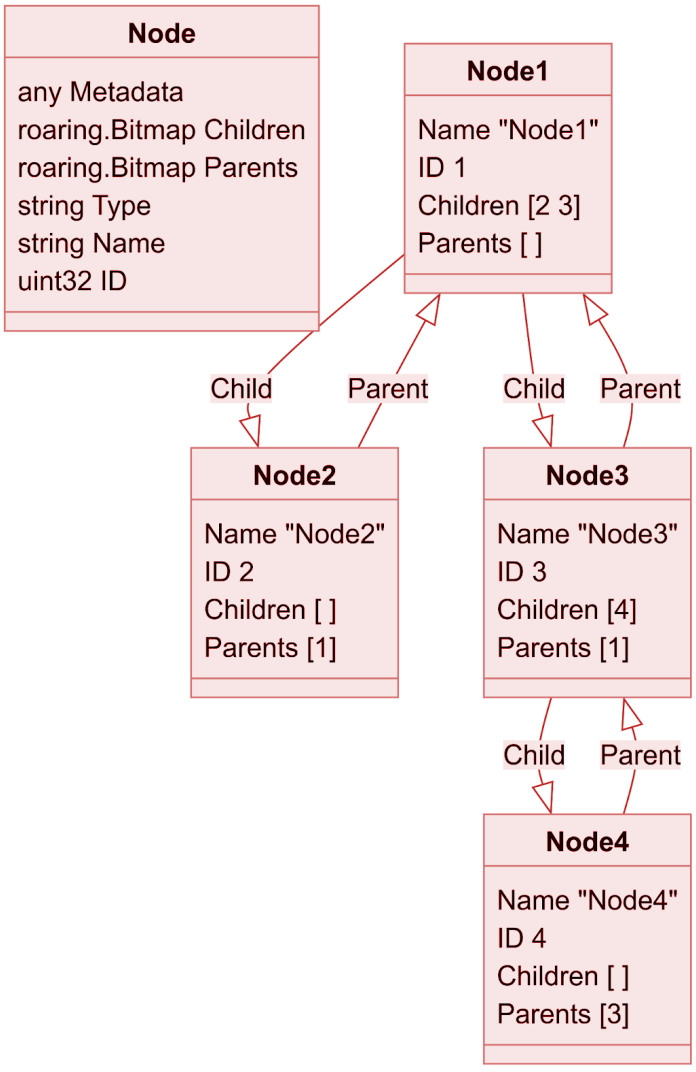
bitbomdev/minefield

https://github.com/bitbomdev/minefield

Minefield is a lightweight graph database for dependency-based metadata. It is currently focused on graphing SBOMs and other software supply chain artifacts. It has been designed to handle massive amounts of data from the ground up, offering O(1) query times by storing relationship data in Roaring Bitmaps.

Structure



Almost all of Minefield's data exists within a specific structure. This structure deviates slightly from the traditional node-edge-node approach in graphs, instead adopting a direct node-to-node model. While this results in some loss of information traditionally contained in the edge, all necessary data can be stored in either the start or end node, and the edge type can be inferred based on the types of the start and end nodes.

The advantage of this approach is that all relationship data can be compressed into Roaring Bitmaps. Additionally, adding any custom node does not require a new unique edge for integration into the graph, which is one of the features Minefield supports.

Roaring Bitmaps

Roaring bitmaps are a data structure that creates enormous bitmaps.

A regular bitmap (bitmask) is a highly space-efficient Set (Or in languages that don't use Sets, a `map[int]bool`). A Roaring Bitmap is a regular bitmap on steroids.

For instance, 32-bit integer keys mean it can theoretically store up to 2^32 different values, i.e., around 4.29 billion (Or exactly 4294967296 integers) values in a minimal amount of space (This is stated in the Roaring Bitmap specification).

Bitwise Queries

All of Minefield's relationship data is stored in Roaring Bitmaps, which is very compressed. Another advantage of storing data in roaring bitmaps is that we can perform bitwise operations between two roaring bitmaps with minimal, almost negligible overhead.

When we query graphs, especially SBOM graphs, many of the queries we want to perform are niche, never-to-be-repeated queries, which, while useless in most situations, can be very useful in a few-to-one.

