HybIdx: Indexes for Processing Hybrid Graph Patterns Over Text-Rich Data Graphs

Technical Report

Günter Ladwig Thanh Tran

Institute AIFB, Karlsruhe Institute of Technology, Germany

{guenter.ladwig,ducthanh.tran}@kit.edu

Many databases today are text-rich in that they not only capture structured but also unstructured data. In this work, we propose a *full-text extension to SPARQL* capable of expressing different types of hybrid search queries over text-rich RDF graphs. We study existing indexing solutions to arrive at the conclusions that database extensions are less efficient than native solutions, and the most efficient indexes are limited w.r.t. the types of hybrid search queries they can support, i.e. *entity queries*. We propose an indexing solution call *HybIdx*, which is both efficient and versatile in terms of query type support, i.e. it supports full SPARQL graph patterns where keywords can appear in any position (*relational queries*). Experiments suggest that it can outperform the second best approach by up to three orders of magnitude for complex queries.

1 Introduction

Many databases today are *text-rich* in that they not only capture structured but also unstructured data, a combination called *hybrid data*. This is particular the case with RDF stores. Dealing with unstructured and structured data in an integrated fashion is a problem that is actively studied in the area of DB&IR integration [1]. This research recognizes that while keywords are necessary for querying textual data and also, can be used as an intuitive paradigm to query structured data [2, 3, 4], exploiting the full richness of structure information in hybrid data requires query expressiveness that goes beyond keywords. For this, different types of *hybrid query languages* have been proposed, including content-and-structure queries for XML document retrievals, XQuery Full-Text [5], and a combination of paths and keywords called FleXPath for XML data

retrieval [6]. However, we note there exists no standard hybrid query language for the more general graph-structured RDF data.

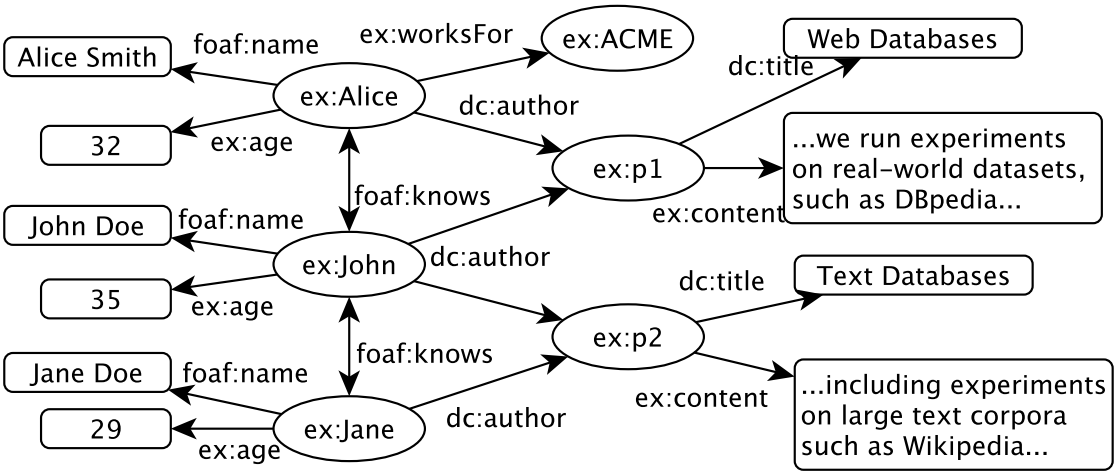
Moreover, while there are many proposals for processing hybrid queries and ranking hybrid results [7, 8, 9, 10, 6], the problem of *building indexes* for supporting efficient hybrid search is largely unexplored. We have identified two main directions of works. On the one hand there are *database extensions*, which add keyword search support to databases by using a separate inverted index for textual data [11]. The other direction is to build *native indexes* capturing both structured and textual data [8, 12, 13]. However, there is no work that systematically studies the differences among the various choices for native index design and in particular, differences between native indexes and database extensions.

To this end, we provide the following main contributions: (1) Firstly, we propose a *full-text extension to SPARQL*, the standard language for querying RDF. We discuss the various types of hybrid search queries that can be supported with this model, i.e. from unstructured to structured to hybrid queries, from attribute to entity to full relational queries that involve several types of entities, and from schema-based queries that require schema knowledge (attribute and relation names) to schema-agnostic queries. We show in the experiment that most information needs studied in existing benchmarks can be expressed as queries belonging to one of these types. (2) We propose a *general hybrid search index scheme* that can be used to specify access patterns needed to support these various query types. (3) We introduce *HybIdx* as one instance of this scheme. (4) We perform a *comprehensive experiment* using several benchmark datasets and queries to systematically study existing solutions and HybIdx in several scenarios, from the text-centric retrieval of documents in Wikipedia and TREC collections annotated with structured data to structure-centric retrieval of data in IMDB and YAGO up to "pure" hybrid data formed by combining Wikipedia and DBPedia. The main conclusions of this experimental study are: native solutions are faster than database extensions by up to an order of magnitude; native solutions that focus on one type of queries, i.e. entity queries, are fastest because of smaller index size. Compared to these, HybIdx provides superior performance for relational and document queries (outperforms the second best approach by up to three orders of magnitude) and yields results close to the ones achieved by the best "focused" solution for entity queries [8]. As opposed to these solutions, it is more complete regarding the types of hybrid search queries that can be supported.

Outline. In Sec. 2 we introduce the hybrid search query model and discuss the problem of hybrid search query processing. Sec. 3 discusses the different types of queries that can be expressed using this model and how they are supported by existing solutions. Sec. 4 presents our indexing solution. We present related work in Sec. 5, experimental results in Sec. 6 and conclusions in Sec. 7.

2 Hybrid Search

Data graphs have been used to model different kinds of data, from interconnected documents to hierarchical XML data to relational data. For dealing with structured data on



$Q_1\langle \mathsf{s}, \mathsf{p}, \textit{“dbpedia experiments”}\rangle$
$Q_2\langle \mathsf{s}, \textit{“name”}, \textit{“alice”}\rangle\langle \mathsf{s}, foaf{:}knows, \textit{“john”}\rangle \quad Q_3\langle \textit{“alice”}, \textit{“age”}, \mathsf{o}\rangle$
$Q_4\langle \mathsf{x}, \textit{“works for”}, \textit{“acme”}\rangle\langle \mathsf{x}, dc{:}author, \mathsf{p}\rangle\langle \mathsf{y}, \textit{“author”}, \mathsf{p}\rangle$
$Q_5\langle \mathsf{s}, dc{:}title, \textit{“databases”}\rangle\langle \mathsf{s}, \mathsf{p}, \textit{“experiments”}\rangle\langle \mathsf{x}, dc{:}author, \mathsf{s}\rangle$

Figure 1: Example data and queries.

the Web that can be conceived as graphs, the W3C introduced RDF[1] and SPARQL[2] as the main standards for data representation and querying, respectively. We build upon these standards.

2.1 Hybrid Data

We use RDF as the data model, omitting the special RDF semantics of blank nodes for the sake of generality:

Definition 1 (RDF Triple, RDF Term, RDF Graph)**.** *Given a set of URI references $\mathcal{U}$ and a set of literals $\mathcal{L}$, elements in $\mathcal{U} \cup \mathcal{L}$ are called* RDF terms*, $\langle s, p, o\rangle \in \mathcal{U} \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{L})$ is an* RDF triple*, and an* RDF graph *G is a set of RDF triples.*

Example 1. *An example RDF graph formed by 17 RDF triples is shown in Fig. 1. For presentation, URI references are shorten using the abbreviations for URI namespaces foaf:, dc: and ex:. For instance, dc: stands for `http://dublincore.org/documents/dcmi-namespace`. All the nodes in the graph without such a namespace prefix represent literals, 32 or John Doe for instance.*

With this model, real-world entities s are represented as URIs and via $\langle s, p, o\rangle$ triples, associated with *attribute values* (o as literals) or *relations* to other entities (o as entities). Clearly, this model can be used to capture structured data of different kinds as well as text-rich data. For instance, (structured) document entities can be represented as URIs and the textual content captured as an attribute value. Similarly, other types of entities

[1] `http://www.w3.org/RDF/`
[2] `http://www.w3.org/TR/rdf-sparql-query/`

with long textual descriptions can be captured via URIs associated with textual attribute values. In fact, some RDF resources are already text-rich as they are associated with long names and descriptions that can be decomposed into several words. Applying standard text processing techniques for extracting, tokenizing and stop word filtering, bags of words can be derived from RDF terms to deal with their text-rich nature:

Definition 2 (Keyword Term)**.** *The function* $text : \mathcal{U} \cup \mathcal{L} \rightarrow \mathcal{K}$ *maps a URI or a literal to a bag of words (also called* keyword terms*)* $K \in \mathcal{K}$*, where* $\mathcal{K}$ *is the set of all bags of keyword terms* $K = \{k_1, \ldots, k_i, \ldots, k_n\}, k_i \in \mathcal{W}$*, and* $\mathcal{W}$ *is the vocabulary of all keyword terms. As a shorthand we also define keyword terms over sets of RDF terms,* $text(T) := \bigcup_{t \in T} text(t)$*, where* $T \subseteq \mathcal{U} \cup \mathcal{L}$*.*

