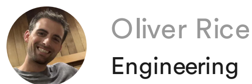
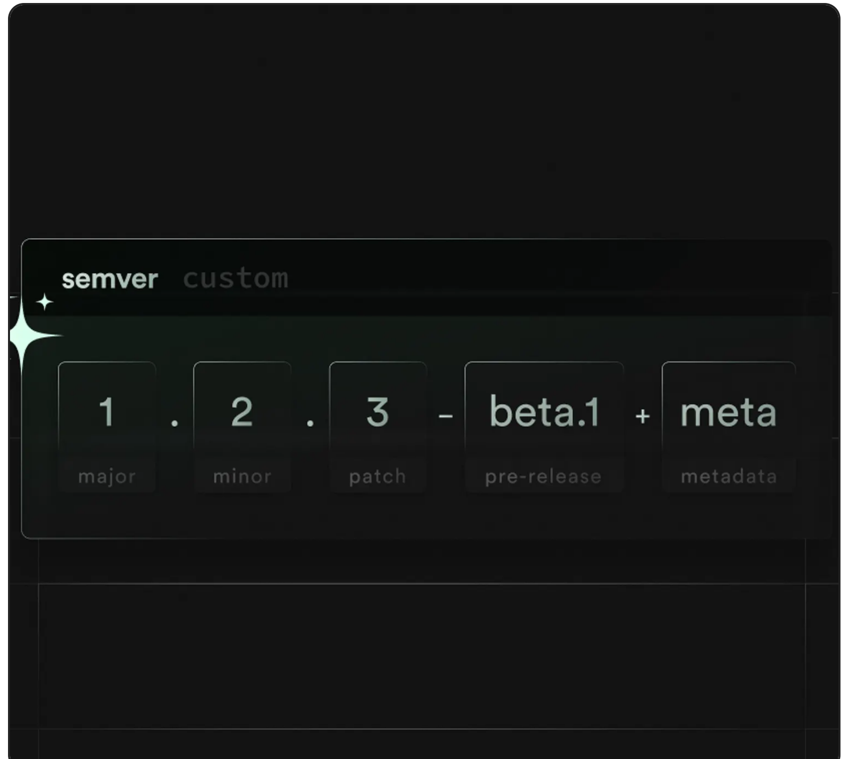
Back

Blog post

Type Constraints in 65 lines of SQL

2023-02-17 • 10 minute read


Oliver Rice
Engineering



PostgreSQL has a rich and extensible type system. Beyond enums and composite types, we can:

- apply data validation rules
- override comparison operators like

- create custom aggregations
- define casting rules between types

With a little effort, a user-defined type can feel indistinguishable from a built-in. In this article we focus on validation and ergonomics while quickly touching on a few other concepts.

To illustrate, we'll create an `semver` data type to represent Semantic Versioning values. We'll then add validation rules to make invalid states unrepresentable.

SemVer

A (very) loose primer on SemVer:

SemVer is a specification for representing software versions that communicate information about backwards compatibility. The type is typically represented as a string with 5 components.

Where `pre-release` and `metadata` are optional.

The intent of each component is outside the scope of this article but, as an example, incrementing the major version number notifies users that the release includes at least one backwards incompatible change.

For a concise representation of the full spec, check out the grammar.

SQL

For our purposes, we'll assume that the SemVer type is a critical component of the application that needs to be queried flexibly and efficiently.

Storing Components

To that end, we'll store each component of the version as a separate field on a composite type.

```
create type semver_components as (
    major int,
    minor int,
    patch int,
    pre_release text[],
    build_metadata text[]
);
```

We can create an instance of this type in SQL by casting a tuple as the `semver_components` type.

```
select
    (1, 2, 3, array['beta', '1'], array['meta']
-- returns: (1,2,3,{'beta','1'},{'meta'})
```

Unfortunately, our definition is far too permissive.

```
select
    (null, -500, null, array['?'], array['']):
```

```
-- returns: (,-500,,{'?'},{''
```

Our data type has no problem accepting invalid components. To list a few of the SemVer rules we violated:

- Major version must not be null
- Minor version must be ≥ 0
- Patch version must not be null
- Pre-release elements must only include characters [A-z0-9]
- Build metadata elements may not be empty strings

We need to add some validation rules to meet our “make invalid states unrepresentable” goal.

Validation

Domains are Postgres’ solution for optionally layering constraints over a data type. Domains are to types what check constraints are to tables. If you’re not familiar with check constraints, you can think of them as equivalent to zod/pydantic in javascript/python.

