SemTree: an index for supporting semantic retrieval of documents

Flora Amato[1], Aniello De Santo[2], Francesco Gargiulo[3], Vincenzo Moscato[1], Fabio Persia[1], Antonio Picariello[1], Silvestro Roberto Poccia[1]

Dipartimento di Ingegneria Elettrica e Tecnologie dellInformazione, University of Naples "Federico II"
via Claudio 21, 80125, Naples, Italy
[1]{flora.amato,vmoscato,fabio.persia,picus,silvestroroberto.poccia}@unina.it
[2]aniello.desanto@gmail.com
[3]f.gargiulo@cira.it

***Abstract*—In this paper, we propose *SemTree*, a novel semantic index for supporting retrieval of information from huge amount of document collections, assuming that semantics of a document can be effectively expressed by a set of ⟨subject, predicate, object⟩ statements as in the RDF model. A distributed version of KD-Tree has been then adopted for providing a scalable solution to the document indexing, leveraging the mapping of triples in a vectorial space. We investigate the feasibility of our approach in a real case study, considering the problem of finding inconsistencies in documents related to software requirements and report some preliminary experimental results.**

I. Introduction

In the last decade digital data are grown in a large scale and in an exponential way in different application fields. Such data are often in the shape of textual documents containing unstructured, structured and/or semi-structured information (e.g. medical records can include both structured information, as example patient vital statistics, and unstructured text as medical reports; web pages can include both structured information in the form of HTML microdata and unstructured text).

Within modern organizations, managing efficiently and effectively very large amount of digital documents has become a more and more important challenging aspect for a wide range of knowledge management applications. This process requires from one hand a model for representing semantics attached to any kind of document (e.g. web pages, medical records, logs and more in general each type of textual documents), and from the other one indexing techniques to support in a scalable manner document retrieval in very large distributed and heterogeneous repositories based on the underlying semantics.

In the literature, the most widely used approaches that combine Knowledge Representation and Natural Language Processing techniques to allow an efficient semantic-based retrieval are: *conceptual indexing*, *query expansion*, and *semantic indexing* [1].

The systems leveraging the conceptual indexing approach usually ground on catalogs of texts belonging to specific domains and exploit ad-hoc ontologies and taxonomies to associate a conceptual description to documents [2].

Differently from the previous ones, the systems using query expansion techniques semantically enrich the user query adding words that have semantic relationships (e.g. synonyms) with the search terms [3].

Eventually, the systems based on semantic indexing techniques exploit the meaning of documents' keywords to perform indexing operations [4], eventually using a latent analysis [5].

In this paper, we propose a novel semantic indexing approach for supporting retrieval of documents from digital collections.

We assume that semantics of a document can be effectively expressed by a set of *assertions* in the shape of $\langle subject, predicate, object \rangle$ statements (which relates a subject to an object by means of a predicate) as in the RDF model, allowing to represent both parts containing structured (whose transformation in a set of triples is immediate) and unstructured information, which requires NLP facilities [6].

SemTree, a distributed version of *Kd-Tree*, has been then adopted for providing a scalable solution to the document indexing, leveraging the mapping of triples in a vectorial space by means of the definition of a proper *semantic distance* between triples. The distance uses some domain specific and/or general vocabularies and can be computed through the most diffused semantic similarity measures [9].

The problem of supporting in a scalable way the fast retrieval of triples related to various pattern queries by translating them into multi-dimensional range queries has been also faced in [7]. Differently from our approach, the authors adopt an *RDF store* that is built on the top of a distributed multi-dimensional index structure.

Furthermore, scalability issues are met in a similar manner to our method by the proposal in [8], wherein a multi-dimensional indexing structures layered over a *key-value* database is opportunely used to obtain an efficient query processing for location based services.

Finally, we investigate the feasibility of our approach in a real case study, considering the problem of finding *inconsistencies* within a large amount of documents related to software requirements.

978-1-4799-8442-8/15/$31.00 © 2015 IEEE

The paper is organized as in the following. Section 2 describe the motivating example of our work, while Section 3 presents the overall framework for semantic indexing. Section 4 illustrates the performed experiments on a distributed environment. Finally, Section 5 shows conclusions and future work.

