Query Processing on RDF data

Hai Huang
B.E. (HFUT, China)
M.E. (IIM of CAS, China)
A dissertation
submitted to
the Faculty of Information and Communication Technologies
Swinburne University of Technology
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

September 2011
Abstract

The RDF (Resource Description Framework) data model has been developed over a decade. It is designed as flexible representation of schema-relaxable or even schema-free information for the Semantic Web. Today, RDF has received more and more attention. Semantic Web style ontologies and knowledge bases with millions of facts from Wikipedia and other sources have been created and are available online. At this moment, there are more than 19 billion RDF triples on the web. With the rapid increase of RDF data, query processing on RDF data is a very important issue to realize the semantic web vision. Recently, the W3C RDF data access group has emphasized the importance of enhancing RDF query abilities to meet the real requirements. This thesis concentrates on some important issues including selectivity estimation, query relaxation and query evaluation on probabilistic RDF data.

Firstly, we study the problem of estimating the selectivity for SPARQL graph queries, which is crucial to query optimization. For an arbitrary SPARQL query represented as a composite graph pattern, we propose algorithms for maximally combining the statistics of chain paths and star paths that we have precomputed to estimate the overall selectivity of the graph pattern. The experiments validate the effectiveness of our methods. Secondly, a user is usually frustrated by no answers returned when he/she pose a query on a RDF database. For this purpose, we investigate how to relax a SPARQL query to obtain approximate
answers. We address two problems in efficient query relaxation. To ensure the quality of answers, we compute the similarities of relaxed queries with regard to the original query and use them to score the potential relevant answers. To obtain top-k approximate answers, we develop two efficient algorithms and conduct experiments to evaluate the efficiency of them. Thirdly, since there are a large number of RDF triples with probabilities, we study the problem of query evaluation for SPARQL queries on probabilistic RDF data. A general framework for supporting SPARQL queries on the probabilistic RDF database is presented. To enable query answering with some basic reasoning capabilities, we also consider transitive inference capability for RDF queries and propose an approximate algorithm for accelerating the query evaluation. The experimental results show that our method is effective and efficient.
Acknowledgment

Firstly, I would like to express my deep and sincere gratitude to my supervisor, Professor Chengfei Liu, for his patience, encouragement and help. He brought me into research world and taught me how to do solid research from critical reasoning to how to present an idea in a paper. His thoughtful and valuable comments have covered every corner in my study and benefited me throughout these years. I cannot finish my study without his guidance. I would also like to thank my associate supervisor Professor Xiaofang Zhou at University of Queensland for his help and insightful comments on my research work.

In addition to the mentors above, the members in Swinburne Web & Data Engineering group were always available for discussion. Especially, Dr. Jianxin Li has given me many helpful advice on my research over the three years. Dr. Rui Zhou has provided me a lot of valuable information to apply in both research and daily life. The other members Jiajie Xu, Sira Yongchareon, Saiful Islam have been great people to share ideas.

Dr Jinjun Chen and Dr. Bao Vo were kind to serve in my PhD review committee and provide helpful comments on my research work. Ms. Mandish Webb, Ms. Li Sim Loh and Ms. Gillian Foster have been very nice to help me with all kinds of administrative problems.

My parents have been always a source of energy for me in my life. Finally, but more importantly, I wish to thank my wife, Ms. Shujing Hao. When I met
obstacles, she was always there to support me; when I had a little success, she was also there to encourage me. I could not finish my PhD study without her.
Declaration

This thesis contains no material which has been accepted for the award of any other degree or diploma, except where due reference is made in the text of the thesis. To the best of my knowledge, this thesis contains no material previously published or written by another person except where due reference is made in the text of the thesis.

Name: Hai Huang
Signature: __________
Contents

Abstract ii

Acknowledgment iv

Declaration vi

1 Introduction 1

1.1 Background .................................................. 1

1.2 Problems and Challenges .................................. 4

1.2.1 Selectivity Estimation for RDF Queries ............... 5

1.2.2 RDF Query Relaxation .................................. 6

1.2.3 Query Evaluation on Probabilistic RDF Databases .... 7

1.3 Contributions of This Thesis ............................... 8

1.4 Structure of This Thesis .................................... 9

2 Technical Preliminaries 11

2.1 Resource Description Framework ........................ 11

2.1.1 RDF Model ............................................. 12

2.1.2 RDF Reification ....................................... 13

2.2 RDF Schema ................................................ 14

2.2.1 Classes and Properties ............................... 14
3 Related work

3.1 RDF Storage Systems ........................................ 19
3.2 Selectivity Estimation ...................................... 22
  3.2.1 Estimation for RDF a Single Triple Pattern .......... 23
  3.2.2 Estimation for RDF Joined Triple Patterns .......... 23
3.3 Query Relaxation ............................................. 27
3.4 Query Evaluation on Probabilistic data .................. 31

4 Selectivity Estimation for RDF Queries ................. 36

4.1 Motivation ................................................... 37
4.2 Estimation for Star Query Patterns ...................... 40
  4.2.1 Estimation for Star Patterns Using Bayesian Networks . 40
  4.2.2 Learning Bayesian Networks ............................ 46
4.3 Estimation for Chain Patterns ............................... 48
  4.3.1 Constructing the Histogram ............................ 49
  4.3.2 Estimation for Chain Patterns Using the Histogram ... 52
4.4 Estimation for Composite Graph Patterns ............... 53
  4.4.1 Decomposition of SPARQL Graph Patterns ............. 54
  4.4.2 Joining a Star Pattern with a Chain Pattern ........ 59
  4.4.3 Other Cases ............................................. 61
4.5 Experiments .................................................. 64
  4.5.1 Experimental Methodology .............................. 64
  4.5.2 Offline Evaluation .................................... 66
  4.5.3 Estimation Accuracy .................................. 68
  4.5.4 Online Running Time .................................. 71
5 Query Relaxation for RDF Queries

5.1 Motivation.............................................. 74
5.2 Background.............................................. 78
  5.2.1 Query Relaxation Model.................................. 78
  5.2.2 Problem Definition.................................... 79
5.3 Ranking Model.......................................... 79
  5.3.1 Similarity of Relaxed Triple Patterns......................... 79
  5.3.2 Similarity of Relaxed Queries................................ 83
  5.3.3 Score of Approximate Answers................................ 84
5.4 Best-First Relaxation Algorithm.............................. 84
  5.4.1 Basic Relaxation Algorithm................................ 84
  5.4.2 Pruning Unnecessary Relaxed Queries.......................... 88
5.5 Batch-based Relaxation Algorithm.............................. 91
  5.5.1 Predicting the Size of Batch................................. 92
  5.5.2 Batch-Based Relaxation Algorithm........................... 99
5.6 Experiments............................................ 101
5.7 Summary................................................. 106

6 Query Evaluation on Probabilistic RDF Data

6.1 Motivation.............................................. 108
6.2 Preliminary.............................................. 112
6.3 Query Evaluation........................................ 115
  6.3.1 Intentional Query Evaluation................................ 115
  6.3.2 Representing Answers of Queries as DNF..................... 117
  6.3.3 Computing the Probability of DNF Formula (BDD)............. 118
6.4 Evaluation of Atomic Query with Transitive Property......... 120
List of Tables

4.1 Statistics of datasets ................................................. 65
4.2 Some information about learning Bayesian networks ............ 67
4.3 Some information about chain histograms ....................... 68
4.4 Query set .................................................................... 71

5.1 Dataset information ..................................................... 101
5.2 Score of approximate answers ..................................... 106

6.1 Database instances with probabilities .............................. 110
6.2 Events of the results of \( \sigma(?X, \text{CauseOf}, \text{Fatigue}) \) ........ 110
6.3 Events of the results of \( \sigma(?X, \text{Associatedwith}^t, \text{Cough}) \) .... 111
6.4 Events of the results of \( \sigma(q_1) \bowtie \sigma(q_2) \) ............... 111
# List of Figures

1.1  The semantic Web stack. ..................................................... 3

2.1  An example of RDF data graph. ............................................ 13
2.2  RDF reification. ................................................................. 14
2.3  Example of a SPARQL query. ............................................... 17
2.4  A SPARQL graph pattern. ................................................... 18

3.1  Types of Jena tables. ......................................................... 21
3.2  A graph pattern. ................................................................. 26
3.3  Data Modeled by Malleable schema. ....................................... 29
3.4  An example of possible worlds. ............................................ 32

4.1  RDF SPARQL query ............................................................... 39
4.2  Star path and star query pattern .......................................... 43
4.3  Bayesian Network ................................................................. 44
4.4  Chain path and chain query pattern ..................................... 48
4.5  An example of the chain frequency table and the chain histogram.
    “?” indicates a variable. ....................................................... 50
4.6  Composite graph pattern ..................................................... 55
4.7  Join between chain patterns and star patterns, where “?Y” is the
    join node. ........................................................................ 59
4.8 Some information about query sets used in experiments. Query sets \( Qs_1, Qs_2, Qs_3 \) are used in the star query experiments. Query sets \( Qc_1, Qc_2, Qc_3 \) are used in the chain query experiments. Query sets \( Qsc_1, Qsc_2, Qsc_3, Qsc_4 \) are used in the composite query experiments.

4.9 Relative errors for star query patterns.

4.10 Relative errors for chain query patterns.

4.11 Relative errors for composite query patterns.

4.12 Online running time

5.1 Academic Staff ontology

5.2 A SPARQL query \( Q \) and its relaxed queries.

5.3 Ranking of the relaxed triple patterns in \( Q \) with their \( Sim \) scores

5.4 The relaxation graph of \( Q \)

5.5 The relaxation graph of \( Q \)

5.6 The relaxation step of query \( Q \)

5.7 The common ancestor of triple patterns and relaxed queries.

5.8 Queries used in the experiments

5.9 Performance of our algorithms (Top-10)

5.10 Performance of our algorithms (Top-50)

5.11 Performance of our algorithms (Top-150)

6.1 SPARQL query on a probabilistic RDF database

6.2 The BDD for DNF formula \((t_1 \land t_2) \lor (t_1 \land t_3 \land t_4)\)

6.3 A sample of transitive relation graph

6.4 Encodings of nodes

6.5 Approximate \( Find \) algorithm on the transitive relation graph

6.6 Queries used in the experiments
6.7 Running time for queries. ............................... 129
6.8 Running time for queries with increasing approximation \( \varepsilon \) . . . . . . . . 130
6.9 Search depth for queries with different approximation \( \varepsilon \) . . . . . . . . 130
6.10 Running time for queries with increasing the number of edges. . . 131
Chapter 1

Introduction

It is widely acknowledged that the Semantic Web will largely be built on RDF (Resource Description Framework). Recently, RDF is widely used in many fields (e.g., bioinformatics and grid). The increasing amount of RDF data has motivated the development of approaches for efficient RDF data management. The W3C RDF data access group has emphasized the importance of enhancing RDF query abilities to meet the real requirements. Thus efficient and effective RDF query processing is an important issue. Due to the lack of data structures and schema information, the conventional techniques are not sufficient for RDF query processing. This thesis focuses on solving some important problems of this issue.

1.1 Background

World Wide Web (WWW) as a system of interlinked hypertext documents was developed to be a pool of human knowledge, and human culture, which would allow collaborators in remote sites to share their ideas and all aspects of a common project. In the last two decades, WWW has experienced very rapid development.
Between 2005 and 2010, the number of Web users doubled, and was expected to surpass two billion in 2011.

However, there are some serious shortcomings in the hypertext paradigm. HTML markup is mainly concerned with layout, font size, colour and other presentational issues. Moreover, web pages increasingly use images, often including active links, to present information. For human users, it is easy to interpret the significance of such features, and thus understand the information being presented, but this may not be so easy for computes or “software agents”. Thus the required content becomes increasingly hard to locate using the search and browse paradigm. More complex tasks may be extremely difficult, or even impossible [MSZ01, VHS\textsuperscript{+}04].

The Semantic Web as the next generation of Web aims to make Web content more accessible to machines. It is a “web of data” that enables machines to understand the semantics, or meaning, of information on the Web. It extends the network of hyperlinked human-readable web pages by inserting machine-readable metadata about pages and how they are related to each other, enabling automated agents to access the Web more intelligently and perform tasks on behalf of users.

The Semantic Web comprises the standards and tools of XML, XML Schema, RDF, RDF Schema and OWL that are organized in the Semantic Web Stack \textsuperscript{1} shown in Figure 1.1:

- XML provides an elemental syntax for content structure within documents, yet associates no semantics with the meaning of the content contained within. XML is not at present a necessary component of Semantic Web technologies in most cases, as alternative syntaxes exists, such as Turtle.

\textsuperscript{1}WWW past & future, available at http://www.w3.org/2003/Talks/0922-rsoc-tbl/
• RDF is a simple language for expressing data models, which refer to objects ("resources") and their relationships. An RDF-based model can be represented in XML syntax.

• RDF Schema extends RDF and is a vocabulary for describing properties and classes of RDF-based resources, with semantics for generalized-hierarchies of such properties and classes.

• OWL adds more vocabulary for describing properties and classes: among others, relations between classes (e.g. disjointness), cardinality (e.g. "exactly one"), equality, richer typing of properties, characteristics of properties (e.g. symmetry), and enumerated classes.

• SPARQL is a protocol and query language for semantic web data sources.
• Rule Interchange Format (RIF) is the Rule Layer of the Semantic Web.

• Unifying Logic and Proof layers are undergoing active research.

As introduced above, RDF is a representation model for expressing data on the Semantic Web. In RDF model, all data items are represented in the form of \((\text{subject}, \text{predicate}, \text{object})\) triples, also known as \((\text{subject}, \text{property}, \text{value})\) triples. Now RDF is widely used in many fields and there are more than 19 billion triples available on the Web. It is widely acknowledged that the Semantic Web will be largely built on RDF.

With interest in the Semantic Web rising rapidly, the problem of SPARQL query processing over RDF data has received a lot of attention. Recently, the W3C RDF data access group has emphasized the importance of enhancing RDF query abilities to meet the real requirements. This thesis concentrates on some issues of SPARQL query processing over RDF data.

### 1.2 Problems and Challenges

Efficient and effective RDF query processing is an important issue for realizing Semantic Web vision. In RDF data, structures do not come in the shape of well-defined database schemas but in terms of semantic annotations that confirm to a schema called RDF schema (or RDFS ontology), which describes taxonomies of classes and properties. The conventional database techniques are not sufficient to make effective and efficient query processing possible on the Semantic Web. Thus our research focuses on new techniques for some problems about query processing on RDF data, which are summarized in the following sections.
1.2.1 Selectivity Estimation for RDF Queries

As we know, query optimization is crucial to efficient query processing. For cost-based query optimizers, they use estimated intermediate result size to choose the optimal query execution plan and thus selectivity estimation becomes one of core problems of query optimization. Mis-estimation of result size could cause an execution delay by a factor of 200 [NW08] or even more. Thus, selectivity estimation is a very important issue. In addition, selectivity estimation can also be used to approximately answer counting (aggregation) queries and to allow efficient load balancing for parallel join on multiprocessor systems.

The problem of selectivity estimation is well studied in the relational database. However, compared to the relational database with strict schemas and structures, RDF database has no such information. In addition, due to the fine-grained characteristic of RDF modeling, queries over RDF data contain a large number of joins. In this case, the join uniformity assumption could lead to the error by orders of magnitude. Our idea is to find frequent star and chain query patterns firstly. Then, we focus on these two basic patterns and propose to use Bayesian network and chain histogram respectively for estimating the selectivity of them. For estimating the selectivity of an arbitrary SPARQL graph pattern, we design algorithms for maximally combining the results of the star and chain patterns we have precomputed.

This problem can be described as:

**Selectivity Estimation for RDF Queries:** Given an RDF database $D$ and a SPARQL graph pattern $Q$, we estimate the selectivity $sel(Q)$ of $Q$, where $sel(Q)$ stands for the count of results in database $D$ satisfying $Q$. 
1.2.2 RDF Query Relaxation

As we know, users expect quality results and tend to pose a query with strict conditions over the database. However, RDF database has no strict schema information and users are not very clear about the contents of the database. Such a strict query often leads to the Few Answers Problem: the user query is too selective and the number of answers is not enough. In this case, users may undergo a “trial-and-error” process of changing the query conditions manually for obtaining approximate answers. In the worst case, they need to consider the combinations of all possible query conditions. Obviously, this is a time-consuming and frustrating task.

We address this problem by relaxing the query, i.e., weakening the query conditions automatically while preserving the intention of users as much as possible. The process of query relaxation can provide users with a means to automatically identify new queries that are related to the user query. These new queries can return generalized or neighborhood information by relaxing the original query conditions.

In contrast to conventional databases, ontology information can be used to relax a SPARQL query to more general queries for capturing potential approximate answers. However, there could be many directions for relaxation and the number of possible relaxed queries is large. Returning all approximate answers to users is neither time efficient nor necessary. To avoid overwhelming users with a big number of answers, it is desirable to obtain a ranked list of approximate answers. Thus we have to design an effective ranking function to score approximate answers according to their semantic relevance to the user query and design algorithms to obtain top-$k$ approximate answers over RDF databases efficiently.

This problem can be described as:

**RDF Query Relaxation:** Given an RDF database $D$ and a SPARQL query
Q, If there is no sufficient answers in \( D \) satisfying \( Q \), we relax the conditions of \( Q \) by replacing a class/property in \( Q \) to a variable (simple relaxation) or its super class/property (ontology relaxation) according to the ontology, and obtain the top-\( k \) approximate answers of \( Q \).

### 1.2.3 Query Evaluation on Probabilistic RDF Databases

RDF has been used as a knowledge representation model in a wide variety of domains, which are full of uncertainty. For example, in biology science, probabilistic links between concepts can be obtained from various prediction techniques. Thus, it is desirable to process and manage probabilistic RDF data. The core operation of queries on an RDF probabilistic database is computing the probability of the result to a query. Query evaluation on the probabilistic relational databases [FR97, GGH98, DS07] and probabilistic XML databases [AS06, HGS03] has been studied. However, this problem has not been addressed in RDF databases. In particular, we consider the transitive inference capability for RDF queries. For example, if we have two triples: \((\text{Pneumonia associatedWith Bronchitis})\) with probability 0.7 and \((\text{Bronchitis associatedWith Cough})\) with probability 0.6, then \( \text{Pneumonia} \) should also be associated with \( \text{Cough} \) with a certain probability. Thus we need to design a general framework for supporting SPARQL queries transitive inference capability on probabilistic RDF databases.

This problem can be described as:

**Query Evaluation on Probabilistic RDF Databases:** Given a probabilistic RDF database \( D \) with a finite set of probabilistic triples and a SPARQL query \( Q \) (containing transitive properties), \( \forall r \in \text{Answer}(Q) \) (answer set of \( Q \) in \( D \)), we compute the probability \( \Pr(r) \) efficiently.
1.3 Contributions of This Thesis

The main contributions of this thesis are as follows:

1. We address how to estimate the selectivity for SPARQL graph patterns. We study the star and chain patterns with correlated properties and propose two methods for estimating the selectivity of them respectively. For star query patterns, we construct Bayesian networks to compactly represent the joint probability distribution over values of correlated properties. And for chain query patterns, we build the chain histogram, which can obtain a good balance between the estimation accuracy and space cost. For an arbitrary SPARQL query represented as a composite graph pattern, we propose algorithms for maximally combining the statistics of chain paths and star paths that we have precomputed to estimate the overall selectivity of the graph pattern.

2. We investigate how to relax RDF queries to obtain approximate answers. We use information content to measure the semantic similarity between the corresponding terms used in the user query and a relaxed query, and combine similarity values of terms to score the relaxed query. The score of an approximate answer depends on the maximum score of the relaxed query it can match. To obtain top-$k$ approximate answers, we propose a relaxation algorithm, which is based on the best-first strategy and executes relaxed queries in the ranking order. To improve the performance of obtaining enough approximate answers, we devise a more efficient relaxation algorithm which executes the relaxed queries as a batch in order to avoid unnecessary execution cost. To predict the size of a batch, we study the problem of selectivity estimation in the context of RDF query relaxation.
and present the method to estimate the selectivity for a set of relaxed queries precisely.

3. We study the problem of evaluating the SPARQL query on Probabilistic RDF data. We propose a framework for supporting SPARQL query is proposed on probabilistic RDF databases based on the intentional query evaluation method. We also consider transitive inference capability on RDF instance data. We show that the Find operation for the atomic query with the transitive property can be formalized as the problem of computing path expressions on the transitive relation graph. In this case we propose an efficient algorithm which computing approximate path expressions efficiently.

1.4 Structure of This Thesis

The rest of this thesis is organized as follows:

- Chapter 2 introduces some knowledge about RDF model, RDF schema, inference on RDF data. The basic knowledge about the standard query language SPARQL on RDF data is also given in this chapter.

- Chapter 3 introduces some related works. The related works fall into the following categories: selectivity estimation, query relaxation and query evaluation on probabilistic RDF databases.

- Chapter 4 focuses on the problem of selectivity estimation for SPARQL graph patterns. We take into account the dependencies among properties in SPARQL graph patterns and propose a more accurate estimation model. Since star and chain query patterns are common in SPARQL graph patterns, we first focus on these two basic patterns and propose to use Bayesian
network and chain histogram respectively for estimating the selectivity of them. Then, for estimating the selectivity of an arbitrary SPARQL graph pattern, we design algorithms for maximally using the precomputed statistics of the star paths and chain paths. Finally, we conduct experiments on both synthetic and real-world datasets and the experimental results show the effectiveness and performance of our approach.

• Chapter 5 presents how to relax RDF queries to obtain approximate answers. We address two problems in query relaxation. First, to ensure the quality of answers, we compute the similarities of relaxed queries with regard to the original query and use them to score the potential relevant answers. Second, for obtaining top-k approximate answers, we develop two relaxation algorithms.

• Chapter 6 addresses to evaluate SPARQL queries on probabilistic RDF databases. A general framework for supporting SPARQL queries over probabilistic RDF data is proposed. In particular, we consider transitive inference capability for RDF instance data. We show that the find operation for an atomic query with the transitive property can be formalized as the problem of computing path expressions on the transitive relation graph and we also propose an approximate algorithm for computing path expressions efficiently.

• Chapter 7 concludes the thesis, summarizes the contributions, and provides a discussion of interesting directions for future works.
Chapter 2

Technical Preliminaries

In this chapter, we introduce the preliminaries of the thesis, including a set of basic concepts, some notations and a few necessary techniques. Firstly, we formally introduce discuss some knowledge about RDF (Resource Description Format). Then we describe some concepts about RDF schema. Finally, we introduce the standard query language SPARQL on the RDF data, which is recommended by W3C.

2.1 Resource Description Framework

The Resource Description Framework (RDF) is a language for representing information about resources in the World Wide Web. It is particularly intended for representing metadata about Web resources, such as the title, author, and modification date of a Web page, copyright and licensing information about a Web document, or the availability schedule for some shared resource. However, by generalizing the concept of a “Web resource”, RDF can also be used to represent information about things that can be identified on the Web, even when they cannot be directly retrieved on the Web.
RDF is intended for situations in which this information needs to be processed by applications, rather than being only displayed to people. RDF provides a common framework for expressing this information so it can be exchanged between applications without loss of meaning. Since it is a common framework, application designers can leverage the availability of common RDF parsers and processing tools. The ability to exchange information between different applications means that the information may be made available to applications other than those for which it was originally created.

2.1.1 RDF Model

RDF is based on the idea of identifying things using Web identifiers (called Uniform Resource Identifiers, or URIs), and describing resources in terms of simple properties and property values. This enables RDF to represent simple statements about resources as a graph of nodes and arcs representing the resources, and their properties and values.

**Definition (RDF Term):** An RDF term is an URI or a literal or a blank node.

**Definition (RDF Triple):** A triple \((s, p, o) \in (I \cup B) \times (I \cup B) \times (I \cup B \cup L)\) is called an RDF triple, where \(I\) is a set of IRIs (Internationalized URIs), \(B\) a set of blank nodes and \(L\) a set of literals. In the triple, \(s\) is called subject, \(p\) the property (or predicate), and \(o\) the object or property value.

**Definition (RDF Graph):** An RDF graph is a set of RDF triples. The set of nodes of an RDF graph is the set of subjects and objects of triples in the graph. The edges of an RDF graph are properties in RDF triples.

An example of RDF graph is shown in Figure 2.1.

**Definition (Graph Map):** Given RDF graphs \(G_1, G_2\), a map from \(G_1\) to \(G_2\) is a function \(\mu\) from the terms (URIs, blank nodes and literals) in \(G_1\) to the terms
in $G_2$, preserving IRIs and literals, such that for each triple $(s_1, p_1, o_1) \in G_1$, we have $(\mu(s_1), \mu(p_1), \mu(o_1)) \in G_2$.

**Definition (Simple Entailment):** An RDF graph $G_1$ *simply entails* $G_2$, denoted by $G_1 \models_{\text{simple}} G_2$, if and only if there is a map $u$ from $G_2$ to $G_1$.

### 2.1.2 RDF Reification

RDF applications sometimes need to describe other RDF statements using RDF, for instance, to record information about when statements were made, who made them, or other similar information (this is sometimes referred to as “provenance” information).

RDF provides a built-in vocabulary intended for describing RDF statements. A description of a statement using this vocabulary is called a reification of the statement. The RDF reification vocabulary consists of the type $\text{rdf} : \text{Statement}$, and the properties $\text{rdf} : \text{subject}$, $\text{rdf} : \text{predicate}$, and $\text{rdf} : \text{object}$. For example,
in Figure 2.2, the reification of statement \textit{exproducts : triple12345} describes that this statement is created by \textit{exstaff : 85740}.

\section{2.2 RDF Schema}

RDF provides a way to express simple statements about resources, using named properties and values. RDF schema RDF Schema (RDFS) is an extensible knowledge representation language, providing basic elements for the description of RDF vocabularies, intended to structure RDF resources.

RDF Schema does not provide a vocabulary of application-specific classes. Instead, it provides the facilities needed to describe such classes and properties, and to indicate which classes and properties are expected to be used together. In other words, RDF Schema provides a type system for RDF.

\subsection{2.2.1 Classes and Properties}

Resources may be divided into groups called classes. The members of a class are known as instances of the class. Classes are themselves resources. They are often identified by RDF URI References and may be described using RDF properties. The \textit{rdf : type} property may be used to state that a resource is an instance of a class.

RDF distinguishes between a class and the set of its instances. Associated
with each class is a set, called the class extension of the class, which is the set of the instances of the class. Two classes may have the same set of instances but are different classes. The group of resources that are RDF Schema classes is itself a class called \texttt{rdfs:Class}. There are some common classes defined in RDF schema:

\textbf{rdfs:Resource.} All things described by RDF are called resources, and are instances of the class \texttt{rdfs:Resource}. This is the class of everything. All other classes are subclasses of this class. \texttt{rdfs:Resource} is an instance of \texttt{rdfs:Class}.

\textbf{rdfs:Class.} This is the class of resources that are RDF classes.

\textbf{rdf:Property.} This is the class of RDF properties. \texttt{rdf:Property} is an instance of \texttt{rdfs:Class}.

Properties describe the relation between subject resources and object resources. There are some common properties defined in RDF schema:

\textbf{rdfs:range.} It is an instance of \texttt{rdf:Property} that is used to state that the values of a property are instances of one or more classes.

\textbf{rdfs:domain.} This is an instance of \texttt{rdf:Property} that is used to state that any resource that has a given property is an instance of one or more classes.

