Enhancing the Usability of Complex Structured Data by Supporting Keyword Searches

by

Ziyang Liu

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved May 2011 by the
Graduate Supervisory Committee:

Yi Chen, Chair
Kasim Candan
Hasan Davulcu
H.V. Jagadish

ARIZONA STATE UNIVERSITY

August 2011

ABSTRACT

As pointed out in the keynote speech by H. V. Jagadish in SIGMOD'07, and also commonly agreed in the database community, the usability of structured data by casual users is as important as the data management systems' functionalities. A major hardness of using structured data is the problem of easily retrieving information from them given a user's information needs. Learning and using a structured query language (e.g., SQL and XQuery) is overwhelmingly burdensome for most users, as not only are these languages sophisticated, but the users need to know the data schema. Keyword search provides us with opportunities to conveniently access structured data and potentially significantly enhances the usability of structured data. However, processing keyword search on structured data is challenging due to various types of ambiguities such as structural ambiguity (keyword queries have no structure), keyword ambiguity (the keywords may not be accurate), user preference ambiguity (the user may have implicit preferences that are not indicated in the query), as well as the efficiency challenges due to large search space.

This dissertation performs an expansive study on keyword search processing techniques as a gateway for users to access structured data and retrieve desired information. The key issues addressed include: (1) Resolving structural ambiguities in keyword queries by generating meaningful query results, which involves identifying relevant keyword matches, identifying return information, composing query results based on relevant matches and return information. (2) Resolving structural, keyword and user preference ambiguities through result analysis, including snippet generation, result differentiation, result clustering, result summarization/query expansion, etc. (3) Resolving the efficiency challenge in processing keyword search on structured data by utilizing and efficiently maintaining materialized views. These works deliver significant technical contributions towards building a full-fledged search engine for structured data.

ACKNOWLEDGEMENTS

I would like to sincerely thank those who made this dissertation possible. The first name undoubtedly goes to my advisor Yi Chen whose supervision and support during the past five years turned me from a college graduate with no research experience to a successful Ph.D. student with an abundance of publications. The time and effort she spent working with and advising me is quite remarkable. The other members of my Ph.D. committee, K. Selcuk Candan, Hasan Davulcu and H. V. Jagadish, have also generously spent time and effort to advise on my work, and I thank them for their contribution. It is also important to acknowledge the contribution of the co-authors of my publications, including Susan B. Davidson, Bin He, Hui-I Hsiao, Yu Huang, Qihong Shao, Peng Sun, Yichuan Cai, Sivaramakrishnan Natarajan, Stephen Booher, Tim Meehan, Jeffrey Walker and Robert Winkler. It has been a great pleasure to work with these people who made important contributions in my research projects. Finally, I'm very grateful to my parents and friends whose support and encouragement was invaluable during my Ph.D. study.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

Chapter 1

INTRODUCTION

1.1 Significance and Advantages of Keyword Search on Structured Data

Structured/Semi-structured data is a type of data that contains meta-data in addition to values. Meta-data may specify the tag/table/attribute/entity names in the data, as well as the connection among data items, e.g., key-foreign key relationships, ID/IDREF, etc. Meta-data may either be specified in a schema, or represented in the data itself. Typical structured/semi-structured data include relational data, XML, RDF, workflow, social network, etc., which have tree, graph or nested graph structures.

Typical ways of accessing structured data is using structured query languages, such as SQL, XPath, XQuery, SPARQL, etc. A structured query specifies a precise information need, and retrieves precise query results. However, in many applications structured data are accessed by non-expert or casual users such as Web search, online shopping, workflow retrieval, enterprise search, etc., and it is very difficult for casual users to issue a structured query for the following reasons.

First, structured query languages are difficult for casual users to learn and use. They are complex and it is easy to make mistakes when issuing structured queries. For example, suppose we have a database with a retailer table, a store table, a merchandise table and a sale table. In order to find the names and addresses of the Brooks Brothers stores that sell both outwear and shirt, the following SQL query is needed:

```
SELECT store.name, store.address from retailer r, store s, clothes c1, clothes
c2, sale s1, sale s2 WHERE r.rid = s.rid AND s.sid = s1.sid AND s.sid = s2.sid
AND s1.cid = c1.cid AND s2.cid = c2.cid AND c1.category = “outwear” and
c2.category = “shirt” AND r.name = “Brooks Brothers”
```

As we can see, this is a very long query in contrast to the simplicity of the information need; it is hard for many users to understand such a query and write it correctly.

Second, to issue a structured query, the user needs to understand the relevant part of the schema, i.e., the tables and their attributes, and how they are connected. However, the schemas of structured data can be complex, evolving or even unavailable. It is not uncommon for a database to have more than ten tables and some have even more (e.g., the database of an online shopping company can have hundreds of tables), and some tables may have many attributes (e.g., a camera may have more than 50 attributes). Understanding such schemas and posting structured queries against them is extremely difficult. The schema may also be evolving, e.g., new attributes may be added, a schema may be normalized or denormalized, etc., and previous queries may no longer work. Moreover, for semi-structured data such as XML, schemas may not be available at all, and issuing structured queries is especially difficult in this case. As a result, the usability of structured data is very limited if structured query is the only way of accessing the data.

Another way of accessing structured data is using query forms or specialized applications. However, they are still not suitable for all users for several reasons. First, even for a medium-size schema, the possible number of query forms are very large, and it is hard to decide which one we should use to satisfy a user's needs. Second, users are often unwilling to do such advanced searches and prefer simple keyword queries. Third, specialized applications are usually costly to develop and inflexible to use.

On the other hand, the success of Web search engines indicate that keyword search is a highly attractive way to access data for casual users. With the observation of the disadvantages of structured query languages/query forms, a natural question is whether we can support keyword searches on structured data. The most important advantage of supporting keyword search on structured data is the ease to issue keyword queries, which is a critical factor for the large population of casual users. For example, for a user who wants to find the names and addresses of the Brooks Brothers stores that sell both outwear and shirt, it will be desirable if s/he can simply use a keyword query "Brooks Brothers, outwear, shirt, name, address". As we can see, keyword search frees the user from the burden of learning the query language and understanding the data schema.

Besides the ease to use, there are several other advantages of supporting keyword searches on structured data. First, compared with using structured query languages, searching structured data using keywords gives users the opportunity of discovering interesting/unexpected information. For example, a user who wants to check whether Stuart is an employee at Brooks Brothers may issue a keyword query “Stuart, Brooks Brothers”. In addition to returning this information, the search engine may additionally return a result showing that Paul Stuart is a competitor of Brooks Brothers, which could be an interesting and useful piece of information to the user. This would not be possible if the user used a structured query, since the structured query needs to precisely specify the user’s intention (i.e., whether Stuart is an employee at Brooks Brothers).

Second, compared to supporting keyword search on text documents, supporting keyword search on structured data is advantageous in that structured data has richer semantics than text documents, which gives us better opportunities to generate high quality results. For example, consider the following text document fragment: “*John is a an employee at Brooks Brothers.......... One of John’s colleagues, Mary, recently published a book about clothing design.*” In this fragment, terms “John” and “clothing design” are close to each other in terms of occurrences. Thus if we post a query “John, clothing design”, this document may be returned with high ranking score. However, the person named John and the book about clothing design in this document are in fact not closely related. On the other hand, if we have a structured data, the search engine will be able to determine that John and clothing design are not closely related since the shortest path between them is long (e.g., John $\leftarrow$ name $\leftarrow$ employee $\leftarrow$ employees $\rightarrow$ employee $\rightarrow$ publications $\rightarrow$ book $\rightarrow$ title $\rightarrow$ clothing design), and it indicates that they belong to two different employees.

Due to these advantages of supporting keyword search on structured data, this dissertation studies the topic of enhancing the usability of structured data by supporting keyword searches. Although keyword search on structured data has the opportunities to generate high quality results and is highly beneficial to the users, doing so is very challenging due to the ambiguity of keyword searches and the large search space. In the next two subsections, we summarize the challenges involved in supporting keyword search on structured data, and our techniques and contributions.

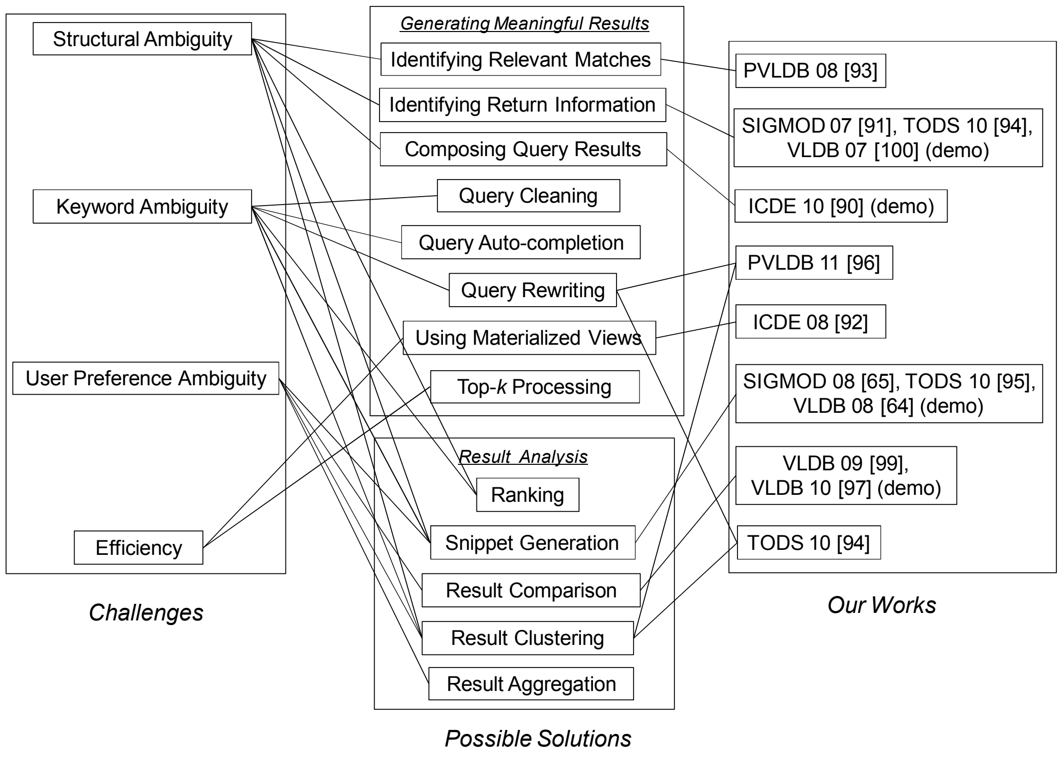


Figure 1.1: Challenges and Solutions for Keyword Search on Structured Data

1.2 Challenges and Solutions of Keyword Search on Structured Data

There are many challenges of supporting keyword search on structured data, which can be summarized into two categories: *ambiguity*, and *efficiency*. Ambiguity can be further divided into three types: *structural ambiguity*, *keyword ambiguity* and *user preference ambiguity*. Figure 1.1 shows the detailed challenges of each category and the solution for each type of challenges. The left box contains the four types of challenges, the middle box contains the potential solutions and the right box contains our published works (other state-of-the-art techniques will be reviewed in Chapter 3). Their relationships are indicated by lines.

Structural Ambiguity. In structured queries, the structure of the result, i.e., the relevant tuples or nodes as well as their connections, are precisely specified. However, they are not specified in keyword searches. Therefore, they need to be inferred by the search engine.

Specifically, we need to infer three types of information: which keyword occurrences are relevant, what other nodes are relevant, how to connect the relevant nodes. For example, for the SQL discussed in Section 1.1, the relevant nodes are the tuples in *merchandise* table whose categories are *outwear* or *shirt*, and the tuples in the *retailer* table whose names are *Brooks Brothers*; the return information is the name and address of the relevant stores; and the connection of nodes is specified by the join conditions, as observed in [91, 90]. If the user uses keyword search, she may issue a query "Brooks Brothers, name, address, outwear, shirt", which specifies none of the three types of information. Therefore, keyword search engines need to infer them automatically.

It is not always possible to completely resolve structural ambiguity and return the perfect results. In other words, there may be irrelevant results despite the effort of resolving structural ambiguity. Therefore, it is desirable to help users analyze the results. For example, structural ambiguity can be alleviated by ranking the query results [89, 76, 102, 60, 122, 55, 84, 48, 70, 60, 62, 21, 50, 75, 115], clustering the query results based on their structures [94], generating result snippets which summarize the structures of the results [65, 95], etc.

Structural ambiguity is a unique challenge of keyword search on structured data. For structured query languages, the user specifies the precise structures, thus there is no structural ambiguity. For text search, since text documents do not have structures, this challenges does not apply either.

Keyword Ambiguity. Keyword ambiguity is due to the imprecise keywords the user may use. There are four types of keyword impreciseness.

Type 1: Misspelled or Unfinished Words. For example, a user searching for "Brooks Brothers" may write "Broks Bro" as the query where "Broks" is a misspelled word and "Bro" is unfinished. Besides, the data searched by the user may also contain misspelled words. For this type of imprecise keywords, the search engine needs to perform query cleaning [101, 113] and query auto-completion [83, 31] before processing the query in order to avoid generating empty results.

Type 2: Under-specified Words. Under-specified words either have multiple meanings, or are too general. In either case, it may lead to a large number of irrelevant results. For example, “Columbia” is a multi-meaning words, which may refer to city, university, sportswear company, etc. Thus query “Columbia” will retrieve results related to all these objects, whereas the user may be only interested in one of them. As another example, “men’s apparel” is a very general query which may retrieve a large number of results. In this case, the search engine should help the user refine the query to narrow down the search scope [76, 96]. For example, for “Columbia”, the search engine may suggest queries like “Columbia University”, “Columbia sportswear”, etc.

Type 3: Over-specified Words. Over-specified words are too specific. Their relevant results not only include those containing these words, but also those that do not, i.e., some relevant results will be missed if no action is taken. There are two types of over-specified words: (1) Words with synonyms. For example, consider query “men, apparel, store”. Word “apparel” is over-specified since it has synonyms like “clothes”. In this case, results that contain “clothes” but not “apparel” are not retrieved, but they are potentially relevant; (2) Queries with too specific restrictions. For example, query “Brooks Brothers clothes for men with price 190-200 USD” may not retrieve any result due to the constraints (190-200). In this case, the search engine needs to perform query rewriting in order to retrieve results that do not contain the query keywords, e.g., find the synonyms of query keywords [138, 37, 106].

Type 4: Non-quantitative words. Queries containing non-quantitative words may both miss relevant results, and retrieve irrelevant results. For example, query “expensive clothes” is non-quantitative since it is unclear what “expensive” means. Clothes with high prices may not have the word “expensive”, while low price clothes may have “expensive” in their descriptions (e.g., “this clothes is as good as a much more expensive one”). In this case, the search engine needs to perform query rewriting, e.g., map “expensive” to “price > 500USD”, to improve the search quality [138].

In summary, there are several types of approaches to address keyword ambiguity, which are based on different inputs such as query and click log [37], historical query results [138], current query results [76, 96], or just the data itself [101, 113, 83, 31].

Note that keyword ambiguity also applies to both structured queries and text search. For structured queries, this problem is not so significant as users of structured queries are usually database experts and typically are able to use the precise words. For text search, this challenge applies and some techniques for addressing this challenge can be reused. However, it is worth noticing the two differences. First, the query result definition on structured data is usually different from that on text documents, which leads to different techniques for resolving keyword ambiguity. For example, when we correct the misspelled words in a keyword query, intuitively, we want the cleaned query to have good results [101], thus the query cleaning method should depend on how results are defined and ranked. Second, structured data provides more opportunities for resolving keyword ambiguity. For example, by analyzing the distribution of attribute values in the results of a query, we can map a possibly over-specified or non-quantitative words to an SQL predicate to improve the search quality [138].

User Preference Ambiguity. Even if the query results are highly relevant to the user's search intention, it is still not the end of the story. Many keyword queries are for information exploration purposes, where the user may not have a clear idea of what s/he wants. Such queries are commonly seen in applications like online shopping, job hunting, employee hiring, etc. For example, for query "men, apparel, store", each result is a store selling men's apparel. Even if all results are relevant to the query, the user may need to check multiple results to determine a few stores to visit. To help users check results quickly, we can generate a snippet for each result [65, 95], and generate a concise comparison table for the user selected results that highlights their key differences [99]. We can also identify interesting aggregation attributes and group the results by these attributes to give the user useful insights of the results. The grouping can be performed in a hierarchical manner to form a navigation tree, typically used in faceted search [71, 29, 81, 34].

Similar as keyword ambiguity, user preference ambiguity also applies to both structured queries and text search. For structured queries, this problem is not so severe as users of structured queries are usually database experts. Compared with text search, structured search results provide much better opportunities for resolving user preference ambiguity.

For example, we can identify *features* from structured search results and generate a table that compares different features in multiple results [99].

Efficiency of Result Generation. Generating results efficiently is very important for a keyword search engine as the users are usually impatient and expect to see the results in a few seconds. Generating keyword search results on structured data efficiently involves several unique challenges compared with structured query and text search. Compared to structured query, since the relevant nodes, their connections and the return information are not specified in keyword queries, the search space of keyword queries can be much larger. Compared to text search, the results of keyword search on structured data are not individual documents, but rather subtrees or subgraphs of the data that need to be dynamically identified. Given a structured data and keyword query, the number of ways to connect nodes matching query keywords can be far bigger compared with the size of the data. Therefore, it is more challenging to generate query results in ranked order. In fact, even finding the top-1 result using a simple ranking function where results are ranked by their sizes, is an NP-hard problem (i.e., the group Steiner tree problem).

In the next subsection we briefly introduce our techniques to address these challenges as shown in the right box of Figure 1.1.

1.3 Overview of Our Techniques

In this dissertation we present a set of techniques that addresses some of the challenges above. Chapters 4 - 12 focus on the challenges in result generation and analysis on tree and/or graph data. Chapter 13 discusses how to define search result on a non-traditional data model: nested graphs, which have three dimensional structures. Nested graphs are useful for modeling workflow hierarchies, which are prevalent in scientific and business domains.

To illustrate our techniques, consider a sample query "*men, apparel, store*" on the XML data in Figure 1.3. Intuitively, the user wants to find the stores that sell men's apparels. A possible XQuery to find such stores is shown in Figure 1.4. As we can see, this XQuery specifies which nodes are relevant in the "for" clause (e.g., only those "store" nodes whose

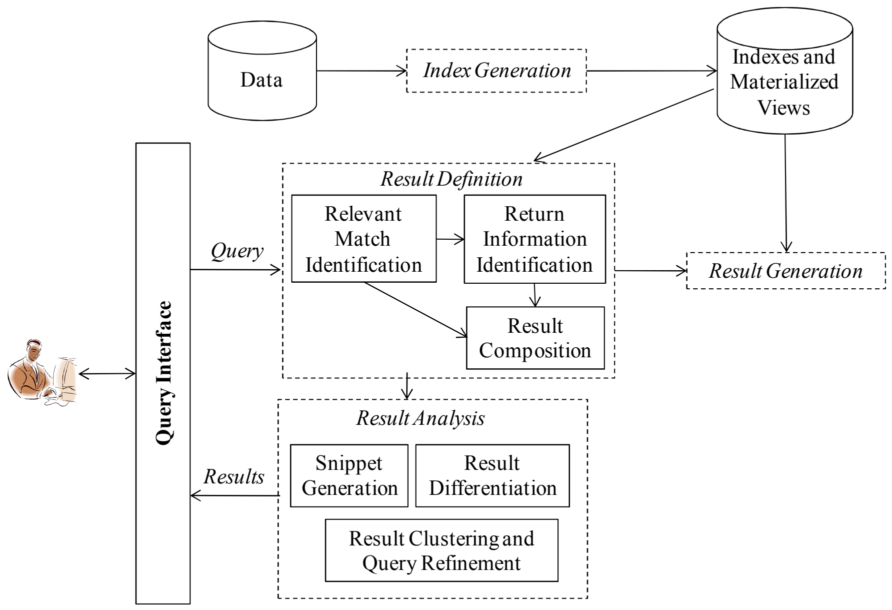


Figure 1.2: Interaction of Proposed Techniques

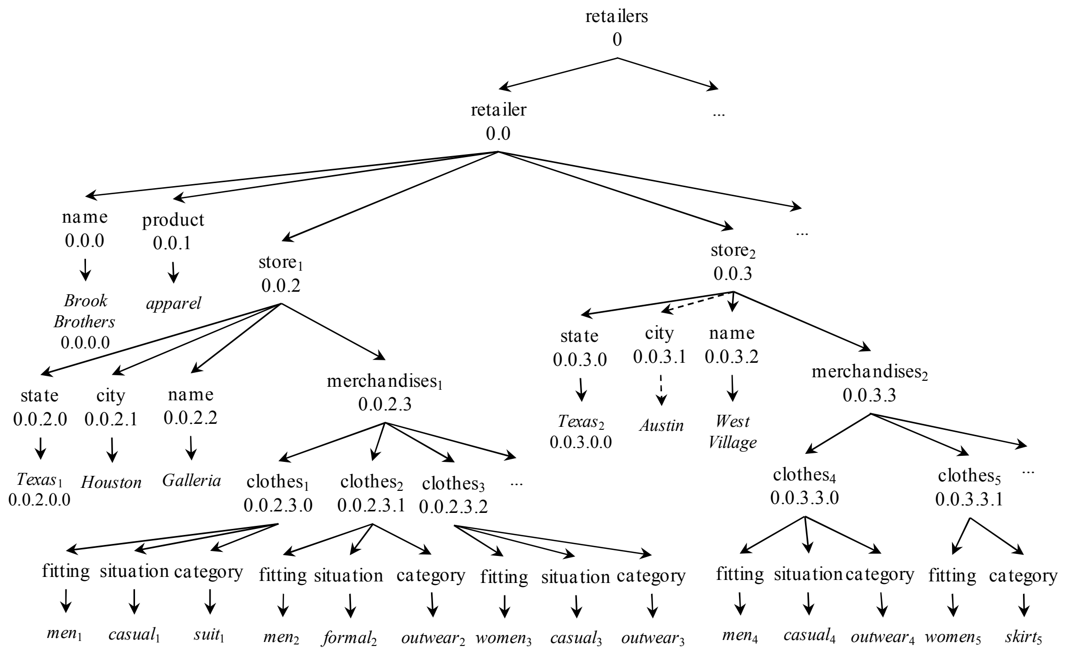


Figure 1.3: Sample XML Document

retailers' product is apparel and who have men's clothes), what is the return information in the "return" clause (e.g., retailer name, retailer product, and the subtree of store) as well as how the returned nodes are connected (e.g., each result contains exactly one store, rather than multiple stores). Since the keyword query has no such information, we first need

```
for $r in doc("retailers.xml")//retailer[./product = "apparel"]
for $s in $r/store[.//clothes/fitting = "men"]
return <result> $r/name, $r/product, $s</result>
```

Figure 1.4: A Possible XQuery for Keyword Search "*men, apparel, retailer*"

to identify them automatically, and define the query results accordingly. After the results are generated, we also proposed various approaches, such as generating result snippets, generating comparison tables, clustering results, generating expanded queries, etc., to help users analyze the results and resolve structural, keyword and user preference ambiguities.

Identifying Relevant Keyword Matches and Reasoning about Effectiveness

Each keyword may have multiple matches in the XML document, and not all of them are necessarily relevant to the query. For query "*men, apparel, store*", there are multiple matches to keyword "store". Both "store" nodes in Figure 1.3 are relevant to the query; however, if a store does not sell men's apparel, then such a "store" node is not relevant. From the XQuery in Figure 1.4, we can clearly see that only stores that sell men's clothes that belong to an apparel retailer will be returned, and other *store* nodes are irrelevant.

In order to automatically identify relevant keyword matches, there are several different systems that use different underlying principles and heuristics, leading to different query results in general. A natural question is how to guide the design and to evaluate strategies for identifying relevant matches. Due to the inherent ambiguity of search semantics, it is hard (if not impossible) to directly assess the relevance of keyword matches and reason about various strategies.

Interestingly, we discover that by examining query results produced by the same approach on similar queries or on similar documents, sometimes abnormal behaviors can be clearly observed, which exhibit the pitfalls that a good approach should avoid. From these observations and analysis, we initiate an investigation of a formal axiomatic framework to express valid changes to a query result upon a change to the user query or to the data. After reviewing the existing strategies on identifying relevant matches using the axiomatic framework, we find that, surprisingly, none of them satisfies all these properties.

We then design a novel algorithm, MaxMatch, which possesses all these properties and efficiently identifies relevant matches. To the best of our knowledge, this is the first work that reasons about keyword search strategies from a formal perspective, and is the first approach of identifying relevant matches that satisfies all properties in the proposed axiomatic framework.

The analysis of existing approaches with respect to the proposed features and the detailed techniques of MaxMatch have been published at VLDB 08 [93], and will be presented in Chapter 4.

Identifying Relevant Return Information

As discussed before, unlike a structured query where the return nodes are specified using either a "return" clause (in XQuery) or a "select" clause (in SQL), we should effectively identify the desired return information. In other words, besides the relevant keyword matches and the paths connecting them, some other nodes in the XML data which are not keyword matches but are relevant to the query should also be returned.

For example, consider the keyword query "*men, apparel, store*", for which a candidate XQuery is shown in Figure 1.4. By issuing this query, the user is likely interested in information about the stores who sells men's apparel, as can be observed from the "return" clause in the XQuery. Therefore, for this keyword query, the information of the store and its clothes, such as store's name and location, should be output, even if they do not match the query keywords.

As we can see, a query keyword can specify a *predicate* for the search, or specify a desired *return* node. For example, in the above query, "*men*" and "*apparel*" are predicates (analogous to the "for" and "where" clauses in the XQuery in Figure 1.4) and "*store*" is a return node (analogous to the *return* clause in the XQuery). Existing approaches fail to effectively identify relevant return nodes, which may lead to low search quality.

In Chapter 5, we present techniques that identify meaningful return nodes for keyword search on XML without user solicitation. To achieve this, we analyze both XML data structure and keyword patterns. We differentiate three types of information represented in

XML data: entities in the real world, attributes of entities, and connection nodes. We also categorize input keywords into two types: the ones that specify search predicates, and the ones that indicate return information that the user is seeking. Based on data and keyword analysis, we discuss how to generate return nodes, which can be explicitly inferred from keywords, or dynamically constructed according to the entities in the data that are relevant to the search. Finally, data nodes that match predicates and return nodes are output as query results with optional expansion links. This work has been published at SIGMOD ’07 [91] and TODS ’10 [94], and has been demonstrated at VLDB ’07 [100].

To the best of our knowledge, this is the first work that addresses the problem of automatically inferring desirable return nodes for XML keyword search. It can be combined with any method for identifying relevant keyword matches.

Composing Results Based on Relevant Nodes

After identifying relevant matches and return information , we need to compose meaningful query results which can be effectively ranked to bring good search experience to the users. As can be seen from the following example, the way of composing results in XML keyword search has crucial effects on result ranking.

Consider the running example “*men, apparel, store*” on the XML tree in Figure 1.3. After identifying relevant matches (i.e., the match nodes relevant to keywords “men”, “apparel”, and “store”) as well as the relevant return information (the information related to retailer, store and clothes), there are three ways to compose query results:

(1) Each result contains one instance of *store*, which corresponds to the XQuery in Figure 1.4. The fragments of two results of this query are shown in Figure 1.5(a).

(2) Each result contains one instance of *retailer* together with all its stores, which corresponds to the following XQuery:

for $r in doc(“retailers.xml”)//retailer[./product = “apparel”]

return <retailer> {$r/name, $r/product

for $s in $r/store[.//clothes/fitting = “men”]

return $s} </retailer>

(3) Each result contains one instance of *clothes*, which corresponds to the following XQuery:

for $r in doc("retailers.xml")//retailer[./product = "apparel"]

for $s in $r/store[.//clothes/fitting = "men"]

for $c in $s/clothes

return <result> {$r/name, $r/product} <store> {$s/state, $s/city, $s/name $c} </store> </result>

These three options return exactly the same information in the set of results; however, the results are composed based on different entities (store, retailer and clothes). Intuitively, (1) is the most desirable way to compose results, as the *search target* of this query is likely *store*. Since each result has a single instance of *store* with all its supporting information (i.e., all clothes), each *store* instance can be correctly gauged by the ranking scheme and different stores can be meaningfully ranked. On the other hand, for option (2), since each result has many instances of *store*, ranking is not performed on target instances. For option (3), the same *store* may be separated into many results, and it is difficult for the user to find the information of a specific store.

As we can see, each keyword search has a goal, which is usually the information of a real world entity or relationship among entities, as observed in [36, 38]. We use the term *search target* to refer to the information that the user is looking for in a query, and *target instance* to denote each instance of the search target in the data. Each desirable query result should have *exactly one target instance* along with all associated evidence, so that ranking and top-k query processing can be based on target instances, and thus become meaningful. Specifically, query results of an XML keyword search should be: (1) Atomic: it should consist of a single target instance; (2) Intact: it should contain the whole target instance as well as all its supporting information. Atomicity ensures that ranking can be performed on target instances; and intactness ensures that the ranking score of each target instance can be correctly gauged by the ranking function.

In Chapter 6 we propose a novel technique to automatically compose atomic and intact query results for XML keyword searches, such that each result contains exactly one search target instance along with all its evidence, as illustrated in the above examples. Unlike the existing approaches, which are oblivious to users' search intentions, the proposed query result composition is *driven by the user search target* and hence *ranking friendly*. This approach has been demonstrated at ICDE '10 [90]. This is, to the best of our knowledge, the first work that composes ranking-aware XML keyword search results.

Generating Query-Biased Result Snippets

So far we have summarized our techniques for resolving structural ambiguity by generating meaningful results. However, it is very difficult, if not impossible, to generate perfect results for every query on every data for every user. In the next few subsections we summarize our techniques for resolving structural, keyword and user preference ambiguity from another perspective: helping the user analyze the query results after they are generated.

Queries issued by web or scientific users, including both keyword and structured queries, may often return a large number of results. Various ranking schemes have been proposed to assess the relevance of query results so that users can focus on the ones that are deemed highly relevant. However, due to the ambiguity of search semantics, it is impossible to design a ranking scheme that always perfectly gauges query result relevance with respect to users' intentions. To compensate the inaccuracy of ranking functions, result snippets are used by almost every web search engine, and are also useful for structured search results.

As an example, consider query "*men, apparel, store*" on the XML data in Figure 1.3. Two sample results of this query are shown in Figure 1.5(a). Some statistics of full query results are presented next to the result. Their snippets are shown in Figure 1.5(b). They capture the essence of each query result in a small tree. For example, result 1 is about store *Galleria* in *Houston*, which mainly features *outwears* and *shirts* for *men*.

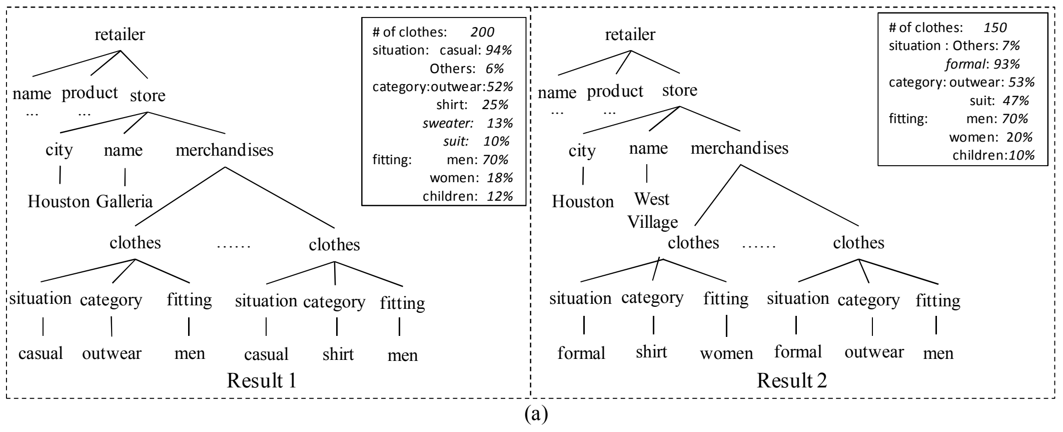
In Chapter 7 we address the problem of generating effective snippets for structured search results, and comprehensively test the snippets generated by our approach in terms of quality, efficiency and scalability. We identify that a good structured result snippet should be an information unit of *a bounded size* that effectively *summarizes* the query result. To achieve this, we first analyze the semantics of the query result. We identify the key of a query result as well as the prominent features, based on which a snippet information list is generated. Then we need to select as many items in the information list as possible given an upper bound of the snippet size. We show that this problem is NP-hard. A novel algorithm is proposed that efficiently generates result snippets with a given size bound. This is the first work that studies how to generate result snippets for structured search results.

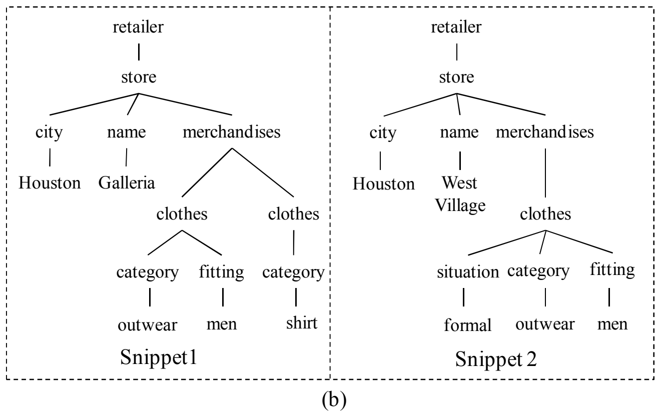
Additionally, good snippets can also be used for efficient result clustering. Since efficiency of result clustering is highly important, a better solution for grouping structured keyword query results is to use a small summary of each result, rather than the results themselves. Therefore, it is a natural idea to use the snippets, instead of the results themselves, for grouping the query results in order to get better efficiency. We have conducted a set of experiments in Chapter 10 on the quality and efficiency of result clustering using snippets compared with using results. The experiments suggest that the processing time using snippets is much faster compared with the processing time using query results. Meanwhile, using snippets do not compromise much quality of clustering. Therefore, clustering structured search results using snippets can be good alternatives when the efficiency of clustering is important. This work has been published at SIGMOD '08 [65], TODS '10 [95] and demonstrated at VLDB '08 [64].

Search Result Differentiation

For many queries, even if the results are highly relevant, the user would still like to investigate, evaluate, compare, and synthesize multiple relevant results for information discovery and decision making. These queries are referred to as *exploration queries*, in contrast to *navigational queries* whose intentions are to reach a particular website. As discussed before, due to the existence of exploration queries, there exists user preference challenge in supporting keyword search on structured data.

For example, consider a customer who issues a keyword query "*men, apparel, store*". There are many results returned, where the fragments of two results are shown in Figure 1.5(a) and some statistics information of the results is shown next to the results. As





Feature Type	DFS of Result 1 (D_1)	DFS of Result 2 (D_2)
store: name	Galleria 100%	West Village 100%
clothes: category	outwear 52% shirt 25% sweater 13%	outwear 53% suit 47%
clothes: situation	casual 94%	formal 93%

(c)

Figure 1.5: Two Results of Query "*men, apparel, store*" (a), Their Snippets (b) and Differentiation Feature Sets with Size Limit = 5 (c)

each store sells hundreds of clothes, it is very difficult for users to manually check each result, compare and analyze these results to decide which stores to visit.

Figure 1.5(b) shows the snippets of results in Figure 1.5(a) generated by our snippet generation approach [65, 95]. These snippets highlight the *most dominant* features in the results. However, snippets are generally *not comparable*. From their snippets, we know result 2 focuses on *formal* clothes, but have no idea whether or not result 1 focuses on *formal* or *casual*, since the information about the store, specifically *situation*, is missing in its snippet due to space limitation. Similarly, result 1 has many *shirts*, but we do not have information about whether result 2 has many *shirts* or not. As we can see, snippets are not designed to help users find out the differences among multiple results.

In Chapter 8, we present the techniques for structured data search result comparison and differentiation, which takes as input a set of structured results, and outputs a

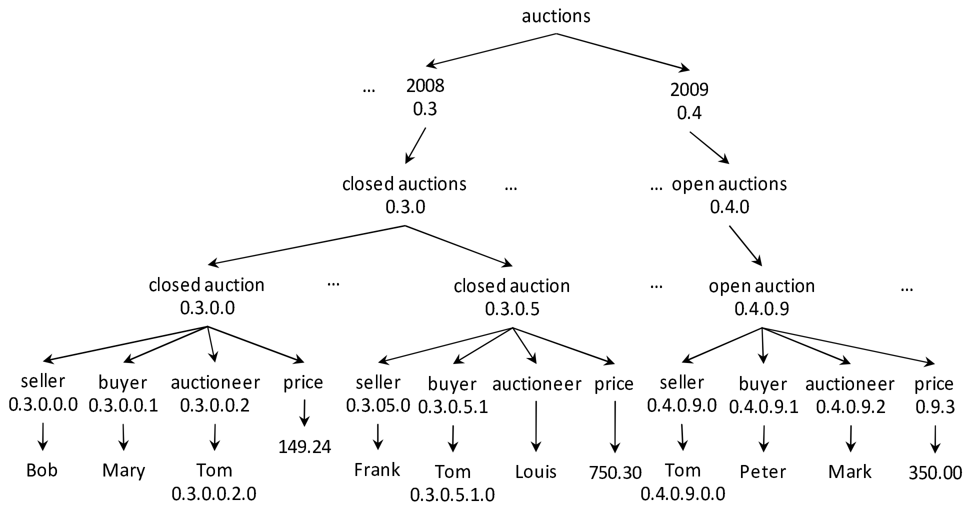


Figure 1.6: XML Data about Auctions

Differentiation Feature Set (DFS) for each result to highlight their differences within a size bound. To show the usefulness of our techniques in the real world, we develop a structured search result differentiation method named *XRed*, and use both real and synthetic data to evaluate our algorithms in experiments. We identify three desiderata of generating DFSs, propose an objective function, and prove the that it is NP-hard to satisfy the objective function. We then design four efficient algorithms for DFS generation.

The XRed method can take the results generated by any of the existing keyword search engines on structured data as the input and generate DFSs for result differentiation. In fact, the generated DFSs can also be used to compare results of structured query (e.g., XPath, XQuery, SQL) upon user request. Sample DFSs for the query results in Figure 1.5(a) are shown in Figure 1.5(c). Our approach has been published at VLDB '09 [99] and demonstrated at VLDB '10 [97]. To the best of our knowledge this is the first study on structured search result differentiation.

Structure Based Result Clustering and Query Expansion

We have discussed the importance of identifying return information in Section 5.1. However, it is very challenging to automatically determine return nodes with minimal user feedback, especially for ambiguous queries, such as query "*auction, seller, buyer, Tom*" on the XML data about auctions in Figure 1.6. There are many valid interpretations of the query semantics, such as: (i) find an auction whose buyer is Tom; (ii) find an auction whose seller is Tom; and (iii) find an auction whose auctioneer is Tom. Though an answer to any above query semantics could be relevant to the user, returning all such answers at the same time is problematic. It is extremely hard or even impossible for a system to automatically determine which semantics are desired by the user, and thus should be given a higher rank. Therefore in case of query ambiguity, instead of displaying a mixture of query results with different semantics, it is desirable to cluster query results such that the results with the same or similar query semantics are clustered together, with a user controlled clustering granularity.

To help users quickly find the most relevant results for ambiguous queries whose results have multiple types, we propose in Chapter 9 a novel technique which utilizes the return information inferences to be discussed in Chapter 5 to produce clusters with describable query semantics in a user specified granularity. Compared with existing studies on structured data clustering, there are two salient features of our approach. First, it is query-aware, considering both query semantics and data structure/content for clustering. Second, the semantics of each cluster is guaranteed to be describable, which helps users prune irrelevant results.

We further discuss that, when the user specifies a desired number of clusters, k, how to further split the clusters and make the number of clusters as close to k as possible. This problem is shown to be NP-hard and we propose an efficient dynamic programming method to address it. This is the first attempt on query-aware result clustering for keyword search on structured data. This approach has been published at TODS '10 [94].

Value Based Result Clustering and Query Expansion

The approach discussed previously is designed to cluster the results based on the structure of the result, specifically, keyword roles and the paths from result root to each keyword match. In other words, it addresses structural ambiguity. In fact, clustering can also be used to resolve keyword ambiguity. For example, for query "*auction, seller, buyer, Tom*", the user may want to find a specific set of auctions about cars. In this case, the query is under-specified, and clustering the results based on values in the results (e.g., description of each auction) can be helpful. In this example, after the previous approach clusters the results according to keyword context, we can further divide the clusters according to information other than keyword context, and generate expanded queries like “auction, car”, “auction, TV”, etc. We would like to find a generic way of generating expanded queries given a set of clusters, where the clusters can be obtained using an existing approach. This approach can be used to complement the structured based clustering method discussed before to produce clusters and expanded queries with finer granularity.

Given a set of clusters of query results, the challenge is how to generate an expanded query for each cluster, whose set of results is as close to the cluster as possible. In other words, if we consider a cluster of results as the ground truth, our goal is then to generate a query whose set of results achieve both a high precision and a high recall. This is a difficult problem as the expanded queries should not only be selective to eliminate as many results in other clusters as possible (maximizing the precision), but also be general to retrieve as many results in this cluster as possible (maximizing the recall).

To tackle the challenges, in Chapter 11, we study the problem of how to generate a set of expanded queries given a set of result clusters, which is independent of the clustering method. We formally define the problem of generating optimal queries given the ground truth of the query results and prove the APX-hardness of the problem. We design two algorithms which generate meaningful expanded queries efficiently given result clusters. This approach has been published at PVLDB ’11 [96].

Efficient Result Generation Using Materialized Views and View Maintenance

After results are defined, the next step is to generate the query results. Since the efficiency of a search engine is highly important, we study the problem of how to speed up query result generation on XML using materialized views and how to maintain the views.

Materialized views have been proved successful for performance optimization in evaluating structured queries on databases [16, 110, 14, 129, 103, 143]. They have also been widely used in web applications. Given the benefits of materialized views in structured query processing, it is a natural idea to explore them in the context of XML keyword search. We study two related problems in this dissertation: (1) How to exploit materialized views for query evaluation? (2) How to incrementally maintain materialized views?

To address these questions, we present in Chapter 12 a general framework for exploiting materialized views for XML keyword search. We first identify the relevant materialized views that potentially can be used to answer a user query. We prove that given a set of relevant materialized views and a keyword query, the decision problem of finding the minimal answer set is NP-hard.

We then design and implement an XML keyword search engine that can answer queries using materialized views. A polynomial time approximation algorithm is proposed that finds the optimal answer set of materialized views to evaluate the input query. We also designed algorithms to answer the keyword query given the answer set. In order to keep the materialized views fresh, they are incrementally maintained upon XML data insertion or deletion. The realization of these functionalities depends on how query results are defined. Our keyword search engine adopts a commonly used semantics to define query results for XML keyword search, the SLCA semantics. Experiments show significant performance improvements in the efficiency of answering queries using views and maintaining views incrementally. Our study of exploiting materialized views for evaluating XML keyword searches was published as a poster paper ICDE '08 [92].

Searching Workflow Hierarchies

Besides tree-structured data (such as XML) and graph-structured data (such as relational data) which are two types of commonly used structured data, another important type of structured data is the nested graph. It is widely used to model workflow hierarchies, used in scientific [17, 59, 123, 134, 22, 127] and business [20, 134] domains. As an example, a workflow hierarchy describing the recipe of curry chicken is shown in Fig. 1.7.[1] As we can see, a workflow hierarchy is a three dimensional data structure. It can be considered as an extension of trees: each group of sibling is no longer a set of nodes, but a graph.

[1] Every node in the figure is associated with an identifier, which will be presented in Chapter 13.

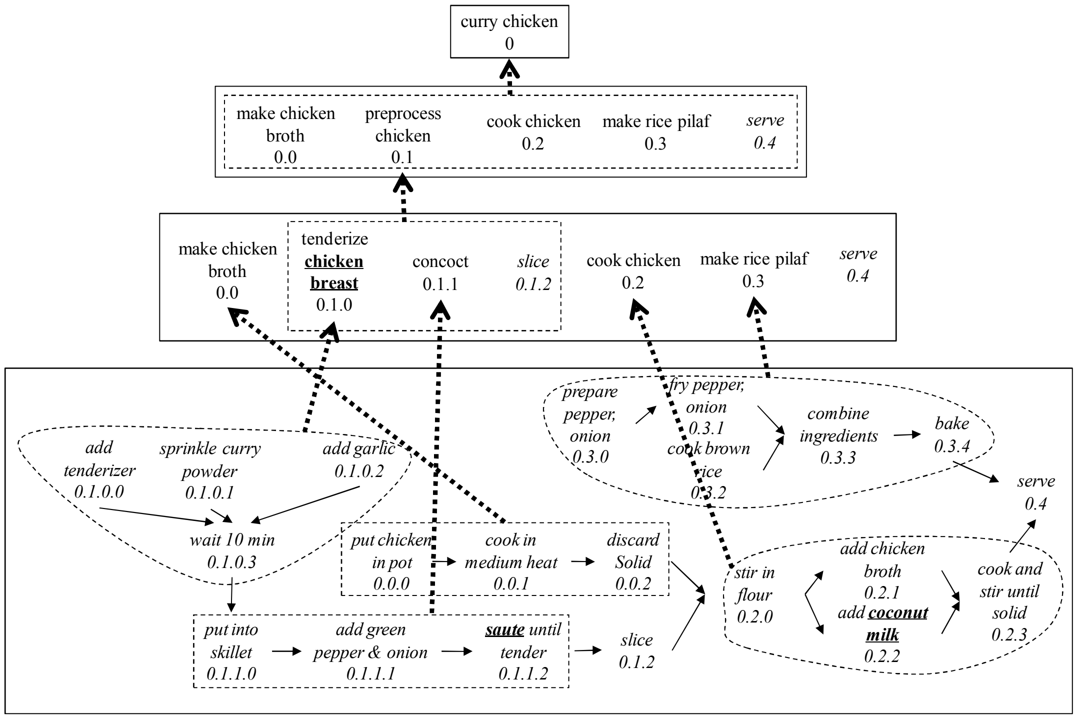


Figure 1.7: A Workflow Hierarchy Describing a Recipe

A node represents a task, which can be a step in a recipe, a web service invocation, a database query, a program run, or an experiment step, etc. A directed solid edge between nodes represents their dependency, dataflows, or control flows (AND/OR/XOR), referred as *dataflow edges*. A dotted edge connects a *composite task* to a group of tasks based on which means the composite task is an abstraction of the group of tasks.

Due to the existence of two types of edges in a nested graph, existing techniques for keyword search on general graphs [55, 72, 102, 104, 48, 50, 70, 12, 60, 62, 21, 122, 89] cannot be applied to searching workflow hierarchies due to two reasons. First, a path between two nodes, e.g., "*tenderize chicken breast - preprocess chicken - cook chicken - add coconut milk*", in a workflow hierarchy does not necessarily capture their dataflow. Second, returning the smallest tree connecting the query keywords does not give a self-contained workflow, and is not informative or meaningful. Therefore, different techniques are needed to resolve the structural ambiguity challenge for processing keyword search on nested graphs.

In Chapter 13, we present WISE, a Workflow Information keyword Search Engine.

Our approach generates informative (capturing the keyword matches and their dataflows), self-contained (having a name/goal) and concise query results for keyword queries on workflow hierarchies using an efficient algorithm. To achieve this, we start with identifying the minimal relevant workflow hierarchies, and define the concept of a *view* of a workflow hierarchy, which can be considered as a projection of the 3D workflow hierarchy onto a 2D plane which hides less important information. We then propose to identify the minimal view of a minimal workflow hierarchy that satisfies the query as the result. We designed an algorithm for identifying minimal views with optimal time complexity. WISE has been published at PVLDB '10 [98] and demonstrated at ICDE '09 [125].

The remaining of this dissertation is organized as follows. Chapter 2 introduces data and query models. Chapter 3 discusses related works. Our proposed techniques and experimental evaluations are introduced in Chapters 4 - 13. We conclude the dissertation and discuss future works in Chapter 14.

Chapter 2

PRELIMINARIES

2.1 Data Model

In this dissertation, we consider three types of structured data: structured data that can be modeled as a tree (such as XML without ID/IDREF), structured data that can be modeled as a graph (such as XML with ID/IDREF, RDF, relational database, social network data, etc.), as well as structured data that can be modeled as a nested graph (such as workflow hierarchies). Next, we introduce these three data models.

Tree Data Model

A tree data model is a connected, acyclic graph that consists of a set of nodes and edges. Node u is the parent of node v if there is an edge from u to v, and inversely, v is a child of u. u is an ancestor of v if there is a directed path from u to v, and inversely, v is a descendant of u. Each node (except the root) has a single parent, and each node is a descendant-or-self of the root. A node can be labeled with a set of keywords. Each node and edge may have a weight.

A most commonly used tree-structured data is XML. An XML document (without ID/IDREF) can be naturally modeled as a tree: the root node represents the root element. Each element or attribute name is represented by an internal node, and each text value or attribute value is represented by a leaf node, such as the one shown in Figure 1.3.

Labeling the tree nodes is important for performing operations on trees such as computing the lowest common ancestor of two nodes. In this dissertation we use the *Dewey* labeling scheme [133] to label tree data. Each node has a Dewey label as a unique ID. We record the relative position of a node among its siblings, and then concatenate these positions using dot '.' starting from the root to compose the Dewey ID for the node. For example, node with Dewey ID 0.2.3 is the 4-th child of its parent node 0.2. Dewey ID can be used to detect the order, sibling and ancestor information of a node.

1. The start tag of a node u appears before that of node v in an XML document if and

only if Dewey(u) is smaller than Dewey(v) by comparing each component in order.

2. A node u is an ancestor of node v if and only if Dewey(u) is a prefix of Dewey(v).
3. A node u is a sibling of node v if and only if Dewey(u) differs from Dewey(v) only in the last component.

We can also infer that given two nodes u and v, their lowest common ancestor (LCA) has a Dewey ID that is equal to the longest common prefix of Dewey(u) and Dewey(v).

Graph Data Model

A graph data model consists of a set of nodes and edges. Each edge connects two nodes. A node can be labeled with a set of keywords. Each node and edge may have a weight.

The graph data model can be used to model a lot of data types people commonly use. For example, XML with ID/IDREF can be modeled as a graph, which augments the XML tree model by adding referential edges from each IDREF node to the corresponding ID node. A relational database can be modeled as a graph, where each node represents a tuple, and there is an edge from u to v if the corresponding tuples can be joined by key/foreign key. A social network can also be modeled as a graph, where a node represents a user and an edge represents the relationship between two users.

Structured/semi-structured data can optionally have a schema (such as relational schema, XML schema/DTD, etc.). For example, in the DTD of an XML document, for each XML node, it specifies the names of its sub-elements and attributes using regular expressions with operators * (a set of zero or more elements), + (a set of one or more elements), ? (optional), and | (or). Figure 2.1 shows a fragment of the DTD for the XML document in Figure 1.3. “ELEMENT retailer (name, product, store*)” indicates that a *retailer* element should have exactly one child *name*, one child *product*, and zero or more child *store*. “ELEMENT name (#PCDATA)” specifies that *name* node has a value child. We refer to the nodes that can have siblings of the same name as **-node*, as they are followed by a “*” in the DTD, such as the *retailer* and *store* nodes.

```
<!ELEMENT retailers (retailer*)>

<!ELEMENT retailer (name, product, store*)>

<!ELEMENT name (#PCDATA)>

<!ELEMENT product (#PCDATA)>

<!ELEMENT store (state, city, name, merchandises)>
```

Figure 2.1: Sample DTD Fragment

In this dissertation, we do not require a structured document to be associated with a schema or DTD. If the schema or DTD is available, we can make use of it to infer search semantics; however, in case the schema and DTD is absent, we are still able to intelligently infer the semantics and produce reasonable results solely based on the data itself. Details will be shown in the following chapters.

Nested Graph Data Model

A nested graph data model consists of a graph G and a set of abstract node specifications. Nodes in G are called atomic nodes or leaf nodes. Each abstract node represents an abstraction of a set of nodes, each of which is either an atomic node, or another abstract node. For an abstract node v which abstracts a set S of nodes, each node $s \in S$ is a child of v and v is a parent of s. Descendants and ancestors can be defined accordingly. There is a root node which is the ancestor of any other node.

This data model is useful for describing workflow hierarchies, such as the one shown in Figure 1.7. A workflow consists of a set of tasks and their relationships/dependencies for accomplishing a scientific or business goal (e.g., testing which two proteins are related). Since a workflow may be large, complex and contain details that are uninteresting to most users, people create composite tasks for workflows, each of which abstracts a set of tasks in the workflow. This forms a three dimensional structure which corresponds to the nested graph data model.

2.2 Keyword Query

In this dissertation, we consider the user input as a set of keywords, each of which may match name and/or value nodes in the XML tree. Since we use an unordered model, query (*men*, *apparel*) and query (*apparel*, *men*) have the same effect. If a keyword k is contained in the label of u, we say that u is a match to k.

In this dissertation we consider AND semantics (i.e., conjunctive queries) for keyword queries, i.e., each result of a keyword query should contain all keywords in the query. Chapters 4 - 6 discuss result definition and generation for keyword search on tree data. They take tree data and keyword query as input, and generate tree-structured query results. Chapters 7 - 11 discuss result analysis for tree-structured results, which are returned by most existing keyword search systems on tree/graph data. Chapter 12 discusses efficient result generation on trees. Chapter 13 discusses result definition and generation on nested graphs.

Chapter 3

RELATED WORK

This chapter briefly discusses the literature on keyword searches on structured data. Section 3.1 discusses works on query result definition and generation (i.e., resolving structural ambiguity and efficiency challenges), Section 3.2 discusses works on query cleaning, auto-completion and rewriting (i.e., resolving keyword ambiguity), and Section 3.3 discusses works on query result analysis (i.e., resolving structural, keyword and user preference ambiguities).

3.1 Result Definition and Generation

Since keyword search has no structures, it is important to resolve the structural ambiguity and find the meaningful structures for keyword queries. To do so, we can either infer the structures based on the data/schema and define query results accordingly, or make use of query forms.

Result Definition on XML. For keyword search on XML trees, it is observed that descendant nodes provide more specific information than ancestor nodes, thus many approaches are based on lowest common ancestors (LCA) for identifying relevant matches and composing query results. There are a number of variants of selecting particular LCA nodes as the roots of results, e.g., smallest lowest common ancestor (SLCA) [144, 91, 93, 61], efficient lowest common ancestor (ELCA) [56, 145], meaningful lowest common ancestor [86], compact valuable lowest common ancestor (CVLCA) [82], etc. XReal [18] is an XML search engine that computes the root nodes of the results and the relevant nodes using the statistics of the XML data. An algorithm for computing top-k results using ranking functions for SLCA and ELCA is developed [33]. The ranking factors adopted in [33] include node weight, node depth, and the score of a result can be any monotonic function that aggregates the node scores. Compared to the our MaxMatch method, none of these approaches satisfies all properties in our proposed axiomatic framework and may exhibit counter intuitive behaviors, as to be discussed in details in Chapter 4.

Result Definition on Graph. For keyword search on graphs, most approaches [55, 72, 102, 104, 48, 50, 70, 12, 60, 62, 21, 122, 89] adopt the minimal tree semantics. A minimal tree is a tree containing all keywords in the query, such that the removal of any node will make it no longer a tree containing all keywords. Since finding minimal trees is NP-hard (in particular, the minimal tree with the smallest tree size is actually the group Steiner tree), these approaches focus on different techniques to efficiently generate ranked results, in the presence of schema or the absence of the schema. Certain approaches allow a minimal tree to omit some keywords if a connected component of the data graph does not contain all keywords [102].

Query Forms. As discussed in Chapter 1, query forms is another way to resolve structural ambiguity and help casual users access structured data. They are often used as an advanced search feature, which are suitable for the users who want to retrieve precise results without the need to learn structured queries or the data schema. Since the possible number of query forms for a database can be very large, existing approaches selects the most promising query forms, either by analyzing the data and schema, or by asking the user to issue a keyword query and returning the relevant forms. [40] generates a set of query forms dynamically given a keyword query, each of which is an unfinished SQL query, to help users better express their needs. A set of form templates is pre-defined according to the schema, and upon receiving a keyword query, certain forms relevant to the query are selected and returned, which can be considered as processing keyword search on text documents. [67, 68] generate a set of promising query forms based solely on the data and schema. They identify the desirableness of a query form by analyzing the relationship between schema nodes as well as statistics in the data.

Answering Queries Using Materialized Views. There is a lot of work on evaluating queries using materialized views for XPath, XQuery and their subsets [16, 110, 14, 129, 103, 143]. Most of the work focuses on queries with child/descendant axes, wildcards, XPath predicates, equality and inequality comparisons, nested FLWR blocks and joins. [129] studies the strategy of selecting multiple materialized views for rewriting XPath queries. Materialized view maintenance for XPath/XQuery and their subsets has also been studied [120, 121]. However, due to the completely different result definition, answering keyword queries on structured data using materialized views as well as maintaining the views require novel techniques.

3.2 Query Cleaning, Auto-completion and Rewriting

As discussed before, the keywords submitted by the user may be imperfect, thus query cleaning, auto-completion and rewriting are helpful for resolving keyword ambiguity.

[113] studies methods for query cleaning, which corrects misspelled keywords and groups adjacent keywords that are semantically related for better efficiency. [101] proposes an improved query cleaning approach by using a noisy channel model, which guarantees that the cleaned query has non-empty results. [83] studies keyword query auto-completion using a trie-based approach. [31] also proposes a trie-based auto-completion approach for keyword search which tolerates errors in user's input. [138] studies query rewriting, which maps each keyword to a predicate or an ORDER BY clause based on analyzing the results retrieved by similar queries. [37] is another query rewriting approach which, given a keyword, generates its synonyms, hypernyms and hyponyms based on the click logs of Web search engines. There are also works that generate new queries based on popular words in the original query result [142, 28, 26, 130, 76, 119], considering factors like term frequency, inverse document frequency, ranking of the results in which they appear, etc. In particular, [76, 130] exploit relational databases instead of text documents, and [130] only considers term frequency but has the advantage of generating expanded queries without evaluating the original one. [132] additionally considers the proximity to the original query keywords when selecting words from results or corpus to compose new queries. Compared to these approaches, our query refinement methods to be presented in Chapter 11 is based on clustered query results, thus it provides a better categorization of the query results for ambiguous queries.

3.3 Result Analysis

Result analysis techniques, including result ranking, clustering, snippets, differentiation, aggregation, etc., help users quickly understand and analyze the results, hence gain useful insights from the results. Using these techniques, the user will be able to quickly understand the structure of a result, the keyword semantics in a result, as well as the comparison of the results and the meaningful aggregation of the results. Therefore, these techniques help resolve all three types of ambiguities: structural ambiguity, keyword ambiguity and user preference ambiguity.

Ranking. There are much research on effective ranking schemes for keyword search on structured data. There are two commonly adopted ranking factors: IR-style ranking and proximity-based ranking. IR-style ranking uses factors like term frequency (TF) and inverse document frequency (IDF), which are adapted from the traditional metrics in IR, to measure the weight of a keyword in a document. The relevance of a result is measured by an aggregation of the weight of each keyword in the result [89, 76, 84, 102, 60, 122]. Proximity-based ranking measures proximity of keywords in a result based on the observation that a shorter path between two nodes indicates a closer relationship than a longer path does [55, 84, 48, 70, 60, 62, 21, 50, 75, 115, 122].

Clustering. A number of approaches have been proposed for clustering XML documents. Most of the approaches utilize tree edit distances or its variances as the similarity measure [11, 87, 47, 45, 107, 140, 139, 78, 128] and others adopt a vector space model [52, 137, 136]. [11] uses the frequent substructures set to measure the similarity between documents. [87, 47, 45] propose different methods to summarize the graph. Based on the summarization they calculate the similarity between XML documents, which is more efficient compared with using the document trees. However, they do not consider data values, thus are inapplicable for clustering search results. [140, 139] measure the similarity between different XML documents by leveraging the schemata of documents. [128] proposes to incorporate ontology (i.e. WordNet) into the XML clustering. [131] performs an empirical study and verified that query-specific clustering increases the effectiveness of information retrieval compared with static clustering. Note that existing approaches focus on clustering XML documents (which can be search results) in a way independent of the user query.

XBridge [85] performs query-dependent query result clustering on XML based on the path from the XML root to each result root. Compared to this approach, our clustering method to be discussed in Chapter 9 has a finer granularity as our approach additionally considers the information within each result. However, [85] is able to rank the clusters and return the promising clusters without actually generating all query results. A software company, Vivisimo[1], provides applications for enterprises and governments which clusters search results on text documents, which addresses the keyword ambiguity issue for text search. However, the techniques used in Vivisimo is proprietary.

Snippet, Differentiation and Aggregation. As discussed in Chapter 1, user preference ambiguity can be tackled by result snippets, result differentiation and result aggregation. Our proposed methods for result snippets and differentiation are presented in Chapters 7 and 8. For result aggregation, [147] proposes to find minimal sets of tuples when the user searches a relational table, such that they contain all the query keywords, and share all or most of the values of a set of user specified attributes. [51] automatically identifies group-by attributes and values of search results to maximize the average score of the selected results. There are also a number of faceted search approaches [71, 29, 81, 34] that generate navigation trees for structured query results, in which each level has one or more facets (attributes). Navigation trees are generated with the goal of minimizing the user's navigation cost to find relevant results, where the navigation cost is estimated through a variety of sources such as query log, result entropy, etc.

[1]http://vivisimo.com

Chapter 4

REASONING AND IDENTIFYING RELEVANT MATCHES

4.1 Motivation and Goal

As discussed in Chapter 1, structural ambiguity of keyword search is caused by the lack of structures in a keyword search. A structured query, such as XQuery, specifies the precise structure of the result, i.e., what are the relevant nodes that should be selected (in the "for" and "where" clauses); what are the nodes that should be returned (in the "return" clause); how should the nodes be connected (how the XQuery should be structured, e.g., which node is the result root, how many nodes of each type are contained in a result, etc.). To resolve structural ambiguity, this information needs to be automatically inferred for keyword searches on structured data. In this chapter, we work on the problem of identifying relevant keyword matches (analogous to inferring "for" and "where" clauses for keyword searches). Chapters 5 and 6 discusses inferring return information and node connection, respectively.

For a keyword search, each keyword may have multiple matches in the XML document, and not all of them are necessarily relevant to the query. Consider query "*Galleria, city*" on the XML data in Figure 1.3 for example. There are multiple matches to keyword "city", but only the one with ID 0.0.2.1 is relevant to the query. Many different approaches have been proposed for identifying relevant keyword matches on XML using various concepts of Lowest Common Ancestor (LCA), including [144, 86, 56, 82, 43]. For example, a match is considered to be irrelevant in XKSEarch [144] if it is a descendant-or-self of a smallest LCA (SLCA) node, which is an XML tree node that contains all keywords in its subtree and none of its descendants does.

Different systems use different underlying principles and heuristics, leading to different query results in general. How to guide the design and to evaluate XML keyword search strategies are becoming critical research problems. However, due to the inherent ambiguity of search semantics, it is hard (if not impossible) to directly assess the relevance of query results and reason about various strategies.

Interestingly, we discover that by examining query results produced by the same

Q_1	Galleria, state
Q_2	Brooks Brothers, Galleria, state
Q_3	Brooks Brothers, Galleria, West Village, city
Q_4	store, Texas
Q_5	store, Texas, Galleria
Q_6	Brooks Brothers
Q_7	Galleria, Texas
Q_8	Texas, apparel, retailer
Q_9	men, apparel, store

Figure 4.1: Sample Keyword Searches

XML keyword search strategy on similar queries or on similar documents, sometimes abnormal behaviors can be clearly observed, which exhibit the pitfalls that a good search strategy should avoid. Let us start with some examples.