We model hybrid data as data graphs associated with a textual representation, as captured by the *text* function:

Definition 3 (Hybrid Data)**.** *Hybrid data is the tuple* $(G, text)$*, where* G *is the* RDF graph *and* text *the function mapping RDF terms in* G *bags of to keyword terms.*

Example 2. *The RDF graph in Fig. 1 represents structured data as triples but also captures textual data embedded in these triples. Literals representing attribute values for* dc:title *and* ex:content *for instance, contain a large number of words. Applying the* text *function to the URI* ex:Alice *yields the term* $\{alice\}$*. Similarly, applying* text *to the literal associated with the* ex:content *of* ex:p1 *yields the terms* $\{run, experiments, real\text{-}world, \ldots\}$*.*

2.2 Hybrid Query

The core feature of SPARQL is Basic Graph Pattern (BGP), which is composed of triple patterns $\langle s, p, o \rangle$, where each s, p and o is either a variable or a constant. As constants, RDF terms are specified in BGP queries such that matching results are corresponding RDF terms in the data. For instance, we specify the pattern $\langle s, ex{:}name, John\ Doe \rangle$, where s is a variable, to obtain the corresponding triple $\langle ex{:}John, ex{:}name, John\ Doe \rangle$. Given hybrid data where RDF terms and especially literals, contain a large number of words, it is desirable to query data not only based on entire RDF terms such as *John Doe* but some containing words, i.e. keyword terms, such as *John* or *Doe*, to obtain results with RDF terms that contain these words.

With the introduced *text* function that represents the textual data embedded in structured data and the resulting model of hybrid data, we can now query for RDF triples and subgraphs as well as the textual data contained in them. We extend the notion of BGP such that not only RDF terms but also keyword terms can be used as constants. This yields a kind of hybrid queries called *Hybrid Graph Pattern* (HGP):

Definition 4 (Hybrid Graph Pattern)**.** *Let* $\mathcal{V}$ *be the set of all variables. A* hybrid triple pattern $\langle s, p, o \rangle \in (\mathcal{V} \cup \mathcal{U} \cup \mathcal{K}) \times (\mathcal{V} \cup \mathcal{U} \cup \mathcal{K}) \times (\mathcal{V} \cup \mathcal{U} \cup \mathcal{L} \cup \mathcal{K})$ *is a triple where the subject, predicate and object can either be a* variable *in* $\mathcal{V}$ *or a* constant*. The latter is either an* RDF term *in* $\mathcal{U} \cup \mathcal{L}$ *or a* bag of keyword terms *in* $\mathcal{K}$*. A* hybrid graph pattern *is a set of hybrid triple patterns* $Q = \{h_1, \ldots, h_n\}$*.*

For brevity, we use the following syntactic convention in this paper: single letters stand for variables; to distinguish RDF terms from keyword terms, we put the latter in quotes.

Example 3. *Fig. 1 shows five example queries.* Q_1 *retrieves entities whose objects contain the keyword terms* dbpedia *and* experiments. Q_2 *and* Q_3 *also contain a keyword term at the predicate position, i.e.* name *and* age. *The second triple pattern in* Q_2 *involves a constant,* foaf:knows. *Whereas* Q_1 *and* Q_3 *are hybrid triple patterns,* Q_2, Q_4 *and* Q_5 *capture more complex hybrid graph patterns.*

2.3 Matching

The usual SPARQL semantics can be used for evaluating HGPs without keywords terms (i.e. those which are BGPs). Extending this semantics to incorporate keyword terms, a result to a HGP query can be defined as follows:

Definition 5. *Let* G *be an RDF graph,* Q *be a HGP query, and* $\mathcal{V}$ *be the set of all variables and* $\mathcal{K}$ *the set of all bags of keyword terms. Let* μ' *be the function that map elements* Q *to elements* G*:*

$$\mu' : \mathcal{V} \cup \mathcal{U} \cup \mathcal{L} \cup \mathcal{K} \rightarrow \mathcal{U} \cup \mathcal{L} \begin{cases} (1)\ v \mapsto \mu(v) & \text{if } v \in \mathcal{V} \\ (2)\ K \mapsto t & \text{if } K \in \mathcal{K} \\ (3)\ t \mapsto t & \text{if } t \in \mathcal{U} \cup \mathcal{L} \end{cases}$$

where the mapping $\mu : \mathcal{V} \rightarrow \mathcal{U} \cup \mathcal{L}$ *is employed to map variables in* Q *to RDF terms in* G*. A mapping* μ *is a* result *to* Q *if it satisfies* $\langle \mu'(s), \mu'(p), \mu'(o) \rangle \in G,\ \forall \langle s, p, o \rangle \in Q$*.*

In other words, computing results amounts to the task of *graph pattern matching*, where the query graph pattern is matched against the data graph and the results are matches to variables in the query. The matching of (1) variables v and (3) RDF terms t in the query to RDF terms in the data is analogous to BGP matching. The HGP extension to BGP matching is (2), the mapping of bags of keyword terms in the query, K, to the textual representation of RDF terms in the data, t. For this we propose the semantics commonly used in Information Retrieval (IR) tasks, the one based on *IR-style relevance*: an RDF term t is considered a match to K if $text(t)$ is relevant for K, otherwise t is not a match.

Example 4. Q_1 *involves a bag of keyword terms,* $\{dbpedia, experiments\}$*, thus it represents a hybrid triple pattern matching problem. A match to the variable* s *and* p *in the query is* ex:p1 *and* ex:content*, respectively, when the literal* $t =$ *...we run experiments... associated with them is relevant. This might be the case, depending on the IR ranking function discussed next, because its textual representation,* $text(t) = \{\ldots, we, run, experiments, \ldots\}$*, contains both the keyword term* dbpedia *and* experiments. *In the second triple pattern of* Q_2*, one constant is an RDF term,* foaf:knows*, which directly matches the same RDF term in the data.*

2.4 Ranking

There exist different IR models to compute the degree of relevance in which results may vary. Mainly, they employ TF-IDF [13] or probability-based [14] distance metrics between query and results and additionally, heuristics such as proximity or PageRank-based popularity [15].

To focus on the indexing problem studied in this paper, we make the general assumption that ranking approaches rely on some term-based scores for matches. This means for computing the relevance of $text(t)$ w.r.t. to K, the ranking function employs relevance scores for individual keyword terms $k \in K$, i.e. there is a function $score : text(\mathcal{U} \cup \mathcal{L}) \times \mathcal{W} \to \mathbb{R}^+$ that assigns a score to every possible pair of RDF term, its textual representation to precise, and keyword term. This is for instance the case with TF-IDF based approaches, where there is a TF-IDF weight for every document-term pair. This *term-based score assumption* is employed for the design of our rank-aware index, which provides an abstraction from the specific IR models used for ranking.

3 Processing Hybrid Queries

Existing solutions target specific types of queries. In this section, we show that the proposed hybrid search query model is sufficiently general to capture the main existing types of queries as particular kinds of HGPs.

3.1 Hybrid Query Types

Unstructured vs. Structured. Standard keyword queries are *unstructured*, which are sets of keywords, each of the form $K = \{k_1, \ldots, k_n\}$. As a HGP, every such query can be expressed as $\langle x_1, y_1, k_1\rangle, \ldots, \langle x_n, y_n, k_n\rangle$ where x_i and y_i, $1 \leq i \leq n$, are variables. A HGP that is a fully *structured* BGP is simply a pattern that does not involve the use of keywords, i.e. it is of the form $\langle x_1, y_1, z_1\rangle, \ldots, \langle x_n, y_n, z_n\rangle$ where x_i, y_i and z_i, $1 \leq i \leq n$, are variables or RDF terms.

Entity vs. Attribute vs. Relational. *Entity* queries, especially the kind that seeks for entities of the type document, capture a large fragment of real-world information needs commonly supported by standard IR solutions and Web search engines. Instead of searching for documents, Semantic Web search engines and recent IR solutions focusing on structured data, support the retrieval of entities in general. A HGP corresponding to this type is star-shaped, whose triple patterns share the same variable at their subject position, i.e. $\langle x, y_1, z_1\rangle, \ldots, \langle x, y_n, z_n\rangle$ where y_i and z_i, $1 \leq i \leq n$, are variables, RDF terms or keywords, and the "center node" variable, x, stands for the entities to be retrieved (see Q_1 and Q_2 in our example). An *attribute* query $\langle x, y, a\rangle$ retrieves the attribute value for a given entity and attribute. That is, x and y are RDF/keyword terms representing the given entity and attribute and a is a variable that captures the attribute value (see Q_3). Note that while other types of queries typically involve a variable at the subject position, this one explicitly specifies the subject entity and seeks for information about that entity. Thus, the ability to specify the subject using

simple keywords is crucial for this type. A *relational* query is more complex in that it involves several entities and their relations. That is, it is composed of triple patterns that have different entities at their subject position, and these entities are connected through relations as captured by the query triple patterns, i.e. queries of this type are of the general form $\langle x_1, y_1, z_1\rangle, \ldots, \langle x_n, y_n, z_n\rangle$ where x_i, y_i and z_i, $1 \leq i \leq n$, are variables, RDF terms or keywords (see Q_4, Q_5).

