Exploring Linked Data with Contextual Tag Clouds

Xingjian Zhang, Dezhao Song, Sambhawa Priya, Zachary Daniels, Kelly Reynolds, Jeff Heflin

Department of Computer Science and Engineering, Lehigh University. USA

Abstract

In this paper we present the contextual tag cloud system: a novel application that helps users explore a large scale RDF dataset. Unlike folksonomy tags used in most traditional tag clouds, the tags in our system are ontological terms (classes and properties), and a user can construct a context with a set of tags that defines a subset of instances. Then in the contextual tag cloud, the font size of each tag depends on the number of instances that are associated with that tag and all tags in the context. Each contextual tag cloud serves as a summary of the distribution of relevant data, and by changing the context, the user can quickly gain an understanding of patterns in the data. Furthermore, the user can choose to include RDFS taxonomic and/or domain/range entailment in the calculations of tag sizes, thereby understanding the impact of semantics on the data. In this paper, we describe how the system can be used as a query building assistant, a data explorer for casual users, or a diagnosis tool for data providers. To resolve the key challenge of how to scale to Linked Data, we combine a scalable preprocessing approach with a specially-constructed inverted index, use three approaches to prune unnecessary counts for faster online computations, and design a paging and streaming interface. Together, these techniques enable a responsive system that in particular holds a dataset with more than 1.4 billion triples and over 380,000 tags. Via experimentation, we show how much our design choices benefit the responsiveness of our system.

Keywords: Linked Data, Tag Cloud, Semantic Data Exploration, Scalability

1. Introduction

We present the contextual tag cloud system[1] as an attempt to address the following questions: How can we help casual users explore the Linked Open Data (LOD) cloud? Can we provide a more detailed summary of linkages beyond the LOD cloud diagram[2]? Can we help data providers find potential errors or missing links in a multi-source dataset of mixed quality? When a user wants to design a SPARQL query for an unfamiliar dataset, they must resolve three basic questions: (1) Syntactic Correctness: "What classes are available?" (2) Semantic Correctness: "Does this class refer to the concept I expect?" (3) Meaningful Results: "Does the dataset hold enough knowledge coded with the vocabulary I choose?" Since there are two aspects of a dataset: the ontological terms (classes and properties) and the instances, the questions cannot be answered by only viewing the ontology axioms or only inspecting a small sample of instances. A combined view of both aspects is necessary. Furthermore, there are two types of linkages: ontological alignment and `owl:sameAs` links between instances. The usability of multi-source RDF dataset is largely affected by the erroneous or missing links of both kinds in the dataset. If we can emphasize the unlikely facts, then data providers will have a tool to help them uncover such problems in the dataset.

Our solution is to use tag clouds to display statistical information about the distribution of instances among various ontological terms. A key feature is that each tag cloud is relative to a type consisting of ontological terms that is dynamically defined by the user. In analogy to traditional Web 2.0 tag cloud systems, an instance is like a web document or photo, but is "tagged" with formal ontological classes, as opposed to folksonomies. Tags are then another name for the categories of instances. We extend the expressiveness and treat classes, proper-

[1] `http://gimli.cse.lehigh.edu:8080/btc/`
[2] `http://lod-cloud.net/`

ties and inverse properties as tags that are assigned to any instances that use these ontological terms in their triples. The font sizes in the tag cloud reflect the number of matching instances for each tag. We allow the user to change their focus on a specific subset of instances in the dataset by specifying a combination of ontological terms as the context on the fly, and then the resulting contextual tag cloud will resize tags to indicate intersection with this context.

With any uncurated dataset, one must maintain a healthy skepticism towards all axioms. Although materialization can lead to many interesting facts, a single erroneous axiom could generate thousands of errors. Rather than attempting to guess which axioms are worthwhile, our system supports multiple levels of inference; and at any time a user can view tag clouds with the same context under different entailment regimes, which helps users understand the dataset better and helps data providers investigate possible errors in the dataset.

Starting from our initial version of the system [1] that used DBPedia data, we add features and load the entire BTC2012 dataset. This complex dataset contains 1.4 billion triples, from which we extract 198.6M unique instances, and assign more than 380K tags to these instances. This multi-source, large-scale dataset brings us challenges in achieving acceptable runtime performance, affordable preprocessing, and user-interface design. The rest of the paper is organized as follows: we first formally define the concepts and computation problems, and then showcase some use scenarios along with introduction to system functionalities; then we discuss the preprocessing steps, online computation and multi-level inference; after that we provide some experimental results; then we compare with related works; and lastly we conclude.

2. Basic Concepts

Given an RDF dataset, an entailment regime R defines what kind of entailment rules will be applied to the explicit triples. In our implementation, we have two specific sets of rules: R_{Sub} for sub/equivalent class/property entailment (rdfs5, rdfs7, rdfs9, rdfs11 [3]); and R_{DR} for property domain/range entailment (rdfs2, rdfs3). We also support the combination of these two sets, leading to four distinct entailment regimes $\mathcal{R} = \{\emptyset, R_{Sub}, R_{DR}, R_{Sub} \cup R_{DR}\}$.

