aws

Wartungsaktivitäten für PostgreSQL-Datenbanken in Amazon RDS und Amazon Aurora zur Vermeidung von Leistungsproblemen

AWS Prescriptive Guidance

AWS Prescriptive Guidance: Wartungsaktivitäten für PostgreSQL-Datenbanken in Amazon RDS und Amazon Aurora zur Vermeidung von Leistungsproblemen

Table of Contents

Einführung 1
Gezielte Geschäftsergebnisse 1
Multiversion-Parallelitätskontrolle (MVCC) 1
Automatisches Staubsaugen und Analysieren von Tabellen 3
Speicherbezogene Autovacuum-Parameter 5
Einstellung der Autovakuum-Parameter 6
Cluster- oder Instanzebene 6
Tabellenebene 6
Verwenden Sie aggressive Autovakuum-Einstellungen auf Tabellenebene 7
Vorteile und Einschränkungen 8
Manuelles Absaugen und Analysieren von Tabellen 9
Paralleles Ausführen von Saug- und Säuberungsvorgängen 10
Eine ganze Tabelle mit VACUUM FULL neu schreiben 14
Blähungen mit pg_repack entfernen 16
Indizes neu erstellen 19
Einen neuen Index erstellen 23
Einen Index neu aufbauen 23
Beispiele 24
Beispiel: Rückgewinnung von Speicherplatz mithilfe von Autovacuum und VACUUM FULL 26
Ressourcen 31
Dokumentverlauf 32
..... xxxiii

Wartungsaktivitäten für PostgreSQL-Datenbanken in Amazon RDS und Amazon Aurora zur Vermeidung von Leistungsproblemen

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

August 2025 (Geschichte der Dokumente)

Amazon Aurora PostgreSQL-Compatible Edition und Amazon Relational Database Service (Amazon RDS) for PostgreSQL sind vollständig verwaltete relationale Datenbankservices für PostgreSQL-Datenbanken. Diese verwalteten Services befreien den Datenbankadministrator von vielen Wartungs- und Verwaltungsaufgaben. Einige Wartungsaufgaben, wie z. B.`VACUUM`, erfordern jedoch eine genaue Überwachung und Konfiguration auf der Grundlage Ihrer Datenbanknutzung. Dieses Handbuch beschreibt die PostgreSQL-Wartungsaktivitäten in Amazon RDS und Aurora.

Gezielte Geschäftsergebnisse

Die Datenbankleistung ist eine wichtige Kennzahl, die dem Erfolg eines Unternehmens zugrunde liegt. Die Durchführung von Wartungsaktivitäten an Ihren Aurora PostgreSQL-kompatiblen Datenbanken und Amazon RDS for PostgreSQL PostgreSQL-Datenbanken bietet folgende Vorteile:

- Hilft, eine optimale Abfrageleistung zu erzielen
- Macht aufgeblähten Speicherplatz für die Wiederverwendung durch future Transaktionen frei
- Verhindert den Umbruch von Transaktionen
- Hilft dem Optimierer, gute Pläne zu erstellen
- Sorgt für die korrekte Verwendung des Index

Multiversion-Parallelitätskontrolle (MVCC)

Die Wartung von PostgreSQL-Datenbanken erfordert ein Verständnis von Multiversion Concurrency Control (MVCC), einem Mechanismus von PostgreSQL. Wenn mehrere Transaktionen gleichzeitig in der Datenbank verarbeitet werden, stellt MVCC sicher, dass Atomizität und Isolation, zwei Merkmale von Transaktionen mit Atomarität, Konsistenz, Isolation und Haltbarkeit (ACID), beibehalten werden. In MVCC generiert jeder Schreibvorgang eine neue Version von Daten und speichert die

vorherige Version. Leser und Schreiber blockieren sich nicht gegenseitig. Wenn bei einer Transaktion Daten gelesen werden, wählt das System eine der Versionen für die Transaktionsisolierung aus. PostgreSQL und einige relationale Datenbanken verwenden eine Anpassung von MVCC, die als Snapshot Isolation (SI) bezeichnet wird. Beispielsweise implementiert Oracle SI mithilfe von Rollback-Segmenten. Während eines Schreibvorgangs schreibt Oracle die alte Version der Daten in ein Rollback-Segment und überschreibt den Datenbereich mit der neuen Version. PostgreSQL-Datenbanken implementieren SI, indem sie Regeln zur Sichtbarkeitsprüfung verwenden, um Versionen auszuwerten. Wenn neue Daten in eine Tabellenseite eingefügt werden, verwendet PostgreSQL diese Regeln, um die entsprechende Version der Daten für einen Lesevorgang auszuwählen.

