04 DECEMBER 2024

Unique Constraint Violations During Inserts Cause Bloat in PostgreSQL

The issue of table and index bloat due to failed inserts on unique constraints is well known and has been discussed in various articles across the internet. However, these discussions sometimes lack a clear, practical example with measurements to illustrate the impact. And despite the familiarity of this issue, we still frequently see this design pattern—or rather, anti-pattern—in real-world applications. Developers often rely on unique constraints to prevent duplicate values from being inserted into tables. While this approach is straightforward, versatile, and generally considered effective, in PostgreSQL, inserts that fail due to unique constraint violations unfortunately always lead to table and index bloat. And on high-traffic systems, this unnecessary bloat can significantly increase disk I/O and the frequency of autovacuum runs. In this article, we aim to highlight this problem once again and provide a straightforward example with measurements to illustrate it. We suggest simple improvement that can help mitigate this issue and reduce autovacuum workload and disk I/O.

Two Approaches to Duplicate Prevention

In PostgreSQL, there are two main ways to prevent duplicate values using unique constraints:

1. Standard Insert Command (INSERT INTO table)

The usual INSERT INTO table command attempts to insert data directly into the table. If the insert would result in a duplicate value, it fails with a "duplicate key value violates unique constraint" error. Since the command does not specify any duplicate checks, PostgreSQL internally immediately inserts the new row and only then begins updating indexes. When it encounters a unique index violation, it triggers the error and deletes the newly added row. The order of index updates is determined by their relation IDs, so the extent of index bloat depends on the order in which indexes were created. With repeated "unique constraint violation" errors, both the table and some indexes accumulate deleted records leading to bloat, and the resulting write operations increase disk I/O without achieving any useful outcome.

2. Conflict-Aware Insert (INSERT INTO table … ON CONFLICT DO NOTHING)

The INSERT INTO table ON CONFLICT DO NOTHING command behaves differently. Since it specifies that a conflict might occur, PostgreSQL first checks for potential duplicates before attempting to insert data. If a duplicate is found, PostgreSQL performs the specified action—in this case, "DO NOTHING"—and no error occurs. This clause was introduced in PostgreSQL 9.5, but some applications either still run on older PostgreSQL versions or retain legacy code when the database is upgraded. As a result, this conflict-handling option is often underutilized.

Testing Example

To be able to do testing we must start PostgreSQL with "autovacuum=off". Otherwise with instance mostly idle, autovacuum will immediately process bloated objects and it would be unable to catch statistics. We create a simple testing example with multiple indexes:

```
CREATE TABLE IF NOT EXISTS test_unique_constraints(
  id serial primary key,
  unique_text_key text,
  unique_integer_key integer,
  some_other_bigint_column bigint,
  some_other_text_column text);

CREATE INDEX test_unique_constraints_some_other_bigint_column_idx ON test_unique_constraints (some_other_bigint_column );
CREATE INDEX test_unique_constraints_some_other_text_column_idx ON test_unique_constraints (some_other_text_column );
CREATE INDEX test_unique_constraints_unique_text_key_unique_integer_key__idx ON test_unique_constraints (unique_text_key,
unique_integer_key, some_other_bigint_column );
CREATE UNIQUE test_unique_constraints_unique_integer_key_idx INDEX ON test_unique_constraints (unique_text_key );
CREATE UNIQUE test_unique_constraints_unique_text_key_idx INDEX ON test_unique_constraints (unique_integer_key );
```

And now we populate this table with unique data:

```
DO $$
BEGIN
  FOR i IN 1..1000 LOOP
    INSERT INTO test_unique_constraints
    (unique_text_key, unique_integer_key, some_other_bigint_column, some_other_text_column)
    VALUES (i::text, i, i, i::text);
  END LOOP;
END;
$$;
```

In the second step, we use a simple Python script to connect to the database, attempt to insert conflicting data, and close the session after an error. First, it sends 10,000 INSERT statements that conflict with the “test_unique_constraints_unique_int_key_idx” index, then another 10,000 INSERTs conflicting with “test_unique_constraints_unique_text_key_idx”. The entire test is done in a few dozen seconds, after which we inspect all objects using the “pgstattuple” extension. The following query lists all objects in a single output:

```
WITH maintable AS (SELECT oid, relname FROM pg_class WHERE relname = 'test_unique_constraints')
SELECT m.oid as relid, m.relname as relation, s.*
FROM maintable m
JOIN LATERAL (SELECT * FROM pgstattuple(m.oid)) s ON true
UNION ALL
SELECT i.indexrelid as relid, indexrelid::regclass::text as relation, s.*
FROM pg_index i
JOIN LATERAL (SELECT * FROM pgstattuple(i.indexrelid)) s ON true
WHERE i.indrelid::regclass::text = 'test_unique_constraints'
ORDER BY relid;
```

