# An XSLT Based RDF Ingest Example

Author: Joseph R. Mc Enerney (jrm424@cornell.edu)

The purpose of this example is to illustrate XSLT based techniques that have been used successfully to ingest data from more than a dozen sources into VIVO at Cornell. Instead of a simplified ‘toy’ example, the source data used will display many of the data quality problems often found in practice. The goal is to transform this source data into RDF that conforms to a specific data model and can be loaded into a VIVO instance. The example is based on educational credentials and the central objects in the RDF data model are instances of the class **vivo:EducationalTraining**. In addition, we want to prevent duplication of Person and Organization RDF. Experience has taught us that this XSLT transform methodology performs well in terms of processing time and is scalable to tens of thousands of source data records.

### Assumptions

This example expects the reader to have some familiarity with the following notions and technologies:

1. VIVO
2. XML, RDF
3. XSLT 2.0 (Saxon 9.x he), XPATH 2.0
4. SPARQL 1.1
5. Unix/Linux Bash Shell Command Line

### The Source Data

Suppose that a query is run against a RDBMS to get our source data. Example rows from the XML result set source file are shown below in Figure 1. The entire file can be found in the result set file **EduRS.xml** found in the **example/source** directory of the online example. Also see Appendix C. There is frequently more than one row for a given person and the data may or may not be complete as indicated in the Figure 1 below.

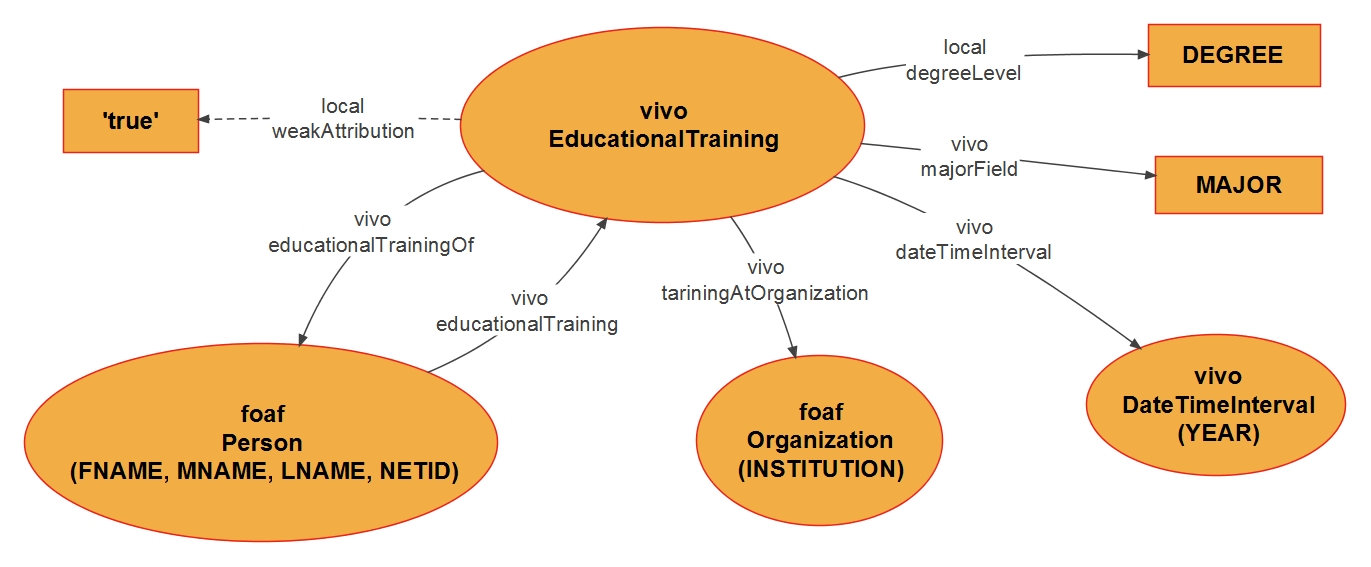
Notice also, that there is a ‘row id’ number (guaranteed to be unique) associated with each row and a unique **NETID** associated with most rows (see the result set file description for a full list of source abnormalities in the Appendix B). We are interested only in rows that have a first name (**FNAME**), a last name (**LNAME**), a degree level (**DEGREE**), a major (**MAJOR**), a year (**YEAR**) and an institution (**INSTITUTION**). We will try to deal appropriately with missing netids (**NETID**), character case discrepancies, and extra white space. We will ignore the **MINOR** and **LAST\_UPDATED** data. Part of the process will involve result set row rejection when there is no sense in its inclusion. For example, notice that the second record has no listed major. Clearly this row is a good candidate for rejection just as a missing institution would also be.

Row rejection could be performed as part of our SQL query. This is also true of correcting rows with case and whitespace issues. However, in an institutional environment with complex data stewardship, we may not be in control of the query and just have to deal with whatever result set that we are given. In our work we have used this as an opportunity to pass the rows that we reject back to the managers of the original data source for repair.



Result Set Fragment - Figure 1

The following figure illustrates the RDF data model we intend to populate using the source data from the tags in the result set rows. Note that the predicate **local:weakAttribution** is conditional.



RDF Data Model - Figure 2

In Figure 2 the rectangles represent data; the oval shapes represent object classes and the labeled arrows predicates. A namespace abbreviation is also given e.g. **vivo** or **foaf**. The string **local:degreeLevel,** like **local:weakAttribution,**is the name of a yet to be defined predicate – more later. The central object will be of type **vivo:EducationalTraining** as mentioned above.

### The Accumulator Classes

Special attention must be given to the **foaf:Person** and the **foaf:Organiztion** classes. As time goes by and data sets are ingested into VIVO, instances of people and organizations will accumulate at a dramatic rate. We want there to be only one URI for an instance of a given person (or organization). This means that we will need to check for pre-existing people and organizations to ensure that we don’t accidently create duplicates as we construct the ingest RDF. To this end we will employ two XML files **Per0.xml** and **Org0.xml** that contain the currently known people and organizations in our VIVO instance. In practice these files are created by SPARQL queries (See Appendix E). For purposes of this example, we have prepared greatly reduced sample files. Figures 3 and 4 illustrate the entries in these files. For **Per0.xml** we include name parts, label, netid and a URI. A netid is a unique string assigned to each person at Cornell to serve as a public identifier. Any uniquely assigned string by another name could be used instead. For **Org0.xml** we include the name of the organization and the assigned URI.

