Amazon RDS Performance Insightsから始めるPostgreSQL パフォーマンス深掘り生活

出利葉
Ideriha

内容についての注意点

- 本資料では資料作成時点のサービス内容および価格についてご説明しています。AWS のサービスは常にアップデートを続けているため、最新の情報は AWS 公式ウェブサイト (https://aws.amazon.com/) にてご確認ください
- 資料作成には十分注意しておりますが、資料内の価格と AWS 公式ウェブサイト記載の価格に相違があった場合、AWS 公式ウェブサイトの価格を優先とさせていただきます
- 価格は税抜表記となっています。日本居住者のお客様には別途消費税をご請求させていただきます
- 技術的な内容に関しましては、有料の AWS サポート窓口へお問い合わせください
- 料金面でのお問い合わせに関しましては、カスタマーサポート窓口へ
お問い合わせください (マネジメントコンソールへのログインが必要です)

前提

- 本セッションでは、RDS for PostgreSQL 16.3 の動作を基にご説明します
- Performance Insights のご利用方法など多くの点は、 Amazon Aurora PostgreSQL でも同様です
- ただし、例えば本セッションで言及する checkpoint や Full Page Writes による書き込みの増加は Aurora では発生しないなど一部 RDS と異なる動作があります
- Performance Insights は AWS の機能ですが、OSS の PostgreSQL を利用されている方にも参考になる部分があると考えます

性能問題は難しい～「遅い」を具体化したい～

- 何が遅いのか
  - アプリか、DB か、単一の SQL か、全 SQL か?「遅い」の定義は?
- 何故遅いのか
  - 原因は多岐にわたり複雑、調査に必要なメトリクス・ログはあるか
    - CPU/メモリ/ネットワーク/ディスク
    - ロック、実行計画、バージョンアップ
    - アプリの挙動の変化
  - メトリクス・ログを解釈する PostgreSQL の知識や調査ノウハウはあるか
- 対処はどうするのか
  - 応急措置を実施するにもどの様な措置が妥当か判断できるか

遅い場合と早い場合の比較が必要

- 性能に影響を与える要素は多様。メトリクスも多様。「平常運転」の性能・メトリクスを比較し、差分に注目して原因追及するのが重要
- ログの設定例（一部を抜粋）：
  - auto_explain : 遅いクエリの実行計画を出力
  - log_lock_waits : deadlock_timeout 以上待機したテーブルロックの情報を出力
  - log_temp_files : 一時ファイルが生成された場合に情報を出力
  - and more ...
- OS/ハードウェア観点: CloudWatch メトリクス、拡張モニタリング
- 統計情報ビュー
  - 定期的にpg_stat_activity, pg_stat_statements, pg_stat_database... etc を確認

Amazon RDS Performance Insights（PI）

PI 不使用の場合

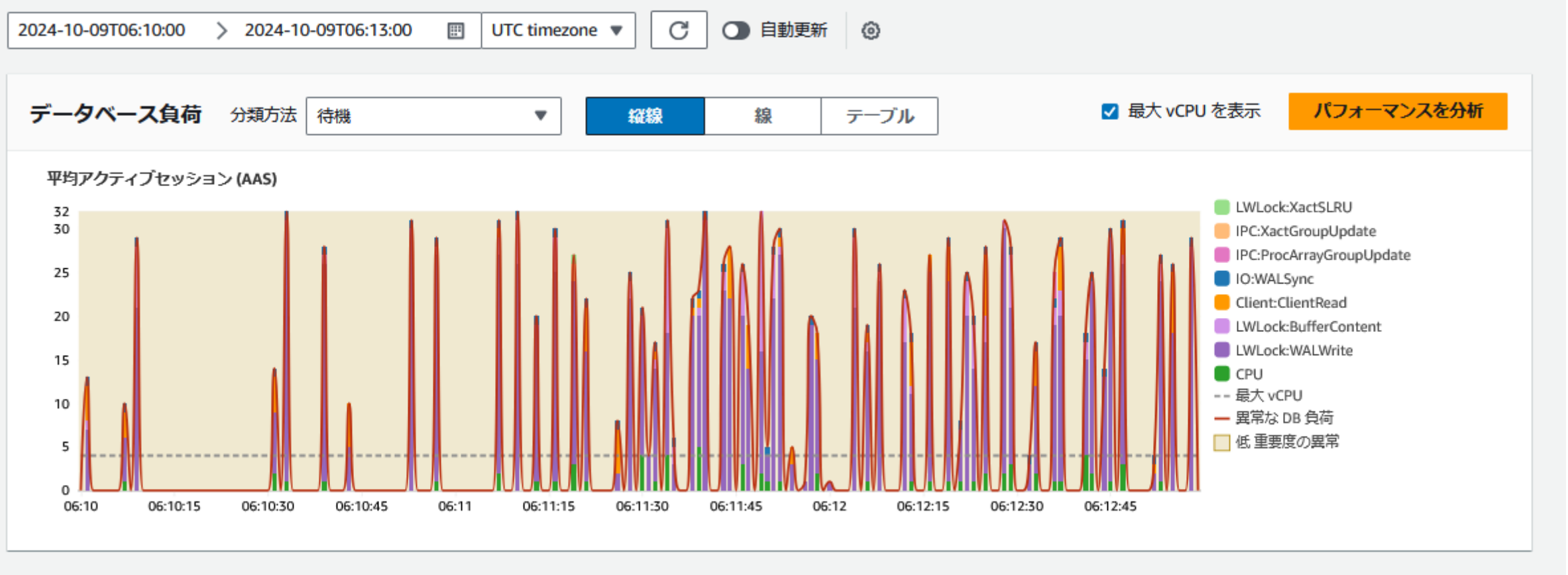
- データの収集時の問題：
  - 定期的に各ビューをサンプリング
  - どこでどの様に保存・管理するか
- データ参照時の問題：
  - pg_stat_statements などは累積値。前回の取得との差分計算が必要
  - 多次元の時系列データを瞬時に解釈できる可視化が必要

PI 利用の場合

- マネジメントコンソールで有効化
- データの収集
  - AWS 基盤内で保存・管理
- データ参照時：
  - 表示範囲に応じて自動集計
  - 分かりやすいグラフ表示

