Numbering Schemes and How to Save DB Roundtrips Using PostgreSQL

Holger Jakobs – holger@jakobs.com – http://plausibolo.de

2025-09-17

Inhaltsverzeichnis

1 Identifying Numbers **1**
1.1 Sequences 1
1.2 Universally Unique Identifiers 3
1.3 Random Numbers 4
1.4 Table With Potential Customer Numbers 4
1.5 Customer Table 5
1.6 Extended INSERT Statement 5
1.7 Implementation For Non-PostgreSQL Databases 6

2 Generating an Initial Password **7**
2.1 Customer Table 7
2.2 Function to Insert a New Customer 7
2.3 Using the Function To Insert a New Customer 8

Sometimes new elements have to get identifying numbers to be used as primary keys, like articles, customers or employees. There are several ways of finding numbers to assign – each with its own advantages and disadvantages. In case of a new customer registrating, the new number has to be told, and maybe some random number like a PIN as well. This little article sheds some light on the possibilities PostgreSQL has to offer.

1 Identifying Numbers

1.1 Sequences

The easiest way to number articles, orders, customers, employees, invoices and the like is using a sequence. In PostgreSQL, this is no longer being done by manually creating a sequence and dealing explicitly with it or attaching the “next values” from the sequence as a default value to the primary key column of the table, but by using the standard feature of `GENERATED ALWAYS AS IDENTITY`, which PostgreSQL has implemented.

This guarantees unique numbers and allows for high concurrency, because there is no waiting whether the number picked from the sequence will actually produce a new row in the table. In case the `INSERT` statement fails after a number has been acquired from the sequence (due to a `CHECK` constraint or the like), this number will never appear in the table. The numbers of new items will thus be increasing, but not always without gaps.

The sequence pattern is robust and can be used with `smallint`, `integer`, or `bigint` columns. If in doubt, use `bigint`, because it is very unlikely to overflow and run out of numbers. The size of the identifier is 16, 32 or 64 bits.

An example can look like this:

```
CREATE TABLE article (
  short_name    varchar(50)    NOT NULL UNIQUE,
  description   text,
  retail_price  numeric(12,2)  NOT NULL CHECK (retail_price > 0),
  art_no        integer        NOT NULL GENERATED ALWAYS AS IDENTITY (start with 10000)
);
```

`psql` describes this table as follows:

```
                                  Table "public.article"
    Column     |         Type          | Collation | Nullable |           Default
--------------+-----------------------+-----------+----------+------------------------------
 short_name    | character varying(50) |           | not null |
 description   | text                  |           |          |
 retail_price  | numeric(12,2)         |           | not null |
 art_no        | integer               |           | not null | generated always as identity
Indexes:
    "article_short_name_key" UNIQUE CONSTRAINT, btree (short_name)
Check constraints:
    "article_retail_price_check" CHECK (retail_price > 0::numeric)
```

Simple numbers are fine, but they expose some information about time. A higher number was generated later than a smaller one. Maybe of even more concern may be that valid numbers can easily be guessed. If a customer numer 2322 is valid, there is a high probability that 2321 and 2323 are valid as well. This can lead – and already has lead – to data breaches, because half of the information `(number, password)` was easily guessable.

When a new record is inserted, data for all columns except `art_no` are provided. The last columns will be filled from the sequence. If you need the new article number for further processing, make use of the `RETURNING` clause of the `INSERT` statement.

```
INSERT INTO article VALUES ('pencil', 'Faber Castell pencil HB', 2.20) RETURNING art_no;
```

In case of Java's JDBC, make sure you call this statement with `executeQuery()` and not with `executeUpdate()`, so that you get a `ResultSet` with the `art_no`. In other languages, similar procedures, functions or methods exist.

1.2 Universally Unique Identifiers

Another way of generating unique identifiers is using the standardised system of UUIDs[1], of which there are several variants. PostgreSQL 18 introduces UUIDv7, which are especially suitable for the task we are trying to solve here. Variant UUIDv4, which exists in earlier versions, can work as well. If you prefer other variants, have a look at the extension called `uuid-ossp`.

