TopicQuests Solr API

Latest: 20130615

Contents

Background

In the largest picture, a *topic map* is a collection of *topics* represented as *nodes*, where those nodes can be connected with each other through relations which are represented as *tuples*, where a tuple is just a specialization of a node. There are two *configurations* of nodes and tuples applied in the topic map:

- One in which a Node is *related* to another Node: Figure 1
- One in which a Node has a particular property type and value which, itself, is considered a *topic*, in which case, a tuple represents, not a relation to another topic, but, instead, the property type, its value, and the value's type (e.g. date, weight in pounds, say, ...): Figure 2

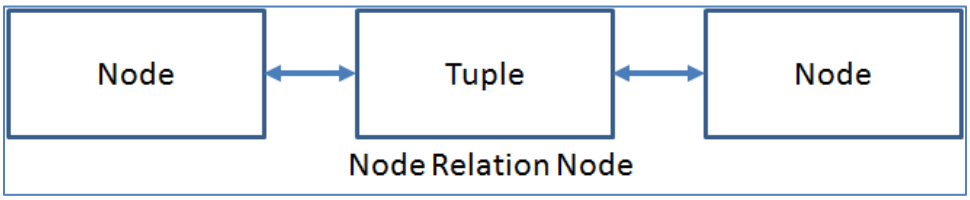

Figure 1. Node-Tuple-Node Configuration

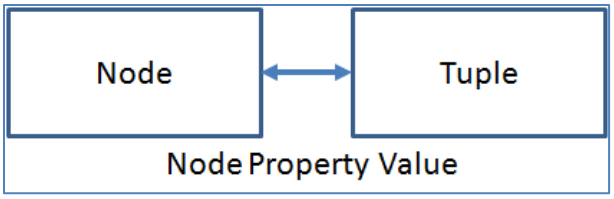

Figure 2. Node-Tuple Configuration

We will have much more to say about these configurations as we proceed, but that's the really big picture of what is captured in Solr as a topic map. The intent, here, is to build on that big picture with successively deeper dives into what the topic map is, how it is represented, and how the APIs provided in the collection of TopicQuests code interact. First, however, we sketch the project's codebase, the libraries on which the entire SolrSherlock system is built.

On Graphs

Appendix A (below) is included. There, we provide a glossary of terms used in a set of sketches which describe various types of graphs. Overall, here is a brief description of three kinds of graphs which are common to many conversations:

- A *mind map* is a kind of graph which centers a prime topic, then radiates lines (arcs) out from that topic which end in various other topics, each of which can then radiate lines out to new topics. The result is a simple graph which shows *that* topics are related to each other. It does not show *how* those topics are related.
- A *concept map* is a like a mind map but with *labeled arcs*; thus *how* the concepts (topics) are related is defined.
- A *topic map*, especially as we define them in TopicQuests, takes the *labeled arcs* one step further: the relations (connections, links, assertions) which connect one topic to another *are topics themselves*. We care about representing a relation as a topic in the following sense. A relation is an asserted connection between two topics. If the relation is "causes", then when topic A gets related to topic B with the "causes" relation, the *meaning* of the topic which does the relation is, in fact, "A causes B". In many cases, a causal claim like that is not controversial, but in some cases (think: climate change, say), causal claims are frequently at issue. We represent those claims as topics precisely so that the claim itself ("A causes B") as represented in that topic can be the target of conversations. For instance, consider the assertion "A causes B" is false because.... The causal claim is now the subject in another claim. We need many relations to be topics.

