Using Tag Clouds to Quickly Discover Patterns in Linked Data Sets

Xingjian Zhang and Jeff Heflin

Department of Computer Science and Engineering, Lehigh University
19 Memorial Drive West, Bethlehem, PA 18015, USA
xiz307@lehigh.edu, heflin@cse.lehigh.edu

Abstract. Casual users usually have knowledge gaps that prevent them from using a Knowledge Base (KB) effectively. This problem is exacerbated by KBs for linked data sets because they cover ontologies with diverse domains and the data is often incomplete with regard to the ontologies. We believe providing visual summaries of how instances use ontological terms (classes and properties) is a promising route to reveal patterns in the KB and quickly familiarize users with it. In this paper we propose a novel contextual tag cloud system, that treats the ontological terms as tags and uses the font size of tags to reflect the number of instances related to the tags. As opposed to traditional tag clouds, which have a single view over all the data, our system has a dynamically generated set of tag clouds each of which shows proportional relations to a context specified as a tag set of classes and properties. Furthermore, our tags have a precise semantics enabling inference of tags. We optimize the infrastructure to enable scalable online computation. We give several examples of discoveries made about DBPedia using our system.

Keywords: Tag Cloud Browsing, Semantic Web Exploration, Linked Data, Knowledge Discovery

1 Introduction

As the Semantic Web has evolved over the last decade, the amount of interlinked structured data has grown tremendously. While such a huge amount of information potentially enables many different powerful applications, we also notice that there are some obstacles preventing people from taking full advantage of this interlinked web-scale knowledge base (KB). One of the challenges is how to present this huge KB to casual users and familiarize them with it so that they can quickly start building interesting queries and get useful answers. The users' unfamiliarity with the KB (which we refer to as their knowledge gap) arises due to various aspects with regard to both the ontology and data.

- Knowing the ontological terms (classes and properties) is usually the first task. If the user does not know what terms are available in the KB, there is no way for the user to build a formal query.
- Knowing the terms also includes correctly understanding them, not only knowing their names. Suppose the user knows all the URIs of these ontological terms, but if any of them is ambiguous, the user might still issue a meaningless query.

– Knowing the distribution of data in the KB can also be important for a successful query. A query can be semantically correct but practically less helpful, simply because the coverage of the data is incomplete with regard to the ontology.

All these aspects are knowledge gaps casual users typically have, though sometimes some of them might be less important. e.g. if the KB covers **a very focused domain**, with **full-fledged data**, then the second and third gaps are less relevant: the terms with a specific focused domain usually have clear meanings with little ambiguity, and if the data is complete, then presumably any semantically meaningful query will be productive. In this case, providing some simple descriptions about the KB and a whole list of ontological terms should mostly resolve the gaps.

However, this is not the case in the Linked Data world. Even taking an adequate subset of the linked data cloud, we can see wide spread domains from different sources. Then ambiguity becomes a common issue: sometimes a word is used with different senses (e.g. "Bridge" may refer to a structure or a card game), and sometimes the same sense is refined within different domains (e.g. "Person" in a scientific ontology may just refer to "Scientist"). One way to help users understand these terms is to show the axioms related to each term, however sometimes the axioms are missing or too complex to present in a user-friendly way. Another approach is to examine the instances related to each term. While looking into the related instances one by one provides the most details, it is very time consuming, and there is a risk of getting misled by coincidently looking into some erroneous data. Instead, we think providing a summary of a "type" of data can be more efficient. Ideally, a type is more than the named classes in the ontologies, but something defined by users on the fly. Showing summaries of customized types is also helpful for understanding the distributions of data, which helps users to decide which terms to use and how to express the query. If we can properly define "types", by simply providing the count of each type, it will actually reflect the patterns of co-occurrence of ontological terms in instances. From these patterns, a user can get various information: the common patterns help users understand the terms and also understand which queries are selective and which queries have adequate data; the rare patterns can lead users to interesting facts or indicate possible errors in the data. By using the tag cloud paradigm to convey this information, we provide a more straightforward way of showing these patterns: a large tag suggests a common pattern with more instances of the type, while the smaller tags indicate rarer relationships, which may be either very special facts or erroneous data. Such patterns can also be interesting to present to users even in the focused domain with complete data.

