SemIndex+: a Semantic Indexing Scheme for Structured, Unstructured, and Partly Structured Data

E.C.E. Department
Lebanese American Univ.
36, Byblos, Lebanon
joe.tekli@lau.edu.lb

Univ. of Pau & Pays Adour
ES2/UPPA
64600 Anglet, France
richard.chbeir@univ-pau.fr

ICMC Institute
University of Sao Paulo,
13566 Sao Carlos, Brazil
agha@icmc.usp.br

ICMC Institute
University of Sao Paulo,
13566 Sao Carlos, Brazil
caetano@icmc.usp.br

Abstract. While Information Retrieval (IR) systems have gained success in Web-style search engines in the past two decades, nonetheless, the DataBase (DB) paradigm remains prevalent in handling data in enterprise environments and digital libraries, and is gaining even more importance in the Semantic Web with the increasing need to handle partly structured (NoSQL) data. This paper describes SemIndex+, a semantic-aware indexing and querying framework that allows semantic search, result selection, and result ranking of structured (relational DB-style), unstructured (IR-style), and partly structured (NoSQL) data. Various weighting functions and a parallelized search algorithm have been developed for that purpose and are presented here. We provide a general keyword query model allowing the user to choose the results’ semantic coverage and expressiveness based on her needs. Different from alternative solutions involving query relaxation, query refinement, or query disambiguation, our approach incorporates semantics at the most basic data indexing level: providing more opportunities toward speedups and semantic coverage. An extensive experimental evaluation, comparing SemIndex+ with alternative methods, highlights our approach’s flexibility and effectiveness, which in turn impact efficiency (requiring less or more time following the user specified index and query semantic coverages).

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

1. Introduction

Full-text search techniques originally developed in Information Retrieval (IR) systems, and more recently imbedded in DataBase (DB) systems, aim at providing the most relevant textual data objects (e.g., documents in IR, or tuples in a DB) to a user query consisting of a set of keywords [9, 10, 30]. The brute force approach is to sequentially scan the data objects in the collection, in order to search for terms matching those of the user query, which could be extremely inefficient for big data collections. Most IR/DB systems use some sort of indexing to speed up the search process. Currently, the foremost indexing structure used with full-text search is the inverted index model [56, 61, 79]. Inverted indexes associate each term in the text with a list of pointers to the data objects 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 query, identifying as candidate results all data objects that contain the query terms in just one scan of the index [9]. The inverted index is widely adopted for full-text indexing of large textual collections [9], and is supported by many DBMSs1 for storing and handling structured data (cf. Table 1) [4, 9], unstructured data (cf. Table 2) [30, 60], as well as partly structured2 data (cf. Table 3) [40, 86]. Yet, as full-text search systems became available to non-expert users, keyword queries become noisier, where non-experts have poor or no knowledge about the data being searched. As a result, they tend to formulate query keywords which are often syntactically different from those used in indexing relevant documents in the DB [39], thus returning non-relevant results or completely missing relevant ones.

1.1. Motivation Examples

To illustrate this, consider a textual data collection Δ from a movie database, where each movie in Δ is identified with an ID and is described with some text, including the movie title, year, plot, genre, and info. The data collection can be presented in different forms: i) structured, following the traditional relational (SQL) DB model, noted Δstruc as shown

---

* Corresponding author. This study was partly conducted during the author’s Fulbright Visiting Scholar research mission conducted in the Computer and Information Science Department, University of Michigan, Dearborn, USA

1 Database Management Systems.

2 We use the expression partly structured, to distinguish basic NoSQL data consisting of attribute-value items, from hierarchically structured data such as XML and RDF-based serializations, which are commonly referred to as semi-structured data [81]. The latter are out of the scope of this paper, and will be addressed in a dedicated study.
in Table 1, ii) unstructured, made of free text, noted \( \Delta_{\text{free}} \) as shown in Table 2, or iii) partly structured, following the basic attribute-value (NoSQL) DB model, noted \( \Delta_{\text{part}} \) as shown in Table 3.

### Table 1. Sample movies data collection extracted from IMBD\(^1\), provided in the form of structured text, noted \( \Delta_{\text{struct}} \).

<table>
<thead>
<tr>
<th>ID</th>
<th>title</th>
<th>year</th>
<th>plot</th>
<th>genre</th>
<th>info</th>
</tr>
</thead>
<tbody>
<tr>
<td>O₁</td>
<td>When a Stranger calls</td>
<td>2006</td>
<td>A young high school student babysits for a very rich family. Driving by the house, a strange car...</td>
<td>Horror, Thriller</td>
<td>When the Mandrakises were about to leave...</td>
</tr>
<tr>
<td>O₂</td>
<td>Days of Thunder</td>
<td>1990</td>
<td>Cole Trickle is a young racer from California with years of experience in open-wheel racing winning championships in Sprint car racing...</td>
<td>Action, Drama</td>
<td>In the clip from the race at Rockingham...</td>
</tr>
<tr>
<td>O₃</td>
<td>The Sound of Music</td>
<td>1965</td>
<td>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...</td>
<td>Drama, Family</td>
<td>The 1996 video fits the movie onto one VHS...</td>
</tr>
</tbody>
</table>

### Table 2. Sample data collection from IMBD (cf. Table 1), provided in the form of unstructured free text, noted \( \Delta_{\text{free}} \).

<table>
<thead>
<tr>
<th>ID</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>O₁</td>
<td>When a Stranger Calls (2006), Horror, Thriller: A young high school student babysits for a very rich family. Driving by the house, a strange car...</td>
</tr>
<tr>
<td>O₂</td>
<td>Days of Thunder (1990), Action, Drama: Cole Trickle is a young racer from California with years of experience in open-wheel racing winning championships in Sprint car racing...</td>
</tr>
<tr>
<td>O₃</td>
<td>Sound of Music, The (1965), Drama, Family: 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...</td>
</tr>
</tbody>
</table>

### Table 3. Sample data collection from IMBD (cf. Table 1), provided in the form of partly structured text, noted \( \Delta_{\text{part}} \).

| ID: O₁ | title: When a Stranger Calls | year: 2006 | plot: A young high school student babysits for a very rich family. Driving by the house, a strange car... | genre: Horror, Thriller | info: When the Mandrakises were about to leave... |
| ID: O₂ | description: Days of Thunder (1990), Action, Drama: Cole Trickle is a young racer from California with years of experience in open-wheel racing winning championships in Sprint car racing... |
| ID: O₃ | title: The Sound of Music | plot: 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... | genre: Drama, Family |

Consider keyword-based queries \( q₁, q₂, \) and \( q₃ \) applied respectively on each of the above data collections, formulated as standard DB selection (\( \sigma \)) containment (\( \epsilon \)) queries (formally described in Section 5): \( q₁ = \sigma_{\text{title}} \epsilon \{\text{sound}, \text{of}\}, \text{music}\} \Delta_{\text{struct}} \), \( q₂ = \sigma_{\text{description}} \epsilon \{\text{open-wheel}, \text{racing}\} \Delta_{\text{free}} \) and \( q₃ = \sigma_{\text{genre}} \epsilon \{\text{thunder}\} \Delta_{\text{part}} \). The search result for query \( q₁ \) is movie \( O₁ \) (from \( \Delta_{\text{struct}} \)) which title attribute contains occurrences of each of the query’s terms. Similarly, search results for \( q₂ \) and \( q₃ \) are: \( O₂ \) (from \( \Delta_{\text{free}} \)) and \( O₃ \) (from \( \Delta_{\text{part}} \)) respectively. Yet, if the user wants to search for a particular movie but cannot recall its exact title, plot, or genre descriptions, 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’, ‘auto rallying’, or ‘suspense’. Such terms might not exactly match those used to describe (and index) the movie objects (which is the case in our examples), and thus will miss movies \( O₁, O₂, \) and \( O₃ \) as relevant results. As a matter of fact, there are typically many ways to specify a given concept: as textual descriptions may involve terms with multiple meanings (homonymy, e.g., term “sound” could mean a particular auditory effect or a narrow channel in the sea paper sheet according to a general purpose knowledge base such as WordNet [66]), terms implied by other terms (metonymy, e.g., term “thunder” implies “lightning”, and “nun” implies “religion”), several terms having the same meaning (synonymy, e.g., terms “thunder”, “roar”, and “boom” all refer to the same meaning: a deep prolonged loud noise), or terms related by some semantic relationships (e.g., hypernymy (isA), holonymy (partOf), such as thunder-isA-noise, or engine-partOf-car). Therefore, the terms used as keywords in a user's query might literally match terms in irrelevant movie objects (decreasing search precision\(^2\)), or might completely miss relevant movies where no exact query term matches are identified (decreasing search recall\(^3\)). In addition, the movie

---

1 Internet Movie Database (http://www.imdb.com/).
2 Precision highlights the percentage of correctly selected results, whereas recall designates the percentage of wrongly missed results (i.e., results that should have been selected, cf. Section 7.6).
objects might not be extensively described or well-tagged in the DB, or might not be described using the same attributes; namely in a partly structured DB (cf. Table 3).

This shows that the standard inverted index cannot deal with these cases. 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 of traditional containment queries, encouraged by (non-expert) user demands. Most existing works in this area (cf. Background in Section 2) have incorporated semantic knowledge at the query processing level, to: i) pre-process queries using query relaxation and rewriting [17, 27, 64], ii) disambiguate queries using semantic disambiguation and entity recognition techniques [17, 58, 71], and/or iii) post-process query results using query refinement and results re-ranking [71, 80, 88]. Yet, various challenges remain unsolved, namely: i) time latencies when involving query pre-processing and post-processing [27, 64], ii) reduced quality of query relaxation/rewriting and query disambiguation results: requiring context information (e.g., expanded queries, user profiles, or query logs) which are not always available [19, 32], and iii) limited user involvement, where the user is usually constrained to providing feedback and/or performing query refinement only after the first round of results has been provided by the system [21, 67].

1.2. Proposal Overview

In this work, we adopt another alternative: building a semantic-aware inverted index called SemIndex+, integrating textual information with domain knowledge (not only at the querying level, but rather) at the most basic data indexing level, in order to support semantic-aware querying and answer most challenges identified above. Fig. 1 depicts the overall framework of our approach and its main components. Briefly, SemIndex+ maps two data resources, namely a textual data collection (represented as an inverted index), and a semantic knowledge base (represented as a semantic network) into a single and tightly coupled semantic-aware data structure. This is performed offline – prior to online query execution, providing more opportunities toward both speed-ups and semantic-based filtering, thus minimizing the need for query pre- and post-processing.

An initial solution design titled SemIndex was given in [23] aimed at indexing unstructured (free-text) data only. This paper introduces SemIndex+, a new framework allowing to index and search unstructured, structured (relational), and partly structured (NoSQL) textual data. SemIndex+’s main additions to the previous framework are highlighted in Fig. 1. At the indexer level, we add: i) an extension of SemIndex’s logical design to handle varying multi-attribute datasets (using attribute sensitive indexers), ii) a dedicated algorithm to handle terms with missing semantic connections (which we designate as missing terms), and iii) a mathematical model for weighting SemIndex+ entries (i.e., the graph’s nodes and edges). At the query processing level, we develop: iii) a parallelized (multithreaded) query processing algorithm, coupled with iv) a dedicated relevance scoring measure, required in the query evaluation process to retrieve and rank relevant query answers. In addition, we add: iv) a detailed complexity analysis covering the new index construction and querying algorithms, and v) an extensive experimental study comparing SemIndex+’s effectiveness and efficiency with various generic approaches (including inverted index search, query relaxation, query disambiguation, and query refinement). Results highlight SemIndex+’s flexibility (involving the user in the whole process: during initial index building, query writing, and then query refinement) and effectiveness (producing significantly more semantically relevant results compared with existing solutions) which in turn affect its efficiency (requiring less or more processing time following user-specified index and query semantic coverages).

The rest of this paper is organized as follows. Section 2 briefly reviews related works. Section 3 describes SemIndex+’s input resources. Section 4 develops SemIndex+’s design, including the index construction process and weighting functions. Section 5 develops SemIndex+’s query model and query processing algorithm. The computational complexity of SemIndex+ construction and querying algorithms is provided in Section 6. Experimental results are presented in Section 7. Section 8 concludes the paper with ongoing and future directions.

![Fig. 1. Overall architecture of the SemIndex+ framework (added components are highlighted in red).](Image)
2. Related Works

2.1. Keyword Search in Textual Databases

Early approaches on keyword search queries for RDBs use traditional IR scores (e.g., TF-IDF) to find ways to join tuples from different tables in order to answer a given keyword query [4, 35, 47]. 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. Recent methods on RDB full-text search in [14, 60] 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 [63] tackle the issue of keyword search on streams of relational data, whereas the approach in [89] 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 [57, 73], or extracting structured information (e.g., identifying entities) from text (e.g., Web documents) and storing it in a DBMS to simplify querying [25, 26]. Few recent approaches have addressed keyword-search in NoSQL DBs such as HBase [40, 49, 86], using dedicated (row or column) table schemas coupled with (horizontal or vertical) index partitioning, to support parallel index storage and search [40], as well as multi-term indexing and incremental query result fetching [86]. Keyword-based search for other data models, such as XML [2, 24] and RDF [13, 15] have also been studied.

Discussion: 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.

2.2. 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 the so-called concept-based (or knowledge-based) IR [8, 11, 55]. The latter is an alternative IR approach that aims to tackle the semantic relatedness problem by transforming both documents and queries into semantic representations, using semantic concepts in a reference knowledge base (instead of syntactic keywords/terms) such that the retrieval process is undertaken in the concept space [12, 42, 55]. Existing concept-based methods, e.g., [8, 11, 12, 42, 54, 55], 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 [54]; 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 [66] or Yago [45], or using machine learning [41] 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 [12]. Yet, most existing solutions apply concept analysis in one stage only, performing so-called query relaxation/refinement or query disambiguation over the bag of words retrieval model [44], to reduce processing time.

2.2.1. Query Relaxation and Refinement

Traditional query relaxation and refinement methods (cf. surveys in [19, 77]), rely on corpus-based evidence: expanding user queries by adding words that often co-occur with the query terms in a given corpora (e.g., France and Paris; car and driver). Query expansion terms can also be identified from user feedback (frequent terms occurring in previous results) [20, 68, 76], as well as query logs (terms related to past queries and accessed documents) [29, 43, 51]. We distinguish between query relaxation and query refinement on the grounds that query relaxation occurs before query execution and is considered as a query pre-processing phase [6, 17, 68], whereas query refinement occurs after the first round of query results have been acquired by the user and is thus considered as a query post-processing phase [20, 21, 67]. Such approaches usually require manual tuning to improve performance: too few expansion terms may have no impact, and too many can cause a query drift [68]. In addition, corpus-based approaches require extensive training and huge corpora, which make such methods less practical especially in the context of Web applications. This has led to a growing interest knowledge-based solutions, e.g., [17, 58, 72], investigating the use of ontological information (rather than corpus statistics) to assist the user in formulating and/or expanding keyword queries by: i) allowing some user interaction to accurately identify the intended senses of query-terms, and then ii) expanding/rewriting query keywords via their most related semantic concepts in the reference semantic source [7] (such as WordNet [66] or Yago [45]). Note that query relaxation and query refinement techniques introduce additional
query pre-processing and post-processing overhead respectively compared with the traditional bag-of-words approach, adding statistical and/or knowledge-based information to expand/rewrite each query before/after processing, such that users are generally involved in the query refinement process after the system provides the first round of results.

