User Manual

Entity Relations in Bevy

Relational semantics for
the Bevy game engine.

Student 1: Yogesvara Das
Student Number: 19712101
Student 2: David Mc Guigan
Student Number: 19425942
Project Supervisor: David Sinclair

Completion Date: 05/05/2023

0. Table of Contents

1. Setup and Prerequisites

1. Install the rust toolchain including cargo and rustc must be installed. Instructions are available here: https://www.rust-lang.org/tools/install.
2. Clone the repo from: https://gitlab.computing.dcu.ie/dasy2/2023-ca400-dasy2-mcguigd2.
3. Create a new cargo project with **`cargo init`**.
4. Add the crate as a dependency by providing the path to the **`.../2023-ca400-dasy2-mcguigd2/src`** directory in the **`Cargo.toml`** file. Further docs: https://doc.rust-lang.org/cargo/reference/specifying-dependencies.html#specifying-path-dependencies
5. Be familiar with query basics in bevy:
   - https://bevyengine.org/learn/book/getting-started/ecs/
   - https://bevy-cheatbook.github.io/programming/queries.html?highlight=query#queries

2.0 Defining Relation Types

Relations can be created by impling the relation trait for any desired type or using the provided derive macro for more common trivial cases to quickly have some sane defaults.

```rust
pub trait Relation: 'static + Sized + Send + Sync {
    type Storage: ComponentStorage;
    const DESPAWN_POLICY: DespawnPolicy = DespawnPolicy::Orphan;
    const EXCLUSIVE: bool = false;
}
```

```rust
#[derive(Relation)]
struct Child;
```

When using the derive macro the **`Storage`** will be set to **`TableStorage`** and the other parameters will be using the defaults set in the trait definition. By default relations will not be exclusive meaning they can form one to many connections. The default cleanup policy will be to "**`Orphan`**" which will not perform any cleanup. More powerful behaviour can be achieved by impling the trait and overriding the defaults.

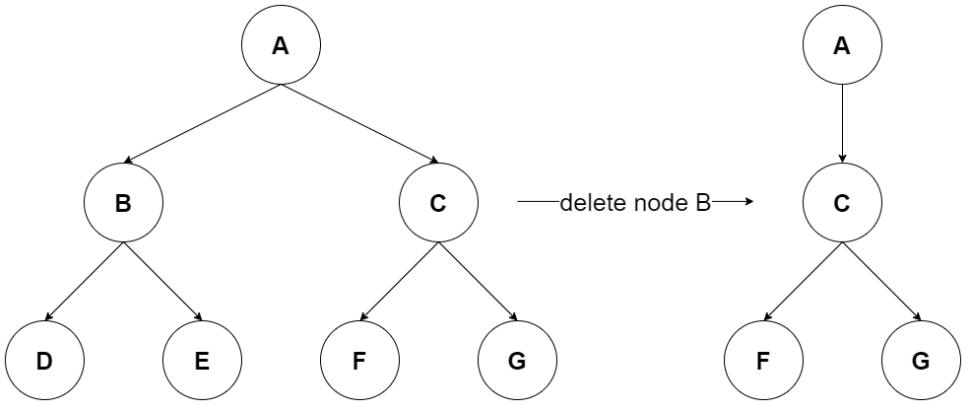
```rust
impl Relation for DespawnRelation {
    type Storage = TableStorage;
    const EXCLUSIVE: bool = true;
    const DESPAWN_POLICY: DespawnPolicy = DespawnPolicy::RecursiveDespawn;
}
```

Exclusivity can be changed from **`false`** to **`true`** making the relation type enforce 1 to 1 relationships. The despawn policy can be changed to one of the available patterns.

