# Azure Cosmos DB for PostgreSQLの概要

Azure Cosmos DB for PostgreSQLは、クラウドで可用性の高いPostgreSQLデータベースを実行、管理、および拡張するために使用するマネージドサービスです。この手順では、Azureポータルを使用してAzure Cosmos DB for PostgreSQLのクラスターを作成する方法について説明します。分散データ（ノード間でのシャーディングテーブル、サンプルデータの読み込み、複数のノードで実行されるクエリーの実行）について説明します。

# Azure Cosmos DB for PostgreSQLの概要

Azure Cosmos DB for PostgreSQLは、スケールアウトするために構築された安心なPostgreSQLです。複数のマシンのクラスターに、データ、およびクエリーを配布（シャード）します。Azure Cosmos DB for PostgreSQLは、（フォークではなく）拡張機能として、新しいPostgreSQLリリースをサポートしており、既存のPostgreSQLツールとの互換性を維持しながら、ユーザーが新機能の恩恵を受けられるようにします。既存のワークロードにおけるパフォーマンスとスケーラビリティの2つの主要な問題を解決します。通常のPostgreSQLと同様に、Azure Cosmos DB for PostgreSQLもデータベースを管理します。高可用性、バックアップ、監視、アラート、その他の追加機能を提供し、これらはPaaS（サービスとしてのプラットフォーム）の一部として提供されます。

# Azure Cosmos DB for PostgreSQLのアーキテクチャ:

クラスターには、コーディネーターと呼ばれる1つの特殊ノードが必ずあります（他のノードはワーカーと呼ばれます）。アプリケーションは、コーディネーターにクエリーを送信し、コーディネーターは関連付けられたワーカーにクエリーを中継し、結果を蓄積します。

各クエリーについて、**コーディネーター**は、それを単一のワーカーにルーティングするか、必要なデータが単一あるいは複数のワーカーに存在するかに応じて、クエリーをいくつかに並列化します。Azure Cosmos DB for PostgreSQLが複数のワーカーにクエリーを分散する方法に関するシナリオを次に示します。

スクリーンショット が含まれている画像

自動的に生成された説明

# Azure Cosmos DB for PostgreSQLの概要

まず、提供された資格情報を使用してAzureポータルにログインする必要があります。

# Azureポータルへのサイン・イン

□1. Azure Portalに既にログインしている場合は、次のページにスキップします。このウィンドウの右下にある**Next**をクリックします。

□2. ブラウザで <https://portal.azure.com> を開き、ブラウザのウインドウを最大化します。

□3. [アカウントを選択する]というダイアログが表示されたら、[別のアカウントを使用する] を選択します。

□4. [サインイン]ダイアログの、[メール、電話、Skype]フィールドに  
userxxxxxx@cloudplatimmersionlabs.onmicrosoft.com を入力し[次へ]をクリックします。

□5. **[パスワード]**フィールドに**xxxxxxxx**を入力します。

□6. **[サインイン]**をクリックします。

□7. **[サインインの状態を維持しますか?]**とタイトルがついた、**[いいえ]**と**[はい]**ボタンがあるポップアップが表示されるかもしれません。**[いいえ]**を選択します。

□8. Welcome to Microsoft AzureとタイトルがついたStart TourとMaybe Laterボタンがあるポップアップが表示されるかもしれません。Maybe Laterを選択します。

# Azure Cosmos DB for PostgreSQLを使い始める

これらの手順では、Azureポータルを使用してAzure Cosmos DB for PostgreSQLクラスターを作成する方法について説明します。クラスターの規模によって作成に必要な時間は変わりますが、このハンズオンで作成するクラスターの規模であれば、通常約10分かかります。以下の手順では、プロセスを通じてこれがいかにシンプルで簡単であるかが分かります。

# Azure Cosmos DB for PostgreSQLを作成する

次の手順に従って、Azureポータルを使用してAzure Cosmos DB for PostgreSQLクラスターを作成するプロセスを理解します。

□1. Azureポータルの左上にある**[+リソースの作成]**をクリックします。

□2. Azure Marketplaceの**[リソースの作成]**ページで**[データベース]**を選択し、**[人気のAzureサービス]**ページで**[Azure Cosmos DB]**を選択します。[Azure Database for PostgreSQL]からでも作成は可能です。

□3. **[Azure Cosmos DBアカウントの作成]**ページで、**[Azure Cosmos DB for PostgreSQL]**の下の**[作成]**ボタンをクリックします。

□4. 以下の情報を**[Azure Cosmos DB for PostgreSQLクラスターの作成]**の基本タブに入力します。

* サブスクリプション: あなたのセッションのサブスクリプションがデフォルトになっています。
* リソースグループ: **[新規作成]をクリックし、**任意のリソースグループ名（例: citushandsonlab\_rg）を入力します。
* クラスター名: 任意のクラスター名（例: citushandsonlab）を入力します。
* 場所: **[(Asia Pacific) Japan East]**あるいは任意のAzureリージョンを選択します。
* スケール: **[構成]**をクリックし、ハンズオンのトレーナーの指示に従ってください。通常は**[ノードカウント]**から**[2 nodes]**を選択し、コーディネーターとワーカーの設定はそのままにして**[保存]**をクリックします。
* 管理者ユーザー名: 現時点では**citus**という値がデフォルトで変更することは出来ません。
* パスワード: **xxxxxxxx**を入力し、**[パスワードの確認]**にも同じパスワードを入力します。

|  |
| --- |
| 注: Azure Cosmos DB for PostgreSQLデプロイを作成する場合、最大20のワーカーノードを水平方向にスケーリングできます。20以上のノードが必要な場合は、サポートリクエストを作成するだけで、有効になります。コーディネーターと同様に、すべてのワーカー（コア、ストレージ）をスケールアップできますが、ストレージはスケールダウンが出来ません。RAMは、コア数とサーバーの種類（コーディネーターまたはワーカー）で決まります。 |

□5. [ネットワーク]タブで、[Allow public access from Azure services and resources within Azure to this cluster]を有効にします。また、[+ Add 0.0.0.0 – 255.255.255.255]をクリックして、ファイアウォールのルールを追加します。これは後ほど利用するクラウドシェルがAzure Cosmos DB for PostgreSQLクラスターに接続出来るようにするためです。

□6. **[確認および作成]**をクリックするとサマリーが表示されます。サマリーを確認後、**[作成]**ボタンをクリックするとデプロイが開始されます。こ

|  |
| --- |
| 注: [作成]ボタンをクリックするとデプロイが開始され、デプロイメントを監視するページにリダイレクトされます。 |

□7. Azureポータルの左上にある**[ホーム]**をクリックします。

□8. **[Azureサービス]**の下にある**[Azure Database for PostgreSQL]**をクリックします。

□9. 作成時に入力したクラスター名（例: citushandsonlab）をクリックします。  
Azure Cosmos DB for PostgreSQLクラスターを管理するAzure Portalの概要タブが表示されます。この概要タブの右上には、クラスターへの接続に使用するコーディネーター名（例: c.citushandsonlab.postgres.database.azure.com）が表示されます。

□10. 左にある**[接続文字列]**をクリックし、接続文字列を表示して確認します。

□11. 左にある**[コンピューティングとストレージ]**をクリックし、クラスターを構成する各ノードのサイズを表示します。

# Azure Cosmos DB for PostgreSQLの使用を開始する

Azureポータルのクラウドシェルを使用してAzure Cosmos DB for PostgreSQLクラスターに接続するには、ストレージアカウントを作成する必要があります。ストレージアカウントを使用すると、クラウドシェルに関連付けられたファイルを保存できるため、スクリプトの実行、データファイルのダウンロード、Azureリソースの管理など、さまざまなAzureポータルアクティビティで使用できます。

# クラウドシェルを作成する

□1. ポータルのバナーでクラウドシェルのアイコンをクリックします。アイコン

自動的に生成された説明

□2. **[ストレージがマウントされていません]**の画面で、**[詳細設定の表示]**をクリックします。

□3. サブスクリプションを選択します。

□4. リージョンに**[東日本]**あるいは任意のリージョンを選択します。

□5. リソースグループは既存のAzure Cosmos DB for PostgreSQLのクラスターを作成したグループあるいは任意のグループを使うようにしてください。