Wenn Sie Daten in einer Tabellenzeile ändern, verwendet PostgreSQL MVCC, um mehrere Versionen der Zeile zu verwalten. Während der `UPDATE` `DELETE` Operationen an der Tabelle behält die Datenbank die alten Versionen der Zeilen für andere laufende Transaktionen bei, die möglicherweise eine konsistente Ansicht der Daten benötigen. Diese alten Versionen werden tote Zeilen (Tupel) genannt. Eine Ansammlung toter Tupel erzeugt Blähungen. Ein hohes Maß an Aufblähung in der Datenbank kann zu einer Reihe von Problemen führen, darunter eine schlechte Generierung von Abfrageplänen, eine langsame Abfrageleistung und eine erhöhte Speicherplatznutzung zum Speichern älterer Versionen.

Um aufgeblähte Daten zu entfernen und eine Datenbank funktionsfähig zu halten, ist eine regelmäßige Wartung erforderlich. Dazu gehören auch die folgenden Aktivitäten, die in den folgenden Abschnitten behandelt werden:

- Automatisches Löschen und Analysieren von Tabellen
- Manuelles Staubsaugen und Analysieren von Tabellen
- Blähungen mit pg_repack entfernen
- Indizes neu aufbauen

Automatisches Staubsaugen und Analysieren von Tabellen

Autovacuum ist ein Daemon (d. h. er läuft im Hintergrund), der automatisch tote Tupel absaugt (bereinigt), Speicherplatz zurückfordert und Statistiken sammelt. Er sucht nach aufgeblähten Tabellen in der Datenbank und löscht die aufgeblähten Tabellen, um den Speicherplatz wiederzuverwenden. Es überwacht Datenbanktabellen und Indizes und fügt sie einem Vakuumjob hinzu, sobald sie einen bestimmten Schwellenwert für Aktualisierungs- oder Löschvorgänge erreicht haben.

Autovacuum verwaltet das Staubsaugen, indem es PostgreSQL und Befehle automatisiert. `VACUUM ANALYZE` `VACUUM`entfernt überflüssige Daten aus Tabellen und beansprucht den Speicherplatz zurück. Gleichzeitig werden die Statistiken `ANALYZE` aktualisiert, die es dem Optimierer ermöglichen, effiziente Pläne zu erstellen. `VACUUM`führt außerdem eine wichtige Aufgabe aus, die so genannte Vakuumfixierung, um Probleme beim Umschließen von Transaktions-IDs in der Datenbank zu verhindern. Jede Zeile, die in der Datenbank aktualisiert wird, erhält eine Transaktions-ID vom PostgreSQL-Transaktionskontrollmechanismus. Diese IDs steuern die Sichtbarkeit der Zeile für andere gleichzeitige Transaktionen. Die Transaktions-ID ist eine 32-Bit-Zahl. Zwei Milliarden IDs werden immer in der sichtbaren Vergangenheit aufbewahrt. Die verbleibenden (etwa 2,2 Milliarden) IDs sind für Transaktionen reserviert, die in der future stattfinden werden, und sind vor der aktuellen Transaktion verborgen. PostgreSQL erfordert ein gelegentliches Bereinigen und Einfrieren alter Zeilen, um zu verhindern, dass Transaktionen umlaufen und alte, bestehende Zeilen unsichtbar machen, wenn neue Transaktionen erstellt werden. Weitere Informationen finden Sie unter Transaktions-ID-Wraparound-Fehler vermeiden in der PostgreSQL-Dokumentation.

Autovacuum wird empfohlen und ist standardmäßig aktiviert. Zu seinen Parametern gehören die folgenden.

Parameter	Beschreibung	Standard für Amazon RDS	Standard für Aurora
`autovacuum_vacuum_threshold`	Die Mindestanzahl von Tupel-Aktualisierungs- oder Löschvorgängen, die in einer Tabelle ausgeführt werden müssen, bevor sie durch	50 Operationen	50 Operationen

	Autovacuum gelöscht wird.		
`autovacuum_analyze_threshold`	Die Mindestanzahl von Tupeleinfügungen, Aktualisierungen oder Löschungen, die in einer Tabelle vorgenommen werden müssen, bevor Autovacuum sie analysiert.	50 Operationen	50 Operationen
`autovacuum_vacuum_scale_factor`	Der Prozentsatz der Tupel, die in einer Tabelle geändert werden müssen, bevor sie mit dem Autovakuumverfahren abgesaugt wird.	0.1	0.1
`autovacuum_analyze_scale_factor`	Der Prozentsatz der Tupel, die in einer Tabelle geändert werden müssen, bevor sie von Autovacuum analysiert wird.	0,05	0,05

`autovacuum_freeze_max_age`	Das maximale Alter, das eingefroren ist, IDs bevor eine Tabelle gelöscht wird, um Probleme beim Umschließen von Transaktions-IDs zu vermeiden.	200.000.000 Transaktionen	200.000.000 Transaktionen

Autovacuum erstellt wie folgt eine Liste der zu verarbeitenden Tabellen auf der Grundlage bestimmter Schwellenwertformeln.

