International Journal of Multidisciplinary Allied Research Review and Practices
IJMARRP ISSN No: 2455-1570 (Volume 3, Issue 2, April 2016)

SemRel: Indexing framework for Semantic Relations

Shivakumar Swamy N[1]
Sanjeev C. Lingareddy[2]
[1]PhD Scholar, Department of CSE, JJTU, Jhunjhunu
[2]Prof. and Head, Department of CSE, Alpha college of Engineering, Bangalore
Correspondence Author:nsshivakumar519@gmail.com

***Abstract*—**Semantic web has been the center of research in Information retrieval in the last decade or so. The concentration has been in extracting the relations from the web pages and retrieves them using SPRQL queries. DBpedia has been one of the greatest developments in storing the semantic relations of Wikipedia pages. However, the research has lacked in building an indexing framework for semantic relations. Database is an efficient storage technique for these relations but lacks in retrieval efficiency especially in case of natural language queries. In this work, we propose adaptation semantic relation indexing in an open source indexing framework *Lucene*. The experimental results show that, the framework could be used to store a large scale semantic relation.

I. INTRODUCTION

Semantic web retrieval deals with retrieving the documents by matching the query meaning with the document. The process of semantic web retrieval has multiple steps: (a) Extracting relations from the document, (b) Storing/Indexing the relations for efficient retrieval and (c) matching the entities in the query with the relations in the document. One of the reasons that semantic IR has not been a commercial success is the lack of efficient storage of the relations extracted in the document.

Many applications that deal with semantic relations uses database for storing them and hence lack in efficient retrieval. Current research in this field involves information extraction from the web pages and present it to the user in a structured format. For example, *DBpedia* is used to create the *infobox* for the wiki pages. The question that still persists in this research area is can a semantic IR replace the traditional IR? Some of the recent advances try to addresses the problem of storing semantic relations extracted from web pages. Apache Jena [] and Neo4J [] have been few open source projects built for storing graphical models. These would be beneficiary for extracting the semantically related information from the web pages for a given query. However just building the graphical models from the semantic relations discard the relation-document association information from the index. Also, these projects demand the query to be structured (eg. SPARQL) to retrieve relevant relations. Semantic Similarity relates to computing the similarity between concepts which are not necessarily lexically similar. Semantic similarity aims at providing robust tools for standardizing the content and delivery of information across communicating information sources. This has long been recognized as a central problem in Semantic Web where related sources need to be linked and communicate information to each other. Semantic Web will also enable users to retrieve information in a more natural and intuitive way (as in a query-answering interaction).

In the existing Web, information is acquired from several disparate sources in several formats (mostly text) using different language terminologies. Interpreting the meaning of this information is left to the users. This task can be highly subjective and time consuming. To relate concepts or entities between different sources (the same as for answering user queries involving such concepts or entities), the concepts extracted from each source must be compared in terms of their meaning (i.e. semantically). Semantic similarity offers the means by which this this goal can be realized.

II. RELATED WORK

There have been many works using the Lucene field based indexing for achieving multiple applications. In order to incorporate multilingual setting in standard lucene index is discussed in [1]. Semantic Intelligence defines the power of unstructured data analysis by identifying the required semantics [2]. The detailed description of

semantic web is in [4]. Some work indicates the use of Lucene of rsemantic web as mentioned in [5], However none of them discuss the usage of field indexer for retrieving the semantic relations. The detailed description of the Lucene framework is described in [6]. One of the closely related works to this work is [7], but it does not discuss the usage of Lucene for the same.

The intelligent systems are built and help in decision making process. Especially semantic analysis required in Industries to analyze the Business data. In the world of semantic web semantic analysis is performed on the web data. The web has disjoint set of objects in both text and multimedia space. Semantics can be applied to any type of information. In WWW the semantics are embedded to web pages by the author of web pages to define the context of the information. The tools like search engines [3], utilizes these embedded semantics to provide context aware information for the user query. Semantics can be applied in various domains like Knowledge Management [8], Decision making process [9] and Corporate Intelligence. The basic elements of Semantic Web are Ontology [10], OWL [11], [12] and RDF [13].

III. PROBLEM STATEMENT

In this section, we formally define the problem being addressed in this research work.