In any case, the appropriate data type for UUIDs is – not surprisingly – called `uuid`. It is to be preferred over saving a UUID in some text column because it checks the value for validity and output is always nicely formatted with hyphens, while input formats may vary to a certain extent.

The bad thing about UUIDs is their size: 128 bits, which is twice the size of the biggest integer type. But there is no need for a sequence, and UUIDs can be generated independently anywhere offline without any UUID ever appearing twice.

The most popular variants are UUIDv4, which is totally random, and UUIDv7, which contains a millisecond-grained timestamp, so that the values can be time-sorted. Plus, they don't expose any data about the system where they were generated, which is the case of UUIDv1. UUIDs can compared like numbers, so that they can be used with BTree-Indexes.

If you know one customer number, you cannot derive any other customer number from this. No customer number can be guessed. But if a customer always has to provide a 128 bit number, this can be quite a hassle, unless the whole process is completely automatic, which it hardly is if you are dealing with consumers.

An example can look like this:

```
CREATE TABLE article (
  short_name    varchar(50)    NOT NULL UNIQUE,
  description   text,
  retail_price  numeric(12,2)  NOT NULL CHECK (retail_price > 0),
  art_no        uuid           NOT NULL DEFAULT gen_random_uuid ()
);
```

`psql` describes this table as follows:

```
                             Table "public.article"
    Column    |         Type          | Collation | Nullable |      Default
--------------+-----------------------+-----------+----------+-------------------
 short_name   | character varying(50) |           | not null |
 description  | text                  |           |          |
 retail_price | numeric(12,2)         |           | not null |
 art_no       | uuid                  |           | not null | gen_random_uuid()
Indexes:
    "article_short_name_key" UNIQUE CONSTRAINT, btree (short_name)
```

1 https://en.wikipedia.org/wiki/Uuid

```
Check constraints:
    "article_retail_price_check" CHECK (retail_price > 0::numeric)
```

The function used for generating the UUID may vary depending on the variant you want to use. Inserting a new row works exactly like before, only the data type of the `art_no` column is different.

1.3 Random Numbers

In case of article numbers, there is no concern about privacy, so using numbers in sequence is no problem. In case of invoices, numbers in sequence (with no gaps) may even be a necessity due to legal regulations. But in case of customers, you might want numbers a lot shorter than UUIDs, but not in sequence and therefore not guessable.

So random numbers between 100,000 and 999,999 may be a good idea. But how do you get random numbers in this range, which are guaranteed to be unique, which is necessary for using them as primary key?

You could just try to be lucky and take random numbers and hope that no collision will occur. And if it does, just get another random number until it succeeds. This doesn't look like a nice way of doing it, but it will work. The disadvantage is that you might have to do several tries in a loop within your application, which is complicated and error prone.

Another suggestion is to have a table with all possible numbers, which you can populate using the function `generate_series()`, but shuffle the numbers, so that when you pick one, you get a random one. The delete it if has actually been used, but keep if if the `INSERT` fails.

But how to achieve this? First, let's see how the table with potential customer numbers is created and populated. The rows have to be inserted in random order, so that a common table expression is being used here.

1.4 Table With Potential Customer Numbers

```
-- Table with potential customer numbers
CREATE TABLE cust_no (
  num integer
);
-- Populate table with 899,999 numbers in random order
WITH nums AS (SELECT generate_series(100000, 999999))
INSERT INTO cust_no SELECT * FROM nums ORDER BY random();
-- Add a primary key index (don't do it in the CREATE statement)
ALTER TABLE cust_no ADD PRIMARY KEY (num);
```

Then, the table for customers is created. Yes, we have left out some important columns to keep it small, but included a simple check for the email address and that the customer is at least 18 years old, so that there is a chance of a failing `INSERT`. And it would be nice to keep record of the timestamp when the row was inserted. We chose to leave the phone number optional, but if it is supplied, it must be an international number with plus sign and country code.