### 2.2.2. Query Disambiguation

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 [59], 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 (cf. WSD surveys in [69, 81]). Semantic query analysis in IR 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 [7, 58]. The disambiguated query terms are then used in query processing, so that only documents that match the correct sense are retrieved [59]. Yet, the performance of WSD-based approaches depends on the performance of the automated WSD process [37] which generally: i) is computationally complex requiring substantial execution time [69], ii) depends on the context of the query/data processed (e.g., surrounding terms) [22, 90] which is not always sufficiently available (e.g., keyword queries on the Web are typically 2-to-3 words long [53, 86]), and thus iii) do not guarantee correct results [37, 52] as incorrect disambiguation is likely to harm performance [37].

**Discussion:** Most existing methods focus on query pre-processing using query relaxation [17, 64, 68], query disambiguation [7, 58, 59], or query post-processing using query refinement (exploiting user feedback after the system provides the first round of results) [20, 21, 67]. In contrast, our study encloses the semantic knowledge directly into an inverted index so that semantic-aware processing can be done at the most basic indexing level, where more opportunities can be explored toward semantic-aware filtering. This allows users to be involved in the whole process: during data indexing, initial query writing, processing, and then performing query refinement and rewriting.

### 2.3. 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 [54] 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 [92], extending the inverted index structure by adding extra 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. Yet, the authors do not describe how they consider semantic relatedness between terms (what kinds of semantic relations and processing are used), nor how the index is actually built based on linked Web documents. A second approach to semantic indexing is to add words as entities in the ontology [11, 85]. For instance, adding triples of the form \(word \text{ occurs-in-context concept}\), 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 an indexed entry such as with the inverted index. A possible optimization would be to split the relation into \(word \text{ occurs-in-context}\) and \(concept \text{ occurs-in-context}\), yet the relations would remain huge and \(concept \text{ occurs-in-context}\) always has to be processed entirely [11]. A related approach has been used to disambiguate WordNet glosses [70, 85], 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). They produce a 4-tuples index structure \((prefix, terms[]) \leftrightarrow (term, context, concepts[])\) where a prefix contains one index item per occurrence of a term starting with that prefix, adding an entry item for each occurrence of an ontological concept in the same context as one of these words. The authors tailor their method toward incremental query construction, performing context-sensitive prefix suggestions of terms in building queries.

**Discussion:** 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 DB-style (SQL and NoSQL based) querying capability toward semantic full-text search. Hence, while the authors in [11] focus on the IR aspects of indexing, keyword query 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 algorithms designed for seamless storage and manipulation within a typical DB system, allowing to process unstructured (IR-style), structured (SQL-style), and partly structured (NoSQL) data.

3. Input Resources

As indicated previously, our work consists in combining two resources, a *textual data collection* and a *semantic knowledge base*, in order to build SemIndex+. We first describe the textual resource in Section 3.1, and then describe the semantic resource in Section 3.2.

3.1. Textual Data Collection

In our study, a textual data collection can be a set of: i) documents or unstructured text fields in a textual DB, ii) structured tuples in a relational DB, or iii) partly structured data items in a NoSQL DB, as shown in Tables 1 to 3 respectively. Here, we provide a unified formal definition:

**Definition 1 - Textual Data Collection:** A textual data collection $\Delta$ (i.e., *textual collection* for short) is defined as a collection of data objects where every object $O_i \in \Delta$ has a unique identifier $id(O_i)$ and is made of a set of attribute-value pairs $O_i[A_{i1}, a_{i1}, ..., A_{in}, a_{in}]$. Each attribute $A_i$ has domain $\text{dom}(A_i)$ designating the set of values (strings, numbers, etc.) allowed in $A_i$, where $a_i \in \text{dom}(A_i)$. Each data value $a_j$ from $O_i$ associated to attribute $A_i$ is denoted as $O_i.a_j$. We designate by $\Delta.A = \{A_1, A_2, ..., A_n\}$ the set of all attributes associated to all objects in $\Delta$.

Definition 1 - can be used to describe: structured, unstructured, and partly structured (NoSQL) data. It can describe a structured (relational) data collection, such as $\Delta_{\text{struct}}$ in Table 1, which consists of 3 data objects with textual contents organized following a set of 6 attributes: $\Delta_{\text{struct}} = \{ID, title, year, plot, genre, info\}$. Similarly, an unstructured (free text) data collection, such as $\Delta_{\text{free}}$ in Table 2, would be represented as a set of data objects having each an object $ID$ coupled with its textual description, e.g., $\Delta_{\text{free}} = \{ID, \text{description}\}$. Definition 1 - also allows to represent partly structured data collections, such as $\Delta_{\text{part}}$ in Table 3, where every data object in the data collection is made of the object $ID$ combined with a different subset of attributes defined in the data collection, e.g., a subset of $\Delta_{\text{part}}.A = \{ID, \text{title}, \text{year}, \text{plot}, \text{genre}, \text{info}, \text{description}\}$. Here, we adhere to the most basic form of partly structured data collections known as NoSQL attribute-value or key-value stores [33], where every data object (identified by its key) is made of a set of attribute-value items (e.g., $title$—“When a Stranger Calls” is the first item in data object $O_1$ of $\Delta_{\text{NoSQL}}$). More sophisticated NoSQL models such as document DBs [13] or graph stores [74], where attribute-value items can be linked/nested with hierarchical or cross relations, are disregarded here and will be covered in a dedicated study.

Given a textual data collection $\Delta$, an inverted index (also referred to as a posting file, or inverted list) built upon $\Delta$, is (in its most basic form) a sorted list of index terms associated each with a set of object identifiers from $\Delta$ [61], disregarding attribute information as shown in Fig. 2.a. In this study, we extend the basic inverted index to handle multi-attribute data objects, introducing an object-attribute (OA) index:

**Definition 2 - Object-Attribute (OA) Inverted Index:** Given a textual data collection $\Delta$, an OA inverted index built on $\Delta$, denoted as $\text{InvIndexOA}(\Delta)$, is a structure of the form $(\text{dom}(A), \text{OAs}, f)$ where:

- $\text{dom}(A)$ designates the set of values within the domains of all attributes $\forall A_i \in \Delta.A$. Considering text-only domains, values come down to textual tokens, i.e., terms (words/expressions),
- $\text{OAs}$ designates the set of object (identifier)-attribute doublets, i.e., $\text{OAs} = \{(id(O_i), A_j)\} \forall O_i \in \Delta$ and $\forall A_j \in \Delta.A \cap \exists O_i, a_j \neq \utilde$, where $A_j$ is an attribute for which object $O_i$ has a non-null value,
- $f$ is a function mapping each term $\text{term} \in \text{dom}(A)$ with a list of object-attribute doublets $\text{OAs}[\text{term}]$ designating the term’s occurrence locations in $\Delta$, i.e., $\text{OAs}[\text{term}] = \langle(id(O_i), A_j)\rangle^1 / \text{term} \in O_i, a_j$

A term used as textual token in the inverted index is referred to as *index term*, whereas the list of object-attribute doublets, i.e., $\text{OAs}[\text{term}]$, mapping to each index term is referred to as the term’s *posting list* (cf. example in Fig. 2.b).

Fig. 2 shows extracts of a basic inverted index (without attribute information, which was adopted in the initial SemIndex study [23]) as well as the corresponding OA multi-attribute inverted index, built on the sample movie data collection $\Delta_{\text{part}}$ in Table 3, where data objects $O_1$, $O_2$, and $O_3$ have been indexed using index terms extracted from the data collection, and sorted in alphabetic order. Another form of a multi-attribute inverted index can also be defined as a *term-attribute* (TA) index, where term-attribute doublets are mapped with object identities (as shown in Fig. 2c). Both (logically equivalent yet technically different) variants can be straightforwardly integrated in SemIndex+.

---

1 We use symbols ( and ) to designate an *ordered* list of elements, and symbols { and } to designate an *unordered* set.
that we will use \( \Delta_{\text{part}} \) as the running example data collection in the remainder of the paper since it integrates and generalizes all interesting properties from its \( \Delta_{\text{invtext}} \) and \( \Delta_{\text{pure}} \) counterparts.

Semldindex+ allows both attribute-sensitive and attribute-free indexing and querying (as opposed to only attribute-free processing in [23]). We can choose to disregard attributes: i) at the index level (using the traditional InvIndex, adopted in [23]), or ii) at the querying level, following the user query at hand (since InvIndex\(_{\text{OA}}\) and InvIndex\(_{\text{TA}}\) can be straightforwardly processed as traditional attribute-free indexes by simply disregarding attribute information and merging corresponding posting list entries). For instance, running query \( q_4 = \sigma_{\text{\textquoteleft light\textquoteright \textquoteleft horror\textquoteright}} \) \( \Delta_{\text{part}} \) on either InvIndex\(_{\text{OA}}\)(\( \Delta_{\text{part}} \)) or InvIndex\(_{\text{TA}}\)(\( \Delta_{\text{part}} \)), where \( \perp \) designates the combined textual content from all attributes, would return as answer \( O_3 \), since \( O_3 \)'s textual content in \( \Delta_{\text{part}} \) contains both terms “light” and “horror” (despite the fact that the two terms occur in two separate attributes: plot and genre respectively).

<table>
<thead>
<tr>
<th>Term</th>
<th>Object IDs[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Horror”</td>
<td>(O1, O2)</td>
</tr>
<tr>
<td>“Light”</td>
<td>(O1)</td>
</tr>
<tr>
<td>“Sound”</td>
<td>(O3)</td>
</tr>
<tr>
<td>“Zen”</td>
<td>(O1)</td>
</tr>
</tbody>
</table>

Table a. Simple (attribute free) inverted index InvIndex\(_{\text{OA}}\)(\( \Delta_{\text{part}} \)), adopted in our original study in [23]

<table>
<thead>
<tr>
<th>Term</th>
<th>(Object id, Att.[])</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>“Car”</td>
<td>(O1, plot)</td>
<td>(O2, description)</td>
</tr>
<tr>
<td>“Horror”</td>
<td>(O1, genre)</td>
<td></td>
</tr>
<tr>
<td>“Light”</td>
<td>(O1, plot)</td>
<td></td>
</tr>
<tr>
<td>“Sound”</td>
<td>(O1, plot)</td>
<td>(O3, title), (O3, plot)</td>
</tr>
<tr>
<td>“Zen”</td>
<td>(O1, plot)</td>
<td></td>
</tr>
</tbody>
</table>

Table b. Object-Attribute inverted index InvIndex\(_{\text{OA}}\)(\( \Delta_{\text{part}} \))

<table>
<thead>
<tr>
<th>Term, Attribute</th>
<th>Object IDs[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Car, plot”,</td>
<td>(O1)</td>
</tr>
<tr>
<td>“Car, description”</td>
<td>(O2)</td>
</tr>
<tr>
<td>“Horror, genre”</td>
<td>(O1)</td>
</tr>
<tr>
<td>“Light, plot”</td>
<td>(O1)</td>
</tr>
<tr>
<td>“Sound, title”</td>
<td>(O3)</td>
</tr>
<tr>
<td>“Sound, plot”</td>
<td>(O1, O3)</td>
</tr>
<tr>
<td>“Zen, plot”</td>
<td>(O1)</td>
</tr>
</tbody>
</table>

Table c. Term-Attribute inverted index InvIndex\(_{\text{TA}}\)(\( \Delta_{\text{part}} \))

3.2. Semantic Knowledge Base

In the Natural Language Processing (NLP) and Information Retrieval (IR) fields, semantic knowledge bases (i.e., thesauri, taxonomies, and/or Ontologies such as WordNet [66], Roget’s thesaurus [90], and Yago [45]) provide a framework for organizing words/expressions into a semantic space [16]. A knowledge base\(^1\) is usually modeled 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 graph-based structure to model semantic knowledge bases. In such a structure, entities are represented as vertices, and the semantic relationships between entities are represented as directed edges. Formally:

**Definition 3 - Semantic knowledge base:** A semantic knowledge base \( KB \) (i.e., knowledge base for short) is represented as a semantic network graph, also known as knowledge graph, \( G_{\text{KB}}(V, E, L, f_1, f_2) \) where:

- \( V \) is a set of vertices (nodes), designating 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 denoting 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”)
- \( f_1 \) is a function defined on \( V \), designating 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,
- \( f_2 \) 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 \( G_{\text{KB}} \) a multi-graph

An extract from the WordNet lexical 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 \( T_1, T_2, \ldots, T_7 \) represent terms, and their string values (i.e., literal words/expressions) shown along aside 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 opposite directions (instead of two edges), labeled with the names of the symmetrical relationships.

\(^1\) In the remainder of the paper, we will use WordNet [66] as the illustrative semantic knowledge base (cf. Fig. 3).
A simple inverted index $\text{InvIndex}(G_{KB})$ 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.

**4. SemIndex+ Design**

In this section, we present the logical design techniques of SemIndex+, highlighting extensions to our initial model from [23] necessary to handle structured and partly structured data. In the following, we first present SemIndex+‘s graph model, and then describe its construction process.

**4.1. SemIndex+ Graph Model**

We define SemIndex+ as an extension of the SemIndex knowledge graph developed in [23], where additions to the initial model necessary to handle multi-attribute indexes are highlighted in bold:

**Definition 4 - SemIndex+ graph:** Given an input textual collection $\Delta$ and an input knowledge base $KB$, we define $\text{SemIndex+}(\Delta, KB)$ as an extended knowledge graph $\tilde{G}_{\Delta}$ ($V_i, V_a, V_s, L, E_a, f_v, f_e, f_w$) where:

- $V_i$ is a set of index nodes, denoting i) entities (senses and terms) from $KB$, and ii) index terms from $\Delta$:
  - $V_i \subseteq V_i$ the subset of term nodes designating searchable terms\(^1\) in $\tilde{G}_{\Delta}$, i.e., nodes referring to terms from $KB$ and index terms from $\Delta$ (represented visually as circle nodes $\bigcirc$)
  - $V_i^a \subseteq V_i$ the subset of sense nodes in $\tilde{G}_{\Delta}$ referring to senses from $KB$ (represented as double circles $\bigcirc$)

Naturally, $V_i = V_i^t \cup V_i^a$

- $V_a$ is a set of data nodes, denoting data objects from $\Delta$ (represented visually as square shaped nodes $\bigsquare$)

- $V_s$ is a set of attribute nodes, denoting attributes from $\Delta$ (represented as polygon shaped nodes $\bigtriangleup$)

- $V_s$ 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 $\rightarrow$)

- $E_a$ is the set of 3-uniform hyper-edges linking index nodes with data nodes through attribute nodes, called data edges (represented visually as dashed arrows $\rightarrow$)

- $L$ is a set of edge labels including:
  - $\text{Index edge}$ ($E_v$) labels which represent semantic/lexical relationships between index nodes (e.g., hyponymy, meronymy, has-sense, etc.)
  - A single data edge ($E_a$) label: contained, designating the containment relationship between term nodes in $V_i^t$ and data nodes in $V_a$

- $f_v$ is a function defined on $V_i \cup V_a \cup V_s$ representing the string value of each node in $V_i \cup V_a \cup V_s$

- $f_e$ is a function defined on $E_a \cup E_a$, assigning a label from $L$ to each edge in $E_a \cup E_a$

- $f_w$ is a weighting scheme defined on the nodes in $V_i \cup V_a \cup V_s$ and the edges in $E_a \cup E_a$. The weights will be used in selecting and ranking semantic-aware query results (cf. Section 4.5)\(^2\)

\(^1\) 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 5.2).
Building the SemIndex+ graph comes down to: i) generating two separate graph representations for each of the input resources: the textual collection (noted \( \tilde{G}_t \)) and the knowledge base (noted \( \tilde{G}_k \)), following our SemIndex+ graph model, and then ii) tightly coupling the resulting graphs into a single SemIndex+ graph structure (noted \( \tilde{G}_{\Delta} \)), which we describe in the following subsections. A sample SemIndex+ graph is shown in Fig. 7 (Section 4.4), built based on textual collection \( \Delta_{\text{pat}} \) from Table 3 (where corresponding \( \tilde{G}_t \) is provided in Fig. 5.a) and the KB extract in Fig (where corresponding \( \tilde{G}_k \) is reported in Fig. 5.b). It comprises 3 data nodes (\( O_1 - O_2 \)), 4 attribute nodes (\( A_1 - A_4 \)), 3 index sense nodes (\( S_i - S_j \)), and 12 index term nodes (\( T_j - T_{12} \)) along with corresponding data and index edges.

