの別のリストのエントリを作成できます。`pgpass_users` リストはデフォルト値をオーバーライドするため、`postgres_user postgres /enterprisedb` は、`pgpass_users` リストに明示的に含めない限り含まれないことに注意してください。

フックスクリプトから `postgres/pgpass` ロールを含めて、独自の `.pgpass` ファイルを作成することもできます。

```
- include_role: name=postgres/pgpass
  vars:
    pgpassfile: ~otheruser/.pgpass
    pgpass_owner: otheruser
    pgpass_group: somegroup
    pgpass_users:
      - xyzuser
      - pqruser
```

The postgres Unix user

このページには、`postgres` ユーザーとそのホームディレクトリがどのように構成されているかについて文書化されています。

Creating Postgres users の作成方法に関する別のページがあります。

シェル構成

TPA は `.bashrc` ファイルをインストールし、それが `.profile` または `.bash_profile` ファイルにも含まれることを確認します。

ユーザー名、ホスト名、作業ディレクトリを含むプロンプトを設定し、`postgres_bin_dir` が `PATH` にあることを確認し、`PGDATA` を `postgres_data_dir` の場所に設定します。

オプションで `extra_bashrc_lines` を指定して、`.bashrc` に任意の行を追加できます。YAML 複数行文字列構文 `>-` を使用して、シェルメタキャラクタの引用符とエスケープについて心配する必要を回避します。

```
cluster_vars:
  extra_bashrc_lines:
    - alias la=ls\ -la
    - >-
      export PATH="$PATH":/some/other/dir
```

`sudoers` を編集して `sudo systemctl start/stop/reload/restart/status postgres` を許可し、また `ulimits` を変更して無制限のコアダンプを許可し、ファイル記述子の制限を引き上げます。

SSH キー

TPA は `ssh-keygen` を使用して `postgres` ユーザーの SSH キーペアを生成およびインストールし、クラスター内のインスタンスが `postgres` として互いに `ssh` できるように `.ssh/authorized_keys` を編集します。

TLS 証明書

デフォルトでは、TPA はプライベートキーと自己署名 TLS 証明書を生成します。これらは Postgres がそれぞれ `ssl_key_file` および `ssl_cert_file` として使用します。ファイルは、TPA クラスター名 `cluster_name.key` および `cluster_name.crt` を使用して名前付けられ、`/etc/tpa` にあります。独自のキーと証明書を提供する方法を含む詳細については、[SSL 構成](#) のドキュメントを参照してください。

自己署名 TLS キーのサイズは、変数 `postgres_rsa_key_size` を `cluster_vars` セクションに追加して変更できます。

```
(...)
cluster_vars:
  postgres_rsa_key_size: 4096
```

ユーザー名

`postgres_user` および `postgres_group` 設定デフォルトでは両方とも `postgres` は、どこでも一貫して使用されます。何らかの理由で Postgres を別のユーザーとして実行する必要がある場合、それらを変更できます。

Creating Postgres users

デプロイ中に Postgres ユーザーを作成するには、`config.yml` の `cluster_vars` または特定のインスタンスの `vars` の下の `postgres_users` のリストにエントリを追加します。

```
cluster_vars:
  postgres_users:
    - username: example

    - username: otheruser
      generate_password: true
      role_attrs:
        - superuser
        - replication
      granted_roles:
        - r1
        - r2
```

上記の例では、2つのユーザー `repmgr` や `barman` など、TPA 自分自身が作成することにしたユーザーを除きます。

各エントリは、作成する `username` を指定する必要があります。

`granted_roles` リストのロールは、新しく作成したユーザーに付与されます。

`role_attrs` リストには、特定の [CREATE ROLE options](#) が含まれる場合があります

`[NO]SUPERUSER`、`[NO]CREATEDB`、`[NO]LOGIN` ユーザーまたはロールを作成するためなど。

パスワードの生成

デフォルトでは、TPA はユーザーのランダムなパスワードを生成し、クラスターのインベントリの `<username>_password` という名前のボールトで暗号化された変数に保存します。後で値を取得できます。

```
$ tpaexec show-password ~/clusters/speedy example
beePh~iez6lie4thi5KaiG%eghaeT]ai
```

`config.yml` でパスワードを明示的に指定することはできませんが、代わりに別の `<username>_password` をインベントリに保存できます。

```
$ tpaexec store-password ~/clusters/speedy example --random
$ tpaexec show-password ~/clusters/speedy example
)>tkc}}kly4&epaJ?;NJ:luT{C7D*<p
$ tpaexec store-password ~/clusters/speedy example
Password:
$ tpaexec show-password ~/clusters/speedy example
terrible insecure password
$
```

ユーザーにパスワードをまったく与えたくない場合は、`generate_password: false` を設定できます。

7.5.6 Commands

tpaexec archive-logs

ログディレクトリを作成し、インスタンスからログをアーカイブするには、次のコマンドを実行します。

```
tpaexec archive-logs <cluster-dir>
```

これにより、クラスターディレクトリに `logs/YYYYMMDDHHMMss/` ディレクトリが作成され、クラスター内の各インスタンスの `/var/log` の下のすべてのファイルの `tar.gz` アーカイブが別のディレクトリにダウンロードされます。

前提条件

既存のクラスターがある場合は、`tpaexec archive-logs` をすぐに実行できます。ただし、新しいクラスターを構成する場合は、少なくともクラスターを *tpaexec provision* する必要があります。クラスターを *tpaexec deploy* する場合、より多くのログを取得します。

クイックスタート

```
[tpa]$ tpaexec archive-logs ~/clusters/speedy
```

```
PLAY [Prepare local host archive] *****
->*****
```

(次のページに続く)

