StarfishQL

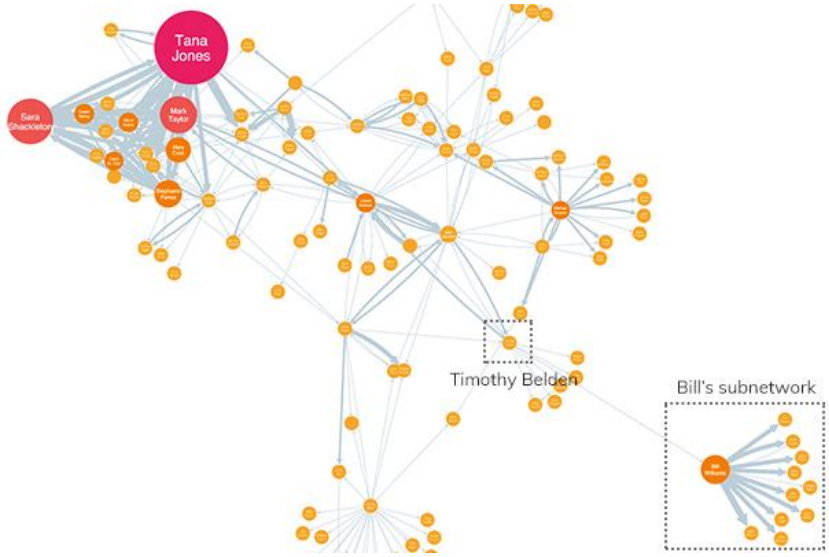
A graph database and query engine

Objective

To create an interactive of visualization of Rust crates and their dependencies

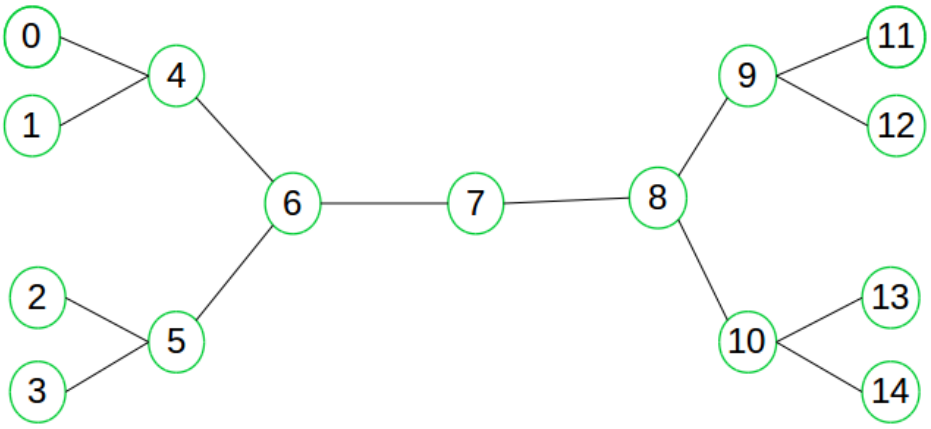
1. connected graph visualizations

Ref: [Cambridge Intelligence - Connected data and timeline visualization (cambridge-intelligence.com)]



Starting from the N most depended root crates and go down M layers and render an interactive graph

2. bidirectional tree



Say 7 is the crate we want to query, all nodes on the left are dependencies while all nodes on the right are dependents

Written by Chris, Dec 2021

Concepts

1. Entity
   It could be homogeneous (like in the crates.io example, all are crates) or heterogeneous (say we distinguish between library crates and application crates)
2. Nodes
   An instance of the Entity in the graph
3. Relation
   Relation between entities. It can have a direction or being equivalent on both directions
4. Edge
   A connection between two nodes
5. Graph
   A data structure where a number of nodes are connected by a number of edges
6. Attribute
   Quantity (scaler values) and qualities (enum labels) we attach onto each node or edge
7. Query
   A description of the data we want to extract from the database
   The query result can be graph, vector or scaler
8. Constraint
   When performing a query, the requirements of the graph (say we only limit graph to 3 levels deep). Or we apply certain criteria on the topology (say nodes having 3 or more edges).
9. Criteria
   When performing a query, the filters being applied to nodes and edges (we only want them to have certain attributes)

Deliverables

1. A graph query engine written in Rust
2. A web backend written in Typescript
3. A web frontend written in Javascript

At the end of the project, we should have a working website visualizing the live data from crates.io

The web backend should be able to continually ingest update from crates.io

Hint: crates.io is actually a git repository, where we can follow the git history

Query Language

The engine should have a REST API: several HTTP endpoints, receiving query in JSON and returning result in JSON.