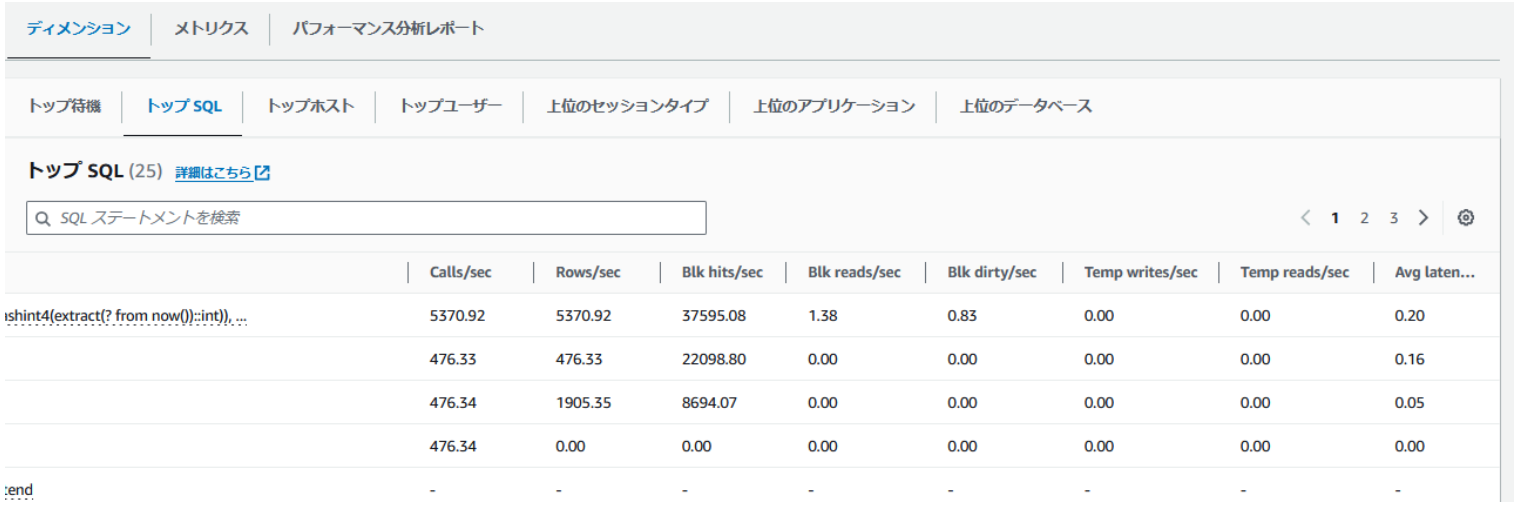
データベース負荷

- DB 全体の負荷がグラフで一目瞭然 （最小 1 秒単位）
- Average Active Sessions (AAS): ある時刻に実行中のセッション数



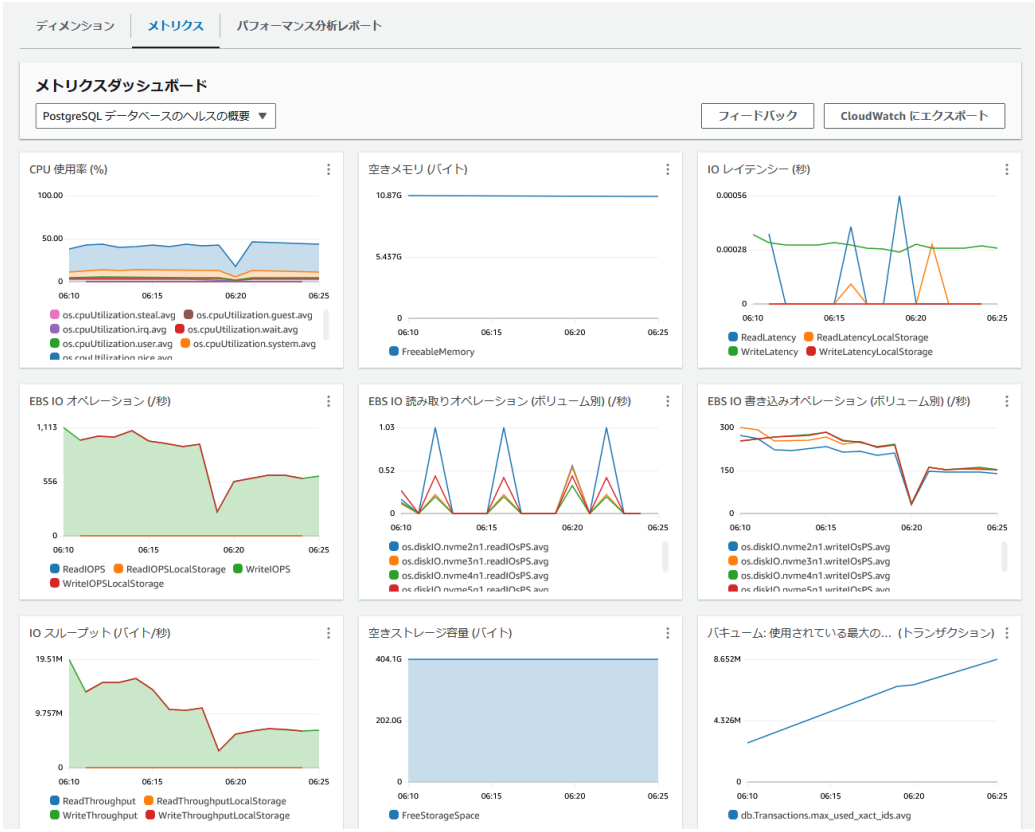
トップ SQL

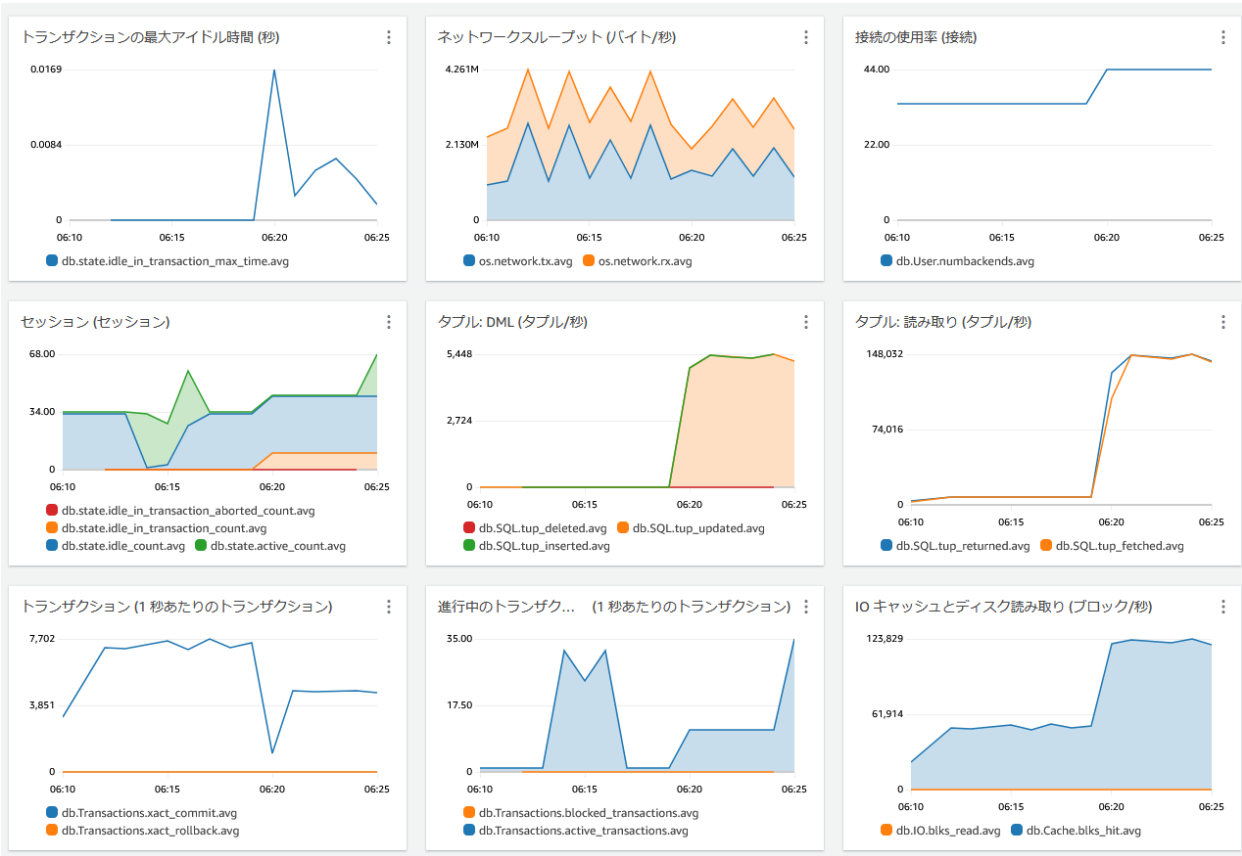
- SQL ごとの**負荷の割合**
- SQLごとの**統計情報**
  - pg_stat_statements から導出
  - Call/sec, Blks hits/sec などの**秒単位の統計**
  - Avg latency/call, Rows/call などの**コールごとの統計**
- 期間の指定
  - 正常時と異常時の性能低下を定量的に確認可



OS/PostgreSQL のメトリクス

- OS レベルの詳細なメトリクスも収集
- tup_inserted や checkpoints_req など一部の pg_stat_* の情報を 1 分粒度で収集・集計