Example 4.1 *Consider XML tree D_1 that consists of the nodes in solid lines in Figure 1.3 (i.e., all nodes except the subtree rooted at* city *(0.0.3.1), where each node is associated with a unique ID (to be discussed later; some node IDs are omitted), and the keyword searches listed in Figure 4.1.*

$\underline{[Q_1, D_1]}$*:Processing query Q_1 (*Galleria, state*) which searches for the state of Galleria on D_1, a reasonable result contains matches in the subtree rooted at the* store$_1$*, but not the match node* state *(0.0.3.0) of store West Village. Such a query result will be produced by many existing XML keyword search systems [144, 56, 86, 43, 82].*

$\underline{[Q_2, D_1]}$*: To search for the state of Galleria for retailer Brooks Brothers, the user would issue Q_2 (*Brooks Brothers, Galleria, state*), containing one more keyword* Brooks Brothers *than Q_1.*[1] *A query result as produced by [144, 56, 43] includes all matches in the subtree rooted at* retailer *(0.0). Thus node* state *(0.0.3.0) is now included, which is unlikely to be justifiable. A reasonable result should still exclude this* state *node since it is not related to keyword* Galleria*.*

Example 4.2 $\underline{[Q_3, D_1]}$*: Consider Q_3 (*Brooks Brothers, Galleria, West Village, city*) on XML tree D_1, searching for the city of both* Galleria *and* West Village*. A reasonable query result*

[1]Suppose query segmentation has been performed and *Brooks Brothers* is considered as a single keyword.

should include the matches to Brooks Brothers*,* Houston *and* West Village *as well as the* city *node with ID 0.0.2.1, as produced by many existing XML keyword search systems [144, 56, 86, 43, 91].*

$\underline{[Q_3, D_2]}$*: Suppose the* city *information of store* West Village *is now inserted to the XML tree D_1 as represented by the dotted line, which results in an XML tree D_2. Now D_2 has the city information for both stores, as requested by Q_3.*

An empty result, as produced by [86, 82, 43], is unlikely to be desirable. When AND *semantics is considered, a keyword search is a* positive query*. A search strategy that outputs one query result for Q_3 but outputs nothing when a new data node is inserted is abnormal. A more reasonable query result for Q_3 on D_2 should additionally include the* city *node with ID 0.0.3.1 compared to the result on D_1.*

From these examples we can observe that there should be some correlation of the query results generated by a desirable XML keyword search engine of two similar queries on the same document or the same query on two similar documents. Since abnormal behaviors can be more easily identified when checking query pairs or document pairs than considering a single query on a single document, we attempt to assess the quality of XML keyword search engines from a new angle: capturing desirable *changes* to a query result upon a *change* to the query or data in a general framework.

Indeed, the approach that formalizes broad intuitions as a collection of simple axioms and evaluates various solutions based on the axioms has been successfully used in many areas, such as mathematical economics [111],[2] clustering [73], discrete location theory [57], and collaborative filtering [112].

In light of the success of an axiomatic approach in those areas, we initiate an investigation of a formal axiomatic framework to express valid changes to a query result upon an addition to the user query or to the data.[3] This is, nevertheless, very challenging. As we can see, when a new keyword is added to a query or when a new data node is inserted, some

[2]A striking case is the axioms on social choice functions [15], which can not be simultaneously satisfied by any solution, proposed by Kenneth Arrow, a co-recipient of the 1972 Nobel Prize in Economics.

[3] The desirable behaviors of an algorithm are symmetric for deletions, whose details are omitted.

keyword matches should become relevant and be added to the query result, such as *Brooks Brothers* (0.0.0.0) for $[Q_2, D_1]$ and *city* (0.0.3.1) for $[Q_3, D_2]$; but not all the matches, such as *state* (0.0.3.0) for $[Q_2, D_1]$. Similarly, some keyword matches should become irrelevant and be removed from the query result, but not all the matches.

After analyzing valid changes to query results, independently of any particular algorithm, we find that the properties that an XML keyword search algorithm should possess are very simple and intuitive: *data monotonicity*, *query monotonicity*, *data consistency* and *query consistency*. An algorithm that shows the abnormal behaviors illustrated in the above examples violates at least one of the properties, as will be analyzed later in this chapter.[4]

After reviewing the existing strategies on XML keyword search, we find that, surprisingly, none of them satisfies all these properties. We then design a novel XML keyword search engine, MaxMatch, which possesses all these properties and efficiently processes user queries.

4.2 Axiomatic Framework for Identifying Relevant Matches

Assumptions

We first introduce an assumption of query results before we proceed to discuss the properties.

Assumption: Processing query Q on XML data D will return *a set of query results*, denoted as $R(Q, D)$. Each query result is a tree defined by a pair $r = (t, M)$, where t is the root, M is a subset of matches in the tree, consisting of all the matches in the tree that are considered as relevant to Q. Every keyword in Q has at least one match in M. A *query result* is a tree consisting of the paths in D that connect t to each match in M (as well as its value child, if any). The number of query results, $|R(Q, D)|$, is the number of (t, M) pairs.

Note that one query result should not be *subsumed* by another, therefore the root nodes t in $R(Q, D)$ should not have ancestor-descendant relationship.

Example 4.3 *Consider Q_2 (*Brooks Brothers, Galleria, West Village, city*) on D_1 in Fig-*

[4]However, the proposed properties may not be complete, i.e. a system that satisfies these properties is not necessary a perfect system.

ure 1.3, which searches for the city of Galleria and West Village which are stores of Brooks Brothers. There is only one correct query result, where t= retailer*, M = $\{$* Brooks Brothers*,* Houston*,* West Village*,* city *(0.0.2.1) $\}$. The query result is a tree consisting of the paths from* retailer *to each node in M, including their value children.*

In this section we reason about identifying relevant matches to generate a query result. Instead of directly assessing the relevance of match nodes, we propose an axiomatic framework that characterizes the valid connection between the original query results and the new query results generated by the same algorithm when an update is performed on the query and/or the data. Specifically, property *monotonicity* captures reasonable changes to the number of query results (i.e. $|R(Q, D)|$); and *consistency* captures reasonable changes to the content of the set of query results (i.e. M sets in $R(Q, D)$).

Note that our approach is independent of any particular algorithm, therefore in this section we present some meaningful query results to illustrate the proposed properties, without considering *how to design an algorithm* that generates those results.

Monotonicity

Monotonicity describes the desirable change to the number of query results with respect to data updates and query updates.

Data Monotonicity. *If we add a new node to the data, then the data content becomes richer, therefore the number of query results should be (non-strictly) monotonically increasing.* Analogous to keyword search on text documents, adding a word to a document that is not originally a query result may qualify the document as a new query result. Similarly, for keyword search on XML trees, adding a node to the data may enable an XML subtree that is not originally a query result to be a new query result. Let us look at an example before defining data monotonicity formally.

Example 4.4 *Adding a new data node may increase the number of query results. Consider query "*store, city*" on XML data D_1 shown in Figure 1.3, which searches for the city of a store. Ideally, there should be one query result, rooted at* store *(0.0.2) with the matches* *in its subtree, and the paths connecting them. Now consider an insertion of a* city *node (0.0.3.1) and its value* Austin *to D_1, which results in XML tree D_2. Ideally, we should have one more query result: a tree rooted at* store *(0.0.3).*

*The number of query results may also stay the same after a data insertion. Consider Q_3 (*Brooks Brothers, Galleria, West Village, city*), on D_1 and D_2, respectively. For each document there should be a single query result tree, rooted at* retailer*, as this subtree contains at least one relevant match to each keyword. Though the set of relevant matches in the subtree rooted at the* retailer *node are different for D_1 and D_2*[5]*, the number of query result is one for both XML documents. On the other hand, if $R(Q_3, D_2)$ has an empty set of query results as discussed in Example 4.2, it violates data monotonicity. This is undesirable as the cities of both Galleria and West Village are indeed present in D_2.*

Definition 4.1 (Data Monotonicity) *An algorithm satisfies* data monotonicity *if for a query Q and two XML documents D and D', $D' = D \cup \{n\}$, where n is an XML node, $n \notin D$, the number of query results on document D' is no less than that on D, i.e., $|R(Q, D)| \leq |R(Q, D')|$.*

Query Monotonicity. *If we add a keyword to the query, then the query becomes more restrictive, and the number of query results should be (non-strictly) monotonically decreasing*. Analogous to keyword search on text documents, adding a keyword to the query can disqualify a document that is originally a query result to be a result of the new query. Similarly, for XML keyword search, adding a new keyword can disqualify a query result of the original query if it is far away from any match to the new keyword.

Example 4.5 *Adding a new keyword may decrease the number of query results. For example, there are two query results when processing Q_4 on the XML data D_2, rooted at* store *(0.0.2) and* store *(0.0.3) respectively. Now suppose we add one more keyword* Galleria *to the query, which results in Q_5 in Figure 4.1, searching for the store named Galleria. The query result rooted at node* store *(0.0.2) is still a relevant query result, However, the*

[5]We discuss the desirable changes of relevant matches in consistency property.

one rooted at store *(0.0.3) becomes invalid, as it does not contain any match to* Galleria *in its subtree. To satisfy the* AND *semantics of the query, one would think of replacing the query result rooted at* store *(0.0.3) with the one rooted at* retailer *which contains at least one match to each keyword in its subtree. However, since match node* Galleria *belongs to a different store than match nodes* store *(0.0.3) and* Texas *(0.0.3.0.0), they are unlikely to be meaningfully related to define a relevant query result. Therefore, the number of query results of* Q_5 *is reduced to one.*

The number of query results may stay the same after a query insertion. Consider Q_1 *and* Q_2 *on* D_1*, where* Q_2 *contains one more keyword* Brooks Brothers *than* Q_1*. The query result* $R(Q_1, D_1)$ *would be* store *(0.0.2) along with the match nodes in the subtree; the query result* $R(Q_2, D_1)$ *would be* retailer *along with the relevant match nodes. Though the returned information of processing* Q_1 *and* Q_2 *is different, both queries have the same number of query result: one.*

Definition 4.2 (Query Monotonicity) *An algorithm satisfies* query monotonicity *if for two queries* Q *and* Q' *and an XML document* D*,* $Q' = Q \cup \{k\}$*, where* k *is a keyword,* $k \notin Q$*, the number of query results of* Q' *is no more than that of* Q*, i.e.,* $|R(Q, D)| \geq |R(Q', D)|$*.*

Consistency

Monotonicity describes how the number of query results should change upon an update to the data or query. Consistency describes how the content of query results should change upon an update to the data or query. Intuitively, the delta of two sets of query results can be defined as the biggest subtrees that are in one set of query results but not in the other, named as *delta result trees*.

Example 4.6 *Consider* $R(Q_3, D_1)$ *and* $R(Q_3, D_2)$*. The subtree rooted at* city *(0.0.3.1) is the biggest subtree that is in* $R(Q_3, D_2)$ *but not in* $R(Q_3, D_1)$*, i.e., a delta result tree. Indeed, every node in this subtree is in* $R(Q_3, D_2)$*, and none of them is in* $R(Q_3, D_1)$*. On the other hand, the subtree rooted at its parent node* store *(0.0.3) is not a delta result tree, as some nodes, e.g.* store *(0.0.3),* state *(0.0.3.0) are in* $R(Q_3, D_1)$*.*

Now we formally define a *delta result tree* in XML keyword search as the subtree that newly becomes part of the set of query results upon an insertion to the data or query. Note that a delta result tree could be a query result itself, or could be part of a query result.

Definition 4.3 (Delta Result Tree (δ)) *Let R be a set of query results of processing query Q on data D, and R' be the set of updated query results after an insertion to Q or D. A subtree rooted at a node n in a query result tree $r' \in R'$ is a* delta result tree *if $desc\text{-}or\text{-}self(n, r') \cap R = \emptyset$ and $desc\text{-}or\text{-}self(parent(n, r'), r') \cap R \neq \emptyset$, where $parent(n, r')$ and $desc\text{-}or\text{-}self(n, r')$ denotes the parent, and the set of descendant-or-self nodes of node n in a tree r', respectively. The set of all delta result trees is denoted as $\delta(R, R')$.*

Data Consistency. *After a data insertion, each additional subtree that becomes (part of) a query result should contain the newly inserted node.* Analogous to keyword search on text documents, after we add a new word to the data, if there is a document that becomes a new query result, then this document must contain the newly inserted word. Similarly, for keyword search on XML trees, after we add a new node to the XML data, if there exists a delta result tree in the new query result, then this delta result tree should contain the newly inserted node to be qualified (because otherwise, this sub-tree should not be part of the new query result in order to be consistent with the original query result). Let us look an example before defining data consistency formally.

Example 4.7 *Consider query "*store, city*" on XML data D_1 shown in Figure 1.3, which searches for the city of a store. There is one query result, consisting of* store *(0.0.2) and the matches in its subtree. Now consider D_2 obtained after an insertion of a* city *(0.0.3.1) node along with its value* Austin *to D_1. This insertion qualifies a new query result for query "*store, city*", consisting of* store *(0.0.3) and the matches in its subtree. This new query result is a delta result tree, as it is the biggest subtree that is in the new result but not in the original result. It is valid with respect to data consistency since the delta result tree contains the newly inserted match node* city *(0.0.3.1).*

*Consider Q_1 "*Galleria, city*" on D_1 and D_2, searching for the city of Galleria. Although the newly inserted node* city *(0.0.3.1) is a match of Q_1, the query result should not* *change. Intuitively, this match refers to the* city *of a store other than* Galleria*, and therefore is irrelevant. In this case, there does not exist a delta result tree, and data consistency holds trivially.*

It is easy to verify that the changes to the query results in the above examples also satisfy data monotonicity.

Definition 4.4 (Data Consistency) *An algorithm satisfies* data consistency *if for query Q and two XML documents D and D', $D' = D \cup \{n\}$, where n is an XML node, $n \notin D$, if $\delta(R(Q, D), R(Q, D'))$ is not empty, then every delta result tree contains n (so there can only be one delta result tree).*

Query Consistency. *If we add a new keyword to the query, then each additional subtree that becomes (part of) a query result should contain at least one match to this keyword.* Analogous to keyword search on text documents, after we add a new keyword to the query, if a document remains to be a query result, then it must contain a match to the new keyword. Similarly, for keyword search on XML trees, after we add a new keyword to the query, if there exists a delta result tree in the new query result, then this delta result tree must contain at least one match to the new keyword (because otherwise, this sub-tree should not be part of the new query result in order to be consistent with the original query result).

Example 4.8 *Consider again Q_4 on XML data D_2 in Figure 1.3. We have two query results:* store *(0.0.2) and the matches in its subtree;* store *(0.0.3) and the matches in its subtrees. If we add one more keyword* Galleria *to the Q_4, which results in Q_5, then* store *(0.0.3) should no longer be a query result, which satisfies query monotonicity. On the other hand, the query result related to* store *(0.0.2) should add the subtree rooted at* city *(0.0.2.1), which is a delta result tree. This is valid with respect to query consistency since this subtree contains a node matching the new keyword* Galleria*.*

Now consider Q_1 and Q_2 on D_1, where Q_2 has one new keyword Brooks Brothers *compared with Q_1. Compared with $R(Q_1, D_1)$, $R(Q_2, D_1)$ should additionally contain* *the subtree rooted at* name *(0.0.0). This is valid with respect to query consistency since the delta result tree rooted at* name *(0.0.0) contains a match to the new keyword* Brooks Brothers*. On the other hand, if $R(Q_2, D_1)$ also contains* store *(0.0.3), then there is another delta result tree compared to $R(Q_1, D_1)$, rooted at* store *(0.0.3). However, since this delta result tree does not contain any match to the new keyword* Brooks Brothers*, this violates query consistency. This query result is indeed undesirable as* state *(0.0.3.0) is irrelevant with store Galleria.*

Definition 4.5 (Query Consistency) *An algorithm satisfies* query consistency *if for two queries Q and Q' and an XML document D, $Q' = Q \cup \{k\}$, where k is a keyword, $k \notin Q$, if $\delta(R(Q, D), R(Q', D))$ is not empty, then every delta result tree contains at least one match to k.*

Monotonicity and consistency properties with respect to data and queries are non-trivial, non-redundant, and satisfiable. They are not trivial, as to the best of our knowledge, there is no existing XML keyword algorithm that satisfies all of them. They are not redundant since we can find algorithms that satisfy one property but fail another. Detailed analysis will be discussed in Section 4.3. Furthermore, we show that these properties are satisfiable by proposing a keyword search semantics that satisfies all of them in Section 4.4.

4.3 Analyzing Existing algorithms

Several approaches have been proposed for identifying relevant matches for XML keyword search, including XKSearch [144], XRank [56], XSEarch [43], Compact Valuable LCA [82] and Schema-free XQuery [86]. In this section, we review and analyze these approaches in terms of monotonicity and consistency with respect to data and query.

XKSearch [144]. XKSearch proposes a concept of *Smallest Lowest Common Ancestor* (*SLCA*). For a query Q on data D, an XML node is an SLCA if it contains matches to all keywords in Q in its subtree, and none of its descendants does. For each SLCA, all its descendant matches are considered as relevant to Q.

However, not all such matches are necessarily relevant. For example, consider Q_2 (*Brooks Brothers, Galleria, state*) on D_1, where the SLCA node is *retailer*. Although node *state* (0.0.3.0) is a match in the subtree rooted at the SCLA node, it is irrelevant to the query as it is not the *state* of *Galleria*. This undesirable behavior can be detected by analyzing consistency on XKSearch.

XKSearch does not satisfy query consistency. Consider Q_1 and Q_2 on D_1. As we can see, the subtree rooted at *store* (0.0.3) is a delta result tree in $\delta(R(Q_1, D_1), R(Q_2, D_1))$. However, it does not contain matches to the new keyword *Brooks Brothers*, and therefore violates query consistency.

XRank [56]. According to the definition in XRank, an XML node is the root of a query result if it contains at least one occurrence of each keyword in its subtree, after excluding the occurrences of the keywords in its descendants that already contain all the keywords. All descendant matches of such nodes are considered relevant.

XRank does not satisfy query consistency. For Q_1 and Q_2 on D_1, XRank produces the same result as XKSearch.

The following approaches, XSEarch, Compact Valuable LCA, and schema-free XQuery, use a group of matches containing one match to each keyword, referred as *pattern match*, to identify relevant matches. For a query Q on data D, these approaches find qualified pattern matches according to their specific metrics. A match that is in a qualified pattern match is considered as a relevant match. For example, consider Q_2 on D_1 in Figure 1.3, there are two pattern matches, {*Brooks Brothers*, *Galleria*, *state* (0.0.2.0)}, and {*Brooks Brothers*, *Galleria*, *state* (0.0.3.0)}. Suppose the first pattern match is considered to be qualified, but not the second. Then all the matches in the first pattern match are relevant.

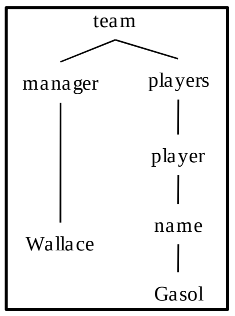
XSEarch [43]. XSEarch defines more expressive search terms than keywords. There are three types of search terms. A node n satisfies term $l : k$ if $n.name = l$ and n has a descendant leaf node whose value contains k; a node n satisfies term l : if n's name contains l; a node n satisfies term : k if it has a child leaf node whose value contains k. Each word in a search term can be considered as a user input keyword. A node that satisfies a search term is a match.

To identify relevant matches, XSEarch defines *interconnection* relationship among two matches, and uses two semantics, namely *all-pair* semantics and *star* semantics, to identify relevant pattern matches. Two matches n and n' are interconnected if the shortest path between n and n' (through LCA(n, n')) does not have two distinct nodes with the same name, except n and n'. All-pair semantics considers a pattern match P to query Q on data D as qualified if any two nodes in P are interconnected. Star semantics considers a pattern match P as qualified if there is a node in P such that every other node in P is interconnected with it. As we can see, for a query containing two search terms, all-pairs and star semantics are equivalent.

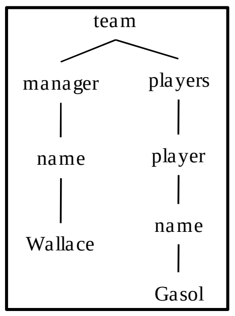
XSEarch may fail to identify relevant matches even though they exist. Consider query (*:Galleria, :West Village*) on D_1. The nodes that satisfy the search terms are *name* (0.0.2.2) and *name* (0.0.3.2), composing a pattern match. However, this pattern match is not qualified since the two match nodes are not interconnected: there are two distinct nodes (0.0.2, 0.0.3) with the same name *store* on the shortest path connecting them. Thus XSEarch gives an empty result for this query. This is unlikely to be desirable as the user may be interested in finding the relationship among two persons whose names are specified in the query. Such a behavior can be captured by analyzing query monotonicity.

XSEarch, both all-pair semantics and star semantics, do not satisfy query monotonicity. For query (*:Galleria, :West Village*) on D_1, XSEarch has an empty query result, as discussed above. Now consider query (*store:Galleria, store:West Village*), which has one more keyword than the previous query. There is one qualified pattern match: $\{$*store* (0.0.2) : *Galleria*, *store* (0.0.3) : *West Village*$\}$, which constitutes a query result. After increasing a keyword *store*, the number of query results produced by XSEarch increases, thus it violates query monotonicity.

Besides, XSEarch star semantics does not satisfy query consistency. Consider query (*:Galleria, state:*) (corresponding to Q_1) on D_1. *state* (0.0.3.0) is not considered as a relevant match, as it is not interconnected with node *Galleria* that matches term *:Galleria*. Now if we add one more search term to form a new query (*:Brooks Brothers, :Galleria,*



(a) D_3



(b) D_4

Figure 4.2: D_3 and D_4

state:) (corresponding to Q_2), then *state* (0.0.3.0) is considered as relevant. This is because there is a qualified pattern match $\{$*name* (0.0.0), *Galleria*, *state* (0.0.3.0)$\}$, where *name* (0.0.0) is interconnected with the other two nodes. Note that the relevant matches identified by XSEarch for this query is the same as those identified by XKSearch and XRank. As discussed, such behavior is undesirable and XSEarch star semantics violates query consistency.

Compact Valuable LCA (CVLCA) [82]. This approach proposes the concept of *Compact Valuable LCA*. For a keyword query Q on data D, a node u is considered a valuable LCA (VLCA) if there is a pattern match P that satisfies XSEarch all-pair semantics, and u is the LCA of the nodes in P. A node u is a CVLCA if it is a VLCA of pattern match P, and dominates every node in P. u dominates a node v in P if for any other pattern match P' that contains v, the LCA of nodes in P' is an ancestor-or-self of that of P.

CVLCA does not satisfy data monotonicity. Consider query (*Wallace, Gasol*) on D_3 in Figure 4.2. Pattern match (*Wallace, Gasol*) is qualified as the two nodes are interconnected. Now we insert a *name* node between nodes *manager* and *Wallace*, resulting in the XML tree D_4. Nodes *Wallace* and *Gasol* are no longer interconnected, thus the query result on D_4 is empty. Since the insertion to data results in fewer query results, this violates data monotonicity. Such a behavior is indeed counter intuitive. If *Wallace* and *Gasol* are considered to be relevant to each other in D_3, adding the description that *Wallace* is a *name* in D_4 should not disqualify its relevance.

MLCA [86]. The concept of *Meaningfully Lowest Common Ancestor (MLCA)* is proposed as part of Schema-free XQuery which allows users to query XML with arbitrary knowledge of the underlying schema. MLCA can be used to identify qualified pattern matches and thus relevant matches in XML keyword search. Two XML nodes n_1 and n_2 that match keywords k_1 and k_2 are meaningfully related if there does not exist n'_1 ad n'_2 that match k_1 and k_2, such that LCA(n_1, n_2) is an ancestor of LCA(n'_1, n'_2). For a keyword search Q on data D, a pattern match P is qualified if every two nodes in P are meaningfully related. If P qualifies, the LCA of all nodes in P is defined as an MLCA of Q on D, and the matches in a qualified pattern match are relevant.

MLCA does not satisfy data monotonicity. Consider Q_3 on D_1 and D_2. There is one query result produced by MLCA on D_1, which is rooted at *retailer*. The query result $R(Q_3, D_2)$ produced by MLCA is empty, since we are not able to find a pattern match, such that every pair of nodes in it are meaningfully related. No matter which *city* match we choose in a pattern match, it is not related to at least one another node. For instance, if we choose *city* (0.0.2.1), it is not related to *West Village*. This is because they have an LCA node *retailer*. However, *city* (0.0.3.1) has a lower LCA with *West Village*, which is *store* (0.0.3). Similarly, we cannot choose the other *city* match to compose a qualified pattern match. Therefore MLCA violates data monotonicity since the number of query results decreases when we add a new node to the data. Such a behavior is not desirable. Q_3 is likely to search the *city* of both *Galleria* and *West Village* of retailer *Brooks Brothers*. Since this information is present in the data, the query result should not be empty.

In summary, none of the existing approaches for identifying relevant matches in XML keyword search satisfies all four properties.

4.4 MaxMatch: An Approach that Satisfies the Desirable Axioms

In this section, we show that the properties proposed in Section 4.2 are satisfiable by presenting MaxMatch, an effective and efficient XML keyword search technique. We first introduce the semantics of MaxMatch for identifying relevant matches, then propose an efficient algorithm to achieve it.

Recall that a query result tree is define by a pair, $r = (t, M)$, where t is the root, M is the set of matches in the tree that are considered as relevant to Q, and every keyword in Q has at least one match in M. We adopt a commonly used approach in the literature [86, 144, 61], namely *SLCA*, to identify t, as reviewed in this section. MaxMatch addresses the challenge of identifying relevant matches M within t.

The intuition of SLCA is that only the matches in a *smallest subtree* in the XML data that contains matches to every keyword in a query are possibly relevant. A tree rooted at node n_1 is smaller than the one rooted at node n_2 if n_1 is a descendant of n_2. Let us look at an example.

Example 4.9 *For* Q_1 *(*Galleria, state*), node* state *(0.0.2.0) and* Galleria *are relevant matches since they refer to the same store. Indeed they are in the subtree rooted at* store *(0.0.2), which is a smallest subtree that contains matches to both keywords. On the other hand, irrelevant match* state *(0.0.3.0) can be detected since the subtree that contains this match and a match to* Galleria *is rooted at* retailer*, which is not a smallest subtree that contains matches to both keywords.*

As can be seen from the example, matches that are not in a smallest subtree that contains matches to all keywords are unlikely to be relevant, such as the *state* node 0.0.3.0. Choosing such smallest subtrees can prune some irrelevant matches.[6]

Given the intuition of such smallest subtrees, now we formally define *Descendant Matches* and *SLCA* [144].

Definition 4.6 (Descendant Matches) *For query* Q *on XML data* D*, the* descendant matches *of a node* $n \in D$*, denoted by* $dMatch(n)$*, is a set of keywords in* Q*, each of which has at least one match in the subtree rooted at* n*.*

[6]Additional irrelevant matches need to be pruned as will be discussed later in this section.

Definition 4.7 (SLCA) *A set of* smallest lowest common ancestor (SLCA) *of matches to Q on D, denoted by $SLCA(Q, D)$, consists of nodes $t \in D$ that satisfy the following: (i) the set of descendant matches of t is Q, i.e. $dMatch(t) = Q$, and (ii) there does not exist a node t' which is a descendant of t, such that $dMatch(t') = Q$.*

Example 4.10 *Continuing the previous example, there are three nodes in D_1 that contain matches to both keywords in Q_1 in their subtrees, $dMatch(0.0.2) = dMatch(0.1) = dMatch(0) = Q$. Therefore we have $SLCA(Q_1, D_1)$=$\{0.0.2\}$.*

In MaxMatch, a query result tree is identified as $r = (t, M)$, where $t \in SLCA(Q, D)$ is the root. Next we introduce an XML keyword search strategy that selects relevant matches M in the subtree rooted at t, which satisfies both monotonicity and consistency.

Semantics of Selecting Relevant Matches

Not all matches in the subtrees rooted at SLCA nodes are relevant. Recall $R(Q_2, D_1)$ in Example 4.3, where match node *state* (0.0.3.0) in the subtree rooted at the SLCA node *retailer* in D_1 is irrelevant to Q_2 (*Brooks Brothers, Galleria, state*) since it corresponds to store *West Village*.

To identify irrelevant matches, we observe that not every descendant of an SLCA node is equally important in contributing to query results. A node in the subtree rooted at an SLCA node may provide strictly less information than its sibling nodes. Continuing the example of $R(Q_2, D_1)$, *store* (0.0.3) provides strictly less information than its sibling node *store* (0.0.2), since the set of descendant matches of *store* (0.0.3) ({*state*}), is a proper subset of that of *store* (0.0.2) ({*Galleria*, *state*}). Therefore *store* (0.0.3) is considered to be inferior than *store* (0.0.2) and the matches in its subtree are considered as irrelevant.

Definition 4.8 (Contributor) *For an XML tree D and a query Q, a node n in D is a* contributor *to Q if (i) n has an ancestor-or-self n_1 in the SLCA set, $n_1 \in SLCA(D, Q)$, and (ii) n does not have a sibling n_2, such that $dMatch(n_2) \supset dMatch(n)$, where $dMatch(n)$ is the set of descendant matches of node n (Definition 4.6).*

By Definition 4.8, an SLCA node is a contributor. Furthermore, there is at least one contributor among sibling nodes. Now we define relevant matches based on contributors.

Definition 4.9 (Relevant Match) *For an XML tree D and a query Q, a match node m in D is* relevant *to Q if (i) m has an ancestor-or-self n, $n \in SLCA(D, Q)$, and (ii) every node on the path from n to m is a contributor to Q.*

For Q_2 on D_1, every node on the path from SLCA *retailer* to *Galleria* is a contributor, therefore *Galleria* is a relevant match. Similarly, *Brooks Brothers* and *state* (0.0.2.0) are relevant matches.

Note that we consider the partial order among sibling nodes induced by the $\subset$ relationship of the sets of their descendant matches, therefore a contributor is a *maximal* node among its siblings. A match is considered to be relevant if its ancestors are all maximal nodes among the siblings.

Proposition 4.1 *(t, M) qualifies to be a query result, where SLCA node $t \in SLCA(Q, D)$, M is the set of relevant matches in the subtree rooted at t. In other words, M contains at least one match to every keyword in Q.*

Proof. Let M' be the set of all the match nodes in the subtree rooted at t in D. According to the definition of SLCA (Definition 4.7), M' contains at least one match to each keyword in Q. Removing the irrelevant matches from M' result in set M. If an irrelevant match m_1 to keyword k is pruned, then it must have an ancestor-or-self n_1 that is not a contributor. That is, n_1 has a sibling contributor node n_2, $dMatch(n_1) \subset dMatch(n_2)$. Therefore $k \in dMatch(n_2)$. By induction, there still exists at least one match to keyword k in the subtree of t, for every keyword $k \in Q$, and finally M contains at least one match to each keyword.

Definition 4.10 (Query Results of MaxMatch) *For an XML tree D and query Q, each query result generated by MaxMatch is defined by $r = (t, M)$ for every $t \in SLCA(Q, D)$, where M is the set of* relevant matches *to keywords in Q in the subtree rooted at t. A query result tree r consists of the contributors and relevant matches that are descendants of t.*

Property Analysis

We prove that MaxMatch satisfies both monotonicity and consistency with respect to data and query.

Proposition 4.2 *MaxMatch satisfies data monotonicity.*

Proof. Consider query Q and two XML documents D and D', $D' = D \cup \{n\}$, where n is an XML node, $n \notin D$. According to Definition 4.10, the number of query results generated by MaxMatch is equal to the number of SLCA nodes. For any $t \in SLCA(Q, D)$, there are two possibilities.

- $t \in SLCA(Q, D')$.
- $t \notin SLCA(Q, D')$. *We still have $dMatch(t) = Q$, but t is no longer the root of a smallest subtree that contains matches to all keywords. Then there must exist at least one descendant of t that qualifies to be in $SLCA(Q, D')$.*

Therefore, we have $|SLCA(Q, D')| \geq |SLCA(Q, D)|$, and $|R(Q, D')| \geq |R(Q, D)|$.

Proposition 4.3 *MaxMatch satisfies query monotonicity. Proof. Consider two queries Q and Q' and an XML document D, $Q' = Q \cup \{k\}$, where k is a keyword, $k \notin Q$. For any $t \in SLCA(Q, D)$, there are two possibilities:*

- $t \in SLCA(Q', D)$
- $t \notin SLCA(Q', D)$. *This is because t no longer contains matches to all keywords in its subtree, i.e. $k \notin dMatch(t)$, $dMatch(t) \neq Q'$. It is not possible for a descendant of t to be in $SLCA(Q', D)$. At most one ancestor of t can be in $SLCA(Q', D)$, since SLCA nodes do not have ancestor-descendant relationship by Definition 4.7.*

Therefore, we have $|SLCA(Q', D)| \leq |SLCA(Q, D)|$, and $|R(Q', D)| \leq |R(Q, D)|$.

Proposition 4.4 *MaxMatch satisfies data consistency.*

Proof. *Consider query Q and two XML documents D and D', $D' = D \cup \{n\}$, where n is an XML node, $n \notin D$. If $\delta(R(D,Q), R(D',Q))$ is empty, then MaxMatch trivially satisfies this property.*

If $\delta(R(D,Q), R(D',Q))$ is not empty, let n_1 be the root of a delta result tree in $\delta(R(D,Q), R(D',Q))$, and n_2 be the parent of n_1. By Definition 4.3 and 4.8, n_1 is not a contributor of (D, Q), but a contributor of (D', Q), and n_2 is a contributor of both (D, Q) and (D', Q). Therefore, there must be a node n_3 which is a sibling of n_1, such that $dMatch(n_1) \subset dMatch(n_3)$ holds for D, but not for D'. This shows that the delta result tree rooted at n_1 must contain the newly inserted node n, and n must match a keyword in Q.

Proposition 4.5 *MaxMatch satisfies query consistency.*

Proof. *Consider two queries Q and Q' and an XML document D, $Q' = Q \cup \{k\}$, where k is a keyword, $k \notin Q$. The proof is similar to that of Proposition 4.4. If $\delta(R(D,Q), R(D,Q'))$ does not exist, then MaxMatch trivially satisfies the property. Otherwise, for the root n_1 of each delta result tree in $\delta(R(D,Q), R(D,Q'))$, n_1 must contain a match to the new keyword k in its subtree.*

Algorithm

The algorithm of realizing the semantics of MaxMatch is presented in Algorithm 1. There are four stages in the processing. First, we retrieve the matches to each keyword in the query using procedure $findMatch$. Then we compute the set of SLCA nodes from the matches using procedure $findSLCA$. We group all keyword matches according to their SLCA ancestors using procedure $groupMatches$. Each group $group[i] = \{t, M\}$ consists of an SLCA node t, and the set of matches M that are descendants of t. Finally, the $pruneMatches$ procedure identifies and outputs contributors and relevant matches in M.

Next we illustrate each stage of the algorithm using Q_2 (*Brooks Brothers, Galleria,*

Algorithm 1 MaxMatch

```
MaxMatch ($keyword[w]$)
 1: $kwMatch \leftarrow findMatch(keyword)$
 2: $SLCA \leftarrow findSLCA(kwMatch)$ {adopted from [144]}
 3: $group \leftarrow groupMatches(kwMatch, SLCA)$
 4: for all $group[j]$ do
 5:    $pruneMatches(group[j])$
GroupMatches ($kwMatch[w]$, $SLCA[u]$)
 1: $match[v] \leftarrow merge(kwMatch[0], ..., kwMatch[w-1])$
 2: $i \leftarrow 0, j \leftarrow 0$
 3: while $(i \neq u)$ or $(j \neq v)$ do
 4:    $group[i].t \leftarrow SLCA[i]$
 5:    if $isAncestor(group[i].t, match[j])$ then
 6:       $group[i].M = group[i].M \cup match[j]$
 7:       $j \leftarrow j+1$
 8:    else if $group[i].M \neq \emptyset$ then
 9:       $i \leftarrow i+1$
10:    else
11:       $j \leftarrow j+1$
PruneMatches ($group = (t, M)$)
 1: $i \leftarrow M.size$
 2: $start \leftarrow t$
 3: while $i \geq 0$ do
 4:    for each node $n$ on the path from $M[i]$(exclusively) to $start$ do
 5:       if $n$ matches $keyword[j]$ then
 6:          set the $j^{th}$ bit of $n.dMatch$ to $1$
 7:       $n_p \leftarrow n.parent$, $n_c \leftarrow n.child$ on this path
 8:       if $n_c \neq Null$ then
 9:          $n.dMatch \leftarrow n.dMatch\ OR\ n_c.dMatch$
10:       $n.last \leftarrow i$ {record the last descendant match of $n$}
11:       $n_p.dMatchSet[num(n.dMatch)] \leftarrow true$ {let $num$ be the function converting a binary
          number to a decimal number}
12:    $i \leftarrow i-1$
13:    $start \leftarrow LCA(M[i], M[i+1])$
14: $i \leftarrow 0$
15: $start \leftarrow t$
16: while $i \leq M.size$ do
17:    for each node $n$ from $start$ to $M[i]$ do
18:       if $isContributor(n) = false$ then
19:          $i \leftarrow n.last + 1$ {skip the matches in the subtree rooted at $n$}
20:          $break$
21:       else
22:          output $n$
23:    $i \leftarrow i+1$
24:    $start \leftarrow LCA(M[i-1], M[i])$
IsContributor ($n$)
 1: $n_p \leftarrow n.parent$
 2: $i \leftarrow num(n.dMatch)$
 3: for $j \leftarrow i+1$ to $2^w - 1$ do
 4:    if $n_p.dMatchSet[j] = true$ && $AND(i, j) = i$ then
 5:       return $false$
 6: return $true$
```

state) on D_2 in Figure 1.3 as a running example, which searches for the position of store Galleria and West Village in retailer Brooks Brothers.

Matching Keywords. For a set of input keywords $keyword[w]$, procedure $findMatch$ retrieves the list of data nodes sorted in the order of their ID, $kwMatch[j]$, that match keyword $keyword[j], 1 \leq j \leq w$. To enable efficient retrieval, an inverted index is built from a word to the XML name or value nodes that contain this word.

Example 4.11 *We start with retrieving the list of nodes matching each keyword in Q_2:* Brooks Brothers*,* Galleria*,* city *(0.0.2.1, 0.0.3.1), respectively.*

Computing SLCA. The procedure $findSLCA$ computes the SLCA nodes from $kwMatch$ according to the algorithm proposed in [144], which utilizes the Dewey labeling scheme introduced in Chapter 2. To efficiently retrieve the information of a node with its Dewey ID, we build a Dewey index of Btree structure, clustered by Dewey ID.

Example 4.12 *In our example, the SLCA node is* retailer *(0.0), as this is a lowest (in fact, the only) node that contains matches to all the keywords.*

Grouping Matches. Then the $groupMatches$ procedure groups keyword matches $kwMatch$, such that the matches in each group are descendants of the same SLCA node.

First we merge $kwMatch[j]$ ordered by Dewey ID, $1 \leq j \leq w$, to produce a $match$ list ordered by Dewey ID. Then we build groups according to $match$ and the SLCA nodes, such that for each $t \in SLCA$, we have $group[i] = (t, M)$, M is the set of matches in the subtree rooted at t. According to Definition 4.7, $M \neq \emptyset$. Furthermore, SLCA nodes do not have ancestor-descendant relationship, a match can have at most one SLCA ancestor, and therefore belong to at most one group. Due to these two properties and the fact that $SLCA$ and $match$ are sorted by Dewey ID, the grouping can be achieved by a single traversal of $SLCA$ and $match$, during which matches that do not belong to any group (i.e. do not have an $SLCA$ ancestor) are discarded. We set the cursor to the beginning of $SLCA$, $group$, and $match$ ($i = 0$, $j = 0$). For each $SLCA[i]$, we set $group[i].t = SLCA[i]$. If the current node $match[j]$ is a descendant of $group[i].t$, then it is added into $group[i].M$. Otherwise, if $group[i].M$ is not empty, it is possible that $match[j]$ is a descendant of $SLCA[i+1]$, we move the cursor of $SLCA$ and $group$ $(i+1)$. If $group[i].M$ is empty, then $match[j]$ does not have an SLCA node as its ancestor and does not belong to any group, therefore it is discarded.

Example 4.13 *Continuing our running example, we merge four lists of $kwMatch$ and produce $match$:* Brooks Brothers *(0.0.0.0),* state *(0.0.2.0),* Galleria *(0.0.2.2.0),* state *(0.0.3.0). We group $match$ based on the SLCA nodes. In this example, there is only one SLCA node:* retailer *(0), therefore we have $group[0]$, where $group[0].t$ is the SLCA node, and $group[0].M$ is equal to $match$.*

Pruning Matches. Each group $group = (t, M)$ defines a tree T_g composed of the nodes on the paths from t to each match in M. Procedure $pruneMatches$ identifies and outputs the *contributors* and *relevant matches* in T_g as query results.

According to Definition 4.8, a node n is a contributor if n does not have any sibling whose descendant match set is a proper superset of that of n. We use a boolean array $n.dMatch$ of size w to record the set of descendant matches of node n with respect to query $keyword[w]$. This array is represented as a binary number that has 1 at position j if and only if $keyword[j]$ has a match in the subtree rooted at n, $1 \leq j \leq w$. Let $num(n.dMatch)$ be the decimal value of $n.dMatch$.

Example 4.14 *In the running example of processing Q_2 (*Brooks Brothers*,* Galleria*,* state*), node* store *(0.0.2) contains matches to* Galleria *and* state *in its subtree, therefore its $dMatch$ is 011. Similarly, the $dMatch$ of* store *(0.0.3) and* name *(0.0.0) are 001 and 100, respectively.*

Instead of checking $n.dMatch$ with respect to each of its siblings, we associate its parent node n_p a boolean array $dMatchSet$ of size 2^w to record the $dMatch$ information of n_p's children. Specifically, $n_p.dMatchSet[i] = true$ if and only if n_p has a child n, such that $num(n.dMatch) = i$.

If the values of $dMatch$ and $dMatchSet$ are set, procedure $isContributor$ determines that a node n is a not a contributor if there exists a number j that *subsumes* i, and we have $n_p.dMatchSet[j]$
$= true$. A number j subsumes i if bitwise $AND(i, j) = i, j \neq i$.

Example 4.15 *Continuing our running example, let n be the node* store *(0.0.3), $n.dMatch = 001$. Let n_p be its parent:* retailer *(0.0). Since the $dMatch$ of n_p's two children are 011 and 001, respectively, we set $n_p.dMatchSet[1]$ = $n_p.dMatchSet[3]$ = $true$. Since 011 subsumes 001 and $n_p.dMatchSet[3]$ = $true$, n is not a contributor. On the other hand, the* store *node with ID 0.0.2 is a contributor.*

To set the values of $dMatch$ and $dMatchSet$ for each group $g = (t, M)$, the procedure $pruneMatches$ performs a post-order traversal of the tree T_g. For each node $n \in T_g$, if n matches a keyword $keyword[j]$, $1 \leq j \leq w$, then we set the j^{th} position of $n.dMatch$ to be 1. If n has children, we further set $n.dMatch$ according to the bitwise OR of the $dMatch$ of n's children. Then we set $dMatchSet(n_p)[num(dMatch(n))] = true$, where n_p is the parent of n.

To identify relevant matches, recall that a match in M is relevant if and only if all its ancestors up to SLCA node t are contributors (Definition 4.9). Equivalently, if a node is disqualified as a contributor, then none of the matches in its subtree can be relevant. The *pruneMatches* procedure performs a pre-order traversal of tree T_g in identifying relevant matches. If a node reached is not a contributor, then we skip its subtree, discarding all its descendant-or-self matches. The matches that remain in the traversal are considered to be relevant and are then output.

Example 4.16 *In the running example, tree T_g consists of the paths in D_2 from $group[0].t$,* retailer*, to each match in $group[0].M$. We perform a pre-order traversal on T_g. We start with outputting the root* retailer*. Then since* name *(0.0.0) is a contributor, it is output. So does its child, a relevant match* Brooks Brothers *(0.0.0.0). Similarly we output the nodes on the path from* retailer *to relevant matches* state *(0.0.2.0) and* Galleria*. When we visit* store

Baseball	
QB_1	Jim, Abbott, Outfield
QB_2	Jim, Abbott, James, Baldwin, Starting Pitcher
QB_3	store, Abbott, Baldwin
QB_4*	Tigers, Starting Pitcher, surname
QB_4	Tigers, Starting Pitcher, surname
QB_5	Tigers, Starting Pitcher, Outfield, surname
QB_6	Cordero, First Base
QB_7	Cordero, First Base, Tigers
QB_8	1998 Abbott retailer
Mondial	
QM_1	United States, Birmingham, Population
QM_2*	United States, United Kingdom, Birmingham, Population
QM_2	United States, United Kingdom, Birmingham, Population
QM_3	Tasmania, Sardinia, Gotland, Area
QM_4	Ethnicgroups, Chinese, Indian, Capital
QM_5	Mondial, Country, Muslim
QM_6	Country, Muslim
QM_7	Asia, China, Government
QM_8	Organization, Name, Member

Figure 4.3: Part of Query Sets for Testing MaxMatch

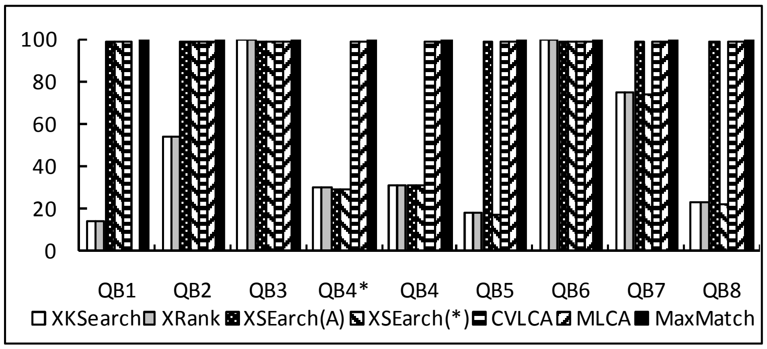


Figure 4.4: Precision of MaxMatch on Baseball Data Set

(0.0.3), since it is not a contributor, we move to the next match in $group[0].M$ *that is not a descendant of* store *(0.0.3). In this example, there is no more matches in* $group[0].M$ *so the process is complete.*

4.5 Experiments

To evaluate the effectiveness of MaxMatch, we tested three metrics: *search quality* measured by precision, recall and F-measure compared with the relevant query results obtained from user studies, *processing time* and *scalability*.

For speed and scalability test, since only XKSearch/SLCA [144] and Timber's im-

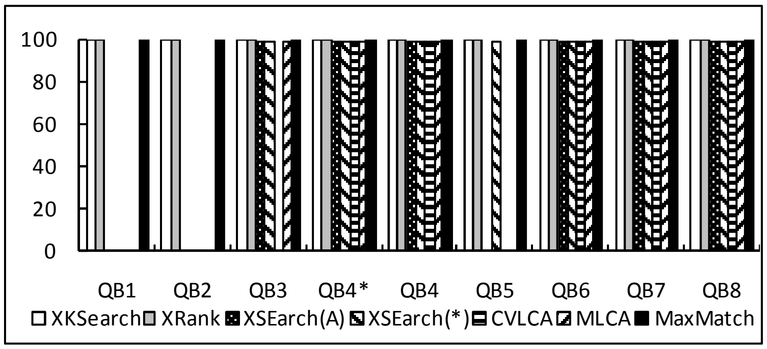


Figure 4.5: Recall of MaxMatch on Baseball Data Set

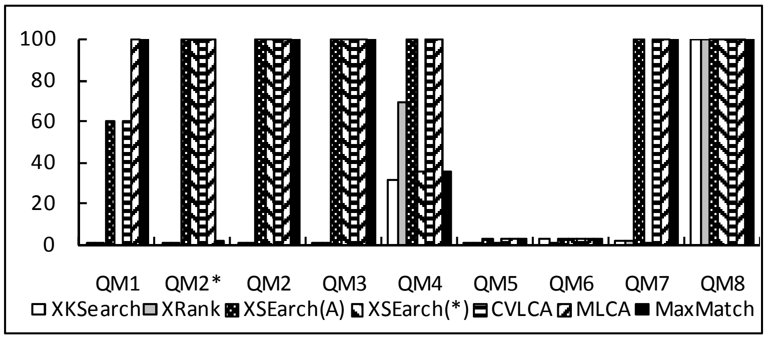


Figure 4.6: Precision of MaxMatch on Mondial Data Set

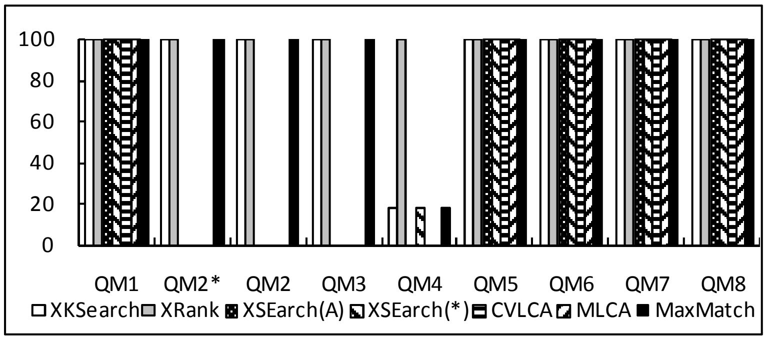


Figure 4.7: Recall of MaxMatch on Mondial Data Set

plementation (MLCA) [86] are available, we only compare with these two approaches and use their query results for quality test. The quality of other approaches of identifying relevant matches (XRank [56], XSEarch [43] all-pairs and star semantics and CVLCA [82]) are tested based on the semantics described in those papers.

Equipment. The experiments are performed on a 3.0GHz AMD Athlon(TM) dual-core CPU running Microsoft Windows Server 2008 Enterprise operating system with 4.0GB memory. The algorithms are implemented in Microsoft Visual C++ 8.0. We use Oracle Berkeley DB [5] as the tool for creating inverted index and Dewey index.

The test data and part of the query sets are shown in Figure 4.3.

Data Set. We have tested two data sets: Baseball and Mondial. Baseball is a data set about the retailers and stores of North American baseball league.[7] Mondial is a world geographic data set.[8]

Query Set. Our queries consist of two parts. First we pick eight distinct queries for each data set, which are shown in Figure 4.3. These queries are chosen to represent a variety of cases, where both tag names and values are used.

To test the validity of data monotonicity and data consistency, the queries in Figure 4.3 with a '$*$' in their names are evaluated on the modified data sets. QB_4* is evaluated on the Baseball data set after removing a *Starting Pitcher* node within retailer *Tigers*. QM_2* is evaluated on the Mondial data set after removing the *Birmingham* node of United Kingdom.

To test the validity of query monotonicity and query consistency, we design some query pairs, such that one contains one more keyword than the other, including QB_4 and QB_5, QB_6 and QB_7, QM_1 and QM_2, QM_5 and QM_6.

In addition, ten test queries for each data set are issued by students who are not involved in this project, which are omitted due to limited space.

[7]http://www.ibiblio.org/xml/books/biblegold/examples/baseball/.
[8]http://www.cs.washington.edu/research/xmldatasets.

Search Quality

To measure search quality, we need to assess the relevance of query results. We have conducted user surveys on the test data and queries to set the ground truth of relevant matches of each query. Thirteen students participated in the survey. Each participant was asked to specify relevant matches for each query. The ground truth of relevant matches are the ones selected by at least seven out of the thirteen users.

Perception of the Proposed Properties. For the queries designed to test the proposed properties, user study results confirm our intuition. Whenever we add a new keyword to a query, the number of relevant query results should not increase; and if a delta result tree exists, it should contain at least one match to the new keyword. Whenever we add a new data node, the number of relevant query results should not decrease; and if a delta result tree exists, it should contain the new data node.

Quality of MaxMatch. We compared the search quality of MaxMatch with XK-Search [144], XRank [56], XSEarch [43] (including all-pair semantics and star semantics), CVLCA [82] and MLCA [86].

To measure the search quality, we use *precision*, *recall*, and *F-measure*. Precision measures the percentage of the output nodes that are desired, recall measures the percentage of the desired nodes that are output. F-measure is the weighted harmonic mean of precision and recall, and is computed as:

$$F = \frac{(1+\alpha^2) \times precision \times recall}{\alpha^2 \times precision + recall} \tag{4.1}$$

The precision and recall of each approach on the test queries in Figure 4.3 are shown in Figure 4.4 - 4.7 respectively.

Now we analyze each approach on the test queries. XKSearch always has a perfect recall except QM_4 (which will be discussed later), but generally has a very low precision as it outputs all the match nodes under each SLCA node, which are not necessarily relevant. Take QB_4 for example, SLCA outputs the *surname* nodes of all the stores, including the

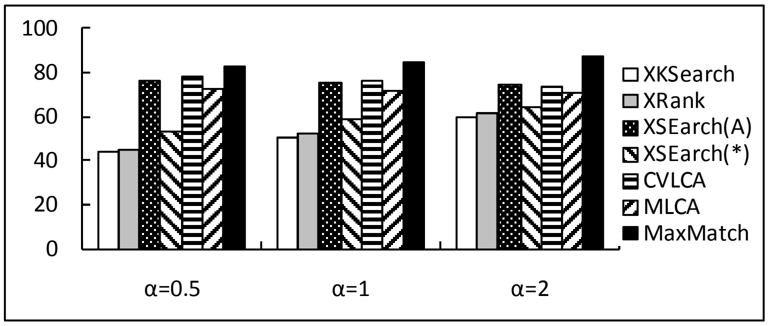


Figure 4.8: F-measure of All 36 Test Queries

ones who are not *Starting Pitcher*s, and therefore has a low precision.

The query result of XRank, for a given query and document, is a superset of that of XKSearch. For many test queries, XRank has the same query results as XKSearch, such as QB_4, where the *surname* nodes of all the stores (including the ones who are not *Starting Pitcher*s) are output. It has a different precision and recall from XKSearch for queries like QM_4. The semantics of QM_4 is to find capitals of the countries that have ethnicgroups Chinese or Indian. In the data, some countries have both Chinese and Indian, some have one of them, and some have neither. XKSearch outputs the countries that have both. On the other hand, besides outputting all such countries, XRank additionally considers the LCA that contains all query keywords of the remaining keyword matches, which is the document root. Therefore all keyword matches are considered to be relevant, and the information of the countries that have either ethnicgroups of Chinese or India are output, achieving a better recall than XKSearch. Both approaches have a low precision as XKSearch outputs all *ethnicgroups* nodes within the countries that have both Chinese and Indian, while XRank outputs all *capital* and *ethnicgroups* nodes in the entire data.

XSEarch star semantics has the same precision and recall as XKSearch on some queries such as QB_4 and QM_5. Take QB_4 for example. Keyword *Tigers* is a retailer name, and all *Starting Pitcher* and *surname* nodes within the retailer are interconnected with *Tigers*, and are considered relevant, which gives the same set of relevant matches as XKSearch. XSEarch star semantics has a zero recall on several queries, for example, QB_1. In the data, the store named Jim Abbott is not an outfield, and the semantics of the query is to find the stores that play with Jim Abbott whose position is outfield. However, since the matches of *Jim* and *Abbott* are not interconnected with those of *Outfield*, XSEarch star semantics gives an empty result, leading to zero recall.

XSEarch all-pair semantics and CVLCA have zero recall on more queries than XSEarch star semantics, as they require that in a qualified pattern match, every two nodes must be interconnected. XSEarch all-pair and CVLCA have the same result for test queries except QB_3. For this query, XSEarch all-pair semantics correctly identifies relevant matches, as the match nodes are two *store* nodes which are interconnected. However, CVLCA generates empty result as the matches to *Abbott* and *Baldwin* are not interconnected.

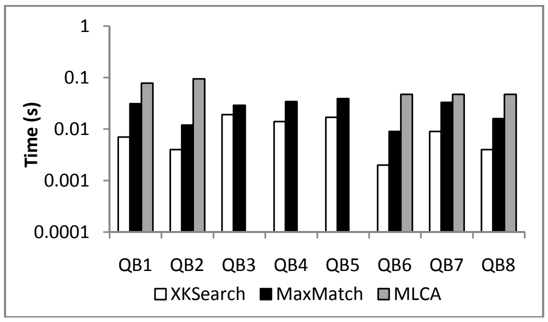
The query result output by MLCA, for a given query and document, is a subset of that output by XKSearch. Some query results generated by MLCA are the same as XKSearch, such as QB_6 and QM_6. MLCA has a better precision on many queries such as QB_4, where irrelevant *surname* matches of stores other than *Starting Pitcher* that are output by XKSearch are avoided. MLCA has the worse recall for queries like QB_5, where an empty result is returned since there is no *surname* node that is meaningfully related with both *Starting Pitcher* and *Outfield*. In general MLCA has a high precision and a low recall.

MaxMatch has perfect precision and recall for most of the test queries, especially when the data structure is simple and regular, such as the Baseball data set. However, there are queries it can further be improved.

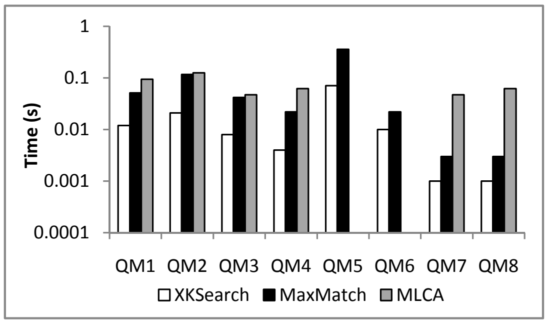
For QM_2*, since the *Birmingham* of United Kingdom is removed from the data, all *population* nodes under country United Kingdom is output by MaxMatch, leading to a low precision.

For QM_4, according to user study, the user would like to find the capitals of the countries that have ethnic group of Chinese or Indian. MaxMatch has a low recall because it only outputs the captials of the countries that have both Chinese and Indian ethnic groups. MaxMatch also has a low precision as it outputs not only the capital of each country, but also the capital of each province, which is irrelevant to the query.

For QM_5, the user would like to search the countries with religion *Muslim*. However,



(a) Processing Time of MaxMatch on Baseball



(b) Processing Time of MaxMatch on Mondial

Figure 4.9: Processing Time of MaxMatch

in the Mondial data set, a *Province* and a *City* node can have a child *Country*. All the approaches output these *Country* nodes of a *Muslim* country and thus have a low precision. In fact, for QM_5, XKSearch, XRank and XSEarch star semantics output all the *Country* nodes in the data set, which leads to a lower precision.

QM_6 searches for the country with Muslim. Since each province and city has a country child, and all these country nodes are output by all approaches, thus they all suffer from low precision.

Figure 4.8 shows the F-measure with all 36 test queries (including the 20 queries issued by users and 16 queries in Figure 4.3) with $\alpha = 0.5$, 1 and 2. As we can see, overall MaxMatch outperforms other approaches.

Processing Time

The XML keyword search systems XSEarch, XRank and CVLCA are not available online. We have implemented XKSearch and MaxMatch, both of which use the approach in [144] for computing SLCA of keyword matches. We use Timber [8] with default settings for identifying relevant matches using MLCA semantics.

We use the Baseball data of size 1014KB, and a portion of the Mondial data of size 515KB.[9] The processing times of XKSearch, MLCA, and MaxMatch over the queries on

[9]We did not use a larger data set because Timber reported error on larger data when evaluating most

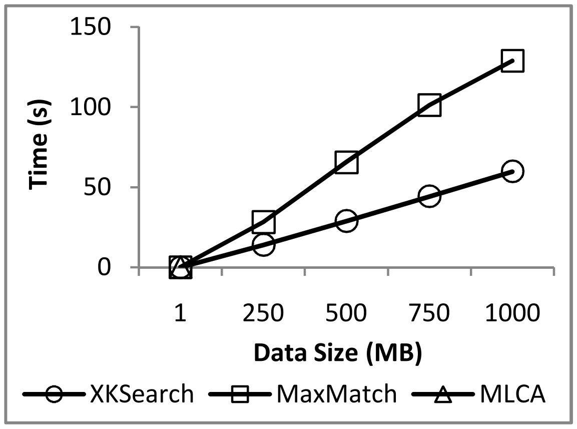


Figure 4.10: Scalability of MaxMatch

Baseball and Mondial data sets are shown in Figure 4.9 (a) and (b), where the y-axis is in logarithmic scale.

Both XKSearch and MaxMatch retrieve keyword matches and compute the SLCA nodes. XKSearch additionally needs to output all the matches in the trees rooted at SLCA nodes. MaxMatch additionally needs to group matches according to their SLCA ancestors, traverse subtrees rooted at SLCA nodes, identify relevant matches according to the descendant matches of each node in these subtrees. Therefore MaxMatch has processing overhead compared with XKSearch. For queries like QM_6, there are a lot of match nodes that are not descendants of any SLCA node, therefore the time for pruning irrelevant matches in the subtrees rooted at SLCA nodes is small, and the keyword match retrieval and SLCA computation time is the bottleneck. In this case, the processing time of XKSearch and MaxMatch are close. For queries where most of the keyword matches have an ancestor SLCA node, MaxMatch is slower than XKSearch. MLCA retrieves pattern matches to a query, and then checks whether the nodes in the pattern matches are pairwise meaningfully related, which is usually expensive.

Scalability

We have tested the scalability of MaxMatch, XKSearch and MLCA with respect to the increase of XML data size. The result is shown in Figure 4.10, which represents the process-

queries.

ing of QB_1 on the Baseball data with size increased from 1MB to 1GB by replicating the original data set. It can be observed that the processing times of MaxMatch and XKSearch both increase linearly with the increase of the data size.

In summary, MaxMatch achieves improved search quality compared with existing XML keyword search engines in identifying relevant matches with efficiency.

4.6 Summary

In this chapter we addresses an open problem of reasoning about XML keyword search algorithms. We take an axiomatic approach and have identified the properties that an XML keyword search algorithm should ideally possess in identifying relevant matches to keywords. Monotonicity states that data insertion (query keyword insertion) causes the number of query results to non-strictly monotonically increase (decrease). Consistency states that after data or query keyword insertion, if an XML subtree becomes valid to be part of new query results, then it must contain either the new data node or a match to the new query keyword. We have shown that these properties are non-trivial, non-redundant, and satisfiable.

We have proposed MaxMatch, a novel semantics for identifying relevant matches and an efficient algorithm to realize this semantics, which is the only known algorithm that satisfies all properties. Experimental studies have verified the intuition of the properties and shown the effectiveness of our approach.

Chapter 5

IDENTIFYING RETURN INFORMATION

5.1 Motivation and Goal

Recall that inferring structures for a keyword query on XML involves three tasks: inferring relevant keyword matches (analogous to the “for” and “where” clauses in an XQuery), inferring the information that should be returned (analogous to the “return” clauses in an XQuery), and inferring how the nodes to be returned should be connected to compose the query results (analogous to inferring how the XQuery should be structured). We have discussed the first task in Chapter 4. In this chapter, we focus on the second task: inferring return information.[1] In other words, besides the relevant keyword matches and the paths connecting them, other nodes in the XML data can be relevant to the query, and should be identified and returned.

As mentioned in Section 4.1, several recent attempts have been made on supporting keyword search on XML documents, however, all of them focus on the problem of identifying relevant keyword matches, and identifying return information is an open challenge. There are two baseline approaches for determining return nodes adopted in the existing works. One is to return the subtrees rooted at selected LCA nodes [56, 43, 144], referred to as *Subtree Return*. Alternatively, we can return the paths in the XML tree from each LCA node to its descendants that is a relevant match, as described in [21, 61], which are referred to as *Path Return*. However, neither approach is effective in identifying return nodes.

Let us look at some sample queries listed in Figure 4.1. For Q_6, it is likely that the user is interested in the information about *Brooks Brothers*. Both *Subtree Return* and *Path Return* first compute the LCA of the keyword matches and find relevant matches, which is the node with ID 0.0.0.0, then output the node *Brooks Brothers* itself. However, to echo print the user input without any additional information is not informative. Ideally, we would like to return the subtree rooted at the *retailer* node for information about *Brooks Brothers*.

[1]In addition to the returned information, we also return the relevant matches, serving as evidence that the returned information is relevant.

Now let us consider Q_1 and Q_7. By issuing Q_7, the user is likely to be interested in information about the store whose name is *Galleria* and which is located in *Texas*. Therefore the subtree rooted at the *store* node with ID 0.0.2 is a desired output. In contrast, Q_1 indicates that the user is interested in a particular piece of information: the *state* of *Galleria*.

As we can see, an input keyword can specify a *predicate* for the search, or specify a desired *return* node. However, existing approaches fail to differentiate these two types of keywords. In particular, *Path Return* approach returns the paths from the LCA node *store* (0.0.2) to *Galleria* and to *Texas* for Q_7, and the path from *store* to *Galleria* and *state* for Q_1, respectively. On the other hand, since Q_1 and Q_7 have the same LCA node *store* (0.0.2), *Subtree Return* outputs the subtree rooted at this node for both queries, although the user indicates that only the *state* information is of interest in Q_1.