□6. ストレージアカウントには、**[新規作成]**を選択し、任意のストレージアカウント名（例: citushandsonlab）をペーストします。

□7. ファイルシェアには、**[新規作成]**を選択し、任意の共有ファイル名（例: citushandsonlab）を入力してください。

□8. **[作成]**をクリックします。

|  |
| --- |
| 注: クラウドシェルを作成・開始するのに1分程度を要します。 |

|  |
| --- |
| 注: bashコンソールでペーストするには右クリック後にpasteを選択します。 |

# Azure Cosmos DB for PostgreSQLに接続する

Azure Cosmos DB for PostgreSQLを作成すると、**citus**という名前の既定のデータベースが作成されます。データベースサーバーに接続するには、接続文字列と管理者パスワードが必要です。最初の接続には最大2分かかる場合があります。

# Psqlでデータベースに接続する

□1. クラウドシェルの右上にある**最大化**ボックスをクリックして全画面にします。

□2. bashプロンプトで、psqlユーティリティを用いてAzure Cosmos DB for PostgreSQLに接続します。最初の接続には最大2分かかる場合があります。以下のコマンドをコピー＆ペーストして[enter]を押します。

|  |
| --- |
| psql "host=c.citushandsonlab.postgres.database.azure.com port=5432 dbname=citus user=citus password='xxxxxxxx' sslmode=require" |

# テーブルを作成しスケールアウトする

psqlを使用してAzure Cosmos DB for PostgreSQLコーディネーターノードに接続すると、いくつかの基本的なタスクを完了できます。

この経験では、主に分散テーブルとそれらに慣れることに焦点を当てます。これから作業するデータモデルは単純です：GitHubのユーザーデータとイベントデータ。イベントには、フォークの作成、組織に関連するgitコミットなどが含まれます。psql経由で接続したら、テーブルを作成してみましょう。

□3. psqlコンソールで以下をコピー＆ペーストしてテーブルを作成します。

|  |
| --- |
| CREATE TABLE github\_events(  event\_id bigint,  event\_type text,  event\_public boolean,  repo\_id bigint,  payload jsonb,  repo jsonb,  user\_id bigint,  org jsonb,  created\_at timestamp  );  CREATE TABLE github\_users(  user\_id bigint,  url text,  login text,  avatar\_url text,  gravatar\_id text,  display\_login text  ); |

github\_eventsテーブルのペイロードフィールドには、JSONBデータ型があります。JSONBはPostgreSQLのバイナリ形式のJSONデータ型です。JSONデータ型を使用すると、柔軟なスキーマを1つの列に簡単に格納できます。PostgreSQLは、この型にGINインデックス（Generalized Inverted Index、汎用転置インデックス）を作成し、その中のすべてのキーと値にインデックスを付けることができます。インデックスを使用すると、さまざまな条件でペイロードを高速かつ簡単に照会できます。データを読み込む前に、いくつかのインデックスを作成してみましょう。

□4. psqlコンソールで以下をコピー＆ペーストしてインデックスを作成します。

|  |
| --- |
| CREATE INDEX event\_type\_index ON github\_events (event\_type);  CREATE INDEX payload\_index ON github\_events USING GIN (payload jsonb\_path\_ops); |

次に、コーディネーターノード上のPostgreSQLテーブルを指定し、Azure Cosmos DB for PostgreSQLにワーカーノード全体でシャードするように伝えます。そのために、それをシャードするキーを指定する各テーブルに対してクエリーを実行します。この例では、user\_idのイベントテーブルとユーザーテーブルの両方をシャードします。

□5. psqlコンソールで以下をコピー＆ペーストします。

|  |
| --- |
| SELECT create\_distributed\_table('github\_events', 'user\_id');  SELECT create\_distributed\_table('github\_users', 'user\_id'); |

このコマンドはテーブルごとに、ワーカーノードにシャードを作成します。各シャードは、一連のユーザーを保持する単純なpostgresqlテーブルです（user\_idでシャード化したので）。また、コーディネーターノードにメタデータを作成して、分散テーブルのセットとワーカーノードのシャードの局所性を追跡します。user\_idで両方のテーブルをシャードしたので、テーブルは自動的にコロケーションされます。つまり、両方のテーブルの1つのuser\_idに関連するすべてのデータが同じワーカーノード上にあります。これは、コロケーションされたシャード全体で、ワーカーノード上での２つのテーブル間の結合をローカルに実行する場合に役立ちます。コロケーションの詳細は次の「マルチテナントアプリケーション」で説明します。

event\_id

…

user\_id

org

event\_id

…

user\_id

org

user\_id

url

login

avatar\_url

user\_id

url

login

avatar\_url

user\_id

url

login

avatar\_url

github\_users

Node 2

event\_id

…

user\_id

org

github\_events

event\_id

…

user\_id

org

event\_id

…

user\_id

org

user\_id

url

login

avatar\_url

user\_id

url

login

avatar\_url

user\_id

url

login

avatar\_url

github\_users

Node 1

event\_id

…

user\_id

org

github\_events

**5** h… a h…

**5** h… b h…

**5** h… c h…

3h… **5** h…

2 h… **5** h…

5 h… **5** h…

... WHERE user\_id = 5

|  |
| --- |
| 注: Azure Cosmos DB for PostgreSQLサーバー内には、3種類のテーブルがあります。   * **分散テーブル** – ワーカーノードを跨いで分散（スケールアウト）。一般的に大きなテーブルはパフォーマンスを改善するために分散したテーブルであるべきです。 * **参照テーブル** – 全てのノードに複製されます。分散テーブルとの結合を可能にします。典型的には国や製品カテゴリのような小さなテーブルに用いられます。 * **ローカルテーブル** – コーディネーターノードに置かれるテーブルで、シャーディングのメタデータの管理テーブルが典型例です。 |

データをロードする準備が整いました。以下のコマンドでbashを「シェル実行」し、ファイルをダウンロードします。

□6. psqlコンソールでデータファイルをダウンロードするために以下をコピー＆ペーストとします。

|  |
| --- |
| \! curl -O https://examples.citusdata.com/users.csv  \! curl -O https://examples.citusdata.com/events.csv |

□7. psqlコンソールでデータファイルをロードするために以下をコピー＆ペーストします。

|  |
| --- |
| \copy github\_events from 'events.csv' WITH CSV  \copy github\_users from 'users.csv' WITH CSV |

重い本番ワークロードの場合、COPYコマンドが単一ノードのPostgreSQLよりもAzure Cosmos DB for PostgreSQLで高速な理由は、COPYがファンアウトされ複数のワーカーノードで並行して実行されることによります。

# クエリーの実行

ここから、実際にいくつかのクエリーを実行するので、楽しい時間です。簡単なcount(\*)から始めて、読み込んだデータの量を確認します。

□8. psqlコンソールでgithub\_eventsテーブルのレコードカウントを取得するために以下をコピー＆ペーストします。

|  |
| --- |
| SELECT count(\*) from github\_events; |

この単純なクエリーは、先ほど作成されたシャードキー**user\_id**に基づいてコーディネーターがすべてのワーカーに対して伝播（プロパゲート）し、コーディネーターが集計したレコード数が返されました。

JSONBペイロード列には、多くのデータがありますが、イベントの種類によって異なります。PushEventイベントには、プッシュの個別のコミットの数を含むサイズが含まれています。これを使用して、1時間あたりのコミットの合計数を検索できます。

□9. psqlコンソールで時間あたりのコミット数を見るために以下をコピー＆ペーストします。

|  |
| --- |
| SELECT date\_trunc('hour', created\_at) AS hour,  sum((payload->>'distinct\_size')::int) AS num\_commits  FROM github\_events  WHERE event\_type = 'PushEvent'  GROUP BY hour  ORDER BY hour; |

|  |
| --- |
| 注: 結果ビューでスタックした場合は、[q]と入力し、[Enter]を押してビューモードを終了します。 |

これまでのところ、クエリーにはgithub\_eventsテーブルだけが関係していましたが、この情報をgithub\_usersテーブルと組み合わせることができます。ユーザーとイベントの両方を同じ識別子（**user\_id**)でシャードしたので、一致するユーザーIDを持つ両方のテーブルの行は同じワーカーノードに配置され、簡単に結合できます。user\_idでクエリーを結合すると、Azure Cosmos DB for PostgreSQLコーディネーターは、ワーカーノードで並行して実行するために、結合の実行をシャード、つまりワーカーノード、に指示することになります。

