Amazon Aurora チューニング

アマゾン ウェブ サービス ジャパン株式会社
データベース スペシャリスト ソリューション アーキテクト
新久保 浩二

自己紹介

新久保　浩二 (しんくぼ　こうじ)

Database Specialist SA

Amazon Web Services Japan K.K.

- AWSでデータベースを動かす際のお手伝い

ライフワーク

- データベースのパフォーマンス問題の分析や最適化

お気に入りのAWSサービス

- Amazon RDS / Amazon Aurora
- Amazon RDS Performance Insights

aws

アジェンダ

- 一般的なチューニングサイクルと課題
- チューニングに必要なもの
- AWSでチューニングを支援する便利なサービス
- クエリーの実行計画を管理する
- サンプルのチューニングシナリオ
    - パフォーマンスダウンにおける実際のチューニング作業をやってみよう
    - パフォーマンスを安定化させるクエリー計画管理(QPM)を使ってみよう
- まとめ

一般的なチューニングサイクルと課題

aws

一般的なチューニングサイクル

一般的なチューニングサイクル

- OSレイヤーやDBの特定の統計情報を定期的に取得
  - 独自プログラム
  - 3rd パーティツール
  - CloudWatch
  - CloudWatch Logs
  - 拡張モニタリング

一般的なチューニングサイクル

コレクション

(データ収集)

- OSレイヤーやDBの特定の統計情報を定期的に取得
  - 独自プログラム
  - 3rd パーティツール
  - CloudWatch
  - CloudWatch Logs
  - 拡張モニタリング

アナリシス

(分析)

- 1つのメトリクスでは複数のメトリクスを組み合わせてデータベースがどのような挙動をしていたか類推する必要がある
  - CPU使用率 + キャッシュヒット率
  - 論理読み込み数 + 物理読み込み数
  - トランザクション数 + ロック待ちセッション数

など

チューニング

(最適化)

aws

一般的なチューニングサイクル

コレクション

(データ収集)

- OSレイヤーやDBの特定の統計情報を定期的に取得
  - 独自プログラム
  - 3rd パーティツール
  - CloudWatch
  - CloudWatch Logs
  - 拡張モニタリング

アナリシス

(分析)

- 1つのメトリクスでは複数のメトリクスを組み合わせてデータベースがどのような挙動をしていたか類推する必要がある
  - CPU使用率 + キャッシュヒット率
  - 論理読み込み数 + 物理読み込み数
  - トランザクション数 + ロック待ちセッション数

など

チューニング

(最適化)

- ある程度トライ & エラーによるチューニングの実施
  - 熟練DBAの経験に基づいたチューニング
  - データ収集できていない項目は、その都度手動で取得

aws

一般的なチューニングサイクルの課題

コレクション

(データ収集)

アナリシス

(分析)

チューニング

(最適化)

- 細かい粒度のパフォーマンスデータを長期間にわたり、収集、管理していくのは運用コストが高い
  - OSのメトリクスだけではなくDB特有のメトリクスを対象データベースに負荷なく細かい粒度で取得することが困難
  - データ量が多くなる中で安定したパフォーマンス分析インフラを維持管理するコストが高い

aws

一般的なチューニングサイクルの課題

コレクション

(データ収集)

- 細かい粒度のパフォーマンスデータを長期間にわたり、収集、管理していくのは運用コストが高い
  - OSのメトリクスだけではなくDB特有のメトリクスを対象データベースに負荷なく細かい粒度で取得することが困難
  - データ量が多くなる中で安定したパフォーマンス分析インフラを維持管理するコストが高い

アナリシス

(分析)

- パフォーマンス問題のような複雑で広範な事象を特定の統計情報から分析するのは困難
- DBの統計方法の組み合わせやその傾向などからボトルネックを特定することに限界がある
- どの程度リソースが不足、余剰なのかを見積もることが困難もしくは不可能

チューニング

(最適化)

aws

一般的なチューニングサイクルの課題

コレクション

(データ収集)

- 細かい粒度のパフォーマンスデータを長期間にわたり、収集、管理していくのは運用コストが高い
  - OSのメトリクスだけではなくDB特有のメトリクスを対象データベースに負荷なく細かい粒度で取得することが困難
  - データ量が多くなる中で安定したパフォーマンス分析インフラを維持管理するコストが高い

アナリシス

(分析)

- パフォーマンス問題のような複雑で広範な事象を特定の統計情報から分析するのは困難
- DBの統計方法の組み合わせやその傾向などからボトルネックを特定することに限界がある
- どの程度リソースが不足、余剰なのかを見積もることが困難もしくは不可能

チューニング

(最適化)

- ボトルネックの分析が曖昧、不十分なため、SQLやDBの効果的なチューニングが困難もしくは非効率
  - 完全に最適化された場合のゴールを示すことが難しい(要はどの程度チューニングされたら成功とみなすか判断できない)
  - リアルタイムでDB全体のパフォーマンス測定できていないため、チューニング効果の確認に時間がかかる

aws

チューニングに必要なもの

aws

パフォーマンスデータを収集、管理、分析するインフラ

コレクション

(データ収集)

アナリシス

(分析)

チューニング

(最適化)

- 突発的なパフォーマンス障害に対応するためには細かい粒度のデータが必要
- 過去に発生した問題を分析するためにはある程度長期間パフォーマンスデータの保存、削除するといった管理が必要
- パフォーマンス指標として一般的なOSのリソース情報やDB内部の動作を推測するためのDB統計情報に加えてDBのワークロードを端的に示す指標が必要

aws

パフォーマンスデータを収集、管理、分析するインフラ

コレクション

(データ収集)

- 突発的なパフォーマンス障害に対応するためには細かい粒度のデータが必要
- 過去に発生した問題を分析するためにはある程度長期間パフォーマンスデータの保存、削除するといった管理が必要
- パフォーマンス指標として一般的なOSのリソース情報やDB内部の動作を推測するためのDB統計情報に加えてDBのワークロードを端的に示す指標が必要

アナリシス

(分析)

- シンプルなパフォーマンス指標を元に、該当時間帯でパフォーマンス問題が発生しているか否かを端的に理解できる必要がある
- パフォーマンス問題が発生している場合、問題の原因をドリルダウンして究明できる必要がある
- チューニングした後の姿を予測し、チューニングのゴールを明確にしておく必要がある

チューニング

(最適化)

aws

パフォーマンスデータを収集、管理、分析するインフラ

コレクション (データ収集)

- 突発的なパフォーマンス障害に対応するためには細かい粒度のデータが必要
- 過去に発生した問題を分析するためにはある程度長期間パフォーマンスデータの保存、削除するといった管理が必要
- パフォーマンス指標として一般的なOSのリソース情報やDB内部の動作を推測するためのDB統計情報に加えてDBのワークロードを端的に示す指標が必要

アナリシス (分析)

- シンプルなパフォーマンス指標を元に、該当時間帯でパフォーマンス問題が発生しているか否かを端的に理解できる必要がある
- パフォーマンス問題が発生している場合、問題の原因をドリルダウンして究明できる必要がある
- チューニングした後の姿を予測し、チューニングのゴールを明確にしておく必要がある

チューニング (最適化)

- チューニング対象におけるボトルネックを排除し、事前に設定したチューニングのゴールに向けて最適化を実施
  - 推測ではないエビデンスに基づいたチューニング
  - データ収集 → 分析の流れがシステム化されていればチューニング後即座に効果の測定が可能

aws

パフォーマンスデータを収集、管理、分析するインフラ

パフォーマンスデータを平時から収集、管理し、有事には即座にそのデータを活用、分析できるインフラが必要

さらに、突発的なパフォーマンス障害を想定してリアルタイムに近いデータ取集、分析を可能にすることが望ましい

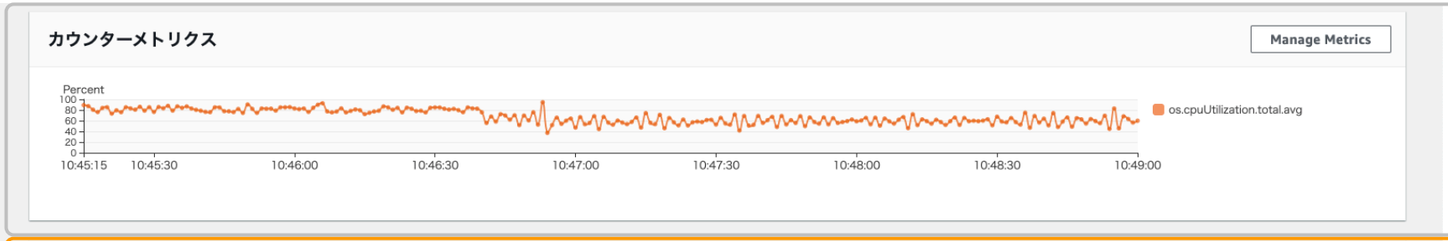
- チューニング対象におけるボトルネックを排除し、事前に設定したチューニングのゴールに向けて最適化を実施
  - 推測ではないエビデンスに基づいたチューニング
  - データ収集 → 分析の流れがシステム化されていればチューニング後即座に効果の測定が可能