- Schwellenwert für die Ausführung `VACUUM` auf einer Tabelle:

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

- Schwellenwert für das Ausführen `ANALYZE` auf einem Tisch:

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

Für kleine bis mittelgroße Tabellen können die Standardwerte ausreichend sein. Eine große Tabelle mit häufigen Datenänderungen weist jedoch eine höhere Anzahl toter Tupel auf. In diesem Fall kann Autovacuum die Tabelle zu Wartungszwecken häufig verarbeiten, und die Wartung anderer Tabellen kann verzögert oder ignoriert werden, bis die große Tabelle fertig ist. Um dies zu vermeiden, können Sie die im folgenden Abschnitt beschriebenen Autovakuum-Parameter einstellen.

Speicherbezogene Autovacuum-Parameter

`autovacuum_max_workers`

Gibt die maximale Anzahl von Autovacuum-Prozessen (mit Ausnahme des Autovacuum-Starters) an, die gleichzeitig ausgeführt werden können. Dieser Parameter kann nur festgelegt werden, wenn Sie den Server starten. Wenn der Autovacuum-Prozess mit einer großen Tabelle ausgelastet ist, hilft dieser Parameter dabei, die Bereinigung für andere Tabellen durchzuführen.

`maintenance_work_mem`

Gibt die maximale Speichermenge an, die für Wartungsvorgänge wie `VACUUM`, `CREATE INDEX` und `ALTER` verwendet werden soll. In Amazon RDS und Aurora wird Speicher basierend auf der Instance-Klasse mithilfe der Formel zugewiesen`GREATEST({DBInstanceClassMemory/63963136*1024},65536)`. Wenn Autovacuum ausgeführt wird, kann der berechnete Wert bis zu `autovacuum_max_workers` oft zugewiesen werden. Achten Sie also darauf, den Wert nicht zu hoch festzulegen. Um diesen Wert zu kontrollieren, können Sie ihn `autovacuum_work_mem` separat einstellen.

`autovacuum_work_mem`

Gibt die maximale Speichermenge an, die von jedem Autovacuum-Worker-Prozess verwendet werden soll. Dieser Parameter ist standardmäßig auf -1 voreingestellt, was bedeutet, dass Sie stattdessen den Wert von `maintenance_work_mem` verwenden sollten.

Weitere Informationen zu Autovacuum-Speicherparametern finden Sie unter Allocating memory for autovacuum in der Amazon RDS-Dokumentation.

Einstellung der Autovakuum-Parameter

Abhängig von ihren Aktualisierungs- und Löschvorgängen müssen Benutzer möglicherweise die Autovakuum-Parameter anpassen. Die Einstellungen für die folgenden Parameter können auf Tabellen-, Instanz- oder Clusterebene festgelegt werden.

Cluster- oder Instanzebene

Schauen wir uns als Beispiel eine Bankdatenbank an, in der kontinuierliche Datenmanipulationssprachen (DML) -Operationen erwartet werden. Um den Zustand der Datenbank aufrechtzuerhalten, sollten Sie Autovacuum-Parameter auf Cluster-Ebene für Aurora und auf Instance-Ebene für Amazon RDS optimieren und dieselbe Parametergruppe auch auf den Reader anwenden. Im Fall eines Failovers sollten dieselben Parameter für den neuen Writer gelten.

Tabellenebene

In einer Datenbank für die Lieferung von Lebensmitteln, in der kontinuierliche DML-Operationen für eine einzelne aufgerufene Tabelle erwartet werden`orders`, sollten Sie in Betracht ziehen, den

`autovacuum_analyze_threshold` Parameter auf Tabellenebene mithilfe des folgenden Befehls zu optimieren:

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

Verwenden Sie aggressive Autovakuum-Einstellungen auf Tabellenebene

Die `orders` Beispieltabelle mit fortlaufenden Aktualisierungs- und Löschvorgängen eignet sich aufgrund der Standardeinstellungen für das automatische Absaugen. Dies führt zu fehlerhafter Planerstellung und langsamen Abfragen. Um die Fülle zu beseitigen und die Statistiken zu aktualisieren, sind aggressive Autovaccum-Einstellungen auf Tabellenebene erforderlich.

Um die Einstellungen zu ermitteln, sollten Sie die Dauer der Abfragen verfolgen, die in dieser Tabelle ausgeführt werden, und den Prozentsatz der DML-Vorgänge ermitteln, die zu Planänderungen führen. Die `pg_stat_user_tables` Ansicht hilft Ihnen dabei, Einfüge-, Aktualisierungs- und Löschvorgänge nachzuverfolgen.

