

Azure Database for PostgreSQL のハイパースケール (Citius) の概要

Azure Database 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 Database for PostgreSQL のハイパースケール (Citius) のサーバーグループ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の実行）について説明します。

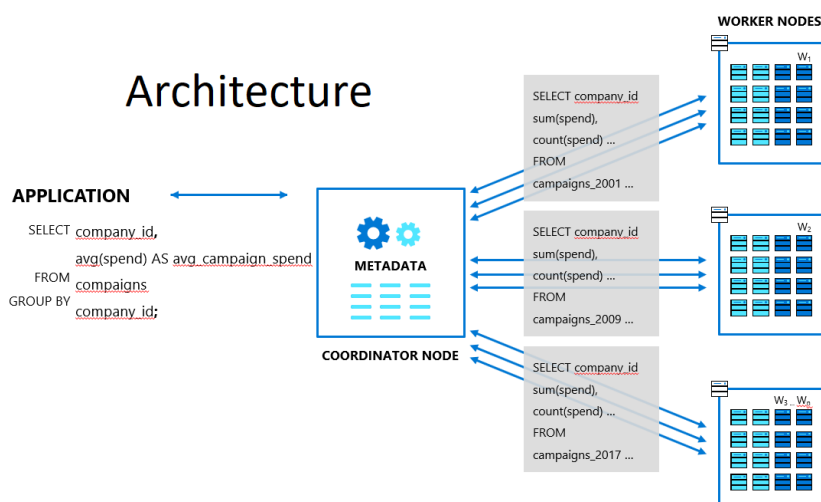
ハイパースケール (Citius) の概要

Azure Database for PostgreSQL のハイパースケール (Citius) は、スケールアウトするために構築された安心な Postgres です。複数のマシンのクラスターに、データ、およびクエリを配布（シャーディング）します。ハイパースケール (Citius) 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、ハイパースケール (Citius) 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。

ハイパースケール (Citius) 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を中継し、結果を蓄積します。

各クエリについて、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、クエリをいくつかに並列化します。ハイパースケール (Citius) が複数のワーカーにクエリを分散する方法に関するシナリオを次に示します。



1. このウインドウの右下にある **Next** をクリックします。

Azure Database for PostgreSQL のハイパースケール (Citrus) 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、ブラウザのウィンドウを最大化します。
- 3. Pick an account というダイアログが表示されたら、+ Use another account を選択します。
- 4. Sign in ダイアログの、Email, phone or Skype フィールドに userxxxxx@cloudplatimmersionlabs.onmicrosoft.com を入力し Next をクリックします。
- 5. Password フィールドに xxxxxxxx を入力します。
- 6. Sign in をクリックします。
- 7. Stay signed in? とタイトルがついた、No と Yes 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。No 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。
- 9. このウィンドウの右下にある **Next** をクリックします。

PostgreSQL のハイパースケール (Citius) 拡張を使い始める

これらの手順では、Azure ポータルを使用してハイパースケール (Citius) サーバークラスタを作成する方法について説明します。通常、これは約 10 分かかりますが、時間の節約のために事前に作成してあります。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が、最終的には事前に作成されたものを使用します。

Azure Database for PostgreSQL のハイパースケール (Citius) を作成する

次の手順に従って、Azure ポータルを使用して PostgreSQL ハイパースケール (Citius) サーバークラスタ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 **+ Create a resource** をクリックします。
- 2. Azure Marketplace の New ページで **Databases** を選択し、Databases ページで **Azure Database for PostgreSQL** を選択します。
- 3. デプロイメントオプションページで、Hyperscale (Citius) on Azure Database for PostgreSQL の下の **Create** ボタンをクリックします。
- 4. 以下の情報を新しいサーバーの詳細に入力します。
 - Subscription: 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。
 - Resource Group: **select existing...** ドロップダウンをクリックし、**rg000000** を選択します。
 - Server group name: **sg000000** を入力します。
 - Admin username: 現時点では **citius** という値が必須です。
 - Password: **xxxxxxx** を入力し、**Confirm Password** にも同じパスワードを入力します。
 - Location: **eastus2** を選択します。
 - Compute + Storage: **Configure server group** をクリックします。このセクションの設定はそのままにして **Save** をクリックします。

