Top-k search over rich web content

Raphaël Bonaque

NNT : 2016SACLS291

THÈSE DE DOCTORAT
DE
L'UNIVERSITÉ PARIS-SACLAY
PRÉPARÉE À
L'UNIVERSITÉ PARIS SUD

ÉCOLE DOCTORALE N°580
Sciences et technologies de l'information et de la communication

Spécialité de doctorat : Informatique

Par

M. Raphaël Bonaque

Recherche top-k pour le contenu du Web

Thèse présentée et soutenue à Saclay, le 30 septembre 2016 :

Composition du Jury :

M. Colazzo Dario	Professeur, Université Paris-Dauphine	Président du Jury
M. Amann Bernd	Professeur, Université de Pierre et Marie Curie	Rapporteur
Mme Hose Katja	Associate Professor, Aalborg University	Rapporteur
Mme Cohen Boulakia Sarah	Maître de Conférences HDR, Université Paris-Sud	Examinatrice
M. du Mouza Cédric	Maître de Conférences HDR, CNAM	Examinateur
Mme Manolescu Ioana	Directeur de recherche, Inria	Directrice de thèse
M. Cautis Bogdan	Professeur, Université Paris-Sud	Co-directeur de thèse
M. Goasdoué François	Professeur, Université de Rennes 1	Co-directeur de thèse

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Recherche top-k pour le contenu du Web

Mots clés : Web sémantique, réseaux sociaux, documents structurés, top-k, RDF, XML, JSON

Résumé : Les réseaux sociaux sont de plus en plus présents dans notre vie de tous les jours et sont en passe de devenir notre moyen de communication et d'information principal. Avec l'augmentation des données qu'ils contiennent sur nous et notre environnement, il devient décisif d'être en mesure d'accéder et d'analyser ces données. Aujourd'hui la manière la plus commune d'accéder à ces données est d'utiliser la recherche par mots-clés : on tape une requête de quelques mots et le réseau social renvoie un nombre fixe de documents qu'il juge pertinents. Dans les approches actuelles de recherche top-k dans un contexte social, la pertinence d'un documents dépend de deux facteurs : la proximité sociale entre le document et l'utilisateur faisant la requête et le recoupement entre les mots-clés de la requête et les mots contenus dans le document. Nous trouvons cela limité et proposons de prendre en compte les interactions complexes entres les utilisateurs liés à ce document mais aussi sa structure et le sens des mots qu'il contient, au lieu de leur formulation. Dans ce but, nous identifions les exigences propres à la création d'un modèle qui intégrerait pleinement des données sémantiques, structurées et sociales et proposons un nouveau modèle, S3, satisfaisant ces exigences. Nous rajoutons un modèle de requêtes à S3 et développons S3k, un algorithme personnalisable de recherche top-k par mots-clés sur S3. Nous prouvons la correction de notre algorithme et en proposons une implémentation. Nous la comparons, à l'aide de jeux de données créés à partir du monde réel, avec celle d'une autre approche de recherche top-k par mots-clés dans un contexte social et montrons les différences fondamentales entre ces approches ainsi que les avantages qu'on peut tirer de la nôtre.

Title : Recherche top-k pour le contenu du Web

Keywords : semantic Web, social network, structured documents, top-k, RDF, XML, JSON

Abstract : Social networks are increasingly present in our everyday life and are fast becoming our primary means of information and communication. As they contain more and more data about our surrounding and ourselves, it becomes vital to access and analyze this data. Currently, the primary means to query this data is through top-k keyword search: you enter a few words and the social network service sends you back a fixed number of relevant documents. In current top-k searches in a social context the relevance of a document is evaluated based on two factors: the overlapping of the query keywords with the words of the document and the social proximity between the document and the user making the query. We argue that this is limited and propose to take into account the complex interactions between the users linked to the document, its structure and the meaning of the words it contains instead of their phrasing. To this end we highlight the requirements for a model integrating fully structured, semantic and social data and propose a new model, called S3, satisfying these requirements. We introduce querying capabilities to S3 and develop an algorithm, S3k, for customizable top-k keyword search on S3. We prove the correctness of our algorithm and propose an implementation for it. We compare this implementation with another top-k keyword search in a social context, using datasets created from real world data, and show their differences and the benefits of our approach.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France

Résumé en Français

French summary

Introduction

Dans sa proposition initiale pour le World Wide Web, Tim Berners-Lee imagine un énorme graphe de nœuds dispersés sur un réseau mais interconnectés et accessibles grâce au protocole hypertexte, un protocole qui a évolué vers l'HTTP qu'on connait. Ces nœud ne sont pas que des documents ; pages web ou ressources mais aussi des concepts, des objets physiques, et des personnes. On pourrait voir dans ces nœuds, qui n'existent pas dans le protocole HTTP actuel, comme les limites de la clairvoyance d'un homme qui, en inventant le Web et en prédisant qu'il s'étendrait pour devenir un système d'information universel à déjà fait beaucoup.

L'histoire pourrait s'arrêter là mais il semble de plus en plus probable qu'elle continue : depuis 2004, Tim Berners-Lee travaille sur le Web Sémantique, une expression qu'il a créée et qui désigne l'intégration dans le Web d'un des nœuds qu'il liste dans sa proposition initiale: les concepts. Avec des initiatives comme Linked Open Data, la taille du Web Sémantique est en train d'exploser, sans doute même plus vite que celle du Web en son temps. Depuis 2007, LOD en est venu à intégrer plus de mille bases de données, qui contiennent entre elles des dizaines de milliards de faits et des millions de concepts.

Par ailleurs, l'Internet of Things, l'idée selon laquelle les objets physiques de la proposition de Tim Berners-Lee se connectent au Web, est en plein essor, avec plus de vingt quatre mille articles publiés en 2015 sur le sujet.

A voir les progrès de l'intégration des objet physiques et des concepts sur le Web il semble intéressant de se pencher sur l'évolution de l'intégration du dernier type de nœud manquant : les personnes. Aucun standard ne s'est vraiment répandu pour décrire les personnes comme des ressources du Web au même titre que des pages web mais il existe néanmoins des efforts comme OpenID qui est accepté par de nombreux services même si il ne permet pas d'établir des connections entre utilisateurs. D'un autre coté les réseaux sociaux ont, depuis leurs créations, acquis une popularité très importante : on estime qu'un adulte passe en moyenne douze heures par semaine sur les réseaux sociaux, avec l'important impact économique et sociétale que cela entraine. Par example, les gros sites de réseaux sociaux ont des revenus proches du PIB de petit pays. Les chiffres sont marquants, avec 82% des 16-64 ayant accès à internet qui déclarent avoir un compte Facebook. Alors même que Facebook se targue de posséder aussi WhatsApp avec ses un milliard d'utilisateurs, ainsi que les 400 millions d'utilisateurs d'Instagram. Tencent's QQ dit avoir 850 millions d'utilisateurs actifs, Wechat 700 milions, Twitter 310 et LinkedIn 106.

Tous ces sites avec plus d'utilisateurs que la plupart des pays du G8 ont moins de 15 ans. Lorsqu'on compare aux 25 ans qu'a environ le Web il ne semble pas irraisonnable que les personnes puissent accéder dans un future proche au même statut que les pages web. De plus plus de services, comme youtube ou Tumblr qui n'avaient pas au début vocation à être sociaux de sont mis à intégrer des fonctionnalités sociales et les réseaux sociaux sont devenus le principal moyen de communication d'une part importante de la population et de nombreuses personnalités publiques, artistes, hommes politiques, etc ...

Les réseaux sociaux s'avèrent être de très bons sujets de recherche, et pas seulement dans le domaine de l'informatique, mais aussi en sociologie, ou dans la détection des catastrophes naturelles. Dans le cas des tremblements de terre par example, les nouvelles se propagent bien plus vite par les réseaux sociaux que par les canaux officiels.

Le Web est entrain de devenir dépositaire de l'ensemble du savoir de l'humanité, et il est presque certain qu'il sera social.

Démarche

Les réseaux sociaux actuels sont plus qu'un ensemble de relations entre des gens, ils sont construit autour d'un contenu. C'est particulièrement visible sur des sites qui se sont développés à partir de plateformes de contenu comme des blogs, ou des sites d'images ou de vidéos mais c'est aussi le cas pour les réseaux sociaux plus traditionnels comme Facebook ou Twitter où la majorité des données prennent la forme de documents (appelés post ou tweet pour ces sites).

Lorsqu'ils utilisent des réseaux sociaux, les utilisateurs s'attendent à trouver des contenus liés à leurs relations présentes sur le site, par example des nouvelles de leurs amis, des informations sur des associations qui les intéressent ou sur leurs institutions locales. Ce contenu peut être fournis de manière semi automatique par une page d'accueil, sur la page d'un utilisateur ou d'un groupe mais la grande majorité de ces services (et tous les plus gros) proposent une fonctionnalité de recherche par mots clés. Cette fonctionnalité prend souvent la forme d'une barre de recherche où l'utilisateur peut taper des mots clés et, lorsqu'ils sont soumis, un nombre déterminé de résultats sont proposés à l'utilisateur. Ce procédé, appelé recherche top-k, est le principal moyen d'accès pour les utilisateurs aux données des réseaux sociaux ; Facebook par example affirme traiter plus de 1,5 milliards de recherches de ce genre par jour. Il est difficile de réaliser à quel point cette recherche est devenue importante dans notre vie de tout les jours car de nombreux moteurs de recherche sociaux peuvent paraitre ne pas traiter cette dimension sociale. Par example depuis l'intégration de Google+ dans le moteur de recherche de Google et de Youtube la recherche sur ces deux sites omniprésents (les deux plus utilisés du monde) est devenue sociale et utilise des connaissances extraites de Google+

pour adapter ses résultats. Cela influe tout les jours sur plus de la moitié de la population mondiale connectée à Internet ailleurs qu'en Chine .

La fonctionnalité de recherche proposée par ces sites, dans la mesure où on arrive à l'observer[1], ou telle qu'elle est décrite dans la littérature scientifique, s'appuie essentiellement sur deux facteurs. Tout d'abord la présence dans les documents des mots clés de la requête et de petites variations autour de ceux-ci (corrections orthographiques, forme plurielle ou singulière, féminine ou masculine, transformation d'un adjectif en nom ou l'inverse ...). Ensuite, le voisinage social du demandeur (l'utilisateur qui pose la requête), c'est à dire ses relations directes, les relations directes de celles-ci ou les personnes populaires du réseau social en question. Les résultats d'une requête contiennent donc des mots très proches de la formulation de celle-ci et leurs auteurs doivent être directement reliés à une connaissance du demandeur ou être une personnalité connue.

Au vu de l'importance de la recherche par mots clés et de son omniprésence dans les contextes sociaux, cette approche parait limitée, en particulier elle limite les résultats possibles car elle ne considère pas le sens de la requête mais sa formulation. Cela peut marcher dans certaines niches mais est insuffisant face à la complexité et la richesse des langues naturelles. Prenons un example très simple, une recherche sur "chasse espèces menacées". Des documents contenant "la traque d'un lion Asiatique", "le braconnage d'Ara hyacinthes" ou "la prise de raies requins" sont pertinent pour cette requête et ne seront pourtant pas retenus par les moteurs de recherches sociaux actuels à moins qu'ils ne contiennent aussi explicitement des références à "chasse" et à "espèces menacées". Cela aboutit à de nombreux utilisateurs n'obtenant que peu de résultats à leur recherche et devant la reformuler, souvent pour de maigres résultats. Dans ce cas précis il existe de nombreux mots pour désigner des actions ou des évènements impliquant une chasse et plus encore pour les espèces menacées, aboutissant à un nombre de reformulations possibles colossal.

Les fournisseurs de services de recherche par mots clés connaissent ces limitations et prennent lentement des mesures pour essayer de les dépasser. L'un des meilleurs exemple est sans doute la recherche par mots clés de Google en anglais, le moteur de recherche top-k le plus utilisé du monde. Depuis 2009 Google a commencé à intégrer des aspects sémantiques dans son moteur sous la forme de *special features* qui s'activent lorsque certains termes sont présents dans une requête. Ainsi la présence du mot *etymology* suivit d'un autre mot va afficher chez l'utilisateur un petit cadre décrivant l'étymologie du mot en question. Il existe ainsi quelques dizaines de *special features*, pour la météo, l'heure, des calculs simples, des traductions ... toutes retournant des informations sur le sens de la requête et non sa formulation. C'est un progrès mais cela ne fonctionne que pour un petit nombre de cas préprogrammés et ne retourne pas des documents pertinents pour

[1] Il est souvent impossible d'accéder au code des services web privés.

la requêtes de l'utilisateur, à la place cela construit un unique résultat limité à un environnement bien contrôlé mais n'identifie pas de document sur le Web. Google étend aussi les requêtes des utilisateurs pour rajouter des termes qui sont soit des synonymes soit des mots fortement corrélés par leur cooccurrence aux mots clés originaux. Cela aussi est très limité : "lion Asiatique" n'est pas un synonyme d'"espèces menacées" et leur corrélation est sans doute très faible vu le nombre d'espèces menacées.

Une autre limitation des approches top-k actuelles est la prise en compte de la structure des documents. Considérons un requête avec deux mots clés, k_1 et k_2 et les deux réponses possibles suivantes:

- un texte t_1 de plusieurs paragraphes, où k_1 n'est présent que dans un paragraphe et k_2 est présent dans certains des paragraphes ou k_1 n'est pas.
- un texte t_2 ayant la même longueur, le même nombre de paragraphes et le même nombre d'occurrences de k_1 et de k_2 mais cette fois-ci toujours dans le même paragraphe, mettons le deuxième du texte.

D'une part il existe un paragraphe dans t_2 qui contient les deux mots clés simultanément ce qui suggère que le sujet de ce paragraphe les relie. D'autre part il n'existe pas ce genre de relations entre k_1 et k_2 dans t_1, les deux mots clés y sont présents mais peuvent n'être que vaguement liés. Dans ce contexte il est probable que t_2 soit plus pertinent que t_1 mais en poussant l'argument on pourrait même dire que le deuxième paragraphe de t_2 est en fait le meilleur candidat car il contient uniquement la partie du texte la plus pertinente pour la requête.

Cette subtilité ne peut être perçue par les moteurs de recherche qui ignorent la structure des documents, par exemple ceux qui fonctionnent sur le modèle du *bag of words* dans lequel on considère uniquement le multi-ensemble des mots contenus dans chaque document. Il existe bien des approches top-k qui tiennent compte de la structure des documents mais jusqu'à présent jamais dans un contexte social.

La recherche par mots clés, en tant que moyen principal d'accéder aux contenus sociaux, est omniprésente, il est donc crucial de la rendre la plus efficace possible et d'exploiter tous les aspects disponibles. La portée de cette problématique est extrêmement vaste, dans cette thèse nous nous intéresserons uniquement à des requêtes top-k exploitant au mieux la sémantique de la requête et des documents, leur structures et les connections sociales entre utilisateurs.

Contributions

Dans le cadre de cette thèse nous présentons tout d'abord, les notions élémentaires de recherche top-k, de graphes de connaissance, de documents structurés et semi-structurés et de réseaux sociaux. Nous parcourons ensuite l'état de l'art de la

recherche top-k dans des contextes sociaux, structurels, sémantiques et, lorsqu'ils existent, sur des modèles incluant plusieurs de ces dimensions.

Nous introduisons ensuite un ensemble de contraintes nécessaires à l'intégration de données à la fois sémantiques, structurelles et sociales dans un modèle unifié. Nous présentons un nouveau modèle, `S3` qui intègre ces contraintes et montrons comment des instances d'autre modèles de données ayant une composante sociale, structurelle ou sémantique peuvent être plongés dans `S3`.

Nous développons un moteur, $\texttt{S3}_k$ pour la recherche top-k sur `S3`, basé sur un score attribué aux documents pour estimer leur pertinence. Nous avons voulu ce score le plus générique possible pour s'adapter à tous les cas possibles en lui demandant uniquement de respecter un petit ensemble de propriétés mathématiques pour assurer un bon niveau de performances à notre moteur. En plus de la définition de notre moteur, nous proposons un algorithme réalisant $\texttt{S3}_k$. Nous fournissons aussi la preuve de la correction et la terminaison de cet algorithme.

Une implémentation concrète de cet algorithme est proposée et est évaluée sur des données réelles puis comparée avec un autre système de recherche top-k dans un contexte social.

Enfin, nous concluons cette thèse et introduisons des perspectives pour l'utilisation de `S3` dans le domaine du journalisme de données.

Acknowledgements

I guess every PhD is an adventure, with some problems to slay and some papers to rescue. This one was interesting, with a lots of old grimoires to read, some mystical animals like the Python 27 and the DB Elephant, a good amount of friendly encounters and travels to foreign places. Yet, I do fear that the tale before you might be a little boring to read, and could contain some mistakes. This is my fault alone for my advisors Ioana, Bogdan and François always tried to steer the ship in the right direction, and sometime they had quite some wind to go against. I'm thankful to them, to Ioana who was always there and never afraid to speak her mind, to Bogdan who, the first, offered me the opportunity to do this PhD and introduced me to social networks and top-k, and to François who was always available and helpful, no matter the distance. I was glad I could do this PhD with you and I hope you had some good moments too.

I thank the members of my jury and especially the reviewers, who spent a lot of time in the midst of the summer holidays to read my manuscript, for accepting to be part of my jury.

My companion, Nolwenn, and my family were a huge support during this PhD and I certainly won't have done it without them, they deserve many thanks.

I would like to extend my thanks to my circle of friends and especially RxPz's: Trolin, ZenithM, CPA, Masda, Levity, Milou (and Guillaume) and other friends from Cachan: Poussinet, Harry, PEB, Ping and Jonas for providing me support and fun moments.

I award a prize to all my flatmates: Ariane, Emeline, Emile, Fabrice, Lucas, Marion, Martin and Nicolas that have accepted to live with me for some time, and I tell you right away that I won't be taking the monkey painting, no need to try and sneak it in.

Some of my teachers, Mr Belazreg, Ms. Benhamou and M. Tosel, were very influential in my understanding of the world and they have all my gratitude.

Finally, I also thank the many people that were part of a team with me, even if for a short time: Alessandro, Alexandra, Benjamin, Benoit, Camille, Celine, Chantal, Damian, Danai, Elham, Emmanuel, Enhui, Fatiha, Francesca, Guillaume, Hassan, Ioana, Javier, Jesús, Juan, Katerina, Martin, Meghyn, Michaël, Nathalie,

Nicole, Oscar, Paul, Philippe, Rana, Rémi, Romain, Šejla, Silviu, Soudip, Stamatis, Swen, Tianqi, Tien-Duc, Yanlei, Yifan, Yue and Zheng. It was a great pleasure to meet and work with you.

Contents

Chapter 1

Introduction

1.1 The rise of social networks

In his original proposal of the World Wide Web [94], Tim Berners-Lee envisioned a large graph of interconnected nodes, scattered across a network and accessible through the hypertext protocol, a protocol that will evolve to become HTTP. These nodes are not limited to documents such as web pages and resources, but also include concepts, specific hardware objects and, first to be mentioned in the proposal, people. One could see this as the limit of the vision that one man can have: inventing the Web and predicting that it would go "toward[s] a universal linked information system" is already a lot to be credited for, and the story might stop here.

However, it now appears more likely that the story continues to unfold. Since 2004, Tim Berners-Lee has been working on the Semantic Web - an expression that he coined -, which aims at integrating into the Web the "concepts" he listed in his proposal. With initiatives like Linked Open Data, the size of the Semantic Web is exploding, possibly even faster than the Web did in its time: since its inception in 2007, LOD has grown to include more than 1,000 datasets, containing collectively dozens of billions of facts over millions of concepts.[1]

In parallel, the Internet of Things, i.e., the idea according to which the "specific hardware objects" from Tim Berners-Lee's proposals become connected to the Web, is booming, with more than 24,000 published articles containing "internet of things"

[1]Sources for this statement are, in this order: https://en.wikipedia.org/wiki/Linked_data and http://linkeddatacatalog.dws.informatik.uni-mannheim.de/state/, https://scholar.google.com/scholar?q=%22internet+of+things%22&btnG=&hl=en&as_sdt=1%2C39&as_ylo=2015&as_yhi=2015&as_vis=1, http://www.globalwebindex.net/blog/gwi-social-facebook and https://blog.whatsapp.com/616/One-billion and http://www.statista.com/statistics/272014/global-social-networks-ranked-by-number-of-users/ and http://www.businessinsider.com/wechat-pay-fees-kick-in-2016-5?IR=T and https://about.twitter.com/company and https://press.linkedin.com/site-resources/news-releases/2016/linkedin-announces-first-quarter-2016-results, http://www.globalwebindex.net/blog/daily-time-spent-on-social-networks-rises-to-1-72-hours

in 2015 alone.[1]

Seeing how concepts and specific hardware objects are increasingly present on the Web, it is interesting to look at the evolution of the integration of people on the Web. No real standard has emerged to integrate people as basic web resources at the same level as documents and concepts: there are some standardization efforts such as OpenID[2] that is accepted by many services, but they are not currently universally accepted, and lack connections between users. On the other hand, social networking web sites have, since their inception, gained a huge popularity. An average adult would spend more than 12 hours per week on social networks[1],which have a huge impact on the global economy: a big social services company may have a revenue equivalent to the GDP of a small country. The figures are striking: 82% of the 16-64 years old having access to internet outside China claim to have at least one Facebook account, but Facebook also happens to own WhatsApp, which claims more than 1 billion users and Instagram and its 400 million users. Tencent's QQ claims 850 million active users, Wechat claims 700 millions, Twitter 310 millions and LinkedIn 106 millions[1].

All these websites with more users than the population of most G8 countries are less than 15 years old. Compared to the approximative 25 years of the Web it is not a great leap of faith to say that people will achieve the same status as other web resources in the near future: many web services that did not used to be social are now integrating social services, such as for instance Youtube or Tumblr currently. It can be confidently said that social networks are already so common and widely used, that they have become the main means of communication for a large part of the population and for many public figures such as artists or politicians and entities, whether NGOs, governmental agencies or private companies.

Social networks also turn out to be great research fields, even outside computer sciences, for instance in sociology but also more diverse fields like natural disaster detection, e.g., earthquakes [67], where the information spreads much faster through social networks than via the official channels.

The Web is emerging as the repository of all the human knowledge, and with very little doubt it will be social.

1.2 Motivation

Current social networks are more than just a series of relationships between people: they are built around content. This is very visible for websites that grew out of content-centric platforms such as blogs or image or video providers, but it is also the case for more traditional social network services such as Facebook or Twitter where the majority of the data takes the form of documents (called posts

[2] `https://openid.net/`

or tweets for these sites).

When using social network services, users expect to find content related to their acquaintances on the service: news or information from their friends, NGOs they are interested in, their local institutions, etc. This content can be provided semi-automatically on a home page or on the page of a user or a group of users, but the overwhelming majority of these services, and all the major ones, provide a keyword-based search functionality. This functionality often takes the form of a search bar: a text zone where the user can input keywords, and, when these keywords are submitted, a fixed number of results are returned. This process, known as top-k keyword search, is the primary means by which users access data on social network services: for instance Facebook claims to have more than 1.5 billions of such searches per day. It is difficult to realize how important this search functionality has become in our daily life because numerous social search engines might not appear to be social: for instance, since the integration of Google+ in the Google search engine and in Youtube, the search on these two omnipresent services (the two most used websites in the world[3]) has become social and uses knowledge derived from the social network of Google+ to customize its results. This impacts on a daily basis more than half the world population outside China having an internet connection.

The search functionality provided by these services to the extent that we can observe them[4], or as they are described in the scientific literature, relies heavily on two factors: (*i*) the presence in documents of the keywords of the query and their simple variations (orthographic correction, plural from a singular or adjective from a noun and vice versa), (*ii*) the social neighborhood of the seeker (the user making the query), that is to say who are his or her direct relations, the direct relations of his or her direct relations and the well-known people of the social network. The results of a query therefore contain words that adhere strictly to the phrasing of the query and they must originate in or be directly connected to someone the seeker might know, either in his or her direct neighborhood or amongst public figures.