\textbf{rdf:type.} It is an instance of \texttt{rdf:Property} that is used to state that a resource is an instance of a class. A triple of the form (R \texttt{rdf:type} C) states R is an instance of class C. The \texttt{rdfs:domain} of \texttt{rdf:type} is \texttt{rdfs:Resource}. The \texttt{rdfs:range} of \texttt{rdf:type} is \texttt{rdfs:Class}.

\textbf{rdfs:subClassOf.} The property \texttt{rdfs:subClassOf} is an instance of \texttt{rdf:Property} that is used to state that all the instances of one class are instances of another. A triple of the form (C1 \texttt{rdfs:subClassOf} C2) states that C1 is a subclass of C2. The \texttt{rdfs:subClassOf} property is transitive. The \texttt{rdfs:domain} and \texttt{rdfs:range} of \texttt{rdfs:subClassOf} are both \texttt{rdfs:Class}.

\textbf{rdfs:subPropertyOf.} The property \texttt{rdfs:subPropertyOf} is an instance of \texttt{rdf:Property} that is used to state that all resources related to one property are
also related to another. A triple of the form \((P1 \ rdfs:subPropertyOf \ P2)\) states \(P1\) is a sub property of \(P2\). The \(rdfs:subPropertyOf\) property is transitive. The \(rdfs:domain\) and \(rdfs:range\) of \(rdfs:subPropertyOf\) are both \(rdf:Property\).

### 2.2.2 RDFS Entailment

Different with simple entailment, RDFS entailment considers RDFS ontologies. We assume that predicates of triples should be in the set \(\{type, domain, range, sp, sc\}\), where \(sc, sp\) are short for \(rdfs:subclassOf\) and \(rdfs:subpropertyOf\) respectively. Intuitively, this means that the ontology does not interfere with the semantics of the RDFS vocabulary.

We say that an ontology is acyclic if the subgraphs defined by \(sc\) and \(sp\) are acyclic. Acyclicity is considered good practice in modeling ontologies. In this thesis, we also use a notion of closure of an RDF graph \(G\), denoted by \(closure(G)\), which is the closure of \(G\) under the rules. RDFS entailment can be characterized as follows:

**Definition (RDFS Entailment):** An RDF graph \(G_1\) **RDFS entails** \(G_2\), denoted by \(G_1 \models_{\text{rule}} G_2\), if and only if \(G_2\) can be derived from \(G_1\) by iteratively applying rules in groups (A), (B), (C):

- **Group A (Subproperty)** (1) \(\frac{(a \ sp \ b)(b \ sp \ c)}{(a \ sp \ c)}\); (2) \(\frac{(a \ sp \ b)(x \ a \ y)}{(x \ b \ y)}\)
- **Group B (Subclass)** (3) \(\frac{(a \ sc \ b)(b \ sc \ c)}{(a \ sc \ c)}\); (4) \(\frac{(a \ sc \ b)(x \ type \ a)}{(x \ type \ b)}\)
- **Group B (Type)** (5) \(\frac{(a \ domain \ c)(x \ a \ y)}{(x \ type \ c)}\); (6) \(\frac{(a \ range \ d)(x \ a \ y)}{(y \ type \ d)}\)

\(^1\)RDF Semantics, W3C Recommendation: http://www.w3.org/TR/rdf-mt/
2.3 SPARQL Queries

SPARQL is a query language for querying RDF stored natively as RDF or viewed as RDF via middleware, which is recommended by W3C. SPARQL contains capabilities for querying required and optional graph patterns along with their conjunctions and disjunctions. SPARQL also supports extensible value testing and constraining queries by source RDF graph. The results of SPARQL queries can be triple sets or RDF graphs.

The SPARQL is a SQL-like language for querying RDF data. For expressing RDF graphs in the matching part of the query, TURTLE syntax is used. An example of a SPARQL query is shown in Figure 2.3. The first line defines namespace prefix, the last two lines use the prefix namespace to express a RDF graph to be matched. Identifiers beginning with question mark ? identify variables. In this query, we are looking for resource ?x participating in triples with predicates foaf:name and foaf:mbox and want the subjects of these triples.

*Definition (RDF Triple Pattern).* An RDF triple pattern is \((s, p, o) \in (I \cup V) \times (I \cup V) \times (I \cup V \cup L)\), where \(I\) is a set of IRIs, \(B\) a set of blank nodes and \(L\) a set of literals. \(V\) is a set of variables disjoint from the sets \(I\), \(B\) and \(L\).

