towards data science

Sign in Get started

Follow 578K Followers · Editors' Picks Features Deep Dives

You have **2** free member-only stories left this month.
Sign up for Medium and get an extra one

How To Visualize Databases as Network Graphs in Python

Build a Dash web application to interactively explore database structures.

Thomas Baumgartner 2 days ago · 14 min read ★

Original image by geralt (Pixabay). Edited by the author.

At work I recently faced the challenge of having to analyze the data model of an SQL database consisting of more than 500 tables with thousands of relations. At this scale, the built-in visualization function of *phpMyAdmin* is insufficient for getting a deep understanding of the structure. What I needed was a tool in which I can apply various filters (e.g., table and column names, row counts, number of connections), and then view the filtered tables and their relations in an easy-to-grasp visual representation. So, I decided to build such a tool using Python.

Note that, in the interest of being concise and focusing on the important points, I don't provide the complete code here (approximately 1000 lines). Also, I removed comment lines to keep the code sections brief.

Getting the data

To visualize the structure of a database, we first need to get data on table and column names as well as primary and foreign keys. Luckily, SQL databases come with a very good source of such information: the information schema. That is a meta-database, i.e., a database describing other databases — not their contents, but their structure, constraints, data types, and more. It consists of a bunch of tables, but we will need only three of them:

TABLES: From here we get the names of all tables, the databases (schemas) to which they belong, their row counts,

and optional comments describing the tables.

COLUMNS: This table tells us for each column in all our tables the column name, and to which database and table it belongs. Furthermore, we can get the data types, the default values, and comments describing the columns.

KEY_COLUMN_USAGE: For our purposes, this is the most important table. It tells us how the tables are connected, i.e., it contains one record for each reference from one column to another. Also, it identifies the primary key columns.

I decided to export these three tables in JSON format using my *phpMyAdmin* access to the database. Of course, you could also access the information schema directly (for instance with *sshtunnel* and *sqlalchemy*), but I wanted a solution that works offline. The following code loads data from an information schema table into a *pandas* (imported as `pd`) data frame.

Due to the structure of the JSON export (the file contains a header we are not interested in), we need to extract the data column and remove empty rows. Then we make a new frame out of the data contained in the first cell of our data frame. The resulting data frame contains the same data as the information schema table whose JSON file was passed to the function `load_from_json`.

Now we use this function to get the data on tables, columns, and references from the information schema tables described above. We start with the *TABLES* table.

With the `iterrows` function of the *pandas* data frame we loop through all rows, each of which gives us the name, database (schema), comment, and rows count of a table. The last two lines deserve a bit of explanation: `information_schema` is a module in which I defined structures for storing data on tables, columns, and their relationships. With `information_schema.Table(...)` we create an object that collects all data on a specific table, and the last line of code adds this table to a `TableCollection` object, which looks like this:

```
1  class TableCollection():
2
3      def __init__(self):
4
5          self.tables = {}
6
```

```
7
8       def __len__(self) -> int:
9
10          return len(self.tables)
11
12
13      def add_table(self, table: Table):
14
```

Inside this table collection, the table objects are stored in a dictionary which uses a tuple consisting of database (schema) name and table name as key (the same table name can potentially be used in multiple databases).

Now that we have all the tables inside the `TableCollection` structure, it is time to add information on columns, which we load from the JSON export of the *COLUMNS* table of the information schema.

```
1   def make_cols(self):
2
3       df = self.load_from_json(self.path + 'COLUMNS.json')
4       for index, row in df.iterrows():
5           name = row['COLUMN_NAME']
6           db = row['TABLE_SCHEMA']
7           table = row['TABLE_NAME']
8           dtype = row['COLUMN_TYPE']
9           default = row['COLUMN_DEFAULT']
10          comment = row['COLUMN_COMMENT']
11          col = information_schema.Column(name, dtype, default, comment)
12          self.tables.tables[(db, table)].add_col(col)
```

Again, we loop through all rows, each of which describes one

column of one of the tables whose names et cetera we already loaded. In the last two lines we create an object which stores the properties of a specific column, and then add this `information_schema.Column` object to the table it belongs to by passing it to a function belonging to the appropriate dictionary entry in the table collection.

