KEYWORD SEARCH ON LARGE-SCALE DATA: FROM RELATIONAL AND GRAPH DATA TO OLAP INFRASTRUCTURE

by

Bin Zhou
M.Sc., Simon Fraser University, 2007
B.Sc., Fudan University, 2005

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
in the School
of
Computing Science

SIMON FRASER UNIVERSITY
Summer 2011

APPROVAL

Name:	Bin Zhou
Degree:	Doctor of Philosophy
Title of Thesis:	Keyword Search on Large-Scale Data: from Relational and Graph Data to OLAP Infrastructure

Examining Committee: Dr. Ke Wang
Chair

Dr. Jian Pei, Senior Supervisor
Associate Professor of Computing Science, SFU

Dr. Wo-Shun Luk, Supervisor
Professor of Computing Science, SFU

Dr. Martin Ester, SFU Examiner
Professor of Computing Science, SFU

Dr. Jeffrey Xu Yu, External Examiner
Professor of Systems Engineering and Engineering Management, The Chinese University of Hong Kong

Date Approved: 6 May 2011

Abstract

In recent years, the great success of Web search engines has shown the simplicity and the power of keyword search on billions of textual pages on the Web. In addition to textual pages, there also exist vast collections of structured, semi-structured, and unstructured data in various applications that contain abundant text information. Due to the simplicity and the power of keyword search, it is natural to extend keyword search to retrieve information from large-scale structured, semi-structured, and unstructured data.

In this thesis, we study a class of important challenging problems centered on keyword search on large-scale data. We propose various techniques for different types of important data sources, including relational tables, graphs, and search logs. Specifically, for relational tables, we show that, while searching individual tuples using keywords is useful, in some scenarios, it may not find any tuples since keywords may be distributed in different tuples. To enhance the capability of the keyword search technique on relational tables, we develop the *aggregate keyword search* method which finds aggregate groups of tuples jointly matching a set of query keywords. For graphs, we indicate that keyword queries are often ambiguous. Thus, developing efficient and effective query suggestion techniques is crucial to provide satisfactory user search experience. We extend the *query suggestion* technique in Web search to help users conduct keyword search on graphs. For search logs, we study various types of keyword search applications in Web search engines, and conclude that all of those applications are related to several novel mining functions on search logs. We build a universal *OLAP infrastructure* on search logs which supports scalable online query suggestion.

The proposed techniques have several desirable characteristics which are useful in different application scenarios. We evaluate our approaches on a broad range of real data sets and synthetic data sets and demonstrate that the techniques can achieve high performance. We also provide an overview of the keyword search problem on large-scale data, survey the literature study in the field, and discuss some potential future research directions.

Keywords: keyword search; aggregate keyword search; query suggestion; OLAP

To my family

"I like the dreams of the future better than the history of the past."

— Thomas Jefferson*(1743 – 1826)*

Acknowledgments

My foremost thank goes to my supervisor and mentor Dr. Jian Pei. I thank him for his patience and encouragement that carried me on through difficult times, and for his insights and suggestions that helped to shape my research skills and critical thinkings. His valuable feedback contributed greatly to this thesis. As an advisor, he taught me practices and skills that I will use in my future career.

I am very thankful to my supervisor, Dr. Wo-Shun Luk, for providing insightful comments and helpful suggestions that helped me to improve the quality of the thesis. His visionary thoughts and energetic working style have influenced me greatly.

Part of this work is done in collaboration with Dr. Daxin Jiang and Dr. Hang Li at Microsoft Research Asia. I thank them for the knowledge and skills they imparted through the close collaborations. Working with them is always so enjoyable. My gratitude and appreciation also goes to Dr. Martin Ester, Dr. Jeffrey Xu Yu and Dr. Ke Wang for serving on the thesis examining committee. I thank them for advising me and helping me in various aspects of my research, and their precious time reviewing my work.

I also want to thank Dr. Joseph G. Peters for providing me insightful suggestions and great supervision during my Master's study at Simon Fraser University.

I would also like to thank many people in our department, support staff and faculty, for always being helpful over the years. I thank my friends at Simon Fraser University for their help. A particular acknowledgement goes to Xu Cheng, Yi Cui, Yi Han, Ming Hua, Jin Huang, Bin Jiang, Mag Lau, Luping Li, Zhenhua Lin, Hossein Maserrat, Brittany Nielsen, Guanting Tang, Kathleen Tsoukalas, Feng Wang, Haiyang Wang, Raymond Chi-Wing Wong, Crystal Xing, Ji Xu, Xinghuo Zeng, Geoffrey Zenger.

Last but not least, I am very grateful to my dearest wife, my parents and my sister for always being there when I needed them most, and for supporting me continuously through all these years. I hope I will make them proud of my achievements, as I am proud of them. Their love accompanies me wherever I go.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Keyword search is a type of search that looks for matching documents which contain one or more keywords specified by a user [73]. Recently, the great success of Web search engines such as Google and Microsoft's Bing has shown the simplicity and the power of keyword search on large collections of Web documents. Typically, a user needs to submit to a Web search engine a keyword query which contains one or more keywords. The Web search engine then searches the pools of indexed Web documents and returns a ranked list of matching documents to the user. The simplicity and the power of keyword search have made it the most popular way for millions of users to retrieve information on the Web through Web search engines.

In the current information era, in addition to textual Web documents, there also exist vast collections of structured, semi-structured, and unstructured data in various applications that contain abundant text information. Typical examples of those data sources include relational tables, XML data, labeled graphs, and so on. Traditionally, in order to retrieve information from those data sources, a user has to learn some specific structured query languages, such as SQL and XQuery [32]. In some particular scenarios such as forming a SQL query in a relational database or issuing an XQuery over large-scale XML data, a user also needs to be familiar with the underlying data schemas. The data schemas may be very complex and fast evolving. As a result, it is impractical for those general users to look up information using the traditional methods based on structured query languages.

Due to the simplicity and the power of keyword search, it is natural to extend keyword search to retrieve information from large-scale structured, semi-structured, and unstructured data. Thus, developing effective and efficient keyword search techniques for different types

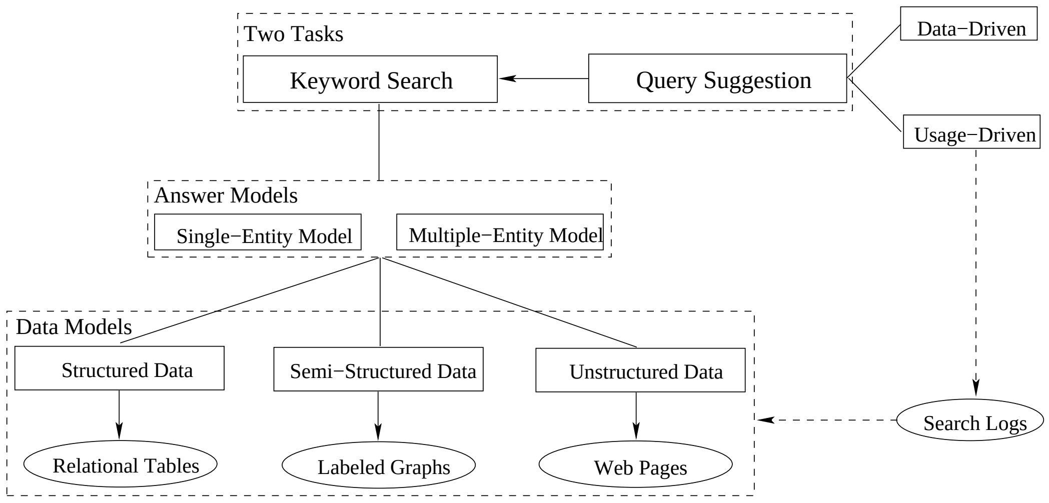


Figure 1.1: Overview of the keyword search problem.

of data sources is an important and interesting research problem [4, 14, 79, 13, 15, 86, 87].

1.1 Motivation and Problem Overview

Many studies about keyword search in various aspects have been conducted in recent years. Figure 1.1 provides an overview of the keyword search problem on large-scale data.

Given users' keyword queries, the core task in the keyword search problem is finding related and meaningful answers from different data sources. To understand the problem well, it is necessary to first understand the different data models and answer models.

In terms of the data models, keyword search can be applied not only on unstructured data (e.g., Web documents), but also on structured data (e.g., relational tables) and semi-structured data (e.g., XML data and labeled graphs). For instance, the problem of keyword search on labeled graphs tries to assemble semantically relevant connections (modeled as a subgraph) among keywords in the query [39, 45, 23, 20]. As another example, the problem of keyword search on relational tables focuses on finding individual tuples matching a set of query keywords from one table or the join of multiple tables [3, 10, 41, 52, 54, 61, 62].

In different data models, the answers to keyword search are quite different. Without loss of generality, the answer models of keyword search can be roughly classified into two categories: the single-entity model and the multiple-entity model. In the single-entity answer model, all the keywords in a query must appear together in a single entity (e.g., a Web document, a tuple in a relation table, or a node in a labeled graph). However, in the multiple-entity answer model, the keywords in a query can be distributed into several entities which are closely related.

In the keyword search problem, another core task is providing related queries as suggestions to assist users' search process. With the assistance of keyword search, both experienced users and inexperienced users can easily retrieve valuable information from large-scale structured, semi-structured, and unstructured data. However, in most cases, a keyword query typically is very short, which on average only contains 2 or 3 keywords [77]. Such short keyword queries may not be able to precisely capture users' information needs. As a result, keyword queries in practice are often ambiguous. To improve the effectiveness of keyword search, most popular commercial Web search engines currently provide a useful service called *query suggestion*. By conjecturing a user's real search intent, the Web search engines can recommend a set of related queries which may better reflect the user's true information need.

To provide meaningful recommendations, query suggestion has to take into account different types of information sources. Generally, all the existing query suggestion methods for keyword search can be roughly classified into either data-driven methods or usage-driven methods. The data-driven query suggestion methods utilize the data itself. They apply data mining techniques to identify the correlations among data, and select closely related queries as recommendations. The usage-driven query suggestion methods focus on the search activities. For example, search logs trace users' search behavior. In Web search engines, a commonly used query suggestion method [5] is to find common queries following the current user query in search logs and use those queries as suggestions for each other.

As keyword search is a useful tool to retrieve information from large-scale structured, semi-structured, and unstructured data, developing efficient and effective keyword search techniques on different data sources has been an important research direction in data mining, database systems, Web Search and information retrieval.

Although keyword search has attracted much attention in recent years, there still exist several challenging problems which remain unsolved.

- *How can we enhance the capability of the keyword search technique*? Recently, keyword search has been applied successfully on relational databases where some text attributes are used to store text-rich information. As reviewed in Chapter 2, all of the existing methods focus on finding a set of tuples that are most relevant to a given keyword query. Here, each tuple in the answer set may be from one table or from the join of multiple tables. Generally, the existing methods try to identify interesting connections using primary-foreign key relationships among those tuples which match the query keywords. While searching matching tuples using keywords is useful, in some scenarios, it may not find any tuples, since keywords may be distributed into a set of different tuples without primary-foreign key relationships. To improve the capability of keyword search on relational databases, is there any way to obtain interesting information even though the keywords appear in a set of different tuples in the relational databases (e.g., different types of connections among those keywords and tuples)?

- *How can we extend the query suggestion technique to various scenarios*? As analyzed before, keyword queries are often ambiguous. Query suggestion has been shown an important and powerful tool for the Web search engines to achieve good search performance. Although the problem of keyword search on different data sources has been studied for several years, and several prototype systems supporting keyword search have been developed, as far as we known, there is no previous work on systematically analyzing query suggestion methods for keyword search on different data sources, such as relational databases and graphs. As relational databases and graphs carry rich information, it is very essential to explore the data-driven based recommendation. Thus, can we develop any methods to derive effective data-driven query suggestions to assist users' search process on relational databases and graphs?

- *How can we strengthen the scalability of the query suggestion technique*? Search logs, which contain rich and up-to-date information about users' needs and preferences, are inherently an indispensable component in keyword search. Search logs have become a critical data source to analyze users' search behavior, which in turns have many potential applications, such as usage-driven query suggestion. On one hand, analyzing large-scale search logs requires a system infrastructure with high efficiency. On the other hand, given that there are many different applications using search logs, how many specific search log analysis tools do we have to build? Obviously, building one specific search log analysis tool per application is neither effective nor efficient. Can we develop a search log analysis infrastructure supporting the essential needs of many different applications efficiently?

This thesis tries to make good progress in answering the above questions.

1.2 Contributions

In this thesis, we study a class of important challenging problems centered on keyword search on large-scale data. The proposed techniques are designed for different types of important data sources, including relational tables, graphs, and search logs. In particular, we make the following contributions.

- For relational tables, we systematically develop the *aggregate keyword search* method so as to enhance the capability of the keyword search technique. In particular, we conduct a *group-by*-based keyword search. We are interested in identifying a minimal context where all the keywords in a query are covered. We further extend our methods to allow partial matches and matches using a keyword ontology.
- For graphs, we identify the importance of query suggestion for keyword search on graphs, and propose a practical solution framework. We develop efficient methods to recommend keyword queries for keyword search on graphs. The general idea is to cluster all the valid answers, and recommend related queries from each cluster. We develop a hierarchical decomposition tree index structure to improve the performance of query suggestion.
- For search logs, we study various types of keyword search applications in Web search engines, and conclude that all of those applications are related to three novel mining functions on search logs, namely *forward search*, *backward search*, and *session retrieval*. We build a universal OLAP infrastructure on search logs which supports online query suggestion. To build the OLAP infrastructure, we develop a simple yet scalable and distributable index structure.

Although the techniques are developed for different types of data sources, the problems we studied in this thesis are highly related. Firstly, the techniques we used for relational tables and search logs can be categorized as OLAP. For relational tables, we conduct a *group-by*-based keyword search. For search logs, we design OLAP operations on search logs. Secondly, the results returned by different techniques can be used for recommendation. For relational tables, the aggregate keyword search method can identify a minimal context (e.g., attribute values) where the keywords in a query are covered. Some related information in the same context can be recommended to the users. For graphs and search logs, the proposed techniques are used to recommend meaningful keyword queries to the users.

Some major components of this thesis were published as journal and conference papers. Particularly, the major results in Chapter 3 were published in [89, 90] and the major results in Chapter 5 were published in [88]. Part of the contents in Chapter 2 were published in [89, 90, 88, 68].

1.3 Organization of the Thesis

The remainder of the thesis is structured as follows:

- In Chapter 2, we review the related work and explain how they are related to and different from this thesis.
- In Chapter 3, we focus on relational tables. A novel keyword search method, *aggregate keyword search*, is developed. The efficiency and the effectiveness of aggregate keyword search are verified by both theoretical analysis and experimental evaluations.
- Query suggestion is a very useful tool to improve the effectiveness of keyword search. However, to the best of our knowledge, there are no previous studies about developing query suggestion methods on labeled graphs. In Chapter 4, we take an initiative step towards addressing the problem of query suggestion for keyword search on graphs.
- Search logs are inherently an indispensable output for the keyword search process. Search logs contain crowd intelligence accumulated from millions of users. Using search logs, we may develop a good variety of useful keyword search applications. In Chapter 5, we propose an OLAP system on search logs which serves as an infrastructure to support many different applications in Web search engines.
- The thesis concludes in Chapter 6. Some interesting extensions and future research directions of keyword search are also discussed.

Extensive experimental results are presented in Chapters 3, 4, and 5 to demonstrate that the techniques we proposed can achieve high performance.

Chapter 2

Related Work

In this chapter, we discuss two major areas of research work related to this thesis, and also briefly point out their relationships to our studies and the significant differences.

2.1 Keyword Search

Keyword search is a well-studied problem in the world of text documents and Web search engines [73]. The Informational Retrieval (IR) community has utilized the keyword search techniques for searching large-scale unstructured data, and has developed various techniques for ranking query results and evaluating their effectiveness [6, 56]. Meanwhile, the Database (DB) community has mostly focused on large-collections of structured data, and has designed sophisticated techniques for efficiently processing structured queries over the data [32]. In recent years, emerging applications such as customer support, health care, and XML data management require high demands of processing abundant mixtures of structured and unstructured data. As a result, the integration of Databases and Information Retrieval technologies becomes very important [4, 14, 79].

Keyword search provides great flexibility for analyzing both structured and unstructured data that contain abundant text information. In this section, we summarize some representative studies in different research areas including Information Retrieval, Databases, and the integration of Databases and Information Retrieval.

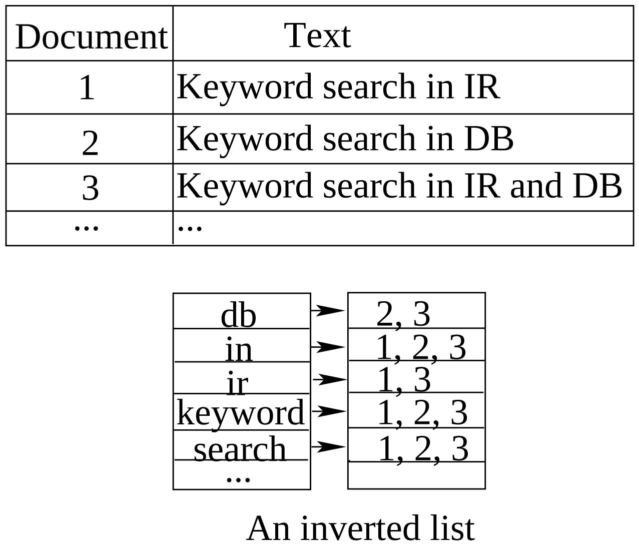


Figure 2.1: An example of the inverted list.

2.1.1 Keyword Search in Information Retrieval

In Information Retrieval, keyword search is a type of search method that looks for matching documents which contain one or more keywords specified by a user [73]. The *Boolean retrieval model* [56] is one of the most popular models for information retrieval in which users can pose any keyword queries in the form of a Boolean expression of keywords, that is, keywords are combined with some Boolean operators such as AND, OR, and NOT. The Boolean retrieval model views each document as just a set of keywords. A document either matches or does not match a keyword query.

Inverted lists are commonly adopted as the data structure for efficiently answering various keyword queries in the Boolean retrieval model [6, 56, 92, 93, 38]. The basic idea of an inverted list is to keep a dictionary of keywords. Then, for each keyword, the index structure has a list that records which documents the keyword occurs in. Figure 2.1 shows a simple example of the inverted list for a set of documents.

In the case of large document collections, the resulting number of matching documents using the Boolean retrieval model can far more than what a human being could possibly scan through. Accordingly, it is essential for a search system to rank the documents matching a keyword query properly. This model is called *ranked retrieval model* [56]. The vector space model [74] is usually adopted to represent the documents and the keyword queries. The relevance of a document with respect to a keyword query can be measured using the well-known Cosine similarity [72].

An important and necessary post-search activity for keyword search in Information Retrieval is the ranking of search results [6, 48, 59]. In general, the ranking metrics take into account two important factors. One is the relevance between a document and a keyword query. The other is the importance of the document itself. The term-based ranking and the link-based ranking are the two most popular ranking methods used widely in practice.

The term-based ranking methods, such as TFIDF [6], captures the relevance between documents and keyword queries based on the content information in the documents. A document d and a keyword query q can be regarded as sets of keywords, respectively. The TFIDF score of a document d with respect to a keyword query q is defined as $TFIDF(d, q) = \sum_{t \in d \cap q} TF(t) \times IDF(t)$, where $TF(t)$ is the term frequency of keyword t in d, and $IDF(t)$ is the inverse document frequency of keyword t which is the total number of documents in the collections divided by the number of documents that contain t.

The link-based ranking methods, such as PageRank [59] and HITS [48], are widely adopted by Web search engines for ranking the search results. The link structure of the Web can be viewed as a directed graph in which each node is a Web page, and each directed edge is a hyperlink between two pages. The link-based ranking methods assign scores to Web pages to reflect their importance. In PageRank [59], each page on the Web has a measure of page importance that is independent of any information need or keyword query. Roughly speaking, the importance of a page is proportional to the sum of the importance scores of pages linking to it. In HITS [48], a query is used to select a subgraph from the Web. From this subgraph, two kinds of nodes are identified: authoritative pages to which many pages link, and hub pages that contain comprehensive collections of links to authoritative pages.

2.1.2 Keyword Search in Databases

Relational databases are widely used in practice. Recently, the burst of the Internet has given rise to an ever increasing amount of text data associated with multiple attributes. For instance, `customer reviews` in a laptop shopping website are always associated with attributes like `manufacturer`, `brand`, `CPU`, and `OS`. Such data can be easily maintained using a relational table where specific text attributes (e.g., `review`) are used to store the text data.

The database management systems, also known as DBMS (e.g., Oracle, Microsoft's SQL

Server, MySQL), utilize a built-in full-text search engine to retrieve the tuples that contain keywords in some text attributes [71, 75, 70]. To make sure that full-text queries can be applied on a given table in a database, the database administrator has to create a full-text index on the table in advance. The full-text index includes one or more text attributes in the table. The results to the full-text queries are individual tuples in the corresponding relational databases.

In recent years, the Boolean retrieval model [6, 56, 92, 93, 38] and natural language processing techniques have been integrated into the full-text functionalities in those database management systems.

2.1.3 Keyword Search in Databases & Information Retrieval

The integration of Databases and Information Retrieval provides flexible ways for users to query information using keyword search. A few critical challenges have been identified such as how to model the query answers in a semantic way and how to address the flexibility in scoring and ranking models. To obtain more details in this direction, survey studies in [4, 14, 79, 13, 15, 86, 87] provide excellent insights into those issues.

As a concrete step to provide an integrated platform for text-rich and data-rich applications, keyword search on relational databases becomes an active topic in database research. Several interesting and effective solutions and prototype systems have been developed [3, 41, 10, 39, 54, 66].

For instance, DBXplorer [3] is a keyword-based search system implemented using a commercial relational database and Web server. DBXplorer returns all rows, either from individual tables or by joining multiple tables using primary-foreign keys, such that each row contains all keywords in a query. It uses a symbol table as the key data structure to look up the respective locations of query keywords in the database. DISCOVER [41] produces without redundancy all joining networks of tuples on primary and foreign keys, where a joining network represents a tuple that can be generated by joining some tuples in multiple tables. Each joining network collectively contains all keywords in a query. Both DBXplorer and DISCOVER exploit the schema of the underlying databases. Hristidis *et al.* [40] develop efficient methods which can handle queries with both AND and OR semantics and exploit ranking techniques to retrieve top-k answers.

Quite a few studies about keyword search on relational databases model the database as a graph. BANKS [10] models a database as a graph where tuples are nodes and application-oriented relationships are edges. Under such an extension, keyword search can be generalized on trees and graphs. BANKS searches for minimum Steiner trees [25] that contain all keywords in the query. Some effective heuristics are exploited to approximate the Steiner tree problem, and thus the algorithm can be applied to huge graphs of tuples. To improve the search efficiency on large graphs, Kacholia *et al.* [45] introduce the bidirectional expansion techniques to improve the search efficiency on large graph databases. Dalvi *et al.* [20] study the problem of keyword search on graphs which are significantly large to be held in main memory. To provide keyword search on graphs which are stored in external memory, Dalvi *et al.* [20] build a graph representation which combines both condensed version of the graph and the original graph together. The resulting graph representation is always memory resident.

Because keyword search on relational databases and graphs takes both vertex labels and graph structures into account, there are many possible strategies for ranking answers. Different ranking strategies reflect designers' respective concerns. Various effective IR-style ranking criteria and search methods are developed, such as those studies in [52, 54, 23, 49, 15, 62].

Most of the previous studies concentrate on finding minimal connected trees from relational databases and graphs [3, 41, 40, 10, 46, 52, 47, 54, 23]. Recently, several other types of semantics of query answers have been proposed. Under the graph model of a relational database, BLINKS [39] proposes to find distinct roots as answers to a keyword query. An m-keyword query finds a collection of tuples that contain all the keywords reachable from a root tuple within a user-given distance. BLINKS [39] builds a bi-level index for fast keyword search on graphs. Recently, Qin *et al.* [62] model a query answer as a community. A community contains several core vertices connecting all the keywords in the query. Li *et al.* [49] study keyword search on large collections of heterogenous data. An r-radius Steiner graph semantic is proposed to model the query answers. Later on, Qin *et al.* [61] consider all the previous semantics of query answers, and show that the current commercial database management systems are powerful enough to support keyword queries in relational databases efficiently without any additional indexing to be built and maintained.

Instead of returning subgraphs that contain all the keywords, ObjectRank [7] returns individual tuples as answers. It applies a modified PageRank algorithm to keyword search in a database for which there is a natural flow of authority between the objects (e.g., tuples in relational databases). To calculate the global importance of an object, a random surfer has the same probability to start from any object in the base set [67]. ObjectRank returns objects which have high authority with respect to all query keywords.

The quality of approximation in top-k keyword proximity search is studied in [47]. Yu *et al.* [85] use a keyword relationship matrix to evaluate keyword relationships in distributed databases. Most recently, Vu *et al.* [78] extend the work [85] by summarizing each database using a keyword relationship graph, which can help to select top-k most promising databases effectively in query processing.

How Is Our Study Different?

The existing studies about keyword search on relational databases and our work in Chapter 3 address different types of keyword queries on relational databases. First, previous studies focus on relational databases which consist of a set of relational tables. In Chapter 3, an aggregate keyword query is conducted on a single large relational table. Second, previous studies look for *individual tuples* (or a set of tuples interconnected by primary-foreign key relationships) such that all the query keywords appear in at least one returned tuple in the database. In other words, the answers to be returned are tuples from the original database. However, in Chapter 3, the answers to the aggregate keyword queries are aggregate cells (group-bys). All the query keywords must appear in at least one tuple in the corresponding group-bys. In other words, the answers to be returned are aggregated results from the original relational table. Third, in the previous studies, the tuples to be returned come from different relational tables which are connected by the primary-foreign key relationships. However, in Chapter 3, the tuples are in the same table and those tuples in the same group-by have same values on those group-by attributes. As a result, the problem of aggregate keyword query is quite different from those existing studies about keyword search on relational databases.

The existing methods about keyword search on relational databases cannot be extended straightforwardly to tackle the aggregate keyword search problem in Chapter 3. First, previous studies find tuples from different tables which are interconnected by primary-foreign key relationships, it is possible to extend some of the existing methods to compute those joining networks where tuples from the same table are joined (that is, self-join of a table). However, for a query of m keywords, we need to conduct the self-joins of the original table for $m - 1$ times, which is very time-consuming. Second, the number of joining networks generated by the self-joins can be much larger than the number of valid answers due to the monotonicity of aggregate cells. Such extensions cannot compute the valid answers to aggregate keyword queries efficiently. As a result, we need to develop specific query answering techniques to conduct the aggregate keyword queries efficiently and effectively.

2.2 Query Suggestion

In those popular commercial Web search engines, query suggestion has become a well-accepted functionality to assist the users to explore and formulate their precise information needs during the search process. The objective of query suggestion in search engines is transforming an initial search query into a better one which is capable of satisfying the users' real information need by retrieving more relevant documents. There are quite a lot of studies conducted on generating different types of query suggestions, such as query auto-completion [17], query spelling correction [51], query expansion [80], and query rewriting [44]

To provide a good query suggestion, it is necessary to understand the users' precise search intents behind search queries. Generally, additional information rather than search queries should be taken into consideration for query understanding. These pieces of information include users' explicit feedbacks (e.g., [55]), users' implicit feedbacks (e.g., [12, 11]), users' personal profiles (e.g., [16]), search result snippets (e.g., [64]), and so on.

Search logs contain crowd intelligence accumulated from millions of users, that is, a large number of people simultaneously converge upon the same point of knowledge. As such, search logs recently have been widely used for the purpose of query suggestion. For instance, Cui *et al.* [19] extract probabilistic correlations between query keywords and document terms by analyzing search logs and used the correlations to select high-quality expansion terms for new queries. Jones *et al.* [44] identify typical substitutions the users made to their queries from search logs, and leverage the information to improve the quality of user queries. Recently, search context has been identified as one piece of important source to understand users' search behavior. For instance, Cao *et al.* [12, 11] propose a general context-aware model for query suggestion and ranking. These studies indicate that search contexts are effective for disambiguating keyword queries and improving the quality of multiple search services.

Most recently, the query suggestion technique has attracted some attention for keyword search on structured and semi-structured data. Zhou *et al.* [91] propose a query relaxation scheme to deal with imprecise document models and heterogeneous schemas. The major idea is to utilize duplicates in co-related data sets to identify important correlations. Then, these correlations are used to appropriately relax users' search queries. Recently, Li *et al.* [50] study a keyword search problem on XML data by mining promising result types. Li *et al.* [50] claim that keyword query is hard to be precise. As a result, the number of returned answers may be too huge, and only a few types of them are interesting to the users. With the aim to help users disambiguate possible interpretations of the query, Li *et al.* [50] propose a ranking methods by taking into account the query results for different interpretation. Only the most promising result types are returned.

How Is Our Study Different?

The methods of query suggestion in Web search engines cannot be extended straightforwardly to solve the problem in Chapter 4. First, the answer models for Web search and search on graphs are completely different. In Web search, an answer to a keyword query is a textual Web page. However, when conducting keyword search on graphs, an answer to a keyword query can have many different types of semantics, such as a Steiner tree or a root node. Second, the ranking models for Web search and search on graphs are not the same. In Web search, the set of textual Web pages are ranked based on their relevance to the query. For keyword search on graphs, the answers are ranked according to specific ranking scores, such as the weight of the answer. Third, many pieces of information in Web search, especially search logs, can be used to analyze users' search activities. However, when conducting keyword search on graphs, we do not have rich sources to understand users' search behavior. Those critical differences make the problem of query suggestion for keyword search on graphs more challenging.

The problem studied in [50] is quite different from our study in Chapter 4. First, the work in [50] focuses on XML data which is a tree structure. However, our work in Chapter 4 focuses on general graphs. Thus, the methods in [50] cannot be used to solve our problem in Chapter 4. Second, while the work in [50] identifies most promising result types, the goal of our work in Chapter 4 is to recommend a set of related queries to assist users' search process. As far as we know, our work in Chapter 4 is the first systematic study on providing query suggestions for keyword search on general graphs.

Many previous studies have been conducted on analyzing search logs for various keyword search applications such as query suggestion and keyword bidding. However, as far as we know, there is no previous work on building a general search log analysis infrastructure to support various online applications in Web search engines. In Chapter 5, we present an initiative towards building an *online analytic processing* (OLAP for short) system on search logs as the universal search log analysis infrastructure.

Chapter 3

Aggregate Keyword Search on Relational Tables

Relational databases are widely used in practice. Recently, keyword search has been applied successfully on relational databases where some text attributes are used to store text-rich information. As reviewed in Chapter 2, all of the existing methods address the following search problem: given a set of keywords, find a set of tuples that are most relevant (for example, find the top-k most relevant tuples) to the set of keywords. Here, each tuple in the answer set may be from one table or from the join of multiple tables.

While searching individual tuples using keywords is useful, in some application scenarios, a user may be interested in an *aggregate group of tuples* jointly matching a set of query keywords. In this chapter, we motivate a novel problem of aggregate keyword search: finding minimal group-bys covering a set of query keywords well, which is useful in many applications.

In this chapter, we first provide a motivation example to illustrate the usage of aggregate keyword search in Section 3.1. Then, in Section 3.2, we formulate the problem of aggregate keyword search on relational databases, and discuss some significant differences between our work and some others. We develop the maximum join approach in Section 3.3, and the keyword graph approach in Section 3.4. In Section 3.5, the complete aggregate keyword search is extended and generalized for partial matching and matching based on a keyword ontology. A systematic performance study is reported in Section 3.6. Section 3.7 summarizes this chapter.

Month	State	City	Event	Description
December	Texas	Houston	Space Shuttle Experience	rocket, supersonic, jet
December	Texas	Dallas	Cowboy's Dream Run	motorcycle, culture, beer
December	Texas	Austin	SPAM Museum Party	classical American Hormel foods
November	Arizona	Phoenix	Cowboy Culture Show	rock music

Table 3.1: A table of tourism events.

3.1 A Motivation Example

Example 1 (Motivation) *Table 3.1 shows a database of tourism event calendar. Such an event calendar is popular in many tourism Websites and travel agents' databases (or data warehouses). To keep our discussion simple, in the field of* `description`*, a set of keywords are extracted. In general, this field can store text description of events.*

Scott, a customer planning his vacation, is interested in seeing space shuttles, riding motorcycle and experiencing American food. He can search the event calendar using the set of keywords {"space shuttle", "motorcycle", "American food"}. Unfortunately, the three keywords do not appear together in any single tuple, and thus the results returned by the existing keyword search methods may contain at most one keyword in a tuple.

However, Scott may find the aggregate group (December, Texas, $*$, $*$, $*$) interesting and useful, since he can have space shuttles, motorcycle, and American food all together if he visits Texas in December. The $*$ signs on attributes `city`*,* `event`*, and* `description` *mean that he will have multiple events in multiple cities with different descriptions.*

To make his vacation planning effective, Scott may want to have the aggregate as specific as possible – it should cover a small area (for example, Texas instead of the whole United States) and a short period (for example, December instead of year 2009).

In summary, the task of keyword search for Scott is to find minimal aggregates in the event calendar database such that for each of such aggregates, all keywords are contained by the union of the tuples in the aggregate. ■

Different from the existing studies about keyword search on relational databases which find a tuple (or a set of tuples interconnected by primary-foreign key relationships) matching the requested keywords well, the aggregate keyword search investigated in this chapter tries to identify a minimal context where the keywords in a query are covered. In addition to the example of the tourism event query in Example 1, there are quite a few interesting applications of the aggregate keyword queries in practice.

- For movie retailer stores, their databases store all the information about thousands of movies, such as `title` of the movie, `director`, `leading actor`, `leading actress`, `writer`, and so on. The aggregate keyword queries can suggest to the customers some interesting factors of movies that customers may find interesting. For example, a customer favoring in science fiction movies may be suggested that the director `Steven Spielberg` has directed quite a few sci-fi movies. Later on, the customer may want to watch some more movies directed by Steven Spielberg.
- For online social networking sites, their databases store all the information about different user communities and groups, such as `member's location`, `member's interest`, `member's gender`, `member's age`, and so on. The aggregate keyword queries can recommend to those new users some important factors of those social communities that they may prefer to joining in. For example, a female university student may find that most people in her age likes `shopping`. Thus, she may want to first join those communities which focus on shopping information.

Similar situations arise in some related recommendation services, such as restaurant recommendation, hotel recommendation, friend finder suggestion, shopping goods promotion, and so on.

As analyzed in Chapter 2, aggregate keyword search cannot be achieved efficiently using the keyword search methods developed in the existing studies, since those methods do not consider aggregate group-bys in the search which are critical for aggregate keyword search.

3.2 Aggregate Keyword Queries

For the sake of simplicity, in this chapter, we follow the conventional terminology in online analytic processing (OLAP) and data cubes [35].

Definition 1 (Aggregate cell) *Let $T = (A_1, \ldots, A_n)$ be a relational table. An* **aggregate cell** *(or a* **cell** *for short) on table T is a tuple $c = (x_1, \ldots, x_n)$ where $x_i \in A_i$ or $x_i = *$ $(1 \leqslant i \leqslant n)$, and $*$ is a meta symbol meaning that the attribute is generalized. The* **cover**

of aggregate cell c is the set of tuples in T that have the same values as c on those non-$*$ attributes, that is,

$$Cov(c) = \{(v_1, \ldots, v_n) \in T | v_i = x_i \text{ if } x_i \neq *, 1 \leqslant i \leqslant n\}$$

A **base cell** *is an aggregate cell which takes a non-$*$ value on every attribute.*

For two aggregate cells $c = (x_1, \ldots, x_n)$ and $c' = (y_1, \ldots, y_n)$, c is an **ancestor** *of c', and c' a* **descendant** *of c, denoted by $c \succ c'$, if $x_i = y_i$ for each $x_i \neq *$ $(1 \leqslant i \leqslant n)$, and there exists i_0 $(1 \leqslant i_0 \leqslant n)$ such that $x_{i_0} = *$ but $y_{i_0} \neq *$. We write $c \succeq c'$ if $c \succ c'$ or $c = c'$.* ■

For example, in Table 3.1, the cover of aggregate cell (December, Texas, $*$, $*$, $*$) contains the three tuples about the events in Texas in December. Moreover, ($*$, Texas, $*$, $*$, $*$) $\succ$ (December, Texas, $*$, $*$, $*$).

