パフォーマンスの問題を回避するための Amazon RDS および Amazon Aurora の PostgreSQL データベースに対するメンテナンスアクティビティ

AWS 規範ガイダンス

AWS 規範ガイダンス: パフォーマンスの問題を回避するための Amazon RDS および Amazon Aurora の PostgreSQL データベースに対するメンテナンスアクティビティ

Table of Contents

序章 1
ターゲットを絞ったビジネス成果 1
マルチバージョン同時実行制御 (MVCC) 1
テーブルのバキューム処理と分析の自動実行 3
自動バキュームのメモリ関連パラメータ 5
autovacuum パラメータの調整 5
クラスターまたはインスタンスレベル 6
テーブルレベル 6
テーブルレベルで積極的な自動バキューム設定を使用する 6
利点と制限事項 7
テーブルのバキューム処理と分析の手動による実行 9
バキューム操作とクリーンアップ操作を並行して実行する 9
VACUUM FULL を使用してテーブル全体を書き換える 14
pg_repack を使用した肥大化の削除 15
インデックスの再構築 17
新しいインデックスを作成する 21
インデックスの再構築 21
例 22
例: autovacuum と VACUUM FULL を使用したスペースの再利用 24
リソース 29
ドキュメント履歴 30
..... xxxi

パフォーマンスの問題を回避するための Amazon RDS および Amazon Aurora の PostgreSQL データベースに対するメンテナンスアクティビティ

Amazon Web Services (AWS)、Anuradha Chintha、Rajesh Madiwale、Srinivas Potlachervoo

2025 年 8 月 (ドキュメント履歴)

Amazon Aurora PostgreSQL 互換エディションデータベースおよび PostgreSQL 用の Amazon Relational Database Service (Amazon RDS) は、完全に管理された PostgreSQL データベース用リレーショナルデータベースサービスです。これらのマネージドサービスにより、データベース管理者は多くのメンテナンスおよび管理タスクから解放されます。ただし、VACUUM などの一部のメンテナンスタスクでは、データベースの使用状況に基づいて綿密なモニタリングと設定が必要となります。このガイドでは、Amazon RDS および Aurora における PostgreSQL のメンテナンスアクティビティについて説明します。

ターゲットを絞ったビジネス成果

データベースのパフォーマンスは、ビジネス成功の基礎となる重要な指標です。Aurora PostgreSQL 互換データベースおよび Amazon RDS for PostgreSQL データベースでメンテナンスアクティビティを実行することには、次の利点があります。

- 最適なクエリパフォーマンスを達成できる
- 将来のトランザクションで再利用できるように肥大した領域を解放する
- トランザクションの循環を防止する
- オプティマイザにより適切なプランを生成できる
- インデックスの適切な使用を確保する

マルチバージョン同時実行制御 (MVCC)

PostgreSQL データベースのメンテナンスでは、PostgreSQL のメカニズムであるマルチバージョン同時実行制御 (MVCC) について理解する必要があります。データベースで複数のトランザクションが同時に処理される場合、MVCC により ACID トランザクション (原子性、一貫性、分離性、持続性) のうちの 2 つの特性である原子性と分離性が維持されるようにします。MVCC では、書き込み操

作のたびに新しいバージョンのデータが生成され、以前のバージョンが保存されます。リーダーとライターが互いにブロックしあうことはありません。トランザクションでデータを読み取ったら、システムでトランザクションを分離するためにいずれかのバージョンが選択されます。PostgreSQL と一部のリレーショナルデータベースでは、スナップショット分離 (SI) と呼ばれる MVCC の適応が使用されます。例えば、Oracle ではロールバックセグメントを使用して SI が実装されています。書き込み操作中、Oracle では古いバージョンのデータをロールバックセグメントに書き込み、データ領域を新しいバージョンで上書きします。PostgreSQL データベースには、可視性チェックルールを用いて SI が実装され、各バージョンが評価されます。新しいデータがテーブルページに配置されると、PostgreSQL はこれらのルールを使用して、読み取り操作に適したバージョンのデータを選択します。

テーブル行のデータを変更すると、PostgreSQL は MVCC を使用して複数のバージョンの行を維持します。テーブルの UPDATE および DELETE 操作中、データベースには、データの一貫したビューを必要とする可能性のある他の実行中のトランザクションの古いバージョンの行が保持されます。これらの古いバージョンはデッド行 (タプル) と呼ばれます。デッドタプルが集合することで肥大化します。データベースの肥大化は、クエリプランの生成不足、クエリパフォーマンスの低下、古いバージョンを保存することによるディスク容量の使用増大など、多くの問題を引き起こす可能性があります。

肥大化を削除し、データベースを正常な状態に保つには、定期的なメンテナンスが必要です。これには、以下のセクションで説明するアクティビティが含まれます。

- テーブルのバキューム処理と分析の自動実行
- テーブルのバキューム処理と分析の手動による実行
- pg_repack を使用した肥大化の削除
- インデックスの再構築

テーブルのバキューム処理と分析の自動実行

自動バキュームは、デッドタプルのバキューム処理 (クリーンアップ)、ストレージの再利用、統計の収集を自動的に行うデーモンです (つまり、バックグラウンドで実行されます)。データベース内で大きくなったテーブルをチェックし、肥大化を削除してスペースを再利用します。データベースのテーブルとインデックスをモニタリングし、特定の更新または削除オペレーションのしきい値に達したらバキュームジョブに追加します。

自動バキュームは PostgreSQL `VACUUM` および `ANALYZE` コマンドを自動化することでバキューム処理を管理します。`VACUUM` を使用してテーブルの肥大化を削除し、スペースを再利用する一方で、`ANALYZE` を使用してオプティマイザが効果的なプランを作成できるよう統計を更新します。また、`VACUUM` はバキュームフリーズと呼ばれる主要なタスクを実行して、データベース内のトランザクション ID の循環に関する問題を回避します。データベースで更新されたすべての行は、PostgreSQL トランザクション制御メカニズムからトランザクション ID を受け取ります。これらの ID によって、他の同時トランザクションに対する行の可視性が制御されます。トランザクション ID は 32 ビットの数値です。目に見える過去として、常に 20 億個の ID が保持されます。残りの ID (約 22 億個) は、今後実施されるトランザクション用に保持され、現在のトランザクションでは非表示になります。PostgreSQL では、新しいトランザクションの作成時にトランザクションが循環したり、既存の古い行が非表示になったりしないように、古い行のクリーニングとフリーズが必要になることがあります。詳細については、PostgreSQL ドキュメントの「トランザクション ID 循環の失敗を防ぐ」を参照してください。

