Entity Relations in Bevy

Functional Spec

Yogesvara Das	19712101
David Mc Guigan	19425942

Supervisor: David Sinclair

Last edited: 10/11/2022

Table of contents

1. Introduction

1.1 Overview

Entity relations will be an addition to the Bevy open source game engine. Bevy consists of multiple crates. Entity relations will be a feature added to the ECS crate “bevy_ecs“. Entity relations will enable library level relational programming in a manner very similar to prolog where the atoms are Entities. Our take on relations has 2 notable differences to prolog’s predicates. Relations can have values and relations enforce graph connectivity properties.

There is a strong consensus among bevy maintainers, contributors and users that this is a needed feature that would complement Bevy very well. Current alternative patterns to relations are not ergonomic. They create spaghetti code and do not allow expressing connectivity guarantees. This project consists of the two pieces of work most desired by Bevy:

- Entity relation storage and retrieval mechanisms behind an ergonomic library level API that makes the magic happen in a performant fashion.

- Relational connectivity policies that allow for expressing properties that alter graph behaviour for adding relations between atoms, removing relations between atoms, removing atoms that have relations. Eg. exclusive, reparenting, recursive removal, orphaning, acyclic.

1.2 Business Context

Not applicable. This is a contribution to an open source project. We do not have commercial rights over bevy and private commercialization efforts of a contribution would be bizarre and predatory practice.

Our work will remain free and open source licensed under the same licences bevy uses. At time of writing this is a dual licence consisting of MIT and Apache 2.0.

1.3 Glossary

- **"ECS"** Entity Component System. A data oriented style of game engine architecture.
- **"Entity"** A unique identifier typically represented by an unsigned integer that owns zero or more components.
- **"Component"** Data types that store data for entities.
- **"System"** A procedure that queries for components and other world items and then performs computation with those items.
- **"Archetype"** Collections of entities that have the same set of components.
- **"Fragmentation"** The splitting up of data corresponding to similar archetypes. For example the archetypes themselves or the tables used for component storage.
  .
- **"Relation"** Relational primitive that sugars indirection semantics.
- **"Target"** The participant in a relation whose entity archetype is unaware/not responsible for maintaining the relation.
- **"Atom"** Relational primitive that refers to the participants of relations.
- **"Command"** Deferred execution mechanism in bevy.
- **"Policy"** Procedures to execute that uphold a desired property of a relation graph.

2. General Description

2.1 Product / System Functions

Entity relations will allow developers using Bevy to express indirection at an ECS level with relational semantics as opposed to the current pattern of manually storing an Entity ID in a component and manually checking if entities still exist when querying.

Developers will be able to declaratively enforce graph properties for different types of relations with connectivity policies allowing for greater ease of use across vast numbers of scenarios. For example hierarchical relations that need to be trees can reparent entities when entities in the middle of the tree are removed.

2.2 User Characteristics and Objectives

Our users are the users of Bevy which are game developers. As they are developers they will have lots of experience with software so it is ok to expect that they are willing to read a reasonable amount of dense technical documentation.

Users are not expected to have experience with relational programming languages like prolog. Since this is a library level DSL we have the opportunity to not inherit relational notation which many programmers often find confusing. We will instead just have ordinary familiar functions and methods.

However as they are developers their needs are quite demanding and there are a number of quite involved identified objectives that must be fulfilled.

- `Ergonomic` - Ergonomics is one of Bevy's core values. At the very least the ergonomics of relations should be better than the current "Entity in Component" pattern. However It's desired to be much better than that. Ideally leveraging Rust's sophisticated type system and resulting in an API that incorporates itself into the type level DSL that Bevy has created.

- `Performant` - Ideally we would like dense iteration as well as fast lookup for all data touched but a more realistic initial performance goal for relations is that it should not seriously degrade the performance of any unrelated code as past attempts at relations have.

- `Correct` - Relations between entities form a graph and a large number of use cases require that some property be maintained in the graph. Connectivity policies must support a number of common use case properties.