Beispiel:

Nehmen wir an, dass der Optimierer immer dann fehlerhafte Pläne generiert, wenn sich 5 Prozent der `orders` Tabelle ändern. In diesem Fall sollten Sie den Schwellenwert für den Skalierungsfaktor wie folgt auf 2 Prozent ändern:

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

> **Tip**
>
> Wählen Sie aggressive Autovakuum-Einstellungen sorgfältig aus, um einen hohen Ressourcenverbrauch zu vermeiden.

Weitere Informationen finden Sie hier:

- Grundlegendes zu Autovacuum in Amazon RDS for PostgreSQL PostgreSQL-Umgebungen (Blogbeitrag)AWS
- Automatisches Staubsaugen (PostgreSQL-Dokumentation)
- Optimierung der PostgreSQL-Parameter in Amazon RDS und Amazon Aurora (AWS Prescriptive Guidance)

Um sicherzustellen, dass Autovacuum effektiv funktioniert, sollten Sie tote Zeilen, die Festplattennutzung und den Zeitpunkt, zu dem der Autovacuum-Vorgang zuletzt durchgeführt wurde oder `ANALYZE` regelmäßig ausgeführt wurde, überwachen. Die `pg_stat_all_tables` Ansicht enthält Informationen zu jeder Tabelle (`relname`) und darüber, wie viele tote Tupel (`n_dead_tup`) sich in der Tabelle befinden.

Durch die Überwachung der Anzahl toter Tupel in jeder Tabelle, insbesondere in häufig aktualisierten Tabellen, können Sie feststellen, ob die Autovakuum-Prozesse die toten Tupel regelmäßig entfernen, sodass ihr Festplattenspeicher für eine bessere Leistung wiederverwendet werden kann. Sie können die folgende Abfrage verwenden, um die Anzahl der toten Tupel und den Zeitpunkt der letzten Autovakuierung in den Tabellen zu überprüfen:

```
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;
```

Vorteile und Einschränkungen

Autovacuum bietet die folgenden Vorteile:

- Es entfernt automatisch Blähungen aus Tabellen.
- Es verhindert den Umbruch der Transaktions-ID.
- Es hält die Datenbankstatistiken auf dem neuesten Stand.

Einschränkungen:

- Wenn Abfragen Parallelverarbeitung verwenden, reicht die Anzahl der Worker-Prozesse möglicherweise nicht für die automatische Bereinigung aus.
- Wenn Autovacuum zu Spitzenzeiten ausgeführt wird, kann sich die Ressourcenauslastung erhöhen. Sie sollten die Parameter anpassen, um dieses Problem zu lösen.
- Wenn in einer anderen Sitzung Tabellenseiten belegt sind, überspringt Autovacuum diese Seiten möglicherweise.
- Autovacuum kann nicht auf temporäre Tabellen zugreifen.

Manuelles Absaugen und Analysieren von Tabellen

Wenn Ihre Datenbank durch den Autovacuum-Prozess gelöscht wird, empfiehlt es sich, zu vermeiden, dass die gesamte Datenbank zu häufig manuell gelöscht wird. Ein manuelles Löschen kann zu unnötigen I/O Lasten oder CPU-Spitzen führen und eventuell auch keine toten Tupel entfernen. Führen Sie ein manuelles Absaugen nur dann table-by-table aus, wenn es wirklich notwendig ist, z. B. wenn das Verhältnis von lebenden zu toten Tupeln gering ist oder wenn zwischen den Autovakuumvorgängen große Lücken bestehen. Darüber hinaus sollten Sie bei minimaler Benutzeraktivität manuelles Staubsaugen ausführen.

Autovacuum hält auch die Statistiken einer Tabelle auf dem neuesten Stand. Wenn Sie den `ANALYZE` Befehl manuell ausführen, werden diese Statistiken neu erstellt, anstatt sie zu aktualisieren. Das Neuerstellen von Statistiken, wenn sie bereits durch den regulären Autovacuum-Prozess aktualisiert wurden, kann zu einer Auslastung der Systemressourcen führen.

In den folgenden Szenarien wird empfohlen, die Befehle VACUUM und ANALYZE manuell auszuführen:

- Zu schlechten Spitzenzeiten an stark frequentierten Tischen, wenn das automatische Absaugen möglicherweise nicht ausreicht.
- Unmittelbar nach dem Massenladen von Daten in die Zieltabelle. In diesem Fall werden die Statistiken bei `ANALYZE` manueller Ausführung vollständig neu erstellt. Dies ist eine bessere Option, als auf den Beginn der automatischen Bereinigung zu warten.
- Um temporäre Tabellen zu vakuieren (Autovacuum kann nicht auf diese zugreifen).