自動バキュームが推奨され、デフォルトで有効になっています。パラメータには、以下が含まれます。

パラメータ	説明	Amazon RDS のデフォルト	Aurora のデフォルト
`autovacuum_vacuum_threshold`	自動バキュームでバキューム処理する前にテーブルで実行する必要があるタプルの更新または削除オペレーションの最小数。	50 オペレーション	50 オペレーション

autovacuum_analyze_threshold	自動バキュームで分析を実行する前にテーブルで実行する必要があるタプルの挿入、更新、または削除の最小数。	50 オペレーション	50 オペレーション
autovacuum_vacuum_scale_factor	自動バキュームでバキューム処理する前にテーブルで変更する必要があるタプルの割合。	0.1	0.1
autovacuum_analyze_scale_factor	自動バキュームで分析を実行する前にテーブルで変更する必要があるタプルの割合。	0.05	0.05
autovacuum_freeze_max_age	トランザクション ID の循環に関する問題を回避するためにテーブルがバキューム処理される前の、フリーズ ID の最大有効期間。	200,000,000 件のトランザクション	200,000,000 件のトランザクション

自動バキュームでは、次のように特定のしきい値式に基づいて処理するテーブルのリストが作成されます。

- テーブルで VACUUM を実行するためのしきい値:

```
vacuum threshold = autovacuum_vacuum_threshold + (autovacuum_vacuum_scale_factor *
 Total row count of table)
```

- テーブルで ANALYZE を実行するためのしきい値:

```
analyze threshold = autovacuum_analyze_threshold + (autovacuum_analyze_scale_factor *
 Total row count of table)
```

中小規模のテーブルの場合、デフォルト値で十分です。ただし、データ変更が頻繁に発生するような大きなテーブルでは、デッドタプルの数が多くなります。この場合、自動バキュームによってメンテナンスのためにテーブルが頻繁に処理され、大きなテーブルの処理が終了するまで他のテーブルのメンテナンスが遅延または無視される可能性があります。これを回避するには、次のセクションで説明する autovacuum パラメータを調整できます。

自動バキュームのメモリ関連パラメータ

`autovacuum_max_workers`

同時に実行できる自動バキュームプロセス (自動バキュームランチャーを除く) の最大数を指定します。このパラメータは、サーバーを起動するときにのみ設定できます。自動バキュームプロセスが大きなテーブルでビジー状態の場合、このパラメータを使用して他のテーブルのクリーンアップを実行できます。

`maintenance_work_mem`

メンテナンスオペレーション (`VACUUM`、`CREATE INDEX`、`ALTER` など) で使用されるメモリの最大量を指定します。Amazon RDS と Aurora では、式 `GREATEST({DBInstanceClassMemory/63963136*1024},65536)` を使用してインスタンスクラスに基づきメモリが割り当てられます。自動バキュームを実行する場合、その計算値を割り当てることができるのは最大で `autovacuum_max_workers` 回までとなるため、値を高く設定しすぎないように注意してください。これを制御するには、`autovacuum_work_mem` を個別に設定できます。

`autovacuum_work_mem`

各自動バキュームワーカープロセスで使用するメモリの最大量を設定します。このパラメータのデフォルトは -1 です。これは、代わりに `maintenance_work_mem` の値を使用する必要があることを示します。

autovacuum メモリパラメータの詳細については、Amazon RDS ドキュメントの「Allocating memory for autovacuum」を参照してください。

autovacuum パラメータの調整

ユーザーは、更新および削除オペレーションに応じて autovacuum パラメータを調整する必要が生じる場合があります。次のパラメータの設定を、テーブル、インスタンス、またはクラスターレベルで実行できます。

クラスターまたはインスタンスレベル

例として、継続的データ操作言語 (DML) オペレーションが予想される銀行データベースを見てみましょう。データベースの正常性を維持するには、Aurora のクラスターレベルと Amazon RDS のインスタンスレベルで autovacuum パラメータを調整し、リーダーにも同じパラメータグループを適用する必要があります。フェイルオーバーの場合は、同じパラメータを新しいライターに適用する必要があります。

テーブルレベル

例えば、orders という 1 つのテーブルで継続的な DML オペレーションが予想されるフードデリバリー用のデータベースでは、次のコマンドを使用して autovacuum_analyze_threshold パラメータをテーブルレベルで調整することを検討する必要があります。

```
ALTER TABLE <table_name> SET (autovacuum_analyze_threshold = <threshold rows>)
```

テーブルレベルで積極的な自動バキューム設定を使用する

継続的な更新および削除オペレーションを行うサンプルの orders テーブルは、デフォルトの自動バキューム設定によりバキューム処理の候補となります。これにより、プランの生成が不正となり、クエリが遅くなります。肥大化の削除と統計の更新には、テーブルレベルの積極的な自動バキューム設定が必要です。

設定を決定するには、このテーブルで実行されているクエリの期間を追跡し、プランの変更につながる DML オペレーションの割合を特定します。pg_stat_user_tables ビューは、挿入、更新、削除オペレーションを追跡するのに役立ちます。

例:

orders テーブルの 5% が変更されるたびに、オプティマイザで不正なプランが生成されると仮定します。この場合、次のようにスケール係数のしきい値を 2% に変更する必要があります。

```
ALTER TABLE orders SET (autovacuum_vacuum_scale_factor = 0.02)
```

> **Tip**
>
> リソースの大量消費を避けるため、積極的な自動バキュームの設定では慎重に選択を行ってください。

詳細については次を参照してください:

- Amazon RDS for PostgreSQL 環境での autovacuum について (AWS ブログ記事)
- 自動バキューム処理 (PostgreSQL ドキュメント)
- Amazon RDS と Amazon Aurora での PostgreSQL パラメータのチューニング (AWS 規範ガイダンス)