### 4.2. Indexing the Textual Collection

Given an input textual collection \( \Delta \), we use a conversion function following Definition 4 - to produce a SemIndex+ graph representation of \( \Delta \) denoted as \( \tilde{G}_{\Delta} = \text{SemIndex}+ (\Delta, \emptyset) \). It comes down to first generating \( \Delta \)'s inverted index: either using \( \text{InvIndex}_t(\Delta) \) (cf. Fig. 2.b) or \( \text{InvIndex}_s(\Delta) \) (cf. Fig. 2.e), which will be represented as the same SemIndex+ graph \( \tilde{G}_i \) (cf. Fig. 5.a). \( \tilde{G}_i \) consists of: i) a set of index nodes \( V_i \) representing index terms in \( \Delta \) (searchable term nodes), i.e., \( V_i = V_i^+ \) (since \( \Delta \) does not contain senses), i.e., \( V_i^+ = \emptyset \), ii) a set of data nodes \( V_d \) representing data objects in \( \Delta \), iii) a set of attribute nodes \( V_a \) representing attributes from \( \Delta, A \), and iv) a set of edge labels \( L \) including one single label: contained, underlining containment relationships represented as 3-uniform hyper-edges linking index nodes in \( V_i \) with data nodes in \( V_d \) through attribute nodes \( V_a \).

**Definition 5 - 3-Uniform Data Edge in SemIndex+:** Given a SemIndex+ graph \( \tilde{G}_{\Delta} \), we define \( e^{d}_{i} \in \tilde{G}_{\Delta}.E_{d} \) as an ordered 3-uniform hyper-edge connecting data node \( n_d \in \tilde{G}_{\Delta}.V_d \) with searchable term node \( n_i \in \tilde{G}_{\Delta}.V_i^+ \) through attribute node \( n_a \in \tilde{G}_{\Delta}.V_a \) (cf. Fig. 4).

A 3-uniform hyper-edge is a generalized definition of an edge connecting 3 nodes, forming a so-called hyper-graph\(^1\) [36]. We adopt the 3-uniform hyper-edge relationship model to handle attribute nodes in our index, compared with the (attribute-free) 2-uniform data edge relationship model used in the original SemIndex [23] (cf. Fig. 4).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>[image: days_of_thunder_graph.png]</td>
<td>[image: days_of_thunder_graph.png]</td>
<td>[image: stranger_calls_graph.png]</td>
<td>[image: days_of_thunder_graph.png]</td>
<td>[image: sound_of_music_graph.png]</td>
<td>[image: stranger_calls_graph.png]</td>
<td>[image: days_of_thunder_graph.png]</td>
<td>[image: sound_of_music_graph.png]</td>
</tr>
<tr>
<td>a. 2-uniform edge following [23]</td>
<td>b. 3-uniform hyper-edge</td>
<td>c. Multiple 3-uniform hyper-edges</td>
<td>d. Simplified representation of hyper-edges, adopted in remaining examples</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 4.** Sample 3-uniform data edges following SemIndex+ extended model.

A sample \( \tilde{G}_i \), 3-uniform hyper-graph representing our running example inverted indexes (cf. Fig. 2.b and c), based on textual collection \( \Delta_{\text{pat}} \) in Table 3, is shown in Fig. 5.a.

### 4.3. Indexing the Knowledge Base

Similarly, given a semantic knowledge base KB, represented as a knowledge graph \( G_{KB} \), we use a conversion function following Definition 4 - to produce a SemIndex+ graph representation of KB denoted as \( G_{KB} = \text{SemIndex}+ (\emptyset, KB)\).

---

1 Creating the textual collection index graph does not involve an input knowledge base, thus \( KB = \emptyset \) (following Definition 5).

2 A hyper-graph is a generalization of a graph \( G(V, E) \) where \( V \) is a set of nodes and \( E \) is a set of edges underlining non-empty subsets of nodes from \( V \). A hyper-graph is said to be \( r \)-uniform if all edges have cardinality \( r \), i.e., if each edge connects \( r \) nodes together (e.g., a traditional graph comes down to a 2-uniform hyper-graph). An \( r \)-uniform hyper-graph is ordered if the occurrence of nodes in every edge is ordered from 1 to \( r \) [36].
GB's inverted index InvIndex(GKB) is generated and then represented as a SemIndex+ graph $\tilde{G}_{\Delta}$ which inherits the properties of $G_{\Delta}$, such that: i) the set of index nodes $V_i$ represents all nodes in $G_{\Delta}$, including term nodes ($V_i^+$) and sense nodes ($V_i^-$), ii) the sets of data nodes $V_d$ and attribute nodes $N_a$ are 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 $G_{\Delta}$ (e.g., hyponymy, meronymy, has-sense, derivation, etc.). A sample $G_{KB}$ graph representing our running example knowledge base is shown in Fig. 5.

4.4. Coupling Resources to Build SemIndex+

Producing the combined SemIndex+ graph structure $\tilde{G}_{\Delta}$ comes down to coupling both $\tilde{G}_{\Delta}$ and $G_{\Delta}$, noted as: $\tilde{G}_{\Delta} = G_{\Delta} \oplus \tilde{G}_{KB}$, where: i) the set of index nodes $\tilde{G}_{\Delta}.V_i = G_{\Delta}.V_i \cup GB_{\Delta}.V_i$, including corresponding index edges from $\tilde{G}_{KB}$ such that $\tilde{G}_{\Delta}.E = G_{\Delta}.E \oplus \tilde{G}_{KB}.E$, ii) the set of data nodes $\tilde{G}_{\Delta}.V_d = G_{\Delta}.V_d$ and the set of attribute nodes $\tilde{G}_{\Delta}.V_a = G_{\Delta}.V_a$, including corresponding data edges from $\tilde{G}_{\Delta}$ such that $\tilde{G}_{\Delta}.E_d = G_{\Delta}.E_d$, and iii) the set of edge labels $\tilde{G}_{\Delta}.L = G_{\Delta}.L \cup G_{\Delta}.L_{\Delta}$, including all index node semantic/lexical relationships as well as the contained data edge label. The pseudo-code of algorithm SL_Construct to build $\tilde{G}_{\Delta}$ consists of 6 main steps as shown in Fig. 6. Each step is described as follows:

- **Step 1**: Given an input textual collection $\Delta$, build the corresponding inverted index, i.e., using either InvIndex$_{OA}(\Delta)$ or InvIndex$_{TA}(\Delta)$, and generate the corresponding $\tilde{G}_{\Delta}$ 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, to access them more efficiently during resource coupling, and later during query execution), and then construct $\tilde{G}_{KB}$ graph as illustrated previously.

1. Creating the knowledge base index graph does not involve an input textual collection, thus $\Delta = \emptyset$ (following Definition 5).
2. The missing term problem is discussed in Section 4.4.
3. The set of index edges in $\tilde{G}_{\Delta}$ is not exactly equivalent to that in $\tilde{G}_{\Delta}$ since it might contain additional index edges connected with index terms in $\tilde{G}_{\Delta}$ which do not map to any term node in $\tilde{G}_{\Delta}$. This is discussed as the missing terms problem in Step 4 of algorithm SL_Construct (cf. Fig. 6).
- **Step 3**: Combine the two SemIndex+ graphs into a single graph structure $\tilde{G}_3$. To do so, we map and then merge all searchable term nodes in $\tilde{G}_2$, i.e., $\tilde{G}_2.V^+_i$, with searchable term nodes in $G_{KB}$, i.e., $G_{KB}.V^+_i$, as follows:

1. For each pair of searchable term nodes in $\tilde{G}_2.V^+_i$ and $G_{KB}.V^+_i$, if their string values are equal, then remove one of them and merge all the connected edges.
2. Sense nodes in $G_{KB}$ are kept the same in $\tilde{G}_2$, i.e., $\tilde{G}_2.V^+_\Delta \equiv G_{KB}.V^+_\Delta$, but are connected with the corresponding searchable term nodes $\tilde{G}_2.V^+_i$.
3. Data nodes and attribute nodes in $\tilde{G}_2$ are kept the same in $\tilde{G}_2$, i.e., $\tilde{G}_2.V^+_d \equiv G_{KB}.V^+_d$ and $\tilde{G}_2.V^+_a \equiv G_{KB}.V^+_a$, but are connected with the corresponding searchable term nodes $\tilde{G}_2.V^+_i$ using the contained data edge relationship.

Fig. 7.a shows the result of combining the two sample SemIndex+ graphs used in our running example: $\tilde{G}_3$ of the extract of textual collection $\Delta_{ext}$ and $G_{KB}$ of the extract knowledge base $KB$.

```
Algorithm SI_Construct // SemIndex+ construction
Input: Δ // Textual data collection
      KB // Semantic knowledge base
      W // Weighting function parameters
Output: $G_S$ // SemIndex+ graph
Begin
Step 1: Build InvIndex(Δ) to construct $\tilde{G}_2$ 1
Step 2: Build InvIndex(KB) to construct $G_{KB}$ 2
Step 3: Coupling $\tilde{G}_2$ and $G_{KB}$ into $G_S$ by:
         1. Mapping & merging searchable term nodes in $\tilde{G}_2.V^+_i$ and $G_{KB}.V^+_i$ 4
         2. Including sense nodes from $G_{KB}.V^+_\Delta$ 5
         3. Including data nodes from $\tilde{G}_2.V^+_d$ 6
Step 4: Run MissingTerms_Linkage algorithm
         // Connect missing terms in $G_S$
         Step 5: Assign weights to edges & data nodes in $G_S$
         - According to parameters W and weighting function $f_W$ 8
         Step 7: Remove from $\tilde{G}_2$
         1. Labels from all edges: $\tilde{G}_2.E$ 11
         2. String values from all nodes except searchable terms: $\tilde{G}_2.V^+_d$ 12
Return $G_S$
End
```

**Fig. 6.** Pseudo-code of SemIndex+ construction algorithm.

- **Step 4**: Searchable terms from $\tilde{G}_2.V^+_i$ which do not map to any searchable term in $G_{KB}.V^+_i$ can exist, which we identify as missing terms (e.g., term “Horror” in Fig. 5). These come down to terms from the data collection with no semantic cues in the knowledge base (e.g., “horror” appears in object $O_1$ of $\Delta_{ext}$ but does not appear in the extract knowledge base $KB$ in Fig. 3). To solve this, we introduce algorithm MissingTerms_Linkage (cf. Appendix II) inspired from distributional thesaurus construction methods, e.g., [18, 87], which creates links from each missing term to one or more closely related terms, i.e., terms that co-occur together in the text collection (e.g., term “horror” links with “car”, considered as its most related – highest co-occurrence frequency term – in $\Delta_{ext}$, cf. Fig. 7). The new co-occurrence links (index edges) are labeled occurs-with. Our MissingTerms_Linkage algorithm is provided in Appendix II since it’s outside of the main scope of this paper and will be evaluated in a dedicated future study.

- **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, which we describe in Section 4.5.
- **Step 6**: It removes edge labels and string values of all nodes in $\tilde{G}_{\mathcal{S}}$ except for $V^*_t$ (searchable term nodes) and $V_a$ (attribute nodes), since all other nodes are not required for processing semantic queries. Removing node string values helps improve $\text{SemIndex}^+$’s scalability in terms of size, construction time, and query processing time (cf. experiments in Section 7).

![a. SemIndex+ graph before removing edge labels and string values.](image1)

![b. Final SemIndex+ graph representation.](image2)

Fig. 7. $\text{SemIndex}^+$ graph $\tilde{G}_{\mathcal{S}}$ obtained after coupling the data collection and the knowledge base graphs in Fig. 5.

Fig. 7.b illustrates our running example $\tilde{G}_{\mathcal{S}}$ excluding edge and node labels except for searchable term nodes and attribute nodes which are required in the querying process. Edge and node weights are omitted from the figure for clearness of presentation.

### 4.5. SemIndex+ Weighting Functions

After indexing and coupling the textual resource and the semantic resource into a unified $\text{SemIndex}^+$ graph (i.e., Step 3 of algorithm $\text{SI_Construct}$), and handling the missing terms problem (Step 4), we introduce a set of weighting functions (Step 5) to assign weight scores to $\text{SemIndex}^+$’s entries, including: data nodes, index nodes, attribute nodes, as well as data edges and index edges. The weighting functions will be used to effectively select and rank semantically relevant results w.r.t. the user’s query (cf. $\text{SemIndex}^+$ query processing in Section 5). Other weight functions could be later added to cater to the index designer’s needs.

#### 4.5.1. Index Node Weight

Considering an index node $n_i \in \tilde{G}_{\mathcal{S}} V^*_i$, the weight of $n_i$ denoted as $W_{\text{IndexNode}}(n_i)$, is computed according to the below formula where we consider “Fan-in” to be the number of nodes connected with the target index node:

$$W_{\text{IndexNode}}(n_i) = \frac{\text{Fan-in}(n_i)}{\text{Max Fan-in}(n_j)} \in [0,1]$$

The rational here is that an index node is more important if it receives more links from other indexing nodes.

