# Social, Structured and Semantic Search

Raphaël Bonaque, Bogdan Cautis, François Goasdoué, Ioana Manolescu

# Social, Structured and Semantic Search

Raphaël Bonaque
INRIA & LIX (CNRS UMR 7161 and Ecole Polytechnique)
raphael.bonaque@inria.fr

Bogdan Cautis
U. Paris-Sud & INRIA, France
bogdan.cautis@lri.fr

François Goasdoué
U. Rennes 1 & INRIA, France
fg@irisa.fr

Ioana Manolescu
INRIA & LIX (CNRS UMR 7161 and Ecole Polytechnique)
ioana.manolescu@inria.fr

## ABSTRACT

Social content such as blogs, tweets, news etc. is a rich source of interconnected information. We identify a set of requirements for the meaningful exploitation of such rich content, and present a new data model, called `S3`, which is the first to satisfy them. `S3` captures *social* relationships between users, and between users and content, but also the *structure* present in rich social content, as well as its *semantics*. We provide the first top-$k$ keyword search algorithm taking into account the social, structured, and semantic dimensions and formally establish its termination and correctness. Experiments on real social networks demonstrate the efficiency and qualitative advantage of our algorithm through the joint exploitation of the social, structured, and semantic dimensions of `S3`.

## 1. INTRODUCTION

The World Wide Web (or Web, in short) was designed for users to interact with each other by means of pages interconnected with hyperlinks. Thus, the Web is the earliest inception of an *online* social network (whereas "real-life" social networks have a much longer history in social sciences). However, the technologies and tools enabling large-scale online social exchange have only become available recently. A popular model of such exchanges features: *social network users*, who may be connected to one another, *data items*, and the possibility for users to *tag* data items, i.e., to attach to an item an annotation expressing the user's view or classification of the item. Variants of this "user-item-tag" (*UIT*) model can be found e.g., in [18, 21, 30]. In such contexts, a user, called *seeker*, may ask a query, typically as a set of keywords. The problem then is to find the best query answers, taking into account both the relevance of items to the query, and the social proximity between the seeker and the items, based also on tags. Today's major social networks e.g., Facebook [7], all implement some UIT variant. We identify a set of basic requirements which UIT meets:

**R0.** UIT models *explicit social connections* between users, e.g., $u_1$ is a friend of $u_0$ in Figure 1, to which we refer throughout this paper unless stated otherwise. It also captures *user endorsement (tags)* of data items, as UIT search algorithms *exploit both the user endorsement and the social connections* to return items most likely to interest the seeker, given his social and tagging behavior.

To fully exploit the content shared in social settings, we argue that the model used for such data (and, accordingly, the query model) must also satisfy the requirements below:

**R1.** The current wealth of publishing modes (through social networks, blogs, interlinked Web pages etc.) allows many different relations between items. For example, document $d_1$ *replies to* document $d_0$ (think for instance of opposite-viewpoint articles in a heated debate), while document $d_2$ *comments on* the paragraph of $d_0$ identified by the URI $d_{0.3.2}$. The model must capture **relations between items**, in particular since **they may lead to implicit relations between users**, according to their manipulations of items. For instance, the fact that $u_2$ posted $d_1$ as a reply to $d_0$, posted by $u_0$, entails that $u_2$ at least read $d_0$, and thus some form of exchange has taken place between $u_0$ and $u_2$; if one looked for *explicit* social connections only, we would wrongly believe that $u_0$ and $u_2$ have no relation to each other.

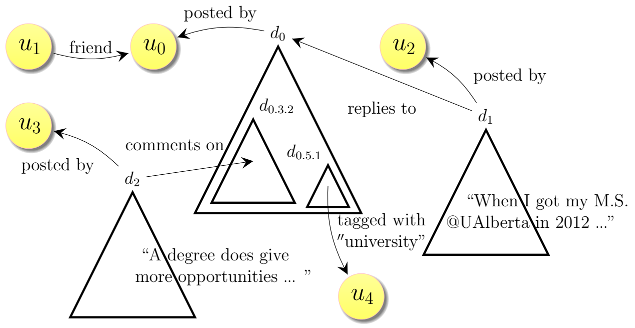

**Figure 1: Motivating example.**

**R2.** Items shared in social media often have a rich structured content. For instance, the article $d_0$ comprises many sections, and paragraphs, such as the one identified by the URI $d_{0.3.2}$. **Document structure must be reflected in the model** in order to return *useful* document fragments as query results, instead of a very large document or a very small snippet of a few words (e.g., exactly the search keywords). Document structure also helps discern when users have *really* interacted through content. For instance, $u_3$ has interacted with $u_0$, since $u_3$ comments on the fragment $d_{0.3.2}$ of $u_0$'s article $d_0$. In contrast, when user $u_4$ tags with "university" the fragment $d_{0.5.1}$ of $d_0$, disjoint from $d_{0.3.2}$, $u_4$ may not even have read the same text as $u_3$, thus the two likely did not interact.

**R3. Item and tag semantics** must be modelled. Social Web data encapsulates users' knowledge on a multitude of topics; ontologies, either general such as DBPedia or Google's Knowledge Base, or application-specific, can be leveraged to *give query answers which cannot be found without relying on semantics*. For instance, assume $u_1$ looks for information about *university graduates*: document $d_1$ states that $u_2$ holds a M.S. degree. Assume a knowledge base specifies that *a M.S. is a degree* and that *someone having a degree is a graduate*. The ability to return as result the snippet of $d_1$ most relevant to the query is directly conditioned by

This research was partially supported by Labex DigiCosme (project ANR-11-LABEX-0045-DIGICOSME) operated by ANR as part of the program "Investissement d'Avenir" Idex Paris-Saclay (ANR-11-IDEX-0003-02).

| | | | |
|---|---|---|---|
| $U$ URIs | $L$ literals | $\mathcal{K}$ keywords | $Ext(k)$ extension of $k$ |
| $\Omega$ users | $D$ documents | $T$ tags | I graph instance |

**Table 1: Main data model notations.**

the ability to exploit the ontology (and the content-based interconnections along the path: $u_1$ friend of $u_0$, $u_0$ posted $d_0$, $d_1$ replied to $d_0$).

**R4.** In many contexts, tagging may apply to tags themselves, e.g., in annotated corpora, where an annotation (tag) obtained from an analysis can further be annotated with provenance details (when and how the annotation was made) or analyzed in its turn. Information from higher-level annotations is obviously still related to the original document. The model should allow expressing **higher-level tags**, to exploit their information for query answering.

**R5.** The data model and queries should have **well-defined semantics**, to precisely characterize computed results, ensure correctness of the implementation, and allow for optimization.

**R6.** The model should be **generic** (not tied to a particular social network model), **extensible** (it should allow easy extension or customization, as social networks and applications have diverse and rapidly evolving needs), and **interoperable**, i.e., it should be possible to get richer / more complete answers by integrating different sources of social connections, facts, semantics, or documents. This ensures in particular independence from any proprietary social network viewpoint, usefulness in a variety of settings, and a desirable form of "monotonicity": the more content is added to the network, the more its information value increases.

This work makes the following contributions.

**1**. We present S3, a novel *data model* for structured, semantic-rich content exchanged in social applications; it is the first model to meet the requirements **R0** to **R6** above.

**2**. We revisit *top-k social search for keyword queries*, to retrieve the most relevant *document fragments* w.r.t. the social, structural, and semantical aspects captured by S3. We identify a set of *desirable properties of the score function* used to rank results, provide a *novel query evaluation algorithm* called $S3_k$ and *formally establish its termination and correctness*; the algorithm intelligently exploits the score properties to stop *as early as possible*, to return answers fast, with little evaluation effort. $S3_k$ is the first to formally guarantee a specific result in a structured, social, and semantic setting.

**3**. We implemented $S3_k$ based on a concrete score function (extending traditional ones from XML keyword search) and experimented with *three real social datasets*. We demonstrate the *feasibility* of our algorithm, and its *qualitative advantage* over existing approaches: it finds relevant results that would be missed by ignoring any dimension of the graph.

An S3 instance can be exploited in many other ways: through structured XML and/or RDF queries as in [9], searching for users, or focusing on annotations as in [4]; one could also apply graph mining etc. In this paper, we first describe the data model, and then revisit the top-k document search problem, since it is the most widely used (and studied) in social settings.

In the sequel, Section 2 presents the S3 data model, while Section 3 introduces a notion of generic score and instantiates it through a concrete score. Section 4 describes $S3_k$, we present experiments in Section 5, then discuss related works in Section 6 and conclude.

## 2. DATA MODEL

We now describe our model integrating social, structured, and semantic-rich content into a *single weighted RDF graph*, and based on a small set of S3*-specific RDF classes and properties*. We present weighted RDF graphs in Section 2.1, and show how they model social networks in Section 2.2. We add to our model structured documents in Section 2.3, and tags and user-document interactions in Section 2.4; Section 2.5 introduces our notion of social paths. Table 1 recaps the main notations of our data model.

| Constructor | Triple | Relational notation |
|---|---|---|
| Class assertion | s type o | o(s) |
| Property assertion | s p o | p(s, o) |

| Constructor | Triple | Relational notation |
|---|---|---|
| Subclass constraint | s $\prec_{\rm sc}$ o | s $\subseteq$ o |
| Subproperty constraint | s $\prec_{\rm sp}$ o | s $\subseteq$ o |
| Domain typing constraint | s $\hookleftarrow_{\rm d}$ o | $\Pi_{\rm domain}$(s) $\subseteq$ o |
| Range typing constraint | s $\hookrightarrow_{\rm r}$ o | $\Pi_{\rm range}$(s) $\subseteq$ o |

**Figure 2: RDF (top) and RDFS (bottom) statements.**

**URIs and literals** We assume given a set $U$ of Uniform Resource Identifiers (URIs, in short), as defined by the standard [28], and a set of literals (constants) denoted $L$, disjoint from $U$.

**Keywords** We denote by $\mathcal{K}$ the set of all possible *keywords*: it contains all the URIs, plus the stemmed version of all literals. For instance, stemming replaces "graduation" with "graduate".

### 2.1 RDF

An *RDF graph* (or *graph*, in short) is a set of *triples* of the form s p o, stating that the *subject* s has the *property* p and the value of that property is the *object* o. In relational notation (Figure 2), s p o corresponds to the tuple (s, o) in the binary relation p, e.g., $\rm u_1$ hasFriend $\rm u_0$ corresponds to hasFriend($u_1$,$u_0$). We consider every triple is *well-formed* [27]: its subject belongs to $U$, its property belongs to $U$, and its object belongs to $\mathcal{K}$.