We believe the idea of tag cloud summaries of customized types can really help for both KB exploration and query building. Since the types are defined on the fly, there is a trade-off between the expressiveness and time cost. In this paper we formally define these types and the contextual tag cloud of a type (in Section 2), propose an efficient approach to dynamically compute necessary statistics for the tag clouds (in Section 3), and implement a system that demonstrates our idea with examples (in Section 4). We discuss related work in Section 5 and conclude in Section 6.

2 Tag Cloud for RDF Data

In traditional tag cloud applications, every tag is a link to a set of related web pages that are marked with that tag. A tag, in the Web 2.0 sense, is usually a folksonomy provided by users that helps categorize the content of a web page. We can make analogies here for RDF data. An instance is like a web page document, and is already tagged with formal ontological classes, as opposed to folksonomies. In addition, we can also include properties as tags. An instance that has one or more triples involving a property is considered to have this property as one of its tags.

Formally, consider a KB defined by $\mathcal{S}$, a set of RDF statements. Each statement $s \in \mathcal{S}$ can be represented as a triple of subject, predicate and object, i.e. $s = <\text{sub}, \text{pre}, \text{obj}>$. By applying RDF entailment rules [3], we can get $\mathcal{S}^*$, a closure of $\mathcal{S}$ which completes $\mathcal{S}$ with the entailed statements. Using simple queries, we can also extract $\mathcal{C}$ the set of classes, $\mathcal{P}$ the set of properties, $\mathcal{I}$ the set of instances and $\mathcal{L}$ the set of literals. Given an instance $i \in \mathcal{I}$, there are two types of statements with regard to i in $\mathcal{S}$: the ones with i as subjects and the ones with i as objects in the statements. From these statements we can extract the tags for instance i: all the classes and properties that describe i. The predicates in the statements where i is the object are recorded as inverse properties (denoted as $p-$ if p is used in the statement) in order to distinguish from the predicates in the statements where i is the subject . We define $\mathcal{P}^*$ as extended $\mathcal{P}$ including inverse object properties.

We further introduce the negation of tags. While a tag represents that an instance is described by a particular class or property, we use a negated tag to indicate that such a description is missing. This can be useful for inspecting what portions of the data are missing important properties, e.g., how many politicians are missing a political party. We considered three possible semantics for the negated tags: (1) classical negation: Instances have the tag only if the negation of the corresponding concept is logically entailed; (2) negation-as-failure: Instances have this tag if the system fails to infer the positive tag, i.e. it does not have the positive tag in $\mathcal{S}^*$; and (3) explicit negation: Instances have this tag if they do not explicitly have the positive tag in $\mathcal{S}$. Since classical negation cannot be used to find missing properties and explicit negation could lead to confusing scenarios where an instance has a positive inferred tag and a corresponding explicit negation tag, we find negation-as-failure best fits our requirement. A negation of a class $c \in \mathcal{C}$ or a property $p \in \mathcal{P}^*$ can be denoted as $\sim c$ or $\sim p$. The extended class set and property domain with negations are $\hat{\mathcal{C}} = \mathcal{C} \cup \{\sim c | c \in \mathcal{C}\}$ and $\hat{\mathcal{P}}^* = \mathcal{P}^* \cup \{\sim p | p \in \mathcal{P}^*\}$.

Each instance $i \in \mathcal{I}$ is explicitly associated with a set of tags: $Tags : \mathcal{I} \rightarrow 2^{\mathcal{T}}$, where $\mathcal{T} = \hat{\mathcal{C}} \cup \hat{\mathcal{P}}^*$ is the set of all the possible tags of the KB.

$$
\begin{aligned}
Tags(i) = &\{c | c \in \mathcal{C} \wedge <i, \text{rdf:type}, c> \in \mathcal{S}^*\} \cup \{p | p \in \mathcal{P} \wedge \exists j : <i, p, j> \in \mathcal{S}^*\} \\
&\cup \{p- | p \in \mathcal{P} \wedge \exists j : <j, p, i> \in \mathcal{S}^*\} \cup \{\sim c | c \in \mathcal{C} \wedge <i, \text{rdf:type}, c> \notin \mathcal{S}^*\} \\
&\cup \{\sim p | p \in \mathcal{P} \wedge \nexists j : <i, p, j> \in \mathcal{S}^*\} \cup \{\sim p- | p \in \mathcal{P} \wedge \nexists j : <j, p, i> \in \mathcal{S}^*\}
\end{aligned}
$$

In traditional tag cloud systems, tags are typically listed alphabetically, where the importance of each tag is represented by a different font size as a summary of all the web pages. In our scenario, the importance is the size of instances of the type decided by the tag. Currently, most of the popular tag cloud systems provide only a top level

tag cloud, and not a set of **contextual tag clouds** that depend on some selected tags. However in the RDF data scenario, showing contextual tag clouds is very helpful since combinations of ontological tags have more precise semantics than folksonomies, and helps reveal the information about ontological terms and distributions of data.

