Maintaining Temporal Perspective

Ian Emmons and Douglas Reid

Raytheon BBN Technologies, Arlington, VA 22209, USA
{iemmons,dreid}@bbn.com

Abstract. We present methods for annotating data with the time when it was learned and for answering queries according to what was known at any point in time. Specifically, we present an RDF knowledge representation that associates facts with their transaction times, and a query mechanism that transforms a time-agnostic SPARQL query and a point in time into a new, time-sensitive query. The transformed query yields the subset of the results of the original query that were valid at the indicated time. In addition, the methods presented here enable non-destructive merging of coreferences. These techniques apply broadly to storage and retrieval systems that require time-based versioning of data and are essential for maintaining temporal perspective in rapidly-evolving analytical environments.

1 Background

There is a large body of work in the theory and construction of temporal databases, as summarized in [4]. This paper describes an application of that body of research to support the development of an operational, temporally-annotated *semantic* database.

The solution presented here grew from a project to develop a risk analysis application for assessing risks to one particular high-value resource. This system continually gathers data from five relational databases (non-temporal) and stores it as RDF in a triple store. The data includes events as well as latest current state, both of which are time-sensitive.

Analysts use the application to perform daily risk assessments, and these results are also placed in the triple store. Later review of these analyses is an important part of the analysts' work. This leads directly to the time-validity requirement: the triple store must maintain the temporal perspective of the data for subsequent review and inspection, because older analyses mean little in the context of current data. Note that temporal perspective may also be useful in the analysis task itself. For instance, the age of what we know and the order in which we learned it sometimes affect the interpretation.

The solution consists of two primary components. First is the RDF knowledge representation, which associates facts with the time intervals in which they were known (traditionally referred to as the *transaction time* of a given fact[4]). The second is the query rewriter, which transforms a time-agnostic SPARQL query and a point in time into a new, time-sensitive query. The transformed query yields the subset of the results of the original query that were known at the given time.

This solution aims to track when facts were considered true. For that purpose, we have chosen a linear, discrete temporal model (defined in [4]) that deals with transaction times of facts. It is not intended to manage the explicit temporal aspects of data, such as occurrence times of events. In this way, this solution may be thought of as a form of provenance tracking.

2 The Knowledge Representation

We describe the knowledge representation through an example in which three data imports occur (see the summary timeline shown in Table 1). The first import yields the following statements:

```
:Person1 a :Person ;
  :name "Robert Jones" ;
  :ssn "123-45-6789" .
```

These describe a person with name and social security number (SSN). Identifiers preceded by colons are URIs whose base URI has been suppressed for brevity. To this the system adds a proxy consisting of the following statements:

```
:Proxy1 a :Individual ;
  :hasPrimitive :Person1 ;
  :usesValue [ :Person1 :name "Robert Jones" ] ,
             [ :Person1 :ssn "123-45-6789" ] ;
  :temporalIndex :TmpIdx1 .
:TmpIdx1 a :KbProperInterval ;
  :startedBy :Time1 ;
  :finishedBy :EndOfTime .
:Time1 a :DateTimeInterval ; :xsdDateTime "2009-08-17T00:00:00" .
:EndOfTime a :DateTimeInterval ; :xsdDateTime "9999-12-31T23:59:59" .
```

Proxies represent the sum total of information known about an entity for a specified time interval within our system. Borrowing from situation calculus, the proxies provide a mechanism for encoding the history of knowledge (or situation) and for resolving the truth values of properties (or fluents) of entities throughout that history[3]. From a philosophical perspective (i.e., BFO[5]), proxies can be viewed as Processual Entities that capture the time-specific Qualities and Realizable Entities of a particular Independent Continuant within our system.

The example proxy is of type `:Individual`, one of two subclasses of `:Proxy`, and points to the person entity via the `hasPrimitive` property. The choice of terminology "primitive" here will make more sense when we discuss coreference resolution below. The `usesValue` properties point to the attribute values of the person that are known during the time period of this proxy. Note that they point not to the objects of the person attribute statements, but rather to reifications of those statements.

There is no `usesValue` for the type of the person entity. This reflects a conscious design decision to avoid introducing time dependence into the RDFS inference performed by our triple store, greatly simplifying the implementation. One consequence of this decision is a restriction on the ontology: Classes must not carry time-dependent meaning. For instance, `:Person` is a perfectly reasonable class, but introducing a subclass of it like `:CEO` would be a mistake, because a person who is a CEO holds that position for only a portion of their life.