(前のページからの続き)

```

TASK [Collect facts] *****
  ↳ *****
ok: [localhost]

TASK [Set time stamp] *****
  ↳ *****
ok: [localhost]

TASK [Create local log archive directory] *****
  ↳ *****
changed: [localhost]

PLAY [Archive log files from target instances] *****
  ↳ *****

...

TASK [Remove remote archives] *****
  ↳ *****
changed: [kinship]
changed: [khaki]
changed: [uncivil]
changed: [urchin]

PLAY RECAP *****
  ↳ *****
khaki           : ok=3    changed=3    unreachable=0    failed=0
kinship         : ok=3    changed=3    unreachable=0    failed=0
localhost       : ok=3    changed=1    unreachable=0    failed=0
uncivil         : ok=3    changed=3    unreachable=0    failed=0
urchin          : ok=3    changed=3    unreachable=0    failed=0

```

より詳細な出力が必要な場合は、コマンドに `-v`、`-vv` などを追加できます。

生成されたファイル

クラスターディレクトリの下で、各インスタンスのログを見つけることができます。

```

~/clusters/speedy/logs/
`-- 220220306T185049
    |-- khaki-logs-20220306T185049.tar.gz
    |-- kinship-logs-20220306T185049.tar.gz
    |-- uncivil-logs-20220306T185049.tar.gz
    `-- urchin-logs-20220306T185049.tar.gz

```

アーカイブコンテンツの例

```
khaki-logs
|-- anaconda
|   |-- anaconda.log
|   |-- dbus.log
|   |-- dnf.librepo.log
|   |-- hawkey.log
|   |-- journal.log
|   |-- ks-script-ipdkisn0.log
|   |-- ks-script-jr03uzns.log
|   |-- ks-script-mh2iidvh.log
|   |-- lvm.log
|   |-- packaging.log
|   |-- program.log
|   |-- storage.log
|   |-- syslog
|   |-- X.log
|-- bttmp
|-- dnf.librepo.log
|-- dnf.log
|-- dnf.rpm.log
|-- hawkey.log
|-- lastlog
|-- private
|-- tpaexec.log
|-- wtmp
```

tpaexec download-packages

ダウンローダーの目的は、既存の構成から TPA クラスターの完全なインストールを実行するために必要なパッケージを提供することです。これは、インターネットアクセスのない安全なクラスターにパッケージを出荷する場合、またはテストクラスター用にパッケージを繰り返しダウンロードすることを避けたい場合に役立ちます。

ダウンローダーは、必要なパッケージの完全な依存関係ツリーをダウンロードし、結果のパッケージリポジトリには、ターゲットディストリビューションパッケージマネージャーのメタデータファイルが含まれるため、クラスターを構築するためにのみ使用できます。現時点では、パッケージマネージャー Apt および YUM がサポートされています。

注釈

ダウンロードパッケージ機能を使用するには、TPA ホストに Docker をインストールする必要があります。これは、ダウンローダーがターゲットオペレーティングシステムのコンテナを作成することにより動作し、そのシステムのパッケージマネージャーを使用して依存関係を解決し、必要なパッケージをすべてダウンロードするためです。ダウンロードパッケージに必要な Docker セットアップは、[Docker のインストール](#) の場合と同じです。

使用法

既存のクラスター構成が存在する必要があります。これは、`tpaexec configure` コマンドを使用して実現できます。ダウンローダーを使用するために特定のオプションは必要ありません。[クラスター構成](#) を参照

。

`download-packages` サブコマンドを実行して、ダウンロードプロセスを開始します。ダウンローダーが使用する OS と OS バージョンを提供します。

```
tpaexec download-packages cluster-dir --os RedHat --os-version 8
```

これは、特定の docker イメージとして表現することもできます。次の例のように、接頭辞がついた tpa イメージのいずれかを使用することを強くお勧めします。

```
tpaexec download-packages cluster-dir --docker-image tpa/redhat:8
```

ダウンローダーは、ダウンロードしたファイルをデフォルトで `local-repo` ディレクトリに配置します。オプション `--download-dir path` を使用して、別のディレクトリにダウンロードすることができます。

結果の使用

`local-repo` ディレクトリの内容には、docker イメージに含まれる OS に従って、ansible によって決定された構造が設定されます。たとえば、docker イメージ `tpa/redhat:8` には次のものが含まれます。

```
cluster-dir/
`-- local-repo
    |-- RedHat
        |-- 8
            |-- *.rpm
            |-- repodata
                |-- *repodata-files*
```

これをクラスターでそのまま使用するか、ターゲット制御ノードにコピーできます。[Managing clusters in a disconnected or air-gapped environment](#) を参照してください。[Creating and using a local repository](#) は、TPA によって自動的に検出および使用されます。

失敗したダウンローダーコンテナのクリーンアップ

ダウンロードプロセス中にエラーが発生した場合、コマンドはデバッグを支援するために実行中のダウンローダーコンテナを残します。たとえば、障害が発生したダウンローダーコンテナにログインして、ログまたはネットワークを検査することができます。ダウンローダーコンテナの名前は、コンテナ名の許可制限 64 文字を超えない限り、通常 `$cluster_name-downloader` になります。`docker ps` を実行して実行中のコンテナをリストし、クラスター名と一致するコンテナ名を探すことにより、正確な名前を確認できます。ほとんどの場合、`docker exec -it $cluster_name-downloader /bin/bash` を実行することにより、実行中のコンテナにログインできます。検査後、`--tags cleanup` で `download-packages` コマンドを実行することにより、残ったコンテナをクリーンアップできます。例

```
tpaexec download-packages cluster-dir --docker-image tpa/redhat:8 --tags cleanup
```

TPA custom commands

TPA クラスター内のインスタンスで環境に固有のタスクを実行するカスタムコマンドを定義できます。

このメカニズムを使用して、クラスターに適用されるプロセスを自動化できます。これらのコマンドは、ビルトインのクラスター管理コマンドと同様に、クラスターディレクトリに対して呼び出すことができます。このようなプロセスを定義および実行する統一の方法があると、誤解やオペレーターエラー、または過去に正しかったが、それ以降現実から乖離したプロセスドキュメントによって引き起こされるエラーの可能性が減少します。

Ansible プレイブックを書くということは、任意に複雑なタスクを実装できることを意味します。カスタムコマンド規約に従うということは、config.yml と、TPA が実行するクラスター検出タスクに基づいて設定されるさまざまなファクトを利用でき、接続、認証、その他の基本機能などの詳細を考える必要がないことを意味します。