We define a contextual tag cloud, given a set of tags T_0 as the context, as a list of tags $[t_1, \ldots, t_n]$ with various font sizes $[fs_1, \ldots, fs_n]$ that reflects the instance sizes of types $[T_0 \cup \{t_1\}, \ldots, T_0 \cup \{t_n\}]$. Formally, we can define the process of computing the sizes of instances as a function $count$: $2^{\mathcal{T}} \rightarrow \mathcal{N}$. Semantically, given a tag set T, $count(T) = |\{i | T \subseteq Tags(i)\}|$. Note that $count(T \cup \{\sim t\}) = count(T) - count(T \cup \{t\})$. The font sizes for a tag t_i in the tag cloud is

$$fs_i = (FS_{MAX} - FS_{MIN}) \frac{\log count(T_0 \cup \{t_i\})}{\log count(T_0)} + FS_{MIN}$$

where the max and min font sizes are denoted as FS_{MAX} and FS_{MIN}. We use log functions on the *count* so that the tag cloud shows differences of tags in orders of magnitude.

In addition to calling *count* for generating the contextual tag cloud, in the implemented system, $count(T_0 - \{t_x\})$ is also called for every existing tag $t_x \in T_0$ for the removal of tags. An example of the contextual tag cloud based on {dbp:Person} is shown in Fig. 1. We shall come back to this figure when we study use cases in Section 4.

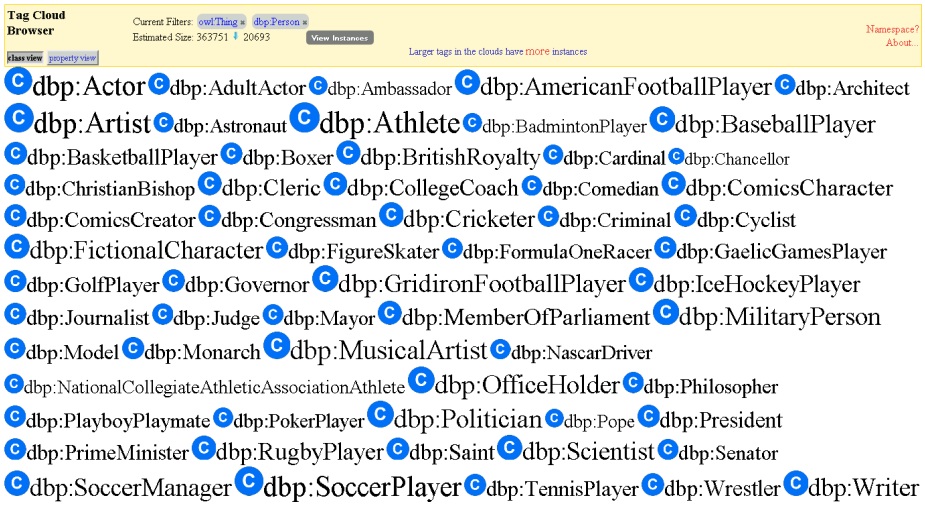

Fig. 1. Class view tag cloud of class Person, while mouse is hovering over Politician in the cloud.

The initial tag cloud has context $T = \emptyset$ or semantically T = {owl:Thing}, and the tags in the cloud reflect the absolute sizes of instances related to each tag. There is no limit on the number of current tags $|T|$, so there can be at most $2^{|C|+|\mathcal{P}^*|}$ combinations (negations can be calculated by the subtraction equation); this is far too many to precompute, thus calculations must be performed in real-time, making performance an

important issue. In order to make this idea scale and practicable for real time online system, we need to work on two sides for the efficiency purpose: (1) decrease the time cost for calling *count*, (2) reduce the calls to *count*. Thus we will discuss this crucial choice of infrastructure in the following section.