These two files will be fed into the XSLTs to supply sets of person and organization objects to match. The nature of the matching methods used will be described in significant detail later. Note that our process must also create RDF for any new persons or organizations that we encounter during the ingest process. This accumulator RDF must be asserted before the next round of ingest so that duplicates are not generated by other unrelated ingest processes that use similar methodologies.

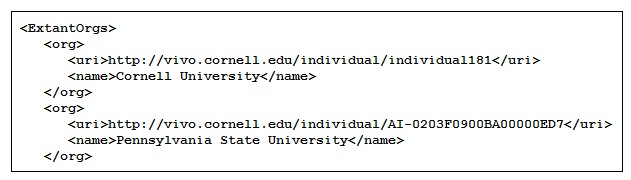
Notice that the second **<person>** element has no netid. This situation can arise when publication data which lists co-authors from other institutions, who have no netid, is ingested. It can also arise when a person’s netid is simply unknown or forgotten at the time **foaf:Person** data is being entered.



Per0.xml Fragment - Figure 3

It is expected that **Per0.xml** will be maintained for a given Vivo so that, ignoring case and whitespace differences:

* Duplicate **<person>** records are removed.
* A given netid will not be found in more than one **<person>** entry with distinct URIs.
* Among **<person>** entries with no netid, a given set of name parts will not be found in two or more <person> entries with distinct URIs.



Org0.xml Fragment - Figure 4

### The Process

The translation of the query result set into RDF proceeds along the following steps:

1. Gather the data items of interest and reject any malformed records that can’t reasonably be dealt with. During this step we will populate an intermediate XML called **ED0.xml** with our selections and try to assign URIs to the people and organizations that we encounter. The unresolved people and organizations are those we cannot match to an existing **foaf:Person** or a **foaf:Organization** so the corresponding URI elements are left blank.
2. Count the unresolved people (URP) and unresolved organizations (URO).
3. Make up a URI for each URP and one for each URO. This must be done in such a way that will guarantee uniqueness in your VIVO instance. In this example, these URIs will be provided. These files are called **EX-URP-UNOs.xml** and **EX-URO-UNOs.xml** respectively.
4. Generate XML files in the same form of **Per0.xml** and **Org0.xml** using the URPs and UROs found in **ED0.xml**. These files are needed to fill in the missing URIs. We call these **NewPers.xml** and **NewOrgs.xml** respectively.
5. Fill in the missing organization URIs by transforming **ED0.xml** into **ED1.xml** using **NewOrgs.xml** as a source of Orgs to match against. Next transform **ED1.xml** into **ED2.xml** filling in the URPs. If either count is 0 then one or both of these transforms is unnecessary.
6. For each source row create the RDF for an instance of type **vivo:EducationalTraining** according to the Education Model shown in Figure 2. This will be straightforward since all unresolved issues are now resolved. In this step we create the file **ED.rdf**.
7. Create any needed RDF for **NewPers.xml** and **NewOrgs.xml** and put it in **NewPers.rdf** and **NewOrgs.rdf** respectively.
8. Declare the **local** ontology predicates **local:degreeLevel** and **local:weakAttribution** and then apply all the new RDF to VIVO.

Next we will elaborate each of these steps and describe key sections of each XSLT.

### Gather

This step is performed by the **gather.xsl** transform which can be found in its entirety the ‘**example/xslt**’ folder. Figure 5 shows the first portion of this XSLT. We will now comment on the highlighted sections. In what follows, we will use [FnHm] to denote Figure n, Highlight m.



gather.xsl Fragment 1 - Figure 5

* [F5H0] This is the usual XSLT boiler plate where namespaces and short hand prefixes are declared. In this example, the **vfx** prefix will be used to identify functions that were created to support the ingest process.
* [F5H1] Two required parameters that are file system paths that point to the Accumulator files.
* [F5H2] The usual output declaration – including normalization of character sets.
* [F5H3] A variable is declared here to define the namespace we use for individual URIs at Cornell. This will be used as a prefix for new URI local names. You should change this variable to reflect your Vivo instance in **gather.xsl**
* [F5H4] A variable used to append a newline character to the output where needed. This is mainly for adjusting output for readability.
* [F5H5-6] Two variable declarations that provide a means to access the pre-existing people and organization data from **Per0.xml** and **Org0.xml**.

Next we consider the highlights of the second half of the **gather.xsl** file. This part of the code handles filtering of the source, construction of the output xml and resolving as many URIs as possible by comparing name information against the elements in **Per0.xml** in the case of people and against **Org0.xml** in the case of organizations. Figure 6 shows more of **gather.xsl**.

