aws

Atividades de manutenção de bancos de dados PostgreSQL no Amazon RDS e no Amazon Aurora para evitar problemas de performance

AWS Recomendações

AWS Recomendações: Atividades de manutenção de bancos de dados PostgreSQL no Amazon RDS e no Amazon Aurora para evitar problemas de performance

Table of Contents

Introdução 1
Resultados de negócios desejados 1
Controle de concorrência multiversão (MVCC) 1
Limpeza e análise automática de tabelas 3
Parâmetros relacionados à memória do autovacuum 5
Ajuste dos parâmetros de autovacuum 6
Nível de cluster ou instância 6
Nível de tabela 6
Uso de configurações agressivas de autovacuum no nível de tabela 6
Vantagens e limitações 7
Limpeza e análise manual de tabelas 9
Execução de operações de vacuum e limpeza em paralelo 10
Reescrita de toda uma tabela com VACUUM FULL 14
Remoção do inchaço com pg_repack 15
Recriação de índices 17
Criação de um novo índice 21
Recriação de um índice 21
Exemplos 22
Exemplo: recuperação de espaço usando autovacuum e VACUUM FULL 24
Recursos 29
Histórico do documento 30
..... xxxi

Atividades de manutenção de bancos de dados PostgreSQL no Amazon RDS e no Amazon Aurora para evitar problemas de performance

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

Agosto de 2025 (histórico do documento)

A edição do Amazon Aurora compatível com PostgreSQL e o Amazon Relational Database Service (Amazon RDS) para PostgreSQL são serviços de banco de dados relacionais totalmente gerenciados para bancos de dados PostgreSQL. Esses serviços gerenciados liberam o administrador do banco de dados de muitas tarefas de manutenção e gerenciamento. No entanto, algumas tarefas de manutenção, como `VACUUM`, exigem monitoramento e configuração rigorosos com base no uso do banco de dados. Este guia descreve as atividades de manutenção do PostgreSQL no Amazon RDS e no Aurora.

Resultados de negócios desejados

A performance do banco de dados é uma medida fundamental que sustenta o sucesso de uma empresa. A realização de atividades de manutenção em seus bancos de dados do Aurora compatível com PostgreSQL e do Amazon RDS para PostgreSQL proporciona as seguintes vantagens:

- Ajuda a alcançar a performance ideal de consulta
- Libera espaço inchado para reutilização em transações futuras
- Evita o wraparound da transação
- Ajuda o otimizador a gerar bons planos
- Garante o uso adequado do índice

Controle de concorrência multiversão (MVCC)

A manutenção do banco de dados PostgreSQL requer uma compreensão do controle de concorrência multiversão (MVCC), que é um mecanismo do PostgreSQL. Quando várias transações são processadas simultaneamente no banco de dados, o MVCC garante que a atomicidade e o

isolamento, que são duas características das transações de atomicidade, consistência, isolamento e durabilidade (ACID), sejam mantidos. No MVCC, cada operação de gravação gera uma nova versão dos dados e armazena a versão anterior. Leitores e gravadores não bloqueiam uns aos outros. Quando uma transação lê dados, o sistema escolhe uma das versões para fornecer isolamento da transação. O PostgreSQL e alguns bancos de dados relacionais usam uma adaptação do MVCC chamada isolamento de snapshot (SI). Por exemplo, o Oracle implementa o SI usando segmentos de reversão. Durante uma operação de gravação, o Oracle grava a versão antiga dos dados em um segmento de reversão e substitui a área de dados pela nova versão. Os bancos de dados PostgreSQL implementam o SI usando regras de verificação de visibilidade para avaliar as versões. Quando novos dados são colocados em uma página de tabela, o PostgreSQL usa essas regras para selecionar a versão apropriada dos dados para uma operação de leitura.

Quando você modifica dados em uma linha da tabela, o PostgreSQL usa o MVCC para manter várias versões da linha. Durante as operações `UPDATE` e `DELETE` na tabela, o banco de dados mantém as versões antigas das linhas para outras transações em execução que talvez precisem de uma visão consistente dos dados. Essas versões antigas são chamadas de linhas inativas (tuplas). Uma coleção de tuplas inativas produz inchaço. Um grande volume de inchaço no banco de dados pode causar vários problemas, incluindo geração deficiente de planos de consulta, performance lenta de consultas e maior uso de espaço em disco para armazenar as versões mais antigas.