これにより、アドホックシェルスクリプトは可能ですが、通常はそうではない方法で、復元力の高いべき等コマンドを書くのがはるかに簡単になります。

クイックスタート

- クラスターディレクトリ内に `commands/mycmd.yml` を作成します
- `tpaexec mycmd /path/to/cluster` を実行します

例

次に、クラスター内のすべてのインスタンスで単一のコマンドを実行するコマンドの例を示します。ユースケースに応じて、さまざまなホストをターゲットとするコマンドたとえば、Postgres インスタンスでのみ実行する `hosts: role_postgres` を書くか、追加のタスクを実行して条件を評価して、何を行うかを正確に決定できます。

```
- --

# Always start with this

- import_playbook: "{{ tpa_dir }}/architectures/lib/init.yml"
  tags: always

- name: Perform custom command tasks
  hosts: all
  tasks:
    - name: Display last five lines of syslog
      command: tail -5 /var/log/syslog
```

(次のページに続く)

(前のページからの続き)

```
become_user: root
become: yes
```

TPA custom tests

環境とアプリケーションに固有の詳細なテストを簡単に定義して、TPA の *tpaexec test* を強化できます。

どんなに単純なタスクでも、クラスターで実行するテストを作成して、すべてが期待どおりに動作していることを確認することを強くお勧めします。このようなテストを実行する統一的で反復可能な方法があると、危機に対処する場合でも、日常的なクラスター管理を行う場合でも、重要な点を見逃すことはありません。

構成されたロールまたはその他のプロパティによってクラスターインスタンスをターゲットとするテストを記述する場合、該当するすべてのテストが適切なインスタンスで実行されることを保証できます。レプリケーションステータスを確認するレプリカの数、pgbouncer を実行しているサーバー、または手動で確認するときに間違いを招きやすいその他の詳細を調べたり覚えたりする必要はありません。

テストはクラスターに重要な変更を加えてはなりません。実稼働サーバーで実行することを考えない場合、おそらくテストすべきではありません。

クイックスタート

- クラスターディレクトリ内に `tests/mytest.yml` を作成します
- `tpaexec test /path/to/cluster mytest` を実行します

`tpaexec test` への `--include-tests-from /other/path` オプションを使用して、他の場所にテストを作成し、クラスター全体で使用することもできます。

使用方法については、`tpaexec help test` を実行してください。

例

次に、すべての Postgres インスタンスで実行されるテストを記述する方法を示します。`hosts: all` の代わりに `hosts: role_postgres` に注意してください。

任意の Ansible タスクを使用して、クラスターから情報を収集し、テストを実行できます。いくつかの期待が満たされない場合に失敗するタスクを書くだけです `assert`、`fail` … `when` など。

```
- --

- name: Perform my custom tests
  hosts: role_postgres
  tasks:

    # Always start with this
    - include_role:
        name: test
        tasks_from: prereqs.yml
```

(次のページに続く)

(前のページからの続き)

```

# Make sure that the PGDATA/PG_VERSION file exists. (This is just a
# simplified example, not something that actually needs testing.)
- name: Perform simple test
  command: "test -f {{ postgres_data_dir }}/PG_VERSION"
  become_user: "{{ postgres_user }}"
  become: yes

- name: Run pg_controldata
  command: >
    {{ postgres_bin_dir }}/pg_controldata {{ postgres_data_dir }}
  register: controldata
  become_user: "{{ postgres_user }}"
  become: yes

# Write output to clusterdir/$timestamp/$hostname/pg_controldata.txt
- name: Record pg_controldata output
  include_role:
    name: test
    tasks_from: output.yml
  vars:
    output_file: pg_controldata.txt
    content: |
      {{ controldata.stdout }}

```

上記のようにビルトイン `output.yml` を使用して、クラスターディレクトリのタイムスタンプ付きテストディレクトリに任意のテスト出力を記録できます。

各テストは、完全な Ansible プレイブックつまり、タスクのリストではなくプレイのリストである必要があります。これは、基本的な TPA セットアップタスクの後にインポートおよび実行されます。

破壊的テスト

デフォルトでは、テストはクラスターに重要な変更を加えないでください。レプリケーションをテストするテーブルを作成するようなことをする場合でも、自分自身の後のクリーンアップに注意する必要があります。

実稼働クラスターでは受け入れられないクラスターに変更を加えるテストは、**destructive** としてマークされなければなりません。これらは、開発時、または初期クラスターの「バーンイン」プロセス中にのみ実行するテストである場合があります。

テストに `prereqs.yml` を含める場合、**destructive: yes** を設定することにより、「破壊的」テストを定義できます。

```

- hosts: ...
  tasks:
    - include_role:
      name: test

```

(次のページに続く)

(前のページからの続き)

```
tasks_from: prereqs.yml
vars:
  destructive: yes
```

誰かが `tpaexec test /path/to/cluster mytest` を実行すると、`--destroy-this-cluster` オプションを使用して実行を確認するように求めるエラーが発生します。

`--destroy-this-cluster` の使用は、コマンドを実行するリスクを認識することを示します。テストが実際にクラスターを破壊することを保証するものではありません。

Configuring the beacon agent

TPA は、`config.yml` のロール `beacon-agent` を持つノードにビーコンエージェントをインストールおよび構成します。`--enable-beacon-agent` が `tpaexec configure` に渡される場合、クラスター内のすべての postgres ノードがこのロールになります。

Beacon エージェントパッケージバージョン

デフォルトでは、TPA は `beacon-agent` の利用可能な最新バージョンをインストールします。

`config.yml` ファイルの `cluster_vars` セクションの下に `beacon_agent_package_version: xxx` を含むことにより、インストールされる `beacon-agent` パッケージのバージョンを指定できます。

```
cluster_vars:
  ...
  beacon_agent_package_version: 1.56.2-1
  ...
```

`apt` または `yum` が受け入れるバージョン指定子を使用できます。

バージョンが一致しない場合は、* ワイルドカードを追加してみてください。これは、パッケージバージョンに `2:...` のようなエポック修飾子がある場合に必要になることがよくあります。

