ggsql: A grammar of graphics for SQL

Introducing ggsql, a grammar of graphics for SQL that lets you describe visualizations directly inside SQL queries

Thomas Lin Pedersen, Teun Van den Brand, George Stagg, Hadley Wickham

Apr 20, 2026

SQL

VISUALIZATION

Today, we are super excited to announce the alpha-release of ggsql . As the name suggests, ggsql is an implementation of the grammar of graphics based on SQL syntax, bringing rich, structured visualization support to SQL. It is ready for use in Quarto, Jupyter notebooks, Positron and VS Code among others.

In this post we will go over some of the motivations that lead us to develop this tool, as well as give you ample examples of its use; so you can hopefully get as excited about it as we are.

Meet ggsql

Before we discuss the why, let's see what ggsql is all about with some examples.

The first plot

To get our feet wet, lets start with the hello-world of visualizations: A scatterplot, using the built-in penguins dataset:

```
VISUALIZE bill_len AS x, bill_dep AS y FROM ggsql:penguins
DRAW point
```

That wasn't too bad. Sure, it has the verbosity of SQL, but that also means that you can speak your plot code out loud and understand what it does. We can break down what is going on here line-by-line:

1. We initiate the visual query with **`VISUALIZE`** and provide a mapping from the built-in penguins dataset, relating **`x`** to the data in the **`bill_len`** column, and **`y`** in the **`bill_dep`** column.
2. We draw a point layer that, by default, uses the mapping we defined at the top.

With this in place, we can begin to add to the visualization:

```
VISUALIZE bill_len AS x, bill_dep AS y, species AS color FROM ggsql:pe
DRAW point
```

We see that a single addition to the mappings adds colored categories to the plot. This gradual evolution of plot code is one of the biggest strengths of the grammar of graphics. There are no predefined plot types, only modular parts that can be combined, added, and removed. To further emphasize this, let's add a smooth regression line to the plot:

```
VISUALIZE bill_len AS x, bill_dep AS y, species AS color FROM ggsql:pe
DRAW point
DRAW smooth
```

We add a new layer on top of the point layer. This layer also borrows the same mapping as the point layer. Since we color by species, the smooth line is split into one for each species.

We can continue doing this, adding more mappings, adding or swapping layers, controlling how scales are applied etc until we arrive at the plot we need, however simple or complicated it may be. In the above example we may well end up deciding we are more interested in looking at the distribution of species across the three islands the data was collected from:

```
VISUALIZE island AS x, species AS color FROM ggsql:penguins
DRAW bar
```

While a completely different plot, you can see how much of the code from the previous plot carries over.

A complete example

With our first couple of plots under the belt, let's move on to a complete example. It will contains parts we have not seen before, but don't worry, we will go through it below, even the parts we've already seen before. The example is an adaptation of a visualization created by Jack Davison for TidyTuesday.

```
WITH astronauts AS (
  SELECT * FROM 'astronauts.parquet'
  QUALIFY ROW_NUMBER() OVER (
    PARTITION BY name
    ORDER BY mission_number DESC
  ) = 1
)
SELECT
  *,
  year_of_selection - year_of_birth AS age,
  'Age at selection' AS category
FROM astronauts
UNION ALL
SELECT
  *,
  year_of_mission - year_of_birth AS age,
  'Age at mission' AS category
FROM astronauts

VISUALIZE age AS x, category AS fill
DRAW histogram
  SETTING binwidth => 1, position => 'identity'
PLACE rule
  SETTING x => (34, 44), linetype => 'dotted'
PLACE text
  SETTING
    x => (34, 44, 60),
    y => (66, 49, 20),
    label => (
      'Mean age at selection = 34',
      'Mean age at mission = 44',
      'John Glenn was 77\non his last mission -\nthe oldest person to\
    ),
    hjust => 'left',
    vjust => 'top',
    offset => (10, 0)
SCALE fill TO accent
LABEL
  title => 'How old are astronauts on their most recent mission?',
  subtitle => 'Age of astronauts when they were selected and when they
  x => 'Age of astronaut (years)',
  fill => null
```

That was a lot of code, but on the flip-side we have now covered a lot of the most important aspects of the syntax with one example.

At the topmost level there are two parts to this query: The SQL query, and the visualization query. The SQL query is anything from the beginning to the `VISUALIZE` clause. It is your standard SQL, and it accepts anything your backend accepts (in this blog post we use a DuckDB backend). The result of the query is funnelled directly into the visualization rather than being returned as a table like you'd normally expect.

Since the point of this post is not to teach you SQL we won't spend much more time discussing the SQL query part. The main take away is that everything before the `VISUALIZE` clause is pure SQL, any resulting table is automatically used by your visualization, and any table or CTE created there is available for referencing in the visualization query.

