Undefined 1 (2009) 1–5
IOS Press

Tag Cloud Query Builder for Semantic Web Knowledge Bases

Xingjian Zhang [a] and Jeff Heflin [a]
[a] *Department of Computer Science and Engineering, Lehigh University, 19 Memorial Drive West, Bethlehem, PA 18015*
E-mail: xiz307@lehigh.edu and heflin@cse.lehigh.edu

Abstract. As the number of available Semantic Web triples continues to grow, the need for effective tools to browse large Semantic Web Knowledge Bases (SWKBs) becomes more and more evident. In this paper, we present the novel Tag Cloud Query Builder (TCQB), which allows users to visually browse ontologies and then generate queries by connecting browsing chains to a query graph. TCQB utilizes font size and text color in the tag cloud to convey the contextual data frequency and click frequency of each concept, and enables users to quickly grasp the usage pattern of unfamiliar ontologies and the domain of the SWKB. To build a query, the user can store historical browsing chains in a query cart, which is visualized by a query graph, and then link different chains by specifying the instance nodes in these chains are equivalent or adding filters that involves two literal nodes.

Keywords: Semantic Web, Interface, Tag Cloud, Browsing

1. Introduction

The body of knowledge encoded in Semantic Web format is larger and more diverse than ever before. The current Sindice [5] index contains about 67 million Semantic Web and microformat documents. In addition, as more mapping axioms and linking data become available, it becomes possible to query across different sources. Also, many Semantic Web Knowledge Bases (SWKBs) have been designed to store and manage voluminous Semantic Web data. As a result, users are expecting more powerful and practical use of these systems and data. However, most common users are still not able to interact with these systems and data. We identify the following three knowledge gaps for non-expert users attempting to interact with the SWKB:

- The user may not be familiar with the ontological terms in the SWKB. It is very unlikely for the users to memorize the exact URIs and understand the real concept of the terms. It is even harder if these terms come from many different sources and involves many different relations among them.
- Even if the user is familiar the terms, he or she might not know which terms are sufficiently populated in order to support different types of queries. Two SWKBs could have exactly the same ontologies but very different distribution of data because of different focus of domain.
- Manually writing SPARQL queries is tedious for most users.

While many previous works have presented new interfaces for SWKBs, we find most of them cannot resolve all the three aspects at the same time. Users often feel frustrated when they attempt to issue queries or fail to get meaningful results. In particular, the second gap is very rarely discussed. On the other hand, The “tag cloud” or “word cloud” concept is currently widely used and helpful in browsing collections of free text documents, particularly in Web 2.0, where the importance of folksonomy tags is often shown with font size or color in a cloud to guide users the topics of the documents. Based on the aforementioned findings and the motivation for filling these gaps, in this paper we

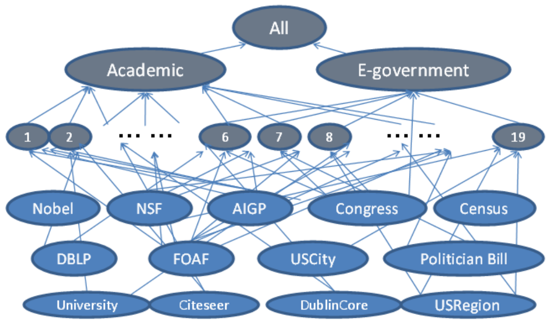


Fig. 1. Ontologies and mappings used in this paper.

propose the Tag Cloud Query Builder (TCQB) [1] for exploring and querying a SWKB. It inherits the advantages of browsing from the traditional tag cloud system, but also integrates more ontological information to the interface and reduces the effort of constructing queries after browsing and finding the desired terms.

We design TCQB as a generic exploration and query interface to the SWKBs, without fixing it to any specific SWKB. In practice, we use DLDB [7] in this paper as our underlying Knowledge Base System. DLDB partially materializes the inferred facts at load time, and integrates sources at query time yet maintains reasonable query performance. DLDB also employs a DL reasoner to supplement its reasoning on TBox. We use Hawkeye as our data set which contains 13 ontologies and more than 24 million triples[2]. These ontologies mainly focus on 2 different categories: academic domains and E-government domains. Some of them are new ontologies created by us and some are adapted from existing ontologies or data schema. An overview of mappings is shown in Figure 1. We also use over 20,000 *owl:sameAs* statements to link instances from different sources.

The rest of the paper is organized as follows. In Section 2 and 3, we describe our approach to browsing and query building; In Section 4 we explain the underlying computations for the system; In Section 5 we compare TCQB to similar previous work; Lastly in Section 6 we conclude and discuss the future work.

[1]http://gimli.cse.lehigh.edu:8080/TCQB/

[2]See http://swat.cse.lehigh.edu/resources/ for details.

2. Tag Cloud Browsing

TCQB is motivated by the knowledge gaps aforementioned in Section 1 and resolves these three aspects by representing the terms from a set of ontologies as a tag cloud, where a user can explore the ontologies by making a series of tag selections. To convey information about the domain and focus of the ontologies, the tag cloud presents the tags in different font sizes and colors. While browsing, the historical chains records every step the user clicks, and are reused as components to compose a query, which allows the user to quickly query an unfamiliar set of ontologies, relieves the burden on a user's memory and reduces the need for manual query writing.

