Aurora MySQL 互換のハイパースケーリングでトラフィックの急増に対応

AWS 規範ガイダンス

AWS 規範ガイダンス: Aurora MySQL 互換のハイパースケーリングでトラフィックの急増に対応

Table of Contents

Aurora MySQL 互換のハイパースケーリングでトラフィックの急増に対応

Oliver Francis、Shyam Sunder Rakhecha、Vikram singh Rai (Amazon Web Services (AWS))

2022 年 10 月

インターネットでビジネスをしていると、既存のアプリケーションインフラストラクチャでは対処できないような、かつてない急激な成長に直面することがあります。こうしたことは、製品広告の内容が想定を越える人々の関心を引いた、顧客の購買傾向が対面からオンラインへと突然シフトしたなど、さまざまな要因で起こります。

こうした予期せぬ負荷に対処するには、インフラストラクチャの大規模な拡張が必要になります。アプリケーション層をスケールする場合は、サーバーあるいはポッドもさらに追加する必要があります。しかし、ハイパースケーリングを実現する上で最も難しいのはデータベースです。データベースインスタンスを入手可能な最も大きなインスタンスサイズにスケールすることはできるかもしれませんが、それでは問題が解決しない場合があります。

本ガイドでは、データベースワークロードを Amazon Aurora MySQL 互換エディションで実行しているユーザーの方を対象に、突然の急激な成長にも対応できるようデータベースをハイパースケールする際に、実行すべき推奨事項について解説します。これらの推奨事項は、すべてが長期的なベストプラクティスというわけではありません。急激な成長が予想され、それに対処するためのプランを事前に作成できる余裕がある場合は、リソース セクションにあるブログ記事を参照してください。こちらの記事は、長期的なベストプラクティスを実装する際に役立ちます。一方、急激な成長に予期せず直面した場合は、こちらのガイドをお読みいただくと、負荷に対処し、適度な安定性を実現できます。その間に、自社のビジネスに適した長期的なソリューションを計画して実行していきましょう。

本ガイドで概説している推奨事項は、開発チームによる実装支援が必要になります。

ターゲットを絞ったビジネス成果

本ガイドで取り上げている方法は、以下の作業に役立ちます。

- 予期せぬ急激な成長に備えてビジネスを安定させる。急激な成長を見すえた長期的なベストプラクティスを実装するための、十分な安定性を実現しましょう。

- 財務上の損失を回避する。ハイパースケール後の環境で突然中断が生じると、顧客がアプリケーションで実行するビジネストランザクションが、減少する可能性があります。場合によっては、これにより多額の経済的損失が発生します。ハイパースケール後の環境を安定させることは、ビジネスの損失につながる長期的な中断を回避するためにきわめて重要です。

接続の管理

アプリケーションの需要が高まると、フロントエンドのトラフィックが増加します。一般的なシナリオでは、こうした受信トラフィックの爆発的増加に対処する場合、アプリケーション層に自動スケーリングが設定されます。それにより、アプリケーション層は自動スケーリングを開始し、トラフィックの増加に対応するためにアプリケーションサーバー (インスタンス) がさらに追加されます。すべてのアプリケーションサーバーでは、データベース接続プールの設定が事前に完了しているため、データベースが受信する接続の数は、新たにデプロイされたインスタンスの数に比例して増加します。

例えば、データベース接続の件数が 200 件ずつに設定されているアプリケーションサーバーが 20 台ある場合、合計 4,000 件のデータベース接続がオープンすることになります。アプリケーションプールが 200 インスタンスまでスケールアップされると (ピーク時など)、接続の合計数は 40,000 件に達します。一般的なワークロードでは、こうした接続の大半はアイドル状態である可能性が高いです。ただし、接続が突然増加すると、Amazon Aurora MySQL 互換エディションのデータベースのスケーリング能力が制限されることがあります。これは、接続がアイドル状態であっても、メモリや、ファイル記述子などのその他のサーバーリソースが使用されるためです。Aurora MySQL 互換が使用するメモリの量は、通常、同じ数の接続を維持するため MySQL Community Edition よりも少なくなります。ただし、接続がアイドル状態のときのメモリ使用量は、やはりゼロではありません。

