



Trusted Postgres Architect

リリース *23.24.0*

EnterpriseDB Corporation

2023 年 10 月 19 日

目次

第 1 章 Introduction	3
1.1 TPA でできること	3
1.2 使用方法	4
1.2.1 構成	4
1.2.2 プロビジョニング	5
1.2.3 展開	5
1.2.4 テスト	5
1.2.5 インクリメンタルな変更	6
1.2.6 クラスター管理	6
1.2.7 Ansible で拡張可能	6
1.3 Postgres がベース	6
1.4 TPA でのバージョン管理	7
1.4.1 下位互換性	7
1.5 はじめに	7
第 2 章 Trusted Postgres Architect release notes	9
2.1 Trusted Postgres Architect 23.24 release notes	9
2.2 Trusted Postgres Architect 23.23 release notes	10
2.3 Trusted Postgres Architect 23.22 release notes	11
2.4 Trusted Postgres Architect 23.21 release notes	11
2.5 Trusted Postgres Architect 23.20 release notes	12
2.6 Trusted Postgres Architect 23.19 release notes	13
2.7 Trusted Postgres Architect 23.18 release notes	14
2.8 Trusted Postgres Architect 23.17 release notes	14
2.9 Trusted Postgres Architect 23.16 release notes	15
2.10 Trusted Postgres Architect 23.15 release notes	16
2.11 Trusted Postgres Architect 23.14 release notes	16
2.12 Trusted Postgres Architect 23.13 release notes	16
2.13 Trusted Postgres Architect 23.12 release notes	17
2.14 Trusted Postgres Architect 23.1 to 23.11 release notes	18
2.14.1 TPA 23.11	18
2.14.2 TPA 23.10	18
2.14.3 TPA 23.9	19
2.14.4 TPA 23.8	19
2.14.5 TPA 23.7	19
2.14.6 TPA 23.6	20
2.14.7 TPA 23.5	21
2.14.8 TPA 23.4	21
2.14.9 TPA 23.3	21
2.14.10 TPA 23.2	22

2.14.11	TPA 23.1	23
第 3 章	TPA installation	25
3.1	クイックスタート	25
3.1.1	Debian または Ubuntu にリポジトリを追加し、TPA をインストールする	25
3.1.2	RHEL、Rocky、AlmaLinux、または Oracle Linux にリポジトリを追加し、TPA をインストールします	26
3.1.3	追加の依存関係をインストールする	26
3.1.4	インストールを確認します（通常ユーザーとして実行）	26
3.2	TPA をインストールする場所	26
3.3	TPA パッケージのインストール	26
3.3.1	Debian または Ubuntu にリポジトリを追加する	27
3.3.2	RHEL、Rocky、AlmaLinux、または Oracle Linux にリポジトリを追加します	27
3.3.3	Debian または Ubuntu にインストールする	27
3.3.4	RHEL、Rocky、AlmaLinux または Oracle Linux にインストール	27
3.4	TPA Python 環境のセットアップ	27
3.5	インターネットまたはネットワークアクセスなしで TPA をインストールする（エアギャップ）	28
3.5.1	TPA パッケージのダウンロード	28
3.5.2	Python パッケージインデックスにアクセスせずにインストールする	28
3.6	TPA のインストールの確認	29
3.7	TPA のアップグレード	29
3.8	Ansible コミュニティのサポート	29
第 4 章	Open source TPA	31
4.1	Trusted Postgres Architect（TPA）とは何ですか？	31
4.2	次のステップ	31
4.3	TPA オープンソースの FAQ	31
4.3.1	EDB の顧客でなくても、これを使用できますか？	31
4.3.2	問題を報告できますか？	31
4.3.3	貢献できますか？	32
第 5 章	Installing TPA from source	33
5.1	クイックスタート	33
5.2	クローンとセットアップ	33
5.3	依存関係	34
5.3.1	Python 3.6+	34
5.3.2	仮想環境オプション	35
第 6 章	Creating a cluster	37
6.1	A First Cluster Deployment	37
6.1.1	TPA のインストール	37
6.1.2	Docker のインストール	37
6.1.3	CgroupVersion	38
6.1.4	TPA で構成を作成する	38
6.2	Cluster configuration	42
6.2.1	クイックスタート	42
6.2.2	構成オプション	42

6.2.3	ソフトウェアの選択	45
6.2.4	オーバーライド	48
6.2.5	アンシブルタワー	49
6.2.6	git リポジトリ	49
6.2.7	例	49
6.3	tpaexec provision	52
6.3.1	前提条件	53
6.3.2	クイックスタート	53
6.3.3	オプション	54
6.3.4	インスタンスへのアクセス	54
6.3.5	生成されたファイル	55
6.4	tpaexec deploy	56
6.4.1	前提条件	57
6.4.2	クイックスタート	57
6.4.3	選択的展開	58
6.4.4	deploy.yml	58
6.5	tpaexec test	58
6.5.1	クイックスタート	59
第 7 章	Architectures	61
7.1	PGD-Always-ON	61
7.1.1	クラスター構成	61
7.2	BDR-Always-ON	63
7.2.1	クラスター構成	64
7.3	M1	65
7.3.1	デフォルトのレイアウト	66
7.3.2	アプリケーションとバックアップのフェイルオーバー	66
7.3.3	クラスター構成	67
第 8 章	Platforms	69
8.1	aws	69
8.1.1	API アクセス設定	69
8.1.2	はじめに	71
8.1.3	構成	72
8.2	bare(-metal servers)	76
8.2.1	SSH アクセス	77
8.2.2	配布サポート	77
8.2.3	IP アドレス	78
8.3	Docker	78
8.3.1	概要	78
8.3.2	オペレーティングシステムの選択	78
8.3.3	Docker のインストール	79
8.3.4	Docker コンテナ管理	81
第 9 章	Customizations	83
9.1	Cluster configuration	83
9.1.1	config.yml	83
9.1.2	変数	84

9.1.3	クラスター変数	85
9.1.4	インスタンス変数	85
9.1.5	instance_defaults	86
9.1.6	場所	87
9.2	Instance configuration	88
9.2.1	システムレベルの構成	88
9.2.2	Postgres	89
9.2.3	その他のコンポーネント	90
9.2.4	Postgres 起動後	91
9.3	Building from source	91
9.3.1	クイックスタート	92
9.3.2	構成	92
9.3.3	ローカルソースディレクトリ	93
9.3.4	リビルド	94
9.4	TPA hooks	94
9.4.1	まとめ	95
9.4.2	汎用フック	95
9.4.3	PGD フック	96
9.4.4	その他のフック	97
9.4.5	新しいフック	97
第 10 章	Cluster management	99
10.1	Upgrading your cluster	99
10.1.1	はじめに	99
10.1.2	構成	100
10.1.3	BDR-Always-ON から PGD-Always-ON へのアップグレード	100
10.1.4	PGD-Always-ON	101
10.1.5	BDR-Always-ON	101
10.1.6	M1	102
10.1.7	パッケージバージョンの選択	102
10.2	tpaexec switchover	103
10.2.1	例	103
10.2.2	アーキテクチャオプション	103
10.3	BDR/HAProxy server pool management	103
10.3.1	例	104
10.4	tpaexec rehydrate	104
10.4.1	前提条件	105
10.4.2	例	105
10.4.3	段階的に水分を補給する	106
10.4.4	付録	106
第 11 章	Miscellaneous	109
11.1	TPA and Ansible Tower/Ansible Automation Controller	109
11.1.1	TPA を使用するための Tower の準備	109
11.1.2	クラスターのセットアップ	110
11.2	TPA, Ansible, and sudo	111
11.2.1	Ansible sudo 呼び出し	111

11.2.2	推奨事項	112
11.2.3	SSH および sudo パスワード	112
11.2.4	sudo オプション	113
11.2.5	ロギング	113
11.2.6	ローカル権限	113
11.3	TPA - PuTTY Configuration guide	114
11.3.1	キー変換	114
11.3.2	PuTTY を構成する	115
11.4	Troubleshooting	115
11.4.1	Python 仮想環境を再作成する	115
11.4.2	認証情報に関する奇妙な AWS エラー	115
11.4.3	ロギング	116
11.4.4	クラスターテスト	116
11.4.5	TPA サーバーテスト	117
11.4.6	特定のタグをスキップまたは含める	117
第 12 章	Reference	119
12.1	Installation	119
12.1.1	Running TPA in a Docker container	119
12.1.2	Managing clusters in a disconnected or air-gapped environment	120
12.1.3	Distribution support	122
12.1.4	TPA capabilities and supported software	123
12.2	Services	124
12.2.1	EDB Postgres Distributed configuration	124
12.2.2	Barman	127
12.2.3	Configuring EFM	129
12.2.4	Configuring haproxy	130
12.2.5	Configuring HARP	132
12.2.6	Configuring Postgres Enterprise Manager (PEM)	135
12.2.7	Configuring pgbouncer	138
12.2.8	Configuring pgd-proxy	140
12.2.9	pglogical configuration	142
12.2.10	Configuring repmgr	147
12.3	Configuration	147
12.3.1	How TPA uses 2ndQuadrant and EDB repositories	147
12.3.2	Configuring EDB Repos 2.0 repositories	150
12.3.3	Configuring 2ndQuadrant repositories	150
12.3.4	Configuring APT repositories	151
12.3.5	Configuring YUM repositories	152
12.3.6	Creating and using a local repository	153
12.3.7	Installing from source	156
12.3.8	Git credentials	157
12.3.9	Environment	159
12.3.10	Python environment	159
12.3.11	Configuring /etc/hosts	160
12.3.12	Filesystem configuration	161
12.3.13	Uploading artifacts	167

12.3.14	ssh_key_file	168
12.3.15	Managing SSH host keys	168
12.3.16	Postgres source installation	169
12.3.17	Installing packages	171
12.3.18	Running initdb	172
12.3.19	Installing Postgres-related packages	173
12.3.20	SSL Certificates	173
12.3.21	Setting sysctl values	174
12.4	Postgres configuration	175
12.4.1	Creating Postgres databases	175
12.4.2	Creating Postgres tablespaces	176
12.4.3	postgresql.conf	177
12.4.4	pg_hba.conf	179
12.4.5	pg_ident.conf	180
12.4.6	Configuring .pgpass	181
12.4.7	The postgres Unix user	181
12.4.8	Creating Postgres users	182
12.5	Commands	183
12.5.1	tpaexec archive-logs	183
12.5.2	tpaexec download-packages	186
12.5.3	TPA custom commands	187
12.5.4	TPA custom tests	188
12.5.5	Shared instances	191
12.5.6	Locale	191
12.5.7	Patroni cluster management commands	191
12.5.8	tpaexec info	195
12.5.9	tpaexec reconfigure	196

© Copyright EnterpriseDB UK Limited 2015-2023-全著作権所有。

第1章 Introduction

TPA は、Ansible を使用して EDB の推奨に従って Postgres クラスターをデプロイするオーケストレーションツールです。

TPA は、Postgres の展開とサポートでの長年の苦労した経験に基づいた EDB に続く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、実稼働環境と同様に、クイックテストベッドのセットアップにも適用できます。

1.1 TPA でできること

TPA は、トポロジーから構成の細部に至るまで、Postgres クラスターを説明するために使用できる宣言型構成メカニズムを中心に構築されています。

`tpaexec configure` を実行することから始めて、いくつかの高レベルの選択（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します。デフォルトの構成はそのまま使用する準備ができていますが、ニーズに合わせて編集できます（生成された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）と、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 にデプロイできます）。
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セットアップなど）。
3. Postgres と関連するコンポーネント（例、PGD、Barman、pgbouncer、repmgr、およびさまざまな Postgres 拡張機能）をインストールして構成します。
4. 展開後、クラスターで自動テストを実行します。
5. 構成に対する将来の変更を展開します（たとえば、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）。

