University of Alberta

RELATIONAL DATABASES FOR QUERYING NATURAL LANGUAGE TEXT

by

Pirooz Chubak

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2007

Your file Votre référence
ISBN: 978-0-494-33221-4
Our file Notre référence
ISBN: 978-0-494-33221-4

Abstract

With the vast amount of information stored in natural language text, sophisticated query engines are needed to pull data and effectively relate the pieces. While there has been a great deal of activity around semi-structured data and in particular XML, there hasn't been much work on querying natural language text, despite the regularities that exist. This thesis explores a more conservative approach where natural language text is stored in a relational database. We present a framework for querying and integrating natural language text with relational data and investigate different strategies for optimizing queries. Our results show that the size of the plan space depends on the number of query terms and the overlap between query rewritings. One of our results, in particular, show that the complexity of finding an optimal plan in the presence of rewritings is NP-hard. We develop a cost model and the pruning techniques to reduce the size of the search space, and a polynomial-time greedy algorithm that finds a sub-optimal plan over a set of rewritings. Our experimental results indicate great savings in the evaluation costs of the optimized queries and that our greedy algorithm finds either an optimal plan or a plan that is very close to optimal in terms of cost.

Acknowledgements

My Masters program has been an excellent opportunity for me to face challenges, learn how to deal with them and improve my problem solving abilities. However, much of the progress I made was not accomplished without the help of my friends, colleagues and family. I would like to thank the people, whose supports and guidance made it possible for me to successfully complete my program of study and have a wonderful experience as a graduate student.

My deepest thanks to my parents who have always supported me during my life. Without their love and encouragement I could have never been where I am right now. I owe them a large part of all my personal and academic success. Also, lots of thanks to my beloved sister and brother, Golrokh and Puyan, who have always provided me, as their younger brother, with their kindest support and love. Finally, lots of thanks and love to my girlfriend, Sara, who has never stopped dedicating her love to me and patiently helped me through the hardest challenges of my graduate life.

Special thanks to my supervisor, Dr. Davood Rafiei, for his constructive guidance and help. As my supervisor, he has patiently put a lot of time and effort into my research. His nice personality and excellent supervision made my 2 years of Masters much easier and fruitful.

My appreciation and gratitude to my examiners, Dr. Mario Nascimento and Dr. Ali Shiri, for carefully reading my thesis and providing me with lots of invaluable comments. Their nice comments and feedback made a big difference in the quality of this thesis. Moreover, thanks to Dr. Mohammadreza Salavatipour who helped me a lot in improving the theoretical depth of this work.

And finally, My special thanks to my friends, Navid Paydavousi, Raman Yazdani, Ali Hendi and Mohammad Behnam. Without them, graduate life in Edmonton would have been very frustrating. Also, many thanks to the people in the database group. I had a very pleasant experience interacting with database faculty and graduate students. I would like to thank Reza Sherkat, Fan Deng, Vahid Jazayeri, Pouria Pirzadeh, Amit Satsangi, Gabriela Moise, Baljeet Malhotra, Alex Coman and Louisa Antonie as members of database lab who made database lab a nice place to work in. More specifically, I would like to thank Reza Sherkat for his useful hints both on my research and how to carry a successful graduate life.

Table of Contents

List of Tables

List of Figures

Glossary

$a(n_1, n_2)$: Term association score

$c(n)$: Cost of evaluation the query plan rooted at node n

$c_1(n)$: Cost of joining children of node n using index nested loop join

$c_2(n)$: Cost of joining children of node n using the index that retrieves all the matching terms

C_a: Index access cost

C_p: Page size

$c_{rw}(n)$: Cost of reading and writing the result set at node n

$c_r(n)$: Cost of reading the result set at node n

$c_s(n)$: Cost of performing disk storage operations on the result set at node n

C_t: Index entry size

$c_t(n)$: Estimated total cost of evaluating the result set at node n

$c_w(n)$: Cost of writing the result set at node n

$f_j(n_1, n_2)$: Joint frequency of terms at nodes n_1 and n_2

$f(n)$: Frequency of the term at node n

N_a: Average number of terms per rewriting

$R(Q)$: Set of rewritings of query Q

S_n: Selectivity of the term at node n

S_t: Total number of sentences

$sel(Q)$: Selectivity of query Q

$std(Q)$: Standard deviation for the query terms of query Q

$T(Q)$: Set of query terms of query Q

Chapter 1

Introduction

There is a large volume of facts expressed in natural language text. The web in general and electronic versions of library archives, news and encyclopedia in particular are examples of huge natural language text sources used daily. We often want to extract facts from these sources, relate them with facts extracted from other sources, or ask more specific questions. As an example we may be interested in extracting the list of genes mentioned in a collection of biomedical literature and find if a paper has experimental evidence for gene products [7]. Another interesting application could be gathering opinions on a topic or a product from forums and newsgroups, etc.

Past work in this area is usually either specific to a particular domain or task (e.g. named-entity recognition [20]) or assumes a clean and relatively small text collection such as news corpora [65]. There are recent attempts to scale up fact extraction to large collections such as the Web with some limited success (e.g. [51]). However, to the best of our knowledge, there is little work on more general approaches for querying natural language text and evaluation strategies that can scale up to very large text corpora. The problem is challenging because natural language text has little structure (compared to relational standards or even XML). However, natural language text is also richer than pure text in terms of the way facts are organized; there are rules and regularities governing natural language text that can be exploited by a querying engine. As an example, natural language text benefits from a sentence structure which limits the boundary where facts are expressed. Moreover, each sentence has a grammar and the terms and phrases could be tagged by their part of speech. These taggings and the grammar can be exploited by the querying engine to get the desired results more accurately and more efficiently. Finally, term frequencies and the co-occurrence statistics are meaningful. As an example, we expect to have the ratio of sentences which contain a given term over the total number of sentences to be approximately the same over different text collections.

In this thesis, we study the scenario where queries over text are expressed in *Natural Language Text Queries* (NLTQ) and a rewriting engine is used for query expansion [44]. The syntax and the semantics of the queries and the rewritings are discussed in Chapter 3. We further assume that NLTQ queries are integrated into SQL, allowing data from both text and relational sources to be integrated in a query.

1.1 A Motivating Example

Suppose we have partial lists of genes and syndromes, and we want to search for more genes, syndromes and possible relationships that may have been reported in a text collection such as Medline [4]. In a typical setting, the set of known genes and syndromes may be stored in relational tables. Assuming that we are interested in casual relationships between gene defects and syndromes, we may write the query as follows:

```
(SELECT x, y
FROM genes g, "%x gene defects responsible for %y" on Medline
WHERE g.name=x)
UNION
(SELECT x, y
FROM syndromes s, "%x gene defects responsible for %y" on Medline
WHERE s.name=y)
```

The expression **"%x gene defects responsible for %y" on Medline** is expected to return pairs of genes and syndromes that are reported to have the given relationship (see Section 3.1 for the full syntax and explanation). It is feasible that the texts of queries and data may not exactly match. For example, the query does not exactly match **"the X-linked form is a result of mutations in the CD40 ligand gene"**. Therefore, a rewriting engine can expand the query into alternative expressions such as **"%x mutations in %y"** and **"%y is a result of mutations in %x**

gene".

1.2 Objective

This thesis addresses the following question.

"How can we increase the efficiency of querying over natural language text?" There is a large difference between natural language text queries and traditional relational queries expressed in SQL. SQL queries have no ambiguity and the result set returned by the relational engine is always the "correct" answer. On the other hand, queries over natural language text could often be ambiguous and there is no way for the querying engine to identify the correctness of the answer. Therefore, most of the efforts so far are concentrated on how to come up with a list of "good" answers for queries over natural language text. By a "good" answer we don't necessarily mean a "correct" answer. As an example, consider the query **"Albert Einstein was born in %x."** There could be many different answers to the above query, like *'1879'*, *'Ulm'*, *'Germany'* and *'a Jewish family'* which are all correct answers. However, a querying engine can never know which answer the user is looking for. Moreover, there are answers that are not correct but are meaningful like *'United States'* for the above query. Such answers are more likely to be a correct answer, and one challenge is to identify such answers and favor them to less meaningful answers like *'theoretical physicist'* for the above query. Therefore, the result of a natural language text query is a set of matching entries, ranked according to their "goodness".

Most of the efforts on natural language text querying focus on increasing the accuracy of the result set and finding a ranked list of "good" answers. However, to the best of our knowledge, there has not been much work on increasing the efficiency of querying. The problem is important especially when queries are posed over large collections such as the Web. This thesis addresses the issue of efficiency by trying to reduce the cost[1] of a query (which can be expanded using a rewriting engine) while returning the same set of answers.

[1] in terms of the number of I/Os

1.3 Loose vs. Tight Integration

When a query is over multiple different sources, we may have to extract data from each source and join the results. As an example, consider a question like "where was Albert Einstein born?" is given. Since the question is a "where" question, the querying engine can guess that the answer is the name of a location, (e.g. a city or a country). City names and country names can be compiled and stored in relations on a database. Therefore, in order to interrelate the answers extracted from our query on natural language text, we need to integrate our answers with the data stored in relations.

There are generally two different approaches for joining structured data and text. In the first approach, we assume that the text is stored outside DBMS and queried by an external text engine. Therefore, multiple invocations of the external text engine may be needed for data integration. This approach is called a *loose* integration. On the other hand, a *tight* integration is possible when text is directly stored in relations and standard SQL joins are used for integrating data. In both approaches, structured data is stored in relations.

In this thesis, we take a conservative approach and assume that natural language text is stored in a relational database (discussed thoroughly in Chapter 3). This has the benefit that data from both text and relations can be joined in queries, and relational engine functionalities can be exploited for expressing queries and query optimization. Moreover, relational databases are widely used and the data stored in them are highly portable without any need for specialized text engines.

1.4 The Problem Statement and Our Contributions

The problem to be addressed is that given a SQL query with NLTQ expressions, for example in the *from clause*, we want to map the query to an equivalent query or query plan that can be efficiently evaluated by a SQL engine. Given that relational query optimization is a well-studied subject, our focus in this thesis is on mapping and optimizing NLTQ expressions. Our cost models and estimates are based on the same statistics that are typically available to a relational optimizer, hence our methods can potentially be integrated into a relational engine.

Our first contribution is a cost model for estimating the efficiency of a mapping, in terms of the expected number of I/Os, and our experimental results on the accuracy of our estimations. As the second contribution, we develop strategies for pruning query plans that are guaranteed not to be optimal; therefore the size of the search space for an optimal plan is significantly reduced. Our third contribution is on optimizing query rewritings. Given the set of rewritings of a query, some overlap is expected between the terms of the query and its rewritings and also between the terms of the rewritings. In the presence of an overlap, independently optimizing each rewriting is not guaranteed to give the overall best plan. We formalize the search for an optimal plan for a set of rewritings as an optimization problem and derive analytical results on its complexity; our results show that the problem is at least NP-hard (non-deterministic polynomial-time hard). As our last contribution, we relax our optimality criterion and develop an efficient greedy algorithm for finding sub-optimal plans. Our experimental results show that the greedy algorithm finds either an optimal or an almost optimal plan at a much lower cost.

1.5 Structure of the thesis

The rest of this thesis will be organized as follows.

Chapter 2 discusses the background and reviews related work, including query optimization, question answering, information extraction, multi-query optimization and data integration from multiple sources.

A formulation of the problem is given in **Chapter 3** where we formalize the problem as an optimization problem. General concepts related to this thesis, such as natural language text queries (NLTQs) and rewritings are also formally introduced in this chapter.

Chapter 4 (a) introduces a cost model for query plans, in terms of the number of I/Os, and discusses the estimation of frequencies and association scores. (b) provides the formal notations and mathematical definitions for the optimization model we use in this thesis. (c) provides theoretical arguments that help us reduce the size of the search space for optimal plans. Moreover, it shows that the size of the search space for a single query could be significantly reduced and provides an algorithm for finding the optimal query plan with a linear time complexity.

In **Chapter 5**, we study the optimization of queries with multiple rewritings and discuss why the optimization is challenging. We then show that in general, finding an optimal solution is at least NP-Hard. This chapter also introduces a greedy algorithm that finds a sub-optimal solution, with a polynomial time complexity.

Chapter 6 provides experimental evidence in favor of our estimates and algorithms for optimizing single queries and multiple rewritings and also actual execution times on a commercial relational database. The experimental results show significant savings in the costs of optimized plans.

Finally, **Chapter 7** wraps up the thesis with conclusions and future extensions.

Chapter 2

Background and Related Work

The related work can be grouped into 6 categories: (a) query optimization over text, (b) integration of relational data and text, (c) multi-query optimization, (d) full-text support in commercial databases, (e) named entity recognition and question answering, and (f) extracting relations and patterns from text.

2.1 Query Optimization over Text

Increasing the efficiency of the system has been an area of interest in most computer systems and is always in trade-off with effectiveness or accuracy of the system. Querying systems, in particular, benefit from a large body of research on query optimization, indexing and heuristics for finding an approximation of the answer to a given query. Our work in this thesis focuses on increasing the efficiency of queries on natural language text. We will study query optimization over two relevant sources, namely relational data and semi-structured data (XML in particular), as a background for our text query optimization.

2.1.1 Query optimization over relational and semi-structured data

Traditional query optimization over relational data is a well-studied subject. All current commercial relational databases benefit from query optimizers as a major part of their system. Query optimizers always trade between the time required for processing queries and the time required for executing them. However, it turns out that in most cases, the saving in execution time is much greater than the overhead of processing a fixed query plan (otherwise it does not make sense to do an optimization). A few examples of the query optimizers studied in full detail are the query optimizers in System R [58], a leading database engine developed by IBM at the time, and in Ingres [66], the ancestor of Postgres [3], which is an open source relational database engine. Our theoretical arguments on optimizing NLTQs in Section 4.2 agree with the heuristics used in System R for only scanning plans represented in left-deep trees. There are many different types of queries and therefore different methods for query optimization. An example is query optimization over multiple queries which is related to our work in Chapter 5 and will be discussed in Section 2.3. [42] and [40] provide good surveys of query optimization over relational databases.

There's also a large body of work on optimizing queries over semi-structured data and eXtensible Markup Language (XML), e.g. see the work in [49]. A similar approach to ours is taken by Shanmugasundaram et al. [60] on storing XML documents in relational tables and mapping XML queries into SQL expressions over relational tables. It is related to our work in the sense that the data from two sources of different structures are integrated together and techniques for efficient querying and merging of information are studied. However, unlike our approach, the mapping here is directed by Document Type Declaration tags (DTDs) which vary from one document to the next. Moreover, XML documents are more structured and have more well-defined queries. Therefore, it would be easier to plug them into a relational databases framework.

2.1.2 Query optimization over text

There is work on querying and optimizing queries over text but not specifically on natural language text. PAT [57] for example, is a system for searching text with some commercial success [25]; it introduces text regions as first class citizens. Regions are substrings on text with known beginning and end characters. PAT introduces six different operations for text search, including lexical, proximity, and frequency searches. The algebra behind PAT is studied more closely by Consens and Milo where they show the relationship between region algebra and Monadic first order theory of binary trees and the complexity of optimizing queries in the algebra [23]. Unlike PAT, NLTQ is restricted to natural language text and somewhat makes use of the structure of the sentences. There is also work on estimating the selectivity of match conditions over text (also known as text predicates) [18]. This work is also related to ours and may be used for estimating the selectivity of our text queries after being mapped into SQL. The selectivity of terms and queries are used in our system for optimizing natural language text queries (See Section 4.1).