1.5 Customer Table

```
-- Table for customers
CREATE TABLE customer (
  family_name  varchar(50) NOT NULL CHECK (trim(both from family_name) <> ''),
  first_name   varchar(50) NOT NULL CHECK (trim(both from first_name) <> ''),
  email        varchar(50) NOT NULL CHECK (email ~ '.@.+\....*') UNIQUE,
  birthdate    date        NOT NULL CHECK (birthdate + interval '18 years' <= current_date),
  phoneno      varchar(20)          CHECK (phoneno ~ '^\+[0-9 -]{9,}$'),
  cust_no      integer     PRIMARY KEY,
  entered      timestamptz NOT NULL DEFAULT current_timestamp
);
```

1.6 Extended INSERT Statement

Now for inserting a new customer. The command has to get a number from the table `cust_no` to be used for the new customer, but delete it from this table, so that no other customer can get the same number. The clause `FOR UPDATE` prevents the deletion by others) and the clause `SKIP LOCKED` makes sure that concurrent sessions can also acquire and lock a number by skipping those already locked by the `FOR UPDATE` clause. Without using `SKIP LOCKED`, concurrent `SELECT` statements with a `FOR UPDATE` clause would block until the end of the transaction holding the lock. This may seem negligible because the `INSERT` takes so little time, but the whole thing could be part of some long running transaction, which then would be harmful.

The new customer number has to be returned to the application for the new customer to receive his number, which is achieved by using the `RETURNING` clause appended to the `INSERT` statement.

So, three commands have to be combined into one: `SELECT`, `DELETE` and `INSERT`. Since PostgreSQL allows using all of these in Common Table Expressions and the `INSERT` to return generated values, the following construction is feasible.

```
WITH cno AS (
  SELECT num FROM cust_no LIMIT 1 FOR UPDATE SKIP LOCKED
), del AS (
  DELETE FROM cust_no WHERE num = (select num from cno)
)
 INSERT INTO customer VALUES (
    'Meier', 'Franz', 'franz@meier.com',
    '1984-03-12', NULL,
    (SELECT num FROM cno))
  RETURNING cust_no;
```

Getting exactly one row from the table `cust_no` produces a random number. This row gets deleted in the second common table expression and is used in the main query, which is an `INSERT` with a `RETURNING` clause. A simple test has shown that the `INSERT` takes about 1 ms to execute, because getting a single row without specifying which one is always fast, as is the deletion of a row with the primary key in the `WHERE` clause. The `INSERT` statement itself is the same as if there hadn't used this method for generating the customer number.

This is the query plan of the operation:

```
                                                              QUERY PLAN
------------------------------------------------------------------------------------------------------------------------------------------
 Insert on customer  (cost=8.51..8.52 rows=1 width=428) (actual time=0.823..0.827 rows=1 loops=1)
   CTE cno
     ->  Limit  (cost=0.00..0.02 rows=1 width=10) (actual time=0.048..0.050 rows=1 loops=1)
           ->  LockRows  (cost=0.00..21982.98 rows=899999 width=10) (actual time=0.046..0.047 rows=1 loops=1)
                 ->  Seq Scan on cust_no  (cost=0.00..12982.99 rows=899999 width=10) (actual time=0.021..0.022 rows=1 loops=1)
   CTE del
     ->  Delete on cust_no cust_no_1  (cost=0.45..8.46 rows=0 width=0) (actual time=0.053..0.053 rows=0 loops=1)
           InitPlan 2
             ->  CTE Scan on cno  (cost=0.00..0.02 rows=1 width=4) (actual time=0.002..0.003 rows=1 loops=1)
           ->  Index Scan using cust_no_pkey on cust_no cust_no_1  (cost=0.42..8.44 rows=1 width=6) (actual time=0.042..0.043 rows=1 loops=1)
                 Index Cond: (num = (InitPlan 2).col1)
   InitPlan 4
     ->  CTE Scan on cno cno_1  (cost=0.00..0.02 rows=1 width=4) (actual time=0.056..0.058 rows=1 loops=1)
   ->  Result  (cost=0.00..0.01 rows=1 width=428) (actual time=0.066..0.066 rows=1 loops=1)
 Planning Time: 0.296 ms
 Execution Time: 0.970 ms
```

Of course the numbers in the table `cust_no` could as well be alphanumeric instead of purely consisting of numbers. Sometimes, a mixture of letters and digits is easier for humans to remember. Also spaces and/or hyphens are possible.