Considering how important and pervasive keyword search in a social context is, this functioning appears limited: in particular, it restricts the results because it does not consider the meaning of the query but only the words it contains. This works well on very specialised or formatted terms but falls short when confronted with the complexity and richness of the real world language. Consider this very simple example: a keyword search for "hunt endangered species". Documents containing "track of an Asiatic lion", "hyacinth macaw's poachings" or "the catching of blackchin guitarfish" should be relevant for this query, but will not be returned by current social search engines unless they also contain minor variations around "endangered species" and "hunt". This results in many users obtaining too few results

[3]source: `http://www.alexa.com`

[4]It is often impossible to have access to the code behind private web services.

and having to reformulate their queries, often several times, for limited outcomes. In this particular case there exist many words for actions and events that imply a hunt and even more for words designating endangered species, leading to a very large number of possible reformulations.

Keyword search service providers are aware of this shortcoming, and are slowly taking steps to try and fix it. The best example of this is probably Google search in English[5], the most used top-k keyword search engine in the world. Since 2009, Google has started integrating semantics into its search engine in the form of special features that are activated by the presence of a set of words or abbreviations in the query. For instance the query "etymology" followed by another word will return a card: a small rectangular area shown before the other results on the screen of the client displaying text, images or diagrams, in this cases it shows a short text explaining the etymology of the second word. Another example is concatenating relations such as "date of death", "spouse of" or "place of birth" to the name of famous people, which will return a card with the names of the associated persons, and possibly a place, a date or a picture. There exist a few dozens of such handcrafted features: for the weather, time, simple computations, translations, etc., all returning a card with information relevant to the semantics of the query, and then documents relevant to the phrasing of the query. This is a step forward but it only works for a small set of pre-programmed cases and it does not actually return documents relevant to the semantics of the query. Rather, it constructs a result in a well-controlled environment, but does not identify the relevant documents on the Web. Google search also does query expansion, adding to the query terms that are either synonyms or words statically correlated with the query's keywords, yet this is also very limited: " Asiatic lion" is not a synonym of "endangered species", and their correlation is likely very low, given the number of endangered species.

Another shortcoming of current top-k approaches in a social context is the inclusion of structure. Consider a query with two keywords, k_1 and k_2, and the following candidates for the top-k:

- a text t_1 of several paragraphs, where k_1 occurs only in one paragraph, and k_2 also has several occurrences, always in another paragraph,
- a text t_2 of equal length and number of paragraphs, where k_1 and k_2 occur as many times as in t_1, but always in the second paragraph.

On the one hand, there exists a paragraph in t_2 containing both keywords simultaneously, which suggests that the topic of this paragraph relates them to one another. On the other hand, there is no such relation in t_1: k_1 and k_2 are loosely

[5]Other social and non social search engines are following the same tracks, but it is simpler to see this evolution on the Google search engine as it is well documented.

linked to the topic of the text but they could pertain to very different aspects of this topic and they are probably unrelated. In this context it is easy to argue that t_2 is more relevant than t_1, but if we extend the argument we could say that the second paragraph of t_2 is in fact an even better candidate as it contains only the relevant part of t_2 for this query.

This subtlety is completely lost to a search engine ignoring the structure of documents, for instance one operating on the so-called "bags of words" model, which considers only the multiset of the words contained in the text. There exist top-k approaches utilizing the structural aspect of the data they consider but, until now, these kind of approaches have never been embedded into social search engines.

Last but not least, the metadata of the query and the documents, most specifically their timestamps and localisations, are useful for addressing queries related to news, such as "strikes today", or to local information, for instance "nearby restaurants".

As keyword search is the primary means to access data in a social context, and this data is everywhere in our society, it crucially needs to make full use of all the available information: from the query, exploiting its semantics, from the documents, using their content, structure and metadata, and from the social network, with all the connections it contains. The scope of this problem is very large and in this thesis we will ignore the metadata and focus on top-k keyword search of documents in a social context, using fully the semantic, structured and social aspects available. To this end, our goals are:

- to give a complete, formal, description of the problem and to create a unified data model encompassing, without loss, the three dimensions: semantics, structure and social;
- to establish a formal framework for querying this data model and to implement it,
- to demonstrate that our query framework does indeed improve on the previous works, by taking into account these three dimensions had a positive impact on the search results.

1.3 Contributions and outline

Below, we present the contributions of this thesis and its general organization:

Chapter 2 recalls basic notions for the standard representation and querying of knowledge graphs, of structured and semi-structured documents and of social data.

Chapter 3 surveys the literature to present the state of the art in terms of top-k search in a social, semantic or structured context, as well as hybrid approaches trying to query models integrating several of these dimensions.

Chapter 4 identifies a set of requirements for a model integrating comprehensively the social, semantic and structural aspects of the description of users, documents and their interactions with each other. It also provides a model, S3 that meets these requirements. The main contributions of this chapter are:

- the S3 model and its diverse components;
- the explanation, with examples of how the instances of models presented in the previous chapter can be mapped into S3.

Chapter 5 deals with querying S3 in a new, top-k keyword-based framework that we introduce: $\texttt{S3}_k$. The main contributions of this chapter are:

- the querying framework $\texttt{S3}_k$;
- the notion of generic and concrete scores for this query model, and the mathematical properties attached to them;
- the algorithm we developed to implement this query model;
- the proof of termination and correctness of said algorithm.

Chapter 6 presents the technical details of implementation and the evaluation of our algorithm $\texttt{S3}_k$ compared to another system for top-k keyword query in a social context. The main contributions of this chapter are:

- a description of our implementation of $\texttt{S3}_k$;
- the analysis of the performances of our implementation;
- the qualitative and quantitative comparisons with another system for top-k keyword search in a social context.

Chapter 7 concludes this thesis and presents perspectives for future use of S3. The main contribution of this chapter is the introduction of TATOOINE, a lightweight integration architecture for data journalism.

Chapter 2

Preliminaries

This chapter introduces the de-facto standard formalisms to model semantic, structured and social data.

In Section 2.1 we present the Resource Description Framework, a universal model on the Web to describe data and knowledge. Then, in Section 2.2, we recall the two most accepted formats for semi-structured data, XML and JSON, and their associated query languages. Finally, in Section 2.3 we survey models for popular social networks.

2.1 The RDF model

URIs and literals We assume given a set U of Uniform Resource Identifiers (URIs, in short), as defined by the standard [92], 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".

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 2.1), `s p o` corresponds to the tuple $(\mathtt{s}, \mathtt{o})$ in the binary relation `p`, e.g., u_1 hasFriend u_0 corresponds to hasFriend(u_1,u_0). We consider every triple is *well-formed* [84]: 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.

Constructor	Triple	Relational notation
Class assertion	$\mathtt{s}$ type $\mathtt{o}$	$\mathtt{o}(\mathtt{s})$
Property assertion	$\mathtt{s}$ $\mathtt{p}$ $\mathtt{o}$	$\mathtt{p}(\mathtt{s}, \mathtt{o})$

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

Figure 2.1: RDF (top) and RDFS (bottom) statements.

RDF types and schema The built in type property provided by 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* that person;
- 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 2.1 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 [7], i.e., as *deductive constraints*. For instance, if a graph includes the triples hasFriend $\hookleftarrow_{\mathrm{d}}$ Person and u_1 hasFriend u_0, then the triple 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 2.1.

Figure 2.2 shows an example of RDF graph, describing some facts about the Secretary-General of the United Nations, Ban Ki-moon, and some universities.

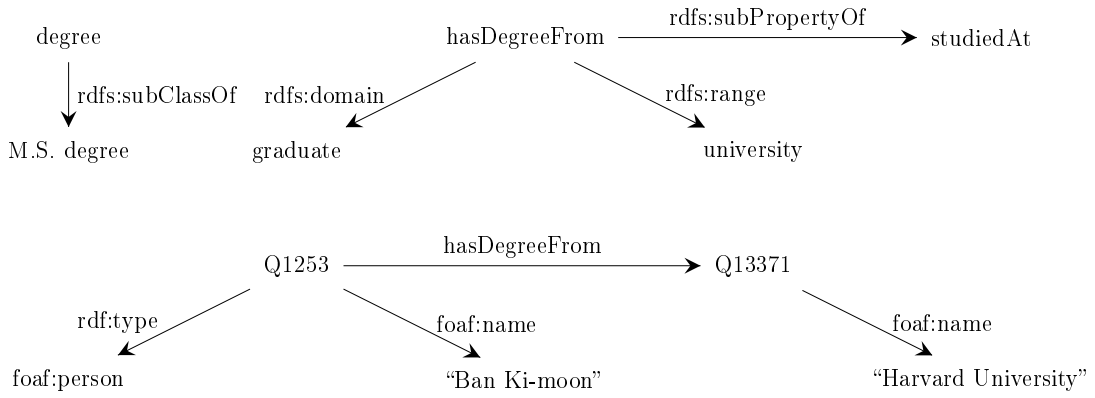

Figure 2.2: Sample RDF graph.

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 $\vdash^{i}_{\text{RDF}}$) 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,

$$\text{u}_1 \text{ hasFriend } \text{u}_0,\ \text{hasFriend } \hookrightarrow_{\text{r}} \text{ Person } \vdash^{i}_{\text{RDF}} \text{u}_0 \text{ type Person}$$

Applying immediate entailment $\vdash^{i}_{\text{RDF}}$ 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* [84].

In the following, we assume, without loss of generality, that all RDF graphs are saturated; many saturation algorithms are known, including incremental [34] or massively parallel ones [78].

SPARQL is a recursive acronym for SPARQL Protocol and RDF Query Language, defined by the W3C [80]; in particular, SPARQL provides a query language for RDF. The basic constructs of this language is the *graph pattern*: a set of triples in which every subject, property, or object position can also be a variable. An *embedding* of such a graph pattern into an RDF graph is an assignment of URIs and literals to the query variables such that replacing the variables with these URIs/literals in the graph pattern leads to a subgraph of the RDF graph.

Using selection, aggregation and construction operators it is then possible to create and return either tuples of bindings or RDF graphs. For instance the query in Figure 2.3 returns tuples of one element (*person_name*, *univ_name*) such that *person_name* is the name of a person who studied at the university *univ_name*.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?person_name ?univ_name
WHERE {
   ?x foaf:name ?name ;
      type foaf:person .
   ?x studiedAt ?y .
   ?z rdf:type university ;
      foaf:name ?univ_name .
}
```

Figure 2.3: Sample SPARQL query.

2.2 Semi-structured document models

Extensible Markup Language, or *XML* for short, is a format of semi-structured documents defined by the W3C [85] and omnipresent on the Web. An XML document represents an ordered, unbounded, finite tree.

XML tree nodes can be either elements, attributes, or text. An element has a name, and may have: a set of attributes, and a list of element and/or text children. A complete order holds over the element and text children of a given element, whereas the order among the attributes of the same element node is undefined. An attribute has only a name and a value, both of which are strings. A sample XML document can be found in Figure 2.4.

A header called *prolog* contains an XML declaration describing for instance the encoding or the language of the document, or the version of the XML specification in use.

Schema information describing the structured allowed in an XML document can be specified using various formalisms. The most commonly used for this task are Document Type Definitions (DTD) [86] and XML Schema Definitions (XSD) [81, 82]. The former is an integral part of the specification of SGML, a language that generalises and precedes XML, and allows specifying the types of attributes, as well as restricting which elements and attributes an element may have as children. For instance, one can specify that only a certain number of a given element are allowed as children of one parent element.

XML Schema, on the other hand, is a newer separate standard from W3C. Almost all of the specifications using DTD can be translated into equivalent XML Schema[1]. The main advantage of XSD with respect to DTD is that it is possible to specify different legal internal structures for an element depending on the context

[1]DTD supports entities, a feature similar to macros that doesn't exist in XSD.

```
<person firstName="John" lastName="Smith">
        <address>
                <street>21 2nd Street</street>
                <city>New York</city>
                <state>NY</state>
        </address>
        <address>
                <street>Rue d'Argout</street>
                <city>Paris</city>
                <country>France</state>
        </address>
        <phone type="home">212 555-1234</phone>
        <phone type="office">646 555-4567</phone>
</person>
```

Figure 2.4: Example of XML document

of its occurrences. Thus, an *author* element may have certain constraints on its structure if it appears within *book*, and other constraints if it appears within *movie*. Further, XML Schema provides support for namespaces, allows specifying the order of the children of a node, not just their number of occurrences, and provides a rich typing system for literals [83].

XPath, the XML Path Language [87], is a query language for XML documents based on the description of paths. Recent iterations of the specifications have introduced higher level features but its core concept remains that of *path expression*, which can be seen as a sequence of *steps* separated by / or //.

Each step contains an *axis*, for instance *child*, *descendant*, *parent*, *preceding-sibling*. If no axis is specified, it defaults to the *child* axis. Additionally a step may contain a *test* such as testing for a specific node name or type and a *list of predicates*, to be evaluated for instance over the value of a node.

The semantics of evaluating a path expression on an XML document is as follows:

- Replace all occurrences of // with a step of axis *descendant-or-self* having no test nor predicate.
- Create a set of nodes, the *context items*, containing initially only the root of the document if the first character of the path expression is /. If there is no initial /, the initial context items should be provided, for instance by the evaluation of another path expression. Note that all the non initial / act only as syntactical separators.

- For each step, following their order in the path expression: replace the context items by the set of nodes reachable from its current elements using the current step axis (the new context items are the children, descendants, parents etc. of the nodes in the current context items). Prune from the context items the nodes that fail the test or do not satisfy all the predicates in the predicate list of the current step, then proceed to the next step. When there are no more steps, the context items are returned as the evaluation of the path expression.

For instance, `/person//phone[@type="home"]` is an XPath expression consisting of three steps: the first step selects all children of the root node labelled "person", the second, inserted by `//`, adds to the context items its descendants, and the third step replaces the context items with the nodes labelled "phone", children of a node in the context items, having an attribute named "type" whose value is "home".

This expression evaluated on the document from Figure 2.4 yields as result the one-element sequence:

```
<phone type="home">212 555-1234</phone>
```

XQuery is a query language using XPath expressions as building blocks and more traditional programming operators such as *for*, *if then else*, *try catch*, *where* and variables to construct and output XML documents [88].

Figure 2.5 shows an example of XQuery query that outputs an XML document containing one item element for each child of the root of a document named *"inventory.xml"*. The resulting items also get a *name* and a *quantity* child from the original document.

Note that XQuery has many features of a functional language, including functions as first class citizens, and is Turing complete.

JavaScript Object Notation or *JSON* for short, is another highly popular semi-structured document format that arose from the use of JavaScript in particular in Web site development. As of today, however, JSON has evolved as an independent specification [2, 1]. Similarly to XML, it can represent ordered unbounded finite trees, however JSON is more expressive: a JSON node can be either a dictionary (a set of key-value pairs), an array, or a set of values, whereas XML element children of the same node are always ordered. A JSON value can either be a node, or a typed literal such as a boolean, a string or a number. In contrast, XML typing of leaf (text) nodes can only be made through a "second" (external) schema specification such as a DTD or XML schema.

Figure 2.6 shows a simple JSON document.

JSON is becoming more present in web applications and is a default format for

```
<inventory>
{
  for $item in fn:doc("inventory.xml")
    let $quantity := $item/details/stock/@quantity
    return
      <item>
        <name>{$item/name}</name>
        { if ($quantity!=1)
        then return
          <quantity>{$quantity}</quantity>
        }
      </item>
}
</inventory>
```

Figure 2.5: Sample XQuery query.

exchanging or retrieving information through web services, for instance from social networks or content publishers.

Querying JSON Several query languages for JSON exist, such as JSONPath[2], that mimics XPath, Javascript[3], JSON Pointer[4], jaql[5], JSONiq[6], and JSONSelect[7] but none of them has the status of a standard.

2.3 Models for social data management

The advent of social network systems has lead to very large amounts of interesting data being exchanged through this medium. Research works have attempted to propose models for such data oriented social networks.

User-item-tag models In a social context, one is interested both in modelling the data posted by the users (which can be assimilated to *documents*), and the *relationships* between users and the documents they interact with. *User-item-tag* (UIT, in short) models represent social tagging applications, such as social networks or social bookmarking services like Flickr, Reddit, Delicious etc. in

[2] http://goessner.net/articles/JsonPath/
[3] https://tc39.github.io/ecma262/
[4] https://tools.ietf.org/html/rfc6901
[5] https://code.google.com/archive/p/jaql/
[6] http://jsoniq.org/index.html
[7] http://jsonselect.org/

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "height": 171.1,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

Figure 2.6: Sample JSON document.

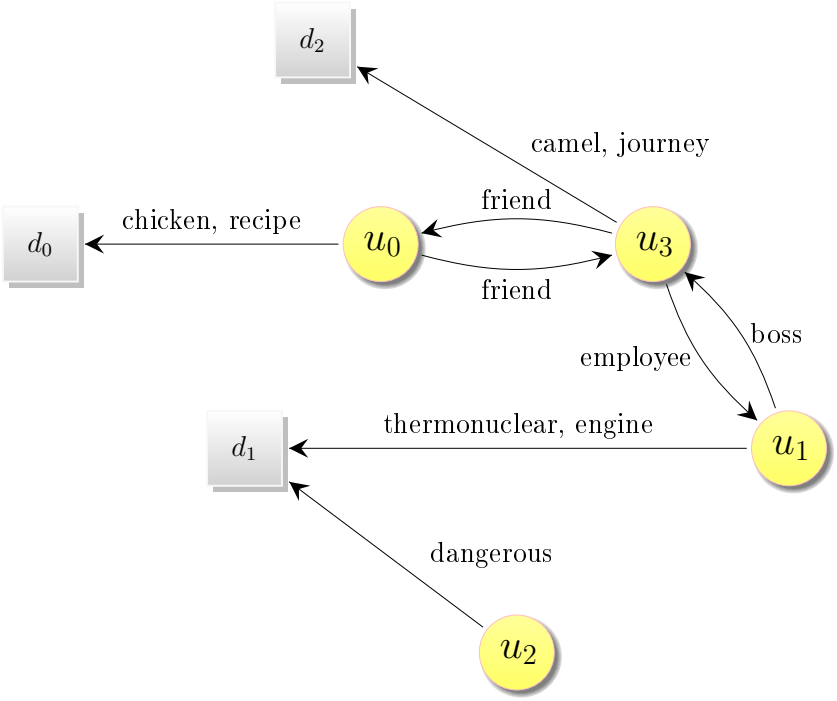

Figure 2.7: Sample UIT graph.

terms of a graph of users and items, where users may author or tag documents using keywords. The users may be linked one to another using named relations or weighted links.

This generic model for social tagging can use *explicit* tags [97, 58], i.e., when users manually enter textual tags on content, for instance in social networks like Del.icio.us or Last.fm. *Implicit* tagging refers to a context where the tags are not explicitly given but inferred from the users' behaviour. An example of implicit tagging would be the query logs of a search engine, from which we can interpret that a user tags with the query terms the search results that have been clicked on [10].

A sample instance of this model can be seen in Figure 2.7, where four users tag three documents.

Facebook is probably the most well known social network, being ranked as the second most popular website after google.com by Alexa[8] and the first by SimilarWeb[9]. Its model, called the *social graph*, is not known precisely, but the details made available [18, 24] differ significantly from UIT models. The nodes of the so-

[8] `http://www.alexa.com/siteinfo/facebook.com`
[9] `http://www.similarweb.com/website/facebook.com`

cial graph are not necessarily users or documents: they can also be places, pages, photos, groups etc. and are identified using a common scheme. The edges of the graph, that is the relationships between the nodes, are directed named relations such as friend (between two users), like (between a user and a page), live-in (between a user and a place) etc. Note that only one relation of each type is possible between two nodes: thus, a user cannot like twice the same page, or to live twice in the same place at a given time. Both nodes and edges have a content made of (key, value) pairs, for instance timestamps as values for the key "time". While the "like" tag behaves to some extent similarly to the UIT models (a relation from a user to a page, characterized by the keyword "like"), the restriction to at most one edge with the same source, target and relation makes tags with general keywords impractical, as it wouldn't be possible for a user to have several tag relations with different keywords to a document. Instead, new nodes are created to contain comments and other tags, and each of these new nodes has one relation with the user and one with the document. Another major conceptual difference between Facebook's graph model and UIT is that the friend relation (the main relation between users) is always symmetric. The social graph is typically *sparse*: there are only a few hundreds out-edges per node, while there are billions of nodes. At the same time, its diameter is very small: more than 99% of the nodes are in the same connected component and within 6 or less hops one to each other [77, 4].

Twitter Among the 10 most popular social network websites at the time of writing, Twitter also operates on a graph, but has fewer node and edge types than Facebook. Twitter nodes are either users or status (i.e., tweets) and edges are either follower, posted by, reply to status, or retweet [5]. To compensate for this smaller number of objects, Twitter's nodes include directly information that would be in another object in Facebook's social graph, for instance geographical information, or URLs. A distinctive feature of Twitter is the brevity of its messages: 140 characters. Another specificity is the concept of *hashtag*: a list of characters following a hash character ('#'), that allows tagging a status with this list of characters: Twitter supports real-time search for posts labelled with a given hashtag.

Yelp! is another social network that focuses on reviewing businesses and establishing partnerships with such businesses, allowing to order or book online [56, 6]. Much like Twitter, it is closer to UIT models in that it has few nodes types; users, businesses, and reviews, authored by a given user for a given business; review in particular may feature rich content.

2.4 Conclusion

This chapter has introduced the standard formalisms to represent and query data that is either of a semantic nature, with RDF and SPARQL, or that is semi-structured, with XML queried using XPath and XQuery, or JSON and jaql, JSONIQ, etc. We also exposed UIT, a generic but somewhat limited model for social data, and surveyed the models of some of the most popular social networks.

Chapter 3

State of the art

This chapter presents the state of the art in the different fields that study the core concepts used in this thesis: top-k search (Section 3.1) and social, semantic and structured data (Section 3.2).

3.1 Top-k search

Top-k search aims at computing and returning the most relevant results to a query instead of all its answers. Focusing on these particularly interesting results both improves the user experience, and decreases the search effort.

Section 3.1.1 presents seminal contributions on top-k search both in the database community, where there is a strong emphasis on the relations and the structure of the data searched over, and in the information retrieval community, which focuses more on the textual properties of the queried documents. These methods underlie top-k search algorithms in several settings. A first setting, the one of objects stored across several relations via foreign keys in relational databases, is presented in Section 3.1.2. Section 3.1.3 focuses on top-k search over semi-structured data, building on the top-k querying algorithms introduced in Section 3.1.1 and on the notions of documents and their adapted query languages presented in Section 2.2. Section 3.1.4 presents works on top-k search over RDF graphs. Finally, Section 3.1.5 presents works on top-k search looking for documents or users, based on the social links between them.

3.1.1 Foundations of top-k search

Fagin's Threshold Algorithm Fagin, Lotem and Naor published an algorithm [28] often referred to as Fagin's Threshold Algorithm (TA). This algorithm considers

objects with several attributes, all scored beforehand within $[0, 1]$. These scores, $x_1, \ldots, x_n$, are aggregated into a single one using a function, t, given at the time of the query. The role of the scoring function t is to compute objects' relevance score (also in $[0, 1]$, higher is better) with respect to the query: for instance if the objects stand for pieces of text, and each x_i is 1 if the word w_i is present in this object and 0 otherwise, then the query composed of the two words w_i, w_j could be associated to the function $t(X) = \frac{x_i + x_j}{2}$. t would yield a score of 1 on a text having both words, a score of 0.5 on a text having only one, and a score of 0 for a text having none of them. One of the interests of this model is its generality: the x_i could also score other attributes than the presence of a word in a text, for instance features on audio or video content, on web pages etc.

When the aggregation function is increasing monotonously over each attribute, i.e., if $X = (x_1, \ldots, x_n)$ and $X'_i = (x_1, \ldots, x_{i-1}, x'_i, x_{i-1}, \ldots, x_n)$ with $x'_i \geq x_i$ then $t(X'_i) \geq t(X)$, it is possible to find the k objects with the highest aggregated score using inverted lists. For each attribute, a list of pairs (object, score) is created for every object and the score it has for the attribute under consideration; each such list is sorted in the decreasing order of its elements' scores. To find the top-k objects, one can go through the inverted lists sequentially (and potentially gather more information by looking for the objects' other attributes). Because the lists are ordered, it is always possible to have an upper bound on each unexplored object's attributes score. Moreover, t being monotonously increasing over each attribute, it yields an upper bound on the score of unexplored objects. When the minimum score of the k best-scored objects becomes higher than the best possible score of the unexplored ones, the search may stop with the guarantee that the best k results have been explored. This algorithm can be tuned to adapt to the cost of sequential and random accesses, so that the overall cost of the algorithm is within a constant factor of the optimal.

Top-k keyword search in Information Retrieval Top-k search in Information Retrieval focuses on retrieving relevant data using its textual properties. Documents and queries are typically represented as multisets of words, and the score of a documents for a query is a function of the size and intersection of their representation, leading to two strategies: *Term At A Time* (TAAT), which inspects documents once for each term in the query, and *Document At A Time* (DAAT), where each document is inspected once, using all the terms of the query [76].

In [17], the authors introduce Weak AND (WAND), a DAAT method bearing similarity with Fagin's TA and introduced in a similar timeframe. WAND proceeds in two phases. In the first phase, document scores are computed approximately and tested against a threshold, the k-th best score seen so far; documents scoring below this threshold are pruned. In the second phase, the exact scores of the remaining documents are computed, and the k best documents are returned.

3.1.2 Top-k search in relational databases

Top-k keyword search is a powerful tool on relational databases, as it allows users to easily obtain results without having to study the database schema in order to formulate structured queries.

In [42], the authors propose a method for computing the top-k answers to queries consisting of a set of keywords, over relational databases. It produces ranked trees of tuples (sometimes called Joining Network of Tuples) found in the database instance I. Such a tree of tuples $(t_i)_{i \in I}$ is built as follows:

- each t_i is a tuple from a relation R_i of the relational database;
- if t_i is the child of t_j then the primary key of R_i is a foreign key in the R_j and the values of these keys coincide on t_i and t_j;
- each leaf of the tree must have a positive score with respect to the query.

The score of such a tree is a combination of the scores of its tuples, which can for instance aggregate attribute scores like the Fagin's TA algorithm, or use classical scores from other fields such as information retrieval.

In this setting, the authors define the *tuple monotonicity* property for scores:

- if T and T' are two results with the same number of nodes for a query q, and it is possible to find a numbering of the nodes of T, $(t_i)_{i \in I}$, and a numbering of the nodes of T', $T' = (t'_i)_{i \in I}$, such that for all i $score(t_i, q) \leq score(t'_i, q)$
- then $score(T, q) \leq score(T', q)$.

Note that the two numberings need not be related: even if T and T' are isomorphic, t_i does not necessarily map to t'_i. This is a consequence of the fact that *score* takes as input a set of nodes without any order. This property is similar to the monotonicity property for Fagin's TA algorithm.

The top-k results for scores having the tuple monotonicity property can be computed in a similar fashion to Fagin's TA. The method consists of iteratively (i) examining new candidates created from the highest scored tuples of the database relations, (ii) establishing an upper bound on the score of the unexplored tuple trees, based on the tuples they may contain.

Extensions of this method include:

- mapping certain query keywords to predicates over the retrieved objects, instead of trying to find the keywords in the objects themselves [31];

- adding type-ahead search, to suggest completion while the user types the query [51];
- ensuring a certain diversity in the results [79];
- performance-improving techniques [12, 64].

Another perspective for extension is to add *user preferences* to the search queries. The *qualitative* approach to user preference is to explicitly define relations between items or keywords: in [32], users have a set of preferences over attributes, stating for instance that they prefer "English" to "French" for certain columns of certain relations. In [71], user profiles consist of a set of pairs $(C, k_1 \succ k_2)$ indicating that when all the keywords in the set C are in a query, results including the keyword k_1 are to be preferred over those including the keyword k_2. The *quantitative* approach [49, 50] relays instead the users preferences into the items score and selects the best items accordingly.

Other approaches, such as [43], return graphs of database entries and focus on the links between these entries using PageRank-like algorithms [62] to score them. This approach is not limited to relational databases, and the database entries need not be tuples, but may also be structured documents. As such, this allows more general forms of Joining Network of Tuples, where the relation between two nodes need not be the foreign key to primary key relationship, and can for instance be a named relationship in a graph.

3.1.3 Top-k search in semi-structured documents

At the crossroad between keyword search on structured data (for instance relational databases) and unstructured data, i.e., plain text, the search over semi-structured documents fuses both approaches. In this context, the search can not only return roots of documents but also their descendants, which are considered as valid documents. Top-k search over semi-structured documents using the XML format is introduced in [37]. The authors propose the idea that the more specific a document is, the better it should be ranked. Further, if an element contains a keyword of the query, then it is more specific than its ancestors for this query, unless they contain additional keywords. In particular, if each keyword of the query is present once in a tree, then the most specific result, and therefore the one returned, is the lowest common ancestor, abbreviated LCA, of the nodes where the keywords of the query occur.

In [37], the evaluation is done via an algorithm based on Fagin's TA: documents and their subtrees are given a score for each keyword they contain, depending on the query, and these scores are aggregated using a monotonous function into a global score. The major difference with Fagin's TA settings is that the keyword

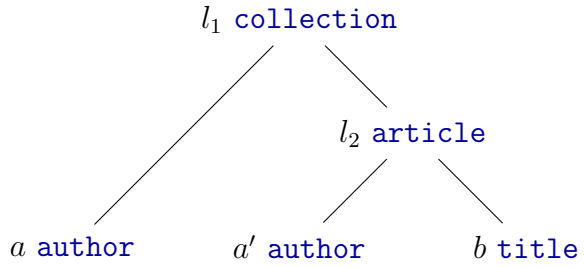

Figure 3.1: A sample document to illustrate MLCAs

score, for a particular subtree, contains a decay factor, $decay^{t-1}$, where t is the depth of the keyword in the subtree, and *decay* is a constant between 0 and 1.

This framework also supports keyword proximity: if a keyword k_1 not in the query is registered as close to a keyword k_2, part of the query, then the score for k_1 is taken into account, but reduced by a constant factor reflecting the proximity between k_1 and k_2. Because of keyword proximity, the score of a document for a particular keyword may depend on multiple nodes within this document, bearing proximate keywords. These nodes and keywords produce partial scores that are aggregated using a function that must be specified, for instance *max* or *sum*.

This initial approach has many natural extensions, notably variants of LCA. For example, Meaningful LCA [54] gives a type to documents' nodes and considers that for a query $\{k_1, k_2\}$, if within a single document a node a of type A contains k_1, and a node b of type B contains k_2, their LCA is meaningful only if:

1. there exists no ancestor of a that is also a descendant of the LCA and has a descendant of type B, and
2. there exists no ancestor of b that is also a descendant of the LCA and has a descendant of type A.

For instance, Figure 3.1.3 depicts a documents with 2 nodes a and a' of type *author*, and one node b of type *title*. In this document, the node l_1 is the lowest common ancestor of a and b but it is not a Meaningful LCA because there exists l_2: an ancestor of b and descendant of $l_1 = LCA(a, b)$ having a descendant a' of type *author*. The intuition behind the definition of MLCA is that l_2 associates elements of type *author* and *title* more directly than l_1 and that the association in l_1 may be incidental and not meaningful. Indeed in this case *article* are reasonable documents to contain related *title* and *author* while this is less the case for *collection*.

Another variant is Smallest LCA [95], that also requires the LCA not to contain a descendant with all the keywords of the query. Other notable works focus on providing search algorithms with better complexity, and improving the performances

for different variants of LCA [44, 98]. Going beyond classical XML documents, it is also possible to consider probabilistic XML documents [52], where some nodes have a given probability to appear.

Keyword search over structured documents can naturally be extended to incorporate structural constraints. The algorithms of [74] allow processing efficiently fuzzy XPath queries such as:

```
//book[about(., Information Retrieval XML) and about(.//reference, PageRank)]
                  //author[about(.//affiliation, Stanford)]
```

This query finds the authors of books containing the terms "Information Retrieval XML" and having descendants tagged as reference and affiliation containing the terms "PageRank" and "Stanford", respectively. But it can also find books with related contents, reference or affiliation using similarity measures coming from the information retrieval community. This is closely related to graph pattern matching with top-k [36]: given an input graph with some variables, the goal is to find an assignment of the variables and a subgraph of the instance that correspond to the query where the variables are replaced by their value, these assignments are ranked and a given number of the ones with the highest scores must be returned.

3.1.4 Top-k search in RDF graphs

Top-k search on RDF graphs focuses on finding minimal subgraphs of the search instance – an RDF graph – that contains all the keywords of the query. Because these subgraphs are minimal in terms of number of RDF triples included, they cannot contain loops and are therefore trees. Most works focus on scoring such trees and finding the top-scored ones.

In [14] the score of a tree depends on two factors:

- The lengths of the paths from the tree root to the keywords. Notice that the edges of the RDF graph may be weighted, in which case the length of a path is the sum of the weights of the edges it traverses.
- The importance of the nodes of the graph, for instance computed using PageRank [62] like algorithms.

Additional requirements can be imposed on the results, for instance that only the best tree rooted in each node may be considered [40], or that the returned trees should not overlap [55].

The exploration of the graph to find such trees can be done using several strategies, such as directly following the RDF graph's edges: starting from the query

keywords, and following the graph's directed edges, sets of potential roots for each keyword are found and then intersected to find roots of trees linked to all the keywords [14]. Alternatively, if the number of nodes related to one keyword is smaller than the ones related to the other keywords of the query, it it possible to find potential roots starting from this node and then check which of these candidates are indeed linked to all the keywords [46]. More recent exploration strategies make heavy use of precomputed indexes to alleviate the complexity: in [40], the authors partition the graph into blocks of customizable size and record

- the node distances within each block,
- which blocks contain which keyword,
- how to reach one block from another.

With a good partitioning strategy, it is possible to use this stored information to obtain a search complexity within a constant multiplicative factor of the optimal.

We distinguish two other approaches that don't return trees but nodes or general subgraphs instead. The former roughly corresponds to only returning the roots of the trees returned by the previous approaches [13]. The latter, studied in [75], expand on the tree approaches and consider one result to be an aggregation of paths between the query's keywords (or closely related terms such synonyms, hyponyms and hypernyms as they can be fond in Wordnet [29]). Because for a given result these paths link the same nodes, their aggregation is a subgraph that may include cycles.

3.1.5 Top-k search in a social context

Top-k techniques can also be used to search over social data. One can simply store this data into a relational database and use the methods discussed in Section 3.1.2. However, several improvements can be made by taking into account the social nature of the data. For instance one could add a *reputation score* to the attributes of each user or to the documents they authored. Reputation, trust, influence, authority, or other "people metrics" can be computed using implicit data (such as links with PageRank-like algorithms [63]) or using explicit data (such as user ratings [47]); such metrics are then integrated into top-k keyword search engines [68].

The expansion of social networks have lead to a scenario where users asking queries over social data are part of the social data itself. In this scenario, it is possible to use the knowledge on users to return them personalized results. Early takes on this approaches include personalized PageRank based on bookmarks [45]: each user has a set of preferred pages P, and the importance of each page is

computed using a modified PageRank algorithm where the random walk, instead of having a certain probability at each step to jump to a random page, jumps on one of the preferred pages P of the user. Because the computation of this modified PageRank algorithm for each user (in real time) is not tractable, precomputations are done on partial sets of preferred pages using an approximation of the graph, which are then assembled at run time. An extension of this work is to automatically infer this set of preferred pages [73].

Going beyond a pre-recorded set of preferences per user, it is possible to have the user be part of the social graph. In [96], the authors introduce *network-aware search* on a tagging network. This setting reproduces UIT models with a slight difference: the users are either taggers, who behave like normal users in UIT models, tagging items with keywords, or seekers who can only ask queries using a set of keywords. Based on Fagin's TA, this work introduces a *per-user network distance* with the searched items, and uses it as an attribute taken into account in the document score. The exact computation being too expensive, the authors introduce clustering techniques over the seekers to improve performance. While they show that finding optimal clusters of seekers is NP-Hard, they provide heuristics that work well in practice. [57] introduces the TOPKS algorithm that generalises this approach to UIT models, which allow seekers to also be taggers. It can use both approximate and exact social distance, and provides a simple parameter, $\alpha \in [0, 1]$, to tweak the importance of the social score with respect to the textual one. Network-aware approaches can be extended with attributes for the time elapsed since a document was created, or for the real world distance between the seeker and the authors of documents [53].

While all the previous works focused on searching documents, it is also useful to search for users. This is for instance the case on7 Facebook [24], where users are treated as rich objects with linked to names, places, interests etc. Several works are also interested in delegating the query of the user to another user of the network that might be better suited to answer it [41], which is indeed a form of top-k textual search for users in a social context. In [23], the authors consider network-aware search of user: in this work the social distance from the seeker comes from the paths leading from the seeker to the search candidates. The users along these paths are decomposed into several categories depending on the topology of the set of paths passing through them and the categories are used to score the paths.

Another approach known as Personalized Information Retrieval (PIR) stems from classical IR systems, the main difference being that, in PIR, the results for a query typically change from one user to another. This is achieved by expanding the query's set of keywords with keywords representing the users' interests, interests which are collected either in an implicit or an explicit manner. In the former case, the information is learned from the user behaviour, for instance using the previously submitted queries and clicked results [69, 70]. In the latter case, users' interests are explicitly stated, either directly as in [59] or by giving feedback to

proposed results [21, 11, 39].

PIR systems typically use only information from the user asking the query, but it is possible to integrate a social context to determine users close to the seeker and also exploit their interests. For instance, in [19], the neighborhood of the querier is ranked using direct and indirect social links, and their known interests are aggregated into the document scores, thus allowing top-k search integrating social data. The real-world location of the the social entity can be also integrated in the search [8].

3.2 Hybrid data models

We have seen in the previous chapter the common models for semantic, structured and social data taken separately. We focus in this section on models that integrate several of this aspects, not necessarily in a top-k context.

There exist few hybrid models including social and semantics aspects, examples of such models being mostly personal information management systems, where RDF data is queried using social information from the user. For instance, in [33], user recorded preferences are fed to a machine learning algorithm for training, and this algorithm is used to evaluate the relevance of RDF graph's nodes based on the preference of the user asking the query.

The rest of this section will focus on hybrid models including semantics and structure as, to our knowledge, there is no model integrating both social and structural data into a coherent whole.

Most models including both semantics and structure put semantics on top of structured documents: the semantics is either deduced from documents and used as a medium to query them or recorded on the documents for later use. The former direction is covered by document transformation and data integration, while the latter direction is part of document annotation.

The transformation of structured documents into semantic instances, such as XML documents into an RDF graph [30, 26], allows to use off-the-shelf query languages for RDF, like SPARQL, to examine structured data, potentially completed with semantic sources, natively in RDF.

The idea of extracting semantics from structured documents can also be used in data integration, to interface multiple documents with heterogeneous schemas. In [25], a set of rules on the labels and properties of HTML documents is used to extract RDF-like triples and a query language analogous to SPARQL is used to query the resulting ontology. More modern takes on data integration [22, 9], instead of extracting the semantics from the documents and using only this semantics afterwards, adopt a *mediator* approach: the documents are kept as they were and the queries on the (virtual) resulting ontology are rewritten as queries on the

documents that account for the rules mapping structure to semantics.

Document annotations are a way for users to attach semantics to particular nodes of structured documents such as Web pages, and to query these annotations. Annotations take the form of RDF triples associated to a node within a document and can be attached manually [38], semi-automatically [60], or directly by the authors of Web pages using a W3C standard [89]. The querying capabilities of document annotations models are often limited: typically they only provide a way to retrieve documents annotated with a given keyword.

In contrast with models that put semantics on top of structured documents, other approaches treat the two equally: this is the case for approaches that transform semantic graphs into structured documents or that use a model [35] integrating semantics and structure on an equal footing. Transforming RDF graphs into XML documents can be useful to use query language not targeted at RDF, such as XQuery [66], or to integrate an RDF graph and XML corpus by providing both translation from XML to RDF and the other way around [15]. This allows to handle both queries that may return XML document and queries that may return RDF graph at the cost of a loss of information due to the translations.

In [35], the authors integrate structural and semantics aspects from XML and RDF into a unified model, XR, and provide a rich query language, combining features from XQuery and SPARQL. An instance of this model is a set of XML documents, and an RDF graph. Each node of each document is assigned a unique URI that may be used as a subject or an object in the RDF data. Note that due to the large number of such XML nodes, the assignment is not materialized. The query language XRQ over this model is a combination of Basic Graph Patterns over the RDF graph and of tree patterns over the XML documents.

3.3 Conclusion

In Chapter 2, we discussed the representation and querying of data that have either a social, semantic or structured dimension. In this chapter, we build upon those representations to introduce the most prominent models for top-k search on at least one of those dimensions and for representing and querying data including at least two of those dimensions.

In the next chapter we introduce our model `S3`, that includes the three dimensions and in the following chapters we develop top-k querying an `S3` instance.

Chapter 4

S3: A model for structured, social and semantic data

4.1 Requirements

There exists currently no formal framework unifying the three dimensions of structured, social, and semantic data. We first identify a list of requirements such a framework should meet to fully exploit the content shared in social settings. We illustrate our requirements using Figure 4.1.

R0. The model must capture **explicit social connections** between users, e.g., u_1 is a friend of u_0 in Figure 4.1, and **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

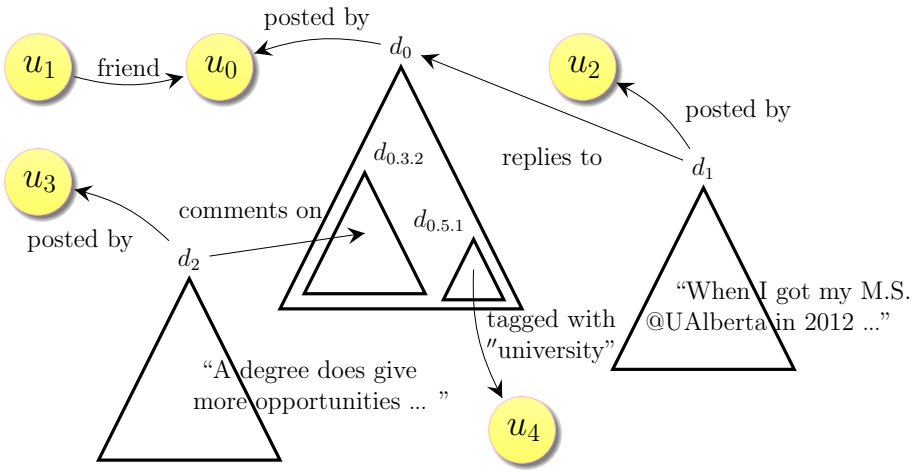

Figure 4.1: Motivating example for the data model requierements.

tagging behavior.

R1. The current wealth of ways to publishing and disseminate content on the Web (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 may wrongly believe that u_0 and u_2 have no relation to each other.

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 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 analysed 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.

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

Table 4.1: Main data model notations.

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.

4.2 Model definition

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 4.2.2. We add to our model structured documents in Section 4.2.3, and tags and user-document interactions in Section 4.2.4. Section 4.2.5 introduces our notion of social paths. Table 4.1 recaps the main notations of our data model.

4.2.1 Weighted RDF graphs

In the following, we assume, without loss of generality, that all RDF graphs are saturated; many saturation algorithms are known, including incremental [34] or massively parallel ones [78].

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 $a,\ b \vdash^i_{\text{RDF}} c$ 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"

In other settings, it may be possible to generalize this to support inference over triples of any weight, e.g., if u_1 hasName "AntonioMarez" and hasName $\hookleftarrow_d$ Person hold with weight 0.5, this could lead to u_1 type Person with a weight of 0.25, in the style of probabilistic databases. However this would also create new challenges as a triple could be deduced in several ways, requiring to aggregate the probability of a fact based on different reasons for it to happens whose independence is unknown.

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 4.2.1 (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 $\prec_{sc}$ k *or* b $\prec_{sp}$ k *in* I*, we have* b $\in Ext$(k)*.*

For example, given the keyword *degree*, and assuming that M.S. $\prec_{sc}$ 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[1].

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.

4.2.2 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", "co-worker" etc., we introduce the special property S3:social, and model any concrete relationship

[1] 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.

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 u_1 to a user u_2: u_1 S3:social 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 [35] 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 $\prec_{\text{sp}}$ S3:social.

4.2.3 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 that each text appearing in a document has been segmented 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 4.1, 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 4.2.1. *Based on the sample document shown in Figure 4.1, the following triples are part of* I*:*

$$\begin{array}{ll} \mathrm{d}_{0.3.2}\ \text{S3:partOf}\ \mathrm{d}_{0.3} & \mathrm{d}_1\ \text{S3:contains}\ \text{“M.S.”} \\ \mathrm{d}_{0.3}\ \text{S3:partOf}\ \mathrm{d}_0 & \mathrm{d}_1\ \text{S3:nodeName text} \end{array}$$

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

$$\begin{array}{ll} \text{S3:partOf} \hookleftarrow_{\mathrm{d}} \text{S3:doc} & \text{S3:partOf} \hookrightarrow_{\mathrm{r}} \text{S3:doc} \\ \text{S3:contains} \hookleftarrow_{\mathrm{d}} \text{S3:doc} & \text{S3:nodeName} \hookleftarrow_{\mathrm{d}} \text{S3:doc} \end{array}$$

which read as follows: 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.

Fragment position We need in our model to assess how closely related a given fragment is to one of its ancestor fragments. For that, we use a function $pos(d, f)$ that 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 [61, 72]. Then, $pos(d, f)$ returns the list of integers $(i_1, \ldots, i_n)$ 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 4.1, $pos(d_{0.3.2}, d_0)$ may be $(3, 2)$.

Example 4.2.2. *Considering again Figure 4.1, sample outputs of the pos function are:*

$$\begin{array}{c} pos(d_0, d_{0.3.2}) = [3, 2] \\ pos(d_1, d_2) = [\,] \end{array}$$

4.2.4 Relations between structure, semantics, users

We now show how dedicated `S3` classes and properties are used to encode all necessary 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 as 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 4.1, u_4 tags $d_{0.5.1}$ with the keyword "university", leading to the triples:

$$\begin{array}{cc} \text{a type S3:relatedTo} & \text{a S3:hasSubject } \mathrm{d}_{0.5.1} \\ \text{a S3:hasKeyword "university"} & \text{a S3:hasAuthor } \mathrm{u}_4 \end{array}$$

In this example, a is a *tag* (or annotation) resource, encapsulating the various tag properties: its content, the author, and the tagged ressource. 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 not necessarily consist of 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. Thus, in a restaurant rating site, a tag ranges from $\star$ (terrible) to $\star\star\star\star\star$ (excellent); in a collaborative question answering site, users tag questions with one of the existing discussion topics, e.g., "clock" and "cpu-speed" for a question related to CPU overclocking etc. Tags may also be produced by programs, e.g., a natural language processing (NLP) tool may recognize a text fragment related to a person, an image processing (IP) software may identify that an image is related to a certain object, etc.

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 an NLP software, this leads to the I triples:

$$\begin{array}{c} \mathrm{a}_2 \text{ type NLP:recognize} \\ \text{NLP:recognize } \prec_{\mathrm{sc}} \text{ S3:relatedTo} \end{array}$$

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. When user u posts document c, which comments on document d and possibly cites part of it, each fragment copied (cited as such) from d into c is now part of c and thus has a new URI.

Class	Semantics
S3:user	the users (the set of its instances is Ω)
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 4.2: Classes and properties in the `S3` namespace.

Example 4.2.3. *In Figure 4.1,* d_2 *is posted by* u_3*, as a comment on* $d_{0.3.2}$*, leading to the following* I *triples:*

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

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: s $\bar{\text{p}}$ o $\in$ I iff o p s $\in$ I where $\bar{\text{p}}$ is the inverse property of p. For instance, u_0 S3:$\overline{\text{friend}}$ u_1 holds in the example of Figure 4.1.

Table 4.2 summarizes the previously discussed `S3` classes and properties, while Figure 4.2 illustrates an I instance.

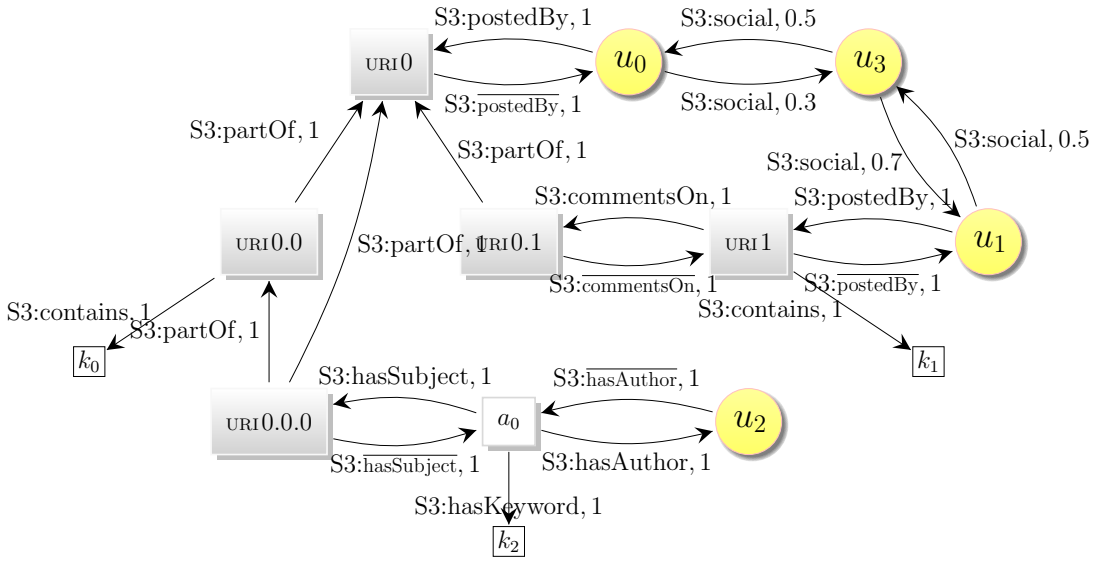

Figure 4.2: Sample S3 instance I.

4.2.5 Social paths

We define here a representation of the relations between users established either through explicit social links, or through user interactions, that we call *social paths*. To formulate their definition, we first identify the steps of the paths, denoted network edges, and when two steps can be chained via a notion that we call *vertical neighborhood*.

We call **network edges** those I edges encapsulating quantitative information on the links between user, documents and tags, i.e., *the set of edges whose properties are in the namespace* S3 *other than* S3:partOf*, and whose subjects and objects are either users, documents, or tags.*

Definition 4.2.2 (Network edges)**.** *We define:*

$$\mathrm{I}_{\text{net}} = \{\mathtt{s}\ \mathtt{p}\ \mathtt{o}\ \mathrm{w} \,|\, \mathtt{p} \in \mathtt{S3} \setminus \{\text{S3:partOf}\}, \mathtt{s}, \mathtt{o} \in (\Omega \cup D \cup T)\}$$

For instance, in Figure 4.2, u_1 S3:social u_3 0.5 and URI0 S3:$\overline{\text{postedBy}}$ u_1 are network edges; URI0.0 S3:contains 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, solely 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 4.1), this is indeed a form of social interaction, one we account for.

But when two users interact with unrelated fragments of the same document, such as u_3 and u_4 on disjoint subtrees of d_0, this does not establish a social link

between u_3 and u_4, since they may not even have read the same text[2]. We introduce:

Definition 4.2.3 (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.*

In Figure 4.2, URI0 and URI0.0.0 are vertical neighbors, so are URI0 and URI0.1, but URI0.0.0 and URI0.1 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[3]. We can now define social paths:

Definition 4.2.4 (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 4.2, $u_2 \xrightarrow{\text{u}_2\ \text{S3:}\overline{\text{hasAuthor}}\ \text{a}_0\ 1} a_0 \xrightarrow{\text{a}_0\ \text{S3:hasSubject}\ \text{URI0.0.0}\ 1} \text{URI}0.0.0 \dashrightarrow \text{URI}0 \xrightarrow{\text{URI0}\ \text{S3:}\overline{\text{postedBy}}\ \text{u}_0\ 1} u_0$ is an example of such a path (the dashed line: URI0.0.0 $\dashrightarrow$ URI0, is not an edge in the path but a connection between vertical neighbors, URI0.0.0 being the end of an edge and URI0 the beginning 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 URI0.1 to URI0.0.0 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-based 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:

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

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

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

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 4.2.4. *In Figure 4.2, consider the path:*

$$p = u_0 \xrightarrow{\text{u}_0\ \text{S3:postedBy}\ \text{URI0}\ 1} \text{URI}0 \dashrightarrow \text{URI}0.0.0 \xrightarrow{\text{URI}0.0.0\ \text{S3:}\overline{\text{hasSubject}}\ \text{a}_0\ 1} a_0$$

Its first edge is normalized by the edges leaving u_0*: one leading to* URI0 *(weight* 1*) and the other leading to* u_3 *(weight* 0.3*). Thus, its normalized weight is* $1/(1 + 0/3) = 0.77$*.*