On Nodes

<ToDo>

On Tuples

<ToDo>

TopicQuests Libraries

Here, we sketch the core libraries, followed later by the

Core Libraries

TopicQuestsSupport[1]

A very low-level library of components collected and adapted from other project. Included here are these:

- `ConfigPullParser` which is the parser used for the various `config.xml` files in our projects
- `DateUtil`
- `LRUCache` and its *interface* `IRemovableCache`
- `ISO8601DateParser`
- `StringUtil`
- `LoggingPlatform` which is a class which can be called statically to provide a uniform logging infrastructure
- `Tracer` which is a trace utility by way of the `LoggingPlatform` that creates gzipped files of a fixed size, and is used when one wants to *trace* particular events in ways other than debug log entries
- `TextFileHandler` provides a variety of file handling features

SolrPlatformCore[2]

This is the library which supports building and maintaining topic maps by way of interactions with a Solr installation. In the end, it communicates with Solr over HTTP connections by way of the SolrJ client. What is important here is that the entire collection of APIs which drive that interaction occur through *interface-based* implementations. It is those interface specifications which are important to users. Here are the API interfaces, ordered somewhat in ascending level of use (most-internal first):

- `IResult` which defines a *return value* used in many high-level APIs. The value of an `IResult` object is that we can isolate errors rather than tossing and trapping `Exception`s. We can also return more than one value from a given Java or Scala method. The object carries error messages and return values.
- `IBootstrap` which is used to define a common API for writing bootstrapping code. Bootstrapping is further supported by the class `BootstrapBase`. To see it in use, see `CoreBootstrap`.
- `IMergeImplementation` is used in the evolution of the merge platform
- `INode` which, in a sense, is the core object of the topic map, which, by extension, means it is the core object throughout SolrSherlock to the extent that SolrSherlock interfaces with the topic map; it is true that SolrSherlock will build data structures

[1] TopicQuestsSupport: https://github.com/SolrSherlock/TopicQuestsSupport
[2] SolrPlatformCore: https://github.com/SolrSherlock/SolrPlatformCore

where do not have anything to do with the topic map except indirectly where they support building topics. We have much more to say about nodes.

- `ITuple` which extends `INode` (any tuple is also a node).
- `IDataProvider` which is extended by `ISolrDataProvider`. This is the core mechanism (API) by which code *talks* to Solr. At this point, we have nodes and we have Solr. Both are defined. It remains, then, to define how to unite create and manipulate the nodes we send to and retrieve from Solr.
- `INodeModel` which is the primary collection of methods useful for creating nodes which are subclasses or instances, together with other useful methods.
- `ISolrModel` provides a collection of methods for *listing* nodes by various features.
- `ITupleQuery` provides a collection of methods for finding tuples by tuple features
- `ISolrQueryIterator` provides an *iterator* for walking along a query to Solr which returns a collection of values longer than is typically served. For instance, if one wants to query a topic which has a particular collection of values, say, 100 in count; we fetch those, typically, 20 at a time. `ISolrQueryIterator` maintains the ability to repeat the query using an offset and count.
- `ISolrClient` which is the primary interface with which to implement instances of SolrJ, the HTTP client for Solr. It is implemented for both single-instance Solr installations, and SolrCloud installations. That implementation remains to be tested.

At the same time, many *constants* are defined in interfaces as well. Some of these interfaces are known as *legends* in the sense that any good map has a *legend* which defines the terms and symbols used by that map; we define *types* of all kinds through legends. Each legend is *bootstrapped* into the topic map before that topic map is ready for use. In SolrPlatformCore, the key legends are:

- `ITopicQuestsOntology` which is the core ontology/typology for the topic map. This interface is the result of evolving topic maps since the very beginning of this project in 2003
- `IBiblioLegend`
- `ISocialLegend`
- `IPersonLegend`
- `IRelationsLegend`
- `INodeTypes`
- `ISolrLanguageCodes` (reflected in the creation of Solr *fields*)
- `IHarvestingOntology` (not yet bootstrapped)
- `IDublinCoreOntology` (not yet bootstrapped)
- `IConceptualGraphLegend` (not yet bootstrapped)
- `ICoreIcons` which is not a legend in the usual sense; this defines a collection of icons used during bootstrapping to give various classes both small and large icons

IResult
<ToDo>

INode
<ToDo>

ITuple
<ToDo>

ISolrDataProvider
<ToDo>

INodeModel
<ToDo>

ISolrModel
<ToDo>

Extension Libraries
We now sketch those libraries which rely on the core libraries to add functionality to both the topic map and to SolrSherlock