□10. psqlコンソールでレポジトリ数が最大のユーザを見つけるために以下をコピー＆ペーストします。

|  |
| --- |
| SELECT login, count(\*)  FROM github\_events ge  JOIN github\_users gu  ON ge.user\_id = gu.user\_id  WHERE event\_type = 'CreateEvent' AND  payload @> '{"ref\_type": "repository"}'  GROUP BY login  ORDER BY count(\*) DESC  LIMIT 20; |

本番ワークロードでは、次の理由により、上記のクエリーはAzure Cosmos DB for PostgreSQL上で高速に実行されます。

* シャードが小さく、インデックスも小さい。これはリソースの利用効率の向上とインデックス/キャッシュのヒット率の向上に寄与します。
* 複数のワーカーノードによる並列実行

# 結論

このラボではデータベースクラスターを展開しシャーディングキーを設定することで、Microsoft Azure上でPostgreSQLを水平にスケールする方法を学びました。

他にも特にここで学んだこととして以下がありました。

* Azure Cosmos DB for PostgreSQLを展開する方法
* Azureクラウドシェルの作り方
* psqlを用いたAzure Cosmos DB for PostgreSQLへの接続方法
* スキーマの作成方法、シャーディングキーの設定方法、サーバーグループへのデータのロード方法

Azure Cosmos DB for PostgreSQLを使用すると、PostgreSQLデータベースクラスターにデータとクエリーを分散できるため、サーバーグループ内のすべてのノードで、すべてのメモリ、コンピューティング、およびディスクが利用できるというパフォーマンス上の利点をアプリケーションに提供できます。

自分のサブスクリプションでAzure Cosmos DB for PostgreSQLを試してみたい場合は、以下のリンクを参照してください。

* Azure Portal で Azure Cosmos DB for PostgreSQLクラスターを作成する  
  (<https://learn.microsoft.com/ja-jp/azure/cosmos-db/postgresql/quickstart-create-portal>)

また、学習用にDockerコンテナを手元のPCで実行することも可能です。

|  |
| --- |
| docker run --name citus\_standalone -p 5432:5432 citusdata/citus |

Azure Cosmos DB for PostgreSQL全般についてのドキュメントは以下から参照可能です。

* Azure Cosmos DB for PostgreSQLのドキュメント (<https://learn.microsoft.com/ja-jp/azure/cosmos-db/postgresql/introduction>)

# マルチテナントアプリケーション

この経験では、Azure Cosmos DB for PostgreSQL上でマルチテナントアプリケーションを作成するプロセスを説明します。

サービスとしてのソフトウェア (SaaS) アプリケーションを構築する場合は、データモデルにテナントの概念が既に組み込まれている可能性があります。通常、ほとんどの情報はテナント/顧客/アカウントに関連し、データベーステーブルはこの自然な関係をキャプチャします。

SaaSアプリケーションの場合、各テナントデータは１つのデータベースインスタンスにまとめて格納され、他のテナントから分離され、非表示に保つことができます。これは３つの点で効率的です。まず、アプリケーションの改善はすべてのクライアントに適用されます。次に、テナント間でデータベースを共有する場合、ハードウェアを効率的に使用します。最後に、すべてのテナントに対して、テナントごとに異なるデータベース サーバーよりも、すべてのテナントに対して１つのデータベースを管理する方がはるかに簡単です。

従来、単一のリレーショナルデータベースインスタンスでは、大規模なマルチテナントアプリケーションに必要なデータの量にスケーリングするのが困難でした。開発者は、データが単一のデータベースノードの容量を超えた場合に、リレーショナルモデルの利点を放棄することを余儀なくされました。

Azure Cosmos DB for PostgreSQLを使用すると、データベースが実際には水平方向にスケーラブルなマシンクラスタであるのに、ユーザーは単一のPostgreSQLデータベースに接続しているかのようにマルチテナントアプリケーションを作成できます。クライアントコードは最小限の変更のみを必要とし、完全なSQL機能を引き続き使用できます。

このガイドでは、マルチテナントアプリケーションのサンプルを取り上げ、Azure Cosmos DB for PostgreSQLを使用してスケーラビリティを考慮してモデル化する方法について説明します。その過程で、ノイズの多い隣のテナントからテナントを分離する、より多くのデータに対応するハードウェアのスケーリング、テナント間で異なるデータの格納など、マルチテナントアプリケーションの一般的な課題について検討します。Azure Cosmos DB for PostgreSQLは、これらの課題を処理するために必要なすべてのツールを提供します。では作成してみましょう。

# 広告業向けのマルチテナントアプリケーションを作成する

顧客がオンライン広告のパフォーマンスを追跡できるようにするマルチテナントSaaSアプリケーションのバックエンドの例を構築します。ユーザーはある瞬間に自らの会社（自分の会社）に関連するデータをリクエストするので、マルチテナントアプリケーションが向いているのは当然のことです。

このマルチテナントSaaSアプリケーションの簡略化されたスキーマを検討することから始めましょう。アプリケーションは、広告キャンペーンを実行する複数の企業を追跡する必要があります。キャンペーンには多数の広告があり、各広告にはクリック数とインプレッションの記録が関連付けられています。

以下はスキーマの例です。

□1. psqlコンソールに以下のCREATE TABLEコマンドをコピー＆ペーストして会社（テナント）とそのキャンペーンのテーブルを作成します。

|  |
| --- |
| CREATE TABLE companies (  id bigserial PRIMARY KEY,  name text NOT NULL,  image\_url text,  created\_at timestamp without time zone NOT NULL,  updated\_at timestamp without time zone NOT NULL  );  CREATE TABLE campaigns (  id bigserial,  company\_id bigint REFERENCES companies (id),  name text NOT NULL,  cost\_model text NOT NULL,  state text NOT NULL,  monthly\_budget bigint,  blacklisted\_site\_urls text[],  created\_at timestamp without time zone NOT NULL,  updated\_at timestamp without time zone NOT NULL,  PRIMARY KEY (company\_id, id)  ); |

□2. psqlコンソールに以下のCREATE TABLEコマンドをコピー＆ペーストして会社の広告のテーブルを作成します。

|  |
| --- |
| CREATE TABLE ads (  id bigserial,  company\_id bigint,  campaign\_id bigint,  name text NOT NULL,  image\_url text,  target\_url text,  impressions\_count bigint DEFAULT 0,  clicks\_count bigint DEFAULT 0, created\_at timestamp without time zone NOT NULL,  updated\_at timestamp without time zone NOT NULL,  PRIMARY KEY (company\_id, id),  FOREIGN KEY (company\_id, campaign\_id)  REFERENCES campaigns (company\_id, id)  ); |

□3. psqlコンソールに以下のCREATE TABLEコマンドをコピー＆ペーストして各広告のクリック数とインプレッション数の状態を追跡します。

|  |
| --- |
| CREATE TABLE clicks (  id bigserial,  company\_id bigint,  ad\_id bigint,  clicked\_at timestamp without time zone NOT NULL,  site\_url text NOT NULL,  cost\_per\_click\_usd numeric(20,10),  user\_ip inet NOT NULL,  user\_data jsonb NOT NULL,  PRIMARY KEY (company\_id, id),  FOREIGN KEY (company\_id, ad\_id)  REFERENCES ads (company\_id, id)  );  CREATE TABLE impressions (  id bigserial,  company\_id bigint,  ad\_id bigint,  seen\_at timestamp without time zone NOT NULL,  site\_url text NOT NULL,  cost\_per\_impression\_usd numeric(20,10),  user\_ip inet NOT NULL,  user\_data jsonb NOT NULL,  PRIMARY KEY (company\_id, id),  FOREIGN KEY (company\_id, ad\_id)  REFERENCES ads (company\_id, id)  ); |

□4. psqlコンソールに以下のCREATE TABLEコマンドをコピー＆ペーストして今作成したテーブルを確認します。

|  |
| --- |
| \dt |

マルチテナントアプリケーションはテナントごとにのみ一意性を適用できるため、すべての主キーと外部キーに会社 ID が含まれます。この要件により、分散環境ではこれらの制約を強制的に適用する必要が生じます。

# リレーショナルデータモデルをスケールする

リレーショナル データ モデルは、アプリケーションに最適です。データの整合性を保護し、柔軟なクエリーを可能にし、変化するデータに対応します。従来の唯一の問題は、リレーショナル データベースが、大規模な SaaS アプリケーションに必要なワークロードにスケーリングできないと考えられていたことです。開発者は NoSQL データベースを受け入れるか、そのサイズに到達するまでバックエンド サービスの塊を受け入れるしかありませんでした。