設定変数

2 つの主要なサーバー設定変数、`max_connections` と `max_connect_errors` を使用すると、データベースに許可される受信接続の数を制御することができます。

設定変数 max_connections

設定変数 `max_connections` は、各 MySQL インスタンスのデータベース接続の上限数を設定します。ベストプラクティスは、各データベースインスタンスでオープンすることが見込まれる接続の最大数より、若干高めに設定しておくことです。

併せて `performance_schema` も有効にする場合は、`max_connections` 設定に特に注意が必要です。Performance Schema のメモリ構造は、サーバー設定変数 (`max_connections` など) に基づいて自動的にサイズが設定されます。変数を高く設定すれば Performance Schema が使用するメモリの量も増えます。極端な場合、それによって小規模なインスタンスタイプでメモリ不足の問題が生じる可能性があります。Performance Insights を有効にすると Performance Schema が自動的に有効になることに注意してください。

設定変数 max_connect_errors

設定変数 max_connect_errors は、所定のクライアントホストからの、連続して中断した接続リクエストを、何件許可するかを決定します。クライアントホストが、失敗した接続試行の、指定された連続回数を超えると、サーバーはその接続を遮断します。そのクライアントからさらに接続しようとするとエラーが発生します。

```
Host 'host_name' is blocked because of many connection errors. Unblock with
'mysqladmin flush-hosts'
```

「ホストがブロックされている」エラーが発生した場合は、 max_connect_errors変数の値を増やさないでください。代わりに、aborted_connects ステータス変数と host_cach テーブルにあるサーバーの診断カウンターを調べます。そこで収集した情報を使って、接続の問題が発生しているクライアントを特定し、修正します。また、このパラメータは、skip_name_resolve が 1 (デフォルト) に設定されている場合は無効になるため注意しましょう。

以下に関する詳細は、MySQL のリファレンスマニュアルを参照してください。

- Max_connect_errors variable
- 「ホストがブロックされました」エラー
- Aborted_connects status variable
- Host_cache table

接続プールを実装する

スケーリングイベントによってアプリケーションサーバーが増えると、その結果、DB サーバーが、完全にロードされたアクティブな接続件数を超える可能性があります。アプリケーションサーバーとデータベースの間に接続プールまたはプロキシレイヤーを追加すると、ファネルのように機能して、データベース上の接続の合計数が減少することになります。プロキシの主な目的は、多重化を用いてデータベース接続を再利用することです。

プロキシは、一方では接続数が制限されているデータベースに接続します。もう一方では、アプリケーションの接続を受け入れます。また、クエリキャッシュ、接続バッファリング、クエリの書き換えとルーティング、ロードバランシングなどの追加機能も提供します。接続プール層は、データベースへの最大接続数が完全にロードされたときの数を下回るように設定する必要があります。Amazon RDS Proxy は、こうした目的のために実装できるフルマネージドのプロキシです。Amazon RDS Proxy は、ほとんどのアプリケーションに関してコード変更が不要で、ソリューションを実装するためにインフラストラクチャを追加で管理する必要もありません。

Aurora MySQL 互換で使用できる以下のサードパーティーのプロキシもご覧ください。

- ProxySQL
- MariaDB MaxScale
- ScaleARC
- Heimdall Data

コネクションストームを回避する

データベースのオーバーロードや、レプリカのプライマリノードからの大幅な遅れなどが発生した場合に、接続プールをどのように機能させるかを検討します。プロキシサーバーや接続プールを設定するときは、データベースの応答が遅いこと (基盤となるハードウェアかストレージの問題または DB リソースの制約が原因) を理由に、接続プール全体をリセットすることのないようにします。

突然、何百件もの接続が開始されると、データベースへの接続を求める大量のリクエストがすべて同時に接続を始めることになるため、コネクションストームが発生します。コネクションストームはリソースを大量に消費します。MySQL で新しいデータベース接続を作成することは、コストのかかる操作です。バックエンドで初回のハンドシェイクのために複数のネットワークパケットを交換したり、新しいプロセスを生成したり、メモリを割り当てたり、認証を処理したりするためです。短時間のうちに大量のリクエストが届くと、データベースが応答していないように見えることがあります。