*Definition (Basic Graph Pattern (BGP)).* A basic graph pattern \(Q =

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?name ?mbox
WHERE { ?x foaf:name ?name .
    }
```

Figure 2.3: Example of a SPARQL query.
Figure 2.4: A SPARQL graph pattern.

\((q_1, q_2, \ldots, q_n)\) is a set of triple patterns, where \(q_i\) is a triple pattern.

A SPARQL query can also be represented as a graph pattern. For example, the SPARQL query shown in Figure 2.3 is represented as a graph pattern shown in Figure 2.4, where the nodes are bound or unbound subjects and objects and the labels are properties.
Chapter 3

Related work

In this chapter, we review the state-of-the-art work. There has been a lot of interesting work in the field of query processing over RDF data recently. We first summarize the work about RDF storage system in Section 3.1. And then, we discuss the related research about selectivity estimation and RDF query relaxation in Section 3.2 and Section 3.3, respectively. Finally, we introduce the related work for query evaluation on probabilistic RDF in Section 3.4.

3.1 RDF Storage Systems

Many RDF systems map RDF triples onto relational tables. In RDFSuite [ACK+01], Sesame [KBF+03], all triples are stored in a single, giant table with generic attributes: subject, predicate, object. In C-Store based RDF engine of [AMH07] and Oracle’s RDF MATCH implementation [CDES05], triples are grouped by their predicate name, and all triples with the same predicate name are stored in the same property table. The extreme form of property tables with a separate table for each predicate name can be made more flexible, leading to a hybrid approach.
In Jena [WSKR03], triples are clustered by predicate names, based on predicates for the same entity class or co-occurrence in the workload; each cluster-property table contains the values for a small number of correlated predicates, and there may additionally be a “left-over” table for triples with infrequent predicates. A cluster-property table has a class-specific schema with attributes named after the corresponding RDF predicates.

Jena supports three kinds of property tables as well as a triple store (shown in Figure 3.1). A single-valued property table stores values for one or more properties. The subject column serves as the table key (the unique identifier for the row). Each property column may store an object value or be null. Thus, each row represents RDF statements which share the same subject. The property column value may not be null.

A multi-valued property table is used to store a single property that has a maximum cardinality greater than one (or that is unknown). The subject and object value serve as the table key, a compound key in this case. Each row in a multi-valued property table represents a single RDF statement.

Note that, in principle, the distinction between single-valued and multi-valued tables should not be needed since various strategies could be used to store multiple values for a property in a multi-column property table. However, these strategies greatly complicate query processing so they were not seriously considered for this first implementation. Consequently, an application must know in advance which properties are single or multi-valued. This is a loss of flexibility compared to a triple store. In the absence of any knowledge, many multi-valued tables may still be used. For some applications, this can still provide performance advantages over a single triple store.

There are other RDF storage systems (e.g., Hexastore [WKB08] and RDF-3X
CHAPTER 3. RELATED WORK

Figure 3.1: Types of Jena tables.

rieve [NW08, NW09]) being developed for RDF data storage and querying. RDF3x makes use of the fact that an RDF triple is a fixed 3-dimensional entity and stores all triples in a (compressed) clustered B+-tree, which allows the conversion of SPARQL patterns into range scans. In the pattern (literal1, literal2, ?x), the literals specify the common prefix and thus effectively a range scan. Each possible binding of ?x is found during a single scan over a moderate number of leaf pages.

RDF-3X creates all 6-way indexes (SPO, SOP, PSO, POS, OPS, OSP). Additionally it also builds aggregated indexes, each of which stores only two out of the three columns of a triple. More precisely, they store two entries (e.g., subject and object), and an aggregated count, namely, the number of occurrences of this pair in the full set of triples. This is done for each of the three possible pairs out of a triple and in each collation order (SP, PS, SO, OS, PO, OP), thus
adding another six indexes. RDF-3X also implements several other join optimization techniques like RDF specific Sideways-Information-Passing, selectivity estimation, merge-joins, and using bloom-filters for hash joins.

Along with these, BRAHMS [JK05] and GRIN [UPS07] focus more on path-like queries on RDF data, typically which cannot be expressed using existing SPARQL syntax.

### 3.2 Selectivity Estimation

Selectivity estimation is well studied in relational databases. Table-level synopses, such as, histograms [Ioa03, AC99, DGR01, PIHS96], wavelets [CGRS00, MVW98], sketches [DGGR02], and table samples [GGMS96, LNSS93] approximate the joint frequency distribution of values that appear in a single table. Hence, the optimizer estimates the selectivity of a join query by combining information from several synopses. Histograms and wavelets summarize the frequency matrix of the relation, and are most effective when the matrix contains relatively few contiguous regions comprising similar frequencies. These techniques, however, do not perform well for key/foreign-key joins, where the inclusion of a key attribute results in a frequency matrix with widely dispersed non-zero frequencies. Similarly, join synopses [AGPR99] enable accurate estimation by summarizing the combined join and value distribution across foreign-key joins of several tables. Unfortunately, in RDF database, there is no strict schema information. These these techniques cannot be used directly on RDF.

Recently, there is some related work on selectivity estimation for RDF query patterns.
CHAPTER 3. RELATED WORK

3.2.1 Estimation for RDF a Single Triple Pattern

Stocker et al. [SSB+08] estimate the selectivity of a single triple pattern \( t \) as
\[
\text{sel}(t) = \text{sel}(s) \times \text{sel}(p) \times \text{sel}(o),
\]
where \( \text{sel}(s) \), \( \text{sel}(p) \), \( \text{sel}(o) \) are the selectivity of subject, property and object. The selectivity of the subject \( \text{sel}(s) \) is estimated as \( \frac{1}{|\text{Res}|} \), where \( |\text{Res}| \) is the number of resources in the database. The selectivity of the property \( \text{sel}(p) \) is computed as \( \frac{|T_p|}{|T|} \), where \( |T_p| \) is the number of triples matching property \( p \) and \( |T| \) is the number of all triples in the database. For the object \( o \), they estimate the selectivity \( \text{sel}(o) \) using equal-width histograms which model the distribution of values for every property:
\[
\text{sel}(o) = \begin{cases} 
\frac{\text{freq}(h(p,o))}{|T_p|} & \text{if } p \text{ is bound}, \\
\sum_{p_i \in P} \frac{\text{freq}(h(p_i,o))}{|T_p|} & \text{otherwise},
\end{cases}
\]
where \( h(p, o) \) is the subset of the histogram for predicate \( p \) in which object \( o \) falls and \( \text{freq}(h(p,o)) \) is the frequency of subset \( h(p, o) \). \( P \) is the property set.

In [NW08], the authors harness the aggregated indexes (which are presented in Section 3.1), which are perfectly suited for the calculation of triple-pattern selectivities: for each literal or literal pair, the exact number of matching triples can be obtained in one index lookup. Unfortunately this is not sufficient for estimating join selectivities.

3.2.2 Estimation for RDF Joined Triple Patterns

For estimating the selectivity for joined triple patterns, [SSB+08] summarizes upper bound sizes for related predicates, where size denotes the result set size of a joined triple pattern with unbound subjects and objects. Given two related properties \( p_1, p_2 \) and their join relation (relation type), i.e. whether they define the same class \( C \) for their domains, domain/range, range/domain or both ranges,
the result set size of the corresponding basic graph pattern (BGP) is computed.

For instance, for \( p_1 \) and \( p_2 \) with a relation type \( SS \) (i.e. the two properties define the same class \( C \) for their domains), the corresponding BGP might be described by the two triple patterns \((?x p_1 ?y)\) and \((?x p_2 ?z)\). The returned size is used as an upper bound for the size of any BGP involving the two properties \( p_1 \) and \( p_2 \) with join relationship \( SS \). The number of entries in the summary is a quadratic function of the number of distinct predicates, more precisely \( f(n) = 4n^2 \), where \( n \) is the number of distinct predicates. For instance, for an ontology with 15 distinct predicates, the summary for joined triple patterns is of size 900.

The selectivity estimation of joined triple patterns is supported by the summary described above. Given the upper bound size \( S_P \) for a joined triple pattern \( P \), the selectivity of \( P \) is estimated as:

\[
sel(P) = \frac{S_P}{|T|^2}
\]

where \( |T|^2 \) denotes the square of the total number of triples in the RDF dataset. The square of the total number of triples equals the size of the Cartesian product of two triple patterns with pairwise distinct unbound components. The selectivity computed is corrected by a specific factor for joined triple patterns with bound subjects or objects. This factor is a function of the selectivities for the bound components (subject and object) of the triple patterns. For instance, if the pattern \( P \) is joined over the subjects by a variable and the first triple pattern has a bound object \( o \), i.e. \( P \) contains the triple pattern \((?x p_1 o)\) and \((?x p_2 ?y)\), the selectivity of \( P \) is estimated as:

\[
sel(P) = \frac{S_P}{|T|^2} \times sel(p_1, o)
\]
where \( \text{sel}(p_1, o) \) is the object selectivity of the first triple pattern in \( P \).

In [NW08], the authors propose two kinds of statistics. The first one, specialized histograms, is generic and can handle any kind of triple patterns and joins. Its disadvantage is that it assumes independence between predicates, which frequently does not hold in tightly coupled triple patterns. The second statistics therefore computes frequent join paths in the data, and gives more accurate predictions on these paths for large joins.

\[\text{Algorithm 3.1} \quad \text{MiningFrequentPath}\]

**Input:** RDF dataset \( D \);

**Output:** The \( k \) most frequent paths;

1. \( C_1 = \{ P_p | p \text{ is a predicate in } D \} \);
2. sort \( C_1 \), keep the \( k \) most frequent;
3. \( C = C_1, \ i = 1; \)
4. repeat
5. \( C_{i+1} = \phi; \)
6. for all \( p' \in C_i \), predicate \( p \) in the database do
7. \( \text{if top } k \text{ of } C \cup C_{i+1} \cup \{ P_{p'p} \} \text{ includes all subpaths of } p'p \text{ then} \)
8. \( C_{i+1} = C_{i+1} \cup P_{p'p} \)
9. \( \text{end if} \)
10. \( \text{if top } k \text{ of } C \cup C_{i+1} \cup \{ P_{pp'} \} \text{ includes all subpaths of } pp' \text{ then} \)
11. \( C_{i+1} = C_{i+1} \cup P_{pp'} \)
12. \( \text{end if} \)
13. end for
14. \( C = C \cup C_{i+1}, \text{ sort } C, \text{ keep the } k \text{ most frequent} \)
15. \( C_{i+1} = C_{i+1} \cap C, \ i = i + 1 \)
16. until \( C_i \neq \phi \)
17. return \( C \);

They compute the most frequent paths, i.e., the paths with the largest cardinalities, and materialize their result cardinalities. A path set \( \rho \) defined below is the result of the chain pattern with the sequence of properties \( p_1, ..., p_n \) as labels.
in its traversal:

\[
\rho = \text{select}(r_1, r_{n+1}) \\
\text{where}(r_1, p_1, r_2) \cdots (r_n, p_n, r_{n+1})
\]

Apriori-like frequent path mining algorithm is presented in Algorithm 3.1. Using this information they can exactly predict the join cardinality for the frequent paths that occur in a query.

For estimating the overall selectivity of an entire composite query, they combine the histograms with the frequent paths statistics. A long join chain with intermediate nodes that have triple patterns with object literals is decomposed into subchains of maximal lengths such that only their end nodes have triple patterns with literals.

Figure 3.2: A graph pattern.

For example, a query like in Figure 3.2 with attribute flavored predicates \(a_1, a_4, a_6\), literals \(v_1, v_4, v_6\), and relationship-flavored predicates \(p_1\) through \(p_5\) will be broken down into the subchains for \(p_1 - p_2 - p_3\) and for \(p_4 - p_5\) and the per-subject selections \(a_1 - v_1, a_4 - v_4,\) and \(a_6 - v_6\). They use the frequent paths statistics to estimate the selectivity of the two join subchains, and the histograms for selections. Then, in the absence of any other statistics, they assume that the different estimators are probabilistically independent, leading to a product formula with the per-subchain and per-selection estimates as factors. If instead
of a simple attribute-value selection like $(?x_6 a_6 v_6)$, it would invoke the estimator for the star pattern such as $(?x_6 a_6 u_6)(?x_6 b_6 v_6)(?x_6 c_6 w_6)$ with properties $a_6$, $b_6$, $c_6$ and corresponding object literals $u_6$, $v_6$, $w_6$ using the frequent star paths statistics. Then they combine them with the other estimates in the product form.

Notice that [SSB+08, NW08] both take the join uniformity assumption when estimating the selectivity of joined triple patterns. In our work, we avoid this assumption and propose a more accurate estimation model that takes correlations among properties into account.

### 3.3 Query Relaxation

The previous works [HPW06] generalize user RDF queries through RDF(s) entailment to capture potential approximate answers. They model relaxation as a combination of two types of relaxations, ontology relaxation and simple relaxation. Relaxation will be defined in the context of an ontology, denoted by $O$, and a set of variables, called fixed variables, denoted by $F$.

**Ontology relaxation on triple pattern:** This type of relaxation exploits RDFS entailment in the context of an ontology $O$. $\text{closure}(O)$ denotes the closure of $O$ under the rules defined in Section 2.2.2.

**Definition (Ontology relaxation):** Let $t$, $t'$ be triple patterns where $t \notin \text{closure}(O)$, $t' \notin \text{closure}(O)$. $t'$ is an ontology relaxation of $t$, denoted by $t \prec_{\text{onto}} t'$, if $(t \cup O) \models_{\text{rule}} t'$.

**Simple relaxation on triple pattern:** Simple relaxation consists only of replacements of terms of the original triple pattern (e.g., replacing a literal or URI with a variable).

**Definition (Simple relaxation):** Let $t_1$, $t_2$ be triple patterns, if $t_1 \models_{\text{simple}} t_2$ via a map $\mu$ that preserves $F$ (recall the notion of a map preserving a set
of variables from Section 2.1), then \( t_2 \) is a simple relaxation of \( t_1 \), denoted by \( t_1 \prec_{\text{simple}} t_2 \).

Now the notion of relaxation is defined as follows:

**Definition (Relaxation):** Triple pattern \( t_2 \) is a relaxation of triple pattern \( t_1 \), denoted \( t_1 \prec t_2 \), if one of the following hold: (i) \( t_1 \prec_{\text{onto}} t_2 \), (ii) \( t_1 \prec_{\text{simple}} t_2 \), or (iii) there exists a triple pattern \( t \) such that \( t_1 \prec t \) and \( t \prec t_2 \).

Using the relaxation framework, a user query can be relaxed to approximate answers. To guarantee the quality of answers, it is desirable to design ranking model for relaxed queries. However the ranking model proposed in [HPW06], is non-quantifiable and imprecise. The authors introduced the ranking model for relaxed queries: Given two relaxed queries of a user query, if the former is subsumed by the later, then the former relaxed query is better. This ranking model fails to rank relaxed queries when there does not exist subsume relationships among them. In [HLZ08], Huang at al. propose the method to rank the relaxed queries depending on their similarities to the user query. The similarities between relaxed queries and the user query are measured using ontology information without considering the real data distribution in the database. In this thesis, we propose a new ranking model which is quantifiable and fine-grained. The semantic similarity between the corresponding terms used in the user query and a relaxed query is measured based on their information contents in the database instead of distance-based measures. In addition, we also propose a more efficient relaxation algorithm which executes the relaxed queries as a batch in order to avoid unnecessary execution cost.

Recently, Zhou et. al. in [ZGBN07] propose a query relaxation method according to malleable schemas on a relational database system where they first utilize the duplicates in different data sets to discover the correlations within a malleable schema and then relax users queries based on the derived correlations.
Since each data instance uses only a subset of the attributes or relationships defined in a malleable schema, the predicates in a query have to be properly relaxed to retrieve all relevant results. In this work, such relaxation is achieved by extending the types of attributes or relationships. For example, Given a query $q_1$ based on the schema (please refer to Figure 3.3).

$$q_1 = \{ E_1 | E_1\text{.title} \ni \text{“XML”} \land E_1\text{.title} \ni \text{“Query”} \land E_1\text{.ISA - paper} \ni \text{“True”} \land E_1\text{.author} \ni E_2 \land E_2\text{.name} \ni \text{“Daniel”} \}$$

In order to retrieve more relevant results, $q_1$ can be relaxed by extending $E_2\text{.name} \ni \text{“Daniel”}$ to $E_2\text{.firstname} \ni \text{“Daniel”}$. By applying query relaxation, a query will be turned into a set of relaxed queries. However, users would like to see the results with the higher relevance. Therefore, the system in this work returns query results based on their probabilities of relevance. That means, given a query $q_0$ that could be relaxed to $q_1 \lor q_1 \lor \cdots q_n$, the system returns the results of the relaxed queries according to the probabilities $\Pr(q_0|q_1)$, $\Pr(q_0|q_2)$, $\cdots$, $\Pr(q_0|q_n)$, where $\Pr(q_0|q_j)$ represents the probability that a result of $q_j$ is also a
relevant result of \( q_0 \). As an example, \( q_0 = \{ E | A \ni a \land B \ni b \} \) is relaxed to \( q_1 \lor q_2 \), where \( q_1 = \{ E | A_1 \ni a \land B_1 \ni b \} \) and \( q_2 = \{ E | A_2 \ni a \land B_2 \ni b \} \). If the system can know that \( \Pr(A \ni a \land B \ni b | A_1 \ni a \land B_1 \ni b) < \Pr(A \ni a \land B \ni b | A_2 \ni a \land B_2 \ni b) \), then the system will return the results of \( q_2 \) prior to the results of \( q_1 \) because \( q_2 \) will retrieve more relevant results than \( q_1 \).

The authors assume that the entity-relationship data are stored in relational database. A query using malleable schema will be relaxed to multiple queries that are executed on different columns or tables. The major performance consideration is to find a plan that executes as less queries as possible to retrieve sufficient relevant results. The optimal plan is to execute relaxed queries in a sequence based on the expected precisions of their result sets. However, sometimes it is infeasible to evaluate all relaxed queries. In practice, it is more desirable to evaluate the relaxed queries in the order of their precisions until users obtain enough results. To achieve this, the authors exploit the relationship between the relaxed queries. In [ZGBN07], the concept of child query is defined as: Given two relaxed queries \( Q_1, Q_2 \) of \( Q \), \( Q_2 \) is a child query of \( Q_1 \) if \( Q_1 \) can be turned into \( Q_2 \) by substituting an attribute \( A_1 \) in \( Q_1 \) with its child attribute \( A_2 \) such that \( \Pr(A_1|A_0) > \Pr(A_2|A_0) \) and \( A_0 \) is the original attribute in \( Q \). They show that a relaxed query always yields better precision than its child queries, so that it should always be evaluated prior to its child queries.

Efforts have been made to the problem of query relaxation on XML data [AYCS02, BSAY04, LLYZ10]. Amer-Yahia et al. [AYCS02, BSAY04] propose to compute approximate answers for weighted patterns by encoding the relaxed queries in join evaluation plans. In [LLYZ10], Liu at al. propose an adaptive relaxation approach which relases a query against different data sources differently based on their conformed schemas. Note that the techniques of approximate XML query matching are mainly based on structure relaxation and can not be
used to handle query relaxation on RDF data directly.

Cooperative query answering is to provide an intelligent interface to a relational database system by facilitating cooperative query handling with query relaxation mechanisms and producing approximate answers. Such relaxation is usually based on rules. The rule-based approaches [Mot86, Mot90, Gaa97, Mus04] represent the semantic relationship and integrity constraints among data values using first-order logic predicates. The query is answered through conflict resolution and inference mechanisms, and query relaxation is carried out by coordinating the integrity constraints. In contrast, in our work the user query is generalized to more general ones according to the RDFs ontology to capture potential approximate answers. Some techniques such as Top-k processing are also related to obtaining approximate answers. Top-k processing techniques can be classified according to the query model they assume. [CK97, BCG02, CL02, FLN01, HKP01] assume a selection query model, where scores are attached directly to base tuples. Other techniques [NCS+01, IAE03] assume a join query model, where scores are computed over join results. [LCI06] assumes an aggregate query model, where we are interested in ranking groups of tuples. In this thesis, we focus on SPARQL queries and scores are attached directly to the relaxed queries which are different from the above approaches.

3.4 Query Evaluation on Probabilistic data

Many systems adopt the possible worlds semantics. The possible worlds semantics, originally put forward by Kripke for modal logics, is commonly used for representing knowledge with uncertainties. Halpern and Baccus [FH88, BGHK96] have shown the use of possible worlds semantics to assign degrees of beliefs to
statements based of the probabilities in the knowledge base. In [DS04], the authors also use possible worlds semantics in the probabilistic relational database as follows.

**Possible Worlds.** Fix a relational schema $S$, consisting of relation names $R_1, R_2, \cdots, R_m$, a set of attributes $\text{Attr}(R_i)$ and a key $\text{Key}(R_i) \subseteq \text{Attr}(R_i)$ for each $i = [1, m]$. A probabilistic database over schema $S$ is a pair $(W,P)$ where $W = \{W_1, \cdots, W_n\}$ is a set of database instances over $S$, and $P : W \rightarrow [0,1]$ is a probability distribution (i.e. $\sum_{j=1}^{n} P(W_j) = 1$). Each instance $W_j$ for which $P(W_j) > 0$ is called a possible world. A example of possible worlds with probability is shown in Figure 3.4.

<table>
<thead>
<tr>
<th>No.</th>
<th>W</th>
<th>$P(W)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$t_1$</td>
<td>$p_1(1-p_2)(1-p_3)$</td>
</tr>
<tr>
<td>2</td>
<td>$t_2$</td>
<td>$p_2(1-p_1)(1-p_3)$</td>
</tr>
<tr>
<td>3</td>
<td>$t_2t_1$</td>
<td>$p_1p_2(1-p_3)$</td>
</tr>
<tr>
<td>4</td>
<td>$t_3$</td>
<td>$p_3(1-p_1)(1-p_2)$</td>
</tr>
<tr>
<td>5</td>
<td>$t_3t_1$</td>
<td>$p_3p_1(1-p_2)$</td>
</tr>
<tr>
<td>6</td>
<td>$t_3t_2$</td>
<td>$p_3p_2(1-p_1)$</td>
</tr>
<tr>
<td>7</td>
<td>$t_3t_2t_1$</td>
<td>$p_3p_2$</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>$(1-p_2)(1-p_1)(1-p_2)$</td>
</tr>
</tbody>
</table>

**Table**

<table>
<thead>
<tr>
<th>Id</th>
<th>Year</th>
<th>Prob</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1$</td>
<td>m_1</td>
<td>2007</td>
</tr>
<tr>
<td>$t_2$</td>
<td>m_2</td>
<td>2009</td>
</tr>
<tr>
<td>$t_3$</td>
<td>m_3</td>
<td>2008</td>
</tr>
</tbody>
</table>

Figure 3.4: An example of possible worlds.

It indicates that the exact state of the database is uncertain: the database has several possible instances, and for each instance it has a probability.

Let $(W,P)$ be a probabilistic database and let $t_1, t_2, \cdots, t_n$ be all the tuples in all possible worlds. We interpret each tuple as a boolean propositional variable, and each possible world $W$ as a truth assignment to these propositional variables, as follows: $t_i = \text{true}$ if $t_i \in W$, and $t_i = \text{false}$ if $t_i \notin W$. Consider now a DNF formula $E$ over tuples: clearly $E$ is true in some worlds and false in others. Define
its probability $P(E)$ to be the sum of $P(W)$ for all worlds $W$ where $E$ is true. For example, the expression $E = (t_1 \lor t_3) \land t_2$ is true in the possible worlds $W_2, W_3, W_5, W_6, W_7$ and its probability is thus $P(E) = P(W_2) + P(W_3) + P(W_5) + P(W_6) + P(W_7)$.

Though there has been extensive work on probabilities in AI, relatively little work has been done on probabilistic databases. There are probabilistic frameworks [CP87, BGMP92, KL83] proposed for databases, but each makes simplified assumptions for getting around the problem of high query evaluation complexity that lessens their applicability. Fuhr and Rolleke [FR97] define probabilistic $NF^2$ relations and introduce the intensional semantics for query evaluation. As we saw, this is correct, but impractical. Many of these works specialize on logic programming in deductive databases. Ng and Subrahmaniam [NS92] extend deductive databases with probabilities and give fixed point semantics to logic programs annotated with probabilities, but they use absolute ignorance to combine event probabilities.

Query evaluation on the probabilistic relational database [GGH98, DS07] and probabilistic [AS06, HGS03] has been studied. In [DS04, DS07], the authors introduce a new semantics for database queries that supports uncertain matches and ranked results, by combining the probabilistic relational algebra [FR97] and models for belief [BGHK96]. Given an SQL query $q$ with uncertain predicates, they start by assigning a probability to each tuple in the input probabilistic database $D^p$ according to how well it matches the uncertain predicates. Then compute the probabilistic rankings $q^{rank}(D^p)$.

They adopt extensional query evaluation method which computes probabilities rather than complex events. This is more efficient, since it involves manipulating real numbers rather than event expressions. $Pr_p(t) \in [0,1]$ is the probability score for each tuple $t$, by induction on the structure of the query plan.
\[ Pr_{\sigma^e(p)}(t) = \begin{cases} Pr_p(t), & \text{if } c(t) \text{ is true,} \\ 0 & \text{otherwise.} \end{cases} \]

\[ Pr_{\Pi^e_A(p)}(t) = 1 - \prod_{t' : \Pi(c(t)) = t} (1 - Pr_p(t)) \]

\[ Pr_p(t) \times^e Pr_{p'}(t') = Pr_p(t) \times Pr_{p'}(t') \]

Here, \( \sigma^e \) propagates the tuples probabilities from the input to the output; \( \prod^e \) computes the probability of a tuples \( t \) as \( 1 - (1 - p_1)(1 - p_2) \cdots (1 - p_n) \), where \( p_1, \ldots, p_n \) are the probabilities of all input tuples that project to \( t \), while \( \times^e \) computes the probability of each tuple \( (t, t') \) as \( p \times p' \).

Thus, \( p^e(D^p) \) is an extensional probabilistic relation, which they call the extensional semantics of the plan \( p \). If \( p^e(D^p) = q^{\text{rank}}(D^p) \), then simply execute the plan under the extensional semantics. But, unfortunately, this is not always the case so it needs to find a plan for which the extensional semantics is correct.

**Definition (Query plan):** Given a schema \( R^p \), a plan \( p \) for a query \( q \) is safe if \( p^e(D^p) = q^{\text{rank}}(D^p) \) for all \( D^p \) of that schema.

The following theorem is used to test whether plan \( p \) is a safe plan:

**Theorem.** Let \( q, q' \) be conjunctive queries and \( PRel(q) \) refers to the probabilistic relation names in \( q \),

- \( \sigma^e \) is always safe in \( \sigma(c(q)) \).

- \( \times^e \) is always safe in \( q \times q' \).

- \( \Pi^e_{A_1, \ldots, A_k} \) is safe in \( \Pi_{A_1, \ldots, A_k}(q) \) iff for every \( R^p \in PRel(q) \) the following can be inferred from \( \Gamma^p(q) : A_1, \ldots, A_k, R_p.E \rightarrow \text{Head}(q) \)

A plan \( p \) is safe iff all operators are safe.
There is also some research about uncertainty in logic and Web languages. In terms of representation, Fukushige [Fuk05] provides a comprehensive method for representing probabilistic relations in RDF. In terms of probabilistic extensions to logic and ontology languages, Koller et. al [KLP97] and Giugno et al. [GL02] propose probabilistic extensions of description logics. They present P-CLASSIC, a probabilistic version of the description logic CLASSIC. In addition to terminological knowledge, the language utilizes Bayesian networks to express uncertainty about the basic properties of an individual. Costa et al. [dCLL05] extend OWL with uncertainty based on first-order Bayesian logic. PR-OWL is an extension that enables OWL ontologies to represent complex Bayesian probabilistic models in a way that is flexible enough to be used by diverse Bayesian probabilistic tools based on different probabilistic technologies. That level of flexibility can only be achieved using the underlying semantics of first-order Bayesian logic, which is not a part of the standard OWL semantics and abstract syntax. Therefore, it seems clear that PR-OWL can only be realized via extending the semantics and abstract syntax of OWL. However, in order to make use of those extensions, it is necessary to develop new tools supporting the extended syntax and implied semantics of each extension. In [USM06], the authors provide a syntax as well as a model theoretic and fixpoint semantics for acyclic pRDF theories. They also provide algorithms to efficiently answer queries posed to pRDF ontologies.