2.1. Tag Cloud Display of Ontology Terms

In traditional tag cloud applications, every tag is a link to a set of related resources that are marked with that tag. Tags are typically listed alphabetically, where the importance of each tag is represented by a different font size or color. We have developed a similar technique in order to represent the information contained in a SWKB. The tags in our application are the qualified names of ontological *terms*, including both classes and properties. These names consist of both an ontology prefix, which usually follows the conventions of the sources themselves, and the local name of the term. The sort order is based on the local name first and the prefix second, so that terms from different sources with similar syntactic forms (e.g. *foaf:Person* and *dc:Person*) tend to be clustered.

We convey two major SWKB metrics in the tag cloud: the system's focus, *data frequency* and users' focus, *click frequency*. First, the data frequency shows the distribution of data and the domain of the SWKB. Sometimes a rich SWKB may have several auxiliary terms that are not so closely related to the focused domain. For example some geographical terms may be used in academic domains to specify the locations of universities. While terms in the focused domain have lots of instances in the SWKB, the auxiliary ones have very few of instances. By simply suggesting the relative quantity of instances of terms, we can provide users a straightforward view of the SWKB and suggest the domain and capability of query answering. Second, the click frequency shows the general users' interests. New users may be curious about how other users are browsing the SWKB. Thus those small but useful enumerated classes, e.g. *region:USState* which has only around 50 instances, will tell users its importance in another way. In our system, we provide the information of the data frequency and click frequency by font size and color respectively in the tag cloud.

While most traditional tag cloud systems have only one tag cloud, we believe when applied to SWKBs, a specific tag cloud for each term is more desirable to show the pairwise relatedness between this term and others. In our application, each term has its own tag cloud of related terms, and is called the *base term* of this cloud. The user selects a base term (owl:Thing by default) and all related ontological terms appear in a tag cloud (we shall define clearly the meaning of "related" shortly). Clicking on any of the related terms selects that term as the base term and repopulates the cloud. Thus users can easily find out the relatedness between a base term and terms in the cloud determined by this base term. While also considering the current base term, we define the contextual frequencies and the details are as follows.

The contextual data frequency of a term relative to a base term is used to determine the font size. This frequency is the number of results if a conjunctive query involving the two terms is issued, which tries to find the shared instances by the base term and the term in the tag cloud. Note that in the TCQB any given base term has many related terms and that a related term can either be a class or a property, so TCQB consists of two alternate *views*: the *class view* and the *property view*. In the class view, only the related classes of the base term are shown in the tag cloud; and in the property view, only the related properties are shown. Thus we have four possible combinations of terms.

1. When the base term is a class C_0, in its class view, a class C_1 is related to C_0 and will be shown in the tag cloud iff their intersection is nonempty. The larger their intersection is, the stronger their relatedness is, and the larger the font of C_1 is in the tag cloud of C_0. Figure 2 shows an example of the class view of the tag cloud of *foaf:Person*. For example, the intersection of *foaf:Person* and *nsf:Person* is larger than the intersection of *foaf:Person* and *bill:Senator* in our SWKB, so *nsf:Person* has a larger font in the tag cloud.
2. In the property view of the tag cloud of C_0, a property P_1 is related to C_0 iff there exist some instances of the class C_0 that appear as subjects in the triples with P_1 as the predicate. The number of the distinct instances of C_0 that meet such criteria decides the relatedness between C_0 and P_1 and font size of P_1 in the tag cloud of C_0. Figure 3 shows an example of the property view of the tag cloud of *foaf:Person*. Note that we also include inverse properties (e.g. *dblp:author-* in Figure 3), which are denoted as the original properties (e.g. *dblp:author*)followed by a "-". The base term then is actually the type of objects of some triples with the original properties as the predicate. Since the direction of an object property is usually an arbitrary choice by the creator of the ontology, by introducing the inverse properties, we provide users more comprehensive information on how instances of the base term class can be involved in a triple in the SWKB. Inverse properties are also included in the following cases.
3. When the base term is a property P_0, in its class view, a class C_1 is related to P_0 iff in the object of of triples with the predicate P_0 and an object that is an instances of C_1. The number distinct triples of such pattern decides the relatedness and the font size of C_1.
4. In the property view of P_0, a property P_1 is related to P_0 iff there exist some pairs of the instances that have both the relation P_0 and P_1. The fact that P_0 and P_1 have shared pairs implies they may have some semantical relatedness. The relationship may be just subproperty - superproperty, or may be some possible coincidence, such as "classmate" and "coauthor". The number of distinct instance pairs decides the relatedness between P_0 and P_1 and the font size of P_1 in the tag cloud.