**Notations** We use s, p, o to denote a subject, property, and respectively, object in a triple. Strings between quotes as in *"string"* denote literals.

**RDF types and schema** The property type built in the RDF standard is used to specify to which *classes* a resource belongs. This can be seen as a form of resource typing.

A valuable feature of RDF is RDF Schema (RDFS), which allows enhancing the resource descriptions provided by RDF graphs. An RDF Schema declares *semantic constraints* between the classes and the properties used in these graphs, through the use of four RDF built-in properties. These constraints can model:

- subclass relationships, which we denote by $\prec_{sc}$; for instance, any *M.S.Degree* is also a *Degree*;
- subproperty relationships, denoted $\prec_{sp}$; for instance, *workingWith* someone also means being *acquaintedWith* him;
- typing of the first attribute (or domain) of a property, denoted $\hookleftarrow_d$, e.g., the domain of *hasDegreeFrom* is a *Graduate*;
- typing of the second attribute (or range) of a property, denoted $\hookrightarrow_r$, e.g., the range of *hasDegreeFrom* is an *University*.

Figure 2 shows the constraints we use, and how to express them. In this figure, domain and range denote respectively the first and second attributes of a property. The figure also shows the relational notation for these constraints, which in RDF are interpreted under the open-world assumption [1], i.e., as *deductive constraints*. For instance, if a graph includes the triples hasFriend $\hookleftarrow_{\rm d}$ Person and $\rm u_1$ hasFriend $\rm u_0$, then the triple $\rm u_1$ type Person holds in this graph even if it is not explicitly present. This *implicit* triple is due to the $\hookleftarrow_d$ constraint in Figure 2.

**Saturation** *RDF entailment* is the RDF reasoning mechanism that allows making explicit all the implicit triples that hold in an RDF graph G. It amounts to repeatedly applying a set of normative immediate *entailment* rules (denoted $\vdash^i_{\rm RDF}$) on G: given some triples explicitly present in G, a rule adds some triples that directly follow from them. For instance, continuing the previous example,

$$\mathrm{u_1\ hasFriend\ u_0,\ hasFriend\ \hookrightarrow_r\ Person\ \vdash^i_{RDF}\ u_0\ type\ Person}$$

Applying immediate entailment $\vdash^i_{\mathrm{RDF}}$ repeatedly until no new triple can be derived is known to lead to a unique, finite fixpoint graph, known as the *saturation* (a.k.a. closure) of G. RDF entailment is part of the RDF standard itself: the answers to a query on G must take into account all triples in its saturation, since *the semantics of an RDF graph is its saturation* [27].

*In the following, we assume, without loss of generality, that all RDF graphs are saturated*; many saturation algorithms are known, including incremental [10] or massively parallel ones [26].

**Weighted RDF graph** Relationships between documents, document fragments, comments, users, keywords etc. naturally form a graph. We encode each edge from this graph by a *weighted RDF triple* of the form $(s, p, o, w)$, where $(s, p, o)$ is a regular RDF triple, and $w \in [0, 1]$ is termed the *weight* of the triple. Any triple whose weight is not specified is assumed to be of weight 1.

We define the saturation of a weighted RDF graph as the saturation derived *only from its triples whose weight is* 1. Any entailment rule of the form $a,\ b \vdash^i_{\mathrm{RDF}} c$ applies only if the weight of $a$ and $b$ is 1; in this case, the entailed triple $c$ also has the weight 1. We restrict inference in this fashion to distinguish triples which certainly hold (such as: "a M.S. is a degree", "$u_1$ is a friend of $u_0$") from others whose weight is computed, and carries a more quantitative meaning, such as "the similarity between $d_0$ and $d_1$ is 0.5"[1].

**Graph instance** I **and** S3 **namespace** We use I to designate the weighted RDF instance we work with. The RDF Schema statements in I allow a semantic interpretation of keywords, as follows:

DEFINITION 2.1 (KEYWORD EXTENSION). *Given an* S3 *instance* I *and a keyword $k \in \mathcal{K}$, the extension of $k$, denoted $Ext(k)$, is defined as follows:*

- $k \in Ext(k)$
- *for any triple of the form* b type k, b $\prec_{\mathrm{sc}}$ k *or* b $\prec_{\mathrm{sp}}$ k *in* I, *we have* b $\in Ext$(k).

For example, given the keyword *degree*, and assuming that M.S. $\prec_{\mathrm{sc}}$ degree holds in I, we have M.S. $\in Ext$(degree). *The extension of k does not generalize it*, in particular it does not introduce any loss of precision: whenever $k'$ is in the extension of $k$, the RDF schema in I ensures that $k'$ is an *instance*, or a *specialization* (particular case) of $k$. This is in coherence with the principles behind the RDF schema language[2].

For our modeling purposes, we define below a small set of RDF classes and properties used in I; these are shown prefixed with the S3 namespace. The next sections show how I is populated with triples derived from the users, documents and their interactions.

## 2.2 Social network

We consider a set of social network users $\Omega \subset U$, i.e., each user is identified by a URI. We introduce the special RDF class S3:user, and for each user $u \in \Omega$, we add: u type S3:user $\in$ I.

To model the relationships between users, such as "friend", "coworker" etc., we introduce the special property S3:social, and model any concrete relationship between two users by a triple whose property specializes S3:social. Alternatively, one may see S3:social as the *generalization of all social network relationships*.

Weights are used to encode the strength $w$ of each relationship going from a user $\mathrm{u_1}$ to a user $\mathrm{u_2}$: $\mathrm{u_1}$ S3:social $\mathrm{u_2}$ w $\in$ I. As customary in social network data models, the higher the weight, the closer we consider the two users to be.

**Extensibility** Depending on the application, it may be desirable to consider that two users satisfying some condition are involved in a social interaction. For instance, if two people have worked the same year for a company of less than 10 employees (such information may be in the RDF part of our instance), they must have *worked together*, which could be a social relationship. This is easily achieved with a query that retrieves all such user pairs (in SPARQL or in a more elaborate language [9] if the condition also carries over the documents), and builds a u workedWith u′ triple for each such pair of users. Then it suffices to add these triples to the instance, together with the triple: workedWith $\prec_{\mathrm{sp}}$ S3:social.

## 2.3 Documents and fragments

We consider that content is created under the form of structured, tree-shaped *documents*, e.g., XML, JSON, etc. A document is an unranked, ordered tree of *nodes*. Let $N$ be a set of node names (for instance, the set of allowed XML element and attribute names, or the set of node names allowed in JSON). Any node has a *URI*. We denote by $D \subset U$ the set of all node URIs. Further, each node has a *name* from $N$, and a *content*, which we view as *a set of keywords* from $\mathcal{K}$: we consider each text appearing in a document has been broken into words, stop words have been removed, and the remaining words have been stemmed to obtain our version of the node's text content. For example, in Figure 1, the text of $d_1$ might become {"M.S.", "UAlberta", "2012"}.

We term any subtree rooted at a node in document $d$ a *fragment* of $d$, implicitly defined by the URI of its root node. The set of fragments (nodes) of a document $d$ is denoted $Frag(d)$. We may use $f$ to refer interchangeably to a fragment or its URI. If $f$ is a fragment of $d$, we say $d$ is an *ancestor* of $f$.

To simplify, *we use document and fragment interchangeably*; both are identified by the URI of their unique root node.

**Document-derived triples** We capture the *structural relationships* between documents, fragments and keywords through a set of RDF statements using S3-specific properties. We introduce the RDF class S3:doc corresponding to the documents, and we translate:

- each d $\in D$ into the I triple d type S3:doc;
- each document d $\in D$ and fragment rooted in a node n of d into n S3:partOf d;
- each node n and keyword k appearing in the content of n into n S3:contains k;
- each node n whose name is m, into n S3:nodeName m.

EXAMPLE 2.1. *Based on the sample document shown in Figure 1, the following triples are part of* I*:*

$$\begin{array}{ll} \mathrm{d_{0.3.2}\ S3{:}partOf\ d_{0.3}} & \mathrm{d_1\ S3{:}contains\ "M.S."} \\ \mathrm{d_{0.3}\ S3{:}partOf\ d_0} & \mathrm{d_1\ S3{:}nodeName\ text} \end{array}$$

The following constraints, part of I, model the natural relationships between the S3:doc class and the properties introduced above:

$$\begin{array}{ll} \mathrm{S3{:}partOf\ \hookleftarrow_d\ S3{:}doc} & \mathrm{S3{:}partOf\ \hookrightarrow_r\ S3{:}doc} \\ \mathrm{S3{:}contains\ \hookleftarrow_d\ S3{:}doc} & \mathrm{S3{:}nodeName\ \hookleftarrow_d\ S3{:}doc} \end{array}$$

which read: the relationship S3:partOf connects pairs of fragments (or documents); S3:contains describes the content of a fragment; and S3:nodeName associates names to fragments.

[1]One could generalize this to support inference over triples of any weight, leading to e.g., "$u_1$ is of type Person with a weight of 0.5", in the style of probabilistic databases.

[2]One could also allow a keyword $k' \in Ext(k)$ which is only close to (but not a specialization of) $k$, e.g., "student″ in $Ext$("graduate″), at the cost of a loss of precision in query results. We do not pursue this alternative here, as we chose to follow standard RDF semantics.

**Fragment position** We will need to assess how closely related a given fragment is to one of its ancestor fragments. For that, we use a function $pos(d, f)$ which returns the *position* of fragment $f$ within document $d$. Concretely, $pos$ can be implemented for instance by assigning Dewey-style IDs to document nodes, as in [19, 22]. Then, $pos(d, f)$ returns the list of integers $(i_1, \dots, i_n)$ such that the path starting from $d$'s root, then moving to its $i_1$-th child, then to this node's $i_2$-th child etc. ends in the root of the fragment $f$. For instance, in Figure 1, $pos(d_{0.3.2}, d_0)$ may be $(3, 2)$.

## 2.4 Relations between structure, semantics, users

We now show how dedicated `S3` classes and properties are used to encode all kinds of connections between users, content, and semantics in a single `S3` instance.

**Tags** A typical user action in a social setting is to *tag* a data item, reflecting the user's opinon that the item is related to some concept or keyword used in the tag. We introduce the special class S3:relatedTo to *account for the multiple ways in which a user may consider that a fragment is related to a keyword.* We denote by $T$ the set of all tags.