The remainder of the proxy, from the `:temporalIndex` property on, encodes the proxy's transaction time. This extends from the time of import (the opening second of August 17, 2009 in this case) until the end of time. If the representation seems more complex than necessary, this is because it must comply with the combined requirements of the OWL-Time ontology[1] and of the temporal index associated with our triple store,[2] Parliament™[2]. The temporal index, based upon Allen's Interval Algebra[1], allows us to query efficiently for such things as time intervals containing, intersecting, before, or after a given time interval.

Now suppose a second import, from a different data source the next day, yields this data:

```
:Person2 a :Person ;
  :name "Bob Jones" ;
  :ssn "123-45-6789" .
```

Due to the matching SSN and the similar names, most would say that these two data entities are "obviously" the same real-world entity, in other words that they form a coreference that we want to resolve, or merge, into a single entity. This is a common problem with multi-source data. It often arises simply because there is no universal system of unique identifiers, but it might also happen when entities are viewed from different domains. For instance, a bridge can be viewed as a transportation resource, a maintenance responsibility, or a target.

When we merge the entities of a coreference, there are two non-obvious but important requirements. First, merging should be reversible and non-destructive, and second we must maintain temporal perspective. To accomplish this, we first "retire" the original proxy created after the first import, which simply means that we delete the single `:finishedBy` statement and add a new one so that the proxy's transaction time is a closed interval:

```
:Proxy1 a :Individual ;
  :hasPrimitive :Person1 ;
  :usesValue [ :Person1 :name "Robert Jones" ] ,
             [ :Person1 :ssn "123-45-6789" ] ;
  :temporalIndex :TmpIdx1 .
:TmpIdx1 a :KbProperInterval ;
  :startedBy :Time1 ;
  :finishedBy :Time2 .
:Time1 a :DateTimeInterval ; :xsdDateTime "2009-08-17T00:00:00" .
:Time2 a :DateTimeInterval ; :xsdDateTime "2009-08-17T23:59:59" .
```

[1] http://www.w3.org/TR/owl-time/

[2] http://parliament.semwebcentral.org/

Then we add a second proxy like so:

```
:Proxy2 a :Merge ;
  :hasPrimitive :Person1, :Person2 ;
  :usesValue [ :Person1 :name "Robert Jones" ] ,
             [ :Person2 :name "Bob Jones" ] ,
             [ :Person1 :ssn "123-45-6789" ] ;
  :temporalIndex [ "2009-08-18T00:00:00" .. "9999-12-31T23:59:59" ] .
```

Here we have abbreviated the `:temporalIndex` for brevity. This proxy is of type `:Merge`, the other subclass of `:Proxy`, and has two primitives, namely both of the `:Person` entities imported so far. The `:usesValue` statements call out both of the names and one of the SSNs. (The other SSN is left out because it has the same value.)

Both `:Proxy1` and `:Proxy2` exist in the triple store at this point, and they have disjoint time intervals. This allows us to choose the appropriate proxy for any given point in time and then look up the corresponding state of the associated entity. Prior to August 17, 2009, there is no proxy and so this person is unknown. During August 17, 2009, the first proxy calls out just one name, and after that day the second proxy calls out both names. In addition, the proxy has effectively merged the coreference without changing the original two entities.

Now suppose that a third import from the original data source happens at 9:35:20 Zulu time on August 18, 2009, and that this re-imports the same "Robert Jones" record that we saw in our first import. However, suppose that in the interim the SSN was changed to correct a typo:

```
:Person1 a :Person ;
  :name "Robert Jones" ;
  :ssn "123-45-6789" , "123-45-6798" .
```

Importing the same record from the same database results in the same `:Person1` entity, since the import process forms the URI from the primary key of the record, but it creates a new `:ssn` property value, such that `:Person1` now has two SSN values associated with it. One comes from the third import itself, and the other is left over from the first import. Naturally, we now need to expire the second proxy and create new ones:

```
:Proxy2 a :Merge ;
  :hasPrimitive :Person1, :Person2 ;
  :usesValue [ :Person1 :name "Robert Jones" ] ,
             [ :Person2 :name "Bob Jones" ] ,
             [ :Person1 :ssn "123-45-6789" ] ;
  :temporalIndex [ "2009-08-18T00:00:00" .. "2009-08-18T09:35:19" ] .

:Proxy3 a :Individual ;
  :hasPrimitive :Person1 ;
  :usesValue [ :Person1 :name "Robert Jones" ] ,
             [ :Person1 :ssn "123-45-6798" ] ;
  :temporalIndex [ "2009-08-18T09:35:20" .. "9999-12-31T23:59:59" ] .
```

```
:Proxy4 a :Individual ;
  :hasPrimitive :Person2 ;
  :usesValue [ :Person2 :name "Bob Jones" ] ,
             [ :Person2 :ssn "123-45-6789" ] ;
  :temporalIndex [ "2009-08-18T09:35:20" .. "9999-12-31T23:59:59" ] .
```