- `Small` - This work will be done on a fork of the Bevy repo which is intended to be upstreamed to the main Bevy repo eventually outside of college time after the module has concluded. A smaller feature set is easier to review and merge. Additionally features such as conditional relations multiply the scale of the scope by a large factor. It would not be reasonable to try to focus on more than core relational features in the timeframe we have.

2.3 Operational Scenarios

Relations can be used anywhere one would normally use indirection. Storing references or pointers to other data that exists can all be replaced with relations. In general this removes spaghetti code and lets you give code more meaning as you have an additional opportunity to name something through the relation.

2.3.1 Identified usage scenarios

- `Bevy's transform hierarchy` - A ChildOf and ParentOf mechanism already exists in Bevy. It's a more secure version of the entity in component pattern but is far from relations. Bevy uses this to construct a hierarchy in the ECS that propagates transform information like scale and rotation through child entities for global and local transformations.

- `System chaining/piping` - Currently the API to schedule systems is more verbose than it should be. This is all due to limitations in the engine internals that rely on labels and before/after conditions. With relations as system graphs can be constructed instead for more ergonomic system scheduling.

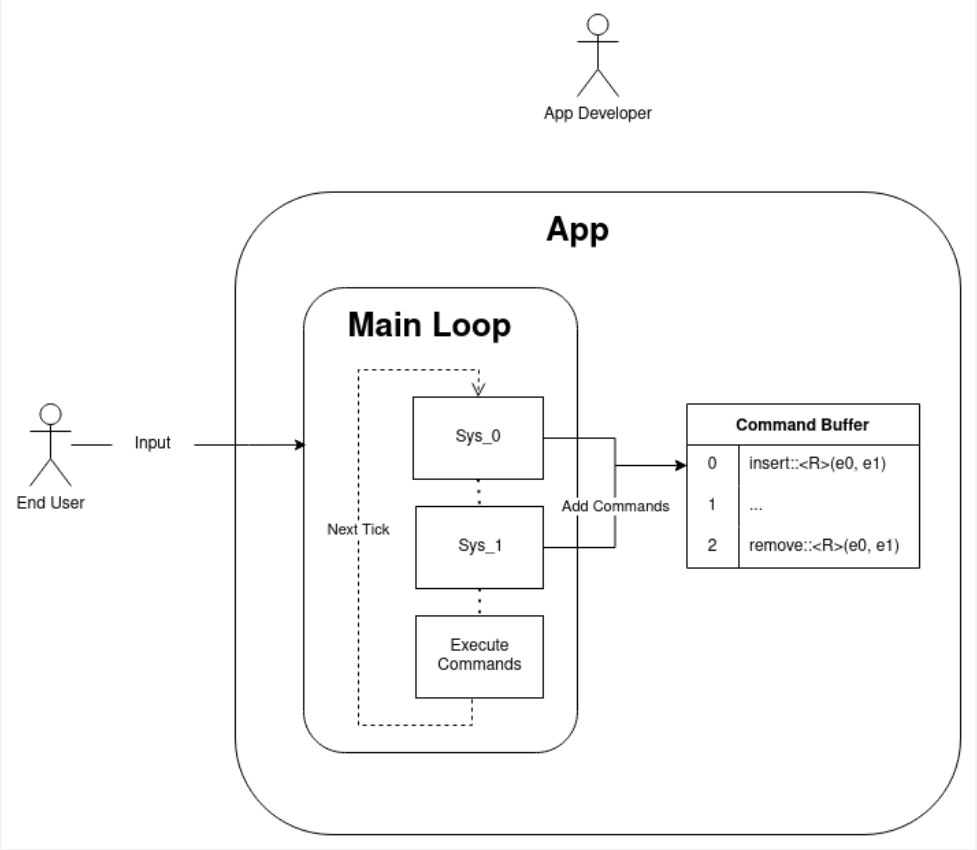
- `Pathfinding` - Terrain components can be linked with relations to create paths for systems responsible for adversary navigation to find paths to the player or other things in the world.

