Selectivity Estimation of Approximate Predicates on Text

by

Hongrae Lee

M.Sc., Seoul National University, 2004

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

The Faculty of Graduate Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA
(Vancouver)
September 2010
© Hongrae Lee 2010
Abstract

This dissertation studies selectivity estimation of approximate predicates on text. Intuitively, we aim to count the number of strings that are similar to a given query string. This type of problem is crucial in handling text in RDBMSs in an error-tolerant way.

A common difficulty in handling textual data is that they may contain typographical errors, or use similar but different textual representations for the same real-world entity. To handle such data in databases, approximate text processing has gained extensive interest and commercial databases have begun to incorporate such functionalities. One of the key components in successful integration of approximate text processing in RDBMSs is the selectivity estimation module, which is central in optimizing queries involving such predicates. However, these developments are relatively new and ad-hoc approaches, e.g., using a constant, have been employed.

This dissertation studies reliable selectivity estimation techniques for approximate predicates on text. Among many possible predicates, we focus on two types of predicates which are fundamental building blocks of SQL queries: selections and joins. We study two different semantics for each type of operator. We propose a set of related summary structures and algorithms to estimate selectivity of selection and join operators with approximate matching. A common challenge is that there can be a huge number of variants to consider. The proposed data structures enable efficient counting by considering a group of similar variants together rather than each and every one separately. A lattice-based framework is proposed to consider overlapping counts among the groups.

We performed extensive evaluation of proposed techniques using real-world and synthetic data sets. Our techniques support popular similarity measures including edit distance, Jaccard similarity and cosine similarity and show how to extend the techniques to other measures. Proposed solutions are compared with state-of-the-arts and baseline methods. Experimental results show that the proposed techniques are able to deliver accurate estimates with small space overhead.
Preface

Most of this dissertation is the result of collaboration with my supervisor Raymond T. Ng at the University of British Columbia, Canada and Kyuseok Shim at Seoul National University, Korea. Published work with them is integrated into this dissertation in a relatively self-contained way. Each publication constitutes a main technical chapter. The basic framework and solutions for string matching semantics in Chapter 3 are described in a VLDB 2007 paper [89]. Techniques for substring matching semantics in Chapter 4 are described in a EDBT 2009 paper [90]. Algorithms for set similarity join in Chapter 5 are described in a VLDB 2009 paper [92]. As the first author of all the papers, I was in charge of all aspects of research including formulating research problems, literature review, developing algorithms, and conducting/analyzing experiments with their guidance.
Table of Contents

Abstract ................................................................. ii
Preface ................................................................. iii
Table of Contents ......................................................... iv
List of Tables ......................................................... viii
List of Figures ......................................................... ix
Acknowledgments ...................................................... xi
Dedication ................................................................. xii

1 Introduction .......................................................... 1
  1.1 Approximate Text Processing ..................................... 1
  1.2 Selectivity Estimation for Query Optimization ..................... 2
  1.3 Other Applications ................................................ 4
     1.3.1 Query Optimization in Text-related Tasks .................... 4
     1.3.2 Query Formulation ............................................ 5
  1.4 Dissertation Goals and Challenges .................................. 6
     1.4.1 String Similarity Models and Measures ....................... 7
     1.4.2 Selection Problems .......................................... 8
     1.4.3 Join Problems ................................................ 10
  1.5 Outline ................................................................ 13

2 Related Work .......................................................... 14
  2.1 Selection Selectivity Estimation .................................... 14
     2.1.1 Exact Selectivity Estimation .................................. 14
     2.1.2 Approximate Selectivity Estimation ......................... 17
     2.1.3 Join Size Estimation .......................................... 17
  2.2 Signatures and Hashing Techniques for Similarity Matching ....... 19
     2.2.1 Min-Hashing .................................................. 19
     2.2.2 Locality Sensitive Hashing .................................... 20
  2.3 Approximate String Matching ........................................ 21
     2.3.1 Web Data Integration and Text Databases .................... 22
     2.3.2 Data Cleaning ................................................ 23
## Table of Contents

- 2.3.3 Other Areas ................................................. 24
- 2.4 Similarity Joins ................................................. 24
  - 2.4.1 Inverted Index-Based Solutions ................................. 25
  - 2.4.2 Signature-Based Solutions .................................... 26
  - 2.4.3 RDBMS-Based Solutions ...................................... 26

### 3 String Selectivity Estimation ........................................... 28
  - 3.1 Introduction ..................................................... 28
  - 3.2 Extended Q-grams ............................................... 29
    - 3.2.1 Extending Q-grams with Wildcards ......................... 29
    - 3.2.2 Replacement Semi-lattice ................................... 30
    - 3.2.3 An Example Replacement Formula .......................... 31
    - 3.2.4 The General Replacement Formula .......................... 32
    - 3.2.5 The General Deletion Formula .............................. 34
    - 3.2.6 The Formula for One Insertion .............................. 35
  - 3.3 Basic Algorithm for Size Estimation ............................ 35
    - 3.3.1 Procedure BasicEQ and Generating Nodes of the String Hierarchy ............................................. 35
    - 3.3.2 Node Partitioning ............................................ 37
    - 3.3.3 An Example and a Formula for Two Insertions .............. 38
    - 3.3.4 Procedure PartitionEstimate ................................. 39
    - 3.3.5 Procedure ComputeCoefficient ............................... 41
    - 3.3.6 Procedure EstimateFreq ..................................... 41
  - 3.4 Optimized Algorithm OptEQ ...................................... 42
    - 3.4.1 Approximating Coefficients with a Replacement Semi-lattice ............................................. 42
    - 3.4.2 Fast Intersection Tests by Grouping ........................ 44
  - 3.5 Empirical Evaluation ............................................ 46
    - 3.5.1 Implementation Highlights .................................. 46
    - 3.5.2 Experimental Setup .......................................... 47
    - 3.5.3 Actresses Last Names ....................................... 48
    - 3.5.4 DBLP Authors ................................................ 49
    - 3.5.5 DBLP Titles .................................................. 50
    - 3.5.6 Space vs Accuracy .......................................... 51
    - 3.5.7 Summary of Experiments .................................... 51
  - 3.6 Conclusions ...................................................... 51

### 4 Substring Selectivity Estimation ...................................... 53
  - 4.1 Introduction ..................................................... 53
  - 4.2 Preliminaries ................................................... 55
    - 4.2.1 Extended Q-grams with Wildcards ............................ 55
    - 4.2.2 Min-Hashing ................................................ 55
    - 4.2.3 Set Hashing ................................................ 57
  - 4.3 Estimation without Signatures .................................. 58
    - 4.3.1 MOF: The MOst Frequent Minimal Base String Method ...... 58
    - 4.3.2 Algorithms not Based on Extended Q-grams ................ 60
Table of Contents

4.4 Estimation with Set Hashing Signatures .............................................. 61
  4.4.1 LBS: Lower Bound Estimation .................................................. 61
  4.4.2 Improving LBS with Extra Minima ............................................ 64
4.5 Extensions to Other Similarity Measures .......................................... 66
  4.5.1 SQL LIKE Clauses .................................................................... 66
  4.5.2 Jaccard Coefficient ................................................................. 67
4.6 Empirical Evaluation ........................................................................ 68
  4.6.1 Experimental Setup .................................................................... 68
  4.6.2 Short Query Set: Estimation Accuracy vs Space Overhead vs Query Time ................................................................. 69
  4.6.3 Query Sets on DBLP Titles .......................................................... 71
  4.6.4 Other Similarity Measures ............................................................ 73
  4.6.5 Impact of Parameters: $k_{min}$, $PT$ .............................................. 73
  4.6.6 Impact of Data Set Size ............................................................... 74
  4.6.7 Recipe: Balancing Space vs. Accuracy ......................................... 74
  4.6.8 Summary of Experiments .............................................................. 74
4.7 Conclusions ......................................................................................... 75

5 Set Similarity Join Size Estimation ......................................................... 76
  5.1 Introduction ....................................................................................... 76
  5.2 Signature Pattern ............................................................................ 78
    5.2.1 Min-Hash Signature ................................................................. 78
    5.2.2 The FSP Problem ...................................................................... 79
  5.3 Lattice Counting ............................................................................... 80
    5.3.1 Computing The Union Cardinality ............................................ 80
    5.3.2 The Union Formula Exploiting Lattice ...................................... 81
    5.3.3 Level Sum Computation ............................................................ 84
  5.4 Power-Law Based Estimation ............................................................. 85
    5.4.1 Level Sum with Power-Law Distribution .................................. 85
    5.4.2 Approximate Lattice Counting ................................................ 87
    5.4.3 Estimation With Limited Pattern Distribution ........................... 88
  5.5 Correction of The Estimation ............................................................. 89
    5.5.1 Systematic Overestimation By Min-Hash .................................... 89
    5.5.2 Error Correction By State Transition Model ............................. 89
  5.6 Experimental Evaluation .................................................................. 92
    5.6.1 Experimental Setup ................................................................. 92
    5.6.2 DBLP Data Set: Accuracy and Efficiency .................................. 94
    5.6.3 Effectiveness of Lattice Counting and Error Correction .......... 94
    5.6.4 Scalability .................................................................................. 95
    5.6.5 Synthetic Data Set: Accuracy, Efficiency and Scalability ........ 96
    5.6.6 On Power-law Hypotheses ....................................................... 97
    5.6.7 Summary of Experiments .......................................................... 97
  5.7 Conclusion ......................................................................................... 97
# Table of Contents

6 Vector Similarity Join Size Estimation ............................................ 99
   6.1 Introduction ................................................................................. 99
   6.2 Baseline Methods ..................................................................... 100
      6.2.1 Random Sampling .......................................................... 100
      6.2.2 Adaptation of Lattice Counting ...................................... 101
   6.3 LSH Index for the VSJ Problem .................................................. 101
      6.3.1 Preliminary: LSH Indexing .............................................. 101
      6.3.2 Estimation with Uniformity Assumption ......................... 102
      6.3.3 LSH-S: Removing Uniformity Assumption ..................... 104
   6.4 Stratified Sampling using LSH .................................................. 106
      6.4.1 LSH-SS: Stratified Sampling ........................................... 106
      6.4.2 Analysis .......................................................................... 109
   6.5 Additional Discussions ............................................................ 112
      6.5.1 The Optimal-k for The VSJ Problem ................................... 112
      6.5.2 Extensions ...................................................................... 113
   6.6 Experimental Evaluation .......................................................... 115
      6.6.1 Set Up ............................................................................ 115
      6.6.2 DBLP: Accuracy, Variance and Efficiency ....................... 116
      6.6.3 NYT: Accuracy, Variance and Efficiency ......................... 118
      6.6.4 PUBMED: Accuracy, Variance, Efficiency ..................... 118
      6.6.5 Impact of Parameters ...................................................... 119
      6.6.6 Summary of Experiments ................................................. 121
   6.7 Conclusion .............................................................................. 122

7 Conclusion .................................................................................... 123
   7.1 Key Contributions ..................................................................... 123
   7.2 Future Work ............................................................................ 124
      7.2.1 Supporting Diverse Similarity Measures ......................... 124
      7.2.2 Adaptation for Query Processing ...................................... 125
      7.2.3 Integration with Full-Fledged RDBMSs ............................ 125
      7.2.4 Incremental Maintenance of Summary Structures ............ 126

Bibliography .................................................................................... 127
List of Tables

2.1 Selectivity Estimation Techniques on Text .............................................. 15

6.1 An Example Probabilities in DBLP .......................................................... 106
6.2 Summary of Notations .............................................................................. 115
6.3 Relative Error on DBLP ................................................................. 117
List of Figures

1.1 The optimizer generated plan (a) for the example query runs 3 times slower than an alternative plan (b) on the DBLP database. The estimated selectivity of the approximate text predicate (output of node 1 in plan (a)) was 1,722 whereas the true selectivity was 57,602. This poor selectivity estimation caused the selection of a sub-optimal plan such as plan (a) rather than plan (b). 3
1.2 An Example of Using Answer Cardinality to Help Formulating Queries 6
1.3 Overview of Contributions 8

3.1 String Semi-lattice for \textit{Ans}(“abcd”, 2R) 30
3.2 Number of Resulting Intersections for \textit{Ans}(“abcd”, 2R) 32
3.3 Various Representative String Hierarchies 34
3.4 A Skeleton for Estimating Frequency 36
3.5 A Skeleton for Procedure BasicEQ 37
3.6 Examples of Local Semi-lattice 38
3.7 A Skeleton of Procedure PartitionEstimate 39
3.8 A Skeleton of Procedure ComputeCoefficient 40
3.9 String Hierarchy of \textit{Ans}(“abc”, 1I1R) 42
3.10 Completion of a Replacement Semi-lattice of \textit{Ans}(“aabc”, 2R) 42
3.11 Approximating Coefficients for “aabc” 44
3.12 A Skeleton of Procedure OptEQ 45
3.13 Actress Last Name, \(\tau = [1, 3]\) 47
3.14 DBLP Authors, \(\tau = 1, 2, 3\) 48
3.15 Error Distribution: \textit{OptEQ}(MM, 9, 5, 2) vs \textit{SEPIA} 49
3.16 DBLP Titles, \(\tau \in [1, 3]\) 50

4.1 Set Resemblance Example 57
4.2 An Illustration of LBS 62
4.3 LBS Using the First and Second Minima 65
4.4 Estimation of Signatures 65
4.5 Short Query Set on DBLP Authors 68
4.6 Short Query Set on DBLP Titles 70
4.7 Error Distributions on DBLP Authors 70
4.8 Long Query Sets on DBLP Titles 72
4.9 Other Similarity Measures 72
4.10 Impact of \(k_{\text{min}}\) and \(PT\) on IMDB Keywords 72
4.11 Impact of Data Set Size on Error 74
## List of Figures

5.1 An Example DB .................................................... 78
5.2 An Example of Signature Patterns (freq ≥ 2) ............................. 79
5.3 Overlapping Relationship ............................................. 82
5.4 Example Pattern Lattice Structures .................................... 82
5.5 Signature Pattern Distribution .......................................... 86
5.6 Relaxed Lattice ....................................................... 88
5.7 # pair-similarity Plot of 2 Subsets of the DBLP Data Set ............... 89
5.8 SSJoin Size Distribution in the DBLP data ................................ 90
5.9 SSJoin size: True vs. MinHash ......................................... 90
5.10 An Example of Imbalance in Transitions .................................. 90
5.11 Accuracy on the DBLP Data Set ......................................... 94
5.12 Performance on the DBLP Data Set ....................................... 95
5.13 Effectiveness of Lattice Counting and the Error Correction ............. 95
5.14 Scalability Using the DBLP Data Set ..................................... 96
5.15 Accuracy on the Synthetic Data Set ....................................... 96

6.1 Probability Density Functions of (non-) Collision in the LSH Scheme ...... 103
6.2 Accuracy/Variance on DBLP ............................................... 117
6.3 Accuracy/Variance on NYT .................................................. 118
6.4 Impact of k on DBLP ....................................................... 119
6.5 Accuracy/Variance on PUBMED ........................................... 119
6.6 Relative Error Varying δ (the Answer Size Threshold) in SampleL ........ 121
6.7 The Number of τ with \(\hat{J}/J \geq 10\) (overest.) or \(J/\hat{J} \geq 10\) (underest.) Varying δ. The total number of τ is 10: \{0.1,0.2,...,1.0\} ...................... 121
6.8 Relative Error Varying the Sample Size \(m\) ................................. 122
6.9 The Number of τ with \(\hat{J}/J \geq 10\) (overestimation) or \(J/\hat{J} \geq 10\) (underestimation) Varying the sample size \(m\). The total number of τ is 10: \{0.1,0.2,...,1.0\} 122
Acknowledgments

Looking back, I realize my life and academic pursuit has been blessed with many people. I would like to express my sincere thanks here.

My supervisor Raymond Ng has enormously influenced me in every aspect throughout my years at UBC. I simply cannot think of a better adviser than him. He showed deep insights whenever I was at a loss and encouraged me whenever I felt down. I am also shaped by his confidence, determination, his managerial and interpersonal skills. He has been my role model and I hope to continue his examples. I am also greatly indebted to Kyuseok Shim at Seoul National University. I was lucky to meet him in a class during my Masters and it was then when I started developing my interests in research. Let alone lessons I learned while working with him in many years, it was he that introduced Raymond and Surajit. Most of my research has been done with Raymond and Professor Shim and they always have impressed me with their insights, technical depth and knowledge. I would like to express my gratitude to them. My appreciation also goes to Professor Hyoung-Joo Kim at Seoul National University, who was my Master’s supervisor, for his support and advises.

I would like to thank faculty members at the DMM lab at UBC: Laks V.S. Lakshmanan and Rachel Pottinger. My research skills have been greatly influenced through close interaction with them by personal meetings, classes, and group seminars. They have been there for me whenever I was seeking advise and never turned down my unsolicited visits and requests.

Another two people who have had deep influence on my research style, passion, skills and career are Surajit Chaudhuri and Vivek Narasayya at Microsoft Research. During the two summers in the DMX group at Microsoft Research, I was greatly inspired by their knowledge, sharpness, communication skills and personalities. I truly enjoyed my life at Microsoft Research and interactions with the DMX group members. I thank them for all the opportunities they offered.

I thank other graduate students in the DMM lab at UBC for their friendship, feedback on my work and advises on various matters: Byung-Won On, Xiadong Zhou, Michael Lawrence, April Webster, Andrew Carbonetto, Shaofeng Bu, Amit Goyal, Mohammad Tajer, Min Xi, Xu Jian, and Jie Zhao. My graduate life also has been enriched by friendship of other fellow graduate students at UBC or SNU: Ewout van den Berg, Sang Hoon Yeo, Argun Arpaslan, Steve Chang, Billy, Jaehyok Chong, and Jongik Kim.

Last, but most importantly, I am greatly indebted to my family. My PhD has been a rather long journey since I decided to pursue an academic career, but my wife, Jungmi, has always been there for me. My parents and brothers have been very understanding and supportive all the years. I also would like to thank my mother in law for her belief and supports as well. I was able to follow my passion since I knew I have home and loving family. Nothing would have been possible without their love, support and sacrifice. They are behind all I do.
To my loving wife Jungmi, brothers, and parents
Chapter 1

Introduction

This dissertation studies the problem of selectivity estimation of approximate predicates on text. Intuitively, approximate text query processing is to find records that are similar to the query string. An example query is “find papers written by ‘Przymusinski’ in a citation database allowing errors” Such similarity queries are of practical importance since textual data may not be clean; they often have errors such as typographical errors. Moreover, users may experience difficulties in correctly spelling the query.

In this dissertation, we focus on selectivity estimation of similarity queries. Selectivity estimation is different from query processing in that it focuses on the question of how many: query processing retrieves actual records satisfying given predicates and selectivity estimation estimates the number of records satisfying the predicates. An example query is “how many names are similar to ‘Przymusinski’ in the author name column of a citation database allowing errors?” As will be shown shortly, these types of questions are central in query optimization of such queries in a relational database management system (RDBMS). We begin this chapter by surveying recent developments in approximate text processing, especially in RDBMSs. Then we consider the selectivity estimation of approximate text processing and see its main motivation, which is query optimization in RDBMSs. However, its application is not limited to query optimization in RDBMSs and we examine other applications.

We largely consider two categories of approximate predicates on text: selection operators and join operators. They are the “two workhorse operators” in RDBMSs. We give overviews of the sub-problems that we study, and then present challenges, goals and summary.

1.1 Approximate Text Processing

Text is ubiquitous and textual data are found in virtually every database. Examples of textual data include names, addresses, product descriptions, profiles, comments and reviews. They generally have distinctive features that require special handling in databases. Some of the important characteristics of textual data that are relevant to this dissertation are listed as follows:

- Typographical errors: Text often has typographical errors (typos). The major source of this type of error is user input. Even professionally edited text often has typos. With the advance of personal publishing mechanisms such as blogging, twitting or text messaging, more and more textual data are generated by end-users in a relatively short time and in large quantities. More errors are expected in these environments. Accordingly, there exist a large number of business products to handle this type of errors. For
instance, there are many specialized software products that clean textual data such as Trillium [137]. Typosquatting\(^1\) or websites like ‘Typo Bay’ \(^2\) are also examples.

- Diverse textual representations: In many cases, similar spellings exist for the same real-world entity. For instance, both ‘center’ and ‘centre’ are used and ‘Abbey’, ‘Abby’, and ‘Abbie’ are variants of similar names. It is hard to expect everyone to use the same standard in real-world and different textual representations are inherent even without typos.

The implication of above challenges is that exact matching for text often fails to deliver intended information. To address this problem, approximate text query processing techniques have been proposed in the literature, e.g., [53, 85, 114, 115]. It is also called fuzzy or similarity matching, or proximity search. Intuitively, it retrieves records that have similar text and allows errors or different textual representations. The problem, in its most general form, is “to find a text where a given pattern occurs, allowing a limited number of errors in the match” [114]. Examples of approximate text matching are recovering the original signals after their transmission over noisy channels, finding DNA subsequences after possible mutations, and text searching where there are typing or spelling errors. Each application has different similarity (or error) models and different characteristics of text. In this dissertation, we focus on the last example of text such as addresses, names or titles which are generally found in string columns in RDBMSs. See [86] for a survey on similarity measures and algorithms in this domain and [114] for approximate string matching techniques in a broader range of domains.

1.2 Selectivity Estimation for Query Optimization

With the widespread use of the Internet, more and more textual data are generated and stored with unprecedented rate. Accordingly, commercial RDBMSs have begun to incorporate approximate query processing in their core parts. For instance, Microsoft SQL Server Integration Services supports approximate text matching with ‘Fuzzy Lookup’ and ‘Fuzzy Grouping’ operators. Oracle Text supports new index types and several operators like ‘CONTAINS’. IBM DB2 Text Search also has related index types and operators for approximate text matching.

To successfully incorporate approximate text processing, database systems need many components. First, an index structure that enables efficient look-up is needed. Depending on the data type, RDBMSs build indexes like B+ tree, R-tree, or inverted index. Commercial database systems have developed specialized index structures for approximate text processing purposes. For example, Oracle Text has several index types including CONTEXT or CTXCAT depending on the type of queries. Second, operators or algorithms that can utilize the index and process data integrated with other operators are needed. The previously mentioned ‘CONTAINS’ operator is an example for approximate text processing in Oracle [117]

\(^1\)The controversial practice of registering misspelled variant of popular url

\(^2\)These sites help users reformulate queries so that they include popular typos of product names. This enables users to find hidden items that otherwise would not normally appear in the search result due to their typos.
1.2. Selectivity Estimation for Query Optimization

Figure 1.1: The optimizer generated plan (a) for the example query runs 3 times slower than an alternative plan (b) on the DBLP database. The estimated selectivity of the approximate text predicate (output of node 1 in plan (a)) was 1,722 whereas the true selectivity was 57,602. This poor selectivity estimation caused the selection of a sub-optimal plan such as plan (a) rather than plan (b).

or IBM DB2 [68]. Another crucial component, which is the most relevant to this dissertation, is the selectivity estimation module. Roughly speaking, selectivity of a predicate is the number of records satisfying the predicate. Selectivity estimation is essential in generating optimized query execution plans. A query optimizer has to make numerous choices during query optimization, many of which crucially depend on selectivity. It makes differences in many optimizing decisions including access path selection, join ordering or join algorithm. Despite the importance, for approximate text query processing, ad-hoc solutions have been employed in RDBMSs. An example of such ad-hoc solution is to use constant selectivity for any query. This ad-hoc approach is not ideal and has the risk of producing arbitrarily bad sub-optimal plans. The following real-world example illustrates this point.

Consider a citation database with the following schema:

\[ \text{paper}(\text{pid}, \text{title}, \text{year}, \text{venue}) \]
\[ \text{and} \]
\[ \text{pauthor}(\text{pid}, \text{name}) \]

Suppose that a user wants to find a paper by ‘Indyk’ published in 2008 but she is not sure about the spelling, or the database may not be clean. That is, the query is “find papers in 2008 which has an author similar to ‘Indyk’ allowing errors”. To decide whether two names are similar or not, we need a similarity measure and a similarity threshold. The following query showcases an actual SQL query in a commercial RDBMS that implements approximate text matching.

\[
\text{SELECT title, name FROM paper, pauthor}
\text{WHERE paper.pid = pauthor.pid AND}
\text{CONTAINS(name, fuzzy(Indyk, 60, 100, W)) > 0 AND year=2008}
\]

The approximate text matching is specified by ‘CONTAINS’ and ‘fuzzy’ keywords. The matching is based on edit-distance and the similarity threshold is 60.\(^4\) The query also has additional parameters to fine tune the query.

---

3 The selectivity of a predicate is often defined as the fraction of records that satisfy the predicate. We will use the term as the number of records since it does not make much difference in our context. We also use ‘cardinality’ or ‘size’ interchangeably.

4 The maximum similarity value is 80 in this database.
1.3 Other Applications

Figure 1.1 shows plans for the example query with the DBLP data set [93] loaded into the above schema. Figure 1.1(a) is the optimizer generated plan for the query. However, plan (a) is not the optimal plan for the query and took 10 seconds to run while plan (b) ran in 3 seconds on the DBLP database. After detailed analysis, the main reason for this sub-optimal plan choice was poor selectivity estimation of the approximate text predicate. There were 57,602 tuples satisfying the CONTAINS predicate in the database. However, the optimizer estimated selectivity of the predicate, which was the number of output tuples for node 1 in plan (a), was 1,722. Recall that using an index, we can efficiently identify and retrieve tuples satisfying some predicates with small overhead for index look-up. Thus, high selectivity, where there are only a small number of tuples satisfying the predicate, enables a much faster look-up than scanning and checking the whole data. However, at low selectivity, the overhead of index look-up becomes dominant and scan-based methods generally outperform index-based methods [124]. In the example, since the estimated selectivity was high, the optimizer chose a nested loop utilizing indexes, which turns out to be less efficient than a hash join with scans.

We can observe that just like other operators, operators for approximate text matching are integrated into the optimizing process and can affect the choice of optimal plans. As poor selectivity estimation can lead to arbitrarily bad sub-optimal plans, it is crucial to have accurate and efficient selectivity estimation techniques to successfully support approximate text matching in RDBMSs. This is the main motivation of this dissertation.

A possible alternative of processing approximate text query would be to clean the text first and load it into databases, which includes correcting typos or transforming data into standard representations. Then standard exact matching may work well. However, this may not be always possible or desirable depending on the tasks. First, cleaning all the data can be expensive and it may not be a feasible option. Second, even if it is possible, users may want to keep the original data. For instance, consider a user profile with textual data including hobbies or favorite movies. Users may not like it if the system automatically modifies their profiles without their consent. Moreover, it is hard to guarantee 100% correctness in data cleaning; users may want to conserve dirty data and perform approximate matching as necessary.

1.3 Other Applications

We illustrate two applications other than query optimization in RDBMSs where selectivity estimation of approximate text matching plays an important role.

1.3.1 Query Optimization in Text-related Tasks

Different algorithms often show different efficiency behaviors depending on the selectivity, skewness or distribution of data rather than an one-size-fits-all algorithm that outperforms others in all cases. Selectivity can guide the choice of algorithms in many of such cases.

Execution plans for text-centric tasks such as Information Extraction or Focused Resource Discovery can show different efficiency pattern depending on the selectivity. They follow two general paradigms: either we can scan, or crawl, the text database or we can exploit search engine indexes and retrieve documents of interest via carefully crafted queries constructed
1.3. Other Applications

in task-specific ways [73]. The choice between crawl- and query-based execution plans can have a substantial impact on both execution time and output completeness (e.g., in terms of recall). The query selectivity can be used as an important piece of information in deciding which execution plan to use [73]. For instance, when there are only a small number of results for a task, query-based execution plan can be sufficiently effective. When the result size is large, however, crawl-based strategy can outperform the query-based strategy with higher recall. Recent studies have noted that approximate text matching is important in those tasks as well [21, 97, 143].

Given a set of keywords, the goal of keyword search in RDBMSs, e.g., [4, 66, 122, 131, 151], is to find tuples connected by join relationship such that each keyword is found at least one tuple. There are also two general paradigms: either we can map the keyword query to a set of SQL queries and execute them, or we can merge tuples at a separate middleware module using RDBMSs only for retrieving raw column data. It is observed that when there are only a small number of answers the SQL-mapping approach can outperform the middleware based approach [66]. A hybrid approach is proposed in the work and selectivity is again used as a basis for the algorithm choice.

Join selectivity can affect the choice of algorithms in similarity joins or data cleaning as well. There have been many studies on similarity joins and they can be roughly categorized into two types: signature-based algorithms, e.g., [8], and inverted index based algorithms, e.g., [13, 130]. Signature based algorithms generally have two phases: (1) quickly identify candidate pairs using signatures and (2) verify candidate pairs by actually performing similarity computation. When the join size is small, signature based methods can be more efficient than other algorithms [8]. However, if the join size is large, naturally there are more candidate pairs and the pair-wise verification phase may become a bottleneck.

In all the above applications, selectivity of approximate text predicate plays an important role in the choice of execution algorithm. The proposed study can be valuable in the context.

1.3.2 Query Formulation

The selectivity (or cardinality) of an answer is important for users in many applications. When a user wants to manually browse through results, she would not want to either go through millions records or see empty results. For example, a user in a housing website would want to see an appropriate, not overwhelming, number of options. Formulating queries such that they can generate an appropriate number of results can be highly non-trivial, especially when there are more than one predicates [84]. It is very easy to generate too many answers or few answers, which is known as many/few answer problem [20, 84, 99]. To address this issue, techniques for specifying cardinality constraints have been proposed in the literature [20, 84]. Note that this is an orthogonal issue to top-k processing. Mishar and Koudas proposed a query design framework that explicitly takes into account user preferences about the desired answer size, and subsequently modifies the query with user feedback to meet this target [109].

In approximate queries, formulation of a good query can be trickier due to the similarity threshold. Similarity threshold is just a number, often between 0 and 1, and it generally does

---

5This definition corresponds to the AND semantics. There exist other semantics as well.
1.4. Dissertation Goals and Challenges

The goal of this dissertation is to develop reliable selectivity estimation techniques for approximate predicates on text. Desirable properties of such estimation techniques are as follows. First, the estimation has to be reasonably accurate. Poor selectivity estimation can lead to suboptimal plans with performance loss. Second, the estimation has to be efficient. When estimation is for query optimization, it has to be much more efficient than the actual query processing. Otherwise, the overhead for selectivity estimation is not justified. Third, smaller space overhead is desirable.

Among several approximate predicates, we focus on two types of operators: selections and joins. They are the two workhorse operators in RDBMSs. For selection operators, we first study *string* predicates and then *substring* predicates. For join operators, we first study *set* similarity joins and then *vector* similarity joins.

Similarity measures are at the core of approximate matching applications; similarity measures largely decide how to process strings and they affect the efficiency of an algorithm. Different similarity measures are used depending on the matching purposes. For instance, Jaro/Winkler distance is known to work best for person names [146]. Thus, we first briefly...
review similarity measures and string models. We then introduce each problem, its challenges and overviews of the solutions.

1.4.1 String Similarity Models and Measures

Similarity measures on string can be roughly categorized into three groups: edit based, token based and hybrid methods [86]. We overview representative string similarity models and measures. See [86] or [40] for more complete introduction on similarity measures and algorithms.

Edit distance, Soundex, Jaro/Winkler distances are examples of edit based measures. They model similarity matching of two strings by transforming one string into another one with *edit* operations. Edit distance (or Levenshtein distance) is a representative edit based measure. It defines three edit operations on characters: insertion, deletion or replacement (substitution) of a single character. Edit distance between two strings is the minimum number of edit operations to transform one string into the other one. For instance, the edit distance between ‘Silvia’ and ‘Sylvi’ is two because we can transform the former to the latter by one replacement (‘i’ to ‘y’) and one deletion (‘a’). There exist numerous variations from this basic definition. For instance, each operation may have different weights for different characters. This model is prevalent for strings like DNA sequences. In some extensions, different operations such as ‘gap’ or ‘swap’ are considered. In this dissertation, we focus on the basic edit distance (Levenshtein distance) as a base similarity measure since it is a basis for other variations and does not need domain specific knowledge.

TF-IDF cosine similarity, Jaccard coefficient, and probabilistic models [45] can be categorized as token-based methods. In this model, rather than directly modeling modifications or edits, a string is first decomposed into a set or vector of tokens, which are generally words or n-grams. An n-gram is simply a string of length n. For example, if we build 3-gram from ‘Microsoft’, we have ‘Mic’, ‘icr’, ‘cro’, ‘ros’, ‘oso’, ‘sof’, and ‘oft’. This tokenization enables non-exact matching. We can approximately match ‘Microsoft’ and ‘Macrosoft’ because they still share many of their 3-grams: ‘cro’,..., ‘oft’. Then (dis)similarity is measured with the distance between the two sets or vectors. For instance, a web document can be first tokenized into words and then represented by a vector where each dimension corresponds to a word with its TF-IDF value. Then we measure, for example, cosine similarity between their TF-IDF vectors. For vector representation, widely used (dis)similarity measures include cosine similarity, Euclidean distance, Manhattan distance and Hamming distance. For set representation, Jaccard similarity, dice similarity, and overlap count are representative ones. We support Jaccard similarity measure and show how to support other measures for set representation. We also support cosine similarity and \( \ell_p \) distance for vector representation. More sophisticated measures, mostly developed in IR community, include BM25 [126], language modeling [120], Hidden Markov Models [108].

A hybrid model combines the two models, e.g., [6, 25]. It generates tokens, but considers transforming between tokens so that finer modeling is possible. FMS [25] is one example of a hybrid model. Suppose two strings are tokenized as follows: {‘Boeing’, ‘Corporation’}, {‘Boeing’, ‘Company’}. FMS matches two token pairs ‘Boeing’ and ‘Boeing’, and ‘Corporation’ and ‘Company’, and considers edit distance between the pairs.
1.4.2 Selection Problems

In selection problems, a predicate, which is a query string and a similarity threshold for a given similarity measure, selects a set of records in the database. Our goal is to estimate how many records the predicate selects. That is, we estimate the number of strings in the database that satisfy the given similarity threshold; “how many strings are similar to the query string in the database?” As mentioned previously, there are many types of strings, and we focus on plain text that it not too long, say up to a few hundred characters. Such strings are found in typical text columns including names, addresses or titles. We do not consider non-plain-text strings such as signals or DNA sequences.

Based on how a predicate selects or matches strings in the database, we consider two different matching semantics: string and substring. We first define the STR problem with string matching semantics, and then define the SUBSTR problem with substring matching semantics.

The String Problem

When a string in the database is compared with the query string as a whole, we call the problem of estimating selectivity of approximate text matching the STR problem. It is defined as follows:

**Definition 1** (The STR Problem). Given a query string $s_q$ and a bag of strings $DB$, estimate the number of strings $s \in DB$ satisfying $ed(s_q, s) \leq \tau$, where $ed$ is edit distance between two strings and $\tau$ is the edit distance threshold.

For example, if $DB = \{'Silvia', 'Sylvi', 'Sofia'\}, q = 'Sylvia' and $\tau = 1$, then the answer is 2 because ‘Silvia’ and ‘Sylvi’ can be transformed into ‘Sylvia’ with 1 edit operation but ‘Sofia’ needs 3 edit operations.
The main challenge comes from the fact that there can be a huge number of possible variants of the query string with edit operations. Let us denote the cardinality of a string \( str \) in \( DB \) by \( \text{freq}(str) \) or \( |str| \) exploiting set cardinality notation. For example, \( |'rdbms'| \) is the number of strings ‘rdbms’ that appear in \( DB \). We can estimate the selectivity of a single variant using (exact) substring selectivity estimation algorithms, e.g., [26, 77, 87], with a small summary structure.

A naive approach using substring selectivity estimation techniques for the string problem would be to enumerate all possible variants and sum up their selectivities. For simplicity, suppose that \( q = 'rdbms' \) and we only allow one replacement. Then, assuming English alphabet ‘a’ to ‘z’, the desired answer is \( |'abms'| + |'bbms'| + |'cbms'| + \cdots + |'rdbmz'| \), which is the number of string in \( DB \) that can be converted to ‘rdbms’ with at most one replacement. However, this approach is not scalable at all. When \( q = 'rdbms' \) and \( \tau = 3 \), there are more than 4 million strings that are within edit distance of 3 from \( q \). In our experiments, it took more than 20 seconds to enumerate all variants, estimate each selectivity and sum them up, which is clearly too much for query optimization purposes.

To solve the problem, we introduce the wildcard ‘?’ that represents any single character. For instance, \( |'rdbm?'| \) is the number of strings that start with ‘rdbm’ and end with a single following character. We extend summary structures for exact substring selectivity estimation with wildcards and propose a new data structure called the \( EQ \)-gram table. This introduction of wildcard enables more efficient counting of strings. When \( q = 'rdbms' \) and we only allow one replacement, all possible variants are considered in \( |'?dbms'|, |'r?bms'|, |'rd?ms'|, |'rdb?s'| \) and \( |'rdbm?'| \). However, the difficulty is that there are overlaps among the counts. For example, ‘rdbms’ is counted in all 5 possible forms. We first develop a novel replace-only formula that considers this overlap among counts when only replacement is allowed. We then propose the Opt\( EQ \) algorithm that handles general edit operations. This contribution is shown in at the second row of Figure 1.3 marked STR, and our solutions for the STR problem are in Chapter 3.

**The Substring Problem**

In the substring problem, a string is the database is considered as a match if it contains a (sub)string that is similar to the query; it does not have to match with the query string as a whole. That is, our goal is to estimate the number of strings \( DB \) that contain a substring that is similar to the query string. This is useful when strings in \( DB \) are generally longer than expected query strings and specifying only part of the data string is desirable. For instance, in a paper title, users are more likely to issue queries using important keywords rather than the full title. The substring problem, SUBSTR, is formulated as follows:

**Definition 2** (The SUBSTR Problem). Given a query substring \( s_q \) and a bag of strings \( DB \), estimate the number of strings \( s \in DB \) satisfying \( \text{ed}(s_q, b) \leq \tau \), where \( b \) is a substring of \( s \) and \( \tau \) is the edit distance threshold.

For example, if \( DB = \{ 'Information Integration', 'Informatino Retrieval', 'Web Data Integration' \} \), \( q = 'Information' \) and \( \tau = 2 \), then the answer is 2 because each ‘Information Integration’
and ‘Informatino Retrieval’ contains a substring that can be converted to ‘Information’ within 2 edit operations but ‘Web Data Integration’ does not.

The substring problem is a generalization of the string problem. It has additional challenges on top of those of the STR problem: a string has many substrings, each of which may have many variants that can potentially match the given query string. Moreover, the substrings can overlap or be correlated which also needs special attention.

Based on information stored in an EQ-gram table, which is proposed for the string problem, we propose two estimation algorithms: MOF and LBS. MOF is a light-weight solution based on the idea that although many variants are possible for a given query string and a similarity threshold, there are typical variants. In the DBLP data set, most of names that are similar to ‘Sylvia’ within the edit distance of 1 are one of ‘Silvia’, ‘Sylvi’, or ‘Sylva’. It is less likely that we have strings like ‘SZlvia’ or ‘Syyvia’. Using the wildcard ‘?’ , more than 70% of the similar strings match the form ‘S?lvi’. Thus, MOF estimates that this is the most probable variant and scales up its estimated selectivity appropriately. A shortcoming of MOF is that its estimation relies only on a single selectivity without considering other variants. LBS overcomes this drawback by extending EQ-gram tables with Min-Hash signatures [17, 34] and considering overlaps with the Set-Hashing technique [30]. These relationships are shown at the third row in Figure 1.3 marked SUBSTR, and our solutions for the SUBSTR problem are in Chapter 4.

1.4.3 Join Problems

In join problems, we are given two collections of records, and we are to estimate the number of pairs, one from each collection, that satisfy the given predicate. When the two collections are the same, it is called a self-join. Given a similarity measure and a minimum similarity threshold, a similarity join finds all the pairs of objects in the database whose similarity under the measure is greater than or equal to the minimum threshold. Similarity joins have a wide range of applications including query refinement for web search [127], near duplicate document detection and elimination [17]. They also play a crucial role in data cleaning process which detects and removes errors and inconsistencies in data [8, 130]. Accordingly, the similarity join problem has recently received much attention [8, 13, 27, 60, 61, 130]. Our goal in this dissertation is to estimate the similarity join size.

Objects in similarity joins are often represented by sets or vectors. For instance, a text may be considered as a vector in a vector space for the cosine similarity measure. Depending on the representation, we distinguish the following two problems: the set similarity join size estimation problem and the vector similarity join size estimation problem. For the set representation, we focus on Jaccard similarity measure, and for the vector representation, we focus on cosine similarity measure.

The Set Similarity Join Size Problem

In the set similarity join size (SSJ) problem, an object is represented by a set. For instance, a large scale customer database may have many redundant entries which may lead to duplicate mails being sent to customers. To find candidates of duplicates, addresses are converted into
sets of words or n-grams and then a set similarity join algorithm can be used. A variety of (dis)similarity measures have been used in the literature such as Jaccard similarity and overlap threshold [13, 27, 130]. The Jaccard similarity for two sets \( r, s \), \( J_S(r, s) \) is defined as \( \frac{|r \cap s|}{|r \cup s|} \). It is one of the most widely accepted measures because it can support many other similarity functions [17, 27, 130]. Thus, we focus on the Jaccard similarity for our similarity measure.

Definition 3 (The SSJ Problem). Given a collection of sets \( R \) and a threshold \( \tau \) on Jaccard similarity \( J_S \), estimate the number of pairs \( SSJ(\tau) = |\{(r, s) : J_S(r, s) \geq \tau, r, s \in R \text{ and } r \neq s,\}| \).

For example, if \( DB=\{\{1, 2, 3, 4\}, \{1, 2, 3, 5\}, \{2, 3, 5, 6\}, \{4, 6\}\} \), and \( \tau = 0.5 \), then the join size is 2 because there are two pairs with \( J_S \geq 0.5: \{1, 2, 3, 4\}, \{1, 2, 3, 5\} \) and \( \{1, 2, 3, 5\}, \{2, 3, 5, 6\} \) among \( \binom{4}{2} \) possible pairs.

Several facts are worth mentioning in the problem formulation. First it corresponds to a self-join case. Although the join between two collections of sets is more general, many applications of set similarity join (SSJoin) are actually self-joins: query refinement [127], duplicate document or entity detection [17, 64], or coalition detection of click fraudsters [107]. Naturally, a majority of the SSJoin techniques proposed are evaluated on self-joins [8, 13, 27, 130]. Second, self-pairs \((r, r)\) are excluded in counting the number of similar pairs so that the answers are not masked by the big count, \( |R| \). Since \( J_S(r, r) = 1 \) for any \( r \), \((r, r)\) is trivially included in the answer size for any \( \tau \). The number of pairs satisfying a minimum threshold of interest is usually much smaller than the total number of self-pairs (i.e., \( |R| \)). Thus including the count will make it hard to fairly evaluate the proposed technique. In addition, we do not distinguish between the two different orderings of a pair (i.e., \((r, s) = (s, r)\)). It is, however, trivial to adapt the proposed technique to consider the self-pairs or the ordering of pairs in the answer if necessary. Regarding the universe of set elements, we consider integral domains such as integers or n-grams, which is natural for similarity measures for sets.

One of the key challenges in similarity join size estimation is that the join size can change dramatically depending on the input similarity threshold. While the join size can be close to \( n^2 \) at low thresholds where \( n \) is the database size, it can be extremely small at high thresholds. For instance, in the DBLP data set, while the selectivity of similarity join at \( \tau = 0.1 \) is larger than 30 %, the join selectivity is only about 0.00001 % at \( \tau = 0.9 \). While many sampling algorithms have been proposed for the (equi-)join size estimation, their guarantees fail in such a high selectivity range, e.g., [49, 58, 96]. Intuitively, it is not practical to apply random sampling in such a high selectivity. This is problematic since similarity thresholds between 0.5 and 0.9 are typically used [13].

