ORE Ontology GEM Implementation

Table of Contents

- Overview
- Ontologies
    - ORE Triples Examples
- Data Structure Approaches
        - Limit testing for array in-memory data structure
        - Analysis:
        - Alternate Approaches:
    - Use Cases:
        - Use Case: Fetch the first N items in the list and display to user; Fetch next N and display; Fetch prev N and display; Fetch page X and display
        - Use Case: Add additional information to ListItemInfo before display
        - Use Case: Move position of an item in the list
        - Use Case: Add (append) item to end of list
        - Use Case: Sort list items by value not stored in list.
- List Header Info Structure
- List Item Info Structure

Overview

This page explores design issues for an efficient ORE ontology gem implementation based off the ActiveTriples framework.

Ontologies

The following is a list of all ontologies used by the Triple Examples.

Ontology Name	Prefix	URL	Details	Comments
RDF	`rdf`	`http://www.w3.org/1999/02/22-rdf-syntax-ns#`	specification	
RDF Schema	rdfs	`http://www.w3.org/2000/01/rdf-schema#`	specification	
Dublin Core	dc	`http://purl.org/dc/elements/1.1/`	specification	
ORE	ore	`http://www.openarchives.org/ore/terms/`	specification	Represents both ordered and unordered items using the Aggregation class.
IANA	iana	`http://www.iana.org/assignments/relation/`	specification	ORE uses this ontology for first, last, next, and prev predicates.
Friend of a Friend	foaf	`http://xmlns.com/foaf/0.1`	specification	Uses ld4l-foaf_rdf gem.

ORE Triples Examples

Turtle using ORE ontology's Aggregation class

```
@prefix ore:     <http://www.openarchives.org/ore/terms/> .
@prefix iana:    <http://www.iana.org/assignments/relation/> .
@prefix dc:      <http://purl.org/dc/elements/1.1/> .

<http://localhost:3000/individual/vc155> a ore:Aggregation ;
  ore:aggregates <http://da-rdf.library.cornell.edu/individual/b3652730> ;
  ore:aggregates <http://da-rdf.library.cornell.edu/individual/b3652234> ;
  ore:aggregates <http://da-rdf.library.cornell.edu/individual/b3652543> ;
  iana:first <http://localhost:3000/individual/vci162> ;
  iana:last <http://localhost:3000/individual/vci164> ;
  dc:title "My list" ;
  dc:description "This is my list of references." .

<http://localhost:3000/individual/vci162> a ore:Proxy ;
  ore:proxyFor <http://da-rdf.library.cornell.edu/individual/b3652730> ;
  ore:proxyIn <http://localhost:3000/individual/vc155> ;
  iana:next <http://localhost:3000/individual/vci163> .

<http://localhost:3000/individual/vci163> a ore:Proxy ;
  ore:proxyFor <http://da-rdf.library.cornell.edu/individual/b3652234> ;
  ore:proxyIn <http://localhost:3000/individual/vc155> ;
  iana:prev <http://localhost:3000/individual/vci162> ;
  iana:next <http://localhost:3000/individual/vci164> .

 <http://localhost:3000/individual/vci164> a ore:Proxy ;
  ore:proxyFor <http://da-rdf.library.cornell.edu/individual/b3652543> ;
  ore:proxyIn <http://localhost:3000/individual/vc155> ;
  iana:prev <http://localhost:3000/individual/vci163> .
```

Data Structure Approaches

The ORE implementation in the triple store will maintain a doubly linked list using IANA ontologies first and last predicates on the list and next and prev on the proxy. Once loaded into memory, the ORE triples will be converted into a List Header Info hash data structure with an array of items each holding a List Item Info hash data structure. Testing was executed to analyze the efficiency of working with arrays of List Item Info hash data structures.

Limit testing for array in-memory data structure

Tests descriptions:

- array_create - use Array fill method to add array items with values from 0 to max_items
- array_move - use Array insert(to, delete_at(from)) to move an item from the end of the filled array to the beginning (worst case scenario)
- list_create - create a list header data structure and list item data structures with sample real world data using items[i]=item_info to add each item to the items array
- list_move - use Array insert(to,delete_at(from)) to move an item from the end of the filled array to the beginning AND update prev and next links
- list_find - use 0.upto(items.size-1) to check the value of items[i][:uri] to see if it matches the search value for uri - Test looks search for last item in list

Environments for testing:

- Laptop
  - RAM: 16 G (approximately 4.5 G is allocated to other running programs prior to running tests)
  - Processor: 2.3 GHz quad core
- DEV VM
  - RAM: 2 G (approximately 0.6 G is allocated to other running programs prior to running tests)
  - Processor: 2.3 GHz dual core

Target production system:

- RAM: 2-8 G
- Processor: 2.3 GHz dual core

Test code: https://gist.github.com/elrayle/f5f559f8c10243600dc6

Results:

Max Items	array_create		array_move		list_create		list_move		list_insert		list_append		list_find_last		list_find_random	
	Laptop	DEV VM	Laptop	DEV VM	Laptop (16Gb)	DEV VM (2Gb)	Laptop	DEV VM	Laptop	DEV VM	Laptop	DEV VM	Laptop	DEV VM	Laptop	DEV VM
1,000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10,000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
100,000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
500,000	0	0	0	0	1	2	0	0	0	0	0	0	0	0	0	0
1,000,000	0	0	0	0	4	5	0	0	0	0	0	0	0	0	0	0
2,000,000	0	0	0	0	11	13	0	0	0	0	0	0	0	0	0	0
4,000,000	0	0	0	0	35	OOM	0	OOM	0	OOM	0	OOM	1	OOM	0	OOM
8,000,000	1	1	0	0	115		0		0		0		2		0	
10,000,000	1	1	0	0	167		0		0		0		2		0	
20,000,000	1	2	0	0	OOM		OOM		OOM		OOM		OOM		OOM	
40,000,000	4	4	0	0												
80,000,000	6	8	0	0												
100,000,000	8	11	1	1												

* Time measured in seconds

* OOM - Out of Memory

Analysis:

Converting triples data into an array of items is constrained by memory and the execution time of retrieving data from the triple store and constructing the list. A modest 2Gb of RAM will support holding all data up to approximately 100,000 items and maintain processing efficiency. Implementations supporting a list of greater than 100,000 items or many concurrent lists of smaller item counts will likely require a more sophisticated approach to retrieving and storing items in-memory.

Alternate Approaches:

- Combine ActiveTriples with Solr. In this implementation, ActiveTriples will be used for managing and updating individual resources and/or a few of resources at a time (e.g. the Aggregation resource and any modified Proxy resources). For managing viewing and paging through lists, data will be stored in Solr and use built in Solr features for sorting, requesting pages of data and paging forward and back, extending Proxy resource data to include data about the item being aggregated that comes from different data stores, etc.

- Inclusion of an order predicate in the triples for the list AND/OR in the Solr fields for the proxies. More exploration needs to happen to determine the feasibility of inclusion of an order triple in the list triples. Any change to the order will require a change to potentially all proxy triples in the list. The same scale of change would be required on the Solr side, but could be done asynchronously. In this approach, the triple store will be immediately correct upon persistence of the change and the Solr index may lag. As the Solr index is used primarily for UI display, the lag is likely acceptable with the proxies near the display being changed first and the out of date proxies being farther away from those being displayed.

Use Cases:

Use Case: Fetch the first N items in the list and display to user; Fetch next N and display; Fetch prev N and display; Fetch page X and display

Scenario:

User is browsing the list starting from the beginning, displaying N items at a time with the ability to move forward and back through the display of N items.

Operations:

- Fetch range starting from first and retrieving the next N items
- Iterate over list from ListHeaderInfo.first_loaded to last_loaded using ListItemInfo.next (next method) to traverse the list
    - Display items in order with value=ListItemInfo.proxyFor URI and id=ListItemInfo.uri (Actually will get the title from a proxyFor triple and use that as the display value, but proxyFor's title isn't available from List Item Info. It requires extra processing beyond the ORE GEM.)
- User presses next button causing fetch of next N items starting from ListHeaderInfo.resume_next_token
- Iterate and display
- User presses previous button causing fetch of prev N items starting from ListHeaderInfo.resume_prev_token
- User presses button for page X; Find first item on page X
    - For Doubly linked list + order index, the first item on page X with page size = N can be calculated.
    - For Doubly linked list, not directly supported. Could do traversal of list (potentially asynchronous) to determine first item on Y pages beyond and preceding the current page.

Analysis:

- In-memory Data Structure
  - All three will perform equally well for iteration from first to last.
- Triple Store Data Structure
  - Doubly linked list + order index will perform better for fetch N items. However, this is likely to not be significant at low N. Not sure how large N needs to be before this becomes an issue.
  - Doubly linked list + order index Triple Store Data Structure will perform much better for fetch page X.

Use Case: Add additional information to ListItemInfo before display

Scenario:

Display information is not part of ORE triples. For example, proxyFor is a URI to a book which has a DC.title, DC.description, etc. Would like to be able to add this information to each ListItemInfo structure and pass that to the UI code for display.

Operations:

- Fetch occurs according to previous user case.
- Before display,
  - traverse the loaded items, fetch(rdf_subject=ListItemInfo.proxyFor)
  - get object values to be displayed
  - add to ListItemInfo hash structure
- Send updated ListItemInfo structures to UI for display

Analysis:

- In-memory Data Structure
  - All three will perform equally well for traverse and update.
  - See also analysis of Fetch N items use case.
- Triple Store Data Structure
  - Both will perform equally well for traverse and update
  - See also analysis of Fetch N items use case.

Use Case: Move position of an item in the list

Scenario:

User is browsing the list and drags an item to a new position in the list.