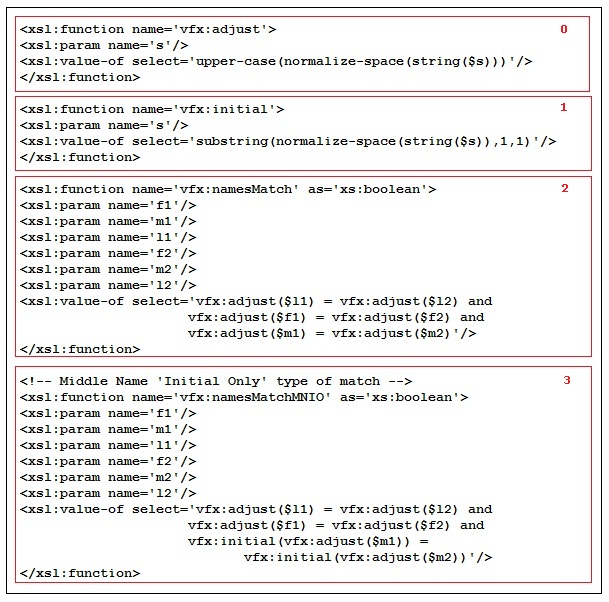


gather.xsl Fragment 2 - Figure 6

* [F6H0] Establish which source rows will pass the rejection criteria filter and start an **EduRecord**.
* [F6H1] Find any and all matching school URIs among the known organizations. Notice that the variable **school** contains a space normalized copy of the **INSTITUTION** and is further shifted to uppercase before comparison with each adjusted organization **name**. The **vfx:adjust** function, shown in Figure 7, applies the standard XPATH functions **normalize-space** and **upper-case**. The variable **schoolUri** is now a sequence of 0 or more matching organization URIs.
* [F6H2] Collect and create the required XML elements, applying **normalize-space** to fix any white space issues in the source data. You may choose to add other normalizations at this point.
* [F6H3] If **schoolUri** contains a URI then use the first one; otherwise leave **edSchoolUri** empty. If **schoolUri** has more than one term then there are duplicate entries in **Org0.xml**. This is not the case for this example. However steps must be taken to prevent duplicates by properly maintaining organization triples in Vivo.
* [F6H4] Since we may not find a URI for the school or person, we include that as an empty element along with the name parts and netid of the person who received the degree in the output XML for downstream remediation.
* [F6H5] In this step we look for a matching person by calling the name matching function **vfx:findMatchingPeople** which will return a URI or the empty string. We will describe this function shortly. Several versions of this function are included in the source code so that the reader can experiment.
* [F6H6] This shows the end of the **gather.xsl** transform with the inclusion of a file of auxiliary functions that contains the definitions of **vfx:adjust** and **vfx:findMatchingPeople** and other functions. The function **vfx:findMatchingPeopleI** , not shown here, is much stricter in terms of what can match. It is included in the source code files so that the reader can compare it to the superior alternative **vfx:findMatchingPeople**.

The name comparison functions are built up from XPATH functions and are shown in the next figure. These functions can be found in **example/xslt/auxfuncs.xsl**.

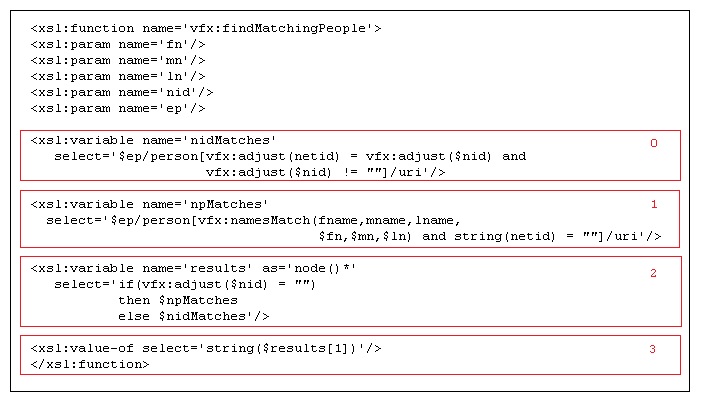
* [F7H0] The **vfx:adjust** function is meant to take an argument convert it to a string, normalize white space and uppercase all characters. Note that the call to ‘**string**()’ ensures that if the argument is a tag that does not appear in the source XML the result will be an empty string. Note also that white space normalization produces a string with no leading or trailing white space and all runs of white space characters in the argument will be reduced to a single space in the result. This function could be augmented to remove other extraneous characters that might appear in source data. For example, the period (.) character could be removed. The author has seen parentheses, colons, semicolons and numbers included in name parts. This is a good function to experiment with.
* [F7H1] The **vfx:initial** function is meant to return the first non-white space character in the argument.
* [F7H2] The **vfx:namesMatch** function compares two sets of name parts after **vfx:adjust** is applied. Comparing this way is an equivalence relation. It is reflexive, symmetric and transitive. This is the match function actually used in our example. The next function is included to show what can happen with heuristic matching methods.
* [F7H3] The **vfx:namesMatchMNIO** function is compares two sets of name parts but only requires that the middle names match on the first initial. This is riskier but also more likely to find a match. Consider comparing “Jon J Smyth” with “Jon James Smyth”. Clearly there is a risk in declaring these to be equivalent. On the other hand, by treating this name pair as different we run the risk of creating two distinct **foaf:Person**s where one would suffice. Use of this equivalence function is tempting. Although this relation is reflexive, symmetric and transitive, it is too coarse. For example “Jon J Smyth” matches “Jon James Smyth” and “Jon Joseph Smyth” but “Jon James Smyth” and “Jon Joseph Smyth” are clearly not equivalent to a human reader but they are equivalent using this function.



auxfuncs.xsl Fragment 1 - Figure 7

The person name finding function illustrated in Figure 8 is in some ways just an elaborate version of the simpler organization match XPATH expression described in Figure 6 Highlight 1. Notice that the name part matching function does not contain the heuristic middle initial weakness described above.

* [F8H0] The variable **nidMatches** is the list of all matching person nodes from **Per0.xml** whose **netid**s match the nonempty netid (**nid**) of the person for whom we are searching. Note that there may be no matches. This is a normal XPATH predicate expression like that done for organizations.
* [F8H1] The variable **npMatches** is the list of all matching person nodes with no netid whose name parts match those of the person for whom we are searching. There may be no matches in this case as well.
* [F8H2] The variable **results** will contain the name part match list **npMatches** when the search subject has no netid. Otherwise it will contain the netid match list **nidMatches**. Use of the adjust function here deals with the case where a source NETID only contains whitespace.
* [F8H3] The function returns the first URI on the results list or the empty string. If there is more than one possible result then person and organization triples have not been well maintained in Vivo or there are different but equivalent names with the same URI.

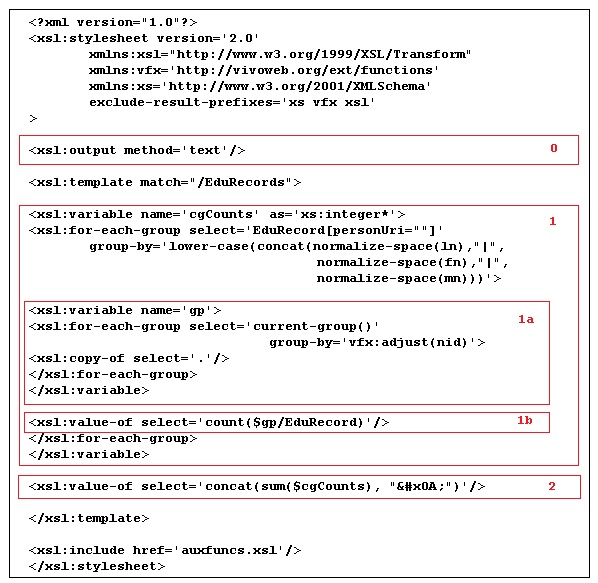


auxfuncs.xsl Fragment 2 - Figure 8

### Count

The next phase is to count the unresolved person (URPs) matches and the unresolved organizations (UROs). To do this we apply the transforms **countURPs.xsl** and **countUROs.xsl**. Figure 9 shows the more complex URP case. The URO case is much simpler since organizations are compared based on a single string.