II. Motivating Example

Recent studies on requirement engineering demonstrate that software teams still face problems in the formalization of textual requirements, in the verification of consistency of requirements not yet formalized, and even in the definition of specification models [10].

To this goal, some works have recently proposed an approach to verify *inconsistencies* in Natural Language documents, leveraging the adoption of similarity measures between concepts modeled using RDF [11]. Following this idea, we model each software requirement as a set of triples and then we find possible contradictions and conflicts within this set through the detection of appropriately defined patterns, expressed as particular *target triples*.

In requirement engineering, two triplets t_i and t_j are *inconsistent* if: (i) they have the same subject, (ii) they have the same object; (iii) the two predicates are linked by an *antinomy* relationship in a given vocabulary [11]. Leveraging such property, we may query the documents using as target triples possible inconsistent requirements, in order to detect all semantically similar triples that could then correspond to contradictions or conflicts.

As an example, if we consider the following requirement ⟨OBSW001, accept_cmd, start-up⟩ for an airplane on-board software (specifying that the command 'start-up' has to be accepted by the software component 'OBSW0054'), then possible inconsistencies can be retrieved in the result set related to the query triple ⟨OBSW0001, block_cmd, start-up⟩ that contains all the triples "semantically close" to the target one.

Furthermore, since a requirement contains more than one sentence and a sentence can include several triples, even the number of triples generated by a small set of requirements is considerable. Therefore, there is a strong need for a framework to efficiently implement semantic queries (i.e. range query and k-nearest query) on large set of documents.

The purpose of our semantic index is to give an effective solution to this kind of problems.

III. A Framework for semantic indexing

A. Background

We propose a framework for big data indexing based on the following features: (i) the semantics attached to each document can be represented as a set of triples; (ii) it is possible to define "semantic distance" between two triples; (iii) the triples, together with related distances, are mapped into a vectorial space - using *FastMap* algorithm [12] - on which it is possible to define an efficient indexing structure; (iv) the index is designed to work in a distributed environment supporting more efficiently query by examples on large set of data.

In our model, data are represented by means of a set of *assertions* reflecting specific data properties (i.e. *predicates* with a *subject* and an *object*, as in RDF model).

Following our motivating example, data come from software requirements' documents that are expressed in Natural Language and that are composed by a set of sections, each one containing the definition of a specific requirement. In this case, the predicates correspond to particular unary "functions" (i.e., 'accept a command', 'send a message', 'acquire an input', etc.), having as subject the *Actor* (software component or hardware device) and as object the related *Parameter*.

An example of resources (in a *Turtle*-like format) that could be derived from such data source is described in the following[1].

⟨'OBSW001', Fun:acquire_in, InType:pre-launch phase⟩
⟨'OBSW001', Fun:accept_cmd, CmdType:start-up⟩
⟨'OBSW001', Fun:send_msg, MsgType:power amplifier⟩

The notation $X : x$, expresses that the meaning of the concept x can be found by using the prefix X. If X is not specified, we use a standard vocabulary.

Clearly, *similar* resources contain the triples that are able to express similar information content. In this paper we are not interested in how it is possible to transform documents into a set of assertions/triples (techniques to map relation data to RDF or NLP facilities to transform a text in a set of triples can be easily exploited [6]); in the opposite, we focus on how to compute the distance between two generic triples, eventually considering the related semantics.

To evaluate a distance between two triples t_i and t_j, we consider the following formula:

$$d(t_i, t_j) = \alpha \cdot d_s(t_i^s, t_j^s) + \beta \cdot d_p(t_i^p, t_j^p) + \gamma \cdot d_s(t_i^o, t_j^o) \quad (1)$$

t_k^s, t_k^p , t_k^o being the projection of a triple t_k on the one subject, predicate and object respectively, $d_s(t_i^s, t_j^s)$, $d_p(t_i^p, t_j^p)$, $d_o(t_i^o, t_j^o)$ the distances between triples' subjects, predicates and objects respectively, α, β, γ are set of weigths such that $\alpha + \beta + \gamma = 1$.

To compute the specific sub-distances, we have to consider two main cases: (i) the two triples' elements are both literals/constants of the same type (we can apply any distance function between strings, i.e. *Levenshtein*); (ii) the two triples' elements are both concepts (we can apply any distance semantic based on the available ontologies, taxonomies or vocabularies, i.e. *Wu & Palmer*).