Azure Cosmos DB for PostgreSQLを使用すると、データモデルを維持し、スケールすることができます。Azure Cosmos DB for PostgreSQLはアプリケーションに単一の PostgreSQL データベースとして表示されますが、内部的には、要求を並列処理できる調整可能な数の物理サーバー (ノード) にクエリーをルーティングします。

マルチテナント アプリケーションには、通常、クエリーはテナントの組み合わせではなく、一度に 1 つのテナントに対して情報を要求するという、優れたプロパティを持っています。たとえば、営業担当者が CRM で見込顧客情報を検索する場合、検索結果は営業担当者の雇用主、つまり所属企業に固有であって、その他の企業の見込み案件情報およびメモは含まれません。

アプリケーション クエリーは店舗や会社などの単一のテナントに制限されているため、マルチテナント アプリケーション クエリーを高速化する方法の 1 つは、特定のテナントのすべてのデータを同じノードに格納することです。これにより、ノード間のネットワークオーバーヘッドが最小限に抑えられ、Azure Cosmos DB for PostgreSQLがすべてのアプリケーションの結合、キー制約、およびトランザクションを効率的にサポートできるようになります。これにより、アプリケーションを完全に書き直したり再設計したりしなくても、複数のノードにまたがってスケーリングできます。

電子機器 が含まれている画像

自動的に生成された説明

Azure Cosmos DB for PostgreSQLでは、テナントに関連するすべてのテーブルに、どのテナントがどの行を所有しているかを明確にマークする列があることを確認します。広告運用分析アプリケーションでは、テナントは会社なので、会社に関連するすべてのテーブルに company\_id 列が含まれるようにする必要があります。これらのテーブルは分散テーブルと呼ばれています。たとえば:キャンペーンは企業向けであるため、キャンペーンテーブルには company\_id 列が必要です。広告、クリック、インプレッションについても同じです。

したがって、この例では company\_id は “distribution column” (シャーディング キーまたは分散キーとも呼ばれます) になります。つまり、company\_id を使用して、ワーカー ノード間のすべてのテーブルをシャード/分散します。これにより、すべてのテーブルが結び付きます。つまり、すべてのテーブルの 1 つの会社に関連するすべてのデータが同じワーカー ノード上にあります。このようにして、Azure Cosmos DB for PostgreSQLに対して、同じ会社の行がマークされている場合に、この列を使用して同じノードに行を読み取りおよび書き込むように伝えることができます。たとえば、上記の図では、company\_id 5 のすべてのテーブルのすべての行が同じワーカー ノード上にあります。

この時点で、SQL をダウンロード・実行してスキーマを作成することにより、あなた専用のAzure Cosmos DB for PostgreSQLクラスターを目で追ってみましょう。スキーマの準備ができたら、Azure Cosmos DB for PostgreSQLにワーカーノードにシャードを作成するように伝えることができます。

# ノード間にテーブルをシャードする

create\_distributed\_table 関数は、テーブルをノード間で分散する必要があり、それらのテーブルへの将来の着信クエリーを分散実行用に計画する必要があることを Azure Cosmos DB for PostgreSQLに通知します。この関数は、ワーカー ノード上のテーブルのシャードも作成します。

この具体的な例では、company\_id に基づいてデータベース全体でテーブルをシャーディングして作成したアプリケーションをスケーリングします。

□1. psqlコンソールに以下をコピー＆ペーストして分散キー（シャード）を作成します。

|  |
| --- |
| SELECT create\_distributed\_table('companies', 'id');  SELECT create\_distributed\_table('campaigns', 'company\_id');  SELECT create\_distributed\_table('ads', 'company\_id');  SELECT create\_distributed\_table('clicks', 'company\_id');  SELECT create\_distributed\_table('impressions', 'company\_id'); |

□2. psqlコンソールに以下をコピー＆ペーストしてシャードの状態を確認します。

|  |
| --- |
| SELECT \* from pg\_dist\_shard; |

# データを取得する

前のセクションでは、マルチテナント アプリケーションの正しい分散列を特定しました：company id。単一コンピューター データベースでも、行レベルのセキュリティや追加のインデックス作成のいずれであっても、company id を追加してテーブルを非正規化すると便利です。他の利点は、追加の列を含めると、マルチマシンのスケーリングにも役立ちます。

次の手順では、サンプル データをコマンド ラインからクラスターに読み込みます。

# シェルでデータをダウンロードし取得する

□1. psqlコンソールに以下をコピー＆ペーストしてサンプルデータをダウンロードします。

|  |
| --- |
| \! curl -O https://examples.citusdata.com/mt\_ref\_arch/companies.csv  \! curl -O https://examples.citusdata.com/mt\_ref\_arch/campaigns.csv  \! curl -O https://examples.citusdata.com/mt\_ref\_arch/ads.csv  \! curl -O https://examples.citusdata.com/mt\_ref\_arch/clicks.csv  \! curl -O https://examples.citusdata.com/mt\_ref\_arch/impressions.csv  \! curl -O https://examples.citusdata.com/mt\_ref\_arch/geo\_ips.csv |

PostgreSQLの拡張機能であるcitusはCOPYコマンドを使用した一括読み込みをサポートします。ダウンロードしたデータを取り出し、ファイルを他の場所にダウンロードした場合は、正しいファイル パスを指定してください。

□2. psqlコンソールに以下をコピー＆ペーストしてテーブルをロードします。

|  |
| --- |
| \copy companies from 'companies.csv' with csv  \copy campaigns from 'campaigns.csv' with csv  \copy ads from 'ads.csv' with csv  \copy clicks from 'clicks.csv' with csv  \copy impressions from 'impressions.csv' with csv |

# テナントデータをクエリーする

company\_id のフィルターを含むアプリケーション クエリーまたは更新ステートメントは、引き続きそのまま動作します。前述のように、この種のフィルタはマルチテナント アプリで一般的です。オブジェクト リレーショナル マッパー (ORM) を使用する場合、これらのクエリーは、場所やフィルタなどの方法によって認識できます。

基本的に、データベースで実行される結果の SQL に、すべてのテーブル (JOIN クエリー内のテーブルを含む) に **WHERE company\_id = value** 句が含まれている場合、Azure Cosmos DB for PostgreSQLはクエリーを単一のノードにルーティングし、その状態で実行する必要があることを認識します。これにより、すべての SQL 機能を使用できます。結局のところ、各ノードは通常のPostgreSQLサーバです。

# company\_id = 5の単一のテナントのクエリー

アプリケーションが単一のテナントのデータを要求すると、データベースは単一のワーカー ノードでクエリーを実行できます。シングル テナント クエリーは、単一のテナント ID でフィルター処理します。たとえば、次のクエリーは、広告とインプレッションに対して company\_id = 5 をフィルター処理します。

□1. psqlコンソールに以下をコピー＆ペーストして単一のテナントのクエリーと更新を実行します。

|  |
| --- |
| -- campaigns with highest budget  SELECT name, cost\_model, state, monthly\_budget  FROM campaigns  WHERE company\_id = 5  ORDER BY monthly\_budget DESC  LIMIT 10;  -- double the budgets  UPDATE campaigns  SET monthly\_budget = monthly\_budget\*2  WHERE company\_id = 5; |

NoSQL データベースを使用してアプリケーションをスケーリングするユーザーが直面する一般的な問題は、トランザクションと結合機能が欠如していることです。ですが、Azure Cosmos DB for PostgreSQLではトランザクションは期待どおりに機能します。

□2. psqlコンソールに以下をコピー＆ペーストしてトランザクション（更新クエリー）を実行します。

|  |
| --- |
| -- transactionally reallocate campaign budget money  BEGIN;  UPDATE campaigns  SET monthly\_budget = monthly\_budget + 1000  WHERE company\_id = 5  AND id = 40;  UPDATE campaigns  SET monthly\_budget = monthly\_budget - 1000  WHERE company\_id = 5  AND id = 41;  COMMIT; |

SQL サポートの最後のデモとして集計関数とウィンドウ関数を含むクエリーがあり、これは単一ノードのPostgreSQL の場合と同じように Azure Cosmos DB for PostgreSQLで動作します。クエリーは、各キャンペーンの広告をインプレッション数でランク付けします。