Let $\mathcal{I}$ be the set of all the instances, and $\mathcal{T}$ be the set of all possible tags assigned to instances in the dataset. Given R, the function $\text{Tags}_R : \mathcal{I} \to 2^{\mathcal{T}}$ returns all the tags that are assigned to the given instance under R-inference closure. For $i \in \mathcal{I}$ we assign three types of tags: (1) **Class** C, if $\langle i,$ `rdf:type`, $C\rangle$ is entailed under R. (2) **Property** p, if $\exists j \in \mathcal{I}$, $\langle i, p, j\rangle$ is entailed. (3) **Inverse Property** $p-$, if $\exists j \in \mathcal{I}$, $\langle j, p, i\rangle$ is entailed. Note under monotonic logic, $R_1 \subseteq R_2 \Rightarrow \text{Tags}_{R_1}(i) \subseteq \text{Tags}_{R_2}(i)$. The function $\text{Inst}_R : 2^{\mathcal{T}} \to 2^{\mathcal{I}}$ returns the set of instances that have been assigned the given set of tags. For convenience, we define the frequency of a set of tags T as $f_R(T) = |\text{Inst}_R(T)|$.

Given that we are substituting tags for triples, we can generalize various entailment rules into tag subsumptions. Tag t_1 is a sub tag of tag t_2 if and only if for all sets of assertional triples $\text{Inst}_R(\{t_1\}) \subseteq \text{Inst}_R(\{t_2\})$. Then the domain/range entailment can be turned into sub tag relations. If $\langle p,$ `rdfs:domain`, $C\rangle$ and $\langle p,$ `rdfs:range`, $D\rangle$, then p is a sub tag of C and $p-$ is a sub tag of D.

A context is an expression of tags dynamically constructed by a user. In our implementation, we allow intersections of any number of tags or the negation of tags. A **Negation Tag** $\sim t$ is virtually assigned to an instance i, if $t \notin \text{Tags}_R(i)$. Note that the semantics are based on negation-as-failure. We argue that this is the correct semantics for a system where what is not said is sometimes as important as what is said. Thus a context with $\{t_1, \ldots, t_n, \sim s_1, \ldots, \sim s_m\}$ actually defines a subset of instances: $\text{Inst}_R(\{t_1, \ldots, t_n\}) - \bigcup_{x=1,\ldots,m} \text{Inst}_R(\{s_x\})$. For a given context and entailment regime R, the system shows all the tags used by any instance in the subset specified by the context, and the size of each tag reflects the number of instances having this tag within the subset.

For convenience, we omit the subtle details required to process negation tags for the remainder of this paper. This allows us to present a simplified exposition where a context $T \subset \mathcal{T}$ is a set of tags, and the instances specified by the context is $\text{Inst}_R(T)$.

We define a contextual tag cloud, given context $T \in \mathcal{T}$ and entailment regime R, as a list of tags $[t_1, \ldots, t_n]$ with various font sizes $[fs_1, \ldots, fs_n]$ that reflects the instance sizes $[f_R(T \cup \{t_1\}), \ldots, f_R(T \cup \{t_n\})]$. We always map the total number of instances to the max font size,

[3] `http://www.w3.org/TR/rdf-mt/#RDFSRules`

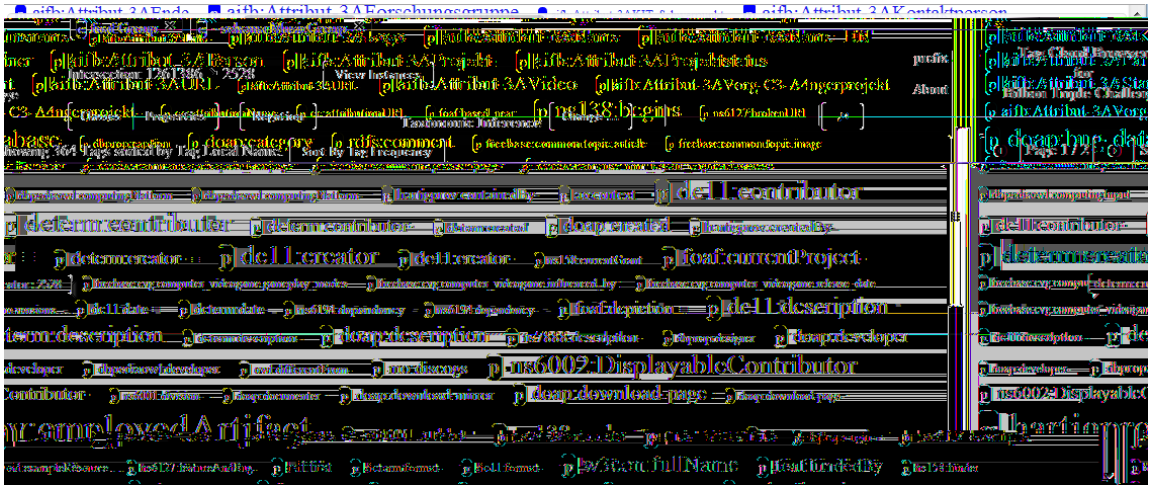

Figure 1: Property Tag Cloud with contexts `foaf:Group` and ~`schema:MusicGroup`.

map 1 to the min font size, and for any given tag frequency, we use log functions on it to calculate the font sizes so that the tag cloud shows differences of tags in orders of magnitude.

