Tesi di Dottorato

Università degli Studi di Napoli "Federico II"

Dipartimento di Ingegneria Elettrica
e delle Tecnologie dell'Informazione

Dottorato di Ricerca in
Ingegneria Informatica ed Automatica

A Semantic Index for Linked open Data and Big Data applications

Francesco Gargiulo

Il Coordinatore del Corso di Dottorato   Il Tutore

Ch.mo Prof. Franco Garofalo   Ch.mo Prof. Antonio Picariello

A. A. 2016–2017

"Ai miei genitori."

Acknowledgments

Desidero ringraziare tutti coloro che mi hanno aiutato nella realizzazione della mia Tesi.

Un ringraziamento particolare va innanzitutto al Professore A. Picariello grazie al quale ho avuto coraggio di intraprendere a 43 anni il percorso di dottorato. E' stata un'esperienza travolgente nella quale ho fatto ricorso alla sua grande competenza e disponibilitÃ e alla sua inesauribile pazienza. Non lo ringraziero' mai abbastanza per il sostegno scientifico ed amicale.

Rigrazio il coordinatore, il prof. F. Garofalo, per l'aiuto e la disponibilita' dimostrata.

Al prof. V. Moscato vanno i miei ringraziamenti per i preziosi insegnamenti e i tanti buoni consigli che ho cercato di recepire nel modo migliore. In lui ho trovato mentore e un compagno di viaggio.

Un affettuoso ringraziamento va poi alla Professoressa F. Amato per gli innumerevoli contributi, per il costante confronto e per i suggerimenti ricevuti.

Non avrei mai potuto concludere questo lavoro se non avessi avuto il sostegno di Nunzia e dei miei ragazzi che mi hanno seguito e mi seguono con affetto e pazienza, incentivandomi anche nei momenti piu' duri.

Desidero ringraziare anche tutte le altre persone con le quali ho discusso e che mi hanno concesso il loro prezioso tempo e che non posso ricordare qui singolarmente. A loro va la mia gratitudine ed il mio affetto. Tutte le persone citate in questa pagina hanno svolto un ruolo fondamentale nella stesura della tesi, ma desidero precisare che ogni errore o imprecisione e' imputabile soltanto a me.

Contents

List of Figures

Introduction

The *Linked Open Data* (LOD) paradigm is gaining increasing attention in recent years. The term *Linked Data* is a set of best practices for publishing and connecting structured data on Web whereas the term *Open Data* refers to data accessible to everyone. The main reasons for the growing interest in the LOD are essentially two.

First, the LOD fits into a broader trend in which the content of the documents is no longer primarily human-readable but it is also machine-readable. This shift in perspective is the basic principle of the Semantic Web. The LOD is designed as a tool conceiving the Web as a data container that can be processed directly or indirectly by machines. The W3C is very active on the this topic and it published a series of standards and committed several groups on related topics.

Secondly, LOD receives a big boost to the its affirmation by Governments of many major countries. The Obama administration enacted a directive in December 2009 regarding Open Government that imposed to the agencies the publication of information in a format that is "open" and "independent" from the platform. In Europe, the British Government was the precursor in publishing their data as Linked Data. The European Community issued directives that go in the same direction and also invested in LOD by funding projects under the 7th Framework Programme (i.e. the project LOD2).

The potential and the benefits of the LOD model are numerous, such as: the dissemination of "collective" knowledge resulting from the combination of several data sources; the extraction of new knowledge and identification of regularities emerging from the analysis of large amounts of data; the use of data for the creation of new economic activities; greater transparency and participation of public policies.

This broadening of perspective in way of accessing information on the Web is often referenced as *Semantic Web*. The possibility of using complex tools to navigate, correlate, analyze and synthesize vast amounts of data opens very

interesting perspectives both economically and scientifically.

After overcoming the legislative constraints regarding the licensing, the use and the ownership of the data, it is reasonable to assume the proliferation of new applications and new services that are able to take advantage of this huge quantity of information. In fact, the size of the LOD phenomenon is considerable, we are talking about hundreds of millions of information published in the form of "concepts" and of billions of connections between these concepts.

On the other hand, from the scientific perspective were observed some limitations of traditional and well-known technologies, such as relational databases, relative to exploitation of data shared with the model LOD and this therefore opened several interesting theoretical and technological problems.

Big data is a term for data sets that are so large or complex that traditional data processing software application are inadequate to deal with them. Basically the main features of Big Data, can be summarized in five "V":

- *Volume*: ability to capture, store, and access large volumes of data;

- *Velocity*: ability to perform data analysis in real time or near;

- *Variety*: refers to different types of data from different sources (structured and unstructured);

- *Variability*: Inconsistency of the data set can hamper processes to handle and manage it;

- *Veracity*: The quality of captured data can vary greatly, affecting accurate analysis.

In the context of business analytics new models of representation that can handle large amount of data with parallel processing databases have emerged. Architectures of distributed processing of large data sets were offered by Google's *MapReduce* and Apache's *Hadoop*. With these frameworks, the elaboration is distributed over several nodes and then it run in parallel (map phase). Then, the results are collected and returned (reduce phase).

There are a number of open challenges related to LOD and Big Data, for example, their characteristics make ineffective and inefficient the use of traditional RDBMS. In fact, these datasets potentially embrace the entire Web and can be provided by various publishers around the world [5]; they are typically based on some well-known vocabularies such as: Dublin Core, FOAF, SIOC,

etc. but they can use also other domain-specific vocabularies [6]; other than the traditional database queries also Information Retrieval and Ranking capabilities are needed with different levels of granularity [7], such as document-based or entity-based search/retrieval capabilities [8]; being predominantly read-only transactions, ACID properties (Atomicity, Consistency, Isolation, and Durability) are not required and this leads to a significant simplification for LOD and Big Data management and consequently to a general improvement of performance [9]. From the foregoing considerations arise actual limitations of RDBMS and then the operators are moving towards the use of systems with high scalability and NoSQL solutions.

First of all, the discovery and exploration of the datasets and therefore the need to index properly resources is certainly the most problematic step already partially addressed in the scientific community of LOD. There are a few browsers and search engines developed for the Linked Open Data, such as: SWSE [10], Falcons [11] and providing search capabilities based on keywords; Disco, Sig.ma, Visinav, Tabulator, Watson, Marbles [12] to navigate the Semantic Web as a set of sources not linked to each other and explore the resources identified in each dataset (in various formats, including Resource Description Framework and Microformats); Another crucial aspect is the identification of common entities in separate datasets; in fact, in addition to using the link "explicit" as relations "SameAs" by OWL, some approaches for automatic discovery of links "sure" or "approximated" between entities that are conceptually identical or similar have been proposed.

One of the greatest benefits resulting of adoption of the paradigm LOD is the possibility of making automatic reasoning combining data sources. This opens entirely new scenarios which brings enormous opportunities for the development of innovative services and on the other hand opens a series of questions about querying and reasoning.

From the conception of the LOD as a set of *Resource Description Framework* RDF documents or as a set of RDF triples-each different approaches to querying and reasoning follow. In any case, it is necessary to consider some compromises (i.e. expressiveness vs. performance) when you design a system that provides functionality for querying and reasoning about LOD.

In the first part of the dissertation, the focus is on the concept of *index* and on the main features that a semantic index for LOD and Big Data applications should have. It illustrates the explanation of the choice of k-d tree as base data structure for the proposed index. The meaning of the term *semantic* in this context is specified and the approach to text indexing is presented. It introduces

the semantic similarity measures and explains how to build a metric space after the extraction of a set of concept from the text. This part illustrates the concept of *mapping* between metric space and vector space, analyze its properties and presents a couple of well-known mapping algorithms. The resulting vector space populates a k-d tree and a description of the execution of a semantic query with this k-d tree follows. Finally, a non-trivial example based on such k-d tree is presented. The first part ends with the analysis of the drawbacks following the application of a naive approach in the distribution of a k-d tree over a network.

The second part of this work starts from the these drawbacks and proposes an approach that solves them. This part and the following are the main contribution of this dissertation. For the sake of simplicity, it considers a simplified version of k-d trees, i.e. binary trees, and proofs two properties of binary tree that allows a novel approach in query processing. In this novel approach, a query starts in a randomly chosen node s of the tree and ends as soon as possible in a node n and, with very high probability, neither s nor n are the root of the tree. Because the root of the tree is the bottleneck of a naive distributed k-d tree then this approach in query processing solves this drawback. This section analyzes a number of variants of the proposed approach and made a comparison among them on order to choose the best tradeoff. This part also formally describes all algorithms implementing the proposal and shows their efficiency in terms of time complexity.

The following part of the dissertation addresses the problem of the allocation of the nodes of the tree over the network peers. It is noted that, in order to ensure the efficiency of the algorithms not only the k-d tree must be a balanced tree but also the peers of the network must form a balanced tree. This part proposes an allocation strategy that ensures this balance.

The remaining part of the work extends of the previous results to k-d trees. It provides the proofs of the two basic properties for k-d trees and formally describes the search algorithms as in previous section.

The outline of the dissertation is the following:

Chapter 1 presents the general background theory. The concept of index is introduced and the requirements of a semantic index for LOD e Big Data are described. Furthermore, this chapter introduces our approach to indexing the text and presents a non-trivial example of the usage of a semantic index. The chapter ends with a discussion about a naive approach in the

distribution of the k-d tree over a network and shows the main drawbacks of this solution.

Chapter 2 moves the first step toward the creation of a distributed version of k-d tree by means the description of the main ideas applied to binary trees. This chapter provides a proof of two basic properties of binary trees on which the proposed novel approach to k-nearest neighbor query is based.

Chapter 3 proposes an allocation strategy that ensures the balancing of the network of peers and illustrates this strategy by an example.

Chapter 4 extends of the previous results to k-d trees. It provides the detailed proofs of the two basic properties for k-d trees and formally describes the search algorithms as in previous section.

Chapter 5 presents the the state of the art of the related works. It describes the approaches adopted in other works and made a comparison with the proposal of current work.

Conclusions summarizes the objective and the results of the work.

Appendix A-F lists all the algorithms referred in the dissertation.

Appendix G presents a brief introduction to the *Automatic Query Expansion* (AQE).

Chapter 1

Background Theory

T he term "index" means an organization of data aimed at improving data search and storage, the objective of this work is to propose an index with the following characteristics:

1. Must be used on a large amount of data. The assumption is that it is not possible or convenient to use a single workstation to host all the data. Despite the speed of processing and storage capacity of computers increases with some regularity ("Moore's law"), on the other hand the amount of data produced daily grows at a rate significantly higher. In fact, in addition to the Big Data and Linked Open Data, just think of specific domains such as research on DNA sequencing that makes available an amount of data greater than the processing capacity of the processors [13].

2. In addition to the traditional search (keyword search), must make search by semantic available. The term "semantic" means the ability to use lexical relations (hypernymy, hyponymy[1], meronymy, synonymy [2], holonymy[3], etc.) to improve the quality of search. Referring therefore to

---

[1]In linguistics, an hyponym is specific term used to designate a member of a class. For instance, *oak* is a hyponym of *tree*, and *dog* is a hyponym of *animal*. The opposite of a hyponym is a hypernym.

[2]Synonyms are different words which have the same meaning, or almost the same meaning.

[3]A meronym denotes a constituent part of, or a member of something.That is, "X" is a meronym of "Y" if Xs are parts of Y(s), or "X" is a meronym of "Y" if Xs are members of Y(s). For example, *finger* is a meronym of *hand* because a finger is part of a hand. Holonymy is the opposite of meronymy.

the ability to index not only structured information such as points with numeric coordinates, but also unstructured information as text files.

3. Must be distributed over a computer network and ensure the greatest possible benefits in terms of efficiency (insert, delete), i.e. the performance should be close to the traditional indexes that use a single workstation.

When we talk about indexes, we can think of, for simplicity, data in the form $\langle K_i, P_i \rangle$ where K_i is the value of the key attribute and P_i is the pointer to the data records. Typically, indexes are based on data structures that use trees or some form of hashing. In the first case ($B-tree$, B^+-tree, B^*-tree, $k-dtree$, $R-tree$, etc.) the search is based on comparisons between keys. The goal is to minimize the number of comparisons to find the search key, typically $O(\log N)$ where N is the number of keys. In the second case (*Hash tables*, *SHA1*, *MD5*, etc.) the search points directly to the key using arithmetic operations that transform the keys in addresses.

In addition, a search index should implement at least the insertion and deletion of key/value pairs. It should be noted that:

1. Tree-based search indexes in order to ensure $O(\log N)$ steps in search must be balanced;

2. Generally an hash table needs much more space than a tree-based index and it requires careful management of collisions.

It follows that insertions and deletions require other tasks related to balancing (in the case of the trees) or the management of collisions and space (in the case of the hash tables). Also indexes can be specialized depending on how you want (or need) to consider the data-type (*one-dimensional* or *multi-dimensional*) and the kind of search to implement (*point query*, *range query*, *k-nearest neighbor queries*, etc.).

Traditional tools such as RDBMS, are inadequate for several reasons well outlined in [14] including: the presence of ontologies that do not lend themselves to be inserted into a relational schema, the difficulty of using semantic capabilities in sql-like languages, inadequacy of the approach ACID (Atomicity, Consistency, Isolation, and Durability).

The approach taken in this paper (hereinafter described and motivated with more details) is to propose an extension of a tree based data structures at the base of traditional indexes. In particular, attention has been paid to indexes that can handle multi-dimensional data.

1.1 A comparison between multi-dimensional data structures

There have been a number of indexing data structures suggested to handle high-dimensional data: $R-tree$, $k-dtree$, $X-tree$, $SS-tree$, $M-tree$, $Quadtree$ [15], etc. Almost all are tree structures for partitioning space and the most common in practice are R-trees (and its variants R^+-tree, R^*-tree), $k-dtrees$ and $Quadtrees$.

A Quadtree is a k-ary tree, where $k = 2^d$ and d is the number of dimensions. Quadtrees do not scale well to high dimensions, due to the exponential dependency in the dimension. Range trees provide fast multi-dimensional range queries at the cost of higher space usage. Performance acceptable only in low dimensions. Range trees are mainly applicable where a considerable space overhead is acceptable. Best for prefix queries, but also reliable performance for range queries. Especially good in 2D (and 3D) [15].

$R-tree$ theoretically, not known to be stronger than k-d trees. Except in special cases. $R-trees$ and $R^*-trees$ are known to suffer in high dimensionality settings, which carries over to their decentralized counterparts [16]; e.g., the experiments in [17] showed that for dimensionality close to 20, this method was outperformed by the non-indexed approach of [18]. Also in R-tree is crucial the insertion order of the points to get a more balanced tree [19] [20]. The query performance of the resulting target R-tree is likely to be degraded because of the increased overlapping area among rectangles. For these reasons R-trees come with various optimization strategies, different splits, bulk-loaders, insertion and reinsertion strategies etc.

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

On the basis of the above considerations, we chose k-d tree as the data structure to extend. A k-d tree indexes a set of N points in k-space size and allows search nearest neighbors and range queries. Therefore we investigated the way to distribute a k-d tree over a network of peers and the approach to index the text indexing on order to perform semantic query on the index (i.e. the lexical relationship mentioned above).

1.2   Indexing the text

Regarding the indexing of text, the approach followed is similar to that described in [21] in which semantic analysis, including the disambiguation of the terms, boils down to label each term of a natural language text with a corresponding node in an ontology.

The word "disambiguation" means an activity in which a specification of a polysemous term is chosen based on the use of the term in a sentence. On order to semantically analyze a text a dictionary/ontology that lists the terms of language and relationships of meaning between the is essential.

It is useful, though not essential, to have an ontology describing the use of predicates and complements. This ontology allows to improve the disambiguation of terms because it suggests the most likely meaning of a word in accordance with a specific predicate.

