Build your Data Dictionary in PostgreSQL

By Ryan Lambert -- Published September 24, 2018

This post provides an introduction to building a data dictionary directly in our PostgreSQL databases. The steps outlined here are specific to PostgreSQL, though every database platform has these basics components in place.

> My 2022 post Postgres Data Dictionary for everyone shows how to use the PgDD extension to make it easy to query data dictionary details using standard SQL sytnax.

What is a Data Dictionary?

Documentation is a good thing. Data dictionaries are one important component of documenting your databases. A data dictionary is a common tool used to provide database-specific documentation to analysts, developers, and other business users. A good data dictionary provides insights into a database's structure, constraints, relationships, and sources of data found inside a system.

According to Wikipedia, a data dictionary is a "centralized repository of information about data such as meaning, relationships to other data, origin, usage, and format".

In my post From Idea to Database: Define, Design, Repeat, I introduced a data dictionary using Google Sheets that looked like the following image.

Table Name	piws.observation
Table Source	Data comes in from PiWS Python program running directly on the PiWS (Raspberry Pi).
Description	This table stores the raw observation data as it comes in to the PiWS database. Data is queried from here to send data to the API for long term storage and analysis.
Docs Updated:	9/3/2018

Field Name	Data Type	Attributes	PK	FK	Ix	Bytes / Row	Default Value	Expected Size (bytes)	Max Size (bytes)	Description
observation_id	SERIAL (INT)	NOT NULL	Y			4		4	4	
sensor_id	INT	NOT NULL				4		4	4	
calendar_id	INT	NOT NULL		Y		4		4	4	
time_id	INT	NOT NULL		Y		4		4	4	
timezone	TEXT	NOT NULL				14		14	50	Indicates the time zone the sensors are in. e.g. "America/Denver"
sensor_values	JSONB	NOT NULL				50		50	300	Raw data received from PiWS Python program.

Keep it updated

The above example in a spreadsheet format is great for initial design purposes, before your database even exists. Long-term, a data dictionary built and maintained manually in a spreadsheet is ~~nearly~~ impossible to keep updated. For example, if an early design designated a column with a data type of `TIMESTAMPTZ` (timestamp with time zone), how would you know if that column had been changed to a plain `TIMESTAMP` during a later development phase? You have to go manually inspect the database and compare it to your static documentation and hope that you can spot any (and all!) differences.

Those types of minor changes are both common and incredibly difficult to capture complete or accurately. That kind of documentation error is prevalent in manually maintained documentation.

The easiest way to solve these problems is to keep the data dictionary directly within the database itself.

Built-in tools

Luckily for us, `psql` provides a number of powerful and helpful built in commands. Some of the ones I run most frequently are:

- `\dn` - List schemas
- `\dt` - List tables
- `\dv` - List views
- `\df` - List functions

> To see all `psql` commands and other help use `psql --help` from the Linux command line. From within `psql` use `\?`.

List tables

Using `\dt` will list the tables in the active database. To see extra information use `\dt+` and it will include the table's size on disk.

```
 \dt+
                        List of relations
┌────────┬──────────┬───────┬──────────────┬──────────┬─────────────┐
│ Schema │   Name   │ Type  │    Owner     │   Size   │ Description │
╞════════╪══════════╪═══════╪══════════════╪══════════╪═════════════╡
│ public │ calendar │ table │ rpl_db_admin │ 3728 kB  │             │
│ public │ time     │ table │ rpl_db_admin │ 7200 kB  │             │
└────────┴──────────┴───────┴──────────────┴──────────┴─────────────┘
(2 rows)
```

In the above example, the helpful "Description"column is empty. To fix that we can add comments to the tables.

```
COMMENT ON TABLE public.calendar IS 'Standard calendar table.  One row per date (datum) with common
human friendly grouping columns.';
```

Running our `\dt+` command again shows our new description.

```
\dt+
                                              List of relations
┌────────┬──────────┬───────┬──────────────┬──────────┬──────────────────────────────────────────────────────────────────────────┐
│ Schema │   Name   │ Type  │    Owner     │   Size   │                                Description                               │
├────────┼──────────┼───────┼──────────────┼──────────┼──────────────────────────────────────────────────────────────────────────┤
│ public │ calendar │ table │ rpl_db_admin │ 3728 kB │ Standard calendar table.  One row per date (datum) with…                 │
│        │          │       │              │         │… common human friendly grouping columns.                                │
│ public │ time     │ table │ rpl_db_admin │ 7200 kB │                                                                          │
└────────┴──────────┴───────┴──────────────┴──────────┴──────────────────────────────────────────────────────────────────────────┘
(2 rows)
```

Describe objects

The `\d` command in `psql` can be used to provide details about a single object, such as a table, view or index. In the case of a table, it will list the columns with their data type and other helpful details.