Now let us look at a more complex query Q_8, intending to find information about the *retailer* who sells *apparel* and has stores in *Texas*. *Subtree Return* approach outputs the whole tree rooted at *retailer* (0.0), and requires the user him/herself to search the relevant stores in this big tree. On the other hand, *Path Return* approach outputs the path from *retailer* to *apparel* and to *Texas*, without providing any additional information about the retailer and its stores.

Finally, if the input keywords match a big subtree, displaying the whole tree at once can be overwhelming to the user. It is more desirable to output the most relevant information first, and then provide *expansion links*, so that the user can click and browse detailed information. For example, to process Q_6, we do not directly output all stores of *Brooks Brothers*, but output an expansion link which, upon click, provides information about all the stores of *Brooks Brothers*. Each store also has expansion links to the clothes it sells. The question is how to identify the information to be displayed at the beginning, and subsequently, at each expansion step.

As we can see from the above sample queries, existing approaches fail to effectively identify relevant return nodes. Sometimes they suffer low precision, such that users need to browse and screen relevant information from large output themselves, which can be time consuming. Sometimes they have low recall, such that users are not able to get informative results.

The only works that have considered the problem of identifying return nodes is [63, 75]. Both of them require schema information. In addition, [63] requires a system administrator to split the schema graph into pieces, called *Target Schema Segments (TSS)* for search result presentation. [75] requires users or a system administrator to specify a weight of each edge in the schema graph, and then each user needs to specify a *degree constraint* and *cardinality constraint* in the schema to determine the return nodes.

There are several desirable features for determining the return nodes that an XML keyword search system should achieve. First, though schema information can be used whenever possible, its presence should be optional, since XML data may not have an associated schema. Second, it is important for the system to automatically infer return nodes without eliciting preference from users and system administrators due to two reasons: the users who do not issue structured queries are probably unwilling or unable to specify the output schema; and it is hard for a system administrator to specify return nodes that reflect individual user needs. Furthermore, return node identification should consider not only data structure, but also keyword patterns, as shown in Q_1 and Q_7.

In this chapter, we present techniques that allow users to search information in XML documents by keywords, and identify meaningful return nodes as exemplified in the above sample data and queries without user solicitation. To achieve this, we analyze both XML data structure and keyword patterns. We differentiate three types of information represented in XML data: entities in the real world, attributes of entities, and connection nodes. We also categorize input keywords into two types: the ones that specify search predicates, and the ones that indicate return information that the user is seeking. Based on data and keyword analysis, we discuss how to generate return nodes, which can be explicitly inferred from keywords, or dynamically constructed according to the entities in the data that are relevant to the search. Finally, data nodes that match predicates and return nodes are output as query results with optional expansion links.

5.2 Identifying Explicit and Implicit Return Information by Keyword Match and Data Analysis

As we have discussed in Chapter 1, besides relevant keyword matches, other data nodes that do not match keywords may also be relevant to the user's query, which correspond to the *return* clause in XPath or *SELECT* clause in SQL. For example, if the user wants to search the information of *Galleria* using a structured query, she would likely select all attributes of the *store* table and possibly other tables such as *clothes*. However, if she uses keyword search she may simply use a single keyword *Galleria*. Besides returning the relevant matches to *Galleria*, it is desirable to also return the general information related to the stores named *Galleria*. In this chapter we focus the discussion on how to identify meaningful return information given a group of relevant matches.

Analyzing XML Data Structure

To decide what information should be returned, we need to understand the roles and relationships of nodes in the data. The information in XML documents can be recognized as a set of real world entities, each of which has attributes [2], and interacts with other entities through relationships. This mimics the *Entity-Relationship* model in relational databases.

For example, for the XML data in Figure 1.3, conceptually we can recognize three types of entities: *retailer*, *store* and *clothes*. Each type of entity has certain attributes. *retailer* has *name*, *product*, *store* has attributes *state*, *city* and *name*, etc. The relationships between entities are represented by the paths connecting them. For example, a *retailer* has one or more *store*s.

We believe that by issuing a query a user would like to find out information about entities along with their relationships in a document. Therefore to determine the search result, we should consider the entities in the document that are related to the input keywords.

For example, consider query Q_6 in Figure 4.1 that searches *Brooks Brothers*. Very

[2]In the rest of the paper, *attribute* refers to the one defined in an ER-model, rather than the one defined in XML specification [141].

likely the user would like to find out the information about the real world entity that *Brooks Brothers* corresponds to. Therefore, we first identify that the *retailer* node with ID 0.0 is the corresponding entity of *Brooks Brothers*. Then, we output the information of this *retailer* entity. The attributes associated with the *retailer* entity are considered as the most important and relevant information, and are output at the first place. The relationship between *retailer* and *store* is considered to be of secondary importance. A link is generated for the relationship to *store*, which allows users to click and browse the interacting entities.

The first guideline of identifying return information is to differentiate nodes representing entities from nodes representing attributes, and generate return nodes based on the entities related to the keyword matches.

However, since XML data may be designed and generated by autonomous sources, we do not necessarily know which nodes represent entities, and which represent attributes directly. Next we present heuristics for inferring entities and attributes in two situations: when the schema is available, and when it is absent.

When the schema of an XML document is available, we classify nodes into entities and attributes according to their node relationships. If a node of name n_1 has a one-to-many relationship with nodes of name n_2, then very likely n_2 represents an entity rather than an attribute of n_1. On the other hand, a one-to-one relationship is more likely to introduce an attribute.

In general, we make the following *inferences on node categories*.

Definition 5.1 1. *A node represents an* entity *if it corresponds to a *-node in the DTD.*

2. *A node denotes an* attribute *if it does not correspond to a *-node, and only has one child, which is a value.*

3. *A node is a* connection *node if it represents neither an entity nor an attribute. A connection node can have a child that is an entity, an attribute or another connection node.*

For example, consider the DTD fragment in Figure 2.1 for the XML data in Figure 1.3. *retailer* is a *-node, indicating a many-to-one relationship with its parent node *retailers*. We infer that *retailer* represents an entity that has a relationship with *retailers*, instead of an attribute of *retailers*. *name*, and *product* are considered as attributes of a *retailer* entity. On the other hand, *merchandises* is not a *-node and it does not have a value child, therefore it is considered as a connection node.

It is worthwhile to notice that the notion of entity used throughout this dissertation is different from the entities declared in XML DTDs. A DTD entity is a variable which defines a shortcut to standard text or special characters. On the other hand, the entities in this thesis refer to real world objects similar as the entities in the E-R model.

Though these inferences do not always hold (for example, one person may have multiple attributes of phone number, and the *league* node should be considered as an entity instead of a connection node), they provide good heuristics in the absence of the E-R model.

When the schema information is not available, we infer the schema based on data summarization, similar as [49, 146]. Then we identify the entities and attributes in the data according to the inferred schema. For example, since the *retailer* node can occur more than once within its parent *retailers*, it is considered as a *-node.

Analyzing Keyword Patterns

Besides studying the structure of XML data and inferring inherent entities and attributes presented in the data, we also analyze the pattern of the input keywords to infer search predicate and return node specifications.

Recall Q_7 and Q_1 in Figure 4.1, which have the same set of SLCA nodes and relevant keyword matches. Existing approaches do not differentiate these two queries while generating results. However, different patterns of these two queries imply different user intensions. Q_7 searches information about *Galleria* located in *Texas*. Q_1 searches the *state* information of *Galleria*.

The second guideline of the XSeek system is to take keyword patterns into consideration when generating search results, by classifying input keywords into two categories: search predicates and return nodes.

1. Some keywords indicate *predicates* that restrict the search, corresponding to the *where* clause in XQuery or SQL.

2. Some keywords specify *return nodes* as the desired output type, corresponding to the *return* clause in XQuery or the *select* clause in SQL.

Since keywords do not have any structure, the immediate question is how we should infer predicates and return nodes. To achieve this, we first differentiate data *types* from data *values*. If the XML schema is available, we can obtain type information directly. Otherwise, we use node *names* to indicate data types. Recall that in a structured query language such as XQuery or SQL, typically a predicate consists of a pair of type and value, while a return clause only specifies data types without value information (whose values are expected to be query results). For example, consider an SQL query: *select state from DB where name = "Galleria"*.

Based on this observation, we make the following *inference on keyword categories* for each match group.

1. If an input keyword k_1 has a relevant match which is a node name (type) u, and there does not exist an input keyword k_2 who has a relevant match which is a value v, such that u is an ancestor of v, then we consider k_1 as a *return node*.

2. A keyword that is not a return node is treated as a *predicate*. In other words, if a keyword has a relevant match which a node value, or is a node name (type) that has a value descendant matching another keyword, then we consider this keyword as a predicate.

For example, in Q_7, *Texas* is considered as a predicate since it matches a value. Similarly, *Galleria* in both Q_7 and Q_1 are considered as a predicate. *retailer* in Q_8 is also inferred as a predicate since it matches a name (0.0) which has a descendant value node (0.0.0.0) that matches another keyword *Brooks Brothers*. On the other hand, *state* in Q_1 is considered as a return node since it matches the name of a node (0.0.2.0, 0.0.3.0), neither of which has any descendant value node matching another keyword in Q_1. Similarly, *city* in Q_3 is also treated as a return node.

However, note that an ill-designed XML may invalidate the heuristics we have proposed, such as an XML document that converts all text values in Figure 1.3 into tag names with dummy value children. Such an XML document will affect the correctness of our inference of node and keyword categories. Therefore, in this thesis we assume that an XML document is reasonably designed with meaningful and proper tag names and values.

Generating Search Results

Once we have identified the inherent entities and attributes in the data and the keyword patterns according to the previous discussion, we generate search results accordingly.

The third guideline of XSeek is to output data nodes that match query predicates and return nodes as search results.

Outputting Predicate Matches. One difference between performing keyword search and processing a structured query is that, besides outputting the data matching the potential return nodes, we should also output the data matching predicates. Since in face of the inherent ambiguity of keyword search, the user often would like to check the predicate matches and make sure that the predicates are satisfied in a meaningful way. Therefore, the paths from the SLCA nodes[3] to each match in the XML subtrees will be output as part of search results, indicating how the keywords are matched and how the matches are connected to each other. For example, for Q_1, even the user is only interested in the *state* information, we also output the path from the SLCA *store* (0.0.2) node to the predicate match *Galleria*.

Identifying Return Nodes. Return nodes can be *explicit* or *implicit*. For some queries, explicit return nodes can be inferred from the input keywords as we have discussed

[3]In fact, the paths start from the master entities as defined later.

before. For example, *state* is an explicit return node in Q_1. For other queries, all the input keywords are considered as predicates, and no return nodes can be inferred from the keywords themselves, such as Q_7. In this case, we believe that the user is interested in the general information about the entities related to the search. We define master entity and relevant entity in the following, and consider them as implicit return nodes when the input keywords do not have explicit return nodes specified.

Definition 5.2 *If an entity e is the lowest ancestor-or-self of the SLCA node in a group g, then e is named as the* master entity *of group g. If such an entity e can not be found, the root node of the XML tree is considered as the master entity.*

Since SLCA is the lowest common ancestor of all the matches in a group, master entity is the lowest common entity ancestor of them. All the information that is considered to be relevant to the query is in the subtree rooted at the master entity.

Definition 5.3 *If an entity e is an ancestor-or-self of a keyword match v in a group g, and it is a descendant-or-self of the master entity, then e is a* relevant entity *with respect to v for the group g.*

In Q_7, since both *Galleria* and *Texas* are inferred as predicates, there is no explicit return nodes specified in the keywords. We first identify the *store* node (0.0.2) as the SLCA node. It is in fact the master entity, and the relevant entities are *store* and *clothes*. Therefore *store* and *clothes* are treated as the implicit return node for Q_7.

Outputting Return Nodes Based on Node Categories. After we have identified explicit or implicit return nodes, the next question is how to display them appropriately. The data nodes that match return nodes will be displayed according to their categories: attributes, entities, and connection nodes. To output an attribute, we display its name and value. On the other hand, the subtrees rooted at entities and connection nodes can be big. Rather than outputting the whole subtree at once, it is often more user-friendly to display the most relevant information at the first stage with expansion links to less relevant information. Then the user may click links and browse for more details. The question is what information should be returned at the first stage. For an entity or a connection node, we first output its name. For an entity, we additionally output all of its attributes. Then we generate a link to each group of children entities that have the same name (type), and a link to each child connection node (except those who have descendant matches to keywords, which will be output explicitly).

For example, consider Q_1. We infer *store* as an implicit return node since it is a relevant entity and no return node is specified in the keywords. We display its name *store*, the names and values of its attributes *state*, *city* and *name*. Then we generate an expansion link to its connection child *merchandises*.

In summary, we make the following *inference on search result generation*.

1. We infer return nodes either explicitly from the input keywords by analyzing keyword patterns, or implicitly by considering both keyword predicates and relevant entities in the data.

2. The data nodes that match return nodes are output based on their node categories: attributes, entities and connection nodes.

3. Besides outputting the matches to return nodes, data nodes that match search predicates are also output such that the user can verify the meaning of the matches.

5.3 Algorithms

After we have discussed the semantics of XSeek, we will present the algorithms that process keyword searches on XML data and achieve the semantics efficiently.

Indexes

To efficiently retrieve the information of a node with its Dewey ID, XSeek uses a similar Dewey Index as used for MaxMatch, except that each node contains more information, which is the category of the node. The Dewey Index is structured as B+ tree.

The algorithm $KeywordSearch$ is presented in Figure 5.1, which identifies and outputs meaningful return information for XML keyword search. There are two stages in the process. First, the $findRelMatch$ procedure retrieves all the XML nodes matching the input keywords which are deemed relevant. We adopt MaxMatch for this purpose. Then the procedure $genResult$ generates search results for each group. Next we will discuss procedure $genResult$ in detail. We use Q_3 on the XML data D_2 in Figure 4.1 which intends to search for the city of *Galleria* and *West Village* for retailer *Brooks Brothers*.

Example 5.1 *We first adopt MaxMatch for identifying relevant matches. For Q_3, there is on SLCA node* retailer *(0.0), and the relevant matches are* Brooks Brothers*,* Galleria*,* West Village*,* city *(0.0.2.1) and* city *(0.0.3.1), in the order of the Dewey label, as output by MaxMatch.*

After we find the relevant matches, we check whether there are explicit return nodes in each group. Since each group of relevant matches are ordered by Dewey ID, we can do this by a single traversal. If a relevant match, $group[i][j]$ is a node name, and it is not an ancestor of $group[i][j+1]$, then it means $group[i][j]$ does not have a descendant relevant match, therefore it is considered as a return node, and we set $retSpecified[i]$ = true for this group.

Generating Results. Once we have the SLCA nodes and the groups of relevant matches, we generate search results by outputting data nodes that match search predicates and return nodes. Note that implicit return nodes will be inferred if no explicit return nodes are specified.

In the following, we use a *group of matches* to refer to a set of matches which are descendants of the same SLCA node.

We start by retrieving the master entity of a group using procedure $findEntity$, which accesses the ancestors of a SLCA node in order until an entity or the XML tree root

```
KeywordSearch($keyword[n], indexes$)
1: $SLCA, group \leftarrow findRelevantMatch(keyword, indexes)$ {adopting MaxMatch}
2: for each $SLCA[i]$ do
3:     $retSpecified \leftarrow$ false
4:     for each $group[i][j]$ do
5:         if $group[i][j]$ is a tag name and $group[i][j]$ is not an ancestor of $group[i][j+1]$ then
6:             $retSpecified[i] \leftarrow$ true
7:     $currMatch \leftarrow group[i][1]$
8:     $genResult(findEntity(SLCA[i]))$

genResult($v$)
1: if $(v.name = currMatch)$ then
2:     {v is an explicit return node}
3:     $outputRet(v, explicit)$
4: else if $retSpecified[j] = false$ and $isEntity(v)$ then
5:     {v is an implicit return node}
6:     $outputRet(v, implicit)$
7: else if $isAttribute(v)$ then
8:     output $v.name$ and $v.value$
9: else
10:    output $v.name$
11: $u$ = first child of $v$
12: while $(currMatch \neq null)$ and $(u \neq null)$ do
13:    if $u$ is an ancestor-or-self of $currMatch$ then
14:        $genResult(u)$
15:        if $u.name = currMatch$ or $u.value = currMatch$ then
16:            $currMatch++$ {move to the next match in $group[j]$}
17:    $u \leftarrow v$'s next child that is either an attribute or an ancestor-or-self of $currMatch$

outputRet($v$)
1: if $isAttribute(v)$ then
2:     if $v.name$ and $v.value$ don't match a keyword then
3:         {Otherwise, $v.name$ or $v.value$ will be output by function $genResult$}
4:         output $v.name$ and $v.value$
5: else
6:     {$v$ is an entity or connection node}
7:     output $v.name$, and $v$'s attribute children that don't match a keyword
8:     for each of $v$'s children entities with distinct name and connection nodes $w$ do
9:         if $type = explicit$ then
10:            generate a link to $w$
11:        else
12:            if $w$ doesn't have a descendant match then
13:                generate a link to $w$

findEntity($v$)
1: for each node $u$ along the path from $v$ to the root node do
2:     if $isEntity(u)$ then
3:         return $u$
```

Figure 5.1: Identifying Meaningful Return Information

is reached.

The $genResult$ procedure navigates the paths from the master entity to each match in a group, identifies and outputs the matches to predicates and explicit or implicit return nodes. Initially $genResult$ is invoked on the master entity v, and then it is recursively invoked on the attribute children of v and the children of v that are on the path from v to a match in the group, in document order. If return nodes are not explicitly specified ($retSpecified = false$), then an entity node v on those paths (i.e. a relevant entity) is considered as an implicit return node. In this case, v's attribute children and a link to each distinct name (type) of v's children that are entities or connection nodes and don't have descendant matches is displayed. Eventually, $genResult$ is invoked on one of the matches v. If the match is a node name ($v.name = currMatch$), it is considered as an explicit return node ($retSpecified = true$). We invoke procedure $outputRet(v)$, which display v according to its category, an attribute, entity or connection node. After a match is processed, we move to the next match in the group.

Example 5.2 *In the running example, since the SLCA node* retailer *(0.0) is itself an entity, it is the master entity and a relevant entity of this group. The* $genResult$ *procedure is invoked on the master entity. The variable* $currMatch$ *is initialized to be* Brooks Brothers *(0.0.0.0). In the group of relevant matches, the two* city *nodes are node names with no descendant match, therefore they are considered as explicit return nodes, and no implicit return node is considered. After we output the path from* retailer *to* $currMatch$*,* Brooks Brothers*,* $currMatch$ *is moved to the next match in the group, the* city *node 0.0.2.1. Recursively, the path from* retailer *to* city *node 0.0.2.1 is output. Since it is considered as an explicit return node, its corresponding value,* Houston*, is output. The algorithm continues until all relevant matches have been processed.*

5.4 Experiments

To evaluate the effectiveness of XSeek, we compare its performance with two search result generation approaches as introduced in Chapter 1. *Subtree Return* outputs the whole subtree rooted at each SLCA. *Path Return* outputs the paths from each SLCA to the matches

WSU: 4.7MB	
QW_1	course, title
QW_2	course, title, days crs credit sect
QW_3	ECON, 572, place, times
QW_4	CAC, 101
QW_5	42879, title, days
QW_6	42606, TU, TH
QW_7	root, MILES, course
QW_8	ECON
Mondial: 6.0MB	
QM_1	organization, name, members
QM_2	country, population
QM_3	mondial, Africa
QM_4	Belarus, population
QM_5	mondial, country, Muslim
QM_6	Croatia
QM_7	Bulgaria, Serb
QM_8	Group_of_77, members
Auction: 24.5MB	
QA_1	closed_auction, price
QA_2	closed_auction, price, date, itemref, quantity, type, seller, buyer
QA_3	open_auction, person257
QA_4	person0, address
QA_5	closed_auction, buyer, person133
QA_6	person257, person133
QA_7	seller, person179, buyer, price, date
QA_8	seller, 04/02/1999

Figure 5.2: Data and Query Sets for Testing Return Information Identification

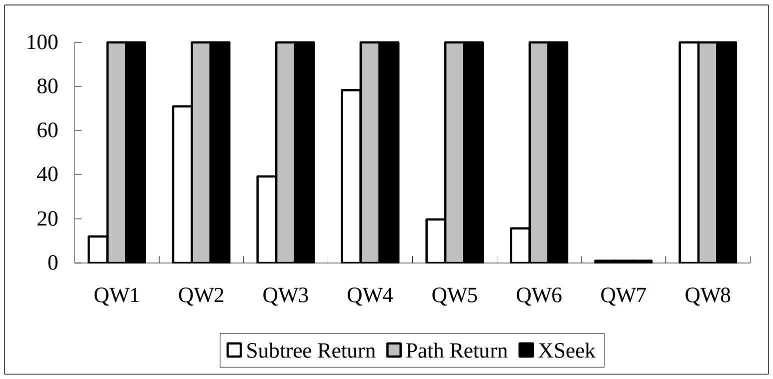


Figure 5.3: Precision of XSeek on WSU Data Set

in the subtree rooted at the SLCA. All three approaches adopt [144] for computing SLCA from keyword matches.

We have tested three metrics to compare these approaches: the *quality* of the search results measured by precision, recall and F-measure, the *speed*, and *scalability*

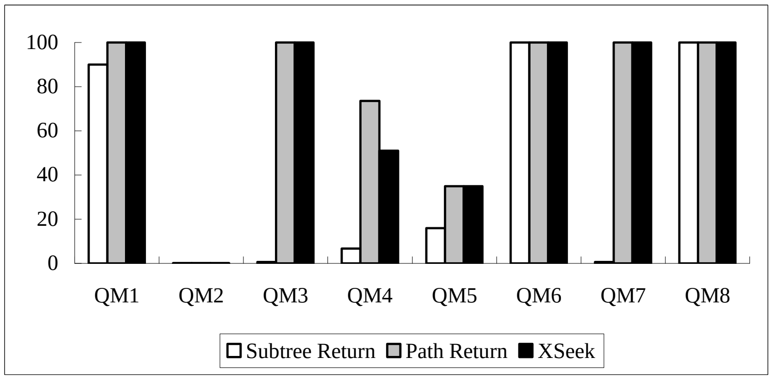


Figure 5.4: Precision of XSeek on Mondial Data Set

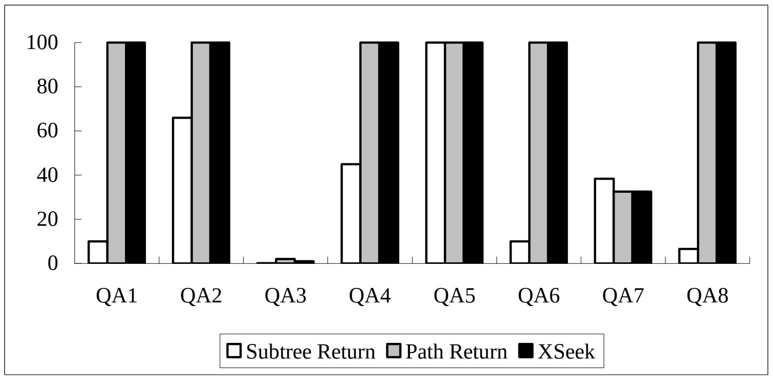


Figure 5.5: Precision of XSeek on Auction Data Set

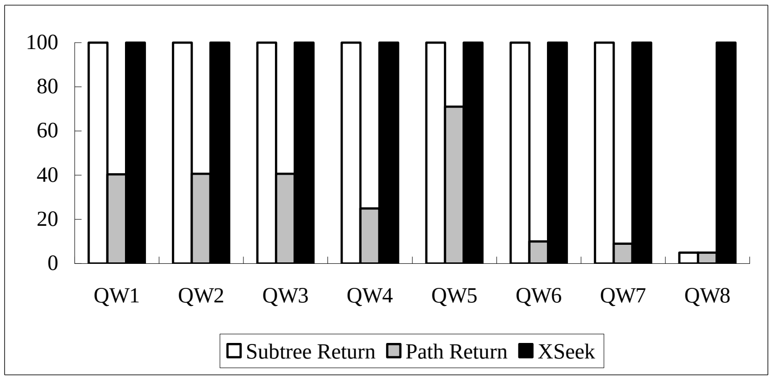


Figure 5.6: Recall of XSeek on WSU Data Set

upon increase of document and query size, and decrease of the height of SLCA.

Experimental Setup

The experiments were performed on a 3.60GHz Pentium 4 machine running Windows XP, with 2GB memory and one 160GB hard disk (7200rpm).

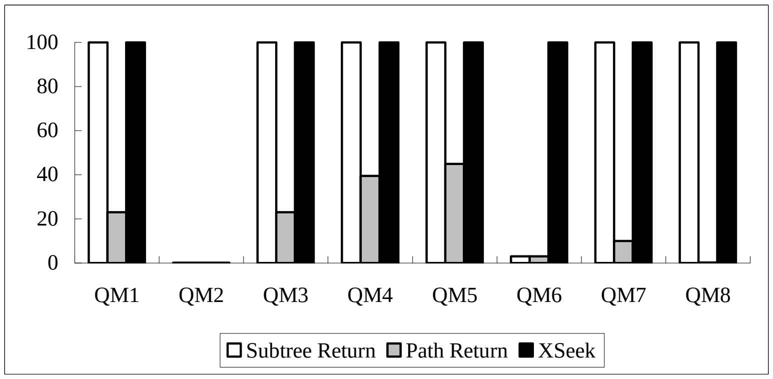


Figure 5.7: Recall of XSeek on Mondial Data Set

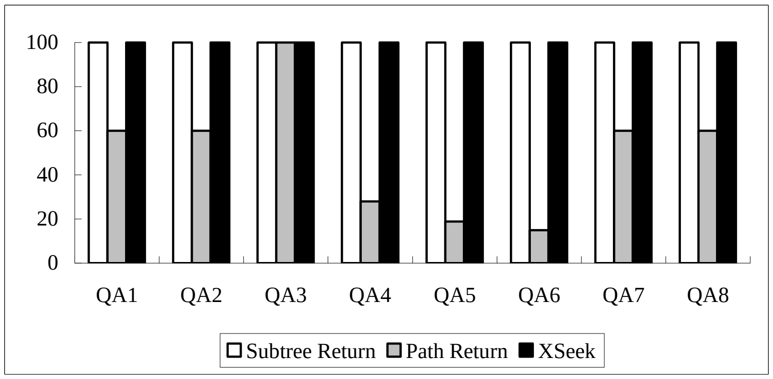


Figure 5.8: Recall of XSeek on Auction Data Set

The experiments are performed on three XML data sets. The characteristics of the data sets and query sets are shown in Figure 5.2.

Data Sets. We have tested three XML data sets, Mondial, WSU[4], and Auction[5]. Mondial is a world geographic database integrated from the CIA World Factbook, the International Atlas, and the TERRA database among other sources. WSU is a course description database. Auction is a synthetic benchmark data set generated by the XML Generator from XMark using the default DTD.

Query Sets. We have tested eight queries for each data set. Among them, the first two only involve node names without values, therefore have relatively low selectivity (and large query result size); the rest six involve node values.

[4] Both Mondial and WSU are available at http://www.cs.washington.edu/research/xmldatasets. We replicated the WSU and Mondial data sets three times to get a larger data size.

[5]http://monetdb.cwi.nl/xml/

Search Quality

To measure the search quality, we use *precision*, *recall*, and *F-measure*, defined as follows:

$$precision = \frac{|Rel \cap Ret|}{|Ret|}, recall = \frac{|Rel \cap Ret|}{|Rel|}$$

where Rel is the set of relevant nodes (i.e. desired search results), Ret is the set of nodes returned by a system, and we use $|S|$ to denote the number of elements in a set S. Precision measures the percentage of the output nodes that are desired, recall measures the percentage of the desired nodes that are output. F-measure is calculated using Equation 4.1. We use α = 0.5, 1 and 2 for computing F-measures.

Each keyword search is expressed as an English sentence, according to which XML fragments are extracted from the original document and are set as ground truth, Rel. Then, we exam the output generated by each system, Ret, and count how many nodes in Ret appear in Rel.

The precision and recall of three approaches on each test data and query set are shown in Figure 5.3 - 5.8, respectively.

As we can see, *Subtree Return* usually has a perfect recall as the whole subtree rooted at each SLCA is returned. The only exceptions are QM_6 *Croatia* and QW_8 *ECON*, both of which consist of a single value, which is in turn returned as the search result. The precision of *subtree return* is usually low as not all the nodes in the subtree rooted at each SLCA are relevant. In particular, consider QM_3 *mondial, Africa*, the whole document tree rooted at *mondial* is returned, even the user is only interested in the information about *Africa* in the *mondial* document.

On the other hand, *Path Return* has the best precision, even perfect precision in many cases, since the matches to predicates and the paths connecting these matches are almost always considered to be relevant. However, it often has a low recall. Consider QM_3 again, no information about *Africa* will be returned except the input keyword themselves.

XSeek has in general a high precision and recall across different queries on the data sets. Its precision is almost the same as *Path Return*, and its recall is almost the same

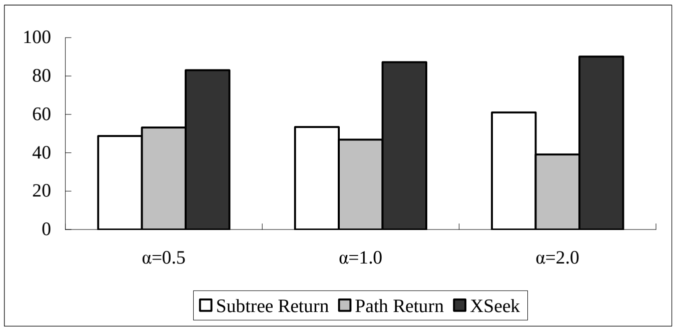


Figure 5.9: F-measure of XSeek All Test Queries

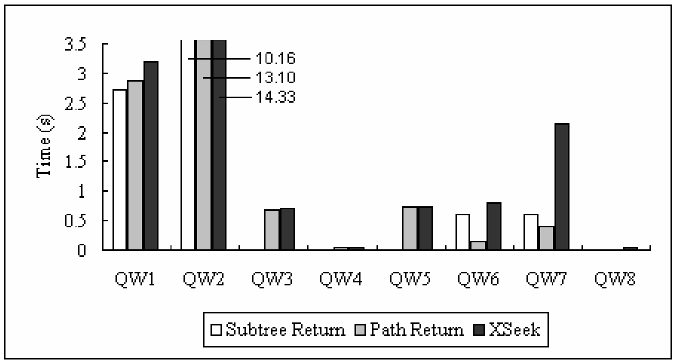


Figure 5.10: Processing Time of XSeek on WSU Data Set

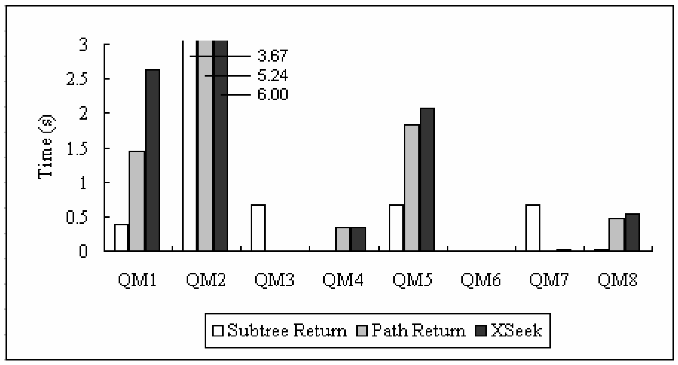


Figure 5.11: Processing Time of XSeek on Mondial Data Set

as *Subtree Return*, or even better for some queries.

However, there are several cases that XSeek needs to be improved. For example, QM_2 *country population* intends to search the population of each country. However, XSeek returns the population of all the cities instead of the population of countries, therefore suffers a zero precision and a zero recall. This is because in Mondial document, each *country* node has *city* children, and each *city* node has an attribute node named *country*. Both *country*

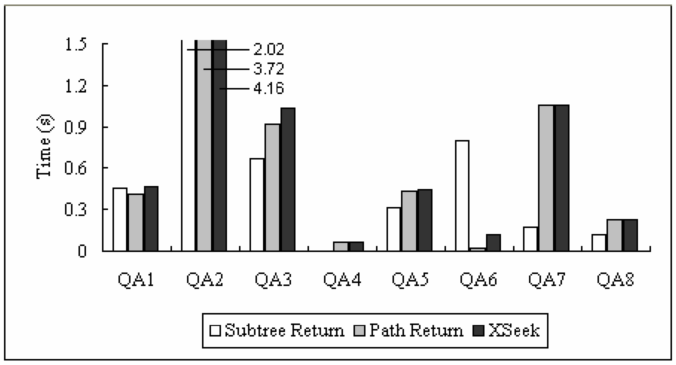


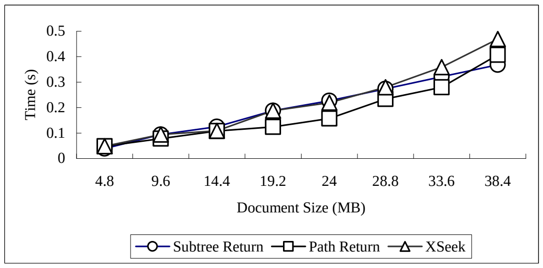
Figure 5.12: Processing Time of XSeek on Auction Data Set

and *city* element nodes have a child *population*. According to the definition of SLCA that these three approaches adopt from [144], *city* instead of *country* is the SLCA. The element nodes matching *country* are not descendants of any SLCA nodes and are discarded, and all three approaches fail. A similar problem exists for QM_4. The only difference is that the SLCA node is the *country* element node that has a descendant *Belarus*, and therefore the population of the country will be output along with city populations.

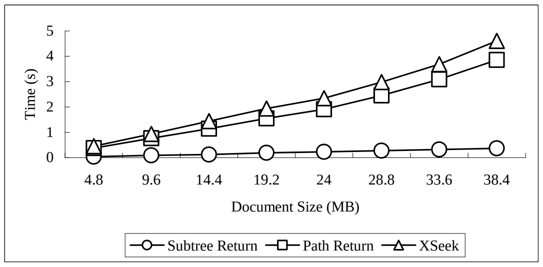
For QW_7, all three systems suffer a low precision. This is because the *root* node is considered as the SLCA, all the matches of *course* and *MILES* are in the same group and are returned even the course instructor is not *MILES*. Similar situation occurs for QM_5 where the root node *mondial* is searched.

To process QA_3, under the *open_auctions* node we find an *open_auction* node which has a descendant *person257*, so this *open_auction* node is a SLCA, and its subtree should be the desired output. Besides, under the *people* node, there's a person named *person257* and there are a lot of *open_auction* nodes, but no *open_auction* node is associated with the person *person257*, so these nodes are not desired. However, since the *people* node is another SLCA of this query, all approaches output the corresponding information within the subtree rooted at *people* (which is very large), leading to low precisions.

In QA_7, the order of the keywords indicates that the user is interested in the *buyer*, *price* and *date* of the auction whose *seller* is *person179*. However, XSeek does not consider the order of keywords, and returns auctions whose *buyer* is *person179*, and therefore has



(a) QA_1



(b) QA_2

Figure 5.13: Processing Time of XSeek with Increasing Document Size

a low precision.

Furthermore, we compute the F-measure of each approach according to the average precision and recall across all the test queries, with parameter $\alpha = 0.5$, 1 and 2, as presented in Figure 5.9. As we can see, XSeek significantly outperforms the *Subtree Return* and *Path Return* approaches.

Processing Time

We have tested the processing time for three approaches. The processing time for all queries is shown in Figure 5.10, Figure 5.11 and Figure 5.12.

All three approaches need to search for keyword matches and compute SLCA from the matches. Then *Subtree Return* requires time to output subtrees rooted at SLCA. *Path Return* requires time to group keyword matches, and then output the paths from each SLCA to matches in the corresponding group. XSeek also groups matches and accesses the path from the master entity to the matches in each group. Furthermore, XSeek needs to determine the predicates and explicit or implicit return nodes based on XML data structure and keyword patterns, and output them accordingly as described in Section 5.3. Since XSeek needs to output at least as much information as *Path Return*, the processing time of *Path Return* is the lower bound of that of XSeek.

The processing time of *Subtree Return* depends on the total number of nodes under all SLCAs, and the processing times of *Path Return* and XSeek depend mainly on the number of matches. For example, QW_1 and QW_2 have the same SLCA, the latter has more keywords and therefore more matches. *Subtree Return* requires the same amount of time to process these two queries, while *Path Return* and XSeek require longer processing for QW_2. We observe the same situation for QA_1 and QA_2. Note that since *Path Return* and XSeek need to additionally group matches compared with *Subtree Return*, they may require more time even their output size is smaller than that of *Subtree Return*. Besides, it is worth mentioning that *Path Return* and XSeek need to merge $KWmatch$ into $Allmatch$, whose processing time is proportional to the $total$ number of matches in a document. On the other hand, *Subtree Return* outputs the subtree rooted at each SLCA, and doesn't process those matches that do not belong to any SLCA. Therefore, when there are a lot of matches that do not belong to any SLCA (e.g. for Q_4 in Figure 4.1, those *retailer* nodes except the first one don't belong to any SLCA), the processing times of *Path Return* and XSeek are relatively longer compared with *Subtree Return*.

As we have discussed, XSeek requires at least as much time for node processing and output as *Path Return*. For queries that XSeek outputs a little more information than *Path Return* such as QW_1, QM_4, QA_4 and QA_5, it takes almost the same amount of time as *Path Return*. On the other hand, for those queries that XSeek outputs a lot more information than *Path Return*, such as QW_6, QM_1, QA_6, XSeek is slower. In summary, XSeek generates search results with improved quality and reasonable cost for most of the test queries.

Scalability

We tested the scalability of XSeek on the Auction data set over three parameters: document size, query size, and the depth of SLCA. Since the complexity and scalability of calculating SLCA were presented in [144], we only test the scalability of grouping matches and generating search results in this section.

Document Size. We replicated the Auction data set of size 4.8MB between 1 and 8 times to get increasingly large data sets. The processing time of queries QA_1 and QA_2 is shown in Figure 5.13. As we can see, the processing time of all three approaches increases linearly when the document size increases. In QA_1, there is only one return node, and the

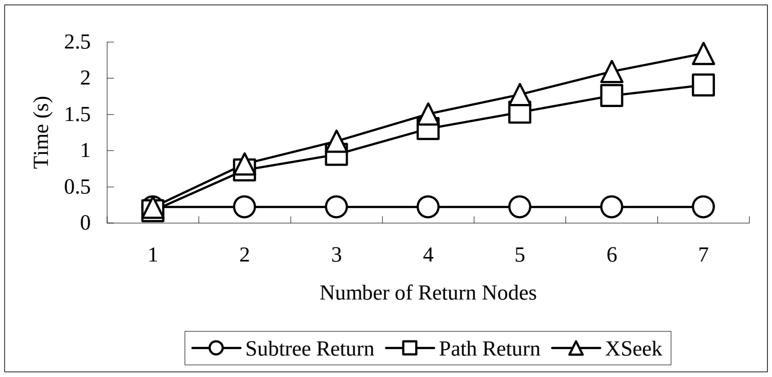


Figure 5.14: Processing Time of XSeek with Increasing Return Nodes

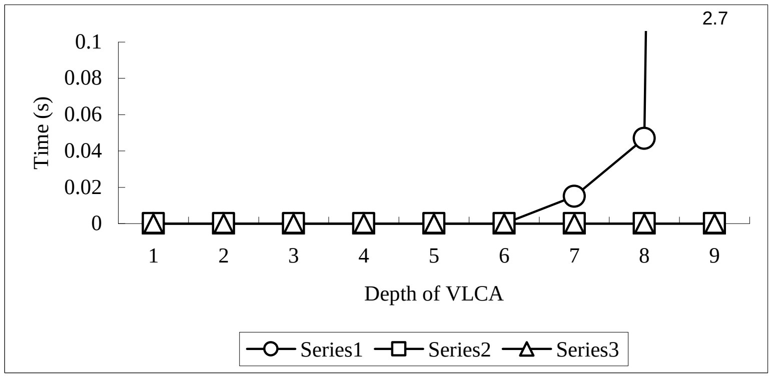


Figure 5.15: Processing Time of XSeek with Decreasing SLCA Depth

processing times of three approaches are close, while in QA_2 where seven return nodes are inferred, *Subtree Return* is the fastest.

Number of Keywords. The experiments were performed on the Auction data set of size 24.5 MB for queries with an increasing number of keywords. We differentiate two different cases.

For queries with an increasing number of return nodes, a constant number of predicates, and a constant depth of SLCA nodes, the performance of three approaches are presented in Figure 5.14. Seven queries are tested, from QA_1: *closed_auction price* to QA_2: *closed_auction price date itemref quantity*
type sellar buyer, adding one keyword each time.

As we can see, since the set of SLCA nodes and the subtrees rooted at SLCAs remain the same across different queries, the result size and therefore the result generation time of *Subtree Return* remains the same across the queries. On the other hand, as the number of return nodes increases, the result size and therefore the result generation times of *Path Return* and XSeek increase linearly. Note that these two approaches have almost the same processing as *Subtree Return* when there is only one return node in the query, but require more time when the number of return nodes is larger than one.

We have also tested queries with an increasing number of predicates, a constant number of return nodes and a constant depth of SLCA nodes. The test queries are constructed by replacing the node names in the queries of the previous test to their corresponding values. The processing time of all three approaches can almost be neglected as there are few value nodes in the data that match the keywords, and therefore the experimental figure is omitted.

Depth of SLCA. The experiments were performed on the Auction data set of size 38.4 MB for queries that have SLCAs of a decreasing depth in the XML data tree. Nine queries are tested, from the SLCAs of depth 9 to the document root, decreasing the depth by one each time. The number of match nodes are the same for all nine queries.

From the experimental results presented in Figure 5.15, we can see that when the depth of SLCA becomes smaller, the size of the subtree rooted at a SLCA increases exponentially, therefore the processing time of *Subtree Return* increases exponentially. On the other hand, the number of output nodes and therefore the result generation times of *Path Return* and XSeek, increase very slowly.

In summary, XSeek has an improved search quality compared with *Subtree Return* and *Path Return* by analyzing XML data structure and keyword patterns without eliciting user specifications. The processing overhead for return node inferences is reasonable. XSeek scales well when the document size and/or the query size increase, or the depth of the SLCAs of matches decrease.

5.5 Summary

In this chapter we present an XML keyword search engine XSeek that addresses an open problem of inferring desirable return nodes in keyword search without elicitation of user preferences and have achieved promising results. we analyze both the XML data structure and keyword patterns, and generate meaningful return nodes accordingly. Data nodes that match predicates and return nodes are output as search results with optional expansion links. Compared with two baseline approaches, XSeek achieves improved precision and recall with reasonable cost and good scalability.

Chapter 6

RANKING FRIENDLY RESULT COMPOSITION

6.1 Motivation and Goal

So far we have addressed two problems in identifying structures for a keyword search: identifying relevant keyword matches (analogous to identifying which nodes should be bound to a variable in an XQuery) in Chapter 4, identifying relevant return information (analogous to identifying which information should be returned in the "return" clause) in Chapter 5. In this chapter, we focus on the problem of how to connect the nodes to be returned and compose results. This is analogous to inferring how to compose query results in the "return" clause in XQuery using variable binding. As discussed in Chapter 1, for keyword query "*men, apparel, store*", there are three possible way to compose the results: (1) each result contains a single retailer with all stores and all clothes in each store; (2) each result contains a single store with all clothes; (3) each result contains a single clothes. As shown in the following examples, although different ways of composing results return the same amount of information in all results, the way of composing results in XML keyword search has crucial effects on result ranking. Example 6.1 illustrates a favorable way of composing results and Examples 6.2 and 6.3 show the undesirable behaviors of existing approaches.

Example 6.1 *Consider query "*men, apparel, store*" on the XML tree in Figure 1.3. After identifying relevant matches (i.e., the match nodes relevant to keywords "men", "apparel", and "store") as well as the relevant return information (the information related to retailer, store and clothes), ideally we should compose the results in such a way that each query result should contain one instance of* store*, along with the related matches to* men *and* apparel *as evidence of its relevance, such as the two query results shown in Figure 6.1. Besides, results should be properly ranked. For instance, many ranking schemes [18, 43, 56, 124, 24] will rank the store that has more men's apparels higher.*

Intuitively, each keyword search has a goal, which is usually the information of a real world entity or relationship among entities, as observed in [36, 38]. We use the term *search target* to refer to the information that the user is looking for in a query, and *target*

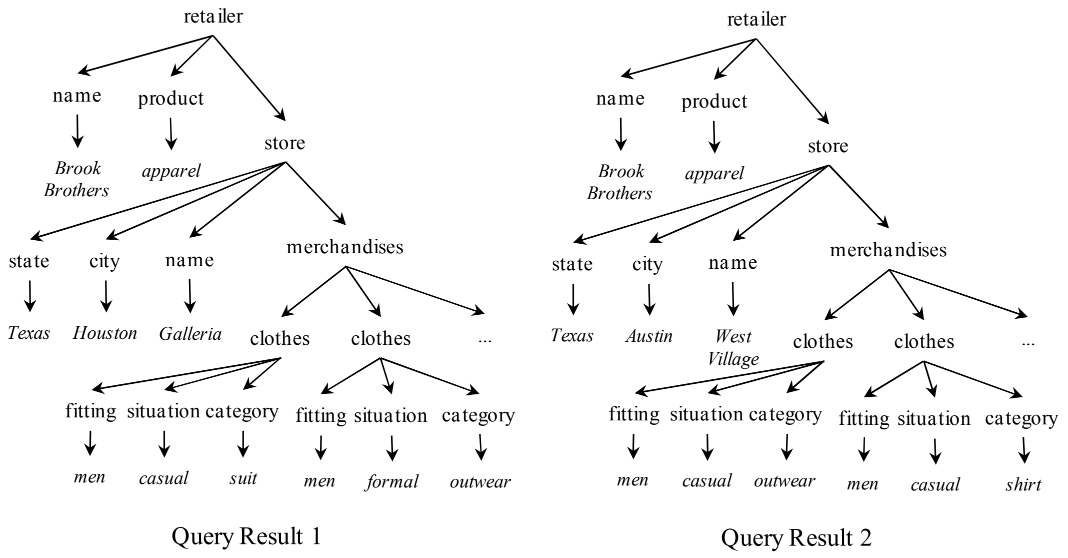


Figure 6.1: Desirable Query Results of Q_9

instance to denote each instance of the search target in the data. Each desirable query result should have *exactly one target instance* along with all associated evidence, so that ranking and top-k query processing can be based on target instances, and thus become meaningful. Specifically, query results of an XML keyword search should be: (1) Atomic: it should consist of a single target instance; (2) Intact: it should contain the whole target instance as well as all its supporting information.

However, the query result composition methods adopted in existing XML keyword search engines, named as *Subtree Result* and *Pattern Match* respectively in this dissertation, fail to satisfy the atomicity and intactness properties. Subtree Result defines a query result as a tree rooted at a selected LCA node consisting of *all* relevant matches that are descendants of this LCA node and the paths connecting them, as adopted in [56, 144, 91, 93]. The results generated by Subtree Result generally fail to be atomic, and such information overload causes ineffective ranking.

Example 6.2 *For query "*men, apparel, store*", a result produced by Subtree Result generally contains many target instances: the tree rooted at a* retailer *node that contains the match to* apparel *and* all *the matches to* store *and* men*, such as the ones shown in Figure 6.2. As we can see, subtree result violates the* Atomicity *property. With many target*

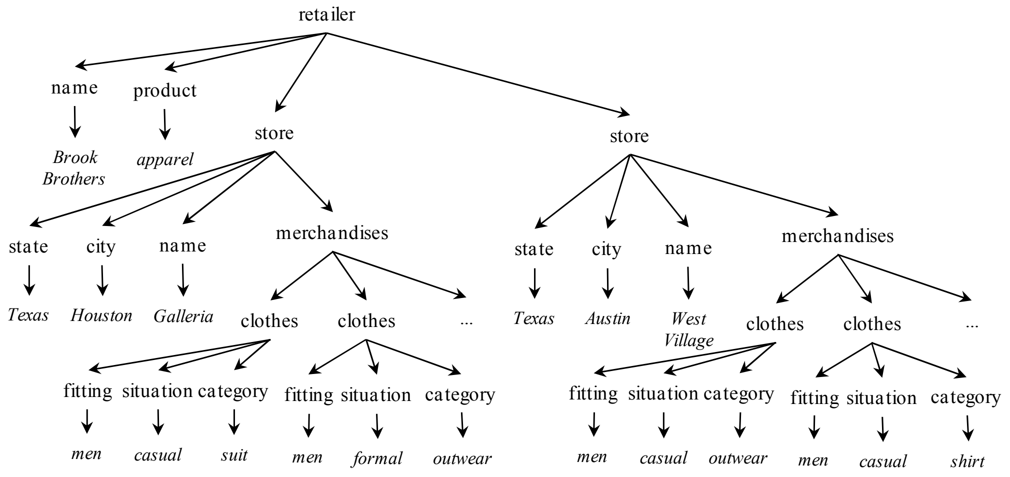


Figure 6.2: A Query Result of Q_9 Returned by Subtree Result

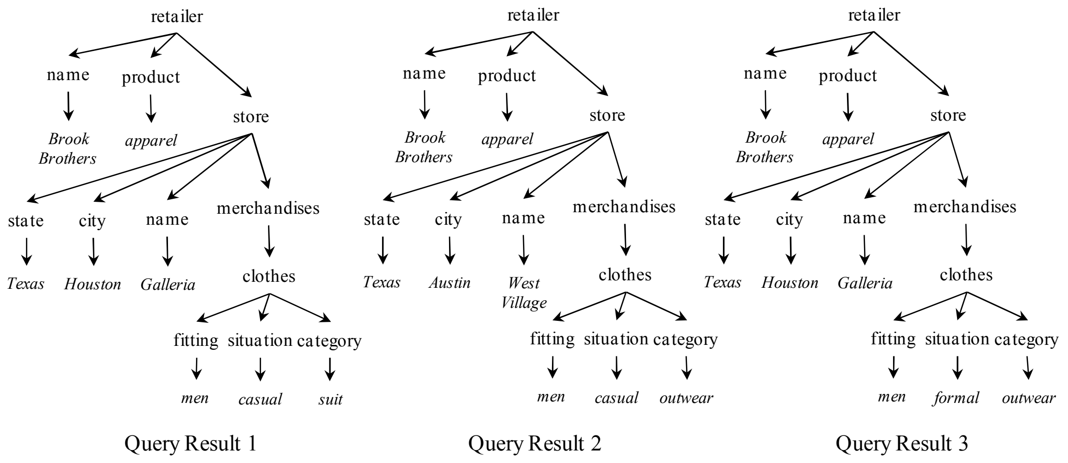


Figure 6.3: Three Query Results of Q_9 Returned by Pattern Match

instances (stores) in a single result, ranking is not performed on target instances, and can be totally unreasonable. Suppose there are two results result 1 and result 2 which have similar sizes, and result 1 has more matches to query keywords than result 2. Then result 1 is ranked higher by existing ranking schemes [18, 43, 56, 124, 24]. However, it could be the case that result 2 has a store which sells many clothes for men. It may also be the case that in both results, there are some stores with very few or no clothes for men, but such stores are still returned, and may be returned earlier than the stores that are more relevant.

Besides, Subtree Result may return excessively big results, e.g., for Q_9, many stores with a large number of clothes are returned as a single huge result without being ranked, which is very overwhelming and unreadable.

On the other hand, *Pattern Match* defines a query result as a tree rooted at an LCA node consisting of *exactly one match to each query keyword* which are meaningfully related with each other and the paths connecting them, used in [43, 61, 74, 82, 86]. The results generated by Pattern Match Result generally fail to be intact, and such information underload causes ineffective ranking.

Example 6.3 *The top 3 results of query "*men, apparel, store*" generated by Pattern Match are shown in Figure 6.3. Although each result is atomic, it is not intact: the same target instance (store) named* Galleria *with two* men*'s clothes is presented as two results, one for each match of* men*.*

The lack of intactness results in several problems. First, the top-k results generally contain information about less than k target instances, since multiple results can describe the same target instance. In this example, top 3 results presented in Figure 6.3 are about two stores. This not only wastes the user's time but also makes it difficult for a user to find the top k ranked target instances.

Second, from such results the user loses information, e.g., the first and third results are actually the same store.

*Furthermore, separating the supporting information (*men*) of the same target instance (*store*) into multiple results will divide the ranking signals among these results. It demotes all results corresponding to this target instance and potentially puts them in lower positions. In this example, all results returned by Pattern Match have the same size and the same number of keyword matches, thus they are almost indifferentiable by any ranking scheme. However, the store with more matches to* men *should intuitively be ranked higher.*

As we can see from Examples 6.1 - 6.3, the way of result composition has a big effect on the effectiveness of ranking.

Intuitively, each keyword search has a goal, which is usually the information of a real world entity or relationship among entities, as observed in [36, 38]. We use the term *search target* to refer to the information that the user is looking for in a query, and *target instance* to denote each instance of the search target in the data. Each desirable query result should have *exactly one target instance* along with all associated evidence, so that ranking and top-k query processing can be based on target instances, and thus become meaningful. Specifically, query results of an XML keyword search should satisfy:

Atomicity. A query result should be *atomic*: it should consist of a single target instance. In the above sample query, each result should correspond to a distinct store. Atomicity enables the ranking method to rank the target instances and show the top-k most relevant ones to the user.

Intactness. Each query result should be *intact*: containing the whole target instance as well as all its supporting information. In the above sample query, all keyword matches related to the same store should be in one result. With intactness, a ranking method has the whole view of each target instances to give a fair ranking.

In this chapter we propose a novel technique to automatically compose atomic and intact query results for XML keyword searches, such that each result contains exactly one search target instance along with all its evidence, as illustrated in the above examples. Unlike the existing approaches, which are oblivious to users' search intentions, the proposed query result composition is *driven by the user search target* and hence *ranking friendly*.

Two technical challenges are addressed. First, we need to identify *user search target*. Although we could define a special query syntax and ask the users to explicitly specify the search target, such as some existing works [36, 38], not all users are willing to take this extra effort. Thus, to relieve users' burden, our system supports simple keyword queries and makes *best-effort* for automatic search target inferences. However, such inferences are very challenging, as the search target may not even appear as a keyword in a user query. Interestingly, we discover that the meta-information of the data, the matching patterns of the query keywords, as well as the mined *modification power* of keyword matches often provide hints on identifying search targets. The second challenge is, given the inferred search targets, how to compose atomic and intact query results, each of which is an XML tree that is *centered* around a single instance of search target and encompasses all keyword matches *related* to this instance.

6.2 Target Driven Query Result Composition

Users who issue queries often desire the information of one or a set of entities that satisfy certain conditions. We name such entities as *target entities*. In this section we first define two properties based on target entities that an ideal XML search engine should satisfy: *atomicity* and *intactness*. Then we propose a strategy, called *Targeted Return*, which automatically infers target entities and composes meaningful query result based on target entities and their relationships toward achieving these two properties.

Atomicity and Intactness

As discussed and illustrated by examples in Section 6.1, results of XML keyword search should be *atomic*, i.e., consist of a single target instance; and *intact*, i.e., contain the whole target instance together with all its supporting information. Here we formally define atomicity and intactness.

Definition 6.1 (Atomicity) *Given a set of target entity type $\mathbb{E} = \{E_1, E_2, \cdots, E_n\}$ of a query Q, a result of Q is* atomic*, if it contains exactly one instance of each $E_i \in \mathbb{E}$.*

Definition 6.2 (Intactness) *Given a set of target entity type $\mathbb{E} = \{E_1, E_2, \cdots, E_n\}$ of a query Q, a result of Q is* intact*, if it contains all supporting nodes of the instance of each $E_i \in \mathbb{E}$ that match keywords in the result.*

The concept of "supporting node" in Definition 6.2 is defined as following.

Definition 6.3 (Supporting Node) *A node u labeled U is a* supporting node *of a node v labeled V, if there does not exist a node u' labeled U, such that $LCA(u', v)$ is a descendant of $LCA(u, v)$, where LCA denote the lowest common ancestor of two nodes.*

Intuitively, if we can find such a u', then u' has a closer, or more specific, relationship with v. Thus it is likely that u' is a supporting node of v, not u. For example, in Figure 1.3, *apparel* is a supporting node of *retailer*.

Other measurements of node relationships, such as interconnection relationship1 [43] and "meaningfully related" relationship [86], can also be used to define supporting node. Note that the supporting relationship between two nodes is similar but different as the "meaningfully related" relationship. That u is a supporting node of v does not imply that v is also a supporting node of u. Two nodes u and v are meaningfully related if they are supporting nodes of each other.

Next we present techniques for identifying search targets as well as composing results.

Identifying Target Entities

We infer target entities by analyzing the matches to input keywords and the XML data structure. Two different situations are considered:

CASE 1. In many queries, users provide hints about the XML nodes they are looking for as well as the conditions these nodes should satisfy. We call these XML nodes *return nodes* and the conditions *search predicates*.

Intuitively, if an entity is specified in a query without information about its associated attributes, then likely their instances that satisfy the search predicates should be returned. If a connection node or attribute node is specified in a query, but none of their value descendants matches any keyword, then probably the instances of these nodes along with the values are what the user is searching for. In this case, the entity associated with this attribute, or the nearest descendant entity of the connection node, is considered as the target entity. Note that a value node (e.g., *Houston*), or a name node, which, together with its descendant attribute value(s), (e.g., *city, Houston*) can serve as search predicate.

Example 6.4 *For query "*Brooks Brothers, city*",* city *is considered as a return node as it matches attribute name and its subtree does not contain any keyword match. It is likely that* *the user wants to retrieve the* city *of store that satisfy the search predicates: it is a store of* Brooks Brothers*. Therefore, the entity associated with* city*, which is* store*, is the search target.*

Return nodes can be inferred using the method discussed in Chapter 5. After we have the return nodes, we can infer target entities from return nodes. If a return node is an entity node, then it is a target entity. If a return node is an attribute (e.g., *city* in the previous example), then the associated entity (e.g., *store*) is considered as a target entity. Otherwise, a return node is a connection node (e.g., *merchandises* in sample XML document), then its nearest descendant entities are considered as target entities (e.g., *store*).

However, some queries may have more than one return nodes and they have more than one associated entities, and it is not obvious which associated entity should be the search target. Therefore we discuss case 2, in which there are entities associated with return nodes.

CASE 2. In case there are more than one entities associated with return nodes, we exam all such entities and the modifying relationship between search predicates and theses entities to identify target entities. Let us look at an example.

Example 6.5 *Consider query "*Galleria, casual*", where both keywords are search predicates. The implicit return nodes are entities* store *and entities* clothes*. Let us judge two candidate semantics of this search that consider different entities as the target entity:*

1. *Find the* store *that (i) is named* Galleria *(ii) has* casual *clothes.*
2. *Find the* clothes *who (i) are for* casual *situations and (ii) are sold in* Galleria*.*

If we take a closer look, semantics (1) is counter intuitive: it specifies two conditions for search target store*. However, nearly every store has casual clothes. The second condition does not* modify *(or restrict/constrain) the* store *entity, and is unlikely to be used by a reasonable user for searching stores.*

On the other hand, all search conditions for semantics (2) are reasonable as they indeed modify *the clothes entity: (i) Not all* clothes *are for* casual *situations; (ii) Not all* clothes *are sold in* Galleria*. Therefore, both input keywords can be viewed as predicates for selecting the specific* clothes *entity.*

Enlightened by this example, when all the keywords are predicates, we can infer the modifying relationship between the keywords that match attribute values and the relevant entities. Intuitively, if an attribute value A is always related to an entity type E, then A does *not* modify E; otherwise A is a *modifier* of E. Note that this intuition is analogous to the concept of information gain in machine learning: if attribute value A is always related to entity type E, using keyword A does not distinguish any instance of E, thus the information gain is zero if the search target is E. Therefore, A should not be a modifier of E.

Definition 6.4 (Modifier) *In an XML document D, an attribute value A is a* modifier *of entity E, if and only if there is at least one instance of E in document D which does not have any instance of A as its descendants.*

In Example 6.5, suppose in the XML data in Figure 1.3, every *store* instance has at least one instance of attribute value *casual* in its descendants, then *casual* and is not *retailer*'s modifier. On the other hand, all attribute values in the keyword query modify *clothes*.

Note that attribute values of an entity are always considered to be modifiers of its non-ancestor entities, as we assume that the attribute value will not be shared by all instances of the entity. For example, all attribute values of entity *store* modify a descendant entity *clothes*, as any attribute value of *store*, unless shared by all the instances of *store*, can always be used as a predicate to constrain the instances of *clothes* entity. We may search *clothes* who are sold in *Galleria*, *Texas*, *Houston*, etc.

Besides the modifying relationship among attribute values and entities, in fact, different modifiers have different *modification power*, as illustrated in the following example.

Example 6.6 *Consider query "*Brooks Brothers, Houston*". Two candidate semantics of this query are:*

1. *Find the* retailer *named* Brooks Brothers *that has one or more stores located in* Houston*.*

2. *Find the* store *for* Brooks Brothers *located in* Houston*.*

In the data, Brooks Brothers *uniquely identifies a retailer. If the user's search target is a retailer, s/he does not need additional keywords like* Houston*. Therefore, the first semantics is less likely to reflect the user intention and it is more likely that the second semantics is what the user actually means.*

As we can see, although both *Brooks Brothers* and *Houston* are modifiers of *retailer* by Definition 6.4, they have different *modification power*. A key of an entity, such as *Brooks Brothers*, has the strongest modification power, whose presence shadows/disables all other modifiers. In analogous to the concept of information gain, if the search target is *retailer*, using *Brooks Brothers* already identifies the instance of the search target, thus the information gain of using other modifiers is zero. Therefore, *retailer* is unlikely the search target.

The key attributes of an entity is usually specified using the "ID" attribute in DTD. In case we do not have the DTD, it can be retrieved by mining method: if every value combination in an attribute set S associated with entity E is unique across all the instances of E, then S is considered as a key for entity E.

Note that the rules of defining modifiers and the process of key mining can be relaxed in practise. For example, in an XML document, it is possible that a few stores do not have casual clothes, but as long as the vast majority of the stores have at least one casual clothes, *casual* can be considered a non-modifier of *store*. Similarly, an attribute can be considered as a key of an entity if the number of its duplicate values is quite small, but not necessarily zero. The threshold can be flexibly set in different applications.

In summary, we identify target entities according to the following heuristics.

Definition 6.5 (Target Entity) *For a keyword search Q on XML data D, for each relevant entity E:*

1. *There are return nodes inferred in Q: E is a* target entity *if it is a return node, or the entity associated with a return node that matches an attribute, or a nearest descendant entity of a return node that matches a connection node.*
2. *There is no return node in Q: E is a* target entity *if both of the following two conditions hold:*
   a) *All keywords in Q that match attribute values in D are modifiers of E.*
   b) *None of the modifiers of E are keys, or all of them are keys.*
3. *When we can not find one target entity from the last two steps, we treat all relevant entities as target entities.*

When there is only a single target entity, it is called the *center entity* of the query result.

Definition 6.6 (Center Entity) *For a keyword search Q on XML data D, if the set of target entities is a singleton {E}, then E is defined as the* center entity *of Q on D.*

Composing a Query Result

After identifying target entities and center entity, we discuss how to compose atomic and intact results according to the inferred target entities and center entity.

When a query has a *center entity*, we construct one query result for each instance of the center entity to make sure that it is *atomic*. We also include into a result the matches of the keyword that are supporting nodes of this center entity instance to make the result *intact*.

Example 6.7 *Take query "*Galleria, casual*" as an example, which is likely to search the* casual *clothes sold by* Galleria*. We identify* clothes *as the center entity, as discussed* *in previously. Each query result includes an instance of* clothes*. Consider the result that includes* clothes *(0.0.2.3.0). For keyword* Galleria*, the supporting node to* clothes *(0.0.2.3.0) is* Galleria *(0.0.2.2.0) since their LCA is node 0.0. Other matches to* Galleria*, if any, has a higher LCA (0) with* clothes *(0.0.2.3.0) which is an ancestor of 0.0. For keyword* casual*, similarly, its supporting node to* clothes *(0.0.2.3.0) is node* casual$_1$ *(0.0.2.3.0.1.0).*

We also include the data node Galleria *(0.0.2.2.0) into other query results rooted at* store *(0.0.2), serving as evidence that the corresponding* clothes *node is relevant.*

When a query has multiple target entities, we observe that atomicity and intactness may not be simultaneously achievable. Consider Figure 1.3 for example. If both *store* and *clothes* are known as target entities, then both *casual* nodes under *store* (0.0.2) are supporting nodes of *store* (0.0.2). According to intactness, they should both be included in the result that contains *store* (0.0.2). However, according to atomicity, only one *clothes*, and hence one *casual* node can be included in one result.