The semantic analysis of text provides an intermediate result that is the identification of the parts of speech (subjects, predicates, objects, etc.) and their association with the corresponding nodes in the ontology. In particular, the text is divided into sentences and in each sentence are detected the subject, the predicate and the complements. The next step is to construct one or more triples RDF (Resource Description Framework) in the form $\langle subject, predicate, complement1 \rangle$, $\langle subject, predicate, complement2 \rangle$, and so on, where *subject*, *predicate*, *complement1* and *complement2* are the nodes of ontology. Of course, in some triples the complement could be empty but for the sake of simplicity and without loss of generality, in our model the complement is assumed always to be a not-empty entity. It is also assumed that the lexical ontology contains all subjects, predicates and complements related to all the triples. In particular, as stated later, the lexical ontology used in this work is *Wordnet*.

Then the main step to index a text is the extraction of a set of RDF triples that represent it; Note that there aren't optimal algorithms that perform this transformation but there are several heuristics that provide good results [22] [23] [24]. The next step is to index the set of triples so that queries can be performed on that index.

The outcome of a query search is a set of triples itself and the final result is the set of fragments of text corresponding to those triples (or the entire document from which have been extracted). It should be noted that by implication it was assumed that the "semantic content" of a text and the set of RDF triples it are equivalent. Of course this is not admissible in an absolute sense but it is acceptable and advantageous from the point of view of processing semantic

queries on natural languages texts.

1.2.1 Similarity measures and metric spaces

In order to insert the RDF triples in a k-d tree in the present work it has been proposed to transform them, i.e. map them, in points of a k dimensions vector space. The first step is to build a metric space (T, d) where T is the set of all RDF triples (whose subjects, predicates and complements belong to ontology) and d is a distance function (or metric) $d : T \times T \rightarrow R$; specifically, the distance between two RDF triples represents their *semantic distance*. The function d can be any function that satisfies the known properties (reflexivity, symmetry and triangle inequality). A number of semantic similarity measures between concepts in an ontology have been proposed in the literature and almost all of them rely on lexical relationships present in the ontology itself. The most known are: *Resnik*, *Leacock & Chodorow*, *Wu & Palmer* [25]. Such semantic similarity measures are defined between pairs of concepts of ontology and the present study has has proposed an extension of them to pairs of RDF triples by defining a family of distances between two triples as a linear combination of distances of their respective subjects, predicates and complements. In particular, let $T = \{T_1, ..., T_n\}$ the set of RDF triples. Each $T_i = (S_i, P_i, C_i)$ has a subject S_i, a predicate P_i and a complement C_i. S_i, P_i and C_i are nodes (i.e. concepts) belonging to an ontology that represent the human language, for instance the *Wordnet* [26]. Leacock & Chodorow is defined as:

$$d\left(w_1, w_2\right) = -\log\left(\frac{length\left(w_1, w_2\right)}{2D}\right) \tag{1.1}$$

Where D id the maximum depth of the taxonomy and $lenght\left(w_1, w_2\right)$ is the shortest path between the nodes representing w_1 and w_2 in the taxonomy. Instead, Wu & Palmer is defines as:

$$d\left(w_1, w_2\right) = \frac{2depth\left(LCS\right)}{depth\left(w_1\right) + depth\left(w_2\right)} \tag{1.2}$$

Where LCS is the first common ancestor of w_1 and w_2 and $depth\left(w\right)$ is the length of the path from the root of the taxonomy to the node w. If we consider only the lexical relation of the type hypernymy/hyponymy in the Wordnet then the Wordnet reduces itself to a taxonomy and both of the previous distances can be calculated. At this point, it is possible to define a metric over two

triples T_1 and T_2 in T:

$$h\left(T_1, T_2\right) = \alpha d_1\left(S_1, S_2\right) + \beta d_2\left(P_1, P_2\right) + \gamma d_3\left(C_1, C_2\right) \qquad (1.3)$$

Where $\alpha + \beta + \gamma = 1$ and $\alpha, \beta, \gamma \in R$; d_i is Leacock & Chodorow or Wu & Palmer distance (or any other semantic similarity measure). Now, (T, h) is a *metric space*. The choice of the coefficients α, β and γ determines the importance to be assigned to that part of the speech, for example you can give more emphasis to the similarities of subjects or predicates according to the specific context. The choice instead of metrics allows you to use the most appropriate metrics in a that context. The simplest case is $d_1 = d_2 = d_3$ and $\alpha = 0.4, \beta = 0.3$ and $\gamma = 0.3$. Under the assumption that the lexical ontology contains all the concepts of RDF triples, it is always possible to measure the semantic distance between two triples. In the case of multiple senses, words are opportunely disambiguated choosing the most fitting sense for the considered domain using a context-aware and taxonomy-based approach [27], if necessary.

The table 1.1 shows the results of a comparison between three open-source Java libraries to calculate semantic similarity measures. These libraries are:

1. Java WordNet Similarity (JWS): http://www.sussex.ac.uk/Users/drh21/

2. SimPack (SP): https://files.ifi.uzh.ch/ddis/oldweb/ddis/research/simpack/index.html

3. Semantic Measures Library & Toolkit (SML): http://www.semantic-measures-library.org/sml/index.php?q=sml-semantic-measures

The criteria considered for comparison are: the support for OWL[4] and Wordnet; the support for the best known similarity measures and the ease of use.

The JSW is library used in the tests.

1.2.2   Mapping metric space in vector space

The remaining task is to map the triples of the metric space (T, h) in points (i.e. vectors) of a vector spaces R^k, i.e. for each triple T must be found a point

---

[4]The Web Ontology Language (OWL) is a family of knowledge representation languages for authoring ontologies.

Criteria	JWS	SP	SML
OWL		X	
Wordnet	X		
Wu & Palmer	X		X
Lin	X	X	X
Resnik	X	X	X
Jiang Conrath	X	X	X
Levenshtein	X	X	
Jensen Shannon		X	
Leacock & Chodorow	X		X
Ease of use	X	X	

Table 1.1: Java Open-Source libraries to calculate similarity measures.

in R^k that represents it. The need of mapping comes from the fact the only points of R^k can inserted in a k-d tree. In particular, we need a function M:

$$M : T \to R^k \tag{1.4}$$

The mapping M associates a triple $T_i \in T$ to a point $P(x_1, ..., x_k) \in R^k$, in other words:

$$P(x_1, ..., x_k) = M(T_i)\, 1 \leq i \leq n \tag{1.5}$$

The mapping M must have the following properties:

1. Must be injective, namely:

$$T_i \neq T_j \Rightarrow M(T_i) \neq M(T_j) \tag{1.6}$$

2. It must preserve distances, namely:

$$d(T_i, T_j) < d(T_i, T_k) \Rightarrow d_1(M(T_i), M(T_j)) < d_1(M(T_i), M(T_k)) \tag{1.7}$$

Where d_1 represent the distance function in R^k. These properties ensures that a query can be indistinctly executed both in metric space that vector space having the same result. In fact, the first one requires that distinct triples are mapped in distinct points and the second one ensure that triples close together are mapped in points close together. Finally the mapping algorithm which maps must have sub-quadratic time complexity otherwise it would be unworkable in practice.

1.2.3 Multidimensional Scaling (MDS)

Multidimensional scaling (MDS) is a well-known mapping algorithm that map each object to a point in a k-dimensional space, to minimize the *stress* function. There are several variations, but the basic method is described in [28]. The input of the MDS algorithm is the distance matrix D where the element $d_i j$ is the distance between the object i and object j. This the major drawback of the algorithm because the calculation of the matrix D requires $O\left(\log N^2\right)$ in time (N is the number of objects) and for this reason is not applicable on large amounts of data.

1.2.4 FastMap

FastMap [29] is a much faster mapping algorithm that does not require the calculation of the entire distance matrix. The time complexity of the FastMap algorithm is $O\left(\log kN\right)$ where k is the number of dimensions of the target vector space and it is much smaller than N. FastMap does not guarantee the injectivity of the returned mapping but usually in practice it does not seems a problem because the number of triples associated to the same point (collisions) is very low. The number of collisions is a measure of the goodness of the mapping algorithms and a some tests on small dataset suggest that MDS algorithm is better than FatMap. FastMap is the algorithm used to obtain the required mapping. The distance measure used in FastMap is L_2, the euclidean distance.

   FastMap executes k iterations and in the *i-th* iteration it performs the following main tasks:

1. Randomly select (and memorize) two triplets $T_a, T_b \in T$, named *pivots*

2. For each $T_j \in T$:

    (a) Calculate the *i-th* coordinate of P_j using:

    $$x_i = \frac{d\left(T_a, T_j\right)^2 + d\left(T_a, T_b\right)^2 - d\left(T_b, T_j\right)^2}{2d\left(T_a, T_b\right)}$$

    (b) Project T_j on a hyper-plane perpendicular to the line (T_a, T_b). Use in the next iteration the distances between the projections on that hyper-plane to calculate x_{i+1}

At the end of the execution of FastMap, N points $P_j \in R^k$ are generated. The time required to map each, or a new, RDF triplet in a point of R^k is $O(k)$. See [29] for a more detailed description of FastMap.

It is important to note that there are circumstances where it is not possible to have a mapping without collisions. For example, because it is not possible to draw up a plan 4 points equidistant from each other it follows that any set of 4 triples equidistant from each other cannot be mapped into R^2 without collisions. On the other hand, can map the same triples in R^3 without collisions. In General, to map a set of m triple equidistant from each other it is necessary that the vector space has at least $m-1$ dimensions. This observation is a only a necessary condition and the fact that theoretically there exists a mapping without collision does not imply that MDS or FastMap can find it.

1.2.5 Semantic Query Processing

Once all RDF triplets $T_j \in T$ are mapped in a point $P_j(x_1, ..., x_k) \in R^k$ as described in previous section, every point P_j is inserted in the k-d tree. Given N points, the average cost of inserting, as well as searching for (i.e., an exact match query), a node is $O(logN)$, then the average time complexity to build the k-d tree is $O(NlogN)$.

The problem of *similarity search* - the process of finding and retrieving triples that are similar to a given triple or set of triplet - reduces to finding the nearest or m nearest neighbors or to performing a range query.

Given an RFD query triplet q, the execution of a *m-nearest neighbors* query is accomplished through the following steps:

1. Map q in $P_q \in R^k$ using the memorized pivots

2. Find the closest m points to P_j in the k-d tree

3. Return the RFD triples associated to the retrieved points.

In a similar way, a *range query* can be performed. Hence, this kind of search can be done efficiently by using the well-known properties of k-d trees.

1.2.6 A motivating example: Requirements Validation

As an applicative example of possible usages of our approach, let us consider the problem of finding inconsistencies in software requirements written in natural language. Recent studies on requirement engineering demonstrate that

software teams still face difficulties in the transition from theory to practice: formalization of requirements, consistency of textual requirements, requirements that describe solutions, definition of specification models [30]. The idea discussed in some works [31] investigates an alternative approach to verify requirements inconsistencies leveraging on the adoption of similarity measures between concepts modeled in RDF. It starts from the recent studies on semantic web techniques and intends to investigate if concepts related to semantic distance between RDF triplets can allow any evaluation of consistency within each single software artifact and between different software artifacts written in natural language. For example, *consistency* requires that no two or more facts contradict each other. The approach in [31] is essentially based on the intuition of modeling each software artifact as sets of RDF triplets and try to find contradiction, conflicts, obstacles, etc. analyzing only the triplet sets. The verification consists in the detection of a set of patterns (rules) in the sets of triplets. For instance, a simple pattern is: Two triplets T_i and T_j are inconsistent if they:

1. Have the same subject s;

2. Have the same object o;

3. Have predicates contradict each other.

This rule defines the simplest case of inconsistency rising from an explicit contradiction and it requires a search over the set of triplets modeling the requirements. Some other rules require the detection of similar (i.e. semantically close) triplets. Since a requirement has more than one sentence and a sentence often results in more than one triplet, even a small set of requirements can generates a considerable number of RDF triplets. Therefore, there is a 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.

1.3   Distributed k-d trees

A distributed k-d tree is a data structure that maintains the links and the nodes of a k-d tree but nodes are distributed over more than one processor. This section describes a naive distribution approach and the nearest neighbor queries algorithms with this distributed k-d trees. In general, a distributed k-d tree has mainly two advantages over a sequential k-d tree (a k-d tree one processor):

1. It can handle more nodes/points than a sequential k-d tree.

2. It allows the elaboration of multiple queries simultaneously.

Many strategies can be used to achieve a distributed version of a k-d tree. In general, a mapping between the set of nodes of the tree and the set of processors is required. In particular, if $N = \{n_1, ..., n_h\}$ is the set of the nodes of the k-d tree and $PR = \{pr_1, ...pr_t\}$ is the set of the available processors then a function $MAP : N \rightarrow PR$ must be defined, where $MAP(n_i) = pr_j$ means that the processor pr_j hosts the node n_i. In order to reuse the well-know searching algorithms all the links of the k-d tree must be preserved in the distributed k-d tree. This fact ensures that the elaboration of search algorithms with a distributed k-d tree will visit the same nodes, in the same order with respect the elaboration with a sequential k-d tree. During the elaboration of these algorithms with a distributed k-d tree, if n_i is current node and n_j is the next visited node, the main problem is that n_i and n_j may be on different processors, that is, $MAP(n_i) = pr_i$ and $MAP(n_j) = pr_j$ with $pr_i \neq pr_j$. To cope with this problem, an update to search algorithms needs. In particular, the processor pr_i will delegate to pr_j the remaining part of search. To do this, pr_i will send a message containing all necessary information to pr_j. If the original search algorithm requires a feedback from the node n_j to the node n_i, after its own execution pr_j will send a message to pr_i with the result. The amount of information needed to implement this message passing mechanism is $O(1)$ in space, in fact, each node n of a k-d tree has at most three neighbors (the parent, the left child and the right child) and for each of them n must store their processor. section A.1 and section A.1 contain the pseudocode of the *k-nearest neighbor* (KNN) and the *distributed k-nearest neighbor* (DKNN) as described.

The time complexity KNN and DKNN algorithms is $O(\log N)$, where N is the number of points stored in the tree, both of them visit the same nodes in the same order. In addition, the DKNN algorithm must exchange a number of messages (hop) between processors. The number of hops is less than the number of nodes visited during the execution of the KNN algorithm because a message can be sent only when the KNN algorithm moves from one node to another one. Therefore, the number of hops is $O(\log N)$ and the DKNN is an efficient search algorithm. Formally the objectives (1) and (2) were achieved.

Example The following example illustrates the elaboration of a query with distributed k-d tree and shows the limits of this approach. Suppose T is a dis-

tributed k-d tree and q_1, q_2, q_3 and q_4 are queries. If $T.root$ is the node root of T then the processor $pr_1 = MAP(T.root)$ at the beginning has four message in its queue (one for each query). The processors pr_1 starts the elaboration of the message for q_1 and cannot elaborate the message for q_2 until the query q_1 ends or the execution of q_1 requires a node belonging to another processor pr_2. In the second case, pr_1 sends a message containing q_1 and the current result (if the current node does not have matching points it is empty) to pr_2. From now on, pr_2 carries out q_1 and pr_1 starts the execution of q_2. When pr_2 ends its elaboration it sends a message, e.g. $q1ResultMessage$, to pr_1 containing the final result of q_1. The message $q1ResultMessage$ enters the message queue of pr_1 and without any priority associated to $q1ResultMessage$ there is no guarantee that it will be processed before the execution of the remaining query q_3 and q_4.

From the previous example it is possible to make some general remarks: the processor pr_1 is the bottleneck of the entire system and without an accurate message priority management the throughput of a distributed k-d tree can be worse than a traditional k-d tree. This behavior does not depend on the distribution strategy (i.e. the mapping function MAP cannot solve this problem). This issues represent a substantial limit for distributed k-d tree.