Apparently, aggregate cells have the following property.

Corollary 1 (Monotonicity) *For aggregate cells c and c' such that $c \succ c'$, $Cov(c) \supseteq Cov(c')$.* ■

For example, in Table 3.1, Cov($*$, Texas, $*$, $*$, $*$) $\supseteq$ Cov(December, Texas, $*$, $*$, $*$).

In this chapter, we consider keyword search on a table which contains some text-rich attributes such as attributes of character strings or large object blocks of text. Formally, we define an aggregate keyword query as follows.

Definition 2 (Aggregate keyword query) *Given a table T, an* **aggregate keyword query** *is a 3-tuple $Q = (\mathcal{D}, \mathcal{C}, W)$, where $\mathcal{D}$ is a subset of attributes in table T, $\mathcal{C}$ is a subset of text-rich attributes in T, and W is a set of keywords. We call $\mathcal{D}$ the* **aggregate space** *and each attribute $A \in \mathcal{D}$ a* **dimension**. *We call $\mathcal{C}$ the set of* **text attributes** *of Q. $\mathcal{D}$ and $\mathcal{C}$ do not have to be exclusive to each other.*

An aggregate cell c on T is an ***answer*** *to the aggregate keyword query (or c* **matches** *Q for short) if (1) c takes value $*$ on all attributes not in $\mathcal{D}$, that is, $c[A] = *$ if $A \notin \mathcal{D}$; and (2) for every keyword $w \in W$, there exists a tuple $t \in Cov(c)$ and an attribute $A \in \mathcal{C}$ such that w appears in the text of $t[A]$.* ■

Example 2 (Aggregate keyword query) *The aggregate keyword query in Example 1 can be written as Q = ({`Month`, `State`, `City`, `Event`}, {`Event`, `Description`}, {"space shuttle", "motorcycle", "American food"}) according to Definition 2, where table T is shown in Table 3.1.* ■

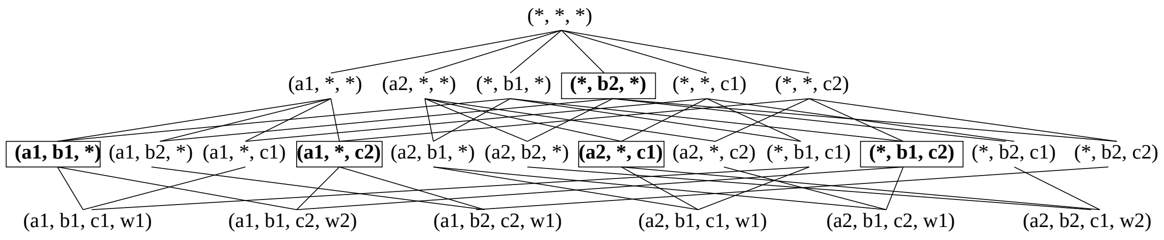


Figure 3.1: The aggregate lattice on ABC.

Due to the monotonicity of aggregate cells in covers (Corollary 1), if c is an answer to an aggregate keyword query, then every aggregate cell which is an ancestor of c (that is, more general than c) is also an answer to the query. In order to eliminate the redundancy and also address the requirements from practice that specific search results are often preferred, we propose the notion of minimal answers.

Definition 3 (Minimal answer) *An aggregate cell c is a* **minimal answer** *to an aggregate keyword query Q if c is an answer to Q and every descendant of c is not an answer to Q.*

The problem of aggregate keyword search *is to find the complete set of minimal answers to a given aggregate keyword query Q.* ■

In some situations, the minimal answers can be ranked according to a specific ranking function (e.g., TFIDF based ranking function). Users may be only interested in top-k answers. However, defining a meaningful ranking function in practice is a challenge. Moreover, in different relational databases, the ranking functions can be quite different, since the data stored in the databases may be quite different. In this chapter, we focus on finding the complete set of minimal answers. The problem of top-k answer retrieval is interesting for future research.

It is well known that all the aggregate cells on a table form a lattice. Thus, aggregate keyword search is to search the minimal answers in the aggregate cell lattice as illustrated in the following example.

Example 3 (Lattice) *In table $T = (A, B, C, D)$ in Table 3.2, attribute D contains a set of keywords w_i $(i > 0)$. Consider query $Q = (ABC, D, \{w_1, w_2\})$.*

A	B	C	D
a_1	b_1	c_1	w_1, w_3
a_1	b_1	c_2	w_2, w_5
a_1	b_2	c_2	w_1
a_2	b_1	c_1	w_1, w_4
a_2	b_1	c_2	w_1, w_3
a_2	b_2	c_1	w_2, w_6

Table 3.2: Table T in Example 3.

Figure 3.1 shows the aggregate cells in aggregate space ABC and the lattice. Aggregate cells $(a_1, b_1, *)$, $(a_1, *, c_2)$, $(*, b_1, c_2)$, $(*, b_2, *)$, and $(a_2, *, c_1)$ are the minimal answers to the query. They are highlighted in boxes in the figure. ■

In this chapter, for simplicity, we only consider two-level aggregations, that is, for an aggregate cell $c = (x_1, \ldots, x_n)$, x_i is either a value in the corresponding dimension A_i or an aggregated value $*$. The proposed aggregate keyword search can be extended to support multiple-level aggregations. For example, consider the dimension `Month` in Table 3.1, the values in this dimension could be organized into a hierarchy, such as `Month` – `Season` – `Year`. For an aggregate cell c, the value of the dimension `Month` can be a specific month such as `December`, or an aggregated value such as `Winter` or `2011`, or even a generalized value $*$. Then, the **cover** of an aggregate cell c contains a set of tuples satisfying the following two conditions: (1) for those non-aggregated attributes, those tuples have the same values as c; and (2) for those aggregated attributes, those tuples have the values which are descendants of the values of c in the corresponding dimension hierarchy. The definition of minimal answers can be extended similarly. However, this problem modeling introduces more challenges, since every dimension has its own hierarchy. We leave the detailed discussion of aggregate keyword search using multiple-level aggregations as an interesting future research direction.

It is also possible to consider whether the aggregate keyword search can be applied on multiple tables. Intuitively, we can conduct join operations on multiple tables to form a global table. Then, we can apply the aggregate keyword search on the global table. However, the joined table could be very huge. Moreover, the results of aggregate keyword search may contain unnecessary information. For example, consider two tables $T_1(A, B)$ and $T_2(B, C)$. T_1 has one tuple $t_{11} = (a_1, b_1)$, and T_2 has two tuples $t_{21} = (b_1, c_1)$ and $t_{22} = (b_2, c_1)$. We may find a minimal answer $c = (a_1, *, c_1)$ if we conduct joins on T_1 and T_2. However, both of the two tuples in T_2 are in the cover of c. But only t_{21} can be joined with t_{11} to obtain the minimal answer. Thus, it is interesting to explore whether some efficient and effective solutions are available to answer aggregate keyword queries on multiple tables.

Another interesting direction to explore in the future is the ranking of minimal answers. As shown in Table 3.4, the number of minimal answers to an aggregate keyword query could be very large. In some cases, users may be only interested in the top-k results where k is a small number specified by the user. There are some factors need to be considered for ranking minimal answers. For example, the number of tuples in the cover is useful to examine whether the minimal answer contains too much irreverent information. As another example, different dimensions may have different importance. The rankings of answers need to consider whether important dimensions are aggregated or not. We will develop detailed ranking measures in the future study.

3.2.1 Comparisons with Keyword-Driven Analytical Processing

Keyword-driven analytical processing (KDAP) [82] probably is the work most relevant to our study. KDAP involves two phases. In the differentiate phase, for a set of keywords, a set of candidate subspaces are generated where each subspace corresponds to a possible join path between the dimensions and the facts in a data warehouse schema (for example, a star schema). In the explore phase, for each subspace, the aggregated values for some predefined measure are calculated and the top-k interesting group-by attributes to partition the subspace are found.

For instance, as an example in [82], to answer a query "Columbus LCD", the KDAP system may aggregate the sales about "LCD" and break down the results into sub-aggregates for "Projector Technology = LCD", "Department = Monitor, Product = Flat Panel (LCD)", etc. Only the tuples that link with "Columbus" will be considered. A user can then drill down to aggregates of finer granularity.

Both the KDAP method and our study consider aggregate cells in keyword matching. The critical difference is that the two approaches address two different application scenarios. In the KDAP method, the aggregates of the most general subspaces are enumerated, and the top-k interesting group-by attributes are computed to help a user to drill down the results. In other words, KDAP serves the interactive exploration of data using keyword search.

In this study, the aggregate keyword search is modeled as a type of queries. Only the minimal aggregate cells matching a query are returned. Moreover, we focus on the efficiency of query answering. Please note that [82] does not report any experimental results on the efficiency of query answering in KDAP since it is not a major concern in that study.

3.2.2 Comparisons with Iceberg Cube Computation

As elaborated in Example 3, aggregate keyword search finds aggregate cells in a data cube lattice (that is, the aggregate cell lattice) in the aggregate space $\mathcal{D}$ in the query. Thus, aggregate keyword search is related to the problem of iceberg cube computation which has been studied extensively.

The concept of data cube is formulated in [35]. [28] proposes iceberg queries which find in a cube lattice the aggregate cells satisfying some given constraints (for example, aggregates whose `SUM` passing a given threshold).

Efficient algorithms for computing iceberg cubes with respect to various constraints have been developed. Particularly, the BUC algorithm [9] exploits monotonic constraints like `COUNT(*)` $\geqslant v$ and conducts a bottom-up search (that is, from the most general aggregate cells to the most specific ones). [37] tackles non-monotonic constraints like `AVG(*)` $\geqslant v$ by using some weaker but monotonic constraints in pruning. More efficient algorithms for iceberg cube computation are proposed in [83, 30]. The problem of iceberg cube computation on distributed network environment is investigated in [58].

A keyword query can be viewed as a special case of iceberg queries, where the constraint is that the tuples in an aggregate cell should jointly match all keywords in the query. However, this kind of constraints have not been explored in the literature of iceberg cube computation. The existing methods only consider the constraints composed by SQL aggregates like `SUM`, `AVG` and `COUNT`. In those constraints, every tuple in an aggregate cell contributes to the aggregate which will be computed and checked against the constraint. In aggregate keyword search, a keyword is expected to appear in only a small subset of tuples. Therefore, most tuples of an aggregate cell may not match any keyword in the query, and thus do not need to be considered in the search.

Due to the monotonicity in aggregate keyword search (Corollary 1), can we straightforwardly extend an existing iceberg cube computation method like BUC to tackle the aggregate keyword search problem? In aggregate keyword search, we are interested in the minimal aggregate cells matching all keywords in the query. However, all the existing iceberg cube computation methods more or less follow the BUC framework and search from the most general cell to the most specific cells in order to use monotonic constraints to prune the search space. The general-to-specific search strategy is inefficient for aggregate keyword search since it has to check many answers to the query until the minimal answers are computed.

3.3 The Maximum Join Approach

As reviewed in Chapter 2, inverted indexes of keywords [6, 56, 92, 93, 38] are heavily used in keyword search and have been supported extensively in practical systems. It is natural to exploit inverted indexes of keywords to support aggregate keyword search.

3.3.1 A Simple Nested Loop Solution

For a keyword w and a text-rich attribute A, let $IL_A(w)$ be the inverted list of tuples which contain w in attribute A. That is, $IL_A(w) = \{t \in T | w \text{ appears in } t[A]\}$.

Consider a simple query $Q = (\mathcal{D}, C, \{w_1, w_2\})$ where there are only two keywords in the query and there is only one text-rich attribute C. How can we derive the minimal answers to the query from $IL_C(w_1)$ and $IL_C(w_2)$?

For a tuple $t_1 \in IL_C(w_1)$ and a tuple $t_2 \in IL_C(w_2)$, every aggregate cell c that is a common ancestor of both t_1 and t_2 matches the query. We are interested in the minimal answers. Then, what is the most specific aggregate cell that is a common ancestor of both t_1 and t_2?

Definition 4 (Maximum join) *For two tuples t_x and t_y in table R, the* **maximum join** *of t_x and t_y on attribute set $\mathcal{A} \subseteq R$ is a tuple $t = t_x \vee_{\mathcal{A}} t_y$ such that (1) for any attribute $A \in \mathcal{A}$, $t[A] = t_x[A]$ if $t_x[A] = t_y[A]$, otherwise $t[A] = *$; and (2) for any attribute $B \notin \mathcal{A}$, $t[B] = *$.*

We call this operation maximum join since it keeps the common values between the two operant tuples on as many attributes as possible.

Example 4 (Maximum join) *In Table 3.2,* $(a_1, b_1, c_1, \{w_1, w_3\}) \vee_{ABC} (a_1, b_1, c_2, \{w_2, w_5\}) = (a_1, b_1, *, *)$. ■

Algorithm 1 The simple nested loop algorithm.

```
Input: query $Q = (\mathcal{D}, C, \{w_1, w_2\})$, $IL_C(w_1)$ and $IL_C(w_2)$;
Output: minimal aggregates matching $Q$;
    Step 1: generate possible minimal aggregates
 1: $Ans = \emptyset$; // $Ans$ is the answer set
 2: for each tuple $t_1 \in IL_C(w_1)$ do
 3:   for each tuple $t_2 \in IL_C(w_2)$ do
 4:     $Ans = Ans \cup \{t_1 \vee_{\mathcal{D}} t_2\}$;
 5:   end for
 6: end for
    Step 2: remove non-minimal aggregates from $Ans$
 7: for each tuple $t \in Ans$ do
 8:   for each tuple $t' \in Ans$ do
 9:     if $t' \prec t$ then
10:       $Ans = Ans - \{t'\}$;
11:     else if $t \prec t'$ then
12:       $Ans = Ans - \{t\}$;
13:       break;
14:     end if
15:   end for
16: end for
```

As can be seen from Figure 3.1, the maximal-join of two tuples gives the least upper bound (supremum) of the two tuples in the lattice. In general, we have the following property of maximum joins.

Corollary 2 (Properties) *The maximum-join operation is associative. That is,* $(t_1 \vee_{\mathcal{A}} t_2) \vee_{\mathcal{A}} t_3 = t_1 \vee_{\mathcal{A}} (t_2 \vee_{\mathcal{A}} t_3)$*. Moreover,* $\vee_{i=1}^{l} t_i$ *is the supremum of tuples* $t_1, \ldots, t_l$ *in the aggregate lattice.* ■

Using the maximum join operation, we can conduct a nested loop to answer a simple query $Q = (\mathcal{D}, C, \{w_1, w_2\})$ as shown in Algorithm 1. The algorithm is in two steps. In the first step, maximum joins are applied on pairs of tuples from $IL_C(w_1)$ and $IL_C(w_2)$. The maximum joins are candidates for minimal answers. In the second step, we remove those aggregates that are not minimal.

The simple nested loop method can be easily extended to handle queries with more than two keywords and more than one text-rich attribute. Generally, for query $Q = (\mathcal{D}, \mathcal{C}, \{w_1, \ldots, w_m\})$, we can derive the inverted list of keyword w_i ($1 \leqslant i \leqslant m$) on attribute set $\mathcal{C}$ as $IL_{\mathcal{C}}(w_i) = \cup_{C \in \mathcal{C}} IL_C(w_i)$. Moreover, the first step of Algorithm 1 can be extended so that m nested loops are conducted to obtain the maximal joins of tuples $\vee_{i=1}^{m} t_i$ where $t_i \in IL_{\mathcal{C}}(w_i)$.

To answer query $Q = (\mathcal{D}, \mathcal{C}, \{w_1, \ldots, w_m\})$, the nested loop algorithm has the time complexity $O(\prod_{i=1}^{m} |IL_{\mathcal{C}}(w_i)|^2)$. The first step takes time $O(\prod_{i=1}^{m} |IL_{\mathcal{C}}(w_i)|)$ and may generate up to $\prod_{i=1}^{m} |IL_{\mathcal{C}}(w_i)|$ aggregates in the answer set. To remove the non-minimal answers, the second step takes time $O(\prod_{i=1}^{m} |IL_{\mathcal{C}}(w_i)|^2)$. In a relational table $T = (A_1, \ldots, A_n)$ with N rows, the total number of aggregate cells is $O(N \cdot 2^n)$ or $O(\Pi_{i=1}^{n}(|A_i| + 1))$, where $|A_i|$ is the number of distinct values in dimension A_i. Thus, the simple nested algorithm needs $O(N^2 \cdot 2^{n+1})$ time in the worst case.

Clearly, the nested loop method is inefficacious for large databases and queries with multiple keywords. In the rest of this section, we will develop several interesting techniques to speed up the search.

3.3.2 Pruning Exactly Matching Tuples

Hereafter, when the set of text-rich attributes $\mathcal{C}$ is clear from context, we write an inverted list $IL_{\mathcal{C}}(w)$ as $IL(w)$ for the sake of simplicity. Similarly, we write $t_1 \vee_{\mathcal{D}} t_2$ as $t_1 \vee t_2$ when $\mathcal{D}$ is clear from the context.

Theorem 1 (Pruning exactly matching tuples) *Consider query $Q = (\mathcal{D}, \mathcal{C}, \{w_1, w_2\})$ and inverted lists $IL(w_1)$ and $IL(w_2)$. For tuples $t_1 \in IL(w_1)$ and $t_2 \in IL(w_2)$ such that $t_1[\mathcal{D}] = t_2[\mathcal{D}]$, $t_1 \vee t_2$ is a minimal answer. Moreover, except for $t_1 \vee t_2$, no other minimal answers can be an ancestor of either t_1 or t_2.*

Proof. The minimality of $t_1 \vee t_2$ holds since $t_1 \vee t_2$ does not take value $*$ on any attributes in $\mathcal{D}$. Except for $t_1 \vee t_2$, every ancestor of t_1 or t_2 must be an ancestor of $t_1 \vee t_2$ in $\mathcal{D}$, and thus cannot be a minimal answer. ■

Using Theorem 1, once two tuples $t_1 \in IL(w_1)$ and $t_2 \in IL(w_2)$ are found such that $t_1[\mathcal{D}] = t_2[\mathcal{D}]$, $t_1 \vee t_2$ should be output as a minimal answer, and t_1 and t_2 should be ignored in the rest of the join.

3.3.3 Reducing Matching Candidates Using Answers

For an aggregate keyword query, we can use some answers found before, which may not even be minimal, to prune matching candidates.

Theorem 2 (Reducing matching candidates) *Let t be an answer to $Q = (\mathcal{D}, \mathcal{C}, \{w_1, w_2\})$ and $t_1 \in IL(w_1)$. For any tuple $t_2 \in IL(w_2)$, if for every attribute $D \in \mathcal{D}$ where $t_1[D] \neq t[D]$, $t_2[D] \neq t_1[D]$, then $t_1 \vee t_2$ is not a minimal answer to the query.*

Proof. For every attribute $D \in \mathcal{D}$ where $t_1[D] \neq t[D]$, since $t_1[D] \neq t_2[D]$, $(t_1 \vee t_2)[D] = *$. On every other attribute $D' \in \mathcal{D}$ where $t_1[D'] = t[D']$, either $(t_1 \vee t_2)[D'] = t_1[D'] = t[D']$ or $(t_1 \vee t_2)[D'] = *$. Thus, $t_1 \vee t_2 \preceq t$. Consequently, $t_1 \vee t_2$ cannot be a minimal answer to Q. ■

Using Theorem 2, for each tuple $t_1 \in IL(w_1)$, if there is an answer t (not necessary a minimal one) to query $Q = (\mathcal{D}, \mathcal{C}, \{w_1, w_2\})$ such that $t[D] = t_1[D]$ on some attribute $D \in \mathcal{D}$, we can use t to reduce the tuples in $IL(w_2)$ that need to be joined with t_1 as elaborated in the following example.

Example 5 (Reducing matching candidates) *Consider query $Q = (ABC, D, \{w_1, w_2\})$ on the table T shown in Table 3.2. A maximum join between (a_1, b_1, c_2) and (a_1, b_2, c_2) generates an answer $(a_1, *, c_2)$ to the query. Although tuple (a_1, b_1, c_2) contains w_1 on D and tuple (a_2, b_2, c_1) contains w_2 on D, as indicated by Theorem 2, joining (a_1, b_1, c_2) and (a_2, b_2, c_1) results in aggregate cell $(*, *, *)$, which is an ancestor of $(a_1, *, c_2)$ and cannot be a minimal answer.* ■

For a tuple $t_1 \in IL(w_1)$ and an answer t to a query $Q = (\mathcal{D}, \mathcal{C}, \{w_1, w_2\})$, the tuples in $IL(w_2)$ surviving from the pruning using Theorem 2 can be found efficiently using inverted lists. In implementation, instead of maintaining an inverted list $IL(w)$ of keyword w, we maintain a set of inverted lists $IL_{A=a}(w)$ for every value a in the domain of every attribute A, which links all tuples having value a on attribute A and containing keyword w on the text-rich attributes. Clearly, $IL(w) = \cup_{a \in A} IL_{A=a}(w)$ where A is an arbitrary attribute. Here, we assume that tuples do not take null value on any attribute. If some tuples take null values on some attributes, we can simply ignore those tuples on those attributes, since those tuples do not match any query keywords on those attributes.

Suppose t is an answer to query $Q = (\mathcal{D}, \mathcal{C}, \{w_1, w_2\})$ and t_1 contains keyword w_1 but not w_2. Then, t_1 needs to join with only the tuples in

$$Candiate(t_1) = \cup_{D \in \mathcal{D}: t_1[D] \neq t[D]} IL_{D=t_1[D]}(w_2).$$

Other tuples in $IL(w_2)$ should not be joined with t_1 according to Theorem 2.

An answer t is called *overlapping* with a tuple t_1 if there exists at least one attribute $D \in \mathcal{D}$ such that $t[D] = t_1[D]$. Clearly, the more answers overlapping with t_1, the more candidates can be reduced.

Heuristically, the more specific $t_1 \vee t$ in Theorem 2, the more candidate tuples may be reduced for the maximum joins with t_1. Therefore, for each tuple $t_1 \in IL(w_1)$, we should try to find $t_2 \in IL(w_2)$ such that $t = t_1 \vee t_2$ contains as few $*$'s as possible. To implement the heuristic, for query $Q = (\mathcal{D}, \mathcal{C}, (w_1, w_2))$ and each tuple $t_1 \in IL(w_1)$ currently in the outer loop, we need to measure how well a tuple in $IL_{\mathcal{C}}(w_2)$ matches t_1 in $\mathcal{D}$. This can be achieved efficiently using bitmap operations as follows.

We initialize a counter of value 0 for every tuple in inverted list $IL_{\mathcal{C}}(w_2)$. For each attribute $D \in \mathcal{D}$, we compute $IL_{D=t_1[D]}(w_1) \cap IL_{\mathcal{D}}(w_2)$ using bitmaps. For each tuple in the intersection, the counter of the tuple is incremented by 1. After checking all attributes in $\mathcal{D}$, the tuples having the largest counter value match t_1 the best. Thus, we can sort tuples in $IL_{\mathcal{C}}(w_2)$ in the counter value descending order and conduct maximum joins with t_1 on them. In this order, the most specific matches can be found the earliest.

Comparing to the traditional solution which builds an inverted list $IL(w)$ for each keyword w, in our method, the number of inverted lists is larger. However, in terms of the space complexity, maintaining one inverted list per keyword and multiple lists per keyword only differs in $O(|T| \cdot |D|)$ space, where $|T|$ is the number of keywords, and $|D|$ is the number of dimensional attributes. Since $|D|$ is a constant, the space complexity is the same. Moreover, the inverted lists can be accessed in $O(1)$ time using hashing. Furthermore, as explained above, the number of inverted lists to be accessed only depends on the number of dimensions on which a tuple t_1 and an answer t have different values. The heuristic used in our method can find the most specific answers as early as possible, which in turn is beneficial for the pruning strategy, since the number of dimensions that t_1 and t having different values is small. As a result, maintaining the inverted lists for each dimension values is useful to improve the query performance.

3.3.4 Fast Minimal Answer Checking

In order to obtain minimal answers to an aggregate keyword query, we need to remove non-minimal answers from the answer set. The naïve method in Algorithm 1 adopts a nested loop. Each time when a new answer t (not necessary a minimal answer) is found, we need to scan the answer set once to consider if t should be inserted into the answer set. If t is an ancestor of some answers in the answer set, t cannot be a minimal answer, thus t should not be inserted into the answer set. In the other case, if t is a descendant of some answers in the answer set, those answers cannot be the minimal answers, thus need to be removed from the answer set. Without any index structures, it leads to $O(n^2)$ time complexity, where n is the size of the answer set. Can we do better?

The answers in the answer set can be organized into inverted lists. For an arbitrary attribute A, $IL_{A=a}$ represents the inverted list for all answers t such that $t[A] = a$. Using Definition 1, we have the following result.

Corollary 3 (Ancestor and descendant) *Suppose the answers to query $Q = (\mathcal{D}, \mathcal{C}, W)$ are organized into inverted lists, and t is an answer. Let*

$$\begin{aligned} Ancestor(t) = \quad & (\cap_{D \in \mathcal{D}: t[D] \neq *}(IL_{D=*} \cup IL_{D=t[D]})) \\ & \cap(\cap_{D \in \mathcal{D}: t[D] = *} IL_{D=*}) \end{aligned}$$

and $Descendant(t) = \cap_{D \in \mathcal{D}: t[D] \neq *} IL_{D=t[D]}$. That is, if an answer $t_1 \in Ancestor(t)$, for those dimensions that t has a $*$ value, t_1 also has a $*$ value; for those dimensions that t has a non-$*$ value, t_1 must either have the same non-$*$ value or a $*$ value; if an answer $t_2 \in Descendant(t)$, for those dimensions that t has a non-$*$ value, t_2 must have the same non-$*$ value.

Then, the answers in $Ancestor(t)$ are not minimal. Moreover, t is not minimal if $Descendant(t) \neq \emptyset$. ■

Both $Ancestor(t)$ and $Descendant(t)$ can be implemented efficiently using bitmaps. For each newly found answer t, we calculate $Ancestor(t)$ and $Descendant(t)$. If $Descendant(t) \neq \emptyset$, t is not inserted into the answer set. Otherwise, t is inserted into the answer set, and all answers in $Ancestor(t)$ are removed from the answer set. In this way, we maintain a small answer set during the maximal join computation. When the computation is done, the answers in the answer set are guaranteed to be minimal.

3.3.5 Integrating the Speeding-Up Strategies

The strategies described in Sections 3.3.2, 3.3.3, and 3.3.4 can be integrated into a fast maximum-join approach as shown in Algorithm 2.

For a query containing m keywords, we need $m - 1$ rounds of maximum joins. Heuristically, the larger the sizes of the two inverted lists in the maximum join, the more costly

Algorithm 2 The fast maximum-join algorithm.

```
Input: query $Q = (\mathcal{D}, C, \{w_1, \ldots, w_m\})$, $IL_C(w_1)$, ..., $IL_C(w_m)$;
Output: minimal aggregates matching $Q$;
1: $Ans = \emptyset$; // $Ans$ is the answer set
2: $CandList = \{IL_C(w_1), \ldots, IL_C(w_m)\}$;
3: initialize $k = 1$; /* $m$ keywords need $m - 1$ rounds of joins */
4: while $k < m$ do
5:    $k = k + 1$;
6:    pick two candidate inverted lists $IL_C^1$ and $IL_C^2$ with smallest sizes from $CandList$,
      and remove them from $CandList$;
7:    for each tuple $t_1 \in IL_C^1$ do
8:       use strategy in Section 3.3.3 to calculate the counter for tuples in $IL_C^2$;
9:       while $IL_C^2$ is not empty do
10:         let $t_2$ be the tuple in $IL_C^2$ with largest counter;
11:         if the counter of $t_2$ is equal to the dimension then
12:            use strategy in Section 3.3.4 to insert an answer $t_1$ into $Ans$; /* $t_1$ exactly
               matches $t_2$, as described in Section 3.3.2 */
13:            remove $t_1$ and $t_2$ from each inverted list;
14:            break;
15:         else
16:            maximum join $t_1$ and $t_2$ to obtain an answer;
17:            use strategy in Section 3.3.4 to insert the answer into $Ans$;
18:            use strategy in Section 3.3.3 to find candidate tuples in $IL_C^2$, and update $IL_C^2$;
19:         end if
20:      end while
21:   end for
22:   build an inverted list for answers in $Ans$ and insert it into $CandList$;
23: end while
```

the join. Here, the size of an inverted list is the number of tuples in the list. Thus, in each round of maximum joins, we pick two inverted lists with the smallest sizes.

It is a challenging problem to develop a better heuristic to decide the orders of maximum joins for those keywords in the query. The general principle is to reduce the total number of maximum joins which are needed in the algorithm. However, there are a set of challenges which need to be addressed.

The first challenge is how to accurately estimate the number of maximum joins which are needed. In relational databases, we can accurately estimate the number of joins (e.g., equi-join, left outer join) based on the correlations of attribute values. However, in our case, the situation is much more complicated. We developed a set of pruning strategies to prune the number of maximum joins in the algorithm. Those pruning strategies may affect the estimation greatly. As a result, the estimation needs to take into account the pruning strategies, and is far from trivial.

The second challenge is how to estimate the number of intermediate minimal answers generated in each round of maximum joins. In our problem, the candidate answers generated by any two keywords will become the input to the next rounds of maximum joins. Thus, the size of the intermediate minimal answers plays an important role for the performance of the maximum-join algorithm. Consider two pairs of keywords (w_1, w_2) and (w_3, w_4). Suppose the number of maximum joins for w_1 and w_2 is smaller than that for w_3 and w_4, we cannot simply make a conclusion that w_1 and w_2 should be maximum-joined first, since it is possible that the number of the intermediate minimal answers generated by joining w_1 and w_2 is much larger than that by joining w_3 and w_4.

The current heuristic used in our algorithm is very simple, and it achieves relatively good performance. For a performance comparison, we compared our simple heuristic to a random selection method (that is, for a query containing m keywords, we randomly pick two words and get the corresponding lists of tuples for the next round of maximum joins). In Figure 3.9, we show the performance of the maximum-join algorithm using the simple heuristic we developed in this chapter, as well as that using the random method. The simple heuristic clearly outperforms the random method. We leave the problem of developing a better heuristic for deciding the orders of maximum-joins as an interesting future direction.

When we conduct the maximum joins between the tuples in two inverted lists, for each tuple $t_1 \in IL_C^1$, we first compute the counters of tuples in IL_C^2, according to the strategy in Section 3.3.3. Apparently, if the largest counter value is equal to the number of dimensions in the table, the two tuples are exactly matching tuples. According to the strategy in Section 3.3.2, the two tuples can be removed and a minimal answer is generated. We use the strategy in Section 3.3.4 to insert the answer into the answer set. If the largest counter value is less than the number of dimensions, we pick the tuple t_2 with the largest counter value and compute the maximum join of t_1 and t_2 as an answer. Again, we use the strategy in Section 3.3.4 to insert the answer into the answer set. The answer set should be updated accordingly, and non-minimal answers should be removed.

Based on the newly found answer, we can use the strategy in Section 3.3.3 to reduce the number of candidate tuples to be joined in IL_C^2. Once IL_C^2 becomes empty, we can continue to pick the next tuple in IL_C^1. At the end of the algorithm, the answer set contains exactly the minimal answers.

3.4 The Keyword Graph Approach

If the number of keywords in an aggregate keyword query is not small, or the database is large, the fast maximum join method may still be costly. In this section, we propose to materialize a keyword graph index for fast answering of aggregate keyword queries.

3.4.1 Keyword Graph Index and Query Answering

Since graphs are capable of modeling complicated relationships, several graph-based indexes have been proposed to efficiently answer some queries. For example, [85] proposes a keyword relationship matrix to evaluate keyword relationships in distributed databases, which can be considered as a special case of a graph index. Moreover, [78] extends the work [85] by summarizing each database using a keyword relationship graph, where nodes represent terms and edges describe relationships between them. The keyword relationship graph can help to select top-k most promising databases effectively during the query processing. Recently, [21] presents a personalized search approach that involves a graph-based representation of the user profile.

However, those graph indexes are not designed for aggregate keyword queries. Can we develop keyword graph indexes for effective and efficient aggregate keyword search?

Apparently, for an aggregate keyword query $Q = (\mathcal{D}, \mathcal{C}, W)$, $(*, *, \ldots, *)$ is an answer if for every keyword $w \in W$, $IL_{\mathcal{C}}(w) \neq \emptyset$. This can be checked easily. We call $(*, *, \ldots, *)$ a *trivial answer*. We build the following keyword graph index to find non-trivial answers to an aggregate keyword query.

Definition 5 (Keyword graph index) *Given a table T, a* **keyword graph index** *is an undirected graph $G(T) = (V, E)$ such that (1) V is the set of keywords in the text-rich attributes in T; and (2) $(u, v) \in E$ is an edge if there exists a non-trivial answer to query $Q_{u,v} = (\mathcal{D}, \mathcal{C}, \{w_1, w_2\})$, where $\mathcal{D}$ and $\mathcal{C}$ represent the aggregate space and the set of textual attributes in T, respectively. Both $\mathcal{D}$ and $\mathcal{C}$ can be set to be the complete set of attributes in T. Edge (u, v) is associated with the set of minimal answers to query $Q_{u,v}$.* ■

Obviously, the number of edges in the keyword graph index is $O(|V|^2)$, while each edge is associated with the complete set of minimal answers. In practice, the number of keywords in

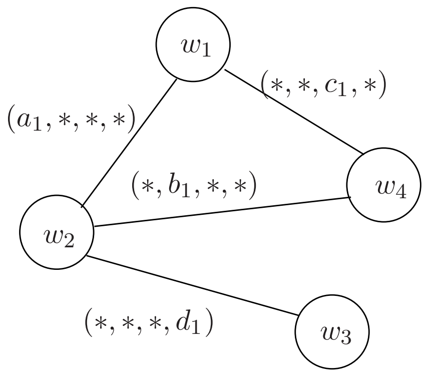


Figure 3.2: A counter example showing that the condition in Theorem 3 is not sufficient.

the text-rich attributes is limited, and many keyword pairs lead to trivial answers only. For example, we conduct the experiments using the IMDB data set. It contains 134,080 tuples, 7 dimensions and 3 textual attributes. On average each tuple contains 9.206 keywords in the text attributes. Among 16,297 distinct keywords in the IMDB data set, only 305,412 pairs of keywords (that is, 0.23%) have non-trivial answers. The average size of the minimal answer set on edges (that is, average number of minimal answers per edge) is 26.0. The size of the whole keyword graph index is 103.3 MB. Thus, a keyword graph index can be maintained easily in the main memory.

Keyword graph indexes have the following property.

Theorem 3 (Keyword graph) *For an aggregate keyword query $Q = (\mathcal{D}, \mathcal{C}, W)$, there exists a non-trivial answer to Q in table T only if in the keyword graph index $G(T)$ on table T, there exists a clique on the set W of vertices.*

Proof. Let c be a non-trivial answer to Q. Then, for any $u, v \in W$, c must be a non-trivial answer to query $Q_{u,v} = (\mathcal{D}, \mathcal{C}, \{u, v\})$. That is, (u, v) is an edge in $G(T)$. ■

Theorem 3 is a necessary condition. It is easy to see that the condition is not sufficient.