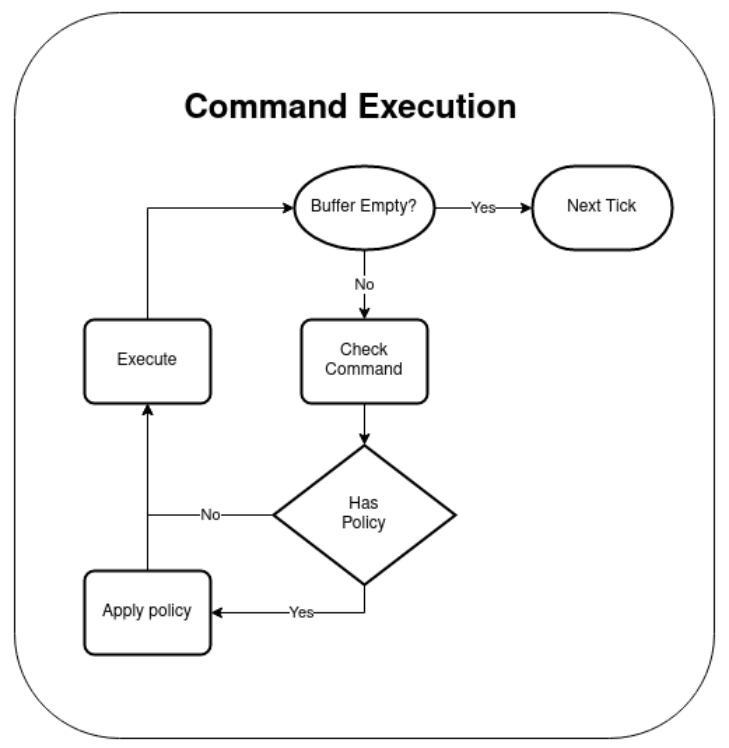
- `Spring and joints for colliders` - Relations can model springs for soft bodies and joints for inverse kinematics. Doing physics this way with looser data coupling allows for physics engines to be written with less edge cases because things like type specific field access and the methods to address those problems like bloated amounts of polymorphism are eliminated. Everything incorporates better into the ECS and the required information can be requested in queries.

- `GUI` - Graphical user interface framework APIs have had an indirection problem since their inception. You don't want to redundantly recreate your entire app state every update. However you can't use indirection everywhere because this creates spaghetti code. So we made callbacks which then created callback hell. The whole problem here is that the GUI has its own state that alters the state of something else that itself is a state. Relations with connectivity policies can remove some statekeeping GUIs do because they allow for self cleaning indirection.

2.3.2 Policy use case diagram

Policies are automatically applied. How depends on the policy. In general this is what happens when we want a certain property upheld.





2.4 Constraints

Since the Rust ecosystem is very cross platform and we are in somewhat uncharted waters with this project we do not have any precisely measurable constraints. We have a lot of freedom in how we approach our objectives. That said, we have one major design limitation.

2.4.1 Policy execution

ECS is a highly parallel architecture and as such needs a deferred execution mechanism for threads to not block each other. In bevy these are referred to as “commands”.

Certain properties require fallibility which commands are incapable of. Further more simple error handling for each command is not enough as commands can result in intermediate states that might not be valid but a final state that is valid.

- This would be impossible for us to reason about as commands are received from multiple sources in parallel.
- It would be impossible for us to give useful information to developers to recover from errors.

The required solution to this would be transactional commands which would allow for reasoning about:

- What to revert.
- What is allowed to have invalid intermediate states.
- How to prioritise transactions.

Sadly this is not easy to shoehorn into bevy as commands are type erased. To add transactional commands would be a massive undertaking as they’re highly integrated into everything bevy does. Transactional commands would have enough scope to be their own fourth year project. As such we cannot pursue more advanced policies like:

- Acyclic policies.
- Component requirement policies for relation participants.

3. Functional Requirements

3.1 Entity Relations

- ### 3.1.1 Relation Storage

  `Description` - Before we start adding, querying or removing relations we need to figure out how to store them in a manner that will allow for the API we desire.