Hence the need for a new distributed search algorithm that can:

1. Start a query from any randomly chosen node/processor.

2. Terminate the execution of a query as soon as the result was obtained without visiting the root node, if not necessary.

Chapter 2

A First Step Toward Distributed k-d trees

In order to derive a new distributed searching algorithm with distributed k-d trees and to explain how it works, an example of the elaboration of the DKNN with binary tree follows. Consider the binary tree in Figure 2.1 with bucket size equal to 2. For the sake of simplicity, the internal nodes are labeled with the split value (of the unique coordinate). The buckets are represented with braces.
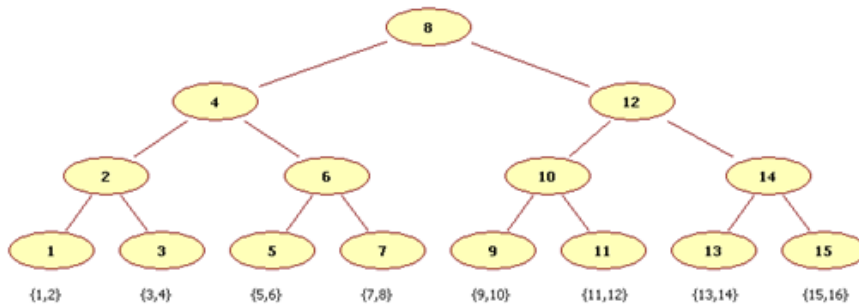

Figure 2.1: A binary tree with 16 points and bucket size of 2.

Let $q = (12, 3)$ be a query requiring the three points closer to 12. The query result is $\{11, 12, 13\}$ and the sequence of the steps and the nodes visited with their states by the KNN algorithm are listed in Tables 2.1 and 2.2.

Table 2.1: First execution: steps from 1 to 8

Step	1	2	3	4	5	6	7	8
Status	None	rightVisited	None	rightVisited	none	leftVisited	None	rightVisited
Node	8	8	12	12	14	14	13	14
Result							(13, 14)	(13, 14)

Table 2.2: First execution: steps from 9 to 15

Step	9	10	11	12	13	14	15
Status	rightVisited	None	rightVisited	none	rightVisited	rightvisited	rightvisted
Node	12	10	10	11	10	12	8
Result	(13, 14)	(13, 14)	(13, 14)	(11, 12, 13)	(11, 12, 13)	(11, 12, 13)	(11, 12, 13)
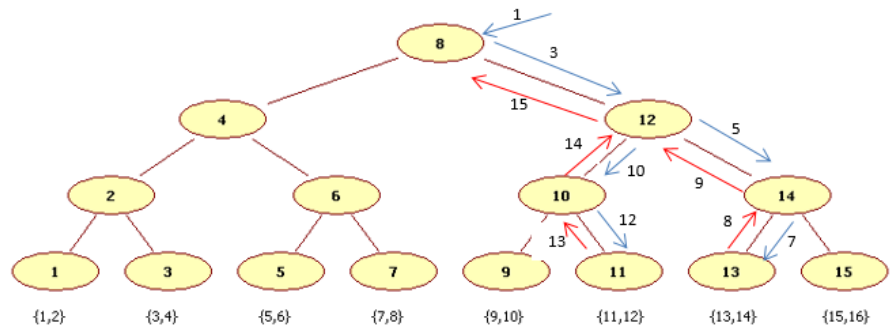
Figure 2.2: Steps of the execution in which the execution moves from a node to another.

Figure 2.2 shows the steps in which the elaboration moves from a node to another one. Steps in which only a change of status occurs are not represented (they should be outcoming and incoming edges to the same node). Please, note that DKNN algorithm elaborates the same steps of KNN algorithm and then it visit the same nodes of the k-d tree in the same order therefore the observations on the elaboration of KNN holds for DKNN also. Suppose a second elaboration of the same query q starts at node 10 instead of the root node of the k-d tree. In that case, a number of steps of the KNN can still be carried out, in particular the steps: 11, 12 and 13. The contents of the $Result$ in this case is different from the previous run. The row $Step2$ of Table 2.3

Table 2.3: Second execution: steps from 1 to 5.

Step2	1	2	3	4	5
Status	None	rightVisited	none	rightVisited	*leftVisited*
Node	10	10	11	10	12
Result			(11, 12)	(11, 12)	(11, 12)

enumerate the steps performed in this elaboration, the situation is represented in Table 2.3.

The Figure 2.3 shows the steps performed in the second elaboration.
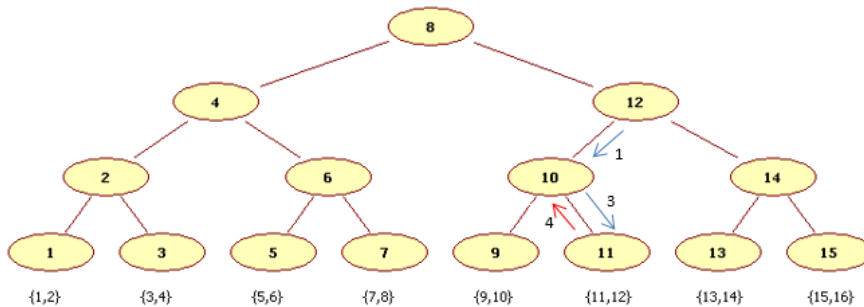
Figure 2.3: Steps from 1 to 3 of the second elaboration.

At this point the KNN algorithm moves from node 10 to node 12 which, until now, has never been visited and then it has $status = NIL$. Node 12 in the first execution had the $status =' rightVisited'$ as in Table 2.2, step 14. Because the node 10 is the left child of the node 12 then the status of the node 12 might be set to $leftVisited$ (this is a change to the KNN algorithm). Now, the KNN algorithm checks if the right child of the node 12 node must be visited and then it performs the steps from 5 to 9 of the first execution. At this point the situation is represented in Tables 2.4 and 2.5.

Table 2.4: Second execution: steps from 1 to 6.

Step2	1	2	3	4	5	6 (5 in table 2.1)
Status	None	rightVisited	none	rightVisited	leftVisited	None
Node	10	10	11	10	12	14
Result			(11, 12)	(11, 12)	(11, 12)	(11, 12)

Table 2.5: Second execution: steps from 7 to 11.

Step2	7 (6 in table 2.1)	8 (7 in table 2.1)	9 (8 in table 2.2)	10 (9 in table 2.2)	11 (15 in table 2.2)
Status	leftVisited	None	rightVisited	rightVisited	*rightvisted*
Nodo	14	13	14	12	8
Result	(11, 12)	(11, 12,13)	(11, 12,13)	(11, 12,13)	(11, 12, 13)
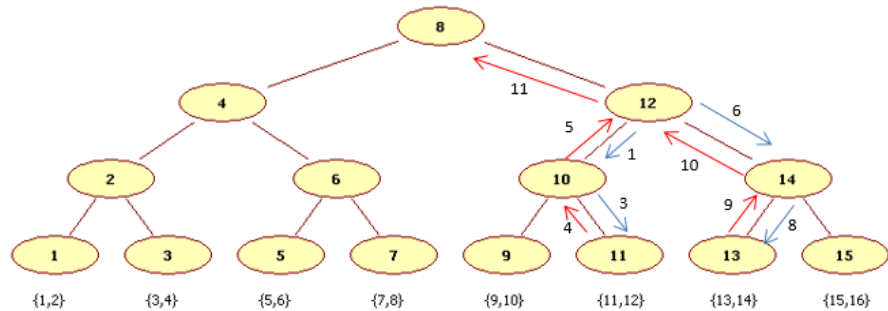
Figure 2.4: Steps performed in the second elaboration.

Since both subtrees of the node 12 were visited, the KNN algorithm moves elaboration to node 8 which again haa $status = NIL$ (it has never been visited also). Node 8 in the first elaboration had the $status =' rightVisited'$ as Table 2.2, step 15. Again, the status of the node 8 might be set to $'rightVisited'$ (this is a change to the KNN algorithm) because node 12 is the right child of the node 8. The elaboration can continue as in KNN algorithm and it performs the step 11 (which corresponds to step 15 of Table 2.2). Finally, the elaboration ends with the same result as the KNN algorithm. Reviewing the previous example the following general observations about the two elaborations can be stated:

1. They visit the same nodes: this fact ensures that the same leaves are visited and the same results are retrieved;

2. They do not visit the nodes in the same order: in fact, the second elaboration visit the subtrees of the node 12 node in reverse order;

3. They do not visit nodes the same number of times: in the first elaboration nodes 8 and 12 are visited respectively 3 times 2 instead in the second elaboration they are visited respectively 1 and 2 times;

4. The second elaboration gets the same result the first elaboration if:

   (a) It starts from one of the nodes visited of the KNN algorithm;

   (b) The status of the nodes that the KNN algorithm would visit before the node from which the second elaboration starts are properly set. The status of these nodes is NIL because they have not yet been visited.

2.1 Starting a search from a node visited by KNN algorithm

This section introduces a new algorithm for k-nearest neighbor query that starts the search from a visited by KNN algorithm. Assume that:

- T is a binary tree and $q = (p, k)$ is a k-nearest neighbor query where p is the query point and k is the number of neighbors;

- n is a node visited by KNN algorithm during its elaboration.

In particular, the node n is defined the *starting node* of the query. The result of the query q is:

$$result = randomNearestQuery(p, k, n) \qquad (2.1)$$

Where $result$ is an instance of data structure $Result$ described in section A.1. The algorithm $randomNearestQuery$ is the following:

Algorithm 1 randomNearestQuery(queryPoint, k, n)

Require: $queryPoint$ and k are integer values, n is a node of T

Ensure: return the k-nearest neighbor points of $queryPoint$ in T

1. $result \leftarrow$ createResultInstance(queryPoint, k) {create a new instance $result$ of $Result$ for $queryPoint$ with size k}
2. randomNearestNeighbor(n, queryPoint, k, result, 'none')
3. **return** $result$ {$result$ contains the results of the query}

Algorithm 2 randomNearestNeighbor(v, p, k, result, status)

Require: n is a node of T, p and k are integer values, $result$ is an instance of $Result$ and $status$ is a string

1. **if** $status \neq NIL$ **then**
2. $v.status \leftarrow status$
3. **end if**
4. randomNN(v, p, k, result)

Algorithm 3 randomNN(v, p, k, result)

Require: v is a node of T p and k are integer values and $result$ is an instance
of $Result$
 1. **if** $v.isLeaf$ **then**
 2. $result.add(v.getBucket)$
 3.   **if** mustBeSetParentStatus(v) **then**
 4.   randomNearestNeighbor(v.parent, p, k, result, NIL)
 5.   **end if**
 6. **else**
 7.   **if** $v.status =' none'$ **and not** $v.isLeaf$ **then**
 8.   **if** $p < v.SplitValue$ **then**
 9. $v.status \leftarrow' leftVisited'$
10.   randomNearestNeighbor(v.left, p, k, result, 'none')
11.   **else**
12. $v.status \leftarrow' rightVisited'$
13.   randomNearestNeighbor(v.right, p, k, result, 'none')
14.   **end if**
15.   **end if**
16.   **if** $v.status =' rightVisited'$ **then**
17.   **if** $v.left \neq NIL$ **and** $mustBeVisited(v, p, result)$ **then**
18.   randomNearestNeighbor (v.left, p, k, result, 'none')
19.   **end if**
20.   **if** $mustBeSetParentStatus(v)$ **then**
21.   randomNearestNeighbor(v.parent, p, k, result, NIL)
22.   **end if**
23.   **end if**
24.   **if** $v.status =' leftVisited'$ **then**
25.   **if** $v.right \neq NIL$ **and** $mustBeVisited(v, p, result)$ **then**
26.   randomNearestNeighbor (v.right(), p, k, result, 'none')
27.   **end if**
28.   **if** mustBeSetParentStatus(v) **then**
29.   randomNearestNeighbor(v.parent, p, k, result, NIL)
30.   **end if**
31.   **end if**
32. **end if**

The $mustBeVisited$ procedure is described in section A.1, algorithm 15.
The differences between the algorithms 3 (randomNN) and 14 (NN) are the

calls to the $mustBeSetParentStatus$ procedure. In particular, at the end of each recursive call, the algorithm 3 checks the status of the parent node of the current node, e.g. w, and if w has never been visited it sets its status calling $mustBeSetParentStatus$. A this point, the node w is elaborated by a call to $randomNearestNeighbor$. Without this change the the algorithm 3 would stop its elaboration because there is not a pending recursive call to $randomNearestNeighbor$ and it would return an incorrect result. The algorithm $mustBeSetParentStatus$ is the following:

Algorithm 4 mustBeSetParentStatus(Node v)

Require: v is a node of T

Ensure: Checks if the $status$ parent of the current node must be set and it sets the correct value.

1. **if** $v.parent \neq NIL$ **then**
2.   **if** $v.parent.status = NIL$ **then**
3.   **if** $v.splitValue < v.getParent.splitValue$ **then**
4. $v.parent.status \leftarrow' leftVisited'$ {v is a left child of $v.parent$}
5.   **return true**
6.   **else**
7. $v.parent.status \leftarrow' rightVisited'$ {v is a right child of $v.parent$}
8.   **return true**
9.   **end if**
10.   **else**
11.   **return false**{$n.parent.status \neq NIL$ then do nothing}
12.   **end if**
13. **else**
14.   **return false**{n is the root of the tree then do nothing}
15. **end if**

2.2 Finding a starting node

The algorithm 22 (randomNearestQuery) works well only if it starts from a node that is visited by the algorithm 12 (nearestQuery) in in section A.1. The following property helps to characterize this kind of nodes:

Theorem 2.1 *Let T be a binary tree and $M = \{m_1, ..., m_j\}$ the nodes visited at least once by the algorithm 12 in section A.1 during the of elaboration of*

the query $q = (p, k)$. If for a node m holds (Starting Node Property - SNP):

$$\begin{cases} m.minValue \leq p \leq m.maxValue & \text{if } m \text{ is a leaf} \\ m.left.splitValue \leq p \leq m.right.splitValue & \text{otherwise} \end{cases} \quad (2.2)$$

Then $m \in M$ ($m.minValue$ and $m.maxValue$ are respectively the minimum and maximum values contained in the bucket of the leaf m).

Proof: It is a proof by contradiction. Let x be an internal node of the tree for which the (4.1) holds but such that $x \notin M$. Because (4.1) holds then in the bucket of the leaves of the subtree rooted in x there might be at least a point t that may be returned in the result of the query q. The value k determines whether the point t will be part of the result. If the algorithm does not visit the node x would not have the opportunity to evaluate whether to add t to the query result and then the search result may be incorrect. This is a contradiction of the correctness of standard search algorithm then x ? M. The proof in the case x is a leaf is the same. The SNP in (4.1) can be used to build a recursive algorithm that starting from a random node n of T and a query point p reaches a node $m \in M$.

Algorithm 5 findStartingNode(p, n)

Require: p is an integer value and n is a node of T
Ensure: return a starting node $m \in M$ for query point p

1. **if** $n.isRoot$ **then**
2.   **return** n
3. **else**
4.   **if** $n.isLeaf$ **then**
5.   **if** $n.minValue \leq p \leq n.maxValue$ **then**
6.   **return** n
7.   **else**
8.   **return** $findStartingNode(p, n.getParent)$
9.   **end if**
10.   **else**
11.   **if** $n.left.splitValue \leq p \leq n.right.splitValue$ **then**
12.   **return** n
13.   **else**
14.   **return** $findStartingNode(p, n.getParent)$
15.   **end if**
16.   **end if**
17. **end if**

In particular, the algorithm $findStartingNode\,(p,n)$ returns the node n if the SNP holds for it otherwise it moves to the father of n. It moves from bottom to top in the tree therefore its time complexity is $O\,(\log N)$. Please, note that for given the query $q = (p,k)$:

1. The starting node depends only on the query point p and it do not depend on the value of k.

2. The (4.1) is a sufficient but not necessary condition. In fact, in the example in Figure 2.2 the node 10 is a node visited by the KNN algorithm during the of elaboration of the query but the (4.1) do not apply.