Now we have information on tables and their columns in our `TableCollection` object. What's still missing is information on primary and foreign keys, which we obtain from the *KEY_COLUMN_USAGE* table of the information schema.

```
def set_keys(self):

    df = self.load_from_json(self.path + 'KEY_COLUMN_USAGE.json')
    for index, row in df.iterrows():
        col_name = row['COLUMN_NAME']
        db = row['TABLE_SCHEMA']
        table = row['TABLE_NAME']
        if row['CONSTRAINT_NAME'].lower() == 'primary':
            is_primary = True
        else:
            is_primary = False
        ref_db = row['REFERENCED_TABLE_SCHEMA']
        ref_table = row['REFERENCED_TABLE_NAME']
        ref_col = row['REFERENCED_COLUMN_NAME']
        self.tables.tables[(db, table)].cols[col_name].is_primary_key =
        if not ref_col is None:
            tab = self.tables.tables[(db, table)]
```

A column which is a primary key in the table it belongs to is marked by the value "PRIMARY" in the column "CONSTRAINT_NAME" in *KEY_COLUMN_USAGE*. We use this

information to set the corresponding property `is_primary` of the `Column` object. If a column is a foreign key, the referenced database, table, and column are specified in "REFERENCED_TABLE_SCHEMA", "REFERENCED_TABLE_NAME", and "REFERENCED_COLUMN_NAME". These data are added to the `Column` object in the last line of the code above.

Creating the graph

In case you are not familiar with graph theory: A graph is a mathematical structure consisting of a set of objects (nodes), and a set of connections between these objects (edges). Such a structure is exactly what we need to explore the data model: The tables will be the nodes of the graph, and the references between tables will be its edges.

We will use the *networkx* package to create the graph, which requires four steps:

1. Import the package: `import networkx as nx`
2. Initialize a graph object, for instance: `g = nx.Graph()`
3. Add nodes using `g.add_node(node)`, where `node` can be any hashable object except `None`. We can also pass keyword arguments which are interpreted as node attributes.
4. Add edges between our nodes using `g.add_edge(node_1,`

`node_2)`, optionally also with keyword arguments containing edge attributes. Note that in principle one can skip step 3, since nodes will automatically be created along with the edges in case they don't exist.

But we're not done after building the graph. We also want to manipulate it later (remember I was dealing with more than 500 tables and needed to apply filters), and for that we write the `GraphManipulator` class:

```
class GraphManipulator():

    def __init__(self):

        self.graph = nx.DiGraph()
        self.graph_backup = None
        self.tables = None


    def load(self, tables: TableCollection):

        self.tables = tables
        self._make_nodes()
        self._make_edges()
        self._add_edge_count_attribs()
```

Upon instantiation of a `GraphManipulator` object, a graph object is created. Note that we use `DiGraph()` here (the *networkx* package was imported as `nx`), which creates a directed graph, i.e., when an edge is created, the order of the two connected nodes matters. We do that so we can later draw arrows pointing from the referencing table to the referenced table.

The `load` function takes our `TableCollection` object as argument, so the `GraphManipulator` object has access to the data we obtained from the information schema. The graph is then built by the three functions whose names start with an underscore, and finally a copy of the graph object is saved for restoring it after the use of filters (more on that later). Let's have a look at the three functions in which the graph is constructed.