Observed Results

After running the whole test several times, we observe the following:

- The main table “test_unique_constraints” always has 1,000 live tuples, and 20,000 additional dead records, resulting in approx 85% of dead tuples in the table

- Index on primary key always shows 21,000 tuples, unaware that 20,000 of these records are marked as deleted in the main table.
- Other non unique indexes show different results in different runs, ranging between 3,000 and 21,000 records. Numbers depend on the distribution of values generated for underlying columns by the script. We tested both repeated and completely unique values. Repeated values resulted in less records in indexes, completely unique values led to full count of 21,000 records in these indexes.
- Unique indexes showed repeatedly tuple counts only between 1,000 and 1,400 in all tests. Unique index on the "unique_text_key" always shows some dead tuples in the output. Precise explanation of these numbers would require deeper inspection of these relations and code of the pgstattuple function, which is beyond scope of this article. But some small bloat is reported also here.
- Numbers reported by pgstattuple function raised questions about their accuracy, although documentation seems to lead to the conclusion that numbers should be precise on tuple level.
- Subsequent manual vacuum confirms 20,000 dead records in the main table and 54 pages removed from primary key index, and up to several dozens of pages removed from other indexes – different numbers in each run in dependency on total count of tuples in these relations as described above.
- Each failed insert also increments the Transaction ID and thus increases the database's transaction age.

Here is one example output from the query shown above after the test run which used unique values for all columns. As we can see, bloat of non unique indexes due to failed inserts can be big.

```
 relid |                        relation                                   | table_len | tuple_count | tuple_len |
tuple_percent | dead_tuple_count | dead_tuple_len | dead_tuple_percent | free_space | free_percent
-------+-------------------------------------------------------------------+-----------+-------------+-----------+------------
--+------------------+----------------+--------------------+------------+-------------
 16418 | test_unique_constraints                                           |   1269760 |        1000 |     51893 |
4.09 |            20000 |        1080000 |              85.06 |       5420 |         0.43
 16424 | test_unique_constraints_pkey                                      |    491520 |       21000 |    336000 |
68.36 |                0 |              0 |                  0 |      51444 |        10.47
 16426 | test_unique_constraints_some_other_bigint_column_idx             |    581632 |       16396 |    326536 |
56.14 |                0 |              0 |                  0 |     168732 |        29.01
 16427 | test_unique_constraints_some_other_text_column_idx               |    516096 |       16815 |    327176 |
63.39 |                0 |              0 |                  0 |     101392 |        19.65
 16428 | test_unique_constraints_unique_text_key_unique_integer_key__idx |   1015808 |       21000 |    584088 |
57.5 |                0 |              0 |                  0 |     323548 |        31.85
 16429 | test_unique_constraints_unique_text_key_idx                      |     57344 |        1263 |     20208 |
35.24 |                2 |             32 |               0.06 |      15360 |        26.79
 16430 | test_unique_constraints_unique_integer_key_idx                   |     40960 |        1000 |     16000 |
39.06 |                0 |              0 |                  0 |       4404 |        10.75
(7 rows)
```

In a second test, we modify the script to include the ON CONFLICT DO NOTHING clause in the INSERT command and repeat both tests. This time, inserts do not result in errors; instead, they simply return "INSERT 0 0", indicating that no records were inserted. Inspection of the Transaction ID after this test shows only a minimal increase, caused by background processes. Attempts to insert conflicting data did not result in increase of Transaction ID (XID), as PostgreSQL started first only virtual transaction to check for conflicts, and because a conflict was found, it aborted the transaction without having assigned a new XID. The "pgstattuple" output confirms that all objects contain only live data, with no dead tuples this time.

Summary

As demonstrated, each failed insert bloats the underlying table and some indexes, and increases the Transaction ID because each failed insert occurs in a separate transaction. Consequently, autovacuum is forced to run more frequently, consuming valuable system resources. Therefore applications still relying solely on plain INSERT commands without ON CONFLICT conditions should consider reviewing this implementation. But as always, the final decision should be based on the specific conditions of each application.

ABOUT THE AUTHOR

Josef Machytka

View posts