1. Problem Statement

Problem. *Define a coherent but flexible way to represent the relationships of works to one another.*

The following examples represent some of the diversities of ways works might inter-relate

Example 1. One work (a chapter) is contained within another (a novel)

Example 2. One work (a one-shot) is contained within another (an anthology)

Example 3. One work (a chapter) is contained within another (a novel)

Example 4. One work (a chaptered story) is a sequel to another (*ibid.*)

Example 5. One work (a one-shot) is a retelling of part of another work (a chaptered story) from a different chacater's point of view.

Example 6. One work (a chaptered story) is book three of a sprawling epic work

Example 7. One work shares a continuity with another work, but does not directly rely on the events of it (e.g., *Wide Sargasso Sea*'s relationship to *Jane Eyre*).

However, there are many more possible relations, and we'd like a schema that allows easy expansion for frequently recurring relationships not on this list, perhaps even going so far as to allow authors to define their own relations between works; this flexibility would be very useful for fandoms. At the same time a too-rarified view of Works obscures the details and distinguishing marks of relations from clients' queries.

2. Options

2.1. **Works nest other works.**

In this scenario, a Work would allow arbitrary nesting by having a Content property that was itself an array of works. This would allow, *e.g.*, novels to contain their chapters, and series to contain their books.

```
interface Work
{
  title: String!
  author: User!
}

# sequenced works, anthologies, etc.
type BranchWork implements Work
{
  parts: [Work!]!
  # fields implementing work
}

#Chapters, one-shots, things which can be read without
# subdivisions
type LeafWork implements Work
{
  text: String!
  # fields implementing work
}
```

This does one thing well and three other things poorly. On the positive side, this gives the user an enormous amount of freedom in segmenting their work into pieces. If they want subsections or subchapters of a work they can go ham. But the aribitrariness of the containment structure here also means that this has three severe limitations

(1) It flattens all relationships between works to containment, which isn't necessarily right. Indeed, that's flat-out wrong (translations, remixes, etc).
(2) The simplicity makes the meaning of the containment relationship opaque to queries. Is this a chapter divided into subparts? Is this a novel divided into chapters? An anthology divided into entries? A reading list with entries? We have no way of knowing.
(3) Nothing actually enforces the tree structure of works-containing-works in this schema. A BranchWork could, e.g., reference itself and never expand into a LeafWork with actual readable text.

2.2. **Work types are defined in an enum.** This takes the previous approach and enriches the works which contain other works with a field identifying the nature of the containment.

```
enum WorkContainer{
    ANTHOLOGY
    SERIAL
}

type BranchWork implements Work
{
  # sequenced works, anthologies, etc.
  containerType: WorkContainer
  title: String!
  author: User!
  parts: [Work!]!
}
```

This somewhat alleviates problem 2, but does not move us forward on either 1 or 3. Moreover, it resolves 2 by doubling down on 1 – i.e., the containment relationship is transparent to queries because it's restricted to values in the enumeration. Maybe even worse, it throws a wrench in the one unequivocal good thing the previous option had going for it, because now the sequence of containers for a sub-division of a work all must have a corresponding value somewhere in WorkContainer.

2.3. **Relations are defined as connection objects.** Consider a different appraoch

```
interface Work{
  title: String!
  author: User!
  anthologizedIn: anthologyConnection
  serializedIn: serialConnection
  translatedBy: translationConnection
  retoldBy: retellingConnection
  text: String!
}
```

```
type GroupedWork implements Work
{
  # fields implementing interface
}

type ReadableWork implements Work
{
  # fields implementing interface, AND
  text: String!
}
```

2.4. **Relation is disaggregated from reading sequence.**