The proposed solution analyzes Min-Hash signatures of the sets, which are succinct representations of them. We observe that there are close relationships between the number of frequent patterns in the signatures and the similarity join size. We rely on the recently observed Power-Law relationship between the number of frequent patterns and their support counts [32] for efficient estimation of the number of frequent patterns. We also show that there is a side effect in using the Min-Hash scheme for counting purposes; there is a distribution shift when data is skewed. We propose a method to correct this error. These contributions
1.4. Dissertation Goals and Challenges

are shown at the fourth row in Figure 1.3 marked SSJ, and our solutions for the SSJ problem are in Chapter 5.

The Vector Similarity Join Size Problem

In the vector similarity join size (VSJ) problem, an object is represented by a vector. For instance, a document can be represented by a vector of words in the document, or an image can be represented by a vector from its color histogram.

Definition 4 (The VSJ Problem). Given a collection of real-valued vectors \( V = \{v_1, ..., v_n\} \) and a threshold \( \tau \) on a similarity measure \( \text{sim} \), estimate the number of pairs \( \text{VSJ}(\tau) = |\{(u, v) : u, v \in V, \text{sim}(u, v) \geq \tau, u \neq v\}| \).

Note that our formulation of similarity joins with vectors is more general and can handle more practical applications. For instance, while in the SSJ problem a document is simply a set of words in the document, in the VSJ problem a document can be modeled with a vector of words with TF-IDF weights. It can also deal with multiset semantics with occurrences. In fact, most of the studies on similarity joins first formulate the problem with sets and then extend it with TF-IDF weights, which is indeed a vector similarity join.

A straightforward extension of SSJ techniques for the VSJ problem is to embed a vector into a set space. We convert a vector into a set by treating a dimension as an element and repeating the element as many times as the dimension value, using standard rounding techniques if values are not integral [8]. In practice, however, this embedding can have adverse effects on performance, accuracy or required resources [8]. Intuitively, a set is a special case of a binary vector and is not more difficult to handle than the vector. For instance, Bayardo et al. [13] define the vector similarity join problem and add special optimizations that are possible when vectors are binary vectors (sets).

We propose sampling based techniques that exploit the Locality Sensitive Hashing (LSH) scheme, which has been successfully applied in similarity searches across many domains. LSH builds hash tables such that similar objects are more likely to be in the same buckets. Our key idea is that although sampling a pair satisfying a high threshold is very difficult, it is relatively easy to sample the pair using the LSH scheme because it groups similar objects together. We show that the proposed algorithm \( \text{LSH-SS} \) gives good estimates at both high and low thresholds with a sample size of \( \Omega(n) \) pairs of vectors (i.e. \( \Omega(\sqrt{n}) \) tuples from each join relation in an equi-join) with probabilistic guarantees. Our key idea is to consider two different partitions of pairs of vectors that are induced by an LSH index: the pairs of vectors that are in the same buckets and those that are not. The opportunity is that it is relatively easy to sample the pair that are in the same buckets in the LSH index even at high thresholds.

We apply sampling algorithms separately for the two partitions such that we can provide provably reliable estimates at both high and low thresholds. This approach is in principle analogous to the classic approach [28, 41, 49] of distinguishing high frequency values from low frequency values to meet the challenge of skewed data. The proposed solution only needs minimal addition to the existing LSH index and can be easily applied. It is readily applicable to many similarity search applications. For the similarity measure \( \text{sim} \), we focus on cosine similarity [10], but in principle we can support any measure for which a corresponding LSH
1.5 Outline

There are four main technical components in this dissertation. Two are on the selection operators and the other two are on the join operators as described in the previous section. Each component comprises one chapter. The rest of this dissertation is organized as follows. In Chapter 2, we review related work. In Chapter 3, we present our solution for the STR problem. It also serves as a basis for following chapters. The EQ-gram table proposed in this chapter is also used in the substring problem, and the replace-only formula is adapted for set similarity join size estimation. In Chapter 4, we study the SUBSTR problem and propose two solutions: MOF and LBS. MOF which is a light-weight solution is based on EQ-gram tables, and LBS augments EQ-gram tables with Min-Hashing signatures. The following two chapters are on join problems. In Chapter 5, we study the SSJ problem. The proposed solution LC performs frequent pattern analysis on Min-Hash signatures of sets. In Chapter 6, we look at a generalized problem, the VSJ problem. The proposed technique performs stratified sampling using the LSH scheme. In Chapter 7, we conclude and present future work.

In summary, this dissertation proposes a set of related summary structures and algorithms to estimate selectivity of selection and join operators with approximate matching. The main contributions of this dissertation are as follows:

- We extended summary structures for exact text matching to support approximate matching and show how we can apply recent developments for approximate matching for the selectivity estimation purposes. We introduce the wildcard and Min-Hashing techniques to q-gram tables. A common challenge in the selectivity estimation of approximate matching is that there can be a huge number of variants to consider. The proposed data structures enable efficient counting by considering a group of similar variants together rather than each and every variant. A lattice-based framework is proposed to handle overlapping counts among the groups.

- Based on the framework, we present estimation algorithms for approximate selection and join operators. The proposed solutions show trade-offs between space and accuracy. We also conduct extensive experiments using real-world and synthetic data sets comparing them with state-of-the-art techniques and baseline methods.

- We present random sampling based algorithms for the vector similarity join size problem. The proposed algorithm is based on the idea of distinguishing high frequency values from low frequency values to meet the challenge of skewed data. It utilizes the LSH index, which is a recently developed technique for similarity search.
Chapter 2

Related Work

This chapter consists of three main components: (1) selectivity estimation techniques, (2) signature and hashing techniques and (3) query processing algorithms including joins.

We first review selectivity estimation techniques. There are many structure and algorithms for selectivity estimation depending on the data type, e.g., histograms or wavelets. Our focus is on techniques for strings. We categorize them into two groups: exact matching and approximate matching. Table 2.1 summarizes studies on selectivity estimation. Bold faced entities depict how a record is defined at each study. Notice that depending on similarity model (see Section 1.4.1), records are defined as strings, sets or vectors. Next, we study various signatures or hashing techniques for similarity matching. These techniques are building blocks for many similarity related algorithms and are used throughout this dissertation as well. We then turn our attention to query processing and introduce approximate string processing techniques. The existence of many approximate string processing algorithms motivates the selectivity estimation study for query optimization. As joins often pose additional challenges, we separately overview literature on similarity joins in Section 2.4.

2.1 Selection Selectivity Estimation

The problem of estimating selectivity of a string predicate is to estimate the number of records in the database that satisfy the given predicate. With the popularity of textual data, there have been many studies on estimating selectivity of string predicates. They are categorized into two groups: (1) exact matching and (2) approximate matching.

2.1.1 Exact Selectivity Estimation

The substring selectivity estimation problem is to estimate the number of tuples in a database that contain the query as a substring. Note that in exact matching semantics we do not consider errors or possible variants. The problem was studied in the context of SQL queries with the LIKE predicate (e.g., `name like 'jones'`). Initial studies considered only one substring predicate, but they are extended to multiple substring predicates in later work.

Single Substring Predicate

Krishnan et al. formulate the substring selectivity estimation problem [87]. The proposed method called KVI is based on a variant of suffix trees called count-suffix trees whose node is augmented with the count of occurrences of the associated substring. As the full suffix tree can be quite big, the pruned suffix tree (PST) is used, where nodes with count below a
2.1. Selection Selectivity Estimation

<table>
<thead>
<tr>
<th>Selection</th>
<th>Join</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exact Matching</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Substring</strong></td>
<td><strong>Equi-Join</strong></td>
</tr>
<tr>
<td>KVI [87, 142]</td>
<td>Adaptive Sampling</td>
</tr>
<tr>
<td>MO [75–77]</td>
<td>[95, 96]</td>
</tr>
<tr>
<td>CRT [26]</td>
<td>Tuple/Index/Cross</td>
</tr>
<tr>
<td></td>
<td>[58, 59]</td>
</tr>
<tr>
<td></td>
<td>Bifocal Sampling</td>
</tr>
<tr>
<td><strong>Approximate Matching</strong></td>
<td></td>
</tr>
<tr>
<td><strong>String</strong></td>
<td><strong>Set</strong></td>
</tr>
<tr>
<td>SEPIA [79]</td>
<td>LC [90], [Ch. 5]</td>
</tr>
<tr>
<td>VSol, HSol [105]</td>
<td></td>
</tr>
<tr>
<td>OptEQ [89], [Ch. 3]</td>
<td></td>
</tr>
<tr>
<td><strong>Set</strong></td>
<td><strong>Vector</strong></td>
</tr>
<tr>
<td>Hashed Samples [61]</td>
<td>LSH-SS [Ch. 6]</td>
</tr>
<tr>
<td><strong>Substring</strong></td>
<td></td>
</tr>
<tr>
<td>MOF, LBS [92], [Ch. 4]</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Selectivity Estimation Techniques on Text

threshold are pruned away. Given a query string, KVI tries to find the string in the PST. If the query is not present in the PST, the selectivity has to be estimated. The KVI estimation parses the query string into substrings that are found in the PST. Assuming independence among the parsed substring, it multiplies each selectivity. For instance, assume that ‘jones’ is the query string and it is not found in the PST but ‘jon’ and ‘es’ are in the PST. Then $\Pr(\text{‘jones’})$ is estimated as $\Pr(\text{‘jon’}) \cdot \Pr(\text{‘es’}) = (C_{\text{jon}}/N) \cdot (C_{\text{es}}/N)$, where $N$ is the total number of tuples in the database and $C_\alpha$ is the count of the node corresponding to $\alpha$ in the PST.

Jagadish et al. improve KVI with an algorithm called MO. The MO estimation removes the independence assumption of KVI based on the Markov assumption [77]. MO parses a string into maximally overlapping substrings, and multiplies conditional selectivities of parsed substrings given the overlaps. In the case of the previous example, assume that ‘jon’, ‘one’, ‘nes’, and ‘es’ are in the PST along with all single alphabets. Then MO will make use of the overlapping strings in the PST (i.e., ‘jon’, ‘one’, and ‘nes’) whereas KVI will parse it as ‘jon’ and ‘es’ without exploiting all the information (e.g., ‘one’) in the PST. MO estimates the selectivity as follows: $\Pr(\text{‘jones’}) = \Pr(\text{‘jon’}) \cdot \Pr(\text{‘e’|‘jon’}) \cdot \Pr(\text{‘s’|‘jon’}) \approx \Pr(\text{‘jon’}) \cdot \Pr(\text{‘e’|‘on’}) \cdot \Pr(\text{‘s’|‘ne’}) = (C_{\text{jon}}/N) \cdot (C_{\text{one}}/C_{\text{on}}) \cdot (C_{\text{nes}}/C_{\text{ne}}).$ Note that MO makes use of the overlapping of ‘on’ in ‘jon’ and ‘one’, and ‘ne’ in ‘one’ and ‘nes’. The KVI estimation will ignore the overlapping strings like ‘on’ or ‘ne’ and simply compute $P(\text{‘jones’})$ as $P(\text{‘jon’}) \cdot P(\text{‘es’})$ as above. Although the original algorithm is suggested on PST, it is applicable to other summary structures that can give the frequency of a substring in the database.

Chaudhuri et al. observe that MO often under-estimates and introduce the CRT algorithm [26]. It is based on the Short Identifying Substring (SIS) assumption, which states that “a query usually has a ‘short’ substring $s$ such that if a value contains $s$, then the attribute value almost always contains $s$ as well.” Given a string, the CRT algorithm estimates the frequency of identifying substrings of each length, and combines the frequencies using a
regression tree.

In Chapter 3.3, we propose a simpler solution called MO+ for the underestimation problem of MO. The CRT algorithm requires many invocations of substring selectivity estimation algorithms for the selectivity estimation of a string. Because we need to consider many variants, not just a single string, the additional runtime overhead of CRT is not desirable for the selectivity estimation of approximate string predicates. MO+ addresses the same underestimation issue, but requires much fewer invocations of substring selectivity estimation algorithms as its subroutines.

2.1. Selection Selectivity Estimation

Multiple Substring Predicates

Wang et al. generalize KVI and propose WVI estimation algorithm for two correlated columns [142]. The algorithms described above (KVI, MO, CRT) assume that strings are from a single column in a table. In Wang’s problem formulation, the query is conjunction of two substring predicates, one from each column, e.g., color like %green% and shape like %circle%. The idea is to build two suffix trees, one for each column, and represent the correlation between two columns by weighted edges between the two trees. The trees are pruned to reduce the size. The WVI estimation is very similar to KVI with an independence assumption. For example, if the query is (‘jon’, ‘172’) and ‘jo’, ‘n’, ‘17’, ‘2’ are in the suffix trees, $Pr(’jon’, ‘172’) \approx Pr(’jo’, ’17’) \cdot Pr(’n’, ’17’) \cdot Pr(’jo’, ’2’) \cdot Pr(’n’, ’2’).$ Their work can be extended to $k$-columns. In general, there are $k$ PSTs and a multi-dimensional table.

Jagadish et al. approach the same problem of multi-dimensional substring selectivity estimation and propose a $k$-D count-suffix tree [76]. The data structure is a generalization of a count suffix tree, where each node has $k$-dimensional strings as its label. The propose an extension of the MO estimation algorithm [77] in a $k$-D count-suffix tree.

Chen et al. propose a technique called Set Hashing for selectivity estimation of boolean predicates on substrings [30]. They consider multiple substrings with boolean predicates in 1-D column, e.g., color like %green% or color like %yellow%. The proposed technique stores Min-Hash signatures for nodes in a suffix tree. A basic method to calculate union and intersection size with signatures is presented, and then is extended to handle negations. When the query does not contain negations, Set Hashing transforms the original query into a conjunctive normal form (CNF) and estimates the size. With negations, it transforms the original query into a disjunctive normal form (DNF), removes negations using the set Inclusion-Exclusion principle, and estimates its size. In the PST case, it parses each string using MO and decomposes original probability into several probabilities. The proposed algorithm is exponential under negations or PST.

Chen et al. extend the technique to the multi-dimensional conjunctive case [31]. The problem with the previous approaches [76, 142] is that they approximate selectivity by explicitly storing frequently co-occurring combinations of substrings incurring exponential space increase with the number of dimensions. They side-step this space dimensionality explosion by set hashing signatures. The estimation details are very similar to those in [30].

These algorithms only consider exact matching and do not deal with selectivity estimation with edit distance or other similarity measures. However, the proposed data structures are
extended for approximate matching and algorithms are used as subroutines in our work. These studies deal with selectivity estimation of LIKE clauses, but do not consider clauses with ‘_’, which matches any single character. In Chapter 4, we show how to estimate LIKE predicates with the presence of both constructs.

### 2.1.2 Approximate Selectivity Estimation

As will be introduced in the following section, approximate string processing algorithms have recently received much focus in diverse fields. Naturally, efforts have been put toward developing good selectivity estimation algorithms allowing similarity matches.

Jin and Li formulate the problem of estimating string selectivity with edit distance [79]. Their technique, called SEPIA, clusters similar strings, selects a pivot string for each cluster, and captures the edit distance distribution with histograms. Given a query, SEPIA visits each cluster and estimates the number of strings in the cluster that are within the edit distance threshold using histograms. The drawback is that runtime computation of edit distance between the query string and the pivot string of each cluster can hamper its scalability.

Mazeika et al. propose a technique called VSol [105]. The VSol estimation employs an inverted index like data structure for q-grams. As the inverted list for a q-gram, i.e., the IDs of all strings that contain the q-gram, can be quite lengthy, Min-Hash [17, 34] signatures are stored instead to succinctly and approximately store the IDs. Our proposed algorithm, LBS, in Chapter 4 employs a similar approach. Similar ideas are often found in algorithms using Min-Hashing, e.g., [35]. The estimation idea is that a satisfying string must share at least a certain number of q-grams given an edit distance threshold $\tau$. They consider all combinations of possible q-grams that an answer string may contain. To save space, clustering is considered, and inclusion of positional information of q-grams is suggested to increase the accuracy. However, the positional information requires an amount of space that is higher by one or two orders of magnitude.

These studies are the most closely related body of work to the algorithms presented in Chapter 3 and Chapter 4. Chapter 4 addresses the STR problem as in [79, 105]. We compare the proposed solution with the prior art, SEPIA [79], in Chapter 3. Also note that these studies are on full string matching. So if we want to search through an address column, users have to input a whole address as the query. A more natural extension of related work in this context is approximate substring selectivity estimation problem which is addressed in Chapter 4. The substring problem poses additional challenges due to overlapping and correlation among substrings. Details are given in Chapter 4.

### 2.1.3 Join Size Estimation

Join size estimation has been one of core problems in selectivity estimation and known to be very difficult [71]. A majority of works in join size estimation considered equi-join where we join pairs of tuples that have the same value on the join column. It is studied in diverse contexts: query optimization in RDBMSs, e.g., [57, 59, 71, 95, 96, 133], stream data, e.g., [5, 22], spatial data [43], distributed data [119] or approximate query answering [1]. We briefly review studies that are more relevant to query optimization in RDBMSs.
2.1. Selection Selectivity Estimation

A simplistic approach is to use histograms with containment assumption [18, 133]. With the assumption, values on the join column are assumed to be uniformly distributed in each bucket of a histogram. It is very efficient since it only requires simple computations scanning buckets in the histograms. However, it is not expected to be accurate due to its assumptions within buckets.

Random sampling has been a major tool for join size estimation. Some examples include adaptive sampling [96], sequential sampling [59], cross/index/tuple sampling [58], bifocal sampling [49], tug-of-war [5] and more recently, end-biased sampling [41]. However, most of them have focused on equi-joins and do not directly deal with similarity joins. Some of them can be adapted to similarity joins, but their guarantees do not hold due to differences in sampling cost models or they require impractical sample sizes.

Equi-join size estimation techniques mostly do not offer clear benefits over the simple random sampling when applied to the similarity join size estimation problem. We note two challenges in the similarity join size estimation problems compared to the equi-join size estimation. In the equi-join of \(|R \bowtie S|\), we can focus on frequency distribution on the join column of each relation \(R\) and \(S\). For instance, if we know a value \(v\) appears \(n_r(v)\) times in \(R\) and \(n_s(v)\) times in \(S\), the contribution of \(v\) in the join size is simply \(n_r(v) \cdot n_s(v)\), i.e., multiplication of two frequencies. We do not need to compare all the \(n_r(v) \cdot n_s(v)\) pairs. In similarity joins, however, we need to actually compare the pairs to measure the similarity. This difficulty invalidates use of popular auxiliary structures such as indexes [28, 58] or (end-biased) histograms [28].

Furthermore, the similarity join size at high thresholds can be much smaller than the join size assumed in equi-joins. For instance, in the DBLP data set \((n = 800K)\), the join size of \(\Omega(n \log n)\) assumed in bifocal sampling is more than 15 million pairs and corresponds to the cosine similarity value of only about 0.4. In the most cases, users will be interested in much smaller join sizes and thus higher thresholds. However, we adapt those classic studies and use them as subroutines in Chapter 6 for the VSJ problem in Section 1.4.2. We outline two techniques that are closely related with the proposed techniques in the following.

Adaptive sampling is proposed by Lipton et al. [96]. Their problem formulation is very general and can handle similarity joins as well. Its main idea is to terminate sampling process when the query size (the accumulated answer size from the sample so far) reaches a threshold not when the number of samples reaches a threshold. A shortcoming is that when the data is highly skewed, adaptive sampling only gives an upper bound. The bound in similarity joins is a very loose bound at high similarity thresholds as will be shown in Chapter 6.4. Its implication is that adaptive sampling cannot handle high thresholds. While it does not produce reliable estimates in skewed data, we observe that its adaptive nature can still be useful for the VSJ problem. Adaptive sampling is used as a subroutine in our technique in Section 6.4.

Bifocal sampling is proposed by Ganguly et al. to cope with the skewed data problem\(^6\) [49]. It classifies tuples in the join relations into two groups; dense tuples that are frequent and sparse tuples that are not. They apply distinct estimation procedures that focus on various

\(^6\)When a join value is very sparse (infrequent) in one relation, but frequent in the other relation, it is not easy to sample pairs with such join values but their contribution to the join size may be significant.
combinations of joining tuples: dense-dense, dense-sparse, and sparse-dense. However, bifocal sampling cannot guarantee good estimates at high thresholds when applied to the VSJ problem. A join size of $\Omega(n \log n)$ is assumed in [49], but even a similarity threshold of 0.5 (e.g., in Jaccard or cosine similarity) can give a join size smaller than the assumed join size; Bifocal sampling does not overcome the difficulty of sampling at high thresholds in the VSJ problem. Our proposed techniques in Chapter 6 are analogous to their approach in that we apply different sampling procedures to handle high thresholds and low thresholds separately.

### 2.2 Signatures and Hashing Techniques for Similarity Matching

Intuitively, a signature is a succinct representation of an object. Signatures are often used in similarity search applications when directly comparing objects is expensive, e.g., comparing two images or web documents. An object in a similarity search application is often represented by a vector. In low-dimensional applications, there exist efficient solutions using space-partitioning methods or data-partitioning indexes such as grid-files, R-trees, Rf-trees or KD-trees [124, 135]. When the dimensionality is large, say greater than 10, they provide little improvement over a brute-force algorithm with a linear scan [70, 144]. However, when a small decrease in recall is acceptable, similarity hashing techniques can offer very efficient solutions. Similarity hashing is basically a hashing scheme that preserves some similarity measure. That is, the probability of two objects’ having the same hash value is proportional to (or highly correlated with) their similarity. Similarity hashing is often used as a way to construct a signature by concatenating multiple hash values. The dimensionality of a signature is in general much smaller than that of its original object.

Popular signature schemes include Mod-$p$ shingles [65, 132], shingles [17], Min-Hashing [16, 17, 34], prefix signature [8], k-signature scheme [21] and simhash [23] (They are not mutually exclusive). Signature-based algorithms often do not yield to an exact solution; objects that satisfy the given predicate might be missing from answers. However, this approximate nature is fine for estimation purposes and their efficiency makes them good tools for selectivity estimation.

We examine two signature schemes based on similarity hashing that we make use of: Min-Hashing and Locality Sensitive Hashing (LSH). See [104] for a brief survey on various signature schemes and [135] for more details on hash-based text retrieval. Min-Hashing supports Jaccard similarity and LSH is developed for several (dis)similarity measures. We briefly review related work on Min-Hashing and LSH, and defer technical details to Section 4.2 and Section 6.3.1 respectively.

#### 2.2.1 Min-Hashing

Min-Hashing is a similarity hashing scheme that preserves Jaccard similarity among objects. As defined in Section 1.4.3, Jaccard similarity (or resemblance) between two sets $A$ and $B$, $J_S(A, B)$ is $|A \cap B|/|A \cup B|$. Min-Hashing has been widely used in many similarity-related applications for its simplicity and effectiveness.
Hashing Technique

Cohen studies how to estimate the size of the reachable set for a node and the transitive closure (sum of such sizes) in a graph [34]. Given a graph and a node in the graph, the algorithm assigns a key ($\in \mathbb{R}_+$) drawn from some distribution (exponential or uniform) to each node. We keep the minimum key value among the keys of the reachable nodes from the node, and repeat this procedure. It is shown that the distribution of the minimum key is only dependent on the reachable set size and we can estimate the reachable set size of a node from the kept minimum keys. She proposes two estimators; (1) averaging based estimator which uses average of the minimums, (2) selection based estimator which uses only some of the minimums. Two asymptotic bounds, confidence and accuracy levels, are shown as well as the bias and variance of the estimator. Estimating the resemblance between reachability sets is mentioned as one of applications.

The Min-hashing technique relies on min-wise permutations (see [17, 34] for details). Diverse characteristics on min-wise independent permutations are dealt by Broder et al. [16]. The main focus is to analyze minimum sizes and the quality of approximation of various hash families including practical linear hashing families. Together with Cohen’s work, this paper provides theoretical backgrounds for the Min-Hashing technique.

Applications of Min-Hash Signatures

The application of Min-Hash signature is found in diverse areas including string selectivity estimation with edit distance [105], association rule mining [35], approximate string join [25], near duplicate document detection [17], selectivity estimation of boolean query [30] and audio fingerprinting [11].

The approximate substring selectivity estimation techniques in Chapter 4 relies on Min-Hash signatures. Technical details and its application are given in the chapter. Our proposed solution for the SSJ problem in Chapter 5 is also based on Min-Hash signature representation of a database.

2.2.2 Locality Sensitive Hashing

Locality Sensitive Hashing (LSH) is a special family of hash functions where close objects have higher collision probability. To build an LSH index, all the objects in the database are hashed into LSH hash tables\(^7\) by scanning the database. For the nearest neighbor search of a given query, we examine objects in the bucket where the query is hashed. The nearest object from the union of objects in the same bucket of all the hash tables is returned as the answer. Those buckets have objects that are more likely to be close to the query from the property of LSH. Several LSH schemes are developed for various (dis)similarity measures [7]. In fact, Min-Hashing is an LSH scheme that supports Jaccard similarity.

\(^7\)To increase the probability of finding similar objects in a bucket, usually many hash tables are employed.
2.3 Approximate String Matching

Hashing Technique

LSH is developed by Indyk and Motwani as an approximate solution for the Nearest Neighbor Search (NNS) problem [70]. It gives different collision probability to two objects based on their distance. If two objects are close, they are more likely to be in the same bucket. It achieves worst-case $O(dn^{1/\epsilon})$-time over an $n$-point database. This bound is enhanced to $O(dn^{1/(1+\epsilon)})$ and the algorithm is generalized and analyzed to the case of external memory in [52].

LSH Forest by Bawa et al. [12] improves upon LSH by (a) eliminating the different data-dependent parameters for which LSH must be constantly hand-tuned, and (b) improving on LSH’s performance guarantee for skewed data distributions while retaining the same storage and query overhead. The LSH Forest constructs a (logical) prefix tree on the all hash values, with each leaf corresponding to a point.

Panigrapy addresses the drawback of LSH that it needs a large number of hash tables in order to achieve good search quality [118]. The entropy-based LSH scheme is proposed to increase the utility of a hash table. At query time, the scheme generates random neighbor points and probes their buckets. Lv et al. further improve the entropy-based LSH [98]. As entropy-based LSH, it overcomes the drawback by probing multiple buckets that are likely to contain query results in a hash table. The key idea is to use a carefully derived probing sequence to check multiple buckets that are likely to contain the nearest neighbors of a query point.

Shinde et al. propose an improved LSH scheme in Euclidean space using a Ternary Content Addressable Memory (TCAM), which needs near linear space and has $O(1)$ query time [134]. TCAM is widely used in networking applications such as address lookups and access control lists and can query for a bit vector with wildcard ‘*’ representing either 0 or 1. They leverage TCAMs to design a variant of LSH, called Ternary Locality Sensitive Hashing (TLSH) wherein we hash database entries represented by vectors in the Euclidean space into \{0, 1, *\}. Their scheme enables much faster query times.

Applications

LSH is widely used in many applications for its efficient construction and query processing. Example include applications on text-retrieval [135], near-duplicate detection [104], image [81], and audio [98]. Algorithms that use Min-Hash signatures in the previous sections are in fact relying on LSH. Cohen et al. use double layers of LSH to identify association rules with extremely low support [35]. A Min-Hash signature is built for each transaction and is hashed into an LSH index. LSH searching algorithms are used to find rules with high confidence.

2.3 Approximate String Matching

Navarro provides a good survey on this large topic and classifies application areas into three major groups: (1) computational biology, (2) signal processing, and (3) text retrieval [114]. Strings in each area often have unique characteristics, so different string models and similarity measures have been used depending on the area. In this chapter, we focus on the text retrieval application. Even in the text retrieval area, studies have been performed under different names
in diverse fields: information retrieval [10, 113], pattern matching [38], data cleaning [25], record linkage (see [55] for a survey), and databases [53, 78, 94]. We introduce studies that are more closely related to this dissertation regarding text joins, data cleaning and query processing in databases.

2.3.1 Web Data Integration and Text Databases

Approximate string matching is central in data integration and detecting duplicated web documents. In the web environment, as the amount of data to be processed makes the exact processing impossible, many studies resort to hashing or signature techniques for efficient filtering that are introduced in the previous section.

Broder et al. study the problem of detecting “roughly the same” or “roughly contained” web documents [17]. The basic idea is to reduce these issues to set intersection problems that can be easily evaluated by a process of random sampling that can be done independently for each document. A succinct representation of document called *shingle* is proposed, which is basically a Min-Hash signature [34] of a document.

Jin et al. propose a methodology that utilizes known multidimensional indexing structures for the problem of approximate multi-attribute retrieval [78]. Their method enables indexing of a collection of string and/or numeric attributes to facilitate approximate retrieval using edit distance for string-valued attributes and numeric distance for numeric attributes. The proposed MAT tree is an R-tree like index structure for mixed type of string and numeric attributes to estimate combined selectivity of edit distance and numeric range predicates.

Jiong et al. study approximate substring search on large string databases [149]. They proposed a B-tree like index structure with regular expression label at a node. Given a query and weighted edit distance threshold, the index structure returns the set of substrings in the database whose similarity is above the given threshold.

Many algorithms are developed using fixed-length grams. Li et al. study the problem of how to choose high-quality grams of variable lengths from a collection of strings to support queries on a collection [94]. Specifically, they address how to select high-quality grams from the collection, how to generate variable-length grams for a string based on the preselected grams, and what is the relationship between the similarity of the gram sets of two strings and their edit distance.

There are several studies that focus on sets. Helmer and Moerkotte study the set containment join problem and propose a new signature based hash join [63]. Ramasamy et al. improve its efficiency by partitioning data. They propose the Partitioning Set Join Algorithm (PSJ), which uses a replicating multi-level partitioning scheme on a combination of set elements and signatures [125]. A major limitation of PSJ is that it quickly becomes ineffective as set cardinalities grow. Melink and Garcia-Molina generalize and extend the PSJ algorithm and propose two partitioning algorithms, called the Adaptive Pick-and-Sweep Join (APSJ) and the Adaptive Divide-and-Conquer Join (ADCJ) [106]. Mamoulis makes improvements over PSJ by building an inverted file for the “container” relation and using it to process the join [101]. These studies are different from the techniques in Section 2.4 in that they focus on equality or containment predicates, not similarity predicates.

Gionis et al. study indexing techniques for set-valued attributes based on similarity [51].
The proposed technique is based on embedding of the set collections into a space of vectors of fixed dimensionality using Min-Hashing. An additional embedding into the Hamming space with no distortion is used and the resulting Hamming space is indexed by introducing hash based schemes optimized for similarity querying. Mamoulis et al. propose a method that represents set data as signatures and organizes them into a hierarchical index, called SG-tree, which is suitable for similarity search and other related query types [102]. Unlike the study by Gionis et al. [51], they find the exact answers to queries. Terrovitis et al. address superset, subset, and equality queries on sets and propose the HTI index, which superimposes a trie-tree on top of an inverted file that indexes a relation with set-valued data [136]. Agrawal et al. study the indexing problem for the asymmetric Jaccard containment similarity function that is an error-tolerant variation of set containment [2]. They propose a parameterized index structure that explicitly stores inverted lists for token-sets that are minimal-infrequent: the frequency is less than a given parameter (infrequent) whereas the frequency of any subset is larger than this parameter (frequent).

### 2.3.2 Data Cleaning

Data cleaning, also called data cleansing or scrubbing, deals with detecting and removing errors and inconsistencies from data in order to improve the quality of data [123]. The practical importance of data cleaning is well reflected in the commercial marketplace in the form of the large number of companies providing data cleaning tools and services [128]. However, it is only recently that this topic has caught attention of research community since it was primarily viewed as a labor-intensive task. We selectively introduce recent movements toward automated data cleaning tasks. See [123] for a broader survey.

Sarawagi and Bhamidipaty present a learning-based deduplication system that uses a method of interactively discovering challenging training pairs using active learning [129]. The key insight is to simultaneously build several redundant functions and exploit the disagreement amongst them to discover new kinds of inconsistencies amongst duplicates in the data set, which is similar to active learning methods [36, 47].

Chaudhuri et al. propose a new string similarity function called FMS in [25]. The problem is, given a query tuple and a threshold, to find at most \( k \) most similar tuples from the reference table that are similar to the query tuple under FMS similarity and the threshold. FMS is a generalized edit distance which combines IDF and the edit distance measure. FMS considers token level edit operations instead of character level ones and assigns different weights to each token according to its importance captured by IDF. A relational table index is used for efficiently identifying candidate tuples. It stores selected q-grams of each column using Min-Hashing for the efficiency and score approximation. They also provide heuristics that can enhance the performance.

A number of studies have been proposed to exploit additional information beyond textual affinity and to exploit object-relationships or additional group information. Dong et al. exploit the associations between references to provide evidence for reconciliation decisions [39]. Kalashnikov et al. exploit inter-object relationships to improve the disambiguation quality [80]. Malin concentrates on ambiguity and investigates unsupervised methods which simultaneously learn (1) the number of entities represented by a particular name and (2) the
set of observations correspond to the same entity [100]. On et al. consider matching of highly similar groups of records and proposed a group linkage measure based on bipartite graph matching [116]. Chaudhuri et al. show how aggregate constraints (as opposed to pairwise constraints) that often arise when integrating multiple sources of data can be leveraged to enhance the quality of deduplication [29].

One direction of efforts is to reduce (or eliminate) manual tasks throughout the data cleaning process. They are beginning to be integral parts of database engines. For instance, FMS measure and related index features are integral parts of MS SQL Server [24]. To successfully coalesce these techniques with databases, it is vital to have a good selectivity estimator. The proposed works in Chapter 3 and Chapter 4 have contributions in this vein and algorithms in Chapter 5 and Chapter 6 are on similarity joins. They can also guide the choice of algorithms as explained in Section 1.3.1.

2.3.3 Other Areas

Bioinformatics

Many studies focus on approximate string matching in the computational biology community [56]. However, we believe that the string matching studies in bioinformatics are different in at least two key aspects. First, for efficient selectivity estimation, our techniques in Chapter 3 and Chapter 4 rely on the SIS assumption for database applications. It is doubtful whether the SIS assumption holds in a typical bioinformatics application. Second, for database applications, the alphabet size is typically much larger than 4 in DNA sequences. The larger alphabet size presents a harder problem to deal with, particularly in terms of efficiency.

Record Linkage

There have been intensive studies in the record linkage community (see [55, 146] for survey) to handle similar difficulties as in data cleaning [140, 145]. Even just for medical records, more than 100 papers have studied matching since the 1950s under the name “record linkage” [110]. However, they are mostly very domain specific and designed for particular applications. We briefly mention one of the general approaches: blocking technique [55, 82, 110]. It first clusters data and then considers matching of pairs within each cluster. The main purpose of blocking is to reduce the number of comparisons of record pairs by grouping records that have potential of matching. Blocking techniques assume the existence of a good blocking attribute that contain a large number of attributes values that are fairly uniformly distributed and have a very low probability of error. Errors in the blocking attribute can lead to missing true answers. Note that all these studies focus on finding matches (query processing) and do not consider selectivity estimation problems.

2.4 Similarity Joins

Intuitively, a similarity join is to find pairs of objects that are similar. Depending on objects of interest, similarity joins have been discussed in largely two contexts: spatial data and
2.4. Similarity Joins

textual data. In the former case, objects are generally geographical entities, and in the latter case, objects are texts. Although their problem formulations can be very similar, they often assume very different types of data and thus rely on different indexes. For instance, while R-trees or kd-trees (cf., for example, [124]) are primary index structures for spatial query processing, inverted indexes or tries are dominant for text-related processing. There are recent efforts that combine spatial and textual predicates with exact matching [44] or approximate matching [150], but they target selection predicates. See [74] for a detailed survey on spatial join techniques.

Near duplicate detection is very closely related to similarity joins, which aim to identify pairs of objects that differ in a very small portion. The problem is studied for objects of various types including web documents [104], files in a file system [103], emails [83], images [81] or domain-specific documents with different goals such as web mirrors, clustering, data extraction, plagiarism, and spam [104].

In this section, we focus on similarity joins that rely on a set (or vector) representation of objects without domain-specific knowledge and target primarily textual data. The set similarity join (SSJoin) is defined as follows: given two collections of sets and a similarity threshold, find all pairs of sets, one from each collection, whose similarity is greater than or equal to the threshold. The basic approach is to decompose a string into a set of grams or words. Then, the text join problem is solved by joining the sets. Chaudhuri et al. identify SSJoin as a key operator for various data cleaning operations [27]. Techniques developed for SSJoin can be roughly categorized into three groups: (1) inverted index-based solutions, (2) signature-based solutions, or (3) RDBMS-based solutions.

2.4.1 Inverted Index-Based Solutions

The inverted index from the IR community is a common data structure for SSJoin. For each set element, an inverted index stores the list of record (or object) ids that contain the element. A general approach is, for each set, to find similar sets by merging the id lists of elements in the set exploiting pruning heuristics.

Sarawagi and Kirpal formulate the SSJoin problem with overlap match, Jaccard coefficient, weighted match and cosine similarity [130]. They use the inverted index and propose heuristics for efficient merging of inverted lists. Several ideas for efficiently merging the lists are proposed. The first idea is to use the similarity threshold to avoid merging long inverted lists. In a self join case, they integrate index creation and probing processes. In addition to reducing data passes, a key advantage is that each probe step is performed on the partial list rather than the full list. Other heuristics, like pre-sorting data and clustering related records, are also suggested.

Bayardo et al. study similarity joins on a collection of vectors [13]. The studied problem is basically a weighted self-join. They develop several heuristics to improve inverted list merging efficiency. A key idea is to exploit the threshold during indexing. Rather than simply exploiting the threshold to reduce candidate pairs, they exploit the threshold to reduce the amount of information indexed in the first place. The proposed algorithms also make use of a specific sort order and the threshold during matching.

Xiao et al. propose an algorithm called PPJoin for SSJoin that combines positional fil-
2.4. Similarity Joins

tering with the prefix filtering-based algorithms [148]. PPJoin exploits the ordering of tokens in a record and leads to upper bound estimates of similarity scores. Xiao et al. observe that mismatching q-grams can provide information on the similarity of strings, and propose the ED-join algorithm for similarity joins of strings with edit distance threshold [147]. Traditional filtering methods are often based on the count of matching q-grams. They derive two new edit distance lower bounds by analyzing the locations and contents of mismatching q-grams. Hadjieleftheriou el al. concentrate on set similarity selection queries [60] and introduce simple variants of traditional TF-IDF and BM25 weighted measures [10] that exhibit certain very desirable semantic properties. An improved NRA [42], a DFS style algorithm, and a hybrid algorithm that exploit those semantic properties to prune the search space are proposed. They are not SSJoin algorithms but can be adapted to SSJoin easily.

2.4.2 Signature-Based Solutions

As described in Section 2.2, signatures are commonly used in similarity-related applications. Similarity join algorithms using signatures generally comprise two phases: (1) identify candidate pairs by some filters, and then (2) verify the pairs by measuring actual similarity. Many near duplicate detection techniques are also based on signatures [65, 81, 83, 104].

PartEnum is a signature-based SSJoin algorithm proposed by Arasu et al. [8]. It first scans the database building a signature for each set, and then performs joins on the signatures producing candidate pairs. Candidate pairs are verified by comparing actual sets. The key point is how to build the signature. PartEnum is based on two idea; (1) it partitions the dimensions so that matching pairs should match on at least one partition and (2) it enumerates various combinations of partitions to make the signature more selective. Hamming distance, Jaccard similarity, weighted join are considered.

Cohen et al. present a method to find association rules when the confidence is very high but the support is extremely low [35]. As briefly introduced in the previous section, their scheme is basically an approximate SSJoin algorithm. It first generates a compact representation of itemsets using Min-Hash signature [34], and then finds highly similar signatures by LSH [70].

2.4.3 RDBMS-Based Solutions

Several algorithms on similarity joins are designed for RDBMSs. The solutions are implemented using standard SQL or relational operators although they are based on similar filtering heuristics.

Gravano et al. develop a technique for building approximate string join capabilities on top of commercial databases by exploiting facilities already available in them [53]. Given an edit distance threshold, they employ cheap filters such as the number of matching q-grams, count filter and position filter. They are realized as SQL statements. This study is extended in [54] for text joins in an RDBMS for web data integration. A sampling-based join approximation strategy is proposed and cosine similarity metric from the information retrieval field is adapted. Their techniques work on a standard, unmodified RDBMS.

Chaudhuri et al. propose a new primitive operator called SSJoin operator which can be used as a foundation to implement similarity joins according to a variety of popular string
2.4. Similarity Joins

A primitive operator which supports the overlapping condition \(^8\) is defined and operators with other similarity measures are implemented by composing the basis operator with other relational operators. Their proposed techniques are implemented using a combination of standard relations operators such as group by, order by, and joins.

As identified in [27], similarity joins are at the core of many recent efforts for data cleaning or text join for RDBMSs. Again, the proposed techniques in Chapter 5 and Chapter 6 are valuable in implementing similarity joins in RDBMSs. They can also be used for choosing appropriate algorithms depending on the selectivity.

\(^8\)A similarity measure that counts the number of overlapping elements in two sets.
Chapter 3

String Selectivity Estimation

In the following two chapters, we study selectivity estimation of similarity selection queries. We consider two different matching semantics. In this chapter, we consider the ‘string matching’ semantics defined in Section 1.4.2. Intuitively, we aim to estimate the number of strings in the database that are similar to the query string. The framework presented in this chapter serves as a basis for later chapters. The proposed summary structure, the EQ-gram table is used for the SUBSTR problem in Chapter 4 as well. The concept of wildcard is extended and the replace-only formula is adapted for the join size estimation problem in Chapter 5. The proposed techniques in this chapter are based on our VLDB 2007 paper [89].

3.1 Introduction

In this chapter, we study the STR problem in Definition 1. In the STR problem, a query string $s_q$ and a similarity threshold $\tau$ select strings in the database that are similar to $s_q$ and the goal is to estimate the number of selected strings. The matching semantics is ‘string matching.’ That is, when we compare two strings, entire strings are compared.

For similarity measure, we use edit distance, which is one of the most widely accepted measures in text retrieval [10, 79]. The edit distance between two strings $s_1$ and $s_2$, denoted as $ed(s_1, s_2)$, is the minimum number of edit operations of single characters that are needed to transform $s_1$ to $s_2$. For instance, when the query string $s_q = \text{Michael}$ and $\tau = 1$, $s_q$ matches ‘Michal’ but does not match ‘Michal Tompton’. Thus, this semantics is suitable for strings that are not too long.

Chaudhuri et al. proposed the Short Identifying Substring (SIS) assumption stating that: "a query string $s$ usually has a ‘short’ substring $s'$ such that if an attribute value contains $s'$, then the attribute value almost always contains $s$ as well" in [26]. Thus, for estimating the frequency of $s$, it is desirable to estimate instead the frequency of $s'$. They show that the SIS assumption appears to hold for long strings in many real life data sets. Specifically, if one defines $s'$ to be a short identifying substring when its selectivity is within 5% of the selectivity of $s$, then there exist $s'$ with length $\leq 7$ for over 90% of strings $s$ of longer lengths. Thus, in this chapter, we focus our attention on queries $s_q$ with length shorter than 40. Given such queries $s_q$, we restrict our attention to low edit distance, more specifically $\tau = 1, 2$ or 3. For instance, if $s_q = \text{sheffey}$, the following seemingly unrelated strings are within an edit distance of 3: ‘hefte’, ‘yeffet’, ‘elfey’, etc. Thus, a fast solution for $\tau \leq 3$ can be valuable for many database applications. As a preview, we make the following contributions.