SemiSpaceBean[3]

SolrInterceptor[4]

SolrAgentCoordinator[5]

SolrAgentFramework[6]

SolrMergeAgent[7]

[3] SemiSpaceBean: https://github.com/SolrSherlock/SemiSpaceBean
[4] SolrInterceptor: https://github.com/SolrSherlock/SolrInterceptor
[5] SolrAgentCoordinator: https://github.com/SolrSherlock/SolrAgentCoordinator
[6] SolrAgentFramework: https://github.com/SolrSherlock/SolrAgentFramework
[7] SolrMergeAgent: https://github.com/SolrSherlock/SolrMergeAgent

Using those APIs: Communicating with the Topic Map

In this section, we sketch the processes by which it is possible to use SolrSherlock Java libraries described above to fabricate systems which can communicate with the topic map. This means:

- Accessing the topic map to fetch topics
- Creating topics to add to the topic map

<ToDo>

Appendix A: Graph Structures

1 Terminology

Actor

An actor is any *subject* that can play a *role* in a typed relationship with another subject.

Addressable

Any object that is *identified* is *addressable*. For instance, if a document contains several paragraphs, the paragraphs are rendered addressable if they each are surrounded by markup that includes a unique identifier.

Arc / Arrow

In simple graph notation, as in concept maps, an arc (aka: arrow) is the line, typically directed and labeled, that represents a relation between two nodes. I topic and subject maps, arcs are replaced by association or assertion nodes.

Assertion

An assertion is, at once, a statement of fact, and, in particular, the name given to a particular node in a subject map graph, the *aNode* or *assertion node*. An assertion node is the proxy that represents an instance of a typed relation between two actors.

Frame

An identified container of slots that serves the purpose of representing a subject. In Minsky's (1975) terms, a frame was defined as a representation of a stereotypical situation or concept. More recently, frames came to serve the need to represent entire subjects or concepts. Frame notation is an early form of a graph representation, but without the nodes and arcs. Frames are the nodes, and some slot values point to other nodes; thus a graph exists within a knowledge base of frames.

Key

In our parlance, a *key* is, itself, a *subject* defined in a particular subject map. Typically, that subject defines a particular *property type*. For instance, `NameString` is a property type that defines a property, the value of which will be a String object that is taken to be a *name* for a particular subject.

Locator

In our parlance, a *locator* is the equivalent of a database identifier. For each identified object in the database, the locator must be unique to that database.

Node

In a graph, a node is also known as a vertex. In our parlance, *node* is synonymous with *proxy* or *SubjectProxy*.

Property

An identified container for one *key-value* pair.

Property value

A value associated with a particular *key*. Said value could be either a single object, or a collection of objects. In computer representations, all property values must be expressed as strings, but those strings can represent numbers, symbols (*locators* of *subjects*) or just words, say, in a sentence.

Proxy

An identified container of property instances. SubjectProxy is the term given by the TMRM (ISO, 2005) as that which represents a particular subject.

Relation

A potentially ambiguous term, but our use of it relates closer to the philosophical notion that it is *a property or predicate ranging over more than one argument*.[8]

Role

Actors can play different roles in relationships held with other actors. Examples of roles include *husband*, *wife*, *employer*, and *employee*.

Scope

A scope is a *subject* that *constrains* another subject. For instance, a scope might be a date on which an event occurred, or the language in which a particular subject name is written.

Semantic Network

Quoting John Sowa (2006):

> "A semantic network or net is a graphic notation for representing knowledge in patterns of interconnected nodes and arcs. Computer implementations of semantic networks were first developed for artificial intelligence and machine translation, but earlier versions have long been used in philosophy, psychology, and linguistics."

Slot

Slot is a synonym for *property* in the context of frame-based knowledge representation

Slot value

See *property value*.