aws

AWSでチューニングを支援する便利なサービス

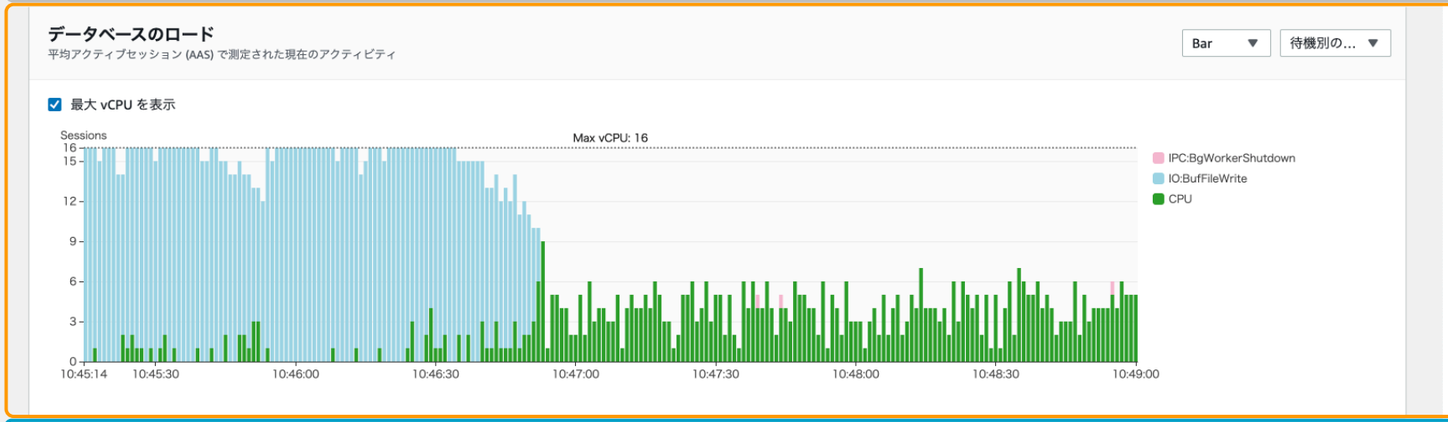
Amazon RDS Performance Insights

データベース内のパフォーマンスデータを蓄積してボトルネックを特定



カウンターメトリクス

- OSのリソース情報(CPU、Memoryなど)
- DBの統計情報(セッション数など)



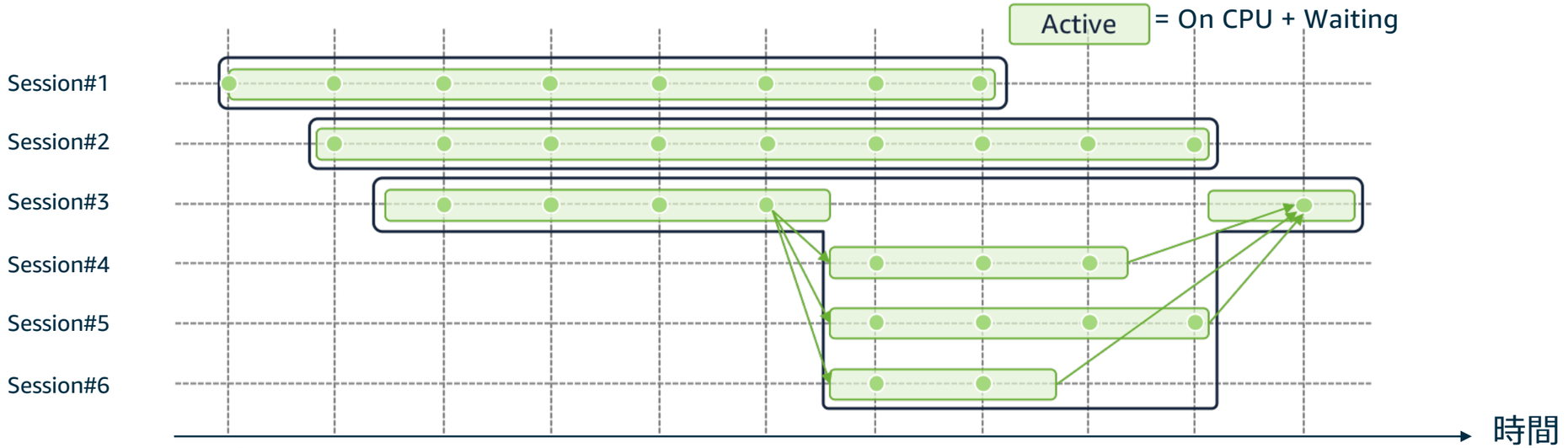
データベースのロード

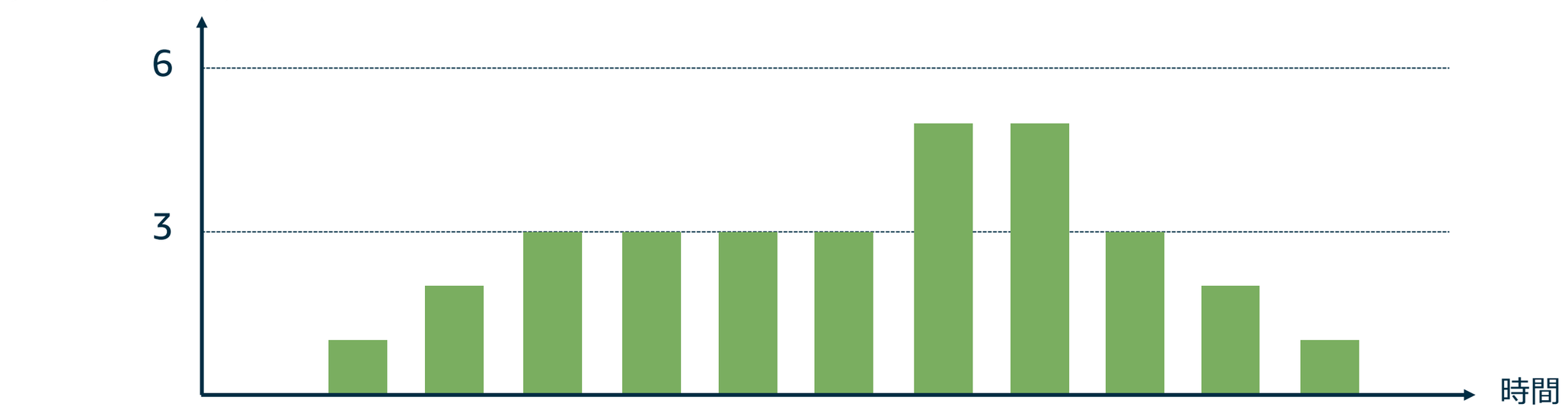
- 平均アクティブセッション数(AAS)
- CPU時間と待機イベント内訳
- RDS/Aurora全てのエンジンをサポート