1.2 使用方法

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

TPA は 4 つの異なる段階で動作し、Postgres クラスターを起動します。

- クラスター **クラスター構成** を生成します
- クラスターをホストする **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 **展開** 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
```

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 TPA を実行できます。

これが *list of capabilities and supported software* です。

1.2.1 構成

tpaexec configure

コマンドは、選択したオプションに基づいて、クラスターを説明する単純な YAML 構成ファイルを生成します。構成はすぐに使用する準備ができていますが、ニーズに合わせて変更できます。構成ファイルを編集するのは、作成の前と後の両方で、

make any configuration changes to your cluster への通常の方法です。

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、特定の目的のために 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

トです。例には、「M1」（プライマリレプリカと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 Distributed 5）が含まれます。**プラットフォーム**は、サーバーをホストして、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手段です。

1.2.2 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、必要な権限を持つ 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。これらがベアメタルサーバーでも、クラウドプラットフォームで個別にプロビジョニングされたものでも、まるで TPA で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、クラスターを一部の AWS インスタンス（マルチリージョン）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。

プロビジョニング段階の最後に、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作成され、TPA が SSH 経由でアクセスできます（ルートに `sudo` を使用）。

1.2.3 展開

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そうでない場合があります。ただし、SSH および `sudo` アクセスが利用可能な限り、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。

1.2.4 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およびプラットフォーム固有のテストを実行し、期待どお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。

1.2.5 インクリメンタルな変更

TPA は、プロビジョニング、デプロイ、およびテストがべき等になるように慎重に設計されています。それらを実行し、`config.yml` に変更を加え、プロセスを再度実行して変更をデプロイできます。構成またはインスタンスで何も変更されていない場合、プロセス全体を再実行しても何も変更されません。

1.2.6 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、構成の変更、スイッチオーバー、ダウンタイムゼロのマイナーバージョンアップグレードなど、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、クラスターを簡単かつ安全に管理できます。

1.2.7 Ansible で拡張可能

TPA は *variety of configuration options* をサポートしているため、`config.yml` を編集して `provision/deploy/test` を再実行するだけで、たくさんことができます。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*、これにより、クラスターで実行する Playbook を簡単に作成できます。クラスターディレクトリに `commands/xyz.yml` を作成し、`tpaexec xyz /path/to/cluster` を使用して呼び出します。自動化が必要な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組み込みテストを強化します。`tpaexec test` を使用して、統一された反復可能な方法ですべてのテストを実行すると、危機に対処するとき、または日常のクラスター管理中に、重要なことを見逃すことはありません。
- *Hook scripts*、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` を含む他の多くのフックがあります。これにより、あらゆる種類の Ansible 機能を自由に利用できます。

1.3 Postgres がベース

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 Postgres のみです。インストーラーは、人生をよりシンプルにするため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隠された魔法や何かが立つことはありません。他の Postgres インストーラーで実行できるすべてのことを、TPA クラスターで実行できます。

1.4 TPA でのバージョン管理

TPA は以前、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、セマンティックバージョニングの派生に移動しました。歴史的な理由により、完全な 3 部構成のセマンティックバージョン番号を使用していません。代わりに、TPA は 2 つの部分からなる `major.minor` フォーマットを使用します。マイナーバージョンはリリースごとにインクリメントされます。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必要がある場合にのみインクリメントされます。

1.4.1 下位互換性

TPA の主要な開発原則は、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を必要とする理由はありません。後方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 で有効になります。 $b \geq a$
- その `config.yml` から作成されたクラスターは、TPA X.b で保守および再デプロイ可能になります

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、メジャーバージョンのリリースを避けることを目指しており、例外的な状況でのみそうします。

1.5 はじめに

システムの *TPA installation instructions*、次に *configure your first cluster* に従います。

第2章 Trusted Postgres Architect release notes

Trusted Postgres Architect のドキュメントには、Trusted Postgres Architect 23 の最新バージョンについて説明されています。

Version	Release date
<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>tpaexec-configure</i>	•

2.1 Trusted Postgres Architect 23.24 release notes

リリース：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 リポジトリが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。大多数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 Community 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、デフォルトで <i>2ndQuadrant ansible</i> ではなく <i>community ansible</i> を使用するようになりました。オプション <code>--use-2q-ansible</code> を使用すると、 <i>2ndQuadrant ansible</i> の使用を強制できます。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は、 <code>:ref: `tpaexec-configure <TPA installation></code> を参照してください。
Change	<code>config.yml</code> の <code>`edb_repositories`</code> からリポジトリが削除された場合、 <code>`tpaexec deploy`</code> はノードからリポジトリを削除するようになりました。
Change	TPA は、 <code>harp-proxy</code> と <code>harp-manager</code> が同じノードで実行されていることを検出し、 <code>harp-proxy</code> に別の構成ファイルを使用するようになりました。
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 <code>pg_basebackup</code> 呼び出しで <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 <code>repmgr</code> を受け入れ、そ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 が Ubuntu 22.04 Docker イメージをビルドできないタイプミスを修正しました。
Bug Fix	TPA は、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構成時に拒否します。

2.2 Trusted Postgres Architect 23.23 release notes

リリース：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以下が含まれます。

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はまだ利用できないことに注意してください。
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、JDK 11 が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 PGD からパッケージを除外しません（以前は Barman と psycopg2 は除外されていました）。
Change	SLES でのインストールをサポートするために、 <code>etcd</code> と <code>Patroni</code> のパッケージ名を追加しました。

次のページに続く

表 3 – 前のページからの続き

Type	Description
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	pg_backup_api テストが誤った権限で実行され、失敗する問題を修正しました。
Bug Fix	pg_backup_api 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	`bdr.standby_slot_names` および `bdr.standby_slots_min_confirmed` チェックが bdr3 クラスターで不適切なスキーマを使用してい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 Patroni に渡され、「警告：予期しないパラメーターを削除しています」という問題を修正しました。
Bug Fix	`docker_images` の中間ベースイメージオプションを使用すると、結果のイメージ名が正しくない問題を修正しました。

2.3 Trusted Postgres Architect 23.22 release notes

リリース：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以下が含まれます。

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 ライセンスの下でソースをクローンできます。

2.4 Trusted Postgres Architect 23.21 release notes

リリース：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以下が含まれます。

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、EDB Repos 2.0 を使用し、それ以外の場合は PGDG が使用されるようになりました。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` を実行するときは、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	PGD サブスクライバー専用ノードの作成に関する問題を修正しました。TPA では、アップストリームインスタンスではなくレプリカに「subscriber-only」を設定する必要があります。

次のページに続く

表 5 – 前のページからの続き

Type	Description
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、コンテナにデプロイするときに適用外の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	<code>`/etc/edb`</code> のアクセス許可の問題を修正しました。展開された PGD5 クラスター内のデータノードに pgd-proxy ロールを追加した場合、pgd-proxy は pgd-proxy-config.yml を開くアクセス許可がないため、起動に失敗します。
Bug fix	厳密な umask が設定されている場合、 <code>`/var/log/postgres`</code> が不適切な権限 (0600) になる問題を修正
Bug fix	PEM に正しく登録された Barman インスタンスで <code>`tpaexec deploy`</code> を繰り返すと、PEM Agent Barman の構成が失われる問題を修正しました。

2.5 Trusted Postgres Architect 23.20 release notes

リリース：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以下が含まれます。

Type	Description
New feature	TPA は、新しいコマンド <code>`tpaexec reconfigure`</code> を実行して改訂された config.yml を生成し、 <code>`tpaexec upgrade`</code> 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 インストールに対して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	TPA は、RHEL 8 または 9 に Failover Manager 用の OpenJDK をインストールするときに、 <code>tzdata-java`</code> を明示的に追加するようになりました。これは、 <a href="#">`this OpenJDK bug` の回避策です。
Change	TPA は現在、AWS で利用可能な最新の Debian AMI（このリリースの時点で最新）を使用しています。
Change	config.yml 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> を自動的に実行します。
Bug fix	pgd-proxy ノードで pgd-cli を実行するときに、TPA が存在しないユーザーの使用を試みることができ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。

2.6 Trusted Postgres Architect 23.19 release notes

リリース：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以下が含まれます。

Type	Description
New feature	TPA では、加入者専用の 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 で、HTTP(S) HARP および PGD プロキシ正常性プローブの構成がサポートされるようになりました。
New feature	TPA では、M1 アーキテクチャの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であり、実稼働での使用はまだお勧めできません。
Enhancement	TPA では、常に最新バージョンを使用するのではなく、edb-pgd-proxy および edb-bdr-utilities 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 ライクなシステムでは、パッケージ選択コードが特定のパッケージに-dbgsymではなく-dbgを使用するようになりました。
Change	レプリケーションスロットを構成するときに、TPA は、primary_slot_name で有効な文字のみが使用されることを保証します。以前は、TPA はハイフンを含む可能性があるデフォルトとして inventory_hostname を使用していました。これらは現在、アンダースコアに置き換えられています。
Change	デフォルトのフェールオーバーマネージャーバージョンは現在 4.7 です。
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 を展開するときに 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失敗します。
Bug fix	Debian ライクなプラットフォームで誤った etcd サービス名が使用され、TPA が etcd を起動できない問題を修正しました。
Bug fix	TPA が RHEL 8 に etcd 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	configure 中に `Failed to commit files to git: b`メッセージが表示される問題を修正しました。
Bug fix	`generate_password: false`のときに TPA が Postgres ユーザーパスワードを誤って生成して上書きする問題を修正しました。
Bug fix	eu-west-1 以外のリージョンを使用している場合、AWS でのボリュームマップ作成がリージョンを考慮せずに失敗する問題を修正しました。

2.7 Trusted Postgres Architect 23.18 release notes

リリース：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以下が含まれます。

Type	Description
Enhancement	TPA は、初期レプリカ作成に repmgr の代わりに pg_basebackup を使用するようになりました。
Enhancement	TPA は、エアギャップ展開用のローカルリポジトリの作成を除き、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、`tpaexec test` の実行時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従来の 2ndQuadrant リポジトリを使用しようとする問題を修正しました。
Bug fix	TPA が必要な Barman インスタンスに pg_receivewal をインストールしない問題を修正しました。
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、デプロイが失敗する問題を修正しました。

2.8 Trusted Postgres Architect 23.17 release notes

リリース：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以下が含まれます。

Type	Description
Enhancement	configure コマンドに新しい `--pgd-proxy-routing` パラメーターを追加しました。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 プロキシルートを実地の場所内の書き込みリーダーにします。グローバルルーティングは、すべてのプロキシルートを実地の場所内で使用可能なすべてのデータノードの中から選択された単一の書き込みリーダーにします。
Change	configure コマンドから `--active-locations` パラメーターを削除しました。
Enhancement	TPA で Ubuntu 22.04 がサポートされるようになりました
Change	RHEL 7 および 8 に使用される AWS AMI を更新しました。

[次のページに続く](#)

表 9 – 前のページからの続き

Type	Description
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 commit 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 構成後に必要に応じて 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、それらがべき等になり、後続のデプロイメントでの etcd の不必要な再起動を回避します。

2.9 Trusted Postgres Architect 23.16 release notes

リリース：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以下が含まれます。

Type	Description
Change	デフォルトの PGD-Always-ON クラスターは、2つのデータノードと1つの監視ノードを含む関連付けられたサブグループを持つ1つの場所になりました。
Change	TPA は、デフォルトですべてのデータノードに pgd-proxy をデプロイするようになりました。
Enhancement	個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
Enhancement	TPA は、 <code>--data_nodes_per_location`</code> が偶数の場合に監視ノードを自動的に追加し、2つの場所のみでクラスターを指定すると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、すでに名前付けされている (<code>--location-names`</code> で) ロケーションを <i>witness-only</i> として指定しているため、パラメーター <code>--add-witness-only-location`</code> は <code>--witness-only-location`</code> に名前が変更されました。
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 の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。例、 <code>--postgresql 14</code> または <code>--edbpg 15</code>
Enhancement	サポートされているさまざまな EDB ソフトウェアリポジトリの処理とドキュメントの改善
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なくなりました
Bug fix	EDB フェールオーバーマネージャーがデフォルトで 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

次のページに続く

表 10 – 前のページからの続き

Type	Description
Bug fix	M1 アーキテクチャで pglogical が不必要にインストールされる問題を修正

2.10 Trusted Postgres Architect 23.15 release notes

リリース：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以下が含まれます。

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

2.11 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以下が含まれます。

Type	Description
Bug fix	PGD 3 および 4 構成で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 でマルチライン BDR DCS 構成を使用する（TPA-360、RT90034）

2.12 Trusted Postgres Architect 23.13 release notes

リリース：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以下が含まれます。

Type	Description
Bug fix	PGD-Always-ON と `--epas` で古い EDB リポジトリを有効にしないでください。
Bug fix	PGD-Always-ON と `--postgres-version 15` のエラーを修正。 該当なし

2.13 Trusted Postgres Architect 23.12 release notes

リリース：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以下が含まれます。

Type	Description
Feature	コミットスコープに基づいた Commit At Most Once (CAMO) 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 Extended リポジトリとパッケージのサポートを紹介します。
Enhancement	マルチリージョン AWS クラスター構成の暫定サポート。 マルチリージョンクラスターでは、VPC と VPC を手動でセットアップする必要があります。
Enhancement	`--active-locations` のプロキシルーティング (したがって、サブグループ RAFT) を自動的に有効にします。サブグループ RAFT をグローバルに有効にする configure オプションを削除します。
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 products/default/release リポジトリの tpa_2q_repositories への追加を抑制することを許可します。
Bug fix	利用可能な場合、ノードが bdr_child_group にサブスクライブすることを確認します。 ノードのサブスクライ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。
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 のバグにより、Postgres が一部のインスタンスで起動せず、他のインスタンスで起動した場合 (たとえば、構成の違いにより)、デプロイメントは失敗しませんでした。デプロイを続行すると、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 witness で `bdr.alter_node_replication_sets()` を呼び出さないでください。 これは、witness 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されます。
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 で言及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) を修正します。
Bug fix	ファイルの書き換えを回避するために、pgd-proxy 構成のエンドポイントを並べ替えます。 これには、次のデプロイ時に pgd-proxy の再起動が必要になる可能性があります (ただし、後続のデプロイでの将来の不要な書き換え/再起動を回避します)。
Bug fix	RH システムにローカルリポジトリから rsync をインストールする際のエラーを修正します。

次のページに続く

表 14 – 前のページからの続き

Type	Description
Bug fix	Debian システム上の 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 拡張機能が意図的に作成されている場合は、未使用であっても削除しないでください。
	該当なし

2.14 Trusted Postgres Architect 23.1 to 23.11 release notes

2.14.1 TPA 23.11

リリース：2023-01-31 ###重要な変更

- TPA-180 PGD-Always-ON アーキテクチャの実験的なサポートを導入（今年後半にリリース予定）。PGD-Always-ON アーキテクチャは、今後の BDR バージョン 5 を使用します。初期サポートは内部目的で追加され、今後のリリースで改善される予定です。###マイナーな変更
- TPA-349 依存関係のバージョンをバンプ暗号化バージョンを 38.0.4 から 39.0.0 にバンプ jq バージョンを 1.3.0 から 1.4.0 にバンプ
- TPA-345 ドキュメント内の TPAexec 参照を TPA に変更。製品を参照する際に、「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。

2.14.2 TPA 23.10

リリース：2023-01-04 ###マイナーな変更

- TPA-161 harp_manager_restart_on_failure 設定（デフォルトは false）を導入して、harp-manager systemd サービスの障害時のプロセス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します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を解除できない問題を修正します。
- TPA-305 Tower の事前生成されたシークレットに enterprisedb_password を追加
- TPA-306 PEM_PYTHON_EXECUTABLE を優先します、存在する場合、/usr/bin/python3 PEM 9.0 で の展開中の Python モジュール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 がデフォルトで bdr_database を監視するようにします

2.14.3 TPA 23.9

リリース：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、デプロイされているインスタンスではなく、コントローラー上のディレクトリの存在を正しくチェック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`」ディレクティブを削除します。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した。
- TPA-291 Barman インスタンスで `pg_receivewal` を取得するために Postgres サーバーパッケージをインストールするときに `postgres_package_version` 設定を尊重します。

2.14.4 TPA 23.8

リリース：2022-11-30 ### 主な変更

- TPA-18 サポート Ansible Tower 3.8 このリリースは、Ansible Tower 3.8 を介して bare クラスター（既存のサーバーを使用）での `deploy.yml`（のみ）の実行をサポートします。Tower サーバーに TPAexec をインストールし、`tpaexec setup` を実行して仮想環境を作成します。Tower テンプレートを使用して TPAexec Playbook を実行できます。 `--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。###マイナーな変更
- 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、以前と同じように続行されます。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。

2.14.5 TPA 23.7

リリース：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ていました。このリリースでは、代わりに `tpaexec setup --use-community-ansible` で Ansible のコミュニティリリースを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 を使用します。これは将来のリリースで変更されます。2ndQuadrant/ansible のサポートは終了し、Ansible が新しいデフォルトになります。###マイナーな変更
- TPA-209 Postgres 15 のリリース後、`--postgres-version 15` を有効な `tpaexec configure` オプションとして受け入れます

- TPA-226 `--hostnames-from` ファイルで IP アドレスを受け入れます以前は、`tpaexec configure` に渡されるファイルには、1 行に 1 つのホスト名が含まれる必要がありました。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。（独自の `--hostnames-from` ファイルを指定すると、ホスト名は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の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（デフォルトのモジュールは常に `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する
- TPA-82 BDR-internal `${group_name}_ext` レプリケーションセットの削除を避ける
- TPA-247 AWS での「`'str object' has no attribute 'node_dsn'`」エラーを修正しました コードは `hostvars[hostname]` を中間変数に割り当てなくなり、後で通常の `dict` のように動作することを想定しています（これは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 でのリグレーションが修正されますが、コードベース全体で他の同様のエラーも修正されます。
- TPA-232 「リンク中にエラーが発生しました: [Errno 17] ファイルが存在します」エラーが発生した、インベントリ内の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解消します
- TPA-252 すべての BDR ノードが `witness-only` レプリケーションセットにパブリッシュするコードを復元するこのコードブロックは、TPA-193 のために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。

2.14.6 TPA 23.6

リリース：2022-09-28 ###重要な変更

- TPA-21 AWS デプロイメントに（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。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た `aws` デプロイメントを修正。このようなデプロイは、プロビジョニング時に失敗していましたが、期待どおりに動作するようになりました。
- TPA-189 デプロビジョニング時に `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 と PEM のバージョン設定を許可します。これにより、特定のバージョンが指定されていて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が修正されます。
- TPA-172 EFM を正しいホストセットにインストールします。EFM は、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。PEM サーバーなど、他の理由で postgres がインストールされているサーバーにはインストールしないでください。
- 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を回避できます。

2.14.7 TPA 23.5

リリース：2022-08-23 ###重要な変更

- TPA-81 Ubuntu 22.04 Jammy 用の tpaexec および tpaexec-deps パッケージを公開
- 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。

2.14.8 TPA 23.4

リリース：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時のロケール検出の問題を修正します。最初のブートを完了できません。これは、SSH キーネゴシエーションの問題と、デプロイ中にディスクが見つからない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

2.14.9 TPA 23.3

リリース：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。最初のものは HARP harp_db_request_timeout を dcs request_timeout と同様に設定しますが、データベース接続用で、2 番目の harp_ssl_password_command は難読化を解除するコマンドを指定します
- 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した。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 ソフトウェアには、ユーザーロケールを en_US.UTF-8 に設定する要件があります。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することもできま

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て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るようになります。デフォルトのロケールは `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 のログ設定を追加します。指定されたログサーバーは、これらのサービスの `/var/log/hosts` ディレクトリに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たロックタイムアウト構文の選択に関する `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、Debian および Rocky Linux のいくつかの不足している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け取ります。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-repo` 構成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上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、ローカルリポジトリ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に失敗しました。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るようになります
- 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、これにより問題は発生しませんでした。現在削除されている最初の引数は 2 番目の引数によってオーバーライドされるためです。
- TPA-108 Postgres がソースからインストールされている場合、`postmaster` 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、ソースから `postgres` をインストールしている SELinux 対応 OS を使用するクラスターは、`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れないため、Postgres の再起動に失敗しました。これは、AWS で最近採用された、SELinux が有効でデフォルトで `enforcing` モードの Rocky Linux イメージを使用したテストで発見されました

2.14.10 TPA 23.2

リリース：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可能。
- 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 SSL 証明書を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 *Etc*d 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 ロジカルスタンバイ (’ *bdr*’ +’ *readonly*’ ロール) ブロンズレイアウトの場合のみ：BDR Witness (’ *bdr*’ +’ *witness*’ ロール) *ゴールドレイアウトの場合のみ：Barman (’ *barman*’ ロール) クレジット：Gianni Ciolli gianni.ciolli@enterprisedb.com ### マイナーな変更

- `configure` 引数 `--2q` を `--pgextended` に置き換えて、製品ブランドの変更を反映します。既存の構成は期待される動作を維持します。
- Debian 古い安定版に付属するバージョン 18 の `docker` 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に関するエラー報告を改善します。
- CLI ヘルプドキュメントに欠落しているコマンドをいくつか追加。
- `configure` コマンドのエラー報告を改善。
- BDR 5 をソースからビルドするための初期サポートを追加。クレジット：Florin Irion florin.irion@enterprisedb.com
- Postgres の古いバージョンから EDB 製品への移行に関する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、`etcd` のパッケージが見つからない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依存関係の選択に影響を与える問題を修正

2.14.11 TPA 23.1

リリース：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（そうしないと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（たとえば、`postgres_package_version` を設定することにより）、`tpaexec deploy` を実行すると、インストールされているパッケージを潜在的にアップグレードできました。ただし、これは安全、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でした。特に、展開中のパッケージのアップグレード後、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しません。最初の展開では、必要なすべてのパッケージ（設定されている場合は特定のバージョン、または利用可能な最新のパッケージ）をインストールし、後続の実行ではパッケージがインストールされ、それ以上何も行いません。これは、予測可能で安全な新しいデフォルトの動作です。

コンポーネントを更新する必要がある場合は、`tpaexec update-postgres` を使用してください。このリリースでは、コマンドは、BDR や `pglogical` などの Postgres および Postgres 関連のパッケージ、および

HARP、pgbouncer、etcd（特定のクラスターに該当する場合）などの他の特定のコンポーネントを更新できます。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します。

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`」を実行します将来の展開で
- `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、エアギャップ/切断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 には `--harp-consensus-protocol <etcd|bdr>` 構成オプションが必要です-Always-ON クラスター TPAexec はここでデフォルト値を提供しなくなりました位置を使用して、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 `etcd` の以前のデフォルトを使用する既存のクラスターには影響しません ### マイナーな変更

*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で EFM 用に `openjdk-8` の代わりに `openjdk-11` をインストールします

- `harp_log_level` 設定（例、`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（情報）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 のリストではなく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

BDR のローカル（Unix ソケット）DSN から余分な接続属性（例、`ssl`）を省略 `harp-manager` の DCS ### バグ修正

*構成が変更された場合、`harp-manager` と `harp-proxy` が再起動され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-proxy` 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 対応の BDR4 クラスターでサーバーが再起動します（とにかくストリーミングが有効になっていると動作しませんでした）。

第3章 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があり、したがって EDB リポジトリにアクセスできる場合は、次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、
[配布サポート](#) を参照してください。

注釈: さまざまなことが失敗するため、システムに正しい日付と時刻が設定されていることを絶対に確認してください。 `sudo ntpdate pool.ntp.org` などのネットワーク時間を使用することをお勧めします

3.1 クイックスタート

EDB Repos 2.0 にログイン

あなたのトークンを取得します。次に、次のコマンドを実行し、`<your-token>` をトークンに置き換え、`<your-plan>` を、サブスクライブしている EDB プランに応じて次のいずれかで置き換えます。enterprise、standard、community360、postgres_distributed。

3.1.1 Debian または Ubuntu にリポジトリを追加し、TPA をインストールする

```
curl -sLf https://downloads.enterprisedb.com/<your-token>/<your-plan>/setup.deb.sh |  
→ sudo -E bash  
sudo apt-get install tpaexec
```

3.1.2 RHEL、Rocky、AlmaLinux、または Oracle Linux にリポジトリを追加し、TPA をインストールします

```
curl -sLf https://downloads.enterprisedb.com/<your-token>/<your-plan>/setup.rpm.sh |  
→ sudo -E bash  
sudo yum install tpaexec
```

3.1.3 追加の依存関係をインストールする

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

3.1.4 インストールを確認します（通常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は以下に記載されています。

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 を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は適しています。TPA は *deploying to Docker containers* をサポート

自分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は、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のためにチームが使用できるようになるため、これをお勧めします。また、これは、TPA の 1 つのコピーを更新するだけでよく、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よいことを意味します。

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、最初にそれを提供する EDB リポジトリにサブスクライブする必要があります。リポジトリの推奨ソースは EDB Repos 2.0 です。

EDB Repos 2.0 にログイン

あなたのトークンを取得します。次に、次のコマンドを実行し、<your-token> をトークンに置き換え、<your-plan> を、サブスクライブしている EDB プランに応じて次のいずれかで置き換えます。enterprise、standard、community360、postgres_distributed。

3.3.1 Debian または Ubuntu にリポジトリを追加する

```
curl -1sLf https://downloads.enterprisedb.com/<your-token>/<your-plan>/setup.deb.sh |  
→sudo -E bash
```

3.3.2 RHEL、Rocky、AlmaLinux、または Oracle Linux にリポジトリを追加します

```
curl -1sLf https://downloads.enterprisedb.com/<your-token>/<your-plan>/setup.rpm.sh |  
→sudo -E bash
```

または、従来の 2ndQuadrant リポジトリから TPA を取得できます。これを行うには、EDB カスタマーサポートポータルにログインし、
[products/tpa/release](#) にサブスクライブします

Support/Software/Subscriptions の下にサブスクリプションを追加し、指示に従ってシステムでリポジトリを有効にします。

これらのリポジトリのいずれかを有効にしたら、次のように TPA をインストールできます。

3.3.3 Debian または Ubuntu にインストールする

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

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

```
sudo yum 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 3.6 以降など）がインストールされていることも確認します。

ここでは、どのコマンドに root 権限が必要かを示すためにのみ、`sudo` について言及します。他の方法を使用して、ルートとしてコマンドを実行できます。

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` 中の後続の許可エラーを回避してください

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

/opt/EDB/TPA に書き込むため、これを root として実行する必要がありますが、このプロセスは、インストールしたシステム全体の 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
```

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

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。このような環境での TPA の使用方法については、

Managing clusters in a disconnected or air-gapped environment をご覧ください

3.5.1 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は、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。これを達成する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は、yumdownloader（RHEL 風）または apt download（Debian 風）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。

これが不可能な場合は、EDB サポートに連絡してください。サブスクリプションに適したダウンロードリンクまたは手順を提供します。

3.5.2 Python パッケージインデックスにアクセスせずにインストールする

tpaexec setup を実行すると、通常、Python パッケージインデックスから Python パッケージがダウンロードされます。あなたの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。パッケージインデックスが利用できない場合は、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可能）は、ダウンロードされるすべてのものをバンドルしているため、ネットワークにアクセスせずに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。

3.6 TPA のインストールの確認

上記の手順をすべて完了したら、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 インストールを使用する準備が整います。

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、以下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します
2. 最新の `tpaexec-deps` パッケージをインストールします（必要な場合。上記を参照）
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がインストールされます。そうでない場合は、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たびに、`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では、これを厳密に必要としない場合がありますが、他のリリースではこれがないと動作しません。

3.8 Ansible コミュニティのサポート

TPA はデフォルトで `ansible` のコミュニティディストリビューションを使用するようになりました。`--use-2q-ansible` オプションを `tpaexec setup` に渡すことで、2ndQuadrant / `ansible` フォークを引き続き使用できます。将来の TPA リリースでは、2ndQuadrant `ansible fork` のサポートが削除されます。

ほとんどのユーザーにとって、これは違いはありません。ただし、2ndQuadrant `ansible` で `--skip-tags` を使用している場合、これはサポートされていないことに注意してください 2ndQuadrant `ansible` のサポートが削除される前に、すべての `ansible` バージョンと互換性のあるタスクをスキップする代替手段が提供されます。

第4章 Open source TPA

4.1 Trusted Postgres Architect (TPA) 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あり、Ansible を使用して、EDB の推奨に従って Postgres クラスターをデプロイします。

TPA は、Postgres の展開とサポートでの長年の苦労した経験に基づいた EDB に続く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、実稼働環境と同様に、クイックテストベッドのセットアップにも適用できます。

4.2 次のステップ

- *Installing TPA from source*
- *Deploying your first cluster*
- [TPA's full documentation online](#)

4.3 TPA オープンソースの FAQ

4.3.1 EDB の顧客でなくても、これを使用できますか？

はい、TPA は GPLv3 ライセンスの下で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、または EDB のプロプライエタリ製品、または組み合わせで構成されるクラスターのデプロイをサポートします。

4.3.2 問題を報告できますか？

はい、EDB のお客様は、サポートにお問い合わせください。それ以外の場合は、GitHub Issue を開いてください。

4.3.3 貢献できますか？

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 Issue をオープンしてください。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くなります。

第5章 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 のお客様は、ソフトウェアの EDB サポートを受けるために *install TPA from packages* する必要があります。

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（Python 3.6 以降など）をインストールするか、ソースと *run TPA in a Docker container* をダウンロードする必要があります。（どちらの方法でも、Linux および macOS では正常に動作します。）

5.1 クイックスタート

まず、さまざまな依存関係 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`

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たら、リポジトリをクローンできます。

```
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 でチェックアウトしたい場合：

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

新しく作成したクローンにある bin ディレクトリを、次のコマンドを使用してパスに追加します。

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

この行を .bashrc ファイル（または好みの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 環境を作成できます。

```
tpaexec setup
```

これにより、TPA が将来使用する 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ている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。

5.3 依存関係

5.3.1 Python 3.6+

TPA には、最新のディストリビューションのほとんどで利用可能な Python 3.6 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、好きなバージョンの 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 3.9.0
Downloading Python-3.9.0.tar.xz...
- > https://www.python.org/ftp/python/3.9.0/Python-3.9.0.tar.xz
Installing Python-3.9.0...
Installed Python-3.9.0 to /home/ams/.pyenv/versions/3.9.0

$ pyenv local 3.9.0
$ pyenv version
3.9.0 (set by /home/ams/.pyenv/.python-version)

$ pyenv which python3
/home/ams/.pyenv/versions/3.9.0/bin/python3
$ python3 --version
3.9.0
```

まだ `pyenv` を使用していない場合は、.bashrc の PATH に `pyenv` を追加し、
`pyenv documentation` で説明されているように `eval "$(pyenv init -)"`

を呼び出すことを忘れないでください。

5.3.2 仮想環境オプション

デフォルトでは、`tpaexec setup` は組み込みの 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化します。

`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` ロケーションに固執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、たとえば、次の行を `.bashrc`（または他のシェル起動スクリプト）に追加して、環境変数 `TPA_VENV` をその場所に設定する必要があります。

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


第6章 Creating a cluster

6.1 A First Cluster Deployment

この短いチュートリアルでは、単純な *ML architecture* デプロイメントをローカルの Docker インストールにデプロイする方法について説明します。最終的には、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、バックアップノードが構成され、探索する準備が整います。

この例では、Ubuntu システムで TPA を実行しますが、考慮事項はほとんどの Linux システムで似ています。

6.1.1 TPA のインストール

EDB のお客様は、

EDB Repo instructions に従ってください。EDB のリポジトリから TPA パッケージを直接インストールします。

TPA のオープンソースユーザーは、Github.com からダウンロードできる *instructions on how to build from the source* があります。

それらのガイドに従って、ここに戻ります。

6.1.2 Docker のインストール

前述のように、デプロイ例を Docker にデプロイします。Docker がすでにインストールされている場合を除き、セットアップする必要があります。

Debian または Ubuntu では、次を実行して 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。

次のコマンドを使用して、ユーザーを docker グループに追加します。

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

6.1.3 CgroupVersion

現在、TPA では、システムで Cgroups バージョン 1 を構成する必要があります。

実行：

```
mount | grep cgroup | head -1
```

出力に `tmpfs` への参照が表示されない場合は、cgroups v2 を無効にする必要があります。

実行：

```
echo GRUB_CMDLINE_LINUX=systemd.unified_cgroup_hierarchy=false | sudo tee /etc/
→default/grub.d/cgroup.cfg
```

適切な変更を行うには、Grub を更新し、次のコマンドでシステムをリブートします。

```
sudo update-grub
sudo reboot
```

警告： ユーザーに Docker デーモンと話す機能を与えると、ユーザーは簡単に Docker ホストで root を取得できます。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ようにする必要があります。

6.1.4 TPA で構成を作成する

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` コマンドを介してほとんどの作業を行います。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提供して、大まかに言って、展開するものを選択することです。ここに私たちの `tpaexec configure` コマンドがあります：

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

これにより、*M1 architecture* を持つ `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、および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 の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 [認証情報に関する奇妙な AWS エラー](#)や、オペレーティングシステムのプロビジョニングにスキップし、構成済みの Linux ホストへのソフトウェアのインストールに直接進む

bare(-metal servers) があります。

--postgresql 15 を使用して、Community Postgres バージョン 15 を使用するように TPA に指示します。ここでは、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、

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行うようにデプロイメントを構成するように TPA 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、明らかに、コマンドラインで何も起こりません。ただし、demo というディレクトリが作成され、新しい展開の青写真である config.yml ファイルを含むいくつかのファイルが含まれていることがわかります。

デプロイメントのプロビジョニング

これで、新しいデプロイ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：

```
tpaexec provision demo
```

構成の展開の準備に必要なさまざまな操作を通じて TPA が機能することがわかります。

展開中

プロビジョニングが完了したら、デプロイ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、必要なすべての構成を実行します。