MySQL には、こうした急激な接続リクエストに対処するためのメカニズムがあります。`back_log` 変数は、MySQL が新規リクエストへの応答を一時的に停止するまでの、短時間の間にスタックできるリクエストの数に設定することができます。この値は接続処理スレッドによって適用されるため、スレッドそれ自体がコネクションストームに対応できなくなる可能性があります。詳細については、「MySQL リファレンスマニュアル」を参照してください。

データベースの動作が遅いときはリセットするように接続を設定しておくと、このサイクルが何度も繰り返されます。同様に、日中の特定の時間帯 (株式市場の開始直後など) にデータベーストラフィックの急増が予想される場合は、接続プールを事前に実行しておき、トラフィックの負荷の急増と、多数の接続のオープンが同時に発生しないようにします。

クエリワークロードの調整

ワークロードを適切に調整すれば、急激な成長に対処するための安定したソリューションを手に入れるのに大いに役立ちます。ワークロードを適切に調整していないと、使用する Amazon Aurora クラスターにどれほどの性能があっても、パフォーマンスを下げたりアプリケーションのユーザーエクスペリエンスに影響を与えたりするボトルネックが生じる可能性があります。ベストプラクティスは、システム内の問題のあるクエリを特定して調整するのに役立つプロセスを、最初から用意しておくことです。

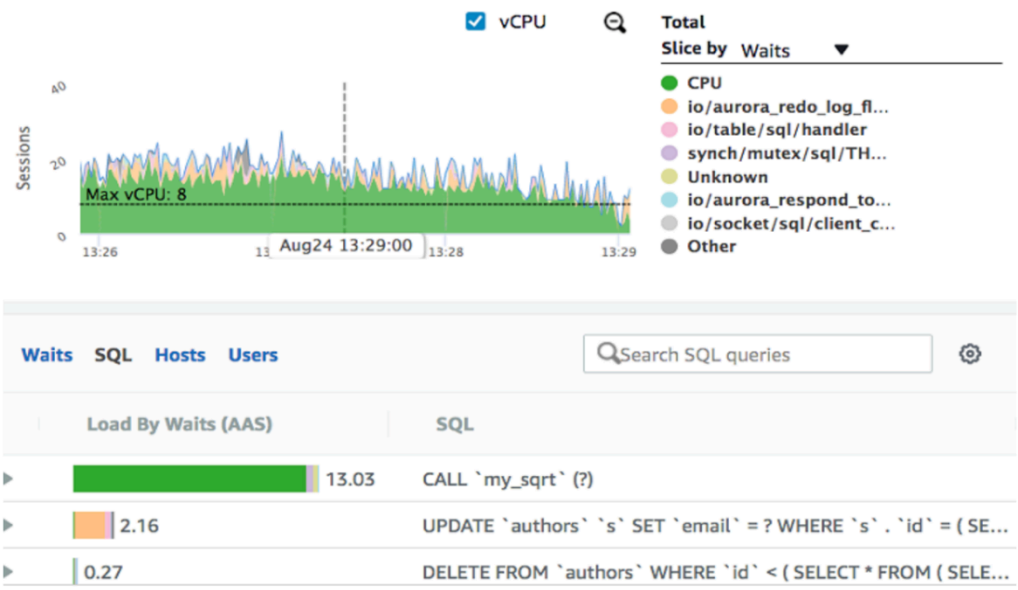
ワークロード内の問題のあるクエリの追跡と調整

急激な成長に遭遇したときは、ワークロードを適切に調整すれば問題を解決したも同然になります。Aurora クラスターが直面している、ワークロードとパフォーマンスに関するリアルタイムの問題の性質を把握するには、チームでライターインスタンスとリーダーインスタンスの両方から問題のあるクエリを収集します。このような問題のあるクエリは、ワークロードが最適な状態で実行されるように調整する必要があります。Amazon Aurora MySQL 互換エディションには、そのための 2 つの方法が用意されています。

- Performance Insights
- スロークエリログ

Performance Insights の使用

Performance Insights は、平均アクティブセッション (AAS) に基づいて Aurora ライターインスタンスまたはリーダーインスタンスの負荷を追跡します。AAS の値は、サンプリングと、CPU がクエリのワークロードを取得し処理するのを待機しているアクティブなセッションの数とを使用して計算されます。Performance Insights のグラフィカルインターフェイスでは、アクティブなセッションを待機しているために最も高い負荷を引き起こしている SQL ステートメントを確認できます。



