Full-fledged Semantic Indexing and Querying Model designed for Seamless Integration in Legacy RDBMS

Joe Tekli[1*], Richard Chbeir[2], Agma J.M. Traina[3], Caetano Traina Jr.[3], Kokou Yetongnon[4], Carlos Raymundo Ibanez[5], Marc Al Assad[1], and Christian Kallas[1]

[1] Lebanese American University, ECE Dept., Byblos, Lebanon
[2] University of Pau & Pays Adour, LIUPPA Lab., Anglet, France
[3] University of Sao Paulo, ICMC, Sao Carlos, Brazil
[4] University of Bourgogne, LE2I Lab. UMR-CNRS, 9 Alain Savary, 21000 Dijon, France
[5] Universidad Peruana de Ciencias Aplicadas, Lima, Peru

Abstract. In the past decade, there has been an increasing need for semantic-aware data search and indexing in textual (structured and NoSQL) databases, as full-text search systems became available to non-experts where users have no knowledge about the data being searched and often formulate query keywords which are different from those used by the authors in indexing relevant documents, thus producing noisy and sometimes irrelevant results. In this paper, we address the problem of semantic-aware querying and provide a general framework for modeling and processing semantic-based keyword queries in textual databases, i.e., considering the lexical and semantic similarities/disparities when matching user query and data index terms. To do so, we design and construct a semantic-aware inverted index structure called *SemIndex*, extending the standard inverted index by constructing a tightly coupled inverted index graph that combines two main resources: a semantic network and a standard inverted index on a collection of textual data. We then provide a general keyword query model with specially tailored query processing algorithms built on top of *SemIndex*, in order to produce semantic-aware results, allowing the user to choose the results' semantic coverage and expressiveness based on her needs. To investigate the practicality and effectiveness of *SemIndex*, we discuss its physical design within a standard commercial RDBMS allowing to create, store, and query its graph structure, thus enabling the system to easily scale up and handle large volumes of data. We have conducted a battery of experiments to test the performance of *SemIndex*, evaluating its construction time, storage size, query processing time, and result quality, in comparison with legacy inverted index. Results highlight both the effectiveness and scalability of our approach.

Keywords: Semantic Queries, Inverted index, NoSQL indexing, Semantic Network, Semantic-aware data processing, Textual databases.

1. Introduction

Processing keyword-based queries is a fundamental problem in the domains of Information Retrieval (IR) and more recently textual DataBase (DB) search, where several studies have been conducted to develop effective keyword-based search techniques, e.g., [10, 31]. In most existing approaches, standard containment keyword queries are supported by a full-text index, namely an *inverted index* which is considered as one of the most useful full-text indexing techniques for large textual collections [8], supported by many DB Management Systems (i.e., DBMSs) [2, 58], and recently extended toward semi-structured [1, 10] and NoSQL data [37, 93].

Inverted indexes associate each term (word/expression) in the text with a list of pointers to the data objects (e.g., data records, or documents) that contain the term, in the form of a list of (*term*, *objectIDs*[]). Then when an enquiry is performed, the index is queried with every term within the user's request, identifying all data objects that contain the query terms in just one search operation [52, 60]. Nonetheless, the standard inverted index, only supports exact term matching and cannot deal with cases of lexical and/or semantic similarities/relationships among query/data terms (despite the use of basic language pre-processing capabilities, like stemming or stop word removal, which only help support basic lexical disparities among terms).

1.1. Motivation Scenarios

To illustrate this, consider a dataset Δ from a movie database, as shown in Table 1. Each movie in Δ, identified with an *id*, is described with some text, including the movie *title*, ***year*** and *plot*. An extract of Δ's inverted index is shown in Fig. 2.a. For queries *"sprint car racer"* and *"sound of music"*, the search results are movies O_2 and

* Corresponding author. Tel.: +9619547262; fax: +9619546262; e-mail: joe.tekli@lau.edu.lb

O_3 respectively, which texts contain occurrences of each of the corresponding query's terms. However, if the user wants to search for a particular movie but cannot recall its exact title or plot description, she will likely use her own terminology in choosing query terms which (we naturally assume) are lexically and/or semantically similar to the movie's description terms, e.g., *"voice of melody"* or *"auto rallying"*. Such terms might not exactly match those used to describe (and index) the movie objects (which is the case in our example), and thus will miss movies O_2 and O_3 as relevant results. In addition, the movies might not be extensively described or well-tagged in the database, or might not be described using the same attributes (e.g., in a NoSQL or semi-structured database), which would also result in missing relevant search results. Similar scenarios and needs can be identified in various areas, e.g.:

- A database storing research proposals granted by different funding agencies (describing the research itself, the granting institutions, and the involved researchers' expertise): Could a scientist user easily retrieve information related to her own research? Could a non-scientist user, e.g., a company manager, with a specific production problem, find the projects, institutions, or researchers able to solve her problem?
- A database storing information related to airline disasters (describing airplanes, crashes, investigations, findings, and so on): Could an investigator efficiently retrieve information related to a given new case investigation path?

Table 1. Sample Movie data collection extracted from IMDB[1].

ID	Textual content
O_1	*Street Kinds (***2008***): Tom Ludlow is a ruthless undercover cop. Locating his stolen car at a gang's hideout, Tom storms in to find thugs getting high on Zen. He hears a light voice…*
O_2	*Days of Thunder* (**1990**): *Cole Trickle is a young racer from California with years of experience in open-wheel racing winning championships in sprint car racing…*
O_3	*Sound of Music, The (***1965***): Maria had longed to be a nun since she was a young girl, yet when she became old enough discovered that it wasn't at all what she thought...*

1.2. Challenges

In the above scenarios, the textual descriptions may involve terms with multiple meanings (*homonymy*, e.g., term "paper" could mean *scientific publication* or *paper sheet*), terms implied by other terms (*metonymy*, e.g., term "wings" implies *airplane*, "suit" implies a *business person*), several terms having the same meaning (*synonymy*, e.g., terms "plane", "airplane", and "aircraft"), or terms related by some semantic relation (e.g., *hypernymy* (*isA*), *holonymy* (*partOf*), such as *plane-isA-machine*, or *wing-partOf-plane*). Hence, when the user needs to search for information using traditional keyword queries based on typical inverted indexes, she will have to manually and iteratively formulate multiple keyword combinations to be evaluated through the inverted index, verifying the results and re-formulating the query accordingly at each iteration, in the hope of finally retrieving relevant results, which is naturally time and effort consuming, as well as error prone.

Solving this issue has been the main motivation for developing so-called *semantic-aware* or *knowledge-aware* (keyword) query systems, which have emerged since the past decade as a natural extension to traditional containment queries, encouraged by (non-expert) user demands. Most existing works in this area (cf. Background in Section 8) have incorporated semantic knowledge at the *query processing* level, to: i) pre-process queries using query rewriting/relaxation and query expansion [19, 29, 62], ii) disambiguate queries using semantic disambiguation and entity recognition techniques [19, 54, 70], and/or iii) post-process query results using semantic result organization and re-ranking [70, 81, 95]. Yet, various challenges remain unsolved, namely: i) time latencies when involving query pre-processing and post-processing [29, 62], ii) complexity of query rewriting/relaxation and query disambiguation requiring context information (e.g., user profiles or query logs) which is not always available [33, 56], and iii) limited user involvement, where the user is usually constrained to providing feedback and/or performing query refinement after the first round of results has been provided by the system [21, 67].

In this work, we adopt another alternative: having an adapted index structure able to integrate and extend textual information with domain knowledge (not only at the querying level, but rather) at the most basic *data indexing* level, providing a semantic-aware inverted index capable of supporting semantic-based querying, and allowing to answer most challenges identified above.

[1] Internet Movie DataBase (http://www.imdb.com/).

1.3. Index Design Strategy

In short, our proposal consists in combining two resources, a *textual data collection* (represented as a traditional inverted index), and a *semantic knowledge base* (represented as a traditional semantic network), in order to build a stand-alone semantic-aware inverted index structure, called *SemIndex*. Yet, this can be performed following three different strategies:

1) *Including semantic knowledge into an inverted index*. The main idea consists in adding an additional entry in the index structure to designate semantic concepts [50] or to link related concepts together [101]. In other words, the traditional (*term*, *objectIDs*[]) index is extended toward some form of (*term*, *context*, *objectIDs*[]) structure where *contexts* designate the semantic meanings of terms (expressed as: concepts, senses, or references) extracted from the knowledge base. While this approach seems simple and straightforward, it can nonetheless lead to a potential explosion in the index size depending on the number of concepts in the knowledge base, thus worsening querying capabilities and system performance.
2) *Including full-text information into the semantic knowledge base*, i.e., adding textual terms to the knowledge base as concept instances, linked using dedicated semantic relationships [11, 92]. For instance, adding new triples of the form *term_occurs-in-context_concept* to the knowledge base, such that each term can be related to a certain ontological concept, when used in a certain context. Yet, one can clearly realize this approach risks exploding the knowledge base size, depending on the number of terms in the text corpus being semantically enhanced. Moreover, extra processing overhead is required to link terms with concepts using meaningful semantic relationships in the knowledge base.
3) *Building an integrated hybrid structure*, i.e., somehow combining the powerful functionalities of inverted indexing with semantic processing capability to allow semantic aware querying, while avoiding the above mentioned limitations of alternative semantic indexing strategies. In our current study, we investigate the latter approach to fully and efficiently support full-text semantic search. Enclosing semantic knowledge directly into the inverted index, and doing it offline – prior to online query execution, underlines major potential benefits over existing methods namely: i) providing more opportunities toward both speed-ups and semantic-based filtering, thus minimizing the need for sophisticated (and time/effort consuming) query pre- and post-processing, ii) finding semantically relevant results without having to perform expensive query disambiguation, iii) allowing end-users to be involved in the whole process: during initial query writing while manipulating the semantic-aware index, and then performing query rewriting (if needed).

1.4. Overall Architecture and Organization

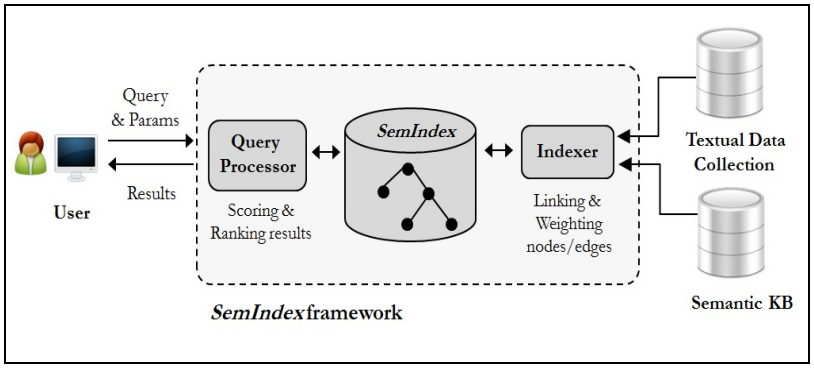


Fig. 1. Overall architecture of *SemIndex* framework.

This paper describes how to design and construct *SemIndex*, and how to use it to process semantic-aware queries. An extended query model with different levels of semantic awareness is also defined, so that both semantic-aware queries and standard containment queries are processed within the same framework. Fig. 1 depicts the overall framework of our approach and its main components. Briefly, the *Indexer* manages the index generation and maintenance, while the *Query Processor* processes and answers semantic-aware (or standard) queries issued by the user using *SemIndex* component.

A summary description of *SemIndex*'s architecture was given in [23]. This paper adds: i) an extended mathematical description of *SemIndex*'s logical design and dedicated graph model, ii) an extended description of *SemIndex*'s algorithms for index construction and query processing, iii) *SemIndex*'s physical design using an extension of SQL used within a standard commercial Relational DBMS (i.e., RDBMS), iv) detailed complexity analyses covering index construction and querying algorithms, v) extensive experimental results evaluating *SemIndex*'s build time, storage size and characteristics, query processing time, and quality of returned results in comparison with legacy inverted index, as well as vi) an extended discussion of the state of the art solutions.

The rest of this paper is organized as follows. Section 2 described input resources required to build *SemIndex*. Section 3 introduces *SemIndex*'s the logical design and data graph model, and develops the index construction process. Section 4 describes *SemIndex*'s physical design and implementation within a standard RDBMS. Section 5 presents our query model for designing and processing semantic-aware queries. Our algorithms' complexity analysis is provided in Section 6. Experimental results evaluating the different aspects of *SemIndex* construction and querying are presented in Section 7. Section 8 briefly reviews the related works in semantic full text search, with special emphasis on semantic indexing techniques, before concluding the paper with ongoing works and future directions in Section 9.

2. Input Resources

2.1. Textual Data Collection

In our study, a textual data collection can be a set of documents, or tuples in a relational or NoSQL database, as shown in Table 1. More formally:

Definition 1 - Textual Data Collection: A textual data collection Δ (i.e., *textual collection* for short) is represented as a relation defined over a set of attributes $A = \{A_1, \ldots, A_p\}$ where each A_j is associated with a set of values (such as strings, numbers, BLOB, etc.) called the domain of A_j and denoted by $dom(A_j)$. Thus, given a relation Δ defined over attributes A, each data object (record) $O_i \in \Delta$ having a unique identifier $id(O_i)$ is denoted as $O_i \langle a_1, \ldots, a_p \rangle$[1], where $a_j \in dom(A_j)$. Each a_j from O_i is denoted as $O_i.a_j$ •

Given a textual data collection Δ, an inverted index (also referred to as a posting file, or inverted list) built upon Δ, in its simplest form, is a sorted list of index terms and object identifiers from Δ, as shown in Fig. 2.a. More formally:

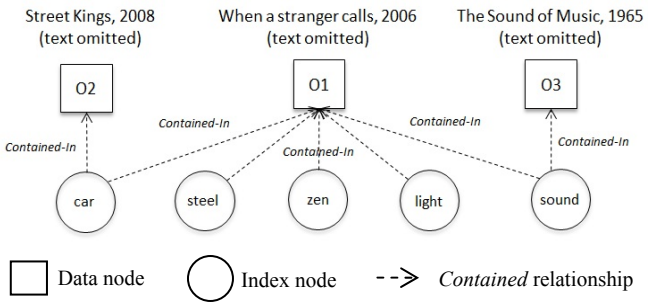
Definition 2 - Inverted Index: Given a textual data collection Δ, an inverted index built on Δ, designated as *InvIndex(*Δ), is a structure of the form ($dom(A)$, *IDs*, *f*) where:

- $dom(A)$ designates the set values within the domains of all attributes $A \in \Delta$. Considering text-only domains, values come down to textual tokens, i.e., *terms* (words/expressions).
- *IDs* designates the set of identifiers of the objects in Δ, i.e., $IDs = \{id(O_i)\} \; \forall \; O_i \in \Delta$
- *f* is a function mapping each $term \in dom(A)$ to a list of object identifiers *IDs*[] designating the term's occurrence locations in Δ, i.e., $IDs[] = \langle id(O_i) \rangle$ / $term \in$ any $O_i.a_j$

A *term* used as textual token in the inverted index is referred to as *index term*, whereas the list of data object identifiers, i.e., *IDs*[], mapping to each index term is referred to as the term's *posting list* •

Term	Object IDs[ ]
"car"	⟨O1, O2⟩
"light"	⟨O1⟩
"sound"	⟨O3⟩
"steel"	⟨O1⟩
"zen"	⟨O1⟩
…	…

a. Inverted index *InvIndex*(Δ).



b. *SemIndex* graph $\tilde{G}_\Delta$ representing *InvIndex*(Δ).

Fig. 2. Sample inverted index (a) and corresponding *SemIndex* graph (b), based on the textual collection Δ in Table 1.

Fig. 2 shows an extract from an inverted index built on the sample movie database in Table 1, where data objects O_1, O_2, and O_3 have been indexed using index terms extracted from the database, sorted in alphabetic order. It is important to note that this simple index is typically used to answer containment queries [99], aiming at finding data objects that contain one or more terms. When a keyword query mapping two or more index terms must be processed, the corresponding posting lists are read and merged. The index terms and their mappings with the data objects can be generated using classical Natural Language Processing (NLP) techniques (including

[1] We use symbols ⟨ and ⟩ to designate an *ordered* list of elements, and symbols { and } to designate an *unordered* set.

stemming, lemmatization, and stop-words removal) [65], which could be either embedded in the DBMS or supplied by a third-party provider.

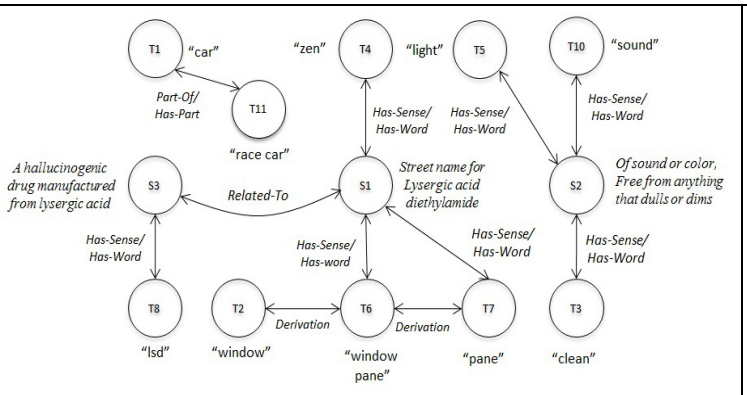
In its more elaborated form [8, 17], a posting list may also store along with each object identifier: the term frequency (*tf*), a list of positions where the given term appears (e.g., the element/attribute in which the term appears in semi-structured text, such as XML [71, 88]), and/or other features including whether the term is capitalized, is part of a title, is in the URL, etc. These extra data are kept for advanced functionality like phrase searching and result ranking, which we will address in an upcoming study.

2.2. Semantic Knowledge Base

In the Natural Language Processing (NLP) and Information Retrieval (IR) fields, semantic knowledge bases (i.e., ontologies, thesauri and/or taxonomies, such as WordNet [64], Roget's thesaurus [98], and Yago [42]) provide a framework for organizing words/expressions into a semantic space [18]. A knowledge base[1] usually can be represented as a semantic network made of a set of entities representing semantic concepts or groups of words/expressions, and a set of links between the entities, representing semantic relationships (*synonymy*, *hyponymy*, etc.). In this study, we adopt a structure based on graphs to model semantic knowledge bases. In such a structure, entities are represented as vertices, and the semantic relationships between entities are modeled as directed edges. Formally:

Definition 3 - Semantic knowledge base: A semantic knowledge base *KB* (i.e., *knowledge base* for short) can be represented as a semantic network graph, also known as knowledge graph, $G_{KB}(V, E, L, f_V, f_E)$ where:

- V is a set of vertices (nodes), denoting entities in the knowledge base. To illustrate this with WordNet for example, V includes both: i) *sense* nodes, representing semantic senses (*synsets*) with glosses, and ii) *term* nodes, representing literal words/expressions
- E is a set of directed edges, an edge consisting of an ordered pair of vertices in V.
- L is a set of edge labels designating semantic/lexical relationships. For WordNet, L includes:
  - Semantic relationships between concepts, e.g., *hyponymy*, *hypernymy*, *meronymy*, etc.
  - Semantic relationships between concepts and terms, namely *has-sense* and *has-term* (e.g., in Fig. 3, word *"Zen" has-sense* S_1, and S_1 *has-term "Zen"*)
  - Lexical relationships between terms, namely *derivation* (e.g., term *"Zen" derives* term "*Buddhist Zen*", and "*Buddhist Zen*" *is-derived-from* "*Zen*")
- f_V is a function defined on V, representing the string value of each node in V. For WordNet, string values include: i) glosses/definitions, when dealing with *sense* nodes, and ii) and literal words/expressions,
- f_E is a function defined on E, assigning a label from L to each edge in E. Multiple edges may exist between the same pair of vertices when dealing with *term* nodes, which makes G_{KB} a multi-graph •



a. Sample G_{KB} graph representing a *KB* extract from WordNet.

Term	Sense IDs[]
"acid"	⟨S1, S3⟩
"clean"	⟨S2⟩
"light"	⟨S2⟩
"lsd"	⟨S3⟩
"lysergic"	⟨S1, S3⟩
…	…

b. Extract of inverted index $InvIndex(G_{KB})$ connecting terms in G_{KB} with corresponding senses (to speed up term/synset lookup)

Fig. 3. Extract from the knowledge graph of WordNet, with corresponding inverted index.

An extract from the WordNet ontology is shown in Fig. 3, where S_1, S_2 and S_3 represent senses (i.e., synsets), and their string values (i.e., the synsets' glosses/definitions), and T_1, T_2, …, T_{11} represent terms, and their string values (i.e., literal words/expressions) shown alongside the nodes. Given that most semantic/lexical relationships are symmetrical (*hyponymy*/*hypernymy*, *meronymy*/*holonymy*, *has-sense/has-term*, etc.), and given that a relationship cannot exist without its symmetrical counterpart, we simplify our graph model by representing each couple of symmetrical relationships between senses and/or terms with one edge having

[1] In the remainder of the paper, we will use WordNet as the illustrative semantic knowledge base (cf. Fig. 3).

opposite directions (instead of two edges), labeled with the names of the symmetrical relationships. For instance, if one meaning of a term belongs to a synset, it is represented with one edge between the corresponding sense (synset) node and the term node with opposite directions, labeled *has-sense/has-term*.

An inverted index *InvIndex*(G_{KB}) can be subsequently built for the textual tokens of each G_{KB} entity (i.e., string values of *term* nodes and *sense* nodes, cf. Fig. 3.b) to speed up term/sense lookup when creating and then querying the integrated *SemIndex* structure (cf. Section 3).

3. *SemIndex* Logical Design

In this section, we introduce the logical design techniques of *SemIndex*. As mentioned previously, *SemIndex* adapts tight coupling techniques to index the textual data collection and the semantic knowledge base in one single index structure, creating a single set of posting lists for all searchable content in both input resources. In the following, we first present *SemIndex*'s graph model, and then describe its construction process.

3.1. *SemIndex* Graph Model

To combine the resources, we define *SemIndex* as an extended knowledge graph:

Definition 4 - *SemIndex* graph: Given an input textual collection Δ and an input knowledge base *KB*, we define *SemIndex*(Δ, *KB*) as an extended knowledge graph $\widetilde{G}_{SI}$ (V_i, V_d, E_i, E_d, L, f_V, f_E, f_W) where:

- V_i is a set of *index nodes*, denoting i) entities (*senses* and *terms*) from *KB*, and ii) *index terms* from Δ:
  - $V_i^+ \subseteq V_i$ designates the subset of *term* nodes designating *searchable terms*[1] in $\widetilde{G}_{SI}$, i.e., nodes referring to *terms* from *KB* and *index terms* from Δ (represented visually as *circle* nodes)
  - $V_i^\# \subseteq V_i$ designates the subset of *sense* nodes in $\widetilde{G}_{SI}$ referring to *senses* from *KB* (represented visually as *double circle* nodes)

  Naturally, $V_i = V_i^+ \cup V_i^\#$
- V_d is a set of *data nodes*, denoting data objects from Δ (represented visually as *square* shaped nodes[2])
- E_i is the set of edges between index nodes, called *index edges*, defined as ordered pairs of index nodes in V_i (represented visually as straight arrows)
- E_d is the set of edges linking index nodes with data nodes, called *data edges* (represented visually as dashed arrows)
- L is a set of edge labels including:
  - *Index edge* (E_i) labels which represent semantic/lexical relationships between index nodes (e.g., *hyponymy*, *meronymy*, *has-sense*, etc.)
  - A single *data edge* (E_d) label: *contained*, designating the containment relationship between term nodes in V_i^+ and data nodes in V_d
- f_V is a function defined on $V_i \cup V_d$, representing the string value of each node in $V_i \cup V_d$
- f_E is a function defined on $E_i \cup E_d$, assigning a label from L to each edge in $E_i \cup E_d$
- f_W is a weighting function defined on the nodes in $V_i \cup V_d$ and the edges in $E_i \cup E_d$. The weights will be used in selecting and ranking semantic-aware query results, described in Section 5 •

A sample *SemIndex* graph is shown in Fig. 6 (cf. Section 3.3), built based on the textual collection Δ from Table 1 (where $\widetilde{G}_\Delta$ is reported in Fig. 4.a) and the *KB* extract in Fig. 3 (where $\widetilde{G}_{KB}$ is provided in Fig. 4.b). It comprises 3 data nodes ($O_1 - O_3$), 3 index *sense* nodes ($S_1 - S_3$), and 11 index *term* nodes ($T_1 - T_{11}$) along with data and index edges. The *SemIndex* graph construction process is described in detail in the following subsections.

3.2. Indexing Input Resources

Building our *SemIndex* graph comes down to: i) generating two separate graph representations, for each of the input resources (textual collection and knowledge base) following our *SemIndex* graph model, and then ii) combining the resulting graphs into a single *SemIndex* graph structure.

Given an input textual collection Δ, we use a simple conversion function noted S*emIndex*(Δ) to produce a *SemIndex* graph representation of Δ designated $\widetilde{G}_\Delta$ = *SemIndex*(Δ). It comes down to first generating Δ's inverted index *InvIndex*(Δ) (cf. Definition 2 -), which is straightforwardly represented as a *SemIndex* graph (cf. Definition 4 -) $\widetilde{G}_\Delta$ where: i) the set of index nodes V_i represents *index terms* in Δ (*searchable term* nodes), i.e.,

---

[1] Searchable terms will be mapped against query terms when performing query processing (cf. Section 5).

[2] Data nodes will designate (potential) query search results (Section 5).

$V_i = V_i^+$ (since Δ does not contain *senses*, i.e., $V_i^{\#} = \phi$), ii) the set of data nodes V_d represent data objects in Δ, and iii) the set of edge labels L includes one single label: *contained*, underlining the containment relationship between index nodes in V_i and data nodes in V_d. The weighting function f_W assigns weights to data nodes and data edges in $\widetilde{G}_\Delta$ based on certain strategies (related to the importance/frequency/diversity of terms, cf. Section 3.3) within the textual collection. A sample $\widetilde{G}_\Delta$ graph representing our running example inverted index based on the textual collection in Table 1 is shown in Fig. 4.a.