In this case, since atomicity and intactness are not both achievable, we choose to achieve intactness using subtree result. The reason is that subtree result can be achieved much more efficiently and scalably than pattern match as shown in the experiments.

6.3 Algorithms

In this section, we present the algorithms to efficiently identify target entities and generate meaningful query results.

Building Indexes

We build three indexes offline to speed up query processing. A *Label Index* is an inverted index, supporting operation $Label2ID(k)$ which retrieves a sorted list of Dewey IDs of the data nodes matching input keyword k. A *Dewey Index* and a *Modifier Index* are used to determine target entities.

As discussed in Section 6.2, we have different strategies for determining target entities in two different situations. Case 1: if there are return nodes specified in input keywords, target entities are the entities associated with the return nodes. Case 2: If return nodes are not specified, we check the modifying relationship between each attribute value that matches a keyword and each relevant entity involved in the query.

For the first case, we need to quickly determine a return node's associated entity. If a return node is an entity, then it is a target entity. Otherwise, if a return node is an attribute, we need to efficiently find out the attribute-entity association relationship; and if it is a connection node, its nearest descendant entities. To support this, we build a *Dewey index* whose entries are either entity instances with their corresponding attribute instances, or connection node instances. Each entry records the node label, the names and values of associated attribute instances (if any), as well as the information of parent and children that are not attribute instances. For a given Dewey ID, Dewey index returns the corresponding entry that contains the information of this node.

For the second case, we need to quickly determine whether a predicate is a modifier of an entity type. To efficiently determine it, we build a *modifier index* that records the modifying relationship between attribute values and entities. In this index, each attribute value has an associated list that records two types of information: (1) the entities of which it is a key attribute; and (2) the entities that are *not* modified by this attribute value. Since the entities that are modified by an attribute value are far more than those that are not modified by it, we record the negative cases. The list of attribute values and the list of entities associated to each attribute value are implemented using hash table. Only the attribute values whose associated lists are not empty are recorded. To determine whether an attribute value A is a key of entity E or whether A modifies E, we first search the entry of A in the attribute list using hashing, then search E in the associated list of A. Modifier index allows three operations to identify the relationship of A and E: $keyModifier(A, E)$, $nonKeyModifier(A, E)$, and $nonModifier(A, E)$ whose functionalities are self-explanatory.

For the XML tree in Figure 1.3, its modifier index looks like the one shown in Figure 6.4. For attribute value *Brooks Brothers*, it is a key attribute of entity *retailer*, therefore its associated list contains *retailer* with an underline to denote the key attribute relationship. Since entity *retailer* and *store* are not modified by *casual*, they are in the list of *casual*.

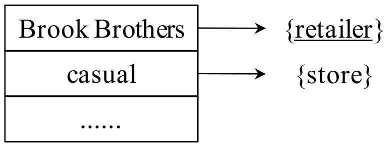


Figure 6.4: Modifier Index for the XML tree in Figure 1.3. An underlined entity indicates that the value is the key of the entity. Otherwise, the value is not a modifier of the entity.

For a keyword search, we first use the label index to find the Dewey IDs of match nodes. Then using Dewey index, we can determine the node category of each keyword match and then infer the return nodes and search predicates of the query. If return nodes are present, we can find the corresponding target entity instances by accessing the Dewey index. Otherwise, we find the modifying relationship and consequently the target entity instances by accessing the modifier index, as will be discussed in details later.

Algorithm 2 Building Modifier Index

```
buildModifierIndex (Label Index: $Lidx$; Dewey Index: $Didx$)
1: Initialize $Midx$ as empty
2: for each attribute value $A$ in $Lidx$ do
3:   $C[E] = 0$ for all entity type $E$
4:   $L(A) = Label2ID(Lidx, A)$ {$L(A)$ is the list of DeweyID of nodes matching $A$ retrieved from $Lidx$}
5:   for each instance $a$ of $A$ in $L(A)$ list order do
6:     $a'$ = node in $L(A)$ immediately before $a$ {$a'$ is the instance of $A$ that has the biggest Dewey ID smaller than that of $a$}
7:     for each entity $e$ on the path from $a$ (inclusive) to LCA($a, a'$) (not inclusive) obtained from $Didx$ do
8:       {if $a'$ does not exist, let LCA($a, a'$) be the XML root}
9:       $C[E]++$
10:      if $C[E]$ = $|E|$ then
11:        add $E$ to $Midx[A]$ with non-modifier relationship
12:      if $A$ is a key attribute of $E$ then
13:        add $E$ to $Midx[A]$ as key attribute relationship
```

The construction of label index and Dewey index is omitted due to space limitation. The construction of the modifier index (denoted as $Midx$) is presented in Algorithm 2. Initially, the list associated with each attribute value is empty (line 1). The algorithm examines all the distinct attribute values in turn. For each attribute value A, we count the number of entity instances of type E that are related to A, denoted by $C[E]$. For A we use the label index to retrieve a list of instances of A in their Dewey ID order, denoted as $L(A)$. For each instance a in $L(A)$, we find the entity instances e on the data path from a to $LCA(a, a')$, where a' is the node in $L(A)$ that appears immediately before a. That is, a' is an instance of A that has the largest Dewey ID which is smaller than that of a. For each such entity instance e, whose type is E, we increment the counter $C[E]$ by 1, denoting that there is one more attribute value a that is related to E (line 9). The information of e is obtained by accessing Dewey index using e's Dewey ID. To guarantee that every entity instance will be counted at most once in the counter $C[E]$, we should not increment counters for entity instances on the data path from $LCA(a, a')$ to the root of the data tree, as they have already been taken care of when we process the nodes in $L(A)$ appearing before a. If at the end of processing $L(A)$, the counter $C[E]$ is equal to the total number of instances of E in the XML data, denoted as $|E|$, it implies that every instance of E is related to an instance of A. Therefore, we conclude that A does not modify E, and add E to the list associated with A in the modifier index (line 11). E is also added to the list of A if A is a key attribute value of E (line 13). For simplicity, our implementation only mines singleton keys (a key that consists of a single attribute), and the procedure is omitted in the algorithm.

Example 6.8 *Consider the modifier index entry for attribute value* casual *for the XML tree in Figure 1.3 as an example to illustrate Algorithm 2. From label index, we retrieve the list of nodes that match* casual *($L(casual)$): $\{$0.0.2.3.0.1.0, 0.0.2.3.2.1.0, ...$\}$. For node 0.0.2.3.0.1.0 which does not have a preceding node in the list, we examine the nodes on the path from this node to root* retailers *(0). Given the Dewey ID of a node, we can find out whether the node is an entity instance or not, and if it is, find the entity type, using Dewey index. Since* clothes*,* store *and* retailer *are entities on the path, we increase $C[clothes]$, $C[store]$ and $C[retailer]$ by 1. Then we move to the next node* casual *(0.0.2.3.2.1.0) in $L(casual)$ list, and process the nodes on the path from this node to LCA(0.0.2.3.0.1.0, 0.0.2.3.2.1.0) = 0.0.2.3. We increase $C[clothes]$ by 1, and now we have $C[clothes]$ = 2. We continue the processing for nodes in the list. At the end, we find that $C[store]$ is equal to the number of* store *instances in the data, which means that every* store *has at least one* casual *clothes in its subtree. Therefore* casual *is not a modifier of* store*, so we add* store *to the list associated with* casual *in $Midx$. On the other hand, $C[clothes]$ is smaller than the number of* clothes *nodes, as not all* clothes *are for* casual *situations, therefore we do not* *need to update* $Midx$*'s* casual *entry with respect to* clothes*.*

Generating Query Results

Now we present Algorithm 3, which takes relevant matches to a keyword query (which are obtained by adopting one of the existing approaches [43, 56, 61, 82, 86, 144]) and indexes as input, and composes meaningful query results.

First the algorithm computes VLCA nodes, each of which is the root of one or more query result trees (line 1 of procedure $TargetedReturn$), for which purpose we use the algorithm proposed in XKSearch [144]. Then it finds relevant entity instances, and invokes procedure $findCenterEntity$ for identifying the center entity of the query (line 3). If the center entity exists, then each relevant entity instance e that is an instance of the center entity leads to a query result, denoted by QR. For each such entity instance e, a query result is generated consisting of e and the matches to each keyword k that are supporting nodes of e (Definition 6.3), together with their connections (line 6-7). Such a result satisfies Definitions 6.1 and 6.2, and is atomic and intact. If there is no center entity, it takes the default mode, which adopts Subtree Result as its efficiency is superior to Pattern Match as shown in Section 6.4. That is, it generates query results by returning the relevant matches in the subtrees rooted at VLCA nodes, which are *intact* (line 9-10).

Example 6.9 *Take query "*Galleria, casual*" as a running example for this algorithm. The relevant matches of this query are* Galleria *(0.0.2.2.0),* casual *(0.0.2.3.0.1.0, 0.0.2.3.2.1.0). Their SLCA is node* store *(0.0.2). Relevant entity instances are* store *(0.0.2),* clothes *(0.0.2.3.0, 0.0.2.3.2). The center entity is* clothes *(returned by procedure* $findCenterEntity$*, which will be illustrated shortly). There are three entity instances of* clothes*, each forming a query result. Consider* clothes *(0.0.2.3.0), for keyword* Galleria*, node 0.0.2.2.0 is the supporting node and therefore is included into the corresponding query result. Similarly, we also add the supporting node that match keyword* casual *(0.0.2.3.0.1.0) into this query result.*

Procedure $findCenterEntity$ finds the center entity given relevant matches and

Algorithm 3 Composing Ranking Friendly Results

```
TargetedReturn ($relMatches, indexes, Q$)
 1: $VLCA$=$computeVLCA(relMatches)$ [144] {Group $relMatches$ such that each group shares
    the same $VLCA$ node}
 2: $relEntityIns$ = the set of entity instances that are on a data path from a node in $VLCA$ to a
    node in $relMatches$
 3: $centerEntity$ = $findCenterEntity(relMatches)$
 4: if $centerEntity \neq null$ then
 5:   for each $e \in relEntityIns$ whose type is $centerEntity$ do
 6:     for each keyword $k \in Q$ do
 7:       add the matches to $k$ that are supporting nodes of $e$, along with the edges connecting
          them, to $QR$
 8: else
 9:   for each $v \in VLCA$ do
10:     $QR$ = a tree rooted at $v$ which contains nodes in $relMatches$ that are descendants of $v$
findCenterEntity ($relMatches, indexes$)
 1: $retNode, tarEntity$ = $findReturn(relMatches)$
 2: if $tarEntity \neq \emptyset$ then
 3:   if $tarEntity$ is a singleton then
 4:     return the element in $tarEntity$
 5:   else
 6:     return $null$
 7: else
 8:   for each $E$ that is the type of a target entity instance do
 9:     $S[E]$=0 {use $S[E]$ to encode the modifying status of entity $E$}
10:     for each keyword $k$, that matches a value do
11:       if $nonModifier(k, E)$ then
12:         $S[E]$=-1
13:       else if $nonKeyModifer(k, E)$ then
14:         if $S[E]$=2 then
15:           $S[E]$=-1
16:         else if $S[E]$=0 then
17:           $S[E]$=1
18:       else
19:         if $S[E]$ = 0 or 2 then
20:           $S[E]$=2
21:         else
22:           $S[E]$=-1
23:   if there is exactly one entity type $E$ such that $S[E]$=1 or 2 then
24:     return $E$
25:   else
26:     return $null$
findReturn ($relMatches, indexes$)
 1: $retNode$=$\emptyset$, $tarEntity$=$\emptyset$
 2: for each $m \in relMatches$ do
 3:   if ($m$ matches a node name $N$) and (there does not exists $m'$, such that $m' \in relMatches$
      and $m'$ is a descendant of $m$) then
 4:     find the corresponding entity instance $e$ of $m$, $E$=the type of $e$
 5:     $retNode$=$retNode \cup \{N\}$, $tarEntity$=$tarEntity \cup \{E\}$
 6:   if ($m$ matches an instance of entity $E$) and (none of the attributes associated with $m$ matches
      a keyword in $Q$) then
 7:     $retNode$=$tarEntity$=$tarEntity \cup \{E\}$
 8: return $retNode, tarEntity$
```

indexes as input. First, procedure $findReturn$ is called to find the return nodes $retNode$ (if any). It also returns entities these return nodes are associated with: $tarEntity$, which is a list of all target entities. If there is only one element in $tarEntity$, then it is the center entity, otherwise there is no center entity.

If there are multiple candidate center entities, an entity is a target entity if it is modified by every keyword k that is a predicate value; and if one modifier is a key attribute, then all modifiers are key attributes. We use an integer $S[E]$ as a flag to denote whether E is a target entity. Specifically, $S[E]$=0 indicates that E is not a target entity since some attribute values that match input keywords do not modify it. $S[E]$=-1 indicates that E is not a target entity since some of its modifiers are key attributes, but others are not. $S[E]$=1 means that E is a target entity modified by non-key attributes. $S[E]$=2 indicates that E is a target entity exclusively modified by key attributes. Now let us look at how to set $S[E]$. If k does not modify E, we simply set $S[E]$=-1 (line 12 of procedure $findCenterEntity$). Now consider the case when k is non-key modifier of E (line 13). If $S[E]$=2, indicating that there is a already keyword representing a key attribute modifier of E, entity E becomes disqualified, and $S[E]$= -1 (line 15). If $S[E]$ still has the initial value 0, then now it has a non-key modifier, so $S[E]$= 1 (line 17). On the other hand, when k is a key attribute modifier of E, $S[E]$ is set to -1 if E has found a non-key modifier (line 23). Otherwise, $S[E]$ =2 (line 21). Finally, if there is only one entity type E such that $S[E]$ = 1 or 2, then E is the center entity.

Example 6.10 *Continuing the previous example. For query "*Galleria, casual*, there are two candidate center entities:* store *and* clothes*. Since keyword* Galleria *is a modifier of entities* store *and* clothes*, we have $S[store]$=2 and $S[clothes]$=1. For keyword* casual*, it is a non-modifier of* store *and a non-key modifier of* clothes*, therefore $S[store]$=-1 and $S[clothes]$=1. Finally,* clothes *is the only target entity, hence the center entity of this query.*

6.4 Experiments

In this section we present experimental study of our approach for composing query results, *Targeted Return*. Three metrics are tested: quality of query results, efficiency of generating query results and scalability with respect to increasing data and query size.

Table 6.1: Data and Query Sets for Testing Result Composition

Baseball	**Mondial**	**Synthetic**
Q_1 American, Relief Pitcher	Q_{16} France, population, 95	Q_{31} bookstore, book
Q_2 Central, Abbott	Q_{17} mondial, Tasmania, Gotland, Area	Q_{32} location, book
Q_3 starting Pitcher, Cleveland	Q_{18} mondial, Country, Muslim	Q_{33} price, bookstore, Seattle, Borders
Q_4 First Base, east	Q_{19} country, Berlin	Q_{34} book, tempe
Q_5 Team, 1998	Q_{20} mondial, organization, member	Q_{35} subsection, Breaking Dawn
Q_6 player,name,Minnesota	Q_{21} Germany datacode	Q_{36} subsection, New Moon
Q_7 Jim,wins	Q_{22} democracy, muslim	Q_{37} Bookstore, Harry Potter
Q_8 Cleveland,losses	Q_{23} country, province	Q_{38} star, 22.5
Q_9 team, player	Q_{24} Austria,gdp	Q_{39} Borders, Tom
Q_{10} Dwight, Gooden	Q_{25} china, province	Q_{40} Best selling, Tempe
Q_{11} Paul, team	Q_{26} Albania, city	Q_{41} Amazon, Borders, Seattle
Q_{12} Chicago, Central	Q_{27} cuba, gdp, area	Q_{42} caption, price, author, book
Q_{13} white sox , Abbott	Q_{28} ethnicgroups, Slovene	Q_{43} Breaking Dawn, subsection, English
Q_{14} Joey, Jim	Q_{29} Canada, province, Edmonton	Q_{44} Harry Potter
Q_{15} Mariners, player	Q_{30} river, Mississippi	Q_{45} author, English

Experimental Setup

Equipment. The experiments were performed on a 3.60GHz Pentium 4 machine with 2GB memory running Windows XP Professional. We implemented Targeted Return in Microsoft Visual C++, in which we used Oracle Berkeley DB[1] for label index and Dewey index. The test data and query sets are shown in Table 6.1.

Data. We have tested three data sets: Baseball, Mondial and Bookstore. Baseball is a data set recording the teams and players of North American baseball league.[2] Mondial is a world geographic data set.[3] To show the applicability of our approach on a variety of XML data, we also generate a synthetic bookstore data set, which contains recursive elements of average depth 5.4.

A Dewey index, label index and modifier index is built on each data set. The average ratios of the size of these indexes to the size of the data are 3.90, 1.42 and 0.01, respectively.

Queries. Our query set, shown in Table 6.1, consists of three parts with 45 queries in total. We invited 10 graduate students majoring in computer science who did not participate in this project for a user survey to obtain test queries. We asked each of them to

[1]http://oracle.com/technology/products/berkeley-db/index.html
[2]http://www.ibiblio.org/xml/books/biblegold/examples/baseball
[3]http://www.cs.washington.edu/research/xmldatasets

issue keyword queries on each data set. Since the queries issued by the users may not be complex enough to cover a variety of query patterns, we also issued five queries ourselves for each dataset. $Q_1 - Q_5$, $Q_{16} - Q_{20}$, $Q_{31} - Q_{35}$ are the queries we issued, and the other queries were issued by the users. Our query set contains a number of different query patterns: some queries only have tag names thus have lower selectivity, some only have attribute values, and some have both. Besides, some queries have VLCA nodes of small depths and potentially each query result tree is large, while some have VLCA nodes of large depths. Some queries have a single inferred target entity, while others have multiple, as indicated by user study to be introduced shortly.

Comparison Systems. We compare Targeted Return with Subtree Result and Pattern Match. We implemented Subtree Result as [144], and used Timber [8] for Pattern Match approach.

Quality of Query Results

We study the quality of query results generated by different strategies. Note that for a given query, the union of the content of all results is the same for all three approaches. Therefore, common metrics for measuring whether relevant nodes are retrieved, such as precision, recall and F-measure, are not helpful for this experiment. Instead we performed user study to score the quality of query results generated by each approach in terms of the accuracy of inferring search targets, as well as the users' overall satisfaction with the results.

Also note that since no ranking scheme is perfect, introducing a particular ranking method in the experiment may bring unnecessary bias and distract the user. Therefore, we made a questionnaire for the users, which is composed of two parts. As discussed in Section 1, an approach that provides 1-1 mapping between a result and a search target is likely a ranking-friendly result composition method, since it enables ranking to be based on target instances, and users will see exactly k target instances in the top-k results. Thus Part 1 evaluates whether our proposed approach provides 1-1 mapping between a result and a search target, which does not depend on any particular ranking function. Additionally, to verify that 1-1 mapping enables good results, Part 2 asks for users' overall impression on the result generated by our approach compared with those generated by Subtree Result and Pattern Match.

In the following, we use Q_6 in our query set, "*player, name, Minnesota*", as an example to illustrate the process of the user study.

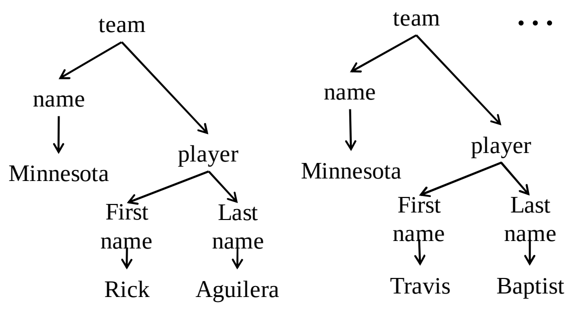
Part 1. In order to verify our claim that Targeted Return makes query results cater to user's search intention, we asked the users to write down an English sentence to illustrate their search goal for each query. We also required the users to either underline the search target in the sentence or explicitly specify search target followed by the sentence. For example, for Q_6 (player, Minnesota,name), the sentence describing its semantics is "Find the player's name, who played in team Minnesota" (i.e., the search target is *player*).

After collecting the queries and their semantics, we use Target Return, Subtree Result and Pattern Match to generate the query results and based on the semantics of each query, categorize the results into the following three categories:

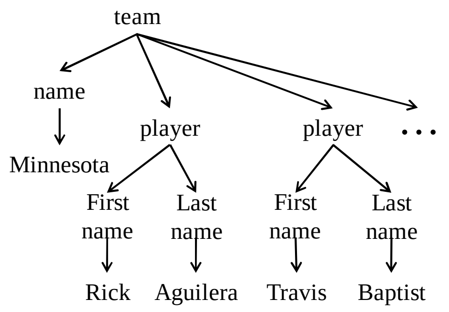
- (A) There is one to one mapping between results and search target instances.
- (B) There is one to many relationship between results and search target instances.
- (C) There is many to one relationship between results and search target instances.

Take Q_6 as an example, whose search target is *player*. The results generated by the three approaches are shown in Figure 6.5. As we can see, Targeted Return places each distinct player in one result, and therefore the results of Targeted Return on Q_6 belong to category (A).

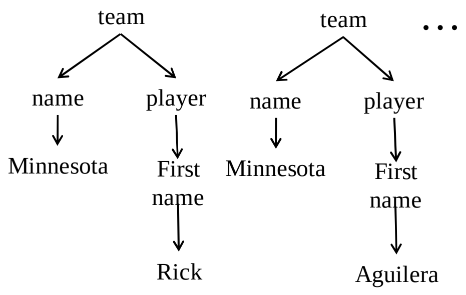
On the other hand, Subtree Result has only one result for this query, consisting of the *team* node whose name is *Minnesota* as well as all the *players* in its subtree. The information of all players is returned in one result of Q_6, hence category (B). Pattern Match produces two results for each player, and therefore, it is categorized as (C). Such an organization of results would cost the user additional effort to browse the query results to get the desired information.



(a) Targeted Return



(b) Subtree Result



(c) Pattern Match

Figure 6.5: Results of Q_6 generated by Targeted Return, Subtree Result and Pattern Match

The distribution of categories the results of 45 test queries belong to is shown in Figure 6.6. As we can see, a large number of results produced by Subtree Result get option (B), since results of Subtree Return usually contain multiple search target instances. On the other hand, Pattern Match mainly gets option (C) for the opposite reason. The query results given by Targeted Return, in most cases, have one to one mapping between a result and a target instance. Therefore, ranking of results generated by Targeted Return can be based on each target instance, and is thus desirable.

Part 2. For each query, the users were given the results generated by the three approaches with shuffled order, and were asked to give an overall satisfaction score based on their impression of each query result of scale [1, 3]:

- 3: I have no trouble finding what I am looking for.
- 2: I need efforts to extract the desired information from the results.
- 1: I cannot find what I am looking for without re-organizing the results.

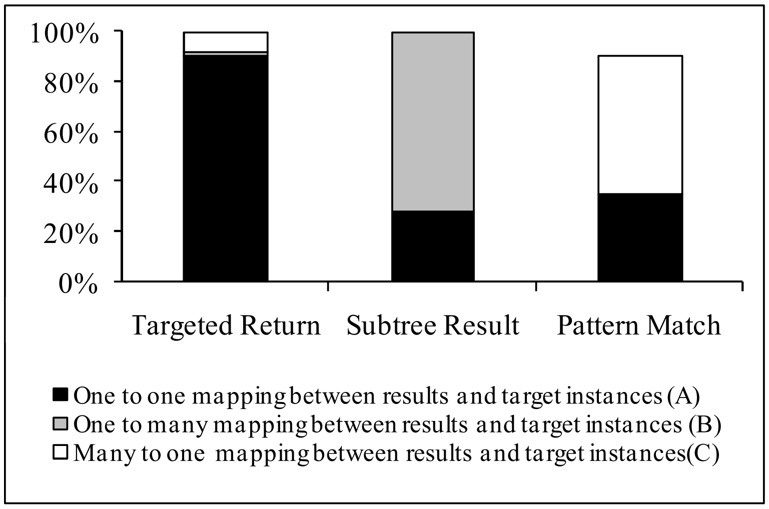


Figure 6.6: Quality Survey on Top-k Query Results

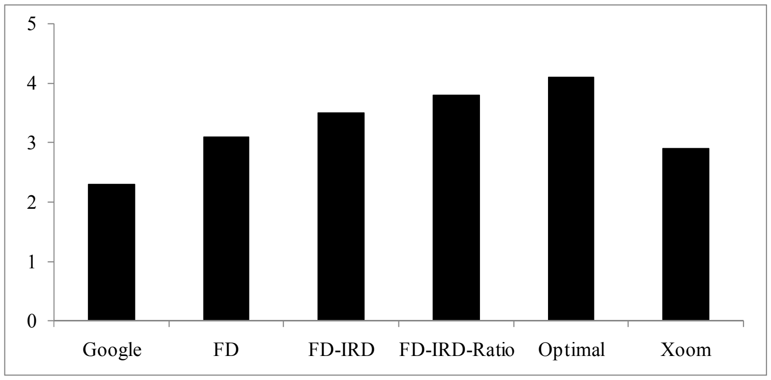


Figure 6.7: Scores of Targeted Return, Subtree Result and Pattern Match

The average scores of the three approaches of all 45 test queries given by the user is shown in Figure 6.7. Targeted Return got the best score of 2.76, followed by Pattern Match 1.92 and Subtree Result 1.15. This indicates that the organization of query results by Targeted Return is closest to users' expectations.

Now we analyze the quality of the query results generated by Targeted Return. Generally, when a user's query has a return node, or only one entity is modified by all predicates, Targeted Return can successfully infer the user's search target, and the query results are very close to, if not exactly the same as, user's expectation. Furthermore, for queries like Q_{35} where the search target, *subsection*, is a recursive element in the data, Targeted Return also produces desirable results by having each result contain one instance of *subsection*. This verifies the applicability of our heuristics of inferring target entities and center entities.

On the other hand, there are two cases that Targeted Return needs improvements (though it is not worse than existing approaches – Subtree Result and Pattern Match). First, Targeted Return may fail to infer center entities from multiple inferred target entities. In this case Targeted Return adopts Subtree Result for result generation, which may include multiple target instances in a result. An example is Q_2, where there are two target entities: *team* in the central division, or *player* named Abbott. The results of Targeted Return have one team in each result, which is not desirable as the user's search goal is to find the player named Abbott who plays in a team in the central division. Targeted search do not compose desirable results for Q_{23} and Q_{31} for the same reason. The other case is that the user searches for a relationship, rather than the information of a single entity, such as Q_{14}, which searches for the relationship of two players. Targeted Return infers *player* as the center entity, and outputs two players in each result. The results have overlaps, as each result is based on a single instance of *player* named Joey (Jim), and includes the matches of Jim (Joey) that are supporting nodes of *player*. This is undesirable as the results may be confusing and the users need more efforts to find the desired information. It is our future works to infer the correct center entity in face of multiple target entities, and deal with target relationships.

Processing Time

To test the processing time of each approach, we choose the Baseball data with size 1014KB, a portion of the Mondial data with size 515KB and the Bookstore data with size 423 KB.[4]

The processing time of each approach in log scale is shown in Figure 6.8 - 6.10.[5] When the processing time of a query is not perceivable, we set it as 0.002 second in order to be shown in the figures.

[4]We did not use larger data set because Timber reports error when evaluating most queries on larger data. Larger data are used in the scalability test.

[5]"e" means that the system reports error for query evaluation.

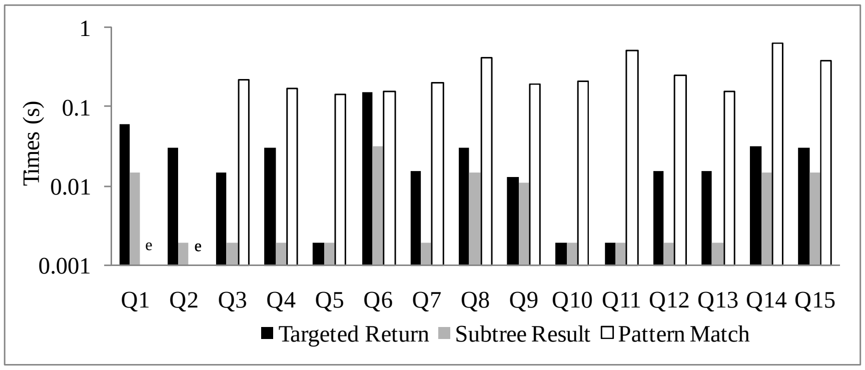


Figure 6.8: Processing Time of Targeted Return on Baseball Data

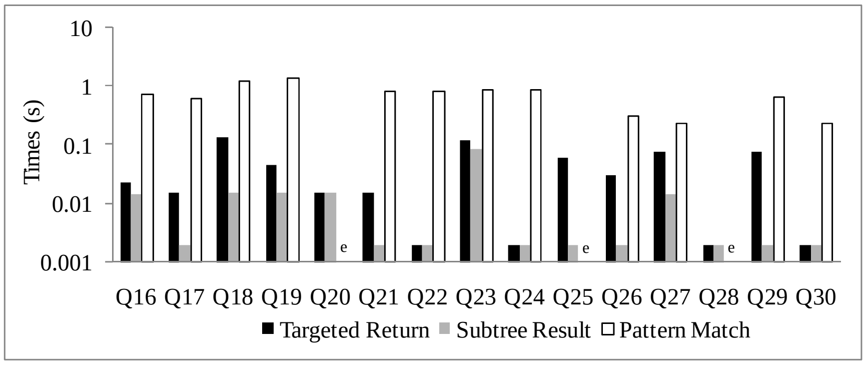


Figure 6.9: Processing Time of Targeted Return on Mondial Data

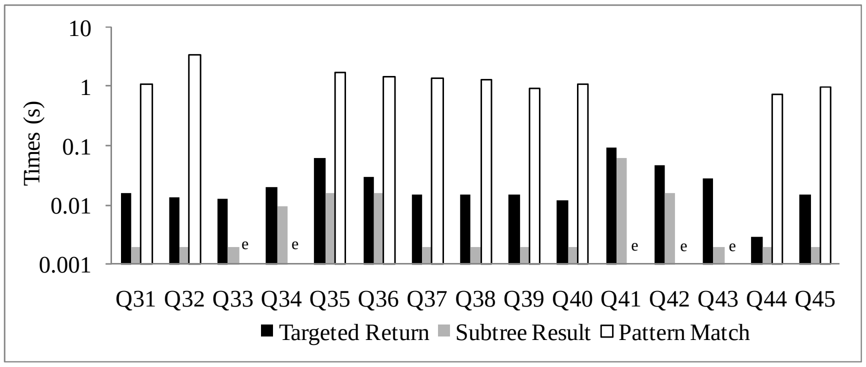


Figure 6.10: Processing Time of Targeted Return on Synthetic Data

As shown in the figures, the processing time of Targeted Return is always longer than that of Subtree Result. This is because both approaches need to calculate VLCA nodes and find relevant matches within VLCA nodes, and besides, Targeted Return also needs to differentiate return nodes and predicates in a query, find target entities from Dewey index when there are return nodes or identify target entities according to modifying relationship

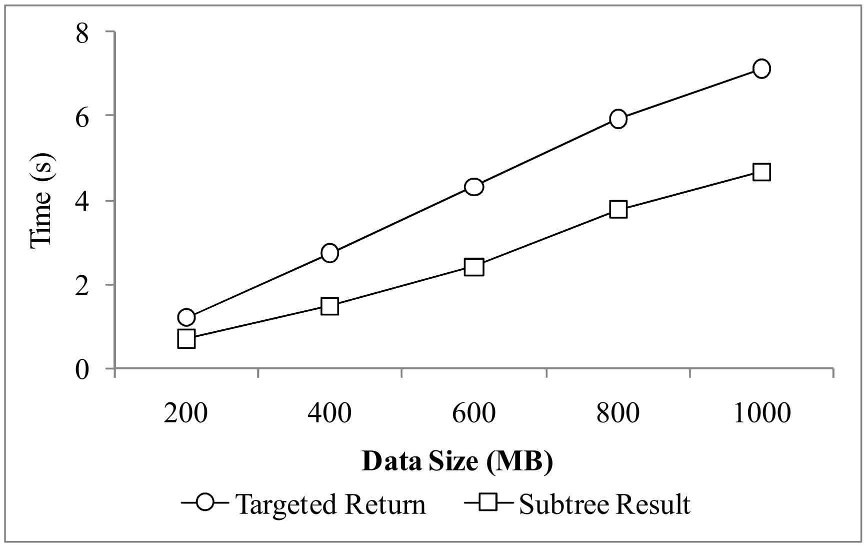


Figure 6.11: Scalability of Targeted Return over Data Size

among attribute values and relevant entities, and generate a query result for each center entity instance by outputting the matches of each keyword that are supporting nodes of the center entity instance, together with the paths connecting them when there is a center entity.

Therefore, when there is no center entity for a query, such as Q_9 and Q_{23}, the processing time of Targeted Return is almost the same as that of Subtree Result. For some queries that have center entity, The time consumed by Targeted Return is still close to that of Subtree Result, such as Q_{16}. When the number of instances of return nodes or center entity is large, Targeted Return needs to process each individual ones, group them according to VLCA nodes, and therefore takes a longer time, such as Q_{18}, Q_{37} and Q_{39}.

The processing time of Pattern Match is always longer due to the cost of enumerating patterns.

Scalability

We test the scalability of our approach with respect to data size. The data size is increased by replicating the data. The result of processing query Q_1 on the Baseball dataset whose size is increased from 200MB to 1GB is shown in Figure 6.11. The processing times of

Targeted Return and Subtree Result both increase linearly with the increase of the data size. Results of processing other queries are similar and are omitted.

In summary, the query results given by Targeted Return have much better quality than those generated by Subtree Result and Pattern Match. Targeted Return has a reasonable processing time overhead compared with the Subtree Return approach, and is more efficient than Pattern Match.

6.5 Summary

Properly defining query results has a significant benefit on query result ranking, top-k query evaluation as well as user-friendliness. From the semantics point of view, each user query has a search target. Therefore each query result should be atomic and intact, i.e., containing exactly a single instance of the search target along with all the relevant match or non-match nodes related to this instance. Subsequently, the ranking defined on query results should reflect the ranking of these target instances. A query result that contains multiple instances of a target entity, as produced by Subtree Result, overwhelms the user and messes up the ranking among target entity instances. On the other hand, multiple query results that correspond to the same target entity instance, as generated by Pattern Match, annoy the user with repeated result and disturb the ranking among target entity instances.

Our approach of composing query results is driven by user search targets, and produces atomic and intact query results. To identify user search targets, we infer return nodes for keyword searches and the modifying relationship among attribute values and entities in the data. Then we determine the target entities and center entity for a keyword search, based on which query results are composed. Experimental evaluation has shown the effectiveness and efficiency of our approach. The proposed query result generation technique can be adopted in existing XML keyword systems [43, 56, 61, 82, 86, 91, 93, 144].

Chapter 7

RESULT SNIPPET

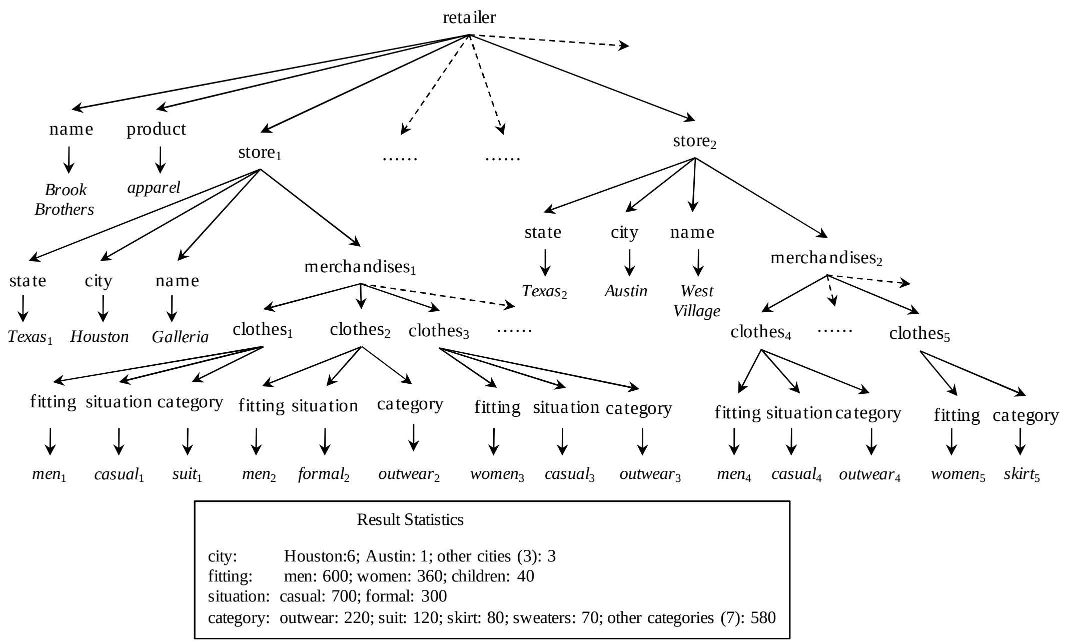
7.1 Motivation and Goal

As discussed Chapter 1, structural ambiguity, keyword ambiguity and user preference ambiguity can be alleviated by result analysis techniques. By helping users view and analyze the query results conveniently, the user will be able to quickly understand the structure of a result, the keyword semantics in a result, the comparison of different results, etc. In this chapter as well as the following few chapters, we will discuss result analysis techniques. In particular, this chapter discusses how to generate snippets for structured search results.

Result snippets help users quickly judge the relevance of query results by providing a brief quotable passage of each query result. Note that although various ranking schemes have been proposed to assess the relevance of query results, it is impossible to design a ranking scheme that always perfectly gauges query result relevance with respect to users' intentions. To compensate the inaccuracy of ranking functions, result snippets are used by almost every web search engine.

Compared with result snippets for text document search, structured data presents better opportunities for generating meaningful and helpful search result snippets. In document search, due to the lack of structure, a common strategy is to use document fragments that contain keywords along with the surrounding words as snippets in order to approximate a semantic summary of the document. On the other hand, structured data contains metadata, providing meaningful annotations to the data content, thus presenting a better hope of generating semantically meaningful snippets.

To generate meaningful snippets for keyword search on structured data, we begin with identifying the specific goals that a semantically meaningful result snippet should meet. First, different result snippets should be *distinguishable* from one another, so that the user can differentiate the results from their snippets with little effort. Second, a snippet should be *representative* to the query result, thus the user can grasp the essence of the result from its snippet. At last, a result snippet should be *small* so that the user can quickly browse



Result Statistics

city: Houston:6; Austin: 1; other cities (3): 3
fitting: men: 600; women: 360; children: 40
situation: casual: 700; formal: 300
category: outwear: 220; suit: 120; skirt: 80; sweaters: 70; other categories (7): 580

Figure 7.1: Part of a Query Result and Statistics about Value Occurrences.

several snippets. However, achieving these goals is highly challenging.

To illustrate these goals, we use query Q_8 in Figure 4.1 on the XML data in Figure 1.3. A sample result of this query is shown in Figure 7.1. Some statistics of the full query result in this example are presented at the bottom of the figure, where for each distinct name-value pair, we record the number of its occurrences in the query result. For instance, “city: Houston: 6" indicates that there are 6 occurrences of *Houston* as a value of node *city* in the query result. Values with low occurrences are omitted.

The first goal of snippets is to allow the user to easily distinguish different query results from one another. To achieve this in text document search, result snippets often include the document titles. Analogously, we propose to select the unique identifier (aka. key) of a query result into its snippet to identify this query result and highlight the fundamental differences among results. Intuitively, a node in a query result is a key if the values of the nodes with the same name are all distinct in all the query results. A plausible solution would use such nodes as the keys of a query result. For example, we may consider that *name* of a *store* as a key of the results of query *Texas apparel retailer*. However, this is unlikely to be reasonable, as the user searches for *retailer*, which can have hundreds of stores. It is an open question how to infer the key of a query result.

The second goal is to design snippets that provide representative summaries of the query results by including the most prominent features in each result. Intuitively, a prominent feature is often reflected by a large number of occurrences of such a feature in the result. Continuing our example, suppose *Brooks Brothers* has 1000 clothes of different styles, among which 600 are for men and 40 for children. Therefore including clothes of fitting *men* instead of *children* in the snippet shows a prominent feature of this query result: this retailer targets clothes for men. However, the relationship between the prominence of a feature and the number of occurrences is not always reliable. In our example, the number of occurrences of *Houston*: 6, is much less than that of *children*: 40. However, considering that the majority of *Brooks Brothers* stores are in *Houston* in the query result, *Houston* should be considered as a prominent feature.

Besides, the prominence of a feature can also be measured by its general importance in other results, analogous to measuring the importance of a keyword via inverse document frequency (IDF). Similar as IDF, a feature with many occurrences in many results should be considered as a non-prominent feature. Consider a result of query "*Esprit, store, shirt*" in Figure 7.4(a). It is common sense that most shirts are made of cotton, thus cotton is an uninteresting and non-prominent feature. This is captured by the fact that *cotton* has many occurrences in all results.

Furthermore, a snippet should be faithful to the original query result. This could be achieved by keeping the distributions among the selected values as much as possible. For example, the query result in Figure 1.5 about *Brooks Brothers* retailer has 600 different clothes styles for *men*, 360 for *women* and 40 for *children*. Suppose another query result is about *Talbots* retailer which has 500 different clothes styles for *women*, 280 for *men*, and 20 for *children*. As we can see, although both retailers sell clothes for men and women, clearly they have different specialities. A good snippet should reflect the differences in terms of occurrences of features in the corresponding result.

The two goals discussed above address the requirements on the semantics of a result snippet. Clearly the larger a snippet is, the more information it has and the better it

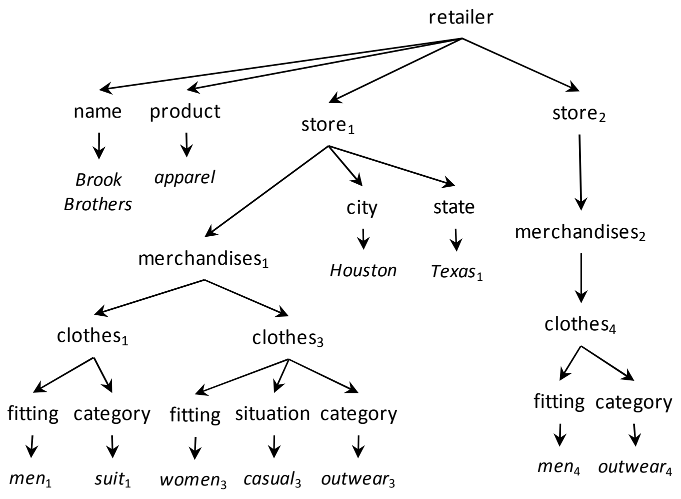


Figure 7.2: A Snippet of the Query Result in Figure 1.5

can meet these goals. A trivial solution to generate snippets that meet these goals could be using the query result itself as the snippet. Nevertheless, this is obviously undesirable. The last goal specifies a conflicting requirement: a snippet should be small so that a user can quickly and efficiently browse and understand snippets of several query results. Therefore the challenge is how to provide as much information as possible in a snippet to meet the first two goals within an upper bound of the snippet size.

In this chapter we address the problem of generating effective snippets for structured search results, and comprehensively test the snippets generated our approach in terms of quality, efficiency, scalability, as well as the effectiveness of grouping query results based on snippets.

We assume a query result is subtree of the data (which is either a tree or a graph). A snippet of a result R is a subtree of R. There are two steps to generate result snippets.

Step 1: Identifying Important Information in the Result (Section 7.2). In this step, we identify what information in the result serves as best summarization of the result, hence should be included in the snippet. To do so, we identify three desirable properties of a snippet: a snippet should be an information unit of *a bounded size* that effectively *summarizes* the query result and *differentiates* itself from others. To achieve this, we analyze the semantics of the query result. We identify the key of the result as well as the prominent features in the result. We put this information in a *snippet information list*, in which items are ordered by their importance to be included in the snippet.

A feature in a query result is defined as an “entity:attribute:value” triplet, such as “clothes:fitting:men”. Features can be identified from search results on many common data models, e.g., XML and relational databases. Data in relational database are often organized based on the Entity-Relationship model, from which we can identify entities. Column name and the value in each cell can be considered as attribute and value, respectively. For XML data, it is modeled as a rooted labeled tree. Entities and attributes can be identified by the heuristics discussed in Chapter 5. Specifically, a node is an entity if it corresponds to a *-node in the DTD, i.e., it has siblings with the same label. A node is an attribute if it is not an entity and has only one leaf child.

Given a result, we initialize the snippet information list with the keywords in the keyword query, as at least one match to each keyword should be included in the snippet. For example, if the result in Figure 1.5 is obtained by a keyword query “Texas, apparel, retailer”, then $IList$ initially consists of these three keywords as shown in Step 1 in Figure 7.3. In the next several subsections we discuss what other items should be added to the list in order to make the snippet distinguishable and representative.

Step 2: Selecting Instances of Items in the Snippet Information List (Section 7.3). An item in the snippet information list may have multiple occurrences in the result. Selecting different occurrences to compose the snippet will lead to different snippet sizes. Since snippet should be concise, we would like to select as many items in the snippet information list as possible given a snippet size limit. We show that this problem is NP-hard. A novel algorithm is proposed that efficiently generates semantic snippets with a given size bound for structured search results.

As an illustration, the snippet for the query result in Figure 7.1 is shown in Figure 7.2. It captures the heart of the query result in a small tree: the query result is about retailer *Brooks Brothers*, which has many stores in *Houston* and features *casual* clothing for *men* more than *women*. While for categories of clothing, it features mainly *suit*s and *outwear*s.

7.2 Identifying Significant Information in Query Results

We have discussed three goals in generating result snippets for structured search results and the challenges to achieve them. In this section we discuss how to tackle the challenges to meet the first two goals, such that the most significant information in a query result to be selected in its snippet is identified.

Distinguishable Snippets

Goal 1: A snippet should make the corresponding query result distinguishable from (the snippets of) other query results such that the users can differentiate them with little effort.

As discussed in Section 7.1, we propose to select the key of a query result into its snippet, reminiscent to the document title in result snippets in text document search, such that a query result can be identified and differentiated from other results.

However, it is not obvious how to identify the key of a query result, as a query result may contain several entities, each of which has a key. Thus the question is which entities' keys should be considered as the key of the query result. Intuitively, each query has a search goal. The search goal can be used to classify the entities in a query result into two categories.

1. *return entities* are what the users are looking for when issuing the query.
2. *supporting entities* are used to describe the return entities in a query result.

Since return entities are the core of a query result, their keys can function as the key of the query result and can be used to differentiate this query result from others. There can be many heuristics for inferring return entities, and we adopt the following heuristics: An entity in a query result is a *return entity* if its name matches a query keyword or its attribute name matches a keyword. If there is no such entity, that is, no keyword matches node names, then the highest entity (i.e., entities that do not have ancestor entities) in the query result are considered the default return entity.

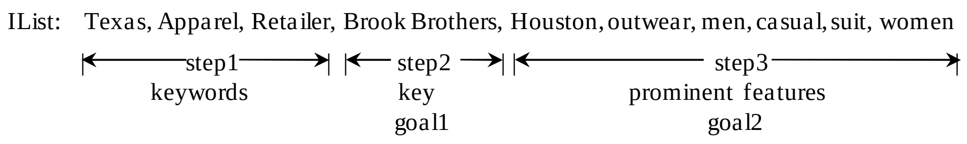

IList: Texas, Apparel, Retailer, Brook Brothers, Houston, outwear, men, casual, suit, women
step1
keywords
step2
key
goal1
step3
prominent features
goal2

Figure 7.3: $IList$ of the Query Result in Figure 1.5

Example 7.1 *In our running example, for query* Texas apparel retailer*, entity* retailer *matches a keyword, and therefore is considered as a return entity, corresponding to the user's search goal. The key attribute of* retailer*:* name *is considered the key of this query result, and is added to the snippet information list. The current $IList$ comprises the first two steps in Figure 7.3.*

The key of XML nodes/database tuples can be directly obtained from the schema or DTD, if available. Otherwise, we find the most selective attribute of the return entity and use it as the key. Specifically, for all query results, we find the attribute of the return entity that has the fewest duplicate values.

Representative Snippets

Goal 2: A snippet should be representative of the query result, so that the users can grasp the essence of the result from its snippet.

Similar as text document search, a snippet should provide a summary of the query result. A good summary should be concentrated on the most prominent features of a query result, and at the same time be faithful to the meaningful statistical value distribution in the query result.

We define a *feature* in the result as a triplet (entity name e, attribute name a, attribute value v). For example (*store*, *city*, *Houston*) is a feature. The pair (entity name e, attribute name a) is referred as the *type* of a feature, and attribute value v is referred as the value of a feature. If the number of words contained in a text value is larger than a threshold (which is likely a long text value consisting of sentences/paragraphs), then each individual word is considered as a distinct feature value. Intuitively, outputting sentences/paragraphs is space-consuming, and the significance of the sentences is hard to judge, thus we choose to output statistically representative words instead. The threshold can be flexibly set; in our implementation and experiments, the threshold is set as 5. For this type of long values, we can also consider each important phrase as a single feature value by applying the phrase identification techniques in the literature, such as [88] and [135]. Note that the entity name is taken into account because different entities may share the same attribute names. For example, both *retailer* and *store* have attribute *name*. For presentation purpose, we refer to a feature by its value when there is no ambiguity.

Next we will discuss two factors that affect the prominence of a feature: feature dominance (FD) and inverse result dominance (IRD). We will then discuss feature value distribution.

Feature Dominance

As discussed in Section 7.1, a dominant feature of a query result is often reflected by a large number of occurrences of the feature in the result. For example, the fact that there are more clothes for *men* than *children* in the query result indicates that *Brooks Brothers* is specialized for *men* instead of *children* clothes.

However, the relationship between the dominance of a feature and the number of occurrences is not reliable due to two reasons. First, different features have different domain sizes. The domain size of a feature type (e, a) is defined as the number of distinct values (e, a, v) of this type, denoted as $D(e, a)$. The smaller size a domain has, the more chances for a value to have more occurrences in the result. For example, the number of occurrences of *outwear*: 220, is less than that of *women*: 360. However, considering the domain sizes of their corresponding feature types in the query result: D(*clothes*, *category*) = 11, D(*clothes*, *fitting*) = 3, *outwear* could be more dominant than *women* in their respective domains.

Second, due to the tree structure of the query result, different features have different total number of occurrences in the query result, denoted as $N(e, a)$. The more occurrences of a feature type, the more chances that a value of this feature type to occur. For example, a value *Houston* only occurs 6 times, while *children* occurs 40 times in the query result. However, considering the number of occurrences of their corresponding feature types: N(*store*, *city*) = 10, N(*clothes*, *fitting*) = 1000, *Houston* is likely to be more dominant than *children*.

As observed from these examples, comparing the number of occurrences of values of different feature types may not make sense in determining dominant features. To quantify the above intuition, we propose to use normalized frequency, called *feature dominance*, to measure the significance of a feature in a query result.

Definition 7.1 *We define* feature dominance *of a feature $f = (e, a, v)$ as follows:*

$$FD(f, r) = \frac{N(e, a, v)}{\frac{N(e,a)}{D(e,a)}} \tag{7.1}$$

where R is a query result, $N(x)$ denotes the number of occurrences of x in R, $D(e, a)$ denotes the domain size of (e, a) in R.

Example 7.2 *Continuing our example, we compute the feature dominance for features in the query results. In the following the corresponding feature types are omitted for conciseness.*

*FD(*Houston*) = 6 / (10 / 5) = 3.0*

*FD(*men*) = 600 / (1000 / 3) = 1.8*

*FD(*women*) = 360 /(1000 / 3) = 1.08*

*Similarly, we get FD(*casual*) = 1.4, FD(*outwear*) = 2.2, and FD(*suit*) = 1.2.*

In summary, a feature has a higher feature dominance if it occurs more frequently in the result compared with other features of the same type.

Inverse Result Dominance

While feature dominance measures one aspect of the prominence of a feature, it is not sufficient. Recall that in information retrieval, the importance of a keyword in a text document is measured from two aspects: its frequency in the document (TF) and the appearance of the keyword in all documents (IDF). TF measures the importance of a keyword in a single document in terms of its frequency in the document; IDF measures the general importance of keywords in terms of how many documents contain this keyword. It is easy to see that feature dominances resemble term frequencies by evaluating the feature within a result. Analogously, a feature is prominent if it is dominant in only a small number of results, as to be shown in the following example.

Example 7.3 *Consider a query "*Esprit, store, shirt*" on an XML document about apparel retailers. Each result is a store that sells shirts, one of which is shown in Figure 7.4 (a). Suppose that the majority of shirts are made of cottons, and that the Esprit store in each result either only sells or mainly sells women clothes, while in the result in Figure 7.4 (a), 40% of the shirts are for men, which is unusual.*

*Two possible snippets of the result in Figure 7.4 (a) are shown in (b) and (c). The snippet in (b) is generated according to feature dominance discussed before, which consists of the features with the highest feature dominance. The one in (c) does not have feature (*clothes,material,cotton*), but shows a feature (*clothes,fitting,men*), which has a lower feature dominance in the result. The snippet in (c) is more informative, as* cotton *has a high feature dominance in all results, and thus outputting* cotton *is not interesting or helpful. Indeed, it is common sense that most shirts are made of cotton. On the other hand, the result in Figure 7.4 (a) has unusually many shirts for men compared with other results, so it is desirable to instead show the feature* men*, which informs the user of the uniqueness of this store.*

As we can see, to identify prominent features, we should not only use the feature dominance which is a score within a result, but also measure its dominance over all results. However, it is an open question how to measure it and it is challenging to do so. Due to the differences between features and keywords, the formula of IDF cannot be directly applied for computing the weight of a feature. Recall that IDF is calculated as:

$$IDF(k) = \log\frac{|D|}{|D_k|}$$

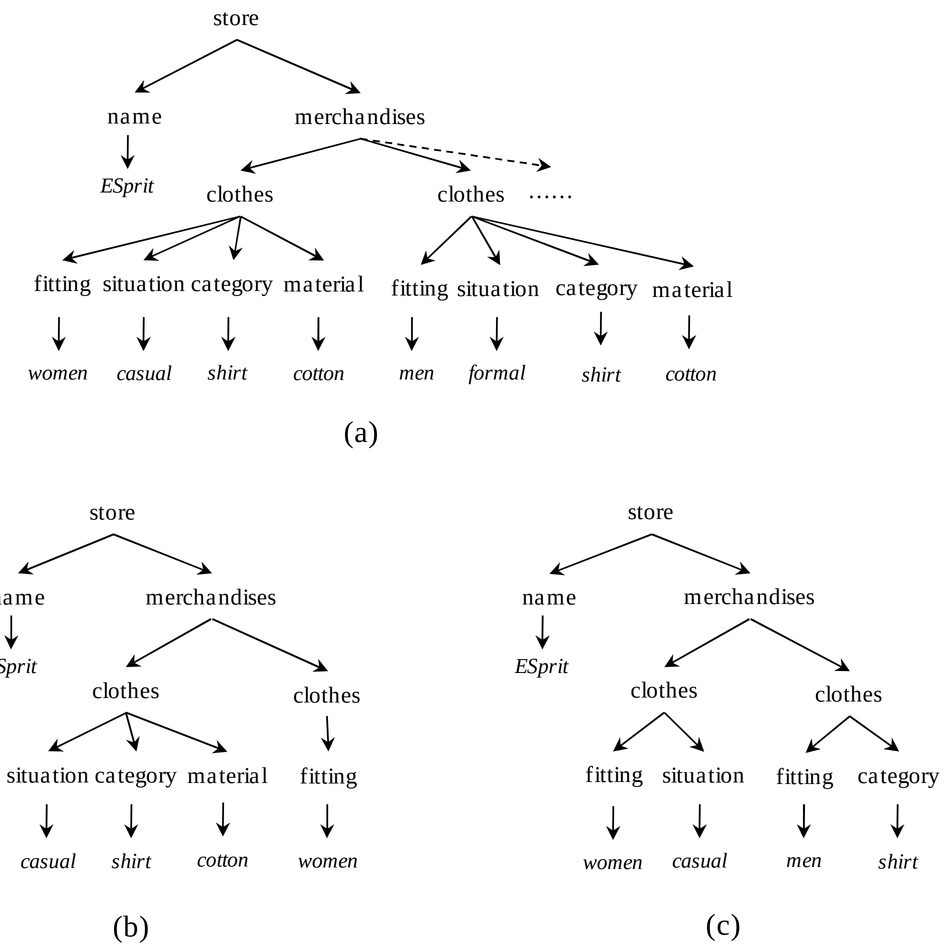


Figure 7.4: Query Result Fragment and Snippets of Query “*Esprit, store, shirt*”

where k is a keyword, $|D|$ is the number of documents and $|D_k|$ is the number of documents that contain keyword k. There are two important differences between the importance of a feature and the importance of a keyword (IDF).

First, IDF considers the frequency of a keyword in the universe of text documents; however, we propose to use the set of results of a query, rather than the entire data repository, as the universe, to measure whether a feature is interesting with respect to a query. The reason is that whether a feature is prominent or not varies by queries. For example, recall the query “*Esprit, store, shirt*”. By common sense, cotton is unlikely an interesting feature as most of the shirts are made of cotton, which can be captured by the fact that *cotton* has a high feature dominance in almost all results. However, *cotton* can be a prominent feature for other queries such as “*store, outwear*”, if it has a small number of occurrences in most results.

Second, even if a feature appears in all results, it may still be a prominent feature and its weight should not be zero. Consider query "*Esprit, store, shirt*" again, if each store sells at least one shirt for *men*, then feature *men* appears in all results. According to the IDF formula, *men* should get a weight of 0, which is implausible. In fact, if *men* occurs only a few times in most results but occurs more frequently in one result, then it is likely an interesting feature to be included in the snippet of this result. Thus a feature should have a low weight only if it has a high feature dominance in most results. As we can see, the second difference between IDF and feature weight is: the number of documents are huge and are from different domains, so a keyword that appears in many or all documents must be a very unimportant word, such as a stop word. On the other hand, the results of the same query are generally similar with each other, thus it is possible that they have many common features, and these features should not be regarded as useless.

Despite these differences, the idea that a feature occurring in a small number of results is important, is still valid. We define the *Inverse Result Dominance* (IRD) of a feature in a way similar as IDF of a keyword, which is tailored for computing the weight of a feature under the guidance of the two differences mentioned above. We use the number of query results, rather than the number of all structured data, as the numerator. Besides, instead of using the number of results containing a feature as the denominator, we measure the total feature dominance of the features in all results. The *inverse result dominance* of a feature f, $IRD(f)$, is computed via the following formula:

$$IRD(f) = \log_2\left(\frac{|R|}{\sum\limits_{r' \in R} FD(f, r')} + 1\right) \tag{7.2}$$

According to Formula (7.2), if the average feature dominance of feature f over all results is lower than 1, which means that f is on average a non-dominant feature, then its IRD is higher than 1, and vice versa.

We define the score of a feature as the product of its feature dominance and inverse result dominance:

$$score(f, r) = FD(f, r) \times IRD(f) \quad (7.3)$$

Definition 7.2 *A feature f is a* prominent feature *in query result r if $score(f, r) \geq 1$.*

Example 7.4 *Consider the query result in Figure 7.4. Let r denote the result shown in the figure. Assume that the average feature dominance of feature* cotton *and* men *is as follows:*

	average FD in all results	*FD in result r*
cotton	*10*	*10*
men	*0.1*	*0.8*

Then according to Formulae 7.2 and 7.3, $IRD(cotton)$ = 0.14, $score(cotton, r)$ = 1.4. On the other hand, $IRD(men)$ = 3.46, and $score(men, r)$ = 2.77. As we can see, men *now gets a higher score and hence a higher priority over* cotton*, and will be output in the snippet before* cotton*.*

We include the prominent features into the snippet information list in the decreasing order of their scores, which now contains the items in all three steps in Figure 7.3.

Faithful Summary of Results

As we have discussed, the prominent features are added to the snippet information list in order to present a representative summary of the query result. For a snippet to be faithful to the original query result, it should also try to keep the meaningful distributions among the select features as much as it can. Being unaware of the relative significance of different features, the users may not be able to determine whether a query result is relevant or not.

Recall the example discussed in Section 7.1. Both *men* and *women* are prominent features, with different numbers of occurrences in the query result about retailer *Brooks Brothers* in Figure 1.5. Suppose there is another query result about *Talbots* (not shown in the figure), which sells clothes for both *men* and *women*, but focusing on *women*'s. Including

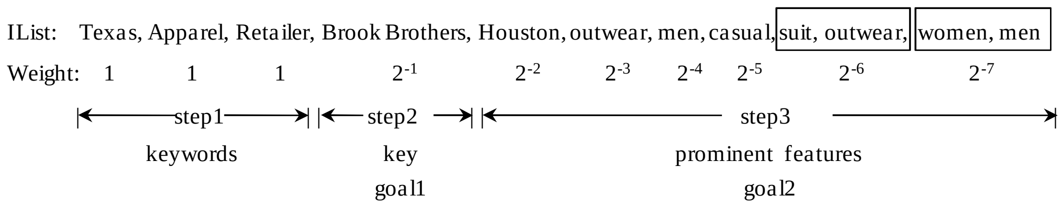

IList: Texas, Apparel, Retailer, Brook Brothers, Houston, outwear, men, casual, suit, outwear, women, men
Weight: 1 1 1 2-1 2-2 2-3 2-4 2-5 2-6 2-7
step1
step2
step3
keywords
key
goal1
prominent features
goal2

Figure 7.5: Adjusted $IList$ of the Query Result in Figure 1.5

one *men* and one *women* in the snippet for each query result can mislead users about their specialization.

To better capture the characteristics of the original query result, for the prominent features of the same type in the snippet, we propose to preserve the ratios of their number of occurrences. For example, since *Brooks Brothers* has 600 clothes for *men* and 360 for women, selecting 5 clothes for *men* and 3 clothes for *women* shows that this retailer targets more for men than women. Note that including the minimum number of occurrences that can keep the ratio already makes the information saturated. For example, showing 10 *men* and 6 *women* for *Brooks Brothers* does not add any additional information in the snippet, and therefore is unnecessary.

Note that keeping the ratio across different feature types won't make much sense, as the user rarely compares the values from different features. For example, it is not comparable whether the retailer has more clothes for *men* or has more clothes in the *outwear* category. We also do not keep the ratio among entities as it can be too costly to be meaningful. For example, if each *retailer* has 20 *store*s, each of which has 20 to 1000 *clothes*. Keeping the ratio among *retailer*, *store* and *clothes* in a reasonably small snippet is difficult.

In fact, due to space limitation, we often keep a rough ratio among features. Let m_f be the maximum number of prominent features that we want to select into the snippet per feature type (e, a). Then the number of occurrences of a feature (e, a, v) in the snippet is set to:

$$occ(e, a, v) = \lfloor m_f \times N(e, a, v) / \sum_{j=1}^{n} N(e, a, v_j) \rfloor \tag{7.4}$$

Where (e, a, v_j), $1 \leq j \leq n$, is a prominent feature and $N(x)$ is the number of occurrences of x.

Based on the above discussion, we adjust the snippet information list of a query result such that multiple appearances of the same features may be included, add the prominent features into the snippet information list, each of which may have multiple appearances, in order to keep the distribution information of this feature type.

Example 7.5 *Referring to the snippet information list of the sample query result in Figure 7.3. Note that feature* outwear *and* suit *are of the same type (*clothes*,* category*). In the query result, the number of occurrences of* outwear *and* suit *are 220 and 120, respectively. Assuming the upper bound for each feature type in the snippet is 4, we select 220*(4/(220+120))=2 clothes for* outwear *and 120*(4/(220+120))=1 for* suit *into the information list. Similarly we choose 2 clothes for* men *and 1 for* women*.*

The snippet information list is adjusted to reflect the distribution, as shown in Figure 7.5.[1] Compared with Figure 7.3, when it comes to *suit*, since a feature of the same type, *outwear*, is already in the $IList$, therefore to keep their ratio we should output another *outwear* together with *suit*. Similarly, a *men* node needs to be output together with *women*. Note that since a snippet has a limited size, we might not be able to include all the features necessary to keep the faithful ratio of prominent features. For example, for features *suit, outwear* bounded by black box in Figure 7.5, suppose we can include feature *suit* in the snippet, but then we do not have enough space to include *outwear*. In this case, only including *suit* will mislead the users for the same reason as we have mentioned before, and only including *outwear* is useless. Therefore, in this case we include neither *suit* or the additional *outwear* in the snippet. Generally, for the items in the same rectangle in $IList$, they are either included together or none of them is included in the snippet.