We start with the nodes:

```
def _make_nodes(self):

    for tab in self.tables.tables.values():
        identifier = self._build_node_id(tab.db, tab.name)
        f_keys = tab.get_foreign_keys()
        references = []
        for key in f_keys:
            ref_str = ''
            col = tab.cols[key]
            N_refs = len(col.referenced_cols)
            for i in range(N_refs):
                ref_db = col.referenced_dbs[i]
                ref_table = col.referenced_tables[i]
                ref_col = col.referenced_cols[i]
                ref_id = self._build_node_id(ref_db, ref_table)
                ref_str += '%s:%s, ' % (ref_id, ref_col)
            ref_str = ref_str[:-2]
            references.append(ref_str)
        attributes = {'db': tab.db,
                      'table': tab.name,
                      'primary_key': tab.get_primary_key(),
                      'foreign_keys': f_keys,
                      'references': references,
                      'cols': list(tab.cols.keys()),
                      'N_rows': tab.N_rows,
```

The code above loops through all tables whose information is stored in the `TableCollection` object (here `self.tables`). The function `_build_node_id` simply returns a string consisting of database name and table name. This string is unique for each table and serves as node object. With `tab.get_foreign_keys()` we get a list of the columns in the table which serve as foreign keys, i.e., they reference primary key columns of other tables.

Next, we loop through all the foreign keys in the list `f_keys`, adding a string to the list `references` for each one of them. This string contains the database (schema), table, and column name of the referenced column. Finally, we put all the properties we are interested in, such as the reference list we just compiled, into the `attributes` dictionary and add a node to our graph with these attributes attached. Note that the number of edges in both directions (the last two entries in the dictionary) are set later.

Now that we have all our nodes, it is time to add the edges to our graph.

```
def _make_edges(self):

    for tab in self.tables.tables.values():
        id_from = self._build_node_id(tab.db, tab.name)
        for col in tab.cols.values():
            for i in range(len(col.referenced_tables)):
                ref_tab = col.referenced_tables[i]
                ref_db = col.referenced_dbs[i]
```

```
                id_to = self._build_node_id(ref_db, ref_tab)
                self.graph.add_edge(id_from, id_to)
```

Again, we loop through all the tables, and for each of them through all contained columns. The innermost loop looks at all the references of a column (one column can potentially reference several others). The last line of the code above adds an edge pointing from the referencing column to the referenced column (remember, we have a directed graph here) to the graph object.

I thought it would be useful to have both the number of references from a given table to others and the number of references from other tables to that table in the node attributes.

```
def _add_edge_count_attribs(self):

    for node_1 in self.graph.nodes:
        N_down = 0
        N_up = 0
        for node_2 in self.graph.nodes:
            N_down += self.graph.number_of_edges(node_1, node_2)
            N_up += self.graph.number_of_edges(node_2, node_1)
        self.graph.nodes[node_1]['N_edges_upstream'] = N_up
        self.graph.nodes[node_1]['N_edges_downstream'] = N_down
```

I named these attributes `N_edges_downstream` and `N_edges_upstream`, and obtained their values using the `number_of_edges` function of the graph object as shown above. As you can see in the code, accessing node attributes is easy: `graph.nodes[name_of_node]['name_of_attribute']`. Now our

graph is complete, and we can visualize it. But before we do that, let us briefly talk about filtering.

Filtering by node attributes

As I said in the beginning of this article, I was dealing with a database consisting of more than 500 tables. Clearly, this is too much information to take in all at once for most human brains. Therefore, I equipped my Python script with filtering options. The principle is simple and consists of these steps:

1. Copy the backup of the graph we saved right after building it to the graph object, like this: `self.graph = self.graph_backup.copy(as_view=False)`
   This is necessary because we want to apply the filters to the original graph, not to an already filtered one.
2. Loop through all the nodes in the graph.
3. Compare the node attributes to your filter criteria, and if any of these comparisons says that the node does not match the criteria, remove it using `graph.remove_node(node)`.

After this procedure, only the tables matching the filter criteria will be left as nodes in your graph. The edges between them remain unaffected, whereas edges connecting them to removed nodes are no longer part of the graph. I implemented the following filters:

- Names of databases (schemas), tables, and columns (string comparison with wildcards).
- Number of rows.
- Number of columns.
- Number of connections, i.e., references to or from other tables.