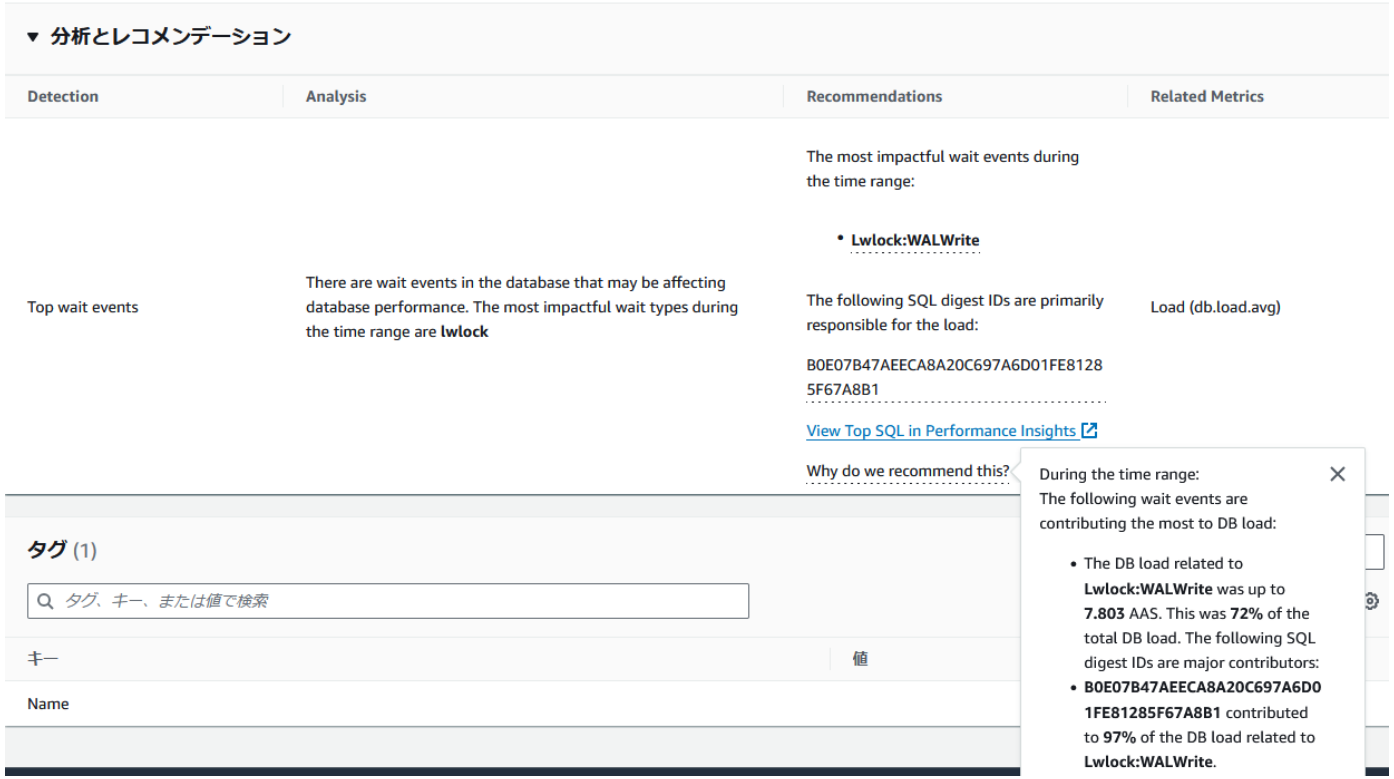




パフォーマンス分析レポート

- 保持期間 （デフォルトは 7 日間）を 1 か月以上にしている場合に利用可能
- 指定した範囲のパフォーマンスを自動で分析

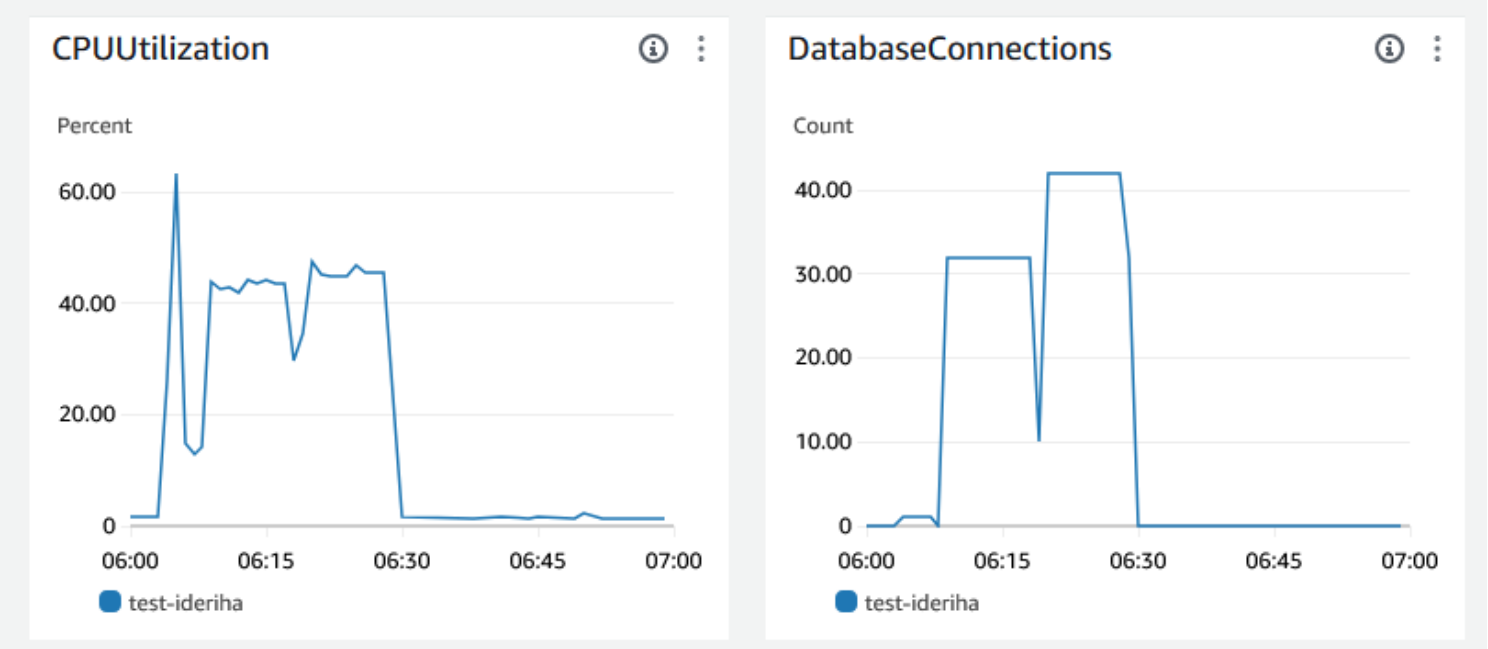
※ 保持期間に関する補足
例えば月 1 回だけ実行する処理を
過去の実行と比較するには
保持期間を 1 か月以上にすると良い



Performance Insights 活用事例

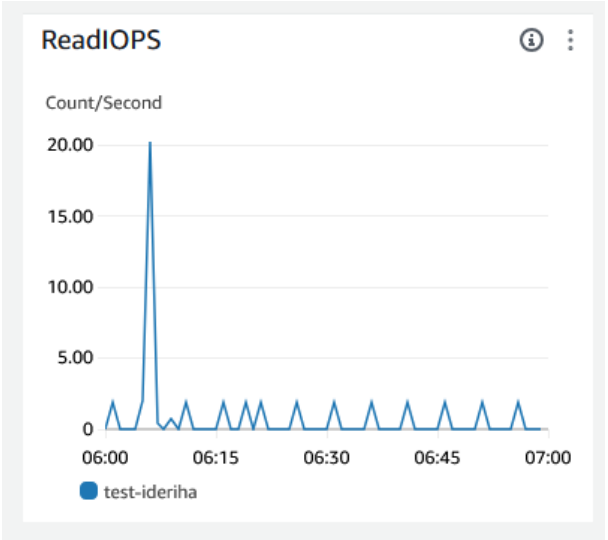
事例: バッチ処理が遅い

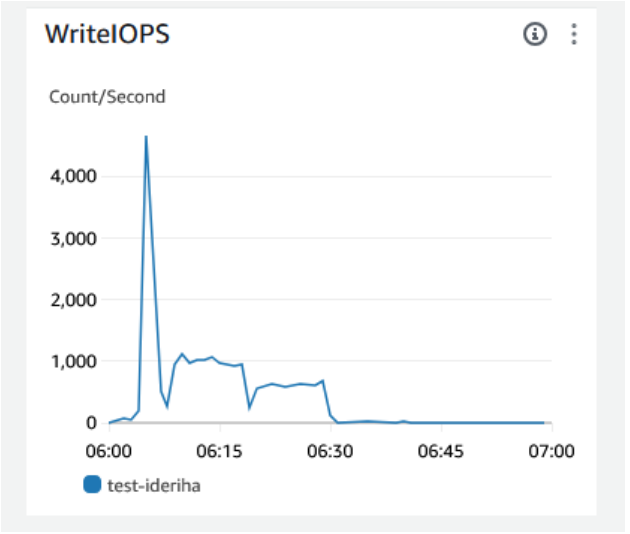
- 06:00 – 06:30 ごろの事象
- 同時実行数を上げると遅くなった
- CPU やメモリは足りている