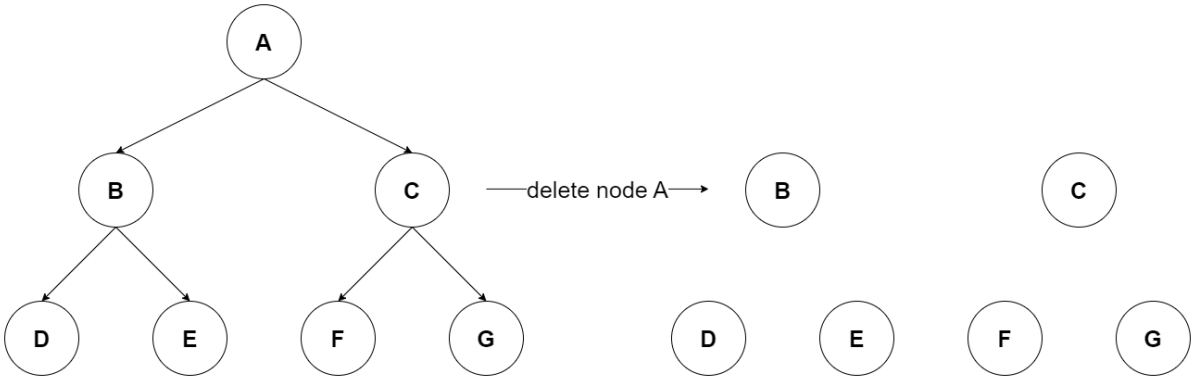
```rust
// Precedence: Most data latering operation is preferred.
// Smaller number -> Higher precedence
#[derive(Copy, Clone)]
pub enum DespawnPolicy {
    RecursiveDespawn = 0,
    RecursiveDelink = 1,
    Reparent = 2,
    Orphan = 3,
}
```

2.1 Despawn Policies

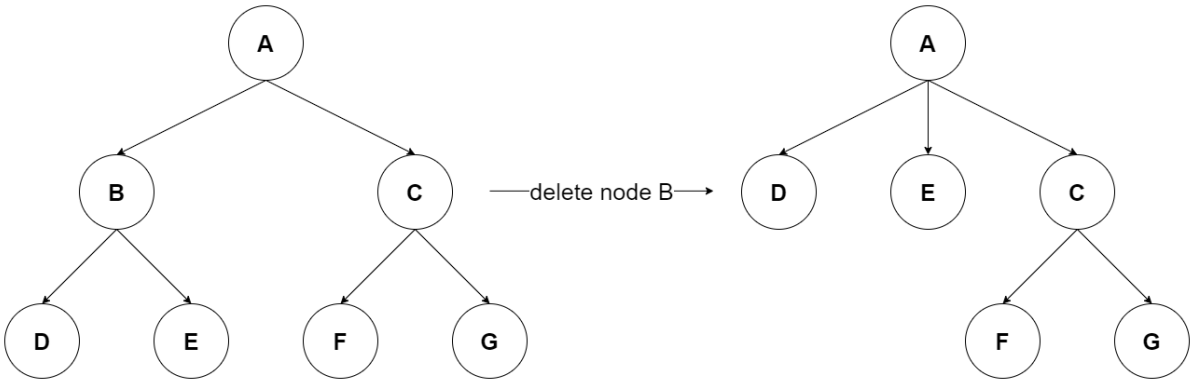
- **`RecursiveDespawn`**: Will descend the graph and despawn any entities. Cycles are checked for to prevent infinite traversal.



- **`RecursiveDelink`**: Will descend the graph and remove any relations found on any participating entities. Cycles are checked for to prevent infinite traversal.



- **`Reparent`**: Will descend the graph and will reconnect nodes to keep the graph from becoming disjoint. In the case of cycles the graph simplifies down to a bse case with 2 participants. Whichever of the participants is not removed will simply be orphaned.



Despawn policies have a precedence to them to allow users to reason about what happens when entities have multiple relations with different despawn policies. The policy that alters the most data is preferred.

3.0 Querying

Relation queries can be written like ordinary queries. Since **`WorldQuery`**s can themselves contain **`WorldQuery`**s relation queries can be done by:

1. Taking any world query **`(&A, &B, ...)`**
2. With any relation query defined with **`Relations<(&R0, &R1, ...)>`**
3. Combining them into a tuple to form a new **`WorldQuery`**

```rust
fn system(left: Query<(&A, &B, &C)>) {
    //..
}
```

|
V

```rust
fn system(left: Query<((&A, &B, &C), Relations<(&R0, &R1)>)>) {
    //..
}
```

The relations for any given relation query must be unique. Ie. any **`R`** cannot appear more than once in any **`Relations<...>`**. From there the **`.ops()`** or **`.ops_mut()`** methods can be called to perform immutable or mutable join and traversal operations with query.

The **`Relations<...>`** parameter must be the second item in the outermost tuple which has to be of arity of 2.