Example 6 (A counter example) *Consider a keyword graph index in Figure 3.2. There are 4 distinct keywords. For a query $Q = \{w_1, w_2, w_4\}$, there exists a clique in the keyword graph index. However, by joining minimal answers on edges connecting those keywords, we can find that Q does not have a non-trivial answer.* ■

Once the minimal answers to aggregate keyword queries on keyword pairs are materialized in a keyword graph index, we can use Theorem 3 to answer queries efficiently. For a query $Q = (\mathcal{D}, \mathcal{C}, \{w_1, \ldots, w_m\})$, we can check whether there exists a clique on vertices $w_1, \ldots, w_m$. If not, then there is no non-trivial answer to the query. If there exists a clique, we try to construct minimal answers using the minimal answer sets associated with the edges in the clique.

If the query contains only two keywords (that is, $m = 2$), the minimal answers can be found directly from edge (w_1, w_2) since the answers are materialized on the edge. If the query involves more than 2 keywords (that is, $m \geqslant 3$), the minimal answers can be computed by maximum joins on the sets of minimal answers associated with the edges in the clique. It is easy to show the following.

Lemma 1 (Maximal join on answers) *If t is a minimal answer to a query $Q = (\mathcal{D}, \mathcal{C}, \{w_1, \ldots, w_m\})$, then there exist minimal answers t_1 and t_2 to queries $(\mathcal{D}, \mathcal{C}, \{w_1, w_2\})$ and $(\mathcal{D}, \mathcal{C}, \{w_2, \ldots, w_m\})$, respectively, such that $t = t_1 \vee_{\mathcal{D}} t_2$.* ■

Let $Answer(Q_1)$ and $Answer(Q_2)$ be the sets of minimal answers to queries $Q_1 = (\mathcal{D}, \mathcal{C}, \{w_1, w_2\})$ and $Q_2 = (\mathcal{D}, \mathcal{C}, \{w_2, w_3\})$, respectively. We call the process of applying maximal joins on $Answer(Q_1)$ and $Answer(Q_2)$ to compute the minimal answers to query $Q = (\mathcal{D}, \mathcal{C}, \{w_1, w_2, w_3\})$ the *join* of $Answer(Q_1)$ and $Answer(Q_2)$. The cost of the join is $O(|Answer(Q_1)| \cdot |Answer(Q_2)|)$.

To answer query $Q = (\mathcal{D}, \mathcal{C}, \{w_1, \ldots, w_m\})$, using Lemma 1 repeatedly, we only need to check $m - 1$ edges covering all keywords $w_1, \ldots, w_m$ in the clique. Each edge is associated with the set of minimal answers to a query on a pair of keywords. The *weight* of the edge is the size of the answer set. In order to reduce the total cost of the joins, heuristically, we can find a spanning tree connecting the m keywords such that the product of the weights on the edges is minimized.

The traditional minimum spanning tree problem minimizes the sum of the edge weights. Several greedy algorithms, such as Prim's algorithm and Kruskal's algorithm [18], can find the optimal answers in polynomial time. The greedy selection idea can also be applied to our problem here. The greedy method works as follows: all keywords in the query are unmarked in the beginning. We sort the edges in the clique in the weight ascending order. The edge of the smallest weight is picked first and the keywords connected by the edge are marked. Iteratively, we pick a new edge of the smallest weight such that it connects a marked keyword and an unmarked one until all keywords in the query are marked.

3.4.2 Index Construction

A naïve method to construct a keyword graph index is to compute maximum joins on the inverted lists of every keyword pairs. However, the naïve method is inefficient. If tuple t_1 contains keywords w_1 and w_2, and tuple t_2 contains w_3 and w_4, $t_1 \vee t_2$ may be computed up to 4 times since $t_1 \vee t_2$ is an answer to four pairs of keywords including (w_1, w_3), (w_1, w_4), (w_2, w_3) and (w_2, w_4).

As an efficient solution, we conduct a self-maximum join on the table to construct the keyword graph. For two tuples t_1 and t_2, we compute $t_1 \vee t_2$ only once, and add it to all edges of (u, v) where u and v are contained in t_1 and t_2, but not both in either t_1 or t_2. By removing those non-minimal answers, we find all the minimal answers for every pair of keywords, and obtain the keyword graph.

Trivial answers are not stored in a keyword graph index. This constraint improves the efficiency of keyword graph construction. For a tuple t, the set of tuples that generate a non-trivial answer by a maximum join with t is $\cup_D IL_{D=t[D]}$, where $IL_{D=t[D]}$ represents the inverted list for all tuples having value $t[D]$ on dimension D. Maximum joins should be applied to only those tuples and t. The keyword graph construction method is summarized in Algorithm 3.

3.4.3 Index Maintenance

A keyword graph index can be maintained easily against insertions, deletions and updates on the table.

When a new tuple t is inserted into the table, we only need to conduct the maximum join between t and the tuples already in the table as well as t itself. If t contains some new keywords, we create the corresponding keyword vertices in the keyword graph. The maintenance procedure is the same as lines 3-19 in Algorithm 3.

When a tuple t is deleted from the table, for a keyword only appearing in t, the vertex and the related edges in the keyword graph should be removed. If t also contains some other keywords, we conduct maximum joins between t and other tuples in the table. If the join result appears as a minimal answer on an edge (u, v) where u and v are two keywords, we re-compute the minimal answers of $Q_{u,v} = (\mathcal{D}, \mathcal{C}, \{u, v\})$ by removing t from T.

Algorithm 3 The keyword graph construction algorithm.

```
Input: A table $T$;
Output: A keyword graph $G(T) = (V, E)$;
1: initialize $V$ as the set of keywords in $T$;
2: for each tuple $t \in T$ do
3:    initialize the candidate tuple set to $Cand = \emptyset$;
4:    let $Cand = \cup_D IL_{D=t[D]}$;
5:    let $W_t$ be the set of keywords contained in $t$;
6:    for each tuple $t' \in Cand$ do
7:       if $t = t'$ then
8:          for each pair $w_1, w_2 \in W_t$ do
9:             add $t$ to edge $(w_1, w_2)$, and remove non-minimal answers on edge $(w_1, w_2)$
               (Section 3.3.4);
10:         end for
11:      else
12:         let $W_{t'}$ be the set of keywords contained in $t'$;
13:         $r = t \vee t'$;
14:         for each pair $w_1 \in W_t - W_{t'}$ and $w_2 \in W_{t'} - W_t$ do
15:            add $r$ to edge $(w_1, w_2)$;
16:            remove non-minimal answers on edge $(w_1, w_2)$ (Section 3.3.4);
17:         end for
18:      end if
19:   end for
20: end for
```

When a tuple t is updated, it can be treated as one deletion (the original tuple is deleted) and one insertion (the new tuple is inserted). The keyword graph index can be updated accordingly.

3.5 Extensions and Generalization

The methods in Sections 3.3 and 3.4 look for *complete matches*, that is, all keywords are contained in an answer. However, complete matches may not exist for some queries. For example, in Table 3.1, query $Q = (\{$`Month, State, City, Event`$\}, \{$`Event, Descriptions`$\}$, $\{$"space shuttle", "motorcycle", "rock music"$\})$ cannot find a non-trivial answer.

In this section, we propose two solutions to handle queries which do not have a non-trivial answer or even an answer. Our first solution allows partial matches (for example, matching m' of m keywords $(m' \leqslant m)$). In our second solution, a keyword is allowed to be matched by some other similar keywords according to a keyword ontology (for example, keyword "fruit" in a query can be matched by keyword "apple" in the data).

3.5.1 Partial Keyword Matching

Given a query $Q = (\mathcal{D}, \mathcal{C}, \{w_1, \ldots, w_m\})$, **partial keyword matching** is to find all minimal, non-trivial answers that cover as many query keywords as possible.

For example, in Table 3.1, there is no non-trivial answer to query $Q = (\{$`Month, State, City, Event`$\}$, $\{$`Event, Descriptions`$\}$, $\{$"space shuttle", "motorcycle", "rock music"$\})$. However, a minimal answer (December, Texas, $*$, $*$, $*$) partially matching $\frac{2}{3}$ of the keywords may still be interesting to the user.

For a query containing m keywords, a brute-force solution is to consider all possible combinations of m keywords, $m-1$ keywords, $\ldots$, until some non-trivial answers are found. For each combination of keywords, we need to conduct the maximum join to find all the minimal answers. Clearly, it is inefficient at all.

Here, we propose an approach using the keyword graph index. Theorem 3 provides a necessary condition that a complete match exists if the corresponding keyword vertices in the keyword graph index form a clique. Given a query containing m keywords $w_1, \ldots, w_m$, we can check the subgraph $G(Q)$ of the keyword graph which contains only the keyword vertices in the query. By checking $G(Q)$, we can identify if some non-trivial answers may exist for a subset of query keywords. Moreover, we can identify the maximum number of query keywords that can be matched by extracting the maximum clique from the corresponding keyword graph.

Although the problem of finding the maximum clique is one of the first problems shown to be NP-complete [33], in practice, an aggregate keyword query often contains a small number of keywords (for example, less than 10). Thus, the query keyword subgraph $G(Q)$ is often small. It is possible to enumerate all the possible cliques in $G(Q)$.

To find partial matches to an aggregate keyword query, we start from those largest cliques. By joining the sets of minimal answers on the edges, the minimal answers can be found. If there is no non-trivial answer in the largest cliques, we need to consider the smaller cliques and the minimal answers. The algorithm stops until some non-trivial minimal answers are found.

Alternatively, a user may provide the minimum number of keywords that need to be covered in an answer. Our method can be easily extended to answer such a constraint-based partial keyword matching query – we only need to search answers on those cliques whose size passes the user's constraint.

In some other situations, a user can specify a subset of keywords that must be covered in an answer. Our method can also be easily extended to deal with such cases. Those selected keywords are chosen as seed nodes in the keyword graph index. We only need to find larger cliques containing those seed nodes.

3.5.2 Hierarchical Keyword Matching

Keywords generally follow a hierarchical structure. A *keyword hierarchy* (or a *keyword ontology*) is a strict partial order $\prec$ on the set L of all keywords. We assume that there exists a meta symbol $* \in L$ which is the most general category generalizing all keywords. For two keywords $w_1, w_2 \in L$, if $w_1 \prec w_2$, w_1 is *more general* than w_2. For example, `fruit` $\prec$ `apple`. We write $w_1 \preceq w_2$ if $w_1 \prec w_2$ or $w_1 = w_2$.

If an aggregate keyword query cannot find a non-trivial, complete matching answer, alternatively, we may loosen the query requirement to allow a keyword in the query to be matched by another keyword in data that is similar in the keyword hierarchy.

There are two ways of hierarchical keyword matching.

Specialization matches a keyword w in a query with another keyword w' which is a descendant of w in the keyword hierarchy. For example, keyword "fruit" in a query may be matched by keyword "apple" in the hierarchy.

Generalization matches w with keyword w'' which is an ancestor of w. For example, keyword "apple" in a query maybe matched by keyword "fruit" in the hierarchy.

In this chapter, we only discuss specialization here in detail, partly because in many applications users may be interested in more specific answers. However, our method can be easily extended to support generalization matching.

Using a keyword graph index, we can greedily loosen the query requirement to allow hierarchical keyword matching. For a query containing m keywords $w_1, \ldots, w_m$ which does not have a non-trivial complete matching answer, we can specialize a query keyword to a more specific one according to the hierarchical structure, and find the complete matching

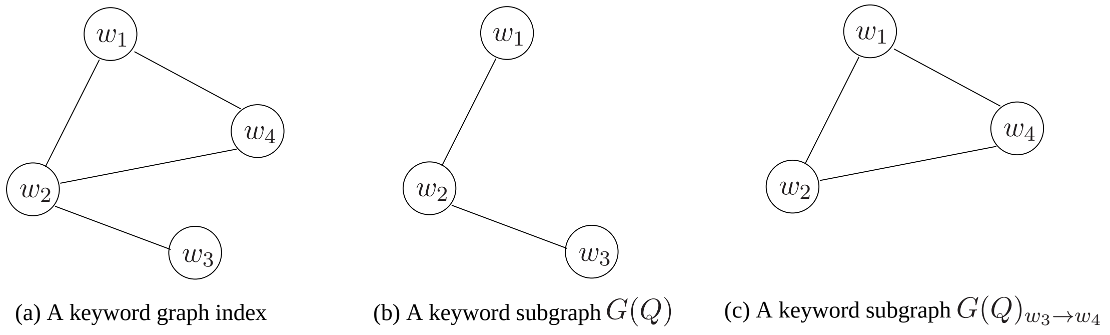

w_1
w_2
w_3
w_4
(a) A keyword graph index
(b) A keyword subgraph $G(Q)$
(c) A keyword subgraph $G(Q)_{w_3 \to w_4}$

Figure 3.3: An example of keyword specialization.

answers for the new set of keywords. The search continues until some non-trivial minimal answers can be found.

When we choose a query keyword to specialize, we should consider two factors.

First, by specializing w_1 to w_2 such that $w_1 \prec w_2$, how many new edges can be added to the query subgraph if w_1 is replaced by w_2? Since keywords are specialized in the matching, the edges between w_1 and some other query keywords can be retained. We are interested in the new edges that can be added to the query subgraph by specializing the keyword which can help to find a clique and a non-trivial answer.

Example 7 (Specialization) *Consider a keyword graph index in Figure 3.3(a). There are 4 distinct keywords. For a query $Q = \{w_1, w_2, w_3\}$, its corresponding keyword subgraph $G(Q)$ is shown in Figure 3.3(b). Clearly, keywords in Q only have 2 edges connecting them, and they do not form a clique structure, thus there are no exact answers to Q. However, we can replace keyword w_3 with another keyword w_4 if w_4 is a descendent of w_3. The keyword subgraph $G(Q)_{w_3 \to w_4}$ is shown in Figure 3.3(c). All the keywords in Figure 3.3(c) form a clique structure. By replacing keyword w_3 with keyword w_4, there are 3 edges connecting the keywords in the query.* ■

Second, how many descendant keywords of w_1 are also descendants of w_2 in the keyword hierarchy? Formally, we consider the ratio $\frac{\alpha(w_2)}{\alpha(w_1)}$ where $\alpha(w)$ is the number of leaf keywords in the keyword hierarchy that are descendants of w. The larger the ratio, the less generality is lost in the specialization. By taking the above two factors into account, we can assign an "extendability" score for each keyword in the query.

Definition 6 (Extendability) *Given a keyword w_1 in a query keyword subgraph $G(Q)$ and a keyword hierarchy, the* **extendability score** *of w_1 with respected to its descendant w_2 is*

$$ext(w_1 \rightarrow w_2) = \frac{\alpha(w_2)}{\alpha(w_1)}(|E(G(Q)_{w_1 \rightarrow w_2})| - |E(G(Q))|),$$

where $|E(G(Q))|$ is the number of edges in graph $G(Q)$ and $|E(G(Q)_{w_1 \rightarrow w_2}|$ is the number of edges in graph $G(Q)$ by replacing w_1 with w_2. The **extendability** *of w_1 is the maximum extendability value of $ext(w_1, desc(w_1))$ where $desc(w_1)$ is a descendant of w_1 in the keyword hierarchy.* ■

We can calculate the extendability of each keyword in a query. The keyword w with the highest extendability is selected, and the keyword w is replaced by w' such that $ext(w, w')$ is the maximum among all descendants of w. We call this a *specialization* of the query.

After a specialization of the query, we examine if a clique exists in the new query keyword subgraph. The more rounds of specialization, the more uncertainty introduced to the query. If a clique exists and the clique leads to a non-trivial answer, we terminate the specialization immediately and return the query answers; if not, we conduct another round of specialization greedily.

When the specialization of all keywords reaches the leaf keywords in the hierarchy, no more specialization can be conducted. In such an extreme case, the hierarchical keyword matching fails.

Too many rounds of specializations may not lead to a good answer loyal to the original query objective. A user may specify the maximum number of specializations allowed as a control parameter. Our method can be easily extended to accommodate such a constraint.

There are some other interesting directions to explore in the future study. For example, a keyword query typically is very short, which on average only contains two or three keywords [77]. As a result, keyword queries in practice are often ambiguous. To overcome this disadvantage, a useful tool is to develop some error tolerant keyword matching algorithms such that keywords in the query do not need to be exactly matched in the results. As another example, tuples in the relational databases may be semantically related. It is interesting to examine whether the semantic relationship among those tuples can be utilized to improve the performance of keyword search on relational databases. Furthermore, in Web search, approximate keyword matching is widely adopted. A user may mis-spell some keywords in the query, and the Web search engines could automatically find information using related keywords. The approximate keyword matching technique can be straightforwardly incorporated into the aggregate keyword search. For each keyword w in the query, we can first find its related keyword w' using the approximate keyword matching technique. Then, the inverted list of w' is retrieved for the query processing of aggregate keyword search.

3.6 Experimental Evaluation and Performance Study

In this section, we report a systematic empirical study to evaluate our aggregate keyword search methods using both real data sets and synthetic data sets. All the experiments were conducted on a PC computer running the Microsoft Windows XP SP2 Professional Edition operating system, with a 3.0 GHz Pentium 4 CPU, 1.0 GB main memory, and a 160 GB hard disk. The programs were implemented in C/C++ and were compiled using Microsoft Visual Studio .Net 2005.

3.6.1 Results on Real Data Sets

We first describe the data set we used in the experiments. Then, we report the experimental results.

The IMDB Data Set

The Internet Movie Database (IMDB) data set [76] has been used extensively in the previous work on keyword search on relational databases [39, 54, 23]. We use this data set to empirically evaluate our aggregate keyword search methods.

We downloaded the whole raw IMDB data. We pre-processed the data set by removing duplicate records and missing values. We converted a subset of its raw text files into a large relational table. The schema of the table and the statistical information are shown in Table 3.3.

We used the first 7 attributes as the dimensions in the search space. Some attributes such as "actor" and "actress" may have more than one value for one specific movie. To use those attributes as dimensions, we picked the most frequent value if multiple values exist on such an attribute in a tuple. After the pre-processing, we obtained a relational table of $134,080$ tuples.

Among the 10 attributes in the table, we use "genre", "keyword" and "location" as the

Attribute	Description	Cardinality
Movie	movie title	134,080
Director	director of the movie	62,443
Actor	leading actor of the movie	68,214
Actress	leading actress of the movie	72,908
Country	producing country of the movie	73
Language	language of the movie	45
Year	producing year of the movie	67
Genre	genres of the movie	24
Keyword	keywords of the movie	15,224
Location	shooting locations of the movie	1,049

Table 3.3: The IMDB database schema and its statistical information.

text attributes, and the remaining attributes as the dimensions. Table 3.3 also shows the total number of keywords for each text attribute and the cardinality of each dimension. On average each tuple contains 9.206 keywords in the text attributes.

In data representation, we adopted the popular packing technique [9]. A value on a dimension is mapped to an integer between 1 and the cardinality of the dimension. We also map keywords to integers.

Index Construction

Both the simple nested loop approach in Algorithm 1 and the fast maximum join approach in Algorithm 2 need to maintain the inverted list index for each keyword in the table. The total number of keywords in the IMDB data set is $16,297$. The average length of those inverted lists is 87.1, while the largest length is $13,442$. The size of the whole inverted list is 5.5 MB.

We used Algorithm 3 to construct the keyword graph index. The construction took 107 seconds. Among $16,297$ keywords, $305,412$ pairs of keywords (that is, 0.23%) have non-trivial answers. The average size of the minimal answer set on edges (that is, average number of minimal answers per edge) is 26.0. The size of the whole keyword graph index is 103.3 MB.

Both the inverted list index and the keyword graph index can be maintained on the disk. We can organize the indexes into chunks, while each chunk only contains a subset of the whole index. In the query answering, only those related inverted lists, or the related keywords and the answer sets on the related edges need to be loaded into the main memory. Since the number of keywords in a query is often small, the I/O cost is low.

Recently, most of the modern computers have large main memories (typically several gigabytes). In general, the whole keyword graph index is small enough to be held in the main memory. As shown later in the performance comparisons, the query answering algorithm using the keyword graph index can achieve the smallest response time. In practice, which query answering algorithm to choose really depends on the requirements on the response time and the memory usage. If the response time is more crucial than the memory usage, the keyword graph algorithm is a better choice.

We will examine the index construction cost in more detail using synthetic data sets.

Aggregate Keyword Queries – Complete Matches

We tested a large number of aggregate keyword queries, which include a wide variety of keywords and their combinations. We considered factors like the frequencies of keywords, the size of the potential minimal answers to be returned, the text attributes in which the keywords appear, etc.

We first focus our performance evaluation on a representative test set of 12 queries here. Our test set has 12 queries (denoted by Q_1 to Q_{12}) with query length ranging from 2 to 5. Among them, each length contains 3 different queries. Note that the query Q_{i+3} $(1 \leqslant i \leqslant 9)$ is obtained by adding one more keyword to the query Q_i. In this way, we can examine the effect when the number of query keywords increases. The queries are shown in Table 3.4.

We list in Table 3.4 the number of minimal answers for each query. When the number of query keywords increases from 2 to 3, the number of minimal answers may increase. The minimal answers to a two keyword query are often quite specific tuples. When the third keyword is added, for example, query Q_4, the minimal answers have to be generalized. One specific answer can generate many ancestors by combining with other tuples. Query Q_3, however, is an exception, since most of the movies containing keywords “mafia-boss” and “Italy” also contain “revenge”. Thus, many minimal answers to Q_3 are also minimal answers to Q_6.

When the number of query keywords increases further (for example, more than 3 keywords), the number of minimal answers decreases. When some new keywords are added into the query, the minimal answers are becoming much more general. Many combinations

Query ID	Query Keywords in Text Attributes			# answers
	Genre	Keyword	Location	
Q_1	Action	explosion	/	1,404
Q_2	Comedy	/	New York	740
Q_3	/	mafia-boss	Italy	684
Q_4	Action	explosion, war	/	2,109
Q_5	Comedy	family	New York	1,026
Q_6	/	mafia-boss, revenge	Italy	407
Q_7	Action	explosion, war	England	724
Q_8	Comedy	family, christmas	New York	308
Q_9	Crime	mafia-boss, revenge	Italy	341
Q_{10}	Action	explosion, war, superhero	England	215
Q_{11}	Comedy	family, christmas, revenge	New York	0
Q_{12}	Crime	mafia-boss, revenge, friendship	Italy	43

Table 3.4: The aggregate keyword queries.

of tuples may generate the same minimal answers. The number of possible minimal answers decreases.

As discussed in Chapter 2, the existing studies about keyword search on relational databases and our work address different types of keyword queries on relational databases. The existing methods cannot be extended straightforwardly to tackle the aggregate keyword search problem. Thus, we do not conduct the performance comparison with those existing keyword search algorithms on relational databases. Alternatively, to examine the efficiency of our methods, we compare our methods to the following baseline algorithm which is an extension of the conventional iceberg cube computation [9]. Consider Example 3 and the corresponding lattice in Figure 3.1, we compute the cells using the processing tree in Figure 3.4. The numbers in Figure 3.4 indicate the orders in which the baseline algorithm visits the group-bys.

The baseline algorithm first produces the empty group-by. Next, it partitions on dimension A. Since there are two distinct values on dimension A (that is, a_1 and a_2), we produce two partitions $\langle a_1 \rangle$ and $\langle a_2 \rangle$. Then, the baseline algorithm recurses on partition $\langle a_1 \rangle$. The $\langle a_1 \rangle$ partition is aggregated and produces a cell $c_1 = (a_1, *, *)$ for the A group-by. To examine whether c_1 is a valid answer, we need to examine whether all the query keywords w_1 and w_2 appear in some tuples in the cover $Cov(c_1)$. To efficiently achieve that goal, for each

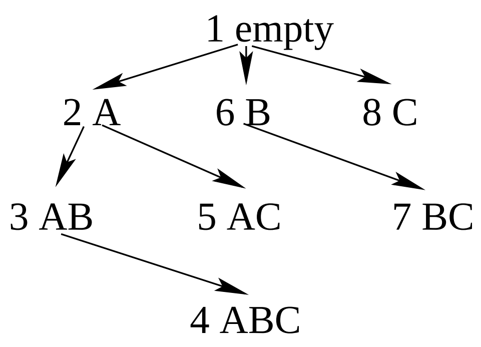


Figure 3.4: The processing tree of the baseline algorithm.

keyword w in the database, we maintain an inverted list $IL(w)$. For each keyword w in the query, we scan $Cov(c_1)$ once to examine if there exists at least one tuple in $IL(w)$. If it is not the case, we conclude that cell c_1 is not a valid answer, and the baseline algorithm stops the current loop, since there is no need to further examine the children in the processing tree. If all keywords in the query appear in $Cov(c_1)$, c_1 is a valid answer, and it is placed in the candidate answer set. We continue to partition the $\langle a_1 \rangle$ partition on dimension B. It recurses on the $\langle a_1, b_1 \rangle$ partition and generates a cell $c_2 = (a_1, b_1, *)$ for the AB group-by. The baseline algorithm again examines whether c_2 is a valid answer. If c_2 is a valid answer due to c_2 being more specific than c_1, c_1 cannot be a minimal answer, thus, c_1 is removed from the candidate answer set and c_2 is placed into the candidate answer set. Similarly, we process partitions $\langle a_1, b_1, c_1 \rangle$ and $\langle a_1, b_1, c_2 \rangle$. The baseline algorithm then recurses on the $\langle a_1, b_2 \rangle$ partition. When this is completed, it partitions the $\langle a_1 \rangle$ partition on dimension C to produce the $\langle a_1, C \rangle$ aggregates.

After we obtain the candidate answer set, the baseline algorithm needs to scan the candidate answer set once and remove any non-minimal answers. Finally, the baseline algorithm can find the complete set of minimal answers for an aggregate keyword query.

To load the whole keyword graph index into the main memory, it took 9.731 seconds. On average, the loading time for one specific edge in the keyword graph index is 0.032 milliseconds, which is a very small number. In the experimental evaluation, we focus only on the efficiency of the query answering algorithms. Thus, for the response times with respect to different queries in the following analysis, we ignore the I/O costs and only consider the response time of the query answering algorithms.

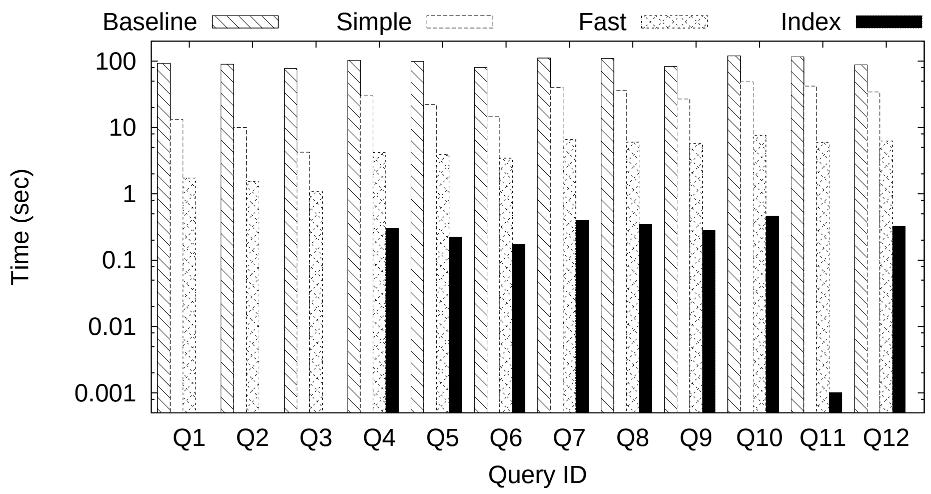


Figure 3.5: Query time of queries with different sizes.

Figure 3.5 compares the query answering time for the 12 queries using the baseline algorithm (denoted by *Baseline*), the simple nested loop algorithm (Algorithm 1, denoted by *Simple*), the fast maximum join algorithm (Algorithm 2, denoted by *Fast*), and the keyword graph index method (denoted by *Index*). The time axis is drawn in logarithmic scale.

The baseline algorithm performs the worst among the four methods. The major reason is that the baseline algorithm needs to compute the whole lattice. Moreover, the baseline algorithm needs to consider all the tuples in the database. In practice, given a keyword query, only a part of tuples are needed to be considered. The simple nested loop algorithm performs better than the baseline algorithm, but it is the worst among the three methods other than the baseline algorithm. The fast algorithm adopts several speed up strategies, thus, its query time is about an order of magnitude shorter than the simple nested loop algorithm. The keyword graph index based algorithm is very fast in query answering. When the number of query keywords is 2, we can directly obtain the minimal answers from the edge labels, and thus, the query answering time is ignorable. When the number of query keywords is at least 3, the query time is about 20 times shorter than the fast maximum join algorithm. One reason is that the keyword graph index method already calculates the minimal answers for each pair of keywords, thus, given m query keywords, we only need to conduct maximum joins on $m-1$ edges. Another reason is that only the minimal answers

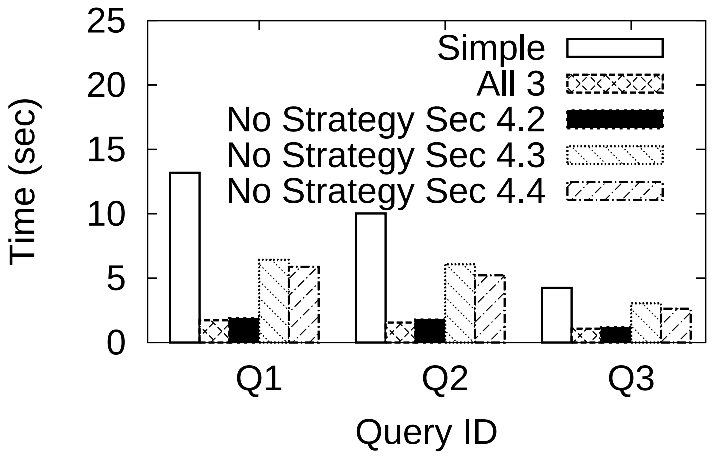


Figure 3.6: Effectiveness of each pruning strategy in the maximum-join method.

stored on the edges participate in the maximum joins, which are much smaller than the total number of tuples involved in the maximum join methods.

Generally, when the number of query keywords increases, the query time increases, since more query keywords lead to a larger number of maximum join operations. It is interesting to find that when the number of query keywords increases, the query time of the baseline algorithm does not increases greatly. The reason is that the size of queries is not the bottleneck in the baseline algorithm. The major time-consuming part of the baseline algorithm is to compute the whole lattice.

Using the keyword graph index, the query time for Q_{11} is exceptionally small. The reason is that by examining the keyword graph index, the 5 query keywords in Q_{11} do not form a clique in the graph index, thus we even do not need to conduct any maximum join operations. The results confirm that the query answering algorithm using keyword graph index is efficient and effective.

We also examine the effectiveness of the speed up strategies in Algorithm 2. In this chapter, we only show the basic cases Q_1, Q_2 and Q_3 each of which has 2 keywords. The results for other queries are similar. In Figure 3.6, for each query, we tested the query answering time of adopting all three speed up strategies, as well as leaving one strategy out.

All the speed up strategies contribute to the reduction of query time. However, their contributions are not the same. The strategy of reducing matching candidates (Section 3.3.3) contributes the best, since it can reduce the candidate tuples to be considered greatly.

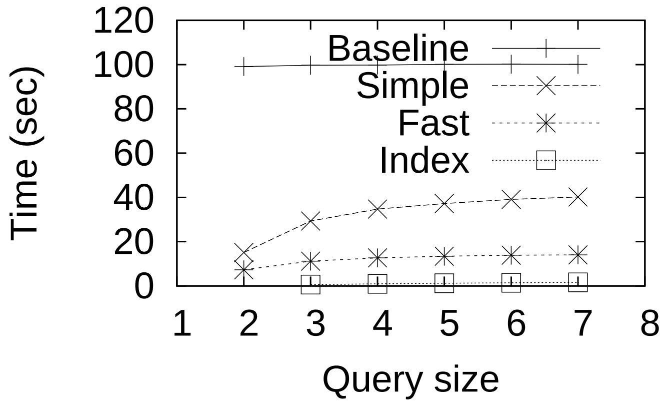


Figure 3.7: Query time of queries with different sizes (random queries).

The fast minimal answer checking strategy (Section 3.3.4) removes non-minimal answers. Comparing to the nested-loop based superset checking, it is much more efficient. The effect of pruning exactly matching tuples (Section 3.3.2) is not as much as the other two, since the exact matching tuples are not frequently met.

In the above analysis, we focus on a set of 12 representative queries. To examine the performance of the query answering algorithms in general situations, we also conduct the performance evaluation on a large set of randomly generated queries.

We first examine the effect of the query size. The query size plays an important role in query answering. In practice (e.g., Web search), the number of keywords in a keyword query is relatively small (e.g, 2 or 3 words in Web search queries [77]). As explained in Section 3.3, the running time of our algorithms are highly related to two factors: (1) the rounds of maximum joins (the maximum join algorithm) as well as the number of edges to be examined in the keyword graph index (the keyword graph algorithm); (2) the size of the intermediate minimal answers after each round of maximum joins. When the number of keywords in the query increases, the first factor obviously increases as well. However, the second factor does not increases always. At some stage, the number of intermediate minimal answers achieves the maximum value; then, it decreases even more keywords appear in the query. This can be verified based on the results shown in Table 3.4. In general, in the IMDB data set, the size of the minimal answers is largest when the number of keywords in the query is equal to 3.

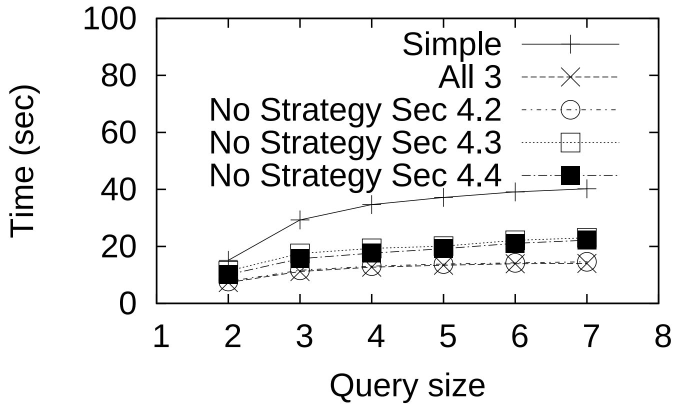


Figure 3.8: Effectiveness of each pruning strategy in the maximum-join method (random queries).

In Figure 3.7, we plot the running time of the two algorithms with respect to different sizes of queries. For each fixed query size, the running time refers to the average time of 100 randomly generated queries. When the number of keywords is small (e.g., less than or equal to 3), the increase of the running time is large; however, when the number of keywords is large (e.g., larger than 4), the increase of the running time is small. The results in Figure 3.7 also verify that even if the size of the query is very large, our algorithms still can achieve reasonably good performance.

We next examine the effect of each pruning strategy in the maximum-join method using randomly generated queries. In Figure 3.8, we plot the running time of the queries with different sizes in terms of adopting all three speed up strategies, as well as leaving one strategy out. For each fixed query size, the running time refers to the average time of 100 randomly generated queries. The results are similar to those in Figure 3.6.

As we mentioned in Section 3.3.5, for a keyword query which contains m different keywords, we need to conduct $m - 1$ rounds of maximum joins. The orders of maximum joins is crucial to the performance of the query answering algorithms. In this chapter, we adopt a simple heuristic, that is, in each round of maximum joins, we pick two inverted lists with the smallest sizes.

The heuristic used in our algorithm is very simple, and it achieves relatively good performance. For a performance comparison, we compared our simple heuristic to a random

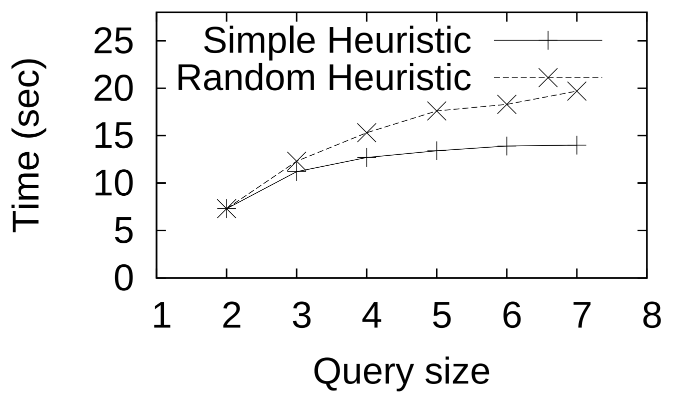


Figure 3.9: Effectiveness of the heuristic for the orders of maximum joins (random queries).