2.1.3 Increasing the efficiency of querying over text

There is also a body of work that addresses the issue of increasing the efficiency of querying over text. However, unlike the work in this thesis, it does not directly focus on text query optimization. An example is the work in [41] which studies the trade-off between crawling (retrieving the documents of interest) and searching (using the indexes on a search engine for filtering the documents which may include our answer) for different applications over text. The Information Retrieval literature also has examples of related work on improving the efficiency of querying. [46] uses algorithms and heuristics for pruning the size of the inverted lists of documents. An inverted list for a term t is a list of documents that t appears in and the position of t in each document. A set of inverted lists for all terms are collected and used in almost all search engines. Since these lists could grow dramatically, as the number of indexed documents increase, their pruning can have a great influence on the execution time of the queries and therefore on the response time of search engine. Finally, in [22] Consens and Milo study the querying of data over files. Their focus is on files with semi-structured content like emails. They show that similar to the querying over databases there is a trade-off between the amount of indexing and the efficiency of the querying over files.

2.2 Integration of relational data and text

For a large number of information extraction and question answering applications, more than one source of information has to be processed. Often these sources have different types of content; e.g. text, images, audio, video or relational data. Querying heterogeneous sources [43, 19] targets to extract pieces of information from different sources and combine the results. Among these systems, there has been more focus on the integration of relational data and text. In particular, integration of structured data and web, as a huge source of text, has been given special attention because of the growing popularity of the web. More on database techniques focusing on web can be found in [31].

Text and relational data have very different properties. The former is classified as unstructured whereas the latter is well-structured [37]. This makes their processing completely different and all the systems that integrate them have to somehow bridge the gap. As discussed earlier there are two common approaches for integrating relational data with text; a "loose" or a "tight" integration. It turns out that both methods have benefits and drawbacks and depending on the application, either approach might be preferable. We will discuss these approaches separately here.

2.2.1 Loose integration

Chaudhuri, Dayal and Yan [17] study several techniques for joining relational data with external text sources. The join methods include semi-join, tuple substitution where every tuple of the relational table is instantiated and probing where text queries that are expected to fail are not sent to the text system. The authors show that the best performing method varies with the selectivity and the fraction of joining tuples. Unlike [17] which treats text system as a black box, our work optimizes text queries based on their declarative expressions. Also since text data is stored in relational tables, a "tight" join is feasible making it more efficient to interrelate data from facts and relations.

The works on combined querying of text documents and relations (such as

WSQ/DSQ[1] [34] and [29]) are also related. WSQ/DSQ uses a module that takes a query as an input, queries a relational engine and a general purpose search engine separately, combines their results and returns the combined result to the user. Unlike our work in this thesis, the result of a text search is usually a set of matching documents.

2.2.2 Tight integration

Grossman et al. use relational databases to implement functionalities of an information retrieval system [35]. The authors support boolean queries and provide keyword and proximity searches and relevance ranking. One interesting contribution is that regardless of the number of terms in the original form of the query, the SQL query on the mapped data has a fixed number of joins. Similar approaches are taken by Discover [39] and SIRE [32]; both support keyword search over relational databases.

More recent work on keyword-based search over text stored in a relational database include the work of Hristidis et al. [38] and DBXplorer [10], where the output is a list of matching rows, ordered according to a relevance ranking function. Unlike our focus in this thesis, these systems are not designed to support tasks like fact extraction and question answering. Earlier work on tight integration of text and relational data can be found in [47] which proposes an indexing for improving keyword searches over relational databases.

2.3 Multi-query optimization

Sometimes optimization must be performed over multiple queries especially when a given query is the union of a set of smaller queries. The example given in Section 1.1 is a simple case where the answer is the union of two SQL queries or predicates. The idea behind optimization of multiple queries is to make use of the common subexpressions in the queries. There are two major steps for any multi-query optimization: (a) finding the common subexpressions and (b) deciding which

[1]stands for Web Supported Queries/Database Supported Queries

set of subexpressions to be materialized. We can list a number of early works in this area [61, 59, 50, 55].

Roy et al. [56] study the optimization of multiple queries more closely. Their method benefits from materializing and re-using common sub-expressions between different queries. They model their optimization problem with a Directed Acyclic Graph (DAG) and introduce heuristics and a greedy algorithm for improving the performance of multi-query evaluation. Our work on optimizing the evaluation of multiple rewritings in Chapter 5 is similar to [56] but has a few important differences. First, given a query (or a rewriting) in our scheme, any subset of the query terms can be used for filtering and for each subset, there is a different set of possible plans. This is unlike the queries in Roy et al. where the query terms or relations are fixed for each query. Second, the search space for the best plan is the union of the plans for all possible term subsets. In the presence of multiple rewritings, the search space is the Cartesian product of the plan sets for different rewritings. We are not sure if a DAG in the style of Roy et al. can be constructed or would be effective for this search space. Third, our greedy algorithm is similar to the one by Roy et al., with a difference that ours enumerates term overlaps whereas theirs iterate over the DAG nodes.

There is also work on pipelining the materialized queries when possible [27]. In this method, materialized subplans are not necessarily written on the disk. The idea is to use the subplans immediately after they are materialized to avoid extra disk reads/writes and directly pipeline the result set of a common subexpression already computed to the other queries or predicates which share it. This will further improve the cost of evaluating multiple queries.

2.4 Full-text support in commercial databases

Many commercial database management systems support facilities for full-text search functionalities integrated into their relational engine. Even open source databases support text searches and specific indexes to improve querying of text stored in relations. Postgresql uses Generalized Inverted Indexes for indexing text [2] and Mysql supports full-text searches and keyword matching [1].

IBM DB2 provides text searches using an extension called Text Extender (TE) [48] and Oracle provides text search support using its InterMedia Text(IMT) extension [8]. DB2 TE provides linguistic functionalities for IR-style keyword matching and text extraction. It uses linguistic indexes for matching all forms of a word, like plural/singular nouns and different verb tenses. The other functionalities include proximity searches, relevance ranking, thesaurus [5] and XML support. Oracle IMT provides a similar set of functionalities. The output of a text search again is a set of matching documents or text fields and it is not easy to tightly integrate and interrelate them with relational data.

Our natural language text specific mappings can be used for question answering on natural language text and relations which are not supported by text extensions available on current commercial DBMSs which treat the text as large strings of characters. The wild card queries that we use in this thesis can be mapped to SQL queries; however, commercial databases have their own query optimizers, which are not designed to specifically optimize natural language text queries. Full-text support integrates fancy indexing and more flexible functions for IR-style search over text. However, to the best of our knowledge, full-text support does not perform query optimization in the extent of that discussed in this thesis over relational tables.

2.5 Named entity recognition and question answering

Related work also includes the literature on named entity recognition where given a fixed set of categories such as person names, locations, percentages and monetary values, the task is to extract and classify the elements in text to one of those categories [20]. This has several benefits. First, as discussed in Chapter 1, it is often required that the answers are bounded in particular classes, such as only location. Therefore, the classified text can result in more accurate results, because we could automatically filter the results that are not in certain classes. Second, the text classification helps other applications on text like pattern matching and relation extraction.

There has been quite a few works on named entity recognition and classification. The work in [26], uses morphological information (such as prefixes and suffixes) of the terms or phrases in a context to classify them. There are other works that usually apply machine learning tools for solving the named entity recognition problem [67, 21, 14]. Since the task is to classify the terms in a text collection into a set of pre-defined categories (or none of the categories), machine learning techniques are very appropriate for this kind of problem.

There is also work on question answering [15, 54, 63] where given a natural language question, the task is to find the most relevant answer from a text collection [65]. The common techniques used in question answering are extracting patterns and relations, using natural language processing, using Wordnet [6] and thesauri for stemming and finding synonyms, antonyms and hypernyms. Our work here is different in that our main focus is on the efficiency of natural language text queries; whereas the question answering task is focused on finding the most relevant answer. Moreover, question answering tracks focus on human readable questions, while our focus is on wild card queries which are more appropriate for a tight integration of text and relational data.

2.6 Extracting relations and patterns from text

As discussed earlier, information extraction from text is important because text is pervasive and many applications depend on text data. Moreover, there is a growing interest in text applications, because text is easy to generate, it is easy to query and it is highly available. As an example, text data does not need any special purpose system to store and manage it, and could be stored and retrieved easily in different file types and shared over any network. The best known of these file types are HTML shared over the world-wide web. However, information extraction from text is challenging because text has little structure and is very noisy. Moreover, learning extraction rules are challenging because the structure behind the text is highly domain dependent. In this section, the literature on automatically extracting relations and patterns will be reviewed. These patterns are used for increasing the recall of the text queries. Our query rewritings are examples of such patterns.

A thorough work on learning rules for information extraction from semi-structured data and text is discussed in WHISK[62]. WHISK uses machine learning for gathering information extraction rules from a set of manually tagged instances. The drawback here is that the set of extracted rules are highly dependent on the quality of the seed instances. Brin [16] develops a system that given a small set of example tuples (e.g. author names and titles), searches the Web (or a text collection) for a larger set of similar tuples. In that work, the collection is first searched for patterns surrounding example tuples; those patterns are later used to find more similar tuples. One interesting property of information extraction over the web is that in most cases a certain fact is expressed in different formats and in different places. This helps the information extraction process to be able to validate the extracted facts, patterns and rules. The drawback is that the web text could contain a lot of noise or false information because there is no validation or supervision process on the authored text.

There is also work on extracting patterns and relations from text. These patterns are used for extracting similar facts and classifying them. The work in [64] finds the entailment relations of the verbs. These entailments are used for paraphrasing by replacing the verbs with their entailment and restructuring the sentence; e.g. by replacing the subject and object of the verb when appropriate. Agichtein thoroughly studies the problem of extracting relations based on a given set of examples and comes up with strategies to address what he calls portability and scalability issues in his Ph.D. dissertation [9]. DIRT [45], a system developed for discovery of inference rules, is also related. The authors claim to have found 182,000 classes of patterns. These classes are used for paraphrasing and generating query rewritings.

There is a large class of work on wrapper generation. Wrappers are programs that make it possible for users to issue queries on the semi-structured data available on the web. In other words, wrappers provide an interface for information extraction and querying of HTML and XML documents. Automatic generation of wrappers has been the area of interest for many web applications. Lixto [13] is a system that provides a graphical user interface for users to issue queries. It uses a supervised algorithm for automatically generating wrappers. Other work on wrapper generation are [11, 36].

A similar approach to our work on fact extraction has been taken in KnowItAll [30] and [52]. However, both of these are applied to web text and the latter only extracts patterns of a given type. KnowItAll is domain independent, but depends on the description of the classes given as input. This description can often be hard to generate. Our work is based on natural language queries of DeWild [44] which have a more confined syntax, allowing us to map text queries to SQL. These wild card queries are almost as easy to write as text queries used by search engines. Moreover, they have a high expressive power and can be easily plugged into a relational database framework.

Chapter 3

Problem Formulation

This chapter presents the syntax and the semantics of the natural language text query (NLTQ) and our mapping of NLTQ expressions to query plans over relations that store natural language text. Finally, we define the problem as an optimization problem and discuss the challenges in Section 3.2.

3.1 Mapping Natural Language Text to Relations

A natural language text query aims to extract certain pieces of data from a collection of natural language text. The syntax and semantics of a NLTQ is defined as follows.

Syntax. A natural language text query (NLTQ) is a sequence of terms, phrases and wild cards. Each NLTQ can be represented with the following grammar.

$$\begin{aligned}
PHRASE &\longrightarrow term \mid PHRASE\ term \\
WILDCARD &\longrightarrow \%variable \mid *PHRASE* \\
NLTQ &\longrightarrow \%variable \\
NLTQ &\longrightarrow PHRASE\ NLTQ \mid NLTQ\ PHRASE \\
NLTQ &\longrightarrow WILDCARD\ NLTQ \mid NLTQ\ WILDCARD
\end{aligned}$$

Semantics. The extractor wild card, denoted by *%variable*, can replace a noun or a noun phrase. The variable in this notation is used for naming and referring to the extracted phrase. The query syntax includes another wild card, denoted by *PHRASE*. This wild card is used for query expansion [53], and indicates that the enclosed phrases can be replaced with similar terms and phrases without much affecting the meaning of the query. Similar terms are usually considered as the terms with roughly the same meaning. A more thorough discussion on the semantics of similar terms and how to obtain them can be found in [44]. The result of a query on a text collection is a table with one column for each extractor and includes all assignments of the variables that give rise to a match. More on querying text using wildcards can be found in [44].

Example 1. The query *%x is the author of %y* extracts pairs of x and y, where x is an author of y. Table 3.1 shows a possible result set for this query.

x	y
J. R. R. Tolkien	The Lord of the Rings
Goerge Orwell	1984
James Joyce	Ulysses

Table 3.1: A sample result set for the query of Example 1

Example 2. Suppose we change the query in Example 1 to *%x is the *author* of %y*. Since 'author' is enclosed in *'s, similar terms to 'author' are considered and the query is re-evaluated. For the given query, similar terms to 'author' can be 'writer', 'co-author', 'editor', etc. The result set of the query will be the union of the results in Table 3.1 and the results of the query with any of the similar terms of 'author'.

Definition 1. *A rewriting for a natural language text query Q is a query Q' such that the two queries have different expressions but the same extractors.*

Query rewriting is intended to increase recall without much affecting the precision. In other words, a query rewriting is a different way of expressing the same question. We expect that the answer sets of a query and its rewritings have a lot of overlap. The reason why rewritings are helpful is that there are many different ways of expressing the same concept in a natural language and we do not know which one is used when writing a query. Query rewritings suggest different patterns that can lead to the same answer and might be used in the text collection instead of the original query. Finding accurate and noiseless query rewritings is a challenge.; However, there are techniques for ranking the result set according to the quality of the rewritings. As an example [44] discusses ranking heuristics like number of ranked pages and mutual information for ranking the result set.

Example 3. The followings are a few of the rewritings for the query given in Example 1.

%x, author of %y
%y is written by %x
%x wrote %y
%x is a novel by %y

We denote the set of terms of a query Q by $T(Q)$ and the set of rewritings of Q by $R(Q)$. For the above example, $T(Q) = \{\text{'}is', \text{'}the', \text{'}author', \text{'}of'\}$. In the presence of rewritings, the result set of a query is defined as the union of the result sets of the query and its rewritings.

In this thesis, we take a relational database approach to query optimization. Input text is parsed and the terms and sentences are extracted and stored in two separate tables. The schema of these two tables are as follows:

- *Terms(term, docid, sid, offset, length, pos). docid* and *sid* are the document ID and sentence ID respectively. *offset* shows the location of the term in the sentence and together with *sid* and *docid* uniquely identify the term. *pos* is the part of speech of the term and is used for filtering the result set of query to a particular part of speech. *length* is the number of characters of the term.

- *Sentences (sentence, sid, docid).* The key for *Sentences* is *docid, sid* and the same set of attributes is a foreign key in *Terms* referring *Sentences.*

There are indexes on *term, docid, sid, offset* and on *sid, docid, term* of *Terms* and on *sid, docid* of *Sentences*, and we assume partial match searches are also supported which is the case in most commercial relational databases. The choice of the schema is largely influenced by the syntax and semantics of our queries. The matching boundary of a NLTQ cannot be larger than a sentence; i.e. NLTQ terms and extractors cannot exceed the limits of a sentence. Hence, the *Sentences* table is sufficient to answer any NLTQ. On the other hand, the smallest unit that can match a wild card is a term. Also, the *Terms* table allows us to do IR-style inverted-list pruning for our queries. Table of n-grams and phrases may also be beneficial for some queries (e.g. [12]); but since they are less general in the sense that they apply to a smaller set of queries, we do not consider them in our schema.