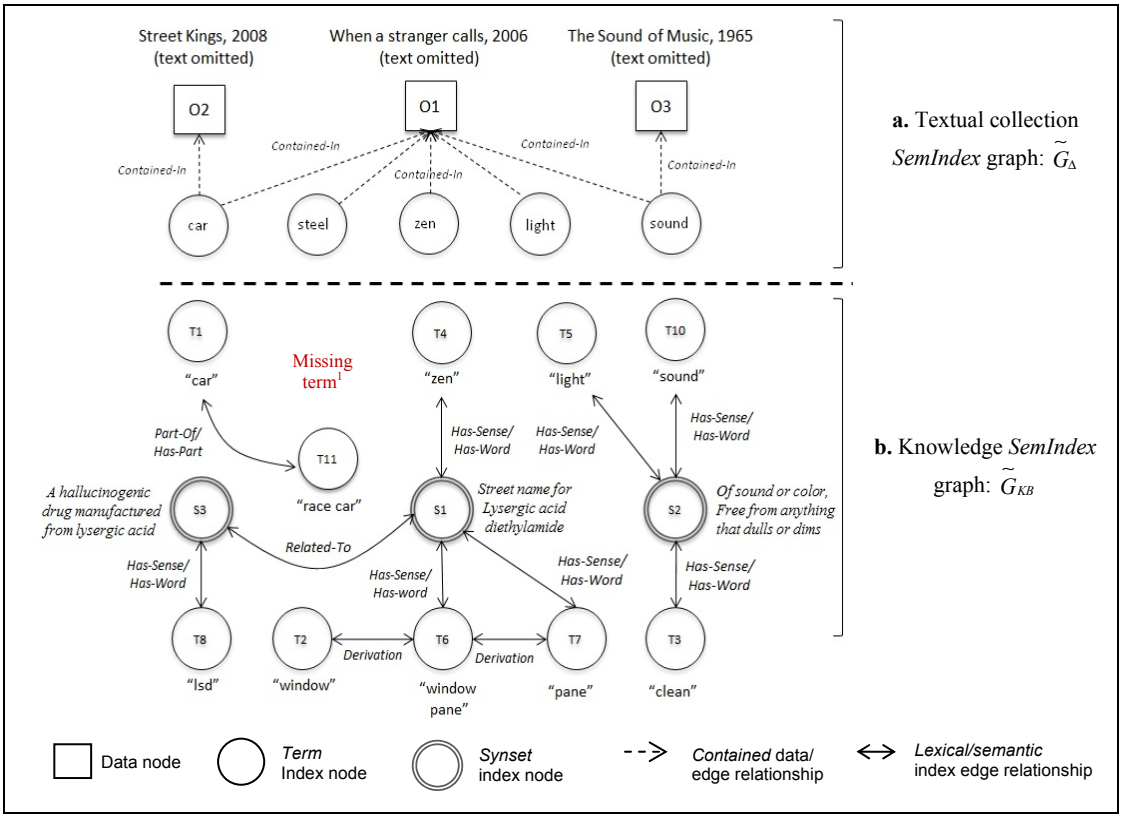


Fig. 4. *SemIndex* graph representations of input resources.

Similarly, given a semantic knowledge base *KB*, represented as a knowledge graph G_{KB}, we use a simple conversion function noted *SemIndex*(G_{KB}) to produce a *SemIndex* graph representation of G_{KB} designated: $\widetilde{G}_{KB}$= *SemIndex*(G_{KB}). G_{KB}'s inverted index *InvIndex*(G_{KB}) is generated and then straightforwardly represented as a *SemIndex* graph $\widetilde{G}_{KB}$ which inherits the properties of G_{KB}, in such a way that: i) the set of index nodes V_i represents all nodes in G_{KB}, and includes *term* nodes (V_i^+) and *sense* nodes ($V_i^{\#}$), ii) the set of data nodes V_d is empty (since *KB* does not contain data objects), and iii) the set of edge labels L includes all *index edge* labels designating semantic/lexical relationships in G_{KB} (e.g., *hyponymy*, *meronymy*, *has-sense*, *derivation*, etc.). The weighting function f_W assigns weights to index nodes and index edges in $\widetilde{G}_{KB}$ based on node/edge properties in the semantic graph (e.g., based on the type of the semantic/lexical relationship, cf. Section 3.3). A sample $\widetilde{G}_{KB}$ graph representing our running example knowledge base in Fig. 3.a is shown in Fig. 4.b.

3.3. Coupling Resources to Build *SemIndex*

Producing the combined *SemIndex* graph structure $\widetilde{G}_{SI}$ comes down to coupling both $\widetilde{G}_\Delta$ and $\widetilde{G}_{KB}$, noted as: $\widetilde{G}_{SI} = \widetilde{G}_\Delta \oplus \widetilde{G}_{KB}$, in such a way that: i) the set of index nodes $\widetilde{G}_{SI}.V_i = \widetilde{G}_\Delta.V_i \cup \widetilde{G}_{KB}.V_i$, including corresponding index edges from $\widetilde{G}_{KB}$ such that[2] $\widetilde{G}_{SI}.E_i \cong \widetilde{G}_{KB}.E_i$, ii) the set of data nodes $\widetilde{G}_{SI}.V_d = \widetilde{G}_\Delta.V_d$, including

[1] The *missing term* problem is discussed in Section 3.4.

[2] The set of index edges in $\widetilde{G}_N$ is not exactly equivalent to that in $\widetilde{G}_{KB}$ since it might contain additional index edges connected with *index terms* in $\widetilde{G}_\Delta$ which do not map to any *term* node in $\widetilde{G}_{KB}$. This is discussed as the *missing terms problem* in Step 4 of algorithm *SemIndex_Construction* (cf. Fig. 5).

corresponding data edges from $\widetilde{G}_{\Delta}$ such that $\widetilde{G}_{SI}.E_d = \widetilde{G}_{\Delta}.E_d$, and iii) the set of edge labels $\widetilde{G}_{SI}.L = \widetilde{G}_{\Delta}.L \cup \widetilde{G}_{KB}.L$ including all index node semantic/lexical relationships as well as the *contained* data edge label. The weighting function f_W compiles weights for all nodes and edges in the graph, as described in the following.

Algorithm *SemIndex_Construction*

Input: Δ // Textual data collection
KB // Semantic knowledge base
W // Weighting function parameters

Ouput: $\widetilde{G}_{SI}$ // *SemIndex* graph

Begin

Step 1: Build *InvIndex*(Δ) to construct $\widetilde{G}_{\Delta}$ 1

Step 2: Build *InvIndex*(G_{KB}) to construct $\widetilde{G}_{KB}$ 2

Step 3: Coupling $\widetilde{G}_{\Delta}$ and $\widetilde{G}_{KB}$ into $\widetilde{G}_{SI}$ by: 3

1. Mapping & Merging *searchable term* nodes in $\widetilde{G}_{\Delta}.V_i^+$ and $\widetilde{G}_{KB}.V_i^+$ 4
2. Including *sense* nodes from $\widetilde{G}_{KB}.V_i^{\#}$ 5
3. Including *data* nodes from $\widetilde{G}_{\Delta}.V_d$ 6

Step 4: Run *MissingTerms_Linkage* algorithm
// Connect *Missing terms* in $\widetilde{G}_{SI}$ 7

Step 5: Assign weights to edges & data nodes in $\widetilde{G}_{SI}$
- According to parameters *W* and weighting function f_W 8 9

Step 6: Aggregate edges between each pair of nodes in $\widetilde{G}_{SI}$ 10

Step 7: Remove from $\widetilde{G}_{SI}$: 11

1. Labels from all edges: $\widetilde{G}_{SI}.E$ 12
2. String values from all nodes except *searchable terms*: $\widetilde{G}_{SI}.V_i^+$ 13

Return $\widetilde{G}_{SI}$

End

a. Pseudocode of *SemIndex_Construction* algorithm.

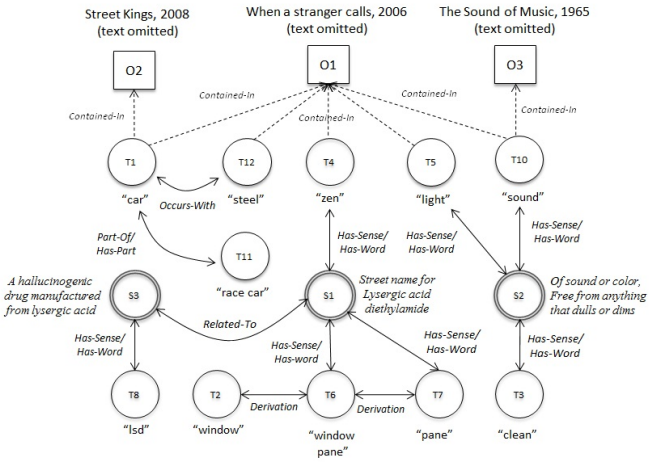
Fig. 5. Pseudocode of *SemIndex_Construction* algorithm.

The pseudo-code of the algorithm to construct $\widetilde{G}_{SI}$ consists of 7 main steps as shown in algorithm *SemIndex_Construction* in Fig. 5.a. Each step is detailed as follows:

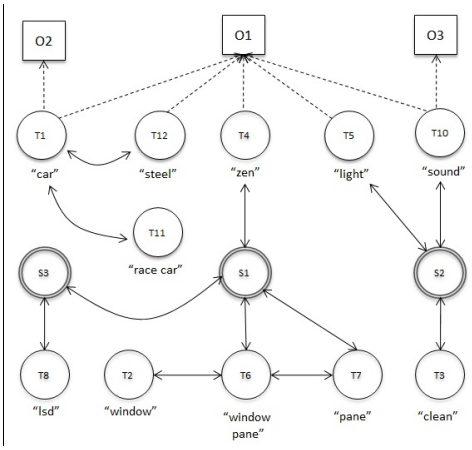
- **Step 1:** Given an input textual collection Δ, build the corresponding inverted index *InvIndex*(Δ), and generate the corresponding $\widetilde{G}_{\Delta}$ graph as previously defined.
- **Step 2:** Receiving a semantic knowledge graph G_{KB} representing the semantic knowledge base *KB* provided as input, build an inverted index *InvIndex*(G_{KB}) for the string values of each *KB* entity (i.e., *sense* nodes and *term* nodes, in order to access them more efficiently during resource coupling, and later during query execution), and then construct the corresponding $\widetilde{G}_{KB}$ graph as illustrated previously.
- **Step 3:** Combine the two *SemIndex* graphs into a single graph structure $\widetilde{G}_{SI}$. To do so, we map and then merge all *searchable term* nodes in $\widetilde{G}_{\Delta}$, i.e., $\widetilde{G}_{\Delta}.V_i^+$, with *searchable term* nodes in $\widetilde{G}_{KB}$, i.e., $\widetilde{G}_{KB}.V_i^+$, as follows:
  1. For each pair of *searchable term* nodes in $\widetilde{G}_{\Delta}.V_i^+$ and $\widetilde{G}_{KB}.V_i^+$, if their string values are equal, then remove one of them and merge all the connected edges.
  2. Sense nodes in $\widetilde{G}_{KB}$ are kept the same in $\widetilde{G}_{SI}$, i.e., $\widetilde{G}_{SI}.V_i^{\#} = \widetilde{G}_{KB}.V_i^{\#}$, but are connected with the corresponding *searchable term* nodes $\widetilde{G}_{SI}.V_i^+$
  3. Data nodes in $\widetilde{G}_{\Delta}$ are kept the same in $\widetilde{G}_{SI}$, i.e., $\widetilde{G}_{SI}.V_d = \widetilde{G}_{\Delta}.V_d$, but are connected with the corresponding *searchable term* nodes $\widetilde{G}_{SI}.V_i^+$ using the *contained* data edge relationship.

Fig. 6 shows the result of combining the two *SemIndex* graphs of the sample textual collection and the WordNet extract used in our running example.

– **Step 4:** *Searchable terms* from $\widetilde{G}_{\Delta}.V_i^+$ which do not map to any searchable term in $\widetilde{G}_{KB}.V_i^+$ can exist, which we identify as *missing terms* (e.g., term "*steel*" in Fig. 4). To solve the *missing terms* problem, we create links from each missing term to one or more closely related terms, connecting the missing and related terms using new index edges labeled *related-to*. The process is described in detail in Section 3.4.



a. *SemIndex* graph before removing edge labels and string values.



b. Final *SemIndex* graph representation.

Fig. 6. *SemIndex* graph $\widetilde{G}_{SI}$ obtained after coupling the data collection and the knowledge base graphs in Fig. 4.

– **Step 5:** Assign weights to edges and textual objects, according to f_W. The weights will be used to select and rank query results. Different weighting functions can be used, adopted from string indexing in IR [8, 50], similarity queries [57, 80], XML and graph-based processing [69, 89], and semantic processing [62, 84], which we briefly describe in the Appendix[1].

– **Step 6:** If an ordered pair of vertices is connected by two or more edges, it merges the edges and aggregates the weights. This means that $\widetilde{G}_{SI}$ becomes a graph, when generating *SemIndex*, rather than a multi-graph, in order to simplify query processing.

– **Step 7:** Finally, remove edge labels and string values of all nodes in $\widetilde{G}_{SI}$ except for V_i^+ (*searchable term* nodes), since all other nodes are not required for processing semantic queries. Removing node string values helps improve *SemIndex*'s scalability in terms of size, construction time, and query processing time (cf. experiments in Section 7).

Fig. 6 illustrates two instances of our running example $\widetilde{G}_{SI}$: including edge and node labels (preliminary version, cf. Fig. 6.a), and excluding edge and node labels except for *searchable term* nodes (final version, cf. Fig. 6.b). Edge and node weights were omitted for clearness.

3.4. Handling *Missing Terms*

Connecting unmapped *searchable term* nodes between $\widetilde{G}_{\Delta}.V_i^+$ and $\widetilde{G}_{KB}.V_i^+$, which we identify as *missing terms* in $\widetilde{G}_{SI}$, can be handled using an adaptation of distributional thesauri construction methods, e.g., [75, 94], to allow mining the syntactic/lexical relatedness between the *missing terms* and *index terms*. Note that a distributional thesaurus is a thesaurus generated automatically from a given textual corpus (such as the Brown corpus[2] [36], COCA [32], or even the textual collection Δ being indexed), by finding words that co-occur together or that have similar contexts in the corpus.

To that end, we introduce algorithm *MissingTerms_Linkage* in Fig. 7. It accepts as input: the *SemIndex* graph $\widetilde{G}_{SI}$, a reference text corpus *C*, as well as two input parameters: c_1 and c_2 designating respectively the co-occurrence *window size* and the number of *top-ranked terms* needed to identify related terms. For each missing term t_i in $\widetilde{G}_{SI}$ (cf. Fig. 7, line 1), the algorithm creates a *relatedness vector* $RV(t_i)$ (line 3) to store the co-occurrence frequencies of surrounding terms. It identifies a window of size c_1, consisting of c_1 terms occurring

[1] We report the detailed description and evaluation of the weighting scheme and its different variants to a dedicated study.

[2] We use the Brown text corpus in our current study since it is general purpose and widely known in the literature.

to the left and right of the missing term in the reference corpus and which also exist among the index terms of $\tilde{G}_{SI}$ (line 4), and adds all window term frequencies to the *relatedness vector* (line 5). For example, suppose "*steel*" is a missing term, i.e., it does not appear in the WordNet lexicon extract but appears in object O_1 of the data collection (cf. Fig. 4). Considering window size $c_1 = 2$[1], using the data collection itself Δ as reference corpus, then terms "*cop*", "*locate*", "*car*" and "*gang*" would be in the surrounding window of "*steel*", and hence the relatedness score between "*steel*" and all these terms is increased. Once the vector has been obtained, we normalize vector scores w.r.t.[2] overall maximum term co-occurrence frequency (line 6), and identify the c_2 top-ranked terms of the missing term t_i, which are considered as the most related terms to t_i in $\tilde{G}_{SI}$ (line 7). Then, a link is created to connect t_i's term node with each top-ranked term t_k node in $\tilde{G}_{SI}$. These links are represented as index edges in $\tilde{G}_{SI}.E_i$ labeled: *occurs-with* (cf. Fig. 6 where term "*steel*" links with "*car*", considered as its most related – top-ranked, i.e., highest co-occurrence frequency – term[3]).

Algorithm *MissingTerms_Linkage*

Input: $\tilde{G}_{SI}$ // *SemIndex* graph
C // Reference text corpus
c_1, c_2 // Input parameters: *window size* and *top-ranked terms*

Ouput: $\tilde{G}_{SI}$ // *SemIndex* graph with *missing term* links

```
Begin
For each missing term t_i in G̃_SI                                          1
{                                                                          2
    Create RV(t_i) from C given G̃_SI   // Relatedness vector for term t_i   3
    For each term t_j in window(t_i, c_1, C)                               4
    { Add Freq(t_j) to RV(t_i) }                                           5
    RV(t_i) = RV(t_i) / Max(RV(t_i))   // Normalizing RV(t_i) scores        6
    T_i = set of c_2 top-ranked terms in RV(t_i)                           7
    For each term t_k in T_i                                               8
    {  Create link between term nodes t_i and t_k in G̃_SI                  9
       Label the link "occurs-with" }                                      10
   }                                                                       11
   Return G̃_SI                                                            12
End
```

Fig. 7. Pseudocode of *MissingTerms_Linkage* algorithm.

The effectiveness of algorithm *MissingTerms_Linkage* depends on the number of missing terms, which in turn depends on the semantic coverage and expressiveness of the knowledge base used and its relatedness with the input textual collection (e.g., using a *medical* knowledge base to semantically map terms in a textual collection describing *sports events* will obviously lead to a substantial number of *missing terms* in the resulting *SemIndex* graph, thus negatively affecting index construction performance, cf. experiments in Section 7).

4. *SemIndex*'s Physical Design and Implementation in a Standard RDBMS

In this section, we show how to extend SQL in order to easily setup the graph of *SemIndex* on disk as a set of relational tables[4], and then formulate corresponding queries. The aim of building *SemIndex* on an off-the-shelf RDBMS, although it can be built directly on top of the file system, is to take advantage of the fact that RDBMSs are capable of efficiently storing and handling large volumes of data. This also allows us to benefit from other RDBMS features including concurrency control, as well as index and memory management on the database.

---

[1] A window size of 2 (or 3) is often utilized in the word context analysis and disambiguation literature [9, 91], and is considered to produce good results, compared with larger window sizes which include noisy terms thus lowering performance.

[2] with respect to

[3] A *missing term* can link with more than one (top-ranked) related terms, if more than one related terms were ranked with the same maximum co-occurrence frequency with the *missing term*.

[4] Note that *SemIndex* can be created using legacy SQL, without the use of our extended SQL specification commands, which we introduce to simplify the set-up of the index structure.

4.1. Extending SQL

To simplify creating *SemIndex*, we propose three specification commands[1], following the DDL (Data Definition Language) command style: WEIGHTING MODEL, KNOWLEDGE MODEL, and SEMANTIC INDEX.

4.1.1. Weighting Model

A *weighting model* allows to store and handle *SemIndex* edge and node weights (cf. Section 3.3) and can be defined using the following statement:

```
<Define weighting model statement>::=
[CREATE | ALTER] WEIGHTING MODEL <weighting name>
ON DATA EDGE
    [CONTAINMENT [<alg name>] [({<param list>})] [DEF <value>][,]]
ON INDEX EDGE
    [SYNONYMY [<alg name>] [({<param list>})] [DEF <value>][,]]
    [HYPONYMY [<alg name>] [({<param list>})] [DEF <value>][,]]
    [MERONYMY [<alg name>] [({<param list>})] [DEF <value>][,]]
    [HYPERNYMY [<alg name>] [({<param list>})] [DEF <value>][,]]
    [HOLONYMY [<alg name>] [({<param list>})] [DEF <value>][,]]
    [SENSE [<alg name>] [({<param list>})] [DEF <value>][,]]
    [SENSEINV [<alg name>] [({<param list>})] [DEF <value>][,]]
    [DERIVATION [<alg name>] [({<param list>})] [DEF <value>][,]]
    [OTHERS [<alg name>] [({<param list>})] [DEF <value>]]
ON INDEX|DATA NODE
    [<alg name>] [({<param list>})] [DEF <value>]
```

```
# Creating a sample weighting model:

CREATE WEIGHTING MODEL myweighting
ON DATA EDGE
    CONTAINMENT tfidf
ON INDEX EDGE
    SYNONYMY alg1(0.5),
    HYPONYMY alg1(1), HYPERNYMY alg1(1),
    MERONYMY alg1(2), HOLONYMY alg1(2),
    OTHERS alg1(2.5)
ON DATA NODE
    DEF 1
ON INDEX NODE
    DEF 1
```

This command creates and/or updates a weighting scheme called <weighting name>, based on the different weighting algorithms associated to data/index edges and nodes. Each algorithm indicated in the <alg name> clause must be individually developed and integrated into the *SemIndex* stored procedures. The parameters of each algorithm are optional and depend on the particular algorithm specified. All required parameters are included in the <param list> nested in the <alg name> clause. The optional <DEF value> allows to assign an edge or a node a given default parameter value. For instance, the above command (to the right) could be issued by a user to create a sample weighting scheme considering that: i) all movies are equally important, ii) synonymy is more important in weighting index edges than all other semantic relationships, and iii) hypernymy/hyponymy relations are more important in weighting index edges than holonymy/meronymy relations. Here, *tfidf* and *alg1* are two predefined algorithms to compute statistical and structural information related to data edges and index edges respectively, such that *alg1* takes different input parameters whose increasing values produce decreasing weight scores, and *def 1* assigns default weight value 1 to data and index nodes. An existing weighting model, named <weighting name>, can be dropped as follows:

```
<Drop weighting model statement>::= DROP [WEIGHTING MODEL] <weighting name>
```

4.1.2. Semantic Knowledge Model

A *semantic knowledge model* allows to store and handle the semantic knowledge base in the RDB, and can be defined using the following command statement. This command allows to create and/or update a knowledge base model called <knowledge name>, based on a set of relations or a given SQL script. For instance, in order to use WordNet as the reference knowledge base, the user can issue the following statement (to the right).

```
<Define knowledge model statement>::=
[CREATE | ALTER] KNOWLEDGE MODEL <knowledge name>
USING [{<relations list>} | <script filename>]
```

```
# Creating a sample knowledge model:
CREATE KNOWLEDGE MODEL wordnet
USING 'C:\WORDNET\script.sql'
```

Similarly, an existing semantic knowledge model, named <knowledge name>, can be dropped as follows:

```
<Drop knowledge model statement>::= DROP [KNOWLEDGE MODEL] <knowledge name>
```

4.1.3. Semantic Index Model

Similarly to the traditional SQL CREATE INDEX statement syntax, creating and/or altering *SemIndex* on one or several attributes can be issued using the following statement. The optional HASH/BTREE clause allows to create

[1] Syntax in is EBNF (Extended Backus-Naur Form) notation [46].

SemIndex nodes using hash-based or B-Tree indexing techniques to speed-up data look-up. The following example (to the right) shows how to create *SemIndex* based on our MOVIES relation, considering two of its attributes *title* and *plot* using the weighting scheme and WordNet semantic model defined above:

```
<Define SemIndex statement>::=
[CREATE | ALTER] [HASH | BTREE] SEMANTIC INDEX <index name>
ON <relation name> [({<att list>})]
[WHERE <predicate>]
USING
WEIGHTING <weighting model>,
KNOWLEDGE <knowledge name>
```

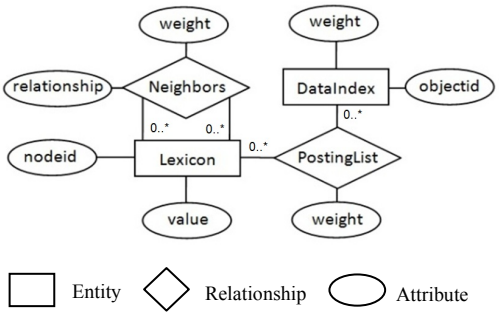
```
# Creating a sample semantic index:

CREATE SEMANTIC INDEX mysemindex
ON MOVIES (Title, Plot)
USING WEIGHTING myweighting,
KNOWLEDGE wordnet
```

In order to drop an existing *SemIndex* structure or rebuild it (after modifying its weighting scheme and/or knowledge base), the following statement can be used:

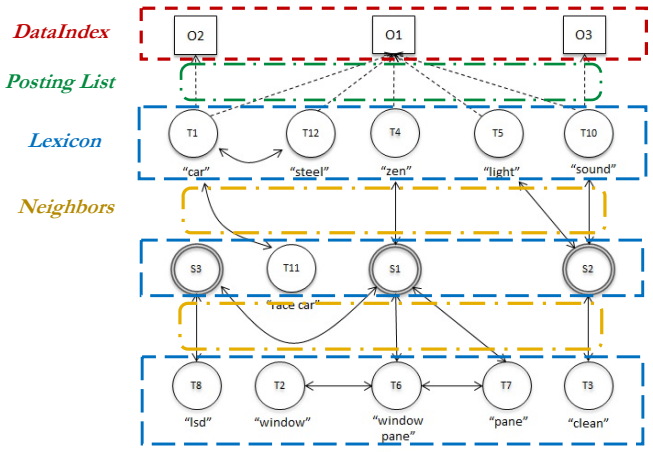
```
<Rebuilding SemIndex statement>::= [DROP | REBUILD] [SEMANTIC INDEX] <index name>
```

4.2. *SemIndex* Physical Design

Fig. 8.a shows the ER conceptual diagram of the *SemIndex* data graph, whereas Fig. 8.b depicts the data coverage of each relation in the resulting RDB schema. The relations are described separately in the following subsections.



a. Conceptual ER model describing *SemIndex* physical design.



b. Data representation of each relation in the resulting RDB schema.

Fig. 8. *SemIndex* physical design.

4.2.1. Data Index

The DDL[1] statements for creating relation DataIndex is shown below:

```
CREATE TABLE DataIndex(objectid INT PRIMARY KEY, weight DECIMAL);
```

Relation DataIndex stores (in attribute DataIndex.objectid) the identifiers of data objects from the data collection (Δ), represented as data nodes in *SemIndex*, i.e., $\tilde{G}_{SI}.V_d$ (cf. Fig. 8) along with data node weights (e.g., an *object rank* score, stored in attribute DataIndex.weight, which is computed and then updated during query processing, cf. Section 5). An extract of DataIndex's content, following our running example *SemIndex* graph (from Fig. 6) is shown in Fig. 9.a. Other information, such as the publication date or the original full text of a data object, may also be stored in this relation, depending on the output requirements and on the system environment.