2.3   Analysis of the $findStartingNode$ algorithm

The $findStartingNode$ algorithm moves to the parent of the current node if the SNP does not hold and, of course, there is no guarantee that it has not reached the root of the tree. Let T be a tree and p the query point, in order to estimate how many times in average the $findStartingNode$ returns the root of T the following test can be executed:

Algorithm 6 testFindStartingNode(T, p, percRoot, percNoRoot)

Require: T is binary tree, p is an integer value, $percRoot$ and $percNoRoot$ are double values

Ensure: Calculate how many times in percentage the $findStartingNode$ returns the root node with the tree T and query point p. It returns $percRoot$ and $percNoRoot$

1. $nrRoot \leftarrow 0$
2. $nrNoRoot \leftarrow 0$
3. **for all** $n \in T.allNodes$ **do**
4.   **if** $n \neq T.root$ **then**
5. $resultNode \leftarrow findStartingNode\,(p, n)$
6.   **if** $resultNode = T.root$ **then**
7. $nrRoot \leftarrow nrRoot + 1$
8.   **else**
9. $nrNoRoot \leftarrow nrNoRoot + 1$
10.   **end if**
11.   **end if**
12. **end for**
13. $percRoot \leftarrow 100 * nrRoot/(T.allNodes.size - 1)$ {do not count the root}
14. $percNoRoot \leftarrow 100 * nrNoRoot/(T.allNodes.size - 1)$

In other words, the algorithm elaborates all nodes in the tree but the root. At the end, $percRoot$ is the percentage in average of how many times the algorithm returns the root node. Of course, $percNoRoot = 100percRoot$. Now, suppose that $treeSet = \{T_{256}, T_{512}, T_{1.024}, T_{2.048}, T_{4.096}, T_{8.192}, T_{16.384}, T_{32.768}\}$ is a set of binary trees where the tree T_i contains points from 0 to $i - 1$. A test that calculates the same percentage of previous test over all the trees in $treeSet$ varying the query point p is the following:

Algorithm 7 $testAverageFindStartingNode$ ()

Ensure: Calculate how many times in percentage the $findStartingNode$ returns the root node with the trees in $treeSet$

1. $avgPercRoot \leftarrow 0$
2. $avgPercNoRoot \leftarrow 0$
3. **for all** $T \in treeSet$ **do**
4. $queryPoint \leftarrow 0$
5.   **for** $queryPoint < T.numPoints$ **do**
6.   testFindStartingNode(T, queryPoint, percRoot, percNoRoot)
7. $sumPercRoot \leftarrow sumPercRoot + numRoot$
8. $sumPercNoRoot \leftarrow sumPercNoRoot + numNoRoot$
9. $queryPoint \leftarrow queryPoint + 1$
10.   **end for**
11. $avgPercRoot \leftarrow sumPercRoot/T.numPoints$
12. $avgPercNoRoot \leftarrow sumPercNoRoot/T.numPoints$
13. **end for**

At the end, the $avgPercRoot$ variable will contains percentage of how many times on average the root node on each tree in $treeSet$ is returned by the algorithm $findStartingNode$ regardless of the query point. Of course, $avgPercNoRoot = 100avgPercRoot$. The results of tests with bucket size of 5, 10, 20, 30 and 40 points are shown in the Table 2.6.

Table 2.6: Test results of $testAverageFindStartingNode$.

bucket dim.	%	Number of points in the tree							average
		512	1024	2048	4096	8192	16384	32768	
5	root	65	65	65	65	65	65	65	*65*
	no root	35	35	35	35	35	35	35	*35*
10	root	65	65	65	65	65	65	65	*65*
	no root	35	35	35	35	35	35	35	*35*
20	root	65	65	65	65	65	65	65	*65*
	no root	35	35	35	35	35	35	35	*35*
30	root	68	68	68	67	67	67	67	*67.4*
	no root	35	32	32	32	33	33	33	*32.4*
40	root	65	65	65	65	65	65	65	*65*
	no root	35	35	35	35	35	35	35	*35*
average	root	*65.6*	*65.6*	*65.6*	*65.4*	*65.4*	*65.4*	*65.4*	**65.5**
	no root	*34.4*	*34.4*	*34.4*	*34.6*	*34.6*	*34.6*	*34.6*	**34.5**

Table 2.6 shows that if the bucket size is much smaller than the number of

points the results do not depend on either the bucket size nor the number of points and in about 65.5% of runs it returns the root node. When the bucket size approaches the number of points, the results should not be considered significant because the trees that result would have very few nodes, e.g. for 64 points and a bucket dimension of 40 points it results a tree with only 3 nodes.

2.4   Improving the $findStartingNode$ algorithm

Intuitively, the algorithm $findStartingNode$ can be improved choosing a random node in the left subtree of the root r of T if:

$$p \leq r.splitValue \tag{2.3}$$

The $findStartingNodeSide$ algorithm is the following:

Algorithm 8 $findStartingNodeSide\,(queryPoint)$

Require: $queryPoint$ is an integer value
Ensure: return a starting node for query point p
 1. **if** $queryPoint < root.splitValue$ **then**
 2.   {let $randomNode$ be a randomly chosen node in left subtree of the root of T}
 3. **else**
 4.   {let $randomNode$ be a randomly chosen node in right subtree of the root of T}
 5. **end if**
 6. $startNode \leftarrow findStartingNode\,(queryPoint, randomNode)$
 7. **return** $startNode$

Each node of the tree can easily be labeled with $left$ if it belongs to the left subtree of T and $right$ otherwise. During construction of the tree, every new node inherits the label of the father. The Table 2.7 show the results of the same tests of Table 2.6 carried out on the $findStartingNodeSide$.

Table 2.7: Test results of $testAverageFindStartingNode$ with $findStartingNodeSide$ algorithm.

bucket dim.	%	Number of points in the tree							average
		512	1024	2048	4096	8192	16384	32768	
5	root	35	35	35	35	35	35	35	*35*
	no root	65	65	65	65	65	65	65	*65*
10	root	35	35	34	34	34	34	34	*34.3*
	no root	65	65	66	66	66	66	66	*65.7*
20	root	35	35	35	34	34	34	34	*34.4*
	no root	65	65	65	66	66	66	66	*65.6*
30	root	36	36	36	34	34	34	34	*34.9*
	no root	64	64	64	66	66	66	66	*65.1*
40	root	36	35	35	35	34	34	34	*34.7*
	no root	64	65	65	65	66	66	66	*65.3*
average	root	*35.4*	*35.2*	*35*	*34.4*	*34.2*	*34.2*	*34.2*	**34.7**
	no root	*64.6*	*64.8*	*65*	*65.6*	*65.8*	*65.8*	*65.8*	**65.3**

Results in Table 2.7 shows that in about 65% of runs the $findStartingNodeSide$ algorithm does not returns the root node of T. Therefore, about 65% of the queries does not start in the root of the tree.

2.5   Finding starting node conclusions

Table 2.8: Test results of $testAverageFindStartingNode$ with $findStartingNodeSide$ algorithm.

Algorithm	returns root	does not return root	Notes
findStartingNode	65,5%	34,5%	Does not require any changes to the structure of the node.
findStartingNodeSide	34,7%	65,3%	It requires the lists of left and right nodes.

The data shown in Table 2.7, although based on binary trees with maximum 16,384 points, show that on average the probability to start a k-nearest neighbor query on a binary tree in node other than the root is 65%.

2.6   Finding ending nodes

Suppose that the $q = (p, k)$ has two boolean attributes named: $upFromLeft$ and $upFromRight$ both of them set to $false$ at start. Furthermore, suppose

that the KNN algorithm is updated with the following: Let v be the current node:

1. If KNN algorithm visited the left subtree of v and the right subtree of v must not be visited (i.e. $mustBeVisited$ returns $false$) and $upFromLeft = TRUE$ then mark v as an $endingnode$ otherwise set $upFromRight = TRUE$.

2. If KNN algorithm visited the right subtree of v and the left subtree of v must not be visited (i.e. $mustBeVisited$ returns $false$) and $upFromRight = TRUE$ then mark v as an $endingnode$ otherwise set $upFromLeft = TRUE$.

Theorem 2.2 *If the KNN algorithm stops its elaboration in an* ending node *it returns the correct results to query q.*

Proof: Suppose the case 1) holds, (the proof of the case 2) is the same) then $upFromRight = upFromLeft = TRUE$. Of course, all points p such that $p < v.splitValue$ cannot belong to the result of the query, Figure 2.5.
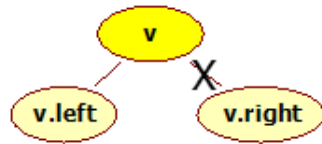

Figure 2.5: The points in the right subtree of v cannot be in the result of the query.

In addition, there must be a node w for which the algorithm previously set $upFromLeft = TRUE$. Only one of the following conditions holds:

1. w is in the left subtree of v, Figure 2.7.

2. w is in the right subtree of $v.parent$, Figure 2.8.
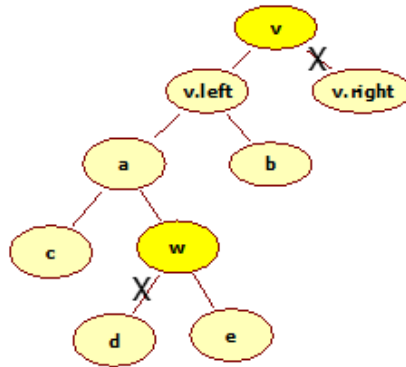
Figure 2.6: w is in the left subtree of v.
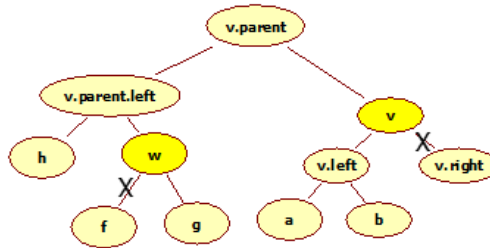
Figure 2.7: w is in the right subtree of $v.parent$.

In both of the previous conditions, all points p such that $p < w.splitValue$ cannot belong to the result of the query. Now, it must be proved that in remaining part of its elaboration, the algorithm does not visit any subtree in the path from v to the $root$. Let $Path = \{n_0, n_1, ..., n_x\}$ be that path, where $n_0 = root$ (i.e. the gray nodes in Figure 2.8 and Figure 2.9). One of the following alternatives holds:

1. If w belongs to the left subtree of v then $n_x = v.parent$, Figure 2.8.

2. If w belongs to the left subtree of v then $n_x = v.parent.parent$, Figure 2.9.

Figure 2.8: Case a)

Figure 2.9: Case b)

In both cases a) and b) it must be proved that $\forall n_i, n_{i+1} \in Path$:

1. If n_{i+1} is a right child of n_i then the left subtree of n_i will not be visited.

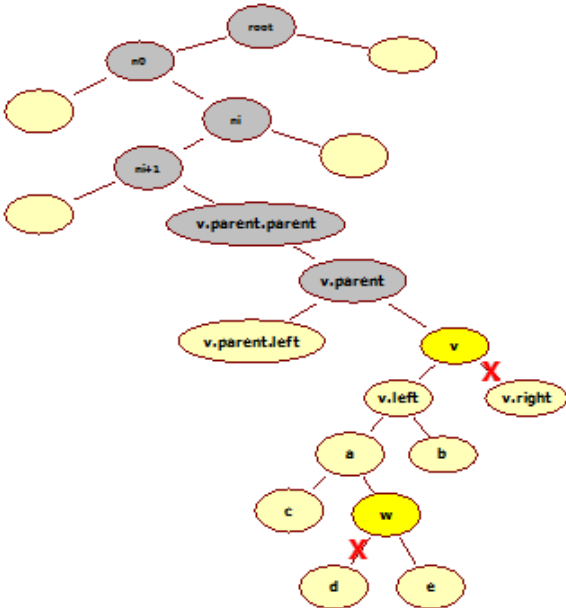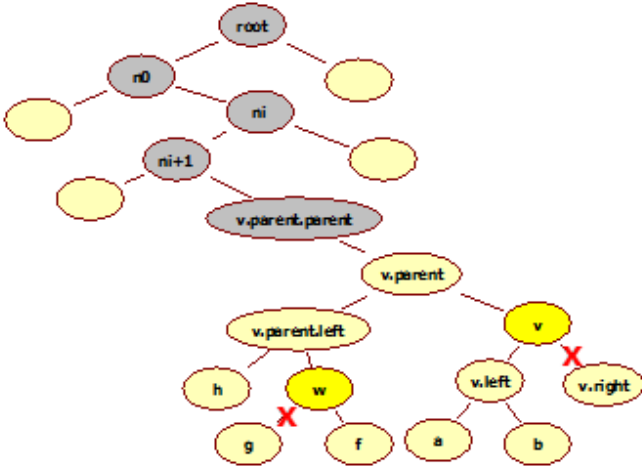2. If n_{i+1} is a left child of n_i then the right subtree of n_i will not be visited.

In Figure 2.8 and Figure 2.9 the node v is a right child of $v.parent$, the proof is the same if v is a left child.

Case a) If n_{i+1} is a right child of n_i and the algorithm should visit the left subtree of n_i then the point p such that $p < n_i.splitValue$ could be in the result of the query. This is a contradiction because $n_i.splitValue < w.splitValue$ and the points $p < w.splitValue$ cannot be in the result of the query. If n_{i+1} is a left child of n_i and the algorithm should visit the right subtree of n_i then the point p such that $p > n_i.splitValue$ could be in the result of the query. This is a contradiction because $n_i.splitValue > v.splitValue$ and the points $p > v.splitValue$ cannot be in the result of the query.

Case b) Same of the previous one. In conclusion, if the algorithm during the elaboration of the nodes along the path from v to the $root$ does not visit any other subtree therefore it cannot add any new point to the result of the query then the search can be stopped in v. section A.1 lists the pseudocode of the KNN using ending nodes. Table 2.6 shows test results of KNN algorithms using ending nodes. In particular, it was used the traditional algorithm of k-nearest neighbor queries varying the values of k between 3 and 10 for each value of p.

Table 2.9: Test results of KNN using ending nodes.

bucket dim.	%	Number of points in the tree							average
		512	1024	2048	4096	8192	16384	32768	
5	end root	7	4	3	2	1	1	0	2.6
	end no root	93	96	97	98	99	99	100	97.4
10	end root	0	1	1	1	1	1	0	0.7
	end no root	100	99	99	99	99	99	100	99.3
20	end root	1	1	1	1	1	1	1	1
	end no root	99	99	99	99	99	99	99	99
30	end root	1	1	1	1	1	1	1	1
	end no root	99	99	99	99	99	99	99	99
40	end root	1	1	1	1	1	1	1	1
	end no root	99	99	99	99	99	99	99	99
average	end root	2	1.6	1.4	1.2	1	1	0.6	**1.3**
	end no root	98	98.4	98.6	98.2	99	99	99.4	**98.7**

Table 2.9 shows that in about 98% of cases the ending node is different from the root node of the tree.