Chapter 4

Selectivity Estimation for RDF Queries

A fundamental problem related to RDF query processing is selectivity estimation, which is crucial to RDF query optimization for determining a join order of RDF triple patterns. In this chapter, we focus research on selectivity estimation for SPARQL graph patterns. The previous work takes the join uniformity assumption when estimating the joined triple patterns. This assumption would lead to highly inaccurate estimations in the cases where properties in SPARQL graph patterns are correlated. We take into account the dependencies among properties in SPARQL graph patterns and propose a more accurate estimation model. Due to the fine-grained characteristic of RDF modeling, star and chain query patterns are common in SPARQL graph patterns. Thus, we first focus on these two basic patterns and propose to use Bayesian network and chain histogram respectively for estimating the selectivity of them. Then, for estimating the selectivity of an arbitrary SPARQL graph pattern, we design algorithms for maximally using the precomputed statistics of the star paths and chain paths. The experiments on both synthetic and real-world datasets show that our method
outperforms existing approaches in accuracy.

4.1 Motivation

The Resource Description Framework (RDF) is a standard format for encoding machine-readable information on the Semantic Web. Recently, more and more data is being stored in RDF format. RDF data is a set of triples and each triple called statement is of the form \((subject, property, object)\). RDF data can also be represented as a graph with nodes representing resources or their property values and labeled arcs representing properties. This data representation is general and flexible. Almost any kind of data can be represented in this format. However, this fine-grained model leads to queries on RDF data with a large number of joins, which is an inherent characteristic of querying RDF data [NW09].

Since the use of RDF to represent data has grown dramatically over the last few years, query processing on RDF data becomes an important issue in realizing the semantic web vision. Some query languages such as SPARQL \(^1\) have been developed. As we know, accurate estimation of the result size of queries is crucial to query processing. Cost-based query optimizers use estimated intermediate result size to choose the optimal query execution plan. In addition, selectivity estimation can also be used to approximately answer counting (aggregation) queries and to allow efficient load balancing for parallel join on multiprocessor systems.

As a SPARQL query has a large number of joins, estimating precisely the joined triple pattern is very important. Some work has been done in this area. In [SSB+08, NW08] the join uniformity assumption is made when estimating the joined triple patterns with \textit{bound} subjects or objects (i.e., the subjects or objects are concrete values). They assume that each triple satisfying a triple pattern is

\(^1\)SPARQL Query Language: http://www.w3.org/TR/rdf-sparql-query/
equally likely to join with the triples satisfying the other triple pattern. However, this assumption does not hold in many cases. And when the data are inconsistent with this assumption, it could cause a highly inaccurate estimation.

For example, in Figure 4.1, a SPARQL query is posed on an RDF database, which retrieves academic staff members and the courses they teach with some conditions. Suppose we want to estimate the selectivity \( \text{sel}(t_1 \Join t_3) \) of two joined triple patterns \( t_1 \) and \( t_3 \). It would overestimate the result size \( \text{sel}(t_1 \Join t_3) \) using the formula (1) proposed in [SSB+08] as follows:

\[
\text{sel}(t_1 \Join t_3) = \frac{\text{SP}}{|T|^2} \times \text{sel}(\text{Income, } ' \leq 70K') \\
\times \text{sel}(\text{Position, } '\text{Prof}')
\]

where \( \text{SP} \) is the result upper bound of the joined triple patterns \((?Z, \text{Income}, ?W)(?Z, \text{Position}, ?Y)\) and \(|T|\) is the number of triples in the database. \( \text{sel}(\text{Income, ' \leq 70K'}) \) and \( \text{sel}(\text{Position, 'Prof.'}) \) are the object selectivities of \( t_1 \) and \( t_3 \). It assumes that each triple satisfying triple pattern \( t_1 \) is equally likely to join with triples satisfying triple pattern \( t_3 \). But in fact, the triple matching \( t_3 \) which indicates that the person is a professor, who is supposed to have higher income ('\( > 70k \)'). Thus, the triples matching \( t_1 \) are less likely to join with triples matching \( t_3 \).

There are two observations from this example. First, we can observe that the join uniformity assumption is not applicable in those cases where properties have correlations (dependencies) with each other. In fact, this assumption is rarely satisfied in real data, so we need a more accurate model to relax this assumption for estimating the result size of a SPARQL query. Second, the SPARQL query in Figure 4.1 is a composition of a chain query pattern and a star query pattern, and these two types of patterns are very common in SPARQL queries. It is desirable
CHAPTER 4. SELECTIVITY ESTIMATION FOR RDF QUERIES

Figure 4.1: RDF SPARQL query

to find appropriate ways to estimate the selectivity of these common patterns.

Based on the observations, we propose new methods for selectivity estimation of SPARQL queries. The contributions of this chapter can be summarized as follows:

- We study the star and chain patterns with correlated properties and propose two methods for estimating the selectivity of them respectively. For star query patterns, we construct Bayesian networks to compactly represent the joint probability distribution over values of correlated properties. And for chain query patterns, we build the chain histogram, which can obtain a good balance between the estimation accuracy and space cost.

- For an arbitrary SPARQL query represented as a composite graph pattern, we propose algorithms for maximally combining the statistics of chain paths and star paths that we have precomputed to estimate the overall selectivity of the graph pattern.

- We conduct experiments on both synthetic and real-world datasets show the effectiveness and performance of our approach.
In this chapter, we adopt the method in [SSB+08] to estimate the selectivity for single triple patterns and we mainly focus on selectivity estimation for joined triple patterns with select and join operators.

**Problem Definition.** The problem that we tackle in this chapter can be summarized as follows: Given an RDF database $D$ and a SPARQL graph pattern $Q$, we estimate the selectivity $\text{sel}(Q)$ of $Q$. $\text{sel}(Q)$ stands for the count of results in database $D$ satisfying $Q$.

The remainder of this chapter is organized as follows. In Section 4.2 and Section 4.3, we propose two methods to estimate the selectivity of star and chain query patterns, respectively. Section 4.4 presents the algorithms for estimating the selectivity of arbitrary composite SPARQL graph patterns. Section 4.5 describes an experimental evaluation of our approach. At last, in Section 4.6 we conclude our work.

## 4.2 Estimation for Star Query Patterns

In this section, we present the method of using Bayesian networks to estimate the selectivity of star query patterns.

### 4.2.1 Estimation for Star Patterns Using Bayesian Networks

Star query pattern that commonly occurs in SPARQL queries is used to retrieve entities with some constraints. It has the form of a number of triple patterns with different properties sharing the same subject variable. Normally, the properties involved in a star query patterns are correlated. For example, Figure 4.2 shows a star query pattern $Q$ and the properties $\text{Income, Education, Position,}$
TeacherOf are correlated. Thus, as shown in the introduction, the join uniformity assumption would cause a highly inaccurate selectivity estimation for star query patterns.

For estimating the selectivities of star query patterns precisely, we need some helpful statistics. We precompute statistics for star paths. A star path is a set of properties which share the same domain. Given a star path with properties \(prop_1, prop_2, \cdots, prop_n\), we use \((prop_1, prop_2, \cdots, prop_n)\) to denote. In Figure 4.2, \(Q_s(Income, Education, Position, TeacherOf)\) is a star path. Since the number of possible star paths could be huge \((2^n, \text{where } n \text{ is the number of properties in the database})\), it is impossible to precompute statistics for all star paths. In this chapter, we target the star paths in which properties are strongly correlated.

We assume that if a star path appears in the data graph frequently then the properties contained in this star path are strongly correlated. This assumption is also made in data mining field to find association rules. We consider the frequent star paths in the data graph and construct appropriate structures to store statistics of them. For finding frequent star paths, we note that the Apriori property is available here. With this property, we can easily find frequent star paths. The pseudocode and details of this algorithm are presented in Algorithm 4.1. In this algorithm, it first filters some non-promising candidates according to the Apriori property (procedure \(Gen\_Candidates\)) and then finds top-k frequent star paths (Line 6 - Line 15).

A frequent star path is logically associated with a table \(R\) called cluster-property table [WSKR03] that contains a set of entities to be queried. For example, Figure 4.2 shows a cluster-property table of star path \(Q_s\). Each row of the table is an academic staff with values of properties \(Education, Income, Position\) and \(TeacherOf\). Note that different with [WSKR03], we do not store cluster-property tables for frequent star paths. More compressed structures would be
Algorithm 4.1 FindingFreqStarPaths

Input: Property set PropSet, minimum support min_sup;
Output: frequent star path set L;

1: for each prop_i ∈ PropSet do
2:   if |(prop_i)| > min_sup then
3:      L_1 = L_1 ∪ {(prop_i)}; { |(prop_i)| means the frequency of star path (prop_i) }
4: end if
5: end for
6: for (k = 2; L_k ≠ Ω; k + +) do
7:   C_k = Gen_Candidates(L_{k-1}, min_sup);
8:   for each c’ ∈ C_k do
9:       if |c’| > min_sup then
10:          L_k = L_k ∪ {c’};
11:     end if
12: end for
13: end for
14: return L = ⋃_{k=2}^L_k

1: Procedure Gen_Candidates(L_{k-1}, min_sup)
2: for each path l with length k do
3:   if ∀ sub path l_{sub} of l with length k - 1 ∉ L_{k-1} then
4:      delete l_{sub};
5:   else
6:      C_k = C_k ∪ {l_{sub}};
7:   end if
8: end for
9: return C_k

constructed for selectivity estimation.

Given a frequent star path Q_s(prop_1, prop_2, · · ·, prop_n) and its cluster-property table R, the selectivity of star query Q(?x, prop_1, o_1) (?x, prop_2, o_2) · · ·, (?x, prop_n, o_n) denoted by sel(Q), can be computed as follows:

\[ sel(Q) = \text{Pr}(\text{prop}_1 = o_1, \text{prop}_2 = o_2, \ldots, \text{prop}_n = o_n) \cdot |R| \]

where \( \text{Pr}(\text{prop}_1 = o_1, \text{prop}_2 = o_2, \ldots, \text{prop}_n = o_n) \) denotes the joint probability distribution over object values of properties \( \text{prop}_1, \text{prop}_2, \ldots, \text{prop}_n \) in table \( R \).
and $|R|$ is the number of rows in the cluster-property table $R$.

Since the number of rows $|R|$ is easy to know, we will focus on computing the joint probability distribution $\Pr(prop_1 = o_1, prop_2 = o_2, \cdots, prop_n = o_n)$. Unfortunately, it is impossible to explicitly store $\Pr(prop_1 = o_1, prop_2 = o_2, \cdots, prop_n = o_n)$ because the possible combinations of property values would be exponential. Thus, we need an appropriate structure to approximately store the joint probability distribution information.

![Star path and star query pattern](image)

### Figure 4.2: Star path and star query pattern

The Bayesian network [CHS97] can approximately represent the probability distribution over a set of variables using a little space. Bayesian networks make use of Bayes’ Rule and conditional independence assumption to compactly represent the full joint probability distribution. Let $X, Y, Z$ be three discrete valued random variables. We say that $X$ is conditionally independent of $Y$ given $Z$ if the probability distribution of $X$ is independent of the value of $Y$ given a value...
for \( Z \); that is:

\[
\Pr(X = x_i | Y = y_j, Z = z_k) = \Pr(X = x_i | Z = z_k)
\]

where \( x_i, y_j, z_k \) are values of variables \( X, Y, Z \). The conditional independence assumptions associated with a Bayesian network and conditional probability tables (CPTs), determine a joint probability distribution. For example, in Figure 4.3, a Bayesian network is constructed on cluster-property table in Figure 4.2. We can see that properties \( \text{Education} \) and \( \text{Income} \) are conditionally independent given condition \( \text{Position} \), which means if we already know the position of some person, knowing his education information does not make any difference to our beliefs about his income.

![Figure 4.3: Bayesian Network](image)

For a star query pattern \( Q \) with properties and object values \( prop_1 = o_1, \)
prop_2 = o_2, \ldots, prop_n = o_n$, given a Bayesian network $\beta$, we have:

$$Pr_\beta(prop_1 = o_1, prop_2 = o_2, \ldots, prop_n = o_n)$$

$$= \prod_{i=1}^{n} Pr(prop_i = o_i \mid Parents(prop_i) = \vec{o}_k)$$

where $Parents(prop_i)$ denotes the set of immediate predecessors of $prop_i$ in the network and $\vec{o}_k$ denotes the set of values of $Parents(prop_i)$. Note that for computing $Pr(prop_i \mid Parents(prop_i) = \vec{o}_k)$, we only need to know the values of $prop_i$’s parent properties, which would save a lot of space in practice. So given the Bayesian network $\beta$, we can use $Pr_\beta(prop_1 = o_1, prop_2 = o_2, \ldots, prop_n = o_n)$ to approximately represent $Pr(prop_1 = o_1, prop_2 = o_2, \ldots, prop_n = o_n)$. We have:

$$sel(Q) = Pr(prop_1 = o_1, prop_2 = o_2, \ldots, prop_n = o_n) \cdot |R|$$

$$\approx Pr_\beta(prop_1 = o_1, prop_2 = o_2, \ldots, prop_n = o_n) \cdot |R|$$

$$= \prod_{i=1}^{n} Pr(prop_i = o_i \mid Parents(prop_i) = \vec{o}_k) \cdot |R| \quad (4.1)$$

**Example 1.** Given the star pattern $Q(\textnormal{?Z, income, } \leq \textnormal{‘70K’}) (\textnormal{?Z, Education, ‘PhD’}) (\textnormal{?Z, Position, ‘Prof.’}) (\textnormal{?Z, TeacherOf, ‘Adc’})$ shown in Figure 4.2 and Bayesian network described in Figure 4.3, we compute the selectivity of $Q$ as
follows:

\[
\text{sel}(Q) = \Pr(Edu = 'PhD', Pos = 'Prof.', Inc \leq '70K', TOf = 'Adc') \cdot |R|
\]

\[
= \Pr(Pos = 'Prof.' | Edu = 'PhD')
\cdot \Pr(Inc \leq '70K' | Pos = 'Prof.'
\cdot \Pr(TOf = 'Adc' | Pos = 'Prof.'
\cdot \Pr(Edu = 'PhD') \cdot |R|
\]

\[
= 0.4 \cdot 0.01 \cdot 0.7 \cdot 0.3 \cdot |R| = 0.00084 \cdot |R|
\]

4.2.2 Learning Bayesian Networks

To approximately represent the joint distribution of property values for selectivity estimation, we first generate the cluster-property tables for frequent star paths, and then Bayesian networks can be learned from these tables. From the Apriori property, we know that if a star path is frequent then its sub paths are also frequent. Given a frequent star path \((p_1, p_2, p_3)\) and one of its sub path \((p_1, p_3)\), if they have equal frequencies, namely \(|(p_1, p_2, p_3)| = |(p_1, p_3)|\), it indicates that they have the same instance set. In this case, the information contained in the cluster-property table of path \((p_1, p_3)\) is fully covered by the cluster-property table of path \((p_1, p_2, p_3)\) and we do not need to generate the cluster-property table for path \((p_1, p_3)\). Note that we will drop all cluster-property tables once Bayesian networks have been constructed.

Before building Bayesian networks, the property values are first clustered into equi-width subsets (abstracted values). Bayesian network construction algorithms can be grouped into two categories: one category of algorithms uses heuristic searching methods to construct a model and then evaluates it using a
scoring method [Bun94, HGC95, CH92]. The other category of algorithms constructs Bayesian networks by analyzing dependency relationships among nodes. The dependency relationships are measured by using some kind of conditional independence (CI) test. We adopt the algorithm in [CBL97], which employs mutual information to measure how close the relationship between two variables. The mutual information \( I(X, Y) \) of two variables \( X, Y \) is defined as follows:

\[
I(X, Y) = \sum_{x, y} \Pr(x, y) \log \frac{\Pr(x, y)}{\Pr(x) \cdot \Pr(y)}
\]

and the conditional mutual information is defined as

\[
I(X, Y|M) = \sum_{x, y, m} \Pr(x, y, m) \log \frac{\Pr(x, y|m)}{\Pr(x|m) \cdot \Pr(y|m)}
\]

where \( M \) is a set of observed variables. When \( I(X, Y|M) \) is smaller than a threshold \( \epsilon \), \( X \) and \( X \) are conditionally independent and we also call \( X \) and \( X \) are \( d \)-separated by the condition set \( M \).

The construction algorithm contains three steps. The first step is to compute mutual information of each pair of nodes as a measure of closeness, and creates a draft graph based on this information. The draft graph is a singly connected graph (a graph without loops). In the second phase, the algorithm adds edges when the pairs of nodes cannot be \( d \)-separated. In the third step, each edge of the Independent-map is examined using CI tests and will be removed if the two nodes of the edge can be \( d \)-separated. At the end of the third phase, the algorithm also carries out a procedure to orient the edges of the graph. For details on specific three steps, please see [CBL97]. The time complexity of this algorithm is \( O(n^4) \), where \( n \) is the number of properties involved in the cluster property table. In our
problem, it requires that the conditional probability tables learned fit in a small amount of memory, therefore we restrict the threshold $\epsilon$ for space limit when building Bayesian networks.

4.3 Estimation for Chain Patterns

The other kind of correlated properties commonly occur in SPARQL graph patterns is chain path. The chain path consists of a sequence of triple patterns where the object of a triple pattern is also the subject of the next triple pattern. For example, in Figure 4.4, $(?course, TakenBy, ?student)(?student, Age, ?age)$ is a chain path. Note that triple patterns of a chain path have only unbound subjects and objects. Given a chain path with a sequence of properties $p_1, \ldots, p_n$, we use $(p_1-p_2-\cdots-p_n)$ to denote it. For instance, in Figure 4.4, $(TakenBy-Age)$ is a chain path and tuples in the data graph matching this chain path fall in the subject-property matrix table of this chain path.

**Figure 4.4:** Chain path and chain query pattern

A chain query pattern is a chain path with possible literal constraints on the
star node (the subject of the first triple pattern) or end node (the object of the last triple pattern) in the chain path. For example, \( c_1(\text{?course}, \text{TakenBy}, \text{?student})(\text{?student}, \text{Age}, '22') \) shown in Figure 4.4 is a chain query pattern. The straightforward way to estimate the selectivity of a chain query pattern is combining the selectivity of each triple pattern in the product form. Obviously, it would be highly inaccurate if the properties are correlated. An alternative way is using the same method as we deal with star query patterns, namely, constructing the Bayesian networks to capture the dependencies among property values in frequent chain paths. However, this method is not applicable to chain query patterns. Unlike star paths, different orderings of the properties correspond to different chain paths. The number of chain paths (even frequent chain paths) could be huge and it would be very expensive to construct Bayesian networks for these chain paths. Thus, we hope to find another method.

Neumann et al. [NW08] precompute frequent chain paths in the RDF data graph and keep exact join statistics (frequencies) for them. With these statistics, it is easy to get the selectivity of frequent chain paths. However, for a chain query pattern (e.g., \( c_1 \) in Figure 4.4) that has literal constraints on the star node or end node in the chain path, they have to adopt the join uniformity assumption for selectivity estimation, which would be inaccurate. To deal with this problem, we adopt the histogram technique to solve it.

### 4.3.1 Constructing the Histogram

We first select frequent chain paths and construct the histogram for them. For these frequent chain paths, we can construct a chain frequency table \( CFT \) (shown in Figure 4.5), which has two attributes: Abstracted chain query pattern and Frequency. The values of attribute Abstracted chain query pattern are abstracted chain query pattern descriptors consisting of frequent chain paths with possible
value combinations of the start node and end node. Note that the possible values occurring on the start node and end node of paths are clustered into equi-width subsets called abstracted values. Each value of attribute Frequency is the count of tuples that match the abstracted chain pattern query in the data graph. We can see that each row of CFT indicates an abstracted chain query pattern with its frequency, so it is easy to get the selectivity of abstracted chain patterns from CFT. For example, the first tuple (\texttt{Prc/p1/20-22}, 26) of chain frequency table shown in Figure 4.5 indicates a chain query pattern (‘\texttt{Prc\_TakenBy}\_\texttt{student}’) (\texttt{?student\_Age\_20-22}) with its frequency 26. However, chain frequency table CFT could be too large to fit in a small amount of main memory. Thus, we construct the new data structure called \textit{chain histogram} to approximate the selectivity of chain query patterns.

![Chain Frequency Table CFT](image)

![Chain Histogram H](image)

Figure 4.5: An example of the chain frequency table and the chain histogram. “?” indicates a variable.

There are two considerations that should be taken into account when constructing a chain histogram. First, the histogram should be space efficient and
can be fitted in a small amount of main memory. Second, the histogram can also provide us accurate estimation. Our aim is to obtain a balance between the space and accuracy.

We first group the abstracted chain query patterns in $CFT$ into $B$ buckets according to their frequencies. That means the abstracted chain query patterns with similar frequencies are grouped into one bucket, which is helpful to guarantee the accuracy of estimation. For example, in Figure 4.5, abstracted chain patterns $(Prc/p1/20-22)$ and $(Adc/p1/23-25)$ are grouped into one bucket since they have similar frequencies. For each bucket we only need to save the average frequency and its abstracted chain query pattern members. Given an abstracted chain query pattern and which bucket it belongs to, we can easily get the approximate frequency of this chain query pattern. To space-efficiently store the elements of each bucket and accelerate the membership query processing, we use bloom filter technique.

A bloom filter [Blo70, JLY04] is a bit array of $m$ bits, all initially set to 0. There are $k$ different hash functions $h_1, h_2, \cdots h_k$ defined, each of which maps or hashes some set elements to one of the $m$ array positions with a uniform random distribution. To add an element $b$, feed it to each of the $k$ hash functions and all bits at positions $h_1(b), h_2(b), \cdots h_k(b)$ are set to 1. To query for an element $b$, feed it to each of the $k$ hash functions to get $k$ array positions. If any of the bits at these positions are 0, the element is not in the set; otherwise $b$ is in the set with a certain probability. Nevertheless, the false positive case may occur. The probability of false positives can be computed as follows:

$$Pr_{error} \approx (1 - e^{-kn/m})^k$$

Obviously, the probability of false positives decreases as $m$ (the number of
bits in the array) increases, and increases as \( n \) (the number of inserted elements) increases. Bloom filters have the unusual property that the time needed to either add items or to check whether an item is in the set is a fixed constant, \( O(k) \), completely independent of the number of items already in the set. Bloom filters have a strong space advantage over other data structures for representing sets. A bloom filter with 1% error and an optimal value of hash function number \( k \), requires only 9.6 bits per element.

We use bloom filters to store buckets and construct the chain histogram. An example of chain histogram is shown in Figure 4.5.

4.3.2 Estimation for Chain Patterns Using the Histogram

Given a chain query pattern \( C \) and a histogram \( H \), for estimating the selectivity of \( C \), we first map the basic level values of start node and end node of \( C \) to their abstracted values and generate the abstracted chain query pattern \( C_a \). For example, if a chain query pattern \( C(\text{?course}, \text{Takenby}, \text{?student})(\text{?student}, \text{Age}, '22') \) is posed on the database, we map the value ‘22’ to its abstracted value ‘20-22’ and get the abstracted chain query pattern \( C_a(\text{?y}, \text{Takenby}, \text{?student})(\text{?student}, \text{Age}, '20-22') \). The uniformity assumption is made here when computing the coefficient \( \eta \) between the selectivities of \( C \) and \( C_a \). We can acquire the approximate selectivity of chain query pattern \( C_a \) from the chain histogram \( H \). Note that though it is rare, it is possible that the pattern \( C_a \) is reported to fall in multiple buckets due to the false positive error of the bloom filter. In this case, we simply return the count of the first bucket which reports that the pattern \( C_a \) is its member. This process is given in Algorithm 4.2 \( \text{GetChainSel} \). In this algorithm, we first map a chain query pattern to its abstracted chain query pattern and compute the coefficient between them (Line 1 - Line 2). Then check all buckets in the histogram and compute the selectivity according to the count of the bucket
which the abstracted chain query falls into (Line 3 - Line 8).

The time complexity of obtaining the selectivity of a chain pattern is $O(KB)$, where $K$ is the maximal number of hash functions used in a bloom filter and $B$ is the number of buckets (or bloom filters) in the chain histogram.

\begin{algorithm}
\caption{GetChainSel}
\begin{algorithmic}[1]
\Require Chain pattern $C$, histogram $H$;
\Ensure The selectivity $sel$ of chain pattern $C$;
\State Map chain query pattern $C$ to $C_a$; \COMMENT {$C_a$ is abstracted chain pattern of $C$}
\State Compute the coefficient $\eta$ between $C$ and $C_a$;
\ForAll{buckets $b_k$ in $H$ }
\If{$C_a$ falls into a certain bucket $b_j$ }
\State $sel_a = b_j.count$; \COMMENT {$sel_a$ is the selectivity of $C_a$}
\State $sel = \eta \times sel_a$;
\EndIf
\EndFor
\State \textbf{return} $0$;
\end{algorithmic}
\end{algorithm}

For example, given a chain query pattern $(?y, \text{Takenby}, ?\text{student})$ $(?\text{student}, \text{Age}, '22')$, we compute the selectivity of this chain query pattern as follows:

\begin{align*}
\text{sel} &= \frac{1}{3} \times \text{sel}(?/p1/20 - 22) \\
&= \frac{1}{3} \times 74 \approx 25;
\end{align*}

\section{4.4 Estimation for Composite Graph Patterns}

In this section, we discuss how to estimate the selectivity of an arbitrary composite graph pattern with the maximum use of the precomputed statistics of star and chain paths.
4.4.1 Decomposition of SPARQL Graph Patterns

From previous sections, we know that the precomputed statistics of the star and chain paths can help to estimate the selectivity of star and chain query patterns, respectively. As a SPARQL graph pattern $SC$ (an example is shown in Figure 4.6) is a composite graph pattern, we decompose it into a set of star and chain query patterns and then use the precomputed statistics of star and chain paths to obtain the overall selectivity. For maximally using precomputed statistics, we hope to obtain a set of edge-disjoint precomputed star and chain paths as building blocks that constitute a subgraph $SC'$ of graph pattern $SC$, which has the largest number of edges.

There is a special case we need to consider. A SPARQL query may include non-property-bound triple patterns. For this case, we remove such triple patterns from graph patterns (retaining join nodes) and estimate the selectivities of these triple patterns as single triple patterns. At last, we combine the selectivity of the whole graph pattern with the independent assumption.

For a precomputed chain path, we have the statistics (provided by the chain histogram) of value distributions over the start node and end node of this path. Thus, the chain path can only be selected as a whole to constitute a query graph pattern. For a precomputed star path, since we have the statistics (provided by the Bayesian networks) of value distributions over any subset of properties in this star path, the star path and its sub paths can be selected as building blocks.

We use $PS$ to refer to the set of star and chain paths whose statistics have been precomputed. Removing a path $p$ from graph $Q$, denoted by $Q \setminus p$, consists of removing all edges of $p$ from $Q$, followed by removing all stand-alone nodes. Given a query pattern $Q$ and precomputed path set $PS$, the maximum path cover of $Q$ with respect to $PS$ denoted by $Cov_{PS}(Q)$ is a subset of edge-disjoint paths in $PS$, which can constitute a subgraph $Q'$ of $Q$ such that $Q'$ has the largest
Our problem is to obtain $\text{Cov}_{PS}(Q)$. It is similar to the classic 0-1 knapsack problem: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than a given limit and the total value is as large as possible.