ビーコンエージェントの構成

ビーコンエージェント構成には、クラスターごとに設定する必要がある 2 つのパラメーター、アクセスキーとプロジェクト ID が含まれています。

アクセスキーはクラスターディレクトリで暗号化されたままで、`tpa` の `store-password` および `show-password` コマンドを使用して設定または読み取ることができます。

```
$ tpaexec store-password . beacon_agent_access_key
Password:
```

```
$ tpaexec show-password . beacon_agent_access_key
```

`tpaexec provision` の実行時に環境変数 `BEACON_AGENT_ACCESS_KEY` が設定されている場合、アクセスキーはその値から設定されます。

プロジェクト ID は、`cluster_vars` の `beacon_agent_project_id` キーの下に `config.yml` に保存されます。 `--beacon_agent_project_id` 引数が `tpaexec configure` に渡される場合、その値は `config.yml` に適切に書き込まれます。

ビーコンエージェントのインストール

TPA は EDB のリポジトリからビーコンエージェントをインストールし、`beacon` というオペレーティングシステムユーザーと `beacon` というデータベースユーザーを作成します。エージェントの構成ファイルは、ビーコンユーザーのホームディレクトリの `.beacon/beacon_agent.yml` に書き込まれます。

ビーコンエージェントの実行

TPA は、`systemd` サービスユニットファイルをインストールして、ブート時にエージェントを起動し、ビーコンユーザーとして実行します。

Compliance

TPA は、クラスターが STIG または CIS 標準に準拠することを簡単にするように設計された構成を生成できます。 `--compliance stig` または `--compliance cis` を `tpaexec configure` に渡すと、TPA は次のことを行います。

- 他のオプションが適切な規格と互換性があることを確認します。
- これが特定の標準に準拠するクラスターであることのマーキングや、標準の要求に応じて Postgres 構成の設定を含む、生成された `config.yml` にさまざまなエントリを追加します。
- コンプライアンスを適用するように一部の展開タスクを調整します。
- 展開の最後にチェックを実行します。

オプション `--excluded_tasks=compliance ~tpaexec deploy` を指定することにより、展開時チェックをスキップできます。この機能は、完全なコンプライアンスが不可能なテストシステムを使用する場合、SSL 証明書が使用できない場合など、テストのみを対象としています。

TPA が選択した標準に意図的に準拠しない場合があります。これらは以下の例外に文書化されています。

STIG

STIG 準拠は、`tpaexec configure` への `--compliance stig` オプションによって示されます。

オプションの互換性

STIG コンプライアンスには、`bare` プラットフォームと `epas` フレーバーが必要です。バージョン 8 または 9 の RedHat OS が必要です。

config.yml の設定

EDB Postgres Advanced Server の SQL /Protect 機能を使用するために、次のエントリが `cluster_vars` に追加されます。

```
extra_postgres_extensions: [ sql_protect ]
```

次のエントリが `cluster_vars` に追加されて、クライアントに SSL 認証を使用するように強制します。

```
hba_force_hostssl: True
hba_force_certificate_auth: True
hba_cert_authentication_map: sslmap
```

次のエントリが `cluster_vars` に追加されて、`postgresql.conf` で GUC が設定されます。

```
tcp_keepalives_idle: 10
tcp_keepalives_interval: 10
tcp_keepalives_count: 10
log_destination: "stderr"
postgres_log_file_mode: "0600"
```

次のエントリが `cluster_vars` の `postgres_conf_settings` に追加されて、`postgresql.conf` で GUC を設定します。

```
edb_audit: "xml"
edb_audit_statement: "all"
edb_audit_connect: "all"
edb_audit_disconnect: "all"
statement_timeout: 1000
client_min_messages: "ERROR"
```

デプロイメントの違い

TPA は、展開中に、通常の操作に必要な接続数に対応して、作成するデータベースユーザーの接続制限を設定します。各ユーザーが設定されると、SSL クライアント証明書が提供されたことも確認します。

クライアント SSL 証明書の提供

STIG は、クライアント接続に国防総省承認の ssl 証明書を必要とします。これらの証明書は TPA では生成できないため、提供する必要があります。クラスター内のノードからユーザーの認証を設定すると、TPA はノードで証明書/キーペアを検索します。証明書とキーは、`ssl_client_cert_dir` 設定で指定されたディレクトリの `<username>.cert` および `<username>.key` というファイルにある必要があります。この設定のデフォルトは `/` であるため、ファイルは、たとえば、`barman` ユーザーがセットアップされている場合の `/barman.cert` および `/barman.key` になります。

最終チェック

展開の最後に、TPA はサーバーで FIPS が有効になっていることを確認します。

例外

フェールオーバーマネージャーとして EFM を選択すると、TPA は EFM ユーザーのパスワード認証を構成します。これは、すべての TCP 接続が証明書認証を使用するという STIG 要件に反しています。この例外の理由は、EFM が証明書認証をサポートしていないためです。

CIS

CIS 準拠は、`tpaexec configure` への `--compliance cis` オプションによって示されます。

config.yml の設定

次のエントリが `cluster_vars` に追加されて、`postgresql.conf` で GUC が設定されます。

```
log_connections: "on"
log_disconnections: "on"
```

次のエントリが `cluster_vars` に追加されて、必要な拡張機能を有効にします。

```
extra_postgres_extensions: ["passwordcheck", "pgaudit"]
```

次のエントリが `cluster_vars` に追加されて、postgres OS ユーザーの `umask` を設定します。

```
extra_bash_rc_lines: "umask 0077"
```

次のエントリが `cluster_vars` の `postgres_conf_settings` に追加されて、`postgresql.conf` で GUC を設定します。

```
log_error_verbosity: "verbose"
log_line_prefix: "%m [%p]: [%l-1] db=%d,user=%u,app=%a,client=%h "
log_replication_commands: "on"
temp_file_limit: "1GB"
```

最終チェック

展開の最後に、TPA はサーバーで FIPS が有効になっていることを確認します。

例外

CIS 仕様で記載されているように、TPA は `pgBackRest` をサポートしていません。代わりに、TPA は `Barman` をインストールします。

TPA は、CIS 仕様で必要とされる `set_user` のインストールと構成を行いません。これは、Postgres ユーザーによるログオンを防止すると、TPA がデータベースに接続および構成できなくなるためです。