A remoção do inchaço e a manutenção de um banco de dados íntegro requer manutenção periódica, que inclui essas atividades, que são abordadas nas seguintes seções:

- Limpeza e análise automática de tabelas
- Limpeza e análise manual de tabelas
- Remoção do inchaço com pg_repack
- Recriação de índices

Limpeza e análise automática de tabelas

O Autovacuum é um daemon (ou seja, ele é executado em segundo plano) que aspira (limpa) automaticamente as tuplas inativas, recupera o armazenamento e reúne estatísticas. Ele verifica se há tabelas inchadas no banco de dados e limpa o inchaço para reutilizar o espaço. Ele monitora tabelas e índices do banco de dados e os adiciona a uma tarefa de vacuum após atingirem um limite específico de operações de atualização ou exclusão.

O Autovacuum gerencia a limpeza automatizando os comandos `VACUUM` e `ANALYZE` do PostgreSQL. O `VACUUM` remove o inchaço das tabelas e recupera o espaço, enquanto o `ANALYZE` atualiza as estatísticas que permitem que o otimizador gere planos eficientes. O `VACUUM` também executa uma tarefa importante chamada congelamento de vacuum para evitar problemas de wraparound de IDs de transação no banco de dados. Cada linha atualizada no banco de dados recebe um ID de transação do mecanismo de controle de transações do PostgreSQL. Eles IDs controlam a visibilidade da linha em relação a outras transações simultâneas. O ID de transação é um número de 32 bits. Dois bilhões IDs são sempre mantidos no passado visível. O restante (cerca de 2,2 bilhões) IDs é preservado para transações que ocorrerão no futuro e está oculto da transação atual. O PostgreSQL exige uma limpeza e um congelamento ocasionais de linhas antigas para evitar que as transações sejam agrupadas e tornem as linhas antigas e existentes invisíveis quando novas transações forem criadas. Para obter mais informações, consulte Evitar falhas de conclusão do ID da transação na documentação do PostgreSQL.

O autovacuum é recomendado e habilitado por padrão. Seus parâmetros incluem o seguinte:

Parâmetro	Descrição	Padrão para Amazon RDS	Padrão para Aurora
`autovacuum_vacuum_threshold`	O número mínimo de operações de atualização ou exclusão de tuplas que devem ocorrer em uma tabela antes que o autovacuum a limpe.	50 operações	50 operações

`autovacuum_analyze_threshold`	O número mínimo de inserções, atualizações ou exclusões de tuplas que devem ocorrer em uma tabela antes que o autovacuum a analise.	50 operações	50 operações
`autovacuum_vacuum_scale_factor`	A porcentagem de tuplas que devem ser modificadas em uma tabela antes que o autovacuum a limpe.	0.1	0.1
`autovacuum_analyze_scale_factor`	A porcentagem de tuplas que devem ser modificadas em uma tabela antes que o autovacuum a analise.	0,05	0,05
`autovacuum_freeze_max_age`	A idade máxima de congelamento IDs antes de uma mesa ser limpa para evitar problemas de encapsulamento do ID da transação.	200 milhões de transações	200 milhões de transações

O Autovacuum faz uma lista de tabelas para processar com base em fórmulas de limite específicas, conforme a seguir.

- Limite de execução do `VACUUM` em uma tabela:

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

- Limite de execução do ANALYZE em uma tabela:

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

Para tabelas de pequeno a médio porte, os valores padrão podem ser suficientes. No entanto, uma tabela grande com modificações de dados frequentes terá um número maior de tuplas inativas. Nesse caso, o autovacuum pode processar a tabela com frequência para fins de manutenção, e a manutenção de outras mesas pode ser atrasada ou ignorada até que a tabela grande seja concluída. Para evitar isso, você pode ajustar os parâmetros de autovacuum descritos na seção a seguir.

Parâmetros relacionados à memória do autovacuum

`autovacuum_max_workers`

Especifica o número máximo de processos de autovacuum (exceto o inicializador de autovacuum) que podem ser executados ao mesmo tempo. Esse parâmetro só pode ser definido quando você inicia o servidor. Se o processo de autovacuum estiver ocupado com uma tabela grande, esse parâmetro ajudará a executar a limpeza de outras tabelas.