In our problem, the items are precomputed chain paths and star paths. For a chain or star path item, the value of this item is the length of the path and the weight is just the path since we use this path as a building block. Now, we need to determine the status for each path item ("picked" or "not picked") so that the picked edge-disjoint paths (or sub paths) can constitute a subgraph $Q'$ of $Q$, which has the largest number of edges. The weight limit here is the query graph pattern $Q$.

The straightforward way of finding $\text{Cov}_{PS}(Q)$ is to adopt greedy strategy...
which is presented in Algorithm 4.3 *GreedyFindpathCover*. It first inserts all paths into priority queue \( PQ \) (Line 2-Line 6) and then eagerly uses the longest paths in \( PS \) to constitute a subgraph of \( Q \) (Line 7 - Line 16). The time complexity of the algorithm is \( O(|PS| \cdot |E|) \), where \(|PS|\) is the number of paths in \( PS \) and \(|E|\) is the number of edges of \( Q \). Due to the local optimum nature of the algorithm, *GreedyFindpathCover* would fail to achieve the global optimum (i.e., the maximum path cover of \( Q \)) in some cases. However this algorithm is still of interest because it provides an efficient way to find a path cover of \( Q \) when graph pattern \( Q \) is huge and dense.

Algorithm 4.3 *GreedyFindpathCover*

**Input:** Path set \( PS \), Query \( Q \);

**Output:** A maximum path cover set \( Cov_{PS}(Q) \);

1: priority queue \( PQ = \Omega \), \( Cov_{PS}(Q) = \Omega \);
2: for all path \( p_k \) in \( PS \) do
3: if \( p_k \subseteq Q \) then
4: insert \( p_k \) to the priority queue \( PQ \); {according to \( p_k \)'s length}
5: end if
6: end for
7: while not empty(\( PQ \)) do
8: remove(\( PQ, p_j \));
9: if \( p_j \subseteq Q \) then
10: \( Cov_{PS}(Q) = Cov_{PS}(Q) \cup p_j \);
11: \( Q = Q \setminus p_j \);
12: end if
13: if \( Q \) is empty then
14: return \( Cov_{PS}(Q) \);
15: end if
16: end while
17: return \( Cov_{PS}(Q) \);

Now we propose an optimal algorithm based on *dynamic programming*. Dynamic programming is a method for solving those problems which exhibit the property of *optimal substructures* by breaking them into overlapping subproblems. It is also applicable to our problem. Given a graph pattern \( Q \), we represent
PS as a path array $PS[1..n]$ that contains all precomputed chain paths, star paths. Let $\text{maxCov}(i, Q)$ be the number of edges of subgraph $Q'$ of $Q$ such that $Q'$ consists of edge-disjoint paths from $PS[1..i]$ and has the largest number of edges. Here we use the last item $i$ and the remaining graph pattern (current weight limit) to index subproblems. $\text{edgeNum}(PS[j])$ indicates the edge number of path item $j$ in $PS$. $\text{maxCov}(i, Q)$ can be obtained through combining the optimal solutions of its subproblems. We have the following properties:

\[
\begin{align*}
\text{maxCov}(0, Q) &= 0; \\
\text{maxCov}(i, \Omega) &= 0, \text{ if } Q \text{ is null graph}; \\
\text{maxCov}(i, Q) &= \text{maxCov}(i - 1, Q), \text{ if } PS[i] \not\subseteq Q; \\
\text{maxCov}(i, Q) &= \text{Max}(\text{maxCov}(i - 1, Q), \\
&\quad \text{edgeNum}(PS[i]) + \text{maxCov}(i - 1, Q \setminus PS[i]));
\end{align*}
\]

Based on these properties, we present the algorithm in Algorithm 4.4 $\text{OptimalFC}$. In this algorithm, when $i = 0$ or graph pattern $Q$ is a null graph, $\text{maxCov}(i, Q) = 0$ (Line 2 - Line 4 of Function $\text{maxCov}$). If path $PS[i]$ is not a subgraph of $Q$, we drop $PS[i]$ and $\text{maxCov}(i, Q)$ is equal to its subproblem $\text{maxCov}(i - 1, Q)$ (Line 5 - Line 7 of Function $\text{maxCov}$); otherwise, $\text{maxCov}(i, Q)$ is computed through comparing the values in two cases where $PS[i]$ is picked or not picked (Line 8 - Line 13 of Function $\text{maxCov}$). At last, we add all picked paths into $\text{Cov}_{PS}(Q)$ (Line 3 - Line 8).

As we know, 0-1 knapsack problem is a classic NP-hard problem. And the time complexity of our optimal algorithm is $O(|PS| \cdot |Q|)$, where $|Q|$ is the number of subgraphs of $Q$, which is exponential to the number of edges of $Q$. In practice, if query graph pattern $Q$ is sparse (i.e., the constituent vertices are of low degree) and not large, we can employ this optimal algorithm; otherwise, the
Algorithm 4.4 OptimalFC

Input: A path array $PS[1..n]$, A boolean array flag[1..n], Graph pattern $Q$;
Output: A path cover set $Cov_{PS}(Q)$ of $Q$;

1: Initial flag[1..n] = false;
2: int maxNum = maxCov(n, Q); {get the number of edges covered, function $maxCov(n, Q)$ is defined below}
3: for $k := 1$ to $n$ do
4:   if flag[$k$] = true then
5:     add $PS[k]$ to $Cov_{PS}(Q)$;
6:   end if
7: end for
8: return $Cov_{PS}(Q)$;

1: Function int $maxCov(n, Q)$;
2: if $n = 0$ or $Q = \Omega$ then
3:   return 0;
4: end if
5: if path $PS[n] \not\subseteq Q$ then
6:   return $maxCov(n - 1, Q)$;
7: end if
8: if $maxCov(n - 1, Q) \leq edgeNum((PS[n]) + maxCov(n - 1, Q \setminus PS[n]))$ then
9:   $Q = Q \setminus PS[n]$;
10: flag[$n$] = true; {indicating that path $PS[n]$ is picked}
11: return $maxCov(n - 1, Q \setminus PS[n])$;
12: else
13:   return $maxCov(n - 1, Q)$;
14: end if
greedy algorithm proposed can be used instead.

### 4.4.2 Joining a Star Pattern with a Chain Pattern

After a composite graph pattern is decomposed into a set of star and chain query patterns, how do we compute the selectivity of this graph pattern? To address this problem, let us first begin with a simple case shown in Figure 4.6.

In Figure 4.6, the composite graph pattern \( SC \) can be decomposed into a star query pattern \( S \) and a chain query pattern \( C \) whose corresponding star path and chain path have been computed. The variable \( ?Y \) is the join node. We use \( \text{val}(C.Y) \) to refer to the value set of variable \( C.Y \) and \( \text{val}(S.Y) \) for variable \( S.Y \). Our aim here is to estimate the selectivity of \( SC \) denoted by \( \text{sel}(SC) \).

Using the Bayesian networks and the chain histogram constructed, we can get \( \text{sel}(S) \) and \( \text{sel}(C) \). However, estimating precisely the selectivity of \( SC \) is not trivial. A tuple \( s \) matching star query pattern \( S \) may not join with a tuple \( c \) matching chain query pattern \( C \), since they may have different values on variable \( ?Y \). For estimating precisely the selectivity of \( SC \), it is crucial to know the probability distributions over \( \text{val}(S.Y) \) and \( \text{val}(C.Y) \). However, obtaining the exact
probability distributions of values over \( \text{val}(S.Y) \) and \( \text{val}(C.Y) \) is hard without executing the star query pattern \( S \) and chain query pattern \( C \). Fortunately, the probability distribution of abstracted values on the join variable \( ?Y \) (target variable) can be inferred from the Bayesian network.

For example, suppose we have the Bayesian network shown in Figure 4.3 and we can infer the probability distribution of abstracted values over \( \text{val}(S.Y) \) (i.e., the probability distribution over the object values of property \( \text{TeacherOf}(TOf) \) for short) as follows:

\[
\text{Pr}(TOf = 'Adc'|Edu = 'PhD', Pos = 'Prof.', Inc \leq '70K') = \text{Pr}(\text{TeacherOf} = 'Adc'|Pos = 'Prof.') = 0.7
\]

\[
\text{Pr}(TOf = 'Prc'|Edu = 'PhD', Pos = 'Prof.', Inc \leq '70K') = \text{Pr}(\text{TeacherOf} = 'Prc'|Pos = 'Prof.') = 0.3
\]

We can see that 70 percent of tuples matching pattern \( S \) have value “Adc” and 30 percent of tuples have value “Prc” on variable \( ?Y \). Using the probability distribution over \( \text{val}(S.Y) \), we can easily compute the selectivity of \( SC \). Recall that we use the chain histogram to estimate the selectivities of chain query patterns. Now we take the values “Adc” and “Prc” as the values of start node of chain \( C \) separately, then test chain query patterns “\( Adc/C \)” “\( Prc/C \)” in the chain histogram and get the selectivities \( sel(Adc/C) \) and \( sel(Prc/C) \). Since bloom filter is efficient for answering set membership queries, this process is time efficient. At last, we can obtain the selectivity of \( SC \) as \( 0.7 \cdot \text{sel}(S) \cdot \text{sel}(Adc/C) + 0.3 \cdot \text{sel}(S) \cdot \text{sel}(Prc/C). \)
This process is described in Algorithm 4.5 *ComSelStar-Chain*. We go through all abstract values of the join node and then get the selectivity of the composite graph pattern by cumulating the count of tuples from the chain query pattern and the star query pattern which have the same values on the join node.

**Algorithm 4.5 ComSelStar-Chain**

**Input:** A composite graph pattern $SC(S \Join C)$ consisting of a chain pattern $C$ and a star pattern $S$; Join node $?Y$;  

**Output:** The selectivity of chain pattern $SC$;  

1: $\text{count} = 0$;  
2: for all abstracted values $a_i$ over the join node $Y$ do  
3: \quad Compute the probability $\Pr(Y = a_i| M(S))$; \{ $M(S)$ are the observed values in $S$ \}  
4: \quad $\text{count}_1 = \Pr(Y = a_i| M(S)) \times \text{sel}(S)$; \{ compute $\text{sel}(S)$ using formula 4.1 \}  
5: \quad $\text{count}_2 = \text{GetChainSel}(a_i/C)$;  
6: \quad $\text{count} = \text{count}_1 \times \text{count}_2$;  
7: end for  
8: \text{return} $\text{count}$;  

### 4.4.3 Other Cases

We have discussed the case where a composite graph pattern can be decomposed into one star pattern and one chain pattern (Figure 4.6). What if a composite graph pattern is decomposed into two star patterns (Figure 4.7(b)) or two chain patterns (Figure 4.7(c))? If a graph pattern $S(S_1 \Join S_2)$ is decomposed into two star patterns $S_1$ and $S_2$, we go through each possible abstracted value $a_i$ on the join node $?Y$ of two star paths. Through inference on the Bayesian networks constructed, it is easy to acquire the probabilities $\Pr(Y = a_i| M(S_1))$ and $\Pr(Y = a_i| M(S_2))$ such that $M(S_1)$ and $M(S_2)$ are observed object values in two star patterns. We then get the selectivity of the composite graph pattern through cumulating the count of tuples from two star patterns which have the same values on the join node. The algorithm is described in Algorithm 4.6 *ComSelStar-Star*.  

Algorithm 4.6 ComSelStar-Star

**Input:** A composite graph pattern $S(S_1 \bowtie S_2)$ consisting of two star pattern $S_1$ and $S_2$, Join node $Y$;

**Output:** The selectivity of $S$;

1: $count = 0$;
2: for all abstracted values $a_i$ over the join node $Y$ do
3: Compute the probability $Pr(Y = a_i|M(S_1))$, $Pr(Y = a_i|M(S_2))$ ; \{M(S_1), M(S_2)\} are the observed values in $S_1$, $S_2$\}
4: $count_1 = Pr(Y = a_i|M(S_1)) \times sel(S_1));
5: $count_2 = Pr(Y = a_i|M(S_2)) \times sel(S_2))$;
6: $count+ = count_1 \times count_2$;
7: end for
8: return $count$;

Algorithm 4.7 ComSelChain-Chain

**Input:** A composite graph pattern $C(C_1 \bowtie C_2)$ consisting of two chain patterns $C_1$, $C_2$; Join node $Y$;

**Output:** The selectivity of chain pattern $C$;

1: $count = 0$;
2: for all abstracted values $a_i$ over the join node $Y$ do
3: $count_{1} = GetChainSel(a_i/C_1)$;
4: $count_{2} = GetChainSel(a_i/C_2)$;
5: $count+ = count_{1} \times count_{2}$;
6: end for
7: return $count$;

In the case where the graph pattern $C(C_1 \bowtie C_2)$ is decomposed into two chain query patterns $C_1$, $C_2$ that join on variable $?Y$, the method is similar (presented in Algorithm 4.7 ComSelChain-Chain). We go through each possible abstracted value $a_i$ of the join node, and the selectivities of two chain query patterns with value $a_i$ can be obtained using the chain histogram. $sel(C_1 \bowtie C_2 | Y = a_i)$ is computed through combining $sel(a_i/C_1)$ and $sel(a_i/C_2)$ in the product form. At last, we cumulate $sel(C_1 \bowtie C_2 | Y = a_i)$ with all possible abstracted values. Since Bloom filter is quick to get the selectivities of two chain patterns, this method is also time efficient.

Now we extend our method to the more general case where the path cover
Algorithm 4.8 ComSelectivity

Input: A query $Q$, a path cover set $\text{Cov}_{PS}(Q)$ of query $Q$;
Output: The selectivity $\text{sel}(Q)$ of query $Q$;

1: while $\exists$ two patterns $p_i, p_j \in \text{Cov}_{PS}(Q)$ which can join together do
2: $p_{ij} = p_i \bowtie p_j$;
3: if $p_i$ and $p_j$ are star or chain patterns then
4: $\text{sel}(p_{ij}) = \text{ComTwoPatterns}(p_{ij})$;
5: Maintain a selectivity distribution table $T_{p_{ij}}$ for $p_{ij}$;
6: end if
7: if $p_i$ or $p_j$ is a composite pattern then
8: Compute $\text{sel}(p_{ij})$; \{Using the chain histogram or inference on Bayesian networks and information provided by $T_{p_i}$ or $T_{p_j}$ \}
9: Maintain a selectivity distribution table $T_{p_{ij}}$ for $p_{ij}$;
10: end if
11: $Q = Q \setminus p_i \setminus p_j$;
12: $\text{Cov}_{PS}(Q) = \text{Cov}_{PS}(Q) \cup p_{ij} \setminus p_i \setminus p_j$;
13: if $Q$ is $\Omega$ then
14: return $\text{sel}(Q) = \text{sel}(p_{ij})$;
15: end if
16: end while
17: return $\text{sel}(Q) = \text{sel}(p_{ij}) \cdot \prod_{k=1}^{m} \text{sel}(p_{uk})$; \{$p_{u1}...p_{um}$ are uncovered single paths in $Q$ \}

1: Function $\text{ComTwoPatterns}(p_{ij})$
2: if $p_{ij}$ is a star-chain pattern then
3: return $\text{ComSelStar-Chain}(p_{ij})$;
4: end if
5: if $p_{ij}$ is a chain-chain pattern then
6: return $\text{ComSelChain-Chain}(p_{ij})$;
7: end if
8: if $p_{ij}$ is a star-star pattern then
9: return $\text{ComSelStar-Star}(p_{ij})$
10: end if
set $Cov_{PS}(Q)$ of query $Q$ consists of multiple star patterns and chain patterns. This algorithm is presented in Algorithm 4.8 ComSelectivity. We first select two patterns $p_i$ and $p_j$ from $Cov_{PS}(Q)$ that can join together. Compute the selectivity for the composite pattern $p_{ij} = p_i \bowtie p_j$ maintain a selectivity distribution table $T_{p_{ij}}$ indicating the selectivity distributions over the possible abstracted values of the join node in $p_{ij}$ (Line 1-Line 10). Iterate this process until there is only one composite graph pattern left in $Cov_{PS}(Q)$. For the part of $Q$ that is not covered by $Cov_{PS}(Q)$, we deal with it as a set of single triple patterns and combine the results of them in the product form with the independence assumption (Line 17).

4.5 Experiments

We describe in this section the result of experiments we have conducted to compare our method with existing work. We first describe the experiment setup and then present results of different methods.

4.5.1 Experimental Methodology

Experiment setup. We implemented the proposed algorithms of selectivity estimation in this chapter. They are Bayesian network based selectivity estimation method for star query patterns (BNM); chain histogram based method for chain query patterns (CHM), Greedy and Optimal decomposition algorithms for composite query patterns, respectively. All algorithms are run on a Windows XP professional operating system. The hardware is a PC with Intel Pentium 4 3.0GHz CPU, 4 GB memory and 150G hard disk.

Data sets. Both synthetic and real-world datasets are used in our experiments. We present results on three datasets: (1) LUBM $^2$ [GPH04] is developed

\[http://swat.cse.lehigh.edu/projects/lubm/\]
by Lehigh University. It consists of a university domain ontology (containing 43 classes and 32 properties) and we generated about 25M distinct triples; (2) SwetoDBLP \(^3\) [AMHAS07] is a dataset describing real computer science bibliography information. It contains about 11M triples; (3) YAGO \(^4\) [SKW07] consists of facts extracted from Wikipedia and integrated with the Wordnet thesaurus and contains about 40M triples. Compared with the former two datasets, it is relatively heterogeneous. Some details of three datasets are shown in Table 4.1.

### Table 4.1: Statistics of datasets

<table>
<thead>
<tr>
<th></th>
<th>LUBM</th>
<th>DBLP</th>
<th>YAGO</th>
</tr>
</thead>
<tbody>
<tr>
<td># Triples</td>
<td>25,511,071</td>
<td>11,014,618</td>
<td>40,114,899</td>
</tr>
<tr>
<td># Instances</td>
<td>4,555,428</td>
<td>2,395,467</td>
<td>3,883,360</td>
</tr>
<tr>
<td># Properties</td>
<td>43</td>
<td>45</td>
<td>93</td>
</tr>
</tbody>
</table>

**Query Loads.** There are three kinds of query patterns used in our experiments: star query patterns, chain query patterns and composite query patterns, which are combinations of star and chain query patterns. Figure 4.8 shows some information about query sets used in the experiments. We use the seed queries (part of them are shown in Appendix A) with some random literal values to generate 10 query sets for each dataset. Each query set contains 5 queries.

**Previous Techniques.** We compare our method with two previous works: 1) In [SSB+08], the authors propose the selectivity estimation method PF based on the probabilistic framework. They build summary statistics of joined triple patterns for RDF data. Notice that they only discuss the case of two joined triple patterns and in the experiment we extend statistics to multiple joined triple patterns; 2) In [NW08], the authors propose the estimation method in RDF-3X system, which computes two kinds of statistics for selectivity estimation. The first one is called specialized histograms which assumes independence among

---

\(^3\)http://lsdis.cs.uga.edu/projects/semdis/swetodblp/

\(^4\)http://www.mpi-inf.mpg.de/yago-naga/yago/downloads_yago.html
CHAPTER 4. SELECTIVITY ESTIMATION FOR RDF QUERIES

Figure 4.8: Some information about query sets used in experiments. Query sets \(Qs_1, Qs_2, Qs_3\) are used in the star query experiments. Query sets \(Qc_1, Qc_2, Qc_3\) are used in the chain query experiments. Query sets \(Qsc_1, Qsc_2, Qsc_3, Qsc_4\) are used in the composite query experiments.

Evaluation Method. We use relative error \(RE\) to present the performance of different techniques of selectivity estimation. Given a true selectivity \(sel\) and its estimate \(\tilde{sel}\), we use the relative error formula \(RE(sel, \tilde{sel}) = \frac{|sel - \tilde{sel}|}{\max(1, sel)}\) for positive queries \((sel > 0)\) and negative queries \((sel = 0)\).

4.5.2 Offline Evaluation

Learning Bayesian Networks. Before evaluating our methods for star query patterns, we first constructed Bayesian networks for frequent star paths in three datasets. For LUBM dataset, we constructed cluster property tables for 6 common star paths that correspond entities such as Faculty, Student, Course and so on. An entity may correspond several classes. Similarly, we constructed cluster property tables for 7 common star paths) in DBLP dataset. Then Bayesian networks are learned from these cluster property tables. Clearly, with the increase of space for storing learned Bayesian networks, the constructed Bayesian networks and CPTs are more accurate over the data. Due to the efficiency consideration,
we set the space limit of each dataset for storing each Bayesian network to 16 KB. Some details about learning Bayesian networks are shown in Table 4.2. From Table 4.2, we can see that the learning time of Bayesian networks for three datasets are 127.3s, 232.2s, 2112.9s, respectively. In DBLP dataset, we have learned 7 Bayesian networks from cluster property tables and the largest cluster property table contains 362,907 tuples. However, the time of learning Bayesian network for this cluster property table is 72.2s, which is acceptable. For YAGO dataset, since the data is heterogeneous there are more frequent paths and we construct 100 Bayesian Networks for them.

<table>
<thead>
<tr>
<th>#</th>
<th>LUBM</th>
<th>DBLP</th>
<th>YAGO</th>
</tr>
</thead>
<tbody>
<tr>
<td># Learned Bayesian Networks</td>
<td>6</td>
<td>7</td>
<td>100</td>
</tr>
<tr>
<td># Maximum tuples in cluster property tables</td>
<td>100,532</td>
<td>362,907</td>
<td>161,496</td>
</tr>
<tr>
<td># Properties in cluster property tables</td>
<td>2-9</td>
<td>2-8</td>
<td>2-8</td>
</tr>
<tr>
<td>Learning time (s)</td>
<td>227.3</td>
<td>372.7</td>
<td>2112.9</td>
</tr>
<tr>
<td>Space limit (KB) for storing each Bayesian Network</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>

**Constructing chain histograms.** To employ our chain histogram based method to estimate the selectivity of chain query patterns, we constructed chain histograms for chain paths \(^5\) of three datasets. We fix the space limit for storing the chain histogram of each dataset to 16 KB. Some details of constructing chain histograms are shown in Table 4.3.

\(^5\)Some infinite long loops are broken into short chain paths.
4.5.3 Estimation Accuracy

In this set of experiments, we study and compare the accuracy of different selectivity estimation methods. Figure 4.9, 4.10 and 4.11 show the accuracy of three methods for different query sets on three datasets. In all figures, X-axis are different query sets and Y-axis is the average relative error RE.

Performance of methods for star query patterns. In this experiment, we evaluate our method of selectivity estimation for star query patterns. For each dataset, 15 star query patterns are developed. They are grouped into 3 query sets: $Q_s^1$, $Q_s^2$, $Q_s^3$ (see Figure 4.8). $Q_s^1$, $Q_s^2$ and $Q_s^3$ contain star query patterns that have 2, 3 to 4, and more than 5 properties, respectively. Each query set contains 5 star query patterns. The relative errors of different methods on three datasets are shown in Figure 4.9(a), (b) and (c). “BNM” indicates our Bayesian network based selectivity estimation method for star query patterns.

Figure 4.9: Relative errors for star query patterns.
From Figure 4.9(a), (b) and (c), we can see that our method for star patterns dominates the other methods because we capture the dependencies among properties in the queries. The RDF-3x method also have better results on query set $Qs_1$ in which query patterns have the small number of properties and literal constraints. In this case, the join uniformity assumption does not affect much. Conversely, in query sets $Qs_2$ and $Qs_3$, queries have more properties and literal constraints. It is more likely that the join uniformity assumption affects the accuracy of estimation. Thus our Bayesian network-based method obtains much better results on query sets $Qs_2$ and $Qs_3$ than the other two methods that adopt the join uniformity assumption.

![Figure 4.10: Relative errors for chain query patterns.](image)

**Performance of methods for chain query patterns.** From Figure 4.10(a), (b) and (c), we can see performance of methods on chain query patterns. “CHM” indicates our chain histogram based method for estimating the selectivity of chain patterns. Our method dominates the other methods. The RDF-3x method also have better results on query set $Qc_1$ in which the lengths of chain query patterns are relatively small. Conversely, in query sets $Qc_2$ and $Qc_3$, lengths of chain query patterns are longer. It is more likely that the join uniformity assumption affect the accuracy of estimation. Thus our method obtains much better results...
on query sets $Q_{c2}$ and $Q_{c3}$ than the other two methods.

**Performance of methods for composite query patterns.** In this experiment, we evaluate our method for composite query patterns. We developed 20 composite query patterns for each dataset. All these queries are also grouped into 4 query sets: $Q_{sc1}$, $Q_{sc2}$, $Q_{sc3}$, $Q_{sc4}$. Some details of query sets are shown in Figure 4.8. Figure 4.11(a), (b) and (c) show the results of four methods on three datasets, where “Greedy” and “Optimal” stand for two decomposition algorithms (shown in Algorithm 4.3 and Algorithm 4.4 respectively). Our methods obtain more accurate estimation since we construct the refined model when dealing with joined triple patterns and do not adopt join uniformity assumption. The optimal algorithm also outperforms the greedy algorithm because the latter may not find the maximum path cover of query patterns. And for the uncovered part of queries, greedy algorithm would adopt the independence assumption for selectivity estimation, which affects the estimation accuracy. However, figures show that Greedy algorithm still performs better than PF and RDF-3x methods.

![Figure 4.11: Relative errors for composite query patterns.](image-url)
Figure 4.12: Online running time

### 4.5.4 Online Running Time

In this experiment, we study the on-line running time of our estimation methods adopting two decomposition algorithms: Greedy algorithm and Optimal algorithm on YAGO dataset which is relatively heterogeneous. Figure 4.12 shows the average running time of two methods as the average edge number of query sets increases. The running time of the method with the Greedy algorithm reveals a slow growth with the increase of the average edge number of queries sets. We can observe that the time cost is reasonable even when the average edge number of query set $Q_{sc4}$ rises to 11. Compared with the Greedy algorithm, the running time of the method with the Optimal algorithm is higher because the Optimal algorithm tries to find the maximum path cover of each composite query pattern. The time costs of two cases are close when graph patterns are small (the number of edge $\leq 8$), but the running time of Optimal algorithm rises significantly
when the number of edge exceeds 8. It suggests the Greedy decomposition algorithm is preferable for larger and dense graph patterns.

4.6 Conclusion

With the increasing and Future work amount of RDF data, efficient and scalable management of RDF data has become a fundamental challenge to achieve the semantic web vision. Selectivity estimation is critical to processing RDF queries efficiently. To address this problem, we employ the Bayesian network and chain histogram techniques for estimating the selectivity of star and chain query patterns. For an arbitrary composite graph pattern, the algorithm which combines the precomputed statistics of chain and star paths to estimate the overall selectivity is proposed. The experiments show that our method outperforms existing approaches in accuracy significantly. In the future, we will consider some advanced features of SPARQL queries such as OPTIONAL and UNION operations and investigate the appropriate way to generalize our methods to support other applications in SPARQL query processing.
Chapter 5

Query Relaxation for RDF Queries

Database users may be frustrated by no answers returned when they pose a query on the database. In this chapter, we study the problem of relaxing queries on RDF databases in order to acquire approximate answers. We address two problems for efficient query relaxation. First, to ensure the quality of answers, we compute the similarities of relaxed queries with regard to the original query and use them to score the potential relevant answers. Second, for obtaining top-k answers, we develop two algorithms. One is based on the best-first strategy and relaxed queries are executed in the ranking order. The other executes the relaxed queries as a batch and avoids unnecessary execution cost. At last, we implement and experimentally evaluate our approaches.
5.1 Motivation

In the last few years, the amount of RDF (Resource Description Framework) data has grown by several orders of magnitude. Some RDF data sets such as DBPedia \(^1\) \cite{BLK+09}, Uniprot \(^2\) and Geonames \(^3\) with billions of entity descriptions are made available on the Web. Query languages on RDF data such as SPARQL \(^4\) have been developed. Since the use of RDF to represent data has grown dramatically, query processing on RDF data becomes an important issue in realizing the semantic web vision.

Normally, users expect quality results and pose a query with strict conditions over the database as a start. However, as users are not very clear about the contents and the data distribution of the database, such a strict query often leads to the Few Answers Problem: the user query is too selective and the number of answers is not enough. In this case, users may undergo a “trial-and-error” process of changing the query conditions manually for obtaining approximate answers. In the worst case, they need to consider the combinations of all possible query conditions. Obviously, this is a time-consuming and frustrating task.

The effective way to cope with this problem is query relaxation: weakening the query conditions automatically while preserving the intention of users as much as possible when the query fails to produce enough answers. The process of query relaxation can provide users with a means to automatically identify new queries that are related to the user query. These new queries can return generalized or neighbourhood information by relaxing the original query conditions.

In contrast to conventional databases, structures of RDF databases do not come in the shape of well-defined schemas but in terms of semantic annotations.

\(^1\)http://dbpedia.org/About
\(^2\)http://www.uniprot.org/
\(^3\)http://www.geonames.org
\(^4\)http://www.w3.org/TR/rdf-sparql-query/
that confirm to a schema called RDF schema (or RDFs ontology shown in Figure 5.1), which describes the taxonomies of classes and properties as well as relationships between them. In this case, the terms (classes or properties) used in the user query can be generalized according to the ontology information and the specific user query can be rewritten to a more general one to capture potential approximate answers.

![Academic Staff ontology](image)

**Figure 5.1: Academic Staff ontology**

For example, a user wants to retrieve 10 proceedings editors who acquired doctoral degree from University of Queensland (UQ) and work for Swinburne University (Swin) from a given RDF database that confirms to the RDFs ontology shown in Figure 5.1. For this purpose, the user may issue the SPARQL query $Q$ shown in Figure 5.2 on the database.

Suppose that the number of exactly matched answers is not enough ($< 10$), we need to relax query $Q$ for obtaining approximate answers. Since ontology information is available in the RDF database, $Q$ can be relaxed through replacing the specific classes or properties in $Q$ to more general ones. For example, property “DegreeFrom” is the super property of “doctoralDegreeFrom”, so we can replace “doctoralDegreeFrom” in triple pattern $(?x, \text{doctoralDegreeFrom}, \text{'UQ'})$ with “DegreeFrom” and get the relaxed query $Q_1$, which is more general than $Q$. 
Similarly, user query $Q$ can also be relaxed to other queries such as $Q_2$, $Q_3$, $Q_4$, $Q_5$ and so on.

**Query Q:**