Locale

一部のプラットフォームイメージと環境では、地域と言語設定を設定することが望ましい場合があります。

デフォルトでは、TPAexec は `en_US.UTF-8` ロケールシステムファイルをインストールします。 `config.yml` で目的のロケールを設定できます。

```
user_locale: en_GB.UTF-8
```

サポートされているロケールを見つけるには、次のコマンドの出力を参照します。

```
localectl list-locales
```

または、Debian または Ubuntu のファイル `/etc/locales.defs` の内容。

Patroni cluster management commands

Patroni は、次のコマンドオプションを使用して、M1 アーキテクチャでシングルマスターフェールオーバーマネージャーとして使用できます。

```
tpaexec configure cluster_name -a M1 --enable-patroni --postgresql 14
```

または `config.yml` オプションを設定することにより

```
cluster_vars:
  failover_manager: patroni
```

TPA は、 `patroni` パッケージ PGDG リポジトリからまたは `edb-patroni` パッケージ EDB リポジトリからを使用して Patroni クラスターを展開できます。これは、 `config.yml` の `cluster_vars` の下の `patroni_package_flavour` オプションを介して構成できます。これは、 `--patroni-package-flavour` コマンドライン引数を介して設定することもできます。 `patroni_package_flavour` が明示的に設定されていない場合、TPA は、構成されたリポジトリに基づいてフレーバーを推論しようとします。EDB リポジトリが構成されている場合、暗黙的に `edb` フレーバーを選択し、それ以外の場合は暗黙的に `community` フレーバーを選択します。

TPA `configure` は 3 つの `etcd` ノードを追加し、オプション `--enable-haproxy` を指定する場合 2 つの `haproxy` ノードを追加する場合があります。 `etcd` は、分散構成ストア DCS に使用されます。 Patroni は他の DCS バックエンドをサポートしていますが、現在 EDB または TPA ではサポートされていません。

HAProxy の代替として、 `--enable-pgbouncer` オプションを使用して、Postgres ノードで PgBouncer を構成できます。 PgBouncer は、プライマリの接続をプールするように構成されます。 Patroni は、クラスター内のフェールオーバーまたはスイッチオーバー時に PgBouncer を再構成するように構成されるため、PgBouncer は新しいプライマリ Postgres インスタンスに従います。

TPA は、既存の PostgreSQL クラスターを変換する Patroni の機能を使用します。これにより、TPA は構成を初期化および管理できます。 PostgreSQL クラスターが作成されると、Patroni が管理を引き継ぎます。 TPA は、セットアップ中に使用された postgres 構成ファイルを削除します。

セットアップが完了すると、クラスターの TPA と `config.yml` の設定を使用して、Postgres を引き続き管理できます。 コマンドライン `patronictl` や REST API などの Patroni インターフェイスを使用することもできますが、可能な限り TPA メソッドを使用することをお勧めします。

Patroni パッケージバージョン

デフォルトでは、TPA は Patroni の最新バージョンをインストールします。

インストールされる Patroni パッケージのバージョンは、`config.yml` ファイルの `cluster_vars` セクションの下の `patroni_package_version`: `xxx` を含めることにより指定できます。

```
cluster_vars:
...
  patroni_package_version: 4.0.0-1PGDG.rhel8`
...
```

`apt` または `yum` が受け入れるバージョン指定子を使用できます。

バージョンが一致しない場合は、* ワイルドカードを追加してみてください。これは、パッケージバージョンに `2:...` のようなエポック修飾子がある場合に必要になることがよくあります。

これらの構成変数は、TPA での Patroni の展開での特定の動作を制御するために使用できます。

Variable	Default value	Description
<code>patroni_super_user</code>	postgres	スーパーユーザーロールの postgres で作成するユーザー。
<code>patroni_replication_user</code>	replicator	レプリケーションロールの postgres で作成するユーザー名。
<code>patroni_restapi_user</code>	patroni	patroni REST API 用に構成するユーザー名。
<code>patroni_rewind_user</code>	rewind	<code>pg_rewind</code> ファンクションの postgres で作成するユーザー名。
<code>patroni_installation_method</code>	pkg	パッケージまたはソースたとえば git リポジトリまたは docker の場合はローカルソースディレクトリなどから patroni をインストールします。
<code>patroni_package_flavour</code>	community if no EDB repository is configured, else edb	<code>`edb-patroni`</code> パッケージ <code>`edb`</code> フレーバー、EDB リポジトリが必要 または <code>`patroni`</code> パッケージ <code>`community`</code> フレーバー、PGDG および EPEL RedHat ベースのみのリポジトリが必要 をインストールするかどうか。
<code>patroni_ssl_enabled</code>	no	REST API および ctl 接続の SSL を有効にするかどうか。可能な場合は、クラスター SSL 証明書と CA を使用します。
<code>patroni_rewind_enabled</code>	yes	postgres rewind を有効にするかどうか、 <code>patroni_rewind_user</code> で定義されたユーザーを作成し、 <code>config</code> セクションを追加します。
<code>patroni_watchdog_enabled</code>	no	追加のスプリットブレイン防止のためにカーネルウォッチドッグを構成するかどうか。
<code>patroni_dcs</code>	etcd	DCS に使用するバックエンド。現時点では etcd のみがオプションです。

次のページに続く

表 10 – 前のページからの続き

Variable	Default value	Description
<code>patroni_listen_port</code>	8008	REST API TCP ポート番号
<code>patroni_conf_settings</code>	{}	パトローニ構成のオーバーライドを含む構造化データオブジェクト。部分的なデータを提供でき、生成された構成とマージされます。実行時に既知のインスタンス情報に基づいて生成された値をオーバーライドしないように注意してください。
<code>patroni_dynamic_conf_settings</code>	{}	DCS 設定専用のオプションの構造化データ。これは <code>patroni_conf_settings</code> にマージされます。
<code>patroni_repl_max_lag</code>	None	これは、 <code>haproxy_read_only_load_balancer_enabled</code> が <code>true</code> の場合にのみ <code>haproxy</code> バックエンドヘルスチェックで使用されます。 <code>/replica?lag</code> の可能な値については、 :ref: 'Configuring lvm-direct for production <Patroni cluster management commands>' を参照してください