```
tpaexec deploy demo
```

構成の展開に必要なさまざまな操作を TPA が実行することがわかります。

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、新しくデプロイした構成をすばやくテストできます。

```
tpaexec test demo
```

接続する

psql プロンプトを表示する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 1 つにログインすることです。実行

```
tpaexec ping demo
```

展開内のすべての接続可能なホストに ping を実行します。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、コマンド `docker exec -it uptake /bin/bash` を使用し、適切なホスト名に置き換えます。

すべてのタイプの TPA 展開で動作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、私たちの構成は `demo` と呼ばれるので、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 はすでにそのユーザーを .pgpass ファイルで構成しているため、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、私たちはデータベースに接続されています。

コンテナの 1 つに SSH 接続せずにホストシステムから接続できます。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
```

取り込みに接続するため、IP アドレスは 172.17.0.11 です。

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ているパスワードを取得します。

```
tpaexec show-password demo postgres
a9LmI1X^uM0pPoEnLuRdL%L$0RQak3om
```

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、次を実行できます。

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

先ほど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)
Type "help" for help.

postgres=#
```

これで、Docker ホストから、TPA でデプロイされた Docker コンテナの 1 つで実行されている Postgres に

接続されました。

6.2 Cluster configuration

`tpaexec configure` コマンドは、プロビジョニング/デプロイ/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。

6.2.1 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` を含む最初のコミットを行います。

このコマンドは、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オプション）も受け入れますが、デフォルトは賢明であり、すぐに使用できることを目的としています。生成された `config.yml` を読んで、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。（*configuration settings that affect the deployment* の概要はこちら。）

`config.yml` を完全に手で書くことは可能ですが、生成されたファイルを編集する方がはるかに簡単です。

6.2.2 構成オプション

最初の引数は、クラスターディレクトリ、例、`speedy` または `~/clusters/speedy` である必要があります（クラスターにはどちらの場合も `speedy` という名前が付けられます）。すべてのクラスターを共通のディレクトリ（上記の例の `~/clusters` など）に保持することをお勧めします。

アーキテクチャ、*M1* または *BDR-Always-ON* を選択するには、次の引数を `--architecture <name>` にする必要があります。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 flavour and version of Postgres を指定する必要があります。

上記の引数は常に必須です。ここで説明する残りのオプションは、上記の例のように、安全に省略できます。デフォルトは使用可能な結果につながります。

一般的なオプションのリストについては、`tpaexec help configure-options` を実行してください。

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って、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、以下に説明する汎用オプションだけでなく、いくつかの一意のオプションを受け入れます。

たとえば、M1 では `--num-cascaded-replicas 3` を使用して、3つのカスケードレプリカでクラスターを作成できます。受け入れる（または、場合によっては必要とする）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。

プラットフォームオプション

次に、`--platform <name>` を使用して、[認証情報に関する奇妙な AWS エラー](#) または *bare(-metal servers)* などのプラットフォームを選択できます。

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。そうでない場合、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します。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](#) でなければなりません

そして [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、デフォルトは現在 aws です。

注：TPA は、異なるプラットフォーム上のインスタンスを持つクラスターの作成を完全にサポートしていますが、`tpaexec configure` は現在そ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（AWS タグなどのプラットフォーム固有の手段によって）責任者の名前に関連付けます。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、「Firstname Lastname」、またはあなたを一意に識別する他のものを使用できます。）

地域

`--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で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、複数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` は現在そのような構成を生成できません。複数のリージョンを指定するには、`config.yml` を編集する必要があります。

ネットワーク構成

デフォルトでは、各クラスターは、選択したアーキテクチャに応じて、CIDR 範囲 10.33.0.0/16 からランダムに選択された多数の/28 サブネットで構成されます。

--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。

注：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、[認証情報に関する奇妙な AWS エラー](#) ドキュメントを参照してください。

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

--no-shuffle-subnets を指定して、ランダム化なしでネットワーク CIDR 範囲の先頭からサブネットを割り当てます（例、10.33.0.0/28、10.33.0.16/28 など）。

--exclude-subnets-from <directory> を指定して、既存のクラスター config.yml 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。

注：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。

インスタンスタイプ

インスタンスタイプを選択するには--instance-type <type> を指定します。

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

ディスクスペース

--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最小サイズが必要な場合があります。）

--postgres-volume-size <size> および--barman-volume-size <size> オプションは、Postgres および Barman の個別のボリュームをサポートするアーキテクチャおよび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 つの名前でファイルからホスト名を選択します。ファイルには、少なくともクラスター内のインスタンスと同じ数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。

`--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ないようにします。この場合、それらは `hostnames` ファイルで見つかった順序で割り当てられます。これは、`hostnames` ファイルが明示的に指定された場合のデフォルトです。

ホスト名に使用できるのは、文字 (a~z)、数字 (0~9)、および「-」のみです。選択したプラットフォームによっては、FQDN の場合があります。ホスト名は小文字にする必要があります。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。

6.2.3 ソフトウェアの選択

配布

`--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って、使用可能なディストリビューションと、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できるはずです。

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

2ndQuadrant および EDB リポジトリ

TPA は、サブスクリプションを通じてアクセスできる 2ndQuadrant または 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、2ndQuadrant パブリックリポジトリ (サブスクリプションは必要ありません) をインストールし、アーキテクチャに必要な製品リポジトリ (PGD リポジトリなど) を追加します。

TPA が 2ndQuadrant および EDB リポジトリを使用する方法のより詳細な説明が利用可能です

TPA をインストールする場所

`--2Q-repositories source/name/maturity …` または `--edb-repositories repository …` を指定して、2ndQuadrant パブリックリポジトリに加えて、各インスタンスにインストールする 2ndQuadrant または EDB リポジトリの完全なリストを指定します。

EDB リポジトリが指定されている場合、2ndQuadrant リポジトリは無視されます。

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、名前付きリポジトリを構成します。

これらのオプションを使用するには、`tpaexec` を実行する前に `export TPA_2Q_SUBSCRIPTION_TOKEN=xxx` または `export EDB_SUBSCRIPTION_TOKEN=xxx` を行う必要があります。2ndQuadrant ポータルの左側メニュー「Company info」→「Company」から 2ndQuadrant トークンを取得できます。enterprisedb.com/repos から EDB トークンを取得できます。

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配布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上の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することができます。

[ローカルリポジトリのサポート](#) に関するページに詳細があります。

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 (EPAS) バージョン 11~16 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。次に例を示します。

- `--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 EPAS 15 を非 Redwood モードでインストールします

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

EDB Postgres Extended または Postgres Advanced Server をインストールするには、有効な *EDB repository subscription* 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.pgdg90+1
- 4.--pglogical-package-version '2.2.0*'
- 5.--bdr-package-version '3.0.2*'
- 6.--pgbouncer-package-version '1.8*'

apt または yum が受け入れる任意の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しばしば必要になります。

--extra-packages p1 p2 ...または--extra-postgres-packages p1 p2 ...を指定して、追加の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。（前のリストで Postgres に依存するパッケージに言及した場合、Postgres はまだインストールされないため、インストールは失敗します。）引数は、変更なしで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、指定されたパッケージをインストールできない場合、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 に永続的に追加されたときに*_package_version でワイルドカードを使用すると、RHEL8 以降（または RockyLinux などの dnf を使用する派生 OS）を使用するノードで tpaexecdeploy 中にインストールされたソフトウェアが予期せず更新される可能性があることに注意してください。パッケージが既にインストールされている既存のクラスターで deploy が実行されると、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* に設定されていた場合、ansible はこれをインストールされている PGD バージョンと照合できません。パッケージがインストールされていないと想定し、次のパッケージで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。構成されたリポジトリ内の同じ名前、例 3.7。

この制限は 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されてインストールされます。postgres の代わりに 2ndqpostgres を使用して、2ndQPostgres をビルドおよびインストールします。デフォルトでは、これにより適切な REL_nnn_STABLE ブランチがビルドされます。

`--install-from-source 2ndqpostgres pglogical3 bdr3` を使用して 3 つすべてのコンポーネントをソースからビルドしてインストールするか、`--install-from-source pglogical3 bdr3` を使用して 2ndQ-Postgres のパッケージを使用できますが、ソースから pglogical v3 および PGD v3 をビルドしてインストールします。デフォルトでは、これにより pglogical および PGD の master ブランチがビルドされます。

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。たとえば、`--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で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。

6.2.4 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、結果が YAML データ構造としてロードされ、config.yml の生成に使用される引数に再帰的にマージされます（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）。このプロセスは、指定された追加の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から見え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、そうでなければ config.yml にある cluster_tags および cluster_vars を拡張します。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。言い換えれば、config.yml の特定のサブキーを設定またはオーバーライドできますが、cluster_tags または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。cluster_vars および instance_defaults を環境の一般的な設定で拡張することが最も役立ちます。

とはいえ、このメカニズムは制限を強制するものではないため、十分に注意してください。オーバーライドの有無にかかわらず 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。

6.2.5 アンシブルタワー

--use-ansible-tower および --tower-git-repository オプションを使用して、Ansible Tower でのデプロイメントに適したクラスターを作成します。詳細は *TPA and Ansible Tower/Ansible Automation Controller* を参照。

6.2.6 git リポジトリ

デフォルトでは、git リポジトリはクラスターの名前が付けられた最初のブランチで作成され、コミットメッセージで使った構成オプションを使用して単一のコミットが行われます。\$PATH に git がない場合、tpaexec はエラーを発生しませんが、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あり、まだ存在しない場合は tpaexec provision によって後で作成されることに注意してください。)

6.2.7 例

次のコマンドを実行するとどうなるかを見てみましょう。

```
[tpa]$ tpaexec configure ~/clusters/speedy --architecture M1 \
--num-cascaded-replicas 2 --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]$
```

出力がないため、エラーはありませんでした。クラスターディレクトリが作成され、設定されました。

```
$ ls ~/clusters/speedy
total 8
drwxr-xr-x 2 ams ams 4096 Aug  4 16:23 commands
-rw-r--r-- 1 ams ams 1374 Aug  4 16:23 config.yml
lrwxrwxrwx 1 ams ams   51 Aug  4 16:23 deploy.yml ->
/home/ams/work/2ndq/TPA/architectures/M1/deploy.yml
```

クラスター構成は config.yml にあり、その近隣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: {}

cluster_rules:

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

-   cidr_ip: 10.33.76.176/28
    from_port: 0
    proto: tcp
    to_port: 65535

-   cidr_ip: 10.33.148.240/28
    from_port: 0
    proto: tcp
    to_port: 65535
ec2_ami:
  Name: debian-10-amd64-20210721-710
  Owner: 136693071363
ec2_instance_reachability: public
ec2_vpc:
  us-east-1:
    Name: Test
    cidr: 10.33.0.0/16

cluster_vars:
  enable_pg_backup_api: false
  failover_manager: repmgr
  postgres_flavour: postgresql
  postgres_version: 14
  preferred_python_version: python3
  use_volatile_subscriptions: false

locations:

-   Name: main
    az: us-east-1a
    region: us-east-1
```

(次のページに続く)

(前のページからの続き)

```

subnet: 10.33.76.176/28

- Name: dr
  az: us-east-1b
  region: us-east-1
  subnet: 10.33.148.240/28

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

instances:

- Name: upsets
  backup: kayak
  location: main
  node: 1
  role:
    - primary

- Name: zebra
  location: main
  node: 2
  role:
    - replica
  upstream: upsets

- Name: kayak
  location: main
  node: 3
  role:
    - barman

```

(次のページに続く)

```

- log-server
- monitoring-server
- witness
upstream: upsets
volumes:
- device_name: /dev/sdf
  encrypted: true
  vars:
    volume_for: barman_data
  volume_size: 128
  volume_type: gp2

- Name: queen
  location: dr
  node: 4
  role:
  - replica
  upstream: zebra

- Name: knock
  location: dr
  node: 5
  role:
  - replica
  upstream: zebra

```

次のステップは、*tpaexec provision* を実行することです

または、*the cluster as a whole* の構成をカスタマイズする方法または
individual instance を構成する方法の詳細を学んでください。

6.3 tpaexec provision

Provision 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（*MI* など）とプラットフォーム（[認証情報に関する奇妙な AWS エラー](#)

など）の両方によって異なります。

プロビジョニング段階の最後に、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作成されます。TPA は ssh を介してアクセスできます（ルートに sudo を使用）。

6.3.1 前提条件

クラスターをプロビジョニングする前に、*tpaexec configure* でクラスター構成を生成する必要があります (および *config.yml* を編集して、必要に応じて構成を微調整します)。

追加のプラットフォーム依存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。

6.3.2 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:quirk)
changed: [localhost] => (item=us-east-1:keeper)
changed: [localhost] => (item=us-east-1:zealot)
changed: [localhost] => (item=us-east-1:quaver)
changed: [localhost] => (item=us-east-1:quavery)
...

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

PLAY RECAP *****
-> *****
localhost                : ok=128  changed=20  unreachable=0  failed=0

real    2m19.386s
user    0m51.819s
sys     0m27.852s
```

このコマンドは、多く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で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 の実行に進み、ソフトウェアをインストールして構成できます。

6.3.3 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の責任を負う人間に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 EC2 インスタンスに、`tpaexec provision` を実行しているユーザーのログイン名が所有しているものとしてタグ付けします。

`--owner <name>` を指定して名前を変更します（たとえば、ユーザー名が `postgres` や `ec2-user` などの一般的なものである場合）。イニシャル、「名姓」、またはその他のものを使用して、個人を一意に識別できます。

指定した他のオプションはすべて Ansible に渡されます。

6.3.4 インスタンスへのアクセス

プロビジョニングが完了したら、インスタンスに SSH できるはずです（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なるまで少し遅延します）。上記の出力に示すように、`tpaexec` は使用する `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 quirk
    User admin
    HostName 54.227.207.189

Host keeper
    User admin
    HostName 34.229.111.196

Host zealot
    User admin
    HostName 18.207.108.211

Host quaver
    User admin
    HostName 54.236.36.251

Host quavery
    User admin
    HostName 34.200.214.150
[tpa]$ ssh -F ssh_config quirk
Linux quirk 4.9.0-6-amd64 #1 SMP Debian 4.9.82-1+deb9u3 (2018-03-02) x86_64
```

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the

(次のページに続く)

(前のページからの続き)

```
individual files in /usr/share/doc/*/copyright.
```

```
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
```

```
Last login: Sat Aug  4 12:31:28 2018 from 136.243.148.74
```

```
admin@quirk:~$ sudo -i
```

```
root@quirk:~#
```

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、必要なだけ待ちます。デプロイを開始する前に、インスタンスが起動するのを待ったり、インスタンスに ssh でログインしたりする必要はありません。

6.3.5 生成されたファイル

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

```
[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
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
```

これは、`id**` ファイル（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 `thehostkeys/` の署名）を参照します。

`vars.json` ファイルは、`--cached` による後続の呼び出しで `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ファイル、および `config.yml` のグループ変数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]
quirk ansible_host=54.227.207.189 node=1 platform=aws
```

(次のページに続く)

(前のページからの続き)

```

keeper ansible_host=34.229.111.196 node=2 platform=aws
zealot ansible_host=18.207.108.211 node=3 platform=aws
quaver ansible_host=54.236.36.251 node=4 platform=aws
quavery ansible_host=34.200.214.150 node=5 platform=aws

[tpa]$ cat inventory/group_vars/tag_Cluster_speedy/01-speedy.yml
cluster_name: speedy
cluster_tag: tag_Cluster_speedy
postgres_version: 15
tpa_version: v23.10-22-g30c1d5ea
tpa_2q_repositories: []
vpn_network: 192.168.33.0/24

[tpa]$ cat inventory/host_vars/zealot/02-topology.yml
role:

- barman

- log-server

- openvpn-server

- monitoring-server

- witness
upstream: quirk

```

config.yml の変数を変更して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。設定を変更しない場合、何もしません。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ずに新しいインスタンスが起動されます。

6.4 tpaexec deploy

デプロイとは、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して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。

展開段階の最後に、Postgres は repmgr、Barman、pgbouncer などの他のコンポーネントとともに起動して実行されます（選択したアーキテクチャによって異なります）。

6.4.1 前提条件

`tpaexec deploy` を実行する前に、`tpaexec configure` 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。

デプロイする前に、`export TPA_2Q_SUBSCRIPTION_TOKEN=xxx` して、サブスクリプションが必要な 2ndQuadrant リポジトリを有効にする必要があります。

Debian または *Ubuntu* にリポジトリを追加し、*TPA* をインストールする 自分自身に使用したサブスクリプショントークンを使用できます。これを忘れた場合、エラーメッセージがすぐにあなたに思い出させます。

6.4.2 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との間のレイテンシー、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）。展開中に少なくとも 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。

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

`deploy` 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、変更されずに Ansible に渡されます（例、`-v`）。

Ansible に精通している人は、時折赤い「failed」タスク出力がスクロールすることに気になるかもしれません。その後すぐにプロセスが停止しない場合、エラーは重要ではなく、コードは自動的に回復します。

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

6.4.3 選択的展開

`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、デプロイメント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デプロイメントが実行されますが、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` の設定は、TPA でサポートされていない Ansible の `--limit` オプションを使用する代わりに推奨される方法です。）

6.4.4 deploy.yml

展開プロセスはアーキテクチャ固有です。これが、さまざまな *configuration settings that affect the deployment* の概要です。Ansible Playbook 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手順をたどることができます。

`config.yml` とは異なり、`deploy.yml` は編集するように設計されていません（通常はアーキテクチャディレクトリへのリンクです）。デプロイメントプロセスを拡張して独自の Ansible タスクを実行する場合でも、

creating hooks

までに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での将来の実装の変更から保護されます。

6.5 tpaexec test

次に、デプロイ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、インストールを検証します。このステージの終わりには、完全に機能するクラスターがあります。

`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。

6.5.1 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` に再度記録されます。

このコマンドが成功すると、クラスターは機能します。

おめでとうございます。

第7章 Architectures

7.1 PGD-Always-ON

EDB Postgres Distributed 5 は常時オン構成で、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は、EDB Postgres Distributed 5 での使用にのみ有効であり、*Configuring EDB Repos 2.0 repositories* へのサブスクリプションが必要です。

7.1.1 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
```

次の表では、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> Postgres flavour and version (e.g. <code>--postgresql 15</code>)	<code>`PGD-Always-ON`</code> に設定する必要があります 有効な <i>tpaexec-configure</i> 。
<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>	ロケーション名のスペース区切りリスト。ロケーションの数は、提供された名前の数と同じです。	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 でなければなりません。	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-proxy の http(s) api エンドポイントを有効にして、プロキシの状態のプローブを許可します。	Disabled by default.

PGD-Always-ON 構成の詳細

PGD-Always-ON クラスタは、多数の、好ましくは奇数の場所を含み、それぞれが同じ数の、好ましくは奇数のデータノードを有する。--location-names を指定しない場合、デフォルトでは 3 つのデータノードで単一の場所を使用します。

クラスタの場所名は --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意味をなす方法で、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保持されます (Witness には、データノードと比較して最小限のハードウェア要件があります)。

ロケーションが 2 つしかないクラスターは、ロケーションの 1 つで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機能を完全に失います。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が含まれず、監視ノードのみが含まれ、再び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、場所の 1 つを証人として指定します。

デフォルトでは、(すべての場所の) すべてのデータノードも、接続ルーティングのために PGD-Proxy を実行します。代わりに別の PGD-Proxy インスタンスを作成するには、 `--add-proxy-nodes-per-location 3` (または追加したい数のプロキシ) を使用します。

ユースケースに応じて、 `--pgd-proxy-routing local` または `global` を指定して、PGD-Proxy が接続を書き込みリーダーに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 プロキシルートに独自の場所内の書き込みリーダーに作成します (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、すべてのプロキシルートに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た単一の書き込みリーダーにします。

オプションで `--bdr-database dbname` を指定して、BDR を有効にし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、各リージョンに 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(s) api` エンドポイントを有効にすることができます。

`tpaexec help configure-options` で説明されているオプションのいずれかを指定することもできます。

7.2 BDR-Always-ON

EDB Postgres Distributed 3.7 または 4 Always-ON 構成で、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では、従来の 2ndQuadrant リポジトリへのサブスクリプションが必要であり、一部のオプションでは EDB Repos 1.0 へのサブスクリプションが必要です。 [How TPA uses 2ndQuadrant and EDB repositories](#) を参照

このトピックの詳細については。

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

1. ブロンズ: $2 \times \text{bdr} + \text{プライマリ}$ 、`bdr + Witness`、バーマン、 $2 \times \text{harp-proxy}$
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ました
3. ゴールド: それぞれ $2 \times \text{bdr} + \text{プライマリ}$ 、 $2 \times \text{harp-proxy}$ 、および `barman` を持つ 2 つの対称的な場所。プラス 3 番目の場所に `bdr + 証人`

4. プラチナ : ゴールド、ただし、各メインの場所に 1 つの bdr + readonly (論理スタンバイ) が追加されています

EDB の Postgres Distributed Always On Architectures

[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 で使用するためのものです。

7.2.1 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/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
```

次の表は、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

Option	Description
<code>--architecture (-a)</code>	‘BDR-Always-ON’ 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 tpaexec-configure 。
<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.
<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-harp-probes</code>	<code>`health/is-ready`</code> などの harp の http(s) 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。	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 ユースケースの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、プラチナのレイアウトには、Raft コンセンサスマジョリティのノード数が奇数であることを保証する PGD Witness 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。詳細については、[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 を有効にし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 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、http(s) api エンドポイントを有効にして、harp のヘルスを簡単にプローブできるようにします。

BDR-Always-ON アーキテクチャでは、デフォルトで HARP2 を有効にすることに注意してください。

`tpaexec help configure-options` で説明されているオプションのいずれかを指定することもできます。

7.3 M1

プライマリとストリーミングレプリカ、1 つの Barman サーバー、および最初のレプリカからカスケードされた任意の数の追加レプリカを備えた Postgres クラスタ。このアーキテクチャは、生産に適しており、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、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、それは EDB Repos 2.0 から供給され、トークンを提供する必要があります。

How TPA uses 2ndQuadrant and EDB repositories を参照

このトピックの詳細については。

7.3.1 デフォルトのレイアウト

デフォルトでは、プライマリには 1 つの読み取り専用レプリカが同じ場所に接続されています。レプリカには、別の場所に接続された 1 つのカスケードレプリカがあり、Barman サーバーもプライマリからバックアップを取得するように構成されています。

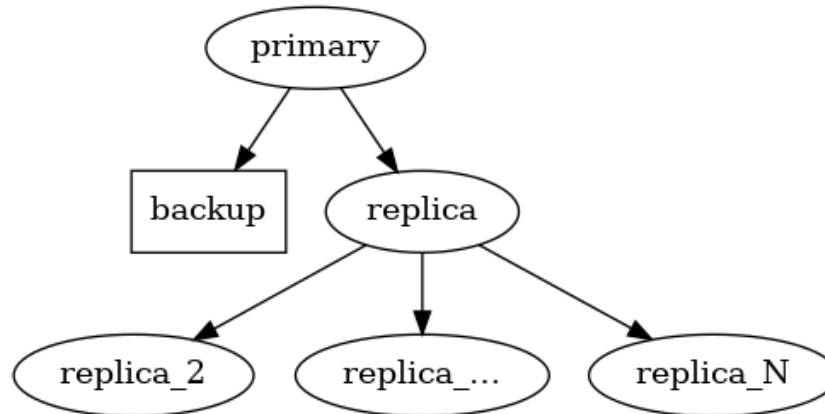


図 1: Cluster with cascading replication

Postgres ノードの数が偶数の場合、Barman ノードは witness として追加で構成されます。これにより、ノードの数が常に奇数になることが保証され、自動フェールオーバーを有効にする場合に便利です。

7.3.2 アプリケーションとバックアップのフェイルオーバー

M1 アーキテクチャは、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の代わりに昇格することを保証するという点で、フェールオーバー管理を実装します。ただし、**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は提供しません**。アプリケーショントラフィックの自動フェールオーバーが必要な場合は、アプリケーション自身自身で（マルチホスト接続を使用するなど）、または適切なプロキシまたはロードバランサーと、選択したフェールオーバーマネージャーが提供する機能を使用して、これを構成する必要があります。

上記は、バックアップノードと TPA によって作成されたプライマリ間の接続にも当てはまります。バックアップは、フェールオーバーが発生した場合に新しいプライマリをターゲットにするように自動的に調整されません。代わりに、元のプライマリに接続されたままになります。手動フェールオーバーを実行していて、バックアップを新しいプライマリに接続したい場合は、`tpaexec deploy` を再実行するだけです。バックアップソースを自動的に変更する場合は、上記のように選択したフェールオーバーマネージャーを使用してこれを実装する必要があります。

7.3.3 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
```

`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

Parameter	Description
<code>--architecture (-a)</code>	<code>M1</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>tpaexec-configure</i> 。
One of: <code>* --failover-manager {efm, repmgr, patroni}</code> <code>* --enable-efm</code> <code>* --enable-repmgr</code> <code>* --enable-patroni</code>	<i>tpaexec-configure</i> 、 <i>tpaexec-configure</i> 、および <i>tpaexec-configure</i> からフェー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>--num-cascaded-replicas</code>	最初のレプリカからのカスケードレプリカの数。	Defaults to 1.