注: ハイパースケール (Citius) 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ルアップできます。20 以上のノードが必要な場合は、サポートチケッ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）をセットアップ/ダウンできます。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. Review + create をクリックするとサマリーが表示されます。ここでは**絶対に Create をクリックしない**でください。時間を節約するために事前に作成してあります。

注: Create をクリックすると、デプロイに最大 10 分かかります。待機中に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。

- 6. Azure ポータルの左上にある **Home** をクリックします。
- 7. Azure services の下にある **Azure Database for PostgreSQL servers** をクリックします。

□8. **sg000000** をクリックします。

これは、ハイパースケール (Citrus) サーバーグループを管理できる Azure Portal の概要ブレードです。この概要タブには、サーバーグループへの接続に使用するコーディネーター名が右上に表示されます。

□9. 左にある **Connection Strings** をクリックし、接続文字列の書式の数字を表示します。


□10. 左にある **Configure** をクリックし、デプロイメントの設定を表示します。

□11. このウィンドウの右下にある **Next** をクリックします。

ハイパースケール (Citrus) 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ハイパースケール (Citrus) サーバークラウドグループに接続するには、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。 
- 2. Welcome to Azure Cloud Shell で **Bash** をクリックします。
- 3. You have no storage mounted の画面で、Show advanced settings をクリックします。
- 4. サブスクリプションとリージョンのデフォルト値を使います。
- 5. リソースグループは既存の **rg000000** 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、Create new を選択し、**sg000000shell** をペーストします。
- 7. ファイルシェアには、Create new を選択し、**sg000000shell** を入力してください。
- 8. Create Storage 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。

□9. 次の手順でファイアウォールを構成するには、クラウドシェルのクライアント IP アドレスが必要です。コマンドプロンプトで次のコマンドを入力し、return キーを押してから、クラウドシェルの IP アドレスをコピーまたはメモします。

```
curl -s https://ifconfig.co
```

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。

- 11. このウィンドウの右下にある **Next** をクリックします。

ハイパースケール (Citrus) の利用を開始する

Azure Database for PostgreSQL のハイパースケール (Citrus) は、サーバーレベルでファイアウォールを使用します。既定では、ファイアウォールはすべての外部アプリケーションとツールがコーディネーターノードおよび内部のデータベースに接続するのを防ぎます。特定の IP アドレス範囲のファイアウォールを開くルールを追加する必要があります。

右上の Overview ペインには、接続先のクラスターのコーディネーターホスト名のアドレスが表示されます。

サーバーレベルのファイアウォールのルールを設定する

- 1. Security の下の Overview ペインの左のナビゲーションで **Firewall** をクリックします。
- 2. クラウドシェルで確認した **IP アドレス** を **START IP** と **END IP** に入力します。
- 3. **FIREWALL RULE NAME** に CloudShell と入力します。
- 4. ペインの左上にある **Save** をクリックします。

注: ハイパースケール(Citrus) サーバーはポート 5432 を介して通信します。企業ネットワーク内から接続しようとしている場合、ポート 5432 を超える送信トラフィックは、ネットワークのファイアウォールで許可されない場合があります。その場合は、IT 部門がポート 5432 を開かない限り、ハイパースケール (Citrus) サーバーに接続できません。

Azure Database for PostgreSQL のハイパースケール (Citus)に接続する

ハイパースケール(Citus)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。何らかの理由でシェルがタイムアウトして再起動した場合は、`curl -s https://ifconfig.co` コマンドをもう一度実行し、ファイアウォールが新しい IP アドレスで更新されていることを確認する必要があります。