- We propose in Section 3.2 the concept of extended q-grams, which can contain the wildcard symbol $\text{?}$, representing any single character from the alphabet. Based on the
3.2. Extended Q-grams

3.2.1 Extending Q-grams with Wildcards

Let $\Sigma$ be a finite alphabet with size denoted as $|\Sigma|$. The bag $DB$ consists of strings drawn from $\Sigma^*$. To mark the beginning and the end of a string, we use the symbols ‘#’ and ‘$’, which are assumed not to be in $\Sigma$. For a string $s$, we prefix $s$ with ‘#’ and suffix with ‘$’. For example, “beau” is transformed into “#beau$” before processing. Throughout this chapter, we use double quotes for strings, but do not use any quotation marks for substrings. Whenever there is no confusion, we omit ‘#’ and ‘$’ for clarity.

Any string of length $q > 0$ in $\Sigma \cup \{‘#’, ‘$’\}$ is called a q-gram. An N-gram table over $DB$ consists of the frequencies of all q-grams for $1 \leq q \leq N$ [26]. If an entry $s$ in the table contains both # and $\$, the entry gives the frequency of the strings in $DB$ that are exactly $s$. Otherwise, the entry gives the frequency of the strings that contain $s$ as a substring.

For edit distance, edit operations considered are deletion (D), insertion (I) and replacement (R). Given a query string $s_q$, we use $\text{Ans}(s_q, iDjImR)$ to denote the set of strings $s'$ such that $s_q$ can be converted to $s'$ with at most $i(\geq 0)$ D(eletion) operations, $j(\geq 0)$ I(nsertion) operations and $m(\geq 0)$ R(eplacement) operations applied in any order. For example, $\text{Ans}(“abcd”, 2D1R)$ denotes the set of strings that can be converted from “abcd” with at most two deletions and one replacement applied in any order. The notation $2D0I1R$ is simplified to just $2D1R$. We also use the notation $\text{Ans}(s_q, k)$ to denote the set of strings $s'$ such that $s_q$ can be converted to $s'$ with at most $k$ operations when deletion, insertion and replacement are allowed, i.e.,

$$\text{Ans}(s_q, k) = \bigcup_{i+j+m=k \text{ and } i,j,m \geq 0} \text{Ans}(s, iDjImR).$$

For instance, $\text{Ans}(“abcd”, 1R)$ consists of the strings of one of the following forms: “?bcd”, “a?cd”, “ab?d” or “abc?”, where the wildcard symbol ‘?’ denotes a single character from $\Sigma$. To estimate the frequency for string matching with edit distance, we extend q-grams with the wildcard symbol. Any string of length $q > 0$ in $\Sigma \cup \{‘#’, ‘$’, ‘?’\}$ is called an extended...
3.2. Extended Q-grams

We generalize an N-gram table by keeping extended q-grams, and call it an extended N-gram table. For instance, a 3-gram table for the string “beau” contains 1-gram (i.e., for b, e, a, u individually), 2-grams (i.e., #b, be, ea, au, au$), and 3-grams (i.e., #be, bea, eau, au$, ??, #? and $?$. The additional 3-grams include (non-exhaustively) ?ea, #?e, etc. The entry ?ea gives the frequency of strings that include a substring of length 3 ending in ea. The entry #?e gives the frequency of strings that begin with the form ?e.

3.2.2 Replacement Semi-lattice

We introduce below a replacement semi-lattice and show how this structure can be used to derive a formula for estimating frequencies when only replacements are allowed. We assume all characters are distinct. We will also show how this formula can be exploited by the optimized algorithm OptEQ in Section 5.

Consider the set \( \text{Ans}(“abcd”, 2R) \), which consists of strings of one of the following forms: “ab??”, “a?c?”, “?bc?”, “a??d”, “?b?d” and “??cd”. Hereafter, we refer to these as the base strings of \( \text{Ans}(“abcd”, 2R) \). Formally, a base string for a query string \( s_q \) and the edit distance threshold \( k \) is any string \( b \) such that \( ed(b, s_q) = k \) and insertion/replacement modelled by ‘?’.

Intuitively, base strings represent possible forms of strings satisfying the query.

Let \( S_1 \) consist of strings of the form “ab??”, and \( S_2 \) of the form “a?c?”, and so on, then \( \text{Ans}(“abcd”, 2R) = S_1 \cup \ldots \cup S_6 \). Thus, the cardinality of \( \text{Ans}(“abcd”, 2R) \) can be computed using the Inclusion-Exclusion principle:

\[
|S_1 \cup S_2 \cup \ldots \cup S_n| = \sum |S_i| - \sum |S_i \cap S_j| + \sum |S_i \cap S_j \cap S_k| - \ldots + (-1)^{n-1} |S_1 \cap S_2 \cap \ldots \cap S_n|
\]

The problem is that this formula requires a number of terms which is exponential with respect to \( n \). Fortunately, as shown below, the structure of character replacements helps to collapse the formula significantly.

The structure of character replacements is captured in a replacement semi-lattice. Figure 3.1 shows the semi-lattice for \( \text{Ans}(“abcd”, 2R) \). A node in the lattice represents a base string or any string that is a result of intersection among base strings. The intersection is defined as follows; (i) the intersection of ? and ? gives ?; (ii) the intersection of ? and a character \( c \) gives \( c \); and (iii) the intersection of two characters \( c_1, c_2 \) is \( c_1 \) if \( c_1 = c_2 \), and empty.
3.2. Extended Q-grams

if \( c_1 \neq c_2 \). For example, intersection between “ab??” and “a?c?” gives “abc?”. The union operation is defined similarly. We define the level of a node to be the number of wildcard symbols in the node. Each of the aforementioned sets \( S_1, \ldots, S_6 \) is represented as a level-2 node. An edge in the semi-lattice represents an inclusion relationship between nodes in adjacent levels. For example, there in an edge between the level-2 node “ab??” and the level-1 node “abc?” and it indicates that the set \( S_7 \) consisting of strings of the form “abc?” is a subset of \( S_1 \) corresponding to “ab??”. Note that it is also a subset of \( S_2 \) and \( S_4 \). Six nodes at the top row (i.e., level-2 nodes here) in the lattice are base strings, and they give rise to four level-1 nodes, which correspond to \( \text{Ans}(“abcd” \cup 1R) \). In turn, they are parents of the single level-0 node “abcd”, which is the bottom element of the semi-lattice. As shown later, we exploit the property of a semi-lattice that the intersection between any two nodes results in a node at a lower level, decreasing the occurrence of wildcards by at least one.

3.2.3 An Example Replacement Formula

Continuing the example of \( \text{Ans}(“abcd” \cup 2R) \), there are 6 level-2 nodes, thus there are \( \binom{6}{2} = 15 \) ways of choosing two nodes for intersections between two nodes. We call these 2-intersections. An \( \ell \)-intersection is any intersection of \( \ell \) base strings. Among the fifteen 2-intersections, twelve correspond to level-1 nodes (e.g., “ab??” \( \cap “a?c?” = “abc?” \)), but three correspond to the level-0 node of “abcd” (e.g., “ab??” \( \cap “??cd” = “abcd” \)).

Let us continue with the 3-intersections of the six level-2 nodes. Some of the 3-intersections correspond to level-1 nodes. For example, the 3-intersection of “ab??”, “a?c?” and “?bc?” gives “abc?”). Among the \( \binom{6}{3} = 20 \) 3-intersections, four result in level-1 nodes, with the remaining sixteen resulting in the level-0 node. The table shown in Figure 3.2 also summarizes the numbers for 4-, 5- and 6-intersections. In the table, the row for level-1 indicates that twelve 2-intersections and four 3-intersections of base wild strings result in a level-1 node. One important observation is that given the highly regular structure of the semi-lattice, each level-1 node appears as results in exactly the same number of times. That is, among the twelve 2-intersections over four level-1 nodes, each level-1 node is the result of a 2-intersection three (\( = 12/4 \)) times. Similarly, among the four 3-intersections over four level-1 nodes, each level-1 node is the result of a 3-intersection once.

We can now calculate the size of \( \text{Ans}(“abcd” \cup 2R) \). Given a string \( s \), we abuse notation by using \( |s| \) to denote the size of the set \( \{t \mid t \in DB \text{ and } t \text{ is a string obtained by replacing all wildcards, if any, in } s \text{ with characters in } \Sigma\} \). By using the Inclusion-Exclusion principle and the numbers given in Figure 3.2 to simplify the calculation, we have:

\[
|“ab??” \cup “a?c?” \cup “a??d” \cup “?bc?” \cup “?b?d” \cup “??cd”| \\
= |“ab??”| + |“a?c?”| + \ldots + |“??cd”| \\
+(-3+1)(|“abc?”| + |“ab?d”| + |“a?cd”| + |“bcd”|) \\
+(-3+16-15+6-1)|“abcd”| \\
= F_2 + (-3+1)F_1 + (-3+16-15+6-1)F_0
\]

where \( F_i \) denotes the sum of the frequencies of all the level-\( i \) nodes. Thus, with the analysis on the replacement semi-lattice, the formula is significantly simplified.
3.2. Extended Q-grams

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
 & 2\text{-inter.} & 3\text{-inter.} & 4\text{-inter.} & 5\text{-inter.} & 6\text{-inter.} \\
\hline
\# at level-1 & 12 & 4 & 0 & 0 & 0 \\
\# at level-0 & 3 & 16 & 15 & 6 & 1 \\
\hline
\text{Total} & 15 \cdot \binom{\ell}{q} & 20 \cdot \binom{\ell}{q} & 15 \cdot \binom{\ell}{q} & 6 & 1 \\
\hline
\end{array}
\]

Figure 3.2: Number of Resulting Intersections for \textit{Ans}(“abcd”, 2R)

3.2.4 The General Replacement Formula

Below we generalize this analysis to give the cardinality of \textit{Ans}(s, kR), where the length of \(s\) is \(\ell\). There are \(\binom{\ell}{q}\) base strings, i.e., the number of strings with exactly \(k\) characters replaced by \(\ast\) in \(s\). Let \(B_{\ell,k,r}\) denote the number of \(r\)-intersections \((2 \leq r \leq \binom{\ell}{k})\) among the \(\binom{\ell}{k}\) base strings. It is easy to see that:

\[
B_{\ell,k,r} = \binom{\binom{\ell}{k}}{r} \tag{3.1}
\]

Among these \(r\)-intersections, let \(D_{\ell,k,r}\) denote the number of those that give \(s\), i.e., there is no wildcard contained in the \(r\)-intersection. For our example of \textit{Ans}(“abcd”, 2R), we have \(\ell = 4\) and \(k = 2\). Thus, there are \(B_{4,2,3} = 15\) 2-intersections, \(B_{4,2,3} = 20\) 3-intersections and so on (cf: the last row in Figure 3.2). Then \(D_{4,2,2}, D_{4,2,3}\) and so on correspond to the second last row of the table.

Let us take a closer look into \(D_{4,2,2}\) which is the number of 2-intersections giving \(s\). Since performing intersections of base strings always decreases the number of wildcards in the result string, the intersections for \(B_{4,2,2}\) can only contain strings with zero or one wildcard position. Thus, \(D_{4,2,2}\) is equal to subtracting from the total number of 2-intersections, which is given by \(B_{4,2,2}\), the number of 2-intersections with 1 wildcard position in the intersection.

Let \(s_1\) and \(s_2\) be the two non-identical base strings forming such a 2-intersection. Let the wildcard positions of \(s_1\) be \(p_{1,1}\) and \(p_{1,2}\) and those of \(s_2\) be \(p_{2,1}\) and \(p_{2,2}\). The two base strings must agree on exactly one wildcard position. Thus, without loss of generality, we can assume that \(p_{1,1} = p_{2,1}\) and \(p_{1,2} \neq p_{2,2}\). There are \(\binom{4}{1}\) combinations for choosing the common wildcard position \(p_{1,1}\). For the remaining positions, there must be 1 wildcard for \(s_1\) and 1 wildcard for \(s_2\) in different positions since \(p_{1,2} \neq p_{2,2}\). This is exactly as if we were counting the number of 2-intersections between \(s_1'\) and \(s_2'\) which were strings of length 3 with 1 wildcard character in different positions in each. This is \(B_{3,1,2}\), which automatically guarantees that \(p_{1,2} \neq p_{2,2}\) because the base strings always have different wildcard positions (i.e. \(p_{1,2} \neq p_{2,2}\)). Thus, in sum, \(D_{4,2,2} = B_{4,2,2} - \binom{4}{1} \ast B_{3,1,2}\). By using Eqn. (3.1) for the \(B\) terms, \(D_{4,2,2} = 15 - 4 \ast 3 = 3\) (cf: the 2-intersection column in Figure 3.2). By a similar analysis, \(D_{4,2,3}\) is given by \(B_{4,2,3} - \binom{4}{1} \ast B_{3,1,3}\), which gives \(D_{4,2,3} = 20 - 4 \ast 1 = 16\) (cf: the 3-intersection column). We have the following proposition for the general case.

**Proposition 1.** The general equation of \(D_{\ell,k,r}\) is:

\[
D_{\ell,k,r} = \sum_{i=0}^{\ell-k} (-1)^i \cdot \binom{\ell}{i} \cdot B_{\ell-i,k-i,r} \tag{3.2}
\]
3.2. Extended Q-grams

Proof. Given \( \ell \) and \( r \), define \( A_i, 1 \leq i \leq \ell \) to be the set of \( r \)-intersections that have a wildcard at position \( i \) at their results. Then \( \mathcal{A} = |\bigcup_{i=1}^{\ell} A_i| \) is the number of \( r \)-intersections that have at least one wildcard at their results. Using the Inclusion-Exclusion principle, \( \mathcal{A} \) can be computed as follows:

\[
|\bigcup_{i=1}^{\ell} A_i| = \sum_{i=1}^{\ell} |A_i| - \sum_{1 \leq i_1 < i_2 \leq \ell} |A_{i_1} \cap A_{i_2}| + \sum_{1 \leq i_1 < i_2 < i_3 \leq \ell} |A_{i_1} \cap A_{i_2} \cap A_{i_3}| - \cdots + (-1)^{\ell-1} |A_1 \cap \cdots \cap A_{\ell}|.
\]

Let \( S_m = \sum_{1 \leq i_1 < \cdots < i_m \leq \ell} |A_{i_1} \cap \cdots \cap A_{i_m}|. \) Then \( S_m = 0 \) for \( m \geq k \) because any \( r \)-intersection with \( r > 1 \) reduces the number of wildcards and there are \( k \) wildcards at each base string. That is, the set of \( r \)-intersections with \( k \) or more wildcards at its result is empty. Thus, \( \mathcal{A} = \sum_{m=1}^{k-1} (-1)^{m-1} \cdot S_m \).

Note that \( A_{i_1} \cap \cdots \cap A_{i_m} \) is the set of \( r \)-intersections that have wildcards at least at positions \( i_1, \cdots, i_m \). \( |A_{i_1} \cap \cdots \cap A_{i_m}| \) is the number of ways to arrange the remaining \( k - m \) wildcards to \( \ell - m \) positions. Since we are considering \( r \)-intersections, we have the following:

\[
|A_{i_1} \cap \cdots \cap A_{i_m}| = \binom{\ell - m}{k - m}.
\]

Since there are \( \binom{\ell}{m} \) ways of choosing \( i_1, \cdots, i_m \) such that \( i \leq i_1 < \cdots < i_m \leq \ell \), using Equation (3.1) in the above equation gives

\[
S_m = \binom{\ell}{m} \cdot B_{\ell - m, k - m, r},
\]

\[
\mathcal{A} = \sum_{m=1}^{k-1} (-1)^{m-1} \cdot \binom{\ell}{m} \cdot B_{\ell - m, k - m, r}.
\]

The number of \( r \)-intersections is \( B_{\ell, k, r} \), and there can be 0 to \( k - 1 \) wildcards. Thus, the number of \( r \)-intersections without any wildcards at their result is

\[
B_{\ell, k, r} - \mathcal{A} = \binom{\ell}{0} \cdot (-1)^0 \cdot B_{\ell, k, r} + \sum_{m=1}^{k-1} (-1)^m \cdot \binom{\ell}{m} \cdot B_{\ell - m, k - m, r}
\]

\[
= \sum_{i=0}^{i<k} (-1)^i \cdot \binom{\ell}{i} \cdot B_{\ell - i, k - i, r}.
\]

This proves the theorem because \( D_{\ell, k, r} = B_{\ell, k, r} - \mathcal{A} \) from the definition of \( D_{\ell, k, r} \).

Proposition 2. The cardinality of \( \text{Ans}(s_q, kR) \) is given by:

\[
|\text{Ans}(s_q, kR)| = F_k + \sum_{i=0}^{k-1} F_i \ast \sum_{r=2}^{i} (-1)^{r+1} D_{\ell - i, k - i, r}
\]

(3.3)
3.2. Extended Q-grams

Proof Sketch: Recall that $D_{\ell,k,r}$ gives the number of $r$-intersections with no wildcard contained in the intersection, i.e., level-0. We can apply the same reasoning for a node in level-$i$ ($1 \leq i \leq k$). A node in level-$i$ has exactly $i$ wildcards. There are $\binom{\ell}{i}$ possibilities to pick these $i$ wildcard positions. For the remaining positions, there must not be any wildcard in the $r$-intersection. Thus, $\binom{\ell}{i} \cdot D_{\ell-i,k-i,r}$ counts the total number of $r$-intersections resulting in a node in level-$i$. Recall that due to the regular structure of the semi-lattice, each node in this level occurs the same number of times as the result of $r$-intersections. Because there are $\binom{\ell}{i}$ nodes in level-$i$, dividing $\binom{\ell}{i} \cdot D_{\ell-i,k-i,r}$ by $\binom{\ell}{i}$ gives the number of $r$-intersections resulting in a specific node in level-$i$, which is $D_{\ell-i,k-i,r}$. Recall that $F_i$ gives the sum of the frequencies of all the level-$i$ nodes. The alternating sign comes from the Inclusion-Exclusion principle.

As a side benefit, this formula gives an exact answer for the hamming distance query. For small edit distance threshold $k$ and $l$ within, say, 40, the magnitude of these terms is manageable computationally. In our implementation, we use Java's BigInteger package for high precision.

3.2.5 The General Deletion Formula

Deletion operations do not insert any wildcards. This makes the string hierarchy and computation quite trivial. We use the example $Ans("abcd", 2D)$ as an illustration. Figure 3.3(a) gives the string hierarchy. There are six base strings, as the strings in $Ans("abcd", 2D)$ must be one of “ab”, “ac”, “ad”, “bd” and “cd”. Notice that as no wildcards are involved, any $r$-intersection ($r \geq 2$) gives the empty set. Thus, the hierarchy becomes a trivial single-level hierarchy:
3.3. Basic Algorithm for Size Estimation

The formula for one insertion

Next we turn to insertion operations. We begin with the example \( \text{Ans}("abcd", 1I) \). The string hierarchy is shown in Figure 3.3(b). There are five base strings, representing all the possible positions for the insertion. Note that only the 2-intersections between two adjacent base strings are non-empty.

\[
\begin{align*}
|\text{Ans}("abcd", 1I)| &= |"\text{abcdf}" \cup "\text{ab?c?d?}" \cup "\text{ab?cd?}" \cup "\text{abc?d?}" \cup "\text{abcd?!}"| \\
&= |"\text{abcdf}"| + |"\text{ab?c?d?}"| + |"\text{ab?cd?}"| + |"\text{abc?d?}"| + |"\text{abcd?!}"| \\
&- (|"\text{abcdf}"| + |"\text{ab?c?d?}"| + |"\text{ab?cd?}"| + |"\text{abc?d?}"|)
\end{align*}
\]

This extends to the general case: the size of \( \text{Ans}(s_q, 1I) \) is given by \( F_1 - F_0 \).

The analysis for \( \text{Ans}(s_q, kI) \) for \( k > 1 \) can be complex. Figure 3.3(c) shows the string hierarchy for \( \text{Ans}("abcd", 2I) \). The structure of the hierarchy is less uniform than that of a replacement semi-lattice. This motivates the machinery to be presented next. Specifically, we tackle the general case where:

\[
\text{Ans}(s_q, k) = \bigcup_{i+j+m=k \text{ and } i,j,m \geq 0} \text{Ans}(s, iDjImR).
\]

We develop the basic algorithm BasicEQ to estimate the size of \( \text{Ans}(s_q, k) \).

3.3.1 Procedure BasicEQ and Generating Nodes of the String Hierarchy

To estimate the size of \( \text{Ans}(s_q, k) \), the strategy is to partition \( \text{Ans}(s_q, k) \) by the length of the strings in \( \text{Ans}(s_q, k) \). Specifically, if the length of \( s_q \) is \( l \), then the strings in \( \text{Ans}(s_q, k) \) vary in length from \((l - k)\) to \((l + k)\). Thus, \( \text{Ans}(s_q, k) \) can be partitioned – according to the length of strings – into \((2k + 1)\) non-overlapping subsets. The size of each of these \((2k + 1)\) partitions is then estimated. Figure 3.4 shows a skeleton of this algorithm.

To illustrate, consider \( \text{Ans}("abcde", 2) \). This answer set can be partitioned into five non-overlapping subsets consisting of strings of length from 3 to 7. Answer strings of length 3 are all contained in \( \text{Ans}("abcde", 2D) \). This being a pure case, it can be handled directly by the formula given in Section 3.2.5. Similarly, answers of length 7 are all contained in \( \text{Ans}("abcde", 2I) \). This is also a pure case and a formula for determining its size will be given later in this section.

Line (7) of Figure 3.4 deals with all the general cases when the formulas in the previous section cannot be used directly. For instance, for \( \text{Ans}("abcde", 2) \), answer strings of length
3.3. Basic Algorithm for Size Estimation

**Input:** query $s_q$ of length $l$, edit distance threshold $k$

**Output:** estimated frequency $c$

1: $c = 0$
2: for len $= l - k$ to $l + k$ do
3:   find all combinations $(i, j, m)$ for $\text{Ans}(s_q, iDjImR)$ such that $i + j + m = k$ and $l - i + j = \text{len}$
4:   if $\text{Ans}(s_q, iDjImR)$ is one of the special cases then
5:     get $c'$ from the corresponding algorithm
6:   else
7:     $c' = \text{BasicEQ}(s_q, k, \text{len})$
8:   end if
9:   $c = c + c'$
10: end for
11: return $c$

Figure 3.4: A Skeleton for Estimating Frequency

5 are contained in $\text{Ans}("abcde", 2R) \cup \text{Ans}("abcde", 1D1I)$. While we have a formula for $\text{Ans}("abcde", 2R)$, we do not have a formula for $\text{Ans}("abcde", 1D1I)$. Even if we had both formulas, estimating the sizes of the two sets separately and adding the two estimates would give a large error because the two answer sets overlap significantly. Thus, we have to resort to a combined treatment of the two sets, i.e., operating from a single combined string hierarchy. The set of base strings is first computed based on either two replacements, or one deletion and one insertion. During the computation, all redundant base strings are removed. For example, $\text{Ans}("abcde", 2R)$ produces a base string “abc??” while $\text{Ans}("abcde", 1D1I)$ generates a base string “abcd?” . However, “bc??” contains all strings represented by “abcd?” . Thus, we delete “abcd?” in the base strings for the combined string hierarchy. From then on, the procedure BasicEQ generates the string hierarchy and compute the estimate.

**Procedure BasicEQ:** A skeleton of Procedure BasicEQ is presented in Figure 3.5. Given a query $s_q$ and edit distance threshold $k$, it gives a frequency estimate of all the answer strings of a specific length len. Line (2) of the procedure generates all the base strings for len. The set baseNodes is the set of base strings that remain after removing redundant base strings as above. Then the bulk of the computation is to generate the nodes of the string hierarchy.

A naive approach is to perform this generation by considering all $r$-intersections ($2 \leq r \leq |\text{baseNodes}|$) of the base strings in baseNodes. Obviously, due to the exponential nature, this is impractical. Instead, the procedure generates the $r$-intersections in a level-wise fashion. The algorithmic structure follows the well-known Apriori algorithm in [3]. The procedure first generates 2-intersections, such as $u \cap v$, $v \cap w$ and $u \cap w$ and so on. An $r$-intersection is non-empty only if all of its $(r - 1)$-intersections among base strings appearing in the $r$-intersection are non-empty. This explains why it is sufficient to consider only new nodes in line (6).

However, the apriori strategy alone is not sufficient, and the following optimization is
3.3. Basic Algorithm for Size Estimation

**Input:** query \(s_q\) of length \(l\), edit distance threshold \(k\)
- length \(l\) of answer strings

**Output:** estimated frequency \(c\)

1. find all combinations \((i, j, m)\) for \(\text{Ans}(s_q, iDjImR)\)
   such that \(i + j + m = k\) and \(l - i + j = \text{len}\)
2. generate the set \(\text{baseNodes}\) of non-redundant base strings for the combinations above
3. initialize \(\text{newNodes}\) as \(\text{baseNodes}\), \(\text{totalNodes}\) as \(\varnothing\), \(c\) as 0
4. repeat
   5. \(\text{tmpNodes} = \varnothing\)
   6. for all \(u \in \text{newNodes}\) and \(b \in \text{baseNodes}\) do
      7. if \(u \cap b \neq \varnothing\) then
         8. \(\text{tmpNodes} = \text{tmpNodes} \cup \{u \cap b\}\)
      end if
   end for
   9. \(\text{newNodes} = \text{tmpNodes} - \text{totalNodes}\)
   10. \(c = c + \text{PartitionEstimate}(\text{newNodes})\)
   11. \(\text{totalNodes} = \text{totalNodes} \cup \text{newNodes}\)
   12. until \(\text{newNodes}\) does not contain any \(u\) with a wildcard
13. return \(c\)

Figure 3.5: A Skeleton for Procedure BasicEQ

critical. In order for a new intersection \((u \cap b)\) to be non-empty, \(u\) must contain at least one wildcard. If there is no wildcard in \(u\), an additional intersection to it is either itself or the empty string. As each iteration of the repeat-until loop of the procedure decreases the number of wildcards by at least one and the edit distance threshold \(k\) is not large, the procedure halts with a small number of iterations.

3.3.2 Node Partitioning

Line (12) of BasicEQ attempts to estimate the frequency of every newly generated node in the hierarchy. Recall from the previous section that the frequency contribution of a node \(q\) in the hierarchy is a combination of \(C_q \cdot |q|\), where coefficient \(C_q\) denotes the number of times \(q\) appears in all the intersections of base strings, and \(|q|\) denotes the frequency of string \(q\) occurring in the database \(DB\). For instance, recall from the example \(\text{Ans}(\text{“abcd”}, 2R)\) in Section 3.2.3 that if \(q\) is the node “abcd”, \(C_q\) is \((-3+16-15+6-1) = 3\). The same example also illustrates that many nodes \(q\) have the same coefficient \(C_q\). For the example \(\text{Ans}(\text{“abcd”}, 2R)\), all the nodes in the same level of the semi-lattice have exactly the same coefficient (e.g., the coefficient for \(F_1\) is \(-3+1 = -2\)).

The same principle applies to a general string hierarchy. The goal of PartitionEstimate is to group the nodes of the hierarchy into partitions, so that every node \(q\) in a partition has the same coefficient \(C_q\). In that case, it is sufficient to compute the coefficient of all the nodes in the partition only once. Below we give a sufficient condition for partitioning.

Given a string hierarchy \(H\) and a node \(q\) in \(H\), the local semi-lattice of \(q\) is defined to be the
3.3. Basic Algorithm for Size Estimation

Let us consider the example of Ans(“abcd”, 2I). Recall that Figure 3.3(c) shows the string hierarchy. As examples of local semi-lattices, Figure 3.6 shows the local semi-lattices of “ababcd”, “aaabcd” and “aabbcd”. To illustrate how BasicEQ operates, recall that every 2-intersection node is generated in its first iteration. Thus, all the level-0 and level-1 nodes in Figure 3.6(a) are produced in the first iteration (e.g., “ab??cd” = “aa??cd” ∩ “??abcd”). However, “aaabcd” in Figure 3.6(b) is generated in the second iteration because no 2-intersection can generate that node.

To illustrate node partitioning, all the level-0 nodes of the hierarchy shown in Figure 3.3(c) can be grouped into the three partitions as shown in Figure 3.6. The first partition, shown in Figure 3.6(a), consists of the nodes with a local semi-lattice isomorphic to that of “ababcd”. (Below we refer to this partition as \( P_{0,1} \).) Every node in this partition appears once in 2-intersections and once in 3-intersections of base strings with alternating sign, giving rise to the coefficient of \(-1+1=0\). For example, among all possible 2-intersections of base strings in Figure 3.3(c), only “ab??cd” ∩ “??abcd” results in “ababcd”.

The second partition illustrated in Figure 3.6(b) consists of the nodes with a local semi-lattice isomorphic to that of “aaabcd”. (Below we refer to this partition as \( P_{0,2} \).) Every node

Figure 3.6: Examples of Local Semi-lattice

sub-hierarchy of \( H \) that includes only nodes that are ancestors of \( q \) and \( q \) itself. By definition, \( q \) is the bottom element, or the root, of the semi-lattice. For instance, if \( H \) is the semi-lattice shown in Figure 3.1, \( q_1 \) is the node “abcd” and \( q_2 \) is “abc?”, then the local semi-lattice of \( q_1 \) is the entire semi-lattice \( H \), and the local semi-lattice of \( q_2 \) is the sub-hierarchy with “abc?” as the bottom node and the 3 parents being “ab??”, “a?c?” and “?bc?”.

We have the following proposition stating that local semi-lattices can form the basis of node partitioning.

**Proposition 3.** For nodes \( q_1, q_2 \) in string hierarchy \( H \), if the local semi-lattices rooted at \( q_1, q_2 \) are isomorphic, then \( C_{q_1} = C_{q_2} \).

**Proof Sketch:** When computing \( C_{q_1} \) using Equation (3.2), for any intersection that gives \( q_1 \), there is a corresponding intersection that gives \( g_2 \) because of the isomorphism. The same arguments hold for \( C_{q_2} \). Thus, \( C_{q_1} = C_{q_2} \).

The above proposition can be proved by induction on the depth of the nodes in the semi-lattices. The converse of the proposition is not true, namely \( C_{q_1} = C_{q_2} \) does not imply isomorphic local semi-lattices.

3.3.3 An Example and a Formula for Two Insertions

Let us consider the example of Ans(“abcd”, 2I). Recall that Figure 3.3(c) shows the string hierarchy. As examples of local semi-lattices, Figure 3.6 shows the local semi-lattices of “ababcd”, “aaabcd” and “aabbcd”. To illustrate how BasicEQ operates, recall that every 2-intersection node is generated in its first iteration. Thus, all the level-0 and level-1 nodes in Figure 3.6(a) are produced in the first iteration (e.g., “ababcd” = “aa??cd” ∩ “??abcd”). However, “aaabcd” in Figure 3.6(b) is generated in the second iteration because no 2-intersection can generate that node.

To illustrate node partitioning, all the level-0 nodes of the hierarchy shown in Figure 3.3(c) can be grouped into the three partitions as shown in Figure 3.6. The first partition, shown in Figure 3.6(a), consists of the nodes with a local semi-lattice isomorphic to that of “ababcd”. (Below we refer to this partition as \( P_{0,1} \).) Every node in this partition appears once in 2-intersections and once in 3-intersections of base strings with alternating sign, giving rise to the coefficient of \(-1+1=0\). For example, among all possible 2-intersections of base strings in Figure 3.3(c), only “ab??cd” ∩ “??abcd” results in “ababcd”.

The second partition illustrated in Figure 3.6(b) consists of the nodes with a local semi-lattice isomorphic to that of “aaabcd”. (Below we refer to this partition as \( P_{0,2} \).) Every node

38
3.3. Basic Algorithm for Size Estimation

**Input:** `newNodes`, a set of newly formed nodes

**Output:** estimated frequency `c`

1. for all node `q` ∈ `newNodes` do
2. form the set `Par_q` of all the parents of `q`
3. form the multiset `PID_q` of the partition id of parents in `Par_q`
4. form the set `B_q` of all the base strings which are ancestors of `q`
5. end for
6. given `newNodes = {q_1, . . . , q_t}` for some `t > 0`
   - put `q_i, q_j (1 ≤ i, j ≤ t)` in one partition if the multisets `PID_q_i` and `PID_q_j` are identical and `|B_q_i| = |B_q_j|`
7. update the global partition table and `c = 0`
8. for all partition `P` formed in the previous step do
9. `C_p = ComputeCoefficient(P)`
10. `c = c + C_p * ∑_{q ∈ P} EstimateFreq(q)`
11. end for
12. return `c`

Figure 3.7: A Skeleton of Procedure PartitionEstimate

`q` in this partition does not appear in its local semi-lattice as a 2-intersection, but appears once as a 3-intersection, giving a coefficient `C_q` of 1.

Finally, the third partition presented in Figure 3.6(c) consists of nodes with a local semi-lattice isomorphic to that of “aabbcd” (Below we refer to this partition as `P_{0,3}`). With a similar argument, the coefficient for each node in this partition becomes `-2 + 4 − 1 = 1`.

It is not hard to verify that |`Ans("abcd", 2I")`| is given by `F_2 − F_1 + F_{0,2} + F_{0,3}`, where `F_i` give the sum of frequencies of all the nodes in level-`i` for `i = 1, 2,` and `F_{0,1}, F_{0,2}, F_{0,3}` denote the sums of the frequencies in partitions `P_{0,1}, P_{0,2}` and `P_{0,3}`. This result in fact generalizes to arbitrary length `s_q` as the following:

**Proposition 4.** The cardinality of `Ans(s_q, 2I)` is given by `F_2 − F_1 + F_{0,2} + F_{0,3}`.

Proof Sketch: With two insertions, the length of nodes is `|s_q| + 2`. The coefficient of all level-2 nodes is 1. Note that all level-1 nodes have one wildcard and one character that is repeated twice, which implies that the coefficient of all level-1 nodes is -1. Nodes at level 0 can have either a single character that is repeated three times or two characters that are repeated twice. In the former case, the local lattice is isomorphic to Figure 3.3(b). In the latter case, the local lattice is isomorphic to either Figure 3.3(a) or (c) depending on the relative order of the two characters. Considering the coefficients of all levels, we have the above formula for `Ans(s_q, 2I)`.

3.3.4 Procedure PartitionEstimate

In order to form partitions to use Proposition 3, an isomorphic test on the structure of the local semi-lattices is required. A brute-force approach is computationally expensive. Fortunately, in our framework, this test can be integrated into the level-wise computation of `BasicEQ`. As
3.3. Basic Algorithm for Size Estimation

**Input:** a partition $P$

**Output:** estimated frequency $C_P$

1. take any node $q$ in partition $P$ and set $\text{compBase}$ consisting of all the base strings which are ancestors of $q$
2. initialize $rInter = \{\{b\} | b \in \text{compBase}\}$; $r = 2$; $C_P = 0$
3. while $rInter \neq \phi$ and $r \leq |\text{compBase}|$ do
   4. $\text{tmpInter} = \phi$, $C'_P = 0$
   5. for all $\text{base} \in \text{compBase}$ and $\text{inter} \in rInter$ and $\text{base} \notin \text{inter}$ do
      6. $\text{tmpInter} = \text{tmpInter} \cup \{\text{inter} \cup \{\text{base}\}\}$
      7. if $\text{intersect}(\text{inter}) = q$ then
         8. $C'_P = C'_P + 1$
      end if
   end for
   9. $C_P = C_P + (−1)^{r+1}C'_P$
   10. $rInter = \text{tmpInter}$; $r = r + 1$
end while
11. return $C_P$

Figure 3.8: A Skeleton of Procedure ComputeCoefficient

newly created nodes are computed in each iteration of $\text{BasicEQ}$, all these nodes are passed to $\text{PartitionEstimate}$, of which a skeleton is presented in Figure 3.7. Each node generated by $\text{BasicEQ}$ is assigned a partition id. For bookkeeping purposes, there is a partition table which maps a node to its partition id. There is a second table that maps a partition id to the set of nodes contained in the partition. In our experimentation, there are typically fewer than 40 partitions; thus, lookups can be done very quickly.

When $\text{PartitionEstimate}$ starts, all the base strings are in one partition because each local semi-lattice consists of just itself. Then inductively, to check whether the local semi-lattices of two nodes $q_1, q_2$ have the same structure, it checks whether the partition ids of these two multisets of parents and the number of base strings that are ancestors are identical. These are necessary but not sufficient conditions for isomorphism of two lattices. In Section 6, the empirical results will show that the two conditions are rather effective heuristics.

It is possible that both a node and its parents are generated, put into $\text{newNodes}$, and fed as input in the same invocation of $\text{PartitionEstimate}$. However, as ancestors cannot be generated later than their descendants, level-wise processing in the first for-loop in Figure 3.7 insures that the $\text{PID}$ of parents are available when it processes a node. This detail is omitted in Figure 3.7.

Once the partitions are formed, then $\text{PartitionEstimate}$ invokes $\text{ComputeCoefficient}$ in line (9) to compute the coefficient $C_P$ for each newly formed partition $P$, a procedure to be detailed in the next subsection. Once $C_P$ is computed, line (10) multiplies this coefficient to the frequency of each node in the partition $P$. While every node in a partition $P$ has the same coefficient $C_P$, each may have its own frequency of occurrence in the database $DB$. Estimating these frequencies will be discussed at the end of this section.
3.3.5 Procedure ComputeCoefficient

Given a partition $P$ of nodes computed by $\text{PartitionEstimate}$, the procedure $\text{ComputeCoefficient}$ shown in Figure 3.8 computes the coefficient $C_P$, the number of times a node $q \in P$ appears in all the $r$-intersections of the base strings. Essentially, it starts with the set $\text{compBase}$ of base strings that are ancestors of $q$ in the hierarchy. Then the algorithm iterates to find $r$-intersections, starting from $r = 2$. The set $r\text{Inter}$ consists of all non-empty $r$-intersections. In other words, a set $\text{inter}$ in $r\text{Inter}$ represents a particular non-empty $r$-intersection. The for-loop starting in line (5) constructs $(r+1)$-intersections by testing one base string at a time from $\text{compBase}$. To perform the intersection step in line (7), $\text{intersect(\text{inter})}$ computes the intersection of all elements in $\text{inter}$. However, we can store the intersection in previous step and just return it here. If this new intersection is equivalent to $q$ itself, the coefficient $C_q$ is incremented.

Let us illustrate how $\text{ComputeCoefficient}$ works. Assume that a partition $P$ with a node of “ababcd” in Figure 3.6(a) and $q$ is selected as “ababcd” at the beginning of the execution. Then, we get $\text{compBase} = \{“ab??cd”, “a??bcd”, “??abcd”\}$. At the first iteration of the while-loop in line (3), $\text{tmpIter}$ becomes $\{\{“ab??cd”, “a??bcd”\}, \{“ab??cd”, “??abcd”\}, \{“a??bcd”, “??abcd”\}\}$ after exiting the for-loop. Furthermore, the if-statement in line (7) finds that only “ab??cd” $\cap$ “??abcd” results in $q$. Thus, $C'_P$ becomes 1 and $C_P$ is -1. Since we set $r\text{Iter}$ to $\text{tmpIter}$ and $r\text{Iter}$ is not empty, we perform the next iteration of the while-loop. Now, there is only one intersection $\{“ab??cd”, “a??bcd”, “??abcd”\}$ that is the only element in $\text{tmpIter}$ and the result of the intersection is $q$. Thus, $C'_P$ becomes 1 which is added to $C_P$ by line (11) and $C_P$ becomes zero. The value of $C_P$ does not be changed later on and finally $C_P$ becomes zero.

One reason why this algorithm is slow is that we need to maintain every intersection generated even though the current $r$-intersection does not result in $q$. To alleviate this problem, we develop an approximation method later that avoids the maintenance of all $r$-intersections generated.

3.3.6 Procedure EstimateFreq

For the material presented so far, the discussion is mainly based on strings (e.g., nodes in string hierarchy). It is only in line (10) of $\text{PartitionEstimate}$ that requires the use of the N-gram table. The task is that for a given string $q$, the frequency that this string appears in the database $\text{DB}$ is returned. For an extended string $q$, if the extended $N$-gram table kept by the system contains an entry for $q$ (e.g., when $|q| \leq N$), then the frequency $|q|$ is immediately returned. Otherwise, $|q|$ needs to be estimated using estimation algorithm as MO [77]. The experimental evaluation section will compare the accuracy when $\text{EstimateFreq}$ implements MO, as well as other variants below.

- If $s$ is obtained from $t$ by replacing at least one character in $t$ with $\text{?}$, then by definition $|s| \geq |t|$ (e.g., $|“abc?”| \geq |“abcd”|)$. However, MO estimation may violate this condition, i.e., $\text{MO}(s)$ may be smaller than $\text{MO}(t)$. In that case, the MO estimate of $s$ is reset to the maximum MO estimate of all such $t$’s. We call this the MAX estimate.
3.4 Optimized Algorithm OptEQ

BasicEQ is efficient whenever the general formulas presented in Section 3 can be applied. For other situations, BasicEQ is sufficient for small problems. However, it is clear that it does not scale up with respect to query length \( l \) and \( \tau \) when the formulas cannot be used. The complexity of BasicEQ is exponential with respect to \( l \). In this section, we propose the optimized algorithm OptEQ which extends BasicEQ with two enhancements. We show experimentally in the next section how OptEQ can handle long queries efficiently.

3.4.1 Approximating Coefficients with a Replacement Semi-lattice

When the set \( \text{compBase} \) is large (e.g., query \( s_q \) is long or \( k \) is large), computing the exact value of \( C_q \) as in \( \text{ComputeCoefficient} \) is prohibitive. We next present a strategy to approximate \( C_q \) by avoiding the generation of all \( r \)-intersections.

• Among all the substrings \( q' \) of \( q \) that are kept in the \( N \)-gram table, let \( q'_{\text{min}} \) be the substring with the smallest frequency. By definition, \( |q| \leq |q'_{\text{min}}| \). In other words, this is an over-estimate of \( q \). It has been well documented that the MO-estimate of \( q \) is often an underestimate, particularly when \( q \) is long [26]. Thus, one estimate is to take the geometric mean of \( |q'_{\text{min}}| \) and the MO-estimate of \( q \). We call this MO+ estimate.

• We can combine the MAX estimate and the MO+ estimate to give another estimate referred to as MM. That is, it gives the geometric mean of \( |q'_{\text{min}}| \) and the MAX estimate of \( q \).

Section 6 will compare empirically the effectiveness of these instances of \( \text{EstimateFreq} \).
3.4. Optimized Algorithm OptEQ

Let us revisit the example of \( \text{Ans}(\text{"abcde"}, 2) \). As the computation iterates over the length of the answer strings, one iteration deals with the strings of length five contained in \( \text{Ans}(\text{"abcde"}, 1D1I) \cup \text{Ans}(\text{"abcde"}, 2R) \). The next iteration deals with those of length six in \( \text{Ans}(\text{"abcde"}, 1I1R) \). The string hierarchies in both cases are rather complex and thus the approximation strategy is applied.

For ease of presentation, we use \( \text{Ans}(\text{"abc"}, 1I1R) \) as an example to illustrate the strategy. To generate all the base strings, we can first consider the position of one insertion, followed by one replacement. Specifically, we can group the base strings into four and the 4 groups are as follows:

- (inserting into first position) “??bc”, “?a?c” and “?ab?”
- (inserting into second position) “??bc”, “a??c” and “a?b?”
- (inserting into third position) “b?c”, “a??c” and “ab??”
- (inserting into fourth position) “bc?”, “a?c?” and “ab??”

Note that the strings in DB belonging to “?abc” in the first group above is also included in other base strings. Thus, “?abc” is pruned in the first group. Similarly, “a?bc”, “ab?c” and “abc?” are removed from their group’s base strings.

With the base strings of \( \text{Ans}(\text{"abc"}, 1I1R) \), we can build the string hierarchy presented in Figure 3.9. In the string hierarchy, the three base strings in the first group are organized in a sub-semi-lattice rooted at “?abc”. Similarly, the three base strings in the second group form a sub-semi-lattice rooted at “a?bc”. Note that “??bc” is shared between two groups and thus these two groups correspond to a general situation when two semi-lattices overlap on some base strings. To compute the size of \( \text{Ans}(\text{"abc"}, 1I1R) \), eventually the coefficient of every node in the string hierarchy in Figure 3.9 is needed.