As we saw in the first examples, the SQL query part is optional. If your data is already in the right shape for plotting you can skip it and instead name the source directly in the `VISUALIZE` clause:

```
VISUALIZE year_of_selection AS x, year_of_mission AS y FROM 'astronauts
```

Now, let's look at the visual query — everything from `VISUALIZE` and onwards. `VISUALIZE` marks the end of the SQL query and the beginning of the visualization query (or `VISUALISE` for those who prefer UK spelling). It can stand on its own or, as we do here, have one or more mappings which will become defaults for every

subsequent layer. Mappings are purely for relating data to abstract visual properties. A mapping is like a `SELECT` where you alias columns to a visual properties (called *aesthetics* in the grammar of graphics). In the visualization above we say that the age column holds the values used for `x` (position along the x axis) and the category column holds the values used for `fill` (the fill color of the entity). We do not say anything about how to draw it yet.

Following the `VISUALIZE` query we have a `DRAW` clause. `DRAW` is how we add layers to our visualization. There is a large selection of different layer types in ggsql. Some are straightforward: e.g. `point` for drawing a scatterplot. Some are more involved: `histogram` (which we use here) requires calculation of derived statistics like binned count. A visualization can have any number of layers and layers will be rendered in the sequence they are defined. `DRAW` has a sibling clause called `PLACE`. It is used for annotation and works like `DRAW` except it doesn't get data from a table but rather as provided literal values. It follows that our visualization above contains three layers: A histogram layer showing data from our table, a rule annotation layer showing precomputed mean values for each category, and a text annotation layer adding context to the visualization. It is worth mentioning that a layer does not correspond to a single graphical entity. Like with the text layer above, each layer can render multiple separate entities of its type so there is no need to have e.g. 3 line layers to render line plots for 3 different categories.

After the `DRAW` and `PLACE` clauses we have a `SCALE` clause. This clause controls how data values are translated into values that are meaningful for the aesthetic. In our case, the `category` column holds the strings "Age at mission" and "Age at selection" which doesn't in itself translate to a color value. The clause `SCALE fill TO accent` tells ggsql to use the "accent" color palette when converting the values mapped to fill to actual colors. Scales can be used for much more, like applying transformations to continuous data, defining break points, and setting specific scale types (like ordinal or binned).

The last clause in our visual query is `LABEL` which allows us to add or modify various text labels like title, subtitle, and axis and legend titles.

Stepping back

That was a mouthful. But there are two very silvery linings to it all:

1. You now know the most important aspects of the syntax (there are more, of course, but you can grow into that)
2. Many visualization queries will be much simpler than the one above

We have already seen examples of shorter visual queries above but let's continue with a boxplot of astronaut birth year split by sex:

```
VISUALIZE sex AS x, year_of_birth AS y FROM 'astronauts.parquet'
DRAW boxplot
```

That's much shorter than the last plot code but still, if you are coming from a different plotting system you may even think this is overly verbose (e.g. compared to something like `boxplot(astronauts.sex, astronauts.year_of_birth)`). Yes, it is longer, but it is also more structured, composable, and self-descriptive. These features (which are a direct result of its grammar of graphics lineage) means that both you and your future LLM coding buddy will have an easier time internalizing the workings of *all* types of plots that can be made. The 18 years of dominance of ggplot2 (which shares these features) in the R ecosystem is a testament to this.

As an example, let's change the above plot to instead show the same relationship as a jittered scatterplot.

```
VISUALIZE sex AS x, year_of_birth AS y FROM 'astronauts.parquet'
DRAW point
```

```
  SETTING position => 'jitter'
```

Or perhaps the jitter follows the distribution of the data so it doubles as a violin plot:

```
VISUALIZE sex AS x, year_of_birth AS y FROM 'astronauts.parquet'
DRAW point
  SETTING position => 'jitter', distribution => 'density'
```

As you can see the syntax and composable nature makes visualization iteration very ergonomic, something that is extremely valuable in both explorative analyses and

visualization design.

Why ggsql?

Writing a new visualization library from scratch is a big task and you might wonder why we're doing it again. Some of the reasons are:

- We want to engage with and help data analysts and data scientists that predominantly work in SQL
- SQL and the grammar of graphics fit together extremely well
- We want to create an extremely powerful, code-based, visualization tool that doesn't require an entire programming language (like R or python)
- LLMs speak SQL very well and also presents a new interface to data visualization creation
- We have learned so much from 18 years of ggplot2 development that we're excited to apply to a blank canvas

Let's discuss each in turn.