ボトルネックの分析軸

- ボトルネックの原因の待機
- ボトルネックとなっているSQL
- 性能影響の高いホスト、ユーザー

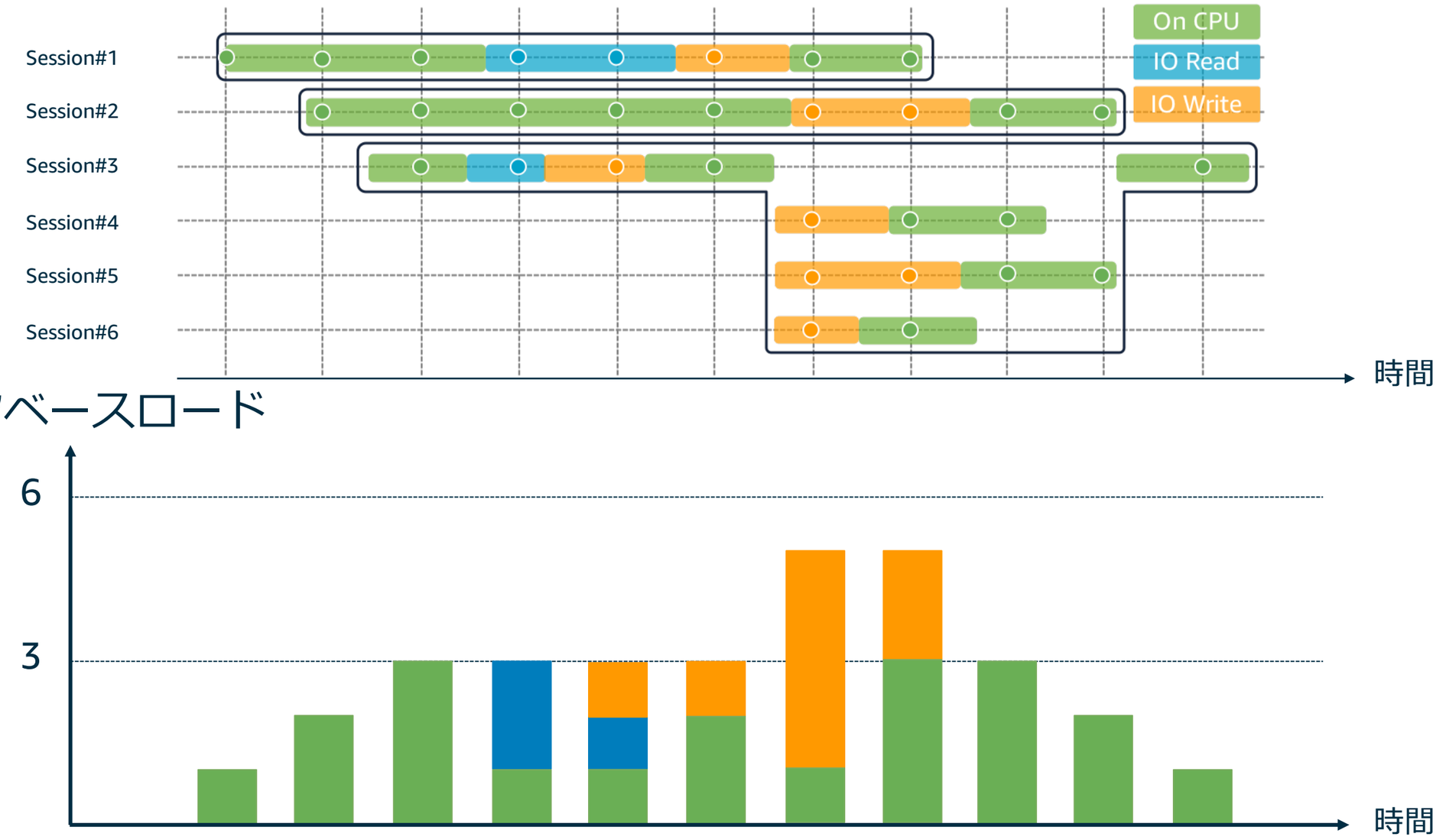
データベースロード (Average Active Sessions)





アクティブセッションの状態でドリルダウン

さらに”待機イベント”でドリルダウン



データベースロード (AAS)

- ✓ データベースロード(AAS) ≒ 0

  基本的にデータベースが使用されていない
  または、リソースが有効に使えていない

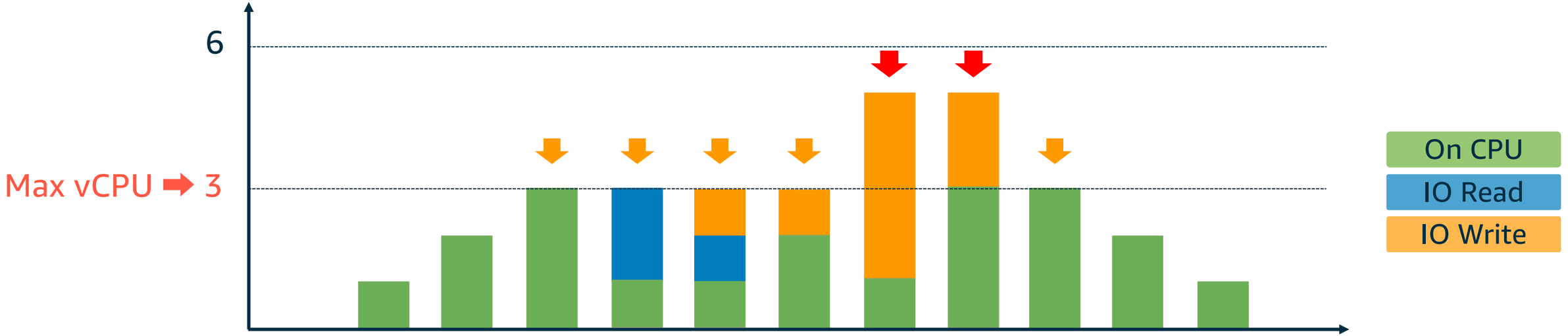
- ✓ データベースロード(AAS) >= # of vCPUs

  パフォーマンス問題の可能性がある

- ✓ データベースロード(AAS) >> # of vCPUs

  パフォーマンス問題

➡ パフォーマンス問題の有無は、データベースロードが
インスタンスの最大vCPUを超えているかどうかが基準



Performance Insightsの特徴 (データ収集、管理)

データ取得の粒度

毎秒アクティブなセッションの状態を収集しています。
これにより突発的なパフォーマンストラブル時でも粒度の高いデータを使用してボトルネックの分析が可能です。

データの保存期間

7日分は無料でパフォーマンスデータの収集、保存が可能になっています。一般的に7日分で十分ですが、さらに長い保存期間が必要な場合、わずかな追加コストで最大2年分のデータ保存が可能です。

ワークロードを端的に示す指標

Performance Insightsでは、アクティブなセッションの総数を重要な指標とします(データベースロード、AAS)。
また、待機情報も収集されているためボトルネックを分析する際に、ドリルダウンが可能です。

フルマネージ型のサービス

Performance Insightsの利用は、取集データを保存する際の暗号化キー、保存期間の指定など簡単な操作で始めることが可能です。
定期的に保存データを削除する等の管理作業に頭を悩ます必要はありません。

aws

Performance Insightsの特徴 (パフォーマンス分析)

データベースロード

単一の指標で、その時間帯でデータベースにチューニングが必要か否かを判別可能。

データベースロード >> vCPU

チューニングが必要な時間帯でボトルネックが何かドリルダウンして分析可能。

カウンターメトリクス

データベースロードを補足するOSのリソース情報(CPU、メモリー、I/Oリソースなど)やデータベースの統計情報(セッション数、トランザクション数など)を確認可能。

ボトルネックの分析

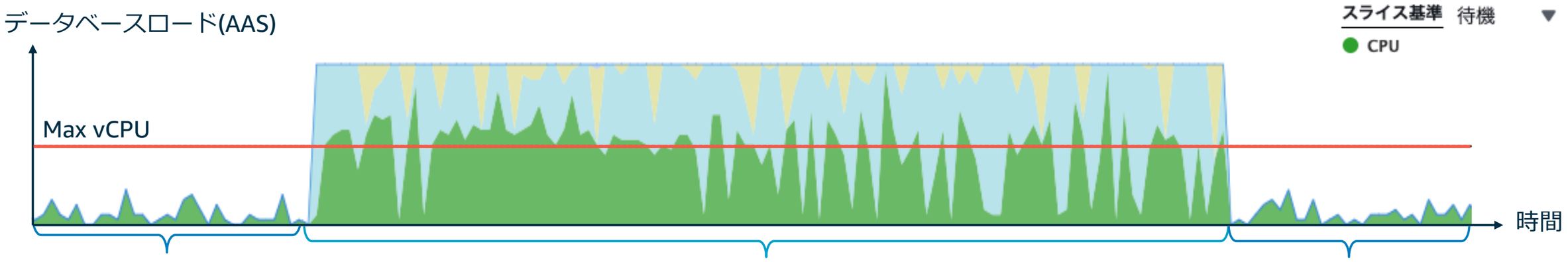
データベースロードからデータベース全体で問題のある時間帯を発見した後、その時間帯で何がボトルネックだったのか分析可能。
パフォーマンスにインパクトを与えた(要はチューニング対象となる)待機イベントは?
SQL文は? ホストは? DBユーザーは?