□3. psqlコンソールに以下をコピー＆ペーストしてデータベースをまたぐ集計クエリーを実行します。

|  |
| --- |
| SELECT a.campaign\_id,  Rank()  OVER (  partition BY a.campaign\_id  ORDER BY a.campaign\_id, Count(\*) DESC ),  Count(\*) AS n\_impressions, a.id  FROM ads AS a  JOIN impressions AS i  ON i.company\_id = a.company\_id  AND i.ad\_id = a.id  WHERE a.company\_id = 5  GROUP BY a.campaign\_id, a.id  ORDER BY a.campaign\_id, n\_impressions DESC  Limit 20; |

|  |
| --- |
| 注: 結果ビューでスタックした場合は、「q」と入力し、「Enter」を押してビューモードを終了します。 |

要するに、クエリーがテナントにスコープを設定すると、挿入、更新、削除、複雑な SQL、およびトランザクションがすべて期待どおりに動作します。

# 参照テーブルでテナント間でデータを共有する

これまで、すべてのテーブルは company\_id によって分散されていましたが、すべてのテナントで共有できるデータがあり、特にどのテナントにも “所属” が存在しない場合があります。たとえば、この例の広告プラットフォームを使用するすべての企業は、IP アドレスに基づいてオーディエンスの地理情報を取得する必要があるとします。単一ノードのデータベースでは、次のような geo-ip のルックアップ テーブルによってこれを実現できます。(現実的なテーブルはおそらく PostGIS を使用しますが、簡略化された例に準拠しています)。

共有の地理情報を保持するテーブルを作成します。

□1. psqlコンソールに以下をコピー＆ペーストしてgeo\_ipsテーブルを作成します。

|  |
| --- |
| CREATE TABLE geo\_ips (  addrs cidr NOT NULL PRIMARY KEY,  latlon point NOT NULL  CHECK (-90 <= latlon[0] AND latlon[0] <= 90 AND  -180 <= latlon[1] AND latlon[1] <= 180)  );  CREATE INDEX ON geo\_ips USING gist (addrs inet\_ops); |

分散セットアップでこのテーブルを効率的に使用するには、geo\_ips テーブルをクリック数、それも 1 つだけではなく、すべての会社の、と共に再配置する方法を見つける必要があります。この方法では、クエリー時にネットワーク トラフィックは発生しません。これをAzure Cosmos DB for PostgreSQLで行うには、geo\_ips をすべてのワーカー ノードにテーブルのコピーを格納する参照テーブルとして指定します。

□2. psqlコンソールに以下をコピー＆ペーストしてgeo\_ipsテーブルを作成します。

|  |
| --- |
| -- Make synchronized copies of geo\_ips on all workers  SELECT create\_reference\_table('geo\_ips'); |

参照テーブルはすべてのワーカー ノードにレプリケートされ、Azure Cosmos DB for PostgreSQLは変更時に自動的に同期を維持します。create\_distributed\_table ではなく、create\_reference\_table と呼ばれることに注意してください。

□3. psqlコンソールに以下をコピー＆ペーストしてgeo\_ipsにデータをロードします。

|  |
| --- |
| \copy geo\_ips from 'geo\_ips.csv' with csv |

これで、このテーブルでクリック数を結合すると、すべてのノードで効率的に実行されます。広告 290 をクリックしたすべてのユーザーの場所を尋ねることができます。

□4. psqlコンソールに以下をコピー＆ペーストしてSELECTを実行します。

|  |
| --- |
| SELECT c.id, clicked\_at, latlon  FROM geo\_ips, clicks c  WHERE addrs >> c.user\_ip  AND c.company\_id = 5  AND c.ad\_id = 290; |

# スキーマのオンライン変更

マルチテナント システムのもう 1 つの課題は、すべてのテナントのスキーマの同期を維持することです。スキーマの変更は、すべてのテナントに一貫して反映する必要があります。Azure Cosmos DB for PostgreSQLでは、標準の PostgreSQL DDL (データ定義言語) コマンドを使用してテーブルのスキーマを変更するだけで、Azure Cosmos DB for PostgreSQLは 2 フェーズ コミット (2PC) プロトコルを使用してコーディネータ ノードからワーカーに伝播します。

たとえば、このアプリケーションの提供情報は、キャプション テキスト列を使用できます。コーディネータで標準 SQL を発行することで、テーブルに列を追加できます。

□1. psqlコンソールに以下をコピー＆ペーストして新しい列を追加します。

|  |
| --- |
| ALTER TABLE ads ADD COLUMN caption text; |

これにより、すべてのワーカーノードも更新されます。このコマンドが完了すると、Azure Cosmos DB for PostgreSQLクラスターは、新しいキャプション列のデータの読み取りまたは書き込みを行うクエリーを受け入れます。

# テナント間でデータが異なる時

各テナントは、他のテナントが必要としない一意の情報を格納する必要がある場合があります。たとえば、広告データベースを使用するテナント アプリケーションの 1 つでクリック トラッキング情報を保存する場合がありますが、別のテナントがブラウザ エージェントを必要とする場合があります。ただし、すべてのテナントは、同じデータベース スキーマを持つ共通のインフラストラクチャを共有します。

従来、マルチテナントに共有スキーマアプローチを使用するデータベースは、事前に割り当てられた “custom“ 列の固定数を作成するか、外部の “extension” テーブルを持つことに頼っていました。この問題を解決する方法は非構造化データ、特に JSONB を利用することです。JSONB (B はバイナリの意) は、NoSQL データベースの利点の一部を提供し、よりクエリーを高速化するためインデックスを作成することもできます。

この例では、テナント (company\_id = 5 など) は、JSONB 列を使用して、ユーザーがモバイル デバイス上にあるかどうかを追跡します。

□1. psqlコンソールに以下をコピー＆ペーストしてモバイルデバイスである会社 5のユーザを検索します。

|  |
| --- |
| SELECT  user\_data->>'is\_mobile' AS is\_mobile,  count(\*) AS count  FROM clicks  WHERE company\_id = 5  GROUP BY user\_data->>'is\_mobile'  ORDER BY count DESC; |

□2. psqlコンソールに以下をコピー＆ペーストして**部分インデックス**を作成することによりクエリーを最適化します。

|  |
| --- |
| CREATE INDEX click\_user\_data\_is\_mobile  ON clicks ((user\_data->>'is\_mobile'))  WHERE company\_id = 5; |

PostgreSQLはJSONB列の**GIN**インデックスをサポートしています。JSONB 列に GIN インデックスを作成すると、その JSON ドキュメント内のすべてのキーと値にインデックスが作成されます。これにより、?, ?|、?&などのいくつかのJSONB 演算子が高速化されます。

□3. psqlコンソールに以下をコピー＆ペーストして**GINインデックス**を作成します。

|  |
| --- |
| CREATE INDEX click\_user\_data  ON clicks USING gin (user\_data);  -- this speeds up queries like, "which clicks have  -- the is\_mobile key present in user\_data?"  SELECT id  FROM clicks  WHERE user\_data ? 'is\_mobile'  AND company\_id = 5  Limit 5; |

# 結論

これで、どのようにAzure Cosmos DB for PostgreSQLで、マルチテナントアプリケーションがスケールするようにする方法が分かりました。

このチュートリアルでは、Azure Cosmos DB for PostgreSQLを使い以下の方法を学びました。

* Azure Cosmos DB for PostgreSQLクラスターの作成
* psqlユーティリティを使ったスキーマの作成
* ノード間でのテーブルのシャード
* サンプルデータの取得
* テナントデータのクエリー
* テナント間でのデータの共有
* テナント毎のスキーマのカスタマイズ

追加のドキュメントにも興味があるかもしれません。

* Migration Guides  
  **Ruby on Rails** (<https://docs.citusdata.com/en/v11.0/develop/migration_mt_ror.html>) **Django** (<https://docs.citusdata.com/en/v11.0/develop/migration_mt_django.html>)  
  **ASP.NET** (<https://docs.citusdata.com/en/v11.0/develop/migration_mt_asp.html>)  
  **Java** (<https://www.citusdata.com/blog/2018/02/13/using-hibernate-and-spring-to-build-multitenant-java-apps/>)
* マルチテナントフィルタを全てのクエリーに自動的に追加するライブラリ  
  Rails用の**activerecord-multi-tenant (****<https://github.com/citusdata/activerecord-multi-tenant>)** ライブラリ  
  Django用の**django-multitenant (**[**https://github.com/citusdata/django-multitenant**](https://github.com/citusdata/django-multitenant)**)** ライブラリ