Schema-based vs. Schema-agnostic. Another dimension based on which queries can be distinguished is whether they involve schema information, i.e. attributes and relations. *Schema-agnostic* querying does not require users to precisely know the attributes and relations such that the query may contain variables or keywords at the predicate position: $\langle x_1, y_1, z_1\rangle, \ldots, \langle x_n, y_n, z_n\rangle$ where y_i, $1 \leq i \leq n$, is a variable or a keyword (see Q_1-Q_5). As opposed to that, y_i is an RDF term in *schema-based* queries (e.g. see second triple pattern in Q_2). Standard queries supported by database engines (e.g. SQL) are schema-based while BGPs supported by RDF stores might be schema-agnostic. As opposed to BGPs, users can specify not only variables but also keywords in the predicate position of patterns in HGPs.

Given a HGP, the matches are computed using two main operations, (1) retrieving data and (2) combining partial results. We now discuss the main existing solutions addressing these two tasks, indexes and join processing techniques in particular. We show their limitations in terms of the types of queries they can support and other drawbacks w.r.t. the processing of general hybrid search queries.

3.2 Indexes

Indexes are employed to enable quick lookup of (partial) results, given a key. In Fig. 2, we provide several example indexes to illustrate the solutions discussed in the following. Traditional solutions can be distinguished in those targeting *structured queries* and those that deal with *keyword queries*. Regarding the former, there are many types of indexes for different types of structured data, e.g. special indexes for geo-spatial or multi-dimensional data [16]. For graph-structured data, such as RDF, many indexes are *triple-based* in that they return triples as results, given the constant(s) specified in the triple pattern as key. Since constants can be specified in different positions, there are different combinations of keys that can be supported (Fig. 2b shows an index supporting the combination of predicate and object as key). Different indexes are created to support these different access patterns captured by the keys [17]. As opposed to that, the latter focusing on keyword queries is *entity-based* because it returns entities, given a keyword in the query as key. Traditionally, these indexes return document entities. Recently, they are applied to structured data to support keyword-based object retrieval [18], e.g. to return RDF entities (see Fig. 2a). While these solutions focus on general structured queries or unstructured entity queries, there are also proposals targeting the *hybrid case*. Here, we distinguish native approaches from database extensions.

Native Approaches.

(1)The *vertical index* (VI) [13] supports hybrid triple patterns that have an RDF term at the predicate position and keyword at the object position. That is, it assumes the

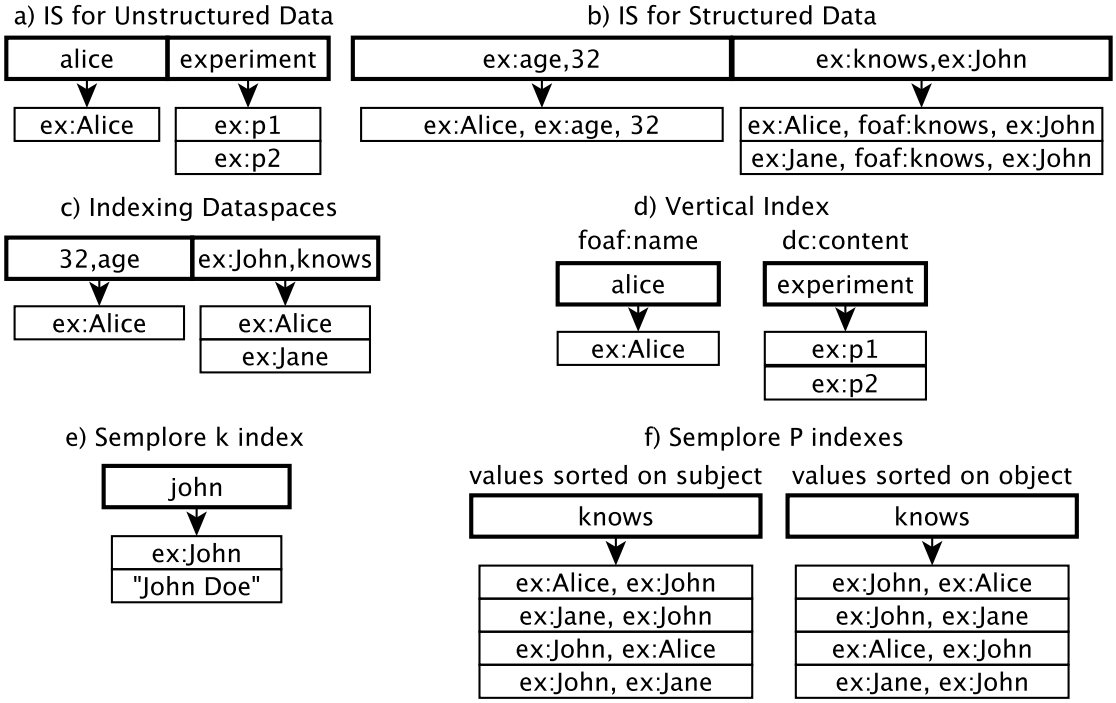


Figure 2: Example indexes.

predicate is known and thus, employs one index for every attribute in the dataset, e.g. for foaf:name. It is an entity-based index because instead of returning triples for the given attribute-value key, it provides matching entities as results (see Fig. 2d for results to the key foaf:name, "alice" and dc:content, "experiment"). Thus, this index only supports *schema-based entity queries* (to be precise, schema-agnostic queries are possible but not manageable because they require lookups to be performed on all indexes). Another drawback is that a large number of indexes have to be created, depending on the number of predicates to be supported for hybrid search. In their experiment, the authors only index a predefined set of popular attributes [13].

(2) The solution proposed for *indexing dataspaces* (ID) [8] supports hybrid triple patterns where keywords may occur both at the object and predicate position. The indexed keys are terms extracted from attribute values as well as combinations of terms extracted from both attribute names and values (see Fig. 2c). Just like VI, this solution takes advantage of the fact that the unit of retrieval is an entity, which is treated as a document such that existing inverted indexes originally built for documents are directly applicable. Thus, while it is *schema-agnostic*, it is limited to *entity queries*.

(3) *Semplore* (SE) [12] is the only solution supporting the more general *relational queries*, employing three indexes. The first simply maps keywords to the RDF terms they appear in, i.e. URIs and literals (see Fig. 2e). The others are to support hybrid triple patterns with keywords at the predicate position. They return objects and subjects of the matching triples. In order to enable fast merge joins (discussed next), the values

are sorted on the subjects for one index and on the objects for the other. The limitation of this solution is that while it enables the use of keywords at predicate position, it is *not entirely schema-agnostic* because it still requires the predicate to be specified, i.e. it cannot be a variable. Further, it is an entity-based solution, meaning that (partial) results returned are entities. As discussed in their paper, the authors show that when queries are indeed relational, i.e. contain patterns capturing relations, a relation expansion step involving a mass-union operator is needed, which "might lead to prohibitive I/O as it requires a large number of back-and-forth disk seeks" [12].

Database Extensions. While native approaches discussed before employ one single index solution, database extensions have separate indexes for dealing with structured and textual data [7, 9, 19, 6, 20]. We identify two main types of indexes employed by database extensions. Many RDF stores for instance, such as OWLIM [20], can be configured to support either of these two types. (1) The first [20] employs an indexing strategy that is analogous to the VI solution. It creates a separate index for every attribute. Thus, it shares VI's merits and drawbacks. (2) Just like the k index illustrated for Semplore in Fig. 2e, the second type maps keywords to RDF terms containing them [19]. Solutions built upon this are similar to Semplore in that they are capable of supporting *relational queries* but also, suffer from large joins. We discuss this in detail in the following.

3.3 Join Processing

Given the indexes for retrieving results for hybrid triple patterns, the processing of HGPs is similar to the processing of BGPs supported by RDF stores. Consequently, query optimization techniques such as join ordering or sideways information passing [17, 21] are applicable. Here, we focus the discussion on the aspects of join processing that are directly affected by the choice of indexes.

Entity-based vs. Triple-based. Note that the native solutions above employ the entity-based strategy. With this, entity queries, such as Q_2, can be efficiently processed by intersecting the lists of bindings for the variable s obtained for each of the patterns. Efficient bit set representations have been developed through the long history of IR research that allow for very fast intersection operations on lists of integers [22]. With triple-based solutions, intersections are performed on lists of tuples instead of single values. For this, there is a large body of database solutions for join processing that are applicable. Semplore and the database extensions discussed above that aim at full relational queries (type 2), however, require a mixture of the two strategies. The results they obtain for the structured query parts are triples, while results for keywords are entities (RDF terms in general). This mismatch requires additional joins to be performed to combine the two types of results. These additional joins are even needed for processing single hybrid triple patterns. To evaluate the second pattern in Q_2 for instance, they use the k index to retrieve all RDF terms matching "john", and one of the two P indexes (or a similar index that supports this access pattern, in the case of the database extensions) to obtain all triples matching $\langle \mathsf{s}, foaf{:}knows, o \rangle$, and finally, join these results on the variable o.