Its second edge exits URI0.0.0 *after a vertical neighborhood traversal* URI0 $\dashrightarrow$ URI0.0.0*. It is normalized by the edges leaving* $\texttt{neigh}(\text{URI}0)$*, i.e., all the edges leaving a fragment of* URI0*. Its normalized weight is* $1/(1 + 1 + 1 + 1) = 0.25$*.*

4.3 Relationships with existing models

We argue in this section that most existing data models, as described in the state of the art (Chapter 3), can be integrated in `S3`.

Due to the way it is built, `S3` includes natively RDF and models effortlessly most structured and semi-structured document formats. There exist however a few minor limitations to the fidelity of the representation of XML and JSON documents in `S3`:

- the order of siblings, in XML, or of the elements of an array, in JSON, is not expressed in `S3`. Note however, that this information is only used rarely. Indeed, it can be exploited by systems supporting the full expressivity of XPath or JSON equivalents, but we found no evidence of it being actually used in the literature and it has no effect on the score of documents or on the value obtained by LCA or its variants.

- the content of the nodes are kept unordered: here again loosing the ordering has no actual impact on the integration of existing models, where the de-facto standard is to check if a keyword is included, and not to use its position relatively to other keywords of the same node.

- schemas such as DTD or XSD (recall Section 2.2) can no longer be enforced once a document is represented in S3. However, since we do not modify documents, there is no need to check them again, and this is never an issue.

Including RDF, structured, and semi-structured documents allows to integrate many models seamlessly, we discuss in the rest of this section the integration of data models with other distinctive features.

Social data models Because of the lack of a well established standard to model social relations each data model must me mapped manually to S3, we present here a few examples.

In UIT models, such as [96], user can relate to each other via weighted relations and can tag items with keywords. This translates easily in S3: items are mapped to single node documents without contents, tags become S3tags, that is resources of type S3:relatedTo and weighted relations between users are encoded using the S3:social property in weighted triples.

In [23], the authors use simple directed link to connect the users, which are annotated with keywords: this could be translated as a S3:social triple with weight 1, and one document with a single node per user, which they have authored and that contain all the keywords they are annotated with.

Integration of relational databases in S3 While, for our purposes, integrating relation database was not a priority, this can be done in a relatively straightforward manner, as relations can be interpreted as both RDF graphs or structured documents. Thus one can export a relational database into S3, for instance by creating a class for each relation and a property for each attribute of the class: for the relation $R_1(att_1, att_2)$ one could create the class R_1 and the associated properties att_1 and att_2 such that a tuple $(x_1, x_2) \in R_1$ is mapped into the triples

$$\mathrm{b\ type\ R}_1 \quad \mathrm{b\ att}_1\ \mathrm{x}_1 \quad \mathrm{b\ att}_2\ \mathrm{x}_2$$

4.4 Conclusion

We have presented in this chapter a model: S3, that integrates social, semantic and structural data into a single weighted RDF graph. S3 meets the requirements (Section 4.1) for a model unifying these three dimensions, as we show here.

Genericity, extensibility and interoperability (**R6**) are guaranteed by the reliance on the Web standards RDF (Section 4.2.1) and XML/JSON (Section 4.2.3). The small set of predefine S3 classes and properties can be specialized to models

many different types of social and structural interactions, e.g., through application-dependent queries (see Extensibility in Section 4.2.2). The modelling capacities of `S3` include most popular social networks and exiting data models (Section 4.3) and can be used to unify several social networks into a single instance.

Our document model (Section 4.2.3) meets the requirement of rich structured data (**R2**) where interactions such as comments target a specific part of the documents. The usage of RDF in documents and items ensures that they support a rich semantics (**R3**). The relations between structure, semantics and users (Section 4.2.4) allow stating the explicit relations within users and documents, while the implicit relations between users are modelled via social paths (Section 4.2.5) satisfying **R1**. Higher-level tags (**R4**) also stem from the relations between documents. For what concerns formal semantics (**R5**), the data model has been described formally in this chapter and we will consider the description of queries in the next one.

Chapter 5

Top-k search in S3

In this chapter, we study top-k keyword search within S3 instances.

In Section 5.1 we devise a ranking-based query model and provide concrete examples of score functions. Then we propose a query answering algorithm for this model in Section 5.2. Finally, we discuss the similarities and differences with other query models from the literature in Section 5.3.

5.1 Query model

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 5.1.1 defines queries and their answers. After some preliminaries presented in Section 5.1.2, we introduce a *generic score*, which can be instantiated in many ways, and a set of *feasibility conditions* on the score, which suffices to ensure the termination and correctness of our query answering algorithm (Section 5.1.3). We present our concrete score function in Section 5.1.4.

5.1.1 Queries

S3 instances are queried as follows:

Definition 5.1.1 (Query). *A query is a pair (u, ϕ) where u is a user and ϕ 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[1].

[1] This assumption is standard in XML keyword search, e.g., [20].

As customary, top-k answers are ranked using a score function: $s(q, d)$ returns for a document d and query q a value in $\mathbb{R}$, based on the graph I. The top k results are recursively defined as the top $k-1$ best results, plus the best among the documents which are neither fragments nor ancestors of the $k-1$ best results.

Definition 5.1.2 (Query answer)**.** *A* top-k answer *to the query q using the score s, denoted $T_{k,s}(q)$, is defined inductively over k as follows:*

- $T_{0,s}(q) = \emptyset$
- *if $k > 0$, $T_{k,s}(q)$ contains exactly $T_{k-1,s}(q)$ plus any one document from*

$$\underset{d \in D \setminus \mathtt{neigh}(T_{k-1,s}(q))}{\operatorname{argmax}}(s(q, d))$$

that is: any document having the best possible scores among the ones not in the vertical neighbourhood of the query answer $T_{k-1,s}(q)$.

Here, argmax is not necessarily unique: there might be ties. If a group of documents have identical scores, any of them can be in the query answer (ties are broken in some non-deterministic way). For instance, if the graph consists of four documents scored $(d_1, 0.8)$, $(d_2, 0.6)$, $(d_3, 0.6)$, $(d_4, 0.5)$ and we ask for the top 2 results, $\{d_1, d_2\}$ as well as $\{d_1, d_3\}$ are valid answers.

On the other hand when argmax is empty (i.e., less than k documents match q) every relevant document is returned.

5.1.2 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.

Definition 5.1.3 (Connections between a document d and a keyword k)**.** *We denote the set of direct and indirect connections between a document d and a keyword k by $con(d, k)$. Each connection is a three-tuple $(type, f, src)$ such that:*

- $type \in \{\text{S3:contains}, \text{S3:relatedTo}, \text{S3:commentsOn}\}$ *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 let f, f' be fragments of d and d', respectively[2]. Further, let k, k' be keywords such that $k' \in Ext(k)$, and let $src \in \Omega \cup D$ be a user or a document.

Documents connected to the keywords of their fragments If the fragment f contains a keyword k, i.e., f S3:contains k $\in$ I, 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 4.1, if the keyword "university" appears in the fragment whose URI is $d_{2.7.5}$, then $con(d_2$,"university") includes (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 (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 4.1 creates the connection (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 (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.

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

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 4.1 through a no-keyword tag a_5, the latter tag is related to "university" through: (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 4.1:

- since d_2 is connected to "university" through (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 (S3:commentsOn, $d_{0.3.2}$, d_2).

Formally, whenever d′ S3:commentsOn f and there exist $type'$, $frag'$, src such that $(type', frag', src) \in con(d', k)$, we have:

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

5.1.3 Generic score model

We now introduce a set of *proximity* notions, based on which we discuss the scoring of candidate answers. Specifically, we shall 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:

- $\overrightarrow{prox}(\cdot) = 1$, 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 strengths of all connections between two nodes.

Definition 5.1.4 (Social proximity)**.** *The social proximity measure* $prox : (\Omega \cup D \cup T)^2 \rightarrow [0, 1]$*, 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 $|\cdot|$ *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 4.2, 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 5.1.2, we define the generic score as follows:

Definition 5.1.5 (Generic score)**.** *Given a document* d *and a query* $q = (u, \phi)$*, the score of* d *for* q *is:*

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

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 ϕ. 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 4.2.3) 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 5.1.2).

- 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 *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 of length n going from a to b:

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

Then, the first property states that there exists a function U_{prox} with values in $[0, 1]$, taking as input:

- the bounded social proximity for paths of length at most $n - 1$,
- the proximity along paths of length n, and
- the length n, and such that:

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

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.

5.1.4 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 4.2.5) 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 5.1.4, $prox$ 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 5.1.1 (Social proximity). *Let us consider in Figure 4.2 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:*

$$p = u_0 \xrightarrow{\text{u}_0\ \text{S3:postedBy}\ \text{URI0}\ \frac{1}{1+0.3}} \text{URI0}$$

which accounts for a partial social proximity:

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

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 5.1.6 ($\texttt{S3}_k$ score). *Given a query (u, ϕ), the $\texttt{S3}_k$ 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 5.1.5 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., [20] (see also Section 2.2).

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).

Lemma 5.1.1 (Score feasibility)**.** *The* $\texttt{S3}_k$ *score function (Definition 5.1.6) has the feasibility properties (Section 5.1.3).*

Proof. (Lemma 5.1.1)

In this proof we examine all the feasibility properties, in the order they are defined, and show that they apply on the concrete score, denoted *score* for the rest of this proof.

Relationship with path proximity

For any path p, $\overrightarrow{prox}(p)$ is the product of the normalized weights (which are always non-negative, from Definition 4.2.5) along p, therefore it is always non-negative.

This implies that, for all a, b in an instance I, $prox(a,b) = C_\gamma \times \sum_{p \in a \rightsquigarrow b} \frac{\overrightarrow{prox}(p)}{\gamma^{|p|}}$ increases if you add paths in I (more paths in I can only mean 0 or more new paths in $\{p \in a \rightsquigarrow b\}$, which can only increase *prox*, since paths add non-negative terms to the summation). The score on any document and query, defined as:

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

also increases if there are more paths, since *prox* increases and $\eta > 0$. We have shown the first part of this requirement: the score should only increase if one adds more paths between a seeker and a data item.

We now need to exhibit a function U_{prox} with values in $[0,1]$, such that, for all $a, b \in$ I and $n > 0$:

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

To this end, let us develop $prox^{\leq n}(a,b) - prox^{\leq n-1}(a,b)$, as follows:

$$\begin{aligned} prox^{\leq n}(a,b) - prox^{\leq n-1}(a,b) =& C_\gamma \times \sum_{p \in a \rightsquigarrow_{\leq n} b} \frac{\overrightarrow{prox}(p)}{\gamma^{|p|}} - C_\gamma \times \sum_{p \in a \rightsquigarrow_{\leq n-1} b} \frac{\overrightarrow{prox}(p)}{\gamma^{|p|}} \\ =& C_\gamma \times \sum_{p \in a \rightsquigarrow_{n} b} \frac{\overrightarrow{prox}(p)}{\gamma^{|p|}} \\ =& \frac{C_\gamma}{\gamma^n} \sum (ppSet^n(a,b)) \end{aligned}$$

In particular, if we choose $U_{prox}(x, pp, n) = \frac{C_\gamma}{\gamma^n} \sum(pp)$, then:

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

Note that the first argument of U_{prox} is not used here; it would be used, for instance, for a function considering only the path with highest proximity: if $prox(a,b) = max_{p \in a \rightsquigarrow b}(\overrightarrow{prox}(p))$ then $U_{prox}(x, pp, n) = max(\{x\} \cup pp) - x$.

We now only have to prove that U_{prox}, as we defined it, is in $[0,1]$. Since $\frac{C_\gamma}{\gamma^n} = \frac{\gamma - 1}{\gamma^{n+1}} \in [0,1]$ and $\sum(ppSet^n(a,b)) \geq 0$, it is enough to prove that $\sum(ppSet^n(a,b)) \leq 1$.

We actually prove a stronger result: we show by induction on n that for all $a \in \mathrm{I}, n \geq 0$, $\sum_{\substack{b \in \mathrm{I} \\ p \in a \rightsquigarrow_n b}} \overrightarrow{prox}(p) \leq 1$, as follows:

- for $n = 0$, there exists only the empty path of length 0, and therefore $\sum_{\substack{b \in \mathrm{I} \\ p \in a \rightsquigarrow_0 b}} \overrightarrow{prox}(p) = \overrightarrow{prox}(\emptyset) = \prod_{e \in \emptyset} e.n\_w = 1$
- for $n > 0$, observe that a path of length n may be decomposed into a path of length $n-1$, followed by an optional jump between vertical neighbors (when $n > 1$), and then a path of length 1 (a network edge):

$$\begin{aligned}
\sum_{\substack{b \in \mathrm{I} \\ p \in a \rightsquigarrow_n b}} \overrightarrow{prox}(p) &= \sum_{\substack{b \in \mathrm{I} \\ p \in a \rightsquigarrow_n b}} \prod_{0 \leq i < n} p[i].n_w \\
&= \sum_{\substack{x \in \mathrm{I} \\ p' \in a \rightsquigarrow_{n-1} x}} \sum_{\substack{b \in \mathrm{I} \\ e \in \mathrm{I}_{\mathrm{net}} \\ p = (p':e) \in a \rightsquigarrow_n b}} \prod_{0 \leq i < n} p[i].n_w \\
&= \sum_{\substack{x \in \mathrm{I} \\ p' \in a \rightsquigarrow_{n-1} x}} \prod_{0 \leq i < n-1} p'[i].n_w \times \sum_{\substack{b \in \mathrm{I} \\ e \in \mathrm{I}_{\mathrm{net}} \\ p = (p':e) \in a \rightsquigarrow_n b}} p[n-1].n_w
\end{aligned}$$

Recall from the definition of path normalisation, Section 4.2.5, that if x is the ending point of a social edge in a path, and e is the next edge in this path, the normalized weight of e for this path, denoted $e.n\_w$, is defined as:

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

If we use this to compute the normalized weight of the last edge of paths of length n, we obtain:

$$\begin{aligned}\sum_{\substack{b\in \mathrm{I}\\ p\in a\rightsquigarrow_n b}} \overrightarrow{prox}(p) &= \sum_{\substack{x\in \mathrm{I}\\ p'\in a\rightsquigarrow_{n-1}x}} \prod_{0\le i<n-1} p'[i].n\_w \;\times \sum_{\substack{b\in \mathrm{I}\\ e\in \mathrm{I}_{\mathrm{net}}\\ p=(p':e)\in a\rightsquigarrow_n b}} (e.w/\sum_{e'\in out(\mathtt{neigh}(x))} e'.w)\\ &= \sum_{\substack{x\in \mathrm{I}\\ p'\in a\rightsquigarrow_{n-1}x}} \overrightarrow{prox}(p') \;\times \sum_{\substack{b\in \mathrm{I}\\ e\in \mathrm{I}_{\mathrm{net}}\\ p=(p':e)\in a\rightsquigarrow_n b}} (e.w/\sum_{e'\in out(\mathtt{neigh}(x))} e'.w)\end{aligned}$$

Here, $\{e \in \mathrm{I}_{\mathrm{net}} | b \in \mathrm{I}, p = (p' : e) \in a \rightsquigarrow_n b\}$ is simply the set of network edges e that can continue the path p' (which is ending in x). If p' is the empty path and $(p' : e)$ is a path starting from a (note that $a = x$ since x is the end of the path p' after starting from a and following 0 edge), then e must start from x and $e \in out(x) \subset out(\mathtt{neigh}(x))$. Otherwise, if p' has a positive length, then e must start from a vertical neighbor of x and $e \in out(\mathtt{neigh}(x))$. In any case, $e \in out(\mathtt{neigh}(x))$, therefore we have:

$$\begin{aligned}\sum_{\substack{b\in \mathrm{I}\\ p\in a\rightsquigarrow_n b}} \overrightarrow{prox}(p) &\le \sum_{\substack{x\in \mathrm{I}\\ p'\in a\rightsquigarrow_{n-1}x}} \overrightarrow{prox}(p') \;\times \sum_{e\in out(\mathtt{neigh}(x))} (e.w/\sum_{e'\in out(\mathtt{neigh}(x))} e'.w)\\ &\le \sum_{\substack{x\in \mathrm{I}\\ p'\in a\rightsquigarrow_{n-1}x}} \overrightarrow{prox}(p')\\ \sum_{\substack{b\in \mathrm{I}\\ p\in a\rightsquigarrow_n b}} \overrightarrow{prox}(p) &\le 1\end{aligned}$$

(The first implication accounts for when $out(\mathtt{neigh}(x)) = \emptyset$, the last implication is the induction hypothesis).

This proves that for all $a \in \mathrm{I}, n \ge 0$, $\sum_{\substack{b\in \mathrm{I}\\ p\in a\rightsquigarrow_n b}} \overrightarrow{prox}(p) \le 1$, and in particular that the first feasibility property is verified.

Long paths attenuation

We have just seen that for all $a \in \mathrm{I}, n \ge 0$, $\sum_{\substack{b\in \mathrm{I}\\ p\in a\rightsquigarrow_n b}} \overrightarrow{prox}(p) \le 1$. In particular, for all $n \ge 0,\ a, b \in \mathrm{I}$:

$$\begin{aligned} prox(a,b) - prox^{\leq n}(a,b) =& \sum_{i>n} C_\gamma \times \sum_{p \in a \rightsquigarrow_i b} \frac{\overrightarrow{prox}(p)}{\gamma^i} \\ \leq& C_\gamma \times \sum_{i>n} \frac{1}{\gamma^i} \\ \leq& \frac{\gamma - 1}{\gamma} \times \frac{\gamma}{(\gamma-1)\gamma^{n+1}} \\ prox(a,b) - prox^{\leq n}(a,b) \leq& \frac{1}{\gamma^{n+1}} \end{aligned}$$

Therefore, we can chose $B^{>n}_{prox} = \frac{1}{\gamma^{n+1}}$, which obviously tends to 0 as n tends to ∞, and verify the second feasibility condition.

Score soundness

Let us consider $f : g \mapsto score_{[g]}$. We show that it is monotonically increasing and continuous for the uniform norm.

First of all, let us explicit f, for all $g \in (\Omega \cup D \cup T)^2 \to [0,1]$:

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

Since f is defined only using projections, additions, multiplications, and exponentiations, it is continuous for the uniform norm. Let us now show that it is monotonically increasing. Let $g_1, g_2 \in (\Omega \cup D \cup T)^2 \to [0,1]$, such that $g_1 > g_2$ for the uniform norm. We have:

$$f(g_1) - f(g_2) = d, (u, \phi) \mapsto \prod_{k \in \phi} \Big(\sum_{(type,f,src) \in con(d,k)} \eta^{|pos(d,f)|} \times g_1(u, src) \Big) - \prod_{k \in \phi} \Big(\sum_{(type,f,src) \in con(d,k)} \eta^{|pos(d,f)|} \times g_2(u, src) \Big)$$

We observe that for all $d, (u, \phi), k \in \phi, (type, f, src) \in con(d, k)$ we have: $\eta^{|pos(d,f)|} \times g_1(u, src) \geq \eta^{|pos(d,f)|} \times g_2(u, src)$, this is because $\eta \geq 0$ and $g_1 > g_2$. Hence for all $d, (u, \phi)$: $(f(g_1) - f(g_2))(d, (u, \phi)) \geq 0$ and $f(g_1) \geq f(g_2)$. This proves that $f : g \mapsto score_{[g]}$ is monotonically increasing for the uniform norm and with it the third feasibility property follows.

Score convergence

Let $q = (u, \phi)$ be a query and $B \geq 0$ an upper bound on the social proximity between the seeker and any source. For all documents d, we have:

$$
\begin{aligned}
score(d,(u,\phi)) =& \prod_{k\in\phi}\Big(\sum_{(type,f,src)\in con(d,k)} \eta^{|pos(d,f)|}\times prox(u,src)\Big)\\
\leq& \prod_{k\in\phi}\Big(\sum_{(type,f,src)\in con(d,k)} 1\times B\Big) && \text{since } \eta<1 \text{ and } prox<B,\\
\leq& (O\times B)^{|\phi|}
\end{aligned}
$$

where O is the maximum number of connections a document may have in I (this does not depend on the query, nor on B). If we chose $B_{score}((u,\phi),B) = (O\times B)^{|\phi|}$, as we have already shown that $B_{score}(q,B) \geq score(d,q)$, it remains to show that this also tends to 0 when B does.

When $B \leq 1/O$: $(O\times B)^{|\phi|} \leq O\times B$, therefore, independently of q:

$$\lim_{B\to 0}(B_{score}(q,B)) \leq \lim_{B\to 0}(O\times B) \leq 0$$

and, since $B_{score} \geq 0$, we have $\lim_{B\to 0}(B_{score}(q,B)) = 0$, which proves the last feasibility property.

□

5.2 Query answering algorithm

We describe in Section 5.2.1 our *Top-k* algorithm called $\texttt{S3}_k$, which computes the answer to a query over an S3 instance using scores with the feasibility properties. We illustrate the functioning of our algorithm on a sample run in Section 5.2.2, and formally state its correctness in Section 5.2.3.

5.2.1 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 *highest possible score*, denoted *c.upper*. These scores are updated as new paths between the seeker and the candidates are found. When a candidate document is reached, we are aware of at least one path

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

Input : a query $q = (u, \phi)$
Output: the best k answers to q over an S3 instance I, $T_{k,s}(q)$

1 $candidates \leftarrow [\,]$ // initially empty list
2 $discarded \leftarrow \emptyset$
3 $borderPath \leftarrow [\,]$
4 $allProx \leftarrow \delta_u$ // $\delta_u[v] = \begin{cases} 1 & \text{if } v = u \\ 0 & \text{otherwise} \end{cases}$
5 $threshold \leftarrow \infty$ // Best possible score of a document not yet explored, updated in `ComputeCandidatesBounds`
6 $n \leftarrow 0$
7 **while not** `StopCondition`($candidates$) **do**
8 $n \leftarrow n + 1$
9 `ExploreStep`()
10 `ComputeCandidatesBounds`()
11 `CleanCandidatesList`()
12 **return** $candidates[0, k-1]$

from the seeker to the document, but we may lack the complete information needed to evaluate exactly its relevance with respect to the query, since there may be *other paths yet to be discovered between the two*, and such paths may impact the relevance bounds. Therefore, a key recurring task is to *refine as tightly and as early as possible our knowledge of the candidate documents scores*. During the exploration, candidates are kept sorted *by their highest possible score*; the exploration ends when we are certain that no candidate document outside the current top-k candidates may 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 node c the set of the connections (Section 5.1.2) 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, which moves away from the seeker as the search progresses.

$q = (u, \phi)$	Query: seeker u and keyword set ϕ
k	Required 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 5.1: 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

