Development Call 20120105

Notes on improving full lifecycle data management in VIVO

UF has been tasked with providing automated and continual ingests from Peoplesoft and grants databases; last fall they encountered a problem when a bad harvest that had created a lot of duplicates was not discovered until about 3 days after it happened. They had to try and figure out how to restore from that point in time.

The question became, "What are the easy ways to figure out what manual edits had happened since that time?" – it's almost impossible to know when any particular RDF statement was added to the database, or whether it has been manually edited since. At UF they keep track in the Harvester of all the additions and retractions so they can roll back and roll forward again. This is not perfect – it can be very cumbersome and nerve-wracking. With the bad harvest, it turned out they were missing one of the additions and retractions file for one day, so they couldn't go back past that.

On traditional systems, there would typically be additional data fields such as the last modified date and time and the id of the responsible user or process added to each record in the database that could be tapped programmatically to make intelligent decisions based on that information. In thinking about traditional approaches, the UF team made up a wish list of things they'd love to know about a triple to be able perform intelligent actions on any triple as part of an ingest process. Some of these are:

- CreatedBy = What user or person created this triple
- CreateDate = When was this triple first created
- LastModBy = What user or person last modified this triple
- LastModDate = When was this triple last modified
- Public = Am I allowed to show this triple to the public
  wish list items we wanted to talk about

The same problems can be evident in SPARQL queries – if multiple editors have touchec the same data, and if one did bad work, you will likely not know the state of your database – whether this bad data or some garbage left behind by a botched index rebuild or a data migration or an ingest process that went awry.

Data can change

If you keep files of all the triples added via an ingest process, in theory one could simply retract the same triples at a future date and undo all the additions. In practice, there are at least 4 problems with this assumption.

1. Some or all of the triples added may have been edited or deleted by users with rights to do so
2. Since triple stores do not duplicate triples (and quad stores, which VIVO uses, don't duplicate statements with the same graph id, the 4th value), removing all the triples that were added may leave orphan triples behind – you don't know whether any given triple such as an rdf:type statement may be associated with other statements about the same URI
3. An upgrade to VIVO or other ontology changes may have occurred since you added the triples and the data may have been migrated.
4. You may lose track of the triples added or retract the wrong set of triples

At Cornell, we have addressed some of the above issues (having experienced all the above problems) by changing update procedures to issue queries for all the statements about the URI's inserted (say a new set of grants). A SPARQL construct query can create a new file of RDF to retract that matches what now exists. This approach is itself not foolproof, however, and certainly cannot responsibly be run on a fully automated basis.

Alternative choices

Separate audit model

A version of VIVO predating our NIH grant (circa 2007 or 2008) had a listener hooked into an older editing system that deteced every time a triple was added or removed. The listener create a **reified** version of the triple in a separate Jena database store, together with a timestamp, an indication of whether the triple was added or retracted, and the user performing the action. We never made that much use of it except to see who had recently logged in as a self editor; it would certainly have been possible to build more elaborate reports on these additional, reified statements, but we never had the requirement to do so.

The reified triples were stored in a separate store so the application didn't have wade through all the extra logging data when doing a query.

Data separation by graph

Any modern triple store, including SDB, is really a quad store – each collection of triples is a **graph**, and the graph id forms the 4th value in the quad store, together with the subject, predicate, and object of the triple itself.

There can be additional statements about the graph itself, either in that graph or a different one.

VIVO by default uses several different graphs – if you poke around in the ingest tools, we have vivo-kb2 – the main graph. There is also another graph for the ontology that is separate from the data about individuals; there is an inference graph, too.

You can create any number of different graphs – and this can provide many of the advantages of reification with fewer triples to store. Instead of blowing up each triple with 4 additional triples to represent that statement and its subject, predicate, and object, you leverage the graph identifier on each triple and have the option of making additional statements about the graph itself.

With a graph approach to data ingest and removal, you can add additional statements via the graph as a whole such as the date and time of ingest, the source of the data, and the user performing the ingest. When it's time to update, you can retract all the data in the graph and replace it with all the new data from that source in one fell swoop.

If you are adding and removing data from a system of record that you don't want changed in VIVO, such as a human resources database where any changes should be applied to the source rather than a downstream copy of the data, a graph approach may work very well. If data from HR data are loaded in their own graph, the application will not allow that data to be edited by a human user through an interface; you can then just blow away that particular graph and reconstitute it without doing elaborate queries about what you can retract.

VIVO supports this graph approach but not always with the best UI support, such as mouseovers to indicate what graph a piece of content belongs to, or the ability to search for data in a particular graph. Graph identifiers can be used in SPARQL queries, but it will be an advanced user who can use them fully fluidly without additional development work in VIVO. This only works in situations where an ingest process populates a different kind of data from what a human editor will modify, but a VIVO installation may have a relatively high percentage of data that users are not allowed to modify, such as grants.

There would be no problem in then having other statements about the same URIs (in this case people, positions, or departments) in different graphs – phone numbers, address information, etc. The VIVO application can ask for data from any graph in rendering pages. (Note that the MySQL version of the Jena SDB triple store does not efficiently allow querying data from a subset of graphs – just a single graph or the union of all graphs. We don't know whether a Jena SDB triple store in PostgreSQL or Oracle would have the same problem).

A graph identifier could in theory be encoded with information about the source and date of the data associated with the graph.

Graph per user or even triple

The graph approach could be extended to provide one or more graphs per user. In the pathological end case, the graph approach could devolve into a separate graph per triple. This would still likely be more efficient than in storage than the full reified statement approach, but we think it might cause some problems for the Jena SDB triple store we use now.