Patroni 構成ファイル設定

patroni の構成は3つのレイヤーから構築され、Patroni デーモンによって設定されたデフォルトから始まり、DCS からロードされた構成、最後にローカル構成から構築されます。最後は、構成ファイルから制御でき、環境を介してオーバーライドできます。TPA は構成ファイルを制御し、値はこの順序で構築されます。

API に送信され、構成ファイルのブートストラップセクションに保存される DCS 構成。

- postgres の TPA 変数は DCS 設定にロードされます。 [postgresql.conf](#) を参照してください。一部の機能はサポートされていません。以下の注意事項を参照してください。
- DCS 設定の Patroni のデフォルト
- `patroni_dynamic_conf_settings` のユーザー指定のデフォルト。DCS 設定をオーバーライドする場合は、ここで行うことができます。

YAML 構成ファイルに保存されるローカル構成

- 上記の前の手順からロードされた `bootstrap.dcs`。
- `patroni_ssl_enabled` などの機能フラグによって有効になる構成。上記の表を参照してください。
- 次に、最後にユーザー指定設定からオーバーロードされます `patroni_conf_settings` オプション。機能フラグによって制御されない構成を変更または追加する場合、これはそれに最適な場所です。

構成は、IP アドレス、ポート番号、クラスターロールなどのクラスター情報から TPA によって生成された構成にマージされることに注意してください。これは、クラスターの安定したオペレーションに影響を与える可能性があるため、オーバーライドするものに注意してください。

Patroni はすべての postgres 構成を DCS に保存し、これが postgres に配布される方法と時期を制御するため、TPA の一部の機能は patroni と互換性がありません。

- `postgres_conf_template` 設定で `postgresql.conf` の生成に使用されたテンプレートを変更することはできません。
- 設定 `postgres_conf_dir` では、Postgres 構成ファイルの場所を変更することはできません。

TPA の Patroni 構成 `config.yml`

単一の値をオーバーライドできます。

```
cluster_vars:
  patroni_conf_settings:
    bootstrap:
      dcs:
        ttl: 120
```

またはフルブロック Patroni ドキュメントからの例を使用して

```
cluster_vars:
  patroni_conf_settings:
    restapi:
      http_extra_headers:
        X-Frame-Options: SAMEORIGIN
        X-XSS-Protection: 1; mode=block
        X-Content-Type-Options: nosniff
      https_extra_headers:
        Strict-Transport-Security: max-age=31536000; includeSubDomains
```

デフォルトの TPA 構成変数に存在する値またはセクションを無効にする場合は、値を `null` に設定できます。これにより、パトロニは、構成ファイルをロードするときにこのセクションを無視します。

たとえば、`log` のデフォルトの TPA 構成は次のとおりです。

```
log:
  dir: /var/log/patroni
```

ロギングをオフにするには、これを `config.yml` に追加します。

```
cluster_vars:
  patroni_conf_settings:
    log: null
```

TPA は、Patroni クラスターを管理するためのこれらの最小限のツールセットを提供します。

ステータス

Patroni 実行に従って TPA クラスターの現在のステータスを確認するには

```
tpaexec status cluster_name
```

スイッチオーバー

レプリカノードへのスイッチオーバーを実行するにはメンテナンスを実行するには、次のコマンドを実行します。

```
tpaexec switchover cluster_name new_primary
```

`new_primary` 引数は、現在正常なレプリカとして実行されている既存のクラスターノードの名前である必要があります。スイッチオーバーが実行される前に、これが `true` を確認するためにチェックが実行されます。

スイッチオーバーが実行されたら、`deploy` および `test` を実行して、正常なクラスターを確認することをお勧めします。

```
tpaexec deploy cluster_name
tpaexec test cluster_name
```

TPA は、`config.yml` の内容に関係なく、展開中にノードの現在のロールを検出しますたとえば、別のノードがリーダーである場合。

Adding Postgres extensions

デフォルトの Postgres 拡張機能

デフォルトでは、TPA は次の拡張機能をすべての Postgres データベースに追加しますおよび必要に応じて、対応するエントリを共有プリロードライブラリに自動的に追加します

- `pg_stat_statements`
- `pg_freespacemap`
- `pg_visibility`
- `pageinspect`
- `pgstatus`

ユーザー定義の拡張機能

拡張機能名、必要な共有プリロードエントリ、および拡張機能を含むパッケージを指定することにより、`config.yml` 内で追加の拡張機能を構成できます。

拡張機能を追加するときは、パッケージ名を `extra_postgres_packages` に含め、拡張機能名の両方を `extra_postgres_extensions` または `postgres_databases` の下で定義されたデータベースの `extensions` リストに含めてください。

拡張機能が必要な場合は、拡張機能の共有プリロードエントリ名を `preload_extensions` リストに追加します。この名前は拡張機能名自分自身とは異なる場合があることに注意してください、拡張機能自分自身のドキュメントを必ず確認してください。

次に、拡張機能とライブラリ名の別のエントリを使用して共有プリロード拡張機能リストに追加する必要がある拡張機能の簡単な例を示します。

```
cluster_vars:
  [...]
  extra_postgres_packages:
    - postgresql-17-my-extension
  extra_postgres_extensions:
    - my-extension
  preload_extensions:
    - my_extension
```

以下のセクションでは、詳細な情報を提供します。

- *Reconciling changes made outside of TPA*
- *Creating Postgres databases*
- *shared_preload_libraries*
- *Installing Postgres-related packages*

TPA 認識される拡張機能

次の拡張機能のリストでは、`config.yml` `extra_postgres_extensions` または `postgres_databases` で指定されたデータベースの `extensions` リストのいずれかに拡張機能名を追加するだけでよく、TPA は自動的に正しいパッケージと必要なエントリを `shared_preload_libraries` に含めます。