Natural language text may be parsed and the parsed tree of a sentence can give more accurate part-of-speech information about elements and their relationships. As an example, it is desired that the query in Example 1 also matches the text "J.K. Rowling, an English fiction writer, is the author of the Harry Potter fantasy series" and extracts 'J.K. Rowling' and 'the Harry Potter fantasy series' respectively for x and y. Without loss of generality, we assume any parsing of text is done in advance and that any additional information can be stored in relations and may be used by the query evaluation engine. For example, the above sentence may be mapped to two sentences "J.K. Rowling is an English fiction writer" and "J.K. Rowling is the author of the Harry Potter fantasy series" and both sentences may be stored. Note that all necessary parsings and speech taggings are done by dedicated NLP tools before the text is stored in relations. A study of these preprocessings and parsing rules are outside the scope of this work.

Given an NLTQ over text, the query can be mapped into an execution plan over the base tables. In the style of relational query optimizers, a query plan is best described as a tree with base tables at the leaves and the operations either at intermediate nodes or at edges. In our settings, a query plan is a description how to filter the sentences in the text collection using the terms in the query. Therefore, a query plan tree shows which terms and in what order those terms should be used to filter sentences. A left-deep plan for our query in Example 1 is shown in Figure 3.1.(a). Since the *Terms* table is always at the leaves followed by a selection predicate, we can simplify the tree, as shown in Figure 3.1.(b).

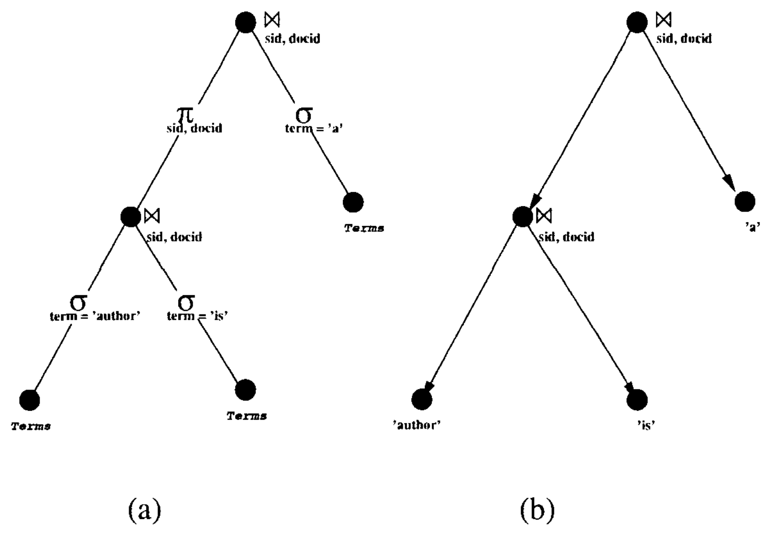

Figure 3.1: A Left-Deep Tree Query Plan for the query of Example 1

3.2 Problem Statement

The space of possible plans for a query Q in general is exponential on $|T(Q)|$ as shown in Section 4.2. For instance in Figure 3.1, the number of terms used for filtering sentences can vary from 0 to $|T(Q)|$ before joining the results with *Sentences*. Filtering sentences based on all query terms is not necessarily the best strategy since each filtering also introduces an overhead. Other plan trees are possible by changing the order of the selections and considering other tree structures such as bushy trees. The space of possible plans is even larger if we consider rewritings and the overlaps between their query plans. For instance, there are terms that appear in multiple rewritings and a query optimizer should be aware of such overlaps in enumerating the plans and cost estimations. We are interested in plans with minimal expected costs. The problem to be addressed is *given a query Q and its rewritings $R(Q)$, find the "best" evaluation plan.* Here the "best" refers to the plan with the least cost, according to our cost model to be discussed next. The right choice of a plan can have a great influence on the cost of query evaluation as shown in some of our experiments.

Chapter 4

Natural Language Text Query Optimization

4.1 Cost Model

Given a query plan, its cost can be estimated in terms of the expected number of I/Os. Each node in the query plan tree can be considered as the root of a subplan tree; the evaluation cost for each node is the sum of the costs of evaluating its children and the cost of joining the results.

4.1.1 Join Cost Estimations

Given nodes n_1 and n_2, the join conditions are $n_1.sid = n_2.sid$ and $n_1.docid = n_2.docid$. Let $l(n)$ and $r(n)$ respectively give the left child and the right child of node n. The following two cost models are used for join.

$$c_1(n) = \begin{cases} C_a + f_{low}\left(\frac{C_t}{C_p} + C_a\right) & if\ \ l(n), r(n)\ \ are\ \ leaves \\ f_{low} \cdot C_a & otherwise \end{cases} \tag{4.1}$$

$$c_2(n) = \begin{cases} 2C_a + \frac{C_t}{C_p}\left(f_{low} + f_{high}\right) & if\ \ l(n), r(n)\ \ are\ \ leaves \\ C_a + \frac{C_t}{C_p} \cdot f_{high} & otherwise \end{cases} \tag{4.2}$$

where

$$f_{low} = \min\left\{f(l(n)), f(r(n))\right\}$$

$$f_{high} = \max\{f(l(n)), f(r(n))\}$$

c_1 models the cost of an index nested loop join, whereas c_2 models the scenario where all the results from both left and right subtrees are retrieved before a join. In our cost models, C_a is the cost of retrieving the first page from an index with matching entries for a given query. In a typical setting, we can assume that C_a is equal to 1.2 I/Os on average [1]. C_t is the size of an index entry in bytes, and C_p is the page size. Therefore, $\frac{C_t}{C_p}$ gives the fraction of a page that is occupied by a single index entry. $f(n)$ is the size of the result set at node n, in terms of the number of distinct sentences that are retrieved. Assuming that the terms occur independently in sentences [2], the frequency of the result set at a node n is given by:

$$f(n) = \begin{cases} S_n & n = leaf \\ \frac{f(l(n)) \times f(r(n))}{S_t} & otherwise \end{cases} \tag{4.3}$$

where S_n is the number of sentences that contain the term at node n and S_t is the total number of sentences. S_n can be evaluated for small or moderate size text collections. For larger collections of text we can either evaluate the values of S_n or estimate them using the value of S_n evaluated for smaller text collections.

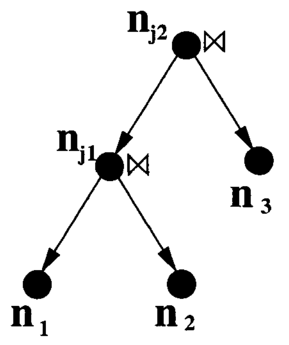

Figure 4.1: A sample query plan tree with two joins

To give a better understanding of the cost models, we show our cost estimations for the query plan tree in Figure 4.1. Consider the cost estimations for n_{j1} and n_{j2}.

[1]This is a conservative estimate for a B+-tree assuming that the first few levels of the index are cached.

[2]It should be noted that in a more realistic setting, terms that appear in a sentence are not independent. For example, the terms of compound words and phrases are more likely to appear together. Section 4.1.3 discusses this issue in more details.

Suppose $f(n_1) \leq f(n_2) \leq f(n_3)$. Since we are using index nested loops join, the cost will be smaller if the outer loop has fewer iterations. Therefore, we always choose the node with lower frequency estimate to form the outer loop of the joins.

- $c_1(n_{j1})$: We use the index on columns <*term, docid, sid, offset*> of the *Terms* table and fetch the matching entries. The cost to fetch the first page is C_a and for the remaining pages is $f(n_1)\frac{C_t}{C_p}$. Finally, for each <*sid,docid*> retrieved in the previous step, we use the index <*sid, docid, term*> of the *Terms* table to select the sentences that also match n_2; the cost here is $f(n_1) \times C_a$.

- $c_1(n_{j2})$: Since the <*sid,docid*> of the sentences that match the subplan rooted at n_{j1} are piped from the lower level, we only need to make $f(n_{j1})$ direct accesses to the index on columns <*sid, docid, term*> of *Terms* to select the sentences that match n_3. The cost is therefore $f(n_{j1}) \times C_a$.

- $c_2(n_{j1})$: As for c_1, we first find the <*sid,docid*> of all sentences that match n_1 using the index on columns <*term, docid, sid, offset*> of the *Terms* table. We also do the same for n_2 and join the results. The total cost is $2C_a + (f(n_1) + f(n_2))\frac{C_t}{C_p}$.

- $c_2(n_{j2})$: Since the matching <*sid,docid*>'s of the n_{j1} are in the memory[3], we only make a direct access using the index on <*term, docid, sid, offset*> of *Terms* to find entries that match n_3 before joining the results. The cost here is $C_a + f(n_3)\frac{C_t}{C_p}$.

4.1.2 Estimating the Cost of a Plan

The cost of evaluating a plan rooted at node n is defined as the sum of the costs of evaluating the left and the right subtrees, the cost of the join and the cost of storing and retrieving any intermediate results (if needed). More formally, the cost can be recursively defined as follows:

$$c(n) = \begin{cases} 0 & n = leaf \\ \min\{c_1(n), c_2(n)\} + c(l(n)) + c(r(n)) + c_s(n) & n \neq leaf \end{cases} \tag{4.4}$$

[3]Assuming that the matching documents fit in the memory

where $c_s(n)$ is the cost of storing and retrieving any intermediate results. When evaluating a non-leaf node, we might need to store the results of left or right subtree on the secondary storage before evaluating their join. The cost of writing the result set of a node n directly depends on the size of the result set and is

$$c_w(n) = \left(\frac{C_t}{C_p}\right) f(n) \tag{4.5}$$

The reading cost can also be defined similarly. Reading from disk is usually a little bit faster than writing. However, since costs of reading and writing are small compared to join costs, we can make the simplifying assumption that $c_r(n) = c_w(n)$. Moreover, reading and writing costs always appear together in the cost formulas we use. Therefore, we define a read-write cost as follows

$$c_{rw}(n) = c_r(n) + c_w(n) \tag{4.6}$$

The intermediate results often are not stored on disk. For instance, for a left-deep tree plan, data from leaves is already on disk and the results of non-leaf nodes can be piped from one operator to next without an actual storage of the results. When the result of left or right subplan must be stored on disk which is the case for bushy trees, for instance, it is desirable to store the one with the smaller result set. This cost here can be estimated as follows:

$$c_s(n) = \begin{cases} 0 & l(n) \vee r(n) = leaf \\ c_{rw}(l(n)) & f(l(n)) < f(r(n)) \\ c_{rw}(r(n)) & f(l(n)) \geq f(r(n)) \end{cases} \tag{4.7}$$

The final result of a plan is a set of matching tuples for the extractor wild cards of the query; therefore, the *sid* and *docid* of the qualifying sentences must be joined with *Sentences*. The cost function should include the cost of this join. Assuming that $r(n)$ is the *Sentences* table, the total cost c_t at the root n can be given as

$$c_t(n) = c(l(n)) + c_1(n). \tag{4.8}$$

It might be desirable to confine the result set of the query to terms with specific part of speech tags. For example we would only like to have nouns in the result set. This could be done by a final join of the result set on the *pos* column of *Terms* table. Since this additional cost does not change our optimization results, we do not include it in our optimization model.

4.1.3 Term Associations

Although assuming independence for terms in a sentence somewhat simplifies the cost estimates, it is not hard to list many cases where this assumption fails.

Definition 2. *Term association score is a number between 0 and 1 which describes the confidence that two terms occur in the same sentence relative to their expected co-occurrence value when the terms are assumed to be independent. For nodes n_1 and n_2, the term association score is defined as:*

$$a(n_1, n_2) = \begin{cases} 0 & f_j(n_1, n_2) < \frac{f(n_1)f(n_2)}{S_t} \\ \frac{f_j(n_1,n_2) - \frac{f(n_1)f(n_2)}{S_t}}{\min(f(n_1),f(n_2))} & f_j(n_1, n_2) \geq \frac{f(n_1)f(n_2)}{S_t} \end{cases} \quad (4.9)$$

where $f_j(n_1, n_2)$ is the joint frequency of two terms and is defined as the number of sentences that contain the terms at nodes n_1 and n_2. Note that n_1 and n_2 must both be leaves, otherwise the term association will be undefined. Since the joint frequency of two nodes is at most $\min(f(n_1), f(n_2))$, the term association score cannot exceed 1. If the expected frequency and real frequencies are equal, association will be zero. On the other hand the higher the association, the more we are underestimating the expected joint frequency.

Storing association scores for all term pairs can be costly. For example, a text collection we have been experimenting with had 64,783 terms and 4.2 billion association pairs. To reduce the size, one heuristic is to filter pairs that have joint frequency zero. This reduced the number of pairs in our collection to 8.3 million pairs. Another heuristic is to remove the pairs whose associations do not have a significant effect on the joint frequency. These would include the entries with an association score less than a threshold. As Figure 4.2 suggests for our dataset, with a threshold of 0.2, we can reduce the number of entries to almost 10% of its previous size, leaving less than 850,000 term pairs in the association table. Finally, to remove the noisy and meaningless terms that have been seen very infrequently (e.g. terms with misspelling, particular names, symbols, etc.), we define a *support* factor for the minimum frequency of terms. This leaves only around 26,000 tuples in the association table when the support factor is 5. With that many pairs, the association table can be cached by the query optimizer for fast look-ups. Note that we only maintain the scores of frequent pairs and as figure 4.2 suggests the number of those pairs drops exponentially as the association score increases. Therefore, we expect an association table to be scalable for large text collections. If the terms at nodes n_1 and n_2 have an entry in the association table, their actual joint frequency is given by

$$f_j(n_1, n_2) = a(n_1, a_2) \min\left(f(n_1), f(n_2)\right) + \frac{f(n_1)f(n_2)}{S_t}. \tag{4.10}$$

Otherwise, the terms can be treated independent. In order to generalize the above formula, we need to check all the term pairs in a query sub-tree for which we are estimating the joint frequency. Suppose we would like to join a subtree T_1 with another subtree T_2 and estimate their joint frequency. Given the estimated frequency for T_1 and T_2, the joint frequency can be estimated as follows:

$$f_j(T_1, T_2) = f(T_1)f(T_2) \prod_{n_1 \in T_1} \prod_{n_2 \in T_2} \frac{f_j(n_1, n_2)}{f(n_1)f(n_2)} \tag{4.11}$$

where n_1 and n_2 represent nodes in subtrees T_1 and T_2 respectively.

4.2 Query Plan Optimization

In this section we discuss the case where there is only one NLTQ to be evaluated. We show that the search space for a single query can be very large. However, most of the query plans are guaranteed not to be optimal and we will provide pruning techniques to reduce the size of the search space while guaranteeing to find the optimal solution.

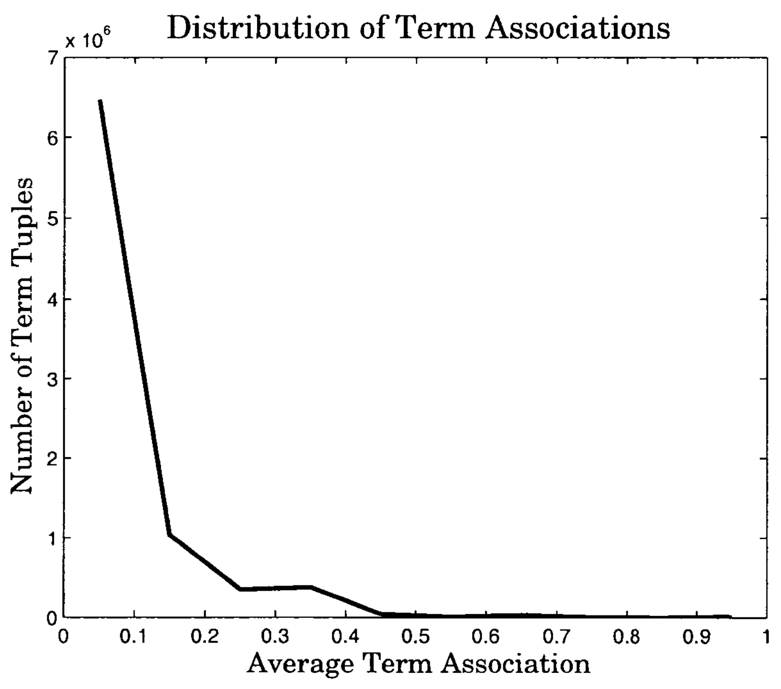

Figure 4.2: Distribution of term associations for 64783 unique terms

4.2.1 Complexity Analysis

Given a query with N terms, a query plan can choose any combination of the terms and place them at the leaves of a plan tree.

