Effective Searching of RDF Knowledge Bases

Dissertation
zur Erlangung des Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultät I
der Universität des Saarlandes

Shady Elbassuoni
Max-Planck-Institut für Informatik

Saarbrücken
21.02.2012

Dekan der Naturwissenschaftlich-Technischen Fakultät I	Prof. Dr.-Ing. Holger Herrmanns
Vorsitzender der Prüfungskommission	Prof. Dr. Jens Dittrich
Berichterstatter	Prof. Dr.-Ing. Gerhard Weikum
Berichterstatter	Prof. Dr. Maya Ramanath
Berichterstatter	Prof. Dr.-Ing. Wolfgang Nejdl
Beisitzer	Dr.-Ing. Ralf Schenkel
Tag des Promotionskollquiums	06.02.2012

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe.

Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.

Die Arbeit wurde bisher weder im In- noch im Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Saarbrücken, den 21.02.2012

(Shady Elbassuoni)

To my mum, Alia Hanem Abdel-Ghaffar

Acknowledgment
I would like to express my deepest gratitude to my advisor, Prof. Dr.-Ing. Gerhard Weikum. I moved to Saarbrücken six years ago without much prior knowledge about information retrieval. After attending an IR class taught by Gerhard, I instantly fell in love with this amazing field of science. Since then, Gerhard has been always a scientific reference I relied on to tackle many of the problems I faced throughout the course of my studies. I would like to warmly thank him for being extremely patient with me, supportive, and more importantly for his immense scientific contributions in most of the work I have done throughout the thesis, which would not have been at all possible without his excellent and friendly guidance. I would like to also thank Prof. Dr.-Ing. Wolfgang Nejdl for reviewing the thesis.

I am indebted to my co-advisor Dr. Maya Ramanath. Maya has initiated many of the ideas that we carried out in this thesis and was involved in most of the details of the work. I particularly thank her for always being ambitious and determined, especially when it comes to submission deadlines. I would like to also thank my colleagues whom I worked with on many of the publications that I published throughout the course of my thesis. These include Julia Luxenburger, with whom I first learnt how to do proper research. Even though the output of our work together was not included in this thesis, I am still very proud of it and I believe it gave me a first-hand experience on how to carry out scientific research. I am also thankful to Gjergji Kasneci, Nicoleta Preda, Ralf Schenkel, Steffen Metzger, and Katja Hose, who also was very generous to help me translate the abstract and summary of this thesis into German. I would like to also thank both Roi Blanco and Hugo Zaragoza for their guidance and support during my internship at Yahoo! Research, Barcelona.

I am very thankful to the International Max-Planck Research School and Microsoft Research Lab, Cambridge for financially supporting my studies. Finally, I truly thank all my friends in MPI and Saarbrücken (particularly my office mate Dimitar), with whom I had stimulating discussions and with whom I made great memories that would last forever. I also thank my BAConfidential friends (especially Nazlu) who were always there for me through the good times and the bad times. Last but not least, I would like to thank my family and in particular my dad and my brother Khaled who supported me both morally and financially. Hanging in there these past six years could not have been possible without them.

Abstract

RDF data has become a vital source of information for many applications. In this thesis, we present a set of models and algorithms to effectively search large RDF knowledge bases. These knowledge bases contain a large set of subject-predicate-object (SPO) triples where subjects and objects are entities and predicates express relationships between them. Searching such knowledge bases can be done using the W3C-endorsed SPARQL language or by similarly designed triple-pattern search. However, the exact-match semantics of triple-pattern search might fall short of satisfying the users needs by returning too many or too few results. Thus, IR-style searching and ranking techniques are crucial.

This thesis develops models and algorithms to enhance triple-pattern search. We propose a keyword extension to triple-pattern search that allows users to augment triple-pattern queries with keyword conditions. To improve the recall of triple-pattern search, we present a framework to automatically reformulate triple-pattern queries in such a way that the intention of the original user query is preserved while returning a sufficient number of ranked results. For efficient query processing, we present a set of top-k query processing algorithms and for ease of use, we develop methods for plain keyword search over RDF knowledge bases. Finally, we propose a set of techniques to diversify query results and we present several methods to allow users to interactively explore RDF knowledge bases to find additional contextual information about their query results.

Kurzfassung

Eine Vielzahl aktueller Anwendungen basiert auf RDF-Daten als essentieller Informationsquelle. Daher sind Modelle und Algorithmen zur effizienten Suche in RDF-Wissensdatenbanken ein entscheidender Aspekt, der über Erfolg und Nichterfolg entscheidet. Derartige Datenbanken bestehen aus einer großen Menge von Subjekt-Prädikat-Objekt-Tripeln (SPO-Tripeln), wobei Subjekt und Objekt Entitäten darstellen und Prädikate Beziehungen zwischen diesen Entitäten beschreiben. Suchanfragen werden in der Regel durch Verwendung des W3C Anfragestandards SPARQL oder ähnlich strukturierte Anfragesprachen formuliert und basieren auf Tripel-Patterns. Werden nur exakte Treffer in die Ergebnismenge übernommen, wird das Informationsbedürfnis des Nutzers häufig nicht befriedigt, wenn zu wenige oder zu viele Ergebnisse ausgegeben werden. Techniken, die ihren Ursprung im Information-Retrieval haben, sowie ein geeignetes Ranking können diesem Problem entgegenwirken.

Diese Dissertation stellt daher Modelle und Algorithmen zur Verbesserung der Suche basierend auf Tripel-Patterns vor. Die im Rahmen der Dissertation erarbeitete Strategie zur Lösung der oben geschilderten Problematik basiert auf der Idee, die Tripel-Patterns einer Anfrage durch Schlüsselwörter zu erweitern. Um den Recall dieser Suchvariante zu verbessern, wird ein Framework vorgestellt, welches die vom Nutzer übergebenen Anfragen automatisch in einer Weise umformuliert, dass die Intention der ursprünglichen Nutzeranfrage erhalten bleibt und eine ausreichende Anzahl an sortierten Ergebnissen ausgegeben wird. Um derartige Anfragen effizient bearbeiten zu können, werden Top-k Algorithmen und Methoden zur Schlüsselwortsuche auf RDF-Datenbanken vorgestellt. Schließlich werden einige Methoden zur Diversifikation der Anfrageergebnisse präsentiert sowie einige Ansätze vorgestellt, die es Benutzern erlauben, RDF-Datenbanken interaktiv zu explorieren und so zusätzliche Kontextinformationen zu den Anfrageergebnissen zu erhalten.

Summary

The Semantic-Web data model RDF (Resource Description Framework) has gained popularity in many domains as a representation format for heterogeneous structured data on the Web. In addition, the growing popularity of knowledge-sharing communities such as Wikipedia and the advances in automatic information-extraction have contributed to the presence of large general-purpose RDF knowledge bases.

RDF knowledge bases consist of subject-property-object (SPO) triples, where subjects and objects are generally entities and predicates represent relationships between entities. RDF knowledge bases are rich information sources that can be leveraged to quickly and precisely find answers to advanced informational queries. This is typically done by means of expressive triple-pattern queries, such as the queries written in the W3C-endorsed SPARQL language. However, in order to truly utilize such new information-retrieval framework and to deploy it on a Web scale, many challenging research problems must be addressed. This thesis presents solutions to key aspects of these problems as follows.

- **Data Incompleteness:** While large RDF knowledge bases contain a vast amount of information in the form of SPO triples, the majority of information on the Web is available in the form of free text. Thus, combining RDF with text can increase the scope of such knowledge bases making them very rich sources of information. In this thesis, we show how to augment traditional RDF knowledge bases with text to extend their scope of coverage, and we propose an extension to triple-pattern search that allows users to augment triple-pattern queries with keywords to allow them to express a wider range of information needs.

- **Result Ranking:** Large RDF knowledge bases may contain noisy or incorrect information and thus queries may produce many results of highly varying quality. It is thus highly desirable to present users with a ranked list of results rather than just a set of unranked matches. Moreover, when keywords are expressed in a triple-pattern query, result ranking is crucial to ensure that query results that are relevant to the keyword conditions are ranked on top. To address this, we develop a ranking model based on statistical language models for ranking the results to triple-pattern queries. Our ranking model is general enough and handles both cases of triple-pattern queries only and keyword-augmented triple-pattern queries.

- **Approximate Matching:** Even though triple-pattern queries are highly expressive, especially when augmented with keywords, they are also very restrictive since they deploy Boolean matching (i.e., a result is either a match to a query or not). By allowing approximate matching for queries with very few or no results, the recall of such queries can be highly improved. To do this, we develop a framework for automatic query reformulation that generates a set of reformulated queries that are close in spirit to a given triple-pattern query. Moreover, we extend our ranking model for triple-pattern queries and show how it can be used to merge and rank the results of the original query and all its reformulations.

- **Efficient Query Processing:** Triple-pattern search over RDF knowledge bases involves pattern matching. This becomes in particular very expensive when keyword conditions are allowed and when automatic query reformulation is supported. Moreover, result ranking adds an additional level of complexity. Incremental retrieval and ranking of results is thus needed to improve the response time of such queries. We develop a framework for efficient top-k triple-pattern query processing that also handles the cases of keyword-augmented triple-pattern queries and automatic query reformulation.

- **Keyword Search:** Triple-pattern search, even when augmented with keywords, is still best targeted for expert users or programming APIs. Casual users are accustomed to keyword search which is the paradigm to search for information on the Web. To increase the usability of RDF knowledge bases, we propose a framework for plain keyword search over RDF knowledge bases, where result ranking is again based on statistical language models.

- **Result Diversity:** While ranking ensures that the most relevant results are ranked on top, it is often the case that the top results tend to be homogeneous, making it difficult for users interested in less popular aspects to find relevant results. Thus, result diversity can play a big role in ensuring that the users get a broad view of the different aspects of the results matching their queries, and ensures that, on average, almost all users can find relevant results to their queries in the top ranks. We provide a notion of diversity for results to queries over RDF knowledge bases and develop a general framework that can be used to provide diverse top-k query results.

- **Knowledge Exploration:** While the results to queries over RDF knowledge bases provide very concise answers to users' information needs, it is often the case that users like to explore the knowledge base in order to learn more about a certain topic or subject. It is thus necessary to provide tools to interactively explore RDF knowledge bases. We present two systems to allow users to explore RDF knowledge bases and to combine the information there with information retrieved from external sources. The first system is a document retrieval system that retrieves a list of ranked documents given a set of RDF triples. The second system is an entity-summarization system that constructs a comprehensive timeline summarization for a given entity of interest.

Zusammenfassung

Als Semantic-Web-Datenmodell hat das Resource-Description-Framework (RDF) in vielen Bereichen zur Darstellung heterogen strukturierter Daten im Web an Bedeutung gewonnen. Darüber hinaus haben die Popularität von Systemen wie Wikipedia sowie Fortschritte im Bereich der automatischen Informationsextraktion zur Entstehung von großen RDF-Wissensdatenbanken beigetragen.

RDF-Wissensdatenbanken bestehen aus Subjekt-Prädikat-Objekt-Tripeln, wobei Subjekt und Objekt Entitäten darstellen und Prädikate Beziehungen zwischen Entitäten repräsentieren. Diese Datenbanken sind reichhaltige Informationsquellen, die zur schnellen und präzisen Beantwortung von Informationsbedürfnissen verwendet werden können. Zur Formulierung eines Informationsbedürfnisses werden typischerweise Anfragesprachen basierend auf Tripel-Patterns, zum Beispiel die vom W3C unterstützte Anfragesprache SPARQL, verwendet. Effektives und effizientes Information Retrieval für RDF-Daten und dessen Skalierbarkeit auf Web-Dimensionen beinhaltet herausfordernde Forschungsproblem. Diese Dissertation präsentiert Lösungen zu den Kernaspekten der folgenden Problembereiche.

- **Datenunvollständigkeit:** Während RDF-Datenbanken Informationen in Form von SPO-Tripeln bereitstellen, ist ein Großteil der im Web verfügbaren Daten nur als Freitext auf Webseiten enthalten. Daher kann die Kombination von RDF und Freitext die Reichhaltigkeit von Wissensdatenbanken erheblich erweitern. Diese Arbeit zeigt eine lösung. Es wird eine Erweiterung der Suche basierend auf Tripeln vorgestellt, die es Nutzern ermöglicht, Anfragen um Schlüsselwörter zu erweitern und somit eine größere Bandbreite von Informationsbedürfnissen zu befriedigen.

- **Ergebnisranking:** Große RDF-Wissensdatenbanken enthalten verfälschte oder fehlerhafte Informationen und liefern Anfrageergebnisse mit stark

schwankender Qualität. Daher ist es generell von Vorteil, Anfrageergebnisse in Form von sortierten Ranglisten anstelle unsortierter Mengen zu präsentieren. Sind zu einer Anfrage basierend auf Tripeln Schlüsselwörter definiert worden, so ist das Ranking besonders wichtig, um garantieren zu können, dass bezüglich der Schlüsselwörter relevante Anfrageergebnisse an den Anfang der Ergebnisliste gestellt werden. Zu diesem Zweck stellt diese Dissertation einen Ansatz auf Basis statistischer "Language-Models" vor, welcher nicht nur auf reine Tripel-Pattern-basierte Anfragen anwendbar ist, sondern auch auf deren Erweiterung mit Schlüsselwörtern.

- **Approximative Treffer:** Obwohl Tripel-Pattern-Anfragen sehr ausdrucksstark sind, insbesondere wenn sie durch Schlüsselwörter ergänzt werden, sind sie andererseits durch die Boolesche Auswertung der Bedingungen auch sehr restriktiv. Besonders bei Anfragen, die im nicht-approximativen Fall zu sehr wenigen Ergebnissen führen, kann der Recall durch die Anwendung eines approximativen Ansatzes deutlich gesteigert werden. Um dieses Ziel zu erreichen, wird ein Framework vorgestellt, welches eine Tripel-Pattern-Anfrage automatisch umformuliert und eine Menge von ähnlichen Anfragen generiert. Zusätzlich wird ein Ranking-Model entwickelt, welches auf die Vereinigung der Ergebnisse der Originalanfrage und der generierten Varianten angewand wird.

- **Effiziente Anfrageverarbeitung:** Eine Teilaufgabe der Suche basierend auf Tripel-Patterns in einer RDF-Wissensdatenbank ist das Pattern-Matching, welches insbesondere im Zusammenhang mit Schlüsselwortanfragen und dem automatischen Umschreiben sehr teuer werden kann. Durch das Ranking wird das Verfahren zusätzlich komplexer. Um die Antwortzeit dennoch gering halten zu können, werden inkrementelle Retrieval- und Rankingverfahren benötigt. Diese Dissertation stellt Top-k Algorithmen vor, welche die effiziente Bearbeitung von Tripel-Pattern-Anfragen mit Schlüsselwörtern und die automatische Generierung von Alternativen unterstützen.

- **Suche mit Schlüsselwörtern:** Tripel-Pattern-Anfragen sind in der Regel nur für Experten geeignet. Nicht-Experten sind eher mit der Schlüsselwortsuche vertraut. Um die in RDF-Datenbanken enthalten Informationen auch für solche Nutzer zugänglich zu machen, wird ein Framework vorgestellt, welches eine schlüsselwortbasierte Suche in RDF-Wissensdatenbanken unter Verwendung von statistischen "Language-Models" zum Ranking der Ergebnisse ermöglicht.

- **Diversifikation von Ergebnissen:** Während das Ranking sicherstellt, dass die relevantesten Ergebnisse an den Anfang der sortierten Ergebnisliste gestellt werden, sind die besten Ergebnisse oftmals sehr homogen und erschweren die Suche für Nutzer, die an weniger populären Aspekten interessiert sind. Das Prinzip des Ergebnisdiversifikation stellt sicher, dass das Ranking eine gewisse Vielfalt an Ergebnissen liefert und gibt Nutzern einen besseren Gesamtüberblick. Zu diesem Zweck wird in dieser Dissertation ein Framework vorgestellt, welches dieses Prinzip anwendet, um diversifizierte Top-k-Anfrageergebnisse zu ermitteln.

- **Exploration von Wissensdatenbanken:** Auch wenn die Bearbeitung von Anfragen in RDF-Wissensdatenbanken sehr präzise Ergebnisse liefert, bevorzugen Nutzer gelegentlich das Explorieren von Daten, um mehr über ein bestimmtes Thema zu erfahren. Es ist daher notwendig, Tools zur interaktiven Exploration von RDF-Datenbanken zur Verfügung zu stellen. In dieser Dissertation werden zwei Systeme vorgestellt, welche die explorative Suche unterstützen sowie Informationen externer Quellen berücksichtigen können. Das erste System ist ein Dokumenten-Retrieval-System, welches eine Rangliste von Dokumenten zu einer gegebenen Menge von RDF-Tripeln ermittelt. Das zweite System ist ein Entitäten-Visualisierung-System und fasst Entitäten unter Berücksichtigung zeitlicher Aspekte zusammen.

Contents

Chapter 1.

Introduction

1.1. Structured Data on the Web

While the World-Wide-Web is best known as a large repository of hyperlinked documents, it also contains a significant amount of structured data. The prime example of such data is the Deep Web, referring to data stored in databases that are typically served by querying HTML forms. This includes a vast amount of data such as the data provided by online retailers, news portals, and even social networks like Facebook and Twitter. Annotated data produced by social-tagging communities are yet another example of structured data on the Web. In addition, the increasing popularity of knowledge-sharing communities such as Wikipedia and the recent success in automatically extracting structured-information from semi-structured as well as natural-language Web sources have resulted in a strong leap in the amount of structured data available on the Web.

Exploiting structured data on the Web to improve search quality has been a constant goal of Web-Search engines. For instance, there are current trends to present search results in an entity-centric manner and in some cases to provide users with concise answers to their queries as opposed to a mere list of ranked documents. Utilizing structured data is also evident in general-purpose semantic-search services on the Web, such as WolframAlpha[1], Google Squared[2] and EntityCube[3], as well as in domain-specific Web portals such as news, stocks, government, health and medical portals and so on.

[1]http://www.wolframalpha.com/
[2]http://www.google.com/squared
[3]http://entitycube.research.microsoft.com/

Structured data on the Web is largely heterogeneous and exists in several formats. Such data shares many similarities with the kind of data traditionally managed by database systems but also reflects some unusual characteristics of its own; for example, there is no definite schema or a centralized data design as there is in a traditional database; and, unlike traditional databases that focus on a single domain, Web data covers a wide range of data types. Existing data-management systems do not address these challenges and assume their data is modeled within a well-defined domain, and are thus non-suitable to store and manage the wealth of heterogeneous structured data on the Web.

1.2. RDF Knowledge Bases

To overcome the aforementioned issue of data representation and management, many solutions have been proposed. In recent years, the Semantic-Web data model RDF (Resource Description Framework) has gained popularity in many domains as a representation format for heterogeneous Web-style structured data. For instance, RDF is now heavily used in many applications on scientific data such as biological networks like the Universal Protein Resource[4], social Web2.0 applications [8], large-scale knowledge bases such as DBpedia [4] or YAGO [78], and more generally, as a light-weight representation for the "Web of data" [6].

RDF data consists of a set of subject-predicate-object (SPO) triples, where subjects and objects are generally entities and predicates represent relationships between entities. For example, the triple

```
Woody_Allen  directed  Annie_Hall
```

is a triple with subject `Woody_Allen`, object `Annie_Hall` and predicate `directed`. In RDF, subjects, predicates and objects are either URIs (Uniform Resource Identifiers), literals or blank nodes [71]. However, for the sake of simplicity, URI references will be mentioned here using the URI suffix only. We explain this in more details in Chapter 2.

An RDF knowledge base is a collection of RDF triples, such as the one shown above. Today, there are many RDF knowledge bases; some contain more than a billion triples about various entity types such as people, companies, books,

[4]http://www.uniprot.org/

Subject (S)	Predicate (P)	Object (O)
Woody_Allen	actedIn	Annie_Hall
Woody_Allen	directed	Annie_Hall
Woody_Allen	hasWonPrize	Academy_Award_for_Best_Director
Paul_Simon	actedIn	Annie_Hall
Diane_Keaton	actedIn	Annie_Hall
Diane_Keaton	hasWonPrize	Academy_Award_for_Best_Actress
Mel_Gibson	directed	Braveheart
Mel_Gibson	actedIn	Braveheart
Mel_Gibson	produced	Braveheart
Mel_Gibson	hasWonPrize	Academy_Award_for_Best_Director
Clint_Eastwood	actedIn	Million_Dollar_Baby
Clint_Eastwood	directed	Million_Dollar_Baby
Clint_Eastwood	hasWonPrize	Academy_Award_for_Best_Director
Morgan_Freeman	actedIn	Million_Dollar_Baby
Morgan_Freeman	hasWonPrize	Academy_Award_for_Best_Actor
George_Clooney	directed	Leatherheads
George_Clooney	actedIn	Leatherheads
George_Clooney	wasNominatedFor	Academy_Award_for_Best_Director
Roman_Polanski	actedIn	The_Tenant
Roman_Polanski	directed	The_Tenant
Roman_Polanski	hasWonPrize	Golden_Globe_Award_for_Best_Director

Table 1.1.: An excerpt from an RDF knowledge base about movies

scientific publications, films, music, television and radio programs, genes, proteins, drugs and clinical trials, online communities, statistical and scientific data, and reviews.

Table 1.1 shows an excerpt from an RDF knowledge base about movies. Despite the repetitive structure in some parts of the data, there is often a high diversity of predicate names across the entire knowledge base. Thus, a flexible means of searching and exploring *schema-less* data such as RDF data is needed.

1.3. Searching RDF Knowledge Bases

Searching RDF knowledge bases is typically done using *triple-pattern* queries, such as the queries written in the W3C-endorsed SPARQL language. A triple-pattern query consists of conjunctions of elementary SPO search conditions, the so-called *triple patterns*. Triple-pattern queries offer the equivalent of SQL select-project-join queries, but in contrast to SQL, allow wildcards in place of predicate names. For example, to find a list of directors who have won an Academy Award and movies they directed and in which they also acted, the following triple-pattern query can be issued:

```
?d  hasWonPrize  Academy_Award_for_Best_Director
?d  directed     ?m
?d  actedIn      ?m
```

The above query is composed of a conjunction of three triple patterns. Each triple pattern is a triple where one or more of its SPO components are variables. In the example query, the first pattern has a variable subject denoted by `?d`. Similarly, the second and third patterns have the same variable subject `?d` and a variable object denoted by `?m`. Using the same variable in different triple patterns denotes a join condition. The results to such a query would then be *tuples* consisting of three triples, where the variable `?d` is bound to a director, `?m` is bound to a movie and `?a` is bound to an actor in the same movie. Table 1.2 shows all results retrieved when running the example query against the RDF knowledge base shown in Table 1.1.

RDF knowledge bases equipped with triple-pattern search provide a very powerful tool for knowledge discovery. Unlike traditional Web search where queries are typically few keywords and there is no explicit means of defining semantic relations between keywords, triple-pattern queries allow users to explicitly express semantic relations between entities. Moreover, while results in a traditional Web search are typically a list of ranked documents, the result of a triple-pattern query is a list of tuples consisting of one or more triples. These tuples provide concise answers to the user's information need as opposed to a list of documents, where information extraction has then to be deployed, at least cognitively, in order to extract the necessary answers. In addition, these tuples can conceptually have different sources and the information they contain may not be necessarily existing in one single document.

R_1	Woody_Allen	hasWonPrize	Academy_Award_for_Best_Director
	Woody_Allen	directed	Annie_Hall
	Woody_Allen	actedIn	Annie_Hall
R_2	Mel_Gibson	hasWonPrize	Academy_Award_for_Best_Director
	Mel_Gibson	directed	Braveheart
	Mel_Gibson	actedIn	Braveheart
R_3	Clint_Eastwood	hasWonPrize	Academy_Award_for_Best_Director
	Clint_Eastwood	directed	Million_Dollar_Baby
	Clint_Eastwood	actedIn	Million_Dollar_Baby

Table 1.2.: Results for the example query "directors who have won an Academy Award and movies they directed and in which they also acted"

1.4. Research Challenges

In order to fully utilize RDF Knowledge bases, there are still major challenges to be addressed which we highlight next.

Data Incompleteness. While large RDF knowledge bases contain a vast amount of information in the form of SPO triples that are either obtained from structured data sources, or via automatic information extraction from semi-structured and textual sources, the majority of information on the Web is available in the form of free text. Thus, augmenting RDF knowledge bases with text can increase the scope of such knowledge bases making them very rich sources of information. For example, the set of RDF triples in Table 1.1 represents information about movies, directors, actors and awards. While this covers a wide range of interesting information, there is still information that cannot be easily captured in terms of RDF triples. For example movie plots, taglines, users' comments and so on. Such information naturally appears as free text and by omitting them altogether, we lose a lot of valuable information.

Flexible Querying. Even though triple-pattern queries are highly expressive, they are also very restrictive since they deploy Boolean matching (i.e., a result is either a match to a query or not). It is thus crucial to equip triple-pattern search with flexible querying capabilities and to support approximate matching to allow a more effective searching of RDF knowledge bases. For example, consider our example query asking for directors who have won an Academy Award and movies they directed and in which they also acted. Directors who have been *nominated* for an Academy Award or have won a *Golden Globe*, and movies they directed and in which they also acted, are all potentially relevant results to the original information need. Similarly, directors or even actors who have won *any* award and movies they directed and in which they also acted are again somehow relevant to the given query. Thus, allowing *approximate* matches can improve the recall of such advanced queries, especially for queries with insufficient number of exact matches.

In addition, assume that the user is interested in finding movies that have something to do with, say, boxing or movies that were directed by controversial directors. Unless there exists explicit entities corresponding to "boxing" and "controversy", and explicit relationships linking these entities to movies and directors, there is noway such queries can be expressed. However, if RDF knowledge bases were extended with text, and keyword conditions were allowed, this can go a long way in addressing a wider range of information needs such as the ones just mentioned.

Finally, triple-pattern search, even when augmented with keywords, is still best targeted for expert users or programming APIs. Average users are accustomed to keyword search which is the paradigm to search for information on the Web. It is thus beneficial to consider sacrificing the expressiveness of triple-pattern queries, and also support plain keyword search over RDF knowledge bases. While we sacrifice query expressiveness, searching RDF knowledge bases with keywords still gains from the conciseness of RDF data combining information from different sources; information that does not necessarily exist in one particular source and thus could not be retrieved by traditional search engines.

Result Ranking. Large RDF knowledge bases may contain noisy or incorrect information and thus queries may produce many results of highly varying quality, in particular when keyword conditions are allowed or approximate matching is deployed. It is thus highly desirable to present users with a ranked list of results rather than a mere a list of unranked matches. For example, when asking for directors who have won an Academy Award and movies they directed and in which they also acted, it is essential to provide exact matches first, followed by any approximate matches. Also, if we add keyword conditions to such a query, say finding those movies that have something to do with ”boxing”, ranking of results should take into consideration how relevant they are to the keyword conditions. Finally, with keyword search in place, we add an additional level of ambiguity that is not present in the case of triple-pattern search, and in that case result ranking is again very crucial.

Efficient Query Processing. Triple-pattern search over RDF knowledge bases involves pattern matching. This becomes in particular very expensive when keyword conditions are allowed and when approximate matching is supported. Moreover, result ranking adds another level of complexity since all matches for a given query should be identified, ranked based on some scoring function and then returned to the user in the order of their scores. Incremental retrieval and ranking of results is thus needed to improve the response time of such queries.

Result Diversity. While ranking ensures that the most relevant results are ranked on top, it is often the case that the top results tend to be homogeneous, making it difficult for users interested in less popular aspects to find relevant results. For example, considering our example query, we do not want to have movies by the same director dominating the top results, or movies of the same genre, or in case query reformulation is allowed, people that have won the same award. Thus, result diversity can play a big role in ensuring that the users get a broad view of the different aspects of the results matching their queries, and ensures that, on average, almost all users can find relevant results to their queries in the top ranks.

Knowledge Exploration. As mentioned earlier, results to queries over RDF knowledge bases are typically tuples of triples joined together. While this is a very concise representation of answers to users’ information needs, it is often the case that users like to explore the knowledge base in order to learn more about a certain topic or subject. It is thus necessary to provide users with tools that allow them to interactively explore an RDF knowledge base.

1.5. Contributions

In this thesis, we present a number of novel models and algorithms to effectively search RDF knowledge bases. Our models address the issues pointed out in the previous section. In particular, our contributions can be summarized as follows.

1. **Adding Text and Result Ranking:** We show how to augment traditional RDF knowledge bases with text to extend their scope of coverage, and we propose an extension to triple-pattern search that enables expressing keyword conditions in combination with triple patterns. We also develop a ranking model based on statistical language models for ranking the results to triple-pattern queries, possibly augmented with keywords. Our framework for triple-pattern search over RDF knowledge bases was published in the proceedings of the 18th ACM Conference on Information and Knowledge Management (CIKM 2009) [23], the proceedings of the first International Workshop on Keyword Search on Structured Data (KEYS 2009) [25] which was co-located with the 2009 ACM SIGMOD/PODS Conference, and as a journal article in the IEEE Data Engineering Bulletin, Vol. 33 No. 1, March 2010 [24].

2. **Automatic Query Reformulation:** We present a framework for query reformulation that automatically reformulates triple-pattern queries in such a way that the original query intention is preserved. This has the advantage of improving the recall of such queries without unduly sacrificing precision. Our query reformulation framework was published in the proceedings of the 8th Extended Semantic Web Conference (ESWC 2011) [26].

3. **Top-k Query Processing:** To be able to efficiently process triple-pattern queries over large RDF knowledge bases, we develop a framework for efficient top-k query processing based on the family of top-k rank-join algorithms for traditional databases [43, 28]. Our top-k query processing framework also handles the cases of keyword-augmented triple-pattern queries and automatic query reformulation.

4. **Keyword Search:** To increase the usability of RDF knowledge bases, and to allow casual users to be able to search such knowledge bases, we propose

a framework for plain keyword search over RDF knowledge bases. Analogous to triple-pattern search, our framework also retrieves a set of tuples matching the user query, rather than entities or documents. We provide result ranking as well, which is again based on statistical language models. Our framework for keyword search over RDF knowledge bases was published in the proceedings of the 20th ACM Conference on Information and Knowledge Management (CIKM 2011) [21].

5. **Result Diversity:** Diversifying the search results in order to ensure that the top-k results would cover different aspects of the searched space is a very important component. We define a notion of result diversity in an RDF setting and develop a general technique based on the Maximal-Marginal-Relevance[10] in order to provide diverse results to queries over RDF knowledge bases.

6. **Knowledge Exploration:** We propose two different systems to explore RDF knowledge base, in order to learn more about query results, an entity or a set of RDF triples. The first system is a document retrieval system that can be used to retrieve a set of documents that contain the information encoded in a given set of RDF triples. Our document retrieval approach was published in the proceedings of the 36th International Conference on Very Large Data Bases (VLDB 2010) [22] and the proceedings of the 20th Conference on Information and Knowledge Management (CIKM 2011) [59]. The second system is an entity summarization tool that combines structured information from RDF knowledge bases with semi-structured and unstructured information from external sources and displays such combined information in an interactive timeline fashion. Our entity summarization approach was published in the proceedings of the 20th International World Wide Web Conference(WWW 2011) [83].

1.6. Thesis Outline

We give an overview on RDF knowledge bases and show how we can augment them with text in Chapter 2. We present our ranking model for triple-pattern search in Chapter 3. Chapter 4 describes our framework for automatic query-reformulation. Our top-k query processing techniques are described in Chapter 5. We describe our framework for keyword search over RDF knowledge bases in Chapter 6 and our result diversity approach is covered in Chapter 7. Finally, we describe our knowledge exploration tools in Chapter 8 and summarize our findings and highlight future research directions in Chapter 9.

Chapter 2.

RDF Knowledge Bases

RDF has become a common standard for representing structured data on the Web. In this chapter, we give an overview on RDF, and we formally define RDF knowledge bases. We then show how we can extend RDF knowledge bases with text, in order to expand their scope. Finally, we give a short overview on the current state of RDF data on the Web.

2.1. Resource Description Framework

RDF is a standard model endorsed by the W3C Consortium to represent information on the Web [71]. It was originally designed to represent metadata about Web resources, such as the author, title, date of creation, etc. However, RDF is not restricted to "Web resources" only, and today it is used to represent various information about many different types of resources including people, organizations, products, books, movies and so on.

Using RDF, information about the described resources can be represented in a common standard format so that this information can be easily exchanged among different applications without a loss of meaning. In RDF, there are three types of identifiers that can be used to describe information: *URIs*, *literals* and *blank nodes*. We explain each one separately.

URIs. A URI (Uniform Resource Identifier) is a string of characters used to identify a resource on the Internet. In RDF terminology, a URI is a unique identifier used to identify a single resource. For example, the director Woody Allen can be identified using the URI

```
http://en.wikipedia.org/wiki/Woody_Allen
```

Note that there might exist more than one URI to identify the same resource. For example, Woody Allen can also be identified using the URI

```
http://www.imdb.com/name/nm0000095/
```

Also note that URIs in RDF do not necessarily correspond to a Web address or URL. For example, the URI

```
http://www.knowledgebase.com/Woody_Allen
```

which does not physically exist in the Internet, can also be used to identify the director Woody Allen.

To simplify reference to resources, RDF is equipped with namespaces. A namespace is an abbreviation for the prefix of a URI. For example, the namespace `w` can be used to abbreviate the URI prefix

```
http://en.wikipedia.org/wiki/
```

In such case, Woody Allen can be identified by the identifier `w:Woody_Allen` which is a shorthand for

```
http://en.wikipedia.org/wiki/Woody_Allen
```

For the sake of readability, we will omit the namespaces when referring to a resource and we assume that given a URI suffix, the full URI can be uniquely resolved.

Literals. A literal is a string representation of a certain value. For example, the string `"09.03.1981"` is a string representation of the 9th day of March of the year 1981. RDF consists of two types of literals: *plain* literals and *typed* literals. Plain literals have a lexical form and optionally a language tag whereas typed literals have a lexical form and a data type that describes the type of the value they represent. For example, the literal `"09.03.1981"` has type date, and the literal `"9"` has type Integer, and so on.

Blank Nodes. A blank node represents a resource whose URI is not known or is irrelevant. The resource represented by a blank node is also called an anonymous resource.

SPO Triples. Given these three types of identifiers, URIs, literals and blank nodes, RDF can be used to encode all sorts of information on the Web. In RDF, information is represented in the form of statements. An RDF statement is a triple consisting of three fields: a *subject*, a *predicate* and an *object*. These triples are typically referred to as *SPO triples*.

Definition 2.1 : Triple
Let U be the infinite set of all possible URIs, L be the infinite set of all possible literals and B be the infinite set of all possible blank nodes. Furthermore, assume that the sets U, L and B are pairwise disjoint. An SPO triple $t = (s, p, o)$ *is a* 3*—tuple such that* $s \in U \cup B$, $p \in U$ *and* $o \in U \cup L \cup B$.

In other words, an SPO triple consists of three components:

- the subject, which is an RDF URI or a blank node,
- the predicate, which is an RDF URI, and
- the object, which is an RDF URI, a literal or a blank node.

An SPO triple is conventionally written in the order subject, predicate, object. The predicate is also known as the property of the triple.

RDF uses SPO triples to represent information about resources. For example, assume we want to represent the information that Woody Allen is the director of the movie Annie Hall. This information can be expressed using the following SPO triple:

```
Woody_Allen  directed  Annie_Hall
```

where the subject `Woody_Allen` is the URI of the director Woody Allen, the object `Annie_Hall` is the URI of the movie Annie Hall and the predicate `directed` is the URI of the relation ″is the director of″.

As mentioned earlier, subjects and objects are not restricted to URIs. For instance, the information that Woody Allen's first name is ″Woody″ and last name is ″Allen″ can be expressed using the following two SPO triples:

```
Woody_Allen  hasFirstName  "Woody"
Woody_Allen  hasLastName   "Allen"
```

where the URIs `hasFirstName` and URIs `hasLastName` reference the properties ″first name″ and ″last name″ respectively, and `"Woody"` and `"Allen"` are two plain literals.

Subject (S)	Predicate (P)	Object (O)
Woody_Allen	directed	Match_Point
Woody_Allen	directed	Hollywood_Ending
Woody_Allen	actedIn	Hollywood_Ending
Woody_Allen	hasWonPrize	Academy_Award
Woody_Allen	hasWonPrize	BAFTA_Award
Scarlett_Johansson	actedIn	Match_Point
Tea_Leoni	actedIn	Hollywood_Ending
Match_Point	type	English_Movie
Hollywood_Ending	type	English_Movie
Vicky_Cristina_Barcelona	type	English_Movie

Table 2.1.: A small RDF knowledge base

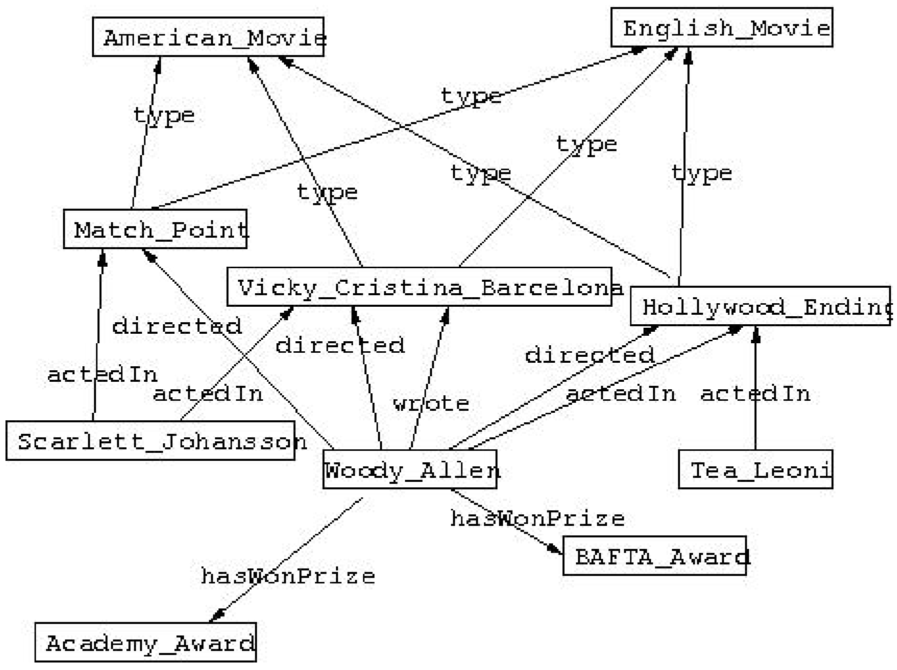


Figure 2.1.: An RDF graph corresponding to the knowledge base in Table 2.1

2.2. RDF Knowledge Bases

Definition 2.2 : RDF Knowledge Base
An RDF knowledge base KB *is a finite set of SPO triples.*

Table 2.1 represents an example RDF knowledge base. RDF knowledge bases can also be viewed as graphs which are often referred to as *RDF graphs*. An edge in an RDF graph is an SPO triple with the subject and object corresponding to vertices and the predicate corresponding to the label of the edge.

Definition 2.3 : RDF Graph
Given an RDF knowledge base KB*, let* S *be the set of all subjects in* KB*,* O *be the set of all objects in* KB*, and* P *be the set of all predicates in* KB*. The RDF graph* G *corresponding to* KB *is a tuple* $(\mathsf{V}, \mathsf{LABS}, \mathsf{E})$ *where*

1. V *is a finite set of vertices such that* $\mathsf{V} = \mathsf{S} \cup \mathsf{O}$*,*
2. LABS *is a finite set of edge labels such that* $\mathsf{LABS} = \mathsf{P}$*, and*
3. E *is a finite set of edges* (v_1, l, v_2) *such that* $v_1, v_2 \in \mathsf{V}$ *and* $l \in \mathsf{LABS}$*.*

Figure 2.1 shows the RDF graph corresponding to the example knowledge base in Table 2.1. Assuming that subjects and objects are entities, and predicates are relationships, RDF graphs are in this case equivalent to the more classical Entity-Relationship graphs.

2.3. Text-Augmented RDF Knowledge Bases

While RDF provides a very general framework to represent precise information about resources, many information is better suited as text and cannot be represented in the form SPO triples. For example, consider the plot of the movie `Annie_Hall`:

> *Romantic adventures of neurotic New York comedian Alvy Singer and his equally neurotic girlfriend Annie Hall. The film traces the course of their relationship from their first meeting, and serves as an interesting historical document about love in the 1970s.*

Such text snippet contains interesting information about the movie `Annie_Hall`, and including it in our example knowledge base can enhance its knowledge coverage. However, a text snippet cannot be represented in the form of SPO triples. A plot is naturally a sequence of keywords describing a movie; it is neither a literal nor a resource that can be uniquely identified using a URI.

As another example, assume we want to represent the information that the movie reviewer `Alain_Malek` has given the movie `Annie_Hall` the following review:

> *Woody Allen never created a more enjoyable film. Annie Hall is as innovative and clever as any movie has ever been. What makes Annie Hall such a great film is Allen's carefree screenplay and direction, in which he breaks all of the rules, giving the viewer the sense that anything can happen. Allen makes us characters into his story by talking to the camera, telling us jokes, and sharing his opinions with us.*

The fact that Alain has reviewed `Annie_Hall` can be easily represented in RDF using the following SPO triple:

```
Alain_Malek  reviewed  Annie_Hall
```

However, representing the review itself in RDF is not possible since the review is again merely a text snippet that cannot be represented in the form of SPO triples.

Thus, it is crucial to empower RDF knowledge bases with means for representing textual information that cannot be represented in the form of SPO triples. To this end, we associate with each triple t a text snippet $\mathsf{TEXT}(t)$ which is a simple bag-of-words.

Definition 2.4 : Text-Augmented RDF Knowledge Base
A text-augmented RDF knowledge base KB *is an RDF knowledge base such that each SPO triple* $t \in \mathsf{KB}$ *is associated with a, possibly empty, text snippet* $\mathsf{TEXT}(t)$.

2.4. RDF Data on the Web

The number of RDF datasets available on the Web is constantly increasing. The W3C Semantic Web Education and Outreach group's Linking Open Data project [56] is considered the biggest effort in inter-connecting and publishing such datasets on the Web. This has resulted in what is now known as *Linked Open Data Cloud* which is shown in Figure 2.2. As of August, 2011 , the Linked Open Data Cloud consists of *256* datasets with over *30 billion* triples and over *400 million* links between them [57].

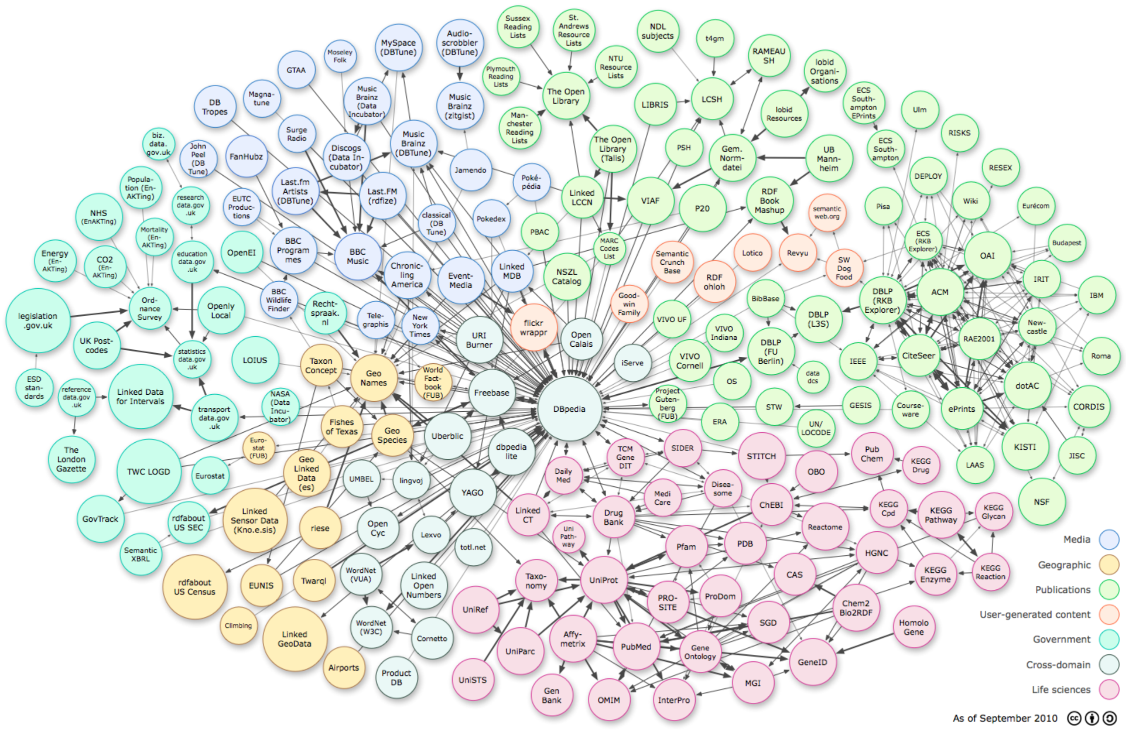


Figure 2.2.: The Linking Open Data Cloud Diagram

As Figure 2.2 shows, certain datasets serve as linking hubs in the Linked Open Data Cloud . Examples of such cenetral datasets are DBpedia [4] and YAGO [78] which both consist of RDF triples extracted from infoboxes and categories in Wikipedia. DBpedia has focused on recall by gathering all infobox attribute name-value pairs, at the risk of incorporating noise or inconsistent triples. DBpedia currently consists of 1 billion RDF triples, out of which 385 million were extracted from the English edition of Wikipedia and roughly 665 million were extracted from editions in other languages and links to external datasets.

YAGO, on the other hand, pursued the philosophy of high - near-human-quality - precision by employing database style consistency checking on triple candidates. YAGO primarily gathers its knowledge by rule-based information extraction on the infoboxes and category system of Wikipedia, and reconciles the resulting triples with the taxonomical class system of WordNet [61]. The resulting knowledge base contains more than 2 million entities and 20 million RDF triples, with at least 95 percent accuracy. YAGO has been incorporated into

DBpedia as well as other datasets. In addition to harvesting semistructured data and structured databases, there has been efforts to expand YAGO with triples extracted from natural-language text such as the information extraction tool SOFIE [79].

Another example of a heavily-linked dataset in the Linked Open Data Cloud is Geonames [31]. Geonames provides RDF descriptions of millions of geographical locations worldwide. Geonames was recently integrated with a new edition of YAGO known as YAGO2 [38] which contains around 10 million entities and more than 460 million RDF triples about them.

2.5. Summary

There are many RDF data and knowledge bases on the Web today. In RDF, data is represented in the form of SPO triples consisting of three fields: subject, predicate and object. While RDF is a general framework that can be used to represent a wide range of information, some information is better suited as text, and cannot be naturally represented in the form of SPO triples. It is thus crucial to enable RDF knowledge bases to represent and store free text in combination with structured RDF data. We refer to these knowledge bases that combine RDF data with text as text-augmented RDF knowledge bases.

Chapter 3.

Triple-Pattern Search

RDF knowledge bases consisting of SPO triples are typically searched using structured query-languages such as SPARQL [76], or similarly designed triple-pattern-based search. In this chapter, we formally define triple-pattern search, and we show how we can augment triple-pattern queries with keywords in order to search text-augmented RDF knowledge bases. Moreover, we present a novel ranking model for results to triple-pattern queries, possibly augmented with keywords. In order to improve the recall of such queries, we also present a relaxation paradigm, that relaxes triple-pattern queries in order to retrieve additional potentially-relevant results.

3.1. Query Framework

3.1.1. Triple-Pattern Queries

The most prominent way to search RDF knowledge bases such as the one shown in Table 3.1 is using structured *triple-pattern queries*. A triple-pattern query consists of a set of *triple patterns*, where a triple pattern is an SPO triple with one or more *variables*. We first define what a variable is, and then define triple patterns, and triple-pattern queries. We then show few examples of triple-pattern queries.

Definition 3.1 : Variable
Given an RDF knowledge base KB, *let* U *be the infinite set of all URIs,* L *be the infinite set of all literals, and* B *be the infinite set of all blank nodes. The set of variables* VAR *is an infinite set of labels such that* $\mathsf{VAR} \cap \mathsf{U} = \phi$, $\mathsf{VAR} \cap \mathsf{L} = \phi$ *and* $\mathsf{VAR} \cap \mathsf{B} = \phi$. *That is,* VAR *is pairwise disjoint from* U, L *and* B.

Subject (S)	**Property (P)**	**Object (O)**
`Woody_Allen`	`actedIn`	`Annie_Hall`
`Woody_Allen`	`directed`	`Annie_Hall`
`Woody_Allen`	`hasWonPrize`	`Academy_Award_for_Best_Director`
`Paul_Simon`	`actedIn`	`Annie_Hall`
`Diane_Keaton`	`actedIn`	`Annie_Hall`
`Diane_Keaton`	`hasWonPrize`	`Academy_Award_for_Best_Actress`
`Mel_Gibson`	`directed`	`Braveheart`
`Mel_Gibson`	`actedIn`	`Braveheart`
`Mel_Gibson`	`produced`	`Braveheart`
`Mel_Gibson`	`hasWonPrize`	`Academy_Award_for_Best_Director`
`Jules_Dassin`	`actedIn`	`Rififi`
`Jules_Dassin`	`directed`	`Rififi`
`Jules_Dassin`	`hasWonPrize`	`Academy_Award_for_Best_Director`
`Morgan_Freeman`	`actedIn`	`Rififi`
`Morgan_Freeman`	`hasWonPrize`	`Academy_Award_for_Best_Actor`
`George_Clooney`	`directed`	`Leatherheads`
`George_Clooney`	`actedIn`	`Leatherheads`
`George_Clooney`	`wasNominatedFor`	`Academy_Award_for_Best_Director`
`Roman_Polanski`	`actedIn`	`The_Tenant`
`Roman_Polanski`	`directed`	`The_Tenant`
`Roman_Polanski`	`hasWonPrize`	`Golden_Globe_Award_for_Best_Director`

Table 3.1.: An excerpt from an RDF knowledge base about movies

We denote a variable $var \in VAR$ by a single letter preceded by a question mark. For example, the label `?x` denotes a variable.

Definition 3.2 : Triple Pattern
A triple pattern $q = (s, p, o)$ *is an SPO triple such that* $s \in VAR \cup U \cup B$ *and* $p \in VAR \cup U$ *and* $o \in VAR \cup U \cup L \cup B$.

That is, a triple pattern q is an SPO triple such that one or more of its SPO components are variables. For example, the triple

```
Woody_Allen  directed  ?m
```

is a triple pattern whose object is the variable ?m. Similarly, the triple

```
?x  ?y  Academy_Award_for_Best_Director
```

is a triple pattern whose subject is the variable ?x and whose predicate is the variable ?y. A triple pattern is the basic unit of triple-pattern queries, with which many information needs can be addressed.