2.7 Random k-nearest neighbor query on binary trees

section A.1 shows the pseudocode for the random k-nearest neighbor queries on binary trees. The algorithm described has the following features:

1. It can start the query with a randomly chosen binary tree node;

2. In 65% of cases the beginning of the query does not involve the root of the tree;

3. In 98% of cases the end of the query does not involve in the root of the tree;

4. It has the same time complexity of the algorithm KNN.

Chapter 3

Allocation strategy

T he allocation of the nodes of the tree over the processors is based on
the *full condition* of the processor. The full condition is a criteria in-
dicating that the processor cannot contains other nodes of the tree because it
consumed its resources. Resources can be, for example RAM, disk space or
a combination of them. If there are identical processor then the full condition
is the same for all of them but if not, each processor may have its own full
condition. When a processor reach its full condition it means that a predeter-
mined constant number of nodes can be added to it, e.g. 2, 3, 5 nodes, and
the processor will redirect to others processors the insertion of new nodes in
the tree. Assume the scenario of bulk load points in the tree, this ensure that
the resulting tree is a balanced tree. Suppose $N = \{p_1, ... p_n\}$ is the set of the
points, the proposed distribution strategy works as follow:

1. **while** $N \neq \varnothing$ **do**
2.   {Choose next point p in N to be inserted (e.g. the median with respect
      the split coordinate)}
3.   SendMessage(pr_0, INSERT(p))
4. **end while**

Follows the pseudocode for each processor.

33

Algorithm 9 $onReceiveMessage(MessageM)$

Require: M is message
Ensure: Elaborate every new message
  1. **if** M is an INSERT message **then**
  2.   insert(M.p, M.senderProcessor)
  3. **else**
  4.   **if** if M is a MOVE message **then**
  5.   moveRightSubtree(M.subtree)
  6.   **end if**
  7. **end if**

Algorithm 10 $insert(p, sender)$

Require: p is a point and $sender$ is the processor that sent the message
  1. **if** full condition is TRUE **then**
  2.   nextProcessor = requireNewProcessor
  3.   **if** $next \neq NIL$ **then**
  4.   insertion (p)
  5. $subtree \leftarrow$ getNextRightSubtree (root)
  6.   SendMessage($nextProcessor$, MOVE($subtree$))
  7.   create a proxy node $proxy$ for the root of $subtree$ pointing to $nextProcessor$
  8.   replace $subtree$ with $proxy$
  9.   **else**
 10.   SendMessage($sender$, REJECT(p))
 11.   **end if**
 12. **else**
 13.   insertion (p)
 14. **end if**

Algorithm 11 $getNextRightSubtree(n)$

Require: n is a node
  1. **if** n.right is not a proxy node **then**
  2.   **return** n.right
  3. **else**
  4.   **return** getNextRightSubtree (n.left)
  5. **end if**

The insertion procedure is the well-known insert algorithm with binary trees or k-d trees.

The following example shows how the distribution strategy works. Suppose that:

1. A processor reach its full condition when it contains 7 nodes;

2. The bucket dimension is 1, therefore the insertion of each new point will create a new node;

3. Nodes are labeled with integers following the creation order.

After the insertion of the first seven points the processor pr_0 reached its full condition, Figure 3.1.
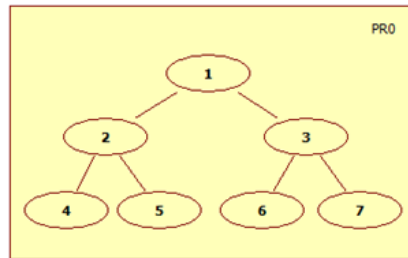
Figure 3.1:

The insertion of the next point will require a new processor, pr_0 will move its right subtree on it and will create the proxy node for node 3, Figure 3.2.
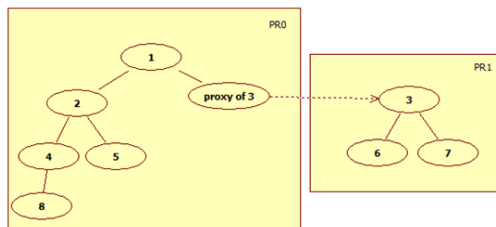
Figure 3.2:

Next point, point 9, causes pr_0 full condition become again true then point 10 (inserted under node 5) will require again a new processor and pr_0 will move another subtree. The procedure $getNextRightSubtree$ will return the subtree rooted at node 5 therefore pr_0 will move it and will create the proxy node for node 5, Figure 3.3.
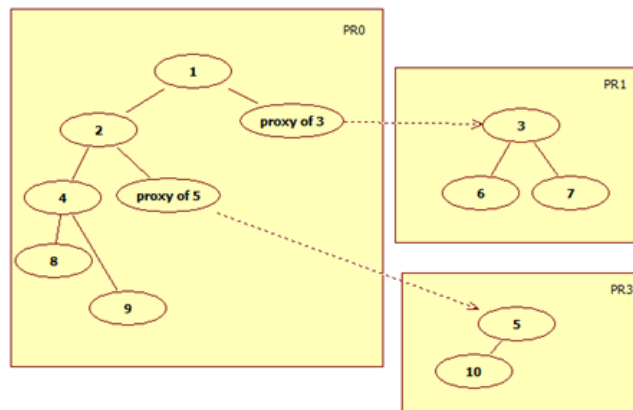
Figure 3.3:

At this point, because the resulting tree should be balanced (bulk load), suppose for the sake of simplicity that the following four points (11, 12, 13 and 14) will be added on processor pr_1, Figure 3.4.
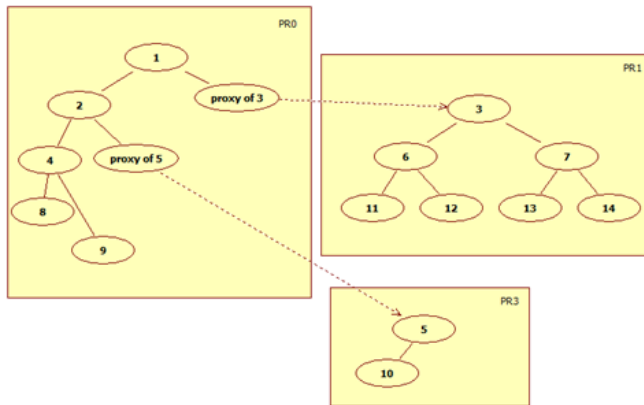
Figure 3.4:

If next point, the point 15, will be inserted on pr_1 under node 11 the situation is depicted in Figure 3.5.
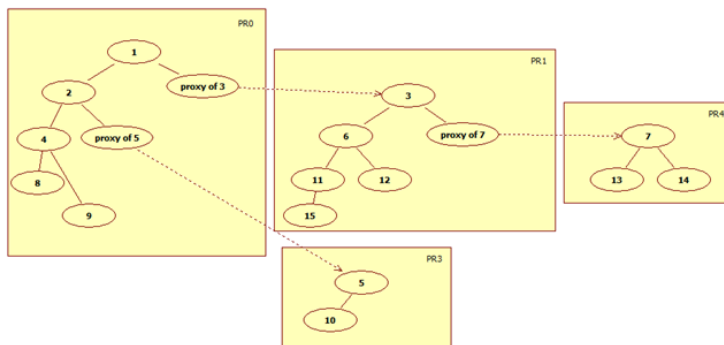
Figure 3.5:

Please, note that:

1. Each processor never contains more than seven nodes, i.e. the full condition

2. Every subtree moved contains approximatively about half of the nodes defined in the full condition. This is true because about half of the tree hosted in the processor will be moved when the full condition became true

3. Every subtree moved is a balanced subtree then every processor will contain a balanced subtree. If a search algorithm does not move on another processor and it will follows links on the left subtree it will traverse a balanced tree.

Chapter 4

Query processing: a novel approach

In order to introduce the random k-nearest neighbor algorithm with k-d trees the FindSartingNode and FindEndingNode algorithms with binary trees must be extended to k-d trees.

4.1 Starting nodes with k-d trees

The Starting Node Property with k-d trees is the following:

Theorem 4.1 *Let T be a d-dimensional k-d tree, $p = (x_1, \mathring{A}\neg x_d)$ the query point and $M = \{m_1, ..., m_j\}$ the nodes visited at least once by the algorithm 12 in section A.1 during the of elaboration of the query $q = (p, k)$. If for a node m holds (k-d tree Starting Node Property - KSNP):*

If x_j is the split coordinate of $m.left$ and $m.right$ and if:

$$
\begin{cases}
m.minValue \leq x_j \leq m.maxValue & \text{if } m \text{ is a leaf} \\
m.left.splitValue \leq x_j \leq m.right.splitValue & \text{otherwise}
\end{cases}
\tag{4.1}
$$

Then $m \in M$ ($m.minValue$ and $m.maxValue$ are respectively the minimum and maximum values contained in the bucket of the leaf m).

Proof: If the KSNP applies to m then the subtree of m can contains points matching query q and if the KNN algorithm do not visit this subtree the result may be incomplete.

Example   Let T be the k-d tree in the Figure 4.1. Each node is labeled with the split coordinate and the split value. For instance, the label $y = 110$ means that y is the split coordinate and 11 is the split value. Let $p = (47, 114)$ be the query point and $k = 4$ the number of nearest neighbor points required. The KNN algorithm in 17 steps retrieves the result of the query $q = (p, k)$. The result contains the points $\{(49, 109), (51, 114), (52, 115), (53, 113)\}$.
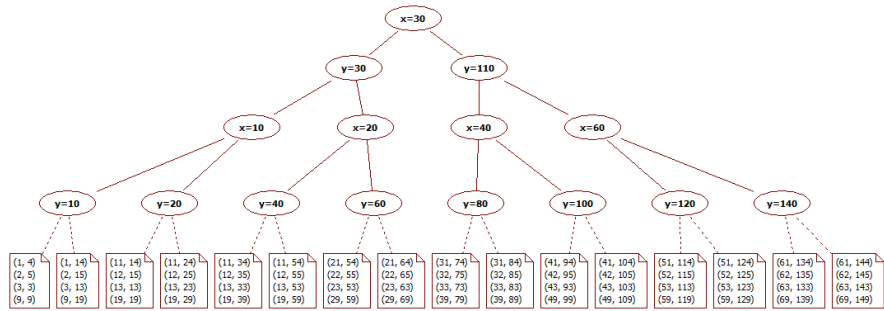
Figure 4.1: A 2-dimensional k-d tree with bucket size of 4.

The Figure 4.2 highlight the nodes visited by the KNN algorithm. Next to each visited nodes there are the steps in which each nodes is traversed. For instance, the node labeled as $y = 110$ is traversed three times during the elaboration of KNN, that is, in the steps n.2, n.10 and n.16.
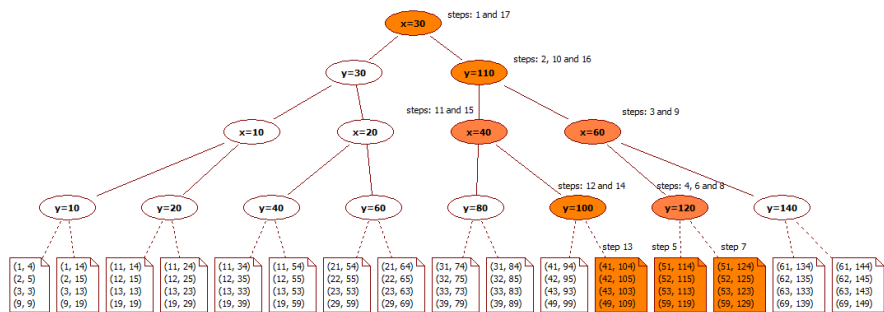

Figure 4.2: Nodes and steps traversed during the elaboration of the KNN algorithm.

Please, note that the KSNP applies to the node $y = 110$. The KSNP is a sufficient condition but it is not necessary. In fact, the node $y = 100$ is traversed by the KNN but the KSNP condition do not apply.

4.2 Ending nodes with k-d trees

Denote the split coordinate related to a node v as $v.splitCoordinate$ and suppose that the $q = (p, k)$, where $p = (x_1, Å\neg x_d)$, for each x_i has two boolean attributes named: $upFromLeft(x_i)$ and $upFromRight(x_i)$ both of them set to $false$ at start. Furthermore, suppose that the KNN algorithm is updated with the following:

Let v be the current node:

1. If KNN algorithm visited the left subtree of v and the right subtree of v must not be visited (i.e. $mustBeVisited$ returns $false$) and $upFromLeft(v.splitCoordinate) = TRUE$ then mark v as an $ending\,node\,for\,v.splitCoordinate$ otherwise set $upFromRight(v.splitCoordinate) = TRUE$.

2. If KNN algorithm visited the right subtree of v and the left subtree of v must not be visited (i.e. $mustBeVisited$ returns $false$) and $upFromRight(v.splitCoordinate) = TRUE$ then mark v as an $ending\,node\,for\,v.splitCoordinate$ otherwise set $upFromLeft(v.splitCoordinate) = TRUE$.

Theorem 4.2 *If a node v is an ending node for each x_i then v is an* ending node. *If the KNN algorithm stops its elaboration in an ending node it returns the correct results of query q.*

Proof: Iterating the the proof of the demonstration for ending nodes in binary tree for each coordinates follows that no subtree will be visited in the path from the ending node to the root.

Example Consider the k-d tree of the Figure 4.2 and the query q of the previous example, the node $y = 110$ in is an ending node.

Chapter 5

Related Works

In the last decade multi-dimensional and high-dimensional indexing in decentralized peer-to-peer (P2P) networks, received extensive research attention. Naturally, most such methods are tree-based and the data space is hierarchically divided into smaller subspaces (regions), such that the higher level data subspace contains the lower level subspaces and acts a guide in searching. These methods can be data-partitioning based, where data subspaces are allowed to overlap (eg. R-tree) or space-partitioning based, where data subspaces are disjoint (eg. k-d tree) and they can be classified into three broad categories: *tree-based*, *DHTs-based* and *skiplist-based*.

5.1 Tree-based

TerraDir [32] is a tree-based structured P2P system. It organizes nodes in a hierarchical fashion according to the underlying data hierarchy. Each query request will be forwarded upwards repeatedly until reaching the node with the longest matching prefix of the query. Then the query is forward to the destination downwards the tree. In TerraDir, each node maintains constant number of neighbors and routing hops are bounded in $O(h)$, where h is the height of the tree. In [2] Mohamed et al. proposed a distributed k-d tree based on MapReduce framework [33]. In such index structures queries are processed similar to the centralized approach, i.e., the query starts in root node and traverse the tree. These methods exhibit logarithmic search cost, but face a serious limitation. Peers that correspond to nodes high in the tree can quickly become overloaded as query processing must pass through them. In centralized indexes this was a desirable property because maintaining these nodes in main

memory allow the minimization of the number of I/O operations. In distribu-
ted indexes it is a limiting factor leading to bottlenecks. Moreover, this causes
an imbalance in fault tolerance: if a peer high in the tree fails than the system
requires a significant amount of effort to recover. Finally, R-trees are known
to suffer in high dimensionality settings, which carries over to their decen-
tralized counterparts; the experiments in [17] showed that for dimensionality
close to 20, this method was outperformed by the non-indexed approach of
[18]. The VBI-tree proposed in [17] provided a solution to the bottlenecks and
imbalance problems introducing a distributed framework (inspired to BATON
– Balanced Tree Overlay Network [34]) based on multidimensional tree struc-
tured overlays, e.g., R-tree. It provides an abstract tree structure on top of an
overlay network that supports any kind of hierarchical tree indexing structu-
res. However, it was shown in [35] that for range queries the VBI-tree suffers
in scalability in terms of throughput. Furthermore, it can cause unfairness as
peers corresponding to nodes high in the tree are heavily hit.