Example 4. Suppose the query to be evaluated is *"%x is a car manufacturing company"*. Figure 4.3 shows a few different filtering plans for evaluating the query. As discussed in Section 4.1, all these plans must be joined with the *Sentences* table, in order to find the value of x. In this figure, the plan trees from left to right indicate a left-deep tree with 4 terms, a bushy tree with 5 terms and a right-deep tree with 3 terms respectively. As this figure suggests, a query plan can have any number of terms, can form many different structures, and any combination of the terms could be selected to construct a new plan tree. Hence, the number of total tree plans that could be constructed is large.

Lemma 1. *The number of query plans for a NLTQ with N terms is given by*

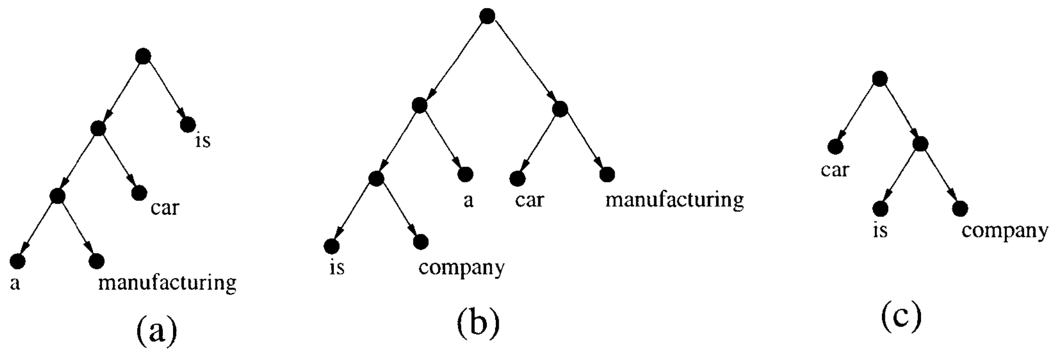

Figure 4.3: A few sample plan trees for the query of Example 4

$QSS(N) = \sum_{n=1}^{N} qss(N,n)$ *where*

$$qss(N,n) = \begin{cases} N & n = 1 \\ \frac{1}{2}\sum_{i=1}^{n-1} qss(N,i)qss(N-i,n-i) & n > 1. \end{cases}$$

Proof. Appears in Appendix A. □

The number of query plans in general can be huge, and searching the plan space for a plan with the least cost can be computationally-intensive. A weak bound for the time complexity of $QSS(N)$ is given by $O(N^N)$. Details can be found in appendix following the proof for Lemma 1.

We want to reduce the size of the search space while still keeping the plan with the least cost in the reduced space. Section 4.2.2 discusses the theorems which show that there exists a linear solution that finds the optimal plan for a single query.

4.2.2 Reducing Search Space Size

The following theorems show that in order to find an optimal solution for a single query, only N query plans need to be evaluated, and the optimal solution is guaranteed to be within the searched space.

Theorem 1. *Assuming query terms are independent, for a query plan with n leaves represented in a Left Deep Tree (LDT), the lowest cost can always be obtained by sorting terms according to their frequencies and placing lower frequency terms on leaves in higher depths.*

Proof. Appears in Appendix A. □

Definition 3. *We say two binary trees t_1 and t_2 are traversal equivalent when there exists a Depth First Search (DFS) traversal that produces the same leaf sequence on both t_1 and t_2. We denote a traversal equivalence by $\cong$.*

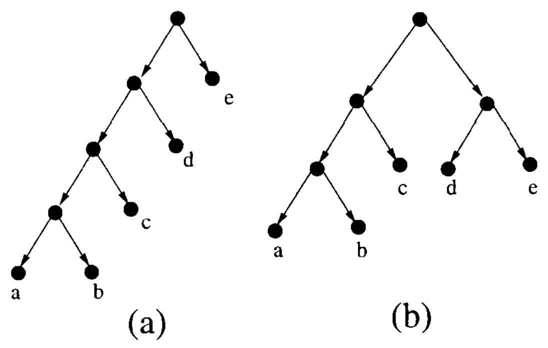

Figure 4.4: An example of Traversal Equivalent trees

As an example, the binary trees shown in Figure 4.4 are traversal equivalent; i.e. (((a b) c) (d e)) $\cong$ ((((a b) c) d) e). Since both trees only have labels at the leaves, all the DFS traversals produce the same leaf sequence.

Theorem 2. *Assuming query terms are independent, any query plan t with n leaves has a traversal equivalent left deep tree that always has a cost less than or equal to the cost of t.*

Proof. Appears in Appendix A. □

Based on Theorems 1 and 2, we know that an optimal plan is always a left-deep tree with terms sorted by their frequencies and placed at the leaves with lower frequency terms in higher depths. The number of those plans cannot exceed N. A linear search over that many plans is guaranteed to find an optimal plan.

Figure 4.5 shows the algorithm to find the local optimal plan. The algorithm takes a query Q as input and returns the best plan and its cost. A list called $terms$ is populated with the elements of $T(Q)$ and then this list is sorted ascending based on the frequency of the terms. As lines 3-5 indicate, the initial best plan includes only the least frequent term. Then in each step of lines 6-11, we consequently add higher frequency terms to our plan. Since we are only interested in left deep trees, in each step we append $terms(i)$ to the right of the current plan P and add one level

```
local_optimal_plan(Q)
1  fill $terms$ list with elements of $T(Q)$
2  Sort elements of $terms$ based on their frequencies
   such that $freq(terms(i)) \leq freq(terms(j))$ iff $i < j$
3  $P \leftarrow terms(1)$
4  $best\_plan \leftarrow P$
5  $min\_cost \leftarrow compute\_plan\_cost(P)$
6  foreach $i = 2, \ldots, |terms|$ do
7     $P \leftarrow$ "$(P\, terms(i))$"
8     $cost \leftarrow compute\_plan\_cost(P)$
9     if $cost < min\_cost$ then
10       $min\_cost \leftarrow cost$
11       $best\_plan \leftarrow P$
12 return $< best\_plan, min\_cost >$
```

Figure 4.5: Algorithm to find local optimal plan

to the height of the plan tree. For each new plan created this way, we compute the total plan cost and find the plan with the minimum cost.

In the next chapter, we will study the problem of optimizing natural language text queries expanded by a rewriting engine.

Chapter 5

Query Optimization Over Multiple Rewritings

In the presence of multiple rewritings, finding an optimal plan for each rewriting is not guaranteed to give an overall plan with the total least cost. In particular, if the plan trees of two or more rewritings have the same subplans, it might be cheaper to evaluate each subplan only once and feed the results to both plans.

5.1 Overlap Handling

The main idea for optimizing a set of rewritings is to take advantage of common terms in their query expressions. If there are two or more terms that are shared among multiple queries, it may be worth to isolate the terms into a subplan and evaluate and store the result for future uses. These subplans can be evaluated once and used many times. We use the term subplan to refer to a part of a plan that may be shared between more than one rewriting. To find a plan with the least total cost over a set of rewritings, we would need to first find a set of subplans that are shared by multiple rewritings and are also worth materializing. Then, for each rewriting, the best local plan has to be re-constructed using the materialized subplans. A set of subplans are worth materializing if the overall cost[1] reduces when subplans are materialized. An optimal solution would be a set of materialized subplans with the minimum overall cost.

[1]sum of the costs of individual rewritings

5.1.1 Handling shared subplans

The following example motivates how reusing materialized subplans can reduce the estimated overall cost.

Example 5. Consider a query that would give a list of athletes who achieved a gold medal in any of the Olympics from a corpus of archived data on sports news. Suppose the query to evaluate is *R0 = "%x, an Olympics gold medalist"* and its set of rewritings is given as follows.

R1 = "%x, the Olympics gold medalist"
R2 = "%x won an Olympics gold medal"
R3 = "%x was a champion in Olympics"
R4 = "%x stood first in Olympics"
R5 = "the gold medal of Olympics was given to %x"

Table 5.1 gives the plan with the minimum cost for *R0* and each of its rewritings when each query is optimized individually; the third column of the table shows the estimated cost of the plan with the least cost in terms of the number of I/Os. We refer to these plans as *local optimal* or *best local* plans.

Rewriting	Best Local Plan	Cost
R0	(Olympics medalist)	29.2
R1	(Olympics medalist)	29.2
R2	(won Olympics)	26.2
R3	(champion Olympics)	20.2
R4	(stood Olympics)	21.4
R5	(Olympics given)	31.8

Table 5.1: The set of rewritings in Example 5, their best local plans and the corresponding estimated costs in terms of the number of I/Os

The set of all the possible subplans for the above query rewritings are given in Table 5.2. The last column in this table shows the amount of saving in terms of the reduction in the number of I/Os if the subplan is materialized. The saving is measured over the sum of the costs of local optimal plans. A negative saving means that materializing the subplan will add additional cost. Notice that this table only provides us with the reduction in cost when each subplan is materialized alone.

However, any combination of the subplans could be selected to be materialized and the saving may or may not be the sum of the savings of each individual subplan.

Subplan	Rewritings	Saving
S1:((Olympics medalist) gold)	R0,R1	26.4
S2:(Olympics medalist)	R0,R1	23.3
S3:((Olympics medal) gold)	R2,R5	10.9
S4:(Olympics medal)	R2,R5	4.9
S5:(Olympics was)	R3,R5	-1.3
S6:(Olympics gold)	R0,R1,R2,R5	-46.9
S7:((Olympics gold) an)	R0,R2	-79.4
S8:((Olympics gold) the)	R1,R5	-98.2
S9:(Olympics in)	R3,R4	-598.7

Table 5.2: The subplans shared by the rewritings of Example 5 and their corresponding estimated cost saving

Later in this chapter we will discuss the approach we take to find a suboptimal solution for this problem. Table 5.3 shows how the solution for this algorithm looks like. Example 5 discusses a very simple problem. More intricate cases will be discussed later.

	overall cost	materialized subplans
best local plans	157.9	
greedy plan	131.5	((Olympics medalist) gold)
optimal plan	120.5	((Olympics medalist) gold) ((Olympics medal) gold)

Table 5.3: The overall cost and materialized subplans for the optimal solution, greedy solution and sum of local optimal plans in Example 5

5.1.2 Handling shared rewritings

It may happen that two or more subplans share a rewriting. In other words, it may happen that a rewriting, share more than one subplan with other rewritings. As an example, as shown in Table 5.2, *R0* shares subplans *S1, S2, S6* and *S7*. How can the optimizer decide if materializing *S1* and *S2* is beneficial? How much saving do we get if we want to materialize both of these subplans? In order to answer these questions, we need to study different scenarios under which a rewriting shares more than one subplan with other rewritings. We can identify the following relationships between subplans.

- **Independent subplans.** When two subplans of a query plan have no common terms or have only one term in common[2], including or excluding a subplan does not affect the cost of the other subplan. Therefore, the subplan selection and optimization algorithm can treat each subplan individually. In Example 5, *S2:(Olympics medalist)* and *S6:(Olympics gold)* are independent subplans for *R0* and *R1*. In the case of independent subplans, the overall saving in cost when materializing both subplans is the sum of cost savings when subplans are materialized individually. (In this case -23.6 I/Os)

- **Intersecting subplans.** If there are two or more common terms between two subplans, including one subplan can reduce the amount of saving for the other subplan. Interdependence between subplans and their costs generally makes it difficult to optimize the queries. One solution is to consider the three cases where one subplan, the other subplan, or both are selected, estimate the cost in each case and select the plan with the least cost. In Example 5, both *S1:((Olympics medalist) gold)* and *S7:((Olympics gold) an)* use the terms 'Olympics' and 'gold'. The set of possible plans for *R0*, in addition to other plans, would include

  (((Olympics medalist) gold) an)
  (((Olympics gold) an) medalist)
  (((Olympics medalist) gold) ((Olympics gold) an))

  However, the number of such plans is exponential on the number of subplans. An approach to reduce the complexity of the solution is either to disallow or to limit the amount of backtracking. For instance, once we choose two subplans to be materialized, we do not check our decision in later steps.

- **Included subplans.** If all the terms of one subplan is included in the other subplan, materializing the plan with the least terms may or may not reduce

[2]One term can always be accessed with an index and there is no need to materialize the results for one term and reusing the results.

the cost of the plan with most terms. In Example 5, *S2* is included in *S1* and the following cases are possible.

- Only materialize *S1*
- Only materialize *S2*
- Materialize *S2* and use the results for evaluating *S1* and materializing it.

In our approach, all three cases can possibly be evaluated and used, depending on the saving of the subplans. This will be discussed in Section 5.3.

5.1.3 Problem statement

Deciding which subplans to materialize, and which rewritings should use the materialized subplans can be tricky. Selecting a subplan to be materialized can influence the cost of other subplans. The cost function for a given set of rewritings is not necessarily linear, if the subplans are not independent; i.e. for two interdependent subplans s_1 and s_2 the total cost when both subplans are materialized is not the sum of the costs when each subplan is materialized.

The problem to be addressed is to find a set of plans, one for each rewriting, and a set of subplans, materialized in advance and reused in plans, such that the total cost of evaluating the plans and materializing the subplans is minimal. The next section analyzes the complexity of problem in its general form, and in Section 5.3 we introduce a heuristic and a greedy algorithm that selects the subplans according to their estimated cost savings.

5.2 Complexity analysis

Given a query q, let $R(q)$ be the collection of its rewritings including the query and $T = \cup_{r_i \in R(q)} T(r_i)$.

Definition 4. *A subplan s is legal over $R(q)$ if there is $r_i \in R(q)$ such that $T(s) \subseteq T(r_i)$.*

Definition 5. *Let M denote the set of all legal subplans of $R(q)$. We call $M' \subseteq M$ an optimal materialization set over $R(q)$ if the sum of the costs of materializing M' and evaluating $R(q)$ with M' materialized is minimal; i.e.*

$$\sum_{m_i \in M'} C(m_i, \emptyset) + \sum_{r_j \in R(q)} C(r_j, M')$$

is minimal where $C(x, Y)$ is the cost of evaluating x given that Y is evaluated in advance and the result is materialized.

Theorem 3. *The problem of finding an optimal materialization set is NP-hard.*

Proof Consider a slightly simpler version of the problem where we want to find if there exists $M' \subseteq M$ such that

$$\sum_{m_i \in M'} C(m_i, \emptyset) + \sum_{r_j \in R(q)} C(r_j, M') \leq k$$

for a fixed k. If we show the NP-hardness for this simplified version, the proof for the more general version follows. We prove this by providing a reduction from the minimum cover problem [33]. Given a collection B of subsets of S, a minimum cover of size k or less for S is $B' \subseteq B$ such that he union of the sets in B' is S and $|B'| \leq k$. Define a cost function C as follows: (1) $C(b_i, \emptyset) = 1$ for every $b_i \in B$, (2) $C(s_j, B') = 0$ if $B' \subseteq B$ and there exists $b_i \in B'$ that covers s_j, and (3) $C(s_j, B') = \infty$ otherwise. B' is a minimum cover of size at most k for S if

$$\sum_{b_i \in B'} C(b_i, \emptyset) + \sum_{s_j \in S} C(s_j, B') \leq k.$$

□

Having a polynomial time algorithm for finding an optimal materialization set implies that we have a polynomial time algorithm for the minimum cover which is unlikely (unless P=NP [33]). A naive algorithm may examine all possible query plans which is expected to be large for large number of terms and rewritings. Next we give a sub-optimal algorithm that runs in polynomial time.