Assume that we want to compute the coefficient of “aabc” in Figure 3.9. Instead of applying ComputeCoefficient directly on “aabc”, the approximation strategy completes the semi-lattice of “aabc” with two replacements. This completed replacement semi-lattice is shown in Figure 3.10 where the two sub-semi-lattices rooted at “?abc” and “a?bc” are shown in solid lines, while the other parts, which do not appear in the string hierarchy of Figure 3.9 and thus are not required for \( \text{Ans}(\text{"abc"}, 1I1R) \), are shown in dotted lines. The beauty of the replacement semi-lattice rooted at “aabc” is that Eqn. (3.3) immediately gives a formula for \( C_{aabc} \), if the entire replacement semi-lattice appears (i.e. are involved) in the sub-semi-lattice rooted at “aabc” in Figure 3.9. Specifically, recall from the discussion in Section 3.2.3 that the root of the semi-lattice is the level-0 node. Thus, the relevant coefficient is the one for \( F_0 \), which is:

\[
C_{\text{root}} = \sum_{r=2}^{t} (-1)^{r+1} D_{\ell,k,r}.
\]

When parts of the replacement semi-lattice are not involved, the approximation strategy assumes that the number of \( r \)-intersections leading to a root node is proportional to the number of its base strings that are involved. This assumption arises from the highly uniform structure of a replacement semi-lattice. Thus, each term in the summation is scaled relative to the
3.4. Optimized Algorithm OptEQ

<table>
<thead>
<tr>
<th></th>
<th>2-inter.</th>
<th>3-inter.</th>
<th>4-inter.</th>
<th>5-inter.</th>
<th>6-inter.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_r$ for</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bottom node</td>
<td>-3</td>
<td>16</td>
<td>-15</td>
<td>6</td>
<td>-1</td>
</tr>
<tr>
<td>estimated $C_r$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>for “aabc”</td>
<td>$\left(\frac{5}{2}\right) / \left(\frac{6}{2}\right) \ast$</td>
<td>$\left(\frac{5}{3}\right) / \left(\frac{6}{3}\right) \ast$</td>
<td>$\left(\frac{5}{4}\right) / \left(\frac{6}{4}\right) \ast$</td>
<td>$\left(\frac{5}{5}\right) / \left(\frac{6}{5}\right) \ast$</td>
<td>$\left(\frac{5}{6}\right) / \left(\frac{6}{6}\right) \ast$</td>
</tr>
<tr>
<td>$\text{exact } C_r$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>for “aabc”</td>
<td>-2</td>
<td>8</td>
<td>-5</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 3.11: Approximating Coefficients for “aabc”

number of the base strings involved. Specifically, let $B$ the size of $\text{compBase}$, which is the set of base strings that are involved. Then, $C_{\text{root}}$ is estimated by:

$$C'_{\text{root}} = \sum_{r=2}^{\ell} (-1)^{r+1} D_{\ell,k,r} \ast \left(\frac{|B|}{r}\right) \left(\frac{\ell}{r}\right)$$

(3.4)

That is, in a completed replacement semi-lattice, there are $\left(\frac{\ell}{k}\right)$ base strings. But as there are only $B$ base strings in the completed replacement semi-lattice that are involved, every term is scaled proportionally.

Let us return to the example of “aabc”. Recall from Figure 3.10 that there are five base strings being involved for the query of $\text{Ans}("aabc", 1I1R)$, as supposed to six (i.e., “aa??” being the only base string not involved) in the completed replacement semi-lattice. Thus, in this example, $|B| = 5$ and $\left(\frac{\ell}{k}\right) = \left(\frac{5}{2}\right) = 6$. The table shown in Figure 3.11 applies the approximation strategy to $C_{aabc}$. The second row of the table is identical (modulo the sign) to the third row of the table in Figure 3.2, as it gives the coefficients $C_r$ for the root node. The third row applies the proportionality factor to each term. The last row of the table shows the true coefficients, which turn out to be identical to the approximated ones! In general this is not always true. For example, the division in Eqn. (3.4) does not always give an integer value.

3.4.2 Fast Intersection Tests by Grouping

Another way to optimize $\text{BasicEQ}$ is to optimize the for-loop in line (6) of Figure 3.5. Specifically, for every possible pair of a base string $b \in \text{baseNode}$ and $u \in \text{newNodes}$, we perform the intersection. However, as seen in the previous example shown in Figure 3.10, base strings may naturally group into a collection of semi-lattices. In the case of approximating by completion of a replacement semi-lattice, the individual semi-lattices are all parts of a larger, completed semi-lattice. In a more general setting, this condition may not hold for all the semi-lattices. Nevertheless, some of the semi-lattices may overlap. Below we explore further the notion of grouping and show how group-wise operations can be exploited.

Note that a replacement operation does not cause any character shifting in the result, while an insertion or a deletion does. Thus, we organize all the base strings for a query of the
3.4. Optimized Algorithm OptEQ

**Input:** query $s_q$ of length $\ell$, edit distance threshold $k$
length $\text{len}$ of answer strings

**Output:** estimated frequency $c$

1: Identical to procedure BasicEQ except using grouping in line 6 and invoking $\text{ComputeCoefficient}(q)$ by PartitionEstimate:
2: if $q$ is the root of a replacement semi-lattice then
3: apply the formula in Eqn. (3.3)
4: else if $\text{compBase}$ of $q$ exceeds a certain size then
5: apply the approximation strategy based on Eqn. (3.4)
6: else if $\text{ComputeCoefficient}$ is to be applied then
7: use grouping to speed up and computation of coefficients as discussed in Section 5.2
8: end if

Figure 3.12: A Skeleton of Procedure OptEQ

form $\text{Ans}(s_q, jIjD1R)$ by considering the position of insertion and deletions only.

To illustrate the power of grouping, let us consider the example of $\text{Ans}(“abcd”, 1D1I1R)$, as part of a query with the edit distance $k = 3$. We group all the base strings for $\text{Ans}(“abcd”, 1D1I1R)$ by first considering the deletion and insertion. The following array shows all the possible combinations for one deletion and one insertion in any order.

$$G = \begin{pmatrix}
  “?abc” & “a?bc” & “ab?c” & “abc?” \\
  “?abd” & “a?bd” & “ab?d” & “abd?” \\
  “?acd” & “a?cd” & “ac?d” & “acd?” \\
  “?bcd” & “b?cd” & “bc?d” & “bcd?”
\end{pmatrix}$$

We can view the first row as deleting the character $d$ and inserting a character at various positions. Similarly, we can view the first column as inserting a character at the first position and deleting a character at various positions. Then the replacement can be applied to each of the 16 elements of matrix $G$. For instance, applying one replacement to “?abc” gives “??be”, “?a?c” and “?ab?”. Note that these three strings form a replacement semi-lattice rooted at “?abc”. Indeed, the same phenomenon applies to every element of $G$. In other words, the base strings of $\text{Ans}(“abcd”, 1D1I1R)$ are split into sixteen groups, or more specifically, semi-lattices with the roots being the elements of $G$.

The first benefit of grouping in this manner can be exemplified by noting that $G(1,1) = “?abc”$ and $G(4,1) = “?bcd”$ generate empty intersection. More importantly, any one of the base strings within the group of “?abc” (namely, “??be”, “?a?c” and “?ab?”) and any one of the base strings within the group of “?bcd” (namely, “??cd”, “?b?d?” and “?bc?”) always produce an empty intersection. This is because regardless of which pair, there is at least one position where there are no wildcards and the characters do not match. This motivates the following proposition.

**Proposition 5.** Let $g_1, g_2$ be two elements in matrix $G$. Let $\text{grpDist}(g_1, g_2)$ be defined as the number of non-matching positions between $g_1, g_2$. Let $s_1$ be any node within the $r_1$-replacement
3.5. Empirical Evaluation

semi-lattice rooted at \( g_1 \), where \( r_1 \) is defined by the query. Similarly, let \( s_2 \) be any node within the \( r_2 \)-replacement semi-lattice rooted at \( g_2 \). Then \( \text{grpDist}(g_1, g_2) > r_1 + r_2 \) implies \( s_1 \cap s_2 = \emptyset \).

For the running example of \( \text{Ans}("abcd", 1D1I1R) \), if \( g_1, g_2 \) are "?abc" and "?bcd" respectively, then \( r_1 = r_2 = 1 \) because of the 1R specification in the query. However, the group distance \( \text{grpDist}(g_1, g_2) = 3 \) because there are 3 non-matching positions. The proposition then allows us to immediately conclude that for any pair of \( s_1 \) from the 1-replacement semi-lattice rooted at \( g_1 \) and \( s_2 \) from the 1-replacement semi-lattice rooted at \( g_2 \), their intersection must be empty. Thus, grouping and group distance defined via the roots of two groups provide a fast negative test for groups of base strings. This speeds up line (6) of \( \text{BasicEQ} \).

Another benefit of grouping is that each replacement semi-lattice allows the coefficient to be computed directly by the formula given in Eqn. (3.3), thus avoiding the use of \( \text{ComputeCoefficient} \). Specifically, the coefficient of the root of a group is precisely \( C_{\text{root}} = \sum_{r=2}^{(1)} (-1)^{r+1} D_{\ell,k,r} \) and the coefficient of an intermediate level-\( i \) node in the group is \( C_q = \sum_{r=2}^{(i)} (-1)^{r+1} D_{\ell-i,k-i,r} \), which is the coefficient of \( F_i \) in Eqn. (3.3). However, we have to be careful in defining \( \ell \) and \( k \) before the above formulas can be used. In the "?abc" group for our example, every node in the replacement semi-lattice has a wildcard at (at least) the first position. This position should not be considered in the replacement semi-lattice. Effective string length is defined as the length of the root after excluding any wildcard in the root which is shared in the group. Similarly, effective number of wildcards is the number of wildcards excluding the wildcards common to all within the group. In applying the formulas above, \( \ell \) and \( k \) are set to the effective string length and effective number of wildcards. For example, in Figure 3.9, the sub-semi-lattice rooted at "?abc" has 3 base strings. Its effective string length is 3 instead of 4 and its effective number wildcards is 1 instead of 2. Thus, we compute the coefficient of "?abc" using Eqn. (3.3) by setting \( \ell = 3 \) and \( k = 1 \).

Procedure \( \text{OptEQ} \) is an optimized version of \( \text{BasicEQ} \) that incorporates approximation and grouping into the size estimation. The skeleton shown in Figure 3.12 highlights the key differences. \( \text{OptEQ} \) is scalable to deal with size estimation of queries like \( \text{Ans}("abcd", 1D1I1R) \) and more complex combinations. In general, as query \( s_q \) becomes longer, the \( G \) matrix shown earlier becomes larger. However, there are more and more groups with large group distance and fast negative tests provide scalability. Similarly, for the general situation of \( \text{Ans}(s_q, iDjImR) \), if \( (i+j) \) is large, again the \( G \) matrix becomes larger. Yet, like before, grouping helps significantly. On the other hand, if \( m \) is large (while \( k \) remains constant), the replacement semi-lattice becomes dominant. In that case, \( \text{OptEQ} \) either applies Eqn. (3.3) if possible, or Eqn. (3.4) if approximation is needed.

3.5 Empirical Evaluation

3.5.1 Implementation Highlights

The proposed algorithms \( \text{BasicEQ} \) and \( \text{OptEQ} \) were developed using Java 1.5. These algorithms were applied to different settings of \( N \)-gram tables parameterized by the triplet \((N_B, N_E, PT)\):
3.5. Empirical Evaluation

Figure 3.13: Actress Last Name, $\tau = [1, 3]$

- all q-grams without wildcards for $|q| \leq N_B$ are kept;
- all q-grams with wildcards for $|q| \leq N_E$ are kept; but
- all the entries kept must have a count $> PT$.

OptEQ(method, $N_B, N_E, PT$) denotes the variants when the method used for EstimateFreq was one of: MO, MO+, MAX and MM, and using the $N_B$-gram table and the extended $N_E$-gram table with prune threshold $PT$. For instance, OptEQ(MM,9,6,1) gives the version of OptEQ using MM for frequency estimation on the 9-gram table and the extended 6-gram table, where only entries of count $\geq 2$ are kept. Below we will evaluate the trade-offs among $N_B$, $N_E$ and $PT$, with respect to accuracy and memory size.

In our implementation, the (extended) q-gram tables are optimized in size in two ways. First, in hash buckets of tables, the string corresponding to a q-gram entry is hashed again and stored as a byte key not as a full string key. Thus, there may be collisions introduced by hashing, representing a trade-off between accuracy and size. The results reported below include the errors arisen due to collisions. Second, for each entry, the count is initially restricted to a short integer. When the count exceeds the range of a short integer, the count is maintained separately in an overflow table.

The discussion so far on the computation of coefficients shows how they can be computed - for once. In our implementation, these coefficients are pre-computed and stored in a coefficient table. This is possible because the coefficients depend only on the length and the $\tau$ value of the query. In “query time”, once the actual frequencies of the required nodes are determined, the selectivity can be estimated very efficiently using the pre-computed coefficients. Note that this table remains unchanged from one data set to another.

3.5.2 Experimental Setup

We conducted a series of experiments using several data sets. The results shown here are based on three benchmarks.

- (Short strings) The Actresses last name data set consists 392,132 last names of actresses downloaded from www.imdb.com. The maximum and average length are 16 and 6.3.
• (Medium-length strings) The DBLP authors data set consists of 699,198 authors’ full names from DBLP [93]. The maximum and average length are 43 and 14.3.

• (Long strings) The DBLP titles data set consists of 53,365 titles collected from DBLP. The maximum and average length are 40 and 29.6. Longer titles are excluded. For longer strings, a substring query which will be discussed in Chapter 4 may fit better.

SEPIA was downloaded from the FLAMINGO project homepage [138] and was written in C++. We tuned SEPIA to maximize its performance. The results reported here are based on applying the error correction step included in the software, and on using 2,000 clusters, a sampling ratio of 5% and the CLOSE RAND method to populate the PPD table.

Accuracy is measured by average relative error, defined as \(|est - true|/true\), where \(est\) and \(true\) are the estimated and true frequencies respectively. To avoid very low frequency queries from skewing the average relative error figure, we exclude queries whose true frequencies are below 3. We also exclude best and worst 3 queries from the analysis. The experiments were run on an Intel P4 3GHz desktop PC with 1 GB memory and 40GB disk space running GNU/Linux with kernel 2.6. For the memory size figures cited below, the figure for SEPIA is based on the file size written on disk which includes the PPD table and the frequency table. For OptEQ, the size figure is based on multiplying the number of entries in the q-gram and extended q-gram tables with the number of bytes per entry.

### 3.5.3 Actresses Last Names

Figure 3.13 compares the average relative error between SEPIA and variants of OptEQ for the Actresses data set. A total of 300 queries are randomly selected, of which 272 have true frequencies exceeding 3. The average relative error is obtained based on the 272 queries. The edit distance threshold \(\tau\) is selected among 1, 2 and 3 with equal likelihood. The first column shows that the average relative error for SEPIA is around 40\%. The next four columns show the version of OptEQ when the different frequency estimation methods are used, \(N_B = N_E = 5\), and \(PT = 0\). Compared with MO and MO+, MAX and MM were consistently superior. In the experiments to follow, we only show the results of MM.

When OptEQ(MAX, 6, 6, 5) or OptEQ(MM, 6, 6, 5) are used, the last two columns show that the average relative errors are reduced in half from 40\% to about 20\%. This shows
the effectiveness of increasing $N_B$ and $N_E$ in reducing relative error. The reason for setting the prune threshold of $PT$ to 5 is to keep the memory utilization of $OptEQ$ to be more or less the same as that used by $SEPIA$ (3.3MB vs 3.6MB). Thus, using less memory, $OptEQ(MM, 6, 6, 5)$ reduced the average relative error by half.

In terms of running time, the “build” time to construct the extended N-gram tables took about 5 minutes. The average query processing time for $OptEQ$ was about 13 msec using the pre-computed coefficient table (or 1.2 seconds if $OptEQ$ were to compute every coefficient on-the-fly). The build time of $SEPIA$ to construct the clusters and the histograms took around 60 minutes. The query processing time for $SEPIA$ was about 8 msec. We do not include the results for $BasicEQ$ because the execution time was very slow when the query exceeded a length of 10. For instance, for a query of length 11, $BasicEQ$ took 74 seconds.

If we only keep simple q-grams, not extended q-grams, the viable approach would be to enumerate all possible strings within the threshold and sum up their frequencies. We do not present detailed results, but the performance was unacceptable. For example, even for the simple query $s_q = “blank”$, there are more than 4 million possible strings within edit distance threshold of 3. It took more than 20 seconds just to estimate the frequency of each string and sum them up. It took only 7 msec for $OptEQ$ to process the same query which is three orders of magnitude faster. This clearly shows the utility of the proposed extended q-grams.

### 3.5.4 DBLP Authors

This set of experiments used queries of an average length more than double than those used in the Actresses data set. Out of the 900 randomly selected queries, 674 have true frequencies exceeding 3, and are more or less equally distributed among $\tau = 1, 2, 3$. Figure 3.14 gives the average relative errors, with $\tau$ separated into 1, 2 and 3 respectively. $SEPIA$ consistently hovers around 70% in relative error. The average relative error given by $OptEQ(MM, 12, 5, 2)$ increases from around 15% for $\tau = 1$ to about 30% for $\tau = 3$. The memory used by $OptEQ(MM, 12, 5, 2)$ and $SEPIA$ were 13.7MB and 14MB respectively. Again, with similar or less memory, $OptEQ$ delivered significantly more accurate estimations. Furthermore, we reduce the memory consumption by using $OptEQ(MM, 9, 5, 2)$, which occu-
3.5. Empirical Evaluation

Figure 3.16: DBLP Titles, $\tau$ in [1, 3]

Compared with the other two data sets, the DBLP titles data set contains the longest strings. Out of the 600 randomly generated queries, only 31 have true frequencies exceeding 3. Among them 16, 8 and 7 have $\tau = 1, 2, 3$ respectively. In particular, for $\tau = 3$, 5 out of 7 queries have a length 15 or higher. To reduce the processing time for OptEQ, we employed sampling and limited the maximum number of groups and base strings in each group to 15 and 10 respectively for any query of length $\geq 10$ and $\tau = 3$. Figure 3.16 gives the average relative errors. The average relative error of SEPIA is about 27%, while occupying 13MB of memory. In contrast, OptEQ($MM, 12, 5, 1$), occupying a space of 4.3MB, gives an average relative error of about 20%. If more space is allowed, OptEQ($MM, 15, 5, 1$) with a space of 5.4MB, gives an average relative error of 12%. The average query time is 76 msec.

Although our main focus lies in low edit distance thresholds (i.e. $\tau = 1, 2, 3$), we briefly present results on higher thresholds, $\tau = 4$ and 5. To handle higher thresholds, sampling approach is essential. We randomly sample 10 base strings for each string hierarchy. We exclude too short sampled queries (length $\leq 5$) as they are meaningless in the increased thresholds. The average relative error given by OptEQ($MM, 12, 5, 5$) is around 60%. The error is increased significantly as all the base strings are not considered due to sampling. However, it is still better than the average relative error of SEPIA which was 150%. It occupied 1.5MB and the running time was 10 msec. SEPIA occupied 10MB and it took 30 msec to estimate.

The maximum edit distance threshold we support is limited by the maximum number of wildcard in a q-gram, which in turn is bounded by $N_E$. Based on the discussion in section 3.5.4

pies only 9.2MB. Yet the average relative error is still significantly lower than that of SEPIA. Note that we increased $N_B$ not $N_E$ from the Actresses data set. $N_E = 5$ or 6 generally give good accuracy with reasonable space. However, increasing $N_E$ beyond 6 is not recommended as it incurs combinatorial increase in the space. $N_B$ offers a tunable alternative, and we used higher $N_B$ for the authors data set as strings are generally longer.

While a single value of average relative error could be misleading, Figure 3.15 shows the distribution of errors with respect to $\tau$. The x-axis divides the error range into ranges of 25% width, e.g., ranges [-100%, -75%), [-75%, -50%), etc. The y-axis shows the numbers of queries within the error ranges. For $\tau = 1, 2, 3$, OptEQ($MM, 9, 5, 2$) gives a more balanced distribution. In contrast, SEPIA suffers from rather serious under-estimation problem here.

3.5.5 DBLP Titles

Compared with the other two data sets, the DBLP titles data set contains the longest strings. Out of the 600 randomly generated queries, only 31 have true frequencies exceeding 3. Among them 16, 8 and 7 have $\tau = 1, 2, 3$ respectively. In particular, for $\tau = 3$, 5 out of 7 queries have a length 15 or higher. To reduce the processing time for OptEQ, we employed sampling and limited the maximum number of groups and base strings in each group to 15 and 10 respectively for any query of length $\geq 10$ and $\tau = 3$. Figure 3.16 gives the average relative errors. The average relative error of SEPIA is about 27%, while occupying 13MB of memory. In contrast, OptEQ($MM, 12, 5, 1$), occupying a space of 4.3MB, gives an average relative error of about 20%. If more space is allowed, OptEQ($MM, 15, 5, 1$) with a space of 5.4MB, gives an average relative error of 12%. The average query time is 76 msec.

Although our main focus lies in low edit distance thresholds (i.e. $\tau = 1, 2, 3$), we briefly present results on higher thresholds, $\tau = 4$ and 5. To handle higher thresholds, sampling approach is essential. We randomly sample 10 base strings for each string hierarchy. We exclude too short sampled queries (length $\leq 5$) as they are meaningless in the increased thresholds. The average relative error given by OptEQ($MM, 12, 5, 5$) is around 60%. The error is increased significantly as all the base strings are not considered due to sampling. However, it is still better than the average relative error of SEPIA which was 150%. It occupied 1.5MB and the running time was 10 msec. SEPIA occupied 10MB and it took 30 msec to estimate.

The maximum edit distance threshold we support is limited by the maximum number of wildcard in a q-gram, which in turn is bounded by $N_E$. Based on the discussion in section 3.5.4
3.6 Conclusions

regarding $N_E$, we do not recommend $OptEQ$ for the thresholds of 6 or higher.

3.5.6 Space vs Accuracy

The three parameters $N_B$, $N_E$ and $PT$ provide a very tunable setting. As shown in Figure 3.16, an increase in $N_B$ leads to increase in space consumed but a decrease in relative error. The same can be said for $N_E$ as shown in Figure 3.13. However, incrementing $N_E$ by 1 requires a lot more space than incrementing $N_B$ by 1. In our experimentation, we found that there is a significant reduction in error when $N_E$ is raised from 4 to 5, whereas the reduction in error becomes much smaller when $N_E$ exceeds 7.

When a large $N_E$ is used, one way to keep memory consumption in check is to apply a larger prune threshold $PT$. In general, an increase in $PT$ leads to a reduction in space but an increase in relative error. Consider $OptEQ(MM, 15, 5, 1)$ in Figure 3.16. The following table shows the impact of $PT$ on accuracy and size.

<table>
<thead>
<tr>
<th></th>
<th>avg. rel. error</th>
<th>size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$OptEQ(MM, 15, 5, 0)$</td>
<td>11.8%</td>
<td>19.4</td>
</tr>
<tr>
<td>$OptEQ(MM, 15, 5, 1)$</td>
<td>12.0%</td>
<td>5.4</td>
</tr>
<tr>
<td>$OptEQ(MM, 15, 5, 4)$</td>
<td>14.7%</td>
<td>1.9</td>
</tr>
</tbody>
</table>

As compared with $SEPIA$ which uses 13MB of memory and gives an average relative error of 27%, the last two variants shown in the above table are better alternatives.

3.5.7 Summary of Experiments

We evaluated the proposed algorithm, OptEQ, using three real-world data sets: IMDB actresses’ last names, DBLP author names, and DBLP paper titles. The performance improvement of OptEQ over BasicEQ clearly showed that optimization techniques in Section 3.4 are very effective. In all the three data sets, OptEQ consistently showed relative errors less than 40% and did not exhibit over- or underestimation tendency. As an example, using only about 17% of the original data size, OptEQ’s relative error was only about 15% in the DBLP Titles data set. We compared OptEQ with the prior art, SEPIA, and the experimental results showed that OptEQ delivered more accurate estimation with less space overhead.

3.6 Conclusions

In this chapter, we develop the algorithm $OptEQ$ for estimating selectivity of approximate string matching with edit distance. The proposed solution aims for low edit distance like 1, 2 or 3. The low edit distance may not cover every single database applications, but we believe that a good solution for selectivity estimation for $\tau = 1, 2, 3$ already has a lot to offer to many database applications, given the SIS observation.

We propose extended q-grams with wildcard to efficient count variants of strings. We show that possible variants of a query string with an edit distance threshold can be modeled with a lattice and extended q-grams. We develop techniques for efficient counting of the variants. The proposed techniques serve as bases for Chapter 4 and Chapter 5. The main algorithm,
3.6. Conclusions

OptEQ, further exploits several observations in the lattice for efficiency. For all the three benchmarks, OptEQ delivers more accurate selectivity estimations than SEPIA. OptEQ is capable of exploiting available disk space to give higher precision.

In ongoing work, we explore further the relationship between disk space utilization and estimation accuracy given by OptEQ. We also plan to extend the current framework to deal with higher edit distance threshold. While this extension may not be too essential for database applications, the extension is essential for applications such as DNA sequence matching.
Chapter 4

Substring Selectivity Estimation

In this chapter, we consider selectivity estimation of selection queries with 'substring matching' semantics in Section 1.4.2, which is the second problem on selection operators. Intuitively, we aim to estimate the number of strings in the database that contain a similar substring to the query string. Suppose that a user wants paper abstracts that mention 'Przymusinski' in a publication database. With approximate substring matching, she can find abstracts like ‘The stable model semantics for disjunctive logic programs ... proposed by Przymusinski ...’ even if she does not recall the exact spelling of the name. It is substring matching because a database string can contain the query string.

We propose two solutions based on the EQ-gram table presented in the previous chapter. We first propose a simple solution using an EQ-gram table. We then extend it with Min-Hash signatures for more accurate estimation. The proposed techniques in this chapter are based on our EDBT 2009 paper [90].

4.1 Introduction

In this chapter, we study the SUBSTR problem in Definition 2. In the SUBSTR problem, a query string, \( s_q \), and a similarity threshold, \( \tau \), select strings in the database that contain similar substrings to \( s_q \); the goal is to estimate the number of selected strings. As in the previous chapter, we first consider edit distance to measure similarity. We then extend the solution to other similarity measures in later sections.

The matching semantics in the SUBSTR problem is 'substring matching.' That is, when we compare \( s_q \) and a string \( s \) in the database, \( s_q \) can match a substring in \( s \) and does not need to match entire string \( s \). In the STR problem, when the query string \( s_q = Michael \) and \( \tau = 1 \), \( s_q \) matches ‘Michal’ but does not match ‘Michal Tompton’. However, in the SUBSTR problem, \( s_q \) matches ‘Michal Tompton’ as well since it contains ‘Michal’ that is similar to \( s_q \) within 1 edit distance. See the following example.

Example 1. Consider a \( DB = \{ \text{‘kullback’}, \text{‘bach’}, \text{‘eisenbach’}, \text{‘bacchus’}, \text{‘baeza-yates’} \} \). Suppose the query \( s_q = ‘bach’ \) and \( \tau = 1 \). For substring selectivity, all 5 elements in \( DB \) except ‘baeza-yates’ satisfy the edit distance threshold, i.e., \( |Ans(‘bach’,1)| = 4 \). For full string selectivity, the corresponding answer \( Ans_f(‘bach’,1) \) is \{‘bach’\}.

The class of applications for substring selectivity estimation is broader than the class of applications for string selectivity estimation (e.g., “name like %pr%sinski%”). Note that substring matching is also a generalization of string matching. A standard technique for handling string matching with substring matching is to incorporate two special characters that are not in the alphabet to mark the start and end of a string, e.g., ‘#’ for start and ‘$’
for end. For instance, the string ‘walnut croissant’ is transformed into ‘#walnut croissant$’ and stored. When we want to perform string matching we only need to augment the start and end of a query with ‘#’ and ‘$', then those strings that match the query as a whole are considered as a match.

Several techniques [79, 89] were proposed to handle string selectivity estimation with edit distance. Direct application of those techniques to the substring problem is not an option since it will almost always under-estimate the true selectivity, which may change the ordering of predicates producing a bad query execution plan.

Furthermore, it is not trivial to adapt those techniques to give estimation of substring selectivity with edit distance. For the substring selectivity estimation problem, the biggest challenge is the estimation of intersection sizes among correlated substrings. Previous studies on string selectivity estimation do not need to consider this complication. To illustrate, let us return to $|Ansf('bach',1)|$. Estimation methods for string selectivity partition strings into groups (e.g., clusters [79], extended q-gram entries [89]). For $|Ansf('bach',1)|$, these groups include ‘back’, ‘bacc’, etc. Note that a string cannot simultaneously be the string ‘back’ and the string ‘bacc’. This non-overlapping condition greatly simplifies counting for string selectivity estimation. For substring matching, however, a string contain substrings like ‘kull’, ‘back’ and many more substrings simultaneously, which are not necessarily similar (e.g., ‘Solomon Kullback’). In other words, the substring condition gives rise to too many possible cases to consider imposing a major challenge. To the best of our knowledge, this is the first study developing algorithms for approximate substring selectivity estimation. As a preview, we make the following contributions.

- The first algorithm we propose is called MOF, which is based on the Most Frequent minimal base substring. Substring frequencies are estimated from extended q-grams.
- Recall that a key challenge for substring selectivity estimation is to estimate the overlaps among groups of substrings. MOF sidesteps this challenge by basing its estimation on a single substring. It is expected that the estimation may be improved by basing the estimation on multiple substrings. We propose in Section 4.4 an estimation algorithm called LBS, which uses signatures generated by set hashing techniques. However, standard set hashing is not sufficient. We extend set hashing in two ways. The first one is the approximation of signatures for set intersections. The second one is the use of multiple minimum values to improve accuracy. Depending on the amount of available space, LBS can be tuned to use larger signatures for improved estimation quality.
- The proposed algorithms can be extended to string similarity measures other than the edit distance. We show in Section 4.5 how to extend the algorithms to deal with SQL LIKE operator and Jaccard similarity.
- We show in Section 4.6 a comprehensive set of experiments. We compare MOF and LBS with two baseline methods. One is based on random sampling and the other one is a generalization of SEPIA designed for strings [79]. We explore the trade-offs between estimation accuracy, size of the intermediate data structure, and the CPU time taken for the estimation. Our results show that for fast runtime and low intermediate
size, MOF is an attractive light-weight algorithm. However, if more space is available for the signatures, so that the overlaps can be more accurately estimated, LBS is the recommended choice.

4.2 Preliminaries

4.2.1 Extended Q-grams with Wildcards

We use the extended q-gram table which is presented in Section 3.2. We define additional notations for this chapter.

For edit distance, edit operations considered are deletion (D), insertion (I) and replacement (R). A query \( Q \) is a pair \( Q \equiv (s_q, \tau) \), where \( s_q \) is the query string and \( \tau \) is the edit distance threshold. For any substring \( b \) such that \( ed(b, s_q) \leq \tau \) and insertion/replacement modeled by '?' (e.g., 'nua', 'nub', etc. are modeled by 'mu?'), \( b \) is called a base substring of \( Q \). Base substrings represent possible forms of substrings satisfying the query. Then the set of tuple ids in \( DB \) that have a substring which can be converted to \( s_q \) with at most \( \tau \) edit operations is:

\[
\text{Ans}(s_q, \tau) = \bigcup_b G_b, \tag{4.1}
\]

for all base substrings \( b \), and \( G_b \) denotes the set of tuple ids of string \( s \) in \( DB \) containing \( b \) as a substring. \(^9\) For example, \( \text{Ans}('sylvie', 1) \) contains strings like 'sylvia carbonetto', or 'sylvestre', but not 'cecilia van den berg'.

To compute \( |\text{Ans}(s_q, \tau)| \), the size of the answer set, all possible base substrings \( b \) are enumerated. Specifically, when the length of \( s_q \) is \( l \), the base substrings vary in length from \( (l - \tau) \) to \( (l + \tau) \). For each possible length, we consider all \( iDjIkR \) combinations, where \( iDjIkR \) denotes \( i \) deletion, \( j \) insertion and \( k \) replacement operations, with \( i + j + k = \tau \) and \( i, j, k \geq 0 \). For example, the base substrings of \( Q \equiv (van, 1) \) can be partitioned into 3 groups of length of 2, 3 and 4. Each group is from 1D, 1R, or 1I edit operation respectively. The group of 1I consists of ?van, v?an, va?n and van?. The group of 1R consists of va?, v?n, and ?an. The desired answer \( |\text{Ans}(van, 1)| \) is \( |G_{va} \cup G_{an} \cup G_{vn} \cup \ldots \cup G_{?van} \cup \ldots| \).

The above description considers all possible base strings and combinations for illustration purposes. In practice, particularly for larger edit distance thresholds \( \tau \) and long query substrings, a sampling strategy can be applied. However, Equation (1) still requires the accurate estimation of the overlaps among the group \( G_b \)'s.

4.2.2 Min-Hashing

In this section, we introduce Min-Hashing \([17, 34]\) which is a basis of the Set Hashing technique in the following section and will be used in Chapter 5 as well. Intuitively, Min-Hashing enables

\(^9\)Because \( DB \) is a bag of strings, \( DB \) may contain duplicates of the same string \( s \). We assume that each of those duplicates has its own distinct tuple id. This treatment is consistent with the studies in \([26, 79, 89]\).
Min-Hashing has widely been used in similarity related applications [17, 30, 34, 37, 91] since it enables a succinct representation of a set and preserves resemblance. In practice, \( L \) does not need to be big for \( \hat{\gamma} \) to be a good estimate of \( \gamma \). In testing similarity among web
4.2 Preliminaries

A set \( A \) and \( B \):

\[
\begin{align*}
\text{Set } A \{11,13,15\} & \quad \text{sig}A \\
\pi_1: & \begin{bmatrix} 1 \end{bmatrix} \begin{bmatrix} 3 \end{bmatrix} \begin{bmatrix} 4 \end{bmatrix} \begin{bmatrix} 1 \end{bmatrix} \\
\pi_2: & \begin{bmatrix} 3 \end{bmatrix} \begin{bmatrix} 1 \end{bmatrix} \begin{bmatrix} 5 \end{bmatrix} \begin{bmatrix} 1 \end{bmatrix} \\
\pi_3: & \begin{bmatrix} 4 \end{bmatrix} \begin{bmatrix} 5 \end{bmatrix} \begin{bmatrix} 2 \end{bmatrix} \begin{bmatrix} 2 \end{bmatrix} \\
\pi_4: & \begin{bmatrix} 5 \end{bmatrix} \begin{bmatrix} 2 \end{bmatrix} \begin{bmatrix} 3 \end{bmatrix} \begin{bmatrix} 1 \end{bmatrix}
\end{align*}
\]

\( B \):

\[
\begin{align*}
\text{Set } B \{14, 15\} & \quad \text{sig}B \\
\pi_1: & \begin{bmatrix} 1 \end{bmatrix} \begin{bmatrix} 2 \end{bmatrix} \begin{bmatrix} 4 \end{bmatrix} \begin{bmatrix} 2 \end{bmatrix} \\
\pi_2: & \begin{bmatrix} 4 \end{bmatrix} \begin{bmatrix} 5 \end{bmatrix} \begin{bmatrix} 4 \end{bmatrix} \begin{bmatrix} 4 \end{bmatrix} \\
\pi_3: & \begin{bmatrix} 5 \end{bmatrix} \begin{bmatrix} 2 \end{bmatrix} \begin{bmatrix} 2 \end{bmatrix} \begin{bmatrix} 2 \end{bmatrix} \\
\pi_4: & \begin{bmatrix} 3 \end{bmatrix} \begin{bmatrix} 3 \end{bmatrix} \begin{bmatrix} 1 \end{bmatrix} \begin{bmatrix} 1 \end{bmatrix}
\end{align*}
\]

(c) Signature generation of set \( A \) and \( B \):

\[
\begin{align*}
\text{sig}A & = \begin{bmatrix} 1 \end{bmatrix} \begin{bmatrix} 1 \end{bmatrix} \begin{bmatrix} 2 \end{bmatrix} \begin{bmatrix} 1 \end{bmatrix} \\
\text{sig}B & = \begin{bmatrix} 2 \end{bmatrix} \begin{bmatrix} 4 \end{bmatrix} \begin{bmatrix} 1 \end{bmatrix} \begin{bmatrix} 1 \end{bmatrix}
\end{align*}
\]

(d) Resemblance computation:

\[
\pi_1, \pi_2, \pi_3, \pi_4
\]

Figure 4.1: Set Resemblance Example

documents [17], 100 samples were considered to be good enough. Chen et al. used 50 for \( L \) to process boolean predicates [30].

4.2.3 Set Hashing

Chen et al. generalized the above idea to estimate the size of boolean queries on sets including intersection, union and negation [30]. The size of the union of \( m \) sets \( |A_1 \cup \ldots \cup A_m| \) can be estimated as follows. Let \( A = A_1 \cup \ldots \cup A_m \) and \( B = A_j \) for any \( A_j \in \{A_1, \ldots, A_m\} \). By Equation (4.2),

\[
\gamma = \frac{|A \cap A_j|}{|A|}.
\]

But as \( A \cap A_j = A_j \) and \( A \cup A_j = A \), we get:

\[
|A| = |A_1 \cup \ldots \cup A_m| = \frac{|A_j|}{\gamma}. \tag{4.4}
\]

\( A_j \) in Equation (4.4) can be any \( A_j \in \{A_1, \ldots, A_m\} \), but the one whose size is the biggest generally gives the best performance [30]. To estimate \( \gamma \), the signature of union, \( \text{sig}A \), can be constructed with the individual signatures as follows:

\[
\text{sig}_{A}[i] = \min(\text{sig}_{A_1}[i], \ldots, \text{sig}_{A_m}[i]) \tag{4.5}
\]

Let us return to the example in Figure 4.1. Based on Figure 4.1(d) and Equation (4.5), the constructed signature \( \text{sig}_{A \cup B} \) is \([1,1,1,1]\). Now applying Equation (4.3) to estimate the resemblance between \( A \cup B \) and \( A \) gives \( \hat{\gamma} = 3/4 \) \([1,1,1,1] \) vs. \([1,1,2,1]\). Finally, by Equation (4.4), the size of \( |A \cup B| = \frac{|A|}{\hat{\gamma}} = \frac{3}{3/4} = 4 \). This turns out to be exact, as \( A \cup B = \{t_1, t_3, t_4, t_5\} \).

As will be shown later, applying the above set hashing equations is not sufficient for our task. We extend the above scheme in two key ways in Section 4.
4.3. Estimation without Signatures

4.3.1 MOF: The MOst Frequent Minimal Base String Method

The first method we propose, called MOF, is based on extended q-grams. Recall from Equation (4.1) and the corresponding discussion that \( \text{Ans}(Q) = \bigcup_b G_b \) for all the base substrings \( b \) of \( Q \). However, it is obvious that given two base substrings \( b_1 \) and \( b_2 \), it is \( G_{b_2} \subseteq G_{b_1} \) if \( b_1 \) is a substring of \( b_2 \) (e.g., \( b_1 = va, b_2 = va? \) in the earlier example). We define a base substring \( b_i \) as minimal if there is not any other base substring \( b_j \) with \( i \neq j \) that is substring of \( b_i \). Thus, Equation (4.1) can be simplified to:

\[
|\text{Ans}(s_q, \tau)| = |\bigcup_{b \in MB(s_q, \tau)} G_b| \quad (4.6)
\]

where \( MB(s_q, \tau) \) is the set of all the minimal base substrings of \( s_q \) within \( \tau \) edit operations. We simply use \( MB \) when \( s_q \) and \( \tau \) are implied in the context.

Example 2. Let us consider the example of \( Q \equiv (\text{boat}, 2) \). Possible base substring length varies from 2 to 6, which correspond to the situations with 2 deletions and 2 insertions respectively. The following table enumerates all the possibilities for illustration purposes.

Starting from the base substrings of length 2, we eliminate base substrings that contain another base substring. For instance, ‘bo’ is a substring of ‘bo?’ ‘boa?’ ‘bo?a’; and etc. After removing redundant base substrings, the set of remaining minimal base substrings is \( MB = \{ ‘bo’, ‘ba’, ‘bt’, ‘oa’, ‘ot’, ‘at’, ‘b?a’, ‘b?t’, ‘b?a’, ‘o?t’, ‘b??t’ \} \). Thus, \( |\text{Ans}(\text{boat}, 2)| = |G_{bo} \cup G_{ba} \cup G_{bt} \cup G_{oa} \cup G_{ot} \cup G_{at} \cup G_{b?a} \cup G_{b?t} \cup G_{b?a} \cup G_{o?t} \cup G_{b??t}| \).

Algorithm 1 shows an outline to find all the minimal base substrings. The first for loop from line (2) to (8) generates all possible base substrings. In the most general case when \( \ell \) and \( \tau \) are large, the loop may be computationally expensive. However, as motivated by the Short Identifying Substring (SIS) assumption in [26], \( \tau \leq 3 \) can find many database applications.
4.3. Estimation without Signatures

For $\tau \leq 3$, the following table enumerates all the combinations with deletions, insertions and replacements ($\ell$ being the length of the query substring). For example, for $\tau = 3$, there are only two possibilities to obtain a substring of length $(\ell - 1)$, namely either by $2D1I$ or by $1D2R$. Thus, even for $\tau = 3$, there are only 10 combinations to be considered with a complexity of $O(\ell^3)$.

<table>
<thead>
<tr>
<th>$\tau$</th>
<th>$\ell-3$</th>
<th>$\ell-2$</th>
<th>$\ell-1$</th>
<th>$\ell$</th>
<th>$\ell+1$</th>
<th>$\ell+2$</th>
<th>$\ell+3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1D</td>
<td>1R</td>
<td>1I</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2D</td>
<td>1D1R</td>
<td>2R, 1DII</td>
<td>1I1R</td>
<td>2I</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3D</td>
<td>2D1R</td>
<td>2D1II, 1D2R</td>
<td>3R, 1DII1R</td>
<td>1D2I, 2I1R</td>
<td>2I1R</td>
<td>3I</td>
</tr>
</tbody>
</table>

The for loops in the lines (9) to (14) find all the minimal base substrings. The loop exploits the simple fact that a string cannot be a substring of something shorter. Finally, the set $MB$ of all the minimal base substrings is returned. For situations when $\ell$ and $\tau$ are larger, it is too expensive to fully implement line (6), and a sampling strategy is applied. The effectiveness of the sampling is evaluated in Section 5.

With the set $MB$ computed, the MOF (“MOst Frequent”) estimation algorithm uses the following heuristic based on the most frequent minimal base substring $b_{\text{max}}$ among all the base substrings in $MB(Q)$:

$$MOF(Q) = \frac{|G_{b_{\text{max}}}|}{\rho}$$

(4.7)

The parameter $\rho$ is called coverage. This simple heuristic is based on a generalization of the SIS assumption stating that: “a query string $s$ usually has a ‘short’ substring $s'$ such that if an attribute value contains $s'$, then the attribute value almost always contains $s$ as well” [26]. Our generalization states that a query substring $s$ usually has an identifying minimal base substring $s'$.

The requirement of almost always in the SIS assumption is more realistically relaxed to a fraction $\rho$ in our case; that is, a fraction $\rho$ of the strings in the $\text{Ans}(s_q, \tau)$ also contain the most frequent minimal base substring. As an example, $|\text{Ans}(\text{‘sylvia’, 1})|$ is 538 in the DBLP author names data set in Section 4.6, and typical variations of the query ‘sylvia’ are ‘silvia’ and ‘sylvia’ occurring 202 and 100 times respectively. So the base string ‘s?lvia’ alone explains more than half of the answer set size. The validity of our assumption is extensively evaluated in Section 4.6 by MOF.