`maintenance_work_mem`

Ele especifica a quantidade máxima de memória a ser utilizada por operações de manutenção, como `VACUUM`, `CREATE INDEX` e `ALTER`. No Amazon RDS e no Aurora, a memória é alocada com base na classe da instância usando a fórmula `GREATEST({DBInstanceClassMemory/63963136*1024},65536)`. Quando o autovacuum é executado, até `autovacuum_max_workers` vezes esse valor calculado pode ser alocado, portanto, tenha cuidado para não definir o valor muito alto. Para controlar isso, você pode definir `autovacuum_work_mem` separadamente.

`autovacuum_work_mem`

Especifica a memória máxima a ser usada por cada processo de operador de autovacuum. Esse parâmetro é padronizado para -1, o que indica que você deve usar o valor de `maintenance_work_mem`.

Para obter mais informações sobre parâmetros de memória de autovacuum, consulte Alocar memória para autovacuum na documentação do Amazon RDS.

Ajuste dos parâmetros de autovacuum

Os usuários talvez precisem ajustar os parâmetros de autovacuum, dependendo das operações de atualização e exclusão. As configurações dos parâmetros a seguir podem ser definidas no nível de tabela, instância ou cluster.

Nível de cluster ou instância

Como exemplo, vejamos um banco de dados do setor financeiro em que se espera um volume contínuo de operações de Linguagem de Manipulação de Dados (DML). Para manter a integridade do banco de dados, você deve ajustar os parâmetros de autovacuum no nível de cluster para o Aurora e no nível de instância para o Amazon RDS, além de aplicar o mesmo grupo de parâmetros ao leitor. No caso de um failover, os mesmos parâmetros devem ser aplicados ao novo gravador.

Nível de tabela

Por exemplo, em um banco de dados para entrega de alimentos em que se espera operações contínuas de DML em uma única tabela chamada `orders`, você deve considerar o ajuste do parâmetro `autovacuum_analyze_threshold` no nível de tabela usando o seguinte comando:

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

Uso de configurações agressivas de autovacuum no nível de tabela

A tabela `orders` de exemplo que tem operações contínuas de atualização e exclusão torna-se candidata à limpeza devido às configurações padrão de autovacuum. Isso leva à geração incorreta de planos e à lentidão nas consultas. Limpar o inchaço e atualizar as estatísticas requer configurações agressivas de autovacum em nível de tabela.

Para determinar as configurações, acompanhe a duração das consultas em execução nessa tabela e identifique a porcentagem de operações de DML que resultam em alterações no plano. A visuazliação de `pg_stat_user_tables` ajuda a monitorar as operações de inserção, atualização e exclusão.

Exemplo:

Vamos supor que o otimizador gere planos ruins sempre que 5% da tabela `orders` muda. Nesse caso, você deve alterar o limite do fator de escala para 2% da seguinte forma:

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

> **Tip**
>
> Selecione configurações agressivas de autovacuum com cuidado para evitar o alto consumo de recursos.

Para obter mais informações, consulte o seguinte:

- Entendendo o autovacuum nos ambientes Amazon RDS forAWS PostgreSQL (postagem no blog)
- Automatic Vacuuming (documentação do PostgreSQL)
- Ajuste dos parâmetros do PostgreSQL no Amazon RDS e no Amazon Aurora (orientação prescritiva)AWS

Para garantir que o autovacuum funcione de forma eficaz, monitore as linhas inativas, o uso do disco e a última vez em que o autovacuum ou o `ANALYZE` foi executado regularmente. A visualização `pg_stat_all_tables` fornece informações sobre cada tabela (`relname`) e quantas tuplas inativas (`n_dead_tup`) estão na tabela.