# リアルタイムダッシュボード

Azure Cosmos DB for PostgreSQLは、複数のワーカー/リソース間でクエリーを並列化して、パフォーマンスを大きく向上させることができます。並列処理の機能の恩恵を受けることができるワークロードの1つは、イベントデータのリアルタイムダッシュボードの機能です。

たとえば、顧客企業がHTTPトラフィックを監視できるようにするクラウドサービスプロバイダーになることができます。顧客の1社がHTTP要求を受け取るたびに、サービスはログレコードを受け取ります。これらのレコードをすべて取り入れ、HTTP運用分析ダッシュボードを作成すると、サイトが提供したHTTPエラーの数などの洞察が顧客に提供されます。顧客がサイトの問題を解決できるように、このデータができるだけ少ない待機時間で表示されることが重要です。また、ダッシュボードに過去の傾向のグラフを表示することも重要です。

また、広告ネットワークを構築し、キャンペーンのクリック率を顧客に表示することもできます。この例では、待機時間も重要であり、生データ量も多く、履歴データとライブデータの両方が重要です。

この経験では、Azure Cosmos DB for PostgreSQLを使用して、リアルタイムおよびスケーリングの問題に対処する方法について説明します。

# データモデル

私たちがこれから扱うデータは、Azure Cosmos DB for PostgreSQLに直接挿入されるログデータで事後に変更されることのないストリームです。また、ログデータを最初にKafkaのようなサービスにルーティングすることも一般的です。Kafkaには、大量のデータを管理できるように、データを事前に集計できるなど、多くの利点があります。

このページでは、HTTPイベントデータを取り込み、シャードし、読み込みとクエリーを作成するための単純なスキーマを作成します。

# アプリケーションのテーブルを作成する

http\_requests のテーブル、分単位の集計、および最後のロールアップの位置を維持するテーブルを作成してみましょう。

□1. psqlコンソールに以下のCREATE TABLEコマンドをコピー＆ペーストしてテーブルを作成します。

|  |
| --- |
| -- this is run on the coordinator  CREATE TABLE http\_request (  site\_id INT,  ingest\_time TIMESTAMPTZ DEFAULT now(),  url TEXT,  request\_country TEXT,  ip\_address TEXT,  status\_code INT,  response\_time\_msec INT  );  CREATE TABLE http\_request\_1min (  site\_id INT,  ingest\_time TIMESTAMPTZ, -- which minute this row represents  error\_count INT,  success\_count INT,  request\_count INT,  average\_response\_time\_msec INT,  CHECK (request\_count = error\_count + success\_count),  CHECK (ingest\_time = date\_trunc('minute', ingest\_time))  );  CREATE INDEX http\_request\_1min\_idx ON http\_request\_1min (site\_id, ingest\_time);  CREATE TABLE latest\_rollup (  minute timestamptz PRIMARY KEY,  CHECK (minute = date\_trunc('minute', minute))  ); |

□2. psqlコンソールに以下をコピー＆ペーストして作成したものを確認します。

|  |
| --- |
| \dt |

# ノード間にテーブルをシャードする

Azure Cosmos DB for PostgreSQLをデプロイすると、ユーザーが指定した列の値に基づいて、異なるノードにテーブルの行が格納されます。この「分散列」は、ノード間でデータをシャードする方法を示します。分散列をsite\_id、つまりシャードキーに設定してみましょう。

□3. psqlコンソールに以下をコピー＆ペーストしてテーブルをシャードします。

|  |
| --- |
| SELECT create\_distributed\_table('http\_request', 'site\_id');  SELECT create\_distributed\_table('http\_request\_1min', 'site\_id'); |

上記のコマンドは、ワーカーノード間の2つのテーブルのシャードを作成します。シャードは、一連のサイトを保持するPostgreSQLテーブルにすぎません。テーブルの特定のサイトのすべてのデータは、同じシャードに保持されます。

両方のテーブルがsite\_idでシャードされていることに注意してください。したがって、http\_requestシャードとhttp\_request\_1minシャード、つまり同じサイトのセットを保持する両方のテーブルのシャードが同じワーカーノード上にある、1対1の対応があります。これは**コロケーション**と呼ばれています。コロケーションを使用すると、結合などのクエリーがより高速になり、ロールアップが可能になります。次の図では、両方のテーブルのsite\_id 1と3がWorker1にあり、site\_id 2と4 がWorker2 にあるコロケーションの例が表示されます。

時計 が含まれている画像

自動的に生成された説明

|  |
| --- |
| 注: create\_distributed\_table UDF (ユーザー定義関数) は、シャードカウントのデフォルト値を使用します。デフォルトは32です。HTTPトラフィックの監視と同様のリアルタイム分析のユースケースでは、クラスター内のCPUコアと同じ数のシャードを使用することをお勧めします。これにより、新しいワーカーノードを追加した後、クラスター全体でデータのバランスを取り直すことができます。シャードカウントは、citus.shard\_countを使用して設定できます。これは、create\_distributed\_tableコマンドを実行する前に構成する必要があります。 |

# データの生成

システムはデータを受け入れ、クエリーを提供する準備が整いました。次の一連の命令では、この資料の他のコマンドを続行しながら、バックグラウンドでpsqlコンソールで次のループを実行し続けます。それは1秒または2秒ごとに偽のデータを生成します。

□4. psqlコンソールに以下をコピー＆ペーストしてbashコンソールから抜けます。

|  |
| --- |
| \q |

□5. クラウドシェルのバナー上の編集アイコンをクリックします。

壁, 室内 が含まれている画像

自動的に生成された説明

□6. クラウドシェルのエディターに、以下をコピー＆ペーストして（エディターにペーストするには、**[control] + [v]**を使います）、http\_requestの負荷を生成させます。

|  |
| --- |
| -- loop continuously writing records every 1/4 second  DO $$  BEGIN LOOP  INSERT INTO http\_request (  site\_id, ingest\_time, url, request\_country,  ip\_address, status\_code, response\_time\_msec  ) VALUES (  trunc(random()\*32), clock\_timestamp(),  concat('http://example.com/', md5(random()::text)),  ('{China,India,USA,Indonesia}'::text[])[ceil(random()\*4)],  concat(  trunc(random()\*250 + 2), '.',  trunc(random()\*250 + 2), '.',  trunc(random()\*250 + 2), '.',  trunc(random()\*250 + 2)  )::inet,  ('{200,404}'::int[])[ceil(random()\*2)],  5+trunc(random()\*150)  );  COMMIT;  PERFORM pg\_sleep(random() \* 0.25);  END LOOP;  END $$; |

□7. クラウドシェルのエディターの右上にある、省略記号のアイコンをクリックし、**Close Editor**を選びます。

□8. “Do you want to save”ダイアログで、**Save**をクリックします。

□9. ファイル名として以下を入力し、**Save**をクリックします。

|  |
| --- |
| load.sql |

□10. クラウドシェルのbashコンソールに以下をコピー＆ペーストして、バックグラウンドでload.sqlを実行するために[Enter]を押します。load.sqlに続く”&”はバックグラウンド実行に必要です。

|  |
| --- |
| psql "host=c.citushandsonlab.postgres.database.azure.com port=5432 dbname=citus user=citus password='spxxxxxxxx' sslmode=require" -f load.sql & |

# ダッシュボードのクエリー

Azure Cosmos DB for PostgreSQLホスティングオプションを使用すると、複数のノードがクエリーを並列処理して高速化できます。たとえば、データベースはワーカーノードのSUMやCOUNTなどの集計を計算し、結果を最終的な回答に結合します。

□11. クラウドシェルのbashコンソールに以下をコピー＆ペーストして、再度Psqlを実行するために[Enter]を押します。

|  |
| --- |
| psql "host=c.citushandsonlab.postgres.database.azure.com port=5432 dbname=citus user=citus password='spxxxxxxxx' sslmode=require" |