  `Criticality` - This is highly critical as everything relies on this to work.

  `Technical Issues` - Performance. One of the reasons Entity Component System architectures are desired is because they are very performant. The reason for this is they maximise CPU cache hits. The most desired operations are lookup and iteration. Memory locality and density is everything for iteration but trying to incorporate fast lookup naively incurs cache misses without clever data structure usage.

- ### 3.1.2 Relation Creation

  `Description` - Creating new relations and applying the relevant connectivity policy rules.

  `Criticality` - This is a highly critical piece of core functionality.

  `Dependencies` - Relation Storage.

- ### 3.1.3 Relation Querying

  `Description` - Creating a query API for relations.

  `Criticality` - This is a highly critical piece of core functionality.

  `Dependencies` - Relation Storage, Relation Creation

- ### 3.1.4 Relation Removal

  `Description` - Applying relevant connectivity policy rules when a relation gets removed or an entity with relations gets despawned.

  `Criticality` - This is a highly critical piece of core functionality.

  `Dependencies` - Relation Storage, Relation Creation

3.2 Relational Connectivity Policies

- ### 3.2.1 Orphaning

  `Description` - The default behaviour for relations, orphaned subgraphs will remain intact after the removal of a parent node.

  `Criticality` - This is a highly critical piece of core functionality

  `Dependencies` - Relation Removal, Relation Creation, Relation Querying

---

- ### 3.2.2 Exclusive

  `Description` - An entity can only contain one instance of an exclusive relation. Attempts to add further instances of an exclusive relation to an entity will replace the original relation.

  `Criticality` - This is a highly critical piece of core functionality

  `Dependencies` - Relation Removal, Relation Creation, Relation Querying

---

- ### 3.2.3 Recursive Despawning

  `Description` - When despawning an entity with a relation, the recursive despawn policy removes any resulting orphaned subgraphs as a result of that despawn.

  `Criticality` - This is a highly critical piece of core functionality

  `Dependencies` - Relation Removal, Relation Querying

- ### 3.2.4 Recursive Disconnecting

  `Description` - When despawning an entity with a relation, the recursive disconnect policy removes edges between all entities in the resulting orphaned subgraph.

  `Criticality` - This is a highly critical piece of core functionality

  `Dependencies` - Relation Removal, Relation Querying

- ### 3.2.5 Reparenting

  `Description` - When despawning an entity with a relation, the reparenting policy makes the parent graph adopt any subgraphs that were orphaned as a result of that despawn.