Sometimes it is useful to filter certain tables, e.g., all with a given string in their name, and then also display all the tables connected to those left after filtering. For this purpose, I wrote the following function.

```
def extend_depth(self, depth: int):

    g0 = self.graph_backup
    keep_list = []

    for i in range(depth):
        for node in self.graph.nodes:
            keep_list.append(node)
            neighbors = (list(g0.neighbors(node)) +
                         list(g0.predecessors(node)))
            for neighbor in neighbors:
                keep_list.append(neighbor)
        self.graph = g0.copy(as_view=False)
        node_list = list(self.graph.nodes)
        for node in node_list:
            if not node in keep_list:
```

The integer argument `depth` determines how far this extension by connections goes. The case `depth = 1` corresponds to what I

said above, whereas `depth = 2` includes not only the connected tables but also the ones connected to these tables, and so on. In other words, all nodes will be kept which are no further than `depth` edges away from the filtered nodes (`self.graph.nodes`).

We accomplish that by looping through all nodes in the filtered graph and adding them as well as their neighboring nodes to a list of nodes to keep. Note that for a directed graph, `DiGraph.neighbors(node)` and `DiGraph.predecessors(node)` will give us the successors and predecessors, respectively, of a node, i.e., its neighbors in both directions. For an undirected graph, you would only use `Graph.neighbors(node)`. Once we have our list of nodes to keep, we simply overwrite our graph with the unfiltered backup and then remove all nodes which are not on the list. If `depth` is greater than one, a new iteration will start with `self.graph` containing all the nodes we just put on the keep list.

Visualizing the graph in a web application

The *networkx* package provides basic visualization functionality, but this is not where its strengths lie. I decided to go for a web application using *Dash* because that allows an interactive exploration of the database structure. If you're not familiar with Dash, have a look at its documentation and check out the *plotly* visualization examples to get an impression of what you can do with it. You can also find some useful articles on using *Dash* here on Medium.

For my purpose of visualizing network graphs and applying filters, I found these *Dash / plotly* features particularly useful:

- You can build a web application (which can also run locally) without a lot of coding.
- You don't need to write HTML or JavaScript code, unless you want to extend your application beyond the scope offered by *Dash*.
- You don't need to reinvent the wheel for your visualizations, since *plotly* already comes with features such as zooming, panning, displaying annotations upon mouse hover events, and the ability to save plots in PNG format.

For details on how to build web applications with *Dash*, please follow the link I provided above. Here I will just give a brief outline of the application, and then discuss the interesting part, i.e., the code for visualizing the graph. These are the basic steps for creating the web application:

1. Import the modules you need, in this case `dash`, `dash_core_components` (as `dcc`), `dash_bootstrap_components` (as `dbc`), and `dash_html_components` (as `html`) for creating the application and adding controls. Since we want to access the contents of our controls in callback functions, we will also need this:
   `from dash.dependencies import Input, Output`
2. Create a Dash application object, like this:

`app = dash.Dash(...)`

In this line you can do a bunch of things by passing arguments to the function. For instance, you can change the appearance of your GUI by specifying a CSS file.

3. Create all the controls (text labels, input fields, dropdowns, etc.) you need. With *dash_bootstrap_components* you can easily arrange them in rows and columns to create a responsive layout. The drawing area is a `dash_core_components.Graph` object (not to be confused with our network graph).
4. These groups of controls (columns inside rows) are collected in a container object and inserted into the `layout` property of our Dash application object:
`app.layout = html.Div(dbc.Container([...]))`
5. Finally, we link functions to our controls using the `@app.callback` decorator, as described here.