Top-k vs. No top-k. The index solutions mentioned above have not been studied

in the top-k setting. Especially when the use of keywords is involved, it is often only necessary to obtain the top-k ranked results. Top-k processing techniques can improve performance through early termination after obtaining the top-k results, *rank join* in particular [23]. The downside is that it requires inputs to be sorted on the score, which is usually achieved by indexing the data in a sorted fashion to avoid the high cost of online sorting. However, entries in the index are often sorted according to values instead of scores. Given sorted values, efficient *merge join* can be employed to obtain running times linear to input size. Further, employing rank join also means that the engine cannot take advantage of the indexes on the join variables, when available, to perform *index-based join*.

4 Hybrid Search Index

We propose a native indexing solution that fully supports the proposed query model. Compared to previous works, the main novelties are:

- *Full hybrid search support*: so far, the proposed native indexes focus on entity queries. Semplore is the only solution capable of answering relational queries. However, it does not support patterns where the predicate is a variable, and returns only entities as results. That is, it is not fully schema-agnostic and also does not support attribute queries.
- *Efficient hybrid search*: while the database extensions can support all the discussed types, they require a large number of joins due to the mixture of entity and tuple results. Our native solution supports all the query types without incurring this additional cost.
- *Top-k and no top-k*: our indexes are designed to include term score information. This is to support ranking schemes that conform with the term-based score assumption and top-k processing techniques that rely on these scores. As discussed, top-k rank join might be preferred over other join implementations, or vice versa, depending on the nature of the data and query. Our solution supports both top-k and non-top-k joins.

We first present a *general hybrid index scheme* based on which all the access patterns needed to support the proposed query model can be specified. Then, we present our solution *HybIdx* that instantiates this scheme.

4.1 Hybrid Index Schemes

An index is a data structure that enables lookups of values given a key. With respect to the data model, values that can be indexed correspond to elements in the RDF graph, i.e. RDF terms and triples (or even subgraphs). The keys are RDF or keyword terms. Conceptually, index solutions can be conceived as particular index schemes consisting of key-value lookup patterns:

Definition 6 (Key/Value Pattern)**.** *Let atomic* RDF key patterns *be the sets* $\mathbf{t}_s^{RDF}$, $\mathbf{t}_p^{RDF}$ *and* $\mathbf{t}_o^{RDF}$ *of all RDF terms that appear at the position s, p and o of the triples*

$\langle s, p, o\rangle \in G$, *respectively. Likewise, let atomic* keyword key patterns *be the sets* $\mathbf{t}_s^k$, $\mathbf{t}_p^k$ *and* $\mathbf{t}_o^k$ *of all keyword terms that appear in* $text(s)$, $text(p)$ *and* $text(o)$ *of all triples* $\langle s, p, o\rangle \in G$, *respectively. Correspondingly, the combined sets of atomic RDF and keyword key patterns are denoted as* $\mathbf{t}_s, \mathbf{t}_p$ *and* $\mathbf{t}_o$. *A* compound key pattern *is the tuple* $\mathbf{t}_1, \mathbf{t}_2, \mathbf{t}_3$, *where* $\mathbf{t}_i, 1 \leq i \leq 3$, *might be* $\mathbf{t}_s, \mathbf{t}_p$ *or* $\mathbf{t}_o$, *or simply unspecified. A* value pattern *is the set of all RDF triples in* G, $\langle \mathbf{s}, \mathbf{p}, \mathbf{o}\rangle$, *or the set of all RDF terms* $\mathbf{t}_s^{RDF}$, $\mathbf{t}_p^{RDF}$ *or* $\mathbf{t}_o^{RDF}$ *as defined before.*

Note that compound key pattern is simply a generalization of atomic key pattern, i.e. it is atomic when two elements are unspecified.

Definition 7 (Index Scheme)**.** *An* index scheme *is a set of key-value patterns* **key** $\mapsto$ **value**, *where* **key** *denotes the set of all compound key patterns and* **value** *the set of all value patterns.*

With compound key patterns, RDF/keyword terms at position s, p or o or a combination of them can be used as keys. In fact, every key represents a query triple pattern: while atomic key pattern captures keys that correspond to triple patterns with exactly one RDF/keyword term, triple patterns with several RDF/keyword terms are supported by compound key patterns. Matches to these patterns are returned as the result of an index lookup. However, instead of the matching triples, any RDF term at position s, p or o of these triples can be specified as the value to be returned. For instance, the key-value pattern $\mathbf{t}_o \mapsto \mathbf{t}_s^{RDF}$ supports the retrieval of RDF terms $t \in \mathbf{t}_s^{RDF}$ that appear at the subject position of triples having an object matching the given key $t \in \mathbf{t}_o$. As opposed to that, the pattern $\mathbf{t}_o, \mathbf{t}_p \mapsto \langle \mathbf{s}, \mathbf{p}, \mathbf{o}\rangle$ enables the retrieval of RDF triples having an object matching $t_o \in \mathbf{t}_o$ and a predicate matching $t_p \in \mathbf{t}_p$.

Further, we leverage *prefix lookups* to reduce the number of key-value patterns needed to support the many kinds of hybrid search queries. This lookup capability is for instance, supported by standard implementations of the inverted index. With this, we can further distinguish between standard compound key patterns $\mathbf{t}_s, \mathbf{t}_p, \mathbf{t}_o$ and *prefix compound key patterns* $\mathbf{p}(\mathbf{t}_s, \mathbf{t}_p, \mathbf{t}_o)$. With the latter, lookups are possible even when only a prefix of the key is specified. For a compound key with n elements, the prefix of n with length i is simply n without the last $n - i$ elements. For instance, given $\mathbf{p}(\mathbf{t}_o, \mathbf{t}_p) \mapsto \langle \mathbf{s}, \mathbf{p}, \mathbf{o}\rangle$, the prefix key with length one is simply $t_o \in \mathbf{t}_o$, which yields all triples with an object matching t_o. Conceptually, a prefix pattern $\mathbf{p}(\mathbf{t})$ can be treated as a set of patterns, i.e. all patterns that correspond to $\mathbf{p}(\mathbf{t})$ or any prefix of $\mathbf{p}(\mathbf{t})$. For instance, the set of patterns represented by $\mathbf{p}(\mathbf{t}_o, \mathbf{t}_p)$ comprises $\mathbf{t}_o, \mathbf{t}_p$ and $\mathbf{t}_o$.

4.2 HybIdx: Hybrid Search Index

Our solution is a generalization of indexing approaches proposed for unstructured and structured data, i.e. those that map (1) keyword terms to documents [22] or (2) compound RDF terms representing triple patterns to RDF triples [17]. We use the general key pattern $\mathbf{t}$, which can be either a keyword key pattern or an RDF key pattern. Accordingly, a compound key pattern $\mathbf{t}_s, \mathbf{t}_p, \mathbf{t}_o$ can be composed of RDF key patterns and/or keyword key patterns. In particular, the index scheme is defined as follows:

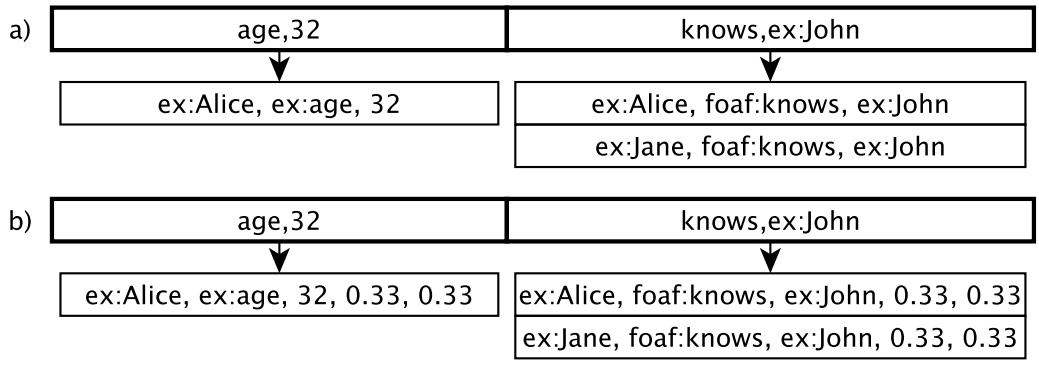


Figure 3: Examples of a) reduced hybrid index and b) rank-aware reduce hybrid index.

