NodeSQL

ECE464 Final Project Description

Gavri Kepets
gabriel.kepets@cooper.edu

Fall 2021

1 Abstract

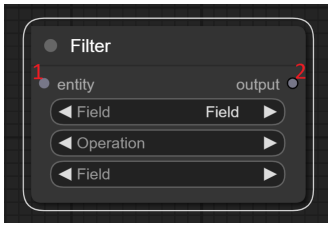
NodeSQL is a graphical interface that allows users to construct SQL queries without writing any SQL code. The interface is based on a node graph, which is a series of nodes that are connected to each other with links. Each node has input and outputs, and is responsible for performing some sort of computation on data, while links are responsible for transferring data from node to node. NodeSQL includes a collection of nodes that perform common computations in database querying, such as filtering and aggregation. This can be useful for non-technical users who don't know how to write SQL code and for users that would benefit from a visual representation of a query. SQL queries can get confusing and verbose, and NodeSQL aims to fix that by providing an intuitive graphical environment for building SQL queries.

2 Changes Since Initial Proposal

Since the initial proposal, a number of changes were made. The collection of available nodes was adjusted. Initially, there was a one-to-one relationship between SQL commands and nodes. In order to make the interface less messy and more intuitive, some nodes can represent multiple commands. Additionally, the nodes are now designed such that the nodes in the graph can be read from left to right (or top to bottom) like an English sentence. Finally, I decided to use a library for the front-end interface, as it would have taken a long time to build from scratch.

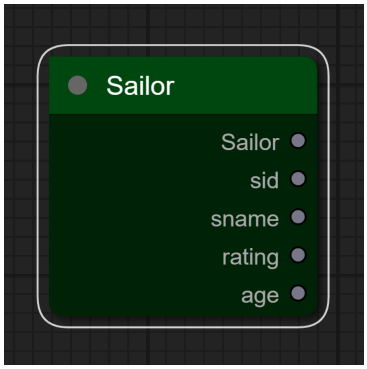
3 Functionality of NodeSQL

NodeSQL is an interface made up of nodes and links. There are three types of nodes- entities, which represent data types in a database, operations, which represent various SQL commands, and display, which is the endpoint for the data and where the user can execute the query. Nodes can have inputs and outputs, as shown below on the filter node below:



The input to the block is labeled 1, and the output is labeled 2. Nodes can have any amount of inputs or outputs, depending on what they do. Additionally, nodes have options in the center, which define how the node behaves. This will be further described below.

3.1 The Entity Node



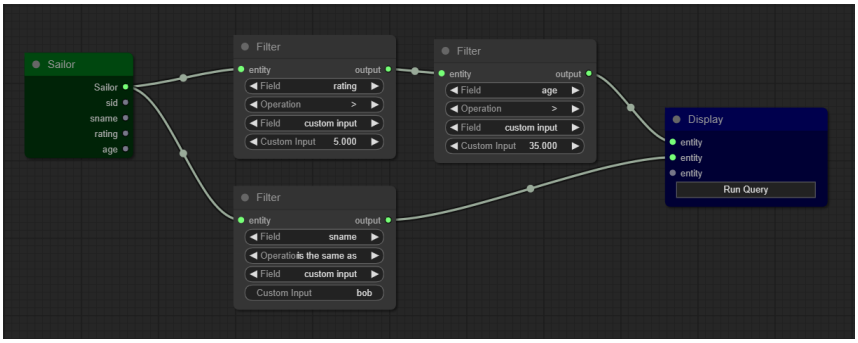
For this project, I have been working with an example dataset of Sailors. Each sailor has an ID, a name, an age and a rating. As of now, the project is hard-coded to deal with the Sailor database, but ideally should work for any datatype. The top output, labeled "Sailor", can be used to output all of the sailor data from the database into the operations. Below the top output are all of the attributes of that entity in the database.

