26 MARCH 2024

Bloat of internal tables due to frequently created and dropped relations

Overview

Tables that are created and dropped on demand, whether they are temporary or regular, are frequently used by application developers in PostgreSQL to simplify the implementation of various functionalities and to expedite responses. Numerous articles on the internet describe the advantages of using such tables for storing search results, precalculating figures for reports, importing data from external files, and more. One can even define a TEMP TABLE with the condition ON COMMIT DROP, allowing the system to clean up automatically. However, like most things, this solution has potential drawbacks, because size matters. A solution that functions smoothly for dozens of parallel sessions may suddenly begin to cause unexpected issues if the application is used by hundreds or thousands of users simultaneously during peak hours. Frequently creating and dropping tables and related objects, can cause significant bloat of certain PostgreSQL system tables. This is a well-known problem that many articles mention, but they often lack detailed explanations and quantification of the impact. Several pg_catalog system tables can become significantly bloated. Table pg_attribute is the most affected, followed by pg_attrdef and pg_class.

What is the main issue with the bloating of system tables?

We already encountered this issue in the PostgreSQL logs of one of our clients. When the bloat of system tables became too extensive, PostgreSQL decided to reclaim free space during an autovacuum operation. This action caused exclusive locks on the table and blocked all other operations for several seconds. PostgreSQL was unable to read information about the structures of all relations. And as a result, even the simplest select operations had to be delayed until the operation was resolved. This is, of course, an extreme and rare scenario that can only occur under exceptionally high load. Nevertheless, it's important to be aware of it and be able to assess if it could also happen to our database.

Example of reporting table in accounting software

Let's examine the impact of these short-lived relations on PostgreSQL system tables using two different examples. The first is a comprehensive example of TEMP TABLE where we will explain all the details, and the second is for benchmarking purposes. Our first example involves an imaginary accounting software that generates a wide variety of reports, many of which require some precalculation of results. The use of temporary tables for these purposes is a fairly obvious design choice. We will discuss one such example — a temporary pivot table for a report storing monthly summaries for an entire year, with one row per client_id:

```
CREATE TEMP TABLE pivot_temp_table (
   id serial PRIMARY KEY,
   inserted_at timestamp DEFAULT current_timestamp,
   client_id INTEGER,
   name text NOT NULL,
   address text NOT NULL,
   loyalty_program BOOLEAN DEFAULT false,
   loyalty_program_start TIMESTAMP,
   orders_202301_count_of_orders INTEGER DEFAULT 0,
   orders_202301_total_price NUMERIC DEFAULT 0,
   ...
   orders_202312_count_of_orders INTEGER DEFAULT 0,
   orders_202312_total_price NUMERIC DEFAULT 0);
```

We also want to create some indexes because some results can be quite huge:

```
CREATE INDEX pivot_temp_table_idx1 ON pivot_temp_table (client_id);
CREATE INDEX pivot_temp_table_idx2 ON pivot_temp_table (name);
CREATE INDEX pivot_temp_table_idx3 ON pivot_temp_table (loyalty_program);
CREATE INDEX pivot_temp_table_idx4 ON pivot_temp_table (loyalty_program_start);
```

Summary of the created objects:

- A temporary table, pivot_temp_table, with 31 columns, 27 of which have default values.
- Some of the columns are of the TEXT data type, resulting in the automatic creation of a TOAST table.
- The TOAST table requires an index on chunk_id and chunk_seq.
- The ID is the primary key, meaning a unique index on ID was automatically created.
- The ID is defined as SERIAL, leading to the automatic creation of a sequence, which is essentially another table with a special structure.
- We also defined four additional indexes on our temporary table.

Let's now examine how these relations are represented in PostgreSQL system tables.

Table pg_attribute

The pg_attribute table stores the attributes (columns) of all relations. PostgreSQL will insert a total of 62 rows into the pg_attribute table:

- Each row in our pivot_temp_table contains six hidden columns (tableoid, cmax, xmax, cmin, xmin, ctid) and 31 'normal' column. This totals to 37 rows inserted for the main temp table.
- Indexes will add one row for each column used in the index, equating to five rows in our case.
- A TOAST table was automatically created. It has six hidden columns and three normal columns (chunk_id, chunk_seq, chunk_data), and one index on chunk_id and chunk_seq, adding up to 11 rows in total.
- A sequence for the ID was created, which is essentially another table with a predefined structure. It has six hidden columns and three normal columns (last_value, log_cnt, is_called), adding another nine rows.