Query size	2	3	4	5	6	7
# queries	100	100	100	100	100	100
# queries that the traditional keyword search methods cannot find a single matching tuple	61	82	87	88	93	95
# queries that the aggregate keyword search cannot find a non-trivial answer	2	6	7	11	13	13

Table 3.5: The effectiveness of the aggregate keyword queries.

selection method (that is, for a query containing m keywords, we randomly pick two words and get the corresponding lists of tuples for the next round of maximum joins). In Figure 3.9, we show the performance of the maximum-join algorithm using the **Simple Heuristic** we developed in this chapter, as well as that using the **Random Method**. All the pruning strategies are adopted. The simple heuristic clearly outperforms the random method.

We examine the effectiveness of the proposed aggregate keyword search using randomly generated queries. We vary the number of keywords in the query from 2 to 7; and for each fixed query size, we randomly generate 100 different queries. For each query, we examine whether there exists a single tuple matching all the keywords in the query. If there does, the traditional keyword search algorithms on relational databases can retrieve some results; if not, the traditional keyword search algorithms cannot find any results. Table 3.5 shows the results. When the number of keywords in a query is not very small (e.g., more than 3), the majority of queries cannot find any single tuple matching all the keywords in the query.

Query size	2	3	4	5	6	7
# queries	100	100	100	100	100	100
# queries that the aggregate keyword search cannot find a non-trivial answer	2	6	7	11	13	13
Percentage of queries that the aggregate keyword search cannot find a non-trivial answer	2%	6%	7%	11%	13%	13%
Average # matched keywords using partial keyword matching	1	2	2.86	3.73	4.23	4.85

Table 3.6: The effectiveness of the partial keyword matching.

However, the aggregate keyword search is still able to find some useful results even if the query keywords do not appear in a tuple together. In the worst case, the aggregate keyword search may just return a trivial answer (that is, an answer with $*$ on all the dimensions). We count the number of queries when the aggregate keyword search only returns a trivial answer, and the results are shown in Table 3.5. In general, only a small number of queries cannot find a non-trivial answer using the aggregate keyword search. The results clearly indicate that the aggregate keyword search is quite useful in practice.

Partial Keyword Matching Queries

We first examine the effectiveness of the partial keyword matching. We use the same set of randomly generated queries in Section 3.6.1. As discussed in Section 3.6.1, some aggregate keyword queries may return a trivial answer that is not informative to users. For those queries which cannot find a non-trivial answer using the aggregate keyword search, we adopt the partial keyword matching technique. We count the maximal number of keywords in the query that can be matched using the partial keyword matching. If most of the keywords in the query can be matched, the answers returned by the partial keyword matching may still be interesting to users. Table 3.6 shows the results. In general, the majority of keywords in the query are matched using the partial keyword matching. The results indicate that the partial keyword matching is useful to find some good answers even if the aggregate keyword search cannot find a non-trivial answer.

We conducted the experiments to evaluate our partial keyword matching queries using the keyword graph index. To simulate the partial keyword matching scenario, we used queries Q_7, Q_8 and Q_9 in Table 3.4 as the base queries and add some irrelevant keywords into each query. Specifically, for each query, we manually added 1, 2 and 3 irrelevant keywords

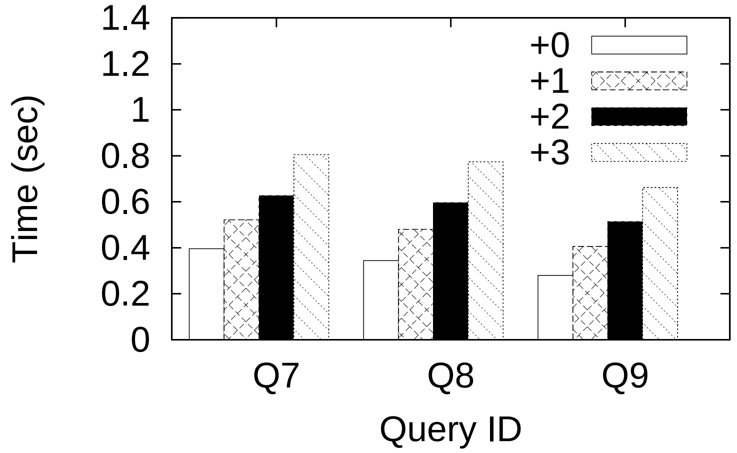


Figure 3.10: Query time for partial keyword matching queries.

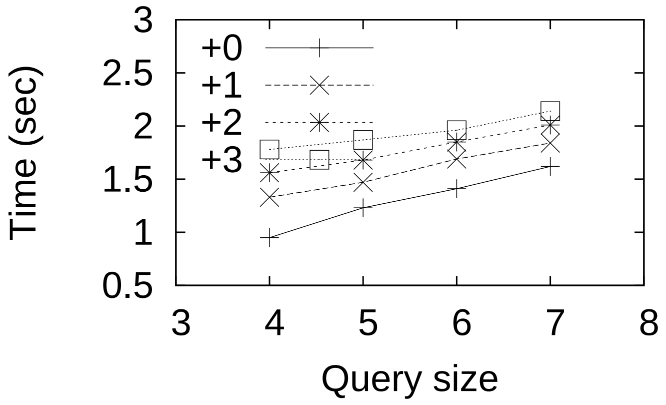


Figure 3.11: Query time for partial keyword matching queries (random queries).

into the query to obtain the extended queries, and make sure that any of those irrelevant keywords do not have non-trivial complete match answers together with the keywords in the base query.

Our method returns the answers to the base queries as the partially match answers to the extended queries. The experimental results confirm that our partial matching method is effective. Moreover, Figure 3.10 shows the runtime of partial matching query answering. When the number of irrelevant query keywords increases, the query time increases, because more cliques need to be considered in the query answering.

We also examine the performance of the partial matching using randomly generated

Query size	2	3	4	5	6	7
# queries	100	100	100	100	100	100
# queries that the aggregate keyword search cannot find a non-trivial answer	2	6	7	11	13	13
Percentage of queries that the aggregate keyword search cannot find a non-trivial answer	2%	6%	7%	11%	13%	13%
Average # rounds of specialization	2	2.5	2.71	3.09	3.46	3.62

Table 3.7: The effectiveness of the hierarchical keyword matching.

queries. Figure 3.11 shows the results of the running time with respect to different sizes of queries. For each randomly generated query, we randomly added 1, 2 and 3 irrelevant keywords into the query to obtain the extended queries. The results are similar to those in Figure 3.10.

Hierarchical Keyword Matching Queries

We first examine the effectiveness of the hierarchical keyword matching. We use the same set of randomly generated queries in Section 3.6.1. As discussed in Section 3.6.1, some aggregate keyword queries may return a trivial answer that is not informative to users. For those queries which cannot find a non-trivial answer using the aggregate keyword search, we adopt the hierarchical keyword matching technique. We count the number of rounds that the specification needs to be conducted. Too many rounds of specializations may not lead to a good answer loyal to the original query objective. Table 3.7 shows the results. In general, only a few rounds of specification are needed for the hierarchical keyword matching technique to find a non-trivial answer. The results indicate that the hierarchical keyword matching is useful to find some good answers even if the aggregate keyword search cannot find a non-trivial answer.

We also conducted experiments to evaluate our hierarchical keyword matching methods using the keyword graph index. A critical issue is to build a keyword hierarchy for the keywords in the data set. Since the number of keywords is large, we adopted the WordNet taxonomy [29] as the hierarchy. WordNet is a semantic lexicon for the English language. It groups English words into sets of synonyms. We queried the WordNet taxonomy database for each keyword, and built the hierarchical relations among them according to the WordNet synonyms.

To simulate the hierarchical keyword matching scenario, we used queries Q_7, Q_8 and

Q_9 in Table 3.4 as the base queries, and replaced some keywords in those queries by their ancestors in the hierarchy. Specifically, for each query, we manually replaced 1, 2 or 3 keywords with some ancestor keywords to obtain the extended queries, and make sure that the keywords in each extended query do not form a clique in the keyword graph index.

When only 1 keyword is replaced with an ancestor keyword, among the 3 queries, all the returned results match the answers to the base queries. When the number of keywords to be replaced increases, however, the results become weaker. In the case of 2 keywords are replaced, only the extended queries based on Q_7 and Q_8 return the expected answers. In the case of 3 keywords are replaced, only the extended query based on Q_7 returns the expected answer. The reason is, when more keywords are replaced, when we select a keyword to specialize, the number of choices increases, thus the probability to return the expected answers becomes smaller. However, our hierarchical keyword matching algorithm can still find meaningful answers. For example, Q_4 is a 3-keyword query {"Animation", "Animal", "USA"}. There is no exact answer in the table for this query. We can find an approximate keyword matching {"Animation", "lions", "Walt Disney World"}, and one answer with the movie title "The Lion King" will be returned.

Figure 3.12 shows the query answering time in hierarchical matching. Similar to the partial matching method, when the number of query keywords to be replaced increases, the query time increases. This is because more calculation for extendability score is needed. Generally, the hierarchical keyword matching queries can be answered efficiently using the keyword graph index.

We also examine the performance of the hierarchical matching using randomly generated queries. Figure 3.13 shows the results of the running time with respect to different sizes of queries. For each randomly generated query, we randomly replaced 1, 2 or 3 keywords with some ancestor keywords to obtain the extended queries. The results are similar to those in Figure 3.12.

3.6.2 Results on Synthetic Data Sets

To test the efficiency and the scalability of our aggregate keyword search methods, we generated various synthetic data sets. In those data sets, we randomly generated 1 million tuples for each data set. We varied the number of dimensions from 2 to 10. We tested the data sets of the cardinalities 100 and $1,000$ in each dimension. Since the number of text attributes does not affect the keyword search performance, for simplicity, we only generated

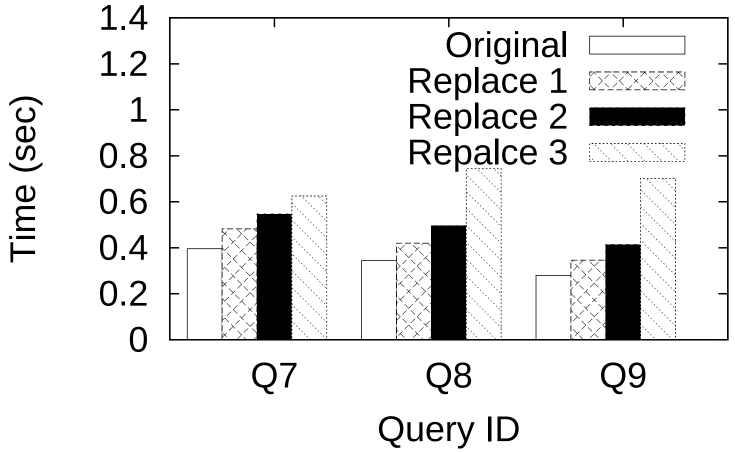


Figure 3.12: Query time for hierarchical keyword matching queries.

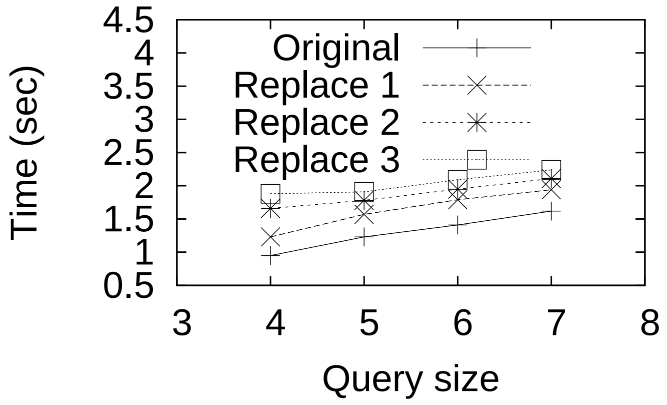
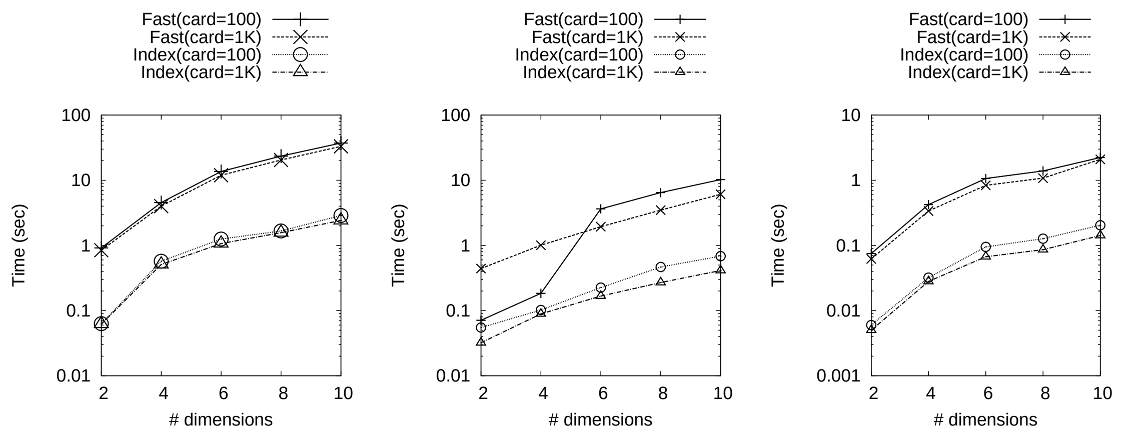


Figure 3.13: Query time for hierarchical keyword matching queries (random queries).

1 text attribute. Each keyword appears only once in one tuple. We fixed the number of keywords in the text attribute for each tuple to 10, and varied the total number of keywords in the data set from $1,000$ to $100,000$. The keywords are distributed uniformly except for the experiments in Section 3.6.2. Thus, on average the number of tuples in the data set that contain one specific keyword varied from $10,000$ to 100. We also use the packing technique [9] to represent the data sets.



Number of keywords is $1K$. Number of keywords is $10K$. Number of keywords is $100K$.

Figure 3.14: Query time of the two methods on different synthetic data sets.

Efficiency and Scalability

To study the efficiency and the scalability of our aggregate keyword search methods, we randomly picked 10 different keyword queries, each of which contains 3 different keywords. Figure 3.14 shows the query answering time.

The keyword graph index method is an order of magnitude faster than the fast maximum join algorithm. The simple nested loop method is an order of magnitude slower than the fast maximum join method. To make the figures readable, we omit them here.

The number of dimensions, the cardinality of the dimensions and the total number of keywords affect the query answering time greatly. In general, when the number of dimensions increases, the query answering time increases. First, the maximum join cost is proportional to the dimensionality. Moreover, the increase of runtime is not linear. As the dimensionality increases, more minimal answers may be found, thus more time is needed. When the cardinality increases, the query answering time decreases. The more diverse the dimensions, the more effective of the pruning powers in the maximum join operations. The total number of keywords in the text attribute highly affects the query time. The more keywords in the table, on average less tuples contain a keyword.

We generated the keyword graph index using Algorithm 3. The keyword graph generation is sensitive to the number of tuples in the table. We conducted the experiments on 10 dimensional data with cardinality of $1,000$ on each dimension, set the total number of

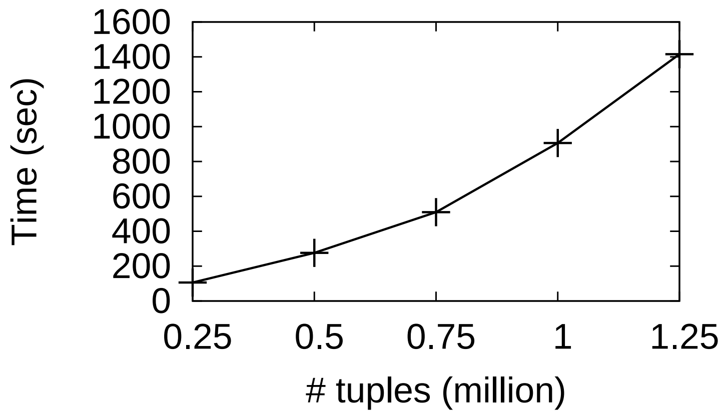


Figure 3.15: Running time for the keyword graph index generation.

keywords to $10,000$, and varied the number of tuples from 0.25 million to 1.25 million. The results on runtime are shown in Figure 3.15. The runtime increases almost linearly as the number of tuples increases, since the number of maximum join operations and the number of answers generated both increase as the number of tuples increases.

Figures 3.16 and Figure 3.17 examine the size of the keyword graph index with respect to the number of tuples, where the settings are the same as Figure 3.15. To measure the index size, we used the number of edges in the graph and the average number of minimal answers on each edge. Generally, when the number of tuples increases, the number of edges increases because the probability that two keywords have a non-trivial answer increases. Meanwhile, the average number of minimal answers on each edge also increases because more tuples may contain both keywords.

Skewness

The aggregate keyword search methods are sensitive to skewness in data. In all of the previous experiments, the data was generated uniformly. We ran an experiment on the synthetic data set with 1 million tuples, 10 dimensions with cardinality $1,000$, and a total number of $10,000$ keywords. We varied the skewness simultaneously in all dimensions. We used the Zipf distribution to generate the skewed data. Zipf uses a parameter α to determine the skewness. When $\alpha = 0$, the data is uniform, and as α increases, the skewness increases

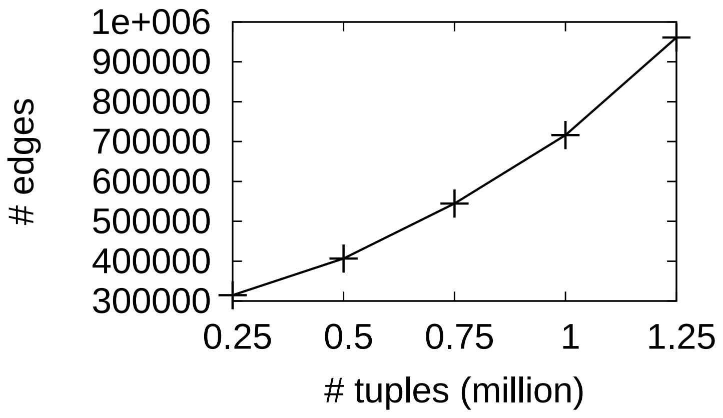


Figure 3.16: The number of edges in the keyword graph index.

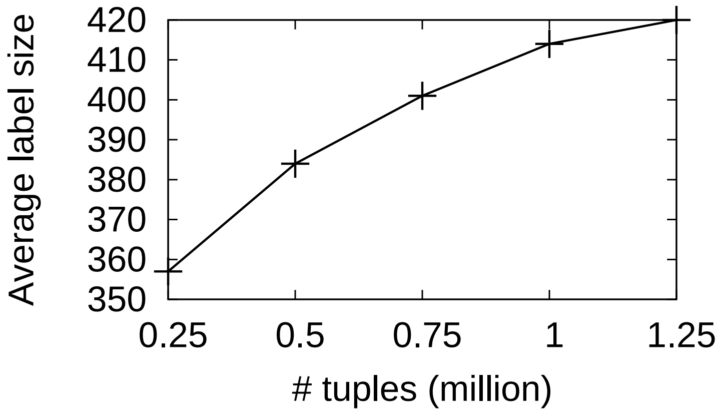


Figure 3.17: The average length of edge labels in the keyword graph index.

rapidly: at $\alpha = 3$, the most frequent value occurs in about 83% of the tuples. We randomly picked 10 different keyword queries with query size 3. The average query answering time is shown in Figure 3.18. The performance of the two methods becomes as the skewness on dimensions increases. However, the keyword graph index method still performs well.

Skewness may occur on text attributes, too. We ran another experiment on the data set with 1 million tuples, 10 dimensions with cardinality of $1,000$, and a total number of $10,000$ keywords. We made the skewness happen in the text attribute only. We also used a Zipf distribution. We randomly picked 10 different keyword queries with query size 2, in which one keyword has a high frequency and the other does not. The average query

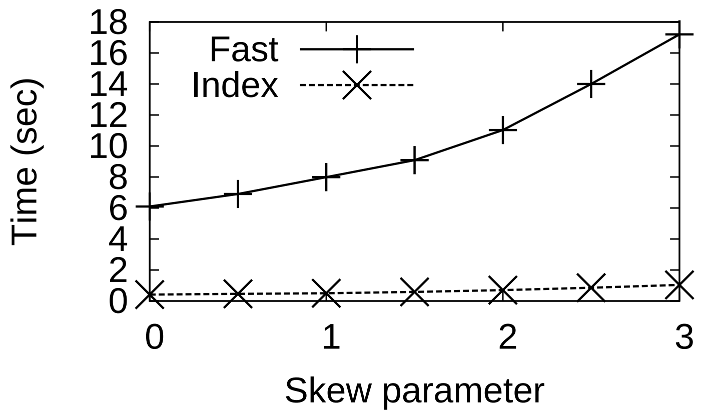


Figure 3.18: Skew on dimensional attributes.

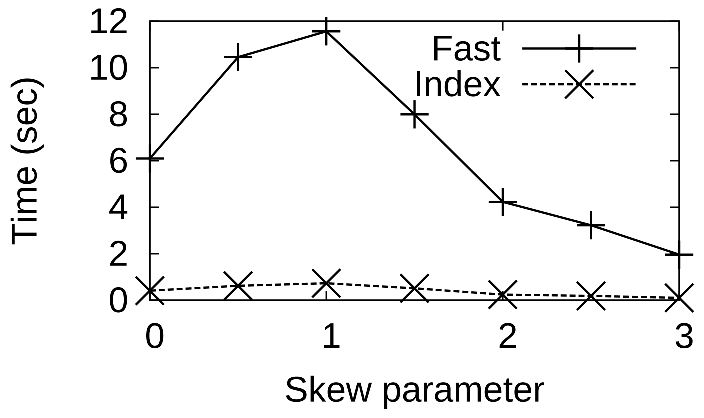


Figure 3.19: Skew on text attribute.

answering time is shown in Figure 3.19. When the parameter α is small (for example, 0.5), the query answering time increases when the data becomes skewed. This is because the tuples containing the frequent keyword in a query increases dramatically. However, when α is further larger, the query answering time decreases because the number of tuples containing the infrequent keyword in a query decreases dramatically. The query answering time is dominated by the infrequent keyword.

In summary, our experimental results on both real data and synthetic data clearly show that aggregate keyword queries on large relational databases are highly feasible. Our methods are efficient and scalable in most of the cases. Particularly, the keyword graph index approach is effective.

3.7 Conclusions

In this chapter, we identified a novel type of aggregate keyword queries on relational databases. We showed that such queries are useful in some applications. We developed the maximum join approach and the keyword graph index approach. Moreover, we extended the keyword graph index approach to address partial matching and hierarchical matching. We reported a systematic performance study using real data sets and synthetic data sets to verify the effectiveness and the efficiency of our methods.

The techniques developed in this chapter are useful in some other applications. For example, some techniques in this chapter may be useful in KDAP [82]. As future work, we plan to explore extensions of aggregate keyword queries and our methods in those applications. Moreover, in some applications, a user may want to rank the minimal answers in some meaningful ways such as finding the top-k minimal answers. It is interesting to extend our methods to address such a requirement.

Chapter 4

Query Suggestion for Keyword Search on Graphs

As discussed in Chapter 2, in recent years, many studies about keyword search in various aspects have been conducted on structured and semi-structured data, especially on relational databases and graphs. Several recent studies [3, 41] have shown that relational databases can also be modeled as graphs. In the graph model, nodes refer to entities, such as a tuple in a relational database; and edges refer to relationships among entities, such as primary-foreign key relationships in a relational database. A keyword query on graphs tries to assemble semantically relevant connections in the graph among those keywords in the query (e.g., finding embedded trees or subgraphs connecting all the keywords in the query).

With the assistance of keyword search, both experienced users and inexperienced users can easily retrieve valuable information from large-scale data. However, a keyword query typically is very short, which on average only contains two or three keywords [77]. As a result, keyword queries in practice are often ambiguous. To improve the effectiveness of keyword search and assist users' search process, most popular commercial Web search engines currently provide a useful service called *query suggestion*. By conjecturing a user's real search intent, the Web search engines can recommend a set of related keyword queries which may better capture the user's real information needs.

Query suggestion has been shown an important and powerful tool for the Web search engines to achieve good search performance. Although the problem of keyword search on graphs has been studied for several years, and several prototype systems supporting keyword search on graphs have been developed, as far as we known, there is no previous work on systematically analyzing query suggestion methods for keyword search on graphs.

Due to the ambiguity of keyword queries, as well as the complex structures of answers to keyword search on graphs, query suggestion becomes very important to improve the search performance of keyword search on graphs. In order to provide meaningful recommendations for keyword search on graphs, several critical challenges need to be properly addressed.

- First, it is necessary to maintain a clear understanding of different scenarios when query suggestion becomes necessary to capture users' real information needs. For example, query suggestion is necessary if the size of the answer set is too small. An extreme case refers to the *failing query* [57], where the answer set to a keyword query is empty. The previous studies about keyword search on graphs cannot properly deal with the issue of failing query, since they just notify the user there is no valid answer at all. Such a simple negative response will easily frustrate the users' enthusiasm, and at last, results in bad user search experience. Alternatively, it is better to recommend a set of related queries to users to assist their search process.

  In addition to the scenario of failing query, are there any other scenarios that query suggestion is necessary? Understanding the different scenarios for query suggestion is crucial to develop query suggestion methods for keyword search on graphs.

- Second, it is important to identify the relationships among different scenarios for query suggestion. Since there are several different scenarios for query suggestion, a straightforward solution is to develop some specific query suggestion methods for each scenario. However, it is obviously neither practical nor efficient.

  Although different scenarios for query suggestion exist, it is possible that different scenarios have some underlying relationships. Once such relationships can be identified, we only need to develop specific query suggestion methods for some scenarios, and the remaining ones could be solved straightforwardly.

- Third, it is possible to exploit the data-driven query suggestion methods for keyword search on graphs. Most of the query suggestion methods for Web search engines utilize large collections of search logs. Search logs record the history of search behaviors from millions of users, which are beneficial for providing high quality of query suggestions. However, search logs for keyword search on graphs are not largely available at this moment. There are no publicly available search logs on graphs. Moreover, users' search activities on graphs may be quite different from those through Web search engines. Thus, search logs from Web search engines cannot be used to analyze users' search behavior on graphs.

The data-driven query suggestion methods in Web search engines utilize the data itself. The central idea is to apply data mining techniques to identify the correlations among data, and select closely related queries as recommendations. Thus, can we develop some data-driven query suggestion methods on graphs? If so, what kind of information should we extract from graphs to provide query suggestion?

- Last but not least, it is critical to provide online query suggestion for keyword search on graphs. The query response time is a crucial performance measure for different search systems. It is not desirable for a user to wait a long time to obtain the suggestions. Can we design efficient solutions of query suggestion for keyword search on graphs?

In this chapter, we take an initiative towards the challenging problem of query suggestion for keyword search on graphs. The remainder of the chapter is organized as follows. In Section 4.1, we formulate the problem of query suggestion for keyword search on graphs. We present the solution framework in Section 4.2 and describe our recommendation methods in Section 4.3. A systematic performance study is reported in Section 4.4. Section 4.5 concludes the chapter.

4.1 Problem Definition: Query Suggestion for Keyword Search on Graphs

In this section, we first introduce the graph model of the data and the problem of keyword search on graphs, then we formalize the problem of query suggestion for keyword search on graphs.

4.1.1 Graph Model of the Data

Graphs can be used to represent complex data structures. For instance, the Web can be modeled as a directed graph $G = (V, E)$, in which each node $v_i \in V$ represents one page on the Web, and two nodes v_i and v_j are connected by a directed edge $e_{ij} = e_i \to e_j$ if there exists a hyperlink from page v_i to page v_j on the Web. Some other typical graphs include social networks, road networks, mobile phone communication networks, etc.

Several recent studies [3, 41, 10, 40, 45, 52, 47, 54, 39, 20, 49, 61] have shown that the structured data and semi-structured data can also be modeled using graphs. For instance, several keyword search prototype systems (e.g., DBXplorer [3], Discover [41], BANKS [10], BLINKS [39], and SPARKS [54]) have been developed to use the graph model to represent the relational databases. In the graph model, nodes refer to entities, such as a tuple in a relational database; and edges refer to relationships among entities, such as primary-foreign key relationship in relational databases.

Depending on specific applications, either directed graph model or undirected graph model can be adopted. For instance, while some systems (e.g., DBXplorer [3] and Discover [41]) model the data as an undirected graph, some other systems (e.g., BANKS [10], BLINKS [39], and SPARKS [54]) model the data as a directed graph.

Without loss of generality, in this chapter, we take the directed graph model to represent the data. The major reason is that the tightness of connections between two nodes in the graph is not necessary to be symmetric, thus, modeling directionality in graphs becomes a natural strategy in many applications. Moreover, the undirected graph model can be considered as a special case of the directed graph model, since each undirected edge in an undirected graph can be represented using two directed edges in the corresponding directed graph.

The nodes in a graph contain a finite set of keywords. For instance, in the Web graph, the keywords that each node associated with can be those keywords appearing in the body of the corresponding HTML source file; in the graph model of a relational database, the keywords that each node associated with can be those keywords extracted from the textual attributes of the corresponding tuple in the relational database.

Definition 7 (Graph Model of the Data) *The complex data are modeled using a directed graph $G = (V, E)$. Each node $v \in V$ represents an entity in the data. For a pair of nodes $v_i, v_j \in V$, there exists a directed edge $e_{ij} = v_i \to v_j$ if the corresponding entities of v_i and v_j in the data have a specific relation.*

The dictionary, D, is a finite set of keywords that appear in the data. There is a node-to-keyword mapping $\phi : V \to 2^D$, which lists the finite set of keywords contained in a node. We use $W(v)$ to represent the set of keywords that a node v contains. The number of distinct

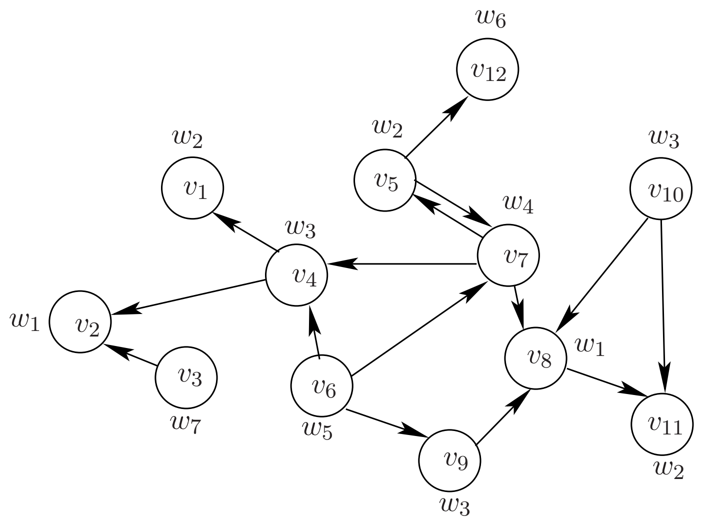


Figure 4.1: Keyword search on a directed graph.

keywords contained in a node v is denoted as $|W(v)|$.

There is also a keyword-to-node mapping $\kappa : D \to 2^V$, which lists the finite set of nodes that contain the keyword. We use $S(w)$ to represent the set of nodes that contain a keyword w. The number of distinct nodes containing keyword w is denoted as $|S(w)|$.

There is a cost function $\delta : E \to \mathbb{R}^+$, which models the distance of the nodes between each other. ■

For simplicity, in this chapter, we assume that each edge in a graph has the same weight. This assumption can be easily removed by taking into account the edge weights when defining the ranking functions.

Example 8 (Graph Model) *Consider the graph in Figure 4.1. There are 12 nodes in the graph, and each node is labeled with a set of keywords. For example, $W(v_1) = \{w_2\}$. Moreover, a keyword may appear in a set of different nodes. For example, $S(w_1) = \{v_2, v_8\}$. Furthermore, each edge has the same weight in the graph, thus, we do not explicitly label the edge weight in the graph.* ■

4.1.2 Keyword Search on Graphs

Keyword search on graphs tries to assemble semantically relevant connections in the graph among those keywords in the query. In the directed graph model, an answer to a keyword query on graphs may have different types of semantics, such as a Steiner tree [23], an r-radius subgraph [49], or a community [62]. Following the studies in BLINKS [39] and many recent ones, in this chapter, we adopt the rooted tree semantic.

Definition 8 (Rooted Tree Semantic) *Given a keyword query $q = \{w_1, w_2, \ldots, w_m\}$ and a directed graph G, an answer to q is a rooted tree with a root node r and m leaf nodes, represented as $n_1, \ldots, n_m$, where r and n_i's are nodes in G (may not be distinct) satisfying the following two properties:*

Containment: *$\forall i$, n_i contains keyword w_i, that is, $w_i \in W(n_i)$;*

Connectivity: *$\forall i$, there exists a directed path in G from r to n_i.* ■

Intuitively, in the rooted tree semantic, the directed paths from the root node to the leaf nodes in an answer describe how the keywords are semantically related in the graph.

We can assign numerical scores to each answer to a keyword query on graphs. In many cases, the score of an answer is a function of all the weights on the edges in the answer. Several models of the scoring function have been proposed in the literature. For instance, the model used in [10] defines the score of an answer as the number of edges in the answer. The model used in [45] treats an answer as a set of paths, with one path per keyword, where each path starts from the root node and points to the leaf node that contains the keyword in the query. The score of an answer is defined as the sum of all the path lengths in the answer. In general, if the scoring function of the answers is monotonic, the keyword search algorithms on graphs are not affected by which specific scoring function is used.

The problem of finding minimal rooted trees as answers to a keyword query on graphs is known as the group Steiner tree problem, which is NP-hard [33]. Some recent studies [45, 39] model the rooted tree semantic using the *set-of-paths* model, that is, each rooted tree is considered to be a set of paths, from the root node to each of the leaf node in the tree. The results in [45, 39] shown that this set-of-paths model allows keyword search queries on graphs to be answered in polynomial time. Moreover, the set-of-paths model avoids the situation of generating a large amount of similar answers with the same root node. As a result, we adopt the set-of-paths model for rooted trees in this chapter.

Accordingly, the set of answers under the set-of-paths model can be ranked based on the total path lengths in the answer.

Example 9 (Keyword Search on Graphs) *Consider the graph in Figure 4.1. For a keyword query $q = \{w_1, w_2, w_3\}$, nodes v_8, v_5, v_4 can be leaf nodes in an answer Ans while*

v_7 is the root node in Ans. Based on the set-of-paths model, this answer can be modeled as three distinct paths: $v_7 \to v_8$, $v_7 \to v_5$, and $v_7 \to v_4$. The total path lengths in Ans is equal to 3. ■

4.1.3 Query Suggestion for Keyword Search on Graphs

Many existing studies about keyword search on graphs focus on finding top-k answers to a keyword query, where all the valid answers are ranked according to a specific ranking function.

However, keyword queries are typically very short. Thus, in many situations, keyword queries are very ambiguous. Answers to a keyword query on graphs may be quite different from each other.

Example 10 (Different Answers) *Consider the graph in Figure 4.1. For a keyword query $q = \{w_2, w_3\}$, we can find a set of different answers, e.g., $Ans_1 = \{v_4 \to v_1, v_4\}$, $Ans_2 = \{v_6 \to v_7 \to v_5, v_6 \to v_9\}$, $Ans_3 = \{v_5, v_5 \to v_7 \to v_4\}$.*

However, those answers capture different relationships among the two keywords w_2 and w_3 in the query. For example, for Ans_1, there is a directed connection from keyword w_3 to keyword w_2. For Ans_3, there is a directed connection from keyword w_2 to keyword w_3 (through keyword w_4). For Ans_2, there is no directed connections between keywords w_2 and w_3. However, another keyword w_5 (contained in the root node v_6) has directed connections to both of them.