3 Infrastructure

Our first attempt to build a system involved an RDBMS, but due to the large number of queries needed, we could not achieve desirable performance. Therefore, we explored the use of Information Retrieval (IR) techniques, which are known to scale well. Treating instances as documents consisting of tags as terms, we can index all the instances, and thus $count(T)$ can be computed by recording the total number of hits for a boolean IR keyword query "t1 AND t2 ..." [1]. The problem for indexing these instances is that we must find all tags regarding a subject in order to create the virtual documents, but in order to scale we must do this with a minimal number of passes through the data. Our solution is as follows:

1. Parse and Write the Big Triple File. The raw RDF files are parsed and written into triples (one triple per line) with qualified URIs (*prefix:local_name*), where the prefix are automatically generated and the namespace mappings are recorded. During the process, if a triple from the raw file is parsed as $<s, p, o>$ and p is an object property, then a flipped equivalent triple $<o, p-, s>$ is also recorded to the output file, which we call the Big Triple File. Thus semantically, if we select all the triples with pattern $<s, ?, ?>$, we can get all the information about this instance s. Note by duplicating the object property statements, the output can have up to twice as the original triple size.

2. Sort the Big Triple File. The Big Triple File is then sorted by the Unix command *sort*. The output of this step is another big triple file with the same file size, but is organized such that all the triples $<s, ?, ?>$ about every instance s are contiguous.

3. Ontology Inference. The ontology file is usually provided in the datasets as separate files. At this step, we only apply RDF entailment rules to ontology triples, preparing for materialization in the following step. Thus we only apply the taxonomy entailments, i.e. superclass and super property entailments: superclass super property hierarchy inferences(rdfs5 and rdfs11) are done at this step and results are kept in memory.

4. Index the Sorted File. The indexer then reads through the sorted file. When reading all the lines that start with an instance s, a virtual document is created with all the ontological terms as the document content, i.e. $\{t|\exists <s, rdf{:}type, t> \vee \exists <s, t, ?>\}$. Based on the entailments in previous step, superclasses or super properties of the explicit tags are added to each instance by entailment rules (rdfs7 and rdfs9). Eventually we can get the set $Tags(s)$ as the virtual document to index. Note that in theory for materialization, we can apply all entailment rules, however, in practice we focus on rules that infer class and property information, and thus generate new tags. Also we find domain and range entailment (rdfs2 and rdfs3) usually introduces wrong classification due to many erroneous statements (mostly flipped subjects/objects, we shall give an example in Scenario 3 in Section 4) in the explicit statements, so we decide not to support such entailments.

[1] Negation can be constructed by specifying a term MUST NOT occur in the boolean query.

In comparison, we discuss how to optimize the DB approach. Firstly, we did not consider holding the KB with a triple table, otherwise our task need expensive join operations over multiple selections on the giant table. On the other hand, the decomposed storage mode [9][1] seems more promising: the triples are inserted into n two-column tables (much smaller) where n is the number of unique properties (including rdf:type) in the data. In each of these tables, the first column contains the subjects that define that property and the second column contains the object values for those subjects. However the join operation on selections on tables is inevitable for our task, thus we made several simplifications and optimizations, in order to minimize the cost of the DB approach. For each $t \in C \cup \mathcal{P}^*$ we create a table with a single indexed column, *id*, an integer that represents each unique instance of tag t. Thus $count(T)$ can be computed by joining the tables of classes and properties. For faster table joins, we use a dictionary that maps all strings (either full URI or its qualified name) to integer ids, and use the ids in the property tables. However this might require maintenance of the dictionary and frequent look-ups slow down the process. Given that the task is only collecting summary information, we do not have to record the real URIs, but only need to know whether an instance has appeared before when processing a new triple. So if we reuse the first three steps in the IR approach, we can just assign an auto increasing integer as id to the different instance from the last one while reading through the sorted file line by line. Then this auto id can be inserted into tables corresponding to the tags in its cluster of triples. Similarly, the superclass and super property inference is materialized at this step. In practice, if the insertion is done line by line there is much waste in the overhead cost of DB operations. Thus instead, we first generate the script file of insertions, and run it in a batch.