3. System Features and Use Cases

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. We put classes and properties into two separate views, so that users will not treat a property called "author" (which may have domain Publication) as a class name by mistake. To emphasize that difference, we also add an icon with "C" or "P" in front of each tag. If a tag is clicked, it will be added to the current context, and then a new tag cloud will be shown for the updated context. A user can add/remove any tags to/from the context, and explore any dynamically defined types of instances. A user can also switch to Instance View to investigate the detailed triples of instances specified by the context.

A user can also change the inference regime, which by default is R_{Sub}, the subsumption inference. Usually we can expect tags to become larger when more inference is introduced. If R entails that a set of tags are equivalent, we choose a canonical tag to group them under. We display a $\equiv$ after the canonical tag to indicate this; clicking it will display the equivalent tags. Also for any tag cloud, we can turn on the negation mode, and then the tag sizes indicate how many instances do not have this tag under the current context and inference level. A negation tag can be also added to the context, which mathematically means the relative complement. For example in Fig. 1, the property tag cloud with context `foaf:Group` and ~`schema:MusicGroup` shows us the common property usages of instances of `foaf:Group` that are not instances of `schema:MusicGroup`.

With the BTC dataset, a challenging problem for UI design is how we can show so many tags in the tag cloud. A straightforward idea is to show tag clouds in pages. To help users locate specific tags in the tag cloud, we initially sort the tags alphabetically by their local names. When the system receives a request (context T and inference R), it will process tags in the same alphabetic order, and then stream out whatever is available for the requested page. If the user chooses to browse tags alphabetically, then the streaming of results is generally able to stay ahead of the user by pre-fetching results for tags on subsequent pages. Instead of browsing, a user can also search for tags by keywords. We index the local name, `rdfs:label` and `rdfs:comment` (if it exists) for each tag to support such keyword search. The retrieved tags will then be shown in the tag cloud sorted by their relevance to the keyword with their frequencies under the current context and inference regime. In addition, we provide sorting by tag frequency as another option, so that users can easily see the most popular tags under the current context and inference. However, we have to wait until all the frequencies are computed to enable this sort option. For some contexts,

Figure 2: Tag cloud returned by keyword search "area" with context `dbpediaowl:Lake`. The tags are sorted by the relevance to the query, while their sizes reflect the frequencies under current context.

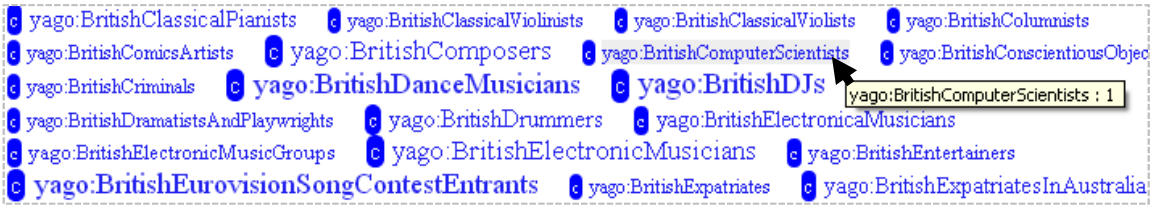

Figure 3: Tag cloud with context `dbpediaowl:MusicalArtist`

it can take a few minutes for the overall computation of thousands of pages of results. We show a progress bar of the computation and the estimated time left; and while waiting for frequency sorting to be available, users can still browse by alphabetical order or search with keywords.

We believe our system can be used for multiple purposes. Here we shall briefly describe four scenarios of a user interacting with the BTC dataset.

Choose the right terms for SPARQL. A user wants to build a SPARQL query on lakes, but does not know what classes about lakes are available. Then by starting with a keyword search "lake", the user is presented with a tag cloud with all tags that match the keyword, and finds that `dbpediaowl:Lake` contains the most instances. After picking this class, the user wonders what property to use for querying the area of a lake. Then by searching again with keyword "area", the user is presented with the contextual tag cloud (as shown in Fig. 2) with keyword-matched tags whose sizes reflect the intersection of the instances of `dbpediaowl:Lake` and the tags. It turns out `dbpediaowl:areaTotal` is the best choice of the property.

Learn interesting facts. A casual user tries a keyword search on "Manhattan". There are classes of parks, streets, hotels, etc. located in Manhattan. However, it also has the class `yago:ManhattanProjectPeople`; the user adds this to the context to explore in more detail. In the resulting tag cloud, the user finds various categories for such people, and then searches again for "scientist". Then surprisingly there is a tag `freebase:computer.computer_scientist`. The user is intrigued, because she did not know that any computer scientists were involved in the effort to build the first atomic bomb. By adding that tag and switching to the Instance View, she finally learns that this scientist is John von Neumann.

Detect Co-reference Mistakes. Sometimes when two tags have a small unexpected intersection, it is due to an error, rather than an interesting fact. For example in Fig. 3, a user finds the tag `yago:BritishComputerScientist` has one common instance with `dbpediaowl:MusicalArtist` (as shown by a very small tag). By adding this tag and looking into the triple details in the Instance View, we find the two `dbpediaowl:abstract` values clearly refer to two different people who have the same name but different birth years and occupations, and are incorrectly linked by an `owl:sameAs` statement in the dataset.