4.2.2. Lexicon

The DDL statement for creating the Lexicon relation is shown below:

```
CREATE TABLE Lexicon (nodeid INT PRIMARY KEY, value VARCHAR, weight DECIMAL);
```

[1] Data Definition Language.

Relation Lexicon stores the lexicon of the knowledge base (*KB*) used to index the data collection (Δ), i.e. the set of all index nodes in *SemIndex*, i.e., $\widetilde{G}_{SI}.V_i$ (cf. Fig. 8). *Searchable term* nodes, i.e., $\widetilde{G}_{SI}.V_i^+$, are stored in their lemmatized form (in attribute Lexicon.value) along with corresponding node identifiers (e.g., WordNet node identifiers, stored in Lexicon.nodeid), whereas *sense* nodes, i.e., $\widetilde{G}_{SI}.V_i^{\#}$, have null values (in attribute Lexicon.value). Note that Lexicon also includes *missing terms* (as briefly described in Section 3.4) stored in their lemmatized form (in attribute Lexicon.value) along with special system generated node identifiers (different from WordNet's, stored in Lexicon.nodeid). Index node weights are then computed and dynamically updated during query processing, stored in attribute Lexicon.weight. An extract of Lexicon's content, following our running example *SemIndex* graph (from Fig. 6) is shown in Fig. 9.b.

In some commercial keyword search engines, the lexicon is generally kept in memory for fast response time, since its size is not related to the size of the indexed dataset and is generally much smaller than the term posting lists. In *SemIndex*, we adopt the same idea by allowing relation Lexicon to be kept in memory, when supported by the DBMS.

objectid	weight
O1	$w_{DataNode}(T_1)$
O2	$w_{DataNode}(T_2)$
O3	$w_{DataNode}(T_3)$

a. DataIndex

nodeid	value	weight
T1	"car"	$w_{IndeNode}(T_1)$
T2	"window"	$w_{IndeNode}(T_2)$
T3	"clear"	$w_{IndeNode}(T_3)$
T4	"zen"	$w_{IndeNode}(T_4)$
T5	"light"	$w_{IndeNode}(T_5)$
...	...	...

b. Lexicon

nodeid	objectid	weight
T1	O1	$w_{DataEdge}(e_{T_1}^{O_1})$
T1	O2	$w_{DataEdge}(e_{T_1}^{O_2})$
T4	O1	$w_{DataEdge}(e_{T_4}^{O_1})$
T5	O1	$w_{DataEdge}(e_{T_5}^{O_1})$
...	...	...

c. PostingList

id[1]	node1id	node2id	relationship[2]	weight
0	T1	T11	PartOf/ HasPart	$w_{IndexEdge}(e_{T_1}^{T_{11}})$
1	T2	T6	Derivation	$w_{IndexEdge}(e_{T_2}^{T_6})$
2	T3	S2	HasWord/ HasSense	$w_{IndexEdge}(e_{T_3}^{S_2})$
3	T4	S1	HasWord/ HasSense	$w_{IndexEdge}(e_{T_4}^{S_1})$
...	...	...	...	...

d. Neighbors

Fig. 9. Extracts of *SemIndex*'s RDB relations' contents, based on the running example *SemIndex* graph (cf. Fig. 6).

4.2.3. Posting List

The DDL statement for creating the PostingList relation is shown below:

```
CREATE TABLE PostingList(nodeid INT, objectid INT, weight DECIMAL, PRIMARY KEY(nodeid, objectid) );
```

Relation PostingList stores the inverted index of the textual collection (Δ), which comes down to *data edges* in *SemIndex*, i.e., $\widetilde{G}_{SI}.E_d$ (cf. Fig. 8). PostingList results from joining relations Lexicon with DataIndex, linking data nodes $\widetilde{G}_{SI}.N_d$ (PostingList.objectid) with corresponding searchable term nodes $\widetilde{G}_{SI}.V_i^+$ (PostingList.nodeid), each with its corresponding data edge weight (e.g., *term frequency* score). PostingList is clustered on attribute nodeid, and for each nodeid, the posting list is sorted on objectid to optimize search time. An extract of PostingList's content, following our running example *SemIndex* graph (from Fig. 6) is shown in Fig. 9.c.

4.2.4. Concepts/Terms Links

The DDL statement for creating the links between terms and concepts, in a Neighbors relation, is shown below:

```
CREATE TABLE Neighbors(id INT PRIMARY KEY, node1id INT, node2id INT, relationship VARCHAR, weight DECIMAL);
```

Relation Neighbors stores all *index edges* in *SemIndex*, i.e., $\widetilde{G}_{SI}.E_i$ (cf. Fig. 8) linking index nodes $\widetilde{G}_{SI}.N_i$ (stored as pairs of index node identifiers in attributes Neighbors.node1id and Neighbors.node2id), including: *term-to-*

[1] We include an artificial identifier since multiple index edges (i.e., multiple semantic relationships) may exist between the same pair of index nodes in our *SemIndex* graph, which comes down to a multi-graph.

[2] Relationships are only required in evaluating the weights of index edges (cf. Appendix), and can be removed after index edge weights have been computed (cf. Step 5 of *SemIndex_Contruction* algorithm in Fig. 5), since they are not needed in the query evaluation process.

term, *term-to-sense* and *sense-to-sense* relationships, along with index edge labels (stored in Neighbors.relationship) and corresponding index edge weights (stored in Neighbors.weight). When using WordNet, the label of the relationship includes 28 possible lexical/semantic relationship types (e.g., *hypernym*, *hyponym*, *meronym*, *related-to*, etc.), as well as the *has-sense*/*has-term* introduced to explore WordNet term nodes relations. An id attribute is added since several edges can exist between two index nodes. An extract of Neighbors' content, following our running example *SemIndex* graph (from Fig. 6) is shown in Fig. 9.d. Note that we design our query processor to follow each edge on its direction from node1id, thus relation Neighbors is clustered on node1id but not on node2id. Also note that relation Neighbors remains unused (un-accessed) when executing standard containment queries (i.e., semantic-free queries, as shown in the following section).

5. Query Processing with *SemIndex*

In this section, we define our query model and present a processing algorithm to perform semantic-aware search with the help of *SemIndex*.

5.1. Query Model

Definition 5 - Semantic-aware query: Given *SemIndex*(Δ, *KB*) and its graph representation $\widetilde{G}_{SI}$, we define a semantic-aware query as a *projection selection* query of the form $q = \pi_{Ai}\, \sigma_P\, l(\Delta)$, defined over data collection Δ, where $A_i \in A$ is a string-based attribute, $l \in \mathbb{N}$ represents a link distance threshold designating different levels of semantic awareness in query execution on $\widetilde{G}_{SI}$, and P is a selection predicate of the form $(A_i\ \theta\ s)$, where s is a user-given string value (e.g., a selection term/keyword), and $\theta \in \{=, \textit{like}\}$ whose evaluation against values in $dom(A_i)$ is previously defined •

Following the value of link distance threshold l, we consider four query types:

- **Standard Query**: When $l = 1$, the query is a standard containment query, involving only *data edges* (connecting *data nodes* with *searchable term* nodes using the *contained* relationship), such that no semantic information is involved.
- **Lexical Query**: When $l = 2$, the link distance threshold is increased by 1 to include (in addition to *data edges*), first level *index edges*. They designate lexical relationships between *searchable term* nodes (namely the *derivation* relationship, where one term *derives* another term), such that basic lexical information is involved.
- **Synonym-based Query**: When $l = 3$, the senses (synsets) are also involved. Here, link distance threshold covers the second level *index edges*: connecting *searchable term* nodes with corresponding *sense nodes* (via the *has-sense* and *has-term* semantic relationships), such that synonymous terms corresponding to the *sense nodes* are involved. Note that there is no direct edge between *data nodes* and *sense nodes.*
- **Extended Semantic Query**: When $l \geq 4$, the data graph of *SemIndex* can be explored in all possible ways, covering *index edges* designating all kinds of semantic relationships (*hyponymy*, *meronymy*, etc.) between *index nodes*, to reach even more semantically relevant results.

Regarding *SemIndex*'s physical design, relation *Neighbors* is completely disregarded when executing *standard containment queries* ($l = 1$) which are semantic-free. The *Neighbors* relation is required to execute the remaining semantic-aware queries ($l > 1$) in order to explore lexical/semantic relationships.

5.2. Query Answer

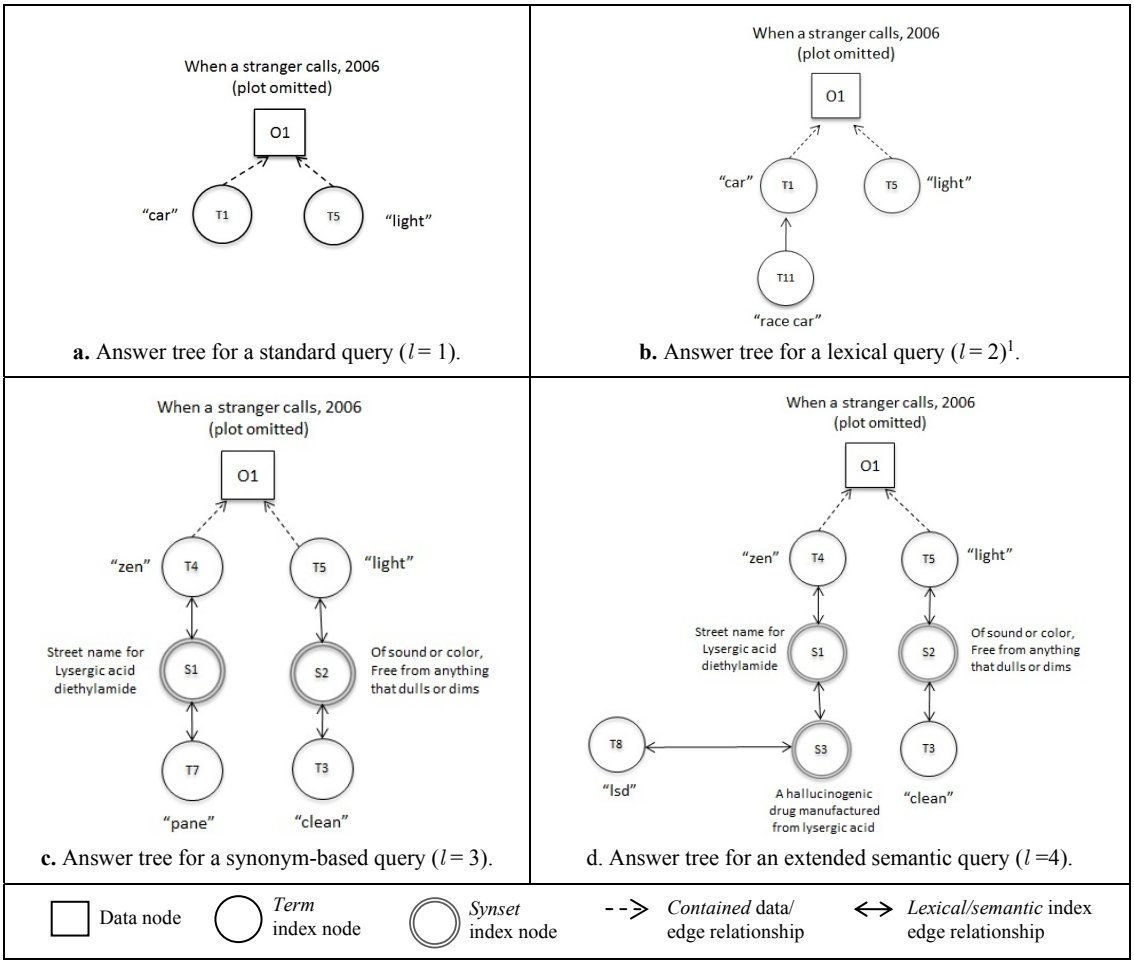
The answer to a query $q = \pi_X \sigma_P\, l\,(\Delta)$ in *SemIndex*(Δ, *KB*), noted $q(\Delta)$, is defined as follows.

Definition 6 - Query answer: Given *SemIndex*(Δ, *KB*) and its graph representation $\widetilde{G}_{SI}$, a query answer $q(\Delta)$ is the set of distinct root nodes of all answer trees in $\widetilde{G}_{SI}$, retrieving data objects in Δ. We define an answer tree as a connected sub-graph $T \subseteq \widetilde{G}_{SI}$ satisfying the following conditions:

- *Tree structure*: For each node $n \in T$, there exists exactly one directed path from n to T's root node $R(T)$,
- *Root node*: T's root is a *data node*, i.e., $R(T) \in \widetilde{G}_{SI}.N_d$, and it is the only data node in T, designating the corresponding textual object in Δ to be returned to the user,
- *Conjunctive selection*: When q consists of a multi-valued predicate P: $(A_i \in S)$, the *index node* matching every query term (keyword) in S occurs in the answer tree T,

- *Leaf nodes*: All leaf nodes in the answer tree T are *searchable term* nodes mapping to query terms (keywords). When q consists of a single-valued predicate P: (A_i θ s), the answer tree T would contain one single leaf node designating the *index node* matching s,
- *Height boundary*: The height T, i.e., the maximal number of edges between the root and a leaf node, is not greater than the link distance threshold l,
- *Minimal tree*: No node can be removed from T without violating some of the above conditions.

It can be proven that the maximal in-degree of all nodes in T is at most k, where k is the number of query terms (keywords). Hence, the answer tree comes down to a conjunction of paths starting at leaf nodes designating each a query term, and ending at a common root designating the textual data object to be returned as result •



a. Answer tree for a standard query ($l = 1$).

b. Answer tree for a lexical query ($l = 2$)[1].

c. Answer tree for a synonym-based query ($l = 3$).

d. Answer tree for an extended semantic query ($l = 4$).

Fig. 10. Sample answer query trees[2] with different link distance threshold values l, extracted from our running example *SemIndex* graph (Fig. 6).

According to the value of the link distance threshold l which serves as an interval radius in the *SemIndex* graph, various answer trees can be generated for a number of query types:

- **Standard Query**: When $l = 1$, the root of the answer tree is linked directly to all leaves, representing the fact that the result data object contains all query terms directly. A sample answer tree is shown in Fig. 10.a for query $q = \pi_A \sigma_{A \in (\text{“}car\text{”}, \text{“}light\text{”})}\, l = 1(\Delta)$ considering our running example data collection Δ (Table 1) and the corresponding *SemIndex*(Δ, *KB*) (Fig. 6),

[1] Node S_2 does appear in the answer tree since it's not a *searchable term node*: it is a *synset index node* (designating a *concept meaning*, and not a *textual term*). Recall that user queries start only from *searchable term nodes* (e.g., node T_{11} in Fig. 10.b), and navigate their way toward the closes data nodes within the query's link distance threshold l (e.g., O_1 is at distance $\ell \leq 2$ from T_{11}).

[2] While all edge and node labels are removed from the *SemIndex* graph except for *searchable term* nodes (cf. Section 3.3), we show *synset* node glosses here for the sake of presentation.

- **Lexical Query**: When $l = 2$, the answer tree includes lexical connections between query term nodes and other index term nodes. Fig. 10.b is an example answer tree for query $q = \pi_A \sigma_{A \in (\text{“race car”},\text{“light”})}\ l=2(\Delta)$,
- **Synonym-based Query**: When $l = 3$, the answer tree includes *sense nodes*, in addition to the two previous cases. Note that due to the *minimal tree* restriction (Definition 6 -), a sense node cannot be a leaf node of an answer tree. Thus, if an answer tree contains a sense node, the height of the tree is not less than 3. A sample answer tree is shown in Fig. 10.c for query $q = \pi_A \sigma_{A \in (\text{“pane”},\text{“clean”})}\ l=3(\Delta)$. The synonyms of the two query terms, *“zen”* and *“light”* are also contained in the answer tree rooted at the data node of object O_1,
- **Extended Semantic Query**: When $l = 4$, the answer tree contains additional index nodes connected via index edges designating different semantic relationships, according to the provided input selection terms. An example answer tree is shown in Fig. 10.d for query $q= \pi_A \sigma_{A \in (\text{“lsd”},\text{“clean”})}\ l=4(\Delta)$.

Note that it is possible to have more than one path from a query term node to a data node in the *SemIndex* graph (through different semantic links), which will naturally result in more than one answer tree.

5.3. Query Processing

The pseudo-code for our *SemIndex* query processing algorithm is shown in Fig. 11. It takes as input a *SemIndex* graph $\tilde{G}_{SI}$, a set of query selection terms (keywords) S, and a link distance threshold l, and produces as output the set of data nodes N_{d_Out} (the answer trees' root nodes) designating the data objects returned as the query answer. The overall process can be described as follows:

- **Step 1:** The algorithm starts by identifying in $\tilde{G}_{SI}$ the index (*searchable term*) nodes mapping to each query term (using function *getNodeID( )*, line 4). At the physical level, this is performed by invoking relation Lexicon (e.g., *SendSQL*(“SELECT nodeid FROM Lexicon WHERE value = s_i”),
- **Step 2:** Then, for each of the selected index nodes, it identifies the minimum distance paths at distance l, i.e., using *Dijkstra*'s shortest path algorithm (performed by function *findShortestPaths( )*, line 5). At the physical level, this is performed by invoking relation Neighbors (e.g., *SendSQL*(“SELECT node2id FROM Neighbors WHERE node1id = n_{i_In}”),
- **Step 3.** Of these shortest paths, the algorithm then identifies those which contain data edges linking to data nodes (using function *getDataNodeIDs()*, line 6), and then adds the resulting data nodes to the list of output data nodes N_{d_Out}. At the physical level, this is done by querying the PostingList relation with the index nodes returned from *findShortestPaths()* (e.g., *SendSQL*(”SELECT objectid FROM PostingList WHERE nodeid = SP.n_i”),
- **Step 4:** Consequently, we merge the resulting data nodes with the list of existing answer data nodes. At the physical level, this is done by computing node intersection using PostingList (e.g., *SendSQL*(”SELECT objectid FROM PostingList WHERE objectid = n_{d_si}”). Each answer node is then assigned a score by adding its distance from every query term index node (using *mergeAndRank( )*, line 7). The algorithm finally returns the list of answer data nodes ranked by order of path scores in ascending order.

Algorithm *SearchTerms*

Input: $\tilde{G}_{SI}$ // *SemIndex* graph
S // A set of query selection terms
l // A link distance value, designating query-type

Ouput: N_{d_Out} // A list of ranked data nodes from $\tilde{G}_{SI}$ designating query answers

```
Begin
   N_d_Out = ϕ                                                                          1
   For each term s_i ∈ S          // For each selection term                            2
   {                                                                                    3
      Step 1: n_i_In = getNodeID(s_i, G̃_SI)          // Identify index node            4
      Step 2: SP = findShortestPaths(n_i_In, l, G̃_SI)   // Identify shortest path within distance l from n_i_In   5
      Step 3: N_d_si = getDataNodeIDs(SP, G̃_SI)      // Identify the set data (root) nodes in each shortest path   6
      Step 4: N_d_Out = mergeAndRank(N_d_si , N_d_Out)                                  7
   }                                                                                    8
   Return N_d_Out                                                                       9
End
```

Fig. 11. Pseudo-code of algorithm *SearchTerms*.

Note that the scores of data nodes returned as query answers (i.e., answer tree root nodes) are computed/updated dynamically while executing function *findShortestPaths()* based on typical *Diskstra*-style shortest distance computations [30]. Basically, *findShortestPaths()* explores the *SemIndex* graph with *Dijkstra*'s

algorithm from multiple starting index nodes n_{i_In} (multiple query terms $s_i \in S$). For each visited node n_j, it stores its shortest distances from all starting nodes (query terms). The path score of an index node n_j to a starting node (query term) n_{i_In} is the sum of all weights on index edges along the path between n_{i_In} and n_j (cf. examples hereunder). Similarly, the path score of a data node n_d to a starting node n_{i_In} adds, to the sum of all index edge weights in the path, the weight of the data edge connecting n_d to the path. In other words, the shortest distances of n_i (n_d) from n_{i_In} are also the minimal path scores of n_i (n_d) to all query terms.

For example in Fig. 10.c, given query terms "pane" and "clean", the algorithm starts to expand from index nodes T_7 and T_3. The weight score of T_7 is initialized to be a vector of path scores <0, ∞>[1], since the shortest distance from T_7 to "pane" is 0, but the node is not reachable from "clean". Similarly, the weight score of $T3$ is initially <∞, 0>. The weights of all other index nodes are initialized to <∞, ∞>[13]. The minimal path scores are then updated when each edge is explored in the graph. For example, starting from T_7, the weight of index node S_1, which was initialized to <∞, ∞> becomes <1,∞> when the node is reached, considering unit (=1) edge weight scores[2]. Likewise, the weights of nodes T_4 and O_1 become <2, ∞> and <3, ∞> respectively when the nodes are reached from T_7, and so forth. On the other hand, starting from T_4, the weights of nodes S_2, T_5, and O_1 become <∞, 1>, <∞, 2>, <∞, 3> respectively.

Consequently, given that a data node n_d can be reached from multiple starting nodes N_{i_In} (i.e., multiple leafs in the answer tree), function *mergeAndRank()* computes the combined path score of a data node (i.e., answer tree root node) as the aggregate path score from each starting node (each answer tree leaf node). As for the aggregation function, various mathematical formulations for combining path scores can be used [4, 89], among which the *maximum*, *minimum*, *average* and *weighted sum* functions. Here, we utilize the *maximum* aggregation function to account for the maximum distance (i.e., minimum semantic relatedness) between the query answer root node and all tree leaf nodes:

$$\text{score}(n_d) = \max_{\forall\, n_{i\_In} \text{ having } s_i \in S} pathScore\ (n_{i\_In},\ n_d) \quad \textbf{(1)}$$

For instance, considering the example in Fig. 10.c, the vector path score of data node O_1 would be <3, 3>, and thus its combined path score becomes 3. Considering the example in Fig. 10.b, starting from query terms "race car" and "light", the vector path score of data node O_1 would be <2, 1> (assuming unit edge weights as in the previous example), and thus its combined path score becomes 2. A data node which is not reachable from all query term nodes will have at least one infinite path score (i.e., zero semantic relatedness), along one (or more) of its path score vector dimensions.

Note that while we currently focus on relaxing "strict" conjunctive querying by increasing link distances between query and data nodes, yet our query model and processing approach can also incorporate different kinds of "weak AND" operators such as *fuzzy* predicates [45, 100] (which we are currently investigating).

6. Complexity Analysis

Table 2 summarizes the list of parameters and symbols used to explain the time complexity of our algorithms for building *SemIndex* and executing semantic-aware queries.

6.1. Building *SemIndex*

6.1.1. Time Complexity

Building *SemIndex* using algorithm *SemIndex_Construction* (cf. Fig. 5) is done in quadratic time and simplifies to $O(N^2)$ since:

- Step 1: Building the inverted index, and consequently the *SemIndex* graph for the textual collection Δ, i.e., $\widetilde{G}_\Delta$, is of typical $O(|\Delta| \times |A| \times N_\Delta)$ complexity, which simplifies to $O(|\Delta| \times N_\Delta)$ since $|A|$ is usually limited,
- Step 2: Also, building the *SemIndex* graph for the knowledge base *KB*, $\widetilde{G}_{KB}$, is of $O(|KB| \times N_{KB})$,
- Step 3: Coupling both Δ and *KB*'s *SemIndex* graphs by mapping and merging *searchable term* nodes in both $\widetilde{G}_\Delta$ and $\widetilde{G}_{KB}$ can be performed in $O(N_\Delta + N_{KB})$ time, given that both underlying structures are sorted,
- Step 4: Connecting missing terms with the merged index, using algorithm *MissingTerms_Likage* (cf. Fig. 7) can be performed in worst case $O(N_{miss} \times N_{term})$. Note that building the distributional thesaurus (to identify term relativeness vectors, based on their co-occurrences in the reference corpus) is conducted offline prior to *SemIndex* building and thus does not affect its complexity.

---

[1] Instead of ∞, we could have an initial weight value computed based on a given weight scheme.

[2] Any other edge weight function can be considered here, as discussed in the Appendix.

- Step 5: The complexity of the weighting process varies according the weight functions used. It amounts to $O(1)$ when assigning equal weights, or can vary as follows:
  - Data edges: performing typical *term frequency* computations to assign data edge weights simplifies to $O((N_{term}) \times |\Delta|)$ time,
  - Data nodes: assigning an *object rank* score to compute data node weights simplifies to $O(|\Delta|)$, taking into account one or several data object features (e.g., source, format, date, etc.) each feature being processed in typical constant time.
  - Index edges: computing index edge weights can be done in $O(N_{KB}{}^2 \times |L|)$, which simplifies to $O(N_{KB}{}^2)$ since $|L|$ is usually small,
  - Index node weights are computed during query processing, and thus do not affect *SemIndex* construction time.
- Step 6: Edge aggregation between each pair of index nodes in the *SemIndex* graph can be performed in $O((N_{term} + N_{syn})^2/2)$ time, which is the time needed to go through all pairs of index nodes in *SemIndex*,
- Step 7: Removing edge labels and string values from non-searchable (i.e., sense) nodes in *SemIndex* can be executed in $O(N_{E} + N_{syn})$.

Hence, the overall complexity of our *SemIndex* building process is bounded by $O(N^2) > \sum_{i=1...7} Complexity(Step_i)$ since $N \geq param, \forall param \in$ complexity parameters.

It is to be noted that building the inverted indexes and *SemIndex* graph for each of the input resources (i.e., Steps 1 and 2 of the algorithm), can be handled using multi-threading.

Table 2. Set of complexity symbols and related descriptions.