次のページに続く

表 6 – 前のページからの続き

Parameter	Description	Behaviour if omitted
<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。 フェイルオーバーマネージャーとしての Patroni でのみサポートされます。	HAproxy nodes will not be added to the cluster.
<code>--patroni-dcs</code>	Patroni の Distributed Configuration Store バックエンドを選択します。 現時点では、オプションのみが `etcd` です。 フェイルオーバーマネージャーとしての Patroni でのみサポートされています。	Defaults to <i>etcd</i> .

M1 構成の詳細

オプションで `--num-cascaded-replicas N` を指定して、N 個のカスケードレプリカを要求できます（ない場合は 0 を含みます。デフォルト：1）。

`tpaexec help configure-options` で説明されているオプションのいずれかを指定することもできます。

第8章 Platforms

8.1 aws

TPA は、AWS EC2 での本番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。

8.1.1 API アクセス設定

AWS API を使用するには、以下を行う必要があります。

- Obtain an access keypair
- Add it to your configuration

たとえば、

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

AMI ユーザーには、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、少なくとも次の一連のアクセス許可が必要です。

```
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
```

(次のページに続く)

```
ec2:DescribeInstanceStatus
ec2:DescribeInstances
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:RunInstances
ec2:TerminateInstances
iam:AddRoleToInstanceProfile
iam:CreateInstanceProfile
iam:CreateRole
iam>DeleteInstanceProfile
iam>DeleteRole
iam>DeleteRolePolicy
iam:GetInstanceProfile
iam:GetRole
iam:GetRolePolicy:
iam:ListGroups
iam:ListInstanceProfiles
iam:ListInstanceProfilesForRole
iam:ListRolePolicies
iam:ListRoles
iam:ListUsers
iam:PassRole
iam:PutRolePolicy
iam:RemoveRoleFromInstanceProfile
kms:CreateGrant
kms:GenerateDataKeyWithoutPlaintext
s3:GetObject
s3:ListAllMyBuckets
s3:ListBucket
s3:PutObject
s3:PutObjectAcl
```

8.1.2 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、リージョンコードとそれに続く単一の文字で表されます（例：eu-west-1a）（ただし、その名前は異なる AWS アカウントの異なる場所を指す場合があり、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ており、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、論理的にはほとんど分離されていますが、低レイテンシーのリンクで接続されており、特定のネットワーキング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キング構成は、

Virtual Private Cloud のコンテキストで発生します

地域内。VPC 内で、**subnets** を作成できます

これは、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を構成します（プロトコル、宛先ポート範囲、およびソースまたは宛先 IP アドレス範囲の観点から）。

インスタンス

AWS EC2 はさまざまな **instance types** を提供しています

さまざまな価格/パフォーマンスポイントでさまざまなハードウェア構成を使用できます。特定のアベイラビリティゾーンのサブネット内で、**EC2 instances** を作成できます

AMI として知られる配布画像に基づいて、1 つ以上の **EBS volumes** を添付します

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 で接続できます。

インスタンスには、常にサブネット内のプライベート IP アドレスが割り当てられます。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もあります

（インスタンスがシャットダウンされると失われ、再度起動すると別のエフェメラル 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な 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。

インスタンスが外部から到達可能であるには、ルーティング可能な IP アドレスが必要であるだけでなく、VPC のネットワーキング構成（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。

8.1.3 構成

`tpaexec configure` を介して指定するか、`config.yml` で直接定義できる AWS 固有の設定の簡単な説明を次に示します。

地域

`--region` または `--regions` で使用するクラスターの 1 つ以上の領域を指定できます。TPA は、それぞれに関連付けられた必要な `vpc` エントリーを生成し、可能な限り異なるアベイラビリティゾーンを使用し、これらのリージョンに場所を均等に配布します。

`regions` は `locations` とは異なり、各場所はリージョン（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` エントリーには、AWS `vpc` ピアリングの使用を許可する非重複 `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、それらが属する `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` の下に基本ルールを持つセキュリティグループを作成します。それらは、各 `subnet cidr` の `ec2_vpc cidr` に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。
- VPC ピアリングは、`tpaexec deploy` の前に手動でセットアップ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
```

これはデフォルトの設定で、存在しない場合は指定された CIDR で `Test` という名前の VPC を作成し、存在しない場合は既存の VPC を使用します。

VPC を作成するには、`Name` 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 が存在しないと TPA は失敗します。

TPA が VPC を作成する場合、`tpaexec deprovision` はそれを削除しようとしませんが、既存の VPC はそのままにします。（AWS にはアカウントあたりの VPC の数に 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。）

よりきめ細かなマッチングが必要な場合、または異なるリージョンで異なる VPC を指定する場合は、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ー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。)

このプラットフォームは、Debian 9 (ストレッチ)、RedHat Enterprise Linux 7、Rocky 8、Ubuntu 16.04 (Xenial)、および SUSE Linux Enterprise Server 15 をサポートしています。

サブネット (オプション)

すべてのインスタンスは、そのサブネットを指定する必要があります (CIDR 形式、または subnet-xxx id として)。オプションで、作成するサブネットごとに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
      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、次を設定します。

```
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 範囲からの任意のポートでの TCP 接続を許可します。(注: from_port と 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は、クラスター内のインスタンスが必要に応じて相互に通信でき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、以下を設定しない限り、すべての VPC にインターネットゲートウェイを作成します。

```
ec2_instance_reachability: private
```

よりきめ細かな制御を行うには、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）にも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 で作成することを確認するように求められます。）

代わりに既存の S3 バケットを使用するには、設定します

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

（確認プロンプトなしで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しても、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権が必要です。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。

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
```

8.2 bare(-metal servers)

config.yml で 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、任意のクラウドプラットフォーム（AWS を含む）に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
```

あなたはそれを保証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、インスタンスに対する sudo アクセスがあります

8.2.1 SSH アクセス

上記の例では、TPA は `xyzyy@192.0.2.1` に `ssh` 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。秘密鍵の名前は `id_cluster_name`、公開鍵は `id_cluster_name.pub` に格納されます。

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` の内容を `~xyzyy/.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、クラスターディレクトリの `hostkeys/ssh_host_*_key*` から TPA で生成されたホストキーをインスタンスの `/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、既存のホストキーを保持するには、次のようになります。

```
$ cd ~/clusters/speedy
$ ssh-copy-id -i id_speedy xyzyy@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。そうでない場合は、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は、これを、代わりに `'id'`、いくつか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：

- *Use a different ssh key*
- *Manage ssh host keys for bare instances*

8.2.2 配布サポート

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ていますが、サポートされていない場合は失敗します。TPA は現在、Debian (8/9/10; または `jessie/stretch/buster`)、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。

8.2.3 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。最初に、デプロイ中にインスタンスに `ssh` してコマンドを実行します。第二に、クラスター内の通信をセットアップするため、たとえば、`/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` 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のセットアップに使用されます。両方を指定すると、展開中に `public_ip` が使用され、クラスター通信に `private_ip` が使用されます。

どちらか一方のみを指定した場合、アドレスは両方の目的で使用されます。たとえば、異なるネットワーク上のサーバーには `public_ip` のみを設定できます。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。)

8.3 Docker

TPA は、Docker コンテナを作成し、クラスターをそれらにデプロイできます。現時点では、コンテナをセットアップして `systemd` やその他のサービスを通常の VM のように実行します。

docker コンテナへのデプロイは、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。実稼働での使用を意図したものではありません。

8.3.1 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
```

8.3.2 オペレーティングシステムの選択

標準の `--os Debian/Ubuntu/RedHat/SLES` 構成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。これらの画像には、`tpa/` プレフィックスが付いた名前が付けられます (例、`tpa/redhat:8`)。

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

```
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 コンテナの 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古いバージョンの systemd にバグがあるためです。

8.3.3 Docker のインストール

最新の安定した Docker-CE パッケージで TPA をテストします。

このドキュメントでは、Docker が正常にインストールされており、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](#) に関するヘルプが必要な場合は、[Docker documentation](#) にご相談ください。

MacOS X では、[install "Docker Desktop for Mac"](#) できます

アプリケーションメニューから Docker を起動します。

CgroupVersion

CgroupVersion 2 のサポートは、ansible および関連するツールの docker sdk に対してまだ完全には焼き付けられていません。そのため、最新バージョンの docker を使用することをお勧めしますが。バージョン 2 が完全にサポートされるまで、CgroupVersion 1 に依存しています。以下の手順は、プラットフォームがデフォルトで CgroupVersion 2 を使用している場合、CgroupVersion 1 に切り替えるための変更を提案します。

Linux の場合：

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

権限

TPA は、実行しているユーザーが Docker デーモンにアクセスする権限を持っていることを想定しています (通常、`/var/run/docker.sock` を所有する `docker` グループのメンバーであることによる)。次のようなコマンドを実行して、アクセスでき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 デーモンと話す機能を与えると、ユーザーは簡単に Docker ホストで `root` を取得できます。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ようにする必要があります。

Docker コンテナの権限

特権コンテナ

デフォルトでは、TPA は非特権モードで Docker コンテナをプロビジョニングし、Linux 機能フラグは追加されません。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、またはホストでの真の `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に `privileged` ブール変数を設定するか、`instance_defaults` でグローバルに設定します。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の変更、コンテナ名前空間のエスケープ、またはその他の方法で実際のホスト「`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、コンテナを特権モードで実行しないでください。

Linux ケーパビリティフラグの詳細については、`man capabilities` を参照してください。

security_opts および no-new-privileges フラグ

`tpaexec` は、プロセスが特権を増やすことが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。`tpaexec` で `instance_defaults` またはコンテナごとの変数 `docker_security_opts` を使用して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 ユーザーとロール。

Linux 機能フラグ

tpaexec は、コンテナごとまたは `instance_defaults` で設定できる `docker_cap_add` リスト変数を使用して、Linux 機能フラグに対する Docker の制御を公開します。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、システムクロックを変更できるようにするには、次のように記述します。

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

Docker ストレージ構成

注意：多くのホストのデフォルトの Docker 構成は `lvm-loop` ブロックストレージを使用し、実稼働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、次のように表示されます。

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

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

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

8.3.4 Docker コンテナ管理

`start-containers` および `stop-containers` コマンドを使用して、クラスター内のすべての docker コンテナを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たり、接続したりしません。これらは、将来の使用に備えて利用可能な状態にしておく必要がある docker クラスターを一時的に使用していないときに、リソースを節約することを目的としています。

クラスター内のプロビジョニングされた docker コンテナの概要については、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9章 Customizations

9.1 Cluster configuration

TPA を使用して、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/デプロイ/テストサイクルを実行することです。このプロセスは、べき等であり、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加えるように慎重に設計されています。

`tpaexec configure` コマンドは、適切な `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。デフォルトを超えるものが必要な場合、またはクラスターのプロビジョニング後に変更を加える必要がある場合は、とにかく `config.yml` を編集する必要があります。

このページは、利用可能な構成メカニズムの概要です。特定の *variables you can set to customise the deployment process* に関する詳細を記載した別のページがあります。

9.1.1 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
```

(次のページに続く)

```

role: replica
platform: docker
upstream: one
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です。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）が含まれる場合がありますが、1つのインスタンスまたはクラスター全体に対して `vars` が設定された `instances` のリストは、すべての TPA 構成の基本的なビルディングブロックです。

すべて *tpaexec configure*

`options` は、何らかの方法で `config.yml` 変数に変換されます。単一のオプションがいくつか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`、`tpa_2q_repositories`、`edb_repositories`、`extra_postgres_extensions` などを設定できます）が、コマンドを実行することにより、エディターでいつでもできることを達成できます。

YAML 構文に関しては、`config.yml` 全体は `cluster_vars` や `instances` などのキーを持つハッシュを表します。各キーが1回だけ定義されることを保証する必要があります。誤って `cluster_vars` を繰り返した場合、たとえば、2番目の定義が前者を完全にオーバーライドし、次の展開で不足している（シャドウされた）変数が原因で意図しない変更が行われる可能性があります。

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「`replica`」でインスタンスを宣言する場合、その上流のインスタンスの名前も宣言する必要があり、そのインスタンスが存在する必要があります）。インスタンスで好きな変数を設定します。十分な注意を払い、変更をテスト環境で試してから、本番環境にロールアウトする必要があります。

9.1.2 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 `cluster_vars` を `group_vars`（クラスター全体に適用）に変換し、プロビジョニング中にインベントリー内の各インスタンスの `vars` を `host_vars` に変換し、デプロイメントではインベントリー値を使用します。`config.yml` を変更した後、`tpaexecdeploy` 前に `** tpaexec provision` を実行することを忘れないでください。

変数は、クラスター全体、または個々のホスト、またはその両方に設定できます。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` に `x: 42` を設定することは、すべてのホストの `vars` に設定することと同じです。展開中に `x` を必要とするホストには、いずれかの方法で値 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でオーバーライドすると便利です。

可能な場合はいつでも、`cluster_vars` で変数を定義し、特定のインスタンスに対してそれらをオーバーライドすると、確認と変更が容易な簡潔な構成になります（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理にかなっているかどうかを判断するのはあなた次第です。

9.1.3 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 インベントリーのグループ変数に直接変換されます。

```
cluster_vars:
  postgres_version: 14
  tpa_2q_repositories:
    - products/bdr3/release
    - products/pglogical3/release
  postgres_conf_settings:
    bdr.trace_replay: true
```

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。

9.1.4 インスタンス変数

このドキュメントでは、「インスタンス変数」という用語を使用して、`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します。例、一般的なインスタンス定義は次のとおりです。

```
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:
    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内のホスト変数になりますが、インベントリ内のすべてのホスト変数は `vars` 単独からのものではありません。 `platform`、`location`、`volumes`、および `role` を含む他のいくつかのインスタンス設定もホスト変数としてインベントリにコピーされます（ただし、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（`Name` および `node` など）のプロパティ、またはクラスターのトポロジ内の場所（`role`、`backup` など）、またはプラットフォーム固有の属性（インスタンス `type` および `volumes` など）を説明する場合があります。 `vars` で定義できないことを知っている以外に、これらのインスタンス「設定」とインスタンス「変数」を区別する必要はほとんどありません。

この場合、`tpaexec provision` は `host_vars/unwind/01-instance_vars.yml` のインベントリに多数のホスト変数を書き込みます。

9.1.5 instance_defaults

これは、`config.yml` での繰り返しをさらに減らすためのメカニズムです。 `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。たとえば、次のように記述できます。

```
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` で指定したものは何でも、`instances` のすべてのエントリ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` のスペルを節約し、すべてのインスタンスを別のタイプに変更しやすくします。いずれかのインスタンスが別の値を指定する場合、もちろんデフォルトよりも優先されます。

`instance_defaults` を、`instances` の定義に使用するマクロ機能と考えると役立つ場合があります。最終的にインベントリに書き込まれるのは、`instances` のみの（拡張された）定義から来ます。 `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、多くのプラットフォーム固有のプロパティ（AWS リソースタグなど）プロビジョニングでのみ使用され、デプロイ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` に入力するために使用することを停止することはありません（デフォルトのハッシュ値は、`instances` エントリ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で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ありません。疑わしい場合は、`cluster_vars` を使用してください。

9.1.6 場所

`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`) を指定している場合、その場所の下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。繰り返しますが、`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がいくつかあり、残りの半分には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ように）。

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。

9.2 Instance configuration

このページでは、TPA がクラスターインスタンスでのデプロイメントプロセスをカスタマイズするために提供するさまざまなコントロールの概要と、より詳細なドキュメントへのリンクを示します。

デプロイメントの詳細に入る前に、

an overview of configuring a cluster を読んで、クラスター変数とインスタンス変数、および config.yml 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記述できるかを理解することが役立つ場合があります。

9.2.1 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、さまざまなシステムレベルの構成タスクを完了してから、以下の *Postgres configuration* に進みます。

- **配布サポート**
- *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

pre-deploy hook 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。

- *Configure YUM repositories*

(RHEL、Rocky、AlmaLinux の場合)

- *Configure APT repositories*

(Debian および Ubuntu の場合)

- *Configure 2ndQuadrant and EDB repositories*

(任意のシステム)

- *Configure a local package repository*

(パッケージをターゲットインスタンスに出荷するため)

post-repo hook 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します (例、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、システムパッケージのバッチから始めて、展開全体のさまざまな段階でインストールされます。

- *Install non-Postgres packages*

(例、acl、openssl、sysstat)

Postgres およびその他のコンポーネント（例、Barman、repmgr、pgbouncer）は、クラスター構成に従って個別にインストールされます。これらは、以下のそれぞれのセクションで文書化されています。

その他のシステムレベルのタスク

- *Create and mount filesystems* (RAID、LUKS セットアップを含む)
- *Upload artifacts* (ファイル、ディレクトリ、tar アーカイブ)
- *Set sysctl values*
- *Configure /etc/hosts*
- *Manage ssh_known_hosts entries*
- *Add system locale*

9.2.2 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。Postgres 自分自身をインストールすることから始まります。

バージョン選択

configure options 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。

作業クラスターをセットアップするために本当に必要なことはこれだけです。このページの他のすべてはオプションです。必要に応じて、展開のあらゆる側面を制御できますが、デフォルトは慎重に調整され、出発点として実用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` 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とさまざまな拡張機能をインストールすることです。

- *Install Postgres and Postgres-related packages*

(例、pglogical、BDR など)

- *Build and install Postgres and extensions from source*

(開発およびテスト用)

どちらのインストール方法を選択しても、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

- *Configure the postgres Unix user*
- *Run initdb to create the PGDATA directory*
- *Configure pg_hba.conf*
- *Configure pg_ident.conf*
- *Configure postgresql.conf*

postgres-config hook 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ため (例、追加の構成ファイルをインストールするため)。

Postgres 構成が整ったら、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続行します。

9.2.3 その他のコンポーネント

- *Configure Barman*
- *Configure pgbouncer*
- *Configure haproxy*
- *Configure HARP*
- *Configure 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。

最初のデプロイでは、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、行った構成の変更に応じて、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、サービスがすでに実行されていて変更がない場合、そのままです。

いずれの場合も、このステップの最後に Postgres が実行されます。

9.2.4 Postgres 起動後

- *Create Postgres users*
- *Create Postgres tablespaces*
- *Create Postgres databases*
- *Configure pglogical replication*
- *Configure .pgpass*

postgres-config-final hook 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（たとえば、SQL クエリを実行してオブジェクトを作成したり、データをロードしたりするため）。

- *Configure BDR*

post-deploy hook 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。

9.3 Building from source

TPA は、Postgres およびその他の必要なコンポーネントをソースからビルドし、デフォルトのパッケージインストールとまったく同じ構成でクラスターをデプロイできます。これにより、ソースから繰り返し展開して、アーキテクチャやプラットフォームに関係なく、特定のセットアップのあらゆる側面を再現する現実的で完全に構成されたクラスターで変更をすばやくテストできます。

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、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、およびサポートオペレーションでの使用を目的としています。

9.3.1 クイックスタート

すべて安定したブランチからビルドされた 2ndQPostgres、pglogical3、および bdr 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
```

上記と同じですが、公式の 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 \
  --local-source-directories \
  pglogical3:~/src/pglogical bdr3:~/src/bdr
```

クラスターをデプロイした後、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、ビルドを開始する前にビルドディレクトリが空になります。）

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お読みください。

9.3.2 構成

ソースビルドの構成には 2 つの側面があります。

ソースの特定の組み合わせを実行するクラスターが必要な場合は、`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適切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、ビルド元の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が許可するよりもはるかに多くのことができます。`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、さまざまな構成またはビルドパラメーターを指定したり、ビルドコマンド全体を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。

利用可能なオプションはここに文書化されています：

- *Building Postgres from source*
- Building extensions with ``install_from_source <Installing from source>``

9.3.3 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ことを指定できます。

```
$ 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、リポジトリから Postgres Extended をインストールしますが、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 リポジトリが複製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（ツリー外）のビルドが通常どおり続行されます。

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）。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 もです。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すべて取得します。