* [F9H0 Notice that this transform outputs text not XML.
* [F9H1] This code builds up a variable **gcCount** that contains a sequence of integers each of which represents the number of *distinct* names/netids in each group formed by name parts. This will also include names without **NETID**s in the source. The grouping order used here does not handle the case of name variants with the same netid in the *same result set*. Such cases can arise because of change of marital status which is not easily distinguished, in an automated way, from other cases such as a mistyped netid. Any such problems can be fixed by post processing to correct any misattributions based on common netid. Changing the grouping order (i.e. by netid then name parts) also works but with additional coding complexity. Note that we are only interested in **EduRecord**s where the **personURI** is empty – those are the URPs in **ED0.xml**.
  + [F9H1a] Collect all nodes in the current group formed by grouping by name parts and then group them by **nid** the tag we use in **ED0.xml** to represent a source **NETID**. The variable **gp** refers to this sequence of nodes.
  + [F9H1b] Count the nodes in each sub-group and append the count to the **cgCounts** sequence.
* [F9H2] Output the grand total by summing the sequence **cgCounts**. This is the number of distinct URPs found in **ED0.xml**.



count.xsl - Figure 9

In our example, we find 11 unresolved people URIs and 5 unresolved organization URIs. Next we will construct enough URIs to fill in the URPs and UROs.

### Make URIs

This step produces a pair of XML files that are used to provide unique numbers (UNOs) from which we can construct URIs for the URPs and UROs just counted. The unique numbers were generated by a Perl utility, developed by the author, called **nuno** (included in the **example/bin** directory) that uses an 8 character hexadecimal counter, an Inode number, a Unix user id and a token to ensure uniqueness. The counter is stored in a UNIX file and so can be protected by UNIX security. The Inode number ensures that if the file is moved or copied the unique number sequence is different from the original sequence. The UNIX user id is the ID of the utility caller (not the owner of the counter file) this ensures that if there are multiple users of the utility on the same counter file then each user will obtain sequence different from all other users. The token mentioned above is useful when trying to SPARQL query for individuals from a specific source. In our case this is ‘EX’ standing for ‘Example’.

The calls to the **nuno** utility are shown next.

* **bin/nuno -t EX- -X -n 11 > EX-URP-UNOs.xml**
* **bin/nuno -t EX- -X -n 5 > EX-URO-UNOs.xml**

An example output file the first command, **EX-URP-UNOs.xml** is shown next (the second file is similar).

**<?xml version='1.0'?>**

**<Mapping>**

**<map n='1' nuno='EX-0203EF6807A00000000'/>**

**<map n='2' nuno='EX-0203EF6807A00000001'/>**

**<map n='3' nuno='EX-0203EF6807A00000002'/>**

**<map n='4' nuno='EX-0203EF6807A00000003'/>**

**…**

**</Mapping>**

The attribute ‘n’ will be used to select which URI is assigned when creating new **foaf:Persons** as described in the next section. In our example, the following URIs will be assigned to the URPs.

http://vivo.cornell.edu/individual/EX-0203EF6807A00000000

http://vivo.cornell.edu/individual/EX-0203EF6807A00000001

http://vivo.cornell.edu/individual/EX-0203EF6807A00000002

http://vivo.cornell.edu/individual/EX-0203EF6807A00000003 ...

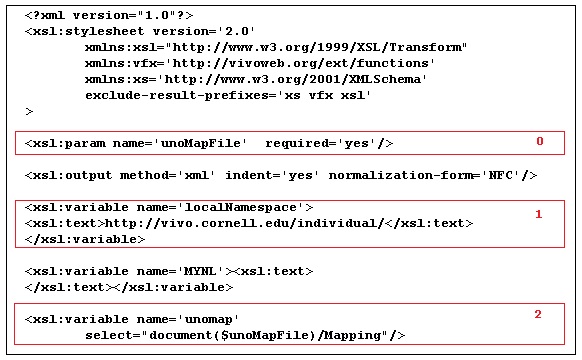
It is worth noting that your data source may guarantee that a particular data element is *always present* and is *always uniquely associated with a person*. If this is the case, then that element would be appropriate for use as a local name in your URIs provided it also satisfies the URI formation rules. Suppose that we had such guarantees in our example in the case of **NETID**. This would mean that a URI for Arthur R. Fuller (arf72) like that shown next could be assigned during the Gather step.

http://vivo.cornell.edu/individual/arf72

Unfortunately we can’t do this since the **NETID** element might be empty or missing in our source data. While a **NETID** is uniquely associated with a person we are not assured by the source that one will always be present in each record. Hence we will still have to create URIs from scratch.

### Create New Persons and Organizations

The transform for Organizations, **makeUROs.xsl**, is much simpler than that of Persons and so we shall describe **makeURPs.xsl** in this section. As mentioned previously, in the case of Persons, we have name parts and a uniquely assigned, but possibly missing, source **NETID** for matching. We also want that URIs be assigned uniquely even when we have an exact name part match for a set of records but **NETID** is missing in one (or more) records but not in others. Indeed it quite possible for there to be several URPs in our source that have exactly the same name parts, character for character, but different netids. In our example there are at least three (and possibly four) distinct people named “Arthur R Fuller” and multiple **EduRecords** that have to be assigned to the correct person. Misattribution should be rare although it is unavoidable in an automated system when there is no sure way to distinguish between different people. Thus when there is no netid associated with an **EduRecord** we will add a “weak attribution” triple in the RDF that we generate. Our main purpose here is to create a file of new person records, in the **PER0.xml** style, that specifies name parts, **uri** and **netid** if possible. These will be used in a later step to fill in missing URIs in **ED0.xml**.



makeURPs.xsl Fragment 1 - Figure 10

* [F10H0] Declare the parameter that contains the name of the UNO file for Persons.
* [F10H1] Identify the namespace used to fully qualify a URI local name.
* [F10H2] Make the UNO file accessible as a set of nodes.
* The next figure details how **Per0.xml** style person records are created. This part of the transform is the most complex in our example. This is because of the fact that we must index correctly through the set of 11 URIs constructed above as we assign them to the URPs**.**
* [F11H0] This is the same grouping method employed in the Count Step. The variable **cgCounts** refers to a set of counts each of which is the number of distinct people sharing a name.
* [F11H1] The call to the recursive named template **cumulativeSum** produces a sequence containing the 0 based cumulative sums for the counts in **cgCounts**. These are the offsets into the list of 11 URIs that we need to do URI assignment to the new people. The variable **cumulativeCgCounts** contains the cumulative sum sequence. The recursive named template can be found in the Appendix D and source file **makeURPs.xsl**.