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 Database 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=sg000000-c.postgres.database.azure.com port=5432 dbname=citus user=citus
password='xxxxxxx' sslmode=require"
```

テーブルを作成しスケールアウトする

Psql を使用してハイパースケール (Citus) 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 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 Postgres のバイナリ形式の JSON データ型です。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。Postgres 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 Postgres テーブルを指定し、ハイパースケール（Citus）にワーカー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');

```


テーブルごとに、このコマンドは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。

注: ハイパースケール (Citius) 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、管理テーブルがローカルテーブルの典型例です。

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 のクラウドシェルを「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＆ペーストとします。

```
¥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 Postgres よりもハイパースケール (Citius) で高速な理由は、COPY がファンアウトされワーカーノード間で並行して実行されることによります。

クエリの実行

ここから、実際にいくつかのクエリを実行するので、楽しい時間です。簡単なカウント(*)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 のレコードカウントを取得するために以下をコピー＆ペーストとします。

```
SELECT count(*) from github_events;
```

この単純なクエリは、先ほど作成されたシャードキー **user_id** に基づいてコントローラーがすべてのワーカーに対してリファクタリングし、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、クエリには github_events だけが関係していましたが、この情報を github_users 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データベースノードと同じ位置に配置され、簡単に結合できます。user_id でクエリを結合すると、ハイパースケール (Citius) コントローラーは、ワーカーノードで並行して実行するために結合実行をシャードに押し下げるでしょう。

□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、上記のクエリはハイパースケール (Citius) 上では高速です。

- 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

□11. このウインドウの右下にある **Next** をクリックします。

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 Postgres 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 Database for PostgreSQL にハイパースケール (Citus) を展開する方法
- Azure クラウドシェルの作り方
- Psql を用いたハイパースケール (Citus) 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 Database for PostgreSQL のハイパースケール (Citus) を使用すると、Postgres データベースクラスター (「サーバーグループ」と呼ばれる) にデータとクエリを分散できるため、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ハイパースケール (Citus) 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。

- Quickstart create Hyperscale (Citus) (<https://docs.microsoft.com/en-us/azure/postgresql/quickstart-create-hyperscale-portal>)

今日は時間をいただきありがとうございます。またこの経験を完了されおめでとうございます。

マルチテナントアプリケーション

この経験では、Azure Database for PostgreSQL のハイパースケール(Citus)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。

ただし、単一のリレーショナルデータベースインスタンスでは、従来、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。

ハイパースケール(Citus)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、ハイパースケール (Citus) 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 Database for PostgreSQL のハイパースケール (Citus)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。

1. このウィンドウの右下にある **Next** をクリックします。

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。

□5. このウィンドウの右下にある **Next** をクリックします。

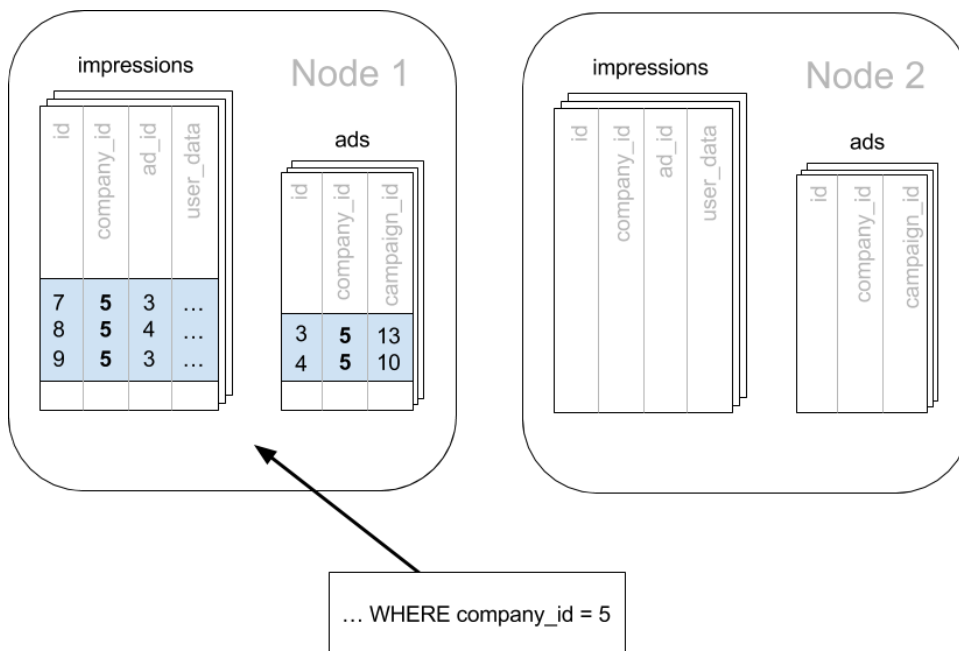
リレーショナルデータモデルをスケールする

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。

ハイパースケール(Citus)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。ハイパースケール (Citus) 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、検索結果は雇用主に固有であって、その他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、ハイパースケール (Citus) 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書き直しや再設計しなくても、複数のノードにまたがってスケーリングできます。



ハイパースケール (Citius) 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、ハイパースケール (Citius) 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、あなた専用のハイパースケール (Citius) クラスタを目で追ってみましょう。スキーマの準備ができれば、ハイパースケール (Citius) 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を分散実行用に計画する必要があることを Hyperscale (Citius) 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ルリングします。

□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. このウィンドウの右下にある **Next** をクリックします。

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 の拡張機能であるハイパースケール(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
```