B. SemTree algorithms

A variety of indexing data structures are available to deal with high-dimensional data: R-tree, Kd-tree, X-tree, SS-tree, M-tree, Quadtree, etc.

[1]The order of the triples reflect the temporal sequence of the requirement elements.

Kd-trees are more efficient in bulk-loading situations (as required by our approach), they can adapt to different densities in various regions of the space and are easier to implement in memory (which actually is their key benefit). On the other hand, once built, modifying or rebalancing a Kd-tree is a non-trivial task.

Here, we present SemTree, a distributed index particularly suitable for managing semantic extracted data. In the following, we describe the algorithm for *dynamic insertion* of new points into SemTree and present *k-nearest* and *range query* procedures; in addition, we assume that our data can be stored only into the leaf nodes. Each tree node can be either a *routing* or a *leaf* node.

1) The Distributed Insertion Algorithm: this algorithm has been designed for performing an insertion of a point into SemTree, whose data structure can be distributed through different *partitions* usually managed by a single compute node. When a new point has to be added into a partition of the tree, there is the need to verify a specific *condition* based on the available resources of the specific compute node. The condition can be dynamically evaluated at run-time – for example, it may depend on the percentage of the available storage resources of each partition – or statically fixed.

When there are not enough resources into the selected partition, new partitions are created invoking a specific procedure, as will be described in the following. The purpose of this procedure consists in moving each leaf node of the current partition into a different newly created partition.

The insertion procedure starts from the root node of the root partition of SemTree; then, the tree is navigated with the aim to locate the leaf node in which the point has to be inserted.

More in details, SemTree navigation is performed on the basis of the S_I (the Split Index of the current node) and S_V (the value of the node splitting variable) variables, as in the standard Kd-Tree [13]. The result of the comparison between the S_I coordinate of the point P (P[S_I]) and S_V at each level allows to navigate in the left/right sub-tree of the current node; if such a sub-tree is not located on the same partition of the current node, a message containing the point to be added has to be sent to the correct partition.

When a leaf node l_n (the blue node in the left sub-figure of Figure 1) saturates the bucket, two new child nodes are instantiated using an apposite function. Because l_n is no longer a leaf node, the related points are moved into the new child nodes (the red nodes in the right sub-figure of Figure 1).

In a given partition, we can distinguish between *internal* and *edge* nodes. Each leaf is an edge node, but a routing node can be either an *internal* or an *edge* node. A routing node is an *edge* node when at least one of its children is the root in a different partition; on the contrary, it is an *internal* when all its children are on the same partition.

The tree navigation through different partitions takes place by a proper communication protocol (in our implementation based on MPJ libraries).

When a routing node is reached, the sub-tree in which to delegate the insertion operation is determined.

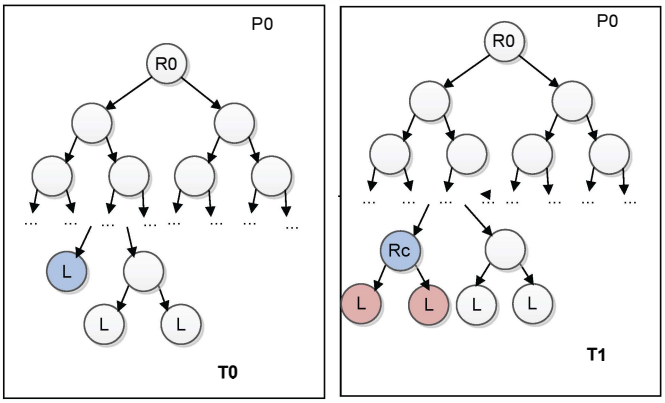

Fig. 1: Example of SemTree distributed insertion

Once the target sub-tree is identified, we need to determine whether it is on the same partition of the routing node by comparing two proper variables C_P (the identifier of the partition hosting the current node) and $Child_P$ (the identifier of the partition hosting the child node of the current node). When $C_P = Child_P$, the tree is navigated as a sequential Kd-Tree; otherwise, we insert the point into the partition that contains the child node.