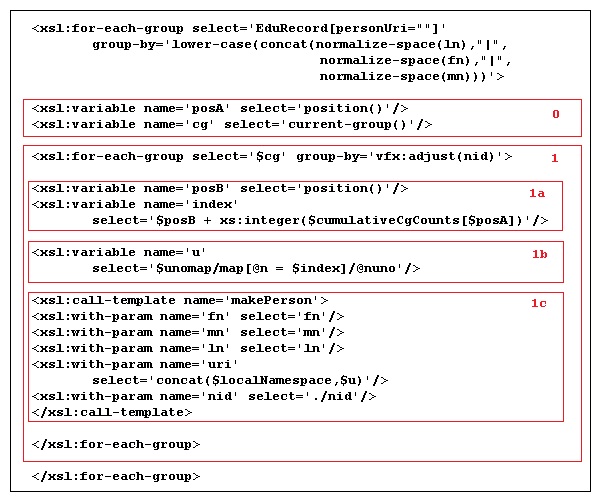
makeURPs.xsl Fragment 2 - Figure 11

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Netid | Count | Offset | Index |
| David Augustus Green | dag065 | 1 | 0 | 1 |
| Don A Horsham | dah3507 | 2 | 1 | 2 |
|  | dah256 |  |  | 3 |
| Denise Hortense Valencia |  | 1 | 3 | 4 |
| Cathrine A Dale | cad2616 | 1 | 4 | 5 |
| Andrea S Killian |  | 1 | 5 | 6 |
| Arthur R Fuller |  | 4 | 6 | 7 |
|  | arf27 |  |  | 8 |
|  | arf33 |  |  | 9 |
|  | arf72 |  |  | 10 |
| Suzy Beltaine | sb016 | 1 | 10 | 11 |

Indexing Details - Figure 12

Figure 12 summarizes the counting, grouping and indexing for our result set. There are 7 names, 11 people and 3 people have no netid.

With these counts and sums in hand we can now iterate through the groups and subgroups to collect names and assign them URIs. Figure 13 displays the XSL code to finish this step. We begin as before by grouping **EduRecord**s with URPs by name parts.



makeURPs.xsl Fragment 3 - Figure 13

* [F13H0] The variable **posA** is a counter that ranges over the 7 name groups and **cg** is the variable that contains the current group.
* [F13H1] This code groups the current group by **nid** (i.e. netid).
  + [F13H1a] The variable **posB** ranges over the number of records in the current group (by netid) of the current group (by name parts) which varies in size (see Figure 12).
  + [F13H1b] The variable **index** combines **posB** and the offsets in the sequence **cumulativeCgCounts** which is then used to choose a URI for assignment to the variable **u**.
  + [F13H1c] A named template, **makePerson**, is called to construct the person XML with all the name parts, netid (if possible) and the fully qualified URI assigned.

The files **NewPers.xml** and **NewOrgs.xml** are created by applying the transforms **makeURPs.xsl** and **makeUROs.xsl** successively.

### Fill in URPs and UROs

This step is actually a replay of the part of the gather XSLT that tries to find an existing match for URIs among the pre-existing **foaf:Person**s and **foaf:Organization**s. The main difference is that we use **NewPers.xml** and **NewOrgs.xml** in place of **Per0.xml** and **Org0.xml**. Figure 14 shows the code for the main XSLT template in **fillURPs.xsl** . The code for organizations is found in **fillUROs.xsl**.



fillURPs.xsl Main Template - Figure 14

* [F14H0] Copy **EduRecord** as a container.
* [F14H1] When the current tag name matches **personUri** and the tag is empty, try to find a match to the name/netid peer tags. A match is a sure thing now because of all of the effort that came before.
* [F14H2] Otherwise just deep copy the peer tag data.
* [F14H3] Close out the outer copy operation.

### Create RDF

At this point we have everything needed to generate the RDF for each **EduRecord**. We will create RDF/XML that describes each as an instance of the class **vivo:EducationalTraining**. This object is identified by a URI as are the educational institution conferring the degree and the recipient. Each instance is further enriched by other data properties. The special **local:weakAttribution** predicate is used when the recipient has no netid. The full transform can be found in **mkEduRDF.xsl**, the main parts are shown in the next two figures.



mkEduRDF.xsl Fragment 1 - Figure 15a

* [F15H0] Pretty printing that produces nicely formatted output.
* [F15H1] Declare the type of each instance.
* [F15H2] Using a **local:degreeLevel** predicate assert the degree level.
* [F15H3] Assert a weak attribution when the **EduRecord** has no netid.
* [F15H4] Assert a label for each instance.
* [F15H5] Declare the major field of study.



mkEduRDF.xsl Fragment 2 - Figure 15b

* [F15H6] Call a named template (found in **auxfuncs.xsl**) to create a **vivo:DateTimeInterval** for the year the degree was conferred.
* [F15H7] Assert the organization RDF for each instance (with pretty-print code).
* [F15H8] Assert the degree recipient RDF (with pretty-print code).
* [F15H9] Close the loop and finish up.

The results of this step can be found in **example/rdf/ED.rdf**.

### Create RDF for New Persons and Organizations

This part of the process is needed to provide RDF that describes the new people and organizations discovered in earlier steps. The transforms are **mkAccPeopleRdf.xsl** and **mkAccOrgRdf.xsl** . These transforms are fairly simple and are applied to the **NewPers.xml** and **NewOrgs.xml** respectively. The output of these transformations can be found in **example/rdf** in the files **NewPers.rdf** and **NewOrgs.rdf**. These RDF files must be asserted using the VIVO user interface so that the data model of Figure 2 can be traversed and so that when **Per0.xml** and **Org0.xml** are reconstructed for the next round of ingest the new people and organizations names together with their URIs are available for matching and assignment. Otherwise duplicates with different URIs are likely to be created.

### Add Predicates and RDF to VIVO

The VIVO user interface should now be used to add the two new predicates to the local ontology *before* adding the RDF files. If this isn’t done first you won’t get another chance until the RDF in **ED.rdf** is retracted.