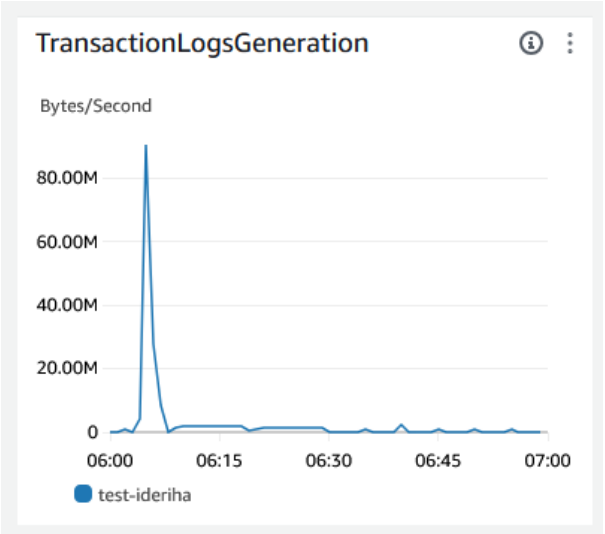


メトリクス確認続き

- ディスク IO では書き込みが支配的
  - TransactionLogsGenerationを見ると06:05ごろにWALが大量生成されている
  - 06:05から06:30 では書き込みはそこまで顕著ではない