The aim of the work is adapt the *Lucene* framework for storing and retrieving semantic relations extracted from the web pages. The idea is to develop a framework that could be used to store semantic relations along with traditional inverted index. This framework should aid basic IR retrieval techniques like vector space and probabilistic techniques to incorporate semantic relations for efficient retrieval.

IV. SEMANTIC RELATIONS AND N-TRIPLES

The purpose of properties is to enable inference. For all the explicit information that has been modeled, what information can be implied is the concentration of this section. RDFS provides a very limited set of inference capabilities. The Web Ontology Language (OWL) provides more elaborate constraints on how information can be specified. A subset of these constraints are discussed in this article.

- Transitive Property (Inference of a relationship between two unrelated nodes)
- Functional Property (Inference that two nodes are the same)
- Inverse Functional Property (Inference that two nodes are the same)
- Symmetric Property (Inference of an additional relationship between two related nodes)
- Asymmetric Property (Prevents a symmetrical inference)
- Reflexive Property (Inference of an additional relationship of one node back to itself)
- Irreflexive Property (Prevents a reflexive inference)
- Property Chains (Inference of a relationship between two unrelated nodes)
- Putting it all together (Combining properties)
- Classic Mereology

A. Transitive Property

In mathematics, a relationship R is said to be transitive if R(a, b) and R(b,c) implies R(a,c). The same idea is used for the OWL construct owl:Transitive Property. A simple example of a transitive property would be if the Windows Operating System has a version Windows XP, which in turn has a version (or type) of Windows XP SP2, then Windows has version Windows XP SP2.

- Windows *hasVersion* Windows XP
- Windows XP *hasVersion* Windows XP SP2
- Windows *hasVersion* Windows XP SP2

Because owl:TransitiveProperty is a class of properties, we can assert in our model that a given property (such as hasVersion) is a member of the class: ?p rdf:type owl:TransitiveProperty :hasVersion rdf:type owl:TransitiveProperty

Consider the property hasAncestor which is meant to link individuals X and Y whenever X is a direct descendant of Y. The ”special case” of hasAncestor is the property hasParent and it can be defined as a subproperty of hasAncestor.

if X is an ancestor of Y and Y is an ancestor of Z, then X is also an ancestor of Z.Any ancentor of an ancestor X is aslo an ancestor for Y if X is an ancestor for Y.A transitive property interlinks two individuals X and Z whenever it interlinks X with Y and Y with Z for some individual Y.

/*Now let us have a look at a property hasAncestor which is meant to link individuals A and B whenever A is a direct descendant of B. Clearly, the property hasParent is a special case of hasAncestor and can be defined as a subproperty thereof. Still, it would be nice to ”automatically” include parents of parents (and parents of parents of parents). This can be done by defining hasAncestor as transitive property. A transitive property interlinks two individuals A and C whenever it interlinks A with B and B with C for some individual B. */

B. Functional Property

A function property is one for which there can be just one value. Genealogy seems to lend itself here, as it does for many of these examples. A person can have just one biological parent, so hasMother (or, more precisely, the subproperty hasBiologicalMother) should be marked functional. Likewise, a social security number is (or, should be) unique. If two entities have the same social, we can reasonably draw an inference that these two entities refer to the same person.

- David Harris hasMother Julie Harris
- David Harris hasMother Julee Harrys
- Julie Harris owl:sameAs Julee Harrys

The owl:FunctionalProperty provides us with a means of inference that two resources are the same, even if the explicit relationships in our graph might say otherwise. ssn-name rdf:type owl:FunctionalProperty

We might consider the rdf:type property as a functional property. Consider the use of is-a pattern extraction on unstructured data in a corpus. The following triples are extracted:

/*France rdf:type placeFrance rdf:type countryFrance rdf:type land */ Slovakia rdf:type placeSlovakia rdf:type countrySlovakia rdf:type land

We might start to infer that (place == country == land) with some degree of confidence.

Consider a property hasHusband. As every person can have only one husband (which we take for granted for the sake of the example), every individual can be linked by the hasHusband property to at most one other individual. Note that this statement does not require every individual to have a husband, it just states that there can be no more than one. Moreover, if we additionally had a statement that Mary’s husband is James and another that Mary’s husband is Jim, it could be inferred that Jim and James must refer to the same individual.