Um die I/O Auswirkungen der Ausführung der `ANALYZE` Befehle `VACUUM` und auf die gleichzeitige Datenbankaktivität zu verringern, können Sie den `vacuum_cost_delay` Parameter verwenden. In vielen Situationen müssen Wartungsbefehle wie `VACUUM` und `ANALYZE` nicht schnell abgeschlossen werden. Diese Befehle sollten jedoch die Fähigkeit des Systems, andere Datenbankoperationen auszuführen, nicht beeinträchtigen. Um dies zu verhindern, können Sie mithilfe des `vacuum_cost_delay` Parameters kostenabhängige Vakuumverzögerungen aktivieren. Dieser Parameter ist standardmäßig für manuell ausgegebene `VACUUM` Befehle deaktiviert. Um ihn zu aktivieren, setzen Sie ihn auf einen Wert ungleich Null.

Paralleles Ausführen von Saug- und Säuberungsvorgängen

Die `VACUUM` Befehlsoption PARALLEL verwendet parallele Worker für die Phasen Indexsaugung und Indexbereinigung und ist standardmäßig deaktiviert. Die Anzahl der parallelen Worker (der Grad der Parallelität) wird durch die Anzahl der Indizes in der Tabelle bestimmt und kann vom Benutzer angegeben werden. Wenn Sie parallel `VACUUM` Operationen ohne ein Integer-Argument ausführen, wird der Grad der Parallelität anhand der Anzahl der Indizes in der Tabelle berechnet.

Die folgenden Parameter helfen Ihnen bei der Konfiguration von parallelem Vacuuming in Amazon RDS for PostgreSQL und Aurora PostgreSQL-kompatibel:

- max_worker_processes legt die maximale Anzahl gleichzeitiger Worker-Prozesse fest.
- min_parallel_index_scan_size legt die Mindestmenge an Indexdaten fest, die gescannt werden müssen, damit ein parallel Scan berücksichtigt wird.
- max_parallel_maintenance_workers legt die maximale Anzahl parallel Worker fest, die mit einem einzigen Utility-Befehl gestartet werden können.

> **Note**
>
> Die `PARALLEL` Option wird nur für Vakuumzwecke verwendet. Sie hat keinen Einfluss auf den Befehl. `ANALYZE`

Das folgende Beispiel veranschaulicht das Datenbankverhalten, wenn Sie manuell `VACUUM` und in `ANALYZE` einer Datenbank verwenden.

Hier ist eine Beispieltabelle, in der Autovacuum deaktiviert wurde (nur zur Veranschaulichung; die Deaktivierung des Autovakuums wird nicht empfohlen):

```
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
```

Fügen Sie der Tabelle t1 1 Million Zeilen hinzu:

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

Statistik der Tabelle 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
```

Fügen Sie einen Index hinzu:

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

Führen Sie den EXPLAIN Befehl aus (Plan 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)
```

Führen Sie den EXPLAIN ANALYZE Befehl aus (Plan 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)
```

Die EXPLAIN ANALYZE Befehle EXPLAIN und zeigen unterschiedliche Pläne an, da Autovacuum für die Tabelle deaktiviert war und der ANALYZE Befehl nicht manuell ausgeführt wurde. Lassen Sie uns nun einen Wert in der Tabelle aktualisieren und den Plan neu generieren: EXPLAIN ANALYZE

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

Der EXPLAIN ANALYZE Befehl (Plan 3) zeigt jetzt an:

```
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
```

Wenn Sie die Kosten zwischen Plan 2 und Plan 3 vergleichen, werden Sie die Unterschiede in der Planungs- und Ausführungszeit feststellen, da wir noch keine Statistiken gesammelt haben.

Lassen Sie uns nun ein Handbuch `ANALYZE` für die Tabelle ausführen, dann die Statistiken überprüfen und den Plan neu generieren:

```
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
```

Führen Sie den `EXPLAIN ANALYZE` Befehl aus (Plan 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
```

Wenn Sie alle Planergebnisse vergleichen, nachdem Sie die Tabelle manuell analysiert und Statistiken gesammelt haben, werden Sie feststellen, dass Plan 4 des Optimierers besser ist als die anderen und auch die Ausführungszeit der Abfrage verkürzt. Dieses Beispiel zeigt, wie wichtig es ist, Wartungsaktivitäten in der Datenbank auszuführen.

Eine ganze Tabelle mit VACUUM FULL neu schreiben

Wenn `VACUUM` Sie den Befehl mit dem `FULL` Parameter ausführen, wird der gesamte Inhalt einer Tabelle in eine neue Festplattendatei ohne zusätzlichen Speicherplatz umgeschrieben und ungenutzter Speicherplatz an das Betriebssystem zurückgegeben. Dieser Vorgang ist viel langsamer und erfordert eine `ACCESS EXCLUSIVE` Sperre für jede Tabelle. Außerdem benötigt er zusätzlichen Speicherplatz, da er eine neue Kopie der Tabelle schreibt und die alte Kopie erst freigibt, wenn der Vorgang abgeschlossen ist.