- `edb_pg_tuner`
- `query_advisor`
- `edb_wait_states`
- `sql_profiler`
- `pg_failover_slots`
- `sql_protect`
- `edb_stat_monitor`
- オートクラスター
- `refdata`
- ブルーフィン
- `postgis`
- `pgaudit`
- パスワードチェック

tpaexec deprovision

プロビジョニングを解除すると、クラスターと関連するリソースが破壊されます。

aws プラットフォームを使用するクラスターの場合、クラスターに設定されたインスタンスとすべてのキーペア、ポリシー、ボリューム、セキュリティグループ、ルートテーブル、VPC サブネット、インターネットゲートウェイ、および VPC が削除されます。

docker プラットフォームを使用するクラスターの場合、コンテナ、コンテナでソースビルド用に設定された ccache ディレクトリ、およびクラスター用に設定された Docker ネットワークを削除します。

すべてのプラットフォームで、ssh キー、保存されたパスワード、Ansible インベントリ、ログを含む、tpaexec provision によってローカルに作成されたすべてのファイルが削除されます。

tpaexec info

info コマンドを使用して、TPA インストールに関する情報を出力できます。この情報の提供は、トラブルシューティングに役立ちます。

使用法

- tpaexec info を実行します

サブコマンド

- tpaexec info version

現在の TPA バージョンを表示します

- tpaexec info platforms

使用可能な展開プラットフォームを表示します

- tpaexec info architectures

使用可能な展開アーキテクチャを表示します

- tpaexec info platforms/<name>

特定のプラットフォームに関する情報を表示します

- tpaexec info architectures/<name>

特定のアーキテクチャに関する情報を表示します

出力例

tpaexec info コマンドは次を出力します。

```
# TPAexec 23.29

tpaexec=/opt/EDB/TPA/bin/tpaexec
TPA_DIR=/opt/EDB/TPA
PYTHON=/opt/EDB/TPA/tpa-venv/bin/python3 (v3.12.18, venv)
```

(次のページに続く)

```
TPA_VENV=/opt/EDB/TPA/tpa-venv
ANSIBLE=/opt/EDB/TPA/tpa-venv/bin/ansible (v2.16.3)
Validated: ea844d1b90295597d080bbf824dbbc6954886cb54ffdb265c7c71b99bedee67b [OK]
```

tpaexec reconfigure

`tpaexec reconfigure` コマンドは、`config.yml` を読み取り、引数に応じてさまざまな方法でクラスターを変更する変更バージョンを生成します。

引数

他の `tpaexec` コマンドと同様に、クラスターディレクトリは常に指定する必要があります。

クラスターのアーキテクチャの変更

次の引数により、クラスターのアーキテクチャを変更できます。

- `--architecture <architecture>` (必須) クラスターの新しいアーキテクチャ。PGD-Always-ON および PGD-X を有効な引数として受け入れます
- `--pgd-proxy-routing <global|local>` PGD-Always-ON に必要 PGD-Proxy が接続をルーティングする方法。この引数の意味の詳細については、[PGD-Always-ON](#) を参照してください。
- `--edb-repositories <repositories>` オプション EDB パッケージリポジトリのスペース区切りリスト。通常、これを指定する必要はありません。`tpaexec configure` は、クラスターで使用されている postgres フレーバーに基づいて、適切なリポジトリを選択します。

アーキテクチャを変更した後、[Upgrading your cluster](#) を実行して、クラスターに必要な変更を加えます。

クラスターを PGD-プロキシから PGD-Always-ON の接続マネージャーに変更する

- `--enable-connection-manager(required)` PGD 5.9+クラスター上の PGD-プロキシから新しいビルトイン接続マネージャーへの移行を許可するオプション。

クラスターを 2q から EDB リポジトリへの変更

`--replace-2q-repositories` 引数は、クラスターが使用する 2ndQuadrant リポジトリを削除し、必要に応じて EDB リポジトリを追加して、それらを置き換えます。

この引数で再構成した後、`[tpaexec promote](tpaexec-deploy.md)` を実行して、クラスターに必要な変更を加えます。

出力フォーマット

次のオプションは、出力の形式を制御します。

- `--describe` 何も変更せずに、変更される内容の説明を表示します。
- `--check` 加えられる変更を検証し、エラーがあれば何も変更せずに、検証の結果発生したエラーまたは警告を表示します。

- `--output <filename>` `config.yml` 以外のファイルに出力を書き込みます。

サンプル呼び出し

```
$ tpaexec reconfigure ~/clusters/speedy\  
--architecture PGD-Always-ON\  
--pgd-proxy-routing local
```

7.6 Selective task execution

7.6.1 タスクセクターを使用する

`tpaexec deploy` への `--excluded_tasks` および `--included_tasks` オプションを使用して、完全な展開を構成するタスクのサブセットのみを実行するように TPA に指示できます。これらの引数はそれぞれ、セクターのコンマ区切りリストとして扱われる文字列です。同様に、クラスター全体または個別のインスタンスに対して、`config.yml` の `excluded_tasks` および `included_tasks` 変数を設定できます。`config.yml` では、コンマ区切り文字列または `yaml` リストを使用できます。

`excluded_tasks` によって一致するタスクは常に除外されます。`included_tasks` を指定すると、一致しないタスクは暗黙的に除外されます。

以下に詳細を示すように、一部のセクターはどちらのリストでも使用でき、一部は `excluded_tasks` リストでのみ使用できます。`tpaexec test` には、別のセクターセットが使用できます。

7.6.2 例

バーマン関連のタスクを実行せずに展開するには

```
[tpa]$ tpaexec deploy <clustername> --excluded_tasks=barman
```

`repmgr` 関連のタスクのみを実行しているを展開するには

```
[tpa]$ tpaexec deploy <clustername> --included_tasks=repmgr
```

インスタンスにホスト名を設定せずに展開するには

```
[tpa]$ tpaexec deploy <clustername> --excluded_tasks=hostname
```

ブートストラップおよび `ssh` タスクが実行されないようにするには、次を `config.yml` に追加します。

```
cluster_vars:  
  excluded_tasks:  
    - bootstrap  
    - ssh
```