#### 4.5.2. Index Edge Weight

Given an index edge $e'_i \in \tilde{G}_{\mathcal{S}} E$, outgoing from index node $n_i$ and incoming toward index node $n_j$ in the $\text{SemIndex}^+$ graph, we define the weight of $e'_i$ as:

$$W_{\text{IndexEdge}}(e'_i) = \frac{1}{\text{Fan-out}_i(n_j)} \in [0,1]$$

The weight of an index edge increases with the number of outgoing links from a certain index node to another, taking into account the semantic relation type of the index link at hand.
4.5.3. Data Node Weight

The weight of a data node \( n_d \in \tilde{G}_d \mathcal{V}_d \) in the \textit{SemIndex+} graph is defined as:

\[
W_{\text{DataNode}}(n_d) = \frac{\text{Fan-In}(n_d)}{\max_{n_e \in \tilde{G}_d \mathcal{V}_d} (\text{Fan-In}(n_e))} \in [0, 1]
\]

where \( \text{Fan-In}(n_d) \) designates the number of foreign key/primary key data links (joins) outgoing from data nodes (tuple) where the foreign keys reside, toward data node (tuple) \( n_d \) where the primary key resides. Similarly to index node weight, the rational is that a data node is more important when it receives more links from other data nodes.

4.5.4. Attribute Node Weight

The weight of an attribute node \( n_a \in \tilde{G}_d \mathcal{V}_a \) in \textit{SemIndex+} is manually (or semi-automatically\(^1\)) acquired from the data creator/user, since it is a data design issue, such that:

\[
W_{\text{AttNode}}(n_a) \in [0, 1] \quad \text{where for each } n_a \in \tilde{G}_d \mathcal{V}_a, \quad \sum_{n_a \in \mathcal{A}} W_{\text{AttNode}}(n_a) = 1
\]

In other words, the sum of the weights of all attribute nodes (excluding the identifier attribute) connected with a given data node \( n_d \) through any index term node \( \forall n_e \in \tilde{G}_d \mathcal{V}_d \) needs to be normalized in order to sum up the full (100%) descriptive power of \( n_d \). For instance, considering our running example \textit{SemIndex+} graph from Fig. 7, we (as users) consider \( W_{\text{AttNode}}(\text{title}) = 0.4 \), \( W_{\text{AttNode}}(\text{genre}) = 0.3 \), \( W_{\text{AttNode}}(\text{description}) = 0.3 \), and \( W_{\text{AttNode}}(\text{year}) = W_{\text{AttNode}}(\text{plot}) = W_{\text{AttNode}}(\text{info}) = 0.1 \). Hence:
- For data object \( O_1 \), attribue weights are already normalized such that: \( W_{\text{AttNode}}(\text{title}) + \ldots + W_{\text{AttNode}}(\text{info}) = 1 \)
- For \( O_2 \), \( W_{\text{AttNode}}(\text{description}) \) needs to be normalized to become \( 1 \), since description is the only attribute describing \( O_2 \) (and thus should sum the full descriptive power of the data node).
- Similarly for \( O_3 \), attribute weights need to be normalized to obtain \( W_{\text{AttNode}}(\text{title}) + W_{\text{AttNode}}(\text{plot}) + W_{\text{AttNode}}(\text{genre}) = 1 \). By applying linear normalization to the above user chosen weights for instance, we obtain \( W_{\text{AttNode}}(\text{title}) = 0.5 \), \( W_{\text{AttNode}}(\text{genre}) = 0.375 \), and \( W_{\text{AttNode}}(\text{plot}) = 0.125 \).

4.5.5. Data Edge Weight

Given a data edge \( e_d^q \in \tilde{G}_d \mathcal{E}_d \) connecting an index node \( n_i \) with a data node \( n_d \) through an attribute node \( n_a \) (e.g., data edge connecting index node \( T_i \) with data node \( O_2 \) through attribute \( A_2 \) since the term “car” occurs in the textual description of \( O_2 \) through its desc attribute, likewise for \( T_i-A_2-O_2, T_i-A_2-O_3, \ldots, T_i-A_2-O_k \), in Fig. 7), we compute the weight of \( e_d^q \) as an adapted TF-IDF (Term Frequency Inverse Document 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, and IDF underlines the number of data edges connecting the same index node with other data nodes (i.e., the fan-out of the index node in question). Hence, given a data edge \( e_d^q \) incoming from index node \( n_i \) toward data node \( n_d \) through attribute node \( n_a \) where \( n_i, l \) denotes the string value of \( n_i \), we define:

\[
W_{\text{DataEdge}}(e_d^q) = \text{TF}(n_i, l, n_d, n_a) \times \text{IDF}(n_i, l, \tilde{G}_d \mathcal{V}_d)
\]

where TF and IDF are calculated as follows:

\[
\text{TF}(n_i, l, n_d, n_a) = \frac{\text{NbOcc}(n_i, l) \times W_{\text{AttNode}}(n_a)}{\max_{n_j \in \tilde{G}_d \mathcal{V}_d} \text{NbOcc}(n_j, l)} \in [0, 1]
\]

where the number of occurrences of a term \( n_i, l \), denoted as \( \text{NbOcc}(n_i, l) \), is weighted by the corresponding attribute (such that terms occurring through higher weight/more relevant attributes will have a higher impact on the data edge.

---

\(^1\) Learning algorithms can be devised to evaluate the relevance of attributes based on existing data collections [46]. Yet, such approaches also require manual expert input to train the learning algorithms.
weight), and where TF is normalized w.r.t. the maximum number of occurrences of any index node string literal \( n_d.l \) within the target data node \( n_d \).

\[
\text{IDF}(n_d.l, \sim G_{SI} V_d) = 1 - \frac{DF(n_d.l, \sim G_{SI} V_d)}{N} \in [0, 1]
\]

where \( N \) is the total number of data nodes in the \( \text{SemIndex}^+ \) graph, and \( DF(n_d.l, \sim G_{SI} V_d) \) is the number of data nodes in the graph containing at least one occurrence of \( n_d.l \).

5. \textit{SemIndex}^+ \textit{ Query Processing}

Given the above weighting functions (others could be added later), we define our query model and present our algorithm to perform semantic-aware search with \( \text{SemIndex}^+ \).

5.1. Query Model

The \textit{semantic-aware queries} considered in our approach are \textit{conjunctive projection selection} queries of the form \( q = \pi_X \sigma_P (\ell(\Delta)) \), defined over a data collection \( \Delta \) (structured, unstructured, or partly structured, cf. Definition 1), where \( X \) is a subset of attributes \( X \subseteq A \cup \perp \) (where \( \perp \) designates the combined textual content from all attributes, allowing both \textit{attribute-sensitive} and \textit{attribute-free} querying), \( \ell \in \mathbb{N} \) represents a link distance threshold designating different levels of semantic awareness in query execution, and \( P \) is a conjunctive selection predicate defined as follows:

**Definition 6 - Conjunctive Selection Predicate:** It is defined as an expression \( P \) on a string-based attribute\(^1\) or on the combined textual content of all attributes \( A \cup \perp \), of the form: \( (A, \not\exists s) \), where \( s \) is a user-given string value (e.g., a selection term/keyword), and \( \not\exists \in \{=, \text{like} \} \) whose evaluation against values in \( \text{dom}(A) \) is previously defined.

Following the value of link distance \( \ell \), we consider four semantic-aware query types:

i. **Standard Query:** When \( \ell = 1 \), the query is a standard containment query, involving only \textit{data edges} (connecting \textit{data nodes} with \textit{searchable term nodes} through \textit{attribute nodes}), using the \textit{contained} relationship, such that no semantic information is involved.

ii. **Lexical Query:** When \( \ell = 2 \), the link distance is increased by 1 to include (in addition to \textit{data edges}), first level \textit{index edges}. They designate lexical relationships between \textit{searchable term nodes} (namely the \textit{derivation} relationship, where one term \textit{derives} another term), such that basic lexical information is involved.

iii. **Synonym-based Query:** When \( \ell = 3 \), the senses (synsets) are also involved. Here, link distance includes the second level \textit{index edges}: connecting \textit{searchable term nodes} with corresponding \textit{sense nodes} (via the \textit{has-sense} and \textit{has-term} semantic relationships), such that synonymous terms corresponding to the \textit{sense nodes} are involved. Note that there is no direct edge between \textit{data nodes} and \textit{sense nodes}.

iv. **Extended Semantic Query:** When \( \ell \geq 4 \), the data graph of \( \text{SemIndex}^+ \) can be explored in all possible ways, covering \textit{index edges} designating all kinds of semantic relationships (\textit{hyponymy}, \textit{meronymy}, etc.) between \textit{index nodes}, to reach even more semantically relevant results.

While we currently focus on relaxing “strict” conjunctive querying by increasing link distances between query and data nodes, yet our query model and approach can also incorporate different kinds of “weak AND” operators such as \textit{fuzzy} predicates [50, 91] (which we will investigate in an upcoming study).

5.2. Query Answer

The answer to a query \( q = \pi_X \sigma_P (\ell(\Delta)) \) in \( \text{SemIndex}^+(\Delta, KB) \), noted \( q(\Delta) \), is defined as follows:

**Definition 7 - Query answer:** Given \( \text{SemIndex}^+(\Delta, KB) \) and its graph representation \( \sim G_{SI} \), a query answer \( q(\Delta) \) is the set of distinct root nodes of all answer trees in \( \sim G_{SI} \), where every root node represents a data object in \( \Delta \). We define an answer tree as a connected sub-graph \( T \subseteq \sim G_{SI} \) satisfying the following conditions:

- **Root node:** \( T \)'s root is a \textit{data node}, i.e., \( R(T) \in \sim G_{SI}.N_d \) and it is the only data node in \( T \), designating the corresponding textual object in \( \Delta \) to be returned to the user.

---

\(^1\) Although our approach is generic and can be defined on other types of attributes.

\(^2\) \( \perp \) designates the combined textual content from all attributes.
- **Leaf nodes**: All leaf nodes in the answer tree \( T \) are *searchable term* nodes mapping to query terms (keywords).
- **Tree structure**: For each node \( n \in T \), there exists exactly one directed path from \( n \) to \( T \)'s root node \( R(T) \).
- **Depth boundary**: The depth of \( T \), i.e., the maximal number of edges between the root and a leaf node, is not greater than the link distance threshold \( \ell \).
- **Minimal tree**: No node can be removed from \( T \) without violating some of the above conditions.

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.

<table>
<thead>
<tr>
<th>a. Answer tree for a standard query ( (\ell = 1) ).</th>
<th>b. Answer tree for a lexical query ( (\ell = 2) ).</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Standard Query Tree" /></td>
<td><img src="image" alt="Lexical Query Tree" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>c. Answer tree for a synonym-based query ( (\ell = 3) ).</th>
<th>d. Answer tree for an extended semantic query ( (\ell = 4) ).</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Synonym Query Tree" /></td>
<td><img src="image" alt="Semantic Query Tree" /></td>
</tr>
</tbody>
</table>

**Fig. 8.** Sample answer query trees\(^1\) with different link distance threshold values \( \ell \), extracted from our running example *SemIndex*+ graph (Fig. 7).

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

i. **Standard Query**: When \( \ell = 1 \), 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. 8.a for query \( q = \sigma_{\text{plot}} \in (\text{"car"}, \text{"light"}) \, \ell_{\Delta}(\Delta_{\text{pat}}) \) considering our running example data collection \( \Delta_{\text{pat}} \) (Table 3) and the corresponding *SemIndex*+\((\Delta_{\text{pat}}, \text{KB}) (Fig. 7)\).

---

\(^1\) While all edge and node labels are removed from the *SemIndex* graph except for *searchable term* nodes, we show *synset* node glosses here for the sake of presentation.
ii. **Lexical Query:** When \( \ell = 2 \), the answer tree includes lexical connections between query term nodes and other index term nodes. Fig. 8.b is an example answer tree for query \( q = \sigma_{\text{plot}} \in \{ \text{"race car";"light"} \} \).  

iii. **Synonym-based Query:** When \( \ell = 3 \), the answer tree includes *sense nodes*, in addition to the two previous cases. Note that due to the *minimal tree* restriction (Definition 7 - i), 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. 8.c for query \( q = \sigma_{\text{plot}} \in \{ \text{"pave";"clean"} \} \). The synonyms of the two query terms, “zen” and “light” are also contained in the answer tree rooted at the data node of object \( O_1 \).  

iv. **Extended Semantic Query:** When \( \ell = 4 \), 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. 8.d for query \( q = \sigma_{\text{plot}} \in \{ \text{"hot";"clean"} \} \).  

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 possible answer tree.  

### 5.3. Relevance Ranking  
While a huge number of query answers could be identified for a given query, the objective of any typical IR system would be to identify the most relevant of these candidate results and rank them based on their respective relevance w.r.t. the query [9]. In *SemIndex*+, we evaluate the relevance of data nodes returned as candidate query answers (i.e., answer tree root nodes) using typical *Dijskstra*-style shortest distance computations (described in the following section). Yet, instead of identifying the shortest distance between *searchable term* nodes and *data nodes* in the *SemIndex*+ graph, we compute their maximum similarity (as the inverse of distance). Formally:  

| Definition 8 - Relevance Score measure: Given a *SemIndex*+ graph \( \tilde{G}_{ST} \), a data node \( n_d \in \tilde{G}_{ST} V_d \) and a searchable term node \( n_l \in \tilde{G}_{ST} V_l \) (cf. visual representation in Fig. 9), we define the relevance (similarity) score of \( n_d \) w.r.t. \( n_l \), noted \( \text{score}(n_d, n_l) \), as the sum of the inverse of the lexical/semantic distances (in number of edges) between \( n_d \) and every index node on the path leading from \( n_d \) to \( n_l \), noted \( \text{path}(n_d, n_l) = (n_d, n_{p_1}, n_{p_2}, \ldots, n_{p_i}, n_l) \), where every node and edge on \( \text{path}(n_d, n_l) \) is weighted following *SemIndex*+’s weighting functions:  

\[
\text{score}(n_d, n_l) = \sum_{i=1}^{n} \left( \frac{1}{d(n_d, n_{p_i})} \right)
\]

where \( d(n_d, n_l) \) designates the distance in number of edges between data node \( n_d \) and index node \( n_l \).  

![Fig. 9. Sample node linkage representation in the *SemIndex*+ graph.](image)

Our relevance score measure in Definition 8 - produces normalized relevance scores \( \in [0, 1] \) where:

- A minimum relevance score \( =0 \) is reached when searchable term node \( n_l \) is not connected with \( n_d \), i.e., there is no path \( \text{path}(n_d, n_l) \) leading from \( n_d \) to \( n_l \) in \( \tilde{G}_{ST} \) .
- A maximum relevance score \( =1 \) is reached when searchable node \( n_l \) is directly connected with data node \( n_d \) through data edge \( e^i \) (the searchable term occurs in the string value of the data node). For instance, this is the case of index node \( n_{p_i} \) in Fig. 9, which can produce \( \text{score}(n_d, n_{p_i}) = W_{\text{data}}(n_d) \times W_{\text{att}}(n_{p_i}) \times W_{\text{edge}}(e^i) \times \frac{1}{d(n_d, n_{p_i})} = 1 \) where \( d(n_d, n_{p_i}) \) is minimum (\( =1 \)) and given that all three: data node, attribute node, and data edge weights are maximum (\( =1 \)).
The relevance score increases with the semantic/lexical closeness between \( n_d \) and \( n_i \) in \( \tilde{G}_{SI} \), and decreases with their distance. In other words, the farther away a searchable term node \( n_i \) from data node \( n_d \) in \( \tilde{G}_{SI} \) (i.e., the higher the distance \( d(n_d, n_i) \)), the lesser its semantic/lexical relatedness with \( n_d \), and thus the lower its relevance score w.r.t. \( n_d \).