Definition 3.3 : Triple-Pattern Query
A triple-pattern query $Q = (q_1, q_2, ..., q_n)$ *is a tuple of* n *triples* q_i *such that each* q_i *is a triple pattern.*

For example, to find directors that have won the Academy Award for Best Director, movies they directed and in which they also acted, the following query consisting of three triple patterns can be issued:

```
?d  hasWonPrize  Academy_Award_for_Best_Director
?d  directed     ?m
?d  actedIn      ?m
```

We write each triple pattern in a separate line. Using the same variable in more than one triple pattern denotes a join condition that is used to combine different triples to form query results.

3.1.2. Keyword-Augmented Triple-Pattern Queries

It is often the case that certain information needs cannot be completely expressed in terms of triple patterns. In such cases, it would be beneficial to allow keyword conditions to be expressed in combination with structured triple patterns. This can be done by the so-called *keyword-augmented triple patterns* which we define next.

Definition 3.4 : Keyword-Augmented Triple Pattern
A keyword-augmented triple pattern is a triple pattern q *augmented with a set of keywords* $W = \{w_1, w_2, .., w_k\}$ *where* w_i *is a single keyword.*

For example, the following triple pattern

`Woody_Allen  directed  ?m` *complicated relationships* is a triple pattern augmented with two keywords *complicated* and *relationships*.

As another example, consider the keyword-augmented triple pattern:

```
?x  ?y  Academy_Award_for_Best_Director   controversy scandal
```

This triple pattern is again augmented with two keywords: *controversy* and *scandal*.

Triple patterns and keyword-augmented triple patterns can be combined together to form keyword-augmented triple-pattern queries.

Definition 3.5 : Keyword-Augmented Triple-Pattern Query
A keyword-augmented triple-pattern query $Q = (q_1, q_2, ..., q_n)$ *is a tuple of* n *triples* q_i *such that each* q_i *is a triple pattern, and such that one or more triple pattern* q_i *is a keyword-augmented pattern.*

For example, to find controversial or scandalous directors who have won the Academy Award for Best Director, and movies they directed and in which they also acted, the following query consisting of three triple patterns can be issued:

```
?d  hasWonPrize  Academy_Award_for_Best_Director   controversy scandal
?d  directed     ?m
?d  actedIn      ?m
```

We write each triple pattern in a separate line. The first triple pattern in the above query is augmented with the keywords *controversy* and *scandal*. Note that such information need cannot be served using triple patterns only (i.e., without keywords).

The keywords specified in a keyword-augmented query serve as additional filters or conditions that the query results must satisfy. Keyword augmentation is a powerful machinery that can be used to enhance the functionality of the Boolean-match triple-pattern queries. For example, in addition to associating a triple pattern with a set of disjunctive keywords as in our previous examples, we can allow users to associate the triple patterns with phrases, where the whole phrase must be matched. In addition, we can enable the usage of logical operators (negation, AND and OR operators) among keywords or phrases. For example, the following keyword-augmented query:

```
?d  hasWonPrize  Academy_Award_for_Best_Director   USA AND UK
?d  directed     ?m
?d  actedIn      ?m                                "major role"
```

?d	**hasWonPrize**	**Academy_Award_for_Best_Director**
Woody_Allen	hasWonPrize	Academy_Award_for_Best_Director
Mel_Gibson	hasWonPrize	Academy_Award_for_Best_Director
Jules_Dassin	hasWonPrize	Academy_Award_for_Best_Director

Table 3.2.: Triples from the knowledge base in Table 3.1 instantiating an example triple pattern

asks for directors that have won the Academy Award for Best Director and have something to do with *both* the USA and the UK, and movies in which they also played a ”major role”.

3.1.3. Query Results

Results to triple-pattern queries are tuples of triples *instantiating* the query triple patterns and satisfying the query join conditions which are denoted by using the same variable in more than one triple pattern. We first explain what a triple-pattern instantiation is and then show how this can be used to retrieve results to queries consisting of more than one triple pattern.

Definition 3.6 *:***Mapping**
A mapping μ *is a partial function* $VAR \rightarrow U \cup L \cup B$. *The domain of* μ, $dom(\mu)$, *is the subset of* VAR *where* μ *is defined.*

Definition 3.7 *:* **Compatible Mappings**
Two mappings, μ_1 *and* μ_2 *are said to be compatible if* $\forall var \in dom(\mu_1) \cap dom(\mu_2)$, $\mu_1(var) = \mu_2(var)$.

Note that two mappings with disjoint domains are always compatible, and that the empty mapping (i.e., the mapping with empty domain) is compatible with any other mapping.

Definition 3.8 *:* **Pattern Instantiation**
Let q *be a triple pattern, and let* $VAR(q)$ *be the set of variables occurring in* q. *A triple* $t \in KB$ *is said to instantiate triple pattern* q *in knowledge base* KB *if there exists a*

mapping μ *such that* $\mathrm{dom}(\mu) = \mathrm{VAR}(q)$ *and triple* t *is obtained by substituting each* $var \in \mathrm{VAR}(q)$ *with* $\mu(var)$.

For example, consider the triple pattern

```
?d  hasWonPrize  Academy_Award_for_Best_Director
```

and consider the mapping μ such that $\mu(?d) =$ `Woody_Allen`. Substituting ?d with `Woody_Allen` results in the following triple

```
Woody_Allen  hasWonPrize  Academy_Award_for_Best_Director
```

which is an instantiation of the above triple pattern in the knowledge base shown in Table 3.1. Table 3.2 shows the set of all instantiations of the above triple pattern from the knowledge base shown in Table 3.1.

Definition 3.9 : Query Result
Given a triple pattern query $Q = (q_1, q_2, ..., q_n)$ *where* q_i *is a triple pattern, possibly augmented with keywords, a result is defined as a tuple* $T = (t_1, t_2, ..., t_n)$ *consisting of* n *triples* t_i *such that there exist mappings* $\mu_1, \mu_2, ..., \mu_n$ *which are all pairwise compatible and* t_i *instantiates* q_i *by substituting each* $var \in \mathrm{VAR}(q_i)$ *with* $\mu_i(var)$.

For example, consider the following query consisting of three triple patterns

```
?d  hasWonPrize  Academy_Award_for_Best_Director
?d  directed     ?m
?d  actedIn      ?m
```

Table 3.3 shows the results to such query when run against the knowledge base in Table 3.1. All results are tuples consisting of three triples where the first triple is an instantiation of the first query triple-pattern, the second triple is an instantiation of the second query triple-pattern and the third triple is an instantiation of the third query triple-pattern. Moreover, each result tuple satisfies the join conditions specified in the query which requires that the subject of all the three triples be the same and the objects of the second and third triples be the same.

Results to keyword-augmented queries are defined in a similar way. They are also tuples of triples instantiating the query triple patterns and satisfying the join conditions specified in the query. In addition, the triples must also satisfy any keyword conditions specified in the triple pattern they instantiate . Recall

Q	**?d**	**hasWonPrize**	**Academy_Award_for_Best_Director**
	?d	**directed**	**?m**
	?d	**actedIn**	**?m**
R_1	Woody_Allen	hasWonPrize	Academy_Award_for_Best_Director
	Woody_Allen	directed	Annie_Hall
	Woody_Allen	actedIn	Annie_Hall
R_2	Mel_Gibson	hasWonPrize	Academy_Award_for_Best_Director
	Mel_Gibson	directed	Braveheart
	Mel_Gibson	actedIn	Braveheart
R_3	Jules_Dassin	hasWonPrize	Academy_Award_for_Best_Director
	Jules_Dassin	directed	Rififi
	Jules_Dassin	actedIn	Rififi

Table 3.3.: Results of an example query as tuples of triples

that each triple in a keyword-augmented knowledge base is associated with a text snippet which is constructed from any contextual text present in the sources the triples were extracted from. Now, given a keyword-augmented query, the results are tuples of triples that instantiate the query triple patterns and in addition, whose text snippets satisfy the keyword conditions specified in the query.

For example, consider the following keyword-augmented query asking for scandalous directors that have won the Academy Award for Best Director, and movies they directed and in which they also acted:

?d	hasWonPrize	Academy_Award_for_Best_Director	*scandal controversy*
?d	directed	?m	
?d	actedIn	?m	*"major role"*

One possible result to such query is the tuple:

```
Mel_Gibson  hasWonPrize  Academy_Award_for_Best_Director
Mel_Gibson  directed     Braveheart
Mel_Gibson  actedIn      Braveheart
```

provided that the text snippet of the first triple contains the keyword *scandal* or *controversy* and the text snippet of the third triple contains the phrase *"major role"*.

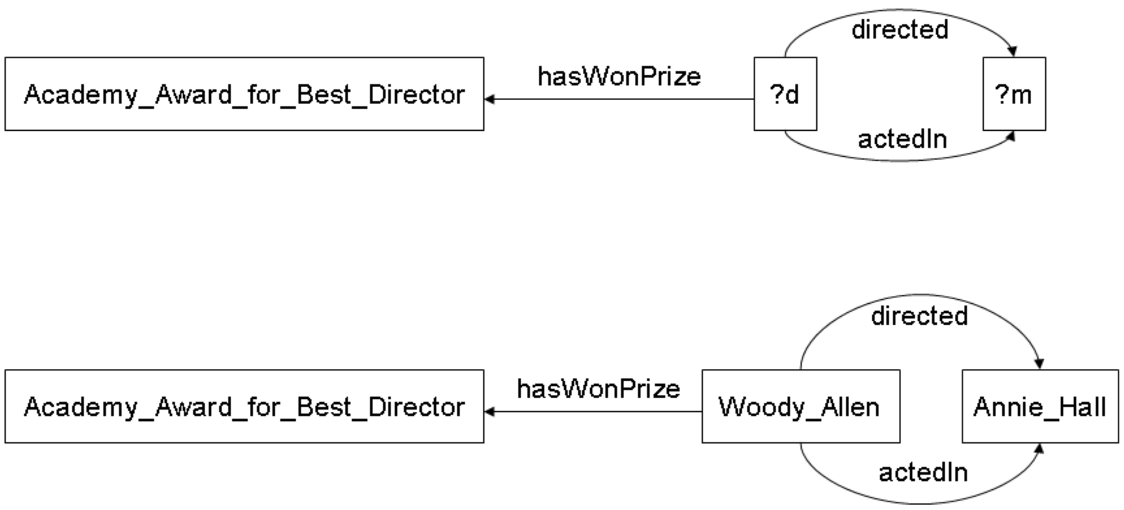


Figure 3.1.: A query as a graph (above) and one result (below) as a subgraph of the RDF graph

In graph terminology, a triple-pattern query can be viewed as a graph whose vertices $v \in U \cup L \cup B \cup VAR$ and whose edge labels $e \in U \cup VAR$. Similarly, a query result can be viewed as a subgraph of the RDF knowledge graph queried, that is isomorphic to the query graph when query variables are mapped to URIs or literals that appear in the RDF graph. Figure 3.1 shows an example query and one of its result as graphs.

Next, we motivate why result ranking is crucial, and we present a novel ranking model for triple-pattern queries, possibly augmented with keywords.

3.2. Ranking Model

While results to triple-pattern queries are concise tuples of triples, it is often the case that there are *too many* results for a given query. In such cases, users often prefer seeing a ranked list of results rather than a set of unranked answers to their queries. For example, consider the information need of finding directors that have won the Academy Award for Best Director, and movies they directed and in which they also acted. Such information need can be precisely expressed using a set of triple patterns. Table 3.3 shows the triple-pattern query corresponding to such information need, and the results to such query from our example knowledge base.

For instance, the first result in Table 3.3 states that Woody Allen has won the Academy Award for Best Director and that he directed and acted in the movie Annie Hall. Similarly, the second result in Table 3.3 is about Mel Gibson and his movie Braveheart, and the third result is about Jules Dassin and his movie Rififi. Even though all three results are considered as valid results to our example query, they might vary in their relevance to the underlying information need in various ways, and we discuss some of them next. Note that this query would yield much more results when run against a large movie knowledge base, and in such case the need for result ranking is even more obvious.

3.2.1. Ranking Criteria

Informativeness. Query results differ in how informative they are to the user that issued the query. For example, the user might be interested in finding new information that she does not already know. On the other extreme, the user might have a certain preference in movies or directors. Between these two extremes, one can assume there is no prior knowledge about the users issuing the queries. In such case, the system must try to infer what is the most likely ranking that would yield the best user satisfaction on average. For example, for the query about directors that have won the Academy Award for Best Director, and movies they directed and in which they also acted, we could assume that, in general, users are interested in in popular or well known movies and directors, and thus results about well-known people or movies should precede those about less-known ones in the ranking.

The informativeness of results becomes particularly evident in the case of keyword-augmented queries. For example, consider the following query asking for controversial or scandalous directors that have won the Academy Award for Best Director and movies they directed and in which they also acted:

```
?d  hasWonPrize  Academy_Award_for_Best_Director  controversy scandal
?d  directed     ?m
?d  actedIn      ?m
```

The results to such query are the same as the ones shown in Table 3.3. The ranking of such results must take into consideration how well they match the keyword conditions in the above keyword-augmented triple-pattern query. Recall that we assume that each triple is associated with a text snippet. Taking into consideration how well the text snippet of the triple matches the keyword conditions in the corresponding triple pattern is crucial to better serve the information need represented by the keyword-augmented query. For our example query, the results about the director Mel Gibson should thus be ranked on top.

Balancing all these factors is a challenging task, and thus it is very important to develop a ranking model that is general enough to easily accommodate such factors or a subset of them, which highly depends on the available information to the system as well as the assumptions made about the users' goals.

To capture informativeness, we rely on the so-called *witness counts* of the triples. The witness count of a triple indicates the number of sources the triple was extracted from. In case the triples were extracted from one source, say a Deep-Web source, the witness counts can be estimated from another corpus, such as the Web for instance. The witness count of a triple is an estimate of the *importance* of the triple. In addition, for each keyword present in a triple text snippet, a witness count of the triple-keyword pair is also stored, which is the number of sources from which the triple was extracted and which in addition contain the keyword.

Definition 3.10 : Witness Count
$\forall\ t \in \mathsf{KB}$, *the witness count* $c(t)$ *is the number of sources from which the triple* t *was extracted. In addition,* $\forall\ w \in \mathsf{TEXT}(t)$, *the witness count* $c(t;w)$ *is the number of sources from which the triple* t *was extracted and which contain the keyword* w.

Confidence. RDF knowledge bases are very precise information sources that involve minimal ambiguity on the representation level. However, they often involve inaccuracies. Recall that most such knowledge bases are automatically constructed using information extraction techniques from Deep-Web or text sources. This information-extraction procedure often involves errors, and results in some inaccuracies in the data. For example, consider the following text snippet

> *At the 2006 Academy Awards, Clooney was nominated for an Academy Award for Best Director for Good Night, and Good Luck, as well as best supporting Actor for Syriana, which he won.*

Assume that a triple stating that George Clooney has won the Academy Award for Best Director has been extracted from such a snippet. This triple is wrong since Clooney was only nominated for an Academy Award for Best Director and he won the award for Best Supporting Actor for his role in the movie Syriana.

Even when the accuracy of the triple extraction is guaranteed, the quality of the source from which such triple was extracted can be a measure of how well the triple holds. For example, triples extracted from the online encyclopedia *Wikipedia* are more likely to be true than those extracted from exotic blogs or random Web pages, which are more likely to contain more noisy facts. To this end, each triple in our knowledge base can be associated with a confidence value, reflecting the accuracy of the employed extraction method (e.g., regular-expression matching vs. natural-language parsing vs. statistical learners) and the authenticity and authority of the data sources.

Finally, combining informativeness and confidence, is a key factor for the success of any ranking approach for results to queries over RDF knowledge bases. To combine these two measures, we can scale down the witness count of a triple based on its confidence, by multiplying the witness count with the confidence value for instance.

We next present a novel ranking model based on statistical language models [36, 52] that can be used to rank the results of triple-pattern queries, whether augmented with keywords or not. Our ranking model takes into consideration both the informativeness of the triples and their confidence, and is general enough to accommodate any additional factors that might be domain or application specific. Before we present our model, we give a short overview on language-model-based ranking for information retrieval.

3.2.2. Language-Model-Based Ranking for Information Retrieval

There have been many information retrieval models proposed over the years. Among the most effective ones are the vector-space model with heuristic tf-idf weighting and the popular BM25 (Okapi) retrieval function [72]. In 1998, a new class of probabilistic models emerged. These new models are based on statistical language models which have been successfully used in related areas such as speech recognition and machine translation.

The term language model refers to a probability distribution over words. Many studies have shown that using statistical language models does not only lead to superior empirical performance, but also facilitates parameter tuning and opens up possibilities for modeling non-traditional retrieval problems. Since then, Statistical language models have been successfully applied to many advanced information retrieval problems such as cross-lingual retrieval [53], expert finding [29], XML retrieval [49] and many others. In general, statistical language models provide a principled way of modeling various kinds of retrieval problems.

There are two major classes of approaches that rely on language models to rank query results. The first class uses the so-called *query likelihood* to rank results, whereas the second class relies on the *Kullback-Leibler divergence* between a query language model and a result language model for ranking. We briefly explain each model next.

Query Likelihood Model

The use of language models for information retrieval was first introduced by Ponte and Croft in 1998 [69] in what is known as the query likelihood approach. In the query likelihood approach, we assume there exists a language model for each document which is a probability distribution over a set of terms V. The query is then assumed to be a sample of terms drawn from one of these language models. A documents D can thus be ranked based on the query likelihood which is defined as follows.

Definition 3.11 *:* **Query Likelihood**
Given a query Q, the query likelihood of document D is the probability of generating the query Q given the language model of document D which is denoted by $P(Q|D)$.

We thus set the score of document D with respect to query Q as the query likelihood of document D as follows:

$$s(Q, D) = P(Q|D) \tag{3.1}$$

Intuitively, if the query likelihood of document D is high, the query terms must have high probabilities in the document language model, which further means that the query terms occur frequently in document D.

In order to use such a model to score documents, we must solve two problems: (1) how to define the language model of a document D? and (2) how to estimate the parameters of the language model based on the document?. While Ponte and Croft defined the document language model as a multivariate Bernoulli distribution [69], it is more common to define the document language model as a multinomial distribution over a set of terms V, which is commonly referred to as a unigram language model [35, 60, 75]. A unigram language model M defined over a set of terms V would have exactly $|V|$ parameters $P(w_i|M)$ where $P(w_i|M)$ is the probability of term $w_i \in V$ in the language model M and $\Sigma_{i=1}^{|V|} P(w_i|M) = 1$.

Under the multinomial assumption, given a query $Q = \{q_1, q_2, .., q_m\}$, the query likelihood of document D is equal to:

$$P(Q|D) = \prod_{i=1}^{m} P(q_i|D) \tag{3.2}$$

Our ranking problem now boils down to estimating the parameters of the language model of document D (i.e., $P(w_i|D)$ for every term $w_i \in V$). In order to do so, we assume that the document is a sample of its language model and estimate the probability $P(w_i|D)$ using a maximum-likelihood estimator as follows:

$$P(w_i|D) = \frac{c(w_i; D)}{|D|} \tag{3.3}$$

where $c(w_i; D)$ is the frequency of term w_i in D and $|D|$ is the length of document D (i.e., the total frequencies of all terms in D).

Using the above equation, the query likelihood of document D can be computed. However, one problem with this maximum-likelihood estimator is that terms that do not appear in document D would get a zero probability, making all documents that do not contain all terms in the query Q have zero likelihood $p(Q|D)$. This is clearly undesirable. More importantly, since a document is a very small sample for our model, the maximum-likelihood estimate is generally not accurate (what is known as overfitting). So it is important to *smooth* the maximum-likelihood estimator so that we do not assign zero probabilities to terms that do not appear in the document and thus improve the accuracy of the estimated language model in general.

One way to do so is interpolating the maximum-likelihood estimate with a background language model estimated using the entire collection:

$$P(w_i|D) = \alpha \frac{c(w_i; D)}{|D|} + (1 - \alpha) P(w_i|C) \tag{3.4}$$

where $p(w_i|C)$ is a collection (background) language model estimated based on word counts in the entire collection and $\alpha \in [0, 1]$ is a smoothing parameter.

Kullback-Leibler Divergence Model

Rather than estimating the probability of generating the query from the document language model (as in the query likelihood model), both the query and the document can be viewed as samples from two different language models and the similarity between these two language models can be employed for ranking. In 2001, Lafferty and Zhai [52] introduce a risk minimization framework for information retrieval. One incarnation of their framework is to compute the score of a document with respect to a query by the Kullback-Leibler (KL) divergence between their respective language models which is defined as follows.

Definition 3.12 *: Kullback-Leibler Divergence*
Given two probability distributions Q *and* D *which are defined over the set of terms* V*, the Kullback-Leibler divergence between the two, denoted by* $KL(Q||D)$*, is computed as follows:*

$$KL(Q||D) = \sum_{w \in V} P(w|Q) \log \frac{P(w|Q)}{P(w|D)} \tag{3.5}$$

The KL divergence is an asymmetric, information-theoretic measure of how different two probability distributions are. We thus set the score of document D with respect to Q as follows:

$$S(Q, D) = KL(Q||D) \tag{3.6}$$

The documents are then ranked in ascending order of their scores. Again, the ranking problem here boils down to estimating the parameters of the language models of the query and the document (i.e., $P(w|Q)$ and $P(w|D)$ for each term $w \in V$). The probabilities $P(w|Q)$ and $P(w|D)$ can be estimated using a maximum-likelihood estimator as described in Equation 3.3, where we assume that the query language model and the document language model are both multinomial distributions over the set of terms V. Furthermore, we can also smooth the document language using a background language model as described in Equation 3.11. In such case, it can be easily shown that the ranking achieved using the KL divergence framework is equivalent to that achieved using the query likelihood model. However, in contrast to the query likelihood model, the KL divergence approach has the advantage that there is an explicit query language model that is estimated using the query. This enables us to extend this query language model to incorporate additional information such as relevance feedback for instance.

3.2.3. Language-Model-Based Ranking for Triple-Pattern Search

Our ranking model for triple-pattern queries is based on the Kullback-Leibler divergence framework and assumes there exists a language model for the query Q and a language model for each query result R. The results are ranked in increasing order of the KL divergence between the query language model and the result language model. The KL divergence between the two gives a measure of relevance of R with respect to Q.

We make two distinctions in our setting as compared to traditional keyword queries on documents. First, there is no notion of a document in our setting. Instead, we have a large knowledge base of triples from which results are constructed at query time as tuples of triples that instantiate the query triple patterns and satisfy the query join and keyword conditions.

Second, queries are made up of triple patterns, while results are made up of triples. A probability distribution over triple patterns is incomparable to a probability distribution over triples. We need to overcome this *vocabulary gap* in order to compare the query and result language models.

Query Language Model

Our ranking model assumes there exist three types of query-related language models: 1) an overall query language model, 2) a triple-pattern language model and 3) a triple-pattern-keyword pair language model. We define each one of these language models next, and then show how the parameters of these three types of language models can be estimated.

Definition 3.13 : Query Language Model
The language model of a query $Q = (q_1, q_2, ..., q_n)$*, where* q_i *is a triple pattern, is a probability distribution over all tuples of* n *triples of the form* $T = (t_1, ..., t_n)$ *where* $t_i \in KB$ *is a triple. The language model of query* Q *consists of* $|KB|^n$ *parameters* $P(T|Q)$

where $|KB|$ *is the total number of triples in the knowledge base* KB. *The parameter* $P(T|Q)$ *denotes the probability of tuple* T *in the query language model.*

Definition 3.14 : Triple-Pattern Language Model
The language model of a triple pattern q *is a probability distribution over all triples* $t \in KB$. *The language model of triple pattern* q *consists of* $|KB|$ *parameters* $P(t|q)$ *where* $|KB|$ *is the total number of triples in the knowledge base* KB. *The parameter* $P(t|q)$ *denotes the probability of triple* t *in the language model of triple pattern* q.

Definition 3.15 : Triple-Pattern-Keyword Language Model
The language model of a triple pattern q *and a keyword* w *is a probability distribution over all triples* $t \in KB$. *The language model of triple pattern* q *and keyword* w *consists of* $|KB|$ *parameters* $P(t|q, w)$ *where* $|KB|$ *is the total number of triples in the knowledge base* KB. *The parameter* $P(t|q, w)$ *denotes the probability of triple* t *in the language model of triple-pattern* q *and keyword* w.

Estimating the Language Models. Given a query $Q = (q_1, ..., q_n)$ where q_i is a triple pattern, the probability $P(T|Q)$ of tuple $T = (t_1, t_2, ..., t_n)$ in the query language model is estimated as follows (assuming independence between the triples):

$$P(T|Q) = \prod_{i=1}^{n} P(t_i|q_i) \tag{3.7}$$

where $P(t_i|q_i)$ is the probability of triple t_i in the language model of triple pattern q_i. The probability $P(t_i|q_i)$ can be estimated in two ways, depending on whether q_i is augmented with keywords or not.

In case q is not augmented with any keywords, the probability $P(t_i|q_)$ can be estimated as follows. Let $\hat{q_i}$ denote the set of instantiations of q_i(i.e., the set of triples that instantiate triple pattern q_i in the knowledge base KB). The probability $P(t_i|q_i)$ is set to:

$$P(t_i|q_i) = \begin{cases} \dfrac{c(t_i)}{\sum_{t \in \hat{q_i}} c(t)} & \text{if } t_i \in \hat{q_i} \\ 0 & \text{otherwise} \end{cases} \tag{3.8}$$

where $c(t)$ is the witness count of triple t.

In case q_i is augmented with keywords $w_1, w_2, ..., w_m$, the probability $P(t_i|q_i)$ is estimated as follows (assuming independence between the keywords $w_1, w_2, .., w_m$):

$$P(t_i|q_i) = \prod_{j=1}^{m} P(t_i|q_i, w_j) \tag{3.9}$$

where $P(t_i|q_i, w_j)$ is the probability of triple t_i in the language model of triple-pattern q_i and keyword w_j which in turn can be estimated as follows. Let $\hat{q_i}$ denote the set of instantiations of q_i. The probability $P(t_i|q_i, w_j)$ is set to:

$$P(t_i|q_i, w_j) = \begin{cases} \dfrac{c(t_i; w_j)}{\sum_{t \in \hat{q_i}} c(t; w_j)} & \text{if } t_i \in \hat{q_i} \\ 0 & \text{otherwise} \end{cases} \tag{3.10}$$

where $c(t; w)$ is the witness count for the triple-keyword pair (t, w).

Note that using Equation 3.9, the probability of a triple t_i in the language model of triple pattern q_i is estimated as the product of the probabilities of the triple t_i in the language models of the triple pattern q_i and keywords $w_1, w_2, ..., w_j$ (a language model corresponding to each (q_i, w_j) pair). These probabilities are estimated according to Equation 3.10. This would mean that $P(t_i|q_i)$ would be *zero* if $c(t_i; w_j)$ is zero for any keyword w_j. To interpret keywords as *disjunctive*, we need to smooth the language model of the triple pattern q_i and keyword w_j which can be done as follows. Given a triple pattern q_i and a keyword w_j, the smoothed probability of triple t in the language model of triple-pattern q_i and keyword w_j is set to:

$$P(t_i|q_i, w_j) = \begin{cases} \alpha \frac{c(t_i; w_j)}{\sum_{t \in \hat{q_i}} c(t; w_j)} + (1 - \alpha) \frac{c(t_i)}{\sum_{t \in \hat{q_i}} c(t)} & \text{if } t_i \in \hat{q_i} \\ 0 & \text{otherwise} \end{cases} \tag{3.11}$$

where $c(t; w)$ is the witness count of the triple-keyword pair (t, w), $c(t)$ is the witness count of triple t and α is a smoothing parameter. The above linear interpolation smoothing of the probability of a triple t in the language model of a triple pattern q and keyword w ensures that all triples that instantiate triple pattern q would have non-zero probabilities even if they are not associated with any of the keywords specified in the triple pattern. This is in contrast to triples that do not instantiate the triple pattern q which would all have zero probabilities.

Query Language Model Properties . Our estimation method for the query language model relies on the witness count $c(t)$ of a triple t and the witness count $c(t; w)$ of a triple t and keyword w as can be seen in Equation 3.8 and Equation 3.10. These witness counts can be seen as term frequencies ($\mathtt{tf}$ in IR jargon) for triples. The higher such values are, the higher the probability of a triple t in the corresponding language model. The witness counts are normalized using the sum of witness counts of all triples that instantiate a triple pattern (denominators in Equation 3.8 and Equation 3.10). These normalizations act as a weighting component that demotes triple patterns that have many instantiations or triple patterns whose instantiations' sum of witness counts are very high (i.e., the bigger the sum of witness counts of the instantiations of a triple pattern is, the smaller the probability of a triple t that instantiates this pattern is). This might be seen as a counter part to the inverse-document frequency ($\mathtt{idf}$) in traditional IR. While this does not make a difference for the case of triple patterns only since triple patterns in a query are assumed to be conjunctive and thus every result tuple must contain triples that instantiates all the triple patterns in the query, the weighting is in particular very appealing for the case of keyword-augmented triple patterns which are considered to be disjunctive. This means that a result tuple that matches a highly weighted keyword would have a high probability in the query language model as we show in the following examples.

Query Language Model Examples. Consider running the following query $Q = (q_1, q_2)$ with two triple patterns asking for Australian actors and their movies against a small knowledge base about movies:

q_1 :	?a	bornIn	Australia
q_2 :	?a	actedIn	?m

Table 3.4 shows the list of triples that instantiate triple pattern q_1 and their witness counts. Similarly, Table 3.5 shows the list of triples that instantiate triple pattern q_2 and their witness counts.

Consider the tuple $T = (t_1, t_6)$ which is composed of the following two triples

t_1 :	Mel_Gibson	bornIn	Australia
t_6 :	Mel_Gibson	actedIn	Braveheart

#	**Triple** t			**c**(**t**)
t_1	Mel_Gibson	bornIn	Australia	40
t_2	Nicole_Kidman	bornIn	Australia	30
t_3	Heath_Ledger	bornIn	Australia	20
t_4	Russel_Crow	bornIn	Australia	10
$\Sigma c(t)$				100

Table 3.4.: The instantiation list of triple pattern q_1 : `?a bornIn Australia`

#	**Triple** t			**c**(**t**)
t_5	Tom_Hanks	actedIn	Forest_Gump	70
t_6	Mel_Gibson	actedIn	Braveheart	40
t_7	Nicole_Kidman	actedIn	The_Others	20
t_8	George_Clooney	actedIn	Syriana	10
t_9	Julia_Roberts	actedIn	Pretty_Woman	30
t_{10}	Heath_Ledger	actedIn	Brokeback_Mountain	10
t_{11}	Russel_Crow	actedIn	Gladiator	20
$\Sigma c(t)$				200

Table 3.5.: The instantiation list of triple pattern q_2 : `?a actedIn ?m`

The probability $P(T|Q)$ of tuple T in the language model of query Q is computed according to Equation 3.7 as follows:

$$P(T|Q) = P(t_1|q_1).P(t_6|q_2)$$

In turn, $P(t_1|q_1)$ and $P(t_6|q_2)$ are computed according to Equation 3.8 as follows:

$$P(t_1|q_1) = \frac{c(t_1)}{\Sigma_{t \in \hat{q}_1} c(t)} = \frac{40}{100} = 0.4$$

and

$$P(t_6|q_2) = \frac{c(t_6)}{\Sigma_{t \in \hat{q}_2} c(t)} = \frac{40}{200} = 0.2$$

Thus, even though the witness count of triple t_6 is the same as that of triple t_1, the probability of t_6 in the language model of triple pattern q_2 is smaller than that of triple t_1 in the language model of triple pattern q_1. This is due to the

#	**Triple** t			**c(t**; *dead*)
t_5	`Tom_Hanks`	`actedIn`	`Forest_Gump`	2
t_6	`Mel_Gibson`	`actedIn`	`Braveheart`	5
t_7	`Nicole_Kidman`	`actedIn`	`The_Others`	10
t_8	`George_Clooney`	`actedIn`	`Syriana`	3
t_9	`Julia_Roberts`	`actedIn`	`Pretty_Woman`	0
t_{10}	`Heath_Ledger`	`actedIn`	`Brokeback_Mountain`	0
t_{11}	`Russel_Crow`	`actedIn`	`Gladiator`	0
$\Sigma c(t; \textit{dead})$				20

Table 3.6.: The instantiation list of triple pattern q_2 : `?a actedIn ?m` with the count of witnesses containing the keyword *dead*

fact that the sum of witness counts for triples that instantiate pattern q_2 is bigger than sum of witness counts for triples that instantiate pattern q_1. This in turn means that triples that instantiate triple pattern q_1 (which might be viewed as more exotic) would be given a higher weight than those that instantiate triple pattern q_2.

Finally, the probability of our result tuple $T = (t_1, t_6)$ in the language model of query Q is equal to:

$$P(T|Q) = 0.4 * 0.2 = 0.08$$

For keyword-augmented queries, the query language model can be estimated in a similar way. For example, Consider augmenting our example query with the keywords *dead people* to find Australian actors that acted in movies that have something to do with dead people:

q_1 :	`?a`	`bornIn`	`Australia`	
q_2 :	`?a`	`actedIn`	`?m`	*dead people*

Table 3.6 shows the list of triples that instantiate triple pattern q_2 and for each such triple, the count of witnesses that contain the keyword *dead*. Similarly, Table 3.7 shows the list of triples that instantiate triple pattern q_2 and the count of witnesses of such triples that contain the keyword *people*.

Consider the tuple $T = (t_1, t_6)$ which is composed of the following two triples

#	Triple t			c(t; *people*)
t_5	`Tom_Hanks`	`actedIn`	`Forest_Gump`	30
t_6	`Mel_Gibson`	`actedIn`	`Braveheart`	10
t_7	`Nicole_Kidman`	`actedIn`	`The_Others`	8
t_8	`George_Clooney`	`actedIn`	`Syriana`	18
t_9	`Julia_Roberts`	`actedIn`	`Pretty_Woman`	14
t_{10}	`Heath_Ledger`	`actedIn`	`Brokeback_Mountain`	10
t_{11}	`Russel_Crow`	`actedIn`	`Gladiator`	10
$\Sigma c(t; \textit{people})$				100

Table 3.7.: The instantiation list of triple pattern q_2 : `?a actedIn ?m` with the count of witnesses containing the keyword *people*

t_1 :	`Mel_Gibson`	`bornIn`	`Australia`
t_6 :	`Mel_Gibson`	`actedIn`	`Braveheart`

The probability $P(T|Q)$ of tuple T in the language model of query Q is computed according to Equation 3.7 as follows:

$$P(T|Q) = P(t_1|q_1).P(t_6|q_2)$$

As explained before, $P(t_1|q_1)$ is computed according to Equation 3.8 as follows:

$$P(t_1|q_1) = \frac{c(t_1)}{\Sigma_{t \in \hat{q_1}} c(t)} = \frac{40}{100} = 0.4$$

Since q_2 is augmented with two keywords *dead* and *people*, $P(t_6|q_2)$ is computed according to Equation 3.9 as follows:

$$P(t_6|q_2) = P(t_6|q_2, \textit{dead}).P(t_6|q_2, \textit{people})$$

$P(t_6|q_2, \textit{dead})$ is computed according to Equation 3.10 (omitting smoothing for simplicity) as follows:

$$P(t_6|q_2, \textit{dead}) = \frac{c(t_6; \textit{dead})}{\Sigma_{t \in \hat{q_2}} c(t; \textit{dead})} = \frac{5}{20} = 0.25$$

Similarly, $P(t_6|q_2, \textit{people})$ is computed according to Equation 3.10 as follows:

$$P(t_6|q_2, \textit{people}) = \frac{c(t_6; \textit{people})}{\Sigma_{t \in \hat{q_2}} c(t; \textit{people})} = \frac{10}{100} = 0.1$$

That is, even though the count of witnesses for triple t_6 that contain the keyword *people* is twice as much as those that contain the keyword *dead*, the probability $P(t_6|q_2, \textit{people})$ is smaller than $P(t_6|q_2, \textit{dead})$. This is again due the fact that the sum of witness counts for triples that instantiate pattern q_2 and contain the keyword *people* is bigger than the sum of witness counts for triples that instantiate pattern q_2 and contain keyword *dead*. This in turn means that triples that instantiate triple pattern q_2 and whose text snippets contain the keyword *dead* (which might be viewed as less frequent in the knowledge base) would be given a higher weight than those whose text snippets contain the keyword *people*.

Finally, the probability of triple t_6 in the language model of triple pattern q_2 would be equal to:

$$P(t_6|q_2) = 0.1 * 0.25 = 0.025$$

and the probability of the whole tuple $T = (t_1, t_2)$ in the language model of query Q can thus be computed as follows:

$$P(T|Q) = 0.4 * 0.025 = 0.01$$

Result Language Model

Similar to the case of queries, we assume there exists a language model for each query result which is defined as follows.

Definition 3.16 : Result Language Model
Given a query $Q = (q_1, q_2, ..., q_n)$, the language model of a result R is a probability distribution over all tuples of n triples of the form $T = (t_1, ..., t_n)$ where t_i is a triple. The language model of result R consists of $|KB|^n$ parameters $P(T|R)$ where $|KB|$ is the total number of triples in the knowledge base KB. The parameter $P(T|R)$ denotes the probability of tuple T in the result language model.

Estimating the Result Language Model. The probability $P(T|R)$ of a tuple T in the language model of result R is estimated as follows:

$$P(T|R) = \frac{c(T, R)}{|R|} \tag{3.12}$$

where $c(T|R)$ is the number of times tuple T occurs in result R and $|R|$ is the length of result R (i.e., total number of occurrence of all tuples in R). Now, recall that given a query $Q = (q_1, q_2, ..., q_n)$ with n triple patterns q_i, a result R is a tuple $T_R = (t_1, t_2, ..., t_n)$ consisting of n triples t_i such that t_i instantiates the triple pattern q_i and the triples in T_R satisfy the query join conditions denoted by using the same variable in more than one triple pattern.

That is, each result R would contain only a single tuple T_R. Thus, the probability $P(T|R)$ of tuple T in the language model of result R would be 1 if $T = T_R$ and 0 otherwise. To avoid this overfitting in the estimation of the tuple probabilities in the result language model, we utilize smoothing as follows:

$$P(T|R) = \beta\frac{c(T, R)}{|R|} + (1-\beta)P(T|C) \tag{3.13}$$

where $P(T|C)$ is the probability of tuple T in the collection (background) language model and the parameter β is a smoothing parameter. The probability $P(T|C)$ of tuple T in the collection language model could be estimated in various ways. For example, we could assume a uniform distribution over all tuples of n triples, in which case the probability $P(T|C)$ would be the same for every tuple. We could also assume independence between the triples constituting the tuple T as follows:

$$P(T|C) = \prod_{i=1}^{n} P(t_i|C) \tag{3.14}$$

$P(t_i|C)$ can then be estimated using the witness count of the triple t with respect to the whole knowledge base as follows:

$$P(t_i|C) = \frac{c(t_i)}{\sum_{t\in KB} c(t)} \tag{3.15}$$

where $c(t)$ is the witness count of triple t.

Result Ranking

Given a query $Q = (q_1, q_2, ..., q_n)$ and a result R consisting of a tuple T_R, we set the score of the result with respect to the query as the KL divergence between the language model of query Q and the language model of result R. The KL divergence was defined in Subsection 3.2.2. The KL divergence between the query and a result language model is computed as follows:

$$S(Q, R) = KL(Q||R) = \sum_{i=1}^{|KB|^n} P(T_i|Q)\log\frac{P(T_i|Q)}{P(T_i|R)} \tag{3.16}$$

Rank	?a ?a	bornIn actedIn	Australia ?m	S(Q, R)
1	Mel_Gibson Mel_Gibson	bornIn actedIn	Australia Braveheart	0.08
2	Nicole_Kidman Nicole_Kidman	bornIn actedIn	Australia The_Others	0.03
3	Heath_Ledger Heath_Ledger	bornIn actedIn	Australia Brokeback_Mountain	0.01
4	Russel_Crow Russel_Crow	bornIn actedIn	Australia Gladiator	0.01

Table 3.8.: The ranked list of results for an example query

The results are then returned to the user in ascending order of their scores. The above computation involves a summation over all tuples of n triples, which are the terms the query and result language models are defined on. In Appendix A, we show that:

$$S(Q,R) \propto -P(T_R|Q)\log(1+\frac{\beta}{(1-\beta)P(T_R|C)}) \tag{3.17}$$

where T_R is the tuple constituting result R. Furthermore, if we assume that the background language model is uniform over all tuples, we have:

$$S(Q,R) \propto -P(T_R|Q) \tag{3.18}$$

That is, we can now rank the query results in descending order of the probabilities of their tuples in the query language model $P(T_R|Q)$ as we show in the following examples.

Result Ranking Examples. Consider the following query $Q = (q_1, q_2)$ with 2 triple patterns asking for Australian actors and their movies:

q_1 :	?a	bornIn	Australia
q_2 :	?a	actedIn	?m

Table 3.4 shows the list of triples that instantiate triple pattern q_1 and their witness counts. Similarly, Table 3.5 shows the list of triples that instantiate triple

Rank	**?a** **?a**	**bornIn** **actedIn**	**Australia** **?m** *dead people*	**S(Q, R)**

1	`Nicole_Kidman`
	 `Nicole_Kidman`
		`bornIn`
		 `actedIn`
			`Australia`
			 `The_Others`
				0.012

2	`Mel_Gibson`
	 `Mel_Gibson`
		`bornIn`
		 `actedIn`
			`Australia`
			 `Braveheart`
				0.01

3	`Heath_Ledger`
	 `Heath_Ledger`
		`bornIn`
		 `actedIn`
			`Australia`
			 `Brokeback_Mountain`
				0

4	`Russel_Crow`
	 `Russel_Crow`
		`bornIn`
		 `actedIn`
			`Australia`
			 `Gladiator`
				0

Table 3.9.: The ranked list of results for an example query

pattern q_2 and their witness counts. Table 3.8 shows all results which consist of tuples of 2 triples instantiating patterns q_1 and q_2 and satisfying the query join condition (i.e., the subject of the first triple is the same as that of the second triple). Next to each result R we show the score of each result $S(Q, R)$ which is computed as the probability of the result tuple in the query language model.

As another example, consider the keyword-augmented query to find Australian actors that acted in movies that have something to do with dead people:

q_1 : `?a bornIn Australia`
q_2 : `?a actedIn ?m` *dead people*

Table 3.6 shows the list of triples that instantiate triple pattern q_2 and for each such triple, the count of witnesses that contain the keyword *dead*. Similarly, Table 3.7 shows the list of triples that instantiate triple pattern q_2 and the count of witnesses of such triples that contain the keyword *people*. Table 3.9 shows all results which consist of tuples of 2 triples instantiating patterns q_1 and q_2 and satisfying the query join conditions. Next to each result R we show the score of each result $S(Q, R)$ which is computed as the probability of the result tuple in the query language model (without smoothing).

3.2.4. Query Relaxation

Even though triple-pattern queries allow users to search RDF knowledge bases in a very precise manner, they often lack flexibility on the result retrieval side. Recall that a query result is a tuple of triples that instantiate the query triple patterns. This instantiation is assumed to be exact. Allowing approximate pattern-instantiation can improve the recall of such queries and can alleviate the problem of ”too few results”. For example, consider the following query consisting of 2 triple patterns

```
?d  hasWonPrize  Academy_Award_for_Best_Director
?d  directed     ?m
```

The above query asks for directors that have won the Academy Award for Best Director, and movies they directed. Directors that were *nominated for* an Academy Award for Best Director or directors that have won a *Golden-Globe Award for Best Director* and movies they directed are all potentially-relevant results to the original query. To retrieve such results, relaxed versions of the given query can be issued, in addition to the original query, and their results can be combined with the original query results before returning them to the user that issued the query. For example, the following relaxed query

```
?d  hasWonPrize  ?x
?d  directed     ?m
```

asks for directors or actors who have won *any* award, and movies they directed and in which they also acted. The above query is a relaxed version of the original example query, where the object of the first triple pattern is replaced with the variable `?x`.

We start by explaining how, given a triple-pattern query, a set of relaxed queries can be generated. We then explain how our ranking model is extended to handle query relaxation in order to provide a ranked list of exact and approximate query results.

Generating Relaxed Queries

A relaxed query is generated by relaxing one or more triple pattern. In turn, a triple pattern is relaxed by replacing one or more of the constants (i.e., a URI or a literal) specified in the triple pattern with a variable.

?d	**hasWonPrize**	**Academy_Award_for_Best_Director**
?d	**directed**	**?m**
?d	hasWonPrize	?x
?d	directed	?m
?d	?x	Academy_Award_for_Best_Director
?d	directed	?m
?d	?x	?y
?d	directed	?m
?d	hasWonPrize	Academy_Award_for_Best_Director
?d	?x	?m
?d	hasWonPrize	?x
?d	?y	?m
?d	?x	Academy_Award_for_Best_Director
?d	?y	?m
?d	?x	?y
?d	?z	?m

Table 3.10.: Relaxed queries for a two triple-pattern query

Definition 3.17 : Relaxed Query
Given a triple-pattern query $Q = (q_1, q_2, ..., q_n)$ *where* q_i *is a triple pattern, let* $VAR(Q)$ *be the set of variables that appear in* Q*. Let* $VAR_i \subset VAR$ *be a set of infinite variables corresponding to triple pattern* q_i *such that* $VAR_1, VAR_2, ..., VAR_n$ *are all pairwise disjoint and* $\forall\ 1 \leq i \leq n, VAR_i \cap VAR(Q) = \phi$*. Let* $CONST(q_i)$ *be the set of constants specified in triple pattern* q_i*. Let* $r(q_i)$ *be the set of relaxed triple patterns obtained by replacing one or more constants* $const_i \in CONST(q_i)$ *with a variable* $var_i \in VAR_i$*. The set of all relaxed queries* $R(Q)$ *is then:* $\{r(q_1) \cup \{q_1\} \times r(q_2) \cup \{q_2\} \times ... \times r(q_n) \cup \{q_n\}\}$.

Table 3.10 shows all possible relaxed queries for an example query.

Extending the Ranking Model

Our ranking model is extended to handle query relaxation as follows. Given a query $Q = (q_1, q_2, .., q_n)$ where q_i is a triple pattern, we rank the results to the query Q and all its relaxations using the ranking model described in Subsection 3.2.3. A result R is ranked by computing the KL divergence between the query language model and the result language model. While the estimation of the result language model follows the exact same procedure described in Subsection 3.2.3, the estimation of the query language model is slightly different when query relaxation is allowed. We describe next how to estimate the query language model when relaxations are allowed.

Estimating the Query Language Model with Relaxations. Given a query $Q = (q_1, q_2, ..., q_n)$ where q_i is a triple pattern, we estimate the probability of a tuple in the language model of query Q as follows (assuming independence between the triples):

$$P(T|Q) = \prod_{i=1}^{n} P(t_i|q_i) \tag{3.19}$$

Now, assume that triple pattern q_i has the set of relaxations $r(q_i) = \{q_i^1, q_i^2, ..., q_i^{m_i}\}$ where q_i^j is a relaxed triple pattern obtained by replacing one or more constants in q_i with a variable. The probability $P(t_i|q_i)$ is then estimated as a weighted sum of the following $m + 1$ probabilities:

$$P(t_i|q_i) = \lambda_0 P(t_i|q_i^0) + \lambda_1 P(t_i|q_i^1) + ... + \lambda_{m_i} P(t_i|q_i^{m_i}) \tag{3.20}$$

where q_i^0 is the original triple pattern q_i (i.e., without any relaxations). The probability $P(t_i|q_i^j)$ is the probability of triple t_i in the language model of triple pattern q_i^j which is estimated according to Equation 3.8 in case q_i^j is a simple triple pattern (i.e., not augmented with any keywords) and according to Equation 3.9 in case q_i^j is keyword augmented. The parameters λ_j weigh the contribution of each triple pattern (whether the original or its relaxations) and $\Sigma_{j=0}^{m_i} \lambda_j = 1$. In general, the λ_js are set based on the "closeness" of the relaxed pattern to the original one. We thus set the λs based on the number of relaxations in the pattern (i.e., constants replaced with variables). This means that the larger the number of relaxations is, the lower the weight is. This implies that the original triple pattern gets the highest weight, and relaxed patterns with the same number of relaxations get equal weight.

3.3. Related Work

Our work on ranking the results to triple-pattern queries on RDF knowledge bases is closely related to work on IR over structured data in general. Based on the types of data and queries handled, we classify prior work on ranking as follows: i) keyword queries on unstructured data (documents), ii) structured queries on structured data, iii) keyword queries on structured data, and iv) keyword-augmented structured queries on structured data.

3.3.1. Keyword Queries on Unstructured Data

The main technique that we use from the standard IR literature is that of language models and KL-divergence for result ranking [52]. An overview on language models for IR was given in Subsection 3.2.2. In recent years, keyword querying has been carried over to the extended setting of *entity search and ranking*, also referred to as expert finding [68, 73]. Here, results are named entities (e.g., companies, products, publications, authors), but the queries are still keyword-based. In most of these approaches, entities are assumed to be embedded in textual form in Web pages and other traditional kinds of documents. For the approaches that treat entities as first-class citizens [66], see Subsection 3.3.3 below. Extended forms of language models and PageRank-inspired spectral analyses are used to rank the entities that qualify for a keyword query. The key difference to our setting is that our corpus is a single redundancy-free RDF knowledge base instead of a set of documents, our queries consist of triple patterns rather than keywords and the output is a ranked list of tuples of triples instead of documents. We have described how to adapt language modeling techniques for this new setting.

3.3.2. Structured Queries on Structured Data

Ranking for structured queries has been investigated, for restricted forms of SQL queries. Ranking models have been developed for selection-join queries, using either tf-idf based models [16] or probabilistic-IR models [11] that leverage attribute-value statistics in both the database and the workload. It is thus not possible to use these models for schema-less and redundancy-free RDF data.

The closest work to ours is the ranking model in the NAGA system [48]. NAGA introduced a query language similar to SPARQL triple patterns and used a (simpler) LM for computing a notion of informativeness. But NAGA can rank only exact matches to a given query; so the ranking is helpful only for the too-many-answers case but not for the too-few-answers problem. In contrast, our triple-pattern search framework goes beyond this limited setting by supporting query relaxation and introducing the new notion of keyword-augmented queries.

3.3.3. Keyword Queries on Structured Data

The work on keyword search over structured data can be classified into two classes. The first class aims at mapping the keyword query into one or more structured query [82, 58]. In this chapter, we assumed that structured triple-pattern queries were given and we were interested in ranking the results to such queries. Inferring a structured query from a given keyword query is a different problem.

The second class of work on keyword search over structured data tries to directly retrieve structured results for a given keyword query. The work on keyword search over XML data for instance falls into this category. XKSearch [88] returns a set of nodes that contain the query keywords either in their labels or in the labels of their descendant nodes and have no descendant node that also contains all keywords. Similarly, XRank [34] returns the set of elements that contain at least one occurrence of all the query keywords, after excluding the occurrences of the keywords in sub-elements that already contain all the query keywords. However, all these techniques assume a tree-structure and thus can not be directly applied to graph-structured data such as RDF data.