While the first case tells the direct relatedness of two classes by showing the common instances, the second case and the third case defines a partial triple patten (either the subject or the object is not specified to be an instance of a given class). When the two partial pattens are applied together with the same predicate, a full pattern is created and tells how one class is related to another via a property. So when a user click a class then a property then another class, a natural interpretation it that he is trying to find a relation between these two classes, and might like to issue a query on such a pattern. So in TCQB, we make it as default that the property view is shown for a class tag cloud and the class view is shown for a property tag cloud to facilitate users' exploration on patterns. However in the fourth case, the property view of a property, we had two design choices of what information should be conveyed. Besides the current design, we can also make the property view of a property tag cloud as a selection of property composition, i.e. we show all the properties that appear in triples with at least one of the objects of the last property as their subjects. However in order to mirror the definition in the first case, we believe the current approach minimizes the confusion of users. A user can always express a property composition $P_1 \odot P_2$ by clicking P_1 then *owl:Thing* then P_2. Note that in all cases, the consideration of whether a class has an instance or whether a property appears in a triple is based on the inference results by the SWKB, thus whatever entailments are supported, TCQB provides a consistent view of it. More implementation details will be further discussed in Section 4.1.

In contrast to traditional tag cloud browsing, our system is designed to display the relevant sizes instead of absolute sizes of terms in the tag cloud. The justification for this approach is that a user is frequently curious about how close other terms are to the current term during his browsing. Additionally, a user can always obtain an absolute size by specifying owl:Thing as the base term.

Similar to contextual data frequency, contextual click frequency may provide hints as to how likely it is for a related term to be examined next after the current term. Brighter colors signify a more popular term w.r.t the current base term (i.e., one with a larger click count than its neighbors in the current cloud).

2.2. *Conveying Structural Information of the Cloud*

While inheriting the advantages from traditional tag clouds, we also leveraged the highly structured relationships available from ontologies (e.g., subclasses, subproperties, domains and ranges, etc) to increase the quality of the display. On one hand, this structural information more accurately determines which terms should be linked to a base term. On the other, it provides the capability to represent clusters of terms in a group within the tag cloud. In other words, a user can discover both the terms that are related to a base term, and relationships among them. For example, given *foaf:Person* as the base term, both *aigp:Researcher* and *dblp:Person* are shown in the tag cloud. In addition, *aigp:Researcher* is a subclass of *dblp:Person*. Such important information may also help users to decide which term is more accurate for his or her information need, and should also be conveyed in the system.

To avoid busying the display with too many colorful edges, and to make the relationships between the terms clearer, we design the system as follows. The user can focus on the neighborhood of a particular term simply by hovering the mouse over the term for a few seconds[3]. Related terms in the cloud (e.g. subclass or superclass) display special icons representing their relationship with the term, while other unrelated terms and their icons become grey and less noticeable. Figure 4 shows an example of a class tag cloud when a user hovers the mouse on dblp:Publication within the base term *owl:Thing*.

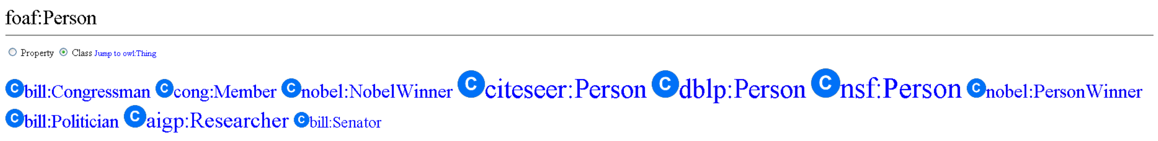
In TCQB, we define three different relationships between any two terms in the tag cloud: *subclass* (or *subproperty*), *superclass* (or *superproperty*), and *from same ontology*. We call each a *highlight feature*. More highlight features, e.g. semantically related according to some linguistic analysis, may be added to the system in the future. Note that the relationships between two terms may include more than one highlight feature and we represent this case with an icon that consists of sectors with different colors corresponding to all matching features. An example of this scenario is *dblp:Article* in Figure 4, which is a subclass and also from the same ontology of the hovering term *dblp:Publication*. The user can disable one or more highlight features, but by default all the features are selected.

3. Query Construction based on Browsing Chains

Another unique feature of our system is the query construction component. Most existing tag cloud systems only provide a ranked list of documents after a user browses the tag cloud. In the case of a folksonomy, the list would be the instances of a specific tag class. However with a SWKB, users require customized query answers. Customizing the query functionality too greatly may require the user to possess background knowledge and/or require tedious interactions between the user and the system. On the other hand, a specialized approach (e.g., canned queries) cannot always fulfill a user's needs. An intuitive compromise between the two extremes is to browse the tag cloud with the support of the aforementioned relative size and frequency features.

As the user browses through terms, TCQB maintains a browsing history chain similar to a bread-

[3]Currently we set the hovering time as 3 seconds. A thorough use study may help decide such setting for better use experience.

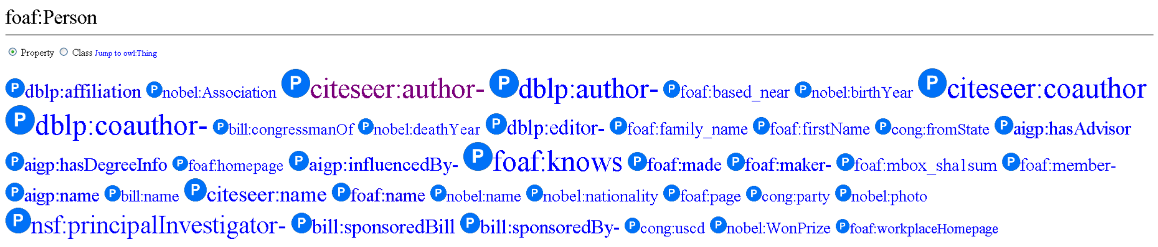


foaf:Person

○ Property ● Class Jump to owl:Thing

bill:Congressman cong:Member nobel:NobelWinner citeseer:Person dblp:Person nsf:Person nobel:PersonWinner bill:Politician aigp:Researcher bill:Senator

Fig. 2. Example of a class view of the tag cloud of the base term*foaf:Person*.