SQL単位の統計情報など

チューニング対象がSQL文の場合、SQL単位で統計情報(実行数、実行時間、論理/物理読み込み量など)があるとさらに精度の高い分析が可能になる。

aws

Performance Insightsの使用イメージ



AAS << vCPU

データベースの負荷は高くないと判断できます。

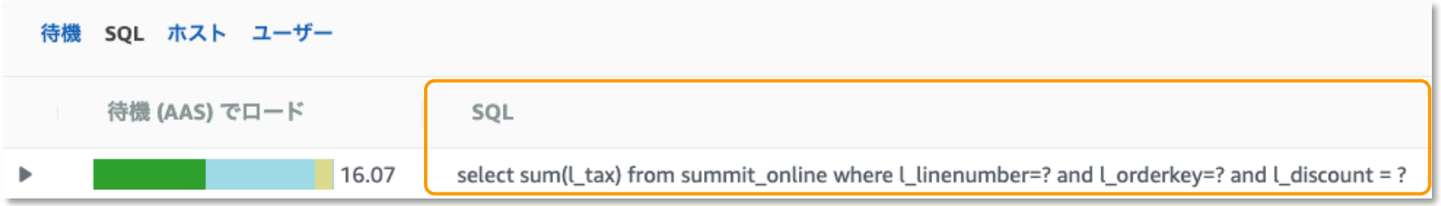
AAS >> vCPU

データベースの負荷は高いと判断できます。

また、高負荷の原因となっている待機イベントを判断できます。さらに、パフォーマンスにインパクトを与えているSQL文も同時に確認可能です。

AAS << vCPU

チューニングの効果をリアルタイムに確認することもできます。



クエリーの実行計画を管理する

aws

クエリーの実行計画を制御する一般的な方法

	メリット	デメリット
SQLヒント	• SQL文毎にヒント句で柔軟に実行プランの制御が可能	• SQL文にヒント句を入れることはSQL文(アプリケーション)の修正を伴う • 3rdパーティーパッケージなどヒント句を入れることが不可能な場合がある • より効率の良い実行プランが存在しても選択されない
SQL Plan Stability	• 個別にヒント句をSQL文に入れるのではなく、ヒント句とSQL文を関連づけることができる • SQL文(アプリケーション)の修正を伴わない	• より効率の良い実行プランが存在しても選択されない
SQL Plan管理	• 実行計画のバージョンを管理しどのバージョンを使うかは管理者の承認により制御可能 • 統計情報変更等により効率の良い実行プランが生成された場合は確認及び使用が可能 • SQL文(アプリケーション)の修正を伴わない • 個別に実行計画の修正が必要な場合は、ヒント句で実行計画を修正した後に承認が可能	• SQL Plan管理が可能なデータベースエンジンが限られる

aws

クエリーの実行計画を制御する一般的な方法

	RDS Oracle	RDS SQL Server	RDS PostgreSQL	Aurora PostgreSQL	RDS MySQL / MariaDB	Aurora MySQL
SQLヒント	ヒント句による実行計画の調整		pg_hint_plan拡張によるヒント句の使用		index hint、optimizer hintによる実行計画の調整	
SQL Plan Stability	STORED OUTLINEによる実行計画の固定	プランガイドによる実行計画の固定	pg_hint_plan拡張のヒントテーブルにより実行計画の固定		なし	
SQL Plan 管理	SQL Plan Managementによる実行計画の管理と固定	クエリストアによる実行計画の一部管理と固定	なし	Query Plan Management (QPM)による実行計画の管理と固定	なし	

aws

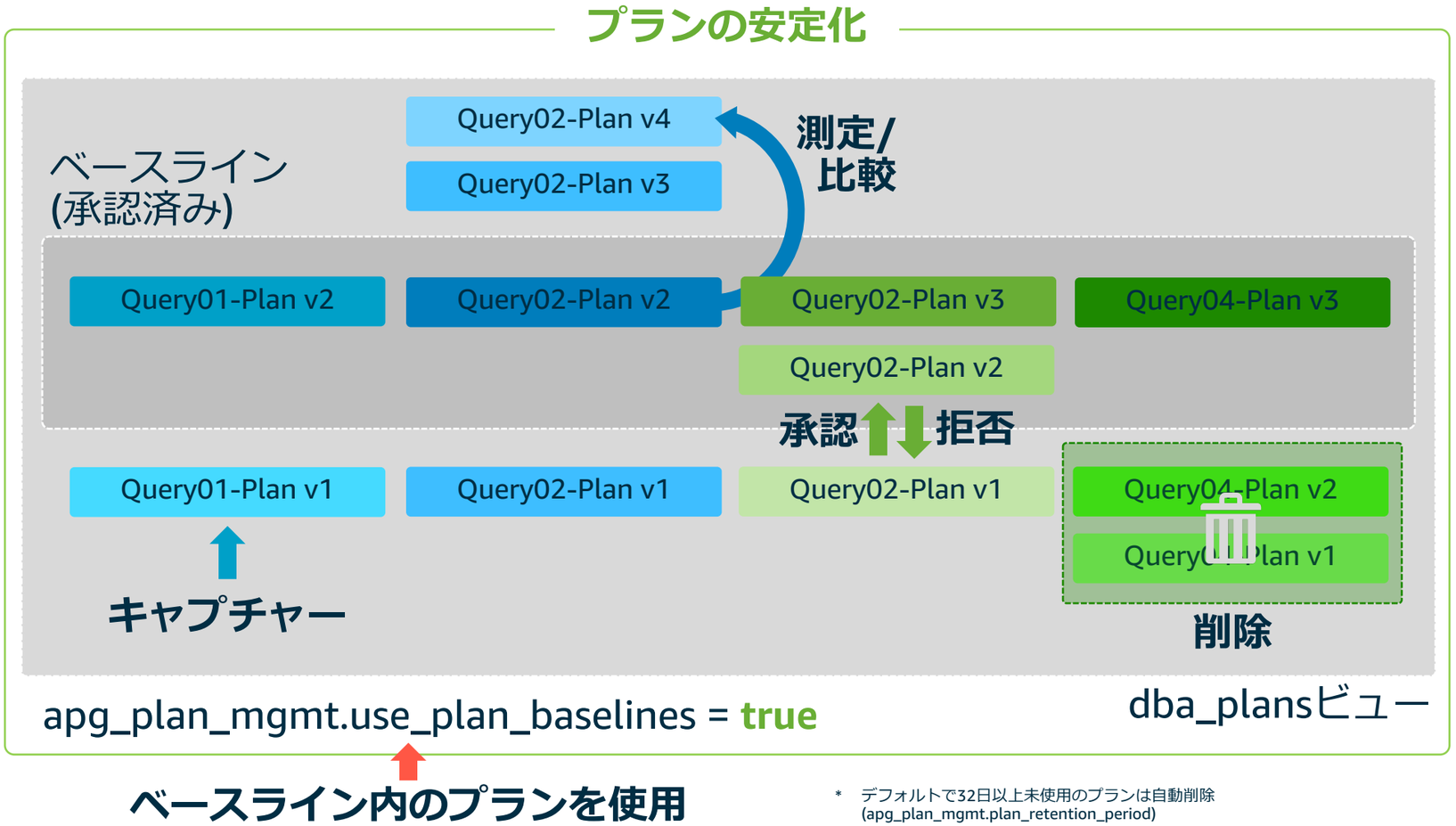
Aurora PostgreSQL: Query Plan Management

機能概要

- ✓ 手動/自動でプランのキャプチャー
- ✓ ベースライン内のプランを使用
- ✓ プランの承認/拒否
- ✓ プランの測定/比較
- ✓ pg_hint_planを使ったプランの修正
- ✓ プランの削除
- ✓ プランのエクスポート/インポート