At this point the **ED.rdf** file can be asserted, followed by the **NewPers.rdf** and **NewOrgs.rdf** files.

### Final Considerations

It is unlikely that you will use this code ‘as is’. Your data source is likely to be very different and possibly much more complex. In fact this is the case for many of the data sources ingested here at Cornell. Consider, for example, a data source of Academic Articles where there are typically several co-authors listed with each article and since many co-authors are not at Cornell, missing netids is a commonplace. However these techniques easily extend to handle such cases.

Until now no mention has been made about how to update information once in VIVO. Two possibilities suggest themselves:

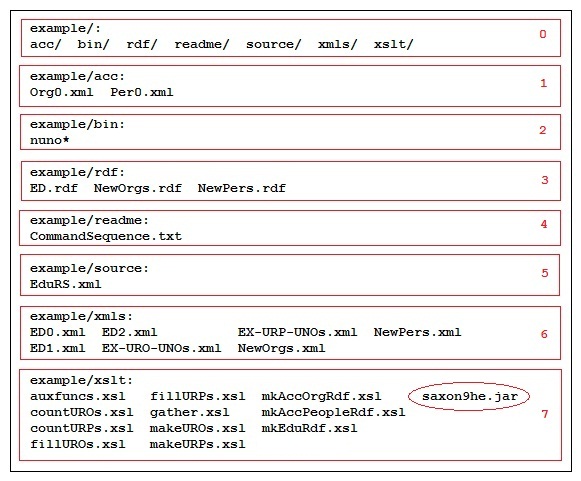
* Wholesale replacement and
* Incremental update

Both have advantages and disadvantages depending on how complex the source is. Wholesale replacement is attractive because all you have to do is retract what was once asserted; build new RDF which is then asserted. If you keep (i.e. don’t retract) the new people and organization RDF then chances are they will be available in **Per0.xml** or **Org0.xml** when needed. On the other hand, wholesale replacement can be very time consuming. To replace the many tens of thousands of Academic Articles in VIVO at Cornell would take several hours just to retract. However, for small sources wholesale replacement is a good choice.

Incremental update involves a differencing process where the source data for one query result set is compared to the data for the next. Differences in terms of additions, changes and deletes are tabulated to determine what must be retracted (deletes and changes) and what must be asserted (changes and additions). The granularity of the differencing process depends on the complexity of the source. Deeply nested XML structures can be tough to compare. Look for ‘last modified dates’ and similar elements, that signal a change, to simplify the problem.

### Appendix A

The following table shows the directory hierarchy of the example.



Directory Layout - Figure 16

* [F16H0] This is a list of the **example** subdirectories.
* [F16H1] This is where the accumulator files are found. **Per0.xml** for people and **Org0.xml** for organizations. In the case of this example, there are only a few entries in each of these files. In *vivo.cornell.edu* there are tens of thousands of people and thousands of organizations. These files are constructed by SPARQL query and XSLT. Since this is really a sideshow, it won’t be covered in detail here (see Appendix E for the SPARQL query and XSLT).
* [F16H2] The Perl script **nuno** is a utility for creating sequences of unique numbers and is used to generate an XML containing local names used to construct URIs for unresolved people and organization URIs. The script is self-describing when invoked as ‘**nuno –u**’ on the bash shell command line.
* [F16H3] This directory is where RDF is placed when produced. There are files for new people and organizations and of course for the **vivo:EducationalTraining** instances (**ED.rdf**).
* [F16H4] The command sequence that produces the RDF is found here.
* [F16H5] This directory contains the result set as an XML in the file **EduRS.xml**.
* [F16H6] This directory contains the intermediate XML files produced along the way to the RDF.
* [F16H7] This directory contains the XSLTs used to transform **EduRS.xml** into **ED.rdf**. Notice that the file **saxon9he.jar** is the transform engine that executes the XSLTs.

### Appendix B

The following figure enumerates the process commands that can be issued from a bash shell. Some of the commands are split across two lines.

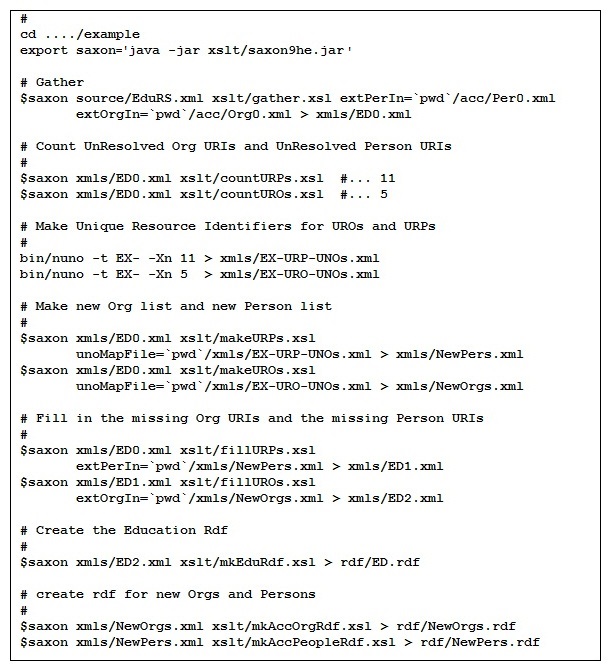
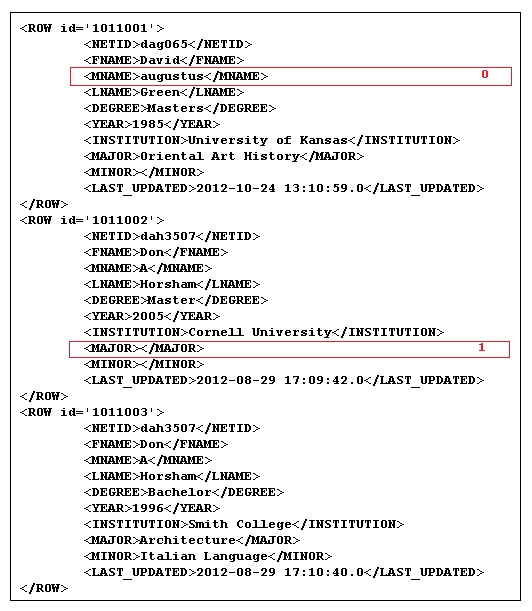