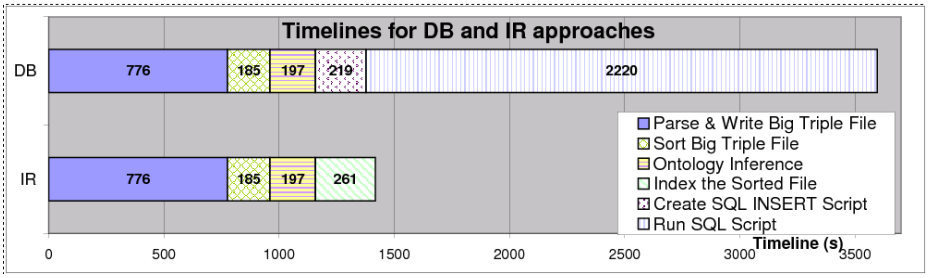

Fig. 2. Timeline for loading data with DB and IR approaches

In order to justify our proposed approach, we choose DBPedia 3.6 [2] as our dataset. Specifically, we load two raw data files, i.e. Ontology Infobox Types and Ontology Infobox Properties, together with the ontology. Although DBPedia itself is not a multi-source interlinked dataset, we still believe that it is a proper dataset for our preliminary experiment and prototype demo system due to the facts that (1) it plays a central role in the Linked Data world; and (2) it has two important features of multi-source linked datasets: an ontology with broad scope and a large scale of data. However in this paper we do not have particular procedures for *owl:sameAs* data, which means if we want to adapt the current work to multi-source datasets, we need either maintain an instance

map before any aforementioned steps or merge the tags of such instances after those steps. The optimization requires further study as one of our major areas for future work.

We use MySQL 5.0.21 and Lucene 3.3.0 as the underlying DB and search engine. In Fig. 2 we illustrate the process of both approaches for loading the DBPedia dataset and compare the timeline between them. There are in total 2,446,683 instances with 27,221,328 triples, including the flipped ones. Using the IR approach, most of the time is the cost of parsing and writing the file. However, comparing between the DB and the IR approaches, we find that the step for running the SQL script itself is already more than the total time of the IR approach. To compare the time cost of *count* by both approaches, we conduct another experiment: run $count(\{t_x, t_y\})$ for a comprehensive set of combinations pairwise $t_x, t_y \in C \cup \mathcal{P}^*$. There are 1952 tags, and thus we call *count* for 1,904,176 ($= \frac{1952 \times 1951}{2}$) times. It turns out that it takes more than 2 hours if we issue so many queries to DB, but only less than 8 minutes if we search the index. This shows the great advantage of the IR approach compared to the DB approach, and thus in the rest of the paper we focus on optimization of the IR approach.

The experiment on pairwise combinations can also be used to reduce the number of times calling $count(T)$ in the online computation. Using offline computation, we can find a disjoint tag set $dis(t_x) = \{t_y | count(\{t_x, t_y\}) = 0\}$ for each tag t_x, and then $dis(T) = \cup_{\forall t_x \in T} dis(t_x)$. Then we can improve the previous naive approach with pruning to speed up the preprocess. For pruning, we apply two rules: (1) whenever $t_x \sqsupset t_y$, compute $count(\{t_1, t_x\})$ before $count(\{t_1, t_y\})$, then $count(\{t_1, t_x\}) = 0$ implies $count(\{t_1, t_y\}) = 0$; and (2) ignore $count(\{t_1, t_2\})$ if $t_1 \sqsupseteq t_2 \vee t_1 \sqsubseteq t_2$, since they are non-disjoint by semantics. We could also use owl:disjointWith axioms, however they do not exist in the DBPedia ontology. After pruning, the number of calls to *count* is reduced to 1,545,754 (saving 19%) and the time cost reduced from 451 s to 387 s (saving 14%) .

With the precomputed $dis(T)$, we can apply three pruning rules for online computation of the tag cloud for any given context T: (p1) ignore $t \in dis(T)$; (p2) ignore $t \in super(T)$, where $super(T)$ is the union of all the super classes and super properties of tags in T; and (p3) we always compute $count(\{t_1, t_x\})$ before $count(\{t_1, t_y\})$ whenever $t_x \sqsupset t_y$ and we can ignore t_y if $count(\{t_1, t_x\}) = 0$. To find how well these pruning rules work, we simulate the real world task by randomly generating 100 contexts T for different context set sizes from 1 to 5. The results are shown in Table 1. Note to generate the tag cloud of T, we need call $count(T \cup \{t_x\})$ for all the candidates t_x in the naive approach, but by pruning, the calls of *count* and the time for these calls are saved for the pruned tags. We also look into how each pruning rule benefits us, and the cost of the pruned ones are recorded as in the naive approach. As we can see the pruning saves around 93% of the calls and the most savings come from Rule (p1), which shows the preprocessing enables a significant optimization of the online computation. We hypothesize that Rule (p2) may save more in some other ontologies with deeper hierarchies, which we want to verify by experiments in future work. Also when we look into the details, we find that the time cost has large variation for different Ts. Generally speaking Ts that contain frequent tags such as Person take longer than those that involve rare tags, which contributes to large standard deviation. That also explains why in general the naive time increases as $|T|$ increases, but it is not the case for $|T| = 5$. The average time for each call is very small, and we can expect thousands of calls within seconds.