Table pg_attrdef

The pg_attrdef table stores default values for columns. Our main table contains many default values, resulting in the creation of 27 rows in this table. We can examine their content using a query:

```
SELECT
   c.relname as table_name,
   o.rolname as table_owner,
   c.relkind as table_type,
   a.attname as column_name,
   a.attnum as column_number,
   a.atttypid::regtype as column_data_type,
   pg_get_expr(adbin, adrelid) as sql_command
FROM pg_attrdef ad
JOIN pg_attribute a ON ad.adrelid = a.attrelid AND ad.adnum = a.attnum
JOIN pg_class c ON c.oid = ad.adrelid
JOIN pg_authid o ON o.oid = c.relowner
WHERE c.relname = 'pivot_temp_table'
ORDER BY table_name, column_number;
```

Our output:

```
    table_name    | table_owner | table_type |         column_name          | column_number |      column_data_type       |
sql_command
------------------+-------------+------------+------------------------------+---------------+-----------------------------+--
--------------------------------------------
 pivot_temp_table | postgres    | r          | id                           | 1             | integer                     |
nextval('pivot_temp_table_id_seq'::regclass)
 pivot_temp_table | postgres    | r          | inserted_at                  | 2             | timestamp without time zone |
CURRENT_TIMESTAMP
 pivot_temp_table | postgres    | r          | loyalty_program              | 6             | boolean                     |
false
 pivot_temp_table | postgres    | r          | orders_202301_count_of_orders | 8            | integer                     | 0
 pivot_temp_table | postgres    | r          | orders_202301_total_price    | 9             | numeric                     | 0
--> up to the column "orders_202312_total_price"
```

Table pg_class

The pg_class table stores primary information about relations. This example will create nine rows: one for the temp table, one for the toast table, one for the toast table index, one for the ID primary key unique index, one for the sequence, and four for the custom indexes.

Summary of this example

Our first example produced a seemingly small number of rows – 62 in pg_attribute, 27 in pg_attrdef, and 9 in pg_class. These are very low numbers, and if such a solution was used by only one company, we would hardly see any problems. But consider a scenario where a company hosts accounting software for small businesses and hundreds or even thousands of users use the app during peak hours. In such a situation, many temp tables and related objects would be created and dropped at a relatively quick pace. In the pg_attribute table, we could see anywhere from a few thousand to even hundreds of thousands of records inserted and then deleted over several hours. However, this is still a relatively small use case. Let’s now imagine and benchmark something even larger.

Example of online shop

Let’s conduct deeper analysis using a more relatable and heavier example. Imagine an online retailer selling clothing, shoes, and other accessories. When a user logs into the shop, the database automatically creates some user-specific tables. These are later deleted by a dedicated process after a certain period of user inactivity. These relations are created to speed up the system’s responses to a specific user. Repeated selects from the main tables would be much slower, even though the main tables are partitioned by days, these partitions can be enormous. For our example, we don’t need to discuss the layout of sessions, nor whether the tables are created as temporary or regular ones, as both have the same impact on PostgreSQL system tables. We will also omit all other aspects of real-life implementation. This example is purely theoretical, inspired by design patterns discussed on the internet, and is not based on any real system. It should not be understood as a design recommendation. In fact, as we will see, this example would more likely serve as an anti-pattern.

1. The "session_events" table stores selected actions performed by the user during the session. Events are collected for each action the user takes on the website, so there are at least hundreds, but more often thousands of events recorded from one session. These are all sent in parallel into the main event table. However, the main table is enormous. Therefore, this user-specific table stores only some events, allowing for quick analysis of recent activities, etc. The table has 25 different columns, some of which are of the TEXT type and one column of the JSONB type – which means a TOAST table with one index was created. The table has a primary key of the serial type, indicating the order of actions – i.e., one unique index, one sequence, and one default value were created. There are no additional default values. The table also has three additional indexes for quicker access, each on one column. Their benefit could be questionable, but they are part of the implementation.