For example, in Figure 1, $u_4$ tags $d_{0.5.1}$ with the keyword "university", leading to the triples:

$$\begin{array}{ll} \mathrm{a\ type\ S3{:}relatedTo} & \mathrm{a\ S3{:}hasSubject\ d_{0.5.1}} \\ \mathrm{a\ S3{:}hasKeyword\ \text{“university”}} & \mathrm{a\ S3{:}hasAuthor\ u_4} \end{array}$$

In this example, a is a *tag* (or annotation) resource, encapsulating the various tag properties: its content, who made it, and on what. The tag subject (the value of its S3:hasSubject property) is either a document or another tag. The latter allows to express *higher-level annotations*, when an annotation (tag) can itself be tagged.

A tag may lack a keyword, i.e., it may have no S3:hasKeyword property. Such no-keyword tags model *endorsement* (support), such as `like` on Facebook, `retweet` on Twitter, or +1 on Google+.

Tagging may differ significantly from one social setting to another. For instance, star-based rating of restaurants is a form of tagging, topic-based annotation of text by expert human users is another, and similarly a natural language processing (NLP) tool may tag a text snippet as being about some entity. Just like the S3:social property can be specialized to model arbitrary social connections between users, subclasses of S3:relatedTo can be used to model different kinds of tags. For instance, assuming $a_2$ is a tag produced by a NLP software, this leads to the I triples:

$$\begin{array}{c} \mathrm{a_2\ type\ NLP{:}recognize} \\ \mathrm{NLP{:}recognize} \prec_{\mathrm{sc}} \mathrm{S3{:}relatedTo} \end{array}$$

**User actions on documents** Users *post* (or author, or publish) content, modeled by the dedicated property S3:postedBy. Some of this content may be *comments* on (or replies / answers to) other fragments; this is encoded via the property S3:commentsOn.

EXAMPLE 2.2. *In Figure 1, $d_2$ is posted by $u_3$, as a comment on $d_{0.3.2}$, leading to the following* I *triples:*

$$\mathrm{d_2\ S3{:}postedBy\ u_3} \qquad \mathrm{d_2\ S3{:}commentsOn\ d_{0.3.2}}$$

As before, we view any concrete relation between documents e.g., *answers to*, *retweets*, *comments on*, *is an old version of* etc. as a specialization (sub-property) of S3:commentsOn; the corresponding connections lead to implicit S3:commentsOn triples, as explained in Section 2.1. Similarly, forms of authorship connecting users to their content are modeled by specializing S3:postedBy. This allows integrating (querying together) many social networks over partially overlapping sets of URIs, users and keywords.

**Inverse properties** As syntactic sugar, to simplify the traversal of connections between users and documents, we introduce a set of *inverse properties*, denoted respectively S3:$\overline{\text{postedBy}}$, S3:$\overline{\text{commentsOn}}$, S3:$\overline{\text{hasSubject}}$ and S3:$\overline{\text{hasAuthor}}$, with the straightforward semantics: $\mathrm{s\ \bar{p}\ o \in I}$ iff $\mathrm{o\ p\ s \in I}$ where $\bar{\mathrm{p}}$ is the inverse property of p. For instance, $\mathrm{u_0}$ S3:$\overline{\text{friend}}$ $\mathrm{u_1}$ in Figure 1.

| Class | Semantics |
|---|---|
| S3:user | the users (the set of its instances is $\Omega$) |
| S3:doc | the documents (the set of its instances is $D$) |
| S3:relatedTo | generalization of item "tagging" with keywords (the set of all instances of this class is $T$: the set of tags) |

| Property | Semantics |
|---|---|
| S3:postedBy | connects users to the documents they posted |
| S3:commentsOn | connects a comment with the document it is about |
| S3:partOf | connects a fragment to its parent nodes |
| S3:contains | connects a document with the keyword(s) it contains |
| S3:nodeName | asserts the name of the root node of document |
| S3:hasSubject | specifies the subject (document or tag) of a tag |
| S3:hasKeyword | specifies the keyword of a tag |
| S3:hasAuthor | specifies the poster of a tag |
| S3:social | generalization of social relationships in the network |

**Table 2: Classes and properties in the `S3` namespace.**

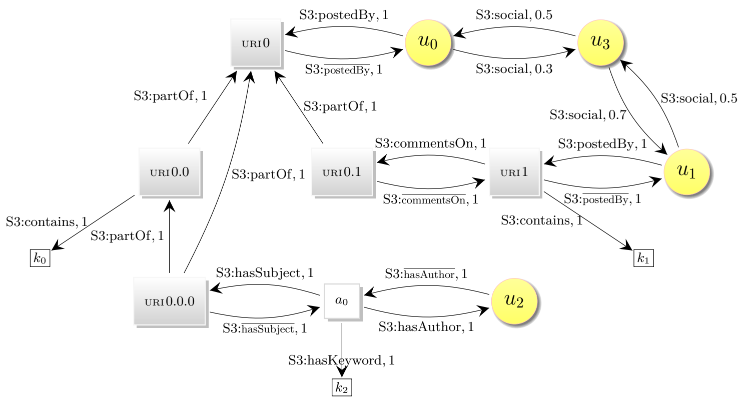

**Figure 3: Sample `S3` instance I.**

Table 2 summarises the above `S3` classes and properties, while Figure 3 illustrates an I instance.

## 2.5 Social paths

We define here social paths on I, established either through explicit social links or through user interactions. We call **network edges** those I edges encapsulating quantitative information on the links between user, documents and tags, i.e., *edges whose properties are in the namespace* S3 *other than* S3:partOf, and whose subjects and objects are either users, documents, or tags. For instance, in Figure 3, $\mathrm{u_1}$ S3:social $\mathrm{u_3}$ 0.5 and URI0 S3:$\overline{\text{postedBy}}$ $\mathrm{u_1}$ are network edges; URI0.0 S3:contains $\mathrm{k_0}$ and URI0.1 S3:partOf URI0 are not. The intuition behind the exclusion of S3:partOf is that *structural relations between fragments, or between fragments and keywords, merely describe data content and not an interaction.* However, if two users comment on the same fragment, or one comments on a fragment of a document posted by the other (e.g., $u_2$ and $u_0$ in Figure 1), this is form of social interaction.

When two users interact with unrelated fragments of a same document, such as $u_3$ and $u_4$ on disjoint subtrees of $d_0$ in Figure 1, this does not establish a social link between $u_3$ and $u_4$, since they may not even have read the same text[3]. We introduce:

DEFINITION 2.2 (DOCUMENT VERTICAL NEIGHBORHOOD). *Two documents are* vertical neighbors *if one of them is a fragment of the other. The function* `neigh`: $U \to 2^U$ *returns the set of vertical neighbors of an URI.*

[3]To make such interactions count as social paths would only require simple changes to the path normalization introduced below.

In Figure 3, $\textsc{uri}0$ and $\textsc{uri}0.0.0$ are vertical neighbors, so are $\textsc{uri}0$ and $\textsc{uri}0.1$, but $\textsc{uri}0.0.0$ and $\textsc{uri}0.1$ are not. In the sequel, due to the strong connections between nodes in the same vertical neighborhood, **we consider (when describing and exploiting social paths) that a path entering through any of them can exit through any other**; a vertical neighborhood acts like a single node only and exactly from the perspective of a social path[4]. We can now define social paths:

DEFINITION 2.3 (SOCIAL PATH). *A* social path *(or simply a* path*) in* I *is a chain of network edges such that the end of each edge and the beginning of the next one are* either the same node, or vertical neighbors.

We may also designate a path simply by *the list of nodes it traverses*, when the edges taken along the path are clear. In Figure 3, $u_2 \xrightarrow{\mathrm{u_2\ S3:\overline{hasAuthor}\ a_0\ 1}} a_0 \xrightarrow{\mathrm{a_0\ S3:hasSubject\ URI0.0.0\ 1}} \textsc{uri}0.0.0 \dashrightarrow \textsc{uri}0 \xrightarrow{\mathrm{URI0\ S3:\overline{postedBy}\ u_0\ 1}} u_0$ is an example of such a path (the dashed line: $\textsc{uri}0.0.0 \dashrightarrow \textsc{uri}0$, is not an edge in the path but a connection between vertical neighbors, $\textsc{uri}0.0.0$ been the end of an edge and $\textsc{uri}0$ the begining of the next edge). Also, in this Figure, there is no social path going from $u_2$ to $u_1$ avoiding $u_0$, because it is not possible to move from $\textsc{uri}0.1$ to $\textsc{uri}0.0.0$ through a vertical neighborhood.

**Social path notations** The set of *all social paths from a node $x$ (or one of its vertical neigbours) to a node $y$ (or one of its vertical neighbors)* is denoted $x \rightsquigarrow y$. The length of a path $p$ is denoted $|p|$. The restriction of $x \rightsquigarrow y$ to paths of length exactly $n$ is denoted $x \rightsquigarrow_n y$, while $x \rightsquigarrow_{\leq n} y$ holds the paths of at most $n$ edges.

**Path normalization** To harmonize the weight of each edge in a path depending on its importance, we introduce path normalization, which modifies the weights of a path's edge as follows. Let $n$ be the ending point of a social edge in a path, and $e$ be the *next* edge in this path. The normalized weight of $e$ *for this path*, denoted $e.n\_w$, is defined as:

$$e.n\_w = e.w / \sum_{e' \in out(\texttt{neigh}(n))} e'.w$$

where $e.w$ is the weight of $e$, and $out(\texttt{neigh}(n))$ the set of network edges outgoing from any vertical neighbor of $n$. This normalizes the weight of $e$ w.r.t. the weight of edges outgoing *from any vertical neighbor of* $n$. Observe that $e.n\_w$ depends on $n$, however $e$ does not necessarily start in $n$, but in any of its vertical neighbors. Therefore, $e.n\_w$ indeed depends on the path (which determines the vertical neighbor $n$ of $e$'s entry point).

*In the following, we assume all social paths are normalized.*

EXAMPLE 2.3. *In Figure 3, consider the path:*

$$p = u_0 \xrightarrow{\mathrm{u_0\ S3:postedBy\ URI0\ 1}} \textsc{uri}0 \dashrightarrow$$
$$\textsc{uri}0.0.0 \xrightarrow{\mathrm{URI0.0.0\ S3:\overline{hasSubject}\ a_0\ 1}} a_0$$

*Its first edge is normalized by the edges leaving $u_0$: one leading to* $\textsc{uri}0$ *(weight* $1$*) and the other leading to $u_3$ (weight* $0.3$*). Thus, its normalised weight is* $1/(1 + 0/3) = 0.77$.

*Its second edge exits* $\textsc{uri}0.0.0$ *after a vertical neighborhood traversal* $\textsc{uri}0 \dashrightarrow \textsc{uri}0.0.0$*. It is normalized by the edges leaving* $\texttt{neigh}(\textsc{uri}0)$*, i.e., all the edges leaving a fragment of* $\textsc{uri}0$*. Its normalised weight is* $1/(1 + 1 + 1 + 1) = 0.25$.