Table 1. Comparison between pruning and naive approaches on averaged time cost (in milliseconds) with standard deviations in parenthesis, and number of calls of *count* for 100 random contexts with size from 1 to 5. We also show how each pruning rule contributes to the savings.

$\|T\|$	pruning		naive		savings		Rule (p1)		Rule (p2)		Rule (p3)	
	time	calls	time	calls	time	calls	time	calls	time	calls	time	calls
1	175(1023)	113	561(1086)	1951	69%	94%	384	1837	2.4	0.4	0	0
2	196(608)	118	1131(1292)	1950	83%	94%	925	1813	1.5	0.4	9	19
3	150(178)	132	1196(878)	1949	87%	93%	1020	1793	1.3	0.3	25	24
4	216(229)	158	1615(1234)	1948	87%	92%	1369	1762	1.4	0.4	29	28
5	144(132)	149	1302(933)	1947	89%	92%	1144	1784	0.5	0.2	14	14

4 Prototype System and Case Study

We implement a prototype system with the IR-based infrastructure. The tags are shown as the qualified names consisting 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 alphabetical 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. Since the semantics for adding a class tag and adding a property tag is slightly different, we decide to present two separate clouds: class view and property view. An example tag cloud is shown in Fig. 1, and illustrates some functionalities of our system. The window contains two parts: the status and the cloud part. The status part is always floating at the top of the window even when scrolling down the page. It shows the current tags [2] $T = \{owl{:}Thing, dbp{:}Person\}$ and the current $count(T) = 363751$. It also contains the control where users can switch between class view and property view. As shown in the figure, currently the tag cloud is the class view for T, where we can see that the two largest intersected classes with T are Artist and Athlete [3]. The tag cloud provides users a quick summary of the distribution of the data, and if the user hovers the mouse over one of the tags, the precise number will also be shown at the status count. Particularly in this example, the mouse is over Politician. Clicking a tag in the cloud will add it to T and a new contextual tag cloud will show. Similarly, a user can also change the contextual tag cloud by clicking the remove button next to each current tag, and the user can see the count change by hovering over the remove button.

One important extension to the bare tag cloud, is to allow users to view the instances of the current type defined by T. When clicking the "View Instances" button in the status part, the page goes to Instance Browser, as illustrated in Fig. 3. In this example, $T = \{dbp{:}Artist, dbp{:}director\text{-}\}$. Each page shows 20 instances of the current type, and each instance is a link to the DBPedia instance page. We further take advantage of the IR approach and attach more information to provide more features. We attach the labels of instances while indexing so that the links are human readable. We also attach the file offset in the Sorted Big Triple File, where the lines of this instance's triples starts in the

[2] In this version, the user interface of our system does not support the negation feature.

[3] Although Actor and SoccerPlayer are large tags, we know that they are subclasses of Artist and Athlete respectively.

Fig. 3. Instance Browser of type {*dbp:Artist*, *dbp:director-*}, while the mouse is hovering over dbp:writer- in the detail info of instance 63.

file, so that the system can quickly display a list of all the triples in the KB regarding that instance. Different display styles distinguish URI resources from literals in the details of instances. Meanwhile we also provide shortcuts to quickly modify the current set of context tags. As in the Tag Cloud Browser page, we can remove some tags. Also while looking into the details of some instances, a user might find new interesting tags and add them by clicking the add button next to these tags. Hovering over the add or remove buttons will show the count changes.

With these two web pages: Tag Cloud Browser and Instance Browser, users can explore the KB and get familiar with it by trying out many different combinations. Our system provides a visual summary of the predominant patterns by way of tag clouds, and then lets users see specific examples using the Instance Browser. Now we want to look into some scenarios that demonstrate the usefulness of our system. We start with an example that shows how our system helps users to understand the ontology and the data distribution.