In MOF, as shown in Equation (4.7), a single default value $\rho$ is used for simplicity. This default value can be obtained by sampling on the data set, which can be easily piggybacked when the extended N-gram table is being constructed.

For a base substring $b$, if the extended N-gram table kept by the system contains an entry for $b$ (e.g., when $|b| \leq N$), then the frequency $|G_b|$ is immediately returned. Otherwise, $|G_b|$ needs to be estimated using a substring selectivity estimation algorithm like MO [77].

$^{10}$Apart from the SIS generalization, the key difference between MOF and the CRT framework in [26] is that the former deals with edit distance.
4.3. Estimation without Signatures

Algorithm 1 MinimalBaseSubstrings

Input: query string $s_q$, edit distance threshold $\tau$

Output: the set $MB$ of minimal base substrings

1: $C = \phi$, $MB = \phi$
2: for all $\ell$ from $(\text{length}(s_q) - \tau)$ to $(\text{length}(s_q) + \tau)$ {
3:   Find all $(i, j, k)$ s.t. $i + j + k = \tau$ and $\text{length}(s_q) - i + j = \ell$
4:   for all $c = (i, j, k)$ found in the above step {
5:     $C = C \cup \{c\}$
6:     $B_c = \text{the set of all base substrings with } iDjIkR \text{ operations}$
7:   } // for all
8: } // for all
9: for all $c = (i, j, k) \in C$, $c' = (i', j', k') \in C$ s.t. $j' - i' > j - i$ {
10:   if $b$ is a substring of $b'$ {
11:     $B_{c'} = B_{c'} - \{b'\}$
12:   } // for all
13: } // for all
14: for all $c \in C$
15: $MB = MB \cup B_c$
16: return $MB$

4.3.2 Algorithms not Based on Extended Q-grams

S-SEPIA: a Method based on Clustering

We adapt SEPIA [79], which is proposed for the STR problem, to the SUBSTR problem by building clusters of substrings instead of strings. However, if we were to build clusters based on all the substrings contained in the database, it would be infeasible for large databases. Thus, we apply random sampling on substrings. Algorithm 2 shows a skeleton of S-SEPIA which is a simple adaptation of SEPIA to substrings. In the construction of clusters, the loop in lines (2) to (4) randomly extracts $c \times \ell$ substrings for each string where $\ell$ is the length of the string. Line (4) runs the normal SEPIA construction procedure to build the clusters and the corresponding histograms.

Another complication in adapting SEPIA to the substring problem arises from the counting semantics. As discussed in [77], there is the difference between presence counting and occurrence counting. If the tuple string is ‘Vancouver Van Rental’ and the query substring is ‘Van’, presence counting gives a value 1 to show that the substring is present in the tuple, whereas occurrence counting gives a value of 2 to indicate that the substring occurs twice in the tuple. For substring selectivity estimation, the intended semantics is presence counting, whereas the construction procedure of S-SEPIA essentially conducts occurrence counting.

To address this issue, we adapt the error-correction phase in SEPIA as follows. In lines (6) and (7), queries are randomly sampled to build a table giving a correction factor between presence and occurrence counting, according to query length, threshold, and frequency range. Then, during estimation time, line (2) of the Estimate procedure adjusts the estimated frequency based on this table.
4.4 Estimation with Set Hashing Signatures

**Algorithm 2 S-SEPIA**

**Procedure Construct**

**Input:** DB  
**Output:** Clusters $C$, Global histogram $PPD$,  
Error correction module $ECM$

1: $SDB = \emptyset$
2: for all tuple $t \in DB$ {
3: Generate $(c \cdot \text{length}(t))$ substrings from $t$; add to $SDB$
4: } // for all
5: Run SEPIA with $SDB$ to construct $C$ and $PPD$
6: Sample substring queries $Q$
7: Produce the error correction info $ECM$ from $Q$ using $freq_{est}$ from $SDB$ and $freq_{true}$ from $DB$

**Procedure Estimate**

**Input:** Query string $s_q$, Edit distance threshold $\tau$

**Output:** $|Ans(s_q, \tau)|$

1: Estimate $freq_{est}$ with $s_q$ and $\tau$ using $C$ and $PPD$
2: Calculate $freq_{corrected}$ with $s_q$, $\tau$ and $freq_{est}$ using $ECM$
3: return $freq_{corrected}$

**RS: a Method with No Space Overhead**

Note that MOF requires space overhead in the form of the extended N-gram table. S-SEPIA also incurs space overhead in the form of the histograms and other auxiliary information associated with the clusters. The algorithm called RS, which stands for “Random Sampling”, represents another extreme. It relies on random sampling from $DB$ at query time. As such, it does not create any intermediate “compile-time” data structure and incurs no space overhead. Specifically, given a query substring $s_q$, all it does is to randomly sample a fixed number of strings $s$ from $DB$ and checks the percentage of them such that $ed(s_q, b) \leq \tau$, where $b$ is a substring of $s$. It is conceivable that RS may incur significant query time computation, particularly for larger edit distance threshold $\tau$ and length $l$. In the experimental results section later, we examine the trade-offs between query time computation, space overhead and estimation accuracy.

4.4 Estimation with Set Hashing Signatures

4.4.1 LBS: Lower Bound Estimation

While MOF estimation is simple and efficient, a key weakness is that the estimation is based on a single minimal base substring which is the most frequent. In general, the estimation may be more accurate if many, if not all of, the minimal base substrings are used. However, if multiple minimal base substrings are used, we need to estimate the size of the union $|\bigcup_{b \in MB} G_b|$ in
4.4. Estimation with Set Hashing Signatures

Equation (4.6). This can be done by applying Equation (4.4) to obtain

\[
\text{LBS}(Q) = \left| \bigcup_{b \in MB(Q)} G_b \right| \approx \frac{|G_{b_{\text{max}}}|}{\gamma} \quad (4.8)
\]

where \(b_{\text{max}}\) is the most frequent minimal base string in \(MB(Q)\). Note that \(|G_{b_{\text{max}}}|\) is exactly the one used in the numerator of the MOF estimation in Equation (4.7). The only difference is that in MOF, \(\rho\) is a single default coverage computed by sampling for each data set. In contrast, the resemblance \(\gamma\) in the above equation is calculated by taking into account all the minimal base strings, making it more specific to the query. It is also possible that the resemblance \(\gamma\) is estimated to be zero for some extreme cases, in which case a default resemblance, like \(\rho\), is used instead since we cannot divide by zero in Equation (4.8).

To compute \(\text{LBS}(Q)\) with Equation (4.8), we need the values of \(|G_{b_{\text{max}}}|\) and \(\gamma\). As mentioned earlier, the former can be estimated using MO if it is not kept in the extended N-gram table. However, the difficulty here is to compute the approximation of \(\gamma\) by Equation (4.3) which requires the signatures to be compared for matching, i.e., we need \(\text{sig}_b(\equiv \text{sig}_G_b)\) for all \(b \in MB\) to compute the approximation of \(\gamma\). But there may be base substrings that are too long to be kept in the N-gram table. For instance, suppose that a base substring \(b \equiv \text{‘database’}\) and an extended N-gram table with \(N = 5\) is maintained.

Although Set Hashing technique [79] is able to estimate the size of intersection or union of strings when the query strings are not kept in the summary structure (PST or N-gram table), it is exponential in the number of strings whose frequency needs to be estimated. It is computationally manageable when the number of terms is expected to be small as in the boolean query problem, but the complexity will be unacceptable in the approximate substring problem where the number of possible forms of substring could be quite large.

Our solution is to rely on the substrings of \(b\) stored as extended q-grams in the N-gram table. Specifically, by letting \(b_1, \ldots, b_w\) be all the substrings of \(b\) of length \(N\) stored in

![Figure 4.2: An Illustration of LBS](image)
4.4. Estimation with Set Hashing Signatures

the N-gram table with $w = \ell - N + 1$, we can approximate the signature $\text{sig}_b$ based on the individual signatures $\text{sig}_{b_1}, \ldots, \text{sig}_{b_w}$. For instance, we calculate the signature $\text{sig}_{\text{database}}$ based on $\text{sig}_{\text{database}}, \text{sig}_{\text{abase}}, \text{sig}_{\text{tabas}}$ and $\text{sig}_{\text{abase}}$.

To illustrate our approach, let us consider an example with $w = 2$. Suppose that we have $b \equiv \text{'data'}, b_1 \equiv \text{'dat'}$ and $b_2 \equiv \text{'ata'}$, and the tuples containing ‘dat’ are exactly $\{t_1, t_3, t_5\}$ as captured by set $A$ in Figure 4.1. Similarly, suppose that the tuples containing ‘ata’ are exactly $\{t_4, t_5\}$ as modeled by set $B$ in Figure 4.1. We assume that the 4 random permutations $\pi_1, \pi_2, \pi_3, \pi_4$ in Figure 4.1 are still used. Now for the signatures, the permuted values of set $A = \{t_1, t_3, t_5\}$ under permutation $\pi_1$ are 1, 3, 4 as shown in Figure 4.1, which is presented as circles in the left diagram of Figure 4.2(a). However, because a signature only retains the minimum value for each permutation, only the value 1 is retained (shown as a solid circle, as opposed to the unfilled circles). The situation is similar for set $B$, as shown by the solid circle in the second column of the left diagram in Figure 4.2(a).

Let us consider how $\text{sig}_{A \cap B}$ can be estimated for $\pi_1$ using $\text{sig}_A$ and $\text{sig}_B$ only. Since each signature keeps the minimum value for each permutation, i.e. 1 for $\text{sig}_A$ and 2 for $\text{sig}_B$, it is not possible to figure out the exact minimum value for the permutation of the intersection $A \cap B$ using $\text{sig}_A$ and $\text{sig}_B$ only. Instead, we try to infer, as tightly as possible, a lower bound of the minimum value for the permutation of the intersection.

The value 1 is not possible for $\text{sig}_{A \cap B}$ because if this were the case, the value for $\text{sig}_B$ should be 1 as well, instead of 2. On the other hand, 2 is still a possible value for $\text{sig}_{A \cap B}$ because all we know from $\text{sig}_A$ is that the minimum value is 1. Thus, given just the two solid circles in the left diagram in Figure 4.2(a), the best inferred signature of $\text{sig}_{A \cap B}$ we can get under permutation $\pi_1$ is 2. Actually, $A \cap B$ has only $t_5$ which is mapped to 4. This is shown by the two unfilled circles in Figure 4.2(a), under the permutation $\pi_1$. It is easy to see that the inferred value 2 is a lower bound of the actual value.

The right diagram in Figure 4.2(a) shows the signatures for permutation $\pi_3$ in Figure 4.1. A similar reasoning determines that the value 2 is the inferred signature value of $\text{sig}_{A \cap B}$ under permutation $\pi_3$. This turns out to be the correct signature value because of the matched unfilled circle in the second column of the right diagram.

Figure 4.2(b) shows the inferred $\text{sig}_{A \cap B}$, [2, 4, 2, 1], for all the four permutations that are computed by selecting the maximum value of the corresponding $\text{sig}_A$ and $\text{sig}_B$ values for each permutation. The true signature $\text{sig}_{A \cap B} (= \text{sig}_{\{5\}})$ is actually [4, 5, 2, 1], as shown in Figure 4.1. As we mentioned previously, the inferred values are always lower bounds to the true values. This observation can be generalized to compute the intersection of $w$ signatures for any permutation $i, 1 \leq i \leq L$:

$$\hat{\text{sig}}_{b_1 \cap \cdots \cap b_w}[i] = \max(\text{sig}_{b_1}[i], \ldots, \text{sig}_{b_w}[i]) \quad (4.9)$$

To verify, suppose that there is a $\text{sig}_{b_k}[k]$ for $1 \leq k \leq L$ such that $\text{sig}_{b_k}[k] < \text{sig}_{b_{k'}}[k]$. Then the value $\text{sig}_{b_k}[k]$ cannot be the true signature value of $\text{sig}_{b_1 \cap \cdots \cap b_w}[k]$. Otherwise, the signature value $\text{sig}_{b_{k'}}[k]$ must be the same as $\text{sig}_{b_k}[k]$. By this argument, Equation (4.9) forms a lower bound estimation for $\text{sig}_{b_1 \cap \cdots \cap b_w}$.

Algorithm 3 sketches the outline of the LBS (“Lower Bound eStimation”) estimation algorithm. It first computes all the minimal base substrings. For each such substring, if it is
Algorithm 3 LBS Estimation

**Input:** query \(s_q\), edit distance threshold \(\tau\), maximum extended q-gram length \(N\), default resemblance \(\rho\)

**Output:** \(|\text{Ans}(s_q, \tau)|\)

1: \(MB = \text{MinimalBaseSubstrings}(s_q, \tau)\) (Algo. 1)
2: \(freq_{\text{max}} = 0, \ sig_{\text{max}} = \text{null}, \ sig_{\text{union}} = \text{null}\)
3: for all \(b \in MB\) 
   4: \(\text{if} \ \text{length}(b) > N\) 
      5: \(\text{Decompose} \ b \ \text{into a set of substrings} \ s_{b,i} \ \text{of length} \ N\)
      6: \(\text{Compute} \ sig_b \ \text{by applying Equation (4.9) with} \ sig_{s_{b,i}}\)
      7: \(\text{Calculate} \ freq_b \ \text{(i.e.} |G_b| \text{)} \ \text{using MO}\)
   8: }
9: \(\text{if} \ freq_b > freq_{\text{max}}\)
10: \(freq_{\text{max}} = freq_b, \ sig_{\text{max}} = sig_b\)
11: }
12: \(\text{sig}_{\text{union}} = \text{Union of} \ \text{sig}_{\text{union}} \ \text{and} \ sig_b \ \text{as in Eqn. (4.5)}\)
13: }
14: \(\text{Compute} \ \hat{\gamma} \ \text{with} \ sig_{\text{max}} \ \text{and} \ sig_{\text{union}} \ \text{as in Eqn. (4.3)}\)
15: \(\text{if} \ \hat{\gamma} = 0 \ \{ \ \hat{\gamma} = \rho \ \}\)
16: \text{return} \(freq_{\text{max}}/\hat{\gamma}\) as in Equation (4.8)

too long to have kept in the extended N-gram table, lines (4) to (8) estimate its frequency and signature. All these signatures are combined together in line (12) for the eventual computation of the resemblance \(\hat{\gamma}\) in line (14). Finally, in line (16), the estimate is returned.

### 4.4.2 Improving LBS with Extra Minima

Recall from Equation (4.9) that the estimated signature value is a lower bound to the true signature value. One possibility to improve the lower bound is to keep extra minimum permuted values such as the second minimum, the third minimum, etc. Certainly, each additional minimum value kept increases the size of the signature linearly. Below we describe the mechanics.

So far we have used the notation \(\text{sig}_A\) to denote the signature of set \(A\) under \(L\) permutations. To introduce additional minimum values kept in the signature, we use \(\text{sig}_{A,1}, \text{sig}_{A,2}, \ldots\) to denote the first minimum, the second minimum and so on in the signatures. (In other words, \(\text{sig}_A\) used previously is now equivalent to \(\text{sig}_{A,1}\).) Figure 4.3 gives an illustration of how to compute the signature with keeping the second minimum value for an intersection of two sets, i.e., \(\text{sig}_{A \cap B, 1}\) and \(\text{sig}_{A \cap B, 2}\) from \(\text{sig}_{A,1}, \text{sig}_{A,2}, \text{sig}_{B,1}\) and \(\text{sig}_{B,2}\). This is a continuation of the situation shown in Figure 4.2. What is different in Figure 4.3 is that there are now two solid circles for each set, corresponding to the first and second minima. Recall from Figure 4.2(a) that with only the first minimum kept, \(\hat{\text{sig}}_{A \cap B, 1}[1] = \max(\text{sig}_{A,1}[1], \text{sig}_{B,1}[1]) = 2\). But with two minima kept, as shown in Figure 4.3, it is clear that no element in \(A \cap B\) can have a hash value of 2 since \(\text{sig}_{A,1}[1] < 2 < \text{sig}_{A,2}[1]\). If there had been an \(x \in A\) such that \(\pi_1(x) = 2\), then we would have chosen 2 as \(\text{sig}_{A,2}[1]\) not 3. Similarly, because \(\text{sig}_{B,1}[1] < 3 < \text{sig}_{B,2}[1]\), no element in \(A \cap B\) can have a hash value of 3. Thus, we can infer \(\text{sig}_{A \cap B, 1}[1] = 4\), which turns out to be the correct value in this example. As there is no additional minimum values kept in
4.4. Estimation with Set Hashing Signatures

**Figure 4.3: LBS Using the First and Second Minima**

**Figure 4.4: Estimation of Signatures**

While the discussion so far is based on the intersection of two sets $A$ and $B$, the computation can be modeled as a binary operator. When more than two sets are intersected, this
binary operator can be applied successively. Note that the LBS estimation algorithm shown earlier does not change, except for line (6). This line is now generalized to compute the first and second minima of the signature. However, only the first minimum values are used to estimate frequency and resemblance in subsequent lines. For our example, if we focus our attention on \( \hat{\text{sig}}_{A \cap B,1} \) for all the four permutations, the signature becomes \([4,4,2,1]\), which is a closer match to the true signature of \([4,5,2,1]\) than \([2,4,2,1]\) in Figure 4.2(b) with only the first minimum. This shows that extra minima may help to better approximate signatures for intersections.

4.5 Extensions to Other Similarity Measures

In this section, we discuss extensions to other similarity measures. We showcase two extensions: the LIKE clause and Jaccard similarity.

4.5.1 SQL LIKE Clauses

We will consider two major constructs of LIKE: `\%` and `\_`. `\%` character matches any substring and `\_` character matches any single character.

Suppose a LIKE pattern `\%s_1\%...\%s_m\%`. For example, the predicate “name like `%silvia%carbonetto%`” selects all the names that contain `silvia` followed by `carbonetto` with any number of characters in between.

Assume for now that each \(s_i\) (\(1 \leq i \leq m\)) is a plain substring without any special character like `\_`. Any string that satisfies the LIKE condition must have all of \(s_i\)’s, \(1 \leq i \leq m\) as its substring. Recall that \(G_b\) denotes the set of tuple ids in the DB that have \(b\) as a substring. Then the size of the set of tuple ids in DB that match the LIKE condition \(s_{\text{like}} \equiv \%s_1\%...\%s_m\%\) can be estimated as:

\[
|\hat{\text{Ans}}(s_{\text{like}})| = |\bigcap_{1 \leq i \leq m} G_{s_i}|. \tag{4.10}
\]

This is an upper bound of the true selectivity but a tighter bound than \(\min |G_{s_i}|\) proposed in [26]. Note that LBS utilizes signature and it can handle intersection as well as union. We make use of the next formula for intersection size estimation [79].

\[
|G_{S_1} \cap \cdots \cap G_{S_m}| = \gamma_{m} \cdot |G_{S_1} \cup \cdots \cup G_{S_m}| \approx \hat{\gamma}_{m} \cdot |\hat{G}|, \tag{4.11}
\]

where \(\gamma_{m}\) is the resemblance of \(G_{S_1}, \ldots, G_{S_m}\) which can be calculated by Equation (4.3) extended to \(m\) sets. \(|\hat{G}|\) is the estimated union size and it is actually the estimation of LBS, Equation (4.8). Thus our estimation of Equation (4.10) is the output of LBS multiplied by \(\hat{\gamma}_{m}\). We only highlight key modifications due to space restriction.

1. It generates base substrings for each \(s_i\) setting \(\tau = 0\) in line (2) to (8) of Algorithm 1. (i.e., each \(s_i\) becomes one base substring.)

2. It returns the value in line (16) of Algorithm 3 multiplied by \(\gamma_{m}\).
4.5. Extensions to Other Similarity Measures

If \( s_i \) contains \( \_ \), which matches any single character, we only need to substitute \( \_ \) with our wildcard \( \? \) when generating base substrings.

One interesting observation is that \( \gamma_m \) can be quite small in Equation (4.11). To estimate \( \gamma_m \) in the style of Equation (4.3), we would need a larger \( L \), the number of random permutations in set hashing, for small \( \gamma_m \). Note that small \( L \) is enough in Equation (4.4). \( \gamma = \frac{|A_j|}{|A|} \) is generally not small since \( |A_j| \), the frequency of the most frequent base substring, is not a small fraction of \( |A| \) from the generalized SIS assumption. However, in Equation (4.11), we cannot make the same assumption on the relative size of \(|G_s_1 \cap \cdots \cap G_{s_m}| \) and \(|G_s_1 \cup \cdots \cup G_{s_m}| \). For example, in the query '%silvia%carbonetto%', \( \text{freq}(\text{silvia}) \) may turn out to be very high whereas \( \text{freq}(\text{carbonetto}) \) can be relatively low. If \( \text{freq}(\text{silvia}) = 10,000 \) and \( \text{freq}(\text{carbonetto}) = 10,000 \) and \( L \) should be at least 1,000 to express the small \( \gamma \). Let \( f_{\min} = \min |G_{s_i}| \) and \( f_{\max} = \max |G_{s_i}| \). Whenever \( |G|/f_{\min} > L \) or \( \gamma_m = 0 \), we know that our estimation is likely to be inaccurate. Thus, before multiplying the output of LBS and \( \hat{\gamma}_m \) we check for the two conditions. If one of them is true, then \( \gamma_m \) is set to \( f_{\min}/f_{\max} \). Experimental results on LIKE predicates in Section 6 reflect this heuristic.

4.5.2 Jaccard Coefficient

The Jaccard similarity [140] between two strings \( a \) and \( b \) is defined as:

\[
J_S(a, b) = J_S(A, B) = \frac{|A \cap B|}{|A \cup B|},
\]

where \( A \) and \( B \) are sets of \( q \)-grams of \( a \) and \( b \) respectively. For instance, when \( q = 3 \), \( J_S(\text{‘tyrannosaurus’, ‘allosaurus’}) = J_S(\{ \text{tyr, yra, ran, ann, noo, nos, osa, sau, aur, uru, rus} \}, \{ \text{all, llo, osa, sau, aur, uru, rus} \}) = \frac{5}{13} \). Our extension starts from the intuition that if two sets are similar, their sizes cannot be too different. If \( J_S(a, b) = \gamma \), then \( \gamma \cdot |A| \leq |B| \leq 1/\gamma \cdot |A| \) [8]. Substituting \( |A| = \text{length}(a) - q + 1 \) and \( |B| = \text{length}(b) - q + 1 \) into the above inequality gives us

\[
[\gamma \cdot (\text{length}(a) - q + 1)] \leq \text{length}(b) - q + 1 \leq [1/\gamma \cdot (\text{length}(a) - q + 1)]
\]

Using this property, given a query string \( s_q \) and a Jaccard similarity threshold \( \gamma \), we can derive conditions on the length of strings that satisfy the similarity condition. We use this minimum and maximum string length in line (2) of Algorithm 1 and generate base substrings in the same way. However, not all the base substrings generated such have \( J_S \geq \gamma \), so we filter base substrings treating the wildcard as a separate character. Before line (12) of Algorithm 3, we check every base substring \( b \) and see if \( J_S(s_q, b) \geq \gamma \). If it is smaller than \( \gamma \), we discard the base substring. So when we compute the union size, we only consider those base substrings that are guaranteed to have \( J_S \geq \gamma \).
4.6 Empirical Evaluation

4.6.1 Experimental Setup

Data sets: We perform a series of experiments using three benchmarks: DBLP author names and titles and IMDB movie Keywords data. There are 699,199 full names in the DBLP authors data set. The average and maximum length are 14.1 and 38 respectively. There are 305,364 titles in the DBLP titles data set. The average and maximum length are 58.6 and 100 respectively. Finally, there are 100,000 keywords in the IMDB Keywords data set, with the average and maximum length being 10.3 and 62. For each data set, we generate random queries $Q \equiv (s_q, \tau)$. These queries are divided into two overlapping query sets:

- The short query set consists of query string $s_q$ which is a substring of a random word of length 5 to 12 in any tuple. When the word length is 7 or less, we use the whole word. There are 200 queries in this set. The edit distance threshold $\tau$ is set to 25% of the query string length (i.e., $\tau \leq 3$). As motivated by the Short Identifying Substring (SIS) assumption in [26], $\tau \leq 3$ can find many database applications. The average selectivity is 1.03% for the Keyword data set and 0.55% for the DBLP authors data set.

- The long query set consists of query string $s_q$ which is a random word of length 10 to 20 in any tuple. The long set is the most meaningful for the DBLP titles data set. There are 100 queries in this set. The edit distance threshold $\tau$ is again set to 25% of the query string length (i.e., $3 \leq \tau \leq 5$). The average selectivity is 3.17%.

- The negative query set consists of queries whose true frequency is 0. We randomly choose a word in a tuple and replace up to 3 random positions with random characters. Out of 200 chosen queries, 52 have true frequency of 0. Notice that there might be tuples that match the newly formed query by chance. This type of query is important especially given the issue of unclean data [87]. If a query optimizer can accurately identify true negative predicates or the estimated frequency is fairly accurate (i.e., the estimation is close to zero), the predicate will remain the most selective condition for query processing.

Evaluation metric: To evaluate the accuracy of an estimation method we rely on three different metrics. The first metric we employ is relative error which is defined as $|f_{est} - f_{true}|$. The relative error is calculated as:

$$\text{Relative Error} = \frac{|f_{est} - f_{true}|}{f_{true}} \times 100\%$$
4.6. Empirical Evaluation

\( f_{\text{true}} / f_{\text{true}} \), where \( f_{\text{true}} \) is the true frequency of the query and \( f_{\text{est}} \) is the estimated frequency. To prevent the accuracy from being distorted by queries with small true frequencies, we exclude queries whose true frequency is less than or equal to 10. For such small queries, the second metric, absolute error \( |f_{\text{est}} - f_{\text{true}}| \) is employed. It is also used to report errors of negative queries to avoid division by 0. The third metric we apply is the relative error distribution. We show the distribution by providing a histogram of the relative errors, i.e., \([-100\%, -75\%), \ldots\), etc.

**Estimation methods implemented:** MOF is parameterized by the extended N-gram table, where \( N \) was varied from 4 to 6. As discussed in Section 3.1, the default coverage \( \rho \) was acquired by sampling. The following table shows the average value and the standard deviation of \( \rho \) using the DBLP Authors data set. The table shows that the average \( \rho \) is rather stable with respect to the sample size.

<table>
<thead>
<tr>
<th>Sample Size</th>
<th>10</th>
<th>20</th>
<th>50</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average ( \rho )</td>
<td>0.6453</td>
<td>0.6445</td>
<td>0.6523</td>
<td>0.6475</td>
</tr>
<tr>
<td>Standard deviation</td>
<td>0.081</td>
<td>0.071</td>
<td>0.034</td>
<td>0.000</td>
</tr>
</tbody>
</table>

Like MOF, LBS is parameterized by \( N \). It is also parameterized by \( k_{\text{min}} \), which controls the number of minimum values for a signature. It was varied from 1 to 3. Thus, the results of LBS are denoted by LBS(\( N, k_{\text{min}} \)). Strictly speaking, MOF and LBS are tunable by two other parameters \( PT \) and \( L \), where \( PT \) determines the minimum frequency for a \( q \)-gram to be kept in the \( N \)-gram table and \( L \) controls the number of permutations used, i.e., from \( \pi_1 \) to \( \pi_L \). For results reported here, we set \( PT = 20 \) and \( L = 10 \), unless specified. Finally, in LBS, the default value \( \rho \) is used when the estimated resemblance by signature is \( \leq 0.2 \) and 10 most frequent base substrings are used in line (12) of Algorithm 3. We implemented our set hashing based techniques in Java 1.5, and a hash space of \( 2^{15} \) was used.

S-SEPIA was implemented in C++ by modifying the SEPIA code downloaded from [138]. We used 2,000 clusters and the CLOSED_RAND [79] method to populate the histograms. To limit the building time to around 48 hours, we restricted the maximum number of sampled substrings per tuple to be 10, and set the sampling ratio according to the data sets. The space consumption was measured by the size of the data structure written on disk.

Finally, RS was also implemented in Java 1.5. Because RS requires sampling from the database \( DB \), the I/O cost may vary depending on the buffering policy. To simplify query time comparisons, we only considered the CPU cost of RS. In other words, we under-estimated the true cost of running RS in practice; it was sufficient for the eventual conclusions.

All experiments were conducted on P4 3GHz machines with 2 GB memory running GNU/Linux with kernel 2.6.

### 4.6.2 Short Query Set: Estimation Accuracy vs Space Overhead vs Query Time

We begin with the short query set, i.e. \( \tau \leq 3 \), for the DBLP author data set. Figure 4.5 shows the average relative error, and the corresponding space overhead and query time in milliseconds. The latter two graphs are drawn in log scale.
MOF vs S-SEPIA: Let us first focus on the comparison between MOF and S-SEPIA. S-SEPIA uses a sampling ratio of 0.25%. For DBLP authors, S-SEPIA gives an average relative error of 64% and uses close to 100 MB of memory, whereas MOF(4) gives a better average error of 53% but using only 0.5 MB! As \( N \) increases, MOF(5) and MOF(6) give significantly better relative average error. Yet the amount of space used as shown in Figure 4.5(b) is still significantly smaller than that used by S-SEPIA.

Recall that S-SEPIA uses the parameter \( c \) to control the number \( c \ell \) substrings of \( s \) to be used for clustering. The results reported in the figures so far are based on \( c = 1 \). We suppress the detailed results here, but point out that increasing \( c \) beyond 1 does not give clear empirical performance benefit.

In terms of runtime, it takes around 30 and 120 minutes to build the extended 5-gram and 6-gram tables. But the building time for S-SEPIA is around 35 hours. For query estimation, as shown in Figure 4.5(c), the processing time of MOF is around 10 milliseconds. S-SEPIA’s runtime is around 30 milliseconds. In sum, MOF outperforms S-SEPIA in providing higher average estimation accuracy, lower building time and less space overhead.

MOF vs RS: For method RS, the estimation accuracy and CPU time required are directly proportional to the amount of sampling done. We show in Figure 4.5 two settings of RS: with 0.1% or 0.5% of the whole database sampled. For estimation accuracy, RS(0.5%) gives an average relative error comparable to MOF(5) and MOF(6). However, all the MOF variants are almost two orders of magnitude faster. In practice, RS may take even longer if I/O cost is incurred. The faster version RS(0.1%) gives significantly higher average relative error. Thus, with a very modest space overhead, MOF dominates RS in providing either superior query time performance or higher estimation accuracy.

A variant of RS is to keep a sampled database around for query time. The hope is that the
space overhead would significantly reduce query time. Unfortunately, the sampled database only reduces the time for sampling, not the time to check the edit distance threshold for all the substrings. The latter by far dominates the former.

**MOF vs LBS:** Next let us compare MOF with LBS on the trade-off between accuracy and space overhead. (The query time of the two algorithms are almost identical.) The ordering in terms of descending average relative error is: MOF(4) > LBS(4,1) > MOF(6) ≈ MOF(5) > LBS(5,1) > LBS(5,2) > LBS(6,1). The corresponding space overhead ordering is almost the reverse. This exactly shows how LBS and MOF can leverage extra space to give lower error. Specifically, the difference in size between the pair MOF(N) and LBS(N,1) is due to the use of signatures by LBS. To formally test whether the differences observed are statistically significant, we use the standard 2-tail Student’s t-test to compute a p-value, i.e., the probability of a chance observation. The difference of LBS over MOF is well below 0.01 and is confirmed to be statistically significant. Moreover, even though there does not appear to be a big difference in average relative error between LBS(5,1) and LBS(5,2), the latter’s superiority is statistically significant with a p-value below 0.01.

**Relative Error Distributions:** As an explanation to the average relative error results shown in Figure 4.5, Figure 4.7 shows the error distributions for DBLP authors. For space reason, only the distribution of S-SEPIA and LBS(5,2) is shown. The downfall of S-SEPIA can be summarized by the high number of queries that are in the [-100%,-50%] range (i.e., under-estimation) and those in the [100%,∞) bucket (i.e., over-estimation). In contrast, LBS(5,2), perform significantly better in those situations.

**Small and Negative Query Sets:** The table below summarizes the average absolute errors on the small and negative query sets. There are 4 and 8 queries whose true frequencies are less than or equal to 20 or 50 respectively.

<table>
<thead>
<tr>
<th></th>
<th>(f_{true} \leq 20)</th>
<th>(f_{true} \leq 50)</th>
<th>Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>RS(0.1%)</td>
<td>15</td>
<td>22</td>
<td>0</td>
</tr>
<tr>
<td>RS(0.5%)</td>
<td>15</td>
<td>44</td>
<td>0</td>
</tr>
<tr>
<td>S-SEPIA</td>
<td>26</td>
<td>45</td>
<td>16</td>
</tr>
<tr>
<td>LBS(4,1)</td>
<td>13</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>LBS(5,1)</td>
<td>13</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>MOF(6)</td>
<td>10</td>
<td>17</td>
<td>0</td>
</tr>
<tr>
<td>LBS(5,2)</td>
<td>12</td>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>LBS(6,1)</td>
<td>13</td>
<td>13</td>
<td>0</td>
</tr>
</tbody>
</table>

For the small query sets, MOF and LBS are superior to RS and S-SEPIA. For the negative query set, we can observe clear benefit from increasing \(N\). When \(N = 6\), both MOF and LBS exactly estimate the frequency of negative queries as zero. For other settings of LBS, even if the estimated frequencies are not exactly zero (e.g., 1 or 3), the frequencies are sufficiently accurate that it is likely that the predicate will be chosen as the most selective condition. In contrast, for S-SEPIA, because the estimated frequency is inaccurate (e.g., 16), there is a higher chance that incorrectly, another predicate will be selected as the most selective condition, resulting in a more expensive query plan.

### 4.6.3 Query Sets on DBLP Titles

Figure 4.6 shows the average relative error, the space overhead and CPU query time in log scale for the short query set on the DBLP titles data set. In the context of the earlier
4.6. Empirical Evaluation

Figure 4.8: Long Query Sets on DBLP Titles

Discussion on Figure 4.5, the key highlights for the DBLP title data set are as follow. First, RS(0.5%) gives better average relative error in Figure 4.6(a) than in Figure 4.5(a). However, the conclusion remains the same in that MOF and LBS are almost two orders of magnitude faster. Second, S-SEPIA is still dominated by MOF and LBS in both estimation accuracy and space overhead. Finally MOF(5) is dominated by MOF(6), LBS(5,1), LBS(5,2) and LBS(6,1).

The results reported so far are based on the short query set with \( \tau \leq 3 \). Figure 4.8 shows the average relative error on the long query set on DBLP Titles with \( 3 \leq \tau \leq 5 \). The results on S-SEPIA are not included as it takes too long to build even with a very low sampling ratio. In long queries with higher edit thresholds, it is not practical for MOF and LBS to enumerate all possible base substrings. One way to cap the computational cost is to generate only up to a specific number of base substrings. In other words, we set a limit for \( |B_c| \) in line (6) of Algorithm 1 by sampling. We randomly generate up to 200 base substrings, and the results
are shown in Figure 4.8.

In terms of the average relative error comparisons amongst RS, MOF and LBS, all the previous observations on the short query sets remain valid here. As for runtime, for MOF and LBS, the query time increases from an average 20 milliseconds for the short query set to an average 120 milliseconds for the long query set. The RS method scales up poorly with respect to \( \tau \) as its query time is well over 2000 milliseconds. In sum, both MOF and LBS are capable of handling long queries and larger \( \tau \) thresholds.

The space overhead graph is not included here. With a larger \( \tau \), additional q-grams are needed to handle the larger number of wildcards (e.g., q-grams with 4 or 5 wildcards for our long query set). Compared with the space overhead shown in Figure 4.6(b), the additional space required turns out to be rather minimal (i.e., between 0.1MB to 0.5MB).

4.6.4 Other Similarity Measures

For the LIKE predicate, we consider two types of conditions, \( %s\% \) and \( %s_1s_2\% \) as is typical in TPC-H benchmark. We randomly select one or two words of length between 5 and 12 and introduced 0 to 2 \( \_\_\_\_\_\_\_\_ \) in each word. Figure 4.9(a) gives the average relative errors on LIKE predicate. MOF and S-SEPIA are not present because MOF does not have signature which is necessary for Equation (4.11) and S-SEPIA does not support LIKE predicate. We observe that LBS outperforms RS by a large margin. Even with 0.5\% of sampling ration, RS’s average relative error is greater than 100\% while LBS(5,1) is 56\%. The effect of increasing \( \tau \) is also clear. In LBS(6,1), the average relative error is as low as 35\%.

For Jaccard similarity, we randomly select a word for similarity threshold \( J_{sim} \) of 0.6, 0.7, 0.8 and 0.9. Figure 4.9(b) plots the average relative error on Jaccard similarity. As in edit distance or LIKE cases, LBS consistently outperforms S-SEPIA, MOF and RS.

4.6.5 Impact of Parameters: \( k_{\text{min}}, PT \)

In Figure 4.5(a), there is the comparison between LBS(5,1) and LBS(5,2) using the DBLP Authors data set. This suggests that as \( k_{\text{min}} \), the number of minima kept, increases from 1 to 2, there is a reduction in average relative error by using additional space. Figure 4.10(a) and (b) analyze the impact of \( k_{\text{min}} \) in greater details. Specifically, the IMDB Keywords data set is used with \( N = 4 \) or 5, and \( PT = 10 \), \( k_{\text{min}} \) is varied from 1 to 3. When an extended 4-gram table is used (i.e., \( N = 4 \)), keeping the second minimum reduces the error by 14\% relative to the first minimum; keeping the second and the third minima reduces the average relative error by 20\%. However, this reduction is less in the 5-gram case. In general, when \( N \) is small, having \( k_{\text{min}} = 2 \) helps. According to Figure 4.10(b), it is important to note that the additional space needed by incrementing \( k_{\text{min}} \) is less than the extra space required by incrementing \( N \). For instance, the space consumption of LBS(4,2) is less than the space consumption of LBS(5,1).

Figure 4.10(c) and (d) show the impact of changing the pruning threshold \( PT \) based on the keywords data set and LBS(5,1). In Figure 4.10(d), we note a sharp drop in the space overhead, especially in the low prune threshold. This is not so surprising considering that many real world text data follow Zipf’s law. The dropping ratio is bigger at higher \( N \).
4.6. Impact of Data Set Size

Figure 4.11 compares the average relative errors of N-gram tables for LBS(4,1), LBS(5,1), and LBS(6,1) varying the data set size. The Title300 data set is the full Titles data set used throughout the experiments and we randomly select 50,000 and 100,000 titles for the Title50 and Title100 data set respectively. We can observe that average relative error does not change much as the data set increases in size.

4.6.7 Recipe: Balancing Space vs. Accuracy

MOF and LBS provide a very tunable setting depending on the space available and CPU time expectations. The table below offers a “recipe” for choosing the method and the parameters.

<table>
<thead>
<tr>
<th>Available Space</th>
<th>Suggested Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>very low (&lt;5%)</td>
<td>MOF(4)</td>
</tr>
<tr>
<td>low (&lt;30%)</td>
<td>MOF(5)</td>
</tr>
<tr>
<td>medium (&lt;70%)</td>
<td>LBS(5,1)</td>
</tr>
<tr>
<td>abundant (&lt;120%)</td>
<td>LBS(5,2)</td>
</tr>
<tr>
<td>non-issue (&lt;400%)</td>
<td>LBS(6,1)</td>
</tr>
</tbody>
</table>

When we have very limited amount of space, say 1% ~ 5% of the original data size, MOF(4) is the recommended choice. If CPU time is not a serious concern, MOF(5) is a good choice. If accuracy is a primary concern, the LBS estimation algorithm is recommended, particularly LBS(5,1) and LBS(5,2). Keeping signatures help to reduce relative error. Increasing \( k_{\text{min}} \) involves less space than increasing \( N \). Finally, if space is not an issue, then LBS(6,1), and even LBS(7,1), is recommended.

4.6.8 Summary of Experiments

We evaluated the proposed solutions, MOF and LBS, using three real-world data sets: IMDB keywords, DBLP author names, and DBLP paper titles. When the space budget is tight, say
4.7. Conclusions

less than 10% of the data size, MOF can deliver good estimation. If more space is allowed, LBS can further improve estimation accuracy. They are compared with S-SEPIA, which is an adaptation of SEPIA [79] to the SUBSTR problem, and random sampling (RS). S-SEPIA’s space overhead was much larger than the space for MOF or LBS. With enough sample size, RS was as accurate as LBS, but its was very CPU intensive.

4.7 Conclusions

In this chapter, we develop algorithms for estimating selectivity of approximate substring matching with edit distance. Two algorithms based on the extended N-gram table, MOF and LBS, show accurate and fast estimation. MOF provides simple yet accurate estimation, and LBS improves MOF by capturing more complex correlation among strings by adapting from set hashing signatures. We extend the proposed algorithms to SQL LIKE predicate and Jaccard similarity. As ongoing work, we explore further utilization of the stored signature information in LBS.
Chapter 5

Set Similarity Join Size Estimation

In the following two chapters, we study similarity join size estimation problems. In the selection problems in Chapters 3 and 4, we estimate the number of records that a given predicate selects. In the join problems in Chapters 5 and 6, we estimate the number of pairs of records that satisfy a given join predicate. Join size estimation has been one of major challenges in query optimization in RDBMSs [71]. Most of studies on this problem have focused on equi-join size estimation. Some of them can be extended to similarity joins, but we cannot leverage many tools that were possible in the equi-join problem, e.g., histograms. In this chapter, we study the SSJ problem in Section 1.4.3 where a database is a collection of sets. The proposed technique is described in our VLDB 2009 paper [92].

5.1 Introduction

Given a similarity measure and a minimum similarity threshold, a similarity join finds all pairs of records whose similarity under the measure is greater than or equal to the minimum threshold. Since a set can generalize many data types, the Set Similarity Join (SSJoin) is a common abstraction of a similarity join problem. For instance, a large scale customer database may have many redundant entries which may lead to duplicate mails being sent to customers. To find candidates of duplicates, addresses are converted into sets of words or n-grams and then an SSJoin algorithm can be used. SSJoin has a wide range of applications including query refinement for web search [127], near duplicate document detection and elimination [17]. It also plays a crucial role in data cleaning process which detects and removes errors and inconsistencies in data [8, 130]. Accordingly, the SSJoin problem has recently received much attention [8, 13, 27, 60, 61, 130].

It is noted that SSJoin is often used as a part of a larger query [8]. In this scenario, a user may want to retrieve all similar pairs possibly conditioned by other predicates or constraints. Thus, it is not a one-time operation and is performed repeatedly with different predicates by users. To handle these general similarity queries, Chaudhuri et al. identified the SSJoin operation as a primitive operator for performing similarity joins [27]. The SSJoin operation can also be important in inconsistent databases. Fuxman et al. [48] proposed a system to answer SQL queries over inconsistent databases where the data cleaning operation is performed on-the-fly.

To successfully incorporate the SSJoin operation in relation database systems, it is imperative that we have a reliable technique for the SSJoin size estimation. The query optimizer needs an accurate estimation of the size of each SSJoin operation to produce an optimized query plan. However, to the best of our knowledge, this problem has not been studied previously in the literature. This motivates us to study the SSJoin size estimation problem.
In this chapter, we study the SSJ problem in Definition 3. Several facts are worth mentioning in Definition 3. First, it corresponds to a self-join case. Although the join between two collections of sets is more general, many applications of SSJoin are actually self-joins: query refinement [127], duplicate document or entity detection [17, 64], or coalition detection of click fraudsters [107]. Naturally, a majority of the SSJoin techniques proposed are evaluated on self-joins [8, 13, 27, 130]. Second, self-pairs \((r, r)\) are excluded in counting the number of similar pairs so that the answers are not masked by the big count, \(|R|\). In addition, we do not distinguish between the two different orderings of a pair (i.e., \((r, s) = (s, r)\)). It is, however, trivial to adapt the proposed technique to consider the self-pairs or the ordering of pairs in the answer if necessary.