The relevance score also increases/decreases with SemIndex+ node/edge weighting functions, allowing users to easily consider or disregard relevance weights following their needs (e.g., one user could prefer to consider data node weights only, while disregarding others).

Consider the example in Fig. 8.c. Here, we assume that users disregard all weighting functions for simplicity:

- The relevance of data node \( O_1 \) w.r.t. term node \( T_1 \), where \( T_1 \) is situated at link distance \( \ell=1 \) from \( O_1 \) (direct linkage, where \( T_1 \) occurs in the string value of \( O_1 \)): \( \text{score}(O_1, T_1) = \frac{1 \times 1 \times 1 \times 1}{1 \times 1 \times 1} = 1 \) (i.e., maximum score).
- The relevance of \( O_1 \) w.r.t. \( S_1 \), at link distance \( \ell=2 \): \( \text{score}(O_1, S_1) = \frac{1 \times 1 \times 1 \times \frac{1}{2} + 1 \times 1 \times \frac{1}{2} + 1 \times \frac{1}{2} \times 1}{2} = 0.75 \)
- The relevance of \( O_1 \) w.r.t. \( T_7 \), at link distance \( \ell=3 \): \( \text{score}(O_1, T_7) = \frac{1 \times 1 \times 1 \times \frac{1}{2} + 1 \times 1 \times \frac{1}{2} + 1 \times 1 \times \frac{1}{2} + 1 \times 1 \times 1}{3} = 0.6112 \)

One can clearly realize that index nodes \( T_7, S_1, \) and \( T_7 \) which are (semantically/lexically) decreasingly related to (they are increasingly more distant in the SemIndex+ graph from) data node \( O_1 \), produce decreasing relevance scores (1, 0.75, and then 0.6112) respectively.

### 5.4. Querying Algorithm

The pseudo-code for our query processing algorithm, titled SI_PSS, is shown in Fig. 10. It takes as input a SemIndex+ graph \( \tilde{G}_{SI} \), a conjunctive projection selection query \( q \) including link distance threshold \( \ell \), as well as range and \( kNN \) (k-nearest neighbor) query selection thresholds \( r \) and \( k \), and produces as output the list of data nodes \( N_{d,out} \) (i.e., the answer trees’ root nodes) designating the data objects returned as query answers, selected and ordered following their relevance w.r.t. the query. It is parallelized (multithreaded), processing query terms and starting index nodes using multiple threads running in parallel. The overall process can be described as follows:

**Step 1:** Every query term is assigned to a dedicated thread, and is thus processed independently from other query terms (line 2).

**Step 2:** The algorithm then identifies in \( \tilde{G}_{SI} \) the index (searchable term) nodes mapping to each query term (using function \( \text{getNodeIDs}(\cdot) \), line 4)\(^1\). These will serve as starting nodes to navigate the SemIndex+ graph.

**Step 3:** Every starting (index) node is then assigned to its own dedicated thread, and processed independently from other starting nodes (line 5).

**Step 4:** For every starting (index) node, the minimum distance paths at \( \ell \) from the starting node to data nodes are identified, i.e., using Dijkstra’s shortest path algorithm (performed by function \( \text{findShortestPaths}(\cdot) \), line 7).

**Step 5:** Of these shortest paths, the algorithm then identifies the paths which contain data nodes (using function \( \text{getDataNodeIDs}(\cdot) \), line 8), reachable through the designated query attribute for every term, and then adds the resulting data nodes to the list of output data nodes \( N_{d,out} \).

**Step 4:** Consequently, the resulting data nodes are gradually merged with the list of existing answer data nodes as they are produced by each thread\(^2\). A score is then assigned to every answer node by computing its relevance score w.r.t. every query term’s index node (using \( \text{mergeAndRank}(\cdot) \), line 9). The algorithm finally returns the list of answer data nodes, ranked in descending order following answer (data) node relevance scores (i.e., from the most to the least relevant answer node).

**Step 5:** Data node result selection (line 12) is undertaken using a combined range-\( kNN \) query selection operator, following range query and \( kNN \) thresholds provided by the user. The user can choose to apply one, both, or none of the two selection operators, by specifying (disregarding) the value of the corresponding threshold(s).

---

\(^1\) Initial index/query term mapping is performed regardless of query attributes. Term mapping identifies the leaf nodes of potential answer trees in the SemIndex+ graph, which will be later pruned following the terms’ container attributes (if any) in Step 3 (graph navigation phase) of the algorithm.

\(^2\) The physical implementation the querying algorithm is configured to run as many threads as necessary to process the different query terms and starting nodes, where thread scheduling and parallel execution are left to the operating system.
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 Dijkstra shortest distance computations [28]. Basically, findShortestPaths() explores the SemIndex+ graph with Dijkstra’s algorithm from multiple starting index nodes \( n_{i,j} \) (multiple query terms \( T_i \)). For each visited node \( n_i \), it stores its maximum relevance scores (minimum distances) from all starting nodes (query terms). The relevance score of an index node \( n_i \) (likewise for a data node \( n_j \)) w.r.t. a starting node (query term) \( n_{i,j} \) is evaluated using our relevance score measure (Definition 8) - applied along the path between \( n_{i,j} \) and \( n_i \). In other words, the shortest distance of \( n_i \) \( n_{i,j} \) from \( n_{i,j} \) is identified by computing the maximum relevance score of \( n_i \) \( n_{i,j} \) w.r.t. \( n_{i,j} \).

**Algorithm SI _ParallelSemanticSearch** // SemIndex+ Parallel Semantic Search

**Input**:
- \( G_\alpha \) // SemIndex+ graph
- \( q \) // A conjunctive projection selection query, including link distance threshold \( \ell \)
- \( \{r, k\} \) // range and k-nearest neighbor selection operators

**Output**: \( N_{inOut} \) // A list of ranked data nodes from \( G_\alpha \) designating query answers

**Begin**

1. \( N_\alpha = \emptyset \)
2. Create Thread for each \( (T_i, A_j) \in q \) // Processing each selection term simultaneously
3. **Step 1**: \( N_{inOut} = \text{getNodeIDs}(T_i, G_\alpha) \) // Identify index nodes for every selection term
4. Create Thread for each \( n_i \in N_{inOut} \) // Processing every (starting) index node simultaneously
5. **Step 2**: \( SP = \text{findShortestPaths}(n_i, \ell, G_\alpha) \) // Identify shortest paths within distance \( \ell \) from \( n_i \)
6. **Step 3**: \( N_\alpha = \text{getDataNodeIDs}(SP, G_\alpha, A_j) \) // Identify the set of data (root) nodes in each shortest path reachable from the term node \( n_i \) through attribute \( A_j \)
7. **Step 4**: \( N_{inOut} = \text{mergeAndRank}(N_{inOut}, N_\alpha) \) // Merging and ranking data nodes based on relevance
8. **Step 5**: \( N_{inOut} = \text{select}(N_{inOut}, \text{range, kNN}) \) // Result Selection using range and/or kNN threshold(s)
9. Return \( N_{inOut} \)

**End**

Fig. 10. Pseudo-code of SemIndex+ parallel semantic search algorithm.

For example in Fig. 8.c, given query \( q = \sigma_{\text{plot} \in \{ \text{"pane"}, \text{"clean"}\}} (\ell=3) (\Delta_{\text{dist}}) \) made of terms “pane” and “clean”, the algorithm starts to expand from index nodes \( T_1 \) and \( T_2 \). In this example, we disregard (i.e., assign unit scores to) all SemIndex+ node/edge weighting functions (to simplify computations) in evaluating our relevance score measure. Hence, the relevance score of \( T_1 \) is initialized as a vector of two scores \( <1, 0> \), the first representing the relevance score w.r.t. \( T_2 \) (“pane”), i.e., \( \text{score}(T_1, T_2)=1 \), and the second representing relevance score w.r.t. \( T_1 \) (“clean”, i.e., \( \text{score}(T_1, T_1)=0 \) since \( T_1 \) is not initially reachable from \( T_2 \)). Similarly, the weight score vector of \( T_2 \) is initialized to \( <0, 1> \). The weights of all other index nodes are initialized to \( <0, 0> \). The relevance scores are then updated when each edge is explored in the graph. For example, starting from \( T_2 \), the weight of index node \( S_7 \), which was initialized to \( <0, 0> \) becomes \( <1, 0> \) when the node is reached, where \( \text{score}(S_7, T_1) = 1 \) (relevance score at link distance \( \ell=1 \) from \( T_2 \)) and \( \text{score}(S_7, T_1) = 0 \) (since \( S_7 \) is not yet reachable from \( T_2 \)). Likewise, the weights of nodes \( T_1 \) and \( O_1 \) become \( <0.75, 0> \) and \( <0.6112, 0> \) respectively when the nodes are reached from \( T_2 \), and so forth. On the other hand, starting from \( T_2 \), the weights of nodes \( S_5 \), \( T_1 \), and \( O_1 \) become \( <0, 1> \), \( <0, 0.75> \), \( <0, 0.6112> \) respectively.

Consequently, given that a data node \( n_k \) can be reached from multiple starting nodes \( N_{i,j} \) (i.e., multiple leaves in the answer tree), function mergeAndRank() computes the combined relevance score of a data node (i.e., answer tree root node) as the aggregate relevance scores from each starting node (each answer tree leaf node). As for the aggregation function, various mathematical formulations for combining relevance scores can be used [5, 83], among which the maximum, minimum, average and weighted sum functions. Here, we utilize the average aggregation function to account for the average semantic relatedness between the query answer root node and all tree leaf nodes:

\[
\text{score}(n_k) = \frac{1}{\text{avg score}(n_{i,j}, n_k)} \forall \text{ } n_{i,j} \text{ having } s_i \in S
\]

(9)

For instance, considering our current example based on Fig. 8.c, the vector path score of data node \( O_1 \) would be \( <0.6112, 0.6112> \), and thus its combined path score becomes 0.6112. Considering the example in Fig. 8.b, starting from query terms “race car” and “light”, the vector path score of data node \( O_k \) would be \( <0.75, 1> \) (assuming unit SemIndex+ node/edge weights as in the previous example), and thus its combined path score becomes 0.875. A data node which is not reachable from all query term nodes will have at least one relevance score =0 (i.e., zero semantic relatedness), along one (or more) of its relevance vector dimensions.
6. Complexity Analysis

Our solution is of quadratic complexity, requiring $O(N^2)$ time for building the $\text{SemIndex}^+$ graph where $N$ represents the maximum size (in number of nodes) between the textual collection and the knowledge base, and $O(N_{i\text{acc}}^2)$ time for executing semantic-aware queries where $N_{i\text{acc}}$ is the number of index nodes accessed during query execution.

6.1. Building $\text{SemIndex}^+$

6.1.1. Time Complexity

Building $\text{SemIndex}^+$ using algorithm $\text{SI}_\text{Construct}$ (cf. Fig. 6) is done in quadratic time and simplifies to $O(N^2)$ since:

- Step 1: Building the inverted index, and consequently the $\text{SemIndex}^+$ graph for the textual collection $\Delta$, i.e., $\tilde{G}_\Delta$, is of typical $O(|\Delta| \times |\Delta| \times N_\Delta)$ complexity, where $|\Delta|$ and $N_\Delta$ designate the number of attributes and the number of searching terms in $\tilde{G}_\Delta$ respectively, which simplifies to $O(|\Delta| \times N_\Delta)$ since $|\Delta|$ is usually limited,

- Step 2: Also, building the $\text{SemIndex}^+$ graph for the knowledge base $KB$, $\tilde{G}_{KB}$, is of $O(|KB| \times N_{KB})$, where $N_{KB}$ is the number of searchable term nodes from $\tilde{G}_{KB}$,

- Step 3: Coupling both $\Delta$ and $KB$’s $\text{SemIndex}^+$ graphs by mapping and merging searchable term nodes in both $\tilde{G}_\Delta$ and $\tilde{G}_{KB}$ 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 the algorithm $\text{MissingTermsLinkage}$ (cf. Appendix II) can be performed in worst case $O(N_{miss} \times N_{term})$, where $N_{miss}$ and $N_{term}$ respectively designate the number of missing terms and the number of term index nodes in the $\text{SemIndex}^+$ graph. Note that building the distributional thesaurus (to identify term relatedness vectors, based on their co-occurrences in the reference corpus) is conducted offline prior to $\text{SemIndex}^+$ building and thus does not affect its complexity.

- Step 5: The complexity of the weighting process varies according to the weight functions used. It amounts to $O(1)$ when assigning equal weights, and varies following our weighting scheme as follows:

  - Data nodes: assigning an object rank score to compute data node weights simplifies to $O(|\Delta| \times |\Delta_{joint}|)$, where $|\Delta_{joint}|$ designates the number of data links (i.e., foreign key/primary key data joins) in $\Delta$,
  
  - Attribute nodes: normalizing user defined attribute weights for every attribute of every data node required $O(|\Delta| \times |\Delta|)$ time, which simplifies to $O(|\Delta|)$ since $|\Delta|$ is small w.r.t. $|\Delta|$,
  
  - Data edges: performing attribute-sensitive term frequency - inverse document frequency computations to assign data edge weights comes down to $O(N_{term} \times |\Delta| \times |\Delta|)$ time, which simplifies to $O(N_{term} \times |\Delta|)$ since $|\Delta|$ is small w.r.t. $N_{term}$ and $|\Delta|$,
  
  - Index nodes: assigning an object rank score to compute index node weights simplifies to $O(N_{i} + N_{E})$, where $N_{i}$ designates the number of index nodes and $N_{E}$, the number of index edges in the $\text{SemIndex}^+$ graph,
  
  - Index edges: computing index edge weights comes down to $O(N_{E} \times |I|)$, where $|I|$ designates the number of distinct lexical/semantic relationships, which simplifies to $O(N_{E})$ since $|I|$ is usually small.

- Step 6: Edge aggregation between each pair of index nodes in the $\text{SemIndex}^+$ graph can be performed in $O((N_{term} + N_{sense})^2)$ time where $N_{sense}$ designates the number of sense index nodes, which is the time needed to go through all pairs of index nodes in $\text{SemIndex}^+$,

- Step 7: Removing edge labels and string values from non-searchable (i.e., sense) nodes in $\text{SemIndex}^+$ can be executed in $O(N_{i} \times N_{sense})$, where $N_{i}$ designates the number of index and data edges.

