In our previous article we saw three classic Database Modelization Anti-Patterns. The article also contains a reference to a Primary Key section of my book The Art of PostgresQL, so it's only fair that I would now publish said Primary Key section!

So in this article, we dive into Primary Keys as being a cornerstone of database normalization. It's so important to get Primary Keys right that you would think everybody knows how to do it, and yet, most of the primary key constraints I've seen used in database design are actually not primary keys at all.

Table of Contents

Before we can get into the details of Primary Keys themselves, let's do a quick review of Normal Forms and why they are interesting to us when we design a database model.

Normal Forms

There are several levels of normalization and the web site dbnormalization.comoffers a practical guide to them. In this quick introduction to database normalization, we include the definition of the normal forms:

- 1st Normal Form (*1NF*)

  A table (relation) is in *1NF* if:
  1. There are no duplicated rows in the table.
  2. Each cell is single-valued (no repeating groups or arrays).
  3. Entries in a column (field) are of the same kind.

- 2nd Normal Form (*2NF*)

  A table is in *2NF* if it is in *1NF* and if all non-key attributes are dependent on all of the key. Since a partial dependency occurs when a non-key attribute is dependent on only a part of the composite key, the definition of *2NF* is sometimes phrased as: "A table is in *2NF* if it is in *1NF* and if it has no partial dependencies."

- 3rd Normal Form (*3NF*)

  A table is in *3NF* if it is in *2NF* and if it has no transitive dependencies.

- Boyce-Codd Normal Form (*BCNF*)

  A table is in *BCNF* if it is in *3NF* and if every determinant is a candidate key.

- 4th Normal Form (*4NF*)

  A table is in *4NF* if it is in *BCNF* and if it has no multi-valued dependencies.

- 5th Normal Form (*5NF*)

  A table is in *5NF*, also called "Projection-join Normal Form" (*PJNF*), if it is in *4NF*and if every join dependency in the table is a consequence of the candidate keys of the table.

- Domain-Key Normal Form (*DKNF*)

  A table is in *DKNF* if every constraint on the table is a logical consequence of the definition of keys and domains.

What all of this say is that if you want to be able to process data in your database, using the relational model and SQL as your main tooling, then it's best not to make a total mess of the information and keep it logically structured.

In practice database models often reach for *BCNF* or *4NF*; going all the way to the *DKNF* design is only seen in specific cases.

Primary Keys

Primary keys are a database constraint allowing us to implement the first and second normal forms. The first rule to follow to reach first normal form says *"There are no duplicated rows in the table"*.

A primary key ensures two things:

- The attributes that are part of the *primary key* constraint definition are not allowed to be *null*.

- The attributes that are part of the *primary key* are unique in the table's content.

To ensure that there is no duplicated row, we need the two guarantees. Comparing *null* values in SQL is a complex matter — read Jeff Davis' What is the deal with NULLs? to convince yourself, and rather than argue if the no-duplicate rule applies to *null = null* (which is *null*) or to *null is not null* (which is false), a *primary key* constraint disallow *null* values entirely.

Surrogate Keys

The reason why we have *primary key* is to avoid duplicate entries in the data set. As soon as a *primary key* is defined on an automatically generated column, which is arguably not really part of the data set, then we open the gates for violation of the first normal form.

As an example of that, we're going to model the publication of articles in a newspaper, where each article belongs to a single category. Here's a first version of our main article table definition:

```
create table sandbox.article
 (
   id          bigserial primary key,
   category    integer references sandbox.category(id),
   pubdate     timestamptz,
   title       text not null,
   content     text
 );
```

This model isn't even compliant with *1NF*:

```
insert into sandbox.article (category, pubdate, title)
     values (2, now(), 'Hot from the Press'),
            (2, now(), 'Hot from the Press')
  returning *;
```

PostgreSQL is happy to insert duplicate entries here:

```
─[ RECORD 1 ]───────────────────────────
id       │ 1001
category │ 2
pubdate  │ 2017-08-30 18:09:46.997924+02
title    │ Hot from the Press
content  │ ¤
═[ RECORD 2 ]═══════════════════════════
id       │ 1002
category │ 2
pubdate  │ 2017-08-30 18:09:46.997924+02
title    │ Hot from the Press
content  │ ¤

INSERT 0 2
```

Of course, it's possible to argue that those entries are not duplicates: they each have their own *id* value, which is different — and it is an artificial value derived automatically for us by the system.