S3 **meets the requirements** from Section 1, as follows. Genericity, extensibility and interoperability (**R6**) are guaranteed by the reliance on the Web standards RDF (Section 2.1) and XML/JSON (Section 2.3). These enable specializing the S3 classes and properties, e.g., through application-dependent queries (see Extensibility in Section 2.2). Our document model (Section 2.3) meets requirement **R2**; the usage of RDF (Section 2.1) ensures **R3**, while the relationships introduced in Section 2.4 satisfy **R1** as well as **R4** (higher-level tags). For what concerns **R5** (formal semantics), the data model has been described above; we consider queries next.

## 3. QUERYING AN S3 INSTANCE

Users can search S3 instances through keyword queries; the answer consists of the $k$ top-score fragments, according to a joint structural, social, and semantic score. Section 3.1, defines queries and their answers. After some preliminaries, we introduce a *generic score*, which can be instantiated in many ways, and a set of *feasibility conditions* on the score, which suffice to ensure the termination and correctness of our query answering algorithm (Section 3.3). We present our concrete score function in Section 3.4.

### 3.1 Queries

S3 instances are queried as follows:

DEFINITION 3.1 (QUERY). *A query is a pair $(u, \phi)$ where $u$ is a user and $\phi$ is a set of keywords.*

We call $u$ the *seeker*. We define the top-$k$ answers to a query as the $k$ documents or fragments thereof with the highest scores, further satisfying the following constraint: the presence of a document or fragment at a given rank precludes the inclusion of its vertical neighbors at lower ranks in the results[5]. As customary, top-$k$ answers are ranked using a score function $s(q, d)$ that returns for a document $d$ and query $q$ a value in $\mathbb{R}$, based on the graph I.

DEFINITION 3.2 (QUERY ANSWER). *A top-$k$ answer to the query $q$ using the score $s$, denoted $T_{k,s}(q)$, is recursively defined as a top-$k{-}1$ answer, plus a document with the best score among those which are neither fragments nor ancestors of the documents in the top-$k{-}1$ answer.*

Observe that a query answer may not be unique. This happens as soon as several documents have equal scores for the query, and this score happens to be among the $k$ highest.

### 3.2 Connecting query keywords and documents

Answering queries over I requires finding best-scoring documents, based on the *direct and indirect connections* between documents, the seeker, and search keywords. The connection can be direct, for instance, when the document contains the keyword, or indirect, when a document is connected by a chain of relationships to a search keyword $k$, or to some keyword from $k$'s extension.

We denote the **set of direct and indirect connections between a document** $d$ **and a keyword** $k$ by $con(d, k)$. It is a set of three-tuples $(type, frag, src)$ such that:

- $type \in \{\mathrm{S3{:}contains}, \mathrm{S3{:}relatedTo}, \mathrm{S3{:}commentsOn}\}$ is the **type** of the connection,
- $f \in Frag(d)$ is the **fragment** of $d$ (possibly $d$ itself) due to which $d$ is involved in this connection,
- $src \in \Omega \cup D$ (users or documents) is the **source** (origin) of this connection (see below).

Below we describe the possible situations which create connections. Let $d, d'$ be documents or tags, and $f, f'$ be fragments of

[4]In other contexts, e.g., to determine their relevance w.r.t. a query, vertical neigbors are considered separately.

[5]This assumption is standard in XML keyword search, e.g., [6].

$d$ and $d'$, respectively[6]. Further, let $k, k'$ be keywords such that $k' \in Ext(k)$, and $src \in \Omega \cup D$ be a user or a document.

**Documents connected to the keywords of their fragments** If the fragment $f$ contains a keyword $k$, then:

$$(\text{S3:contains}, f, d) \in con(d, k)$$

which reads: "*$d$ is connected to $k$ through a* S3:contains *relationship due to $f$*". This connection holds even if $f$ contains not $k$ itself, but some $k' \in Ext(k)$. For example, in Figure 1, if the keyword "university" appears in the fragment whose URI is $d_{2.7.5}$, then $con(d_2, \text{"}university\text{"})$ includes $(\text{S3:contains}, d_{2.7.5}, d_2)$. Observe that a given $k'$ and $f$ may lead to many connections, if $k'$ specializes several keywords and/or if $f$ has many ancestors.

**Connections due to tags** For every tag $a$ of the form

|  |  |
|---|---|
| a type S3:relatedTo | a S3:hasSubject f |
| a S3:hasAuthor src | a S3:hasKeyword k′ |

$con(d, k)$ includes $(\text{S3:relatedTo}, f, src)$. In other words, whenever a fragment $f$ of $d$ is tagged by a source $src$ with a specialization of the keyword $k$, this leads to a S3:relatedTo connection between $d$ and $k$ due to $f$, whose source is the tag author $src$. For instance, the tag $a$ of $u_4$ in Figure 1 creates the connection $(\text{S3:relatedTo}, d_{0.5.1}, u_4)$ between $d_0$ and "university".

More generally, if a tag $a$ on fragment $f$ has *any type of connection* (not just S3:hasKeyword) to a keyword $k$ due to source $src$, this leads to a connection $(\text{S3:relatedTo}, f, src)$ between $d$ and $k$. The intuition is that the tag adds its connections to the tagged fragment and, transitively, to its ancestors. (As the next section shows, the importance given to such connections decreases as the distance between $d$ and $f$ increases.)

If the tag $a$ on $f$ is a simple endorsement (it has no keyword), the tag inherits $d$'s connections, as follows. Assume $d$ has a connection of type $type$ to a keyword $k$: then, $a$ also has a $type$ connection to $k$, whose source is $src$, the tag author. The intuition is that when $src$ endorses (`likes`, `+1s`) a fragment, $src$ agrees with its content, and thus connects the tag, to the keywords related to that fragment and its ancestors. For example, if a user $u_5$ endorsed $d_0$ in Figure 1 through a no-keyword tag $a_5$, the latter tag is related to "university" through: $(\text{S3:relatedTo}, d_{0.5.1}, u_5)$.

**Connections due to comments** When a comment on $f$ is connected to a keyword, this also connects any ancestor $d$ of $f$ to that keyword; the connection source carries over, while the type of $d$'s connection is S3:commentsOn. For instance, in Figure 1, since $d_2$ is connected to "university" through $(\text{S3:contains}, d_{2.7.5}, d_2)$ and since $d_2$ is a comment on $d_{0.3.2}$, it follows that $d_0$ is also related to "university" through $(\text{S3:commentsOn}, d_{0.3.2}, d_2)$.

## 3.3 Generic score model

We introduce a set of proximity notions, based on which we state the conditions to be met by a score function, for our query evaluation algorithm to compute a top-k query answer.

**Path proximity** We consider a measure of proximity *along one path*, denoted $\overrightarrow{prox}$, between 0 and 1 for any path, such that:

- $\overrightarrow{prox}(()) = 1$, i.e., the proximity is maximal on an empty path (in other words, from a node to itself),
- for any two paths $p_1$ and $p_2$, such that the start point of $p_2$ is in the vertical neighborhood of the end point of $p_1$:

$$\overrightarrow{prox}(p_1 || p_2) \leqslant min(\overrightarrow{prox}(p_1), \overrightarrow{prox}(p_2)),$$

where $||$ denotes path concatenation. This follows the intuition that proximity along a concatenation of two paths is at most the one along each of these two components paths: proximity can only decrease as the path gets longer.

**Social proximity** associates to two vertices connected by at least one social path, a comprehensive measure over *all the paths* between them. We introduce such a global proximity notion, because different paths traverse different nodes, users, documents and relationships, all of which may impact the relation between the two vertices. Considering all the paths gives a *qualitative* advantage to our algorithm, since it enlarges its knowledge to the types and strength of all connections between two nodes.

DEFINITION 3.3 (SOCIAL PROXIMITY). *The social proximity measure $prox : (\Omega \cup D \cup T)^2 \to [0, 1]$, is an aggregation along all possible paths between two users, documents or tags, as follows:*

$$prox(a, b) = \oplus_{path}(\{(\overrightarrow{prox}(p), |p|),\ p \in a \rightsquigarrow b\}),$$

*where $|.|$ is the number of vertices in a path and $\oplus_{path}$ is a function aggregating a set of values from $[0, 1] \times \mathbb{N}$ into a single scalar value.*

Observe that the set of all paths between two nodes may be infinite, if the graph has cycles; this is often the case in social graphs. For instance, in Figure 3, a cycle can be closed between $(u_0, \text{URI}0, u_0)$. Thus, in theory, the score is computed over a potentially infinite set of paths. However, in practice, our algorithm works with *bounded social proximity* values, relying only on paths of a bounded length:

$$prox^{\leq n}(a, b) = \oplus_{path}(\{(\overrightarrow{prox}(p), |p|),\ p \in a \rightsquigarrow_{\leq n} b\})$$

Based on the proximity measure, and the connections between keywords and documents introduced in Section 3.2, we define:

DEFINITION 3.4 (GENERIC SCORE). *Given a document d and a query $q = (u, \phi)$, the score of d for q is:*

$$score(d, (u, \phi)) = \oplus_{gen} (\{(k, type, pos(d, f), prox(u, src)) \\ |k \in \phi,\ (type, f, src) \in con(d, k)\})$$

*where $\oplus_{gen}$ is a function aggregating a set of (keyword, relationship type, importance of fragment $f$ in d, social proximity) tuples into a value from $[0, 1]$.*

Importantly, the above score *reflects the semantics, structure, and social content of the* `S3` *instance*, as follows.

First, $\oplus_{gen}$ aggregates over the keywords in $\phi$. Recall that tuples from $con(d, k)$ account not only for $k$ but also for keywords $k' \in Ext(k)$. This is how semantics is injected into the score.

Second, the score of $d$ takes into account the relationships between fragments $f$ of $d$, and keywords $k$, or $k' \in Ext(k)$, by using the sequence $pos(d, f)$ (Section 2.3) as an indication of the structural importance of the fragment within the document. If the sequence is short, the fragment is likely a large part of the document. Document structure is therefore taken into account here both *directly* through *pos*, and *indirectly*, since the *con* tuples also propagate relationships from fragments to their ancestors (Section 3.2).

Third, the score takes into account the social component of the graph through $prox$: this accounts for the relationships between the seeker $u$, and the various parties (users, documents and tags), denoted *src*, due to which $f$ may be relevant for $k$.

**Feasibility properties** For our query answering algorithm to converge, the generic score model must have some properties which we describe below.