Also, closely related to our work is the language-modeling approach for keyword search over XML data proposed in [49]. Their ranking is based on the hierarchal language models proposed in [67]. However, the setting of XML data is quite different from that of RDF since in XML the retrieval unit is an XML document (or a subtree). In an RDF setting, we are interested in ranking tuples of triples that match the user's query. These tuples are not present in advance and are computed on the fly during retrieval time, and thus most of the prior work on XML IR would not apply.

Keyword search over graphs which returns a ranked list of Steiner trees [41, 46, 39, 32] (the exception is [55] which returns graphs) deals with the latter problem of having a predefined retrieval unit. However, the result ranking in each of the above is based on the structure of the results [41, 47] (usually based on aggregating the number or weights of nodes and edges), or on a combination of these properties with content-based measures such as tf-idf [14, 39, 55] or language models [66].

For instance, the BANKS system [41] enables keyword search on graph databases. Given a keyword query, an answer is a subgraph connecting some set of nodes that "cover" the keywords (i.e., match the query keywords). The relevance of an answer is determined based on a combination of edge weights and node weights in the answer graph. The importance of an edge depends upon the type of the edge, i.e., its relationship. Node weights on the other hand represent the static authority or importance of nodes and are set as a function of the in-degree of the node. We adopt the BANKS system to the RDF setting in the experiments section, and compare it to our model.

A closely related work that combines structure and content for ranking is the language-model-based ranking model in [66] for ranking objects (resources in an RDF setting). The model assumes that each resource is associated with a set of records extracted from Web sources. In turn, each record is associated with a "document". The relevance of each such "document" (and correspondingly, the resource associated with it) to a keyword query is estimated using language models. This model however assumes that the retrieval unit is resources only, while our ranking model goes beyond this to treat triples in a holistic manner by taking into account the relationships between the resources. In addition, it assumes the presence of a document associated with each Web Object or resource, something that we lack in the case of RDF data in general.

3.3.4. Keyword-Augmented Structured Queries on Structured Data

XML IR like XPath Full-Text search falls into this category [37, 2]. XPath forms the tree-structured part of the query, while keyword conditions can be specified at each branch of the tree-pattern query. An important difference between XML IR and our setting is that in the former, it is possible to have results of different sizes, while in our case, the results are all of fixed structure. And so, the structure-based aspects are not as relevant to our setting as the content-based ones. The content-based ranking is again based either on tf-idf scores [2] or language models [37].

3.4. Experimental Evaluation

3.4.1. Setup

To evaluate the effectiveness of our ranking model, we conducted a comprehensive user study over two datasets using the Amazon Mechanical Turk service[1]. The first dataset was derived from a subset of the Internet Movie Database (IMDB) and the second dataset was derived from the LibaryThing community, which is an online catalog and forum about books. The data from both sources was automatically parsed and converted into RDF triples. In addition, each triple was also augmented with keywords derived from the data source it was extracted from. In particular, for the IMDB dataset, all the terms in the plots, taglines and keywords fields were extracted, stemmed and stored with each triple. For the LibraryThing dataset, since we did not have enough textual information about the entities present, we retrieved the books' Amazon descriptions and the authors' Wikipedia pages and used them as sources of keywords for the triples. An overview of the datasets is given in Table 3.11.

3.4.2. Evaluation Queries

We used 2 different sets of evaluation queries. The first set consisted of simple triple-pattern queries that ranged from single-pattern queries to multi-pattern graph queries. We constructed *16 queries* for the IMDB dataset and *8 queries* for the LibraryThing dataset. The second set consisted of keyword-augmented queries. Again, we constructed *16 queries* for the IMDB dataset and *8 queries* for the LibraryThing dataset. The queries were triple-pattern queries associated with one or more keywords. Moreover, for each evaluation query, we generated relaxed queries by replacing one or more SPO components in one or more triple patterns with variables. A subset of the evaluation queries used for the IMDB

[1] `http://aws.amazon.com/mturk/`

#entities	Some Entity Types	#triples	Some Relationship Types
IMDB Dataset			
59,000	`movie`, `actor` `director`, `producer` `country` , `language`	600,000	`actedIn`, `directed` `hasWonPrize`, `isMarriedTo` `produced`, `hasGenre`
LibraryThing Dataset			
48,000	`book`, `author` `user`, `tag`	700,000	`wrote`, `friendOf` `hasTag`, `type`

Table 3.11.: Overview of the datasets

dataset is shown in Table 3.12 and in Table 3.13 for the LibraryThing dataset. Appendix B shows the complete list of evaluation queries used in our study.

3.4.3. Competitors

We compared our approach against a number of competitors that represent state-of-the-art methods in ranking over structured data. For our approach, we used the ranking model described in Subsection 3.2.3. Since all triples in our two datasets were extracted using the same technique (parsing of semistructured data), we assumed the confidence of all triples to be the same, and we thus restrict our ranking criteria to the informativeness of the triples only. We computed the informativeness of a triple using its witness count. Since each triple is present only once in both data sources, we had to estimate the witness count for the triples. In order to do so, we relied on the Web corpus. We issued queries to a major search engine, with the subject and object of the triple as keywords, and set the witness count to the number of hits returned by the search engine.

We compared our model against the work done on web object retrieval (WOR) in [66] since they use language models in order to rank results. This covers a class of competitors that deal with term-frequency based ranking of results to keyword queries on structured data. Second, we compared to the class of rankers that utilize graph properties to rank results to keyword queries over structured data by adapting the Steiner weight scoring as used in BANKS [46]. Finally, we compared to the closest work to ours, the language-model-based

?m1	producedIn	Australia	
?m1	hasWonPrize	Academy_Award	
?m	hasGenre	Thriller	
?d	directed	?m	
?a1	isMarriedTo	?a2	
?a1	actedIn	?m	
?a2	actedIn	?m	
?d	hasWonPrize	Academy_Award_for_Best_Director	
?d	directed	?m	
?a	actedIn	?m	
?a	hasWonPrize	Academy_Award_for_Best_Actor	
?m1	hasGenre	Family	
?m1	hasProductionYear	1995	
?a	actedIn	?m1	
?m2	hasGenre	Comedy	
?a	actedIn	?m2	
?x	hasWonPrize	Academy_Award_for_Best_Actor	
?y	hasWonPrize	Academy_Award_for_Best_Actress	
?x	actedIn	?m	*love*
?y	actedIn	?m	*relationship*
?x	hasProductionYear	2001	
?x	hasGenre	Romance	*paris*
?x	directed	?y	*true story*
?x	hasWonPrize	?z	
?x	hasGenre	Comedy	*wedding*
?a	actedIn	?m	*spielberg*
?a	hasWonPrize	?x	

Table 3.12.: A subset of the evaluation queries for the IMDB dataset

ranking in NAGA [48]. We next describe how we adapted each competitor to rank the results of triple-pattern queries, possibly augmented with keywords, and how we handle query relaxation whenever applicable.

?x	wrote	?y	
?x	wrote	?y	
?y	hasTag	Series	
?x	wrote	?y	
?y	hasTag	Fiction	
?x	wrote	?z	
?z	hasTag	Non-fiction	
?x	wrote	?y	
?y	hasTag	20th_Century	
?y	hasTag	Classic	
?x	wrote	?y	
?y	type	Mystery_&_Thrillers	
?x	wrote	?y	*civil war*
?x	type	Novelists	
?y	hasTag	Movie	
?x	wrote	?y	*revolution*
?y	hasTag	Read	
?y	hasTag	Classic	
?x	wrote	?y	
?y	hasTag	Magic	
?y	type	Fiction	*award*
?x	wrote	?y	*pulitzer*
?y	hasTag	Classic	
?x	wrote	?y	*wizard*
?y	hasTag	Sequel	

Table 3.13.: A subset of the evaluation queries for the LibraryThing dataset

WOR. The Web Object Retrieval model proposed by Nie et al. [66] is a language-model-based approach for ranking objects, or *resources* in an RDF setting. The model assumes that each resource is associated with a set of records extracted from Web sources. In turn, each record is associated with a "document". The relevance of each such "document" (and correspondingly, the resource associated with it) to a keyword query is estimated using language models.

We adapted the WOR model to work in our setting as follows. First, we converted our evaluation queries into terms. We then treated triples as records and for a given resource X, we created a language model for X using *all* its triples $\{t_1, t_2, ..., t_n\}$ (i.e., all triples in which X is either a subject or an object). Given a keyword query $Q = \{q_1, q_2, ..., q_m\}$, we then ranked the resources according to their probability of generating the query which is computed as follows:

$$P(Q|X) = \prod_{i=1}^{m} \sum_{j=1}^{n} \frac{1}{n} P(q_i|D_j) \tag{3.21}$$

where $P(q_i|D_j)$ is the probability of generating the term q_i from the language model of triple t_j which was estimated from the document D_j using a maximum-likelihood estimator.

BANKS. The BANKS system enables keyword search on graph databases. Given a keyword query, an answer is a subgraph connecting some set of nodes that "cover" the keywords (i.e., match the query keywords). The relevance of an answer is determined based on a combination of edge weights and node weights in the answer graph. The importance of an edge depends upon the type of the edge, i.e., its relationship. Node weights on the other hand represent the static authority or importance of nodes and are set as a function of the in-degree of the node.

This directly applies to our setting. Given a triple-pattern query, whether keyword-augmented or not, we retrieved all the results of the query and all its relaxations. Recall that each query result is a tuple of triples, which can also be viewed as a subgraph of the RDF knowledge graph searched. We then ranked the subgraphs based on a combination of edge weights and node weights as proposed in their model. That is, the score of a subgraph $G = \{t_1, t_2, ..., t_n\}$ with nodes $N = \{n_1, n_2, ..., n_k\}$ is defined as follows:

$$score(G) = \lambda \sum_{i=1}^{n} score(t_i) + (1 - \lambda) \sum_{i=1}^{k} \frac{1}{k} score(n_i) \tag{3.22}$$

where $score(t_i)$ is the score of triple or edge t_i which was set as the witness count of t. The $score(n_i)$ on the other hand is the score of node n_i which was set to the in-degree of the node n_i. We used log scaling for both scores as advised in

[41]. Finally, the parameter λ controls the influence of both scores, and was set based on training queries.

NAGA. In NAGA, the results are ranked based on their likelihood of generating the query triple patterns. This probability was estimated using 3 different measures: 1) confidence of the result, 2) compactness of the result and 3) informativeness of the result. The first component does not play a role in our setting, since all triples in our datasets were extracted using the same method, and thus all have the same confidence values. Compactness does not play a role in the ranking either, since all result are of the same size which is determined by the number of triple patterns in the query. Thus, the only component that affects the ranking is the informativeness component which we computed using the witness counts. The major difference between our ranking method and that of the NAGA system is that we have explicit query and result language models which makes our model more general and easier to extend. Moreover, NAGA does not support keyword-augmented queries or query relaxation.

3.4.4. Metrics

For the user study, we pooled the top-10 ranked results obtained from each technique (including ours) and presented them to the evaluators in random order. Each result was evaluated by 7 anonymous users on the Amazon Mechanical Turk service. The evaluators were required to indicate whether the result was “highly relevant”, “relevant”, “somewhat relevant”, “undecidable”, “irrelevant” or “wrong”. To measure the ranking quality of each technique, we used the Discounted Cumulative Gain (DCG) [45], which is a measure that takes into consideration the rank of relevant documents and allows the incorporation of different relevance levels. DCG is defined as follows

$$DCG(i) = \begin{cases} G(1) & \text{if } i = 1 \\ DCG(i-1) + G(i)/\log(i) & \text{otherwise} \end{cases}$$

where *i* is the rank of the result within the result set, and *G(i)* is the relevance level of the result. We set $G(i)$ to a value between 0 and 5 depending on the evaluator’s assessment. For each result, we averaged the ratings given by all evaluators and used this as the relevance level for the result. Dividing the ob-

Simple queries with relaxation				
Dataset	**OWN**	**WOR**	**BANKS**	**NAGA**
IMDB	**0.880**	0.751	0.777	0.798
LT	**0.876**	0.787	0.721	0.869
Keyword-augmented queries with relaxation				
Dataset	**OWN**	**WOR**	**BANKS**	**NAGA**
IMDB	**0.884**	0.722	0.782	0.776
LT	**0.853**	0.835	0.690	0.782

Table 3.14.: Avg. NDCG for all evaluation queries

tained DCG by the DCG of the ideal ranking we obtained a *Normalized DCG (NDCG)* which accounts for the variance in performance among queries.

3.4.5. Results

The results of our user evaluation are shown in Table 3.14. The reported NDCG values were averaged over all evaluation queries. In the case of simple triple-pattern queries, our ranking model outperforms all competitors for both datasets. In particular, for the IMDB dataset, we achieved more than 17% significant gain in NDCG over WOR with a one-tailed paired t-test p-value of 0.047, and more than 13% over BANKS with a p-value of 0.004. Similarily, for the Librarything dataset, we achieved more than 11% gain in NDCG over WOR with a p-value of 0.078 and more than 21% over BANKS with a p-value of 0.013.

The effectiveness of our ranking model is especially visible in the case of keyword-augmented queries. Our ranking approach again outperforms all competitors for both datasets. For example, for the IMDB dataset, We achieved a gain of more than 22% over WOR with a p-value of 0.020 and 13% over BANKS with a p-value of 0.002.

Analysis. We outperformed WOR since it supports only entity ranking, and thus does not take into consideration the relations between the results. It also supports keyword queries only, and the ranking is based on term-frequencies. This is in contrast to our approach that treats the triples holistically rather than as a set of terms. This indicates that when the objective is to rank tuples of triples, it is better to treat triples holistically, rather than as a combination of entities.

For BANKS, we adapted their technique to our setting, but their ranking still depended on the weights of the edges and nodes in the results. Even weighting edges using our witness counts proved to be insufficient to deliver better quality results.

Our closest competitor in terms of setting and technique is NAGA. Even though, NAGA returns a ranked list of result tuples to triple-pattern queries, it supports neither keyword-augmentation nor query relaxation. Thus, while many simple queries had similar result lists, when testing our technique as a whole with relaxation, NAGA's techniques failed to give effective results. Similarly, the lack of explicit support for keywords in NAGA resulted in less effective ranking for keyword-augmented queries.

Examples. In Table 3.15 we show some example evaluation queries over the IMDB dataset. For each query, the top-3 results returned by our own approach, as well as the 3 other competitors are given. We just show the variable mappings (i.e., the substitutions of the variables in the queries). Next to each result, we also show the average relevance value given by the evaluators (column titled *Rel.*). Recall that each result was given a relevance value between 0 and 5, with 5 corresponding to highly relevant results.

For query Q1 asking for thriller movies, the top-3 results returned by our approach (OWN) are all well-known movies, as compared to both WOR and BANKS. The results returned by BANKS are not even thriller movies. This is due to the fact that even though BANKS ranking can be adapted to approximate matches, the way approximate and exact matching are later aggregated only depends on the static properties of the result graphs (i.e., edges and nodes weights).

The top-3 results returned by NAGA are the same as the ones returned by our approach since both methods rely on witness counts to estimate ranking probabilities. The superiority of our method over NAGA is more clear in the case of keyword-augmented queries and in the case where there are not enough exact matches to the given query. For example, query Q2 (augmented with keywords "true story") asks for movies based on a "true story" and directed by an award winning director. Our top-3 results are all movies based on a true story, which

Q1	**A thriller movie and its director**			
Rank	**OWN**	**Rel.**	**WOR**	**Rel.**
1	Batman_Begins, Christopher_Nolan	4	Deadly_Intruder, John_McCauley	3.43
2	Murder!, Alfred_Hitchcock	4	Robotix, Wally_Burr	3
3	Spider-Man_3, Sam_Raimi	3.71	Like_Minds, Gregory_J._Read	2.71
Rank	**BANKS**	**Rel.**	**NAGA**	**Rel.**
1	-30-, Jack_Webb	2.86	Batman_Begins, Christopher_Nolan	4
2	Kill!, Kihachi_Okamoto	2.57	Murder!, Alfred_Hitchcock	4
3	If...., Lindsay_Anderson	2	Spider-Man_3, Sam_Raimi	3.71
Q2	**An award winning director who directed a movie that is based on a [true story]**			
Rank	**OWN**	**Rel.**	**WOR**	**Rel.**
1	Martin_Scorsese, Good_fellas	3.14	Joel_Lamangan, Babangon_Ako't_Dudurugin_Kita	0
2	Steven_Spielberg, Schindler's_List	3.43	Tracy_Seretean, Big_Mama	3.14
3	Peter_Jackson, Heavenly_Creatures	3.29	Ben_Burtt, The_American_Gangster	2.71
Rank	**BANKS**	**Rel.**	**NAGA**	**Rel.**
1	Baz_Luhrmann, Australia	3.29	Clint_Eastwood , Million_Dollar_Baby	2.86
2	Woody_Allen, Vicky_Cristina_Barcelona	3.57	Eddie_Murphy, Harlem_Nights	2.71
3	Christopher_Nolan, Batman_Begins	2.14	Robert_Altman, Health	3
Q3	**An academy awarded movie produced in Australia**			
Rank	**OWN**	**Rel.**	**BANKS**	**Rel.**
1	Secrets_of_the_Heart, Academy_Award, Australia	3.71	Chiranjeevi, Padma_Bhushan, India	1.57
2	Before_Sunset, Academy_Award, USA	2.57	Three_Seasons, Independent_Spirit_Award, UK	1.43
3	Innerspace, Academy_Award, USA	1.86	Charlie_Chaplin, Academy_Honorary_Award, India	2

Table 3.15.: Examples of IMDB queries and top-ranked results

is not the case in any of the other 3 result lists returned by the competitors.

Query Q3 illustrates the benefit of query relaxation. The query asks for movies produced in Australia and that won an academy award. In our dataset, there is only one exact match to this query, namely, “Secrets of the Heart”. We only show the top results for the two approaches that support query relaxation: our own approach and BANKS. We also show the award the movie won and the production country, which are variable mappings for the relaxed queries. Recall that a query is relaxed by replacing one of the constants (i.e., either `Australia` or `Academy_Award` by a variable). Our method ranks the exact match at the top, while approximate matches which includes movies which have won an `Academy_Award`, but were produced in another country, or movies produced in Australia but have won other awards are ranked lower. This is in contrast to the list returned by BANKS.

3.5. Summary

In this chapter, we have shown how to concisely search RDF knowledge bases using triple-pattern queries. We have also shown how we can extend the expressiveness of such queries by allowing keyword conditions in combination with structured triple patterns. To improve the recall of triple-pattern queries, we proposed a query relaxation paradigm that automatically relaxes one or more triple patterns in a given triple-pattern query. We presented a general model to rank the results of triple-pattern queries, whether keyword-augmented or not. Our model is based on statistical language models and seamlessly handles query relaxation. We have shown the superiority of our ranking model as compared to the-state-of-the-art ranking models for queries over structured data through a comprehensive user study.

Chapter 4.

Query Reformulation for Triple-Pattern Search

Query reformulation is the process of modifying a search query to address the same information need intended by the modified query. Both users and search engines perform query reformulation to improve the search quality. In this chapter, we study the problem of *automatic* query reformulation for triple-pattern search. That is, given a triple-pattern query, we try to generate reformulations of the given query in order to retrieve more search results without unduly sacrificing precision. This can be seen as the counterpart to query expansion in traditional IR.

4.1. Types of Query Reformulations

Searching RDF knowledge bases is usually done by means of triple-pattern queries. Recall that a triple-pattern query consists of triple patterns where a triple pattern is a triple with at least one variable. For example, consider the following query which consists of four triple patterns, and asks for drama movies that have won an Academy Award and were directed by directors born in Canada:

```
?m  hasGenre     Drama
?m  hasWonPrize  Academy_Award
?d  directed     ?m
?d  wasBornIn    Canada
```

Also recall that we associate with each triple in our knowledge base a text snippet, which contains any contextual text from the sources the triples were extracted from. This way, we can also specify keyword conditions in a keyword-augmented triple-pattern query. For example, the second triple pattern in the above query can be augmented with the keywords *car accident* as follows:

```
?m  hasGenre     Drama             car accident
?m  hasWonPrize  Academy_Award
?d  directed     ?m
?d  wasBornIn    Canada
```

Given a query with n triple patterns, the result of the query is the set of all tuples of n triples that instantiate the query triple patterns and satisfy the query join and keyword conditions. For example, one result for both example queries is the 4-tuple (i.e., tuple of four triples):

```
Crash         hasGenre     Drama
Crash         hasWonPrize  Academy_Award
Paul_Haggis   directed     Crash
Paul_Haggis   wasBornIn    Canada
```

Due to their specificity, the above example queries would return very few results even on large movie collections. However, if the system were able to automatically reformulate the query, say replacing the predicate `hasWonPrize` in the second triple pattern with `wasNominatedFor`, the system would potentially return a larger number of results.

We present a comprehensive query reformulation framework, where the query reformulations can be derived from the RDF knowledge base the queries are issued against as well as from external ontological and textual sources. Our framework is based on statistical language models and provides a principled basis for generating query reformulations. Moreover, we develop a model to holistically merge and rank the results of the original query and all its reformulations. Automatic reformulation of queries in a robust way so that they would not suffer from topic drifts (e.g., overly broad generalizations) is a difficult problem [5].

Our query reformulation framework consists of the following three types of reformulations:

- Resources specified in triple patterns, whether as subjects, predicates or objects, can be reformulated by substituting them with related resources. For example, `Academy_Award` can be replaced by `Golden_Globe` or `BAFTA_Award`.

- Resources in triple patterns can be substituted by variables. For example, `Drama` can be replaced by `?x` to cover arbitrary genres or `directed` can be replaced by `?y` to retrieve movies with Canadian actors or producers instead of Canadian directors.
- Triple patterns from the entire query can be entirely removed. For example, the triple pattern

  ```
  ?d  bornIn  Canada
  ```

  can be entirely removed from the query, thus increasing the number of movies returned.

4.2. Query Reformulation Framework

Given a query $Q = (q_1, q_2, ..., q_n)$ where q_i is a triple pattern, a reformulated query is generated by reformulating one or more of its triple patterns or by entirely removing one or more triple patterns. In turn, a triple pattern q_i is reformulated by substituting one or more resources (i.e., entities or relations) specified in the triple pattern with similar resources or with variables.

In order to do so, we associate with each resource in our knowledge base KB a list of candidate *substitutions*. The substitution list of a resource consists of other resources from the knowledge base ordered by their similarity to the given resource. In order to do this, we need to compute the similarity between two given resources X and Y. We first provide a representation model for resources. We then show how, using this representation model, the similarity between two resources can be computed and how the substitution lists for the resources in our knowledge base are constructed. Finally, we explain how these lists can be used to reformulate triple-pattern queries, possibly augmented with keywords.

4.2.1. Resource Representation Model

For each resource in the knowledge base, we assume there exists a *language model*. To construct the resources' language models, there are at least three different sources of information which we can leverage. First, we have the knowledge base itself – this is also the primary source of information in our case since we are processing our queries on the knowledge base. Second, we can make use of the textual snippets of the triples or any other external textual sources associated with the triples (for example, the sources from which the triples were extracted). Third, we can also utilize external ontologies in order to find semantic descriptions of resources.

We first focus on utilizing the knowledge base as the source of information used to represent a resource and then show how our representation model can be easily extended to incorporate other information sources.

Before we describe our language model construction procedure for resources, we make the distinction between two types of resources: *entities* and *relations*. We make this distinction since both types of resources are inherently different. Entities typically appear as subjects or objects of RDF triples in the knowledge base whereas relations are predicates that express binary relationships between the subjects and objects of the triples.

Entity Representation

Vocabulary. To represent entities in our model, we use two types of bags (multisets) of terms: *unigram bags* and *bigram bags* which are defined next.

Definition 4.1 : Unigram Bag
The unigram bag $U(X)$ *of entity* X *is the bag:* $\{s|t = (s, p, o) \in KB \wedge o = X\} \cup \{o|t = (s, p, o) \in KB \wedge s = X\}$.

That is, the unigram bag of entity X is the union of the bag of subjects of all triples in the knowledge with object X and the bag of objects of all the triples in the knowledge base with subject X.

Definition 4.2 : Bigram Bag
The bigram bag $B(X)$ *of entity* X *is the bag:* $\{(s, p)|t = (s, p, o) \in KB \wedge o = X\} \cup \{(p, o)|t = (s, p, o) \in KB \wedge s = X\}$.

That is, the bigram bag of entity X is the union of the bag of subject-predicate pairs of all triples in the knowledge base with object X and the bag of all predicate-object pairs of all triples in the knowledge base with subject X.

For example, let the set of all triples whose subject or object is the entity `Woody_Allen` be:

Woody_Allen	directed	Manhattan
Woody_Allen	directed	Match_Point
Woody_Allen	actedIn	Scoop
Woody_Allen	type	American_Director
Federico_Fellini	influences	Woody_Allen

The unigram bag of the entity `Woody_Allen` would then be: {`Manhattan`, `Match_Point`, `Scoop`, `American_Director`, `Federico_Fellini`}. The bigram bag of the entity `Woody_Allen` would be: {(`directed`, `Manhattan`), (`directed`, `Match_Point`), (`actedIn`, `Scoop`), (`type`, `American_Director`), (`Federico_Fellini`, `influences`)}.

Note that the bigrams usually occur exactly once per entity, but it is still important to use them for entity representation. We illustrate this via an example. Let the bigram bag of a given entity contain the bigram (`hasWonPrize`, `Academy_Award`). This is different from the bigram bag containing the bigram (`nominatedFor`, `Academy_Award`). This distinction cannot be made if only unigrams are considered.

Now, we are ready to explain how we construct language models for all entities in our knowledge base.

Entity Language Models. We assume there exists three types of language models for each entity in our knowledge base: a unigram language model, a bigram language model and an overall entity language model. The three types of language models are defined next.

Definition 4.3 : Unigram Language Model
Let $U(KB)$ be the union of all the unigram bags of all the entities in the knowledge base KB. The unigram language model of an entity X is a probability distribution over all the terms $w \in U(KB)$. The unigram language model of resource X has n_u parameters $P_u(w|X)$ where n_u is the number of distinct terms in $U(KB)$ and $P_u(w|X)$ is the probability of the term w in the unigram language model of entity X.

The parameters of the unigram language model of entity X are estimated from $U(X)$ using a maximum-likelihood estimator which is smoothed by interpolating with a background (collection) language model estimated from $U(KB)$ as follows:

$$P_U(w|X) = \alpha\frac{c(w, U(X))}{|U(X)|} + (1-\alpha)\frac{c(w, U(KB))}{|U(KB)|} \tag{4.1}$$

where $c(w, U(X))$ and $c(w, U(KB))$ are the frequencies of occurrences of w in $U(X)$ and $U(KB)$ respectively, $|U(X)|$ and $|U(KB)|$ are the lengths of the bags $U(X)$ and $U(KB)$ respectively (i.e., total frequencies of all unigrams in $U(X)$ and $U(KB)$ respectively) and $\alpha \in [0, 1]$ is a smoothing parameter.

Definition 4.4 : Bigram Language Model
Let $B(KB)$ *be the union of all the bigram bags of all the entities in the knowledge base* KB*. The bigram language model of an entity* X *is a probability distribution over all the terms* $w \in B(KB)$*. The bigram language model of resource* X *has* n_B *parameters* $P_B(w|X)$ *which is the probability of the term* w *in the bigram language model of entity* X.

The parameters of the bigram language model is estimated in an analogous manner to the unigram language model as follows:

$$P_B(w|X) = \alpha\frac{c(w, B(X))}{|B(X)|} + (1-\alpha)\frac{c(w, B(KB))}{|B(KB)|} \tag{4.2}$$

where $c(w, B(X))$ and $c(w, B(KB))$ are the frequencies of occurrences of w in $B(X)$ and $B(KB)$ respectively, $|B(X)|$ and $|B(KB)|$ are the lengths of the bags $B(X)$ and $B(KB)$ respectively (i.e., total frequencies of all bigrams in $B(X)$ and $B(KB)$ respectively) and $\alpha \in [0, 1]$ is a smoothing parameter.

Finally, the overall entity language model is defined as follows.

Definition 4.5 : Entity Language Model
The language model of an entity X *is a probability distribution over all the terms* $w \in U(KB) \cup B(KB)$*. The language model of entity* X *has* $n_U + n_B$ *parameters* $P(w|X)$ *which is the probability of the term* w *in the language model of the entity* X.

The probability $P(w|X)$ of a term w in the language model of entity X is estimated as a weighted sum of the following two probabilities:

$$P(w|X) = \mu P_U(w|X) + (1-\mu)P_B(w|X) \tag{4.3}$$

where $P_U(w|X)$ is the probability of term w in the unigram language model of X which is zero if $w \notin U(KB)$. Similarly, $P_B(w|X)$ is the probability of term w in the bigram language model of X which is zero if $w \notin B(KB)$. Finally, the parameter μ weights the two probabilities and should be learnt using training data.

Relation Representation

Vocabulary. To represent relations in our model, we use three types of bags of terms: *subject bags*, *object bags* and *bigram bags* which are defined next. Note that unlike entities, we make a distinction between subjects and objects since relations are directional.

Definition 4.6 : Subject Bag
The subject bag $S(X)$ *of relation* X *is the bag:* $\{s|t = (s, p, o) \in KB \wedge p = X\}$.

That is, the subject bag of relation X is the bag of subjects of all triples with predicate X.

Definition 4.7 : Object Bag
The object bag $O(X)$ *of relation* X *is the bag:* $\{o|t = (s, p, o) \in KB \wedge p = X\}$.

That is, the object bag of relation X is the bag of objects of all triples with predicate X.

Definition 4.8 : Bigram Bag
The bigram bag $B(X)$ *of relation* X *is the bag:* $\{(s, o)|t = (s, p, o) \in KB \wedge p = X\}$.

That is, the bigram bag of relation X is the bag of subject-object pairs of all triples with predicate X.

For example, let the set of all triples whose predicate is the relation `directed` be:

```
James_Cameron  directed  Aliens
Woody_Allen    directed  Manhattan
Woody_Allen    directed  Match_Point
Sam_Mendes     directed  American_Beauty
```

The subject bag of the relation `directed` would be: {`James_Cameron`, `Woody_Allen`, `Woody_Allen`, `Sam_Mendes`}. The object bag of the relation `directed` would be: {`Aliens`, `Manhattan`, `Match_Point`, `American_Beauty`}. Finally, the bigram bag of the relation `directed` would be: {(`James_Cameron`, `Aliens`) , (`Woody_Allen`, `Manhattan`), (`Woody_Allen`, `Match_Point`), (`Sam_Mendes`, `American_Beauty`)}.

Now, we are ready to explain how we construct language models for all relations in our knowledge base.

Relation Language Models. We assume there exists four types of language models for each relation in our knowledge base: a subject language model, an object language model, a bigram language model and an overall relation language model. The four types of language models are defined next.

Definition 4.9 : Subject Language Model
Let $S(KB)$ *be the union of all the subject bags of all the relations in the knowledge base* KB. *The subject language model of relation* X *is a probability distribution over all the terms* $w \in S(KB)$. *The subject language model of relation* X *has* n_S *parameters* $P_S(w|X)$ *where* n_S *is the number of distinct terms in* $S(KB)$ *and* $P_S(w|X)$ *is the probability of term* w *in the subject language model of relation* X.

The parameters of the subject language model of relation X are estimated from $S(X)$ using a maximum-likelihood estimator that is smoothed by interpolating with a background (collection) language model estimated from $S(KB)$ as follows:

$$P_S(w|X) = \alpha \frac{c(w, S(X))}{|S(X)|} + (1-\alpha)\frac{c(w, S(KB))}{|S(KB)|} \tag{4.4}$$

where $c(w, S(X))$ and $c(w, S(KB))$ are the frequencies of occurrences of w in $S(X)$ and $S(KB)$ respectively, $|S(X)|$ and $|S(KB)|$ are the lengths of the bags $S(X)$ and $S(KB)$ respectively (i.e., total frequencies of all unigrams in $S(X)$ and $S(KB)$ respectively) and $\alpha \in [0, 1]$ is a smoothing parameter.

Definition 4.10 : Object Language Model
Let $O(KB)$ *be the union of all the object bags of all the relations in the knowledge base* KB. *The object language model of relation* X *is a probability distribution over all the terms* $w \in O(KB)$. *The object language model of relation* X *has* n_O *parameters* $P_O(w|X)$ *where* n_O *is the number of distinct terms in* $O(KB)$ *and* $P_O(w|X)$ *is the probability of term* w *in the object language model of relation* X.

The parameters of the object language model of relation X are also estimated from $O(X)$ using a maximum-likelihood estimator after smoothing with a background language model estimated from $O(KB)$ as follows:

$$P_O(w|X) = \alpha \frac{c(w, O(X))}{|O(X)|} + (1-\alpha)\frac{c(w, O(KB))}{|O(KB)|} \tag{4.5}$$

where $c(w, O(X))$ and $c(w, O(KB))$ are the frequencies of occurrences of w in $O(X)$ and $O(KB)$ respectively, $|O(X)|$ and $|O(KB)|$ are the lengths of the bags

$O(X)$ and $O(KB)$ respectively (i.e., total frequencies of all unigrams in $O(X)$ and $O(KB)$ respectively) and $\alpha \in [0, 1]$ is a smoothing parameter.

Definition 4.11 : **Bigram Language Model**
Let $B(KB)$ *be the union of all the bigram bags of all the relations in the knowledge base* KB. *The bigram language model of relation* X *is a probability distribution over all the terms* $w \in B(KB)$. *The bigram language model of relation* X *has* n_B *parameters* $P_B(w|X)$ *where* n_B *is the number of distinct terms in* $B(KB)$ *and* $P_B(w|X)$ *is the probability of term* w *in the bigram language model of relation* X.

Analogous to the estimation procedure for the subject and object language models, the parameters of the bigram language model are estimated as follows:

$$P_B(w|X) = \alpha\frac{c(w, B(X))}{|B(X)|} + (1 - \alpha)\frac{c(w, B(KB))}{|B(KB)|} \tag{4.6}$$

where $c(w, B(X))$ and $c(w, B(KB))$ are the frequencies of occurrences of w in $B(X)$ and $B(KB)$ respectively, $|B(X)|$ and $|B(KB)|$ are the lengths of the bags $B(X)$ and $B(KB)$ respectively (i.e., total frequencies of all bigrams in $B(X)$ and $B(KB)$ respectively) and $\alpha \in [0, 1]$ is a smoothing parameter.

Finally, for each relation X, we assume there exists an overall language model which is defined as follows.

Definition 4.12 : **Relation Language Model**
The language model of a relation X *is a probability distribution over the terms* $w \in S(KB) \cup O(KB) \cup B(KB)$. *The language model of the relation* X *has* $n_S + n_O + n_B$ *parameters* $P(w|X)$ *which is the probability of the term* w *in the language model of the relation* X.

The probability $P(w|X)$ of term w in the language model of relation X is estimated as a weighted sum of the following three probabilities:

$$P(w|X) = \mu_S P_S(w|X) + \mu_O P_O(w|X) + (1 - \mu_S - \mu_O)P_B(w|X) \tag{4.7}$$

where $P_S(w|X)$ is the probability of term w in the subject language model of relation X which is zero if $w \notin S(KB)$, $P_O(w|X)$ is the probability of term w in the object language model of relation X which is zero if $w \notin O(KB)$ and $P_B(w|X)$ is the probability of term w in the bigram language model of relation X which is zero if $w \notin B(KB)$. Finally, the parameters μ_S and μ_O weight the three probabilities and should be learnt using training data.

Representing Resources Using Multiple Information Sources

The core of our technique lies in constructing a language model for each resource X, whether an entity or a relation. That is, given an information source, it is sufficient to describe: i) the vocabulary the language models are defined over and ii) how the language models are estimated. We have described above these two steps when the information source is the RDF knowledge base. It is easy to extend the same method for other sources. For example, we could make use of the text snippets the triples are associated with. Then the vocabulary of the language models will be the set of all keywords in all the text snippets in the knowledge base and the language models can be estimated from the text snippets of the triples using maximum-likelihood estimators.

Once individual language models have been estimated for a resource from each information source, a straight-forward method to combine them into a single language model is to use a mixture model of all the language models. The parameters of the mixture model can be set based on the importance of each source. Note that this method does not preclude having different subsets of sources for different resources.

4.2.2. Substitution Lists

We have explained how to construct a language model for each resource X, whether an entity or a relation. Now, we associate each resource X in our knowledge with a list of candidate substitutions which is defined as follows.

Definition 4.13 : Substitution List
Given a resource X*, a substitution list* L *consists of a set of resources* Y *which are ordered by their similarity to the given resource.*

We first explain how to compute the similarity between two given resources X and Y and then explain how we construct the substitution lists.

Similarity between Resources

The similarity between two resources X and Y is computed as the distance between their language models. Specifically, we use the square-root of the Jensen-Shannon divergence (JS divergence) between the language models of the two resources X and Y, which is a *metric*, to measure the distance between the two resources. The JS divergence is defined as follows.

Definition 4.14 : Jensen-Shannon Divergence
The Jensen-Shannon divergence between two probability distributions P and Q, is a symmetric measure of the distance between two probability distributions.

Given two probability distributions P and Q, the JS divergence between them is computed as follows:

$$JS(P||Q) = KL(P||M) + KL(Q||M) \tag{4.8}$$

where $KL(R||S)$ is the Kullback-Leibler divergence (KL divergence) between two probability distributions R and S, which is computed as follows:

$$KL(R||S) = \Sigma_w R(w) \log \frac{R(w)}{S(w)} \tag{4.9}$$

and

$$M = \frac{1}{2}(P + Q) \tag{4.10}$$

We use the square root of the JS divergence since it is a metric between 0 and 1, and thus it can be used to measure the similarity between two resources.

Substitution Lists Construction

We have so far shown how to represent a resource and how to measure the similarity between two resources. To recap, for each resource X in the knowledge base KB, we construct a language model. The similarity between two resources X and Y is then computed as the distance between the language models of the two resources. Specifically, we use the square-root of the Jensen-Shannon divergence (JS divergence) between the two language models. Now, a substitution list for a resource X is a simply a ranked list of other resources, ranked based on the square-root of the JS divergence between their language models and the language model of resource X.

Adding Variables to Substitution Lists

Recall that a triple-pattern query can be reformulated by replacing one of the resources that appear in it with a variable. We interpret replacing a resource

Academy_Award_for_Best_Actor	**Thriller**
BAFTA_Award_for_Best_Actor	Crime
Golden_Globe_Award_for_Best_Actor_Drama	Horror
var	Action
Golden_Globe_Award_for_Best_Actor_Musical_or_Comedy	Mystery
New_York_Film_Critics_Circle_Award_for_Best_Actor	*var*
directed	**bornIn**
actedIn	livesIn
created	originatesFrom
produced	*var*
var	diedIn
type	isCitizenOf

Table 4.1.: Example resources and their top-5 substitutions

with a variable as being equivalent to replacing that resource with *any* other resource in the knowledge base.

To handle variable substitutions, we interpret replacing a resource X with a variable as replacing X with any other resource in the knowledge base. To carry this out, we construct a special language model for all other resources in the knowledge base which is a mixture model of all the language models of all the resources in the knowledge base other than X. The similarity between the resource X and a variable is then computed using the square-root of the JS divergence between the language model of the resource X and the special language model corresponding to all other resources in the knowledge base. Using this technique, a variable is now simply another entry in the substitution list of resource X, .

Table 4.1 shows example resources from an RDF knowledge base about movies. For each resource, it shows the top-5 substitutions from the resource substitution list. The entry *var* represents the variable substitution. As previously explained, a variable substitution indicates that there were no other *specific* substitutions which had a higher similarity to the given resource.

Pruning the Substitution Lists

Maintaining a substitution list for every resource in the knowledge base can be very impractical when these lists are long. Recall that for a given resource, its substitution list contains all other resources in the knowledge base and their similarities to the given resource. These lists can thus be extremely long in large knowledge bases. Pruning such lists is thus crucial to avoid storage bottleneck, as these lists need to be maintained somewhere in the knowledge base. Pruning can also be beneficial for efficient query processing since our query reformulation algorithm described next scans such lists to generate reformulated queries, which would then be evaluated. Pruning can limit the number of such reformulated queries to a reasonable number.

The most basic way to prune substitution lists is to use a pruning threshold. That is, reduce the list and cut off its tail whenever the similarity score between the substitution and the resource the list belongs to becomes less than a predefined threshold. In our framework, we use the score of the variable substitution as the threshold value after which we prune the lists. More precisely, any substitution ranked below the variable substitution is pruned and removed from the substitution list. As an example, consider the substitution list for the relation `directed` shown in Table 4.1. This substitution list can be pruned after the fourth entry. This seems to be very intuitive since substitutions beyond the variable substitution can be seen as very dissimilar from the given resource, that they may as well be ignored and represented by the variable substitution.

4.2.3. Generating Reformulated Queries

Our query reformulation framework associates with each resource, whether an entity or a relation, a substitution list consisting of substitution candidates and their similarities to the resource the list belongs to. Recall that a substitution candidate is either a resource or a variable. The similarity between a resource X and a substitution Y, is measured as the square root of the JS divergence between the language models of the two resources X and Y. We now show how, given a triple-pattern query (possibly augmented with keywords), we can generate a set of reformulated queries using the substitution lists of resources. A reformulated query is defined as follows.

Definition 4.15 *: **Reformulated Query***
Given a query $Q = (q_1, q_2, ..., q_n)$ *where* q_i *is a triple pattern, let* $VAR(Q)$ *be the set of variables that appear in* Q*. Let* $VAR_i \subset VAR$ *be a set of infinite variables corresponding to triple pattern* q_i *such that* $VAR_1, VAR_2, ..., VAR_n$ *are all pairwise disjoint and* $\forall\ 1 \leq i \leq n, VAR_i \cap VAR(Q) = \phi$*. Let* $RES(q_i)$ *be the set of resources that appear in triple pattern* q_i*. Let* $r(q_i)$ *be the set of reformulated triple patterns obtained by replacing one or more resources* $X_i \in RES(q_i)$ *with another resource* $Y_i \in KB$ *or replacing a resource* $X_i \in RES(q_i)$ *with a variable* $var_i \in VAR_i$*. The set of all reformulated queries* $R(Q)$ *is then:* $\{r(q_1) \cup \{q_1\} \times r(q_2) \cup \{q_2\} \times ... \times r(q_n) \cup \{q_n\}\}$.

We now present a general algorithm that can be used to reformulate both triple patterns, and overall queries using the substitution lists of the resources.

Query Reformulation Algorithm

Algorithm 1 can be used to reformulate both triple patterns and overall queries. We consider the case of triple patterns first. Let the triple pattern we want to reformulate be q with subject s, predicate p and object o. Let L_s, L_p and L_o be the substitution lists of s, p and o respectively, where $L_s = \phi$ if s is a variable, $L_p = \phi$ if p is a variable and $L_o = \phi$ if o is a variable. The substitution list of a resource X contains substitutions Y ordered in ascending order of the square root of the JS divergence between the resource X and the substitution Y which we denote by $\delta(X, Y)$. Note that $\delta(X, X) = 0$ and we further assume that $\delta(var, var) = 0$ where var is any variable. To reformulate the triple pattern q, we call $REFORMULATE(s, p, o, L_s, L_p, L_o)$. Algorithm 1 starts at the first position of all the substitution lists L_s, L_p and L_o and scans them iteratively generating a set of *new* reformulated triple patterns $q' = (s', p', o')$ where $s' = s$ or s' is one of the substitutions seen so far from list L_s, and similarly for p' and o'. The algorithm uses a priority queue to maintain the list of reformulations generated so far and at the end of each iteration, it removes the reformulated triple pattern q' with the minimum score $\delta(q, q')$ from the priority queue and adds it to the list of reformulations L. The score of a reformulated triple pattern q' is computed as follows:

$$\delta(q, q') = \delta(s, s') + \delta(p, p') + \delta(o, o') \tag{4.11}$$

The score $\delta(q, q')$ represents how close the reformulated triple pattern q' is to the original triple pattern q. The algorithm terminates when all substitution

Algorithm 1 REFORMULATE($x_1, x_2, ..., x_n, L_1, L_2, .., L_n$)

```
1:  for ($1 \leq i \leq n$) do
2:      $Seen_i \leftarrow \{x_i\}$
3:      $current_i \leftarrow 1$
4:  end for
5:  allocate a list of reformulations $L$
6:  $L.insert((x_1, x_2, ..., x_n))$
7:  allocate priority queue $Queue$
8:  while ($current_1 \leq |L_1|$ OR $current_2 \leq |L_2|$ OR .... OR $current_n \leq |L_n|$) do
9:      for ($1 \leq i \leq n$) do
10:         if ($L_i \neq null$ AND $current_i \leq |L_i|$) then
11:             $Seen_i \leftarrow Seen_i \cup \{L_i(current_i)\}$
12:         end if
13:     end for
14:     for ($(y_1, y_2, ..., y_n) \in Seen_1 \times Seen_2 \times ... \times Seen_n$) do
15:         $\delta((x_1, x_2, ..., x_n), (y_1, y_2, ..., y_n)) = \Sigma_{i=1}^{n} \delta(x_i, y_i)$
16:         if ($(y_1, y_2, ..., y_n) \notin L$ AND $(y_1, y_2, ..., y_n) \notin$ Queue) then
17:             Queue.insert($(y_1, y_2, ..., y_n)$)
18:         end if
19:     end for
20:     $(y_1, y_2, ..., y_n) \leftarrow$ Queue.head()
21:     $L.insert((y_1, y_2, ..., y_n))$
22:     for ($1 \leq i \leq n$) do
23:         if ($L_i \neq null$ AND $y_i = L_i(current_i)$) then
24:             $current_i \leftarrow current_i + 1$
25:         end if
26:     end for
27: end while
28: while (Queue is not empty) do
29:     $(y_1, y_2, ..., y_n) \leftarrow$ Queue.head()
30:     $L.insert((Y_1, Y_2, ..., Y_n))$
31: end while
32: return $L$
```

directed	**hasGenre**	**Thriller**
actedIn: 0.413	type: 0.497	Crime: 0.466
created: 0.418	var: 0.525	Horror: 0.468
produced: 0.438		Action: 0.477
var: 0.472		Mystery: 0.495
		var: 0.503

Table 4.2.: Substitution lists for three resources and the score of each substitution

lists have been completely scanned and all possible triple pattern reformulations have been generated (i.e., the priority queue is also empty). After the algorithm terminates, the list L will contain all the reformulations of triple pattern q sorted in ascending order of their scores as computed according to Equation 4.11.

For example, consider the query asking for thriller movies and their directors which can be expressed using the following 2 triple-pattern query:

```
?d  directed  ?m
?m  hasGenre  Thriller
```

The above query consists of the following three resources: `directed`, `hasGenre` and `Thriller`. Table 4.2 shows the *pruned* substitution lists L_1, L_2 and L_3 for the three resources `directed`, `hasGenre` and `Thriller`, respectively. For each resource, the substitution list contains a ranked list of substitutions, which are pruned after the variable substitution *var* and the substitution score δ which is the square root of the JS divergence between the language model of the resource and the substitution. Table 4.3 shows the reformulated triple patterns generated using our triple pattern reformulation algorithm and their scores which are computed according to Equation 4.11. The substitutions in each reformulated pattern are underlined.

Algorithm 1 can also be used to generate reformulated queries. Given a query $Q = (q_1, q_2, ..., q_n)$, we first reformulate each triple pattern by executing Algorithm 1. Let the output of executing Algorithm 1 for triple pattern q_i be L_i. L_i would then contain a list of reformulated triple patterns $q_i^0, q_i^1, .., q_i^{m_i}$ where q_i^0 is the original triple pattern q_i and the list is sorted based on the reformulations scores $\delta(q_i, q_i^j)$. We then call $\mathsf{REFORMULATE}(q_1, q_2, ..., q_n, L_1, L_2, ..., L_n)$ and the

?d	**directed**	**?m**	δ	**?m**	**hasGenre**	**Thriller**	δ
?d	actedIn	?m	0.413	?m	hasGenre	Crime	0.466
?d	created	?m	0.418	?m	hasGenre	Horror	0.468
?d	produced	?m	0.438	?m	hasGenre	Action	0.477
?d	?x	?m	0.472	?m	hasGenre	Mystery	0.495
				?m	type	Thriller	0.497
				?m	hasGenre	?y	0.503
				?m	?y	Thriller	0.525
				?m	type	Crime	0.963
				?m	type	Horror	0.965
				?m	type	Action	0.974
				?m	?y	Crime	0.991
				?m	type	Mystery	0.992
				?m	?y	Horror	0.993
				?m	type	?y	1.000
				?m	?y	Action	1.002
				?m	?y	Mystery	1.020
				?m	?y	?z	1.028

Table 4.3.: Reformulated triple patterns and their scores

output L would be the list of all reformulated queries $Q^j = (q_1^j, q_2^j, ..., q_n^j)$ of the query Q sorted in ascending order of their scores $\delta(Q, Q')$ which is computed as follows:

$$\delta(Q, Q^j) = \Sigma_{i=1}^{n} \delta(q_i, q_i^j) \tag{4.12}$$

Table 4.4 shows the top-10 reformulated queries for our example query generated by running Algorithm 1. We write both triple patterns in one line separated by a semicolon and next to each reformulated query, the score of the reformulation δ. Note that our algorithm can be used to retrieve only the top-k closest reformulations (of a triple pattern and/or a query).

Removing Triple Patterns. Recall that one type of reformulation our framework supports is removing one or more triple patterns specified in a query. We only remove triple patterns that contain only variables. Using our reformulation

?d directed ?m; ?m hasGenre Thriller	δ
?d <u>actedIn</u> ?m; ?m hasGenre Thriller	0.413
?d <u>created</u> ?m; ?m hasGenre Thriller	0.418
?d <u>produced</u> ?m; ?m hasGenre Thriller	0.438
?d directed ?m; ?m hasGenre <u>Crime</u>	0.466
?d directed ?m; ?m hasGenre <u>Horror</u>	0.468
?d <u>?x</u> ?m; ?m hasGenre Thriller	0.472
?d directed ?m; ?m hasGenre <u>Action</u>	0.477
?d directed ?m; ?m hasGenre <u>Mystery</u>	0.495
?d directed ?m; ?m <u>type</u> Thriller	0.497
?d directed ?m; ?m hasGenre <u>?x</u>	0.503

Table 4.4.: Top-10 reformulated queries for a given example query and their scores

algorithm described above, some of the reformulated queries generated would contain triple patterns consisting of only variables. In such cases, we consider removing such triple patterns provided that they do not result in a *disconnected* set of triple patterns. For example, consider the query asking for comedy movies with British actors:

```
?m  hasGenre  Comedy
?a  actedIn   ?m
?a  bornIn    UK
```

and consider the following reformulated query:

```
?m  hasGenre  Comedy
?a  ?x        ?m
?a  bornIn    UK
```

Even though the second triple pattern contains only variables, retaining this pattern in the query is still crucial – it states that `?a` is related some how to the movie `?m`. Removing this triple pattern from the query would result in a disconnected query, that asks for comedy movies and actors born in the UK.

Now, consider the following reformulation of the same query:

```
?m  hasGenre  Comedy
?a  actedIn   ?m
?a  ?x        ?y
```

We can safely remove the third triple pattern in which case the reformulated query would still be meaningful – find comedy movies and their actors.