```sparql
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX abc: <http://www.w3.org/abc.owl#>

SELECT ?X ?Y
WHERE {
  ?X abc:doctoralDegreeFrom abc:UQ.
  ?X abc:worksFor abc:Swin.
  ?X abc:ProceedingsEditorOf ?Y.
}
```

**Relaxed queries of Q**

- $Q_1$:
  ```sparql
  SELECT ?X ?Y
  WHERE {
    ?X abc:degreeFrom abc:UQ.
    ?X abc:worksFor abc:Swin.
    ?X abc:ProceedingsEditorOf ?Y.
  }
  ```

- $Q_2$:
  ```sparql
  SELECT ?X ?Y
  WHERE {
    ?X abc:doctoralDegreeFrom abc:UQ.
    ?X abc:ProceedingsEditorOf ?Y.
  }
  ```

- $Q_3$:
  ```sparql
  SELECT ?X ?Y
  WHERE {
    ?X abc:degreeFrom abc:UQ.
    ?X abc:worksFor abc:Swin.
    ?X abc:EditorOf ?Y.
  }
  ```

- $Q_4$:
  ```sparql
  SELECT ?X ?Y
  WHERE {
    ?X abc:doctoralDegreeFrom abc:UQ.
    ?X abc:worksFor abc:Swin.
    ?X abc:ProceedingsEditorOf ?Y.
  }
  ```

- $Q_5$:
  ```sparql
  SELECT ?X ?Y
  WHERE {
    ?X abc:degreeFrom abc:UQ.
    ?X abc:worksFor abc:Swin.
    ?X abc:ProceedingsEditorOf ?Y.
  }
  ```

Figure 5.2: A SPARQL query $Q$ and its relaxed queries.

From this example, we can observe that there are many directions for relaxing the user query and the number of possible relaxed queries is large. Returning all approximate answers to users is neither time efficient nor necessary. To avoid overwhelming users with a big number of answers, it is desirable to obtain a ranked list of approximate answers. In this chapter, we take the user query and map it to a top-$k$ query with a ranking function depending on given conditions in the user query. The key questions are:

- How to design an effective ranking function to score approximate answers
accord ing to their semantic relevance to the user query?

• How to obtain top-$k$ approximate answers over RDF databases efficiently?

We address these problems and make the following contributions:

1. We use information content to measure the semantic similarity between the corresponding terms used in the user query and a relaxed query, and combine similarity values of terms to score the relaxed query. The score of an approximate answer depends on the maximum score of the relaxed query it can match.

2. To obtain top-$k$ approximate answers, we propose a relaxation algorithm, which is based on the best-first strategy and executes relaxed queries in the ranking order.

3. To improve the performance of obtaining enough approximate answers, we devise a more efficient relaxation algorithm which executes the relaxed queries as a batch in order to avoid unnecessary execution cost. To predict the size of batch, we study the problem of selectivity estimation in the context of RDF query relaxation and present the method to estimate the selectivity for a set of relaxed queries precisely.

The remainder of this chapter is organized as follows. In Section 5.2, we introduce some preliminary knowledge. In Section 5.3, we discuss the ranking function for relaxed queries. In Sections 5.4 and 5.5, two relaxation algorithms for obtaining top-$k$ approximate answers are proposed. One is based on the best-first search; the other executes the relaxed queries as a batch in order to avoid unnecessary execution cost. We present the experimental evaluation of our approach in Section 5.6. At last, we conclude and discuss the future work.
CHAPTER 5. QUERY RELAXATION FOR RDF QUERIES

5.2 Background

5.2.1 Query Relaxation Model

Hurtado et al. in [HPW06] propose two kinds of relaxation for triple pattern which exploit RDF entailment to relax the queries.

**Simple relaxation on triple pattern:** The simple relaxation exploits the RDF simple entailment to relax a triple pattern. The definition of simple relaxation is presented in Chapter 3.3. It consists only of replacing terms of the original triple pattern to variables. For example, (?x, type, Lecturer) can be relaxed to (?x, type, ?y) by replacing “Lecturer” with the variable “?y”.

**Ontology relaxation on triple pattern:** This type of relaxation exploits RDFS entailment in the context of an ontology. The definition of ontology relaxation is given in Chapter 3.3. It includes relaxing type conditions and properties such as: (1) replacing a triple pattern (a, type, b) with (a, type, c), where (b, sc, c) ∈ closure(onto). For example, given the ontology in Figure 5.1, the triple pattern (?x, type, Proceedings) can be relaxed to (?x, type, Book). (2) replacing a triple pattern (a, p, b) with (a, p, b), where (p, sp, p) ∈ closure(onto). For example, the triple pattern (?x, ReviewerOf, 'WWW') can be relaxed to (?x, ContributerOf, 'WWW').

**Definition (Relaxed Triple Pattern):** Given a triple pattern q, q’ is a relaxed pattern obtained from q, denoted by q ≺ q’, through applying a sequence of one or more of the two types of relaxations: simple relaxation and ontology relaxation.

**Definition (Relaxed Query Pattern):** Given a user query Q(q1, q2, · · · , qn) and a query Q’(q’1, q’2, · · · , q’n), we call Q’ is a relaxed query of Q, denoted by Q ≺ Q’, if there exists one pair (qj, q’j) s.t. qj ≺ q’j and ∀i ∈ [1, n], i ̸= j, either qj = q’j or qj ≺ q’j.
CHAPTER 5. QUERY RELAXATION FOR RDF QUERIES

Definition (Approximate Answers): Given a user query \( Q \), if \( Q' \) is a relaxed query of \( Q \) namely \( Q \prec Q' \), \( r \in Answers(Q') \) and \( r \notin Answers(Q) \) such that \( Answers(Q') \) and \( Answers(Q) \) are answer sets of query \( Q' \) and \( Q \), then we call \( r \) is an approximate answer of query \( Q \).

5.2.2 Problem Definition

Problem Definition. Given a user query \( Q \) which returns too few answers, to capture approximate answers, user query \( Q \) is relaxed to \( Q_1, Q_2, \cdots, Q_n \) through simple relaxation and ontology relaxation, such that \( Q \prec Q_i, i \in [1..n] \). We are concerned with finding the top-\( k \) approximate answers efficiently.

5.3 Ranking Model

Since the number of relaxed queries could be large, returning all answers of these relaxed queries is not desirable. Given a user query \( Q \) and its relaxed queries \( Q_1 \) and \( Q_2 \), if \( Q_1 \) is ranked prior to \( Q_2 \), then we return the answers of \( Q_1 \) prior to the answers of \( Q_2 \). In [HPW06], the authors introduced a ranking model for the relaxed queries. Given two relaxed queries of the original user query, if one is subsumed by the other, then the former relaxed query is better. However, this ranking model fails to rank two relaxed queries in the case when there is no subsume relationship between them. In this section, we present a more fine-grained ranking model for the relaxed queries and approximate answers.

5.3.1 Similarity of Relaxed Triple Patterns

Suppose that \( Q_1 \) is a relaxed query of user query \( Q \). Intuitively, the more similar \( Q_1 \) is to \( Q \), the better \( Q_1 \) is. Thus, we rank the relaxed queries based on their
similarities to the original query. A SPARQL query is a set of triple patterns, so we first discuss how to measure the similarity between the original triple pattern and its relaxed triple patterns.

Note that an RDF triple pattern can be represented as a node-arc-node link with two nodes (subject, object) and one arc (property). Hence we introduce the similarity between nodes as well as arcs, then combine them to measure the similarity of the relaxed triple patterns.

**Similarity between nodes:** Suppose that one node (subject or object) in a triple pattern is a class \( c_1 \) and it is relaxed to its super class \( c_2 \) through ontology relaxation. Now we need to compute the similarity between two classes \( c_1 \) and \( c_2 \).

Several similarity measures for concepts have been proposed such as information content measure [Res95], distance based measure. In this chapter, we use information content to measure the similarity between two classes \( c_1 \) and \( c_2 \). The information contained by a class \( c_i \) is its instances in the database. We use \( \Pr(c_i) \) to denote the probability of encountering an instance of class \( c_i \) in the database. Clearly, a more general class has the higher probability of encountering an instance. According to information theory, the information content of class \( c_i \), \( I(c_i) \) can be quantified as the negative log likelihood, \( I(c_i) = -\log \Pr(c_i) \). For two classes \( c_1 \) and \( c_2 \), the information shared by two classes is indicated by the information content of the class that subsumes them. Here, since \( c_1 \) is a sub class of \( c_2 \), \( c_2 \) is just the class that subsumes \( c_1 \) and \( c_2 \) and we have:

\[
\text{Common}(c_1, c_2) = I(c_2) = -\log \Pr(c_2)
\]

Given two classes \( c_1 \) and \( c_2 \) such that \( c_1 \) is a sub class of \( c_2 \), we use the ratio between \( \text{Common}(c_1, c_2) \) and the information content of \( c_1 \) to define the similarity
\[ \text{Sim}(c_1, c_2) = \frac{\text{Common}(c_1, c_2)}{I(c_1)} = \frac{I(c_2)}{I(c_1)} = -\log \Pr(c_2) = -\log \Pr(c_1) \] (5.1)

If \( \Pr(c_1) \) is fixed, we have:

\[ \text{Sim}(c_1, c_2) \propto -\log \Pr(c_2) \propto \frac{1}{\Pr(c_2)} \]

According to the above formula, we can see that the value of \( \text{Sim}(c_1, c_2) \) is inversely proportional to \( \Pr(c_2) \). It agrees to the fact that the more concrete \( c_2 \) (with lower \( \Pr(c_2) \)) is, the more similar two classes are. Given a class \( c_i \), we compute \( \Pr(c_i) \) as follows:

\[ \Pr(c_i) = \frac{|\text{Instances}(c_i)|}{|\text{Instances}|} \]

where \( \text{Instances}(c_i) \) stands for the instance set of class \( c_i \) and \( \text{Instances} \) indicates the set of all instances in the database. \( |\text{Instances}(c_i)| \) and \( |\text{Instances}| \) are both statistic information, which can be easily obtained from the database in advance. Thus, similarity value of two classes would be computed efficiently.

In the case where a node (subject or object) in a triple pattern is relaxed to a variable through simple relaxation, we define the similarity of the two nodes to be 0. For example, \((?x, \text{type}, \text{Proceedings})\) is relaxed by \((?x, \text{type}, ?w)\), we have \( \text{Sim}(\text{“Proceedings”, “?w”})=0 \).

**Similarity between arcs:** If property \( p_1 \) in a triple pattern is relaxed to its super property \( p_2 \) through ontology relaxation, we use the similar method
for classes to measure the similarity of two properties. Pr($p_i$) stands for the probability of encountering a triple which matches property $p_i$ in the database. We have:

$$\text{Sim}(p_1, p_2) = \frac{\text{Common}(p_1, p_2)}{I(p_1)}$$

$$= \frac{I(p_2)}{I(p_1)} = \frac{-\log \Pr(p_2)}{-\log \Pr(p_1)}$$

(5.2)

Similarly, the value of $\text{Sim}(p_1, p_2)$ is inversely proportional to $\Pr(p_2)$. It also agrees to the fact that the more concrete $p_2$ (with lower $\Pr(p_2)$) is, the more similar two properties are. Given a property $p_i$, we compute $\Pr(p_i)$ as follows:

$$\Pr(p_i) = \frac{|\text{Triples}(p_i)|}{|T|}$$

where $\text{Triples}(p_i)$ means the triple set of property $p_i$ and $T$ is the set of all triples in the database. Obviously $|T|$ and $|\text{Triples}(p_i)|$ are two constants and can be easily obtained from the database in advance.

If a property in a triple pattern is relaxed to a variable through simple relaxation, we define the similarity of the two arcs as 0. For example, $(?x, \text{doctoralDegreeFrom}, 'UQ')$ is relaxed to $(?x, ?w, 'UQ')$, then we have $\text{Sim}(\text{"doctoralDegreeFrom"}, \text{"?w"})=0$.

**Similarity of triple patterns:** Given a triple pattern $q(s, p, o)$ and its relaxed triple pattern $q'(s', p', o')$, we combine the similarity between nodes and arcs to compute the similarity score $\text{Sim}(q, q')$ as follows:

$$\text{Sim}(q, q') = \frac{1}{3} \cdot \text{Sim}(s, s') + \frac{1}{3} \cdot \text{Sim}(p, p')$$

$$+ \frac{1}{3} \cdot \text{Sim}(o, o')$$

(5.3)
5.3.2 Similarity of Relaxed Queries

Now we discuss how to measure the selectivity of relaxed queries. Given a query pattern \( Q(q_1, q_2, \cdots, q_n) \) and its relaxed query \( Q'(q'_1, q'_2, \cdots, q'_n) \), we combine the similarity of triple patterns in the product form and compute the similarity \( \text{SimScore} \) between two queries as follows:

\[
\text{SimScore}(Q, Q') = \prod_{i=1}^{n} w_i \cdot \text{Sim}(q_i, q'_i)
\]  

(5.4)

where \( w_i \in (0, 1] \) is the weight of triple pattern \( q_i \), which could be specified by users to reflect the importance of \( q_i \) in \( Q \). The default value of \( w_i \) is 1.

There are some properties about our \( \text{SimScore} \) function:

**Proposition 1 (Monotonicity).** Given relaxed queries \( Q'(q'_1, q'_2, \cdots, q'_n) \) and \( Q''(q''_1, q''_2, \cdots, q''_n) \) of user query \( Q(q_1, q_2, \cdots, q_n) \), if \( \forall i \in [1, n], \text{Sim}(q_i, q'_i) \geq \text{Sim}(q_i, q''_i) \), then \( \text{SimScore}(Q, Q') \geq \text{SimScore}(Q, Q'') \).

**Proof.** Given \( \forall i \in [1, n] \) \( \text{Sim}(q_i, q'_i) \geq \text{Sim}(q_i, q''_i) \), since \( \prod \) is a monotone operator, we have \( \prod_{i=1}^{n} w_i \cdot \text{Sim}(q_i, q'_i) \geq \prod_{i=1}^{n} w_i \cdot \text{Sim}(q_i, q''_i) \), namely \( \text{SimScore}(Q, Q') \geq \text{SimScore}(Q, Q'') \). \( \square \)

**Proposition 2 (Consistency).** Given relaxed queries \( Q'(q'_1, q'_2, \cdots, q'_n) \) and \( Q''(q''_1, q''_2, \cdots, q''_n) \) of query \( Q \), if \( Q' \preceq Q'' \), then \( \text{SimScore}(Q, Q') \geq \text{SimScore}(Q, Q'') \).

**Proof.** If \( Q' \preceq Q'' \), then \( \forall i \in [1, n], q'_i \preceq q''_i \) or \( q'_i = q''_i \). Obviously, we have \( \text{Sim}(q, q'_i) \geq \text{Sim}(q, q''_i), i \in [1, n] \). According to Proposition 1, \( \text{SimScore} \) is a monotone function and we have \( \text{SimScore}(Q, Q') \geq \text{SimScore}(Q, Q'') \). \( \square \)
5.3.3 Score of Approximate Answers

We have presented the method to measure the similarity between a user query and its relaxed queries. SimScore can be used to score approximate answers which are returned by relaxed queries. The score of an approximate answer is the SimScore of the relaxed query it matches. Note that an approximate answer may match different relaxed queries, e.g., ('John', ReviewerOf, 'WISE') could match queries (?x, ReviewerOf, 'WISE') and (?x, ContributorOf, 'WISE'). To deal with this case, we define the score of an approximate answer as the highest SimScore it can obtain.

**Definition (Score of an approximate answer):** The score of an approximate answer is the maximum SimScore among all of the relaxed queries it matches.

5.4 Best-First Relaxation Algorithm

In this section, we focus on properly relaxing a user query for obtaining top-k approximate answers. To achieve it, a best-first relaxation algorithm is presented. We also characterize a type of unnecessary relaxed queries which do not contribute to the final answers and propose the method to prune them.

5.4.1 Basic Relaxation Algorithm

In this section, we will present a relaxation algorithm that ensures the desired quality of answers. The relaxation strategy (Best-First strategy) is to select the current best relaxed query from candidatures to execute on the database continually till enough answers are returned. In this way, relaxed queries are executed in a sequence based on their SimScore values. For instance, user
CHAPTER 5. QUERY RELAXATION FOR RDF QUERIES

query \( Q((?x, \text{doctoralDegreeFrom, 'UQ'}), (?x, \text{worksFor, 'Swin'}), (?x, \text{ProceedingsEditorOf, } ?y), (?y, \text{type, Proceedings})) \) (shown in Figure 5.2) has relaxed queries such as: 

\[ Q_1((?x, \text{DegreeFrom, 'UQ'}), (?x, \text{worksFor, 'Swin'}), (?x, \text{ProceedingsEditorOf, } ?y), (?y, \text{type, Proceedings})) \]

\[ Q_4((?x, \text{dotoralDegreeFrom, 'UQ'}), (?x, \text{worksFor, 'Swin'}), (?x, \text{ProceedingsEditorOf, } ?y), (?y, \text{type, Book})) \]

Since \( \text{SimScore}(Q, Q_1) \geq \text{SimScore}(Q, Q_4) \), which means that query \( Q_1 \) is more similar to the user query and hence should be executed first. Before giving the details of the relaxation algorithm, we first introduce the concept of the relaxation graph.

**Ranking relaxed triple patterns:** Given a user query denoted by \( Q(q_1^{(0)}, q_2^{(0)}, \ldots, q_n^{(0)}) \), we compute the possible relaxed triple patterns for each triple pattern of \( Q \) and rank them by their similarity scores. \( q_i^{(0)} \) indicates the original triple pattern and \( q_i^{(j)} \) indicates the \( j \)-th best relaxation of triple pattern \( q_i^{(0)} \). An example is shown in Figure 5.3. If the original triple pattern \( q_i^{(0)} \) is relaxed to \( q_i^{(j)} \), we say that triple pattern \( q_i^{(0)} \) goes forward \( j \) steps to \( q_i^{(j)} \). Obviously, if \( j_1 < j_2 \), then \( \text{Sim}(q_i^{(j_1)}) \geq \text{Sim}(q_i^{(j_2)}) \).

![Figure 5.3: Ranking of the relaxed triple patterns in Q with their Sim scores](image)

**Relaxation Graph:** The relaxation graph with respect to user query \( Q \) is a directed graph representing the relaxed queries of \( Q \). Each node in the relaxation graph is a relaxed query and the root node is query \( Q \). Given two query nodes
in the relaxation graph, we call $Q_2$ is $Q_1$'s *succeeding node* if $Q_2 \prec Q_1$ and there exists no relaxed query $Q'$ such that $Q_1 \prec Q'$ and $Q' \prec Q_2$. It means that $Q_2$ is a direct relaxed query of $Q_1$. $Q_1$ is also called $Q_2$'s *preceeding node*. For instance, in Figure 5.4, $Q_3(q^{(1)}_1, q^{(0)}_2, q^{(1)}_3, q^{(0)}_4)$ is a succeeding node of $Q(q^{(0)}_1, q^{(0)}_2, q^{(0)}_3, q^{(0)}_4)$. An edge from node $Q_1$ to $Q_2$ exists if and only if $Q_2$ is $Q_1$'s succeeding node. All relaxed queries can be categorized into different levels according to the lengths of the paths from the root (user query $Q$) to relaxed queries. An example of query relaxation graph is shown in Figure 5.4.

The relaxation graph has some properties as follows:

**Property 5.1.** Given a user query $Q(q^{(0)}_1, q^{(0)}_2, \cdots, q^{(0)}_n)$, if queries $Q_i(q^{(i_1)}_1, q^{(i_2)}_2, \cdots, q^{(i_n)}_n)$, $Q_j((q^{(j_1)}_1, q^{(j_2)}_2, \cdots, q^{(j_n)}_n)$ are relaxed queries of $Q$, and $Q_j$ is a succeeding node of $Q_i$ in the relaxation graph, then $SimScore(Q, Q_i) \geq SimScore(Q, Q_j)$. And there exist at least one relaxed query at level $h$ better than all relaxed queries at level $h + 1$.

**Proof.** Since $Q_j$ is a succeeding node of $Q_i$, $\forall q^{(0)}_k \in Q$, we have $q^{(i_k)}_k = q^{(j_k)}_k$ or $q^{(i_k)}_k \prec q^{(j_k)}_k$ and $Sim(q^{(0)}_k, q^{(i_k)}_k) \geq Sim(q^{(0)}_k, q^{(j_k)}_k)$. $SimScore$ is a monotone function, so $SimScore(Q, Q_i) \geq SimScore(Q, Q_j)$, which means that relaxed queries are better than their succeeding nodes in the relaxation graph. Obviously, queries at level $h + 1$ are succeeding nodes of queries at level $h$, hence the best one at level $h$ must be better than all relaxed queries at level $h + 1$. □

**Property 5.2.** Given a user query $Q(q^{(0)}_1, q^{(0)}_2, \cdots, q^{(0)}_n)$, the $h$-th best relaxed query $Q'$ of $Q$ must be at level $h$ or less.

**Proof.** We use the method of mathematical induction to prove it. When level = 1, the relaxed query with the highest $SimScore$ must fall in level 1 according to property 5.1. Now we assume that when level $= h$, it holds that the relaxed query with the $h$-th highest $SimScore$ falls at level $h$ or less. Given level $= h + 1$, we can find at least one relaxed query at level $h + 1$ has higher
We exploit Property 5.1 and Property 5.2 to define the following best-first relaxation algorithm that executes relaxed queries in the ranking order and returns approximate answers incrementally. From Property 5.2, we know that the relaxed query with the highest $SimScore$ must be in the first level. Thus we first get the succeeding nodes of the original query and choose the query with the highest $SimScore$ to execute on the database. If the number of answers obtained is not enough, then we select the next best relaxed query. This algorithm is described in Algorithm 5.1 $BFS$-$Relaxation$. It first inserts all child nodes (direct relaxed queries) of query $Q$ into priority queue $PQ$ according to their $SimScore$ values (Line 2), then executes the best relaxed query and inserts its child nodes into $PQ$ (Line 4 - Line 6). Repeat this process till $k$ answers are obtained.

In our running example shown in Figure 5.4, user query $Q((\texttt{?x, doctoralDegreeFrom, 'UQ'}, (?x, worksFor, 'Swin'), (?x, ProceedingsEditorOf, ?y), (?y,

Figure 5.4: The relaxation graph of $Q$
CHAPTER 5. QUERY RELAXATION FOR RDF QUERIES

Algorithm 5.1 BFS-Relaxation

Input: Query \( Q \); \( k \) (the number of answers required);
Output: top-\( k \) approximate answers of query \( Q \);

1: \( Answers = \emptyset \); Priority queue \( PQ \);
2: Insert \( Q \)'s child nodes into \( PQ \); \{according to their SimScore values\}
3: \textbf{while} \( |Answers| < k \) \textbf{and} (not empty \( PQ \)) \textbf{do}
4: \textbf{Remove}(\( PQ, Q' \));
5: Insert child nodes of \( Q' \) to \( PQ \);
6: Add answers of \( Q' \) to \( Answers \)
7: \textbf{end while}
8: \textbf{return} \( Answers \);

\( type, Proceedings \)) is executed and its succeeding relaxed queries \( Q_1\)-\( Q_4 \) are inserted into the priority queue \( PQ \). If the number of answers returned by \( Q \) is not enough, we select relaxed query \( Q_1(((?x, DegreeFrom, 'UQ'), (?x, works-For, 'Swin'), (?x, ProceedingsEditorOf, ?y), (?y, type, Proceedings)) \) with highest SimScore from \( PQ \) and execute it on the database. After that, insert succeeding relaxed queries of \( Q_1 \) into \( PQ \). This process is repeated till enough answers are obtained or priority queue \( PQ \) is empty.

5.4.2 Pruning Unnecessary Relaxed Queries

In BFS-Relaxation algorithm, the current best relaxation in Candidates set is selected to execute in each round. However, some relaxed queries may not contribute to generate new answers compared with the former executed queries.

For instance, in our running example, \( Q_4(((?x, DegreeFrom, 'UQ'), (?x, works-For, 'Swin'), (?x, ProceedingsEditorOf, ?y), (?y, type, Book)) \) is a relaxed query of query \( Q \), through replacing the term “Proceedings” in triple pattern \( (?y, type, Proceedings) \) to its superclass “Book”. However, \( Q_4 \) would not generate new answers compared with the answer set of \( Q \). Because triple pattern \( (?y, type, Proceedings) \) has join dependency with triple pattern \( (?x, ProceedingsEditorOf, Proceedings) \).
?y) on variable ?y. Though “Proceedings” in triple pattern (?y, type, Proceedings) is relaxed to “Book”, the range of property “ProceedingsEditorOf” in triple pattern (?x, ProceedingsEditorOf, ?y) is still “Proceedings” (according to the RDFs ontology defined in Figure 5.1). Thus, this relaxed query would never generate new answers.

The BFS-Relaxation algorithm relaxes the classes or properties in the triple patterns of the query locally. It fails to consider the join dependency between the triple patterns. Meanwhile, triple patterns are relaxed only based on ontology information without considering the data distribution in the database. Thus, it is possible that the relaxed triple pattern has no new answers in the database.

There are two cases where relaxed queries could not contribute to new answers. One arises in generalizing the subject or object term in queries, and the other arises in generalizing the property in queries.

We first define the relation “⊂” on classes and properties. If class c1 is the subclass of c2, we have c1 ⊂ c2. Similarly, if property p1 is the sub property of p2, we have p1 ⊂ p2. We also define “subclasses(c)” as the set of sub classes of class c, and “subproperties(p)” the set of sub properties of property p.

Case 1: Given a query Q(q1, · · · , qi(?x, type, c1), · · · , qj(?x, p2, o2) or (s2, p2, ?x), · · · , qn), its succeeding relaxed query Q′(q1, · · · , qi′(?x, type, c), · · · , qj′(?x, p2, o2) or (s2, p2, ?x), · · · , qn) is obtained from query Q through replacing class c1 in qi to class c in q′i, where c1 ⊂ c, p2 is a property.

In this case, query Q is relaxed to Q′ through replacing class c1 in qi to its super class c in q′i. Obviously, if c has no new instances compared with class c1, then relaxed query Q′ would not generate new answers with respect to Q. Thus, we need to check whether the super class has more instances than its sub class. For class c1 and its super class c, it is easy to acquire the statistics: |Instances(c1)| and |Instances(c)|, the number of instances of classes c1 and c.
from the database. If \(|\text{Instances}(c)| > |\text{Instances}(c_1)|\), then the relaxed query may have the chance to generate new answers. Otherwise, it is guaranteed that the relaxed query will not generate new answers compared to query \(Q\).

Now we consider the join dependency between triple patterns in query \(Q\). We define the set difference \(\Delta\omega(c, c_1)\) between subclasses \((c)\) and subclasses \((c_1)\) as follows:

\[
\Delta\omega(c, c_1) = \text{subclasses}(c) - \text{subclasses}(c_1)
\]

If \(\exists \epsilon \in \Delta\omega(c, c_1)\) and \(\epsilon \prec \text{domain}(p_2)\) (or \(\text{range}(p_2)\)), then the relaxed query may have new answers generated. Otherwise, the relaxed query has no new answers generated compared to query \(Q\).

**Case 2:** Given a query \(Q(q_1, \ldots, q_i(?x, p_i, o_i), \ldots, q_j(?x, p_j, o_j), \ldots, q_n)\), its succeeding relaxed query \(Q'(q_1, \ldots, q'_i(?x, p, o_i), \ldots, q_j(?x, p_j, o_j), \ldots, q_n)\) is obtained by replacing the property \(p_i\) in triple pattern \(q_i\) to super property \(p\) in \(q'_i\).

Notice that triple pattern \(q_i(?x, p_i, o_i)\) has join dependency with triple pattern \(q_j(?x, p_j, o_j)\) on variable “?x”. And we define \(\Delta\omega(p, p_i) = \text{subproperties}(p) - \text{subproperties}(p_i)\), which is the set difference of sub properties set of \(p\) and \(p_i\).

If \(\bigcup_{p_s \in \Delta\omega(p, p_i)} \text{subclasses}(\text{domain}(p_s)) \cap \text{subclasses}(\text{domain}(p_j)) = \emptyset\), then relaxed query \(Q'\) will not generate new answers.

It is straightforward to generalize this method to deal with the case where triples \(q_i\) and \(q_j\) take the form \((s_i, p_i, ?x)\) and \((s_j, p_j, ?x)\).

Now we give the optimized relaxation algorithm described in Algorithm 5.2 **OBFS-Relaxation**, which checks the usefulness of relaxed queries and skip those unnecessary relaxed queries. It first inserts all child nodes of query \(Q\) into priority queue \(PQ\) according to their \(\text{SimScore}\) values (Line 2). For the current best relaxed query \(Q_b\), we execute \(Q_b\) and check its child nodes. If child node \(Q_k\) is
Algorithm 5.2 OBFS-Relaxation

**Input:** Query User $Q$; $k$ (the number of answers required);

**Output:** top-$k$ approximate answers of query $Q$;

1. $\text{Answers} = \Omega$; Priority queue $PQ$;
2. Insert $Q$’s succeeding nodes into $PQ$ and Label them unprocessed;
3. while ($|\text{Answers}| < k$) and (not empty $PQ$) do
4. \begin{align*}
5. \quad & \text{Remove}(PQ, Q_b); \\
6. \quad & \text{for all } Q_b’s \text{ succeeding nodes } Q_k \text{ do} \\
7. \quad & \quad \text{Insert } Q_k \text{ to } PQ; \\
8. \quad & \quad \text{Check the necessity of } Q_k; \\
9. \quad & \quad \text{if } Q_k \text{ is an unnecessary relaxation of } Q_b \text{ then} \\
10. \quad & \quad \quad \text{Label } Q_k \text{ processed}; \{Q_k \text{ does not need to execute}\} \\
11. \quad & \quad \text{else} \\
12. \quad & \quad \quad \text{Label } Q_k \text{ unprocessed}; \\
13. \quad & \quad \end{align*}
14. \begin{align*}
15. \quad & \text{if } Q_b \text{ is unprocessed then} \\
16. \quad & \quad \text{Execute}(Q_b); \\
17. \quad & \quad \text{Add answers of } Q_b \text{ to } \text{Answers}; \\
18. \quad & \quad \text{Label } Q_b \text{ processed}; \\
19. \quad & \quad \end{align*}
20. \textbf{return } \text{Answers};

unnecessary, we label it processed (Line 5 - Line 13) and the processed relaxed queries would be skipped. Repeat this process till $k$ answers are obtained.

### 5.5 Batch-based Relaxation Algorithm

A user query is often relaxed to multiple relaxed queries for acquiring top-$k$ approximate answers. We can observe that Algorithm BFS-Relaxation executes relaxed queries in the ranking order (according to their SimScore values). However, in-order execution could lead to some unnecessary costs. For example, in Figure 5.5, suppose user query $Q$ has no answers and we want to acquire top-50 approximate answers for query $Q$. Then, four relaxed queries $Q_1$, $Q_2$, $Q_{12}$, $Q_{21}$...
of $Q$ are executed in the ranking order, such that $Q_1 \prec Q_{12}$ and $Q_2 \prec Q_{21}$. From Figure 5.6, we can observe that the execution of $Q_1$ can be skipped, since $Q_1 \prec Q_{12}$ and the answers of $Q_1$ are fully contained by the answers of $Q_{12}$.

![Relaxation Graph](image)

Figure 5.5: The relaxation graph of $Q$

For accelerating the top-$k$ query processing, we choose to execute relaxed queries as a batch and skip some unnecessary relaxed queries for obtaining top-$k$ answers. To achieve it, the key problem here is to estimate the size of batch, i.e., the number of relaxed queries needed to obtain top-$k$ answers. We will discuss how to estimate the size of batch, and then describe the batch-based relaxation algorithm in the following sections.

### 5.5.1 Predicting the Size of Batch

We present a method to predict the size of batch for top-$k$ query processing. Clearly, it is expensive to obtain the number of relaxed queries by executing them in the database. We resort to find a way to estimate the size of the result set of relaxed queries in the database without executing them.

- Selectivity estimation for relaxed queries
Chapter 5. Query Relaxation for RDF Queries

Given a query $Q(q_1, q_2, \cdots, q_n)$, we have proposed the method to estimate the selectivity (the size of result set) of a single query $Q$ in Chapter 4. Here, we have to solve the problem of selectivity estimation in the context of RDF query relaxation. Let us denote a set of relaxed queries with respect to user query $Q$ as $R_Q = \{Q_i | Q_i$ is a relaxed query of $Q\}$. Then, formally, we consider the following problem: Given a user query $Q$ and a set of relaxed queries $R_Q$, estimate the selectivity $sel(R_Q)$.

There exists a loose lower and upper bounds of $sel(R_Q)$ as follows:

$$\max_{Q_i \in R_Q} sel(Q_i) \leq sel(R_Q) \leq \sum_{Q_i \in R_Q} sel(Q_i)$$

If $\exists Q_k \in R_Q, \forall Q_j \in R_Q, Q_j \prec Q_k$, then $sel(R_Q)$ is equal to the lower bound. On the other hand, if $\forall Q_i, Q_j \in R_Q, i \neq j, Q_i$ and $Q_j$ are disjoint, then $sel(R_Q)$ is equal to the upper bound. Normally these two conditions do not hold. Thus, we need to get a good estimation when there are some overlappings among relaxed
queries.

In combinatorial mathematics, the inclusion-exclusion principle states that if \(sel(Q_1)\) and \(sel(Q_2)\) are estimated selectivities of relaxed queries \(Q_1\) and \(Q_2\), respectively, then

\[
sel(Q_1 \cup Q_2) = sel(Q_1) + sel(Q_2) - sel(Q_1 \cap Q_2)
\]

where \(sel(Q_1 \cap Q_2)\) is the overlapping selectivity of \(Q_1\) and \(Q_2\).

For the general case of the principle, let \(sel(Q_i)\) \((i \in [1,n])\) be the estimated selectivity of relaxed query \(Q_i\). We have:

\[
sel(\bigcup_{i=1}^{n} Q_i) = \sum_{i=1}^{n} sel(Q_i) - \sum_{i,j:1 \leq i < j \leq n} sel(Q_i \cap Q_j) + \sum_{i,j,k:1 \leq i < j < k \leq n} sel(Q_i \cap Q_j \cap Q_k) - \ldots + (-1)^{n-1} sel(Q_1 \cap \cdots \cap Q_n)
\] (5.5)

From the above formula, we can see that for estimating the answer number of a set of relaxed queries \(Q_1, \ldots, Q_n\) precisely, we have to get a good estimation of the overlappings among relaxed queries, i.e., \(sel(Q_1 \cap \cdots \cap Q_n)\). Now we give some useful properties for estimating the selectivity of the overlappings among relaxed queries:

**Property 5.3.** Given two relaxed queries \(Q_i\) and \(Q_j\), if \(Q_i \prec Q_j\), then \(sel(Q_i \cup Q_j) = sel(Q_j)\).

**Property 5.4.** Given two relaxed queries \(Q_i\) and \(Q_j\), if \(Q_i \prec Q_j\), then \(sel(Q_i \cap Q_j) = sel(Q_i)\).

Property 5.3 and Property 5.4 are easy to understand. Since \(Q_i \prec Q_j\), the answers of \(Q_i\) are fully covered by \(Q_j\). Hence \(sel(Q_i \cup Q_j) = sel(Q_j)\). On the