サポートバージョン

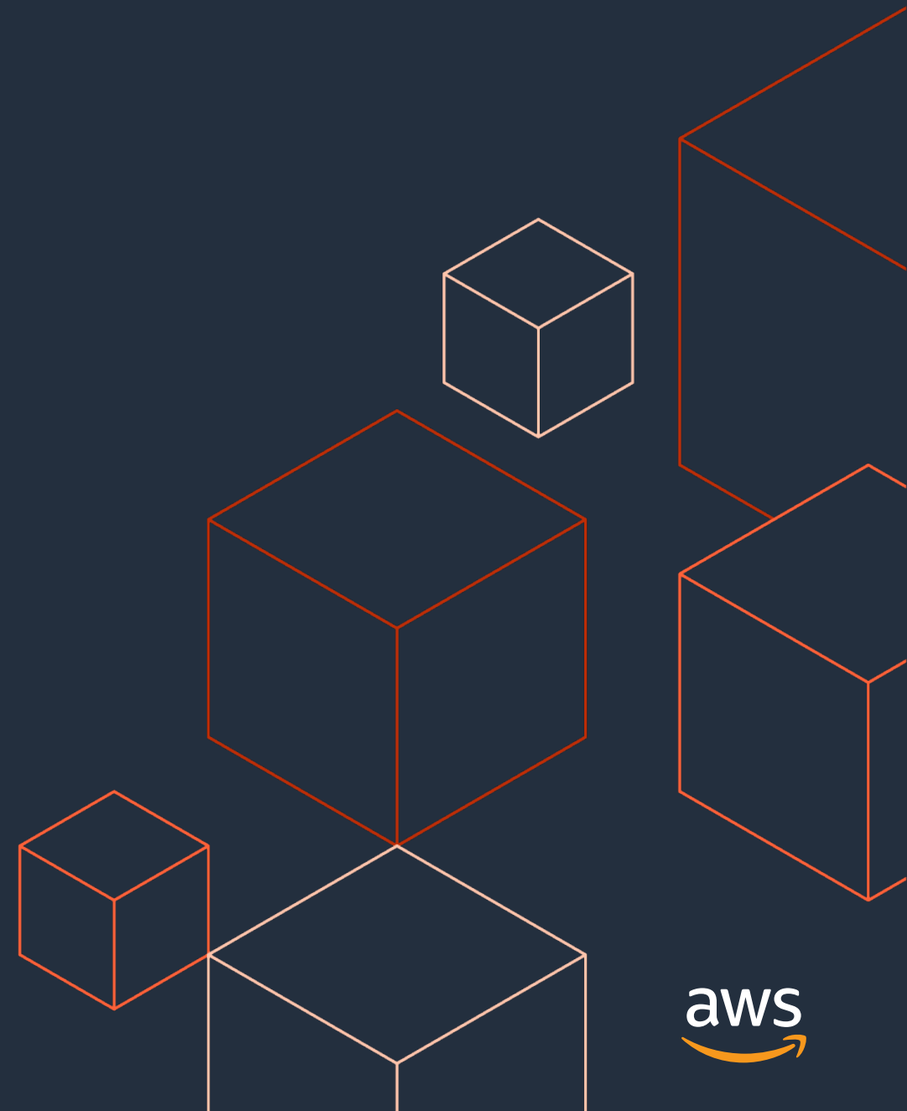
- ✓ Aurora PostgreSQL 2.1.0 (PostgreSQL 10.5互換)以降



* デフォルトで32日以上未使用のプランは自動削除 (apg_plan_mgmt.plan_retention_period)

* デフォルトで最大1,000個のプランをキャプチャー (apg_plan_mgmt.max_plans)

サンプルの
チューニングシナリオ



パフォーマンスダウンの状況

ケース:

- 今まで全くパフォーマンス的に問題がなかったクエリーが、
  ある日突然パフォーマンスダウンした

- 今回のシステムはとあるベンダーのパッケージ製品のため
  すぐにSQLの書き換えを行うことができない

今回はAurora PostgreSQLを前提にしています

aws

チューニングにおけるタスク

ケース:

- ボトルネックの特定
- チューニングにおけるボトルネックの排除
- アプリケーションのロジックを変更することも可能
(可能な限りアプリケーションを修正したくない)

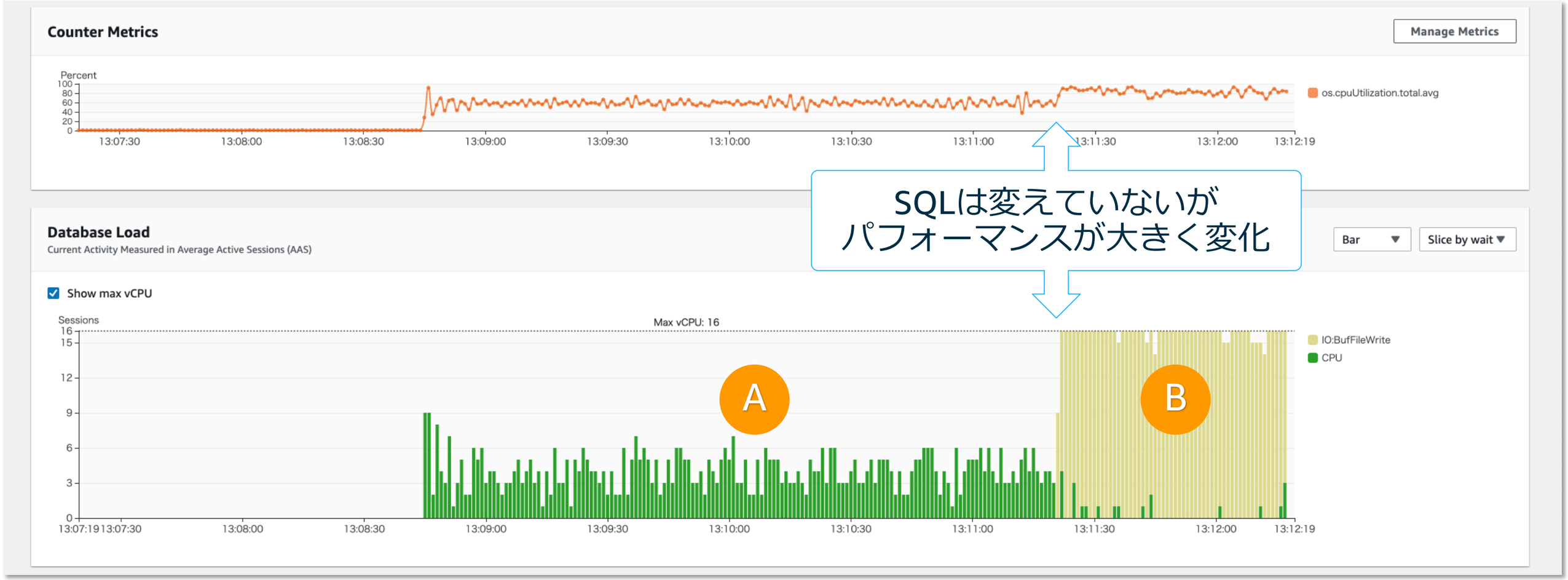
今回はAurora PostgreSQLを前提にしています

aws

パフォーマンスダウン
における実際のチューニング作業
をやってみよう

aws

Performance Insightsのダッシュボード
チューニング前



SQLの実行計画の確認(A)
チューニング前(SQL文と実行計画)

```
select * from sample_table_1 a JOIN sample_table_2 b on a.id2=b.id WHERE a.id > 0 and a.id < 210000;
```

A 通常時

```
 QUERY PLAN
------------------------------------------------------------------------------------------------------------
 Gather  (cost=1000.87..276434.93 rows=209225 width=126)
   Workers Planned: 2
   ->  Nested Loop  (cost=0.87..275434.93 rows=87177 width=126)
       ->  Parallel Index Scan using sample_table_1_idx on sample_table_1 a  (cost=0.43..7847.46 rows=87177 width=65)
             Index Cond: ((id > 0) AND (id < 210000))
       ->  Index Scan using sample_table_2_idx on sample_table_2 b  (cost=0.43..3.06 rows=1 width=61)
             Index Cond: (id = a.id2)
```


aws

SQLの実行計画の確認(B)
チューニング前(SQL文と実行計画)