自動バキュームが効果的に機能することを確認するには、デッド行、ディスク使用量、および最後に自動バキュームまたは ANALYZE を実行した時間を定期的にモニタリングします。pg_stat_all_tables ビューには、各テーブル (relname) に関する情報と、テーブルにあるデッドタプルの数 (n_dead_tup) が表示されます。

各テーブル、特に頻繁に更新されるテーブルのデッドタプルの数をモニタリングすることで、自動バキュームプロセスがデッドタプルを定期的に削除しているかどうかを判断し、ディスク容量を再利用してパフォーマンスを向上させることができます。次のクエリを使用して、デッドタプルの数と、テーブルで最後に自動バキュームが実行された日時を確認できます。

```
SELECT
relname AS TableName,n_live_tup AS LiveTuples,n_dead_tup AS DeadTuples,
last_autovacuum AS Autovacuum,last_autoanalyze AS Autoanalyze_FROM
pg_stat_user_tables;
```

利点と制限事項

自動バキュームには、次のような利点があります。

- テーブルで肥大化が自動的に削除されます。
- トランザクション ID の循環が防止されます。
- データベース統計を最新の状態に保ちます。

機能制限:

- クエリで並列処理を使用する場合、ワーカープロセスの数が自動バキュームの実行に十分ではない可能性があります。
- 自動バキュームがピーク時に実行されると、リソースの使用率が増加する可能性があります。この問題を処理するには、パラメータを調整する必要があります。

- テーブルページが別のセッションで占有されている場合、自動バキュームでそれらのページをスキップすることがあります。
- 自動バキュームは一時テーブルにアクセスできません。

テーブルのバキューム処理と分析の手動による実行

データベースが自動バキュームプロセスによってバキューム処理されている場合は、データベース全体で手動によるバキューム処理をあまり頻繁に実行しないようにすることをお勧めします。手動バキュームを行うことで不要な I/O 負荷や CPU スパイクが発生し、デッドタプルの削除に失敗する可能性があります。デッドタプルに対してライブタプルの比率が低い場合や、自動バキューム間のギャップが長い場合など、本当に必要な場合にのみ、テーブルごとに手動バキューム処理を実行してください。さらに、ユーザーアクティビティが最小限の場合は、手動によるバキューム処理を実行する必要があります。

また、自動バキュームではテーブルの統計が最新の状態に保たれます。ANALYZE コマンドを手動で実行すると、これらの統計は更新されるのではなく再構築されます。通常の自動バキュームプロセスによって既に更新されている統計を再構築すると、システムリソースの使用率が増加する可能性があります。

以下のシナリオでは、VACUUM コマンドと ANALYZE コマンドを手動で実行することをお勧めします。

- 混雑したテーブルのオフピーク時間。この場合、自動バキュームでは不十分な可能性があります。
- ターゲットテーブルにデータを一括ロードした直後。この場合、ANALYZE を手動で実行すると統計が完全に再構築されます。これは、自動バキュームの開始を待つよりも適切なオプションです。
- 一時テーブルをバキューム処理する場合 (自動バキュームは一時テーブルにアクセスできません)。

同時データベースアクティビティで VACUUM および ANALYZE コマンドを実行するときの I/O への影響を軽減するには、vacuum_cost_delay パラメータを使用できます。多くの場合、VACUUM や ANALYZE などのメンテナンスコマンドはすぐに終了する必要はありません。ただし、これらのコマンドが他のデータベース操作を実行するシステムの機能を妨げることがあってはなりません。これを防ぐには、vacuum_cost_delay パラメータを使用してコストベースの真空遅延を有効にできます。このパラメータは、手動で発行された VACUUM コマンドではデフォルトで無効になっています。有効にするには、ゼロ以外の値に設定してください。

バキューム操作とクリーンアップ操作を並行して実行する

VACUUM コマンドの PARALLEL オプションでは、インデックスバキュームフェーズとインデックスクリーンアップフェーズに並列ワーカーを使用しますが、これらはデフォルトで無効になってい

ます。並列ワーカーの数 (並列処理の程度) は、テーブル内のインデックスの数によって決定され、ユーザーが指定できます。整数引数を使用せずに VACUUM 並列オペレーションを実行している場合、並列処理の程度はテーブル内のインデックスの数に基づいて計算されます。

次のパラメータは、Amazon RDS for PostgreSQL および Aurora PostgreSQL 互換で並列バキューム処理を設定するのに役立ちます。

- max_worker_processes で、同時ワーカープロセスの最大数を設定します。
- min_parallel_index_scan_size では、並列スキャンを考慮するためにスキャンする必要があるインデックスデータの最小量を設定します。
- max_parallel_maintenance_workers は、単一のユーティリティコマンドで開始できる並列ワーカーの最大数を設定します。

> **Note**
>
> PARALLEL オプションは、バキューム処理の用途にのみ使用されます。ANALYZE コマンドには影響しません。

次の例は、データベースで VACUUM および ANALYZE を手動で使用する場合のデータベースの動作を図示しています。

以下は、自動バキュームが無効になっているテーブルの例です (説明のみを目的としています。自動バキュームを無効にすることはお勧めしません)。

```
create table t1 ( a int, b int, c int );
alter table t1 set (autovacuum_enabled=false);
```

```
apgl=> \d+ t1
Table "public.t1"
Column | Type | Collation | Nullable | Default | Storage | Stats target | Description
--------+---------+-----------+----------+---------+---------+--------------
+-------------
a | integer | | | | plain | |
b | integer | | | | plain | |
c | integer | | | | plain | |
Access method: heap
Options: autovacuum_enabled=false
```

テーブル t1 に 100 万行を追加する:

```
apgl=> select count(*) from t1;
count
1000000
(1 row)
```

テーブル t1 の統計情報:

```
select * from pg_stat_all_tables where relname='t1';
-[ RECORD 1 ]-------+--------
relid         | 914744
schemaname    | public
relname       | t1
seq_scan      | 0
seq_tup_read  | 0
idx_scan      |
idx_tup_fetch |
n_tup_ins     | 1000000
n_tup_upd     | 0
n_tup_del     | 0
n_tup_hot_upd | 0
n_live_tup    | 1000000
n_dead_tup    | 0
n_mod_since_analyze | 1000000
last_vacuum      |
last_autovacuum  |
last_analyze     |
last_autoanalyze |
vacuum_count     | 0
autovacuum_count | 0
analyze_count    | 0
autoanalyze_count | 0
```