5.2   DHTS-based

Approaches based on distributed hash tables (DHTs) employ a globally con-
sistent protocol to ensure that any peer can efficiently route a search to the
peer that has the desired content, regardless of how rare it is or where it is lo-
cated. A DHT system provides a lookup service similar to a hash table; (key,
value) pairs are stored in a DHT, and any participating node can efficiently
retrieve the value associated with a given key. Responsibility for maintaining
the mapping from keys to values is distributed among the nodes, in such a way
that a change in the set of participants causes a minimal amount of disruption.
This allows a DHT to scale to extremely large numbers of nodes and to handle
continual node arrivals, departures, and failures. DHT systems include Chord
[36], Tapestry [37], Pastry [38], CAN [18] and Koorde [39]. The routing algo-
rithms used in Tapestry and Pastry are both inspired by Plaxton [40]. The idea
of the Plaxton algorithm is to find a neighboring node that shares the longest
prefix with the key in lookup message, repeat this operation until find a des-
tination node that shares the longest possible prefix with the key. In Tapestry
and Pastry, each node has $O(logN)$ neighbors and the routing path takes at
most $O(logN)$ hops. MAAN [41] extends Chord to support multidimensional
range queries by mapping attribute values to the Chord identifier space via uni-
form locality preserving hashing. MAAN and Mercury [42] can support multi-
attribute range queries through single attribute query resolution. They do not

feature pure multidimensional schemes, as they treat attributes independently. As a result, a range query is forwarded to the first value appearing in the range and then it is spread along neighboring peers exploiting the contiguity of the range. This procedure is very costly particularly in MAAN, which prunes the search space using only one dimension. MIDAS [43] is similar to these works and in particular, MIDAS implements a distributed k-d tree, where leaves correspond to peers, and internal nodes dictate message routing. The proposed algorithms process point and range queries over the multidimensional indexed space in $O(logn)$ hops in expectance. Two algorithms for Nearest Neighbor Queries are described: the first (expected $O(logn)$) has low latency and involve a large number of peers; the second (expected $O(log^2n)$) has higher latency but involves far fewer peers.

5.3 Skiplist-based

Skip Graphs [44] and SkipNet [45] are two skip-list based structured P2P systems. Skip Graphs and SkipNet maintain $O(logN)$ neighbors in their routing table. For each node, the neighbor at level h has the distance of 2h to this node, i.e. they are 2h nodes far away. This is very similar to the fingers in Chord. There are $2h$ rings at level h with $n/2h$ nodes per ring. Searching a key in Skip Graphs or SkipNet is started at the top-most level of the node seeking the key. It proceeds along the same level without overshooting the key, continuing at a lower level if required, until it reaches level 0. Their routing hops of searching a key are also $O(logN)$. The above structured P2P systems provide scalable distributed lookup for unique keys. However they can not support efficient search, such as keyword search and multi-dimensional range queries.

Finally, SCRAP [46], ZNet [47], employ a space filling curve , such as Hilbert or z-curve, to map the multidimensional space to a single dimension and then use a conventional system to index the resulting space. For instance, [35] uses the z-curve and P-Grid to support multi-attribute range queries. The problem with such methods is that the locality of the original space cannot be preserved well, especially in high dimensionality. For instance, a rectangular range in the original space corresponds to multiple non-contiguous ranges in the mapped space. As a result a single range query is decomposed to multiple range queries in the mapped space, which increases the processing cost.

5.4   Comparison with Related Works

This section illustrates a set of works at the state of the art related to the topic of the thesis. For each related work this section reports its description and a comparison with the proposal of the thesis.

5.4.1   Index-based Query Processing on Distributed Multidimensional Data (2013) [1]

Description: *Multi-attribute Indexing for Distributed Architecture Systems* (MIDAS) implements a distributed k-d tree, where leaves correspond to peers, and internal nodes dictate message routing. MIDAS distinguishes the concepts of physical and virtual peer. A physical peer is an actual machine responsible for several peers due to node departures or failures, or for load balancing and fault tolerance purposes. A virtual peer in MIDAS corresponds to a leaf of the k-d tree, and stores/indexes all key-value tuples, whose keys reside in the leaves rectangle and for any point in space, there exists exactly one peer in MIDAS responsible for it. Algorithms for reallocation of virtual peers are provided if a new physical peers are added o deleted. Two algorithms $O(logn)$ and $O(log^2n)$ for Nearest Neighbor Queries are also provided. The first one may involve a large number of peers whereas the second has higher latency but involves far fewer peers.

Comparison: The paper does not explicitly mention the term *semantic*. It is possible to think that semantic is embedded in distance function used to build the k-d tree. MIDAS shares with our proposal the usage of k-d trees. A virtual peer in MIDAS contains only partial information about the k-d tree. MIDAS distinguishes the concepts of physical and virtual peer. As our proposal, each virtual peer may start a query. A limitation of this approach could be the fact that initially it assume there is a single physical peer responsible for entire space. In other words they build the entire k-d tree on a single machine. In the case of skewed data distribution a tree balancing could be required. They do not present balancing algorithms.

5.4.2   Distributed k-d Trees for Retrieval from Very Large Image Collections (2011) [2]

Description: A global k-d Tree is built across $M + 1$ machines. The root machine stores the top of the tree, while the leaf machines store the leaves of

the tree. At query time, the root machine directs features to a subset of the leaf machines. They do not build the k-d tree on one machine, they rather build a feature "'distributor'", that represents the top part of the tree, on the root machine. Since can not fit all the features in the database in one machine, they simply subsample the features and use as many as the memory of one machine can take. They claim that this does not affect the performance of the resulting k-d tree since computing the means in the k-d tree construction algorithm subsamples the points anyway. They construct the distributed k-d tree using the *Map Reduce* paradigm.

Comparison: It seems that the root machine could be a bottleneck.

5.4.3 MD-HBase A Scalable Multi-dimensional Data Infrastructure for Location Aware Services (2011) [3]

Description: MD-HBase (Multi Dimensional-HBase), is a scalable data management system for location based sevices. MD-HBase builds two standard index structures the k-d tree and the Quad tree over a range partitioned Key-value store. Their prototype implementation using HBase, a standard open-source Key-value store. They implemented different variants of their proposal and they have made a performance comparison among them.

Comparison: A limitation of this approach could be the centralized query entry point.

5.4.4 Using a distributed quadtree index in peer-to-peer networks (2007) [4]

Description: Their distributed spatial index assigns responsibilities for regions of space to the peers in the system. Using a quadtree, each subregion is uniquely identified by its centroid where the recursive space subdivision lines meet. They call this centroid a control point. They then pass this information as a key and use a key-based routing protocol. A hash function is commonly used to randomize the mapping from keys to address locations. Roughly equal, contiguous blocks of address locations are assigned to each peer. They attach a peer to a region of space and that peer is responsible for all query computations that intersect that region, and for storing the objects that are associated with that region. They use the Chord [48] method to hash these control points. They used the Network Simulator, ns-2

(http://www.isi.edu/nsnam/ns), in tandem with the Georgia-Tech Internetwork Topology Generator, GT-ITM (http://www.cc.gatech.edu/projects/gtitm) in their experiments.

This index is based on key-based routing protocol, named Chord. As side effect, Chord create an implicitly load-balanced method that can handle skewed data distributions.

Comparison: A limitation could be that a query which covers these multiple control points may return the same object a multiple number of times, and thus they have to eliminate such superfluous hits. It is not clear how much does this fact affect the performance.

Conclusions

The main objective of this work is the proposal of index with the following characteristics:

1. Must be used on a large amount of data. The assumption is that it is not possible or convenient to use a single workstation to host all the data.

2. In addition to the traditional search (keyword search), must make search by semantic available. The term "semantic" means the ability to use lexical relations (*hypernymy*, *hyponymy*, *meronymy*, *synonymy*, *holonymy*, etc.) to improve the quality of search. Referring therefore to the ability to index not only structured information ?such as points with numeric coordinates, but also unstructured information as text files.

3. Must be distributed over a computer network and ensure the greatest possible benefits in terms of efficiency (search, insert, delete), i.e. the performance should be as close as to the traditional indexes that use a single workstation.

 The basic ideas behind are:

1. A data structure, called *Random Trees*, based on k-d tree (it has and embedded a k-d tree) that can be deployed over a network of peers. Both Random Trees and k-d trees have the time complexity $O(\log N)$ for search, insertion and deletion operations. The Random Trees represent the main contribution of this work. With a Random Tree distributed over a network of peers a randomly chosen peer can start the propagation of a query in the network and the result will be returned by the first peer that determines that the search is complete. With high probability that peer is not the peer containing the root of the embedded k-d tree. Of course, due the distributed nature of the Random Trees, more than one query can be running at the same time.

2. The indexing of a text written in natural language by means the set of steps:

   (a) Transform of the text in a set $T = \{T_1, ...T_n\}$ of RDF triples

   (b) Define a semantic distance d over T so that (T, d) is a *metric space*

   (c) Define a mapping function $M : T \to R^k$ and calculate the points $P = \{P_1, ...P_n\}$ where $P_i = (x_1, ...x_k) = M(T_i)$

   (d) Insert the points $P_1, ...P_n$ in the Random Tree

   (e) Execute queries over the Random Trees. Suppose the resulting points are $R_1, ...R_m$, return the triples $T_1, ...T_m$ such that $R_i = M(T_i)$.

It was discussed:

1. What a semantic distance between RDF triples is and a set of proposal suggests how it can be defined combining well-known semantic similarity measures such as *Resnik*, *Leacock & Chodorow*, *Wu & Palmer* [25].

2. What a mapping between metric space e vector space is and, in particular, what properties it must have. A well-know algorithm (FastMap) was proposed to calculate the mapping.

An interesting non trivial motivation example related to the software requirements validation (the problem of finding inconsistencies in software requirements written in natural language) was also described [31].

Appendix A

Appendix A

A.1 K-nearest neighbor searching with a k-d tree

The k-nearest neighbor algorithm can be applied of course both with k-d trees and binary trees. Let p be the query point and k the number of neighbors to be extracted from the tree. If T is an instance of the tree and $root$ is its root the result of the query is:

$$result = nearestQuery\,(p, k) \tag{A.1}$$

$Result$ is a data structure that implements a priority queue having at top the farthest point from the query point p and it contains at most k points. When a new instance of a $Result$ is created the query point p and the size k of the queue must be specified. On an instance of a $Result$ the add point operation can be performed. After the new point has been added to queue if the size of the queue exceeds k then the point at the top of the queue is deleted. $Result$ has a flag that indicates if it is full and the operation $getFarthest$ returns the top of the queue.

Algorithm 12 nearestQuery(queryPoint, k)

Require: $queryPoint$ and k are integer values
Ensure: return the k-nearest neighbor points of $queryPoint$ in T
1. $result \leftarrow createResultInstance(queryPoint, k)$ {create a new instance $result$ of $Result$ for queryPoint with size k}
2. nearestNeighbor(root, queryPoint, k, result, 'none')
3. **return** $result$ {$result$ contains the results of the query}

Algorithm 13 nearestNeighbor(v, p, k, result, status)

Require: n is a node of T p and k are integer values, $result$ is an instance of $Result$ and $status$ is a string

1. **if** $status \neq NIL$ **then**
2. $v.status \leftarrow status$
3. **end if**
4. NN(v, p, k, result)

Algorithm 14 NN(v, p, k, result)

Require: v is a node of T, p and k are integer values and $result$ is an instance of $Result$

1. **if** $v.isLeaf$ **then**
2. $result.add(v.getBucket)$
3. **else**
4.   **if** $v.status =' none'$ **and not** $(v.isLeaf)$ **then**
5.   **if** $p < v.splitValue$ **then**
6. $v.status \leftarrow' leftVisited'$
7.   nearestNeighbor(v.left, p, k, result, 'none')
8.   **else**
9. $v.status \leftarrow' rightVisited'$
10.   nearestNeighbor(v.right, p, k, result, 'none')
11.   **end if**
12.   **end if**
13.   **if** $v.status =' rightVisited'$ **then**
14.   **if** $v.left \neq NIL$ **and** $mustBeVisited(v, p, result)$ **then**
15.   nearestNeighbor(v.left, p, k, result, 'none')
16.   **end if**
17.   **end if**
18.   **if** $v.status =' leftVisited'$ **then**
19.   **if** $v.right \neq NIL$ **and** $mustBeVisited(v, p, result)$ **then**
20.   nearestNeighbor(v.right(), p, k, result, 'none')
21.   **end if**
22.   **end if**
23. **end if**

Algorithm 15 mustBeVisited(v, p, result)

Require: v is a node of T, p is integer value and $result$ is an instance of $Result$

1. **if** result is not full **then**
2.   **return true**{do always descend}
3. **end if**
4. **if** $distance(p, v.splitValue) < distance(p, result.getFarthest)$ **then**
5.   **return true**{descend other sibling also}
6. **else**
7.   **return false**{do not descend}
8. **end if**

Appendix A

Appendix B

A.1 K-nearest neighbor searching with a distributed k-d tree

Suppose that:

1. The k-d tree has the set of nodes $N = \{n_1, ..., n_h\}$;

2. There are $PR = \{pr_1, ..., pr_t\}$ processors. Each processor:

   (a) executes the same algorithm;

   (b) can send a message M to any other processor pr calling $SendMessage(pr, M)$;

   (c) has a priority queue containing the messages. After the startup each processor pr waits for a message and on receiving a message the function $onReceivingMessage$ is called.

3. There is a mapping function $MAP : N \rightarrow PR$ between nodes and processor is defined. That is, $MAP(n) = pr$ means that the node n is allocated on processor pr;

4. The query $q = (p, k)$ must be executed on the k-d tree, where $p \equiv (x_1, ..., x_d)$ is a d-dimensional point and k is an integer;

5. Each node $n \in N$:

(a) has a structure $Status$ containing the fields $\langle q, value \rangle$, where $q = (p, k)$ is the current query and $value \in S = \{none, allVisited, rightVisited, leftVisited\}$;

(b) knows the processor pr that hosts it, that is $MAP(n) = pr$.

6. Each message carries the fields $\langle q, v, results, statusValue, priority \rangle$, where $q = (p, k)$ is the current query, $v \in N$, $result$ is the same of section A.1, $statusValue \in S$ and $priority$ is the priority value of the massage.

If $M = \langle q, n, results, status, priority \rangle$ is a message assume, for the sake of simplicity, that it is possible to access to the information in the message M with dot notation. For example, the query point p of the current query $q = (p, k)$ contained in M is $M.q.p$.

On receiving the message M, the $onReceiveMessage$ method starts the distributed search in the current processor.

Algorithm 16 $onReceiveMessage(M)$

Require: M is a message
 1. $DistributedNN(M.q, M.v, M.results, M.status)$

Algorithm 17 $DistributedNN(q, n, results, status, priority)$

Require: q is query, n is a node of T, $results$ is an instance of $Results$, $status$ is an instance of $Status$ and $priority$ is an integer value
 1. **if** $results = NIL$ **then**
 2. $\quad$ {Create a new instance $results$ of type $Results$ for query point $q.p$ and $q.k$ points}
 3. **end if**
 4. $pr \leftarrow MAP(n)$
 5. **if** pr is the current processor **then**
 6. $\quad$ **if** $status \neq NIL$ **then**
 7. $\quad\quad$ $n.setStatus(q, status)$
 8. $\quad$ **end if**
 9. $\quad$ $DNN(q, n, results, priority)$ {same processor}
10. **else**
11. $\quad$ $M \leftarrow \langle q, n, results, status, priority \rangle$
12. $\quad$ $SendMessage(pr, M)$ {another processor}
13. **end if**

Each node n has a distinct $status$ for each q because more than one query could be executed simultaneously. In fact, a new q_1 can start even if the previous query q is still running. Even if a single processor can execute one nearest neighbors at time, it can be idle because it is waiting for the elaboration of q in its subtree that is contained in another processor and it is waiting for a message. In the meantime, the same processor can receive a new message for q_1 and then it starts the elaboration of q_1.