Scenario 1: "find someone who is both a scientist and a writer." v.s. "find someone who is both an actor and a writer". Both queries seem very straightforward because there are classes named Scientist, Writer, and Actor. However in fact, there is no common instance between any two classes of the three. When presented with this odd results, most users will suspect a data quality issue and pursue alternatives. However, if the information need had more conditions, e.g. "find a Canadian who is both a scientist and a writer.", instead of doubting the data, the user may believe that there is no such person in the real world. Aided by our system, the user can clearly see that there is no intersection for {Scientist, Writer} in the data, and then may start with alternative expressions rather than getting misled. For this scenario, there are two properties: dbp:author- and dbp:writer-, which suggest two ways of rephrasing the original queries. It is not clear to us the difference between these two properties since they are synonyms and from the ontology they have the same domain and range. How to choose which property to use? One choice may be the one that will return more answers, because it is more likely that we can still get answers after adding more conditions to the query. In our dataset, dbp:author- has 4825 instances, and dbp:writer- has 13158 instances.

So we always prefer dbp:writer- to dbp:author-? With the help of our contextual tag cloud, we can see that is not true. {dbp:Scientist, dbp:author-} has 167 instances while {dbp:Scientist, dbp:writer-} has only 27 instances. {dbp:Actor, dbp:author-} has 593 instances while {dbp:Actor, dbp:writer-} has 3654 instances. Thus we might choose different properties when building two similar queries.

Scenario 2: Often the interesting instances are the rare ones. While in Scenario 1, we investigate the distribution and pay more attention to common patterns, to use in building queries, in many other cases, users may explore the KB and be curious about those rare patterns. Our system also allows users to invert tag sizes, so that tags that have smaller intersections with the contextual tags will be larger, and thus more "noticeable". e.g. the property view tag cloud of Politician with mouse over doctoralStudent- is shown in Fig. 4. There are only 3 instances, and after examining the details, we find these are all politicians with doctoral degrees. In another example, from the cloud of Mayor, we find starring- a rare but interesting tag, and after examining the only 2 answers, we learned that one mayor was in a documentary film and the other was actually also an actor.

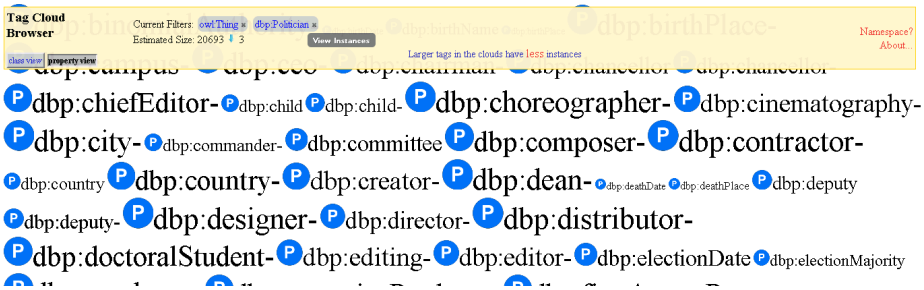

Fig. 4. the property view tag cloud (partial) of Politician with mouse over doctoralStudent-, using inverted tag sizes

Scenario 3: In other cases, rare tags sometimes suggest errors in the data. One common error, made by both humans and automatic programs, is mistakenly inverting the subject and object of an object property. Users can quickly decide which direction is right, by simply taking a vote. e.g. in the property cloud of Work, we find both author and author- are non-empty, but clearly one of them must be wrong. Since author is a much more common tag than author-, the interpretation "has author" is more favored than "is author of" in this dataset. The rare tags also help reveal another common error, i.e. coreference. e.g. in the property cloud of Politician (shown in Fig. 4), choreographer- curiously gives 1 answer. This could be either some very special politician or an error in the data: it turns out there are two Jim Peterson's, a politician and a figure skating coach, but an error in the dataset leads to the politician being described as a choreographer.

In sum, we have found at least three categories of use cases for this system: (1) understand the KB and discover common patterns; (2) learn interesting facts; (3) find

errors in data. As a prototype system, we believe it can also be further extended in any direction of the three as a useful real world application.