Now let's talk about how to draw the network graph. The corresponding code is executed when the callback of the "Update graph" button is triggered. The return value of this callback function is a `plotly.graph_objs.Figure` object, and its output is set to the `'figure'` property of the drawing area (`dash_core_components.Graph` object).

Before the actual drawing happens, we need to compute the positions of the nodes of our network graph. Luckily, the

networkx package can do the heavy lifting for us, as it contains layout functions for arranging the nodes in useful ways. I added a dropdown menu in the GUI, allowing the user to select which layout function they want to apply. I implemented the following options (*networkx* is imported as `nx`):

- `pos = nx.layout.planar_layout(graph)`
  This function attempts to position the nodes such that all edges can be drawn without crossings, thus creating a tidy visualization. If this is not possible, it raises an error.
- `pos = nx.layout.shell_layout(graph, nlist=node_list)`
  This will arrange the nodes in concentric circles (shells) according to the contents of `nlist` (list of lists of nodes). I chose to assign the nodes to shells based on their number of edges (the one with the most connections is placed in the center). This can be useful for getting an overview when there are few nodes with many and many nodes with few edges to be displayed.
- `pos = nx.layout.spring_layout(graph)`
  Applying this layout function creates a force-directed layout, i.e., edges act like springs, and nodes repel each other. By assigning a weight attribute to the nodes, the attractive force of the springs can be changed individually (otherwise it is the same value for all nodes). I find this layout useful for creating order in non-planar graphs where all nodes have a similar number of edges.
- `pos = nx.layout.kamada_kawai_layout(graph)`

> The Kamada-Kawai algorithm also produces a force-directed layout but factors in the graph distance (number of edges in the shortest path) of each pair of nodes. It is useful for identifying clusters of strongly connected nodes.

After computing the node positions by calling one of the layout functions, these positions are passed to the graph object like this: `nx.set_node_attributes(graph, name='pos', values=pos)`

Now that we have the node positions available as node attributes, we can start drawing. We will begin with the edges.

```
line = {'width': 1,
        'color': '#FFFFFF'}

edge_trace = go.Scatter(x=[], y=[], line=line,
                        hoverinfo='none', mode='lines')

annotations = []

for edge in graph.edges():
    x0, y0 = graph.nodes[edge[0]]['pos']
    x1, y1 = graph.nodes[edge[1]]['pos']
    edge_trace['x'] += tuple([x0, x1])
    edge_trace['y'] += tuple([y0, y1])
    edge_annot = {'x': x1,
                  'y': y1,
                  'xref': 'x',
                  'yref': 'y',
                  'text': '',
                  'showarrow': True,
                  'axref': 'x',
                  'ayref': 'y',
                  'ax': x0,
                  'ay': y0,
```

```
                    'arrowhead': 3,
                    'arrowwidth': 1,
```

The code above creates the scatter plot object `edge_trace` (after importing `plotly.graph_objs` as `go`), to which we pass a dictionary describing the line type used for drawing edges. Then we loop through all edges in the graph and extract the coordinate pairs of the corresponding start and end node (index 0 and 1, respectively) by accessing the `'pos'` attribute we just assigned. These positions are added to the coordinate data inside the scatter plot object. Since we want to have arrows pointing from the referencing to the referenced table node, we create the list `annotations`. For each edge we append one entry to this list, which is a dictionary describing the arrow we want to draw.

Next, we deal with the nodes (the actual drawing of all the elements happens at the very end).

```
conn_ticks = [0, 1, 3, 10, 30, 100]
log_arg_ticks = [x + 1 for x in conn_ticks]
tick_vals = list(np.log10(log_arg_ticks))
cbar = {'thickness': 15,
        'title': 'Connections',
        'xanchor': 'left',
        'titleside': 'right',
        'tickvals': tick_vals,
        'ticktext': [str(x) for x in conn_ticks]}

marker = {'showscale': True,
          'colorscale': 'Inferno',
          'color': [],
          'size': [],
```

```
            'colorbar': cbar,
            'cmin': tick_vals[0],
            'cmax': tick_vals[-1]}

line = {'width': 2}