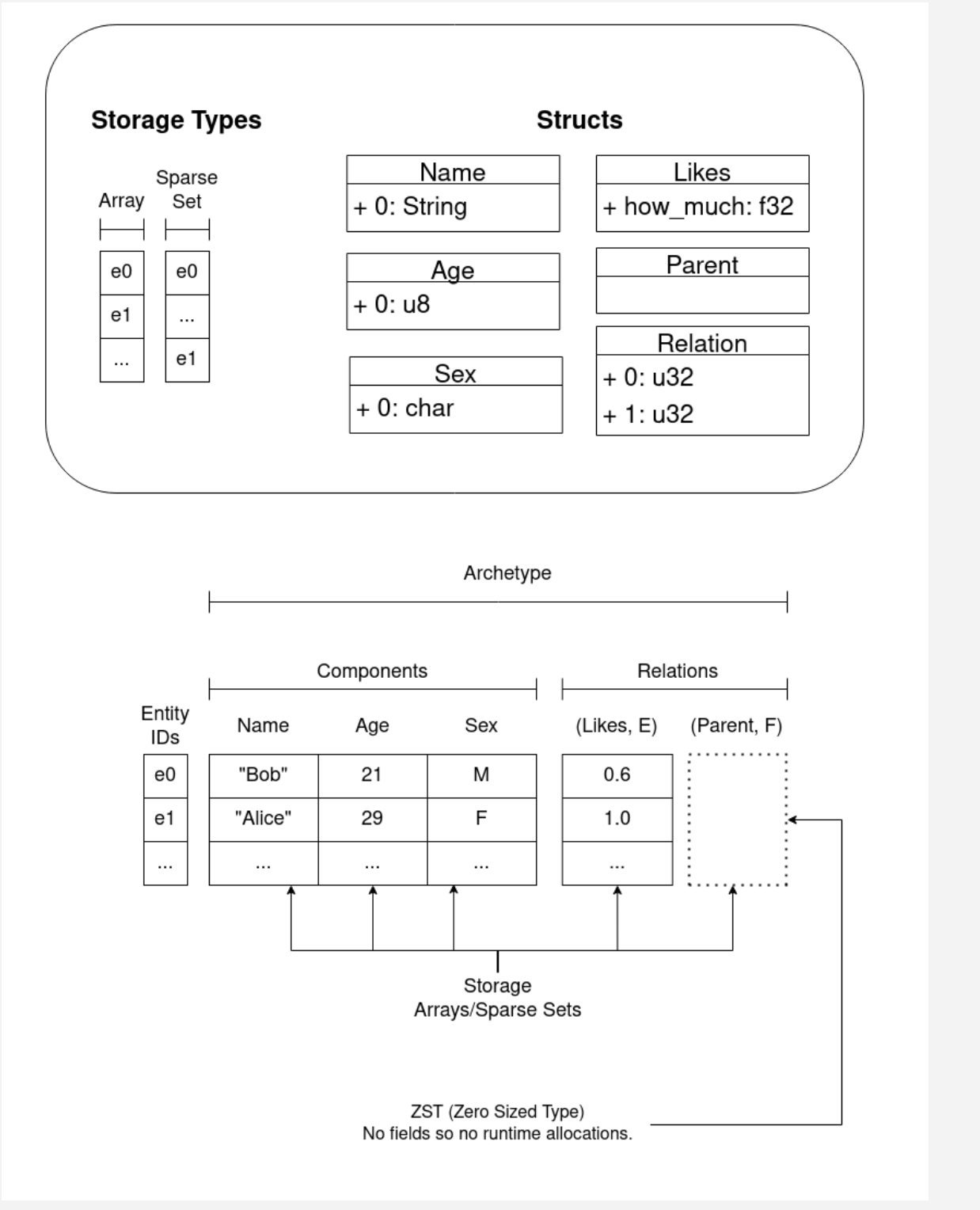
  `Criticality` - This is a highly critical piece of core functionality

  `Dependencies` - Relation Removal, Relation Creation, Relation Querying, Exclusive, Acyclic

---

4. System Architecture

Diagram showing storage architecture. Shows archetype fragmentation strategy with optional table fragmentation by choice of component storage. This allows for both dense iteration and fast lookup depending on the more frequent access pattern.



5. High-Level Design

Diagram showing orphaning policy behaviour. *Subgraphs B and C are unaffected by the removal of their parent node A*

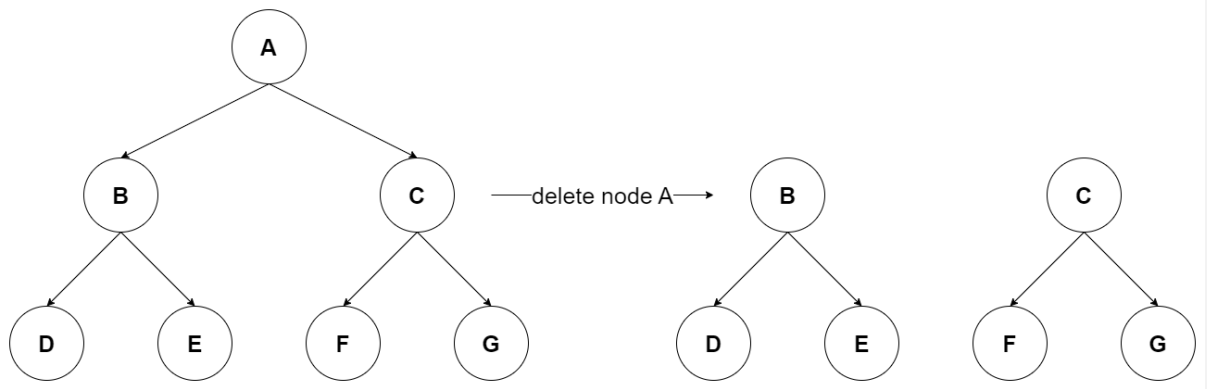


Diagram showing exclusive policy behaviour. *Node A's relation to node B is overwritten when a relation to node C is created*