上記のスクリーンショットでは、ストアドプロシージャ `my_sqrt` の呼び出しのため平均 13.03 のセッションが自らの負荷が処理されるのを待機しています。論理的な次のステップでこの手順を調整します。リーダーとライターで、それぞれのインスタンスに負荷をかけている SQL ステートメントを特定し、AAS 値が低下して Performance Insights の点線 Max vCPU を下回るまで、パフォーマンスが改善するように調整します。調整作業が上限に達したにもかかわらず、AAS が依然 Max vCPU の点線を上回っている場合は、より大きなインスタンスクラスを選択してワークロードに対処します。クエリのワークロードの調整を試みる前からより大きなインスタンスを選択することはしません。トラフィックが増えると、ワークロード内の質の悪いクエリによって生成された断層線が露出し始めるためです。

スロークエリログの使用と CloudWatch へのパブリッシュ

スロークエリログは、MySQL のネイティブな機能であり、Performance Insights を補完する機能です。そのベストプラクティスは、これらの方法を両方とも使用して、インスタンスに損害をもたらす可能性のある問題のあるクエリを未然に防ぐことです。スロークエリログには、動的変数 `long_query_time` よりも遅いクエリがすべて記録されます。この変数は、クラスターインスタンスを再起動することなく設定できます。

調整を柔軟に行い、その作業を隔離するため、ライターインスタンスとリーダーインスタンスには個別のパラメータグループを使用します。このことは、読み取り/書き込みの分割を使用する場合に特に重要になります。必要に応じてクラスターインスタンスの `long_query_time` に十分な上限を設定します。負荷を調整するときは、しきい値をミリ秒単位で設定できるため、`long_query_time`

変数に積極的な値を設定できます。同時実行性が高くワークロードが適切に調整されていれば、ほぼすべてのクエリは数ミリ秒で実行されるはずです。

Amazon Aurora MySQL 互換エディションを、スロークエリをファイルに記録するよう設定すると、Aurora MySQL 互換がスロークエリログを Aurora MySQL 互換ファイルシステムに書き込んでこれを 24 時間保持します。分析のためにスロークエリログを長期間保持するには、Amazon CloudWatch にパブリッシュします。また、CloudWatch ダッシュボードを作成してスロークエリをモニタリングすることもできます。詳細については、ブログ記事「Creating an Amazon CloudWatch dashboard to monitor Amazon RDS and Amazon Aurora MySQL」を参照してください。Amazon CloudWatch にあるスロークエリを分析するだけでなく、Percona Toolkit のツールである pt-query-digest を使用するとスロークエリログをプロファイリングできます。

さらに、クエリのダウンロードとプロファイリングのプロセスを自動化してチームの効率を高めることもできます。頻繁に実行されているクエリや長い間隔を空けて実行されているクエリをチェックして、調整の優先順位を決めるとよいでしょう。スロークエリログに記録されるクエリの数がほとんどない状態を目指すことで、`long_query_time` を減らし、ワークロードを把握して調整する間にさらに積極的な値を目指すことができます。

クエリ調整のガイダンス

ワークロード内で問題のあるクエリを見つけたら、各クエリを調整する必要があります。調整に関する以下のガイドラインを参考にすると、ワークロードをより効率的に実行できます。

スキャンする行数を最小限に抑える

一見基本的なことのようですが、クエリの調整に役立つ重要なアドバイスです。EXPLAIN ステートメントを使用して、row 欄で、各 join でオプティマイザーがスキャンする行数を確認します。最適なインデックスを作成することでスキャンする行数を減らし、クエリを再説明して作業を確認します。詳細については、MySQL ドキュメントを参照してください。

パーティションテーブルを使用する場合は、オプティマイザーがパーティションを 1 つ 1 つスキャンする必要をなくすため、必ず、パーティションの削除を有効にする WHERE 句を使用してクエリを実行します。WHERE 句にパーティション化された列の定数が含まれていると、オプティマイザーは探すべきパーティションを把握できるため、クエリをより効率的に行えます。

このアドバイスにはもう 1 つ、データベースの設計に関する側面があります。クエリに含まれるテーブルの数が少ないほどクエリは速く実行されます。データベースの設計を非正規化できれば、オプティマイザーがスキャンする行の数を減らすことができ、クエリのパフォーマンスを速めることができます。