Subject

In our parlance (ISO, 2005), a subject is anything you want to talk or think about, and represent in a subject map. Subjects can be highly general, as in the subject of *chemistry*, or they can be much more specialized as, say, *stereo-doppler chemotactics.*

---

[8] Relation quote: http://en.wikipedia.org/wiki/Relations

2 frame-based representation

In general, the representations we discuss fall under the rubric *semantic network*. From time to time, it is necessary to express concepts in knowledge work as *frames* (Minsky, 1975). This approach provides a convenient shorthand for illustrating the contents of a subject proxy. Each subject proxy starts with a *locator*, which is a *unique to the database* identifier. Each proxy then contains many *key-value* pairs called properties, where each key is, in *frame parlance*, a *slot*, and the *frame* itself represents the proxy. As an example, consider the subject proxy that represents an 8-year-old female with the name Sue Smith.

```
Proxy locator="HumanType"
  ...
Proxy: locator="123a45fg6_Person"
  instanceOf      #HumanType
  nameString      "Sue Smith"
  hasAge          "8"
  hasFather       #343242ff45_Person
  hasMother       #252589898_Person
```

For purposes of explanation, we included the proxy that is the super-type for objects of this type. In this example, we are able to disambiguate this "Sue Smith" from others if we happen to know her age or either of her parents' identities.

3 Graph representation

One aspect of topic mapping is that of providing *addressable* units of information. This chapter describes three different ways in which collections of subjects can be modeled. We start with the simplest graphical structure, concept maps. We then sketch the two approaches available under ISO 13250 topic mapping standards.

One approach to thinking about the differences is described as taking a *frame-based* view (Section 2 above) of these approaches. This view will be used to compare concept maps to subject maps. Since frame-based views involve properties (slots), differences will be seen to exist in the ways in which properties are used. Start first with two concepts from Figure 12.1: `Concept Maps` and `Organized Knowledge`, and view them in the simplest possible frame-slot representation as follows:

```
Frame:  id="conceptMaps"

     nameString       "Concept Maps"

     represent        #organizedKnowledge
```

That pair of frames represents precisely what Figure 12.1 says, that Concepts Maps represent Organized Knowledge. We went one step further and added the obvious: Organized Knowledge is represented by Concept Maps. The relationship between the two concepts is represented by a particular *property type* (called a *key* in the TMRM vernacular). What if we want to talk about that "represent" relation? What if we want to annotate that particular *instance* of the relationship between those two concepts with a comment?

We need a way of rendering the relations *addressable*. Suppose on this representation by building in a simple typology. With that typology, we can create typed links. The new knowledge base (KB) fragment looks like this:

```
Frame:  id="linkType"

      nameString       "Link Type"


Frame:  id="conceptType"

      nameString       "Concept Type"


Frame:  id="represent"

      nameString       "Represent"

      instanceOf       #linkType


Frame:  id="representedBy"

      nameSTring       "Represented By"
```

In this new KB fragment, we note that we now have moved the links *outside* as a new *type*. We then aggregate those links in a "links" property. Did we render our relations addressable? It would appear so, given the two frames identified as `represent` and `representedBy`. Can we use those two frames to attach comments? The correct answer is "Yes" and "No". We *can* indeed attach comments to either of the two frames, but we now create a scenario in which our comment might be ambiguous. Suppose the "represent" relation is used elsewhere in the map. What, then, does our comment mean?

We are now confronted with the issue of altering our KB fragment to permit individual addressability, to separate subjects. That is, the *class* of relationships known as "Represent" or "Represented By" must have individual *instances* created. Each individual instance is, itself, a *subject* distinct from all other

instances of the same relation type. Subject identity for each instance can be defined by three elements: the relationship type, and the identity of each of the participants in that relationship instance. Our new KB fragment will borrow from the URI and PSI concepts and create an identity property value that reflects the specific identity of the relationships. Our `identity` property, for simplicity, will not include the rest of the string; for sake of illustration, one value from the following KB fragment might look like this:

```
http://topicspaces.org/PSI#conceptMaps.represent.organizedKnowledge
```

Note that alternatives to subject identification might include simply listing the identities of each member in a property and combining that with the relation type's identity, which that single string already does. Consider this: a single string notation as illustrated above does not as easily satisfy the needs of a parametric search. For instance, one might ask:

What do Concept Maps represent?