In Minefield, we can execute these operations since our queries can be stringed together with bitwise operations

For example, we can check if a package in our graph has circular dependencies by running this query, which, with caching, is instantaneous. (The next section is on how our queries run in O(1) time)

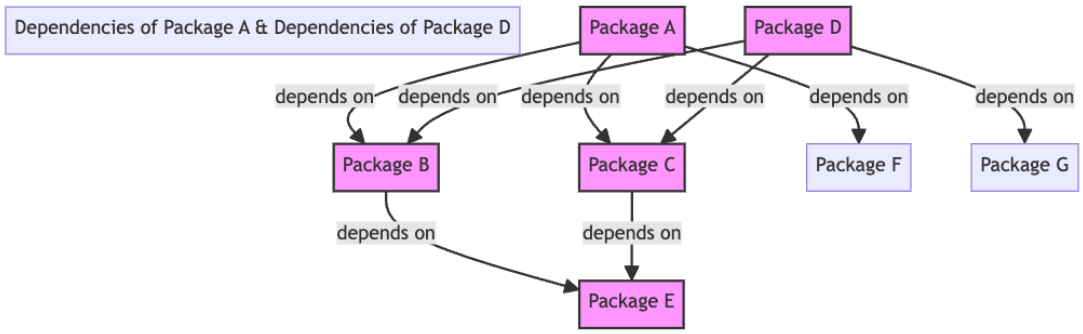
`minefield query "dependents library package1 and dependencies library package1"

This gets the dependents of package 1 that are of type library and the dependencies of package 1 that are of type library and then runs a bitwise and between the two results.

If we get an output, that means some nodes are both dependencies and dependents of package 1. Since this query also queries transitive dependencies, it not only outputs packages that are the queried nodes' direct circular dependencies but also the dependencies of the nodes in the circle.

A circular dependency is a simple query, which is quite helpful and may be used often, but there are many more that you can create.

Here is another example: We can output common packages amongst two different package's dependencies.



Caching

One of Minefield's main features is its speed. Ingestion runs with no subprocesses, achieving O(n) time, meaning there is no extra overhead when ingesting data. Regular querying without caching operates in O(number of queried nodes * number of edges), the fastest achievable without pre-computing data. Minefield can perform queries in O(1) time using the cache, which works with circular dependencies.

There are two primary challenges with caches: they are memory-intensive, which may not be desirable for many users, and they require time. Most caches take an exponential amount of time to cache the entire graph, which becomes problematic as the number of nodes and connections increases.

For example, caching 100,000 nodes, each with 10 edges, using traditional methods would take O(n^2 * m) time, where n is the number of nodes and m is the number of edges.

This would take approximately 16.67 minutes if each operation took 1 nanosecond. While this duration is manageable when caching once or twice daily when dealing with large data chunks, it becomes infeasible as the scale increases. Caching 10 million nodes each with 100 connections would involve 10^18 operations, taking 31.7 years if each operation took 1 nanosecond. This duration is clearly unacceptable.

Minefield addresses both the space and time issues in caching.

Roaring Bitmaps, which are very lightweight data structures, effectively utilize storage space. These bitmaps store the complete cache, as they can efficiently represent each dependency and dependent in a single Roaring Bitmap.

Minefield effectively handles time by utilizing a pre-compute cache that stores every potential query in the graph. This process takes O(n * m) time, where n represents the

number of nodes and m represents the number of connections. Once a node has been processed and stored in the cache, it won't be revisited.

This is possible by breaking down the caching problem like a Dynamic Programming problem, where recursive sub-problems are solved and combined to address the next query without redoing work. In this approach, each node's dependents and dependencies are calculated by summing its own dependencies and dependents along with those of its dependencies and dependents.

While this might sound complex, it makes the Minefield cache incredibly efficient.

Returning to our earlier example, caching 100,000 nodes with 10 connections each, with each operation taking 1 nanosecond, would take around 16 minutes using a traditional cache. With Minefield's cache, this would take just 1 millisecond. For 10 million nodes, each with 100 connections, a traditional cache would take 31.7 years, whereas Minefield's cache would take just 1 second.

The difference of 1 second versus 31.7 years is monumental.

Extensions

The Minefield system consists of two main components: the graph and the tools built around it. The graph is purely a dependency-based structure and does not possess knowledge of SBOMs (Software Bill of Materials) or any other metadata. It is agnostic and independent. On the other hand, the tools interact with the graph and enable actions such as ingesting SBOMs, OSV (Open Source Vulnerabilities) data, and other metadata. They achieve this by employing a set of fundamental functions to create and retrieve information from the graph.

The tools used in the project can be replicated outside the project as they interact with the graph through exported functions. This capability empowers end users to create personalized query and ingestion functions without making changes to the upstream repository.

This means any private type of metadata can be ingested into the graph without having to fork and change Minefield itself. Creating these functions is also not a complicated endeavor.

- All you need to do is create "nodes" in Minefield's graph with the AddNode function.

- Use the SetDependency function to link nodes to other nodes.

- Minefield is then ready to query that data.

AddNode and SetDependency are API's. This will become a gRPC/REST endpoint.

Minefield offers the flexibility to ingest and query custom data.

Additionally, it allows users to create custom storage options. While Minefield defaults to Redis for storing nodes, relationships, and cache data, users have the option to choose an alternative storage solution. Whether it's a non-Redis option or a custom storage type that Minefield doesn't currently support, users can implement it by ensuring it adheres to Minefield's Storage interface.