If the answers are ranked according to the total path lengths in the answers. Ans_1, Ans_2, Ans_3 have scores 1, 3, 2, respectively. If only top-2 answers are required, Ans_2 will not be returned. Thus, the relationships captured by Ans_2 cannot be identified. ■

Other than returning top-k answers on graphs, alternatively, a search system on graphs can recommend a set of related queries to users to assist their search process (e.g., help them formalize their real information needs). In Example 10, the search system may recommend a query $q' = \{w_2, w_3, w_5\}$ which can help users to find the information they want. Since keyword w_5 appears in the root node v_6 in the answer Ans_2, thus, Ans_2 is also an answer to query q'. Thus, query suggestion is important for the system to achieve good performance.

In order to recommend meaningful suggestions for keyword search on graphs, the query suggestion methods have to take into account different types of information sources. In general, all the existing query suggestion methods for Web search can be roughly classified into two categories: the usage-driven methods and the data-driven methods. The usage-driven query suggestion methods focus on the users' search activities. For example, search logs are widely used to trace the users' search behaviors. In Web search engines, a commonly used query suggestion method [5] is to find common queries in search logs following the current query and use those queries as suggestions.

While search logs are useful to analyze users' search activities, in the context of keyword search on graphs, search logs are not available at this moment. As far as we know, there is no publicly available large-scale search logs for keyword search on graphs. Moreover, the graph data and the Web pages are quite different in terms of structure, content, data format, and so on. The search behaviors on graphs are quite different from those through Web search engines. As a result, search logs from Web search engines cannot be used to analyze users' search activities on graphs.

To develop query suggestion methods for keyword search on graphs, in this chapter, we focus our attentions on data-driven query suggestion methods. The data-driven query suggestion methods utilize the data itself. Some data mining techniques are conducted on the data to identify the co-relations among entities, and then closely related entities are selected as recommendations.

Generally, a keyword query q on graphs carries a set of different search intents. One search intent represents one unique information need [56]. A crucial challenge for query suggestion is to precisely capture different search intents for a given keyword query. There are several possible ways to understand users' search intents. One straightforward solution is to build a universal knowledge corpus. Given a keyword query, all the possible search intents can be retrieved from the corpus. However, this solution needs a large scale keyword queries and a perfect understanding of the semantic relationships among those keywords. Another solution is letting the user provide some additional information, such as what specific semantic relationships the user is looking for. However, in such a case, a user needs to understand the graph data well enough, which may not be desirable. In Web search engines, the data-driven query suggestion methods utilize the data itself for mining different search intents. Thus, can we learn the search intents directly from the graph data?

For a keyword query q, an answer to q is a rooted directed tree connecting all the keywords in q. There may exist several answers which are quite different from each other in terms of the tree structures, keyword semantics, and so on. As shown in Example 10, different answers captures different semantic relationships among those keywords in q. Answers for the same search intent are likely to be similar to each other, while answers for different search intents are possibly quite different from each other. Thus, the search intents of a keyword query q can be captured if we can cluster all the valid answers to q.

Definition 9 (Search Intents) *For a keyword query q and a graph $G = (V, E)$, we assume there are n valid answers, denoted as $\mathcal{A} = \{Ans_1, Ans_2, \ldots, Ans_n\}$. A similarity function $Sim(Ans_i, Ans_j)$ is used to calculate the similarity between two answers Ans_i and Ans_j. A search intent C_i is captured by a subset of answers $C_i = \{Ans_{i_1}, Ans_{i_2}, \ldots, Ans_{i_{t_i}}\}$ ($1 \leqslant i_1 < i_2 < \ldots < t_i \leqslant n$), such that the answers in C_i are similar to each other while the answers in C_i and $\mathcal{A} - C_i$ are dissimilar, that is, $\forall Ans_{j_1}, Ans_{j_2} \in C_i$ and $Ans_{j_3} \notin C_i$, $Sim(Ans_{j_1}, Ans_{j_2}) > Sim(Ans_{j_1}, Ans_{j_3})$ and $Sim(AAnsj_1, Ans_{j_2}) > Sim(Ans_{j_2}, Ans_{j_3})$.* ■

In the context of keyword search on graphs, each answer is a directed rooted tree, there are many different ways to measure the similarity between two answers. We will discuss the similarity measures in more detail in Section 4.3.

We say a query q captures a search intent C_i if and only if: (1) each answer $Ans_j \in C_i$ is a valid answer to q; (2) if an answer $Ans_j \notin C_i$, Ans_j is not a valid answer to q.

Consider a keyword query q on a graph G, let us denote different search interests as $C_1, C_2, \ldots, C_k$, where k is a user-specified parameter representing the total number of possible search intents underlying q. The goal of query suggestion for keyword search on graphs is to recommend k related queries such that each search intent $C_i (1 \leqslant i \leqslant k)$ underlying q is captured as much as possible.

Definition 10 (Query Suggestion on Graphs) *Consider a directed graph $G = (V, E)$ in which each node $v \in V$ is associated with a set of keywords $W(v)$, an m-keyword query $q = \{w_1, w_2, \ldots, w_m\}$, and a user-specified parameter k, there are k different search intents underlying q, denoted as $C_1, C_2, \ldots, C_k$. The problem of query suggestion for keyword search on graphs is to find a set of k queries $Q = \{q_1, q_2, \ldots, q_k\}$ such that each query q_i captures one unique search intent C_i, where $1 \leqslant i \leqslant k$.* ■

4.2 Query Suggestion Scenarios and A Solution Framework

In this section, we first summarize all the possible scenarios that query suggestion is necessary when we conduct keyword search on graphs, then we provide a general solution framework to address the problem of query suggestion for keyword search on graphs.

4.2.1 Categorization of Scenarios for Query Suggestion

To develop query suggestion methods, it is critical to maintain a clear understanding of different situations when query suggestion becomes necessary to capture users' information needs. In practice, when a user submits a keyword query to a search system on graphs, there are several different scenarios which may result in the necessity of query suggestion.

- *The size of the answer set is too large.* It usually happens when some keywords in a search query are popular keywords contained in many nodes in the graph. If the system returns all the answers, the user may feel very hard to immediately get what she/he is really interested in. In some previous keyword search applications, the system returns top-k answers when k is a small parameters indicating the number of answers to be returned. All the valid answers are ranked according to some pre-defined scoring functions. However, it is quite often that some answers the user is really interested in do not appear in the top-k answer set at all (e.g., the situation in Example 10). For query suggestion, the task is to summarize all the valid answers into several categories and recommend to the user related queries from each category.

  For instance, consider the example in Example 10, the relationships between the two keywords w_2 and w_3 can be generally classified into three categories: (1) there exists a directed connection from w_2 to w_3; (2) there exists a directed connection from w_3 to w_2; (3) there does not exist any directed connections between w_2 and w_3. For each category, we can recommend a related query to capture the corresponding search intent.

- *The size of the answer set is too small.* It usually happens when some keywords in a search query are very rare in the graph. An extreme case refers to the *failing query* [57], where the answer set to a search query is empty. The previous studies about keyword search on graphs cannot properly deal with the issue of failing query, since they just notify the user there is no valid answer at all. Such a simple negative response will easily frustrate users' enthusiasm, and at last, results in bad user search experience. For query suggestion, it is necessary to understand users' search intents and recommend a set of related queries.

For instance, consider the graph in Figure 4.1, if a user submits a keyword query $q = \{w_2, w_7\}$, we find that in the graph there is no valid answer to q at all. The reason is that keyword w_7 only appears in one node v_3, and there does not exist a rooted tree in which v_3 is a leaf node. However, if we know that the user is interested in w_1 as well which is closely related to w_7 in the graph, the system can recommend a related query $q' = \{w_2, w_1\}$ to the user.

- *The quality of answers is too low.* It usually happens when some keywords in a search query have very loose connections to others. Typically, when a user starts a search process, at the very beginning, she/he may not have a very good understanding on what they are really looking for. As a result, the keywords in a search query may not be well formalized. In this case, even if the systems return all the valid answers, the user may not be interested in those answers at all. To address this issue, a reasonable solution is to provide query suggestions to the user to assist her/him on formalizing meaningful keyword queries.

  For instance, consider the graph in Figure 4.1, if a user first submits a query $q = \{w_1, w_6\}$, technically, there are two valid answers satisfying the search requirement, such as an answer tree which takes v_7 as the root node and v_2 and v_{12} as leaf nodes, and another answer tree which takes v_5 as the root node and v_{12} and v_8 as leaf nodes. However, the connections between w_1 and w_6 in the answer trees are very loose (i.e., the number of edges in the answer trees are very large), which infers that the quality of answers is very low. As a result, the user may not be interested in those answers.

The situation of low quality of answers does not conflict with the previous two situations. When we consider the first two scenarios, we assume that the size of the answer set is the major reason to cause unsatisfied search experience. As such, the quality of answers is reasonable in the first two scenarios. However, when we consider the last scenario of low quality of answers, the size of the answer set can be either too large, too small, or even resealable.

In addition to the above three scenarios, query suggestion is also useful to improve users' search experience. For instance, query suggestion can recommend a set of related queries in

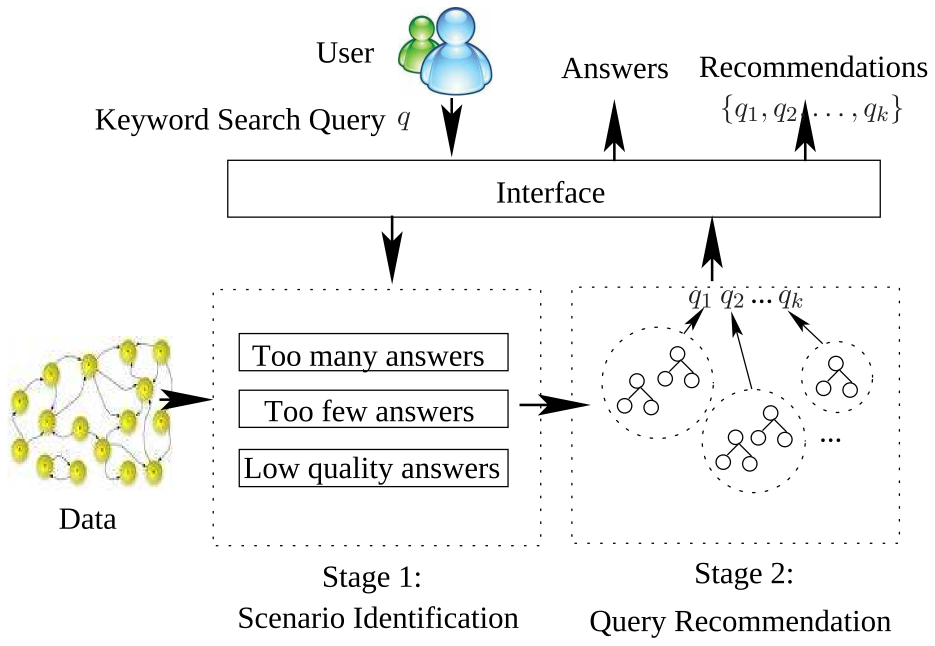


Figure 4.2: The framework of query suggestion for keyword search on graphs.

other disciplines to users so as to broaden users' interests (e.g., recommend a set of recent hot queries). However, those studies are far beyond the scope of the study in this chapter. For simplicity, we focus our analysis on the three scenarios we discussed above, and try to develop query suggestion methods for those three scenarios.

4.2.2 A General Solution Framework

As discussed in Section 4.2.1, query suggestion is necessary in three different scenarios. In each scenario, the underlying reason for query suggestion is different.

Figure 5.1 shows the framework of query suggestion in a keyword search system on graphs. We take a 2-stage solution. In the first stage, when a user submits a keyword query to the system, the system does not know which specific scenario the query suggestion should fall into. Thus, the goal of the first stage is to identify the corresponding scenario for query suggestion. Once the scenario is determined, the second stage is to recommend a set of related queries to the user.

Scenario Identification

In the first stage, we have to address two critical issues. The first issue is how to efficiently estimate the size of the answer set. The second issue is how to efficiently quantify the quality of those answers.

As described in Section 4.1, a valid answer to a keyword query on graphs is a rooted directed tree. For a keyword query $q = \{w_1, w_2, \ldots, w_m\}$, any keyword $w_i \in q$ $(1 \leqslant i \leqslant m)$ must appear in at least one leaf node in the answer tree. Based on the keyword-to-node mapping in the graph G, we can find a set of nodes $v_1, v_2, \ldots, v_m$ such that $v_i \in S(w_i)$, where $S(w_i)$ is the set of nodes containing keyword w_i. If there exists a node in the graph having a directed path leading to each v_i $(1 \leqslant i \leqslant m)$, there must exist one (and also only one) rooted directed tree which takes nodes $v_1, v_2, \ldots, v_m$ as leaf nodes. If such a node does not exist in the graph, we can conclude that no valid answer trees can be found which use nodes $v_1, v_2, \ldots, v_m$ as leaf nodes.

Based on the above finding, we can get an upper bound on the size of the answer set. For a keyword query $q = \{w_1, w_2, \ldots, w_m\}$, based on the keyword-to-node mapping, we can find $S_q = \{S(w_1), S(w_2), \ldots, S(w_m)\}$, where $S(w_i)$ is the set of nodes containing keyword w_i. The number of valid answers to q is at most $max_{size} \prod_{i=1}^{m} |S(w_i)|$, where $|S(w_i)|$ denotes the number of nodes in $S(w_i)$.

The above upper bound may be too loose, we can get an even better estimation on the size of the answer set if we maintain d-neighborhoods for a keyword w_i in the graph, denoted as $Neighborhood(w_i)^d$, where d is a small integer. Consider a keyword query $q = \{w_1, w_2, \ldots, w_m\}$, $S(w_i)$ is the set of nodes containing w_i $(1 \leqslant i \leqslant m)$. A node $v \in Neighborhood(w_i)^d$ if there exists one node $u \in S(w_i)$ such that the shortest distance $dist(u, v) \leqslant d$. If node v appears in the d-neighborhoods of all m keywords in q, v is a possible root node for an answer tree. Thus, we have an estimated size of answer set, that is, $est_{size} = |\{v | \forall i \in [1, m], v \in Neighborhood(w_i)^d\}|$.

The quality of an answer can be quantified using the score associated with each answer. In this chapter, we consider the score of an answer as the sum of lengths of each path connecting a root node and a leaf node. For a node $v \in Neighborhood(w_i)^d$, we denote $d(v, w_i)$ as the minimal distance from v to any nodes in $S(w_i)$, that is, $d(v, w_i) = min\{dist(v, v_i) | v_i \in S(w_i)\}$. Consequently, we have an estimated average quality of answers, that is, $est_{quality} = \frac{\sum_v \sum_i d(v, w_i)}{est_{size}}$, where $1 \leqslant i \leqslant m$ and $v \in \bigcap_{i=1}^{m} Neighborhood(w_i)^d$.

Algorithm 4 The algorithm for scenario identification.

```
Input: a directed graph $G = (V, E)$, a search query $q = \{w_1, w_2, \ldots, w_m\}$, three user
    specified parameters $\theta$, $\delta$, and $\rho$ ($\theta < \delta$) which determine the thresholds for too few
    answers, too many answers, as well as average quality of answers, respectively.
Output: the recommendation scenario;
 1: calculate $est_{quality}$ and $est_{size}$, respectively.
 2: if $est_{quality} < \rho$ then
 3:     return Scenario 3;
 4: else if $est_{size} < \theta$ then
 5:     return Scenario 2;
 6: else if $est_{size} > \delta$ then
 7:     return Scenario 1;
 8: else
 9:     return -1 // no query suggestion is necessary
10: end if
```

Based on the estimations about the size of the answer set and the average quality of the answers, we can identify the corresponding scenario for query suggestion. The algorithm is outlined in Algorithm 4. Suppose we have three user specified parameters θ, δ, and ρ ($\theta < \delta$) which determine the thresholds for too few answers, too many answers, as well as average quality of answers, respectively. When a user submits a query $q = \{w_1, w_2, \ldots, w_m\}$, if $est_{quality} \geqslant \rho$ and $est_{size} > \delta$, we classify it into the first scenario; if $est_{quality} \geqslant \rho$ and $est_{size} < \theta$, we classify it into the second scenario; if $est_{quality} < \rho$, we classify it into the third scenario.

Query Suggestion

As discussed in Section 4.2.1, the scenarios for query suggestion are quite different. Thus, in the second stage, the task is to analyze the scenario and develop recommendation methods for the corresponding scenario.

When the size of the answer set is too large, users may easily get lost for a large answer set. Generally, different answers may carry different semantic relationships among search keywords. It is highly possible that the answers in the answer set are diverse. Beyond returning top-k answers which may not be diverse enough to capture users' possible search intents, a reasonable solution of query suggestion is to return k related queries which captures different search intents.

As discussed in Definition 9, answers for the same search intent are quite similar to each

Algorithm 5 The algorithm for scenario transformation.

```
Input: a directed graph $G = (V, E)$, a search query $q = \{w_1, w_2, \dots, w_m\}$, the recommendation scenario, three user specified parameters $\theta$, $\delta$, and $\rho$ ($\theta < \delta$) which determine the thresholds for too few answers, too many answers, as well as average quality of answers, respectively.
Output: a transformed query $q'$;
1: if Scenario 1 then
2:   return $q$;
3: else if Scenario 2 then
4:   sort $w_1, w_2, \dots, w_m$ in descending order according to $|S(w_i)|$;
5:   while $est_{size} \leqslant \delta$ do
6:     let $w_j$ be the keyword with smallest $S(w_j)$;
7:     find another keyword $w'_j$ such that $w'_j \in Neighborhood(w_j)^1$ and $w'_j$ has the largest occurrence;
8:     let $q' = q - \{w_j\} + \{w'_j\}$;
9:     calculate $est_{size}$;
10:  end while
11:  return $q'$;
12: else
13:  while $est_{quality} < \rho$ do
14:    sort $w_1, w_2, \dots, w_m$ in ascending order according to the average shortest distance to other keywords;
15:    let $w_j$ be the keyword with largest average shortest distance;
16:    let $q' = q - \{w_j\}$;
17:    calculate $est_{quality}$;
18:  end while
19:  return $q'$;
20: end if
```

other, and answers for different search intents are quite different from each other. Thus, the key idea of our method is to cluster all the valid answers into k clusters. Each cluster represents one possible users' search intent. We generate k related queries from k different clusters. The objective is to recommend sufficient diverse suggestions to users such that users' search intents can be captured as much as possible. We present in more detail our recommendation solutions in Section 4.3.

Interestingly, the other two scenarios, that is, the scenario of too few answers and the scenario of low quality of answers, both can be transferred to the first scenario. The algorithm of the transformation is outlined in Algorithm 5.

When the size of the answer set is too small, users may not be able to find useful information they are looking for. The major reason in this scenario is that some keywords in the search query are very rare. In the graph, there are no sufficient nodes containing those keywords. Interestingly, the case of too few answers can be transferred to the case of too many answers easily. For a search query $q = \{w_1, w_2, \ldots, w_m\}$, a keyword w_i appears in $|S(w_i)|$ nodes. We can sort those keywords in q based on the number of nodes containing the keywords. Without loss of generality, we assume that $|S(w_1)| \geqslant |S(w_2)| \geqslant \ldots \geqslant |S(w_m)|$. We first replace keyword w_m in q with one related keyword which is not rare in the graph. Our idea is to pick the keyword $w_{m'}$ contained in a node $v \in Neighborhood(w_m)^1$ with largest occurrence. We next estimate the size of the answer set to a new query $q' = \{w_1, w_2, \ldots, w_{m-1}, w_{m'}\}$. If there are enough answers (e.g., $est_{size} > \delta$), we have transferred the case of too few answers to the case of too many answers. Thus, we can use the method for the case of too many answers to recommend related queries. If there are not enough answers, we can keep replacing keywords in the search query with smallest occurrence until we can get enough answers.

When the quality of the answers is too low, users may not be satisfied on the found answers, or even worse that users may not know what they really want to find. This will result in bad user search experience. Typically, the case of low quality of answers happens mainly due to the reason that keywords in the search query do not have tight connections in the graph. Specifically, the shortest distance between two keywords in the graph is very large. Not surprisingly, the case of low quality of answers can be transferred to the case of too many answers and too few answers in fact. For each pair of keywords w_i and w_j, we calculate the average shortest distance between the two keywords in the graph. For a search query $q = \{w_1, w_2, \ldots, w_m\}$, each time we pick one keyword which is farthest away from other keywords in q and remove it from q, until the quality of the answers to the new keyword query is good enough (e.g., $est_{quality} \geqslant \rho$). It may result in either the case of too many answers or the case of too few answers. We can use the specific methods for the two cases to recommend related queries accordingly.

4.3 Query Suggestion Methods

As discussed in Section 4.2.2, both the scenarios of too few answers and low quality of answers can be transferred to the scenario of too many answers. Thus, in this section, we focus our discussion about query suggestion methods only on the scenario of too many answers.

In this section, we first present a general clustering-based solution to recommend related queries for the scenario of too many answers. Then we develop several effective speed up strategies to improve the performance of query suggestion on graphs.

4.3.1 A General Solution

When a user submits a keyword query to the system, the user is interested in exploring the semantic relationships among those search keywords. In the scenario of too many answers, the answers in the answer set are diverse. At the first glance, we may easily have two options to address the diversity issue of answers. The first solution is to return all the possible answers in the answer set. However, the user may easily get lost. The second solution is to return top-k answers based on some scoring functions, where k is a user defined parameter. Although this solution can reduce the returned answers, it still cannot address the diversity issue properly. The returned top-k answers are likely to be not diverse enough to capture users' possible search intents.

Keywords in a search query are ambiguous. Thus, the search query submitted by the user is not representative enough. Rather than guessing users' real search intents which seems to be subjective, the system may focus on the answers in the answer set. Though the answers are diverse, we are still able to find clusters of answers which are similar in the sense of carrying similar semantic relationships among those query keywords. If we can find all such kinds of answer clusters, we can recommend queries to the user based on those answers in each cluster.

The general solution is outlined in Algorithm 6. The algorithm first extracts all the valid answers $\mathcal{A}$ to the given search query q. Generally, any previous methods for keyword search on graphs can be adopted to extract answers. Conducting the answer generation algorithm on the original graph may not be efficient, since we are looking for the complete set of valid answers. In Section 4.3.2, we present a simple but effective index structure to speed up the answer generation process.

Once the answer set $\mathcal{A}$ to q is constructed, Algorithm 6 conducts a clustering procedure on all the answers in $\mathcal{A}$ to generate k clusters. Here k is a user specified parameter, representing the possible number of different search intents underlying the search query q. Since each valid answer is a minimal rooted directed tree, we can calculate the similarity between two answer trees based on well-defined similarity measures. We discuss similarity

Algorithm 6 A general clustering-based solution to query suggestion for keyword search on graphs.

```
Input: A directed graph $G = (V, E)$, a search query $q = \{w_1, w_2, \ldots, w_m\}$, a user specified
    parameter $k$;
Output: $k$ related queries $Q = \{q_1, q_2, \ldots, q_k\}$;
1: set $Q = \emptyset$;
2: find all valid answers $\mathcal{A}$ to $q$;
3: conduct $k$-medoids clustering algorithm on $\mathcal{A}$ to generate $k$ clusters $C_1, C_2, \ldots, C_k$;
4: for each cluster $C_i$ do
5:     generate a set of keywords $q_i$ to capture those answers in $C_i$;
6:     let $Q = Q \cup \{q_i\}$;
7: end for
8: return $Q$;
```

calculation in more detail in Section 4.3.3.

The answers in $\mathcal{A}$ are diverse. After the clustering procedure, we expect that answers in the same cluster carry similar semantic relationships among those query keywords. Thus, each cluster can be represented as one possible search intent the user is looking for. To recommend related queries, the system can construct recommendation based on answers in the same cluster. It becomes a covering problem, that is, finding a set of keywords such that answers in the same cluster are covered, and answers in the other clusters are not covered. We develop a recommendation solution in Section 4.3.4.

Algorithm 6 requires the system to explicitly enumerate all valid answers to a keyword query. In practice, the size of the answer set is huge (e.g., est_{size} is a large number). As a result, the answer generation step may be very costly. We develop an early clustering method based on a simple heuristic. The heuristic-based recommendation method is not necessary to conduct the clustering until all valid answers are found. We present details of the heuristic-based recommendation method in Section 4.3.5.

Backward Expanding Search

To generate all the valid answers to a keyword query q, any previous studies about keyword search on graphs can be adopted. However, some methods are specifically developed for returning top-k answers, they are not efficient and effective to generate the complete set of answers. By taking into considerations both time complexity and space complexity, in this chapter, we adopt and extend the previous well-known **Backward Expanding Search**

(BES) method [10] to enumerate the complete set of valid answers. However, our query suggestion framework does not have any restrictions on the methods for generating all valid answers to a keyword query.

For a keyword query $q = \{w_1, w_2, \ldots, w_m\}$, generating the complete set of answers using the BES algorithm consists of two main steps. In the first step, for each keyword w_i ($1 \leqslant i \leqslant m$), BES finds the set of nodes $S(w_i)$ containing w_i in the graph. Thus, $S_q = \bigcup_{i=1}^{m} S(w_i)$ is the set of relevant nodes for the search query q. In the second step, BES maintains $|S_q|$ iterators to concurrently run $|S_q|$ copies of the Dijkstra's single source shortest path algorithm on the graph [18]. Each iterator starts with one of the nodes in S_q as source. The iterator traverses the graph edges in reverse direction. The idea of the BES algorithm is to find a common vertex from which a directed path exists to at least one node in each $S(w_i)$. Those directed paths correspond to a minimal rooted directed tree which takes the common vertex as the root node and the corresponding source nodes as leaf nodes. The constructed tree is a valid answer.

During the searching process, each iterator expands and generates more nodes, thus, more valid answers are found. The BES algorithm correctly enumerate the complete set of answers if we continue the searching until no iterator can generate new nodes. We refer to the complete set of answers to the search query q as $\mathcal{A}$.

4.3.2 Hierarchical Decomposition Tree

Directly applying the BES algorithm on the original graph may not be efficient in practice. The first reason is that BES generates valid answers roughly in score descending order, thus, BES is efficient for finding top-k answers. However, we want to find the complete set of answers. The orders of answers to be generated are not important in our problem. The second reason is that the graph may be very large, sometimes it is even not possible to hold the whole graph in the main memory.

To address the efficiency issue in generating the complete set of answers, we adopt the idea of hierarchical decomposition tree proposed by Bartal [8]. Generally, a hierarchical decomposition of a graph $G = (V, E)$ is a recursive partitioning of V into smaller subsets until at the lowest level of the hierarchy, individual subsets only contain single vertices. The decomposition process is associated with a tree structure (e.g., the hierarchical decomposition tree). Each leaf node in the hierarchical decomposition tree $T = (V_T, E_T)$ corresponds to a node in the original graph $G = (V, E)$. This indicates that there is a one-to-one node mapping $\psi : V_T \rightarrow V$. Each internal node, however, do not have mappings to nodes in G. Bartal [8] shows that the hierarchical decomposition tree is useful to develop efficient approximation algorithms for group Steiner tree problem in general graphs.

The hierarchical decomposition tree for general graphs has a nice property [8].

Property 1 (Hierarchical Decomposition Tree [8]) *For a weighted connected graph $G = (V, E)$, G can be f-probabilistically approximated by a weighted tree $T = (V_T, E_T)$, that is, for all nodes $u, v \in V$, $dist_G(u, v) \leqslant dist_T(\psi(u), \psi(v)) \leqslant f \times dist_G(u, v)$, where $f = O(log^2|V|)$, ψ is a one-to-one mapping between V and V_T, and $dist_G(u, v)$ and $dist_T(\psi(u), \psi(v))$ are shortest distances between two nodes in G and T, respectively.* ■

The tree metric is much simpler than the general graph metric. According to Property 1, to find the group Steiner tree on a general graph G, we can first find the group Steiner tree on the corresponding hierarchical decomposition tree T, then transfer the solution from T to the original graph G with a theoretical performance guarantee of a factor f.

We extend the idea of hierarchical decomposition tree to support keyword search on graphs. To construct such a decomposition tree, we adopt the algorithm described in [8]. It is a recursive node partitioning process. At the very beginning, the algorithm performs a bread-first-search transversal on the graph and partitions the set of nodes into subsets such that the diameter of each subset is at most $\frac{diam(G)}{f}$, where $diam(G)$ is the diameter of the original graph G. In each iteration, the algorithm recursively partitions the node sets from the previous iteration to subsets while the diameter of each subset is decreased by a factor of f. The partitioning continues until each node sets contain only one node.

To make the hierarchical decomposition tree be suitable for keyword search on graphs, for each internal node $v \in V_T$ in the tree T, we maintain the list of cutting edges $list_v$ where node partitioning is performed. The idea is that if an internal node $v \in V_T$ is the least common vertex which connects two leaf nodes $u_1, u_2 \in T$, at least one node associated with those cutting edges in v must connect two corresponding nodes $\psi(u_1), \psi(u_2)$ in the original graph G. Property 1 guarantees that the shortest distance between u_1 and u_2 in T is an f-factor of the shortest distance between two corresponding nodes $\psi(u_1), \psi(u_2)$ in G, thus, the node associated with those cutting edges in v refers to the root node of an f-approximate answer which in the leaf nodes containing keywords in $\psi(u_1), \psi(u_2)$.

We incorporate the BES algorithm to generate the complete set of answers using the hierarchical decomposition tree. The algorithm is outlined in Algorithm 7.

Algorithm 7 Generating approximate answers using the hierarchical decomposition tree.

```
Input: A directed graph $G = (V, E)$, a corresponding $f$-approximate hierarchical decomposition tree $T$, a search query $q = \{w_1, w_2, \ldots, w_m\}$;
Output: The complete set of answers $\mathcal{A}$;
1: set $\mathcal{A} = \emptyset$;
2: let $S_q = \bigcup_{i=1}^{m} S(w_i)$;
3: for each node $v_i \in S_q$ do
4:     create an iterators which uses $v_i$ as the source;
5:     let $u$ be the parent node of $v_i$ in $T$;
6:     if $u$ is not visited before by any iterator then
7:         create $Set_j(u)$ for $1 \leqslant j \leqslant |S_q|$, and set $Set_j(u) = \emptyset$;
8:     end if
9:     calculate $CrossProduct = \{v_i\} \times \prod_{j \neq i} Set_i(u)$;
10:    let $Set_i(u) = Set_i(u) \cup \{v_i\}$;
11:    for each tuple $\in CrossProduct$ do
12:        find the cutting edges in $u$ from $T$;
13:        examine associated nodes and pick the one with smallest distances to the corresponding leaf nodes;
14:        update $\mathcal{A}$;
15:    end for
16: end for
```

Let $S_q = \bigcup_{i=1}^{m} S(w_i)$. We concurrently run $|S_q|$ copies of parent finding in the hierarchical decomposition tree. Each iterator expands the current searching by retrieving the parent node in the tree T. Within each vertex u visited by any iterator, we maintain a node list $Set_j(u)$ for each keyword $w_j \in q$. Consider an iterator starting from a node $v_i \in S_q$ currently is visiting u. At that time, some other iterators might have already visited node u before and the nodes in S_q corresponding to those iterators are already in $Set_j(u)$. Thus, we are able to find the least common node u. We retrieve the nodes in the cutting edges in u in the tree T. Then we need to generate the new answers containing node v_i. We simply calculate the cross product and each cross product tuple corresponds to an answer. The corresponding root node appears in the nodes associated with the cutting edges in u. The searching continues until no new least common nodes can be found in the tree T. At last, we can find the complete set of f-approximate answers to a keyword query on G.

4.3.3 Similarity Calculation

Once the search algorithm has found all the valid answers, we need to conduct the clustering algorithm on the answers so that answers carrying similar semantic relationships among keywords can go to the same cluster. To cluster the answers, it is necessary to measure the similarity between two answers.

In this chapter, an answer is a minimal rooted directed tree. Thus, the similarity between two answers can be measured based on the structures of the answers. Without loss of generality, several pieces of information in the answers can be used to measure the similarity, such as the set of keywords, the graph structures, and the neighborhoods structures.

Consider an answer Ans_i, it contains a set of leaf nodes and internal nodes. The leaf nodes containing those keywords in a search query, and the internal nodes containing those keywords which capture semantic relationships among keywords in a search query. Obviously, the simplest way to measure the similarity is to use the keywords in an answer. Typically, the similarity based on keywords can be measured using the well-known Jaccard distance [56].

Definition 11 (Similarity based on Keywords) *Consider two answers Ans_i and Ans_j, each contains a set of keywords $W(Ans_i)$ and $W(Asn_j)$, respectively. The similarity between Ans_i and Ans_j based on keywords can be measured as $Sim(Ans_i, Ans_j)_{keyword} = \frac{|W(Ans_i) \cap W(Ans_j)|}{|W(Ans_i) \cup W(Ans_j)|}$.* ■

The keyword-based similarity is easy to calculate. However, it may not be differentiate enough for the purpose of clustering, since keywords in different answers are still possible similar. In such a case, using the keyword-based similarity is not enough to cluster all the answers.

Similarity based on graph structures may help to address the above issue. We can measure the similarity by taking into account the directed path structures in the answers.

Definition 12 (Similarity based on Structures) *Consider two answers Ans_i and Ans_j, each is a rooted tree.The similarity between Ans_i and Ans_j based on structures can be measured as $Sim(Ans_i, Ans_j)_{structure} = dist(Ans_i, Ans_j)$, where $dist(Ans_i, Ans_j)$ represents the graph distance between two tree structures Ans_i and Ans_j.* ■

In practice, calculating the exact graph distance between two trees may be costly. In fact, when a user submits a keyword query to a graph, the user is interested in the semantic relationships among those keywords, which is captured using paths connecting those keywords through the root node. Thus, we only need to consider the similarity of those connection paths among query keywords in the answers.

In this chapter, we adopt the set-of-paths model. Thus, for two answers, we can have a one-to-one path mapping based on leaf nodes which contain the same keyword in the search query. We can estimate the structure-based similarity by considering path similarity between two answers. Consider a query $q = \{w_1, w_2, \ldots, w_m\}$, there are m distinct paths from root node to leaf node in an answer. Assume we want to calculate the similarity between two answers Ans_i and Ans_j. For each keyword w_i, we consider the corresponding two mapped paths in Ans_i and Ans_j. The similarity of two paths can be simply measured by counting the number of keywords which uniquely appear in one path only. The smaller the number of unique keywords, the more similar the two paths. The sum of similarity for m mapped paths is used to estimate the graph distance between Ans_i and Ans_j.

In some cases, even the structure-based similarity may not be differentiate enough. For instance, suppose each answer is just a single node in the graph (it is possible if the node contains all the keywords in a keyword query). Neither the keyword-based similarity nor the structure-based similarity may work well to cluster those answers. In such a case, neighborhood information may be brought into consideration for similarity calculation.

Definition 13 (Similarity based on neighborhoods) *Consider two answers Ans_i and Ans_j. We can extract the corresponding neighborhoods for the answers. A d-neighborhood structure of answer Ans_i, denoted as $Neighbor_d(Ans_i)$, is an induced subgraph which contains the nodes in Ans_i and any other nodes in G which has a shortest distance to a node in Ans_i less than or equal to d. The similarity between Ans_i and Ans_j based on neighborhoods can be measured as $Sim(Ans_i, Ans_j)_{neighbor} = dist(Neighbor_d(Ans_i), Neighbor_d(Ans_j))$.* ■

While the keyword-based similarity is easy to calculate, it may not be differentiate enough for clustering. Moreover, although the neighborhood-based similarity is costly to calculate, it is differentiate enough for clustering. When we conduct the clustering on the complete set of answers, we have to balance the efficiency for similarity calculation and effectiveness of similarity measures for clustering.