```

I decided to use color as an indicator for the number of connections N a node has. The code above creates a logarithmic color bar, which is helpful when there are many tables with few connections and few tables with many connections. The list `conn_ticks` contains the values of N at which we want to have labelled ticks. Since there are nodes with $N = 0$, we will use $\log(N + 1)$ to assign a color, and we choose the values at which the tick labels are placed, `tick_vals`, accordingly. The dictionary `marker` defines the appearance of the nodes, making use of the color bar we just set up and the built-in "Inferno" color scale of *plotly*.

As we did with the edges, we create a scatter plot object which must now be filled with data describing each node.

```
for node in graph.nodes:
    node_name = '%s/%s' % (graph.nodes[node]['db'],
                           graph.nodes[node]['table'])
    connections = (graph.nodes[node]['N_edges_upstream'] +
                   graph.nodes[node]['N_edges_downstream'])
    N_rows = graph.nodes[node]['N_rows']
    N_cols = len(graph.nodes[node]['cols'])
    prim_key = graph.nodes[node]['primary_key']
    f_keys = graph.nodes[node]['foreign_keys']
    refs = graph.nodes[node]['references']

```

```
        node_info = node_name
        node_info += '<br>'
        node_info += 'primary key: %s' % prim_key
        node_info += '<br>'
        if len(f_keys) > 0:
            node_info += 'foreign keys:<br>'
        for i in range(len(f_keys)):
            ref_str = '  %s -> %s<br>' % (f_keys[i], refs[i])
            node_info += ref_str
        node_info += 'rows: %i<br>' % N_rows
        node_info += 'columns: %i<br>' % N_cols
        node_info += 'connections: %i' % connections

        x, y = graph.nodes[node]['pos']
        node_trace['x'] += tuple([x])
        node_trace['y'] += tuple([y])
        if show_table_names:
            node_annot = {'showarrow': False,
                          'text': '/<br>'.join(node_name.split('/')),
                          'xref': 'x',
                          'yref': 'y',
                          'x': x,
                          'y': y + 0.03}
            annotations.append(node_annot)
```

The code shown above loops through all nodes in the graph and accesses the node properties of interest. These properties are compiled into the string `node_info`, which is set as mouse hover text in the very last line of the code. As we did with the edges, we extract the node position (the line `x, y = ...`) and add it to the coordinate data of the scatter plot object `node_trace`. If table names are to be displayed, the necessary data (dictionary `node_annot` containing text and position) are added to the annotations list which we used before for adding arrows. The node color is set in accordance with our