一時テーブルの使用と、ディスク上の一時テーブルの数を最小限に抑える

Aurora MySQL 互換オプティマイザーは、クエリに関する望ましい結果をインデックスから直接得られない場合は、RAM とディスクの両方に一時テーブルを作成します。そのため、ワークロードに適したインデックスを用意することに、調整作業の大半を費やすことになります。ただし、ワークロードの中にはインデックス以外のものに依存しているクエリもあり、オペレーションによっては一時ファイルで実行される場合があります。こうした処理が最小限に抑えられ、ディスク上に作成されるテーブルがごくわずかであれば、このようなことは問題はありません。MySQL は、一時テーブルのサイズが大きくなりすぎてメモリに収容できなくなったとき、ディスクテーブルを作成します。MySQL が、内部の一時テーブルのサイズを確認するときに使用する論理は、tmp_table_size と max-heap-table-size の 2 つの変数値のうちいずれか小さい方です。これらの変数は、ワークロードに基づいて最適な値に調整できるため、一時テーブルの使用を回避できない場合であっても、ディスクへのプッシュはまれにしか行われません。

ファイルのソートを回避する

ワークロードに大量の `ORDER BY` クエリがある場合、これらを解決する最善の方法は、テーブルで適切なインデックスを使用することです。お使いのマルチカラムインデックスが、ファイル内でソートされないように適切に設計されていることを確認しましょう。前列が定数でスキャンされていない列は、ソートを実行できません (`in`、`>`、`<`、`!=`、`BETWEEN` は、右側にある次の列のソートを許可しません)。MySQL でソートを実行する最適な方法は、連続構造の中に、クエリで指定された定数値を含む列を、ソート対象の列の隣接する左側に配置するマルチカラムインデックスを配置することです。最後の手段として、ファイルソートを行わないとクエリの結果が返されない場合は、ソートをアプリケーションに移します。

集計クエリが高い同時実行性で実行されることを回避する

ワークロードには、アプリケーション内の一部の機能を満たすために少数の集計クエリが含まれていることがあります。このユースケースには、細心の注意が必要です。InnoDB エンジンは、オンライントランザクション処理 (OLTP) の適切な負荷に対応するように設計されていますが、group by クエリが高い同時実行性で数回実行されただけでも CPU には非常に重い負荷がかかり、クラスターのパフォーマンスが急速に下がる可能性があります。集計結果のセットが必要になるユースケースを解消するには、group by クエリをまったく行わずに済むように、読み取りの準備が整ったテーブルに、データを事前に集計します。

クエリの同時実行性をテストする

クエリを個別に調整するときは、それらのクエリが Aurora MySQL 互換の、複数の vCPUs で同時に実行されていることに留意します。テスト環境では 1 回につき数ミリ秒でクエリが実行されることもあります。しかしこれがすべてではありません。必ず、本番環境のクラスターで予想されるレベルの同時実行性でクエリをテストし、パフォーマンスのベンチマークを評価するようにします。クエリは、同時実行性の目標を満たしている場合のみ、本番環境にリリースするようにします。テストスクリプトでは、キャッシュから結果を取得することのないよう、必ずオプティマイザー hint `sql_no_cache` を使用します。同時実行でテストを実行し、結果を評価するには、mysqlslap などのツールを使用します。

書き込みワークロードの調整

ロードバランシングを実装し、ライターインスタンスを解放すると、書き込みワークロードが急増したときのパフォーマンスを改善することができます。高い同時実行性での書き込みパフォーマンスをさらに改善するには、次の追加手順に従います。

参照整合性をアプリケーション層に移行する

参照整合性チェックは重要なチェックですが、ハイパースケール時には負荷にマイナスの影響を及ぼす可能性があります。書き込みのたびに、追加のスキャンを行ってから書き込み自体を実行する必要があるため、パフォーマンスの低下につながります。アプリケーションに厳密な整合性チェックが必要な場合は、書き込みのスロットリングを回避するためチェックをアプリケーション層に組み込みます。