**1. Relationship with path proximity** This refers to the relationship between path proximity and score. First, the score should only

[6] We here slightly extend notations, since tags do not have fragments: if $d$ is a tag, we consider that its only fragment is $d$.

*increase* if one adds *more paths* between a seeker and a data item. Second, the contribution of the paths of length $n \in \mathbb{N}$ to the social proximity can be expressed using the contributions of shorter "prefixes" of these paths, as follows. We denote by $ppSet^n(a, b)$ the set of the path proximity values for all paths from $a$ to $b$ of length $n$:

$$ppSet^n(a, b) = \{\overrightarrow{prox}(p) \,|\, p \in a \rightsquigarrow_n b\}$$

Then, the first property is that there exists a function $U_{prox}$ with values in $[0, 1]$, taking as input $(i)$ the bounded social proximity for path of length at most $n-1$ , $(ii)$ the proximity along paths of length $n$, and $(iii)$ the length $n$, and such that:

$$prox^{\leq n}(a, b) = prox^{\leq n-1}(a, b) + U_{prox}(prox^{\leq n-1}(a, b), ppSet^n(a, b), n)$$

**2. Long paths attenuation** The influence of social paths should decreases as they get longer; intuitively, the farther away two items are, the weaker their connection and thus their influence on the score. More precisely, there exists a bound $B^{>n}_{prox}$ tending to 0 as $n$ grows, and such that:

$$B^{>n}_{prox} \geq prox - prox^{\leq n}$$

**3. Score soundness** The score of a document should be positively correlated with the social proximity from the seeker to the document fragments that are relevant for the query.

Denoting $score_{[g]}$ the score where the proximity function $prox$ is replaced by a continuous function $g$ having the same domain $(\Omega \cup D \cup T)^2$, $g \mapsto score_{[g]}$ must be monotonically increasing and continuous for the uniform norm.

**4. Score convergence**

This property bounds the score of a document and shows how it relates to the social proximity. It requires the existence of a function $B_{score}$ which takes a query $q = (u, \phi)$ and a number $B \geq 0$, known to be an upper bound on the social proximity between the seeker and any source: for any $d$, query keyword $k$, and $(type, f, src) \in con(d, k)$, we know that $prox(u, src) \leq B$. $B_{score}$ must be positive, and satisfy, for any $q$:

- for any document $d$, $score(d, q) \leq B_{score}(q, B)$;
- $\lim_{B\to 0}(B_{score}(q, B)) = 0$ (tends to 0 like $B$).

We describe a concrete *feasible score*, i.e., having the above properties, in the next section.

## 3.4 Concrete score

We start by instantiating $\overrightarrow{prox}$, $prox$ and $score$.

**Social proximity** Given a path $p$, we define $\overrightarrow{prox}(p)$ as the product of the normalized weights (recall Section 2.5) found along the edges of $p$. We define our concrete social proximity function $prox(a, b)$ as a weighted sum over all paths from $a$ to $b$:

$$prox(a, b) = C_\gamma \times \sum_{p \in a \rightsquigarrow b} \frac{\overrightarrow{prox}(p)}{\gamma^{|p|}}$$

where $\gamma > 1$ is a scalar coefficient, and $C_\gamma = \frac{\gamma-1}{\gamma}$ is introduced to ensure that $prox \leq 1$. Recall that by Definition 3.3, $prox$ requires a $\oplus_{path}$ aggregation over the (social proximity, length) pairs of the paths between the two nodes. Hence, this concrete social proximity corresponds to choosing:

$$\oplus_{path}(S) = C_\gamma \times \sum_{(sp, len) \in S} \frac{sp}{\gamma^{len}}$$

where $(sp, len)$ is a (social proximity, length) pair from its input.

EXAMPLE 3.1. ***Social proximity*** *Let us consider in Figure 3 the social proximity from* $u_0$ *to* URI0*, using the* $\overrightarrow{prox}$ *and* $\oplus_{path}$ *previously introduced. An edge connects* $u_0$ *directly to* URI0*, leading to the normalized path* $p$*:*

$$p = u_0 \xrightarrow{u_0 \text{ S3:postedBy URI0 } \frac{1}{1+0.3}} \text{URI0}$$

*which accounts for a partial social proximity:*

$$prox^{\leq 1}(u_0, \text{URI0}) = \frac{\overrightarrow{prox}(p)}{\gamma^{|p|}} = \frac{1/(1+0.3)}{\gamma^1}$$

This social proximity generalizes Katz distance [17]; other common distances may be used, e.g., SimRank [14].

**Score function** We define a simple concrete S3 *score function* which, for a document $d$, is the product of the scores of each query keyword in $d$. The score of a keyword is summed over all the connections between the keyword and the document. The weight for a given connection and keyword only depends on the *social distance between the seeker and the sources of the keyword*, and the *structural distance between the fragment involved in this relation and* $d$, namely the length of $pos(d, f)$. Both distances decrease exponentially as the path length grows. Formally:

DEFINITION 3.5 (S3$_k$ SCORE). *Given a query* $(u, \phi)$*, the* S3$_k$ *score of a document d for the query is defined as:*

$$score(d, (u, \phi)) = \prod_{k \in \phi} \Big( \sum_{(type, f, src) \in con(d,k)} \eta^{|pos(d,f)|} \times prox(u, src) \Big)$$

*for some* damping factor $\eta < 1$.

Recall from Definition 3.4 that an aggregation function $\oplus_{gen}$ combines the contributions of (keyword, relationship type, importance, social proximity) tuples in the score. The above definition corresponds to the following $\oplus_{gen}$ aggregator:

$$\oplus_{gen}(S) = \prod_{k \in \phi} \Big( \sum_{\substack{rel, prox \\ \exists type, (k, type, rel, prox) \in S}} \eta^{|rel|} \times prox \Big)$$

Note that if we ignore the social aspects and restrict ourselves to top-k search on documents (which amounts to $prox = 1$), $\oplus_{gen}$ gives the best score to the lowest common ancestor (LCA) of the nodes containing the query keywords. Thus, our score extends typical XML IR works, e.g., [6] (see also Section 6).

Obviously, there are many possible ways to define $\oplus_{gen}$ and $\oplus_{path}$, depending on the application. In particular, different types of connections may not be accounted for equally; our algorithm only requires a *feasible score* (with the feasibility properties).

**THEOREM** 3.1 (SCORE FEASIBILITY). *The* S3$_k$ *score function (Definition 3.5) has the feasibility properties (Section 3.3).*

The proof appears in our technical report [3].

## 4. QUERY ANSWERING ALGORITHM

In this section, we describe our *Top-k* algorithm called S3$_k$, which computes the answer to a query over an S3 instance using our S3$_k$ score, and formally state its correctness.

### 4.1 Algorithm

The main idea, outlined in Algorithm 1, is the following. The instance is explored starting from the seeker and going to other vertices (users, documents, or resources) at increasing distance. At the $n$-th iteration, the I vertices explored are those connected to the seeker by at least a path of length at most $n$. We term *exploration border* the set of graph nodes reachable by the seeker through a path of length exactly $n$. Clearly, the border changes as $n$ grows.

During the exploration, documents are collected in a set of *candidate* answers. For each candidate $c$, we maintain a score interval: its *currently known lowest possible score*, denoted $c.lower$, and its

**Algorithm 1:** $\texttt{S3}_k$ – Top-$k$ algorithm.

```
Input : a query $q = (u, \phi)$
Output: the best $k$ answers to $q$ aver an S3 instance $I$, $T_{k,s}(q)$
$candidates \leftarrow [\,]$                                  // initially empty list
$discarded \leftarrow \emptyset$
$borderPath \leftarrow [\,]$
$allProx \leftarrow \delta_u$            // $\delta_u[v] = \begin{cases} 1 & \text{if } v = u \\ 0 & \text{otherwise} \end{cases}$
$threshold \leftarrow \infty$      // Best possible score of a document not yet
  explored, updated in ComputeCandidatesBounds
$n \leftarrow 0$
while not StopCondition($candidates$) do
    $n \leftarrow n + 1$
    ExploreStep()
    ComputeCandidatesBounds()
    CleanCandidatesList()
return $candidates[0, k-1]$
```

| $q = (u, \phi)$ | Query: seeker $u$ and keyword set $\phi$ |
|---|---|
| $k$ | Result size |
| $n$ | Number of iterations of the main loop of the algorithm |
| $candidates$ | Set of documents and/or fragments which are candidate query answers at a given moment |
| $discarded$ | Set of documents and/or fragments which have been ruled out of the query answer |
| $borderPath[v]$ | Paths from $u$ to $v$ explored at the last iteration ($a \rightsquigarrow_n v$) |
| $allProx[v]$ | Bounded social proximity ($prox^{\leq n}$) between the seeker $u$ and a node $v$, taking into account all the paths from $u$ to $v$ known so far |
| $connect[c]$ | Connections between the seeker and the candidate c: $connect[c] = \{(k, type, pos(d,f), src) \mid k \in \phi,\ (type, f, src) \in con(c,k)\}$ |
| $threshold$ | Upper bound on the score of the documents not visited yet |

**Table 3: Main variables used in our algorithms.**

**Algorithm 2:** Algorithm StopCondition

```
Input : $candidates$ set
Output: true if $candidates[0, k-1]$ is $T_{k,s}(q)$, false otherwise
if $\exists d, d' \in candidates[0, \ldots, k-1]$, $d \in$ neigh($d'$) then
    return false
$min\_topk\_lower \leftarrow \infty$
foreach $c \in candidates[0, \ldots, k-1]$ do
    $min\_topk\_lower \leftarrow \min(min\_topk\_lower, c.lower)$
$max\_non\_topk\_upper \leftarrow candidates[k].upper$
return $\max(max\_non\_topk\_upper, threshold) \leq$
  $min\_topk\_lower$                                   // Boolean result
```

*highest possible score*, denoted $c.upper$. These scores are updated as new paths between the seeker and the candidates are found. Candidates are kept sorted *by their highest possible score*; the $k$ first are the answer to the query when the algorithm stops, i.e., when no candidate document outside the current first $k$ can have an *upper bound* above the *minimum lower bound* within the top $k$ ranks.

Further, the search algorithm relies on three tables:

- $borderPath$ is a table storing, for a node $v$ in I, the set of paths of length $n$ between $u$ (the seeker) and $v$, where $n$ is the current distance from $u$ that the algorithm has traversed.
- $allProx$ is a table storing, for a node $v$ in I, the proximity between $u$ and $v$ taking into account all the paths known so far from $u$ to $v$. Initially, its value is 0 for any $v \neq u$.
- *connect* is a table storing for a candidate $c$ the set of connections (Section 3.2) discovered so far between the seeker and $c$

These tables are updated during the search. While they are defined on all the I nodes, we only compute them gradually, for the nodes on the exploration border.