other hand, the common answers they share are just the answers of $Q_i$. So $sel(Q_i \cap Q_j) = sel(Q_i)$.

Figure 5.7: The common ancestor of triple patterns and relaxed queries.

**Definition (Least Common Ancestor of Relaxed Triple Patterns):**
Given two relaxed triple patterns $q^{(i)}$ and $q^{(j)}$, we call $q^{(L)}$ is the least common ancestor of $q^{(i)}$ and $q^{(j)}$, if $q^{(L)} \prec q^{(i)}$, $q^{(L)} \prec q^{(j)}$ and $\exists q^{(L')} \text{ s.t. } q^{(L)} \prec q^{(L')}$, $q^{(L')} \prec q^{(i)}$, $q^{(L')} \prec q^{(j)}$.

Obviously, if $q^{(i)} \prec q^{(j)}$ (or $q^{(j)} \prec q^{(i)}$), then $q^{(L)} = q^{(i)}$ (or $q^{(L)} = q^{(j)}$). Figure 5.7(a) shows two relaxed triple patterns with their least common ancestor.

Similarly, we have the definition of least common ancestor of relaxed queries as follows:

**Definition (Least Common Ancestor of Relaxed Queries):**
Given two relaxed queries $Q_i$ and $Q_j$, we call $Q_L$ is the least common ancestor of $Q_i$ and $Q_j$ in the relaxation graph, if $Q_L \prec Q_i$, $Q_L \prec Q_j$ and $\exists Q' \text{ s.t. } Q_L \prec Q'$, $Q' \prec Q_i$ and $Q' \prec Q_j$. 
According to the above definition, for two relaxed queries \( Q \) of properties \( p \), the least common ancestor \( \mathcal{L}q \) of \( q_1, q_2 \) and matches triple patterns \( \mathcal{Q} \), \( q_L \) is the least common ancestor of \( q_{k_1}^{(i_k)} \) and \( q_{k_2}^{(j_k)} \).

**Lemma 5.1.** If triple \( t \) matches triple patterns \( q_1 \) and \( q_2 \), then \( t \) must match the least common ancestor \( q_L \) of \( q_1 \) and \( q_2 \).

**Proof.** Let us first look at the trivial case. If \( q_1 \prec q_2 \) (or \( q_2 \prec q_1 \)), then \( q_L = q_1 \) (or \( q_L = q_2 \)). Given triple \( t \) matching triple patterns \( q_1 \) and \( q_2 \), \( t \) must match \( q_L \). In the case where \( q_1 \not\prec q_2 \) and \( q_2 \not\prec q_1 \), suppose \( q_1, q_2 \) have forms \((?x, p_1, o_1)\) and \((?x, p_2, o_2)\), respectively. We know that there exist subsume relationships between properties \( p_1 \) and \( p_2 \), and between objects \( o_1 \) and \( o_2 \). If \( p_1 \) is subsumed by \( p_2 \) and \( o_2 \) is subsumed \( o_1 \), then \( q_L \), as the least common ancestor of \( q_1 \) and \( q_2 \), has form \((?x, p_1, o_2)\). Given that \( t \) matches triple patterns \( q_1 \) and \( q_2 \), namely \((?x, p_1, o_1)\) and \((?x, p_2, o_2)\), we can see that \( t \) has form \((s, p_1, o_2)\), and matches \( q_L \). In the same way, the conclusion can be obtained in the case when \( p_2 \) is subsumed by \( p_1 \) and \( o_1 \) is subsumed \( o_2 \).

We can generalize this lemma to the situation of query patterns that have a set of triple patterns:

**Lemma 5.2.** If a tuple (joined triples) \( \text{tup} \) matches relaxed queries \( Q_1 \) and \( Q_2 \), then \( \text{tup} \) must match the common ancestor of \( Q_1 \) and \( Q_2 \), denoted by \( Q_L \).

**Proof.** Since \( \text{tup} \) matches \( Q_1 \) and \( Q_2 \), \( \forall \) triple \( t_k \in \text{tup} \), it must match triple patterns \( q_{k_1}^{(i_k)} \) and \( q_{k_2}^{(j_k)} \) in \( Q_1 \) and \( Q_2 \). According to Lemma 5.1, \( t_k \) must also matches their common ancestor \( q_k^{(i_k)} \) in \( Q_L \). Thus, \( \text{tup} \) must match all triple patterns in \( Q_L \), namely, match \( Q_L \).

**Property 5.5.** Given two relaxed queries \( Q_1 \) and \( Q_j \), if \( Q_1 \not\prec Q_j \) and \( Q_j \not\prec Q_i \), then \( \text{sel}(Q_1 \cap Q_j) = \text{sel}(Q_L) \), such that \( Q_L \) is the least common ancestor of \( Q_1 \) and \( Q_j \) in the relaxation graph.
CHAPTER 5. QUERY RELAXATION FOR RDF QUERIES

Proof. Since $Q_L$ is the least common ancestor of $Q_i$ and $Q_j$ in the relaxation graph, it holds that $Q_L \preceq Q_i$ and $Q_L \preceq Q_j$. $\forall r_m \in \text{Answers}(Q_L)$ (answer set of $Q_L$), we have $r_m \in \text{Answers}(Q_i)$, $r_m \in \text{Answers}(Q_j)$, i.e., $r_m \in \text{Answers}(Q_i \cap Q_j)$. So $\text{Answers}(Q_L) \subseteq \text{Answers}(Q_i \cap Q_j)$. Conversely, $\forall r_n \in \text{Answers}(Q_i \cap Q_j)$, according to Lemma 5.2, it is easy to prove $r_n \in \text{Answers}(Q_L)$. So $\text{Answers}(Q_i \cap Q_j) \subseteq \text{Answers}(Q_L)$. At last we have $\text{Answers}(Q_i \cap Q_j) = \text{Answers}(Q_L)$, which means $\text{sel}(Q_i \cap Q_j) = \text{sel}(Q_L)$. □

Using properties 5.3, 5.4 and 5.5, we can easily estimate the selectivity of a set of relaxed queries. For example, suppose that we have the user query $Q$ and its relaxed queries shown in Figure 5.5, then:

$$\text{sel}(Q_1 \cup Q_2 \cup Q_{12} \cup Q_{21}) = \text{sel}(Q_{12} \cup Q_{21}) = \text{sel}(Q_{12}) + \text{sel}(Q_{21})$$

$$- \text{sel}(Q_{12} \cap Q_{21}) = \text{sel}(Q_{12}) + \text{sel}(Q_{21}) - \text{sel}(Q)$$

- Predicting the Size of Batch

Given a user query $Q_0$ and its possible relaxed queries $Q_1, Q_2, \cdots, Q_n$ (which are ranked in order), suppose we want to obtain top-$k$ answers. We use $|\bigcup_{i=0}^{m} Q_i|$ to indicate the real number of answers returned by $Q_1, \cdots, Q_m$ and $\hat{\text{sel}}(\bigcup_{i=0}^{m} Q_i)$ indicates our estimation for it.

If $\hat{\text{sel}}(\bigcup_{i=0}^{bs-1} Q_i) < k$ and $\hat{\text{sel}}(\bigcup_{i=0}^{bs} Q_i) \geq k$, then we can estimate the batch size as $bs$. However, the batch size predicted is an estimation. Thus, we have to consider the following two cases: the estimated batch size is smaller or larger than the real size. Suppose $|\bigcup_{i=0}^{m} Q_i| = k$, so the real batch size should be $m$. If $\hat{\text{sel}}(\bigcup_{i=0}^{bs} Q_i) \geq k$, $bs < m$, it means that the estimated batch size is smaller than the real one and extra batches are needed to obtain enough answers. As
a consequence, relaxed queries which have the subsume relationship may fall in different batches thus lowering the possibility of reducing execution costs. The extreme situation is that relaxed queries are executed one by one in the ranking order, which is essentially the best-first relaxation algorithm. Thus, the efficiency of batch-based relaxation algorithm should be better than the best-first relaxation algorithm.

If \( \widehat{sel}(\bigcup_{i=0}^{m} Q_i) < k \) (the number of answers returned by \( Q_1, \ldots, Q_m \) is underestimated), the estimated batch size \( bs \) would be larger than the real size \( m \). In this case, some executed queries are not supposed to be executed, which may result in an extra cost. To avoid this case of overestimating the batch size as much as possible, we set a discount factor \( \gamma \) to adjust the estimated batch size according to heuristic information from recently executed queries.

Suppose that \( Q_0, \ldots, Q_j \) are executed queries before we predict the batch size such that \( Q_0 \) is the user query. \( |Q_i| \) means the number of answers of \( Q_i \). We assume that user query \( Q_0 \) is executed before we relax it. Thus, initially \( |Q_0| \) is available. With the process of relaxation, there are more \( |Q_i| \) available. Since the estimated batch size depends on estimation of the number of answers returned by queries, we employ \( |\bigcup_{i=0}^{j} Q_i| \) and \( \widehat{sel}(\bigcup_{i=0}^{j} Q_i) \) of such executed queries as heuristic information to adjust the estimation of batch size. If \( \widehat{sel}(\bigcup_{i=0}^{j} Q_i) < |\bigcup_{i=0}^{j} Q_i| \), it means that the estimated number of answers returned by recently executed queries is smaller than the real one, which indicates that an overestimation of the current batch size is possible. Thus, we use the ratio: \( \frac{\widehat{sel}(\bigcup_{i=0}^{j} Q_i)}{|\bigcup_{i=0}^{j} Q_i|} < 1 \) as the discount factor to adjust the estimation of batch size. We have:

\[
\gamma = \begin{cases} 
\frac{\widehat{sel}(\bigcup_{i=0}^{j} Q_i)}{|\bigcup_{i=0}^{j} Q_i|}, & \text{if } \widehat{sel}(\bigcup_{i=0}^{j} Q_i) < |\bigcup_{i=0}^{j} Q_i|; \\
1, & \text{otherwise.}
\end{cases}
\] (5.6)
Algorithm 5.3 Predict

**Input:** Query $Q_0$; $k$ (the number of answers required);

**Output:** The estimated batch size $bs$;

1: $\eta = 0$;
2: $sel = 0$;
3: Compute discount factor $\gamma$; {according to Equation (5.6) on the preceding page}
4: while $sel < k$ do
5: $\eta += 1$;
6: Select the best $\eta$ relaxed queries behind $Q_0$; {not including $Q_0$}
7: $sel = sel(\bigcup_{j=1}^{\eta} Q_j)$; {according to Equation (5.5) on page 94 and properties 5.3, 5.4 and 5.5}
8: end while
9: return $bs = \lfloor \gamma \cdot \eta \rfloor$;

Note that with the process of relaxation, we can update discount factor $\gamma$ dynamically.

The process of estimating the number of relaxed queries needed (batch size) is presented in Algorithm 5.3 Predict. It first computes discount factor $\gamma$ (Line 3), then add the best $\eta$ relaxed queries to the batch. If $sel < k$, it increases $\eta$ and continues this process till $sel >= k$ (Line 4 - Line 8). At last, it returns the predicted batch size $bs = \lfloor \gamma \cdot \eta \rfloor$.

### 5.5.2 Batch-Based Relaxation Algorithm

We devise an optimized relaxation algorithm. This process is described in Algorithm 5.4 Batch-Relaxation. The idea is to take the relaxed queries as a batch and avoid the execution of some relaxed queries subsumed by others for reducing the overall execution cost. The algorithm is divided into three steps. In the first step, we predict the batch size, namely, the number of relaxed queries needed (Line 5). The predicted relaxed queries is inserted into the priority queue $PQ$ according to their $SimScore$ scores (Line 6). We check whether there exists $Q' \prec Q''$ in $PQ$. If $Q' \prec Q''$, we label $Q'$ processed (Line 8 - Line 12). In the second step
Algorithm 5.4 Batch-Relaxation

Input: Query $Q_0$, $k$ (the number of answers required);
Output: top-$k$ approximate answers of query $Q_0$;

1: $Answers = \Omega$; Priority queue $PQ$; $BatchSize=0$;
2: $k = k - |Q_0|$;
3: $currentQuery = Q_0$;
4: repeat
5: $BatchSize=\text{Predict}(currentQuery, k - |Answers|)$;
6: Insert $BatchSize$ best relaxed queries of $Q$ into $PQ$;
7: Label all elements in $PQ$ unprocessed;
8: for all $Q_k \in PQ$ do
9: if $\exists Q_j \in \text{Candidates}$ s.t. $Q_k \prec Q_j$ then
10: Label $Q_k$ processed;
11: end if
12: end for
13: while not empty $PQ$ do
14: $Remove(PQ, Q_i)$;
15: $currentQuery = Q_i$;
16: if $Q_i$ is unprocessed then
17: Execute $Q_i$;
18: Add answers of $Q_i$ to $Answers$;
19: end if
20: end while
21: until $|Answers| \geq k$
22: Rank the results in $Answers$: {according to the best relaxed query they directly match}
23: return top-$k$ results in $Answers$;

(Line 13 - Line 20), we skip the execution of queries which are labeled processed.
If the predicted relaxed queries return the answers less than we need, iterate this process until we get enough answers. In the last step (Line 22 - Line 23), we rank the answers according to the best relaxed queries that the corresponding triples (which contains the answers) can match and return them.

We illustrate this algorithm using the example in Figure 5.5 and Figure 5.6. We first predict that four relaxed queries $Q_1$, $Q_{12}$, $Q_2$, $Q_{21}$ are needed. And in this query set, we have $Q_1 \prec Q_{12}$ and $Q_2 \prec Q_{21}$. Thus, in the query execution step, $Q_1$ and $Q_2$ are skipped and query $Q_{12}$, $Q_{21}$ are executed. If the number of
obtained answers is not sufficient, repeat this process till we get enough answers.

5.6 Experiments

Experiment setup. We run all algorithms on a windows XP professional system with P4 3G CPU and 8 GB RAM. All algorithms are implemented using Jena TDB \(^5\), which provides for large scale storage and query of RDF datasets.

<table>
<thead>
<tr>
<th>Table 5.1: Dataset information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of triples</td>
</tr>
<tr>
<td>Number of classes</td>
</tr>
<tr>
<td>Number of properties</td>
</tr>
<tr>
<td>Number of instances</td>
</tr>
</tbody>
</table>

Data sets. We generate the dataset based on the Lehigh University Benchmark LUBM [GPH04]. It consists of a university domain ontology (containing 32 properties and 43 classes). The generated dataset contains about 10M distinct triples. Some details of the dataset are shown in Table 5.1.

Query Loads. There are three kinds of query patterns used in our experiments: star query patterns, chain query patterns and composite query patterns. A star query pattern has the form of a number of triple patterns with different properties sharing the same subject. A chain query pattern consists of a sequence of triple patterns where the object of a triple pattern is also the subject of the next triple pattern. A composite query pattern is a combination of star and chain patterns. We developed 7 queries (shown in Figure 5.8) such that \(Q_1, Q_2, Q_3\) are star query patterns; \(Q_4, Q_5\) are chain query patterns and \(Q_6, Q_7\) are composite query patterns.

Techniques. We conduct some experiments to evaluate the performance of

\(^5\)http://openjena.org/wiki/TDB
### Queries

<table>
<thead>
<tr>
<th>Query</th>
<th>Description</th>
</tr>
</thead>
</table>

Figure 5.8: Queries used in the experiments
our proposed three algorithms: Algorithm \textit{BFS-Relaxation} (BFSR), \textit{optimized BFS-Relaxation} (OBFSR) and \textit{Batch-Relaxation} (BR). Figure 5.9, Figure 5.10 and Figure 5.11 show the performance of three algorithms for 7 queries with the increase of $k$ (number of approximate answers). In Figure 5.9(a), Figure 5.10(a) and Figure 5.11(a), Y-axis is the running time. In Figure 5.9(b), Figure 5.10(b) and Figure 5.11(b), Y-axis is the number of executed relaxed queries.

In the first experiment, we fixed the number of answers $k=10$ and performed the 7 queries on the dataset. Figure 5.9(a) and Figure 5.9(b) show the execution time and the number of executed relaxed queries for each query. In the cases of queries $Q_2$, $Q_3$ and $Q_7$, we can see that the performance of OBFSR algorithm is better than basic BFSR algorithm. OBFSR algorithm can reduce the relaxation steps and execution time compared with basic BFSR algorithm. We use query $Q_2$ as an example to illustrate how OBFSR algorithm reduces the relaxation steps. Query $Q_2$ is relaxed to $Q_2'$ through replacing “Professor” in triple pattern (?y \texttt{rdf:type} Professor) to its super class “Person”. However, in
triple pattern (?y ub:researchInterest 'Research2') of Q2, the domain of property “ub:researchInterest” is still “Professor”. Thus, replacing “Professor” to “Person” will not generate new answers compared to the answers of query Q2 and Q2’ is an unnecessary relaxed query to Q2.

![Figure 5.10: Performance of our algorithms (Top-50)](image)

It also shows that in the cases Q2 and Q7 where the number of executed relaxed queries is more than 3, BR algorithm outperforms BFSR algorithm and OBFSR algorithm. In these cases, BFSR and OBFSR algorithm execute the relaxed queries in the ranking order, while BR algorithm executes the relaxed queries as a batch and can skip the execution of some relaxed queries that subsumed by others. When the number of executed relaxed queries is relatively small, such as the cases of Q1, Q4 and Q6 (which have only one relaxed query executed), there is no relaxed query subsumed by others in a batch. Thus the performance of BR algorithm is almost the same as that of BFSR algorithm.

From Figure 5.10 and Figure 5.11, we can see that when increasing k to 50 and 150, BR algorithm performs much better than BFSR and OBFSR algorithm.
Figure 5.11: Performance of our algorithms (Top-150)

Figure 5.11(b) shows that BR algorithm can reduce the number of executed relaxed queries for $Q_2$ from 16 to 4, and 10 to 3 for $Q_7$, which are significant reductions. With the increase of $k$, more relaxed queries are needed to execute. It is more likely that relaxed queries are subsumed by others in a batch. BR algorithm could skip these relaxed queries and reduce the overall execution time. Thus BR algorithm is suggested in the case when users expect more approximate answers. However, OBFSR algorithm still has its advantage. Since it executes the relaxed queries in order according to their SimScore values and approximate answers are returned incrementally, users can see early answers in a short time.

Table 5.2 shows the scores of approximate answers. Since approximate answers in an answer set are returned by many relaxed queries, the score of the answer set is an interval. The upper bound is the highest score that an answer set can have and the lower bound is the lowest score that the answer set can have. We can see that with the increase of $k$, the lower bound of answer sets decrease.
5.7 Summary

In this chapter we addressed the issue of relaxing a user query on RDF databases and computing approximate answers. We measured the similarity degrees of the relaxed queries with regard to the original user query and designed two algorithms to obtain top-$k$ approximate answers. The first algorithm is based on best-first strategy and relaxed queries are executed in order. We also characterized a type of unnecessary relaxed queries which do not contribute to the final answers and proposed the method to prune them. The other algorithm executes the relaxed queries as a batch and avoids the unnecessary execution cost. The experiments validated our approach. A further optimization is subject to our future work such as multiple query optimization on a sequence of relaxed queries by reusing the intermediate answers of selection or join operations.
Chapter 6

Query Evaluation on
Probabilistic RDF Data

Over the last few years, RDF has been used as a knowledge representation model in a wide variety of domains. Some domains are full of uncertainty. Thus, it is desirable to process and manage probabilistic RDF data. The core operation for queries on the RDF probabilistic database is computing the probabilities of the answers. In this chapter, we describe a general framework for supporting SPARQL queries on probabilistic RDF databases. In particular, we consider transitive inference capability for RDF queries. We show that the Find operation for an atomic query with the transitive property can be formalized as the problem of computing path expressions on the transitive relation graph and we also propose an approximate algorithm for computing path expressions efficiently. At last, we implement and experimentally evaluate our approach.
6.1 Motivation

The Resource Description Framework (RDF) is the proposal of the W3C for a standard metadata model to describe resources on the semantic web. An increasing amount of data is becoming available in RDF format. In real applications, there are many domains that are full of uncertainty. We face large volumes of data generated with uncertainty. For example, in biology science, probabilistic links between concepts can be obtained from various prediction techniques and probabilistic links are mutually independent. Thus, it is desirable to process and manage probabilistic RDF data.

To process and manage probabilistic RDF data, the central problem is query evaluation on RDF probabilistic databases. In a traditional database system, answers to a query are distinctive. However, in a probabilistic database, the system not only returns the answers but also computes probabilities for answers. Query evaluation on the probabilistic relational databases [FR97, GGH98, DS07] and probabilistic XML databases [AS06, HGS03] has been studied. However, this problem has not been addressed in RDF databases. In this chapter, we propose a framework for supporting SPARQL queries on the probabilistic RDF database. And we also consider query evaluation on RDF data with transitive properties.

For example, in Figure 6.1, a SPARQL query is posed on a probabilistic diseases database. The database consists of 7 RDF triples with probabilities indicating the confidence about the relationship. The query asks for the diseases that are associated with cough and cause of fatigue. We use superscript “\(t\)” to denote “transitive”. “AssociatedWith\(^t\)” means that \(AssociatedWith\) is a transitive property. Now the problem is how to compute the probability for each answer. We will interpret this example in detail for introducing the main concepts and techniques in this chapter.
We define the RDF probabilistic diseases database based on the possible worlds semantics. We use $\text{triples}(R)$ to indicate the set of all triples in the database. A database instance $I$ is a subset of $\text{triples}(R)$ (i.e., $I \subseteq \text{triples}(R)$) with a certain probability. It is assumed that all triples in the database are probabilistically independent, which implies that $\Pr(t_1, t_2, \cdots, t_n) = \Pr(t_1) \cdot \Pr(t_2) \cdots \Pr(t_n)$, where $t_1, t_2, \cdots, t_n$ are events of triples. For instance, in Table 6.1, $I_1 = \{t_1\}$, $\Pr(I_1) = \Pr(t_1) \cdot (1 - \Pr(t_2)) \cdot (1 - \Pr(t_3)) \cdot (1 - \Pr(t_4)) \cdot (1 - \Pr(t_5)) \cdot (1 - \Pr(t_6)) \cdot (1 - \Pr(t_7)) = 0.000288$.

We now illustrate query evaluation on the probabilistic RDF database. We adopt the intentional query evaluation [FR97] that associates each triple with a probabilistic event. The intermediate and result tuples are associated with a complex event through the operations of $\text{Find}(\sigma)$, $\text{Join}(\bowtie)$ and $\text{Project}(\pi)$.

Consider the query in Figure 6.1 that consists of two atomic queries or called triple patterns: $q_1(\text{?X, CauseOf, Fatigue})$ and $q_2(\text{?X, Associatedwith\footnote{i}}\text{, Cough})$. 

![Figure 6.1: SPARQL query on a probabilistic RDF database](image-url)
Table 6.1: Database instances with probabilities

<table>
<thead>
<tr>
<th>Database instances</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>(I_0 = {})</td>
<td>((1 - \Pr(t_1)) \cdot (1 - \Pr(t_2)) \cdot (1 - \Pr(t_3)) \cdot (1 - \Pr(t_4)) \cdot (1 - \Pr(t_5)) \cdot (1 - \Pr(t_6)) \cdot (1 - \Pr(t_7)) = 0.000192)</td>
</tr>
<tr>
<td>(I_1 = {t_1})</td>
<td>(\Pr(t_1) \cdot (1 - \Pr(t_2)) \cdot (1 - \Pr(t_3)) \cdot (1 - \Pr(t_4)) \cdot (1 - \Pr(t_5)) \cdot (1 - \Pr(t_6)) \cdot (1 - \Pr(t_7)) = 0.000288)</td>
</tr>
<tr>
<td>(\ldots)</td>
<td>(\ldots)</td>
</tr>
<tr>
<td>(I_{127} = {t_1, t_2, t_3, t_4, t_5, t_6, t_7})</td>
<td>(\Pr(t_1) \cdot \Pr(t_2) \cdot \Pr(t_3) \cdot \Pr(t_4) \cdot \Pr(t_5) \cdot \Pr(t_6) \cdot \Pr(t_7) = 0.190512)</td>
</tr>
</tbody>
</table>

We first do the Find(\(\sigma\)) operation which extracts all triples that match the atomic queries. For \(q_1(\?X, \text{CauseOf}, \text{Fatigue})\), since triples \(t_7\) and \(t_6\) match \(q_1\) and we get the tuples “Pneumonia” and “Flu” with events \(t_7\) and \(t_6\) respectively (shown in Table 6.2).

Table 6.2: Events of the results of \(\sigma(\?X, \text{CauseOf}, \text{Fatigue})\)

<table>
<thead>
<tr>
<th>(?X)</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pneumonia</td>
<td>(t_7)</td>
</tr>
<tr>
<td>Flu</td>
<td>(t_6)</td>
</tr>
</tbody>
</table>

The second triple pattern \(q_2(\?X, \text{Associatedwith}^1, \text{Cough})\) is distinguished by its transitive property “\(\text{Associatedwith}^1\)”. Note that there are two transitive paths \(t_1 \rightarrow t_3 \rightarrow t_4, t_1 \rightarrow t_2\) between resources “Pneumonia” and “Cough”. The tuple “Pneumonia” is obtained when \(t_1 \land t_3 \land t_4\) or \(t_1 \land t_2\) is true. So we assign the complex event \((t_1 \land t_3 \land t_4) \lor (t_1 \land t_2)\) to the tuple “Pneumonia”. Similarly, there are four tuples that match \(q_2\) associated with events (shown in Table 6.3). We can observe that if two nodes are strongly connected, the event expression would be huge. We will discuss how to process the Find(\(\sigma\)) operation efficiently in Section 6.4.

Then for getting the results of \(\sigma(q_1) \bowtie \sigma(q_2)\), we join the tuples from two
Table 6.3: Events of the results of $\sigma(?X, Associated\text{with}^t, Cough)$

<table>
<thead>
<tr>
<th>$q_2$: ?x</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pneumonia</td>
<td>$(t_1 \land t_2) \lor (t_1 \land t_3 \land t_4)$</td>
</tr>
<tr>
<td>Flu</td>
<td>$t_5$</td>
</tr>
<tr>
<td>RSV</td>
<td>$t_4$</td>
</tr>
<tr>
<td>Bronchitis</td>
<td>$t_2 \lor (t_3 \land t_4)$</td>
</tr>
</tbody>
</table>

atomic queries using the logic operator “$\land$”, if the two tuples have the same value on the shared attributes. We obtain two possible answers “Pneumonia” and “Flu” associated with events $(t_7 \land t_1 \land t_2) \lor (t_1 \land t_2 \land t_3 \land t_4)$, $(t_6 \land t_5)$ respectively (shown in Table 6.4). The probabilities of answers are obtained by cumulating the probabilities of database instances that make the events of answers true. For example, database instance $I_{127} = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7\}$ can make the events of two answers true, so the probability of $I_{127}$ should be counted to the probabilities of two answers. Given the probabilistic distribution over database instances, computing the probability of event expressions is a NP-hard problem. We will employ binary decision diagram (BDD) [Bry86] for computing the probabilities of event expressions.

Table 6.4: Events of the results of $\sigma(q_1) \bowtie \sigma(q_2)$

<table>
<thead>
<tr>
<th>?x</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pneumonia</td>
<td>$t_7 \land ((t_1 \land t_2) \lor (t_1 \land t_3 \land t_4))$</td>
</tr>
<tr>
<td></td>
<td>$= (t_7 \land t_1 \land t_2) \lor (t_7 \land t_1 \land t_3 \land t_4)$</td>
</tr>
<tr>
<td>Flu</td>
<td>$t_6 \land t_5$</td>
</tr>
</tbody>
</table>

In this chapter, we discuss the problem of evaluating SPARQL queries on probabilistic RDF databases. We claim the following contributions:

- We propose a framework for supporting SPARQL query on probabilistic RDF databases.
- We consider transitive inference capability for RDF queries. We show that
the \textit{Find} operation for the atomic query with the transitive property can be formalized as the problem of computing path expressions on the transitive relation graph. In this case, an approximate algorithm for computing path expressions efficiently is proposed.

The remainder of this chapter is organized as follows. Some preliminary is given in Section 6.2. Section 6.3 describes the query evaluation on RDF probabilistic databases. In Section 6.4, we design the approximate algorithms on the transitive relation graph. Section 6.5 presents an experimental evaluation of our approach.

6.2 Preliminary

An RDF term is an URI or a literal or a blank node. An RDF triple \((s, p, o) \in (I \cup B) \times (I \cup B) \times (I \cup B \cup L)\) is called an RDF triple, where \(I\) is a set of IRIs (Internationalized URIs), \(B\) a set of blank nodes and \(L\) a set of literals. In the triple, \(s\) is called subject, \(p\) the property (or predicate), and \(o\) the object or property value. An RDF triple pattern \((s, p, o) \in (I \cup V) \times (I \cup V) \times (I \cup V \cup L)\), where \(V\) is a set of variables disjoint from the sets \(I\), \(B\) and \(L\). An RDF graph pattern \(G = (q_1, q_2, ..., q_n)\), \(q_i \in T\), where \(T\) is a set of triple patterns.

We assume the following: \(\text{Indvs} \subseteq U\) is a set of individuals (instances); \(C \subseteq U\) is the set of classes; \(\text{Props} \subseteq U\) is the set of properties. \(\text{Props}^t \subseteq \text{Props}\) is a set of transitive properties and \(\text{Props}^n \subseteq \text{Props}\) is a set of normal properties. We have \(\text{Props}^t \cup \text{Props}^n = \text{Props}\) and \(\text{Props}^t \cap \text{Props}^n = \emptyset\).

An RDF tuple is a partial function from variables to RDF terms. Note that an RDF tuple is different from an RDF triple (a statement implying a semantic relation between resources). Conversely, an RDF tuple does not carry meaning and it just maps some variables to some RDF terms.
Definition (RDF Relation) An RDF relation is a set of RDF tuples. It can also be represented as a table. Each row corresponds to an RDF tuple and each column is an attribute named by a variable. For instance, in Table 6.2, \( q_1.x \) is the variable of the relation containing tuples “Pneumonia” and “Flu”.

Definition (Probabilistic RDF Database) A probabilistic RDF database \( D \) is a finite set of probabilistic triples. We denote \( \text{triples}(D) \) as all triples in the database. Each triple is associated with a probability and a unique identifier (event). The triple \( t \) has the form \(< s, p, o, \Pr(t), \tau(t) > \) where \( s \in \text{Indvs}, p \in \text{Props} \cup \{ \text{rdf : type} \}, o \in \text{Indvs} \cup \text{L} \cup \text{C} \). \( s, p \) and \( o \) are subject, property and object of triple \( t \). \( \Pr : \text{triples}(D) \to [0, 1] \) is a probability function and \( \tau : \text{triples}(D) \to \text{strings} \) is a mapping form each \( t \) to a unique identifier which we call the event of triple \( t \) in this chapter.

We interpret the RDF probabilistic database \( D \) in terms of possible worlds. A database instance \( I_i \) is a subset of \( \text{triples}(D) \). We call all database instances \( \{I_1, I_2, \cdots, I_n\} \) possible worlds and we have \( \Pr(I_i) = \prod_{t \in I_i} \Pr(t) \cdot \prod_{t \notin I_i} (1 - \Pr(t)) \). The sum of all probabilities of the possible instances is 1, i.e., \( \sum_{I_i} \Pr(I_i) = 1 \).

A set of RDF triples can be represented as a graph. Now we give the concept of probabilistic RDF graph as follows:

Definition (Probabilistic RDF Graph) A probabilistic RDF graph is a labeled directed graph denoted by \( G = (N, E, \rho, \tau, \Pr) \), where

1. \( N \subseteq \text{Indvs} \) is a set of nodes;
2. \( E \) is the set of edges in \( G \) where each edge is a triple \( (s, p, o) \);
3. \( \rho \) is a property labeling function such that \( \rho(s, p, o) = p \);
4. \( \tau : E \to \text{Strings} \) is a mapping from edges to a unique identifier (event);
5. \( \Pr : E \to [0, 1] \) is a mapping from edges to a probability value.
CHAPTER 6. QUERY EVALUATION ON PROBABILISTIC RDF DATA 114

For example, Figure 6.1 shows the probabilistic RDF graph for the diseases RDF databases.

**Definition (Probabilistic Transitive Relation Graph)** A probabilistic transitive relation graph \( G_t = (N^t, E^t, \rho, \tau, \Pr) \) for transitive property \( p^t \) is a subgraph of the probabilistic RDF graph \( G \), where \( N^t \subseteq N, E^t \subseteq E \) and we have \( \forall (s_i, p_i, o_i) \in E^t, \rho(s_i, p_i, o_i) = p^t \).

For example, in Figure 6.1 the probabilistic transitive relation graph for the transitive property \( AssociatedWith^t \) is the part surrounded with dashed lines.

Since in probabilistic transitive relation graph \( G^t \), the property \( p^t \) is fixed, we denote an edge as \((s, o)\) instead of \((s, p^t, o)\) for brevity. Now we define the concept of path and reachability on \( G_t \). A path \( ps \) from node \( n_1 \) to \( n_k \) is a sequence of edges: \((n_1, n_2) - (n_2, n_3) - (n_3, n_4) - \cdots - (n_{k-1}, n_k)\). We say that \( n_k \) is reachable from \( n_1 \) if there exists a path from \( n_1 \) to \( n_k \) denoted by \( n_1 \mapsto n_k \), else \( n_1 \not\mapsto n_k \). Given a path \( ps \) from \( s \) to \( v \), we define the path expression \( \lambda(ps) \) as the conjunction of events of all edges in the path:

\[
\lambda(ps) = \bigwedge_{(n_i, n_j) \in ps} \tau(n_i, n_j) \quad (6.1)
\]

Given two nodes \( s \) and \( v \) on \( G^t \), the path expression \( T(s, v) \) from node \( s \) to \( v \) is defined as the disjunction of expressions of paths connecting \( s \) to \( v \):

\[
T(s, v) = \bigvee_{ps \in PS} \lambda(ps) \quad (6.2)
\]

where \( PS \) is the set of paths from \( s \) to \( v \).

**Definition (Query Semantics)** Let \( q \) be a query over the probabilistic RDF database \( D \). The database is given by a probability distribution over database instances. We denote \( \text{Answers}_q(I_j) \) as the set of results of \( q \) over the database.
instance $I_j$. For any result tuple $ans$, we have: $\Pr(ans) = \sum_{I_j} \Pr(I_j)$, where $I_j$ is any database instance s.t. $ans \in Answers_q(I_j)$.