We adopt a flexible solution based on the idea of clusterability [27]. The term "clusterability" is used to determine whether the answers are differentiate enough for clustering based on specific similarity measure. We first use the keyword-based similarity for clustering. If we find that the answers are not clusterable, that is, keyword-based similarity is not differentiate enough to separate answers, we adopt the next level of similarity, that is, structure-based similarity. We continue to the next level of neighborhood-based similarity, if the answers are still not clusterable. If we find that the answers are already clusterable enough, we just adopt the current choice of the similarity measure, and conduct the clustering on the answers.

It is crucial to measure the clusterability given a set of answers. We adopt a simple heuristic. For a uniformly distributed dataset, we randomly pick k data as centers and calculate the inner-distance of k clusters. If we conduct such a trial for enough times, we may find that the differences among different trials is not large. However, the differences is very likely to be large on a perfectible clusterable dataset (e.g., centers are fixed). Using this simple idea, we conduct the trials on the answers using different levels of similarity measures. After a set of trials (in practice, we choose a reasonable number, e.g., 5), we monitor the average inner-distance of each cluster. If the variance is too small, we argue that the current similarity measure is not good enough, thus, we choose the next level of similarity measure. It stops when either the similarity measure is good enough or the last level of similarity measure has been chosen.

In some scenarios, the locations of answers in the graph are also useful to differentiate the answers. Two answers may be more similar to each other if the two answers appear closely in the graph. As an interesting future research direction, we will explore the distance measure of answers in the graph for query suggestion.

4.3.4 Recommendation Generation

Once the similarity measure is chosen, we can conduct the conventional k-medoids clustering algorithm to cluster the answers into k clusters. Once we obtain those clusters, in the next step, we need to recommend keywords from each cluster.

The recommendation should try to capture semantic relationship among those search keywords as much as possible. In general, we say a query q_i captures a search intent C_i if and only if: (1) each answer $Ans_j \in C_i$ is a valid answer to q_i; (2) if an answer $Ans_j \notin C_i$, Ans_j is not a valid answer to q_i.

An answer is a tree, in which each leaf node contains a keyword in the query. In order to satisfy the first condition, for each leaf node v_i which contains a keyword w_i in the query, the leaf node must be retained in the answer for a recommended query. In other words, in the recommended query, at least one keyword must come from v_i. As a result, the keywords in the recommended query comes from m leaf nodes in an answer.

In order to satisfy the second condition, we have to make sure that some keywords in a recommended query for a search intent C_i must not appear in the cluster of answers for other search intents.

One simple recommendation generation solution is as follows. For a specific search intent C_i which contains a set of answers, we consider the keywords contained in those leaf nodes. Let $S(w_i)_{C_i}$ represents the set of leaf nodes in C_i that contain the keyword w_i. We find the keywords commonly appear in all those nodes in $S(w_i)_{C_i}$, and denoted as $W(w_i)_{C_i}$. If there exists a keyword $w \in W(w_i)_{C_i}$ and $w \notin W(w_i)_{C_j}$ for any $j \notin i$, we can recommend a query $q' = q - \{w_i\} + \{w\}$.

It is easy to see that q' satisfies the two requirements of query suggestions. However, the simple recommendation generation solution may not work in practice. First, such a keyword w ($w \in W(w_i)_{C_i}$ and $w \notin W(w_i)_{C_j}$ for any $j \notin i$) may not exist at all. Second, if we replace w_i with w in the original query, some more answers to the new query q' may exist in the graph. We may need to conduct the query suggestion process repeatedly.

To address the issue, we propose an alternative solution for generating recommended queries. We relax the two requirements for query suggestion a little bit. Rather than making sure that each answer $Ans_j \in C_i$ is a valid answer to q_i, we only require that each answer $Ans_j \in C_i$ is a valid subtree in an answer Ans'_j to q_i.

Generally, the root node in an answer plays an important role to capture the semantic relationships among keywords. For the answers in the same cluster, keywords in a search query must appear in at least one of the leaf nodes in the answer. Thus the keywords in the root node are used to build linkages among those keywords in the query. The recommended queries should satisfy the requirement that the answers to the recommended queries should capture similar semantic relationship among search keywords, i.e, having the same root nodes as those answers in the original answer set. Not surprisingly, if we recommend keywords from the root nodes in the answer trees from the same cluster, we are able to provide to users the detailed semantic relationship they are interested in.

The above query suggestion method is based on Property 2.

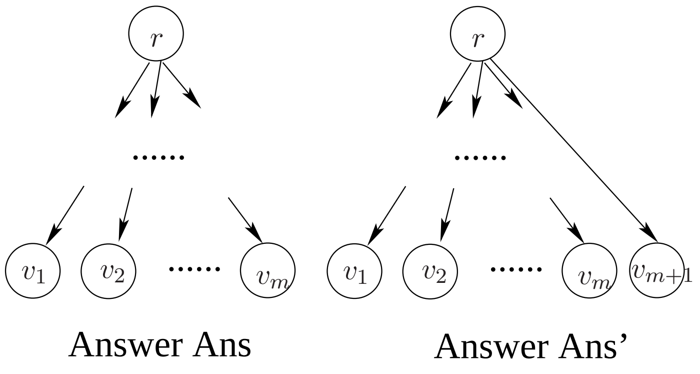


Figure 4.3: The rationale of query suggestion.

Property 2 (Query Suggestion) *Consider a keyword query $q = \{w_1, w_2, \ldots, w_m\}$ in Figure 4.3, Ans is an answer with r as the root node. Let w_{m+1} be a keyword contained in r in Ans. We construct another answer Ans' by adding a leaf node v_{m+1} which contains keyword w_{m+1} (v_{m+1} is the same to r). Ans' must be a valid answer to the query $q' = \{w_1, w_2, \ldots, w_m, w_{m+1}\}$.* ■

Based on the above finding, our recommendation is constructed from keywords appearing in those root nodes in the answers. The query suggestion method is outlined in Algorithm 8.

Given a cluster of answers $C_i = \{Ans_{i_1}, Ans_{i_2}, \ldots, Ans_{i_s}\}$, we extract the common keywords appearing in the root nodes in the answers in C_i. The algorithm next tries to find a keyword w which only appears in $Cand_i$ but not appear in any other $Cand_j$ where $j \neq i$. If such a keyword w exist, we can generate a recommendation query q_i by appending w to q; however, if such a keyword w does not exist, we have to compromise the requirement of query suggestions. The algorithm tries to find the keyword w which appears frequently in $Cand_i$ and appears rarely in other $Cand_j$. Similarity, the keyword w is appended to q to generate a recommended query. The algorithm constructs the recommendations for each cluster. Finally a set Q of k recommendations are returned.

4.3.5 Early Clustering and Query Suggestion

Before conducting the clustering on the answers, Algorithm 6 requires that all the valid answers should be explicitly extracted. However, in practice, it may not be efficient, since enumerating all the valid answers and clustering them may be quite time-consuming. Can we avoid explicitly generating all the valid answers but still be able to recommend related

Algorithm 8 Generating query suggestions.

```
Input: a directed graph $G = (V, E)$, a search query $q = \{w_1, w_2, \dots, w_m\}$, $k$ clusters of
    answers $\mathcal{A} = \{C_1, C_2, \dots, C_k\}$;
Output: $k$ recommended queries $Q$;
 1: initialize $i = 1$;
 2: initialize $q_i = \emptyset$;
 3: for $i \leqslant k$ do
 4:     let $Cand_i$ contain those common keywords in the root nodes in the answers in $C_i$;
 5:     $i = i + 1$;
 6: end for
 7: let $i = 1$;
 8: for $i \leqslant k$ do
 9:     find a keyword $w \in Cand_i$ and $w \notin Cand_j$ for any other $j \neq i$;
10:     if such a keyword $w$ does not exist then
11:         find a keyword $w$ such that $\frac{freq(w,Cand_i)}{\sum_{j \neq i} freq(w,Cand_j)}$ is smallest;
12:     end if
13:     let $q_i = q_i \cup \{w\}$;
14:     $i = i + 1$;
15: end for
16: return $Q$;
```

keywords with high quality?

Consider the searching algorithm on the hierarchical decomposition tree T in Algorithm 7. We want to find an internal node $v \in V_T$ such that v is the least common node in T which connects to nodes containing all the search keywords as leaf nodes. The distance from v to the leaf nodes is at most f-factor of the shortest distance in the original graph. The length of the paths in the answers actually play an important role in the similarity calculation. The larger the path lengths, the more likely that the answers carry different semantic relationship among keywords. Thus rather than conducting clustering on the explicitly extracted answers, we can use internal nodes in the hierarchical decomposition tree for early clustering.

In the hierarchical decomposition tree, we maintain the list of keywords appearing in the cutting edges of each internal nodes. Those keywords appear in some root nodes of the answers. The algorithm starts the searching exactly the same as that in Algorithm 7. We want to find a set of internal nodes in the hierarchical decomposition tree which connect corresponding leaf nodes in the tree. However, we do not look at the original graph when internal nodes are identified. We argue that the group Steiner trees in the hierarchical decomposition tree are representative enough for answer clustering.

There are several reasons. First, the group Steiner tree in the decomposition tree can be easily extended to be an answer in the original graph, by simply mapping internal nodes into nodes in the original graph. Second, each internal node contains keywords which appear in the cutting edges during the partitioning. Correspondingly, in the original graph, those keywords are very likely to appear in the paths from the root node to the leaf nodes. Third, the lengths of paths in the tree is an approximation of the shortest distance in the original graph.

Based on the above findings, we conduct the clustering on those internal nodes in the hierarchical decomposition tree. The paths from the root node to the leaf nodes are used for measuring the similarity during the clustering procedure. The keywords in the internal nodes are used for candidates for query suggestion generation. The generation algorithm is similar to the one in Algorithm 8.

4.4 Experimental Results and Performance Study

In this section, we report a systematic empirical study conducted on a real data set to evaluate our query suggestion methods for keyword search on graphs. All the experiments were conducted on a PC computer running the Microsoft Windows XP SP3 Professional Edition operating system, with a 3.0 GHz Pentium 4 CPU, 1.0 GB main memory, and a 250 GB SATA hard drive. The programs were implemented in C++ and were compiled using Microsoft Visual Studio .Net 2008.

4.4.1 The DBLP Data Set

We downloaded the raw DBLP data from the Website [69] and constructed the graph version of the DBLP data following the steps in [39]. Since the original DBLP data is a tree in which each paper is represented as a small subtree, we made several modifications on the data by including two new types of non-tree edges. The first type of edges we added is by connecting papers through citation relationship. The second type of edges we added is by merging the same author node under different paper subtree as one unique author node. Moreover, we modified the DBLP data by removing non-interesting elements in each paper such as URL, EE, etc. Furthermore, we removed those papers not being referenced by other papers. At last, we obtain a graph version of the DBLP data which contains about $91K$

Query ID	Keywords	# Nodes Containing Keywords	# Answers
Q_1	{blink, graph}	(3, 1704)	39
Q_2	{jagadish, optimization}	(16, 311)	101
Q_3	{spamicity, cook}	(11, 79)	0
Q_4	{approximation, dynamic, optimal}	(332, 630, 411)	703
Q_5	{multimedia, type, object}	(304, 110, 91)	66
Q_6	{vapnik, support, vector}	(11, 103, 329)	1198
Q_7	{database, olap, grey, cube}	(1105, 396, 27, 91)	598
Q_8	{hector, jagadish, performance, improving}	(7, 16, 996, 104)	337
Q_9	{jiawei, xifeng, sigmod, graph}	(17, 6, 731, 1704)	206

Table 4.1: Keyword queries.

papers and $171K$ authors, in total $534K$ nodes and $709K$ edges. The number of distinct lower-cased keywords in the DBLP data is about $63K$.

4.4.2 Answers to Keyword Search on Graphs

To extensively demonstrate the performance of our query suggestion methods for keyword search on graphs, we carefully select a set of 9 different keyword queries with different sizes (i.e., sizes are varied from 2 to 4). We tried to include a wide variety of keywords by taking into considerations the selectivity of keywords, the size of the relevant answers, etc. Table 4.1 lists the keywords in each search query.

Among the 9 different keyword queries, 3 of them have 2 keywords, 3 of them have 3 keywords, and 3 of them have 4 keywords. We counted the number of nodes containing the specific keywords in the queries, and the results are shown in Table 4.1. Generally, the keywords in 9 queries have different number of occurrences in the graph.

Generally, the number of the valid answers varies when the keywords in the query are different. In some situations, such as Q_4 and Q_6, the number of the valid answers is very large; in some other situations, such as Q_1 and Q_5, the number of the valid answers is very small. We also find that in Q_3, there does not exist any valid answers at all. The results indicate that the categorizations of scenarios for query suggestions described in Section 4.2.1 is meaningful.

Moreover, among 9 different keyword queries, different recommendation scenarios exist. For example, in Q_1 and Q_5, the quality of the answers is fine, but the number of the valid answers is too few. Thus in our recommendation methods, we first need to replace one keyword in the query with another keyword which has a larger occurrence in the original

ID	Simple Method	Decomposition Tree Method	Fast Method
Q_1	Q_1 - {blink} + {search} + {sigmod}	Q_1 - {blink} + {search} + {sigmod}	Q_1 - {blink} + {search} + {sigmod}
	Q_1 - {blink} + {search} + {vision}	Q_1 - {blink} + {search} + {vision}	Q_1 - {blink} + {search} + {vision}
Q_2	Q_2 + {database}	Q_2 + {database}	Q_2 + {database}
	Q_2 + {srivastava}	Q_2 + {srivastava}	Q_2 + {srivastava}
Q_3	Q_3 - {spamicity} + {complexity}	Q_3 - {spamicity} + {complexity}	Q_3 - {spamicity} + {complexity}
	Q_3 - {spamicity} + {algorithm}	Q_3 - {spamicity} + {algorithm}	Q_3 - {spamicity} + {graph}
Q_4	Q_4 + {database}	Q_4 + {sigmod}	Q_4 + {sigmod}
	Q_4 + {algorithm}	Q_4 + {algorithm}	Q_4 + {algorithm}
Q_5	Q_5 - {object} + {visual} + {network}	Q_5 - {object} + {visual} + {system}	Q_5 - {object} + {visual} + {programming}
	Q_5 - {object} + {visual} + {acm}	Q_5 - {object} + {visual} + {acm}	Q_5 - {object} + {visual} + {stream}
Q_6	Q_6 + {learning }	Q_6 + {learning }	Q_6 + {svm}
	Q_6 + {classification}	Q_6 + {classification}	Q_6 + {classification}
Q_7	Q_7 + {mining}	Q_7 + {mining}	Q_7 + {mining}
	Q_7 + {microsoft}	Q_7 + {microsoft}	Q_7 + {microsoft}
Q_8	Q_8 + {database}	Q_8 + {database}	Q_8 + {database}
	Q_8 + {algorithm}	Q_8 + {computer}	Q_8 + {computer}
Q_9	Q_9 + {index}	Q_9 + {index}	Q_9 + {index}
	Q_9 + {kdd}	Q_9 + {kdd}	Q_9 + {kdd}

Table 4.2: Query suggestions using different methods ($k = 2$).

graph. In Q_3, the quality of the answers is very low. Thus in our recommendation methods, we first need to remove one keyword which has a largest shortest distance to other keywords. In this case, keyword `spamicity` is removed.

4.4.3 Accuracy of Query Suggestions

We compared three different recommendation methods developed in this chapter for performance analysis. The first method refers to the query suggestion method without the hierarchical decomposition tree in Section 4.3.2 and the heuristic in Section 4.3.5. We call this method as **Simple** method. The second method refers to the query suggestion method in which the hierarchical decomposition tree is used to approximate the answers. We call this method as **Decomposition Tree** method. The third method refers to the query suggestion method when the heuristic in Section 4.3.5 is used. We call this method as **Fast** method.

In Table 4.1, we list the total number of answers to each query. In Table 4.2, we list the recommended queries provided by three different query suggestion methods. Here we set the parameter k to 2, that is, 2 related queries are generated. If we treat the recommended queries returned by the simple method as the ground truth, we find that the performance of the decomposition tree method and the fast method is quite good. For example, among 18 returned queries, the queries returned by the decomposition tree method have a precision score $\frac{15}{18} = 0.83$, and the queries returned by the fast method have a precision score $\frac{12}{18} = 0.67$.

In Web search, user studies have been widely adopted to examine the accuracy of the query suggestion results. However, user studies are an expensive task, and sometimes it may be biased due to some reasons, such as lack of background knowledge for those participants. In this chapter, we only examine the accuracy of query suggestion based on comparisons with the baseline result. As some future directions, we will develop more reliable measures to evaluate the query suggestion results.

4.4.4 Efficiency of Query Suggestions

We also examine the response time of query suggestions using different methods. Figure 4.4 shows the query response time of 9 different queries in Table 4.1 with respect to three different query suggestion methods.

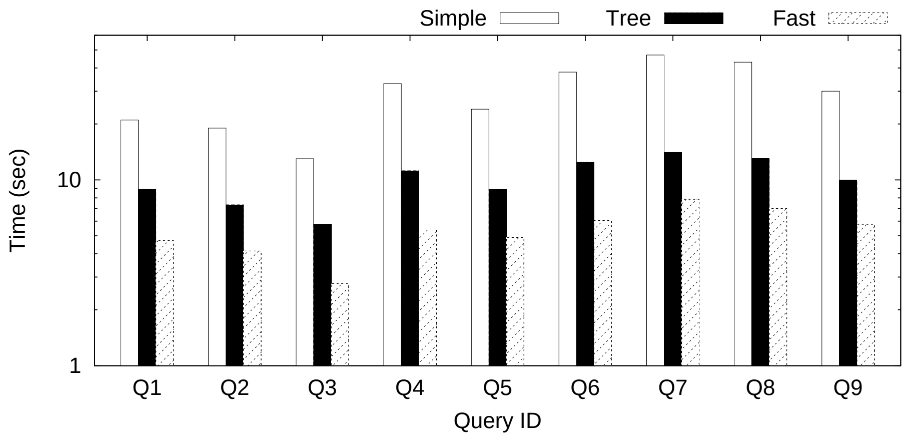


Figure 4.4: The response time of query suggestion.

Generally, the decomposition tree method and the fast method have obvious improvements on the response time. When the simple method needs around 30 seconds to recommend related queries, the other two methods only needs about 10 and 5 seconds, respectively. Moreover, we also find that the number of valid answers to a keyword query have great impact on the query suggestion response time. It is easy to understand, since more answers means the time for answer generation is large, the time for clustering answer is large, and the time for generating related queries from clusters is large.

The results in Table 4.2 and in Figure 4.3 indicate that the decomposition tree method and the fast method can balance the quality of query suggestion and the response time.

4.4.5 Parameter Settings of Query Suggestion Methods

In our query suggestion methods, we have several important parameters, such as the value of k, which represents the number of recommendations to be returned; the number of keywords m in the query. In this section, we report several experiments conducted on the DBLP dataset to examine the effects of different parameter settings.

We first examine the effects of the parameter m. We varied the values of m from 2 to 6. For each specific m, we randomly generated 100 different queries. We adopted the three different query suggestion methods, and examined the average running time for those random queries. We fix the value of k to be 3. Figure 4.5 shows the results.

In general, when the value of m increases, the running time of different query suggestion methods increases as well. This is because the number of valid answers increases when m increases. Among all the cases, the simple method has the largest running time, while the

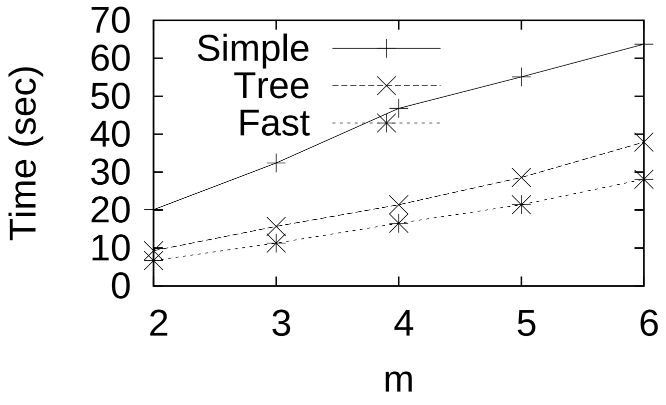


Figure 4.5: The running time with respect to different values of m

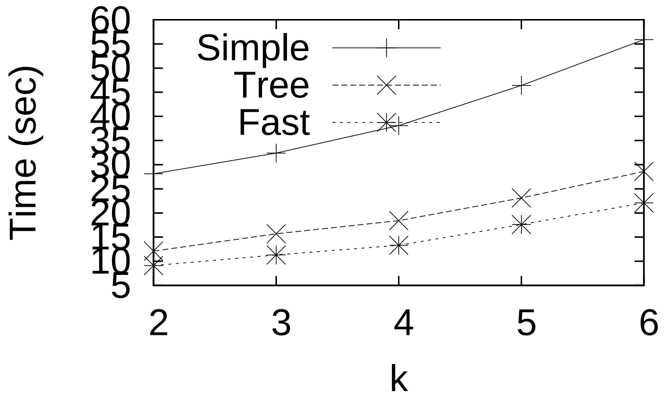


Figure 4.6: The running time with respect to different values of k.

decomposition tree based method is much more better. The fast method has the smallest running time. The results indicate that the speed-up strategies we developed in the chapter are very useful to improve the query suggestion performance.

We next examine the effects of the parameter k. We varied the values of k from 2 to 6. We randomly generated 100 different queries with size 3. We adopted the three different query suggestion methods, and examined the average running time for those random queries. Figure 4.6 shows the results.

Generally, when the value of k increases, the running time of different query suggestion methods increase as well. This is because we need more time to conduct the clustering step on the answers and recommend more related queries.

4.5 Conclusions

In this chapter, we took an initiative step towards the problem of query suggestion for keyword search on graphs. We identified the importance of query suggestion for keyword search systems on graphs. We formalized the problem of query suggestion for keyword search on graphs, and categorized the possible recommendation scenarios into three major categories. We presented a general solution framework to provide recommendations, and developed practical recommendation methods. The experimental results conducted on large real data sets verified the effectiveness and efficiency of our proposed query suggestion methods.

There are several interesting directions we plan to explore in the near future. For instance, in practice, different users may have quite different search behaviors. As a result, personalized and context-aware query suggestions is very popular in Web search engines. It is interesting to analyze and develop recommendation methods for keyword search on graphs based on the personalization and context-aware manner. For another case, in this chapter, we treat k related queries in the recommendation equally. In some applications, it is desirable to recommend a list of k ranked recommendations. Therefore, the user can quickly identify the real information needs from top-ranked recommendations. Finally, the query response time is crucial in the keyword search system. While the methods proposed in this chapter can recommend related queries in a short time, it is interesting to see if there are more speed up strategies or effective index structures can further improve the performance of query suggestion for keyword search on graphs.

Chapter 5

OLAP Infrastructure on Search Logs

With the great success of Web search engines such as Google and Microsoft's Bing, the keyword search technique has been widely used by millions of users for searching interesting information from large-scale data. It is well-known that the Web search engines use search logs to trace users' search history. Inherently, search logs are an indispensable output for the whole keyword search process.

In general, search logs contain rich and up-to-date information about users' needs and preferences. While search engines retrieve information from the Web, users implicitly vote for or against the retrieved information as well as the services using their clicks. Moreover, search logs contain crowd intelligence accumulated from millions of users, which may be leveraged in social computing, customer relationship management, and many other areas. In recent years, search logs have become a more and more important data source for a wide scope of applications to improve the applicability of keyword search.

Using search logs, we may develop a good variety of keyword search applications in a search engine, which are truly useful for the users and highly profitable for the search engine. For example, by examining the queries frequently asked by users after the query *"KDD 2011"*, a search engine can suggest queries such as *"San Diego hotel"* which may improve users' search experience (e.g., [12, 42]). As another example, by analyzing query sequences and click-through information in search logs, a search engine can help an advertiser to bid for relevant keywords (e.g, [31]). Some further examples include using search logs to improve the Web search ranking (e.g., [1, 43]), personalize Web search results (e.g., [24, 63]), correct search query spellings (e.g., [44, 51]), and monitor and evaluate the performance of search engines and other Web services (e.g., [2, 26]).

Given that there are many different keyword search applications in search engines, how many specific search log analysis tools do we have to build? Obviously, building one specific search log analysis tool per application is neither effective nor efficient. Can we develop a search log analysis infrastructure supporting the essential needs of many different keyword search applications? There are some interesting opportunities and challenges.

First, although different keyword search applications carry different purposes and technical demands, by a careful survey of a variety of real applications, we find that many analysis tasks can be supported by a small number of search log pattern mining functions, as exemplified in Section 5.1.2. This observation makes the idea of building a central search log analysis infrastructure feasible and practical. Building a search log analysis infrastructure presents a new opportunity for developing more effective and powerful search engines.

Second, although there are numerous previous studies on extracting interesting patterns from search logs, those methods cannot be directly applied to building the search log analysis infrastructure. The target infrastructure should be able to serve multiple keyword search applications and answer general requests on search log pattern mining. However, almost every previous work is designed for a specific task – the algorithms and data structures are customized for specific tasks. Therefore, those methods cannot meet the diverse requirements in a search log analysis infrastructure.

Third, search logs are often huge and keep growing rapidly over time. Moreover, many keyword search applications require online response to the pattern mining queries. Therefore, a search log analysis infrastructure has to be constructed in a distributed computation environment. Organizing search logs and supporting online mining in a distributed environment present great challenges.

In this chapter, we present an initiative towards building an *online analytic processing* (OLAP for short) system on search logs as the search log analysis infrastructure. The central idea is that we build a simple yet scalable and distributable index on search logs so that sequential patterns in search logs can be mined online to support many real keyword search applications. The role of the OLAP system is shown in Figure 5.1. As shown in Section 5.2, the system not only can support online applications in search engines, but also can facilitate online analysis of search logs by search engine developers. We make the

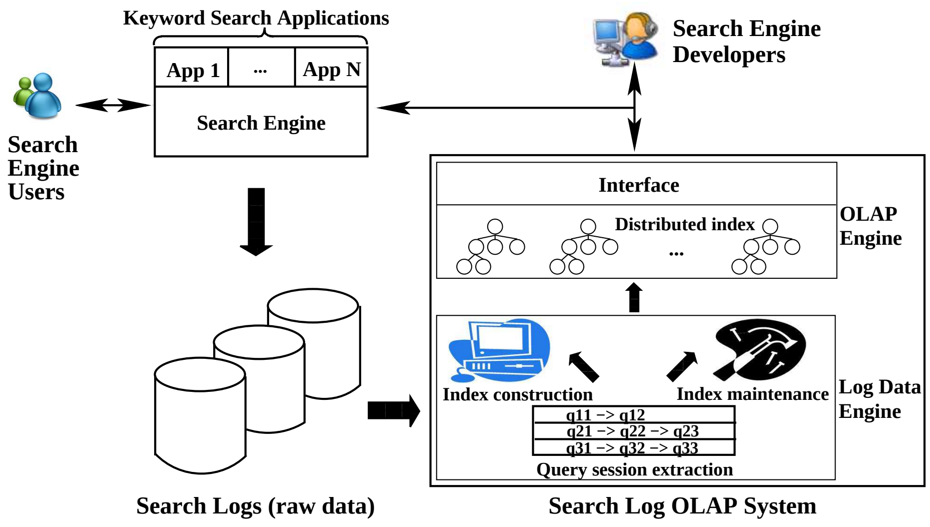


Figure 5.1: The framework of the OLAP infrastructure on search logs.

following contributions.

First, we argue that many traditional aggregate functions, such as *sum*, *min* and *max*, may not be suitable to support OLAP on search logs. Instead, we identify three novel mining functions, namely *forward search*, *backward search*, and *session retrieval*, which are particularly useful for data analysis on search logs. We further illustrate how to integrate those novel functions with OLAP operations, such as roll-up and drill-down, on search logs.

Second, we develop a simple yet scalable and distributable index structure to support OLAP on search logs using the three mining functions. We do not follow the materialization approach widely adopted in building data warehouses on relational data, since those approaches cannot be straightforwardly applied to sequential search logs. Instead, we use a suffix-tree index. Moreover, we develop a distributed suffix tree construction and maintenance method under the *MapReduce* programming model [22].

Last, we demonstrate the feasibility and potential of a search log analysis infrastructure through a systematic empirical study. We extracted from a major commercial search engine a search log containing more than 4.96 billion searches, 2.48 billion sessions, and 1.48 billion unique queries. The experimental results indicate that our OLAP system on search logs is effective to support various keyword search applications, and our design is efficient in practice.

The rest of the chapter is organized as follows. Section 5.1 identifies several OLAP functions and operations on search logs, and briefly discusses the difference between our work and S-OLAP [53]. In Section 5.2, we propose our approach to scalable OLAP on search logs. A systematic empirical study conducted on a large real data set is reported in Section 5.3. Section 5.4 concludes the chapter.

5.1 Problem Definition: OLAP on Search Logs

In this section, we first briefly review how to extract query sessions from search logs. Then, we present three basic sequence mining functions on query sessions, and show that those functions can be used to conduct OLAP on search logs. Furthermore, we outline the major differences between our work and S-OLAP [53].

5.1.1 Query Session Extraction

Conceptually, a search log is a sequence of queries and click events. Since a search log often contains the information from multiple users over a long period, we can divide a search log into sessions.

In practice, we can extract sessions in two steps, as described in [12]. First, for each user, we extract the queries by the user from the search log as a stream. Then, we segment each user's stream into sessions based on a widely adopted rule [81]: two queries are split into two sessions if the time interval between them exceeds 30 minutes.

Formally, let Q be the set of unique queries in a search log. A *query sequence* $s = \langle q_1 \cdots q_n \rangle$ is an ordered list of queries, where $q_i \in Q$ $(1 \leqslant i \leqslant n)$. n is the *length* of s, denoted by $|s| = n$. A *subsequence* of sequence $s = \langle q_1 \cdots q_n \rangle$ is a sequence $s' = \langle q_{i+1} \cdots q_{i+m} \rangle$ where $m \geqslant 1$, $i \geqslant 0$, and $i + m \leqslant n$, denoted by $s' \sqsubseteq s$. In particular, s' is a *prefix* of s if $i = 0$. s' is a *suffix* of s if $i = n - m$. The *concatenation* of two sequences $s_1 = \langle q_1 \cdots q_{n_1} \rangle$ and $s_2 = \langle q'_1 \cdots q'_{n_2} \rangle$ is $s_1 \circ s_2 = \langle q_1 \cdots q_{n_1} q'_1 \cdots q'_{n_2} \rangle$.

In many search engine applications, frequency is often used as the measure in analysis. Given a set of query sessions $D = \{s_1, s_2, \ldots, s_N\}$, the *frequency* of a query sequence s is $freq(s) = |\{s_i | s \sqsubseteq s_i\}|$ $(s_i \in D)$. The *session frequency* of s is $sfreq(s) = |\{s_i | s = s_i\}|$.

5.1.2 Session Sequence Mining Functions

To build a search log analysis infrastructure, we collect different requirements on mining search logs at a major commercial search engine. We find that the major needs can be summarized into three basic session sequence mining functions, namely *forward search*, *backward search* and *session retrieval*. Importantly, many keyword search applications are based on mining frequent sequences. Moreover, different from traditional sequential pattern mining [65] which finds the complete set of sequential patterns, a keyword search application typically relies on the top-k query sequences related to a given query sequence s.

Definition 14 (Forward search) *In a set of sessions, given a query sequence s and a search result size k, the* ***forward search*** *finds k sequences $s_1, \ldots, s_k$ such that $s \circ s_i$ $(1 \leqslant i \leqslant k)$ is among the top-k most frequent sequences that have s as the prefix.* ■

Example 11 (Forward search) *A user's search experience can be improved substantially if a search engine can predict the user's search intent and suggest some highly relevant queries. This is the central idea behind the query suggestion application. For example, a user planning to buy a car may browse different brands of cars. After the user conducts a sequence of queries $s = \langle$"Honda" "Ford"$\rangle$, a search engine may use a forward search to find the top-k query sequences $s \circ q$, and suggest the queries q to the user. Such queries may be about some other brands like "Toyota", or about comparisons and reviews like "car comparison".* ■

A forward search only considers sequences s_i that are consecutive to query sequence s in sessions. This is because non-consecutive queries may not be closely related in semantics. For example, a user may raise a sequence of queries $\langle$"*Beijing Olympics*" "*US basketball team 2008*" "*Kobe Bryant*" "*LA Lakers*"$\rangle$. Although each pair of consecutive queries are closely related, the relationship between "*Beijing Olympics*" and "*LA Lakers*" is relatively weak. Thus, we only consider consecutive query sequences in our mining functions.

Symmetrically, we have backward search.

Definition 15 (Backward search) *In a set of sessions, given a query sequence s and a search result size k, the* ***backward search*** *finds k sequences $s_1, \ldots, s_k$ such that $s_i \circ s$ $(1 \leqslant i \leqslant k)$ is among the top-k most frequent sequences that have s as the suffix.* ■

Example 12 (Backward search) *Keyword bidding is an important service in sponsored search. A search engine may provide a keyword generation application to help a customer to select keywords to bid.*

*A small electronic store may find keyword “*digital camcorder*” expensive. By a backward search, the search engine can find the query subsequences that often appear immediately before query “*digital camcorder*” in query sessions. For example, some users may raise queries “*digital video recorder*”, “*DV*”, or “*DC*” before “*digital camcorder*” in sessions, since they may not get the term “*camcorder*” at the first place. The small electronic store may bid for those keywords which are cheaper than “*digital camcorder*” but carry similar search intent.* ■

Forward search and backward search focus on finding subsequences. In some situations, the whole sessions may need to be retrieved as a pattern.

Definition 16 (Session retrieval) *In a set of sessions, given a query sequence s, the* ***session retrieval*** *finds the top-k query sessions $s_1, \dots, s_k$ in session frequency ($sfreq$) that contain s.* ■

Example 13 (Session retrieval) *Search logs can be analyzed by search engine developers to monitor the search quality and diagnose the causes of user dissatisfactory queries. For example, suppose the click-through rate of query “*Obama*” is high in the past, but drops dramatically recently. To investigate the causes, a dissatisfactory query diagnosis (DSAT) application can find the top-k sessions containing “*Obama*” using a session retrieval function. By analyzing those sessions, if a search engine developer finds that the sessions containing query “*election*” have high click-through rate, while the recent sessions containing query “*inauguration*” have low click-through rate, the reason for the decrease of the click-through rate may be that the search engine does not provide enough fresh results about Obama’s inauguration.* ■

5.1.3 OLAP on Search Log Session Data

OLAP is well defined on relational data. For example, consider a transaction table on sales of electronic goods in Canada T(*tid*, *prod*, *cust*, *agent*, *amount*) where the attributes *tid*, *prod*, *cust*, *agent*, and *amount* are transaction-id, product name, customer name, agent name, and sales amount, respectively. A user may want to analyze how the sales amount is related to various factors such as product category, customer group, agent group, as well as their combinations. OLAP queries retrieve group-by aggregates on *amount* using various combinations of dimension values, such as the sum of amount for all cameras sold by the Vancouver agents. Using OLAP, a user can roll up or drill down along different group-by levels, such as comparing the sales amount of cameras by agents in Vancouver and that by agents in Canada.

Sessions are query sequences. To conduct OLAP on session data, a user can specify a query sequence s. Analogous to the relational case, each query in s can be considered as a dimension, while the frequency of s can be considered as the measure. The three basic session sequence mining functions in Section 5.1.2 are then regarded as the aggregate functions. The *sequential drill-down operations* (drill-down for short) on s can be defined as aggregations by either pre-pending a sequence s_1 at the head of s or appending s_1 at the tail of s. In other words, the sequential drill-down operations perform on either sequence $s_1 \circ s$ or sequence $s \circ s_1$. Reversely, the *sequential roll-up operations* (roll-up for short) on s can be defined as aggregations by removing a subsequence of s either at the head or tail.