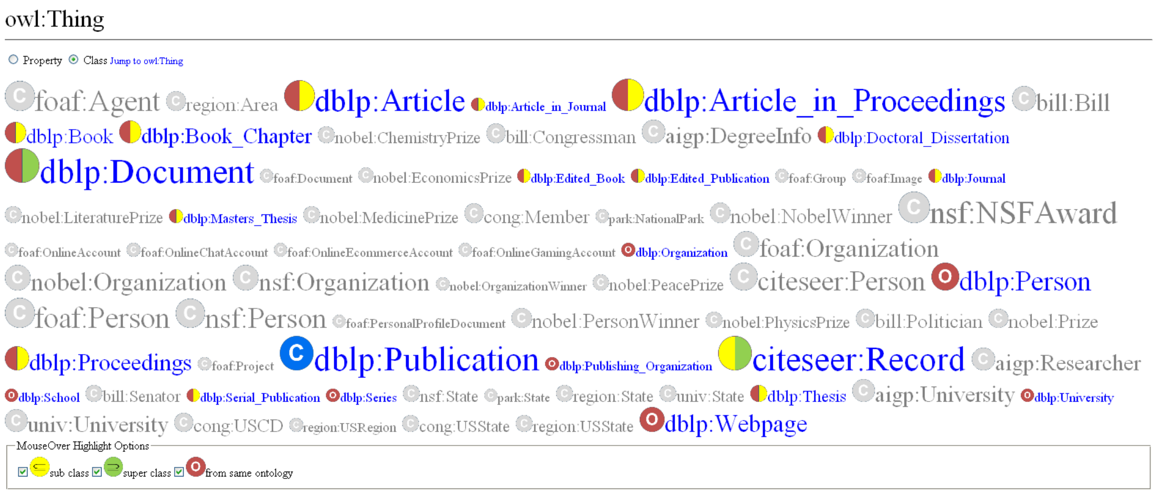


foaf:Person

● Property ○ Class Jump to owl:Thing

dblp:affiliation nobel:Association citeseer:author- dblp:author- foaf:based_near nobel:birthYear citeseer:coauthor dblp:coauthor- bill:congressmanOf nobel:deathYear dblp:editor- foaf:family_name foaf:firstName cong:fromState aigp:hasAdvisor aigp:hasDegreeInfo foaf:homepage aigp:influencedBy- foaf:knows foaf:made foaf:maker- foaf:mbox_sha1sum foaf:member- aigp:name bill:name citeseer:name foaf:name nobel:name nobel:nationality foaf:page cong:party nobel:photo nsf:principalInvestigator- bill:sponsoredBill bill:sponsoredBy- cong:uscd nobel:WonPrize foaf:workplaceHomepage

Fig. 3. Example of a property view of the tag cloud of the base term*foaf:Person*.



owl:Thing

○ Property ● Class Jump to owl:Thing

foaf:Agent region:Area dblp:Article dblp:Article_in_Journal dblp:Article_in_Proceedings bill:Bill dblp:Book dblp:Book_Chapter nobel:ChemistryPrize bill:Congressman aigp:DegreeInfo dblp:Doctoral_Dissertation dblp:Document foaf:Document nobel:EconomicsPrize dblp:Edited_Book dblp:Edited_Publication foaf:Group foaf:Image dblp:Journal nobel:LiteraturePrize dblp:Masters_Thesis nobel:MedicinePrize cong:Member park:NationalPark nobel:NobelWinner nsf:NSFAward foaf:OnlineAccount foaf:OnlineChatAccount foaf:OnlineEcommerceAccount foaf:OnlineGamingAccount dblp:Organization foaf:Organization nobel:Organization nsf:Organization nobel:OrganizationWinner nobel:PeacePrize citeseer:Person dblp:Person foaf:Person nsf:Person foaf:PersonalProfileDocument nobel:PersonWinner nobel:PhysicsPrize bill:Politician nobel:Prize dblp:Proceedings foaf:Project dblp:Publication dblp:Publishing_Organization citeseer:Record aigp:Researcher dblp:School bill:Senator dblp:Serial_Publication dblp:Series nsf:State park:State region:State univ:State dblp:Thesis aigp:University dblp:University univ:University cong:USCD region:USRegion cong:USState region:USState dblp:Webpage