□3. このウィンドウの右下にある **Next** をクリックします。

テナントデータをクエリする

`company_id` のフィルターを含むアプリケーション クエリ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 クエリ内のテーブルを含む) に **WHERE `company_id` = value** 句が含まれている場合、Hyperscale (Citius) はクエリ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

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 ノードでクエリを実行できます。シングル テナント クエリは、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. Psq| 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の一般的な問題点は、トランザクションと結合の欠如です。ただし、トランザクションはハイパースケール (Citius) で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である更新を実行します。

```
-- 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があり、PostgreSQL の場合と同じように Hyperscale (Citius) で動作します。クエリは、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。

□4. このウィンドウの右下にある **Next** をクリックします。

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ります。単一のマシン 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。(実際の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. PsqI 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、クエリ時にネットワーク トラフィックは発生しません。これをハイパースケール (Citius) 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. PsqI 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、ハイパースケール (Citius) 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. PsqI 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、このテーブルを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;
```

□5. このウィンドウの右下にある **Next** をクリックします。

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。ハイパースケール (Citus) 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、ハイパースケール (Citus) 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、ハイパースケール (Citus) クラスタは、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を受け入れます。

□2. このウィンドウの右下にある **Next** をクリックします。

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。非構造化列の種類、特に JSONB です。JSONB (B はバイナリ用) は、NoSQL データベースの利点の一部を提供し、より高速なクエリに対して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 (会社 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. PsqI 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. PsqI 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;
```

□4. このウィンドウの右下にある **Next** をクリックします。

結論

これで、どのように Azure Database for PostgreSQL のハイパースケール(Citus)で、マルチテナントアプリケーションがスケールする方法が分かりました。

このチュートリアルでは、Azure Database for PostgreSQL のハイパースケール(Citus)を使い以下の方法を学びました。

- ハイパースケール(Citus)サーバグループの作成
- 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/v8.1/develop/migration_mt_ror.html#rails-migration)
 - Django (https://docs.citusdata.com/en/v8.1/develop/migration_mt_django.html#django-migration)
- マルチテナントフィルタを全てのクエリ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>) ライブラリ

1. この経験がどのくらい素晴らしかったか **Feedback** をクリックして教えてください。

リアルタイムダッシュボード

Hyperscale (Citus) は、複数のワーカー/リソース間でクエリを並列化して、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、他の企業が 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 Database for PostgreSQL の Hyperscale (Citus) 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。

1. このウィンドウの右下にある **Next** をクリックします。

データモデル

私たちがこれから扱うデータは、ハイパースケール (Citrus) 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

ハイパースケールをデプロイすると、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 がワーカー1 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。



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