Algorithm 18 $DNN\,(q, n, results, status, priority)$

Require: q is query, n is a node of T, $results$ is an instance of $Results$, $status$ is an instance of $Status$ and $priority$ is an integer value

1. **if** $n.isLeaf$ **then**
2. $results.add\,(q, n.points)$
3.   **if** $n.parent \neq NIL$ **then**
4. $DistributedNN\,(q, n.parent, results, NIL, prty + 1)$
5.   **else**
6. $G \leftarrow \langle q, results \rangle$
7. $SendMessage\,(S, G)$ {the search has been completed. Returns $results$ to sender S}
8.   **end if**
9. **else**
10.   **if** $n.getStatus\,(q) =' leftVisited'$ **then**
11.   **if** $n.right \neq NIL$ **and** $mustBeVisited\,(n.right)$ **then**
12. $n.setStatus(q,' allVisited')$
13. $DistributedNN\,(q, n.right, results,' none', prty + 1)$
14.   **end if**
15.   **end if**
16.   **if** $n.getStatus\,(q) =' rightVisited'$ **then**
17.   **if** $n.left \neq NIL$ **and** $mustBeVisited\,(v.left)$ **then**
18. $n.setStatus\,(q,' allVisited')$
19. $DistributedNN\,(q, n.left, results,' none', prty + 1)$
20.   **end if**
21.   **end if**
22.   **if** $n.getStatus\,(q) =' none'$ **then**
23.   **if** $p\,(v.splitAxis) < v.splitValue$ **then**
24. $n.setStatus(q, v,' leftVisited')$ {Descend Left}
25. $DistributedNN(q, n.left, results,' none', prty + 1)$
26.   **else**
27. $n.setStatus(q, n,' rightVisited')$ {Descend right}
28. $DistributedNN(q, n.right, results,' none', prty + 1)$
29.   **end if**
30.   **end if**
31. **end if**

Please, note that the older messages the higher the priority. Let r the root of the k-d tree and suppose that $pr = MAP\,(r)$. The k-nearest neighbor search

for query point p starts with:

$$M \leftarrow \langle Q, q, v, NIL,' none' \rangle \tag{A.1}$$

$$SendMessage(pr, M) \tag{A.2}$$

At the end of elaboration the data structure results in pr contains the results of the query.

Appendix A

Appendix C

A.1   K-nearest neighbor searching with a k-d tree using ending nodes

The result of the query is:

$$result = nearestQueryWithEndingNode(p, k) \qquad (A.1)$$

The data structure $Result$ of the previous sections is was updated with the boolean attributes $upFromLeft$ and $upFromRight$.

Algorithm 19 $nearestQueryWithEndingNode(queryPoint, k)$

Require: $queryPoint$ and k are an integer values
Ensure: return the k-nearest neighbor points of $queryPoint$ in T
  1. $result \leftarrow createResultInstance(queryPoint, k)$ {create a new instance $result$ of $Result$ for queryPoint with size k}
  2. $result.EndingNode \leftarrow root$
  3. nearestNeighborWithEndingNode(root, queryPoint, k, result, 'none')

Algorithm 20 $nearestNeighborWithEndingNode(v, p, k, result, status)$

Require: v is a node, p and k are an integer values, $result$ is an instance of $Result$ and $status$ is a string
1. **if** $status \neq NIL$ **then**
2. $v.status \leftarrow status$
3. **end if**
4. NNWithEndingNode (v, p, k, result)

Algorithm 21 $NNWithEndingNode(v, p, k, result)$

Require: v is a node, p and k are an integer values, $result$ is an instance of
    $Result$

1. **if** $v.isLeaf$ **then**
2. $result.add(v.Bucket)$
3. **else**
4.   **if** $v.Status =' none'$ **then**
5.   **if** $p < v.SplitValue$ **then**
6. $v.status \leftarrow' leftVisited'$
7.   nearestNeighborWithEndingNode(v.left, p, k, result, 'none')
8.   **else**
9. $v.status \leftarrow' rightVisited'$
10.   nearestNeighborWithEndingNode(v.right, p, k, result, 'none')
11.   **end if**
12.   **end if**
13.   **if** $result.UpFromLeft \neq TRUE$ **or** $result.UpFromRight \neq$
    $TRUE$ **then**
14.   **if** $v.status =' rightVisited'$ **then**
15.   **if** $v.left \neq NIL$ **and** $mustBeVisited(v, p, result)$ **then**
16.   nearestNeighborWithEndingNode(v.left, p, k, result, 'none')
17.   **else**
18. $result.UpFromLeft \leftarrow TRUE$
19.   **end if**
20.   **end if**
21.   **if** $v.status =' leftVisited'$ **then**
22.   **if** $v.right \neq NIL$ **and** $mustBeVisited(v, p, result)$ **then**
23.   nearestNeighborWithEndingNode(v.right, p, k, result, 'none')
24.   **else**
25. $result.UpFromRight \leftarrow TRUE$
26.   **end if**
27.   **end if**
28.   **else**
29.   **if** $result.EndingNode = root$ **then**
30. $result.EndingNode \leftarrow v$
31.   **end if**
32.   **end if**
33. **end if**
34. $v.setStatus \leftarrow NIL$

Please, note that the procedure $NNWithEndingNode$ stops if the following condition is true and it does not elaborate any other subtrees:

$$result.UpFromLeft = TRUE \text{ } OR \text{ } result.UpFromRight = TRUE \qquad (A.2)$$

Appendix A

Appendix D

A.1 Random k-nearest neighbor searching with a k-d tree using ending nodes

These section lists the complete pseudocode of the random k-nearest neighbor algorithm with binary trees. The following code assume that:

1. The function $randInt(inf, sup)$ returns a random integer such that: $inf \leq i \leq sup$

2. The list leftSideNodes contains nodes belonging to the left subtree of the root.

3. The list rightSideNodes contains nodes belonging to the right subtree of the root.

Algorithm 22 $randomNearestQuery(queryPoint, k)$

Require: $queryPoint$ and k are an integer values

1. $randomNodeIndex \leftarrow 0$
2. $randomNode \leftarrow NIL$
3. {select the side}
4. **if** $queryPoint < root.splitValue$ **then**
5.   {choose a node in left subtree}
6. $randomNodeIndex \leftarrow randInt(0, (leftSideNodes.size() - 1))$
7. $randomNode \leftarrow leftSideNodes.get(randomNodeIndex)$
8. **else**
9.   {choose a node in right subtree}
10. $randomNodeIndex \leftarrow randInt(0, (rightSideNodes.size() - 1))$
11. $randomNode \leftarrow rightSideNodes.get(randomNodeIndex)$
12. **end if**
13. $startNode \leftarrow findStartingNode(queryPoint, randomNode)$
14. {create a new object result}
15. $result \leftarrow newResult(queryPoint, k)$
16. $result.endingNode \leftarrow root$
17. randomNearestNeighbor(startNode, queryPoint, k, result, none)
18. {return result}

Algorithm 23 $nearestNeighbor(v, p, k, result, String status)$

Require: v is a node, p and k are an integer values, $result$ is an instance of $Result$ and $status$ is an instance of $Status$

1. **if** $status \neq NIL$ **then**
2. $v.status \leftarrow status$
3. **end if**
4. RandomNN (v, p, k, result)

Algorithm 24 $RandomNN(v, p, k, result)$

Require: v is a node, p and k are an integer values, $result$ is an instance of
    $Result$

1. **if** $v.isLeaf$ **then**
2. $result.add(v.Bucket)$
3.   **if** $mustBeSetParentStatus(v)$ **then**
4.   nearestNeighbor(v.parent, p, k, result, null)
5.   **end if**
6. **else**
7.   **if** $v.Status =' none'$ **then**
8.   **if** $p < v.SplitValue$ **then**
9. $v.status \leftarrow' leftVisited'$
10.   nearestNeighbor(v.left, p, k, result, 'none')
11.   **else**
12. $v.status \leftarrow' rightVisited'$
13.   nearestNeighbor(v.right, p, k, result, 'none')
14.   **end if**
15.   **end if**
16.   **if** $result.UpFromLeft \neq TRUE$ **or** $result.UpFromRight \neq TRUE$ **then**
17.   **if** $v.status =' rightVisited'$ **then**
18.   **if** $v.left \neq NIL$ **and** $mustBeVisited(v, p, result)$ **then**
19.   nearestNeighbor(v.left, p, k, result, 'none')
20.   **else**
21. $result.UpFromLeft \leftarrow TRUE$
22.   **end if**
23.   **if** $mustBeSetParentStatus(v)$ **then**
24.   nearestNeighbor(v.parent, p, k, result, null)
25.   **end if**
26.   **end if**
27.   **if** $v.status =' leftVisited'$ **then**
28.   **if** $v.right \neq NIL$ **and** $mustBeVisited(v, p, result)$ **then**
29.   nearestNeighbor(v.right, p, k, result, 'none')
30.   **else**
31. $result.UpFromRight \leftarrow TRUE$
32.   **end if**
33.   **if** $mustBeSetParentStatus(v)$ **then**
34.   nearestNeighbor(v.parent, p, k, result, null)
35.   **end if**
36.   **end if**
37.   **else**
38.   **if** $result.EndingNode = root$ **then**
39. $result.EndingNode \leftarrow v$
40.   **end if**
41.   **end if**
42. **end if**
43. $v.setStatus \leftarrow null$

Appendix A

Appendix E

A.1 Starting Node Property with min/max values (SNPMinMax)

If the internal nodes of the tree also contain the minimum and maximum values of the points contained its subtree (leaves already have this information) the *Starting Node Property* could be rewritten as:

Theorem A.1 *Let* $M = \{m_1, ..., m_j\}$ *be the set of nodes visited by the KNN algorithm during the execution of the query* $q = (p, k)$. *If the point p is in the tree then (SNPMinMax):*

$$m.minValue \leq p \leq m.maxValue \Leftrightarrow m \in M \qquad (A.1)$$

The *SNPMinMax* is a necessary and sufficient condition to determine whether a node is visited by the KNN algorithm. If the node is a leaf $m.minValue$ and $m.maxValue$ are respectively the minimum and maximum values contained in the bucket.

Proof: $m.minValue \leq p \leq m.maxValue \Rightarrow m \in M$. The proof is the same of SNP.

$m \in M \Rightarrow m.minValue \leq p \leq m.maxValue$. Suppose m is a left child of $m.parent$, then:

$$p \leq m.parent.splitValue \qquad (A.2)$$

Note that:

$$m.maxValue = m.parentSplitValue \qquad (A.3)$$

Then:
$$p \leq m.maxValue \tag{A.4}$$

The proof of the other inequality:
$$m.minValue \leq p \tag{A.5}$$

is by induction. Because p is in the tree then for the root node holds that:
$$r.minValue \leq p \tag{A.6}$$

Suppose the inequality applies for the parent of m, that is:
$$m.parent.minValue \leq p \tag{A.7}$$

Then:
$$m.parent.minValue = m.minValue \tag{A.8}$$

It follows that the inequality applies to m also:
$$m.minValue \leq p. \tag{A.9}$$

If m is a right child the proof is symmetrical.

The $findStartingNodeminMax$ uses SNPMinMax property to find a starting node.

Algorithm 25 $findStartingNodeMinMax(p, n)$

Require: p is an integer vale and n is a node
Ensure: return a starting node $m \in M$ for query point p
  1. **if** $n.isRoot$ **then**
  2.   **return** n
  3. **else**
  4.   **if** $n.minValue \leq p \leq n.maxValue$ **then**
  5.   **return** n
  6.   **else**
  7.   **return** findStartingNodeMinMax(p, n.parent())
  8.   **end if**
  9. **end if**

Table 2.6 shows test results of the execution of $testAverageFindStartingNode$ using $findStartingNodeMinMax$ algorithm.

Table A.1: Test results of $testAverageFindStartingNode$ with $findStartingNodeMinMax$ algorithm.

bucket dim.	%	Number of points in the tree							average
		512	1024	2048	4096	8192	16384	32768	
5	root	65	65	65	65	65	65	65	*65*
	root	35	35	35	35	35	35	35	*35*
10	root	65	65	65	65	65	65	65	*65*
	no root	35	35	35	35	35	35	35	*35*
20	root	64	65	65	65	65	65	65	*64.9*
	no root	36	35	35	35	35	35	35	*35.1*
30	root	66	67	67	67	67	67	67	*66.9*
	no root	34	33	33	33	33	33	33	*33.1*
40	root	63	64	65	65	65	65	65	*64.6*
	no root	37	35	35	35	35	35	35	*35.4*
average	root	*64.6*	*65.2*	*65.4*	*65.4*	*65.4*	*65.4*	*65.4*	**65.3**
	no root	*35.4*	*34.8*	*34.6*	*34.6*	*34.6*	*34.6*	*34.6*	**34.7**

A comparison between Table 2.7 and Table 2.9 shows that there are no significant improvement using the $findStartingNodeMinMax$ instead of $findStartingNode$.

Appendix A

Appendix F

A.1 Iterating the $findStartingNode$ algorithm approach

It is interesting to try to extend the $findStartingNode$ algorithm approach to the second level of the tree. Basically, node are labeled as:

1. left children of left child of the root as *left-left*;

2. right children of right child of the root as *left-right*;

3. left children of right child of the root as *right-left*;

4. right children of right child of the root as *right-right*;

The new $findStartingNodeSide2$ algorithm is the algorithm 26.

Algorithm 26 $findStartingNodeSide2\,(queryPoint)$

Require: $queryPoint$ is an integer value
Ensure: return a starting node for query point p

  1. **if** $queryPoint < root.splitValue$ **then**
  2.   **if** $queryPoint < root.left.splitValue$ **then**
  3.   {let randomNode a left-left randomly chosen node}
  4.   **else**
  5.   {let randomNode a left-right randomly chosen node}
  6.   **end if**
  7. **else**
  8.   **if** $queryPoint < root.right.splitValue$ **then**
  9.   {let randomNode a right-left randomly chosen node}
 10.   **else**
 11.   {let randomNode a right-right randomly chosen node}
 12.   **end if**
 13. **end if**
 14. $start \leftarrow tree.findStartingNode(queryPoint, randomNode)$
 15. **return** start

The Table A.1 show the results of the same tests of Table 2.7 carried out on the $findStartingNodeSide2$.

Table A.1: Test results of $testAverageFindStartingNode$ with $findStartingNodeSide2$ algorithm.

bucket dim.	%	Number of points in the tree							average
		512	1024	2048	4096	8192	16384	32768	
5	root	34	34	34	34	34	34	34	*34*
	no root	66	66	66	66	66	66	66	*66*
10	root	38	39	38	38	38	38	38	*38.1*
	no root	62	61	62	62	62	62	62	*61.9*
20	root	38	38	39	38	38	38	38	*38.1*
	no root	62	62	61	62	62	62	62	*61.9*
30	root	35	36	37	34	34	34	34	*34.9*
	no root	66	64	63	66	66	66	66	*65.1*
40	root	38	38	38	38	38	38	38	*38*
	no root	62	62	62	62	62	62	62	*61*
average	root	*36.6*	*37*	*37.2*	*36.4*	*36.4*	*36.4*	*36.4*	**36.6**
	no root	*63.2*	*63*	*62.8*	*63.6*	*63.6*	*63.6*	*63.6*	**63.4**

A comparison between Table A.1 and Table 2.7 shows that there are no significant improvement using the $findStartingNode2$ instead of $findStartingNode$

Appendix A

Appendix G

Follows a brief introduction to the Automatic Query Expansion (AQE) although it is not directly related to Multi-dimensional data structures. The AQE is a widespread approach for information retrieval, and it can provide useful insights to the definition of new measures of semantic similarity.