重いプライマリキーの使用は避ける

プライマリキーは軽い状態に保つようにします。InnoDB のストレージエンジンは、テーブルに作成される他のすべてのインデックスにプライマリキーを追加します。プライマリキーが大きいと、インデックスのサイズに影響します。プライマリキーが非常に大きいと、データページの保存および取得のスピードが遅くなります。一般的な例として、汎用一意識別子をプライマリキーとして使用する例が挙げられます。この方法は、ハイパースケール後の環境でのパフォーマンスを対象としている場合は適していません。

パーティション交換を使用してパーティションテーブルにデータを読み込む

大量のデータセットをパーティションテーブルに書き込む場合は、LOAD DATA FROM S3 とパーティション交換を組み合わせるとパフォーマンスが向上します。挿入の際にメインテーブルにアクセスしないためです。パーティション交換にはデータ定義言語 (DDL) を使用し、テーブルにメタデータロックを適用します。この処理は、パーティション交換 DDL が待機することなくメタデータロックを取得できるよう、テーブルで実行されているクエリが最小限かまたはまったくないときに実行します。パーティション交換それ自体は数ミリ秒で完了します。

未使用インデックスの削除

InnoDB はデータの増加に応じてクエリプランを最適化するため、データベース内で未使用のインデックスがないかチェックし、あれば削除することが推奨されます。未使用のインデックスは、アプリケーションがテーブルにデータを書き込む際に IO を使用します。未使用のインデックスのリストをチェックし、それらが四半期レポートなどまれな状況で使われるインデックスではないことを確認します。また、インデックスの中には一意性や順番を強化するために使用され、考慮する必要があるものもあるため、注意します。

古い行バージョンは効率的に削除する

多版型同時実行制御 (MVCC) の InnoDB 実装では、レコードが変更されると、変更されたデータの現行 (古い) バージョンが undo ログに undo record として記録されます。履歴リストの長さ (HLL) の値が増えた場合、InnoDB ガベージコレクションスレッド (パージスレッド) が書き込みワークロードに追いついていないか、パージが長時間のクエリまたはトランザクションによってブロックされていることを意味します。ガベージコレクションがブロックされたり遅延したりすると、データベースで大幅なパージの遅延が発生し、クエリのパフォーマンスにマイナスの影響を及ぼす可能性があります。次の推奨事項を参考にすると、パージのプロセスを最適化できます。

- トランザクションを少量に抑えます。
- 読み取りクエリに READ COMMITTED 分離レベルを使用します。
- パージスレッドの数を増やします (innodb_purge_threads と innodb_purge_batch_size)。これらのパラメータを調整するには再起動が必要になります。
- HLL を定期的にモニタリングし、ガベージコレクションの進行を妨げているワークロードの問題を解決します。

ログ記録のせいで競合がさらに発生しないようにする

一般的なクエリログには、サーバーが受信したすべてのステートメントが受信順に記録されているほか、クライアントの接続と切断も記録されます。ログ記録は有効にすると同期的に実行されるため、負荷の高いシステムではパフォーマンスが大幅に低下する可能性があります。必要なければ、一般ログは非アクティブにすることが推奨されます。

スロークエリログには、実行に要する秒数 long_query_time より長く時間がかかったステートメントが記録されます。デフォルトの設定は 10 秒です。0 に設定するとすべてのステートメントが同期的にログ記録されるため、負荷の高いデータベースではパフォーマンスが低下する可能性があります。

負荷を軽減する

応答時間を改善し、重要なワークフローに利用できるリソースを増やすには、無関係な負荷を取り除くことが必要になる場合があります。このセクションで取り上げるソリューションの多くは、トレードオフによる意思決定を伴います。これらはアプリケーションに影響を及ぼすため、慎重に検討する必要があります。以下の状況でこのソリューションを使う場合について検討してみましょう。

- すでに最大サイズのインスタンスを使用しています。特にプライマリインスタンスには、書き込み可能なデータベースインスタンスを使用しています。
- 最後の手段として、他の変更を実装するために十分なヘッドルームを短期間で提供します。

すぐに行うべき変更には、以下が含まれます。

- 重要でない読み取りトラフィックをプライマリ DB インスタンスから移動させます。
- 古いデータをアーカイブするか削除します。
- 参照整合性を削除します。
- binlog を無効にします (使用している場合)。
- 重要でない抽出、変換、ロード (ETL) の処理を延期します。
- 重要でないアプリケーション機能を停止するかグレードを下げます。