```
1 if ∃d, d' ∈ candidates[0, ..., k − 1], d ∈ neigh(d') then
2     return false
3 if length(candidates) < k and candidates ∪ discard = D then
4     return true
5 min_topk_lower ← ∞
6 foreach c ∈ candidates[0, ..., k − 1] do
7     min_topk_lower ← min(min_topk_lower, c.lower)
8 max_non_topk_upper ← candidates[k].upper
9 return max(max_non_topk_upper, threshold) ≤ min_topk_lower        // Boolean result
```

Termination condition Of course, the search should not explore the whole graph, but instead stop as early as possible, while returning the correct result. To this aim, we maintain during the search an upper bound on the 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 or even order the returned results; we just need to make sure that no document unexplored so far may be among the top k. Algorithm 2 outlines the procedure to decide whether the search is complete.

First, if the current candidate set comprises two documents such that one is a

Algorithm 3: Algorithm `ExploreStep`

Update: $borderPath$ and $allProx$

```
1  if n = 1 then
2      foreach network edge e in out(u) do
3          m ← e.target
4          if m is a document or a tag then
5              GetDocuments(m)
6          BorderPath[m].add(e)
7  else
8      foreach v ∈ I do
9          newBorderPath[v] ← ∅
10     foreach p ∈ borderPath do
11         foreach network edge e in out(neigh(p.end)) do
12             m ← e.target
13             if m is a document or a tag then
14                 GetDocuments(m)
15             newBorderPath[m].add(p||e)
16     borderPath ← newBorderPath
17 foreach v ∈ I do
18     newAllProx[v] ← allProx[v] + U_prox(allProx[v],
19         {prox→(p), p ∈ borderPath[v]}, n)
20 allProx ← newAllProx
```

fragment of another, the search must continue (lines 1-2), since it follows from our definition of query answers (Definition 5.1.2) that such a (document, fragment) pair cannot be returned in the result.

Next, the algorithm finds the smallest lower-bound relevance value among the top-k candidates, and compares it with the upper bound of the score of the best document *outside* the current top-k. If it can be established that the first document outside the current top-k is less relevant than all the documents currently in the result, the search can stop.

Graph exploration Algorithm 3 describes one search step (iteration), which visits nodes at a social distance n from the seeker. The algorithm updates the candidate set, the set of examined and discarded candidate documents, the set of paths of length n, and (most importantly) the seeker-centric proximity function $allProx$. 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

Algorithm 4: Algorithm `ComputeCandidateBounds`

Input : loop counter n
Update: *upper* and *lower* bounds of the *candidates*, *threshold*

```
1 foreach c ∈ candidates do
2     c.lower ← ⊕_gen({(kw, t, p, allProx[src]) |
3         (kw, t, p, src) ∈ connect[c]})
4     c.upper ← ⊕_gen({(kw, t, p, allProx[src] +
5         B^{>n}_prox(u, src)) | (kw, t, p, src) ∈ connect[c]})
6 threshold ← B_score(q, B^{>n}_prox)
```

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 5.1.3), 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, $allProx[src] = prox^{\leq n}(u, src)$.

In some social networks, users can only have a limited knowledge of the social interactions: for instance on Facebook, privacy settings may prevent a user, her relations or her content to be seen by other users. $\texttt{S3}_k$ can take into account this limited knowledge by exploring only the edges known by the seeker: *out*, the network edges outgoing for a set of nodes, become dependent on the seeker and is restricted to the network edges that she knows of. The publicly available component of *out* may be pre-computed and the a priori much smaller component specific to u added at run-time.

Computing candidate bounds The `ComputeCandidateBounds` algorithm (Algorithm 4) maintains during the search the lower and upper bounds of the documents in candidates as well as the best possible score of unexplored documents. A candidate's *lower* bound is computed as its score where its social proximity to the user[3] 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).

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

Algorithm 5: Algorithm `CleanCandidatesList`

Update: *candidates* and *discarded*

```
1  lower_bounds_list ← []                                  // Initially empty list
2  max_rank = k
3  foreach c ∈ candidates do
4      if |lower_bounds_list| < max_rank or
       c.lower ≥ lower_bounds_list[max_rank] then
5          lower_bound_list.insertSort(c.lower)
6      else if c.upper < lower_bounds_list[max_rank] then
7          candidates.remove(c)
8          discarded.add(c)
9      if No vertical neighbor of c may have a score equal or higher than c then
10         foreach f ∈ neigh(c) do
11             candidates.remove(f)                        // If f was a candidate
12             discarded.add(f)
13     else if |lower_bounds_list| < max_rank then
14         max_rank += 1
```

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 5.1.3). 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 5.1.2); 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[4]; it tends to 0 as n grows, due to the score convergence feasibility property (Section 5.1.3).

[4]See the Threshold Correctness Lemma (5.2.2).

Algorithm 6: Algorithm `GetDocuments`

Input : Document or tag x
Update: *candidates* and *discarded*

```
1  foreach d such that there exists a chain of triples from x to d in I using
   only S3:partOf, S3:commentsOn, S3:commentsOn⁻¹, S3:hasSubject and
   S3:hasSubject⁻¹ labels do
2      if d is a document and d ∉ (candidates ∪ discarded) then
3          connect[d] ← ∅
4          kw_index ← 0
5          while kw_index < length(φ) do
6              kw ← φ[kw_index]
7              if con(kw, d) is empty then
8                                          //A keyword from the query is missing
9                  kw_index ← length(φ)
10                 discarded.add(d)
11             else
12                 kw_index += 1
13                 foreach (type, frag, source) ∈ con(kw, d) do
14                     connect[d].add(
15                             (kw, type, pos(d, frag), source))
16     candidates.add(d)
```

Cleaning the candidate set Algorithm 5 removes from *candidates* some 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. In the former case, this translates into removing every document d whose score is less than that of k other candidates which are neither (i) neighbors pairwise nor (ii) neighbors with other candidates with scores greater than that of d.

Getting candidate documents Algorithm 6 checks whether every unexplored document reachable from a given document or tag through a chain of triples labelled S3:partOf, S3:commentsOn, S3:$\overline{\text{commentsOn}}$, S3:hasSubject, or S3:$\overline{\text{hasSubject}}$ (i.e., any edge that may connect two documents or tags), is a candidate answer. If yes, it is added to *candidates* and the necessary information to estimate its score, derived from *con*, is recorded in *connect*.

Anytime termination Algorithm 7 can be used to make our query answering algorithm (Algorithm 1) *anytime*. It outputs the k best candidate documents known so far, based on their current upper bound score.

Algorithm 7: Algorithm `AnytimeTermination`

Output: A list of candidates