**Termination condition** Of course, search should not explore the whole graph, but instead stop as early as possible, while returning

**Algorithm 3:** Algorithm ExploreStep

```
Update: $borderPath$ and $allProx$
if $n = 1$ then
    $borderPath \leftarrow out(\{u\})$
else
    foreach $v \in$ I do
        $newBorderPath[v] \leftarrow \emptyset$
    foreach $p \in borderPath$ do
        foreach network edge e in $out$(neigh($p.end$)) do
            $m \leftarrow e.target$
            if $m$ is a document or a tag then
                GetDocuments($m$)
            $newBorderPath[m].add(p||e)$
    $borderPath \leftarrow newBorderPath$
foreach $v \in$ I do
    $newAllProx[v] \leftarrow allProx[v] + U_{prox}(allProx[v],$
      $\{\overrightarrow{prox}(p), p \in borderPath[v]\}, n)$
$allProx \leftarrow newAllProx$
```

the correct result. To this aim, we maintain during the search an upper bound on the score of score of all documents unexplored so far, named $threshold$. Observe that we do not need to return the exact score of our results, and indeed we may never narrow down the (lower bound, upper bound) intervals to single numbers; we just need to make sure that no document unexplored so far is in among the top $k$. Algorithm 2 outlines the procedure to decide whether the search is complete: when ($i$) the candidate set does not contain documents such that one is a fragment of another, and ($ii$) no document can have a better score than the current top $k$.

**Any-time termination** Alternatively, the algorithm can be stopped at any time (e.g., after exhausting a time budget) by making it return the $k$ best candidates based on their current upper bound score.

**Graph exploration** Algorithm 3 describes one search step (iteration), which visits nodes at a social distance $n$ from the seeker. For the ones that are documents or tags, the GetDocuments algorithm (see hereafter) looks for related documents that can also be candidate answers (these are added to *candidates*); *discarded* keeps track of related documents with scores too low for them to be candidates. The $allProx$ table is also updated using the $U_{prox}$ function, whose existence follows from the first score feasibility property (Section 3.3), to reflect the knowledge acquired from the new exploration border ($borderPath$). Observe that Algorithm 3 computes $prox^{\leq n}(u, src)$ iteratively using the first feasibility property; at iteration $n$, $allProx[src] = prox^{\leq n}(u, src)$.

**Computing candidate bounds** The ComputeCandidateBounds algorithm (shown in [3]) maintains during the search the lower and upper bounds of the *candidates*, as well as *threshold*. A candidate's *lower* bound is computed as its score where its social proximity to the user[7] is approximated by its bounded version, based only on the paths explored so far:

$$\oplus_{gen}(\{(kw,\ type,\ pos(d,f),\ allProx[src]) \mid kw \in \phi,\ (type, f, src) \in con(d, kw)\})$$

This is a lower bound because, *during exploration, a candidate can only get closer to the seeker* (as more paths are discovered).

A candidate's *upper* bound is computed as its score, where the social proximity to the user is replaced by the sum between the bounded proximity and the function $B^{>n}_{prox}(u, src)$, whose existence follows from the long path attenuation property (Section 3.3).

[7]The actual (exact) social proximity requires a complete traversal of the graph; our algorithms work with approximations thereof.

The latter is guaranteed to offset the difference between the bounded and actual social proximity:

$$\oplus_{gen}(\{(kw,\, type,\, pos(d,f),\, allProx[src] + B^{>n}_{prox}(u, src)) \mid kw \in \phi, (type, f, src) \in con(d, kw)\})$$

The above bounds rely on $con(d,k)$, the set of all connections between a candidate $d$ and a query keyword $k$ (Section 3.2); clearly, the set is not completely known when the search starts. Rather, connections accumulate gradually in the *connect* table (Algorithm `GetDocuments`), whose tuples are used as approximate (partial) $con(d,k)$ information in `ComputeCandidateBounds`.

Finally, `ComputeCandidateBounds` updates the relevance threshold using the known bounds on $score$ and $prox$. The new bound estimates the best possible score of the unexplored documents.

**Cleaning the candidate set** Algorithm `CleanCandidateList` removes from $candidates$ documents that cannot be in the answer, i.e., those for which $k$ candidates with better scores are sure to exist, as well as those having a candidate neighbor with a better score. The algorithm is delegated to [3].

**Getting candidate documents** Given a candidate document or tag $x$, Algorithm `GetDocuments` checks whether some documents unexplored so far, reachable from $x$ through a chain of S3:partOf, S3:commentsOn, S3:$\overline{\text{commentsOn}}$, S3:hasSubject, or S3:$\overline{\text{hasSubject}}$ edges, are candidate answers. If yes, they are added to $candidates$ and the information necessary to estimate their score is recorded in $connect$. The algorithm is detailed in [3].

## 4.2 Correctness of the algorithm

The theorems below state the correctness of our algorithm for *any score function with the feasibility properties* identified in Section 3.3. The proofs are quite involved, and they are delegated to [3]. The core of the proofs is showing how the score feasibility properties entail a set of useful properties, in particular related to early termination (convergence).

THEOREM 4.1 (STOP CORRECTNESS). *When a stop condition is met, the first $k$ elements in candidates are a query answer.*

We say the tie of two equal-score documents $d, d'$ is *breakable* if examining a set of paths *of bounded length* suffices to decide their scores are equal. (In terms of our score feasibility properties, this amounts to $B^{>n}_{prox} = 0$ for some $n$). Our generic score function (Definition 3.5) does not guarantee all ties are breakable. However, any finite-precision number representation eventually brings the lower and upper bounds on $d$ and $d'$'s scores too close to be distinguished, de facto breaking ties.

THEOREM 4.2 (CORRECTNESS WITH BREAKABLE TIES). *If there exists a query answer of size $k$ and all ties are breakable then Algorithm 1 returns a query answer of size $k$.*

THEOREM 4.3 (ANYTIME CORRECTNESS). *Using anytime termination, Algorithm 1 eventually returns a query answer.*

In our experiments (Section 5), the threshold-based termination condition was always met, thus we never needed to wait for convergence of the lower and upper bound scores.

## 5. IMPLEMENTATION & EXPERIMENTS

We describe experiments creating and querying `S3` instances. We present the datasets in Section 5.1, while Section 5.2 outlines our implementation and some optimizations we brought to the search algorithm. We report query processing times in Section 5.3, study the quality of our returned results in Section 5.4, then we conclude.

$I_1$ (Twitter)

| | |
|---|---|
| Users | 492,244 |
| S3:social edges | 17 544 347 |
| Documents | 467,710 |
| Fragments (non-root) | 1,273,800 |
| Tags | 609,476 |
| Keywords | 28,126,940 |
| Tweets | 999,370 |
| Retweets | 85% |
| Reply to users' status | 6.9% |
| String-keyword associations extracted from DBpedia | 3,301,425 |
| S3:social edges per user having any (average) | 317 |
| Nodes (without keywords) | 2 972 560 |
| Edges (without keywords) | 24 554 029 |

$I_2$ (Vodkaster)

| | |
|---|---|
| Users | 5,328 |
| S3:social edges (vdk:follow) | 94,155 |
| Documents (movie comments) | 330,520 |
| Fragments (non-root) | 529,432 |
| Keywords | 3,838,662 |
| Movies | 20,022 |

$I_3$ (Yelp)

| | |
|---|---|
| Users | 366,715 |
| S3:social edges (yelp:friend) | 3,868,771 |
| Documents (reviews) | 2,064,371 |
| Keywords | 59,614,201 |
| Businesses | 61,184 |

**Figure 4: Statistics on our instances.**

## 5.1 Datasets, queries, and systems

**Datasets** We built three datasets, $I_1$, $I_2$, and $I_3$, based respectively on content from Twitter, Vodkaster and Yelp.

The instance $I_1$ was constructed starting from tweets obtained through the public streaming Twitter API. Over a one-day interval (from May 2nd 2014 16h44 GMT to May 3rd 2014 12h44 GMT), we gathered roughly one million tweets. From every tweet that is not a retweet, we created a document having three nodes $(i)$ a *text* node: from the *text* field of the tweet, we extracted named entities and words (using the Twitter NLP tools library [20]) and matched them against a general-purpose ontology we created from DBpedia (see below); $(ii)$ a *date* node, and $(iii)$ a *geo* node: if the tweet included a human readable location, we inserted it in this node. The RDF graph of our instance was built from DBPedia datasets, namely: *Mapping-based Types*, *Mapping-based Properties*, *Persondata* and *Lexicalizations Dataset*. These were chosen as they were the most likely to contain concepts (names, entities etc.) occurring in tweets. Tweet text was *semantically enriched* (connected to the RDF graph) as follows: within the *text* fields, we replaced each word $w$ for which a triple of the form u foaf:name w holds in the DBPedia knowledge base, by the respective URI $u$.

When a tweet $t'$ authored by user $u$ is a *retweet* of another tweet $t$, for each hashtag $h$ introduced by $t'$, we added to $I_1$ the triples: a type S3:relatedTo, a S3:hasSubject t, a S3:hasKeyword h and a S3:hasAuthor u. If a tweet $t''$ was a *reply* to another tweet $t$, we considered $t''$ a comment on $t$. Whenever $t$ was present in our dataset[8], we added the corresponding S3:commentsOn triple in $I_1$. The set of users $\Omega_{I_1}$ corresponds to the set of user IDs having posted tweets, and we created links between users as follows. We assigned to every pair of users $(a,b)$ a value $u_\sim(a,b) = t \cdot js_1(a,b) + (1-t) \cdot js_2(a,b)$, where $js_1$, $js_2$ give the Jaccard similarities of the *sets of keywords* appearing in each user's *posts*, respectively, in each user's *comments*. Whenever this similarity was above a threshold, we created an edge of weight $u_\sim$ between the two users. Through experiments on this dataset, we set the threshold to $0.1$.

The instance $I_2$ uses data from Vodkaster, a French social network dedicated to movies. The data comprises *follower* relations between the users and a list of comments on the movies, in French, along with their author. Whenever user $u$ follows user $v$ we in-

[8]The corpus may contain a re-tweet of a tweet we do not capture; this is unavoidable unless one has access to the full Twitter history.

cluded u vdk:follow v 1 in $\mathrm{I}_2$, where vdk:follow is a custom subproperty of S3:social. We translate the first comment on each film into a document; each additional comment was then considered a comment on the first. The text of each comment was stemmed, then each stemmed sentence was made a fragment of the comment.