Figure 17

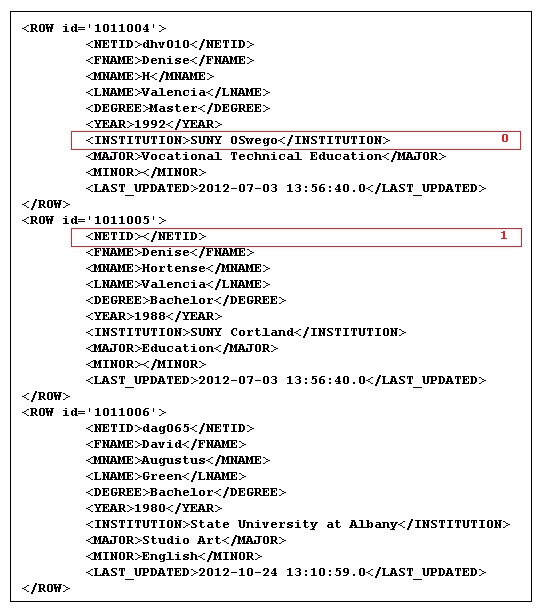
### Appendix C

The next topic concerns the problems in the source data.



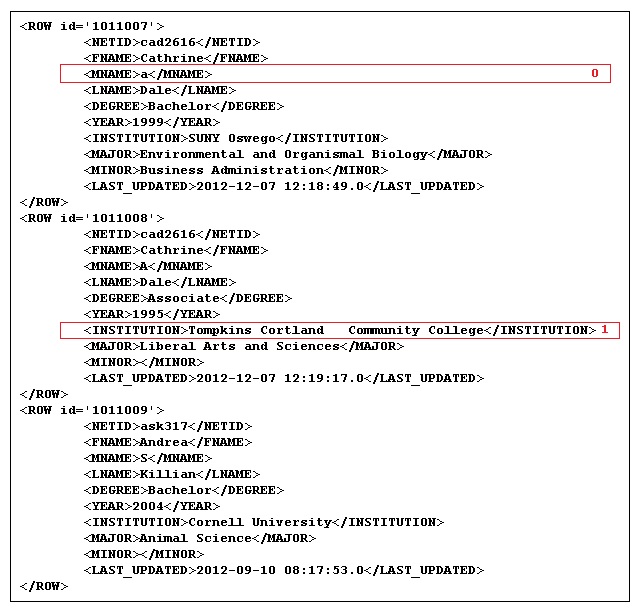
EduRS.xml Fragment 1 - Figure 18

* [F18H0] The middle name is all in lower case. If name parts are all shifted to upper case or all shifted to lower case before comparison then this becomes unimportant.
* [F18H1] The **MAJOR** data field is empty. This is needed to properly label the **EducationalTraining** instance and so this row will be rejected.



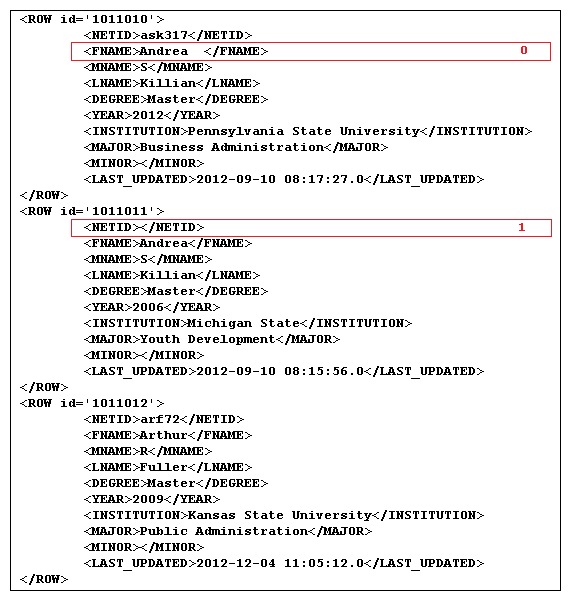
EduRS.xml Fragment 2 - Figure 19

* [F19H0] The **INSTITUTION** field has mixed case characters which will cause a mismatch if character for character equality is required for organizational matching. However, if all characters are shifted to lower case before comparison this will not be a problem.
* [F19H1] The **NETID** field is missing. This causes uncertainty when comparing names. Since it is not uncommon for there to be several distinct people with exactly the same name parts, a token like netid is perfect for disambiguation of people. This is because netid is guaranteed to be unique even when a person’s name has changed due to choice, marriage or divorce. A missing netid weakens the association of a degree record and a person.



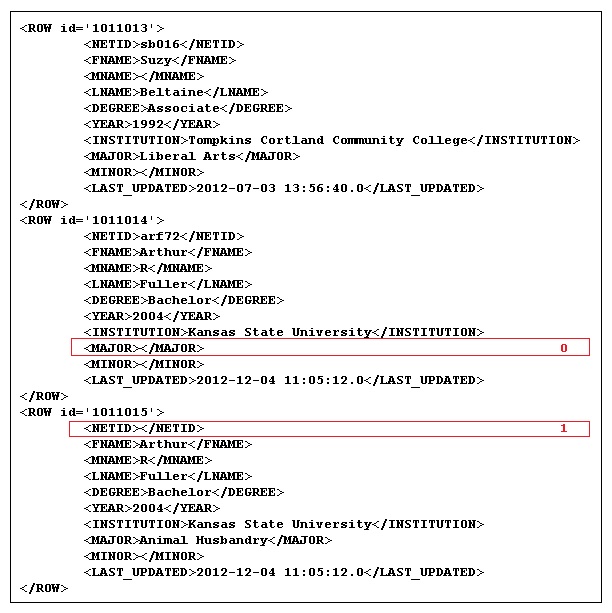
EduRS.xml Fragment 3 - Figure 20

* [F20H0] See Figure 18, highlight 0.
* [F20H1] Extra embedded whitespace. The XPATH function normalize-space deals effectively with this sort of issue.