```
1 return_list ← [ ]
2 while length(return_list) < k and candidates ≠ ∅ do
3 |   d ← candidates[0]          //candidates is kept sorted by upper bound
4 |   return_list.add(d)
5 |   candidates.remove(neigh(d))
6 return return_list
```

5.2.2 Sample run

We illustrate a run of $\texttt{S3}_k$, using the motivating example from Section 4.1, enriched with a query of two keywords originating from user u_2, as shown in Figure 5.1.

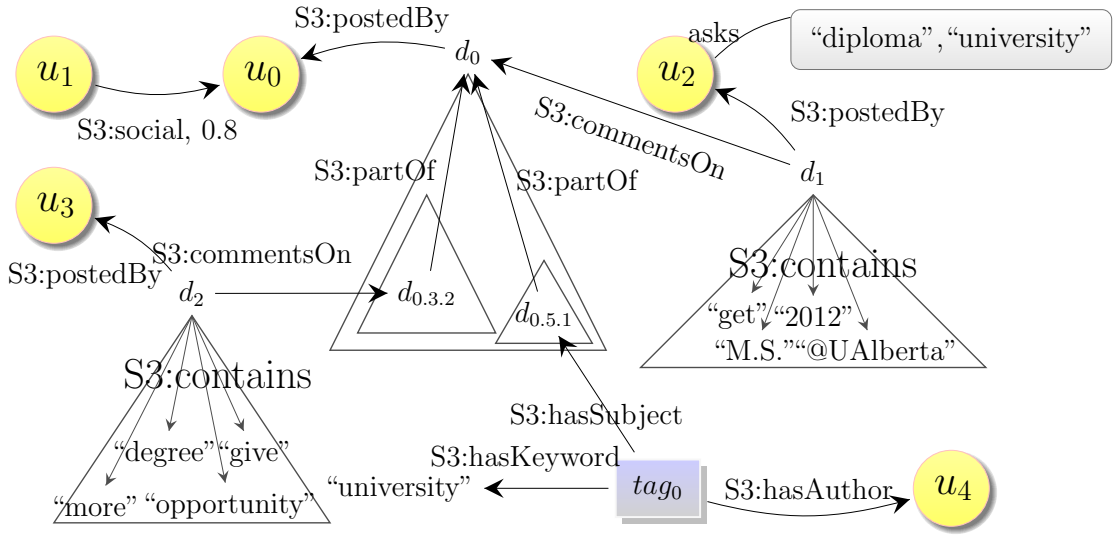

Figure 5.1: Graph for the sample run: data instance and a query.

For this run we consider that the expected number of results is $k = 5$, and the query is $\phi = (u_2, \{\text{"diploma"},\text{"university"}\})$. The graph I is the one presented in Figure 5.1 enriched with:

- typing of the URIs as suggested by the shape and color scheme: u_x in yellow circles are users (in S3:user), URIs of the form d_x with a triangle are of type S3:doc and tag_0 is a tag i.e., a resource of type T.
- triples with inverse properties: recall from Section 4.2.4 that we introduced

S3:$\overline{\text{postedBy}}$, S3:$\overline{\text{commentsOn}}$, S3:$\overline{\text{hasSubject}}$ and S3:$\overline{\text{hasAuthor}}$ and triples s $\bar{\text{p}}$ o $\in$ I whenever o p s $\in$ I

To preserve the readability of the figures we only show the inverse edges when they are relevant in the figure's context.

- triples from an ontology that don't contain `S3` classes or properties. These triples will only be used to compute the expansion of the query's keyword. We have shown the relevant triples in Figure 5.2: "diploma" get mapped to the URI diploma and because of the triple degree $\prec_{sc}$ diploma, *degree* $\in$ *Ext*("diploma") holds.

  Further, through RDFS entailment, degree $\prec_{sc}$ diploma and M.S. type degree produce M.S. type diploma and M.S. $\in$ *Ext*("diploma"). This leads to *Ext*("diploma") = {diploma, degree, M.S.}. Similarly the expansion of "university" is computed, for the purpose of the demonstration it is here limited to *Ext*("university") = {university}.

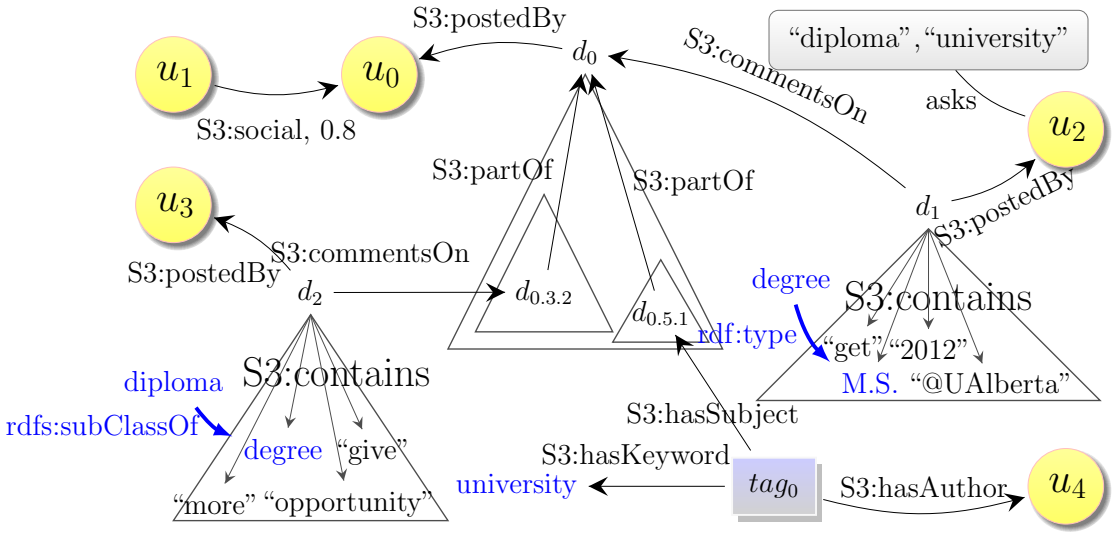

Figure 5.2: Triples relevant for keyword expansion.

After the initialization phase of the main algorithm (Alorithm 1, lines 1-6), there is no candidate and the threshold is infinite, therefore the stop condition is not verified and Algorithm 1 enters it's main loop (lines 7-11), starting an exploration step. Because this is the first time we execute Algorithm 3, the network edges going out of the seeker, u_2, are added to *borderpath* and the documents at the end of those edges are considered via Algorithm 6: GetDocuments. *out*(u) contains only one network edge: $u_2 \xrightarrow{\text{u}_2\ \text{S3:}\overline{\text{postedBy}}\ \text{d}_1\ 1} d_1$, as shown on Figure 5.3.

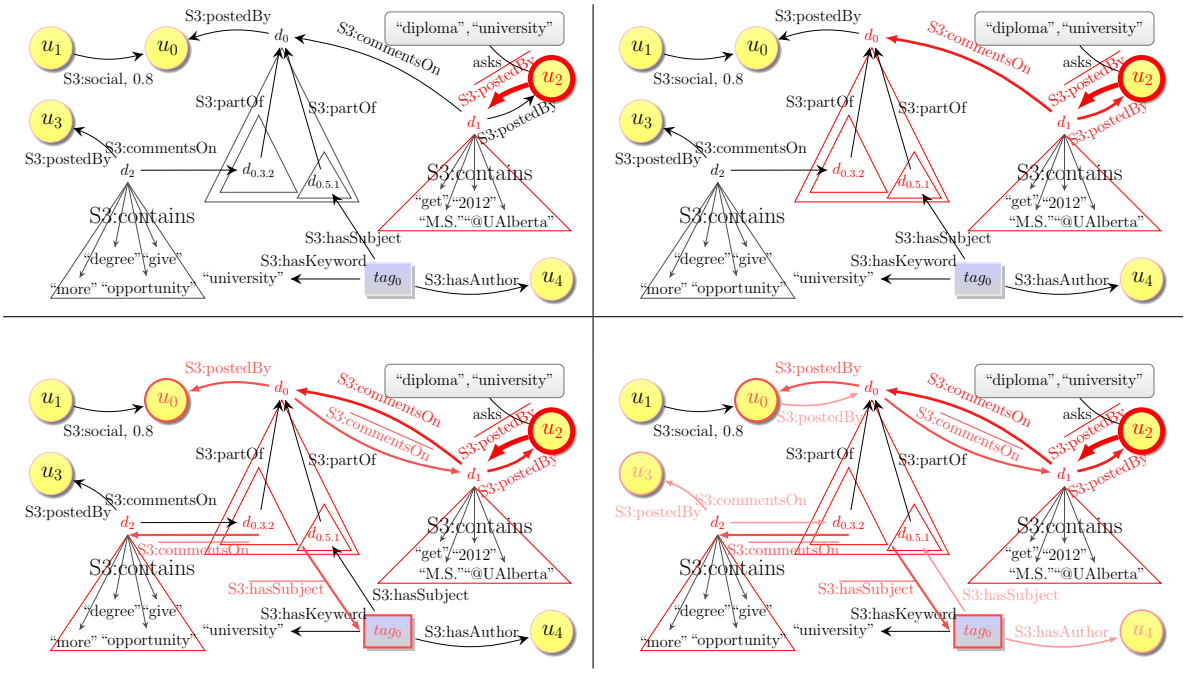

Figure 5.3: Social paths starting from the seeker, u_2, of length 1 (top left), 2 (top right), 3 (bottom left) and 4 (bottom right).

The document d_1 is the only document at a end of a network edge starting from u_2 and is therefore the only document tested by Algorithm 6: line 1 will select all the d, connected to d_1 using only S3:partOf, S3:commentsOn, S3:$\overline{\text{commentsOn}}$, S3:hasSubject and S3:$\overline{\text{hasSubject}}$. This is the case of:

- d_0, connected to d_1 via S3:commentsOn;
- $d_{0.3.2}$ and $d_{0.5.1}$, connected to d_0 with S3:partOf;
- tag_0 via $d_{0.3.2}$ and S3:hasSubject, and
- d_2 through $d_{0.3.2}$ and S3:commentsOn.

These 6 documents and tags are considered together because they might share connections with the keywords of the query: after making sure that we didn't test them already (line 2), Algorithm 6 proceeds to establish their connections and store them in the *connect* table. If a document doesn't have a connection with all the keywords of the query, the document is discarded (added to the *discarded* set); otherwise it is added to *candidates*. Here, only d_0 is not discarded, since it has a connection to "university" from tag_0 via its fragment $d_{0.5.1}$ and two connections to "diploma": one from d_2 via its fragment $d_{0.3.2}$ and one

from d_1. More formally, $con(d_1$,"university") $= \{$(S3:relatedTo, $d_{0.5.1}, u_4)\}$ and $con(d_1$,"diploma") $= \{$(S3:partOf, $d_{0.3.2}, d_2)$, (S3:commentsOn, $d_0, d_1)\}$, and therefore $connect[d_0] = \{$("university", S3:relatedTo, $d_{0.5.1}, u_4)$, ("diploma", S3:partOf, $d_{0.3.2}, d_2)$, ("diploma", S3:commentsOn, $d_0, d_1)\}$.

Once d_0 is added to *candidates* and all the other documents and tags are added to *discarded*, Algorithm 6 terminates and Algorithm 3 resumes at line 16, establishing the new *borderPath* as a single path: $u_2 \xrightarrow{\text{u}_2\ \text{S3:}\overline{\text{postedBy}}\ \text{d}_1\ 1} d_1$. Using the first feasibility property we update the proximity of all the nodes in I to take into account all paths of length at most $n = 1$. The next step in the main loop, after Algorithm 3, is to compute the bounds of the candidates using Algorithm 4: this simply updates the lower and upper bounds of each candidate (here only d_0) and the threshold using the feasibility properties and our complete knowledge of the paths of length at most n (here 1). When this is done Algorithm 5 is triggered to remove the candidates that have no chance to make it to the top-k.

We stress here that our running example does not illustrate all the aforementioned aspects, as only one possible candidate is present, no other candidates are ever removed, and no new documents can ever be found. Each iteration of the loop of Algorithm 1 will only increase n and compute all paths of length n and the proximities with the seeker taking into account paths of lengths at most n then recompute, through $ComputeCandidateBound$, the lower and upper bounds of the only candidate, d_0, using these proximities and refine the threshold.

The feasibility properties ensure that $threshold = B_{score}(q, B^{>n}_{prox})$ tends to 0 while n increases and that the lower bound of d_0:

$$\oplus_{gen}(\{(kw, t, p, allProx[src])|(kw, t, p, src) \in connect[d_0]\})$$

tends to the (positive) score of d_1. When the lower bound of d_1 becomes greater than *threshold*, the best possible score of an unseen document, i.e., the stop condition, Algorithm 2 returns $True$ on line 9 and $\texttt{S3}_k$ terminates, returning its k best candidates, that is $\{d_0\}$.

Figure 5.4 shows the evolution, for a sample run with numerous ties, of the number of candidates and their bounds as the number of iterations increases. The algorithm discovers rapidly some potential candidates and keeps track of their upper and lower bounds, then after a certain number of iterations (here approximately 10) all meaningful candidates have been found and the algorithm continues to explore until it can break the ties and return k candidates.

5.2.3 Correctness of the algorithm

The theorems below state the correctness of our algorithm for *any score function having the feasibility properties* identified in Section 5.1.3.

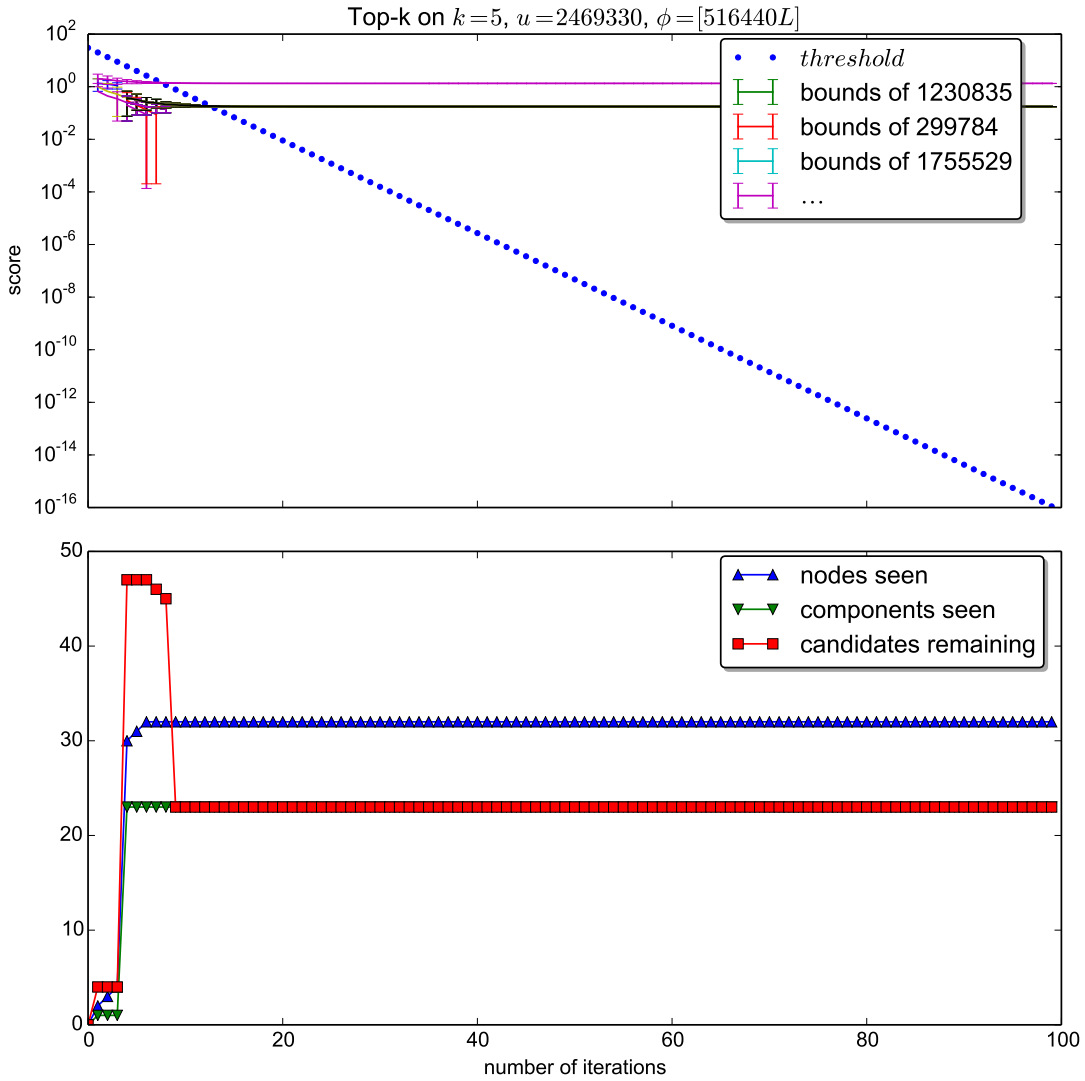

Figure 5.4: Internal state of the algorithm run on a query with ties.

Lemma 5.2.1 (Bounds correctness and convergence)**.** *At the end of each iteration of Algorithm 1, for every candidate c in candidates: $c.upper \geq score(c,q) \geq c.lower$. Furthermore, $\lim_{n\to+\infty}(c.upper) = score(c,q)$ and $\lim_{n\to+\infty}(c.lower) = score(c,q)$ where n is the number of iterations.*

Proof. (Lemma 5.2.1) *c.upper* and *c.lower* are only modified in Algorithm 4, so at the n-th iteration after the execution of Algorithm 4, we have:

$$c.upper = \oplus_{gen}\left(\{(k,t,p,allProx(src)) \mid (k,t,p,src) \in connect[c]\}\right)$$
$$c.lower = \oplus_{gen}\left(\{(k,t,p,allProx(src) + B^{>n}_{prox}(u,src)) \mid (k,t,p,src) \in connect[c]\}\right)$$

Since documents are added to *candidates* only through Algorithm 6 we know that $(k,t,p,src) \in connect[c] \Leftrightarrow k \in \phi,\ (t,f,src) \in con(c,k), p = pos(f,c)$.

Because we also know that $allProx[src] = prox^{\leq n}(u,src)$ we have:

$$c.upper = score_{[prox^{\leq n}]}(c,q)$$
$$c.lower = score_{[prox^{\leq n} + B^{>n}_{prox}]}(c,q)$$

where $score_{[f]}$ denotes our score function (Section 5.1.3) using f as the social proximity function. By definition, $prox^{\leq n} \leq prox$ and by the second feasibility property we know that $prox \leq prox^{\leq n} + B^{>n}_{prox}$ and $\lim_{n\to+\infty}(prox^{\leq n} - prox) = 0 = \lim_{n\to+\infty}(prox^{\leq n} + B^{>n}_{prox} - prox)$. The third feasibility property ensures that $g \mapsto score_{[g]}$ is monotonically increasing and continuous and therefore that $c.lower \leq score(c,q) \leq c.upper$ and that $\lim_{n\to+\infty}(c.lower) = \lim_{n\to+\infty}(c.upper) = score(c,q)$. □

Lemma 5.2.2 (Threshold correctness)**.** *At any time, for any document d not in candidates nor in discarded: threshold $\geq score(d,q)$.*

Proof. (Lemma 5.2.2) Let d be a document neither in *candidates* nor in *discarded*, at an iteration $n > 0$[5]. Because d is not in *candidates* nor in *discarded*, there is no path of length smaller than $n-1$ from u to a source for $con(d,k), k \in \phi$ (otherwise, this path would have been found by Algorithm 6 and d would have been added to *candidates* or *discarded*). Let src be such a source. Because $prox^{\leq n}(u,src) = 0$, and due to the long path attenuation property on the score (Section 5.1.3), we know that $prox(u,src) \leq B^{>n}_{prox}$. Therefore, thanks to the fourth score feasibility property, $score(d,q) \leq B_{score}(q, B^{>n}_{prox})$. This is exactly the *threshold* value set by Algorithm 4 at line 6. □

We partition *candidates* into *groups* of documents. Two documents in *candidates* are in the same group if and only if *candidates* comprises a list of nodes, containing both documents, such that each node is a vertical neighbor of the next one in the list. Observe that in particular candidates that are not part of the same document are never in the same group. We call *score of a group* the highest score of its elements, and *lower bound of a group* the highest lower bound of its elements.

[5]The theorem also holds if $n = 0$, as $threshold = \infty$.

Lemma 5.2.3. *The scores of the k groups with the best scores can only increase, and only documents with a lower score than the score of the k-th group or having vertical neighbors in the k groups with the best scores can be in discarded.*

Proof. (Lemma 5.2.3)

Observe that the lemma's claim is true when *discarded* is empty. We show that it is also true when documents are added to *discarded* or when the k best groups change.

Unless a document's score is null and gets discarded by GetDocuments (line 10), a document can only go to *discarded* by leaving *candidates* due to CleanCandidatesList (algorithm 5).

The document with the best score in each group only has neighbors with lower scores than itself and therefore cannot be removed by lines 11-12. The only other way of discarding documents from candidates (lines 7-8) removes documents having an upper bound lower than the lower bound of the k-th group with the best lower bound. Because lower bounds are always smaller than the scores the best elements of the k groups with the best scores, they cannot be removed this way either. Therefore, at each call of CleanCandidatesList, each document with the best score in the k groups with the best scores remains in *candidates*, guaranteeing that the scores of the k best group can only increase and that adding new documents to *discarded* respects the lemma's claim.

As for the change of the k best groups, if a new group introduced in *candidates* removes the current k-th best group from its k-th position then the score of the new group is better than that of the removed one. Hence, documents which were in *discarded* because they were a neighbor of some element of the removed group, are now in *discarded* because they have a lower score than the new k-th best group (documents that had a lower score than the previous k best groups continue to do so with the new k ones). □

Theorem 5.2.1 (Stop correctness)**.** *When a stop condition is met, the first k elements in candidates are a query answer.*

Proof. (Theorem 5.2.1)

Let us show by induction on $i \in [0, k]$ that if the stop condition is met then the i documents with the best scores among the first k elements of *candidates* form a $T_{i,score}(q)$: a top-i answer to the query.

For $i = 0$, the 0 highest scored elements of the k first candidates is the empty set and we purposefully expand the notion of query answer such that: $T_{0,score}(q) = \emptyset$.

For $i > 0$, given that the $i - 1$ documents with the best scores among the first k elements of *candidates* form a $T_{i-1,score}(q)$, we have to show that adding a i-th document with the i-th best score among the first k elements of *candidates*, to this

$T_{i-1,score}(q)$ forms a $T_{i,score}(q)$. In order to do so, let d be a such a document. By the definition of query answer, we only have to show that d is in $D \setminus \texttt{neigh}(T_{k-1,score}(q))$ and has the maximum score ($score(d,q)$) on that set.

d is not in $\texttt{neigh}(T_{k-1,score}(q))$ since it follows from the stop condition that $\nexists d, d' \in candidates[0,\ldots,k-1]$, $d \in \texttt{neigh}(d')$ (lines 1-2) and from the induction hypothesis that $T_{k-1,score}(q) \subset candidates[0,\ldots,k-1]$. Therefore, d is in $D \setminus \texttt{neigh}(T_{k-1,score}(q))$.

Let us show now that d maximizes the score over $D \setminus \texttt{neigh}(T_{k-1,score}(q))$, by showing it does so over the intersection of $D \setminus \texttt{neigh}(T_{k-1,score}(q))$ with the following partition of D:

- the k first elements of *candidates*: by construction d has the best score among the k first elements of *candidates* which are not in $T_{k-1,score}(q)$
- the other elements of *candidates*: if d' is a document in *candidates*, which is not ranked among the first k, then:

  $d'.upper \leq candidates[k].upper$ because *candidates* is kept sorted by upper bound

  $candidates[k].upper \leq min_{c \in candidadtes[0,k-1]}(c.lower) \leq d.lower$ because of the stop condition (line 9)

  $score(d',q) \leq d'.upper$ and $d.lower \leq score(d,q)$ by bounds correctness (Lemma 5.2.1).

  Hence $score(d',q) \leq score(d,q)$.

- *discarded*: From Lemma 5.2.3, it follows that if d' is a document in *discarded* then either it has a lower score than d or it has a neighbor d" in *candidates* such that $score(d",q) \geq score(d',q)$. In the latter case, if this neighbor d" is not in $T_{k-1,score}(q)$ then $score(d",q) \leq score(d,q)$ and $score(d',q) \leq score(d,q)$. In any case there exists no d' in $discarded \setminus \texttt{neigh}(T_{k-1,score}(q))$ with a better score than d.
- $D \setminus (candidates \cup discarded)$: from the Threshold correctness (Lemma 5.2.2), we know that documents not in *candidates* nor in *discarded* have a score lower than *threshold*, thus the line 9 in Algorithm 2 and bounds correctness (Lemma 5.2.1) guarantee that $score(d,q) \geq threshold$.

We have shown that the k documents with the best score among the first k elements of *candidates*, i.e., the first k elements of *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 $B_{prox}^{>n} = 0$ for some n). Our generic score function (Definition 5.1.6) 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 5.2.2 (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.

Proof. (Theorem 5.2.2) If the algorithm terminates then it follows from Theorem 5.2.1 that the algorithm returns an answer. Now, to exhibit a contradiction, let us suppose that it does not terminate.

If there exists a query answer of size k then *candidates* will eventually contain at least k groups; otherwise some documents from this query answer would have gone to *discarded*, which by Lemma 5.2.3 is impossible if *candidates* doesn't contain at least k groups with better scores than these of the k documents from this query answer.

Because it is impossible to add new documents to *candidates* when all documents have been explored, after some time, the size of *candidates* will not change and no document will leave it anymore.

By Lemma 5.2.3, at least one document in each of the k best groups remains in *candidates*. Consider an element d with the best score in one the k best groups at a time where *candidates* has reached its final size. Because of the bound correctness and convergence lemma (Lemma 5.2.1), and the fact that all ties are breakable, eventually $\forall d' \in \texttt{neigh}(d)$ $d.lower \geq d'.upper$ holds. Therefore, the CleanCandidatesList algorithm, at lines 11-12, remove documents from *candidates* if any such d' is in candidates. Hence, d has no neighbors in *candidates* and is the only document in its group.

The k best groups are therefore k documents without neighbors in *candidates*, with the k best scores in *candidates*, and by bound convergence the best lower bounds eventually. This automatically triggers the stop condition and forces the algorithm to terminate, which contradicts our hypothesis.

We have shown that if there exists a query answer and all ties are breakable then the algorithm terminates and thus returns an answer (Theorem 5.2.1). □

Theorem 5.2.3 (Anytime correctness)**.** *Using anytime termination, Algorithm 1 eventually returns a query answer.*

Proof. (Theorem 5.2.3) Because (i) there is a finite number of documents that can be candidates and (ii) the simple convergence of bounds (Lemma 5.2.1) is actually a uniform convergence: for any query q and positive value ϵ, after some number of iterations, the lower and upper bounds of any candidate can differ from its score of at most ϵ.

If we choose for ϵ a value smaller than half the minimum positive difference between two document scores, then eventually any two candidates with different scores will have their upper bounds ordered in the same way as their score. In particular, after some time, argmax($c.upper$) = argmax($score(q, c)$) and therefore the anytime termination algorithm (Algorithm 7) produces a query answer. □

It is worth noting that in our experiments (Chapter 6), the threshold-based termination condition was always met, thus we never needed to wait for convergence of the lower and upper bound scores, in order to find the top-k answers.

5.3 Relationship with existing query models

Non-keyword based query models, for instance those inspired by XQuery or SPARQL can not be translated into $\texttt{S3}_k$keyword queries. In this section, we briefly show how the state of the art top-k keyword query models can be expressed. Depending on the dimensions considered in the original work, the scores of the literature can be translated as distances in our model.

Structural distance

Approaches based on the Lowest Common Ancestor can be emulated by our score model, this is the case for instance with the concrete score proposed in Section 5.1.4: the damping factor η ensures that, when two documents have the same connections' sources to the keywords of the query, then the one with the greater depth in the tree will have a better score. For instance, consider Figure 5.5 with the query $(u, \{k1, k2\})$: the connection between the nodes of the trees and the keyword $k2$ are all through the fragment $d_{0.0.0}$, and so are the connections with $k1$ all through the fragment $d_{0.0.1}$. Since

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

and $pos(d_{0.0}, d_{0.0.1}) = pos(d_{0.0}, d_{0.0.0}) = 1$ while $pos(d_0, d_{0.0.1}) = pos(d_0, d_{0.0.0}) = 2$ we have:

$$score(d_0, (u, \phi)) = score(d_{0.0}, (u, \phi)) \times \eta$$

with $\eta < 1$ and the Lowest Common Ancestor indeed has a better score.

Social distance

The proposed concrete score realizes naturally the Katz distance [48] if we ignore the semantic and structural aspects, that is if we consider that all documents have one node, containing all their keywords.

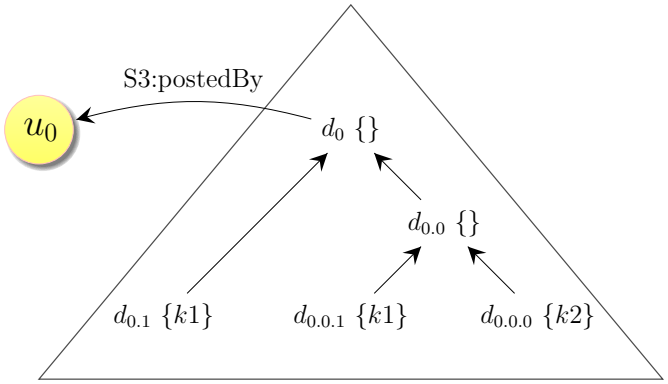

Figure 5.5: Sample structured document, posted by a user.

The other social distances presented in Section 3.1.5 are variations around the shortest path, i.e.,

$$max_{p \in u \rightsquigarrow d} \, \Pi_{edge \in p} \, weight(edge)$$

As we shown in Section 5.1.3, this is a social distance that respects the feasibility properties. Furthermore its ties are breakable since it only require a finite number of paths to give an exact score.