There are two new proxies because the new SSN indicates that these two entities most likely do not represent the same person after all. The proxy for `:Person1` has `:usesValue` properties for the name and for the new `:ssn`, but not for the old `:ssn` value. Thus just before the third import, `:Proxy2` is valid and we see a single entity with two names and SSN, but just after the import we see two distinct entities with different names and SSNs.

Table 1. Summary timeline for example scenario illustrating knowledge store evolution

Date and Time	Actions
2009-08-17 00:00:00	1. `:Person1` added to the KB 2. `:Proxy1` created for `:Person1` and added to the KB
2009-08-18 00:00:00	1. `:Person2` added to the KB 2. `:Person2` discovered to be same person as `:Person1` 3. `:Proxy2` created to merge `:Person1` and `:Person2` 4. `:Proxy2` added to the KB 5. `:Proxy1` retired
2009-08-18 09:35:20	1. `:Person1` re-imported with a different SSN value 2. `:Person1` and `:Person2` un-merged by Analyst 3. `:Proxy3` created for `:Person1` going forward 4. `:Proxy4` created for `:Person2` going forward 5. `:Proxy2` retired

3 Query Rewriting

Writing time-sensitive queries according to the knowledge representation scheme can be a complex, error-prone, and tedious chore. To alleviate the burden of composing temporally-annotated queries, a query rewriting service was developed. This service automatically transforms a time-agnostic SPARQL query and a provided time into a new, time-sensitive SPARQL query. The resultant query yields a subset of the results from the initial time-agnostic query that are considered valid for the submitted time.

The rewriting service does not alter the meaning of the original query bindings to completely obscure the existence of proxies (merges and individuals) within the system. Rather, the service leaves the original query bindings in-place and adds variables to the result set to represent entity proxies. This behavior was requested by our customer, as they wanted direct access to the unproxied entities in the query results. The alternative is to alter the meanings of the binding variables to refer to the proxies and not return the primitive entities themselves.

Query rewriting takes place in three distinct phases. First, we make an exact copy of the original query. Second, proxy representations are appended to the original query to match the underlying knowledge representation, appropriately following the structure of the submitted query. Finally, temporal selection information is appended for each proxy added during the second step.

To demonstrate the query rewriting service, consider this example:

```
SELECT DISTINCT ?person ?name
WHERE {
   ?person a :Person ; :ssn "123-45-6789" .
   OPTIONAL { ?person :name ?name . }
}
```

When submitted with the time `2007-08-17T12:00:00` to the query rewriting service, the resulting time-sensitive query is:

```
SELECT DISTINCT ?person_proxy ?person ?name
WHERE {
   ?person a :Person ; :ssn "123-45-6789" .
   ?person_proxy a :Proxy ;
      :hasPrimitive ?person .
      :usesValue [ rdf:predicate :ssn ;
                   rdf:object "123-45-6789" ] .
   OPTIONAL {
      ?person :name ?name .
      ?person_proxy :usesValue [ rdf:predicate :name ;
                                 rdf:object ?name ] .
   }

   ?person_proxy :temporalIndex ?interval1 .
   ?interval1 a :ProperInterval ;
       :intervalContains [ a :DateTimeInterval ;
                :xsdDateTime "2009-08-17T12:00:00Z"^^xsd:dateTime ] .
}
```

The primary reason for copying the original query exactly is that it improves the baseline query performance of the modified translated query by providing optimization hints to our system's query optimizer. Our knowledge representation scheme relies heavily on statement reification to encode temporal validity. During development, we discovered that our query optimizer is easily confused by reification, often producing inefficient query clause orderings that result in time-prohibitive query executions. Leaving the original clauses in place essentially enables our optimizer to ignore the reified statements without altering the validity of the query results. Retaining the original query structure also improves readability, which can be invaluable for system development and testing.

Proxy expansion closely mirrors the knowledge representation scheme for temporal annotation, with some noteworthy exceptions. Type clauses were ignored in the proxy expansion step, as type information is considered invariant. In transforming query clauses to match the knowledge representation scheme, individual property clauses of a given entity are expanded to match the `:usesValue` construction of the knowledge representation scheme. However, the `rdf:subject` component of statement reification is dropped. This enables multiple underlying primitive entities to provide clause matches when the proxied entity is a Merge proxy in the knowledge store.