C. Inverse Functional Property

In Allemang and Hendler’s seminal work on Ontology Modeling, inverse functional properties are described as being perhaps the most important of all modeling constructs in RDFS-Plus, particularly in situations where a model is being used to manage data from multiple sources. This property is simply the inverse of the functional property.

Given this example:

- David Harris hasId 123-45-6789
- Robert Harrys hasId 123-45-6789
- David Harris owl:sameAs Robert Harrys

Rather than the directed relationships of the functional property:(X, Y) and (X, Z) giving (Y == Z) we have (Y, X) and (Z, X) giving (Y == Z).

In relational database the inverse functional property plays similar role as key field. The key field must be unique and it can not be duplicate in more than one row. The single value property can not be shared between two entities.

/*This property plays a similar role as a key field in a relational database. A single value of the property can not be shared between two entities, just as a key field can not be duplicated in more than one row. It is important to note that RDFS-Plus does not signal an error if two entities are found to share a value for an inverse functional property. Instead, an inference is enabled that these two entities must be the same*/

Using functional property we can assign at most one distinct value to any given individual. Where as in case of inverse functional property, we can not assign two different individualthe same value. An inverse functional property can also be used as a “key” property

/* If a property is functional, then at most one distinct value can be assigned to any given individual via this property. An inverse functional property can be regarded as a ”key” property, i.e. no two different individuals can be assigned the same value via this property.*/

D. Symmetric Property

The symmetric property is easily understood. As the name suggests, this implies a relationship is bi-directional, even if the relationship was only modeled in one direction.

- Rob siblingOf Bob
- Bob siblingOf Rob

In some cases, the symmetry property and its inverse coincide, i.e the direction of a property doesn't matter. For instance the property hasSpouse relates X with Y exactly if it relates Y with X. For obvious reasons, a property with this characteristic is called symmetric.

/*In some cases, a property and its inverse coincide, or in other words, the direction of a property doesn't matter. For instance the property hasSpouse relates A with B exactly if it relates B with A. For obvious reasons, a property with this characteristic is called symmetric.*/

E. Asymmetric Property

The OWL 2 construct AsymmetricObjectProperty allows it to be asserted that an object property expression is asymmetric - that is if the property expression OPE holds between the individuals x and y, then it cannot hold between y and x. Note that asymmetric is stronger than simply not symmetric.[9, 4]

Stewie hasParent Peter Peter hasParent Stewie

From the OWL 2.0 Primer On the other hand, a property can also be asymmetric meaning that if it connects A with B it never connects B with A. A note was added in the RDF-based semantics rec pointing out that a property being asymmetric is a much stronger notion than its being nonsymmetric, and that being symmetric is a much stronger notion than being non-asymmetric.

F. Reflexive Property

The reflexive property is such a property realtes everything to itself. For example, note that everybody has himself has arelative. Note that this does not necessarily mean that every two individuals which are related by a reflexive property are identical.

Use of the reflexive property allows to cover relationhimself. In a social network, Peter knows JimBob. Use of the reflexive property allows us to cover the obvious case Peter knows Peter and JimBob knows JimBob.

Peter knows JimBob Peter knows Peter

From the OWL 2.0 Primer: Properties can also be reflexive: such a property relates everything to itself. For the following example, note that everybody has himself as a relative. Note that this does not necessarily mean that every two individuals which are related by a reflexive property are identical.

The reflexive property is used frequently in partonomy car is a part of a car /* A property P is said to be reflexive when the property must relate individual to itself. We can see an example of this: using the property knows, an individual George must have a relationship to itself using the property knows. In other words, George must know herself. However, in addition, it is possible for George to know other people; therefore the individual George can have a relationship with individual Somon along the property knows*/

G. Irreflexive Property

If a property P is irreflexive, it can be described as a property that relates an individual X to individual Y, where individual X and individual Y are not the same. An example of this would be the property is MotherOf: an individual Alice can be related to individual Bob along the property isMotherOf, but Alice cannot be the mother of herself.