Example 14 (OLAP operations on session data) *In query suggestion application, the search engine provides suggestions each time the user raises a query. For example, when a user raises a query "Honda", the search engine can apply a forward search function on $s_1 = \langle$ "Honda"$\rangle$ and get the top-k queries as the candidates for query suggestion. Suppose the user raises a second query "Ford", the search engine can drill down using the forward search function to find out the top-k queries following sequence $s_2 = \langle$ "Honda" "Ford"$\rangle$ as the candidates for query suggestion.*

In keyword bidding application, suppose a user applies the backward search function on sequence $s_1 = \langle$ "digital camcorder"$\rangle$ and finds the sequence $s_2 = \langle$ "DV" "digital camcorder"$\rangle$ interesting. The user may further roll up using the forward search function and find out the top-k queries following $s_3 = \langle$ "DV"$\rangle$. Those top-k queries may also be good candidates to bid. ■

5.1.4 Comparisons with S-OLAP

Most of the OLAP methods focus on relational data. Recently, OLAP has been extended to other data. In particular, Lo *et al.* [53] extend OLAP to S-OLAP on sequence data, which is highly related to our study. In S-OLAP, a user defines a pattern template of interest. For example, template $\langle XYYX \rangle$ captures a round trip by travelers where X and Y are two subway stations. Aggregates such as `count()` are computed on a sequence database for each instance of the pattern template such as $\langle$"Downtown" "Waterfront" "Waterfront" "Downtown"$\rangle$. An inverted index and a join-based algorithm are proposed to support efficient computation.

S-OLAP and the OLAP on search logs in this chapter extend OLAP to sequence data from different angles. First, S-OLAP assumes that a user specifies pattern templates. However, in keyword search applications in search engines, pattern templates are often unknown. Instead, specific query sequences are used as constraints in search. Second, S-OLAP is inefficient to support online response to several sequence mining functions, such as forward search, backward search, and session retrieval, since the results have to be computed by joining the inverted lists on the fly. As shown in Section 5.1, these functions are commonly required by many keyword search applications in search engines. Finally, the join-based algorithm in S-OLAP cannot be scaled up to distributed computation environment. Our empirical study shows that the join-based algorithm may cause heavy network traffic, and thus, severely degrade the system performance when the inverted lists are stored in different machines. Thus, S-OLAP cannot handle a huge amount of session data generated by commercial search engines.

5.2 Scalable OLAP on Search Logs

In relational data, materialization is often used to make OLAP operations effectively and efficiently. That is, all aggregates are pre-computed and indexed so that whenever a query comes, the result can be retrieved promptly. In search logs, the size of search logs in search engines is usually very large. It is infeasible to scan a search log on the fly to conduct OLAP operations based on the three sequence mining functions. Thus, we need an effective index on search logs. Moreover, even an index on search logs is often too large to be computed and stored in a single machine. Index construction has to be distributed.

Another challenge of supporting OLAP on search log session data using the three basic sequence mining functions is that the dimensions are not explicitly pre-defined. Analogous to the relational case, each query in the session data can be considered as a dimension, while the frequency of query sequences can be considered as the measure. There are a huge number of possible query sequences. Quantitatively, if a length up to l is considered, then the total

SID	Query sequences	SID	Query sequences
s_1	$\langle q_1 q_2 q_3 q_4 \rangle$	s_5	$\langle q_6 q_1 q_2 q_5 \rangle$
s_2	$\langle q_1 q_2 q_4 q_5 \rangle$	s_6	$\langle q_1 q_2 q_3 q_5 \rangle$
s_3	$\langle q_6 q_1 q_2 q_5 \rangle$	s_7	$\langle q_1 q_2 q_3 q_6 \rangle$
s_4	$\langle q_1 q_2 q_3 q_4 \rangle$	s_8	$\langle q_6 q_1 q_2 q_5 \rangle$

Table 5.1: A running example of 8 query sessions.

number of possible query sequences is $\sum_{i=1}^{l} |Q|^l$, where Q is the set of unique queries. In a search engine, there can be billions of unique queries, which make the pre-computation for even short query sequences (e.g., $l = 3$) infeasible.

The framework of our OLAP system on search logs, as shown in Figure 5.1, consists of a *log data engine* and an *OLAP engine*. The log data engine extracts query sessions from search logs as described in Section 5.1.1, constructs distributed indexes from query sessions, and maintains the indexes when new logs arrive. We introduce the index structure in Section 5.2.1, and describe in Section 5.2.2 how to construct and maintain the index structures in a distributed manner.

The OLAP engine delegates online mining requests from keyword search applications to corresponding index servers, and integrates results from index servers. We describe online mining in Section 5.2.3.

5.2.1 Suffix Trees and Reversed Suffix Trees

A core task in the three basic sequence mining functions is *subsequence matching*: given a set of sequences D and a query sequence s, find the sequences in D of which s is a subsequence. Suffix trees and their compacted forms [36] are effective data structures to solve the problem, since a subsequence s' of a sequence s must be a prefix of a suffix of s.

A suffix tree organizes all suffixes of a given sequence into a prefix sharing tree such that each suffix corresponds to a path from the root node to a leaf node in the tree. By organizing all the suffixes of s into a tree structure, to check whether sequence s' is a subsequence of s, we can simply examine whether there is a path corresponding to s' from the root of the suffix tree.

Most of the existing methods [34] construct a suffix tree for indexing one (long) sequence. However, in the case of query session mining, the average sequence length is short, but the

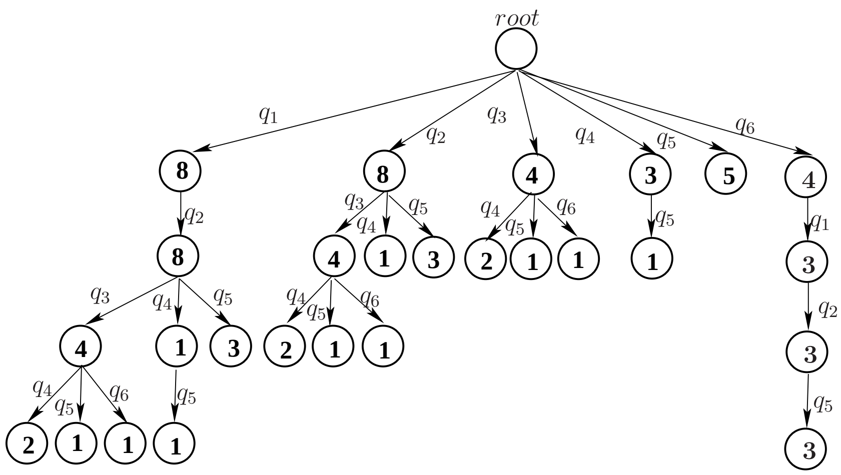


Figure 5.2: A suffix tree for Table 5.1.

number of sequences is huge. We extend the suffix tree structure for indexing query sessions straightforwardly. Figure 5.2 shows a suffix tree for the query sessions in Table 5.1. In the tree, each edge is labeled by a query and each node except for the root corresponds to the query sequence constituted by the labels along the path from the root to that node. Moreover, each node is associated with a value representing the frequency of the corresponding query sequence in the search log. Algorithm 9 outlines the pseudo code of the suffix tree construction algorithm.

We use two special symbols, § and #, in the suffix tree to denote the start and the end of a query session. For example, query session $s = \langle q_1 q_2 q_3 \rangle$ is converted to $s' = \langle \S q_1 q_2 q_3 \# \rangle$ before it is indexed into the suffix tree. As to be shown in Section 5.2.3, these two symbols are important in online mining.

To serve backward search, we also build a reversed suffix tree. For each query session $s = \langle q_1 q_2 \ldots q_n \rangle$, we obtain a reversed query sequence $s' = \langle q_n q_{n-1} \ldots q_1 \rangle$ and insert all suffixes of s' into the reversed suffix tree. Figure 5.3 shows the reversed suffix tree for the query sessions in Table 5.1.

5.2.2 Distributed Suffix Tree Construction

A search log may contain billions of query sessions. The resulting suffix tree and reversed suffix tree cannot be held into the main memory or even the disk of one machine. The

Algorithm 9 Building suffix and reverse suffix trees

```
Input: A set of query sessions $D = \{s_1, \ldots, s_N\}$;
Output: A suffix tree T and a reverse suffix tree RT;
Initialization: T = RT = new tree($\emptyset$);
 1: for all session $s_i = \langle q_1 \ldots q_{ni} \rangle$ in $D$ $(1 \leqslant i \leqslant N)$ do
 2:     $s_i = \langle \S q_1 \ldots q_{ni} \# \rangle$; $rs_i = \langle \# q_{ni} \ldots q_1 \S \rangle$
 3:     for all suffix $s'_i$ of $s_i$ do
 4:         if $(|s'_i| > 1)$ then Insert($s'_i$, T);
 5:     end for
 6:     for all suffix $rs'_i$ of $rs_i$ do
 7:         if $(|rs'_i| > 1)$ then Insert($rs'_i$, RT);
 8:     end for
 9: end for
10: return T and RT;
Method: Insert($s$, T)
 1: curN = T;
 2: for all element $q$ in $s$ do
 3:     if $q \in$ curN.child then
 4:         curN = curN.child[q]; curN.count++;
 5:     else
 6:         chN = new tree($q$); chN.count = 1;
 7:         curN.child.add(chN); curN = chN;
 8:     end if
 9: end for
```

existing studies on disk-based suffix tree construction (e.g., [60]) target at indexing one long sequence. To the best of our knowledge, there is no existing work on efficient construction of a suffix tree for a large number of sequences on multiple computers.

We develop a distributed suffix tree construction method under the MapReduce programming model [22], a general architecture for distributed processing on large computer clusters. The central idea of MapReduce is to divide a large task into chunks so that they can be assigned to computers for processing in parallel. At the beginning, the whole data set is stored distributively in the cluster; each computer possesses a subset of data. In the map phase, each computer processes its local subset of data and emits pieces of intermediate results, where each piece is associated with a key. In the reduce phase, all pieces of intermediate results carrying the same key are collected and processed on the same computer.

Figure 5.4 shows the major steps of our distributed index construction method. In the first step, we compute all suffixes and the corresponding frequencies using the MapReduce model. In the second step, we partition the entire set of suffixes into several parts such that each part can be held in the main memory of one index server. The first two steps are

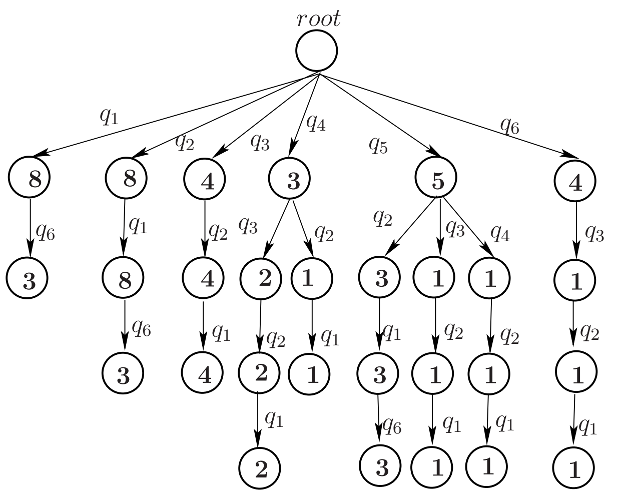


Figure 5.3: A reversed suffix tree for Table 5.1.

conducted under the MapReduce model. In the last step, we construct the local suffix trees and reversed suffix trees on each index server. As to be explained shortly, the partitioning method in the second step guarantees that the local (reversed) suffix trees are subtrees of the global (reversed) suffix trees.

In the map phase of Step 1, each computer processes a subset of query sessions. For each query session s, the computer emits an intermediate key-value pair $(s', 1)$ for every suffix s' of s, where the value 1 here is the contribution to frequency of suffix s' from s. In the reduce phase, all intermediate key-value pairs having suffix s' as the key are processed on the same computer. The computer simply outputs a final pair $(s', freq(s'))$ where $freq(s')$ is the number of intermediate pairs carrying key s'.

The above MapReduce method returns all suffixes of sessions and their frequencies. We need to organize them into a suffix tree. Ideally, we want that the suffix tree can be held in main memory so that online mining can be conducted quickly. However, since the number of all suffixes is usually very large, the whole suffix tree cannot fit in one machine. To tackle the problem, we partition a suffix tree into subtrees so that each subtree can be held into the main memory of an index server. Moreover, we require all subtrees are exclusive from each other so that there are no identical paths on two subtrees. Finally, we try to make sure that the sizes of the subtrees do not vary dramatically in the hope that the online mining

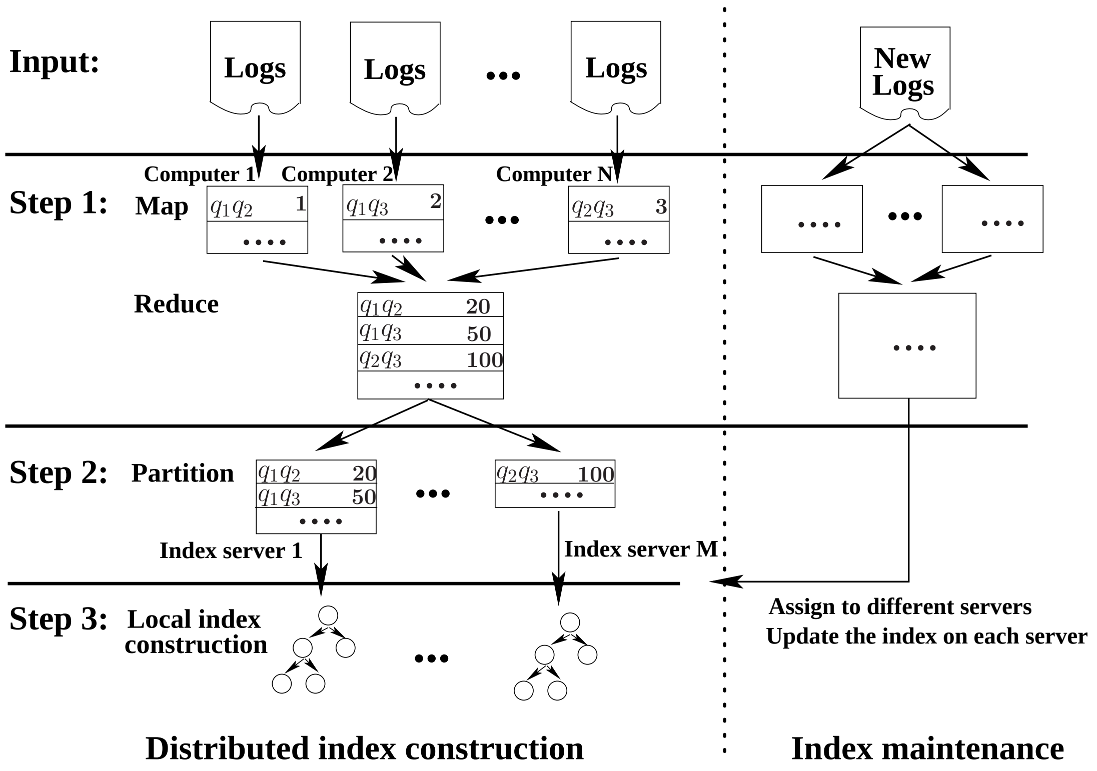


Figure 5.4: Distributed index construction in 3 steps.

workload can be distributed relatively evenly on the index servers.

One challenge is that it is hard to estimate the size of a subtree using only the suffixes in the subtree, since the suffixes may share common prefixes. For example, a subtree with two suffixes $s_1 = \langle q_1q_2q_3 \rangle$ and $s_2 = \langle q_1q_2q_4 \rangle$ has only 4 nodes since the two suffixes share a prefix $\langle q_1q_2 \rangle$.

Given a set of suffix sequences, a simple yet reachable upper bound of the size of the suffix tree constructed from the suffix sequences is the total number of query instances in the suffix sequences. For example, the upper bound of the size of the suffix tree constructed from $s_1 = \langle q_1q_2q_3 \rangle$ and $s_2 = \langle q_1q_2q_4 \rangle$ is 6. Using this upper bound in space allocation is conservative. The advantage is that we reserve sufficient space for the growth of the tree when new search logs are added.

To partition the suffix tree, for each query $q \in Q$, we again apply the MapReduce approach and compute the upper bound of the subtree rooted at q. Algorithm 10 outlines the

Algorithm 10 Suffix tree partition.

Input: The set of unique suffixes S, the set of unique queries Q, and the number of nodes M that can be held in one index server;
Output: The set of sequences S where each $s \in S$ corresponds to the root node of a subtree;
Initiate: $S = \emptyset$;

```
1: for all $q \in Q$ do
2:     $s_q$ = new Element(); $s_q.seq = \langle q \rangle$; $s_q.size = 0$; S.add($s_q$);
3: end for
4: estimate size for each element in S using MapReduce;
5: return Split(S);
```

Method: Split(S)

```
1: toEstimate = $\emptyset$; finished = false;
2: while (!finished) do
3:     finished = true;
4:     for all $s \in$ S do
5:         if (s.size > M) then
6:             finished = false;
7:             for all $q \in$ Q do
8:                 $s_q$ = new Element(); $s_q.seq = s.seq \circ \langle q \rangle$;
9:                 $s_q.size = 0$; toEstimate.add($s_q$);
10:            end for
11:            S.Remove($s$);
12:        end if
13:    end for
14:    estimate the size for each element in toEstimate;
15:    add all the elements in toEstimate into S;
16: end while
```

details of the algorithm. In the map phase, each suffix sequence s generates an intermediate key-value pair $(q_1, |s| - 1)$, where q_1 is the first query in s, and $|s| - 1$ is the number of queries in s except for q_1. In the reduce phase, all intermediate key-value pairs carrying the same key, say q_1, are processed by the same computer. The computer outputs a final pair $(q_1, size)$ where $size$ is the sum of values in all intermediate key-value pairs with key q_1. Clearly, $size$ is the upper bound of the size of the subtree rooted at query q_1. If $size$ is less than the size limit of an index server, the whole subtree rooted at q_1 can be held in the index server. In this case, we assign all the suffixes whose first query is q_1 to the same index server. Otherwise, we can further divide the subtree rooted at q_1 recursively and assign the suffixes accordingly. In this way, we can guarantee that the local suffix trees on different index servers are exclusive. Algorithm 10 provides the pseudo codes of the partition procedure.

New search log sessions keep arriving incrementally. To incrementally maintain the suffix tree, when a new batch of query sessions arrive, we only process the new batch using the MapReduce process similar to that in Step 1 and compute the frequency $freq(s)$ of each suffix s within the new batch. Then, the new set of suffixes are assigned to the index server according to the subtrees that they should be hosted. If a subtree in an index server exceeds the memory capacity after the new suffixes are inserted, the subtree is partitioned recursively as in Step 2 and more index servers are used. Finally, each local suffix tree is updated to incorporate the new suffixes.

The reversed suffix trees can be constructed and maintained in a similar way.

5.2.3 Online Search Log Mining

To conduct OLAP on session data, a user can specify a query sequence s. Analogous to the relational case, each query in s can be considered as a dimension, while the frequency of s can be considered as the measure. The three basic session sequence mining functions in Section 5.1.2 are then regarded as the aggregate functions. The *sequential drill-down operations* (drill-down for short) on s can be defined as aggregations by either pre-pending a sequence s_1 at the head of s or appending s_1 at the tail of s. In other words, the sequential drill-down operations perform on either sequence $s_1 \circ s$ or sequence $s \circ s_1$. Reversely, the *sequential roll-up operations* (roll-up for short) on s can be defined as aggregations by removing a subsequence of s either at the head or tail.

In this section, we discuss the implementation of the three basic sequence mining functions online using a suffix tree and a reversed suffix tree.

Forward Search and Backward Search

A forward search can be implemented using a suffix tree. Let s be a query sequence. We can search the suffix tree and find a path from the root to a node v that matches s. If such a path does not exist, then no query session contains s and thus nothing can be returned by the forward search. If such a path is found, then we only need to search the subtree rooted at v, and find the top-k nodes in the subtree with the largest frequencies. The paths from v to those nodes are the answers.

A thorough search of the subtree rooted at v can be costly if the subtree is large. It is easy to see that in a suffix tree, the frequency at a node v is always larger than or equal to that at any descendant of the node. Thus, we can conduct a best-first search to find the top-k answers.

Example 15 (Forward search) *Consider the sessions in Table 5.1 and the corresponding suffix tree in Figure 5.2. Suppose the query sequence is $s = \langle q_1 q_2 \rangle$. A forward search starts from the root node and finds a path matching s (the left most path in the figure).*

The node v corresponding to s has 3 child nodes. We can follow the labels on the edges from v and form a candidate answer set $Cand$. During the forward search process, $Cand$ is maintained as a priority queue in frequency descending order. Therefore, $Cand = \{q_3, q_5, q_4\}$ at the beginning. If the user is interested in top-2 answers, we first pick the head element q_3 from $Cand$. As $Cand$ is maintained as a priority queue, q_3 has the largest frequency and can be safely placed into the final answer set R. This is due to a good property of the suffix tree: any descendant node v' cannot have a frequency higher than that in any of its ancestor nodes v. In the next step, all the sequences corresponding to the child node of v are inserted into $Cand$. The priority queue now becomes $Cand = \{q_5, q_3q_4, q_4, q_3q_5, q_3q_6\}$. Again, we pick the head element q_5 from $Cand$ and place it in R. Therefore, the top-2 answers are $R = \{q_3, q_5\}$. If the user is interested in top-3 answers, the queue is updated to $Cand = \{q_3q_4, q_4, q_3q_5, q_3q_6\}$ since q_5 does not have a child. The top-3 answers are $R = \{q_3, q_5, q_3q_4\}$. ■

A suffix tree may be distributed in multiple index servers. When the search involves multiple index servers, each index server looks up the local subtree and returns the local top-k results to the OLAP server. Since the local subtrees are exclusive, the global top-k results must be among the local top-k results. Therefore, the OLAP server only needs to examine all the local top-k results and select the most frequent ones as the global top-k results.

Similarly, a backward search can be conducted using a reversed suffix tree.

Session retrieval

To conduct session retrieval online, we pre-compute the frequencies of sessions in a session table by a MapReduce process. Specifically, in the map phase, each session s generates a key value pair $(s, 1)$. In the reduce phase, all identical sessions are assigned to the same computer and thus the frequency of the session can be computed.

We enhance the suffix tree by adding the session information. Each leaf node of the suffix tree is attached a list of the IDs of the sessions containing the suffix. This can be

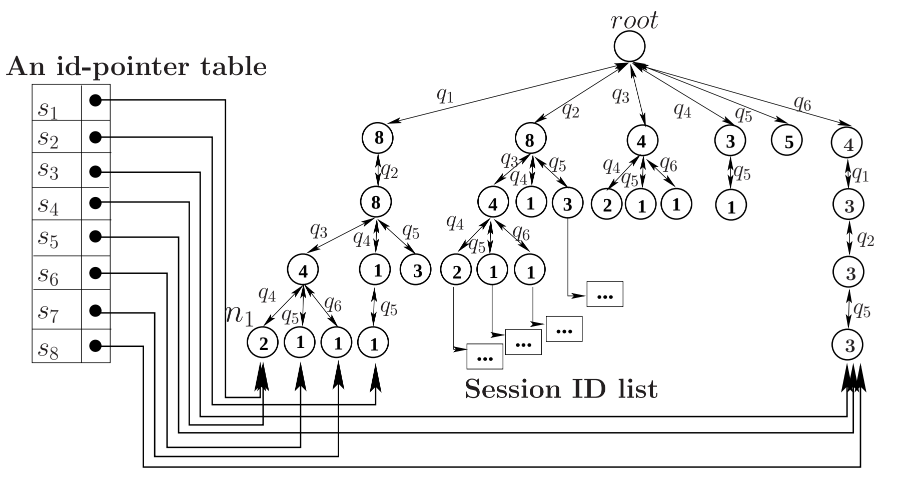


Figure 5.5: An enhanced suffix tree.

easily computed as a byproduct in the suffix tree construction. Moreover, all session IDs in the list are sorted in the frequency descending order. Figure 5.5 shows the enhanced suffix tree of our running example.

Session retrieval with a query sequence s can be conducted online using the enhanced suffix tree. We first find the node v such that the path from the root of the suffix tree to v matches s. We then search the leaf nodes in the subtree rooted at v and find the IDs of the top-k frequent sessions. Once we have the IDs of the top-k sessions, the last step is to get the query sequences of the corresponding sessions. To avoid physically storing the query sequences, which could be expensive in space, we reuse the query sequences on the suffix tree. That is, instead of storing the actual query sequence in the mapping table, for each session ID, we store a pointer to the leaf node corresponding to the query session in the suffix tree. We call this mapping table *id-pointer table*. As an example, in Figure 5.5, the entry for session s_1 in the id-pointer table points to leaf node n_1. To find out the sequence of s_1, we only need to trace the path from the leaf node n_1 back to the root and then reverse the order of the labels on the path. In this example, the path from n_1 to the root is $\langle q_4 q_3 q_2 q_1 \rangle$, and thus $s_1 = \langle q_1 q_2 q_3 q_4 \rangle$.

To speed up the search, we can further store at each internal node v in the suffix tree a list of k_0 sessions that are most frequent in the subtree of v, where k_0 is a number so that most of the session retrieval requests ask for less than k_0 results. In practice, k_0 is often a

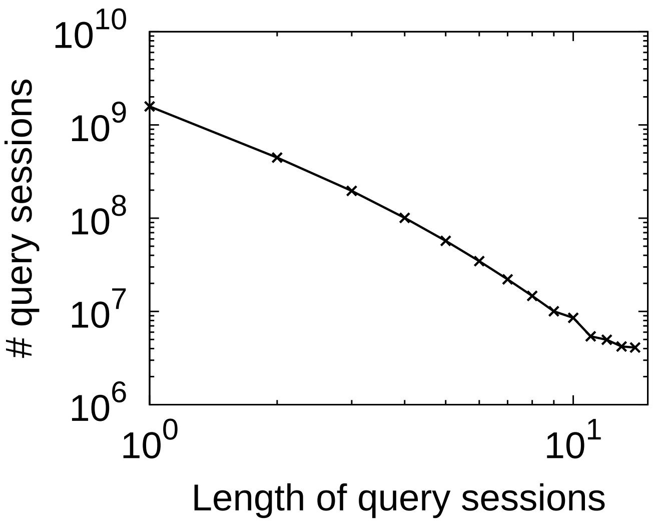


Figure 5.6: The distributions of session lengths.

small number like 10. Once the list is stored, the session retrieval operations requesting less than k_0 results obtain the IDs of the top-k sessions directly from the node and thus do not need to search the leaf nodes in the subtree. Only when a session retrieval requests more than k_0 results we need to search the subtree.

5.3 Performance Study and Experimental Results

In this section, we report a systematic evaluation of our OLAP system using a real query log data set extracted from a major commercial search engine. The data set contains $4,963,601,307$ searches and $1,481,946,526$ unique queries. We apply the session segmentation method discussed in Section 5.1.1 and derive $2,488,594,484$ sessions.

We first examine the characteristics of session data. Figure 5.6 shows the distribution of the length of sessions. For those sessions of lengths greater than 1, the length of sessions follows a power-law distribution. It indicates that the majority of sessions in the search log are not long. As introduced in Section 5.2, the level of a suffix tree is at most the length of the longest query sequence. Therefore, the suffix tree for the session data is not high, which suggests that the query answering process could be very efficient since it is not necessary to traverse deep in the suffix tree index.

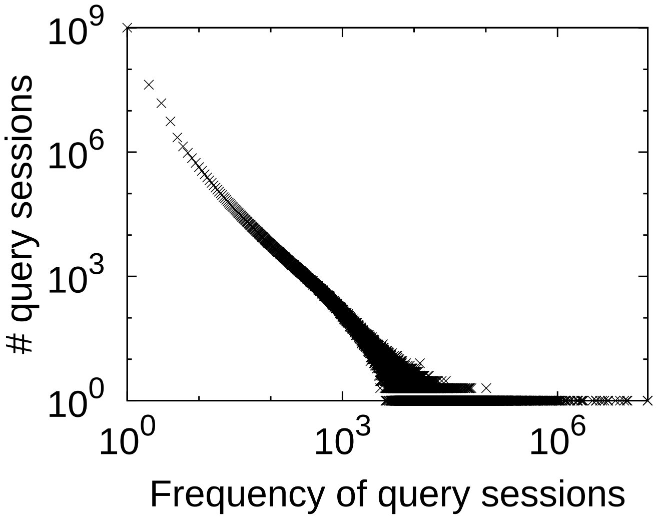


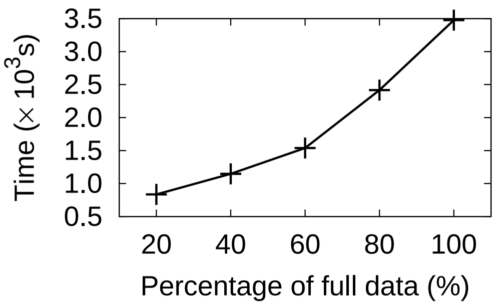
Figure 5.7: The distributions of session frequencies.

We also plot the number of unique sessions with respect to their frequencies in Figure 5.7. The figure shows that the frequency of unique query sessions also follows a power-law distribution. This indicates that the suffix tree index can greatly compress the data since many sessions have identical query sequences.

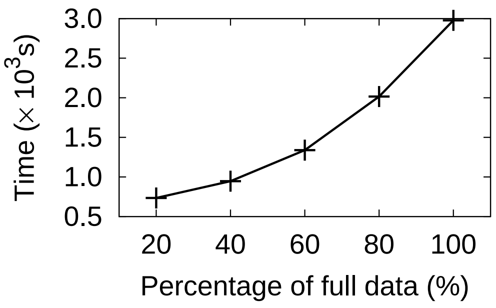
5.3.1 Index Construction

We build the suffix tree index and the reversed suffix tree index using the methods developed in Section 5.2. In Section 5.2.3, we presented several strategies to achieve the sessional retrieval function. Based on the analysis, the strategy in which each leaf node is associated with a list of session IDs can balance the storage size and the query answering response time. By default, the index built in the following subsections refers to that strategy.

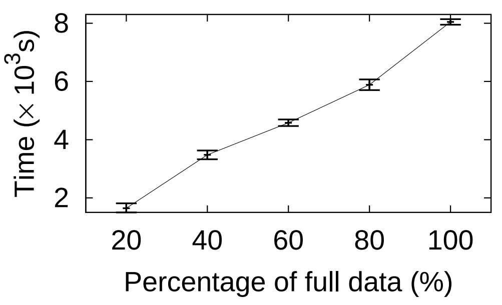
Our index construction methods follow a distributed implementation. We use a MapReduce cluster consisting of 20 computers. The map phase and the reduce phase are automatically scheduled by the system. Moreover, up to 6 computers are used as index servers. All the 26 computers run the 64 bit Microsoft Windows Server 2003 operating system, each with an Intel Xeon 2.33 GHz CPU and 8 GB main memory.



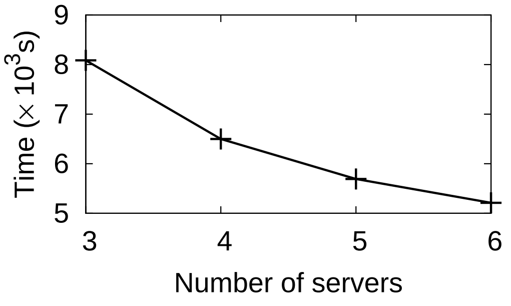
(a) Step 1



(b) Step 2



(c) Step 3 (three servers)



(d) Step 3 (more servers)

Figure 5.8: The index construction time for each step.

To examine the scalability of our index construction algorithm, we measure the construction time with respect to uniform samples of the whole search log data with different sizes. As elaborated in Section 5.2.2, the index construction consists of three steps. The first step extracts all suffixes and their frequencies using a MapReduce approach. The second step partitions the suffixes and allocates them to individual index servers. These two steps are carried out in the MapReduce cluster. The last step constructs the local suffix tree as well as the reversed suffix tree on each index server. We first use three computers as index servers. The run time for each step is shown in Figures 5.8(a), 5.8(b) and 5.8(c), respectively.

For the first two steps, the runtime increases over-linearly when data size increases. By monitoring the CPU usage and I/O status of the computers in the MapReduce system, we find that the overall time of each step is dominated by writing intermediate and final results to disks. In the first step, both intermediate and final results are (suffix, frequency) pairs. Since the total number of intermediate and final pairs increases over-linearly with respect to the size of sessions, so does the total runtime of the first step. In the second step, both intermediate and final results of the MapReduce procedure are (prefix, size) pairs, where prefix corresponds to a subtree and size is the estimated number of nodes in the subtree. To reduce the chances of recursive partition of the suffix and reverse suffix trees, in our implementation, we choose the length of prefixes to be 2. Consequently, the overall time for the second step is over-linear since the total numbers of intermediate and final (prefix, size) pairs increase over-linearly with respect to the size of sessions.

Although the first two steps do not scale linearly, the overall time is acceptable. In particular, even on the whole data set containing about 2.5 billion sessions, the first step and the second step still finish within one hour, respectively. This verifies that our algorithms under the MapReduce programming model are efficient to handle large data sets.

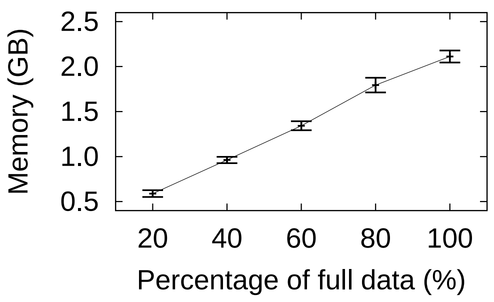
In Figure 5.8(c), the curve shows the average runtime, while the deviation bars indicate the variance of the runtime among the three index servers. We can see the average run time scales well with respect to the size of data. Moreover, the variation of runtime is very small, which indicates that the suffixes are partitioned evenly and the load of the index servers is well balanced.

To measure the effectiveness of distributed indexing, we use more machines as index servers. The runtime for the first two steps is similar to that in Figures 5.8(a) and 5.8(b), since we only add index servers without changing the MapReduce cluster. Figure 5.8(d) shows the average runtime for the third step using four, five, and six machines, respectively. All machines have the same configuration. We can see that the average runtime per index server drops dramatically when more machines are used. Although the decrease is not strictly linear, the average runtime when six machines are used is only 64.4% of that when three machines are used.

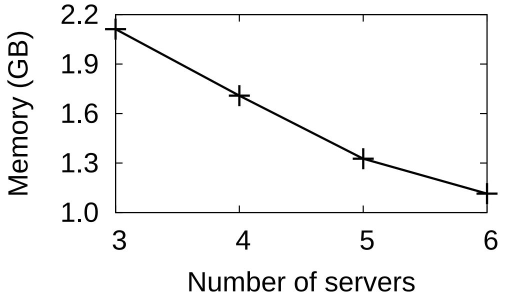
We also measure the size of index storage. Figure 5.9(a) shows the memory usage of the index structure using three servers. Clearly, the index size is roughly linear with respect to the data set size. Moreover, the deviation bars show that the index servers have balanced load. Last, as shown in Figure 5.9(b), the memory usage per index server drops to nearly half when the number of index servers increases from three to six.

5.3.2 OLAP Performance

To test the efficiency of our system, we randomly extract a sample set of $1,000$ query sequences of length varying between 1 and 4 from the original search log. We evaluate the response time on the three OLAP functions, i.e., forward search (FS), backward search

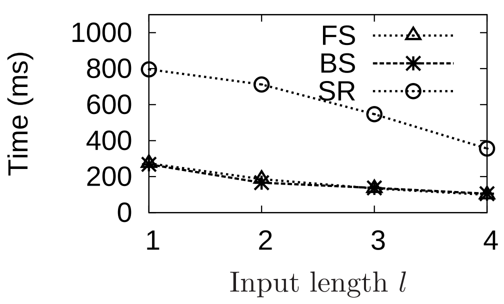


(a) three servers

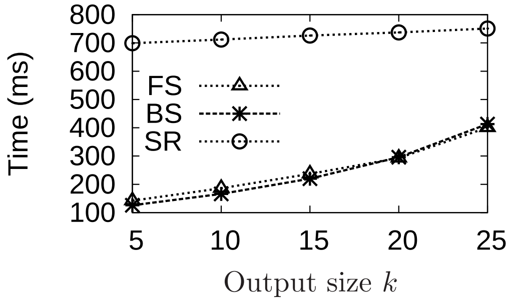


(b) more servers

Figure 5.9: Memory usage of index servers.