2) The Build Partition Algorithm: this functionality allows us to build new partitions if there are some available compute nodes able to host them. This function starts when the maximum allowed number of resources of a partition is reached; the build partition algorithm starts from the root node of such a partition and goes on navigating the tree to find each leaf in the partition and to move them into a new different partition. The result is a partition tree in which some partitions are used just for routing and others for storing data.

A direct link between different partitions is instantiated in order to allow the navigation across the partitions. For instance, Figure 2 shows how a leaf node candidate L_C is moved to a newly created partition and a link between the two partitions is then created. This process is replicated on each leaf node exceeding the allowed number of storage resources.

3) The Distributed K-nearest Search Algorithm: this algorithm describes the solution to the *k-nearest query* problem on SemTree.

The navigation of the tree is similar to the previously described insertion algorithm. In a *k-search* we look for the *k* points closest to the one of interest. To illustrate the *distributed k-search algorithm*, we introduce the parameters listed in table I. The tree is navigated until a leaf node is reached and its bucket points are added to the result set; then we start going backward to the tree.

During such a kind of visit, we evaluate, for each node, weather to start forward visiting the not yet analyzed sub-tree having the current node as root. This condition, according to the well known Kd-Tree sequential k-nearest search, is composed by the logical disjunction between two sub-conditions, the former based on a comparison between dis-

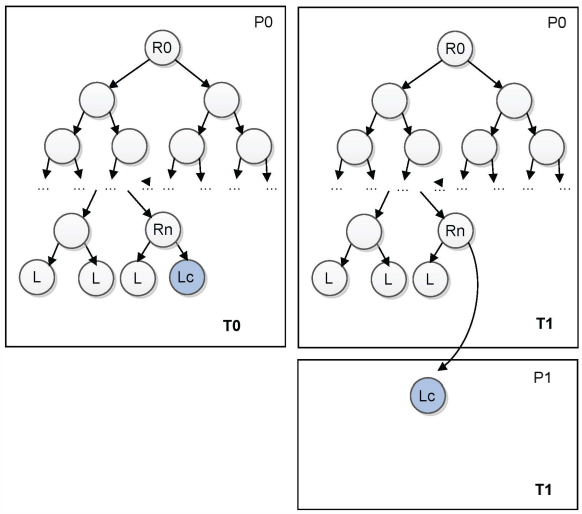

Fig. 2: Build Partition

TABLE I: Imput Parameters of K-Search

Field	Reference	Possible Values
Node Status	S	Not Visited (N_V); Left Visited (L_V); Right Visited (R_V); All Visited (A_V).
Number of Points	K	The number of points that we have to find.
Distance	D	The distance between the interested point and the most distant one in the result-set.
Result-set	R_S	A structure able to store in memory the k points of interest found.
Point	P	The point of interest.

tances ($|\max(R_S[S_I]) - P[S_I]| > |P[S_I] - S_V|$) ($S_I$ is the space index coordinate and S_V the split value of the current node), and the latter based on the replenishment of R_S against k ($R_S.lenght() < K$).

The protocol for exchanging messages between different partitions is basically the same as the one described in the insertion algorithm. The algorithm ends when the backward visit reaches the root node of the root partition.

4) The Range search Algorithm: in a *Range-search* we search the points that are closest to the query point within a specific range. To illustrate the *distributed Range search algorithm*, we refer to the parameters listed in table I, considering in this case *D* as a distance range.

The navigation strategy is composed by two steps. First, we navigate the tree from the root node to the leaves to find the points within a specific distance range from a point P. During this phase the navigation through the current node is driven by the comparison between $|(P[S_I] - S_V)|$ and the range distance D. We can have two possible cases:

- $|(P[S_I] - S_V)| < D$: in this case we navigate across the two children of the current node; note that, if the current node is a border node, the navigation is performed in a parallel way.
- Otherwise, the navigation is performed with a comparison of the S_V variable of the current node, as in the insertion algorithm.

Second, we navigate backward from the leaves to the root, adding the results to the R_S structure. The different result sets obtained through a parallel computation are merged into a unique one during this phase.

C. Complexity

To calculate the complexity of our algorithm we have to make some assumptions. Let us:

- insert *K* points into the tree;
- be able to use *M* partitions;
- suppose to have a bucket, of size B_S, in each leaf node to store the points;
- suppose an equal distribution of points in each leaf node.