5.3 A Suboptimal Solution

We propose a greedy algorithm called Common Subplan Greedy (CSGreedy) that gives a suboptimal solution with a polynomial time complexity. This algorithm chooses the subplans according to their savings, and the subplan with the highest saving is chosen first. In each step, the algorithm estimates the total cost of rewritings using the subplans chosen so far, and continues until the cost is not decreasing any more. The overall suboptimal plan is the plan with the minimum total cost. Intuitively, subplans which have low costs and are shared among a large number of rewritings, have priority to be selected. This algorithm finds a plan with a suboptimal cost. Experimental results in Chapter 6 show that the cost of plans found using this algorithm for a rewriting set are equal or comparable to that of an optimal plan for the same set.

The algorithm, as presented in Figure 5.1, takes a set R of rewritings and returns a suboptimal solution P and an estimated total cost for P. B is the set of subplans that would need to be materialized; it is initially empty. In steps 2-4, P is initialized to an empty set and is incrementally populated with the tuples $< r, p >$ where $r \in R$ is a rewriting and p is a local optimal plan for r. The search space for finding a local optimal plan for each rewriting has a size linear to the size of the query, as discussed in Section 4.2.

In line 5, we find the total cost of the rewriting set which is the sum of minimum local costs for each plan since B is empty. In lines 6 and 7 we find all subplans that can be built from our set of rewritings and sort the subplans according to their savings over the total cost estimated in Step 5. In lines 8-14, we iterate over the subplans, from the one with the greatest saving to the one with the least and add each subplan to B. Then we recompute the total cost of rewritings given that the set B is materialized; the cost here also includes the cost of materializing B and any additional readings that may be needed. The iteration continues until the point where materializing a subplan does not reduce the cost. The algorithm returns a suboptimal plan and its cost.

In order to compute the complexity of our algorithm, we assume that N_a is the

```
CSGreedy(R)
1  $B \leftarrow \emptyset$
2  $P \leftarrow \emptyset$
3  foreach $r \in R$ do
4     $P \leftarrow P \cup \{< r,\ local\_optimal\_plan(r) >\}$
5  $total\_cost \leftarrow compute\_total\_cost(P, B)$
6  $subplans \leftarrow get\_all\_subplans(R)$
7  Sort the subplans based on their savings over P
   such that $saving(subplans(i)) \geq saving(subplans(j))$ iff $i < j$
8  foreach $i = 1, \ldots, |subplans|$ do
9     $B \leftarrow B \cup subplans(i)$
10    $cost \leftarrow compute\_total\_cost(P, B)$
11    if $cost < total\_cost$ then
12       $total\_cost \leftarrow cost$
13       Update the plans in $P$ assuming that $B$ is materialized
      else
14       return $< P, total\_cost >$
15 return $< P, total\_cost >$
```

Figure 5.1: Common Subplan Greedy Algorithm Pseudo code

average number of terms per rewriting and k is the total number of rewritings. For lines 3 and 4, the algorithm iterates over $\sum_{i=1}^{k} |r_i|$ operations, for which $|r_i|$ is the size of rewriting i in terms of the number of terms. Therefore, the complexity for this section of CSGreedy is approximately $\theta(k \times N_a)$. Since we find and store the costs of local optimal plans in lines 3 and 4, line 5 has a complexity no more than $\theta(k)$, which is negligible. In line 6, CSGreedy computes all subplans that are shared between two or more rewritings. Each subplan must have at least two terms before it is useful. In order to compute the complexity of the algorithm in lines 8-14 we model each term as a random variable T that may happen to be in a rewriting $R_i, i = 1..k$ with a probability of

$$P(X \in R_i) = N_a/N \tag{5.1}$$

where N is the total number of terms. The following Lemma shows that lines 8-14 iterate at most $O(k^2)$ times.

Lemma 2. *Let k be the number of rewritings, assuming that terms are selected using the probability given in Equation 5.1, the expected number of unique overlaps among rewritings is at most $O(k^2)$.*

Proof. Appears in Appendix A. □

The most expensive operation in lines 8-14 is estimating the cost of rewritings with set B materialized; the time complexity of this step is kN_a. Hence, using Lemma 2, we conclude that the complexity of the algorithm is at most $O(N_a k^3)$.

Chapter 6

Experimental Results

In order to evaluate our algorithms we conducted several experiments. The real dataset used for all these experiments was a collection of more than 10,000 NSF proposal abstracts. We processed each document and extracted a collection of around 2.5 million terms and 100,000 sentences. We calculated the frequencies for individual terms and term association scores for pairs of terms and pruned the entries according to our discussion in Section 4.1.

6.1 Cost Model Savings

In this experiment, we do a baseline comparison between a local optimal plan and an 'average' plan, in terms of the difference in estimated costs. For 'average' query plan, we estimate the cost for all query plans and compute the average cost. For our testing, we generated 4 sets of queries, with the number of terms per query fixed in each set, but varied from 2 to 5 between sets. For each set we generated 5000 queries with terms chosen randomly from our term collection. To keep the naturalness of the queries, the selection process used frequencies so that terms with higher frequencies appear more often in our generated queries. The probability of selecting term τ is therefore proportional to the frequency of τ and is given by

$$P(t = \tau) = \frac{f(\tau)}{\sum_{t \in \mathbf{C}} f(t)} \tag{6.1}$$

where **C** is the term collection.

After generating the queries, we calculate the selectivity for each query. The selectivity of a query is the expected number of sentences that contain all the terms of the query, and is given by the product of selectivities of its terms or $sel(Q) = \prod_{t \in N_Q} sel(t)$, where $sel(t)$ is the selectivity of term t, which is the ratio of sentences that contain t and N_Q is the set of query terms. The other parameter we compute is the standard deviation of the selectivities of the terms, denoted by $std(Q)$. For each query we build all possible plans, and estimate the cost for each plan using our cost models.

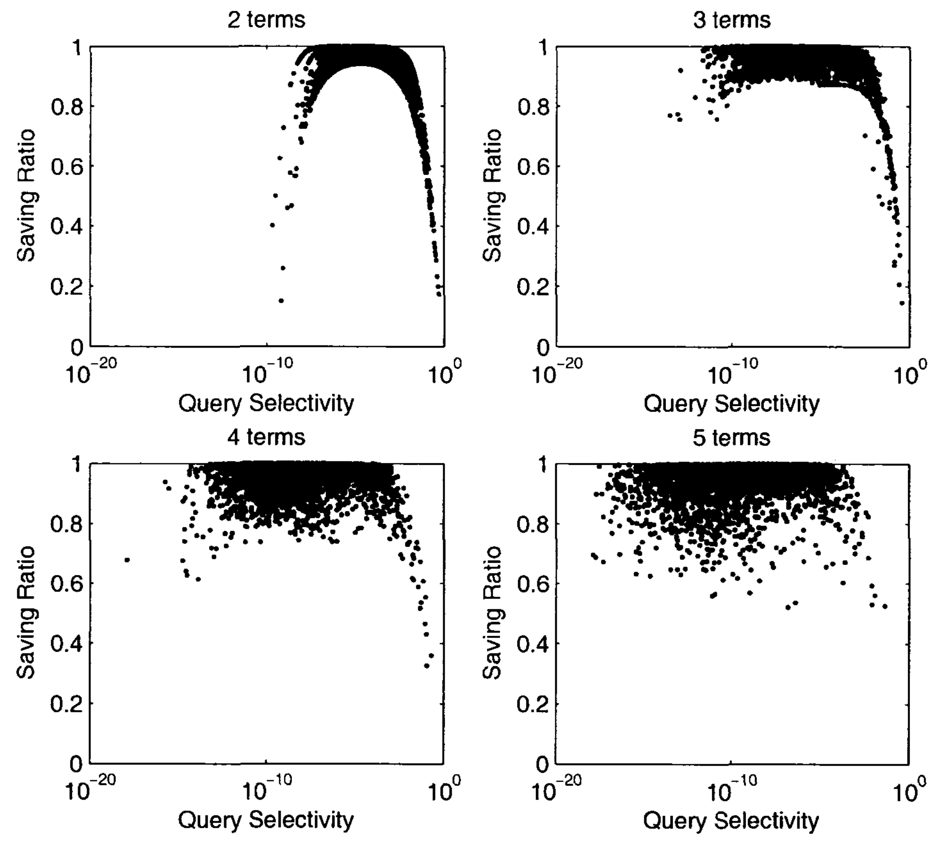

Figure 6.1: Saving Ratios vs. selectivities(log-scale) for queries with different number of terms

The ratio of the saving, defined as (average cost - minimum local cost)/average cost, is shown for each set of our queries in Figure 6.1. As shown, the majority of the savings are close to 1, which indicates that in most cases a best plan has a much lower estimated cost. Also the saving is generally greater when the query selectivity is low, i.e. a small fraction of data is retrieved. This is expected because a high query selectivity is often the result of having only frequent terms in queries,

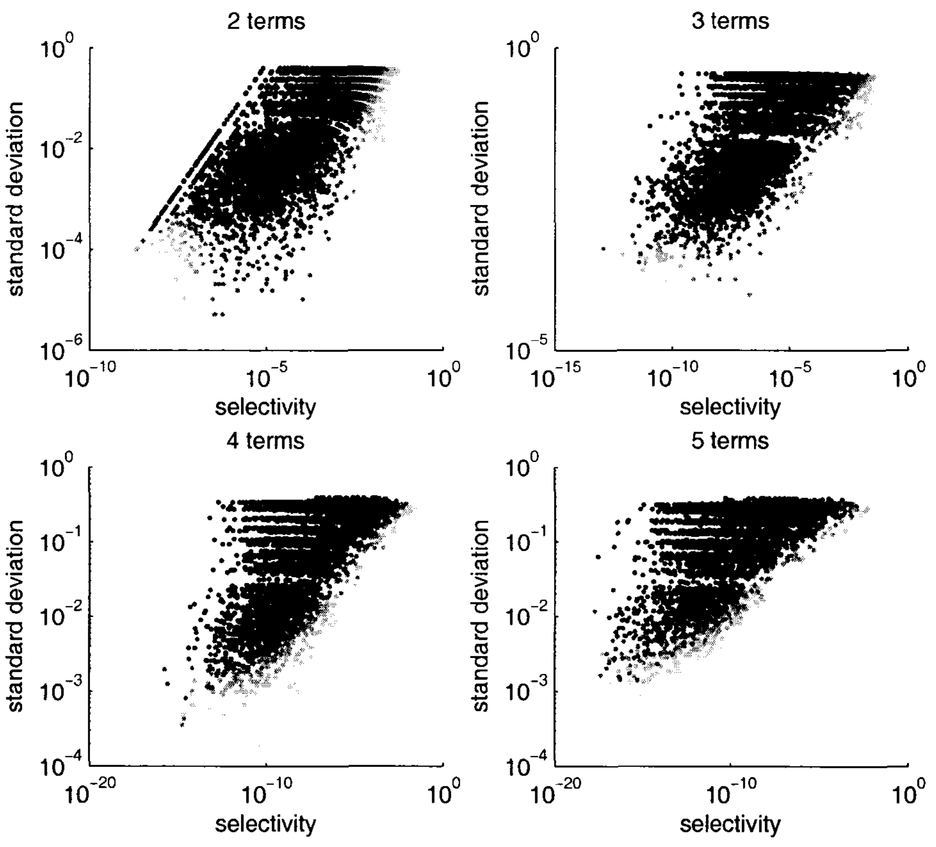

Figure 6.2: Spectrum of Saving Ratio vs. selectivity(log-scale) and standard deviation(log-scale) for queries with different number of terms

and for such queries there is not much difference between the costs of a best plan and an average plan.

However, there are some exceptions; in particular, the plot of 2-term queries shows that not all queries with low selectivities benefit the most. If all query terms have low selectivities, it does not matter much which subset of the terms and in what order they are placed on a plan tree, and as a result there is not much difference between a best plan and an average plan in terms of the estimated number of I/Os. Figure 6.2 shows this intuition for the same set of queries. The intensity of the darkness of each point gives a measure of the saving for a query, with darker points showing higher savings. The saving not only depends on query selectivity but also on the standard deviation of the individual term selectivities. For a fixed selectivity, the saving ratio increases as the standard deviation increases. The reason is that a higher standard deviation results in a greater difference between minimum local cost and the average cost, because best local plans can use low selectivity terms.

6.2 Savings over a Relational Database

To evaluate the accuracy of our cost model in a real setting and its applicability within a relational database framework, we tried to push our query plans to a commercial database engine and compared the costs (in terms of running time) to the costs of the plans fully generated by the relational engine. Most commercial databases that we are aware of impose restrictions that prevent one from passing a query plan. We used IBM DB2 as our relational DBMS, and generated a set of around 5700 random queries having 2 to 5 terms each. For each query we obtained two SQL queries: one query only had the terms of the best local plan and the other had all the terms of the query (referred to as a full filtering plan). Since DB2 does some kind of caching, different invocations of a query can result in different execution times. Therefore, we ran each query three times and only considered the minimum cost (cost of the query with the most caching). To make a fair comparison, we added the CPU overhead of our cost estimation to the execution times of best local plans on DB2. On a modest machine (PIII/933MHz with 2GB RAM), the overhead was on average 0.93ms, 1.8ms, 2.99ms and 4.36 ms for queries with 2, 3, 4 and 5 terms respectively.

Figure 6.3 shows the savings in the execution times of best local plans over the full filtering plans. The majority of the queries have a saving greater than or equal to zero, which means that DB2 has a smaller running time for the best local plan we find over a plan that contains all query terms. Our experiment shows that on average, best local plans run approximately 1.8 times faster on DB2. For queries with more terms, the saving is higher on average. The average savings are 0.09, 0.20, 0.28 and 0.31 for queries with 2, 3, 4 and 5 terms, respectively. The amount of the saving is less than our estimated saving over average plans. There are two reasons for this: first, we could only pass our term selection but not ordering to DB2; second, DB2 was doing its own optimization on query expressions of both

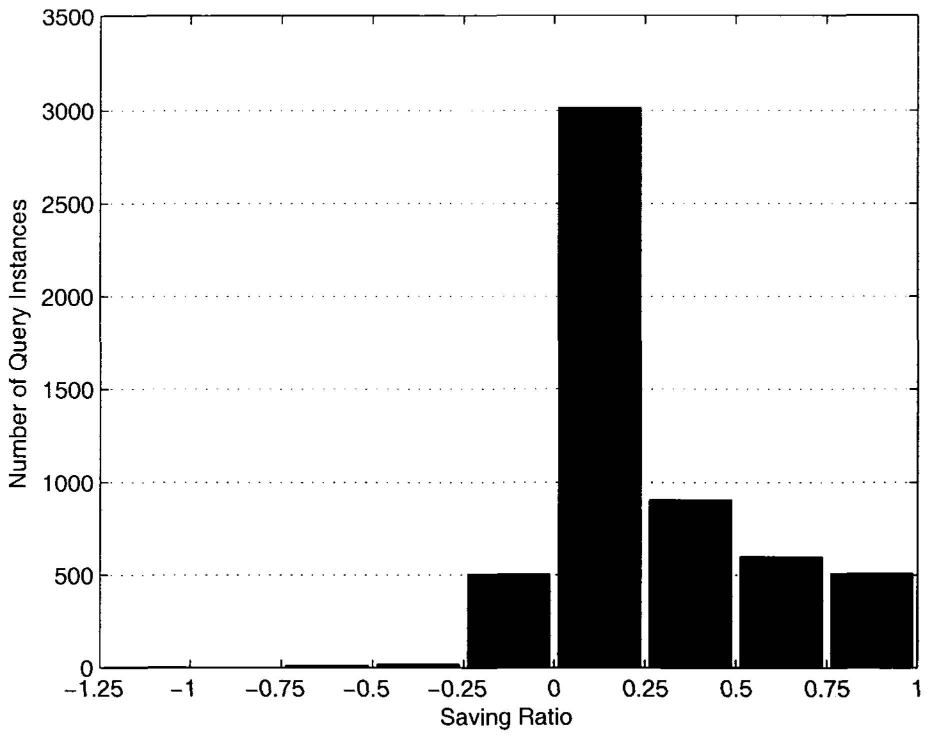

Figure 6.3: The histogram of the savings of db2 running times for 5568 queries

full filtering and best plans. That said, in order to show the statistical significance of the difference, we conducted a student's t-test with $t = 2.892$ and $n > 120$. the probability that our plans executed faster than DB2 plans by chance is less than 0.0025.