上記のアクションを実行するときは、事前に、長期的なビジネス目標とリスクをふまえた評価を行います。

読み取りと書き込みのワークロードを分離する

MySQL を搭載したアプリケーションを実行するときの一般的なテクニックが、読み取りオペレーションをライター (プライマリ) データベースインスタンスから、1 つまたは複数の読み取り専用データベースレプリカにオフロードする方法です。読み取りオペレーションをオフロードすれば、プライマリデータベースインスタンスの全体的な負荷を軽減し、書き込み用にスペースを空けることができます。必ず、レプリカの書き込み直後の読み取り整合性に依存していない読み取りのみを対象にします。より効率的な方法は、すべての読み取りトラフィックをレプリカに移行し、レプリケーションが遅れた場合に書き込み後の読み取りを再試行するよう計画を立てることです。オフロード可能な独立した読み取りワークロードには、例えばレポートサービスなどがあります。他の読み取りでは、アプリケーションレベルでの変更が必要になります。ここでは、読み取りが発行された理由の背景事情は判明しています。

もう 1 つの方法は、アプリケーションとデータベースの仲介役としてデータベースプロキシソリューションを実装する方法です。これにより、アプリケーションを意識することなく、読み取り/書き込みの分割とクエリルーティングの機能を提供できます。MySQL 互換のプロキシソリューションを利用できる製品には、ProxySQL、MariaDB MaxScale、ScaleArc、Heimdall Data などがあります。これらの製品では、キャッシュや接続の多重化など複数の追加機能を利用できます。トラフィックが突然増加したときや、アプリケーションスタックに新しいテクノロジーを実装するときは、読み取り/書き込みクエリを分割する基本機能から使い始め、予期せぬ影響を引き起こす可能性のある追加機能は、基本機能の動作とパフォーマンスをテストした後に使用するとよいでしょう。

古いデータをアーカイブするか削除する

データベースのパフォーマンスを向上させるもう 1 つのテクニックが、過去のデータを別のテーブルか、データベース、または Amazon Simple Storage Service (Amazon S3) にオフロードする方法です。多くのデータベースが、アプリケーションの全履歴のデータをインラインで保持しています。一般的なユーザー向けアプリケーションでは、これのおかげでユーザーは自分の注文履歴をすべて閲覧することができます。需要が突然増加した場合、そのアプリケーションを利用しているユーザーの多くは、新規ユーザーか、新たに注文するために殺到したユーザーであると考えられます。履歴データが、数十億の行を含む単一のテーブルにオンラインで保存されていると、状況はいっそう悪化します。また、このテーブルには大量のインデックスが含まれている可能性があります。テーブルデータとインデックスは共にツリー構造で保存されます。テーブル内のエントリが増えるとツリーのレベルも増えることになり、行にアクセスするには、より多くの I/O オペレーションが必要になります。これにより、個々のレコードを検索するためのアクセス時間が長くなります。さらに重要なこととして、インデックスの大量の不要な要素がデータベースページキャッシュに滞留する原因となります (InnoDB バッファプール)。

古いデータを別のテーブル、別のデータベース、あるいは Amazon S3 にアーカイブすれば、エンドユーザーのアクセス時間を短縮でき、貴重なキャッシュを解放でき、アプリケーション全体のパフォーマンスを改善することができます。イベント前に古いデータをアーカイブすると (30 日のみまたは 90 日のみ保存するなど)、重要なテーブルの、テーブルとインデックスの容量が制限されます。この変更により、二次的な場所からの古いデータのクエリは、履歴データの閲覧を明示的に要求する一部のユーザーのみとするよう、アプリケーションを変更することが必要になります。

重要でない ETL プロセスを延期する

メインのデータベースシステムから ETL プロセスを抽出すると、ハイパースケール中に、トランザクションの多いワークロードや同時実行ワークロードの安定性が損なわれる可能性があります。ETL のプロセスには次のような特徴があります。

- トランザクションの実行時間が長い
- ロックの範囲が広い
- CPU、メモリ、I/O の使用
- 重要なエンドユーザーパスを処理するトランザクションワークロードと競合する