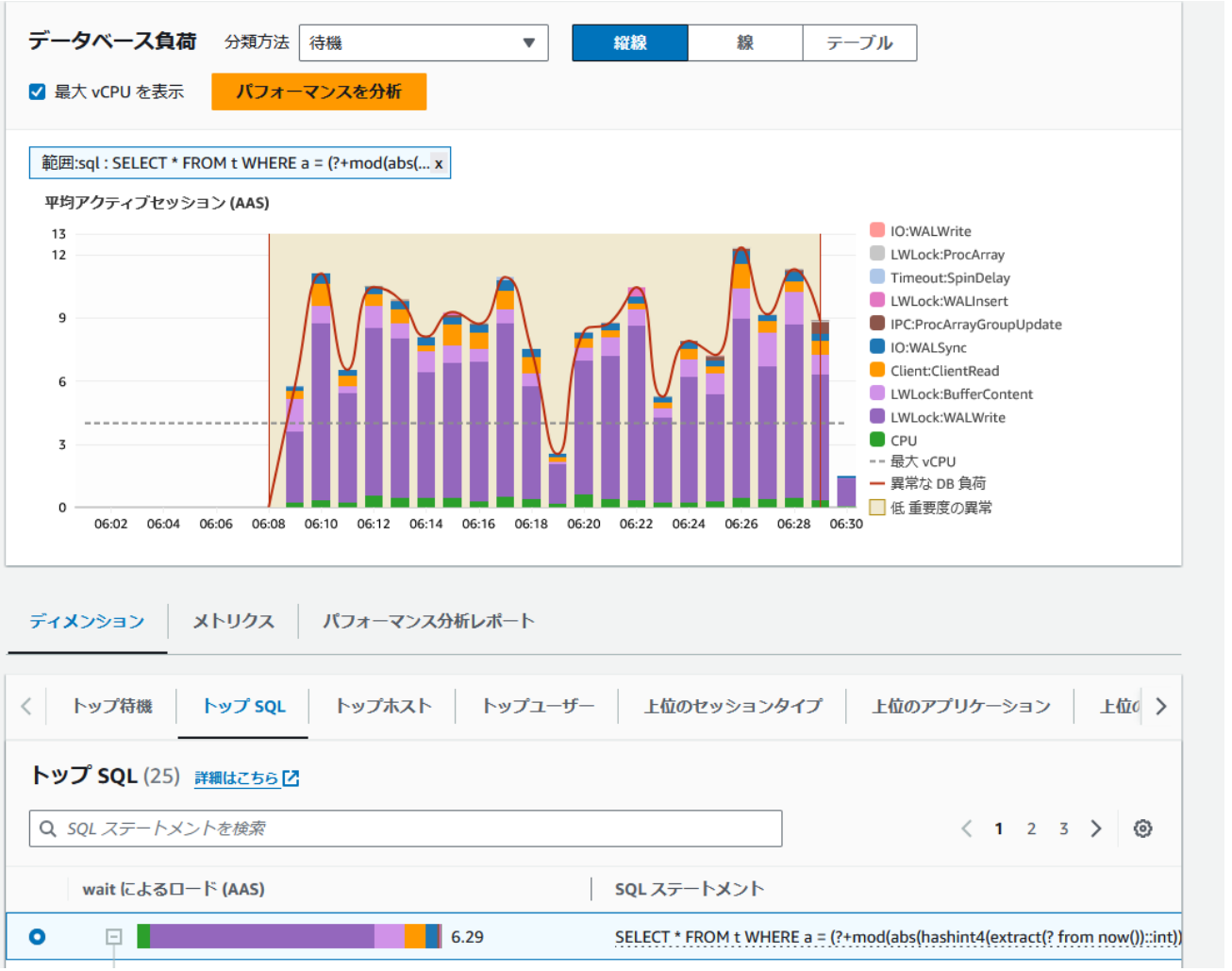






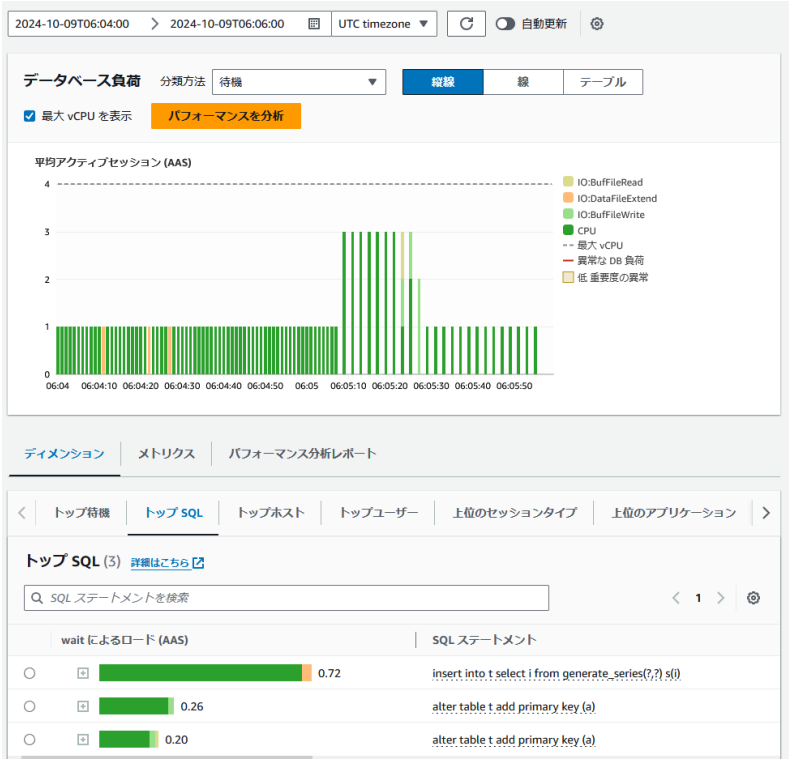
PI で詳細な動作を確認

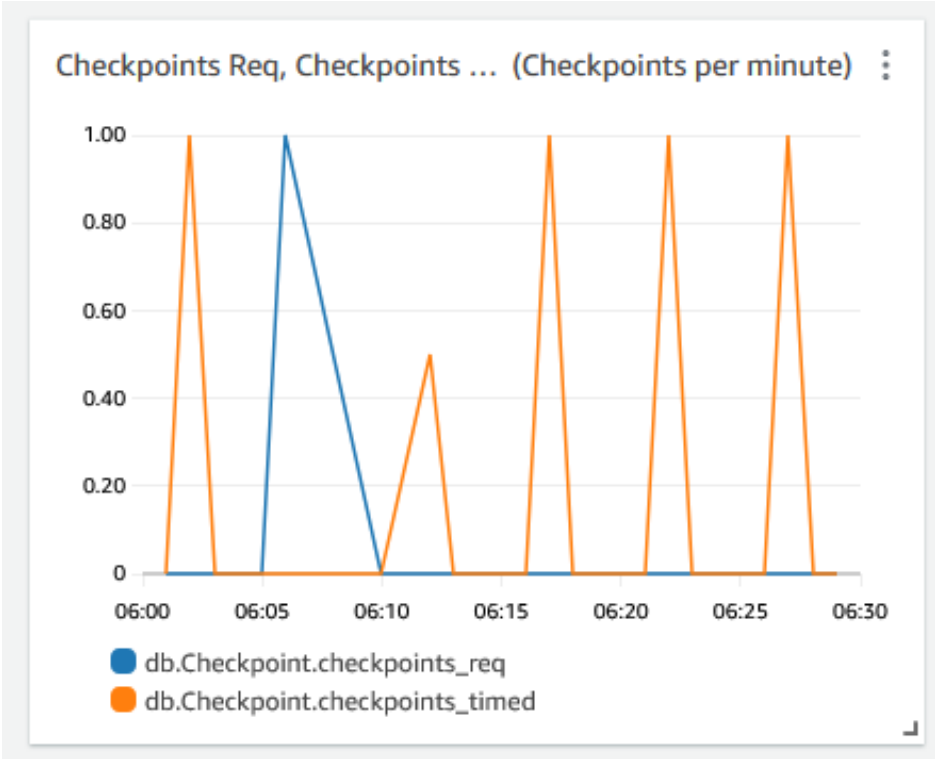
- SELECT … FROM … WHERE … FOR KEY SHARE が遅いようだ
- LWLock:WALWrite が支配的
  - WAL の書き込みを排他制御する軽量ロック
  - 短かいトランザクションが多い場合はトランザクションをまとめるとよい
  - そもそもSELECT クエリで WAL が発生するのか?
- 06:05 に書き込みが多かったが、SELECT クエリ起因ではない



06:05 の書き込みは INSERT 起因

- INSERT および インデックスの作成
- checkpoint_req が 1 であり、WAL が大量生成されたことによる checkpoint が実施されたことも分かる





SELECT で WAL が生成される場合はある

1. ヒントビットの更新
   - ヒントビットはタプルがコミット済みかどうかなどを記録する補助フラグ
   - data checksum 有効または wal_log_hints 有効の場合、ヒントビットの更新も WAL に書かれる。特にページの更新が checkpoint 後初めてだと、ヒントビットの更新で Full Page Writes によりページ全体が WAL に出力される
2. 行ロック（FOR UPDATE や FOR SHARE など）の影響
   - 行ロックではタプルのヘッダの xmax あるいは、multixact が更新され、WALが生成
3. HOT (Heap Only Tuple) 更新でのデフラグ
   - SELECT 時に VACUUM 相当のページの掃除が実行される

今回はこれらの要因と考えられる。この内容を知らない場合はどう調査する？

PI 以上の深い原因追及をするには

1. AWS や PostgreSQL のドキュメント、PostgreSQL の Wiki を確認
   - PostgreSQL の wiki に SELECT での WAL の生成 について記載あり
   - https://wiki.postgresql.org/wiki/Operations_cheat_sheet
   - 代表的な待機イベントの説明とトラブルシューティングについて https://docs.aws.amazon.com/ja_jp/AmazonRDS/latest/UserGuide/PostgreSQL.Tuning.html
2. WAL の中身を確認する
   - RDS では WAL ファイルを直接取得できない。しかし、PG 15 以降であれば **pg_wallinspect** 拡張でWALを確認できる
   - WAL が削除されると確認できないので、その場合は再現試験で確認

WAL の確認方法

- pg_get_wal_records_info(start_lsn pg_lsn, end_lsn pg_lsn) などを利用
- LSN (Log Sequence Number)の指定が必要。例) ”29/E8000180”