インデックスを追加する:

```
create index i2 on t1 (b,a);
```

EXPLAIN コマンドを実行する (プラン 1):

```
Bitmap Heap Scan on t1 (cost=10521.17..14072.67 rows=5000 width=4)
```

```
Recheck Cond: (a = 5)
# Bitmap Index Scan on i2 (cost=0.00..10519.92 rows=5000 width=0)
Index Cond: (a = 5)
(4 rows)
```

EXPLAIN ANALYZE コマンドを実行する (プラン 2):

```
explain (analyze,buffers,costs off) select a from t1 where b = 5;
QUERY PLAN
Bitmap Heap Scan on t1 (actual time=0.023..0.024 rows=1 loops=1)
Recheck Cond: (b = 5)
Heap Blocks: exact=1
Buffers: shared hit=4
# Bitmap Index Scan on i2 (actual time=0.016..0.016 rows=1 loops=1)
Index Cond: (b = 5)
Buffers: shared hit=3
Planning Time: 0.054 ms
Execution Time: 0.076 ms
(9 rows)
```

テーブルで自動バキュームが無効になっており、EXPLAIN ANALYZE コマンドが手動で実行されなかったため、EXPLAIN コマンドと ANALYZE コマンドには異なるプランが表示されます。では、テーブルの値を更新し、EXPLAIN ANALYZE プランを再生成しましょう。

```
update t1 set a=8 where b=5;
explain (analyze,buffers,costs off) select a from t1 where b = 5;
```

EXPLAIN ANALYZE コマンド (プラン 3) が次のように表示されるようになりました。

```
apgl=> explain (analyze,buffers,costs off) select a from t1 where b = 5;
QUERY PLAN
Bitmap Heap Scan on t1 (actual time=0.075..0.076 rows=1 loops=1)
Recheck Cond: (b = 5)
Heap Blocks: exact=1
Buffers: shared hit=5
# Bitmap Index Scan on i2 (actual time=0.017..0.017 rows=2 loops=1)
Index Cond: (b = 5)
Buffers: shared hit=3
Planning Time: 0.053 ms
Execution Time: 0.125 ms
```

プラン 2 とプラン 3 のコストを比較すると、統計情報をまだ収集していないため、プランと実行時間に相違が見られることがわかります。

次に、テーブルで ANALYZE を手動で実行し、統計情報を確認してプランを再生成しましょう。

```
apgl=> analyze t1
apgl# ;
ANALYZE
Time: 212.223 ms

apgl=> select * from pg_stat_all_tables where relname='t1';
-[ RECORD 1 ]-------+------------------------------
relid          | 914744
schemaname     | public
relname        | t1
seq_scan       | 3
seq_tup_read   | 1000000
idx_scan       | 3
idx_tup_fetch  | 3
n_tup_ins      | 1000000
n_tup_upd      | 1
n_tup_del      | 0
n_tup_hot_upd  | 0
n_live_tup     | 1000000
n_dead_tup     | 1
n_mod_since_analyze | 0
last_vacuum       |
last_autovacuum   |
last_analyze      | 2023-04-15 11:39:02.075089+00
last_autoanalyze  |
vacuum_count      | 0
autovacuum_count  | 0
analyze_count     | 1
autoanalyze_count | 0

Time: 148.347 ms
```

EXPLAIN ANALYZE コマンドを実行する (プラン 4):

```
apgl=> explain (analyze,buffers,costs off) select a from t1 where b = 5;
QUERY PLAN
Index Only Scan using i2 on t1 (actual time=0.022..0.023 rows=1 loops=1)
```

```
Index Cond: (b = 5)
Heap Fetches: 1
Buffers: shared hit=4
Planning Time: 0.056 ms
Execution Time: 0.068 ms
(6 rows)

Time: 138.462 ms
```

テーブルを手動で分析して統計情報を収集した後、すべてのプランの結果を比較すると、オプティマイザのプラン 4 が他のプランよりも優れており、クエリの実行時間も短縮されていることがわかります。この例では、データベースでメンテナンスアクティビティを実行することの重要性を示しています。

VACUUM FULL を使用してテーブル全体を書き換える

FULL パラメータを使用して VACUUM コマンドを実行すると、テーブルの内容全体が余分なスペースのない新しいディスクファイルに書き換えられ、未使用のスペースがオペレーティングシステムに返されます。この操作によってかなり遅くなるため、各テーブルで ACCESS EXCLUSIVE のロックが必要になります。また、テーブルの新しいコピーが書き込まれ、操作が完了するまで古いコピーは除去されないため、余分なディスク容量が必要となります。

VACUUM FULL は、次のような場合に便利です。:

- テーブルで大量のスペースを再利用する場合。
- 非プライマリキーテーブルで肥大化した領域を再利用する場合。

非プライマリキーテーブルを使用していて、データベースがダウンタイムを許容できる場合は、VACUUM FULL を使用することをお勧めします。

VACUUM FULL は他の操作よりも多くのロックを必要とするため、重要なデータベースで実行するとコストが高くなります。この方法を置き換えるには、次のセクションで説明する pg_repack 拡張機能を使用できます。このオプションは VACUUM FULL に似ていますが、ロックを最小限にする必要があり、Amazon RDS for PostgreSQL と Aurora PostgreSQL 互換の両方でサポートされています。

pg_repack を使用した肥大化の削除

pg_repack 拡張機能を使用すると、データベースロックを最小限に抑えながらテーブルとインデックスの肥大化を削除できます。この拡張機能はデータベースインスタンスで作成し、Amazon Elastic Compute Cloud (Amazon EC2) またはデータベースに接続できるコンピュータからクライアントのバージョンが拡張機能のバージョンと一致する pg_repack クライアントを実行できます。

VACUUM FULL とは異なり、pg_repack はダウンタイムやメンテナンスの期間を必要とせず、他のセッションをブロックすることもありません。

pg_repack は、VACUUM FULL、CLUSTER、または REINDEX が機能しない状況で役に立ちます。肥大化したテーブルのデータを含む新しいテーブルを作成し、元のテーブルの変更を追跡して、元のテーブルを新しいテーブルに置き換えます。新しいテーブルの構築中は、元のテーブルに対する読み取りまたは書き込みオペレーションがロックされません。