Definition 8 (Full Hybrid Index)**.** *A* full hybrid index *is defined by the scheme:*

$$\begin{aligned}\mathbf{p}(\mathbf{t}_s, \mathbf{t}_p, \mathbf{t}_o) &\mapsto \langle \mathbf{s}, \mathbf{p}, \mathbf{o} \rangle \\ \mathbf{p}(\mathbf{t}_p, \mathbf{t}_o, \mathbf{t}_s) &\mapsto \langle \mathbf{s}, \mathbf{p}, \mathbf{o} \rangle \\ \mathbf{p}(\mathbf{t}_o, \mathbf{t}_s, \mathbf{t}_p) &\mapsto \langle \mathbf{s}, \mathbf{p}, \mathbf{o} \rangle\end{aligned}$$

Due to prefix lookup, this index scheme supports exactly seven hybrid triple patterns $\langle s, p, o \rangle$ on the graph where s, p and o are variables, RDF or keyword terms. These correspond to the key pattern where all three elements are RDF/keyword terms, i.e. (1) $\mathbf{t}_s, \mathbf{t}_p, \mathbf{t}_o$, the patterns where two elements are RDF/keyword terms, i.e. (2) $\mathbf{t}_s, \mathbf{t}_p$, (3) $\mathbf{t}_s, \mathbf{t}_o$ and (4) $\mathbf{t}_o, \mathbf{t}_p$, and the patterns where only one element is an RDF/keyword term, i.e. (5) $\mathbf{t}_s$, (6) $\mathbf{t}_p$ and (7) $\mathbf{t}_o$. Note that the order of elements in the compound key patterns does not matter when considering the entire pattern, e.g. $\mathbf{p}(\mathbf{t}_s, \mathbf{t}_p, \mathbf{t}_o)$ and $\mathbf{p}(\mathbf{t}_p, \mathbf{t}_s, \mathbf{t}_o)$ are the same, representing the same type of triple patterns where s, p and o are RDF/keyword terms. However, it does matter when considering its prefixes, e.g. $\mathbf{p}(\mathbf{t}_s, \mathbf{t}_p, \mathbf{t}_o)$ contains $\mathbf{t}_s, \mathbf{t}_p$ as one prefix but not $\mathbf{t}_p, \mathbf{t}_s$, which is only available with $\mathbf{p}(\mathbf{t}_p, \mathbf{t}_s, \mathbf{t}_o)$.

Clearly, this index supports all keys that can be constructed from the combinations of keyword/RDF terms at subject, predicate and object position. Thus, the upper bound of key-value pairs captured by the index is $|\mathbf{t}_s| \times |\mathbf{t}_p| \times |\mathbf{t}_o| \times |\langle s, p, o \rangle \in G|$. However, it might not be necessary to support all possible access patterns using such a full index. We observe that all triple patterns found in real-world structured SPARQL queries (e.g. queries against DBpedia [24] or queries in the USEWOD2012[3] query logs) contain at least one variable. Consequently, we create indexes that support access patterns with up to two RDF/keyword terms, i.e. the six patterns 2-7 mentioned above:

Definition 9 (Reduced Hybrid Index)**.** *A* reduced hybrid index *is defined by the scheme:*

[3] `http://data.semanticweb.org/usewod/2012/challenge.html`

$$\begin{aligned}\mathbf{p}(\mathbf{t}_s, \mathbf{t}_p) &\mapsto \langle \mathbf{s}, \mathbf{p}, \mathbf{o} \rangle \\ \mathbf{p}(\mathbf{t}_p, \mathbf{t}_o) &\mapsto \langle \mathbf{s}, \mathbf{p}, \mathbf{o} \rangle \\ \mathbf{p}(\mathbf{t}_o, \mathbf{t}_s) &\mapsto \langle \mathbf{s}, \mathbf{p}, \mathbf{o} \rangle\end{aligned}$$

To support ranking, we further introduce a rank-aware index. Prefix key patterns actually capture several patterns. There is more than one score for each prefix key pattern, namely one for the entire pattern and one for each prefix. This results in two scores for each prefix key pattern supported by the reduced hybrid index:

Definition 10 (Rank-Aware Reduced Hybrid Index)**.** *Let* $r = \langle \mathbf{s}, \mathbf{p}, \mathbf{o} \rangle$*, a* rank-aware hybrid index *is defined by:*

$$\begin{aligned}\mathbf{p}(\mathbf{t}_s, \mathbf{t}_p) &\mapsto (r, score(t_s, r), score(\{t_s, t_p\}, r)) \\ \mathbf{p}(\mathbf{t}_p, \mathbf{t}_o) &\mapsto (r, score(t_p, r), score(\{t_p, t_o\}, r)) \\ \mathbf{p}(\mathbf{t}_o, \mathbf{t}_s) &\mapsto (r, score(t_o, r), score(\{t_o, t_s\}, r))\end{aligned}$$

Example 5. *Fig. 3a shows two example entries for the* $\mathbf{p}(\mathbf{t}_p, \mathbf{t}_o) \mapsto \langle \mathbf{s}, \mathbf{p}, \mathbf{o} \rangle$ *access patterns of the reduced hybrid index. Note that the index keys contain keyword terms, RDF terms as well as a combination of both (e.g.* knows,ex:John*). The same example is shown in Fig. 3b for a rank-aware reduced hybrid index where every matching triple is associated with two scores, one for the entire pattern with two elements, and one for the one-element prefix, e.g. the score for matching* age *and the score for matching both* age *and* 32*.*

The merits of this solution is that the same triple-based strategy is used to serve both structured and keyword parts of the hybrid queries. Given any hybrid triple pattern, only lookups have to be performed on these triple indexes, as opposed to costly joins as discussed for Semplore and the database extensions. This is especially beneficial when joining results from two or more hybrid triple patterns. Our solution requires only direct joins between triples while these previous works involve more complex joins between entities and triples. Based on the scores stored in the rank-aware index, rank join can be supported in addition to standard join operators. Now, we show that with the reduced hybrid index, all the discussed query types can be supported:

- *Unstructured vs. structured*: the fragment $\mathbf{t}_o^k \mapsto \langle \mathbf{s}, \mathbf{p}, \mathbf{o} \rangle$ captured by the pattern $\mathbf{p}(\mathbf{t}_o, \mathbf{t}_s) \mapsto \langle \mathbf{s}, \mathbf{p}, \mathbf{o} \rangle$ supports keyword queries where $t \in \mathbf{t}_o^k$ is the keyword term and the returned entity result is $s \in \mathbf{s}$. Structured query patterns are supported by the fragments $\mathbf{p}(\mathbf{t}_o^{RDF}, \mathbf{t}_s^{RDF}) \mapsto \langle \mathbf{s}, \mathbf{p}, \mathbf{o} \rangle$, $\mathbf{p}(\mathbf{t}_p^{RDF}, \mathbf{t}_o^{RDF}) \mapsto \langle \mathbf{s}, \mathbf{p}, \mathbf{o} \rangle$ and $\mathbf{s}(\mathbf{t}_p^{RDF}, \mathbf{t}_s^{RDF}) \mapsto \langle \mathbf{s}, \mathbf{p}, \mathbf{o} \rangle$, i.e. those key-value pairs where the keys are RDF terms.
- *Entity vs. attribute vs. relational*: entity queries require lookups for triple patterns where the subject is a variable and the predicate and object might be keyword or RDF terms. This is supported by $\mathbf{p}(\mathbf{t}_p, \mathbf{t}_o) \mapsto \langle \mathbf{s}, \mathbf{p}, \mathbf{o} \rangle$. Since relational queries also require access patterns where the subject is a variable (the difference to entity queries lies in the use of several variables), they do not require any additional support. For attribute queries, there is $\mathbf{p}(\mathbf{t}_s, \mathbf{t}_p) \mapsto \langle \mathbf{s}, \mathbf{p}, \mathbf{o} \rangle$, which enables the

retrieval of attribute values given a subject, a predicate or both as RDF/keyword terms.

- *Schema-based and schema-agnostic*: The access patterns needed to support these types of queries are $\mathbf{p}(\mathbf{t}_s, \mathbf{t}_p) \mapsto \langle \mathbf{s}, \mathbf{p}, \mathbf{o} \rangle$ and $\mathbf{p}(\mathbf{t}_o, \mathbf{t}_p) \mapsto \langle \mathbf{s}, \mathbf{p}, \mathbf{o} \rangle$, where the attribute/relation names need to be specified, and $\mathbf{p}(\mathbf{t}_s, \mathbf{t}_o) \mapsto \langle \mathbf{s}, \mathbf{p}, \mathbf{o} \rangle$, where they occur as variables.

4.3 HybIdx Implementation

HybIdx as an index scheme, is independent from the concrete index implementation. It can be implemented on top of any data structure that supports the mapping of single items (keys) to lists of items (values).

Inverted Indexes for Compound Keys. We use inverted indexes, which originally, map keyword terms to documents. We extend the inverted index implementation provided by Lucene, which comprise sparse indexes over sorted arrays, to map compound keyword/RDF terms to triples. We implement a compound term as a concatenated term in Lucene, i.e. the compound key t_s, t_o, t_p is simply the term concatenation "$t_s//t_o//t_p$". For instance, the compound key age,32 in our example is stored in the index as the term age//32. We firstly construct the sets of all RDF terms, $\mathbf{t}_s^{RDF}$, $\mathbf{t}_p^{RDF}$ and $\mathbf{t}_o^{RDF}$, each comprising the elements s, p and o of all the triples $\langle s, p, o \rangle \in G$, respectively. By extracting the keywords from these elements, we obtain the set of all keyword terms, i.e. each $\mathbf{t}_s^k$, $\mathbf{t}_p^k$ and $\mathbf{t}_o^k$ contains all the elements $k_s \in text(s), k_p \in text(p)$ and $k_o \in text(o)$, respectively, for all $\langle s, p, o \rangle \in G$. Then depending on the indexes, different compound keys have to be constructed, e.g. with the reduced HybIdx index, we have three 2-elements compound key patterns. For instance for $\mathbf{p}(\mathbf{t}_s, \mathbf{t}_p)$, we construct all combinations of RDF terms from elements in $\mathbf{t}_s^{RDF}$ and $\mathbf{t}_p^{RDF}$, and all combinations of keyword terms from $\mathbf{t}_s^k$ and $\mathbf{t}_p^k$. Finally, we store the resulting compound keys as concatenated terms in the index, e.g. to obtain age//32.