Operations:

- KNOWN: ListItemInfo.uri of item being moved (from id of item in display list) – could also use ListItemInfo.position
- KNOWN: ListItemInfo.uri of item preceding position of insert – could also use ListItemInfo.position
- update links of old prev item, old next item, new prev item, new next item, and item being moved (and potentially list first or last if needed)
- persist all modified resources (perhaps triggered by user clicking Save button or auto-save with each change)

Analysis:

- In-memory Data Structure
  - Hash is most efficient with direct access by ListItemInfo.uri (key of hash) and no moves are required within the data structure. Only links are updated.
  - Doubly linked list is less efficient as the list has to be traversed to find the items to update. No moves are required within the data structure as only links are updated.
  - Array is the least efficient. Direct access using ListItemInfo.position can be used to locate the items, but the array will need to be reordered to reflect the change in order of the list items. (Assumes reorder operation is more expensive than traversal operation.)
- Triple Store Data Structure
  - Both will perform equally well.

Use Case: Add (append) item to end of list

Scenario:

User is browsing the catalog and decides to add one or more bibliographic references to the list. Add items defaults to append to end of list.

Operations:

- create new instance of ORE::Proxy

- fetch ListHeaderInfo.last
- update list items prev and next links and list last
- persist all three resources

Analysis:

- In-memory Data Structure
  - All three will perform equally well.
- Triple Store Data Structure
  - Both will perform equally well.

Use Case: Sort list items by value not stored in list.

Scenario:

User chooses to sort list items by a display value that is not stored as part of the triples associated with the ORE list triples. For example, the list is aggregating bibliographic references. The properties of the bibliographic references may be stored in a separate triplestore. Example sort criteria are title, publication date,

Operations:

- How to query across from one triplestore, TS? Query: fetch where TS.uri in TS.uri.aggregates sort_by TS.title start_at 60 limit 20
- How to query across two triplestores, TS1 for list and TS2 for bibliographic references? Query: fetch where TS2.uri in TS1.uri.aggregates sort_by TS2.title start_at 60 limit 20
- How to query from Solr? How to add to Solr? How to make incremental updates?

Analysis:

- In-memory Data Structure
  - ???
- Triple Store Data Structure
  - ???

List Header Info Structure

The list header info will be a hash. It holds information about the list as a whole and the currently loaded range of items as a sub-structure. The values for the list information stored depends on the type of the loaded items sub-structure. The loaded items sub-structure holds items as a group are in-memory using one of four possible grouping methods: doubly linked lists, doubly linked lists + order index, array, or hash with proxy URI as the key. NOTE: Order index is global to the entire list. The array and hash implementations may also have this global order index information stored with the item info.

		example value types				Comments
key	**description of value**	**doubly linked**	**doubly linked + order**	**array**	**hash**	
first	pointer to first item in list	URI of first proxy	URI of first proxy	URI of first proxy	URI of first proxy	
last	pointer to last item in list	URI of last proxy	URI of last proxy	URI of last proxy	URI of last proxy	
count	count of all items in list	X	last.index	last.index	last.index	Count for full list is available for array and hash when order index is stored in triplestore.
first_loaded	pointer to first item in loaded range of items	item	item	array[0]	URI of first loaded proxy	
current	pointer to current item	item	item	int cpos	URI of current proxy	• points to first item on fetch • updated to next/prev item during traversal • points to last modified item when changes are made
last_loaded	pointer to last item in loaded range of items	item	item	array[size of array]	URI of last loaded proxy	

count_loaded	count of items in currently loaded range	X	=position of last - position of first	size of array	size of hash	
resume_next_token	pointer to item after last_loaded	URI of proxy	URI of proxy	URI of proxy	URI of proxy	
resume_prev_token	pointer to item just before first_loaded	URI of proxy	URI of proxy	URI of proxy	URI of proxy	
loaded_items	pointer to loaded items	use first_loaded	use first_loaded	array	hash	

List Item Info Structure

Each item's info is stored in a hash. It holds information about the individual item. The loaded items as a group are in-memory using one of four possible grouping methods: doubly linked lists, doubly linked lists + order index, array, or hash with proxy URI as the key.

		example value types				
key	**description of value**	**doubly linked**	**doubly linked + order**	**array**	**hash**	**Comments**
uri	URI of proxy for this item	URI of this proxy	URI of this proxy	URI of this proxy	URI of this proxy	This is the hash key for loaded_items when the list is implemented as a hash.
prev	pointer to previous item in list	URI of prev proxy	URI of prev proxy	= p - 1	URI of prev proxy	
next	pointer to next item in list	URI of next proxy	URI of next proxy	= p + 1	URI of next proxy	
prev_loaded	pointer to previous item in loaded range of items	prev item	prev item	= pl - 1	URI of prev proxy	
next_loaded	pointer to next item in loaded range of items	next item	next item	= pl + 1	URI of next proxy	
proxyFor	URI of list item being aggregated	URI	URI	URI	URI	
proxyIn	URI of list aggregation	URI of aggregation	URI of aggregation	URI of aggregation	URI of aggregation	
proxyIn_loaded	pointer to List Header	header	header	header	header	
position (p)	position in full list	X	p	p	p	Position in full list is available for array and hash when order index is stored in triplestore.
position_loaded (pl)	position in loaded range of items	X	pl	pl	X	