The instance $\mathrm{I}_3$ is based on Yelp [29], a crowd-sourced reviews website about businesses. This dataset contains a list of textual reviews of businesses, and the friend list of each user. As for $\mathrm{I}_2$, we considered that the first review of a business is commented on by the subsequent reviews of the same business. For each user $u$ friend with user $v$, we added u yelp:friend v 1 to $\mathrm{I}_3$, where yelp:friend is a S3:social subproperty modeling social Yelp connections. Reviews were also semantically enriched using DBPedia.

Table 4 shows the main features of the three quite different data instances. $\mathrm{I}_1$ is by far the largest. $\mathrm{I}_2$ was not matched with a knowledge base since its content is in French; $\mathrm{I}_2$ and $\mathrm{I}_3$ have no tags.

**Queries** For each instance we created workloads of 100 queries, based on three independent parameters:

- $f$, the keyword frequency: either *rare*, denoted '$-$' (among the 25% least frequent in the document set), or *common*, denoted '$+$' (among the 25% most frequent)
- $l$, the number of keywords in the query: 1 or 5
- $k$, the expected number of results: 5 or 10

This lead to a total of 8 workloads, identified by $qset_{f,l,k}$, for each dataset. To further analyze the impact of varying $k$, we added 10 more workloads for $\mathrm{I}_1$, where $f \in \{+,-\}$, $l = 1$, and $k \in [1, 5, 10, 50]$ (used in Figure 7). *We stress here that injecting semantics in our workload queries, by means of keyword extensions (Definition 2.1), increased their size on average by* 50%.

**Systems** Our algorithms were fully implemented in Python 2.7; the code has about 6K lines. We stored some data tables in PostgreSQL 9.3, while others were built in memory, as we explain shortly. All our experiments were performed on a 4 cores Intel Xeon E3-1230 V2 @3.30GHz with 16Go of RAM, running Debian 8.1.

No existing system directly compares with ours, as we are the first to consider fine-granularity content search with semantics in a social network. To get at least a rough performance comparison, we used the Java-based code provided by the authors of the top-$k$ social search system described in [18], based on the UIT (user, item tag) model, and referred to as **TopkS** from now on. The data model of TopkS is rather basic, since its documents (*items*) have no internal structure nor semantics and tags have no semantic connection between them. Further, *(user, user, weight)* tuples reflect weighted links between users. TopkS computes a social score and a content-based score for each item; the overall item score is then $\alpha\times$ social score $+(1-\alpha)\times$ content score, where $\alpha$ is a parameter of TopkS.

We adapted our instances into TopkS's simpler data model. From $\mathrm{I}_1$, we created $\mathrm{I}'_1$ as follows: $(i)$ the relations between users were kept with their weight; $(ii)$ every tweet was merged with all its retweets and replies into a single item, and $(iii)$ every keyword $k$ in the content of a tweet that is represented by item $i$ posted by user $u$ led to introducing the (user, item, tag) triple $(u, i, k)$. To obtain $\mathrm{I}'_2$ and $\mathrm{I}'_3$, each movie or business becomes an item, each word extracted from a review leads to a (user, item, tag) tuple.

## 5.2 Implementation and optimizations

We briefly discuss our implementation, focusing on optimizations w.r.t. the conceptual description in Section 4.

The first optimization concerns the computation of $prox$, required for the score (Definition 3.5). While the score involves connections between documents and keywords found on any path, in practice `S3`$_k$ explores paths (and nodes) increasingly far from

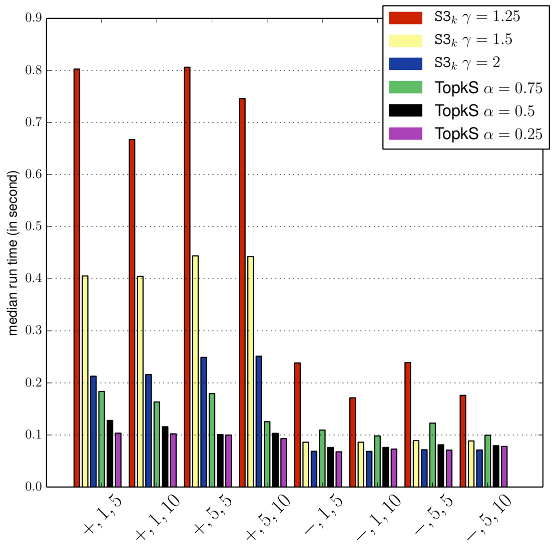

**Figure 5: Query answering times on $\mathrm{I}_1$ (Twitter).**

the seeker, and stores such paths in $borderPath$, which may grow very large and hurt performance. To avoid storing $borderPath$, we compute for each explored vertex $v$ the weighted sum over all paths of length $n$ from the seeker to this vertex:

$$borderProx(v,n) = \sum\nolimits_{p\in u\rightsquigarrow v, |p|=n} \frac{\overrightarrow{prox}(p)}{\gamma^n}$$

and compute $prox$ directly based on this value.

Furthermore, Algorithm `GetDocuments` considers documents reachable from $x$ through edges labeled S3:partOf, S3:$\overline{\text{commentsOn}}$, S3:commentsOn,S3:$\overline{\text{hasSubject}}$ or S3:hasSubject. Reachability by such edges defines a *partition* of the documents into *connected components*. Further, by construction of $con$ tuples (Section 3.2), connections carry over from one fragment to another, across such edges. Thus, a fragment matches the query keywords iff its component matches it, leading to an efficient pruning procedure: we compute and store the partitions, and test that each keyword (or extension thereof) is present in every component (instead of fragment). Partition maintenance is easy when documents and tags are added, and more expensive for deletions, but luckily these are rarer.

The query answering algorithm creates in RAM the $allProx$ table and two sparse matrices, computed only once: *distance*, encoding the graph of network edges in I (accounting for the vertical neighborhood), and *component*, storing the component of each fragment or tag. Thus, Algorithm 3, which computes $allProx$ and finds new components to explore, relies on efficient matrix and vector operations. For instance, the new distance vector $borderProx$ w.r.t. the seeker at step $n+1$ is obtained by multiplying the distance matrix with the previous distance vector from step $n$. The documents and the RDF graph, on the other hand are not stored in RAM, and are queried using a PostgreSQL database.

The search algorithm can be *parallelized* in two ways. First, within an iteration, we discover new documents in different components in parallel. Second, when $borderProx$ is available in the current iteration, we can start computing the next $borderProx$ using the distance matrix. More precisely, Algorithm 3 (`ExploreStep`) can be seen as consisting of two main blocks: $(i)$ computing the new $borderProx$ using the (fixed) distance matrix and the previous $borderProx$ (lines 1-12 except line 10); $(ii)$ computing $allProx$ using the new $borderProx$ and the previous $allProx$ (lines 13-16) plus the call to `GetDocuments` (line 10). The latter algorithm consists of two parts: $(iii)$ identifying the newly discovered components, respectively $(iv)$ testing the documents they contain. We used 8 concurrent threads, each running a task of one of the forms $(i)$-$(iv)$, above, and synchronized them with a custom

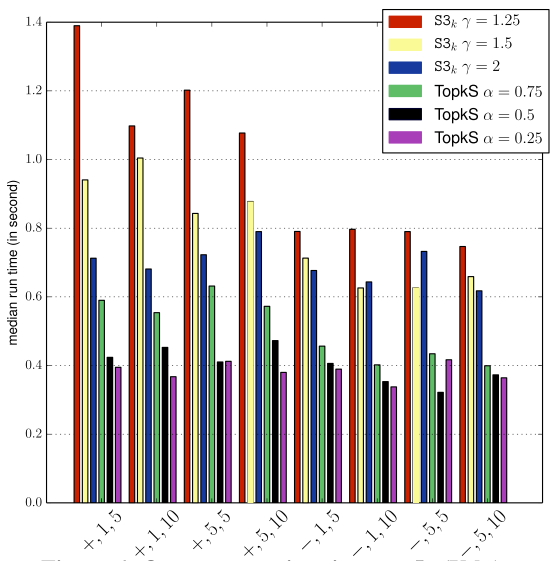

**Figure 6: Query answering times on $\mathrm{I}_3$ (Yelp).**

scheduler. This reduced the query answering time on average by a factor of 2.

## 5.3 Query answering times

Figures 5 and 6 show the running times of $\mathtt{S3}_k$ on the $\mathrm{I}_1$ and $\mathrm{I}_3$ instances; the results on the smaller instance $\mathrm{I}_2$ are similar [3]. We used different values of the $\gamma$ social proximity damping factor (Section 3.4) and the $\alpha$ parameter of TopkS. For each workload, we plot the average time (over its 100 queries). *All runs terminated by reaching the threshold-based stop condition* (Algorithm 2).

A first thing to notice is that while all running times are comparable, TopkS runs consistently faster. This is mostly due to the different proximity functions: our $prox$, computed from all possible paths, has a much broader scope than TopkS, which explores and uses only one (shortest) path. In turn, as we show later, we return a significantly *different* set of results, due to $prox$'s broader scope and to considering document structure and semantics.

Decreasing the $\gamma$ in $\mathtt{S3}_k$ reduces the running time. This is expected, as $\gamma$ gives more weight to nodes far from the seeker, whose exploration is costly. Similarly, *increasing* $\alpha$ in TopkS forces to look further in the graph, and affects negatively its performance.

The influence of $k$ is more subtle. When the number of candidates is low and the exploration of the graph is not too costly, higher $k$ values allow to include most candidates among the $k$ highest-scoring ones. This reduces the exploration needed to refine their bounds enough to clarify their relative ranking. In contrast, if the number of candidates is important and the exploration costly, a small $k$ value significantly simplifies the work. This can be seen in Figure 7 where, with frequent keywords, increasing $k$ does not affect the 3 fastest quartiles but significantly slows down the slowest quartile, since the algorithm has to look further in the graph.

The same figure also shows that rare-keyword workloads (whose labels start by $-$) are faster to evaluate than the frequent-keyword ones (workload labels starting with $+$). This is because finding rare keywords tends to require exploring longer paths. Social damping at the end of such paths is high, allowing to decide that possible matches found even farther from the seeker will not make it into the top-$k$. In contrast, matches for frequent keywords are found soon, while it is still possible that nearby exploration may significantly change their relative scores. In this case, more search and computations are needed before the top-$k$ elements are identified.