Alice isMotherOf Bob Alice isMotherOf Alice

From the OWL 2.0 Primer: Properties can furthermore be irreflexive, meaning that no individual can be related to itself by such a role. A typical example is the following which simply states that nobody can be his own parent. Property Chains

Property chains are used to relate various categories the father of your father is your grandfather the wife of your brother is your sister-in-law the son of your sister is your nephew This works great for genealogies and I suspect thats what it was created for. Im certain there are other uses to it seems like a convenient property. A property chain is similar to a functor. There's no need to understand functors in order to appreciate the value of this property, but the mathematical parallel is evident.

How does this differ from a Transitive property? This is a similar but it involves an additional

property being inferred(rather than an extension of the existing relationship).

rdfs:subPropertyOf hasGrandfather; owl:propertyChain (hasFather hasFather). John III hasFather John JR John JR hasFather John SR John III hasGrandfather John SR

Also of note this is not limited to just two triple as shown above. A property chain can be enacted over two or more triples . Whether a property be enacted over an existing property chain is something I'm not clear about myself.

Is this scenario valid? hasGrandfather of hasFather = hasGreatGrandfather Im not certain myself, but would appreciate insight.

From the OWL 2.0 Primer: While the last example from the previous section implied the presence of an hasAncestor property whenever there is a chain of hasParent properties, we might want to be a bit more specific and define, say, a hasGrandparent property instead. Technically, this means that we want hasGrandparent to connect all individuals that are linked by a chain of exactly two hasParent properties. In contrast to the previous hasAncestor example, we do not want hasParent to be a special case of hasGrandparent nor do we want hasGrandparent to refer to great-grandparents etc.

H. Transitive + Symmetrical

Synonyms can be described as both symmetrical and transitive.

power on hasSynonym turn on turn on hasSynonym switch on power on hasSynonym switch on switch on hasSynonym turn on switch on hasSynonym power on turn on hasSynonym power on Using these two properties in conjuction on the hasSynonym predicate creates an explosion of implicit triples. Turn on is set as a synonym for Power on, and switch on for turn on. Given the predicate properties that are checked here, all of these words are now synonyms of each other. Power on and Switch on have no direct relationship in the explict world, but are related symmetrically via turn on. Note that while a synonym is both transitive and symmetric, an acronym is neither. Digital Video Disc has Acronym DVD Acronyms are typically not transitive (this would imply there was an acronym that represented an acronym). If the acronym was symmetric, this would the same as saying DVD hasAcronym Digital Video Disc Which would likewise be incorrect. It has been said that there are no exact synonyms in the English language; every variation has a subtle difference in meaning (perhaps given the origins of either Germanic- Saxon, Anglo-Norman or Latin). However, the predicate does not need to reflect this nuance, unless the modeler so chooses.

I. Reflexive + Transitive + Symmetrical

We could create a relationship called isRelation and define it as reflexive, transitive and symmetric.

Gordon Jr isRelated Gordon Sr Gordon Sr isRelated Harold Gordon Jr isRelated Gordon Jr Gordon Jr isRelated Harold Harold isRelated Gordon Sr Hardold isRelated Gordon Jr Harold isRelated Harold Gordon Sr isRelated Gordon Sr Gordon Sr isRelated Gordon Jr

J. Transitive + Reflexive + Asymmetric

In mereology, the partOf relation is defined to be transitive, reflexive, and asymmetric.

Classic Mereology (Note - this section quotes liberally from Alan Rector and Chris Welty's W3C note on part-whole relations.)

The classic study of parts and wholes, mereology, has three axioms: the part-of relation is Transitive "parts of parts are parts of the whole If A is part of B and B is part of C, then A is part of C Reflexive "Everything is part of itself" A is part of A Antisymmetric "Nothing is a part of its parts" if A is part of B and A != B then B is not part of A. OWL does not have built-in primitives for antisymmetric or reflexive properties. A note was added in the RDF-based semantics rec pointing out that a property being asymmetric is a much stronger notion than its being non-symmetric, and that being symmetric is a much stronger notion than being non-asymmetric.

In mereology, since everything is a part of itself, it is necessary to define "proper parts" as "parts not equal to the whole". Whereas in OWL we have to do the reverse: i.e.define "parts" (analogous to "proper parts") and then define "reflexive parts" in terms of "parts".