Dictionary Encoding of Key Values. Common among inverted index implementations is the use of a dictionary, where terms and documents are assigned identifiers, which are then used for indexing [22]. We use dictionary encoding also for RDF triples, i.e. for the values $\langle s, p, o \rangle$ or $(r, score(t_s, r), score(\{t_s, t_p\}, r))$ of the HybIdx indexes. Encoding triples requires dealing with three RDF terms. Often, triples returned from an index are further joined to compute the final query results. This involves accessing individual terms in the triples. Therefore, we apply dictionary encoding at the level of individual terms in the triples. Similarly to the document case, the dictionary is only accessed to return the final results while all intermediate processing is performed on the more compact encoded values.

Index Updates. Since HybIdx is implemented as an inverted index, i.e. consists of inverted lists representing (concatenated) term to triple mappings, the many existing techniques developed for updates can be directly applied. In particular, we do not update the indexes in real-time but keep the changes in a temporary in-memory index first, making them immediately available for searches, and then periodically write them to disk as incremental updates [22].

5 Related Work

Indexing Schemes. Works in this direction are most related, which we have discussed in detail in Sec. 3.2. Our solution provides a generalization of inverted indexes for keyword-based querying (see overview in [22]) and triple indexes for BGP-based querying [19, 17] to answer different access patterns possible with HGPs.

SPARQL Fulltext Extension. HGPs extend both the syntax (RDF terms + keyword terms) and semantics (keyword terms are evaluated with IR-style relevance) of SPARQL BGPs. SPARQL fulltext extensions are provided by a number of vendors (e.g. Virtuoso, OWLIM). The semantics supported by them also build upon the IR-style relevance employed by the underlying IR engine. Due to the absence of standardization, each vendor uses its own proprietary syntax. Common is the use of a predefined full-text predicate, e.g. contains[4] such that Q_1 would be expressed as $\langle s, p, k\rangle\langle k, \text{contains}, \text{dbpedia experiments}\rangle$. Possibly influenced by this syntax, the full-text support implemented by vendors is as discussed: the k index is used to retrieve RDF terms matching dbpedia experiments as bindings for k, and then joined with triple bindings for $\langle s, p, k\rangle$. We explicitly distinguish RDF terms from keyword terms, thus avoiding the use of such a predefined predicate. This is close in spirit to languages like XQuery Full-Text [5], with the difference that it deals with RDF graphs not with XML trees. A proposal for SPARQL/RDF that is close to our solution is described in [25], where keyword terms can be associated with triple patterns. Our proposal enables more fine-grained full-text constraints in that every element in the triple pattern can be a keyword term. As a result, the various types of queries discussed in Sec. 3.1 can be expressed as HGPs.

Hybrid Search. Besides works on indexing applicable for this discussed in Sec. 3.2, there are other directions studied in the area of DB & IR integration [1]. Chakrabarti et al. [26] discuss how keyword-based search can be extended by adding structure to data and query answers. Recent work on QUICK by Pound et al. [10] deals with document retrieval based on entity queries where keywords can also match structural elements of the data graph. However, relational queries are not supported. There are also works on auto-completion, which are based on indexing ranges of terms instead of single terms. ESTER [27] answers queries over structured ontologies based on the prefix search capability of this auto-completion index. Also, this index is entity-based. Relational queries are possible but require a large number of joins because ESTER employs a strategy similar to the one discussed for Semplore.

Query Processing. For inverted indexes, query processing is mainly concerned with the fast intersection of inverted lists to obtain matching documents [22]. There is also work on the efficient processing of SPARQL queries over RDF data [17, 21]. As discussed in Sec. 3.3, processing HGPs is more similar to processing BGPs on RDF data than to processing keyword queries on inverted indexes. Optimization techniques, such as join ordering and sideways information passing are therefore applicable. We also employ top-k processing techniques [23] (rank join) to report the top-k results without having

[4] http://www.w3.org/2009/sparql/wiki/Feature:FullText

to process all input data.

6 Evaluation

We perform the evaluation on a total of six indexing schemes, representing HybIdx and existing approaches discussed in Sec. 3.2.

6.1 Systems

We use **OWLIM-SE**[5] as a representative for database extensions of type (2), i.e. it uses a separate Lucene index for indexing RDF terms. **ID** is the native index solution for indexing dataspaces. **VI** is the vertical index scheme where one index is created for every attribute. VI only indexes a maximum of 300 attributes because of the overhead associated with managing and accessing a large number of separate indexes [13]. **SE** is the Semplore system. **HySort** is our system using the reduced hybrid index where values are sorted on the term identifiers and the top-k results are extracted after all results have been calculated. **HyTopK** uses the same reduced hybrid index but with top-k processing. For this, values are sorted on the term scores.

All systems take the parameter k to determine how many top results should be returned. However, only HyTopK employs top-k processing to terminate after computing these results whereas the other systems compute all results and then perform sorting to obtain the top-k ones.

With the exception of OWLIM, all systems are based on the same Lucene index implementation and optimizations discussed in Sec. 4.3. For combining results, hash join, and when possible, merge and index-based joins are executued for all other systems, while HyTopK uses rank join. Query plans are left-deep and created by a heuristic optimizer.

The evaluation was performed on a server with two 2.3 GHz CPUs and 12GB RAM, of which 8 GB were assigned to the JVM. During the evaluation, we cleared all caches after each query evaluation, including the operating system disk caches. All queries were run a total of ten times and the reported times are the average of the last five runs to account for the warm-up of the Java JIT compiler.

6.2 Datasets and Queries

The evaluation was performed on datasets of varying sizes. Some include a large number of documents (WP, AQY, WDB), whereas others contain a large amount of structured data (IMDB, YAGO, WDB). Queries for each dataset are based on the ones used in previous works. Keyword queries are translated manually to hybrid queries, where all keywords are incorporated as keyword terms. Fig. 4 shows a sample of queries that we will discuss later.

WP. This export of Wikipedia used by a recently published keyword search benchmark [15] contains about 5k revisions of pages and information about the revision authors. Keyword queries available for this dataset [15] that were translated to hybrid

[5] `http://owlim.ontotext.com/display/OWLIMv50/OWLIM-SE`

WDB_4 $\langle \mathsf{x}, type, settlement\rangle\langle \mathsf{x}, label, sydney\rangle$
$\langle \mathsf{x}, city, \mathsf{y}\rangle\langle \mathsf{y}, type, airport\rangle$
WDB_6 $\langle \mathsf{x}, type, animal\rangle\langle \mathsf{x}, label, \mathsf{y}\rangle$
AQY_{238} $\langle \mathsf{x}, label, damon\rangle\langle \mathsf{x}, haswonprize, \mathsf{y}\rangle$
$\langle \mathsf{y}, label, 2004\rangle\langle \mathsf{d}, mentions, \mathsf{y}\rangle$

Figure 4: Selected evaluation queries.

queries include 35 schema-agnostic entity queries and one relational query. Queries that require OR semantics were not included.

IMDB. This one contains information about movies and actors. For this, there are 20 schema-agnostic entity queries and 26 relational queries defined in the benchmark [15]. Four keyword queries from this benchmark were left out because they could not be translated to our query model.

YAGO. This is one part of the knowledge base used in [9]. It contains cross-domain knowledge extracted from Wikipedia, such as people, organizations, locations, etc. and relationships between them. From the hybrid queries used in [9], we took 67 queries (4 entity and 63 relational queries) that are compatible with our query model.

AQY. This consists of YAGO and the AQUAINT-2 news document collection annotated with entities from YAGO, as used in [10]. There are about 900k documents and 17M YAGO entity annotations. All queries retrieve documents based on the annotated entities. They all are relational queries as entities are connected to documents via a special mentions predicate.

WDB. We created this dataset by enriching entities in DBpedia with their corresponding Wikipedia page. In total, it includes about 73M triples and 6.5M documents. We add keyword terms to structured queries in the DBpedia SPARQL benchmark [24] to create hybrid queries (9 schema-agnostic entity queries and 10 relational queries). In addition, we use random sampling to create the **WDB-P** query set comprising 500 queries that consist of one single pattern (each with up to 5 keyword terms).

Tab. 1a shows the average number of keyword terms per RDF term at the subject, predicate, and object position. We see the datasets that include documents (WP, AQY and WDB) have a higher number of keyword terms at the object position. WP has a particularly high number of keyword terms as it does not contain as much structured data as AQY and WDB do. Also, subjects in YAGO, AQY, and WDB contain more keyword terms than the subjects in WP and IMDB, which is due to the fact that both WP and IMDB are exports of relational datasets whose entity URIs do not contain words but only numerical identifiers. YAGO, AQY and WDB on the other hand have URIs encoding the names of the resources that can be used in query patterns.