Reformulating Keywords. Our query reformulation framework reformulates queries by replacing resources specified in a triple-pattern query, whether keyword augmented or not, with similar resources, variables or removing triple patterns altogether. We did not provide techniques to reformulate the keywords specified in a keyword-augmented triple-pattern query. There is a wealth of techniques on query expansion for keyword queries and all these methods can be easily incorporated or made use of in our framework. For example, each keyword in our knowledge base can be associated with a set of related keywords [17] such as *conflict* for *war* or *new york city* for *manhattan*, etc. Now, we can extend our query reformulation algorithm to generate reformulated queries so that in addition to substituting resources in the query, the keywords specified in the query are also expanded. Note that query expansion for keyword queries does not necessarily mean adding keywords to the query only, but in principle, means modifying the query in one way or the other, say by removing keywords or substituting keywords with similar ones.

As an example, consider the following query consisting of a single keyword-augmented triple pattern:

`?m  hasGenre  Drama` *manhattan wedding*

We can now generate a set of reformulated queries by substituting the resources `hasGenre` and `Drama` with similar ones or variables, or expanding the keywords *manhattan wedding*. For example, one such reformulated query can be:

`?m  hasGenre  ?x` *new york city wedding*

The above query is a reformulation of our example query where the resource `Comedy` was substituted by the variable `?x` and the keyword *manhattan* was expanded to *new york city*.

4.2.4. Executing Reformulated Queries

The reformulated queries generated by our reformulation algorithm can be presented to the user as *query suggestions* or they can be executed and their results merged and ranked in one way or the other. We present two different modes of execution : an *incremental* mode and a *batch* mode. Both execution modes make use of the ranking model described in Chapter 3 to rank the results of triple-pattern queries.

Incremental Execution.

Let $L = \{Q^0, Q^1, ..., Q^k\}$ be the list of query reformulations obtained using our reformulation algorithm described in the previous subsection, where Q^0 is the original query. In the incremental execution mode, we execute the queries $Q^j \in L$ in order of their scores $\delta(Q^0, Q^j)$ which are defined according to Equation 4.12. That is, we start by executing the original query Q^0, retrieve all its results and rank them according to the ranking model described in Chapter 3. Next, we execute query Q^1, retrieve all its results and rank them according to the same ranking model, and so on. The final result list would be the set of all unique results of all queries $Q^0, Q^1, ..., Q^k$ such that the results of query Q^j are all ranked above the results of query $Q^{(j+1)}$ and the results of each query Q^j are ranked based on the ranking model described in Chapter 3.

Batch Execution. In batch execution mode, we execute the original query and all its reformulations and merge their results into one unified results set eliminating duplicates. We then rank the unified result set using the ranking model described in Chapter 3. Our ranking model assumes there exists a language model for the query which is defined as follows. Given a query $Q = (q_1, q_2, .., q_n)$ where q_i is a triple pattern, the language model of query Q is a probability distribution over all tuples of n triples of the form $T = (t_1, t_2, ..., t_n)$ where t_i is a triple. The probability $P(T|Q)$ of a tuple T in the query language model is then estimated as follows (assuming independence between the triples):

$$P(T|Q) = \prod_{i=1}^{n} P(t_i|q_i) \tag{4.13}$$

Now, assume that triple pattern q_i has the list of reformulations $\{q_i^0, q_i^1, ..., q_i^{m_i}\}$ where q_i^j is a reformulated triple pattern obtained by our reformulation algorithm (q_i^0 is the original triple pattern). The probability $P(t_i|q_i)$ is then estimated as a weighted sum of the following $m_i + 1$ probabilities:

$$P(t_i|q_i) = \lambda_0 P(t_i|q_i^0) + \lambda_1 P(t_i|q_i^1) + ... + \lambda_{m_i} P(t_i|q_i^{m_i}) \tag{4.14}$$

where $P(t_i|q_i^j)$ is the probability of triple t_i in the language model of triple pattern q_i^j which was estimated as described in Chapter 3. The parameters λ_j weigh the contribution of each triple pattern and we set it as a function of the score of the reformulation $\delta(q_i, q_i^j)$ as follows:

$$\lambda_j = \frac{1 - \delta(q_i, q_i^j)}{\Sigma_{j=0}^{m_i}(1 - \delta(q_i, q_i^j))} \tag{4.15}$$

Recall that the smaller $\delta(q_i, q_i^j)$ is, the closer q_i^j is to the original triple pattern q_i. Also recall that $\delta(q_i, q_i^0)$ is equal to 0. This weighting scheme basically gives higher weights to triples that instantiate reformulated patterns which are closer to the original triple pattern q. However, triples instantiating a lower-ranked reformulated triple pattern with sufficiently high scores can have higher probabilities than ones that instantiate a higher-ranked triple patterns.

Once the language model of the triple pattern q_i has been estimated, we use it to compute the probability of a tuple in the query language model. Finally, to rank the overall results of the original query and all its reformulations, we assume there exists a language model for each result (the way we estimate the result language models was explained in Chapter 3). The results are then ranked in ascending order based on the KL divergence between the query language model and the result language models.

Note that both the incremental and the batch modes are only two ways in which we can execute the reformulated queries and present their results to the user. Additional modes can include a mixture of both modes for instance, or any other variations. Our result ranking model described in Chapter 3 is general enough and can support any number of such fine-grained execution modes with minimal changes.

4.3. Related Work

In this chapter we presented a framework for reformulating triple-pattern queries, possibly augmented with keywords. Our framework reformulates queries by substituting resources that appear in a given query, whether entities or relations, with similar ones or with variables. Measuring the similarity between resources is somewhat related to both record linkage [65], and ontology matching [77]. But a key difference is that we are only interested in finding candidate resources which are *close in spirit* to a given resource, and not trying to solve the resource disambiguation problem.

Query reformulation in general has been studied in other contexts such as keyword queries [17] (more generally called query expansion), XML [3, 54], SQL [11, 92]. Our setting of RDF and triple patterns is different in being schemaless (as opposed to relational data) and graph-structured (as opposed to XML which is mainly tree-structured and supports navigational predicates). For RDF triple-pattern queries, query reformulation has been addressed to some extent in [92, 42, 20, 40, 23]. With the exception of [20, 42], the types of reformulations considered in previous work are limited. For example, [92] considers substituting relations only, while [40, 23] consider both entity and relation substitutions. The work in [23], in particular, considers a very limited form of reformulation – relaxing queries by replacing entities or relations specified in the triple patterns with variables. Our approach, on the other hand, considers a comprehensive set of reformulations and in contrast to most other previous approaches, weights the reformulated queries in terms of the *quality* of reformulation (i.e., how close the reformulated queries are to the original query), rather than the number of substitutions that resulted in the reformulated queries.

In addition, our framework stands out in the way reformulated queries are generated and executed. While [20, 42, 40] make use of rule-based rewriting, the approach in [92] and our own approach make use of the data itself to generate query reformulations. Note that rule-based rewriting requires human input, while our approach is completely automatic. Also note that our framework is the only approach that merges the results of the original query and its reformulations in a holistic manner. This allows us to rank results based on both the *relevance* of the results, as well as the *closeness* of the reformulated query to the original query.

#entities	Example entity types	#triples	Example relations
LibraryThing Dataset			
48,000	`book, author` `user, tag`	700,000	`wrote, hasFriend` `hasTag, type`
IMDB Dataset			
59,000	`movie, actor` `director, producer,` `country , language`	600,000	`actedIn, directed` `won, isMarriedTo,` `produced, hasGenre`

Table 4.5.: Overview of the datasets

4.4. Experimental Evaluation

We evaluated our query reformulation framework using three experiments. The first one evaluated the quality of the individual resource substitutions and the second one evaluated the quality of the reformulated queries overall. The third experiment evaluated the quality of the final query results obtained from the original query and its reformulations.

4.4.1. Setup

All experiments were conducted over two datasets using the Amazon Mechanical Turk service[1]. The first dataset was derived from the LibaryThing community, which is an online catalog and forum about books. The second dataset was derived from a subset of the Internet Movie Database (IMDB). The data from both sources was automatically parsed and converted into RDF triples. In addition, each triple was also augmented with keywords derived from the data source it was extracted from. In particular, for the IMDB dataset, all the terms in the plots, tag-lines and keywords fields were extracted, stemmed and stored with each triple. For the LibraryThing dataset, since we did not have enough textual information about the entities present, we retrieved the books' Amazon descriptions and the authors' Wikipedia pages and used them as textual context for the triples. Table 4.5 gives an overview of the datasets.

[1]`http://aws.amazon.com/mturk/`

LibraryThing Dataset			
?b	type	Nonfiction	
?b	hasTag	Greek	
?w	type	Historian	
?w	wrote	?b	
?b	hasTag	Memoir	
?w	wrote	?b	
?b	hasTag	Non-fiction	
?b	hasTag	Pulitzer	
?w	wrote	?b	*nobel prize*
?b	hasTag	British_Literature	
?w	wrote	?b	*civil war*
?b	hasTag	Film	
IMDB Dataset			
?m	hasGenre	Comedy	
?m	hasWonPrize	Academy_Award	
?a	hasWonPrize	Academy_Award_for_Best_Actor	
?a	originatesFrom	New_York	
?m1	hasGenre	Mystery	
?m1	hasPredecessor	?m2	
?d1	directed	?m1	
?d2	directed	?m2	
?d	directed	?m	*true story*
?d	hasWonPrize	Academy_Award_for_Best_Director	
?a	actedIn	?m	*school friends*
?a	type	singer	

Table 4.6.: Subset of the evaluation queries

We constructed *40* evaluation queries for each dataset and converted them into triple-pattern queries. In addition, we constructed *15* keyword-augmented queries for each dataset, where one or more triple-patterns were augmented with one or more keywords. Some example queries are shown in Table 4.6. The complete set of evaluation queries used is listed in Appendix C.

LibraryThing		**IMDB**	
Egypt	**Non-fiction**	**France**	**Titanic_(1997)**
Ancient_Egypt	Politics	Italy	Atlantic_(1929)
Mummies	American_History	Switzerland	The_Abyss
Egyptian	Sociology	Spain	Titanic_(1953)
Cairo	Essays	West Germany	Top_Gun
Egyptology	History	Germany	Britannic

Table 4.7.: Example resources and their top-5 substitutions

4.4.2. Quality of Substitution Lists

To evaluate the quality of individual resource substitutions (i.e., how close substitutions in a resource substitution list are to the resource), we extracted all unique resources, whether entities or relations, occurring in all evaluation queries. The total numbers of entities and relations are given in Table 4.8. For each resource, the top-5 substitutions were retrieved, excluding the variable substitution. Recall that a resource substitution list contains a variable substitution entry, representing substituting the resource with a variable. Also recall that each substitution is associated with a score representing how close the substitution is to the resource it is supposed to substitute. The score of a substitution is measured as the square root of the Jensen-Shannon divergence between the language models of the resource and the substitution. Table 4.7 shows some example resources and their top-5 substitutions excluding the variable substitution.

We presented the resource and each substitution to six evaluators and asked them to assess how closely related the two are on a 3-point scale: 2 corresponding to "closely related", 1 corresponding to "related" and 0 corresponding to "unrelated". Table 4.8 shows the results obtained for resource substitutions. For a given resource, the average rating for each substitution was first computed and then this rating was averaged over the top-5 substitutions for that resource. The second row shows the average rating over all individual resource substitutions. The third row shows the *Pearson correlation* between the average rating and the scores of the substitutions. We achieved a strong *negative* correlation between the score of the substitutions and the average ratings which shows that

	Metric	Entities (0-2)	Relations (0-2)	Queries (0-3)
1	**No. of items**	87	15	80
2	**Avg. rating**	1.228	0.863	1.89
3	**Correlation**	-0.251	-0.431	-0.119
4	**Avg. rating for top substitution**	1.323	1.058	1.94
5	**Avg. rating above variable**	*1.295*	*1.292*	-
6	**Avg. rating below variable**	1.007	0.781	-

Table 4.8.: Quality of individual substitutions and query reformulations

the smaller the score of the substitution is, the higher the rating it was assigned by the evaluators. The fourth row shows the average rating for the top substitution.

The fifth and sixth rows in Table 4.8 show the average rating for substitutions that ranked above and below the variable substitution, respectively. Recall that for each resource, a possible entry in the resource substitution list is a variable as described in Section 4.2. For those substitutions that were ranked above a variable (i.e., whose JS divergence is less than that of a variable), the average rating was more than 1.29 for both entities and relations, indicating how close these substitutions are to the original entity or relation. For those substitutions that were ranked below a variable, the average rating was less than 1.1 for entities and 0.8 for relations. This shows that our pruning strategy for substitution lists, where the lists are pruned after the variable substitution, is indeed effective.

4.4.3. Quality of Query Reformulations

To evaluate the quality of reformulated queries overall, we generated the top-5 reformulated queries for each evaluation query. The reformulated queries were ranked in ascending order of their scores, which were computed according to Equation 4.12. We asked six evaluators to assess how close a reformulated query is to the original one on a 4-point scale: 3 corresponding to ″very-close″, 2 to ″close″, 1 to ″not so close″ and 0 corresponding to ″unrelated″.

Table 4.8 also shows the results obtained for query reformulations. For a given

LibraryThing Dataset			
	Batch Mode	**Incremental Mode**	**Baseline Approach**
NDCG	**0.868**	**0.920**	0.799
Avg. Rating	**2.062**	**2.192**	1.827
IMDB Dataset			
	Batch Mode	**Incremental Mode**	**Baseline Approach**
NDCG	**0.880**	**0.900**	0.838
Avg. Rating	**1.874**	**1.928**	1.792

Table 4.9.: Quality of results for simple triple-pattern queries

query, the average rating for each reformulated query was first computed and then this rating was averaged over the top-5 reformulated queries. Again, the second row shows the average rating over all reformulated queries for a given query. The third row shows the *Pearson correlation* between the average rating and the scores of the reformulated queries. Similar to the case of individual substitutions, we achieved a strong *negative* correlation between the scores of the reformulated queries and the average ratings which shows that the smaller the score of the reformulated query is, the higher the rating it was assigned by the evaluators. The fourth row shows the average rating for the top-scored reformulated query.

4.4.4. Quality of Query Results

We compared our reformulation framework, with its two execution modes, *incremental* and *batch* (see Subsection 4.2.4), against a baseline approach outlined in Chapter 3. The latter generates a set of reformulated queries by *relaxing* one or more triple patterns in a given query. A triple pattern is relaxed by replacing one or more resources (whether an entity or a relation) in the pattern with variables, and the weights of the relaxed triple patterns are set based on the number of resources replaced by variables. The relaxed queries are then executed in batch mode. We used the same ranking model described in Chapter 3 to rank the results with respect to a query Q for all three techniques.

We pooled the top-10 results from all three approaches and presented them to

LibraryThing Dataset			
	Batch Mode	**Incremental Mode**	**Baseline Approach**
NDCG	**0.757**	**0.640**	0.566
Avg. Rating	**1.985**	**1.349**	0.969
IMDB Dataset			
	Batch Mode	**Incremental Mode**	**Baseline Approach**
NDCG	**0.841**	**0.755**	0.623
Avg. Rating	**1.602**	**1.464**	0.922

Table 4.10.: Quality of results for keyword-augmented queries

six evaluators in no particular order. The evaluators were required to assess the results on a 4-point scale: 3 corresponding to "highly relevant", 2 corresponding to "relevant", 1 corresponding to "somewhat relevant", and 0 corresponding to "irrelevant". To measure the ranking quality of each technique, we used the Discounted Cumulative Gain (DCG) [45], which is defined as follows

$$DCG(i) = \begin{cases} G(1) & \text{if } i = 1 \\ DCG(i-1) + G(i)/\log(i) & \text{otherwise} \end{cases}$$

and we set $G(i)$ to a value between 0 and 3 depending on the evaluator's assessment. For each result, we averaged the ratings given by all evaluators and used this as the relevance level for the result. Dividing the obtained DCG by the DCG of the ideal ranking we obtained a *Normalized DCG (NDCG)* which accounts for the variance in performance among queries.

For simple triple-pattern queries (Table 4.9), the batch-execution mode of our approach had over 8% improvement in NDCG over the baseline for LibraryThing and over 5% for IMDB. The incremental-execution mode of our approach had improvements over 15% for LibraryThing and 7% for IMDB. For the keyword-augmented queries (Table 4.10), the improvement is much more evident. The batch-execution mode of our approach outperformed the baseline one with over 33% gain in NDCG for LibraryThing and 21% for IMDB. The incremental-execution mode of our approach had improvements in NDCG of over 13% for LibraryThing and over 8% for IMDB.

Note that in the case of keyword-augmented queries, the batch-execution mode of our approach was preferred by evaluators over the incremental one because

Rank	Result			Rating
Q	**?b**	**type**	**Science_Fiction**	
	?b	**hasTag**	**Film**	
Batch Mode				
1	Star_Trek_Insurrection	type	Science_Fiction	2.50
	Star_Trek_Insurrection	hasTag	Film	
2	Blade	type	Science_Fiction	2.83
	Blade	hasTag	Movies	
3	Star_Wars	type	Science_Fiction	2.00
	Star_Wars	hasTag	Made_Into_Movie	
Incremental Mode				
1	Star_Trek_Insurrection	type	Science_Fiction	2.50
	Star_Trek_Insurrection	hasTag	Film	
2	The_Last_Unicorn	type	Science_Fiction	2.50
	The_Last_Unicorn	hasTag	Movie/tv	
3	The_Mists_of_Avalon	type	Science_Fiction	2.17
	The_Mists_of_Avalon	hasTag	Movie/tv	
Baseline Approach				
1	Star_Trek_Insurrection	type	Science_Fiction	2.50
	Star_Trek_Insurrection	hasTag	Film	
2	Helter_Skelter	type	History	0.83
	Helter_Skelter	hasTag	Film	
3	Fear_&_Loathing_in_Vegas	type	History	1.83
	Fear_&_Loathing_in_Vegas	hasTag	Film	

Table 4.11.: Top-ranked results for an example triple-pattern query

the keywords play a key role in determining the relevance of a result. Since the batch-execution mode does not enforce a strict ordering of the results of the reformulated-queries, it allows results which better match the keyword context (while matching a lower-ranked reformulated query) to be ranked higher.

In Table 4.11 we show an example query and the top-3 results returned by each of the three techniques. Next to each result, we show the average rating given by the evaluators. The resource substitutions are underlined. The example query

Rank	Result			Rating
Q	**?w**	**wrote**	**?b** *civil war*	
	?b	**hasTag**	**Film**	
Batch Mode				
1	Margaret_Mitchell	wrote	Gone_with_the_Wind	2.57
	Gone_with_the_Wind	hasTag	Made_into_a_Movie	
2	Ernest_Hemingway	wrote	For_Whom_the_Bell_Tolls	2.64
	For_Whom_the_Bell_Tolls	hasTag	Made_into_Movie/tv	
3	Charles_Frazier	wrote	Cold_Mountain	2.83
	Cold_Mountain	hasTag	Made_into_Movie	
Incremental Mode				
1	Ernest_Hemingway	wrote	For_Whom_the_Bell_Tolls	2.64
	For_Whom_the_Bell_Tolls	hasTag	Made_into_Movie/tv	
2	Aldous_Huxley	wrote	Brave_New_World	2.12
	Brave_New_World	hasTag	Made_Into_Movie/tv	
3	George_Orwell	wrote	Nineteen_Eighty-four	1.78
	Nineteen_Eighty-four	hasTag	Made_Into_Movie/tv	
Baseline Approach				
1	J_Michael_Straczynski	wrote	Civil_War	1.78
	Civil_War	hasTag	American	
2	Ernest_Hemingway	wrote	For_Whom_the_Bell_Tolls	2.36
	For_Whom_the_Bell_Tolls	hasTag	Europe	
3	Ernest_Hemingway	wrote	A_Farewell_to_Arms	2.00
	A_Farewell_to_Arms	hasTag	World_War_One	

Table 4.12.: Top-ranked results for an example keyword-augmented query

in Table 4.11 asks for science fiction books that have tag Film. There is only *one* one such result which is ranked as the top result by all three approaches. Since the batch-execution mode of our approach ranks the whole set of query results, it allows for more diversity in terms of substitutions. And so, the batch-execution mode of our approach returns the more famous and iconic movies, Blade and Star_Wars as the top results compared to The_Last_Unicorn and The_Mists_Of_Avalon returned by the incremental mode.

The query in Table 4.12 shows an example of a keyword-augmented query which asks for books with the tag Film about a civil war. The top-3 results returned by the batch-execution mode of our approach are all books that were turned into movies. In addition, all of them are about civil wars. The results returned by the incremental mode of our approach were also books that were turned into movies, however only one of them is about a civil war (the first). For the baseline approach, where the tag Film in the second triple pattern was substituted by a variable, the results returned are all books about civil wars, or wars in general, however none of them had a tag related to films.

4.5. Summary

In this chapter we presented a comprehensive query reformulation framework for triple-pattern queries. Our framework reformulates a given query by substituting one or more resources that appears in the query with a similar resource, a variable or removes one or more triple patterns completely. In order to do so, our framework associates with each resource in the knowledge base a substitution list consisting of other resources from the same knowledge base. Each substitution has a score that represents how close the substitution is to the resource it is supposed to substitute.

To construct the substitution lists we construct a language model for each resource in the knowledge base. The resource language models can be estimated using multiple information sources including the knowledge base itself. The similarity between resources is measured as the square root of the Jensen-Shannon divergence between the language models of the resources. The square root of the Jensen-Shannon divergence is a metric that can be summed to compute an overall score for the reformulated queries.

In the substitution lists of resources, an entry for the variable substitution (i.e., substituting the resource the list belongs to with a variable) is maintained. Furthermore, the substitution lists can be pruned after the variable substitution for efficient storage and query processing.

Our query framework generates reformulated queries by utilizing the substitution lists. We presented an algorithm to produce a ranked list of such reformulated queries, where the reformulated queries are ranked based on their scores, which is computed as a sum of the scores of the individual substitutions that result in the reformulated query. Finally, we explained how reformulated queries can be executed along side the original query, and how their results can be merged with the original query results in order to improve the latter's recall. We have backed our framework with experimental evaluation that shows the effectiveness of our approach of query reformulation for triple-pattern queries.

Chapter 5.

Top-k Triple-Pattern Query Processing

Triple-pattern search, where the queries consist of a set of triple patterns possibly augmented with keywords, and the results are ranked using some relevance-based criteria is a very effective way to search RDF knowledge bases. In addition, empowering triple-pattern search with automatic query reformulation can highly improve the quality of the search results. However, query processing for triple-pattern search involves joining large sets of triples and then ranking the resulting tuples. This process of rank-then-join can be very inefficient when the number of result tuples is large. Moreover, processing of keyword conditions alongside triple patterns and performing automatic query reformulation impose additional challenges for efficient query processing.

In this chapter, we develop a set of query processing algorithms to efficiently process triple-pattern queries and their reformulations to retrieve the top-k highest scored results, where the results are scored based on the ranking model described in Chapter 3. Our algorithms are based on the top-k rank-join algorithm introduced by Ilyas et al. [43] as well as the basic Fagin's threshold algorithm [28]. We start by giving an overview of our query processing framework in Section 5.1. We then discuss why a top-k query-processing approach is needed and explain our top-k query-processing framework in Section 5.2. In Section 5.3, we explain the data storage and indexing scheme that our top-k query-processing framework operates on. Finally, we evaluate our framework in Section 5.5.

5.1. Query Processing for Triple-Pattern Search

Our query processing framework handles three types of query processing tasks: triple-pattern queries, keyword-augmented triple-pattern queries, and query reformulation. We explain each task separately in the following.

5.1.1. Triple-Pattern Queries

To search an RDF knowledge base, triple-pattern queries are expressed. For example, to find thriller movies and their directors, the following query consisting of two triple patterns can be issued:

```
?d  directed  ?m
?m  hasGenre  Thriller
```

Given a query with n triple patterns, the results of the query is the set of all tuples of n triples that instantiate the query triple patterns and satisfy the query join conditions denoted by using the same variable in more than one triple pattern. To find such tuples, we need to first retrieve an *instantiation list* L_i for each triple pattern q_i specified in the query which contains all the triples from the knowledge base that instantiate triple pattern q_i. For instance, the instantiation list of the first triple pattern in our example query would consist of all triples with predicate `directed`. Once we have an instantiation list L_i for each triple pattern q_i, we need to join them based on the join conditions specified in the query using some joining strategy. For our example query, a valid result would be a tuple of two triples $T = (t_1, t_2)$, such that the first triple $t_1 \in L_1$, the second triple $t_2 \in L_2$ and the object of t_1 is the same as the subject of t_2. One example result for our example query above is the 2-tuple (i.e., tuple of two triples):

```
Quentin_Tarantino  directed  Pulp_Fiction
Pulp_Fiction       hasGenre  Thriller
```

Moreover, we would like to also provide a ranked list of query results rather than a set of unranked matches. This means that after joining all the triples from the different instantiation lists, we must rank the joined tuples using some ranking strategy. For example, the result tuples can be ranked using our ranking model for triple-pattern queries described in Chapter 3.

5.1.2. Keyword-Augmented Triple-Pattern Queries

In addition to triple-pattern queries, our framework also processes keyword-augmented triple-pattern queries. A keyword augmented triple-pattern query is a triple-pattern query where one or more of the triple patterns are augmented with one or more keywords. For example, the following keyword-augmented query can be issued to retrieve thriller movies about serial killers and their directors:

```
?d  directed  ?m
?m  hasGenre  Thriller  serial killer
```

To be able to process keyword-augmented triple-pattern queries, we augment each triple in our knowledge base with a text snippet, which contains any contextual text from the sources the triples were extracted from. This way, we can also process the keywords specified in a keyword-augmented triple-pattern query. The results of a keyword-augmented query consisting of n triple patterns is the set of all tuples of n triples that instantiate the query triple patterns, satisfy the query join conditions and whose text snippets match the keywords specified in the query. To find such tuples, we need to retrieve an instantiation list L_i for each triple pattern q_i specified in the query which contains all the triples from the knowledge base that instantiate triple pattern q_i and whose text snippets match the keywords specified in q_i, if any exists. Once we have an instantiation list L_i for each triple pattern q_i, we need to join them based on the join conditions specified in the query using some joining strategy. For example, one result for our example query above is the 2-tuple:

```
Alfred_Hitchcock  directed  Psycho
Psycho            hasGenre  Thriller
```

Once we have generated all possible joined tuples for a given query, we need to also rank them according to some ranking strategy such as the ranking model from Chapter 3.

5.1.3. Query Reformulation

Similar to query expansion in traditional IR, triple-pattern queries can be automatically reformulated to improve their recall. In Chapter 4, we presented

directed	**hasGenre**	**Thriller**
`actedIn`: 0.413	`var`: 0.525	`Action`: 0.477
`created`: 0.418		`var`: 0.503
`produced`: 0.438		
`var`: 0.472		

Table 5.1.: Substitution lists for three resources and the score of each substitution

?d	**directed**	**?m**	δ	**?m**	**hasGenre**	**Thriller**	δ
?d	actedIn	?m	0.413	?m	hasGenre	Action	0.477
?d	created	?m	0.418	?m	hasGenre	?y	0.503
?d	produced	?m	0.438	?m	?y	Thriller	0.525
?d	?x	?m	0.472	?m	?y	Action	1.002
				?m	?y	?z	1.028

Table 5.2.: Reformulated triple patterns and their scores

a framework for query reformulation that reformulates a given triple-pattern query by reformulating one or more of its triple patterns. A triple pattern is in turn reformulated by replacing one or more resources (whether entities or relations) that appear in the triple pattern with similar resources or variables. In order to do so, we associate with each resource X in the knowledge base a *substitution list* which consists of a list of resources ordered on some score. More precisely, let the substitution list of resource X be $L(X)$. A resource $Y \in L(X)$ would be associated with a score $\delta(Y, X)$ which measures how close resources X and Y are according to some distance metric. Moreover, $L(X)$ would also contain an entry for the variable substitution (i.e., replacing X with a variable) which we denote by var, and this would also be associated with a score $\delta(var, X)$. The way we construct these substitution lists has been discussed in Chapter 4.

For example, consider the triple-pattern query:

```
?d  directed  ?m
?m  hasGenre  Thriller
```

Table 5.1 shows the substitution lists for the three resources that appear in the example query: `directed`, `hasGenre` and `Thriller` and their scores. Using

?d	**directed**	**?m;**	**?m**	**hasGenre**	**Thriller**	δ
?d	actedIn	?m;	?m	hasGenre	Thriller	0.413
?d	created	?m;	?m	hasGenre	Thriller	0.418
?d	produced	?m;	?m	hasGenre	Thriller	0.438
?d	?x	?m;	?m	hasGenre	Thriller	0.472
?d	directed	?m;	?m	hasGenre	Action	0.477
?d	directed	?m;	?m	hasGenre	?x	0.503
?d	directed	?m;	?m	?y	Thriller	0.525
?d	actedIn	?m;	?m	hasGenre	Action	0.879
?d	created	?m;	?m	hasGenre	Action	0.884
?d	produced	?m;	?m	hasGenre	Action	0.915

Table 5.3.: Top-10 reformulated queries for a given example query and their scores

these substitution lists, a list of reformulated triple patterns for each of the triple patterns in the example query can be constructed. Furthermore, the reformulations in each of these lists are associated with scores that represent how close the reformulations are to the triple pattern the list belongs to. The score of a reformulated triple pattern is computed as the sum of the scores of the substitutions that resulted in the reformulated triple pattern. Table 5.2 shows the set of reformulated triple patterns for each triple pattern in our example query and their scores δ.

Using the reformulation lists of individual triple patterns, a list of reformulated queries can be generated. Again, each such reformulated query is associated with a score which measures how close the reformulated query is to the original query, and the score of a reformulated query is computed as the sum of the scores of its triple patterns. Table 5.3 shows the top-10 reformulated queries for our example query.

Now, to process a query and all its reformulations, we must do the following. Let $Q = (q_1, q_2, ..., q_n)$ be the given query where q_i is a triple pattern. Furthermore, let $\{q_i^1, q_i^2, ..., q_i^{m_i}\}$ be all the reformulations of triple pattern q_i. For each triple pattern q_i, we must:

1. Retrieve the instantiation list L_i of q_i

2. Retrieve the instantiation list L_i^j for each reformulated triple pattern q_i^j where $1 \leq j \leq m_i$

3. Merge all instantiation lists L_i and $L_i^1, L_i^2, ..., L_i^{m_i}$

Once we have retrieved a merged list of triples for each triple pattern q_i, we need to join them based on the join conditions specified in the query to produce joined result tuples which are then ranked using some ranking strategy such as the one explained in Chapter 4.

5.2. Top-k Query Processing Framework

In the previous section, we presented our basic query-processing framework. As we explained, given a triple-pattern query $Q = (q_1, q_2, ..., q_n)$ where q_i is a triple pattern, we must retrieve n instantiation lists L_i, join them based on the query join conditions and then rank the resulting tuples of joined triples. Moreover, in case automatic query reformulation is supported, we need to first merge the instantiation lists of each triple pattern and its reformulations, then join them with the merged lists of the other triple patterns to produce a set of joined tuples that are then ranked to produce the final results.

For large RDF knowledge bases, these instantiation lists might be very long, and the approach of joining all the triples and then ranking the resulting tuples might be too expensive especially when we are just interested in retrieving the top-k highest ranked results for a given query, where k is typically a small value (10, 20 or 100). In such case, it will be very beneficial to develop a top-k processing approach that retrieves only some triples for each triple pattern, which are hopefully few, and then joins these fewer triples together to directly produce a ranked list consisting of the top-k highest scored tuples of joined triples.

In this section, we present a set of algorithms that take as an input a triple-pattern query, possibly augmented with keywords, and retrieves the top-k highest scored results for the query. We start with the easiest case, which is the case of a triple-pattern query. We then explain how we can process a keyword-augmented triple-pattern query and finally we explain how we handle query reformulation.

5.2.1. Triple-Pattern Queries

Basic Setting. To process a triple pattern query $Q = (q_1, q_2, ..., q_n)$ where q_i is a triple pattern, our framework makes the following assumptions:

1. For each triple pattern q_i, there exists an instantiation list L_i which contains all the triples instantiating triple pattern q_i sorted in *descending* order of their scores $S(t_i, q_i)$
2. The score of a result tuple $T = (t_1, t_2, ..., t_n)$ where $t_i \in L_i$ is computed as follows:
$$S(T, Q) = \prod_{i=1}^{n} S(t_i, q_i) \tag{5.1}$$

The way we compute the instantiation lists and the way we set the scores will be discussed in Section 5.3.

Algorithm. Our top-k query processing algorithm for triple-pattern queries (Algorithm 2) is based on the rank-join algorithm introduced by Ilyas et al. [43]. Algorithm 2 takes as an input a triple-pattern query $Q = (q_1, q_2, ..., q_n)$ where q_i is a triple pattern and the number of desired results k. The algorithm reports the top-k highest scored tuples of joined triples where the score of a tuple is computed according to Equation 5.1.

Algorithm 2 maintains two basic data structures: a *list* L that contains the top-k highest-scored tuples and a *hashmap* $seen_i$ for each triple pattern q_i. $seen_i$ contains the triples t_i retrieved for each triple pattern q_i from its instantiation list L_i and their scores $S(t_i, q_i)$. All data structures are assumed to be initially empty. Algorithm 2 also keeps track of two scores for each triple pattern q_i: $first(q_i)$ and $last(q_i)$. $first(q_i)$ is the maximum score of any triple $t_i \in L_i$ (i.e., the score of the first triple in L_i) and $last(q_i)$ is the score of the last triple retrieved from L_i. $last(q_i)$ is initially set to $first(q_i)$ and is then decremented as more triples are retrieved from L_i.

Algorithm 2 starts by determining the next triple pattern q_i to process. We explain how this can be done later in this subsection. Once a triple pattern q_i has been picked, Algorithm 2 utilizes the method $q_i.next()$ which iterates over the triples in the instantiation list L_i from top to down and each time the method is called, it removes the triple t_i from the top of the list L_i and stores it, along

Algorithm 2 RANKJOIN($q_1, q_2, ...q_n, k$)

```
1: $L \leftarrow \phi$
2: for $(1 \leq i \leq n)$ do
3:     $seen_i \leftarrow \phi$
4:     $first(q_i) \leftarrow \max_{t_i \in L_i} S(t_i, q_i)$
5:     $last(q_i) \leftarrow first(q_i)$
6: end for
7: while (all triple patterns have not been completely processed) do
8:     determine next triple pattern $q_i$ to process
9:     $(t_i, S(t_i, q_i)) \leftarrow q_i.next()$
10:    $seen_i.insert((t_i, S(t_i, q_i))$
11:    $last(q_i) \leftarrow S(t_i, q_i)$
12:    $candidates \leftarrow$ join $\{\{t_i\}, seen_1, seen_2, ..., seen_n\} \setminus \{seen_i\}$
13:    for all $(T = (t_1, t_2, ..., t_n) \in candidates)$ do
14:        $S(T, Q) \leftarrow \prod_{i=1}^{n} seen_i.get(t_i)$
15:    end for
16:    $L \leftarrow$ list of k tuples generated so far with highest scores
17:    $min_k \leftarrow \min_{T \in L} S(T, Q)$
18:    $\tau \leftarrow \max_{1 \leq i \leq n} last(q_i) \frac{\prod_{k=1}^{n} first(q_k)}{first(q_i)}$
19:    if $(min_k \geq \tau)$ then
20:        break
21:    end if
22: end while
23: return $L$
```

with its score $S(t_i, q_i)$ in the hashmap $seen_i$. The algorithm then generates a set of candidate result tuples by joining t_i with triples present in the other $n - 1$ hashmaps. The algorithm computes the score of each generated tuple as the product of the scores of its triples as described in Equation 5.1 and stores in list L the top-k highest-scored tuples generated so far. The algorithm halts when the minimum score of the tuples in L is less than the threshold value τ which is computed as the *maximum* of the following n values:

$$\begin{gathered}\mathtt{last}(q_1).\mathtt{first}(q_2).\mathtt{first}(q_3)......\mathtt{first}(q_n),\\ \mathtt{first}(q_1).\mathtt{last}(q_2).\mathtt{first}(q_3)......\mathtt{first}(q_n),\\ \mathtt{first}(q_1).\mathtt{first}(q_2).\mathtt{last}(q_3)......\mathtt{first}(q_n),\\ ....,\\ \mathtt{first}(q_1).\mathtt{first}(q_2).\mathtt{first}(q_3)......\mathtt{last}(q_n)\end{gathered}$$

The threshold value is the maximum score any candidate result tuple not yet generated can acquire. If all tuples in the top-k list have higher scores than τ, then we are sure that the top-k list have the highest-scored result tuples for the query Q. Otherwise, Algorithm 2 determines the next triple pattern q_i to process, retrieves the next triple from L_i by calling the method $q_i.next()$ and repeats the same procedure described above until our halting condition is met or all the triples for all the query triple patterns have been retrieved.

The Processing Order of Triple Patterns. In each iteration of Algorithm 2, the algorithm must determine the next triple pattern q_i to process and then retrieves its next triple from L_i. This can be done in various ways, for instance in a round robin fashion, or by picking the triple pattern q_i that would result in reducing the threshold value which can be done by checking the following n values:

$$\begin{gathered}\mathtt{last}(q_1).\mathtt{first}(q_2).\mathtt{first}(q_3)......\mathtt{first}(q_n),\\ \mathtt{first}(q_1).\mathtt{last}(q_2).\mathtt{first}(q_3)......\mathtt{first}(q_n),\\ \mathtt{first}(q_1).\mathtt{first}(q_2).\mathtt{last}(q_3)......\mathtt{first}(q_n),\\ ....,\\ \mathtt{first}(q_1).\mathtt{first}(q_2).\mathtt{first}(q_3)......\mathtt{last}(q_n)\end{gathered}$$

and greedily picking the next triple pattern q_i such that:

$$i = \underset{1 \leq j \leq n}{\operatorname{argmax}} \ \mathtt{last}(q_j) \frac{\prod_{k=1}^{n} \mathtt{first}(q_k)}{\mathtt{first}(q_j)}$$

Recall that the maximized value is the threshold value and by picking triple pattern q_i this way and retrieving its next triple from L_i, we are trying to reduce the threshold value, which would then result in our algorithm halting as early as possible.

Theorem 5.1 *: Algorithm 2 correctly reports the top-k highest-scored query results.*

Proof: For simplicity, we assume the query Q consists of only two triple patterns q_1 and q_2. The proof can be easily extended (with a more complex notation) to cover the general case where the query consists of n triple patterns.

Assume that the algorithm halts and reports the list L as the top-k results. Furthermore, assume that the tuple $T = (t_1, t_2) \in L$. We thus have $S(T) \geq \tau$, i.e.,

$$S(t_1, q_1).S(t_2, q_2) \geq max(first(q_1).last(q_2), last(q_1).first(q_2)) \tag{5.2}$$

Now assume that there exists a result tuple $T' = (t_1', t_2')$ not yet generated by the algorithm such that $S(T', Q) > S(T, Q)$. This implies that $S(T', Q) > \tau$, i.e.,

$$S(t_1', q_1).S(t_2', q_2) > max(first(q_1).last(q_2), last(q_1).first(q_2)) \tag{5.3}$$

which in turn implies that

$$S(t_1', q_1).S(t_2', q_2) > first(q_1).last(q_2) \tag{5.4}$$

and

$$S(t_1', q_1).S(t_2', q_2) > last(q_1).first(q_2) \tag{5.5}$$

Since, for each triple pattern, the triples are fetched in descending order of their scores, we have $first(q_1) \geq S(t_1', q_1)$ which implies that $S(t_2', q_2) > last(q_2)$. Otherwise Inequality 5.4 would not hold because of the *monotonicity* of multiplication. Therefore, t_2' must have been fetched for triple pattern q_2 before we halted.

Using an analogous argument, we have $first(q_2) \geq S(t_2', q_2)$ which implies that $S(t_1', q_1) > last(q_1)$. Otherwise Inequality 5.5 would not hold because of the *monotonicity* of multiplication. Therefore, t_1' must have been fetched for triple pattern q_1 before we halted.

Since triples t_1' and t_2' have been both fetched before we halted, the tuple $T' = (t_1', t_2')$ must have been generated and must have been in the top-k list L since it has a higher score than $T \in L$ which contradicts our assumption. Thus, when Algorithm 2 halts, it is guaranteed that the top-k list L would contain the top-k highest-scored result tuples for the given query.

5.2.2. Keyword-Augmented Triple-Pattern Queries

The case of keyword-augmented queries is more complicated. Assume query $Q = (q_1, q_2, ..., q_n)$ is a keyword-augmented triple-pattern query such that triple pattern q_i is augmented with keywords $w_1, w_2, ..., w_k$. To be able to utilize Algorithm 2, we must have a list L_i that contains all triples t_i instantiating triple pattern q_i sorted in descending order of their score $S(t_i, q_i)$. Since q_i is augmented with keywords $w_1, w_2, ..., w_k$, it is natural to expect that the scores of the triples instantiating q_i depend on the keywords somehow. Otherwise, the keywords would not play any role in the ranking of the results. Indeed, in the ranking model of Chapter 3, we computed the score $S(t_i, q_i)$ of a triple t_i with respect to triple pattern q_i as a combination over m scores as follows:

$$S(t_i, q_i) = \prod_{j=1}^{k} S(t_i, q_i, w_j)$$

where $S(t_i, q_i, w_j)$ is the score of triple t_i with respect to triple pattern q_i and keyword w_j.

Assuming that there exists a list for each triple pattern q_i that contains all the triples t_i instantiating q_i sorted on their combined scores $S(t_i, q_i)$ is too impractical due to the curse of dimensionality. That is, one need to construct one such list for every possible combination of keywords. Assuming that we have m keywords in our vocabulary that keyword expressions are drawn from, we would need to construct 2^m instantiation lists for each triple pattern.

To overcome this curse of dimensionality, we utilize a second algorithm (Algorithm 3) that processes a keyword-augmented triple pattern separately and returns the triples t_i instantiating triple pattern q_i in order of their *combined* scores $S(t_i, q_i)$. This algorithm is combined with Algorithm 2 as follows. Each time Algorithm 2 invokes the method $q_i.next()$ where q_i is augmented with keywords $w_1, w_2, ..., w_k$, the method $q_i.next(w_1, w_2, ..., w_k)$ is called. The method $q_i.next(w_1, w_2, ..., w_k)$ would then return the next triple t_i with the highest combined score not yet retrieved for triple pattern q_i along with its combined score $S(t_i, q_i)$. The rest of Algorithm 2 behaves in the same way as we explained in the case of triple-pattern queries only.

This *pipelined* approach we have taken to process keyword-augmented triple patterns nicely encapsulates the details of processing keyword-augmented triple patterns and provides a level of parallelism where the more expensive keyword-augmented triple patterns can be processed in parallel with other triple patterns. We next explain our algorithm for processing keyword-augmented triple patterns (i.e., method $q_i.next(w_1, w_2, ..., w_k)$).

Basic Setting. To process a keyword-augmented triple pattern where q_i is augmented with keywords $w_1, w_2, ..., w_k$, our framework makes the following assumptions:

1. For each triple pattern q_i and each keyword w_j, there exists an instantiation list L_{ij} which contains all the triples instantiating triple pattern q_i sorted in *descending* order of their scores $S(t_i, q_i, w_j)$
2. The combined score of a triple t_i instantiating triple pattern q_i is computed as follows:
$$S(t_i, q_i) = \prod_{j=1}^{k} S(t_i, q_i, w_j) \tag{5.6}$$

The way we construct the instantiation lists and the way we set the scores will be discussed in Section 5.3.

Algorithm. Algorithm 3 uses two main data structures: a *priority queue* $Queue$ and a set of *hashmaps* $seen_j$ for each keyword w_j. $seen_j$ stores the set of triples t_i retrieved for each keyword w_j from list L_{ij}. All data structures are assumed to be empty before Algorithm 3 is invoked for the first time. Algorithm 3 also maintains k values $last(w_j)$ which is the score of the last triple triple t_i retrieved for keyword w_j from list L_{ij}. $last(w_j)$ is initially set to the score of the first triple t_i in list L_{ij}.

In each iteration of Algorithm 3, we determine the next keyword w_j to be processed. We will explain how we do this later in this subsection. Once a keyword w_j is picked, Algorithm 3 retrieves the next triple t_i from the list L_{ij} and its score $S(t_i, q_i, w_j)$. We then compute the *best score* this triple can achieve as follows:

$$B(t_i, q_i) = \prod_{j \in keywords(t_i)} S(t_i, q_i, w_j). \prod_{j \notin keywords(t_i)} last(w_j)$$

where $keywords(t_i)$ is the set of indices of the keywords that t_i has been retrieved for so far (initially empty for every triple). The score $B(t_i, q_i)$ is an upper bound for the combined score of triple t_i provided that the lists L_{ij} are sorted in descending order of the scores of the triples. The triple t_i is then inserted into our priority queue $Queue$ along with its best score $B(t_i, q_i)$.Finally, each time a

Algorithm 3 next($w_1, w_2, ..., w_k$)

```
1:  if (FIRST_TIME) then
2:      $Queue \leftarrow \phi$
3:      for ($1 \leq j \leq k$) do
4:          $seen_j \leftarrow \phi$
5:          $last(w_j) \leftarrow \max_{t_i \in L_{ij}} S(t_i, q_i, w_j)$
6:      end for
7:  end if
8:  if ($Queue$ is not empty) then
9:      $(t_i, B(t_i, q_i)) \leftarrow Queue.head()$
10:     if $|keywords(t_i)| = k$ then
11:         $(t_i, S(t_i, q_i)) \leftarrow Queue.remove()$
12:         return $(t_i, S(t_i, q_i))$
13:     end if
14: end if
15: while (all keywords have not been completely processed) do
16:     determine next $w_j$ to process
17:     $(t_i, S(t_i, q_i, w_j)) \leftarrow q_i.next(w_j)$
18:     $Queue.insert(t_i, 0)$
19:     $keywords(t_i) \leftarrow Keywords(t_i) \cup \{j\}$
20:     $seen(w_j).insert(t_i, S(t_i, q_i, w_j))$
21:     $last(w_j) \leftarrow S(t_i, q_i, w_j)$
22:     for all ($t \in Queue$) do
23:         $B(t, q_i) \leftarrow \prod_{j \in keywords(t)} seen_j.get(t). \prod_{j \notin keywords(t)} last(w_j)$
24:         $Queue.updateScore(t, B(t, q_i))$
25:     end for
26:     $(t_i, B(t_i, q_i)) \leftarrow Queue.head()$
27:     if $|keywords(t_i)| = k$ then
28:         break
29:     end if
30: end while
31: if ($Queue$) is not empty then
32:     $(t_i, S(q_i, t_i)) \leftarrow Queue.remove()$
33:     return $(t_i, S(q_i, t_i)$
34: end if
```

triple t_i is retrieved from a list L_{ij}, $last(w_j)$ either stays the same or decreases. We thus need to update the best scores of all the triples in the queue accordingly.

The main loop in Algorithm 3 is broken when the triple at the head of the queue (i.e., with the highest best score) has been retrieved for all keywords (i.e., $|keywords(t_i)| = k$). In this case, we return the triple t_i at the head of the queue since this triple would be the next highest-scored triple instantiating triple pattern q_i and moreover, its best score $B(t_i, q_i)$ would be its true combined score as computed according to Equation 5.6. Note that while we can ensure that a triple t_i is the one with the highest combined score for a triple pattern q_i and keywords $w_1, w_2, ..., w_k$ by keeping a lower bound on the score a triple can acquire, this is not sufficient since these triples would then be joined with other triples instantiating other triple patterns using Algorithm 2 which would not correctly report the top-k result tuples unless we assume that the method $q_i.next()$ retrieves the triples in order of their scores, and that each retrieved triple is associated with its true combined score $S(t_i, q_i)$.

The next time Algorithm 3 is invoked (by calling method $q_i.next()$ from Algorithm 2), Algorithm 3 checks the triple at the head of the queue (if the queue is not empty) and if its true combined score has been computed, it returns it as the next triple along with its score. Otherwise, the procedure we described above is repeated.

Processing Order of Keywords. In each iteration of Algorithm 3, the algorithm must determine the next keyword w_j to process and then retrieves the next triple from list L_{ij}. This can be done in various ways, for instance in a round robin fashion, or by picking the keyword w_j with the maximum $last(w_j)$ which would result in reducing the best scores of all triples in the queue that have not been retrieved for this keyword, or deploying any other heuristics that would result in the algorithm returning the next triple as fast as possible.

Theorem 5.2 : *Given a triple pattern q_i and keywords $w_1, w_2, ..., w_k$, Algorithm 3 correctly returns the triple with the highest combined score instantiating triple pattern q_i and its true combined score.*

Proof: Assume that Algorithm 3 returns triple t_i as the next highest-scored triple instantiating triple pattern q_i. This means that : 1) its best score is indeed the true combined score of the triple because our algorithm would only return a triple if it has been seen for all keywords (i.e., $|\mathtt{keywords}(t_i)| = k$) and 2) t_i was at the head of the queue, which means that the score of the triple $S(t_i, q_i) = \prod_{j=1}^{k} S(t_i, q_i, w_j)$ would be higher than all the best scores of all other triples in the queue. Let one such triple be t'_i. We thus have:

$$\prod_{j=1}^{k} S(t_i, q_i, w_j) \geq \prod_{j \in \mathtt{keywords}(t'_i)} S(t'_i, q_i, w_j). \prod_{j \notin \mathtt{keywords}(t'_i)} \mathtt{last}(w_j) \tag{5.7}$$

where $\mathtt{last}(w_j)$ is the score of the last triple retrieved for keyword w_j. Furthermore, since invoking the method $q_i.\mathtt{next}(w_j)$ would return the triples t_i in order of their scores $S(t_i, q_i, w_j)$, we have:

$$\prod_{j \notin \mathtt{keywords}(t'_i)} \mathtt{last}(w_j) \geq \prod_{j \notin \mathtt{keywords}(t'_i)} S(t'_i, q_i, w_j) \tag{5.8}$$

From Inequality 5.7 and Inequality 5.8, we can conclude that:

$$\prod_{j=1}^{k} S(t_i, q_i, w_j) \geq \prod_{j=1}^{k} S(t'_i, q_i, w_j) \tag{5.9}$$

which implies that the true combined score of t_i must be greater than the true combined score of any triple t'_i in the queue.