The point of using this long statement for inserting a new row saves several round trips to the database, which otherwise would cause some lag due to the latency of the round trip and also cause more load on the server. The implicit transaction introduced by the statement in autocommit mode is easier to handle than an explicit one.

1.7 Implementation For Non-PostgreSQL Databases

The alternative implementation without using the special capabilities of PostgreSQL would be like the following.

```
START TRANSACTION;

-- Retrieve the result of this statement into the variable num of the application.
SELECT num FROM cust_no LIMIT 1 FOR UPDATE SKIP LOCKED;

-- Using the num value for $1, syntax may vary depending on your programming language
DELETE FROM cust_no WHERE num = $1 using num;

-- Using the num value for $1
INSERT INTO customer VALUES (
    'Meier', 'Franz', 'franz@meier.com',
    '1984-03-12', NULL,
    $1) using num;
```

```
COMMIT TRANSACTION;
```

This would mean 5 round trips to the database server.

2 Generating an Initial Password

Providing an initial password or PIN number for the new customer, while saving a hash of it into the table would also be a nice feature. Additionally, some alterations of the values provided can be helpful. Names and E-Mail addresses have no leading or trailing blanks, so in case someone has keyed in those be incident, they should be removed before saving the data.

All of this can be achied by using a function (or a procedure, for that matter) to insert the new customer and returning the customer number and the initial password, which later has to be changed on the first login, which is not part of this document.

2.1 Customer Table

First, we create a new table for the customers with an additional column for saving the hash of the PIN.

```
-- Table for customers with additional column pass_hash
CREATE TABLE customer (
  family_name  varchar(50) NOT NULL CHECK (trim(both from family_name) <> ''),
  first_name   varchar(50) NOT NULL CHECK (trim(both from first_name) <> ''),
  email        varchar(50) NOT NULL CHECK (email ~ '.@.+\....*') UNIQUE,
  birthdate    date        NOT NULL CHECK (birthdate + interval '18 years' <= current_date),
  phoneno      varchar(20)          CHECK (phoneno ~ '^\+[0-9 -]{9,}$'),
  pass_hash    char(66)    NOT NULL,
  cust_no      integer     PRIMARY KEY,
  entered      timestamptz NOT NULL DEFAULT current_timestamp
);
```

2.2 Function to Insert a New Customer

Now for the function taking 5 arguments and returning two values: the customer number and the PIN in clear text. The `sha256` hash of the latter is saved with the customer data. For simplicity's sake, the initial password is the letter „P“ followed by a 6-digit number. The leading „P“ makes sure the customer doesn't mix up the customer number and the PIN.

```
CREATE OR REPLACE FUNCTION new_customer (
  family_name varchar, first_name varchar, email varchar,
  birthdate date, phoneno varchar,
  out _cust_no integer, out pin_number char(7)) AS
$$
BEGIN
  pin_number = 'P' || trim(to_char(random()*900_000 + 100_000, '999999'));
  WITH cno AS (
    SELECT num FROM cust_no LIMIT 1 FOR UPDATE SKIP LOCKED
  ), del AS (
    DELETE FROM cust_no WHERE num = (select num from cno)
  )
  INSERT INTO customer VALUES (
    trim(family_name), trim(first_name), trim(email),
    birthdate, trim(phoneno),
    sha256(pin_number::text::bytea),
    (SELECT num FROM cno))
  RETURNING cust_no INTO _cust_no;
  RETURN;
END;
$$ LANGUAGE plpgsql;
```

The method shown here can also be used to create consecutive numbers for invoices, where a sequence doesn't work well, because it doesn't avoid gaps. Just leave out the `ORDER BY random()` when populating the table with the numbers, so that they will be retrieved in natural order.

2.3 Using the Function To Insert a New Customer

```
postgres# select * from new_customer('Meier','Franzi',
          'franzi@meier.com', '1999-04-11', null);
 _cust_no | pin_number
----------+------------
   872871 | P498402
```

The record coming back has two columns bearing the names of the `OUT` arguments of the function. The leading underbar of `_cust_no` is necessary to avoid ambiguity between the argument and the column name.