変動したり予測不能だったりする ETL プロセス (`INSERT INTO ... SELECT FROM ...;`) のようなクエリなど) は、負荷と競合の全体的な変動性を高め、安定上のリスクを高めます。

可能な場合は、重要な機能を果たしていない場合は特に、ETL プロセスの実行を遅らせるか実行頻度を減らします。重要な ETL プロセスについては、固定された行数でバッチ処理したり(`LIMIT` 句を使用するなど)、個別のトランザクションを使用したり、少量のデータを長期間プルして DB インスタンスのピーク時のリソース需要を減らしたりするなど、制限のある作業単位で実行するように修正します。

重要度の低いアプリケーションの機能をオフにする

急増するリクエストに対処するときは、ユーザーエクスペリエンスのグレードを適切に下げたり、最重要ではない機能を削除したりすると、データベースリソースを節約し、重要な機能に振り向けることができます。そのためには、カスタマーエクスペリエンスを若干変更する必要があるかもしれませんが、サイトがダウンするよりは別のフローの方が良いでしょう。データベース呼び出しを常に行うサイトのフロントページで、パーソナライゼーションを一時的に無効にしてもよいかもしれません。あるいは、ユーザーが製品を選んでいる間はリピーター専用のオファーの表示を停止するという方法もあります。機能を一時的にオフにすると、データベースにアクセスしなくてもページをキャッシュしたり配信したりできるようになります。

その他の例として、データベース呼び出しを必要とするデータ変更をチェックする、ポーリングのメカニズムを備えたフロントエンドがあります。ポーリングの頻度を減らすと、すぐにデータベース呼び出しの回数が減ります。ページ分割が必要なユーザーインターフェースや、ソートされた結果セットを上位の検索結果から遠く離れて取得するオプションを提供しているユーザーインターフェースでは、データベース呼び出しのコストは徐々に高くなります。最も高額なデータベース呼び出しからデータベースを保護するには、結果セットのページ数を制限します。アプリケーション層で機能フラグを使用すると、アプリケーションまたはデータベースの負荷が増加したときに、オペレーションチームが機能を無効にしたり、グレードを下げたりできるようになります。

パラメータの調整

接続関連のパラメータとは別に、ハイパーグロースに直面したときに Amazon Aurora MySQL 互換エディションクラスターのパフォーマンスを向上させるために調整できるパラメータがいくつかあります。データベースパラメータを変更するときは、次のベストプラクティスを使用することが重要です。

- パラメータは一度に 1 つずつ変更し、各変更の影響を測定できるようにしておきます。
- パラメータの中には、変更後にその効果が現れるまでに一定のウォームアップ期間が必要になるものがあります。Aurora MySQL 互換クラスターのパフォーマンスを観察し測定するときはこの点を考慮します。
- 誤ったパラメータ値を使用しないように注意します。値が誤っているとデータベースインスタンスが起動しない可能性があります。

パラメータには以下が含まれます。

- `sync_binlog`
- `table_open_cache`
- `innodb_sync_array_size`
- `innodb_flush_log_at_trx_commit`
- `autocommit`
- `tmp_table_size`
- `max_heap_table_size`

これらのパラメータの設定に関するガイドラインについては、 AWS ドキュメントの「Aurora MySQL 設定パラメータ」、「Aurora MySQL MySQL での MySQL 機能の推奨事項」、および「Aurora MySQL での一時テーブルの動作」を参照してください。

リソース

- Journey to Adopt Cloud-Native Architecture Series: #1 – Preparing your Applications for Hypergrowth (ブログ記事)
- Journey to Adopt Cloud-Native Architecture Series: #2 – Maximizing System Throughput (ブログ記事)
- Amazon RDS Proxy の使用
- キャッシュ戦略
- Performance Insights の有効化と無効化
- The InnoDB Storage Engine
- Aurora レプリカ
- MariaDB Connector/J
- MariaDB MaxScale
- ProxySQL
- Heimdall Data

ドキュメント履歴

以下の表は、本ガイドの重要な変更点について説明したものです。今後の更新に関する通知を受け取る場合は、RSS フィード をサブスクライブできます。

変更	説明	日付
初版発行	—	2022 年 10 月 5 日

翻訳は機械翻訳により提供されています。提供された翻訳内容と英語版の間で齟齬、不一致または矛盾がある場合、英語版が優先します。