

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。

ハイパースケール(Citus)を使ってリアルタイムアプリケーションを作成する

最初に提供されたクレデンシャルを用いて Azure ポータルにログインします。

Azure ポータルにサイン・インする

□1. 既に Azure ポータルにログインしている場合、次のページに進みます。このウイ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. 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
xxxx@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. このウイ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(Citus)の利用を始める

Azure Portal クラウド シェルを使用するには、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、スクリプトを実行して Azure リソースを管理するさまざまな Azure ポータルアクティビティで使用できます。

Cloud Shell を作成する

- 1. ポータルのバナーで Cloud Shell 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: Cloud Shell 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、Cloud Shell 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。

PostgreSQL 用のハイパースケール(Citus)拡張の利用を始める

Azure Database for PostgreSQL ハイパースケール (Citus) サービスは、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。

このラボでは、4 つの vCore と 16 GB の RAM を備えた 1 つのコーディネータと 2 つのワーカーを備えた基本的な本番グレードのハイパースケール (Citus) クラスタを事前にプロビジョニングしました。

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. Azure ポータルの左上にある **Home** をクリックします。
- ☐2. Azure サービスの下にある Azure Database for PostgreSQL servers をクリックします。
- ☐3. **sgxxxxxx** をクリックします。
- ☐4. 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。
- ☐5. Cloud Shell で確認した **IP アドレス**を **START IP** と **END IP** に入力します。
- ☐6. FIREWALL RULE NAME に CloudShell と入力します。
- ☐7. 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(Citus)サーバはポート 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 (Citus) サーバーに接続できません。

- ☐8. 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 (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. Cloud Shell 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='xxxxxxxx' sslmode=require"
```

□3. 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、

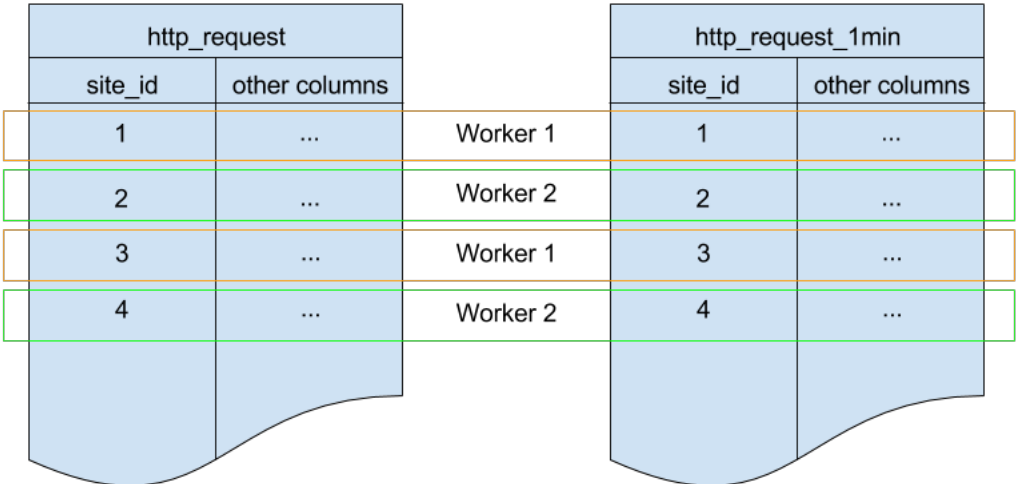
つまりシャードキーに設定してみましょう。

□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

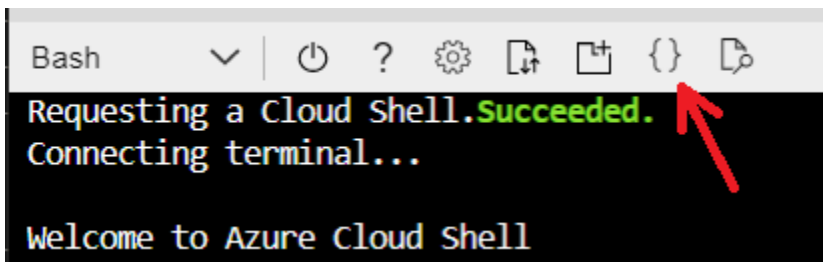
システムはデータを受け入れ、クエリを提供する準備が整いました。次の一連の命令では、この資料の他のコマンドを続行しながら、バックグラウンドで Psql コンソールで次のループを実行し続けます。それは 1

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

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。