`--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

(次のページに続く)

```
- /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対してデフォルトで `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アタッチされ、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、クラスターのプロビジョニングが解除されると削除されます。

`--shared-ccache /path/to/host/ccache` 構成オプションを使用して、より寿命の長い共有 `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、その内容はホストとコンテナの間で共有されます。

(設計上、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。)

9.3.4 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デプロイした後、`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。

9.4 TPA hooks

TPA は、ユーザーの介入なしで完全に機能する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) など、さまざまな *settings to control your cluster configuration* を既に提供しています。

利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するフックスクリプトを作成できます。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、フックを使用してできることに制限はありません。それらを注意して使用し、負担している追加のメンテナンスの負担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以外にリリース間の互換性を保証できません。

9.4.1 まとめ

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、以下で説明するように、TPA はデプロイメント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 ファイルであると想定されます (`play` のリストを含む Playbook ではありません)。以下で特に記載がない限り、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。

9.4.2 汎用フック

事前展開

TPA は、Python のブートストラップ直後に `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も早い段階です。

このフックを使用して、`apt_repositories` または `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。

ポストレポ

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。パッケージのインストールを開始する前に、これを使用してリポジトリ構成を修正できます。

pre-initdb

TPA は、`hooks/pre-initdb.yml` を呼び出してから、

`run initdb to create PGDATA` が存在しない場合にそれを行うかどうかを決定します。通常、このフックを使用する必要はありません (ただし、これを使用して自分で `PGDATA`

を作成すると、TPA は `initdb` をスキップします)。

postgres-config

TPA は、`pg_hba.conf` および `conf.d` 内のファイルを含む 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。

たとえば、このフックを使用して、`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。

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。

たとえば、このフックを使用して、TPA の組み込み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。

デプロイ後

TPA は、デプロイメントの最後に `hooks/post-deploy.yml` を呼び出します。

この段階の後、あなたはやりたいことをやり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` によって変更が上書きされるリスクがあります (TPA はフックが何をしたかわからないため)。

9.4.3 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、インスタンスで PGD ノードを初めて作成する前に、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 PGD ノードがすべて既に存在する場合、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、ノードが参加する前に、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、ノード結合を開始する前に必要になる可能性のあ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するその他の調整を実行できます。

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

9.4.4 その他のフック

postgres-pre-update、postgres-post-update

`upgrade` 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。両方のフックは、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、環境の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）。

9.4.5 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。あなたの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、お問い合わせください。

第10章 Cluster management

10.1 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` コマンドに代わるものです。）

10.1.1 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` を実行してから `tpaexec deploy` を実行することです。

この規則の例外は、`tpaexec deploy` は、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のアップグレードを実行する必要があります。

このコマンドは、クラスター操作の中断を最小限に抑えてアップグレードの実行を試みます。以下に文書化されているように、特殊なアップグレードプロセスの正確な詳細は、クラスターのアーキテクチャによって異なります。

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取得し、アップグレードのために確保された時間中に行わ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を停止し、新しいパッケージをインストールし、必要に応じて構成を更新し、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います。プロセス中はいつでも、クラスターのノードの1つだけが使用できなくなります。

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、`-e enable_proxy_monitoring=true` オプションを `tpaexec upgrade` コマンドラインに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監視を有効にできます。有効にすると、これにより `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。

10.1.2 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 `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適切にアップグレードされます（正確には、クラスターの詳細によって異なります）。

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以外の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設定し、プロセス中に BDR ノードとグループの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行う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` はそ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（この `deploy` は通常も行います）。

ただし、アップグレードプロセス自身自身で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、アップグレード中にそれらの変更は行われません。混乱を避けるため、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、関係のない保留中の変更を `tpaexec deploy` することをお勧めします。

10.1.3 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、BDR4 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、クラスターのアップグレードに必要な変更を実行して、新しい `config.yml` を書き込みます。その呼び出しの詳細については、

[the command's own documentation](#) 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、アンシブルインベントリを更新します。アップグレードプロセス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. クラスター内の各インスタンスについて、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で利用可能であることを確認します。

3. クラスター内の各 BDR ノードについて、一度に 1 つずつ： - ノードをフェンスして、harp-proxy が接続を送信しないようにします。 - BDR4 を PGD5 に置き換えることを含め、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、再度接続を受信できるようにします。 - このノードに必要な応じて、pgbouncer と pgd-cli を更新します。
4. クラスター内の各インスタンスについて、特に BDR v5 専用の BDR 構成を更新します
5. クラスター内の各プロキシノードについて、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。

10.1.4 PGD-Always-ON

既存の PGD-Always-ON (PGD5) クラスターを利用可能な最新の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では次の処理が実行されます。

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. クラスター内の各インスタンスについて、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で利用可能であることを確認します。
3. クラスター内の各 BDR ノードについて、一度に 1 つずつ： - ノードをフェンスして、pgd-proxy が接続を送信しないようにします。 - 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て、再度接続を受信できるようにします。 - このノードに必要な応じて、pgbouncer、pgd-proxy、および pgd-cli を更新します。

10.1.5 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 つずつ調べ、次のことを行います。

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だった場合、pgbouncer に再接続を要求し、アクティブなセッションが閉じられるのを待ちます。
3. Postgres を停止し、パッケージを更新して、Postgres を再起動します。
4. 最後に、サーバーを再度「準備完了」としてマークし、haproxy を介してリクエストを受信します。

PGD ロジカルスタンバイまたはフィジカル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 インスタンスはそのまま残されます。

update_hosts 変数を定義することで、クラスターのインスタンスが更新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、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して、シャドウサーバーが最初に更新されるようにすることにより、更新中のリード/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。

ご使用の環境で追加のアクションが必要な場合、*postgres-pre-update and postgres-post-update 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。

10.1.6 M1

M1 クラスターの場合、`upgrade` は最初にストリーミングレプリカを 1 つずつ更新し、次に **スイッチオーバー**

プライマリからレプリカの 1 つに、プライマリを更新し、再度スイッチオーバーします。

10.1.7 パッケージバージョンの選択

デフォルトでは、クラスターの作成時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利用可能な最新バージョンに更新します。

たとえば、`--xxx-package-version` オプション (postgres、bdr、pglogical) のいずれかを *tpaexec configure* 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はすでに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て、`tpaexec provision` を再実行する必要があります。更新は、利用可能な最新のパッケージをインストールします。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*` のみを認識します (上記の例のように)。

注: *tpaexec-configure* の `package_version` でワイルドカードを使用する場合の制限を参照してください。

要求する 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保証するのはあなたの責任です。つまり、それらは連携して動作するはずであり、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必要です。

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 つすべてのコンポーネント (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。

本番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。

10.2 tpaexec switchover

`tpaexec switchover` コマンドは、*cluster that uses streaming replication* 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、元のプライマリが新しいレプリカになり、クラスター内の他のレプリカが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。)

10.2.1 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
```

10.2.2 アーキテクチャオプション

このコマンドは *M1 clusters* にのみ適用されます

単一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を持つ。

BDR-Always-ON クラスターの場合、*HAProxy server pool management commands* 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。

10.3 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 cluster* 内の 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、後で追加し直すことができます。

これらのコマンドは、デフォルトで *rolling updates* と同じプロセスに従います。そのため、`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するまで待機し、`pgbouncer` は完了する前に新しい接続を別のインスタンスに転送します。アクティブなセッションが終了するのを待ちたくない場合は、`--nowait` オプションを使用します。

インスタンスで `pool-disable-server` を実行した直後に `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を別のインスタンスに移動する効果があります (本質的に、スイッチオーバー)。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。

同じ `haproxy_backend_servers` セットで構成された HAProxy サーバーが複数ある場合、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から並行して削除または追加します。

10.3.1 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た場合でも、HAProxy はトラフィックをそのインスタンスに転送しません。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プールに戻すことを忘れないでください。

10.4 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 インスタンスをリビルドし、TPA によって管理されるクラスターへのセキュリティパッチと OS アップグレードの迅速なデプロイを可能にします。

必要な変更がすべて含まれた新しい AMI を指定すると、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アタッチします (に基づいて `config.yml`)。

最新のイメージを公開し、定期的にサーバーをゼロから再構築することを要求することは、一連のサーバーが個々のセキュリティ更新を自分でダウンロードしてインストールできるようにする代替の方法です。これにより、各サーバーの状態を一目で追跡しやすくなり、個々のサーバーへの手動変更をお勧めしません (インスタンスの置換中に消去されます)。

TPA を使用すると、リハイドレーション中のクラスター全体の中断を簡単に最小限に抑えることができますが、プロセスには個々のサーバーが終了して置き換えられるためのダウンタイムが必然的に伴う必要があります。

streaming replication cluster では、最初にレプリカをリハイドレートしてから *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 clusters* では、リハイドレートする前に `remove each server from the haproxy server pool` し、その後に追加し直すことができます。

Postgres および関連するコンポーネントにマイナーバージョンの更新をインストールしたいだけの場合は、代わりに

tpaexec upgrade コマンドを使用できます。

10.4.1 前提条件

インスタンスをリハイドレートできるようにするには、`config.yml` 内のそ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た 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接続したいため、`delete_on_termination` を無効にする必要があります。`attach_existing` を設定すると、新しいインスタンスをプロビジョニングするときに TPA が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、これを行います。

10.4.2 例

`~/clusters/night` に AWS クラスター構成があるとします。

構成を変更する

まず、`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
```

データボリュームごとに `delete_on_termination` が無効になっていることを確認します。パラメータが存在しない場合、AWS EC2 マネジメント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]` までスクロールダウンして、EBS ボリュームをクリックします。「終了時に削除」フラグが `true` に設定されている場合、`change it using `awscli <#appendix>`` 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。

両方の属性が正しく設定された例を次に示します。

```
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
```

(次のページに続く)

(前のページからの続き)

```
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、特定のインスタンスだけでなく `instance_defaults` でも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 2 つの属性が設定されていることを確認してください。)

水分補給を開始する

これは、`rehydrate` コマンドの構文です。

```
$ 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 (ただし、クラスター内のすべてのインスタンスを一度にリハイドレートすることはできません)。

このコマンドは、最初に、リハイドレートされるインスタンスにアタッチ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。そうでない場合、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、プロビジョニングとデプロイを実行して、新しい AMI を使用して新しいインスタンスに置き換えます。

10.4.3 段階的に水分を補給する

クラスターの継続性を維持するには、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 *cluster that uses streaming replication* では、Barman インスタンスと 1 つのレプリカを最初にリハイドレートし、次に別のレプリカをリハイドレートし、次にプライマリからリハイドレートされたレプリカの 1 つに **スイッチオーバー** し、前者をリハイドレートできます。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可能になります。

10.4.4 付録

awscli を使用してボリューム属性を変更する

まず、AWS マネジメント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。[インスタンス] をクリックしてインスタンスを選択し、[説明] タブを開き、[デバイスのブロック] までスクロールダウンして、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。少し遅れて、管理コンソールで変更を確認できるはずです。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11章 Miscellaneous

11.1 TPA and Ansible Tower/Ansible Automation Controller

TPA は RedHat Ansible Automation Controller (旧称 Ansible Tower) で使用できます。スタンドアロンマシンで構成およびプロビジョニングの手順を実行し、クラスターをベアメタルとして扱い、結果のインベントリおよびその他の生成されたファイルを Tower にインポートします。

独自の仮想環境 (tpa-venv) を含む Tower インスタンスへの TPA インストールは、Tower でのデプロイ手順に使用されます。

このドキュメントでは、Ansible Tower バージョン 3.8.0 に該当する手順について説明します。

11.1.1 TPA を使用するための Tower の準備

TPA 仮想環境のセットアップ

- Tower サーバーの /opt/EDB/TPA の下に tpaexec をインストールします。これは、TPA がパッケージからインストールされる場合のデフォルトの場所です。
- /opt/EDB/TPA/tpa-venv 配下に仮想環境を作成する `tpaexec setup` を実行し、必要なパッケージをインストールします。
- Tower UI の [システム設定] ページの [カスタム仮想環境パス] フィールドに TPA ディレクトリ (/opt/EDB/TPA) を追加します。

TPA_2Q_SUBSCRIPTION_TOKEN 資格情報タイプの作成

以下の説明に従って、「TPA_2Q_SUBSCRIPTION_TOKEN」というカスタム資格情報タイプを作成します。

- Tower UI の Credentials Type ページに移動します。
- 「NAME」フィールドを「TPA_2Q_SUBSCRIPTION_TOKEN」に設定します。
- これを「INPUT CONFIGURATION」フィールドに貼り付けます。

```
fields:
- id: tpa_2q_sub_token
  type: string
  label: TPA_2Q_SUBSCRIPTION_TOKEN
```

(次のページに続く)

(前のページからの続き)

required:

- tpa_2q_subscription_token

- これを「INJECTOR CONFIGURATION」フィールドに貼り付けます。

env:

TPA_2Q_SUBSCRIPTION_TOKEN: {{ tpa_2q_sub_token }}

- 「SAVE」ボタンをクリックします。

11.1.2 クラスターのセットアップ

- クラスターに使用する予定のホストが Tower インストールで認識されていること、SSH 経由でそれらにアクセスできる資格情報があること、および `ansible_host`、`public_ip`、および `private_ip` が設定されていることを確認します。
- `tpaexec` がインストールされたマシンで、ホスト名のリストを 1 行に 1 つずつ含むファイルを準備します。

```
mercury
venus
mars
jupiter
saturn
neptune
```

- `tpaexec configure` を実行します。

```
[tpa]$ tpaexec configure <clustername> \
    --platform bare \
    --use-ansible-tower https://aac.example.com/api \
    --tower-git-repository ssh://git@git.example.com/example \
    --hostnames-from <hostnamefile> \
    --architecture BDR-Always-ON \
    --layout bronze \
    --harp-consensus-protocol bdr \
    --postgresql 14
```

API エンドポイントは現在受け入れられ、`config.yml` に追加されますが、使用されません。git リポジトリは、クラスターデータを Tower にインポートするために使用されます。`tpaexec` はこのクラスターのリポジトリに独自のブランチを作成するため、既にどのブランチが存在してもかまいません。(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 で説明されている `tpaexec configure`

の他のすべてのオプションは引き続き有効です。

これにより、クラスターの名前を付けたクラスターディレクトリが作成されます。

- `config.yml` にはトップレベルの辞書 `ansible_tower` が含まれるようになり、`tpaexec provision` はクラスターを Tower クラスターとして扱います。
- 他の変更を行う必要がある場合は `config.yml` を編集します。
- `tpaexec provision` を実行して、インベントリおよびその他の関連ファイルを生成します。
- `git` リポジトリを使用して Tower にプロジェクトを追加します。
- プロジェクトからインベントリを外部ソースとしてインベントリに追加します。
- Tower でテンプレートを作成し、TPA 仮想環境、インベントリ、新しく作成したプロジェクトを指定します。また、ホストに到達するために必要な資格情報と、TPA サブスクリプショントークン (`TPA_2Q_SUBSCRIPTION_TOKEN`) を含む資格情報を含めます。
- 1 つの追加変数を設定します：`tpa_dir: /opt/EDB/TPA`
- 新しいテンプレートに基づいてジョブを実行して、クラスターをデプロイします。

11.2 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権限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（TPA 固有ではありません）と、TPA で管理されるシステムへの結果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（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て必須パッケージ、および `pip` を使用してオプションパッケージ）
- サービスを停止、リロード、および再起動します（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（例、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 も `sudo` を使用できる必要があります。 `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、`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。

11.2.1 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で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-kfoodiiprztstyerriqbjqughhbemejgpc ; \  
/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。概して、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。（プレイブックは `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使用して、Python インタープリターをブートストラップします。）

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 Playbook は、YAML 形式で記述された単なるシェルスクリプトではありません。任意の Ansible Playbook 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` を使用方法には、1 つの重要な結果があります。

`privilege escalation must be general`

。つまり、一部の管理者が慣れているように、`sudo` 呼び出しを `sudoers.conf` の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。すべてのコマンドの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た可能性があります。Python が `root` として実行されると、とにかくできることに効果的な制限はありません。

ターゲットホストで Python モジュールを実行することは、Ansible の仕組みです。これは、決して TPA に固有のものではなく、これらの考慮事項は、他の Ansible プレイブックにも等しく適用されます。

11.2.2 推奨事項

- 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、メンテナンスウィンドウなど、クラスターを最初にセットアップする場合、または `tpaexec deploy` を再度実行して構成を変更する場合にのみアクセスを必要とします。それまでは、アクセスを完全に無効にできます（`ssh` と `sudo` の両方に対する 1 行の変更）。

デプロイ中に、Ansible が行うことはすべて、プレイブックが何をしているのか、そして提供するパラメーターに基づいて一般的に予測可能であり、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と、`tpaexec` 自分自身が実行されているマシン上の 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ことではなく、実行時に何をするかを可視化することです。

11.2.3 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できます。

`tpaexec` を実行する前に環境で `ANSIBLE_ASK_PASS=yes` と `ANSIBLE_BECOME_ASK_PASS=yes` を設定した場合、Ansible はリモートサーバーのログインパスワードと `sudo` パスワードの入力を求めます。次に、リモートサーバーでログイン/`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、プレイブックの実行に著しく時間がかかります）。

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より効果的なセキュリティ制御であると考えているため、この操作モードはお勧めしません。ssh に公開鍵認証を使用すると、サーバーにアクセスできる人を効果的に制御できます。共有パスワードまたは複数の共有パスワードを保護するよりも、承認されたユーザーごとに単一の秘密鍵を保護する方が簡単です。また、ssh / sudo レベルでアクセスを必要なときに制限すると、メンテナンスウィンドウ中にパスワードによってセキュリティが追加されることはありません。

11.2.4 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`、環境の `ANSIBLE_BECOME_FLAGS`、またはインベントリーの `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正当な理由があります。たとえば、パスワードのない sudo が正しく構成されていない場合、`-S -n` を削除するとタスクがタイムアウトします。

11.2.5 ロギング

プレイブックの実行の場合、sudo ログには、ほとんどが Python の呼び出しが表示されます（誰かが `sudo -i` を使用したときに `bash` の呼び出しのみが表示されるのと同じように）。

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に対する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より高いレベルのビューについては、コントローラーの `ansible.log` に、そのタスクが何をしようとしていたかと結果が表示されます。

さらに詳細な情報、または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。bpfttrace/bcc から、より詳細な syscall 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るすべてのプロセスを表示します）。必要に応じて、これらのことのいずれかまたはすべてを行うことができますが、ロギングが増加するとオーバーヘッドが増加するという明らかな警告があります。

11.2.6 ローカル権限

installation instructions for TPA

sudo については、「これらのコマンドをルートとして実行する」の省略形としてのみ言及してください。TPA をインストールし、`tpaexec setup` を実行すると、TPA 自分自身にローカルマシンでの昇格した特権は必要ありません。（ただし、Docker を使用する場合は、Docker デー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。）

11.3 TPA - PuTTY Configuration guide

Windows で PuTTY を使用して、TPA ユーティリティ **tpaexecprovision** で作成された AWS インスタンスに ssh 経由で接続するには、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。

```
# 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。

11.3.1 キー変換

秘密鍵を見つけます

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-それは、**config.yml** で設定され、**id_** プレフィックスが付いた **cluster_name** 変数に従って名前付けられます。cluster_name が **testenv1** に設定されている場合、秘密鍵は **id_testenv1** と呼ばれます。

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** -テキストファイルへのカットアンドペーストは、これで正常に機能します。

キー変換

PuTTYgen を起動し、[パラメーター] で、適切な [Type of key to generate] を選択します。

PuTTYgen の古いバージョンの場合、**SSH-2RSA** を選択します。最近のバージョンの場合は **RSA** を選択します

SSH-1 (RSA) を選択しないでください

今すぐ **Load** を選択します- **** PuTTY Private Key Files (*.ppk) というボックスで AllFiles(*.*)** を選択する必要があります。

.pem ファイルを選択し、**開く** を選択して、**OK** をクリックします。

Saveprivatekey 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**

11.3.2 PuTTY を構成する

PuTTY を起動し、**Category** ウィンドウから **Session** を選択します。 **HostName** パネルで <user>@<IP address> と入力し、Port Panel で **22** と入力します

<user> および <IP address> は、 ``tpaexecprovision`` 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

認証用のプライベートキーファイル 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 ファイルを選択して、**** Open **** を選択します

Putty **Category** ウィンドウで、再度 **Session** を選択し、**SavedSessions** にセッション名を入力し、**Save** この保存されたセッションを選択すると、PuTTY を介して AWS ホストに接続できるはずです。

11.4 Troubleshooting

11.4.1 Python 仮想環境を再作成する

場合によっては、python venv が矛盾した状態になることがあります。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。venv が壊れた場合の症状には、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たと仮定）：

```
[tpa]$ sudo rm -rf /opt/EDB/TPA/tpa-venv
[tpa]$ sudo /opt/EDB/TPA/bin/tpaexec setup
```

11.4.2 認証情報に関する奇妙な AWS エラー

TPA サーバーの日時が正しくない場合、プロビジョニング中に次のような AWS エラーが発生します。

```
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
```

(次のページに続く)

(前のページからの続き)

```

↳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
```

11.4.3 ロギング

デフォルトでは、すべての tpaexec ロギングはログファイル<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。

```
export ANSIBLE_LOG_PATH=~/.ansible.log
```

ロギングの冗長性を高めるには、`-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

```

11.4.4 クラスタテスト

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行われ、pgbench を使用してクラスターのパフォーマンステストが行われます。pgbench は完了するまでに時間がかかる場合があるため、次を実行してベンチマークを省略できます。

```
[tpa]$ tpaexec test <clustername> --skip-tags pgbench
```

test ロールのタグは `repmgr`, `postgres`, `barman`, `pgbench` です

複数のタグを指定するときは、スペースを入れずにコンマで区切る必要があることに注意してください。例：

```
[tpa]$ tpaexec test <clustername> --skip-tags barman,pgbench
```

11.4.5 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、次を実行します。

```
[tpa]$ tpaexec selftest
```

11.4.6 特定のタグをスキップまたは含める

障害後に `tpaexec provision` または `deploy` を再実行する場合、時間の都合上、特定のタグをスキップしてタスクを逃すと役立つ場合があります。たとえば、`repmgr` タスクを逃すには：

```
[tpa]$ tpaexec deploy <clustername> --skip-tags repmgr
```

タグを指定して特定のタスクの再実行に直接ジャンプするには、たとえば、PGD タスクをすぐに実行します（従来の理由でタグは `bdr` です）：

```
[tpa]$ tpaexec deploy <bdr-clustername> --tags bdr
```

- これは、前のタスクがすべて正常に完了したことを前提としています。

関連するアーキテクチャの役立つタグをすべてを見つけるには、次を実行します。

```
[tpa]$ grep -rs "tags:" /opt/EDB/TPA/architectures
```


第12章 Reference

12.1 Installation

12.1.1 Running TPA in a Docker container

使用可能な *TPA* パッケージのインストールがないシステムを使用しており、

Installing from source の後に TPA を実行するのが難しい場合（たとえば、動作する Python

3.6+インタープリターを取得するのが簡単でないため）、最後の手段は Docker イメージをビルドすることです Docker コンテナ内で TPA を実行します。

deploy to Docker containers を実行するために 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

Docker がシステムにインストールされ、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 -t tpa/tpaexec .
```