A.1 Automatic Query Expansion state of the art

Search Engines are essential tools form most computer users in a wide variety of context. As result, Information Retrieval has become an important field of research over the last 30 years or so. Many document indexing and retrieval have been proposed which have been shown to be generally effective. However, a deeper analysis reveals that even though these techniques improve performance on average, there is often wide variation in the impact of a particular technique on the retrieval effectiveness for individual queries. The problem knows as the vocabulary problem Furnas et al. [49] is the most critical issue for the for retrieval effectiveness: the indexers and the users do often not use the same words. Furthermore, synonymy (different words with the same or similar meanings, such as *tv* and *television* together with word inflections (such as with plural forms, *television* versus *televisions*, may result in a failure to retrieve relevant documents, with a decrease in recall (the ability of the system to retrieve all relevant documents) and polysemy (same word with different meanings, such as ÔªavaÔ© may cause retrieval of erroneous or irrelevant documents, thus implying a decrease in precision (the ability of the system to retrieve only relevant documents).

To cope with the problem of vocabulary, different approaches have been

proposed: interactive query refinement, relevance feedback, word sense disambiguation and search results clustering. One of the most common techniques is to expand the original adding related keywords to a (typically short) query provided by a user. These additional keywords (or expansion terms) generally increase the likelihood of a match between the query and relevant documents during retrieval, thereby improving user satisfaction. Automatic query expansion has a long history in information retrieval (IR), as has been suggested since 1960 by Maron and Kuhns [50].

According to [51] AQE techniques can be classified into five main groups according to the conceptual paradigm used for finding the expansion features: linguistic methods, corpus-specific statistical approaches, query-specific statistical approaches, search log analysis and Web data.

Linguistic methods leverage global language properties such as morphological, lexical, syntactic and semantic word relationships to expand or reformulate query terms. They are typically based on dictionaries, thesauri, or other similar knowledge representation sources such as WordNet. These techniques are usually generated independently of the full query and of the content of the database being searched, they are usually more sensitive to word sense ambiguity. In [52], a semantic query expansion methodology called SQX-Lib, that combines different techniques, such as lemmatization, NER and semantics, for information extraction from a relational repository is presented. It includes a disambiguation engine that calculates the semantic relation between words in case it finds ambiguities and selects the best meaning for those words. SQX-Lib is integrated in a real major Media Group in Spain. Dalton et al. [53] proposed an AQE method based on annotations of entities from large general purpose knowledge bases, such as Freebase and the Google Knowledge Graph. They proposed a new technique, called entity query feature expansion (EQFE) which enriches the query with features from entities and their links to knowledge bases, including structured attributes and text. One limitation of this work is that it depends upon the success and accuracy of the entity annotations and linking. Bouchoucha et al. [54] presented a unified framework to integrate multiple resources for Diversified Query Expansion. By implementing two functions, one to generate expansion term candidates and the other to compute the similarity of two terms, any resource can be plugged into their framework. Experimental results show that combining several complementary resources performs better than using one single resource. In [55] a new way of using WordNet for Query Expansion is proposed with a combination of three QE methods that takes into account different aspects of a candidate expansion

terms usefulness. For each candidate expansion term, this method considers its distribution, its statistical association with query terms, and also its semantic relation with the query.

Corpus-specific statistical approaches analyze the contents of a full database to identify features used in similar ways. Most early statistical approaches to AQE were corpus-specific and generated correlations between pairs of terms by exploiting term co-occurrence, either at the document level, or to better handle topic drift, in more restricted contexts such as paragraphs, sentences, or small neighborhoods. In [56], a new method, called AdapCOT, which applies co-training in an adaptive manner to select feedback documents for boosting QEs effectiveness is proposed. Co-training is an effective technique for classification over limited training data, which is particularly suitable for selecting feedback documents. The proposed AdapCOT method makes use of a small set of training documents, and labels the feedback documents according to their quality through an iterative process. Colace et al. [57] used a minimal relevance feedback to expand the initial query with a structured representation composed of weighted pairs of words. Such a structure is obtained from the relevance feedback through a method for pairs of words selection based on the Probabilistic Topic Model. The proposed approach computes the expanded queries considering only endogenous knowledge

Query-specific techniques take advantage of the local context provided by the query. They can be more effective than corpus-specific techniques because the latter might be based on features that are frequent in the collection but irrelevant for the query at hand. Ermakova et al. [58] proposed a new automatic QE method that estimates the importance of expansion candidate terms by the strength of their relation to the query terms. The method combines local analysis and global analysis of texts. Yang [59] suggested a method that applies a linguistic filter and a C-value method to extend the query terms, and then uses the Normalized Google Distance-based method to calculate the term weight and choose Top-N terms as extended query. They claim that the Normalized Google Distance (NGD) with some global factors enhance the relevance between initial query and extended query, and improve the accuracy of the search results of the expert finding system.

Search log analysis paradigm is based on analysis of search logs. The idea is to mine query associations that have been implicitly suggested by Web users, thus bypassing the need to generate such associations in the first place by content analysis. Search logs typically contain user queries, followed by the URLs of Web pages that are clicked by the user in the corresponding search

results page. One advantage of using search logs is that they may encode implicit relevance feedback, as opposed to strict retrieval feedback. Web data techniques rely on the presence of the anchor texts. Anchor texts and real user search queries are very similar because most anchor texts are succinct descriptions of the destination page.

Bibliography

[1] Sheneela Naz, Muhammad Naeem, and Amir Qayyum. Performance evaluation of index schemes for semantic cache. *International Journal of Information Technology and Computer Science (IJITCS)*, 5(4):40, 2013.

[2] Mohamed Aly, Mario Munich, and Pietro Perona. Distributed kd-trees for retrieval from very large image collections. In *British Machine Vision Conference, Dundee, Scotland*, 2011.

[3] Shoji Nishimura, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. Md-hbase: a scalable multi-dimensional data infrastructure for location aware services. In *Mobile Data Management (MDM), 2011 12th IEEE International Conference on*, volume 1, pages 7–16. IEEE, 2011.

[4] Egemen Tanin, Aaron Harwood, and Hanan Samet. Using a distributed quadtree index in peer-to-peer networks. *The VLDB Journal*, 16(2):165–178, 2007.

[5] Michael Hausenblas, Wolfgang Halb, Yves Raimond, and Tom Heath. What is the size of the semantic web. *Proceedings of I-Semantics*, pages 9–16, 2008.

[6] Chris Bizer, Richard Cyganiak, Tom Heath, et al. How to publish linked data on the web. 2007.

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

[8] Nickolai Toupikov, Jürgen Umbrich, Renaud Delbru, Michael Hausenblas, and Giovanni Tummarello. Ding! dataset ranking using formal descriptions. In *LDOW*. Citeseer, 2009.

[9] Michael Stonebraker, Samuel Madden, Daniel J Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era:(it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007.

[10] Aidan Hogan, Andreas Harth, Jürgen Umbrich, Sheila Kinsella, Axel Polleres, and Stefan Decker. Searching and browsing linked data with swse: The semantic web search engine. *Web semantics: science, services and agents on the world wide web*, 9(4):365–401, 2011.

[11] Gong Cheng and Yuzhong Qu. Searching linked objects with falcons: Approach, implementation and evaluation. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(3):49–70, 2009.

[12] Christian Bizer, Tom Heath, Kingsley Idehen, and Tim Berners-Lee. Linked data on the web (ldow2008). In *Proceedings of the 17th international conference on World Wide Web*, pages 1265–1266. ACM, 2008.

[13] Zachary D Stephens, Skylar Y Lee, Faraz Faghri, Roy H Campbell, Chengxiang Zhai, Miles J Efron, Ravishankar Iyer, Michael C Schatz, Saurabh Sinha, and Gene E Robinson. Big data: astronomical or genomical? *PLoS Biol*, 13(7):e1002195, 2015.

[14] Michael Hausenblas and Marcel Karnstedt. Understanding linked open data as a web-scale database. In *Advances in Databases Knowledge and Data Applications (DBKDA), 2010 Second International Conference on*, pages 56–61. IEEE, 2010.

[15] Hanan Samet. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.

[16] Stefan Berchtold, Daniel A Keim, and Hans-Peter Kriegel. The x-tree: An index structure for high-dimensional data. *Readings in multimedia computing and networking*, page 451, 2001.

[17] HV Jagadish, Beng Chin Ooi, Quang Hieu Vu, Rong Zhang, and Aoying Zhou. Vbi-tree: A peer-to-peer framework for supporting multidimensional indexing schemes. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 34–34. IEEE, 2006.

[18] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. *A scalable content-addressable network*, volume 31. ACM, 2001.

[19] Taewon Lee, Bongki Moon, and Sukho Lee. Bulk insertion for r-trees by seeded clustering. *Data & Knowledge Engineering*, 59(1):86–106, 2006.

[20] Lars Arge, Klaus H Hinrichs, Jan Vahrenhold, and Jeffrey Scott Vitter. Efficient bulk operations on dynamic r-trees. *Algorithmica*, 33(1):104–128, 2002.

[21] Pierpaolo Basile, Marco de Gemmis, Anna Lisa Gentile, Pasquale Lops, and Giovanni Semeraro. Uniba: Jigsaw algorithm for word sense disambiguation. In *Proceedings of the 4th International Workshop on Semantic Evaluations*, pages 398–401. Association for Computational Linguistics, 2007.

[22] Daniel Gerber and Axel-Cyrille Ngonga Ngomo. Extracting multilingual natural-language patterns for rdf predicates. In *International Conference on Knowledge Engineering and Knowledge Management*, pages 87–96. Springer, 2012.

[23] Peter Exner and Pierre Nugues. Entity extraction: From unstructured text to dbpedia rdf triples. In *The Web of Linked Entities Workshop (WoLE 2012)*, pages 58–69. CEUR, 2012.

[24] Jennifer Golbeck, Michael Grove, Bijan Parsia, Adtiya Kalyanpur, and James Hendler. New tools for the semantic web. In *International Conference on Knowledge Engineering and Knowledge Management*, pages 392–400. Springer, 2002.

[25] Martin Warin and HM Volk. Using wordnet and semantic similarity to disambiguate an ontology. *Retrieved January*, 25:2008, 2004.

[26] Christiane Fellbaum. *WordNet*. Wiley Online Library, 1998.

[27] Federica Mandreoli and Riccardo Martoglia. Knowledge-based sense disambiguation (almost) for all structures. *Information Systems*, 36(2):406–430, 2011.

[28] Joseph B Kruskal and Myron Wish. *Multidimensional scaling*, volume 11. Sage, 1978.

[29] Christos Faloutsos and King-Ip Lin. *FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets*, volume 24. ACM, 1995.

[30] Gauthier Fanmuy, Anabel Fraga, and Juan Llorens. Requirements verification in the industry. In *Complex Systems Design & Management*, pages 145–160. Springer, 2012.

[31] Francesco Gargiulo, Gabiella Gigante, and Massimo Ficco. A semantic driven approach for requirements consistency verification. *International Journal of High Performance Computing and Networking*, 8(3):201–211, 2015.

[32] Bujor Silaghi, Samrat Bhattacharjee, and Peter J Keleher. Query routing in the TerraDir distributed directory. In *ITCom 2002: The Convergence of Information Technologies and Communications*, pages 299–309. International Society for Optics and Photonics, 2002.

[33] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[34] Hosagrahar V Jagadish, Beng Chin Ooi, and Quang Hieu Vu. Baton: A balanced tree structure for peer-to-peer networks. In *Proceedings of the 31st international conference on Very large data bases*, pages 661–672. VLDB Endowment, 2005.

[35] Spyros Blanas and Vasilis Samoladas. Contention-based performance evaluation of multidimensional range search in peer-to-peer networks. *Future Generation Computer Systems*, 25(1):100–108, 2009.

[36] Hari Balakrishnan, M Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Looking up data in P2P systems. *Communications of the ACM*, 46(2):43–48, 2003.

[37] Ben Y Zhao, Ling Huang, Jeremy Stribling, Sean C Rhea, Anthony D Joseph, and John D Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *Selected Areas in Communications, IEEE Journal on*, 22(1):41–53, 2004.

[38] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, pages 329–350. Springer, 2001.

[39] M Frans Kaashoek and David R Karger. Koorde: A simple degree-optimal distributed hash table. In *Peer-to-peer systems II*, pages 98–107. Springer, 2003.

[40] C Greg Plaxton, Rajmohan Rajaraman, and Andrea W Richa. Accessing nearby copies of replicated objects in a distributed environment. *Theory of Computing Systems*, 32(3):241–280, 1999.

[41] Min Cai, Martin Frank, Jinbo Chen, and Pedro Szekely. Maan: A multi-attribute addressable network for grid information services. *Journal of Grid Computing*, 2(1):3–14, 2004.

[42] Ashwin R Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: supporting scalable multi-attribute range queries. *ACM SIGCOMM Computer Communication Review*, 34(4):353–366, 2004.

[43] George Tsatsanifos, Dimitris Sacharidis, and Timos Sellis. Index-based query processing on distributed multidimensional data. *GeoInformatica*, 17(3):489–519, 2013.

[44] James Aspnes and Gauri Shah. Skip graphs. *ACM Transactions on Algorithms (TALG)*, 3(4):37, 2007.

[45] Nicholas JA Harvey, Michael B Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *USENIX Symposium on Internet Technologies and Systems*, volume 274. Seattle, WA, USA, 2003.

[46] Prasanna Ganesan, Beverly Yang, and Hector Garcia-Molina. One torus to rule them all: multi-dimensional queries in P2P systems. In *Proceedings of the 7th International Workshop on the Web and Databases: colocated with ACM SIGMOD/PODS 2004*, pages 19–24. ACM, 2004.

[47] Yanfeng Shu, Beng Chin Ooi, K-L Tan, and Aoying Zhou. Supporting multi-dimensional range queries in peer-to-peer systems. In *Peer-to-Peer Computing, 2005. P2P 2005. Fifth IEEE International Conference on*, pages 173–180. IEEE, 2005.

[48] Ion Stoica, Robert Morris, David Karger, M Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[49] George W. Furnas, Thomas K. Landauer, Louis M. Gomez, and Susan T. Dumais. The vocabulary problem in human-system communication. *Communications of the ACM*, 30(11):964–971, 1987.

[50] Melvin Earl Maron and John L Kuhns. On relevance, probabilistic indexing and information retrieval. *Journal of the ACM (JACM)*, 7(3):216–244, 1960.

[51] Claudio Carpineto and Giovanni Romano. A survey of automatic query expansion in information retrieval. *ACM Computing Surveys (CSUR)*, 44(1):1, 2012.

[52] María G Buey, Ángel Luis Garrido, and Sergio Ilarri. An approach for automatic query expansion based on nlp and semantics. In *Database and Expert Systems Applications*, pages 349–356. Springer, 2014.

[53] Jeffrey Dalton, Laura Dietz, and James Allan. Entity query feature expansion using knowledge base links. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, pages 365–374. ACM, 2014.

[54] Arbi Bouchoucha, Xiaohua Liu, and Jian-Yun Nie. Integrating multiple resources for diversified query expansion. In *Advances in Information Retrieval*, pages 437–442. Springer, 2014.

[55] Dipasree Pal, Mandar Mitra, and Kalyankumar Datta. Improving query expansion using wordnet. *Journal of the Association for Information Science and Technology*, 65(12):2469–2478, 2014.

[56] Jimmy Xiangji Huang, Jun Miao, and Ben He. High performance query expansion using adaptive co-training. *Information Processing & Management*, 49(2):441–453, 2013.

[57] Francesco Colace, Massimo De Santo, Luca Greco, and Paolo Napoletano. Weighted word pairs for query expansion. *Information Processing & Management*, 51(1):179–193, 2015.

[58] Liana Ermakova, Josiane Mothe, and Irina Ovchinnikova. Query expansion in information retrieval: What can we learn from a deep analysis of queries? In *International Conference on Computational Linguistics-Dialogue 2014*, volume 20, pages pp–162, 2014.

[59] Kai-Hsiang Yang, Yu-Li Lin, and Chen-Tao Chuang. Using google distance for query expansion in expert finding. In *Digital Information Management (ICDIM), 2014 Ninth International Conference on*, pages 104–109. IEEE, 2014.