Hello, SQL user!

```
SELECT 'Hello' AS greetings FROM SQL_users
WHERE other_language NOT IN ('R', 'Python')
```

While first R and then Python captured all the attention of the data science revolution, SQL chugged along as the reliable and powerful workhorse beneath it all. There are many people who work with data that do so only or predominantly in SQL. The choice they have for visualizing their data are often suboptimal in our view:

- Export the data and use R or Python which may not be within their comfort zone
- Use a GUI-based BI tool with poor support for reproducibility

- Rely on one of the few tools that exist for creating visualizations directly within the query that we feel are not powerful or ergonomic enough

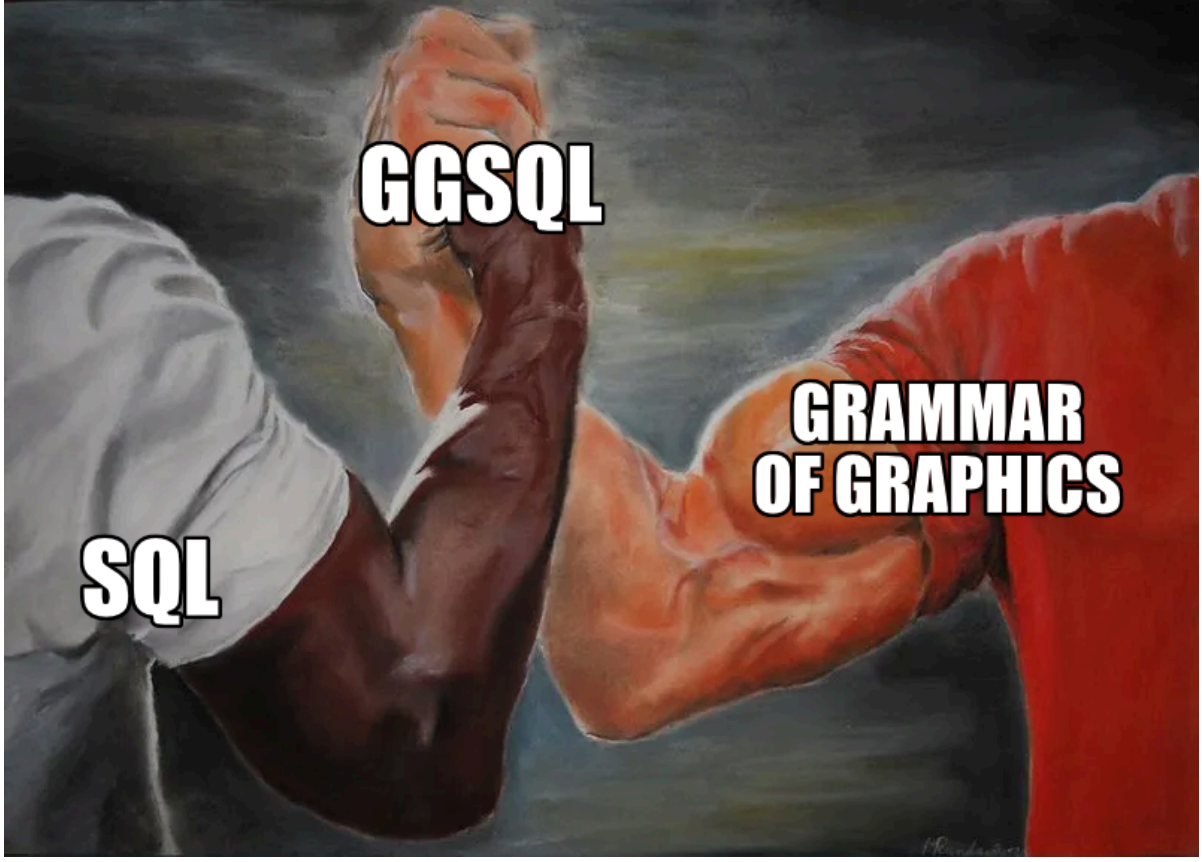
Our goal when designing ggsql was that the syntax should immediately make sense to SQL users, tapping into their expectation of composable, declarative clauses.

Apart from offering a better way to visualize their data, ggsql is also a way to invite SQL users into our rich ecosystem of code based report generation and sharing build on top of Quarto .

Declarative wrangling, declarative visualization

If you are reading this with no prior knowledge of SQL, here's a very brief recap: SQL is a domain specific language for manipulating relational data stored in one or more tables. The syntax is based on the concept of relational algebra which is a structured way to think about data manipulation operations. The semantics defines a set of modular operations that are declarative rather than functional, allowing the user to compose very powerful and custom manipulations using a well-defined set of operations.