5 Related Work

Early researchers used graph representations for browsing Semantic Web data, believing it as a natural choice. But later Karger and schraefel [6] pointed out Big Fat Graphs are not the ideal representation for RDF data. Many recent systems, such as /facet [5], gFacet [4] and BrowseRDF [8], use or extend the faceted browsing idea: the users can construct a selection query by adding constraints and each new added constraint will update the interface to display further facet options based on the current selection query results. The selection operations often vary in different systems. Most systems support selection on property values, and intersection on selections. BrowseRDF supports existential selection on properties and join selection which is equivalent to property composition, and all these operations on inverse properties. Our current system is similar to faceted browsing systems in the sense that our contextual tag cloud idea also has the feature that the new tag cloud is generated based on the previous selected tags. In comparison, our system has less expressiveness than BrowseRDF since it supports only existential, inverse existential and intersection, and does not support any operations on the values of properties. However while none of the 3 papers stressed the scalability issue, our system particularly focuses on limited expressiveness, i.e. the existence of properties and classes, and by optimizing the infrastructure, provides a more scalable performance over large datasets. Meanwhile we want to emphasize, except for the same flavor of adding constrains on the fly and the comparison of expressiveness and scalability, our system has different purposes compared to faceted browsing systems. Our system aims at revealing the patterns of co-occurrence of ontological terms and familiarizing the users with the KB while the faceted browsing systems mostly help users find specific instances that meets some criteria. Another type of exploration tools provides summaries of datasets. e.g. Explod [7] provides a summary graph for class and property usage of grouped instances, however such summary information is buried in bracketed labels, making patterns less obvious.

To visualize the patterns we apply the tag cloud techniques. Traditional tag cloud interfaces are mostly used for displaying the frequency or popularity of tags in some systems, such as in flickr[4] and delicious[5]. While tags in these systems are folksonomies, tags in our system are ontological terms, and thus have precise semantics enabling inference on the tags. Most tag cloud systems only provide a top level tag cloud. TagExplorer [10] is similar to our contextual tag cloud idea, providing a dynamic tag cloud based on the subset of instances selected by the previous selected tags as context. While they treat the tags as values of facets (like predefined properties), and they system classifies folksonomies automatically, our system provides more precise semantics by considering the type defined by the RDF data.

[4] www.flickr.com
[5] www.delicious.com

6 Conclusion and Future Work

In this paper we propose a new approach to familiarize casual users with a KB built on linked datasets: we treat classes and properties as tags with precise semantics and instances as documents, and use contextual tag clouds to visualize the patterns of co-occurrence of ontological terms in the instances specified by the context tags. From the common patterns users can better understand the distribution of data in the KB; and from the rare patterns users can either find interesting special facts or errors in the data.

Our future work mainly lies in applying this technique to larger datasets with more ontologies and data. We believe it can scale up to the whole Linked Open Data cloud, however we understand there are potential challenges. First the infrastructure may need further optimization, new algorithms or even parallel systems for the larger dataset. Also currently we have not developed specific algorithms for owl:sameAs statements, which is critical for visualizing across linked data sets. The interface might also need modifications for new schemes to display larger tag clouds without contributing to information overload. For example, we can hide some subclass tags in the cloud, and these tags can be found in the cloud when the user add a superclass of it into the context.

References

1. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable semantic web data management using vertical partitioning. In: 33rd International Conference on Very Large Data Bases(VLDB'07). pp. 411–422 (2007)
2. Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., Ives, Z.: DBPedia: A nucleus for a web of open data. In: 6th International Semantic Web Conference and 2nd Asian Semantic Web Conference(ISWC'07/ASWC'07). pp. 722–735 (2007)
3. Hayes, P.: RDF semantics: W3C Recommendation 10 February 2004 (Feb 2004), `http://www.w3.org/TR/rdf-mt/`
4. Heim, P., Ziegler, J., Lohmann, S.: gFacet: A browser for the web of data. In: International Workshop on Interacting with Multimedia Content in the Social Semantic Web (IMC-SSW'08). vol. 417, pp. 49–58 (2008)
5. Hildebrand, M., van Ossenbruggen, J., Hardman, L.: /facet: A browser for heterogeneous Semantic Web repositories. In: 5th International Semantic Web Conference(ISWC'06). vol. 4273, pp. 272–285 (2006)
6. Karger, D.R., schraefel, m.c.: The pathetic fallacy of RDF. In: 3rd International Semantic Web User Interaction Workshop (2006)
7. Khatchadourian, S., Consens, M.: Explod: Summary-based exploration of interlinking and rdf usage in the linked open data cloud. In: 7th Extended Semantic Web Conference(ESWC'10). pp. 272–287 (2010)
8. Oren, E., Delbru, R., Decker, S.: Extending faceted navigation for RDF data. In: 5th International Semantic Web Conference(ISWC'06). pp. 559–572 (2006)
9. Pan, Z., Heflin, J.: DLDB: Extending relational databases to support Semantic Web queries. In: Workshop on Practical and Scaleable Semantic Web Systems, ISWC 2003. pp. 109–113 (2003)
10. Sigurbjornsson, B.: TagExplorer: Faceted browsing of flickr photos. Tech. Rep. YL-2010-005, Yahoo! Research (August 2010)