6.3 Adding Rewritings

In the presence of multiple rewritings, finding an optimal plan is computationally expensive, as shown analytically; hence we came up with a greedy algorithm which was significantly faster. The objective of our experiment in this section is to evaluate the effectiveness of our greedy algorithm. For our rewriting set, we generated a number of query seeds and used each seed to produce a set of rewritings by replacing, adding or removing terms from the query seed. We created 2 sets of query seeds, each with 100 seeds; query seeds in one set had 4 terms and in the other set had 5 terms. Query seeds were generated by selecting random terms from our collection of terms. For each query seed of size n, the rewritings were generated by randomly replacing, adding or removing r terms, where r varied from 1 to $n - 1$, giving $n - 1$ different sets of rewritings. The probability of replacing terms was 0.5, while probability of adding and removing terms were 0.25. All our term selections adhered to the term selection probability of Equation 6.1. Finally, we ended up with 700 rewriting sets, each having 10 rewritings.

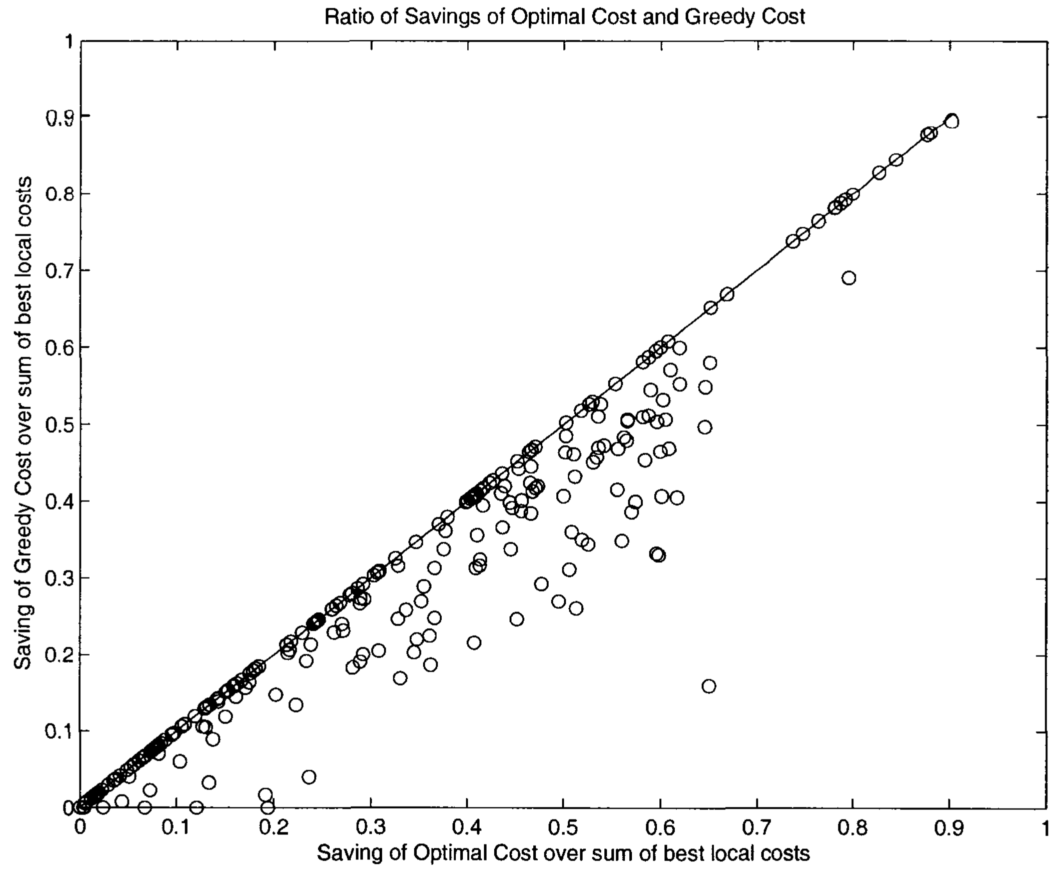

Figure 6.4: The Saving of Optimal Cost vs. the Saving of Greedy Cost

For each rewriting set, we find and estimate the costs of a sub-optimal plan using CSGreedy, an optimal plan using an exhaustive search, and a plan that consists of local optimal plans for each rewriting. We find the saving for both CSGreedy and optimal strategies over the sum of minimum local costs. Figure 6.4 compares the savings of CSGreedy and exhaustive optimal search for 429 rewriting sets that were discussed. Each data point on this figure gives the greedy saving of one set of rewritings. Since the horizontal axis shows the savings of optimal plans, any point on line $y=x$ represents a query for which CSGreedy finds an optimal plan. All the data points must be under the diagonal line of $y=x$ because no rewriting set can have a greedy saving greater than optimal. Also the closer the data point is to the optimal line, the better it estimates the optimal solution. As this figure shows, there are many rewriting sets for which CSGreedy finds an optimal solution. Moreover, there are only a few number of rewriting sets for which CSGreedy cannot find a solution better than the sum of minimum local costs. The amount of saving of CSGreedy is also very much comparable to that of optimal; on average the saving is within 92% of the saving of an optimal plan.

Chapter 7

Conclusions and Future Extensions

To the best of our knowledge, this thesis is the first work that studies querying and query optimization over natural language text within the context of relational databases. Other works either focus on optimizing querying over text or relational data separately, or study a *loose* integration of text and relational data. Moreover, our focus on natural language text (instead of any arbitrary text) provides us with a rich and powerful querying framework, with more room for query optimization.

7.1 Summary

Natural language text is ubiquitous and more information is expressed in natural language text every day. Our focus in this thesis is on querying natural language text and optimizing NLTQ's. We propose relational databases for efficiently querying natural language text and show that they provide a viable option for modeling and optimizing natural language text. Our studied framework offers both the simplicity of text queries and the expressive power of SQL. A rewriting engine is also used for query expansion, which will increase the query recall. Given a set of appropriate query rewritings, We expect that the accuracy of the results are not much affected.

The contributions of this thesis can be summarized as follows:

Mapping. We study the mapping of natural language text and NLTQ into relational tables and SQL queries. We introduce *terms* and *sentences* as the main building blocks of a collection of natural language text and discuss why the choice of schema is a natural selection for our target application.

Cost Model. We formally define a cost model for execution plans of natural language text queries over relational databases. The cost model describes the cost of evaluating a natural language text query plan, given the mapping of text to relations. The statistics used in our cost models are basically the selectivity and co-occurrence association of terms within sentences.

Single Query Optimization. Our study shows that the search space of the plans of a NLTQ can be huge (if done naively), but we derive theoretical results that show the size of the search space can be significantly reduced, while guaranteeing to include an optimal plan in the reduced search space. We also provide the algorithm which finds the optimal plan and has a linear time complexity.

Multiple Query Optimization. We also show that finding the overall optimal plan for a NLTQ and its set of rewritings is at least NP-Hard in general. Therefore, we propose an efficient greedy algorithm with expected $O(N_a k^3)$ time complexity where N_a is the average number of terms per rewriting and k is the number of rewritings. This greedy algorithm sacrifices the optimality condition by reducing the complexity of the algorithm by orders of magnitude.

Theoretical Analysis. We provide theoretical evidence for almost all of the claims about the complexity of the problem and the algorithms used for solving the problem.

Experimental Results. Our experimental results show that the estimated costs of our local optimal plans are usually an order of magnitude less than the costs of average plans, and that query selectivity and standard deviation of term selectivities are two major factors that determine the amount of saving. The actual execution times for our queries mapped into SQL show that our cost model performs well in estimating the cost, and that taking into account the overhead of optimizing queries, it is beneficial to find the optimal plan on a commercial relational database. Our final results on optimizing a set of multiple rewritings demonstrate that our CSGreedy algorithm performs well compared to an optimal plan, with an average saving within 92% of the saving of an optimal plan.

7.2 Future Work

Our work can be extended in a few interesting directions.

- **Indexing.** One direction we are currently pursuing is indexing. The problem would be how to cluster the set of rewritings such that indexing of the information would result in a minimum retrieval cost. This would include traversing the large graph of rewritings and grouping together nodes that are most likely to be queried together. Our initial thoughts suggest that the problem is in general NP-hard.

- **Selectivity Estimation.** Selectivity of terms is the most important statistic used for optimizing text queries. For very large corpora, it is impossible to extract the selectivity of all the terms. Therefore, there are techniques for estimating the selectivities. However, these selectivities can be highly domain dependent. Another direction is doing selectivity estimation that is fine-tuned to natural language text and uses domain information for the estimations. The same domain information can be applied for estimating co-occurrence association.

- **RDB Extension.** One more direction is possibly extending relational databases to better support natural language text. Thus, it might be interesting to plug our optimization strategies into the query optimizers of current commercial relational databases.

Bibliography

[1] MySQL: Full-text search functions. http://dev.mysql.com/doc/refman/5.0/en/fulltext-search.html.

[2] PostgreSQL: Gin indexes. http://www.postgresql.org/docs/8.2/interactive/gin-intro.html.

[3] PostgreSQL: The world's most advanced open source database. http://www.postgresql.org/.

[4] Pubmed. http://www.ncbi.nlm.nih.gov/entrez.

[5] Thesaurus. http://thesaurus.reference.com/.

[6] Wordnet, a lexical database for the english language. http://wordnet.princeton.edu/.

[7] Kdd cup 2002, 2002. http://www.biostat.wisc.edu/~craven/kddcup/.

[8] Oracle text, an oracle technical white paper, 2005. http://www.oracle.com/technology/products/text/pdf/10gR2text_twp_f.pdf.

[9] Eugene Agichtein. *Extracting Relations From Large Text Collections.* PhD thesis, Columbia University, 2005.

[10] Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. Dbxplorer: A system for keyword-based search over relational databases. In *International Conference on Data Engineering (ICDE)*, pages 5–16, 2002.

[11] Naveen Ashish and Craig A. Knoblock. Wrapper generation for semi-structured internet sources. In *Proc. Workshop on Management of Semistructured Data*, Tucson, 1997.

[12] Dirk Bahle, Hugh E. Williams, and Justin Zobel. Efficient phrase querying with an auxiliary index. In *ACM SIG Conference on Information Retrieval (SIGIR)*, pages 215–221, 2002.

[13] Robert Baumgartner, Sergio Flesca, and Georg Gottlob. Visual web information extraction with lixto. In *The VLDB Journal*, pages 119–128, 2001.

[14] Daniel M. Bikel, Richard Schwartz, and Ralph M. Weischedel. An algorithm that learns what's in a name. *Machine Learning*, 34(1-3):211–231, 1999.

[15] Eric Brill, Jimmy J. Lin, Michele Banko, Susan T. Dumais, and Andrew Y. Ng. Data-intensive question answering. In *Text REtrieval Conference (TREC)*, 2001.

[16] Sergey Brin. Extracting patterns and relations from the world wide web. In *WebDB*, pages 172–183, 1998.

[17] Surajit Chaudhuri, Umeshwar Dayal, and Tak W. Yan. Join queries with external text sources: Execution and optimization techniques. In *ACM SIG Conference on management of data (SIGMOD)*, pages 410–422, 1995.

[18] Surajit Chaudhuri, Venkatesh Ganti, and Luis Gravano. Selectivity estimation for string predicates: Overcoming the underestimation problem. In *International Conference on Data Engineering (ICDE)*, pages 227–238, 2004.

[19] Sudarshan S. Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey D. Ullman, and Jennifer Widom. The tsimmis project: Integration of heterogeneous information sources. In *Information Processing Society of Japan (IPSJ)*, pages 7–18, 1994.

[20] Nancy Chinchor. MUC-7 named entity task definition. In *Seventh Message Understanding Conference (MUC-7)*, 1998.

[21] Michael Collins and Yoram Singer. Unsupervised models for named entity classification. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2003.

[22] Mariano P. Consens and Tova Milo. Optimizing queries on files. In *ACM SIG Conference on Management of Data (SIGMOD)*, pages 301–312, 1994.

[23] Mariano P. Consens and Tova Milo. Algebras for querying text regions: Expressive power and optimization. *Journal of Computer and System Sciences (JCSS)*, 57(3):272–288, 1998.

[24] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* The MIT Press and McGraw-Hill Book Company, 2001.

[25] Open Text Corporation. Enterprise content management solutions (ecm). `http://www.opentext.com`.

[26] Silviu Cucerzan and David Yarowsky. Language independent named entity recognition combining morphological and contextual evidence. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 1999.

[27] Nilesh N. Dalvi, Sumit K. Sanghai, Prasan Roy, and S. Sudarshan. Pipelining in multi-query optimization. In *ACM Symposium on Principles of Database Systems (PODS)*, 2001.

[28] Tom Davis. Catalan numbers, 2001. `http://www.geometer.org/mathcircles`.

[29] Stefan Deßloch and Nelson Mendonça Mattos. Integrating sql databases with content-specific search engines. In *International Conference on Very Large Databases (VLDB)*, pages 528–537, 1997.

[30] Oren Etzioni, Michael J. Cafarella, Doug Downey, Stanley Kok, Ana-Maria Popescu, Tal Shaked, Stephen Soderland, Daniel S. Weld, and Alexander Yates. Web-scale information extraction in knowitall: (preliminary results). In *International Conference on World Wide Web (WWW)*, pages 100–110, 2004.

[31] Daniela Florescu, Alon Y. Levy, and Alberto O. Mendelzon. Database techniques for the world-wide web: A survey. *ACM SIGMOD Record*, 27(3):59–74, 1998.

[32] Ophir Frieder. On the integration of structured data and text: A review of the sire architecture (invited talk). In *DELOS Workshop: Information Seeking, Searching and Querying in Digital Libraries*, 2000.

[33] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[34] Roy Goldman and Jennifer Widom. Wsq/dsq: A practical approach for combined querying of databases and the web. In *ACM SIG Conference on Management of Data (SIGMOD)*, pages 285–296, 2000.

[35] David A. Grossman, Ophir Frieder, David O. Holmes, and David C. Roberts. Integrating structured data and text: A relational approach. *Journal of the American Society for Information Science (JASIS)*, 48(2):122–132, 1997.

[36] Jean-Robert Gruser, Louiqa Raschid, M. E. Vidal, and Laura Bright. Wrapper generation for web accessible data sources. In *Conference on Cooperative Information Systems*, pages 14–23, 1998.

[37] Alon Y. Halevy, Oren Etzioni, AnHai Doan, Zachary G. Ives, Jayant Madhavan, Luke McDowell, and Igor Tatarinov. Crossing the structure chasm. In *The Conference on Innovative Data Systems Research (CIDR)*, 2003.

[38] Vagelis Hristidis, Luis Gravano, and Yannis Papakonstantinou. Efficient ir-style keyword search over relational databases. In *International Conference on Very Large Databases (VLDB)*, pages 850–861, 2003.