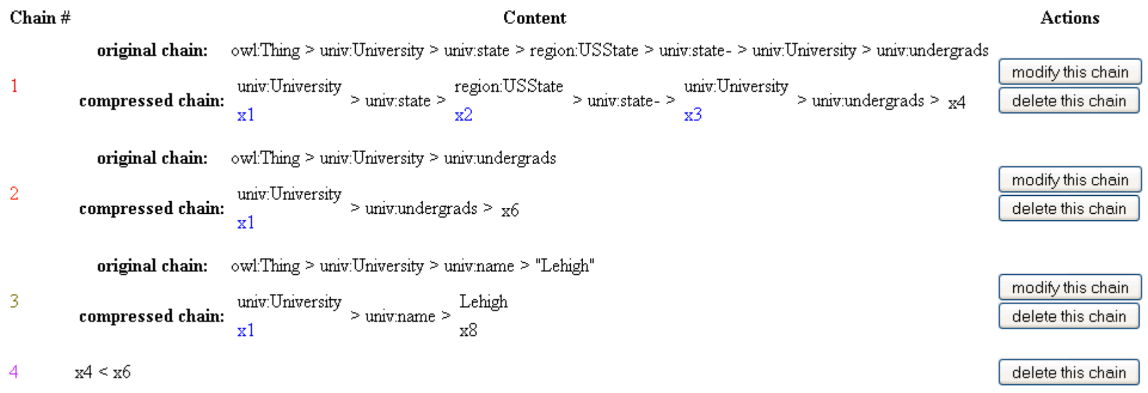
MouseOver Highlight Options

☑ sub class ☑ super class ☑ from same ontology

Fig. 4. Example of highlight feature for *dblp:Publication* in the class tag cloud of *owl:Thing*. *dblp:Article* is a subclass, and also from the same ontology of *dblp:Publication*.

crumb[4]. On one hand, like normal breadcrumbs on web sites, it provides links back to each previous page the user navigated through to get to the current page. On the other, it helps construct a query, i.e. a user can issue a query based on what has been browsed. TCQB extracts a query chain from the browsing chain. For example if the browsing chain is $C_1 - C_2 - P_1 - C_3$, we have a query chain $C_2 - P_1 - C_3$ to find $\{(x_1, x_2)|C_2(x_1) \wedge C_3(x_2) \wedge P_1(x_1, x_2)\}$. Note a query chain must alternate between selections of classes and properties, corresponding to the alternation between nodes and edges in the RDF graph. By default, the views also change alternatively, e.g. after the user clicked a term in a class view, the tag cloud of that term would be the property view. However, since TCQB also allows users to change views, there might be adjacent classes or properties. In such case, TCQB treats the adjacent classes (or properties) in a browsing chain as a refinement and only keeps the last of them in the query chain. Datatype properties cannot have class views because their objects are not instances of a class. As such, data type properties can only occur at

[4]http://en.wikipedia.org/wiki/Breadcrumb_(navigation)



Chain #		Content	Actions
1	original chain:	owl:Thing > univ:University > univ:state > region:USState > univ:state- > univ:University > univ:undergrads	modify this chain
	compressed chain:	univ:University x1 > univ:state > region:USState x2 > univ:state- > univ:University x3 > univ:undergrads > x4	delete this chain
2	original chain:	owl:Thing > univ:University > univ:undergrads	modify this chain
	compressed chain:	univ:University x1 > univ:undergrads > x6	delete this chain
3	original chain:	owl:Thing > univ:University > univ:name > "Lehigh"	modify this chain
	compressed chain:	univ:University x1 > univ:name > Lehigh x8	delete this chain
4	x4 < x6		delete this chain

Fig. 5. Example of a query cart.

the end of chains. Whether datatype properties should have property views is an open question. If we show a tag cloud, it tells different predicates that link the same instance - literal pair. If the same literals do have the same meaning or even refer to the same entity in the real world, users can get the same information as in the property view of an object property. However, the same literals sometimes do not refer to the same thing, especially for integer values. Thus showing datatype properties that share some same instance - literal pairs may result in weird or misleading display. According to our current design of property view, we do not show the property view of datatype properties.

Inverse properties in the browsing chains allow users to build a single chain without being stuck due to the direction of properties and provide more flexibility of expressing a query within a chain. However, sometimes a single query chain is still not expressive enough for a complex query. Thus we also provide the functionality of combining multiple query chains to form a query. When the user thinks a browsing chain is representing *part* of the query, instead of issuing it right away, he can add the browsing chain to the *query cart*. In the query cart, all the previous chains kept by the user are listed, like the items listed in the shopping cart in any online purchase website. In addition, we provide a visualized *query graph* of the current query cart and detect whether all parts are connected to avoid undesired queries. A query graph is very similar to a RDF graph in which nodes are bound with variables, except that literals (data type nodes) can be also connected with filters. Thus, the user can connect query chains by specifying that different variables in different query chains refer to the same instance, or by adding Filters that involve multiple data type variables. Currently, the system only supports Filters with comparisons on two variables, but support for more complex filters is not a difficult problem. Figures 5 and 6 show an example of the chains listed in the query cart and the query graph which tries to find the universities that are located in the same state as Lehigh University but have fewer undergrad students. Such a query can be created by the following actions of the user.