### 6.3 Query Evaluation

We have shown the concepts of a probabilistic RDF database and we now address the problem of how to compute the answers with probabilities given a SPARQL query. We restrict our discussion to conjunctive SPARQL query with the rule form $Q \leftarrow q_1, q_2, q_3, \cdots, q_n$ where $Q$ is the head of the query and $q_1, q_2, q_3, \cdots, q_n$ are atomic queries (triple patterns). We use $\text{var}(Q)$ to denote the variables in $Q$. For example, the SPARQL query in Figure 6.1 can be written as the rule:  

$$Q(\?x) \leftarrow (\?x, \text{CauseOf}, \text{Fatigue}), (\?x, \text{AssociatedWith}, \text{Cough}).$$

#### 6.3.1 Intentional Query Evaluation

We adopt intentional query evaluation of [FR97]. The basic idea of this approach is to associate each triple with a basic probabilistic event. When processing the query, the intermediate and result tuples are associated with a complex event that is the combination of basic events through $\text{Find}(\sigma)$, $\text{Join}(\bowtie)$, and $\text{Project}(\pi)$ operations. We use $ee(t)$ to denote the complex event of tuple $t$ and then show that how the three operations associate the tuples with complex events as follows:

- $\text{Find}(\sigma)$

  The $\text{Find}(\sigma)$ takes an atomic query $q$ (triple pattern) and returns all tuples that match the atomic query. $q$ is of the form $(\?x, p, o)$, $(s, p, \?y)$ or $(\?x, p, \?y)$. There are two cases:

  **Case 1:** If a predicate term in $q$ is a normal property then $\text{Find}(\sigma)$ copies the events of input triples which satisfy the query $q$ to the output tuples i.e., if
CHAPTER 6. QUERY EVALUATION ON PROBABILISTIC RDF DATA

$q(t) = t'$ then

$$ee_{\sigma(q)}(t') = \tau(t)$$

where $t'$ is the output tuple induced by triple $t$. An example of $Find(\sigma)$ operation is shown in Table 6.2.

**Case 2:** In this case, the predicate term in the atomic query is a transitive property. The atomic query is of three possible forms $(a, p^t, ?obj)$, $(?sub, p^t, b)$ or $(?sub, p^t, ?obj)$ where $a$ and $b$ are literals and $p^t$ is a transitive property. We will show that the $Find$ operation can be formalized as the problem of computing the path expression between nodes on the transitive relation graph. Let $G^t = (N^t, E^t, \rho, \tau, Pr)$ be the probabilistic transitive relation graph for property $p^t$. If tuple $b$ is an answer of $(a, p^t, ?obj)$, there must exist a path $ps_i$: $(a, n_1) - (n_1, n_2) - \cdots - (n_k, b)$ from node $a$ to $b$ in $G^t$. It also implies that if events $\tau(a, n_1), \cdots, \tau(n_k, b)$ all are present, then the event $ee(b)$ of tuple $b$ is present. From formula 6.1 and formula 6.2, we get the path expression from node $a$ to $b$:

$$T(a, b) = \bigvee_{ps_i \in PS} \left( \bigwedge_{(n_i, n_j) \in ps_i} \tau(n_i, n_j) \right)$$

where $PS$ is the set of paths from $a$ to $b$. The $Find(\sigma)$ on the transitive graph can be formalized as the single source path expression problem on $G$. Similarly, computing the event of result tuple $a$ of $(?sub, p^t, b)$ or the event of result tuple $(a, b)$ of $(?sub, p^t, ?obj)$ can also be formalized as the single sink path expression problem or the all pairs path expression problem.

Generally, if the property in the atomic query is a transitive property, the $Find$ operation can be formalized as computing the path expression $T(s, v)$ between nodes $s$ and $v$ on the transitive relation graph. We will discuss on how to compute $T(s,v)$ efficiently in Section 6.4. An example of an atomic query with a transitive
property is shown in Table 6.3.

- Join($\bowtie$)

The $\text{Join}$(\$\bowtie\$) operation joins two relations on their shared attributes. $r_1 \bowtie r_2$ contains all combinations of a tuple from relation $r_1$ and a tuple from $r_2$ where they shared attributes are equal.

$$ee_{r_1 \bowtie r_2}(t', t) = ee_{r_1} \bowtie ee_{r_2}$$

An example of join operation is shown in Table 6.4.

- Project($\pi$)

The $\text{Project}$(\$\pi\$) restricts an RDF relation to a subset of its attributes. A tuple belongs to the result relation if at least one of its origin tuples belongs to the argument relation. The event expression of result tuple $t'$ can be represented as the disjunction of the complex events of original tuples:

$$ee_{\pi(A)}(t') = \bigvee_{t : \pi(A)(t) = t'} ee(t)$$

### 6.3.2 Representing Answers of Queries as DNF

In this section, we will show that the answers of a query can be represented as a disjunctive normal formula (DNF). Given a conjunctive query $Q \leftarrow q_1, q_2, q_n$, where $Q$ is the goal and $q_1, q_2, \ldots, q_n$ are subgoals. For obtaining answers, we do the $\text{Find}$ operation for every atomic query first and then do the $\text{Join}$ operation $(q_1 \bowtie q_2 \bowtie \cdots \bowtie q_n)$ between subgoals; we can get tuples with the form of $\bigwedge_{i=1}^n t_i$, where $t_i$ is one of the answers of $q_i$. We do the final $\text{Project}$ operation: $\prod_{\text{var}(Q)}(q_1 \bowtie q_2 \bowtie \cdots \bowtie q_n)$ at the end, which groups tuples based on
the values of attributes $\text{Var}(Q)$. Thus, the result tuples can be represented as the disjunctive normal formula $\bigvee_{j=1}^{n} c_j$, where $c_j$ is the conjunctive clause.

So our problem can be transferred to evaluate the probabilities of DNFs. Note that if a predicate term in the atomic query $q_i$ is a transitive property then the event expressions of answers $t_i$ may be the complex events i.e. DNFs. In this case, we need to convert the formulas to DNF formulas.

### 6.3.3 Computing the Probability of DNF Formula (BDD)

It is known that computing the probability of DNF formula is an $\#P$-hard problem, even if all variables are independent, as they are in our case. Some methods such as Monte Carlo method [KL83] is proposed for computing the probability of a DNF formula. In this chapter, we adopt the binary decision diagrams (BDDs)[Bry86] method which is used widely in digital-system design and combinatorial optimization.

A binary-decision diagram represents a DNF formula as a rooted directed acyclic graph. Each non-terminal vertex $v$ is labeled by a variable $\text{var}(v)$ and has arcs directing toward two children: $1o(v)$ (shown as a dashed line in Figure 6.2) corresponding to the case where the variable is assigned 0; $\hat{h}a(v)$ (shown as
Algorithm 6.1 *ComProbOfFormula*

**Input:** DNF formula \(d\)

**Output:** probability of \(d\);

1: construct the BDD \(\beta\) for \(d\);
2: get the root node \(v\) of BDD;
3: \(\Pr(v) = \text{ComputeProb}(\beta, v)\);
4: return \(\Pr(v)\);

1: **Function** \(\text{ComputeProb}(\text{BDD } \beta, \text{ vertex } v)\)
2: if \(v\) is the 1-terminal then
3: return 1;
4: else
5: if \(v\) is the 0-terminal then
6: return 0;
7: else
8: \(lo=\)the low children of \(v\);
9: \(hi=\)the high children of \(v\);
10: \(\Pr(hi) = \text{ComputeProb}(hi)\);
11: \(\Pr(lo) = \text{ComputeProb}(lo)\);
12: return \(\Pr(v) \cdot \Pr(hi) + (1 - \Pr(v)) \cdot \Pr(lo)\);
13: end if
14: end if

a solid line) corresponding to the case where the variable is assigned 1. Each terminal vertex is labeled 0 or 1. For example, Figure 6.2 shows a BDD for DNF formula \((t_1 \land t_2) \lor (t_1 \land t_3 \land t_4)\). For a given assignment to the variables, the probability value yielded by the DNF formula is determined by tracing a path from the root to a terminal vertex, following the branches indicated by the values assigned to the variables. This process of computing the probability of a DNF formula is given in Algorithm 6.1 *ComProbOfFormula*. Given a DNF formula, we first construct the BDD for this formula using the method in [Bry86] (Line 1 - Line 2), and then compute the probability of the formula by transversing the BDD constructed. If the current vertex \(v\) is not a terminal vertex, then we have \(\Pr(v) \cdot \Pr(hi) + (1 - \Pr(v)) \cdot \Pr(lo)\) (Line 8 - Line 12 of Function *ComputeProb*). Iterate this process until the current vertex is a terminal vertex.
CHAPTER 6. QUERY EVALUATION ON PROBABILISTIC RDF DATA

6.4 Evaluation of Atomic Query with Transitive Property

In Section 6.3.1, we have shown that if the predicate term in the atomic query is a transitive property $p^t$, then the Find operation can be formalized as computing the path expression between two nodes on the transitive relation graph. In this section, we will discuss how to process the Find operation on the transitive relation graph efficiently.

6.4.1 Storing the Transitive Relation Graph

We define the transitive relation as a partial order relation $\preceq_{p^t}$ of node pairs on $G^t$. Each directed edge $(x, y)$ in $G^t$ implies $x \preceq_{p^t} y$. According to transitive property, if there exists a path from node $x$ to $z$ in $G^t$, i.e. $x \leftrightarrow z$, then we have $x \preceq_{p^t} z$ such that $x$ is the ancestor of $z$ and $z$ is the descendant of $x$. For example, in Figure 6.3, node $a$ is the ancestor of node $g$ and $a \preceq_{p^t} g$.

Figure 6.3: A sample of transitive relation graph.
To accelerate the Find operation, we pre-compute the transitive closure of relation and store it. Some previous work has been done on storing the transitive closure relation [AKBLN89, Cas93]. Agrawal et.al [ABJ89] proposed to assign a node multiple intervals that encapsulate reachability information for its descendant nodes. Nodes of the graph can be covered by a spanning tree. Each node is assigned with an integer interval \([\text{start}, \text{end}]\) according to the visited orders in a depth-first traversal such that the integer “start” is the postorder number of the node and the integer “end” is the lowest postorder number among its descendants. The relation between node pairs can be checked by their overlapping intervals. Node \(x\) is an ancestor of node \(y\), only if \(y.\text{start} \geq x.\text{start}\) and \(y.\text{end} \leq x.\text{end}\).

For example, the nodes in Figure 6.3 are assigned integer intervals as shown in Figure 6.4. The dashed arcs are non-spanning tree arcs in the graph. Given two nodes \(a\) and \(e\), since the interval \([1, 4]\) of node \(e\) is overlapped by the interval \([1, 8]\) of node \(a\), we have \(a \prec_{\rho'} e\) and \(a \rightarrow e\).
6.4.2 Find Operation on Transitive Relation Graph

In this section, we discuss the Find operation (σ) on the transitive relation graph. As shown in Section 6.1, if the predicate term in the atomic query is a transitive property, the the Find operation can be formalized as computing the path expression $T(s,v)$ from nodes $s$ to $v$ in the transitive relation graph. We assume no cycles contained in the transitive relation graph.

We first consider the exact algorithm of Find operation on the transitive relation graph. It finds all paths from $s$ to $v$ for computing $T(s,v)$ by depth-first search. Since the reachability information between nodes is known (according to the encoding of nodes), it is possible to avoid transversing the whole graph. The algorithm first expands all successors $v_i$ of node $s$. If $v_i \rightarrow v$, it implies that there exists at least one path from $s$ to $v$ through $v_i$ and the algorithm should expand the successors of $v_i$. Conversely, if $v_i \not\rightarrow v$, it implies that there is no path from $s$ to $v$ through $v_i$ and the algorithm could stop expanding the successors of $v_i$.

The reachability information here is a kind of heuristic information used to focus searching on promising paths from $s$ to $v$. The time complexity of the algorithm is $O(b^h)$ where $h$ is the maximum length of the simple path from nodes $s$ to $v$ and $b$ is the branching factor of nodes.

When pairs of nodes are connected by multiple paths, it fails to discard substantial paths using reachability information. Furthermore, the path expression $T(s,v)$ would be large, which becomes computationally infeasible to compute the probability of the path expression. For example, in Figure 6.3 since node $a$ and node $g$ are strongly connected, it almost traverses all edges in the graph to compute $T(a,g) = t_1 \lor (t_2 \land t_3 \land t_4) \lor (t_{11} \land t_6 \land t_3 \land t_4) \lor (t_{11} \land t_7 \land t_4) \lor (t_{11} \land t_{10})$.

To improve the efficiency of computing $T(s,v)$, we propose an approximate algorithm. The basic idea is trying to obtain the incomplete path expression (denoted by $\tilde{T}(s,v)$) between two nodes, which is smaller than the complete
one with approximate probability. The approximate algorithm will estimate the error of incomplete path expression. If the estimated error is smaller than the approximation threshold $\epsilon$, then the algorithm will stop and return the incomplete path expression.

Note that the path expression between nodes $s$ and $v$ is a DNF formula, where $k$ is the number of paths from $s$ to $v$ and $\lor_{i=1}^{k} ps_i$ is the conjunctive clause representing the path expression of path $ps_i$. Now we give the definition of $\epsilon$-approximation path expression as follows:

**Definition ($\epsilon$-approximation path expression)** Given the path expression $T(s,v) = \lor_{ps_i \in PS} \lambda(ps_i)$, where $PS$ is the set of all paths from $s$ to $v$, we say that $\tilde{T}(s,v)$ is the $\epsilon$-approximation path expression of $T(s,v)$ if:

1. $\Pr(T(s,v)) - \Pr(\tilde{T}(s,v)) \leq \epsilon$;
2. $\tilde{T}(s,v) = \lor_{ps_i \in A} \lambda(ps_i)$, such that $A \subseteq PS$.

For example, in Figure 6.3, given $\epsilon = 0.03$, $T(a,b) = t_2 \lor(t_{11} \land t_6)$ and $\tilde{T}(a,b) = t_2$, since $\Pr(T(a,b)) = \Pr(t_2 \lor(t_{11} \land t_6)) = 0.924$, $\Pr(\tilde{T}(a,b)) = \Pr(t_2) = 0.9$, $\Pr(T(a,b)) - \Pr(\tilde{T}(a,b)) = 0.024 \leq 0.03$, $\tilde{T}(a,b) = t_2$ is the 0.03-approximation of $T(a,b)$.

We employ the iterative-deepening search to find the paths from $s$ to $v$. Iterative-deepening search combines breadth-first optimality with the low space complexity of depth-first search. In this way, we find the paths from $s$ to $v$ by running depth-first search repeatedly with a growing search depth. For each given search depth $h$, we define the incomplete path expression:

$$\tilde{T}_h(s,v) = \lor_{ps_i \in FoundedPathSet} \lambda(ps_i)$$

where $FoundedPathSet$ indicates the set of the paths from $s$ to $v$ that we have
Algorithm 6.2 ApproExpression

**Input:** Nodes $s$, $o$, approximation threshold $\varepsilon$;

**Output:** Approximate path expression $T(s, o)$;

1: $Upper = 1$, $Lower = 0$;
2: $FoundedPathSet = \Omega$; $PromisingPathSet = \Omega$;
3: $depth = 1$;
4: while $Upper - Lower > \varepsilon$ do
5:   for all path $ps_i$ from $s$ with length $= depth$ do
6:     if $ps_i$ is a promising path then
7:       $PromisingPathSet = PromisingPathSet \cup ps_i$;
8:       $d_{upper} = \bigvee_{ps_k \in PromisingPathSet} \lambda(ps_k)$
9:     end if
10:    if $ps_i$ is a path from $s$ to $o$ then
11:       $FoundedPathSet = FoundedPathSet \cup ps_i$;
12:      $PromisingPathSet = PromisingPathSet \cup ps_i$;
13:      $d_{lower} = \bigvee_{ps_k \in FoundedPathSet} \lambda(ps_k)$;
14:      $d_{upper} = \bigvee_{ps_k \in PromisingPathSet} \lambda(ps_k)$;
15:    end if
16:   end for
17: end while
18: return $T(s, o) = d_{lower}$;
Algorithm 6.3 ApproximateFind

Input: Atomic query $q$, approximation threshold $\varepsilon$;

Output: The result set $ResultSet$;

1: $ResultSet = \Omega$;
2: if $q$ has the form of $(?x, p', o)$ then
3:     for all $s \in N(G)$ do
4:         if $s \prec_{p'} o$ then
5:             path expression $T(s, o) = ApproExpression(s, o, \varepsilon)$ \{defined in Algorithm 6.2\};
6:             add $s$ with $T(s, o)$ to $ResultSet$;
7:         end if
8:     end for
9: end if
10: if $q$ has the form of $(s, p', ?y)$ then
11:     for all $o \in N(G)$ do
12:         if $s \prec_{p'} o$ then
13:             path expression $T(s, o) = ApproExpression(s, o, \varepsilon)$;
14:             add $o$ with $T(s, o)$ to $ResultSet$;
15:         end if
16:     end for
17: end if
18: if $q$ has the form of $(?x, p', ?y)$ then
19:     for all $s, o \in N(G)$ do
20:         if $s \prec_{p'} o$ then
21:             path expression $T(s, o) = ApproExpression(s, o, \varepsilon)$;
22:             add $(s, o)$ with $T(s, o)$ to $ResultSet$;
23:         end if
24:     end for
25: end if
26: return $ResultSet$;
founded. We also define the promising path as the path from \( s \) to \( v_i \) with length \( h \) and \( v_i \to v \), and \( PromisingPathSet \) as the set of the all promising paths from \( s \) with length=\( h \).

![Diagram of approximate Find algorithm on the transitive relation graph.](image)

Figure 6.5: Approximate Find algorithm on the transitive relation graph.

Now the problem is how to bound the error of incomplete path expression \( \tilde{T}_h(s,v) \) for each search depth \( h \). We construct two DNF formulas \( d_{lower} \) and \( d_{upper} \) to represent the lower bound and upper bound for \( \tilde{T}_h(s,v) \):

\[
d_{lower} = \bigvee_{ps_i \in FoundedPathSet} \lambda(ps_i)
\]

\[
d_{upper} = \bigvee_{ps_i \in FoundedPathSet \cup PromisingPathSet} \lambda(ps_i)
\]

Here, \( d_{lower} \) is the disjunction of expression of founded paths from \( s \) to \( v \) and \( d_{upper} \) is the disjunction of path expression of founded paths and promising paths from \( s \) to \( v \). For each depth \( h \), we have:

\[
Pr(d_{lower}) \leq Pr(\tilde{T}_h(s,v)) \leq Pr(d_{upper})
\]

We present the process of computing approximate path expression in Algorithm 6.2 \textit{ApproExpression}. In this algorithm, given the nodes \( s \), \( o \) and threshold
we employ the iterative-deepening search to find the paths from s to o. For each search depth we obtain the path set PromisingPathSet, FoundedPathSet, and compute the DNF formulas $d_{lower}$, $d_{upper}$ (Line 4 - Line 15). Then estimate the error of the current incomplete path expression (Line 16 - Line 17). If the error is smaller than $\varepsilon$, it stops and return such approximate path expression; otherwise, it continues until the error is smaller than $\varepsilon$.

The approximate Find algorithm is presented in Algorithm 6.3 ApproximateFind.

A triple pattern may have three forms. In Algorithm ApproximateFind, for each case ((?x, pt, o): Line 2 - Line 9; (s, pt, ?y): Line 10 - Line 17; (?x, pt, ?y): Line 18 - Line 25), we compute the approximate expression for the results and add them to the result set.

For example, to obtain the 0.01-approximation of $T(a, g)$ in the transitive relation graph (shown in Figure 6.3), the algorithm sets search depth=1 first (shown in Figure 6.5), we then get FoundedPathSet = \{(a, g)\}, PromisingPathSet = \{(a, b); (a, d)\}. So $d_{lower}$ = $t_1$, $d_{upper}$ = ($t_1 \vee t_2 \vee t_{11}$), $Pr(d_{lower})$ = $Pr(t_1)$ = 0.8, $Pr(d_{upper})$ = $Pr(t_1 \vee t_2 \vee t_{11})$ = 0.92. Since $Pr(d_{upper}) - Pr(d_{lower})$ = 0.12 > 0.01, we need to increase the search depth to 2.

When depth=2, we get FoundedPathSet=\{(a, g); (a, d)(d, g)\}, PromisingPathSet = \{(a, b)(b, e); (a, d)(d, b); (a, d)(d, e); (a, d)(d, h)\}. So $d_{lower}$ = $t_1 \vee (t_{11} \wedge t_{10})$, $d_{upper}$ = $t_1 \vee (t_2 \wedge t_3) \vee (t_{11} \wedge t_{10}) \vee (t_{11} \wedge t_6) \vee (t_{11} \wedge t_8) \vee (t_{11} \wedge t_7)$. Since $Pr(d_{upper}) - Pr(d_{lower}) < 0.01$, we can obtain $\tilde{T}_2(a, g) = t_1 \vee (t_{11} \wedge t_{10})$ is the 0.01-approximation of $T(a, g)$.

### 6.5 Experiments

In this section, we conduct experiments to verify our methods.

**Experiment setup.** The data is stored in and managed by Mysql 5.0. All
algorithms are implemented using Jena (http://jena.sourceforge.net/). We run all algorithms on a windows XP professional system with P4 3G CPU and 2 GB RAM. We use PSEpro from objectStore $^1$ as the storage engine for storing the encoded probabilistic transitive relation graph. We also use JavaBDD $^2$ for BDD operations.

**Data sets.** We generate the dataset based on the Lehigh University Benchmark LUBM [GPH04]. Each triple is assigned a probability. Probabilities of triples are mutually independent. We modify the data generator UBA1.7 and add a transitive property $\text{pre} \rightarrow \text{requisite}^t$. The domain and the range of $\text{pre} \rightarrow \text{requisite}^t$ are both class $\text{Course}$. The triples with a predicate $\text{pre} \rightarrow \text{requisite}^t$ are generated randomly.

<table>
<thead>
<tr>
<th>Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q_1$ Select ?X ?Y Where { ?X rdf:type ub:GraduateStudent. ?X ub:takesCourse ?Y. }</td>
</tr>
<tr>
<td>$Q_4$ Select ?X ?Y Where { ?X pre-requisite $^1$ GraduateCourse0. ?Y ub:teacherOf ?X. ?Y rdf:type ub:FullProfessor. }</td>
</tr>
</tbody>
</table>

Figure 6.6: Queries used in the experiments.

In the first experiment, we measure the running time for five queries (Shown $^1$http://objectstore.net $^2$http://javabdd.sourceforge.net/)
in Figure 6.6) developed. $Q_4$ and $Q_5$ contain transitive property $pre-requisite^t$. We pre-compute the transitive closure of relation and store the encoded transitive relation graph. We use the exact $Find$ algorithm (with $\varepsilon = 0$) for $Q_4$ and $Q_5$. The running time for five queries is shown in Figure 6.7. It shows that the running time of queries with transitive property increases significantly compared with those of the queries without transitive property, since most of the time is spent in the $Find$ operation on the transitive relation graph. This suggests it is desired to optimize the $Find$ operation on the transitive relation graph.

![Figure 6.7: Running time for queries.](image)

In the next experiment, we verify the efficiency of Algorithm 6.3 $ApproximateFind$ on the transitive relation graph. The dataset contains $150k$ triples and the transitive relation graph for $pre-requisite^t$ consists of 657 edges and 311 nodes. We vary $\varepsilon$ from 0.02 to 0.1. The running time and iterative depth of $Q_4$ and $Q_5$ is shown in Figure 6.8 and Figure 6.9. Note that it applies the exact find algorithm when $\varepsilon = 0$ and the approximate find algorithm when $\varepsilon > 0$. From Figure 6.8, it shows that Algorithm $ApproximateFind$ brings considerable time savings, since iterative depth decreases significantly (shown in Figure 6.9) and path expression generated is relatively small. It suggests that if the users have demands of time
restriction, the system could set an appropriate $\varepsilon$ for speeding up processing of the atomic queries with transitive property.

![Running Time Graph](image)

**Figure 6.8:** Running time for queries with increasing approximation $\varepsilon$.

![Search Depth Graph](image)

**Figure 6.9:** Search depth for queries with different approximation $\varepsilon$.

We also evaluate our approximate find algorithm on the transitive relation graph with edges ranging from 500 to 3000. The approximation $\varepsilon$ is fixed to 0.01. From Figure 6.10, we can see that the running time increases dramatically when the number of edge is more than 2000. It indicates that we need more
efficient algorithms in the case where the transitive relation graph is huge and dense, which is also our future work.

6.6 Conclusion

This chapter addressed the issue of supporting SPARQL queries on the probabilistic RDF databases. The core operation for queries on an RDF probabilistic database is computing the probabilities of the results to a user query. We developed the query evaluation framework based on possible world semantics. In addition, we considered transitive inference on RDF instance data. To accelerate the Find operation for the atomic query with transitive property, we proposed an approximation algorithm on the transitive relation graph. The experiments validate the effectiveness and efficiency of the algorithms proposed. The future work includes designing more powerful find algorithms, developing advanced probabilistic models such as Bayesian Networks, and dealing with distributed RDF data on the web.
Chapter 7

Thesis Conclusions

In this chapter, we summarize the major contributions of the thesis and propose a few interesting directions which may be further explored in future.

7.1 Summary of This Thesis

The series of works in this thesis focus on some issues about query processing on RDF data. We concentrate the problems of selectivity estimation, query relaxation and query evaluation on probabilistic RDF data. We summarize our contributions as the following aspects.

Firstly, we have studied how to estimate the selectivity for SPARQL graph queries, which is crucial to query optimization. Due to the fine-grained characteristic of RDF modeling, SPARQL queries normally contain the huge number of joins. Thus we focus on estimating the selectivity for joined graph patterns. We first study the two kinds of query patterns: star and chain patterns and propose two methods for estimating the selectivity of them respectively. For star query patterns, we construct Bayesian networks to compactly represent the joint probability distribution over values of correlated properties. And for chain query
patterns, we build the chain histogram, which can obtain a good balance between the estimation accuracy and space cost. For an arbitrary SPARQL query represented as a composite graph pattern, we propose algorithms for maximally combining the statistics of chain paths and star paths that we have precomputed to estimate the overall selectivity of the graph pattern. We conduct experiments on both synthetic and real-world datasets to show the effectiveness and performance of our approach.

Secondly, we have investigated how to relax the SPARQL queries to obtain approximate answers. In some real applications, database users may be frustrated by too few answers or even no answers returned when they pose a query on the database. We address two problems in efficient query relaxation. First, to ensure the quality of answers, we compute the similarities of relaxed queries with regard to the original query and use them to score the potential relevant answers. Second, for obtaining top-k approximate answers, we develop two efficient algorithms. The first algorithm is based on the best-first strategy and relaxed queries are executed in the ranking order. The other is a batch based algorithm that executes the relaxed queries as a batch and avoids unnecessary execution cost. We implement the proposed relaxation algorithms and conduct experiments to validate the efficiency of our algorithms.

Finally, we have studied the problem of query evaluation for SPARQL queries on probabilistic RDF data. As we know, in real applications, there are some domains that are full of uncertainty. Thus query evaluation on probabilistic RDF data is a very important issue. We describe a general framework for supporting SPARQL queries on probabilistic RDF databases. In particular, we consider transitive inference capability for RDF queries. We show that the Find operation for an atomic query with the transitive property can be formalized as the problem of computing path expressions on the transitive relation graph and
we also propose an approximate algorithm for computing path expressions efficiently. Experimental results validate the efficiency of the proposed approximate algorithm.

7.2 Future Work

We have discussed a series of works about query processing on RDF data. We now point out several interesting future directions.

The future work includes the query processing on linked data. Linked Data is about using the Web to connect related data that wasn’t previously linked, or using the Web to lower the barriers to linking data currently linked using other methods. More specifically, Wikipedia defines Linked Data as “a term used to describe a recommended best practice for exposing, sharing, and connecting pieces of data, information, and knowledge on the Semantic Web using URIs and RDF”.

With the rapid development of linked data, more and more RDF data sets and RDF triples are available. Note that these data sets are highly distributed on the web.

In this thesis, we have studied the problem of estimating the selectivity for SPARQL graph queries on a single RDF dataset and we have achieved a better result. Obviously, the selectivity estimation is still important to query processing on linked data. However, acquiring enough statistics for selectivity estimation on the linked RDF data is much harder than on a single data set. Thus, our method cannot been used directly on the linked RDF data. We have to investigate the new way to solve this problem on the linked data.

Similarly, another interesting aspect is query relaxation on the linked data. In the existing work, RDF queries are relaxed based on the ontology information.
However, on the linked RDF data, there would be no global ontology available on the web. Searching the best suitable ontology to relax the user queries is important to obtain quality answers, which is not well studied at this moment. In addition, the efficiency of relaxation algorithm should also be considered.

In this thesis, we have studied query evaluation on probabilistic RDF data, which assumes that RDF triples are probabilistically independent. However, in some domains there exist RDF triples that are probabilistically correlated. Thus, the future work includes developing advanced probabilistic models such as Bayesian Networks to deal with probabilistic RDF data. Of course, query evaluation on such probabilistic model is much harder and we have to design a efficient query evaluation method.
Bibliography


Query relaxation using malleable schemas. In *SIGMOD Conference*,
Appendix A

Part of Seed Queries

LUBM.

- Qs1: ?y type ‘%Random_type%’. ?y researchInterest ‘%Random_Interest%’.
- Qs2: ?s takesCourse ‘%Random_Course%’. ?s type ‘%Random_type%’.
- Qc1: ?x advisor ?y. ?y type ‘%Random_type%’.

DBLP.

APPENDIX A. PART OF SEED QUERIES

Yago.

- Qs1: \(?s \text{ hasProductionLanguage} \text{'Random\_Language'}\). \(?s \text{ type} \text{'Movie'}\).

- Qs2: \(?s \text{ yagoResource:interestedIn} \text{'Random\_Interest'}\). \(?s \text{ yagoResource:isCitizenOf} \text{'Random\_Country'}\). \(?s \text{ type} \text{'Philosopher'}\).

- Qs3: \(?s \text{ type} \text{'scientist'}\). \(?s \text{ hasFamilyName} \text{o0}\). \(?s \text{ hasGivenName} \text{o1}\). \(?s \text{ yhasWonPrize} \text{'Random\_Prize'}\). \(?s \text{ bornIn} \text{'Random\_City'}\).

- Qc1: \(?x1 \text{ yagoResource:actedIn} \text{x2}\). \(?x2 \text{ hasProductionLanguage} \text{'Random\_Language'}\).

- Qc2: \(?x0 \text{ isMarriedTo} \text{x1}\). \(?x1 \text{ hasWonPrize} \text{x2}\). \(?x2 \text{ hasProductionLanguage} \text{'Random\_Language'}\).

- Qc3: \(?x0 \text{ hasChild} \text{x1}\). \(?x1 \text{ hasPredecessor} \text{x2}\). \(?x2 \text{ hasPredecessor} \text{x3}\). \(?x3 \text{ politicianOf} \text{x4}\). \(?x4 \text{ hasCapital} \text{'Random\_City'}\).

- Qsc2: \(?s \text{ type wordnet:actor\_109765278}\). \(?s \text{ actedIn} \text{m1}\). \(?m1 \text{ type} \text{'movie'}\). \(?m1 \text{ hasProductionLanguage} \text{'English'}\). \(?s \text{ directed} \text{m2}\). \(?m2 \text{ type} \text{'movie'}\). \(?m2 \text{ hasProductionLanguage} \text{'English'}\). \(?s \text{ livesIn} \text{city}\). \(?\text{city locatedIn} \text{state}\). \(?\text{state locatedIn} \text{'New\_York'}\).

- Qsc3: \(?s \text{ actedIn} \text{o1}\). \(?s \text{ hasGivenName} \text{o2}\). \(?s \text{ livesIn} \text{o3}\). \(?o3 \text{ locatedIn} \text{'Random\_Country'}\). \(?s2 \text{ actedIn} \text{o1}\). \(?s2 \text{ hasGivenName} \text{o1}\). \(?s2 \text{ livesIn} \text{n2}\). \(?n2 \text{ locatedIn} \text{'Random\_Country'}\).

- Qsc4: \(?s \text{ type} \text{'actor'}\). \(?s \text{ actedIn} \text{m1}\). \(?m1 \text{ type} \text{'movie'}\). \(?m1 \text{ hasProductionLanguage} \text{'Random\_Language'}\). \(?s \text{ directed} \text{m2}\). \(?m2 \text{ type} \text{'movie'}\). \(?m2 \text{ hasProductionLanguage} \text{'Random\_Language'}\). \(?s \text{ livesIn} \text{city}\). \(?\text{city locatedIn} \text{state}\). \(?\text{state locatedIn} \text{'Random\_State'}\).
Appendix B

Author’s Publications


