Managing Complexity of Radio Interferometry Observatories Using Graph Technologies

Anatoly Zavyalov, working with Adam Hincks

University of Toronto

August 11, 2021

The Need

Modern radio interferometry observatories consist of thousands of components, so it is necessary to have a system to track, add, update and perform queries on all of the components that make up the experiment.

What is HIRAX?

The **Hydrogen Intensity and Real-time Analysis eXperiment (HIRAX)** is an upcoming observatory consisting of a **32x32 array** of **radio interferometric telescopes**, and will be constructed in the Karoo region of South Africa. Inspired by CHIME, it will be used to study the distribution and evolution of dark matter and dark energy throughout the universe by observing the 21-cm wavelength line of hydrogen.

How the final HIRAX array may look like in Karoo. Credit: HIRAX Collaboration

Common Queries

Some common queries that should be answered using the system are:

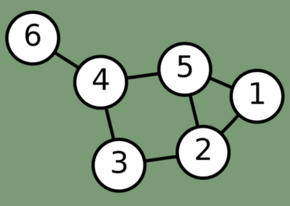
- What input does this antenna connect to?
- What were the position and tilt of this antenna three days ago?
- What components did the signal pass through on its way to the input?
- Were these components connected to each other at this time?

Planning

We knew that users of the system would interact with it either through a Python API by creating scripts, or through a web interface, which would use the Python API to extract information from a database.

The Python API would provide access to the database, and allow the user to access and update the properties of all the components, view connections between components, and more. Most "bulk" operations would be done through the Python API, such as adding or changing many components at once.

The web interface would provide a more readable format for the information, as well as a graphical interface to interact with the data.



A simple graph of 6 vertices labelled with numbers. Credit: Wikipedia

Graphs

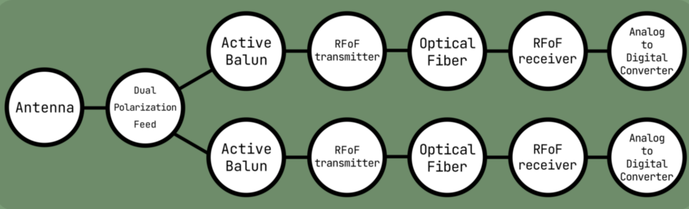
The observatory site could be well represented by a graph - a structure consisting of vertices that can be connected to other vertices by edges. The components would be represented by vertices, and the connections between the components would be represented as edges between these vertices. From the start, we were interested in graph databases that could natively store the data in a graph structure.

Reconstructing the Graph

A major requirement of the HIRAX layout database is to be able to determine the graph's structure at any point in time and be able to reconstruct how the graph looked like given a timestamp. For example, we wish to know, at a specific date and time, whether two vertices of the graph were connected and what the properties of the elements represented by those vertices were.

Graphs in HIRAX

The HIRAX observatory will contain identical signal chains for each of the 1024 antennas, which can be represented with a graph structure by considering each component as a vertex, with connections between components represented by edges. This is shown on the right:



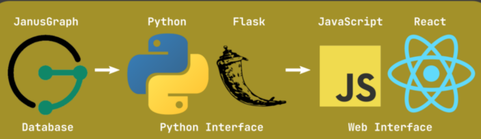
Credit: Anatoly Zavyalov

Graph Databases

If the observatory site can be best modeled with graphs, it makes sense to store the data as a graph as well. Unlike conventional relational databases that store data in tables, graph databases store data in a graph structure. We settled on JanusGraph, a free and open-source graph database actively supported by its developers with a large, active community. JanusGraph also allows to connect to it using a Python interface called Gremlin-Python, making it perfect for our needs.

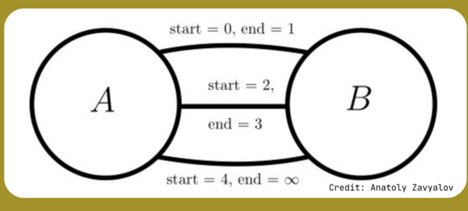
Tools for Web and Connectivity

Since we wanted the web interface to directly connect to the Python API (hence reducing the amount of code we would have to rewrite), we used the Flask web framework, which would query the Python API and allow us to fetch it from the web interface. For the web interface, we use the React library, which contains queries to the Flask framework that interacts with the Python API.



Reconstructing the Graph pt.2

To determine whether two vertices (representing components) were connected or disconnected at some time, we can store the connections as separate edges between the vertices:



Credit: Anatoly Zavyalov

In the above image, two vertices A and B are connected at time 0, disconnected at time 1, connected at time 2, disconnected at time 3, and connected from time 4 to beyond.

With this approach, to determine whether two components were connected at some time **t**, we simply iterate over all edges between the components and see if there exists an edge such that **t** falls in between its **start** and **end** times.

The Process

1. Get acquainted with JanusGraph, determine whether all necessary functionality is achievable with it.
2. Benchmark and optimize JanusGraph: How fast are common operations, and can we optimize the queries to make them even faster?
3. Create a Python API to interact with the JanusGraph database, include the minimum required functionality.
4. Create a web interface to use the Python API to graphically interact with the data.

Python API

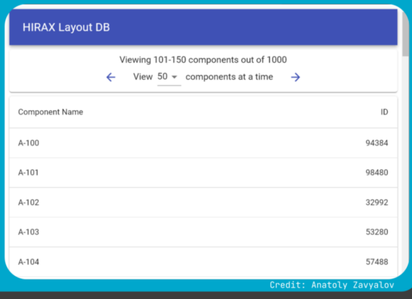
```
>>> c = Component.from_db("ANT0001")
>>> c.component_type.name
'ANT'
>>> c.component_type.comments
'This is the antenna type'
>>> c.revision.name
'A'
>>> c.revision.comments
'This is the first revision of the antenna'
```

So far, we have developed a Python API that allows for simple retrieval and creation of components and their properties.

This section of example code extracts a component from the database and allows you to view its attributes, such as its type and revision.

Web Interface

An early version of the web interface shows a list of components, where one views all components in the database.
Future functionality will include clickable components where one views more details, visualizations of subgraphs, a graphical interface for adding components, changing properties, and more.



Credit: Anatoly Zavyalov

Limitations and Uncertainties

In the JanusGraph experimentation stages, we were looking for ways to easily extract **subgraphs** from the database. JanusGraph had a built-in subgraph extraction method, but it was not supported in the Python implementation, as it was built to be "lightweight".
We also encountered a delay when first querying the database lasting upwards of 10 seconds, which has yet to be fully addressed.
Lastly, due to the quite large software stack, working with the system will require some learning for future contributors to the system.

Did we meet the need?

So far, we created a working Python API as well as a barebones web interface. All the building blocks are in place, but there still remains more work to be done in order to achieve the full functionality of the system.

Next Steps

For the Python API, the next steps to improve and better the functionality of the API are to implement more functions that use the existing foundations that we have built. For example, to find a shortest path between two components in the graph, a single database query is to be integrated into the existing API. To increase the functionality of the web interface, all that is needed to use the existing Python API to access the necessary information and format it.

References

"Flask", https://flask.palletsprojects.com/en/2.0.x/

Hincks, A. D., and Shaw, J. R., "Managing Hardware Configurations and Data Products for the Canadian Hydrogen Intensity Mapping Experiment," arXiv 1410.8418 (Sept. 2015)

"JanusGraph", https://janusgraph.org/

Newburgh, L. et al., "HIRAX: a probe of dark energy and radio transients," arXiv 1607.02059 (Aug. 2016)

"React", https://reactjs.org/