3.2 The Operation Nodes

3.2.1 Filter

Arguably the most common operation node is the filter node. The filter node allows the user to filter out certain data. An entity is inserted into its input, and the node automatically updates its options based on the data type that was inserted into it. The filter node represents a comparison between a field and either another field or a number/string. The node has three dropdown menus on it; the first menu is the first term in the equation, which can be any one of the attributes from the entity. The second menu is the operator, which can be greater than, equal to, etc., and the last menu is what the first term is being compared to. For example, the first term may be 'age', the second may be '=', and the last one may be '30'; this filter only passes sailors who have an age equal to 30.

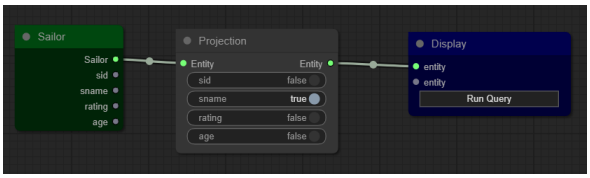
The last field can be a custom input (a number or a string, depending on the first term's type), an external node, which allows other nodes to be used (maybe you want to find sailors whose age is above average- this data would come from an external node), or another field in the entity (maybe you want to find sailors whose ID is greater than their age). Filters are simple on their own, but can be cascaded or split to create more complex queries, as shown below:



The above filter finds all of the sailors who either have a rating above five and an age above 35, or whose name is Bob.

3.2.2 Projection

The projection node passes every entity that it receives, but only certain attributes from those entities. For example, if a user only wants to pass the names of the users, they would only select the name attribute.



The projection above passes the names of all the sailors in the database.

3.2.3 Count

The count block is relatively straightforward- when data is passed into the count block, the user gets back the amount of objects that are in the dataset.

3.2.4 For Each

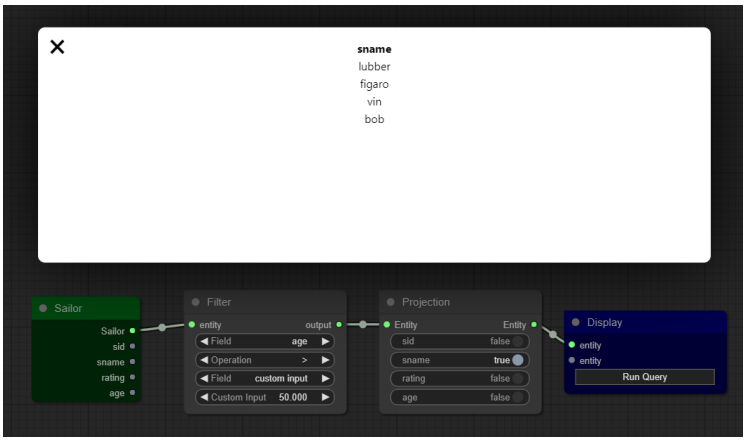
The ForEach block aims to replicate the 'group by' command and aggregation commands in SQL. It allows the user to feed data into the block and choose a field to group the data by as well as an aggregation command. In the example below, the user can uses the node to say: "For each" age, give me the average rating. As read on the node itself, output "average rating per age".

The resulting query uses the AVG aggregation function on the age attribute, and groups the data by the rating attribute.

3.3 The Display Node

The display block is the final destination for all data. Whatever is fed into the display block is accounted for in the final query. There can be multiple paths from the original entity to the display block, and the backend does its best to create one coherent query from all of them. The display block ideally only takes one datatype at a time- it would be impossible / confusing to get multiple different data types from a single query. The display block starts with one input, but can support as many as the user needs. Finally, on the display block is a

button that says “Run Query”. This button sends the graph information to the backend. A popup window is opened, and the data from the query appears on the window.



Above is a query that finds the names of all the sailors who are above the age of 50, with the results in the window.

4 Technology Stack

NodeSQL consists of a frontend, backend, and a database.