作成されたイメージを再確認します。

```
$ docker image ls tpa/tpaexec
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
tpa/tpaexec   latest    e145cf8276fb   8 seconds ago  1.73GB
$ docker run --platform=linux/amd64 --rm tpa/tpaexec tpaexec info
# TPAexec v20.11-59-g85a62fe3 (branch: master)

tpaexec=/usr/local/bin/tpaexec
TPA_DIR=/opt/EDB/TPA
PYTHON=/opt/EDB/TPA/tpa-venv/bin/python3 (v3.7.3, venv)
TPA_VENV=/opt/EDB/TPA/tpa-venv
ANSIBLE=/opt/EDB/TPA/tpa-venv/bin/ansible (v2.8.15)
```

TPA コンテナを作成し、コンテナ内でクラスター構成ディレクトリを利用できるようにします。

```
$ docker run --platform=linux/amd64 --rm -v ~/clusters:/clusters \
  -it tpa/tpaexec:latest
```

コンテナプロンプトで `tpaexec provision /clusters/speedy` などのコマンドを実行できるようになりました。(シェルを終了すると、コンテナは削除されます。)

TPA を使用して Docker コンテナをプロビジョニングする場合は、コンテナがホスト上の Docker 制御ソケットにアクセスできるようにする必要があります。

```
$ docker run --platform=linux/amd64 --rm -v ~/clusters:/clusters \
  -v /var/run/docker.sock:/var/run/docker.sock \
  -it tpa/tpaexec:latest
```

コンテナ内で `docker ps` を実行して、ホスト Docker デーモンへの接続が機能していることを確認します。

Docker のインストール

[install Docker](#) および [get started](#) に関するヘルプが必要な場合は、[Docker documentation](#) にご相談ください。

MacOS X では、[install "Docker Desktop for Mac"](#) できます

アプリケーションメニューから Docker を起動します。

12.1.2 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、

[local-repo](#) を介した各ノードへのカスタムリポジトリの追加をサポートしており、[tpaexec download-packages](#) コマンドを使用して必要なパッケージを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は、[TPA をインストールする場所](#) の指示に従ってください

それをインストールします。

非接続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てそのファイルをコピーし、そのディレクトリで `tpaexec relink` を実行して、通常 `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にアクセスしない環境用に新しい構成を作成するには、次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 ...
```

これにより、`--enable-local-repo` が行うすべてのことが実行され、他のすべてのパッケージリポジトリの構成が無効になります。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 に `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に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。

using local-repo for distributing packages 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 installed* があることを確認し、次を実行します。

```
tpaexec download-packages cluster-dir --os <OS> --os-version <version>
```

:ref:`package downloader <Installing packages>` の詳細な説明を参照してください。

パッケージをターゲット環境にコピーする

結果のリポジトリは `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ーから、クラスターのデプロイに使用される切断されたコントローラーにコピーします。cluster ディレクトリの下の同じ場所にディレクトリを配置します。deploy が実行されると、TPA は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` と `deploy` を実行します。

非接続環境での更新

`upgrade` 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ーで `download-packages`

を実行し、更新されたリポジトリを切断されたコントローラーにコピーし、**そして** `deploy` を実行してこれらのパッケージをターゲットインスタンスにコピーする必要があります。

12.1.3 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、積極的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「レガシーディストリビューション」のリストが表示されます。必要なパッケージにアクセスできる限り、レガシープラットフォームへの展開は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、実稼働での使用には適していません。

完全にサポートされたプラットフォームは、TPA を実行するためのホストシステムと、TPA が Postgres クラスターをデプロイ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 x86

- Debian 11/bullseye は完全にサポートされています
- Debian 10/buster は完全にサポートされています
- Debian 9/stretch はレガシーディストリビューションです
- Debian 8/jessie はレガシーディストリビューションです

Ubuntu x86

- Ubuntu 22.04/jammy は完全にサポートされています
- Ubuntu 20.04/focal は完全にサポートされています
- Ubuntu 18.04/bionic はレガシーディストリビューションです
- Ubuntu 16.04/xenial はレガシーディストリビューションです

RedHat x86

- RHEL/Rocky/AlmaLinux/Oracle Linux 9.x は完全にサポートされています (python3 のみ)
- RHEL/CentOS/Rocky/AlmaLinux 8.x は完全にサポートされています (python3 のみ)
- RHEL/CentOS 7.x に完全対応 (python2 のみ)

SLES

- SLES 15.x は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、ここで説明されている従来のディストリビューションでは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 containers* では利用できません。

12.1.4 TPA capabilities and supported software

- *Python requirements*
- *Supported distributions*

対応ソフトウェア

TPA は、次の主要コンポーネントをインストールして構成できます。

- Postgres 16、15、14、13、12、11
- EPAS (EDB Postgres Advanced Server) 16、15、14、13、12
- PGD 5、4、3.7
- pglogical 3、2 (オープンソース)
- pgd-cli および pgd-proxy
- HARP2
- 担当者
- Barman
- pgbouncer
- haproxy (PGD 3.7 でのみサポート)
- フェールオーバーマネージャー (EFM)
- Postgres Enterprise Manager (PEM)

12.2 Services

12.2.1 EDB Postgres Distributed configuration

TPA は、以前は 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 と呼ばれていた EDB Postgres Distributed（PGD）をインストールして構成できます。

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のみを介して行われます。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触れていますが、詳細については、[PGD documentation](#) 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 とすべての PGD インスタンスに依存関係をインストールします。

基本的な Postgres セットアップを完了して 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に従って PGD クラスターのセットアップに進みます。

インストール

TPA は、使用中の Postgres（Postgres、Postgres Extended、または EPAS など）のバージョンとフレーバーに応じて、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、インストールする PGD のメジャーバージョン（つまり、3、4、5）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（例：最新の 5.0.x をインストールするには '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に進みます。

TPA が実行する手順の概要は次のとおりです。

- 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()` を使用）
- `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（ノードグループのタイプとメンバーシップに応じて）。
- 個々のインスタンスで関連するノードグループに参加します
- 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、追加の構成を実行します。

(このプロセスには、各インスタンスでいくつかを順番に、そして他のものを並列に、複雑な一連のクエリを実行することが含まれます。手順を簡単にするために、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は、これらのクエリのほとんどを実行します。それ以外の点では、インスタンスは特別なものではなく、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** を持つ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。

readonly をロールに持つ PGD インスタンスはロジカルスタンバイノード (**pause_in_standby** が設定された PGD ノードグループに参加します) であり、プロモーションの対象です。

subscriber-only をロールに持つ 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、公開はしません。

そのロールに **witness** を持つ 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、アップストリームの PGD インスタンスの物理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、通常、**cluster_vars** で設定して、クラスター内のすべての PGD インスタンスに対して均一に設定されるようにする必要があります。場合によっては (**bdr_database** など)、異なるインスタンスに異なる値を設定できますが、結果は未定義です (たとえば、すべてのインスタンスの **bdr_node_groups** の値はまったく同じでなければなりません)。

cluster_vars の下のクラスター全体に均一の値を設定して、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_node_group

bdr_node_group の設定（デフォルト：クラスター名に基づく）は、インスタンスがどの PGD クラスターに属するかを識別します。また、外部コンポーネント（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の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）。

一般に、`tpaexec configure` は、選択され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
    ...
```

最初のエントリは、クラスターの **bdr_node_group** 用である必要があります。

リスト内の後続の各エントリは、**parent_group_name** を指定する必要があり、**node_group_type** を指定できます（オプション）。

各エントリには、グループオプションのオプションのキー/値マッピングがある場合があります。利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

bdr_child_group がインスタンスに（**bdr_node_groups** で言及されているグループの名前に）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* のオプションのリストです

PGD データベースに存在する必要があります（PGD 4.1 以降で使用可能）。

```
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** のエントリに対応している必要があります。

bdr_commit_scopes を明示的に設定した場合、TPA は必要に応じてコミットスコープを作成、変更、または削除し、データベースが構成と一致するようにします。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。

雑記

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** **フック** を呼び出します。

データベースの照合

TPA は、クラスター内のすべてのインスタンスの PGD データベースに同じ照合 (**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を持つことは、データ損失のリスクです。

古いバージョンの PGD

TPA は、BDR v1 (パッチ適用バージョンの Postgres 9.4 を使用)、v2 (Postgres 9.6 を使用)、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。

12.2.2 Barman

インスタンスに **config.yml** で **barman** ロールが付与されると、TPA はそれを **Barman** サーバーとして構成し、**backup** 設定で名前を付けた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
...

```

(次のページに続く)

```
- Name: two
  role:
    - barman
  ...
```

複数の postgres インスタンスは、それらの backup と名前付けされた同じ 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 として名前付けられた Barman サーバーのリストを持つことができ、名前付けされたすべてのサーバーにバックアップが取られます。

デフォルトの Barman 構成は、pg_receivewal を使用して PostgreSQL に接続して WAL の継続的なバックアップを取得し、週 2 回 ssh 経由で rsync を使用してインスタンスの完全バックアップを取得します。完全バックアップと WAL は、過去 4 週間の任意の時点へのリカバリを可能にするのに十分な期間保持されます。

Barman 構成

各 barman サーバーで、グローバル構成ファイルが/etc/barman.confとして作成されます。これには、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。サーバー one をバックアップするため、ファイルは/etc/barman.d/one.conf になり、バックアップは/var/lib/barman/one に保存されます。ファイルとディレクトリの名前は、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、デフォルトでインスタンスの名前になります。

次の変数は、バックアップされたインスタンスで設定でき、プレフィックス barman_ が削除されて 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
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	

次のページに続く

表 1 – 前のページからの続き

variable	default
barman_post_backup_script	
barman_streaming_wals_directory	

バックアップスケジュール

TPA は、毎分実行され、`barman cron` を呼び出してメンテナンスタスクを実行する `/etc/cron.d/barman` に cron ジョブをインストールします。

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 `/etc/cron.d/<backup_name>` に別の cron ジョブを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` 変数の決定に従って実行されます。デフォルトでは、毎週水曜日と土曜日の 04:00 にバックアップが作成されます。

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、互いの `authorized_keys` ファイルに追加します。`postgres` ユーザーは、WAL セグメントをアーカイブするために `barman` サーバーに ssh できる必要があります（構成されている場合）。`barman` ユーザーは、Postgres インスタンスに ssh 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。

12.2.3 Configuring EFM

`failover_manager` が `efm` に設定されている場合、TPA は EFM をインストールして構成します。

EFM は EDB のパッケージリポジトリを介してのみ利用可能であり、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します。TPA はまた、基本的にデフォルトでは何も含まない `efm.notification.sh` スクリプトを配置し、ユーザーが自由に入力できるようにします。

EFM documentation を参照

EFM 構成の詳細については。

efm_conf_settings

efm_conf_settings を使用して、TPA で認識されるかどうかに関係なく、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引用する必要があります。

```
cluster_vars:
  efm_conf_settings:
    standby.restart.delay: 1
    application.name: quarry
    reconfigure.num.sync: true
    reconfigure.num.sync.max: 1
    reconfigure.sync.primary: true
```

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 EFM を witness として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 レプリカを設定します。repmgrd つまりこの場合、repmgr のデーモンは無効のままであり、repmgr の唯一の仕事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、どのような形式でも repmgr を使用しません。

12.2.4 Configuring haproxy

TPA は、role に haproxy が含まれるインスタンスに haproxy をインストールして構成します。

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 [Configuring 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 の利用可能な最新バージョンをインストールします。haproxy_package_version: 1.9.15* を設定することで、代わりに特定のバージョンをインストールできます (例)。

注: [tpaexec-configure](#) の package_version でワイルドカードを使用する場合の制限を参照してください。

Haproxy パッケージは、アーキテクチャの種類に応じて選択されます。EDB managed 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 extras リポジトリ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
<code>haproxy_bind_address</code>	127.0.0.1	haproxy がバインドするアドレス
<code>haproxy_port</code>	5432 (5444 for EPAS)	TCP ポート haproxy はリッスンする必要があります
<code>haproxy_read_only_port</code>	5433 (5445 for EPAS)	読み取り専用ロードバランサーの TCP ポート
<code>haproxy_backend_servers</code>	None	Postgres インスタンス名のリスト
<code>haproxy_maxconn</code>	<code>`max_connections` × 0.9</code>	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> 設定から導出されます
<code>haproxy_peer_enabled</code>	True*	既知の haproxy ホストを <code>`peer`</code> リストとして追加します。 <code>
*False`</code> <code>`failover_manager`</code> がハープまたはパトニーの場合。

読み取り専用ロードバランサー

Haproxy は、データベースへの読み取り専用アクセスのために追加のポートでリッスンするように構成できます。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ているかを判断し、ラウンドロビン負荷分散アルゴリズムを使用してトラフィックを送信します。

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` を使用してオンにできます。

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 行を持つバックエンドで `/etc/haproxy/haproxy.cfg` を生成します。最初のサーバーを除くすべてが「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、オプションを `default-server` 行に追加します。 `haproxy_server_options` をバックエンドサーバー上のオプションのリストに設定して、オプション（デフォルトをオーバーライドします）を各バックエンドの個々のサーバー行に追加します。

例

```
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
```

12.2.5 Configuring HARP

failover_manager が harp に設定されている場合、TPA は HARP をインストールして構成します。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 の設定

HARP documentation を参照

HARP 構成の詳細については。

表 3: :header: Variable,Default value,Description :width: 18,8,20 :align: left :class: longtable

<i>cluster_name</i>	``	クラスターの名前。
<i>harp_consensus</i>	``	使用するコンセンサスレイヤー (<i>etcd</i> または <i>bdr</i>)
<i>harp_location</i>	location	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_timeout</i>	10	更新されない場合、ノードの準備状況ステータスが持続する時間 (秒)。
<i>harp_leader_lease</i>	6	Lead Master リースが更新されない場合、持続する時間 (秒)。
<i>harp_lease_refresh</i>	2000	Lead Master リースの更新間隔のミリ秒単位の時間。
<i>harp_dcs_reconnect</i>	1000	切断されたノードが DCS への再接続を試行する間隔 (ミリ秒単位)。
<i>harp_dcs_priority</i>	500	2つのノードに等量のラグとリードマスターリースを取得するためのその他の適格基準がある場合、これは、あるノードを別のノードよりも優先する追加のランキング値として機能します。
<i>harp_stop_data</i>	false	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_no</i>	false	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_lsn</i>	1048	リードマスターロックの取得が許可される前に、以前のリードマスターの最後に記録された LSN とこのノードとの間の最大許容差異 (バイト単位)。
<i>harp_maximum_lsn</i>	1048	最後に受信した LSN と、このノードとその CAMO パートナーとの間で適用された LSN との間の最大許容差異 (バイト単位)。
<i>harp_camo_enqueue_lag</i>	lag	CAMO キューの状態を厳密に適用するかどうか。
<i>harp_use_unix_socket</i>	false	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>	250	DCS へのクエリが成功するまでの時間 (ミリ秒)。
<i>harp_watch_polling</i>	500	DCS のポーリング間でスリープするミリ秒。`harp_consensus_protocol` が <code>bdr</code> の場合にのみ適用されます。
<i>harp_proxy_timeout</i>	1	Lead Master への組み込みプロキシ接続タイムアウト (秒単位)。
<i>harp_proxy_keepalive</i>	5	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_connections</i>	75	harp-proxy が受け入れるクライアントの最大接続数 (<i>max_client_conn</i>)
<i>harp_ssl_password_command</i>	None	stdin で難読化された sslpassword を受け取り、処理された sslpassword を stdout 経由で提供するカスタムコマンド。
<i>harp_db_request_timeout</i>	10s	dcg -> 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-config</i> <i>hook</i> を使用 できます		

HARP 構成ファイルをインストールした後にタスクを実行するため (例、追加の構成ファイルをインストールするため)。

コンセンサスレイヤー

`tpaexec configure` への `--harp-consensus-protocol` 引数は、BDR-Always-ON アーキテクチャには必須です。

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` には次のパラメーターを構成できます。

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` を `bdr` に設定します。この場合、既存の PGD インスタンスがコンセンサスに使用され、これ以上の構成は必要ありません。

harp プロキシ用に別のユーザーを設定する

`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` を設定することで指定できます。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。

カスタム SSL パスワードコマンド

`harp_ssl_password_command` によって提供されるコマンドは、HARP によって使用され、接続文字列で指定された `sslpassword` の難読化が解除されます。`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` は必要ないため、指定しないでください。

harp サービスの設定

`harp` サービスには次のパラメーターを設定できます。

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。

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>"
```

`secure: true` を使用し、独自の証明書とキーを使用する場合は、`cert_file` および `key_file` キーの両方が必要です。

`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上の指定された場所で利用可能であ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 で証明書とキーを生成する場合は、`cert_file` と `key_file` の両方を空のままにします。TPA CA 証明書は「よく知られた」ものではありません。エンドポイントをプローブする各マシンのトラストストアにこの証明書を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にあります。`<cluster_dir>/ssl/CA.crt` after deploy。

利用可能な api エンドポイントの詳細については、harp のドキュメントを参照してください。

12.2.6 Configuring Postgres Enterprise Manager (PEM)

`tpaexec configure` コマンドが `--enable-pem` コマンドラインオプションで実行されると、TPA は PEM をインストールして構成します。

`--enable-pem` のデフォルトの動作は、クラスター内のすべての `postgres` インスタンスに対して `pem-agent` ロールを有効にすることです。`--enable-pg-backup-api` コマンドラインオプションが `--enable-pem` と一緒に使用される場合、`pem-agent` ロールも `barman` ノードに追加されます。

`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` インスタンスは暗黙的に `postgres` ロールも使用します。これにより、`pemserver` は 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` クラスターを取得できます。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。

- [Configure pg_hba.conf](#)
- [Configure 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を注意してください。

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が、EDB Postgres Extended で使用される場合の BDR -Always-ON アーキテクチャは例外です。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 構成

TPA は、`pem` エージェントと `pem` サーバーを適切なインスタンス固有の設定で構成し、残りの設定をそれぞれのデフォルト値に設定します。一部の構成オプションは、将来のある時点でユーザー構成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` で実行するように構成され、同じために 443 ポートを使用します。PEM の Web サーバー構成では、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` に、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。

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を監視および管理するために単一の PEM サーバーを使用したい場合があります。`tpaexec` 内の共有 `pem` サーバー展開は、既存の `pem` サーバーの使用を計画してい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、指定されたクラスター構成に、`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デプロイ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ようになります。

1. `pem-server` ロールを持つ単一のインスタンスで `tpaexec` クラスターを作成します (この例では「`pem-cluster`」と呼びます)。`pemserver` として実行される単一のインスタンスだけでなく、より大きなクラスターの一部として `pem` が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、単一ノードクラスターを使用します。例として使用する方が簡単で、おそらくメンテナンスも簡単。
2. 他のクラスター (`pg-cluster` など) で、`$clusters/pem-cluster` からこの特定の `pemserver` を共有 `pem` サーバーインスタンスとして参照し、`bare` をプラットフォームとして使用するため、新しい `pemserver` イン

タンスを作成しようとしません。また、このクラスターが pemserver インスタンスへのアクセスに使用できる 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 の公開鍵を pem サーバーインスタンスにコピーしてから 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することにより、このアクセスを許可できます。

```
# 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
```

`tpaexec provision $clusters/pem-cluster` を実行してから `tpaexec deploy $clusters/pem-cluster` を実行します。完了すると、新しい pg-cluster のノードは pem サーバーバックエンドと話すことができるはずです。

- pg-cluster 内のノードからの 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 `export EDB_PEM_CREDENTIALS_FILE=/path/to/pem/credentials/file` を実行する必要があります。資格情報ファイルは、pemserver 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
```

バックエンドの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
```

- `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-cluster にデプロイされるようにします。

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\TI\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、それぞれ `postgresql` および `epas` フレーバーに `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

:ref:`PEM documentation <Configuring Postgres Enterprise Manager (PEM)>` を参照
```

PEM の設定と使用法の詳細については。

12.2.7 Configuring pgbouncer

TPA は、role に `pgbouncer` が含まれるインスタンスに `pgbouncer` をインストールして構成します。

デフォルトでは、`pgbouncer` はポート 6432 で接続をリッスンし、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。

`pgbouncer` インスタンスで次の変数を設定できます。

Variable	Default value	Description
<code>pgbouncer_port</code>	6432	<code>pgbouncer</code> がリッスンする TCP ポート

次のページに続く

表 4 – 前のページからの続き

Variable	Default value	Description
<code>pgbouncer_backend</code>	127.0.0.1	接続する Postgres サーバー
<code>pgbouncer_backend_port</code>	5432	<code>pgbouncer_backend</code> がリスンするポート
<code>pgbouncer_max_client_conn</code>	<code>max_connections</code> × 0.9	許可される接続の最大数。可能であれば、デフォルトはバックエンドの <code>max_connections</code> 設定から派生します
<code>pgbouncer_auth_user</code>	<code>pgbouncer_auth_user</code>	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。以下の例に示すように `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） `auth_query` を実行してユーザーを認証します。

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:
```

(次のページに続く)

(前のページからの続き)

```
host: two
port: 6543
```

12.2.8 Configuring pgd-proxy

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 をインストールして構成します。

(デフォルトでは、*PGD-Always-ON architecture*)

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` を使用して追加のプロキシインスタンスをいくつでも作成できます。)

構成

pgd-proxy は、SQL 関数を介して PGD レベルで構成されます。

Hash	Function	Description
<code>pgd_proxy_options</code>	<code>bdr.alter_proxy_option()</code>	pgd-proxy 構成、例ポート
<code>bdr_node_groups</code>	<code>bdr.alter_node_group_option()</code>	プロキシのノードグループの構成、例 <code>'enable_proxy_routing'</code>
<code>bdr_node_options</code>	<code>bdr.alter_node_option()</code>	個々の PGD ノードのルーティング構成

詳細については、PGD のドキュメントを参照してください。

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-proxy を有効にするには、`enable_proxy_routing` を `true` に明示的に設定する必要があります。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、任意の 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

アタッチされているグループまたはノードではなく、pgd-proxy インスタンス自分自身のオプションは、`cluster_vars`（すべてのプロキシに適用）の下で `default_pgd_proxy_options` の下、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

instances:

- Name: someproxy
  vars:
    pgd_proxy_options:
      listen_port: 9000
```

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

プロキシの状態を監視するための 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` の下の `pgd_http_options` またはインスタンス `vars` 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(s) 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>"
```

`secure: true` を使用し、独自の証明書とキーを使用する場合は、`cert_file` および `key_file` キーの両方が必要です。

`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上の指定された場所で利用可能であ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 で証明書とキーを生成する場合は、`cert_file` と `key_file` の両方を空のままにします。TPA CA 証明書は「よく知られた」ものではありません。エンドポイントをプローブする各マシンのトラストストアにこの証明書を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にあります。`<cluster_dir>/ssl/CA.crt` after `deploy`。

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。

12.2.9 pglogical configuration

TPA は、`pglogical v2` および `pglogical v3` で `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
  ...
  vars:
    publications:
      - type: pglogical
        database: example
        name: some_publication_name
        replication_sets:
          - name: custom_replication_set
            ...

- node: 2
  Name: keeper
  vars:
    subscriptions:
      - type: pglogical
```