To add the proxy representation to the original query automatically, the query rewriting system relies upon the underlying domain ontology for hints about which query variables (and clauses) refer to entities (and properties) that require the introduction of temporal sensitivity. Proxy representations are only added for each entity that appears as the subject in a triple clause that involves time-sensitive information. In the application domain, certain classes of entities, such as countries, are considered non-varying entities. In these cases, no additional proxy representation is required. In ambiguous cases (i.e., a clause consisting solely of subject, predicate, and object variables), a clause is considered time-sensitive and treated appropriately. When an entity is proxied in this phase of query rewriting, a new proxy variable representing that entity is added to the variable bindings set for the query.

Care is taken when adding the proxy representation so as to not alter the underlying query logic. Expansion exactly matches the original structure of the query, respecting the original query block scoping for each entity and clause. For example, if an entity is only referred to in an `Optional` block of a query, the proxy expansion for that entity will only appear in the that `Optional` block. In this manner, the original query logic is left intact.

The temporal selection block is appended to the query in the topmost query block where the entity is proxied, respecting the original query logic as with proxy expansion. The temporal block's structure is dictated by the knowledge store's temporal index processor and incorporates the submitted time.

4 Example Scenario Query Results

To illustrate the impact of using our knowledge representation scheme and query rewriting service, let us briefly consider the example first presented in Section 2. Submitting the example query from Section 3 to the query rewriting service with three different time instants and then issuing those queries to the underlying knowledge store produces the results depicted in Table 2.

Note that, as mentioned in Section 3, the results of the rewritten query leave the original query bindings in-place and add proxy variables to capture the proxied state of an entity. This means that when considering the results of issuing the original query with the date-time of `2009-08-18T09:00:00` (the second row in the results table), the following interpretation is the correct one: There exists one `:Person` in the knowledge store with an SSN of `123-45-6789`. That person has two known names ("Robert Jones", "Bob Jones") and represents the merge of two primitives that were considered, for the given time, to be references to the same real-world entity (represented by `:Proxy2`).

Table 2. Query results for the example scenario with different provided times

Provided Time	Query Results		
	?person_proxy	?person	?name
2009-08-17 12:00:00	`:Proxy1`	`:Person1`	"Robert Jones"
2009-08-18 09:00:00	`:Proxy2`	`:Person1`	"Robert Jones"
	`:Proxy2`	`:Person2`	"Bob Jones"
2009-08-18 09:40:23	`:Proxy4`	`:Person2`	"Bob Jones"

5 Conclusion

As evidenced by the scenario presented in Section 4, the system allows users to consider previous valid states of the knowledge store based on times of interest. This feature enables analysts to explore previous analyses in the context of what was known *then,* rather than what is known *now.*

Our system allows for evolution of knowledge while preserving all previous states of a knowledge store for subsequent review and investigation. Additionally, our knowledge representation scheme provides the additional benefit of non-destructive, reversible coreference resolution. These features are essential for conducting analysis in real-world, dynamically-evolving data environments.

References

1. Allen, J.F.: Towards a general theory of action and time. Artificial Intelligence 23(2), 123–154 (1984), http://www.sciencedirect.com/science/article/B6TYF-4811T47-4R/2/27f611303bc842936faa7f168fdcb9da
2. Kolas, D., Emmons, I., Dean, M.: Efficient Linked-List RDF Indexing in Parliament. In: Proceedings of the Fifth International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009). Lecture Notes in Computer Science, vol. 5823, pp. 17–32. Springer, Washington, DC (October 2009), http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-517/ssws09-paper2.pdf
3. McCarthy, J., Hayes, P.J.: Some Philosophical Problems from the Standpoint of Artificial Intelligence. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence 4, pp. 463–502. Edinburgh University Press (1969), http://www-formal.stanford.edu/jmc/mcchay69.html, reprinted in McC90
4. Özsoyoglu, G., Snodgrass, R.T.: Temporal and Real-Time Databases: A Survey. IEEE Transactions on Knowledge and Data Engineering 7(4), 513–532 (August 1995), http://www.cs.arizona.edu/ rts/pubs/TKDEAug95.pdf
5. Smith, B., Grenon, P.: Basic Formal Ontology (BFO) (June 2010), http://www.ifomis.org/bfo