□12. クラウドシェルのPsqlコンソールに以下のコマンドを入力し、リアルタイムの負荷が生成されているかを検証します。

|  |
| --- |
| SELECT Count(\*) FROM http\_request; |

□13. クラウドシェルのPsqlコンソールに以下のコマンドを複数回入力し、カウントが増加していることを確認します。

|  |
| --- |
| SELECT Count(\*) FROM http\_request; |

このクエリーを実行して、1 分あたりの Web 要求といくつかの統計情報をカウントします。

□14. Psqlコンソールに以下をコピー＆ペーストし、サイトに対する平均応答時間を確認します。

|  |
| --- |
| SELECT  site\_id,  date\_trunc('minute', ingest\_time) as minute,  COUNT(1) AS request\_count,  SUM(CASE WHEN (status\_code between 200 and 299) THEN 1 ELSE 0 END) as success\_count,  SUM(CASE WHEN (status\_code between 200 and 299) THEN 0 ELSE 1 END) as error\_count,  SUM(response\_time\_msec) / COUNT(1) AS average\_response\_time\_msec FROM http\_request  WHERE date\_trunc('minute', ingest\_time) > now() - '5 minutes'::interval  GROUP BY site\_id, minute  ORDER BY minute ASC  LIMIT 15; |

|  |
| --- |
| 注: 結果ビューでスタックした場合は、「q」と入力し、「Enter」を押してビューモードを終了します。 |

上記のセットアップは機能しますが、欠点があります。

* HTTP 運用分析ダッシュボードは、グラフを生成する必要があるたびに全ての行を確認する必要があります。たとえば、クライアントが過去1年間の傾向に関心を持っている場合、クエリーは過去1年間の全ての行を最初から集計します。
* ストレージコストは、読み込み速度とクエリー可能な履歴の長さに比例して増加します。実際には、生のイベントを短い期間 (1か月) だけ保持し、より長い期間 (年) については履歴グラフだけを見たいはずです。

# ロールアップ

データが増加しても、パフォーマンスを維持することを考えましょう。生データを集計テーブルに定期的にロールアップすることで、ダッシュボードの高速化を保証します。集計期間を試すことができます。この例では、1**分あたりの集計**テーブルを使用しますが、代わりにデータを5分、15分、または60分に分割できます。

このロールアップをより簡単に実行するには、plpgsql関数に配置します。

**http\_request\_1min**を設定するには、SELECTに挿入を定期的に実行します。これは、テーブルがコロケーションされているために可能となります。次の関数は、便宜上ロールアップクエリーをラップします。

□1. Psqlコンソールに以下をコピー＆ペーストし、rollup\_http\_request関数を作成します。

|  |
| --- |
| -- initialize to a time long ago  INSERT INTO latest\_rollup VALUES ('10-10-1901');  -- function to do the rollup  CREATE OR REPLACE FUNCTION rollup\_http\_request() RETURNS void AS $$  DECLARE  curr\_rollup\_time timestamptz := date\_trunc('minute', now());  last\_rollup\_time timestamptz := minute from latest\_rollup;  BEGIN  INSERT INTO http\_request\_1min (  site\_id, ingest\_time, request\_count,  success\_count, error\_count, average\_response\_time\_msec  ) SELECT  site\_id,  date\_trunc('minute', ingest\_time),  COUNT(1) as request\_count,  SUM(CASE WHEN (status\_code between 200 and 299) THEN 1 ELSE 0 END) as success\_count,  SUM(CASE WHEN (status\_code between 200 and 299) THEN 0 ELSE 1 END) as error\_count,  SUM(response\_time\_msec) / COUNT(1) AS average\_response\_time\_msec  FROM http\_request  -- roll up only data new since last\_rollup\_time  WHERE date\_trunc('minute', ingest\_time) <@  tstzrange(last\_rollup\_time, curr\_rollup\_time, '(]')  GROUP BY 1, 2;  -- update the value in latest\_rollup so that next time we run the  -- rollup it will operate on data newer than curr\_rollup\_time  UPDATE latest\_rollup  SET minute = curr\_rollup\_time;  END;  $$ LANGUAGE plpgsql; |

□2. Psqlコンソールに以下をコピー＆ペーストし、ロールアップ関数を実行します。

|  |
| --- |
| SELECT rollup\_http\_request(); |

|  |
| --- |
| 注: 上記の関数は毎分呼び出す必要があります。これを行うには、**pg\_cron**という PostgreSQL 拡張機能を使用して、データベースから直接定期的なクエリーをスケジュールできます。たとえば、上記のロールアップ関数は、以下のコマンドで毎分呼び出すことができます。 |

|  |
| --- |
| SELECT cron.schedule('\* \* \* \* \*','SELECT rollup\_http\_request();'); |

以前のダッシュボードクエリーよりもずっと良くなっています。１分間の集計ロールアップテーブルを照会して、以前と同じレポートを取得できます。

□3. Psqlコンソールに以下をコピー＆ペーストし、１分毎の集計テーブルでのクエリーを実行します。

|  |
| --- |
| SELECT site\_id, ingest\_time as minute, request\_count,  success\_count, error\_count, average\_response\_time\_msec  FROM http\_request\_1min  WHERE ingest\_time > date\_trunc('minute', now()) - '5 minutes'::interval  LIMIT 15; |

# 古いデータを期限切れにする

ロールアップによってクエリーが高速になりますが、ストレージコストが無限に増大することを回避するために古いデータを期限切れにする必要があります。粒度ごとにデータを保持する期間を決定し、標準クエリーを使用して期限切れのデータを削除します。次の例では、生データを１日、１分単位の集計を１か月間、保持することにしました。期限切れになる古いデータがないため、これらのコマンドを今すぐ実行する必要はありません。

|  |
| --- |
| DELETE FROM http\_request WHERE ingest\_time < now() - interval '1 day';  DELETE FROM http\_request\_1min WHERE ingest\_time < now() - interval '1 month'; |

本番環境では、これらのクエリーを関数にラップし、cronジョブで毎分呼び出すことができます。

データの有効期限は、Azure Cosmos DB for PostgreSQLによるシャーディングに加えて、PostgreSQLの最新の時間パーティショニング機能を使用することで、さらに高速に実行できます。また、Azure Cosmos DB for PostgreSQLが提供するcreate\_time\_partition関数を使用して、時間パーティションの作成と保守を自動化することもできます。

また、Citus 10ではカラムナーストレージが追加されました。この機能によって古くなってアクセス頻度が落ちたパーティションを列方向に圧縮することでストレージ領域を節約することが可能です。

これらは基本に過ぎません！ HTTPイベントを取り込み、これらのイベントを事前に集約された形式にロールアップするアーキテクチャを提供しました。これにより、生のイベントを保存し、１秒未満のクエリーを使用して分析ダッシュボードをより強力にすることもできます。

次のセクションでは、基本的なアーキテクチャについて説明し、よく出てくる質問を解決する方法を示します。

# おおよその個別の数

HTTP運用分析におけるよくあるクエリーは、おおよその個別の数を扱います: 先月のサイトを訪問したユニーク訪問者数はいくつですか。この質問に正確に答えるには、以前にサイトを訪れたすべての訪問者のリストをロールアップテーブルに格納する必要があります。しかし、おおよその答えははるかに管理しやすくなります。

ハイパーログログ (HLL) と呼ばれるデータ型は、クエリーにほぼ答えることができます。セット内のユニークな要素の数を知るには、驚くほど少ないスペースで十分です。その正確さは調節することができます。最大2.2%のエラーがあるものの、1280バイトのみで、何百億という単位のユニーク訪問者数を数えることができるもの、を使用します。

先月に顧客企業のサイトを訪問したユニークなIPアドレスの数など、グローバルクエリーを実行する場合は、同様の問題が発生します。HLLがない場合、ワーカーからコーディネータに送信されるクエリー結果には、コーディネータが重複除外しないとならないIPアドレスの一覧が含まれます。これは、多くのネットワークトラフィックと計算の両方が必要になってしまいます。HLLを使用すると、クエリーの速度を大幅に向上できます。

**Citus** を自分でインストールする場合は、まずHLL拡張機能をインストールして有効にする必要があります。Psqlコマンド**CREATE EXTENSION hll**を実行します。この場合、すべてのノードで実行する必要があります。Azure Cosmos DB for PostgreSQLには、他の便利な拡張機能と共にHLLが既にインストールされているので、この作業はAzureでは必要となりません。

これで、HLLを使用したロールアップでIPアドレスを追跡する準備ができました。最初にロールアップ テーブルに列を追加します。

□1. Psqlコンソールに以下をコピー＆ペーストし、http\_request\_1minテーブルを変更します。

|  |
| --- |
| ALTER TABLE http\_request\_1min ADD COLUMN distinct\_ip\_addresses hll; |