  - end_lsn は ‘FFFFFFFF/FFFFFFFF’ でも良い
  - 直近の LSN は “SELECT * FROM pg_current_wal_lsn(); ” で確認できる
  - 直近の checkpoint の開始時と終了時の LSN は、”SELECT redo_lsn, checkpoint_lsn FROM pg_control_checkpoint();” で確認できる
  - 現存する WAL ファイルは SELECT * FROM pg_ls_waldir(); で確認

    最も古いWAL ファイルが 000000010000006900000018 の場合、最初のLSN は下記で計算

    => ¥set file_name '000000010000006900000018’

    => ¥set offset 0

    => SELECT '0/0'::pg_lsn + pd.segment_number * ps.setting::int + :offset AS lsn FROM

    => pg_split_walfile_name(:‘file_name’) pd, pg_show_all_settings() ps WHERE ps.name =

    => 'wal_segment_size‘;

    結果例）‘69/60000000’ 。このLSN を start_lsn に指定するとこれ以降の WAL を確認できる

行ロックによって WAL が生じている

```
=> SELECT * FROM pg_get_wal_records_info('13/7000000', '13/72B08A0');
-[ RECORD  2]----+-------------------------------------------------
…
xid              | 13486821
resource_manager | MultiXact   ★ 複数のトランザクションが同じタプルに行ロックを取得したことを示す
record_type      | CREATE_ID     情報（MultiXact）を記録
…
description      | 10137658 offset 132166488 nmembers 5: 13486817 (keysh) 13486818 (keysh)
13486819 (keysh) 13486820 (keysh) 13486821 (keysh)
…
-[ RECORD 3 ]----+-------------------------------------------------
…
xid              | 13486821
resource_manager | Heap    ★ タプルのヘッダ（xmax） に作成された MultiXact の情報を記録
record_type      | LOCK
…
description      | xmax: 10137658, off: 152, infobits: [IS_MULTI, LOCK_ONLY, KEYSHR_LOCK], flags: 0x00
block_ref        | blkref #0: rel 1663/24631/24632 fork main blk 140814
```