[1]The weight of each item in the figure is used in the snippet generation algorithm which will be discussed in Section 7.3.

Algorithm 4 Construction of Snippet Information List

```
keywordSearch ($Q, D$)
1:  $QR[m]$ = obtainResults($Q, D$) {using any existing search engine}
2:  constructIList($QR[m], Q$)
3:  for $i = 1$ to $m$ do
4:      selectInstance($QR[i], IList[i], sizeLimit$)
constructIList ($QR[m]$, $Q$) {Generate IList for all query results}
1:  for $i$ = 1 to $m$ do
2:      $IList[i] = \emptyset$
3:      processQR ($QR[i]$, $Q$, $IList[i]$)
4:  compute the IRD and score of each feature using Formula (7.2) and (7.3)
5:  for $i$ = 1 to $m$ do
6:      for each feature $f = (e, a, v)$ do
7:          if $score(f, QR[i]) \geq 1$ then
8:              $IList[i] = IList[i] \cup \{f\}$
9:      adjustIList($IList[i]$)
10: sort all features in $IList$ by their scores
11: return $IList$
processQR ($QR$, $Q$, $IList$) {Traverse a query result to identify return entities and count feature
occurrences}
1:  $returnEntity = \emptyset$
2:  $IList = IList \cup Q$
3:  for each node $n$ in a pre-order traversal of $QR$ do
4:      if $n$ is an entity then
5:          $e$ = $n.name$
6:          if $e$ matches a keyword then
7:              $returnEntity = returnEntity \cup \{e\}$
8:      else if $n$ is an attribute name then
9:          $a$ = $n.name$
10:         $N(e, a) + +$
11:         if $a$ matches a keyword then
12:             $returnEntity = returnEntity \cup \{e\}$
13:     else if $n$ is a value then
14:         $v$ = $n.value$
15:         $N(e, a, v) + +$
16:         if triplet $(e, a, v)$ has not appeared before then
17:             $D(e, a) + +$
18: $IList = IList \cup$ the set of key attribute values of $returnEntity$
19: for each feature $f = (e, a, v)$ do
20:     compute $FD(f, QR)$ using Formula (7.1)
```

Algorithm for Snippet Information List Construction

As has been discussed, the snippet information list $IList$ contains the following three components in order. Each item in the list is referred as an *informative item*.

1. Keywords;

2. The key of the query result, represented by the keys of the return entities, contributing to distinguishable snippets;

3. An ordered list of prominent features whose distribution among all prominent features of the same type is preserved, contributing to faithful summaries of query results.

The algorithm of constructing IList is shown as procedure constructIList in Algorithm 1. It is called in procedure keywordSearch after query results are generated. It first processes each query result (procedure processQR) to add keywords and keys into each snippet, and calculate the feature dominance of features in each result. Then, using the feature dominance of features in all results, it computes the IRD and score for each feature (line 4) and add the prominent features into each IList (line 5-9).

In procedure processQR, we use e, a and v to denote the last entity name, attribute name and attribute value that have been encountered during the traversal, respectively. First, we add the keywords into $IList$ (line 2). To add keys and prominent features, we perform a pre-order traversal of the query result, as shown in procedure processQR in Algorithm 1. The feature dominances of the features in each result are also computed during the traversal. For each node n visited in the traversal, if n is an entity, we set e as $n.name$ (line 5). We consider e as a return entity if e matches a keyword (line 6-7). If n is an attribute name, we set a as $n.name$ (line 9), and increase the number of occurrences of feature type (e, a) (line 10). If a matches a keyword, entity e is considered as a return entity (line 11-12). If n is an attribute value, we set v as $n.value$ (line 14), increase the number of occurrences of feature (e, a, v) (line 15) and increase the domain size of feature type (e, a) if feature (e, a, v) has not appeared before (line 16-17). Then we add the key attribute values of the return entities into $IList$ (line 18), and calculate the feature dominance of each feature (line 19-20).

After all query results are processed, we compute the IRD and the score of each feature using Formulae (7.2) and (7.3). Then we add prominent features into each IList and sort the features in each IList in descending order of their final scores.

Algorithm 5 Adjusting Snippet Information List

```
adjustIList ($initIList$) {Adjust an IList to keep feature distributions}
 1: $IList = \emptyset$
 2: for each item $i$ in $initIList$ do
 3:    if $i$ is a keyword or key then
 4:       $IList = IList \cup i$
 5:    else
 6:       {$i$ is a prominent feature}
 7:       if Feature $j$ of the same type as $i$ has already been added into $IList$ then
 8:          Calculate the number of occurrence of $j$ and $i$ need to be added, based on the statistics
 9:          Add the corresponding number of $j$ and $i$ into $IList$
10:       else
11:          $IList = IList \cup i$
12: return $IList$
```

It is worth mentioning that despite the requirement that IRDs of features are computed at query time, as shown in the experiments in Section 7.4, when there are no more than several hundred query results, the overhead of IRD computation is negligible. When the number of query results is large, we can use only the top-k results for computing IRDs, where k can be flexibly chosen, provided that the search engine produces ranked results. This is reasonable as the user may only care about the top-k results.

Now we analyze the time complexity of Algorithm 1. A hash index was built off-line on the data, which takes an input of a node ID and returns the information about this node, such as node type (entity, attribute, or connection node) and key values (if exists). Hash indexes are also built to access $N(e, a)$, $D(e, a)$ and $N(e, a, v)$ in $O(1)$. Therefore the cost of traversing the query result (line 3-17 of processQR) is bounded by the size of the query result, $O(|QR|)$. Since the number of prominent features is bounded by $|QR|$, computing their feature dominances (line 19-20) also takes $O(|QR|)$ time. Computing the IRDs and scores of all features takes $O(|QR| \times m)$ time, where m is the number of results. Sorting an IList (Line 26) takes $O(|L|\text{log}|L|)$, where $|L|$ is the size of $IList$. Therefore, the complexity of constructing ILists for all results is $O(m \times (|QR| + |L|\text{log}|L|))$.

After adding keywords, keys and prominent features into $IList$, we adjust the number of prominent features in $IList$ to keep the ratio of prominent features, so that the snippet is faithful to the corresponding query result. The $IList$ can be adjusted as shown in Algorithm 2. We use $initIList$ to denote the initial $IList$ and adjust the items in $initIList$.

First we add the keywords, entities and keys into $IList$ (line 3-4). Then for each prominent feature i in $initIList$, if a feature j of the same type has already been included in $IList$, adjust the number of prominent features according to statistics, i.e., put i and j in the same rectangle to enforce that they either both appear or neither appears (line 9-10). Otherwise, simply add this feature into $IList$ (line 12). Apparently a linear scan of the $initIList$ is sufficient for adjusting the number of prominent features and therefore the complexity is $O(|initIList|)$, where $|initIList|$ denotes the length of $initIList$.

In the following section, we discuss how to extract data nodes from the query result to capture the items in the list as much as possible.

7.3 Generating Small and Meaningful Result Snippets

We have discussed how to identify the snippet information list for a query result, which contributes to a representative and distinguishable snippet. Besides the requirements on the semantics, we also need to meet a conflicting goal on snippet size.

Goal 3: A query result snippet should be small so that the user can quickly browse several snippets.

Problem Definition

The challenge is given an upper bound on the size, how to include as many items in the snippet information list as possible into the snippet to make it maximally informative.

Recall that an informative item in the list, such as a keyword and a prominent feature value, can have multiple occurrences in the query result. Although the instances of the same informative item are not distinguishable in terms of semantics, different instances have different impacts on the size of the snippet. To include an instance of an informative item in a tree-structured snippet, we need to add a path to the snippet from its nearest ancestor in the query result that is already in the snippet to this node. Therefore we should carefully select instances such that they are close to each other in order to capture as many informative items as possible given the size limit of the snippet. For example, considering the instance of *Houston*, to capture informative item *outwear*, choosing instance *outwear*$_3$

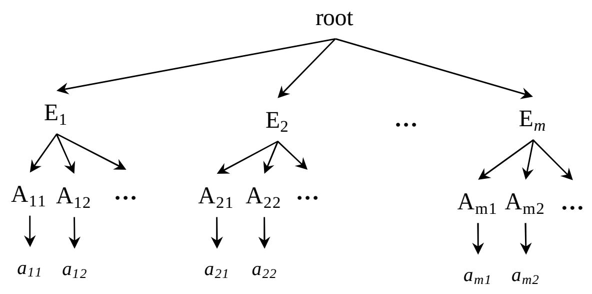


Figure 7.6: Reduction from Set Cover

results in a smaller snippet tree compared with *outwear*$_4$.

However, the problem of maximizing the number of informative items selected in a snippet given the snippet size upper bound is hard. We prove that its decision problem is NP-complete as follows.

Definition 7.3 *For a query result tree T, let $label(u)$ be the label of node $u \in T$, and $label(T) = \bigcup(label(u) \mid u \in T)$. The tree size $|T|$ is the number of words in T. We use a boolean $cont(T, v)$ to denote whether tree T contains a label v.*

Given a query result tree T, an integer c, and a set P of labels v, $v \in label(T)$, the instance selection problem *is to find T's subtree T', such that $|T'| \leqslant c$, and $\forall v \in P$, $cont(T, v) = true$.*

The problem can be illustrated as: given a tree T, a set of labels P and a size bound c, whether it is possible to find a subtree T' of T, such that T' contain every label in P and has a size no more than c.

Theorem 7.1 Instance selection problem *is NP-Complete.*

Proof. It is easy to see that this problem is in NP. Given a T's subtree T', we can check in polynomial time whether $|T'| \leqslant c$, and $\forall v \in P$, $cont(T, v) = true$.

Now we prove that it is NP-Complete by reducing the set cover problem to it, Set Cover $\leq_P$ Instance Selection. Recall the set cover is the following problem: given a universe $U = \{a_1, a_2, \ldots, a_n\}$, a collection C of m subsets $s_i \subseteq U (1 \leqslant i \leqslant m)$ and an integer k, *can we select a collection C' of at most k subset in C, whose union is U, i.e., $\bigcup(s \mid s \in C') = U$?*

For any instance of set cover, we construct a tree as shown in Figure 7.6. For each $s_i \in C$, we construct a node E_i, whose parent is root. *Let the j-th element in s_i be a_{ij}. For each a_{ij}, we create a node A_{ij} with value a_{ij} as a child of E_i. Except leaf nodes, no node label is the same as an element in U. For every $a_i \in U$, we have a corresponding label a_i and let P denote the set of such labels. Let $c = k + 2|U|$. This transformation takes polynomial time. Next we show that this transformation is a reduction: the set cover problem can be answered if and only if this constructed instance selection problem can be answered.*

Given an answer to set cover, we obtain T's subtree T' as follows: $\forall s_i \in C'$, we select E_i and a subset of its children, such that every element $a_i \in U$ has exactly one leaf node in T with value a_i selected. Such leaves along with their ancestors up to the $root$ compose T'. Now we have $\forall a \in P$, $cont(T', a) = true$, and $|T'| \leqslant k + 2|U| = c$. T' is an answer to the instance selection problem.

Given an answer to the instance selection problem T', $|T'| \leqslant c$ and $\forall a \in P$, $cont(T', a) = true$. Let R denote the set of nodes E_i in T'. $2|U| + |R| \leqslant |T'| \leqslant c$, therefore $|R| \leqslant c - 2|U| = k$. Let C' be the collection of set s_i, such that $E_i \in R$. The union of s_i is U and $|C'| = |R| \leqslant k$, therefore C' is an answer to the set cover problem.

Algorithm for Instance Selection

As has been shown, the decision problem of instance selection is NP-complete. However, snippet generation must be efficient as web users are often impatient. We propose a greedy algorithm that efficiently selects instances of informative items in generating a meaningful and informative snippet for each query result given an upper bound on size.

There are several challenges in instance selection. First, nodes in the result interact with each other. Selecting each individual node in isolation can result in a large snippet. Second, the cost associated with a node, measured by the number of words to be added to the snippet if this node is selected, changes dynamically during the selection procedure.

Third, due to dynamic costs of node selection, we are not able to determine the number of informative items that can be covered till the very end.

Next we discuss how to address these challenges, by effectively determining the data unit for selection, measuring the benefit and cost of each selection, and designing an efficient instance selection algorithm.

Since informative items in the information list have different priorities, we assign weights to these items. Though we know that the items will be selected in the order of their appearance in the information list until the snippet size limit is reached, we are not able to determine the number of items to be selected beforehand. Note that an item in the list should not be considered before all its precedents are chosen to be in the snippet, otherwise the dominance of different features of the query result can not be faithfully reflected in the snippet. For example, for two features *men* and *women*, if score (*men*) is bigger than score (*women*) but *men* has a bigger cost to output than *women*, then although outputting *women* saves space, it also gives the user the misleading information that the store sells more clothes for *women* than *men*. Therefore, we choose to enforce the order of features to ensure that each snippet is a faithful summary of the result.

Based on this, we assign the weights to the items in the list to reflect the higher importance of the items that appear earlier in the list. Specifically, the weight of an item is half of the weight of its previous one. For the first several items that are keywords or keys of the entities involved in the query result, each is assigned a weight of 1. It is easy to see that such a weighting scheme satisfies the requirement: an item is more important to be selected than all the items after it in the list combined together, as an item must be included in the snippet before any of its successors is included.

Example 7.6 *The weight of each item in the snippet information list for the example is annotated below the items in Figure 7.5. Note that all keywords and entity names have the highest weight of 1, and that items in the same rectangle has the same weight.*

Entity Path Based Selection. For an instance of an informative item to be selected into the snippet, we need to include the path from the closest ancestor of this node that is in the current snippet to the node. We thus make the selections based on paths instead of nodes, which makes the selection procedure more efficient as fewer data units need to be considered for selection. One solution would consider each root-to-leaf path in the query result tree as a data unit for selection in determining which one to be included in the snippet. However, each path often only contains an instance of a single informative item, as most of the informative items are feature values, which are leaf nodes. Therefore each selection is still based on individual informative items without considering the possible interaction among them.

We consider entity-based paths as data units for selection. We use *leaf entity* to refer to the entities that do not have descendant entities. An *entity path* consists of a path from the closest ancestor of a leaf entity in the query result that is currently included in the snippet, to the leaf entity, along with all the attributes of the entities on the path. If a node instance of an informative item itself or its associated entity is on an entity path, then we say this informative item is *covered* by the path.

Example 7.7 *To concisely illustrate our algorithm, here we only present how to choose from the paths in the query result fragment shown in Figure 1.5, although there are many paths in the query result, based on which the $IList$ in Figure 7.3 is computed. There are five leaf entities in the figure, all of which are* clothes *entity. Therefore there are five entity paths, each of which is from* retailer *to a* clothes *node. We use p_1 - p_5 to denote the path from* retailer *to* clothes$_1$ - clothes$_5$ *respectively.*

Benefit-Cost of an Entity Path. To decide which path to select, we choose the one that has the maximal benefit-cost ratio. The cost of selecting a path p $p.cost$, is the number of words to be added into the snippet tree when selecting p. Initially, $p.cost$ is the number of words on p.

The benefit of selecting path p, $p.benefit$, is the summation of the weights of all the informative items covered by this path. $p.benefit$ is initialized during a depth first traversal of the query result, as presented in procedure init in Algorithm 3. For each node n in the query result, we use $n.ancCover$ to denote the set of the informative items covered if n is

Algorithm 6 Instance Selection of Snippet Information List

```
selectInstance ($QR, IList, sizeLimit$) {Select instances for items in IList}
 1: init($QR$)
 2: $snippet = \emptyset$
 3: $sizeLimitExceeded$ = false
 4: $currSize = 0$
 5: repeat
 6:    if next item in $IList$ is a box of items then
 7:       $V$ = all items in the box
 8:    else
 9:       $V$ = next item $\{v\}$ in $IList$
10:    for each item $v \in V$ do
11:       if $v$ is already covered then
12:          add the covered instance of $v$ to $snippet$ if it is not in $snippet$ yet
13:          $currSize$ += number of edges added to $snippet$
14:       else
15:          find the path $p$ with the highest ($benefit/cost$) that covers $v$
16:          add the shortest prefix $p'$ of $p$ to $snippet$, such that $p'$ covers $v$
17:          $currSize$ += number of words added to $snippet$
18:          if $currSize > sizeLimit$ then
19:             break
20:          if $p' = p$ then
21:             $p.benefit = 0$
22:          for each path $p''$ that share a prefix with $p'$ do
23:             $p''.cost$ -= length of common prefix of $p'$ and $p''$
24:          for each item $v'$ in $IList$ covered by $p'$ do
25:             put a mark on $v'$ to denote that $v'$ has been covered
26:             for each path $p''$ that covers $v'$ do
27:                $p''.benefit$ -= weight of $v'$ in $IList$
28:    if $currSize > sizeLimit$ then
29:       remove all nodes added to $snippet$ in this iteration
30:       $sizeLimitExceeded$ = true
31: until all items in $IList$ are selected in $snippet$ or $sizeLimitExceeded$ = true
32: return $snippet$
init ($QR$) {Compute benefits and costs of paths in a query result}
 1: $QRroot.ancCover$ = the informative items covered by the root of $QR$ if it is an entity
 2: for each entity $n$ in the depth-first traversal of $QR$ do
 3:    $n'$ = the nearest ancestor entity of $n$ in $QR$, and if it does not exist, $QRroot$
 4:    $n.ancCover$ = $n'.ancCover \cup$ the informative items covered by $n$
 5:    if $n$ is a leaf entity then
 6:       $p$ = the path from $QRroot$ to $n$
 7:       $p.benefit$ = the sum of weights of the items in $n.ancCover$
 8:       $p.cost$ = number of words on the path from $QRroot$ to $n$
```

selected. Note that if n is selected, then all its ancestors in path p will be included in the snippet, therefore $n.ancCover = n'.ancCover \cup V$, where n' is the parent of n, and V is the set of informative items that are covered by n's label and its attributes (if exists). We set $p.cover$ = $n.ancCover$, where n is the leaf entity of p.

Example 7.8 *Take the path p_1 from* retailer *to* clothes$_1$ *for example. p_1 covers informative items* Texas, apparel, retailer, Brooks Brothers, Houston, men, casual, *and* suit*, thus $p_1.benefit$ is the summation of their weights, 1 + 1 + 1 + 2^{-1} + 2^{-2} + 2^{-4} + 2^{-5} + 2^{-6} $\approx$ 3.86. Similarly, $benefit$ of the paths p_2, p_3, p_4 and p_5 are initialized to be 3.81, 3.91, 3.72, 3.51 respectively. The cost of all paths is 3 initially.*

Path Selections and Benefit-Cost Updates. The algorithm for selecting informative items is presented in procedure selectInstance in Algorithm 3. For an input query, each of its query results QR, an upper bound of the snippet size $sizeLimit$ and its information list $IList$, we generate a snippet. Initially, the snippet is empty. We process the informative items in $IList$ one by one in order, and select an entity path in the query result that can cover this item with maximal benefit-cost into a snippet, till all the items are covered in the snippet or the upper bound of snippet size is reached.

In each iteration, the informative items to be added to the snippet, denoted by V, is either one item or a box of items (line 6-8). Let v be the current informative item being considered at each step. If a node instance of v is already included in the snippet, nothing needs to be done. If its associated entity is included in the snippet, then it can be easily added into the snippet by adding the path from the associated entity itself to the snippet (line 10-11). For example, if we want to include an instance of item *outwear* in the snippet, and entity *clothes*$_3$ is already in the snippet, then, we simply add the path from *clothes*$_3$ to *outwear*$_3$ without choosing another entity path that covers *outwear*, and therefore has a minimal cost.

Otherwise, we need to choose a new entity path to cover the current informative item v. For all entity paths covering v, we choose the one p that has the best cost-benefit $p.benefit/p.cost$ to the snippet (line 14). Notice that for a chosen entity path, we only need to add its shortest prefix p' to cover the current informative item. If p' is a proper prefix of p, then p will not be removed from the entity path lists, but has its cost adjusted, as to be discussed soon; otherwise p can be disregarded.

Example 7.9 *The running example is continued here. We start with the first informative* *item in the list* Texas*. Since all five entity paths cover it, we choose the one with the highest* $benefit/cost$*, which is* p_3*. In fact, we only need to add a prefix of* p_3*, from entity* retailer *to entity* store$_1$*, together with their associated attributes into the snippet to cover* Texas*.*

After an entity path p' is added to the snippet, we need to update the information list, the cost and benefit of affected paths, according to the following.

First, for each item in $IList$ that is covered by p', we put a mark on it, denoting that it has been covered (line 23). If one of these items is encountered in future, it has a node instance that can be added to the snippet with the low cost (i.e., the number of nodes from this instance to its associated entity, or zero if it is already in the snippet), without the need of choosing another entity path.

Second, we update the costs of the affected entity paths. For each entity path p'' that has a common prefix with p', including p itself, its cost is decreased by the length of the common prefix. We use Dewey labeling to efficient compute the length.

The length of the common prefix of two entity paths can now be easily calculated as the length of the common prefix of the Dewey IDs of the leaf entities of p' and p'' (line 21). To efficiently identify these affected paths, we sort all entity paths by the Dewey ID of their leaf entities in a list. For the first node n in path p', we find the first and the last path in the entity path list whose leaf entity is a descendant of n, using a binary search. Each of the paths in the entity path list between them has a common prefix with p'.

At last, we need to update the benefits of the affected entity paths. For each entity path p'' that covers an item which is already covered by p', its benefit $p''.benefit$ is decreased by the weight of the corresponding item, for all the commonly covered items of p'' and p' (line 24-25).

After an instance of the current informative item v in $IList$ is included into the snippet, we need to check whether the snippet exceeds the size limit. If so, we must remove the nodes that were added into the snippet in this iteration (line 27), and set the flag that no more nodes need to be added into the snippet as its size limit is reached (line 28).

Example 7.10 *Continuing the running example, after selecting the highest benefit-cost entity path* p_3 *to cover item* Texas*, we include its prefix from* retailer *to* store$_1$ *to the snippet, and perform the following updates. First, we annotate in the* $IList$ *that the informative items* Apparel, retailer, Brooks Brothers *and* Houston *are covered. Second, we update the costs of affected path. Since paths* p_1*,* p_2 *and* p_3 *have a common prefix with the path included in the snippet, their costs are decreased by the length of this common prefix. Now the costs of the updated entity paths* p_1*,* p_2*, and* p_3 *are all 2.*

We also update the benefits of the affected paths. Since Texas, apparel, retailer, Brooks Brothers *and* Houston *are all considered to be covered by the first selected path, the benefits of the paths that cover these informative items need to be subtracted accordingly. Specifically, since* p_1*,* p_2 *and* p_3 *originally cover the above six items, each of their benefits is subtracted by the sum of these items' weights: 3.75.* p_4 *and* p_5 *cover the first five items in the above list, and each has its benefit subtracted by 3.5.*

Now, the next uncovered item in $IList$ *is* outwear*. We choose the path that covers it and has the highest benefit-cost, hence* p_3*, which is from* merchandises$_1$ *to* outwear$_3$*. Now* p_3 *is removed from the entity path list, as the entire path is included in the snippet. Now besides* outwear*, items* casual *and* women *in* $IList$ *are also marked as covered. The cost of* p_1 *and* p_2 *is now reduced to 1. For each path that covers* outwear*, i.e.,* p_2 *and* p_4*, its benefit is reduced by the weight of* outwear*:* 2^{-3}*. We also subtract the weight of* casual *from the benefits of* p_1 *and* p_4*, and subtract the weight of* women *from the benefit of* p_5*.*

We cover the items in $IList$ *one by one in this manner. Note that items in the same box, e.g.,* suit *and* outwear*,* women *and* men*, must be all included or not included in the snippets. Suppose the size limit of the snippet allows us to include all the items in the* $IList$ *in Figure 7.3 into the snippet, the final snippet is presented in Figure 7.2.*

To efficiently select the entity path to cover the current item in the snippet information list and to perform updates after a selection, we build a bitmap index for a query result during a traversal. The path dimension has all the paths sorted by the Dewey ID of their leaf entities. The value dimension has all the distinct informative items in the order of their appearance in the query result. Each entry $B(p, v)$ in the index records whether an item v is covered by p, and if so, which node on p covers it. For each path p, we also record its benefit $p.benefit$ and cost $p.cost$.

Now we analyze the complexity of the algorithm. Let QR be the query result, P the set of all entity paths in QR, d the document depth, and $|L|$ the size of $IList$. To include one item v in the information list $IList$ into the snippet, the algorithm searches for the path with the best benefit-cost that covers v (line 15 in selectInstance) by traversing all the entries $B(p, v)$, which entails a cost $O(|P|)$. After selecting the entity path p', we update the cost of all the paths that share a prefix with p'. Finding such a path using binary search on the path list has a cost $O(log|P|)$. Each of such paths has the cost updated according to Dewey label prefix computation, which is bounded by d. The complexity of performing cost updates is $O(|P|d)$, as in the worst case all paths need to be updated. We also update the benefits of the affected paths. For each item v' covered by p', we reduce the benefits of the paths that cover v' by traversing all entries $B(p'', v')$. The complexity of performing benefit updates is $O(|L||P|)$. The total number of iterations is bounded by $|L|$, and the cost of adding nodes into the snippet is $O(|P|d)$. Therefore, the total time complexity for instance selection is max$\{|L||P|d, |L|^2|P|\}$.

7.4 Experiment Evaluation

Experimental Setup

We have developed an effective and efficient snippet generation system for structured search results: eXtract. We tested three versions of the eXtract implementation (FD, FD-IRD, FD-IRD-Ratio). All the systems we have tested are listed as follows:

- **FD**: The version denoted as *FD* is the one reported in a conference paper of this work [65], which generates snippets using the features with the highest feature dominance.
- **FD-IRD**: We improved the algorithms, denoted as *FD-IRD* (Algorithms 1 and 3) which generates snippets based on the $IList$ containing prominent features whose scores are calculated using the improved formulae (7.2 and 7.3).

- **FD-IRD-Ratio**: We further improved snippet generation by additionally containing faithful distribution of dominant features, which is denoted as *FD-IRD-Ratio* (Algorithms 1, 2 and 3).
- **Optimal**: We implemented an algorithm which, given an IList and a snippet size limit, exhaustively search for the best instance for each item in the IList. Thus it outputs an optimal solution with respect to the IList and the size limit.
- **Xoom**: We implemented Xoom [116], which generates a readable query-independent summary for each XML document in a corpus.
- **Google Desktop**: Since there is no existing system for generating query-specific snippets for structured search results in the literature, we also compared our system with a popular text document search engine, Google Desktop. To focus the comparison on snippet generation instead of query result generation, we store each keyword query result as an XML file. Then we issue the test query using Google Desktop on the corresponding XML file to obtain its result snippet.

Metrics: We have evaluated all six approaches for the quality of the snippets generated. We also evaluated our approaches on processing time of snippet generation and scalability over the increase of query result size, as well as the upper bound of snippet size in terms of the number of words in a tree. Since we implemented Xoom ourselves, comparing the efficiency of this implementation and eXtract may not be helpful, and thus we do not report this in the experiment analysis. As snippets can be used to cluster the query results effectively and more efficiently, we also show the performance of grouping results using the snippets or ILists.

Environment: The experiments were performed on a 3.6 GHz Pentium 4 machine, running Windows XP, with 2GB memory and 160 GB hard disk.

Data and Query Set: We have tested four data sets. Film is an XML data set about movies, which has the combined information of 7 XML data available online (main.xml, actor.xml, people.xml, cast.xml, studios.xml, codes.xml, remakes.xml)[2], each of which con-

[2]http://infolab.stanford.edu/pub/movies/dtd.html

Film	
QF_1	films, Hitchcock, Paramount
QF_2	films, Hitchcock
QF_3	films, Disney
QF_4	Lifeboat, 1943
QF_5	Drama, films
QF_6	1922, GB, Famous
QF_7	Hitchcock, Paramount
QF_8	30m, films
QF_9	director, films
QF_{10}	Drama, film
Retailer	
QR_1	Store, formal
QR_2	Store
QR_3	retailer, Texas, men
QR_4	Store, Texas
QR_5	retailer, California, sportswear
QR_6	Store, Houston
QR_7	Store, Texas, men
QR_8	Retailer, apparel, Store, Philadelphia, formal
QR_9	store, shirt
QR_{10}	retailer, athletic, shoe
Auction	
QA_1	Africa, item
QA_2	closed, auctions, 1998
QA_3	item regions
QA_4	charges asia
QA_5	open, auctions
QA_6	closed auctions pays
QA_7	people
QA_8	Auctions asia
QA_9	closed auctions seller 1999
QA_{10}	site asia
Wikipedia	
QW_1	scientist, films
QW_2	programming language
QW_3	American independence
QW_4	Francis Hopkinson
QW_5	guitar
QW_6	Frederick Douglass
QW_7	skating sporting
QW_8	Franklin President
QW_9	FAO
QW_{10}	French, cuisine

Figure 7.7: Data and Query Sets for Testing Snippet Generation

tains information of specific entities related to movie (e.g., people, studios, etc.). Retailer is a synthetic data set that has similar schema and domains for node values as the one in Figure 1.5, while the value of a node is randomly selected from its corresponding domain. For each data set we have tested ten queries, as shown in Figure 7.7. Auction

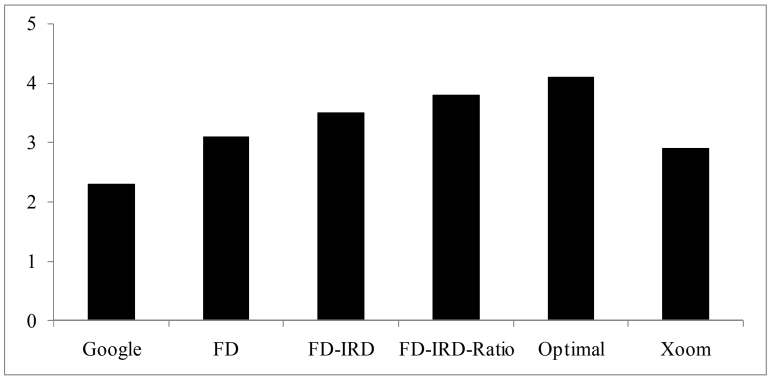


Figure 7.8: Average Scores of Google Desktop, FD, FD-IRD, FD-IRD-Ratio, Optimal Algorithm and Xoom over All Queries

is a synthetic data containing information about items, people and auctions generated by XMark (a publicly available XML benchmark).[3]. Wikipedia is a document-centric XML data set used in INEX 2009. For each query, a snippet size limit is randomly selected ranging from 6 to 23. The query results are generated using one of the existing keyword search approaches [144].

Snippet Quality

As there is no benchmark for evaluating the snippet quality for XML keyword search, we performed a user study. The quality test involves two parts: scoring snippets generated using six different approaches by users, and measuring the quality of snippets by precision, recall and F-measure based on the ground truth selected by users. A group of graduate students who were not involved in our project were invited to participate in the user study to assess the snippet quality.

Assessment of Snippets by User Scoring. For each query result of the forty queries in Figure 7.7, we use six approaches, FD, FD-IRD, FD-IRD-Ratio, Optimal algorithm, Google Desktop and Xoom, to generate snippets. Since a query result may be large, the users are given the statistical information of each query result (like the one shown in Figure 1.5) together with its snippet. Each user is asked to give a score for each snippet generated by each approach, respectively, on a scale of 1-5. The user do not know which

[3]http://monetdb.cwi.nl/xml/

snippet is generated by which approach.

The evaluation result is shown in Figure 7.8. The score for each algorithm shown in the figure is the average score of all the queries provided by all the users. As we can see, the scores of our approaches (FD, FD-IRD and FD-IRD-Ratio) are close to that of the Optimal approach, and are much better than that of Google Desktop. Xoom has a close score to FD, but is significantly outperformed by FD-IRD and FD-IRD-Ratio. For most queries, FD-IRD-Ratio either generates the same snippets as the Optimal approach, or misses one or two items in the snippet information list compared with the Optimal algorithm, thus their scores are close. The figure also suggests that incorporating IRD and ratio improves the quality of the snippets. Google Desktop is a search engine designed for text documents. It does not consider the tree structures of the XML documents when generating snippets, but simply concatenates the values in the XML document and outputs a fragment of it. Since Google Desktop has a low score for snippet generation on XML documents, we do not further compare with it in the experiments.

The Wikipedia data set verifies that the proposed approach generally works well on document-centric XML, compared with Google Desktop and Xoom which output sentences that summarize long text values in the snippets. The advantages of outputting statistically important words rather than sentences include: (1) The sentence that contains the keywords, which is output by Google Desktop, may not be the most informative sentence in the result. This commonly happens if the keywords do not appear in the first sentence of the document or a paragraph. Con- sider QW_2 "processing, input", which retrieves a document about stack-oriented programming language, the sentence that contains the keywords, "After processing all the input, we see the stack contains 56, which is the answer.", does not tell much information about this document. Xoom is query-independent and may not even output the sentence containing keywords. On the other hand, eXtract not only outputs the key of this result, "stack-oriented programming language", but also outputs statistically important words in this document such as "forth" (stack-oriented programming language is also known as the forth programming language), "programming", "code", which helps users understand the content of this document. (2) Due to the necessity to limit the snippet size,

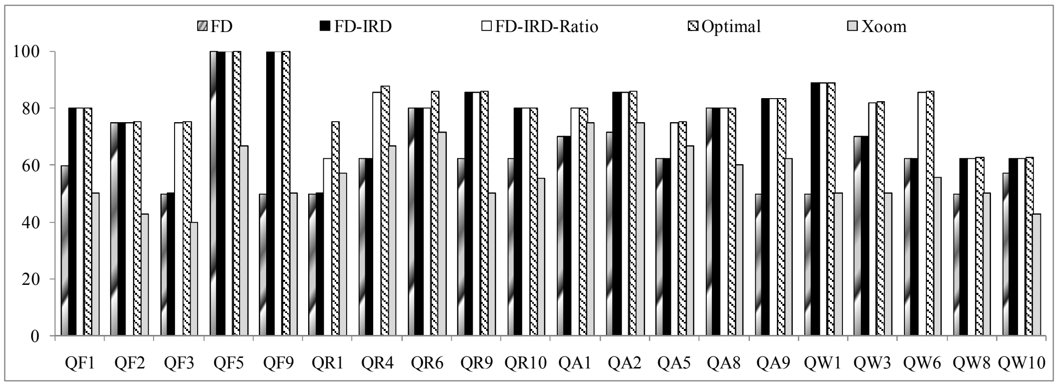


Figure 7.9: Precision Measurements for Snippet Generation

in many cases a snippet can not even afford to include 1 full sentence (as is the case for Google and Bing), which makes it hard to understand. For example, for QW_3 "American, independence", the sentence that summarizes the retrieved document, which is "The American Revolution is the political upheaval during the last half of the 18th century in which thirteen of Britain's colonies in North America at first rejected the governance of the Parliament of Great Britain, and later the British monarchy itself, to become the sovereign United States of America.", is too long to be included in the snippet in its entirety. For this query, eXtract outputs a succinct set of words that achieve similar effects with much less space, including "Douglass", "Garrison", "constitution", etc. (3) There may not even exist a single sentence in a document that can well summarize it, which is typically found in documents about stories, commentaries, product reviews, description of an object, etc. Consider QW_{10} "French cuisine" that retrieves a document about French cuisine, there is no single sentence in the document that can summarize so many aspects of French cuisine. In these cases, outputting important terms better serves the purpose of summarizing a document, such as "elegant", "wine", "seafood", "cheese", which are in the snippet output by eXtract.

On the other hand, if the retrieved document has a center sentence that well summarizes its content (such as QW_3), and if the snippet size limit is big enough to accommodate this sentence, then outputting this center sentence is generally more desirable than outputting words.

Assessment of Snippets by Comparing with Ground Truth. To make a deep

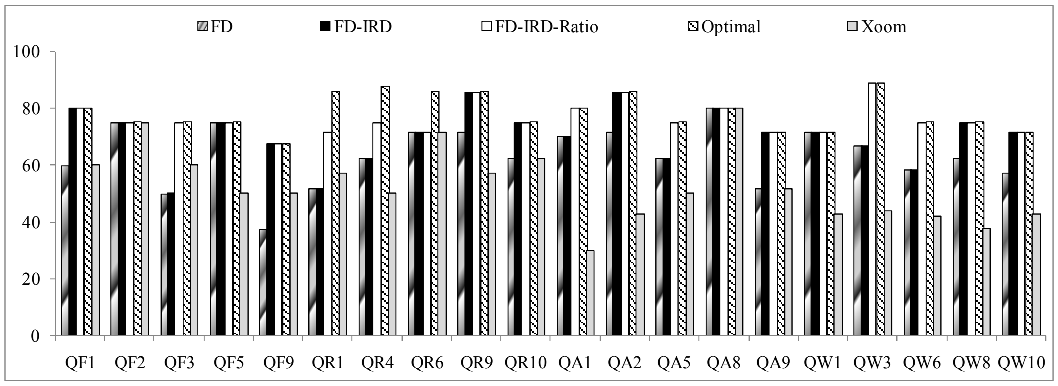


Figure 7.10: Recall Measurements for Snippet Generation

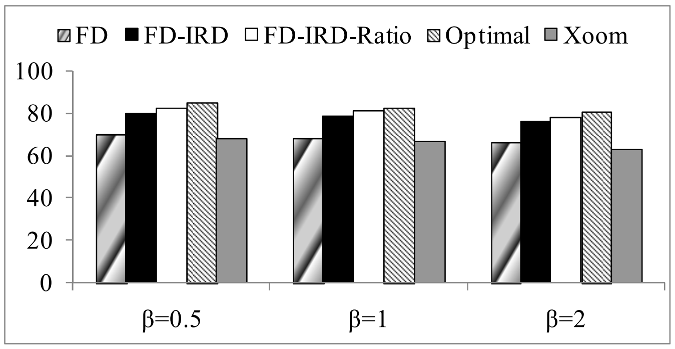


Figure 7.11: F-measure of Snippet Generation

analysis of our approach, we have conducted user surveys to define ground truth for the snippet of a given query result. We found that it is extremely difficult for users, who may not have experience or background in XML, to decide which nodes in the query result should be included in the desired snippet. Therefore, we asked the users to focus on the content, instead of the tree structure, in a query result. For each query result, each user is asked to provide a set of top k most important items in the query result, which they think should be included in the snippet. Since this part of the user study requires a lot of effort from the users, we randomly select twenty queries in Figure 7.7 to perform this study. After collecting the set of items provided by each user, in order to get the sorted snippet information list, we combine the top-k items from all the users together to form a universal set. Then we rank the items according to the numbers of their occurrences in the universal set, i.e., the number of users who think that the item should be selected in the snippet. Then according to user preferences, the list obtained in this way is adjusted as necessary to reflect the ratio among different features, which is considered as the ground truth of the snippet information list. Since the Optimal algorithm guarantees to find the optimal snippet with respect to a given snippet information list, we invoked the instance selection part of the Optimal algorithm on each ground truth information list, whose result is considered as the ground truth snippet for the corresponding query result.

Figures 7.9, 7.10 and 7.11 show the quality assessment of the snippets produced by each approach. Precision measures the percentage of the informative items output by an algorithm that are in the ground truth snippet. Recall measures the percentage of the informative items in the ground truth snippet that are output by an algorithm. F-measure is the weighted harmonic mean of precision and recall and can be computed as:

$$\frac{F = (1+\beta^2) * precision * recall}{\beta^2 * precision + recall}$$

As we can see, on some queries Xoom outperforms FD and FD-IRD. The advantage of Xoom is that, although Xoom does not output features with high FD and IRD, it may happen to output features that users are interested in. For example, for QR_1, Xoom outputs the brand of clothes, which do not have high FD and IRD but are interesting to the user, and thus the precision and recall of Xoom are both higher than FD and FD-IRD. The FD-IRD-Ratio approach still has the highest precision/recall for this query due to its keeping the ratios of features. The drawbacks of Xoom include: first the summary Xoom outputs do not include keywords and keys of the results, thus are not query-based and differentiable, e.g., for QR3 “retailer, Texas, men”, the keywords and the key attribute name of the retailer are preferred by users, but are not in the summary generated by Xoom. Second, Xoom ranks the tag names in each XML file in the order of their importance. For a set of tags, if the number of results containing these tags is the same, Xoom does not differentiate them. For example, for QF10 “drama, film”, since all films have attributes “title”, “year”, “producer”, “director”, “studio”, etc., these tags have the same rank and are selected randomly into the snippet.

FD, FD-IRD and FD-IRD-Ratio all have a good precision and recall, which confirms the intuition of our approach. This is because the snippet information list that our algorithm generates is similar as the one given by the user study, especially the items that appear earlier in the information list. Therefore, the snippets generated according to our information list by FD, FD-IRD, FD-IRD-Ratio and Optimal, have on average a high quality.

On the other hand, the precisions and recalls of FD, FD-IRD and FD-IRD-Ratio are not perfect, mainly because the information list that we generate are often not the same as the ground truth. The main reason of these differences is that the user may prefer interesting features of an entity, instead of/besides prominent features. Take QR_1 (*Store, formal*) for example. This query looks for the information of the stores selling formal clothes. To summarize a query result, our approach selects the most prominent features, such as "*fitting:men*" of the clothes. However, this feature is not necessarily interesting to the user, and the user actually chose "*brand:Adidas*" instead.

Furthermore, we compare the quality of snippets generated by the Optimal algorithm, FD, FD-IRD, FD-IRD-Ratio and Xoom. For test queries QF_2, QF_5 and QA_8, the precisions and recalls of FD, FD-IRD, FD-IRD-Ratio and the Optimal algorithm are the same because the query result returned for these queries are small, and don't contain much information.

For test queries QF_1, QF_9, QR_9, QR_{10}, QA_2, QA_8, QA_9, QW_1, QW_8, QW_{10}, FD-IRD, FD-IRD-Ratio and Optimal have better precision and recalls because they considered demoting features with low IRDs. In QF_1, for example, Hitchcock was well known by the suspicious films he directed. When people search for Hitchcock's film, they already expect the film to be of category suspicious, and expect other features in the snippet such as *title*. Therefore, demoting the feature *category:suspicious* in the query results satisfies the users' information needs. In QR_9, as a common knowledge, shirts are mostly made of cotton. When people search for shirts in a store, they will be more interested in other features of the shirts rather than the material of the shirts, which will mostly be cotton. Therefore, demoting the feature "*material:cotton*" saves the space to include other features which the users might be interested in such as "*situation:formal*".

For test queries QF_3, QR_1, QR_4, QA_1, QA_5, QW_3, QW_6, FD-IRD-Ratio and Optimal achieve better precision and recall compared with other approaches, because the two approaches considered keeping the ratio of prominent features. For example, in the results of QF_3, films produced by Disney are mostly cartoons and fantacy films and they focus more on cartoons. Therefore showing more cartoon than other films provides the user with more accurate information. For QR_4, Brooks Brothers's stores target more on men's clothes than women's. Therefore showing more men's clothes than women's in the snippet gives users the right impression.

For QR_6, Optimal has a higher precision and recall than all other approaches, because it may include additional features in the IList compared with other approaches.

Finally we compute the F-measure of each approach according to the average precision and recall across all the queries in user study, with parameter $\beta =$0.5, 1 and 2, as presented in Figure 7.11. FD-IRD-Ratio outperforms all other approaches except Optimal, which coincides with the user scoring in Figure 7.8.

Besides the techniques used to generate snippets given a fixed size limit, different sizes of the snippet can also influence the quality of snippets generated. To evaluate the effect of the size of snippet on these systems, we test the F-measures of FD-IRD-Ratio and Xoom with varying snippet sizes. We omit FD and FD-IRD in this test as they are clearly outperformed by FD-IRD-Ratio. We used three queries: QA_1, QW_7 and QW_{10}, which represent three different cases: QA_1 represents queries on data-centric XML; QW_7 represents queries on document-centric XML whose result has a center sentence that summarizes it; QW_{10} represents queries on document-centric XML whose result has no such center sentence. The result of this test is shown in Figure 7.12.

As can be seen from Figure 7.12, for QA_1, the features chosen by users include the payment, location, name and category of items, which largely coincides with the output of FD-IRD-Ratio. On the other hand, Xoom does not differentiate the tag names in the results, and output unimportant features such as the id, quantity, etc. For QW_{10} "French, cuisine", since no single sentence in the result can well summarize the document which contains various aspects of French cuisine, users prefer selecting statistically representative words as snippets, such as "elegant", "wine", "seafood", "cheese", even if the snippet size limit is 30, which largely coincides with what FD-IRD-Ratio outputs. Xoom outputs as snippet one or two sentences (depending on the size limit), which are "*French cuisine is a style of cooking derived from the nation of France. It evolved from centuries of social and political change.*". These sentences are not considered as informative by the users. An interesting case if QW_7 "skating, sporting". FD-IRD-Ratio has a better F-measure than Xoom initially, but is outperformed when the snippet size limit reaches 25. The reason is that when the size limit is this large, users prefer using a sentence that summarizes the result as the snippet, rather than individual words. For this query, such a sentence would be "*Figure skating is a Olympic sport in which individuals, pairs, or groups perform spins, jumps, footwork and other intricate and challenging moves on ice.*" which summarizes the definition of figure skating. Xoom outputs part of this sentence when snippet size is 15 or 20, while users prefer seeing words; but when snippet size is 25, users instead prefer this sentence as the snippet, thus the F-measure of Xoom is boosted, and the F-measure of FD-IRD-Ratio suffers a decline. When snippet size is 30, users choose this sentence, as well as a few important words in the document; however, Xoom starts to output a portion of another sentence, thus its F-measure decreases.

As we can see, FD-IRD-Ratio can generally cover the same set of informative items as optimal. FD-IRD and FD-IRD-Ratio achieves a better quality than FD on many queries by adopting IRD and considering ratios of features. Optimal has the best quality for some queries as the greedy algorithm is unable to select the optimal instances for items in the IList. Our proposed approaches, FD, FD-IRD and FD-IRD-Ratio, outperforms Xoom.

Processing Time

To evaluate the efficiency of our approach for generating result snippets, we test the processing times of the queries listed in Figure 7.7. The processing times, comprising the time of generating $IList$ and selecting instances, of the four approaches are shown in Figures 7.13 - 7.16. The sizes of the query results vary from 106KB to 360KB, and the snippet size limits vary from 10 to 20 words. As we can see, FD, FD-IRD and FD-IRD + Ratio are much faster than the Optimal algorithm. Besides, compared with FD, the additional processing time of FD-IRD and FD-IRD-Ratio is usually very small. To verify the applicability,

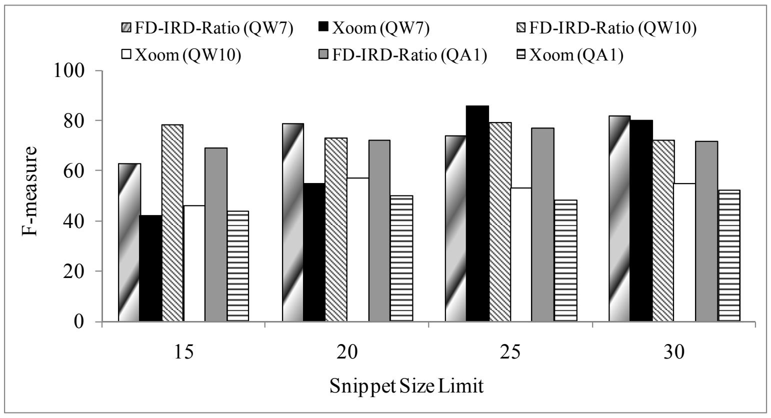


Figure 7.12: Average F-measure of Snippet Generation wrt Snippet Size Limit

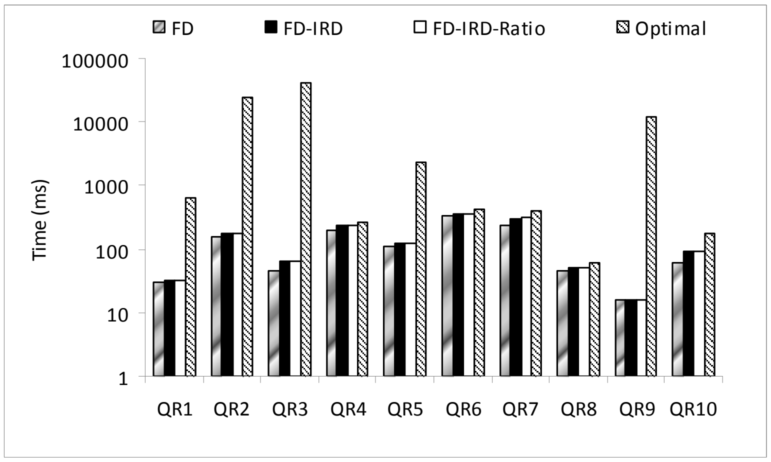


Figure 7.13: Processing Time of Snippet Generation on Retailer Data Set

we compute IRD on all results for each query.

All algorithms need to traverse the query result and construct an ordered snippet information list, in the same way. As we can see, the cost of keeping the ratio and calculating IRDs for features is generally small compared with the processing needed for instance selection, as FD-IRD-Ratio generally has a similar processing time as FD-IRD and FD. However, for some queries who have a large number (several hundred) of query results, such as QF_1 and QF_4, FD-IRD and FD-IRD-Ratio take a relatively longer time to calculate IRDs compared with FD, as it needs to traverse all results for this purpose. This is nonetheless not a problem in practice, as the user is usually interested in the top-k results and hence we can use the top ones rather than all results to computed the IRDs efficiently. The Optimal algorithm searches for the optimal solution by enumerating possible combinations

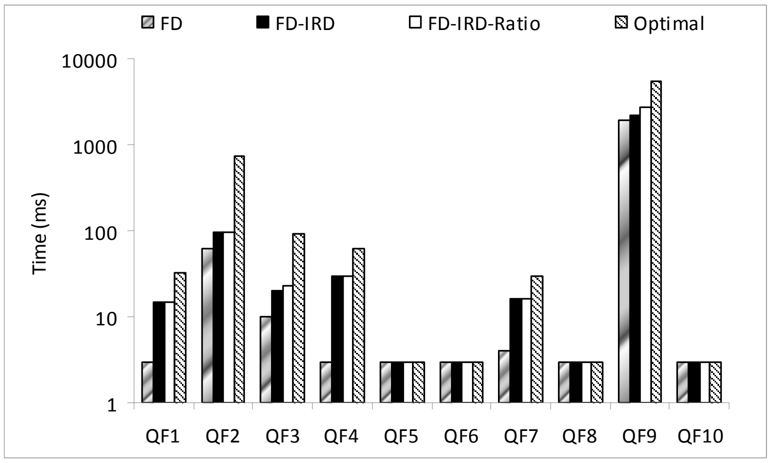


Figure 7.14: Processing Time of Snippet Generation on Film Data Set

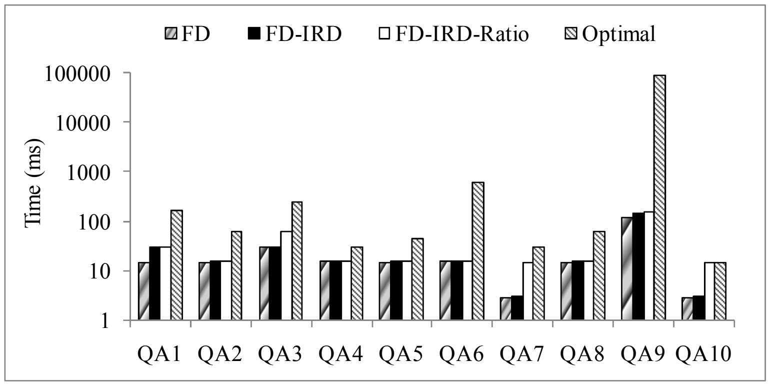


Figure 7.15: Processing Time of Snippet Generation on Auction Data Set

of instances to each item in the snippet information list, which leads to a cost exponential to the size of the information list. When the query result size or the snippet size limit is small (e.g. QF_5 whose query result size is 10KB and snippet size limit is 13, and QF_6 whose result size is 2KB and snippet size limit is 12, as well as QA_4, QA_{10}, QW_1, QW_6 and QW_3 etc, whose details are omitted here), the processing times of all algorithms are small. However, when the query result size is relatively large, indicating a potentially large number of instances of each informative item, the difference between the processing times of these approaches becomes significant (e.g. QR_3 whose result size is 23KB and snippet size limit is 10, and QF_9 whose result size is 102KB and snippet size is 11, as well as QA_6, QA_9, QW_{10} etc, whose details are omitted here).

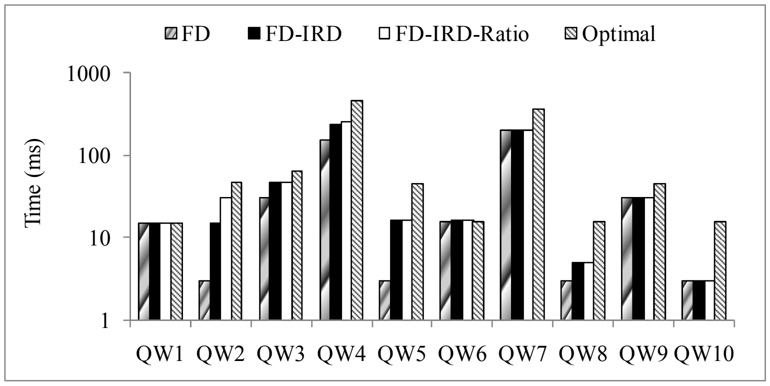


Figure 7.16: Processing Time of Snippet Generation on Wikipedia Data Set

Scalability

We test the scalability of our system on the Film data set over two parameters: *query result size* and *snippet size*.

Query Result Size. The scalability test with respect to query result size is shown in Figure 7.17 (the processing times for Optimal which are longer than 15s are not shown). A query result of QF_9 is replicated between 1 and 6 times to make the size of query result increasingly larger each time. The upper bound for the snippet size is fixed to be 11. We have tested the performances of the Optimal algorithm, FD, FD-IRD and FD-IRD-Ratio on these query results. As we can see, the processing time of the Optimal algorithm grows very rapidly as the complexity of the Optimal algorithm is high-order polynomial to the size of query result, the order of which is the size of the snippet information list. On the other hand, the processing times of the FD, FD-IRD and FD-IRD-Ratio grow slowly. For query result of 360KB, FD and FD-IRD, FD-IRD-Ratio only need 3.6, 4.7 and 5.0 seconds, respectively.

Snippet Size. In this test we evaluate the performance of the three algorithms with respect to the increase of snippet size upper bound, while keeping the query result size to be 106KB. Recall that when the snippet size increases, more items in the snippet information list can be included in the snippet, thus more nodes in the query result need to be processed in order to cover those items. The result in Figure 7.18 (the processing times for Optimal which are longer than 5s are not shown) shows that the processing times of FD,

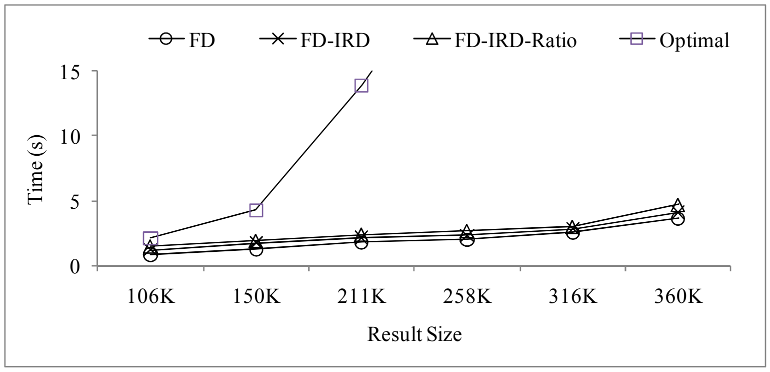


Figure 7.17: Scalability Test for Snippet Generation on Size of Query Results

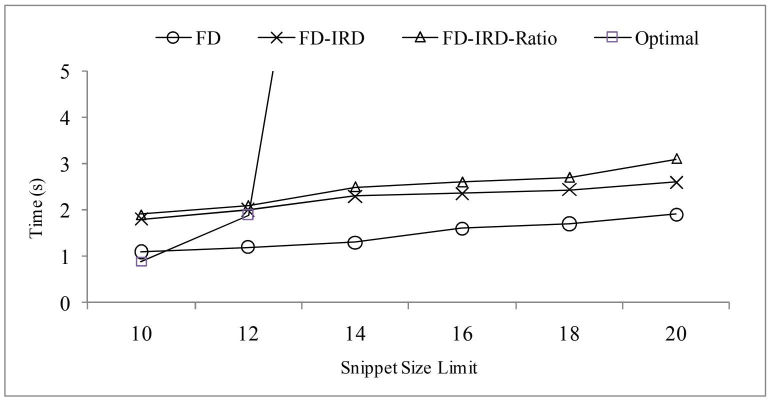


Figure 7.18: Scalability Test for Snippet Generation on Number of Words

FD-IRD and FD-IRD-Ratio increase much slower than that of the Optimal algorithm, where the later has a time complexity exponential to the number of informative items to be output.

In summary, this set of experimental evaluations shows that the snippets generated by our algorithm for XML search has high quality, as reflected by a high score in user evaluations, and high precision , recall and F-measure with respect to the user defined ground truth. The snippet generation is efficient for various queries, and scales well when the query result size and snippet size increase. The improved approach (FD-IRD-Ratio) achieves a better quality without much processing time overhead. Compared with the Optimal algorithm, our algorithm based on a greedy approach has a close quality in practice, and is much more efficient.

7.5 Summary

To the best of our knowledge, this is the first work that addresses the problem of generating result snippets for structured search results. We identify three goals of a good result snippet: *distinguishable*, *representative* and *small*. To meet the first two requirements in generating semantically meaningful snippets, we identify the most significant information in the query result that should be selected into a snippet as a snippet information list. To satisfy the third requirement, we need to generate the snippet that is maximally informative with respect to this list given an upper bound of the snippet size. However, its decision problem is proved to be NP-complete. Finally, we have designed and implemented a novel algorithm to efficiently generate informative yet small snippets. We verified the effectiveness and efficiency of our approach through experiments.

Chapter 8

SEARCH RESULT DIFFERENTIATION

8.1 Motivation and Goal

As we discussed in Chapter 1, many keyword queries are for information exploration purposes, where the user may not have a clear idea of what s/he wants, leading to user preference ambiguity. For this type of queries, even if the query results are highly relevant, the user still needs to check multiple results to make a decision (as opposed to navigational queries whose intent is to reach a particular website). Studies show that about 50% of keyword searches on the web are for information exploration purposes [25]. Information exploration queries are especially common in domains like online shopping, job hunting, etc. Without the help of tools that can automatically or semi-automatically analyze multiple results, a user has to manually read, comprehend, and analyze the results in informational queries. Such a process can be time consuming, labor-intensive, error prone or even infeasible due to possibly large result sizes. Although we have proposed techniques for generating result snippets in Chapter 7, as to be shown later, snippets are not designed to show the differences of multiple query results.

For example, consider a customer who is looking for *stores* that sell *clothes*s in *Houston* and issues a keyword query "*Houston, clothes, store*". There are many results returned, where the fragments of two results by searching on structured data are shown in Figure 1.5(a) and some statistics information of the results is shown next to the results. As each store sells hundreds of clothes, it is very difficult for users to *manually* check each result, compare and analyze these results to decide which stores to visit.

To help users analyze search results, the websites of many banks and online shopping companies, such as Citibank, Best Buy, etc., provide comparison tools for customers to compare specific products based on a set of pre-defined metrics, and have achieved big success. However, in these websites, only pre-defined types of objects (rather than arbitrary search results) can be compared, and the comparison metrics are pre-defined and static. Such hard coded approaches are inflexible, restrictive and not scalable.

As an example, Figure 1.5(b) shows the snippets of results in Figure 1.5(a) generated by eXtract [65, 95], given the upper bound of snippet size of 14 edges. These snippets highlight the *most dominant* features in the results. As we can see from the statistics information in Figure 1.5(a), the store in result 1 mainly sells *outwear* and *shirt casual* clothes, while the store in result 2 mainly sells *outwear* and *suit formal* clothes. However, snippets are generally *not comparable*. From their snippets, we know result 2 focuses on *formal* clothes, but have no idea whether or not result 1 focuses on *formal* or *casual*, since the information about the store, specifically *situation*, is missing in its snippet due to space limitation. Similarly, result 1 has many *shirt* clothes, but we do not have information about whether result 2 has many *shirt* clothes or not. As we can see, snippets are not designed to help users find out the differences among multiple results.

Although a general tool for informative query result comparison is very useful in diverse domains, it is not supported in existing text search engines. The main reason is that text documents are unstructured, making it extremely difficult if not impossible to develop a tool that automatically compares the semantics of two documents.

On the other hand, when searching structured data, the structural information of result may provide valuable meta data, and thus present a potential to enable result comparison. For example, directly generating a “comparison table" of an apple and an orange based on two general textual descriptions is difficult, but it becomes possible if the description is presented in structured format, with markups in XML or column names in relational databases to hint their *features* such as size, color, isFruit, and so on.

However, many challenges remain, even for enabling structured result comparison. For example, which features in the search results should be selected for result comparison? One desideratum is, of course, such features should maximally highlight the differences among the results. Then, how should we define the difference, and the *degree of differentiation* of a set of features? Another desideratum is, the selected features should reasonably reflect the corresponding results, so that the differences shown in the selected features reflect the differences in the corresponding results. Furthermore, how should we select desirable features from the results efficiently?

In this chapter we present the techniques for structured data search result comparison and differentiation, which takes as input a set of structured results, and outputs a Differentiation Feature Set (DFS) for each result to highlight their differences within a size bound. Features in the results are defined in the same way as Chapter 7. To show the usefulness of our technique in the real world, we develop a structured search result differentiation system named *XRed*, and use both real and synthetic data to evaluate our algorithms in experiments. The XRed system can take the results generated by any of the existing keyword search engines on structured data as the input and generate DFSs for result differentiation. In fact, the generated DFSs can also be used to compare results of structured query (e.g., XPath, XQuery, SQL) upon user request. Sample DFSs for the query results in Figure 1.5(a) are shown in Figure 1.5(c).

8.2 Desiderata of DFS and the DFS Generation Problem

In this section we first discuss three desiderata for *Differentiation Feature Set (DFS)*: limited size (Section 8.2), reasonable summary (Section 8.2), and maximal differentiation (Section 8.2). While maximal differentiation is the optimization goal in generating DFSs, limited size and reasonable summary are necessary conditions: the former ensures that the DFSs can be easily checked by a user, and the later ensures that the comparison based on DFS correctly reflects the comparison of results. Then we formalize the problem of generating optimal DFSs for a set of query results with a size bound and prove the NP-hardness of the problem.

Desiderata of Differentiation Feature Sets

Next, we discuss three desiderata for a set of DFSs.

Being Small

To enable users quickly differentiate query results, the first desideratum of a set of DFSs is: *being small*, so that users can quickly browse and understand them. The upper bound size of a DFS can be specified by the user.

Desideratum 8.1 (Small) *The size of each DFS D, denoted as $|D|$, is defined as the number of features in D. $|D|$ should not exceed a user-specified upper bound L, i.e., $|D| \leq L$.*

Summarizing Query Results

For the comparisons based on DFSs to be valid, a DFS should be a reasonable summary of the corresponding result by capturing the main characteristics in the result. Otherwise, the differences shown in two DFSs may not be meaningful.

Example 8.1 *Consider again the two results of query "*Houston, clothes, store*" in Figure 1.5(a). Both results mainly sell* outwear *clothes. Each store also sells some* suit *clothes. Suppose we have the DFS for result 1, D_1={store:category:suit}, and the DFS for result 2, D_2={store:category:outwear}. Obviously these two DFSs are different. However, these DFSs are not meaningful, since it gives the user a wrong impression: the difference of these two stores is that the first store mainly sells suit clothes, and the second store mainly sells outwear clothes. Obviously, this is not true. Intuitively, a feature that has more occurrences in the result should have a higher priority to be selected in the DFS, so that the DFS reflects the most important feature in the result, and the differences among DFSs correctly reflect the main differences of their corresponding results.*

Furthermore, although both stores in these results sells outwear *and* suit*, it is undesirable to simply output a single occurrence of* outwear *and* suit *in the DFS of each result. Such DFSs give users the impression that the two stores are similar in terms of their speciality on* outwear *and* suit*. However, the store in result 1 mainly focuses on* outwear *with just a couple of* suit*; whereas the store in result 2 focuses on both* outwear *and* suit*, with roughly the same number of clothes. Intuitively, the DFS should capture the distributions of features of the same type.*

As we can see from Example 8.1, a *valid* DFS should be a reasonable summary of the result, so that the important differences of the DFSs can be seen by the user. Thus we define the *validity* of a DFS as the following.