Random sampling is a natural technique for the selectivity (or size) estimation problems since it does not suffer from the attribute value independence assumption and is not restricted to equality and range predicates. Recently, the technique of Hashed Sampling was proposed for the result size estimation of the set similarity for selection queries [61]. It builds a randomly sampled inverted index structure for the selection queries and the selectivity of a query is estimated by performing selection query processing on the samples. Basically join size can be estimated using the Hashed Sampling by applying an SSJoin algorithm on the constructed inverted index. However, it suffers from quadratic processing time and may not be useful for query optimization purposes. Furthermore, we observe that the estimate is highly dependent on the actual samples used (i.e., large variance). This can be compensated by having much larger sample sizes, which makes it impractical for query optimization.

In this chapter, we hypothesize that it is possible to perform SSJoin size estimation in a sample-independent fashion. As an overview, our approach works as follows. We first generate Min-Hash signatures for the samples filtered by other predicates. As a succinct representation of a set, min-hash signatures generally result in a faster operations due to its smaller size [17, 34]. Then we perform frequent pattern mining on the signatures. The distribution information of the signature patterns, captured in Power-laws, is used in estimating the size of union. Finally, the estimated size is adjusted so that it correctly reflects the SSJoin size in the original set space. Specifically, we make the following contributions.

- We reduce the SSJ problem to a pattern mining problem on the signatures, which enables very efficient counting of pairs.
- Naive approaches using the signature patterns fail since a pair could be counted multiple times in several patterns. We propose a novel counting technique called Lattice Counting which counts the number of pairs satisfying the threshold minimizing over-counting.
- Recently, it was observed in [33] that pattern frequency-count distribution generally follows a Power-law distribution. We exploit this observation to efficiently extract the necessary information for the Lattice Counting formula. It is possible because we only need the count information and do not need the actual patterns. A high minimum support threshold can be used so that it is fast enough to be used in query optimization.
- We observe that there is a systematic overestimation when Min-Hashing is used. We establish a model to simulate the count shift. An efficient method for correcting the overestimation is proposed.
5.2. Signature Pattern

This chapter is structured as follows. Section 5.2 introduces the Min-Hash signature and signature pattern, and gives an overview of the framework. Section 5.3 presents the Union Formula which gives the SSJoin size based on the mined pattern distribution. Efficient processing based on the Power-law distribution is proposed in Section 5.4. A procedure to overcome the systematic overestimation by Min-Hashing is introduced in Section 5.5. The experimental results are presented in Section 5.6. Section 5.7 concludes with future directions.

5.2 Signature Pattern

5.2.1 Min-Hash Signature

We use Min-Hash signatures introduced in Section 4.2.2 for a succinct representation of a set. The Min-Hash signature of \( r \) is denoted by \( \text{sig}(r) \) and we will call it the signature of \( r \) in the remainder of the chapter. For each set \( r \), we build its Min-Hash signature \( \text{sig}(r) \). We denote the Min-Hash signature representation of \( DB \) by \( \text{sig}(DB) \).

A difference from its use in Chapter 4 is that a Min-Hash signature is built for each record and it concisely encodes elements in the record. In Chapter 4, however, a Min-Hash signature is built for an element (q-gram), not for a record, and it encodes the record ids that contain the element. The use of Min-Hash signature in this chapter (resp. Chapter 4) bears similarity to that of HSol (resp. VSol) in [105].

Figure 5.1 shows an example \( DB \) and corresponding signatures, \( \text{sig}(DB) \). The signature size \( M = 4 \) and the universe of hash functions is \([1, 5]\). Hash functions are not shown for simplicity.

\[
\begin{array}{|c|c|}
\hline
\text{DB} & \text{sig(DB)} \\
\hline
r_1 & \{7, 10, 19, 52, 67\} & \text{sig}(r_1) & \{4, 3, 5, 2\} \\
r_2 & \{10, 19, 43, 52\} & \text{sig}(r_2) & \{4, 3, 5, 2\} \\
r_3 & \{10, 13, 43, 52, 67, 85\} & \text{sig}(r_3) & \{4, 3, 2, 2\} \\
r_4 & \{10, 38, 43, 49, 80, 94\} & \text{sig}(r_4) & \{3, 3, 3, 2\} \\
r_5 & \{3, 25, 29, 47, 50, 66, 73, 75\} & \text{sig}(r_5) & \{1, 1, 3\} \\
\hline
\end{array}
\]

Figure 5.1: An Example DB

Suppose \( \tau = 0.5 \) is given. Using Equation (4.3), we can estimate \( J_S(r_i, r_j) \) for \( r_i, r_j \in DB \) with their signatures. For instance, \( J_S(r_1, r_2) = 0.5 \) since \( \text{sig}(r_1) \) and \( \text{sig}(r_2) \) match at two positions (i.e., the first and the second) out of 4 positions. The true similarity, \( J_S(r_1, r_2) \) is \( \frac{|\{10, 19, 52, 67\} \cap \{7, 10, 19, 43, 52, 67\}|}{|\{7, 10, 19, 43, 52, 67\}|} = 0.5 \). Considering all 5 sets in \( DB \), there are 6 pairs \( \{(r_1, r_2), (r_1, r_3), (r_2, r_3), (r_1, r_4), (r_3, r_4), (r_2, r_4)\} \) that have at least two matching positions out of 4 positions in their signatures and thus \( J_S \geq 0.5 \).

Using the Min-Hash signatures, SSJoin size estimation can be done by estimating the number of pairs \((r, s)\) such that \( \text{sig}(r) \) and \( \text{sig}(s) \) overlap at least at \( \lfloor \tau \cdot M \rfloor \) positions. A naive way to estimate the join size using the signatures is to first build signatures for all sets and then to compute every pairwise similarity. Although similarity computation between two signatures are generally assumed to be cheaper, this approach still has quadratic complexity.
5.2. Signature Pattern

5.2.2 The FSP Problem

An interesting observation is that any pair \((r, s)\) from a group of records which shares the same values at least at \(\lceil \tau M \rceil\) positions in their signatures satisfy \(J_S(r, s) \geq \tau\) with high probability. For instance, in the above example, \(r_1, r_2\) and \(r_3\) have 4 and 3 at the first and the second signature positions. Any pair from \(r_1, r_2, r_3\) matches at least at those two positions in their signatures. There are 3 such pairs: \((r_1, r_2), (r_1, r_3), (r_2, r_3)\) and they are estimated to have \(J_S \geq 0.5\). We formalize this intuition with signature patterns.

**Definition 5 (Signature Pattern).** A signature pattern is a signature where any of its values is possibly substituted with \(X\) which denotes a ‘don’t care’ position. A signature (or a set in the database) matches a signature pattern if its signature values are the same as the pattern values at all positions that are not \(X\). Signature values at positions marked by \(X\) do not matter.

For a pattern, \(\text{sig}\), we define the following notations.

- **length**, \(\text{len(sig)}\): the number of non-\(X\) values in the pattern;
- **frequency**, \(\text{freq(sig)}\): the number of sets in the database whose signatures match the pattern.

**Example 3.** Figure 5.2 shows signature patterns with \(\text{freq} \geq 2\) found in the example of Figure 5.1 with their lengths, matching sets and frequencies. For instance, \(\text{len}([4, 3, X, X]) = 2\) since it has two non-\(X\) values. Its frequency is 3 because three sets, \(r_1, r_2, r_3\) match the pattern.

Based on the signature pattern, we define the Frequent Signature Pattern problem as follows.

**Definition 6 (FSP Problem).** Given a threshold \(t\) on the pattern length, \(\text{FSP}(t)\) is the number of pairs which share a pattern \(\text{sig}\) such that \(\text{len(sig)} \geq t\) and \(\text{freq(sig)} \geq 2\).

To solve the \(\text{SSJ}(\tau)\) problem, we solve the \(\text{FSP}(t)\) problem where \(t = \lceil \tau M \rceil\). In the sequel, we implicitly assume a fixed value of \(M\). If we view the signature of each set as a transaction and each signature value along with its position as an item, we can apply traditional frequent pattern mining algorithms to the \(\text{FSP}\) problem. Given \(\tau\) we discover the signature patterns with \(\text{length} \geq \lceil \tau M \rceil\) and \(\text{freq} \geq 2\). A minimum frequency of 2 is necessary to produce a pair. If the frequency of a pattern is \(k\), there are at least \(n_{\text{pair}}(k) \equiv \binom{k}{2} = k(k - 1)/2\) pairs satisfying the similarity threshold \(\tau\) in the database. Exploiting this observation, we can have the following two step approach in estimating the \(\text{SSJoin}\) size.
5.3. Lattice Counting

1. Compute the frequent signature patterns with \( \text{length} \geq \lceil \tau M \rceil \) and \( \text{freq} \geq 2 \).

2. For each pattern of frequency \( k \), count the number of pairs by \( \text{npair}(k) = k(k - 1)/2 \), and then aggregate all the number of pairs.

However, this approach has several critical challenges.

- There are overlaps among the signature patterns. Consider \( \text{sig}_1 = [4, 3, X, X] \), \( \text{sig}_2 = [X, 3, X, 2] \). If \( \tau = 0.5 \cdot 4 \), both patterns should be considered in the answer since their \( \text{length} \geq 2 = 0.5 \cdot 4 \). There are 3 pairs from \( \text{sig}_1 \) and 3 pairs from \( \text{sig}_2 \) satisfying \( \tau = 0.5 \): \((r_1, r_2)\), \((r_1, r_3)\), \((r_2, r_3)\) from \( \text{sig}_1 \) and \((r_1, r_4)\), \((r_3, r_4)\) from \( \text{sig}_2 \). If we simply add the two counts 3 and 3, we are counting the pair \((r_1, r_3)\) twice. Without consideration of this overlap, the estimate could be a large overestimation.

- We need all signature patterns with \( \text{freq} \geq 2 \). It is equivalent to setting the minimum support threshold to 2 in the frequent pattern mining problem. However, the search space for such a low threshold is not generally feasible.

We address the first issue in the following section and the second issue in Section 5.4.

5.3 Lattice Counting

In this section, we describe how we can consider overlapping counts of signature patterns. We show that join size can be computed with union size of signature patterns and there are underlying lattice structures among the patterns. The framework proposed for the STR problem in Chapter 3 is applied to compute the union size. Although the STR problem and the SSJoin problem do not bear immediate affinity, we show that techniques developed in Chapter 3 can serve as the basis with novel insights on the connection.

5.3.1 Computing The Union Cardinality

Let \( S_p(sig) \) denote the set of pairs not including the self-pairs that match with a pattern \( sig \). We call \( S_p(sig) \) the matching set of \( sig \). Let \( I_{\ell} \) denote the index set of all the signature patterns in \( DB \) such that \( \text{length} \geq \ell \) and \( \text{freq} \geq 2 \). The SSJoin size, \( SSJ(\tau) \), is the same as \( FSP(t) \) which is the number of pairs that match any pattern \( sig_i, i \in I_{t} \), where \( t = \lceil \tau \cdot M \rceil \). As a pair \((r, s)\) can match with multiple signature patterns, we need to compute the cardinality of the union of the set of pairs as follows. Given \( \tau \),

\[
FSP(t) = \left| \bigcup_{i \in I_t} S_p(sig_i) \right| \text{ where } t = \lceil \tau \cdot M \rceil .
\] (5.1)

**Example 4.** Consider Example 3 with \( \tau = 0.5 \) and \( M = 4 \). Every signature pattern with \( \text{freq} \geq 2 \) and \( \text{length} \geq 0.5 \cdot 4 \) contributes to \( FSP(2) \). As shown in Figure 5.2, there are 5 such patterns in \( DB \), we have \( I_2 = \{1, 2, 3, 4, 5\} \) and can compute the size from Equation (5.1) as follows.

\[
SSJ(0.5) = FSP(2) = |S_p(sig_1) \cup \cdots \cup S_p(sig_5)| = 6
\]
5.3. Lattice Counting

<table>
<thead>
<tr>
<th>$\text{sig}_1$</th>
<th>$S_p(\text{sig}_i)$ or $S_i$ (matching set)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[4, 3, X, X]$</td>
<td>${(r_1, r_2), (r_1, r_3), (r_2, r_4)}$</td>
</tr>
<tr>
<td>$[4, X, X, 2]$</td>
<td>${(r_1, r_3)}$</td>
</tr>
<tr>
<td>$[X, 3, X, 2]$</td>
<td>${(r_2, r_4)}$</td>
</tr>
<tr>
<td>$[X, X, 2]$</td>
<td>${(r_1, r_3), (r_1, r_4), (r_3, r_4)}$</td>
</tr>
<tr>
<td>$[4, 3, X, 2]$</td>
<td>${(r_1, r_3)}$</td>
</tr>
</tbody>
</table>

If we ignore the overlap, $FSP(t)$ is $\sum_{i \in T} |S_p(\text{sig}_i)|$. We call this naive strategy the Independent Sum (IS) method. The correct way of computing cardinality of the union is to use the set Inclusion-Exclusion (IE) principle. However, the IE formula has exponential complexity in the number of sets.

5.3.2 The Union Formula Exploiting Lattice

In this section, we describe details on modeling overlaps using lattices. The idea of modeling union with a lattice is the same as the string replacement semi-lattice in Chapter 3, but the actual construction of lattices requires new intuition. This application is possible due to the same underlying idea of the framework; we consider similar groups of possible variants together and model their overlaps with lattice structures. In Chapter 3, groups of similar strings are formed by q-grams with wildcards, and in this section, groups of similar sets are formed by signature patterns. We identify necessary properties of lattices to apply the framework.

Consider an example of $FSP(2)$ which computes $|S_p(\text{sig}_1) \cup S_p(\text{sig}_2) \cup S_p(\text{sig}_4)|$. Let us use $S_i$ to denote $S_p(\text{sig}_i)$ for simplicity. Using the IE principle, $FSP(2)$ is computed in the following way:

$$|S_1| + |S_2| + |S_4| - (|S_1 \cap S_2| + |S_1 \cap S_4| + |S_2 \cap S_4|)$$

$$+ |S_1 \cap S_2 \cap S_4| = 3 + 1 + 3 - (1 + 1 + 1) + 1 = 5.$$

Interestingly, if we look into all the intersections, we see that many of them result in the same set: $S_1 \cap S_2 = S_1 \cap S_4 = S_2 \cap S_4 = \{(r_1, r_3)\}$.

We can structure this overlap with the semi-lattice shown in Figure 5.3. The patterns and their matching sets are organized using lattices (or semi-lattices). The edges represent the inclusion relationship. The level of a pattern is the number of non-X values and the level of a matching set is the level of the corresponding pattern. For instance, $S_5$ has edges to $S_1$ and $S_2$ and $S_1 \cap S_2 = S_5$. $S_5$ has three children and an intersection between any two children will result in $S_5$. This implies that $S_5$ appears three times in the intersections between two sets in the IE formula since there are three ways of choosing two children out of three. Likewise, it appears once in the intersections among three sets, $S_1 \cap S_2 \cap S_4$. The net effect is that $S_5$’s contribution to the IE formula is $|S_5|$ multiplied by $(-3 + 1)$. The negative sign of $-3$ is from the alternating signs in the IE formula. Incorporating this observation, the above IE formula is simplified as follows without actually performing any intersections.

$$|S_1| + |S_2| + |S_2| + (-3 + 1)|S_5| = 3 + 1 + 3 - 2 \cdot 1 = 5.$$

Exploiting the lattice structure, each count $(|S_p(\text{sig}_i)|)$ is processed exactly once whereas the IE method computes every intersection unaware of the underlying structure. Notice that
5.3. Lattice Counting

Figure 5.3: Overlapping Relationship

<table>
<thead>
<tr>
<th>Figure 5.4: Example Pattern Lattice Structures</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Image" /></td>
</tr>
<tr>
<td><img src="image2" alt="Image" /></td>
</tr>
<tr>
<td><img src="image3" alt="Image" /></td>
</tr>
</tbody>
</table>

this approach only involves individual quantities, $|S_p(sig)|$, which can be acquired by pattern mining through $|S_p(sig)| = n_{pair}(freq(sig)) = \binom{freq(sig)}{2}$. From this example, we make the following crucial observations.

- Signature patterns and their matching sets can be organized with the lattice structure. A node in the pattern lattice represents a signature pattern and a node in the matching set lattice represents the set of matching pairs for each pattern.

- The $FSP(t)$ problem can be solved by finding the cardinality of the union of matching sets at level $t$. In Figure 5.3, $FSP(2)$ is the cardinality of the union of matching sets at level 2, $|S_1 \cup S_2 \cup S_4|$.

- The multiplicity of a set in the IE formula is expressed by a coefficient. The coefficient only depends on the lattice structure. For instance, in the above example, $|S_5|$ has a coefficient of $-2$.

Let us generalize this intuition. The pattern lattice and the matching set lattice are isomorphic to a power set lattice, $2^{\{1, \ldots, M\}}$. Figure 5.4 shows the lattice structures for the database in Figure 5.1. Figure 5.4 (a) depicts the underlying power set lattice, $2^{\{1, \ldots, 4\}}$. Figure 5.4 (b) shows the pattern lattice. The pattern lattice needs some modifications; a pattern in the pattern lattice is a generalization of a signature pattern where ‘.’ denotes matching positions and ‘X’ denotes don’t care positions. Figure 5.4 (c) is the matching set lattice corresponding to the pattern lattice for the database in Figure 5.1. A node in the matching set lattice represents the set of pairs whose signatures match at the positions marked ‘.’ in the corresponding pattern. The specific matching values are not important. For instance, [., ., X, X] is a pattern selecting all pairs which match at their first and second
5.3. Lattice Counting

positions, and \{(r_1, r_2), (r_1, r_3), (r_2, r_3)\} are the corresponding matching sets. If there were 
\(r_6, r_7\) such that \(\text{sig}(r_6) = [1, 1, 6, 2]\) and \(\text{sig}(r_7) = [1, 1, 2, 4]\) then \((r_6, r_7)\) would have been in 
the matching set as well.

Consider the coefficient of node \([\_ \_ \_ X]_0\) for \(F_{SP}(2)\). The sublattice structure whose top 
is \([\_ \_ \_ X]_0\) is the same as the lattice in Figure 5.3 (a), and thus we can infer it has the same 
coefficient. In other words, it appears the same number of times in the IE formula for union 
cardinality of sets at level 2. Moreover, observe that all the other nodes at level 3 have the 
identical structure, which gives the same coefficient. We can see that all nodes at the same 
level will have the same coefficient since there will be the same number of ways to choose a 
certain number of nodes at a level. Thus, we can work with the sum of the cardinalities 
of matching sets at each level since they all have the same coefficient. Let the sum at level \(\ell\) be 
\(F_\ell\). We call it the level sum. Define the index set \(I_\ell\) that lists all patterns at level \(\ell\), then 
\[
F_\ell = \sum_{i \in I_\ell} |S_p(p_i)|. \tag{5.2}
\]

Using the level sum, the computation is as follows.

\[
F_2 = |S_p(sig_1)| + |S_p(sig_2)| + |S_p(sig_3)| + |S_p(sig_4)| \\
F_3 = |S_p(sig_5)| \\
F_{SP}(2) = C_2 F_2 + C_3 F_3 = 1 \cdot (3 + 1 + 1 + 3) + (-2) \cdot 1 = 6
\]

Formalizing this intuition, we define the pattern lattice structure and the Lattice Counting 
(LC) problem as follows.

**Definition 7 (Pattern Lattice Structure).** Given a set collection of sets \(R\) and the signature 
size \(M\), the pattern lattice structure is a tuple \(L = (M, \mathcal{L}_P, \mathcal{L}_R)\) where \(M = \{1, ..., M\}\), \(\mathcal{L}_P\) 
is the pattern lattice and \(\mathcal{L}_R\) is the matching set lattice. For each \(m \subset M\), the corresponding 
\(p \in \mathcal{L}_P\) is a signature pattern such that \(p\) has `'` in the position \(i \in m\) and 'X' in the other 
positions. For each \(p \in \mathcal{L}_P\), \(S_p(p) \in \mathcal{L}_R\) is the set of pairs \((r, s), r, s \in R\) such that \(\text{sig}(r)\) and 
\(\text{sig}(s)\) have the same values at position marked `'` in \(p\).

The partial order \(\leq\), the least upper bound \(\lor\), and the greatest lower bound \(\land\) of \(x, y \in \mathcal{L}_P\) 
are defined by \(\subseteq, \cup\) and \(\cap\) between the corresponding subsets in the power set respectively. 
For any \(x, y, z \in \mathcal{L}_P\) the following conditions hold.

- **Inclusion:** \(x \leq y\) iff \(S_p(x) \supseteq S_p(y)\)
- **Refinement:** \(x \lor y = z\) iff \(S_p(x) \cap S_p(y) = S_p(z)\)

**Definition 8 (LC Problem).** Given \(L = (M, \mathcal{L}_P, \mathcal{L}_R)\) and a level threshold \(t\), the LC problem 
is to estimate the cardinality of union of matching sets at level \(t\) in \(\mathcal{L}_R\).

\[
LC(t) = | \bigcup_{i \in I_t} S_p(p_i) |
\]
5.3. Lattice Counting

We can think of the \( LC(t) \) problem as the \( FSP(t) \) problem defined on the lattice. We simplify the IE formula as the sum of the cardinality of each matching set \( S \in L_{DB} \) multiplied by some coefficient. As all the matching sets at the same level have the same coefficient, we can compute \( LC(t) \) using \( \sum_{l=t}^{M} C_{\ell,t} \cdot F_{\ell} \). We give the exact answer for \( LC(t) \) for an arbitrary pattern lattice structure in the next equation.

\[
LC(t) = \sum_{l=t}^{M} C_{\ell,t} \cdot F_{\ell}, \text{ where}
\]

\[
C_{\ell,t} = \sum_{r=2}^{l} (-1)^{r+1} B_{\ell,t,r}
\]

\[
B_{\ell,t,r} = \sum_{i=\ell-t}^{\ell} (-1)^{i} \binom{i}{t} \cdot \binom{\ell-i}{r}
\]

The coefficient \( C \) can be computed by counting how many times a node appears at all intersections of size \( r \geq 2 \) in the IE formula. \( B \) keeps track of the occurrence at each \( r \). The intermediate derivation depends on the inclusion and the refinement properties.

5.3.3 Level Sum Computation

Note that in Equation (5.3), only \( F_{\ell} \) depends on the data. So computing \( LC(t) \) reduces to computing the level sums \( F_{\ell}, t \leq \ell \leq M \). \( F_{\ell} \) is counted as follows.

\[
F_{\ell} = \sum_{i \in I_{\ell}} |S_{p}(p_{i})| = \sum_{i \in I_{\ell}} npair(freq(p_{i}))
\]

(5.4)

The right hand side of Equation (5.4) uses the frequencies of all the signature patterns of length \( \ell \), and \( I_{\ell} \) lists all such patterns. We can simplify the \( F_{\ell} \) computation by noting that patterns with the same \( freq \) value will have the same \( npair(freq(p_{i})) \) value. Rather than repeatedly evaluating the \( npair \) function for the same frequency, a better strategy will be to count how many patterns have the frequency of \( f \) for each \( f \) and to evaluate \( npair \) just once for each \( f \). As there are a large number of patterns at low frequencies this reduction could be huge. Let us group all patterns \( p_{i}, i \in I_{\ell} \) by their frequencies. Let \( I_{\ell,f} \) denote the index set for the patterns such that \( |p_{i}| = l \) and \( freq(p_{i}) = f \) if and only if \( i \in I_{\ell,f} \). For instance, \( I_{2,3} \) lists all the length-2 patterns with frequency of 3. In the example \( I_{2,3} = \{1,4\} \) since \( |sig_{1}| = |sig_{4}| = 2 \) and \( freq(sig_{1}) = freq(sig_{4}) = 3 \) (see Figure 5.2). \( I_{\ell,f} \) defines a partitioning over \( I_{\ell} \) and \( |I_{\ell,f}| \) is the number of patterns with \( freq = f \) in the level \( \ell \). If we use \( mf(I_{\ell}) \) for \( max(freq(x_{i})), i \in I_{\ell} \),

\[
F_{\ell} = \sum_{2 \leq f \leq mf(I_{\ell})} npair(f) \cdot |I_{\ell,f}|
\]

(5.5)

A frequent pattern mining algorithm gives all the frequent patterns above the minimum support threshold. We extract the count information grouped first by pattern length and then by pattern frequency. This gives us \( |I_{\ell,f}| \) for all \( l \) and \( f \) found in the database. For a specific \( \ell \), we call the set of tuples \((f, |I_{\ell,f}|)\) pattern distribution for level \( \ell \).
Example 5. The next table is the signature pattern distribution for the running example.

| level \( \ell \) | \( f \) (freq) | \( |I_{\ell,f}| \) | sig. pattern | matching sets |
|-----------------|----------------|----------------|-------------|---------------|
| 2               | 2              | 2              | \[4, X, X, 2\] | \( r_1, r_3 \) |
|                 |                |                | \[X, 3, 3, X\] | \( r_2, r_4 \) |
| 3               | 2              | 1              | \[4, 3, X, X\] | \( r_1, r_2, r_3 \) |
|                 |                |                | \[X, 3, X, 2\] | \( r_1, r_3, r_4 \) |

\( |I_{2,2}| = 2 \) since two signature patterns, \([4, X, X, 2]\) and \([X, 3, 3, X]\), have two matching sets \( (f = 2) \). Likewise \( |I_{2,3}| = 2 \) since two signature patterns have three matching sets \((f = 3)\). For \( F_2 \), each of the two patterns with \( f = 2 \) generates one pair \((\binom{2}{2} \cdot 2 = 2)\), and each of the two patterns with \( f = 3 \) generates three patterns \((\binom{3}{2} \cdot 2 = 6)\). So \( F_2 = 2 + 6 = 8 \). The whole level sum computation is as follows.

\[
F_2 = \sum_{2 \leq f \leq 3} n_{pair}(f) \cdot |I_{\ell,f}| = \left(\binom{2}{2}\right) \cdot 2 + \left(\binom{3}{2}\right) \cdot 2 = 8
\]

\[
F_3 = \sum_{2 \leq f \leq 2} n_{pair}(f) \cdot |I_{\ell,f}| = \left(\binom{2}{2}\right) \cdot 1 = 1
\]

After computing the level sums, the rest of the LC computation is summing those level sums multiplied by coefficients in Equation (5.3). \( SSJ(0.5) = LC(2) = 6 \) is computed as follows.

<table>
<thead>
<tr>
<th>level ( \ell )</th>
<th>( F_\ell )</th>
<th>( C_4,\ell \cdot F_\ell )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>( F_2 = 8 )</td>
<td>1 ( \times 8 = 8 )</td>
</tr>
<tr>
<td>3</td>
<td>( F_3 = 1 )</td>
<td>-2 ( -2 \times 1 = -2 )</td>
</tr>
<tr>
<td>4</td>
<td>( F_4 = 0 )</td>
<td>3 ( 0 \times 3 = 0 )</td>
</tr>
</tbody>
</table>

\( LC(2) = \sum_{\ell=2}^{4} C_4,\ell \cdot F_\ell = 6 \)

5.4 Power-Law Based Estimation

Note from Equation (5.5) that we only need the frequency of patterns not the actual patterns for our estimation purposes. Thus, our framework only counts patterns and does not exactly generate and store patterns. Equation (5.5) also requires the distribution of all signature patterns with \( \text{freq} \geq 2 \). Most frequent pattern mining algorithms are not designed to handle such a low support threshold. Even if they could, it would take too long to be used for the query optimization task. In this section, we address this issue and show how we can efficiently compute Equation (5.5).

5.4.1 Level Sum with Power-Law Distribution

Chung et al. recently discovered that a Power-law relationship is found in the pattern support distribution [32]. A Power-law is a special relationship between two quantities. It is found in many fields of natural and man-made worlds. A quantity \( x \) obeys a Power-law if it is drawn from a probability distribution \( p(x) \propto x^\alpha, \alpha < 0 \) [33]. The Zipf distribution is one of
5.4. Power-Law Based Estimation

related discrete Power-law probability distributions and it characterizes the “frequency-count” relationship [32]. When $c_i$ is the count of distinct entities that appear $f_i$ times in the data set, the distribution is described by

$$c_i = f_i^\alpha \cdot 10^\beta. \quad (5.6)$$

Chung et al.’s hypothesis refers to the distribution of the pattern count versus the pattern frequency.

Figure 5.5: Signature Pattern Distribution

Figure 5.5 plots the signature pattern distributions of level 3 and 9 in the DBLP data set when $M = 10$. The x-axis is pattern frequency and y-axis is pattern count. For instance, $x=10$, $y=121$ for level 3 in the plot means there are 121 length-3 patterns that appear 10 times in the DB. The pattern count (y-value) decreases as the frequency (x-value) increases; a majority of patterns appear only in a small number of sets and a few patterns appear in many sets. We can also observe that the points of a higher level (e.g., level 9) distribution are on the lower side of points of a low level (e.g., level 3) distribution. This is because when a pattern is frequent, its sub-patterns are at least as frequent.

This distribution corresponds to $(f, |I_{\ell,f}|)$ for each level $\ell$. If we have this information for all $f \geq 2$ and $\ell$, we have the exact level sums by Equation (5.5) and thus the exact answer for the SSJ problem. Consider a practical minimum support threshold of $\xi > 2$, e.g., $\xi = 10$ in Figure 5.5. It is shown as the vertical line in Figure 5.5. The problem of having $\xi > 2$ is that we will miss the pattern distribution on the left side of the vertical line, i.e., patterns with $2 \leq freq < \xi$, because the frequent pattern mining algorithm will only find patterns with $freq \geq \xi$. For patterns at level 3, we see the pattern distribution on the left of the $\xi$ line is missing.

We address the problem of the missing pattern distribution with the Power-law distribution. In accordance with Chung et al.’s hypothesis, a linear relationship between the pattern count and the frequency is observed in Figure 5.5. Exploiting this relationship, the strategy is as follows.

1. Find frequent patterns with $\xi > 2$ for efficiency.
Algorithm 4 Power-Law Estimation

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>build ( \text{sig}(DB) ) by computing ( \text{sig}(s) ) for all ( s \in DB )</td>
</tr>
<tr>
<td>2</td>
<td>run a frequent pattern mining algorithm on ( \text{sig}(DB) ) with ( \xi )</td>
</tr>
<tr>
<td>3</td>
<td>for all ( 1 \leq \ell \leq M ) do</td>
</tr>
<tr>
<td>4</td>
<td>estimate ( \alpha_{\ell} ) and ( \beta_{\ell} ) (using a linear regression)</td>
</tr>
<tr>
<td>5</td>
<td>end for</td>
</tr>
<tr>
<td>6</td>
<td>( t = \lceil \tau \cdot M \rceil )</td>
</tr>
<tr>
<td>7</td>
<td>Est = 0</td>
</tr>
<tr>
<td>8</td>
<td>for all ( t \leq \ell \leq M ) do</td>
</tr>
<tr>
<td>9</td>
<td>( F_{\ell} = \sum_{2 \leq f \leq m_{f}(X_{\ell})} \text{npair}(f) \cdot f^{\alpha_{\ell}} \cdot 10^{\beta_{\ell}} )</td>
</tr>
<tr>
<td>10</td>
<td>Est += ( C_{\ell, t} \cdot F_{\ell} )</td>
</tr>
<tr>
<td>11</td>
<td>end for</td>
</tr>
<tr>
<td>12</td>
<td>return ( \text{Est} )</td>
</tr>
</tbody>
</table>

2. Estimate the parameters for the Power-law distribution at each level with the acquired patterns.

3. Compute the necessary level sums with the pattern distribution based on the estimated parameters.

Our algorithm is independent of the specific choice of the frequent pattern mining algorithm. This gives an opportunity to exploit the extensive studies in the literature.

Algorithm 4 outlines this approach. At line 4, we use linear regression based on the least square fitting to estimate the parameters of the Power-law distribution \( \alpha_{\ell} \) and \( \beta_{\ell} \) for each level \( \ell \). Other methods such as maximum likelihood estimation could be more reliable, but we use linear regression as in [32] since it is very efficient and is sufficient for our purposes. See [32, 33] for more discussions on this topic.

The next challenge is that we may not have enough points for estimating the parameters when we use a high \( \xi \). This is likely for the distribution at higher levels since patterns at higher levels (longer patterns) are fewer than the ones at lower levels. For instance, we cannot estimate the distribution at level 9 in Figure 5.5 since most of the points are on the left side of the \( \xi \) line and we cannot estimate the parameters. Equation (5.3) requires all the level sum \( F_{\ell}, t \leq \ell \leq M \). For example, when \( M = 10 \), \( \text{LC}(5) \) needs \( F_{5}, F_{6}, ..., F_{10} \). We present our solution below.

5.4.2 Approximate Lattice Counting

To compute \( \text{LC}(t) \), we use the pattern lattice structure to consider all the intersections among the patterns at level \( t \). In the lattice, nodes at a higher level represent more complex relationships among nodes at level \( t \). Observe that the counts of those complex intersections are relatively small as we can infer from Figure 5.5; the pattern counts of higher levels are much smaller than those of lower levels. Based on this intuition, we employ a truncation heuristic which considers only parts of the lattice ignoring some of the complex intersections in the IE formula. Consider \{1, 2, 3, 4\} in Figure 5.6 (a). It represents complex intersections such
5.4. Power-Law Based Estimation

Figure 5.6: Relaxed Lattice

as \( \{1, 2\} \cap \{1, 3\} \cap \{3, 4\} \). If we ignore such higher level intersections, we have a part of the original lattice as in Figure 5.6 (b) where we only consider intersections between two nodes. Intersections like \( \{1, 2\} \cap \{1, 3\} \cap \{3, 4\} \) are ignored in the relaxed lattice structure. Generalizing this intuition, we define Max-k relaxed lattice where given \( LC(t) \), we only consider nodes between level \( t \) and \( \min(M, t+k) \) ignoring more complicated intersections at level \( t+k+1 \) or higher. This gives us the relaxed form of Equation (5.3).

\[
LC_k(t) = \sum_{\ell=t}^{K} \hat{C}_{\ell,t}(k) \cdot F_\ell, \quad \text{where} \quad (5.7)
\]

\[
\hat{C}_{\ell,t}(k) = \sum_{r=2}^{K-t} (-1)^{r+1} B_{\ell,t,r}
\]

\( K = \min(M, t+k) \)

Notice the replacement of \( M \) by \( K \) in the summations. We now only sum up to the level \( t+k \). This approximate Union Counting formula lets us estimate the SSJoin size when not all the signature lines are available. With \( LC_k(t) \) where \( k \) is the relaxing parameter, we only need \( F_\ell \) for \( t \leq \ell \leq t+k \). For instance, when \( M = 10 \), \( LC_2(5) \) can be computed with only \( F_5, F_6 \) and \( F_7 \) without \( F_8, F_9 \) and \( F_{10} \).

In Section 3.4, we developed a technique for approximating coefficients with a replacement semi-lattice. The goal was to avoid computing coefficients on-the-fly and to use the formula when a lattice has a similar, but not identical, shape to a replacement semi-lattice. Counts of all the nodes in the lattice are necessary and only their coefficients are approximated. The goal of approximation in this section is to compute the formula without high level nodes. The lattice itself is already in the same shape as a replacement semi-lattice, so the approximation in Section 3.4 cannot be applied.

5.4.3 Estimation With Limited Pattern Distribution

Depending on the available pattern distributions and the similarity threshold, the approximation formula may not be enough to answer \( SSJ(t) \). Assume that signature size \( M = 10 \) and we acquire pattern distributions up to level 6: \( F_0, \ldots, F_6 \). Using the Max-2 relaxed lattice, we can compute \( LC_2(4) \) since it only needs \( F_3, F_4 \) and \( F_5 \). However, we still cannot answer \( LC(t), t \geq 7 \) since we do not have \( F_\ell, \ell \geq 7 \). Recall that the value of \( t \) corresponds to the Jaccard similarity threshold. For instance, when \( M = 10, t = 7 \) corresponds to \( \tau = 0.7 \). Thus, we still cannot handle a threshold such as 0.7 or higher. For this situation, we make use of a new observation found in a skewed distribution.
5.5 Correction of The Estimation

5.5.1 Systematic Overestimation By Min-Hash

While Min-Hash signatures present several benefits, they can cause overestimation. Figure 5.8 shows the SSJoin size by the original sets and by their Min-Hash signatures in the DBLP data set. We see that the overestimation by Min-Hash is rather huge. Figure 5.9 log-plots the true SSJoin size and the size by Min-Hashing. A clear distribution shift is observed. This is problematic since similarity thresholds between 0.5 and 0.9 are typically used [13]. One may suspect that the hash collision is the problem. However, through experiments, we found that this is not the dominating cause as long as the hash domain is not too small. We study the cause of this systematic overestimation and propose a correction procedure below.

5.5.2 Error Correction By State Transition Model

Recall that \( T(i) \) denotes the true number of pairs \((r, s)\) such that \( J_S(r, s) = i/M \). When \( J_S(r, s) = i/M \), we call the pair is in state \( i \) assuming the signature size \( M \) is fixed. We define
5.5. Correction of The Estimation

Figure 5.8: SSJoin Size Distribution in the DBLP data

<table>
<thead>
<tr>
<th>$J_S$</th>
<th>True # pairs</th>
<th># pairs by Min-Hash</th>
<th>Relative Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3,167,244,255</td>
<td>3,167,244,255</td>
<td>0.00%</td>
</tr>
<tr>
<td>0.1</td>
<td>22,750,745</td>
<td>306,062,044</td>
<td>1245.28%</td>
</tr>
<tr>
<td>0.2</td>
<td>577,313</td>
<td>51,556,984</td>
<td>8830.51%</td>
</tr>
<tr>
<td>0.3</td>
<td>128,078</td>
<td>6,470,509</td>
<td>4952.01%</td>
</tr>
<tr>
<td>0.4</td>
<td>5,634</td>
<td>587,761</td>
<td>10332.39%</td>
</tr>
<tr>
<td>0.5</td>
<td>2,049</td>
<td>6,470,509</td>
<td>2614.64%</td>
</tr>
<tr>
<td>0.6</td>
<td>980</td>
<td>6,597</td>
<td>573.16%</td>
</tr>
<tr>
<td>0.7</td>
<td>495</td>
<td>1,477</td>
<td>198.38%</td>
</tr>
<tr>
<td>0.8</td>
<td>384</td>
<td>645</td>
<td>67.97%</td>
</tr>
<tr>
<td>0.9</td>
<td>298</td>
<td>419</td>
<td>40.60%</td>
</tr>
<tr>
<td>1</td>
<td>286</td>
<td>318</td>
<td>11.19%</td>
</tr>
</tbody>
</table>

Figure 5.9: SSJoin size: True vs. MinHash

$O(i)$ to be the number of observed pairs $(r, s)$ such that $J_S(r, s) = i/M$. Let us define two vectors $T$ and $O$ as $T = [T(0), ..., T(M)]^T$ and $O = [O(0), ..., O(M)]^T$. The state transition of a pair from state $i$ to state $j$ means that the pair’s true similarity is $i/M$ but is estimated to be $j/M$. For example, if $M = 4$ and there are 100 pairs such that $J_S = 2/4$, we say that 100 pairs are in state 2. If there is a 30% chance of state transition from state 2 to 3, 30 of them are falsely estimated to have $J_S = 3/4$. The key point is that when the distribution of $T$ is skewed, it can cause overestimation in $O$.

Figure 5.10: An Example of Imbalance in Transitions

Example 6. Assume $M = 4$ and the distribution of $T(i)$ is as in Figure 5.10. For instance, 100 pairs have $J_S = 2/4$ and they are in state 2. There exist state transitions since Min-Hash is probabilistic. In the example of Figure 5.1, $J_S(r_3, r_4) = 1/5$ so the pair’s true state is 1 (rounding up 1/5 to 1/4). However with the signatures, it is estimated as $\hat{J}_S(r_3, r_4) = 2/4$ and is in state 2. Suppose there is a 10% of state transition probability to each neighboring
5.5. Correction of The Estimation

state. If the distribution of \( T(i) \) is skewed, it causes unbalance in the transitions. For instance, although \( T(2) = 100 \), \( O(2) = 181 \) since 20 pairs go to state 1 or 3, and 100 pairs come from state 1, and 1 pair comes from state 3.

Given a pair \((r,s)\) such that \( J_S(r,s) = i/M \), let \( I \) denote its state \( i \), and \( J \) denote the actual number of matching positions in \( \text{sig}(r) \) and \( \text{sig}(s) \). At each position, the matching probability \( Pr(\pi_k(r) = \pi_k(s)) \) is \( i/M \) and all \( \pi_k \) are independent. Thus \( J \) is a binomial random variable with parameter \( i/M \). That is

\[
Pr(J = j | I = i) = \binom{M}{j} \left( \frac{i}{M} \right)^j \left( 1 - \frac{i}{M} \right)^{M-j}.
\]

(5.8)

If we use \( P(j|i) \) for \( Pr(J = j | I = i) \), \( Pr(j|i) \) is the probability of having \( j \) matching positions when the true number of matching positions is \( i \). The observed count \( O(j) \) is the sum of all \( T(i) \) multiplied by \( P(j|i) \) which is moving in mass from state \( i \). Thus \( \sum_{i=0}^{M} Pr(j|i) T(i) = O(j) \). This gives us a system of equations for \( 0 \leq j \leq M \). Define the matrix \( A \) as \( A_{ji} = Pr(j|i) \).

The system is described by

\[
AT = O.
\]

(5.9)

**Example 7.** When \( M = 4 \), based on Equation (5.8), \( A \) is as follows.

\[
A = \begin{pmatrix}
P(0|0) & P(0|1) & P(0|2) & P(0|3) & P(0|4) \\
P(1|0) & P(1|1) & P(1|2) & P(1|3) & P(1|4) \\
P(2|0) & P(2|1) & P(2|2) & P(2|3) & P(2|4) \\
P(3|0) & P(3|1) & P(3|2) & P(3|3) & P(3|4) \\
P(4|0) & P(4|1) & P(4|2) & P(4|3) & P(4|4)
\end{pmatrix}
\]

Then by Equation (5.9), \( AT = O \) becomes

\[
\begin{pmatrix}
1.000 & 0.316 & 0.063 & 0.004 & 0 \\
0 & 0.422 & 0.250 & 0.047 & 0 \\
0 & 0.211 & 0.375 & 0.211 & 0 \\
0 & 0.047 & 0.250 & 0.422 & 0 \\
0 & 0.004 & 0.063 & 0.316 & 1.000
\end{pmatrix}
\begin{pmatrix}
T(0) \\
T(1) \\
T(2) \\
T(3) \\
T(4)
\end{pmatrix}
= \begin{pmatrix}
O(0) \\
O(1) \\
O(2) \\
O(3) \\
O(4)
\end{pmatrix}.
\]

Let us analyze \( O(3) \). \( 0 \cdot T(0) + 0.047 \cdot T(1) + 0.25 \cdot T(2) + 0.422 \cdot T(3) + 0 \cdot T(4) = O(3) \). 4.7\% of \( T(1) \) contributes to \( O(3) \). 25\% of \( T(2) \) goes to \( O(3) \) and 42.2\% of \( T(3) \) goes to \( O(3) \).

The huge incoming counts from \( T(i) \) to \( O(j), i < j \) can inflate \( O(j) \). Since \( \text{LC}(t) = \sum_{i=t}^{M} O(i) \), without appropriate correction, \( \text{LC}(t) \) is doomed to be a massive overestimation. A tempting option is to compute \( T \) by solving the system of linear equations. \( A \) is non-singular and only depends on the choice of \( M \). The join size is computed by \( \hat{T} = A^{-1}O \).