Examine ontological errors. Under inference R_{DR}, a user finds that `foaf:Person` appears in the tag cloud of context `dbpediaowl:Software`, imply-

Figure 4: Examining ontological errors. The first property foaf:homepage in the property view implies class foaf:Person.

ing that some people are software, or vice versa! If the user changes the inference to $R_\emptyset$ or R_{Sub}, this error will disappear. So that means there must be something wrong with the domain-range inference. If there is a property claimed as having `foaf:Person` as it domain, then any instance using this property will be classified as the instance of this class. With this assumption in mind, the user adds both `foaf:Person` and `dbpediaowl:Software` to the context, selects the property view and inference R_{DR}, and sorts the properties by frequency. Then the top tag is `foaf:homepage`, which has all the instances in the current context (by hovering the mouse over the tag, we can see the frequency of this tag). This is very suspicious, and by clicking on the "P" icon before `foaf:homepage`, the user can see (in Fig. 4) that `foaf:Person` is an inferred super tag of this tag, and that causes the error. By checking the raw ontology we find that although the domain of `foaf:homepage` is `owl:Thing` in the `foaf` schema, two other sources in the BTC dataset make the claim that the domain is `foaf:Person` and `foaf:Agent` respectively.

4. Infrastructure

Our main challenge is to compute $f_R(\{t\} \cup T)$ for $\forall t \in \mathcal{T}$ efficiently. There are two ways to approach this problem: (1) ensure efficient calculation of $f_R(T)$ for any T; and (2) prune unnecessary calls of $f_R(\{t\} \cup T)$. Thus we need to correctly structure the repository and develop affordable preprocessing. Our previous experiments [1] showed that an RDBMS with decomposed storage model [2, 3] is not as efficient as using an Information Retrieval (IR) style index for this specific application purpose, both in terms of load time (8X) and online query time (18X). Therefore we extend our IR approach, but meanwhile add more steps to deal with the BTC dataset.

4.1. Preprocessing

Our preprocessing is shown in Fig. 5, where the dashed boxes are input or intermediate data and the solid ones are data results for the online system. First, we parse the raw data and categorize triples into three files.The ontology is processed into a closure set of sub-tag axioms for the given inference regime(s); the closure is then responsible for two functions: $\text{sub}_R(t)$ and $\text{super}_R(t)$ which respectively return the sets of sub/super tags of tag t under inference R. We also use the union-find algorithm to compute the closure for owl:sameAs statements, and pick a canonical id for each `owl:sameAs` cluster by selecting the alphabetically smallest one among the URIs. Then for the instance triples, we replace any instance with its `owl:sameAs` representative (if any). If the object of the triple is also an instance, we flip the triple and add it to the intermediate file, i.e., if the triple is $\langle i, p, j \rangle$, the flipped one is $\langle j, p-, i \rangle$. By this means, we can find all the tags (including inverse property tags) of an instance i by simply looking at the triples with i as a subject. To index an instance, we need to first group all of its triples together. Hence we first output the triples into n files based on the hashcode of the subjects, so that we keep each instance's information in the same file while making each file relatively small. Then we use merge sort on each "replaced and flipped" file, so that triples with the same subject are clustered together. Note that by splitting the triples into n files, we gain benefits from two sides: (1) sorting each file becomes faster, especially since we do not need to merge the sorted files; (2) we can sort in parallel (multiple machines/threads). From the ontology, we compute the inference closure of ontology terms using the chosen entailment regimes. This closure and the sorted triple files are then used to infer tags for each instance and the results are recorded in an inverted index.

The inverted index is built with tags as indexing terms and each tag has a sorted posting list of instances with that tag under each entailment regimes. This means given a "type" defined by a set of tags and an entailment regime, we can quickly find all the instances by doing an intersection over the posting lists. Also, since we use negation as failure, we do not need to index negation tags; their size can be calculated from its complementary tag. i.e. $f_R(\{\sim t\} \cup T) = f_R(T) - f_R(\{t\} \cup T)$. Meanwhile we add other fields such as labels of instances, sameAs sets, file pointers to the raw file, etc. to facilitate other features in our tag cloud system.

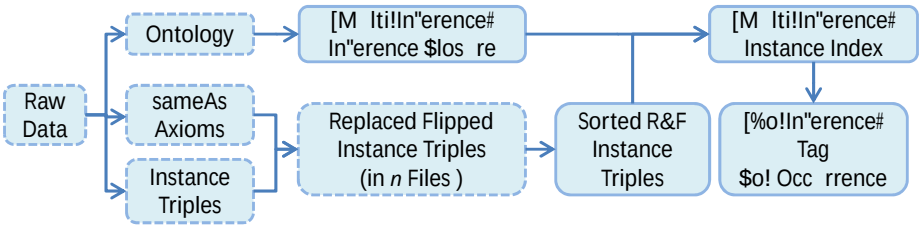

Figure 5: Preprocessing for the tag cloud system

To help prune unnecessary tags under any given context T, we precompute the **Co-occurrence Matrix** M_R: a $|\mathcal{T}| \times |\mathcal{T}|$ symmetric boolean matrix, where $M_R(x, y)$ denotes whether tag t_x and t_y co-occur, i.e. $M_R(x, y) = (f_R(\{t_x, t_y\}) > 0)$. There are three ways to generate M_R.

1. **Traverse all the instances.** For each instance $i \in \mathcal{I}$, get all of its tags $\text{Tags}_R(i)$, for any pair of tags (t_x, t_y) in $\text{Tags}_R(i)$, set $M_R(x, y)$.
2. **Traverse pairs of tags.** For any pair of tags (t_x, t_y) from $\mathcal{T}$, if $f_R(\{t_x, t_y\}) > 0$, set $M_R(x, y)$.
3. **Traverse tag instances.** For each tag $t_x \in \mathcal{T}$, get each instance $i \in \text{Inst}_R(t_x)$, and then set $M_R(x, y)$ for $\forall t_y \in \text{Tags}_R(i)$.