Desideratum 8.2 (Validity) *A DFS D is valid wrt a result R, denoted as $valid(D, R)$, if and only if the following rules are satisfied:*

(1) Dominance Ordered*: A feature can be included in D only if the features of the same type that have more occurrences in R are already included in D. That is, features of the same type should be ordered by dominance (defined as their number of occurrences).*

(2) Distribution Preserved*: A DFS should capture the distributions of features of the same type.*

To ensure a DFS satisfies *Dominance Ordered*, we sort the features of the same type in each result by their number of occurrences. Features of the same type with the same number of occurrences can be sorted in any way that is uniform for all results. We use alphabetical order in our approach. There are various ways to achieve *Distribution Preserved*. For each feature output in a DFS, we also show its percentage of occurrence within its feature type, such as the DFSs in Figure 1.5(c). Another way of achieving Distribution Preserved is to use font size to represent the percentage of features: features with higher percentage are shown in bigger fonts, which is analogous to Tag Cloud. Note that in this case, we may want to use a "weighted size" of each feature: a feature with higher percentage occupies more space of the DFS, since it has a bigger font.

Example 8.2 *In the result 1 in Figure 1.5(a), features of type* clothes:category*, in the descending order of their dominance, are* outwear*,* shirt*,* sweater *and* suit*. Then in order for its DFS to be valid, feature* sweater *can be included in the DFS only if both features preceding it,* outwear *and* shirt*, are already included.*

Differentiating Query Results

Being small and a good summary are necessary conditions for a DFS, yet they are insufficient.[1] In this section, we propose the unique and most challenging requirement for a good DFS: *differentiability*, i.e., a set of features that can differentiate one result from others.

[1]Indeed snippets are generally small and summarize results, nevertheless are ineffective for result comparison and differentiation, as discussed in Section 8.1.

Differentiability of DFSs. We define that two results are *comparable* by their DFSs if their DFSs have common features types. Two results are *differentiable* if the DFSs have different characteristics of those shared feature types.

Intuitively, features of different types are not comparable, e.g., we are not able to compare *clothes:category:outwear* in result 1 with *clothes:situation:formal* in result 2. Therefore, we consider each feature type as the differentiation unit. Each feature type can be considered as a vector, in which each component represents a feature of this type in the DFS, and the value of a component is the percentage of the feature. Typical ways of measuring the distance of two vectors include L_1 distance (i.e., Manhattan distance), Euclidean distance, cosine similarity, etc. Intuitively, the degree of difference of a feature type can be considered as the sum of percentage differences of all its values. Therefore, we use L_1 distance to define the degree of difference of a feature type. Other distance metrics can also be used.

There are also some approaches for measuring the distance of probability distributions, most of which can also be applied to compute the degree of difference of a feature type. However, note that some of them are not applicable. For example, some of these metrics are not symmetric, e.g. KL divergence [77]. The KL divergence of v_1 and v_2 and that of v_2 and v_1 are generally different, which is not suitable for our approach, since the difference of two feature types should intuitively be symmetric. Some other metrics, e.g., Earth Mover's Distance [117], are sensitive to the order of the components in the vector, which is also undesirable for the purpose of computing the difference of a feature type.

Note that there is an important issue when modeling a feature type in a DFS as a vector. Given two DFSs If a feature is included in both DFSs, its difference is simply the difference of percentage in the two DFSs. However, if it is not included in a DFS, the corresponding values in the vector should *not* always be 0. This is because a DFS only records partial information in the corresponding result, i.e., a feature that does not appear in a DFS may have occurrences in the results. If a feature is not included in any DFS, then the corresponding value in the vector should be considered as 0%, since the user cannot see this feature, and thus cannot see its difference in the results. But if a feature is included in some of the DFSs but not the others, we should determine how much difference the user can deduce from the DFSs about this feature. Let us look at an example.

Example 8.3 *For the two results in Figure 1.5(a), suppose feature type* clothes:category *in the two DFSs are:*

D_1*: outwear: 52%, shirt: 25%*

D_2*: outwear: 53%, suit: 47%*

Then for outwear*, the difference is 1%. For* shirt*, its percentage is 25% in result 1. In result 2, since* outwear *and* suit *already sums up to 100%, there is no* shirt *in result 2, thus the difference of* shirt *should be 25%. For* suit*, its percentage in result 2 is 47%. Since in result 1,* outwear *and* shirt *sums up to 77%, the percentage of* suit *in result 1 is at most 23%. Thus the percentage difference of* suit *in the two results is at least 47%-23% = 24%. For any other feature of this type, since it is not shown in either DFS, its difference is 0. Therefore, for this feature type, by* L_1 *distance, its degree of difference in the two results is 1% + 25% + 24% = 50%.*

As we can see, if we simply consider the percentage of suit in D_1 *as 0%, then we would conclude that the difference of suit is 47%, but the real difference may be only 24%. In other words, from these two DFSs, the user can only deduce a difference of 24% for suit.*

Consider another example, in which the two DFSs are:

D_1*: outwear: 40%, shirt: 30%*

D_2*: suit: 20%*

For outwear*, in result 2, its percentage is at most 20% (since we output the features in the order of their percentage). Thus the difference of* outwear *is at least 20%. Similarly, for* shirt*, its difference is at least 10%. On the other hand, for suit, the percentage in result 1 is at most 30%. Therefore, the difference of* suit *in the two DFSs is 0%, because there is a possibility that its percentage is 20% in result 1. Therefore, the total degree of difference of this feature type is 20% + 10% = 30%.*

As we can see, if we consider the percentage of suit in D_1 as 0%, we would get a difference of 20% for suit, which may not be true.

Example 8.3 gives us an idea of how to define the degree of difference of a feature type in two DFSs. Intuitively, the degree of difference of a feature type depends on how many features of this type we can differentiate, and how much difference of each feature is there in the two DFSs. For a feature F, if F appears in both DFSs, then we simply use its percentage difference in the two DFSs. If it appears in neither DFS, it cannot be differentiated. If it appears in one DFS but not the other, we should use the minimum difference of this feature type in the two DFSs that we can infer. For a feature type T, we sum up the differences of all features F of type T in the two results. If we consider feature type T as a vector in each DFS with its features as components, then the degree of difference of T is the L_1 distance of these two vectors. The formal definition is given below.

Definition 8.1 *Given a feature type T in two DFSs D_1 and D_2, the* degree of difference of T in D_1 and D_2, *denoted by $DoD_T(D_1, D_2)$, is computed as*

$$DoD_T(D_1, D_2) = \sum_{F \in T} diff(F)$$

where F is a feature of type T. $diff(F)$ is computed as:

- *If F is included in both D_1 and D_2, let p_1 and p_2 be the percentage of F in D_1 and D_2. The difference of F is $|p_1 - p_2|$.*
- *If F is included in D_1 but not D_2, let p_1 be the percentage of F_1 in D_1. We first compute p_2 as the maximum possible percentage of F in D_2. p_2 is the smaller of the following two numbers: (1) the percentage of the last feature of type T output in D_2; (2) 1 - the total percentage of all features of type T output in D_2. If $p_1 \geq p_2$, the difference of F is $p_1 - p_2$. Otherwise, the difference of F is 0.*
- *If F is included in D_2 but not D_1, its difference is measured in a similar way as above.*
- *If F is not included in either D_1 or D_2, its difference is 0.*

Example 8.4 *Consider the two DFSs in Figure 1.5(c). For feature type* store:name*, we have* $diff(Galleria) = 1$*,* $diff(Adorama) = 1$*, thus* $DoD_{store:name}(D_1, D_2) = 2$*. For feature type* clothes:category*,* $diff(outwear) = 0.01$*,* $diff(shirt) = 0.25$*,* $diff(sweater) = 0.13$*,* $diff(suit) = 0.37$*. Therefore,* $DoD_{clothes:category}(D_1, D_2) = 0.76$*. For feature type* clothes:category*,* $diff(casual) = 0.87$ *and* $diff(formal) = 0.87$*, thus* $DoD_{clothes:situation}(D_1, D_2) = 1.74$*.*

Note that for non-negative numbers $a_1, \cdots, a_n$ and $b_1, \cdots, b_n$, we have

$$\sum_{i=1}^{n} |a_i - b_i| \leq \sum_{i=1}^{n} a_i + b_i$$

Therefore, for any feature type T and two DFSs D_1 and D_2, $DoD_T(D_1, D_2)$ is always between 0 and 2.

Given the definition of the DoD of a feature type T in two results, we define the DoD of a feature type T in multiple DFSs as the sum of the DoD of T in every pair of DFSs, and also define the total DoD of multiple DFSs as the sum of the DoD of all feature types in those DFSs.

Definition 8.2 *Given a set of DFSs* $D_1, \cdots, D_n$*, the DoD of a feature type* T *in these DFSs is defined as*

$$DoD_T(D_1, \cdots, D_n) = \sum_{1 \leq i \leq n} \sum_{i < j \leq n} DoD_T(D_i, D_j)$$

the total DoD of these DFSs is defined as

$$DoD(D_1, \cdots, D_n) = \sum_{T} DoD_T(D_1, \cdots, D_n)$$

Example 8.5 *In the two DFSs in Figure 1.5(c), we have* $DoD_{store:name}(D_1, D_2) = 2$*,* $DoD_{clothes:category}(D_1, D_2) = 0.76$ *and* $DoD_{clothes:situation}(D_1, D_2) = 1.74$*. Thus* $DoD(D_1, D_2) = 3.50$*.*

we have the following desideratum 3 for differentiation feature sets:

Desideratum 8.3 (Differentiability) *Given a set of results* $R_1, R_2, \cdots, R_n$*, their DFSs,* $D_1, D_2, \cdots, D_n$*, should maximize the total degree of differentiation defined in Definition 8.2.*

We will show in the next subsection that, unfortunately, generating valid and small DFSs that maximize their DoD is NP-hard.

Problem Definition and NP-Hardness

In this section, we formally define the problem of generating DFSs for search result differentiation and analyze its complexity.

As we discussed in Sections 8.2 - 8.2, given a set of results, their DFSs should maximize the DoD, i.e., the total degree of differentiation, and the DFSs should be valid with respect to the corresponding result, and be small.

Definition 8.3 *The* DFS construction problem *(*$R_1, R_2, \cdots, R_n$*,* L*) is the following: given* n *search results* $R_1, R_2, \cdots, R_n$*, compute a DFS* D_i *for each result* R_i*, such that:*

- $DoD(D_1, D_2, \cdots, D_n)$ *is maximized.*
- $\forall i$*,* $valid(D_i, R_i)$ *holds.*
- $\forall i$*,* $|D_i| \leq L$*.*

Theorem 8.1 *The DFS construction problem is NP-hard.*

Proof. We prove the NP-completeness of the decision version of the DFS construction problem by reduction from X3C (exact 3-set cover). The decision version of the DFS *construction problem is: given n results $R_1, R_2, \cdots, R_n$, is it possible to generate a DFS D_i for each result R_i, such that $valid(D_i, R_i, p)$, $|D_i| \leq L$, and $DoD(D_1, D_2, \cdots, D_n) \geq S$?*

This problem is obviously in NP, as computing the DoD of a set of DFSs can be done in polynomial time. Next we prove the NP-completeness.

Recall that each instance of X3C consists of:

- *A finite set X with $|X| = 3q$;*
- *A collection C of 3-element subsets of X, i.e., $C = \{C_1, C_2, \cdots, C_l\}$, $|C| = l$, $C_i \subseteq X$ and $|C_i|$ = 3.*

The X3C problem is whether we can find an exact cover of X in C, i.e., a subcollection C^* of C, such that every element in X is contained in exactly one subset in C^*.

Now we transform an arbitrary instance of X3C to an instance of the DFS construction problem. We construct an instance of the DFS construction problem, in which there are $3q$ query results, and l different feature types. Each $C_i \in C$ corresponds to a feature type t_i, which has three different features: F_{i1}, F_{i2}, F_{i3}. For each $C_i = \{X_a, X_b, X_c\}$ in the X3C instance, let feature type t_i appear once in the X_a-th, X_b-th and X_c-th results, with feature F_{i1}, F_{i2} and F_{i3}, respectively. Note that in this way, all features have a percentage of 100% in each results. Let the DFS size limit L be 1, i.e., there can only be one feature in each DFS. The question is: can we find a DFS for each of the $3q$ results, such that $DoD(R_1, \cdots, R_{3q}) \geq 6q$?

If we can find an exact cover C^* for the X3C instance, then we select the corresponding q feature types. For each selected feature type, we add its 3 features to the corresponding 3 DFSs. In this way, each DFS has exactly one feature. Each feature type contributes 6 to the DoD, thus the total DoD is $6q$.

If we can find a set of DFSs such that their DoD is $6q$, then it is easy to see that we must find q feature types, and for each feature type, all its 3 features must appear in the corresponding DFSs. Otherwise, if a feature type has only 1 feature appearing in the DFSs, *then it does not contribute to the DoD; if it has 2 features appearing in the DFSs, it takes 2 slots but only contributes 2 to the DoD, making the total DoD impossible to reach $6q$.*

This means that there is an exact cover for the instance of X3C if and only if we can find a set of DFSs with a DoD of $6q$. Therefore, it is a reduction. Since this reduction obviously can be performed in polynomial time, the decision version of the DFS construction problem is NP-complete, and the DFS construction problem is NP-hard.

8.3 Local Optimality and Algorithms

Due to the NP-hardness of the DFS construction problem, in order to address the problem with good effectiveness and efficiency, we propose two local optimality criteria: single-swap optimality and multi-swap optimality. An algorithm that satisfies a local optimality criterion does not necessarily produce the best possible result, but always produces results that are good in a local sense. Next we show that single-swap optimality can be achieved efficiently in polynomial time. On the other hand, multi-swap optimality is more challenging to achieve, as a naive algorithm would be exponential. We present an efficient dynamic programming algorithm in that realizes multi-swap optimality.

Single-Swap Optimality

In this section we present the first local optimality criterion, *single-swap optimality*, for the DFS construction problem, and present a polynomial time algorithm achieving it.

Definition 8.4 *A set of DFSs is* single-swap optimal *for query results $R_1, R_2, \cdots, R_n$ if, by changing or adding one feature in a DFS D_i of R_i, $1 \leq i \leq n$, while keeping $valid(D_i, R_i)$ and $|D_i| \leq L$, their degree of differentiation, $DoD(D_1, D_2, \cdots, D_n)$, cannot increase.*

Let us look at an example.

Example 8.6 *The two DFSs in Figure 1.5(c) satisfy single-swap optimality, i.e., changing or adding any feature won't increase their DoD. For instance, if we change* clothes:category:suit, 47% *in D_2 to* clothes:fitting:men, 70%*, then the $DOD_{clothes:category}(D_1, D_2)$ reduces from*

Algorithm 7 Algorithm for Single-Swap Optimality

constructDFS (Query Results: $QR[n]$; Size Limit: L)

```
 1: for $i$ = 1 to $n$ do
 2:     arbitrarily generate $DFS[i]$ for $QR[i]$
 3: for $i$ = 1 to $n$ do
 4:     for each feature type $t$ in $DFS[i]$ do
 5:         $f$ = the next feature of type $t$ that is in $result[i]$ but not in $DFS[i]$
 6:         add $f$ into $DFS[i]$
 7:         $sizeinc$ = the size of feature $f$ {features may have different sizes, e.g., a feature with long text or large font may
            have a bigger size}
 8:         $DFS[i].size+ = sizeinc$
 9:         if $DFS[i].size > L$ then
10:             remove $f$ from $DFS[i]$
11:             $DFS[i].size- = sizeinc$
12:         else
13:             $benefit$ = computeBenefit($DFS, i, t, f$, null, null)
14:             if $benefit > 0$ then
15:                 goto line 3
16:             else
17:                 remove $f$ from $DFS[i]$
18:         for each feature type $t'$ in $result[i]$ do
19:             $f$ = the last feature of type $t$ in $DFS[i]$
20:             $f'$ = the next feature of type $t'$ that is in $result[i]$ but not in $DFS[i]$
21:             change the occurrences of $f$ to $occ_{f'}$ occurrences of $f'$ in $DFS[i]$
22:             if $DFS[i].size > L$ then
23:                 undo the change from $f$ to $f'$
24:             else
25:                 $benefit$ = computeBenefit($DFS, i, t, f, t', f'$)
26:                 if $benefit > 0$ then
27:                     goto line 3
28:                 else
29:                     undo the change from $f$ to $f'$
```

computeBenefit ($DFS[n]$; i; Feature Type: t; Feature Value: f; Feature Type: t'; Feature Value: f')

```
 1: {This function computes the delta DoD after adding $f'$ to $DFS[i]$ and removing $f$ from $DFS[i]$}
 2: $benefit = 0$
 3: for $j$ = 1 to $n$ do
 4:     if $j = i$ then
 5:         continue
 6:     $newDoD = DoD_{t'}(DFS[i], DFS[j])$ (Definition 8.1) {$newDoD$ is the current DoD of feature type $t'$}
 7:     remove $f'$ from $DFS[i]$
 8:     $oldDoD = DoD_{t'}(DFS[i], DFS[j])$ {$oldDoD$ is the DoD of feature type $t'$ before adding $f'$}
 9:     $benefit = benefit + newDoD - oldDoD$
10:     add $f'$ to $DFS[i]$
11:     $newDoD = DoD_t(DFS[i], DFS[j])$ {$newDoD$ is the current DoD of feature type $t$}
12:     add $f$ to $DFS[i]$
13:     $oldDoD = DoD_t(DFS[i], DFS[j])$ {$oldDoD$ is the DoD of feature type $t$ before adding $f$}
14:     $benefit = benefit + newDoD - oldDoD$
15:     remove $f$ from $DFS[i]$
16: return $benefit$
```

76% to 1%. On the other hand, D_1 and D_2 are still not differentiable on clothes:fitting*, since there is no feature of this type in D_1. Thus their DoD decreases by 75%.*

Single-swap optimality can be achieved by a polynomial-time algorithm: enumeration. The pseudo code of this algorithm is presented in Algorithm 7. There are four steps.

1. *Initialization.* We start with a randomly generated valid DFS for each result, satisfying the size limit (procedure constructDFS lines 1-2).
2. *Checking.* Performing an iteration of checking and updating DFSs (lines 3-28). For each DFS $DFS[i]$, we check whether the DoD of all DFSs can increase after adding a feature of type t to $DFS[i]$ (lines 4-15) or switching an existing feature of type t to a new feature of type t' that is currently not in DFS (lines 16-28).
3. *Updating and Iteration.* If such a DFS is found, then we make the update and restart the iteration in step 2 (lines 13 and 26).
4. *Termination.* If there is no DFS that can be changed to further improve the DoD, then we terminate and output the DFSs.

As we can see, in the *Initialization* step, DFSs are generated randomly. In fact, the initialization of DFS does not affect the local optimality of the proposed algorithms, but has impact on the generated DFSs and where a local optimal point is achieved. Investigation of good DFS initialization is an orthogonal problem.

Although the high-level description of the algorithm and the example look simple, there are three technical challenges to be addressed. First, when updating a feature in a DFS, we must ensure its validity with respect to the corresponding result and satisfaction of the size limit. Since each DFS must be valid, the addition of a feature to a DFS must be in the dominance order of this feature type, and the removal of features from a DFS must be in the reverse order of feature dominance. For single-swap optimality, we only check whether altering *one* feature can improve the DoD. Thus, to add a feature of type t to a DFS, only the most dominant feature of type t that is not in the DFS can be added; to remove a feature

# of clothes: *120*
situation: formal: *80%*; Others: *20%*
category: sweater: *60%*; skirt: *35%;* Others: *5%*
fitting: women: *70%*; children:*18%*; men: *12%*

(a) Statistics Information of Result 3

iteration	D_1	D_2	D_3
0	store: name: Galleria, 100% clothes: category: outwear, 52% clothes: fitting: men	store: name: West Village, 100% clothes: category: outwear, 53% clothes: category: suit, 47%	store: name: Biltmore, 100% clothes: category: sweater, 60% clothes: situation: formal, 80%
1	store: name: Galleria, 100% clothes: category: outwear, 52% clothes: fitting: men, 80% *clothes: situation: casual, 94%*	same as above	same as above
2	same as above	store: name: West Village, 100% clothes: category: outwear, 53% clothes: category: suit, 47% *clothes: situation: formal, 93%*	same as above
3	same as above	same as above	store: name: Biltmore, 100% clothes: category: sweater, 60% clothes: situation: formal, 80% *clothes: fitting: women, 70%*
4	same as above	store: name: West Village, 100% clothes: category: outwear, 53% clothes: category: suit, 47% clothes: situation: formal, 93% *clothes: fitting: men, 70%*	same as above

(b) Iterations Performed by Algorithm 1

Figure 8.1: Running Example of Algorithm 7

of type t', only the least dominant feature of t' that is in the DFS can be removed. Let us look at an example.

Example 8.7 *To explain the single-swap optimal algorithm, we use the two results in Figure 1.5(a), and another result whose statistics information is shown in Figure 8.1(a), as a running example. Suppose that the three DFSs are randomly initialized as in iteration 0 in Figure 8.1(b), and that the size limit for each DFS is 5. The algorithm updates one DFS for one of the three results in the 4 iterations as shown. In iteration 1, the algorithm attempts to add a feature of type* clothes:situation *to D_1. The feature to be added must be the most dominant one of this type:* casual*. The addition can increase $DoD_{clothes:fitting}(D_1, D_3)$, and thus increase the DoD of the three DFSs..*

The second challenge is that, due to the interactions among DFSs, one DFS may need to be updated multiple times, where the number of updates cannot be determined before the termination of the algorithm.

Example 8.8 *Continuing Example 8.7, in iteration 2, Algorithm 7 tries to add (*formal, 93%*, the most dominant feature of type* clothes:situation*, to D_2. This increases the total DoD. Note that at this time, adding (*clothes: fitting: men, 70%*) to D_2 does not increase the DoD, as D_1 has exactly the same (feature, precentage) pair, and D_3 does not have feature type* clothes:fitting*. However, after adding (*clothes: fitting: women, 70%*) to D_3 in iteration 3, it becomes valuable to add (*clothes: fitting: men, 70%*) to D_2 in iteration 4, which will increase the total DoD. As we can see, after D_2 was first checked and updated in iteration 2, it needs to be updated again to further improve the DoD after other DFSs are updated.*

The iteration continues till no DFSs can be added or changed to improve the DoD. Since the number of times that we may update a DFS is unknown, one question is whether the algorithm terminates and how many iterations will be performed. As will be analyzed shortly, this enumeration algorithm is guaranteed to run in polynomial time in terms of the number of results (n) and the number of features (m).

The third challenge is that we need to compute the delta of DoD upon an altered or added feature (Procedure computeBenefit). Note that adding a feature to a DFS may not increase the DoD, and removing a feature from a DFS may not decrease the DoD. After each iteration of Algorithm 7, we compute the DoD of the altered feature type according to Definition 8.1, then update the total DoD of all DFSs.

Now we analyze the complexity of the Algorithm 7. Let n be the number of query results, and m be the number of feature types in a result.

- In each iteration, we check at most n DFSs. For each DFS $DFS[i]$ (in a single iteration), we check at most m^2 feature pairs to see whether an existing feature should be replaced, and check at most m features to see whether a feature should be added. As discussed earlier, for each feature type, we have to check the features with respect to their dominance order, thus there are only m choices of feature swap or addition for one result in one iteration. Each check will compute the delta of DoD by invoking Procedure computeBenefit. Procedure computeBenefit computes the DoD of feature type t, we sort the features of type t in both DFSs, then scan them and compute the DoD of type t according to Definition 8.1. Therefore, it takes $O(L\text{log}L)$ time, where L is the DFS size limit. Thus, computeBenefit takes $O(nL\text{log}L)$, and each iteration takes at most $O(n^2m^2L\text{log}L)$ time.

- In each iteration except the last one, the DoD of the DFSs at least increases by 1. The maximum possible DoD for each feature type in two results is 2, thus the maximum possible DoD for two results is $2m$, and the maximum DoD of all n DFSs is bounded by $O(n^2m)$. This means we need at most $O(n^2m)$ iterations, and thus the algorithm runs in polynomial time in terms of n and m.

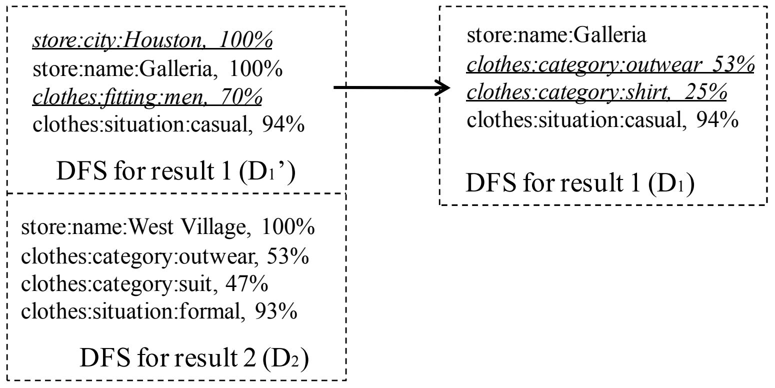
Multi-Swap Optimality

After discussing *single-swap optimality*, we propose *multi-swap optimality*, a stronger criterion. Then we present an efficient dynamic programming algorithm to achieve it.

Recall that single-swap optimality guarantees that the DoD of a set of DFSs won't increase by changing one feature in a DFS. On the contrary, multi-swap optimality requires that the DoD cannot increase by changing *any number of* features in a DFS, as formally defined below.

Definition 8.5 *A set of DFSs is* multi-swap optimal *for query results $R_1, R_2, \cdots, R_n$ if, by making any changes to a DFS D_i of R_i, $1 \leq i \leq n$, while keeping $valid(D_i, R_i)$ and $|D_i| \leq L$, $DoD(D_1, D_2, \cdots, D_n)$ cannot increase.*

Example 8.9 *Figure 8.2 is an example of DFSs achieving single-swap optimality but not multi-swap optimality. D'_1 and D_2 are DFSs of the two results in Figure 1.5(a) (suppose the percentage of feature* clothes:category:outwear *is 53% in Result 1). As we can see, $DoD(D_1, D_2)$ cannot be improved by changing or adding a single feature in either*



store:city:Houston, 100%
store:name:Galleria, 100%
clothes:fitting:men, 70%
clothes:situation:casual, 94%
DFS for result 1 (D1')
store:name:West Village, 100%
clothes:category:outwear, 53%
clothes:category:suit, 47%
clothes:situation:formal, 93%
DFS for result 2 (D2)
store:name:Galleria
clothes:category:outwear 53%
clothes:category:shirt, 25%
clothes:situation:casual, 94%
DFS for result 1 (D1)

Figure 8.2: Single-Swap Optimality and Multi-Swap Optimality

*DFS. However, if we change (*store:city:Houston, 100%*) and (*clothes:fitting:men*, 70%) into (*clothes:category:outwear*, 53%) and (*clothes:category:shirt*, 25%), then feature type* clothes:category *now have a non-zero DoD in the two DFSs, and thus $DoD(D_1, D_2)$ increases.*

In fact, achieving multi-swap optimality is more challenging than achieving single-swap optimality. Consider an enumeration based algorithm, adapted from Algorithm 7. While keeping the *Initialization*, *Updating and Iteration* and *Temination* steps the same, the *Checking* step is different. Instead of checking whether adding a *single* feature or swapping a *single* feature in a DFS can improve the DoD, we now need to check every possible combination of features in a DFS. Since the number of features in a query result is bounded by the result size n, there can be up to 2^n different combinations of features in its corresponding DFS, leading to an exponential time complexity.

In fact, Theorem 8.2 shows that achieving multi-swap optimality is NP-hard.

Theorem 8.2 *Given a set of query results, the problem of constructing a DFS for each result such that the DFSs are multi-swap optimal, is NP-hard.*

Proof. In order to prove that multi-swap optimal is NP-hard to achieve, we can instead prove that it is NP-hard to verify, which indicate the NP-hardness of achieving multi-swap optimality. If we can efficiently answer the question of whether a set of DFSs is multi-swap optimal, then we can efficiently answer the following question: whether we can *update a specified DFS to increase their DoD. Therefore, we only need to prove that the following problem is weakly NP-hard: let there be n results. The DFS of the first $n-1$ results are fixed. How to generate the last DFS such that their DoD is maximized?*

This problem is obviously in NP. We prove its NP-hardness by reduction from 0-1 knapsack. In an instance of 0-1 knapsack, there are m items, item i has value p_i and weight w_i. The knapsack capacity is W. 0-1 knapsack is the problem of maximizing the value of the selected items, such that their total weight does not exceed W.

Now we reduce it to an instance of the problem above. Let the last result have m feature types, one corresponding to one item in knapsack problem. For feature type T_i, let the first feature f_{i1} and the second feature f_{i2} have a ratio of $(w_i - 1) : 1$. Let this feature type appear in another p_i results, in which the first feature in order of their importance is f_{i1}, but the second feature is not f_{i2}. Then, in the DFS of the last result, if we output w_i features of type T_i, we increase the DoD by p_i. Outputting less than w_i features won't increase the DoD, and outputting more than w_i features won't further increase the DoD beyond p_i. Let the DFS size limit be W.

It is easy to see that if we select a subset of items in the 0-1 knapsack instance to achieve maximum total value, then outputting the corresponding subset of features in the instance of the multi-swap optimality problem also gives the maximum DoD. The reverse is also true. This reduction can obviously be performed in polynomial time. Therefore, checking multi-swap optimality is NP-hard, and achieving multi-swap optimality is also NP-hard.

To efficiently achieve multi-swap optimality, we have designed a dynamic programming based algorithm that runs in polynomial time with respect to n (the number of query results) and m (the maximum number of features in a result). We address the technical challenges in Step 2 *Checking*: verifying whether there exists *any change* to a DFS, referred to as "target DFS", that can improve the total DoD. Instead of enumerating changes to a DFS (as the number of possible changes are exponential), our algorithm directly generates a valid *multi-swap optimal* target DFS, given the others DFSs.

To generate such a target DFS, we first need to determine for each feature type, what are the choices of selecting features to compose a valid DFS. Intuitively, for each feature type, there are multiple choices of including its features in the DFS, each with a different number of features included. To measure the effect of each choice, we define *benefit* and *cost* of a feature type. Specifically, if we include x features into the target DFS, then the cost is x, and the increase of DoD obtained by adding these x features is considered as benefit y.

Algorithm 8 Algorithm for Multi-Swap Optimality

```
constructDFS (Query Results: $QR[n]$; Size Limit: $L$)
 1: for $i$ = 1 to $n$ do
 2:     arbitrarily generate $DFS[i]$ for $QR[i]$
 3:     $DoD[i]$ = 0
 4: for $i$ = 1 to $n$ do
 5:     for $j$ = 1 to $n$ do
 6:         for each feature common feature type $t$ in $DFS[i]$ and $DFS[j]$ do
 7:             $DoD[i] += DoD_t(DFS[i], DFS[j])$ (Definition 8.1)
 8: for $i$ = 1 to $n$ do
 9:     $DoD'$, $newDFS$ = checkDFS($QR[i], DFS, i, L$)
10:     if $DoD' > DoD[i]$ then
11:         $DFS[i] = newDFS$
12:         $DoD[i] = DoD'$
13:         goto line 9
checkDFS (Query Result: $QR$; DFSs: $DFS[n]$; $i$; Size Limit: $L$)
 1: {This function constructs the optimal $DFS[i]$ given the other DFSs}
 2: $t$ = number of feature types in $QR$
 3: for $l$ = 1 to $L$ do
 4:     compute $s_{1,l}$ according to Figure 8.3
 5:     Suppose $s_{1,l}$ is maximized by outputting $x$ features of type 1
 6:     $best_{1,l} = x$
 7: for $k$ = 2 to $t$ do
 8:     for $l$ = 1 to $L$ do
 9:         compute $s_{k,l}$ according to Figure 8.3
10:         Suppose $s_{k,l}$ is maximized by outputting $x$ features of type $k$
11:         $best_{k,l} = x$
12: $k = t$
13: $l = L$
14: $newDFS = \emptyset$
15: while $k > 0$ and $l > 0$ do
16:     output $x$ features of type $k$ in $newDFS$
17:     $k--$
18:     $l -= x$
19: $DoD' = 0$
20: for $j$ = 1 to $n$ do
21:     $DoD' +=$ the degree of differentiation between $DFS[i]$ and $DFS[j]$
22: return $DoD'$, $newDFS$
```

Example 8.10 *We use the query results in Figure 1.5(a) to explain the benefits and costs of a feature type. For feature type* clothes:category*, we have*
D_2 = *{*outwear, 53%*,* suit, 47%*},*
Consider D_1 *as the target DFS. According to Figure 1.5(a), the list of features of this type in the order of their dominance in result 1 is {*outwear*,* shirt*,* sweater*,* suit*}.*

(1) If we have D_1 = *{*outwear, 52%*}, then cost=1, benefit=1%. This is because the percentage of* outwear *in* D_1 *and* D_2 *are 52% and 53%, respectively. Note that the difference of suit is 0%, since the maximum possible percentage of* suit *in* D_1 *is 48%.*

(2) If we have D_1 = *{*outwear, 52%*,* shirt, 25%*}, then cost=2, and benefit=50%, as illustrated in Example 8.3.*

(3) If we have D_1 = *{*outwear, 52%*,* shirt, 25%*,* sweater 13%*}, then cost=3, and benefit=76%, as now* $diff(sweater) = 13\%$ *(increased by 13% compared with cost=2) and* $diff(suit) = 37\%$ *(increased by 13% compared with cost=2).*

As we can see, for each feature type, there is a list of choices of how many features can be selected in a DFS, each with a benefit and a cost. We denote the above three choices as (1, 1%), (2, 50%) and (3, 76%), respectively.

Given the choices of generating valid DFSs discussed above, our goal is to calculate the optimal valid target DFS that can maximize the DoD, given the DFSs of the other results. We use $s_{m,L}$ to denote the maximum DoD that can be achieved by a valid optimal target DFS, where m is the total number of feature types in the result and L is the DFS size limit.

$s_{m,L}$ can be computed using dynamic programming. We give an arbitrary order to the feature types in the query result of target DFS. Let $s_{k,l}$ denote the maximum DoD that can be achieved by considering the first k feature types in the result, with DFS size limit l. Each $s_{k,l}$ is calculated using the recurrence relation discussed in the following.

- If k = 1, $s_{k,l}$ = the maximal benefit of the first feature type that can be achieved with cost not exceeding l.

$$s_{k,l} = \begin{cases} \max\{b_{ki} \mid c_{ki} \le l\} & k = 1 \\ \max\{s_{k-1,l}, \max\{s_{k-1,l-c_{ki}} + b_{ki} \mid c_{ki} \le l\}\} & k > 1 \end{cases}$$

Figure 8.3: Recurrence Relation

- If $k > 1$, then we have multiple choices. We can choose not to include any feature of the k-th feature type at all, thus $s_{k,l} = s_{k-1,l}$. Otherwise, for the k-th feature type, suppose the list of feature selections that comprise a valid and small DFS is denoted as a list of benefit and cost pairs: (b_1, c_1), (b_2, c_2), and so on. We can choose any item in this list. For instance, if we choose to output c_1 features, then we can increase the benefit with b_1, but to accommodate the cost c_1, the first $k - 1$ feature types can only include $l - c_1$ features, i.e., $s_{k,l} = s_{k-1,l-c_1} + b_1$.

Therefore, the recurrence relation for calculating $s_{k,l}$ is shown in Figure 8.3, where we assume that the k-th feature type has p_k different benefit and cost pairs, (b_{k1}, c_{k1}), (b_{k2}, c_{k2}), $\cdots$ (b_{kp_k}, c_{kp_k}), and $1 \le i \le p_k$.

The dynamic programming procedure that computes the optimal valid $DFS[i]$ is given in Algorithm 8 procedure checkDFS. We first compute $s_{1,l}$ for each l (lines 2-5), then compute $s_{k,l}$ as discussed. Meanwhile, we record array $best$, which is used to reproduce the optimal DFS, $newDFS$ (lines 11-17). Finally, DoD' is calculated by comparing $newDFS$ with every other DFS (lines 18-21).

The entire algorithm for multi-swap optimality is presented in Algorithm 8. Similar as Algorithm 7, it begins with randomly generating a DFS for each result (Procedure constructDFS lines 1-3). Then it computes $DoD[i]$, the total DoD between $DFS[i]$ and other DFSs (lines 4-8). In each iteration (lines 9-14), instead of tentatively making changes to each DFS as what Algorithm 7 does, this algorithm directly generates a valid multi-swap optimal $newDFS$ given the other DFSs, whose DOD is DoD', by invoking Procedure checkDFS. If DoD' is bigger than $DoD[i]$, then $DFS[i]$ is replaced by $newDFS$ with $DoD[i]$ updated (lines 11-14). Similar as Algorithm 7, Algorithm 8 terminates when no DFS can be changed to further improve the DoD.

Now we analyze the complexity of Algorithm 8. Let n, m, m', L denote the number of results, number of feature types, number of features and DFS size limit, respectively. In procedure checkDFS, we first compute $newDFS$ using the equation in Figure 8.3 (lines 2-17), with complexity $O(m'L)$. Lines 18-20 of checkDFS compute the DoD of two DFSs. Since determining whether two DFSs can be differentiated on a given feature type takes $O(L\text{log}L)$ time, the complexity of $newDFS$ is $O(m'L+mL\text{log}L)$. In constructDFS, we first compute the DoD of every two results in $O(nmL\text{log}L)$ (lines 4-8). Similar as Algorithm 7, the iteration in lines 9-14 is executed at most $O(n^2m)$ times. Therefore, the total complexity of Algorithm 8 is $O(n^2mL\text{log}L(mn+m'))$.

As to be shown in Section 8.5, the algorithm is in fact quite efficient in practice, as the number of iterations is generally far less than n^2m.

8.4 Feature Type Oriented DFS Construction

We have shown how to achieve two local optimality criteria in Section 8.3. Both of them consider one result at a time: single-swap optimal tries to change one feature in one result, and multi-swap optimal tries to change multiple features in one result. These two algorithms have the following disadvantage: *the quality of the algorithms depends on the initialization.* Specifically, if a good feature type does not have enough occurrences in the initialization, then it will not be chosen during the DFS generation.

Example 8.11 *Consider the two results shown in Figure 1.5(a). Consider an initialization of the two DFSs as shown in Figure 8.4. In this case, no matter how we change a single DFS, we cannot increase the DoD. Specifically, for feature type* store:city*, both results have the same feature which is* Houston*. For* clothes:category*, if we only change one or more features in a single DFS, the DoD of this feature type will remain at 1%, and will not increase. Only when the DFSs of both results have two features included, they can have a larger DoD. It is the same for* clothes:fitting*. Besides, some feature types like* store:name *is not in the initial DFSs, and adding it to the DFS of one result does not increase the DoD either. Therefore both Algorithms 7 and 8, which only change one DFS at one time if the change increases the DoD, will not change the initial setting of both DFSs, resulting in low DoD of 1%.*

D1	D2
store:city:Houston, 100% clothes:category:Canon, 52% clothes:fitting:men, 70%	store:city:Houston, 100% clothes:category:Canon, 53% clothes:fitting:men, 70%

Figure 8.4: A Possible Initialization of DFSs for the Results in Figure 1.5(a)

Note that in contrast to single-swap and multi-swap optimality, another possible local optimality criteria is: the DoD of all DFSs cannot increase by changing any one feature in multiple DFSs. The problem above can be largely solved by achieving this local optimality criteria. Unfortunately, using the same proof as the one for Theorem 8.1, it is easy to see that achieving this local optimality criterion is NP-hard.

In observance of the above problem, in this section, we propose heuristics algorithms which, although not necessarily achieving a local optimality criteria, are superior to Algorithms 7 and 8 in that they consider multiple results together when generating DFSs. As we can see from the example above, Algorithms 7 and 8 may miss a good feature type if it does not have enough occurrences in all DFSs in the initialization. Thus our new algorithm considers a feature type at each time, rather than a result. We refer to the algorithms based on this idea as *feature type oriented DFS construction algorithm*.

The intuition of the feature type oriented algorithms is that we can compute how "good" a feature type is, then select the feature types according to certain criteria. However, one barrier of this idea is: under the current problem setting, feature types cannot be completely considered as independent, which makes the problem much harder. For example, consider feature types t_1 and t_2. Let us assume that t_1 and t_2 can significantly differentiate many results, thus they both have a "high" quality. However, suppose that to significantly differentiate many results, both t_1 and t_2 require a large presence in a DFS D_i. Since D_i has a size limit, it may not be able to accommodate many occurrences of both t_1 and t_2, which means using t_1 and t_2 together may not be a good idea.

With this observation, to handle the interaction between feature types, we first attempt to solve an alternative problem, which is the same as the problem defined in Definition 8.3, except that there is a single size limit for *all* DFSs, rather than a size limit for each

DFS. If the size limit of the individual DFS is L and there are n results, then we consider the size limit for all DFSs as $n \times L$. For this problem, we can measure the quality of each feature type independently of other feature types, and select the feature types accordingly. After we get a solution to this problem, since an individual DFS do not have a size limit, there may be some DFSs whose sizes exceeds L and some other DFSs whose sizes are smaller than L. If this happens, then we iteratively remove some features from each DFS whose size exceeds L, and greedily add some features for each DFS whose size is smaller than L, which will be discussed later.

Apparently, the quality of a feature type depends on how many occurrences it is allowed to have. Recall that in Algorithm 8, for each feature type in the result being processed, we compute a set of (benefit, cost) pairs, then use dynamic programming to find the optimal number of occurrences of each feature type. The same idea can be adopted here. We can compute a set of (benefit, cost) pairs for each feature type with respect to *all* results. In other words, each (benefit, cost) pair denotes the DoD contributed by the feature type (benefit) if we allow it to have a certain number of occurrences (cost) in *all* results. After we compute the (benefit, cost) pairs for all feature types, since we consider a single size limit for all DFSs, we can use the same recurrence relation as in Figure 8.3 to compute the optimal number of occurrences of each feature type.

However, computing (benefit, cost) pairs for a feature type with respect to all results is much harder than doing so with respect to one result. In a single result, given a fixed number of occurrences of a feature type, the features that can be output are fixed: we output the features one by one in the order of their dominance, keeping ratios of occurrences whenever necessary, until we reach the given number of occurrences. On the other hand, given a fixed number of occurrences of a feature type in all results, there are many possibilities to assign these slots to all results, and the best slot assignment given a certain number of slots needs to be computed. We discuss two ways to compute the (benefit, cost) pairs for a feature type in the next two subsections: exact computation or heuristics computation.

The pseudo code of the framework of the feature type oriented algorithm is shown in Algorithm 9. We use $\mathcal{L} = n \times L$ as the total size limit for all DFSs, where L is the size limit for each individual DFS. For each feature type in the results, we compute a set of (benefit, cost) pairs with respect to all results (line 4, which will be detailed in the next subsections), then use dynamic programming (procedure DP) to find the optimal number of occurrences of each feature type in all results. Note that since this approach considers a single size limit for all DFSs, it may generate some DFSs whose sizes are larger than L. In this case, we perform a post-processing for these DFSs. For each DFS whose size exceeds L, we iteratively remove some features from it. Specifically, each time we pick one feature, such that removing this feature will cause the smallest loss of DoD. We do so until it has exactly L features. Similarly, for each DFS whose size is smaller than L, we iteratively add some features into it, until its size is L, or all features in the corresponding result has been added.

Since procedure DP computes a two dimensional array with size $nL \times m$, the time complexity of procedure DP is $O(mnL)$. The complexity of $constructDFS$ depends on line 4, which is executed m times, and will be discussed later.

Exact Computation of (Benefit, Cost) Pairs

To compute the exact (benefit, cost) pairs, we compute the benefit for all possible costs, i.e., from 1 to $\mathcal{L}$. For each possible cost c, we enumerate all possible ways to assign these c slots to the n results. The number of ways to assign c slots to n results equals to $\binom{c+n-1}{n-1}$. To see this, note that assigning c slots to n results such that each result has ≥ 0 slots is equivalent to assigning $c+n$ slots to n results such that each result has ≥ 1 slots. The latter problem can be considered as: there are $c+n$ points on the x-axis, each representing a slot. We insert $n-1$ “baffles”, such that each baffle is placed between two adjacent points, and no two baffles coincide. What is the number of ways to place all baffles? Note that for each placement of the $n-1$ baffles, we get an assignment of the slots: the number of points in between two baffles are the number of slots assigned to the corresponding result. For example, in Figure 8.5, result 1 and 2 are assigned 1 slot each; result n is assigned 2 slots. Therefore, the number of ways to assign the slots is equivalent to selecting $n-1$ from $c+n-1$, i.e., $\binom{c+n-1}{n-1}$.

Therefore, the exact computation of (benefit, cost) pairs is to enumerate all $\binom{c+n-1}{n-1}$

Algorithm 9 Feature Type Oriented DFS Construction

```
constructDFS (Query Results: $QR[1 \cdots n]$; Size Limit: $L$)
 1: $\mathcal{L} = n \times L$
 2: $ftype[1 \cdots m]$ =all feature types in $QR[1 \cdots n]$
 3: for $i = 1$ to $m$ do
 4:     $benefit[1 \cdots \mathcal{L}]$, $cost[1 \cdots \mathcal{L}]$ = computeBenefitCost_exact/heuristics($ftype[i], QR, \mathcal{L}$)
 5: $DFS[1 \cdots n] = DP(benefit, cost)$
 6: for each $i = 1$ to $n$ do
 7:     while $DFS[i].size > L$ do
 8:         $F$ = a feature in $DFS[i]$, such that removing $F$ from $DFS[i]$ causes the smallest loss of
            DoD
 9:         remove $F$ from $DFS[i]$
10:     while $DFS[i].size < L$ do
11:         $F$ = a feature in $DFS[i]$, such that adding $F$ to $DFS[i]$ gives the largest increase of DoD
12:         if $F$ = null then
13:             break
14:         add $F$ to $DFS[i]$
DP ($benefit[1 \cdots n], cost[1 \cdots n]$)
 1: $m$ = number of feature types in $QR$
 2: for $l$ = 1 to $\mathcal{L}$ do
 3:     compute $s_{1,l}$ according to Figure 8.3
 4:     Suppose $s_{1,l}$ is maximized by outputting $x$ features of type 1
 5:     $best_{1,l} = x$
 6: for $k$ = 2 to $m$ do
 7:     for $l$ = 1 to $\mathcal{L}$ do
 8:         compute $s_{k,l}$ according to Figure 8.3
 9:         Suppose $s_{k,l}$ is maximized by outputting $x$ features of type $k$
10:         $best_{k,l} = x$
11: $k = m$
12: $l = \mathcal{L}$
13: while $k > 0$ and $l > 0$ do
14:     output $x$ features of type $k$ in all DFSs
15:     $k - -$
16:     $l - = x$
```

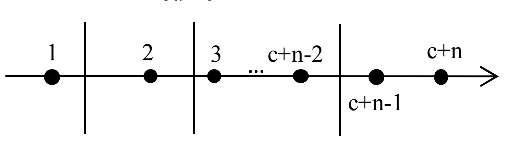


Figure 8.5: Points and Baffles

Algorithm 10 Exact Computation of Benefit and Cost

```
computeBenefitCost_exact ($ftype, result[1 \cdots n], \mathcal{L}$)
1:  for $c = 1$ to $\mathcal{L}$ do
2:      $DoD$ = computeBenefit($ftype, result, c$)
3:      $benefit[c] = DoD$, $cost[c] = c$
computeBenefit ($ftype, result[1 \cdots n], c$)
1:  {consider $n + c - 1$ points on the $x$-axis and $n - 1$ baffles}
2:  {A qualified baffle assignment: (1) each baffle is placed between two adjacent points; (2) no two
    baffles are placed between the same two points}
3:  {The baffles are recorded in $baffle[1 \cdots n - 1]$. $baffle[i] = j$ means the $i$th baffle is placed
    between points $j$ and $j + 1$}
4:  $bestBenefit = 0$
5:  for each qualified baffle assignment $baffle[1 \cdots n - 1]$ do
6:      $size[1] = baffle[1]$
7:      $size[n] = n - baffle[n - 1]$
8:      for $i = 2$ to $n - 1$ do
9:          $size[i] = baffle[i] - baffle[i - 1]$
10:     for $i = 1$ to $n$ do
11:         assign $size[i]$ slots to $DFS[i]$
12:     $benefit = 0$
13:     for $i = 1$ to $n$ do
14:         for $j = i + 1$ to $n$ do
15:             $benefit += DOD_{ftype}(DFS[i], DFS[j])$
16:     if $benefit >= bestBenefit$ then
17:         $bestBenefit = benefit$
18: return $bestBenefit$
```

ways of assigning c slots for each c ($1 \leq c \leq \mathcal{L}$). Note that this number is only exponential wrt n (the number of results), while polynomial wrt all other parameters. Since in reality a user will unlikely select a large number of results for comparison, this algorithm should work well practically.

The pseudo code of the exact computation of (benefit, cost) pairs is presented in Algorithm 10. All baffle assignments are enumerated for each cost $c(1 \leq c \leq \mathcal{L})$, and the assignments that has the largest benefit (DoD) is recorded for each c.

Now we analyze the complexity of Algorithm 10. For each cost c, we enumerate $\binom{c+n-1}{n-1}$ = $O(c^n)$ baffle positions. For each baffle position, we need to check whether the each pair of DFSs are differentiable. Let m denote the maximum number of features of each feature type. Since checking the differentiability of a feature type involves checking the order of features and the ratio of every two features, it takes $O(m^2)$ time, and thus checking the differentiability of all pairs of DFSs takes $O(n^2m^2)$ time. Therefore, the total complexity of Algorithm 10 is $O(\sum_{c=1}^{n\times L} c^n \times n^2m^2)$.

Heuristic Computation of (Benefit, Cost) Pairs

Although Algorithm 10 is only exponential with respect to n, it may still be inefficient in some situations. The main reason that may lead to its inefficiency is that for every possible cost, it needs to compute the optimal slot assignment from scratch, rather than incrementally. It has to do so because this problem does not have the optimal substructure property, in other words, to compute the optimal assignment of cost $c+1$, we are unable to reuse the optimal assignment of c or any other cost, as their optimal assignment may be totally different. When the number of results increases, both n and c increases (as each DFS has a fixed size limit), thus the processing time may increase very quickly.

Now we discuss an algorithm that heuristically computes the (benefit, cost) pairs with much better efficiency. The idea is to reuse the optimal assignment of lower costs when computing the optimal assignment of a higher cost, under the assumption that the optimal assignment of a higher cost likely does not differ too much from the optimal assignment of the lower cost. To do so, we use a vector $OptAsgnmt[1 \cdots \mathcal{L}]$, which records the optimal assignment of all costs we have computed so far. We compute $OptAsgnmt[i]$ by greedily adding a feature from $OptAsgnmt[i-1]$. For each i, suppose $OptAsgnmt[i]$ gives us a DoD of d_i, then we record a (benefit, cost) pair (d_i, i).

We start from processing $OptAsgnmt[0]$. $OptAsgnmt[0]$ is trivial: there is no (feature, percentage) pair in any result. To get $OptAsgnmt[i](1 \geq 1)$, we attempt to add a (feature, percentage) pair to each DFS from the best assignment of cost $i-1$, i.e., $OptAsgnmt[i-1]$. Suppose adding a (feature, percentage) pair to the jth result will give us the largest increase in DoD, then we add a (feature, percentage) pair to jth DFS from $OptAsgnmt[i-1]$ and consider it as $OptAsgnmt[i]$. Then, we go to the next cost, $i+1$, and process $OptAsgnmt[i+1]$.

Example 8.12 *Consider the two results in Figure 1.5(a), and feature type clothes:category (other feature types are processed in the same way). Table 8.4 shows the construction*

Table 8.1: An Illustration of the Heuristics Method for Computing (benefit, cost) Pairs

	D_1	D_2	DoD
$OptAsgnmt[0]$	empty	empty	0
$OptAsgnmt[1]$	clothes:category:outwear 52%	empty	0
$OptAsgnmt[2]$	clothes:category:outwear 52%	clothes:category:outwear 53%	1%
$OptAsgnmt[3]$	clothes:category:outwear 52% clothes:category:shirt 25%	clothes:category:outwear 53%	1%
$OptAsgnmt[4]$	clothes:category:outwear 52% clothes:category:shirt 25%	clothes:category:outwear 53% clothes:category:suit 47%	50%
$OptAsgnmt[5]$	clothes:category:outwear 52% clothes:category:shirt 25% clothes:category:sweater 13%	clothes:category:outwear 53% clothes:category:suit 47%	76%

*of $OptAsgnmt$ for feature type clothes:category. Initially, $OptAsgnmt[0]$ outputs no features of this type in either DFS, thus DoD = 0. Now we try to add a (feature, percentage) pair in a DFS in $OptAsgnmt[0]$, e.g. add (*outwear*, 52%) in D_1, as shown in Table 8.4. At this time, the DoD is still 0. We also attempt to add a (feature, percentage) pair to D_2 in $OptAsgnmt[0]$. Since this does not increase the DoD either, we do not update $OptAsgnmt[1]$ and its DoD. Then, from $OptAsgnmt[1]$, we continue to output one (feature, percentage) pair in a DFS. If we add a (feature, percentage) pair (*shirt*, 25%) in D_1 in $OptAsgnmt[1]$, since D_2 is empty, the DoD is still 0. On the other hand, if we add a (feature, percentage) pair (*outwear*, 53%) in D_2, we get a DoD of 1%. Therefore, in $OptAsgnmt[2]$, we assign one slot to each DFS. The process continues as shown in Table 8.4.*

The pseudo code of this algorithm is presented in Algorithm 11. We use $OptAsgnmt[c]$ to record the optimal slot assignment for cost c, and use $OptDoD[c]$ to record the benefit (i.e., DoD) achieved by $OptAsgnmt[c]$. We first initialize $OptAsgnmt[0]$ (line 1) and $OptDoD[c]$ for each c (line 4). Then start from $c = 1$, for each c, we compute the assignments of cost c based on $OptAsgnmt[c-1]$ (lines 5-12). We try to add a feature to each DFS, and compare the DoD obtained by all these n choices. Then, we select a DFS D such that adding a feature to D will give us the largest DoD. We add a feature to D and consider it as $OptAsgnmt[c]$ (lines 9-11). Finally, we record $\mathcal{L}$ (benefit, cost pairs), i.e., $(OptDoD[1], 1), \cdots, (OptDoD[\mathcal{L}], \mathcal{L})$ (lines 13-16).

Algorithm 11 Heuristic Computation of Benefit and Cost

```
computeBenefitCost_heuristics ($ftype, result[1 \cdots n], \mathcal{L}$)
 1: $OptAsgnmt[0] = \{0, 0, \cdots, 0\}$
 2: $OptDoD[0] = 0$
 3: for $i = 1$ to $\mathcal{L}$ do
 4:     $OptDoD[i] = -1$
 5: for $c = 1$ to $\mathcal{L} - 1$ do
 6:     for $i = 1$ to $n$ do
 7:         add a feature to $DFS[i]$ from $OptAsgnmt[c-1]$
 8:         $currDoD$ = $DOD_{ftype}(DFS[1], \cdots, DFS[n])$
 9:         if $currDoD \geq OptDoD[c]$ then
10:             $OptDoD[c] = currDoD$
11:             $OptAsgnmt[c] =$current slot assignment
12:     remove the newly added feature from $DFS[i]$
13: for $c = 1$ to $\mathcal{L}$ do
14:     if $OptDoD[c] \geq 0$ then
15:         $benefit[c] = OptDoD[c]$
16:         $cost[c] = c$
```

Now we analyze the complexity of Algorithm 11. For each cost c, we try to add a feature to each of the n results, then see whether this result is differentiable with any other results. Recall that checking whether a feature type can differentiate two results takes $O(m^2)$ time, and there are in total $n\mathcal{L}$ different costs. Therefore, the total complexity is $O(n^2\mathcal{L}m^2)$.

8.5 Evaluation

To verify the effectiveness and efficiency of our proposed approach, we implemented the XRed system and performed empirical evaluation from three perspectives: the *usefulness* of DFSs, the *quality* of DFSs, the *time* for generating the DFSs and the *scalability* upon the number of query results and the DFS size limit.

Environments and Setup

The evaluations were performed on a desktop with Intel Core(TM) 2 Quad CPU 2.66GHZ, 8GB memory, running Windows 7 Professional.

We used two data sets in our evaluation: a movie data set and a retailer data set. The movie data set records information about movies, which was extracted from IMDB.[2] The retailer data is a synthetic data set that records the information of apparel retailers and

[2]ftp://ftp.sunet.se/pub/tv+movies/imdb/

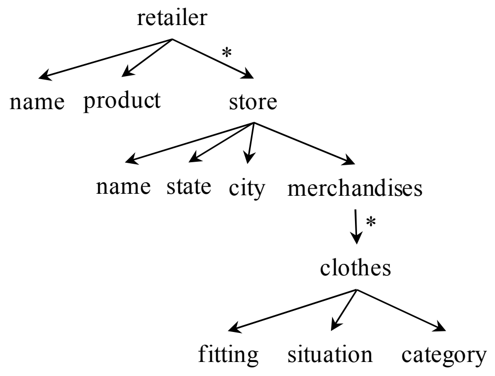


Figure 8.6: Schema of the Retailer Data

their stores. The schema of the retailer data is shown in Figure 8.6. The value of each node is randomly generated without functional dependencies. The test query set is shown in Table 1. The query results of these queries are generated using the approach discussed in Chapter 5.

To verify the usefulness of DFSs, we performed a user study on Amazon Mechanical Turk, in which we compare differentiating results using DFSs with differentiating results using result snippets, and using results themselves. The detailed setting of the user study is presented later in this section. For all other tests, for each query we generate DFSs for the first five results using six approaches: the single-swap optimal algorithm (Algorithm 7), the multi-swap optimal algorithm (Algorithm 8), the two feature type oriented algorithms (Algorithms 10, denoted as FTO-Exact and 11, denoted as FTO-Heuristics), a beam search algorithm and an algorithm that exhaustively searches for the optimal DFSs. The beam search algorithm first generates a set of initial states, and in each iteration, takes the top-k states and generates successors states of the top-k states, then takes the top-k successor states and repeats the iteration, till the search is finished. For our problem, each initial state contains one DFS for each result which contains a single feature. Given a state s containing a set of DFSs, each successor state contains one DFS for each result such that each DFS contains one more feature than the corresponding DFS in s. Thus beam search takes L iterations for our problem where L is the DFS size limit. We set k as 20 in the experiment. The DFS size limit is set as 10.

Table 8.2: Data and Query Sets for Testing Result Differentiation

Film	
QM_1	director, UK
QM_2	Italy, movie
QM_3	Austria, romance
QM_4	director, Yinka Adebeyi, Anthony Ainley, Sean Adames
QM_5	2002, Sci Fi, director
QM_6	UK, comedy, Anita
QM_7	1960, France, comedy
QM_8	actor, 2004, drama
Camera	
QR_1	store
QR_2	retailer, pants, children
QR_3	men, category, outwear, retailer
QR_4	Texas, pants
QR_5	men, outwear, footwear, shirts
QR_6	men, women, children, outwear
QR_7	casual, shirts, store
QR_8	retailer, casual, shirts

Usefulness of DFS

In this test we performed a user study with 50 users on Amazon Mechanical Turk, aiming at verifying the usefulness of DFSs given existing techniques for constructing result comparison tables. We compare with two result comparison approaches: (1) showing the snippets of the results to the user, as developed in [95]; (2) showing the results to the user, which is done by websites of banks such as chase.com (for comparing accounts, credit cards, etc.) and online retailers such as bestbuy.com (for comparing products). We made two modifications to approach (2). First, since a query result may have multiple occurrences of a feature (e.g., a store sells multiple DSLR cameras), we do not show users the entire result, but show them each distinct feature with its percentage, such as the infoboxes next to the results in Figure 1.5(a). Second, since many results are too big for the user to read, for each result, we only show the first few features (the number of features shown is the same as the DFS size limit) to the user (denoted as "result prefix").

For each of the 16 queries, we selected 2 or 3 results, and showed the users all distinct features together with their percentages in these results. Then, for each distinct feature type and each pair of results, the users are asked to select one of the following

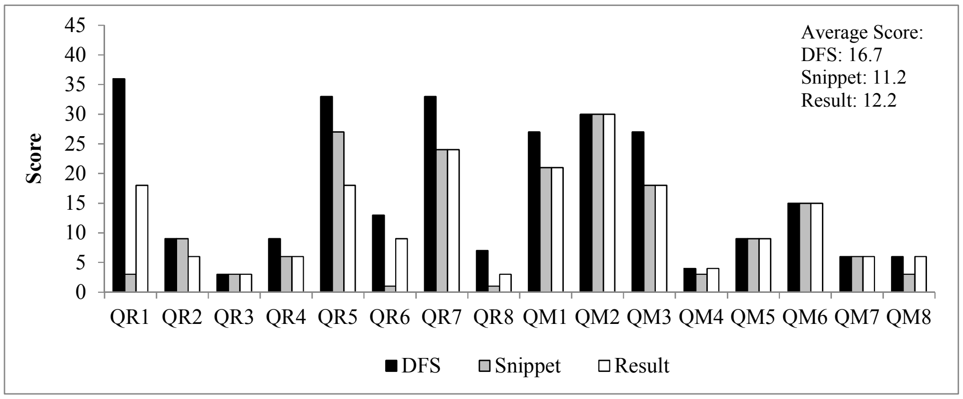


Figure 8.7: The Differentiation Powers of DFSs, Snippets and Result Prefixes

options:

(A) This feature type has the same features in the two results.

(B) This feature type slightly different features in the two results.

(C) This feature type significantly different features in the two results.

For each approach, if it shows the difference of a feature type which got option (A), (B) or (C) by most of the users, we give it a score of 0, 1 and 3, respectively.

The scores of each approach on each query is shown in Figure 8.7. For the DFS approach, we used the FTO-Heuristics method. As we can see, DFS shows significantly more differences than the other two methods. When the user compares the results by reading the result statistics itself, since a result may have many features, the users may often be able to read the first few features. However, the first few features may not show the differences of the results. For result snippets, as discussed in Section 8.1, although they output selected features in the results, the criteria of feature selection is based on whether a feature summarizes a single result, rather than whether a feature differentiates multiple results. Therefore, snippets are not designed for result differentiation and may not be helpful for the users to compare the search results. Note that the snippet method often has a worse performance than result statistics. This is because the snippets of different results may have completely different feature types, which are not comparable. On the other hand, the result statistics method uses the first several features in each result, which usually have the same type and thus it has a better chance of differentiating results.

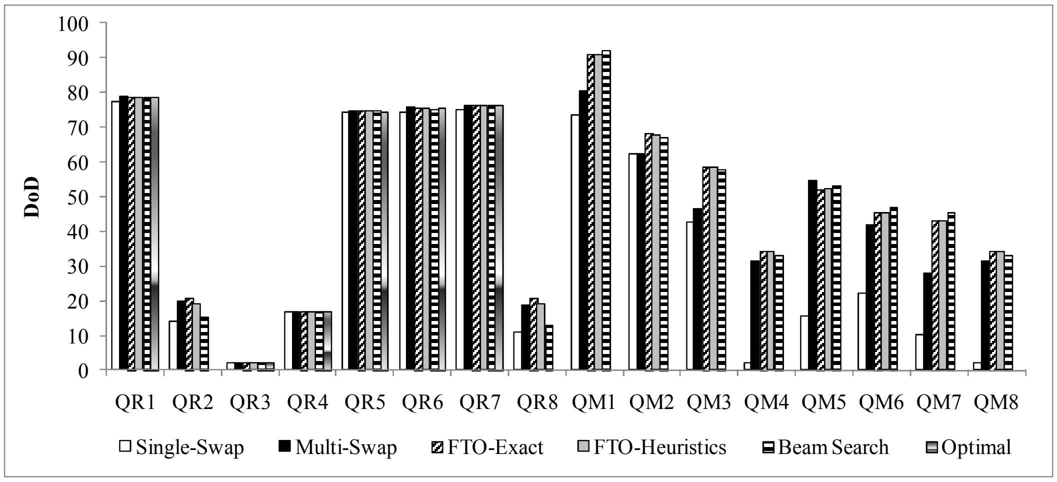


Figure 8.8: Quality of DFSs

Quality of DFS

For each query, the quality of the DFSs for its results is measured by their degree of differentiation (DoD) (Definition 8.2). The Optimal method is only shown for 6 queries, because for the remaining 10 queries it fails to terminate within 10 hours.