Tab. 1c shows how many keywords occur on average at the subject, predicate and object position as well as in the whole query (constants are not counted). For instance, we can see that the WP query set mainly contains keyword-based entity queries where keywords occur almost exclusively at the object position. In all query sets, keywords are also used at the predicate position to query the schema (for constructing schema-agnostic queries). YAGO queries also retrieve attribute values for some entities, given as keywords.

Tab. 1b presents the size of all datasets and indexes for each dataset (in GB). Hy*

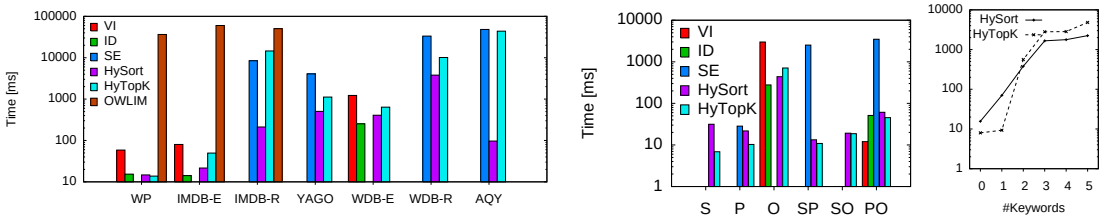


Figure 5: Average query times for a) all query sets (excluding WDB-P), b) single pattern queries WDB-P (at $k = 10$) for different pattern types, and c) O patterns in WDB-P for different numbers of keywords.

requires more storage space than other systems, but no more than 5 times the size of the datasets. This increased space requirement is expected, given Hy* targets many more access patterns and query types that are not possible with the other indexes. The experimental results discussed in the following suggest that this index scheme not only increases the expressiveness of supported queries but also, yields superior time performance.

6.3 Results

For each query set, Tab. 1d shows the number of queries that can be executed using each system. For example, VI is able to execute only 20 of the 46 IMDB queries, which is due to its focus on schema-based entity queries. SE cannot evaluate any WP query because it requires a constant at the predicate position. VI and ID are able to execute all but the one relational query in WP. Whereas the other systems can only execute certain subsets, Hy* and OWLIM can execute all queries.

Overview. Fig. 5a gives an overview of the average query processing times for all systems and queries (excluding WDB-P). For each query set we include only systems that are able to execute at least some of the queries in the query set. Then, we compute the average only over queries that were executable on all such systems in order to make the results comparable. For example, SE cannot execute any query of the WP dataset and is therefore not included in the results. The results for WP only include the average of 35 queries that were executable on VI, ID, OWLIM and Hy* (from a total of 36 queries). We split the IMDB and WDB query sets into the two sets of entity and relational queries (IMDB-E,IMDB-R and WDB-E, WDB-R).

For OWLIM, we only include results for WP and IMDB, as queries for the other datasets timed out (after 60 seconds). As discussed, this is because even for a single hybrid triple pattern, OWLIM needs to join RDF terms obtained for the keyword with triples matching the pattern. Further, OWLIM cannot take advantage of the indexes to perform joins, while HySort can perform index-based joins, where results from one side of the join are used to perform lookups on the other side. This considerably improves performance for queries with selective triple patterns.

We now begin with the analysis of the generated single pattern queries to obtain a basic understanding of the capabilities of different systems. Then, we turn attention to the main queries presented in Fig. 6, based on their decomposition into entity, document and relational queries.

	WP	IMDB	YAGO	AQY	WDB
a) Keyword terms per RDF term					
Subj.	3.00	3.09	4.48	4.19	4.89
Pred.	3.02	3.68	3.02	3.06	2.38
Obj.	34.30	3.17	3.57	11.50	12.38
b) Dataset and index sizes (in GB)					
Dataset	0.22	0.42	1.78	6.6	19.36
VI	0.05	0.11	0.20	1.56	5.37
ID	0.13	0.26	0.77	7.01	16.43
SE	0.05	0.28	0.75	4.85	7.50
Hy*	0.44	1.12	5.01	28.11	86.71
OWLIM	0.27	0.68	1.55	13.03	34.18
c) Query keywords statistics					
Keywords	2.14	5.15	2.70	4.60	3.42
Subj-Kw.	0.00	0.00	0.76	0.00	0.00
Pred-Kw.	0.06	2.43	1.57	2.53	1.21
Obj-Kw.	2.08	2.72	0.37	2.07	2.21
d) Query compatibility					
#Queries	36	46	67	15	19
VI	35	20	0	0	9
ID	35	20	4	0	11
SE	0	26	67	13	7
Hy*	36	46	67	15	19
OWLIM	36	46	67	15	19

Table 1: Dataset and query statistics.

Single Pattern Queries. Fig. 5b shows the average evaluation time for WDB-P queries, each consisting of a single pattern. We group patterns by the positions where constants appear, i.e. PO refers to patterns that have a variable as subject and constants at the predicate and object position. First, we see that ID and VI only answer P and PO patterns. SE only supports S, SP and SO patterns, whereas Hy* support all patterns.

The predicate is specified in PO patterns. Here, VI can perform a single lookup in the index built for a particular predicate, thereby outperform ID and Hy* that have larger indexes. With O patterns, VI has to perform lookups in all its predicate indexes, leading to much worse performance. ID outperforms Hy* for both, O and PO patterns as ID only stores entities instead of triples. SE requires joins to answer SP and PO, leading to worse performance compared to the Hy* approaches. For P patterns, SE does not require joins and therefore has performance comparable to HySort.

For most patterns, HyTopK outperforms HySort as it does not have to process all inputs to obtain the top-k results. The exception are O patterns, where HyTopK has an average query time of 711.3 ms compared to 439.4 ms for HySort. Fig. 5c shows the query times for O patterns on HySort and HyTopK for different numbers of keywords in the patterns. We see that for zero or one keyword, HyTopK performs better, whereas HySort performs better for two or more keywords. When a pattern contains more than

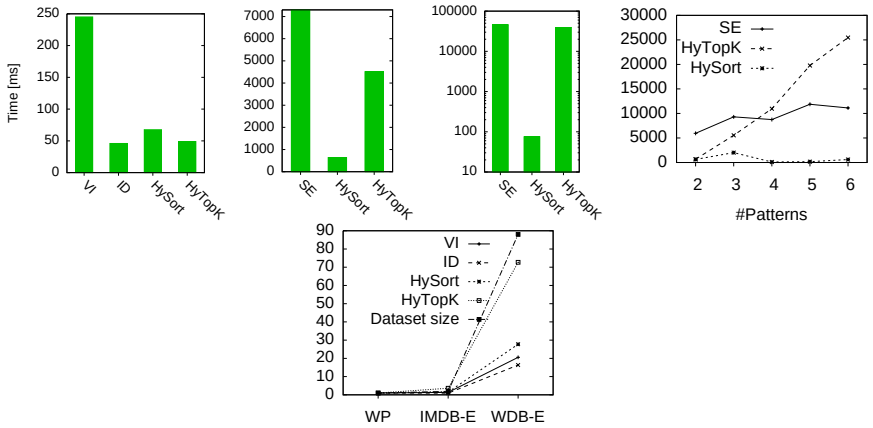


Figure 6: Average query times for a) entity queries, b) relational queries, c) document queries ($k = 1$), d) relational queries in IMDB and YAGO for different numbers of query patterns, and e) entity queries in WP, IMDB and WDB, relative to query times for WP. WDB-P is excluded in all results.

one keyword, the query engine performs an intersection between the matches for each keyword (AND semantics). In this case, HySort uses efficient merge join as all triples are sorted on one of the variables in the pattern. Instead, HyTopK performs a rank join, which compared to merge join, requires overhead in maintaining hash tables.

Entity Queries. Fig. 6a shows average query times for all entity queries that were executable on VI, ID and Hy*, i.e. entity queries where the predicate is a variable or an RDF term. This includes queries from WP, IMDB-E and WDB-E. We can see here that VI is outperformed by all other systems because it has to access all its indexes when the predicates are not given. The average time of VI is 245.1 ms, compared to 45.9 ms for ID, 67.6 ms for HySort and 48.9 ms for HyTopK. While both HySort and HyTopK are faster than VI, HySort is still outperformed by ID because the latter only stores entities instead of triples. Through top-k processing, HyTopK's results are close to the ones achieved by ID.

Relational Queries. Fig. 6b shows the average times for all relational queries that were executable with SE, HySort and HyTopK. This includes queries from IMDB-R and WDB-R. With an average query time of 638.4 ms, HySort outperforms both HyTopK (4511.8 ms) and SE (7310.66 ms). While HySort has to produce all results in order to identify the top-k ones, it still outperforms HyTopK. It seems that the ability to take advantage of available indexes to perform index-based joins is more important w.r.t. relational queries that involve more complex joins, compared to entities queries (where HyTopK is better than HySort). SE has a larger overhead than both HySort and HyTopK as additional merge joins are necessary for patterns that have keywords. Hence, it exhibits the worse performance.

WDB_4 is an example of a relational query (see Fig. 4) where HySort (734.0 ms) outperforms HyTopK (2054.0 ms). The initial join between the first two pattern produces only few results, which HySort uses to perform an index-based join with the third pattern, for which only a few lookups are needed. HyTopK, on the other hand, accesses the matching triples for each pattern in the order of their scores. Depending on the score

distribution it can happen that a large part of the input needs to be read until joining triples are found.

