Postgres OnLine Journal: January 2024 - December 2024

An in-depth Exploration of the PostgreSQL Open Source Database

Table Of Contents

A Product of Paragon Corporation

http://www.paragoncorporation.com/

http://www.postgresonline.com/

PostgreSQL Mistakes and How To Avoid them

The adoption of PostgreSQL is growing each year. Many people coming to PostgreSQL are often coming from other relational databases, with assumptions of how relational databases work. Although PostgreSQL may feel very familiar to these people, it is different enough to cause some misunderstandings which lead to bad and slow queries. There are also people coming often from only programming backgrounds, who assume data processing in SQL is much like data processing in any language. For these two groups of folks, I think the new book written by EDB and 2nd Quadrant Author, Jimmy Angelakos, "PostgreSQL Mistakes and How To Avoid Them" will save them a lot of miss-steps.

The book, **PostgreSQL Mistakes and How To Avoid Them** is currently in Manning Publication Early Action form, meaning it is not complete yet, but so far, I like what I see. It covers quite a few common mistakes many have run into (including myself) when first starting off with PostgreSQL and more importantly how to address them. What I like most about the book, is the real-world examples it provides, and also categorizes them into sections, you can easily find when you are looking for something. One particular one, which I still make often is the mysteriousness of what NULL does when one of your items in an IN is NULL. This one still catches me off-guard often. Another important one which even long standing users of PostgreSQL may be caught off-guard by is the use of CTEs. Pre-PG 12, everyone was told CTEs aren't simply syntactic sugar, they are materialized and so how it behaves when compared to other databases supporting CTEs was different. From PostgreSQL 12 on, this changed, and so you can use CTEs more freely to write more readable code and also to improve query performance in some cases. The book demonstrates an example using both CTE and without CTE and shows the difference in query plans. It also covers the use of the MATERIALIZED keyword in CTEs which is something you won't find in most other relational databases supporting CTES.

One other gotcha I see many newbies making is quoted identifiers. This is a common issue with SQL Server converts, where SQL Server will preserve casing of your identitiers in a table, but not care if you reference them in that casing with brackets or quotes. So you'll often find columns from these converted databases like "Customer", which have to be referenced as **"Customer"** rather than **customer** or **Customer**. This comes as a great shock when you can't do the same in PostgreSQL. This issue is explained in one of the examples.

Back to Table Of Contents

Basics

Some of my favorite PostgreSQLisms *Beginner*

When I work with other relational databases I am reminded about how I can't use my favorite PostgreSQL hacks in them. I call these hacks PostgreSQLisms. A PostgreSQLism is a pattern of SQL unique to PostgreSQL or descendants of PostgreSQL. In this post I'm going to cover some of my favorite ones.

For these exercises I'm going to use these basic tables, so you can follow along. Be warned I'm using arrays here just to say I'm using arrays.

Arrays are so fundamental to my suite of tools that I can't even acknowledge I am using them except when I get stuck working in another relational database that doesn't support them which is pretty much every other non-PostgreSQL database I have the misfortune of working with. But I'm going to save my array love fest for another article, and just lightly use them here to create my test tables.

```
CREATE TABLE members(id bigint GENERATED ALWAYS AS IDENTITY,
    name_first varchar(50), name_last varchar(50) );

INSERT INTO members(name_first, name_last)
   VALUES ('Jill', 'Hill'), ('Jack', 'Hill');

CREATE TABLE orders(id bigint GENERATED ALWAYS AS IDENTITY, id_member bigint, date_add timestamptz DEFAULT CURRENT_TIMESTAMP);
INSERT INTO orders(id_member)
        SELECT i % 2
        FROM generate_series(1,10) AS i;

CREATE TABLE orders_items(id bigint GENERATED ALWAYS AS IDENTITY, id_order bigint, item_code varchar(100), quantity integer);
 INSERT INTO orders_items(id_order, item_code, quantity)
        SELECT i, (ARRAY['orange', 'apple', 'banana', 'chocolate'])[ j ], (random()*6)::integer
        FROM generate_series(1,5) AS i CROSS JOIN generate_series(1, (random()*i)::integer + 1 ) AS j;
```

DISTINCT ON

I have never seen another database offer the **DISTINCT ON** clause. Sure every relational database has a DISTINCT clause, but when was the last time you saw a database support the **DISTINCT ON** clause. For those who are unfamiliar with this gem of a clause. What differentiates a **DISTINCT ON** from a **DISTINCT** is you have a choice of what columns should be considered distinct and your ORDER BY controls your preferred row to return. The only caveat is if you have an ORDER BY then the beginning list of columns should include your DISTINCT ON columns. Here is such an example:

```
SELECT DISTINCT ON(m.name_last, m.name_first, m.id) m.name_last, m.name_first, o.*, oi.item_code
FROM members AS m INNER JOIN orders AS o ON m.id = o.id_member INNER JOIN orders_items AS oi ON o.id = oi.id_order
        ORDER BY m.name_last, m.name_first, m.id, o.date_add DESC;
```

What does this statement do, it returns each members first and last name, the last order they made, and an item_code from an item on that order.