Let's codify some SemVer rules, layer them on the `semver_components` type, and give the new domain a friendly name.

```
create domain semver
    as semver_components
    check (
        -- major: non-null positive integer
        (value).major is not null and (value).m
        -- minor: non-null positive integer
        and (value).minor is not null and (valu
        -- patch: non-null positive integer
        and (value).patch is not null and (valu
        and semver_elements_match_regex(
            (value).pre_release,
            '^[A-z0-9]{1,255}$'
```

```
        )
        and semver_elements_match_regex(
            (value).build_metadata,
            '^[A-z0-9\.]{1,255}$'
        )
    );
```

which references a helper function:

```
create or replace function semver_elements_matc
    parts text[],
    regex text
)
returns bool
language sql
as $$
    -- validates that *parts* nullable array of
    -- where each element of *parts* matches *r
    select
        $1 is null
        or (
            (
                select (
                    bool_and(pr_arr.elem is not
                    and bool_and(pr_arr.elem ~
                )
                from
                    unnest($1) pr_arr(elem)
            )
            and array_length($1, 1) > 0
        )
$$;
```

Now, if we repeat our positive and negative test cases using the `semver` type (vs `semver_components`) we still accept valid states:

```
-- Success Case
select
    (1, 2, 3, array['beta', '1'], array['meta']
-- returns: (1,2,3,{'beta','1'},{'meta'})
```

while invalid states are rejected with an error:

```
-- Failure Case
select
    (null, -500, null, array['?'], array[''])::
-- ERROR:  value for domain semver violates che
-- SQL state: 23514
```

Testing

Our validation doesn't have to be called manually. The `semver` domain can be used anywhere you'd use the `semver_components` type and the validations are automatically applied.

```
-- A table with a semver column
create table package_version(
    id bigserial primary key,
    package_name text not null,
    package_semver semver not null -- semver co
);

-- Insert some valid records
insert into package_version( package_name, pack
values
    ('supabase-js', (2, 2, 3, null, null)),
    ('supabase-js', (2, 0, 0, array['rc', '1'],
);

-- Attempt to insert an invalid record (major i
insert into package_version( package_name, pack
values
    ('invalid-js', (null, 1, 0, array['asdf'],
-- ERROR:  value for domain semver violates che
```

Good stuff!

We're 48 lines of SQL in and have solved for making invalid states unrepresentable. Now lets think about ergonomics.

Displaying

Now that our data type is well constrained, you might notice that selecting values from a `semver` typed

column returns a tuple, rather than the SemVer string we're used to seeing.

```
select
    *
from
    package_version
/*
id | package_name |    package_semver
-----------------------------------------
 1 | supabase-js  |         (2,2,3,,)
 2 | supabase-js  | (2,0,0,"{rc,1}",)
*/
```

For example: `(2,0,0,"{rc,1}",)` vs `2.0.0-rc.1`

We could work around that problem with some custom casts, but I'd recommend keeping everything explicit with a function call.

```
create or replace function semver_to_text(semve
    returns text
    immutable
    language sql
as $$
    select
        format('%s.%s.%s', $1.major, $1.minor,
        || case
            when $1.pre_release is null then ''
            else format('-%s', array_to_string(
        end
        || case
            when $1.build_metadata is null then
            else format('+%s', array_to_string(
        end
$$;
```

Which allows us to query the `package_version` table and retrieve a string representation of the data.

```
select
    id,
    package_name,
    semver_to_text(package_semver) as ver -- ca
```

```
from
    package_version
/*
id | package_name |   ver
-----------------------------
 1 | supabase-js  |      2.2.3
 2 | supabase-js  | 2.0.0-rc.1
*/
```

Or, better yet, use a generated column

```
create table package_version(
    id bigserial primary key,
    package_name text not null,
    package_semver semver not null,
  semver_text text generated always as (semver_
);
```

so the text representation is persisted along with the `semver` type and incurs no query/filter penalty.

Other Tricks

Postgres provides all the tools you could want to make your data types/domains work with SQL as seamlessly as builtins.

For example, you could:

- add convenience functions to parse a semver type from text
- override the equality operator (`=`) to correctly reflect that versions differing only in build metadata are considered equal
- add a `max` function to efficiently query for the newest version of each package from within the database

to name a few.

Aligning the right parts of your business' logic with the database can dramatically improve throughput, decrease IO, and simplify application code.

Conclusion

Admittedly, building performant and ergonomic custom data types in Postgres involves a lot of ceremony.

That said, in cases where:

- the type's data integrity is critical
- the type is well specified
- the type's spec does not change (or changes infrequently)

Teaching Postgres to have first class support for your custom type can be transformative for data integrity and performance.

Share this article

Next post

How to build a real-time multiplayer game with Flutter Flame
14 February 2023

Related articles

- Type Constraints in 65 lines of SQL
- How to build a real-time multiplayer game with Flutter Flame
- Supabase Beta January 2023
- Supabase Clippy: ChatGPT for Supabase Docs
- Storing OpenAI embeddings in Postgres with pgvector

View all posts

Build in a weekend, scale to millions

Start your project

© Supabase Inc