However, this simple approach does not work for two reasons. First, \( \hat{T} \) might have negative values. Second, \( O \) is highly skewed: \( O(i) \gg O(j) \) when \( i < j \). So lower entries in \( O \), say \( O(0), O(1) \) and \( O(2) \), will have a dominating effect making higher entries negligible.

Thus, rather than solving the system of linear equations, we solve a NNLS (non-negative least square) constrained optimization problem [88]. To prevent the solution from being
5.6 Experimental Evaluation

Algorithm 5 LC with Error Correction

Procedure LCWithErrorCorrection

Input: similarity threshold \( t \), max iteration count \( \theta \)

Output: \( LC(t) \)

1: Compute \( LC(\ell) \) for \( 1 \leq \ell \leq M \) (estimates without correction)
2: \( O(0) = N \cdot (N - 1)/2, O(M) = LC(M) \) // \( N \): data set size
3: \( O(\ell) = LC(\ell) - LC(\ell + 1), 1 \leq \ell < M \)
4: SanityCheck(\( LC(\ell), O(\ell), 1 \leq \ell \leq M \)) // desc. in Sec 7.1
5: PowerLawInterpolate(\( O \))
6: minimize \( ||WAX - WO|| \) with \( \theta \), subject to \( X \geq 0 \)
7: PowerLawInterpolate(\( X \))
8: return \( LC(t) = \sum_{\ell=t}^{M} X(\ell) \)

Procedure PowerLawInterpolate

Input: vector \( V = [V_1, ..., V_k] \)

Output: modified \( V = [V_1, ..., V_k] \)

1: Estimate Power-law parameter \( \alpha \) and \( \beta \) from \( (\ell, V_\ell) \)
2: for all \( V(\ell) \leq 0 \) do
3: \( V(\ell) = \ell^\alpha \cdot 10^\beta \)
4: end for

dominated by lower entries, we scale the matrix by a weight matrix \( W \) so that higher entries in \( O \) will have approximately the same effect in the least square solution. We could use \( W \) defined as \( W_{i,j} = 1/O(i) \) and \( W_{i,j} = 0, i \neq j \). In summary, the final step of our estimation corrects the estimation by solving the next NNLS problem.

\[
\text{minimize} \quad ||WAX - WO|| \quad \text{subject to} \quad X \geq 0 \quad (5.10)
\]

The dimension of the matrix of interest is not big, say 10, and we can solve the system fairly efficiently. (i.e., milliseconds)

The NNLS may fail for several reasons. It may diverge or have zero estimates in \( X \). We use the technique in Section 5.4.3 in these cases. Missing \( T \) or \( X \) values are interpolated using the found values. Algorithm 5 outlines the Lattice Counting with correction. It applies the NNLS correction step at line 6. Missing \( O \) or \( X \) values are estimated using the Power-law hypothesis in Section 5.4.3.

5.6 Experimental Evaluation

5.6.1 Experimental Setup

Data sets: We have conducted experiments using two types of data sets: real-life data and synthetic data. The DBLP data set is used as the real-life data. It is built as described in [8]. The DBLP data set consists of 794,061 sets. We call this full data set the 800K data set. Note that this full data set corresponds to more than 300 billion pairs! We also use part of this data set to show scalability. In particular, we use 40K, 80K, 160K, 240K and 400K randomly selected subsets. The average set size is 14, and the smallest is 3 and the biggest is
5.6. Experimental Evaluation

The synthetic data set is generated using the IBM Quest synthetic data generator [69]. The data set contains 50,000 sets and the universe of sets is 10,000. We varied the average set size from 15 to 50. The pattern correlation parameter is set to 0.25.

**Algorithms compared:** We implemented the following algorithms for the SSJoin size estimation.

- **LC(\(\xi\))**: It is Lattice Counting with the approximation in Section 5.4.2. \(\xi\) gives the minimum support threshold and we use a value between 0.015\% and 0.10\%. The estimation is corrected with Algorithm 5. The exact LC is not used due to its long pattern mining time.

- **LCNC(\(\xi\))**: It is identical to LC(\(\xi\)) except that the error correction procedure, Algorithm 5, is not applied after the initial estimation.

- **IS**: This estimation does not rely on the IE principle and Lattice Counting; it only relies on Power-law estimation. Given LC(\(t\)), it assumes that all signature patterns are independent and, sums \(F_\ell\)'s, \(\ell \geq t\).

- **HS(\(\rho\))**: This is an adaptation of Hashed Sampling [61] to the SSJ problem. \(\rho\) gives the sampling ratio.

A signature size of 10 is used with a hash space of \(2^{15}\). For LC and LCNC, we use the relaxing parameter \(k = 2\) as described in Section 5.4.2. Linear regression for the Power-law parameter estimation needs special attention. In general, the tail of the Power-law distribution is not reliable. We used only up to 40 leftmost points. The max iteration count \(\theta\) in Algorithm 5 is set to 100. However, the solution of Equation (5.10) converged without reaching the count limit most of the time. We apply sanity bounds to LC and \(O\) values in line 4 of Algorithm 5. Specifically, we make sure LC(\(\ell\)) is no smaller than LC(\(\ell + 1\)) by setting it to LC(\(\ell + 1\)) if LC(\(\ell\)) < LC(\(\ell + 1\)). Moreover, if LC(\(\ell\)) < 0, we take the geometric mean of LC(\(\ell'\)) and LC(\(\ell''\)) that are the first positive values such that \(\ell' < \ell\) and \(\ell'' > \ell\). In the extreme case of \(O(\ell) = 0\) for \(1 \leq \ell \leq M\), we set \(O(M)\) to 1. For the SSJoin algorithm in HS, we used the ProbeOptMerge algorithm [130], which can be used with Hashed Sampling and Jaccard similarity. We did not implement the clustering version since the improvement was marginal [13, 130].

**Evaluation metric:** We use absolute count and average (absolute) relative error to show the accuracy. A relative error is defined as \(|\text{est\_size} - \text{true\_size}|/|\text{true\_size}|\). We use both measures since the average relative error favors underestimation and can be misleading. That is even if we always answer with 0, the error is capped by -100\%. Counts are shown in log scale as the distribution is highly skewed. Given the nature of random sampling, accuracy figures are given with the median value across three runs. This applies to HS, as well as the subsets of the full DBLP data set.

For efficiency, we measure the runtime, which is divided into pre-processing time and estimation time. HS assumes pre-compiled samples and pre-processing time corresponds to the time for building the sampled inverted index. For LC, LCNC and IS, it is the time necessary for Min-Hash signature generation. The estimation time is the time required for the actual estimation. For HS, it is the SSJoin time on the samples. For LC, LCNC and
5.6. Experimental Evaluation

IS, the time includes the frequent pattern mining time, level sum computation time, as well as the time for Lattice Counting. LC adds the NNLS solving time on top of it. All the times measured include disk I/O’s.

We implemented all the estimation algorithms in Java. We downloaded the NNLS solver package from [62]. For the pattern mining algorithm, we downloaded the trie-based Apriori algorithm from [46], but almost any frequent pattern mining algorithms can be used for our estimation purposes. We ran all our experiments on a desktop PC running Linux Kernel 2.6.22 over a 3.00 GHz Pentium 4 CPU with 1GB of main memory.

5.6.2 DBLP Data Set: Accuracy and Efficiency

We first report the results on the accuracy and runtime using the 40K DBLP data set. Figure 5.11 shows the true and estimated SSJoin sizes of LC(0.1%), LC(0.04%), HS(5%), and HS(2%). HS(5%) and HS(2%) deliver relatively accurate estimates for \( \tau \leq 0.3 \). However, we may be more interested in higher threshold ranges. For \( \tau \geq 0.4 \), both HS(5%) and HS(2%) consistently estimate the join size to be zero, which is not a meaningful estimate. LC(0.1%) performs better than HS(5%) and HS(2%) for \( \tau \) between 0.4 and 0.7. However, the clear winner is LC(0.04%) which is by far the closest to the true size.

Figure 5.12 shows the runtime performance. It is clear that for both estimation time and pre-processing time, LC is at least as efficient as HS. Yet, as shown in Figure 11 and discussed before, LC can be significantly more accurate, particularly for \( \tau \geq 0.4 \).

5.6.3 Effectiveness of Lattice Counting and Error Correction

Figure 5.13 (a) shows the average relative error of IS, LC(0.02%) and LCNC(0.02%) in the 160K DBLP data set. Recall that IS does not apply Lattice Counting and LCNC does not apply the error correction procedure in Section 5.5. A huge overestimation is observed in IS. LCNC is better than IS but still shows a rather large overestimation. LC shows the best accuracy. Its errors are very small especially in the high threshold range. This verifies that the Lattice Counting effectively considers the overlaps in counting and that the error correction step indeed offsets the systematic overestimation by Min-Hashing.
5.6. Experimental Evaluation

Figure 5.12: Performance on the DBLP Data Set

Figure 5.13: Effectiveness of Lattice Counting and the Error Correction

Figure 5.14 compares the accuracy and runtime of $LC(0.02\%)$ and $HS(1\%)$ varying the data set size with $\tau = 0.8$. $HS(1\%)$ does not provide meaningful estimates until the data set size reaches $400K$ where there are sufficiently many highly similar pairs. Its produces zero estimates for size between $40K$ and $160K$.

The estimates of $LC(0.02\%)$ are quite reliable, and are consistently below a relative error of 10. A relative error of 10 may seem big; but it is reasonable considering the high selectivity
5.6. Experimental Evaluation

Figure 5.14: Scalability Using the DBLP Data Set

Figure 5.15: Accuracy on the Synthetic Data Set

of a high threshold such as \( \tau = 0.8 \). For instance, in the 80K data set, there are more than 6 billion pairs and the selectivity of \( \tau = 0.8 \) is about 0.000012% compared to the total number of pairs.

Figure 5.14 (b) and (c) show the pre-processing and estimation time of \( HS(1%) \) and \( LC(0.02%) \). \( LC(0.02%) \) has consistently lower pre-processing time than \( HS(1%) \). We can observe the quadratic increase of estimation time of \( HS(1%) \). \( LC(0.02%) \) also exhibits an increase in the estimation time primarily due to the increase in the frequent pattern mining time. However the increase is rather mild. Chuang et al. proposed a sampling method for computing the pattern distribution \[32\]. Such a technique should make \( LC \) even more scalable.

5.6.5 Synthetic Data Set: Accuracy, Efficiency and Scalability

We now report the results on the synthetic data set. We varied the average set size parameter from 15 to 50. Figure 5.15 (a) and (b) show the accuracy when the average set size is 20 and 25 respectively. In both cases, the estimation of \( LC(0.04%) \) is close to the true SSJoin size. The estimates generated by \( LC(0.1%) \) are not as good as those generated by \( LC(0.04%) \). However, the estimation is still close to the true size and and generates reasonable estimates. Both \( HS(5%) \) and \( HS(10%) \) produce reliable estimates at lower similarity ranges, but fail to create non-trivial estimates at high similarities ranges.
The following table shows how the estimation time (in seconds) of $LC(0.1\%)$ and $HS(5\%)$ varies with average set size. $LC$ is not much affected by increasing the average set size. $LC$ performs analysis on the standardized representation of sets, Min-Hash signatures. Thus, its runtime is almost invariant to the original set size. In contrast, $HS(5\%)$ is affected greatly by increasing the average set size. This is due primarily to the size of the inverted index.

<table>
<thead>
<tr>
<th>avg. set size</th>
<th>15</th>
<th>20</th>
<th>25</th>
<th>30</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>$LC(0.1%)$</td>
<td>1.34</td>
<td>1.39</td>
<td>1.39</td>
<td>1.38</td>
<td>1.34</td>
</tr>
<tr>
<td>$HS(5%)$</td>
<td>0.71</td>
<td>1.17</td>
<td>1.77</td>
<td>2.47</td>
<td>6.43</td>
</tr>
</tbody>
</table>

5.6.6 On Power-law Hypotheses

In this chapter, the key hypothesis is that using simple sampling to estimate SSJoin size is unreliable. A better approach is to rely on Power-laws and to estimate the parameters of Power-laws in a sample-independent fashion. Certainly, one can argue that the minimum support threshold is effectively doing “subsetting”. But we believe that subsetting by the minimum support threshold is a more principled way of selection than sampling is. The experimental results shown so far support this hypothesis.

An interesting question would be if the Power-law really holds in the data set. In Section 5.4, we rely on two Power-law hypotheses. The first one is on the count-frequency relationship in the patterns and the second one is on the number of pairs-similarity relationship. Proving that a distribution really follows a Power-law distribution is not a trivial task. Some of the efforts to quantify the relationship includes the p-value for the Kolmogorov-Smirnov test [33]. However, in our case, the number of points are not sufficient for drawing a conclusion with statistical significance. Moreover, an approximate modeling was enough for our purposes. Therefore we show the log-log plots instead. Figure 5.7 and Figure 5.15 present the number of pairs distribution-similarity distribution and the pattern count-frequency distribution. Our observations agree with the findings in [32], but formally verifying the relationship is an open problem.

5.6.7 Summary of Experiments

We evaluated the proposed solutions, LC, using a real-world data set and a synthetic data set. LC is compared with HS, which is an adaptation of Hashed Samples [61] to the SSJ problem. While random sampling based HS showed poor estimation at high thresholds, LS consistently delivered estimation within relative errors of a few factors. Analysis at each step shows that each step effectively reduces overestimation errors by an order of magnitude.

5.7 Conclusion

We propose an accurate and efficient technique for the SSJoin size estimation. Our technique uses Min-Hash signatures which are succinct representation of sets. We propose a lattice

\[\text{We discuss how we can use random sampling for similarity join size estimation in the following chapter.}\]
based counting technique called Lattice Counting to efficiently count the number of pairs satisfying the similarity threshold. We exploit Power-law distributions to efficiently compute the pattern distribution necessary for the Lattice Counting. A systematic overestimation by relying on Min-Hash signatures is observed and we also propose a procedure to correct it. In the future, we plan to exploit sampling in the frequent pattern mining for the pattern distribution [15, 32].
Chapter 6

Vector Similarity Join Size Estimation

In this chapter, we study the VSJ problem in Section 1.4.3, where a record is a vector. It is a generalization of the SSJ problem in the previous chapter. We first consider the difference between the two problem formulation and the challenges in the following section.

6.1 Introduction

As introduced in the previous chapter, given a similarity measure and a minimum similarity threshold, a similarity join finds all pairs of objects whose similarity is at least as large as the similarity threshold. The object in a similarity join is often a vector. For instance, a document can be represented by a vector of words in the document, or an image can be represented by a vector from its color histogram. In this chapter, we focus on the vector representation of objects and study the VSJ problem in Definition 4.

In the previous chapter, the similarity join size estimation problem has been defined using sets using Definition 3. The formulation of similarity joins with vectors is more general and can handle more practical applications. For instance, while in the SSJ problem a document is simply a set of words in the document, in the VSJ problem a document can be modeled with a vector of words with TF-IDF weights. It can also deal with multiset semantics with occurrences. In fact, most of the studies on similarity joins first formulate the problem with sets and then extend it with TF-IDF weights, which is indeed a vector similarity join.

A straightforward extension of SSJ techniques for the VSJ problem is to embed a vector into a set space. We convert a vector into a set by treating a dimension as an element and repeating the element as many times as the dimension value, using standard rounding techniques if values are not integral [8]. In practice, however, this embedding can have adversary effects in performance, accuracy or required resources [8]. Intuitively, a set is a special case of a binary vector and is not more difficult to handle than the vector. For instance, Bayardo et al. [13] define the vector similarity join problem and add special optimizations that are possible when binary vectors (sets).

In our VSJ problem, we consider cosine similarity as the similarity measure sim since it has been successfully used across several domains [13]. Let \( u[i] \) denote the \( i \)-th dimension value of vector \( u \). Cosine similarity is defined as \( \cos(u, v) = u \cdot v / ||u|| \cdot ||v|| \), where \( u \cdot v = \sum_i u[i] \cdot v[i] \) and \( ||u|| = \sqrt{\sum_i u[i]^2} \). We first focus on self-join size estimation, and discuss non-self joins at Section 6.5.2.

One of the key challenges in similarity join size estimation is that the join size can change
dramatically depending on the input similarity threshold. While the join size can be close to $n^2$ at low thresholds where $n$ is the database size, it can be extremely small at high thresholds. For instance, in the DBLP data set, the join selectivity is only about 0.00001% at $\tau = 0.9$! While many sampling algorithms have been proposed for the (equi-)join size estimation, their guarantees fail in such a high selectivity range, e.g., [49, 58, 96]. Intuitively, it is not practical to apply random sampling in such a high selectivity. This is problematic since similarity thresholds between 0.5 and 0.9 are typically used [13].

In this chapter, we propose sampling based techniques that exploit the Locality Sensitive Hashing (LSH) scheme, which has been successfully applied in similarity searches across many domains. LSH builds hash tables such that similar objects are more likely to be in the same buckets. Our key idea is that although sampling a pair satisfying a high threshold is very difficult, it is relatively easy to sample the pair using the LSH scheme because it groups similar objects together. We show that the proposed algorithm $LSH-SS$ gives good estimates at both high and low thresholds with a sample size of $\Omega(n)$ pairs of vectors (i.e., $\Omega(\sqrt{n})$ tuples from each join relation in an equi-join) with probabilistic guarantees. The proposed solution only needs minimal addition to the existing LSH index and can be easily applied. Thus it is readily applicable to many similarity search applications. As a summary, we make the following contributions:

- We present two baseline methods in Section 6.2. We consider random sampling and adapt Lattice Counting (LC) [92] which is proposed for the SSJ problem.
- We extend the LSH index to support similarity join size estimation in Section 4. We also propose $LSH-S$ which relies on an LSH function analysis.
- We describe a stratified sampling algorithm $LSH-SS$ that exploits the LSH index in Section 6.4. We apply different sampling procedures for the two partitions induced by an LSH index: the pairs of vectors that are in the same buckets and those that are not.
- We compare the proposed solutions with random sampling and LC using real-world data sets in Section 6.6. The experimental results show that $LSH-SS$ is the most accurate with small variance.

6.2 Baseline Methods

6.2.1 Random Sampling

The first baseline method is uniform random sampling. We select $m$ pairs of vectors uniformly at random (with replacement) and count the number of pairs satisfying the similarity threshold $\tau$. We return the count scaled up by $M/m$ where $M$ denotes the total number of pairs of vectors in the database $V$.

We note two challenges in the VSJ problem compared to the equi-join size estimation. In the equi-join of $[R \bowtie S]$, we can focus on frequency distribution on the join column of each relation $R$ and $S$. For instance, if we know a value $v$ appears $n_r(v)$ times in $R$ and $n_s(v)$ times in $S$, the contribution of $v$ in the join size is simply $n_r(v) \cdot n_s(v)$, i.e. multiplication of two
6.3 LSH Index for the VSJ Problem

frequencies. We do not need to compare all the \( n_r(v) \cdot n_s(v) \) pairs. In similarity joins, however, we need to actually compare the pairs to measure similarity. This difficulty invalidates the use of popular auxiliary structures such as indexes [28, 58] or histograms [28].

Furthermore, similarity join size at high thresholds can be much smaller than the join size assumed in equi-joins. For instance, in the DBLP data set \( (n = 800K) \), the join size of \( \Omega(n \log n) \) assumed in bifocal sampling is more than 15M pairs and corresponds to the cosine similarity value of only about 0.4. In the most cases, users will be interested in much smaller join sizes and thus higher thresholds.

6.2.2 Adaptation of Lattice Counting

We proposed Lattice Counting (LC) in the previous chapter to estimate SSJ size with Jaccard similarity. LC performs an analysis on the Min-Hash signatures of all sets. We observe that the analysis of LC is valid as long as the number of matching positions in the signatures of two objects is proportional to their similarity. Note that this requirement is exactly the property of the LSH scheme. Thus LC can be applied for the VSJ problem with an appropriate LSH scheme. In fact, Min-Hashing is the LSH scheme for Jaccard similarity. For the VSJ problem, we first build the signature database by applying an LSH scheme to the vector database and then apply LC.

6.3 LSH Index for the VSJ Problem

We first describe how we extend the LSH index and present a naive method using the extended LSH index with a uniformity assumption. We then present \( LSH-S \) which improves the naive method with simple random sampling.

6.3.1 Preliminary: LSH Indexing

Let \( \mathcal{H} \) be a family of hash functions such that \( h \in \mathcal{H} : \mathbb{R}^d \rightarrow \mathcal{U} \). Consider a function \( h \) that is chosen uniformly at random from \( \mathcal{H} \) and a similarity function \( \text{sim} : \mathbb{R}^d \times \mathbb{R}^d \rightarrow [0, 1] \). The family \( \mathcal{H} \) is called locality sensitive if it satisfies the following property [23].

**Definition 9.** [Locality Sensitive Hashing] For any vectors \( u, v \in \mathbb{R}^d \),

\[
P(h(u) = h(v)) = \text{sim}(u, v).
\]

That is, the more similar a pair of vectors is, the higher the collision (or matching) probability is. The LSH scheme works as follows [23, 70]: For an integer \( k \), we define a function family \( \mathcal{G} = \{ g : \mathbb{R}^d \rightarrow \mathcal{U}^k \} \) such that \( g(v) = (h_1(v), ..., h_k(v)) \), where \( h_i \in \mathcal{H} \), i.e. \( g \) is the concatenation of \( k \) LSH functions. For an integer \( \ell \), we choose \( \ell \) functions \( G = \{ g_1, ..., g_\ell \} \) from \( \mathcal{G} \) independently and uniformly at random. Each \( g_i, 1 \leq i \leq \ell \), effectively constructs a hash table denoted by \( D_{g_i} \). A bucket in \( D_{g_i} \) stores all \( v \in V \) that have the same \( g_i \) values. Since the total number of buckets may be large, we only store existing buckets by resorting to standard hashing. \( G \) defines a collection of \( \ell \) tables \( I_G = \{ D_{g_1}, ..., D_{g_\ell} \} \) and we call it an LSH index.

101
The definition of LSH in Definition 9 is slightly different from the original definition in [70]. However, they are fundamentally the same and many LSH functions satisfy this definition [23].

LSH families have been developed for several (dis)similarity measures including Hamming distance, \( \ell_p \) distance, Jaccard similarity, and cosine similarity [7]. We rely on the LSH scheme proposed by Charikar [23] that supports cosine similarity. The proposed algorithms can easily support other similarity measures by using an appropriate LSH family.

### Extending The LSH Scheme with Bucket Counts

We describe algorithms using a single LSH table: \( g = (h_1, \ldots, h_k) \) and an LSH table \( D_g \). Extensions with multiple LSH tables are in Section 6.5.2. Suppose that \( D_g \) has \( n_g \) buckets; all vectors in the database are hashed into one of the \( n_g \) buckets. We denote a bucket by \( B_j, 1 \leq j \leq n_g \). Given a vector \( v \), \( B(v) \) denotes the bucket to which \( v \) belongs. In the LSH table, each bucket \( B_j \) stores the set of vectors that are hashed into \( B_j \). We extend this traditional LSH table by adding a bucket count \( b_j \) for each bucket \( B_j \) that is the number of vectors in the database that are hashed into \( B_j \). The overhead of adding a bucket count to each bucket is not significant compared to other information such as vectors. Depending on implementation, the count may be freely available.

### 6.3.2 Estimation with Uniformity Assumption

Given a collection of vectors \( V \), and a similarity threshold \( \tau \), let \( M \) denote the number of total pairs in \( V \), i.e. \( M = \binom{n}{2} \). Consider a random pair \((u, v)\) with \( u, v \in V, u \neq v \). We denote the event \( \text{sim}(u, v) \geq \tau \) by \( T \), and the event \( \text{sim}(u, v) < \tau \) by \( F \). We call \((u, v)\) a true pair (resp. false pair) if \( \text{sim}(u, v) \geq \tau \) (resp. \( \text{sim}(u, v) < \tau \)). Depending on whether \( u \) and \( v \) are in the same bucket, we denote the event \( B(u) = B(v) \) by \( H \) and the event \( B(u) \neq B(v) \) by \( L \). With these notations, we can define various probabilities. For instance, \( P(T) \) is the probability of sampling a true pair. \( P(H|T) \) is the probability that a true pair is in the same bucket and \( P(T|H) \) is the probability that a pair of vectors from a bucket is a true pair. We use \( N_\mathcal{E} \) to denote the cardinality of the set of pairs that satisfy the condition of event \( \mathcal{E} \). For instance, \( N_T \) is the number of true pairs (the VSJ size \( J \)) and \( N_H \) is the number of pairs in the same buckets. \( N_H \) can be computed by \( N_H = \sum_{j=1}^{n_g} \binom{b_j}{2} \), where \( b_j \) is the number of vectors in bucket \( B_j \) and \( n_g \) is the number of buckets in LSH table \( D_g \).

The key observation is that if we consider all the pairs of vectors in a bucket, they are either true pairs or false pairs. Using Bayes’ rule [111], we can express this observation as follows:

\[
N_H = N_T \cdot P(H|T) + N_F \cdot P(H|F).
\]

That is, the total number of pairs of vectors in the same buckets is the sum of the number of true pairs that are in the same buckets \( (N_T \cdot P(H|T)) \) and the number of false pairs that happened to be in the same buckets \( (N_F \cdot P(H|F)) \). Since \( N_F = M - N_T \) where \( M \) is the total number of pairs in \( V \), rearranging the terms gives \( N_T = (N_H - M \cdot P(H|F))/(P(H|T) - P(H|F)) \). Using \( \hat{\cdot} \) for an estimated quantity, we have an estimator for the join size \( J = N_T \)
6.3. LSH Index for the VSJ Problem

Figure 6.1: Probability Density Functions of (non-) Collision in the LSH Scheme

as follows:

$$\hat{N}_T = \hat{J}_U = \frac{N_H - M \cdot \hat{P}(H|F)}{\hat{P}(H|T) - \hat{P}(H|F)}.$$  \hfill (6.1)

Note that $M = \left(\binom{|V|}{2}\right)$ is a constant given the database $V$ and $N_H$ can be computed using $N_H = \sum_j^{n_g} (b_j^2)$ given $D_g$. The conditional probabilities in Equation (6.1) need to be estimated to compute $\hat{N}_T$. We next present our first estimator that relies on an LSH function analysis and the uniformity assumption to estimate the conditional probabilities.

Consider a random pair $(u, v)$ such that $\text{sim}(u, v)$ is selected from $[0, 1]$ uniformly at random. Recall that when $\text{sim}(u, v) = s$, $P(h(u) = h(v)) = s$ from Definition 9. $P(g(u) = g(v)) = s^k$ since $g$ concatenates $k$ hash values. Figure 6.1(a) shows the collision probability density function (PDF) $f(s) = s^k$ where $s = \text{sim}(u, v)$. The vertical dotted line represents the similarity threshold $\tau$. The darker area on the right side of the $\tau$ line is the good collision probability, i.e. the probability that $B(u) = B(v)$ ($u$ and $v$ are in the same bucket) and $\text{sim}(u, v) \geq \tau$. Thus the area represents $P(H \cap T)$. Likewise, the area on the left side of the $\tau$ line is $P(H \cap F)$, which is the probability that $\text{sim}(u, v) < \tau$, but $B(u) = B(v)$. Notice that the area below $f(s)$ does not equal to 1 since it does not cover the entire event space; $u$ and $v$ may not be in the same bucket. Figure 6.1(b) shows the other case where $u$ and $v$ are in different buckets. Its probability density function is $1 - f(s)$ as shown as the curve. $P(L \cap T)$ and $P(L \cap F)$ are defined similarly as shown in the figure.

Given $g$ (and thus $f$) and $\tau$, $P(H \cap F)$, $P(H \cap T)$, $P(L \cap F)$ and $P(L \cap T)$ can be estimated by computing the corresponding areas in Figure 6.1. Based on these areas, we can estimate the desired conditional probabilities using the following:

$$P(H|T) = \frac{P(H \cap T)}{P(H \cap T) + P(L \cap T)} \hfill (6.2)$$
$$P(H|F) = \frac{P(H \cap F)}{P(H \cap T) + P(L \cap F)} \hfill (6.3)$$
Since \( f(s) = s^k \), the four areas defined in Figure 6.1 can be calculated as follows:

\[
\begin{align*}
P[H \cap F] &= \int_0^\tau f(s)ds = \frac{\tau^{k+1}}{k+1} \\
P[H \cap T] &= \int_\tau^1 f(s)ds = \frac{1 - \tau^{k+1}}{k+1} \\
P[L \cap F] &= \int_0^\tau 1 - f(s)ds = \tau - \frac{\tau^{k+1}}{k+1} \\
P[L \cap T] &= \int_\tau^1 1 - f(s)ds = 1 - \tau - \frac{1 - \tau^{k+1}}{k+1}.
\end{align*}
\]

Using the above probabilities in Equation (6.2) and (6.3) gives,

\[
\begin{align*}
P(H|T) &= \frac{1}{1 - \tau} \int_\tau^1 f(s)ds = \frac{\sum_{i=0}^{k}\tau^i}{k+1} \quad \text{(6.4)} \\
P(H|F) &= \frac{1}{\tau} \int_0^\tau f(s)ds = \frac{\tau^k}{k+1}. \quad \text{(6.5)}
\end{align*}
\]

Using above \( P(H|T) \) and \( P(H|F) \) in Equation (6.1),

\[
\hat{J}_U = \frac{N_H - M \cdot \hat{P}(H|F)}{\hat{P}(H|T) - \hat{P}(H|F)} = \frac{N_H - M \cdot \tau^k}{\sum_{i=0}^{k}\tau^i - \tau^k} = \frac{(k+1)N_H - M \cdot \tau^k}{\sum_{i=0}^{k-1}\tau^i}.
\]

Thus we have the following estimator \( \hat{J}_U \) for the VSJ size:

\[
\hat{J}_U = \frac{(k+1)N_H - \tau^k \cdot M}{\sum_{i=0}^{k-1}\tau^i}. \quad \text{(6.6)}
\]

A drawback with \( \hat{J}_U \) is that it assumes that the similarity of pairs is uniformly distributed in [0, 1]. However, this distribution is generally highly skewed [92]; most of pairs have low similarity values and only a small number of pairs have high similarity values. We next present \textit{LSH-S} that removes the uniformity assumption with sampling.

### 6.3.3 LSH-S: Removing Uniformity Assumption

We consider two methods for removing the uniformity assumption. First, we can estimate the conditional probabilities by random sampling without resorting to the LSH function analysis. For instance, we can estimate \( P(H|T) \) by counting the number of pairs in the same buckets among the true pairs in the sample. Second, we can estimate the distribution of similarity.
Algorithm 6 LSH-S

**Input:** sample size $m$

1: $s_t = 0, s_f = 0, n_t = 0, n_f = 0$
2: for $k = 1 \ldots m$ do
3: sample a pair $(u, v), u \neq v$ uniformly at random
4: if $\text{sim}(u, v) \geq \tau$ then
5: $s_t = s_t + (\text{sim}(u, v))^k$, $n_t = n_t + 1$
6: else
7: $s_f = s_f + (\text{sim}(u, v))^k$, $n_f = n_f + 1$
8: end if
9: end for
10: $\hat{P}(H|T) = s_t/n_t$, $\hat{P}(H|F) = s_f/n_f$
11: return $J_S = \max(\frac{N_u-M}{\hat{P}(H|T)-\hat{P}(H|F)}, 1)$

For instance, if all the pairs in the sample have a similarity value of 0.3, we can only consider similarity $s = 0.3$ in Figure 6.1 without considering the whole area. We only present the second method since it outperformed the first one in our experiments.

We estimate similarity distribution with random sampling. Marginal probabilities $P(H|T)$ and $P(H|F)$ can be written as the following where $w(s)$ is the probability density function of $\text{sim}(u, v) = s$ and $P(H|s) = f(s)$ is the probability that two vectors are in the same bucket when their similarity is $s$.

\[
P(H|T) = \int_{\tau}^{1} w(s) \cdot P(H|s) ds \quad (6.7)
\]

\[
P(H|F) = \int_{0}^{\tau} w(s) \cdot P(H|s) ds \quad (6.8)
\]

Note that with uniform assumption of similarity, $w(s) = 1$ and the above probabilities are the areas defined in Figure 6.1. With a sample of pairs $S$, we only consider the similarities that exist in the sample discretizing the similarity space. Then $w(s)$ is approximated as follows:

\[
w(s) = |\{(u, v) \in S : \text{sim}(u, v) = s\}|/|S|. \quad (6.9)
\]

For instance, if similarities in $S$ are $\{0.1, 0.1, 0.1, 0.2, 0.2, 0.3\}$, $w(0.1) = 1/2, w(0.2) = 1/3, w(0.3) = 1/6$ and $w(s) = 0$ for $s \notin \{0.1, 0.2, 0.3\}$. Let $S_T$ denote the subset of true pairs in $S$ and $S_F$ denote the subset of false pairs. Since $f(s) = s^k$, using Equation (6.9) in Equation (6.7) and (6.8) gives

\[
\hat{P}(H|T) = \sum_{(u,v) \in S_T} (\text{sim}(u, v))^k / |S_T| \quad (6.10)
\]

\[
\hat{P}(H|F) = \sum_{(u,v) \in S_F} (\text{sim}(u, v))^k / |S_F|. \quad (6.11)
\]

In LSH-S in Algorithm 6, lines 3~8 accumulate counts for $P(H|s)$ values weighted by Equation (6.9). Line 10 computes $\hat{P}(H|T)$ and $\hat{P}(H|F)$ using Equation (6.10) and (6.11). We rely on the uniformity assumption when $s_t = 0$. Line 11 returns the estimate using Equation (6.1).
6.4 Stratified Sampling using LSH

A difficulty in LSH-S is that the conditional probabilities, e.g., \( P(H|T) \), need to be estimated and it is hard to acquire reliable estimates of them. In this section, we present an algorithm that overcomes this difficulty by using the LSH index in a slightly different way.

An interesting view of an LSH index is that it partitions all pairs of vectors in \( V \) into two groups: the pairs that are in the same buckets and the pairs that are not. The pairs in the same buckets are likely to be more similar from the property of the LSH scheme. Recall that the difficulty of sampling at high thresholds is that the join size is very small and sampling a true pair is very hard. Our key intuition to overcome this difficulty is that even with high thresholds it is relatively easy to sample a true pair from the set of pairs that are in the same buckets.

We demonstrate our intuition with a real-world example. Table 6.1 shows actual probabilities varying \( \tau \) using the DBLP data set. We observe that other than at low thresholds, say \( 0.1 \sim 0.3 \), \( P(T) \) is close to 0, which implies that naive random sampling is not going to work well with any reasonable sample size. However, note that \( P(T|H) \) is consistently higher than 0.04 even at high thresholds. That is, it is not difficult to sample true pairs among the pairs in the same buckets. Also observe that \( P(H|T) \) is large at high thresholds but very small at low thresholds. This means that at high thresholds, a sufficient fraction of true pairs are in the same buckets. But at low thresholds, most of true pairs are not in the same buckets, which implies that the estimate from the pairs in the same buckets is not enough. However, at low thresholds, \( P(T|L) \) becomes larger and thus sampling from pairs that are not in the same buckets becomes feasible.

| \( \tau \) | \( P(T) \) | \( P(T|H) \) | \( P(H|T) \) | \( P(T|L) \) |
|---|---|---|---|---|
| 0.1 | 0.682 | 0.31 | 0.00001 | 0.082 |
| 0.3 | 0.00024 | 0.054 | 0.00041 | 0.00024 |
| 0.5 | 0.000003 | 0.049 | 0.0028 | 0.00003 |
| 0.7 | 0 | 0.045 | 0.21 | 0 |
| 0.9 | 0 | 0.040 | 0.86 | 0 |

Table 6.1: An Example Probabilities in DBLP

Our stratified sampling algorithm exploits this trend for reliable estimation at both high and low thresholds. We formalize this idea in the following section.

6.4.1 LSH-SS: Stratified Sampling

We define two strata of pairs of vectors as follows depending on whether two vectors of a pair are in the same bucket.

- Stratum \( H \) (\( S_H \)): \( \{(u,v) : u,v \in V, B(u) = B(v)\} \)
- Stratum \( L \) (\( S_L \)): \( \{(u,v) : u,v \in V, B(u) \neq B(v)\} \).

Note that \( S_H \) and \( S_L \) are disjoint and thus we can independently estimate the join size from each stratum and add the two estimates. \( S_H \) and \( S_L \) are fixed given \( D_g \). Let \( J_H = \)
6.4. Stratified Sampling using LSH

\[ |\{(u, v) \in S_H : \text{sim}(u, v) \geq \tau\}|, \ J_L = |\{(u, v) \in S_L : \text{sim}(u, v) \geq \tau\}|, \] and let \( \hat{J}_H \) and \( \hat{J}_L \) be their estimates. We estimate the join size as follows:

\[ \hat{J}_{SS} = \hat{J}_H + \hat{J}_L \] (6.12)

A straightforward implementation of this scheme would be to perform uniform random sampling in \( S_H \) and \( S_L \), and aggregate the two estimates. However, this simple approach does not work well at high thresholds. The problem is that \( \hat{J}_L \) is unstable with large variance due to very small \( P(T|L) \) at high thresholds. See the next example.

Example 8. Assume that \( |N_L| = 1,000,000 \), \( J_L = 1 \) at \( \tau = 0.9 \), and the sample size is 10; only one pair out of 1,000,000 pairs satisfies \( \tau = 0.9 \) and we sample 10 pairs. In most cases, the true pair will not be sampled and \( \hat{J}_L = 0 \). But if the only true pair is sampled, \( \hat{J}_L = 100,000 \). The estimate fluctuates between 0 and 100,000 and is not reliable.

Our solution for this problem is to use different sampling procedures in the two strata. Recall that similarities of the pairs in \( S_H \) are higher and \( P(T|H) \) is not too small, even at high thresholds. Thus, for \( S_H \), we use uniform random sampling. Relatively small sample size will suffice for reliable estimation in \( S_H \). In \( S_L \), however, \( P(T|L) \) varies a lot depending on the similarity threshold. While at low thresholds \( P(T|L) \) is relatively high and the estimate can be reliable, \( P(T|L) \) becomes very small at high thresholds and the resulting estimate has huge variance as illustrated in Example 8. What is needed is to use \( \hat{J}_L \) only when it is expected to be reliable and discard \( \hat{J}_L \) when it is not. Discarding \( \hat{J}_L \) at high thresholds does not hurt accuracy much since the contribution of \( J_L \) in \( J \) is not large. Observe that in Table 6.1, when the similarity thresholds is high, \( P(H|T) \) is large, which means that a large fraction of true pairs are in \( S_H \), not in \( S_L \). Thus in \( S_L \), we use adaptive sampling [96] since it enables us to detect when the estimate is not reliable. A novelty is that we return a safe lower bound discarding the unstable estimate when it is not expected to be reliable.

Algorithm 7 describes the proposed stratified sampling algorithm \( LSH-SS \). It applies a different sampling subroutine to each stratum. For \( S_H \), it runs the random sampling subroutine \( \text{SampleH} \). For \( S_L \), it runs the adaptive sampling subroutine \( \text{SampleL} \). The final estimate is the sum of estimates from the two Strata as in Equation (6.12) (line 3).

Sampling in Stratum H

\( \text{SampleH} \) of Algorithm 7 describes the random sampling in \( S_H \). First, a bucket \( B_j \) is randomly sampled weighted by the number of pairs in the bucket, \( \text{weight}(B_j) = \binom{b_j}{2} \) (line 3). We then select a random pair \((u, v)\) from \( B_j \) (line 4). The resulting pair \((u, v)\) is a uniform random sample from \( S_H \). \( \text{SampleH} \) has one tunable parameter \( m_H \) which is the sample size. We count the number of pairs satisfying the similarity threshold \( \tau \) in \( m_H \) sample pairs and return the count scaled up by \( N_H/m_H \) (line 9).

Sampling in Stratum L

\( \text{SampleL} \) of Algorithm 7 specifies the adaptive sampling [96] in \( S_L \). It has two tunable parameters \( \delta \) and \( m_L \). \( \delta \) specifies the answer size threshold, the number of true samples to give
6.4. Stratified Sampling using LSH

Algorithm 7 LSH-SS

Procedure LSH-SS

Input: similarity threshold $\tau$, sample size for Stratum H $m_H$, answer size threshold $\delta$, max sample size for Stratum L $m_L$

1: $\hat{J}_H = \text{SampleH}(\tau, m_H)$
2: $\hat{J}_L = \text{SampleL}(\tau, \delta, m_L)$
3: return $\hat{J}_{SS} = \hat{J}_H + \hat{J}_L$

Procedure SampleH

Input: $\tau$, $m_H$ (sample size)

1: $n_H = 0$
2: for $i$ from 1 to $m_H$
3: sample a bucket $B_j$ with weight $(B_j) = (b_j^2)$
4: sample two vectors $u, v \in B_j$, $u \neq v$
5: if $\text{sim}(u, v) \geq \tau$ then
6: $n_H = n_H + 1$
7: end if
8: end for
9: return $\hat{J}_H = n_H \cdot N_H / m_H$

Procedure SampleL

Input: $\tau$, $\delta$ (answer size th.), $m_L$ (sample size th.)

1: $i = 0$, $n_L = 0$
2: while $n_L < \delta$ and $i < m_L$
3: sample a uniform random pair $(u, v), B(u) \neq B(v)$
4: if $\text{sim}(u, v) \geq \tau$ then
5: $n_L = n_L + 1$
6: end if
7: $i = i + 1$
8: end while
9: if $i \geq m_L$ then
10: return $\hat{J}_L = n_L$
11: end if
12: return $\hat{J}_L = n_L \cdot N_L / i$

a reliable estimate, and $m_L$ is the maximum number of samples. We sample a pair from $S_L$ (line 3) and see if it satisfies the given threshold (line 4). In case LSH computation for the bucket checking at line 3 is expensive, we can delay the checking till line 5, which in effect makes $m_L$ slightly smaller. The while-loop at line 2 can terminate in two ways: (1) by acquiring a sufficiently large number of true samples, $n_L \geq \delta$ or (2) by reaching the sample size threshold $i \geq m_L$, where $i$ is the number of samples taken. In the former case, we return the count scaled up by $N_L / i$ (line 12). In the latter case, we cannot guarantee that the estimate is reliable and simply return the number of true samples without scaling up (line 10). We justify this design by the following theorem which is a direct adaptation of Theorem 3.1 in [96].

Theorem 1. Suppose that in a run of SampleL in Algorithm 7, the while-loop terminates when we reach the condition $i \geq m_L$. Then for $0 < p < 1$, if $m_L \geq 1/(1 - p)$, the relative
error in $\hat{J}_L$ is less than $n^2$ with probability $p$, where $n$ is the database size.

Adaptive sampling gives an upper bound when the loop terminates due to the sample size threshold $m_L$. But the bound from Theorem 1 in our case is a trivial bound $n^2$, which implies we cannot guarantee that the scaled-up estimate is reliable. Thus we simply return the number of true pairs found in the sample, $n_L$, as $\hat{J}_L$ without scaling it up. Since $J_L$ is at least as large as $n_L$, we call $\hat{J}_L = n_L$ a safe lower bound.

For the tunable parameters, we used $m_H = n$ and $m_L = n$ and $\delta = \log n$ (all logarithms used are base-2 logarithms) where $n$ is the database size, $n = |V|$. Note that the size is expressed in the number of pairs of vectors, not the number of vectors. If we are sampling from two collections of vectors, $n$ pairs corresponds to $\sqrt{n}$ vectors from each collection ($n = \sqrt{n} \times \sqrt{n}$), which is even smaller than the sample size $\sqrt{n} \log n$ in the equi-join size estimation [49]. The parameter values used in our experiments give provably good estimates at both high and low thresholds. We give the details in the following section.