As we can see from Figure 8.8, all approaches achieve a DoD that is close to the optimal DoD, indicating good qualities of the algorithms. The multi-swap optimal algorithm usually exhibits a superior quality to the single-swap algorithm This is because the single-swap algorithm can only change one feature in a DFS at one time, and terminates if it cannot find such a change that can improve the DoD. For several queries such as QM$_4$, QM$_5$ and QM$_8$, the single-swap algorithm only achieves 5% to 30% of the DoD achieved by the multi-swap algorithm.

The feature type oriented algorithms generally achieve a higher DoD compared with the swap-based algorithms. As discussed in Section 8.4, these algorithms evaluate the quality of each feature type and selects the feature types in the order of their quality, thereby avoiding the problem of missing good feature types if they are not chosen initially. For queries such as QM$_3$ and QM$_7$, the feature type oriented algorithms achieve a significantly higher DoD than the swap-based algorithms. Note that the performance of swap-based algorithms are closer to the feature type oriented algorithms on the shopping data, since there are fewer distinct feature types in the shopping data compared with the movie data, thus the initialization of the swap-based algorithms will likely select all or most of the feature types into the DFSs, which results in a good quality of the swap-based algorithms.

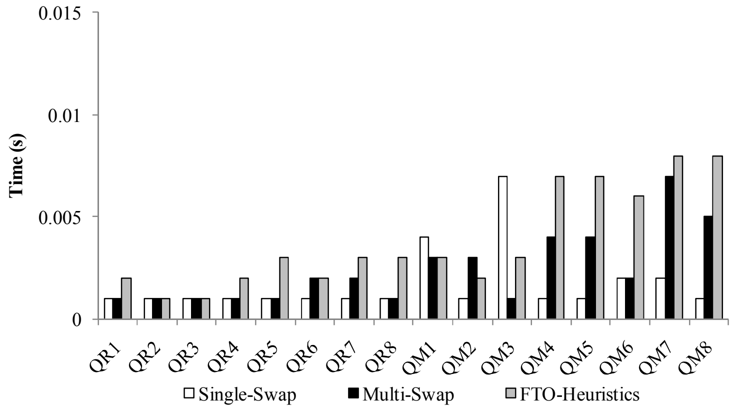
The FTO-Exact approach achieves slightly higher DoD than the FTO-Heuristics approach on four queries (QR_2, QR_8 and QM_2), since it is able to compute the exact set of (benefit, cost) pairs by enumeration. However, for query QM_5, note that FTO-Exact has a slightly lower DoD than that of FTO-Heuristics. This is because both algorithms start with a single DFS size limit for all DFSs, rather than a size limit for each DFS. Thus some of the initial DFSs they generate may have a size larger than L, the size limit for each individual DFS in the problem. If this happens, both algorithms perform a post-processing, which greedily removes some features from each DoD whose sizes exceed the limit, and greedily adds some features to each DoD whose sizes are smaller than the limit. Since this is a greedy procedure, the FTO-Heuristics approach may happen to get a better set of DFSs, which is the case for QM_5. However, for this query, FTO-Exact indeed achieves a better DoD for the initial set of DFSs generated (i.e., without post-processing).

The beam search algorithm has a similar DoD as FTO-Exact and FTO-Heuristics for all queries. It has slightly higher DoD than FTO algorithms for one third of the queries and has slightly lower DoD for another one third of the queries, indicating that the beam search algorithm generally has a good quality. However, as to be shown in the efficiency test, beam search algorithm is much slower since it needs to generate a large number of states.

Processing Time

To evaluate the efficiency of our algorithms, we measure the times that these approaches take to generate DFSs for the results of test queries in Table 8.2, which is shown in Figure 8.9. Since FTO-Exact, Beam Search and Optimal approaches are significantly slower than the others, their time are shown separately in a table to the right. “N/A” means that the Optimal approach does not finish within 10 hours for that query.

As we can see, the single-swap optimal algorithm generally achieve a better effi-



	FTO-Exact	Beam Search	Optimal
QR1	0.30	2.29	56.5
QR2	0.003	0.54	N/A
QR3	0.002	0.018	0.038
QR4	0.003	0.059	0.009
QR5	0.30	2.28	21.1
QR6	0.30	2.18	20.7
QR7	0.32	2.39	57.9
QR8	0.005	0.49	N/A
QM1	1.22	22.4	N/A
QM2	0.87	3.89	N/A
QM3	1.60	21.1	N/A
QM4	2.68	57.9	N/A
QM5	2.30	43.3	N/A
QM6	1.59	13.9	N/A
QM7	2.56	60.6	N/A
QM8	2.67	58.0	N/A

Figure 8.9: Processing Time of Generating DFSs

ciency compared with the multi-swap optimal algorithm. The single-swap algorithm enumerates all possible changes to a single feature in a single DFS in each iteration, and has the iteration repeat till no further improvements can be made. The multi-swap algorithm checks possible changes of any number of features in a single DFS in an iteration, which involves computing a set of (benefit, cost) pairs and a dynamic programming process, and can be potentially more expensive. However, by exploiting dynamic programming, overlapping subproblems are identified in achieving the optimal solution, and thus repetitious computation is avoided, thus the processing time of the multi-swap algorithm is still very short. FTO-Exact has the lowest efficiency as its complexity is exponential to the number of results. On the other hand, FTO-Heuristics generally has a better efficiency compared with multi-swap algorithm, as it directly evaluates each feature type and construct the DFS accordingly, thereby avoiding iteratively modify a DFS. The beam search algorithm is much slower even compared to the FTO-Exact algorithm. This is because the beam search algorithm needs to generate a large number of states to complete the search. In fact even the number of initial states is huge: each initial state corresponds to a set of DFSs, each with one feature, thus the number of initial states is bounded by T^R, where T is the number of feature types and R is the number of results. The beam search algorithm is slower on Movie data than on Retailer data since the Movie data has more feature types. On movie

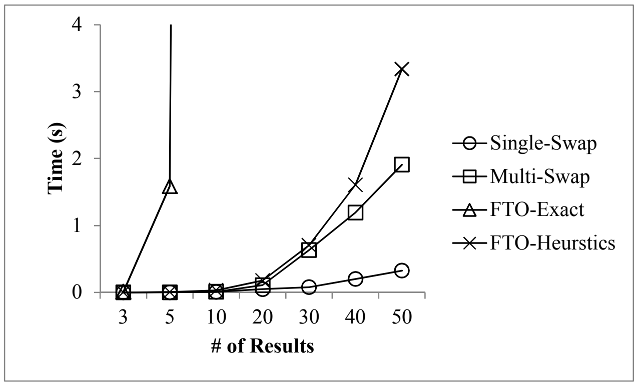


Figure 8.10: Processing Time of DFS Generation with Respect to the Number of Results

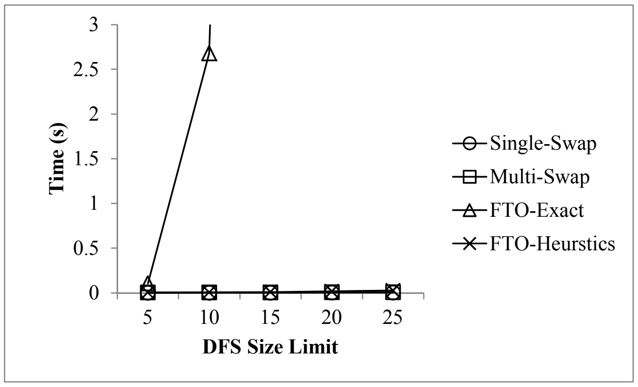


Figure 8.11: Processing Time of DFS Generation with Respect to DFS Size Limit

data, the average DFS generation time for the beam search algorithm is 35 seconds.

Scalability

We have tested the scalability of the single-swap optimal, multi-swap optimal, FTO-Exact and FTO-Heuristics algorithms over two parameters: *Number of Query Results* and *DFS Size Limit*. Since beam search is extremely inefficient, we do not test its scalability.

Number of Query Results. In order to increase the number of query results, we varied the number of results generated for QM$_6$ from 3 to 50. The DFS size limit is set as 10. The performance of the algorithms is shown in Figure 8.10. As we can see, the processing times of all algorithms increases with more results. The single-swap algorithm has the best scalability, as its complexity is proportional to n^2L, where n is the number of results and L is the size limit. The processing time of FTO-Heuristics increases faster than the single swap algorithm, since its complexity is actually proportional to $n^2\mathcal{L} = n^3L$. That being said, it still has a reasonable efficiency: in practice a user would rarely choose more than 50 results for comparison, while the DFS generation time of FTO-Heuristics for 50 results is less than 4 seconds. The multi-swap algorithm also increases faster than the single-swap optimal algorithm, since it changes multiple features of a DFS in each iteration. The processing time of FTO-Exact quickly deteriorates, as it is exponential to the number of results.

DFS Size Limit. In this test we evaluate the DoD and processing times of the four algorithms with respect to the increase of DFS size limit (i.e., the maximum number of features allowed in a DFS). We use the 5 results generated for QM$_8$. The efficiency of each approach is shown in Figure 8.11.

When the DFS size limit increases, the processing time of single-swap, multi-swap and FTO-Heuristics algorithms slightly increases, but the processing times of all these approach are close to 0. On the other hand, since the complexity of the FTO-Exact approach is proportional to L^n, its processing time increases very quickly. As we can see, although the FTO-Exact approach has the best quality among all four approaches, it is practical only if both the number of results and the DFS size limit are small.

To summarize, the FTO-Heuristics algorithm works best among these algorithms. It can achieve almost the same quality as that of FTO-Exact in most cases, but is much more efficient and scalable. It has superior quality and comparable efficiency compared with the single-swap and multi-swap optimal algorithm.

8.6 Summary

Informational queries are pervasive in web search, where a user would like to investigate, evaluate, compare, and synthesize multiple relevant results for information discovery and decision making. We initiate a novel problem: how to design tools that automatically differentiate structured data search results, and thus relieve users from labor intensive procedures of manually checking and comparing potentially large results. Towards this goal, we define Differentiation Feature Set (DFS) for each result and quantify the degree of differentiation. We identify three desiderata for good DFSs, i.e., differentiability, validity and small size. We then prove that the problem of constructing DFSs that are valid and can maximally differentiate a set of results within a size bound is an NP-hard problem. To provide practical solutions, we first propose two local optimality criteria, single-swap optimality and multi-swap optimality, and design efficient algorithms for achieving these criteria. Then we design an improved feature type oriented method which evaluates the quality of feature types using two alternative methods: exact computation and heuristics computation. The feature type oriented method achieves an improved DFS quality by avoiding dependency on the random initialization, which has a significant impact on DFS quality. Experiments verified the efficiency and effectiveness of the proposed approaches. Our proposed method is applicable to general query results which have features defined as (entity, attribute, value). To show the usefulness of our approach, these algorithms are implemented in the XRed system, which can be used to augment any existing structured keyword search engine that returns tree-structured results.

Chapter 9

DESCRIBABLE QUERY AWARE RESULT CLUSTERING

9.1 Motivation and Goal

So far we have discussed two result analysis methods, generating snippets in Chapter 7 and generating comparison tables in Chapter 8. Result clustering is another important approach for result analysis. Through result clustering, the users can quickly view different types of query results and choose the desirable cluster(s) to explore, hence result clustering helps alleviate structural ambiguity. In the same way result clustering also addresses keyword ambiguity and user preference ambiguity. In this chapter, we discuss how to cluster the results based on their structures and automatically generate a describable semantics description for each cluster. Chapter 10 will discuss how to improve the efficiency of clustering using result snippets, and Chapter 11 will discuss how to generate expanded queries from clustered results, where results are clustered based on the values in the results.

Clustering is especially important for queries that naturally have multiple interpretations. Consider the query "*auction, seller, buyer, Tom*" introduced in Chapter 1. There are many possible query semantics, some of which are:

1. Find the *seller* of the auction whose buyer is *Tom*
2. Find the *buyer* of the auction whose seller is *Tom*
3. Find the *buyer* and *seller* of the auction whose auctioneer is *Tom*

Note that the possible semantics are inferred according to the data. Indeed each above semantics has query results in the XML tree in Figure 1.6. For this query, whether a keyword should serve as a predicate or as a return node is ambiguous. For example, keyword *seller* can be either a predicate or a return node, depending on whether its value is *Tom*.

Another example of ambiguous queries could be "*country, government, republic*" where keyword *republic* may match a country name in some query results (such as Czech

Republic), and match the government type in other results, and therefore result in different return nodes for this query (e.g., government versus country). More ambiguous queries have been found and analyzed in Section 9.4.

As we can see, for an inherently ambiguous query, it is hard to infer which semantics among all possibilities the user intention corresponds to. This problem can not be solved even if a ranking scheme is introduced. Instead of returning all query results at one stage to the user, we propose to cluster similar query results, such that each cluster is describable using natural language sentences. Then, we can return only one representative of each cluster as well as the description of the cluster to the user. In this way, the user can easily see the interpretation of the query semantics for each cluster, and click on a cluster representative that s/he thinks relevant to see all the query results that belong to this cluster, which share the same semantic.

For example, for the query above "*auction, seller, buyer, Tom*", its query results can be divided into multiple clusters, such that each of the three semantics above comprises one cluster. Next we discuss how to cluster query results based on their semantics.

9.2 Clustering Search Results Using Inferred Keyword Categories

As can be seen from the example above, a keyword search can be ambiguous because a keyword may serve as a search predicate in some query results, but as a return node in others. We define an equivalence relationship among query results based on the classification of search predicates and return nodes of keywords.

Definition 9.1 *For a keyword search Q on data D, two query results R_1 and R_2 satisfy the equivalence relation* sim*, i.e., R_1* sim *R_2, if and only if for any keyword $k \in Q$, if it serves as a predicate (return node) in R_1, then it also serves as a predicate (return node) in R_2, and vice versa.*

Example 9.1 *Processing query "*auction, seller, buyer, Tom*" on the XML data in Figure 1.6,* auction *nodes (such as 0.3.0.0, 0.3.0.5, 0.4.0.9) are SLCA nodes. There are several types of query results, e.g.: (1) the type that has* auction, buyer, Tom *as search predicates,* seller *as a return node; (2) the type that has* auction, seller, Tom *as search predicates,* buyer *as a return node; (3) the type that has* auction, Tom *as search predicates,* seller *and* buyer *as return nodes. According to the definition, each query result belongs to a distinct equivalence class. Indeed the semantics of each query result is different from the others. Note that Figure 1.6 only shows a fragment of data, in general each equivalence class has many instances.*

For each equivalence class, one query result will be displayed at the first stage, together with the description of the cluster (which will be discussed later).

Algorithms. To effectively process such ambiguous queries, we propose an algorithm for efficiently clustering query results based on their equivalence classes.

Note that a query result contains a SLCA node and its keyword match descendants. To cluster query results into equivalence classes, we need to identify for each query result, whether a keyword is a return node or a search predicate.

We achieve this goal efficiently by assigning each SLCA node a boolean vector of 2$|Q|$ components, named as $dMatch$, where $|Q|$ is the number of keywords in query Q. Each component is either '0' or '1'. Each keyword k in Q corresponds to two components in $dMatch$, one representing a return node and the other representing a predicate. Specifically, 01 indicates that keyword k serves as a search predicate in the query result, 10 indicates that k serves as a return node, and 11 means that k serves as both. Note that 00 is impossible, as there are no other functionality that a keyword can serve.

The algorithm for processing ambiguous queries is presented in Algorithm 12. Procedure $calcDMatchAndRetNode$ (called in $KeywordSearch$ line 4) is an extension of the procedure $detRetNode$ in the algorithm of identifying relevant return information discussed in Chapter 5 (Figure 5.1). It partitions the SLCA nodes into equivalence classes according to their $dMatch$: two SLCA nodes (and hence their query results) are in the same equivalence class if and only if they have the same $dMatch$ value. To set the $dMatch$ of each SLCA node $SLCA[i]$, we check each keyword match in $group[i]$: $group[i][j]$. If $group[i][j]$ matches the p-th query keyword and serves as a return node, then the $2p$-th bit

Algorithm 12 Matching Keywords and Grouping Match Nodes and Clustering Results

```
$KeywordSearch$ ($keyword[1 \cdots n]$, $indexes$)
 1: $KWmatch = findMatch(keyword)$
 2: $SLCA = computeSLCA(KWmatch)$ {adopted from [144]}
 3: $group = groupMatch(KWmatch, SLCA)$
 4: $calcDMatchAndRetNode(keyword, group, SLCA)$
 5: for $j = 1$ to $SLCAclass.size$ do
 6:    $curr = 1$
 7:    Suppose $SLCAclass[j][1]$ is $SLCA[k]$
 8:    $genResult(findEntity(SLCA[k], k))$
$onClick$ ($SLCA[1 \cdots m]$)
 1: $j$ = id of the $SLCAclass$ that $SLCA[m]$ is in
 2: for $i = 1$ to $SLCAclass[j].size$ do
 3:    $curr = 1$
 4:    Suppose $SLCAclass[j][i]$ is $SLCA[k]$
 5:    $genResult(findEntity(SLCA[k], k))$
$groupMatch$ ($KWmatch[1 \cdots n]$, $SLCA[1 \cdots m]$)
 1: $dMatch[i]$ = 0 for all nodes in $SLCA$
 2: $i = 1$
 3: for $j$ = 1 to $n$ do
 4:    $cursor[j]$ = the first node in $KWmatch[j]$ that is a descendant-or-self of $SLCA[1]$
 5: while $(i \leq m)$ do
 6:    $j$ = keyword id such that $cursor[j]$ points to the node with the smallest Dewey ID
 7:    if $j = null$ then
 8:       break {This indicates $cursor[k] = null$, $1 \leq k \leq n$}
 9:    if $isDescendant(KWmatch[j][cursor[j]], SLCA[i])$ = $false$ then
10:       $i++$
11:    $group[i] = group[i] \cup KWmatch[j][cursor[j]]$
12:    $cursor[j]++$
13:    if $cursor[j] = null$ or $KWmatch[j][cursor[j]]$ is a descendant-or-self of $SLCA[i]$ then
14:       continue
15:    else if $i \leq n$ then
16:       $cursor[j]$ = first node in $KWmatch[j]$ that is a descendant-or-self of $SLCA[i+1]$
17:    else
18:       $cursor[j] = null$
$calcDMatchAndRetNode$ ($KWmatch[1 \cdots n]$, $group[1 \cdots m], SLCA[1 \cdots m]$)

 1: for $i$ = 1 to $m$ do
 2:    for $j$ = 1 to $group[i].size$ do
 3:       if $group[i][j]$ matches $keyword[p]$, $isName(group[i][j])$ and not
          $isAncestor(group[i][j], group[i][j+1])$ then
 4:          $retSpecified[i] = false$
 5:          set the $2p$-th bit of $SLCA[i].dMatch$ as 1 {A return node is met.}
 6:       else
 7:          set the $(2p+1)$-th bit of $SLCA[i].dMatch$ as 1 {A search predicate is met.}
 8:          $SLCAclass$ = partition SLCA nodes according to their dMatch
```

of $SLCA[i]$'s $dMatch$ is set as 1. Otherwise, $group[i][j]$ serves as a predicate, and the $(2p+1)$-th bit of $SLCA[i]$'s $dMatch$ is set as 1. The value of $retSpecified$ is also set according to keyword categories in this procedure. Then in the $KeywordSearch$ procedure, for each class, one SLCA and its associated query result is displayed. Upon the click of a query result, the $onClick$ procedure is invoked, which outputs the query results whose SLCAs are in the same equivalence class as the one being clicked.

Example 9.2 *We use the query "*auction, seller, buyer, Tom*" on the data shown in Figure 1.6 as a running example to illustrate Algorithm 12. Each* auction *node shown has at least one match to every keyword in the query and thus is a SLCA node. Since there are four keywords in the query, each SLCA node has a boolean vector $dMatch$ of size 8. Take the first* auction *node for example to see how to set $dMatch$. Initially, its $dMatch$ is 0. We process keyword matches that are descendants of this node in their Dewey ID order: {auction, seller, buyer, Tom}. First,* auction *is processed. Since it matches a node name and is an ancestor of the next match node* seller*, it is considered as a predicate. Therefore, the second bit of this $dMatch$ is set to 1, with $dMatch$ now becoming 01000000. Then we process the second match node in this query result,* seller *node. Since it matches a node name and is not an ancestor of the next match node* buyer*, it is considered as a return node. The third bit of this $dMatch$ is set to 1. Similarly,* buyer *is considered as a return node and has the fifth bit of $dMatch$ set to 1, and* Tom *is a predicate and has the eighth bit set to 1. Finally, the $dMatch$ of the first SLCA node is be 01101001. Similarly, the $dMatch$ of the second and third* auction *nodes are set as 01100101 and 01011001, respectively. Therefore, there are three query result equivalence classes on the sample data, each of which has a distinct $dMatch$.*

9.3 Clustering Search Results with a Controlled Number of Clusters

By clustering query results according to keyword categories as discussed in Section 9.2, we are able to describe the semantics of each cluster so that the user can easily understand what the commonalities of the results in the same cluster are with respect to the query. However, such a clustering scheme does not give users any control of the number of clusters. This can often be undesirable as the user may want to control the granularity of the clustering. Indeed the results that share the same return nodes and predicates can still be quite different from each other, making the semantics of a cluster too general to be useful. In this section, we discuss how to get the best of both: clustering results with a controlled number of clusters and yet has the property that the semantics of each cluster is describable.

Example 9.3 *Continuing Example 9.2: for query "*auction, seller, buyer, Tom*" on the XML data in Figure 1.6, the results are clustered based on keyword categories in three clusters: (1) The seller of the auction is Tom; (2) The buyer of the auction is Tom; (3) The auctioneer of the auction is Tom.*

In a cluster, the results may be rooted at closed auction*, or* open auction*, and the corresponding year information may also be different. Sometimes a user may prefer a larger number of clusters, each with more precise and specific semantics. To this end, we can further cluster the results in a cluster according to the closest context (i.e., parent) of the keyword matches. In this example, we split cluster (1) to two, one containing results where the parents of* seller *and* buyer *are* closed auction*, with the semantics "Find the* seller *and* buyer *of closed auctions whose seller is* Tom*", while the other corresponds to* open auction*. Should more clusters be asked, we further split each cluster according to the next closest context, i.e. the closest ancestors of the previous split points, which can make a difference. In this example, since the parent of* closed auction *(*open auction*) nodes are the same, the cluster is further split into multiple clusters corresponding to the next ancestors* 2008*,* 2009*, etc. The clustering can be further refined recursively.*

In general, a user may specify that the desired number of clusters is k, and our techniques should generate clusters whose number is as close to k as possible, while keeping each cluster describable. Being able to describe the commonalities of results in a cluster is the basic philosophy of our approach, which is especially important for clustering query results. To achieve it, we refine clustering recursively based on the closest ancestors of the previous split nodes, as they have closer relationships to keyword matches and provide more specific contexts. The results of each cluster share the same set of data paths ending at the keyword matches, named as *suffix paths*, which can be used in their semantics description as shown in Example 9.3.

Furthermore, the refinement of clustering should be balanced, in order to avoid clusters with semantics that is neither too general nor too specific. To illustrate it, let us look at an example.

Example 9.4 *Continuing Example 9.3, where there are 3 result clusters according to keyword categories, and a user specifies $k = 20$ as the desirable number of clusters. To achieve this, we could split cluster (1) into 18 clusters according to* closed auction *versus* open auction*, and then the* year *information, and keep the other two original clusters intact. However, it is not a balanced clustering in terms of both the semantics and the size of a cluster. The 18 clusters resulted from the original cluster (1) are likely small, with very specific semantics, while the other 2 clusters are on the opposite side.*

Thus to achieve a balanced clustering, we require that the suffix paths that are used for cluster splitting should not have a length difference larger than 1.

Specifically, the problem of result clustering given a user specified number of clusters is the following:

Definition 9.2 *Given a set of structured search results and a desired number of clusters k, cluster the results according to keyword categories if the number of such clusters is no less than k; otherwise cluster the results with the following two conditions and make the number of clusters is as close to k as possible: (1) The clusters should be split based on the suffix paths of keyword matches, and (2) The lengths of such suffix paths should not be different by more than 1, unless the paths reach the result root.*

It is noteworthy that we do not require the user to specify the number of clusters. If it is not specified, we generate the clusters according to node categories (Section 5.2). If the user is interested in a specific number of clusters, he/she can input the desired number, which is typical for clustering problems. In this case we will generate clusters whose number is as close to the desired number as possible while ensuring each cluster is describable.

Theorem 9.1 *The result clustering problem defined in Definition 9.2 is NP-hard.*

Proof: We prove the NP-completeness of the decision version of the problem by reduction from the subset sum problem.

The decision version of the result clustering problem is: whether the number of generated clusters is exactly k, while satisfying conditions (1) and (2) in Definition 9.2. Obviously, any given solution to this problem can be verified in polynomial time.

Now we prove the subset sum problem is reducible to the result clustering problem in polynomial time. Recall that the subset sum problem is: given a set of n positive integers $d_1, \cdots, d_n$ and a target number t, whether we can find a subset of them such that their sum is exactly t. Given any instance of the subset sum problem, we transform it to an instance of the result clustering problem by the following approach: suppose there are n initial clusters obtained considering keyword categories. According to the parents of keyword matches, each cluster i ($1 \leq i \leq n$) can be further split into c_i clusters, where c_i = $d_i + 1$. The question is whether we can find a way of splitting a subset of clusters such that the total number of clusters obtained is k, $k = t + n$. This transformation obviously takes polynomial time.

It is easy to see that, if in the subset sum problem we can select a subset of set $\{d_1, \cdots, d_n\}$, such that their sum is t, then we can split the corresponding subset of the clusters, and the final number of clusters will be $t + n$. The opposite direction is similar. Therefore the decision version of the result clustering problem is NP-complete, and thus the result clustering problem is NP-hard.

Algorithms. We propose a dynamic programming algorithm to solve the result clustering problem. The main steps of the algorithm are the following:

(1) Cluster the results based on keyword categories using the algorithms in Section 9.2. If the number of clusters is no less than k, stop and output the clusters.

Algorithm 13 Grouping Match Nodes Subject to the Number of Clusters

```
$K - Clustering$ ($keyword[1 \cdots n]$, $indexes$, $group$, $k$)
 1: if $SLCAclass.size \geq k$ then
 2:     return {We cannot have less than $SLCAclass.size$ clusters.}
 3: $level = 0$
 4: $D$ = XML document depth
 5: while true do
 6:     $level++$
 7:     if $level > D$ then
 8:         return {We've reached the root and cannot have more clusters}
 9:     $ancEntitySet$, $newCluster$, $newClusterCount = \emptyset$
10:     $sum = 0$
11:     for $i = 1$ to $SLCAclass.size$ do
12:         for $j = 1$ to $SLCAclass[i].size$ do
13:             $groupID = SLCAclass[i][j]$
14:             for $r = 1$ to $n$ do
15:                 $ancEntitySet[i][j][r]$ = the node labels of the $level$th ancestor of matches in
                    $group[groupID]$ to $keyword[r]$
16:         Split $SLCAclass[i]$ into $newClusterCount[i]$ new clusters according to
            $ancEntitySet[i][j][r]$
17:         add the new clusters into $newClusters$
18:         $sum += newClusterCount[i]$
19:     if $sum < k$ then
20:         $SLCAclass = newCluster$
21:     else
22:         break {We get more than $k$ clusters at the next level.}
23: $v = SLCAclass.size$
24: let $d[i] = newClusterCount[i] - 1$ for each $i$
25: $result1$ = $DP(d[v], k - v, best1[v][result1])$
26: $result2$ = $DP(d[v], sum - k + v, best2[v][result1])$
27: $result$ = $(k - v - result1) < (sum - k + v - result2) ? result1 : result2$
28: $best[result]$ = $(k - v - result1) < (sum - k + v - result2) ? best1 : best2$
29: $i = v$, $j = result$ {The following "While" loop finds the clusters to split.}
30: while $i \geq 1$ and $reuslt \geq 1$ do
31:     if $best[i][j]$ = split then
32:         split $SLCAclass[i]$ into $newClusters$ {Split this cluster.}
33:         $i--$, $j- = d[v]$
34:     else
35:         add $SLCAclass[i]$ into $newClusters$ {Do not split this cluster.}
36:         $i--$
37: $SLCAclass = newClusters$
$DP$ ($d[v]$, $k$, $best$)
 1: for $i = 1$ to $v$ do
 2:     for $j = 1$ to $k$ do
 3:         compute $s[i][j]$ according to the recurrence relation in Figure 9.1
 4:         if $s[i][j]$ is maximized by splitting cluster $i$ then
 5:             $best[i][j]$ = split
 6:         else
 7:             $best[i][j]$ = not split
 8: return $s[v][k]$
```

(2) Determine the shorter lengths of the suffix paths used for clustering by recursively splitting every existing cluster using the next ancestors of keyword matches, until the number of clusters is closest but not exceed k (that is, if we further split every existing cluster, we obtain more than k clusters). Note that after this step, the length of the suffix paths for each cluster is the same. If the total number of clusters is equal to k, stop and output the clusters.

(3) Determine which cluster(s) to further split, such that the total number of clusters is closest to k, using dynamic programming. These clusters correspond to longer suffix paths, whose lengths are one bigger than the ones in step (3). Output the clusters.

Now we focus the discussion on how to address the challenge of step (3), as determining a subset of the clusters in step (2) for further split is NP-hard according to Theorem 9.1. Suppose we have n clusters ($n < k$) after step (2). In step (3), we aim at splitting some of these n clusters to get k clusters. For each cluster i, if we split it, we get c_i clusters, otherwise we get 1 cluster. Equivalently, if we split it, we increase the number of clusters by c_i-1, otherwise we do not increase the number of clusters, and our goal is to increase the number of clusters by $k-n$. This is essentially to find a subset of $\{c_1-1, c_2-1, \cdots, c_n-1\}$, such that their sum is as close to $k-n$ as possible.

The solution to the above problem can be found in the following two steps:

(a) Find a subset of $\{c_1-1, c_2-1, \cdots, c_n-1\}$, such that their sum is as close to $k-n$ as possible, but not bigger than $k-n$.

(b) Find a subset of $\{c_1-1, c_2-1, \cdots, c_n-1\}$, such that their sum is as close to $k-n$ as possible, but not smaller than $k-n$.

Then, we compare the subsets found in (a) and (b), and take the one whose sum is closer to $k-n$.

Note that problem (b) is equivalent to: finding a subset of $\{c_1-1, c_2-1, \cdots, c_n-1\}$, such that their sum is as close to $\sum_{i=1}^{n}(c_i-1)-(k-n)$ as possible, but not bigger than it, then take the complement of the subset. The reason we use this equivalent formulation of problem (b) is that we want to use the same procedure to solve both (a) and (b): given a

$$s_{i,j} = \begin{cases} 0 & \text{if } i = 1, d_1 > j \\ d_1 & \text{if } i = 1, d_1 \le j \\ s_{i-1,j} & \text{if } i = 1, d_i > j \\ \max(s_{i-1,j}, s_{i-1,j-d_i} + d_i) & \text{if } i = 1, d_i \le j \end{cases}$$

Figure 9.1: Recurrence Relation for Generating a Certain Number of Clusters

set of numbers, find a subset whose sum is closest but not exceeding a given number.

Using this procedure, let $d_i = c_i - 1$, now we want to find a subset of a set of integers $\{d_1, \cdots, d_i\}$, such that the sum is as close to an integer t as possible but not exceeding t, where the value of t is $k - n$ for step (a), and $\sum_{i=1}^{n}(d_i) - (k - n)$ for step (b).

We use the recurrence relation in Figure 9.1, in which $s_{i,j}$ denotes the maximum possible sum that does not exceed j, by choosing a subset from the set of the first i numbers: $\{d_1, \cdots, d_i\}$. Note that if $d_i > j$, then we are unable to use d_i. Thus when $i = 1$, $s_{i,j}$ is initialized to be 0; and when $i \ge 1$, we have $s_{i,j} = s_{i-1,j}$. Otherwise ($d_i \le j$), when $i = 1$, $s_{i,j}$ is initialized to be d_1. When $i \ge 1$; we have two choices: (a) use d_i, which means that the sum of the numbers chosen from $\{d_1, \cdots, d_{i-1}\}$ cannot exceed $j - d_i$, and by choosing d_i the sum becomes $s_{i-1,j-d_i} + d_i$; or (b) do not use d_i, which means $s_{i,j}$ is the same as $s_{i-1,j}$. Since we want to maximize the sum, we take the larger value of these two cases.

Example 9.5 *Continuing our example, suppose the user prefers the number of clusters to be 5. Step (1) of result clustering generates three clusters based on keyword categories, as achieved in Algorithm 12: (1) The seller of the auction is Tom; (2) The buyer of the auction is Tom; (3) The auctioneer of the auction is Tom. Suppose each of these three clusters contains results rooted at* closed auction *as well as ones rooted at* open auction*.*

In step (2) of this algorithm, we attempt to split every existing cluster. Take cluster (1) for example. We examine the nearest ancestors of the keyword matches in the results, and find that matches to seller *have two different parents in the results:* open auction *and* closed auction*. So does matches to* buyer*. Thus cluster (1) could be split into two clusters. Same for cluster (2) and (3). Since these splits will result in 6 clusters, which is larger than 5, we do not perform the split, but exit step (3) and enter step (4). Note that if the desired* *number of clusters is more than 6, then we stay at step (3) and try to further split the clusters using the next ancestors of keyword matches.*

In step (3), we run a dynamic programming procedure to determine which cluster(s) obtained from step (3) to further split, so that the total number of clusters is closest to 5. The result of this procedure is that two clusters are split, and one remains.

The pseudo code of the clustering algorithm with a user input number of clusters is presented in Algorithm 13. The $K - clustering$ procedure is called between lines 4 and 5 of procedure *KeywordSearch* in Algorithm 12 to further split each cluster. To implement step (2), in each "while" loop in the $K - clustering$ procedure (between lines 5 and 22), we examine whether further splitting every existing cluster using the next ancestors of keyword matches still results in less than k clusters. If so, we further split every cluster. Otherwise, we proceed to step (3): compute which subset of the clusters to split so that the total number of clusters is closest to k (lines 23-36). Then we call the dynamic programming procedure (DP) to find the largest number of clusters that is smaller than $k - n$ (lines 25), and the smallest number of clusters that is larger than $k - n$ (line 26), respectively. The DP procedure computes the best way of choosing a subset of clusters to split based on the recurrence relation in Figure 9.1. Then we choose the subset of clusters to split so that the total number of clusters is closest to k (lines 27-36).

Note that although Algorithms 12 and 13 is for XML search results, they can be naturally modified to support clustering on tree-structured results generated from other structured data.

Having discussed how to cluster search results based on the classification of predicates and return nodes as well as the user-specified desirable number of clusters, now we discuss how to describe the semantics of each cluster. Note that the results in each cluster have the same set of predicates and return nodes. We use "Find ret" for return nodes, where ret is explicit or implicit return nodes. Recursively, for each ancestor entity of the return nodes, we add a phrase "of $entity$", where $entity$ is the name of the entity if it is consistent throughout all results in the cluster, otherwise we use the word "entity". For predicates, we recursively check whether the same predicate in all the results corresponds to the same entity and/or attribute. If so, we associate the entity / attribute name in the description; otherwise we simply use the term “entity” or “attribute”.

Example 9.6 *For query “*auction, seller, buyer, Tom*”, consider a cluster of results, each of which is rooted at a* closed auction *under* 2008*, the auctioneer is* Tom*, and the buyer and seller are not* Tom*. For return nodes* seller *and* buyer*, we generate phrase “Find seller and buyer of closed auctions”. For predicates* auction *and* Tom*, all results in this cluster have* Tom *as the value of attribute* auctioneer *of entity* closed auction*. We generate the English sentence as “Find seller and buyer of closed auctions of 2008, whose auctioneer is Tom”.*

If some results in this cluster are rooted at open auction *rather than* closed auction*, then we will use “entities” to replace “auctions”, and the description becomes “Find seller and buyer of entities of 2008 whose auctioneer is Tom”.*

The ability of generating an English sentence to describe each cluster is a unique feature of XSeek. Existing tree clustering approaches, which are based on tree edit distances or vector space model [11, 87, 47, 107, 140, 128, 52, 137, 136], are unable to describe the semantics of each cluster, as the results in each cluster may not have a common semantics. Indeed, the three results in Figure 1.6 rooted at 0.3.0.0., 0.3.0.5 and 0.4.0.9 are quite similar to each other and would be put into the same cluster by existing approaches, whose semantics are unclear.

9.4 Experiments

To test the quality and efficiency of clustering, we performed a set of experiments, whose results are reported in this section.

Experimental Setup

Comparison Systems. We compare three approaches. The first approach, named XSeek, is the one discussed in Chapter 5. XSeek does not perform result clustering. The approach discussed in this section has been implemented on top of XSeek and is denoted as

XSeek+Cluster. The desired number of clusters is set as 5. The output of the XSeek+Cluster approach consists of one query result of each cluster, and all the query results in the clusters that are clicked by the user. Which cluster the user clicks depends on the semantics given by the user study. The third approach is XSeek+C-index [46], which performs clustering based on tree edit distance. This approach is query-independent. We use the results generated by XSeek as the input to C-index. For this approach, we pick the first result it outputs in each cluster and show it to the user. Same as XSeek+Cluster, whether the user clicks a cluster depends on the semantics of the query obtained by user study.

Machine. The experiments were performed on a Pentium 4 3.60GHz machine running Windows 2003 Enterprise, with 2GB main memory and 160GB hard disk (7200rpm). All approaches were implemented and compiled with Visual C++. We used Oracle Berkeley DB[1] to build on-disk Dewey index and Keyword index.

Data Sets. We have tested three XML data sets, SigmodRecord[2], Mondial[3] and Auction[4]. SigmodRecord is a list of articles grouped by issue and volume. Mondial is a world geographic database integrated from the CIA World Factbook, the International Atlas, and the TERRA database among other sources. Auction is a synthetic benchmark data set generated by the XML Generator from XMark using the default DTD. These datasets represent XML data of varying structures: SigmodRecord has a tree structured schema with a depth of 6; Mondial has a graph structured schema with a depth of 5, and can have nodes of the same name appear in different depths; and Auction has a recursive graph schema with the test data of depth 11. The numbers of elements in the DTDs of SigmodRecord, Mondial and Auction are 11, 23 and 78, respectively.

Query Sets. We use 30 queries in total: 10 on each data set, as shown in Figures 9.2 - 9.4. The sentence below each query is the query's semantics, obtained through user study. Six users participated in the user study, who are undergraduate and graduate students majoring in computer science who did not involve in this project. For each query, each user was asked to describe in natural language the semantics of the query. If there

[1]http://www.oracle.com/technology/products/berkeley-db/

[2]http://www.cs.washington.edu/research/xmldatasets/data/sigmod-record/SigmodRecord.xml

[3]http://www.cs.washington.edu/research/xmldatasets/data/mondial/mondial-3.0.xml

[4]http://monetdb.cwi.nl/xml/

are more than one semantics, the ground truth is the one provided by the majority of the users.

Our query sets include all the following cases: keywords matching tag names only (such as QS_7, QM_7, QA_5, QA_6), keywords matching values only (such as QM_2, QM_6, QA_7), single-keyword queries (QM_2), SLCA nodes with big depths (thus the subtree rooted at each SLCA is relatively small, such as QS_6, QS_7, QM_{10}, QA_2), SLCA nodes with small depths (QS_5, QM_5, QA_7), keywords serving as predicates only (QS_{10}, QM_5, QA_3), keywords serving as return node only (QS_7, QM_7, QA_5, QA_6), keywords serving as both (QS_8, QA_4, QA_9). Furthermore, 10 ambiguous queries are tested, QS_4, QS_8, QS_9, QM_8, QM_9, QM_{10}, QA_3, QA_8, QA_9, QA_{10}. Each of these 10 queries have at least 2 different semantics and can produce at least 2 describable clusters. For example, QS_4 is ambiguous, because a number (such as 3 and 7) can appear as a value of one or multiple of the four attributes: issue, volume, initPage and endPage. This means that initPage and endPage can both serve as predicates or return nodes, depending on whether their value is 3, or 7, or neither. In fact, this query intends to find the articles with initPage 3 and endPage 7; thus only the articles whose initPage and endPage are both predicates are relevant. Other ambiguous queries are ambiguous for similar reason.

Search Quality

Measurements. To measure the search quality, we use *precision*, *recall*, and *F-measure*. Precision measures the percentage of the output nodes that are desired, recall measures the percentage of the desired nodes that are output, and F-measure is their harmonic mean.

The precision, recall and F-Measure of the 30 queries are shown in Figures 9.5 - 9.7. As we can see, XSeek suffers from a low precision for ambiguous queries. Take QS_4 as an example. According to user intention, both *initPage* and *endPage* are predicates, specifically, the articles whose initPage is 3 and endPage is 7 are desired. However, XSeek will output all subtrees rooted at SLCA that contain the keywords, which includes a lot of irrelevant ones, e.g. (i) the article whose volume number is 3 and initPage is 7, (ii) the issue whose number is 3 and has articles with endPage 7, (iii) the articles in issue 3 and

SigmodRecord: 61MB	
QS_1	B-Tree, volume, author *Find the volume and author of the articles whose titles include “B-tree"*
QS_2	author, position, 01, Harry, article *Find the articles with Harry as the second author*
QS_3	Jim Gray, title, initPage, endPage *Find the initPage and endPage of articles written by Jim Gray*
QS_4	initPage, 3, endPage, 7 *Find the articles whose initPage is 3 and endPage is 7*
QS_5	SigmodRecord, Laura *Find the information of Laura in the SigmodRecord data*
QS_6	author, Nicolas *Find the authors that work together with Nicolas*
QS_7	article, title, author *Find the titles and authors of all articles*
QS_8	initPage, 7, article, endPage *Find the endPage of articles whose initPages are 7*
QS_9	volume, 11, article *Find the articles in volume 11*
QS_{10}	Asuman, Pinar, article *Find the articles written by Asuman or Pinar*

Figure 9.2: Queries on SigmodRecord Data for Testing Result Clustering

volume 7, etc. In fact, XSeek+Cluster produces 5 clusters for this query, and not all of them are relevant. By clustering the results, XSeek+Cluster has a much better precision, since only one result from each irrelevant query result cluster will be output. For similar reasons, XSeek+Cluster achieves an improved precision compared with XSeek for other ambiguous queries QS_8, QS_9, QM_8, QM_9, QM_{10}, QA_3, QA_8, QA_9 and QA_{10}.

Note that XSeek+Cluster always has a non-perfect precision for ambiguous queries, as it outputs one result from each cluster, including the irrelevant clusters. When the irrelevant results are large, the precision of XSeek+Cluster can still be low. In particular, for QA_3, an irrelevant result is an open auction whose “end” attribute matches 2000. Such a query result is larger than the total size of all relevant query results, each of which consists of only one article. For QM_8 and QM_9, there are a lot of “country” nodes occurring as children of

Mondial: 68MB	
QM_1	Torneaelv, country, province *Find the country and province of Torneaelv*
QM_2	Luanda *Find the information of Luanda*
QM_3	Roman Catholic, percentage, United States *Find the percentage of religion “Roman Catholics" in the United States*
QM_4	population, 87, Albania, city *Find the populations of the cities in Albania in 1987*
QM_5	mondial, Africa *Find the information of Africa in the mondial data set*
QM_6	Bulgaria, Serb *Find the countries whose ethnic groups contain “Balgaria" or “Serb"*
QM_7	organization, name, members *Find the name and members of all organizations*
QM_8	country, government, republic *Find the countries whose government types are republic*
QM_9	country, ethnicgroups, German *Find the countries whose ethnic groups contain “German"*
QM_{10}	city, Washington, province *Find the province of the city called “Washington"*

Figure 9.3: Queries on Mondial Data for Testing Result Clustering

“province” and “city” nodes, which are irrelevant as the query asks for the information of the country. Therefore, each relevant result has a large number of irrelevant nodes, and this, together with the irrelevant results, makes the precision quite low.

For ambiguous queries, XSeek+Cluster aims to generate 5 result clusters for each query, but it may not always be able to do so due to the requirement that each cluster has a describable semantics. XSeek+Cluster generate 2 clusters for QM_{10}, 3 clusters for QA_9, 4 clusters for QS_8, QM_9 and QA_8, 6 clusters for QM_8 abd QA_3, and 5 clusters for the remaining ambiguous queries (QS_4, QS_9, QA_{10}).

For XSeek+C-index, its precision and recall are usually lower than XSeek+cluster. This is because XSeek+C-index performs query-independent clustering, thus a cluster may

Auction: 103MB	
QA_1	name, augurers, internationally, description *Find the description of the items, whose names contain “augurers" and whose shipping methods contain “international"*
QA_2	Arun, emailaddress *Find the email address of Arun*
QA_3	date, 2000 *Find the events occurred in 2000*
QA_4	Oakland, zipcode *Find the zip code of Oakland*
QA_5	closed auction, price *Find the price of all closed auctions*
QA_6	closed auction, price, date, itemref, quantity, type, seller, buyer *Find the price, date, itemref, quantity, type, seller, and buyer of all closed auctions*
QA_7	person257, person133 *Find the activities of person257 and person133*
QA_8	auction, seller, person133 *Find the auctions whose sellers are person133*
QA_9	seller, person179, buyer, price, date *Find the buyer, price and date of the auctions whose sellers are person179*
QA_{10}	country, name *Find the country and name of all persons*

Figure 9.4: Queries on Auction Data for Testing Result Clustering

have both relevant and irrelevant results. Whether such a cluster is browsed by the user depends on the first result in the cluster: the cluster is browsed if the first result is relevant. If the cluster is browsed, it leads to a low precision; otherwise it leads to a low recall. Another reason XSeek+C-index has low precision and recall is because it sometimes considers some results as outliers and does not put them into any cluster. Such results will not be seen by the user. In the extreme cases like QA_8 and QA_9, all relevant results are considered outliers, and XSeek+C-index has a zero precision and zero recall. XSeek+C-index outperforms XSeek for most queries since the clustering quality is usually reasonable, thus many irrelevant results are hidden from the user.

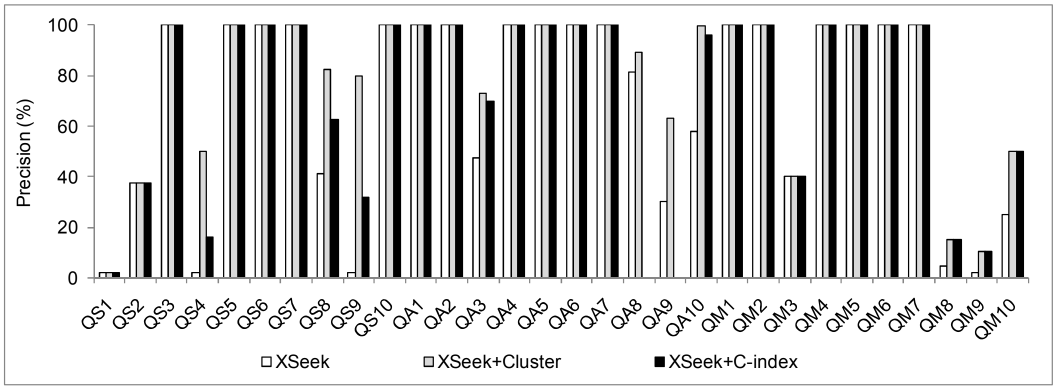


Figure 9.5: Precision of Clustering

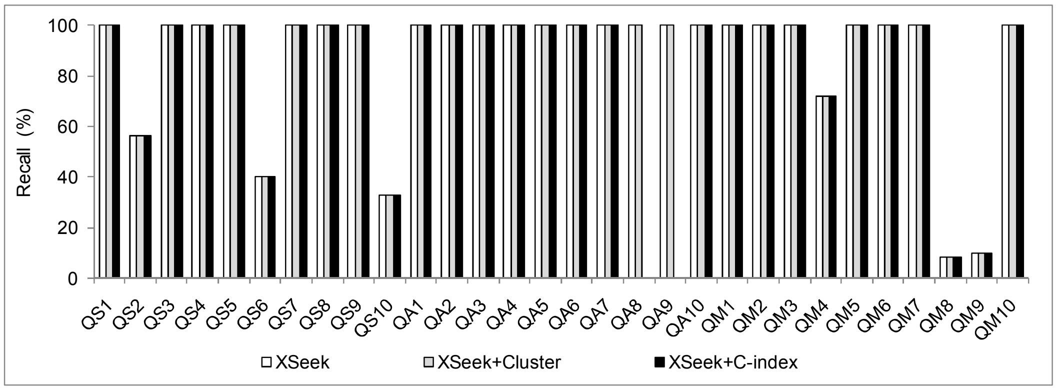


Figure 9.6: Recall of Clustering

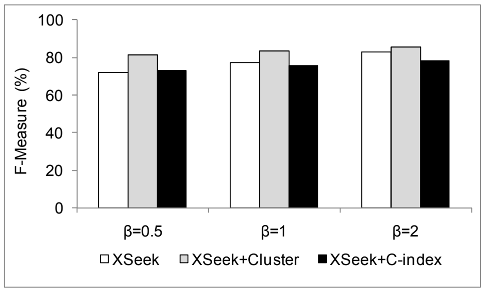


Figure 9.7: F-Measure of Clustering

Efficiency

We evaluated the processing times of index construction as well as query processing. The time for building indexes for Sigmod Record, Mondial and Auction (during which we also infer node categories) are 274 seconds, 320 seconds and 295 seconds, respectively. Note

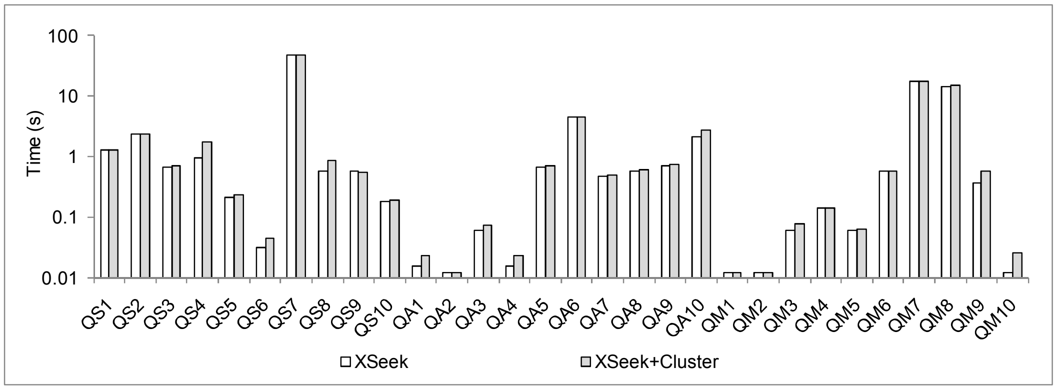


Figure 9.8: Efficiency of Clustering

that indexing building is performed offline, and does not affect the query processing time that a user perceives.

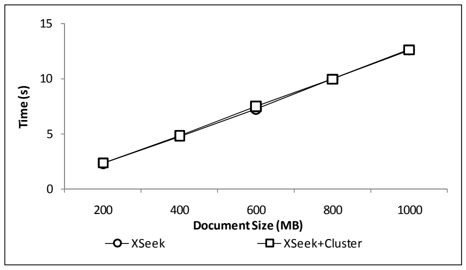
The query processing time of XSeek and XSeek+cluster is shown in Figure 9.8. The speed of XSeek+C-index is extremely slow: even on the original version of these three data sets with 10-20 results, it may take 10-20 seconds to cluster the results. Therefore, we do not include XSeek+C-index in this test as well as the scalability test.

Although XSeek+Cluster takes additional time for clustering compared to XSeek, it also saves time as it avoids outputting many irrelevant results. For QS_9 XSeek+Cluster is faster than XSeek as many irrelevant query results do not need to be output. For other ambiguous queries, the additional time taken by XSeek+Cluster is quite small compared with the query processing time. Some queries such as QS_7, QM_7 and QM_8 take longer time to process due to the low selectivity of keywords (most keywords match tag names).

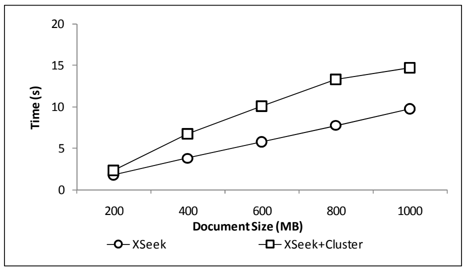
Scalability

We tested the scalability of XSeek and XSeek+Cluster on the Auction data set over three parameters: data size, query size, the depth of SLCA, and the desired number of clusters. Since the complexity and scalability of calculating SLCA were presented in [144], we only test the scalability of grouping matches, generating search results and clustering.

Increasing Data Size. The experiments were performed on the Auction data with an increasing size up to 1GB. The results for queries QA_6 (unambiguous) and QA_{10} (am-



(a) QA_6



(b) QA_{10}

Figure 9.9: Processing Time of XSeek and XSeek+Cluster with Increasing Document Size

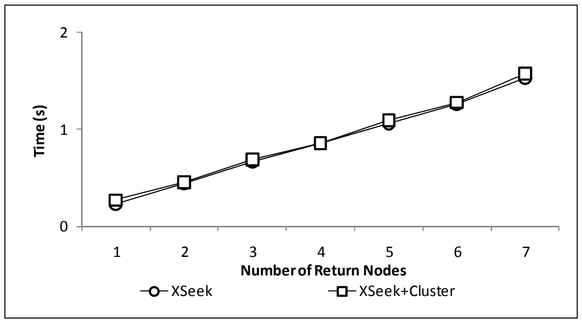


Figure 9.10: Processing Time of XSeek and XSeek+Cluster with Increasing Number of Return Nodes

biguous) are shown in Figure 9.9. Results for other queries are similar and are omitted. As we can see, the processing times of both approaches increase linearly when the data size increases. XSeek+Cluster avoids generating most results of irrelevant clusters, however, it takes additional time for clustering, thus its processing time is longer XSeek. However, their processing times are comparable.

Increasing Query Size. The experiments were performed on the Auction data of 103MB with seven queries starting from QA_5 to QA_6, increasing one keyword at each time, which means that the number of return nodes increases from 1 to 7. The result is shown in Figure 9.10. As the number of return nodes increases, the result size and therefore the result generation times of both approaches increase linearly.

Furthermore, we have tested queries with an increasing number of predicates, a constant number of return nodes and a constant depth of SLCA nodes. The test queries are constructed by replacing the node names in the queries of the previous test to their corresponding values. Since generally keywords that match values have high selectivity

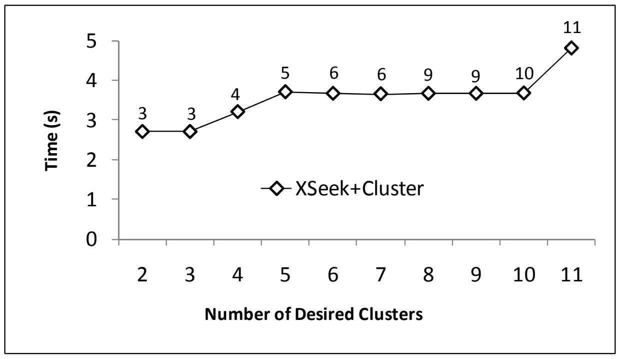


Figure 9.11: Processing Time of XSeek+Cluster with Increasing Desired Number of Clusters

(resulting in a small number of matches), the processing times of both approaches can almost be neglected, and therefore the experimental figure is omitted.

Increasing the Desired Number of Clusters. We use an ambiguous query, QA_{10}, to test the scalability of XSeek+Cluster with respect to the desired number of clusters, which is increased from 2 to 11 (the maximum number of clusters for QA_{10}). The query processing times are presented in Figure 9.11. The numbers above each point in the curve are the actual numbers of clusters generated, which may not necessarily be exactly the same as the desired number of clusters.

As we can see, the processing times are non-decreasing when increasing the desired number of clusters. Furthermore, the processing times are similar for 2-3 clusters, and 5-10 clusters. To understand this, we observe that the clustering time is dominated by the time to determine the lengths of the suffix paths used for clustering, that is, the execution time of the "while" loop (lines 5-22) in Algorithm 13. This time is proportional to the number of results and the number of keyword matches, which are large (in fact, this query has 19866 results, the total size of which is 1.4MB). On the other hand, although the DP procedure takes more time when the number of clusters increases, its processing time is proportional to the number of existing clusters and the number of desired clusters, which are much smaller than the number of results and keyword matches. Thus the running time of DP only constitutes a small portion of the clustering time.

The results of QA_{10} has 3 clusters from node categories, thus we already have

3 clusters after procedure $groupMatch$ in Algorithm 12. When the number of clusters is 4, we split the clusters according to the parents of keyword matches (i.e. suffix paths of length 2). For 5-10 clusters, we split based on the grandparents and parents of keyword matches (i.e. suffix paths of length 3), thus their processing times are similar, which has an increase compared with that of 4 clusters. Considering grandparents can only produce up to 10 clusters. Thus for 11 clusters, we need to consider the next ancestors, which leads to another increase of the processing time.

9.5 Summary

In this section we discuss a query-aware clustering scheme that clusters results according to the structures and a user-specified number of clusters so that users can browse possible semantics interpretation with desirable granularity and quickly find the set of relevant results. We first cluster the results based on the keyword roles (predicates and return nodes). If the user wants more clusters, we further split the clusters them based on the suffix paths. Generating as close as possible to a user specified number of clusters is NP-hard, and we proposed an efficient dynamic programming algorithm for this problem. Experimental evaluation verified the effectiveness and efficiency of the proposed clustering method.

Chapter 10

EFFICIENT RESULT CLUSTERING USING RESULT SNIPPETS

10.1 Motivation and Goal

As discussed in Chapter 9, clustering the query results helps address structural, keyword and user preference ambiguities. A unique challenge of result clustering for a structured search engine is that clustering must be performed online, because each result is dynamically generated, rather than a text document in text search.

With this concern in mind, a better solution for grouping structured search results is to use a small summary of each result, rather than the results themselves. We have discussed in Chapter 7 that result snippets are concise and faithful summaries of the query results, which are constructed by selecting proper instances for items in ILists. Therefore, it is a natural idea to use either the snippets or the ILists, instead of the results themselves, for grouping the query results in order to get better efficiency. Intuitively, similar search results can be expected to have similar snippets and ILists, and using snippets and ILists to group the results should have a similar quality as using results themselves.

In Section 10.2, we will report the effectiveness and efficiency of grouping query results based on their snippets or ILists. Experiments indicate that, by grouping results according to snippets or ILists, we achieve a similar quality compared with directly grouping results, and meanwhile getting significant efficiency improvement and thus greatly reduce the query response time.

Is it better to group results using snippets or ILists? In fact, their qualities and speeds are in general similar, as both are good representatives of query results and both are small. However, in certain cases using IList has some advantages over using snippets. Intuitively, if the scores of the same features differ a lot in two ILists it means the two results are quite different and should be separated into two groups, which is likely the case if clustering using ILists. However, the two snippets may be similar or even the same, making it more likely for the two results to be put into the same group using snippets.

Example 10.1 *Consider two results of query "*Texas, apparel, retailer*". Both retailers sell outwears, suits, skirts, sweaters and shirts. The detailed numbers of these merchandises as well as their scores are shown in the following table:*

	result 1		*result 2*	
	number of occurrences	*score*	*number of occurrences*	*score*
outwear	*5000*	*4.15*	*5000*	*5.00*
suit	*1000*	*0.83*	*10*	*0.10*
skirt	*10*	*0.08*	*10*	*0.10*
sweater	*10*	*0.08*	*10*	*0.10*
shirt	*10*	*0.08*	*10*	*0.10*

As we can see, these two stores sell quite different numbers of suits, and it is reasonable to differentiate them, which is the case if we use IList, as IList records the score of each prominent feature of this type. However, outwear *is the only prominent feature in both results. Assuming that the other features in the two results are similar, then they have similar snippets which means the two results have a much higher possibility of being mistakenly grouped together.*

The problem of grouping XML documents has been studied. Various grouping measures and schemes have been proposed. A number of XML grouping approaches focused on tree edit distance or its variants as the similarity measurement for different XML documents [11, 87, 47, 45, 107, 140, 139, 78, 128], and proposed corresponding grouping algorithm based on different similarities measured from tree edit distances. [87, 47, 45] proposed different methods to summarize the graph. Based on the summarization they calculate the similarity between XML documents, which is more efficient compared with using the document trees. Alternatively, some other works on XML documents grouping [52, 137, 136] proposed to use grouping algorithms based on vector space model.

Since clustering XML results needs to be done online, efficiency is very important. Therefore, we choose to use a grouping algorithm that uses the vector space approach for XML trees, as it has a lower complexity than computing tree edit distances [136]. Among the three approaches that use vector space model, [136] is the most suitable one for XML result clustering because it uses distinct root to leaf paths for clustering. [52] uses parent-child node pairs and [137] uses only tag names, which have less information for clustering. For the methods proposed in [87, 47, 45], since they do not consider the value of attributes when summarizing the XML documents, they can not be used to effectively cluster the results generated by XML keyword search, which usually come from data source with similar schema and mainly differ in values.

In a vector space model, each object is modeled as a vector with a weight on each component. Two vectors are similar if they have a large inner product. When clustering using snippets, we follow [136] which uses each distinct root-leaf path as a component whose weight is its number of occurrences. When clustering using ILists, we model each component of a vector as an item in the IList, whose weight is the feature score if it is a feature, and is 1 otherwise.

10.2 Experiments

To test the effectiveness and efficiency of grouping query results based on snippets and ILists, we implemented the clustering algorithm in [136]. If the input is a set of XML documents (query results or snippets), it converts each XML document into a corresponding vector representation such that each component of the vector is a path from root to leaf whose weight is its number of occurrences in the document, and clusters the input documents using the classical vector space model. For clustering using ILists, each item in the IList is a component in the vector representation, whose weight is its score. We compare the performances of grouping query results using results and snippets, respectively, in terms of processing time and grouping quality.

To obtain the ground truth of grouping query results, we generate multiple retailer documents according to a set of distinct patterns. For example, one pattern has the probability of the value *fitting:men* to be 75%, *women* 20% and *children* 5%. The probabilities of every possible attribute and value, such as *category:outwear*, *situation:casual*, are set. Then we generate multiple documents for each pattern with the given probability distribution. The ground truth is simply obtained by placing the retailers/stores generated from the

Queries for clustering	
Q_1	store
Q_2	retailer
Q_3	store, clothes
Q_4	retailer, apparel

Figure 10.1: Test Queries for Clustering Using Snippets or ILists

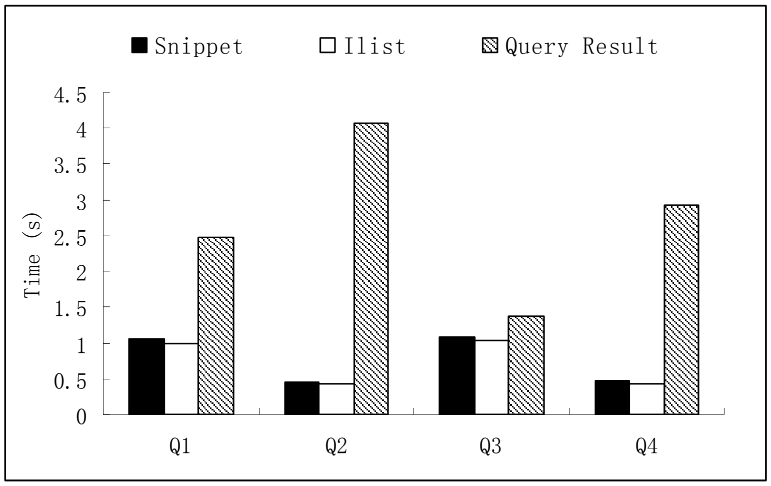


Figure 10.2: Result Clustering Time Using Query Result, Snippets and ILists

same pattern into the same group. We use the queries in Figure 10.1 for evaluating the quality and efficiency of grouping query results.

Processing Time. The processing times of clustering algorithm using the query results, snippets and ILists are shown in Figure 10.2. Query results of the queries in Figure 10.1 are generated by the underlying XML query engine, and we generated the snippets for all these query results. The query results and the corresponding snippets are used as input for the clustering method. We run the clustering method with these inputs for 10 times and report the average processing time. As we can see from Figure 10.2, the processing times of using snippets and ILists are much smaller compared with the processing time using query results, which is important. The difference between using snippets and ILists is very small, as an IList has more features than the corresponding snippet, but it does not have internal nodes in the tree structure.