A number of other relations follow the same pattern as faults, e.g. "Repairs on a part are kinds of repairs on the whole". However, not all relations follow this pattern, e.g. "Purchase of a part is not purchase of the whole" (you can buy the wheels off a car without buying the car). Mechanic repair wheels

mechanic repair car buyer purchase wheels buyer purchase car

V. FIELD BASED INDEXING

Field based indexers stores the terms with its position and fields where it is present in the document. *Apache Lucene* is an open source field based indexer under Apache. Lucene is built to mainly handle two things :

- It creates search indexes.
- It searches for content in those indexes.

An index is a efficiently navigable representation of whatever data available to make searchable. The data might be as simple as a set of word documents in a content management system, or it might be records from a database, HTML pages, or any kind of data object in your system. It's up to the user to decide what entities you want to make searchable. For our discussion, we'll assume that we are working with a set of Word documents.

A. Create the Index

So, step one is to create the index for our set of Word documents. To do this, we need to write some code that takes the information from the Word documents and turns them into a searchable index. The only way to do this is by brute force. We'll have to iterate over each of the Word documents, examining each and converting each into the pieces that Lucene needs to work with when it creates the index. What are the pieces that Lucene needs to create the index? There are two.

- Documents
- Fields

These two abstractions are so key to Lucene that Lucene represents them with two top level Java classes, Document and Field. A Document, not to be confused with our actual Word documents, is a Java class that represents a searchable item in Lucene. By searchable item, we mean that a Document is the thing that you find when you search. It's up to you to create these Documents.

It is a pretty clear step from an actual Word document to a Lucene Document. It will be the Word documents that our users will want to find when they conduct a search. This makes our processing rather simple, we will simply create a single Lucene Document for each of our actual Word documents.

B. Create the Document and its Fields

But how do we do that? It's actually very easy. First, we make the Document object, with the new operator – nothing more. But at this point the Document is meaningless. We now have to decide what Fields to add to the Document. This is the part where we have to think. A Document is made of any number of Fields, and each Field has a name and a value. That's all there is to it.

Two fields are created almost universally by developers creating Lucene indexes. The most important field will be the "content" field. This Field that holds the content the Word document for which we are creating the Lucene Document. Bear in mind, the name of the Field is entirely arbitrary, but most people call one of the Fields "content" and they stick the actual content of the real world searchable object, the Word document in our case, into the value of that Field. In essence, a Field is just a name: value pair.

Another very common Field that developers create is the"title" Field. This field's value will be the title of the Word document. What other information about the Word document might we want to keep in our index. Other common fields are things like "author", "creation date", "keywords", etc. The identification of the fields that you will need is entirely driven by your business requirements.

So, for each Word document that we want to make searchable, we will have to create a Lucene Document, with Fields such as those we outlined above. Once we have created the Document with those Fields, we then add it the Lucene index writer and ask it to write our Index. That's it! We now have a searchable index. This is true, but we may have glossed over a couple of Field details. Let's take a closer look at Fields.

C. Field Details: Stored or Indexed?

A Field may be kept in the index in more than one way. The most obvious way, and perhaps the only way that you might at first suspect the existence of, is the searchable way. In our example, we fully expect that if the user types in a word that exists in the contents of one of the Word documents, then the search will return that Word document in the search results. To do this, Lucene must index that

Field. The nomenclature is a bit confusing a first, but, note, it is entirely possible to ”store” a Field in the index without making it searchable. In other words, it’s possible to ”store” a Field but not ”index” it. Why? You’ll see shortly.

The first distinction that Lucene makes between the way it can keep a Field in the index is whether it is stored or indexed. If we expect a match on a Field’s value to cause the Document to be hit by the search, then we must index the Field. If we only store the Field, it’s value can’t be reached by the search queries. Why then store a Field? Simple, when we hit the Document, via one of the indexed fields, Lucene will return us the entire Document object. All stored Fields will be available on that Document object; indexed Fields will not be on that object. An indexed Field is information used to find an Document, a stored Field is information returned with the Document.

Two different things. This means that while we might not make searches based upon the contents of a given Field, we might still be able to make use of that Field’s value when the Document is returned by the search. The most obvious use case I can think of is a ”url” Field for a web based Document. It makes no sense to search for the value of a URL, but you will definitely want to know the URL for the documents that your search returns. How else would your results page be able to steer the user to the hit page? This is a very important point: a stored Field’s value will be available on the Document returned by a search, but only an indexed Field’s value can actually be used as the target of a search. Technically, stored Fields are kept within the Lucene index. But we must keep track of the fact that an indexed Field is different than a stored Field. Unfortunate nomenclature. This is why words matter. They can save on a lot of confusion.