Note that in a multi-source cross-domain dataset such as the BTC dataset, M_R is very sparse since many tags never co-occur in any instance. That suggests that the second approach will get many many pairs with 0 results, and is not as efficient as the third approach where the number of iterations per instance is usually very small as $|\text{Tag}_R(i)|$ are usually very small. Compared to the other two, the first approach will repeatedly set the same cell in $M_R(x, y)$. Given the number of tags, there are no data structures that allow the matrix to fit in memory and be accessed efficiently. So the first approach is slow due to lots of file I/O. Thus, we choose the third in our implementation. This matrix provides a function for each tag t_x to return all the tags that co-occur with it in at least one instance. i.e. $\text{CO}_R(t_x) = \{t_y | M_R(x, y) = 1\}$, which is used in one of our pruning methods for online computation.

4.2. Pruning for Online Computation

Let CL be the candidate list of tags whose queries are finally issued. There are two special cases of the f_R results: (1) $f_R(\{t\} \cup T) = f_R(T)$; and (2) $f_R(\{t\} \cup T) = 0$. For the first case, if t is a super tag of any tag in T, adding t to T does not change the instance set and thus does not change f_R. For the second case, we propose three different pruning approaches to make CL as short as possible.

1. **Use the Co-occurrence Matrix (M).** Given T, $\bigcap_{t' \in T} \text{CO}_R(t')$ has (and not necessarily only has) all the co-occurred tags $\{t | f_R(\{t\} \cup T) > 0\}$.
2. **Use the previous tag cloud cache (P).** Given context $T = \{t_1, \ldots, t_n\}$, the co-occurred tags are a subset of tags in the cached results (if exists) for context $\{t_1, \ldots, t_{n-1}\}$.
3. **Dynamic update (D).** If $f_R(\{t_x\} \cup T) = 0$, ignore $\forall t_y \in sub_R(t_x)$ in further computation.

The online computation works as shown in Fig. 6, where the pruning steps are marked with red circles. First, the input context T will be simplified (under R-Inference) to its semantic-equivalent T' so that any redundant tags will be removed and any equivalent tag will be changed deterministically to a representative tag. Then the system checks whether this semantic-equivalent request has been kept in cache for direct output. If not, the system will get candidate lists CL_M from approach M with input T' and CL_P from approach P with input T. Then we use the intersection $CL = CL_M \cap CL_P$ as the candidate list for queries and keep updating it using approach D. In theory, approach D can be further optimized if we sort the list of tags such that sub tags always follow super tags. However, our system does not use this optimization because it needs to stream results alphabetically. Using simplified T' as input in approach M will get the same candidate tags as using T but avoids unnecessary intersection of lists when computing the candidates. On the other hand, using the original T as input in approach P is necessary for identifying the previous context; and then this previous context is simplified before it is used in cache-lookup.

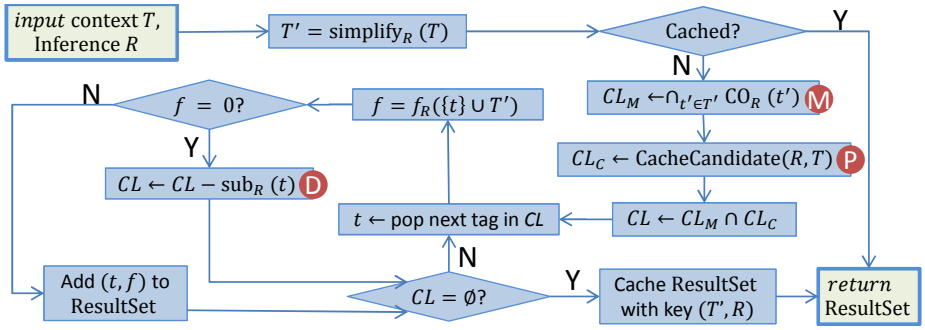

Figure 6: Pruning for Online Computation

4.3. Supporting Different Entailment Regimes

For any given inference R, we can represent Tags_R, Inst_R and CO_R with the primitive no-inference functions: $\text{Tags}_\emptyset$, $\text{Inst}_\emptyset$ and $\text{CO}_\emptyset$ and the tag subsumption axioms: super_R and sub_R.

$$\text{Tags}_R(i) = \bigcup_{t' \in \text{Tags}_\emptyset(i)} \text{super}_R(t') \quad (1)$$

$$\text{Inst}_R(T) = \bigcap_{t \in T} \bigcup_{t' \in \text{sub}_R(t)} \text{Inst}_\emptyset(t') \quad (2)$$

$$\text{CO}_R(t) = \text{super}_R^{\cup}(\text{CO}_\emptyset^{\cup}(\text{sub}_R(t))) \quad (3)$$

where $\text{super}_R^{\cup}(T) = \bigcup_{t' \in T} \text{super}_R(t')$ and $\text{CO}_R^{\cup}(T) = \bigcup_{t' \in T} \text{CO}_R(t')$.