`VACUUM FULL`kann in den folgenden Fällen nützlich sein:

- Wenn Sie eine beträchtliche Menge an Speicherplatz aus den Tabellen zurückgewinnen möchten.
- Wenn Sie in Tabellen, bei denen es sich nicht um Primärschlüsseltabellen handelt, aufgeblähten Speicherplatz zurückgewinnen möchten.

Wir empfehlen die Verwendung, `VACUUM FULL` wenn Sie Tabellen haben, die keine Primärschlüsseltabellen haben, wenn Ihre Datenbank Ausfallzeiten tolerieren kann.

Da mehr Sperren `VACUUM FULL` erforderlich sind als bei anderen Vorgängen, ist die Ausführung in wichtigen Datenbanken teurer. Um diese Methode zu ersetzen, können Sie die `pg_repack` Erweiterung verwenden, die im nächsten Abschnitt beschrieben wird. Diese Option ähnelt der Option, erfordert `VACUUM FULL` jedoch nur minimale Sperren und wird sowohl von Amazon RDS for PostgreSQL als auch von Aurora PostgreSQL-Compatible unterstützt.

Blähungen mit pg_repack entfernen

Sie können die `pg_repack` Erweiterung verwenden, um Tabellen- und Indexaufblähungen mit minimaler Datenbanksperre zu entfernen. Sie können diese Erweiterung in der Datenbank-Instance erstellen und den `pg_repack` Client (wobei die Client-Version der Erweiterungsversion entspricht) von Amazon Elastic Compute Cloud (Amazon EC2) oder von einem Computer aus ausführen, der eine Verbindung zu Ihrer Datenbank herstellen kann.

Im `VACUUM FULL` Gegensatz `pg_repack` dazu erfordert sie weder Ausfallzeiten noch ein Wartungsfenster und blockiert keine anderen Sitzungen.

`pg_repack`ist hilfreich in Situationen`VACUUM FULL`, in denen`CLUSTER`, oder `REINDEX` möglicherweise nicht funktioniert. Es erstellt eine neue Tabelle, die die Daten der aufgeblähten Tabelle enthält, verfolgt die Änderungen gegenüber der Originaltabelle und ersetzt dann die Originaltabelle durch die neue. Es sperrt die Originaltabelle nicht für Lese- oder Schreibvorgänge, während die neue Tabelle erstellt wird.

Sie können es `pg_repack` für eine vollständige Tabelle oder für einen Index verwenden. Eine Liste der Aufgaben finden Sie in der pg_repack-Dokumentation.

Einschränkungen:

- Um ausgeführt zu werden`pg_repack`, muss Ihre Tabelle einen Primärschlüssel oder einen eindeutigen Index haben.
- `pg_repack` funktioniert nicht mit temporären Tabellen.
- `pg_repack`funktioniert nicht bei Tabellen mit globalen Indizes.
- Wenn in Bearbeitung `pg_repack` ist, können Sie keine DDL-Operationen für Tabellen ausführen.

In der folgenden Tabelle werden die Unterschiede zwischen `pg_repack` und `VACUUM FULL` beschrieben.

`VACUUM FULL`	`pg_repack`
Eingebauter Befehl	Eine Erweiterung, die Sie von Amazon EC2 oder Ihrem lokalen Computer aus ausführen

Erfordert eine `ACCESS EXCLUSIVE` Sperre, während es an einer Tabelle arbeitet	Erfordert eine `ACCESS EXCLUSIVE` Sperre nur für kurze Zeit
Funktioniert mit allen Tabellen	Funktioniert nur bei Tabellen mit Primär- und Einzelschlüsseln
Benötigt doppelt so viel Speicherplatz wie die Tabelle und die Indizes	Benötigt doppelt so viel Speicherplatz wie die Tabelle und die Indizes

Verwenden Sie `pg_repack` den folgenden Befehl, um eine Tabelle auszuführen:

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

Verwenden Sie `pg_repack` den folgenden Befehl, um einen Index zu verwenden:

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

Weitere Informationen finden Sie im AWS Blogbeitrag Remove Bloat from Amazon Aurora and RDS for PostgreSQL with pg_repack.

Vorbehalt

Der `error-on-invalid-index` Fehler in bedeutet `pg_repack` normalerweise, dass ein oder mehrere Indizes in der Tabelle beschädigt oder ungültig sind. `pg_repack`kann nicht sicher mit Tabellen arbeiten, die ungültige Indizes haben, da die Datenkonsistenz während des Umpackvorgangs von den Indizes abhängt.
Dieser Fehler tritt auf, wenn:

- Der Index ist als ungültig markiert (z. B. aufgrund einer fehlgeschlagenen `CREATE INDEX CONCURRENTLY` Anweisung).
- Der Index ist beschädigt (möglicherweise aufgrund von Hardwareproblemen oder abrupten Abschaltungen).