Discussion of reification vs. graphs

We have thought some about using the graph approach to data even in situations where users will be allowed to edit the data. For example, any data changed by a front-end user could be moved from the HR ingest graph to a global "end user modified" graph, and perhaps copied to an "HR ingest modified" graph to help identify for an auditing system what had been modified. A global "end user modified" graph would not be sufficiently granular when there is a use case requiring identification of the date any triple in that graph had been changed, for example, or by whom.

Graph-based editing gets more difficult to manage as the use cases get more complicated, but may still be the best way to provide functionality such as allowing users to review and approve new data about themselves – in this case, perhaps via global "new data, "user approved," and "user rejected" graphs. The biggest challenge here might well be the UI work to ensure choices make sense to users and are implemented consistently.

Florida suggests that if they could do the reification graph, they could create it with the Harvester and not mess with the triple store. The main changes would be to the Harvester.

But then anything modifying the triple store go would have to go through one central place to modify the reified metadata with any change; this would require a very streamlined, centralized way to modify the triples.

The VIVO application developers want to support making use of data for provenance purposes and will add functionality in 1.5 to that end. It also makes sense to develop a streamlined way for getting access to data so the Harvester can take advantage of the same data access layer that the Vitro-VIVO application itself does vs. talking directly to the VIVO interface.

In theory we could provide hooks so could do the graph based approach or the reified approach – they are orthogonal to each other, and could be implemented largely independently – perhaps graphs to indicate the source of data and reification for any statements modified by end users.

Can we get an architecture that does this?

Approaches via the connectivity to the triple store

For version 1.5 we want to make VIVO much more triple store agnostic. Brian Lowe has a simple working proof of concept that talks to a remote SPARQL endpoint anywhere on the web, we could push this problem down, so VIVO is triple store agnostic, then this problem is in the triple store layer – the virtuoso and enterprisey triple stores might have features to do the triple-by-triple auditing

They will optimize a lot better than we will, built into the fabric of what we are doing

if certain sites want this detailed data because want customized scripting and other, they could pick a triple store that will support that esoteric need – if only UF wants it
but VIVO would have a standardized way of reporting through the app so consistent across installations

Brian Caruso – we need a mutation api to send changes to this triple store
the transactions could include the triple kind of change and a chunk of information about the change metadata
adds, retracts, and here's a bunch of statements that are about this cjhange

keep it goether, roll it back, remove anything done by a particular person

is SPARQL update the way to send changes to data plus additional metadata?

If neither Florida or Cornell could muster the server power or do the database tuning to do as responsive version of VIVO as Amazon's RDS database service, we need to consider the fact that another abstraction layer will slow things down
put in the hooks so could be developed to a greater extent as layer

what does the commercial version of Virtuoso support?

do we know what any of the other semantic projects addressing these issues? Barend Mons

eagle-i has some of this provenance data – how are they doing it?

this is a good topic to ask the TAB about

next week Jim Blake will talk about extension architectures.

Original agenda for the discussion

Adding data to VIVO is not easy – partly because RDF and semantic tools are unfamiliar, but partly because of fundamental data management challenges facing any large system with multiple sources, especially when some data may be edited after ingest and those changes affect subsequent updates.

We have collectively and individually developed many different approaches for adding VIVO data, most notably the Harvester, but also extensions to Google Refine and to VIVO to support using Google Refine, and modifications to the D2R Map tool, improvements to the ingest tools in VIVO itself, and integration of some Harvester functions into VIVO. Joe McEnerney at Cornell has also worked extensively with data from Activity Insight, a faculty reporting tool used by many universities in the U.S., and in the process developed a number of practices for managing initial data matching, incremental updates, procedures for retraction of data that may have been modified in VIVO, and detection of malformed or orphaned data in VIVO.

As VIVO matures at each of our institutions, we are also being asked more questions about reusing data from VIVO in other applications, about reporting using data in VIVO, and tools for archiving, removing, or exporting what can be very large amounts of data. How can we address these challenges appropriately?

Questions from the UF Harvester team

In our discussions of ingesting Person data from People Soft we have a wish list of things we'd love to know about a triple to allow us to preform intelligent actions on any triple as part of an ingest process. Some of these are:

- CreatedBy = What user or person created this triple
- CreateDate = When was this triple first created
- LastModBy = What user or person last modified this triple
- LastModDate = When was this triple last modified
- Public = Am I allowed to show this triple to the public

Other questions to address

As time permits this week, and for future meetings

1. examples of current best practices for recurring ingest processes
2. fundamentals – strategies for when data can be replaced and updated en masse vs. updating that must allow for the possibility of end-user or curator editing
3. how to segregate data by Jena graph using SDB, and VIVO's current limitations and foibles in this regard
4. how to establish a baseline for each different data source
5. what has to be archived prior to, during, or after each successive update – how much can be done in bulk vs. statement-by-statement
6. what an incremental ingest process typically produces: RDF to add and RDF to remove
7. pitfalls of removing RDF data (it may have been introduced via other processes, and RDF does not accumulate multiples of the same triples – if you remove it, it's gone, even if two separate processes had independently added the same statements)
8. techniques for rolling back additions of RDF via SPARQL CONSTRUCT
9. what can be accomplished by logging actions and/or triples
10. what would be reasonable near-term goals for the Harvester
11. what would be reasonable near-term goals for VIVO (1.5)
12. what use case, requirements gathering, or design work needs to be done and who can participate

VIVO v1.5 release planning