Actually, we now have to deal with two article entries in our publication system with the same category (category 2 is *news*), the same title, and the same publication date. I don't suppose this is an acceptable situation for the business rules.

In term of database modeling, the artificially generated key is named a *surrogate key*because it is a substitute for a *natural key*. A *natural key* would allow preventing duplicate entries in our data set.

We can fix our schema to prevent duplicate entries:

```
 create table sandbox.article
  (
    category   integer references sandbox.category(id),
    pubdate    timestamptz,
    title      text not null,
    content    text,

    primary key(category, title);
  );
```

Now, you can share the same article's title in different categories, but you can only publish with a title once in the whole history of our publication system. Given this alternative design, we allow publications with the same title at different publication dates. It might be needed, after all, as we know that history often repeats itself.

```
 create table sandboxpk.article
  (
    category   integer references sandbox.category(id),
    pubdate    timestamptz,
    title      text not null,
    content    text,

    primary key(category, pubdate, title)
  );
```

Say we go with the solution that allows reusing the same title at a later date. We now have to change the model of our *comment* table, which references the *sandbox.article*table:

```
 create table sandboxpk.comment
  (
    a_category integer     not null,
    a_pubdate  timestamptz not null,

    a title    text        not null.
    pubdate     timestamptz,
    content     text,

    primary key(a_category, a_pubdate, a_title, pubdate, cor

    foreign key(a_category, a_pubdate, a_title)
     references sandboxpk.article(category, pubdate, title)
  );
```

As you can see each entry in the *comment* table must have enough information to be able to reference a single entry in the *article* table, with a guarantee that there are no duplicates.

We then have quite a big table for the data we want to manage in there. So there’s yet another solution to this *surrogate* key approach, a trade-off where you have the generated summary key benefits and still the natural primary key guarantees needed for the *1NF*:

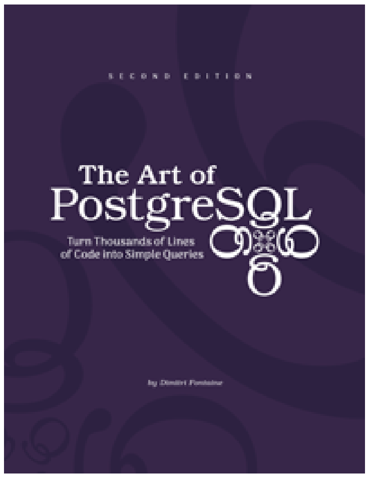
```
create table sandboxpk.article
 (
    id          bigserial primary key,
    category    integer      not null references sandbox.cate
    pubdate     timestamptz  not null,
    title       text         not null,
    content     text,

    unique(category, pubdate, title)
  );
```

Now the *category*, *pubdate* and *title* have a *not null* constraint and a *unique* constraint, which is the same level of guarantee as when declaring them a *primary key*. So we both have a *surrogate* key that's easy to reference from other tables in our model, and also a strong *1NF* guarantee about our data set.

Conclusion

This article is an extract from my book The Art of PostgresQL where we dive into more details and examples around database modeling in Chapter 6. Before that you can read whole chapters about how to be proficient at advanced SQL so that it's easier to make the right choices when designing a database model. After all, an important trade-off in the database design is that we're able to write the queries we need to implement and support users workflows.



After that, Chapter 7 is titled "Data Manipulation and Concurrency Control" and deals with concurrent accesses to your data, which is the most important aspect of a relational database management system!