/schema

Define the database schema, e.g. { define: { entity: { name: “crate”, attributes: [{ name: “name” }] } } }, { define: { relation: { name: “depends”, between: { entity: “crate” }, directed: true } } }

/mutate

Insert, update or delete nodes and edges, e.g. { insert: { node: { of: “crate” }, attributes: [{name: “my_crate”}] } }

/query

e.g. To query the 10 most depended crates

{ query: { vector: { constraints: [{ sortBy: { connectivity: { complex: { of: “depends” } } } }, { limit: { range: { top: 10 } } }] }

e.g. To query the graph starting from the 10 most popular crates

{ query: { graph: { constraints: [{ rootNodes: [...] }, { edge: { of: “depends” } }, { limit: { depth: { to: 3 } } }] } } }

e.g. To query a crate id=888 with all its dependents (reverse of dependency)

{ query: { graph: { constraints: [{ rootNode: { id: 888 } }, { edge: { of: “depends”, traversal: { direction: { reverse: true } } } }] } }

Mode of operation of the query engine

Input JSON -> deserialize -> transform into syntax tree -> build a query plan (multiple SQL queries in form of a tree (i.e. leaf = input, node = SQL query, edge = dependency, root = output)) -> execute query -> serialize result into JSON

Architecture

Obviously, we build on top of SeaQuery, SeaSchema and SeaORM.

We have to first define the static schema of our internal database for storing metadata. These can be used with SeaORM.

We also define an external database where its schema is dynamic, where each Entity & Relation is a table. When we define a new entity, we create a new table entity_xxx via SeaSchema.

Then we use SeaQuery to build the queries runtime.

This project should push the possibilities of SeaQuery, SeaSchema and SeaORM to their limits.

There is also an intermediate database for indexing (e.g. we would like to compute and store the connectivity of each node).

> We have to define connectivity here. Simple connectivity is **simply the number of edges a node has**. If the Relation has a direction, then there should be two numbers, one for incoming and one for outgoing.
> Compound connectivity is, well, compound. The basic idea is, every time we insert a crate to the database, we walk through its dependencies and add 1 to every node’s compound connectivity. It works because cargo dependencies are acyclic. It is more meaningful than simple connectivity because we are not only seeing the direct dependency, but **see through the indirect dependency**. This is interesting because there are lots of people using SeaORM, but they may or may not realize SQLx is under the hood.
> (Anyway, SeaORM will be doomed, because users of SeaORM are apps but apps are not on crates.io. We will have to crawl that data from GitHub)
>
> I am not sure compound connectivity is actually what we want though. If all the so called ‘most depended’ crates turn out to be ‘rand’, ‘chrono’ etc, it is quite boring! They are the most ‘fundamental’ crates for sure, but not necessarily the most ‘interesting’.
> So, to mitigate this, we could devise a final concept, complex connectivity. Instead of adding 1 to every node on the dependency tree, we use an exponential falloff along the depth. Meaning we add 1 to the 1st level dependencies, 0.5 to 2nd level and 0.25 to 3rd level, etc.

Division of Labour

Billy should start from the database side, making /schema and /mutate workable.

Sanford should start from the visualization side (using a static dummy dataset) to render some graphs on frontend (ref: D3.js - Data-Driven Documents (d3js.org), JSON | Graphviz).

Then both should work together on the query engine.

The initial query engine is a toy, it only has to support the limited number of (i.e. 3) queries we need to achieve our objectives. We only ever have one Entity (crate) and Relation (depends) respectively.

But the architecture should be sound and code should not be messy.

Finally, we can ingest data from crates.io to put it to real life. And may be make a pretty UI. These things are nice-to-haves and we can do them later perhaps after the winter period since time is limited.

References

1. Force-Directed Edge Bundling (FDEB) on Force-Directed Graph
   https://bl.ocks.org/vasturiano/7c5f24ef7d4237f7eb33f17e59a6976e
2. How do I group the nodes in a force directed graph
   https://stackoverflow.com/questions/33560783/how-do-i-group-the-nodes-in-a-force-directed-graph?rq=1
   https://bl.ocks.org/mbostock/4062045
3. d3.js: force layout; click to group/bundle nodes
   http://bl.ocks.org/GerHobbelt/3071239

Written by Chris, Dec 2021