Different entailment regimes are supported by the "[Multi-Inference]" steps in Fig 5. First, inference closure is performed for each entailment regime R. Then when populating the instance index, we use Eq. (1) to compute the Tags_R for each instance and store in different index fields. Since we materialize tags for all regimes, we do not need to use Eq. (2) at query time. Finally in the "Tag Co-Occurrence" step, we only compute $\text{CO}_\emptyset$ and use Eq. (3) at query time.

We made our design choices based on two reasons. (I) How much slower will it be if not materialized? Both Eq. (2) and (3) include union and intersection of sets or posting lists, however the lists of instances are usually much larger and using Eq. (2) significantly increases query time compared to the materialized index. (II) How important is the online performance? As in our scenario, for each tag cloud given T and R, CO_R is only called once, however Inst_R is called for each tag from the candidate set. Also note Eq. (3) can be used for either online computation of CO_R or precomputation if it is materialized. Building the co-occurrence matrix M_R is a very time consuming preprocessing step. We do not need to do that four times for four entailment regimes. Instead, we only need to build $M_\emptyset$, which is the easiest because each instance has the minimal number of tags, and the co-occurrence for all the other entailment regimes can be computed based on Eq. (3).

5. Experiments

Our system is implemented in Java and we conduct all experiments on a RedHat machine with a 12-core Intel 2.8 GHz processor and 40 GB memory.

We apply our preprocessing approach to all five subsets of the BTC2012 dataset, as well as the full dataset, and plot the time/space for datasets against their numbers of total triples in Fig. 7, which shows the scalability of our preprocessing approach. The disk space is for both the index and the no-inference co-occurrence matrix, and is dominated by the index (which usually takes $> 90\%$). The time is quite linear with the total number of triples, because most of the major steps are linear w.r.t. the number of triples. The space however is slightly less correlated to the total number of triples, since many different triples might only contribute to a single tag in the index. For example, 1000 triples saying a `foaf:Person` `foaf:knows` 1000 different people only contribute a single property tag to this person. This is what happens in the `timbl` subset, and explains why `timbl` has slightly more triples than `dbpedia` but needs less time/space.

We then test the response time of $f_R(\{t\} \cup T)$ queries, i.e. how long it takes to count the instances of tag t with context T by querying the index. To ensure a random but meaningful context T, i.e. $\text{Inst}_R(T) \neq \emptyset$, we randomly pick an

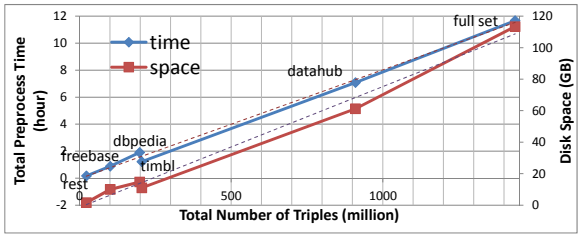

Figure 7: Preprocessing: Time/Space - Total Triples

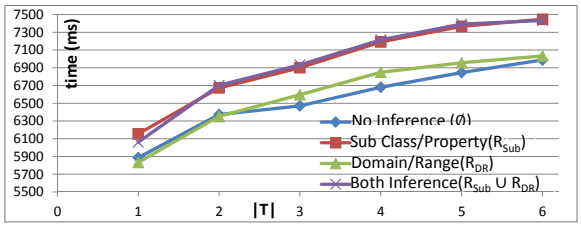

Figure 8: Time for 10K queries averaged over T

Table 1: Average time per 1000 queries over datasets

Dataset	Time	Triples	Instances	Tags	PList
rest	257ms	22.3M	4.1M	3K	6581
freebase	270ms	198.1M	23.5M	308K	3661
timbl	326ms	204.8M	22.7M	12K	11616
dbpedia	341ms	101.2M	31.1M	29K	445
datahub	972ms	910.1M	122.0M	33K	22644
full	1256ms	1436.5M	198.6M	378K	2986

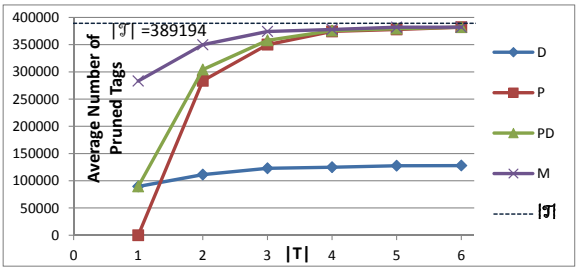

Figure 9: Average Number of Pruned Tags

instance i and get a subset (size of 6) from its tags $\text{Tags}_\emptyset(i)$ as $[t_{i,1}, t_{i,2}, \ldots, t_{i,6}]$. Thus the six tags in this array are known to co-occur under all entailment regimes. We generate 100 such arrays using different i. Additionally, we pick a set S of 10000 random tags. Starting from $k = 1 \ldots 6$ (The initial tag cloud ($|T|$=0) is precomputed and cached, thus we do not test it in the experiments), we use the first k tags in the arrays as contexts T, and we measure the average time of $f_R(\{s\} \cup T)$ for all $s \in S$. While S might overlap with some T, it does not affect measuring the query time since we will issue the same f_R queries without reducing the query terms. We also change $R = \emptyset, R_{Sub}, R_{DR}, R_{Sub} \cup R_{DR}$ to examine the impact of different inference. The average time per 10K queries grouped by $|T|$ is shown in Fig. 8. In average, it takes 0.6∼0.7 milliseconds for a single f_R query. The time slightly increases (sub-linear) when we add more tags to context. It takes longer if R has more inference rules due to longer posting lists of tags in the index. As we expect, since there are fewer tags added to each instance from domain/range inference, we find the curves for R_{DR} and $\emptyset$ are close, while R_{Sub} and $R_{Sub} \cup R_{DR}$ are nearly identical.