6.4.2 Analysis

Let $\alpha = P(T|H)$ and $\beta = P(T|L)$ for the sake of presentation. In our analysis, a similarity threshold $\tau$ is considered high when $\alpha \geq \log n/n$ and $\beta < 1/n$, and is considered low when $\alpha \geq \log n/n$ and $\beta \geq \log n/n$. Our model is analogous to the classic approach [28, 49] of distinguishing high frequency values from low frequency values to meet the challenge of skewed data. Our distinction effectively models different conditional probabilities that hold in different threshold values as in Table 6.1. 12

We first analyze the high threshold case and then analyze the low threshold case.

Guarantees at high thresholds

Recall that $\alpha = P(T|H)$ is the probability that a pair of vectors in a bucket are indeed a true pair and $\beta = P(T|L)$ is the probability that a pair of vectors that are not in the same bucket is a true pair. We assume $\alpha \geq \log n/n$ and $\beta < 1/n$ at high thresholds. The former condition on $\alpha$ intuitively states that even when the join size is small, the fraction of true pairs in $S_H$ is not too small from the property of the LSH scheme. The latter condition on $\beta$ states that it is very hard to sample true pairs in $S_L$ at high thresholds.

As a sanity check, consider the example in Table 6.1. In the data set, $n = 34,000$ and $\beta \sim 0.00003 (= 1/n)$ at $\tau = 0.5$. Thus high thresholds correspond to $[0.5, 1.0]$. At high thresholds, $\alpha = P(T|H)$ is consistently higher than 0.04 which is well over the assumed value of $\alpha$ which is 0.00046 (= $\log n/n$) for the data set. $\beta = P(T|L)$ is also below or very close to the calculated 0.00003 ($\beta = 1/n$) in the range.

The following theorem states that LSH-SS gives a good estimate at high thresholds.

---

12 We can also perform a similar analysis by categorizing high and low thresholds by join size as done in the equi-join size estimation [49]. However, they are fundamentally the same and we feel that our current notations enable a more direct and clearer analysis.
6.4. Stratified Sampling using LSH

**Theorem 2.** Let $0 < \epsilon < 1$ be an arbitrary constant. Assume $\alpha \geq \log n/n$ and $\beta < 1/n$. Then for sufficiently large $n$ with $c = 1/(\log e \cdot \epsilon^2)$, $m_H = cn$ and $m_L = cn$,

$$Pr(|\hat{J}_{SS} - J| \leq (1 + \epsilon)J) \geq 1 - \frac{2}{n}.$$  

To prove Theorem 2, we analyze the behavior of $SampleH$ and $SampleL$ in Algorithm 7 separately by the two lemmas below. We then combine the two results using Equation (6.12).

**Lemma 1.** Let $0 < \epsilon < 1$ be an arbitrary constant. Assume $\alpha \geq \log n/n$. Then with $c = 4/(\log e \cdot \epsilon^2)$ and $m_H = cn$, we have

$$Pr(|\hat{J}_H - J_H| \leq \epsilon J_H) \geq 1 - \frac{1}{n}.$$  

**Proof.** Let $X$ be a random variable denoting the number of pairs in the sample that satisfy $\tau$ in $SampleH$ of Algorithm 7. Then $X$ is a binomial random variable with parameters $(m_H, \alpha)$ [111]. Since $m_H = cn$ and $\alpha \geq \log n/n$, 

$$E(X) \geq c \log n.$$  

For an arbitrary constant $0 < \epsilon < 1$, by Chernoff bounds [111]

$$Pr(|X - E(X)| > \epsilon E(X)) \leq e^{-\frac{c \log n^2}{4}}.$$  

Letting $c = 4/(\log e \cdot \epsilon^2)$ gives

$$Pr(|X - E(X)| > \epsilon E(X)) \leq \frac{1}{n}.$$  

$$E(X) = J_H \cdot \frac{m_H}{N_H}.$$  

Thus, $\frac{N_H}{m_H} \cdot E(X) = \frac{N_H}{m_H} \cdot m_H \cdot \frac{J_H}{N_H} = J_H$. Therefore,

$$Pr(|\frac{N_H}{m_H} \cdot X - J_H| > \epsilon J_H) \leq \frac{1}{n}.$$  

$\hat{J}_H = X \cdot \frac{N_H}{m_H}$ in $SampleH$ of Algorithm 7. Plugging in $\hat{J}_H$ in the above inequality completes the proof.  

Second, the following lemma states that $SampleL$ returns a safe lower bound with high probability.

**Lemma 2.** Assume $\beta < 1/n$. Then for sufficiently large $n$, an arbitrary constant $c'$ and $m_L = c'n$, we have

$$Pr(\hat{J}_L \leq c') \geq 1 - \frac{1}{n}.$$
6.4. Stratified Sampling using LSH

Proof. Let \( Y \) be a random variable denoting the number of pairs in the samples satisfying \( \tau \) in \textit{SampleL} of Algorithm 7. We show that \( Y \) is not likely to be bigger than \( \delta = \log n \). \( Y \) is a binomial random variable with parameters \((m_L, \beta)\).

For an arbitrary constant \( \epsilon' \geq 2e - 1 \), by Chernoff bounds \([111]\)

\[
Pr(Y > (1 + \epsilon')E(Y)) \leq 2^{-E(Y)(1+\epsilon')}.
\]

Therefore, the probability that the loop of \textit{SampleL} terminates by acquiring enough number of true pairs \( (\delta = \log n) \) is as follows:

\[
Pr(Y > \delta = \log n) \leq 2^{-\log n} = \frac{1}{n}.
\]

Since \( m_L = c'n \) and \( \beta < 1/n \),

\[
E(X) < c'.
\]

If \( Y \leq \delta \), \textit{SampleL} of Algorithm 7 returns \( \hat{J}_L = Y \) without scaling it up. Therefore,

\[
Pr(\hat{J}_L = E(Y) < c') \geq 1 - \frac{1}{n}.
\]

Finally, we prove Theorem 2 using Lemma 1 and Lemma 2.

Proof. For sufficiently large \( n \), \( c' \leq J_L \). Thus from Lemma 2,

\[
Pr(|\hat{J}_L - J_L| \leq J_L) \geq 1 - \frac{1}{n}.
\]

Since \( \hat{J}_{SS} = \hat{J}_H + \hat{J}_L \) and \( J = J_H + H_L \), combining the above inequality with Lemma 1 proves the theorem.

Guarantees at low thresholds

At low thresholds, we assume that \( \alpha \geq \log n/n \) and \( \beta \geq \log n/n \). The rationale is that as the actual join size increases, more true pairs are in \( S_L \) and sampling true pairs in \( S_L \) becomes not so difficult any more. Again these conditions are usually met when the threshold is low as in the example in Table 6.1. In fact, the contribution from \( S_L \), \( J_L \) dominates the join size at low thresholds. The following theorem states that our estimate using Equation (6.12) gives a reliable estimate even when the threshold is low.

Theorem 3. Let \( 0 < \epsilon < 1 \) be an arbitrary constant. Assume \( \alpha \geq \log n/n \) and \( \beta \geq \log n/n \). Then with \( c = 4/(\log e \cdot \epsilon^2) \), \( c' = \max(c, 1/(1-\epsilon)) \), \( m_H = cn \) and \( m_L = c'n \),

\[
Pr(|\hat{J}_{SS} - J| \leq \epsilon J) \geq 1 - \frac{3}{n}.
\]
6.5. Additional Discussions

Proof. As we have the same conditions on $\alpha$, $m_H$, and $c$, Lemma 1 still holds in the low threshold range as well. However, due to the increased $\beta$, a different analysis needs to be done for $SampleL$ (Lemma 2). We first show that $SampleL$ returns a scaled-up estimate not a safe lower bound with high probability, and then show that the scaled-up estimate is reliable.

Similarly as in Lemma 2, $Y$ is a binomial random variable with parameters $(m_L, \beta)$. Since $m_L = c'n$ and $\beta \geq \log n/n$,

$$E(Y) \geq c'\log n.$$  

From the given condition, $c' \geq 2/\left(\log e \cdot \epsilon^2\right)$ and $c' \geq 1/(1 - \epsilon)$. Thus, by Chernoff bounds,

$$Pr(Y \geq (1 - \epsilon)E(Y) \geq (1 - \epsilon)c'\log n \geq \log n = \delta) \geq 1 - \frac{1}{n}.$$  

This means that the while-loop (line 2) of $SampleL$Algorithm 7 terminates by reaching the desired answer size threshold $\delta$ with high probability. Then $SampleL$ returns $\hat{J}_L = Y \cdot J_L/m_L$. Moreover, since $c' \geq 4/\left(\log e \cdot \epsilon^2\right)$, by Chernoff bounds,

$$Pr(|Y - E(Y)| \geq \epsilon E(Y)) \leq \frac{2}{n}.$$  

Therefore,

$$Pr(|\hat{J}_L - J_L| \leq \epsilon J_L) \geq 1 - \frac{2}{n}.$$  

Since $\hat{J}_{SS} = \hat{J}_H + \hat{J}_L$ and $J = J_H + H_L$, the above inequality along with Lemma 1 completes the proof.

When $1/n \leq \beta < \log n/n$ for which we do not guarantee the accuracy, one can use a dampened scaling up factor between 1 and $N_L/i$ (instead of 1 or $N_L/i$) at line 12 of $SampleL$. We do not focus on such heuristics in this chapter. We demonstrate that our guarantees indeed hold and thus $LSH-SS$ provides reliable estimates at both high and low thresholds with real-world data sets in the following section.

6.5 Additional Discussions

6.5.1 The Optimal-$k$ for The VSJ Problem

Recall that $k$ is the number of LSH functions for $g$. Ideally, we want high $P(T|H)$ and $P(H|T)$ because the estimate from $S_H$ is more reliable and having more true pairs in $S_H$ reduces the dependency on $S_L$. When $S_H$ has all the true pairs ($P(H|T) = 1$) and does not have any false pairs ($P(T|H) = 1$)), we can trivially solve the VSJ problem; $\hat{J} = N_H = J$. $P(T|H)$ (resp. $P(H|T)$) is analogous to precision (resp. recall) in information retrieval. We note that $k$ value has the following trade-offs between $P(T|H)$ and $P(H|T)$.

- A larger $k$ increases $P(T|H)$ but decreases $P(H|T)$. With a sufficiently large $k$, only exactly the same vectors will be in the same buckets. $P(T|H) = 1$ in this case since any pairs of vectors from a bucket are the same vector. However, only a very small fraction of true pairs is in the same buckets resulting in a very small $P(H|T)$.
6.5. Additional Discussions

- A smaller $k$ decreases $P(T|H)$ but increases $P(H|T)$. In an extreme case of $k = 0$, $S_H$ consists of all the pairs of vectors in $V$, and thus $P(H|T) = 1$. However, $P(T|H) = P(T)$ and the LSH scheme does not offer any benefit.

Observe that $P(T|H) \geq P(T|L)$ from the property of the LSH indexing. The intuition on choosing $k$ is that we want to increase $J_H$ as long as we have good estimates with high probability. Since decreasing $k$ increases $P(H|T)$ and $J_H$, this means that we can decrease $k$ as long as $P(T|H)$ is not too small. Decreasing $k$ also reduces the LSH function computation time. With this intuition, we can formalize the problem of choosing $k$ as follows:

**Definition 10** (The Optimal-$k$ Problem). Given a desired error bound $\epsilon > 0$ and the bound on the probabilistic guarantee $p$, find the minimum $k$ such that $P(T|H) \geq \rho$, where $\rho = \rho(\epsilon, p)$.

Assuming a similarity distribution of the database (and thus a formula on $P(T|H)$, e.g., Figure 6.1), we can analytically find the optimal $k$. However, $P(T|H)$ is dependent on the data and the LSH scheme used, and so is the optimal value of $k$. We empirically observed that $5 \leq k \leq 15$ generally gives the best accuracy for the binary LSH function used for cosine similarity. We also observed that using a too small value of $k$, say 5, had adversary impacts on runtime as well as accuracy.

6.5.2 Extensions

**Using Multiple LSH Tables**

The proposed algorithms so far assume only a single LSH table, but a typical LSH index consists of multiple LSH tables. In this section, we describe how we can utilize more than one LSH tables for the estimation purposes. We consider two estimation algorithms using multiple LSH tables: median estimator and virtual bucket estimator.

**Median estimator.** The median estimator applies LSH-SS to each LSH table independently without modifying LSH-SS and merge the estimates. Suppose an LSH index $I_G = \{D_{g1}, \ldots, D_{g\ell}\}$ with $\ell$ tables. From each table $D_{gi}, 1 \leq i \leq \ell$, we generate an estimate $\hat{J}_i$ with a sample of $n$ pairs. Its estimate, $\hat{J}_m$, is the median of the estimates, $\hat{J}_i, 1 \leq i \leq \ell$. This approach makes the algorithm more reliable reducing the probability that $\hat{J}_m$ deviates much from the true size. From Theorems 2 and 3, the probability that $\hat{J}_i$ differs from $J$ by more than a factor of $1 + \epsilon$ is less than $2/n$ with the assumed join size and sample size. When taking the median, the probability that more than $\ell/2$ $\hat{J}_i$'s deviate by more than $(1 + \epsilon)J$ is at most $2^{-\ell/2}$ by the standard estimate of Chernoff [111]. This states that $\hat{J}_m$ is within the same factor of error with higher probability than the guarantees in Theorems 2 and 3. The effective sample size increases by a factor of $\ell$. When a sample size that is greater than $n$ is affordable, exploiting multiple LSH tables can make the estimate more reliable. However, dividing a total sample size of $n$ into multiple estimates can impair the accuracy of individual estimates, and thus it is not recommended.

**Virtual bucket estimator.** The second approach is to consider virtual buckets formed by multiple LSH tables. In this approach, a pair $(u, v)$ is regarded as in the same bucket if $u$ and $v$ are in the same bucket in any of $\ell$ LSH tables. This can improve the accuracy when an existing LSH scheme has a relatively large $k$ than necessary. Recall from the discussions
in the previous section that when $k$ is too large, the hash function $g$ becomes overly selective and $S_H$ can be too small. Then the true pairs from $S_H$ can be only a small portion of the true pairs. Considering virtual buckets can address this problem by relaxing the bucket conditions; $B(u) = B(v)$ if and only if $\exists D_g, B_j \subseteq D_g, \text{such that } B_j(u) = B_j(v), 1 \leq i \leq \ell$. With virtual buckets, when we check $B(u) = B(v)$ (or $\neq$) for $(u, v)$, we need to check up to $\ell$ tables. At lines 3 and 4 of SampleH in Algorithm 7, a pair $(u, v)$ is chosen from $V$ uniformly at random and discarded if $u$ and $v$ are not in the same bucket in any $D_g, 1 \leq i \leq \ell$. At line 3 of SampleL, $B(u) \neq B(v)$ is checked for all $D_g$, and if $B(u) = B(v)$ in any $D_g$, the sample pair $(u, v)$ is discarded. The analysis remains the same but the set of pairs in same buckets, $S_H$ becomes effectively larger, which gives potentially better accuracy. The runtime increases because now the bucket checking is done for multiple LSH tables.

**Non-self Joins**

In this section, we discuss how to extend the proposed algorithms to handle joins between two collections of vectors $U$ and $V$. The basic ideas remain the same but we need to make sure that a pair under consideration consists of one vector from each collection.

**Definition 11 (The General VSJ Problem).** Given two collection of real-valued vectors $V = \{v_1, ..., v_m\}$, $U = \{u_1, ..., u_n\}$ and a threshold $\tau$ on similarity $\text{sim}$, estimate the number of pairs $J = \{(u, v) : u \in U, v \in V, \text{sim}(u, v) \geq \tau\}$.

Suppose that we have two LSH tables $D_g$ and $E_g$ that are built on $U$ and $V$ respectively using $g = (g_1, \ldots, g_\ell)$. We describe how we modify LSH-\textit{S} and LSH-\textit{SS} for general join processing.

**LSH-S.** We make two changes for general joins in LSH-\textit{S}: $N_H$ computation and sampling in Algorithm 6. In self-joins, $N_H$ in Equation 6.1 is the number of pairs in the same buckets: $N_H = \sum_{j=1}^{n_g} b_j$. In general joins, $N_H = \sum_{j=1}^{n_g} b_j \cdot c_i$ such that $B_j \subseteq D_g, C_i \subseteq E_g, b_j$ is the bucket count of $B_j$, $c_i$ is the bucket count of $C_i$, and $g(B_j) = g(C_i)$, where $g(B)$ denotes the $g$ value that identifies bucket $B$. For $B_i$, if there does not exist a bucket $C_i \subseteq E_g$ such that $g(C_i) = g(B_j)$, $c_i = 0$. Next, the pair $(u, v)$ is sampled uniformly at random such that $u \in U$ and $v \in V$ at line 3 in Algorithm 6.

**LSH-SS.** Intuitively, $S_H$ is the set of pairs $(u, v), u \in U, v \in V$ such that $g(u) = g(v)$. That is, the buckets of $u$ and $v$ have the same $g$ value: $g(B_j) = g(C_i)$ where $u \in B_j \subseteq D_g$ and $u \in C_i \subseteq E_g$. $S_L$ is the set of pairs $(u, v), u \in U, v \in V$ such that $g(u) \neq g(v)$. To sample a pair from $S_H$, we randomly sample a bucket $B_j$ with weight($B_j$) = $b_j \cdot c_i$ where $g(C_i) = g(B_j), B_j \subseteq D_g$ and $C_i \subseteq E_g$. Then we sample $u$ from $B_j$ and $v$ from $C_i$ uniformly at random within the buckets. The resulting pair $(u, v)$ is a uniform random sample from $S_H$. Alternatively, we can sample $u \in U$ and $v \in V$ uniformly at random, and discard $(u, v)$ if $g(u) \neq g(v)$. Lines 3 and 4 of SampleH in Algorithm 7 need to be changed accordingly. To sample from $S_L$, we sample $u \in U$ and $v \in V$ uniformly at random. The pair $(u, v)$ is discarded if $g(u) = g(v)$. This requires corresponding changes in line 3 of SampleH.
6.6 Experimental Evaluation

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V$</td>
<td>a collection of vectors, database</td>
</tr>
<tr>
<td>$J$</td>
<td>join size</td>
</tr>
<tr>
<td>$n$</td>
<td>database size, $</td>
</tr>
<tr>
<td>$m$</td>
<td>sample size</td>
</tr>
<tr>
<td>$u, v$</td>
<td>vectors</td>
</tr>
<tr>
<td>$\tau$</td>
<td>similarity threshold</td>
</tr>
<tr>
<td>$h$</td>
<td>an LSH function, e.g., $h_1(u)$</td>
</tr>
<tr>
<td>$k$</td>
<td>$#\text{LSH functions for an LSH table}$</td>
</tr>
<tr>
<td>$\ell$</td>
<td>$#\text{LSH tables in an LSH index}$</td>
</tr>
<tr>
<td>$g$</td>
<td>$g = (h_1, \ldots, h_k)$, vector of LSH functions</td>
</tr>
<tr>
<td>$D_g, E_g$</td>
<td>an LSH table using $g$</td>
</tr>
<tr>
<td>$G$</td>
<td>$G = {g_1, \ldots, g_\ell}$, vector of LSH functions</td>
</tr>
<tr>
<td>$I_G$</td>
<td>an LSH index with using $G$ that consists of $D_{g_1}, \ldots, D_{g_\ell}$</td>
</tr>
<tr>
<td>$B$</td>
<td>a bucket in an LSH table, e.g., $B(u)$: the bucket of $u$</td>
</tr>
<tr>
<td>$b_j, c_i$</td>
<td>bucket counts, e.g., $b_j$ is the bucket count of bucket $B_j$</td>
</tr>
<tr>
<td>$T$</td>
<td>True pairs, $(u, v)$ such that $\text{sim}(u, v) \geq \tau$</td>
</tr>
<tr>
<td>$F$</td>
<td>False pairs, $(u, v)$ such that $\text{sim}(u, v) &lt; \tau$</td>
</tr>
<tr>
<td>$H$</td>
<td>High (expected) sim. pairs that are in the same buckets</td>
</tr>
<tr>
<td>$L$</td>
<td>Low (expected) sim. pairs that are not in the same buckets</td>
</tr>
<tr>
<td>$P(T</td>
<td>H)$</td>
</tr>
<tr>
<td>$P(H</td>
<td>T)$</td>
</tr>
<tr>
<td>$P(T</td>
<td>L)$</td>
</tr>
<tr>
<td>$P(L</td>
<td>T)$</td>
</tr>
<tr>
<td>$S$</td>
<td>stratum, e.g., $S_H$: set of pairs that are in the same buckets</td>
</tr>
<tr>
<td>$N$</td>
<td>$#\text{pairs, e.g., } N_H$: $#\text{pairs in the same buckets}$</td>
</tr>
<tr>
<td>$M$</td>
<td>$#\text{pairs in } V, M = \binom{</td>
</tr>
</tbody>
</table>

Table 6.2: Summary of Notations

6.6 Experimental Evaluation

6.6.1 Set Up

Data sets: We have conducted experiments using three data sets. The DBLP data set consists of 794,016 publications is the same as the data set used in [13]. Each vector is constructed from authors and title of a publication. There are about 56,000 distinct words in total and each word is associated with a dimension in a vector. The vector of a publication represents whether the corresponding word is present in the publication. Thus this data set is a binary vector data set. The average number of features is 14, and the smallest is 3 and the biggest is 219. The NYT data set is NYTimes news articles downloaded from UCI Machine Learning Repository\(^{13}\) and consists of 149,649 vectors. Again each dimension represents a word and has the corresponding TF-IDF weight. The dimensionality is about 100k and the average number of feature is 232. The PUBMED data set is constructed from PubMed abstracts and is also downloaded from the UCI repository. It consists of 400,151 TF-IDF vectors. The dimensionality is about 140k.

Algorithms compared: We implemented the following algorithms for the VSJ problem.

- $LC(\xi)$ is the Lattice Counting algorithm in [92] with a minimum support threshold $\xi$.
- $LSH-S$ is the LSH index based algorithm in Section 6.3.

\(^{13}\)http://archive.ics.uci.edu/ml/datasets.html
• **LSH-SS** is the LSH index based stratified sampling algorithm in Section 6.4. We used $m_1 = n$, $\delta = \log n$, $m_2 = n$ and $k = 20$ unless specified otherwise.

• **RS** is uniform random sampling with sample size $m_R = d \cdot n$ where $d > 1$ is a data set dependent constant to compare algorithms with roughly the same runtime.

**Evaluation metric:** We use average relative error to show the accuracy. A relative error is defined as $est\_size / true\_size$. We do not take the absolute value to show underestimation more clearly. We report each error averaged over 10 experiments. Positive and negative values can cancel each other when taking the average, but in our experiments, most of the algorithms consistently show either overestimation or underestimation depending on the threshold. Moreover, we separately report variance of errors. To measure reliability of each algorithm, we report the standard deviation (STD) of errors. For efficiency, we measure the runtime, which is time taken for estimation. For all algorithms, we loaded necessary data structures (data set or index) in memory.

We implemented all the algorithms in Java. We ran all our experiments on a server running 64 GNU/Linux 2.6.27 over 2.93 GHz Intel Xeon CPUs with 64GB of main memory.

### 6.6.2 DBLP: Accuracy, Variance and Efficiency

We first report the results on the accuracy, variance and runtime using the DBLP data set. Figure 6.2(a) shows relative error of various algorithms over the similarity threshold range and Figure 6.2(b) shows variance. Figure 6.2(a) is also presented in Table 6.3 for more details.

The estimations of **LSH-S** have large errors at high thresholds, e.g., $\tau \geq 0.6$. This is because the estimations of conditional probabilities are not reliable due to insufficient number of true pairs sampled. Its variance is large since Equation (6.1) is sensitive to those estimations.

**LSH-SS** delivers accurate estimation over the whole range. For most threshold values, its estimate is the most accurate one. The following table shows the actual join size $J$ and its selectivity at various similarity threshold $\tau$. Note the dramatic differences in the join size depending on $\tau$. At $\tau = 0.1$ there are more than 100 billion true pairs, and its selectivity is about 30%. But at $\tau = 0.9$ there are only 42,000 true pairs, and the selectivity is only 0.00001%. Yet **LSH-SS** is able to estimate the join size quite accurately and reliably exploiting the LSH index; its average relative error is 48% at $\tau \geq 0.7$. Moreover, **LSH-SS** has very small variance at high thresholds. At low thresholds, the STD of **LSH-SS** is as big as or larger than that of **RS**. However, in this range, the true join size itself is very large; its STD is still much smaller than the true join size.

**RS** is as accurate as **LSH-SS** in the low threshold range. However as the threshold increases, its error rapidly increases; its average relative error is 745% at $\tau \geq 0.7$. Moreover, the variance of **RS** is an order of magnitude larger than the true join size at high thresholds. This is expected from the hardness of sampling true answers with high selectivity. The high
error and variance of RS at high thresholds are problematic since similarity thresholds between 0.5 and 0.9 are typically used [13].

LC underestimate over the whole threshold range. We hypothesize that it is because the characteristic of the LSH function for cosine similarity. The LSH for cosine similarity we used is a binary function, and thus intuitively it needs more hash functions (larger $k$) to distinguish objects. However, increasing the number of hash function has negative impact on the runtime for its mining process. It appears that LC is not adequate for binary LSH functions.

For runtime, LSH-S took 1028 msec. LSH-SS took 757 msec, and RS took 789 msec on the average. The runtime of LC was 3 sec.

<table>
<thead>
<tr>
<th>$\tau$</th>
<th>LSH-SS</th>
<th>LSH-S</th>
<th>RS</th>
<th>LC</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>$-74.28%$</td>
<td>$-38%$</td>
<td>$-75.50%$</td>
<td>$-99.99%$</td>
</tr>
<tr>
<td>0.2</td>
<td>$-77.83%$</td>
<td>9%</td>
<td>$-79.76%$</td>
<td>$-99.97%$</td>
</tr>
<tr>
<td>0.3</td>
<td>$-63.14%$</td>
<td>171%</td>
<td>$-70.19%$</td>
<td>$-99.81%$</td>
</tr>
<tr>
<td>0.4</td>
<td>$-14.92%$</td>
<td>1843%</td>
<td>$-27.29%$</td>
<td>$-99.58%$</td>
</tr>
<tr>
<td>0.5</td>
<td>$-1.39%$</td>
<td>232%</td>
<td>0.46%</td>
<td>$-99.55%$</td>
</tr>
<tr>
<td>0.6</td>
<td>$-95.29%$</td>
<td>6123%</td>
<td>45.54%</td>
<td>$-95.25%$</td>
</tr>
<tr>
<td>0.7</td>
<td>$-85.67%$</td>
<td>7562%</td>
<td>132.90%</td>
<td>$-92.39%$</td>
</tr>
<tr>
<td>0.8</td>
<td>$-69.66%$</td>
<td>1116%</td>
<td>298.60%</td>
<td>$-92.79%$</td>
</tr>
<tr>
<td>0.9</td>
<td>$-32.68%$</td>
<td>1771%</td>
<td>1021.12%</td>
<td>$-92.35%$</td>
</tr>
<tr>
<td>1.0</td>
<td>$-4.69%$</td>
<td>1810%</td>
<td>1522.46%</td>
<td>$-95.88%$</td>
</tr>
</tbody>
</table>

Table 6.3: Relative Error on DBLP
6.6. Experimental Evaluation

6.6.3 NYT: Accuracy, Variance and Efficiency

Figure 6.3(a) and (b) show relative error and variance on the NYT data set. We only show $LSH-SS$ and $RS$ for clarity since they dominate the other algorithms. We observe the same phenomena in the NYT data set. $LSH-SS$ gives good estimates over the whole thresholds. It shows underestimation at $0.3 \leq \tau \leq 0.5$. This is because it takes conservative approach of not scaling up the number of true pairs in the sample when its reliability is not certain. But in general this is not the most interesting similarity range. Heuristics such as using a dampened scaling up factor as briefly mentioned in Section 6.4.2 can improve the accuracy in this range if needed. Its variance is two orders of magnitude smaller than the variance of $RS$ except at very low thresholds. $RS$ also shows large overestimation at high thresholds. For runtime, $LSH-SS$ took 1091 msec and $RS$ took 920 msec on average.

6.6.4 PUBMED: Accuracy, Variance, Efficiency

Figure 6.5 shows the accuracy and variance on the PUBMED data set with $k = 5$. The average error of $LSH-SS$ is 73% and that of $RS$ is 117%. $LSH-SS$ shows underestimation tendency but its STD is more than an order of magnitude smaller than that of $RS$. When the data set is largely dissimilar, smaller $k$ improves accuracy. In such cases, constructing an LSH table on-the-fly can be a viable option. For runtime, $LSH-SS$ took 1188 msec and $RS$ took 717 msec on average.
6.6. Experimental Evaluation

The relevant parameter for our estimation purposes is $k$ which specifies the number of hash functions used for an LSH table. We analyze the impact of $k$ on accuracy and variance. $RS$ is not relevant to $k$, so we only compare $LSH-SS$ and $LSH-S$.

Figure 6.4 shows accuracy at $\tau = 0.5, 0.8$; the conclusion is the same at other thresholds. We observe that $LSH-SS$ is not much affected by $k$. This is because an LSH table provides sufficient distinguishing power with relatively small $k$. $LSH-SS$ will work with any reasonable choice of $k$. $LSH-S$ is highly sensitive to $k$ for the reason specified in Section 6.6.2. The same observation is made in variance as well.

<table>
<thead>
<tr>
<th>$k$</th>
<th>$k = 10$</th>
<th>$k = 20$</th>
<th>$k = 30$</th>
<th>$k = 40$</th>
<th>$k = 50$</th>
</tr>
</thead>
<tbody>
<tr>
<td>size (MB)</td>
<td>3.2</td>
<td>7.5</td>
<td>12.6</td>
<td>14.1</td>
<td>16.5</td>
</tr>
</tbody>
</table>

The above table shows space occupied by a LSH table on the DBLP data set ignoring implementation-dependent overheads. When $k = 20$, there are about $480K$ non-empty buckets which adds roughly $7.5M$ of space for the $g$ function values (string representation for $k$ concatenated binary values), bucket counts, and vector ids. The DBLP binary file is about
6.6. Experimental Evaluation

If we are given the freedom of choosing $k$, we observe that slightly smaller $k$ values, say between 5 and 15, generally give better accuracy.

**Impact of Parameters: $\delta$ and $m$**

In this section, we study the impact of parameters related with the sample size using the DBLP data set. The goal is to see if our analysis and choice of parameter values are appropriate. Recall that $LSH$-$SS$ and $RS$ have the following parameters.

- $m_H$ ($LSH$-$SS$): the sample size in the $SampleH$ subroutine for Stratum H
- $\delta$ ($LSH$-$SS$): the answer size threshold in the adaptive sampling procedure $SampleL$ for Stratum L
- $m_L$ ($LSH$-$SS$): the (maximum) sample size in the adaptive sampling procedure $SampleL$ for Stratum L
- $m_R$ ($RS$): the sample size.

In our analysis and experiments, we used the following parameter values: $m_H = m_L = n$, $\delta = \log n$, and $m_R = 1.5n$. Since $m_H, m_L$ and $m_R$ controls the overall sample size and $\delta$ specifies the answer size threshold, we use two functions $f_1$ and $f_2$ to control the parameters: $m_H = m_L = f_1(n)$, $m_R = 1.5f_1(n)$ and $\delta = f_2(n)$. We test the following alternatives:

- $f_1(= m)$: $\sqrt{n}$, $n/\log n$, $0.5n$, $n$, $2n$, and $n\log n$
- $f_2$: $0.5\log n$, $\log n$, $2\log n$, and $\sqrt{n}$

We perform two types of combinations of $f_1$ and $f_2$. First, we fix $f_1(n) = n$ and experiment the above alternatives for $f_2$. Next, we fix $f_2(n) = \delta = \log n$ and experiment the above alternatives for $f_1$. For each combination we show two results: the average absolute error for $\tau = \{0.1, 0.2, ..., 1.0\}$ and the number of $\tau$ values that give large. $m(= f_1)$ is fixed at $n$. Recall that $SampleL$ terminates the loop when the number of true sample reaches $\delta$ regardless of the sample size. $\delta > 2\log n$ has a big underestimation problem. A large $\delta$ may prevent even a reliable estimate from being scaled up and result in a huge loss in contributions from $S_L$. For instance, $\delta = \sqrt{n}$ is too conservative and its estimate is less than 10% of the true size at 4 out of 10 $\tau$ values. We recommend $\delta = O(\log n)$ for general settings.

Simple heuristics such as using different $\delta$ depending on the threshold can improve the performance. For instance using $0.1\log n \leq \delta \leq 0.5\log n$ at high thresholds, e.g., $\tau \geq 0.7$,
6.6. Experimental Evaluation

![Graph](image)

Figure 6.6: Relative Error Varying $\delta$ (the Answer Size Threshold) in SampleL

![Graph](image)

Figure 6.7: The Number of $\tau$ with $\hat{J}/J \geq 10$ (overest.) or $J/\hat{J} \geq 10$ (underest.) Varying $\delta$. The total number of $\tau$ is 10: $\{0.1, 0.2, ..., 1.0\}$

greatly improved the runtime without sacrificing accuracy in our experiments. At low thresholds, e.g., $\tau \leq 0.3$, using a slightly bigger value of $\delta$, e.g., $2 \log n$, resulted in better accuracy and variance without increasing the runtime much.

**Sample Size Threshold $m$**

Figure 6.8 gives the average (absolute) relative error varying the sample size and Figure 6.9 gives the number of $\tau$ values that give large errors. $\delta$ is fixed at $\log n$. We see that $f_1 < 0.5n$ causes serious underestimations in both algorithms. With $f_1 = n \log n$, LSH-SS does not give any big over/underestimations. But the runtime roughly increases by $\log n$. $f_1 = O(n)$ is recommended for general settings.

6.6.6 Summary of Experiments

We evaluated the proposed solutions, LSH-S and LSH-SS, using three real-world data sets: DBLP, New York Times articles, and PUBMED abstracts. They are compared with LC, which is adapted to the VSJ problem, and random sampling (RS). While RS exhibited severe under estimation problems at high thresholds, LSH-SS consistently showed relative errors smaller than 50% throughout all the thresholds and data sets, which is big improvements over the adapted LC. The required sample size was only $n$ pairs, which also improves the runtime of the adapted LC.
6.7. Conclusion

We propose size estimation techniques for vector similarity joins. The proposed methods rely on the ubiquitous LSH indexing and enable reliable estimates even at high similarity thresholds. We show that the proposed stratified sampling algorithm LSH-SS gives good estimates throughout the threshold range with probabilistic guarantees. The proposed solution only needs minimal addition to the existing LSH index and can be easily applied.
Chapter 7

Conclusion

Approximate text processing is crucial in handling textual data in databases; exact matching can miss meaningful results due to pervasive errors in text. There has been extensive interest in approximate text processing, and commercial databases have begun to incorporate such functionalities. One of the key components in successful integration of approximate text handling in RDBMSs is to have reliable selectivity estimation techniques, which is central to optimizing queries. However, these developments are relatively new and we lack solid techniques for estimating selectivity of them.

This dissertation aims to provide reliable selectivity estimation techniques for approximate predicates on text. Among many possible predicates, we focus on two types of predicates which are the two fundamental building blocks of SQL queries: selections and joins. Depending on the matching semantics, we study two different semantics for each type of operator. We review the key contributions and discuss future research directions in this chapter.

7.1 Key Contributions

We make three major contributions in this dissertation. The first major contribution is that we propose a lattice based framework for counting possible variants in similarity matching. A common challenge in the selectivity estimation of approximate matching is that there can be a huge number of variants to consider given a query string and a similarity threshold. We consider a group of similar variants for efficient counting rather than each and every possible variant. For selection problems in Chapter 3 and 4, the group is formed by wildcard. Strings that have same characters other than positions marked by wildcards are considered together. For the join problem in Chapter 5, the group is formed by frequent patterns in the signature representation of a database. Although the grouping enables efficient counting for many variants, a challenge is that the groups may have overlaps in their counts. We organize groups with lattice structures, and propose formulas and algorithms that can perform counting without duplicates based on the lattice structures.

The second contribution is that we develop selectivity estimation algorithms for approximate selection and join operators based on the framework. We show how we can apply recent developments for approximate matching on the framework. We extend summary structures for exact text matching to support approximate matching and integrate them with Min-Hashing signatures.

The last contribution is that we adapt classical random sampling algorithms to similarity join size estimation. Similarity joins fall into hard cases for algorithms developed for equi-join size estimation and their guarantees are not meaningful in similarity joins. However, we show that those algorithms can be adapted to similarity join size estimation as well. We
7.2 Future Work

We summarize our contributions by chapters as follows.

- In Chapter 3, we present solutions for the string problem. We extend the Q-gram table [26] with wildcard called the EQ-gram table, which enables efficient counting of strings of a similar form. The lattice is formed by inclusion relationships among possible forms of strings within an edit distance threshold. We then propose the replace-only formula, which handles the duplicated counting problem using the lattice. Algorithm OptEQ is proposed for a general edit distance case and it also exploits observations on the lattice for efficiency.

- In Chapter 4, we develop algorithms for the substring problem, which is a generalization of the string problem. We propose two solutions, MOF and LBS. MOF uses the EQ-gram table and is based on a generalization of the SIS assumption [26]. LBS augments the EQ-gram table with Min-Hash signatures and apply Set Hashing [30] for better accuracy.

- In Chapter 5, we develop algorithms for the set similarity join size estimation (SSJ). We show that there is relationship between the number of frequent patterns in the signature representation of sets, and apply a Power-Law for efficient estimation of the number of frequent signature patterns. The signature pattern conceptually groups similar sets and we define lattice structure to model their relationships. The replace-only formula developed for the string problem is adapted to the SSJ problem to account for the overlaps among signature patterns. We observe that there is distribution shift by relying on Min-Hashing and propose a procedure to compensate the errors from the shift.

- In Chapter 6, we present algorithms for the vector similarity join size estimation (VSJ), which is a generalization of the SSJ problem. The proposed algorithm performs random samplings on the LSH index. We make use of Adaptive Sampling [96] and present probabilistic guarantee on its performance.

7.2 Future Work

This dissertation provides significant initial steps toward selectivity estimation of approximate predicates on text. However, approximate matching has rich contexts and it still leaves ample opportunities for future research. We look into several future research directions.

7.2.1 Supporting Diverse Similarity Measures

We studied several similarity measure in this dissertation including edit distance, hamming distance, overlap count, Jaccard similarity, cosine distance. Although we dealt with major similarity measures and their extensions, there exist many more ways to consider similar strings such as Soundex, synonym, distance query, abbreviation, just to name a few. In
7.2. Future Work

reality, it is not likely that any single measure will work best for all cases, which is one of the reasons why there are diverse similarity measure developed.

Approximate text processing engines often provide more than one similarity semantics. For instance, the Oracle CONTAINS operator supports ‘broader term’, ‘narrower term’, constraints on term proximity, related terms, synonyms, or Soundex. Developing selectivity estimation techniques for all the measure can be tricky.

There are largely two approaches for selectivity estimation purposes: summary structure-based algorithms and random sampling-based algorithms. In this dissertation, we examined both directions. The proposed lattice based framework utilizes summary structures and is generative in the sense that it first generate candidate forms for variants and then estimate the number of strings that match any of the candidate forms. The proposed framework has opportunities for supporting more diverse measure by changing the way candidate forms are generated. Random sampling has several benefits [9] that make it a good candidate for selectivity estimation technique for approximate predicates. For instance, it is not limited to equality and range predicates and does not use the attribute-value-independence assumption. However, it has several additional challenges when applied to similarity matching (Chapter 6) and has to be applied with caution. We looked into the possibility of using random sampling for this purposes in Chapter 6. This direction is worth further investigation.

7.2.2 Adaptation for Query Processing

Although the proposed framework is for selectivity estimation, some concepts have opportunities for approximate query processing as well. The basic idea is to associate, instead of a count, a tuple id list for each entries in the EQ-gram table or signature patterns. For instance the entry ‘rdb??’ will have a list of tuple ids which have a substring that matches ‘rdb??’. The same idea can also be applied to signature patterns. Intuitively, we are pre-computing and storing tuple ids with similar strings, and it is a form of trade-offs between time and space, or build-time and runtime costs. As a tuple may be represented in more than one lists, it can easily blow up the space and may need special attention.

7.2.3 Integration with Full-Fledged RDBMSs

Another next step from this dissertation could be evaluation of the proposed techniques in a full-fledged RDBMS. Commercial database systems are conservative in integrating new techniques into their engines since real-world use of RDBMSs can be very complex and large scale. Many components interact and any integration has to go through a strict validation. For this reason, many research studies are yet to be incorporated into RDBMS engines. For example, even though there have been many studies on multi-dimensional histograms (e.g., [19, 112, 121, 141]) or parametric query optimization (e.g., [14, 67, 72]), most of commercial database systems still take much simpler approaches.

Validating the proposed techniques in a full-fledged RDBMS can open more issues or demand improvements. For instance, can we make it more space efficient or can we better prune the proposed summary structures given a space budget? Would it work well in much larger analytic queries? Are building and maintenance costs acceptable in an operational
RDBMS? This dissertation has addressed these questions to some extent and more through validation can be done by testing them in a full-fledged RDBMS.

7.2.4 Incremental Maintenance of Summary Structures

Another crucial factor in operational RDBMSs is incremental maintenance of data structures such as histograms, and indexes. In general, the summary structures proposed in this dissertation can be incrementally maintained in a rather straightforward way. The EQ-gram table in Chapters 3 and 4 can be maintained by incrementing counts (resp. decrementing counts) of q-grams of the inserted string (resp. the deleted string).

Maintaining an EQ-gram table with Min-Hash signatures is easy with insertions but can be tricky with deletions. When a string is deleted, for each of the q-grams of the string, we compare the Min-Hash values of the string with existing signature values. We do not need to modify the existing signature value when the Min-Hash value of the string is larger. However, when the Min-Hash value (of the string to be deleted) is the same as the signature value, we need to find a new Min-Hash value among the rest of strings that contain the q-gram. This may require a database scan. We can perform periodic updates tolerating some degree of “staleness”, but a more principled approach is desirable.

The Min-Hash signature representation of the database in Chapter 5 does not have the problem since we can simply add or delete the corresponding signature. The maintenance of the LSH index in Chapter 6 is also simple. As records are inserted or deleted, we only need to add or remove them in corresponding buckets [12].

In an update-intensive database, updating the data structures on-the-fly can be a bottleneck and may not be desirable. In such cases, sampling or periodic updates may be the choice, e.g., [5, 50, 139]. More detailed studies on this direction are also worth further investigation.
Bibliography


[132] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. Winnowing: local algorithms for


[134] Rajendra Shinde, Ashish Goel, Pankaj Gupta, and Debojyoti Dutta. Similarity Search
and Locality Sensitive Hashing using Ternary Content Addressable Memories. In Proceed-
ings of the ACM SIGMOD International Conference on the Management of Data,

International Conference on Research and Development in Information Retrieval, pages

[136] Manolis Terrovitis, Spyros Passas, Panos Vassiliadis, and Timos Sellis. A Combination of
Trie-trees and Inverted Files for the Indexing of Set-valued Attributes. In Proceedings
of the International Conference on Information and Knowledge Management, pages


[139] Jeffrey S. Vitter. Random sampling with a reservoir. ACM Transactions on Mathe-


[141] Hai Wang and Kenneth C. Sevcik. A multi-dimensional histogram for selectivity esti-
mation and fast approximate query answering. In Proceedings of the 2003 conference of
the Centre for Advanced Studies on Collaborative research, pages 328–342, 2003.

[142] Min Wang, Jeffrey Scott Vitter, and Bala Iyer. Selectivity Estimation in the Presence
of Alphanumeric Correlations. In Proceedings of the IEEE International Conference on

[143] Wei Wang, Chuan Xiao, Xuemin Lin, and Chengqi Zhang. Efficient approximate en-
tity extraction with edit distance constraints. In Proceedings of the ACM SIGMOD

mane Study for Similarity-Search Methods in High-Dimensional Spaces. In Proceedings