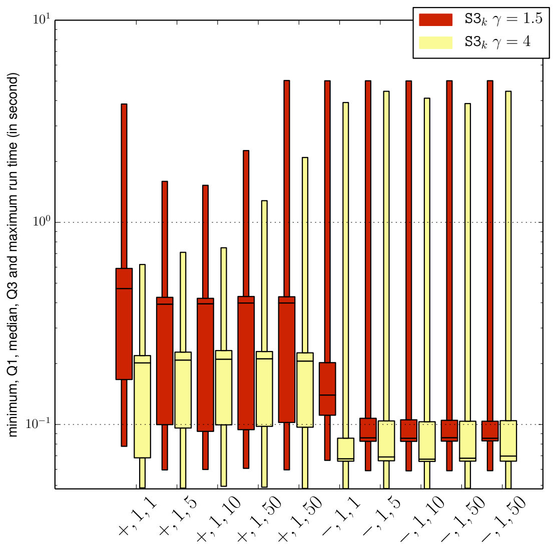

**Figure 7: Query answering times on $\mathrm{I}_1$ when varying $k$.**

| Measure \ Instance | $\mathrm{I}_1$ | $\mathrm{I}_2$ | $\mathrm{I}_3$ |
|---|---|---|---|
| Graph reachability | 12% | 23% | 41% |
| Semantic reachability | 83% | 100% | 78% |
| $L_1$ | 8% | 10% | 4% |
| Intersection size | 13.7% | 18.4% | 5.6% |

**Figure 8: Relations between $\mathtt{S3}_k$ and TopkS answers.**

## 5.4 Qualitative comparison

We compare now the answers of our $\mathtt{S3}_k$ algorithm and those of TopkS from a *qualitative* angle. $\mathtt{S3}_k$ follows links between documents to access further content, while TopkS does not; we term *graph reachability* the fraction of candidates reached by our algorithm which are not reachable by the TopkS search. Further, while $\mathtt{S3}_k$ takes into account semantics by means of semantic extension (Definition 2.1), TopkS only relies on the query keywords. We call *semantic reachability* the ratio between the number of candidates examined by an algorithm *without* expanding the query, and the number of candidates examined *with* query expansion. Observe that some $\mathtt{S3}_k$ candidates may be ignored by TopkS due to the latter's lack of support for *both* semantics and connections between documents. Finally, we report two measures of distance between the results of the two algorithms. The first is the *intersection size* i.e., the fraction of $\mathtt{S3}_k$ results that TopkS also returned. The second, $L_1$, is based on Spearman's well-known *foot rule* distance between lists, defined as:

$$L_1(\tau_1, \tau_2) = 2(k - |\tau_1 \cap \tau_2|)(k+1) + \sum_{i \in \tau_1 \cap \tau_2} |\tau_1(i) - \tau_2(i)| - \sum_{\substack{\tau \in \{\tau_1, \tau_2\} \\ i \in \tau \setminus (\tau_1 \cap \tau_2)}} \tau(i)$$

where $\tau_j(i)$ is the rank of item $i$ in the list $\tau_j$.

The averages of these 4 measures over the 8 workloads on each instance appear in Figure 8. The ratios are low, and show that different candidates translate in different answers (the low $L_1$ stands witness for this). Few $\mathtt{S3}_k$ results can be attained by an algorithm such as TopkS, which ignores semantics and relies only on the shortest path between the seeker and a given candidate.

## 5.5 Experiment conclusion

Our experiments have demonstrated first the ability of the $\mathtt{S3}$ data model to *capture very different social applications*, and to query them meaningfully, accounting for their structure and enriching them with semantics. Second, we have shown that $\mathtt{S3}_k$ *query answering can be quite efficient*, even though considering all paths between the seeker and a candidate answer slows it down w.r.t. simpler algorithms, which rely on a shortest-path model. We have experimentally verified the expected impact of the social damping factor $\gamma$ and of the result size $k$ on running time. Third, and most importantly, we have shown that taking into account in the relevance model the social, structured, and semantic aspects of the instance bring a *qualitative gain*, enabling meaningful results that would not have been reachable otherwise.

## 6. RELATED WORK

Prior work on *keyword search in databases* spreads over different research directions:

**Top-k search in a social environment** uses UIT models [18, 21, 30] we outlined in Section 1. Top-k query results are found based on a score function accounting for the presence of each keyword in the tags of a candidate item, and a simple social distance based on the length of the social edge paths; query answering algorithms are inspired from the general top-k framework of [8]. As documents are considered atomic, and relations between them are ignored, requirements **R1**, **R2** and **R4** are not met. Further, the lack of semantics also prevents **R5**. Recent developments tend to focus on performance and scalability, or the integration of more attributes such as locality or temporality [7, 16], without meeting the abovementioned requirements. Location and time can be added to generic scores but this is outside of the scope of this paper.

**Semi-structured document retrieval based on keywords** relies mostly on the *Least Common Ancestors* approach, by which a set of XML nodes containing the requested keywords are resolved into one result item, their common ancestor node [6, 23]. This field pioneered by [11], encompassed by our model, generalizes LCA constraints but lacks both social and semantics, and thus meets only **R2**. Other recent developments in this area, including more flexible and comprehensive reasoning patterns, have been presented in [2] but have the same limitations. IR-style search in relational databases [12, 13] considers key-foreign key relationships between items, but ignores text structure, semantics, and social aspects.

**Semantic search on full-text documents**, either via RDF [15, 25] or a semantic similarity measure [24], allows to query interconnected, semantic rich unstructured textual documents or entities, thus meeting **R1**, **R5** and **R6**. Efforts to consider XML structure in such semantics-rich models [9] also enable **R2**.

**Personalized IR in a social context** adapts the answers to a user's query, taking into account her interests and those of her direct and indirect social connections [5]; this meets **R1** but not **R2** nor **R3**.

All the aforementioned models can be seen as partial views over the S3 model we devised, and they could easily be transcribed into it modulo some minor variations; for instance, Facebook's Graph-Search [7] is a restricted form of SPARQL query one could ask over an S3 instance. Slight adaptations may be needed for social contexts tolerating *similarity* between keywords that goes beyond the strict specialization relation (in RDF sense) we consider. We have hinted in Section 2 how this could be included.

## 7. CONCLUSION

We devised the S3 data model for structured, semantic-rich content exchanged in social applications. We also provided the $\mathtt{S3}_k$ top-k keyword search algorithm, which takes into account the social, structural and semantical aspects of S3. Finally, we demonstrated the practical interest of our approach through experiments on three real social networks.

Next, we plan to extend $\mathtt{S3}_k$ to a massively parallel in-memory computing model to make it scale further. We also consider generating user-centric knowledge bases to be used in $\mathtt{S3}_k$, to further adapt results to the user's semantic perspective.

## References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] C. Aksoy, A. Dimitriou, and D. Theodoratos. Reasoning with Patterns to effectively answer XML keyword queries.*The VLDB Journal*, 2015.

[3] R. Bonaque, B. Cautis, F. Goasdoué, and I. Manolescu. Social, structured and semantic search, extended version. https://hal.inria.fr/hal-01218116, 2015.

[4] P. Buneman, E. V. Kostylev, and S. Vansummeren. Annotations are relative. In *ICDT*. ACM, 2013.

[5] D. Carmel, N. Zwerdling, I. Guy, S. Ofek-Koifman, N. Har'El, I. Ronen, E. Uziel, S. Yogev, and S. Chernov. Personalized social search based on the user's social network. In *CIKM*, 2009.

[6] L. J. Chen and Y. Papakonstantinou. Supporting top-k keyword search in XML databases. In *ICDE*, 2010.

[7] M. Curtiss, I. Becker, T. Bosman, S. Doroshenko, L. Grijincu, T. Jackson, S. Kunnatur, S. Lassen, P. Pronin, S. Sankar, et al. Unicorn: A system for searching the social graph. *PVLDB*, 2013.

[8] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4), 2003.

[9] F. Goasdoué, K. Karanasos, Y. Katsis, J. Leblay, I. Manolescu, and S. Zampetakis. Growing triples on trees: an XML-RDF hybrid model for annotated documents. *VLDB Journal*, 2013.

[10] F. Goasdoué, I. Manolescu, and A. Roatiş. Efficient query answering against dynamic RDF databases. In *EDBT*, 2013.

[11] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over XML documents. In *SIGMOD*, 2003.

[12] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-style keyword search over relational databases. In *VLDB*, 2003.

[13] V. Hristidis, H. Hwang, and Y. Papakonstantinou. Authority-based keyword search in databases. *ACM TODS*, 33(1), 2008.

[14] G. Jeh and J. Widom. Simrank: a measure of structural-context similarity. In *SIGKDD*, 2002.

[15] W. Le, F. Li, A. Kementsietsidis, and S. Duan. Scalable keyword search on large RDF data. *IEEE TKDE*, 26(11), 2014.

[16] Y. Li, Z. Bao, G. Li, and K.-L. Tan. Real time personalized search on social networks. In *ICDE*, 2015.

[17] D. Liben-Nowell and J. Kleinberg. The link-prediction problem for social networks. *JASIST)*, 2007.

[18] S. Maniu and B. Cautis. Network-aware search in social tagging applications: instance optimality versus efficiency. In *CIKM*, 2013.

[19] P. E. O'Neil, E. J. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. Ordpaths: Insert-friendly XML node labels. In *SIGMOD*, 2004.

[20] A. Ritter, S. Clark, Mausam, and O. Etzioni. Named entity recognition in tweets: An experimental study. In *EMNLP*, 2011.

[21] R. Schenkel, T. Crecelius, M. Kacimi, S. Michel, T. Neumann, J. X. Parreira, and G. Weikum. Efficient top-k querying over social-tagging networks. In *SIGIR*, 2008.

[22] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD*, 2002.

[23] M. Theobald, H. Bast, D. Majumdar, R. Schenkel, and G. Weikum. TopX: efficient and versatile top-k query processing for semistructured data. *The VLDB Journal*, 2008.

[24] M. Theobald, R. Schenkel, and G. Weikum. Efficient and self-tuning incremental query expansion for top-k query processing. In *SIGIR*, 2005.

[25] 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.

[26] J. Urbani, S. Kotoulas, J. Maassen, F. van Harmelen, and H. E. Bal. WebPIE: A web-scale parallel inference engine using MapReduce. *J. Web Sem.*, 2012.

[27] Resource Description Framework. http://www.w3.org/RDF.

[28] Uniform Resource Identifier. http://tools.ietf.org/html/rfc3986.

[29] Yelp Dataset Challenge. http://www.yelp.com/dataset_challenge.

[30] S. A. Yahia, M. Benedikt, L. V. Lakshmanan, and J. Stoyanovich. Efficient network aware search in collaborative tagging sites. *PVLDB*, 2008.