完全なテーブルまたはインデックスに対して pg_repack を使用できます。タスクのリストを確認するには、pg_repack のドキュメントを参照してください。

制限:

- pg_repack を実行するには、テーブルにプライマリキーまたは一意のインデックスが必要です。
- pg_repack は一時テーブルでは機能しません。
- pg_repack は、グローバルインデックスを使用するテーブルでは機能しません。
- pg_repack が進行中の場合、テーブルに対して DDL オペレーションを実行することはできません。

以下の表では、pg_repack と VACUUM FULL の違いについて説明しています。

VACUUM FULL	pg_repack
組み込みコマンド	Amazon EC2 またはローカルコンピュータから実行する拡張機能
テーブルでの作業中に ACCESS EXCLUSIVE のロックが必要	短時間のみ ACCESS EXCLUSIVE のロックが必要

すべてのテーブルで動作	プライマリキーと一意のキーのみを持つテーブルで動作
テーブルとインデックスで使用するストレージの 2 倍が必要	テーブルとインデックスで使用するストレージの 2 倍が必要

テーブルで pg_repack を実行するには、次のコマンドを使用します。

```
pg_repack -h <host> -d <dbname> --table <tablename> -k
```

インデックスで pg_repack を実行するには、次のコマンドを使用します。

```
pg_repack -h <host> -d <dbname> --index <index name>
```

詳細については、AWS のブログ記事「Remove bloat from Amazon Aurora and RDS for PostgreSQL with pg_repack」を参照してください。

注意

pg_repack の error-on-invalid-index エラーは通常、テーブル上の 1 つ以上のインデックスが破損しているか無効であることを意味します。pg_repack は再パック処理中、データ整合性のためにインデックスに依存するため、無効なインデックスを持つテーブルでは安全に動作できません。
このエラーは、次の場合に発生します。

- インデックスが無効としてマークされている場合 (例えば、CREATE INDEX CONCURRENTLY ステートメントが失敗したため)。
- インデックスが破損している場合 (ハードウェアの問題や突然のシャットダウンが原因の可能性があります)。

次のクエリを使用して無効なインデックスを特定し、見つかった場合は最初に削除します。

```
SELECT indexrelid::regclass, indisvalid FROM pg_index WHERE indrelid =
 'orders'::regclass AND NOT indisvalid; Drop the invalid index: DROP INDEX
 index_name;
```

インデックスの再構築

PostgreSQL の REINDEX コマンドで、インデックスのテーブルに保存されているデータを使用し、インデックスの古いコピーを置き換えることでてインデックスを再構築します。次のようなシナリオでは、REINDEX の使用をお勧めします。

- インデックスが破損し、有効なデータが含まれなくなった場合。これは、ソフトウェアまたはハードウェアの障害の結果として発生する可能性があります。
- 以前にインデックスを使用したクエリが使用を停止した場合。
- インデックスが多数の空のページまたはほぼ空のページにより肥大化した場合。肥大化率 (bloat_pct) が 20 より大きいときは、REINDEX を実行する必要があります。