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

6.1.2. Space Complexity

Our approach requires space to store the final $\text{SemIndex}^+$ graph $\tilde{G}_i$, which is also bounded by $O(N^2)$ space, since storing data nodes requires $O(|\Delta|)$ space, storing attribute nodes requires $O(|\Delta|)$ space, storing data edges (connecting data nodes with searchable term nodes through attribute nodes) requires in the worst case $O(|\Delta| \times (N_\Delta \times |\Delta|))$ space, storing index nodes requires $O(N_{term} \times N_{sense})$ space, and storing index edges (connecting pairs of index nodes) requires $O((N_{term} + N_{sense})^2)$ space (recall that only one edge exists between two nodes in $\text{SemIndex}^+$). 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. Query Processing

The complexity of our \( SL_{PSS} \) algorithm (cf. Fig. 10) which performs querying on \( SemIndex^{+} \), simplifies to \( O(M^2) \) in the worst case, and comes down to the sum of the complexities of its underlying functions, where for each query term:

- \( \text{getNodeID}() \) identifies the IDs of term nodes in the \( Lexicon \) corresponding to the query term, and thus requires in the worst case \( O(N_{\text{term}} + N_{\text{acc}}) \) time.
- \( \text{findShortestPaths}() \) runs Dijkstra's algorithm to identify the minimum paths at distance \( \ell \) from each of the starting term nodes \( N_{\text{term,term}} \) (i.e., homonymous query terms), which comes down to \( O(N_{\text{acc}}^2 \times \ell \times N_{\text{term,term}}) \) where \( N_{\text{acc}} \) designates the number of accessed index nodes in \( SemIndex^{+} \) when executing a query,
- \( \text{getDataNodeIDs}() \) identifies the IDs of data nodes in \( PostingList \) for each shortest path, requiring \( O(\|A\| + N_{\text{acc}}) \),
- \( \text{mergeAndRank}() \) merges and ranks data nodes with existing query answer nodes, by comparing the latter with node IDs in the \( PostingList \), thus requiring at most \( O(\|A\| \times N_{\text{acc}}) \) where \( N_{\text{acc}} \) designates the number of accessed data nodes when executing a query,
- \( \text{select}() \) selects data nodes as results, using range and/or \( k\text{NN} \) selection operators, requiring at most \( O(N_{\text{acc}}) \).

Hence, \( SL_{PSS} \)'s complexity comes down to that of function \( \text{findShortestPaths}() \) applied on \( k \) query terms while multitreading, which requires \( O((k \times (N_{\text{acc}}^2 \times \ell \times N_{\text{term,term}})) / \text{Threads}) \), where the algorithm allows as many simultaneous shortest path calls as there are threads\(^1\). It simplifies to \( O(k \times N_{\text{acc}}^2 / \text{Threads}) \), which can further simplify to \( O(N_{\text{acc}}^2 / \text{Threads}) \) given that \( k \) is usually limited (e.g., keyword queries on the Web are usually 2-3 words long [53]), and is bounded by \( O(N_{\text{acc}}^2) \) in the worst case (when applied on single thread/non-parallel systems).

7. Experimental Evaluation

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

7.1. Prototype System

We have implemented the \( SemIndex^{+} \) framework using open source technologies, namely: Java as the programming platform, MongoDB\(^2\) to handle the textual collection, WordNet 3.0 as the reference knowledge base, and MySQL 5.6 to persist the graph structures of \( SemIndex^{+} \). The physical design of \( SemIndex^{+} \) is shown in Fig. 11. We chose to handle it using a well known RDBMS (i.e., MySQL) to take advantage of its different useful features, including concurrency control and powerful index and memory management: allowing bulk index data loading and fast query execution\(^3\). Note that \( SemIndex^{+} \)'s physical design is independent of the DB structure and system used, and can be built directly on top of the file system, or using any other DB system. The prototype system is available online\(^4\).

---

1. In the physical implementation of the algorithm, thread scheduling and parallel execution are left to the operating system.
2. NoSQL DB using BSON binary format for storing documents in the JSON format, [https://www.mongodb.com/](https://www.mongodb.com/)
3. Preliminary experiments showed that handling \( SemIndex^{+} \) using MySQL was more efficient in build time and query execution time, compared with MongoDB, which we will highlight in a dedicated study.
4. Available at: [http://sigmappr.acm.org/Projects/SemIndex/](http://sigmappr.acm.org/Projects/SemIndex/)
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 quality of returned results. To do so, we varied the size of the input textual collection $\Delta$ by generating different extracts w.r.t. its total size (considering 10%, 20%, ..., or 100% of $\Delta$). We also varied 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 $<\Delta$ chunk ; KB chunk>, we evaluated each of the above four criteria by varying related parameters.

We used the IMDB movies dataset as an average-scale input textual collection, including attributes movie_id and (title, year, plot, genre, info) with a total size of around 75 MBytes including more than 7 million data (movie) objects. Three versions of the movies data collection were considered: the original (structured) version (cf. Table 1), a flat version where all attribute contents were concatenated in one column (cf. Table 2), and a partly structured version where certain attribute contents were randomly combined or omitted (cf. Table 3). In the remainder of this paper, we utilize the NoSQL version of IMDB movies to perform our experiments. WordNet 3.0 has a total size of around 26 Mbytes, including more than 117k synsets (senses). IMBD and WordNet chunk characteristics are summarized in Appendix I. Tests were carried out on a PC with an Intel i7 system with 2.9 GHz CPU, and 8GB RAM.

7.3. Index Building Time

Fig. 12a shows the total time required to build $\text{SemIndex}^+$ while varying both IMBD and WordNet chunks. 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). We also measured the total time required to build the traditional inverted index (which we note $\text{InvIndex}$) while varying IMDB chunk size (cf. Fig. 12.c). While both indices require linear building time, yet $\text{SemIndex}^+$ requires almost twice ($\times 2$) as much build time as $\text{InvIndex}$. Also, by disregarding the lemmatization phase in building $\text{InvIndex}$ (which can be ignored following the data manager’s preference: storing words in their actual (which complies with our complexity analysis). We also measured the total time required to build the traditional inverted index (which we note $\text{InvIndex}$) while varying IMDB chunk size (cf. Fig. 12.c). While both indices require linear building time, yet $\text{SemIndex}^+$ requires almost four times ($\times 4$) greater than that of $\text{InvIndex}$. This is encouraging since even the fastest inverted index creation time is only (at best) four times lesser than the creation time of $\text{SemIndex}^+$. The reasons for this are: i) the lightweight physical design of $\text{SemIndex}^+$ which can be easily created using fast legacy database technology, and ii) the sheer difference in size between the textual data collection (IMBD movies) and the reference knowledge graph (WordNet), which renders the build time of $\text{SemIndex}^+$ mostly dependent on IMDB 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.

---

1 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.

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

3 Recall that $\text{InvIndex}$ does not incorporate semantic knowledge and thus is not affected by WordNet chunk size variations.

---

**Fig. 12.** Breakdown of $\text{SemIndex}^+$ build time, compared with $\text{InvIndex}$ build time.
7.4. Index Size and Characteristics

Regarding SemIndex+ size, Fig. 13.a shows that the SemIndex+ graph size 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). The characteristics of SemIndex+ chunks are shown in Fig. 13.b (and Appendix I), where each chunk is generated by merging the corresponding \( \Delta \) chunk : KB chunk> doublet (for instance, the 10% SemIndex+ chunk is generated by merging the 10% \( \Delta \) chunk with the 10% KB chunk, and so forth).

![Fig. 13. SemIndex+ size characteristics and comparison with InvIndex size.](image)

First, results show that the number of nodes in the SemIndex+ graph increases almost linearly w.r.t. IMDB chunks size. Second, 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”, “adyer”, “moustafa”, etc.). We are currently investigating ways to further alleviate the missing terms problem, using dedicated language processors and multilingual dictionaries, which will be covered in an upcoming study.

In addition, we have also measured the characteristics and size of InvIndex in comparison with SemIndex+ (cf. Fig. 13.c). 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+ (which was 4 times larger) compared with InvIndex. This is due to the difference in sizes between the textual data collection (IMDB movies) and the knowledge graph (WordNet) used: WordNet (\( \approx 26 \) MBytes) is almost 1/3rd the size of IMDB (\( \approx 75 \) 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 categorized following four main criteria shown in Table 4: i) unrelated queries with varying number of attributes, ii) unrelated queries without attributes, iii) expanded queries with varying number of attributes, and iv) expanded queries without attributes. The first query group \( Q1 \) consists of queries with varying numbers of selection terms (keywords) from 1-to-5, where all terms are different and all queries are unrelated, such that the number of attributes per query varies from 1-to-3. The second query group \( Q2 \) consists of queries made of unrelated terms without attributes (i.e., queries targeting the whole textual contents of the searched objects). Query groups \( Q3 \) and \( Q4 \) are comparable to \( Q1 \) and \( Q2 \) respectively, except that queries are related such that each query expands its predecessor by adding an additional selection term to the latter. In other words, groups \( Q1 \) and \( Q3 \) consist of NoSQL keyword queries whereas \( Q2 \) and \( Q4 \) consist of “traditional” unstructured keyword queries.

Each query was tested on every one of the 100 combinations of SemIndex+ generated by combining the different chunks of IMDB movies data collection (10%, 20%, ..., 100%) with every chunk of WordNet (10%, 20%, ..., 100%), at link distance threshold values varying from \( \ell = 1 \) to 5 (i.e., increasing semantic coverage).

7.5.1. SemIndex+ Query Processing Time

On the one hand, the graph in Fig. 14.a plots SemIndex+’s query execution time averaged over all queries, considering different IMBD and WordNet chunk sizes, with a fixed number of query terms \( k \) and a fixed link distance threshold \( \ell \).
Results show that query execution is linear in both IMDB and WordNet chunk sizes, and thus is quadratic w.r.t. both of them (verifying our complexity analysis). On the other hand, the graphs in Fig. 14.b highlight the effects of varying the number of query terms \( k \) and varying link distance \( \ell \) w.r.t. fixed IMDB and WordNet chunk sizes. 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 SemIndex+’s querying algorithm in navigating the edges (pairs of nodes) of the SemIndex+ graph.

**Table 4.** Sample test queries used in our experiments.

<table>
<thead>
<tr>
<th>Group Q1</th>
<th>Unrelated queries with varying number of attributes</th>
<th>Group Q2</th>
<th>Unrelated queries without attributes¹</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1_1</td>
<td>( \sigma_{\text{title}}(\text{“time”}) \Delta_{\text{IMDB}} )²</td>
<td>Q2_1</td>
<td>( \sigma_{\text{title}}(\text{“music”}) \Delta_{\text{IMDB}} )</td>
</tr>
<tr>
<td>Q1_2</td>
<td>( \sigma_{\text{title}}(\text{“love”}, \text{“date”}) \Delta_{\text{IMDB}} )</td>
<td>Q2_2</td>
<td>( \sigma_{\text{title}}(\text{“romance”}, \text{“dinner”}) \Delta_{\text{IMDB}} )</td>
</tr>
<tr>
<td>Q1_3</td>
<td>( \sigma_{\text{title}}(\text{“fly”}, \text{“power”}) \sigma_{\text{plot}}(\text{“man”}) \Delta_{\text{IMDB}} )</td>
<td>Q2_3</td>
<td>( \sigma_{\text{title}}(\text{“news”}, \text{“party”}, \text{“home”}) \Delta_{\text{IMDB}} )</td>
</tr>
<tr>
<td>Q1_4</td>
<td>( \sigma_{\text{title}}(\text{“robot”}, \text{“human”}) \sigma_{\text{plot}}(\text{“war”}, \text{“world”}) \Delta_{\text{IMDB}} )</td>
<td>Q2_4</td>
<td>( \sigma_{\text{title}}(\text{“war”}, \text{“cowboy”}, \text{“pacifikop”}, \text{“idol”}) \Delta_{\text{IMDB}} )</td>
</tr>
<tr>
<td>Q1_5</td>
<td>( \sigma_{\text{title}}(\text{“family”}) \sigma_{\text{plot}}(\text{“mafia”}) \sigma_{\text{plot}}(\text{“kill”}) \sigma_{\text{plot}}(\text{“time”}) \Delta_{\text{IMDB}} )</td>
<td>Q2_5</td>
<td>( \sigma_{\text{title}}(\text{“city”}, \text{“group”}, \text{“fat”}) \Delta_{\text{IMDB}} )</td>
</tr>
</tbody>
</table>

Note that varying the number of attributes did not show any detectable impact on query execution time since, they almost do not affect the size of SemIndex+ neither do they affect the complexity of query search algorithm (compared with the relatively huge numbers of data and index nodes involved).

![Fig. 14. Query execution time, using SemIndex+’s SL_PSS algorithm, averaged over all queries.](image)

**7.5.2. Comparing Query Processing Time with Alternative Solutions**

We ran the same querying tasks using four alternative querying approaches adapted from the literature: InvIndex [62], QueryRelax [64], QueryDisam [7], and QueryRefine [67] (cf. Appendix III), and compared the obtained query time

---

¹ Recall that \( \perp \) designates a data object’s combined textual content from all attributes.

² Recall that we utilize the NoSQL version of IMDB movies to perform our experiments (cf. Section 7.2).
results with SemIndex+’s querying algorithm. Fig. 15 and Fig. 16 show average query execution time, plotted by varying the number of query terms $k$ (in Fig. 15) and SemIndex+ link distance threshold $\ell$ (in Fig. 16). First, results show that SemIndex+ and its alternatives have very close query time levels when link distance $\ell$ is small ($\ell=1$ and $\ell=2$), such that SemIndex+ time increases as link distance increases (whereas the other algorithms’ query time is naturally invariant w.r.t. variations in $\ell$, cf. Fig. 16). Second, Fig. 15 shows that query time for SemIndex+ and all three alternative solutions (namely QueryDisam and QueryRefine) almost linearly increases with the number of query terms $k^1$, such that the pace of SemIndex+’s time increase is relatively low with small link distances ($\ell=1$ and $\ell=2$) and the pace augments when reaching higher link distance thresholds ($\ell=3$–to–$5$). Third, one can realize that the most time consuming alternative approaches are QueryDisam and QueryRefine: i) QueryDisam and QueryRefine are more time consuming than SemIndex+ when the latter is run with link distances ($\ell \leq 3$), but they are both surpassed by SemIndex+ when the latter is run with higher link distances ($\ell=4$ and $\ell=5$). Here, QueryDisam’s computational overhead is due to the complexity of the query keywords’ semantic disambiguation process, whereas QueryRefine’s overhead$^2$ is due to post-processing the first round of query results in order to refine/rewrite query keywords accordingly. Algorithms InvIndex and QueryRelax share very close time levels and are the most efficient among the five solutions (including SemIndex+), running most queries almost instantaneously (under 0.14 seconds). This is expected since both approaches perform traditional syntactic query processing (and do not involve computationally expensive semantic processing) with one difference: InvIndex runs on the original query terms, whereas QueryRelax runs on an expansion of the original terms (adding the terms’ synonymous words to the original keyword query).

Fig. 15. Comparing SemIndex+ query time with four alternative solutions while varying the number of query terms $k$ and fixing link distance $\ell$ (the latter affecting SemIndex+ only).

Fig. 16. Comparing SemIndex+ query time with four alternative solutions while varying link distance threshold $\ell$ (affecting SemIndex+) and fixing the number of query terms $k$.