Now, let's consider the case when the triple t' was not in the queue. Since, $\forall\ 1 \leq j \leq k$, we have:

$$S(t_i, q_i, w_j) \geq \mathtt{last}(w_j) \tag{5.10}$$

as the method $q_i.\mathtt{next}(w_j)$ returns the triples in order of their scores $S(t_i, q_i, w_j)$ and t_i must have been seen for all keywords. Given this and the monotonicity of the multiplication, we have:

$$\prod_{j=1}^{k} S(t_i, q_i, w_j) \geq \prod_{j=1}^{k} \mathtt{last}(w_j) \tag{5.11}$$

Similarly, $\forall\ 1 \leq j \leq k$, we have:

$$\mathtt{last}(w_j) \geq S(t'_i, q_i, w_j) \tag{5.12}$$

as the method $q_i.\mathtt{next}(w_j)$ returns the triples in order of their scores $S(t_i, q_i, w_j)$ and t'_i is not in the queue which means it has not been retrieved for any of the keywords. Given this and the monotonicity of the multiplication, we have:

$$\prod_{j=1}^{k} \mathtt{last}(w_j) \geq \prod_{j=1}^{k} S(t'_i, q_i, w_j) \tag{5.13}$$

?d	**directed**	**?m;**	**?m**	**hasGenre**	**Thriller**	δ
?d	actedIn	?m;	?m	hasGenre	Thriller	0.413
?d	created	?m;	?m	hasGenre	Thriller	0.418
?d	produced	?m;	?m	hasGenre	Thriller	0.438
?d	?x	?m;	?m	hasGenre	Thriller	0.472
?d	directed	?m;	?m	hasGenre	Action	0.477
?d	directed	?m;	?m	hasGenre	?x	0.503
?d	directed	?m;	?m	?y	Thriller	0.525
?d	actedIn	?m;	?m	hasGenre	Action	0.879
?d	created	?m;	?m	hasGenre	Action	0.884
?d	produced	?m;	?m	hasGenre	Action	0.915

Table 5.4.: Top-10 reformulated queries for a given example query and their scores

From Inequality 5.11 and Inequality 5.13, we have:

$$\prod_{j=1}^{k} S(t_i, q_i, w_j) \geq \prod_{j=1}^{k} S(t'_i, q_i, w_j) \qquad (5.14)$$

That is, the true combined score of t_i is higher than that of any other triple not yet in the queue.

From all the above, we conclude that Algorithm 3 indeed returns the triple t_i instantiating triple pattern q_i with the highest combined score and its score $S(t_i, q_i)$ is the true combined score as computed according to Equation 5.6.

5.2.3. Query Reformulation

In Subsection 5.2.3, we explained how given a triple pattern query, we can generate a set of reformulated queries that are close in intention to the given query. In our query framework, each such reformulated query is associated with a score δ that measures how close the reformulation is to the original query. Table 5.4 shows an example query and its top-10 reformulations which were generated using the techniques presented in Chapter 4.

We now explain how we handle query reformulation within our top-k query processing framework. We consider two possible modes of operation. The first, which we refer to as the *incremental* mode, processes the queries incrementally, whereas the second mode, which is referred to as the *batch* mode, processes all queries together. We explain each mode in more details next.

Incremental Processing of Reformulated Queries

Assume that query Q has the set of reformulated queries $\{Q^1, Q^2, ..., Q^m\}$ and that each reformulated query Q^j is associated with a score $\delta(Q, Q^j)$ which represents how close the reformulated query Q^j is to the original query Q. To retrieve the top-k highest-scored results of query Q and all its reformulations, we do the following. Let the total number of results of query Q be $|Q|$ and furthermore, let the total number of results of each reformulated query Q^j be $|Q^j|$. We start by retrieving the k highest-scored results of query Q using the algorithms we described in the previous two subsections . If $|Q| < k$, we retrieve the $k - |Q|$ highest-scored results of the reformulated query Q^j with the highest score $\delta(Q, Q^j)$ such that each result retrieved for query Q^j has not been already retrieved for query Q. If $|Q|+|Q^j| < k$, we retrieve the $k-|Q|-|Q^j|$ highest-scored results of the next highest-scored reformulated query, without counting duplicates until we finally retrieve k results or all the results of all the reformulated queries have been completely retrieved.

Batch Processing of Reformulated Queries

The other alternative to retrieve the top-k highest-scored results for a given query and all its reformulations works as follows. Given a query $Q = \{q_1, q_2, ..., q_n\}$ where q_i is a triple pattern, let $\{q_i^0, q_i^1, q_i^2, ..., q_i^{m_i}\}$ be the set of reformulations of the triple pattern q_i where q_i^0 is the original triple pattern q_i. We can directly utilize Algorithm 2 to retrieve the result tuples for query Q if we assume there exists one list L_i for each triple pattern q_i and all its reformulations that contains all the triples t_i instantiating q_i or any of its reformulations sorted in descending order of their scores $S(t_i, q_i)$. Given that a triple t_i might match more than one triple pattern q_i^j, we must combine the individual scores $S(t_i, q_i^j)$ somehow in order to compute the overall combined score of a triple t_i. Again, this might be impractical as the number of such reformulations can be arbitrarily large which means we might have very long lists. Moreover, in case q_i is keyword augmented, then all of its reformulations will also be augmented with the same keywords, and thus in order to keep a single list of triples for a triple pattern and its reformulations, one would need to keep a list for every possible combination of keywords that can be asked, which , as we pointed out in the previous subsection, is exponentially many.

Thus, we need to adapt a similar approach to the one we used to process a keyword-augmented triple pattern. That is, we need to process a triple pattern q_i and all its reformulations separately and retrieve triples t_i in order of their combined scores $S(t_i, q_i)$. In order to do this, we use Algorithm 4 which is combined with Algorithm 2 as follows. Each time Algorithm 2 invokes the method $q_i.next()$, Algorithm 4 would be called which returns a triple t_i and its combined score $S(t_i, q_i)$ to the calling method $q_i.next()$. The rest of Algorithm 2 behaves in the same way as we explained before.

Again, this pipelined approach we have taken to process reformulated triple patterns nicely encapsulates the details of the processing of reformulated triple patterns. In addition, it provides a level of parallelism where all the triple patterns (and their reformulations) can be processed in parallel. Before we explain how our algorithm for processing a triple pattern and all its reformulations works, we explain our basic setting and how we combine the individual scores of the triples to produce the overall combined score of a triple.

Basic Setting. Given a triple pattern q_i and all its reformulations $q_i^0, q_i^1, ..., q_i^{m_i}$, our framework makes the following assumptions:

1. For every triple pattern q_i^j, there exists an instantiation list L_i^j which contains all the triples t_i instantiating triple pattern pattern q_i^j ordered in *descending* order of their scores $S(t_i, q_i^j)$
2. Each triple pattern q_i^j is associated with a weight λ_j
3. The combined score of triple t_i instantiating triple pattern q_i or any of its reformulations is computed as follows:

$$S(t_i, q_i) = \lambda_0 S(t_i, q_i^0) + \lambda_1 S(t_i, q_i^1) + ... + \lambda_{m_i} S(t_i, q_i^{m_i}) \tag{5.15}$$

The way we construct the instantiation lists and the way we set the weights of the triple patterns and the scores of the triples will be explained in Section 5.3.

Equation 5.15 computes the score of a triple t_i instantiating any of the triple patterns $q_i^0, q_i^1, ..., q_i^{m_i}$ as a weighted sum over all the triple patterns. Note that the score $S(t_i, q_i^j)$ would be *zero* if triple t_i does not instantiate pattern q_i^j. A triple t_i would instantiate one particular triple pattern in the set $\{q_i^0, q_i^1, ..., q_i^{m_i}\}$ and all *relaxed* versions of that triple pattern. A relaxed version of a triple pattern q_i^j is a triple pattern where one or more resources specified in q_i^j are replaced by variables. For example, consider the triple

```
Quentin_Tarantino  directed  Pulp_Fiction
```

The above triple instantiates the following two triple patterns only (the first triple pattern in our example query and one of its reformulations):

```
?m  directed  ?m
?d  ?x        ?m
```

Similarly, the triple

```
Pulp_Fiction  hasGenre  Thriller
```

instantiates the following 4 triple patterns (the second triple pattern in our example query and 3 of its reformulations):

```
?m  hasGenre  Thriller
?m  hasGenre  ?y
?m  ?y        Thriller
?m  ?y        ?z
```

Using the above observation, we can group the reformulated triple patterns in the reformulation set of a given triple pattern into subsets of triple patterns that can be instantiated by the same triple, which we call *compatible* triple patterns.

Definition 5.1 : Compatible Triple Patterns
Given two triple patterns $q = (s, p, o)$ *and* $q' = (s', p', o')$*, let* VAR *be the set of all variables.* q *and* q' *are said to be compatible iff:*

- $s = s'$ *or* $s \in VAR$ *or* $s' \in VAR$,
- $p = p'$ *or* $p \in VAR$ *or* $p' \in VAR$, *and*
- $o = o'$ *or* $o \in VAR$ *or* $o' \in VAR$.

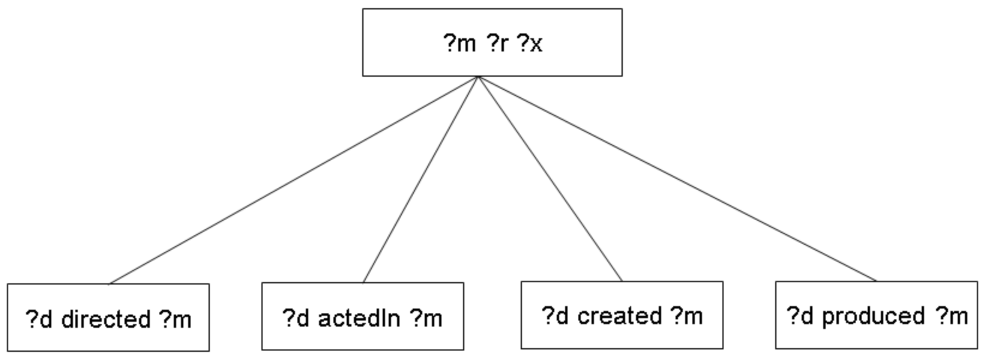


Figure 5.1.: The reformulation graph for the triple pattern `?d directed ?m` and all its reformulations (see Table 5.2)

Given a set of triple patterns, representing a triple pattern and its reformulations, this set can be divided into subsets of compatible triple patterns as follows. We represent each triple pattern as a vertex in a graph, which we refer to as the *reformulation graph*. An edge is then added between two vertices (i.e., two triple patterns) if the two triple patterns are compatible.

Definition 5.2 : Reformulation Graph
Given a set of triple patterns $\{q_i^0, q_i^1, ..., q_i^{m_i}\}$*, the reformulation graph is a graph with* $(m_i + 1)$ *vertices* V *, where* $v_j \in V$ *is a vertex representing triple pattern* q_i^j*, and a set of edges* $E = \{(v_j, v_k) | q_i^j, q_i^k$ *are compatible*$\}$.

Figures 5.1 and 5.2 show the reformulation graphs for the two triple patterns in our example query and their reformulations. To retrieve the sets of compatible triple patterns, we enumerate all *maximal cliques* in the reformulation graph (i.e., complete subgraphs where every triple pattern is compatible with every other triple pattern in the subgraph).

Algorithm 4 which we use to process a triple pattern and all its reformulations makes use of compatible triple patterns in order to efficiently retrieve triples t_i instantiating the triple pattern or any of its reformulations in order of their combined scores $S(t_i, q_i)$ which are computed according to Equation 5.15.

Algorithm. Algorithm 4 takes a set of triple patterns $\{q_i^0, q_i^1, ..., q_i^{m_i}\}$, a set of corresponding weights $\{\lambda_0, \lambda_1, ..., \lambda_{m_i}\}$ and a partitioning C over the triple patterns where each partition $C_k \in C$ contains a subset of the given triple patterns

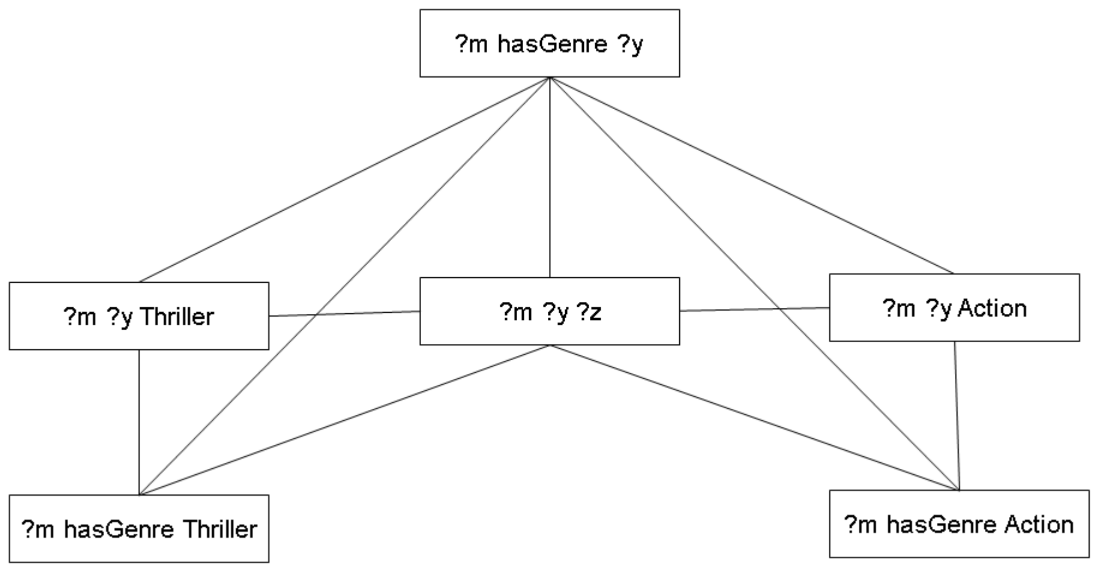


Figure 5.2.: The reformulation graph for the triple pattern `?m hasGenre Thriller` and all its reformulations (see Table 5.2)

that are compatible with each other (i.e., a maximal clique in the reformulation graph). The partitioning C will be used to compute the threshold value the algorithm uses in order to ensure that it retrieves the triple with the highest combined score.

Algorithm 4 uses two main data structures: a *priority queue* Queue and a set of *hashmaps* $seen_j$ for each triple pattern q_i^j. $seen_j$ stores the set of triples t_i retrieved for each triple pattern q_i^j from list L_i^j and their scores $S(t_i, q_i^j)$. All data structures are assumed to be empty before Algorithm 4 is invoked for the first time. Algorithm 4 also maintains $m_i + 1$ values $last(q_i^j)$ which is the score of the last triple triple t_i retrieved from list L_i^j. $last(q_i^j)$ is initially set to the score of the first triple t_i in list L_i^j.

Algorithm 4 iterates over the triple patterns and in each iteration it picks one triple pattern q_i^j to process, calls the method $q_i^j.next()$ and retrieves the next triple for this triple pattern. The algorithm then checks t_i against all the triple patterns $\{q_i^0, q_i^1, ..., q_i^{m_i}\}$ and sets $patterns(t_i)$ to the subset of triple patterns t_i instantiates. Recall that a triple t might instantiate more than one triple pattern q_i^j. This case happens when one of the patterns q_i^j was generated by replacing a resource in q_i with a variable. Once a triple t_i is retrieved, the algorithm adds

Algorithm 4 next($q_i^0, q_i^1, ..., q_i^{m_i} \lambda_0, \lambda_1, ..., \lambda_{m_i}$, C)

```
1: if (FIRST_TIME) then
2:    $Queue \leftarrow \phi$
3:    for ($0 \leq j \leq m_i$) do
4:       $seen_j \leftarrow \phi$, $last(q_i^j) \leftarrow \max_{t_i \in L_i^j} S(t_i, q_i^j)$
5:    end for
6: end if
7: if (Queue is not empty) then
8:    $(t_i, B(t_i, q_i)) \leftarrow Queue.head()$
9:    if $(inst(t_i) = patterns(t_i)$ AND $B(t_i, q_i) \geq \tau)$ then
10:      $(t_i, S(t_i, q_i)) \leftarrow Queue.remove()$
11:      return $(t_i, S(t_i, q_i))$
12:   end if
13: end if
14: while (all triple patterns $q_i^j$ have not been completely processed) do
15:   determine next triple pattern $q_i^j$ to process
16:   $(t_i, S(t_i, q^j)) \leftarrow q_i^j.next()$
17:   $Queue.insert(t_i, 0)$
18:   $inst(t_i) \leftarrow inst(t_i) \cup \{q_i^j\}$
19:   $seen_j.insert(t_i, s(t_i, q_i^j))$
20:   $last(q_i^j) \leftarrow s(t_i, q_i^j)$
21:   for ($t \in Queue$) do
22:      $B(t, q_i) \leftarrow \Sigma_{q_i^j \in inst(t)} \lambda_j seen_j.get(t) + \Sigma_{q_i^j \in patterns(t) \setminus inst(t)} \lambda_j last(q_i^j)$
23:      $Queue.updateScore(t, B(t, q_i))$
24:   end for
25:   if (Queue is not empty) then
26:      $\tau \leftarrow \max_{C_k \in C} \Sigma_{q_i^j \in C_k} \lambda_j last(q_i^j)$
27:      $(t_i, B(t_i, q_i)) \leftarrow Queue.head()$
28:      if ($inst(t_i) = patterns(t_i)$ AND $B(t_i, q_i) \geq \tau$) then
29:         break loop
30:      end if
31:   end if
32: end while
33: if (Queue is not empty) then
34:   $(t_i, S(q_i, t_i)) \leftarrow Queue.head()$
35:   return $(t_i, S(q_i, t_i)$
36: end if
```

the just retrieved triple to the priority queue Queue and updates the best scores of all candidate triples t in the queue as follows:

$$B(t, q_i) = \Sigma_{q_i^j \in inst(t)} \lambda_j S(t, q_i^j) + \Sigma_{q_i^j \in patterns(t) \setminus inst(t)} \lambda_j last(q_i^j)$$

where $inst(t)$ is the set of triple patterns that triple t has been retrieved for so far, $patterns(t)$ is the set of all patterns t instantiates, $S(t, q_i^j)$ is the score of triple t with respect to q_i^j and $last(q_i^j)$ is the score of the last triple retrieved for triple pattern q_i^j.

Once the best scores of all candidate triples have been updated, the algorithm checks the triple t at the head of the queue, and if the true combined score of the triple has been computed (i.e., $inst(t) = patterns(t)$) and its best score is greater than the threshold value τ, it returns the triple as the next triple with the highest score $S(t, q_i)$. The threshold value τ is the best score any triple that has not been retrieved for any of the triple patterns can acquire. The threshold value is computed as follows:

$$\tau = \max_{C_k \in C} \Sigma_{q_i^j \in C_k} \lambda_j last(q_i^j)$$

where C is the set of all maximal cliques in the reformulation graph and C_k is one such clique, and $last(q_i^j)$ is the score of the last triple retrieved for triple pattern $q_i^j \in C_k$.

Processing Order of Reformulations. In each iteration of Algorithm 4, the algorithm must determine the next triple pattern q_i^j to process and then retrieves the next triple from list L_i^j. This can be done in various ways, for instance in a round robin fashion, or by picking the pattern q_i^j with the maximum $last(q_i^j)$ which would result in reducing the best scores of all triples in the queue that have not been retrieved for this triple pattern, or deploying any other heuristics that would result in the algorithm returning the next triple as fast as possible.

Processing Keyword-Augmented Triple Patterns. In case q_i is augmented with keywords $w_1, w_2, ..., w_k$, we would need to also combine Algorithm 3 with Algorithm 4. This can be done very easily using our pipelined architecture. We would just need to modify the call of method $q_i^j.next()$ in Algorithm 4 and call method $q_i^j.next(w_1, w_2, ..., w_k)$ which would then invoke Algorithm 3 to retrieve all the triples t_i instantiating triple pattern q_i^j in order of their combined scores $S(t_i, q_i^j)$ which are computed according to Equation 5.6 given in the previous subsection.

Theorem 5.3 *: Given a triple pattern q_i and a set of its reformulations $\{q_i^0, q_i^1, ..., q_i^{m_i}\}$, Algorithm 4 correctly returns the triple with the highest true combined score.*

Proof: Assume that Algorithm 4 returns triple t_i as the next highest-scored triple instantiating triple pattern q_i. This means that $\mathtt{inst}(t_i) = \mathtt{patterns}(t_i)$, i.e., t_i has been retrieved for all the patterns it instantiates. The score $B(t_i, q_i)$ of triple t_i would then be equal to:

$$B(t_i, q_i) = \Sigma_{q_i^j \in \mathtt{patterns}(t_i)} \lambda_j S(t_i, q_i^j)$$

which is equivalent to:

$$B(t_i, q_i) = \Sigma_{j=1}^{m_i} \lambda_j S(t_i, q_i^j)$$

where $S(t_i, q_i^j) = 0$ if $q_i^j \notin \mathtt{patterns}(t_i)$. Thus, the best score of triple t_i would then be the true combined score of triple t_i.

In addition, since t_i was returned as the next triple, it must have been at the head of the queue which in turn means that the score $S(t_i, q_i)$ would be higher than the best scores of all other triples in the queue. Let one such triple be t_i' and let its current best score stored in the queue be $B(t_i', q_i)$. We thus have:

$$S(t_i, q_i) \geq B(t_i', q_i) \tag{5.16}$$

We now consider three distinct cases. If t_i' has been retrieved for all patterns it instantiates, its best score $B(t_i', q_i)$ would then be its true combined score $S(t_i', q_i)$ which implies that t_i indeed has a higher true combined score than t_i'.

If t_i' has been retrieved for a subset of the patterns it should instantiate, then its best score is computed as follows:

$$B(t_i', q_i) = \Sigma_{q_i^j \in \mathtt{inst}(t_i')} \lambda_j S(t_i', q_i^j) + \Sigma_{q_i^j \in \mathtt{patterns}(t_i) \setminus \mathtt{inst}(t_i')} \lambda_j \mathtt{last}(q_i^j)$$

Since we retrieve the triples for each triple pattern q_i^j in order of their scores, this implies that $\forall\ q_i^j \in \mathtt{patterns}(t_i') \setminus \mathtt{inst}(t_i')$ the following inequality holds:

$$\mathtt{last}(q_i^j) \geq S(t_i', q_i^j) \tag{5.17}$$

Otherwise triple t'_i must have been already retrieved for triple pattern q^j_i or $S(t'_i, q^j_i) = 0$ if t'_i does not instantiate pattern q^j_i. This implies that:

$$B(t'_i, q_i) \geq \Sigma^{m_i}_{j=1} \lambda_j S(t'_i, q^j_i) \tag{5.18}$$

From Inequality 5.16 and Inequality 5.18, we conclude that the true combined score of the triple t_i, which is at the head of the queue must be greater than the true combined score of triple t'_i.

Now, let's consider the case where the triple t'_i was not in the queue. The score of t_i must be greater than or equal to the threshold value τ. That is,

$$S(t_i, q_i) \geq \underset{C_k \in C}{\text{argmax}}\ \Sigma_{q^j_i \in C_k} \lambda_j \mathit{last}(q^j_i) \tag{5.19}$$

Since $\forall\ 1 \leq j \leq m_i$ we have:

$$\mathit{last}(q^j_i) \geq S(t'_i, q^j_i) \tag{5.20}$$

where $S(t'_i, q^j_i)$ is the score of t'_i with respect to triple pattern q^j_i which is zero in case t'_i does not instantiate q^j_i, and since any triple t'_i not yet seen at all, and thus not in the queue, would instantiate at most one of the sets $C_k \in C$, we have:

$$\underset{C_k \in C}{\text{argmax}}\ \Sigma_{q^j_i \in C_k} \lambda_j \mathit{last}(q^j_i) \geq \Sigma^{m_i}_{j=1} \lambda_j S(t'_i, q^j_i) \tag{5.21}$$

From Inequality 5.19 and Inequality 5.21, it is evident that $S(t_i, q_i) \geq S(t'_i, q_i)$ which implies that the true combined score of t_i must be greater than that of any t'_i not in the queue.

From all the above, we conclude that Algorithm 4 indeed returns the next triple t_i instantiating triple pattern q_i with the highest true combined score which is computed according to Equation 5.15.

5.3. Data Store and Indices

We use a relational database as a storage medium for our RDF knowledge base. Our database consists of two tables: TRIPLES and TEXT. The first table stores all the triples in the knowledge base and their witness counts and has the following schema:

TRIPLES(SUBJECT, PREDICATE, OBJECT, WITNESSES).

where WITNESSES is the witness count of the triple. The witness count is an estimate of the number of Web sources that contains the information encoded by the corresponding RDF triple. The witness counts are used by our ranking model to compute the scores of the triples.

In the second table, we store the keywords in the text snippets of the triples. The second table has the following schema:

TEXT(SUBJECT, PREDICATE, OBJECT, KEYWORD, WITNESSES)

where WITNESSES is an estimate of the number of Web sources that contains the information encoded by the corresponding triple and which in addition contain the associated keyword. The witness counts are used by our ranking model to compute the scores of the triples.

We now explain how our algorithms described in Section 5.2 can use these tables in order to *efficiently* retrieve all the necessary information they need to operate. We start with the case of simple triple patterns (i.e., without any keywords) and then consider the case of keyword-augmented triple patterns afterwards.

5.3.1. Instantiation Lists for Triple Patterns.

Algorithms 2 and 4 described in the previous section assume there exists an instantiation list L_i for every triple pattern q_i. L_i contains all the triples t_i instantiating pattern q_i in descending order of their scores $S(t_i, q_i)$ which is computed as follows:

$$S(t_i, q_i) = \frac{c(t_i)}{\Sigma_{t \in L_i} c(t)} \tag{5.22}$$

where $c(t_i)$ is the witness count of triple t_i.

Moreover, Algorithms 2 and 4 keep track of two values in order to compute score bounds: $\mathtt{first}(q_i)$ which is set to the maximum score of all the triples $t_i \in L_i$ and $\mathtt{last}(q_i)$ which is initialized to the maximum score of all the triples $t_i \in L_i$.

Thus, given a triple pattern q_i, we must do the following three tasks:

1. Compute the sum of witness counts of all triples t_i instantiating q_i
2. Compute the maximum score of all triple triples t_i instantiating q_i
3. Retrieve all the triples t_i instantiating triple pattern q_i ordered on theirs scores $S(t_i, q_i)$ which are computed according to Equation 5.22

We now explain how to carry out all these three tasks efficiently using our data store. We start with the third task and then follow with the first two tasks.

Efficient Retrieval of Triples. Given a triple pattern q_i, we can retrieve all the triples instantiating it by issuing a simple SQL select statement. For example, let q_i be:

```
?m  hasGenre  Thriller
```

The following select statement can be issued to retrieve all the triples instantiating this triple pattern ordered on their witness counts:

```
SELECT SUBJECT, PREDICATE, OBJECT, WITNESSES FROM TRIPLES
WHERE PREDICATE = 'hasGenre' AND OBJECT = 'Thriller'
ORDER BY WITNESSES DESC
```

The ResultSet of the above SQL statement would then be the instantiation list of triple pattern q_i and each time the method $q_i.next()$ is invoked from Algorithm 2 or 4, the next triple from the ResultSet corresponding to triple pattern q_i would be retrieved and its score would be computed by normalizing its witness count by the total witness count of all the triples in the ResultSet.

In order to be able to efficiently process SQL statements like the one above, we make use of the following observation. Given a triple pattern $q = (s, p, o)$ with subject s, predicate p and object o, there are three different possibilities: 1) all three components are variables, 2) two of the three components are variables, or 3) one of the three components is a variable.

In case all three components of the triple pattern $q = (s, p, o)$ are variables, all the triples in table TRIPLES would instantiate this pattern. We thus create an index INDEX000 over all the fields in table TRIPLES where the triples are sorted in descending order based on their witness counts WITNESSES.

In case two of the three components of the triple pattern $q = (s, p, o)$ are variables, there are three distinct cases: 1) s and p are variables, in which case all the triples with object o would instantiate the triple pattern, 2) p and o are variables, in which case all the triples with subject s would instantiate the triple pattern or 3) s and o are variables, in which case all the triples with predicate p would instantiate the triple pattern.

We thus create three indices over all the fields in table TRIPLES where in the first index INDEX100 the triples are sorted based on their subjects and then descendingly on their witness counts. In the second index INDEX010, the triples are sorted based on their predicates and then descendingly on their witness counts. In the third index INDEX001 the triples are sorted based on their objects and then descendingly on their witness counts. Using indices INDEX100, INDEX010 and INDEX001, we can quickly retrieve the set of triples with a given subject, predicate or object,respectively, where the triples are sorted descending order of their witness counts.

Finally, in case one of the three components of the triple pattern $q = (s, p, o)$ is a variable, there are again three distinct cases: 1) s is a variable, in which case all the triples with predicate p and object o would instantiate the triple pattern, 2) p is variable, in which case all the triples with subject s and object o would instantiate the triple pattern or 3) o is variable, in which case all the triples with subject s and predicate p would instantiate the triple pattern.

We thus create three indices over all the fields in table TRIPLES where in the first index INDEX110 the triples are sorted based on their subjects, then on their predicates and then descendingly on their witness counts. In the second index INDEX011, the triples are sorted based on their predicates, then on their objects and then descendingly on their witness counts. In the third index INDEX101 the triples are sorted based on their subjects, then on their objects and then descendingly on their witness counts. Using indices INDEX110, INDEX011 and INDEX101, we can quickly retrieve the set of triples with a given subject and a given predicate, a given predicate and a given object, or a given subject and a given object,respectively, where the triples are sorted descendingly based on their witness counts.

Efficient Computation of Sum of Witness Counts and Maximum Scores. To compute the sum of witness counts of all the triples instantiating a triple pattern q_i, we can issue an aggregation SQL statements. For example, let q_i be:

```
?m  hasGenre  Thriller
```

The following aggregation statement can be issued to retrieve the sum of witness counts of all the triples instantiating this triple pattern:

```
SELECT SUM(WITNESSES) FROM TRIPLES
WHERE PREDICATE = 'hasGenre' AND OBJECT = 'Thriller'
```

Similarly, to compute the maximum score of all triples instantiating a triple pattern, an aggregation SQL statement over the table TRIPLES can be issued. For example, the following SQL statement can be issued to compute the maximum witness count of all the triples instantiating our example triple pattern q_i:

```
SELECT MAX(WITNESSES) FROM TRIPLES
WHERE PREDICATE = 'hasGenre' AND OBJECT = 'Thriller'
```

The maximum score would then be the result of the above statement divided by the result of the previous one. To be able to efficiently process SQL statements like the ones above, we make use of a set of materialized views. In particular, we create 7 materialized views that account for all the possible triple patterns that our triples can instantiate. In these 7 materialized views, we store the sum of witness counts of all the triples that instantiate each possible triple pattern and the maximum witness count of all triples instantiating each possible triple pattern.

5.3.2. Instantiation Lists for Keyword-Augmented Triple Patterns

Algorithm 3 described in Section 5.2 assumes there exists an instantiation list L_{ij} for every triple pattern q_i and keyword w_j. L_{ij} contains the triples instantiating pattern q_i and whose text snippets contain the keyword w_j and the triples t_i are sorted in descending order of their scores $S(t_i, q_i, w_j)$ which are computed as follows:

$$S(t_i, q_i, w_j) = \frac{c(t_i; w_j)}{\sum_{t \in L_{ij}} c(t; w_j)} \tag{5.23}$$

$c(t_i; w_j)$ is the number of witnesses of triple t_i that contain the keyword w_j.

Moreover, Algorithms 3 keeps track of the value $last(w_j)$ which is initialized to the maximum score of all the triples $t_i \in L_{ij}$.

Thus, given a triple pattern q_i, we must do the following three tasks:

1. Compute the sum of witness counts of all triples t_i instantiating q_i and whose text snippets contain the keyword w_j

2. Compute the maximum score of all triples t_i instantiating q_i and whose text snippets contain the keyword w_j
3. Retrieve all triples t_i instantiating triple pattern q_i and whose text snippets contain the keyword w_j ordered on theirs scores $S(t_i, q_i, w_j)$ which are computed according to Equation 5.23

We now explain how to carry out all three tasks efficiently using our data store. We start with the third task and then follow with the first two tasks.

Efficient Retrieval of Triples. Given a triple pattern q_i and a keyword w_j, we can retrieve all the triples instantiating q_i and whose text snippets contain the keyword w_j by issuing a simple SQL select statement. For example, let w_j be *killer* and let q_i be:

```
?m  hasGenre  Thriller
```

The following select statement can be issued to retrieve all the triples instantiating this triple pattern and whose text snippets contain the keyword *killer* ordered on their witness counts:

```
SELECT SUBJECT, PREDICATE, OBJECT, WITNESSES FROM TEXT
WHERE PREDICATE = 'hasGenre' AND OBJECT = 'Thriller' AND
KEYWORD='killer'
ORDER BY WITNESSES DESC
```

The ResultSet of the above SQL statement would then be the instantiation list L_{ij} and each time the method $q_i.next(w_j)$ is invoked from Algorithm 3, the next triple from the ResultSet would be retrieved and its score would be computed by normalizing its witness count by the total witness count of all the triples in the ResultSet.

In order to be able to efficiently process SQL statements like the one above, we make use of a set of indices as we did in the case of triple-patterns only. In particular, we create 7 indices that account for the 7 possible triple patterns that a triple can instantiate. For example, we create an index INDEX1000 over all the fields in table TEXT where the triples are sorted first on the keywords and then descendingly based on their witness counts WITNESSES. We also create three indices over all the fields in table TEXT where in the first index INDEX1100 the triples are sorted based on the keywords, then on their subjects and then descendingly on their witness counts. The rest of the indices are analogous to the case of the triple patterns only.

Efficient Computation of Sum of Witness Counts and Maximum Scores. To compute the sum of witness counts of all the triples in an instantiation list L_{ij} corresponding to a triple pattern q_i and a keyword w_j, we can issue an aggregation SQL statements. For example, let w_j be *killer* and let q_i be:

```
?m  hasGenre  Thriller
```

The following aggregation statement can be issued to retrieve the total witness counts of all the triples $t_i \in L_{ij}$

```
SELECT SUM(WITNESSES) FROM TEXT
WHERE PREDICATE = 'hasGenre' AND OBJECT = 'Thriller' AND
KEYWORD='killer'
```

Similarly, to compute the maximum score of all the triples in the instantiation list L_{ij}, a similar aggregation SQL statement over the table TEXT can be issued. To be able to efficiently process such aggregation SQL statements, we make use of a set of materialized views. In particular, we create 7 materialized views that account for all the possible triple patterns. In each one of these materialized views, there is a row for a triple pattern, a keyword and the sum and the maximum of the witness counts of all the triples that instantiate the pattern and whose text snippets contain the keyword.

5.4. Related Work

There is a wealth of work on top-k query processing. A survey of top-k processing techniques is throughly given in [44]. We just pinpoint here the most relevant approaches and contrast them to our setting. Our problem of processing triple-pattern queries and retrieving the top-k highest-ranked results is closely related to the problem of top-k processing of selection queries. For selection queries, the goal is to apply a scoring function on multiple attributes of the same relation to select tuples ranked on their combined scores. The problem is tackled in different contexts. In middleware environments, Fagin [27] and Fagin et al. [28] introduce the first efficient set of algorithms to answer ranking queries. Database objects with m attributes are viewed as m separate lists, and each supports sorted and, possibly, random access to object scores. The TA algorithm [28] assumes the availability of random access to object scores in any list besides the sorted access to each list. The NRA algorithm [28] assumes only sorted access is available to individual lists. The family of Fagin's top-k algorithms are believed to be the essence of top-k processing in the context of document retrieval. However, such family of algorithms is not applicable to our setting, where we process triple-pattern queries that involve joining multiple lists of triples based on join conditions specified in the query.

The more general problem of the top-k rank-join is addressed in [64]. The authors introduce the J^* algorithm to join multiple ranked inputs to produce a global rank. J^* maps the rank-join problem to a search problem in the Cartesian space of the ranked inputs. J^* uses a version of the A^* search algorithm to guide the navigation in this space to produce the ranked results. Although J^* shares the same goal of joining multiple lists, and can be directly adopted to the setting of triple-pattern queries over RDF data, it needs vast changes to be able to handle our advanced setting of keyword-augmented triple-patterns, and automatic query reformulations. Moreover, Ilyas et al. [43] have compared their top-k rank-join algorithm to the J^* algorithm and have shown significant enhancements in the overall performance. We adopt the top-k rank-join algorithm from [43] and extend it to handle keyword-augmented triple-pattern queries, and automatic query reformulation.

Another closely related work is TopX [81], which is a top-k retrieval engine for text and semistructured data. It terminates query execution as soon as it can safely determine the top-k ranked results according to a monotonic score aggregation function with respect to a multidimensional query. It efficiently supports vague search on both content- and structure-oriented query conditions for dynamic query relaxation with controllable influence on the result ranking. However, TopX deals mainly with selection queries and is not directly applicable to the case of triple-pattern queries where a top-k rank-join must be performed.

#entities	Example entity types	#triples	Example relations
LibraryThing Dataset			
48,000	`book`, `author` `user`, `tag`	700,000	`wrote`, `hasFriend` `hasTag`, `type`
IMDB Dataset			
59,000	`movie`, `actor` `director`, `producer`, `country` , `language`	600,000	`actedIn`, `directed` `hasWonPrize`, `isMarriedTo`, `produced`, `hasGenre`

Table 5.5.: Overview of the datasets

5.5. Experimental Evaluation

In this section, we evaluate the performance of our top-k query-processing framework. We conducted two experiments. The first was used to test the performance of our algorithms for triple-pattern queries without any query reformulation. The second set of experiments was used to evaluate the performance of our framework when automatic query reformulation was performed. All experiments were conducted on a Dual-Xeon-3GHz machine with 4GB of RAM, out of which up to 1GB were used for the experiments. We used an Oracle database running on the same machine as the storage backend for our knowledge base.

5.5.1. Datasets

All experiments were conducted over two datasets. The first dataset was derived from the LibaryThing community. The second dataset was derived from a subset of the Internet Movie Database (IMDB). The data from both sources was automatically parsed and converted into RDF triples. In addition, each triple was also augmented with keywords derived from the data source it was extracted from. In particular, for the IMDB dataset, all the terms in the plots, taglines and keywords fields were extracted, stemmed and stored with each triple. For the LibraryThing dataset, since we did not have enough textual information about the entities present, we retrieved the books' Amazon descriptions and the authors' Wikipedia pages and used them as textual context for the triples. Table 5.5 gives an overview of the datasets.

5.5.2. Experiment 1

In the first experiment, we focused on evaluating the performance of our query-processing framework for the case of a single triple-pattern query only, i.e., without taking into consideration query reformulation.

Query Benchmark

We used 2 different sets of evaluation queries. The first set consisted of simple triple-pattern queries that ranged from single-pattern queries to multi-pattern graph queries. We constructed *16 queries* for the IMDB dataset and *8 queries* for the LibraryThing dataset. Using this set we evaluated the performance of Algorithm 2 alone.

The second set consisted of keyword-augmented queries. Again, we constructed *16 queries* for the IMDB dataset and *8 queries* for the LibraryThing dataset. The queries were triple-pattern queries associated with one or more keywords. This set of evaluation queries was used to evaluate the performance of Algorithm 3.

Since we did not consider query reformulation here, all the queries in our benchmark were designed so that they would have a sufficiently large number of results (i.e., more than 100) so that retrieving the top-k highest-ranked results would make sense. Some example evaluation queries for both datasets is shown in Table 5.6. Appendix B shows the complete list of evaluation queries used in our first experiment.

Compared Approaches

Baseline Approach. The baseline approach works as follows. Given a triple-pattern query $Q = (q_1, q_2, .., q_n)$, it retrieves the instantiation lists for each triple pattern q_i. It then utilizes an in-memory hash-join operator to join the instantiation lists of the different triple patterns, and retrieves result tuples satisfying the query join conditions. Finally, the scores of the result tuples are computed according to our scoring function, and the results are ranked descendingly based on their scores.

IMDB			
?m	hasGenre	Thriller	
?d	directed	?m	
?a1	isMarriedTo	?a2	
?a1	actedIn	?m	
?a2	actedIn	?m	
?d	hasWonPrize	Academy_Award_for_Best_Director	
?d	directed	?m	
?a	actedIn	?m	
?a	hasWonPrize	Academy_Award_for_Best_Actor	
?x	hasWonPrize	Academy_Award_for_Best_Actor	
?y	hasWonPrize	Academy_Award_for_Best_Actress	
?x	actedIn	?m	*love*
?y	actedIn	?m	*relationship*
?x	directed	?y	*true story*
?x	hasWonPrize	?z	
LibraryThing			
?x	wrote	?y	
?y	hasTag	Series	
?x	wrote	?y	
?y	hasTag	Fiction	
?x	wrote	?z	
?z	hasTag	Non-fiction	
?x	wrote	?y	
?y	type	Mystery_&_Thrillers	
?x	wrote	?y	*civil war*
?x	type	Novelists	
?y	hasTag	Movie	
?x	wrote	?y	
?y	hasTag	Magic	
?y	type	Fiction	*award*

Table 5.6.: A subset of the evaluation queries

IMDB					
Top-k					
Metric	$k = 5$	$k = 10$	$k = 20$	$k = 50$	$k = 100$
Runtime (in secs)	1.381	1.611	1.616	1.635	1.668
#SA	32000	39243	39306	39558	39858
Baseline					
Metric	$k = 5$	$k = 10$	$k = 20$	$k = 50$	$k = 100$
Runtime (in secs)	91.101	91.101	91.101	91.101	91.101
#SA	93970	93970	93970	93970	93970
LibraryThing					
Top-k					
Metric	$k = 5$	$k = 10$	$k = 20$	$k = 50$	$k = 100$
Runtime (in secs)	1.172	1.213	1.239	1.312	1.668
#SA	9277	9296	9325	9461	10905
Baseline					
Metric	$k = 5$	$k = 10$	$k = 20$	$k = 50$	$k = 100$
Runtime (in secs)	8.517	8.517	8.517	8.517	8.517
#SA	26911	26911	26911	26911	26911

Table 5.7.: Results for triple-pattern queries with no keywords and no reformulation

Top-k Approach. The top-k approach utilizes our top-k processing algorithms (Algorithm 2 and Algorithm 3) where the order in which we process the triple patterns and keywords is guided by the scores of the triples last retrieved as we explained in Section 5.2.

Results

The results of our first experiment are shown in Table 5.7 for simple triple-pattern queries that do not involve any keywords. We computed the runtime taken by each approach to report the top-k highest-ranked results of each query (varying k from 5 to 100). We then averaged the total runtime overall evaluation queries, and reported this average in Table 5.7. Similarly, we computed

Metric	$k = 5$	$k = 10$	$k = 20$	$k = 50$	$k = 100$
IMDB					
Top-k					
Runtime (in secs)	45.787	45.891	46.179	46.194	46.225
#SA	31751	34269	37872	37911	37964
Baseline					
Runtime (in secs)	79.627	79.627	79.627	79.627	79.627
#SA	83681	83681	83681	83681	83681
LibraryThing					
Top-k					
Runtime (in secs)	1.981	2.214	2.320	2.534	2.638
#SA	6609	6922	8155	10565	11914
Baseline					
Runtime (in secs)	7.224	7.224	7.224	7.224	7.224
#SA	24747	24747	24747	24747	24747

Table 5.8.: Results for keyword-augmented triple-pattern queries with no reformulation

the total number of triples accessed in order to report the top-k results which we denote by (#SA) (i.e., sorted accesses) and averaged this overall evaluation queries. Note that the baseline approach would need to retrieve all the triples for each triple pattern in each query, join the triples together and then rank the result tuples obtained from joining the triples retrieved for each triple pattern, and thus it takes the same time to report the top-k results independent of how small or big k is.

As can be seen from Table 5.7, Algorithm 2 substantially improves over the baseline approach in terms of both the response time to queries, as well as the total number of sorted accesses needed to report the top-k results. In particular, our top-k algorithm shows superior performance for the IMDB dataset by orders of magnitude since much more triples were accessed by the baseline approach in order to retrieve the result tuples for the evaluation queries. This indeed proves that for large datasets, where queries involve joining and ranking a large number of triples, top-k processing is crucially needed.

In Table 5.8, we show the results for the case of keyword-augmented triple-pattern queries. Again, we show the average runtimes and the number of triples accessed by each approach, averaged overall keyword-augmented evaluation queries. Our top-k processing framework with its pipelined architecture for processing keyword-augmented triple patterns shows high improvements over the baseline approach in terms of both runtimes and sorted accesses. However, the improvements are not as high as in the case of simple triple-patterns with no keywords. We believe that this is due to the pipelined architecture where keyword-augmented triple patterns are processed separately using Algorithm 3 which fetches the triples for a keyword-augmented triple pattern and feeds them into the main rank-join algorithm to join them with the triples retrieved for other triple patterns. In the next subsection, we propose few modifications to our top-k processing framework that should overcome this additional overhead endured by the pipelined architecture.

5.5.3. Experiment 2

The second experiment was designed to test the effect that automatic query reformulation has on the performance of query processing. We tested the two approaches to process a query and its reformulations that we presented in Subsection 5.2.3. The first, which we coin the *incremental processing* approach, processes the original query first using our top-k query-processing framework. In case k is greater than the total number of results of the original query, the incremental processing approach proceeds by processing the closest reformulation to the original query and appending the results of the reformulated query to the retrieved results of the original query. In case the number of results retrieved so far is still less than k, the incremental processing approach processes the next closest reformulation and so on until k results have been retrieved or all the reformulations have been processed.

The second approach, which we refer to as the *batch processing* approach, considers all the queries together, and retrieves the result tuples in order of their scores with respect to the original query and all its reformulations. The batch processing approach utilizes Algorithm 4 in order to retrieve the triples instantiating a given triple pattern or any of its relaxations in order of their scores which are computed as we described in Subsection 5.2.3.

Query Benchmark

We did not use the same set of evaluation queries from the first experiment since our first query benchmark consisted of queries that have a large number of results (i.e., greater than 100) and thus our top-k processing approaches when $k \leq 100$ would not consider any query reformulations. We thus constructed a new query benchmark where most of the queries have very few results (typically less than 10) and thus we ensured that our top-k processing approach would have to consider reformulated queries in order to retrieve the top-k highest scored results.

Similar to the first experiment, we used 2 different sets of evaluation queries. The first set consisted of simple triple-pattern queries that ranged from single-pattern queries to multi-pattern graph queries.We constructed 40 queries for each dataset. The second set consisted of keyword-augmented queries. We constructed 15 queries for each dataset. The queries were triple-pattern queries associated with one or more keywords. A subset of the evaluation queries used for both datasets is shown in Table 5.9. Appendix C shows the complete list of evaluation queries used in our second experiment.

Compared Approaches

Baseline Approach. The baseline approach works as follows. Given a triple-pattern query $Q = (q_1, q_2, .., q_n)$, it retrieves for each triple pattern q_i an instantiation list consisting of all the triples instantiating pattern q_i or any of its reformulations. It then utilizes an in-memory hash-join operator to join the instantiation lists of the different triple patterns, and retrieves result tuples satisfying the query join conditions. Finally, the scores of the result tuples are computed according to our scoring function, and the results are ranked descendingly based on their scores.

Top-k Approach. The top-k approach works exactly as in the case of no query reformulation. In case we are utilizing the batch processing approach, Algorithm 4 picks the triple pattern to process next based on the scores of the triples last retrieved.

IMDB			
?m	hasGenre	Comedy	
?m	hasWonPrize	Academy_Award	
?a	hasWonPrize	Academy_Award_for_Best_Actor	
?a	originatesFrom	New_York	
?m1	hasGenre	Mystery	
?m1	hasPredecessor	?m2	
?d1	directed	?m1	
?d2	directed	?m2	
?d	directed	?m	*true story*
?d	hasWonPrize	Academy_Award_for_Best_Director	
?a	actedIn	?m	*school friends*
?a	type	singer	
LibraryThing			
?b	type	Nonfiction	
?b	hasTag	Greek	
?w	type	Historian	
?w	wrote	?b	
?b	hasTag	Memoir	
?w	wrote	?b	
?b	hasTag	Non-fiction	
?b	hasTag	Pulitzer	
?w	wrote	?b	*nobel prize*
?b	hasTag	British_Literature	
?w	wrote	?b	*civil war*
?b	hasTag	Film	

Table 5.9.: Subset of the evaluation queries

IMDB					
Top-k					
Metric	$k = 5$	$k = 10$	$k = 20$	$k = 50$	$k = 100$
Runtime (in secs)	1.589	2.597	3.396	7.394	10.812
#SA	15108	16492	18460	22661	25163
Baseline					
Metric	$k = 5$	$k = 10$	$k = 20$	$k = 50$	$k = 100$
Runtime (in secs)	76.982	126.963	164.157	352.854	472.152
#SA	33736	34503	39038	49334	51970
LibraryThing					
Top-k					
Metric	$k = 5$	$k = 10$	$k = 20$	$k = 50$	$k = 100$
Runtime (in secs)	0.733	1.034	2.027	3.048	4.005
#SA	8205	8277	10550	13897	15992
Baseline					
Metric	$k = 5$	$k = 10$	$k = 20$	$k = 50$	$k = 100$
Runtime (in secs)	9.202	12.919	21.178	46.000	56.693
#SA	19735	21027	25884	32723	35753

Table 5.10.: Results for triple-pattern queries with no keywords and with incremental processing of reformulated queries

Incremental Processing Results

Table 5.10 shows the average runtimes and number of triples accessed for simple-triple pattern queries and with incremental processing of query reformulations. We again show superior performance when using our top-k framework as compared to the baseline approach, with gains in the orders of magnitudes, for both datasets. Note that in the case of incremental processing of reformulated queries, the runtime of the baseline approach also increases as the number of required results (k) increases. This is due to the fact that as k increases, more queries would be processed, and thus more triples would be retrieved, joined and ranked.

Similar to the case of simple triple-patterns, our top-k framework clearly outperforms the baseline approach for the case of keyword-augmented queries with

IMDB					
Top-k					
Metric	$k = 5$	$k = 10$	$k = 20$	$k = 50$	$k = 100$
Runtime (in secs)	39.579	39.585	39.701	57.109	62.449
#SA	15835	15854	15981	21200	24526
Baseline					
Metric	$k = 5$	$k = 10$	$k = 20$	$k = 50$	$k = 100$
Runtime (in secs)	65.140	65.140	66. 767	66.771	80.739
#SA	65998	65999	63427	63431	67595
LibraryThing					
Top-k					
Metric	$k = 5$	$k = 10$	$k = 20$	$k = 50$	$k = 100$
Runtime (in secs)	13.721	13.905	21.793	38.716	51.949
#SA	4777	6239	10218	11611	13012
Baseline					
Metric	$k = 5$	$k = 10$	$k = 20$	$k = 50$	$k = 100$
Runtime (in secs)	5.883	5.883	6.340	118.609	121.624
#SA	23133	23133	23234	42658	46264

Table 5.11.: Results for keyword-augmented triple-pattern queries with incremental processing of reformulated queries

incremental processing of reformulations. Similar to the case when we did not consider query reformulation at all, the gains in performance for the top-k approach over the baseline approach are smaller, and we again credit this to the pipelined approach we have opted for to process keyword-augmented triple patterns.