We thus have $N = \frac{2*K}{B_S}$ nodes in the SemTree. Let us consider a Root Partition hosting routing nodes and able to distribute messages between the other partitions. In this case this partition hosts $2 * M - 1$ nodes. If the points are equally distributed, we have $N_l = \frac{N-(2*M-1)}{M-1}$ nodes in every other partition.

When we want to add a point to the tree, the *insertion procedure* starting from the root node of the root partition is invoked: when the tree is well-balanced, the time to navigate the tree to find the correct place is: $\Theta(\log_2(M-1)+\log_2(N_l)+1)$.

If M<<N, we can consider $\log_2 M << \log_2 N$ and $\log_2 M = A$ is a certain number related to the number of used compute nodes. Then, we obtain $\Theta(A + \log_2 \frac{N}{M})$.

This complexity refers to a single insertion; however, using $M-1$ data partitions, we can perform in the best case $M-1$ parallel operations maximizing our throughput.

The complexity of algorithm for bulding partitions is $\Theta(M)$ and is related to the number of machines we are able to manage. Note that if $M << N$, this complexity is not important in tree building process. The search algorithms complexity is the same of the standard Kd-tree.

IV. Experimental results

In this section we evaluate the efficiency and the effectiveness of SemTree.

The efficiency has been calculated considering several case studies, varying the size of the tree and for different types of tree structure [2].

The effectiveness has been computed using the *Precision/Recall* metrics, by comparing the output of our algorithms against ground truth provided by human annotators.

The used dataset is extracted from a set of requirements' documents related to on board software systems. More in details, we deal with several hundreds of documents from which about 100,000 triples were extracted.

[2]All the experiments presented in this Section were conducted on a cluster having 8 processors with 8 GB RAM (compute nodes).

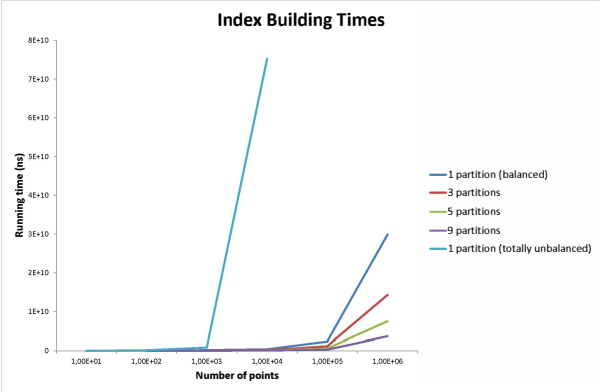

Fig. 3: Index Building Time

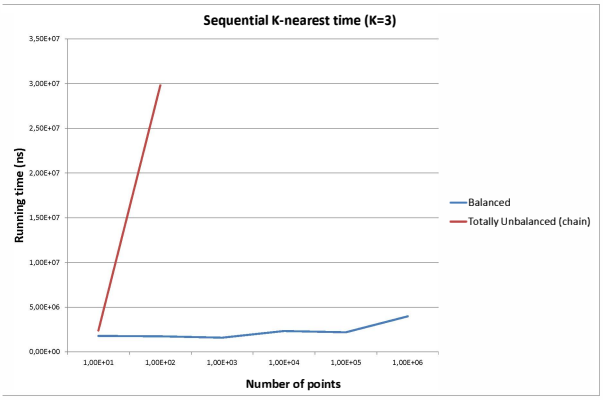

Fig. 4: Sequential K-Nearest Time, K=3

A. *Efficiency computation*

Figure 3 shows the running time achieved by our procedure for index building when varying the size of the input data and the number of used partitions.

In figure 4 the running time of the *sequential K-nearest algorithm* when varying the size of the tree with K fixed to a default value ($K = 3$) are shown, while figure 5 shows the running time of the *distributed K-nearest algorithm* when varying the number of partitions. Similarly, figure 6 shows the running time of our algorithm for *sequential range query* and figure 7 the time of the *distributed range query*. In all the listed case studies, the achieved running time can be considered very good.

B. *Effectiveness computation*

We conducted a preliminary evaluation of SemTree effectiveness with respect to the described case study. In particular, we want to show feasibility of the proposed indexing approach in automatically finding possible inconsistencies among software requirements, expressed as set of triples.