7.5.3. Discussion

By comparing SemIndex+’s query execution time with existing alternative solutions (InvIndex, QueryRelax, QueryDisam, and QueryRefine), we can highlight various observations: i) SemIndex+ is more computationally

---

1 Both InvIndex and QueryRelax time levels increase with $k$, even though these are not clearly visible in the graphs of Fig. 15 and Fig. 15 due to their scale. The reader can refer to the actual data behind the graphs at: http://sigappfr.acm.org/Projects/SemIndex/

2 QueryRefine’s time shown in Fig. 16 and Fig. 17 does not encompass the time it took the testers to manually choose the new query terms (which we did not consider to be part of the algorithm itself), but only considers actual algorithm (CPU and SQL) execution time.
expensive than syntactic solutions such as InvIndex and QueryRelax, ii) Involving query disambiguation (QueryDisam) is clearly computationally expensive (due to the complexity of the word sense disambiguation process applied on the query keywords) and hinders query performance, iii) Involving query refinement (through QueryRefine, producing the first round of results, allowing the user to refine query keywords, and then producing the second round of results) adds computational overhead, iv) The time to navigate the semantic graph, following the allowed link distance $t$, remains the foremost determining factor in SemIndex+ query execution time. SemIndex+ executes faster than QueryDisam and QueryRefine with low $t$ ($\leq 3$) but then requires more time than the latter when $t$ increases (to $t=4$ and 5), v) SemIndex+ search can benefit from parallelization, and can execute even faster when run on more powerful parallel processing systems (such as advanced multi-core, cluster, or grid computing platforms). We omit the latter experimental results here for ease of presentation and report the detailed evaluation of parallelization and its impact on SemIndex+ to a later dedicated study.

7.6. Query Result Evaluation

7.6.1. Result Quality Evaluation Metrics

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 and mapped them against user agreement (user judgments, utilized as golden truth) evaluating the quality of the answers produced by the system by computing precision, recall, $f$-value, and mean average precision metrics commonly utilized in IR evaluation [65]. Formally:

$$\text{PR} = \frac{a}{a+b} \in [0,1]$$

$$\text{R} = \frac{a}{a+c} \in [0,1]$$

$$F-\text{Value} = \frac{2 \times \text{PR} \times \text{R}}{\text{PR} + \text{R}} \in [0,1]$$

(10)

where $a$ is the number of retrieved data objects that indeed correspond to the query’s result list (correctly retrieved), $b$ is the number of retrieved data objects that do not correspond to the query’s result list (wrongly retrieved), and $c$ is the number of data objects that are not retrieved, although they correspond to the query’s result list (data objects that should have been retrieved). $F$-value represents the harmonic mean of precision and recall, such that high precision and recall, and thus high $f$-value characterize good retrieval quality [65]. Also, we employed mean average precision (MAP) to evaluate the ranking of relevant results w.r.t. non-relevant ones in the query result list:

$$\text{MAP} = \frac{\sum_{j=1}^{N} (\text{PR}[j] \times \text{rel}[j])}{N} \in [0,1]$$

(11)

where $n$ be the number of hits (i.e., returned data objects) in the query result list, $\text{PR}[j]$ is precision at hit $j$, $\text{rel}[j]$ is equal to 1 if the $j$th data object in the result list is relevant and 0 otherwise, and $N = a+c$ is the total number of data objects in the data collection which are relevant for the query. MAP is maximum, i.e., $= 1$, when the system retrieves all relevant data objects (i.e., recall $= 1$) and ranks them perfectly: all of them appearing before non-relevant data objects in the query list (i.e., precision at $N$th hit $= 1$); and decreases as more non-relevant data objects are introduced before relevant ones in the result list.

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 100 results (movie objects returned) per query (produced by SemIndex+ and its 3 alternatives). Here, only queries consisting of two keywords or more were considered, given that 1 single term queries (e.g., $Q_i \_1$) were deemed two fuzzy and coarse-grained for the testers to judge their results1. Query results 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 scores2 among testers for each of the rated query answers, and subsequently selected the top 100 hundred answers per query having the highest average inter-tester correlation scores3, which we utilized as the experiment’s golden truth.

---

1 A great many movies can be retrieved as answers for query $Q_1 \_1$: $\sigma_{\text{title} \sim \text{“time”}} \Delta \text{MEN}$ given single term “time” alone is too broad for human testers to make sense of the query. The same goes for the other three single term queries: $Q_2 \_1$, $Q_3 \_1$, and $Q_4 \_1$ (Table 4).

2 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.

3 Having average inter-tester PCC score $\geq 0.5$. 
7.6.2. Comparing SemIndex+ Result Quality with Alternative Solutions

Fig. 17 shows the precision, recall, f-value, and MAP results obtained with SemIndex+’s querying algorithm and alternative solutions: InvIndex [62], QueryRelax [64], QueryDisam [7], and QueryRefine [67]. Results averaged per link distance $\ell$ and number of query terms $k$ are provided in Table 5. These highlight several observations.

1) Precision and recall with link distance: One can realize that precision levels with SemIndex+ while fluctuating, generally increase with link distance ($\ell$) until reaching $\ell = 3$ or $\ell = 4$ where precision starts to decrease toward $\ell = 5$. However, one can realize that recall levels steadily increase with $\ell$ (with reduced fluctuation compared with precision). On the 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 $\ell > 1$). On the other hand, this also shows that over-navigating the SemIndex+ graph to link terms with semantically related ones located as far as $\ell \geq 3$ 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 $\ell = 5$ 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.). Many such examples occurred when running multiple term queries such as $Q_{1..4}$ (consisting of terms “robot”, “human”, “war”, “world”1), where movies like The Taking of Pelham One Two Three2 and Showtime3 (among others) where returned as results by SemIndex+’s querying algorithm when reaching $\ell = 5$. Such results were deemed not relevant by the testers since they do not correspond to the semantics of the query.

2) Precision and recall with the number of query terms: Here, one can realize that precision levels tend to stagnate or even decrease when increasing the number of terms $k$ – with queries having low link distance thresholds ($\ell \leq$ 2-or-3) – whereas precision tends to increase with the increase of $k$ – with queries having higher link distances ($\ell \geq$ 3-or-4). A similar behavior can be seen with recall levels: using more query terms (increasing $k$) produces lesser results when link distance is low. For instance, running query $Q_{1..4}$ with link distance $\ell = 1$ requires the returned movie objects to contain exact occurrences of all 4 query terms: “robot”, “human”, “war” and “world”, hence in our case producing zero (no) results whatsoever with all four SemIndex+ algorithms. Yet, as link distance increases, more and more (semantically related) results are retrieved (i.e., 0, 5, 253, and 1363 results were produced by SemIndex+ as answers for query $Q_{1..4}$ with $\ell = 2$, 3, 4, and 5 respectively). In other words, as $\ell$ increases, so do the chances of producing more results with (lower and especially) higher $k$ values, which in turn improves both precision (retrieving more relevant results) and recall (missing out less relevant results).

3) Regarding f-value and MAP, levels clearly increase with the increase of link distance $\ell$, whereas they show the same fluctuating behavior with the increase of the number of keywords $k$ as mentioned and discussed in the previous paragraph. First, f-value levels confirm the precision and recall levels obtained above, where the determining factor affecting retrieval quality remains link distance $\ell$, whereas an increase in the number of keywords $k$ tends to reduce system recall with lower values of $\ell$ and increase recall with higher values of $\ell$. Second, MAP levels seem to concur with those of f-value, such that the ranking of relevant results compared with non-relevant ones in the queries’ result lists seems to increase with the increase of $\ell$ and fluctuate (based on the values of $\ell$) with the increase of $k$. In other words, increasing $\ell$ not only allowed retrieving more relevant results, but also allowed dropping non-relevant ones (from the selected top 100 results of the query result list), and consequently improved the ranking of relevant results w.r.t non-relevant ones in the query result list.

4) Comparison with alternative solutions: First, considering all four metrics, one can realize that SemIndex+ performs similarity to alternative solutions at lower link distances ($\ell \leq 2$), and then increasingly surpasses the latter as $\ell$ increases ($\ell \geq 3$). This emphasizes the central impact of link distance in improving SemIndex+ performance (highlighted above). Second, considering results compiled over all link distances combined, one can clearly realize that SemIndex+ surpasses its alternatives considering both f-value and MAP, i.e., in both result quality and ranking. InvIndex naturally produced the worst f-value and MAP results since it is not semantic aware and only returns exact (syntactic) matches to query terms.

1 Even though testers had difficulty agreeing on the results of single keyword queries as mentioned previously, yet such cases occurring at link distance $\ell = 4$ or 5 were clearly deemed irrelevant by all testers.
2 Query $Q_{1..4}$: $\{\text{robot}, \text{human}\} \circ \{\text{war}, \text{world}\}$
4 2002 comedy movie starring Eddy Murphy and Robert De Niro, about police officers starring in a reality TV show.
7.7. Discussion

To sum up, we evaluate in Table 6 the ratio (expressed in percentage) of increase in query execution time, as well as the ratio (percentage) of increase in query result quality (considering average MAP scores) when using SemIndex+’s querying algorithm versus alternative solutions (detailed results are provided in Appendix 1):
Results in Table 6 highlight various observations: i) SemIndex+ querying requires an average 1701.74%, 1622.48%, 26.99%, and 21.83% more processing time than InvIndex, QueryRelax, QueryDisam, and QueryRefine respectively; ii) SemIndex+ improves query result quality levels by 543.67%, 472.64%, 238.31%, and 156.31% compared with InvIndex, QueryRelax, QueryRefine, and QueryDisam respectively; iii) SemIndex+ is costlier in computation time compared with its alternatives, nonetheless, it is also clearly and significantly more effective in producing higher quality results; iv) SemIndex+ can be most clearly appreciated when compared with QueryDisam where its improvement in query result quality (156.31%) clearly surpasses by 5.82 times its increased query execution time (26.88%), such that the time “effort” put in query execution time more than quintupled the system’s increase in query result quality. Even more pronounced, SemIndex+’s (21.83%) increase in query time w.r.t. QueryRefine produced a significant 21.65 times increase in result quality (472.64%).

Table 5. Average PR, R, f-value, and MAP (Appendix I)  

<table>
<thead>
<tr>
<th></th>
<th>PR</th>
<th>R</th>
<th>f-value</th>
<th>MAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>SemIndex+</td>
<td>0.2636</td>
<td>0.2085</td>
<td>0.2835</td>
<td>0.1939</td>
</tr>
<tr>
<td>InvIndex</td>
<td>0.2758</td>
<td>0.0327</td>
<td>0.1543</td>
<td>0.0273</td>
</tr>
<tr>
<td>QueryRelax</td>
<td>0.2179</td>
<td>0.1412</td>
<td>0.1796</td>
<td>0.0527</td>
</tr>
<tr>
<td>QueryDisam</td>
<td>0.2639</td>
<td>0.1281</td>
<td>0.1960</td>
<td>0.0570</td>
</tr>
<tr>
<td>QueryRefine</td>
<td>0.3762</td>
<td>0.0508</td>
<td>0.2135</td>
<td>0.0394</td>
</tr>
</tbody>
</table>

Table 6. Average \( \eta_{\text{Efficiency}} \) and \( \eta_{\text{Effectiveness}} \) (Appendix I)  

<table>
<thead>
<tr>
<th></th>
<th>( \eta_{\text{Efficiency}} \times 100 )</th>
<th>( \eta_{\text{Effectiveness}} \times 100 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>InvIndex</td>
<td>1701.74%</td>
<td>543.67%</td>
</tr>
<tr>
<td>QueryRelax</td>
<td>1622.48%</td>
<td>238.31%</td>
</tr>
<tr>
<td>QueryDisam</td>
<td>26.88%</td>
<td>156.31%</td>
</tr>
<tr>
<td>QueryRefine</td>
<td>21.83%</td>
<td>472.64%</td>
</tr>
</tbody>
</table>

8. Conclusion

This paper introduces SemIndex+, a framework for semantic-aware DB indexing and querying of unstructured (free-text), structured (relational), and partly-structured (NoSQL) textual data. At the indexer level, SemIndex+ creates a hybrid graph 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. The index is extended to handle varying multi-attribute data collections (using attribute-sensitive indexes), handling terms with missing semantic connections (i.e., missing terms), and introducing a model for weighting SemIndex+ entries (i.e., the graph’s nodes and edges). At the query processing level, the framework provides a parallelized (multithreaded) querying algorithm, coupled with a dedicated relevance scoring measure allowing to retrieve and rank relevant query answers. Our theoretical study and empirical evaluation highlight interesting observations: i) SemIndex+’s 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 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, iii) following the chosen link distance threshold, SemIndex+ is more or less costly in query processing time compared with alternative solutions (i.e., inverted index search, query relaxation, query disambiguation, and query refinement), nonetheless, iv) it is usually and significantly more effective in producing semantic-aware and higher quality results, such that SemIndex+’s improvement in query result quality clearly surpasses its increased query execution time.

We are currently exploring various techniques to improve SemIndex+ querying processing. First, we have started testing the querying algorithm on a parallel processing platform (using the open source Apache Hadoop\(^1\)). Preliminary results show a significant reduction in query execution time as the number of concurrent threads (multi-cores) increases toward matching the number of query terms and indexing nodes. We are also investigating DB index partitioning techniques (horizontal \([84]\), vertical \([1]\), and graph-based \([48]\)) to distribute SemIndex+ on multiple sites in order to allow more parallelization. We also plan to investigate query-driven optimization techniques, such as multi-term indexing \([86]\) (i.e., associating more than one single term with every entry in the seed inverted index, \(<\text{term1}, \text{term2}, ..., \text{termN}>, \text{docID}\text{s}>\)) which could be useful to reduce processing overhead for intersecting the inverted lists of multi-term queries, as well as incremental query result fetching \([75]\) (returning successive subsets of the partial results until the final result is “good” enough, such that the minimum number of responses or the minimum range satisfying the query are found). Extending SemIndex+ to perform incremental result fetching could prove to be efficient by limiting the breadth and depth of the SemIndex+ graph navigated by the querying algorithm.

Acknowledgements

This study is partly funded by the National Council for Scientific Research – Lebanon (CNRS-L), the Lebanese American University (LAU), the Fulbright Visiting Scholar program (sponsored by the US Department of State), and the Research Support Foundation of the State of Sao Paulo (FAPESP). We would also like to thank LAU graduates Christian Kallas and Marc Al Assad who helped implement different components of the SemIndex+ prototype system.

\(^1\) http://hadoop.apache.org/
References

Appendix I: Experimental Test Data Characteristics and Results

Table 7. Characteristics of IMDB movies table chunks.

<table>
<thead>
<tr>
<th>Chunk %</th>
<th>10%</th>
<th>20%</th>
<th>30%</th>
<th>40%</th>
<th>50%</th>
<th>60%</th>
<th>70%</th>
<th>80%</th>
<th>90%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (in MBs)</td>
<td>7.6483</td>
<td>15.0253</td>
<td>22.1164</td>
<td>29.7024</td>
<td>37.0124</td>
<td>43.8352</td>
<td>52.0641</td>
<td>58.5117</td>
<td>66.3724</td>
<td>74.1111</td>
</tr>
<tr>
<td># of data objects</td>
<td>14,304</td>
<td>28,608</td>
<td>42,912</td>
<td>57,217</td>
<td>71,521</td>
<td>85,825</td>
<td>100,130</td>
<td>114,434</td>
<td>128,736</td>
<td>143,043</td>
</tr>
<tr>
<td># of Attributes</td>
<td>76,511</td>
<td>124,139</td>
<td>198,644</td>
<td>246,714</td>
<td>294,858</td>
<td>351,973</td>
<td>408,081</td>
<td>465,181</td>
<td>523,283</td>
<td>581,382</td>
</tr>
<tr>
<td># of Terms</td>
<td>68,045</td>
<td>120,362</td>
<td>1,047,272</td>
<td>2,534,273</td>
<td>3,199,481</td>
<td>4,014,706</td>
<td>4,723,920</td>
<td>5,377,638</td>
<td>6,225,129</td>
<td>7,089,069</td>
</tr>
</tbody>
</table>

Table 8. Characteristics of WordNet chunks.