次のクエリは、bloat_pct を検出するのに役立ちます。

```
SELECT current_database(), nspname AS schemaname, tblname, idxname,
 bs*(relpages)::bigint AS real_size,
  bs*(relpages-est_pages)::bigint AS extra_size,
  100 * (relpages-est_pages)::float / relpages AS extra_pct,
  fillfactor,
  CASE WHEN relpages > est_pages_ff
    THEN bs*(relpages-est_pages_ff)
    ELSE 0
  END AS bloat_size,
  100 * (relpages-est_pages_ff)::float / relpages AS bloat_pct,
  is_na
  -- , 100-(pst).avg_leaf_density AS pst_avg_bloat, est_pages, index_tuple_hdr_bm,
 maxalign, pagehdr, nulldatawidth, nulldatahdrwidth, reltuples, relpages -- (DEBUG
 INFO)
FROM (
  SELECT coalesce(1 +
         ceil(reltuples/floor((bs-pageopqdata-pagehdr)/(4+nulldatahdrwidth)::float)), 0
 -- ItemIdData size + computed avg size of a tuple (nulldatahdrwidth)
      ) AS est_pages,
      coalesce(1 +
         ceil(reltuples/floor((bs-pageopqdata-pagehdr)*fillfactor/
(100*(4+nulldatahdrwidth)::float))), 0
      ) AS est_pages_ff,
      bs, nspname, tblname, idxname, relpages, fillfactor, is_na
```

```
      -- , pgstatindex(idxoid) AS pst, index_tuple_hdr_bm, maxalign, pagehdr,
 nulldatawidth, nulldatahdrwidth, reltuples -- (DEBUG INFO)
  FROM (
      SELECT maxalign, bs, nspname, tblname, idxname, reltuples, relpages, idxoid,
 fillfactor,
            ( index_tuple_hdr_bm +
                maxalign - CASE -- Add padding to the index tuple header to align on
 MAXALIGN
                  WHEN index_tuple_hdr_bm%maxalign = 0 THEN maxalign
                  ELSE index_tuple_hdr_bm%maxalign
                END
              + nulldatawidth + maxalign - CASE -- Add padding to the data to align on
 MAXALIGN
                  WHEN nulldatawidth = 0 THEN 0
                  WHEN nulldatawidth::integer%maxalign = 0 THEN maxalign
                  ELSE nulldatawidth::integer%maxalign
                END
            )::numeric AS nulldatahdrwidth, pagehdr, pageopqdata, is_na
            -- , index_tuple_hdr_bm, nulldatawidth -- (DEBUG INFO)
      FROM (
          SELECT n.nspname, i.tblname, i.idxname, i.reltuples, i.relpages,
              i.idxoid, i.fillfactor, current_setting('block_size')::numeric AS bs,
              CASE -- MAXALIGN: 4 on 32bits, 8 on 64bits (and mingw32 ?)
                WHEN version() ~ 'mingw32' OR version() ~ '64-bit|x86_64|ppc64|ia64|
amd64' THEN 8
                ELSE 4
              END AS maxalign,
              /* per page header, fixed size: 20 for 7.X, 24 for others */
              24 AS pagehdr,
              /* per page btree opaque data */
              16 AS pageopqdata,
              /* per tuple header: add IndexAttributeBitMapData if some cols are null-
able */
              CASE WHEN max(coalesce(s.null_frac,0)) = 0
                  THEN 8 -- IndexTupleData size
                  ELSE 8 + (( 32 + 8 - 1 ) / 8) -- IndexTupleData size +
 IndexAttributeBitMapData size ( max num filed per index + 8 - 1 /8)
              END AS index_tuple_hdr_bm,
              /* data len: we remove null values save space using it fractionnal part
 from stats */
              sum( (1-coalesce(s.null_frac, 0)) * coalesce(s.avg_width, 1024)) AS
 nulldatawidth,
```

```
            max( CASE WHEN i.atttypid = 'pg_catalog.name'::regtype THEN 1 ELSE 0
END ) > 0 AS is_na
         FROM (
            SELECT ct.relname AS tblname, ct.relnamespace, ic.idxname, ic.attpos,
ic.indkey, ic.indkey[ic.attpos], ic.reltuples, ic.relpages, ic.tbloid, ic.idxoid,
ic.fillfactor,
                coalesce(a1.attnum, a2.attnum) AS attnum, coalesce(a1.attname,
a2.attname) AS attname, coalesce(a1.atttypid, a2.atttypid) AS atttypid,
                CASE WHEN a1.attnum IS NULL
                THEN ic.idxname
                ELSE ct.relname
                END AS attrelname
            FROM (
                SELECT idxname, reltuples, relpages, tbloid, idxoid, fillfactor,
indkey,
                    pg_catalog.generate_series(1,indnatts) AS attpos
                FROM (
                    SELECT ci.relname AS idxname, ci.reltuples, ci.relpages,
i.indrelid AS tbloid,
                        i.indexrelid AS idxoid,
                        coalesce(substring(
                            array_to_string(ci.reloptions, ' ')
                            from 'fillfactor=([0-9]+)')::smallint, 90) AS fillfactor,
                        i.indnatts,
                        pg_catalog.string_to_array(pg_catalog.textin(
                            pg_catalog.int2vectorout(i.indkey)),' ')::int[] AS indkey
                    FROM pg_catalog.pg_index i
                    JOIN pg_catalog.pg_class ci ON ci.oid = i.indexrelid
                    WHERE ci.relam=(SELECT oid FROM pg_am WHERE amname = 'btree')
                    AND ci.relpages > 0
                ) AS idx_data
            ) AS ic
            JOIN pg_catalog.pg_class ct ON ct.oid = ic.tbloid
            LEFT JOIN pg_catalog.pg_attribute a1 ON
                ic.indkey[ic.attpos] <> 0
                AND a1.attrelid = ic.tbloid
                AND a1.attnum = ic.indkey[ic.attpos]
            LEFT JOIN pg_catalog.pg_attribute a2 ON
                ic.indkey[ic.attpos] = 0
                AND a2.attrelid = ic.idxoid
                AND a2.attnum = ic.attpos
        ) i
        JOIN pg_catalog.pg_namespace n ON n.oid = i.relnamespace
```

```
            JOIN pg_catalog.pg_stats s ON s.schemaname = n.nspname
                                      AND s.tablename = i.attrelname
                                      AND s.attname = i.attname
            GROUP BY 1,2,3,4,5,6,7,8,9,10,11
      ) AS rows_data_stats
  ) AS rows_hdr_pdg_stats
) AS relation_stats
ORDER BY nspname, tblname, idxname;
```

完全に空のインデックスページは、繰り返し再利用されます。ただし、ページのインデックスキーが削除されてもスペースが割り当てられている場合は、定期的にインデックスを再作成することをお勧めします。

インデックスを再作成すると、クエリのパフォーマンスが向上します。次の表に示すように、3 つの方法でインデックスを再作成できます。

メソッド	説明	制限
CONCURRENTLY オプションと合わせて使用する CREATE INDEX および DROP INDEX	新しいインデックスを構築し、古いインデックスを削除します。オプティマイザは、古いインデックスの代わりに新しく作成されたインデックスを使用してプランを生成します。オフピーク時間中は、古いインデックスを削除できます。	CONCURRENTLY オプションを使用している場合は、受信するすべての変更を追跡する必要があるためインデックスの作成に時間がかかります。変更が停止すると、処理が完了としてマークされます。
CONCURRENTLY オプションを使用する REINDEX	再構築処理中に書き込み操作をロックします。PostgreSQL 12 以降のバージョンでは CONCURRENTLY オプションが用意されているため、これらのロックを回避できます。	CONCURRENTLY を使用すると、インデックスの再構築に時間がかかります。
pg_repack 拡張機能	テーブルから肥大化を削除し、インデックスを再構築します。	この拡張機能は、EC2 インスタンスまたはデータベースに接続されているローカルコン

ピュータから実行する必要があります。

新しいインデックスを作成する

DROP INDEX コマンドと CREATE INDEX コマンドを併用すると、インデックスが再構築されます。

```
DROP INDEX <index_name>
CREATE INDEX <index_name> ON TABLE <table_name> (<column1>[,<column2>])
```

この方法の欠点は、テーブルに対する排他的ロックであり、このアクティビティ中のパフォーマンスに影響することです。DROP INDEX コマンドによって排他的ロックを取得し、テーブルに対する読み取り操作と書き込み操作の両方をブロックします。CREATE INDEX コマンドを使用して、テーブルに対する書き込み操作をブロックします。これにより読み取り操作が可能になりますが、インデックスの作成時にコストがかかります。

インデックスの再構築

REINDEX コマンドは、一貫したデータベースパフォーマンスを維持するのに役立ちます。テーブルに対して多数の DML 操作を実行すると、テーブルとインデックスの両方が肥大化します。インデックスを使用することでテーブルの検索が速くなり、クエリのパフォーマンスが向上します。インデックスの肥大化は、検索やクエリのパフォーマンスに影響します。したがって、クエリの一貫したパフォーマンスを維持するために、大量の DML 操作が含まれるテーブルに対してインデックスの再構築を実行することをお勧めします。

REINDEX コマンドによって、基になるテーブルの書き込み操作をロックし、インデックスを最初から再構築しますが、テーブルでの読み取り操作は可能です。ただし、インデックスでは読み取り操作がブロックされます。対応するインデックスを使用するクエリはブロックされますが、他のクエリはブロックされません。