Batch Processing Results

In Table 5.12, we show the average runtimes and the average number of triples accessed for our simple triple-pattern evaluation queries. However, we deployed a batch processing of query reformulation in this case. Recall that in the batch processing approach, each triple pattern and all its reformulations are

Metric	k = 5	k = 10	k = 20	k = 50	k = 100
IMDB					
Top-k					
Runtime (in secs)	67.905	68.059	68.318	68.682	68.833
#SA	29206	30986	33062	36509	38561
Baseline					
Runtime (in secs)	121.293	121.293	121.293	121.293	121.293
#SA	77720	77720	77720	77720	77720
LibraryThing					
Top-k					
Runtime (in secs)	69.752	69.881	70.005	70.013	70.460
#SA	15335	15677	15858	15867	15913
Baseline					
Runtime (in secs)	171.650	171.650	171.650	171.650	171.650
#SA	30579	30579	30579	30579	30579

Table 5.12.: Results for triple-pattern queries with no keywords and with batch processing of reformulated queries

processed together in batch mode using Algorithm 4. This is achieved by utilizing a pipelined architecture, where each triple pattern and all its reformulations are handled separately using Algorithm 4 that retrieves the triples in order of their combined scores. The triples retrieved for each triple pattern (and its reformulations) are then fed into the main rank-join algorithm (Algorithm 2), which performs a top-k rank-join and reports the result tuples in order of their overall combined scores.

We again show high improvements in terms of both runtime, and number of triples that must be accessed.

Finally, in Table 5.13, we show the results of our complete top-k processing framework, with keywords and batch processing of reformulated queries. We again show vast improvements over the baseline approach in terms of both runtime and the number of triples that must be accessed before the top-k results are reported.

Metric	$k = 5$	$k = 10$	$k = 20$	$k = 50$	$k = 100$
IMDB					
Top-k					
Runtime (in secs)	52.182	52.314	52.634	72.774	74.586
#SA	21459	22707	24195	32135	34129
Baseline					
Runtime (in secs)	256.826	256.826	256.826	256.826	256.826
#SA	135613	135613	135613	135613	135613
LibraryThing					
Top-k					
Runtime (in secs)	33.666	33.818	41.341	45.882	46.095
#SA	6437	7917	11954	14617	16264
Baseline					
Runtime (in secs)	115.701	115.701	115.701	115.701	115.701
#SA	49883	49883	49883	49883	49883

Table 5.13.: Results for keyword-augmented triple-pattern queries with batch processing of reformulated queries

5.5.4. Discussion and Possible Extensions

In our top-k processing framework, we opted for a pipelined approach to handle keyword-augmented triple patterns, and triple-pattern reformulations. While this nicely encapsulates the way we process the keyword-augmented triple patterns, and the reformulated triple patterns, it introduces additional overhead as we would need to retrieve the triples instantiating a keyword-augmented triple-pattern in order of their *true scores* before joining them with the other triple patterns in our main rank-join algorithm. One way to overcome this is to allow Algorithm 3 and Algorithm 4 to periodically report their best candidates along with their score bounds, and to modify Algorithm 2 to make use of these bounds in order to compute the top-k joined result tuples.

Another possible modification to our algorithms that we believe could improve their performance is to consider different scheduling strategies for processing the different triple patterns in a given query. That is, in addition to the score-guided strategy of picking the next triple pattern to process, this can be combined with a heuristic that retrieves more triples from triple patterns that are not associated with any keywords. By doing so, we reduce the number of expensive calls to Algorithm 3.

All the algorithms we presented make the assumption that only sorted accesses are allowed. If we relax this condition and allow for random accesses, this might prove beneficial in enhancing the performance of our algorithms, since the encapsulated top-k processing algorithms would then report triples much earlier than in the case of sorted accesses only.

Finally, we could also modify our top-k processing framework and instead of using a pipelined architecture, we could fold each keyword-augmented triple pattern and create a replica of the triple pattern for each keyword. We could then holistically process all these folded triple patterns together in a top-k fashion. Similarly, for query reformulation, we can retrieve for each triple pattern, all its reformulations, and then process all such triple patterns with other triple patterns in the query, and all their reformulations. While this might prove beneficial in improving the performance of our top-k algorithms, it might be less appealing from a design point of view. In particular, it endures vast complexities in the algorithms that are used to compute the top-k results, especially for the case when we have both keyword-augmented triple patterns and query reformulation.

5.6. Summary

In this chapter, we presented a set of algorithms to efficiently process triple-pattern queries, possibly augmented with keywords, and with automatic query reformulation in place. Our framework is based on a pipelined architecture that encapsulates the problem of retrieving triples instantiating keyword-augmented patterns in order of their combined scores. We also use encapsulation to retrieve triples instantiating a triple pattern and any of its reformulations in order of their combined scores. We have shown through experimental evaluation that our top-k processing approach outperforms a naive baseline approach that first joins all the triples instantiating the triple patterns in the query (and their reformulations) to retrieve the result tuples for the overall query, and then sorts the result tuples based on their combined scores. We also presented a set of possible extensions to our framework that we believe can prove very beneficial in enhancing the performance of our framework even further.

Chapter 6.

Keyword Search

Searching RDF knowledge bases using the expressive triple-pattern queries is a very powerful tool that allows users to find very concise answers to their information needs. However, triple-pattern search is more tailored for advanced users or search APIs. The casual users are accustomed to keyword search which has become the paradigm to perform IR tasks on the Web. In order to increase the usability of RDF data, it is crucial to allow users to search RDF knowledge bases using keyword queries. In this chapter, we present a retrieval model for keyword queries over RDF knowledge bases. Our model retrieves tuples of RDF triples that match the query keywords, and ranks them based on statistical language-models.

6.1. Query Framework

Our knowledge base consists of a set of SPO triples such as the ones shown in Table 6.1. To be able to process keyword queries, we associate with each triple t_i a document which we refer to as D_i. D_i contains a *textual representation* of the information encoded in t_i. In the simplest case, D_i would contain a set of representative keywords for the subject, predicate and object of the triple. These representative keywords can be generated using an external dictionary, or by utilizing the textual extraction-patterns in case the triples were extracted using some IE technique from free-text [79]. Moreover, in case there is any additional textual information associated with the triple t_i, such as any contextual text present in the sources from which the triple was extracted [23], we can also extract all the keywords there and add them to D_i.

Subject (S)	**Property (P)**	**Object (O)**
`Traffic`	`hasWonPrize`	`Academy_Award`
`Innerspace`	`hasWonPrize`	`Academy_Award`
`Innerspace`	`hasGenre`	`Comedy`
`Joe_Dante`	`directed`	`Innerspace`
`Toy_Story`	`hasWonPrize`	`Academy_Award`
`Road_Trip`	`hasGenre`	`Comedy`
`Toy_Story`	`hasGenre`	`Comedy`
`Tom_Hanks`	`actedIn`	`Toy_Story`
`Diner`	`hasWonPrize`	`Academy_Award`
`Diner`	`type`	`Comedy_films`
`Steve_Guttenberg`	`actedIn`	`Diner`
`The_Pink_Panther`	`type`	`Criminal_comedy_films`
`The_Pink_Panther`	`hasWonPrize`	`Academy_Award`
`Police_Academy`	`type`	`Comedy_films`
`Steve_Guttenberg`	`actedIn`	`Police_Academy`
`The_Darwin_Awards`	`type`	`Comedy_films`

Table 6.1.: An example RDF knowledge base about movies

Definition 6.1 : Triple Document
The document D_i *of triple* t_i *is a bag of words such that* $c(w; D_i)$ *is the frequency of occurrence of term* w *in* D_i*.*

For example, the triple

```
Innerspace  hasWonPrize  Academy_Award
```

would be associated with the following document:

innerspace won prize academy award 1988 oscar best visual effects

In this chapter, we present a retrieval model that takes as an input a keyword query and retrieves a set of tuples consisting of triples matching the given query. More precisely, for each query keyword, our model retrieves all triples whose documents match this keyword. Once we have for each keyword a list of matching triples, we join them based on their *subjects* and *objects* to retrieve tuples with one or more connected triples. However, we only construct tuples that

Innerspace	hasGenre	Comedy
Road_Trip	hasGenre	Comedy
Toy_Story	hasGenre	Comedy
Diner	type	Comedy_films
The_Pink_Panther	type	Criminal_comedy_films
Police_Academy	type	Comedy_films
The_Darwin_Awards	type	Comedy_films

Table 6.2.: The List of matching triples for the keyword *comedy*

contain triples from *different* lists, corresponding to matches to different (sets of) keywords. The intuition behind this is that we assume that the user has a precise information need in mind that can be precisely represented using a set of triple patterns. However, since the user cannot express her information need using triple patterns, she represents each triple pattern using a set of keywords. Without any knowledge about which keywords map to which triple pattern, we need to consider all extremes: from all keywords representing a single triple pattern up to each single keyword representing an individual triple pattern. Thus, the results to the user information need would be tuples with one or more triples up to the number of keywords in the user query.

Definition 6.2 : Query Result
Given a keyword query $Q = \{q_1, q_2, ..., q_m\}$*, where* q_i *is a keyword, a result is defined as a tuple* $T = (t_1, t_2, ..., t_n)$ *where* t_i *is a triple such that*

1. $\forall\, t_i, Q \cap D_i \neq \phi$,
2. $\forall\, t_i, t_j, Q \cap D_i \neq Q \cap D_j$ *and*
3. $\forall\, t_i \in T, \exists\, t_j$ *such that* $\mathsf{subject}(t_i) = \mathsf{subject}(t_j)$, $\mathsf{subject}(t_i) = \mathsf{object}(t_j)$, $\mathsf{object}(t_i) = \mathsf{subject}(t_j)$ *or* $\mathsf{object}(t_i) = \mathsf{object}(t_j)$.

For example, consider the information need of finding comedies that have won the Academy Award. Furthermore assume that the user expressed this information need using the keyword query *comedy academy award*. Tables 6.2, 6.3 and 6.4 show the list of matching triples for each query keyword. The results

Traffic	hasWonPrize	Academy_Award
Innerspace	hasWonPrize	Academy_Award
Toy_Story	hasWonPrize	Academy_Award
Diner	hasWonPrize	Academy_Award
The_Pink_Panther	hasWonPrize	Academy_Award
Police_Academy	type	Comedy_films

Table 6.3.: The list of matching triples for the keyword *academy*

Traffic	hasWonPrize	Academy_Award
Innerspace	hasWonPrize	Academy_Award
Toy_Story	hasWonPrize	Academy_Award
Diner	hasWonPrize	Academy_Award
The_Pink_Panther	hasWonPrize	Academy_Award
The_Darwin_Awards	type	Comedy_films

Table 6.4.: The list of matching triples for the keyword *award*

for such a query would then be single triples matching one or more query keywords, tuples of two triples matching at least two query keywords and so on. Table 6.5 shows all the tuples retrieved for the example query from our example RDF knowledge base in Table 6.1. The second column shows the set of matched keywords by each triple in the corresponding tuple.

Note that all the tuples retrieved are *unique* and *maximal*. That is, each tuple is not a subset of any other tuple retrieved. Also note that the tuples contain only triples that match at least one query keyword. This can be extended to retrieve tuples that contain triples that do not match any query keywords by exploring the underlying RDF knowledge base and retrieving Steiner trees [41, 46, 39, 32] or other graph-structured components [55]. This is similar in spirit to query expansion in traditional IR.

Since keyword queries introduce additional ambiguity that is not present in the case of triple-pattern queries, result ranking becomes very crucial. For instance, the tuples in Table 6.5 differ in their size (i.e., how many triples they contain) as well as how many keywords they match. In addition, they also differ

Tuple	**Keywords**		
`Traffic`	`hasWonPrize`	`Academy_Award`	*academy award*
`Innerspace`	`hasGenre`	`Comedy`	*comedy*
`Innerspace`	`hasWonPrize`	`Academy_Award`	*academy award*
`Toy_Story`	`hasGenre`	`Comedy`	*comedy*
`Toy_Story`	`hasWonPrize`	`Academy_Award`	*academy award*
`Road_Trip`	`hasGenre`	`Comedy`	*comedy*
`Diner`	`type`	`Comedy_films`	*comedy*
`Diner`	`hasWonPrize`	`Academy_Award`	*academy award*
`The_Pink_Panther`	`type`	`Criminal_comedy_films`	*comedy*
`The_Pink_Panther`	`hasWonPrize`	`Academy_Award`	*academy award*
`Police_Academy`	`type`	`Comedy_films`	*academy comedy*
`The_Darwin_Awards`	`type`	`Comedy_films`	*award comedy*

Table 6.5.: All tuples retrieved for the query *comedy academy award*

in their semantics. For instance, consider the last tuple in Table 6.5. It states the fact that the movies `Police_Academy` and `The_Darwin_Awards` are both comedy movies. The rest of the tuples in Table 6.5 describe movies that have genre comedy and have won the Academy Award. Recall our earlier observation that the user keyword query is a representation of an implicit structured triple-pattern query. Thus, in order to provide an effective ranking, the system must infer what is the most likely structured query the user has in mind and then must rank the tuples based on how well they match this implicit structured query. Our ranking model, described in Section 6.3, does this by combining the structure and the contents of the triples in the ranking function. Before we explain the details of our ranking model, we first provide the algorithm we use to retrieve a set of result tuples given a keyword query.

6.2. Retrieval Algorithm

As mentioned in the previous section, the results to a given keyword query are the set of tuples consisting of joined triples matching the query keywords. All the result tuples should satisfy the following two properties:

1. They should be unique and maximal. That is, each result tuple should not be a subset of any other result tuple.

2. They should contain triples matching different sets of keywords. That is, no triples in the same tuple would match the exact same set of keywords. If two triples match the same set of keywords, they are parts of two different possible results to the user query, and should be considered as parts of two separate tuples.

Our retrieval algorithm starts by retrieving the lists of all triples matching the query keywords. That is, given a query $q = \{q_1, q_2, ..., q_m\}$ where q_i is a single keyword, we retrieve m lists $\{L_1, L_2, ..., L_m\}$ where L_i is the list of all triples that match the keyword q_i (see Table 6.2, Table 6.3 and Table 6.4). We then join the triples from these lists to retrieve result tuples that satisfy the aforementioned properties as follows. Let the set of all unique triples in all the lists be E. This set E can be viewed as a disconnected graph which we refer to as the *query graph* . Recall that each triple can be viewed as an edge where its subject and object are nodes. Analogously, a result tuple can be viewed as a subgraph of the query graph. Throughout this section, we will be interchangeably using the terms edge and triple to denote a triple, and the terms tuple and subgraph to denote a result tuple.

We adapt the backtracking algorithm for network-motif detection in [86] to retrieve subgraphs representing result tuples from the query graph. The modified algorithm utilizes adjacency lists for edges. Given an edge t_i from list L_i, its adjacency list $A(t_i)$ would contain all neighbor edges t_j from all other lists L_j. Two edges are considered neighbors if they share a common node. That is, given an edge $t_i = (s_i, p_i, o_i)$, an edge $t_j = (s_j, p_j, o_j)$ would be a neighbor of t_i if $s_i = s_j$, $s_i = o_j$, $o_i = s_j$ or $o_i = o_j$. In order to retrieve only unique subgraphs, we associate with each edge t_i an id and we only add a neighbor to the adjacency list of t_i if its id is greater than that of t_i. Also, to ensure that we do not consider joining triples that match the same set of keywords, we only add a neighbor t_j to the adjacency list $A(t_i)$ of triple t_i if and only if $t_i \notin L_j$ and $t_j \notin L_i$. Finally, we loop over all edges and generate all unique subgraphs using the two algorithms Algorithm 5 and Algorithm 6.

Algorithm 5 RETRIEVESUBGRAPHS(E)

```
1: for each edge $t \in E$ do
2:     EXTENDSUBGRAPH($\{t\}$, $A(t)$)
3: end for
```

Algorithm 6 EXTENDSUBGRAPH(G, X)

```
1: while $X \neq \phi$ do
2:     Remove an arbitrary chosen edge $t$ from $X$
3:     if ($L(\{t\}) \not\subseteq L(G)$ and $L(G) \not\subseteq L(\{t\})$) then
4:         $X' \leftarrow X \cup \{t' \in \mathrm{NEIGHBORS}(t, G)\}$
5:         EXTENDSUBGRAPH($G \cup \{t\}$, $X'$)
6:     end if
7: end while
8: if (MAXIMAL($G$)) then
9:     print $G$
10:    return
11: end if
```

Algorithm 5 loops over all the edges in the query graph and for each edge t, it extracts its neighbors from its adjacency list $A(t)$. Algorithm 6 takes as an input a subgraph and a list of neighbors and recursively tries to add edges to this subgraph. The condition in line 3 of Algorithm 6 ensures that only edges that belong to at least one different list other than the lists the edges of the current subgraph G belong to are considered. This ensures that we construct only subgraphs whose edges match different sets of keywords. The function $L(G)$ returns the set of lists the edges of subgraph G belong to. Once an edge is added to the current subgraph, we also add its neighbors that are not neighbors of edges in G to the current list of neighbors and continue. The function $\mathrm{NEIGHBORS}(t, G)$ retrieves all neighbors of an edge t that are not neighbors to edges in G. Finally, in order to ensure that we retrieve only maximal subgraphs, the algorithm checks if the current subgraph is a subset of another retrieved subgraph before returning it. Each subgraph retrieved by our algorithm represents a result tuple of the given keyword query.

The running time of our retrieval algorithm depends on the number of query keywords m as well as the size of the lists $L_1, L_2, .., L_m$. In the worst case, we would need to consider joining each triple from one list with all triples from all other lists. That is, we assume that each triple in one list is a neighbor to all triples from all other lists. In this case, the running time of our algorithm would be $O(|L_1|.|L_2|.....|L_m|)$. Given that $|L_i|$ is at most n where n is the number of triples in the whole knowledge base, the running time of our algorithm is $O(n^m)$ in the worst case.

However, in practice, our algorithm runs much faster. First, our algorithm depends on the sizes of the lists L_i which are typically much less than n unless the query keywords occur in the documents of all the triples in our knowledge base. Moreover, it is very likely that each triple would need to be joined with only a small subset of triples from the other lists (i.e., each triple would have a small number of neighbors). These two properties make the runtime of our algorithm much smaller than the worst case bound in practice.

6.3. Ranking Model

In the previous section, we explained how a set of tuples of joined triples matching a given keyword query can be retrieved. We now explain how we rank these tuples. Our ranking model is based on statistical language-models [69] and works as follows. We assume there exists a language model for every triple in our knowledge base which is defined as follows.

Definition 6.3 : Triple Language Model
The language model of triple t_i *is a probability distribution over the set* V *of all terms that appear in all the documents of all the triples in the knowledge base. The language model of triple* t_i *has* $|V|$ *parameters* $P(w|t_i)$ *which is the probability of term* w *in the language model of triple* t_i.

Furthermore, we assume that there exists a language model for any tuple of triples which is defined as follows.

Definition 6.4 :Tuple Language Model
The language model of tuple $T = (t_1, t_2, ..., t_n)$ *is a probability distribution over the set* V *of all terms that appear in all the documents of all the triples in the knowledge base. The language model of tuple* T *has* $|V|$ *parameters* $P(w|T)$ *which is the probability of term* w *in the language model of tuple* T.

Now, given a query $Q = \{q_1, q_2, ..., q_m\}$ where q_i is a single term and a tuple $T = \{t_1, t_2, ..., t_n\}$ where t_j is a triple, we rank the tuple T based on the *query likelihood* of tuple T. Assuming independence between the query terms, the query likelihood of tuple T is computed as follows:

$$P(Q|T) = \prod_{i=1}^{m} P(q_i|T) \tag{6.1}$$

where $P(q_i|T)$ is the probability of the term q_i in the language model of tuple T.

Estimating the Language Models. The probability $P(q_i|T)$ of term q_i in the language model of tuple T is computed as a weighted sum of the following n probabilities:

$$P(q_i|T) = \sum_{j=1}^{n} \frac{1}{n} P(q_i|t_j) \tag{6.2}$$

That is, the probability of a term q_i in the tuple language model is the average of its probability in the language models of the triples constituting the tuple. Note that more than one triple in the tuple T can match the same keyword q_i and thus averaging over all the triples is a natural choice. The language model of a triple t_j can then be directly estimated using the document of the triple. Recall from Section 6.1 that each triple t_j is associated with a document D_j which is composed of all the terms associated with the triple. However, this approach completely ignores the structure of the triples and treats every triple as a bag-of-words. For instance, consider the query *comedy academy award* to find comedies that have won the Academy Award. Now, consider the tuple T_1

```
Innerspace  hasGenre     Comedy
Innerspace  hasWonPrize  Academy_Award
```

and the tuple T_2

```
The_Darwin_Awards  type  Comedy_films
Police_Academy     type  Comedy_films
```

Given that the intended information need of the query is to find comedies that have won the Academy Award, we should rank the tuple T_1 higher.

In our ranking model, we try to take into consideration the structure of the triples as an additional evidence of how well they match the structured-information need intended by the keyword query. This is motivated by our earlier remark that we built our retrieval model on: a user keyword query is a representation of an implicit structured triple-pattern query. Considering our example query *comedy academy award*, the term *comedy* most likely refers to the triple pattern

```
?x  hasGenre  Comedy
```

given the fact that the term *comedy* appears more often in the documents of triples with predicate `hasGenre` and object `Comedy`. Similarly, the terms *academy* and *award* would likely refer to the pattern

```
?x  hasWonPrize  Academy_Award
```

Thus, it would be desirable to assign higher probability mass to triples that instantiate these patterns (i.e., triples with predicate `hasGenre` for the keyword *comedy* and *hasWonPrize* for the keywords *academy* and *award*).

To this end, we set the probability of a term q_i in the language model of triple t_j, which we denoted by $P(q_i|t_j)$ in Equation 6.2, to $P(q_i|D_j, r_j)$. That is, the probability of a term in a triple's language model does not only depend on the document of the triple t_j, but also on its predicate which we denote by r_j. This is very intuitive given the fact that predicates are typically the most representative aspect of the semantic of the triples. Applying Bayes' rule, we have:

$$P(q_i|D_j, r_j) = \frac{P(q_i|D_j)P(r_j|q_i, D_j)}{P(r_j|D_j)} \tag{6.3}$$

Furthermore, we set $P(r_j|q_i, D_j)$ as a linear combination of the following two components [74, 51, 50]:

$$P(r_j|q_i, D_j) = \beta P(r_j|q_i) + (1 - \beta)P(r_j|D_j) \tag{6.4}$$

The first component in Equation 6.4 is the probability that the predicate r_j is intended by the term q_i whereas the second component is the probability that the predicate of triple t_j is r_j. The latter can be set to the extraction accuracy of triple t_j. Since this value is not generally present in RDF knowledge bases, we assume that we are always fully confident in the extraction quality of any triple, and set $P(r_j|D_j)$ to 1. The parameter β is a weighting parameter that controls the effect of each component on the ranking and can be set using training queries.

Substituting Equation 6.4 in Equation 6.3 and simplifying, we have:

$$P(q_i|D_j, r_j) = \beta P(q_i|D_j)P(r_j|q_i) + (1-\beta)P(q_i|D_j) \tag{6.5}$$

where $P(q_i|D_j)$ is the probability of generating the term q_i from the document D_j which can be estimated using a maximum-likelihood estimator after smoothing with a background (collection) probability as follows:

$$P(q_i|D_j) = \alpha\frac{c(q_i; D_j)}{|D_j|} + (1-\alpha)\frac{c(q_i; Col)}{|Col|} \tag{6.6}$$

where $c(w; D_j)$ is the frequency of term w in document D_j, $|D_j|$ is the length of document D_j (i.e., the sum of the term frequencies of all terms in D_j), Col is the whole collection constructed by concatenating all the documents of all the triples in the knowledge base, $c(w; Col)$ is the frequency of term w in the whole collection and $|Col|$ is the length of the whole collection.

Finally, the parameter α is a smoothing parameter and is set according to Dirichlet prior smoothing as follows:

$$\alpha = \frac{|D_j|}{|D_j| + \mu} \tag{6.7}$$

where μ is the average document length in the whole collection.

The only remaining component to estimate in Equation 6.5 is the probability that the predicate r_j is intended by the query term q_i. In order to estimate this probability, we first construct a document R_j for each predicate r_j in the knowledge base concatenating the documents of all the triples with such predicate. For instance, given the predicate `hasGenre`, we construct a document which is a concatenation of all the documents of all triples with predicate `hasGenre`. Once we have constructed a document R_j for each predicate r_j, we set the probability that predicate r_j is intended by term q_i (i.e., $P(r_j|q_i)$ in Equation 6.5) to $P(R_j|q_i)$ (i.e., the probability of relevance of the document R_j to the term q_i). Applying Bayes' rule, the probability $P(R_j|q_i)$ can be estimated as follows:

$$P(R_j|q_i) = \frac{P(q_i|R_j)P(R_j)}{P(q_i)} = \frac{P(q_i|R_j)P(R_j)}{\sum_k P(q_i|R_k)P(R_k)} \tag{6.8}$$

where $P(w|R_j)$ is the probability of generating the term w given the document R_j which is estimated using a maximum-likelihood estimator as in Equation 6.6 and $P(R_j)$ is the prior probability of the document R_j being relevant to any term, which we set uniformly. For example, using the above technique, the probability $P(hasGenre|comedy)$ would be higher than $P(actedIn|comedy)$ given the fact that the keyword *comedy* appears much more often in the documents of triples that have predicate `hasGenre` than those that have predicate `actedIn`.

To summarize, our ranking model weights the probability of a query term in the triple language model by the probability of the triple's predicate being intended by the query term. This seems to be very intuitive given that the distribution of terms in the knowledge base over all predicates gives clues on what are the most-likely predicates intended by the user keyword query.

6.4. Related Work

The work on keyword search over structured data can be classified into two classes. The first class aims at mapping the keyword query into one or more structured queries. For instance, the authors in [82] assume that the user keyword-query is an implicit representation of a structured triple-pattern query. They try to infer such triple-pattern query using the RDF knowledge base and retrieve the top-k most relevant triple-pattern queries. They then provide the user with the retrieved queries and let her choose the most appropriate triple-pattern query to be evaluated. Their approach involves user interaction, and in addition suffers from a loss-of-information phenomenon since typically k is set to a small number. One way to overcome the problem of engaging the user in the inference process is to directly evaluate the top-k inferred queries. Again this has the problem of information loss and in addition is typically inefficient since each one of these queries would have to be evaluated.

The work on query inference from a user's natural language question in [58] is also closely related. It utilizes natural-language processing tools and try to parse a user's question in order to infer the most-likely triple-pattern query. Their technique however relies heavily on the quality of the parsing process and it also suffers from the information-loss problem highlighted above.

The second class of work on keyword search over structured data overcomes the aforementioned issues by directly retrieving the results of the keyword query. The work on keyword search over XML data for instance falls into this category. XKSearch [88] returns a set of nodes that contain the query keywords either in their labels or in the labels of their descendant nodes and have no descendant node that also contains all keywords. Similarly, XRank [34] returns the set of elements that contain at least one occurrence of all the query keywords, after excluding the occurrences of the keywords in sub-elements that already contain all the query keywords. However, all these techniques assume a tree-structure and thus can not be directly applied to graph-structured data such as RDF data.

Also, closely related to our work is the language-modeling approach for keyword search over XML data proposed in [49]. The authors assume that a keyword query has an implicit mapping of each keyword into XML element(s). Their ranking is based on the hierarchal language models proposed in [67] and they utilize the distribution of terms in the elements of the XML collection to weight the different components of the language models. However, the setting of XML data is quite different from that of RDF since in XML the retrieval unit is an XML document (or a subtree). In an RDF setting, we are interested in ranking tuples of triples that match the user's query. These tuples are not present in advance and are computed on the fly during retrieval time, and thus most of the prior work on XML IR would not apply.

Keyword search over graphs which returns a ranked list of Steiner trees [41, 46, 39, 32] (the exception is [55] which returns graphs) deals with the latter problem of constructing the results at query time as opposed to having an indexed set of results that needs to be matched and ranked at query time. However, the result ranking in each of the above is based on the structure of the results [41, 47] (usually based on aggregating the number or weights of nodes and edges), or on a combination of these properties with content-based measures such as tf-idf [14, 39, 55] or language models [66].

For instance, the BANKS system [41] enables keyword search on graph databases. Given a keyword query, an answer is a subgraph connecting some set of nodes that "cover" the keywords (i.e., match the query keywords). The relevance of an answer is determined based on a combination of edge weights and node weights in the answer graph. The importance of an edge depends upon the type of the edge, i.e., its relationship. Node weights on the other hand represent the static authority or importance of nodes and are set as a function of the in-degree of the node. However, BANKS completely ignores the content during ranking, and thus does not make use of the content of the triples as an additional evidence of relevance to the query.

A closely related work that combines structure and content for ranking is the language-model-based ranking model in [66] for ranking objects (resources in an RDF setting). The model assumes that each resource is associated with a set of records extracted from Web sources. In turn, each record is associated with a "document". The relevance of each such "document" (and correspondingly, the resource associated with it) to a keyword query is estimated using language models. This model however assumes that the retrieval unit is resources only, while our ranking model goes beyond this to treat triples in a holistic manner by taking into account the relationships between the resources. In addition, it assumes the presence of a document associated with each Web Object or resource, something that we lack in the case of RDF data in general.

The Semantic Search Challenge provided a benchmark for keyword queries over RDF data, however the judgments were made over resources built by assembling all the triples that shared the same subject. The best performing approach [7] ranked the resources using a combination of the BM25F scoring function and additional hand-crafted information about some predicates, properties and sites. In contrast, we retrieve the set of tuples that match the query keywords and rank them. We believe that retrieving tuples of triples rather than just resources provide more concise answers to the user information need.

6.5. Experimental Evaluation

6.5.1. Setup

We evaluated our retrieval model using a comprehensive user-study over two RDF datasets. The first dataset was derived from the LibaryThing community, which is an online catalog about books. The second dataset was derived from the Internet Movie Database (IMDB). The data from both sources was automatically parsed and converted into RDF triples.

Recall that our retrieval model assumes that each triple t_i is associated with a document D_i. The documents were constructed by using keywords derived from the subjects and objects of the triples, and representative words for the predicates. For example, the predicate `isMarriedTo` was represented using the terms {*marry, wife, husband, spouse, etc*}. This was done manually since we did not have that many predicates in our datasets, but for bigger datasets, the rep-

#entities	Example entity types	#triples	Example relations	#unique terms
LibraryThing Dataset				
48,000	`book, author user, tag`	700,000	`wrote, hasFriend, hasTag, type`	21,821
IMDB Dataset				
59,000	`movie, actor director, producer, country , language`	600,000	`actedIn, directed, won, isMarriedTo, produced, hasGenre`	80,584

Table 6.6.: Overview of the datasets

resentations of predicates can be generated automatically using an external dictionary, or by utilizing the textual extraction-patterns in case the triples were extracted using some IE technique from free-text [79]. Once each triple was associated with a set of keywords, we stemmed all the keywords using the Stanford stemmer, removed stop words and created an inverted index over the triples. Table 6.6 gives an overview of the datasets.

To evaluate our approach, we used a subset of the query benchmark in [23]. The benchmark there contains a set of structured triple-pattern queries, possibly augmented with keywords, along with their descriptions. We extracted *30* queries from there, 15 for each dataset and represented each query using a set of keywords. We opted for 30 queries only since we pooled 50 results per each query, and gathered relevance assessment for each result using at least 4 different human judges. Overall, we had about *15,000* unique relevance assessments for the 30 queries. All the evaluation queries are listed in Appendix D. A subset of the evaluation queries used is shown in Table 6.7. We used these queries to compare the performance of four retrieval models, which we describe in detail next.

6.5.2. Retrieval Models

We compared our ranking model, which we refer to as the *Structured LM* approach, to three competitors: 1) a baseline language-modeling approach (Baseline LM), 2) the Web Object Retrieval Model (WOR) [66] and 3) the BANKS

Information need	**Query**
LibraryThing	
Historians who wrote memoir books	historian memoir book
The author of a classic fantasy funny book	classic fantasy funny author
Authors of non-fiction books that won the Pulitzer prize	author non-fiction pulitzer
A crime fiction that has was tagged as favorite by the users	crime fiction favorite
Children's writers who wrote books about were-wolves	children writer were-wolves
IMDB	
Movies with genre Musical that were produced in Italy	musical italy
Actors from New York City that have won the Academy Award for Best Actor	new york academy award best actor
Movies with genre War in which Anthony Quinn acted	anthony quinn war
Movies with genre Comedy that have won the Academy Award	comedy academy award
Movies that Mel Gibson directed	mel gibson director

Table 6.7.: A subset of the evaluation queries

system [41]. We chose these 3 competitors since they represent the family of approaches applicable to our setting, namely: keyword search over structured data. The rest of the approaches sketched in Section 6.4 do not directly apply to our setting and thus were omitted from our evaluation.

Structured LM Approach. The Structured LM approach ranks a set of tuples of triples retrieved using the retrieval algorithm in Section 6.2. It takes into consideration the structure of the triples which is represented by the predicates of the triples as described in Section 6.3. The Structured LM approach weights the probability of a query term in the language model of each triple by the probability of the triple's predicate being intended by the query term. This model involves the single parameter β which must be learnt (see Equation 6.5), and we explain how to do this in Subsection 6.5.4.

Baseline LM Approach. Similar to the Structured LM approach, the Baseline LM approach also ranks a set of tuples of triples that are retrieved using the retrieval algorithm described in Section 6.2. It also uses the query likelihood of the tuples to rank them. However, this approach completely ignores the structure of the triples and treats all the triples as bags-of-words. The Baseline LM approach is a special case of our Structured LM approach and can be achieved by setting the value of the parameter β in Equation 6.5 to 0.

Web Object Retrieval. The Web Object Retrieval model proposed by Nie et al. [66] is a language-model-based approach for ranking objects, or *resources* in an RDF setting. The model assumes that each resource is associated with a set of records extracted from Web sources. In turn, each record is associated with a "document". The relevance of each such "document" (and correspondingly, the resource associated with it) to a keyword query is estimated using language models.

We adapted the WOR model to work on RDF data as follows. We treated triples as records and for a given resource X, we created a language model for X using *all* its triples $\{t_1, t_2, ..., t_n\}$. Given a keyword query $Q = \{q_1, q_2, ..., q_m\}$, we then ranked the resources according to their probabilities of generating the query which is computed as follows:

$$P(Q|X) = \prod_{i=1}^{m} \sum_{j=1}^{n} \frac{1}{n} P(q_i|D_j) \tag{6.9}$$

where $P(q_i|D_j)$ is the probability of generating the term q_i given triple t_j's document which was estimated using a maximum-likelihood estimator as described in Equation 6.6 in Section 6.3.

BANKS. The BANKS system enables keyword search on graph databases. Given a keyword query, an answer is a subgraph connecting some set of nodes that "cover" the keywords (i.e., match the query keywords). The relevance of an answer is determined based on a combination of edge weights and node weights in the answer graph. The importance of an edge depends upon the type of the edge, i.e., its relationship. Node weights on the other hand represent the static authority or importance of nodes and are set as a function of the in-degrees of the nodes.

This directly applies to our setting. Given a keyword query, we retrieved all subgraphs that matched the query using the technique described in [41]. We then ranked the subgraphs based on a combination of edge weights and node weights as proposed in their model. That is, the score of a subgraph $G = \{t_1, t_2, ..., t_n\}$ with nodes $N = \{n_1, n_2, ..., n_k\}$ is defined as follows:

$$\mathrm{score}(G) = \lambda \sum_{i=1}^{n} \mathrm{score}(t_i) + (1 - \lambda) \sum_{i=1}^{k} \frac{1}{k} \mathrm{score}(n_i) \tag{6.10}$$

where $\mathrm{score}(t_i)$ is the score of triple or edge t_i which is computed using the probability that the predicate of t is intended by any of the query keywords as is done in the Structured LM approach. Note that in case a triple matches more than one keyword, it will be counted as many times as the number of keywords it matches. The $\mathrm{score}(n_i)$ on the other hand is the score of node n_i which is set to the in-degree of the node n_i. We used log scaling for both scores as advised in [41]. Finally, the parameter λ controls the influence of both scores, and we explain how it is set when we discuss the evaluation results in Subsection 6.5.4.

6.5.3. Relevance Assessments and Metrics

For each evaluation query, we retrieved the top-50 results retrieved using each one of the four models described above. We then pooled all the results together and presented the set of all unique results from the pool to 13 human judges in no particular order, along with the query description. The judges were all computer-scientists in two different research institutes. For the case of results retrieved using WOR, we presented the resource label as a result and provided the judges with a link to the Wikipedia article for that resource (in case there was one) in order to help them decide whether a result is relevant or not. For the rest of the results, we just presented the result tuples for judgment.

We asked the judges to assess the results on a 4-point scale: 3 corresponding to results that completely match the information need as given by the query description, 2 corresponding to results that do not completely match the information need but are still highly related to it, 1 corresponding to results that do not really match the information need of the query, but the results still make sense and add valuable information to the user and finally 0 corresponding to trivial, or nonsense results. Each result was evaluated by four different judges. The levels of agreement between the judges as measured by the Kappa coefficient were *0.449* for the LibraryThing dataset and *0.542* for the IMDB dataset. We also computed the agreement for relevant and irrelevant results only (i.e., assuming that levels 3,2,1 are relevant and 0 is irrelevant). We obtained a Kappa coefficient of *0.397* for LibraryThing and *0.671* for IMDB which are in line with the numbers reported for standard TREC evaluation campaigns. For instance, the TREC legal track for 2006 reports a Kappa value of 0.49 on 40 queries, the opinion detection task in 2009 reports a Kappa value of 0.34, and the TREC 2004 Novelty track reports a value of 0.54 for sentence relevance.

To compare the 4 retrieval models, we used the Discounted Cumulative Gain (DCG) [45], which is is defined as follows:

$$DCG(i) = \begin{cases} G(1) & \text{if } i = 1 \\ DCG(i-1) + G(i)/\log(i) & \text{otherwise} \end{cases}$$

where *i* is the rank of the result within the result set, and *G(i)* is the relevance level of the result. We set $G(i)$ to a value between 0 and 3 depending on the judge's assessment. For each result, we averaged the ratings given by all judges and used this as the relevance level for the result. Dividing the obtained DCG by the DCG of the ideal ranking we obtained a *Normalized DCG (NDCG)*. We report the NDCG values at rank positions or levels 20, 10 and 5.

6.5.4. Evaluation Results

We conducted three experiments: an overall evaluation using all evaluation queries, a training experiment to set the parameters of the models that involve ones, and a cross-validation to predict how well our parameter-learning approach would generalize.

Model	**NDCG @20**	**NDCG @10**	**NDCG @5**
Structured	**0.764**	**0.817**	**0.840**
BANKS	0.637	0.647	0.639
WOR	0.576	0.596	0.621
Baseline	0.397	0.368	0.351

Table 6.8.: Average NDCG values for both datasets

Model	**NDCG @20**	**NDCG @10**	**NDCG @5**
LibraryThing			
Structured	**0.861**	**0.880**	**0.889**
BANKS	0.762	0.734	0.710
WOR	0.621	0.624	0.623
Baseline	0.395	0.361	0.333
IMDB			
Structured	**0.667**	**0.754**	**0.791**
BANKS	0.513	0.560	0.569
WOR	0.530	0.567	0.618
Baseline	0.399	0.376	0.370

Table 6.9.: Average NDCG values with parameter learning

Overall Evaluation. In the first experiment, we report the average NDCG values over all *30* evaluation queries at levels 20, 10 and 5 using all four different models in Table 6.8. The values for the Structured LM approach and the BANKS system reported in the table are the ones achieved when the models parameters were set to their optimum value (i.e., $\beta = 0.9$ for the Structured LM approach and $\lambda = 1$ for BANKS).

As can be seen from Table 6.8, the Structured LM approach significantly outperforms ($p - value < 0.05$ with a one-tailed t-test) all other methods in terms of NDCG values at all levels. In the next experiment, we explain how to set the models' parameters using training queries.

Training Results. In the second experiment, we used one dataset for training and the other for testing. That is, the 15 queries for the IMDB dataset were used as a training set to learn the optimal parameter setting for the Structured LM approach and BANKS. The 15 queries for LibraryThing were then used to test the performance of the different methods. We repeated the same procedure using the LibraryThing queries for training and the IMDB queries for testing. The learning procedure was as follows. For the Structured LM approach, we computed the average NDCG at level *50* over the 15 training queries, setting the parameter β to a value between 0 and 1. We achieved the highest average $NDCG@50$ for both datasets when β was set to 0.9. For BANKS, we did the same thing using the same set of training queries and setting the parameter λ to a value between 0 and 1, and we achieved the highest average NDCG at level 50 when λ was set to 1. Table 6.9 shows the average NDCG values over the test queries at levels 20, 10 and 5.

Similar to the first experiment, the Structured LM approach significantly outperforms ($p-value < 0.05$ with a one-tailed t-test) all other methods in terms of NDCG values at all levels for both datasets. In order to test how well our training strategy generalizes, we performed a cross-validation experiment which we report next.

Cross-Validation Results. The third experiment was a cross-validation experiment to show how well the parameter learning procedure we described above generalizes over unseen datasets. We performed a leave-one-out cross validation, where 14 out of the 15 queries for each dataset were used as a training set to determine the the value of the parameter β, and then the left-out query was used for testing. We repeated the same process such that each evaluation query is used for validation once, and we averaged the NDCGs over all the validation queries. For BANKS, we also performed a cross-validation to validate the learning of its parameter λ, and again averaged the NDCGs over all the queries. For the IMDB dataset, the results were identical to those reported in Table 6.9 for all approaches, and for the LibraryThing dataset, the results were also the same as in the training experiment, except for a slight change in the case of the Structured LM approach (with NDCG values of *0.814*, *0.833* and *0.841* at levels 20,10 and 5, respectively). That is, similar to the results of the first two experiments, the Structured LM approach outperforms all other methods for both datasets.

Qualitative Results. In Table 6.10, we show the results to the query *anthony quinn war* over the IMDB dataset. The top-4 results returned by the four approaches are given and next to each result, the average relevance value given by the human judges is shown (Column Rel.). Recall that each result was given a relevance value between 0 and 3.

The results returned by the Structured LM approach were all about war movies that Anthony Quinn played a role in. On the other hand, the results returned by BANKS were also movies of genre War, but Anthony Quinn had nothing to do with neither the first nor the fourth movie. This happens because BANKS just relies on edge and node weights to rank the results, without taking into consideration the query keywords. Even when we set the edge weights to represent how well their predicates match the query keywords, BANKS would still favor certain types of edges , as in the case with our example where any subgraphs with an edge of type `hasGenre` were ranked higher.

The WOR on the other hand takes into consideration the query keywords, however it has the drawback of requiring additional result representation strategy. Just looking at the resource labels, it is hard to judge whether or not the resources are relevant to the query unless the user already knows the resources. For instance, the first result is a war movie directed by Anthony Quinn. On the other hand, the approaches that retrieve tuples of triples make use of the triples as a whole and provide the user with a means of interpreting the results.

Finally, the first result returned by the Baseline LM approach states that Anthony Quinn and Warly Ceriani are both actors. Note that the stemming tool we used stemmed the word *Warly* into war, and thus such tuple was retrieved as a result. Since the Baseline LM approach does not take into consideration the structure of the triples and how well they match the implicit structured query intended by the keyword one, such results as the tuple just mentioned can have high ranking as compared to their rank by the Structured LM approach.

6.6. Summary

RDF knowledge bases can be effectively searched using structured triple-pattern-based query languages, such as SPARQL. While such structured queries are very expressive and can represent advanced information needs very precisely, they are tailored for Search APIs rather than casual users. Users prefer searching using keyword queries. In this Chapter, we presented a retrieval model for keyword queries over RDF data. Our model retrieves tuples of triples matching the query keywords using a backtrack-searching algorithm. In addition, we rank the result tuples based on how well they match the given keyword query where the ranking is based on a novel structure-aware language-modeling approach. We have shown through a comprehensive user-study that our retrieval model outperforms well-known techniques for keyword search over structured data.

Q	**anthony quinn war**			
Rank	**Structured**			**Rel.**
1	`Back_to_Bataan`	`hasGenre`	`War`	3
	`Anthony_Quinn`	`actedIn`	`Back_to_Bataan`	
2	`Anthony_Quinn`	`actedIn`	`Lion_of_the_Desert`	3
	`Lion_of_the_Desert`	`hasGenre`	`War`	
3	`The_25th_Hour`	`type`	`World_War_II_films`	3
	`Anthony_Quinn`	`actedIn`	`The_25th_Hour`	
4	`Anthony_Quinn`	`actedIn`	`The_Guns_of_Navarone`	3
	`The_Guns_of_Navarone`	`type`	`World_War_II_films`	
Rank	**BANKS**			**Rel.**
1	`We_Dive_at_Dawn`	`hasGenre`	`War`	1
	`Anthony_Asquith`	`directed`	`We_Dive_at_Dawn`	
2	`Back_to_Bataan`	`hasGenre`	`War`	3
	`Anthony_Quinn`	`actedIn`	`Back_to_Bataan`	
3	`Anthony_Quinn`	`actedIn`	`Lion_of_the_Desert`	3
	`Lion_of_the_Desert`	`hasGenre`	`War`	
4	`Ice-Cold_in_Alex`	`hasGenre`	`War`	1
	`Anthony_Quayle`	`actedIn`	`Ice-Cold_in_Alex`	
Rank	**WOR**			**Rel.**
1	`The_Buccaneer`			2
2	`The_25th_Hour`			3
3	`The_Guns_of_Navarone`			3
4	`The_Secret_of_Santa_Vittoria`			3
Rank	**Baseline**			**Rel.**
1	`Warly_Ceriani`	`type`	`actor`	0
	`Anthony_Quinn`	`type`	`actor`	
2	`Anthony_Quinn`	`directed`	`The_Buccaneer`	2.25
	`The_Buccaneer`	`type`	`Napoleonic_Wars_films`	
3	`Back_to_Bataan`	`hasGenre`	`War`	3
	`Anthony_Quinn`	`actedIn`	`Back_to_Bataan`	
4	`Anthony_Quinn`	`actedIn`	`Lion_of_the_Desert`	3
	`Lion_of_the_Desert`	`hasGenre`	`War`	

Table 6.10.: The top-ranked results for the query *anthony quinn war*

Chapter 7.

Result Diversity

Diversifying the top-k ranked search results has been identified as an important aspect of any successful information retrieval system. Result diversity can play a big role in ensuring that the users get a broad view of the different aspects of the results matching their queries, and aims to ensure that, even if the query is ambiguous and the user's information need is not fully clear, the user can find at least one relevant query result in the top ranks. For queries over RDF knowledge bases, the notion of diversity is much less clear and largely unexplored. In this chapter, we address this problem, and define a notion of diversity in an RDF setting. We propose techniques to diversify the results of queries over RDF knowledge bases in order to cover all possible aspects of the query and ensure that the top-k ranked results are as diverse as possible.

7.1. Result Diversity for Queries over RDF Knowledge Bases

The results of queries over RDF knowledge bases tend to be homogeneous, making it difficult for users interested in less popular aspects to find relevant results. For example, when asking for movies directed by directors who won an Academy Award, it is preferable to represent the user with a list of movies that were directed by different directors, have different genres and so on. If we solely rely on a ranking model that takes into consideration only how relevant the results are to the query, it is possible that the top-k highest-ranked results would be dominated by a certain type of movies, for instance directed by the same director or with the same genre, etc.

R_1	James_Cameron	hasWonPrize	Academy_Award_for_Best_Director
	James_Cameron	directed	Titanic
R_2	James_Cameron	hasWonPrize	Academy_Award_for_Best_Director
	James_Cameron	directed	Avatar
R_3	Steven_Spielberg	hasWonPrize	Academy_Award_for_Best_Director
	Steven_Spielberg	directed	Schindler's_List
R_4	Steven_Spielberg	hasWonPrize	Academy_Award_for_Best_Director
	Steven_Spielberg	directed	Munich
R_5	Woody_Allen	hasWonPrize	Academy_Award_for_Best_Director
	Woody_Allen	directed	Annie_Hall
R_6	Woody_Allen	hasWonPrize	Academy_Award_for_Best_Director
	Woody_Allen	directed	Mighty_Aphrodite
R_7	Woody_Allen	hasWonPrize	Academy_Award_for_Best_Director
	Woody_Allen	directed	Vicky_Christina_Barcelona
R_8	Sam_Mendes	hasWonPrize	Academy_Award_for_Best_Director
	Sam_Mendes	directed	American_Beauty
R_9	Clint_Eastwood	hasWonPrize	Academy_Award_for_Best_Director
	Clint_Eastwood	directed	Million_Dollar_Baby
R_10	Clint_Eastwood	hasWonPrize	Academy_Award_for_Best_Director
	Clint_Eastwood	directed	Mystic_River

Table 7.1.: Results as tuples of triples for the example query "directors who have won an Academy Award and their movies"

Table 7.1 shows the top-10 results retrieved for our example query using some popularity-based criteria. The result set is dominated by well-known directors such as James Cameron and Steven Spielberg who each have more than one movie listed. In order to increase the diversity of the top-10 results, we should present more directors that have won the Academy Award and one of their movies rather than showing many movies by the same director. Moreover, we could also make sure that the top ranked results involve movies with different genres, different types of directors, etc. Finally, we could also present movies that differ in their plots, keywords, user's comments and ratings, etc.

To achieve the above goals, we need to utilize a *diversity-aware* ranking model that trades off the relevance of the top-k highest-ranked results and their diversity. A diversity-aware ranking model should ideally try to produce an ordering or a permutation of the query results such that the top-k results are most relevant to the query and at the same time as diverse from each other as possible. This can be cast into an optimization problem where the objective is to produce an ordering that would maximize both the relevance of the top-k results and their diversity. The objective function for such an optimization problem is very hard to both quantify and solve and thus most diversity approaches [33] try to solve a simpler closely-related problem known as the top-k set selection problem. The top-k set selection problem can be formulated as follows.

Definition 7.1 *:* **Top-k Set Selection**
Let Q be a query and U be its result set. Furthermore, let REL be a function that measures the relevance of a subset of results $S \subseteq U$ with respect to Q and let DIV be a function that measures the diversity of a subset of results $S \subseteq U$. Finally, let f be a function that combines both relevance and diversity. The top-k set selection problem can be solved by finding:

$$S^* = \underset{S \subseteq U}{\operatorname{argmax}} \ f(Q, S, REL, DIV)$$

such that $|S^*| = k$

The objective function $f(Q, S, REL, DIV)$ is clearly underspecified and in order to the solve this optimization problem, one must clearly specify both the relevance function REL and diversity function DIV and how to combine them. Gollapudi and Sharma [33] proposed a set of axioms to guide the choice of the objective function $f(Q, S, REL, DIV)$ and they showed that for most natural choices of the relevance and diversity functions, and the combination strategies between them, the above optimization problem is NP-hard. For instance, one such choice of the objective function is the following:

$$f(Q, S, REL, DIV) = (k-1) \sum_{R \in S} rel(R, Q) + 2\lambda \sum_{R, R' \in S} d(S, S') \tag{7.1}$$

where $rel(R, Q)$ is a (positive) score that indicates how relevant result R is with respect to query Q (the higher this score is, the more relevant R is to Q) and $d(R, R')$ is a discriminative and symmetric distance measure between two results R and R', and λ is a scaling parameter.

The above objective function clearly trades off both relevance of results in the top-k set with their diversity (as measured by their average distance). Solving such objective function is again NP-hard, however there exists known approximation algorithms to solve the problem that mostly rely on greedy heuristics [33].