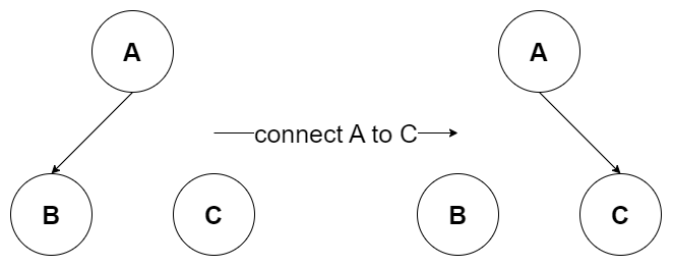


Diagram showing recursive despawning. *Subgraphs D and E are removed when their parent node B is removed*

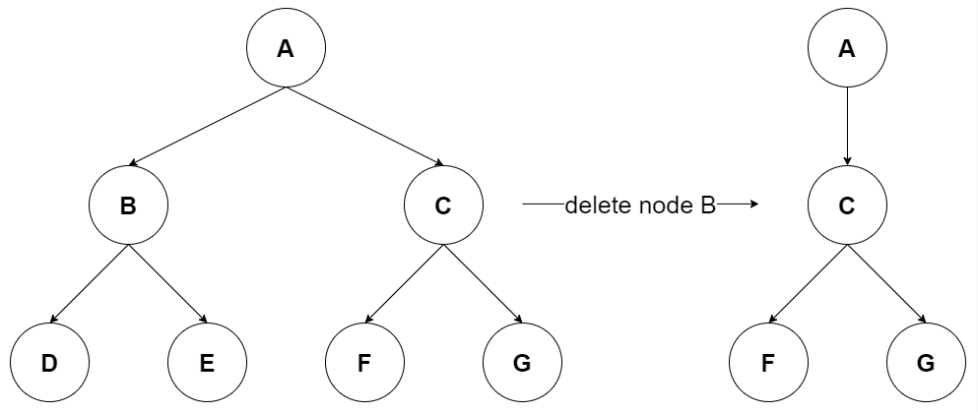


Diagram showing recursive disconnecting. ***All subsequent relations are removed when node A is removed***