(次のページに続く)

(前のページからの続き)

```

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、デフォルトで `pglogical` 拡張機能が作成されますが、どのバージョンをインストールするかを決定するのはあなた次第です（たとえば、`pglogical3` の `products/pglogical3/release` リポジトリに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` には名前付き公開エンティティがありません（組み込み論理レプリケーションに `CREATE PUBLICATION` があるという意味で）。`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るだけで、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。

`pglogical` 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に名前付きローカル `pglogical` ノードが必要です。TPA は、存在しない場合、`pglogical.create_node()` でこのノードを作成します。パブリケーションの場合、パブリケーション名は `pglogical` 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 `pglogical` ノードしか存在できないため、データベースごとに `publications` に 1 つのエントリのみを持つことができます。

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、`pglogical` 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、同じデータベースでパブリケーションを構成し、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 `pglogical` ノードを共有する必要がある場合。

TPA は、構成の基本的な検証を行います。`replication_sets` を `replciation_sets` と綴った場合、または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した場合、エラーを指摘しますが、パブリケーションとサブスクリプションの意味のある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、および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。

`postgres_users` および `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 が構成される前にデータベースにデータを投入するためのフック。

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持つ名前付き 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 や autoadd_existing などのオプションの属性を指定できます。指定された場合、それらは pglogical.create_replication_set() への呼び出しに名前付き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、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以外に、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付き（非修飾名は public にあると想定されます）、上記のように、エントリーには row_filter（テーブルのみ）または synchronize_data などのオプションの属性を含めることができます。

サブスクリプション

subscriptions のエントリーは、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ます。サブスクリプションの他のオプションの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（上記のように）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。その名前のパブリケーションが（クラスター全体にわたって）複数ある場合は、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で参照する場合は、同じインスタンスに同じ名前の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=..."
```

出版物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()` に直接渡されます（そして `publication` は無視されます）。このメカニズムを使用して、TPA クラスター外部のインスタンスに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です。定義されている場合、それらは `pglogical.create_subscription()` への呼び出しに名前付き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。（表示されているいくつかの属性は `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、またはメンバーシップ（それに含まれるテーブルとシーケンス）の変更を行うことができます。

`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（ただし、既存のレプリケーションセットの `autoadd_existing` を変更することはできず、`autoadd_*` パラメーターはすべて `pglogical3` 固有です）。

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、`synchronize_data` またはすでに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、テーブルを削除して、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複製セットは削除されます（サブスクライバーに `pglogical.alter_subscription_remove_replication_set()`、またはパブリッシャーに `pglogical.drop_replication_set()` を使用します。ただし、`default`、`default_insert_only`、および `ddl_sql` という名前のデフォルトの複製セットは削除されません。）

`config.yml` を編集する場合は、`tpaexec deploy` を実行する前に `tpaexec provision` を実行することを忘れないでください。

PGD との相互作用

注意を払えば、同じデータベースで PGD と `pglogical` を一緒に使用することができます。

PGD v3 は `pglogical3` を内部で使用し、`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 1 つの `pglogical` ノードしか存在できないため、`bdr_database` で `pglogical` 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。そうしないと、最初にパブリケーション用にノードが作成され、`bdr.create_node()` は後でノード名の競合に関するエラーで失敗します。`bdr_database` の `subscriptions` も同じ `node_name` を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。

12.2.10 Configuring 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>` です。ここで `<version>` は、このインスタンスにインストールされている `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設定します。

repmgr 構成

次のインスタンス変数を設定できます。

`repmgr_priority` : 構成ファイルに `priority` を設定します `repmgr_location` : 構成ファイルに `location` を設定します : 構成ファイルに `failover` を設定します、デフォルトは `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。

12.3 Configuration

12.3.1 How TPA uses 2ndQuadrant and EDB repositories

このページでは、TPA が EDB (2ndQuadrant を含む) ソフトウェアをダウンロードできるパッケージソース、選択したソフトウェアによってソースがどのように異なるか、各ソースへのアクセスを構成する方法について説明します。

このページでは、EDB および 2ndQuadrant ソースの特別な構成オプションとロジックのみについて説明していることに注意してください。任意の *Configuring YUM repositories* または *Configuring APT repositories* リポジトリは、ここで説明されているロジックとは無関係に追加できます。同様に、パッケージは *downloaded in advance* にすることができます

必要に応じて *Creating and using a local repository* に追加されました。

TPA が使用するパッケージソース

TPA は、3 つのパッケージソースからソフトウェアをダウンロードします。これらの各ソースは、複数のリポジトリを提供します。場合によっては、同じソフトウェアが複数のソースから入手できます。

- EDB Repos 2.0
- EDB Repos 1.0
- 2ndQuadrant Repos

デフォルトでは、TPA は `select sources and repositories automatically`

指定したアーキテクチャおよびその他のオプションに基づいているため、通常、これらを変更する必要はありません。ただし、使用するすべてのソースに対して有効なサブスクリプションがあり、`exported the token` を持っていることを確認する必要があります

`tpaexec deploy` を実行する前にしないと、操作は失敗します。

注釈: EDB は、Repos 2.0 を介してすべてのソフトウェアを公開する過程にあり、最終的に古いリポジトリを削除します。

パッケージソースによる認証

EDB Repos 2.0 を使用するには、`tpaexec` を実行する前に `export EDB_SUBSCRIPTION_TOKEN=xxx` を使用する必要があります。

`the web interface` からサブスクリプショントークンを取得できます。

2ndQuadrant repositories を使用するには、`tpaexec` を実行する前に `export TPA_2Q_SUBSCRIPTION_TOKEN=xxx` する必要があります。サブスクリプショントークンは、2ndQuadrant Portal の左側のメニューの [Company info]、[Company] の下から取得できます。一部のリポジトリは、事前の手配によってのみ利用できます。

EDB Repos 1.0 を使用するには、アクセス資格情報を `username:password` 形式で含むテキストファイルを作成し、`tpaexec` を実行する前に `export EDB_REPO_CREDENTIALS_FILE=/path/to/credentials/file` を実行する必要があります。

リストされているサイトのアカウントをお持ちでない場合は、<https://www.enterprisedb.com/user/register?destination=/repository-access-request> でアクセスに登録できます。

デフォルトでソースが選択される方法

PGD-Always-ON アーキテクチャが選択されている場合、リポジトリは EDB Repos 2.0 から選択され、すべてのソフトウェアはこれらのリポジトリからソースされます。

M1 アーキテクチャが選択され、独自の EDB ソフトウェアが選択されていない場合、すべてのパッケージは PGDG から提供されます。プロプライエタリの EDB ソフトウェアを選択した場合、すべてのパッケージは EDB Repos 2.0 からソースされます。

BDR-Always-ON アーキテクチャの場合、デフォルトのソースは 2ndQuadrant であり、このソースから必要なリポジトリが追加されます。さらに、PGDG リポジトリは、必要に応じて PostgreSQL や etcd などのコミュニティパッケージに使用されます。2ndQuadrant リポジトリで利用できない EDB ソフトウェアが必要な場合（例 EDB Advanced Server）、リポジトリは EDB Repos 1.0 から選択されます。

EDB 2.0 リポジトリの指定

EDB Repos 2.0 から各インスタンスにインストールする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` および `postgres_distributed` リポジトリを構成し、EPAS および PGD5 製品へのアクセスを許可します。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。

EDB リポジトリが指定されている場合、指定された 2ndQuadrant リポジトリは無視され、EDB Repos 1.0 はインストールされません。

2ndQuadrant リポジトリの指定

2ndQuadrant パブリックリポジトリに加えて、各インスタンスにインストールする 2ndQuadrant リポジトリの完全なリストを指定するには、`tpa_2q_repositories` を 2ndQuadrant リポジトリ名のリストに設定します。

```
cluster_vars:
  tpa_2q_repositories:
    - products/pglogical3/release
    - products/bdr3/release
```

この例では、`pglogical3` および `bdr3` リリースリポジトリをインストールします。Debian および Ubuntu システムでは APT リポジトリを使用し、RedHat システムでは YUM リポジトリを使用します。

`dl/default/release` リポジトリは、デフォルトで常にインストールされます。

- `tpa_2q_repositories`: [] を明示的に設定する、または
- `edb_repositories` に少なくとも 1 つのエントリがあります。

上記のいずれかまたはどちらかを実行すると、2ndQuadrant リポジトリがインストールされません。

12.3.2 Configuring EDB Repos 2.0 repositories

このページでは、システムで EDB Repos 2.0 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。

TPA が使用する EDB および 2ndQuadrant パッケージソースの詳細については、

[this page](#) を参照してください。

EDB Repos 2.0 から各インスタンスにインストールする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、エンタープライズサブスクリプションリポジトリと `postgres_distributed` をインストールして、EPAS および PGD5 製品へのアクセスを提供します。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。

EDB リポジトリが指定されている場合、指定された 2ndQuadrant リポジトリは無視され、EDB Repos 1.0 はインストールされません。

EDB Repos 2.0 を使用するには、`tpaexec` を実行する前に `export EDB_SUBSCRIPTION_TOKEN=xxx`

を使用する必要があります。

[the web interface](#) からサブスクリプショントークンを取得できます。

12.3.3 Configuring 2ndQuadrant repositories

このページでは、任意のシステムで 2ndQuadrant 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。

TPA が使用する EDB および 2ndQuadrant パッケージソースの詳細については、

[this page](#) を参照してください。

2ndQuadrant パブリックリポジトリに加えて、各インスタンスにインストールする 2ndQuadrant リポジトリの完全なリストを指定するには、`tpa_2q_repositories` を 2ndQuadrant リポジトリ名のリストに設定します。

```
cluster_vars:
  tpa_2q_repositories:
    - products/pglogical3/release
    - products/bdr3/release
```

この例では、pglogical3 および bdr3 リリースリポジトリをインストールします。Debian および Ubuntu システムでは APT リポジトリを使用し、RedHat システムでは YUM リポジトリを使用します。2ndQuadrant リポジトリは、SLES システムでは使用できません。

2ndQuadrant repositories を使用するには、tpaexec を実行する前に

`export TPA_2Q_SUBSCRIPTION_TOKEN=xxx` する必要があります。サブスクリプショントークンは、2ndQuadrant Portal の左側のメニューの [Company info]、[Company] の下から取得できます。一部のリポジトリは、事前の手配によってのみ利用できます。

`dl/default/release` リポジトリは、デフォルトで常にインストールされます。

- `tpa_2q_repositories`: `[]` を明示的に設定する、または
- `edb_repositories` に少なくとも 1 つのエントリがあります。

上記のいずれかまたはどちらかを実行すると、2ndQuadrant リポジトリがインストールされません。

12.3.4 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` で名前付きリポジトリを定義でき、`apt_repository_list` に名前をリストすることにより、使用するリポジトリを決定できます。

```
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
```

この構成は、GPG キー (`key_url` から取得した `id key_id`、`id key_id`) と PGDG リポジトリ (デフォルトで既に定義されています) と新しいサンプル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 パッケージを引き続き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。

`apt_repository_list`: `[]` を設定して、リポジトリをインストールしないようにできます。

12.3.5 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` で名前付きリポジトリを定義でき、`yum_repository_list` に名前をリストすることにより、使用するリポジトリを決定できます。

```
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」（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 キーを慣習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す。

それ以外の場合は、説明、`baseurl`、および `gpgkey` URL を指定でき、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。正しい操作には EPEL リポジトリが必要なので、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。

`yum_repository_list`: `[]` を設定してリポジトリをインストールしないようにできます（ただし、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ために特別な手順を実行する必要がある場合は、

`pre-deploy hook` を使用して `.repo` ファイルを自分で作成できます。

```
- name: Define Example repository
  copy:
    dest: /etc/yum.repos.d/example.repo
    owner: root
    group: root
```

(次のページに続く)

(前のページからの続き)

```

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。

12.3.6 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使用できるようにします。これは、クラスターに追加のパッケージを出荷する簡単な方法です。

必要に応じて、このリポジトリを **のみ** 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、つまり、他のすべてを無効にすることもできます。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 に作成してもらいます。両方の手順を以下に示します。

注釈: *managing clusters in an air-gapped* 環境用の特定の命令が利用可能です。

TPA でローカルリポジトリを作成する

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` と一緒に使用して、ローカルリポジトリとして使用され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。

details of the layout 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入力し、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 `local-repo` 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るため、すぐに使用できます。

ローカルリポジトリを手動で作成する

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中に OS 固有のサブディレクトリ (`local-repo/Debian/10` など) 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のレイアウトは次のとおりです。

```
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に探し、見つけた最初のサブディレクトリを使用します。

- `<distribution>/<version>`、例えば、RedHat/8.5
- `<distribution>/<major version>`、例えば、RedHat/8
- `<distribution>/<release name>`、例えば、Ubuntu/focal
- `<distribution>`、例えば、Debian
- `local-repo` ディレクトリ自身。

これらのディレクトリがいずれも存在しない場合、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。

リポジトリへの入力とメタデータの生成

`tpaexec deploy` を実行する前に、以下で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入力するには、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、以下で説明するように、システムに適したツールを使用してメタデータを生成します。

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。

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/10`` 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/buster
# download/copy .deb package files

$ dpkg-scanpackages . | gzip > Packages.gz
```

RedHat / SLES リポジトリのメタデータ

まず、`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

$ 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 パッケージをローカルリポジトリに含める必要があります。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、インストールしてから残りをコピーします。

リポジトリ構成

ローカルリポジトリの内容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）リポジトリとして構成します。

リポジトリディレクトリに `example.deb` などを提供する場合、`apt-get install example` を実行すると、他のリポジトリのパッケージと同様にインストールされます。

パッケージのインストール

TPA は、指定したコンテンツでリポジトリを構成しますが、同じパッケージが異なるリポジトリから入手可能な場合、どれをインストールするかを決定するのはパッケージマネージャーです（特定のバージョンを指定しない限り、通常は最新）。

（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` で提供するパッケージのみを使用できます。）

12.3.7 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
      build_directory: /opt/postgres/build/otherext
```

(次のページに続く)

(前のページからの続き)

```

build_commands:
  - "make -f /opt/postgres/src/otherext/Makefile install"
build_environment:
  VAR: value

```

TPA はリストされている順序で拡張機能を 1 つずつビルドおよびインストールするため、正しい順序で言及することにより、別の拡張機能 (pglogical や BDR など) をビルドできます。

各エントリは、ビルドする `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。 *SSH agent forwarding or an HTTPS username/password* を使用できます

Git リポジトリに対して認証します。上記のように `source_directory`、`build_directory`、`build_environment`、および `build_commands` も設定します。

繰り返しデプロイを行うときにビルドディレクトリを再利用するには、`tpaexec deploy ... --skip-tags build-clean` を実行します。(それ以外の場合、ビルドを開始する前に古いビルドディレクトリが空になります。)

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。

開発ビルドをスピードアップします。

ソースビルドを実行するたびに、Postgres が再起動されます。

依存関係をビルドする

ソースからビルドしている場合、TPA は基本的な Postgres ビルドの依存関係がインストールされていることを保証します。追加のパッケージが必要な場合は、

TPA パッケージのインストール に記載してください。例

```

cluster_vars:
  packages:
    common:
      - golang-1.16

```

12.3.8 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 *install Postgres from source* または `use `install_from_source <Installing from source>`` に変更して拡張機能をコンパイルしてインストールするときに必要な場合があります。

ターゲットインスタンスのディスクに資格情報を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、ローカル `ssh-agent` に SSH キーを追加できます。 `--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っています。

- `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` できます。

注釈: macOS 上の Docker にデプロイする場合、この環境のホストから ssh で Docker コンテナにアクセスできないため、`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`) がある場合は、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。デスクトップ環境でこれがまだ設定されていない場合 (ほとんどの場合と同様に、`pgrep ssh-agent` が実行されているかどうかを確認します)、`ssh-agent bash` を実行してエージェントを有効にして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` (秘密鍵ファイル) 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、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特別に生成された使い捨てキーペアを使用してください。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特権を持つユーザーは、他のサーバーへの認証中にエージェントを利用してあなたになりますことができます。

HTTPS 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、環境内の `tpaexec deploy` の前に `export TPA_GIT_CREDENTIALS=username:token` のみ。展開中に、これらの資格情報は、任意の `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。ターゲットインスタンスのディスクには書き込まれません。

12.3.9 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（例、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、展開中および後のクラスター管理コマンドの実行中に、これらの設定が環境に（必要な他の設定とともに）存在することを保証します。

これらの環境設定は永続的ではありませんが、代わりに `extra_bashrc_lines` を使用して postgres ユーザーの環境変数を設定できます。

12.3.10 Python environment

TPA は、ターゲットインスタンスの *distribution it detects* に基づいて、使用する Python インタープリターを決定します。可能な限り 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従って `preferred_python_version` を設定します。

Distribution	Python 2	Python 3
Debian 10/buster	✓	✓ (3.7)
Debian 9/stretch	✓	✓ (3.5)
Debian 8/jessie	✓	☒ (3.4)
Ubuntu 16.04/xenial	✓	✓ (3.5)
Ubuntu 18.04/bionic	✓	✓ (3.6)
Ubuntu 20.04/focal	☒	✓ (3.8)
Ubuntu 22.04/jammy	☒	✓ (3.10)
RHEL 7.x	✓	☒ (3.6)
RHEL 8.x	☒	✓ (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 は Python 3.6 に同梱されていますが、Python 3 の `librpm` バインディングは利用できないため、TPA は代わりに Python 2 を使用する必要があります。Debian 8 には、Ansible のサポートに必要な Python 3.5+がありません。

他のディストリビューションについて `python2` または `python3` のどちらを好む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` で既にデプロイされているシステムでも `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。

12.3.11 Configuring /etc/hosts

デフォルトでは、TPA はターゲットインスタンスの `/etc/hosts` に、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行を追加し、クラスター内の通信に互いの名前を使用できるようにします（例：`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:
- Name: one
  ...
  vars:
    extra_etc_hosts_lines:
      - 192.0.2.1 acid.example.com
      - 192.0.2.2 water.example.com
```

デフォルトの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は、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。

12.3.12 Filesystem configuration

TPA を使用すると、各インスタンスに添付された `volumes` のリストを定義できます。

このリストには、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が含まれています。

まず、`tpaexec provision` は情報を使用してボリュームを作成し、インスタンスにアタッチします（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）。次に、ボリュームの単純化されたリスト（非プラットフォーム固有の設定のみを含む）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、必要に応じて、単純化されたリストに基づいて動作し、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。

AWS クラスターからの適度に複雑な例を次に示します。

```
instances:

- Name: one
  ...
  volumes:
    - device_name: root
      volume_type: gp2
      volume_size: 32
    - raid_device: /dev/md0
      device_name: /dev/xvdf
      volume_type: io2
      volume_size: 64
      raid_units: 2
      raid_level: 1
      iops: 5000
      vars:
        volume_for: postgres_data
        encryption: luks
    - raid_device: /dev/md1
      device_name: /dev/xvdh
      ephemeral: ephemeral0
      raid_units: all
      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 2 つのプロビジョンド iops EBS ボリュームで構成される 64GB RAID-1 ボリューム、および使用可能なすべてのインスタンスで構成される `/tmp/scratch` ファイルシステムになります。-ストア（「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 のセクションに記載されていますが、`volume_type` および `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` または `mountpoint` のような `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ているため、`cluster_vars` で設定することはできません。

instance_defaults

メカニズムは、ボリューム定義を特別に許可します。大規模なクラスターでのボリューム定義は非常に反復的である可能性があるため（特に、クラスター内のインスタンスをできるだけ近くに構成することをお勧めするため、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
      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（インスタンス `one` の場合と同様、別のものを指定しません）。インスタンス `two` は同じルートボリュームを持ちますが、`/dev/xvdf` をオーバーライドして代わりに 64GB になり、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、追加の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`

でデフォルトの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、異なる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
    ...
  - device_name: /dev/xvdg
    ephemeral: ephemeral0
    ...
```

TPA は、`root` の `device_name` をインスタンスタイプに基づいて `/dev/sda` または `/dev/xvda` に変換するため、どちらを使用するかを覚え（または変更）する必要はありません。

`volume_type` は、EBS ボリュームタイプを指定します。例、`gp2`（「汎用」EBS ボリュームの場合）、プロビジョンド IOPS ボリュームの `io1`（この場合、`iops: 5000` も設定する必要があります）など。

`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 の機能であり、新しく生成された TPA 構成でデフォルトで有効になっており、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棄されないようにします（これがデフォルトの動作です）。

`ephemeral: ephemeralN` を設定して、以前はエフェメラルボリュームと呼ばれていた、物理的に接続された

instance store volume

を使用します。利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 ボリュームは気になるデータに使用します。

EBS ボリュームの場合、`snapshot: snap-xxxxxxx` を設定して、既存のスナップショットからボリュームをアタッチすることもできます。スナップショットから復元されたボリュームは、S3 から十分なデータが読み取られてローカルにキャッシュされるまで、非常に遅い場合があります。(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ンアップできますが、全負荷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。)

ボリュームの `attach_existing: yes` を設定し、名前/タイプ/サイズ/iops が一致する既存の接続されていない EBS ボリュームがある場合、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接続されますはじめて始まります。再アタッチされた 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` インスタンスに接続できるボリューム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。

プラットフォーム Docker

Docker コンテナにはボリュームをアタッチできますが、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。このように、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指定は、`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アタッチされ、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 ボリュームを定義できます。

```
instances:
- Name: one
  ...
  volumes:
  - raid_device: /dev/md0
    device_name: /dev/xvdf
    raid_units: 2
    raid_level: 1
    volume_type: gp2
    volume_size: 100
    vars:
      volume_for: postgres_data
```

この例では、4 × 100GB EBS gp2 ボリューム (/dev/xvd[f-i]) をアタッチし、それらを /dev/md0 という名前の RAID-1 ボリュームにアセンブルします。展開中の `volume_for` または `mountpoint` の処理は、他のボリュームと同様に発生します。

TPA は現在、他のプラットフォームでの RAID アレイの作成とアセンブリをサポートしていませんが、`device_name: /dev/md0` または `/dev/mapper/xyz` でボリュームにエントリを追加することにより、既存のアレイを使用できます。TPA は、他のブロックデバイスと同様に `mkfs` および `mount` を処理します。

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` が設定されたボリュームがまだ初期化されていない場合、TPA は `cryptsetup` を使用して最初に `luksFormat`、次に `luksOpen` を使用して、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 `/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に、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` を作成すると、TPA は Postgres をブート時に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、ボールドに保存されます。

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` に渡す `fsopts`（デフォルト：`none`）、および `mount` に渡して `fstab` に書き込む `mountopts` を指定できます（下記参照）。