To derive a response, one might start by first translating the name strings in the query into subject identifiers in the KB, and from there, follow an algorithm that might look like the following:

- Collect all instances of the relation
- For each instance of the relation
    - If instance member = subject
        - Collect other member
- Return collection

If we use the identity string illustrated above, and not a list of members, then we require a method that compares the identity string to the member's identity. If, instead, we include a list of members, we simply test for set membership. In any case, following is the revised KB fragment.

```
Frame:  id=”linkType”
      nameString       “Link Type”

Frame:  id=”conceptType”
      nameString       “Concept Type”

Frame:  id=”represent”
      nameString       “Represent”
      subclassOf       #linkType

Frame:  id=”representedBy”
      nameSTring       “Represented By”
      subclassOf       #linkType

Frame:  id=”123456”
      instanceOf       #represent
     identity
      “conceptMaps.represent.organizedKnowledge”
```

With that KB fragment, we successfully migrated a simple concept map to a structure much closer to a subject map, where each and ever element is an addressable subject. If we want to annotate the "Represent" relation that exists between "Concept Maps" and "Organized Knowledge", we simply address the subject identified as `123456`, and link our comment.

The difference in the way slots (properties) are used is made clear when we illustrated the change from using typed properties to represent a relation type, and using generic properties to collect externally-represented subjects that are each instances of relations in which the subjects are participants.

3.1 Concept Maps

Concept maps are the simplest graphical representation of two entities and a relationship between them.

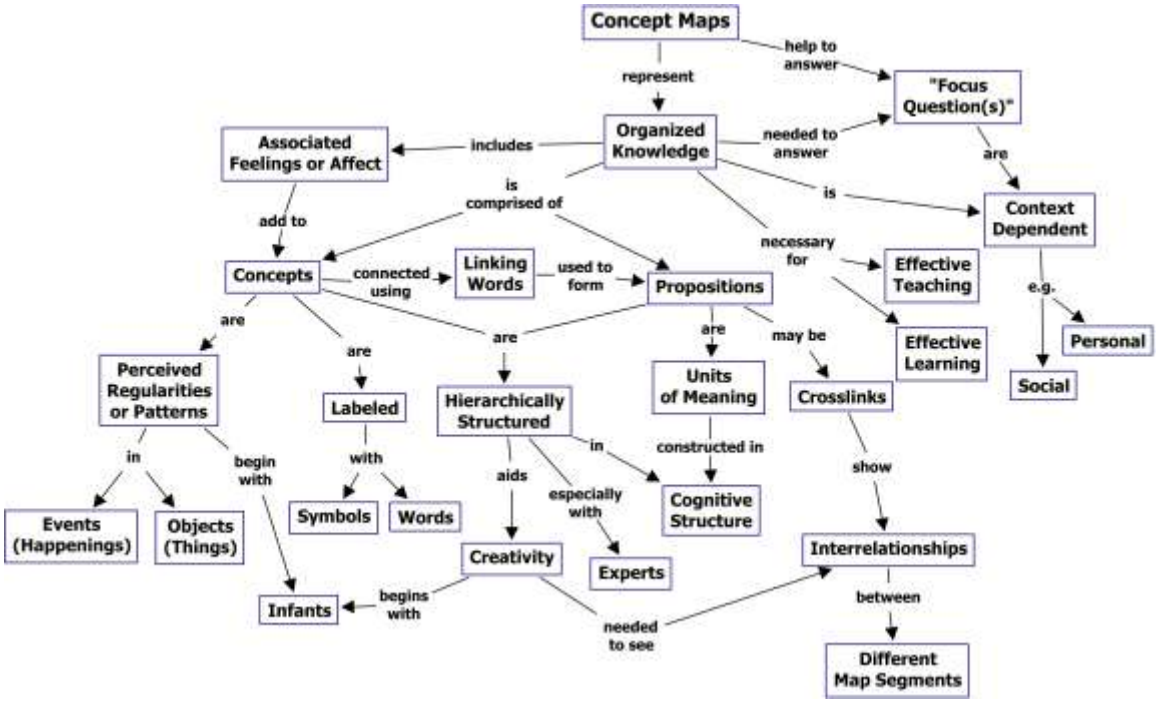

Figure 3.1. Concept Maps (from (Novak & Cañas, 2006) with permission)