次に、カスタム集計を使用して列を設定します。

□2. Psqlコンソールに以下をコピー＆ペーストし、ロールアップ関数のクエリーに追加します。

|  |
| --- |
| -- function to do the rollup  CREATE OR REPLACE FUNCTION rollup\_http\_request() RETURNS void AS $$  DECLARE  curr\_rollup\_time timestamptz := date\_trunc('minute', now());  last\_rollup\_time timestamptz := minute from latest\_rollup;  BEGIN  INSERT INTO http\_request\_1min (  site\_id, ingest\_time, request\_count,  success\_count, error\_count, average\_response\_time\_msec,  distinct\_ip\_addresses  ) SELECT  site\_id,  date\_trunc('minute', ingest\_time),  COUNT(1) as request\_count,  SUM(CASE WHEN (status\_code between 200 and 299) THEN 1 ELSE 0 END) as success\_count,  SUM(CASE WHEN (status\_code between 200 and 299) THEN 0 ELSE 1 END) as error\_count,  SUM(response\_time\_msec) / COUNT(1) AS average\_response\_time\_msec,  hll\_add\_agg(hll\_hash\_text(ip\_address)) AS distinct\_ip\_addresses  FROM http\_request  -- roll up only data new since last\_rollup\_time  WHERE date\_trunc('minute', ingest\_time) <@  tstzrange(last\_rollup\_time, curr\_rollup\_time, '(]')  GROUP BY 1, 2;  -- update the value in latest\_rollup so that next time we run the  -- rollup it will operate on data newer than curr\_rollup\_time  UPDATE latest\_rollup  SET minute = curr\_rollup\_time;  END;  $$ LANGUAGE plpgsql; |

INSERT INTOステートメントに**distinct\_ip\_address**が追加され、  
SELECTには**hll\_add\_agg(hll\_hash\_text(ip\_address)) AS distinct\_ip\_address**がrollup\_http\_request関数に追加されました。

□3. Psqlコンソールに以下をコピー＆ペーストし、更新された関数を実行します。

|  |
| --- |
| SELECT rollup\_http\_request(); |

ダッシュボードクエリーはもう少し複雑です。hll\_cardinality関数を呼び出すことによって、異なるIPアドレスの数を読み取る必要があります。

□4. Psqlコンソールに以下をコピー＆ペーストし、hll\_cardinality関数を利用したレポートを生成します。

|  |
| --- |
| SELECT site\_id, ingest\_time as minute, request\_count,  success\_count, error\_count, average\_response\_time\_msec,  hll\_cardinality(distinct\_ip\_addresses)::bigint AS distinct\_ip\_address\_count  FROM http\_request\_1min  WHERE ingest\_time > date\_trunc('minute', now()) - interval '5 minutes'  LIMIT 15; |

HLLは単に高速なだけではなく、以前はできなかったことができます。ロールアップを実行したが、HLL を使用する代わりに、正確な一意のカウントを保存したとします。これは正常に動作しますが、「この1週間に、生データを破棄したセッションはいくつあったか」などのクエリーには答えられません。

HLLを使用すれば簡単です。次のクエリーを使用して、一定期間における個別のIP数を計算できます。

□5. Psqlコンソールに以下をコピー＆ペーストし、期間中の異なるIP数を計算します。

|  |
| --- |
| SELECT hll\_cardinality(hll\_union\_agg(distinct\_ip\_addresses))::bigint  FROM http\_request\_1min  WHERE ingest\_time > date\_trunc('minute', now()) - '5 minutes'::interval  LIMIT 15; |

# JSONBの非構造化データ

Azure Cosmos DB for PostgreSQLは、PostgreSQLに組み込みでサポートされている非構造化データ型とうまく機能します。これを実証するために、各国から来た訪問者数を追跡してみましょう。半構造化データ型を使用すると、個々の国ごとに列を追加する必要がなくなります。PostgreSQLには、JSONデータを格納するためのJSONBデータ型とJSONデータ型があります。データ型としてJSONBが推奨される理由は、a) JSONと比較してJSONBにはインデックス作成機能 (GINおよびGIST) があり、b) JSONBはバイナリ形式であるため圧縮機能が提供される、ためです。ここでは、JSONB列をデータモデルに組み込む方法を示します。

□1. Psqlコンソールに以下をコピー＆ペーストし、ロールアップのテーブルにJSONB列を新たに追加します。

|  |
| --- |
| ALTER TABLE http\_request\_1min ADD COLUMN country\_counters JSONB; |

□2. Psqlコンソールに以下をコピー＆ペーストし、rollup\_http\_requestをcountry\_countersで更新します。

|  |
| --- |
| -- function to do the rollup  CREATE OR REPLACE FUNCTION rollup\_http\_request() RETURNS void AS $$  DECLARE  curr\_rollup\_time timestamptz := date\_trunc('minute', now());  last\_rollup\_time timestamptz := minute from latest\_rollup;  BEGIN  INSERT INTO http\_request\_1min (  site\_id, ingest\_time, request\_count,  success\_count, error\_count, average\_response\_time\_msec,  distinct\_ip\_addresses,  country\_counters  ) SELECT  site\_id,  date\_trunc('minute', ingest\_time),  COUNT(1) as request\_count,  SUM(CASE WHEN (status\_code between 200 and 299) THEN 1 ELSE 0 END) as success\_count,  SUM(CASE WHEN (status\_code between 200 and 299) THEN 0 ELSE 1 END) as error\_count,  SUM(response\_time\_msec) / COUNT(1) AS average\_response\_time\_msec,  hll\_add\_agg(hll\_hash\_text(ip\_address)) AS distinct\_ip\_addresses,  jsonb\_object\_agg(request\_country, country\_count) AS country\_counters  FROM (  SELECT \*,  count(1) OVER (  PARTITION BY site\_id, date\_trunc('minute', ingest\_time), request\_country  ) AS country\_count  FROM http\_request  ) h  -- roll up only data new since last\_rollup\_time  WHERE date\_trunc('minute', ingest\_time) <@  tstzrange(last\_rollup\_time, curr\_rollup\_time, '(]')  GROUP BY 1, 2;  -- update the value in latest\_rollup so that next time we run the  -- rollup it will operate on data newer than curr\_rollup\_time  UPDATE latest\_rollup SET minute = curr\_rollup\_time;  END;  $$ LANGUAGE plpgsql; |

□3. Psqlコンソールに以下をコピー＆ペーストし、更新した関数を実行します。

|  |
| --- |
| SELECT rollup\_http\_request(); |

ダッシュボードでアメリカから送信されたリクエストの数を取得する場合は、ダッシュボードクエリーを次のように変更できます。

□4. Psqlコンソールに以下をコピー＆ペーストし、アメリカからのリクエストを確認します。

|  |
| --- |
| SELECT  request\_count, success\_count, error\_count, average\_response\_time\_msec, COALESCE(country\_counters->>'USA', '0')::int AS american\_visitors  FROM http\_request\_1min WHERE ingest\_time > date\_trunc('minute', now()) - '5 minutes'::interval  LIMIT 15; |

# 結論

このチュートリアルでは、Azure Cosmos DB for PostgreSQLを使い以下をどのように実行するかを学びました。

* バックグラウンドでリアルタイムの負荷を生成する
* psql 関数を作成して更新する
* アプリケーションがスケールできるようにデータをロールアップする
* 古いデータの有効期限を切る
* 個別のカウントに関するレポート
* 非構造化データ (JSONB) を使用するようにモデルを更新する

追加の参照情報

* Timeseries Data (<https://docs.citusdata.com/en/v11.0/use_cases/timeseries.html?highlight=partition#automating-partition-creation>)
* Archiving with Columnar Storage  
  (<https://docs.citusdata.com/en/v11.0/use_cases/timeseries.html?highlight=partition#archiving-with-columnar-storage>)
* GitHub - PostgreSQL Cron job (<https://github.com/citusdata/pg_cron>)
* GitHub - HLL HyperLogLog (<https://github.com/citusdata/postgresql-hll>)
* When to use unstructured datatypes in Postgres–Hstore vs. JSON vs. JSONB (<https://www.citusdata.com/blog/2016/07/14/choosing-nosql-hstore-json-jsonb/>)