7.6.3 tpaexec deploy でサポートされているセレクトター

次のセレクトターは、包含または除外のいずれかがサポートされています。

- バーテンダー

Barman に関連するタスク。

- bdr

BDR のセットアップに関連するタスク PGD クラスター内で使用される場合を含むこのセレクトターが除外された場合、TPA は config.yml で指定されたとおりに拡張機能をインストールおよび構成しますが、ノードグループを作成せず、ノードに参加しようとしません。

- create_pgd_proxy_system_user

pgd_proxy のシステムユーザーを作成するタスク

- create_postgres_system_user

postgres のシステムユーザーを作成するタスク

- efm

EFM に関連するタスク。

- etcd

etcd に関連するタスク

- first-backup

最小限のバーマンバックアップが存在することを保証するタスク。

- ハプロキシ

haproxy に関連するタスク。

- ハープ

harp に関連するタスク。

- パトローニ

パトローニに関連するタスク。

- pem-agent

PEM エージェントに関連するタスク。

- pem-server

PEM サーバーに関連するタスク。

- pem-webserver

PEM サーバーでの Web サーバーの構成に関連するタスク。

- pg-backup-api

Barman の Postgres バックアップ API に関連するタスク。

- pgbouncer

PgBouncer に関連するタスク。

- pgd-proxy

PGD プロキシに関連するタスク。

- pglogical

pglogical に関連するタスク。

- pkg

システムパッケージマネージャーを使用してパッケージをインストールするタスク。

- 展開後

展開後フックが定義されている場合。

- postgres

postgres に関連するタスク。

- レプリカ

実行されるタスクと postgres レプリカとして機能するインスタンス。

- repmgr

repmgr に関連するタスク

- リスタート

サービスを再起動するタスク

- sys

postgres または関連ソフトウェアに固有のタスクの前のシステムセットアップに関連するタスク。

- zabbix-agent

zabbix エージェントに関連するタスク。

次のセレクトアーは、除外の場合にのみサポートされています。

- アーティファクト

Uploading artifacts に関連するタスク

- パーマン-クリーン

Barman がソースからビルドされている場合、Barman ビルドディレクトリをクリーンアップするタスク。

- barman-pre-config

barman-pre-config フックが定義されている場合。

- bdr-pre-node-creation

bdr-pre-node-creation フックが定義されている場合。

- bdr-post-group-creation

bdr-post-group-creation フックが定義されている場合。

- bdr-pre-group-join

bdr-pre-group-join フックが定義されている場合。

- ブートストラップ

残りの展開が実行される前に、Python およびその他の最小限の依存関係が存在することを確認するタスク。
関連する要件を手動でインストールしたことが確かな場合にのみ、これを除外します。

- ビルドクリーン

ソースからビルドされているソフトウェアのビルドディレクトリをクリーンアップするタスク。

- build-configure

ソースからビルドされているソフトウェアを構成するタスク。

- Cloudinit

Cloud-init によって管理されるホストでのみ実行されるタスク。

- コミットスコープ

BDR コミットスコープの構成に関連するタスク。

- config

構成ファイルを作成するタスク。

- efm-pre-config

efm-pre-config フックが定義されている場合。

- fs

インスタンスでの追加の *Filesystem configuration* のセットアップに関連するタスク。

- ホストキー

Managing SSH host keys を設定するタスク

- ホスト名

ホスト名を設定するタスク。

- ホスト

Configuring /etc/hosts のタスク

- initdb

initdb を実行するタスク

- ローカルリポジトリ

Creating and using a local repository を設定するタスク

- ロケール

Locale をインストールするタスク

- openvpn

OpenVPN を設定するタスク。

- pg-backup-api-clean

Postgres バックアップ API がソースからビルドされている場合、ビルドディレクトリをクリーンアップするタスク。

- pgbouncer-config

pgbouncer の構成ファイルを作成するタスク。

- pgpass

Configuring .pgpass ファイルを作成するタスク。

- リポ後

リポ後フックが定義されている場合。

- postgres-clean

postgres がソースからビルドされている場合、build ディレクトリをクリーンアップするタスク。

- postgres-config

postgres-config フックが定義されている場合。

- postgres-config-final

postgres-config-final フックが定義されている場合。

- 事前展開

展開前フックが定義されている場合。

- pre-initdb

pre-initdb フックが定義されている場合。

- レプリケーションセット

BDR3 クラスター上の監視専用レプリケーションセットに関連するタスク。

- repmgr-clean

repmgr がソースからビルドされている場合、ビルドディレクトリをクリーンアップするタスク。

- repmgr-configure

repmgr がソースからビルドされている場合に構成するタスク。

- リポジトリ

パッケージリポジトリを設定するタスク。

- rsyslog

rsyslog に関連するタスク。

- サービス

構成や再起動を含むシステムサービスに関連するタスク。

- src

ソースからパッケージをビルドおよびインストールするタスク。

- ssh

インスタンス間の ssh の設定に関連するタスク。

- sysctl

sysctl 設定を設定およびリロードするタスク。

- sysstat

sysstat サービスに関連するタスク。

- tpa

インスタンスにインストールされる TPA 独自のファイルに関連するタスク。

- ユーザー

システムユーザーの設定に関連するタスク。

- ウォッチドッグ

パトリニククラスター上のカーネルウォッチドッグに関連するタスク。

7.6.4 tpaexec test でサポートされているセレクトター

次のセレクトターは、`tpaexec test` の実行にのみ適用されます。

- 迷彩柄

BDR または PGD クラスターでの CAMO のテストに関連するタスク。

- ddl

BDR または PGD クラスターでの DDL のテストに関連するタスク。

- 失敗

問題が検出されたかどうかを中止するタスク。障害に関係なくテストを実行するには、このセレクトターを除外します。

- pgbench

pgbench を実行するタスク

- sys

システムレベルのテストを実行するタスク。

- barman、bdr、haproxy、pg-backup-api、pgbouncer、pgd-proxy、postgres、repmgr、

さまざまなソフトウェアコンポーネントをテストするタスク。