For the different subsets, we test the response time of $f_R(T)$ with random T. Since `freebase` does not have any ontology axioms, we choose $R = \emptyset$. For each dataset, we generate 500K random queries for $|T| = 1, \ldots, 5$ (100K each), and record the average time for every 1000 queries. In Table 1, we report the average time for 1000 random queries on each dataset, as well as possible impacting factors such as the number of triples/instances/tags and the average length of posting lists (PList) in the index (i.e. on average, how many instances have each tag). We can see that the numbers of triples/tags do not directly impact query time, but the numbers of instances are very correlated. When the numbers of instances are similar (`timbl` and `freebase`), a huge difference in the average lengths of PLists can also impact the time.

We also test how well our system does for pruning candidate tags under the most complex inference $R = R_{Sub} \cup R_{DR}$. Using the approach above, we generate 100 arrays of length 6 from $\text{Tags}_R(i)$, by changing the length of sub arrays we get 600 random T. As we discussed in Section 4.2, there are three approaches: by co-occurrence matrix (**M**), by previous cache (**P**), or by dynamic update (**D**). By each combination of approaches, we count how many f_R queries are finally issued, and see how many queries are saved. Note there is always some pruning due to super tags of tags in contexts. When using approach P, we always assume the previous cache is available. The average number of pruned tags is shown in Fig. 9. There are $|\mathcal{T}|$ =389K tags in total however most tags only co-occur with a few other tags. Pruning usually saves us many unnecessary queries. When $|T|$ increases any approach will generally prune more tags because more tags in T means a more constrained context. Among the three approaches, M prunes more tags on average,

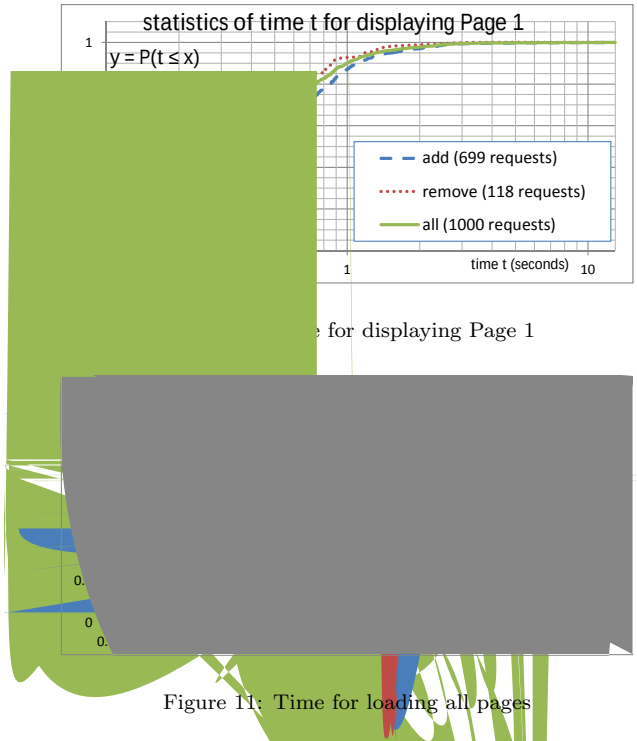

Figure 11: Time for loading all pages

and enabling the other two approaches with M only provides less than 1% more pruning (thus we do not show the overlapping curves for combinations MP, MD and MPD). This justifies the preprocessing for the co-occurrence matrix. P also has good pruning except that when $|T| = 1$, the cache of $|T| = 0$ is a list of every tag and P will not help. However, in the tag cloud scenarios, $|T| = 1$ is important as it will decide the response after the user's first click. Also in the real world the history cache might not be available (e.g. a user adds t_1, t_2, t_3 and then removes t_2). So its availability is a concern although it requires no preprocessing.

Lastly we test with end-to-end web requests. We create a random browser model, with 0.6 probability to add a tag to the context, 0.2 probability to remove a tag from the context, and 0.2 probability to start over with empty context. In order to simulate more realistic requests, when the context size is small, we bias in favor of adding tags. Also when trying to add a tag, we give the more frequent tags in the tag cloud a higher chance to be selected. Using this model, we randomly generate a series of 1000 requests on R_{Sub}, including 699 add requests, 118 remove requests, and 183 start-over requests (note when the context has only one tag, we count the remove request as start-over), with an average context size of 2.53. We record the time spent on displaying all the tags (up to 200 tags per page) in Page 1, and that on finishing computation for all the pages (that is also when the "Sort by Frequency" feature is enabled). In Fig. 10 and Fig. 11 we show the percentage of requests that can be finished within x seconds. For displaying Page 1, 90% of all the requests can be finished within $1s$, and 97.7% within $2s$. Among these requests, start-over is the fastest (not shown) since it just returns the cached results. On average, remove requests are faster than add requests since they are more likely to have a shorter context or be cached due to previous add requests. On the other hand, only 65% of the queries could load all pages in under one second, and the rest varied uniformly up to a minute. This signifies the importance of streaming results and displaying them in pages. Also, this justifies the decision to defer the frequency sort option.