タプル更新に伴い Full Page Writes が発生

- 行ロックによる更新に伴い、Full Page Writes (FPW) が発生している
- ワークロード前後の pg_stat_wal の wal_fpi を比較することでも確認可能

```
=> SELECT * FROM pg_get_wal_records_info('4F/90000000', '4F/98000230') where fpi_length <> 0;
-[ RECORD 1 ]----+------------------------------------------------
start_lsn        | 4F/90002A18
end_lsn          | 4F/90002FC8
prev_lsn         | 4F/900029F0
xid              | 16317069
resource_manager | Heap
record_type      | LOCK
record_length    | 1449
main_data_length | 8
fpi_length       | 1388  ★ Full Page Image の長さ。圧縮されており、サイズは 8kB より少ない
description      | xmax: 16317069, off: 122, infobits: [LOCK_ONLY, KEYSHR_LOCK], flags: 0x01
block_ref        | blkref #0: rel 1663/24631/24682 fork main blk 143748 (FPW); hole: offset: 928,
                   length: 32, compression saved: 6772, method: zstd
```

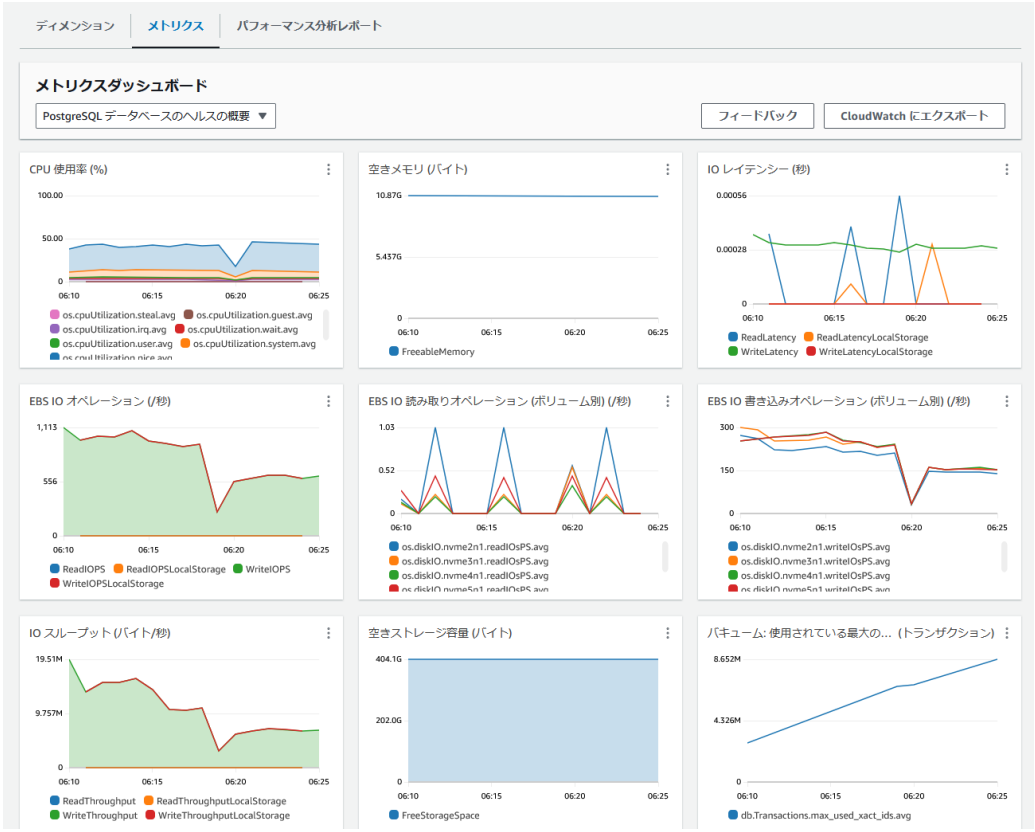
事例のまとめ

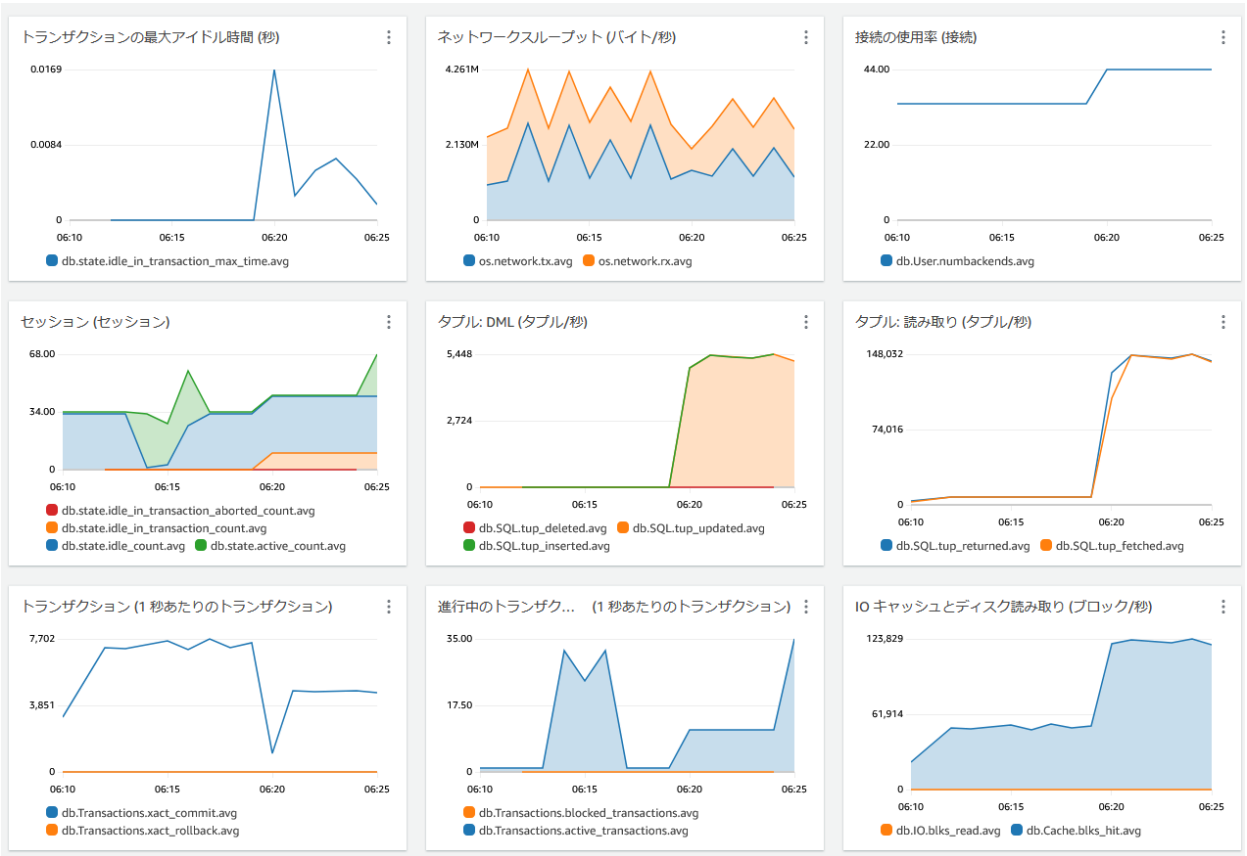
- Performance Insights により、問題の SQL およびその待機イベントが分かり、また関連するメトリクスも確認できた
- SELECT ... FOR SHARE による WAL 書き込みのロックが原因
- SELECT ... FOR SHARE で WAL が生成される理由は、各ドキュメントや実際の WAL を確認することで調査できた
- 今回のケースの解決策