1. adding a query chain: univ:University x1 > univ:state > region:USState x2 > univ:state- > univ:University x3 > univ:undergrads > x4
2. adding a query chain: univ:University x5 > univ:undergrads > x6
3. adding a query chain: univ:University x7 > univ:name > ‘Lehigh’ x8
4. specifying that x1 and x5 are equivalent
5. specifying that x1 and x7 are equivalent
6. specifying a filter that x4 is less than x6

All the variables are randomly assigned to the nodes (including instance nodes and literal nodes), and can be renamed by the user. The user can also modify or delete any chain in the query cart. The colors of edges are also randomly selected and correspond to the color of chains in the query cart, so that users can quickly find the chain from the graph.

After building a query graph via the procedures illustrated above and issuing it, TCQB first displays the SPARQL code and then after user confirms the query, sends it to the underlying SWKB. The result from the SWKB is shown as a table of n-tuples. All the instances in it are displayed with human readable labels, and can be further investigated. After a user clicks an instance in the result, TCQB shows all the properties and values of the properties (up to 5 values are shown

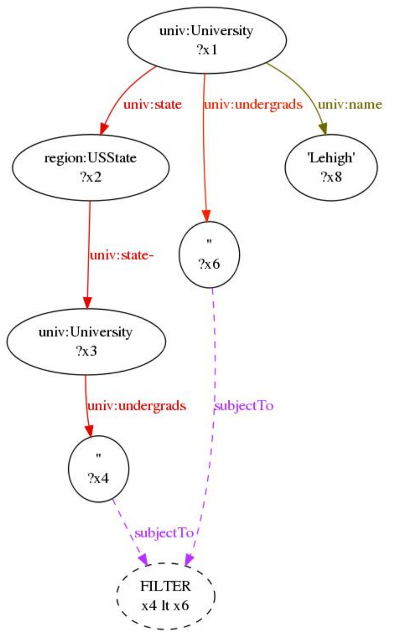


Fig. 6. Example of a query graph.

Table 1

Relevant Subset of Term T_1 to the Base Term T_0, i.e. $T_1 : T_0$. Both terms could be either a class C_i or a property P_i.

	T_1	
T_0	C_1(class view)	P_1(property view)
C_0	$\{x \mid C_0(x) \wedge C_1(x)\}$	$\{x \mid C_0(x) \wedge P_1(x,y)\}$
P_0	$\{(x,y) \mid P_0(x,y) \wedge C_1(y)\}$	$\{(x,y) \mid P_0(x,y) \wedge P_1(x,y)\}$

for each property). The user can further click one of the properties in this *instance view*, and then continue the browsing and query building in the same way except the chain now starts with an instance.

4. Implementation

The tag cloud displaying requires some underlying computations, and we will introduce the implementation in this section.

4.1. Calculating the Font Sizes

The font sizes of terms in a tag cloud are decided by the shared instances in the tag cloud term and the base term. Table 1 shows how to compute the set of shared instances, i.e. the subset of the base term that is relevant to the tag term.

A class can be treated as a set of instances, and a property can be treated as a set of binary relations. So we can use $|S|$, the size of a set, to represent the size of a term. We further define $S_1 : S_0$ the relative subset of term S_1 to the base term S_0, which is shown in Table 1 with different combinations. Thus we can define the relative size of a term T_1 to the base term T_0 as $|T_1 : T_0|$. Now we formally define relatedness between terms. A term T_1 is relevant to a base term T_0 iff $|T_1 : T_0| > 0$. For example, given the base term C_0 *foaf:Person*, the class C_1 *aigp:Researcher* is shown in the class view because the intersection of *foaf:Person*and *aigp:Researcher* has 3655 distinct instances; and the property P_1 *foaf:knows* is shown in the property view because 534943 distinct instances of *foaf:Person* appear in the triples with the predicate *foaf:knows*.

The size of each relevant subset is mapped to a font size, based on the maximum and minimum font size preset for the system. We map the relevant subset with the largest size r_{max} to the maximum font size f_{max}, and then map all other terms, using the formula

$$f_i = f_{min} + (f_{max} - f_{min}) \cdot g(r_i, r_{max}) \qquad (1)$$

where f_i is the font size for a given related term whose relative size to the base term is r_i. g(x,y) is a function with the range (0,1]. It could be as simple as $g(x,y) = x/y$. However, if this implementation is applied, sometimes a few terms tend to have a significantly larger size than all others, which makes only the few large terms noticeable. Thus instead, we use a logarithmic scale: $g(x,y) = \log x / \log y$. This ensures that there is a reasonable distribution of font sizes.

4.2. Preprocessing

Some underlying computation tasks are needed for TCQB, including recording both the ontological axioms (O) and data statistics (D), and manual input (M). For ontological axioms, we record: (O1) the types of ontological terms (i.e., class or object property or data type property), (O2) all the subclass pairs and subproperty pairs, explicitly or implicitly stated, and (O3) the domain and range for each property.

We record some statistics from our KB: (D1) $|T_i|$ for every term T_i in the ontology, and (D2) 4 cases of $|T_i : T_j|$ for all $i \neq j$ in Table 1.