Symbol	Parameter
$\|\Delta\|$	Cardinality, in number of objects (table rows), of the textual data collection Δ
$\|A\|$	Degree, number of attributes, of Δ
N_Δ	Number of *searchable terms* from Δ, which comes down to: $\|\tilde{G}_\Delta.V_i^+\|$
$\|KB\|$	Cardinality, number of entities (senses and terms), of the knowledge base
N_{KB}	Number of *searchable term* nodes from *KB*, which comes down to: $\|\tilde{G}_{KB}.V_i^+\|$
$\|L\|$	Number of distinct lexical/semantic relationships in the knowledge base
N_{miss}	Number of *missing terms*: those *searchable terms* from Δ's *SemIndex* graph $\tilde{G}_\Delta$ which are not connected to those from $\tilde{G}_{KB}$.
N_{term}	Number of *term* index nodes in the *SemIndex* graph: $\|\tilde{G}_{SI}.V_i^+\|$ $(= \|\tilde{G}_\Delta.V_i^+ \cup \tilde{G}_{KB}.V_i^+\|)$
N_{syn}	Number of *sense* index nodes in the *SemIndex* graph: $\|\tilde{G}_{SI}.V_i^\#\|$ $(= \|\tilde{G}_{KB}.V_i^\#\|)$
N	Number of index and data nodes in the *SemIndex* graph: $\|\tilde{G}_{SI}.V_i\| + \|\tilde{G}_{SI}.V_d\|$
N_E	Number of index and data edges in the *SemIndex* graph: $\|\tilde{G}_{SI}.E_i\| + \|\tilde{G}_{SI}.E_d\|$
***Query-related symbols*:**	
k	Number of terms (keywords) in a query
N_{term_hom}	Number of homonymous terms in *SemIndex*, for a given query term,
l	Link distance threshold in a query
N_{i_acc}	Number of accessed index nodes in *SemIndex*, when executing a query
N_{d_acc}	Number of accessed data nodes in *SemIndex*, when executing a query
N_{E_acc}	Number of accessed edges in *SemIndex*, when executing a query

6.1.2. Space Complexity

As for space complexity, our approach requires space to store the final *SemIndex* graph $\tilde{G}_{SI}$, which is also bounded by $O(N^2)$ space. In fact, *SemIndex* is physically broken down into a set of 4 relations in a RDB schema (cf. Section 4) such that:

- *DataIndex*: stores data nodes, and thus requires $O(|\Delta|)$ space,
- *PostingList*: stores data edges connecting data nodes with *searchable term* nodes, and thus requires in the worst case $O(|\Delta| + N_\Delta)$ space,
- *Lexicon*: stores index nodes, and thus requires $O(N_{term} + N_{syn})$ space,

– *Neighbors*: stores index edges connecting pairs of index nodes, and thus requires $O((N_{term} + N_{syn})^2/2)$ space (recall that only one edge exists between two nodes in *SemIndex*).

Note that these relations, whose total size is bounded by $O(N^2)$, can be stored on disk or in memory according to the size of the input textual collection and knowledge base used.

6.2. Executing Queries

The complexity of our *SearchTerms* algorithm (cf. Fig. 11) which performs query execution on *SemIndex*, comes down to $O(N^2)$. In fact, the complexity of *SearchTerms* comes down to the sum of the complexities of its underlying functions, such that for each query term:

- *getNodeID( )* identifies the IDs of *term* nodes in the *Lexicon* corresponding to the query term, and thus requires in the worst case $O(N_{term} + N_{syn})$ time,
- *findShortestPaths( )* identifies the minimum paths at distance l from each of the starting *term* node, which comes down to running *Dijkstra*'s algorithm within distance l from the starting node. Given that *Dijkstra*'s algorithm requires $O(N_{E\_acc} \times l) = O(N_{i\_acc}^2 \times l)$ when applied from one starting node, it would require $O(N_{i\_acc}^2 \times l \times N_{term\_hom})$ when applied on N_{term_hom} starting nodes, given that one single query term could map to multiple starting *term* nodes (i.e., homonyms),
- *getDataNodeIDs( )* identifies the IDs of data nodes in *PostingList* for the each shortest path, and thus requires worst case $O(|\Delta| + N_{\Delta})$ time,
- *mergeAndRank( )* merges and ranks data nodes with existing query answer nodes, by comparing the latter with node IDs in the *PostingList*, thus requiring at most $O(|\Delta| \times N_{d\_acc})$.

Hence, the *SearchTerms* algorithm's complexity comes down to that of function *findShortestPaths( )* which requires $O(N_{i\_acc}^2 \times l \times N_{term\_hom})$, which is bounded by $O(N^2)$ time in the worst case scenario (covering the whole *SemIndex* graph).

When considering multiple query terms k, the complexity comes down to $O(N^2 \times k)$. Yet, given that k is usually limited (e.g., keyword queries on the Web are usually 2-3 words long [48, 82]), thus overall complexity simplifies to $O(N^2)$.

7. Experimental Evaluation

We first start by describing our prototype and experimental scenario, and then we present, compare, and assess empirical results.

7.1. Prototype

To validate our approach, we have implemented our *SemIndex* framework using Java. We also have used MySQL 5.6 as an RDBMS, and WordNet 3.0 as a knowledge base. In addition to the two basic *SemIndex* components (cf. architecture in Fig. 1) consisting of: i) the *indexer* (including our *SemIndex_Construction* and *MissingTerms_Linkage* algorithms, cf. Fig. 7), and ii) the *query processor* (including our *SearchTerms* algorithm, cf. Fig. 11), our implementation also includes: iii) a *lemmatizer*[1] used to transform index terms into their lemmas, as well as iv) extensible *weight computation* components which are called upon within the indexer and/or query processor to compute edge/node weights as needed (recall that weights are computed initially during indexing cf. Section 3.3, and are then updated dynamically during querying, cf. Section 5).

The RDBMS initially holds the input textual data collection and the knowledge base in the form of native RDBs[2]. Java is used to send SQL queries to the RDBMS in the following order required to build *SemIndex*:

1. Import the predefined *SemIndex* RDB schema (cf. Fig. 8).
2. Build the *Lexicon* table by importing the *words* table from WordNet.
3. Build the *Neighbors* table by importing the *senses* (term-to-synset), *semlinks* (synset-to-synset), and *lexlinks* (term-to-term) tables from WordNet. This is followed by computing index edge weights, and initializing index node weights.
4. Build the *DataIndex* table by processing all rows from the input textual database. Every row is tokenized and every token is lemmatized and inserted into *DataIndex*, along with corresponding data node weights.

---

[1] We used the University of Washington's *Morpha* lemmatizer available on the university's projects page: http://mvnrepository.com/artifact/edu.washington.cs.knowitall/morpha-stemmer/1.0.5

[2] WordNet 3.0's RDB is available from http://wordnet.princeton.edu/wordnet/download/

5. Identify all missing terms in *Lexicon*, by finding all terms in *DataIndex* that are not in *Lexicon* (using here a left join), and then include the latter in *Lexicon* (following Step 4 of our *SemIndex_Construction* algorithm).
6. Build the *PostingList* table by joining *Lexicon* with *DataIndex*. This is followed by computing term frequency weights of data edges.

The RDBMS will finally hold *SemIndex*'s RDB representation which will be processed for querying. Note that during the *SemIndex* building phase, Java is primarily used to lemmatize tokens and to create the textual database's inverted index. Yet, during *SemIndex* querying, Java is mainly used to run *Dijkstra*'s shortest path algorithm on every query term, and consequently find the intersection between the returned paths (as described in our *SearchTerms* algorithm). The usage of Java in our implementation is not mandatory and can be replaced by stored-procedures and triggers when supported by the DBMS. The *SemIndex* prototype is available online[1].

7.2. Experimental Scenario and Test Data

We evaluated the practical usability of our indexing approach by assessing four main criteria: i) index building time, ii) index size and characteristics, iii) query processing time, and iv) the number and quality of returned results, comparing in each experiment our *SemIndex* with the legacy *Inverted Index* solution. To do so, we started by varying the size of the input textual collection Δ by generating different extracts with respect to (w.r.t.) its total size (considering 10%, 20%, …, or 100% of Δ). We also vary the size of the input knowledge base by generating different extracts w.r.t. its total size (considering 10%, 20%, …, or 100% of *KB*). Then, for each doublet <Δ chunk ; *KB* chunk>, we evaluated each of the above four criteria by varying related parameters. User feedback tests are not detailed here.

We used the IMBD *movies* table[2] as an average-scale[3] input textual collection, including the attributes *movie_id* and (*title*, *plot*) concatenated in one column (cf. Table 1) with a total size of around 75 MBytes consisting of more than 140k rows and including more than 7 million terms. WordNet 3.0 had a total size of around 26 Mbytes, including more than 117k synsets (senses). The characteristics of the IMBD and WordNet chunks used in our experiments are summarized in Table 3 and Table 4.

Table 3. Characteristics of IMDB' *movies* table chunks.

Chunk %	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
Size (in MBs)	7.6293	15.0629	22.5192	30.0098	37.5452	44.9528	52.5214	59.9325	67.4239	74.8902
N# of Rows	14,304	28,608	42,912	57,217	71,521	85,825	100,130	114,434	128,738	143,043
N# of Terms	724,294	1,422,158	2,125,498	2,834,189	3,547,900	4,247,061	4,959,681	5,661,835	6,378,205	7,086,079
Size (in MBs) **of *InvIndex***	25.5781	49.625	73.6719	98.7188	122.7656	147.8125	171.8594	195.8906	220.9375	244.9844

Table 4. Characteristics of WordNet chunks.

Chunk %	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
Size (in MBs)	2.7707	3.9466	7.6498	9.5691	12.1641	13.8941	18.2191	19.9491	23.4091	26.0041
N# of Senses (Synsets)	11,738	23,475	35,212	46,949	58,686	70,423	82,160	93,897	105,634	117,371
Avg. Branch	1.4533	1.6257	1.7553	1.9236	2.0697	2.2259	2.3736	2.5285	2.6677	2.8223
Avg. Span	2.1035	2.2299	2.3665	2.5213	2.8849	3.5362	3.7411	4.1947	5.9852	7.5119
Size (in MBs) **of *InvIndex***	3.2031	4.5625	8.8437	11.0625	14.0625	16.0625	21.0625	23.0625	27.0625	30.0625

Table 3 provides the IMDB *movies*' table chunk size percentage and actual size (in MBytes), the number of rows and number of terms (e.g., textual tokens) per chunk, as well as the size of the resulting inverted index (i.e., *InvIndex*(Δ)). Table 4 provides the WordNet chunk size percentage and actual size (in MBytes), the number of senses (i.e., synsets) per chunk, the average branch factor[4] and average span factor[5] per chunk, as well as the size of the resulting inverted index (i.e., *InvIndex*(G_{KB})). Note that chunking the IMDB *movies* table was performed w.r.t. to the number of rows in the table, whereas chunking WordNet was performed w.r.t. the

---

1 Available at: http://sigappfr.acm.org/Projects/SemIndex/

2 Internet Movie DataBase raw files are available from online http://www.imdb.com/. We used a dedicated data extraction tool (at http://imdbpy.sourceforge.net/) to transform IMDB files into a RDB.

3 Tests using large-scale TREC data collections and the Yago ontology as a reference KB are underway within an dedicated comparative evaluation study.

4 The branch factor designates the number of outgoing edges per synset node, i.e., synset node fan-out [64].

5 The span factor designates the length of the path from a root (most abstract) synset node to a leaf (most specific) synset node in the WordNet taxonomy (considering hierarchical relations only, e.g., hypernym/hyponym and meronym/holonym, to avoid loops).

number of senses (synsets), which was more coherent in generating WordNet extracts than using the number of rows in WordNet's RDB representation (made of multiple joined tables representing different entities), with slight variations due to varying synset gloss sizes, varying number of synonyms per synset, and varying number of neighboring nodes (branch factor) per synset.

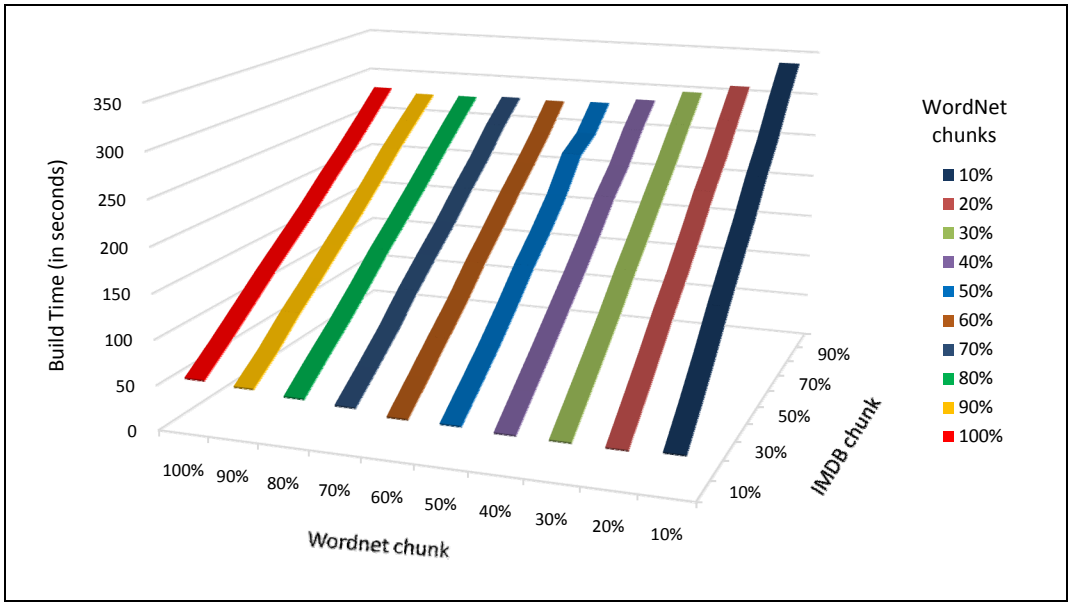


Fig. 12. Total *SemIndex* build time variation w.r.t. input IMBD and WordNet chunk sizes.

Tests were carried out on a PC with an *Intel I7* system with 2.9 GHz CPU, 8GB RAM memory, and a 500 GB built-in NTFS disk drive. The database (IMDB), knowledge graph (WordNet), and index files were stored on the disk drive's main partition.

7.3. Index Building Time

7.3.1. SemIndex Build Time

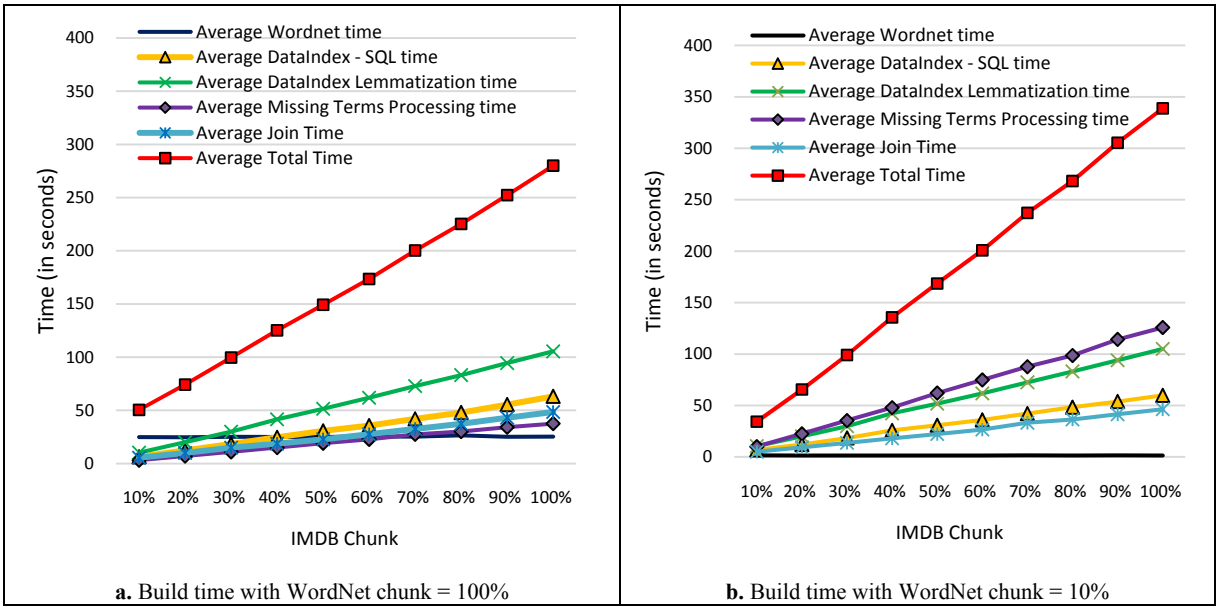
The 3D chart in Fig. 12 shows the total time required to build *SemIndex* while varying both IMBD and WordNet chunks. *SemIndex* construction tests were performed 5 times each, retaining average processing time. One can realize that the building time is linear in the size of the IMDB chunks on one hand (*x* axis), and linear on the size of the WordNet chunks on the other hand (*y* axis), which underlines quadratic time dependency w.r.t. both of them (which complies with our complexity analysis in Section 6.1).

We also note two additional observations. First, one can see that time variation w.r.t. IMDB chunk size (along the *x* axis) is greater than the variation w.r.t. WordNet chunk size (along the *y* axis). This is due to: i) the sheer size of IMDB chunks which are at least twice as big as their WordNet counterparts (and thus require at least twice as much processing time), and ii) due to running the time expensive lemmatization process on the database chunks, which is not required with WordNet chunks, thus inducing additional processing time. A breakdown of the tasks required to build *SemIndex* in Fig. 13.a and b shows the significant impact of lemmatization on the overall building time: the time to lemmatize index terms from IMDB (and term nodes from WordNet, when needed[1]) in order to be stored as *searchable terms* in *SemIndex* amounts to almost $1/3^{rd}$ of the total building time of *SemIndex*.

Second, one can also realize that while *SemIndex* building time slightly increases w.r.t. WordNet chunks varying from 50% to 100%, yet it also increases (rather than decreasing) with WordNet chunks varying from 50% to 10%. While the latter observation might seem counterintuitive (since we would expect build time to decrease when WordNet chunk size decreases), nonetheless the reason for the time increase is also inherent: the smaller the WordNet chunk, the higher the number of *missing terms*, and thus the more time is required to process them (mapping and linking them to WordNet terms). This is also shown in Fig. 13 where the time needed to process *missing terms* jumps from $1/10^{th}$ of the total building time, with WordNet chunk = 100% (i.e., when using the whole of WordNet in Fig. 13.a), to almost $1/3^{rd}$ of total time, with WordNet chunk = 10% (i.e., when using only a small portion of WordNet, Fig. 13.b).

---

[1] Most terms nodes in WordNet are handled in their lemmatized form, and need not be processed for lemmatization.



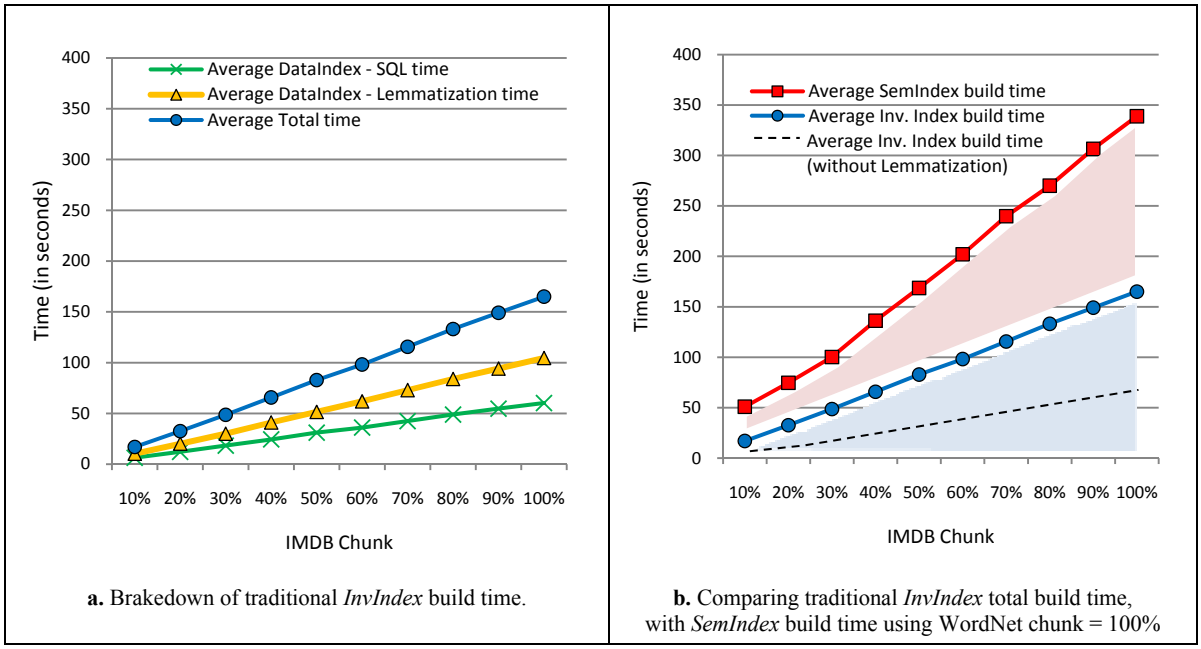
a. Build time with WordNet chunk = 100%

b. Build time with WordNet chunk = 10%

Fig. 13. Breakdown of *SemIndex* build timing.

Fig. 13.a and b also show that the building time of the WordNet part of *SemIndex* remains almost constant regardless of the IMDB chunk size. This is justified since building the *SemIndex* representation of the knowledge base is performed independently of the data collection. Furthermore, for a given knowledge base (like WordNet), the *SemIndex* representation can be produced once, stored in memory, and then made available for coupling with any new data collected to be indexed. Note that updating *SemIndex* incrementally or partially will be explored in a dedicated future study.

7.3.2. Comparison with Legacy Inverted Index Build Time



a. Brakedown of traditional *InvIndex* build time.

b. Comparing traditional *InvIndex* total build time, with *SemIndex* build time using WordNet chunk = 100%

Fig. 14. Comparison with traditional *InvIndex* build time.

To put things into perspective, we have also measured the total time required to build the legacy inverted index (which we note *InvIndex*) while varying IMDB chunk size[1] (cf. Fig. 14.a) and compared results with *SemIndex*'s

[1] Recall that *InvIndex* does not incorporate semantic knowledge and thus is not affected by WordNet chunk size variations.

(cf. Fig. 14.b). While both indices require linear building time, yet *SemIndex* requires almost twice (×2) as much build time as *InvIndex*. Furthermore, by disregarding the lemmatization phase in building *InvIndex* (which can be ignored following the database manager's preference: storing words in their actual rather than their original form), then *SemIndex* build time becomes almost four times (×4) greater than that of *InvIndex*. This is encouraging since even the fastest legacy inverted index creation time is only (at best) four times lesser than the creation time of *SemIndex*. The reasons for this are: i) the lightweight physical design of *SemIndex* which can be easily created using fast legacy database technology, as well as ii) the sheer difference in size between the textual database (IMBD) and the reference knowledge graph (WordNet), which renders the build time of *SemIndex* mostly dependent on IMDB size rather than WordNet size.

Regardless of the above, note that the index building process is done offline, prior (in preparation) to the system usage (query evaluation process), and thus does not affect (online) query execution time.

7.4. Index Size and Characteristics

7.4.1. SemIndex Size and Characteristics

Regarding *SemIndex* size, Fig. 15 shows that the *SemIndex* graph size (which, at the physical layer, comes down to the total size of all *SemIndex* relations following the adopted RDB schema), varies linearly with the size of the IMDB chunks (*x* axis) and WordNet chunks (*y* axis), which underlines quadratic size dependency w.r.t. both of them (conforming with our complexity analysis in Section 6.1.2). The detailed characteristics of *SemIndex* chunks are shown in Table 5 and Fig. 16, where each chunk is generated by merging the corresponding <Δ chunk ; *KB* chunk> doublet (for instance, the 10% *SemIndex* chunk is generated by merging the 10% Δ chunk with the 10% *KB* chunk, and so forth, cf. Δ and *KB* chunk characteristics in Section 7.2).

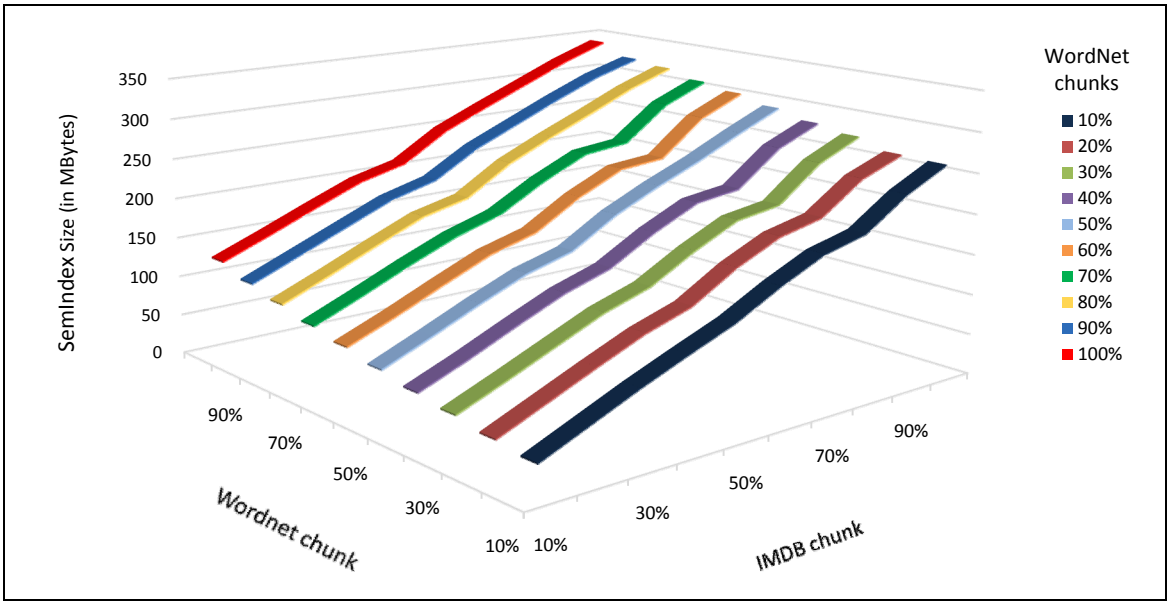


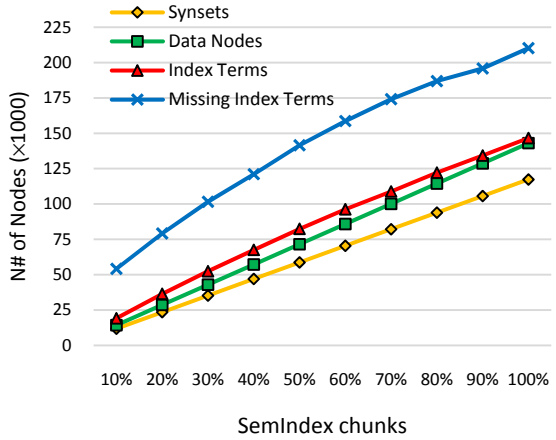
Fig. 15. *SemIndex* size variation.