O monitoramento do número de tuplas inativas em cada tabela, especialmente em tabelas atualizadas com frequência, ajuda a determinar se os processos de autovacuum estão removendo periodicamente as tuplas inativas para que seu espaço em disco possa ser reutilizado para uma melhor performance. É possível usar a consulta abaixo para conferir o número de tuplas inativas e quando o último autovacuum foi executado nas tabelas:

```
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;
```

Vantagens e limitações

O autovacuum oferece as seguintes vantagens:

- Ele remove o inchaço das tabelas automaticamente.
- Isso evita o wraparound do ID de transação.
- Ele mantém as estatísticas do banco de dados atualizadas.

Limitações:

- Se as consultas usarem processamento paralelo, o número de processos worker poderá não ser suficiente para o autovacuum.
- Se o autovacuum for executado nos horários de pico, a utilização de recursos poderá aumentar. Você deve ajustar os parâmetros para lidar com esse problema.
- Se as páginas da tabela estiverem ocupadas em outra sessão, o autovacuum poderá ignorá-las.
- O autovacuum não pode acessar tabelas temporárias.

Limpeza e análise manual de tabelas

Se seu banco de dados for limpo pelo processo de autovacuum, é uma prática recomendada evitar a execução de verificações manuais em todo o banco de dados com muita frequência. Uma aspiração manual pode resultar em I/O cargas desnecessárias ou picos de CPU e também pode falhar na remoção de tuplas mortas. Execute aspiradores manuais somente table-by-table se for realmente necessário, como quando a proporção de tuplas vivas e mortas é baixa ou quando há longos espaços entre os aspiradores automáticos. Além disso, você deve executar aspiradores manuais quando a atividade do usuário for mínima.

O autovacuum também mantém as estatísticas de uma tabela atualizadas. Quando você executa o comando `ANALYZE` manualmente, ele recria essas estatísticas em vez de atualizá-las. A recriação de estatísticas quando elas já estão atualizadas pelo processo normal de autovacuum pode resultar na utilização de recursos do sistema.

Recomendamos que você execute os comandos VACUUM e ANALYZE manualmente nos seguintes cenários:

- Durante horários de baixo pico em tabelas mais movimentadas, quando a limpeza automática pode não ser suficiente.
- Imediatamente depois de carregar dados em massa na tabela de destino. Nesse caso, executar `ANALYZE` manualmente recria completamente as estatísticas, o que é uma opção melhor do que esperar o início do autovacuum.
- Para limpar tabelas temporárias (o autovacuum não pode acessá-las).

Para reduzir o I/O impacto ao executar os `ANALYZE` comandos `VACUUM` e na atividade simultânea do banco de dados, você pode usar o `vacuum_cost_delay` parâmetro. Em muitas situações, comandos de manutenção como `VACUUM` e `ANALYZE` não precisam ser concluídos rapidamente. No entanto, esses comandos não devem interferir na capacidade do sistema de realizar outras operações de banco de dados. Para evitar isso, você pode habilitar atrasos de vacuum baseados em custos usando o parâmetro `vacuum_cost_delay`. Por padrão, esse parâmetro é desabilitado para comandos `VACUUM` emitidos manualmente. Para habilitá-lo, defina-o com um valor diferente de zero.

Execução de operações de vacuum e limpeza em paralelo

A opção PARALLEL do comando VACUUM usa operadores paralelos para as fases de vacuum e limpeza do índice e está desabilitada por padrão. O número de operadores paralelos (o grau de paralelismo) é determinado pelo número de índices na tabela e pode ser especificado pelo usuário. Se você estiver executando operações VACUUM paralelas sem um argumento inteiro, o grau de paralelismo será calculado com base no número de índices na tabela.

Os parâmetros a seguir ajudam você a configurar a operação de vacuum paralela no Amazon RDS para PostgreSQL e no Aurora compatível com PostgreSQL:

- max_worker_processes define o número máximo de processos de operadores simultâneos.
- min_parallel_index_scan_size define a quantidade mínima de dados de índice que devem ser verificados para que uma verificação paralela seja considerada.
- max_parallel_maintenance_workers define o número máximo de operadores paralelos que podem ser iniciados por um único comando de utilitário.

> **Note**
>
> A opção PARALLEL é usada somente para fins de limpeza. Não afeta o comando ANALYZE.

O exemplo a seguir ilustra o comportamento do banco de dados quando você usa o VACUUM e ANALYZE manuais em um banco de dados.