We also manually provided: (M1) the prefixes for all terms in our KB, and (M2) the list of properties whose value can be a human readable name for the instance of domain of that property (e.g., *foaf:name*, *dblp:title*), which enables displaying a user friendly name instead of a URI for most instances. For (M1), the number of prefixes is almost[5] the number of ontologies in the SWKB. For (M2), usually only the super properties need to be specified if the mapping axioms are adequate. These two tasks are not mandatory. Missing a prefix mapping will make the system choose an automatic prefix, such as "*undefined1*". Missing a specification of labeling property will make the system display a URI instead of human friendly words for some instances. Both of these would affect users' understanding, but will not affect the overall functionality of the system. The administrators can fix these problems by simply adding the missing part and restarting TCQB.

Most of the automatic and manual procedures can be done in very short times. However the task (D2) can be very time consuming if we use very naive approaches. With proper materialization of inferred triples and database index, the computation of the relative size for a single pair is very quick. Even the experimental computation of $|T_1 : T_0|$ for both $|T_i| = 10^6$ can be done in seconds. In Hawkeye we only have 3 class, i.e. *dblp:Document*, *dblp:Publication*, and *citeseer:Record*, that have roughly 1.5×10^6 instances each, while other classes have much smaller sizes. However we may have to compute too many pairs. For example, since we have 53 classes, the number of join operations for recording intersections of classes would be $53 \times 52/2 = 1378$ (Note here we do not distinguish between a base term and a term in the class view tag cloud, because the size defined in Table 1 are the same). Instead, we utilize axioms to reduce the number of pairs we need to compute. If C_1 is disjoint with C_2, $\forall C_i \sqsubseteq C_1, |C_i : C_2| = 0$. Also, if $C_1 \sqsupseteq C_2$, we directly use the size of $|C_2 : C_1| = C_2$. The owl:disjointWith axioms can reduce the number of pairs tremendously, thus when they are missing, we create additional ontologies with such disjoint axioms. By doing this we reduce the total number of pairs to a few hundred. For computing the intersections between any two properties, ideally we should examine every pair of object properties (including their inverse properties). Since we have 82 object properties, we need to calculate $82 \times (82 \times 2 - 1) = 13366$ pairs. Also the calculation for intersection between properties takes longer time than that of two classes, because both the subjects and objects are compared. Assume the calculation for each pair of properties takes 10 seconds in average, it would take over 37 hours to finish all the pairs. Thus we trade completeness for efficient computation, and only consider the intersection between pairs of subproperty - superproperty[6], which is exactly the size of the subproperty and need no more calculation since task (D1) already covers it. The other two cases are both computing the intersections between a class and a property. Ideally we should examine any pair in which the domain/range of the property is not disjoint with the class. In practice, based on the result of class intersections, we find the intersection of two classes is usually empty if their relation is not defined as subclass - superclass in the mapping ontology. Thus for calculating the intersection between a class and a property, we only focus on the pairs that the class is a subclass of the domain/range of the property. With all the above optimizations, the total time for preprocessing is about 3 - 4 hours for Hawkeye.

An important question is the impact of data updates on the systems since pre-processing plays an important role in ensuring runtime efficiency. While most of the tasks can be applied to only the additional data and is easy to be finished, it may be necessary to redo (D2) with all the data. However, here we will show that TCQB is relatively robust with the change of data, thus we do not necessarily redo the computation frequently.

Since (D2) is used for deciding the font size of terms in the tag cloud, we now examine the impact of inaccurate size of relevant subset in Table 1. Assume the actual size of a relevant subset is r', in contrast to r that is recorded before updating, and assume the font sizes accordingly are f' and f, we have

$$\delta = f' - f = (f_{max} - f_{min}) \cdot (\log \frac{r'}{r}) / \log r_{max} \quad (2)$$

where δ is the error in displaying the font size. If the change is not in the order of magnitude and r_{max} is big enough, δ is very small. For example, if we set $f_{max} = 40pt, f_{min} = 10pt$, when $r_{max} = 10^5$ and $\frac{r'}{r} = 3.16$, we have $\delta = 3$. This means even if the intersection is more than 3 times of the original one, there will only be a 3 pt difference in font sizes, which is unlikely to be noticed by most users.

[5]There might be multiple prefixes in a single ontologies.

[6]We also consider the inverse properties here, i.e. if $P_1 \sqsubseteq P_2$, we have $P_1^- \sqsubseteq P_2^-$, so the pair of P_1^- and P_2^- is also recorded.

5. Related Work

The TCQB is a unique approach to exploring and querying SWKBs, and to the best of our knowledge, we have not seen another system using a tag cloud display like it. However, it does share similarities with a few existing systems.

Early researchers use graph representations for browsing Semantic Web data, believing it as a natural choice. But later Karger and schraefel [3] pointed out Big Fat Graphs are not the ideal representation for RDF data. Many recent systems, such as VisiNav [2] and BrowseRDF [6], are using or extending the faceted browsing idea. VisiNav allows users to explore a SWKB through "Path traversal", a process which resembles TCQB's notion of browsing chains, and both VisiNav and TCQB allow users to refine the focus of a query while exploring the SWKB. However, one major difference is that VisiNav always returns all the instances that qualify the object of the current property chain, a policy which might have scalability issues for large KBs and long chains. Instead, TCQB only shows the statistics of the SWKB to present an overview of the SWKB and only returns the instances as results of queries. In the aspect of query, faceted browsing only allows users to find answers within a single chain, thus are less expressive than TCQB which allows users to combine multiple chains and issue more complex queries.