Table 5 provides *SemIndex*' chunk size percentage and actual size (in MBytes); the total number of nodes in the *SemIndex* graph ($\widetilde{G}_{SI}$) including: data nodes (N_d), index nodes (V_i^+, including those corresponding to *missing terms*), and sense (synset) nodes ($V_i^\#$); the average branch factor: including and excluding data nodes (which represent leaf nodes in $\widetilde{G}_{SI}$), as well as the average span factor. *SemIndex* characteristics are also visualized in Fig. 16. Three main observations can be made.

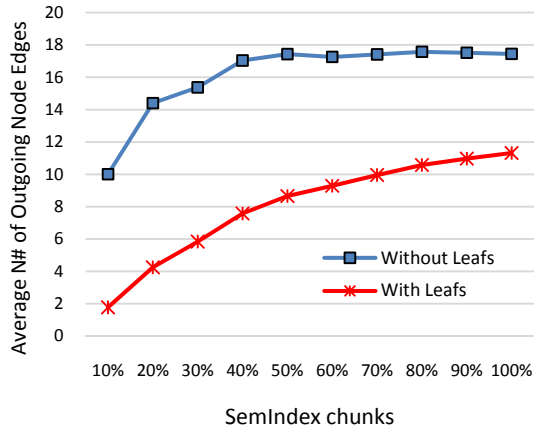
First, while the number of nodes in the *SemIndex* graph increases almost linearly w.r.t. *SemIndex* (and thus IMDB and WordNet) chunks size (cf. Fig. 16.a), one can realize that the number of index nodes resulting from *missing terms* is almost twice that of matching index terms. That is due to the fact that the IMDB *movies* table includes many textual tokens which are not part of the general purpose English language and thus do not appear in WordNet (e.g., terms like "advogado", "advon", "adyeri", "aeer", "moustafa", etc.). Note that we are currently investigating ways to alleviate the *missing terms* problem, using dedicated language processors and multilingual dictionaries, which will be covered in an upcoming study.

Table 5. Characteristics of *SemIndex* chunks.

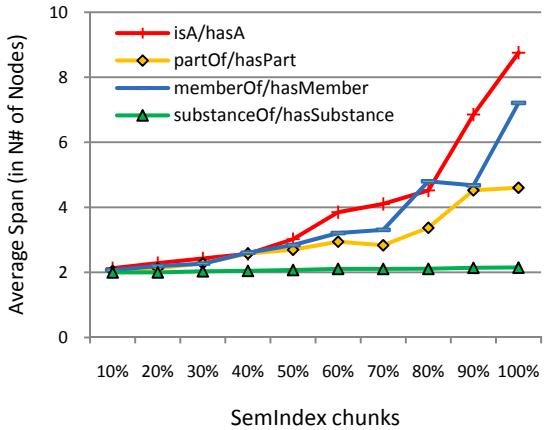
Chunk %	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
Size (in MBs)	36.9219	68.2188	100.2813	133.3281	158.3594	202.4063	237.4688	273.5156	306.5938	339.625
N# of Data Nodes	14304	28608	42912	57217	71521	85825	100130	114434	128738	143043
N# of (Matching) **Index Term Nodes**	19090	36396	52388	67511	82370	96231	108828	122119	134258	146625
N# of (Missing) **Index Terms Nodes**	54165	79174	101594	121078	141534	158663	174111	186930	195897	210279
N# of Sense Nodes (Synsets)	11738	23475	35212	46949	58686	70423	82160	93897	105634	117371
Total N# of Nodes	99297	167653	232106	292755	354111	411142	465229	517380	564527	617318
Avg. Branch (without data nodes)	10.0087	14.4015	15.3758	17.0408	17.4348	17.2604	17.4159	17.5771	17.5184	17.4495
Avg. Branch (with data nodes)	1.7746	4.2493	5.8399	7.5746	8.6564	9.2882	9.9577	10.575	10.9745	11.3173
Avg. Span	2.3886	2.5189	2.7408	3.0089	3.2661	3.6299	4.0109	4.2848	5.9962	7.6178



a. Distribution of the number of nodes in the *SemIndex* graph.



b. Average node branching factor, with and without data (leaf) nodes.data (leaf) nodes.



c. Average spanning factor considering hierarchical relations in *SemIndex*.

Fig. 16. Characteristics of *SemIndex* chunks.

A second observation concerns *SemIndex*'s branching factor, i.e., the average number of outgoing edges per node (cf. Fig. 16.c). Here, we measure the branching factor with and without data nodes (which represent leaf nodes in the *SemIndex* graph, with no outgoing edges). One can realize that the average branch factor with data nodes (when considering the whole of IMDB and WordNet, i.e., 100% *SemIndex* chunk size) amounts to 11.3, whereas it drastically increases to 17.5 when disregarding data nodes. The reason is that *SemIndex* contains a huge number of data nodes (i.e., leaf nodes) which considerably decrease the average branch factor score when

considered. Thus we chose to also measure average branching without data nodes, in order to more precisely reflect *SemIndex*'s rich inner (index) node connectivity (e.g., averaging around 17.5 outgoing edges per node). Here, one can realize that the branching factor varies logarithmically with increasing *SemIndex* – and thus IMDB and WordNet – chunk size, and almost stabilizes (at around 17.5 outgoing edges) with chunk sizes larger than 50%. This means that *SemIndex* node branching becomes more or less uniform when considering more than half of the IMDB and WordNet input sources.

A third observation can be made regarding *SemIndex*'s average spanning factor, i.e., the average length of the path from a root (abstract sense) node to a leaf (data) node in the *SemIndex* graph considering hierarchical relations only (to avoid loops), namely *hyponymy*/*hypernymy* (i.e., *IsA*/*HasA*) and *meronymy*/*holonymy* (i.e., *partOf*/*hasPart*, *memberOf*/ *hasMember*, and *substanceOf*/*hasSubstance*). One can realize that the average span of the *SemIndex* graph (following each hierarchical relation) increases in an almost quadratic manner w.r.t. the size of *SemIndex* chunks (namely with the *IsA/Has/A* relationship) since *SemIndex*' structure maps to that of the adopted reference knowledge base: i.e., WordNet in our case (cf. WordNet's average span factor in Table 4 which is quadratic w.r.t. its chunk size). Note that *SemIndex*' spanning factor is marginally affected by IMDB chunk size since the database does not include hierarchical (semantic/lexical) relations[1].

7.4.2. Comparison with Legacy Inverted Index Size and Characteristics

In addition, we have also measured the characteristics and size of legacy *InvIndex* (cf. Table 6) in comparison with *SemIndex* (cf. Fig. 17).

Table 6. Characteristics of *InvIndex* (w.r.t. IMDB) chunks.

Chunk %	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
Size (in MBs)	25.5781	49.6250	73.6719	98.7188	122.7656	147.8125	171.8594	195.8906	220.9375	244.9844
N# of Data Objects	14304	28608	42912	57217	71521	85825	100130	114434	128738	143043
N# of Index Terms	73255	115570	153982	188589	223904	254894	282939	309049	330155	356904

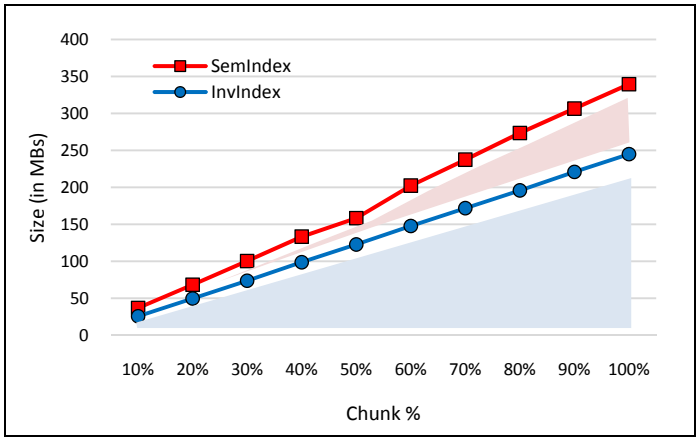


Fig. 17. Comparing *SemIndex* size with *InvIndex* size.

Results show that *SemIndex*'s size is larger only by (almost) 1/3rd of the size of *InvIndex*. This increase in size is less pronounced than the increase in build time of *SemIndex* w.r.t. *InvIndex* (which was 4 times larger, cf. Section 7.3.2), which follows the difference in sizes between the textual database (IMBD) and the knowledge graph (WordNet) used: WordNet (≈ 26 MBytes) is almost 1/3rd the size of IMDB (≈ 75 MBytes), which reflect in the sizes of *SemIndex* (coupling IMDB with WordNet) and *InvIndex* (referencing IMDB only).

7.5. Query Processing Time

To test the performance of *SemIndex*, we formulated different kinds of queries organized in two categories: i) *unrelated queries*, and ii) *expanded queries*, as shown in Table 7.

---

[1] Including the data *contained-in* relation, which originates from the database (e.g., IMDB) index graph ($\tilde{G}_\Delta$), would increase the average *SemIndex* span score by one (i.e., including one additional hierarchical level to access data nodes), regardless of the database chunk size.

Table 7. Test queries.

Query group *Q1* – *Unrelated queries*	
ID	**Terms**
Q1_1	"time"
Q1_2	"love", "date"
Q1_3	"fly", "power", "man"
Q1_4	"robot", "human", "war", "world"
Q1_5	"mafia", "kill", "mob", "hit", "family"

Query group *Q2* – *Expanded queries*	
ID	**Terms**
Q2_1	"car"
Q2_2	"car", "muscle"
Q2_3	"car", "muscle", "classic"
Q2_4	"car", "muscle", "classic", "speed"
Q2_5	"car", "muscle", "classic", "speed", "thrills"

The first category consists of queries with varying numbers of selection terms (keywords), e.g., from 1 (single term query) to 5, where all terms are different and all queries are unrelated (i.e., queries with no common selection terms, cf. sample query group *Q1* in Table 7). The second category consists of queries with varying numbers of selection terms, where terms are different yet queries are related: such that each query expands its predecessor by adding an additional selection term to the latter (cf. sample query group *Q2* in Table 7).

We considered 5 groups of queries (made of 5 queries each) within each category (e.g., *Q1* is one of the 5 groups of queries considered within the category of *unrelated queries*). Each query was tested on every one of the 100 combinations of *SemIndex* generated by combining the different chunks of the IMDB *movies* table (10%,20%,30%, …, 100%) with every chunk of WordNet (10%, 20%, 30%, …, 100%), at link distance threshold values varying from $l = 1$ to 5. All queries were processed 5 times each, retaining average processing time. Hence, all in all, we ran an overall of: 2 (categories) × 5 (groups) × 5 (queries) × 100 (*SemIndex* chunks) × 5 (l values) × 5 (runs) = 125000 query execution tasks. Hereunder, we present and discuss the results obtained with two sample query groups, *Q1* and *Q2*, corresponding to each category as shown in Table 7, compiled in Fig. 18 and Fig. 19 (remaining query groups show similar behavior, cf. technical report in [85], and thus were omitted here for ease of presentation).

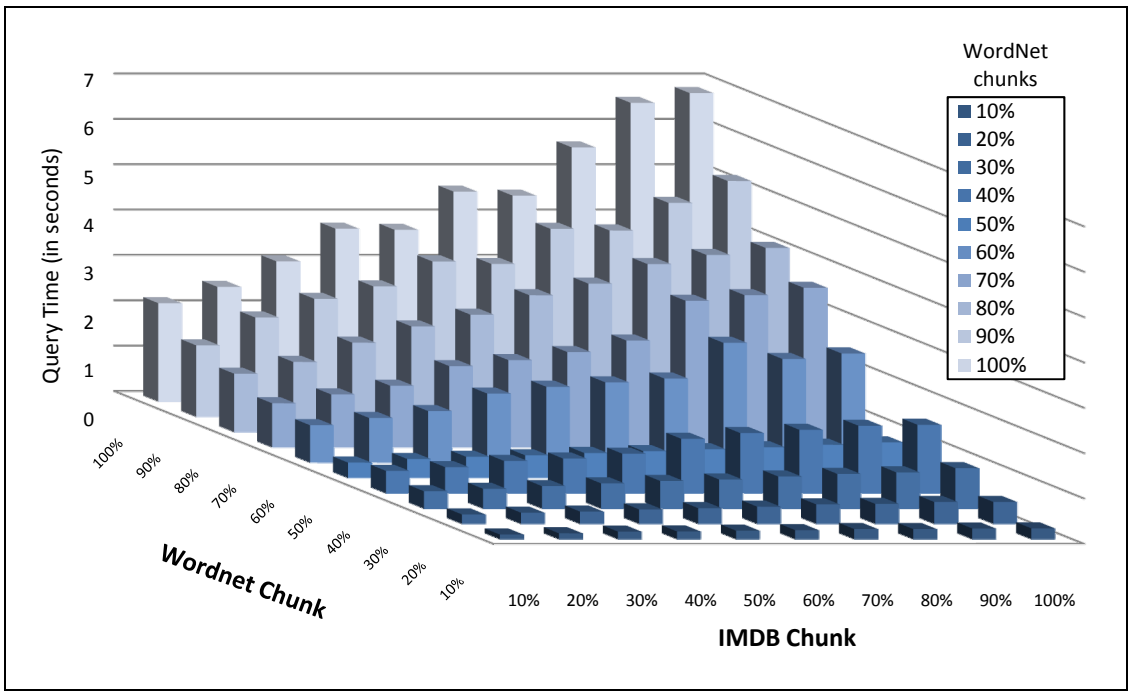


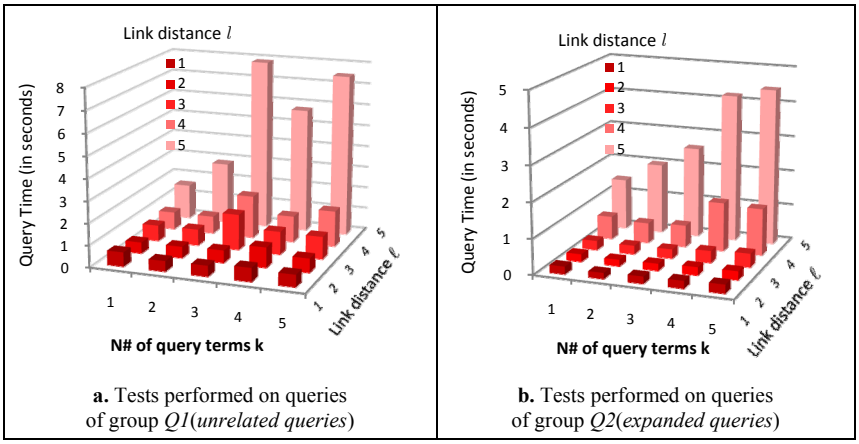
Fig. 18. Query execution time on queries of group *Q1*, considering $k = 5$ and $l = 5$, while varying IMDB and WordNet chunk sizes.

7.5.1. SemIndex Query Processing Time

On one hand, the graph in Fig. 18 plots query execution time on queries of group *Q1* w.r.t. IMBD and WordNet chunk sizes, while considering a fixed number of query terms k and a fixed link distance threshold l. Here, 25 tests (×5 runs) were conducted covering every combination of k (1-to-5) and l (1-to-5), yet we only show the graph plotted with maximum $k = 5$ and $l = 5$, since remaining graphs highlight a similar behavior (with different time amplitudes, cf. technical report in [85]). We omitted here results obtained with queries of groups *Q2* since they show a similar behavior to those of group *Q1* (details can be found in [85]). This shows that query

execution time is linear in both IMBD and WordNet chunk sizes, and thus is quadratic w.r.t. both of them (verifying our complexity analysis in Section 6.2).

On the other hand, the graphs in Fig. 19 highlight the effects of varying the number of query terms k and varying link distance l w.r.t. fixed IMDB and WordNet chunk sizes. Here, 100 tests (×5 runs) were conducted for each query group, covering every combination of IMDB chunk size (10% to 100%) and WordNet chunk size (10% to 100%). We only show the graph plotted with maximum size chunks =100%, since remaining graphs highlight a similar behavior (with different time amplitudes, cf. [85]). One can see that processing time is linear w.r.t. the number of query terms, and quadratic w.r.t. link distance, which corresponds to the time complexity of *Dijkstra*'s algorithm in navigating the edges (i.e., pairs of nodes) of the *SemIndex* graph (cf. Section 6.2).



a. Tests performed on queries of group *Q1*(*unrelated queries*)

b. Tests performed on queries of group *Q2*(*expanded queries*)

Fig. 19. Query execution time when running queries of groups *Q1* and Q2, considering IMDB chunk = 100% and WordNet chunk = 100%, while varying the number of query terms k and link distance threshold l.

Note that with queries of group *Q1* (Fig. 19.a), at k=3, query time seems "strangely" high, in comparison with the overall behavior of the chart, and compared to the charts of query groups *Q2* (Fig. 19.b). This is due to the fact that terms in query *Q1_3* (i.e., "*fly*", "*power*", "*man*") happen to have more neighbors in their *SemIndex* graph (i.e., higher branch factor) than remaining query terms in group *Q1*, which exploration requires more time (as shown in the following section). As for the time results of query group *Q2*, the time slope increases regularly since the processing time of a given query *Q2_i* covers the processing time of *Q2_i-1* plus the time needed to process the additional term in *Q2_i*, given that larger queries in *Q2* expand smaller ones.

7.5.2. Breakdown of SemIndex Query Processing Time

Similarly to *SemIndex* building time experiments, we broke down query execution time in order to better understand the system's behavior (and identify potential time optimization strategies to be investigated in the future). Fig. 20 and Fig. 21 plot CPU time versus SQL time (I/O) while: i) varying IMBD and Wordnet chunk sizes, with fixed query size k and link distance l (Fig. 20), and ii) varying query size k and link distance l, with fixed IMDB and Wordnet chunk sizes (Fig. 21).

Here, 25 tests (×5 runs) were first conducted for each query group covering every combination of k (1-to-5) and l (1-to-5), yet we only show the graph plotted with maximum k = 5 and l = 5 (Fig. 20), since remaining graphs highlight a similar behavior (cf. [85]). Likewise, 100 tests (×5 runs) graphs were then conducted for each query group covering every combination of IMDB chunk size (10% to 100%) and Wordnet chunk size (10% to 100%), yet we only show the graph plotted with maximum size chunks =100% (cf. Fig. 21), since remaining graphs highlight a similar behavior. Also, we omit results obtained with queries of group *Q2* since they show a similar behavior to those of *Q1* (cf. [85]).

At this point, in addition to the quadratic time dependency w.r.t. IMDB and WorldNet chunk sizes (Fig. 20), as well as quadratic time dependency w.r.t. the number of query terms k and link distance l (Fig. 21, which were highlighted in the previous section), one can clearly realize that the bulk of execution time goes to SQL processing (executing SQL statements in order to fetch information from IMBD and WordNet, at the MySQL database server side) which takes up to 96% of total query processing time, whereas CPU processing (running non-SQL instructions at the Java software side) requires less than 4% of total execution time.

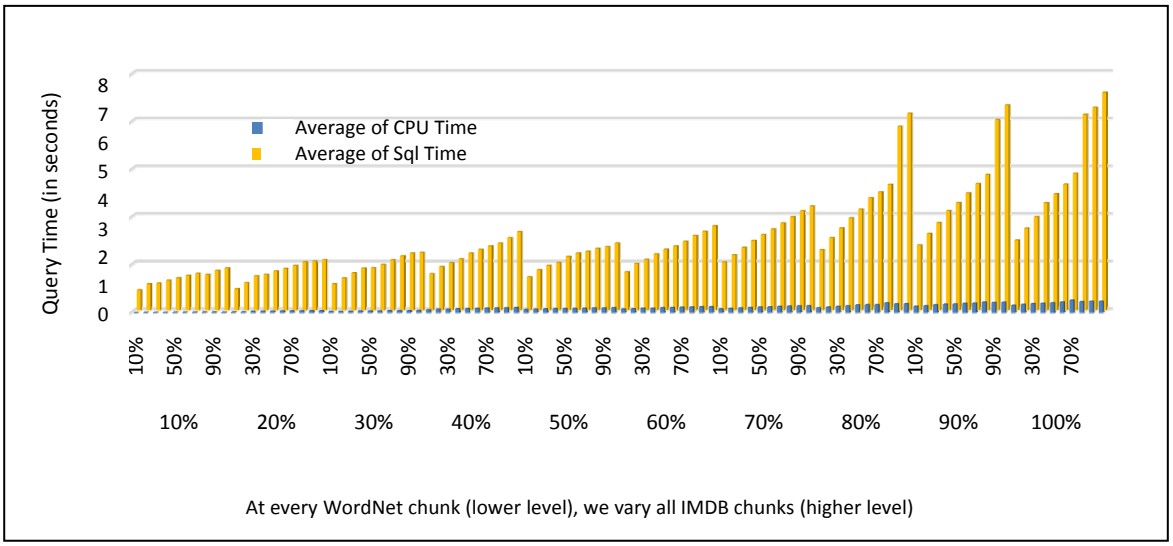


Fig. 20. CPU versus SQL time on queries of group *Q1*, considering $k = 5$ and $l = 5$, while varying IMDB and Wordnet chunk sizes.

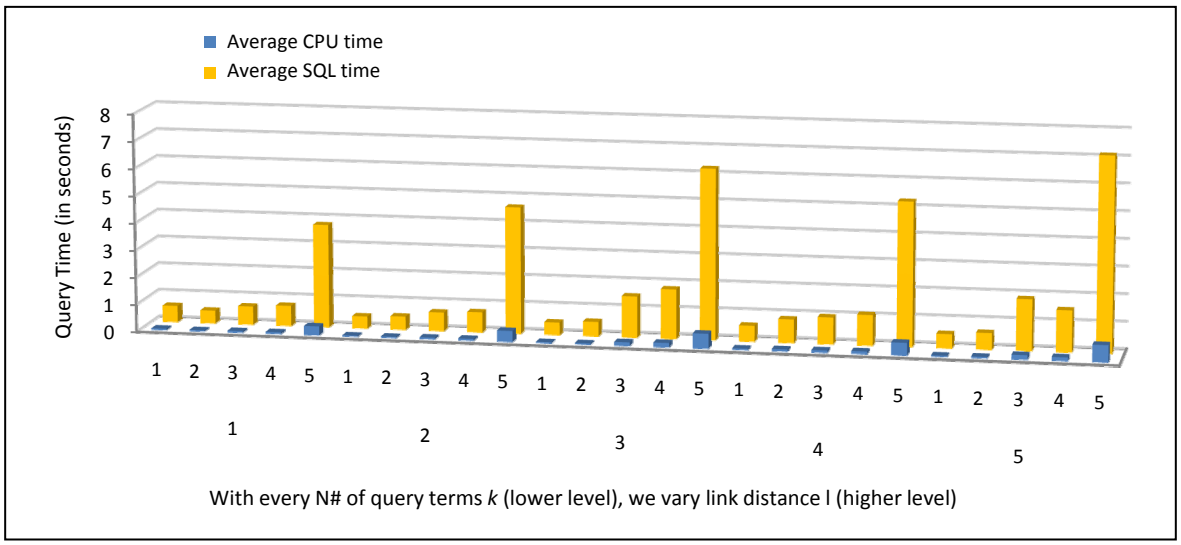
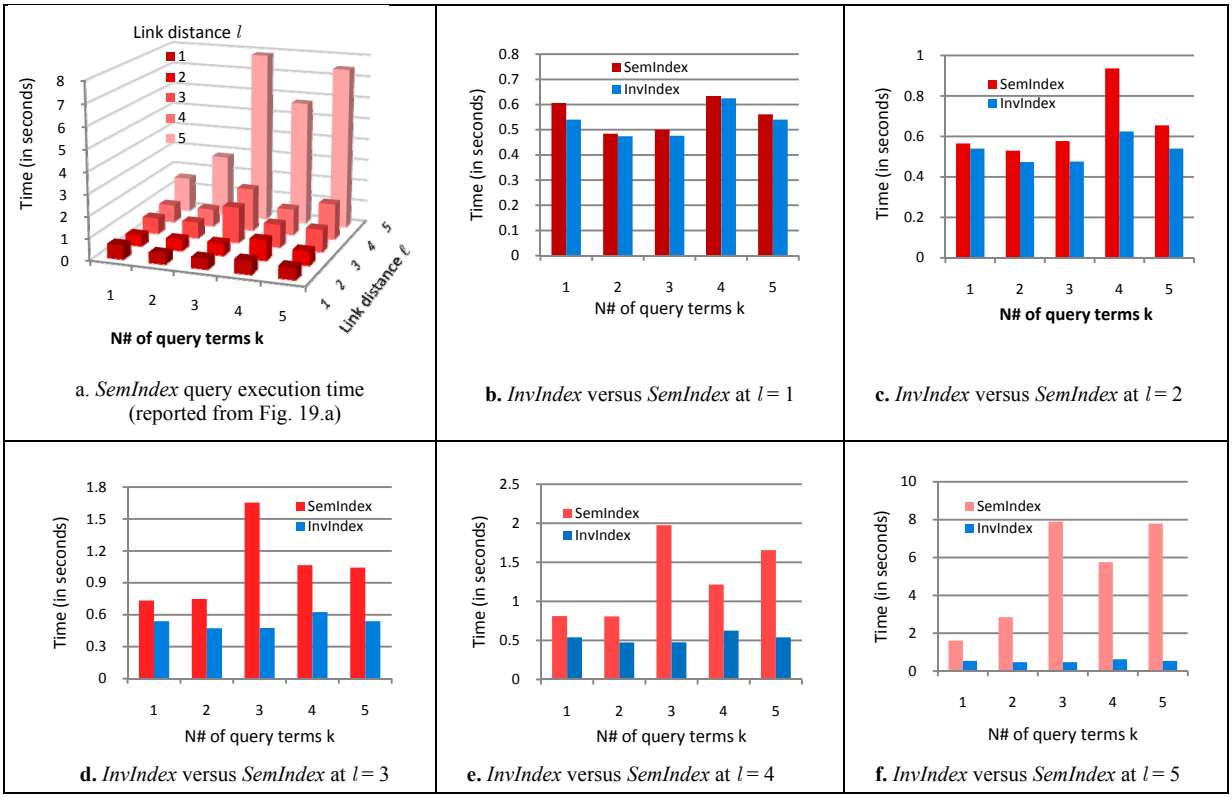


Fig. 21. CPU versus SQL query execution time on queries of group *Q1*, considering IMDB chunk = 100% and Wordnet chunk = 100%, while varying the number of query terms *k* and link distance threshold *l*.