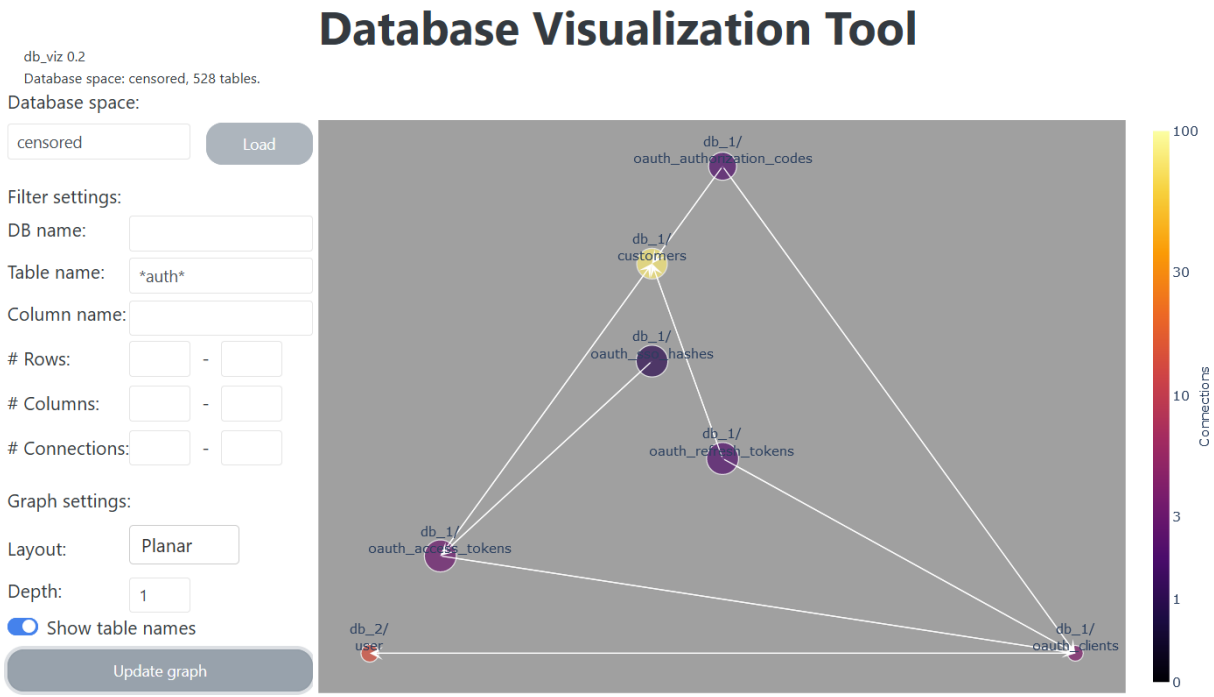
logarithmic color bar using the `node_trace['marker']['color']` property. The node size is set based on its number of rows (logarithmically with a minimum value).

The very last thing we need to do now is to build the actual figure object.

```
margin = {'b': 20, 'l': 5, 'r': 5, 't': 40}

axis_props = {'showgrid': False,
              'zeroline': False,
              'showticklabels': False}

c_axis_props = {'cauto': False}

layout = go.Layout(showlegend=False, hovermode='closest',
                   margin=margin, annotations=annotations,
                   xaxis=axis_props, yaxis=axis_props,
                   coloraxis=c_axis_props,
                   plot_bgcolor='#A0A0A0')

fig = go.Figure(data=[edge_trace, node_trace], layout=layout)

```

The code above sets the margin around the drawing area and defines properties for the x/y-axes (`axis_props`) as well as the color axis (`c_axis_props`; color range auto-scaling is turned off). The margin and axes dictionaries are passed to the layout object together with our annotations list (arrows and node names). Finally we create the `plotly.graph_obs.Figure` object which the callback function of the “Update graph” button returns to the `'figure'` property of the drawing area.