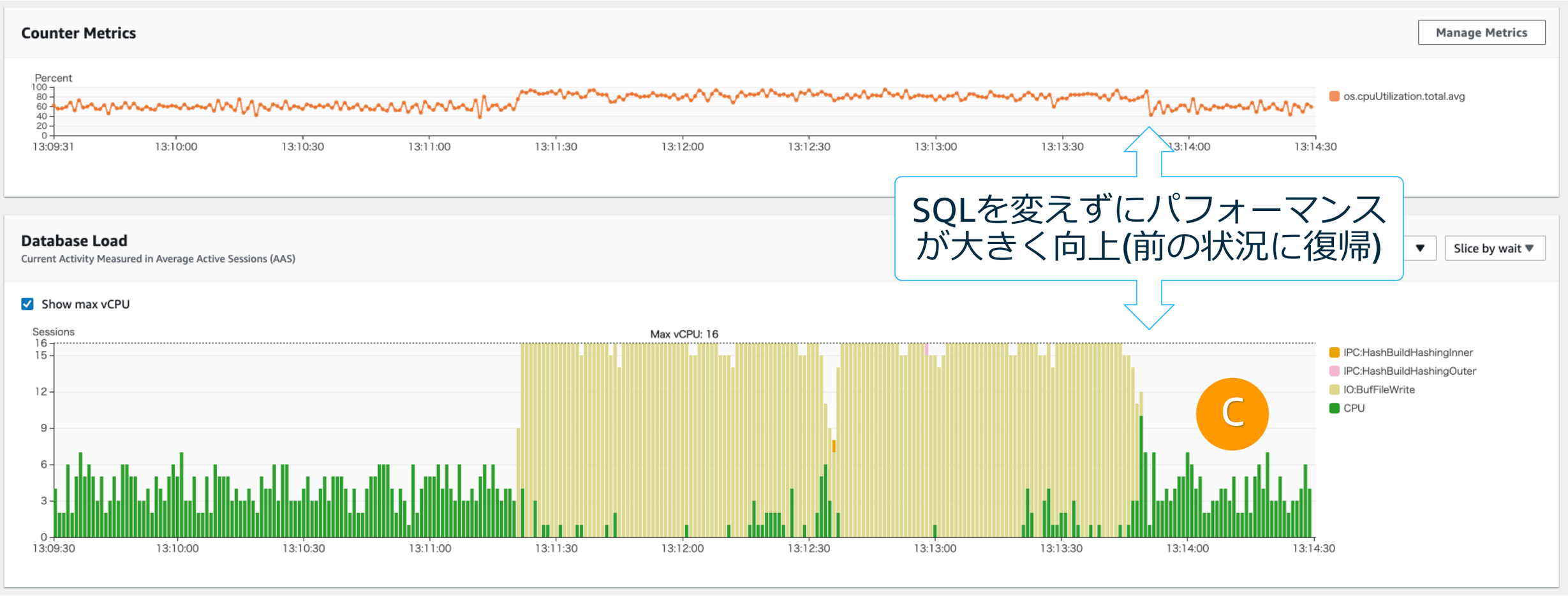
```
select * from sample_table_1 a JOIN sample_table_2 b on a.id2=b.id WHERE a.id > 0 and a.id < 210000;
```

B パフォーマンス劣化時

```
 QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
 Gather  (cost=10959.17..293715.58 rows=209225 width=126)
   Workers Planned: 2
   ->  Parallel Hash Join  (cost=9959.17..271793.08 rows=87177 width=126)
         Hash Cond: (b.id = a.id2)
         ->  Parallel Seq Scan on sample_table_2 b  (cost=0.00..155303.67 rows=4166667 width=61)
         ->  Parallel Hash  (cost=7847.46..7847.46 rows=87177 width=65)
             ->  Parallel Index Scan using sample_table_1_idx on sample_table_1 a  (cost=0.43..7847.46 rows=87177 width=65)
                   Index Cond: ((id > 0) AND (id < 210000))
```


aws

Performance Insightsのダッシュボード
チューニング後



チューニングの例

通常時はNested Loop結合が使われていたが、パフォーマンス劣化時はHash結合が使われるようになっている

↓

pg_hint_planのヒント句で実行計画を修正(SQL文の修正)

↓

今回はアプリケーション(SQL文)の書き換えが不可能

↓

Aurora PostgreSQLのQuery Plan Management(QPM)を使って、SQL文の修正なしに実行計画を修正 (Hash結合からNested Loop結合に戻す)

SQLの実行計画の確認(C)
チューニング後(SQL文と実行計画)

```
select * from sample_table_1 a JOIN sample_table_2 b on a.id2=b.id WHERE a.id > 0 and a.id < 210000;
```

C パフォーマンスチューニング後

アプリケーション(SQL文)の変更なしに、データベースが適切な実行計画を選択するようになっている。

```
 QUERY PLAN
-------------------------------------------------------------------------------------------
 Gather  (cost=1000.87..276434.93 rows=209225 width=126)
   Workers Planned: 2
   ->  Nested Loop  (cost=0.87..275434.93 rows=87177 width=126)
       ->  Parallel Index Scan using sample_table_1_idx on sample_table_1 a  (cost=0.43..7847.46 rows=87177 width=65)
             Index Cond: ((id > 0) AND (id < 210000))
       ->  Index Scan using sample_table_2_idx on sample_table_2 b  (cost=0.43..3.06 rows=1 width=61)
             Index Cond: (id = a.id2)
```


aws

パフォーマンスを安定化させる クエリー計画管理(QPM)を使ってみよう

Query Plan Managementを使用する流れ

クラスターのパラメータグループ
”rds.enable_plan_management” = 1

インスタンス/セッション単位
でのパラメーターを設定

プランのキャプチャー設定
apg_plan_mgmt.capture_plan_baselines

プランベースラインの使用有無
apg_plan_mgmt.use_plan_baselines

拡張のインストール
> create extension apg_plan_mgmt;

QPMで提供されるツール(関数)で定期的に
ベースラインをメンテナンス

最初にキャプチャーされたプランがベースラインに自動登録(Approved)される

自動キャプチャーされる設定の場合、デフォルトで2回以上実行されたプランがキャプチャーされる

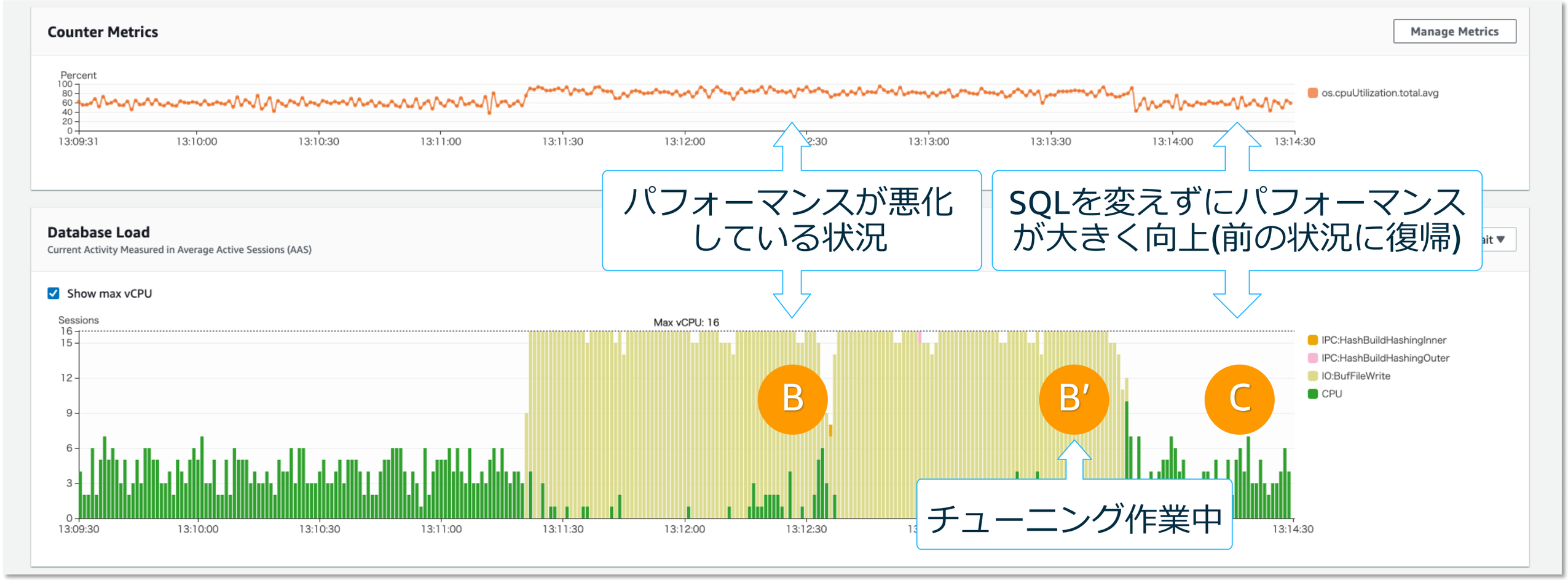
同一SQLで自分自身以外のプランがベースラインに存在し、自分自身のプランがキャプチャーされていない場合、新たにキャプチャーされるプランはベースライン外(Unapproved)となる