7.5.3. Comparison with Legacy Inverted Index Query Processing Time

We ran the same querying tasks through the legacy *InvIndex* built on top of IMDB, and compared the obtained query time results with those of S*emIndex*. Fig. 22 and Fig. 23 provide results obtained when running queries of group *Q1* (*unrelated*), plotted by varying the number of query terms *k* (Fig. 22) and *SemIndex* link distance threshold *l* in (Fig. 23). Similar results were obtained with queries of group *Q2* (*expanded*) and have been omitted here for clarity of presentation (they are provided in [85]).

On one hand, results in Fig. 22 and Fig. 22 show that *SemIndex* and *InvIndex* have very close query time levels when link distance is small (l=1 and l=2), such that *SemIndex* time increases as link distance increases, reaching its highest levels with $l = 5$ (i.e., almost 8 times higher than *InvIndex* time levels, with *SemIndex* reaching 5 hops into the semantic graph structure to identify more semantically related results). On the other hand, Fig. 22 shows that both *SemIndex* and *InvIndex* query time levels slightly increase when increasing the number of query keywords *k* with small link distances (l=1 and l=2), such that the pace of increase tends to augment with *k* when reaching higher link distance thresholds (l=3-to-5).



a. *SemIndex* query execution time (reported from Fig. 19.a)

b. *InvIndex* versus *SemIndex* at $l = 1$

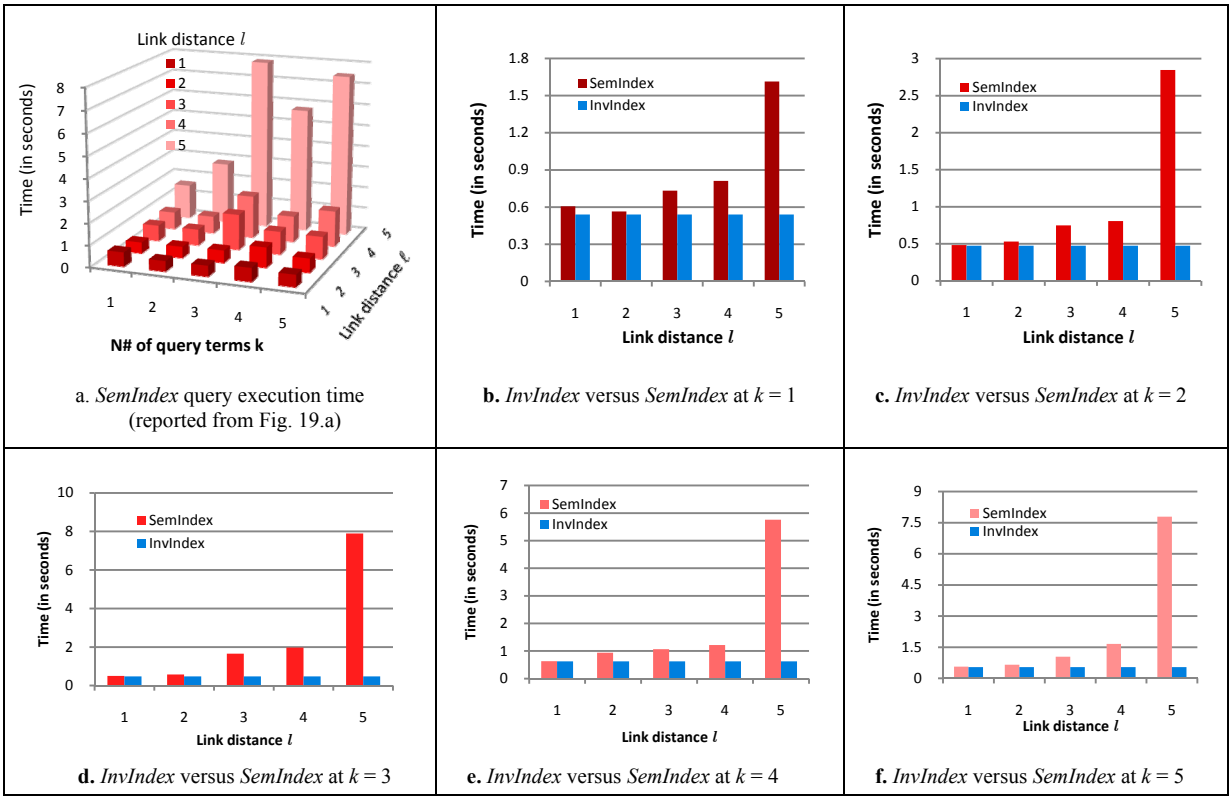
c. *InvIndex* versus *SemIndex* at $l = 2$

d. *InvIndex* versus *SemIndex* at $l = 3$

e. *InvIndex* versus *SemIndex* at $l = 4$

f. *InvIndex* versus *SemIndex* at $l = 5$

Fig. 22. Comparison with legacy *InvIndex* query execution time, considering queries of group *Q1* (*unrelated queries*), while varying the number of query terms k and fixing link distance threshold l (the latter affecting *SemIndex*).



a. *SemIndex* query execution time (reported from Fig. 19.a)

b. *InvIndex* versus *SemIndex* at $k = 1$

c. *InvIndex* versus *SemIndex* at $k = 2$

d. *InvIndex* versus *SemIndex* at $k = 3$

e. *InvIndex* versus *SemIndex* at $k = 4$

f. *InvIndex* versus *SemIndex* at $k = 5$

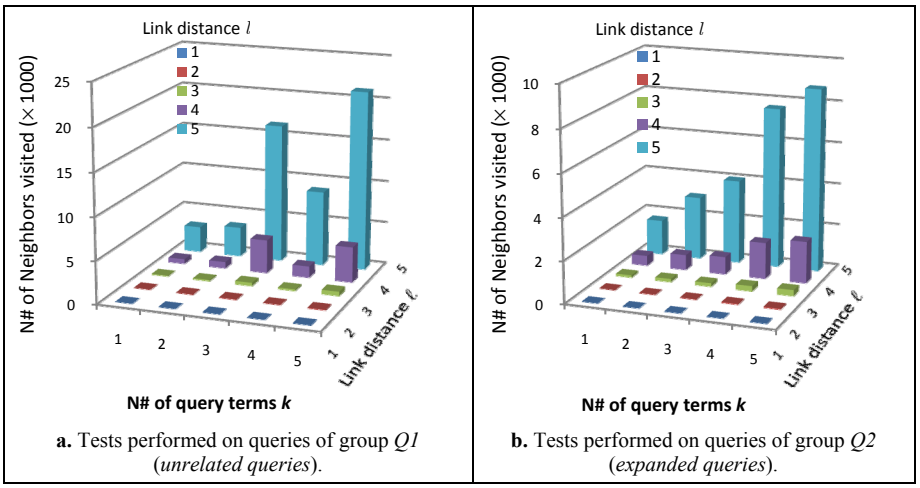
Fig. 23. Comparing *SemIndex* and legacy *InvIndex* query execution time, considering queries of group *Q1* (*unrelated queries*), while varying link distance threshold l (affecting *SemIndex*) and fixing the number of query terms k.

In other words, the time to navigate the semantic graph, following the allowed link distance *l*, remains the foremost determining factor in query execution time. Also, results in Fig. 23 show that *InvIndex* query time is invariant w.r.t. variations in link distance *l* (since it does not navigate the semantic graph, and thus does not perform semantic-aware processing).

7.6. Query Results Evaluation

7.6.1. Number of Returned Results using SemIndex

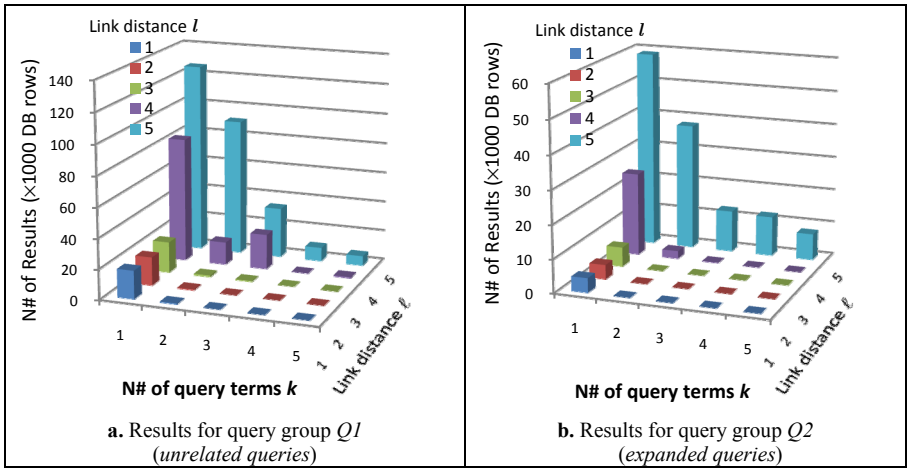
To better evaluate query execution time, we also measured the number of neighboring nodes visited in the *SemIndex* graph when running each query, and the number of results (n# of data objects = n# of IMDB *movies* table rows) returned per query.



a. Tests performed on queries of group *Q1* (*unrelated queries*).

b. Tests performed on queries of group *Q2* (*expanded queries*).

Fig. 24. Number of neighbors visited when running queries of groups *Q1* and *Q2*, using fixed IMDB chunk = 100% and WordNet chunk = 100%, while varying the number of query terms *k* and link distance threshold *l*.



a. Results for query group *Q1* (*unrelated queries*)

b. Results for query group *Q2* (*expanded queries*)

Fig. 25. Number of returned results when running queries of groups *Q1* and *Q2*, using fixed IMDB chunk = 100% and WordNet chunk = 100%, while varying the number of query terms *k* and link distance threshold *l*.

On one hand, the neighbors' charts in Fig. 24.a, b, and c are directly proportional to time charts in Fig. 19.a, b, and c respectively. In other words, the amount of neighboring nodes visited in the *SemIndex* graph (which depends on the graph's connectivity, and the query terms used: the starting index nodes used when exploring the *SemIndex* graph) shows a direct and proportional impact on query execution time: the more neighbors to be explored, the more time it will require our *SearchTerms* (*Dijkstra*-based) algorithm to explore the *SemIndex* graph. This explains the steep increase in query time when running queries *Q1-3* (Fig. 19.a, at $k = 3$) in comparison with the overall behavior of the chart, and compared with the charts of query group *Q2* (Fig. 24.b).

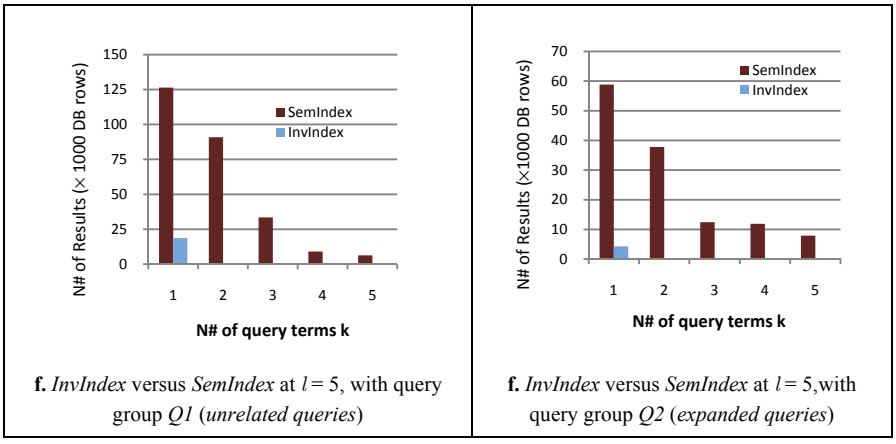
On the other hand, the query result charts in Fig. 25 are proportional to the time charts in Fig. 19 along the *l* (link distance) axis, while inversely proportional along the *k* (number of query terms) axis. In other words, the number of results increases as link distance *l* increases, yet decreases as the number of query terms *k* increases.

The behavior regarding link distance l can be justified since: increasing l would increase the shortest path length per query term, which would increase the number of potential shortest path intersections in the *SemIndex* graph, i.e., returning a higher number of potential results. An extreme case occurs when processing query *Q1_1* and *Q1_2*, with k = 1 (single term query: *"time"*) and l = 5 (maximum link distance in our test, cf. Fig. 25) which returns around 90% of the IMDB *movies* table rows.

The behavior regarding the number of query terms k is due to processing a higher k, which means identifying the intersection between a higher number of shortest paths in the *SemIndex* graph, yielding a lesser number of potentially successful intersections, i.e., a lesser number of returned results. In other words, processing a more selective query (with a higher number of terms) means producing more selective (yet lesser) results. Extreme cases occur with extremely selective queries: i.e., *Q1_4* and *Q1_5* (k = 4 and 5 respectively), run with reduced link distance thresholds: l = 1, 2 and 3, which produce *zero* results (i.e., *zero* path intersections starting from multiple index terms mapping to each of the keywords).

7.6.2. Comparing with Number of Returned Results obtained using Legacy Inverted Index

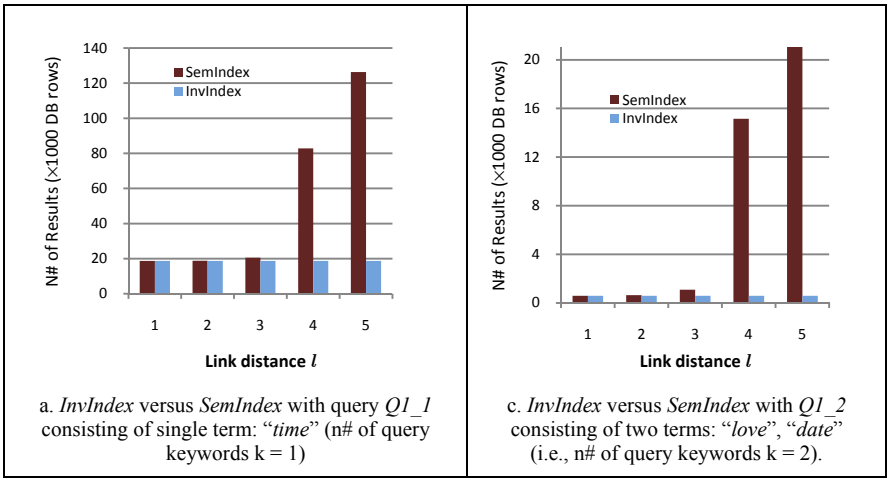
Similarly, we measured and compared the number of results returned per query when using legacy *InvIndex*, with the number of results obtained using *SemIndex*. Fig. 26 and Fig. 27 showcase the huge impact of *SemIndex* in retrieving (4 to 7 times) more results than legacy *InvIndex* (here, we only show a set of select result graphs, the complete set being provided in [85]).



f. *InvIndex* versus *SemIndex* at l = 5, with query group *Q1* (*unrelated queries*)

f. *InvIndex* versus *SemIndex* at l = 5,with query group *Q2* (*expanded queries*)

Fig. 26. Comparing the number of returned results using *InvIndex* versus *SemIndex*, by varying the number of query terms k while fixing *SemIndex*'s link distance threshold l.



a. *InvIndex* versus *SemIndex* with query *Q1_1* consisting of single term: "*time*" (n# of query keywords k = 1)

c. *InvIndex* versus *SemIndex* with *Q1_2* consisting of two terms: "*love*", "*date*" (i.e., n# of query keywords k = 2).

Fig. 27. Comparing the number of returned results using *InvIndex* versus *SemIndex*, by varying *SemIndex*'s link distance threshold l while fixing the number of query terms k.

On one hand, Fig. 25 shows that the number of results returned by *SemIndex* decreases with the increase in the number of keywords k, yet remains far greater than the number of results obtained with *InvIndex*, the latter producing no results with queries made of k=3, 4, and/or 5 keywords. On the other hand, Fig. 26 shows that the

number of results returned by *SemIndex* increases significantly with the increase of link distance l, while it stagnates with *InvIndex* which is not affected by l (since it does not navigate the semantic graph to produce more semantically related results).

7.6.3. Quality of Returned Results using SemIndex versus Legacy InvIndex

In addition to evaluating *SemIndex*' efficiency (processing time), we also evaluated its effectiveness (result quality), i.e., evaluating the interestingness of semantic-aware answers from the user's perspective. To do so, we collected the results of our test queries obtained with and without semantic indexing, i.e., querying using the legacy *InvIndex* (which is equivalent to executing queries as *standard containment queries* (l = 1) in *SemIndex*), and performing *semantic-aware queries* (l = 2-to-5) with *SemIndex*. Results were mapped against user feedback (user judgments, utilized as *golden truth*) evaluating the quality of the matches produced by the system by computing *precision* and *recall* metrics commonly utilized in IR evaluation [8]. *Precision* (*PR*) identifies the number of correctly returned results, w.r.t. the total number of results (correct and false) returned by the system. *Recall* (*R*) underlines the number of correctly returned results, w.r.t. the total number of correct results, including those not returned by the system. In addition to comparing one approach's *precision* improvement to another's *recall*, it is a widespread practice to consider the *f-value*, which represents the harmonic mean of *precision* and *recall,* such that high *precision* and *recall*, and thus high *f-value* characterize good retrieval quality [63]. Ten test subjects (six master students, and four doctoral students, who were not part of the system development team) were involved in the experiment as human judges. Testers were asked to evaluate the quality of the top 1000 results (movie objects returned) per query (since manually evaluating the tens of thousands of obtained results – cf. Fig. 25 – is practically infeasible) obtained with l = 6 (as an upper bound of l=5, including potentially more results than l=1-to-5). These were randomized before being shown to testers. Manual relevance ratings (in the form of integers $\in$ {-1, 0, 1}, i.e., {*not relevant*, *neutral*, *relevant*}) were acquired for each query answer. Then, we quantified inter-tester agreement, by computing pair-wise correlation scores[1] among testers for each of the rated query answers, and subsequently selected the top 500 hundred answers per query having the highest average inter-tester correlation scores[2], which we utilized as the experiment's *golden truth*. Results for queries of groups *Q1* (*unrelated queries*) and *Q2* (*expanded queries*) are shown in Fig. 28 and Fig. 29 respectively, whereas overall *f-value* results are provided in Fig. 30. Results highlight several observations.

1) ***Precision*** and ***link distance***: One can realize that precision levels computed with both query groups *Q1* (*unrelated queries*, Fig. 28.a) and *Q2* (*expanded queries*, Fig. 29.a) generally increase with link distance (l), until reaching l = 3 (with Q2) or l = 4 (with Q1) where precision starts to slightly decrease toward l = 5. On one hand, this shows that the number of correct (i.e., user expected) results increases as more semantically related terms are covered in the querying process (with l > 1). On the other hand, this also shows that over-navigating the *SemIndex* graph to link terms with semantically related ones located as far as $l \geq 3$ hops away might include results which: i) are somehow semantically related to the original query terms, but which ii) are not necessarily interesting for the users. For instance, term *"congo"* (meaning: *black tea grown in China*) is linked to term *"time"* through l = 5 hops in *SemIndex* ("*time*" >> "*snap*" >> "*reception*" >> "*tea*" >> "*congo*"). Yet, results (movie objects) containing term *"congo"* (e.g., movies about the country *Congo*, or its continent *Africa*) were not judged to be relevant by human testers when applying query *"time"* (testers were probably expecting movies about the *passage of time* or *time travel* instead, etc.).

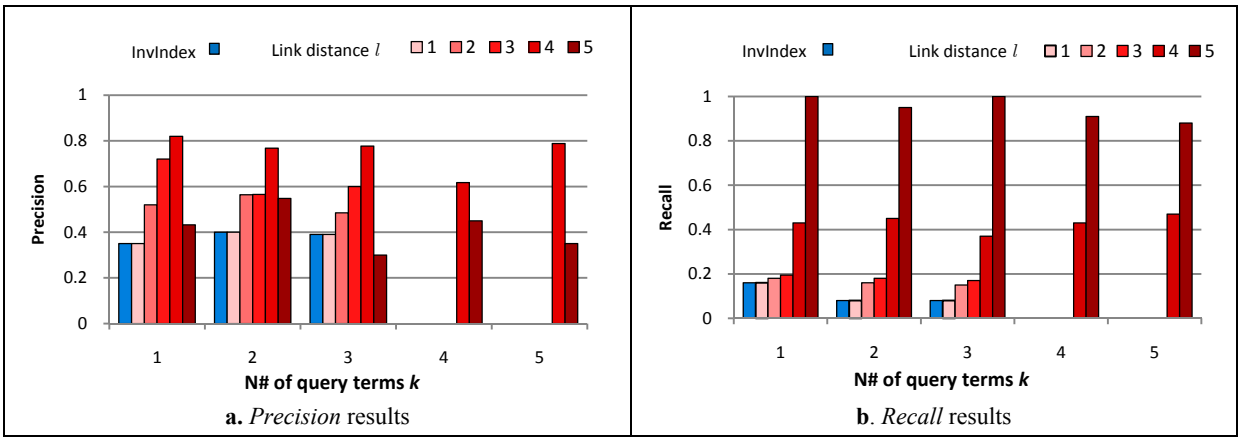
2) ***Precision*** and ***number of query terms***: Here, one can realize that precision levels with queries of group *Q1* (*unrelated queries*, Fig. 28.a) do not seem to largely vary w.r.t. the number of terms (k) per query, whereas precision levels with queries of group *Q2* (*expanded queries*, Fig. 29.a) clearly increase with k. These seemingly different results are due to the human testers' expectations, where testers were required to judge the quality of each query's results given the user's supposed intent. On one hand, given that queries in group *Q1* are unrelated, result quality was evaluated separately for each query, based on the query's own keyword terms (e.g., the intent of query *Q1_1* is identifying movies that have to do with *time*, and the intent of *Q1_3* is movies that have to do with a *flying power man*, etc.). On the other hand, given that queries in group *Q2* are expanded versions of one

---

[1] Using *Pearson Correlation Coefficient* (*PCC*), producing scores $\in$ [-1, 1] such that: -1 designates that one tester's ratings is a decreasing function of the other tester's ratings (i.e., answers deemed relevant by one tester are deemed irrelevant by the other, and vice versa), 1 designates that one tester's ratings is an increasing function of the other tester's ratings (i.e., answers are deemed relevant/irrelevant by testers alike), and 0 means that tester ratings are not correlated.

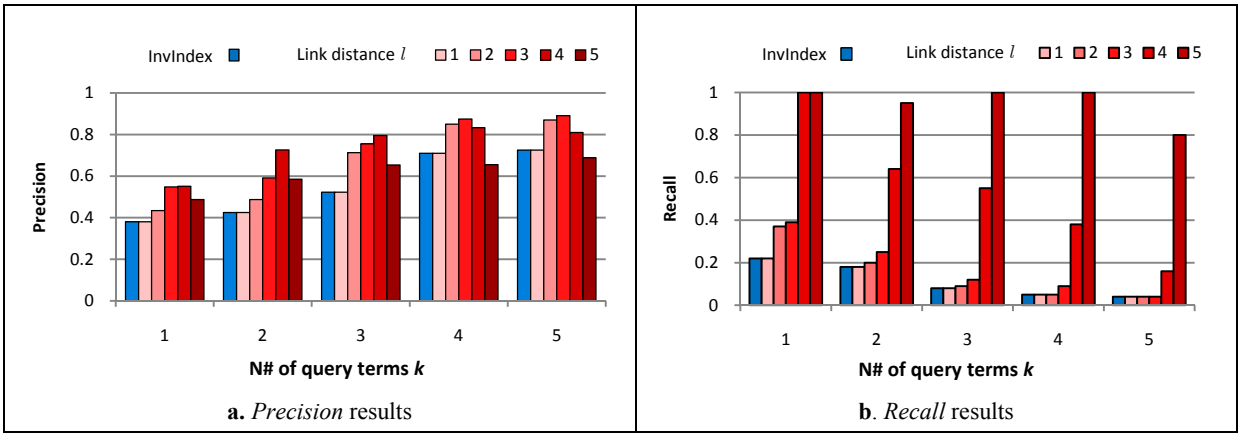
[2] Having average inter-tester *PCC* score $\geq$ 0.4.

another, result quality was evaluated based on the user's intent: which would be naturally expressed with the most expanded (i.e., most expressive) query: *Q2_5*. One can realize that using fewer query terms here produces lower precision levels, which is due to the system returning more results which are (semantically related to the query terms but which are) not necessary related to the user's intent (e.g., query *"car"* might return movies that have to do with *trains* or *taxi cabs*, whereas the user is apparently searching for movies that have to do with *muscle cars* with *speed* driving and *thrills*, cf. *Q2_5*). In other words, with query group *Q2*: the lesser the number of query terms used, the lesser the query's expressiveness w.r.t. user's intent, and thus the larger the number of returned results which are not necessarily related to the user's intent: producing lower precision.