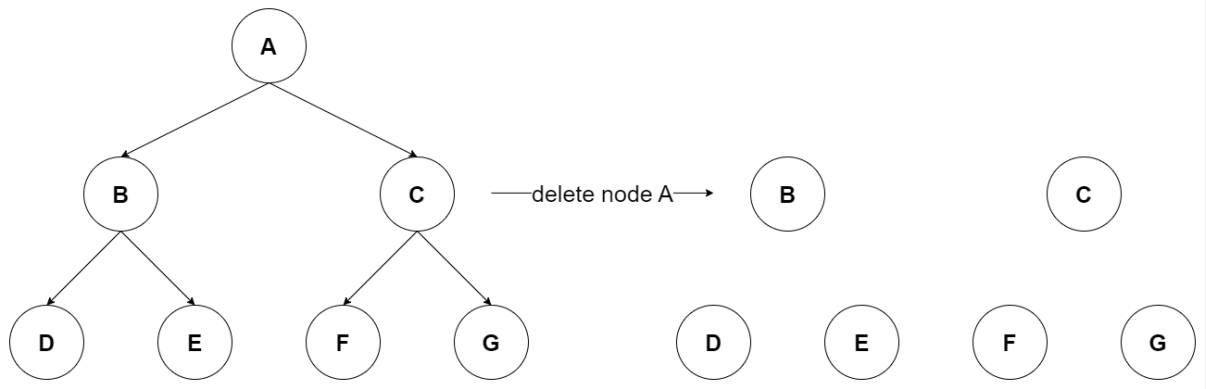
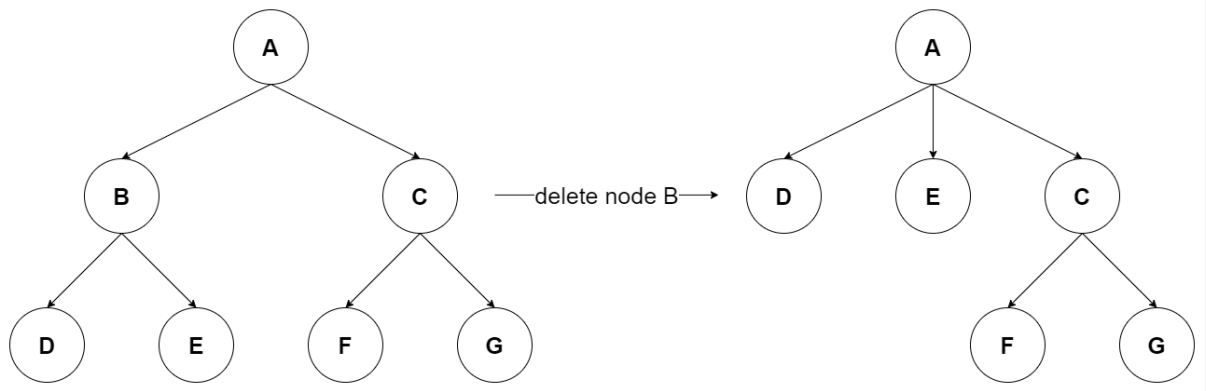
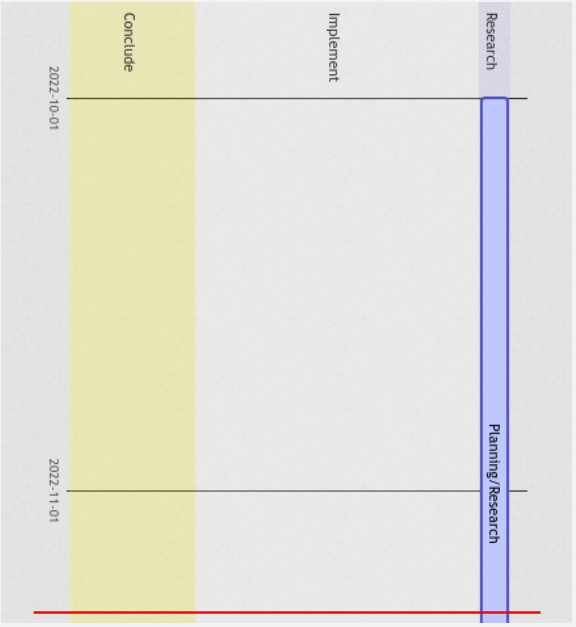