D. Indexed Fields: Analyzed or Not Analyzed?

For the next wrinkle, we must point out that an indexed Field can be indexed in two different fashions. First, we can index the value of the Field in a single chunk. In other words, we might have a ”phone number” Field. When we search for phone numbers, we need to match the entire value or nothing. This makes perfect sense. So, for a Field like phone number, we index the entire value ATOMICALLY into the Lucene index.

But let’s consider the ”content” Field of the Word document. Do we want the user to have to match that entire Field? Certainly not. We want the contents of the Word document to be broken down into searchable tokens. This process is know as analyzation. We can start by throwing out all of the unimportant words like, ”a”, ”the”, ”and”, etc. There are many other optimizations we can make, but the bottom line is that the content of a Field like ”contents” should be analyzed by Lucene. This produces a targeted lightweight index. This is how search becomes efficient and powerful. In the APIs, this comes down to the fact that when we create a Field, we must specify

- Whether to STORE it or not
- Whether to INDEX it or not

If indexing, whether to ANALYZE it or not Now, we should be clear on the details of Fields. Importantly, we can both store and index a given Field. It’s not an either or choice.

VI. SEMREL: INDEXING FRAMEWORK

The idea is to use the field based indexer for indexing the semantic relations that we described in the last section. We adapt the field based indexer and its position information to store the semantic relations and retrieve them efficiently. Every semantic relation has a 3 parts *viz.*, 2 entities and a relation. For every semantic relation that need to be indexed there will be three entries in the index table for each of the 3 parts in the relation. The field names that are indexed for each relation are *entity* and *relation*. For example the relation *John isRelated Gordon* has *john*, *isrelated* and *gordon* being indexed as seperate entries in the index with the same document id. For the term *john* the field value is entity and the position value is 1 whereas for *gordon* the field value remain the same but the position value is changed to 2. For the term *isRelated*, the field value is relation and position is 0 (since every relation is considered as a document in itself).

This adaptation of generic indexing framework to semantic relation is novel and easy. The major advantage of this adaption lies in ease of indexing a semantic relation. This framework is extendable with the normal inverted index. Since we do not do any change in the internals of lucene, the same

index could be used to store both semantic relations and generic terms.

A. Posting lists

Every term in the relation will have a posting list of all the relations in which the term is present and in what position. The relation id is auto generated on first come first service approach. Every relation in our index is like a document in a generic inverted index.

B. Index Size

For very relation, we have three terms to be indexed (2 entities and 1 relation). So the number of terms indexed for *n* relations is 3*n*. However, the index size depends on the length of the posting lists. So, if there are *d* documents in the collection with document having *k* relations, then there exists the *Ed* entries in the index. If a term *t* is present in the *nd* number of documents as a part of any relation present in that document then the length of the posting list for the term *t* is *Lt*.

The total number of terms being indexed *ti* is

$$t_i = 3 * n * \Sigma i_t \quad \text{..... (1)}$$

where *it* is the indicator variable for a term *t* which gives true if the term is unique or not.

C. Relation Retrieval

In this section, we discuss how the semantic relations are retrieved from our index. The index could be queried in a natural language and do not require any structured query language like SPARQL. The natural language query is processed and every term in the query is fed to index for retrieval. Every term has a posting list of all the relations and hence we will pick all the relations in which one or both the query terms are present. This would put the restriction on the number of terms in the query to be maximum three. However this limitation is not very restrictive because it is easy to modify the retrieval process to fetch the relations for every query term independently. Like any generic inverted index, we rank those relations higher in which multiple query terms are present.