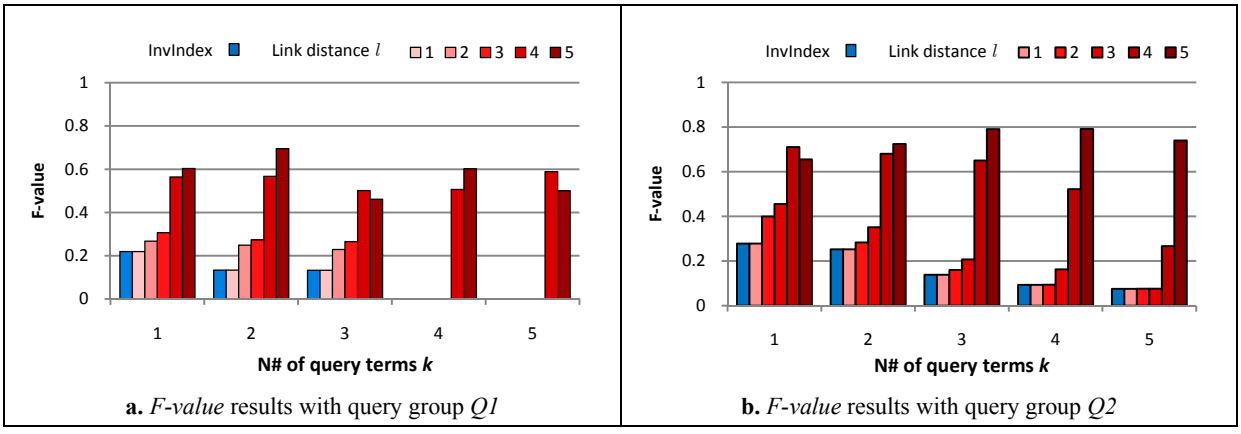
a. *Precision* results **b**. *Recall* results

Fig. 28. Comparing precision (PR) and recall (R) results obtained using *SemIndex* versus legacy *InvIndex*, with query group *Q1* (*unrelated queries*), varying the number of query terms *k* and link distance *l* (the latter affecting *SemIndex*).



a. *Precision* results **b**. *Recall* results

Fig. 29. Comparing precision (PR) and recall (R) results obtained using *SemIndex* versus legacy *InvIndex*, with query group *Q2* (*expanded queries*), varying the number of query terms *k* and link distance *l* (the latter affecting *SemIndex*).



a. *F-value* results with query group *Q1* **b.** *F-value* results with query group *Q2*

Fig. 30. Comparing *f-value* levels obtained using *SemIndex* versus legacy *InvIndex*, with query group *Q1* (*unrelated queries*), and query group *Q2* (*expanded queries*).

3) ***Recall*** and ***link distance***: As for recall, one can realize that levels obtained with both *Q1* and *Q2* steadily increase with link distance (l) varying from $l = 1$ (legacy *InvIndex*) to 5 (Fig. 28.a and Fig. 29.b). This maps to observation 1, where the number of correct (i.e., user expected) results returned by the system increases as more semantically related terms are covered in the querying process. In other words, the more the number of correct results which are returned by the system, the fewer the number of correct results which are not returned, and thus the higher the recall levels. Note that returning noisy (incorrect) results along with the correct ones does not affect recall (but rather affects precision as explained in observation 1).

4) ***Recall*** and ***number of query terms***: Recall levels vary in a similar fashion to precision levels when varying link distance (l): increasing with the increase of l, which accounts for more semantic coverage (returining more semantically related results) in the *SemIndex* graph (Fig. 28.b and Fig. 29.b). However, recall levels tend to decrease (rather than increase) with k. This is due to the fact that shorter (less expressive) queries (i.e., with smaller k values) will naturally return more (semantically related) results than larger queries made of multiple terms (larger k) which will necessarily identify less results (e.g., lesser number of movie objects matching the query's terms). Hence, a decrease in the number of returned results (with increasing k values) meant the number of correct (and incorrect) results (naturally) decreased, which lead to a decrease in recall.

5) As for **f-value** results, levels clearly and significantly increase with the increase of link distance l, whereas they slightly decrease with the increase of the number of query keywords k. This naturally confirms the precision and recall levels obtained above, where the determining factor affecting retrieval quality remains link distance l, whereas an increase in the number of keywords k tends to reduce system recall with higher values of k (queries becoming very selective, thus missing some relevant results). Note that f-value levels are consistently significantly higher than those obtained with the legacy *InvIndex*, highlighting a substantial improvement of semantic-aware retrieval quality over syntactic retrieval quality.

7.7. Evaluating Query Efficiency/Effectiveness Ratios

To sum up, and in order to evaluate the benefits of *SemIndex* querying over legacy *InvIndex* querying, we compute the ratio between improvement in query effectiveness (result quality) and reduction in efficiency (query execution time). In other words, we would like to study if the cost (in execution time) of obtaining (higher quality) semantic-aware query results using *SemIndex* is worthwhile, in comparison with the faster yet less effective *InvIndex*. To do so, we first evaluate the ratio (expressed in percentage) of increase in *query execution time* (cf. Table 8) as well as the ratio (percentage) of increase in *query result quality* (i.e., f-value scores, cf. Table 9) when using *SemIndex* versus *InvIndex*. Both ratios were evaluated for the different combinations of link distance thresholds l and number of query terms k, using the following formulas:

$$\eta_{Efficiency} = \frac{QueryTime_{SemIndex} - QueryTime_{InvIndex}}{QueryTime_{InvIndex}} \quad \textbf{(2)} \qquad \eta_{Effectiveness} = \frac{\text{f-value}_{SemIndex} - \text{f-value}_{InvIndex}}{\text{f-value}_{InvIndex}} \quad \textbf{(3)}$$

Table 8. Percentage of increase in *query execution time*, when using *SemIndex* versus legacy *InvIndex*.

a. Processing queries of group *Q1* (*unrelated*)

	l=1	l=2	l=3	l=4	l=5
Q1_1 (k=1)	12.22%	4.63%	35.74%	50.37%	198.89%
Q1_2 (k=2)	2.11%	11.81%	58.02%	70.46%	500.63%
Q1_3 (k=3)	5.04%	21.22%	247.48%	315.13%	1558.40%
Q1_4 (k=4)	1.12%	49.76%	70.56%	94.56%	821.60%
Q1_5 (k=5)	3.89%	21.30%	93.15%	206.85%	1343.33%

b. Processing queries of group *Q2* (*expanded*)

	l=1	l=2	l=3	l=4	l=5
Q2_1 (k=1)	6.34%	6.34%	29.76%	220.00%	607.80%
Q2_2 (k=2)	5.52%	14.72%	52.76%	253.99%	1125.15%
Q2_3 (k=3)	2.53%	2.53%	33.84%	215.15%	1200.00%
Q2_4 (k=4)	7.92%	15.84%	69.80%	587.13%	1969.80%
Q2_5 (k=5)	1.63%	1.63%	58.54%	439.43%	1701.22%

Table 9. Percentage of increase in *query result quality* (i.e., f-value) when using *SemIndex* versus legacy *InvIndex*.

a. F-value increase with query group *Q1* (*unrelated*)

	l=1	l=2	l=3	l=4	l=5
Q1_1 (k=1)	0.00%	21.78%	39.74%	156.89%	174.74%
Q1_2 (k=2)	0.00%	86.96%	104.77%	325.62%	421.30%
Q1_3 (k=3)	0.00%	72.58%	99.55%	277.57%	247.63%
Q1_4 (k=4)	0.00%	0.00%	0.00%	4968.00%	5922.06%
Q1_5 (k=5)	0.00%	0.00%	0.00%	5788.08%	4908.13%

b. F-value increase with query group *Q2* (*expanded*)

	l=1	l=2	l=3	l=4	l=5
Q2_1 (k=1)	0.00%	43.34%	64.74%	154.97%	135.05%
Q2_2 (k=2)	0.00%	23.23%	77.69%	405.22%	437.75%
Q2_3 (k=3)	0.00%	39.67%	126.88%	612.35%	766.41%
Q2_4 (k=4)	0.00%	0.00%	0.00%	2546.32%	3913.45%
Q2_5 (k=5)	0.00%	0.00%	0.00%	466.33%	3649.94%

On one hand, results in Table 8 show that querying using *SemIndex*, under all considered combinations of parameters l and k, requires between 6.34% (query *Q2_1* of group *Q2*, with l=1 and k=1) and up to 1701.22% (query Q2_5 of group *Q2*, with l=5 and k=5) more processing time than *InvIndex*. On the other hand, Table 9 shows that query result quality levels increase with *SemIndex*, from 0% (when l=1, where *SemIndex* performs semantic-free *standard containment queries*) up to 5922.06% (with query *Q2_4* of group *Q1*, with l=5 and k=4) w.r.t. the quality levels of *InvIndex*.

Consequently, we compute the ratios between improvement in *result quality* and increase in *query execution time*, when using *SemIndex* versus *InvIndex*, for all combinations of link distance l and number of query terms k, using formula (4):

$$\eta_{Quality/Time} = \frac{\eta_{Effectiveness}}{\eta_{Efficiency}} \quad \textbf{(4)}$$

Values of the $\eta_{Quality/time}$ ratio varies as follows:

- $\eta_{Quality/time} < 1$ means *SemIndex*'s improvement in result quality did not surpass the increased cost in query processing time in comparison with *InvIndex*. For instance, a 10% improvement in result quality which requires a 20% increase in query execution time would yield $\eta_{Quality/time} = 0.5$, such that improving result quality requires double the effort in query execution time.
- $\eta_{Quality/time} = 1$ means *SemIndex*'s improvement in result quality exactly matches the increased cost in query processing time in comparison with *InvIndex*. In other words, a 10% improvement in result quality would require exactly a 10% increase in query execution time to obtain $\eta_{Quality/time} = 1$.
- $\eta_{Quality/time} > 1$ means SemIndex's improvement in result quality surpassed the increased cost in query processing time in comparison with *InvIndex*. For instance, a 20% improvement in result quality which requires only a 10% increase in query execution time would yield $\eta_{Quality/time} = 2$, such that the effort put in query execution time doubled the increase in query result quality.

Table 10. Ratio of improvement of *result quality* over increase of *query execution time*, when using *SemIndex* versus legacy *InvIndex*.

a. Ratio with query group *Q1* (unrelated)

	l=1	l=2	l=3	l=4	l=5
Q1_1 (k=1)	0.00	4.70	1.11	3.11	0.88
Q1_2 (k=2)	0.00	7.36	1.81	4.62	0.84
Q1_3 (k=3)	0.00	3.42	0.40	0.88	0.16
Q1_4 (k=4)	0.00	0.00	0.00	52.54	7.21
Q1_5 (k=5)	0.00	0.00	0.00	27.98	3.65

b. Ratio with query group *Q2* (expanded)

	l=1	l=2	l=3	l=4	l=5
Q2_1 (k=1)	0.00	6.84	2.18	0.70	0.22
Q2_2 (k=2)	0.00	1.58	1.47	1.60	0.39
Q2_3 (k=3)	0.00	15.71	3.75	2.85	0.64
Q2_4 (k=4)	0.00	0.00	0.00	4.34	1.99
Q2_5 (k=5)	0.00	0.00	0.00	1.06	2.15

Results in Table 10 show that in most cases, *SemIndex*'s improvement in query result quality surpasses the cost put into query execution time in comparison with *InvIndex*, varying from $\eta_{Quality/Time} = 1.11$ with *Q1_1* at l=3 and reaching as high as $\eta_{Quality/Time} = 52.54$ with query group *Q1_4* at l=4 (i.e., improvement in quality is equivalent to 51.54 times the increase in query execution cost). We also note that $\eta_{Quality/time}$ was less pronounced in certain cases, especially with low link distance values (e.g., with l<2 or l<3), and sometimes with certain specific queries (e.g., query *Q1-3* of group *Q1*). Computing the average and standard deviation scores (considering all queries and link distances) produces $avg(\eta_{Quality/Time}) = 3.36$ and $stdev(\eta_{Quality/time}) = 8.01$, which means that: i) *SemIndex*'s improvement in result quality is on average 3.36 times higher than its increase in query time w.r.t. the legacy *InvIndex*, and ii) the latter average cannot be generalized given the relatively high standard deviation of 8.01, reflecting the ratio's heavy fluctuation among queries (as shown in Table 10), which seems to depend on every query rather on the query category.

Recall that the above results and observations were obtained based on the feedback of ten test subjects (involved in the experiment as human judges), and need to be further investigated and generalized with a larger group of testers (using Amazon's Mechanical Turk for instance[1]). Note that we are currently conducting an extended comparative study comparing *SemIndex*' effectiveness with alternative semantic-aware retrieval

[1] Available at: https://www.mturk.com/

techniques, namely: query expansion and semantic disambiguation methods, where we can evaluate not only the relevance of query answers but also the ordering of the results[1].

8. Related Works

8.1. Keyword Search in Textual Databases

Traditionally, the DB and IR communities have targeted data search and processing mainly independently of each other. The DB community has largely focused on structured data providing sophisticated techniques for processing complex and exact queries, whereas the IR community has focused on searching unstructured data using various techniques for simple keyword-based search and ranking query results [6]. Yet in the past decade, there has been an increasing interest in integrating IR and DB search paradigms, namely: integrating keyword-based search in textual DBs to perform simple and approximate full-text DB querying [25, 58, 97].

Early approaches on keyword search queries for RDBs uses traditional IR scores (e.g., TF-IDF) to find ways to join tuples from different tables in order to answer a given keyword query [2, 15, 34]. The proposed search algorithms focus on enumeration of join networks called *candidate networks*, to connect relevant tuples by joining different relational tables. The result for a given query comes down to a sequence of candidate networks, each made of a set of tuples containing the query keywords in their text attributes, and connected through their primary-foreign key references, ranked based on candidate network size and coverage. The optimal candidate network problem has been shown to be NP-complete w.r.t. the number of relevant tables [34, 44], and various heuristic algorithms for the enumeration of top-*k* candidate networks have been proposed, e.g., [15, 34]. More recent methods on RDB full-text search in [55, 58] focus on more meaningful scoring functions and generation of top-*k* candidate networks of tuples, allowing to group and/or expand candidate networks based on certain weighting functions in order to produce more relevant results. The authors in [61] tackle the issue of keyword search on streams of relational data, whereas the approach in [96] introduces keyword search for RDBs with star-schemas found in OLAP applications. Other approaches introduced natural language interfaces providing alternate access to a RDB using text-to-SQL transformations [53, 72], or extracting structured information (e.g., identifying entities) from text (e.g., Web documents) and storing it in a DBMS to simplify querying [27, 28]. Keyword-based search for other data models, such as XML [1, 24] and RDF [13, 16] have also been studied.

Our work is complementary to most existing DB search algorithms in that our approach extends *syntactic* keyword-term matching: where only tuples containing exact occurrences of the query keywords are identified as results, toward *semantic* based keyword matching: where tuples containing terms which are lexically and semantically related to query terms are also identified as potential results, a functionality which - to our knowledge - remains unaddressed in most existing DB search algorithms.

8.2. Extending *Syntactic* Search toward *Semantic* Search

While DB approaches focused on integrating traditional (syntactic) keyword-based search functionality, many efforts have been deployed by the IR community to extend syntactic processing toward semantic full-text search using dedicated semantic indexing techniques, leading to so-called *concept-based* IR [7, 11, 12]. The latter is an alternative IR approach that aims to tackle the semantic relatedness problems described in this paper (cf. motivation scenarios and challenges in Section 1) by transforming both documents and queries into semantic representations, using semantic concepts in a reference knowledge base, instead of (or in addition to) keywords/terms, such as the retrieval process is undertaken in the concept space [12, 39]. Consequently, an adapted IR engine processes the semantically indexed documents and queries, so as to produce more meaningful results. Existing concept-based methods, e.g., [7, 11, 12, 39, 50, 51], can be characterized by three parameters: i) *Semantic indexing*: consists of the representation model the concepts are based on, as well as the underlying indexing technique used to access the concepts. It attempts to solve the problems of lexical matching by using conceptual indices instead of individual word indices for retrieval [50]; ii) *Mapping method*: the mechanism that maps the lexical terms with these semantic concepts. The mapping can be performed using manual mapping w.r.t. a handcrafted ontology such as WordNet [64] or Yago [42], or using machine learning [38] or graph matching techniques [12], though this would usually imply less accurate mappings, iii) *Usage in the retrieval process*: the stages in which the concepts are used in information retrieval. Concepts would be best used throughout the entire process, in both the indexing and retrieval stages [40]. A simpler but less accurate solution is to apply concept analysis in one stage only: at the query indexing stage, e.g., performing query expansion over the *bag of words* retrieval model [41] by adding to the query keywords their most related semantic concepts in the reference semantic source [5] (e.g., WordNet [64]) or words that co-occur with the query terms

[1] The traditional inverted index produces non-ranked results, which is the reason we did not compare result ordering in this study.

in a corpus (i.e., words that, on a probabilistic ground, are believed to belong to the same *semantic domain,* e.g., *France* and *Paris*; *car* and *driver*) [19], and then performing syntactic query/data matching/retrieval.

An alternative approach to handle semantic meaning is to apply automatic *word sense disambiguation* (WSD) to queries, during query execution time. Disambiguation methods usually use knowledge resources such as WordNet [56], and/or co-occurrence statistical data in a corpus [78] to find the possible senses of a word and map word occurrences to the correct sense. Semantic query analysis in information retrieval usually involves two steps: i) WSD to identify the user's intended meaning for query terms, and ii) semantic query representation/enhancement in order to alter the query so that it achieves better (precision and recall) results [5]. The disambiguated query terms are then used in query processing, so that only documents that match the correct sense are retrieved. Nonetheless, the performance of WSD-based approaches depends on the performance of the automated WSD process [35] which generally: i) is computationally complex requiring substantial execution time [68], ii) depends on the context of the query/data processed (e.g., surrounding terms) [22, 84, 98] which is not always sufficiently available (e.g., keyword queries on the Web are typically 2-to-3 words long [48]), and thus iii) do not guaranty correct results [35, 47] as incorrect disambiguation is likely to harm performance rather than merely not improve it [35].

Our study attempts to extend syntactic keyword search in textual DBs toward concept-based querying, with a special emphasis on semantic data indexing using a hybrid-inverted index: *SemIndex*. In the following, we briefly review the varieties and extensions of existing inverted indexes, and compare them with our proposal.

8.3. Inverted Indexes handling Data Semantics

Various efforts have been recently deployed to extend the inverted index toward handling data semantics. These can be organized in three main categories: i) including semantic knowledge into an inverted index, ii) including full-text information into the semantic knowledge base, and iii) building an integrated hybrid structure.

The first approach consists in adding additional entries in the index structure to designate semantic information. Here, the authors in [50] suggest extending the traditional (*term*, *docIDs*[]) inverted index toward a (*term*, *context*, *docIDs*[]) structure where contexts designates senses (synsets) extracted from WordNet, associated to each term in the index taking into account the statistical occurrences of concepts in Web document [11]. The authors however do not provide the details on how concepts are selected from WordNet and how they are associated to each term in the index. Another approach is introduced in [101], extending the inverted index structure by adding additional pointers linking each entry of the index to semantically related terms, (*term*, *docIDs*[], *relatedTerms*[]). Term links are identified by analyzing term occurrences in Web documents, based on Web document Page-Rank linkage analysis. The authors mention that they consider semantic relatedness between terms, yet they do not describe: how semantically related words are identified (what kinds of semantic relations and processing are used), nor how the index is actually built based on linked Web documents.

Another approach to semantic indexing is to add words as entities in the ontology [11, 92]. For instance, adding triples of the form *word occurs-in-context concept*, such that each word can be related to a certain ontological concept, when used in a certain context. Following such an approach: i) the number of triples would naturally explode, given that ii) query processing would require reaching over the entire left and right hand sides of this *occurs-in-context* index, which would be more time consuming [11] than reading on indexed entry such as with the inverted index. A possible optimization would be to split the relation into *word occurs-in context* and *concept occurs-in context*, yet the relations would remain huge and *concept occurs-in-context* always has to be processed entirely [11]. However, a related approach has been used to disambiguate WordNet glosses [92], and has been proven useful in enhancing WSD-based query expansion.

A third approach to semantic indexing consists in building an integrated hybrid structure: combining the powerful functionalities of inverted indexing with semantic processing capabilities. To our knowledge, one existing method in [11] has investigated this approach, introducing a joint index over ontologies and text. The authors consider two input lists: containing text postings (for words or occurrences), and lists containing data from ontological relations (for concept relations). The authors tailor their method toward incremental query construction with context-sensitive suggestions, and thus use inverted lists for *prefixes* instead of *terms*, in order to allow fast prefix suggestions for words to be used in building queries. They introduce the notion of *context lists* instead of usual inverted lists, where a prefix contains one index item per occurrence of a word starting with that prefix, adding an entry item for each occurrence of an ontological concept in the same context as one of these words, producing an integrated 4-tuples index structure (*prefix*, *terms*[]) ↔ (*term*, *context*, *concepts[]*).

The method in [11] seems arguably the most related to our study, with major differences in objectives and theoretical/technical contributions: the authors in [11] target semantic full-text search with special emphasis on incremental query construction and suggestion based on query term prefixes and result excerpts, whereas we target semantic search in textual DBs extending traditional DB-style (SQL based) querying capability toward semantic full-text search. Hence, while the authors in [11] focus on the IR aspects of indexing, keyword query construction, and query evaluation, we rather present a full-fledged textual DB solution, with structures and

tools designed for seamless storage and manipulation within a typical RDBMS, allowing to process different kinds of DB-style structure queries in a textual DB.

SemIndex brings full-text DB search from traditional syntactic data retrieval toward semantic concept-based retrieval, attempting to benefit from both worlds: allowing i) simple, ii) semantic-aware, and iii) ranked keyword search, while: iv) preserving sophisticated DB indexing and v) structured (SQL-based) querying.

SemIndex can also be extended/adapted toward so-called object (entity)-based retrieval, e.g., [13, 16, 73], where the main objective is to retrieve parts of a KB structure (e.g., sets of triples or components of triples in an RDF or OWL ontology describing Web resources) that best match a user query. In this context, *SemIndex* could be redesigned to integrate: a reference semantic network with a dedicated inverted index built on top of the target KB structure, using dedicated semantic and ontology matching techniques, e.g., [66, 79, 90], which we aim to investigate in a future study.

9. Conclusion

In this paper, we introduce a new semantic indexing approach called *SemIndex*, creating a hybrid structure using a tight coupling between two resources: a general purpose semantic network, and a standard inverted index defined on a collection of textual data, represented as dedicated graph structures. In addition to describing the logical graph-based design of *SemIndex*, we also provide its physical design using a standard commercial RDBMS, and develop the index construction process. We also provide an extended query model and related query processing algorithms, using *SemIndex*, to allow semantic-aware query processing. Our theoretical study and extensive experimental evaluation highlighted the following results: i) our index structure can be built in average linear time, and its size is of average linear space, w.r.t. the sizes of the input data and knowledge sources used, ii) query processing time is also linear in the size of the *SemIndex* structure, and varies linearly w.r.t. to the number query terms (keywords) as well as the link distance threshold designating the breadth of the *SemIndex* graph to be covered during querying, and iii) our approach allows both traditional (syntactic) queries (when using a minimum link distance threshold), as well as semantic-aware queries (when increasing link distance) with a significant and impressive increase in the number of neighboring nodes visited in the *SemIndex* graph as well as the number and quality of semantically-related returned results.

We are currently completing an extended experimental study to evaluate *SemIndex*'s properties in terms of i) genericity: to support different types of textual (structured, semi-structured, NoSQL) data collections[1], and different semantic knowledge sources (general purpose like: Roget's thesaurus [98], Yago [42], and Google [49], as well as domain specific: like ODP [59] for describing semantic relations between Web pages, FOAF [3] to identify relations between persons in social networks, and SSG [76] to describe visual and semantic relations between vector graphics)[2], ii) effectiveness: evaluating the interestingness of semantic-aware query answers considering different query answer weighting and ranking (result ordering) schemes, in comparison with IR-based indexing, query expansion, and semantic disambiguation methods, and iii) efficiency: to reduce the index's building and query processing costs, using multithreading and various index fragmentation and sub-graph mining techniques [26]. In the near future, we plan to investigate the different operations, algorithms, physical structures, as well as possible optimizations needed to update the index [20], based on changes in the textual data collection source as well as changes in the reference knowledge base source. Specifically, we plan to evaluate term context window size and its impact on the missing terms problem, and consequently on the *SemIndex* construction process and its usage in query processing. On the long run, we aim to extend *SemIndex* to handle more expressive semi-structured and linked data collections such as domain-specific RDF/OWL ontologies (e.g., [14, 77] describing health or biomedical data), building on recent solutions for semi-structured semantic analysis [22, 83, 84] and approximate structure mapping [86, 87] to achieve more sophisticated object (entity)-based retrieval capability [73].

Acknowledgements. This study is partly funded by the National Council for Scientific Research (CNRS-L) - Lebanon, project: NCSR_00695_01/09/15, LAU research fund: SOERC1516R003, and the Research Support Foundation of the State of Sao Paulo (FAPESP), project: MIVisBD_2017.

---

[1] While *SemIndex* is currently designed to index relational data in the form of key-value tuples, yet indexing NoSQL attribute-value stores (or semi-structured document stores) requires extending the index' logical and physical designs in order to handle a varying number of attributes describing every data object (as well as hierarchical relations connecting data objects).

[2] The knowledge base (KB) needs to be first represented following the general graph model adopted in *SemIndex* (cf. Definition 3), and then can be straightforwardly used in the *SemIndex* construction and querying processes. Here, dedicated semantic mediators or wrappers need to be designed to allow the mapping of every KB with *SemIndex*'s general graph model.

References

[1] Agarwal M.K., Ramamritham K., and Agarwal P., *Generic Keyword Search over XML Data.* International Conference on Extended DataBase Technology (EDBT'16), 2016. pp. 149-160.

[2] Agrawal S., Chakrabarti K., Chaudhuri s., et al., *Exploiting Web Search Engines to Search Structured Databases.* World Wide Web Conference (WWW'09), 2009. pp. 501-510.

[3] Aleman-Meza B., Nagarajan M., Ding L., et al., *Scalable Semantic Analytics on Social Networks for Addressing the Problem of Conflict of Interest Detection.* ACM Transaction on the Web (TWeb), 2008. 2(1):7.

[4] Algergawy A., Nayak R., and Saake G., *Element Similarity Measures in XML Schema Matching.* Elsevier Information Sciences, 2010. 180(24): 4975-4998

[5] Allan J. and H. Raghavan, *Using Part-of-Speech Patterns to Reduce Query Ambiguity.* In 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, 2002. pp. 307-314, Tampere, Finland: ACM Press, New York.