Return a row as a column

This next PostgreSQLism I consider the foundational piece that makes PostgreSQL object-relational. It's the trick that you can stuff a whole row into a column. This feature has so many uses you can't imagine. In its classic form, it looks like this:

```
SELECT m
FROM members AS m;
```

Which outputs:

```
m
---------------
 (1,Jill,Hill)
 (2,Jack,Hill)
(2 rows)
```

You might be thinking, so what? How is this useful? Its true value only shows itself when you dare pass it to functions that understand composite types which is more functions than you realize. Such as to_jsonb, array_agg, and even window functions etc.

For example

```
SELECT to_jsonb(m) FROM members AS m;
```

OR more usefully combining with another PostgreSQLism

Ordered Aggregates

```
SELECT jsonb_agg(m ORDER BY m.name_first, m.name_last) AS output
        FROM members AS m;
```

Output:

```
 output
-----------------------------------------------------------------------------------------------------------
 [{"id": 2, "name_last": "Hill", "name_first": "Jack"}
       , {"id": 1, "name_last": "Hill", "name_first": "Jill"}]
```

Yes most PostgreSQL aggregates can have an ORDER BY in the clause. This feature is not limited to built-in aggregates, but any aggregate you create or you see in an extension. Of course it's useless unless the answer changes depending on order in which you feed items.

Window Aggs with rows

Sure other relational databases have mastered window functions, just as PostgreSQL has mastered window functions, but can you pass a whole row into a window function in any other relational database you have come across. Try this trick in another relational database and be prepared to be very disappointed. Sometimes I get a little lazy, like for example, I'd like to see Jill's previous order in her current order, but I don't have time to go around and itemize all the fields in that last order, and hell am I going to write a bunch of lead function calls to get all the pieces I need. So I do this instead.

```
WITH a AS (SELECT m.name_last, m.name_first, o,
    lead(o) OVER(PARTITION BY m.id ORDER BY o.date_add) AS lead_o
FROM members AS m INNER JOIN orders AS o ON m.id = o.id_member
WHERE m.name_first = 'Jill'
)
SELECT a.name_last, a.name_first, (o).id AS orig_order_id, (lead_o).*
FROM a;
```

Admittedly that (subtable).field syntax takes a little bit of getting used to but it gets the job done.

Subtracting attributes from a jsonb

I have longed for an SQL syntax

```
SELECT m.* EXCEPT COLUMNS(name_last)
 FROM members AS m
```

But my wish has not been granted. However there is the next best thing, which is generally good enough, cause most of my time is spent slugging data at web apis and applications who only want to be fed JSON for some reason.

So luckily I can do this

```
SELECT jsonb_agg(to_jsonb(m) - 'name_last' - 'id' ORDER BY name_first)
FROM members AS m;
```

Or slightly shorter if I've got a bunch of fields to remove

```
SELECT jsonb_agg(to_jsonb(m) - ARRAY['name_last','id'] ORDER BY name_first)
FROM members AS m;
```

Back to Table Of Contents

Basics

Substring function Regex style *Beginner*

I was reviewing some old code when I stumbled across something I must have completely forgotten or someone else some time ago knew. That is that it is possible to use the function **substring** for regular expression work.

Most of the **regexp** functions in PostgreSQL usually start with regexp or are operators like **~**. But I completely forgot about the **substring** function that it understands regular expressions too. When I have to parse out some little piece of text using regular expressions, I usually reach for one of those **regexp_*** functions like **regexp_replace**, **regexp_match**, or **regexp_matches** and use **substring** only when I want to extract a range of characters by index numbers from a string. But there was code right there telling me perhaps I or someone else was much smarter back then. All these functions are summarized on Documentation: PostgreSQL string functions.

The common substring function most people use is:

```
substring(string, start_position, length)
```

which can also be written as

```
substring(string FROM start_position FOR length)
```

With the `length` and `FOR length` clause being optional and when left out giving you the string from the start position to the end of string.

How you use it is as follows:

```
SELECT substring('Jill lugs 5 pails', 11, 7);
```

returns: `5 pails`

These pedestrian forms of **substring** are even present in databases that completely suck at doing regular expressions. I shall not name these databases. Just know they exist. PostgreSQL has another form which I find sometimes more understandable and shorter to write than doing the same with the aforementioned **regexp_*** functions.