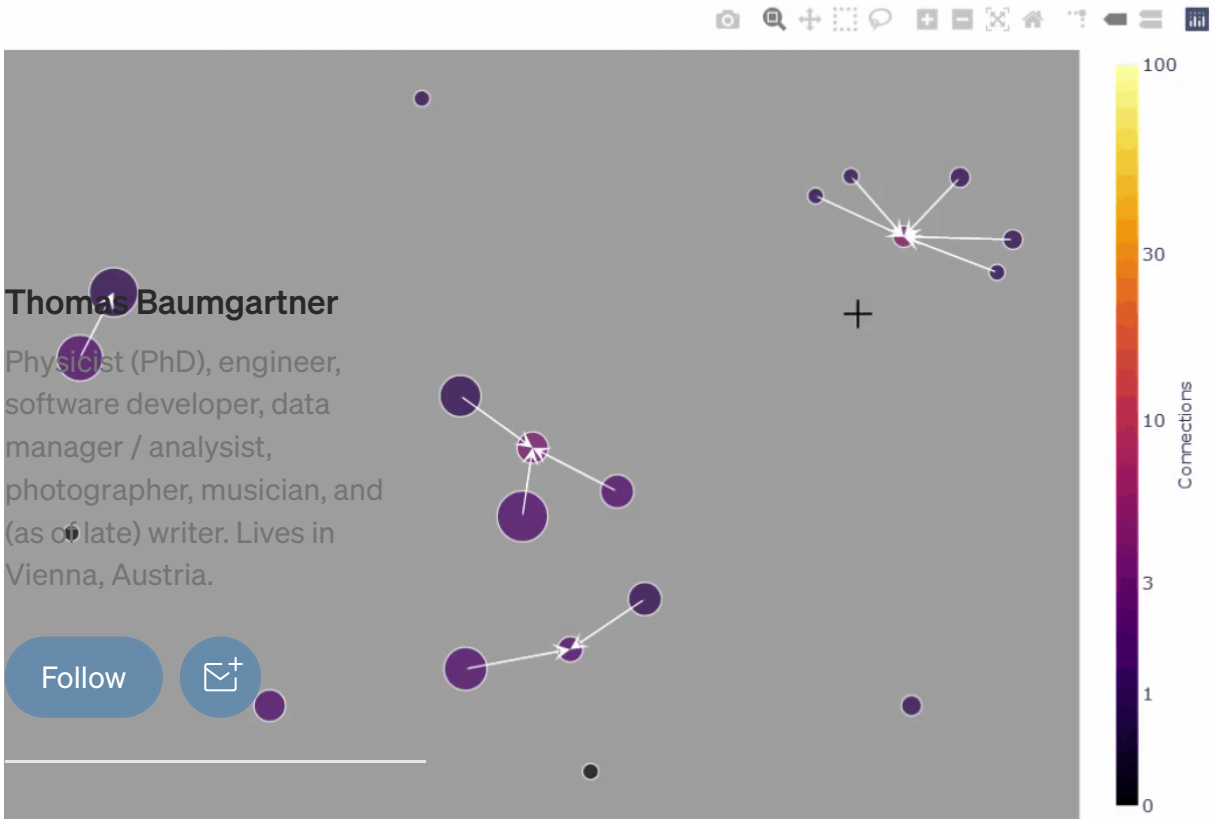
And here's the result of all our efforts:



GUI of the web application.

The picture above shows the web application after running a search for all tables with the string "auth" in their name and setting the depth to 1 (tables connected to the search result tables are included). Note that I changed some names here for data protection reasons.

The screen capture below demonstrates the mouse hover feature which allows to quickly check the most important properties of tables in the network graph (in the example the Kamada-Kawai layout was selected, which is often practical for identifying clusters). Also note the menu bar in the top right, which is a built-in feature of *plotly*, and allows zooming, panning, and saving the image as a PNG file.



THOMAS BAUMGARTNER
FOLLOWS

- Tim Denning
- Ayodeji Awosika
- Soner Yıldırım
- Barack Obama
- Stephen Muskett, M.S.Ed

See all (20)

The mouse hover functionality is a built-in feature of plotly.

61

Time to wrap up and give you a quick summary. In this article you learned how to:

- Obtain details on tables, columns, and references from the information schema of a database.
- Create a (directed) graph using the *networkx* package and represent the tables as nodes and their connections (references) as edges of the graph.
- Add useful attributes such as the foreign keys and number of rows of a table to the nodes.
- Apply filters by creating a copy of the graph object and removing all nodes whose attributes don't match the filter criteria.
- Use the layout functions of *networkx* to arrange the nodes of the graph.
- Draw the graph using *plotly*, with information encoded in the color and size of the nodes.
- Add a mouse hover text containing the most important node data.

Thanks to Anne Bonner.

Sign up for The Variable

By Towards Data Science

Every Thursday, the Variable delivers the very best of Towards Data Science: from hands-on tutorials and cutting-edge research to original features you don't want to miss. Take a look.

Get this newsletter

Database | Exploratory Data Analysis | Visualization | Python | Deep Dives

Medium

About Write Help Legal