Just like the above example added a comment to a table, columns can also have comments. Adding comments on columns can help clear up any ambiguity in what data a particular column stores.

```
COMMENT ON COLUMN public.time.quarterhour IS 'e.g. 10:00 - 10:14';
```

Now using the `\d+` command will include that comment as a description on that column.

```
\d+ public.time
                                                Table "public.time"
┌──────────────┬─────────────────────────────┬───────────┬──────────┬─────────┬──────────┬──────────────┬──────────────────────┐
│    Column    │            Type             │ Collation │ Nullable │ Default │ Storage  │ Stats target │ Description          │
├──────────────┼─────────────────────────────┼───────────┼──────────┼─────────┼──────────┼──────────────┼──────────────────────┤
│ time_id      │ integer                     │           │ not null │         │ plain    │              │                      │
│ timeofday    │ time without time zone      │           │          │         │ plain    │              │                      │
│ hour         │ smallint                    │           │          │         │ plain    │              │                      │
│ minute       │ smallint                    │           │          │         │ plain    │              │                      │
│ second       │ smallint                    │           │          │         │ plain    │              │                      │
│ quarterhour  │ text                        │           │          │         │ extended │              │ e.g. 10:00 - 10:14   │
│ daytimename  │ text                        │           │          │         │ extended │              │                      │
│ daynight     │ text                        │           │          │         │ extended │              │                      │
└──────────────┴─────────────────────────────┴───────────┴──────────┴─────────┴──────────┴──────────────┴──────────────────────┘
Indexes:
    "pk_time_id" PRIMARY KEY, btree (time_id)
```

Reverse engineer built-in commands

The built-in `psql` commands are great. What's even better is those commands give great insight on how to query PostgreSQL's internal meta-data via the `pg_catalog` and `information_schema` objects.

To see how the built-in commands (e.g. `dt+`) work, run this in `psql`:

```
\set ECHO_HIDDEN 1
```

Now running a built-in command will show us the query that runs behind the scenes.

```
\dt+
********* QUERY **********
SELECT n.nspname as "Schema",
  c.relname as "Name",
  CASE c.relkind WHEN 'r' THEN 'table' WHEN 'v' THEN 'view' WHEN 'm' THEN 'materialized view' WHEN
'i' THEN 'index' WHEN 'S' THEN 'sequence' WHEN 's' THEN 'special' WHEN 'f' THEN 'foreign table' WHEN
'p' THEN 'table' END as "Type",
  pg_catalog.pg_get_userbyid(c.relowner) as "Owner",
  pg_catalog.pg_size_pretty(pg_catalog.pg_table_size(c.oid)) as "Size",
  pg_catalog.obj_description(c.oid, 'pg_class') as "Description"
FROM pg_catalog.pg_class c
     LEFT JOIN pg_catalog.pg_namespace n ON n.oid = c.relnamespace
WHERE c.relkind IN ('r','p','')
      AND n.nspname <> 'pg_catalog'
      AND n.nspname <> 'information_schema'
      AND n.nspname !~ '^pg_toast'
  AND pg_catalog.pg_table_is_visible(c.oid)
ORDER BY 1,2;
**************************
```

Cool! The query uses `pg_catalog.pg_class` and `pg_catalog.pg_namespace` among a handful of other functions.

Be sure to set that setting back to `0` so you don't have to see that extra detail every time.

```
\set ECHO_HIDDEN 0
```

Exploring the code behind the built in slash commands in `psql` is a great way to learn more about the internals of PostgreSQL.

> PostgreSQL's internal meta-data via the `pg_catalog` and `information_schema` objects provide all sorts of helpful information.

Summary

PostgreSQL gives us the basic tools needed to provide the meta data for a data dictionary directly in the database. Documenting your databases reduces the reliance on one person knowing all the secrets of what is actually in a given database.

The built-in `psql` commands provide a handy and powerful interface to query the data dictionary. Those queries can be used to help explore and learn how to access more of that data from your existing databases. By bringing in other meta-elements, such as data types, size on disk, and index coverage are all available to query, filter, report and visualize.

This post is the beginning of a full-blown data dictionary for your databases. Stay tuned!

Need help with your PostgreSQL servers or databases? Contact us to start the conversation!

By Ryan Lambert
Published September 24, 2018
Last Updated January 04, 2022