Semantics distance

By design we restricted our semantics distance to an all-or-nothing approach: URIs are either inside or outside of the extension of the keywords of the query, however the keywords used in the connections between the documents and the keywords of the query are taken into account in the final score via $\oplus_{gen}$:

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

Therefore it is possible to integrate into $\oplus_{gen}$ a factor that depends on kw. By adding similar keywords to query and restricting their extension it is possible to mimic approaches where keywords are given a similarity value in $[0, 1]$ that is used multiplicatively with the score.

5.4 Conclusion

This chapter provides a top-k keyword query model for `S3` instances, as well as a query answering algorithm which we demonstrate on a sample run and for

which we prove the termination and correctness. Furthermore we showed that the common top-k score functions on the three dimensions of S3 can be adapted to our query model.

Chapter 6

Implementation and evaluation

We now consider the implementation and evaluation of our algorithm over several scenario, on datasets extracted from real life social networks. Section 6.1.1 outlines our implementation and some practical optimizations we brought to the search algorithm. We present the datasets used to test this implementation in Section 6.2 and the query workloads in Section 6.3. We present a quantitative analysis on the query times in Section 6.4 and study the quality of our returned results in Section 6.5.

6.1 Implementation and optimisations

We describe our implementation in Section 6.1.1, the storage of `S3`instances in a database in Section 6.1.2 and the optimizations we have brought in Section 6.1.3.

6.1.1 Implementation

Our algorithms were fully implemented in Python 2.7. The first implementation represented the users and the connection between them and documents in memory using python objects and accessed the documents using a PostgreSQL database. While this provided a proof of concept, the runtimes were long and the scalability poor.

The second generation of implementation introduced sparse matrices, using the heavily optimized NumPy [3] library, to represent connections within I and proximity between the seeker and other vertices: this allows to hold the entirety of the connections in memory even for large instances, improving a lot on scalability and runtime. The second implementation also introduced an optimized version for the concrete score introduced in Section 5.1.4: while Algorithm 3 is supposed to keep track of all social paths starting from the seeker, to accommodate for all the

possible scores respecting the Feasibility properties, this not needed and is a waste of resources in most practical cases. In practice, it is often possible to aggregate all the proximities of the paths of a given length from the seeker to a node of I into a single scalar. Since Algorithm 6 starts from a document and test all documents d such that "there exists a chain of triples from x to d in I using only S3:partOf, S3:commentsOn, S3:$\overline{\text{commentsOn}}$, S3:hasSubject and S3:$\overline{\text{hasSubject}}$ labels" (line 1), it is possible to regroup the documents that share this property into connected components (connected via the above-mentioned properties in I) and test all the documents within a component at once.

Finally, the third-generation implementation incorporates parallelism: Algorithm 1 spends most of its runtime in a loop (lines 7-11) that increases n and launches *StopConditions*, *ExploreStep* (which in turn calls *GetDocuments*), *ComputeCandidatesBound* and *CleanCandidatesList*. However, it is in fact possible to *partially* execute several instances of this loop for different n in parallel.

Figure 6.1 outlines the main modules of this third-generation implementation:

- ExploreStep mimics Algorithm 3, *ExploreStep*; it explores paths one step further from the seeker, updating the proximities (*AllProx* and *BorderProx* that replace *BorderPath* in the version optimized for the concrete score) and the visited components. For this it requires the previous proximities, and the knowledge of the graph.

- GetDocuments follows Algorithm 6, *GetDocuments*: it looks at the newly visited components for unexplored documents and then based on the documents content and their sources (the last part requiring the graph connectivity), it checks if they have keywords from the keyword extension of the query. If they do, then it adds them to the candidates.

- ComputeBounds and CleanCandidates, as the name suggests, fuses the work of Algorithms 4 and 5, using the threshold and the candidates bounds (computed from the proximities of their sources with the seeker) to filter the list of candidate. It then updates the threshold by dividing it by γ.

- StopCondition computes if the spot condition is reached, as described by Algorithm 2, using the candidates, the threshold and the known and previously known components. When the stop condition is reached, the k best candidates are returned.

- The query is used to compute the first proximity (0 to everyone but the seeker) and the keyword extension based on the keywords of the query.

This last implementation, including both the optimized version for the concrete score and the capabilities to handle the generic scores, was refactored to a code

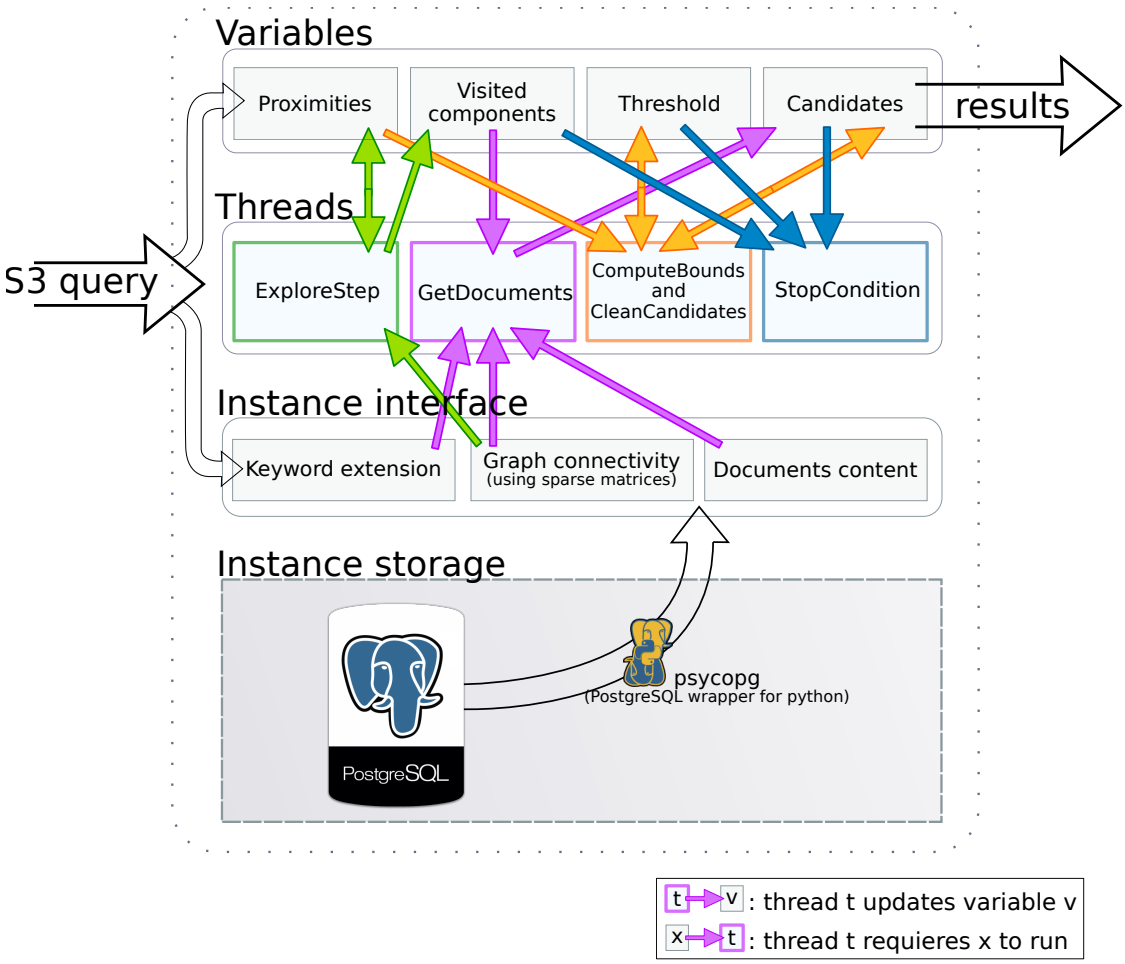

Figure 6.1: Overview of the implementation, and the thread dependencies

length of around **6K** lines of Python and PostgreSQL and is the one used in the rest of the manuscript, and referred to as the implementation.

We store some data tables in PostgreSQL 9.3, while others are 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.

Our system cannot be compared directly with existing systems, 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 of our implementation with other systems, we used the top-k social search system described in [58] and referred to as **TopkS** in Section 3.1.5. We used the Java-based code provided by the authors of [58]. As seen in Section 2.3, the data model of TopkS is rather basic, since its documents (*items*) have no internal structure nor semantics; tags are all independent, i.e., there is no semantic connection between them. Further, *(user,*

user, weight) tuples reflect weighted links between users. TopkS computes a social score for each item, and separately a content-based score; the overall item score is then obtained as

$$\alpha \times \text{social score} + (1 - \alpha) \times \text{content score}$$

where α is a parameter of TopkS.

6.1.2 Data Layout

The instance is stored on disk using PostgreSQL. We describe below our data layout; the underlined attributes denote a (clustering) primary key. For the keyword extension part we store the following relations:

- text_association(name, id), with an additional on name and another one in id;
- DBpediaURI_association(URI, id);
- keyword_extension(id, extended_id) with an additional index on id;

In the following, by the *components* of the S3 graph we mean the maximal subgraphs of documents and tags connected by triples labelled by S3:partOf, S3:commentsOn, S3:$\overline{\text{commentsOn}}$, S3:hasSubject, or S3:$\overline{\text{hasSubject}}$.

The documents and the connections they participate to are compiled into the following set of tables:

- network_edges(source, target, weight, type of relation), with an additional index on source;
- keyword_container(fragment or tag, component, keyword) stores the keywords contained in the documents and tags, grouped by component;
- structure(root, JSON structure) stores for each document tree the URI of the root of the tree and the position of the other URIs as a JSON expression;
- component(document, root of the document if exists, component), with an additional index on the component attribute; this table associates to each document the root of the document tree and the component it is in;
- raw_docs(URI, raw) associates documents with their URIs; this table is only used to return actual documents to the querier (it is not used in the query processing phase);

- original(URI, optional external identifier, type), where *type* is one of user, document root, fragment, or tag, with two additional indexes, on external identifier and on type. This table helps finding where the URIs in the database come from and what they are, in term sof representation outside the algorithm. It is not used in the query processing phase, but serves to connect it it with the application using it.

6.1.3 Optimisations

We briefly discuss our implementation, focusing on optimizations w.r.t. the conceptual description in Section 5.2.

The first optimization concerns the computation of $prox$, required for the score (Definition 5.1.6). While the score involves connections between documents and keywords found on any path, in practice $\texttt{S3}_k$ explores paths (and nodes) increasingly far from the seeker, and stores such paths in $borderPath$, which may grow very large (it can grows exponentially with n in the number of edges in I) 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:

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

and compute $prox$ directly based on this value. Note that $borderProx(v, n+1)$ can be computed from the $borderProx(x, n+1)$, $x \in \mathrm{I}$ because of the form of social distance: recall from the concrete score (Section 5.1.4) that

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

and therefore that

$$\begin{aligned} borderProx(v, n) =& C_\gamma \times \sum_{p \in u \rightsquigarrow v, |p|=n} \frac{\overrightarrow{prox}(p)}{\gamma^{|p|}} \\ =& C_\gamma \times \sum_{p' \in x \rightsquigarrow v, |p'|=1} \sum_{p \in u \rightsquigarrow x, |p|=n-1} \frac{\overrightarrow{prox}(p)}{\gamma^{|p|}} \\ =& \sum_{p' \in x \rightsquigarrow v, |p'|=1} borderProx(x, n-1) \end{aligned}$$

can be computed using only paths of length 1 and $borderProx$ at the previous iteration.

Furthermore, Algorithm `GetDocuments` considers documents reachable from x through edges labeled S3:partOf, S3:commentsOn, S3:$\overline{\text{commentsOn}}$, S3:hasSubject or S3:$\overline{\text{hasSubject}}$. Reachability by such edges defines a *partition* of the documents into *connected components*. Further, by construction of *con* tuples (Section 5.1.2), 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 a boolean vector indexed by the components, initially all false. Further, it creates (in memory) the $allProx$ vector and two sparse matrices which are computed only once: *distance matrix* that encodes the graph of network edges in I (accounting for the vertical neighborhood), and a *component* matrix storing the component of each fragment or tag. This simplifies Algorithm 3, since computing $allProx$ and finding new components to explore can be implemented using matrix and vector operations. For instance, the new distance vector $borderProx$ w.r.t. the seeker at step $n+1$ can be obtained by multiplying the distance matrix with the previous distance vector from step n.

Last but not least, the search algorithm can be *parallelized*, in two independent ways. First, within an iteration we discover new documents in parallel by splitting the search across components. Second, an iteration can start executing before the current one is finished: as long as $borderProx$ is available in the current iteration, one can start computing the next $borderProx$ using the (fixed) distance matrix. More precisely, `ExploreStep` can be seen as consisting of two main blocks:

- (i) computing the new $borderProx$ using the distance matrix and the previous $borderProx$ (lines 1- 16 except for line 14).
- (ii) computing $allProx$ using the new $borderProx$ and the previous $allProx$ (lines 17-20) plus the call to `GetDocuments` (line 14).

The latter algorithm can also be seen as consisting of two parts: (iii) identifying the newly discovered components (line 1), and (iv) testing the documents they contain (the remaining lines). In practice, slightly smaller grain separations are possible, leading to 8 tasks of the forms (i)-(iv) above. A custom scheduler is responsible for spawning and synchronize concurrent threads executing these tasks. This has divided the query answering time on average by a factor of 2 on the testing machine, but in a more parallel environment greater gain could be expected.

With all the optimization described before almost all the runtime of the algorithm is spent in optimized NumPy-based matrix operation or in the PostgreSQL query engine which are both hard to optimize further.

6.2 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, and based on the Tweepy library [91]. 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 create a document having three nodes:

- *text* from the *text* field of the tweet, we extract named entities and words (with the help of the twitter NLP tools library [65]) and match them against a general-purpose ontology we created from DBpedia, as we explain below
- *date*, extracted from the *created_at* tweet field
- a *geo* node: if the tweet includes a human readable location, i.e., recognizable keywords in the *place* field of the tweet (be it from a *name* or a *full_name* property) we insert it in this node.

The RDF graph of our instance is built from four DBPedia datasets, namely: Mapping-based Types, Mapping-based Properties, Persondata and DBpedia Lexicalizations Dataset. These were chosen as they were the most likely to contain concepts (names, entities etc.) occurring in tweets.

Within the *text* fields, we replace each word w for which a triple of the form u http://xmlns.com/foaf/0.1/name w holds in the DBPedia knowledge base, by the respective URI u.

When a tweet t' authored by an user u is a *retweet* of another tweet t, and further if t' adds a hashtag h, then we add to I_{d_1} the following triples:

a type S3:relatedTo a S3:hasSubject t
a S3:hasKeyword h a S3:hasAuthor u

Thus, a retweet that adds multiple keywords leads to the creating several tags. If a tweet t'' is a *reply* to another tweet t (as identified by the *in_reply_to_status_id* field), we consider that t'' is a comment on t. If t is present in our dataset[1], we add the corresponding S3:commentsOn triple in I_1.

For what concerns the social network, we set Ω_{I_1} as the set of IDs of the users having posted some tweets, and we create links between users as follows. We assign

[1]Sometimes our corpus contains a re-tweet of an original tweet that we did not capture. This is unavoidable unless one has access to the full Twitter history.

to every pair of users (a, b) a similarity value $u_\sim(a, b)$ between 0 and 1. $u_\sim$ is a weighted sum of two Jaccard similarity coefficients:

$$u_\sim(a,b) = t \times \frac{|\text{keywords in comments from both } a \text{ and } b|}{|\text{keywords in comments from } a| + |\text{keywords in comments fom } b|} + (1-t) \times \frac{|\text{keywords posted by both } a \text{ and } b|}{|\text{keywords posted by } a| + |\text{keywords posted by } b|}$$

Whenever this similarity is above a certain threshold we create an edge with weight $u_\sim$ between the two users. Based on experiments with the data, we set $t = 0.5$ and the $u_\sim(a, b)$ threshold for creating a link between a and b to 0.1.

The instance I_2 uses data from Vodkaster, a French social network dedicated to movies. The original social network data comprised *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 included u vdk:follow v 1 in I_2, where vdk:follow is a custom subproperty of S3:social expressing the act of following someone in Vodkaster (vdk:follow $\prec_{sp}$ S3:social). The first comment on a movie was translated in I_2 as an original document; each additional comment on this film was then considered a comment on the first. The textual content of each comment was stemmed using Snowball algorithms [90] and each (stemmed) sentence was made a fragment of the comment.

The instance I_3 is based on Yelp [93], a crowd-sourced reviews website about local businesses. This dataset contains a list of textual reviews of businesses, and the friend list of each user. As for I_2, we consider that the first review of a business is commented on by the subsequent reviews of the same business and we introduce a dedicated subproperty of S3:social: yelp:friend to express the friendship on Yelp: if user u is friend with user v then we include u yelp:friend v 1 in I_3. As for I_1, we use DBpedia to gain additional semantic on the extracted keywords.

Table 6.2 shows the main features of the three quite different data instances. I_1 is by far the largest. I_2 was not matched with a knowledge base since its content is in French; I_2 and I_3 have no tags.

6.3 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)

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 6.2: Statistics on our instances.

- 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 I_1, where $f \in \{+, -\}$, $l = 1$, and $k \in [1, 5, 10, 50]$ (used in Figure 6.6). *We stress here that injecting semantics in our workload queries, by means of keyword extensions (Definition 4.2.1), increased their size on average by* 50%.

We adapted our instances into TopkS's simpler data model. From I_1, we created I'_1 as follows:

- the relations between users were kept with their weight,
- every tweet was merged with all its retweets and replies into a single item,
- 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 I'_2 and I'_3:

- the relations between the users are kept with their weight,
- each film or business becomes an item ,
- each word extracted from a user's review on a item becomes a tuple in the (user, item, tag) relation.

6.4 Quantitative analysis

Figures 6.3 – 6.5 show the running times of $\texttt{S3}_k$ on our three instances. We used different values of the γ social proximity damping factor (Section 5.1.4), and of α for 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 γ in $\texttt{S3}_k$ reduces the running time. This is expected, as γ gives more weight to nodes far from the seeker, whose exploration is costly. Similarly,

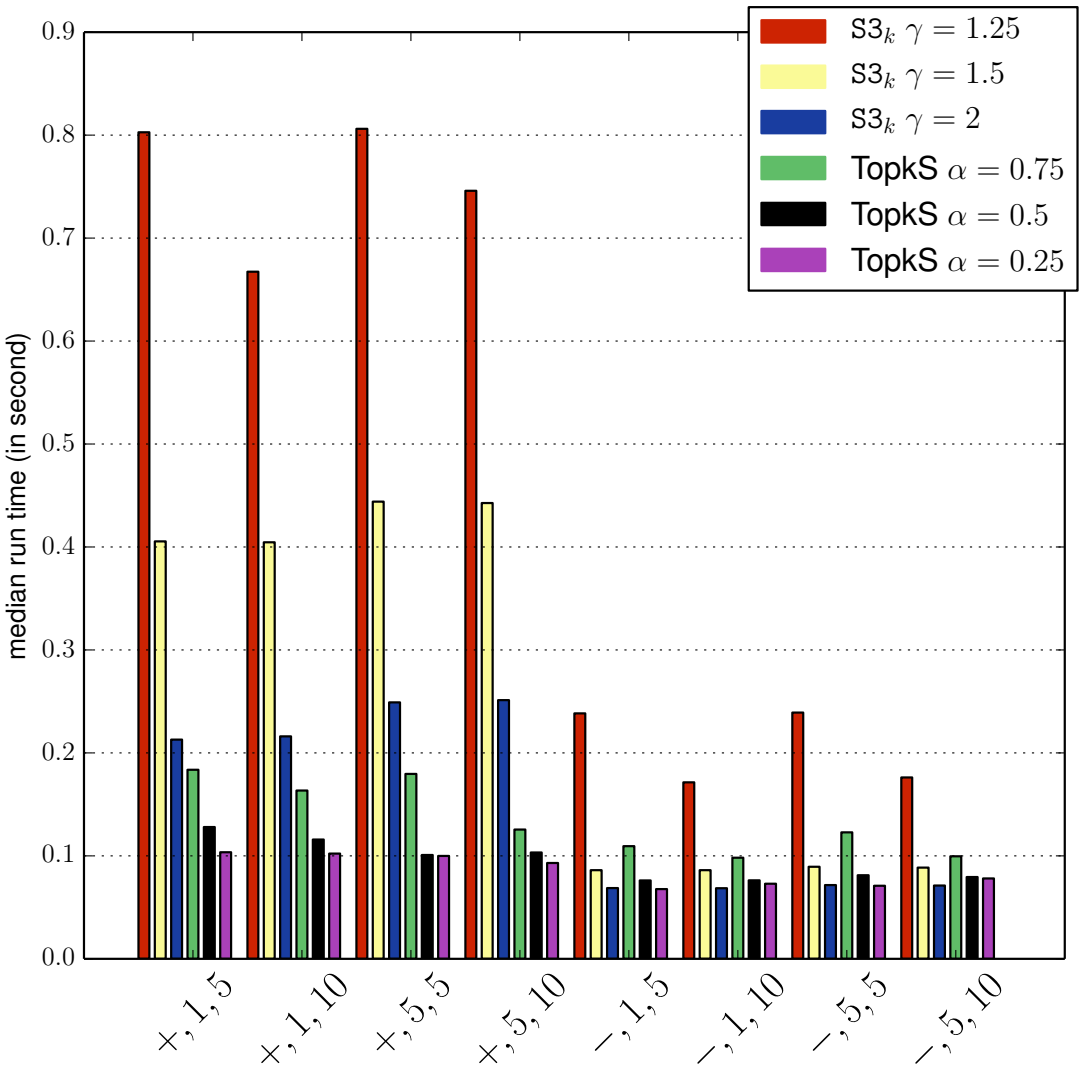

Figure 6.3: Query answering times on I_1 (Twitter).

increasing α 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 6.6 where, with frequent keywords, increasing k does not affect the 3 fastest quartiles but significantly slows

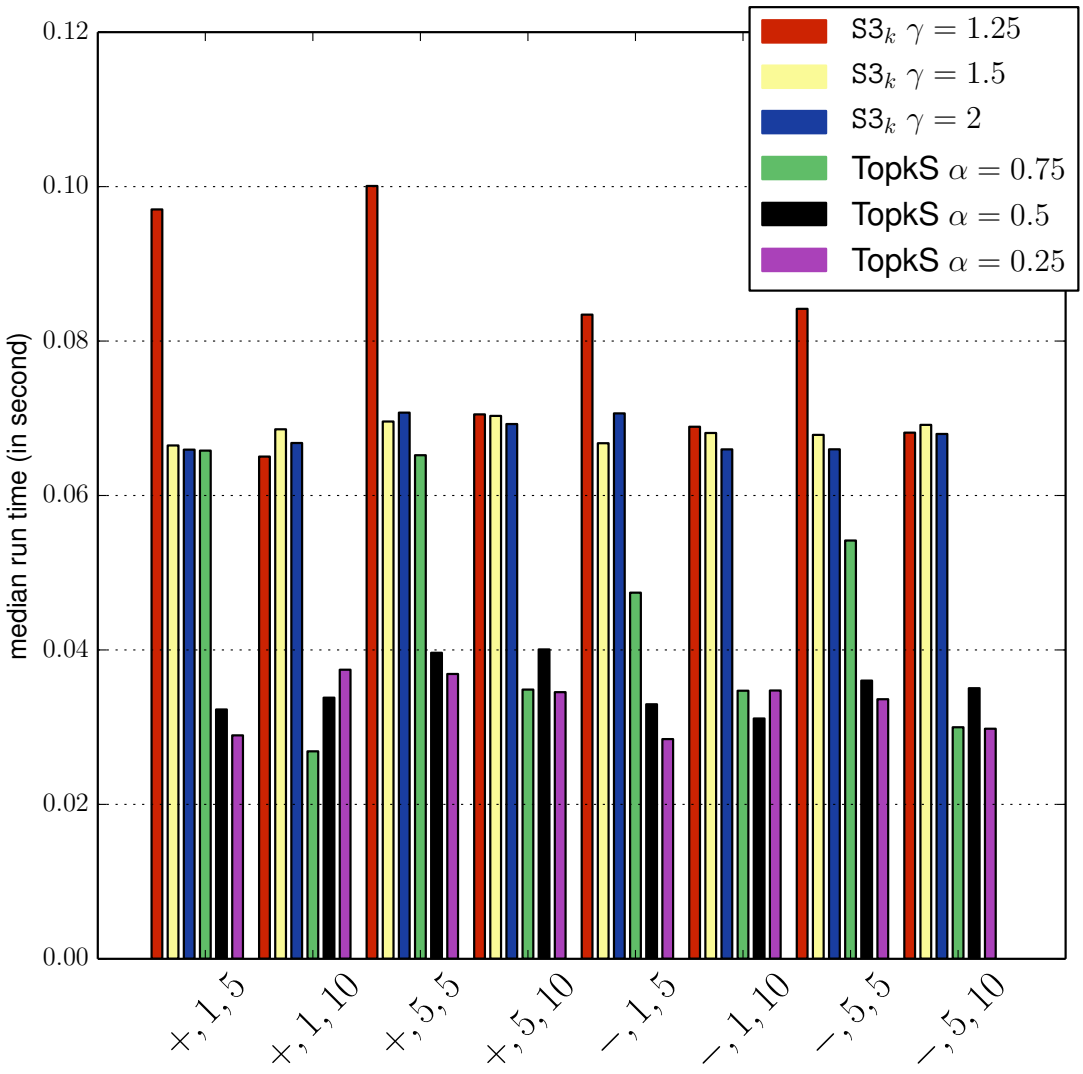

Figure 6.4: Query answering times on I_2 (Vodkaster).

down the slowest quartile, since the algorithm has to look further in the graph to refine the bounds.

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,

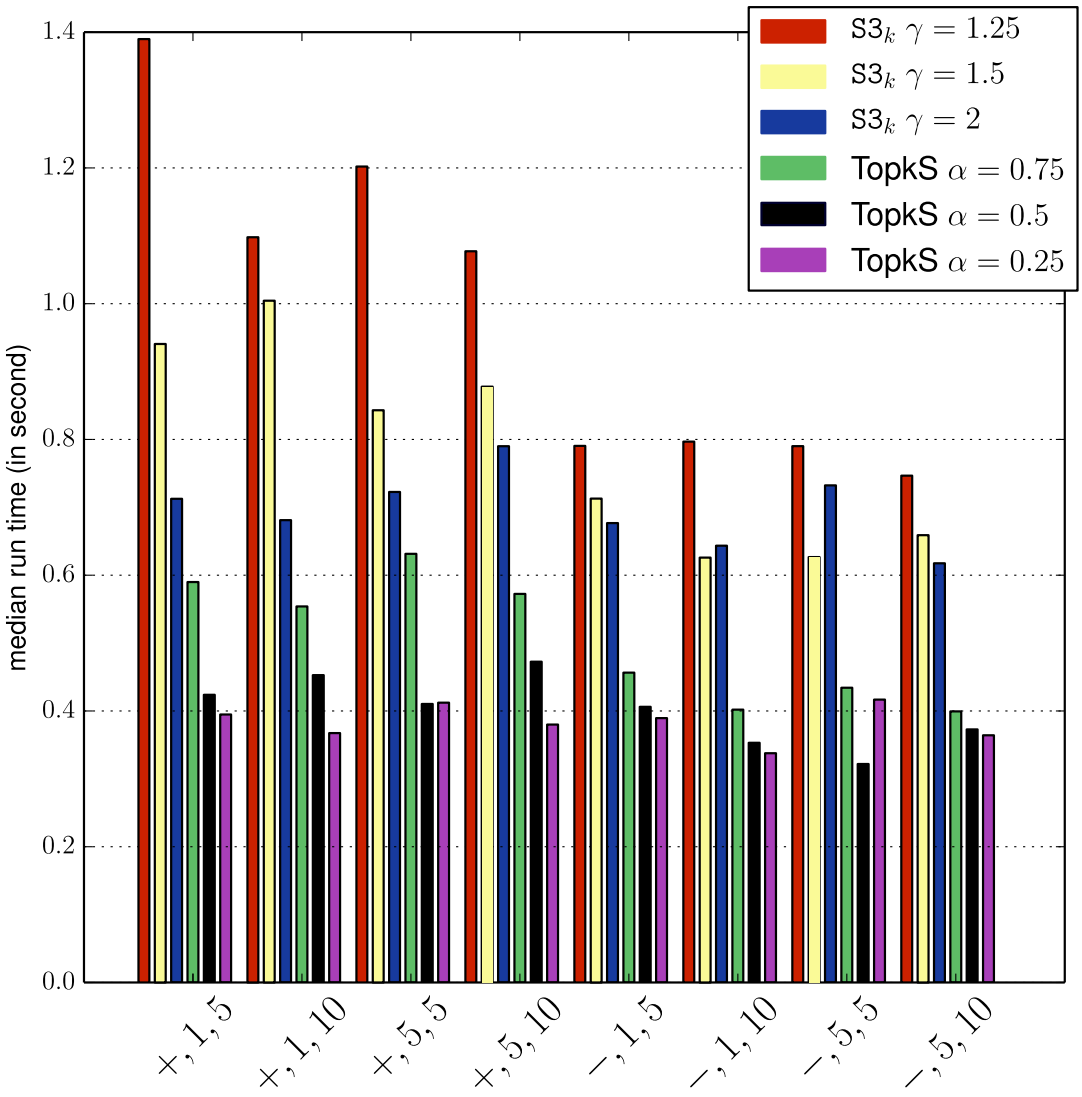

Figure 6.5: Query answering times on I_3 (Yelp).

more search and computations are needed before the top-k elements are discovered and ranked correctly.

6.5 Qualitative analysis

We compare now the answers of our S3$_k$ algorithm and those of TopkS from a *qualitative* angle. S3$_k$ follows links between documents to access further content, while TopkS does not; we term *graph reachability* the fraction of candidates reached

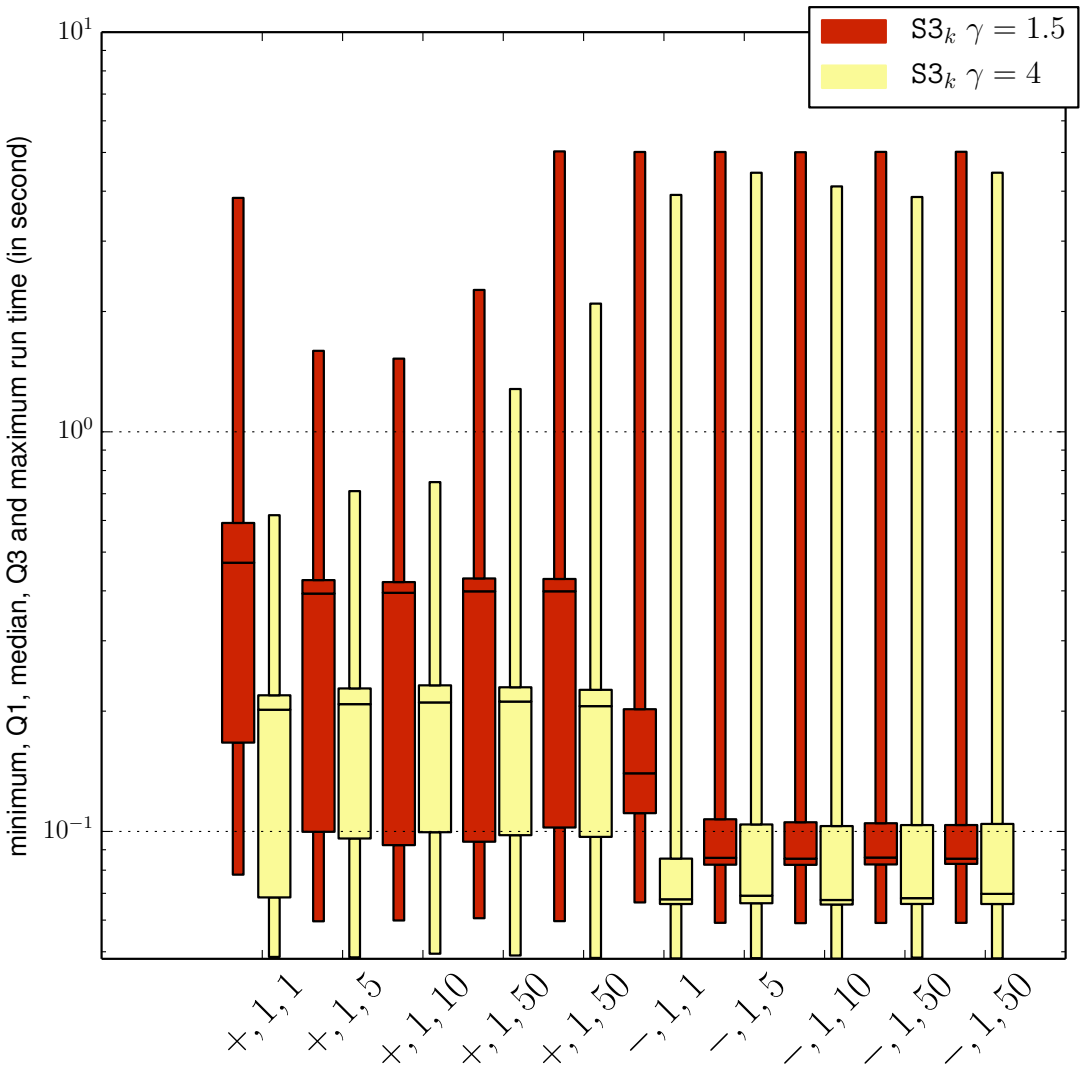

Figure 6.6: Query answering times on I_1 when varying k.

by our algorithm which are not reachable by the TopkS search. Further, while S3$_k$ takes into account semantics by means of semantic extension (Definition 4.2.1), 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 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 S3$_k$ results that

Measure \ Instance	I_1	I_2	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 6.7: Relations between the $\texttt{S3}_k$ and TopkS answers.

TopkS also returned. The second, L_1, is based on Spearman's well-known *foot rule* distance between lists [27]. Modulo normalisation to ensure that $L_1(\tau, \tau) = 1$, $L_1(\tau_1, \tau_2) = 0$ if they share no elements and 0.5 if they have the same elements in reverse order, L_1 is 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 τ_j.

The averages of these 4 measures over the 8 workloads on each instance appear in Figure 6.7. The ratios are low, and show that different candidates translate in different answers (the low L_1 stands witness for this). Few $\texttt{S3}_k$ 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.