3.1 Joins

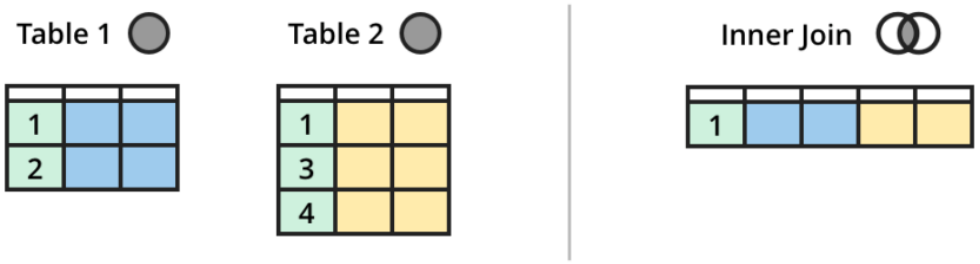
Joins can be used to combine queries with relations. Joins can be either:

1. `.join::<R>(q)`
2. `.total_join::<R>(q)`

Where `R` is a relation parameter that appears in the `Relations<...>` portion of the query and `q` is either `&Query<...>` or `&mut Query<...>`.

Both joins perform inner joins as this is the disproportionately common use case. The difference between the two being **`.join::<R>(q)`** will not give the relation parameter and **`.total_join::<R>(q)`** will give back the relation parameter. This API distinction is useful as ZST (zero sized type) relations are common in which case you can't do anything with the relation value. No **R** can be joined on more than once.

Inner join illustration from: https://dataschool.com/



Join API example

```
fn system(
    left: Query<((&A, &B), Relations<(&R0, &R1, &R2)>)>,
    c: Query<&C>,
    d: Query<&D>
) {
    left.ops()
        .join::<R0>(&c)
        .total_join::<R1>(&d)
        .for_each(|(a, b), (c, (r1, d))| {
            // do stuff
        })
}
```

Note that not all relation values have to be used in an operation. This is useful because control flow might demand multiple passes of a query that all touch different data. Without this multiple queries would be needed which might not always be disjoint: https://docs.rs/bevy/latest/bevy/ecs/prelude/struct.Query.html#disjoint-queries

3.2 Traversals

Traversals can be used to traverse graphs formed by relations. A traversal can be initiated using the **`.breadth_first::<R>(e)`** method where **`R`** is a relation parameter that appears in the **`Relations<...>`** portion of the query and **`e`** is the root entity to start the traversal from.

Currently only breadth first traversals are supported as there are no identified use cases for depth first traversal.

Traversals can be used in conjunction with joins to make some extremely powerful queries.

Traversal API example

```
fn system(
    root: Query<Entity, With<Root>>,
    mut positions: Query<(&mut Pos, Relations<&Children, &Collider>)>,
    meshes: Query<&Mesh>,
) {
    positions
        .ops_mut()
        .join::<Collider>(&meshes)
        .breadth_first::<Children>(root.get_single().unwrap())
        .for_each(|pos, collider_mesh| {
            // do stuff
        });
}
```

4.0 Setting Relations

```rust
pub struct Set<R>
where
    R: Relation,
{
    pub foster: Entity,
    pub target: Entity,
    pub relation: R,
}

pub struct UnSet<R>
where
    R: Relation,
{
    pub foster: Entity,
    pub target: Entity,
    pub _phantom: PhantomData<R>,
}
```

Relations can be set and unset with the provided **`Set`** and **`Unset`** commands. In both commands **`foster`** refers to the entity that harbours the relationship and **`target`** is the target of the relationship. Setting and unsetting can effectively add or remove components from entities creating new archetypes or resulting in table moves for entities, setting and unsetting relations must be done using commands.

Unsetting can trigger cleanup but so can setting since setting an exclusive relationship may involve unsetting that relationship first if it already exists.

Example:

```rust
fn setup(mut commands: Commands) {
    let root = commands.spawn((Pos { x: 0, y: 5 }, Root)).id();
    let a = commands.spawn(Pos { x: 1, y: 5 }).id();
    let b = commands.spawn(Pos { x: 2, y: 5 }).id();
    let c = commands.spawn(Pos { x: 3, y: 5 }).id();
    let d = commands.spawn(Pos { x: 4, y: 5 }).id();

    commands.add(Set {
        foster: root,
        target: a,
        relation: Child,
    });

    commands.add(Set {
        foster: root,
        target: b,
        relation: Child,
    });

    commands.add(Set {
        foster: b,
        target: c,
        relation: Child,
    });

    commands.add(Set {
        foster: b,
        target: d,
        relation: Child,
    });
}
```

4.1 Despawning

```rust
pub struct CheckedDespawn {
    pub entity: Entity,
}
```

When despawning entities there is a convenience command to check for relations before despawning. This command will initiate a cleanup cascade if an entity has relations with cleanup logic.

Example:

```rust
fn despawn(mut commands: Commands, destroyed: EventReader<Destroy>) {
    for entity in &destroyed {
        commands.add(CheckedDespawn { entity });
    }
}
```