[39] Vagelis Hristidis and Yannis Papakonstantinou. Discover: Keyword search in relational databases. In *International Conference on Very Large Databases (VLDB)*, pages 670–681, 2002.

[40] Yannis E. Ioannidis. Query optimization. *ACM Computing Surveys*, 28(1):121–123, 1996.

[41] Panagiotis G. Ipeirotis, Eugene Agichtein, Pranay Jain, and Luis Gravano. To search or to crawl?: towards a query optimizer for text-centric tasks. In *ACM SIG Conference on Management of Data (SIGMOD)*, pages 265–276, 2006.

[42] Matthias Jarke and Jürgen Koch. Query optimization in database systems. *ACM Computing Surveys*, 16(2):111–152, 1984.

[43] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *International Conference on Very Large Databases (VLDB)*, pages 251–262, 1996.

[44] Haobin Li and Davood Rafiei. Dewild: a tool for searching the web using wild cards. In *ACM SIG Conference on Information Retrieval (SIGIR)*, page 731, 2006.

[45] Dekang Lin and Patrick Pantel. Dirt - discovery of inference rules from text. In *ACM Conference on Knowledge Discovery in Databases (KDD)*, pages 323–328, 2001.

[46] Xiaohui Long and Torsten Suel. Optimized query execution in large search engines with global page ordering. In *International Conference on Very Large Databases (VLDB)*, pages 129–140, 2003.

[47] Clifford A. Lynch and Michael Stonebraker. Extended user-defined indexing with application to textual databases. In *International Conference on Very Large Databases (VLDB)*, pages 306–317, 1988.

[48] Albert Maier and Hans-Joachim Novak. Db2's full-text search products - white paper, 2006.

[49] Jason McHugh and Jennifer Widom. Query optimization for xml. In *International Conference on Very Large Databases (VLDB)*, pages 315–326, 1999.

[50] Jooseok Park and Arie Segev. Using common subexpressions to optimize multiple queries. In *International Conference on Data Engineering (ICDE)*, pages 311–319, 1988.

[51] Marius Pasca, Dekang Lin, Jeffrey Bigham, Andrei Lifchits, and Alpa Jain. Names and similarities on the web: fact extraction in the fast lane. In *Proceedings of the 44th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 809–816, 2006.

[52] Marius Pasca, Dekang Lin, Jeffrey Bigham, Andrei Lifchits, and Alpa Jain. Organizing and searching the world wide web of facts - step one: The one-million fact extraction challenge. In *Association for the Advancement of Artificial Intelligence Conference*, 2006.

[53] Yonggang Qiu and Hans-Peter Frei. Concept based query expansion. In *ACM SIG Conference on Informatin Retrieval (SIGIR)*, pages 160–169, 1993.

[54] Deepak Ravichandran and Eduard H. Hovy. Learning surface text patterns for a question answering system. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 41–47, 2002.

[55] Arnon Rosenthal and Upen S. Chakravarthy. Anatomy of a modular multiple query optimizer. In *International Conference on Very Large Databases (VLDB)*, pages 230–239, 1988.

[56] Prasan Roy, Srinivasan Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *ACM SIG Conference on Management of Data (SIGMOD)*, pages 249–260. ACM, 2000.

[57] Airi Salminen and Frank William Tompa. Pat expressions : an algebra for text search. In *COMPLEX*, 1992.

[58] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *ACM SIG Conference on Management of Data (SIGMOD)*, pages 23–34, 1979.

[59] Timos K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, 1988.

[60] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *International Conference on Very Large Databases (VLDB)*, pages 302–314, 1999.

[61] Kyuseok Shim, Timos K. Sellis, and Dana S. Nau. Improvements on a heuristic algorithm for multiple-query optimization. *Data Knowledge Engineering*, 12(2):197–222, 1994.

[62] Stephen Soderland. Learning information extraction rules for semi-structured and free text. *Machine Learning*, 34(1-3):233–272, 1999.

[63] Rohini K. Srihari and Wei Li. A question answering system supported by information extraction. In *ACL Conference on Applied Natural Language Processing (ANLP)*, pages 166–172, 2000.

[64] Idan Szpektor, Hristo Tanev, Ido Dagan, and Bonaventura Coppola. Scaling web-based acquisition of entailment relations. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2004.

[65] Ellen M. Voorhees and Dawn M. Tice. Building a question answering test collection. In *ACM SIG Conference on Information Retrieval (SIGIR)*, pages 200–207, 2000.

[66] Karel Youssefi and Eugene Wong. Query processing in a relational database management system. In *International Conference on Very Large Databases (VLDB)*, pages 409–417, 1979.

[67] Guodong Zhou and Jian Su. Named entity recognition using an hmm-based chunk tagger. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics (ACL)*, 1999.

Appendix A

Proof of Theorems and Lemmas

Lemma 1. *The number of query plans for a NLTQ with N terms is given by* $\sum_{n=1}^{N} qss(N,n)$ *where*

$$qss(N,n) = \begin{cases} N & n = 1 \\ \frac{1}{2}\sum_{i=1}^{n-1} qss(N,i)qss(N-i,n-i) & n > 1. \end{cases}$$

Proof. In order to find the search space size for plans with n nodes, selecting their terms from N different terms from the query, we first need to enumerate the different orientations that a plan tree can take with n leaves and then we will find how many different permutations can N different terms can make on such a tree. For example, the number of different orientations that a tree can take with $n = 1$ or $n = 2$ is only one. These are shown in Figure A.1.(a). Therefore, the number of different trees with $n = 1$ is $\binom{N}{1} = N$ when we have N terms to distribute over leaves.

For $n \geq 2$ we can solve this problem recursively and solve for the left and right subtrees. Therefore, we can have k leaves in the left subtree and $n - k$ leaves in the right one. We denote such a tree with $(k)(n-k)$. However, it turns out that $(k)(n-k)$ and $(n-k)(k)$ give the same filtering sequence and have the same plan cost. This is shown for a query plan with $n = 3$ in Figure A.1.(b). It turns out that for each plan there is a symmetric plan that does the same filtering and can be obtained by rotating the tree horizontally. Therefore, we have to halve the total number of query plans possible for a tree with n leaves. This results in the $qss(N,n)$ recursive function. In order to find the overall space size, we just need to sum over trees with different numbers of leaves, from 1 to N, which gives the formula for our lemma.

□

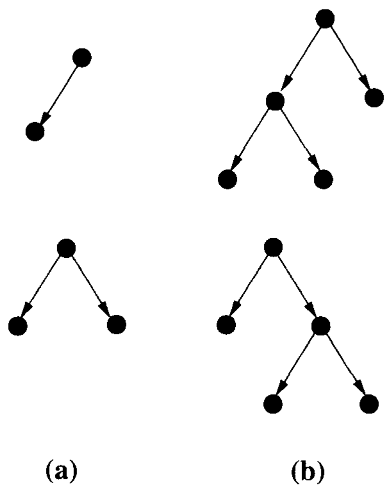
Figure A.1: Examples of binary trees for different number of leaves

Here, we would like to compute an upper bound for the time complexity of $QSS(N)$. We can show that a weak bound is given by $O(N^N)$. The Catalan numbers [28] are given by

$$C_n = \frac{(2n)!}{(n+1)!n!} = \begin{cases} 1 & n = 0 \\ \sum_{i=0}^{n} C_i C_{n-i} & n \geq 0 \end{cases} \tag{A.1}$$

Comparing the recursive definition of Catalan numbers in Equation A.1 with $QSS(N)$, we can show that

$$QSS(N) = \sum_{n=1}^{N} \frac{1}{2^n} \frac{(2n)!}{(n+1)!n!} \frac{N!}{(N-n)!} \tag{A.2}$$

We break Equation A.2 into three different clauses for clarity. The first clause is generated by the $\frac{1}{2}$ coefficient in the formula of $qss(N, n)$ in Lemma 1, the second clause generates the Catalan numbers, and the third clause is generated because the base case of the query space size is N and not 1 as is the case for Catalan numbers.

Solving for Equation A.2, we have

$$
\begin{aligned}
QSS(N) &= N!\sum_{n=1}^{N}\frac{1}{n+1}\binom{2n}{n}\frac{1}{2^n(N-n)!} && \text{(A.3)}\\
&\approx N!\sum_{n=1}^{N}\frac{1}{n+1}\left(\frac{2en}{n}\right)^n\frac{1}{2^n\left(\frac{N-n}{e}\right)^{(N-n)}} && \text{(A.4)}\\
&\approx N!\sum_{n=1}^{N}\frac{e^N}{(N-n)^{(N-n)}(n+1)} && \text{(A.5)}\\
&\approx e^N N!\underbrace{\sum_{n=1}^{N}\frac{1}{n^n(n+1)}}_{O(1)} && \text{(A.6)}\\
&\approx O(N^N) && \text{(A.7)}
\end{aligned}
$$

The approximation used in Line A.4 of the above derivations, is given by the Stirling's approximation [24], which approximates $n!$ for large values of n. Finally, the last approximation in Line A.7 gives a weak upper bound for $e^N N!$.

Theorem 1. *Assuming query terms are independent, For a query plan with n leaves represented in a Left Deep Tree (LDT), the lowest cost can always be obtained by sorting terms according to their frequencies and placing lower frequency terms on leaves in higher depths.*

Proof. Figure A.2.(a) shows our target LDT. terms are sorted according to their frequencies and lower frequency terms are placed at the bottom of the tree. For such a tree, we have

- $\forall\, i, j \le n, i \le j \Leftrightarrow f_i \le f_j$
- $\forall\, i, j \le n, f_i \le f_j \Leftrightarrow H(i) \ge H(j)$

Where $H(k)$ is the height of node (leaf) k. Cost of this ordered LDT is given by

$$
\begin{aligned}
C_1 &= \min\left(C_a + f_1(\frac{C_t}{C_p} + C_a), 2C_a + \frac{C_t}{C_p}(f_1 + f_2)\right)\\
&+ \sum_{k=3}^{n}\min\left(f_{1..k-1}\cdot C_a, C_a + \frac{C_t}{C_p}\cdot f_k\right)
\end{aligned}
$$

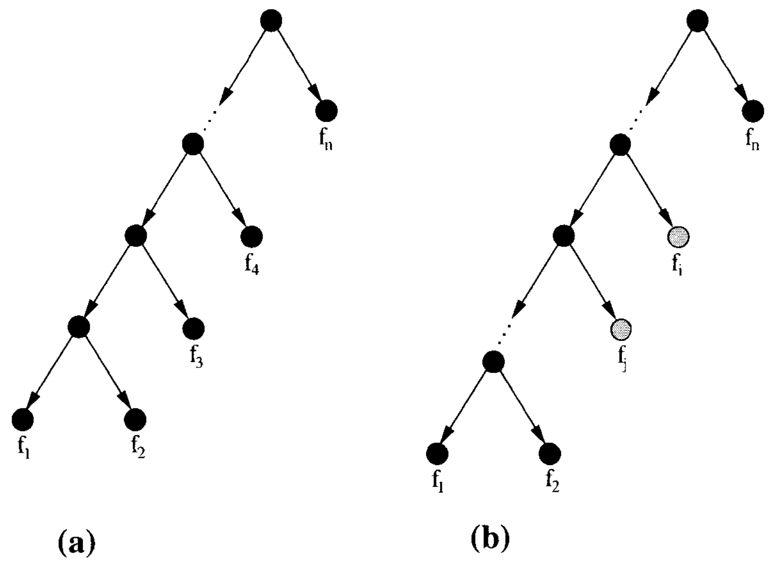

Figure A.2: LDTs used for representing query plans. (a) Best query plan LDT with n leaves (b) terms on i_{th} and j_{th} leaves have been replaced

Where $f_{1..m}$ is the frequency of the resulting subtree and is given by $S_t \times \prod_{i=1}^{m} \frac{f_i}{S_t}$.

Figure A.2.(b) is the same as the LDT to it's left except that terms i and j have been swapped. Therefore, in this LDT we have $i < j$ but $H(j) < H(i)$. The cost of this unordered LDT is given by

$$\begin{aligned}
C_2 &= \min\left(C_a + f_1(\frac{C_t}{C_p} + C_a), 2C_a + \frac{C_t}{C_p}(f_1 + f_2)\right) \\
&+ \sum_{k=3}^{i-1} \min\left(f_{1..k-1} \cdot C_a, C_a + \frac{C_t}{C_p} \cdot f_k\right) \\
&+ \min\left(f_{1..i-1} \cdot C_a, C_a + \frac{C_t}{C_p} \cdot f_j\right) \\
&+ \sum_{k=i+1}^{j-1} \min\left(f_{1..i-1,j,i+1..k-1} \cdot C_a, C_a + \frac{C_t}{C_p} \cdot f_k\right) \\
&+ \min\left(f_{1..i-1,j,i+1..j-1} \cdot C_a, C_a + \frac{C_t}{C_p} \cdot f_i\right) \\
&+ \sum_{k=j+1}^{n} \min\left(f_{1..k-1} \cdot C_a, C_a + \frac{C_t}{C_p} \cdot f_k\right)
\end{aligned}$$

We can easily show that the following two relations hold

- $\forall\, a, b, c \in R,\;\; a \leq b \Leftrightarrow \min(a, c) \leq \min(b, c)$.
- $\forall\; s, s\prime \in P(\{1, \ldots, n\}),\;\; s\prime \subset s \Rightarrow f_{s\prime} \geq f_s$, where $P(s)$ gives the power set of s.