Confira um exemplo de tabela em que o autovacuum foi desabilitado (apenas para fins ilustrativos; desabilitar o autovacuum não é recomendado):

```
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
```

Adicionar 1 milhão de linhas à tabela t1:

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

Estatísticas da tabela t1:

```
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
```

Adicionar um índice:

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

Execute o comando EXPLAIN (Plano 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)
```

Execute o comando EXPLAIN ANALYZE (Plano 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)
```

Os comandos EXPLAIN e EXPLAIN ANALYZE exibem planos diferentes, porque o autovacuum foi desabilitado na tabela e o comando ANALYZE não foi executado manualmente. Agora vamos atualizar um valor na tabela e regenerar o plano EXPLAIN ANALYZE:

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

O comando EXPLAIN ANALYZE (Plano 3) agora exibe:

```
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
```

Se você comparar os custos entre o Plano 2 e o Plano 3, verá as diferenças no tempo de planejamento e execução, porque ainda não coletamos estatísticas.

Agora vamos executar um ANALYZE manual na tabela, verificar as estatísticas e regenerar o plano:

```
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
```

Execute o comando EXPLAIN ANALYZE (Plano 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
```

Se comparar todos os resultados do plano depois de analisar manualmente a tabela e coletar estatísticas, você observará que o Plano 4 do otimizador é melhor do que os outros e que também diminui o tempo de execução da consulta. Este exemplo mostra como é importante executar atividades de manutenção no banco de dados.

Reescrita de toda uma tabela com VACUUM FULL

A execução do comando `VACUUM` com o parâmetro `FULL` regrava todo o conteúdo de uma tabela em um novo arquivo de disco sem espaço extra e retorna espaço não utilizado para o sistema operacional. Essa operação é muito mais lenta e requer um bloqueio `ACCESS EXCLUSIVE` em cada tabela. Também requer espaço extra em disco, pois grava uma nova cópia da tabela e não libera a cópia antiga até que a operação seja concluída.

O `VACUUM FULL` pode ser útil nos seguintes casos:

- Quando você quiser recuperar uma volume significativo de espaço nas tabelas.
- Quando você quiser recuperar espaço inchado em tabelas de chaves não primárias.

Recomendamos que você use `VACUUM FULL` quando tiver tabelas de chaves não primárias, se seu banco de dados puder tolerar tempo de inatividade.

Como o `VACUUM FULL` requer mais bloqueio do que outras operações, é mais caro executá-lo em bancos de dados cruciais. Para substituir esse método, você pode usar a extensão `pg_repack`, descrita na próxima seção. Essa opção é semelhante ao `VACUUM FULL`, mas requer bloqueio mínimo e é compatível com o Amazon RDS para PostgreSQL e o Aurora compatível com PostgreSQL.

Remoção do inchaço com pg_repack

Você pode usar a extensão pg_repack para remover o inchaço da tabela e do índice com o mínimo de bloqueio do banco de dados. Você pode criar essa extensão na instância do banco de dados e executar o cliente pg_repack (em que a versão do cliente corresponde à versão da extensão) do Amazon Elastic Compute Cloud (Amazon EC2) ou de um computador que possa se conectar ao seu banco de dados.

Diferentemente do VACUUM FULL, o pg_repack não exige tempo de inatividade ou janela de manutenção e não bloqueia outras sessões.

O pg_repack é útil em situações em que VACUUM FULL, CLUSTER ou REINDEX pode não funcionar. Ele cria uma tabela que contém os dados da tabela inchada, monitora as alterações na tabela original e, em seguida, substitui a tabela original pela nova. Ele não bloqueia a tabela original para operações de leitura ou gravação enquanto está criando a outra tabela.

Você pode usar pg_repack para uma tabela completa ou para um índice. Para ver uma lista de tarefas, consulte a documentação do pg_repack.

Limitações:

- Para executar o pg_repack, sua tabela deve ter uma chave primária ou um índice exclusivo.
- O pg_repack não funcionará com tabelas temporárias.
- O pg_repack não funcionará em tabelas que tenham índices globais.
- Quando o pg_repack está em andamento, não é possível realizar operações de DDL em tabelas.

A tabela a seguir descreve as diferenças entre pg_repack e VACUUM FULL.

VACUUM FULL	pg_repack
Comando integrado	Uma extensão que você executa no Amazon EC2 ou no seu computador local
Requer um bloqueio ACCESS EXCLUSIVE enquanto está operando em uma tabela	Requer um bloqueio ACCESS EXCLUSIVE apenas por um curto período

Funciona com todas as tabelas	Funciona em tabelas que têm somente chaves primárias e exclusivas
Requer o dobro do armazenamento consumido pela tabela e pelos índices	Requer o dobro do armazenamento consumido pela tabela e pelos índices

Para executar `pg_repack` em uma tabela, use o comando:

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

Para executar `pg_repack` em um índice, use o comando:

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

Para obter mais informações, consulte a publicação do Blog da AWS Remove bloat from Amazon Aurora and RDS for PostgreSQL with pg_repack.

Advertência

O erro `error-on-invalid-index` no `pg_repack` geralmente significa que um ou mais índices na tabela estão corrompidos ou são inválidos. O `pg_repack` não pode operar com segurança em tabelas com índices inválidos, porque depende dos índices para a consistência de dados durante o processo de empacotamento.
Esse erro ocorre quando:

- O índice está marcado como inválido (por exemplo, devido a uma instrução `CREATE INDEX CONCURRENTLY` com falha).
- O índice está corrompido (possivelmente devido a problemas de hardware ou desligamentos abruptos).

Use a consulta a seguir para identificar índices inválidos e eliminá-los primeiro, caso você os encontre.

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

Recriação de índices

O comando REINDEX do PostgreSQL recria um índice usando os dados armazenados na tabela do índice e substituindo a cópia antiga do índice. Recomendamos que use `REINDEX` nos seguintes cenários:

- Quando um índice está corrompido e não contém mais dados válidos. Isso pode acontecer como resultado de falhas de software ou hardware.
- Quando as consultas que anteriormente usavam o índice param de usá-lo.
- Quando o índice fica inchado com um grande número de páginas vazias ou praticamente vazias. Você deve executar `REINDEX` quando a porcentagem de inchaço (`bloat_pct`) for maior que 20.

A consulta a seguir ajuda você a encontrar `bloat_pct`:

```
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;
```

As páginas de índice que estão completamente vazias são recuperadas para reutilização. No entanto, recomendamos a reindexação periódica se as chaves de índice em uma página tiverem sido excluídas, mas o espaço permanecer alocado.

A recriação do índice ajuda a melhorar a performance da consulta. Você pode recriar um índice de três maneiras, conforme descrito na tabela a seguir.

Method (Método	Descrição	Limitações
`CREATE INDEX` e `DROP INDEX` com a opção `CONCURRENTLY`	Cria um novo índice e remove o índice antigo. O otimizador gera planos usando o índice recém-criado em vez do índice antigo. Durante os horários de baixo pico, você pode descartar o índice antigo.	A criação do índice leva mais tempo quando você usa a opção `CONCURRENTLY` , pois ela precisa rastrear todas as alterações recebidas. Quando as alterações são congelada s, o processo é marcado como concluído.
`REINDEX` com a opção `CONCURRENTLY`	Bloqueia as operações de gravação durante o processo de recriação. A versão 12 do PostgreSQL e as versões posteriores oferecem a opção `CONCURRENTLY` , que evita esses bloqueios.	O uso de `CONCURRENTLY` requer mais tempo para recriar o índice.
Extensão do `pg_repack`	Limpa o inchaço de uma tabela e recria o índice.	Você deve executar essa extensão de uma instância do EC2 ou de um computador

local conectado ao banco de dados.

Criação de um novo índice

Os comandos `DROP INDEX` e `CREATE INDEX`, quando usados juntos, recriam um índice:

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

A desvantagem dessa abordagem é seu bloqueio exclusivo na tabela, o que afeta a performance durante essa atividade. O comando `DROP INDEX` adquire um bloqueio exclusivo, que bloqueia as operações de leitura e gravação na tabela. O comando `CREATE INDEX` bloqueia as operações de gravação na tabela. Ele permite operações de leitura, mas elas são caras durante a criação do índice.

Recriação de um índice

O comando `REINDEX` ajuda você a manter a performance consistente do banco de dados. Quando você executa um grande número de operações de DML em uma tabela, elas resultam em inchaço da tabela e do índice. Os índices são usados para acelerar a pesquisa em tabelas e melhorar a performance da consulta. O inchaço do índice afeta as pesquisas e a performance das consultas. Portanto, recomendamos que você execute a reindexação em tabelas que tenham um alto volume de operações de DML para manter a consistência na performance das consultas.

O comando `REINDEX` recria o índice do zero bloqueando as operações de gravação na tabela subjacente, mas permite operações de leitura na tabela. No entanto, ele bloqueia as operações de leitura no índice. As consultas que usam o índice correspondente são bloqueadas, mas outras consultas não.

A versão 12 do PostgreSQL introduziu um novo parâmetro opcional, `CONCURRENTLY`, que recria o índice do zero, mas não bloqueia as operações de gravação ou leitura na tabela ou nas consultas que usam o índice. No entanto, leva mais tempo para concluir o processo quando você usa essa opção.

Exemplos

Criação e exclusão de um índice

Crie um novo índice com a opção `CONCURRENTLY`:

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

Exclua o índice antigo com a opção `CONCURRENTLY`:

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

Recriação de um índice

Para recriar um único índice:

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

Para recriar todos os índices em uma tabela:

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

Para recriar todos os índices em um esquema:

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

Recriação de um índice simultaneamente

Para recriar um único índice:

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

Para recriar todos os índices em uma tabela:

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

Para recriar todos os índices em um esquema:

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

Recriação ou realocamento somente de índices

Para recriar um único índice:

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

Para recriar todos os índices:

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

Exemplo: recuperação de espaço usando autovacuum e VACUUM FULL

Como exemplo, vamos criar uma tabela `emp` com 500 mil linhas e, em seguida, atualizar as linhas com novos valores. O Autovacuum está habilitado, então ele executará os comandos `VACUUM` e `ANALYZE` nessa tabela para remover o inchaço e recuperar espaço. O espaço recuperado pode ser reutilizado, mas não será devolvido ao sistema operacional.

A consulta a seguir determina o inchaço na tabela:

```
-- 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;
```

Os resultados da consulta mostram que a tabela tem um inchaço de cerca de 51%:

```
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
```

Aqui estão as estatísticas da visualização `pg_stat_all_tables`:

```
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
```

Observe que o autovacuum atualizou as colunas `last_autovacuum` e `last_autoanalyze` após sua execução.

Agora, vamos inserir algumas linhas na tabela e verificar `extra_size(bloat_size)`, pois o espaço vazio também é considerado inchaço.

```
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)
```

A coluna `bloat_pct` na saída indica que o espaço limpo foi ocupado por novas inserções. Vamos executar o `VACUUM FULL`:

```
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)
```

Nessa saída, você pode ver que o espaço vazio e o inchaço foram removidos e que o espaço foi retornado ao sistema operacional.

> **Note**
>
> Em vez do `VACUUM FULL`, você pode executar `pg_repack` para obter os mesmos resultados.

Recursos

- Entendendo o autovacuum nos ambientes Amazon RDS forAWS PostgreSQL (postagem no blog)
- Automatic vacuuming (documentação do PostgreSQL)
- Alocar memória para autovacuum (documentação do Amazon RDS)
- Prevenindo falhas no Wraparound do ID da transação; (documentação do PostgreSQL)
- Remove bloat from Amazon Aurora and RDS for PostgreSQL with pg_repack (publicação do Blog da AWS)

Histórico do documento

A tabela a seguir descreve alterações significativas feitas neste guia. Se desejar receber notificações sobre futuras atualizações, inscreva-se em um feed RSS.

Alteração	Descrição	Data
Atualizações	Erros corrigidos e informaçõ es adicionadas às seções Limpeza e análise automátic as de tabelas e Remoção do inchaço com pg_repack.	22 de agosto de 2025
Sintaxe `reindex` corrigida	Na seção para recriar um índice simultaneamente, os exemplos de `reindex` foram corrigidos.	30 de junho de 2025
Publicação inicial	—	22 de dezembro de 2023

As traduções são geradas por tradução automática. Em caso de conflito entre o conteúdo da tradução e da versão original em inglês, a versão em inglês prevalecerá.