Traditional tag cloud interfaces are mostly used for displaying the frequency or popularity of tags in some systems, such as in flickr[7] and delicious[8]. Since the tags in those systems are folksonomies, they need approaches for clustering or trimming. Alternatively, TCQB uses tags from ontological terms, and focuses on very different aspect of displaying, especially in order to show the relations among tags in the system. So TCQB differs significantly from those systems in the contextual tag cloud of each base term, and the highlight features we use to represent the relationships among tags.

There are many different approaches a UI system could provide to users to form a query that would eventually be transformed into a SPARQL query. Natural language input interfaces are most intuitive but also face the challenges in Natural Language Processing (NLP) [1], such as modifer attachment, nominal compound, anaphora, etc. Many other categories of interfaces are investigated. For example, Kaufmann and Bernstein [4] compared 4 different methods: NLP-reduce uses controlled English to simplify the NLP process; Querix asks for interactive feedback from users; Gingseng limits the user input by providing predetermined menus; and Semantic Crystal allows users to drag classes or properties to compose a graph representing a query. NITELIGHT [8] is also a graphical tool similar to Semantic Crystal. TCQB is similar to the graphic tools in the sense that the intermediate step is to generate a graph for a SPARQL query. However, instead of dragging the components, the construction of query in our system is based on browsing chains. These chains guide the user towards meaningful queries, and potentially are more likely to return meaningful results to the user.

[7]www.flickr.com

[8]www.delicious.com

6. Conclusion

Based on our observations of the knowledge gaps that prevent common users from fully utilizing the SWKBs, we propose and implement TCQB, the novel interface for SWKBs with tag cloud browsing and query building based on browsing chains. Tag cloud display indicates the focus of the SWKB, which suggest to users the types of queries it is capable of answering. The contextual tag cloud for each base term helps users find out all the related terms to the current base term and how strongly related they are. The browsing functionality in addition integrates the highlight feature to display the relations among terms in the contextual tag cloud related to the base term. While browsing, the historical browsing chains are recorded, and can be saved into a query cart for later use as part of a query. In the query cart, which is visualized by a (possibly unconnected) query graph, the user can link different chains by specifying the instance nodes in these chains are equivalent or adding filters that involves two literal nodes.

To enable this functionality, several preprocessing tasks need be done. While TCQB itself has no requirement of the underlying SWKB, except the support of standard SPARQL, the preprocessing does add some extra requirement to the choice of SWKB, that is, the ability to finish tasks (D1) and (D2) in reasonable time. It is worth clarifying that all the inferences in these statistical tasks come from the SWKB, thus is consistent with the query results. A SWKB is much easier to apply TCQB to if some interface exists for these tasks. We find that many SPARQL End-

Points have extended the standard SPARQL to allow aggregate functions such as *count*. This enables a simple interface through SPARQL to fulfill the statistical tasks. Our future work includes investigating how to integrate TCQB with these SPARQL EndPoints.

Another potential improvement to TCQB could be efficient estimation of statistical tasks. Since we have shown the robustness of errors in some cases, TCQB may be improved to provide users almost the same information with much less cost of preprocessing time. Ideally we may even achieve runtime estimation instead of preprocessing if efficient approaches such as sampling techniques are applied.

We believe the query cart and the query graph are helpful for the users to build their queries. It could be more helpful if the system allows them to save some of the chains in the query cart or some subgraph in the query graph as a reusable component. Depending on the usage, our system may also be improved to be more socialized by allowing users to share their components, comment and vote on others' components.

References

[1] L. Androutsopoulos. Natural language interfaces to databases - an introduction. *Journal of Natural Language Engineering*, 1:29–81, 1995.

[2] Andreas Harth. Visinav: Visual web data search and navigation. In *Database and Expert Systems Applications (DEXA) 2009*, pages 214–228, 2009.

[3] David R. Karger and m. c. schraefel. The pathetic fallacy of RDF. In *3rd International Semantic Web User Interaction Workshop*, 2006.

[4] Esther Kaufmann and Abraham Bernstein. How useful are natural language interfaces to the semantic web for casual end-users? In *ISWC/ASWC*, pages 281–294, 2007.

[5] Eyal Oren, Renaud Delbru, Michele Catasta, Richard Cyganiak, Holger Stenzhorn, and Giovanni Tummarello. Sindice.com: a document-oriented lookup index for open linked data. *Int. J. Metadata Semant. Ontologies*, 3(1):37–52, 2008.

[6] Eyal Oren, Renaud Delbru, and Stefan Decker. Extending faceted navigation for RDF data. In *International Semantic Web Conference*, pages 559–572, 2006.

[7] Zhengxiang Pan, Xingjian Zhang, and Jeff Heflin. DLDB2: A scalable multi-perspective semantic web repository. In *Web Intelligence*, pages 489–495, 2008.

[8] Alistair Russell and Paul R. Smart. NITELIGHT: A graphical editor for SPARQL queries. In *International Semantic Web Conference (Posters & Demos)*, 2008.