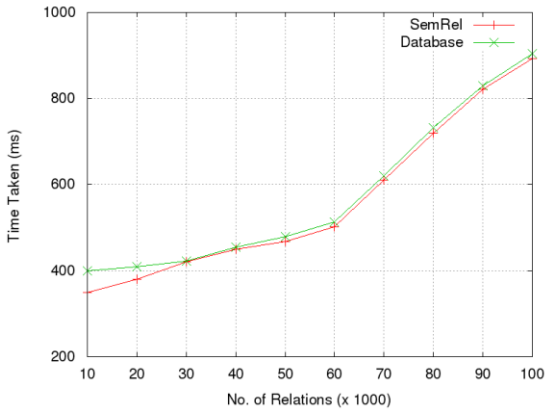

Figure 1: Time Taken for Indexing

VII. EXPERIMENTS AND RESULTS

The experiments conducted in this work involve indexing dbpedia documents that are in the RDF form. DbPedia is a collection of semantic relations extracted from Wikipedia documents. The semantic relations present in the dbpedia were ranked based on their usage and top 100k relations were indexed in the SemRel framework. The time taken for indexing as shown in the figure 1. In the graph 1 it is evident that as we start indexing more and more relations the overhead of indexing becomes negligible. This is in contrast to indexing relations as separate entry in a database. The major benefit of the framework is in retrieval process and we see that retrieval is straight forward as any term based retrieval engine. We can directly search for a term in all the three fields mentioned. For example if we want to search for a term X in field 2 (a relation) and term Y in field 3 (a second entity) then the query formed will be +f2:X +f3:Y. Here '+' indicates the condition MUST on the associated term. The index retrieval would produce all the results matching the template "{Any Entity}X Y".

VIII. CONCLUSIONS

This work proposes a framework to index and retrieve semantic relations efficiently by adapting the open source lucene index. We propose a novel method where every relation is considered as the document and terms in the relation are indexed, For every relation we index three terms associated with the relation and the position information for the same would be decided based on whether the term is subject or object in the relation.

REFERENCES

[1] A. Atreya, S. Chaudari, P. Bhattacharyya, and G. Ramakrishnan. "Building multilingual search index using open source framework," in *24th International Conference on Computational Linguistics*, 2012, p. 201.

[2] S. Kara, O¨ . Alan, O. Sabuncu, S. Akpınar, N. K. Cicekli, and F. N. Alpaslan. "An ontology-based retrieval system using semantic indexing," *Information Systems*, vol. 37, no. 4, 2012, pp. 294–305.

[3] S. S. Kamath, D. Piraviperumal, G. Meena, S. Karkidholi, and K. Kumar. "A semantic search engine for answering domain specific user queries," in *Communications and Signal Processing (ICCSP), 2013 International Conference on*. IEEE, 2013, pp. 1097–1101.

[4] T. Berners-Lee, J. Hendler, O. Lassila *et al.*, "The semantic web," *Scientific American*,2001, vol. 284, no. 5, pp. 28–37.

[5] V. Lopez, M. Fern´andez, E. Motta, and N. Stieler, "Poweraqua: Supporting users in querying and exploring the semantic web," *Semantic Web*, vol. 3, no. 3, 2012, pp. 249–265.

[6] T. Grainger and T. Potter, *Solr in action*. Manning Publications Co., 2014.

[7] A. Harth and S. Decker, "Optimized index structures for querying rdf from the web," in *Web Congress, LA-WEB 2005. Third Latin American*. IEEE, 2005, pp. 10.

[8] M. Alavi and D. E. Leidner, "Review: Knowledge management and knowledge management systems: Conceptual foundations and research issues," *MIS quarterly*, 2001, pp. 107–136.

[9] I. L. Janis and L. Mann, *Decision making: A psychological analysis of conflict, choice, and commitment.* Free Press, 1977.

[10] D. L. McGuinness, F. Van Harmelen *et al.*, "Owl web ontology language overview," *W3C recommendation*, 2004, vol. 10, no. 10.

[11] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, M. Dean *et al.*, "Swrl: A semantic web rule language combining owl and ruleml," *W3C Member submission*,2004, vol. 21, p. 79.

[12] A. Ankolekar, M. Burstein, J. R. Hobbs, O. Lassila, D. Martin, D. McDermott, S. A. McIlraith, S. Narayanan, M. Paolucci, T. Payne *et al.*, "Daml-s: Web service description for the semantic web," in *The Semantic WebISWC 2002*. Springer, 2002, pp. 348–363.

[13] M. Klein, "Interpreting xml documents via an rdf schema ontology," in *Database and Expert Systems Applications, 2002. Proceedings. 13th International Workshop on*. IEEE, 2002, pp. 889-893.