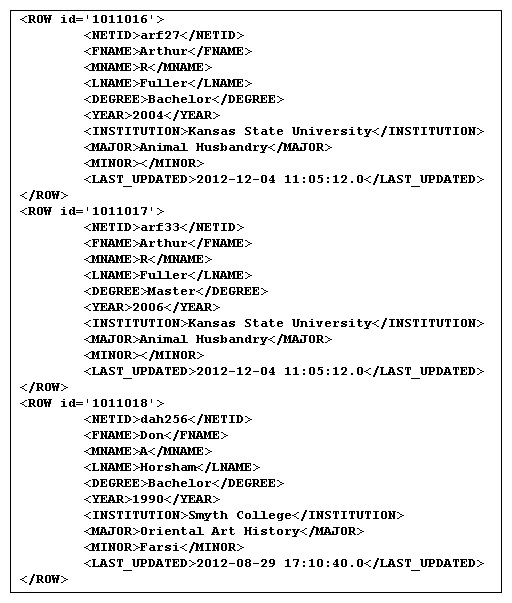
EduRS.xml Fragment 4 - Figure 21

* [F21H0] Trailing whitespace.
* [F21H1] Missing netid. See Figure 19 Highlight 1.



EduRS.xml Fragment 5 - Figure 22

* [F22H0] Same as [F18H1]. This row will also be rejected.
* [F22H1] Same as [F19H1].



EduRS.xml Fragment 6 - Figure 23

### Appendix D

This section is devoted to the recursive template used to create a sequence of partial sums of a sequence of integers. The code for this template is shown in Figure 24.

* [F24H0] The **vals** argument should contain a sequence of integers. In recursive calls, **vals** refers to the tail subsequence formed by dropping the first term.
* [F24H1] The **seq** argument should start as the empty sequence (). In recursive calls, **seq** will refer to the sequence of partial sums.
* [F24H2] The **nxtval** argument starts with an initial value of 0 and so need not be present in the first call. In recursive calls it refers to the sum of its last value and the first term in the **vals** sequence.



makeURPs.xsl Recursive Template - Figure 24

* [F24H3] This test allows the recursion to continue while there are terms left in the sequence **vals** (see [F24H5).
* [F24H4] This is the recursive call where
  + [F24H4a] The XPATH expression is the tail of the **vals** sequence with the first term dropped.
  + [F24H4b] This parameter uses the sequence constructor () to add the value of **nxtval** to the end of the sequence **seq**.
  + [F24H4c] The new value of **nxtval** is the old value plus the first terms of the **vals** sequence.
* [F24H5] This declares the sequence **seq** as the result when **vals** is exhausted (see [F24H3]).

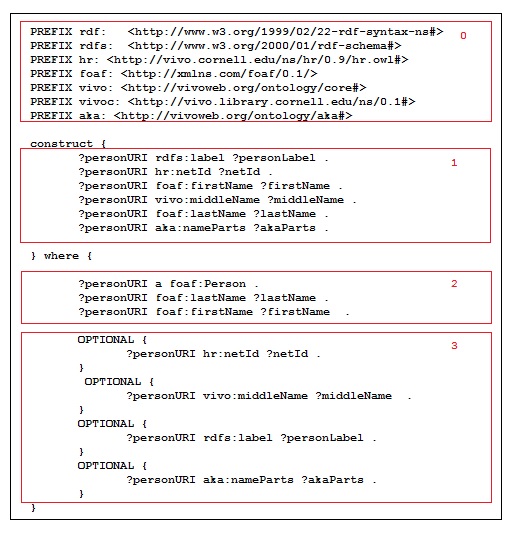
### Appendix E

In this section we consider the process used to create Per0.xml from the **foaf:Person** triples in Vivo. We begin with the **Sparql** query shown in the next figure.

* [F25H0] This is the section where prefixes are declared.
* [F25H1] In the **construct** clause the potential triple set is specified.
* [F25H2] This section of the **where** clause declares the mandatory triples:

1. **personURI** must be a **foaf:Person** and
2. have both a first and last name

* [F25H3] The optional triples are listed here: netid, middle name, label and any **aka:nameParts** triples. These express the list of aliases for the **personURI**.

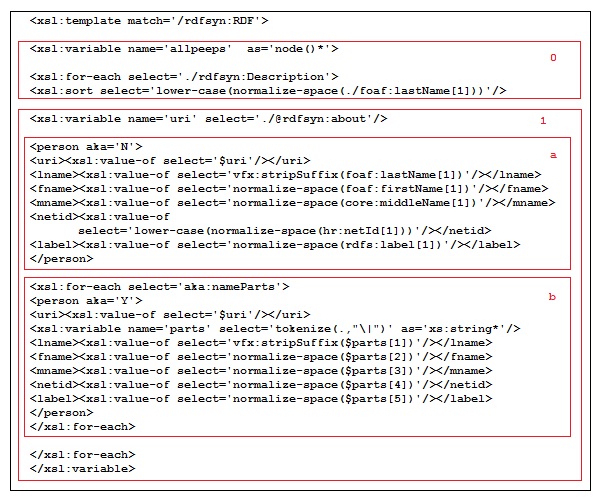


The Sparql Query for Per0.xml Figure 25

This query is invoked with a selection of RDF/XML as the result set format. This XML is transformed into the form used in **Per0.xml** by the transform shown in the next two figures.

* [F26H0] Set up to collect a node list of person information in the **allpeeps** variable.
* [F26H1] Extract the URI from the **rdfsyn:about** attribute and

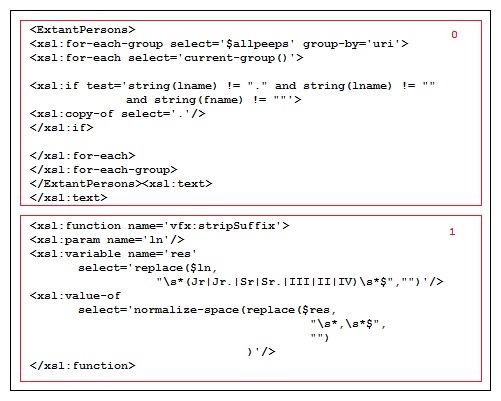
1. [F26H1a] Create a person element from the **foaf**, **core** and **rdfs** elements and set the aka attribute to ‘**N**' indicating that this data is not from **aka:nameParts** triples;
2. [F26H1b] For each **aka:nameParts** triple with this URI as a subject create a person element with the aka attribute set to ‘**Y**’.



Per0 XSLT Part 1 Figure 26

The next figure shows the portion of the transform that outputs the XML **Per0.xml** and the **vfx:stripSuffix** function used to remove such data from the last name of each person.

* [F27H0] This outputs the person elements that have last and first names.
* [F27H1] This function removes common suffixes from the end of a string. These often cause mismatches because they are employed inconsistently.



Per0 XSLT Part2 Figure 27