WDB_6 (Fig. 4) is a query with low selectivity where HyTopK outperforms HySort (7873.3 ms vs. 22669.6 ms). Here, both patterns match many triples. Thus, index-based joins do not perform better as they require many lookups.

In other words, whether top-k processing should be used or not depends on the complexity of the queries (the complexity of joins). Our index solution supports both these paradigms; one way to exploit this index is to apply optimization techniques [23] that not only find the optimal join orders but also which type of joins to be performed.

Document Queries. Fig. 6c shows the average query times for SE, HySort and HyTopk for the 13 queries of the AQY query set. With an average time of 74.9 ms at $k = 1$ compared to 45977.7 ms and 39163.5 ms, HySort outperforms SE and HyTopk by several orders of magnitude. All queries contain the pattern $\langle \mathsf{x}, \mathit{mentions}, \mathsf{y} \rangle$. Using the $\mathbf{t}_p \rightarrow \langle \mathbf{s}, \mathbf{o} \rangle$ index, all annotation triples have to be retrieved for every execution. For HyTopk, the drawback is that all annotations are stored sorted on their score. Hence, a lot of them might have to be loaded to find one matching the join condition. HySort on the other hand takes advantage of the $\mathbf{t}_o, \mathbf{t}_s \rightarrow \langle \mathbf{s}, \mathbf{p}, \mathbf{o} \rangle$ index to retrieve only those triples matching the join condition.

For query AQY_{238} (Fig. 4), which finds all documents that mention an entity labeled *damon* that won a prize in 2004. HySort can first perform joins to find entities that match these conditions and then use an index-based join to find all documents mentioning that entity. HyTopK performs worse (25427.7 ms vs. 86.4 ms) because it may process all *mentions* triples in the worst case and cannot take advantage of the available indexes as HySort does.

Query Size. For different query sizes (measured in terms of the number of patterns), Fig. 6d presents average time for relational queries on IMDB-R and YAGO. We see that for HyTopK and SE, processing times increase with the number of patterns, by factors of 29.7 and 1.9, respectively, from 2 to 6 patterns. Processing times for HySort remain constant or even decrease. The reason for this is that due to index-based joins, HySort's performance is not solely dependent on query size. In the case of highly selective queries, HySort only requires a few index lookups to compute the results whereas SE and HyTopK have to process all the data for each query pattern (in the worst case).

Scalability. Fig. 6e shows processing times for entity queries on WP, IMDB-E, and WDB-E relative to the processing times obtained for WP. Additionally, the graph also shows the relative size of the datasets w.r.t. the WP dataset. The IMDB (WDB) dataset is 2 times (88 times) larger than the WP dataset. We can we see that the average query times for VI and ID increase less (e.g. by factors of 20.6 and 16.3, respectively, for WDB) than those for HySort and HyTopK (27.8 and 72.7). This is due to differences in index size as well as the return values employed by the indexing schemes. Whereas both VI and ID have single entities as values, Hy* stores triples. Processing and decoding triples require higher costs because they contain 3 different terms, thus leading to worse performance for large datasets. Triples however, are needed for relational queries. The results however indicate that the time increase incurred by HyTopK and especially by HySort, is less than the increase in dataset size. This is because the increase in index

size hence the performance, is largely determined by the number of keyword terms, not the data size.

7 Conclusion

Building upon SPARQL and RDF, we propose a hybrid search approach that supports different types of hybrid queries over text-rich data graphs. We provide an indexing solution, HybIdx, which supports the different access patterns needed to support these queries. We collect data and queries from existing benchmarks and experimental studies to perform a systematic comparison of indexing schemes for hybrid search. The conclusions of this study are: (1) existing queries range from attribute to entity to complex schema-agnostic relational queries. (2) Database extensions capable of dealing with this variety of queries are not time efficient, requiring complex joins. (3) Existing native indexes are efficient but focus on specific type of queries, i.e. entity queries. (4) HybIdx is the only solution that is both efficient and versatile in terms of query type support. For relational and document queries, it outperforms the second best approach by one and three orders of magnitude, respectively. For entity queries, the native solution is slightly (6%) faster but is entirely optimized towards these queries such that other types are not possible.

As future work, we study query optimization techniques supporting query plans that may contain both standard join operators and rank join to support an optimal combination of top-k and non-top-k processing. Further, the study suggests that more optimized indexes can be created, given the query types to be supported are known. We will work on techniques that can automatically adapt the employed index scheme to the given workload, to identify and optimize the indexes for the access patterns that are actually needed.

References

[1] G. Weikum, "Db&ir: both sides now," in *SIGMOD Conference*, 2007, pp. 25–30.

[2] V. Hristidis and Y. Papakonstantinou, "Discover: Keyword search in relational databases," in *VLDB*, 2002, pp. 670–681.

[3] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou, "Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data," in *SIGMOD Conference*, 2008, pp. 903–914.

[4] T. Tran, H. Wang, S. Rudolph, and P. Cimiano, "Top-k exploration of query candidates for efficient keyword search on graph-shaped (rdf) data," in *ICDE*, 2009, pp. 405–416.

[5] S. Amer-Yahia and M. Lalmas, "Xml search: languages, inex and scoring," *SIGMOD Record*, vol. 35, no. 4, pp. 16–23, 2006.

[6] S. Amer-Yahia, L. V. S. Lakshmanan, and S. Pandit, "FleXPath: flexible structure and full-text querying for XML," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '04. New York, NY, USA: ACM, 2004, p. 83–94.

[7] M. Theobald, R. Schenkel, and G. Weikum, "An efficient and versatile query engine for topx search," in *VLDB*, 2005, pp. 625–636.

[8] X. Dong and A. Halevy, "Indexing dataspaces," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '07. New York, NY, USA: ACM, 2007, p. 43–54.

[9] G. Kasneci, F. M. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum, "NAGA: searching and ranking knowledge," in *IEEE 24th International Conference on Data Engineering, 2008. ICDE 2008.* IEEE, Apr. 2008, pp. 953–962.

[10] J. Pound, I. F. Ilyas, and G. Weddell, "Expressive and flexible access to web-extracted data: a keyword-based structured query language," in *SIGMOD 2010*.

[11] H. Wang, T. Tran, C. Liu, and L. Fu, "Lightweight integration of ir and db for scalable hybrid search with integrated ranking support," *J. Web Sem.*, vol. 9, no. 4, pp. 490–503, 2011.

[12] H. Wang, Q. Liu, T. Penin, L. Fu, L. Zhang, T. Tran, Y. Yu, and Y. Pan, "Semplore: A scalable ir approach to search the web of data," *J. Web Sem.*, vol. 7, no. 3, pp. 177–188, 2009.

[13] R. Blanco, P. Mika, and S. Vigna, "Effective and efficient entity search in rdf data," in *International Semantic Web Conference (1)*, 2011, pp. 83–97.

[14] J. M. Ponte and W. B. Croft, "A language modeling approach to information retrieval," in *SIGIR*, 1998, pp. 275–281.

[15] J. Coffman and A. C. Weaver, "A framework for evaluating database keyword search strategies," in *CIKM*, 2010, pp. 729–738.

[16] H. Garcia-Molina, J. Ullman, and J. Widom, *Database system implementation.* Prentice Hall Upper Saddle River, NJ:, 2000, vol. 654.

[17] T. Neumann and G. Weikum, "Rdf-3x: a risc-style engine for rdf," *PVLDB*, vol. 1, no. 1, pp. 647–659, 2008.

[18] G. Cheng, W. Ge, and Y. Qu, "Falcons: searching and browsing entities on the semantic web," in *WWW*, 2008, pp. 1101–1102.

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

[20] B. Bishop, A. Kiryakov, D. Ognyanoff, I. Peikov, Z. Tashev, and R. Velkov, "Owlim: A family of scalable semantic repositories," *Semantic Web*, vol. 2, no. 1, pp. 33–42, 2011.

[21] T. Neumann and G. Weikum, "Scalable join processing on very large rdf graphs," in *SIGMOD Conference*, 2009, pp. 627–640.

[22] J. Zobel and A. Moffat, "Inverted files for text search engines," *ACM Comput. Surv.*, vol. 38, no. 2, Jul. 2006.

[23] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A survey of top-k query processing techniques in relational database systems," *ACM Comput. Surv.*, vol. 40, no. 4, 2008.

[24] M. Morsey, J. Lehmann, S. Auer, and A.-C. N. Ngomo, "Dbpedia sparql benchmark - performance assessment with real queries on real data," in *International Semantic Web Conference (1)*, 2011, pp. 454–469.

[25] S. Elbassuoni, M. Ramanath, R. Schenkel, and G. Weikum, "Searching rdf graphs with sparql and keywords," *IEEE Data Eng. Bull.*, vol. 33, no. 1, pp. 16–24, 2010.

[26] S. Chakrabarti, S. Sarawagi, and S. Sudarshan, "Enhancing search with structure," *IEEE Data Engineering Bulletin*, 2010.

[27] H. Bast, A. Chitea, F. M. Suchanek, and I. Weber, "Ester: efficient search on text, entities, and relations," in *SIGIR*, 2007, pp. 671–678.