Lets repeat the above example but with the understanding that we need to pick out what exactly Jill is lugging and how many of them she is carrying.

```
SELECT substring('Jill lugs 5 pails', '[0-9]+\s[A-Za-z]+');
```

Which returns the same thing, but without having to know postion, but only be concerned with phraseology.

Now sure in many cases you would be better off with the more powerful **regexp_match** or **regexp_matches** siblings , like when you are trying to separate parts of a statement in one go. Like just maybe I want my count of pails to be separate from the thing `pails` then sure I'd do.

```
SELECT r[1] AS who, r[2] AS action, r[3] AS how_many, r[4] AS what FROM regexp_match('Jill lugs 5 pails', '([A-Za-z]+)\s+
([A-Za-z]+)\s+([0-9]+)\s([A-Za-z]+)') AS r;
```

```
 who  | action | how_many | what
------+--------+----------+-------
 Jill | lugs   | 5        | pails
(1 row)
```

And when I need to be a bit greedy and return multiple records cause I'm to grab parts of each sentence that fit my phraseology in a pool of word soup, I will reach for regexp_matches.

```
SELECT r[1] AS who, r[2] AS action, r[3] AS how_many, r[4] AS what
FROM regexp_matches('Jill lugs 5 pails. Jack lugs 10 pails.', '([A-Za-z]+)\s+([A-Za-z]+)\s+([0-9]+)\s([A-Za-z]+)', 'g') AS r;
```

```
 who  | action | how_many | what
------+--------+----------+-------
 Jill | lugs   | 5        | pails
 Jack | lugs   | 10       | pails
(2 rows)
```

While both of these are great they both always return arrays or sets of arrays.

regexp_matches on the upside is a set returning function, so if you need multiple answers that match your pattern it does the trick, but when nothing matches, your query blows up returning no records. This makes it slightly dangerous to use in a FROM clause unless you are aware of this and don't care or have come up with ways to avoid such as using a LEFT JOIN.

regexp_match while it always returns back cause it's not a set returning, is annoying cause you always get back an array you must pick out your values. So the original question becomes a slight cacophony of extra `()[]` as follows:

```
SELECT (regexp_match('Jill lugs 5 pails', '[0-9]+\s[A-Za-z]+'))[1];
```

As I get older those extra characters annoy me cause it's more typing and more useless characters to look at. Although I haven't tested, I suspect it's slower too.

Back to Table Of Contents

Using PostgreSQL Extensions

PostgreSQL 17 64-bit for Windows FDWs

We are pleased to provide binaries for *file_textarray_fdw* and *odbc_fdw* for PostgreSQL 17 Windows 64-bit.

To use these, copy the files into your PostgreSQL 17 Windows 64-bit install folders in same named folders and then run CREATE EXTENSION as usual in the databases of your choice. More details in the packaged README.txt

These were compiled against PostgreSQL 17.2 using msys2 / mingw64 and tested against PostgreSQL 17.2 EDB windows distribution.

- **Windows 64-bit package** fdw_win64_17_bin.zip fdw_win64_17_bin.7z

This package contains the following FDWs:

- *odbc_fdw (version 0.5.2.3), patched to work with PostgreSQL 17* for connecting to ODBC data sources such as SQL Server, Oracle, MS Access databases, and anything else that has a 64-bit ODBC driver. Note that since this is for PostgreSQL 64-bit, it can only use ODBC 64-bit connections.
- *file_textarray_fdw*, great and fast for working with arbitrary and pesky delimited data. Especially where they weren't considerate enough to give you the same number of columns per row.

Note this package does not include ogr_fdw since ogr_fdw is packaged as part of PostGIS Bundle packages from EnterpriseDb Stackbuilder (for PostGIS >= 3.0) .

If you do not have PostGIS Bundle installed (and don't want to for some reason) and want to use ogr_fdw on windows, you can download from: Winnie's PG 17 Extras. ogr_fdw is a great FDW for querying not just spatial data, but also a ton of other file formats or relational (including odbc, dbase files, spreadsheets) since spatial is a superset of most file formats and databases. Even sweeter, the ogr_fdw extension fully supports the **IMPORT FOREIGN SCHEMA** feature, thus allowing you to do things like link in a whole folder of CSV files, or a workbook with many sheets with following steps:

```
CREATE EXTENSION IF NOT EXISTS ogr_fdw;
CREATE SERVER fds_csvs
    FOREIGN DATA WRAPPER ogr_fdw
    OPTIONS( datasource  'C:/fdw_data/csvs' , format 'CSV');

CREATE SCHEMA IF NOT EXISTS staging;

IMPORT FOREIGN SCHEMA ogr_all FROM SERVER fds_csvs INTO staging;
```

Back to Table Of Contents