Compendium and Cohere each present nodes and arcs as concept maps. We turn next to a discussion of the ways in which topic maps build on the basic notions of concept maps.

3.2 XML Topic Maps

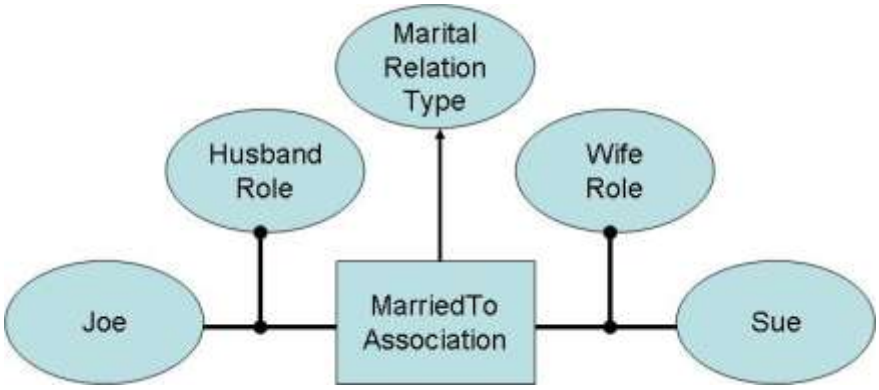

Figure 3.2. XML Topic Maps

An important distinction is made between Figure 12.2 and Figure 12.3. The XML Topic Maps (XTM) standard does not declare an Association object as a subject. Thus, the illustration makes use of a rectangle, where an oval is always a subject.

3.3 TMRM Subject Maps

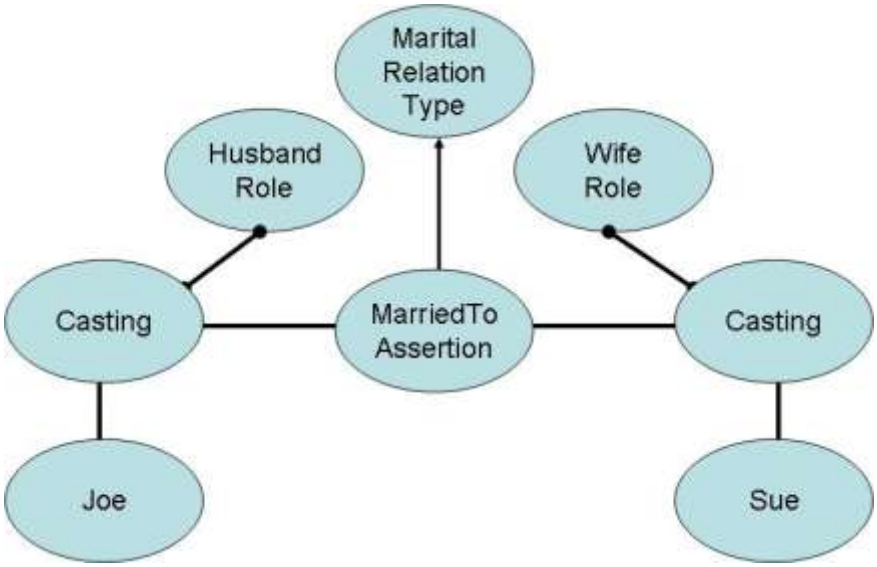

Figure 3.3. Subject Maps—Big Assertion Diagram

The so-called “Big Assertion” is perhaps the largest structure declared in early documents describing the TMRM. It is not necessarily the only approach. A simpler approach, called the TS_Assertion in TopicSpaces, simply does not represent the “Casting” subject on each side of the relationship.