6. Related Work

Many recent systems for exploring RDF datsets, such as /facet [4], gFacet [5] and BrowseRDF [6], use or extend the faceted browsing idea: a user 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. Our system is similar to faceted browsing systems in the sense that each tag in a contextual tag cloud is a "boolean facet" that can be added to the query. Although our system does not currently support constraints on property values, unlike traditional faceted browsers, it supports hundreds of thousands of "facets" and scalably provides the user with information about how each facet would impact the query (via the size of the tag).

To the best of our knowledge, we have not seen any other works like the contextual tag cloud system, nor papers focusing on optimization for the specific kind of query and resolving related problems. Our previous experiments [1] show using an inverted index is much faster than the decomposed storage model [2, 3] for the specific kind of query that counts instances of intersections of classes/properties. We also found [7] this approach is in average 10.2 times faster than the state-of-the-art RDF store RDF-3X [8]. Both experiments indicate that a general purpose SPARQL engine is not always the right choice for a Semantic Web system which requires scalable performance on special kinds of queries. There are many applications using

inverted indices on Semantic Web data. Semantic Web search engines, such as Sindice [9], Watson [10], Falcons [11], etc. create indexes for labels, URLs, literal values or other metadata for locating Semantic documents or entities. Occasionally, question answering systems [12, 13] use inverted indices to help identify entities from natural language inputs. All the above systems index with keywords because the intended usage is to locate relevant resources based on natural language queries posed by users. Our system is very different because the "terms" in our index are no longer keywords but ontological tags. As a result, our index is compatible with entailments sanctioned by the ontologies in the data. This is also why we propose our preprocessing steps prior to indexing, which we have not seen in other works.

7. Conclusion

In this paper we introduce the features and use cases of the contextual tag cloud system, and describe the underlying infrastructure. The contextual tag cloud system is a novel tool that helps both casual users and data providers explore the BTC dataset: by treating classes and properties as tags, we can visualize patterns of co-occurrence and get summaries of the instance data. From the common patterns users can better understand the distribution of data in the KB; and from the rare co-occurrences users can either find interesting special facts or errors in the data. The main challenge is how to provide a responsive system on a large dataset such as the BTC dataset. In addition to the interaction design, we implemented the infrastructure with an inverted index and three pruning approaches, as well as a scalable preprocessing approach. In the experiments, we justified our design choices.

To fully evaluate our current interface, we want to have a formal user study in the future. Meanwhile, we have learned a lot from casual users' feedback. One of the suggestions is to reduce the tags shown on the screen, by clustering similar tags or highlighting the most interesting tags. Another suggestion is to enable search by example, so that users can start with finding a familiar instance and then explore the tags of this instance. We will also study more advanced keyword search algorithms than straightforward string match to improve the recall. There are also ways to integrate our system with other applications. For example, we may add the parallel faceted browsing paradigm [14] to enable comparison between tag clouds. We would also like to extend our system to allow users to annotate tag clouds that are interesting or contain errors, as a social exploration/diagnosis tool for the Linked Data community.

Acknowledgment

This project was partially sponsored by the U.S. Army Research Office (W911NF-11-C-0215). The content of the information does not necessarily reflect the position or the policy of the Government, and no official endorsement should be inferred.

References

[1] X. Zhang, J. Heflin, Using tag clouds to quickly discover patterns in linked data sets, in: COLD Workshop, 2011.
[2] Z. Pan, J. Heflin, DLDB: Extending relational databases to support Semantic Web queries, in: SSWS Workshop, 2003, pp. 109–113.
[3] D. J. Abadi, A. Marcus, S. R. Madden, K. Hollenbach, Scalable Semantic Web data management using vertical partitioning, in: VLDB, 2007, pp. 411–422.
[4] M. Hildebrand, J. van Ossenbruggen, L. Hardman, /facet: A browser for heterogeneous Semantic Web repositories, in: ISWC, 2006, pp. 272–285.
[5] P. Heim, J. Ziegler, S. Lohmann, gFacet: A browser for the web of data, in: IMC-SSW Workshop, Vol. 417, 2008, pp. 49–58.
[6] E. Oren, R. Delbru, S. Decker, Extending faceted navigation for RDF data, in: ISWC, 2006, pp. 559–572.
[7] X. Zhang, D. Song, S. Priya, J. Heflin, Infrastructure for efficient exploration of large scale linked data via contextual tag clouds, in: ISWC2013.
[8] T. Neumann, G. Weikum, The RDF-3X engine for scalable management of rdf data, The VLDB Journal 19 (1) (2010) 91–113.
[9] G. Tummarello, R. Delbru, E. Oren, Sindice.com: Weaving the open linked data, in: The Semantic Web, Vol. 4825 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 552–565.
[10] M. d'Aquin, E. Motta, Watson, more than a semantic web search engine, Semantic Web 2 (1) (2011) 55–63.
[11] G. Cheng, W. Ge, Y. Qu, Falcons: searching and browsing entities on the Semantic Web, in: WWW, 2008, pp. 1101–1102.
[12] C. Unger, L. Bühmann, J. Lehmann, A.-C. Ngonga Ngomo, D. Gerber, P. Cimiano, Tema-50s-based question answering, in: WWW, RDF, 2028, pp.
[[3]