Verwenden Sie die folgende Abfrage, um ungültige Indizes zu identifizieren und sie zuerst zu löschen, wenn Sie sie finden.

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

Indizes neu erstellen

Der PostgreSQL-Befehl REINDEX erstellt einen Index neu, indem er die Daten verwendet, die in der Indextabelle gespeichert sind, und ersetzt die alte Kopie des Indexes. Wir empfehlen die Verwendung `REINDEX` in den folgenden Szenarien:

- Wenn ein Index beschädigt wird und keine gültigen Daten mehr enthält. Dies kann auf Software- oder Hardwarefehler zurückzuführen sein.
- Wenn Abfragen, die zuvor den Index verwendet haben, ihn nicht mehr verwenden.
- Wenn der Index mit einer großen Anzahl leerer oder fast leerer Seiten aufgebläht wird. Sie sollten den Vorgang ausführen`REINDEX`, wenn der Prozentsatz der aufgeblähten Menge (`bloat_pct`) größer als 20 ist.

Die folgende Abfrage hilft Ihnen bei der Suche `bloat_pct` nach:

```
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;
```

Indexseiten, die vollständig leer sind, werden zur Wiederverwendung zurückgewonnen. Wir empfehlen jedoch eine regelmäßige Neuindizierung, wenn die Indexschlüssel auf einer Seite gelöscht wurden, aber noch Speicherplatz zugewiesen ist.

Die Neuerstellung des Index trägt zu einer besseren Abfrageleistung bei. Sie können einen Index auf drei Arten neu erstellen, wie in der folgenden Tabelle beschrieben.

Methode	Beschreibung	Einschränkungen
`CREATE INDEX` und `DROP INDEX` mit der Option `CONCURRENTLY`	Erzeugt einen neuen Index und entfernt den alten Index. Der Optimierer generiert Pläne, indem er den neu erstellten Index anstelle des alten Index verwendet. Zu Zeiten mit geringer Auslastung können Sie den alten Index löschen.	Die Indexerstellung nimmt mehr Zeit in Anspruch, wenn Sie `CONCURRENTLY` diese Option verwenden, da sie alle eingehenden Änderungen verfolgen muss. Wenn Änderungen eingefroren sind, wird der Vorgang als abgeschlossen markiert.
`REINDEX` mit der `CONCURRENTLY` Option	Sperrt Schreibvorgänge während des Neuaufbauvorgangs. PostgreSQL Version 12 und spätere Versionen bieten die `CONCURRENTLY` Option, mit der diese Sperren vermieden werden.	Die Verwendung `CONCURRENTLY` erfordert eine längere Zeit, um den Index neu zu erstellen.

`pg_repack` Erweiterung	Löscht den Bloat aus einer Tabelle und baut den Index neu auf.	Sie müssen diese Erweiterung von einer EC2 Instanz oder Ihrem lokalen Computer aus ausführen, der mit der Datenbank verbunden ist.

Einen neuen Index erstellen

Wenn die `CREATE INDEX` Befehle `DROP INDEX` und zusammen verwendet werden, wird ein Index neu erstellt:

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

Der Nachteil dieses Ansatzes ist die ausschließliche Sperre der Tabelle, was sich negativ auf die Leistung während dieser Aktivität auswirkt. Der `DROP INDEX` Befehl erwirbt eine exklusive Sperre, die sowohl Lese- als auch Schreibvorgänge in der Tabelle blockiert. Der `CREATE INDEX` Befehl blockiert die Schreiboperationen in der Tabelle. Es ermöglicht Lesevorgänge, die jedoch bei der Indexerstellung teuer sind.

Einen Index neu aufbauen

Der `REINDEX` Befehl hilft Ihnen dabei, eine konsistente Datenbankleistung aufrechtzuerhalten. Wenn Sie eine große Anzahl von DML-Vorgängen an einer Tabelle ausführen, führen diese zu einer Aufblähung der Tabelle und des Indexes. Indizes werden verwendet, um das Nachschlagen in Tabellen zu beschleunigen und so die Abfrageleistung zu verbessern. Ein aufgeblähter Index wirkt sich auf Suchvorgänge und die Abfrageleistung aus. Aus diesem Grund empfehlen wir, Tabellen mit einem hohen Volumen an DML-Vorgängen neu zu indizieren, um eine konsistente Abfrageleistung zu gewährleisten.

`REINDEX`Mit dem Befehl wird der Index von Grund auf neu erstellt, indem die Schreibvorgänge in der zugrunde liegenden Tabelle gesperrt werden. Lesevorgänge für die Tabelle sind jedoch zulässig. Es blockiert jedoch die Lesevorgänge für den Index. Abfragen, die den entsprechenden Index verwenden, werden blockiert, andere Abfragen jedoch nicht.