(a) input length l



(b) output size k

Figure 5.10: The effect of l and k on response time.

(BS), and session retrieval (SR). Three index servers are used to hold the index and run the mining tasks jointly.

Two factors may affect the response time: the length l of the input query sequences and the answer set size k. In the following experiments, we set $l = 2$ and $k = 10$ by default.

In the experiments, the average response time for a forward/backward search is 0.19 second and the average response time for a session retrieval is 0.72 second. The results verify the feasibility of our approach for online keyword search applications in search engines. Obviously, the response time can be further decreased by caching technologies and more advanced hardware.

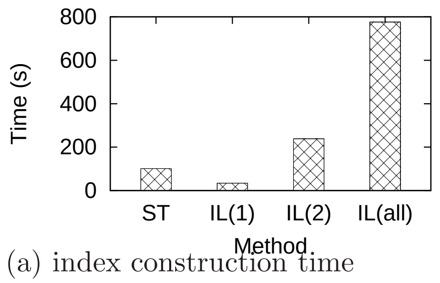
We further investigate the effect of the factors, l and k, respectively, by varying one factor but keeping the other factor fixed at the default value. The effect of the length parameter l is shown in Figure 5.10(a). When l increases, the response time for all three OLAP functions decreases. This is because the longer the query sequence, the more specific the sequence is, and thus the smaller the search space that the query answering algorithms need to traverse.

The effect of the size parameter k is shown in Figure 5.10(b). When k increases, the response time of all the three functions increases. Generating a larger result set requires visiting more candidate nodes during the search procedure.

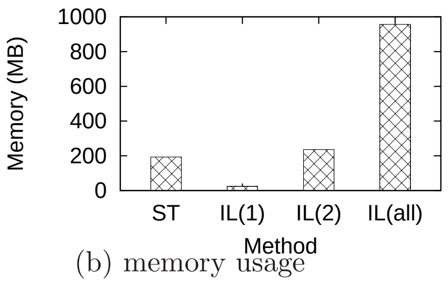
5.3.3 Performance Comparisons with S-OLAP

For comparison, we implement the S-OLAP system as described in [53] and extend the pattern manipulation operations in [53] to answer the forward search, backward search and session retrieval. These extended solutions are referred to as the *inverted list* approach. The main extension is as follows. For each query sequence of length 1, the inverted list method constructs an inverted list in which the IDs of the sessions containing that query sequence are kept. For each query sequence of length greater than one, the inverted list can be derived by joining the two inverted lists for two subsequences. Since the join operation does not guarantee the consecutiveness of subsequences, the joined inverted list has be to scanned once to remove invalid sessions. The join operation in the inverted list approach can be carried out either offline or on the fly. To make a tradeoff between the index storage size and the response time, only the inverted lists for short query sequences, e.g., 1 and 2, are pre-computed offline, while the inverted lists for long sequences are computed on the fly.

It is ideal for the inverted list approach to be deployed distributively on multiple machines. Unfortunately, The inverted list approach cannot be extended to a distributed computation environment straightforwardly. We observe several reasons from the empirical study. First, during the query answering procedure, the inverted lists for long sequences have to be computed on the fly. In a distributed environment, the inverted lists to be joined may be stored in different machines. Consequently, the join operation may make the network traffic a bottleneck of the system. Moreover, the computation of the inverted lists for long sequences may need multiple rounds of join operation, where each round involves the combination of subsequences. Since the number of intermediate joining results may grow exponentially with respect to the number of subsequences, it is a costly to maintain the intermediate joining results in a distributed way.



(a) index construction time



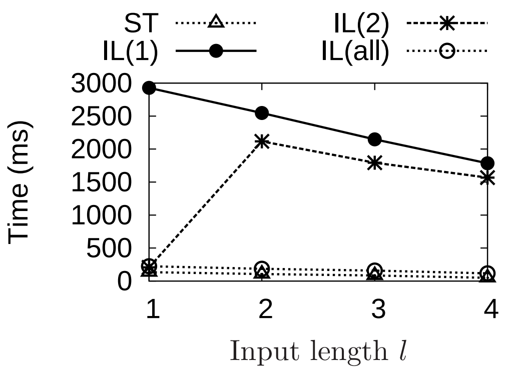
(b) memory usage

Figure 5.11: Comparison between ST and IL on (a) index construction time and (b) memory usage.

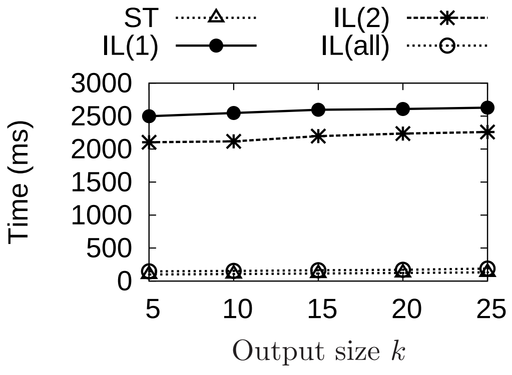
Since the inverted list approach cannot be easily extended to a distributed environment, we use a small uniform sample data set whose inverted lists can be held in the main memory of a single machine. This small set contains $27,091,874$ searches, $13,587,124$ sessions, and $6,579,346$ unique queries. To make a fair comparison, our method also uses only one index server. Moreover, for a thorough comparison, we consider different options to construct the inverted lists. Specifically, **IL(1)** refers to the option that only the inverted lists for sequences of length 1 are pre-computed, **IL(2)** refers to the option that the inverted lists for sequences of lengths 1 and 2 are pre-computed, and **IL(all)** refers to that the complete set of inverted lists for all sequences are pre-computed. Finally, we use **ST** to refer to our suffix tree method.

We first examine the index construction time for the suffix tree approach and the inverted list approach. For the suffix tree approach, the index construction includes building both the suffix tree and the reversed suffix tree. The results are shown in Figure 5.11(a). When the **IL(1)** option is adopted, only the inverted lists for the sequences of length 1 are pre-computed. Therefore, the construction time for the **IL(1)** option is the smallest. However, when the inverted lists for longer sequences are built, the index construction time increases dramatically. The runtime for the **IL(all)** option is the most costly, almost 7 times longer than that of the suffix tree approach.

Figure 5.11(b) shows the actual memory usage of the indexes in different approaches. The results have a trend similar to that in Figure 5.11(a). The **IL(1)** option has the smallest index size, while the inverted lists for option **IL(all)** are the largest. The index of the suffix tree approach is slightly smaller than that of option **IL(2)**.



(a) input length l



(b) output size k

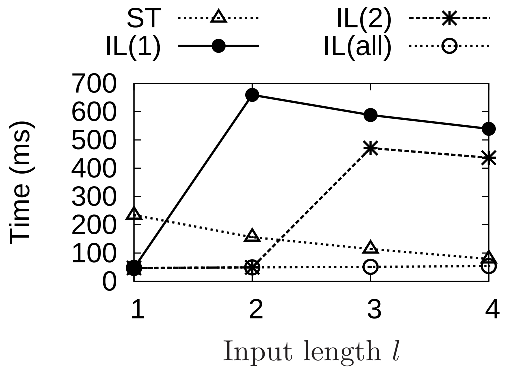
Figure 5.12: Comparison on forward search.

We further analyze the query response time of our approach and the inverted list approach. Similar to the evaluation in Section 5.3.2, we consider the effect of the length l of the input query sequences and the size k of answer sets. By default, $l = 2$ and $k = 10$.

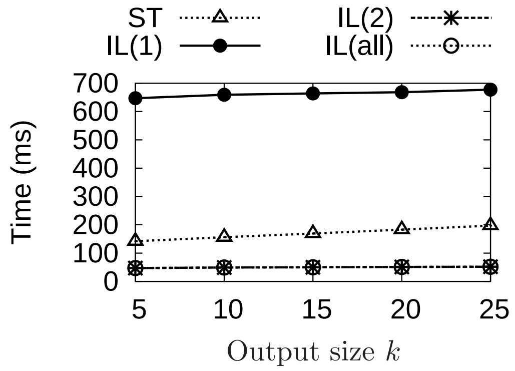
Figure 5.12(a) shows the query response time when the length of input query sequences varies from 1 to 4. Our suffix tree approach has the shortest response time, and the **IL(all)** option has comparable performance. However, the **IL(1)** option is very slow for all lengths of input query sequences, while the response time for option **IL(2)** increases dramatically when the length of input sequences is greater than 1. In general, to conduct the forward search for a query sequence of length l, the inverted list approach needs to look up the inverted lists for sequences of length $l + 1$. If the inverted lists are not pre-computed, they have to be generated by expensive join operations on the fly. That is the reason why option **IL(1)** is slow for all lengths and option **IL(2)** becomes dramatically more costly at length 2.

Figure 5.12(b) shows the query response time when the size k changes. For all the four curves, the response time increases when k increases because the larger the size k, the more candidates need to be processed. The suffix tree approach and the **IL(all)** option are much faster than options **IL(1)** and **IL(2)** because by default $l = 2$. We omit the results on backward search here, since the results are similar to those on forward search.

We finally examine the response time for session retrieval. Figure 5.13(a) shows the results when the length l of the input query sequences varies. In general, to retrieve sessions using query sequence of length l, the inverted list approach needs to look up the inverted



(a) input length l



(b) output size k

Figure 5.13: Comparison on session retrieval.

lists for sequences of length l. Consequently, the response time for option **IL(1)** increases dramatically when the length is larger than 1, since the answer to longer sequences requires more join operations on the fly. Similarly, the response time for option **IL(2)** increases dramatically when the length is over 2. The suffix tree approach and the **IL(all)** option have better performance, and the response time is much shorter than that of the cases when the online join operations are executed. Figure 5.13(b) shows the results when the size k varies. The **IL(1)** option is the slowest, since $l = 2$.

The suffix tree approach is more suitable than the inverted list approach to serve as the index structure of a log analysis infrastructure. First, compared with the options such as **IL(1)** and **IL(2)**, where only partial inverted lists are pre-computed, the suffix tree approach is more scalable to the length of input sequences and the efficiency is comparable to that of the **IL(all)** option. Second, compared with the **IL(all)** option, where all inverted lists are pre-computed, the suffix tree approach has much smaller construction time and memory usage. Finally and most importantly, the suffix tree approach can be scaled up to a distributed environment and handle a huge amount of log data in real applications.

5.4 Conclusions

In this chapter, we proposed an OLAP system on search logs. It can support many keyword search applications in search engines, such as query suggestion and keyword bidding. We formalized three OLAP functions on search logs, and developed a simple yet effective suffix-tree index to support online mining of query sessions. We implemented our methods in a prototype system and conducted a systematic empirical study to verify the usefulness and feasibility of our design.

Our system proposed in this chapter is only a first step. There is much more for future work. For example, besides query sequences, we may consider other dimensions of session data, such as the date of sessions and the location of users. As another example, the personalized search application may want to compare the top-k sessions containing query "*LA Lakers*" from Los Angeles in December 2010 with those from US in the fourth quarter of 2010.

As another example, the current OLAP infrastructure is built using the query session data. In some scenarios, the user data that cover consecutive query sessions of the same user may be more desirable. It is interesting to explore whether the current technique can be applied for the user data. Since the length of the user data could be much longer, the depth of the suffix tree could be large. It introduces tremendous challenges in query answering and index maintenance. Can we develop some compression techniques to effectively maintain the index structure?

Chapter 6

Conclusions and Future Work

As our world is now in its information era, huge amounts of structured, semi-structured, and unstructured data are accumulated everyday. A real universal challenge is to find useful information from large collections of data to capture users' information needs. Keyword search on large-scale data is a fast-growing research direction to meet this challenge. Many kinds of techniques have been developed in the past decade. In this thesis, we focus on some challenging problems in keyword search on large-scale data, centered on relational tables, graphs, and search logs. Moreover, the proposed techniques have several desirable characteristics which are useful in different application scenarios.

In this chapter, we first summarize the thesis, and then discuss some interesting future research directions.

6.1 Summary of the Thesis

Keyword search provides a simple but powerful solution for millions of users to search information from large-scale data. Due to the high demands of managing and processing large-scale structured, semi-structured, and unstructured data in various emerging applications, keyword search becomes an important technique. In this thesis, we study a class of important challenging problems centered on keyword search on relational tables, graphs, and search logs. In particular, we make the following contributions.

- We provide an overview of the keyword search problem on large-scale data. Two important tasks, keyword search and query suggestion, are discussed. The task of keyword search highly relies on understanding two important concepts, that is, the data model and the answer model.

- For relational tables, we show that, while searching individual tuples using keywords is useful, in some scenarios, it may not find any tuples since keywords may be distributed in different tuples. We propose a novel method of *aggregate keyword search* by finding an aggregate group of tuples jointly matching a set of query keywords. We develop two interesting approaches to tackle the problem. We further extend our methods to allow partial matches and matches using a keyword ontology. An extensive empirical evaluation using both real data sets and synthetic data sets is reported to verify the effectiveness of aggregate keyword search and the efficiency of our methods.

- For graphs, we indicate that developing efficient and effective query suggestion techniques is crucial to provide satisfactory user search experience. We propose a practical solution framework, and develop efficient methods to recommend keyword queries for keyword search on graphs. Our experimental results conducted on large real graph data sets indicate that the problem of query suggestion for keyword search on graphs is necessary and important, and our proposed techniques are efficient and effective to provide meaningful recommendations for keyword search on graphs.

- For search logs, we admit that more and more keyword search applications are being developed in search engines based on search logs, such as query suggestion, keyword bidding, and dissatisfactory query analysis. By observing that many keyword search applications in search engines highly rely on online mining of search logs, we develop an OLAP system on search logs which serves as an infrastructure supporting various keyword search applications. An empirical study using real data of over two billion query sessions demonstrates the usefulness and the feasibility of our design.

6.2 Future Research Directions

With the simplicity and the power of the keyword search technique, it is interesting to re-examine and explore many related problems, extensions and applications in the future. Some of them are listed here.

- *Information unit-based keyword search.* Web search engines nowadays are commonly used by users to retrieve information on the Web. To find information about the available funding opportunities in Canada, a user may submit a query with keywords `Canada`, `funding` and `computer science` to a search engine. Issuing the above query to a popular Web search engine may surprisingly find that the returned results missed some relevant funding opportunities. The main reason is that Website owners prefer to create Websites that are composed of multiple pages connected using hyper-links. Therefore, the contents are often distributed into multiple pages. The current Web search engines retrieve individual pages matching all (or most) of the keywords in the query and rank them high in the results. However, individual pages may not contain all the information for a query. It is crucial that the structure of Web pages, as well as the semantic relationships among those pages, are taken into consideration for more accurate Web information search.

An interesting future research direction is to retrieve relevant information based on *Minimal Information Unit*, which is considered to be a minimal logical document. The minimal logical document can be either an individual page or multiple pages forming one atomic unit. For instance, given the query with keywords `Canada`, `funding` and `computer science`, an answer to the information unit-based search could be: (1) an individual page matching all (or most) of the keywords; or (2) a minimal set of pages interconnected with hyper-links together matching all (or most) of the keywords. Therefore, the user can retrieve all the valuable information, without missing any relevant funding opportunities available in Canada.

- *Data-driven information search.* The current keyword search technique in Web search engines tries to match the user query keywords against a large collection of textual pages. In recent years, many research activities have been focusing on either providing meaningful query suggestions or improving the matching techniques between the queries and the pages. However, other than the unstructured textual pages on the Web, there also exist vast collections of structured data (e.g., relational tables) and semi-structured data (e.g., XML data, labeled graphs). There is no effort on analyzing whether the data is suitable for answering specific keyword queries. What if the best answer to a query is contained in structured/semi-structured data other than unstructured textual pages? What if some semantics are included in the query?

  Consider a keyword query `movies in Vancouver`, the user in fact is looking for the showtime lists of movies in Vancouver. If we consider this query as a matching of keywords over unstructured textual pages, only those Web pages containing the keywords will be returned to the user. However, in this case, the movies on shown in Vancouver without a keyword `movie` will not be returned. In fact, such queries can be answered better using structured/semi-structured data. For instance, a showtime movie listing table would provide accurate answers to the query `movies in Vancouver`.

In some other situations, user queries may carry semantic meanings. Consider a keyword query `laptop about $1000`. Here, the user is implying that he is interested in laptop price. The traditional keyword matching cannot return the exact answers to the user. For instance, laptops about $900 or laptops about $1100 will not be matched with the query. When the query itself contains rich semantics, performing those queries over free text documents should not be straightforwardly keyword matching. In such scenarios, data becomes more important. If there exist appropriate structured/semi-structured data which contain information such as laptops and their prices, the query could be well interpreted using those data.

Generally, sorting the results based on IR-style relevance may not be the best solution in some particular scenarios. We may need to perform a deeper analysis of the query in order to understand the semantics and return better results by using appropriate data sources.

- *Real-time information management and search.* The Web 2.0 technique has created huge interactive information sharing. One obstacle nowadays requiring further exploration is searching important information at real-time (e.g., popular tweets on Twitter), even before they are widely spread on the Web. To address the problem, it is necessary to understand and predict the patterns of information spread. In the context of Web 2.0, people are actively involved in the information spread. To understand the dynamics of real-time information search, fundamentally, two important issues need to be considered. The first issue is to reason how information moves between people through social connections. The second issue is to trace the small fragments of information themselves. For practical applications, it is also interesting to obtain a better understanding on how to leverage the vast network data for real-time search and how to maintain the fast-evolving data.

- *Preserving privacy in keyword search.* The Web search engines nowadays must have access to users' personal information to provide accurate search results. For instance, social search is one type of Web search which takes into account the social graph of the person initiating the search query. To perform a search process, social search engines take into account various sources of users' social information, such as collaborative discovery of Web pages, social tags, social comments, and so on. In such a case, the search providers have to be trusted to not abuse their privilege. However, this arrangement in practice is not always desirable. As a real-life example, the AOL company recently released a subset of the Web search log data without any notice. Researchers have shown that some detailed user profiles can be constructed from the released data [84].

Although the search engines could provide personalized search results to users, the dramatic usage of personal social information poses a big privacy concern. It could be desirable for a search engine to perform accurate keyword search for users while protecting their privacy well.

Bibliography

[1] Eugene Agichtein, Eric Brill, and Susan Dumais. Improving web search ranking by incorporating user behavior information. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR'06)*, pages 19–26, New York, NY, USA, 2006. ACM.

[2] Eugene Agichtein, Eric Brill, Susan Dumais, and Robert Ragno. Learning user interaction models for predicting web search result preferences. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR'06)*, pages 3–10, New York, NY, USA, 2006. ACM.

[3] Sanjay Agrawal, Surajit Chaudhuri, and Gautam Das. DBXplorer: A system for keyword-based search over relational databases. In *Proceedings of the 18th International Conference on Data Engineering (ICDE'02)*, pages 5–16, Washington, DC, USA, 2002. IEEE Computer Society.

[4] S. Amer-Yahia, P. Case, T. Rölleke, J. Shanmugasundaram, and G. Weikum. Report on the DB/IR panel at sigmod 2005. *SIGMOD Record*, 34(4):71–74, 2005.

[5] Ricardo A. Baeza-Yates, Carlos A. Hurtado, and Marcelo Mendoza. Query recommendation using query logs in search engines. In *EDBT Workshops*, volume 3268 of *Lecture Notes in Computer Science*, pages 588–596. Springer, 2004.

[6] Ricardo A. Baeza-Yates and Berthier A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press/Addison-Wesley, 1999.

[7] Andrey Balmin, Vagelis Hristidis, and Yannis Papakonstantinou. Objectrank: authority-based keyword search in databases. In *Proceedings of the Thirtieth international conference on Very large data bases (VLDB'04)*, pages 564–575. VLDB Endowment, 2004.

[8] Y. Bartal. Probabilistic approximation of metric spaces and its algorithmic applications. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS'96)*, page 184, Washington, DC, USA, 1996. IEEE Computer Society.

[9] Kevin Beyer and Raghu Ramakrishnan. Bottom-up computation of sparse and iceberg cube. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data (SIGMOD'99)*, pages 359–370, New York, NY, USA, 1999. ACM.

[10] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *Proceedings of the 18th International Conference on Data Engineering (ICDE'02)*, pages 431–440. IEEE Computer Society, 2002.

[11] Huanhuan Cao, Daxin Jiang, Jian Pei, Enhong Chen, and Hang Li. Towards context-aware search by learning a very large variable length hidden markov model from search logs. In *Proceedings of the 18th International World Wide Web Conference (WWW'09)*, pages 191–200, Madrid, Spain, April 20-24 2009.

[12] Huanhuan Cao, Daxin Jiang, Jian Pei, Qi He, Zhen Liao, Enhong Chen, and Hang Li. Context-aware query suggestion by mining click-through and session data. In *Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD'08)*, pages 875–883, New York, NY, USA, 2008. ACM.

[13] Surajit Chaudhuri and Gautam Das. Keyword querying and ranking in databases. *PVLDB*, 2(2):1658–1659, 2009.

[14] Surajit Chaudhuri, Raghu Ramakrishnan, and Gerhard Weikum. Integrating DB and IR technologies: What is the sound of one hand clapping? In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research (CIDR'05)*, pages 1–12, 2005.

[15] Yi Chen, Wei Wang, Ziyang Liu, and Xuemin Lin. Keyword search on structured and semi-structured data. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD'09)*, pages 1005–1010. ACM, 2009.

[16] Paul Alexandru Chirita, Claudiu S. Firan, and Wolfgang Nejdl. Personalized query expansion for the web. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR'07)*, pages 7–14, New York, NY, USA, 2007. ACM.

[17] Kenneth Church and Bo Thiesson. The wild thing! In *Proceedings of the ACL 2005 on Interactive poster and demonstration sessions (ACL'05)*, pages 93–96, Morristown, NJ, USA, 2005. Association for Computational Linguistics.

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[19] Hang Cui, Ji-Rong Wen, Jian-Yun Nie, and Wei-Ying Ma. Probabilistic query expansion using query logs. In *Proceedings of the 11th international conference on World Wide Web (WWW'02)*, pages 325–332, New York, NY, USA, 2002. ACM.

[20] Bhavana Bharat Dalvi, Meghana Kshirsagar, and S. Sudarshan. Keyword search on external memory data graphs. *Proc. VLDB Endow.*, 1(1):1189–1204, 2008.

[21] Mariam Daoud, Lynda-Tamine Lechani, and Mohand Boughanem. Towards a graph-based user profile modeling for a session-based personalized search. *Knowledge and Information Systems*, 21(3):365–398, 2009.

[22] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation (OSDI'04)*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[23] Bolin Ding, Jeffrey Xu Yu, Shan Wang, Lu Qin, Xiao Zhang, and Xuemin Lin. Finding top-k min-cost connected trees in databases. In *Proceedings of the 23rd IEEE International Conference on Data Engineering (ICDE'07)*, pages 836–845, Washington, DC, USA, 2007. IEEE Computer Society.

[24] Zhicheng Dou, Ruihua Song, and Ji-Rong Wen. A large-scale evaluation and analysis of personalized search strategies. In *Proceedings of the 16th international conference on World Wide Web (WWW'07)*, pages 581–590, New York, NY, USA, 2007. ACM.

[25] S. E. Dreyfus and R. A. Wagner. The steiner problem in graphs. *Networks*, 1:195–207, 1972.

[26] Georges E. Dupret and Benjamin Piwowarski. A user browsing model to predict search engine click data from past observations. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR'08)*, pages 331–338, New York, NY, USA, 2008. ACM.

[27] Scott Epter, Mukkai Krishnamoorthy, and Mohammed Zaki. Clusterability detection and initial seed selection in large data sets. Technical Report 99-6, Computer Science, Rensselaer Polytechnic Institute, Troy, NY, USA, 1999.

[28] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing iceberg queries efficiently. In *Proc. 1998 Int. Conf. Very Large Data Bases (VLDB'98)*, pages 299–310, New York, NY, Aug. 1998.

[29] Christiane Fellbaum, editor. *WordNet: an electronic lexical database*. MIT Press, 1998.

[30] Y. Feng, D. Agrawal, A. E. Abbadi, and A. Metwally. Range Cube: Efficient cube computation by exploiting data correlation. In *Proc. 2004 Int. Conf. Data Engineering (ICDE'04)*, pages 658–669, Boston, MA, April 2004.

[31] Ariel Fuxman, Panayiotis Tsaparas, Kannan Achan, and Rakesh Agrawal. Using the wisdom of the crowds for keyword generation. In *Proceedings of the 17th International Conference on World Wide Web (WWW'08)*, pages 61–70. ACM, 2008.

[32] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2 edition, 2008.

[33] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[34] Robert Giegerich and Stefan Kurtz. From ukkonen to mccreight and weiner: A unifying view of linear time suffix tree construction. *Algorithmica*, 19:331–353, 1997.

[35] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab and sub-totals. In *Proc. 1996 Int. Conf. Data Engineering (ICDE'96)*, pages 152–159, New Orleans, Louisiana, Feb. 1996.

[36] Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

[37] J. Han, J. Pei, G. Dong, and K. Wang. Efficient computation of iceberg cubes with complex measures. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data (SIGMOD'01)*, pages 1–12, Santa Barbara, CA, May 2001.

[38] Donna Harman, R. Baeza-Yates, Edward Fox, and W. Lee. Inverted files. In *Information retrieval: data structures and algorithms*, pages 28–43, Upper Saddle River, NJ, USA, 1992. Prentice-Hall, Inc.

[39] Hao He, Haixun Wang, Jun Yang, and Philip S. Yu. Blinks: ranked keyword searches on graphs. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (SIGMOD'07)*, pages 305–316, New York, NY, USA, 2007. ACM.

[40] Vagelis Hristidis, Luis Gravano, and Yannis Papakonstantinou. Efficient ir-style keyword search over relational databases. In *Proceedings of the 29st international conference on Very large data bases (VLDB'03)*, pages 850–861, 2003.

[41] Vagelis Hristidis and Yannis Papakonstantinou. Discover: Keyword search in relational databases. In *Proceedings of the 28st international conference on Very large data bases (VLDB'02)*, pages 670–681. Morgan Kaufmann, 2002.

[42] Chien-Kang Huang, Lee-Feng Chien, and Yen-Jen Oyang. Relevant term suggestion in interactive web search based on contextual information in query session logs. *J. Am. Soc. Inf. Sci. Technol.*, 54(7):638–649, 2003.

[43] Thorsten Joachims. Optimizing search engines using clickthrough data. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining (KDD'02)*, pages 133–142, New York, NY, USA, 2002. ACM.

[44] Rosie Jones, Benjamin Rey, Omid Madani, and Wiley Greiner. Generating query substitutions. In *Proceedings of the 15th international conference on World Wide Web (WWW'06)*, pages 387–396, New York, NY, USA, 2006. ACM.

[45] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S. Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional expansion for keyword search on graph

databases. In *Proceedings of the 31st international conference on Very large data bases (VLDB'05)*, pages 505–516. ACM, 2005.

[46] Varun Kacholia, Shashank Pandit, Soumen Chakrabarti, S. Sudarshan, Rushi Desai, and Hrishikesh Karambelkar. Bidirectional expansion for keyword search on graph databases. In *Proceedings of the 31st international conference on Very large data bases (VLDB'05)*, pages 505–516. ACM, 2005.

[47] Benny Kimelfeld and Yehoshua Sagiv. Finding and approximating top-k answers in keyword proximity search. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS'06)*, pages 173–182, New York, NY, USA, 2006. ACM.

[48] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithm (SODA'98)*, pages 668–677. ACM, 1998.

[49] Guoliang Li, Beng Chin Ooi, Jianhua Feng, Jianyong Wang, and Lizhu Zhou. Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD'08)*, pages 903–914, New York, NY, USA, 2008. ACM.

[50] Jianxin Li, Chengfei Liu, Rui Zhou, and Wei Wang. Suggestion of promising result types for xml keyword search. In *Proceedings of the 13th International Conference on Extending Database Technology (EDBT'10)*, pages 561–572. ACM, 2010.

[51] Mu Li, Yang Zhang, Muhua Zhu, and Ming Zhou. Exploring distributional similarity based models for query spelling correction. In *Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics (ACL'06)*, pages 1025–1032, Morristown, NJ, USA, 2006. Association for Computational Linguistics.

[52] Fang Liu, Clement Yu, Weiyi Meng, and Abdur Chowdhury. Effective keyword search in relational databases. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data (SIGMOD'06)*, pages 563–574, New York, NY, USA, 2006. ACM.

[53] Eric Lo, Ben Kao, Wai-Shing Ho, Sau Dan Lee, Chun Kit Chui, and David W. Cheung. Olap on sequence data. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD'08)*, pages 649–660, New York, NY, USA, 2008. ACM.

[54] Yi Luo, Xuemin Lin, Wei Wang, and Xiaofang Zhou. Spark: top-k keyword query in relational databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD'07)*, pages 115–126, New York, NY, USA, 2007. ACM.

[55] Mark Magennis and Cornelis J. van Rijsbergen. The potential and actual effectiveness of interactive query expansion. In *Proceedings of the 20th annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR'97)*, pages 324–332, New York, NY, USA, 1997. ACM.

[56] Christopher D. Manning, Prabhakar Raghavan, , and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[57] Ion Muslea. Machine learning for online query relaxation. In *Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'04)*, pages 246–255, Seattle, Washington, USA, August 22-25, 2004, 2004. ACM.

[58] Raymond T. Ng, Alan S. Wagner, and Yu Yin. Iceberg-cube computation with PC clusters. In *Proc. 2001 ACM-SIGMOD Int. Conf. Management of Data (SIGMOD'01)*, pages 25–36, Santa Barbara, CA, May 2001.

[59] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford University, 1998.

[60] Benjarath Phoophakdee and Mohammed J. Zaki. Genome-scale disk-based suffix tree indexing. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD'07)*, pages 833–844, New York, NY, USA, 2007. ACM.

[61] Lu Qin, Je Xu Yu, and Lijun Chang. Keyword search in databases: the power of rdbms. In *Proceedings of the 35th SIGMOD International Conference on Management of Data (SIGMOD'09)*, pages 681–694, Providence, Rhode Island, USA, 2009. ACM Press.

[62] Lu Qin, Jeffrey Xu Yu, Lijun Chang, and Yufei Tao. Querying communities in relational databases. In *Proceedings of the 25th International Conference on Data Engineering (ICDE'09)*, pages 724–735. IEEE, 2009.

[63] Feng Qiu and Junghoo Cho. Automatic identification of user interest for personalized search. In *Proceedings of the 15th international conference on World Wide Web (WWW'06)*, pages 727–736, New York, NY, USA, 2006. ACM.

[64] Mehran Sahami and Timothy D. Heilman. A web-based kernel function for measuring the similarity of short text snippets. In *Proceedings of the 15th international conference on World Wide Web (WWW'06)*, pages 377–386, New York, NY, USA, 2006. ACM.

[65] Ramakrishnan Srikant and Rakesh Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proceedings of the 5th International Conference on Extending Database Technology (EDBT'96)*, pages 3–17, London, UK, 1996. Springer-Verlag.

[66] Kamal Taha and Ramez Elmasri. Bussengine: a business search engine. *Knowledge and Information Systems*, 23(2):153–197, 2010.

[67] Hanghang Tong, Christos Faloutsos, and Jia-Yu Pan. Random walk with restart: fast solutions and applications. *Knowledge and Information Systems*, 14(3):327–346, 2008.

[68] Kathleen Tsoukalas, Bin Zhou, Jian Pei, and Davor Cubranic. Personalizing entity detection and recommendation with a fusion of web log mining techniques. In *Proceedings of the 12th International Conference on Extending Database Technology (EDBT'09)*, pages 1100–1103, Saint-Petersburg, Russia, 2009. ACM.

[69] `http://dblp.uni-trier.de/xml/`.

[70] `http://dev.mysql.com/doc/refman/5.0/en/fulltext-search.html`.

[71] `http://download.oracle.com/docs/cd/B28359_01/text.111/b28303/toc.htm`.

[72] `http://en.wikipedia.org/wiki/Cosine_similarity`.

[73] `http://en.wikipedia.org/wiki/Keyword_search`.

[74] `http://en.wikipedia.org/wiki/Vector_space_model`.

[75] `http://msdn.microsoft.com/en-us/library/ms142571.aspx`.

[76] `http://www.imdb.com/interfaces/`.

[77] `http://www.keyworddiscovery.com/keyword-stats.html`.

[78] Quang Hieu Vu, Beng Chin Ooi, Dimitris Papadias, and Anthony K. H. Tung. A graph method for keyword-based selection of the top-k databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD'08)*, pages 915–926, New York, NY, USA, 2008. ACM.

[79] Gerhard Weikum. DB&IR: both sides now. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD'07)*, pages 25–30, New York, NY, USA, 2007. ACM.

[80] Ji-Rong Wen, Jian-Yun Nie, and Hong-Jiang Zhang. Clustering user queries of a search engine. In *Proceedings of the 10th international conference on World Wide Web (WWW'01)*, pages 162–168, New York, NY, USA, 2001. ACM.

[81] Ryen W. White, Mikhail Bilenko, and Silviu Cucerzan. Studying the use of popular destinations to enhance web search interaction. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR'07)*, pages 159–166, New York, NY, USA, 2007. ACM.

[82] Ping Wu, Yannis Sismanis, and Berthold Reinwald. Towards keyword-driven analytical processing. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD'07)*, pages 617–628, New York, NY, USA, 2007. ACM.

[83] D. Xin, J. Han, X. Li, and B. W. Wah. Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. In *Proc. 2003 Int. Conf. on Very Large Data Bases (VLDB'02)*, pages 476–487, Berlin, Germany, Sept. 2003.

[84] Yabo Xu, Ke Wang, Benyu Zhang, and Zheng Chen. Privacy-enhancing personalized web search. In *Proceedings of the 16th International Conference on World Wide Web (WWW'07)*, pages 591–600. ACM, 2007.

[85] Bei Yu, Guoliang Li, Karen Sollins, and Anthony K. H. Tung. Effective keyword-based selection of relational databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD'07)*, pages 139–150, New York, NY, USA, 2007. ACM.

[86] Jeffrey Xu Yu, Lu Qin, and Lijun Chang. *Keyword Search in Databases.* Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010.

[87] Jeffrey Xu Yu, Lu Qin, and Lijun Chang. Keyword search in relational databases: A survey. *IEEE Data Eng. Bull.*, 33(1):67–78, 2010.

[88] Bin Zhou, Daxin Jiang, Jian Pei, and Hang Li. Olap on search logs: an infrastructure supporting data-driven applications in search engines. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'09)*, pages 1395–1404, Paris, France, 2009. ACM.

[89] Bin Zhou and Jian Pei. Answering aggregate keyword queries on relational databases using minimal group-bys. In *Proceedings of the 12th International Conference on Extending Database Technology (EDBT'09)*, pages 108–119, Saint-Petersburg, Russia, 2009. ACM.

[90] Bin Zhou and Jian Pei. Aggregate keyword search on large relational databases. *Knowledge and Information Systems: An International Journal (KAIS), to appear*, 2011.

[91] Xuan Zhou, Julien Gaugaz, Wolf-Tilo Balke, and Wolfgang Nejdl. Query relaxation using malleable schemas. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD'07)*, pages 545–556, New York, NY, USA, 2007. ACM.

[92] N. Ziviani, E. Silva de Moura, G. Navarro, and R. Baeza-Yates. Compression: A key for next generation text retrieval systems. *Computers*, 33(11):37–44, 2000.

[93] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 1(1):1–30, 1998.