ベースライン外のプランを検査し、必要に応じてステータスを変更(Approved, Rejected, Preferred)
- apg_plan_mgmt.set_plan_status関数

各プランのクエリーの実行時間を実測し、高速になるものだけ承認、遅いものは拒否などが可能
- apg_plan_mgmt.evolve_plan_baselines関数

aws

チューニング前後の状況の確認



パフォーマンス劣化時のSQLの実行計画と実行時間を確認

```
EXPLAIN (ANALYZE,HASHES) select * from sample_table_1 a JOIN sample_table_2 b on a.id2=b.id WHERE a.id > 0 and a.id < 210000;

QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------
Gather  (cost=10959.17..293715.58 rows=209225 width=126) (actual time=1654.901..2108.682 rows=209999 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Hash Join  (cost=9959.17..271793.08 rows=87177 width=126) (actual time=1654.232..1931.045 rows=70000 loops=3)
         Hash Cond: (b.id = a.id2)
         ->  Parallel Seq Scan on sample_table_2 b  (cost=0.00..155303.67 rows=4166667 width=61) (actual …)
         ->  Parallel Hash  (cost=7847.46..7847.46 rows=87177 width=65) (actual time=27.412..27.412 rows=70000 loops=3)
               Buckets: 65536  Batches: 8  Memory Usage: 3232kB
               ->  Parallel Index Scan using sample_table_1_idx on sample_table_1 a  (cost=0.43..7847.46 rows=87177 width=65) (…)
                     Index Cond: ((id > 0) AND (id < 210000))
 Planning Time: 0.230 ms
 Execution Time: 2116.716 ms
 Plan Hash: -306787140
```

実行計画としてHash結合を選択し、実行時間としては2,116 ms程度

aws

パフォーマンス劣化時のSQLの実行計画と実行時間を確認

```
-[ RECORD 1 ]--------+-----------------------------------------------------------------------------------
sql_hash             | 699820183
plan_hash            | -306787140
status               | Approved
enabled              | t
sql_text             | select * from sample_table_1 a JOIN sample_table_2 b on a.id2=b.id WHERE a.id > 0 and a.id < 210000;
```

apg_plan_mgmt.dba_plansビューから現在、ベースラインに格納されている実行計画を確認

この時点では、1つの実行計画(Hash結合)のみがベースラインに格納されている

aws

チューニング後のSQLをベースラインに取り込む

```
SET apg_plan_mgmt.capture_plan_baselines = manual

/*+ NestLoop(a b) */ explain select *
                           from sample_table_1 a JOIN sample_table_2 b on a.id2=b.id
                           WHERE a.id > 0 and a.id < 210000;

SET apg_plan_mgmt.capture_plan_baselines = false
```

apg_plan_mgmt.capture_plan_baselinesを手動に設定(自動も可能)し、pg_hint_planのヒント句でチューニング対象のSQLに対して別バージョンの実行計画をベースラインに取り込む

aws

QPMのベースラインを確認(B')

```
-[ RECORD 1 ]--------+-----------------------------------------------------------------------------------------
sql_hash             | 699820183
plan_hash            | -306787140
status               | Approved
enabled              | t
sql_text             | select * from sample_table_1 a JOIN sample_table_2 b on a.id2=b.id WHERE a.id > 0 and a.id < 210000;
-[ RECORD 2 ]--------+-----------------------------------------------------------------------------------------
sql_hash             | 699820183
plan_hash            | 1809927363
status               | Unapproved
enabled              | t
sql_text             | select * from sample_table_1 a JOIN sample_table_2 b on a.id2=b.id WHERE a.id > 0 and a.id < 210000;
```

Hash結合の実行計画

Nested Loop結合の実行計画

チューニング後の実行計画は、Unapproved(未承認)としてベースラインに取り込まれていることが分かる

aws

チューニング後のSQLを承認

```
SELECT apg_plan_mgmt.set_plan_status(699820183, 1809927363, 'Approved');

SELECT apg_plan_mgmt.set_plan_status(699820183, -306787140,  Unapproved');
```

チューニング後のNested Loopの実行計画のステータスをApproved(承認)に変更し、合わせてチューニング前のHash結合の実行計画をUnapproved(未承認)に変更

aws

QPMのベースラインを確認(B')

```
-[ RECORD 1 ]--------+-----------------------------------------------------------------------------------------
sql_hash             | 699820183
plan_hash            | -306787140
status               | Unapproved
enabled              | t
sql_text             | select * from sample_table_1 a JOIN sample_table_2 b on a.id2=b.id WHERE a.id > 0 and a.id < 210000;
-[ RECORD 2 ]--------+-----------------------------------------------------------------------------------------
sql_hash             | 699820183
plan_hash            | 1809927363
status               | Approved
enabled              | t
sql_text             | select * from sample_table_1 a JOIN sample_table_2 b on a.id2=b.id WHERE a.id > 0 and a.id < 210000;
```

Hash結合の実行計画

Nested Loop結合の実行計画

チューニング後の実行計画が、Approved(承認)に変更され、チューニング前の実行計画がUnapproved(未承認)に変更されていることが分かる

aws

QPMのベースライン承認後のSQLの実行計画と実行時間を確認

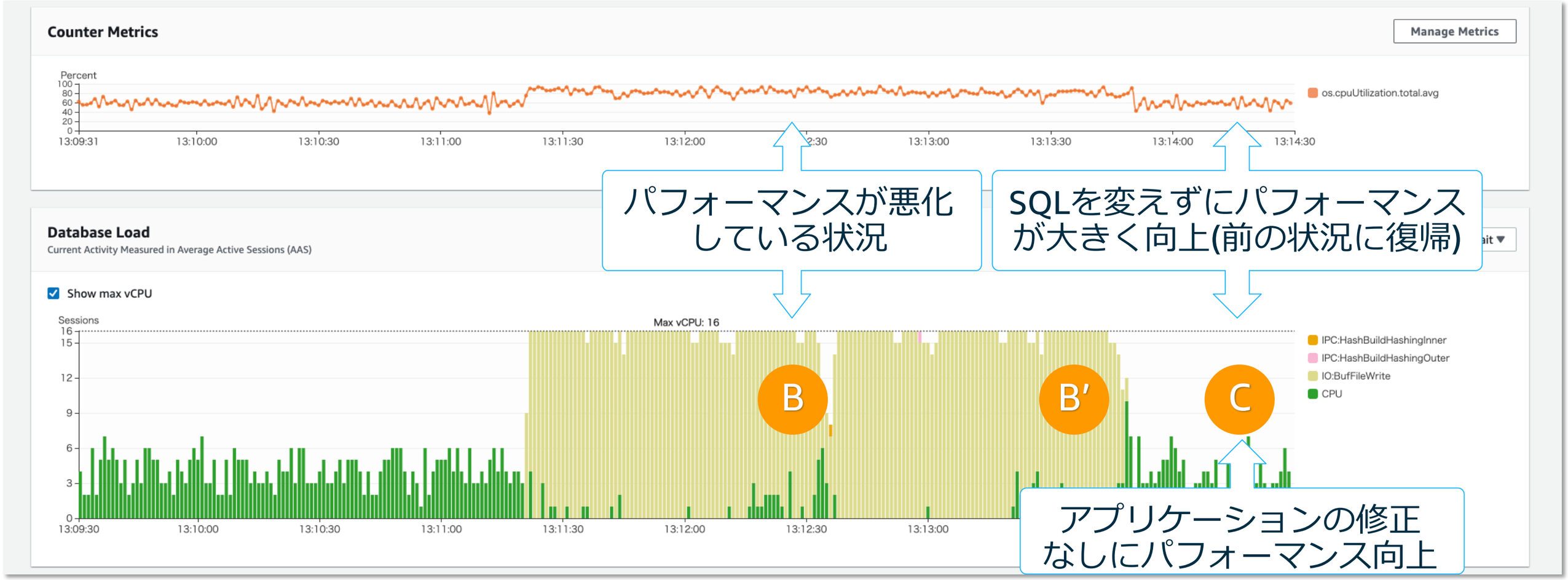
```
SET apg_plan_mgmt.use_plan_baselines = true;
EXPLAIN ANALYZE select * from sample_table_1 a JOIN sample_table_2 b on a.id2=b.id WHERE a.id > 0 and a.id < 210000;

QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------
 Gather  (cost=1000.87..297357.43 rows=209225 width=126) (actual time=0.263..236.801 rows=209999 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Nested Loop  (cost=0.87..275434.93 rows=87177 width=126) (actual time=0.050..203.592 rows=70000 loops=3)
         ->  Parallel Index Scan using sample_table_1_idx on sample_table_1 a  (cost=0.43..7847.46 rows=87177 width=65) (…)
               Index Cond: ((id > 0) AND (id < 210000))
         ->  Index Scan using sample_table_2_idx on sample_table_2 b  (cost=0.43..3.06 rows=1 width=61) (…)
               Index Cond: (id = a.id2)
 Planning Time: 0.502 ms
 Execution Time: 245.751 ms
 Note: An Approved plan was used instead of the minimum cost plan.
 SQL Hash: 699820183, Plan Hash: 1809927363, Minimum Cost Plan Hash: -306787140
```