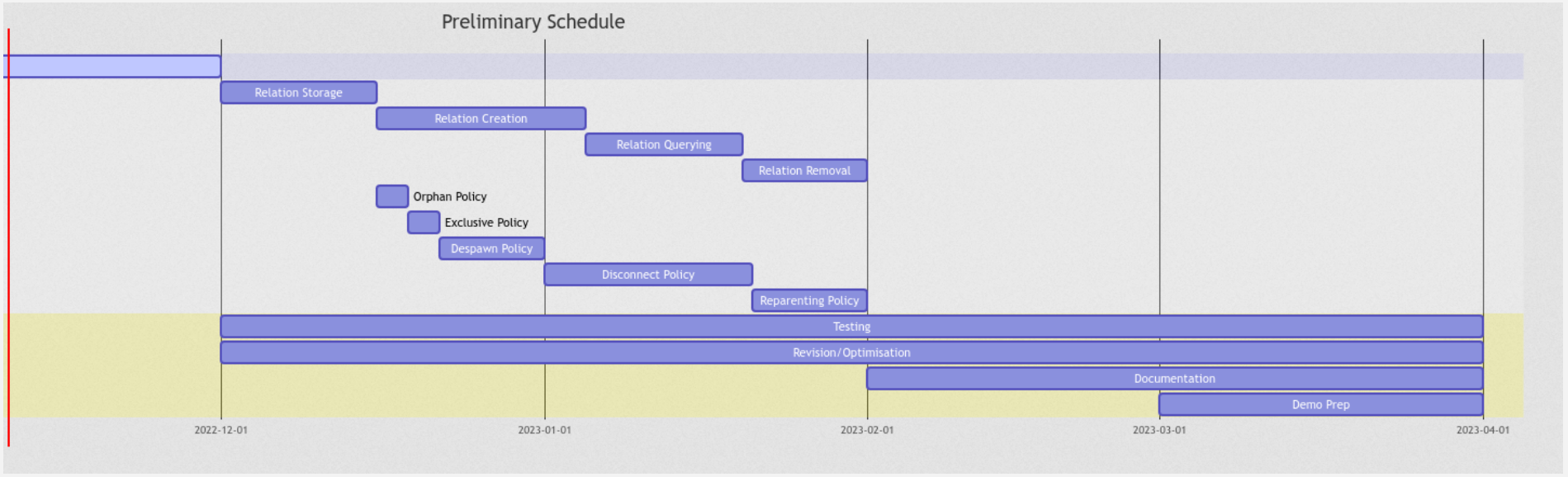
Diagram showing reparenting. ***Node A adopts nodes D and E when node B is removed***



6. Preliminary Schedule

- Planning/Research - Initially, we will spend their time learning the necessary tools, familiarising themselves with the Bevy codebase and researching possible implementation solutions and API designs.
- Initial Implementation - We hope to quickly have an initial implementation ready but it is unlikely to be as performant as required.
- Testing - Continuous testing will be performed throughout development.
- Revisions/Optimisations - Adjustments to functionality and optimisation improvements from feedback received and breakthroughs.
- Documentation - Once satisfied with the general structure of the project we will begin to document it.
- Demo Preparation - Towards the end of the development cycle we will need to focus on creating applications which demonstrate the application of our project in some of the outlined operational scenarios. Potentially GUI or path finding.




Preliminary Schedule
Relation Storage
Relation Creation
Relation Querying
Relation Removal
Orphan Policy
Exclusive Policy
Despawn Policy
Disconnect Policy
Reparenting Policy
Testing
Revision/Optimisation
Documentation
Demo Prep
2022-12-01
2023-01-01
2023-02-01
2023-03-01
2023-04-01

7. Appendices

7.1 Relevant Reading

- ECS architecture
  https://ajmmertens.medium.com/building-an-ecs-1-where-are-my-entities-and-components-63d07c7da742

- ECS architecture
  https://ajmmertens.medium.com/building-an-ecs-2-archetypes-and-vectorization-fe21690805f9

- Relations in ECS
  https://ajmmertens.medium.com/building-games-in-ecs-with-entity-relationships-657275ba2c6c

- Sparse sets
  https://research.swtch.com/sparse

7.2 Conclusions and design rationale

- Relation Storage
  https://gitlab.computing.dcu.ie/dasy2/2023-ca400-dasy2-mcguigd2/-/issues/5

- Relation Arity
  https://gitlab.computing.dcu.ie/dasy2/2023-ca400-dasy2-mcguigd2/-/issues/2

- Undirected Relations
  https://gitlab.computing.dcu.ie/dasy2/2023-ca400-dasy2-mcguigd2/-/issues/4

- Relation Targets
  https://gitlab.computing.dcu.ie/dasy2/2023-ca400-dasy2-mcguigd2/-/issues/3

- Acyclic Relations
  https://gitlab.computing.dcu.ie/dasy2/2023-ca400-dasy2-mcguigd2/-/issues/11