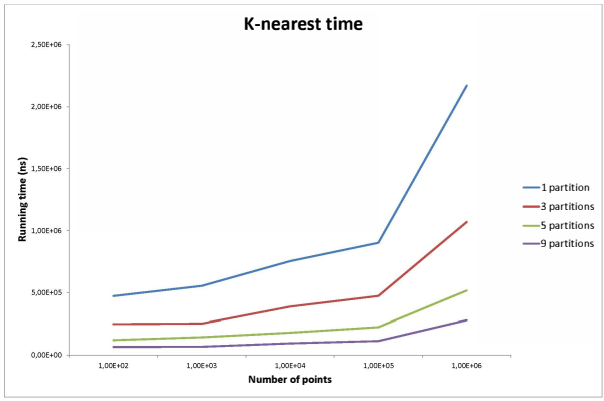

Fig. 5: K-Nearest Time, K=3

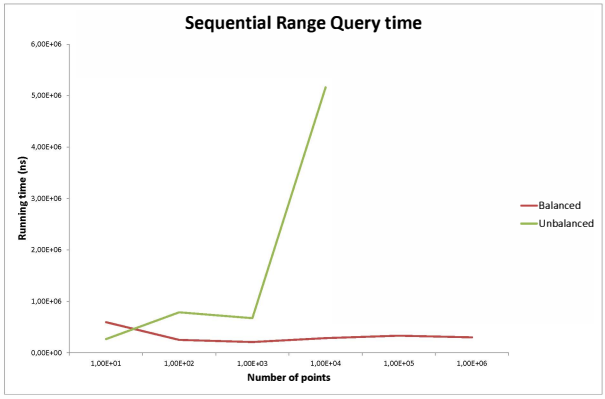

Fig. 6: Sequential Range Query Time

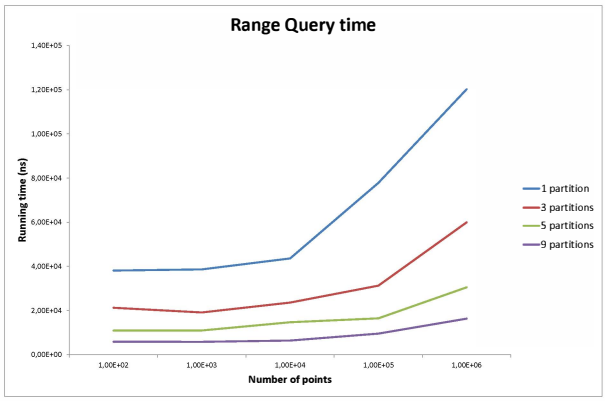

Fig. 7: Range Query Time

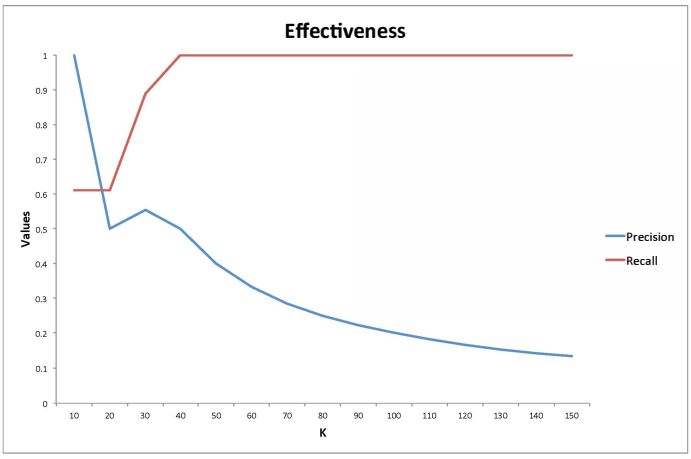

Fig. 8: Effectiveness

To this goal for 100 different requirements, we randomly selected a triple from the related set and generated the equivalent target (query) triple. A target triple was obtained considering subject and object of the selected triple and as predicate an *antinomic* term (retrieved using an ad-hoc requirements vocabulary) with respect to predicate of the selected triple.

Successively, from one hand we asked a group of software engineers [3] to specify - starting from the analysis of requirements documents expressed in terms of set of triples - the set of possible inconsistences (ground truth) for each selected triple.

From the other hand, we performed 100 different K-nearest queries on SemTree using as query points the target triples.