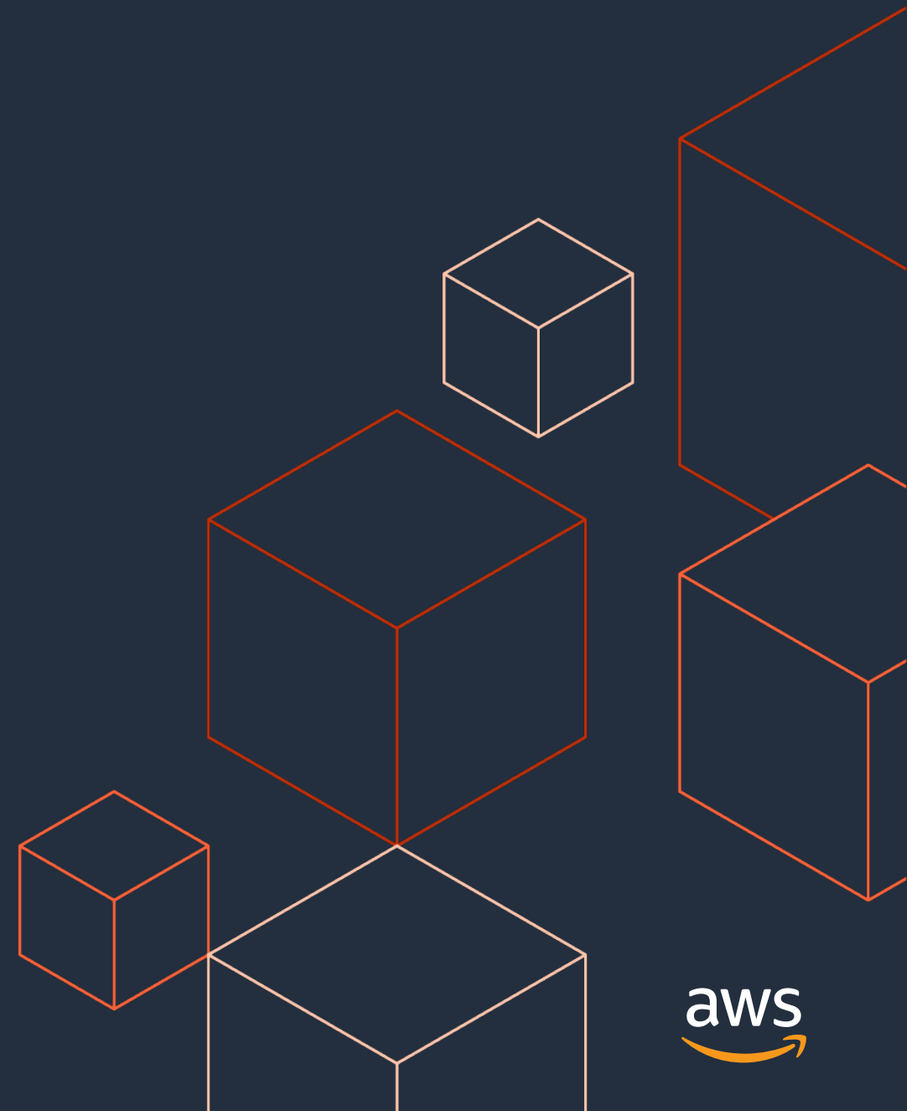
実行計画としてNested Loop結合を選択し、実行時間としては245 ms (チューニング前は2,116 ms)に改善している

aws

チューニング後の状況の確認



まとめ

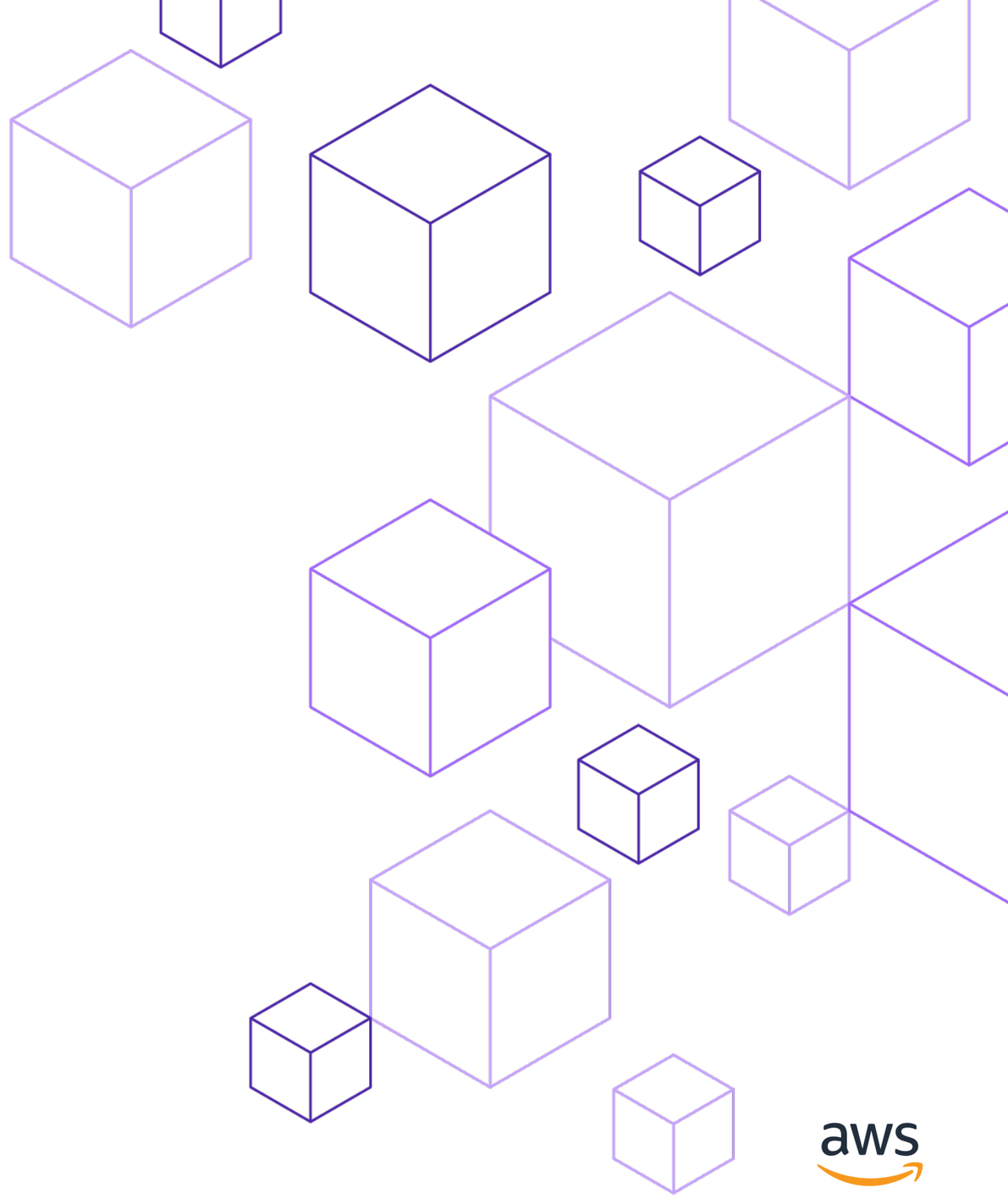


まとめ

- チューニングサイクルの理解
- チューニングに必要なデータやインフラの理解
- チューニングの作業効率を高めるPerformance Insights
- 適切にクエリーの実行計画を管理するためのAurora PostgreSQL Query Plan Management

aws

Q&A



Thank you!