4.1 Frontend

The frontend is built in ReactJS and uses a library called litegraph.js for the node graph. I chose to use ReactJS because it would make implementing a display for the data much easier, and would also allow for the app to be more scalable as I add support for more features. Litegraph.js is a library based on HTML5 Canvas, which allows for the creation of nodes and links. I mainly used this library for the serialization feature that it offers, which takes the entire node graph and returns a JSON object with all of its relevant information.

4.2 Backend

The backend is built in Python and uses Flask as a REST API. The backend uses SQLAlchemy in order to interface with the database. SQLAlchemy was

the main reason I chose to do the backend in python, as I found it to be the most functional and simple ORM to work with (SQLAlchemy felt like it was miles ahead sequelize.js and gorm).

4.3 Database

The database is a standard MySQL database with a table of Sailors in it.

5 Process

The process of using NodeSQL goes as follows:

1. The user uses the front end to create a query
2. The user clicks the run query button, the display window is opened, and the serialized graph is sent to the backend API
3. The backend receives a list of nodes and links, and then finds every path from the entity to the display
4. Once every path is found, the backend runs through each node and constructs the query
5. A query is finalized and executed, and the data is returned to the front end as a JSON object of keys and data.
6. The frontend displays the data on the window

6 Bugs

Unfortunately, there are a number of known bugs/issues with NodeSQL. The backend is not very stable for some reason, and often needs to be restarted after multiple queries. I believe this is an issue with SQLAlchemy and the way they construct their query objects. Another major issue is error handling- there is minimal error handling, and if something goes horribly wrong, the user is not really informed about it. Finally, there are a few UI issues in the frontend, such as the ability to continuously add more custom inputs to the filter node.

7 Future Features

Some of the future features I would like to implement if I had more time:

- Support for multiple different entities in one query, table joins, and temporary tables.
- Restrictions on nodes that only allows certain data types to be linked- as of now, query verification is done on the backend, but the user should be able to see if their graph is invalid before executing the query (or maybe it should be impossible to make an invalid graph in the first place).
- Custom blocks that represent subqueries; users can create complex subqueries and then package them into one block with their own description, which would allow for larger queries to be broken down into smaller chunks.
- The ability to view the data in each link; the user can hover over a link and see some of the data coming out of a node. This would be incredibly useful for debugging queries and understanding exactly what each node is doing.
- Finally, the ability to be able to do anything in NodeSQL that one can do with SQL

8 Concluding Thoughts

This project was a major learning experience. I learned how to use multiple different ORMs, how MySQL databases work, and how SQL works. The most interesting part of this project was figuring out how to make SQL more intuitive for someone who has never seen it before- I felt that because I already knew how SQL worked, I needed to think backwards in order to make NodeSQL feel more natural and less like a programming language. I had an excellent time experimenting with different structures and nodes to make this work, as well as different ORMs and databases while building this project.

There are, however, a number of things I would do differently if I started this project from scratch or had infinite time to work on it. Firstly, I would build my own UI from scratch. I opted to use a prebuilt one because the course is about how databases work and not how HTML5 canvas works, but there were definitely roadblocks while using someone else's UI framework (not to say it is bad, it is actually a great library, it just isn't necessarily built to do what I want it to). I also learned how frustrating ORMs can be. Out of all the ORMs I tested, SQLAlchemy was the easiest to work with and the most functional,

and there were still major issues while designing the backend. I sometimes felt as though I was designing my entire backend to work around SQLAlchemy. If I were to start from scratch, as painful as it might be, I would write my own ORM that is specifically tailored to work with node graphs. That way I won't be limited by SQLAlchemy's custom data structures and functions. Finally, I would ask more people for advice earlier on. When I started, I only spoke to a handful of people about how the nodes should be structured and connected. Further along into the project, I asked way more friends and family who are not familiar with SQL about what makes sense, what is intuitive, and what feels natural to them, and that was incredibly helpful while designing the different blocks and options. I went through multiple iterations before people felt that it was easy to use.