TPA は、デバイスの先読みをデフォルトで 16MB に設定します（そして値をリブートしても持続します）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判断する方法は 2 つあります。`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、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`）で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、これらの属性は `mountpoint` で設定されます。デプロイメントのこの非常に早い段階では、`postgres` ユーザーが存在するとは当てにできないことに注意してください。いずれにせよ、TPA は（個別に）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保証するため、自分で行う必要はありません。

12.3.13 Uploading artifacts

`artifacts` を定義して、ファイルを作成またはターゲットインスタンスにコピーできます。

```
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、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ります）
- `file` を使用して、コントローラー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、コントローラーからターゲットインスタンスにディレクトリを `rsync` 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つのエントリを示していますが、これらまたは対応する Ansible モジュールが受け入れる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をターゲットインスタンスにコピーすることは、この機能が

custom hook

を記述する代わりに使用できる便利なショートカットを提供するための一般的なニーズです。

12.3.14 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` のトップレベルにある `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。対応する公開鍵は、同じ場所で `.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。）

12.3.15 Managing SSH host keys

TPA は、クラスターのプロビジョニング中に一連の SSH ホストキーを生成します。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 instances* にインストールされませんが、`manage_ssh_hostkeys` を設定して有効にできます。

```
instances:
- Name: one
  ...
  platform: bare
  vars:
    manage_ssh_hostkeys: yes
```

`:ref: `bare instances <bare(-metal servers)>`` のドキュメントで説明されているように、最初にクラスターディレクトリに正しいエントリを使用して ``known_hosts``

を設定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、ベアインスタンスで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、すべてのホストのエントリとその公開ホストキー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` ファイルに追加します。

12.3.16 Postgres source installation

`postgres_installation_method` を `src` に設定すると、TPA はソースから Postgres をコンパイルしてインストールします。この機能は、開発とテストでの使用を目的としており、同一に構成された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 をビルドして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（任意の有効な 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、

SSH agent forwarding or an HTTPS username/password 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`）に複製されるか、ディレクトリがすでに存在する場合（例、再デプロイする場合）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 `postgres_build_dir`（デフォルト：`/opt/postgres/build/postgres`）でデバッグを有効にし、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
```

これにより、`postgres_extra_configure_opts` のオプションと環境で定義された `postgres_extra_configure_env` の設定で `./configure` が実行されます。一部のオプションはデフォルトで指定されます（例、`--with-debug`）が、対応する `--disable-xxx` または `--without-xxx` オプションで否定できます。`--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"
```

繰り返しデプロイを行うときにビルドディレクトリを再利用するには、`tpaexec deploy ... --skip-tags build-clean` を実行します。（それ以外の場合、ビルドを開始する前に古いビルドディレクトリが空になります。）

[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。

開発ビルドをスピードアップします。

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してインストールできます。

[install_from_source](#) の構成方法に関する別のページがあります。

ただし、ソースから Postgres をインストールする場合、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、ソースからも拡張機能を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 and Postgres-related packages](#) については別ページです

`postgres_installation_method: pkg`（デフォルト）で。

12.3.17 Installing packages

TPA は、展開の早い段階で Postgres 関連以外のパッケージのバッチをインストールし、次にすべての Postgres 関連のパッケージと一緒にインストールし、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ない `sysstat` や `strace` などのパッケージのインストールに関するものです。

`cluster_vars` の下の `packages`、または `config.yml` の特定のインスタンスの `vars` 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 `default_packages` の独自のリストと `packages.common` にリストされているパッケージをすべてのインスタンスにインストールし、残りのディストリビューション固有のパッケージを、インスタンスが実行しているディストリビューションに基づいてインストールします。これらのパッケージのいずれかが利用できない場合、展開は失敗します。

Postgres に依存するパッケージをリストしないでください。

extra_postgres_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。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
```

12.3.18 Running initdb

TPA は、存在しない場合は最初に `postgres_data_dir` を作成し、正しい所有権、アクセス許可、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` ファイルが既に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` を初期化する場合、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
```

`--data-checksums` オプション（デフォルトで含まれています）を常に含めることをお勧めします。

TPA は、`initdb` を実行するときに、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) が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` の実行後に生成された構成ファイルを新しい場所に移動します。

12.3.19 Installing Postgres-related packages

TPA は、展開の早い段階で Postgres 関連以外のパッケージのバッチをインストールし、次にすべての Postgres 関連のパッケージと一緒にインストールし、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。

Postgres (Postgis など) に依存する追加のパッケージをインストールするには、`cluster_vars` の `extra_postgres_packages` または `config.yml` の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` 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の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。

compiling and installing Postgres from source については別ページです。

12.3.20 SSL Certificates

`config.yml` に `enable_pg_backup_api: true` を設定するか、構成中に `--enable-pg-backup-api` コマンドラインオプションを使用した場合、`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の Apache プロキシをセットアップします。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` 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、[configure-source.md](#) および [install_from_source.md](#) 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」が返ってきたら、`pg-backup-api` サービスが実行されています。

プロキシが機能していることをテストするには、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
```

動作している場合、大きな json 出力が得られます。これを `barman diagnose` の出力と比較できます。正確に一致するはずです。

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。

テスト (`tpaexec test` を介して) またはパーマンホストからのコマンドラインのために、クライアント証明書とキー (`pg-backup-user.crt` および `pg-backup-user.key`) が生成されます。

テスト 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` と `--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。

12.3.21 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、リブートしても持続します。

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 (存在する場合) よりも優先されます。設定は最初に `sysctl.conf` に 1 行ずつ追加され、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はこの手順を完全にスキップします。

12.4 Postgres configuration

12.4.1 Creating Postgres databases

デプロイメント中に Postgres データベースを作成するには、`cluster_vars` の下の `postgres_databases` または `config.yml` の特定のインスタンスの `vars` 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: exampletablespace
```

上記の例では、2つのデータベースを作成します（TPA 自分自身が作成することを決定した `bdr_database` などのデータベースを除く）。

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です。

`owner` はデフォルトでは `postgres` ですが、任意の有効なユーザー名（`postgres_users` のユーザーこの時点で作成されます）。

`encoding`、`lc_collate`、および `lc_ctype` の値は、デフォルトで `initdb` の実行時に設定された `postgres_locale` になります（デフォルトでは、ターゲットシステムの `LC_ALL` または `LANG` 設定が使用されます）。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は、`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。表領域はすでに存在している必要があります（`postgres_tablespace` を参照）。

`extensions` および `languages` のオプションのリストを指定して、各データベース内に作成できます（テンプレートデータベースから継承され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

`extra_postgres_packages` に含めるなどして、インストールしておく必要があります。

TPA は `postgres_databases` に記載されていない既存のデータベースを削除せず、必要に応じて（BDR 用など）追加のデータベースを作成する場合があります。

12.4.2 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、それらの名前と場所を `cluster_vars` の下の `postgres_tablespaces`、または `config.yml` の特定のインスタンスの `vars` 定義します。

`volume_for: postgres_tablespace` を設定し、`tablespace_name` を定義して *define volumes* を実行すると、それらはデフォルトのエントリとして `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が作成されます：明示的（/some/path）および暗黙的（ボリュームに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）。

すべての `postgres_tablespace` ボリュームには `tablespace_name` が定義されている必要があります。表領域の場所は、ボリュームのマウントポイントから導出されます。

`postgres_tablespaces` のすべてのエントリは、テーブルスペース名（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、テーブルスペースを PGDATA 内に置かないでください。また、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、表領域 owner は `postgres` ですが、任意の有効なユーザー名（*postgres_users* のユーザーこの時点で作成されます）。

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_tablespaces` 設定が必要です

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。

12.4.3 postgresql.conf

TPA は、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、メインの `postgresql.conf` で `include_dir` を使用して、これらの追加の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`) と `conf.d` の下に含まれるファイルは、常に `postgres_conf_dir` に保存されます。これはデフォルトでは `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` と `0001-tpa_restart.conf` です。これらには、それぞれ変更するにはサーバーのリロードまたは再起動が必要な設定が含まれています。展開中に、TPA は変更を正しいファイルに書き込み、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など）で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どこで正確に設定されているかを気にする必要はありません。

次に `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、`temp_buffers` や `maintenance_work_mem` などの変数を提供し、使用可能な `postgresql.conf` 設定の多く（すべてではありません）に直接設定できます。

`postgres_conf_settings` を使用して、TPA で認識されるかどうかに関係なく、パラメーターを設定できます。`postgresql.conf` に表示されるとおりに値を引用する必要があります。

```
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も優先されます。

`postgres_conf_settings` の下の値を変更した場合、TPA はリロードで変更を有効にできるかどうか、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、変更を有効にするために常にサーバーを再起動します。これが、可能な限り変数を直接使用することが常に好ましい理由です。

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（これは経験則であり、推奨事項ではありません）。このデフォルトをオーバーライドするには、`shared_buffers_ratio: 0.35` を設定して別の比率を使用するか、`shared_buffers_mb: 796` を特定の MB 数に設定するか、正確な値を直接指定します（例：`shared_buffers: "2GB"`）。

effective_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。このデフォルトをオーバーライドするには、`effective_cache_size_ratio: 0.35` を設定して別の比率を使用するか、`effective_cache_size_mb: 796` を特定の MB 数に設定するか、正確な値を直接指定します（例：`effective_cache_size: "8GB"`）。

shared_preload_libraries

TPA は、`shared_preload_libraries` のエントリが機能するために必要な拡張機能の内部リストを維持しており、そ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、それらを `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、`shared_preload_libraries` は `conf.d/8888-shared_preload_libraries.conf` に設定されます。

`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

以下に全引用符で囲んだ値を設定できます。この場合、値は `conf.d/9900-tpa_postgres_conf_settings.conf` に設定されます。

手で変更する

TPA で生成された構成の内容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（そして推奨される）オプションは、`ALTER SYSTEM` を使用することです。これは、常に構成ファイル内のすべてのファイルよりも優先されます。

```
# 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、構成が変更された場合、展開中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に `9999-override.conf` を変更しません。

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ります。

postgresql.conf をゼロから生成する

デフォルトでは、TPA は `include_dir` を追加する以外は、デフォルト（つまり `initdb` で生成された）の `postgresql.conf` ファイルをそのままにします。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、`postgres_conf_template` を設定してオーバーライドできます。

```
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。

12.4.4 pg_hba.conf

Postgres のドキュメントでは、`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、および認証済み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ーを追加できます。

```
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.j2 を作成して設定することもできます。

```
cluster_vars:
  postgres_hba_template: my_hba.j2
```

既存の pg_hba.conf をそのままにしたい場合は、それもできます。

```
cluster_vars:
  postgres_hba_template:
```

pg_hba.conf をインスタンスごとに異なるように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変更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お勧めします。

12.4.5 pg_ident.conf

通常、pg_ident.conf を変更する必要はありません。デフォルトでは、TPA は変更しません。

postgres_ident_template を設定して、pg_ident.conf を好きなコンテンツに置き換えることができます。

```
cluster_vars:
  pg_ident_template: ident.j2
```

cluster ディレクトリに templates/ident.j2 を作成する必要もあります。

```
{% for u in [unixuser1, unixuser2] %}
mymap {{ u }} dbusername
{% endfor %}
```

12.4.6 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 および repmgr のパスワードを使用して `~postgres/.pgpass` を作成します。 `pgpass_users` を設定して、別のユーザーリストのエントリを作成できます。

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
```

12.4.7 The postgres Unix user

このページでは、postgres ユーザーとそのホームディレクトリの構成方法について説明します。

Postgres users in the database の作成方法については別ページです。

シェル構成

TPA は .bashrc ファイルをインストールし、.profile または .bash_profile ファイルにも含まれていることを確認します。

ユーザー名、ホスト名、および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、任意の行を .bashrc に追加できます。（YAML マルチライン文字列構文 `>-` を使用して、シェルメタキャラクタのクォート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、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、`.ssh/authorized_keys` を編集して、クラスター内のインスタンスが `postgres` として互いに `ssh` できるようにします。

TLS 証明書

デフォルトでは、TPA はクラスター内で使用する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のに十分です。独自の証明書署名インフラストラクチャを使用したい場合は、展開が完了した後にこれらを置き換えるか、[フック](#) を使用して展開中に置き換えます。

ユーザー名

`postgres_user` および `postgres_group` 設定（どちらもデフォルトで `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は、それらを変更できます。

12.4.8 Creating Postgres users

展開中に Postgres ユーザーを作成するには、`cluster_vars` の下の `postgres_users` または `config.yml` の特定のインスタンスの `vars` 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 人のユーザーを作成します（`repmgr` や `barman` など、TPA 自分自身が作成することを決定したユーザーを除く）。

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` リスト内のロールは、新しく作成され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内の<username>_password という名前のボールド暗号化変数に保存します。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持たせたくない場合は、generate_password: false を設定できます。

12.5 Commands

12.5.1 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、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 の下にある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、すぐに tpaexec archive-logs を実行できます。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 **プロビジョニング** する必要があります。クラスターも **展開** すると、より多くのログが取得されます。

クイックスタート

```
[tpa]$ tpaexec archive-logs ~/clusters/speedy

PLAY [Prepare local host archive] *****
->*****

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

各インスタンスのログは、クラスターディレクトリの下にあります。

```
~/clusters/speedy/logs/
`-- 220220306T185049
    |-- khaki-logs-20220306T185049.tar.gz
    |-- kinship-logs-20220306T185049.tar.gz
    |-- uncivil-logs-20220306T185049.tar.gz
    `-- urchin-logs-20220306T185049.tar.gz
```

アーカイブ内容例：

```
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
|-- btmp
|-- dnf.librepo.log
|-- dnf.log
|-- dnf.rpm.log
|-- hawkey.log
|-- lastlog
|-- private
|-- tpaexec.log
`-- wtmp
```

12.5.2 tpaexec download-packages

ダウンローダーの目的は、既存の構成から TPA クラスターのフルインストールを行うために必要なパッケージを提供することです。これは、インターネットにアクセスできないセキュリティで保護された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しないようにする場合に役立ちます。

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配布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、クラスターの構築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 と YUM がサポートされています。

使用法

`tpaexec configure` コマンドを使用して達成できる既存のクラスター構成が存在する必要があります。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。

configuring a cluster 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。以下の例のようにプレフィックスが付いた 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入力されます。たとえば、docker イメージ `tpa/redhat:8` には次のものがあります。

```
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、対象の制御ノードにコピーできます。

recommendations for installing to an air-gapped environment を参照してください。*local-repo* 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に役立つダウンローダーコンテナを実行したままにします。たとえば、失敗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リング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、コンテナ名に許可されている制限である 64 文字を超えない限り、通常 `$cluster_name-downloader` です。 `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` を実行して、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` コマンドを実行して、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
```

12.5.3 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、組み込み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して実行する統一された方法を持つことは、誤解やオペレーターのエラー、または過去に正しかったがそれ以来現実から逸脱しているプロセス文書によって引き起こされるエラーの可能性を減らします。

Ansible Playbook を作成するということは、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則に従うことは、`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事実を利用でき、接続、認証、その他の基本機能などの詳細について考える必要がないことを意味します。

これにより、通常はそうでないアドホックシェルスクリプト（そうかもしれませんが）の方法で、弾力性のあるべき等コマンドを書くこと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

クラスター内のすべてのインスタンスで1つのコマンドを実行するコマンドの例を次に示します。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`）を作成したり、追加のタスクを実行して条件を評価したりして、何をすべき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
      become_user: root
      become: yes
```

12.5.4 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 *builtin tests* を補強できます。

どんなに単純なタスクであっても、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機能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された反復可能な方法を使用すると、危機に対処している場合でも、日常的なクラスター管理を行っている場合でも、重要なこと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）でクラスターインスタンスをターゲットとするテストを作成すると、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確認できます。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、または手動で確認しているときに間違いを犯すように誘う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。

テストでは、クラスターに重要な変更を加えてはなりません。実稼働サーバーで行うことを考えない場合は、おそらくテストに使用しないでください。

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` に `--include-tests-from /other/path` オプションを使用して、他の場所でテストを作成し、クラスター間でそれらを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

すべての Postgres インスタンスで実行されるテストの書き方は次のとおりです (`hosts: all` ではなく `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 (`assert`、`fail` … `when` など) を書くだけです。

```
- --

- name: Perform my custom tests
  hosts: role_postgres
  tasks:

    # Always start with this
    - include_role:
        name: test
        tasks_from: prereqs.yml

    # 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:
```

(次のページに続く)

(前のページからの続き)

```

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の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、単なる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。基本的な TPA セットアップタスクの後にインポートされ、実行されます。

破壊テスト

テストは、デフォルトでは、クラスターに重要な変更を加えるべきではありません。（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受け入れられない変更をクラスターに行うテストは、**destructive** としてマークし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` を含めるときに **destructive: yes** を設定することにより、「破壊的な」テストを定義できます。

```

- hosts: ...
  tasks:
    - include_role:
        name: test
        tasks_from: prereqs.yml
      vars:
        destructive: yes

```

その後、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表示されます。

（注： `--destroy-this-cluster` を使用することは、コマンドを実行するリスクを認識していることを意味します。テストによって実際にクラスターが破壊されることを保証するものではありません。）

12.5.5 Shared instances

12.5.6 Locale

一部のプラットフォームイメージと環境では、地域と言語の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てください。

```
localectl list-locales
```

または、Debian または Ubuntu の `/etc/locales.defs` ファイルの内容。

12.5.7 Patroni cluster management commands

Patroni は、次のコマンドオプションを使用して、M1 アーキテクチャーのシングルマスターフェイ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
```

RedHat にデプロイする場合は、PGDG リポジトリを `config.yml` の `yum` リポジトリリストに追加する必要があります。

```
cluster_vars:
  yum_repository_list:
    - PGDG
```

TPA `configure` は、3 つの `etcd` ノードと 2 つの `haproxy` ノードを追加します。 `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。

TPA は、既存の PostgreSQL スタンドアロンサーバーを変換する `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。単一の PostgreSQL サーバーとデータベースが作成されると、`Patroni` はレプリカを作成し、レプリケーションを構成します。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、Postgres は、TPA とクラスターの `config.yml` 設定を使用して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` インターフェー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。

これらの構成変数を使用して、TPA での Patroni の展開における特定の動作を制御できます。

Variable	Default value	Description
<i>patroni_super_user</i>	postgres	スーパーユーザーロールの postgres で作成するユーザー。
<i>patroni_replication_user</i>	replicator	レプリケーションロール用に postgres で作成するユーザー名。
<i>patroni_restapi_user</i>	patroni	パトロニ REST API 用に設定するユーザー名。
<i>patroni_rewind_user</i>	rewind	pg_rewind 関数のために postgres で作成するユーザー名。
<i>patroni_installation_method</i>	pkg	パッケージまたはソースからパトニーをインストールします (例: 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)。
<i>patroni_ssl_enabled</i>	no	REST API および cti 接続で SSL を有効にするかどうか。利用可能な場合、クラスター SSL 証明書と CA を使用します。
<i>patroni_rewind_enabled</i>	yes	postgres リワインドを有効にするかどうか、atoi_rewind_user で定義されたユーザーを作成し、config セクションを追加します。
<i>patroni_watchdog_enabled</i>	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。
<i>patroni_dcs</i>	etcd	DCS に使用するバックエンド。現時点で唯一のオプションは etcd です。
<i>patroni_listen_port</i>	8008	REST API TCP ポート番号
<i>patroni_conf_settings</i>	{}	パトロニ構成のオーバーライドを持つ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。
<i>patroni_dynamic_conf_settings</i>	{}	DCS 設定専用のオプションの構造化データ。これは `patroni_conf_settings` にマージされます。
<i>patroni_repl_max_lag</i>	None	これは、haproxy_read_only_load_balancer_enabled が true の場合にのみ haproxy バックエンドヘルスチェックで使用されます。`/replica?lag` の可能な値については :ref: `tpaexec-configure` <Patroni cluster management commands> を参照してください

Patroni 構成ファイルの設定

パトロニの構成は、Patroni デーモンによって設定されたデフォルト、DCS からロードされた構成、最後にローカル構成の3つのレイヤーから構築されます。最後はいずれかの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します。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.md](#) 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。以下の注を参照してください。
- DCS 設定の Patroni 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、ここで実行できます。

YAML 構成ファイルに保存されているローカル構成：

- 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される構成。上記の表を参照してください。
- 最後に、ユーザー指定の設定、`patroni_conf_settings` オプションからオーバーロード。機能フラグで制御されない構成を変更または追加する場合は、これが最適な場所です。

構成は、IP アドレス、ポート番号、クラスターの役割などのクラスター情報から TPA によって生成された構成の上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動作に影響を与える可能性があるため、オーバーライドに注意してください。

Patroni はすべての postgres 構成を DCS に保存し、これを postgres に配布する方法と時期を制御するため、TPA の一部の機能はパトロニと互換性がありません。

- `postgres_conf_template` 設定で `postgresql.conf` を生成するために使用される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 Postgres 構成ファイルの場所を変更することはできません。

TPA config.yml での Patroni 構成

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 docs の例）：

```
cluster_vars:
  patroni_conf_settings:
    restapi:
```

(次のページに続く)

(前のページからの続き)

```

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否定する場合は、値を `null` に設定できます。これにより、構成ファイルをロードするときにパトニーは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 run に従って TPA クラスターの現在のステータスを表示するには

```
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（メンテナンスの実行など）、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、これが当てはまることを確認するための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、クラスターが正常であることを確認することをお勧めします。

```

tpaexec deploy cluster_name
tpaexec test cluster_name

```

TPA は、たとえば別のノードがリーダーである場合など、config.yml に含まれる内容に関係なく、デプロイ中にノードの現在のロールを検出します。

12.5.8 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`

利用可能な展開プラットフォームを表示します

- `tpaexec info architectures`

利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 23.18.18

tpaexec=./tpaexec
TPA_DIR=/opt/EDB/TPA
PYTHON=/opt/EDB/TPA/tpa-venv/bin/python3 (v3.7.3, venv)
TPA_VENV=/opt/EDB/TPA/tpa-venv
ANSIBLE=/opt/EDB/TPA/tpa-venv/bin/ansible (v2.9.27)
Validation: e05e5302cd357b8ddb042b7591bf66dfa283213ccbe5073b2cff3c783be1310 [OK]
```

12.5.9 tpaexec reconfigure

`tpaexec reconfigure` コマンドは、`config.yml` を読み取り、クラスターをあるアーキテクチャから別のアーキテクチャに変更する改訂バージョンを生成します。次に

`tpaexec upgrade`

を呼び出して、クラスターを構成するインスタンスに必要な変更を加えることができます。

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。

次の引数は、新しい `config.yml` の内容を制御します。

- `--architecture <architecture>` (必須) クラスターの新しいアーキテクチャ。現在サポートされているアーキテクチャは `PGD-Always-ON` のみです
- `--pgd-proxy-routing <global|local>` (必須) `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、*[the PGD-Always-ON documentation](#)* 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のリスト。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。

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、変更内容の説明を表示します。
- `--check` 行われる変更を検証し、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
```