Why would we care about the casting subject? Humans exhibit a tendency to think in terms of *role models*: people *as* something: Joe *as* Sue's husband, that fellow sitting at the table in a courtroom *as* the driver of the getaway car in a bank robbery. A casting subject allows us the opportunity to represent the concept of *as*, to model roles explicitly. Of course, we can infer that representation when needed.

A variety of graph structures appropriate to TMRM subject maps is given in (Durusau & Newcomb, 2005). Here, we will extend the discussion to illustrate a means by which we can use assertion nodes as actors that play roles in relations. For that, we will create a scenario consisting of two statements, then form graph structures from those statements, and then discuss the results. In the following visual representations, we change the representation to use rectangles for subjects and ovals for relations. We will then show that the relations are, indeed, subjects.

In our parlance, we consider an assertion to be a statement made as a "fact". We also use the term "assert" as an act of making a statement. The term "assertion" is interchangeably used for "relation". In subject mapping, one *asserts* a property on a subject, e.g. "Joe was born on July 10, 2001", which maps to a "birthdate" property type in the representation for the subject we identified by "Joe". One also asserts relations between subjects, as we discuss next.

Consider two assertions:

1. Politician A claims reduced taxes
    a. Subject: Politician A
    b. Assertion type: reduced
    c. Subject: taxes
    d. Evidence: claim
2. Records show Politician A increased taxes
    a. Subject: Politician A
    b. Assertion type: increased
    c. Subject: taxes
    d. Evidence: records (data)

We have options: we could model these assertions in Compendium, or we can model them in a Cohere-like structure[9]. For this discussion, we take the Cohere-like structure approach. Following are graphical illustrations of those two assertions:

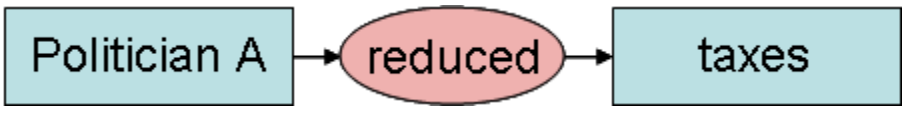

Figure 3.4 Assertion 1

[9] Cohere: http://cohere.open.ac.uk/

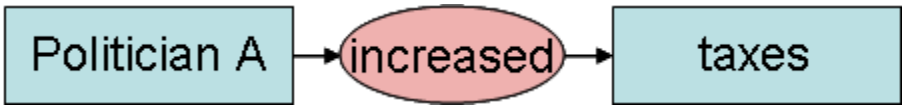

Figure 3.5. Assertion 2

We now perform a Cohere-like connection with a slight difference: in TopicSpaces, there is the opportunity to render connections as bi-directional. Some connections are symmetrical, where the arrows in both directions represent the same assertion type. Other connections are not symmetrical; the arrows in each direction represent a different assertion type. We illustrate a TopicSpaces-like connection in the next figure:

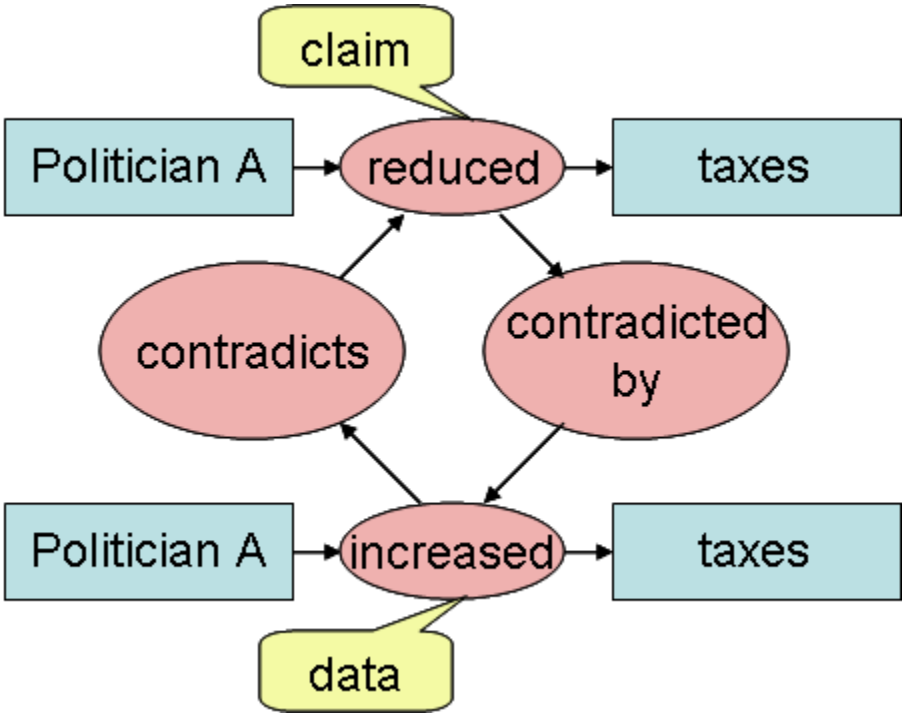

Figure 3.6. TopicSpaces-like coherence relations

In this visual ontology, we have separated subjects from assertions using both color and node shape. We use a *callout* to distinguish *scoping* subjects, `claim` and `data`. The question remains: how does

TopicSpaces allow an asserted relation to serve as a subject such that it can be the target of another coherence relation? For that, we illustrate next an important characteristic of subject maps: all objects in the graph are subjects, even when used as a relation.

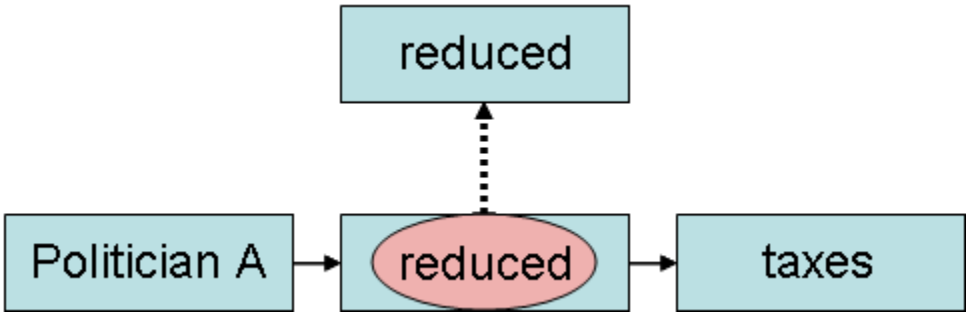

Figure 3.7. TopicSpaces internal representation of a relation

In TopicSpaces, relations are modeled as instances of a subject which is also a relation or assertion *type*. The *identity* of the *subject* represented by the "reduced" relation instance is based on three properties:

- The assertion type: `reduced`
- Both participants (members, actors): `Politician A`, and `taxes`

Thus, the *meaning* of the "reduced" instance node is precisely the same as the entire assertion: "Politician A reduced taxes"; that node stands alone as SubjectProxy, the subject of which is that entire statement. That one node *reifies* the entire claim by `Politician A`. At the same time, it connects the precise subjects `Politician A` and `taxes` together as if a *trail* is being forged between those two subjects.

Since that node contains representations of the subject's identity, it is qualified to be a *subject,* which means that it can serve as an actor or member in relations asserted as needed.

Cohere applies coherence relations as an approach to linking ideas found on the Web. We model coherence relations in a bi-directional way as an *implementation-level* decision. It is also possible to write inference rules that can answer questions such as:

- Is Assertion 1 *contradicted by* any other assertion?

- Which assertions contradict Assertion 1?

The particular implementation-level decision is simply an *editorial* decision made by this topic map author; other topic map authors are free to write inference rules instead. Given inference rules, database queries can be formed that find appropriate answers.