In the next section, we follow the same approach to obtain a top-k set of relevant and diverse results for queries over RDF knowledge bases. In particular, we optimize the above bi-criteria objective function using a greedy algorithm that uses the *Maximal Marginal Relevance (MMR)* [10] to select the top-k set. The advantage of an MMR-based algorithm is that it does not only select the top-k set but also sorts it according to the *marginal relevance* of the results. In addition, we can easily adapt the Maximal Marginal Relevance to the language-modeling result-ranking framework that we assumed for all result-ranking approaches we developed throughout this thesis

7.2. Maximal Marginal Relevance

Carbonell and Goldstein introduced the Maximal Marginal Relevance (MMR) method [10] which they use to re-rank a set of pre-retrieved documents U given a query Q. In order to do so, they combine relevance and diversity using a linear interpolation which they refer to as the *marginal relevance* and then they re-rank the documents incrementally by picking the next document that is not yet selected that has the maximal marginal relevance with respect to the query and the already selected documents.

We now show how we can adapt this diversity-aware ranking model to our setting. Throughout this thesis, we assumed the results to queries, whether triple-pattern queries or keyword queries, to be tuples of triples relevant to the query. For example, Table 7.1 shows the top-10 results for the query "directors who have won the Academy Award and their movies" as tuples of joined triples where the results are ranked based on some popularity measure. We start by defining the marginal relevance in this setting and then provide an algorithm that can be used to re-rank a set of results for a given query based on the marginal relevance that trades off both the results' relevance and their divergence from higher-ranked results.

Definition 7.2 : Marginal Relevance
Given a query Q*, a set of tuples of triples* U *and a subset* $S \subset U$*, the marginal relevance of a tuple* $T \in U \setminus S$ *is equal to:* $\mathrm{MR}(T, Q, S) = \lambda \mathrm{rel}(T, Q) + (1 - \lambda) \min_{T' \in S} \mathrm{div}(T, T')$ *where* $\mathrm{rel}(T, Q)$ *is a measure of how relevant* T *is to* Q*,* $\mathrm{div}(T, T')$ *is a measure of how divergent* T *is from* T' *and* λ *is a controlling parameter.*

The idea behind the marginal relevance metric is very intuitive. Given a query Q and a set of already selected tuples S, the marginal relevance of a tuple T is a measure of how much do we gain in terms of both relevance and diversity by adding the tuple T to the selected set S. Recall, that our objective is to find a set $S \subseteq U$ that is most relevant to the query and most diverse. To measure how much the result tuple T would contribute to the relevance aspect of S, it is straight forward and we can use the tuple's relevance to Q. On the other hand, to measure how much tuple T would contribute to the diversity of S, it is more involved. The most natural thing to do is to compare T with all the tuples $T' \in S$ and compute a similarity (or rather dissimilarity) between T and every other tuple T' and then aggregate these similarities over all the tuples in S. We do exactly this by assuming there is a divergence function that can measure how tuple T diverges from any other tuple T' (i.e., how different T is from T') and then we use the minimum of the divergences of T from all the tuples $T' \in S$ as a measure of the overall contribution of tuple T to the diversity of set S if it were to be added to it. Note that using the minimum is very intuitive as it relies on the closest or most similar tuple $T' \in S$ as an estimate of how adding T to S would affect its diversity. Thus, by maximizing this minimum over a set of tuples $T \notin S$, we can find the tuple that when added to S would render it most diverse as compared to any other tuple. Alternatively, we can also use the average divergence between T and all tuples $T' \in S$ as a measure of the contribution of tuple T to the diversity of set S.

Given all these considerations, we set the relevance $\mathrm{rel}(T, Q)$ to the score of the tuple T which we obtain using any of the ranking models we developed for ranking results to queries over RDF data (see Chapter 3 and Chapter 6). We only assume here that the ranking model would rank the tuples descendingly based on their scores. In case, the ranking is based on ascending order of scores (for instance using Kullback-Leibler divergence), the marginal relevance definition would have to be adapted accordingly, which is straight forward to do.

To measure the divergence of two tuples $div(T, T')$, a natural choice is to use Kullback-Leibler divergence between a language model for tuple T and a language model for tuple T' which we denote by $KL(T||T')$. Note that while the Kullback-Leibler divergence is an *asymmetric* measure, this is actually not a problem. In fact, it is a desired property since we are interested in determining how divergent tuple T is from a reference tuple T' not the other way around. We point out that the same reasoning have been also deployed in other settings (for instance, document retrieval in [90]).

Finally, we explain how the marginal relevance can be used to provide a diverse-aware ranking of results given a query Q. Let U be the set of ranked results using any regular ranking model (i.e., that depends only on relevance without taking into consideration diversity). The algorithm to re-rank the results works as follows:

Maximal Marginal Relevance Re-Ranking Algorithm

1. Initialize our top-k set S with the highest ranked tuple $T \in U$

2. Iterate over all the tuples $T \in U \setminus S$, and pick the tuple T^* with the maximum marginal relevance $MR(T^*, Q, S)$. That is,

$$T^* = \underset{T \in U \setminus S}{\operatorname{argmax}} \ [\lambda rel(T, Q) + (1 - \lambda) \min_{T' \in S} div(T, T')]$$

3. Add T^* to S

4. If $|S| = k$ or $S = U$ return S otherwise repeat steps 2, 3 and 4

Now, the final step in our diversity-aware ranking model is to define how we compute the divergence $div(T, T')$ of a tuple T from another tuple T'. Again, we assume there exists a language model for each tuple and then we set $div(T, T')$ to $KL(T||T')$ where $KL(T||T')$ is the Kullback-Leibler divergence between the two language models. We next explain a set of different approaches to construct these language models that vary in the amount and level of diversity they provide.

7.2.1. Resource-based Diversity

In this method, the goal is to diversify the different resources (i.e., entities and relations) that appear in the result tuples. This ensures that no one resource will dominate the result set. Consider our example query asking for directors that have won the Academy Award and their movies. Table 7.1 shows the top-10 tuples retrieved for the query using some relevance-based criteria. The result set is dominated by well-known directors such as James Cameron and Steven Spielberg who each have more than one movie listed. In order to increase the diversity of the top-10 results, we should present more directors that have won the Academy Award for Best Director and one of their movies rather than showing many movies by the same director. In order to do so, we define a language model for each tuple as follows.

Definition 7.3 *: Resource-based Language Model*
The resource-based language model of tuple T *is a probability distribution over all resources in the knowledge base* KB.

The parameters of the tuple language model are estimated using a maximum likelihood estimator as follows:

$$P(w|T) = \alpha\frac{c(w;T)}{|T|} + (1-\alpha)\frac{c(w;Col)}{|Col|} \tag{7.2}$$

where w is a resource, Col is the set of all resources in the knowledge base, $c(w;T)$ and $c(w;col)$ are the number of times resource w occurs in R and Col respectively, and $|T|$ and $|Col|$ are the number of times all resources occur in T and Col, respectively.

7.2.2. Knowledge-Base-based Diversity

In this model, we try to go one level deeper and make sure we diversify the results not only in terms of their resources, but also take into consideration additional knowledge about them from the underlying knowledge base. For the earlier example query, not only do we want to diversify the results ensuring that no one director would dominate the result set, but we also want to make sure that the top ranked results involve movies with different production locations, genres, different types of directors, etc.

In order to achieve this goal, we define for each resource, whether an entity or a relationship, a language model which takes into consideration the resource's triples in the knowledge base in order to more accurately identify the nature of the resource. For example, consider the resource `Woody_Allen`. This resource is part of triples with relationships `actedIn` and `directed` hinting that such resource is particularly a director/actor. This gives additional evidence beyond just the resource identifier used in the first method.

The resource language model is defined as follows.

Definition 7.4 : Resource Language Model
The language model of a resource X is a probability distribution over ungirams of resources and bigrams of resource pairs.

The parameters of the language models of resources are estimated from the knowledge base using the triples in which the resource participates. This is done by associating each resource with a bag of unigrams and a bag of bigrams which are then used to estimate the probabilities of the unigrams and bigrams in the resource's language model using a smoothed maximum-likelihood estimator. For example, consider the resource `Woody_Allen`, and assume that the triples in which `Woody_Allen` appears as either a subject or an object are

```
Woody_Allen        directed    Manhattan
Woody_Allen        directed    Match_Point
Woody_Allen        actedIn     Scoop
Woody_Allen        type        American_Director
Federico_Fellini   influences  Woody_Allen
```

`Woody_Allen` would then be associated with the unigram bag {`Manhattan`, `Match_Point`, `Scoop`, `American_Director`, `Federico_Fellini`}. Similarly, the bigram bag of `Woody_Allen` would be {(`directed`, `Manhattan`), (`directed`, `Match_Point`), (`actedIn`, `Scoop`), (`type`, `American_Director`), (`Federico_Fellini`, `influences`)}.

The method we use to estimate the parameters of the language models of the resources from their unigram and bigram bags is described in more details in Chapter 4.

Once we have a language model for every resource in the knowledge base, we can use these language models to achieve an additional level of diversity.

We again utilize the MMR framework we described in the previous subsection where the language model of a result tuple T is defined as follows.

Definition 7.5 *:* **Knowledge-Base-based Language Model**
The knowledge-base-based language model of Tuple T *is a mixture model of all the language models of all the resources that appear in* T.

The parameters of the knowledge-base-based language model are estimated as follows. Let $RES(T)$ be the set of all resources X that appear in T, and let $P(w|X)$ be the probability of a term w in the language model of resource X. The probability of a term w in the language model of tuple T is then estimated as follows:

$$P(w|T) = \frac{1}{|RES(T)|}\Sigma_{X \in RES(T)} P(w|X) \tag{7.3}$$

7.2.3. Text-based diversity

Recall that each triple can be associated with a text snippet. Such text snippet can be directly utilized to provide diversity among the different results in the top rankings using the MMR measure. We first define a language model for each triple and then use these language models to construct the language models of the result tuples.

Definition 7.6 *:* **Triple Language Model**
The triple language model is a probability distribution over all the terms in all the text snippets of all the triples in the knowledge base KB.

The parameters of the triple language model is computed using a maximum-likelihood estimator as follows:

$$P(w|t) = \alpha\frac{c(w; D(t))}{|D(t)|} + (1-\alpha)\frac{c(w; Col)}{|Col|} \tag{7.4}$$

where $c(w; D(t))$ is the number of times the term w occurs in $D(t)$ (the text snippet of triple t), Col is the whole collection composed of all text snippets of all triples in the knowledge base KB, and $|D(t)|$ and $|Col|$ are the number of occurrences of all terms in $D(t)$ and Col, respectively. The text-based tuple language model is then defined as follows.

Definition 7.7 : Text-based Language Model
The text-based language model of tuple T *is a mixture model of the language models of the triples that constitute the tuple* T.

The parameters of the text-based tuple language model are then estimated as follows. Let $T = (t_1, t_2, ..., t_n)$ where t_i is a triple, the probability $P(w|T)$ of term w in the language model of tuple T is computed as:

$$P(w|T) = \frac{1}{n}\Sigma_{i=1}^{n} P(w|t_i) \tag{7.5}$$

where $P(w|t_i)$ is the probability of term w in the language model of triple t_i which is estimated according to Equation 7.4.

7.3. Related Work

Result diversity for document retrieval has gained much attention in recent years. The work in this area deals primarily with unstructured and semi-structured documents [1, 10, 12, 15, 33, 89]. Most of the techniques perform diversification by optimizing a bi-criteria objective function that takes into consideration both result relevance as well as result novelty (a measure of diversity) with respect to other results. Gollapudi and Sharma [33] presented an axiomatic framework for this problem and studied various objective functions that can be used to define such optimization problem. They proved that in most cases, such problem is hard to solve and proposed several approximation algorithms to solve such problem. Carbonell and Goldstein introduced the Maximal Marginal Relevance (MMR) method [10] which is one approximation solution to such optimization problem. Zhai et al. [89] studied a similar approach within the framework of language models and derived an MMR-based loss function that can be used to perform diversity-aware ranking. Aragwal et al. [1] assumed that query results belong to different categories and they proposed an objective function that tries to trade off the relevance of the results with the number of categories covered by the selected results. In an RDF setting, where results are constructed at query time by joining triples, we do not have an explicit notion of result categories. We thus adopted [10] to the setting of RDF data since it directly utilizes the results to perform diversity rather than explicitly taking the categories of the results into consideration.

Result diversity can also be achieved by clustering or classification of search results. Both techniques group search results based on their similarities, so that users can navigate to the right groups to retrieve the desired information. For example, clustering and classification have been applied to image retrieval [84] as well as database query-results [13]. Clustering is usually performed as a post processing step. However, with the optimization approaches that use bi-criteria objective functions, one can seamlessly integrate both result relevance and diversity in a controlled manner that automatically balances these two factors for each result. That is, we could ideally have more than one result from the same cluster ranked higher than those from other clusters if these results are much more relevant to the query than the results from other clusters. Similarly, classification suffers from the same problem of ranking result groups rather than the results themselves, and in addition, classes are usually pre-defined which is something we lack in the case of searching RDF knowledge where the results are tuples of joined triples constructed at query time.

Apart from document retrieval, there is very little work on result diversity for queries over structured data. In [13] the authors propose to navigate SQL results through categorization, which takes into account user preferences. They assume there exists query logs and utilize them to identify different categories of user's preferences and then use these categories to classify the query results. In [85], the authors introduce a pre-indexing approach for efficient diversification of query results on relational databases. To do so, they ensure that the results retrieved from a given relation are diverse in terms of their attributes. However, their work applies only to relational databases, where there is an explicit schema, which is something we lack in RDF in general. Moreover, they do not take into consideration the relevance of the results to the query. The work in [19] also deals with relational databases and aims at diversifying the search results to keyword queries. The approach taken in [19] is to identify the different interpretations of the keyword query, which are inherently ambiguous, and then classify the query results according to these different interpretations. While we could make use of this approach for the case of keyword queries over RDF data, it is not applicable to the case of triple-pattern queries, where the queries are in semantically well-defined and do not involve ambiguities. It is nonetheless still desirable to diversify the results of such queries in order to provide the users with a broader view of the possible query results.

More recently, the work on co-reference aware Web object retrieval [18] deals with the problem of retrieving diverse Web objects that are extracted from multiple Web sources. While, their work is very close in spirit, they mainly deal with the issue of duplicates in such collections, where many instances of the same object can be present. Their goal is to retrieve a set of such unique objects in response to a user keyword query. Our setting is in large very different from their setting, since we assume that we focus on a single RDF knowledge base, that in principal, contains no duplicate resources (i.e., Web objects). In addition, our query results are not just a list of objects, but a set of tuples of triples, and our goal is to diversify all the resources in such triples, including the different relationships between pairs of resources.

7.4. Evaluation

Evaluation of non-traditional retrieval models such as the one we presented in this chapter poses a big challenge. Most IR evaluation metrics are designed to evaluate traditional retrieval tasks; i.e., the relevance judgments are made independently for each result. However, in our case, we would want to judge a set of results (say the top-k) collectively. One way to do this, is to provide the top-k diversified results and the top-k results without diversity for a set of benchmark queries and ask a set of human evaluators to judge which result set they find more useful or prefer. However, this is very subjective, and might differ from one user to the other. For instance, one user might always prefer to see highly relevant results regardless of their diversity whereas another might prefer to sacrifice a bit of relevance for the sake of diversity. Indeed, the optimal trade-off between relevance and diversity is a user factor and thus very hard to quantify [90].

An alternative evaluation mechanism is to assume that each user is interested in finding one single result for any query in the query benchmark and then measure the percentage of user satisfaction per result set (i.e., diversified versus non-diversified). This would require a relatively large query benchmark and in addition, a large base of human judges in order to find a significant number of users for each query with different interpretation of the query, or different information needs for the same query. Unfortunately, we could not do this through the course of this thesis, but we believe that creating such an evaluation benchmark would be very useful in the context of evaluating diversity approaches for results to queries over RDF data.

To overcome some of the issues mentioned above, there have been many attempts in order to adapt IR evaluation metrics for the task of diversity-aware retrieval. Most of these approaches rely on the explicit notion of aspects for queries. That is, each query is assumed to be referring to a set of different aspects or subtopics (with different relevance values in some cases), and in turn each query result can cover one or more query aspects. Traditional IR metrics can then be modified to take into consideration these aspects while evaluating the performance of a retrieval model. For example, Zhai et al. [89] developed few precision and recall based measures that take into consideration the number of aspects covered by a result set, the uniqueness of the aspects covered and the relevance of the results to the query. Argawal et al. proposed a similar metric based on NDCG [1] that also takes into consideration the number of aspects covered by a result set, the relevance of the aspects with respect to the query as well as the relevance of the results to the query. Developing similar metrics in the case of RDF search is slightly harder especially for triple-pattern queries that are usually very clear making the process of identifying query aspects more vague to define. This requires a thorough study over a large benchmark of queries which is again outside the scope of this thesis.

Finally, we point out that comparing different diversity methods in the setting of RDF search is also a very challenging problem as results are assumed to be tuples of triples matching a given query. This concise representation lacks context making it hard to judge which diversity method performs better or produces a more useful set of diversified results. To overcome this, one must present additional contextual information with the query results to help evaluators compare the different query results in order to finally judge which they perceive as a more useful diversity approach. Result Representation is something we do not consider in this thesis.

For all the above mentioned reasons, we evaluate our framework for result diversity with its three incarnations via a set of qualitative examples. With these examples, we highlight the merits each method might have over the other methods, and show case why such method might be needed for certain types of queries. We also explain the overhead involved in each of these three methods.

Undiversified Results	Diversified Results
Tom_Hanks actedIn Forrest_Gump	Tom_Hanks actedIn Forrest_Gump
Tom_Hanks actedIn The_Da_Vinci_Code	Tom_Hanks actedIn The_Da_Vinci_Code
Tom_Hanks actedIn Saving_Private_Ryan	Tom_Hanks actedIn Saving_Private_Ryan
Tom_Hanks actedIn Sleepless_in_Seattle	Tom_Hanks hasChild Colin_Hanks
Tom_Hanks actedIn Philadelphia	Tom_Hanks isMarriedTo Rita_Wilson
Tom_Hanks actedIn The_Terminal	Tom_Hanks hasWonPrize Saturn_Award
Tom_Hanks actedIn You've_Got_Mail	Tom_Hanks bornOnDate 1956-07-09
Tom_Hanks actedIn Apollo_13	Tom_Hanks livesIn Concord,_California
Tom_Hanks actedIn Toy_Story	Tom_Hanks produced Charlie_Wilson's_War
Tom_Hanks actedIn Toy_Story_2	Tom_Hanks directed That_Thing_You_Do!

Table 7.2.: Results for the example query "Tom Hanks"

Resource-based Diversity Example. In Table 7.2, we show the results of an example query to find all relevant triples about Tom Hanks. The left column is the top-10 results without any diversification. As can be seen, all the results are about movies that Tom Hanks acted in. While all the results are highly relevant to Tom Hanks, they span only one aspect about him; mainly his career as an actor. On the right column of Table 7.2, we show the top-10 diversified results, where the results were diversified using the resource-based language models for tuples. Clearly, the diversified result set provides many different information about Tom Hanks, including his spouse, his children, birthdate and place of residency, awards, that is in addition to prominent movies he acted in, and even directed and produced.

Knowledge-base-based Diversity Example. In Table 7.3, we show the results of an example query to find all actors that have won an Academy Award for Best Actor. Due to space limitation, we just show the URIs of the actors. The left column is the top-10 results without any diversification. The top-10 results are all about American film actors. On the right column of Table 7.2, we show the top-10 diversified results, where the results were diversified using the knowledge-base-based language models for tuples (Subsection 7.2.2). The diversified result set contains a more diverse set of actors including British,

Undiversified Results	Diversified Results
Dustin_Hoffman	Dustin_Hoffman
Jamie_Foxx	Russell_Crow
Clark_Gable	Nigel_Hawthorne
Sean_Penn	Emil_Jannings
Nicolas_Cage	Maximilian_Schell
Kevin_Spacey	Roberto_Benigni
Robert_De_Niro	Jamie_Foxx
Will_Smith	Leslie_Howard
Jack_Lemmon	Peter_Finch
Tom_Hanks	Clark_Gable

Table 7.3.: Results for the example query "actors that have won the Academy Award for Best Actor"

German, Australian actors, and so on. The results also contain in addition to film actors, TV actors, and a mix of modern alive actors as well classic deceased ones. Note that the resource-based diversity approach would not be able to achieve this level of diversity as it takes into consideration only the direct resources that appear in the query results, which are all different for our example query. The knowledge-base-based approach on the other hand, tries to make use of the knowledge present about each resource in the whole knowledge as a more complete representation of the resource rather than just relying on the resource identifier. This more complete representation of resources however comes with a price, as the knowledge-base-based diversity is more expensive than the simple resource-based one, since the language models for the resources that appear in the results, which are typically quite big, have to be compared against each other.

Text-based Diversity Example. In Table 7.4, we show the results of an example query to find all movies of genre Fantasy. The left column is the top-10 results without any diversification. The top-10 results are all fantasy movies, however many sequels of the same movies are ranked in the top-10 results (for example, Toy Story or Spider Man). On the right column of Table 7.4, we show the top-10

Undiversified Results	**Diversified Results**
Shrek hasGenre Fantasy	Shrek hasGenre Fantasy
Superman_Returns hasGenre Fantasy	Superman_Returns hasGenre Fantasy
Spider-Man_3 hasGenre Fantasy	Toy_Story_3 hasGenre Fantasy
Toy_Story_3 hasGenre Fantasy	Star_Wars hasGenre Fantasy
Batman_Begins hasGenre Fantasy	Clash_of_the_Titans hasGenre Fantasy
Spider-Man_2 hasGenre Fantasy	Kate_&_Leopold hasGenre Fantasy
Shrek_2 hasGenre Fantasy	Bruce_Almighty hasGenre Fantasy
Clash_of_the_Titans hasGenre Fantasy	Jurassic_Park hasGenre Fantasy
Batman_Returns hasGenre Fantasy	Twilight hasGenre Fantasy
Batman_Forever hasGenre Fantasy	Alice_in_Wonderland hasGenre Fantasy

Table 7.4.: Results for the example query "fantasy movies"

diversified results, where the results were diversified using the text-based language models for tuples (Subsection 7.2.3). The diversified result set contains a more diverse set of fantasy movies that include movies about outer space, animation movies, classical fairy tails as well as science fiction movies. Note that neither the resource-based diversity approach nor the knowledge-base-based one would be able to achieve this level of diversity as they do not take into consideration the text associated with the triples which contain additional context for the triples. Again, in comparison to the resource-based approach, the text-based approach involves additional overheard of comparing the text snippets of the different results, which can be in general large as compared to just the list of resources in a given result.

7.5. Summary

In this Chapter, we presented a diversity-aware ranking model for queries over RDF knowledge bases, whether triple-pattern queries or keyword queries. Our model is based on optimizing a bi-criteria objective function that trades off the relevance of results and their diversity using a Maximal Marginal Relevance approach. To measure the diversity of a set of results, our model can use different sources of information ranging from just the resources that appear in the results, to utilizing additional information about the resources from the knowledge base or using the text associated with the triples, if any exists. Our model can also combine these different sources of information to achieve a higher level of diversity. We evaluated our framework using a set of qualitative examples which motivated the need for result diversity, and highlighted the merits and the limitations of the different sources of information that can be used to quantify diversity.

Chapter 8. Knowledge Exploration

By searching RDF knowledge bases, users can find precise information about certain subjects of interest. Query results are typically a single resource, a set of resources or tuples of joined triples. While this is a very concise representation of answers to users' information needs, it is often the case that users would like to learn more about their query results. In this chapter, we present two different tools to allow users to find additional contextual information about their query results. The first tool, ROXXI, is a document retrieval tool that retrieves a set of *witness documents* for a given set of RDF triples. The second tool, CATE, is a context-aware entity summarization tool that retrieves information about an entity of interest from an RDF knowledge base as well as unstructured data sources and displays the retrieved information in an interactive timeline fashion.

8.1. ROXXI: Reviving Witness Documents to Explore Extracted Information

Most large RDF knowledge bases are constructed by deploying information-extraction techniques from structured, semi-structured as well as, to a limited extent, unstructured information sources [78, 4, 30]. However, the-state-of-the-art information-extraction techniques [79, 62] still cannot capture every piece of information present in the world, which renders such knowledge bases often incomplete.

For example, when looking for detailed information about the movie From Dusk Till Dawn, we can query an RDF knowledge base and find the information that Quentin Tarantino wrote, produced, and acted in this movie. However, we cannot find information such as the movie plot or tag lines which are naturally present in the form of free text and cannot be meaningfully expressed in the form of RDF triples. Another piece of information that we will not find in an RDF knowledge base, even though available in the textual documents, is that Quentin Tarantino was originally set to direct the From Dusk Till Dawn, but in the end decided against it so that he could focus more on his tasks as actor and screenplay writer. That is, RDF knowledge bases do not consider "potential" facts or explanations.

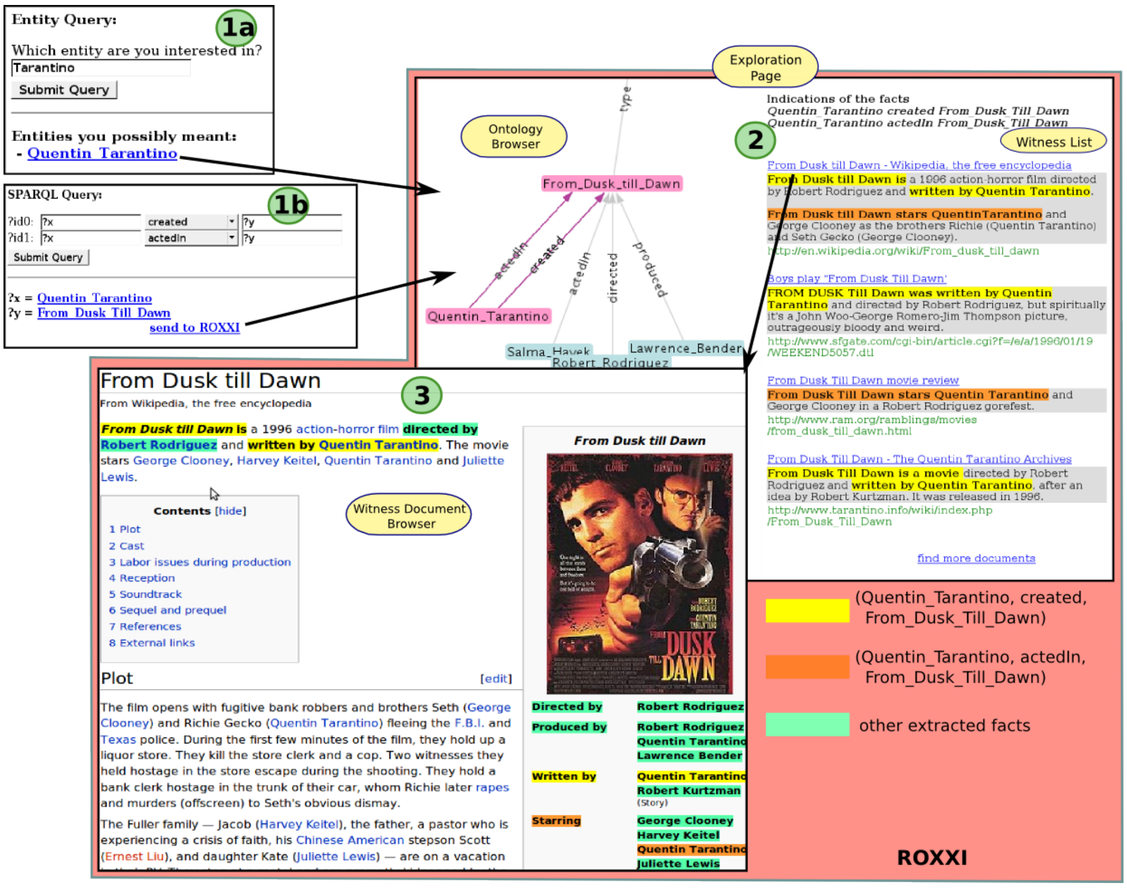
In this section, we present a system that enables knowledge exploration on top of RDF knowledge bases. Our system combines the benefits of structured search in RDF knowledge bases and document retrieval to better serve the user's information needs. In particular, our system retrieves a ranked list of witness documents for a given set of RDF triples, i.e., documents that contain the information encoded in the given RDF triples . The information about the connections between the triples and the documents can either be collected during the knowledge base construction phase or added later on.

8.1.1. Knowledge Exploration with ROXXI

To start knowledge exploration with ROXXI, a user first identifies a set of RDF triples she is interested in, which we refer to as an *RDF subgraph*. Recall that an RDF knowledge base can be viewed as a graph where each RDF triple can be represented using an edge with the subject and object of the triple corresponding to vertices and the predicate corresponding to the label of the edge.

ROXXI offers two ways to provide RDF subgraphs, one for users acquainted with triple-pattern search and the other for casual users. The first allows users to retrieve RDF subgraphs via triple-pattern queries whereas the second allows them to search for entities in the knowledge base. A triple-pattern query and a corresponding result for the example about Quentin Tarantino writing and acting in the same movie are shown as step 1b in Figure 8.1. Step 1a in Figure 8.1 shows the results when performing an entity search for "Tarantino". Once the user clicks on either "send result to ROXXI" below a result in the triple-pattern search interface or on a certain entity, the *Exploration Page* opens.

The *Exploration Page* offers two interconnected ways to learn more about a certain entity or RDF subgraph: a graphical *Ontology Browser* and the *Witness List*.



Entity Query:

Which entity are you interested in?
Tarantino
Submit Query

Entities you possibly meant:
- Quentin Tarantino

SPARQL Query:

?id0:	?x	created	?y
?id1:	?x	actedIn	?y

Submit Query

?x = Quentin Tarantino
?y = From Dusk Till Dawn
send to ROXXI

Indications of the facts
Quentin_Tarantino created From_Dusk_Till_Dawn
Quentin_Tarantino actedIn From_Dusk_Till_Dawn

From Dusk till Dawn - Wikipedia, the free encyclopedia
From Dusk till Dawn is a 1996 action-horror film directed by Robert Rodriguez and **written by Quentin Tarantino**.

From Dusk till Dawn stars QuentinTarantino and George Clooney as the brothers Richie (Quentin Tarantino) and Seth Gecko (George Clooney).
http://en.wikipedia.org/wiki/From_dusk_till_dawn

Boys play "From Dusk Till Dawn'
FROM DUSK Till Dawn was written by Quentin Tarantino and directed by Robert Rodriguez, but spiritually it's a John Woo-George Romero-Jim Thompson picture, outrageously bloody and weird.
http://www.sfgate.com/cgi-bin/article.cgi?f=/e/a/1996/01/19/WEEKEND5057.dtl

From Dusk Till Dawn movie review
From Dusk Till Dawn stars Quentin Tarantino and George Clooney in a Robert Rodriguez gorefest.
http://www.ram.org/ramblings/movies/from_dusk_till_dawn.html

From Dusk Till Dawn - The Quentin Tarantino Archives
From Dusk Till Dawn is a movie directed by Robert Rodriguez and **written by Quentin Tarantino**, after an idea by Robert Kurtzman. It was released in 1996.
http://www.tarantino.info/wiki/index.php/From_Dusk_Till_Dawn

find more documents

🟨	(Quentin_Tarantino, created, From_Dusk_Till_Dawn)
🟧	(Quentin_Tarantino, actedIn, From_Dusk_Till_Dawn)
🟩	other extracted facts

From Dusk till Dawn
From Wikipedia, the free encyclopedia

***From Dusk till Dawn* is** a 1996 action-horror film **directed by Robert Rodriguez** and **written by Quentin Tarantino**. The movie stars George Clooney, Harvey Keitel, Quentin Tarantino and Juliette Lewis.

Contents [hide]
1 Plot
2 Cast
3 Labor issues during production
4 Reception
5 Soundtrack
6 Sequel and prequel
7 References
8 External links

Plot [edit]

The film opens with fugitive bank robbers and brothers Seth (George Clooney) and Richie Gecko (Quentin Tarantino) fleeing the F.B.I. and Texas police. During the first few minutes of the film, they hold up a liquor store. They kill the store clerk and a cop. Two witnesses they held hostage in the store escape during the shooting. They hold a bank clerk hostage in the trunk of their car, whom Richie later rapes and murders (offscreen) to Seth's obvious dismay.

The Fuller family — Jacob (Harvey Keitel), the father, a pastor who is experiencing a crisis of faith, his Chinese American stepson Scott (Ernest Liu), and daughter Kate (Juliette Lewis) — are on a vacation

From Dusk till Dawn

Directed by	**Robert Rodriguez**
Produced by	**Robert Rodriguez** **Quentin Tarantino** **Lawrence Bender**
Written by	**Quentin Tarantino** **Robert Kurtzman** (Story)
Starring	**George Clooney** **Harvey Keitel** **Quentin Tarantino** **Juliette Lewis**

Figure 8.1.: Knowledge Exploration with ROXXI

The *Ontology Browser* allows users to easily navigate through the RDF knowledge base by selecting or deselecting triples in the underlying graph representing the knowledge base and thus defining the RDF subgraph to be explored.

The *Witness List* shows a ranked list of documents containing (some of) the knowledge expressed by the chosen subgraph. In order to do so, ROXXI maintains a list of *textual expressions* for every triple in the RDF knowledge base. These textual expressions can be generated using an information-extraction tool. For example, the textual expression ”Quentin Tarantino was one of the producers of From Dusk Till Dawn” can be used to represent the triple

```
Quentin_Tarantino   produced   From_Dusk_Till_Dawn
```

Using these textual expressions, a ranked list of documents containing the information encoded by the selected RDF subgraph can be retrieved. For each retrieved document, a snippet from the document's content is shown. The snippet shows the textual expressions for some of the triples in the RDF subgraph present in the document. Step 2 in Figure 8.1 illustrates this for the RDF subgraph representing the information that Quentin Tarantino wrote and acted in From Dusk Till Dawn (see highlighted text in *Witness List* frame in Figure 8.1).

Furthermore, the user can explore one or more of the retrieved documents. For each retrieved document, the textual expressions used to express the triples of interest are highlighted inside the document using the same colors as used in the *Witness List*. In addition, textual expressions corresponding to other triples (not in the selected subgraph) are highlighted in a different color (green) as shown in step 3 in Figure 8.1.

8.1.2. System Architecture

ROXXI's system architecture is depicted in Figure 8.2. It consists of 3 main components: a *Data Manager*, a *Query Engine* and a *User Interface*. We next describe each component in more detail.

Data Manager

The *Data Manager* manages the data ROXXI operates on. This data consists of: the RDF knowledge base (RDF KB), a document collection (DOCS) and a dictionary which contains the textual expressions for the triples in the Knowledge base. The textual expressions can be generated using an information-extraction tool such as SOFIE [79]. Each textual expression in ROXXI is associated with a *confidence* value reflecting its accuracy. For example, Table 8.1 shows a set of textual expressions for an example triple and their confidence values. For each textual expression, we also keep track of the set of documents it occurs in and the start and end position of the expression in the document. This is later used to retrieve the witnesses for a given RDF subgraph, which we annotate with the corresponding textual expressions before presenting them to the user.

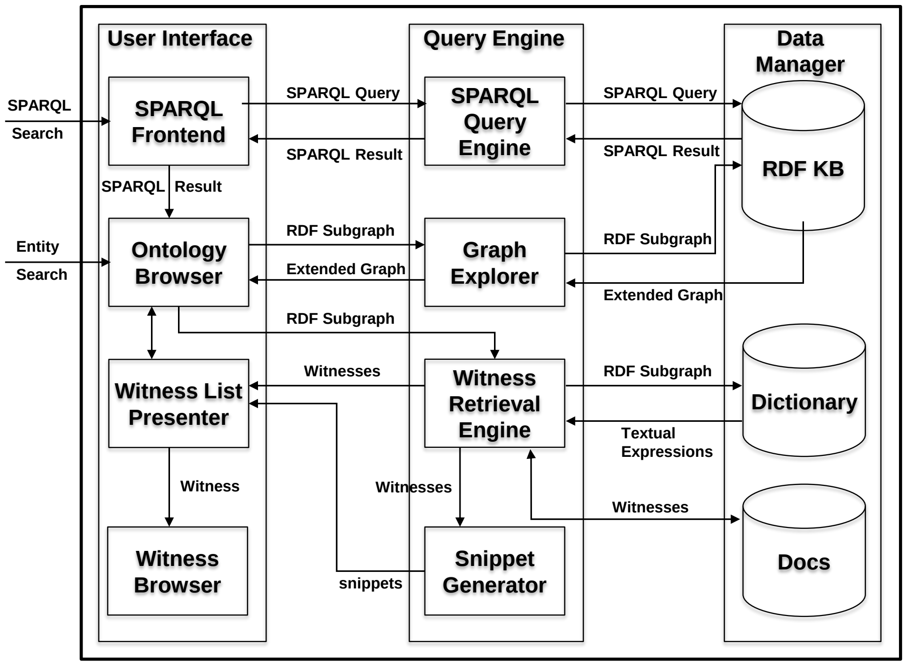


Figure 8.2.: ROXXI's System Architecture

Query Engine

The query engine consists of 4 subcomponents, which we describe separately in the following.

Graph Explorer. The graph explorer takes an RDF subgraph as input. An RDF subgraph can either be an entity, an RDF triple, or a set of RDF triples. The graph explorer expands this subgraph by retrieving additional neighboring triples from the RDF knowledge base and returns an extended graph that can be explored by the user using the *Ontology Browser*.

SPARQL Query Engine. It takes a triple-pattern query as input and returns a ranked list of tuples of joined triples matching the given query.

Textual Expression r	$\mathtt{conf}(r, t)$
Quentin Tarantino acted in From Dusk Till Dawn	0.995
Quentin Tarantino played a role in From Dusk Till Dawn	0.992
Quentin Tarantino starred in From Dusk Till Dawn	0.782
Quentin Tarantino appeared in From Dusk Till Dawn	0.759
Tarantino acted in From Dusk Till Dawn	0.992
Tarantino played a role in From Dusk Till Dawn	0.899
Tarantino starred in From Dusk Till Dawn	0.754
Tarantino appeared in From Dusk Till Dawn	0.700
Quinton Tarantino acted in From Dusk Till Dawn	0.801
Quinton Tarantino played a role in From Dusk Till Dawn	0.754
Quinton Tarantino starred in From Dusk Till Dawn	0.544
Quinton Tarantino appeared in From Dusk Till Dawn	0.432

Table 8.1.: A set of textual expressions and their confidence values for the example triple: `Quentin_Tarantino actedIn From_Dusk_Till_Dawn`

Witness Retrieval Engine. It is responsible for retrieving and ranking the witness documents for a given RDF subgraph. Our ranking model is based on statistical language-models [69]. The documents are ranked based on the probability of being relevant to the given RDF subgraph $g = \{t_1, t_2, ..., t_n\}$, which we denote as $P(d|g)$. Applying Bayes' rule and ignoring $P(g)$ as it is document independent, we have:

$$P(d|g) \propto P(g|d)P(d)$$

$P(d)$ is a prior probability that a document d is relevant to any RDF subgraph. This probability can be estimated in various ways, and in our case we estimate it using the static authority of the page or *pagerank* [9].

The probability $P(g|d)$ is the probability that the given RDF subgraph $g = \{t_1, t_2, ..., t_n\}$ was generated from document d. We assume independence between the triples in g for computational tractability (in-line with most traditional IR models). In addition, we apply smoothing with a collection background model (Col) to avoid overfitting. Thus,

$$P(g|d) = \prod_{i=1}^{n} [\alpha P(t_i|d) + (1 - \alpha)P(t_i|Col)]$$

Since the triples do not directly appear in the documents but are present there in the form of textual expressions, we need to first fold a triple into all its textual expressions. This is similar in spirit to translation models [91], where the query is expressed in one language, and the documents retrieved are in a different language. Let the set $R = \{r_1, r_2, ..., r_m\}$ be the set of all possible textual expressions for all the triples in our knowledge base. Furthermore, let each textual expression r_j be associated with a confidence value $conf(r_j, t_i)$ representing how well expression r_j expresses triple t_i. The confidence $conf(r_j, t_i)$ would be zero in case expression r_j does not express triple t_i at all. The probabilities $P(t_i|d)$ and $P(t_i|col)$ are then estimated as follows:

$$P(t_i|x) = \Sigma_{j=1}^{m} p(t_i|r_j) P(r_j|x)$$

The first component $P(t_i|r_j)$ is the probability of expressing triple t_i using expression r_j. It is estimated using the confidence value $conf(r_j, t_i)$. The second component $P(r_j|x)$ is the probability of expression r_j being generated from x where $x \in \{d, Col\}$. It is estimated using a maximum-likelihood estimator as follows:

$$P(r_j|x) = \frac{c(r_j; x)}{\Sigma_{k=1}^{m} c(r_k; x)}$$

where $c(r; x)$ denotes the frequency of expression r in x.

Snippet Generator. The snippet generator is responsible for generating a snippet for each retrieved document. Similar to most state-of-the-art search engines, we generate a query-biased snippet. The snippet contains (some of) the textual expressions that occur in the retrieved document. For snippet generation in ROXXI, we use the simple technique of presenting the textual expressions that highly match the user query from [87].

User Interface

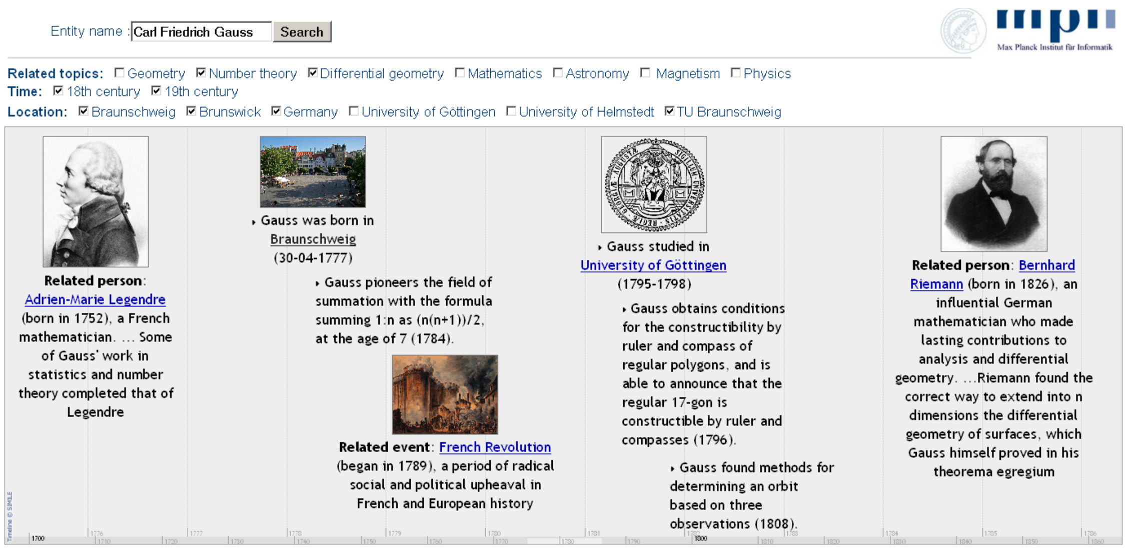
The user interface contains facilities for both the casual as well as the expert user. The expert users can utilize the *SPARQL Frontend* to issue triple-pattern queries and retrieve matching tuples of triples. On the other hand, the less advanced users can navigate directly to the exploration page described in Section 8.1.1 by performing an entity search.

The knowledge base can be browsed using the *Ontology Browser* which renders a hyperbolic visualization of the graph representing the RDF knowledge base. Our ontology browser is based on the Prefuse visualization tool [1]. Furthermore, the user can navigate through the graph as desired and select or deselect triples to retrieve witness documents.

The *Witness List Presenter* displays a set of ranked documents for the selected RDF subgraph in the Ontology Browser. Finally, the *Witness Browser* utilizes a Web Browser plug-in to highlight the textual expressions for the RDF triples in the viewed document.

8.2. CATE: Context-Aware Timeline for Entity Illustration

In this section, we present CATE which is a system that combines structured information present in RDF knowledge bases with semi-structured and unstructured information in order to create a comprehensive summary of a given entity of interest. We focus on Wikipeda as the main information source and utilize a Wikipedia-based RDF knowledge base such as [78, 4, 30] as our RDF knowledge base. Wikipedia has now become one of the most authoritative information-sources on the Web. It contains millions of articles about people, countries, historical events, research topics, inventions, etc. Typically, each article in Wikipedia describes an entity. Wikipedia contains also a hierarchy of categories, where each entity belongs to a set of categories, and each category is a sub-category of one or more categories. For example, the article about the famous German mathematician Carl Friedrich Gauss is included in the categories `1777_births`, `18th-century_mathematicians`, `German_mathematicians`, etc. The categories provide a *context* for the entity. That is, for Gauss, we can infer from his categories that he is German, a mathematician, lived in the 18th century, etc.


Entity name : Carl Friedrich Gauss
Search
Max Planck Institut für Informatik

Related topics: ☐ Geometry ✓ Number theory ✓ Differential geometry ☐ Mathematics ☐ Astronomy ☐ Magnetism ☐ Physics
Time: ✓ 18th century ✓ 19th century
Location: ✓ Braunschweig ✓ Brunswick ✓ Germany ☐ University of Göttingen ☐ University of Helmstedt ✓ TU Braunschweig

Related person: Adrien-Marie Legendre (born in 1752), a French mathematician. ... Some of Gauss' work in statistics and number theory completed that of Legendre

Gauss was born in Braunschweig (30-04-1777)

Gauss pioneers the field of summation with the formula summing 1:n as (n(n+1))/2, at the age of 7 (1784).

Related event: French Revolution (began in 1789), a period of radical social and political upheaval in French and European history

Gauss studied in University of Göttingen (1795-1798)

Gauss obtains conditions for the constructibility by ruler and compass of regular polygons, and is able to announce that the regular 17-gon is constructible by ruler and compasses (1796).

Gauss found methods for determining an orbit based on three observations (1808).

Related person: Bernhard Riemann (born in 1826), an influential German mathematician who made lasting contributions to analysis and differential geometry. ...Riemann found the correct way to extend into n dimensions the differential geometry of surfaces, which Gauss himself proved in his theorema egregium

Figure 8.3.: Knowledge Exploration with CATE

8.2.1. Knowledge Exploration with CATE

Given an entity of interest, CATE constructs a comprehensive timeline that contains all the relevant events related to the given entity. For example, the timeline of Gauss shown in Figure 8.3 contains the information that he was born in Braunschweig and that he studied in the University of Göttingen. In addition, it contains the event that the French revolution broke in 1789 which is a major historical event in Europe during his lifetime. The timeline also contains events related to the entities Legendre and Riemann, two famous mathematicians whose work is closely related to that of Gauss in the fields of number theory and differential geometry, respectively. All these events are retrieved from Gauss Wikipedia page or Wikipedia pages of other entities highly relevant to Gauss and his contexts.

In addition to the events related to the entity, CATE provides the user with a set of relevant *contexts* which are used to focus the timeline on one or more contexts of choice (as can be seen in the upper part of Figure 8.3). We define the context as an object with three attributes, namely time, space and topic. For example, for Gauss, these attribute are:

[1]http://prefuse.org

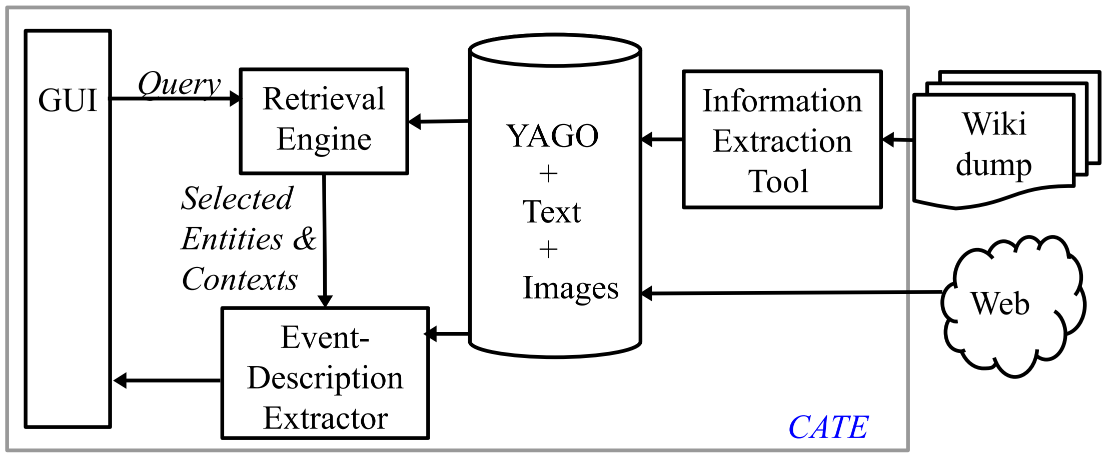

GUI
Query
Retrieval Engine
Selected Entities & Contexts
Event-Description Extractor
YAGO + Text + Images
Information Extraction Tool
Wiki dump
Web
CATE

Figure 8.4.: CATE's System Architecture

- time: 18-th century, 19-th century, ...
- space: Braunschweig, Brunswick, Germany, Europe, ...
- topic: number theory, differential geometry, astronomy, ...

In order to construct an informative timeline such as the one in Figure 8.3, we need to perform three main tasks. The first is to associate each entity with a set of contexts. Second, given an entity (and possibly a subset of its contexts), we need to retrieve relevant entities to such entity or its contexts. For example, for Gauss, the relevant entities are Legendre, Riemann, French revolution, etc. Finally, given the entity and its relevant contexts and entities, we need to extract the related events to place them on the timeline. CATE relies on Wikipedia as the source of information to perform all three tasks and we explain how in the rest of this section.

8.2.2. System Architecture

As shown in Figure 8.4, CATE consists of five main components: a graphical user-interface (GUI), a retrieval engine, an event-description extractor, a data store, and an information-extraction tool. In a nutshell, CATE works as follows. The information-extraction tool is used to populate our data store and uses two sources of information: the Wikipedia corpus and its dumps, and the Web. To interact with CATE, the user inputs the name of an entity into a text box in the

GUI. The GUI passes the input entity to the retrieval engine which retrieves all relevant contexts and entities from the data store. These are further sent to the event-description extractor, which extracts all related textual descriptions of the events. Finally, the results are sent to the GUI to be displayed to the user. We now explain each component in more details.

GUI. The GUI consists of two main sub-components. The first sub-component is the timeline illustrator where the events are positioned on the timeline and illustrated with images and text snippets. The second sub-component is the context selector (upper part of Figure 8.3) where relevant context-attributes are shown in the form of a menu. We do not enumerate the set of relevant contexts as many of them are overlapping. The user can use the context selector to control the visualized information on the timeline by selecting contexts of interest based on their attributes.

Retrieval Engine. The retrieval engine performs three types of retrieval tasks. It retrieves the most relevant contexts given an entity (e.g., `number_theory` and `differential_geometry` for Gauss). It also retrieves the most relevant entities given a certain context (e.g., the entity French revolution given the context `18th_century_Europe`). In addition, given an entity and context, it retrieves the most relevant entities to the given entity and context (e.g., Riemann given the entity Gauss and the context `differential_geometry`). We describe the ranking model used by the retrieval engine in Subsection 8.2.5.