<table>
<thead>
<tr>
<th>Chunk %</th>
<th>10%</th>
<th>20%</th>
<th>30%</th>
<th>40%</th>
<th>50%</th>
<th>60%</th>
<th>70%</th>
<th>80%</th>
<th>90%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td># ofSense(Synsets)</td>
<td>11,738</td>
<td>21,475</td>
<td>35,212</td>
<td>46,949</td>
<td>58,686</td>
<td>70,423</td>
<td>82,186</td>
<td>93,941</td>
<td>105,668</td>
<td>117,973</td>
</tr>
<tr>
<td>Avg Branch</td>
<td>1.4333</td>
<td>1.6257</td>
<td>1.7533</td>
<td>1.9203</td>
<td>2.0967</td>
<td>2.2739</td>
<td>2.3736</td>
<td>2.5285</td>
<td>2.6673</td>
<td>2.8223</td>
</tr>
</tbody>
</table>

Table 9. Characteristics of SemIndex+ chunks.

<table>
<thead>
<tr>
<th>Chunk %</th>
<th>10%</th>
<th>20%</th>
<th>30%</th>
<th>40%</th>
<th>50%</th>
<th>60%</th>
<th>70%</th>
<th>80%</th>
<th>90%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (in MBs)</td>
<td>36.9219</td>
<td>68.2188</td>
<td>100.2613</td>
<td>132.3281</td>
<td>158.3594</td>
<td>202.0863</td>
<td>237.4686</td>
<td>273.5156</td>
<td>309.5026</td>
<td>359.625</td>
</tr>
<tr>
<td># of data nodes</td>
<td>14,904</td>
<td>28,809</td>
<td>62,912</td>
<td>117,311</td>
<td>172,713</td>
<td>228,115</td>
<td>283,517</td>
<td>339,919</td>
<td>395,321</td>
<td>450,724</td>
</tr>
<tr>
<td># of Attribute Nodes</td>
<td>76,351</td>
<td>124,139</td>
<td>198,644</td>
<td>246,714</td>
<td>294,858</td>
<td>351,973</td>
<td>408,081</td>
<td>465,181</td>
<td>523,283</td>
<td>581,382</td>
</tr>
<tr>
<td># of Attribute Node Occurrences</td>
<td>76,351</td>
<td>124,139</td>
<td>198,644</td>
<td>246,714</td>
<td>294,858</td>
<td>351,973</td>
<td>408,081</td>
<td>465,181</td>
<td>523,283</td>
<td>581,382</td>
</tr>
<tr>
<td># of Matching Index Term Nodes</td>
<td>18,098</td>
<td>36,202</td>
<td>56,306</td>
<td>76,405</td>
<td>96,503</td>
<td>116,601</td>
<td>136,703</td>
<td>156,802</td>
<td>176,901</td>
<td>197,009</td>
</tr>
<tr>
<td># of (Missing) Index Term Nodes</td>
<td>84,165</td>
<td>168,330</td>
<td>252,496</td>
<td>336,662</td>
<td>420,828</td>
<td>504,994</td>
<td>589,160</td>
<td>673,326</td>
<td>757,492</td>
<td>841,658</td>
</tr>
<tr>
<td>Total # of Nodes</td>
<td>98,902</td>
<td>166,508</td>
<td>234,011</td>
<td>291,517</td>
<td>349,023</td>
<td>406,529</td>
<td>464,035</td>
<td>521,541</td>
<td>579,047</td>
<td>636,554</td>
</tr>
</tbody>
</table>

Table 10. Characteristics of InvIndex (w.r.t. IMDB) chunks.

<table>
<thead>
<tr>
<th>Chunk %</th>
<th>10%</th>
<th>20%</th>
<th>30%</th>
<th>40%</th>
<th>50%</th>
<th>60%</th>
<th>70%</th>
<th>80%</th>
<th>90%</th>
<th>100%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (in MBs)</td>
<td>25.5781</td>
<td>49.6250</td>
<td>73.6719</td>
<td>98.7188</td>
<td>122.7656</td>
<td>147.8125</td>
<td>171.859</td>
<td>195.8066</td>
<td>220.0735</td>
<td>244.9844</td>
</tr>
<tr>
<td># of data objects</td>
<td>14,304</td>
<td>28,608</td>
<td>42,912</td>
<td>57,217</td>
<td>71,521</td>
<td>85,825</td>
<td>100,130</td>
<td>114,434</td>
<td>128,736</td>
<td>143,043</td>
</tr>
<tr>
<td># of Index Terms</td>
<td>73,255</td>
<td>115,570</td>
<td>157,882</td>
<td>188,589</td>
<td>230,904</td>
<td>254,894</td>
<td>282,939</td>
<td>309,049</td>
<td>330,155</td>
<td>356,994</td>
</tr>
</tbody>
</table>

1 Number of attribute occurrences in all data objects of the IMDB movies dataset.
2 The number of attribute nodes in our current SemIndex+ graph is equal to 5 (denoting title, year, plot, genre and info) regardless of chunk size, since attribute nodes are added once to the index (cf. ER model in Fig. 10) following their first occurrence in the dataset, and are then referenced as many times as needed to construct 3-uniform data edges (corresponding to the number of attribute node occurrences).
Table 11. Precision, recall, f-value, and MAP results obtained with SemIndex+ querying versus alternative solutions, averaged per link distance (t) and per number of terms (k).

<table>
<thead>
<tr>
<th>t=1</th>
<th>t=2</th>
<th>t=3</th>
<th>t=4</th>
<th>t=5</th>
<th>k=2</th>
<th>k=3</th>
<th>k=4</th>
<th>k=5</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>SemIndex</td>
<td>0.2758</td>
<td>0.3234</td>
<td>0.3539</td>
<td>0.3805</td>
<td>0.3189</td>
<td>0.4226</td>
<td>0.3632</td>
<td>0.2502</td>
<td>0.4184</td>
</tr>
<tr>
<td>InvIndex</td>
<td>0.2758 (invariant with t)</td>
<td>0.6532</td>
<td>0.5000</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.2758</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QueryRelax</td>
<td>0.2179 (invariant with t)</td>
<td>0.4243</td>
<td>0.1440</td>
<td>0.2333</td>
<td>0.0800</td>
<td>0.2179</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QueryDisc</td>
<td>0.2633 (invariant with t)</td>
<td>0.6032</td>
<td>0.1183</td>
<td>0.2108</td>
<td>0.1333</td>
<td>0.2633</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QueryRefine</td>
<td>0.3762 (invariant with t)</td>
<td>0.5562</td>
<td>0.3485</td>
<td>0.2667</td>
<td>0.3333</td>
<td>0.3762</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 12. Percentages of increase in query execution time and query result quality, when using SemIndex+ versus alternatives solutions.

<table>
<thead>
<tr>
<th>t=1</th>
<th>t=2</th>
<th>t=3</th>
<th>t=4</th>
<th>t=5</th>
<th>k=2</th>
<th>k=3</th>
<th>k=4</th>
<th>k=5</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>InvIndex</td>
<td>0.59%</td>
<td>501.50%</td>
<td>1273.76%</td>
<td>2600.51%</td>
<td>4195.00%</td>
<td>1231.06%</td>
<td>1393.27%</td>
<td>1321.47%</td>
<td>2020.51%</td>
</tr>
<tr>
<td>QueryRelax</td>
<td>5.90%1</td>
<td>464.52%</td>
<td>1185.03%</td>
<td>2426.09%</td>
<td>3917.60%</td>
<td>2032.25%</td>
<td>1127.12%</td>
<td>1480.63%</td>
<td>1286.97%</td>
</tr>
<tr>
<td>QueryDisc</td>
<td>-63.07%</td>
<td>-58.42%</td>
<td>-5.34%</td>
<td>86.07%</td>
<td>195.94%</td>
<td>23.07%</td>
<td>80.19%</td>
<td>-8.31%</td>
<td>86.81%</td>
</tr>
<tr>
<td>QueryRefine</td>
<td>-51.24%</td>
<td>-59.45%</td>
<td>-7.69%</td>
<td>81.45%</td>
<td>188.59%</td>
<td>35.48%</td>
<td>0.58%</td>
<td>5.66%</td>
<td>31.89%</td>
</tr>
</tbody>
</table>

Appendix II: Missing Terms Linkage Algorithm

Connecting unmapped searchable term nodes from $\tilde{q}_i, V_i^+$ to $\tilde{G}_A, V_A^+$, which we identify as missing terms in $\tilde{G}_A$, can be handled using an adaptation of distributional thesauri construction methods, e.g., [18, 87], to allow mining the syntactic/lexical relatedness between the missing terms and the index terms in $G_A$. Note that a distributional thesaurus is a thesaurus generated automatically from a given textual corpus (such as the Brown corpus [38], COCA) [31], or even the textual collection A being indexed), by finding words that co-occur together or that have similar contexts in the corpus. To that end, we introduce the MissingTerms_Linkage algorithm in Fig. 18. It accepts as input: the SemIndex+ graph $\tilde{G}_A$, a reference text corpus $C$, as well as two input parameters: $c_t$ and $c_t$ 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 $\tilde{G}_A$ (cf. Fig. 18.b, line 1), the algorithm creates a relatedness vector $RV(t_i)$ (line 3) to store the co-occurrence

1 A negative percentage of increase underlines a decrease percentage.
frequencies of surrounding terms. It identifies a window of size \( c_1 \), consisting of \( c_1 \) terms occurring to the left and right of the missing term in the reference corpus and which also exist among the index terms of \( \tilde{G}_M \) (line 4), and adds all window term frequencies to the relatedness vector (line 5). For example, suppose “horror” is a missing term, i.e., it does not appear in the WordNet lexicon extract but appears in object \( O_i \) of the data collection (cf. Fig. 5). If window size \( c_1 = 2 \), using the data collection itself \( \Delta_{\text{part}} \) as reference corpus, then terms “strange”, “car”, “thriller” and “cell” would be in the surrounding window of “horror”, and hence the relatedness score between “horror” and all these terms is increased. Once the vector has been obtained, we normalize vector scores w.r.t. 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 \( \tilde{G}_M \) (line 7). Then, a link is created to connect \( t_i \)’s term node with each top-ranked term \( t_k \) node in \( \tilde{G}_M \). These links are represented as index edges in \( \tilde{G}_M \), labeled: occurs-with (cf. Fig. 7 where term “horror” links with “car”, considered as its most related (top-ranked, i.e., highest co-occurrence frequency) term\(^1\).

\[
\begin{align*}
\text{Algorithm MissingTerms_Linkage} \\
\text{Input: } & \tilde{G}_M & \text{// SemIndex+ graph} \\
& C & \text{// Reference text corpus} \\
& c_1, c_2 & \text{// Input parameters: window size and top-ranked terms} \\
\text{Output: } & \tilde{G}_M & \text{// SemIndex+ graph with missing term links} \\
\text{Begin} \\
\text{For each missing term } t_i \text{ in } \tilde{G}_M & \quad 1 \\
\quad \{ & \quad 2 \\
\quad \quad \text{Create RV}(t_i) \text{ from } C \text{ given } \tilde{G}_M & \text{// Relatedness vector for term } t_i \quad 3 \\
\quad \quad \text{For each term } t_j \text{ in } \text{window}(t_i, c_1, C) & \quad 4 \\
\quad \quad \quad \{ & \quad 5 \\
\quad \quad \quad \quad \text{Add Freq}(t_j) \text{ to } RV(t_i) \} & \quad 6 \\
\quad \quad \quad \text{RV}(t_i) = \text{RV}(t_i) / \text{Max}(RV(t_i)) & \quad 7 \\
\quad \quad \text{T_i = set of } c_2 \text{ top-ranked terms in } RV(t_i) & \quad 8 \\
\quad \quad \text{For each term } t_k \text{ in } T_i & \quad 9 \\
\quad \quad \quad \{ & \quad 10 \\
\quad \quad \quad \quad \text{Create link between term nodes } t_i \text{ and } t_k \text{ in } \tilde{G}_M \} & \quad 11 \\
\quad \quad \quad \quad \text{Label the link “occurs-with”} & \quad 12 \\
\quad \} & \quad \}
\text{End}
\end{align*}
\]

Fig. 18. Pseudocode of the MissingTerms_Linkage algorithm.

Note that 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., 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). The algorithm’s impact on SemIndex+ querying effectiveness and efficiency will be evaluated in a dedicated future study.

**Appendix III: Alternative Algorithms used in our Experimental Study**

1. **Legacy Inverted Index Search (InvIndex)**

It’s a standard containment keyword-based query [62] that:

1- Retrieves textual identities that contain a set of keywords,
2- Queries the inverted (term, objectIDs[]) list with every term in the query, to identify objects IDs associated to all (or at least one) query terms (based on the query type at hand: conjunctive or disjunctive).
3- Assigns a score to every potential query answer (data object) considering a predefined relevance ordering scheme, in order to return the results ranked by their order of scores in ascending order.

\[1\] 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.
2. Query Relaxation (QueryRelax)

It consists in expanding the user query to include more interesting (semantically related) words [64], which would help identify more interesting results. The main steps of the algorithm can be described as follows:

1- Perform Part-Of-Speech tagging,
2- Select the most common sense for each token (e.g., based on WordNet’s usage frequency, computed based on the Brown text corpus),
3- For each selected sense $s_i$, include in the query: synonymous terms in the synset that is $s_i$, as well as the synonymous terms of all senses included in the direct semantic context of $s_i$ [82], i.e., senses that are related to $s_i$ via a semantic relationship (e.g., hypernymy, hyponymy, meronymy, etc.),
4- Run the resulting query using InIndex, as a traditional keyword containment query on the data collection’s inverted index (syntactic processing), and return the results to the user.

3. Query Disambiguation (QueryDisam)

It consists in applying word sense disambiguation [81] on query keywords, associating every keyword with its proper meaning (i.e., synset in WordNet) in order to execute the query accordingly [7]. The main steps of the algorithm can be described as follows:

1- Perform WSD on query terms using the simplified Lesk algorithm [3],
2- For each of the identified senses, include in the query: the sense’s synonymous terms,
3- Run the resulting query using InIndex, and return the results to the user.

4. Query Refinement (QueryRefine)

It consists in refining the user query to remove certain terms and include more (semantically) descriptive terms, which would help identify more interesting results [67]. The main steps of the algorithm can be described as follows:

1- Run the InIndex algorithm on the original query, and return the query results to the user,
2- For every term of the original query, provide the user with alternative suggestions in the form of a list of semantically related terms: the synonyms of all possible senses (synsets) of the query term,
3- Allow the user to add and remove terms to/from the original query, considering system provided suggestions (step 2) or her own choice of terms,
4- Run the refined query using InIndex, and return the results to the user.