PostgreSQL バージョン 12 では、新しいオプションパラメータの CONCURRENTLY が導入されました。これによりインデックスがゼロから再構築されますが、テーブルでの読み取り操作や書き込み操作、またはインデックスを使用するクエリはロックされません。ただし、このオプションを使用すると、処理が完了するまでに時間がかかります。

例

インデックスの作成と削除

CONCURRENTLY オプションを使用して新しいインデックスを作成します。

```
create index CONCURRENTLY on table(columns) ;
```

CONCURRENTLY オプションを使用して古いインデックスを削除します。

```
drop index CONCURRENTLY <index name> ;
```

インデックスの再構築

単一のインデックスを再構築するには:

```
reindex index <index name> ;
```

テーブル内のすべてのインデックスを再構築するには:

```
reindex table <table name> ;
```

スキーマ内のすべてのインデックスを再構築するには:

```
reindex schema <schema name> ;
```

インデックスを同時に再構築する

単一のインデックスを再構築するには:

```
reindex index CONCURRENTLY <indexname> ;
```

テーブル内のすべてのインデックスを再構築するには:

```
reindex table CONCURRENTLY <tablename> ;
```

スキーマ内のすべてのインデックスを再構築するには:

```
reindex schema CONCURRENTLY <schemaname> ;
```

インデックスのみを再構築または再配置する

単一のインデックスを再構築するには:

```
pg_repack -h <hostname> -d <dbname> -i <indexname> -k
```

すべてのインデックスを再構築するには:

```
pg_repack -h <hostname> -d <dbname> -x <indexname> -t <tablename> -k
```

例: autovacuum と VACUUM FULL を使用したスペースの再利用

例えば、500,000 行の emp テーブルを作成し、新しい値で行を更新してみましょう。自動バキュームが有効になっているため、このテーブルで VACUUM と ANALYZE コマンドの両方を実行して肥大化を削除し、スペースを再利用します。再利用されたスペースは繰り返し利用できますが、オペレーティングシステムに返されることはありません。

次のクエリによって、テーブルの肥大化が判断されます。

```
-- WARNING: When run with a non-superuser role, the query inspects only indexes on
 tables you are granted to read.
-- WARNING: Rows with is_na = 't' are known to have bad statistics ("name" type is not
 supported).
-- This query is compatible with PostgreSQL 8.2 and later.
SELECT current_database(), nspname AS schemaname, tblname, idxname,
 bs*(relpages)::bigint AS real_size,
  bs*(relpages-est_pages)::bigint AS extra_size,
  100 * (relpages-est_pages)::float / relpages AS extra_pct,
  fillfactor,
  CASE WHEN relpages > est_pages_ff
    THEN bs*(relpages-est_pages_ff)
    ELSE 0
  END AS bloat_size,
  100 * (relpages-est_pages_ff)::float / relpages AS bloat_pct,
  is_na
  -- , 100-(pst).avg_leaf_density AS pst_avg_bloat, est_pages, index_tuple_hdr_bm,
 maxalign, pagehdr, nulldatawidth, nulldatahdrwidth, reltuples, relpages -- (DEBUG
 INFO)
FROM (
  SELECT coalesce(1 +
         ceil(reltuples/floor((bs-pageopqdata-pagehdr)/(4+nulldatahdrwidth)::float)), 0
 -- ItemIdData size + computed avg size of a tuple (nulldatahdrwidth)
      ) AS est_pages,
      coalesce(1 +
         ceil(reltuples/floor((bs-pageopqdata-pagehdr)*fillfactor/
(100*(4+nulldatahdrwidth)::float))), 0
      ) AS est_pages_ff,
      bs, nspname, tblname, idxname, relpages, fillfactor, is_na
```

```
      -- , pgstatindex(idxoid) AS pst, index_tuple_hdr_bm, maxalign, pagehdr,
 nulldatawidth, nulldatahdrwidth, reltuples -- (DEBUG INFO)
  FROM (
      SELECT maxalign, bs, nspname, tblname, idxname, reltuples, relpages, idxoid,
 fillfactor,
            ( index_tuple_hdr_bm +
                maxalign - CASE -- Add padding to the index tuple header to align on
 MAXALIGN
                  WHEN index_tuple_hdr_bm%maxalign = 0 THEN maxalign
                  ELSE index_tuple_hdr_bm%maxalign
                END
              + nulldatawidth + maxalign - CASE -- Add padding to the data to align on
 MAXALIGN
                  WHEN nulldatawidth = 0 THEN 0
                  WHEN nulldatawidth::integer%maxalign = 0 THEN maxalign
                  ELSE nulldatawidth::integer%maxalign
                END
            )::numeric AS nulldatahdrwidth, pagehdr, pageopqdata, is_na
            -- , index_tuple_hdr_bm, nulldatawidth -- (DEBUG INFO)
      FROM (
          SELECT n.nspname, i.tblname, i.idxname, i.reltuples, i.relpages,
              i.idxoid, i.fillfactor, current_setting('block_size')::numeric AS bs,
              CASE -- MAXALIGN: 4 on 32bits, 8 on 64bits (and mingw32 ?)
                WHEN version() ~ 'mingw32' OR version() ~ '64-bit|x86_64|ppc64|ia64|
amd64' THEN 8
                ELSE 4
              END AS maxalign,
              /* per page header, fixed size: 20 for 7.X, 24 for others */
              24 AS pagehdr,
              /* per page btree opaque data */
              16 AS pageopqdata,
              /* per tuple header: add IndexAttributeBitMapData if some cols are null-
able */
              CASE WHEN max(coalesce(s.null_frac,0)) = 0
                  THEN 8 -- IndexTupleData size
                  ELSE 8 + (( 32 + 8 - 1 ) / 8) -- IndexTupleData size +
 IndexAttributeBitMapData size ( max num filed per index + 8 - 1 /8)
              END AS index_tuple_hdr_bm,
              /* data len: we remove null values save space using it fractionnal part
 from stats */
              sum( (1-coalesce(s.null_frac, 0)) * coalesce(s.avg_width, 1024)) AS
 nulldatawidth,
```

```
            max( CASE WHEN i.atttypid = 'pg_catalog.name'::regtype THEN 1 ELSE 0
END ) > 0 AS is_na
         FROM (
            SELECT ct.relname AS tblname, ct.relnamespace, ic.idxname, ic.attpos,
ic.indkey, ic.indkey[ic.attpos], ic.reltuples, ic.relpages, ic.tbloid, ic.idxoid,
ic.fillfactor,
                coalesce(a1.attnum, a2.attnum) AS attnum, coalesce(a1.attname,
a2.attname) AS attname, coalesce(a1.atttypid, a2.atttypid) AS atttypid,
                CASE WHEN a1.attnum IS NULL
                THEN ic.idxname
                ELSE ct.relname
                END AS attrelname
            FROM (
                SELECT idxname, reltuples, relpages, tbloid, idxoid, fillfactor,
indkey,
                    pg_catalog.generate_series(1,indnatts) AS attpos
                FROM (
                    SELECT ci.relname AS idxname, ci.reltuples, ci.relpages,
i.indrelid AS tbloid,
                        i.indexrelid AS idxoid,
                        coalesce(substring(
                            array_to_string(ci.reloptions, ' ')
                            from 'fillfactor=([0-9]+)')::smallint, 90) AS fillfactor,
                        i.indnatts,
                        pg_catalog.string_to_array(pg_catalog.textin(
                            pg_catalog.int2vectorout(i.indkey)),' ')::int[] AS indkey
                    FROM pg_catalog.pg_index i
                    JOIN pg_catalog.pg_class ci ON ci.oid = i.indexrelid
                    WHERE ci.relam=(SELECT oid FROM pg_am WHERE amname = 'btree')
                    AND ci.relpages > 0
                ) AS idx_data
            ) AS ic
            JOIN pg_catalog.pg_class ct ON ct.oid = ic.tbloid
            LEFT JOIN pg_catalog.pg_attribute a1 ON
                ic.indkey[ic.attpos] <> 0
                AND a1.attrelid = ic.tbloid
                AND a1.attnum = ic.indkey[ic.attpos]
            LEFT JOIN pg_catalog.pg_attribute a2 ON
                ic.indkey[ic.attpos] = 0
                AND a2.attrelid = ic.idxoid
                AND a2.attnum = ic.attpos
        ) i
        JOIN pg_catalog.pg_namespace n ON n.oid = i.relnamespace
```

```
            JOIN pg_catalog.pg_stats s ON s.schemaname = n.nspname
                                      AND s.tablename = i.attrelname
                                      AND s.attname = i.attname
            GROUP BY 1,2,3,4,5,6,7,8,9,10,11
      ) AS rows_data_stats
  ) AS rows_hdr_pdg_stats
) AS relation_stats
ORDER BY nspname, tblname, idxname;
```