– 複数の SQL をできる限り同一トランザクションにまとめる
– もし、FOR SHARE が不要ならば消す

補足：PostgreSQL のメトリクスの読み取り方

以降のページでは、代表的な PostgreSQL のメトリクスの読み取り方を説明します





意外と活用されていない PI のメトリクス活用法1

- tup_returned, tup_fetched でインデックススキャンが利用されているかを推測
  - tup_returned:「シーケンシャルスキャンで取り出された live タプル数」
    + 「インデックススキャンで返されたインデックスエントリ数」
  - tup_fetched: 「インデクススキャンで取り出された live タプル数」
  - 同一のワークロードと比べて性能が悪化し、tup_fetched が減り、tup_returned が増えている場合がある。インデクススキャンが利用できていない可能性あり。実行計画を取得できない場合などに有用
- temp_files, temp_bytes
  - 一時ファイルの生成量を確認
  - work_mem, maintenance_work_mem が不足している可能性

意外と活用されていない PI のメトリクス活用法2

- blks_read, blks_hits
  - blks_read の割合が一貫して高い場合、共有メモリが不足している可能性
  - インスタンスクラスのスケールアップ（shared_buffers を増やす）
  - SQL の見直しや、vacuum full/pg_repack/reindex が有効なケースも
- checkpoint_req
  - 短期間での max_wal_size を超える WAL の生成を示唆
  - max_wal_size を増やす、wal_compression を on にする、一時テーブルや unlogged テーブルの活用
  - ワークロード上WALの大量生成が避けられず、かつIOがボトルネックとなっている場合、インスタンスクラスや EBS のスケールアップ、WAL の格納先をデータの格納先と分ける Dedicated Log Volume が有効

意外と活用されていない PI のメトリクス活用法3

- buffers_backend
  - buffers_backend が多い場合、クエリを処理するバックエンドプロセスがダーティーバッファをディスクに書き込んでいる
  - インスタンスクラスのスケールアップ（shared_buffers を増やす）に加えて、バックグラウンドライター を積極的に動かす方針も
- OS レベルの各種メトリクス
  - 拡張モニタリングを１秒粒度にすると、OS メトリクスも１秒粒度になる
  - デフォルトの１分粒度ではとらえられない急激なメモリ不足などを確認可能

直接 pg_stat_* を実行すると良い場合

- 現在進行形でテーブルロックの待機が発生している場合
  - pg_stat_activity, pg_locks
    - どのプロセスが、どのオブジェクトにロックを保持しているか、どのプロセスがそのロックを待機しているかを確認
  - pg_terminate_backend()
    - ロックを長く保持しているプロセスを停止
- 検証環境で事象再現時に詳細な分析を実行する場合
  - 例 1) ワークロード実行前後で、pg_stat_user_tables/pg_stat_user_indexes をクエリしてワークロード前後でのテーブルごとの seq_scan 数などを分析
  - 例 2) ワークロード実行前後で pg_stat_wal の wal_fpi 列を確認し full page writes の有無や回数を調査

まとめ

- Performance Insights を利用すると、以下の情報を用いて性能問題を調査可能
  - DB 負荷の全体像 （例）発生している待機イベント
  - OSレベルのメトリクス （例）os.cpuUtilization.nice.avg
  - PostgreSQL の統計情報ビューのメトリクス （例）tup_returned
  - SQLごとの統計情報 （例）Avg latency/call
- 問題によっては Performance Insights だけからは性能問題の原因特定が難しい場合がある。そのような場合は、Performance Insights に加えて、他の手段を組み合わせるのが有効
- Performance Insights の情報を正しく読み解くには、PostgreSQL のナレッジの検索、事象を再現させて調査などが必要な場合がある

レファレンス

- Performance Insights のよくある質問
  https://aws.amazon.com/jp/rds/performance-insights/faqs/
- Amazon RDS インスタンスでのメトリクスのモニタリング
  https://docs.aws.amazon.com/ja_jp/AmazonRDS/latest/UserGuide/CHAP_Monitoring.html
  PI の利用方法について記載
- RDS for PostgreSQL の待機イベントでのチューニング
  https://docs.aws.amazon.com/ja_jp/AmazonRDS/latest/UserGuide/PostgreSQL.Tuning.html
  代表的な待機イベントに対するトラブルシューティングについて記載
- Amazon RDS と Amazon Aurora のパフォーマンスとイベントの可視性を高める
  https://aws.amazon.com/jp/blogs/news/increase-visibility-of-performance-and-events-on-amazon-rds-and-amazon-aurora/ PI や拡張モニタリングの活用法について記載
- PostgreSQL のドキュメントに未記載の重要の動作についての wiki
  https://wiki.postgresql.org/wiki/Operations_cheat_sheet

Thank you!