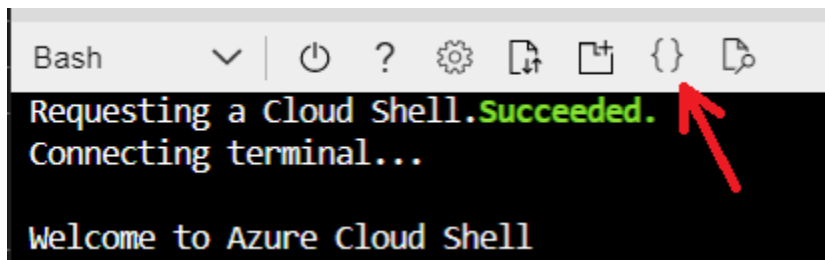
システムはデータを受け入れ、クエリを提供する準備が整いました。次の一連の命令では、この資料の他のコマンドを続行しながら、バックグラウンドで Psql コンソールで次のループを実行し続けます。それは 1 秒

または 2 秒ごとに偽のデータを生成します。

□4. クラウドシェルの 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。 {}



□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。

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

ダッシュボードのクエリー

ハイパースケール(Citus)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=sgxxxxxx-c.postgres.database.azure.com port=5432 dbname=citus user=citus  
password='spxxxxxxx' 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。

□15. このウィンドウの右下にある **Next** をクリックします。

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 $$
DECLARE
curr_rollup_time timestampz := date_trunc('minute', now());
last_rollup_time timestampz := minute from latest_rollup;
BEGIN
INSERT INTO http_request_1min (
    site_id, ingest_time, request_count,
    success_count, error_count, average_response_time_msec
) SELECT
site_id,
date_trunc('minute', ingest_time),
COUNT(1) as request_count, SUM(CASE WHEN (status_code between 200 and 299) THEN 1 ELSE 0 END)
as success_count, SUM(CASE WHEN (status_code between 200 and 299) THEN 0 ELSE 1 END) as
error_count, SUM(response_time_msec) / COUNT(1) AS average_response_time_msec
FROM http_request
-- roll up only data new since last_rollup_time
WHERE date_trunc('minute', ingest_time) <@
    tstzrange(last_rollup_time, curr_rollup_time, '[]')
GROUP BY 1, 2;
-- update the value in latest_rollup so that next time we run the
-- rollup it will operate on data newer than curr_rollup_time
UPDATE latest_rollup SET minute = curr_rollup_time;
END;
$$ 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;
```

□4. このウィンドウの右下にある **Next** をクリックします。

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、Hyperscale (Citius) 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、pg_partman などの拡張機能を使用して、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。

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。

□1. このウィンドウの右下にある **Next** をクリックします。

おおよその個別の数

HTTP 運用分析におけるよくある質問は、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。1280 バイトのみで、最大 2.2%のエラーがあるものの、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。

先月にクライアント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、このクエリには、ワーカーからコーディネータに重複除外する 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、クエリを大幅に向上できます。

ハイパースケール以外 (Citrus)を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。ハイパースケール (Citrus) 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 は単に高速だけでなく、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;
```

□6. このウィンドウの右下にある **Next** をクリックします。

JSONB の非構造化データ

ハイパースケール (Citius) は、Postgres 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. Psq1 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. Psq1 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;

```

□5. このウィンドウの右下にある **Next** をクリックします。

結論

このチュートリアルでは、Azure Database for PostgreSQL のハイパースケール(Citus)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

- Citus and pg_partman: Creating a scalable time series database on Postgres (<https://www.citusdata.com/blog/2018/01/24/citus-and-pg-partman-creating-a-scalable-time-series-database-on-PostgreSQL/>)
- 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/>)

この経験がどのくらい素晴らしかったか **Feedback** をクリックして教えてください。