Comparing C_1 and C_2 shows that LDT costs are equal for the $i-1$ lower leaves and $n-j$ upper leaves. These are first, second and last statement of C_2 which are equal to their corresponding costs in C_1. Moreover, using the relations above we can easily show that the fourth statement of C_2 is always less than or equal to it's corresponding cost in C_1, which is

$$\begin{aligned}
&\sum_{k=i+1}^{j-1} \min\left(f_{1..i-1,j,i+1..k-1} \cdot C_a, C_a + \frac{C_t}{C_p} \cdot f_k\right) \\
\leq &\sum_{k=i+1}^{j-1} \min\left(f_{1..k-1} \cdot C_a, C_a + \frac{C_t}{C_p} \cdot f_k\right)
\end{aligned}$$

Therefore, we only need to compare the costs at leaves i and j. To complete the proof, we need to show that the following always holds

$$\begin{aligned}
&\min\left(\underbrace{f_{1..i-1} \cdot C_a}_{A}, \underbrace{C_a + \frac{C_t}{C_p} \cdot f_j}_{B}\right) \\
+ &\min\left(\underbrace{f_{1..i-1,j,i+1..j-1} \cdot C_a}_{D}, \underbrace{C_a + \frac{C_t}{C_p} \cdot f_i}_{C}\right) \\
\leq &\min\left(f_{1..i-1} \cdot C_a, C_a + \frac{C_t}{C_p} \cdot f_i\right) \\
+ &\min\left(\underbrace{f_{1..j-1} \cdot C_a}_{E}, C_a + \frac{C_t}{C_p} \cdot f_j\right)
\end{aligned}$$

We can see that $A \geq D \geq E$ and $B \geq C$. We have

$$\min(A, B) \geq \min(A, C)$$

$$
\begin{aligned}
I.\min(D,C) = D &\Rightarrow E \le D \le C \le B \\
&\Rightarrow \min(E,B) = E \\
&\Rightarrow C_2 \ge C_1 \\
II.\min(D,C) = C &\Rightarrow C \le D \le A \le B \\
\min(D,C) + \min(A,B) &= \begin{cases} C+A & A \le B \\ C+B & B \le A \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
A \le B &\Rightarrow \min(A,C) + \min(E,B) = C + E \\
&, \quad E \le A \Rightarrow C_2 \ge C_1 \\
B \le A &\Rightarrow \min(A,C) + \min(E,B) = C + \min(B,E) \\
&, \quad \min(B,E) \le B \Rightarrow C_2 \ge C_1
\end{aligned}
$$

□

Theorem 2. *Assuming query terms are independent, Any query plan t with n leaves has a traversal equivalent left deep tree that always has a cost less than or equal to the cost of t.*

Proof. An LDT has the property that it only has one leaf pair at the lowest level of the tree and no other leaf has any leaf siblings. Any non-LDT has at least one extra leaf pair. we compare the cost of a leaf pair and cost of two leaves on consequent levels of a tree as will appear in an LDT. Figure A.3.(a) shows an LDT and Figure A.3.(b) shows one of it's traversal equivalent Bushy trees. As Theorem 1 suggests, we assume the frequencies of the LDT are sorted and assume we have the same frequencies for our traversal equivalent bushy tree. Using Figure A.3 and cost formulas, we compute the cost of the LDT and bushy subtrees as follows.

$$
\begin{aligned}
c_L = \min\left(f_r C_a, C_a + \frac{C_t}{C_P} f_3\right) + \\
\min\left(\frac{f_r f_3}{S_t} C_a, C_a + \frac{C_t}{C_p} f_4\right) + c_P
\end{aligned}
\tag{A.8}
$$

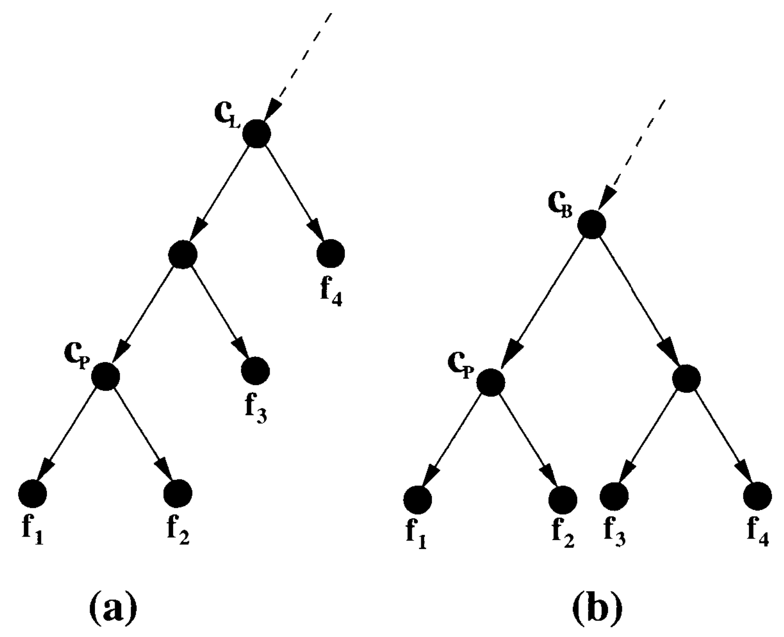

Figure A.3: An LDT versus bushy tree. Bushy trees have at least one leaf pair more than LDTs

$$\begin{aligned} c_B = \min\left(C_a + f_3\left(\frac{C_t}{C_p} + C_a\right), 2C_a + \frac{C_t}{C_P}(f_3 + f_4)\right) \\ +2C_a + \frac{C_t}{C_p}\frac{f_3 f_4}{S_t} + \min\left(f_r C_a, C_a + \frac{C_t}{C_p}\frac{f_1 f_2}{S_t}\right) + c_P \end{aligned} \qquad \text{(A.9)}$$

Where c_L is the cost of the LDT subtree and c_B is the cost of the bushy subtree. Moreover, we can easily show that

$$a + b \le c \Rightarrow \min(a, x) + \min(b, y) \le c \qquad \text{(A.10)}$$

Where x and y could be any numbers. Using Equations(A.8),(A.9) and the above formula, we have

$$\left(C_a + \frac{C_t}{C_P}f_3\right) + \left(C_a + \frac{C_t}{C_p}f_4\right) \le 2C_a + \frac{C_t}{C_P}(f_3 + f_4)$$

Using relation A.10 and adding c_P to both sides of inequality, will result in the following inequality.

$$c_L \le c_P + \underbrace{2C_a + \frac{C_t}{C_P}(f_3 + f_4)}_{m_1} \qquad \text{(A.11)}$$

similarly, we have

$$\left(C_a + \frac{C_t}{C_P} f_3\right) + (cf_r f_3 S_t C_a) \leq C_a + f_3\left(\frac{C_t}{C_p} + C_a\right)$$
$$\Rightarrow c_L \leq c_P + \underbrace{C_a + f_3\left(\frac{C_t}{C_p} + C_a\right)}_{m_2} \qquad \text{(A.12)}$$

Finally, using inequalities (A.11), (A.12) we will have $c_L \leq c_P + \min(m_1, m_2)$ and this proves our theorem or $c_L \leq c_B$

□

Lemma 2. *Let k be the number of rewritings, assuming that terms are selected using the probability given in Equation 5.1, the expected number of unique overlaps among rewritings is at most $O(k^2)$.*

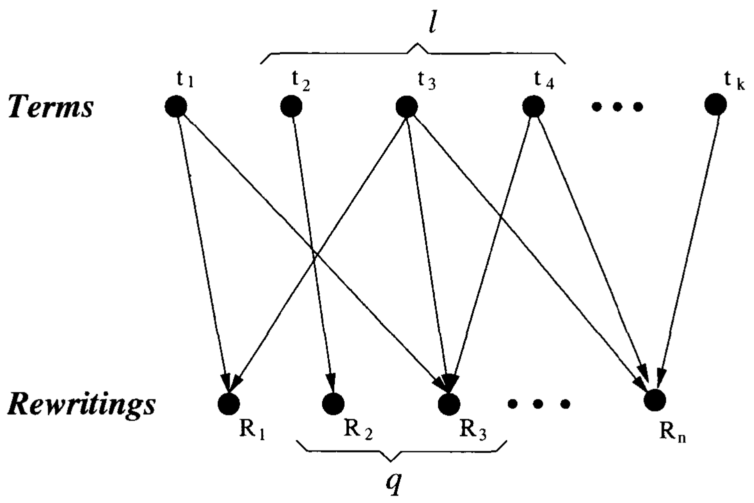

Figure A.4: A graph model for finding the expected number of unique overlaps

Proof. Figure A.4 provides a model for finding the expected number of unique overlaps. As this figure shows, terms and rewritings are denoted by nodes. An directed edge from a term to a rewriting indicates that the term appears in the rewriting. Two rewritings overlap in a term if they both connected to that term. In order to compute the expected number of unique overlaps, we first need to formally define overlaps between rewritings and uniqueness of an overlap.

Definition 6. *A set $\mathcal{T}$ of terms is a unique overlap for a set $\mathcal{R}$ of rewritings if and only if it satisfies the following three conditions.*

1. **Overlap Condition** $\forall \tau \in \mathcal{T} \forall \rho \in \mathcal{R} : \tau \in T(\rho)$

2. **Maximality Condition** $\forall \tau \in \mathcal{T} \nexists \rho \in \mathcal{U}_{\mathcal{R}} - \mathcal{R} : \tau \in T(\rho)$

3. **Uniqueness Condition** $\forall \rho \in \mathcal{R} \nexists \tau \in \mathcal{U}_{\mathcal{T}} - \mathcal{T} : \tau \in T(\rho)$

where $T(\rho)$ is the set of terms of rewriting ρ, $\mathcal{U}_{\mathcal{R}}$ is the set of all rewritings and $\mathcal{U}_{\mathcal{T}}$ is the set of all terms.

According to Definition 6, any set of rewritings overlap in a set of terms if they have the first condition. The second condition guarantees that the set of rewritings is maximal, meaning that no other rewritings shares the same set of terms; Hence, no other rewriting belongs to this set of overlapping rewritings And the last condition guarantees that the set of terms is unique and no other terms are shared by the same set of rewritings. Assuming that terms are uniformly distributed over the rewritings and that the probability that a term occurs in a rewriting is given by Equation 5.1, the probability that a set of size l of terms is a unique overlap of a set of size q of rewritings, is given by

$$(p^{lq}) \left(\sum_{i=0}^{l-1} \binom{l}{i} p^i (1-p)^{(l-i)} \right)^{(n-q)} \left(\sum_{j=0}^{q-1} \binom{q}{j} p^j (1-p)^{(q-j)} \right)^{(k-l)} \quad \text{(A.13)}$$

where p is the probability given in Equation 5.1. The three parts of Equation A.13 enclosed in parentheses satisfy the three conditions of **Overlap**, **Maximality** and **Uniqueness** of Definition 6 respectively.

- In order for all of the q rewritings to overlap in all of the l terms, the terms must all appear in the rewritings. This happens only if there is an edge between all these terms and rewritings. The probability that this happens is p^{lq}.

- In order to satisfy the maximality condition, not all of the l terms must have an edge to any of the rewritings outside the set of q rewritings. In other words, there must be at least one term in the set of l terms that is not connected to a rewriting outside q rewritings. Since the number of such rewritings is $n - q$, maximality is satisfied by

$$\left(\sum_{i=0}^{l-1}\binom{l}{i}p^i(1-p)^{(l-i)}\right)^{(n-q)} = \left(1-p^l\right)^{(n-q)}.$$

The right hand side of above equation better describes the second description of the problem, where as long as there exists a term that is not connected to all the $n - q$ rewritings, maximality holds.

- The uniqueness condition is similar to the maximality. It is given by

$$\left(\sum_{j=0}^{q-1}\binom{q}{j}p^j(1-p)^{(q-j)}\right)^{(k-l)} = (1-p^q)^{(k-l)}$$

Similar to the maximality condition, as long as there is a rewriting that is connected to all $k - l$ remaining terms, the set of l terms is not a unique overlap set.

In order to obtain the expected number of unique overlaps, we need to iterate over any subset of terms and rewritings. Therefore, we have

$$\begin{aligned}\mathcal{E}(p) &= \sum_{l=2}^{k}\sum_{q=2}^{n}\binom{k}{l}\binom{n}{q}p^{lq}\left(1-p^l\right)^{(n-q)}\left(1-p^q\right)^{(k-l)} \quad (A.14)\\ &= \sum_{l=2}^{k}\sum_{q=2}^{n}\frac{\mathcal{B}(l;k,p^q)\times\mathcal{B}(q;n,p^l)}{p^{lq}} \quad (A.15)\end{aligned}$$

where $\mathcal{E}(l,k;p)$ is the expected number of unique overlaps, when the term selection probability is p and $\mathcal{B}$ is the binomial probability distribution function. The reason both l and q are initiated to 2 and greater is that we would like at least two terms be shared with at least two rewritings, otherwise we would not consider them for caching. Finally, we make a simplifying assumption that the number of terms is greater than the number of rewritings, which is the case in most of our rewriting sets.

In order to obtain the time complexity of Equation A.15, we do a numerical analysis of the problem. Figure A.5 shows the expected number of unique overlaps, normalized over the number of terms. As this figure shows, the expected number of unique overlaps increases when the number of terms increase for a fixed number of rewritings. Therefore, the complexity of the expected number of unique overlaps is larger than linear in terms of the number of terms. Figure A.6 gives the expected number of unique overlaps normalized over the number of terms square. As this figure shows, for a fixed number of rewritings, normalized expected number of overlaps decrease as the number of terms increase. Therefore, the complexity of the number of expected overlaps is less than the square of the number of terms. Notice that in both cases the average number of terms per rewriting is set to 5. A numerical analysis of the problem with different number of average terms per rewriting gives similar results. Therefore, the complexity of the expected number of unique overlaps is $O(k^2)$.

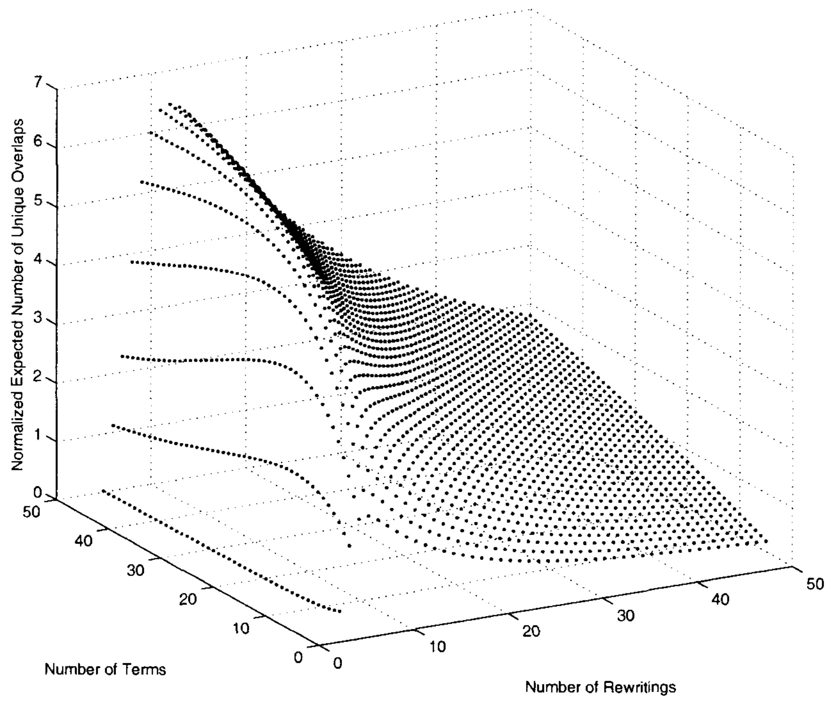

Figure A.5: Expected number of unique overlaps, normalized over the number of terms (Average number of terms per rewriting is 5)

□

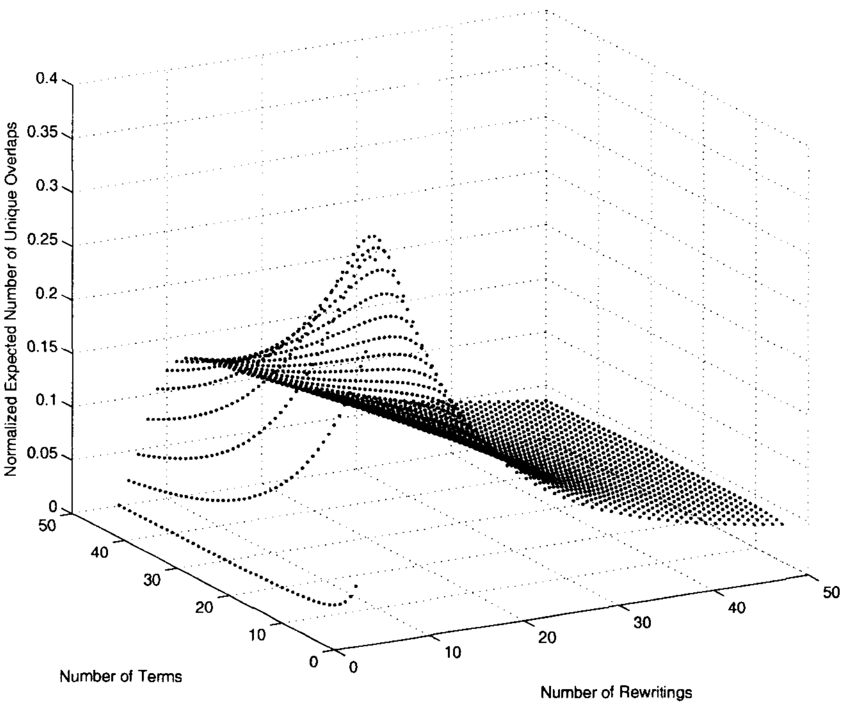

Figure A.6: Expected number of unique overlaps, normalized over the number of terms square (Average number of terms per rewriting is 5)