If you are reading this with no prior knowledge of the grammar of graphics, here's a very brief recap: The grammar of graphics is a theoretical deconstruction of the concepts of data visualization into its modular parts. While purely theoretical, tools such as ggplot2 have implemented the idea in practice. The semantics defines a set of modular operations that are declarative rather than functional, allowing the user to compose very powerful and custom visualizations using a well-defined set of operations.



From the above, slightly hyperbolic, overview it is clear that both SQL and the grammar of graphics have a lot of commonality in their approach to their respective domains. Together they can offer a very powerful and natural solution to the full pipeline from raw data to final visualization.

No runtime, no problem

Why does it matter that ggplot2 and plotnine requires R and Python installed respectively? There are clear benefits to a single, focused executable to handle data visualization:

- Embedding a small executable in other tools is much easier than bundling R/Python (or requiring them to be installed)
- A smaller scope makes it easier to sandbox and prevent malicious code execution (either deliberately or in error)

Both of the above points make ggsql a much more compelling option for integrating into tools such as AI agents assisting you in data analysis, or code based reporting tools that may execute code in different environments.

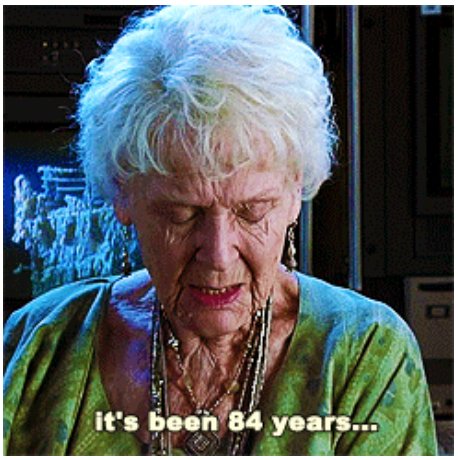
You may think we have had to swallow some bitter pills by moving away from an interpreted language, but it has also given us a lot. Most importantly, the rigid

structure means that we can execute the whole data pipeline as a single SQL query per layer on the backend. This means that if you want to create a bar plot of 10 billion transactions you only ever fetch the count values for each bar from your data warehouse, not the 10 billion rows of data. The same is true for more complicated layer types such as boxplots and density plots. This is in stark contrast to most visualization tools which must first materialize the complete data, then perform the necessary computations on it, then plot it.

```
SELECT pod_door FROM bay WHERE closed
```

LLMs have proven very effective at translating natural language into SQL, and we're bullish that they can be just as effective with ggsql. We've already seen evidence of this in querychat , where you can now visually explore data using natural language via ggsql. And, since ggsql is a much safer and lighter runtime than R or Python, you can much more confidently ship coding agents into a production environment.

We are now wise beyond our years



18 years of ggplot2 development and maintenance also means 18 years of thinking about data visualization syntax, use, and design. While not trying to be boastful we do believe that gives us some expert knowledge on the subject matter. However, not all of this knowledge can be poured back into ggplot2. There are decisions and expectations established many years ago that we have to honor, or at least only challenge very gradually (which we do on occasions).

ggsql is a blank slate. Not only in the sense that we are building it from the ground up, but also in that it is built for an environment with no established expectations for a visualization tool. I cannot stress how liberating and invigorating this has felt, and I am positive that this shines through in how ggsql feels to the user.

The future

We are nearing the end of a rather long announcement — thanks for sticking with us. In the very first line we called this an alpha-release which implies that we are not done yet. To get you as excited about the future as you hopefully are about the present state of ggsql, here is a non-exhaustive list of things we want to add.

- New high-performance writer, written from the ground up in Rust
- Theming infrastructure
- Interactivity
- End-to-end deployment flow from Posit Workbench/Positron to Connect
- Fully fledged ggsql language server and code formatter
- Support for spatial data

What does this mean for ggplot2 development

If you are a current ggplot2 user you may have read this with a mix of fear and excitement (or maybe just one of them). Does this mean that we are leaving ggplot2 behind at Posit to focus on our new shiny toy? Not at all! ggplot2 is very mature and stable at this point but we will continue to support and build it out. We also hope that ggsql can pay back all the experience from ggplot2 that went into its development by informing new features in ggplot2.

Want more?

If you can't wait to learn more about ggsql and begin to use it you can head to the Getting started section of the ggsql website for installation instructions, and a tutorial, or head straight to the documentation to discover everything ggsql is capable of. We can't wait for you to try it out and hear about your experiences with it.

More On Visualization

ggsql

A SQL extension for declarative data visualisation based on the Grammar of Graphics

49

Shiny for Python

Easy interactive web applications with Python

1.7k

rstudio

RStudio is an integrated development environment (IDE) for R

5k

Stay Connected

Get the latest updates on Posit open source projects and insights from our community.

Follow us on GitHub

Subscribe via RSS