Denoting with T the set of triples returned by the K-nearest query related to a given target triple and with T^* the set of inconsistencies expected by ground truth , *Precision* (P) and *Recall* (R) were computed as follows:

$$P = \frac{|T \cap T^*|}{|T|}$$

$$R = \frac{|T \cap T^*|}{|T^*|}$$

Figure 8 shows the average *Precision* and *Recall* values for the 100 query cases when varying K. As expected, the lower is K, the higher is *P* and the lower is *R*; then, when K increases, *R* grows up and *P* decreases.

V. Conclusion

In this paper, we presented *SemTree*, a novel semantic index for supporting retrieval of information from huge amount of document collections whose semantics can be effectively expressed by a set of ⟨subject, predicate, object⟩ statements as in the RDF model.

A distributed version of KD-Tree was then adopted for providing a scalable solution to the document indexing, leveraging the mapping of triples in a vectorial space on the base of a semantic distance .

We investigated the feasibility of our approach in a real case study, considering the problem of finding inconsistencies in documents related to software requirements.

Preliminary experimental results showed as our approach can be considered promising for the problem of semantic information retrieval.

Future work will be devoted to compare the efficiency and effectiveness achieved by our framework with the ones of other approches well-known in literature, such as [7].

[3] 5 persons working at CIRA Institute (Italian Center for Aerospace Research)

References

[1] P. N. Bennett, E. Gabrilovich, J. Kamps, and J. Karlgren, "Report on the sixth workshop on exploiting semantic annotations in information retrieval (esair'13)," in *ACM SIGIR Forum*, vol. 48, no. 1. ACM, 2014, pp. 13–20.

[2] L. B. Ghezaiel, C. C. Latiri, and M. B. Ahmed, "Conceptual indexing documents in ir based on ontology enrichment." in *KES*, 2012, pp. 1920–1931.

[3] C. Carpineto and G. Romano, "A survey of automatic query expansion in information retrieval," *ACM Computing Surveys (CSUR)*, vol. 44, no. 1, p. 1, 2012.

[4] R. Mihalcea and D. Moldovan, "Semantic indexing using wordnet senses," in *Proceedings of the ACL-2000 workshop on Recent advances in natural language processing and information retrieval: held in conjunction with the 38th Annual Meeting of the Association for Computational Linguistics-Volume 11*. Association for Computational Linguistics, 2000, pp. 35–45.

[5] T. K. Landauer, D. S. McNamara, S. Dennis, and W. Kintsch, *Handbook of latent semantic analysis*. Psychology Press, 2013.

[6] A. dAcierno, V. Moscato, F. Persia, A. Picariello, and A. Penta, "iwin: A summarizer system based on a semantic analysis of web documents," in *Semantic Computing (ICSC), 2012 IEEE Sixth International Conference on*. IEEE, 2012, pp. 162–169.

[7] G. Tsatsanifos, D. Sacharidis, and T. Sellis, "On enhancing scalability for distributed rdf/s stores," in *Proceedings of the 14th International Conference on Extending Database Technology*. ACM, 2011, pp. 141–152.

[8] S. Nishimura, S. Das, D. Agrawal, and A. E. Abbadi, "Md-hbase: a scalable multi-dimensional data infrastructure for location aware services," in *Mobile Data Management (MDM), 2011 12th IEEE International Conference on*, vol. 1. IEEE, 2011, pp. 7–16.

[9] P. Resnik, "Semantic similarity in a taxonomy: An information-based measure and its application to problems of ambiguity in natural language," *arXiv preprint arXiv:1105.5444*, 2011.

[10] G. Fanmuy, A. Fraga, and J. Llorens, "Requirements verification in the industry," in *Complex Systems Design & Management*. Springer, 2012, pp. 145–160.

[11] G. Gigante, F. Gargiulo, and M. Ficco, "A semantic driven approach for requirements verification," in *Intelligent Distributed Computing VIII*. Springer, 2015, pp. 427–436.

[12] C. Faloutsos and K.-I. Lin, *FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets*. ACM, 1995, vol. 24, no. 2.

[13] B. C. Ooi, K. J. McDonell, and R. Sacks-Davis, "Spatial kd-tree: An indexing mechanism for spatial databases," in *IEEE COMPSAC*, vol. 87, 1987, p. 85.