[6] Amer-Yahia S.; Case P.; Rolleke T., et al., *Report on the DB/IR Panel at SIGMOD 2005.* Sigmod Record, 2005. 34(4):71-74.

[7] Andreasen T., Bulskov H., Jensen P., et al., *Conceptual Indexing of Text Using Ontologies and Lexical Resources,* . Inter. Conf. on Flexible Query Answering Systems (FQAS'09) 2009. pp 323-332.

[8] Baeza-Yates R. and Ribeiro-Neto B., *Modern Information Retrieval: The Concepts and Technology behind Search.* ACM Press Books, Addison-Wesley Professional, 2nd Ed., 2011. p. 944.

[9] Banerjee S. and Pedersen T., *Extended Gloss Overlaps as a Measure of Semantic Relatedness.* International Joint Conference on Artificial Intelligence (IJCAI'03), 2003. p. 805-810.

[10] Bao Z., Yu Y., Shen J., et al., *A Query Refinement Framework for XML Keyword Search.* World Wide Web 2017. 20(6):1469-1505.

[11] Bast H. and Buchhold B., *An Index for Efficient Semantic Full-Text Search.* Proceedings of the 22nd ACM International Conference on information & knowledge Management (CIKM '13), 2013. pp. 369-378

[12] Baziz M., Boughanem M. and Traboulsi S., *A concept-based approach for indexing documents in IR.* INFORSID 2005, 2005. pp. 489-504, Grenoble, France.

[13] Bednar Peter, Sarnovsky M. and Demko V., *RDF vs. NoSQL databases for the Semantic Web applications.* IEEE 12th International Symposium on Applied Machine Intelligence and Informatics (SAMI'14), 2014. pp. 361-364.

[14] Belleau F., Nolin M.A., Tourigny N., et al., *Bio2RDF: Towards a mashup to build bioinformatics knowledge systems.* Journal of Biomedical Informatics, 2008. 41(5): 706-716.

[15] Bergamaschi S., Guerra F., Interlandi M., et al., *Combining User and Database Perspective for Solving Keyword Queries over Relational Databases.* Information Systems, 2016. 55: 1-19.

[16] Blanco R., Mika P. and Vigna S., *Effective and Efficient Entity Search in RDF data.* In International Semantic Web Conference (ISWC'11), 2011. pp. 83–97.

[17] Brin S. and Page L., *Reprint of: The Anatomy of a Large-Scale Hypertextual Web Search Engine.* Computer Networks, 2012. 56(18): 3825-3833.

[18] Budanitsky A. and Hirst G., *Evaluating WordNet-based Measures of Lexical Semantic Relatedness.* Computational Linguistics, 2006. 32(1): 13-47.

[19] Burton-Jones A.; Storey V.C.; Sugumaran V., and Purao S., *A Heuristic-Based Methodology for Semantic Augmentation of User Queries on the Web.* In Proceedings ot the International Conference on Conceptual Modeling (ER'03), 2003. pp. 476–489.

[20] Chakrabarti S., Pathak A. and Gupta M., *Index design and query processing for graph conductance search.* VLDB Journal, 2011. 20(3):445-470.

[21] Chandramouli K., Kliegr T. , Nemrava J., et al., *Query Refinement and user Relevance Feedback for contextualized image retrieval.* 5th International Conference on Visual Information Engineering (VIE), 2008. pp. 453 - 458.

[22] Charbel N., Tekli J., Chbeir R., et al., *Resolving XML Semantic Ambiguity.* International Conference on Extending Database Technology (EDBT'15), 2015. Brussels, Belgium, pp 277-288.

[23] Chbeir R., Luo Y., Tekli J., et al., *SemIndex: Semantic-Aware Inverted Index.* 18th East-European Conference on Advanced Databases and Information Systems (ADBIS'14), 2014. pp. 290-307.

[24] Chen L.J. and Papakonstantinou Y., *Supporting top-K keyword Search in XML Databases.* International Conference on Data Engineering (ICDE'10), 2010. pp. 689-700.

[25] Chen Yi, Wang W., Liu Z., et al., *Keyword search on structured and semi-structured data.* Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, 2009. pp. 1005-1010.

[26] Cheng J., Ke K., Wai-Chee Fu A.., et al., *Fast graph query processing with a low-cost index.* VLDB Journal, 2011, 20(4): 521-539.

[27] Cheng T., Yan X. and Chang K. C., *EntityRank: searching entities directly and holistically.* Proceedings of the 33rd international conference on Very Large Data Bases (VLDB'07), 2007. pp. 387-398.

[28] Chu E., Baid A., Chen T., et al., *A relational approach to incrementally extracting and querying structure in unstructured data.* Proceedings of the 33rd international conference on Very Large Data Bases (VLDB '07), 2007. pp. 1045-1056

[29] Cimiano P.; Handschuh S. and Staab S., *Towards the Self-Annotating Web.* In Proceedings of the International World Wide Web Conference (WWW'04), 2004. pp. 462-471.

[30] Cormen T.H., Leiserson C.E., Rivest R.L., et al., *Introduction to Algorithms (3rd Ed.).* MIT Press and McGraw-Hill. , 2009.

[31] Das S., e.a., *Making unstructured data sparql using semantic indexing in oracle database.* In Proceedings of 29th IEEE ICDE Conf., 2012. pp. 1405–1416.

[32] Davies M., *The Corpus of Contemporary American English as the first reliable monitor corpus of English.* Literary & Linguistic Computing, 2010. 25(4): 447-464.

[33] de Lima E.F. and Pedersen J.O., *Phrase Recognition and Expansion for Short, Precision biased Queries based on a Query Log.* In Proc. of the 22nd Annual Inter.ACM SIGIR Conf.on Research and Development in Information Retrieval, 1999, pp. 145-152.

[34] Ding B., Xu Yu J., Wang S., et al., *Finding top-k min-cost connected trees in databases.* Proceedings of the International Conference on Data Engineering (ICDE'07), 2007.

[35] Egozi O., Markovitch S. and Gabrilovich E., *Concept-Based Information Retrieval Using Explicit Semantic Analysis.* ACM Transactions on Information Systems 2011, 29(2):8.

[36] Francis W. N. and Kucera H., *Frequency Analysis of English Usage.* Houghton Mifflin, Boston, 1982.

[37] Gao X. and Qiu J., *Supporting Queries and Analyses of Large-Scale Social Media Data with Customizable and Scalable Indexing Techniques over NoSQL Databases.* IEEE/ACM Inter. Symposium on Cluster Computing & the Grid (CCGRID'14), 2014, 587-590.

[38] Gauch S., Ravindran D. and Chandramouli A., *KeyConcept: Conceptual Search and Pruning Exploiting Concept Relationships.* Journal of Intelligent Systems, 2010. 19(3): 265-288

[39] Giunchiglia F., Kharkevich U. and Zaihrayeu I., *Concept Search.* In ESWC - Semantics and Big Data, 2009. pp. 429–444.

[40] Gonzalo J., Verdejo F. and Chugur I., *Using Eurowordnet in a Concept-Based Approach to Cross-Language Text Retrieval.* Applied Artificial Intelligence 1999, 13(7): 647-678.

[41] Grootjen F. and Van Der Weide T.P., *Conceptual query expansion.* Data Knowledge Engineering, 2006. 56:174–193.

[42] Hoffart J., Suchanek F.M., Berberich K., et al., *YAGO2: A spatially and temporally enhanced knowledge base from Wikipedia.* Artif. Intell., 2013, 194: 28-61.

[43] Hopfield J. and Tank D., *Neural Computation of Decisions in Optimization Problems.* . Biological Cybernetics, 1985, 52(3):52–141.

[44] Hristidis V. and Papakonstantinou Y., *DISCOVER: Keyword search in relational databases.* Proceedings of the International Conference on Very Large Databases (VLDB), 2002.

[45] Hudec M., *An approach to fuzzy database querying, analysis and realization.* Comput. Sci. Inf. Syst., 2009, 6(2): 127-140.

[46] International Organization for Standardization, *ISO/IEC 14977:1996 Extended BNF Notation.* Avaialble from: http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf, 1996.

[47] Kamvar M. and Baluja S., *A Large Scale Study of Wireless Search Behavior: Google Mobile Search.* In Proceedings of the SIGCHI Conference on Computer Human Interaction, 2006. pp. 701–709, Montreal, Canada.

[48] Kathuria A., Jansen B.J., Hafernik C.T., et al., *Classifying the User Intent of Web Queries using K-means Clustering.* Internet Research, 2010. 20(5): 563-581.

[49] Klapaftis I. and Manandhar S., *Evaluating Word Sense Induction and Disamiguation Methods.* Language Resourses and Evaluation, 2013. 47(3):579-605.

[50] Kumar S., Rana R.K. and Singh P., *Ontology based Semantic Indexing Approach for Information Retrieval System.* International Journal of Computer Applications, 2012. Volume 49– No.12.

[51] L'Hadj L.S., Boughanem M. and Amrouche K., *Enhancing Information Retrieval through Concept-based Language Modeling and Semantic Smoothing.* Journal of the Association for Information Science and Technology (JASIST), 2016. 67(12): 2909-2927.

[52] Lester N., Zobel J. and Williams H., *Efficient online index maintenance for contiguous inverted lists.* Information Processing and Management, 2006. 42(4):916- 933.

[53] Li F. and J. H.V., *Constructing an Interactive Natural Language Interface for Relational Databases.* Proceedings of the VLDB Endowment, 2014. pp. 73-84.

[54] Li Y., Yang H. and Jagadish H.V., *Term Disambiguation in Natural Language Query for XML.* In Proceedings of the International Conference on Flexible Query Answering Systems (FQAS), 2006. LNAI 4027, pp. 133–146.

[55] Liu F., Yu C., Meng W., et al., *Effective keyword search in relational databases.* Proceedings of the 2006 ACM SIGMOD international conference on Management of data, 2006. pp. 563-574

[56] Liu Y., Scheuermann P., Li X., et al., *Using WordNet to Disambiguate Word Senses for Text Classification.* International Conference on Computational Science (ICCS'07), 2007. pp 781-789.

[57] Lucio F. D. Santos, Willian D. Oliveira, Mônica Ribeiro Porto Ferreira, et al., *Evaluating the Diversification of Similarity Query Results.* Journal of Information and Data Management (JIDM) 2013. 4(3): 188-203.

[58] Luo Y., Lin X., Wang W., et al., *Spark: top-k keyword query in relational databases.* Proceedings of the 2007 ACM International Conference on Management of Data (SIGMOD-07), 2007. pp. 115-126.

[59] Maguitman A., Menczer F., Roinestad H., et al., *Algorithmic Detection of Semantic Similarity.* Proceedings of the International Conference on the World Wide Web (WWW), 2005. pp. 107-116.

[60] Mahapatra A.K. and Biswas S., *Inverted Index: Types and techniques.* International Journal of Computer science Issues,, 2011. 8(4):1.

[61] Markowetz A. and P.D. Yang Y., *Keyword search on relational data streams.* Proceedings of the International Conference on Management of Data (SIGMOD'07), 2007. pp. 605–616.

[62] Martinenghi D. and Torlone R., *Taxonomy-based relaxation of query answering in relational databases.* VLDB Journal, 2014. 23(5):747-769.

[63] McGill M., *Introduction to Modern Information Retrieval.* 1983. McGraw-Hill, New York.

[64] Miller G.A. and Fellbaum C., *WordNet Then and Now.* Language Resources and Evaluation, 2007. 41(2): 209-214.

[65] Miller S., Bobrow R., Ingria R., et al., *Hidden understanding models of natural language.* Proceedings of the 32nd annual meeting on Association for Computational Linguistics, Stroudsburg, PA, USA, 1994. pp. 25–32.

[66] Ming M., Yefei P. and Michael S., *A Harmony Based Adaptive Ontology Mapping Approach.* In Proceedings of the International Conference on Semantic Web and Web Services (SWWS'08), 2008. pp. 336-342.

[67] Mishra C. and Koudas N., *Interactive Query Refinement* International Conference on Extending Database Technology (EDBT'09), 2009. pp. 862-873.

[68] Navigli R., *Word Sense Disambiguation: a Survey.* ACM Computing Surveys, 2009. 41(2):1–69.

[69] Navigli R. and Lapata M., *An Experimental Study of Graph Connectivity for Unsupervised Word Sense Disambiguation.* IEEE Trans. on Pattern Analysis and Machine Intelligence 2010. 32(4): 678-692.

[70] Navigli R. and Crisafulli G., *Inducing Word Senses to Improve Web Search Result Clustering.* In Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing, 2010. pp. 116–126, MIT, USA.

[71] Nayak R., *Fast and effective clustering of XML data using structural information.* Knowledge and Information Systems, 2008. 14 (2): 197-215.

[72] Nihalani N., Silakari S. and Motwani M., *Natural language Interface for Database: A Brief review.* International Journal of Computer Science Issues, 2011. 8(2):600-608.

[73] Pound J., Mika P. and Z. H., *Ad-hoc Object Retrieval in the Web of Data.* International World Wide Web Conference (WWW'10), 2010. pp. 771-780.

[74] Richardson R. and Smeaton A., *Using WordNet in a Knowledge-based approach to information retrieval.* Proceedings of the BCS-IRSG Colloquium on Information Retrieval, 1995.

[75] Rychly P. and Kilgarriff A., *An Efficient Algorithm for Building a Distributional Thesaurus (and other Sketch Engine developments).* ACL 2007, 2007. Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics (ACL)(pp. 41-44).

[76] Salameh K., Tekli J. and Chbeir R., *SVG-to-RDF Image Semantization.* 7th International SISAP Conference, 2014. pp. 214-228.

[77] Samwald M., Jentzsch A., Bouton C., et al., *Linked open drug data for pharmaceutical research and development.* Journal of Cheminformatics, 2011. 3:19.

[78] Schuetze H. and Pedersen J. O., *Information Retrieval based on Word Senses.* In Proceedings of the 4th Annual Symposium on Document Analysis and Information Retrieval. , 1995. pp. 161–175.

[79] Shvaiko P. and Euzenat J., *Ten challenges for ontology matching.* Proceedings of the OTM 2008 Confederated International Conferences, 2008. pp. 1164–1182.

[80] Silva Y.N., Aref W.G., Larson P.A., et al., *Similarity queries: their conceptual evaluation, transformations, and processing.* VLDB Journal, 2013. 22(3):395-420.
[81] Sinh Hoa Nguyen, Wojciech Świeboda and G. Jaśkiewicz, *Semantic Evaluation of Search Result Clustering Methods.* Intelligent Tools for Building a Scientific Information Platform, Studies in Computational Intelligence Volume 467,
, 2013. 467(393-414).
[82] Spink A., Wolfram D., Jansen M., et al., *Searching the Web: The Public and Their Queries.* Journal of the American Society for Information Science, 2001. 52(3):226–234.
[83] Tekli J., *An Overview on XML Semantic Disambiguation from Unstructured Text to Semi-Structured Data: Background, Applications, and Ongoing Challenges.* IEEE Trans. on Knowledge and Data Engineering (IEEE TKDE), 2016. 28(6): 1383-1407.
[84] Tekli J., Charbel N. and Chbeir R., *Building Semantic Trees from XML Documents.* Elsevier Journal of Web Semantics (JWS): Science, Services and Agents on the World Wide Web, 2016. 37–38:1–24.
[85] Tekli J., Chbeir R., Luo Y., et al., *SemIndex: Semantic-Aware Inverted Index - Technical Report.* Available at http://sigappfr.acm.org/Projects/SemIndex/, 2018, 2018.
[86] Tekli J., Chbeir R., Traina A.J.M., et al., *Approximate XML Structure Validation based on Document-Grammar Tree Similarity.* Elsevier Information Sciences, 2015. 295:258-302.
[87] Tekli J., Chbeir R. and Yetongnon K., *A Novel XML Structure Comparison Framework based on Sub-tree Commonalities and Label Semantics.* Elsevier Journal of Web Semantics (JWS): Science, Services and Agents on the World Wide Web, 2012. 11: 14-40.
[88] Tekli J., Chbeir R. and Yétongnon K., *An Overview of XML Similarity: Background, Current Trends and Future Directions.* Elsevier Computer Science Review, 2009. 3(3):151-173.
[89] Tekli J., Chbeir R. and Yétongnon K., *Minimizing User Effort in XML Grammar Matching.* Elsevier Information Sciences Journal, 2012. 210:1-40.
[90] Umer Q. and Mundy D., *Semantically Intelligent Semi-Automated Ontology Integration.* Proceedings of the World Congress on Engineering, 2012. London, U.K. .
[91] Vasilescu F., Langlais P. and Lapalme G., *Evaluating Variants of the Lesk Approach for Disambiguating Words.* Language Resources and Evaluation (LREC'04), 2004. pp. 633-636.
[92] Velardi P., Faralli S. and Navigli R., *OntoLearn Reloaded: A Graph-Based Algorithm for Taxonomy Induction.* Computational Linguistics, 2013. 39(3): 665-707.
[93] von der Weth C. and Datta A., *Multiterm Keyword Search in NoSQL Systems.* IEEE Internet Computing, 2012. 16(1):34-42
[94] Weeds J., e.a., *Characterizing Measures of Lexical Distributional Similarity.* In Proceedings of 20th Int. Conf. on Computational Linguistics (COLING '04), 2004. Article No. 1015.
[95] Wen H., Huang G.S. and L. Z., *Clustering Web Search Results using Semantic Information.* International Conference on Machine Learning and Cybernetics, 2009. 3(1504 - 1509).
[96] Wu P., Sismanis Y. and Reinwald B., *Towards Keyword-Driven Analytical Processing.* Proceedings of the International Conference on Management of Data (SIGMOD'07), 2007. pp. 617–628.
[97] Xu Y., Guan J. and Ishikawa Y., *Scalable Top-k Keyword Search in Relational Databases.* 17th International Conference on Database Systems for Advanced Applications (DASFAA'12), 2012. Volume 7239 of the series LNCS, pp. 65-80.
[98] Yaworsky D., *Word-Sense Disambiguation Using Statistical Models of Roget's Categories Trained on Large Corpora.* Proceedings of the International Conference on Computational Linguistics (Coling), 1992. Vol 2, pp. 454-460. Nantes.
[99] Zhang C., Naughton J., DeWitt D., et al., *On Supporting Containment Queries in Relational Database Management systems.* SIGMOD Record, 2001. 30(2), 425–436.
[100] Zhang P., *A Study on Database Fuzzy Query Method in SQL.* International Conference on Advances in Engineering, 2011. Vol. 24, pp. 340-344.
[101] Zhong S., Shang M. and Deng Z., *A Design of the Inverted Index Based on Web Document Comprehending.* Journal of Computers, 2011. 6(4):664-670.

Appendix: *SemIndex* Weighting Scheme

We propose a set of weighting functions to assign weight scores to *SemIndex* entries, including: *index nodes*, *index edges, data nodes*, and *data edges*. The weighting functions are used to select and rank semantically relevant results w.r.t. the user's query (cf. *SemIndex* query processing in Section 5). Other weight functions could be later added to cater to the index designer's needs.

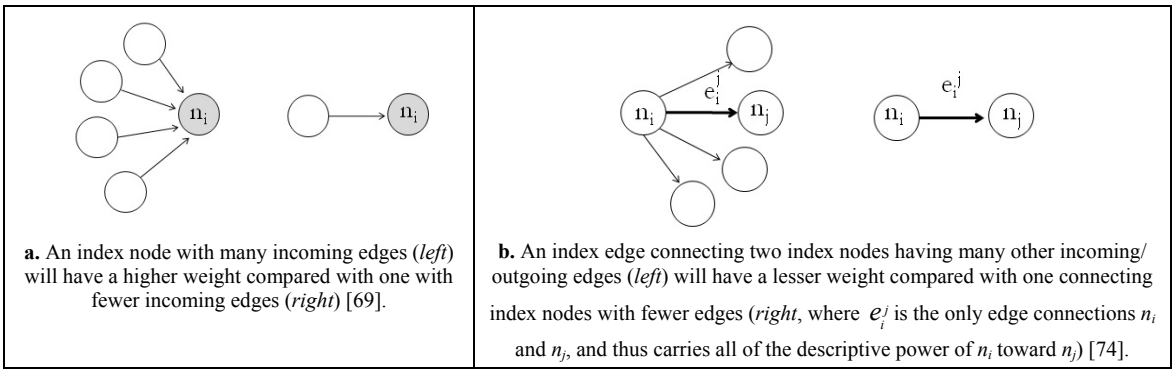
1. *Index Node* Weight

Considering an index node $n_i \in \tilde{G}_{SI}.V_i$, the weight of n_i denoted as $W_{IndexNote}(n_i)$, is evaluated as a node *degree centrality* score [69], computed as the node's in-degree (i.e., number of nodes connected with the target index node) over the maximum node in-degree in $\tilde{G}_{SI}$, according to the below formula:

$$W_{IndexNode}(n_i) = \frac{in\text{-}degree(n_i)}{\underset{\forall v_j \in \tilde{G}_{SI}.V_i}{Max}\ (in\text{-}degree(n_j))} \quad \in [0,1] \qquad \textbf{(5)}$$

Rationale: An index node is more important if it receives many links from other indexing nodes[1] (cf. Fig. 30.a).

[1] A future extension would be to consider *eigenvector centrality*, where node weights are normalized based on centrality scores of connected nodes [69].



a. An index node with many incoming edges (*left*) will have a higher weight compared with one with fewer incoming edges (*right*) [69].

b. An index edge connecting two index nodes having many other incoming/ outgoing edges (*left*) will have a lesser weight compared with one connecting index nodes with fewer edges (*right*, where e_i^j is the only edge connections n_i and n_j, and thus carries all of the descriptive power of n_i toward n_j) [74].

Fig. 30. Visual depictions of *index node* and *index edge* weight evaluation.

2. *Index Edge* Weight

The weight of an index edge $e_i^j \in \widetilde{G}_{SI}.E_i$ outgoing from index node n_i and incoming into index node n_j is determined by the out-degree of n_i, considering the corresponding edge label (e.g., semantic relationship, e.g., *hypernymy*, *meronymy*, etc.) [74], according to the below formula:

$$W_{IndexEdge}(e_i^j) = \frac{1}{out\text{-}degree_{Label}(n_i)} \in]0, 1] \quad \textbf{(6)}$$

Rationale: An index edge designates a stronger connection between two index nodes when it carries most of the descriptive power from the source node to the destination node, such that the source node has few other out-going connections (if any, cf. Fig. 30.b) [1].

3. *Data Node* Weight

The weight of a data node $n_d \in \widetilde{G}_{SI}.V_d$ in the *SemIndex* graph is defined as:

$$W_{DataNode}(n_d) = \frac{in\text{-}degree(n_d)}{\underset{\forall n_q \in \widetilde{G}_{SI}.V_d}{\text{Max}} (in\text{-}degree(n_q))} \in [0, 1] \quad \textbf{(7)}$$

where *in-degree*(n_d) designates the number of foreign key/primary key data links (joins) outgoing from all data nodes (tuples) where the foreign keys reside, toward data node (tuple) n_d where the primary key resides.

Rationale: Similarly to index node weight, we consider that a data node is more important (its weight will increase) when it received many links from other data nodes (cf. Fig. 31).

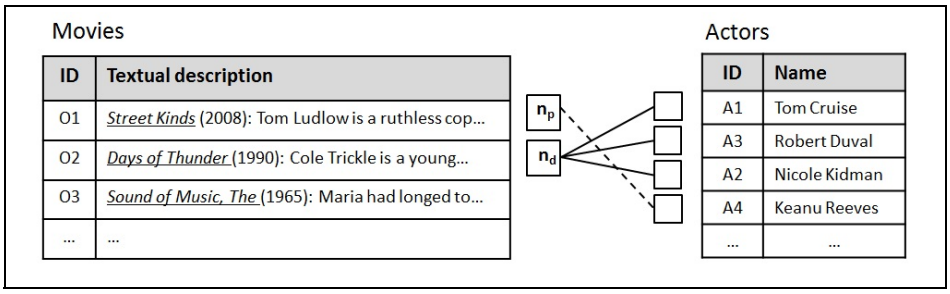


Fig. 31. Visual depiction of *data node* weight evaluation, where data node n_d which has a larger number of foreign key/primary key connections (high in-degree) will have a higher weight compared with n_p.

[1] A future extension would be to assign higher/lower weights to every semantic relation (e.g., *hypernymy* could be considered as a stronger semantic relation compared with *related-to*).

4. *Data Edge* Weight

Given a data edge $e_i^d \in \tilde{G}_{SI}.E_d$ connecting an index node n_i with a data node n_d (e.g., data edge connecting index node T_1 with data node O_2 since the term "*car*" occurs in the textual description of O_2, likewise for T_1-O_2, T_4-O_1,…, T_{12}-O_1, in Fig. 6), we compute the weight of e_i^d as an adapted *tf* (term frequency) score where *tf* underlines the frequency (number of occurrences) of the index node string literal within a given data node, connected via the data edge in question. Hence, given a data edge e_i^d incoming from index node n_i toward data node n_d, where $n_i.l$ denotes the string value of n_i, we define:

$$\mathrm{W}_{\mathrm{DataEdge}}(e_i^d) = \frac{NbOcc(n_i.l)}{\underset{e_i^d \in \tilde{G}_{SI}.E_d}{Max}(\ NbOcc(n_j.l))} \in [0, 1] \quad \textbf{(8)}$$

where $NbOcc(n_j.l)$ designates the number of occurrences of a term $n_i.l$ in n_d's textual description, normalized w.r.t. the maximum number occurrences of any index node string literal $n_j.l$ within the target data node n_d.

Rationale: Following the IR logic of term frequency [8], a data edge is more important if it connects an index term with a data node where the index term occurs many times in the data node's string value (e.g., index term T_1 ("*car*") occurs many times in data object O_2 ("*Days of Thunder*")'s full textual description, resulting in a high $\mathrm{W}_{\mathrm{DataEdge}}(e_{T_1}^{O_2})$).

Note that the user (admin) can also adapt weight functions by tuning their respective weight parameters, activating/de-activating certain functions based on her needs.