PostgreSQL Version 12 führte einen neuen optionalen Parameter ein`CONCURRENTLY`, der den Index von Grund auf neu erstellt, aber die Schreib- oder Lesevorgänge für die Tabelle oder für Abfragen,

die den Index verwenden, nicht sperrt. Wenn Sie diese Option verwenden, dauert es jedoch länger, bis der Vorgang abgeschlossen ist.

Beispiele

Einen Index erstellen und löschen

Erstellen Sie einen neuen Index mit der `CONCURRENTLY` Option:

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

Löschen Sie den alten Index mit der `CONCURRENTLY` Option:

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

Einen Index neu aufbauen

So erstellen Sie einen einzelnen Index neu:

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

Um alle Indizes in einer Tabelle neu aufzubauen:

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

Um alle Indizes in einem Schema neu aufzubauen:

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

Gleichzeitiger Neuaufbau eines Indexes

So erstellen Sie einen einzelnen Index neu:

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

Um alle Indizes in einer Tabelle neu aufzubauen:

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

Um alle Indizes in einem Schema neu aufzubauen:

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

Nur Indizes neu erstellen oder verschieben

So erstellen Sie einen einzelnen Index neu:

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

Um alle Indizes neu aufzubauen:

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

Beispiel: Rückgewinnung von Speicherplatz mithilfe von Autovacuum und VACUUM FULL

Lassen Sie uns als Beispiel eine emp Tabellentabelle mit 500.000 Zeilen erstellen und dann die Zeilen mit neuen Werten aktualisieren. Autovacuum ist aktiviert, sodass sowohl VACUUM Befehle als auch ANALYZE Befehle für diese Tabelle ausgeführt werden, um Blähungen zu entfernen und Speicherplatz zurückzugewinnen. Der zurückgewonnene Speicherplatz kann wiederverwendet werden, wird aber nicht an das Betriebssystem zurückgegeben.

Die folgende Abfrage bestimmt, wie groß die Tabelle ist:

```
-- 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;
```

Die Ergebnisse der Abfrage zeigen, dass die Tabelle eine Aufblähung von etwa 51 Prozent aufweist:

```
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
```

Hier sind die Statistiken aus der `pg_stat_all_tables` Ansicht:

```
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
```

Beachten Sie, dass Autovacuum die `last_autoanalyze` Spalten `last_autovacuum` und nach der Ausführung aktualisiert hat.

Fügen wir nun einige Zeilen in die Tabelle ein und überprüfen wir`extra_size(bloat_size)`, ob der leere Bereich auch als aufgebläht angesehen wird.

```
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)
```

Die `bloat_pct` Spalte in der Ausgabe gibt an, dass der bereinigte Bereich durch neue Einfügungen belegt wurde. Lass uns laufen`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)
```

Aus dieser Ausgabe können Sie sehen, dass der leere Speicherplatz und die Aufblähung entfernt wurden und der Speicherplatz an das Betriebssystem zurückgegeben wurde.

> **Note**
>
> Stattdessen könnten Sie rennen`VACUUM FULL`, `pg_repack` um dieselben Ergebnisse zu erzielen.

Ressourcen

- Grundlegendes zu Autovacuum in Amazon RDS for PostgreSQL PostgreSQL-Umgebungen (Blogbeitrag)AWS
- Automatisches Staubsaugen (PostgreSQL-Dokumentation)
- Zuweisen von Speicher für Autovacuum (Amazon RDS-Dokumentation)
- Verhinderung von Transaktions-ID-Wraparound-Fehlern; (PostgreSQL-Dokumentation)
- Entfernen Sie Bloat aus Amazon Aurora und RDS für PostgreSQL mit pg_repack (Blogbeitrag)AWS

Dokumentverlauf

In der folgenden Tabelle werden wichtige Änderungen in diesem Leitfaden beschrieben. Um Benachrichtigungen über zukünftige Aktualisierungen zu erhalten, können Sie einen RSS-Feed abonnieren.

Änderung	Beschreibung	Datum
Aktualisierungen	Es wurden Fehler behoben und Informationen zu den Abschnitten Automatisches Löschen und Analysieren von Tabellen und Entfernen von Blähungen mit pg_repack hinzugefügt.	22. August 2025
Die `reindex` Syntax wurde korrigiert	Im Abschnitt zur gleichzeitigen Neuerstellung eines Indexes wurden die `reindex` Beispiele korrigiert.	30. Juni 2025
Erste Veröffentlichung	—	22. Dezember 2023

Die vorliegende Übersetzung wurde maschinell erstellt. Im Falle eines Konflikts oder eines Widerspruchs zwischen dieser übersetzten Fassung und der englischen Fassung (einschließlich infolge von Verzögerungen bei der Übersetzung) ist die englische Fassung maßgeblich.