TopkS instances contain much less items than their $\texttt{S3}_k$ counterpart contain documents. This is because all their structure and the social links between them is collapsed into a single item. For instance I'_1 contains approximately 11 times less items than I_1 contains documents and I'_2 43 times less than I_2, I'_3 34 times less than I_3. In particular TopkS cannot return a document that was fused with another, such as a reply to status or a retweet.

6.6 Conclusion

Our experiments have demonstrated, first, the ability of the 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 $\texttt{S3}_k$ *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 γ 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.

Chapter 7

Conclusion and perspectives

In this final chapter, we give, in Section 7.1, a summary of the work proposed through this thesis and we provide, in Section 7.2, a perspective on how this work can be integrated in further developments.

7.1 Conclusion

In the introduction, we have shown that social networks are a crucial area of research, impacting directly the lives of billions, and that more and more of our daily life and our environment is being incorporated into available social data. Keyword search is currently the most common way for users to access the wealth of this social data, and therefore a fundamental feature of social networks. And yet, top-k keyword search in a social context, as it can be found today, is severely limited because there exists no framework that utilizes the three main aspects of the data: its structure, semantics and social scope.

We have presented in Chapter 3 how the current approaches are integrating separately structure, semantics and social aspects into top-k search and into hybrid query systems. We highlighted their limitations and presented, in Chapter 4, a set of requirements for a data model integrating the three dimensions we study. We propose a new model, named `S3`, built to satisfy these requirements and overcome the limitations of the previous models. `S3` integrates the standards of the Web for semantics, as it contains directly RDF graphs, for structured documents, as it encodes easily XML, JSON, and more generally structured and semi-structured documents, and finally it integrates flexible social interactions encompassing the behaviours of most social networks' users. `S3` integrates the three dimensions we mentioned: it allows explicit and implicit relations between users, for instance through their interactions with documents, higher levels of tags and comments and a rich and well-defined semantics while remaining generic enough to be extensible.

S3 can be queried using a top-k, keyword based, query model: $\texttt{S3}_k$, introduced in Chapter 5. In this model, a user inputs a set of keywords and is returned the most relevant documents for these keywords, according to a customizable score function. The restriction on the possible scores are based on relatively simple and straightforward properties, such as that the score of a document increases with the proximity between its author and the seeker (the user making the query). These limitations are light enough that it is possible to emulate most other top-k approaches for the state of the art on their respective dimensions. We provided an algorithm realising $\texttt{S3}_k$, shown how it works through examples, and proved its termination and its correctness.

This algorithm is implemented and tested, in Chapter 6, on datasets extracted from real life social networks. On these datasets we identified the factors impacting the performances of $\texttt{S3}_k$and showed that it returns its results in a reasonable time and that captures significantly more results than a top-k system using only social data: this is a direct consequence of the harmonious integration of the structured, social, and semantic dimensions into S3.

7.2 Perspectives

The S3 model is generic and comprehensive, therefore it can be used in other settings than top-k keyword search. We present here another track of possible extension for S3 that remains to be fully explored: to be part of an integration model for data journalism.

Journalism is evolving to integrate more digital data as a support for investigation. Such data may come from organisations and governmental agencies publishing open data such as demographics, economics, stock quotes etc., or from social network websites providing social and textual data on brands, big and small businesses or public figures such as politicians and artists. There also exist large-scale ontologies integrating manual and automatic effort such as DBpedia, GeoNames or YAGO and other large sources of data may arise in particular cases, for instance the 11.5 million of documents of the Panama Papers. Data journalists are aware of the latent potential behind the usage of this data and would like to access it but lack the tools to do so.

To address this problem, team members are currently collaborating to develop Tatooine, a lightweight data integration prototype, Tatooine [16], devised based on our discussions with Les Decodeurs, a fact-checking team of the major french newspaper "Le Monde"[1]. Tatooine allows journalists to exploit heterogeneous data sources of different data models, which we view as a *mixed data instance*. Within the instance, we distinguish sets of structured, un-structured, or semi-

[1] lemonde.fr/les-decodeurs

structured data sources, of various data models, each of with its own query system. We present a mixed query engine as a mediator, running on top of the different data sources that we bridge using custom and application-dependent RDF data. For instance using the `S3` model, which is perfectly adapted to model social interactions with users and documents. The engine evaluates *mixed queries*, which combine sub-queries expressed in the query language(s) of several heterogeneous sources, and RDF querying on the custom data. A sample query is, for instance: *"for a given hashtag and each political affiliation (left wing, right wing etc.), find the most prolific tweet authors of that affiliation having used that hashtag, and their Facebook accounts"*. Mixed queries can be used also to query dynamically discovered datasets, e.g., the address of a relational database is found in an table and part of the mixed query is shipped there for evaluation.

Writing such queries requires database skills, therefore TATOOINE will also include a *keyword-based query engine*, working on data source digests computed from the data sources. Based on these digests, the keyword-based query engine identifies a set of mixed queries which, evaluated over the set of (joining) datasets, return the results users are interested in.

TATOOINE can be seen as an ad-hoc data integration platform (or mediator) extending the `S3` model to capture data of arbitrary models, and join them by means of commonly-occuring names and URIs. TATOOINE does not attempt to solve the (hard) problems related to the identification of same entities across different data sources; rather, it focuses on the ability to quickly set up integration applications on top of different datasets, taking advantage of the relative ease of named entity recognition in online media content. Work on TATOOINE is ongoing as part of the ANR ContentCheck project (2016-2019).

Bibliography

[1] The Javascript Object Notation (JSON) Data Interchange Format. `https://tools.ietf.org/html/rfc7159`.

[2] The JSON Data Interchange Format. `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`.

[3] NumPy. `http://www.numpy.org/`.

[4] Three and a half degrees of separation. `https://research.facebook.com/blog/three-and-a-half-degrees-of-separation/`.

[5] Twitter Developers. `https://dev.twitter.com/`.

[6] Yelp! Dataset Challenge. `https://www.yelp.com/dataset_challenge`.

[7] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[8] M Albakour, Romain Deveaud, Craig Macdonald, Iadh Ounis, et al. Diversifying contextual suggestions from location-based social networks. In *Proceedings of the 5th Information Interaction in Context Symposium*, 2014.

[9] Bernd Amann, Irini Fundulaki, Michel Scholl, Catriel Beeri, and Anne marie Vercoustre. Mapping XML Fragments to Community Web Ontologies. In *WebDB*, 2001.

[10] Ioannis Antonellis, Hector Garcia-Molina, and Jawed Karim. Tagging with queries: How and why? 2008.

[11] Fabio A Asnicar and Carlo Tasso. ifWeb: a Prototype of User Model-Based Intelligent Agent for Document Filtering and Navigation in the World Wide Web. In *Sixth International Conference on User Modeling*, 1997.

[12] Akanksha Baid, Ian Rae, Jiexing Li, AnHai Doan, and Jeffrey Naughton. Toward scalable keyword search over relational data. *VLDB*, 2010.

[13] Andrey Balmin, Vagelis Hristidis, and Yannis Papakonstantinou. Objectrank: Authority-based keyword search in databases. In *PVLDB*, 2004.

[14] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and Shashank Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, 2002.

[15] Stefan Bischof, Stefan Decker, Thomas Krennwallner, Nuno Lopes, and Axel Polleres. Mapping between RDF and XML with XSPARQL. *Journal on Data Semantics*, 2012.

[16] Raphaël Bonaque, Tien-Duc Cao, Bogdan Cautis, François Goasdoué, Javier Letelier, Ioana Manolescu, Oscar Mendoza, Swen Ribeiro, Xavier Tannier, and Michaël Thomazo. Mixed-instance querying: a lightweight integration architecture for data journalism. *VLDB*, 2016.

[17] Andrei Z Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *Proceedings of the twelfth international conference on Information and knowledge management*, 2003.

[18] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: How Facebook serves the social graph. *SIGMOD*, 2012.

[19] David Carmel, Naama Zwerdling, Ido Guy, Shila Ofek-Koifman, Nadav Har'El, Inbal Ronen, Erel Uziel, Sivan Yogev, and Sergey Chernov. Personalized social search based on the user's social network. In *CIKM*, 2009.

[20] Liang Jeff Chen and Yannis Papakonstantinou. Supporting top-k keyword search in XML databases. In *ICDE*, 2010.

[21] Liren Chen and Katia Sycara. Webmate: a personal agent for browsing and searching. In *Proceedings of the second international conference on Autonomous agents*, pages 132–139. ACM, 1998.

[22] Vassilis Christophides, Sophie Cluet, and Jérome Simèon. On wrapping query languages and efficient XML integration. *SIGMOD*, 2000.

[23] Sara Cohen, Benny Kimelfeld, Georgia Koutrika, and Jan Vondrák. On principles of egocentric person search in social networks. In *VLDS*, 2011.

[24] Michael Curtiss, Iain Becker, Tudor Bosman, Sergey Doroshenko, Lucian Grijincu, Tom Jackson, Sandhya Kunnatur, Soren Lassen, Philip Pronin, Sriram

Sankar, et al. Unicorn: A system for searching the social graph. *PVLDB*, 2013.

[25] Stefan Decker, Michael Erdmann, Dieter Fensel, and Rudi Studer. *Ontobroker: Ontology based access to distributed and semi-structured information*. 1999.

[26] Matthias Droop, Markus Flarer, Jinghua Groppe, Sven Groppe, Volker Linnemann, Jakob Pinggera, Florian Santner, Michael Schier, Felix Schöpf, Hannes Staffler, et al. Translating XPath queries into SPARQL queries. In *On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops*.

[27] Ronald Fagin, Ravi Kumar, and D Sivakumar. Comparing top k lists. *SIAM Journal on Discrete Mathematics*, 2003.

[28] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4), 2003.

[29] Christiane Fellbaum. *WordNet*. Wiley Online Library.

[30] Matthias Ferdinand, Christian Zirpins, and David Trastour. Lifting XML schema to OWL. In *Web Engineering*. 2004.

[31] Venkatesh Ganti, Yeye He, and Dong Xin. Keyword++: A framework to improve keyword search over entity databases. *Proceedings of the VLDB Endowment*, 2010.

[32] Periklis Georgiadis, Ioannis Kapantaidakis, Vassilis Christophides, Elhadji Nguer, and Nicolas Spyratos. Efficient rewriting algorithms for preference queries. In *ICDE*, 2008.

[33] Giorgos Giannopoulos, Evmorfia Biliri, and Timos Sellis. Personalizing keyword search on RDF data. In *Research and Advanced Technology for Digital Libraries*. 2013.

[34] François Goasdoué, Ioana Manolescu, and Alexandra Roatiş. Efficient query answering against dynamic RDF databases. In *EDBT*, 2013.

[35] François Goasdoué, Konstantinos Karanasos, Yannis Katsis, Julien Leblay, Ioana Manolescu, and Stamatis Zampetakis. Growing triples on trees: an XML-RDF hybrid model for annotated documents. *VLDB Journal*, 2013.

[36] Gang Gou and Rada Chirkova. Efficient algorithms for exact ranked twig-pattern matching over graphs. In *SIGMOD*, 2008.

[37] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. XRANK: Ranked keyword search over XML documents. In *SIGMOD*, 2003.

[38] Siegfried Handschuh and Steffen Staab. Authoring and annotation of web pages in CREAM. In *WWW*, 2002.

[39] Donna Harman. Relevance feedback revisited. In *SIGIR*, 1992.

[40] Hao He, Haixun Wang, Jun Yang, and Philip S Yu. Blinks: ranked keyword searches on graphs. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, 2007.

[41] Damon Horowitz and Sepandar D Kamvar. The anatomy of a large-scale social search engine. In *WWW*, 2010.

[42] Vagelis Hristidis, Luis Gravano, and Yannis Papakonstantinou. Efficient IR-style keyword search over relational databases. In *VLDB*, 2003.

[43] Vagelis Hristidis, Heasoo Hwang, and Yannis Papakonstantinou. Authority-based keyword search in databases. *ACM Transactions on Database Systems (TODS)*, 2008.

[44] Vagelis Hristidis, Nick Koudas, Yannis Papakonstantinou, and Divesh Srivastava. Keyword proximity search in XML trees. *Knowledge and Data Engineering, IEEE Transactions on*, 2006.

[45] Glen Jeh and Jennifer Widom. Scaling personalized web search. In *WWW*, 2003.

[46] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional expansion for keyword search on graph databases. In *PVLDB*, 2005.

[47] Sepandar D Kamvar, Mario T Schlosser, and Hector Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. In *WWW*, 2003.

[48] Leo Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1):39–43, 1953.

[49] Georgia Koutrika and Yannis Ioannidis. Personalized queries under a generalized preference model. In *ICDE*, 2005.

[50] Chengkai Li, Kevin Chen-Chuan Chang, Ihab F Ilyas, and Sumin Song. RankSQL: query algebra and optimization for relational top-k queries. In *SIGMOD*. ACM, 2005.

[51] Guoliang Li, Shengyue Ji, Chen Li, and Jianhua Feng. Efficient type-ahead search on relational data: a tastier approach. In *SIGMOD*, 2009.

[52] Jianxin Li, Chengfei Liu, Rui Zhou, and Wei Wang. Top-k keyword search over probabilistic XML data. In *ICDE*, 2011.

[53] Yuchen Li, Zhifeng Bao, Guoliang Li, and Kian-Lee Tan. Real time personalized search on social networks. In *ICDE*, 2015.

[54] Yunyao Li, Cong Yu, and HV Jagadish. Schema-Free XQuery. In *PVLDB*, 2004.

[55] Chengfei Liu, Liang Yao, Jianxin Li, Rui Zhou, and Zhenying He. Finding smallest k-compact tree set for keyword queries on graphs using mapreduce. *WWW*, 2015.

[56] Michael Luca. Reviews, reputation, and revenue: The case of yelp. com. *Com. Harvard Business School NOM Unit Working Paper*, 2011.

[57] Silviu Maniu and Bogdan Cautis. Efficient top-k retrieval in online social tagging networks. *arXiv preprint arXiv:1104.1605*, 2011.

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

[59] Alessandro Micarelli and Filippo Sciarrone. Anatomy and empirical evaluation of an adaptive web-based information filtering system. *User Modeling and User-Adapted Interaction*, 2004.

[60] Saikat Mukherjee, Guizhen Yang, and IV Ramakrishnan. Automatic annotation of content-rich html documents: Structural and semantic analysis. In *ISWC*. 2003.

[61] Patrick E. O'Neil, Elizabeth J. O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. Ordpaths: Insert-friendly XML node labels. In *SIGMOD*, 2004.

[62] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: bringing order to the web. 1999.

[63] Josiane Xavier Parreira and Gerhard Weikum. Jxp: Global authority scores in a p2p network. In *WebDB*, 2005.

[64] Lu Qin, Jeffrey Xu Yu, and Lijun Chang. Ten thousand SQLs: parallel keyword queries computing. *VLDB*, 2010.

[65] Alan Ritter, Sam Clark, Mausam, and Oren Etzioni. Named entity recognition in tweets: An experimental study. In *EMNLP*, 2011.

[66] Jonathan Robie, Lars Marius Garshol, Steve Newcomb, Michel Biezunski, Matthew Fuchs, Libby Miller, Dan Brickley, Vassillis Christophides, and Gregorius Karvounarakis. The syntactic web. *Markup Lang.*, 2001.

[67] Takeshi Sakaki, Makoto Okazaki, and Yutaka Matsuo. Earthquake shakes twitter users: real-time event detection by social sensors. In *Proceedings of the 19th international conference on World wide web*, pages 851–860, 2010.

[68] Karthikeyan Sankaralingam, Madhulika Yalamanchi, Simha Sethumadhavan, and James C Browne. Pagerank computation and keyword search on distributed systems and p2p networks. *Journal of Grid Computing*, 2003.

[69] Barry Smyth and Evelyn Balfe. Anonymous personalization in collaborative web search. *Information retrieval*, 2006.

[70] Micro Speretta and Susan Gauch. Personalized search based on user search histories. In *Web Intelligence, 2005. Proceedings. The 2005 IEEE/WIC/ACM International Conference on*, 2005.

[71] Kostas Stefanidis, Marina Drosou, and Evaggelia Pitoura. Perk: personalized keyword search in relational databases through preferences. In *Proceedings of the 13th International Conference on Extending Database Technology*, 2010.

[72] Igor Tatarinov, Stratis D Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD*, 2002.

[73] Jaime Teevan, Susan T Dumais, and Eric Horvitz. Personalizing search via automated analysis of interests and activities. In *SIGIR*, 2005.

[74] Martin Theobald, Holger Bast, Debapriyo Majumdar, Ralf Schenkel, and Gerhard Weikum. TopX: efficient and versatile top-k query processing for semistructured data. *VLDB*, 2008.

[75] Thanh Tran, Haofen Wang, Sebastian Rudolph, and Philipp Cimiano. Top-k exploration of query candidates for efficient keyword search on graph-shaped (RDF) data. In *ICDE*, 2009.

[76] Howard Turtle and James Flood. Query evaluation: strategies and optimizations. *Information Processing & Management*, 1995.

[77] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. The anatomy of the facebook social graph. *CoRR*, 2011.

[78] Jacopo Urbani, Spyros Kotoulas, Jason Maassen, Frank van Harmelen, and Henri E. Bal. WebPIE: A web-scale parallel inference engine using MapReduce. *J. Web Sem.*, 10, 2012.

[79] Erik Vee, Utkarsh Srivastava, Jayavel Shanmugasundaram, Prashant Bhat, and Sihem Amer Yahia. Efficient computation of diverse query results. In *ICDE*, 2008.

[80] W3C. SPARQL protocol and RDF query language. `https://www.w3.org/TR/rdf-sparql-query`.

[81] W3C. W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures. `https://www.w3.org/TR/xmlschema11-1/`.

[82] W3C. W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes. `https://www.w3.org/TR/xmlschema11-2/`.

[83] W3C. W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes, Primitive Datatypes. `https://www.w3.org/TR/xmlschema-2/#built-in-primitive-datatypes`.

[84] Resource Description Framework. `https://www.w3.org/RDF`.

[85] Extensible Markup Language (XML). `https://www.w3.org/XML`.

[86] Extensible Markup Language (XML) Prolog and Document Type Declaration. `https://www.w3.org/TR/xml/#sec-prolog-dtd`.

[87] XML Path Language. `https://www.w3.org/TR/xpath`.

[88] XQuery 3.1: An XML Query Language. `https://www.w3.org/TR/xquery-3/`.

[89] RDFa Primer. `https://www.w3.org/TR/xhtml-rdfa-primer`.

[90] Snowball stemming library for python. `https://pypi.python.org/pypi/PyStemmer`.

[91] Tweepy library. `http://www.tweepy.org/`.

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

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

[94] The original proposal of the WWW, htmlized. `https://www.w3.org/History/1989/proposal.html`, March 1989.

[95] Yu Xu and Yannis Papakonstantinou. Efficient keyword search for smallest LCAs in XML databases. In *SIGMOD*, 2005.

[96] Sihem Amer Yahia, Michael Benedikt, Laks VS Lakshmanan, and Julia Stoyanovich. Efficient network aware search in collaborative tagging sites. *PVLDB*, 2008.

[97] Zi-Ke Zhang, Tao Zhou, and Yi-Cheng Zhang. Personalized recommendation via integrated diffusion on user–item–tag tripartite graphs. *Physica A: Statistical Mechanics and its Applications*, 2010.

[98] Rui Zhou, Chengfei Liu, and Jianxin Li. Fast ELCA computation for keyword queries on XML data. In *Proceedings of the 13th International Conference on Extending Database Technology*, 2010.