    - Summary of rows in system tables – pg_attribute – 55 rows, pg_class – 8 rows, pg_attrdef – 1 row

2. The "last_visited" table stores a small subset of events from the "session_events" table to quickly show which articles the user has visited during this session. Developers chose to implement it this way for convenience. The table is small, containing only 10 columns, but at least one is of the TEXT type. Therefore, a TOAST table with one index was created. The table has a primary key of the TIMESTAMP type, therefore it has one unique index, one default value, but no sequence. There are no additional indexes.

    - Rows in system tables – pg_attribute – 28 rows, pg_class – 4 rows, pg_attrdef – 1 row

3. The "last_purchases" table is populated at login from the main table that stores all purchases. This user-specific table contains the last 50 items purchased by the user in previous sessions and is used by the recommendation algorithm. This table contains fully denormalized data to simplify their processing and visualization, and therefore it has 35 columns. Many of these columns are of the TEXT type, so a TOAST table with one index was created. The primary key of this table is a combination of the purchase timestamp and the ordinal number of the item in the order, leading to the creation of one unique index but no default values or sequences. Over time, the developer created four indexes on this table for different sorting purposes, each on one column. The value of these indexes can be questioned, but they still exist.

    - Rows in system tables – pg_attribute – 57 rows, pg_class – 8 rows

4. The "selected_but_not_purchased" table is populated at login from the corresponding main table. It displays the last 50 items still available in the shop that the user previously considered purchasing but later removed from the cart or didn't finish ordering at all, and the content of the cart expired. This table is used by the recommendation algorithm and has proven to be a successful addition to the marketing strategy, increasing purchases by a certain percentage. The table has the same structure and related objects as "last_purchases". Data are stored separately from purchases to avoid mistakes in data interpretation and also because this part of the algorithm was implemented much later.

    - Rows in system tables – pg_attribute – 57 rows, pg_class – 8 rows

5. The "cart_items" table stores items selected for purchase in the current session but not yet bought. This table is synchronized with the main table, but a local copy in the session is also maintained. The table contains normalized data, therefore it has only 15 columns, some of which are of the TEXT type, leading to the creation of a TOAST table with one index. It has a primary key ID of the UUID type to avoid collisions across all users, resulting in the creation of one unique index and one default value, but no sequence. There are no additional indexes.

    - Rows in system tables – pg_attribute – 33 rows, pg_class – 4 rows, pg_attrdef – 1 row

The creation of all these user-specific tables results in the insertion of the following number of rows into PostgreSQL system tables – pg_attribute: 173 rows, pg_class: 32 rows, pg_attrdef: 3 rows.

Analysis of traffic

As the first step we provide an analysis of the business use case and traffic seasonality. Let's imagine our retailer is active in several EU countries and targets mainly people from 15 to 35 years old. The online shop is relatively new, so it currently has 100,000 accounts. Based on white papers available on the internet, we can presume the following user activity:

Level of user's activity	Ratio of users [%]	Total count of users	Frequency of visits on page
very active	10%	10,000	2x to 4x per week
normal activity	30%	30,000	~1 time per week
low activity	40%	40,000	1x to 2x per month
almost no activity	20%	20,000	few times in year

Since this is an online shop, traffic is highly seasonal. Items are primarily purchased by individuals for personal use. Therefore, during the working day, they check the shop at very specific moments, such as during travel or lunchtime. The main peak in traffic during the working day is between 7pm and 9pm. Fridays usually have much lower traffic, and the weekend follows suit. The busiest days are generally at the end of the month, when people receive their salaries. The shop experiences the heaviest traffic on Thanksgiving Thursday and Black Friday. The usual practice in recent years is to close the shop for an hour or two and then reopen at a specific hour with reduced prices. Which translates into huge number of relations being created and later deleted at relatively short time. The duration of a user's connection can range from just a few minutes up to half an hour. User-specific tables are created when user logs into shop. They are later deleted by a special process that uses a sophisticated algorithm to determine whether relations already expired or not. This process involves various criteria and runs at distinct intervals, so we can see a large number of relations deleted in one run. Let's quantify these descriptions:

Traffic on different days	Logins per 30 min	pg_attribute [rows]	pg_class [rows]	pg_attrdef [rows]
Numbers from analysis per 1 user	1	173	32	3
Average traffic in the afternoon	1,000	173,000	32,000	3,000
Normal working day evening top traffic	3,000	519,000	96,000	9,000
Evening after salary low traffic	8,000	1,384,000	256,000	24,000
Evening after salary high traffic	15,000	2,595,000	480,000	45,000
Singles' Day evening opening	40,000	6,920,000	1,280,000	120,000
Thanksgiving Thursday evening opening	60,000	10,380,000	1,920,000	180,000
Black Friday evening opening	50,000	8,650,000	1,600,000	150,000
Black Friday weekend highest traffic	20,000	3,460,000	640,000	60,000
Theoretical maximum – all users connected	100,000	17,300,000	3,200,000	300,000

Now we can see what scalability means. Our solution will definitely work reasonably on normal days. However, traffic in the evenings after people receive their salaries can be very heavy. Thanksgiving Thursday and Black Friday really test its limits. Between 1 and 2 million user-specific tables and related objects will be created and deleted during these evenings. And what happens if our shop becomes even more successful and the number of accounts grows to 500 000, 1 million or more? The solution would definitely hit the limits of vertical scaling at some points, and we would need to think about ways to scale it horizontally.

How to examine bloat

Analysis of traffic provided some theoretical numbers. But we need to check the real-time situation in our database. First, if we're unsure about what's happening in our system regarding the creation and deletion of relations, we can temporarily switch on extended logging. We can set 'log_statements' to at least 'ddl' to see all CREATE/ ALTER /DROP commands. To monitor long running vacuum actions we can set 'log_autovacuum_min_duration' to some reasonable low number like 2 seconds. These settings are both dynamic and do not require a restart. However, this change may increase disk IO on local servers due to the increased writes into PostgreSQL logs. On cloud databases or Kubernetes clusters, log messages are usually sent to a separate subsystem and stored independently of the database disk, so the impact should be minimal. To check existing bloats in PostgreSQL tables, we can use the 'pgstattuple' extension. This extension only creates new functions; it does not influence the performance of the database. It can only cause reads when we invoke some of its functions. By using its functions in combination with results from other PostgreSQL system objects, we can get a better picture of the bloat in the PostgreSQL system tables. The pg_relation_size function was added to double-check the numbers from the pgstattuple function.

```
WITH tablenames AS (SELECT tablename FROM (VALUES('pg_attribute'),('pg_attrdef'),('pg_class')) as t(tablename))
SELECT
   tablename,
   now() as checked_at,
   pg_relation_size(tablename) as relation_size,
   pg_relation_size(tablename) / (8*1024) as relation_pages,
   a.*,
   s.*
FROM tablenames t
JOIN LATERAL (SELECT * FROM pgstattuple(t.tablename)) s ON true
JOIN LATERAL (SELECT last_autovacuum, last_vacuum, last_autoanalyze, last_analyze, n_live_tup, n_dead_tup
FROM pg_stat_all_tables WHERE relname = t.tablename) a ON true
ORDER BY tablename
```

We will get output like this one (result is shown only for 1 table)

```
 tablename         | pg_attribute
 checked_at        | 2024-02-18 10:46:34.348105+00
 relation_size     | 44949504
 relation_pages    | 5487
 last_autovacuum   | 2024-02-16 20:07:15.7767+00
 last_vacuum       | 2024-02-16 20:55:50.685706+00
 last_autoanalyze  | 2024-02-16 20:07:15.798466+00
 last_analyze      | 2024-02-17 22:05:43.19133+00
 n_live_tup        | 3401
 n_dead_tup        | 188221
 table_len         | 44949504
 tuple_count       | 3401
 tuple_len         | 476732
 tuple_percent     | 1.06
 dead_tuple_count  | 107576
 dead_tuple_len    | 15060640
 dead_tuple_percent| 33.51
 free_space        | 28038420
 free_percent      | 62.38
```