クエリの結果では、テーブルの肥大化が約 51% であることを示しています。

```
current_database | schemaname | tblname | real_size | extra_size | extra_pct |
 fillfactor | bloat_size | bloat_pct | is_na
------------------+------------+---------+-----------+------------+------------------
+------------+------------+-------------------+-------
apgl | public | emp | 60383232 | 30744576 | 50.91575091575091 | 100 | 30744576 |
 50.91575091575091 | f
```

pg_stat_all_tables ビューの統計は次のとおりです。

```
relid           | 914748
schemaname      | public
relname         | emp
seq_scan        | 5
seq_tup_read    | 1500000
idx_scan        | 0
idx_tup_fetch   | 0
n_tup_ins       | 600000
n_tup_upd       | 500000
n_tup_del       | 0
n_tup_hot_upd   | 0
n_live_tup      | 500000
n_dead_tup      | 0
n_mod_since_analyze | 0
last_vacuum         |
last_autovacuum     | 2023-04-15 11:59:54.957449+00
last_analyze        |
last_autoanalyze    | 2023-04-15 11:59:55.016352+00
vacuum_count        | 0
autovacuum_count    | 2
analyze_count       | 0
autoanalyze_count   | 3
```

autovacuum の実行後に last_autovacuum 列と last_autoanalyze 列が更新されていることに注意してください。

空のスペースも肥大化していると見なされるため、テーブルにいくつか行を挿入して extra_size(bloat_size) を確認しましょう。

```
apgl=> select count(*) from emp;
count | 900000

current_database | schemaname | tblname | real_size | extra_size | extra_pct |
 fillfactor | bloat_size | bloat_pct | is_na
------------------+------------+---------+-----------+------------+------------------
+------------+------------+--------------------+-------
apgl | public | emp | 61349888 | 327680 | 0.5341167044999332 | 100 | 327680 |
 0.5341167044999332 | f
(1 row)
```

出力の bloat_pct 列は、クリーンアップされたスペースが新しい挿入によって占有されていることを示します。VACUUM FULL を実行しましょう。

```
apgl=> vacuum full emp ;
VACUUM

current_database | schemaname | tblname | real_size | extra_size | extra_pct |
 fillfactor | bloat_size | bloat_pct | is_na
------------------+------------+---------+-----------+------------+-----------
+------------+------------+-----------+-------
apgl | public | emp | 60792832 | -229376 | 0 | 100 | 0 | 0 | f
(1 row)
```

この出力から、空のスペースと肥大化が削除され、スペースがオペレーティングシステムに返されたことがわかります。

> **Note**
>
> VACUUM FULL の代わりに、pg_repack を実行して同じ結果を取得できます。

リソース

- Amazon RDS for PostgreSQL 環境での autovacuum について (AWS ブログ記事)
- 自動バキューム処理 (PostgreSQL ドキュメント)
- Allocating memory for autovacuum (Amazon RDS ドキュメント)
- トランザクション ID の循環障害の防止 (PostgreSQL ドキュメント)
- Remove bloat from Amazon Aurora and RDS for PostgreSQL with pg_repack (AWS のブログ記事)

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
更新	エラーを修正し、「テーブルのバキューム処理と分析の自動実行」および「Removing bloat with pg_repack」セクションに情報を追加しました。	2025 年 8 月 22 日
reindex 構文の修正	インデックスを同時に再構築するためのセクションで、reindex の例を修正しました。	2025 年 6 月 30 日
初版発行	—	2023 年 12 月 22 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。