Event-Description Extractor. The event-description extractor takes as input a Wikipedia article identifier and a query and retrieves the set of events related to the query from the article. The query can be either an entity name, a context name or both. The output of the event-description extractor is a set of events in the form of an image, a text snippet and a timestamp.

Information-Extraction Tool. CATE's extraction tool uses two sources of information: the Wikipedia dump and the Web. The Wikipedia dump is used to extract context information as well as hyperlink and text information which are used by both the retrieval engine and the event-description extractor. We explain our extraction algorithms in Subsection 8.2.3. The Web corpus on the other hand is used, via a well-known Web search-engine, to extract images.

Data Store. Our data store contains three databases. The first is the YAGO knowledge base [78], an RDF database that contains the RDF triples and the context information. YAGO has inferred class memberships from Wikipedia category names, and has integrated this information with the taxonomic backbone of WordNet. We extend the YAGO knowledge base with a new class of entities, namely the contexts.

Conceptually, a context C is an object with three types of attributes: time, space and topic. For example, for Gauss the following RDF triples about his contexts are stored in our RDF knowledge base:

```
Carl_Friedrich_Gauss  hasContext  c:18-de-math
c:18-de-math          hasTime     18th-century
c:18-de-math          hasSpace    Germany
c:18-de-math          hasTopic    Mathematics
Carl_Friedrich_Gauss  hasContext  c:19-br-ph
c:19-br-ph            hasTime     19th-century
c:19-br-ph            hasSpace    Braunschweig
c:19-br-ph            hasTopic    Physics
```

The attributes are mapped to YAGO entities. Hence, relationships between attributes are implicitly present in the knowledge base. For instance, for the space attribute we have the relationship `partOf` (e.g., `(France, partOf, Europe)`). The relationships between contexts are characterized based on the relationships between their attributes.

The second database in our data store is a text database. It contains three types of indices that are used by different components in CATE. The first index is an inverted index over the hyperlinks. In particular, for each Wikipedia article, it stores all other articles that have outgoing links to it and the number of such links. The second index in the text database is a traditional inverted index that stores for each term all the articles that contains the term and the term frequency in the articles. These two indices are used by the retrieval engine to rank entities and contexts as we describe in Subsection 8.2.5. The third index in the text database is a full-text index that simply stores the full text of the articles. This is used by the event-description extractor to extract the snippets of the events as we explain in Subsection 8.2.6. Note that YAGO keeps a mapping between the entity and the corresponding Wikipedia article (if any exists). This is used by CATE to connect entities to articles.

The third database in our data store is an image database. It contains an image for each entity in YAGO.

8.2.3. Information extraction

Our extraction tool extracts three types of information: the context information, text and hyperlink information, and images.

Context Extraction. We utilize the Wikipedia categories and their hierarchy to extract contexts. Wikipedia category-names are usually composite names such as `18-th_century_mathematicians` or `German_mathematicians`. These composite names consisting of orthogonal attributes pose a problem especially when constructing ontologies [78, 63]. However, looking closely, the category names in Wikipedia usually follow patterns that combine time, space and topic attribute-values. This observation was the basis of our 3-attributes model for contexts.

CATE extracts context information through a two-phase algorithm. In the first phase, it enumerates all Wikipedia categories and generates possible vocabularies for each of the three attributes. The vocabularies for time and space are extracted based on defined patterns, while those for topic are extracted using a set of automatically-learnt patterns. In the second phase, the algorithm annotates each category using the extracted vocabularies. For example, the category `18-th_century_mathematicians` is associated with the attributes: time equals `18th-century` and topic equals `Mathematics`.

Hyperlink and Text Extraction. The Wikipedia dump provides a table with link information for each article. We utilize this and the full text of the articles to create the three inverted indices in the text database we described in Subsection 8.2.2.

Image Extraction. In CATE, images are extracted from the Web using a Web search-engine. Selecting a relevant image for an entity is not so trivial as some entity names can be ambiguous. However, since this is not the main focus of our work, we use existing tools for resolving such cases such as [80].

8.2.4. Assigning Entities to Contexts

So far we have explained how to extract context objects and how we set their attribute values, and in this section we explain how we associate entities with the extracted contexts. Typically, each page in Wikipedia is annotated by users with the most relevant categories to which it belongs. This serves as the initial set of contexts for a given entity. However, Wikipedia categories alone offer incomplete and imprecise information. For instance, Gauss was included in the category `German_physicists` which is a very broad context. By reading the contents of Gauss' Wikipedia page, one can realize that the majority of his contributions in physics are in the area of `electro-magnetism`. We explain two approaches for assigning entities to additional contexts next.

Hyperlink-Based Assignment. This approach is based on hyperlink analysis. We assign an entity e to context C if the majority of entities that e links to, or the majority of entities that link to e belong to C as well. For example, consider the entity `Gauss` and the context `electro-magnetism` which is not one of Gauss' Wikipedia categories, and hence would not be considered as one of his contexts. However, it is very intuitive that Gauss *should* belong to this context if the article of Gauss contains many links to or from other entities that belong to the context `electro-magnetism`.

Attribute-Based Assignment. The hyperlink-based method would only identify contexts that are part of the Wikipedia-category hierarchy. However, we can combine attributes from different contexts in order to generate new contexts. For example, given that Gauss belongs to the categories `18th_century_mathematicians` and `German_mathematicians`, we can generate the new context with time `18th_century`, space `Germany` and topic `Mathematics`. In addition, we can utilize the categories hierarchy and external ontologies such as WordNet and Geo-Names to generate further contexts such as a context with time `18th_century`, space `Europe` and topic `Mathematics`. That is, we use the fact that Germany is part of Europe from such ontologies to create a more general context for Gauss. Actually, this is crucial for the inclusion of the entity French revolution in the timeline of Gauss because it is relevant (according to the Wikipedia contributors) not only to France but to whole Europe.

8.2.5. Ranking Model

We have three intermingling ranking problems: 1) ranking the contexts given an entity, 2) ranking entities given a context and an entity, and 3) ranking entities given a certain context only. For all three problems, we adopt a statistical-language-modeling approach [75], and we utilize our text database with its three indices to estimate the parameters of our model.

Basic Setting. We use the following notation. Each context C is associated with a set $E(C) = \{e_1, e_2, ..., e_n\}$ which is the set of entities that belong to context C. Additionally, each entity e_i is associated with a document $D(e_i)$ which is the Wikipedia article of e_i. For each such document, we construct a language model (LM) which is a probability distribution over all the entities in our knowledge base. We denote the parameters of the LMs as $P(e|D(e_i))$ which is the probability of generating the entity e from document $D(e_i)$. This probability is estimated using a maximum-likelihood estimator after employing Dirichlet smoothing as in most common LM approaches. The maximum-likelihood estimator can be computed in various ways and we experiment with the following two different methods.

Estimating the LM probabilities using links. In this method, we estimate $P(e|D(e_i))$ using a maximum-likelihood estimator after smoothing with the collection LM as follows:

$$P(e|D(e_i)) = \lambda \frac{lc(e; D(e_i))}{|D(e_i)|} + (1-\lambda)\frac{lc(e; Col)}{|Col|} \tag{8.1}$$

where $lc(e; D(e_i))$ is the number of links to e in the document $D(e_i)$ and $|D(e_i)|$ is the length of document $D(e_i)$ (i.e., the sum of all links to entities in $D(e_i)$). Col refers to the whole collection of documents and $lc(e; Col)$ and $|Col|$ is the total number of links to entity e and the total number of links to any entity in the whole collection respectively. Finally, the parameter λ is a smoothing parameter that controls the effect of smoothing and we set it as done in the Dirichlet smoothing method.

Estimating the LM probabilities using text. In this method, we utilize the text of the documents to estimate the probability $P(e|D(e_i))$. Assume that the entity e is composed of m words $\{w_1, w_2, ..., w_m\}$. For example, the entity Carl_Friedrich_Gauss, is composed of the words: {carl, friedrich, gauss}. We then assume independence between the words, and compute the probability $P(e|D(e_i))$ as follows:

$$P(e|D(e_i)) = \prod_{j=1}^{m} P(w_i|D(e_i)) \tag{8.2}$$

Similar to the previous model, we use a maximum-likelihood estimator smoothed with the collection LM to estimate the probability of generating the word w_j given the document $D(e_i)$ as follows:

$$P(w_j|D(e_i)) = \lambda \frac{tf(w_j; D(e_i))}{|D(e_i)|} + (1-\lambda)\frac{tf(w_j; Col)}{|Col|} \tag{8.3}$$

where $tf(w_j; D(e_i))$ is the term frequency or the number of occurrences of the word w_j in document $D(e_i)$ and $|D(e_i)|$ is the length of the document $D(e_i)$ (i.e., the sum of term frequencies of all the words in $D(e_i)$). Similarly, $tf(w_j; Col)$ is the term frequency of word w_j in the collection Col and $|Col|$ is the length of the collection. Again, the parameter λ is a smoothing parameter that is set as done in the Dirichlet smoothing method.

This method is motivated by the observation that there are many missing links in Wikipedia. To overcome this problem, we try to directly use the text in Wikipedia in order to estimate the parameters of the LMs. Note that having a link to an entity in some Wikipedia page is a much stronger evidence that this entity is relevant to that document as opposed to just being mentioned in the text. Also note that our two estimation methods can be easily integrated into one measure using a weighted combination for instance. Next, we explain how the language models of the entities can be used to solve our three ranking problems we described in the beginning of this subsection.

Ranking Contexts. Given an entity e and the set of contexts it belongs to, we rank the contexts based on their probabilities of generating the entity e. We estimate this probability by constructing a language-model for each context C as a mixture model over the documents of its entities as follows:

$$P(e|C) = \frac{1}{n}\sum_{i=1}^{n} P(e|D(e_i)) \tag{8.4}$$

where $P(e|D(e_i))$ is the probability of generating entity e given the document of entity e_i which can be estimated as described in the beginning of this subsection.

Ranking Entities Relevant to a Given Entity within a Context. The second ranking problem we have is to retrieve the most relevant entities given an entity e and a context C. We rank the entities based on their probability of being generated given e and C which we denote by $P(e'|C, e)$. To compute such probability, we construct a language model for C as a mixture model of the documents LMs of all entities in C. However, we only strict this to documents that *contain* e as well to accommodate for the conditioning over e. That is, we ignore the documents of entities that belong to C and do not contain e. To this end, let the set of documents of entities that belong to C and contain e be $\{D(e_1), D(e_2), ..., D(e_l)\}$. This way, the probability $P(e'|C, e)$ is equal to:

$$P(e'|C, e) = \frac{1}{l}\sum_{i=1}^{l} P(e'|D(e_i)) \tag{8.5}$$

Ranking Entities within a Context. The third and final ranking problem we deal with is ranking entities e' that belong to a certain context C. This can be easily done using the probability $P(e'|C)$ which is computed as described in Equation 8.4.

8.2.6. Extracting Events

Our event-description extractor takes as an input a query and a Wikipedia article, and retrieves the top-k most relevant events associated with the query from the article. The query can be either an entity name, a context name or both. An event is a text snippet, a timestamp and an image. The algorithm works as follows. First, we identify from the article all snippets S that contain time expressions. This can be easily done using tools such as [70]. Next, we rank the identified snippets based on their probability of generating the query $P(Q|S)$ (in line with our ranking model of contexts and entities) and output the highest-ranked k snippets.

Finally, to associate an image to the event which is used for illustration on the timeline, we identify the main entity the event is about, and then retrieve the image associated with that entity from our image database.

8.3. Summary

In this section, we presented two tools for knowledge exploration that utilize RDF knowledge bases as well as additional external sources in order to provide users with contextual information related to their information needs. The first tool is a document retrieval tool that retrieves a set of documents given a set of RDF triples. By doing so, it allows users to find additional contextual information about a set of selected RDF triples; information that is not present in the RDF knowledge base itself.

The second tool we presented in this chapter is an entity summarization tool that combines information from an RDF knowledge base with semi-structured and unstructured information and provides a comprehensive timeline for a given entity of interest. We believe that it is crucial to provide users with interactive tools that allows them to explore RDF knowledge bases, and to combine the information there with semi-structured and unstructured information present in external data sources in order to overcome the data incompleteness issue that most RDF knowledge bases suffer from and to provide users with a comprehensive summary of the information relevant to their queries.

Chapter 9. Conclusion

RDF is heavily used as a data representation format in scientific communities, social networks, news portals and other Web 2.0 domains. In this thesis, we have tackled research problems that are related to searching large RDF knowledge bases. This included supporting different search modes where queries are either structured, semi-structured (i.e., combining structured query-units with keywords) or plain keywords. For all these various search modes, we have developed a set of novel result-ranking models based on statistical language-models. In addition, we have also presented models and algorithms to support automatic query reformulation, result diversity and efficient top-k query processing. Finally, we have provided tools for knowledge exploration using RDF knowledge bases. Our contributions are well suited for RDF data as supported by the extensive experiments we conducted on real-world RDF datasets. Moreover, our models and algorithms can also be applied to a wide range of problems that deal with structured-information retrieval in general.

The set of models and algorithms we presented in this thesis can be further extended in various ways. For example, providing a natural-language question-answering system on top of RDF knowledge bases is a very promising direction of research. Even though we have presented a retrieval model to answer keyword queries over RDF data, we believe that natural-language questions are better suited for retrieving information from RDF knowledge bases. This is due to the fact that natural-language questions are typically richer in context, and can be analyzed using various tools in order to infer implicit structured queries that can be directly answered using the RDF knowledge base the questions are issued against.

Another possible direction of research is related to search personalization. Taking into consideration the users that search the knowledge base, their *preferences* as well as the *context* in which they search the knowledge base can further improve the quality of search results. In addition, considering user feedback, whether individual or collective over a community of users, has been shown to improve the performance of most retrieval models. However, gathering user feedback in the context of RDF search is a challenging task that needs large-scale studies on user's interactions with RDF knowledge bases. A slightly related problem is the problem of result presentation in the context of RDF search. It is crucial to design comprehensive result-presentation schemes that trade off conciseness with richness-in-context. Achieving this goal would increase the usability of RDF data and would allow models that leverage user feedback to be truly applicable.

Finally, metasearch over many RDF data sources, and even semi-structured and unstructured data sources, is also another possible interesting area to explore. Retrieving information from various data sources might be necessary for many information needs. Merging and ranking search results in this case is challenging and requires combining different criteria including the quality of the results and their relevance to the query, the authority of the data source from which the results are retrieved, result diversity, etc.

As of today, the amount of RDF data on the Web might still be small compared to the vast amount of information present in an unstructured form. However, we believe that as more and more RDF data becomes available and richly interconnected, the old dream of having the most comprehensive encyclopedia that can be used to precisely answer every information need will finally become a reality.

Bibliography

[1] Rakesh Agrawal, Sreenivas Gollapudi, Alan Halverson, and Samuel Ieong. Diversifying search results. In *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, WSDM '09, pages 5–14, New York, NY, USA, 2009. ACM.

[2] Sihem Amer-Yahia, Nick Koudas, Amélie Marian, Divesh Srivastava, and David Toman. Structure and content scoring for XML. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 361–372. VLDB Endowment, 2005.

[3] Sihem Amer-Yahia, Laks V. S. Lakshmanan, and Shashank Pandit. FleXPath: flexible structure and full-text querying for XML. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 83–94, New York, NY, USA, 2004. ACM.

[4] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. DBpedia: a nucleus for a web of open data. In *Proceedings of the 6th international The semantic web and 2nd Asian conference on Asian semantic web conference*, ISWC'07/ASWC'07, pages 722–735, Berlin, Heidelberg, 2007. Springer-Verlag.

[5] Bodo Billerbeck and Justin Zobel. When query expansion fails. In *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, SIGIR '03, pages 387–388, New York, NY, USA, 2003. ACM.

[6] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked Data - The Story So Far. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 2009.

[7] Roi Blanco, Peter Mika, and Hugo Zaragoza. Entity Search Track submission by Yahoo! Research Barcelona. http://km.aifb.kit.edu/ws/semsearch10/, 2010.

[8] John G. Breslin, Alexandre Passant, and Stefan Decker. *The Social Semantic Web*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[9] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Comput. Netw. ISDN Syst.*, 30:107–117, April 1998.

[10] Jaime Carbonell and Jade Goldstein. The use of MMR, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '98, pages 335–336, New York, NY, USA, 1998. ACM.

[11] Surajit Chaudhuri, Gautam Das, Vagelis Hristidis, and Gerhard Weikum. Probabilistic information retrieval approach for ranking of database query results. *ACM Trans. Database Syst.*, 31:1134–1168, September 2006.

[12] Harr Chen and David R. Karger. Less is more: probabilistic models for retrieving fewer relevant documents. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '06, pages 429–436, New York, NY, USA, 2006. ACM.

[13] Zhiyuan Chen and Tao Li. Addressing diverse user preferences in SQL-query-result navigation. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 641–652, New York, NY, USA, 2007. ACM.

[14] Tao Cheng, Xifeng Yan, and Kevin Chen-Chuan Chang. EntityRank: searching entities directly and holistically. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 387–398. VLDB Endowment, 2007.

[15] Charles L.A. Clarke, Maheedhar Kolla, Gordon V. Cormack, Olga Vechtomova, Azin Ashkan, Stefan Büttcher, and Ian MacKinnon. Novelty and diversity in information retrieval evaluation. In *Proceedings of the 31st annual*

international ACM SIGIR conference on Research and development in information retrieval, SIGIR '08, pages 659–666, New York, NY, USA, 2008. ACM.

[16] William W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, SIGMOD '98, pages 201–212, New York, NY, USA, 1998. ACM.

[17] Bruce Croft, Donald Metzler, and Trevor Strohman. *Search Engines: Information Retrieval in Practice*. Addison-Wesley Publishing Company, USA, 1st edition, 2009.

[18] Jeffrey Dalton, Roi Blanco, and Peter Mika. Coreference aware web object retrieval. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, CIKM '11, pages 211–220, New York, NY, USA, 2011. ACM.

[19] Elena Demidova, Peter Fankhauser, Xuan Zhou, and Wolfgang Nejdl. DivQ: diversification for keyword search over structured databases. In *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '10, pages 331–338, New York, NY, USA, 2010. ACM.

[20] Peter Dolog, Heiner Stuckenschmidt, Holger Wache, and Jörg Diederich. Relaxing RDF queries based on user and domain preferences. *J. Intell. Inf. Syst.*, 33:239–260, December 2009.

[21] Shady Elbassuoni and Roi Blanco. Keyword search over RDF graphs. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, CIKM '11, pages 237–242, New York, NY, USA, 2011. ACM.

[22] Shady Elbassuoni, Katja Hose, Steffen Metzger, and Ralf Schenkel. ROXXI: Reviving witness dOcuments to eXplore eXtracted Information. *Proc. VLDB Endow.*, 3:1589–1592, September 2010.

[23] Shady Elbassuoni, Maya Ramanath, Ralf Schenkel, Marcin Sydow, and Gerhard Weikum. Language-model-based ranking for queries on RDF-graphs.

In *Proceedings of the 18th ACM conference on Information and knowledge management*, CIKM '09, pages 977–986, New York, NY, USA, 2009. ACM.

[24] Shady Elbassuoni, Maya Ramanath, Ralf Schenkel, and Gerhard Weikum. Searching RDF Graphs with SPARQL and Keywords. *IEEE Data Eng. Bull.*, 33(1):16–24, 2010.

[25] Shady Elbassuoni, Maya Ramanath, and Gerhard Weikum. Language-model-based ranking in entity-relation graphs. In *Proceedings of the First International Workshop on Keyword Search on Structured Data*, KEYS '09, pages 43–44, New York, NY, USA, 2009. ACM.

[26] Shady Elbassuoni, Maya Ramanath, and Gerhard Weikum. Query relaxation for entity-relationship search. In *Proceedings of the 8th extended semantic web conference on The semanic web: research and applications - Volume Part II*, ESWC'11, pages 62–76, Berlin, Heidelberg, 2011. Springer-Verlag.

[27] Ronald Fagin. Combining fuzzy information from multiple systems. *J. Comput. Syst. Sci.*, 58:83–99, February 1999.

[28] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '01, pages 102–113, New York, NY, USA, 2001. ACM.

[29] Hui Fang and ChengXiang Zhai. Probabilistic models for expert finding. In *Proceedings of the 29th European conference on IR research*, ECIR'07, pages 418–430, Berlin, Heidelberg, 2007. Springer-Verlag.

[30] Freebase: A social database about things you know and love. www.w3.org/RDF/.

[31] GeoNames. http://www.geonames.org/.

[32] Konstantin Golenberg, Benny Kimelfeld, and Yehoshua Sagiv. Keyword proximity search in complex data graphs. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 927–940, New York, NY, USA, 2008. ACM.

[33] Sreenivas Gollapudi and Aneesh Sharma. An axiomatic approach for result diversification. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 381–390, New York, NY, USA, 2009. ACM.

[34] Lin Guo, Feng Shao, Chavdar Botev, and Jayavel Shanmugasundaram. XRANK: ranked keyword search over XML documents. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 16–27, New York, NY, USA, 2003. ACM.

[35] Djoerd Hiemstra. A Linguistically Motivated Probabilistic Model of Information Retrieval. In *Proceedings of the Second European Conference on Research and Advanced Technology for Digital Libraries*, ECDL '98, pages 569–584, London, UK, 1998. Springer-Verlag.

[36] Djoerd Hiemstra. *Using Language Models for Information Retrieval*. PhD thesis, University of Twente, Enschede, 2001.

[37] Djoerd Hiemstra. Statistical Language Models for Intelligent XML Retrieval. In *Intelligent Search on XML Data*, 2003.

[38] Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, and Gerhard Weikum. YAGO2: A Spatially and Temporally Enhanced Knowledge Base from Wikipedia. Research Report MPI-I-2010-5-007, Max-Planck-Institut fr Informatik, Saarbrücken, November 2010.

[39] Vagelis Hristidis, Heasoo Hwang, and Yannis Papakonstantinou. Authority-based keyword search in databases. *ACM Trans. Database Syst.*, 33:1:1–1:40, March 2008.

[40] Hai Huang, Chengfei Liu, and Xiaofang Zhou. Computing Relaxed Answers on RDF Databases. In *Proceedings of the 9th international conference on Web Information Systems Engineering*, WISE '08, pages 163–175, Berlin, Heidelberg, 2008. Springer-Verlag.

[41] Arvind Hulgeri and Charuta Nakhe. Keyword Searching and Browsing in Databases using BANKS. In *Proceedings of the 18th International Conference on Data Engineering*, ICDE '02, pages 431–, Washington, DC, USA, 2002. IEEE Computer Society.

[42] Carlos A. Hurtado, Alexandra Poulovassilis, and Peter T. Wood. Journal on data semantics x. pages 31–61, 2008.

[43] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Supporting top-K join queries in relational databases. pages 754–765, 2003.

[44] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40:11:1–11:58, October 2008.

[45] Kalervo Järvelin and Jaana Kekäläinen. IR evaluation methods for retrieving highly relevant documents. In *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '00, pages 41–48, New York, NY, USA, 2000. ACM.

[46] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S. Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional expansion for keyword search on graph databases. In *Proceedings of the 31st international conference on Very large data bases*, VLDB '05, pages 505–516. VLDB Endowment, 2005.

[47] Gjergji Kasneci, Maya Ramanath, Mauro Sozio, Fabian M. Suchanek, and Gerhard Weikum. STAR: Steiner-Tree Approximation in Relationship Graphs. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 868–879, Washington, DC, USA, 2009. IEEE Computer Society.

[48] Gjergji Kasneci, Fabian M. Suchanek, Georgiana Ifrim, Maya Ramanath, and Gerhard Weikum. NAGA: Searching and Ranking Knowledge. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 953–962, Washington, DC, USA, 2008. IEEE Computer Society.

[49] Jinyoung Kim, Xiaobing Xue, and W. Bruce Croft. A Probabilistic Retrieval Model for Semistructured Data. In *Proceedings of the 31th European Conference on IR Research on Advances in Information Retrieval*, ECIR '09, pages 228–239, Berlin, Heidelberg, 2009. Springer-Verlag.

[50] Oren Kurland and Eyal Krikon. The opposite of smoothing: a language model approach to ranking query-specific document clusters. volume 41, pages 367–395, USA, May 2011. AI Access Foundation.

[51] Oren Kurland and Lillian Lee. Corpus structure, language models, and ad hoc information retrieval. In *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '04, pages 194–201, New York, NY, USA, 2004. ACM.

[52] John Lafferty and Chengxiang Zhai. Document language models, query models, and risk minimization for information retrieval. In *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '01, pages 111–119, New York, NY, USA, 2001. ACM.

[53] Victor Lavrenko, Martin Choquette, and W. Bruce Croft. Cross-lingual relevance models. In *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '02, pages 175–182, New York, NY, USA, 2002. ACM.

[54] Dongwon Lee. *Query Relaxation for XML Model*. PhD thesis, University of California, Los Angeles (UCLA), 2002.

[55] Guoliang Li, Beng Chin Ooi, Jianhua Feng, Jianyong Wang, and Lizhu Zhou. EASE: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 903–914, New York, NY, USA, 2008. ACM.

[56] Linking Open Data. www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/.

[57] State of the LOD Cloud. www4.wiwiss.fu-berlin.de/lodcloud/state/.

[58] Vanessa Lopez, Victoria Uren, Enrico Motta, and Michele Pasin. AquaLog: An ontology-driven question answering system for organizational semantic intranets. *Web Semant.*, 5:72–105, June 2007.

[59] Steffen Metzger, Shady Elbassuoni, Katja Hose, and Ralf Schenkel. S3K: seeking statement-supporting top-K witnesses. In *Proceedings of the 20th*

ACM international conference on Information and knowledge management, CIKM '11, pages 37–46, New York, NY, USA, 2011. ACM.

[60] David R. H. Miller, Tim Leek, and Richard M. Schwartz. A hidden Markov model information retrieval system. In *Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '99, pages 214–221, New York, NY, USA, 1999. ACM.

[61] George A. Miller. WordNet: a lexical database for English. *Commun. ACM*, 38:39–41, November 1995.

[62] Ndapandula Nakashole, Martin Theobald, and Gerhard Weikum. Find your advisor: robust knowledge gathering from the web. In *Procceedings of the 13th International Workshop on the Web and Databases*, WebDB '10, pages 6:1–6:6, New York, NY, USA, 2010. ACM.

[63] Vivi Nastase and Michael Strube. Decoding wikipedia categories for knowledge acquisition. In *Proceedings of the 23rd national conference on Artificial intelligence - Volume 2*, AAAI'08, pages 1219–1224. AAAI Press, 2008.

[64] Apostol Natsev, Yuan-Chi Chang, John R. Smith, Chung-Sheng Li, and Jeffrey Scott Vitter. Supporting Incremental Join Queries on Ranked Inputs. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 281–290, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[65] Felix Naumann and Melanie Herschel. *An Introduction to Duplicate Detection*. Morgan and Claypool Publishers, 2010.

[66] Zaiqing Nie, Yunxiao Ma, Shuming Shi, Ji-Rong Wen, and Wei-Ying Ma. Web object retrieval. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 81–90, New York, NY, USA, 2007. ACM.

[67] Paul Ogilvie and Jamie Callan. Hierarchical language models for XML component retrieval. *Advances in XML Information Retrieval*, pages 224–237, 2005.

[68] Desislava Petkova and W. Bruce Croft. Hierarchical Language Models for Expert Finding in Enterprise Corpora. pages 599–608, 2006.

[69] Jay M. Ponte and W. Bruce Croft. A language modeling approach to information retrieval. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '98, pages 275–281, New York, NY, USA, 1998. ACM.

[70] J. Pustejovsky, J. Castano, R. Ingria, R. Sauri, R. Gauzauskas, A. Setzer, and G. Katz. TimeML: Robust Specification of Event and Temporal Expression in Text. *IWCS-5, Fifth International Workshop on Computational Semantics.*, 2003.

[71] W3C: Resource Description Framework (RDF). www.w3.org/RDF/.

[72] Stephen E. Robertson and Stephen Walker. Some simple effective approximations to the 2-Poisson model for probabilistic weighted retrieval. In *Proceedings of the 17th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '94, pages 232–241, New York, NY, USA, 1994. Springer-Verlag New York, Inc.

[73] Pavel Serdyukov and Djoerd Hiemstra. Modeling documents as mixtures of persons for expert finding. In *Proceedings of the IR research, 30th European conference on Advances in information retrieval*, ECIR'08, pages 309–320, Berlin, Heidelberg, 2008. Springer-Verlag.

[74] Luo Si, Rong Jin, Jamie Callan, and Paul Ogilvie. A language modeling framework for resource selection and results merging. In *Proceedings of the eleventh international conference on Information and knowledge management*, CIKM '02, pages 391–397, New York, NY, USA, 2002. ACM.

[75] Fei Song and W. Bruce Croft. A general language model for information retrieval. In *Proceedings of the eighth international conference on Information and knowledge management*, CIKM '99, pages 316–321, New York, NY, USA, 1999. ACM.

[76] W3C: SPARQL Query Language for RDF. www.w3.org/TR/rdf-sparql-query/.

[77] Steffen Staab and Rudi Studer. *Handbook on Ontologies*. Springer Publishing Company, Incorporated, 2nd edition, 2009.

[78] Fabian M. Suchanek, Gjergji Kasneci, and Gerhard Weikum. YAGO: A Large Ontology from Wikipedia and WordNet. *Web Semant.*, 6:203–217, September 2008.

[79] Fabian M. Suchanek, Mauro Sozio, and Gerhard Weikum. SOFIE: a self-organizing framework for information extraction. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 631–640, New York, NY, USA, 2009. ACM.

[80] Bilyana Taneva, Mouna Kacimi, and Gerhard Weikum. Gathering and ranking photos of named entities with high precision, high recall, and diversity. In *Proceedings of the third ACM international conference on Web search and data mining*, WSDM '10, pages 431–440, New York, NY, USA, 2010. ACM.

[81] Martin Theobald. *Efficient Top-k Query Processing for Text, Semistructured, and Structured Data*. PhD thesis, Universität des Saarlandes, May 2006.

[82] 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 *Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 405–416, Washington, DC, USA, 2009. IEEE Computer Society.

[83] Tran Anh Tuan, Shady Elbassuoni, Nicoleta Preda, and Gerhard Weikum. CATE: context-aware timeline for entity illustration. In *Proceedings of the 20th international conference companion on World wide web*, WWW '11, pages 269–272, New York, NY, USA, 2011. ACM.

[84] Reinier H. van Leuken, Lluis Garcia, Ximena Olivares, and Roelof van Zwol. Visual diversification of image search results. In *Proceedings of the 18th international conference on World wide web*, WWW '09, pages 341–350, New York, NY, USA, 2009. ACM.

[85] Erik Vee, Utkarsh Srivastava, Jayavel Shanmugasundaram, Prashant Bhat, and Sihem Amer Yahia. Efficient Computation of Diverse Query Results. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 228–236, Washington, DC, USA, 2008. IEEE Computer Society.

[86] Sebastian Wernicke. Efficient Detection of Network Motifs. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 3:347–359, October 2006.

[87] Ryen W. White, Ian Ruthven, and Joemon M. Jose. Finding relevant documents using top ranking sentences: an evaluation of two alternative schemes. In *Proceedings of the 25th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '02, pages 57–64, New York, NY, USA, 2002. ACM.

[88] Yu Xu and Yannis Papakonstantinou. Efficient keyword search for smallest LCAs in XML databases. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 527–538, New York, NY, USA, 2005. ACM.

[89] Cheng Xiang Zhai, William W. Cohen, and John Lafferty. Beyond independent relevance: methods and evaluation metrics for subtopic retrieval. In *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, SIGIR '03, pages 10–17, New York, NY, USA, 2003. ACM.

[90] Chengxiang Zhai. *Risk Minimization and Language Modeling in Text Retrieval*. PhD thesis, Carnegie Mellon University, 2002.

[91] ChengXiang Zhai. Statistical Language Models for Information Retrieval A Critical Review. *Found. Trends Inf. Retr.*, 2:137–213, March 2008.

[92] Xuan Zhou, Julien Gaugaz, Wolf-Tilo Balke, and Wolfgang Nejdl. Query relaxation using malleable schemas. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 545–556, New York, NY, USA, 2007. ACM.

Appendix A.

Negative Kullback-Leibler Divergence

In Chapter 3, we have developed a ranking model for triple pattern queries. Our ranking model assumes there exists a language model for the query and a language model for each result. Provided that the query consists of n triple patterns, the language models of the query and each result are probability distributions over all tuples of n triples. The score of a result R with respect to a given query Q is then set as the Kullback-Leibler (KL) divergence between the language model of query Q and the language model of result R which is computed as follows:

$$S(Q, R) = KL(Q||R) = \Sigma_{i=1}^{|KB|^n} P(T_i|Q) log \frac{P(T_i|Q)}{P(T_i|R)}$$

where $|KB|$ is the total number of triples in the knowledge base KB, $P(T_i|Q)$ is the probability of tuple T_i in the language model of query Q and $P(T_i|R)$ is the probability of tuple T_i in the language model of result R.

Folding the previous equation, we have:

$$S(Q, R) = \Sigma_{i=1}^{|KB|^n} P(T|Q) log P(T|Q) - \Sigma_{i=1}^{|KB|^n} P(T_i|Q) log P(T_i|R)$$

The first component in the previous equation does not depend on the result R and thus it can be omitted since it does not affect the ranking of the results. Hence,

$$S(Q, R) \propto -\Sigma_{i=1}^{|KB|^n} P(T_i|Q) log P(T_i|R)$$

Folding the summation over all tuples $T_1, T_2, ..., T_m$ (assuming that $m = |KB|^n$), we have:

$S(Q,R) \propto -[P(T_1|Q)\log P(T_1|R)+P(T_2|Q)\log P(T_2|Q)+.....+P(T_m|Q)\log P(T_m|R)]$

The probability $P(T_i|R)$ of tuple T_i in the language model of result R is estimated using a maximum likelihood estimator after smoothing with a collection (background) language model as follows:

$$P(T_i|R) = \beta\frac{c(T_i;R)}{|R|} + (1-\beta)P(T_i|C)$$

where $c(T_i;R)$ is the number of times the tuple T_i occur in R, $|R|$ is the length of R, $P(T_i|C)$ is probability of T_i in the language model of the collection C and the parameter β is a smoothing parameter.

Since R contains only one tuple, T_R, the probability $P(T_R|R)$ is then equal to: $\beta + (1-\beta)P(T_R|C)$ and the probability of $T_j \neq T_R$ is equal to: $(1-\beta)P(T_j|C)$.

This implies that:

$S(Q,R) \propto -[P(T_1|Q)\log((1-\beta)P(T_1|C))+....+ P(T_R|Q)\log(\beta+(1-\beta)P(T_R|C))+ .....+ P(T_m|Q)\log((1-\beta)P(T_m|C))]$

which is equivalent to, after adding and subtracting the term $P(T_R|Q)\log((1-\beta)P(T_R|C))$,

$S(Q,R) \propto -[P(T_1|Q)\log((1-\beta)P(T_1|C))+....+ P(T_R|Q)\log(\beta+(1-\beta)P(T_R|C))- P(T_R|Q)\log((1-\beta)P(T_R|C))+ P(T_R|Q)\log((1-\beta)P(T_R|C)) + ...+ P(T_m|Q)\log((1-\beta)P(T_m|C))]$

Regrouping the terms in the above equation, we have:

$S(Q,R) \propto -[P(T_R|Q)\log(\beta + (1-\beta)P(T_R|C)) - P(T_R|Q)\log((1-\beta)P(T_R|C))+ \Sigma_{i=1}^{m} P(T_i|Q)\log((1-\beta))P(T_i|C)]$

Since the summation $\Sigma_{i=1}^{m} P(T_i|Q)\log((1-\beta))P(T_i|C)$ does not depend on R (i.e., it is the same for every result), it can be omitted. Thus,

$$S(Q,R) \propto -P(T_R|Q)\log\frac{\beta + (1-\beta)P(T_R|C)}{(1-\beta)P(T_R|C)}$$

which is equivalent to:

$$S(Q,R) \propto -P(T_R|Q)\log(1 + \frac{\beta}{(1-\beta)P(T_R|C)})$$

Furthermore, if we assume that the collection language model is a uniform distribution over all tuples T_i, $P(T_R|C)$ becomes a constant and thus does not affect the score of the result R which now solely relies on the probability $P(T_R|Q)$

of tuple T_R in the query language model. In that case, we simply set the score of a result R to the probability of its tuple T_R in the query language model:

$$S(Q, R) \propto -P(T_R|Q)$$

Now, we can rank the results in descending order of the probabilities of their tuples in the query language model.

Appendix B. Evaluation Queries for Triple-Pattern Search

?x directed ?y;?x actedIn ?y
?x hasGenre Action;?x hasSuccessor ?y
?a1 isMarriedTo ?a2; ?a1 actedIn ?m;?a2 actedIn ?m
?a actedIn ?m;?a hasWonPrize ?x
?d hasWonPrize Academy_Award_for_Best_Director;?d directed ?m;?a actedIn ?m;?a hasWonPrize Academy_Award_for_Best_Actor
?d hasWonPrize Academy_Award;?d directed ?m;?a actedIn ?m;?a hasWonPrize Academy_Award
?m hasGenre Comedy
?m hasGenre Comedy;?a actedIn ?m;?a directed ?m
?m hasGenre Thriller;?d directed ?m
?m hasGenre Comedy;?a1 actedIn ?m;?a2 actedIn ?m
?a hasWonPrize Academy_Award_for_Best_Actress;?a actedIn ?m;?a diedOnDate ?t
?d directed ?m;?d hasWonPrize ?x;?m hasWonPrize ?y
?m1 producedIn Australia;?m1 hasWonPrize Academy_Award
?m1 hasGenre Family;?m1 hasProductionYear 1995;?a actedIn ?m1;?m2 hasGenre Comedy;?a actedIn ?m2
?x hasGenre Comedy[wedding]
?a actedIn ?m[spielberg];?a hasWonPrize ?x
?x directed ?y[true story];?x hasWonPrize ?z

?x hasProductionYear 2001;?x hasGenre Romance[paris]
?x actedIn ?y;?x hasWonPrize ?z;?y hasWonPrize Academy_Award
?x hasWonPrize ?y;?x actedIn ?m1;?x actedIn ?m2
?x hasWonPrize Academy_Award_for_Best_Actor;?y hasWonPrize Academy_Award_for_Best_Actress;?x actedIn ?m[love];?y actedIn ?m[relationship]
?a1 isMarriedTo ?a2; ?a1 actedIn ?m[boxing];?a2 actedIn ?m
?d hasWonPrize Academy_Award_for_Best_Director;?d directed ?m[new york];?a actedIn ?m;?a hasWonPrize Academy_Award_for_Best_Actor
?d hasWonPrize Academy_Award;?d directed ?m[soldiers];?a actedIn ?m;?a hasWonPrize Academy_Award
?m hasGenre Comedy[school friends]
?m hasGenre Comedy[police];?a actedIn ?m;?a directed ?m
?m hasGenre Thriller[gang];?d directed ?m
?m hasGenre Comedy[christmas];?a1 actedIn ?m;?a2 actedIn ?m
?a hasWonPrize Academy_Award_for_Best_Actress;?a actedIn ?m[paris];?a diedOnDate ?t
?d directed ?m[love];?d hasWonPrize ?p1;?m hasWonPrize ?p2
?m1 producedIn Australia;?m1 hasWonPrize Academy_Award[romance]
?m1 hasGenre Family[wedding];?m1 hasProductionYear 1995;?a actedIn ?m1;?m2 hasGenre Comedy[serial killer];?a actedIn ?m2

Table B.1.: Evaluation queries for the IMDB dataset

?x wrote ?y; ?y type Mystery_&_Thrillers
?x wrote ?y;?y hasTag 20th_Century;?y hasTag Classic
?x wrote ?y; ?y hasTag Fiction;?x wrote ?z;?z hasTag Non-fiction
?x wrote ?y; ?y hasTag Series
?x wrote ?y
?x wrote ?y; ?y hasTag Classic
?x wrote ?y;?y hasTag Magic;?y type Fiction
?x wrote ?y;?x wrote ?z;?x type Novelists
?x wrote ?y[teenagers]

?x wrote ?y[wizard]; ?y hasTag Sequel
?x wrote ?y[Spain]
?x wrote ?y[pulitzer]; ?y hasTag Classic
?x wrote ?y;?y hasTag Magic;?y type Fiction[award]
?x wrote ?y[civil war]; ?x type Novelists;?y hasTag Movie
?x wrote ?y;?x wrote ?z;?x type Novelists[nobel prize]
?x type Novelists[award];?x wrote ?z;?z type Nonfiction

Table B.2.: Evaluation queries for the LibraryThing dataset

Appendix C.

Evaluation Queries for Query Reformulation

Steven_Spielberg actedIn ?m
Mel_Gibson directed ?m
Britney_Spears actedIn ?m
Anthony_Quinn actedIn ?m;?m hasGenre War
?d directed ?m; ?d actedIn ?m;?m hasWonPrize Academy_Award
?m producedIn Australia;?m hasWonPrize Academy_Award
?a actedIn ?m;?a bornIn London
?m hasGenre Comedy;?m hasWonPrize Academy_Award
?a actedIn ?m;?m hasWonPrize Academy_Award;?a hasWonPrize Academy_Award_for_Best_Actor
?a actedIn ?m;?m hasProductionYear 1995;?a hasWonPrize Academy_Award_for_Best_Actor
?m hasGenre War;?m producedIn France
Woody_Allen directed ?m; ?m hasGenre Romance
Steven_Spielberg directed ?m; ?a actedIn ?m;?a hasWonPrize Academy_Award_for_Best_Actor
Woody_Allen directed ?m;Woody_Allen produced ?m; ?a actedIn ?m
Nicole_Kidman actedIn ?m; ?m hasGenre Thriller
?m1 hasGenre Mystery;?m1 hasPredecessor ?m2;?d1 directed ?m1;?d2 directed ?m2
?m hasGenre Musical;?m producedIn Italy
?a actedIn Saving_Private_Ryan

James_Cameron actedIn ?m
Ben_Stiller directed ?m
Justin_Timberlake actedIn ?m
Woody_Allen directed ?m;?a actedIn ?m;?a hasWonPrize Academy_Award_for_Best_Actress
?d produced ?m; ?d actedIn ?m;?m hasWonPrize Academy_Award
?m producedIn USA; ?m hasWonPrize Academy_Award; ?m hasGenre Romance
?a hasWonPrize Academy_Award_for_Best_Actor;?a originatesFrom New_York_City
?m hasGenre Animation; ?a actedIn ?m; ?a hasWonPrize Academy_Award_for_Best_Actor
?d directed ?m; ?m hasWonPrize Academy_Award;?m hasWonPrize Golden_Globe_Award; ?d hasWonPrize ?p
?a1 hasWonPrize Academy_Award_for_Best_Director;?a2 hasWonPrize Academy_Award_for_Best_Actress; ?a1 isMarriedTo ?a2
?m hasGenre Family; ?d directed ?m; ?d hasWonPrize Academy_Award_for_Best_Director; ?d produced ?m
Tim_Burton directed ?m; ?m hasGenre Family
Martin_Scorsese directed ?m; ?m hasGenre Romance
Steve_Martin directed ?m; ?a actedIn ?m
Tom_Cruise actedIn ?m; ?m hasGenre Drama
?m1 hasPredecessor ?m2; ?m1 producedIn UK; ?m2 producedIn USA
?a actedIn Titanic_(1997_film)
?x hasGenre Comedy[wedding]
?a actedIn ?m[spielberg];?a hasWonPrize Academy_Award_for_Best_Actor
?x directed ?y[true story];?x hasWonPrize Academy_Award_for_Best_Director
?x hasGenre Romance[paris]
?d hasWonPrize Academy_Award_for_Best_Director;?d directed ?m[soldiers]
?x actedIn ?m[school friends]; ?x type wordnet_singer_110599806
?m hasGenre Thriller[police];?a actedIn ?m; ?a produced ?m

?m hasGenre Action[gang];?a actedIn ?m
?m hasGenre Comedy;?a actedIn ?m[christmas]
?a hasWonPrize Academy_Award_for_Best_Actress;?a actedIn ?m[paris];?a diedOnDate ?t
?a actedIn ?m[love];?a hasWonPrize Academy_Award_for_Best_Actress
?a actedIn ?m[romance]; ?a hasWonPrize Academy_Award_for_Best_Actor
?m hasGenre Family[friends];?a actedIn ?m
?a actedIn ?m;?m hasGenre Comedy[serial killer]
James_Cameron directed ?m;?a actedIn ?m;?a hasWonPrize Academy_Award_for_Best_Actor
?a1 hasWonPrize Academy_Award_for_Best_Actor;?a2 hasWonPrize Academy_Award_for_Best_Actress; ?a1 isMarriedTo ?a2
Tom_Cruise actedIn ?m;?m hasGenre Romance
?a actedIn ?m;?m hasWonPrize Academy_Award;?a hasWonPrize Academy_Award_for_Best_Actress
?m hasGenre Musical;?m producedIn France
?d hasWonPrize Academy_Award_for_Best_Director;?d directed ?m[new york]

Table C.1.: Evaluation queries for the IMDB dataset

?w wrote ?b;?b hasTag Classic;?b hasTag Award
?w wrote ?b;?b hasTag British;?b hasTag Award
?w wrote ?b;?b hasTag Non-fiction;?b hasTag Pulitzer
?b type Science_Fiction_&_Fantasy; ?b hasTag Film
?w wrote ?b;?b hasTag Paris;?b hasTag Revolution
?w wrote ?b;?w type Children_Writers;?b hasTag Werewolves
?b hasTag Egypt;?b hasTag Magic
?w wrote Twilight
?w wrote ?b;?b type Non-fiction;?w type Novelists
?b hasTag Booker;?b hasTag Classic
?w wrote ?b;?b hasTag Booker;?w type Children_Writers
?w wrote ?b;?b hasTag Dark_Fantasy;?b hasTag Classic

?w wrote ?b;?b type Literature_&_Fiction;?b hasTag Award
?w wrote ?b;?b hasTag Revolution;?b type Nonfiction
?b hasTag Movie;?b hasTag Vampire
?b type Mystery_&_Thrillers;?b hasTag Funny
?w wrote ?b;?b hasTag Classic;?b hasTag Fantasy;?b hasTag Funny
?w wrote ?b;?b hasTag 19th_Century;?b type Romance
?w wrote ?b;?b hasTag Memoir;?b hasTag Germany
?b hasTag Non-fiction;?b hasTag Jewish
?w wrote ?b;?b hasTag Women_Writers;?b hasTag Award
?b hasTag French;?b hasTag Theatre
?w wrote ?b;?b hasTag Bush;?w type Historians
?w wrote ?b;?b hasTag Sexuality;?b hasTag Murder
?w wrote ?b;?b hasTag Relationships;?b hasTag Paris
?w wrote ?b;?b hasTag Middle_Ages;?b hasTag British
?w wrote Buffy_The_Vampire_Slayer
Cormac_McCarthy type ?t
?w wrote ?b;?b type Mystery_&_Thrillers;?b hasTag Middle_East
?b hasTag American_Civil_War;?b type Romance
?b hasTag Crime_Fiction;?b hasTag Favorites
?w wrote ?b1;?b1 hasTag Philosophy;?w wrote ?b2;?b2 hasTag Theatre
?b hasTag Greek;?b type Nonfiction
?w wrote ?b;?b hasTag Film;?b hasTag Horror
?x wrote ?y[teenagers]; ?y hasTag Film
?x wrote ?y[wizard]; ?y hasTag Children
?x wrote ?y[prize]; ?y hasTag Biography
?x wrote ?y[pulitzer]; ?y hasTag Slavery
?y hasTag Magic[award]; ?y type Science_Fiction_&_Fantasy
?x wrote ?y[revolution];?y hasTag Classic
?x wrote ?y[civil war]; ?y hasTag Film
?x wrote ?z[award];?z hasTag Non-fiction
?x wrote ?y[teenagers]; ?y hasTag Vampire
?x wrote ?y[wizard]; ?y hasTag School
?x wrote ?y[award]; ?y hasTag Jewish

?x wrote ?y[pulitzer]; ?y hasTag Holocaust
?x wrote ?y[friends];?y hasTag Teenage
?x wrote ?y[teenage];?y hasTag Humour
?b hasTag Civil_War;?b hasTag Non-fiction;?w wrote ?b2
?b hasTag France;?w wrote ?b;?b hasTag British;?b hasTag Classic
?w wrote ?b;?b hasTag Suspense;?b hasTag New_York
?w wrote ?b;?w type Historians;?b hasTag Memoir
?w wrote ?b;?w type American_Science_Fiction_Writers;?b hasTag School;?b hasTag Friendship
?x wrote ?y[nobel prize];?y hasTag British_Literature
?b hasTag Revolution;?b type Mystery_&_Thrillers

Table C.2.: Evaluation Queries for the LibraryThing Dataset

Appendix D.

Evaluation Queries for Keyword Search

Information need	**Query**
A movie in which Steven Spielberg acted	spielberg actor
A movie which Mel Gibson directed	mel gibson director
A movie which has genre War in which Anthony Quinn acted	anthony quinn war
A movie with genre Comedy which has won the Academy Award	comedy academy award
A movie with genre War which was produced in France	war france
A movie with genre Romance which was directed by Woody Allen	woody allen romance
A movie with genre Musical which was produced in Italy	musical italy
A movie with Genre Romance that was produced in the USA and has won an Academy Award	usa romance academy award
An actor that has won the Academy Award for Best Actor and the actor is related to New York City	new york academy award best actor
A movie of genre Animation and acted in by an actor that has won an Academy Award for Best Actor	animation academy award best actor
A movie that Tom Cruise acted in and has genre Drama	tom cruise drama

An actor that has won the Academy Award for Best Actor and acted in a movie that has something to do with Spielberg	spielberg academy award best actor
An actress that has won the Academy Award for Best Actress and has acted in a movie that has something to do with love	love academy award best actress
A movie of genre Musical produced in France	musical france
A director that has won the Academy Award for Best Director and has directed a movie that has something to do with New York	best director new york

Table D.1.: Evaluation queries for the IMDB dataset

Information need	**Query**
The author of a classic book that won an award	author classic award
The author of a british book that won an award	author british award
The author of a non-fiction book that won the Pultizer prize	author non-fiction pulitzer
The author of a book about Paris and revolutions	paris revolution author
A children's writer who has a wrote a book about werewolves	childern writer werewolves
The author of a literature and fiction book that won an award	author literature fiction award
The author of a classic fantasy funny book	classic fantasy funny author
The author of a memoir about Germany	author memoir germany
A woman writer who wrote a book that won an award	women writer award
The author of a book about relationships and murder	author relationship murder
The author of a british book about the Middle Ages	middle ages british author

A cirme fiction that has was tagged as favorite by the users	crime fiction favorite
A biography that won a prize	author biography prize
The author of a suspense novel about new york	suspense new york author
A Historian who wrote a memoir	historian memoir book

Table D.2.: Evaluation Queries for the LibraryThing Dataset

List of Figures

List of Tables