If we attempt some calculations, we'll find that the summary of numbers from the pgstattuple function does not match the total relation size. Also, the percentages usually don't add up to 100%. We need to understand these values as estimates, but they still provide a good indication of the scope of the bloat. We can easily modify this query for monitoring purposes. We should certainly monitor at least the relation_size, n_live_tup, and n_dead_tup for these system tables. To run monitoring under a non-superuser account, this account must have been granted or inherited PostgreSQL predefined roles 'pg_stat_scan_tables' or 'pg_monitor'. If we want to dig deeper into the problem and make some predictions, we can, for example, check how many tuples are stored per page in a specific table. With these numbers, we would be able to estimate possible bloat in critical moments. We can use a query like this one:

```
WITH pages AS (
   SELECT * FROM generate_series(0, (SELECT pg_relation_size('pg_attribute') / 8192) -1) as pagenum),
tuples_per_page AS (
   SELECT pagenum, nullif(sum((t_xmin is not null)::int), 0) as tuples_per_page
   FROM pages JOIN LATERAL (SELECT * FROM heap_page_items(get_raw_page('pg_attribute',pagenum))) a ON true
   GROUP BY pagenum)
SELECT
   count(*) as pages_total,
   min(tuples_per_page) as min_tuples_per_page,
   max(tuples_per_page) as max_tuples_per_page,
   round(avg(tuples_per_page),0) as avg_tuples_per_page,
   mode() within group (order by tuples_per_page) as mode_tuples_per_page
FROM tuples_per_page
```

Output will look like this:

```
 pages_total         | 5487
 min_tuples_per_page | 1
 max_tuples_per_page | 55
 avg_tuples_per_page | 23
 mode_tuples_per_page | 28
```

Here, we can see that in our pg_attribute system table, we have an average of 23 tuples per page. So now we can calculate theoretical increase in size of this table for different traffic. Typical size of this table is usually only few hundreds of MBs. So theoretical bloat about 3 GB during Black Friday days is quite significant number for this table.

Logins	pg_attribute rows	data pages	size in MB
1	173	8	0.06
1,000	173,000	7,522	58.77
3,000	519,000	22,566	176.30
15,000	2,595,000	112,827	881.46
20,000	3,460,000	150,435	1,175.27
60,000	10,380,000	451,305	3,525.82
100,000	17,300,000	752,174	5,876.36

Summary

We've presented a reporting example from accounting software and an example of user-specific tables from an online shop. While both are theoretical, the idea is to illustrate patterns. We also discussed the influence of high traffic seasonality on the number of inserts and deletes in system tables. We provided an example of an extremely increased load in an online shop on big sales days. We believe the results of the analysis warrant attention. It's also important to remember that the already heavy situation in these peak moments can be even more challenging if our application is running on an instance with low disk IOPS. All these new objects would cause writes into WAL logs and synchronization to the disk. In the case of low disk throughput, there could be significant latency, and many operations could be substantially delayed. So, what's the takeaway from this story? First of all, PostgreSQL autovacuum processes are designed to minimize the impact on the system. If the autovacuum settings on our database are well-tuned, in most cases, we won't see any problems. However, if these settings are outdated, tailored for much lower traffic, and our system is under unusually heavy load for an extended period, creating and dropping thousands of tables and related objects in a relatively short time, PostgreSQL system tables can eventually become significantly bloated. This will already slow down system queries reading details about all other relations. And at some point, the system could decide to shrink these system tables, causing an Exclusive lock on some of these relations for seconds or even dozens of seconds. This could block a large number of selects and other operations on all tables. Based on analysis of traffic, we can conduct a similar analysis for other specific systems to understand when they will be most susceptible to such incidents. But having effective monitoring is absolutely essential.

Resources

1. Understanding an outage: concurrency control & vacuuming in PostgreSQL
2. Stackoverflow – temporary tables bloating pg_attribute
3. Diagnosing table and index bloat
4. What are the peak times for online shopping?
5. PostgreSQL Tuple-Level Statistics With pgstattuple

JM

ABOUT THE AUTHOR

Josef Machytka

View posts