Quality. In this test we evaluate the qualities, in terms of precisions and recalls, of the result groups generated using query results, snippets and ILists. As discussed earlier in

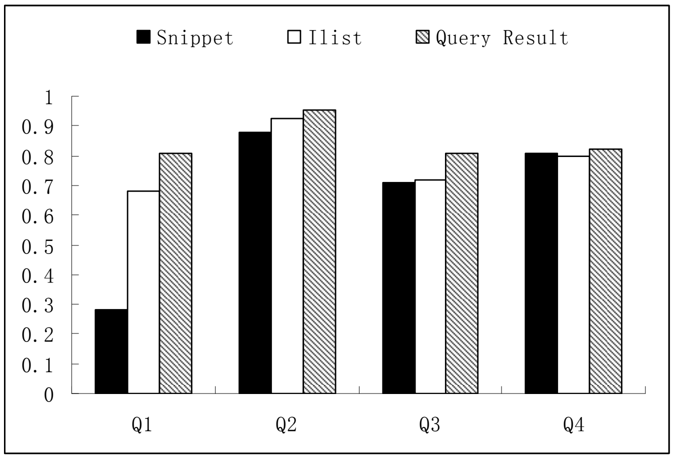


Figure 10.3: Clustering Precision Using Query Results, Snippets and ILists

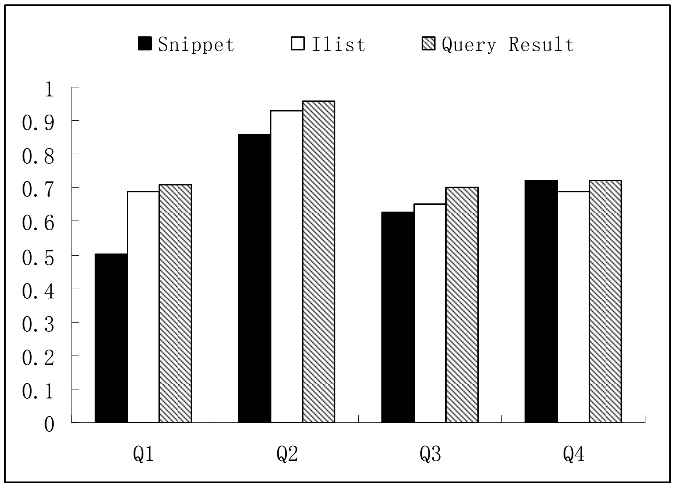


Figure 10.4: Clustering Recall Using Query Results, Snippets and ILists

this section, ground truth is obtained from the generative model. Precision is calculated by counting the number of correct placements of query results divided by the number of results in a cluster, and recall is calculated by counting the number of correct placements divided by the number of results in the ground truth. As we can see in Figure 10.3 and Figure 10.4, although a snippet or IList contains much less information than the corresponding result, the precision and recall of the clustering result using snippets and ILists are generally very close to those of using query results.

Grouping results using snippets and ILists have a similar quality except Q_1, where using IList achieves a better quality than using snippets. Since snippets do not record the accurate score of each feature and only has a subset of prominent features, they may sometimes be outperformed by ILists with respect to result clustering.

10.3 Summary

Grouping results using snippets and ILists achieves a favorable tradeoff between quality and efficiency – it achieves a much better efficiency while generally sacrificing little quality, compared with grouping results using results themselves. It also indicates that the snippets and ILists generated by eXtract are indeed concise yet good summaries of the corresponding results.

Chapter 11

GENERATING EXPANDED QUERIES FROM CLUSTERED RESULTS

11.1 Motivation and Goal

Query expansion, or query refinement, is the process of reformulating a seed query to improve retrieval performance and resolve keyword ambiguity. Web search engines typically make query suggestion based on similar and popular queries in the query log [39, 19]. To handle a bootstrap situation where the query log is not available, there are works on query result summarization [142, 28, 26, 130, 76, 119], where popular words in the results are identified and suggested to the user for query refinement. The popularity of words are typically measured by factors such as term frequency, inverse document frequency, ranking of the results in which they appear, etc.

However, existing query expansion techniques based on result summarization using popular words can not effectively handle ambiguous queries which have multiple possible interpretations of their meanings, or exploratory queries [25] where the user does not have a specific search target, but would like to navigate the space of possibly relevant answers and iteratively find the most relevant ones. The expanded queries generated by such an approach may only cover a subset of the possible query semantics, and fail to provide a classification of the results. The problem becomes especially severe when the expanded queries are generated by summarizing the top-K results, which is typically the case for efficiency reasons. One type of results may have higher ranks than other types and will suppress other result types to be reflected in the expanded queries. For instance, when searching “apple” on Google there is only one result about apple fruit in the top 30 results, whereas the rest are about Apple Inc. Since keywords about apple fruit have a small presence in these results, they are unlikely to be considered as popular words. Expanded queries generated according to popular words will bear the ranking bias and fail to cover the query semantics of searching apple fruit.

To handle ambiguous and exploratory queries, ideally query expansion should provide a *classification of different interpretations of the original query*, and thus guide the user to refine the query in order to get more results of the desirable type. For query “apple”, intuitively, “Apple Inc.” and “apple fruit” would be desirable, even though “fruit” is not a popular word in the set of top ranked results. Note that for this ambiguous query, either interpretation can be relevant to the user, although one interpretation is ranked much higher.

To generate expanded queries that provide a classification of the query results, we propose a technique in Chapter 11, which first clusters the results into k clusters[1] using one of the existing clustering methods, where k is an upper bound specified by the user. In the above example of generating expanded queries for “apple” according to the top 30 results, although there is only one result about apple fruit, since it is significantly different from others, it should comprise a cluster itself, and thus can be covered by an expanded query.

Note that although the approach discussed in Chapter 9 is able to generate a set of clusters and a description for each cluster, it may not work well for document-centric structured data or text data. For example, consider a document-centric XML about research papers. Given a query, e.g., “keyword search”, it may cluster the results according to where the keyword appears (title, abstract, index term, etc.) as well as paper category (conference paper, journal paper, etc.). However, it cannot further cluster the results based on text attributes like abstract. This is undesirable since the abstract of a paper usually contain important information about the topic of the paper. If we use a text clustering method, e.g., K-means, then although we are able to further cluster the papers, it is not obvious how to refine the queries based on the clusters. Therefore in this approach we would like to find a generic way of generating expanded queries given a set of clusters, where the clusters can be obtained using any approach on any type of data.

Given a set of clusters of query results, the challenge is how to generate an expanded query for each cluster, whose set of results is as close to the cluster as possible. We assume that a result of a query is obtained by finding the data unit that contains all the query keywords. If we consider a cluster of results as the ground truth, our goal is then to generate a query whose set of results achieve both a high precision and a high recall. This

[1]We can either cluster the set of all query results or a set of top ranked results.

is a difficult problem as the expanded queries should not only be selective to eliminate as many results in other clusters as possible (maximizing the precision), but also be general to retrieve as many results in this cluster as possible (maximizing the recall). One intuitive approach would apply existing works on cluster labeling / summarization [27, 105] to find the popular words in each cluster, and then use these words as the query for the cluster. However, the set of results retrieved by such a query would unlikely be similar to the original cluster. For example, consider 5 keywords, each appearing in 80% of the results in a cluster, but they do not co-exist in any result. A cluster labeling approach may output these 5 keywords as the label of the cluster. Nevertheless, using these 5 keywords as an expanded query will yield no result under AND semantics. This illustrates a unique challenge in generating queries for clustered results: the *interaction* of the keywords must be considered. Moreover, a potentially large number of results, and a large number of distinct keywords in the results add further challenges to the problem. Exhaustively searching for the optimal query for each cluster will be prohibitively expensive in practice. We formally define the problem of generating an optimal set of queries given the ground truth of each query's results. We show that this problem is NP-hard, and also APX-hard (i.e., it does not have a constant approximation).

To tackle the challenges, we propose two efficient algorithms which are presented in this chapter. The first algorithm, named iterative single-keyword refinement (ISKR), iteratively refines a query in a greedy fashion by adding or removing a keyword to improve the quality of the query. The technical challenge is to dynamically and efficiently select the promising keywords to add to/remove from the current query. The second algorithm, named partial elimination based convergence (PEBC), attempts to find the best tradeoff between precision and recall using a randomized procedure. Specifically, given a set of sample queries and their F-measures, we find the two adjacent queries with the highest average F-measure, and iteratively test more points between them in search of an improved F-measure. Since the space of all possible queries is exponential to the data size, the technical challenge is how to efficiently find effective sample queries. We identify that this problem bears some similarity with the weighted partial set cover problem, but with fundamental differences that demand novel solutions. Compared to ISKR, PEBC in most cases favors more on the efficiency compared with quality, as shown in the experiments. Besides, when the results have ranking scores, both algorithms take the ranking scores into consideration by prioritizing the results with higher ranks when generating expanded queries.

11.2 Problem Definition

In this chapter, we consider keyword queries on either text documents or structured data. A text document is modeled as a set of words, and a structured document is modeled as a set of features defined as (entity:attribute:value) triplets [65], such as product:name:iPad.

Each result is a text document or a fragment of a structured document that contains *all* the keywords in the query.

The goal of this work is to generate a set of expanded queries that provides a classification of possible interpretations of the original user query. The input that we take includes a user query, and a set of clustered query results where the results are optionally ranked. Note that result clustering can be done using any existing clustering method (such as k-means), which is not the focus of this work. The output is one expanded query for each cluster of results, which maximally retrieves the results in the cluster, and minimally retrieves the results not in the cluster.

We now formally define the optimization goal. Considering the cluster as the ground truth, the quality of an expanded query can be measured using precision, recall and F-measure. Precision measures the correctness of the retrieved results, recall measures the completeness of the results, and F-measure is the harmonic mean of them. Let $C_1, \cdots, C_k$ denote the result clusters, q_i denote the query generated for cluster C_i $(1 \leq i \leq k)$, $R(q_i)$ denote the set of results of q_i. The precision, recall and F-measure of q_i are computed as

$$precision(q_i) = \frac{R(q_i) \cap C_i}{R(q_i)}, \quad recall(q_i) = \frac{R(q_i) \cap C_i}{C_i}$$

$$Fmeasure(q_i) = \frac{2 \times precision(q_i) \times recall(q_i)}{precision(q_i) + recall(q_i)}$$

To handle the general case where results are ranked, we use a weighted version of precision and recall. Let $S(\cdot)$ denote the total ranking score of a set of results, then

$$precision(q_i) = \frac{S(R(q_i) \cap C_i)}{S(R(q_i))}, \quad recall(q_i) = \frac{S(R(q_i) \cap C_i)}{S(C_i)}$$

The optimization goal is measured by the overall quality of the set of expanded queries (one for each cluster). We use the harmonic mean of their F-measures, whereas other aggregation functions (e.g., algebraic mean) can also be used.

$$score(q_1, \cdots, q_k) = \frac{n}{\frac{1}{Fmeasure(q_1)} + \cdots + \frac{1}{Fmeasure(q_k)}} \tag{11.1}$$

To summarize, the problem of generating expanded queries based on clustered results is defined as follows.

Definition 11.1 *Given a set of clusters of query results, $C_1, \cdots, C_k$, retrieved by a user query under AND semantics, the* Query Expansion with Clusters *problem (QEC) is to find a set of queries, one for each cluster, such that their score (Eq. 11.1) is maximized.*

Theorem 11.1 *The QEC problem is NP-hard.*

Proof. We reduce the vertex cover problem to the QEC problem. Recall that the vertex cover problem selects the minimum set of nodes in an undirected graph, such that every edge has at least one of its endpoints selected. Given any instance of the vertex cover problem: an undirected graph $G(V, E)$, where each node has at least one edge (otherwise this node does not need to be considered for vertex cover), let $n = |V|$, $m = |E|$, we create an instance of the QEC problem as follows.

1. *Create $n + 1$ keywords, $k_0, \cdots, k_n$, where each keyword $k_i (1 \leq i \leq n)$ corresponds to a node i in graph G.*

2. *Create one cluster C_1 with n results. Let keyword $k_i (1 \leq i \leq n)$ appear in all the results in C_1 except the ith result.*

3. *Create cluster C_2 with $4kn^2m$ results, which are evenly partitioned to m groups, such that each group corresponds to an edge in graph G and has $4kn^2$ results.*

   For each node $i \in V$, if it has x edges, then let keyword k_i not *appear in the corresponding x groups of results in C_2. For example, if node i has edges e_1, e_3, e_6, then k_i does* not *appear in results in groups 1, 3, 6, in C_2, but appears in all other results in C_2.*

4. *Let keyword k_0 appear in all results in C_2, but none of the results in C_1.*

Clearly the construction can be performed in polynomial time. For this instance of QEC, it is easy to see that the optimal query for C_2 is $q_2 = \{k_0\}$, and $F - measure(q_2) = 1$. Let q_1 be the optimal query for C_1. Next we prove that $precision(q_1)$ must be 1.

Suppose we have $precision(q_1) = 1$, that is, q_1 eliminates all the results in C_2. Next we show that $recall(q_1) \geq \frac{1}{|C_1|} = \frac{1}{n}$. To see this, q_1 must not contain all keywords $k_i (1 \leq i \leq n)$, because any group in C_2 can be eliminated by two keywords (according to the construction of the QEC instance, each group corresponds to an edge in the graph, which can be eliminated by the keyword corresponding to either endpoint of the edge), and there must be at least one group that does not need both keywords included in q_1. Therefore, q_1 should at least retrieve one result in C_1, and $recall(q_1) \geq \frac{1}{|C_1|} = \frac{1}{n}$; $F - measure(q_1) \geq \frac{2}{n+1}$, hence

$$S_1 = score(q_1, q_2) \geq \frac{2}{\frac{n+1}{2} + 1} = \frac{4}{n+3}$$

.

Suppose otherwise, $precision(q_1) \neq 1$, q_1 must retrieve some results in C_2. Recall that the results in C_2 has m groups, each with $4kn^2$ identical results, q_1 must retrieve at least $4kn^2$ results in C_2, thus:

$$precision(q_1) < \frac{n}{4kn^2} = \frac{1}{4kn}$$
$$F - measure(q_1) < \frac{2}{4kn + 1}$$

$$S_2 = score(q_1, q_2) < \frac{2}{\frac{4kn+1}{2} + 1} = \frac{4}{4kn+3}$$

Since $S_2 \leq \frac{4}{4kn+3} < \frac{4}{n+3} \leq S_1$, S_2 cannot be the optimal score of q_1 and q_2, thus $precision(q_1)$ must be 1. Furthermore, the optimal q_1 must use the minimum number of keywords to eliminate all results in C_2, since the more keywords q_1 uses, the less results it can retrieve in C_1, the lower F-measure.

Now we show that the optimal solution of vertex cover corresponds to the optimal solution of QEC problem. In the vertex cover instance, if we take the nodes that correspond to the keywords in q_1, we get the minimal vertex cover of G. To see that, if an edge $e_i(n_a, n_b)$ is not covered (i.e., neither k_a nor k_b is selected in q_1), then since the results in group i in C_2 only miss keywords k_a and k_b, these results are retrieved by q_1, which is contradictory with $precision(q_1) = 1$. On the other hand, suppose that we have a minimal vertex cover of G, in which each edge is covered by at least one of its endpoints. If we take the corresponding keywords to compose q_1, then every group in C_2 will be eliminated. Since the number of nodes in vertex cover is minimized, the number of keywords in q_1 is minimized, the number of results retrieved in C_1 is maximized, giving an optimal F-measure of q_1, and hence an optimal score of q_1 and q_2. Therefore, the QEC problem is NP-hard.

Although the vertex cover problem has a simple 2-approximation algorithm and QEC can be reduced from the vertex cover problem, the following theorem shows that the QEC problem does not have a constant polynomial-time approximation, i.e., it is APX-hard.

Theorem 11.2 *The QEC problem is APX-hard.*

Proof. We prove that the QEC problem is as hard as the independent set problem in terms of approximation. Recall that the independent set finds the maximum set of nodes in an undirected graph, such that no two nodes are connected by an edge. The independent set problem has been proved to be APX-hard, i.e., it has no constant approximation ratio [53]. We will prove that, if the QEC problem has an approximation ratio of k, then the independent set problem has an approximation ratio of $4k - 3$.

Given any instance of the independent set problem: an undirected graph $G(V, E)$, let $n = |V|$, $m = |E|$, we create an instance of the QEC problem in the same way as stated in the proof of Theorem 11.1.

We first show that, for any arbitrary k-approximate solution for this QEC instance, consisting of queries q'_1 and q'_2, we must have $q'_2 = \{k_0\}$, $precision(q'_1) = 1$. This is because

$$S_2 \times k \leq \frac{4k}{4kn+3} < \frac{4k}{kn+3k} = \frac{4}{n+3} \leq S_1 = OPT$$

where S_1 and S_2 are defined in the proof of Theorem 11.1, and OPT is the score of the optimal solution. Unless $precision$
$(q'_1) = 1$, it cannot approximate the optimal solution within k.

Suppose we have an algorithm that can give a k-approximate solution to the above QEC instance in polynomial time, now we illustrate how to obtain an approximate solution to the independent set problem with ratio $4k - 3$. Let R, F and S denote $recall(q_1)$, $F\!-\!measure(q_1)$ and $score(q_1, q_2)$ in the optimal solution, and R', F' and S' denote the corresponding values in the approximate solution. From $k \times S' \geq S$, $S = \frac{2F}{1+F}$ and $S' = \frac{2F'}{1+F'}$, we have $\frac{F'}{F} \geq \frac{1}{2k-1}$.

Since $F = \frac{2R}{1+R}$ and $F' = \frac{2R'}{1+R'}$, we obtain

$$\frac{R'}{R} \geq \frac{1}{4k-3} \tag{11.2}$$

Note that in the optimal solution, q_1 is a query which eliminates all results in C_2 using the minimum number of keywords. Let the number of keywords in q_1 be P. Since each keyword in q_1 eliminates a result in C_1, q_1 retrieves $n - P$ results in C_1, thus $R = \frac{n-P}{n}$. Similarly, in the approximate solution, let the number of keywords in q'_1 be P', then $R' = \frac{n-P'}{n}$. According to Equation 11.2, we have

$$\frac{n-P'}{n-P} \geq \frac{1}{4k-3} \tag{11.3}$$

Now let us look at the independent set instance. Recall that each node in G corresponds to a keyword in C_1 in the QEC instance. In the proof of Theorem 11.1, we have shown that the nodes corresponding to the keywords in P comprise the minimal vertex cover of G. Therefore, the set of nodes corresponding to the keywords not *in P comprises the maximal independent set of G, whose size is $n - P$. Similarly, the set of nodes corresponding to the keywords* not *in P' comprises an approximate independent set of G, whose size is $n - P'$. According to Equation 11.3, we have obtained a $4k - 3$ approximate solution for this independent set instance. Since this is an arbitrary independent set instance, it contradicts with the fact that independent set instance is APX-hard. Therefore, the QEC problem is APX-hard.*

In the next two sections we discuss the algorithms for query generation. Note that maximizing the overall score (Eq. 11.1) is equivalent as maximizing the F-measure of each query, thus each query can be generated independently. Specifically, the algorithms solve the following problem.

Definition 11.2 *Given a user query Q, a cluster C of results, and the set of results U in all other clusters, as well as an optional ranking score of each result, the problem is to generate a query q, whose F-measure with C as the ground truth is maximized.*

11.3 Iterative Single-Keyword Refinement

The first algorithm we introduce is named *Iterative Single-Keyword Refinement* (ISKR). Given the user query and a cluster of results, the ISKR algorithm iteratively refines the input query until it cannot further refine the query to improve the F-measure of the query result (considering the cluster as the ground truth). Then, it outputs the refined query as the expanded query for the cluster. Specifically, it quantifies a value of each keyword appearing in the results, and refine the query by choosing the keyword with the highest value in each iteration. Several challenges need to be resolved for this approach to work: (1) How should we quantify and compute the value of a keyword? (2) As discussed in Section 11.1, keywords interact with each other when adding them to be part of a query. After the candidate query is refined, the value of a keyword may be changed. How should we identify the keywords whose values are affected and update the values of these keywords? (3) We start with the original user query, and try to add new keywords in the order of their values to this query to form an expanded query. Are there any case that a previously added keyword should be removed in order to improve the F-measure of the expanded query? (4) Since there can be a potentially large number of results and a large number of distinct keywords in a result, it is time-consuming to find the best set of keywords to add to the original query. How can we ensure efficiency? Next we will present ISKR algorithm, whose pseudo code can be found in Algorithm 14, that addresses these challenges.

Value of a Keyword. We first need to define the value based on which we choose the best keyword to add to or remove from q at each step. When adding a keyword to a query q, the F-measure achieved by q may either increase or decrease. Thus naturally, the value of a keyword should be measured by the delta F-measure of query q after adding this keyword. But a disadvantage of this value function is that the values of the keywords are hard to maintain. The set of query results $R(q)$ is dynamically determined, based on the keywords that are already added to q. Since precision, recall, and thus F-measure are defined based on $R(q)$, the value of every keyword needs to be dynamically computed, and updated after every change to q.

To efficiently measure the values of keywords, we have the following observations. First, when adding a keyword k to q, the positive effect is that q may retrieve less results in U (thus improving precision), and the negative effect is that q may retrieve less results in C (thus decreasing recall). Thus the number of results eliminated from U and C can be used to indicate whether it is good to add keyword k to query q. Second, it is more efficient to maintain the number of results eliminated from U and C by adding a keyword k than to maintain the delta F-measure of a keyword.

To see this, in the following, we use *delta results* of a keyword k with respect to query q (or simply delta results, if k and q are obvious) to denote the set of results retrieved by q, but not retrieved after adding k to q. After adding k to q, let D denote the set of delta results. Consider a keyword k' which appears in all results in D, i.e., it cannot eliminate any result in D. Note that the delta results of k' with respect to query q depends on how many results of q can be eliminated by adding k' to q. Since k' cannot eliminate any result in D anyway, the delta results of k' with respect to q are the same as the delta results of k' with respect to $q \cup \{k\}$. This means that the delta results of k' are not affected after adding k to q.

With these observations, we measure the value of a keyword by benefit and cost. $benefit(k, q)$ is the total ranking score of the results eliminated in U by adding k to q, and $cost(k, q)$ is the total score of the results eliminated in C by adding k to q. Thus

$$benefit(k, q) = S(R(q) \cap U \cap E(k))$$

$$cost(k, q) = S(R(q) \cap C \cap E(k))$$

where $E(k)$ is the set of results that do not have keyword k (hence will not be retrieved by any query that contains k).

We define the value of a keyword with respect to q as its benefit-cost ratio, as commonly adopted in cost-benefit analysis:

$$value(k, q) = \frac{benefit(k, q)}{cost(k, q)} \tag{11.4}$$

We consider $value(k, q)$ as zero if both $benefit(k, q)$ and $cost(k, q)$ are zero.

Identifying Keywords with Affected Values. When we add a keyword to query q, the benefits and costs of other keywords may be affected. As discussed before, the value of a keyword is affected if and only if this keyword does not appear in at least one of the delta results. For each such keyword, we re-compute its benefit, cost and value using Eq. 11.4.

Example 11.1 *We use this example to illustrate the ISKR algorithm. Suppose the original query is "apple". Consider a cluster C with 8 results, $R_1, \cdots, R_8$, and U, the set of results that is not in C, with 10 results, $R'_1, \cdots, R'_{10}$. We consider 4 keywords for query expansion.*

The following table shows the keywords, and the results in C and U that each keyword can eliminate*.*

	k_i	$E(k_i) \cap C$	$E(k_i) \cap U$
$i = 1$	*job*	$R_1, \cdots, R_6$	$R'_1, \cdots, R'_8$
$i = 2$	*store*	$R_1, \cdots, R_4$	$R'_1, \cdots, R'_4$, R'_9
$i = 3$	*location*	$R_2, \cdots, R_5$	$R'_5, \cdots, R'_8$, R'_{10}
$i = 4$	*fruit*	$R_1, \cdots, R_3$	$R'_2, \cdots, R'_4$

The initial benefit, cost and value of each keyword are:

keyword	*benefit*	*cost*	*value*
job	*8*	*6*	*1.33*
store	*5*	*4*	*1.25*
location	*5*	*4*	*1.25*
fruit	*3*	*3*	*1.00*

Since keyword job *has the largest value, we first add* job *into q; so $q = \{$apple, job$\}$. Now q retrieves 2 results in C: R_7 and R_8, and 2 results in U: R'_9 and R'_{10}.*

Now we need to update the benefit, cost and value of each affected keyword. For example, the benefit of store *becomes 1, since adding it to q can further eliminate one result in U: R'_9. The cost of* store *becomes 0, since it does not eliminate any result in C, as both results (R_7 and R_8) contain* store*. The updated benefit, cost, and value of each keyword is shown in the following table (the row for* job *shows the benefit, cost and value of removing* job *from the current query, which will be discussed later).*

keyword	*benefit*	*cost*	*value*
job	*6*	*8*	*0.75*
store	*1*	*0*	∞
location	*1*	*0*	∞
fruit	*0*	*0*	*0*

Thus we add store *to q. After updating the benefit, cost, and value of the affected keywords, we further add keyword* location *to q. At this time, the only remaining keyword,* fruit*, has a value of 0, thus we do not further add keywords to the expanded query.*

Necessity of Keyword Removal. Since keywords added to the query may have

Algorithm 14 Iterative Single-Keyword Refinement

```
ISKR (User Query: $uq$, Cluster: $C$, Results not in $C$: $U$)
 1: $\mathcal{K}$ = the set of keywords in $C \cup U$
 2: $q = uq$
 3: $Refine(C, U, \mathcal{K}, q, weight)$
 4: return $q$
Refine ($C, U, \mathcal{K}, q, weight)$)
 1: $T = \emptyset$
 2: for each $k \in \mathcal{K}$, $k \notin q$ do
 3:     $E(k)$ = the set of results that do not contain $k$
 4:     $benefit(k) = S(R(q) \cap C \cap E(k))$
 5:     $cost(k) = S(R(q) \cap U \cap E(k))$
 6:     $value(k) = benefit(k)/cost(k)$
 7:     insert $k$ into $T$
 8: for each $k \in \mathcal{K}$, $k \in q$ do
 9:     $D(k) = R(q\backslash k)\backslash R(q)$
10:     $benefit(k) = S(D(k) \cap C)$
11:     $cost(k) = S(D(k) \cap U)$
12: while true do
13:     $k$ = top-1 keyword in $T$
14:     if $value(k) \leq 1$ then
15:         break
16:     if $k \in q$ then
17:         $q = q\backslash k$
18:         $MaintainT(T, q, k, E(k), \mathcal{K}, C$, remove)
19:     else
20:         $q = q \cup k$
21:         $MaintainT(T, q, k, E(k), \mathcal{K}, C$, add)
22: return $q$
MaintainT ($T, q, k, E(k), \mathcal{K}, C, type$)
 1: if $type$ = add then
 2:     $deltaResult = R(q\backslash k) \cap E(k)$
 3: else
 4:     $deltaResult = R(q\backslash k)\backslash R(q)$
 5: for each $k' \in \mathcal{K}$ do
 6:     if each $k'$ appears in all results in $deltaResult$ then
 7:         continue
 8:     if type = add then
 9:         $benefit(k') = R(q) \cap U \cap E(k')$
10:         $cost(k') = R(q) \cap C \cap E(k')$
11:     else
12:         $D(k) = R(q\backslash k)\backslash R(q)$
13:         $benefit(k') = D(k) \cap C$
14:         $cost(k') = D(k) \cap U$
15:     remove $k'$ from $T$
16:     $value(k') = benefit(k')/cost(k')$
17:     add $k'$ to $T$
```

complex interactions, it may be beneficial to remove a keyword from q that was added to q earlier, as shown in the following example.

Example 11.2 *Continuing Example 11.1. Note that keyword* job *was added into q at the first step due to its highest value, but after adding* store *and* location *to q, it becomes beneficial to remove* job*, which increases the recall but does not affect the precision. Indeed, the current $q = \{$apple, job, store, location$\}$ retrieves 2 results in C: R_7, R_8, and 0 result in U. If we now remove* job *from q, then q will retrieve 1 more result in C: R_6, but still retrieve 0 result in U. Therefore, we should remove* job *from q at this point.*

When removing a keyword $k \in q$ from q, the benefit, cost and value can be computed in a similar way. In contrast to keyword addition, removing k from q increases the results retrieved by q in both C and U, thus it may decrease the precision (measured by cost) and increase the recall (measured by benefit). For the removal case, the benefit and cost of k with respect to q are computed as

$$benefit(k, q) = S(C \cap D(k)), \quad cost(k, q) = S(U \cap D(k))$$

where $D(k)$ is the delta results after the removal of keyword k. $value(k, q)$ is still the benefit-cost ratio (Eq. 11.4).

Similar as adding a keyword, after removing a keyword, the values of other keywords may be affected. It is easy to see that the affected keywords are also those that do not appear in at least one of the delta results. For these keywords, we recompute their benefits, costs and values.

The ISKR algorithm stops when the query cannot be further improved by adding or removing a keyword, which is the case if the value of the best keyword is less than 1. In the running example, after updating the table, we find that no keyword has a value greater than 1, thus we stop and output the current query, $q = \{$apple, store, location$\}$.

11.4 Partial Elimination Based Convergence

The ISKR algorithm iteratively attempts to add/remove a keyword to/from q, during which process the values of many keywords may change and need to be updated, which incurs a potentially high processing cost. In this section we propose a convergence based algorithm for query expansion named *Partial Elimination Based Convergence* (PEBC). It approaches the optimal solution in a fast and adjustable progress. Considering F-measure as a function over q, our goal is to find the value of q that achieves the maximal value of F-measure. However, since the functional relationship between F-measure and q is unknown, and the space of all possible queries is exponential to the data size, finding the optimal value is very challenging.

We propose algorithms that select several sample queries in the search space, and iteratively test more queries between the promising sample queries toward an improved F-measure. Specifically, given a set of queries and their F-measures, we find the two adjacent ones with the highest average F-measure, and test more points between them in search of an improved F-measure. The iteration continues until the expanded query is good enough, or enough iterations have been performed. The idea of this method is related to interpolation in numerical analysis, however, we do not infer the actual F-measure function from the sampled data points due to its high complexity.

Two questions must be resolved. (1) What type of sample queries we should use to converge to the optimal solution? (2) How can we obtain such sample queries?

Type of sample queries. To answer the first question, we propose to use a set of sample queries, each of which maximizes the number of results to be retrieved in C, given a percentage of results in U to be eliminated. This is in the spirit of maximizing the recall given a fixed precision.[2] If we don't have the ranking scores of the results, we aim at eliminating x% of U's results; otherwise, we aim at eliminating a set of U's results, such that their total ranking score is x% of the total ranking score of all the results in U. In the

[2]Alternatively, we can choose sample queries that maximize the number of results to be eliminated in U given a percentage of results in C to be retrieved.

following, we use “x% of the results in U” to refer to both cases.

Example 11.3 *Suppose we generate five queries, q_1 to q_5, to eliminate 0%, 25%, 50%, 75% and 100% of the results in U, respectively, and maximize the number of results in C to be retrieved. We compute the F-measures of these queries, and suppose they are: 0.5, 0.6, 0.4, 0.8, 0.1, respectively. Note that the F-measures of these queries may not have an obvious relationship. We take the two adjacent queries whose average F-measure is the highest, which are q_3 and q_4. We zoom in the interval between them, further dividing them to several intervals, and repeat the process.*

Generating Sample Queries. The key challenge of the PEBC algorithm is: given a percentage x of results in U to be eliminated, how can we generate query q that eliminates roughly x% of the results in U, and maximizes the number of retrieved results in C? We refer to this problem as *partial elimination*.

This problem bears some similarity with the weighted partial set cover problem, which aims at using a set of subsets with the lowest total weight to cover at least x% of the elements in the universal set. However, in contrast to the partial weighted set cover problem which requires to cover *at least x%* of the elements, our goal is to eliminate *as close to x% of the elements as possible*. This ensures that we can test data points that have roughly uniform distances between each other to better gauge the F-measure function. In the next subsections, we discuss how to address this new challenge and generate queries to achieve partial elimination.

Keyword Selection Based on Benefit/Cost

One intuitive method is to apply the greedy algorithm commonly used in weighted set cover for keyword selection: each time, we select the keyword with the largest benefit/cost ratio, until we have approximately x% of the results in U eliminated. Benefit and cost are defined in the same way as in ISKR: benefit is the total weight of the un-eliminated results in U that a keyword can eliminate, and cost is the total weight of the un-eliminated results in C that a keyword can eliminate.

However, this method has an inherent problem that makes it infeasible: since the benefit/cost ratios of the keywords do not change with varying x, the keywords are always selected in the same order. Specifically, let the list of keywords selected when $x = 100$ be $\mathcal{K} = k_1, \cdots, k_p$. Now we want to select keywords to generate a query for each point in a range of possible values of x. No matter which point it is, the set of keywords selected will be a prefix of $\mathcal{K}$. Such a "fixed-order" selection of keywords makes it very difficult to control the percentage of results being eliminated.

Algorithm 15 Partial Elimination Based Convergence

```
PEBC (User Query: $uq$, Cluster: $C$, Results not in $C$: $U$)
 1: $\mathcal{K}$ = the set of keywords in $C \cup U$
 2: $q = uq$
 3: $Converge(C, U, \mathcal{K}, q)$
 4: return $q$
Converge $(C, U, \mathcal{K}, q)$)
 1: $nseg$ = 5 {set the number of segments to split the interval}
 2: $nit$ = 5 {set the number of iterations}
 3: $left$ = 0, $right$ = 100, $step = (right - left)/nseg$
 4: for $i$=1 to $nit$ do
 5:   for $x = left; x \leq right; x+ = step$ do
 6:     $currC = C$, $currU = U$
 7:     repeat
 8:       $r$ = a randomly selected result
 9:       $bestvalue$ = 0
10:       for each distinct keyword $k \notin r$ do
11:         $E(k)$ = the set of results that do not contain $k$
12:         $benefit(k) = E(k) \cap U$
13:         $cost(k) = E(k) \cap C$
14:         $value(k) = benefit(k)/cost(k)$
15:         if $value(k) > bestvalue$ then
16:           $selecetd = k, bestvalue = value(k)$
17:       $q = q \cup selected$
18:       $currC = C \backslash E(k)$, $currU = U \backslash E(k)$
19:     until roughly $x$% percent of results in $U$ are eliminated
20:   $left, right$ = the interval with the largest average score
```

Example 11.4 *Consider a total of 10 results in U, $R_1, \cdots, R_{10}$, and 4 keywords: k_1=*job*, k_2=*store*, k_3=*location*, k_4=*fruit*. Suppose the set of results eliminated in U by each keyword (benefit) and the number of results eliminated in C by each keywords (cost) are:*

$benefit(k_1) = 4(\{R_1, R_2, R_3, R_4\})$, $cost(k_1) = 2$

$benefit(k_2) = 6(\{R_5, R_6, R_7, R_8, R_9, R_{10}\})$, $cost(k_2) = 6$

$benefit(k_3) = 3(\{R_3, R_4, R_8\})$, $cost(k_3) = 1$

$benefit(k_4) = 4(\{R_4, R_5, R_6, R_7\})$, $cost(k_4) = 4$

Also suppose that the set of results in C that is eliminated by a keyword does not intersect with the set eliminated by another keyword.

In this approach, the keywords are always selected in the decreasing order of their benefit/cost ratio, that is: $k_3 \rightarrow k_1 \rightarrow k_2 \rightarrow k_4$ (recall that after a keyword is selected, the benefit/cost of other keywords may change, as discussed in Section 11.3). Having the order of keyword selection fixed, there is a slim chance to achieve the goal of x% elimination. For instance, in order to eliminate 7 results with the fixed order keyword selection, we will have to either use {k_3, k_1} which eliminates 5 results, or {k_3, k_1, k_2} eliminating all 10 results. This poses a lot of restriction. Note that in this example, if we do not select keywords in this order, we can choose {k_1, k_4} which eliminates exactly 7 results.

As we can see, always selecting keywords based on their benefit/cost ratio makes it hard to eliminate a given percentage of the results. Next we discuss the approaches that overcome this problem using a randomized procedure.

Keyword Selection Based on a Selected Subset of Results

Since selecting keywords in a fixed order is undesirable, we propose to introduce a randomized procedure. First, we randomly select a subset of x% of the results in U. Then, we select the keywords, aiming at eliminating these randomly selected results. In this way, since the set of results to be eliminated is randomly selected, we will not select the keywords in a fixed order. If the randomly selected set of results is "good", we may be able to eliminate exactly this set of results.

Given the randomly selected results, selecting a set of keywords that eliminate these results with minimal cost is NP-hard, as the weighted set cover problem is a special case of it. To see this, assume that each keyword eliminates part of the selected set of results in U, and their costs are independent (i.e., they eliminates distinct sets of elements in C). Then, each keyword is equivalent to a subset in the weighted set cover problem. To choose a set of keywords that covers the randomly selected results, we can use some greedy approaches, e.g., let S be the randomly selected set of results, at each time we choose a keyword which covers the most number of results in S with minimal cost. Other methods can also be used.

Example 11.5 *Continuing Example 11.4, suppose that we want to eliminate 7 results and the subset selected randomly is* $\{R_1, R_2, R_3, R_4, R_5, R_6, R_7\}$*. Given this set of results, we first update the benefits and costs of the four keywords. Keyword* k_1 *is not affected, as all four results it eliminates are selected. For* k_2*, we need to decrease its benefit by 3 because* R_8*,* R_9 *and* R_{10} *are not selected, and increase its cost by 3. For* k_3*, we decrease its benefit and increase its cost by 1.* k_4 *is not affected. In this case, we can select {*k_1*,* k_4*} which exactly eliminates this set of results.*

However, if the randomly chosen subset is $\{R_1, R_2, R_3, R_4, R_8, R_9, R_{10}\}$*, then the best we can do is: either using* k_1 *eliminating 4 of them, or using {*k_1*,* k_2*} eliminating all 10 results.*

As we can see, this approach has two problems. First of all, given a set of randomly selected results, selecting a set of keywords that eliminate exactly this set of results with minimal cost is an NP-hard problem. Second, as illustrated in the above example, the quality of the algorithm highly depends on the selected subset, thus the chance that it can get the optimal answer is still slim.

Keyword Selection Based on a Selected Result

Both approaches discussed before put high restrictions on keyword selection, and thus generally suffer a low quality. We propose another randomized procedure that has a much better chance to eliminate as close to x% of the results in U as possible. Instead of randomly selecting a subset of results, we randomly select *one* result in U that is not eliminated yet, and then select a keyword that (1) can eliminate the selected result, (2) has the highest benefit cost ratio over all such keywords. In case of a tie, we choose the keyword that eliminates fewer results to minimize the risk that we eliminate too many results. If the percentage of the eliminated results is smaller than x%, we continue the procedure; otherwise we stop and determine whether to include the last selected keyword based on which percentage is closer to x%. Compared with the approach presented in Section 11.4, this one has a better chance of approaching the desired percentage, x%, because selecting one result correctly is much easier than selecting a set of results correctly, as shown in Example 11.6.

Example 11.6 *Continuing the example, to eliminate all 7 results, we may get the correct solution if we first choose one of the following five results: R_1, R_2, R_5, R_6 or R_7. Suppose that we choose R_5, and choose k_4 to eliminate it. After k_4 is used, we have the set {R_4, R_5, R_6, R_7} eliminated. Then we can get optimal solution if the next randomly selected result is either R_1 or R_2. To eliminate R_1 or R_2, we choose k_1, which additionally eliminates results {R_1, R_2, R_3}, totaling 7 results eliminated. As we can see, the approach has a much higher chance to achieve the optimal solution (i.e. removing x% of results) than the ones discussed before.*

The pseudo code of the PEBC algorithm is shown in Algorithm 15.

11.5 Experiments

In this section we report a set of experimental evaluations on the quality of the expanded queries generated by our approach, and the efficiency and scalability of query generation.

Experimental Setup

Environment. All experiments were performed on a machine with AMD Atholon 64 X2 Dual Core Processor 6000+ CPU with 3GHz, 4GB RAM, running Windows Server 2008.

Data Set. We tested our approaches on two data sets: shopping and Wikipedia. Shopping is a data set that contains information of electronic products crawled from circuitcity.com. Each product has a title, a category, and a set of features. Wikipedia is a collection of document-centric XML files used in INEX 2009.[3]

Query Set and Result Clustering. We tested 10 queries on each data set, as shown in Table 11.1. The queries on Wikipedia dataset are composed of ambiguous words.

[3]http://www.inex.otago.ac.nz/

Wikipedia	
QW_1	San Jose
QW_2	Columbia
QW_3	CVS
QW_4	Domino
QW_5	Eclipse
QW_6	Java
QW_7	Cell
QW_8	Rockets
QW_9	Mouse
QW_{10}	sportsman, Williams
Shopping	
QS_1	Canon Products
QS_2	Networking Products
QS_3	Networking Products Routers
QS_4	TV
QS_5	TV Plasma
QS_6	HP Products
QS_7	Memory
QS_8	Memory 8GB
QS_9	Memory Internal
QS_{10}	Printer

Table 11.1: Data and Query Sets for Testing Query Expansion

The queries on shopping dataset are to search for specific products. We adopt k-means for result clustering. Each result is modeled as a vector whose components are features in the results and the weight of each component is the TF of the feature. The similarity of two results is the cosine similarity of the vectors.

Comparison Systems. We compared the proposed ISKR and PEBC algorithms with several representative query expansion methods:

(1) *Data Clouds* [76], which takes a set of ranked results, and returns the top-k important words in the results. The importance of a word is measured by its term frequency in the results it appears, inverse document frequency, as well as the ranking score of the results that contain the word. Data Clouds is a representative method for returning important words in the search results, without clustering the results.

(2) *Google*. For each test query, we take the first 3-5 related queries suggested by Google (the number of which is the same as the number of queries generated by other approaches). Google is a representative work of suggesting related queries using query logs.

(3) *CS*, representing Cluster Summarization [27]. It first clusters the results, then generates

a label for each cluster. The label of a cluster is selected based on the term frequency (tf) and inverse cluster frequency (icf) of the words in the cluster. CS is a representative method for cluster summarization and labeling.
(4) *F-measure*, which is an alternative ISKR algorithm that considers the value of a keyword k with respect to a query q as the delta F-measure of q after adding k to q or removing k from q. As discussed in Section 11.3, since our goal function is to maximize the F-measure of a query, the delta F-measure more accurately reflects the value of a keyword than the benefit/cost ratio. However, in this approach, after a keyword is added to or removed form the current query, the values of all keywords will need to be updated, which potentially leads to a low efficiency.

We implemented Data Clouds and CS.

In ISKR and PEBC, we consider the top-20% words in the results in terms of tfidf for query expansion. In PEBC, we empirically set the number of points tested in each iteration as 3, and the number of iterations as 3. Since there are a lot of results for queries on Wikipedia data set, all systems only consider the top 30 results to generate expanded queries, where the results are ranked using tfidf of the keywords. We also set the maximal number of expanded queries for each approach to be 5.

Quality of Query Expansion

The evaluation of the quality of expanded queries consists of a user study and the measurement of scores of the expanded queries (Eq. 11.1).

User Study

We performed a user study an Amazon Mechanical Turk and had 45 users participate in our survey for evaluating the query expansion approaches. The user study consists of three parts.

Part 1: Individual Query Score. In order to test whether an expanded query generated by each approach is helpful for the users, we first asked the users to give a score for each expanded query in the range of 1-5, which is referred to as *individual query*

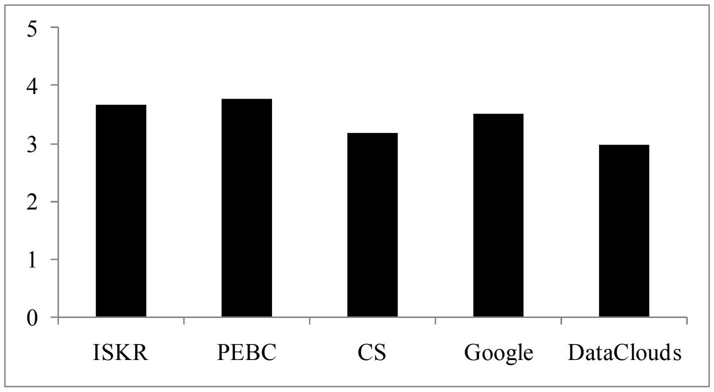


Figure 11.1: Average Individual Query Score

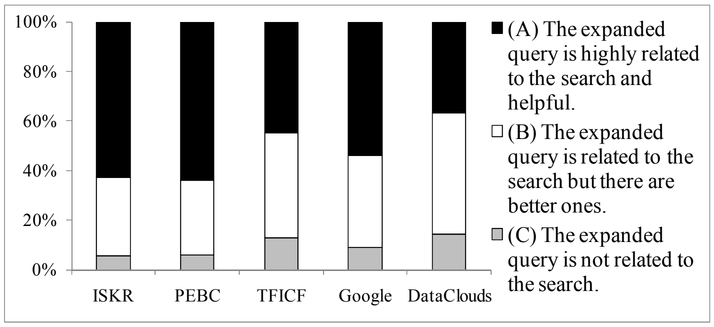


Figure 11.2: Percentage of Users Choosing Options (A), (B) and (C) for Individual Queries

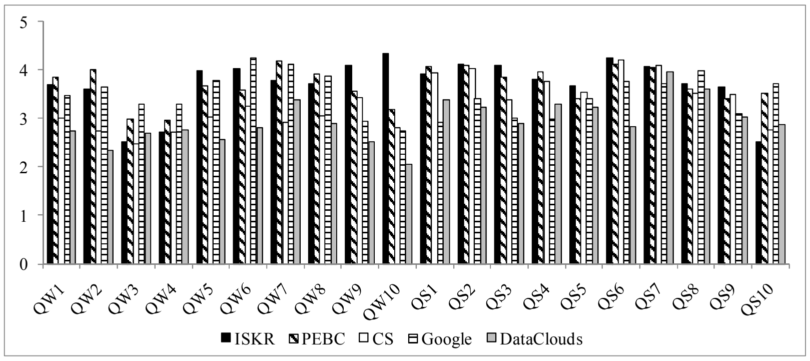


Figure 11.3: Collective Query Score for Each Set of Expanded Queries

score. The users were also asked to choose one of the options shown in Figure 11.2 as the justification of the score.

The average score of all 20 queries given by all users for each approach is shown in Figure 11.1, and the percentage of users choosing each option in this part of the user study is shown in Figure 11.2. As we can see, ISKR, PEBC and Google have higher average

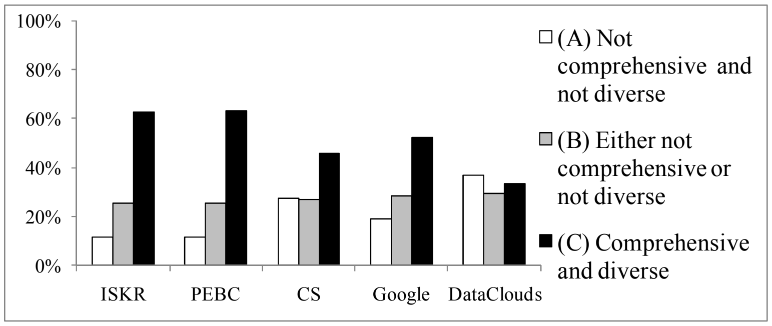


Figure 11.4: Percentage of Users Choosing Options (A), (B), (C) and (D) for Each Set of Expanded Queries

query scores than Data Clouds and CS. Recall that data clouds returns the top keywords in all results in terms of tf, idf and result rank, but such a top keyword may often be too specific (e.g., “multicellular” for QW_7) or too general, which is not informative as an expanded query. On the other hand, an expanded query generated by ISKR and PEBC maximally retrieves a cluster of results, thus is likely to have a better semantics. For example, for QW_6 ISKR and PEBC return “island” and “server”, which are more meaningful. Therefore, most users chose option (A) for both ISKR and PEBC, while data clouds got plenty of (B) and (C).

The CS approach chooses keywords based on TFICF, thus may tend to pick keywords that have high occurrence (TF) in a few results in the cluster. These keywords do not retrieve many results in the cluster, thus the users mostly found it less desirable. For example, for QW_6 “java”, it returns a query “Java, blog, Microsoft”, which is too specific and only cover a small part of the results.

Google chooses keywords based on query log, thus it often returns meaningful and popular keywords. For example, for QW_6 “Java”, Google returns the expanded queries “Java, Tutorials”, “Java, Games” etc., which are generally very popular with the users. However, for some queries Google may return keywords that do not occur in the results. For example, for QS_1 “Canon, products”, Google returns a query “Sony, products”. While this could be useful for some users, our user rating has indicated that many the users prefer the expanded queries to be results oriented.

There are a few queries that ISKR and/or PEBC do not generate the most meaningful expanded queries. This is mainly because the words that appear frequently in a cluster is not necessarily the best one semantically. It is especially likely for the Wikipedia data set, as it consists of document-centric XML with sentences/paragraphs, rather than succinct and informative features. Consider QW_1 "San Jose", for which one of our expanded queries is "player". Although this keyword is related to the sports teams of San Jose, the users suggested that returning team information (e.g., baseball, hockey, etc.) gives better expanded queries.

Part 2: Collective Query Score. Next we test whether the *set* of expanded queries for each user query provides a classification of the original query result set. We asked the users to give a collective score for all expanded queries of each user query, in the range of 1-5, and choose one of the options in Figure 11.4 as the justification of the score.

For all 20 queries, the collective score of each user query for each approach is shown in Figure 11.3, and the percentage of users that chose each option is shown in Figure 11.4. As we can see, ISKR, PEBC consistently received relatively high scores in collective scoring. Since each expanded query of ISKR and PEBC maximally covers the results in a cluster and minimally retrieves the results in other clusters, they are usually comprehensive (i.e., covering various aspects/meanings of the original query) and diverse (i.e., their results have little overlap). This was appreciated by the users as it is easy for the users to see all options and decide the expanded query for retrieving the relevant results, and the users gave favorable scores for ISKR and PEBC.

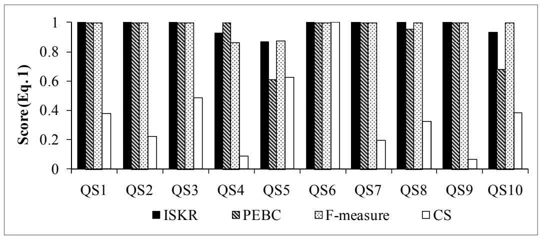
On the other hand, since Data Clouds only returns the top keywords in the results, the expanded queries may lack comprehensiveness and diversity. Consider QS_1 "Canon, products". Both ISKR and PEBC returns three main products of Canon: *camera*, *printer* and *camcorder*. However, data clouds returns *camera*, *printer* and *wp-dc26* (a camera-related product). As we can see, the expanded queries of data clouds do not cover camcorder products, failing to be comprehensive. Besides, the result of *wp-dc26* is contained in the result of *camera*, failing to be diverse. As a result, the users mainly chose options (A) and (B) for data clouds.

For the CS approach, as discussed before, it tends to pick keywords that have high occurrence in fewer results and do not cover the entire cluster. Such queries are usually too specific and thus fail to be comprehensive. For example, for QW_1 “San Jose”, CS returns “San Jose, sabercat, season, arena” and “San Jose, war, California, gold”. Since these expanded queries only retrieve a few results in the corresponding clusters, the user found them not comprehensive. Note that the CS approach has a better score on shopping data than the Wikipedia data. This is because in the shopping data, results are somewhat similar in that they share many common keywords. Therefore, even though the CS approach does not consider the relationship of keywords, the keywords it selects in an expanded query likely co-occur in many results. On the other hand, on the Wikipedia data, it may choose a set of keywords, such that each of them has a high occurrence but they do not necessarily co-occur. Such a query will not retrieve many results which lowers its recall. For example, for QW_9 “mouse”, it returns a query “mouse, technique, wheel, interface”. These keywords have high occurrences but low co-occurrences.

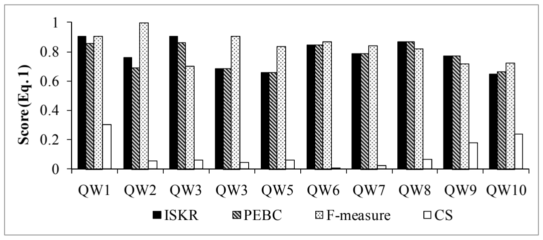
Since Google chooses expanded queries based on query log, it can also achieve comprehensiveness and diversity for some queries. For example, for QW_6 “Java”, Google returns the expanded queries “Java, Tutorials”, “Java, Games” and “Java, test”, which the users considered as comprehensive and diverse. However, sometimes the expanded queries returned by Google may not be diverse. For example, for QW_8 “rockets”, all expanded queries returned by Google are about space rockets, and none of them refers to the Rockets NBA team.

There are also a few cases where users choose (A) or (B) for ISKR and/or PEBC. Due to the limitation of the data, we may not have the results that cover all meanings of a query in the results. As an example, consider query QW_1 “San Jose”, the top-30 results are either about the city of San Jose in California, or about San Jose sports teams. Since San Jose is also a major city in Costa Rica, which is not covered by our expanded queries, some users selected (A) or (B) for our approach. Besides, due to imperfect clustering, sometimes it may be impossible to generate comprehensive and diverse expanded queries.

Part 3. To verify the intuition of our approach, we finally asked the users a general question: What is your opinion about a good set of expanded queries? According to the answers from the users, the majority of users considered comprehensiveness and diversity

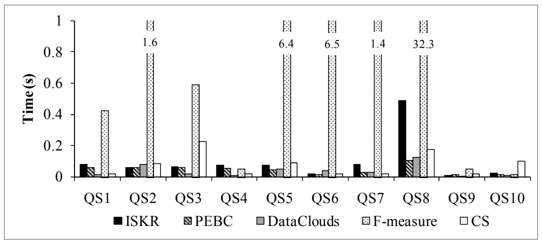


(a) Shopping Dataset

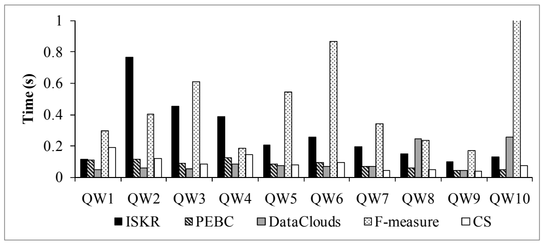


(b) Wikipedia Dataset

Figure 11.5: Scores of Expanded Queries (Eq. 11.1)



(a) Shopping Dataset



(b) Wikipedia Dataset

Figure 11.6: Query Expansion Time

as important properties for a set of expanded queries, which coincides with the philosophy of our proposed approaches.

Scores of Expanded Queries Using Eq. 11.1

As defined in Eq. 11.1, the score (goal function) of a set of expanded queries is the harmonic mean of their F-measures. In this section we test for each user query the score of expanded queries generated by ISKR, PEBC, the F-measure approach and CS, as shown in Figure 11.5. Since the queries generated by Data Clouds and Google are not based on clusters, this score is inapplicable.

As we can see, in general ISKR and PEBC achieve similar and good scores. On the shopping data, both algorithms achieve perfect score for many queries. This is because on the shopping data, products of different categories usually have different features. Thus for queries whose results contain several different product categories (e.g., QS_1 "Canon, products" whose products contain camcorders, printers, and cameras), each category forms a cluster, and it is usually possible to achieve a perfect precision and recall.

The scores of ISKR are generally a little better than those of PEBC. The reason is that in each iteration of ISKR, we select the best keywords to add to q or remove from q. Thus ISKR, although not necessarily produces the optimal expanded queries, does achieve some form of local optimality: it stops only if no single keyword can give a better value if we add it to q or remove it from q. On the other hand, PEBC relies on the assumption that, if two adjacent points have the highest average score, then the optimal query should lie in between these two points. Since this assumption is not always true, sometimes PEBC may not choose the best interval to zoom in. However, if PEBC chooses the right interval at each iteration, then it may achieve a better quality than ISKR as it will converge to the optimal solution, as the case of QS_4, QW_{10}, etc.

The F-measure approach generally has the same or slightly better quality than ISKR since delta F-measure is a more accurate measure of the value of a keyword. For some queries its scores are lower, since both algorithms are heuristics-based and ISKR may occasionally choose better keywords. However, as shown in Section 11.5, the F-measure approach has a poor efficiency, while efficiency is highly important for a search engine.

The CS approach usually has a poor score. This is because it chooses a set of keywords with high TFICF values with respect to a cluster, but these keywords may not occur in many results in the cluster, thus causing a low recall. For example, for QW_5 "eclipse" it returns a query "eclipse, core, plugin, official". Moreover, as discussed before, since the CS approach is designed to return cluster labels rather than query results, it does not consider the interaction of keywords. Therefore, it may return an expanded query whose keywords have high occurrences, but low co-occurrences.

Efficiency and Scalability

In the efficiency test, we measure the running time of all five methods. For all approaches except Data Clouds, the response time that the user perceives (besides the query processing time) includes both clustering and query expansion time. The average clustering time on shopping and Wiki data sets are 0.02s and 0.35s, respectively. For data clouds, we measure the time for finding the top-k words from a ranked list of results.

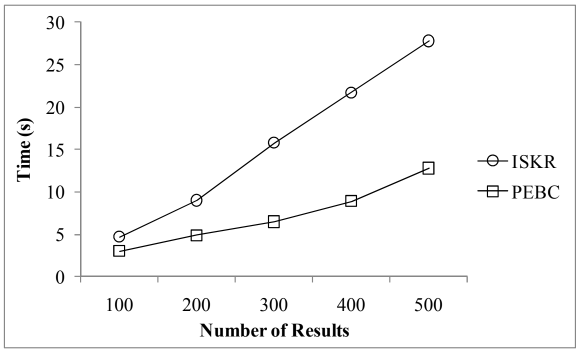


Figure 11.7: Scalability of Generating Expanded Queries over Number of Results

The processing time of query expansion is shown in Figure 11.6. In general, the ISKR algorithm takes more time to generate expanded queries comparing with PEBC. Recall that ISKR allows both adding or removing keywords to the current expanded query, thus it may have a large number of iterations before getting to the point that the query cannot be further improved. Besides, at each iteration ISKR needs to maintain the values of keywords, and in the worst case, the values of all keywords need to be updated. For QS_8 which has a large number of results (557) and a large number of distinct keywords (464 in the largest cluster), it is significantly slower than PEBC.

Both ISKR and PEBC are much more efficient than the F-measure method. As discussed in Section 11.3, the F-measure method needs to update the values (i.e., delta F-measure) of all keywords every time a keyword is added to or removed from a query. On the other hand, ISKR only needs to update the values of the keywords that do not appear in all delta results, the number of which could be a small percentage of all distinct keywords. For some queries the F-measure method takes more than 30 seconds. The efficiency of the CS approach is usually comparable with ISKR and PEBC, since the TFICF of a keyword can be efficiently computed. Data clouds is generally faster than both ISKR and PEBC, as it only needs to compute the tf and idf for each keyword in the results.

We have tested the scalability of all three approaches with respect to the number of results returned by the user query. We use query QW_2 “Columbia”, and vary the number of results from 100 to 500. The time shown for ISKR and PEBC include both clustering and query generation. As shown in Figure 11.7, the processing time of both approaches increases linearly, and they have a reasonable response time even if 500 results are used.

11.6 Summary

In this chapter we propose a novel framework for query expansion: generating a set of expanded queries that provides a classification of the original query result set. Specifically, the expanded queries maximally retrieve the results of the original query, and the results retrieved by different expanded queries are different. To achieve this, we propose to first cluster the results, and then generate an expanded query for each cluster, whose set of results should be as close to the cluster as possible. We formally define the Query Expansion with Clusters (QEC) problem. This problem is APX-hard. We then design two efficient algorithms ISKR and PEBC for generating expanded queries based on the clustered results.

This approach can be integrated with the structural clustering method discussed in Chapter 9 in building a query result clustering framework for structured data. As we discussed before, these two clustering approaches complement each other: the approach in Chapter 9 is based on structures and this approach is based on values in the results. The order of applying these two clustering methods depends on which type of ambiguity (structural or keyword) is considered more important. In general, we believe that structural ambiguity is more important to address since the user is searching structured data. However, value-based clustering is also desirable as it complements structured-based clustering. Therefore, we suggest to interleave these two clustering methods. For example, for query “auction, buyer, seller, Tom”, intuitively, we should first resolve the ambiguity of keyword roles: we cluster the results based on whether Tom is buyer, seller or auctioneer. If more clusters are desired by the user, we can use the value-based clustering method to refine the clusters into, for example, auctions of cars, auctions of houses, etc. If even more clusters are needed, we can further refine the clusters using the structural clustering method based on suffix paths.

Note that both clustering methods generate cluster descriptions: the structural clustering method generates a sentence while this approach generates expanded queries. They can be combined to generate one description for a cluster generated by the integrated approach. We can add phrase “about [expanded query]” to the sentence generated by the structural clustering method. For example, “Find auctions *about car* whose seller is Tom”.

Chapter 12

EXPLOITING AND MAINTAINING MATERIALIZED VIEWS

12.1 Motivation and Goal

Efficiency is highly important for a search engine. As discussed in Chapter 1, efficiently generating results for keyword search on structured data is more challenging than processing structured query or text search due to the larger search space. In this chapter, we discuss how to efficiently generate query results for keyword search on XML using materialized views, and how to incrementally maintain the views when the data is updated.

Materialized views have been proved successful for performance optimization in evaluating structured queries on databases [16, 110, 14, 129, 103, 143]. They have also been widely used in web applications. Caching query results as materialized views at the application tier can decrease the workload of the servers. Exploiting materialized views at client side can further decrease the number of network accesses.

Given the benefits of materialized views in structured query processing, it is a natural idea to explore them in the context of XML keyword search. In this part of the thesis, we study two problems. First, how should we leverage materialized views of previous keyword search results for efficient evaluation of new keyword searches on XML? Second, when the source data is updated, how should we incrementally maintain materialized views of XML keyword search to keep them fresh? Due to the differences of syntax, semantics and query processing techniques between XPath/XQuery and keyword searches, answering these questions for XML keyword searches presents a new set of challenges.[1]

How to exploit materialized views for query evaluation. Suppose we have a set of materialized views of previous XML keyword search results. Upon the arrival of a keyword query, we would like to utilize materialized views as much as possible in order to gain performance improvements. The first question is how to determine which views are *relevant* to a given query. If a relevant view itself can not answer the query, then

[1]A plausible approach would internally convert a keyword query to an XQuery and then directly use XQuery query processing and optimization techniques. However, as evaluated in the literature [86, 43], such an approach incurs unnecessary high time complexity. Therefore query processing techniques are specifically developed to efficiently handle keyword queries in all existing XML search engines.

can we find a set of views which together can answer the query, i.e. an *answer set*? Furthermore, if there are several answer sets, we need to efficiently find the optimal one for query answering.

Different techniques are needed to answer these questions for XML keyword search compared with XPath/XQuery. For instance, unlike XPath/XQuery where relevant views to a query are the ones that have containment relationship with the query,[2] new techniques are needed to identify relevant views for XML keyword search, as no two distinct keyword queries have containment relationship according to query result definitions in the literature [56, 144, 43, 86, 91, 82, 61]. On the other hand, if there are several answer sets to a query in the materialized views, it is often hard to find the optimal one (with an approximation bound) for XPath/XQuery; while we have designed an efficient $2(\ln|Q|+1)$ approximation algorithm for identifying optimal answer set for XML keyword search. Furthermore, the techniques of answering queries using views must be different.

Example 12.1 *Consider a keyword search query (*Brooks Brothers, store, Texas, name, city*) on the XML data in Figure 1.3. Given a set of materialized views, we first need to identify which ones are relevant. Suppose we find relevant views: V_1: (*Brooks Brothers, store, Texas*), V_2: (*Texas, name*), V_3: (*store, city*) and V_4: (*Brooks Brothers, store, city*). Then how should we maximally exploit relevant views with minimal accesses to the source data and index? Assuming that we find two sets of views to answer the query: S_1 = {V_1, V_2, V_3}, S_2 = {V_2, V_4}. Then we need to efficiently select the best answer set in order to minimize query processing cost. Furthermore, techniques need to be developed to use the selected view set for query answering.*

How to incrementally maintain materialized views. For a materialized view to be useful for evaluating new queries, it must be up-to-date with respect to dynamic source data. Given an update to the source data, which materialized views are affected? Can we maintain these materialized views incrementally using the original views and a small fragment of the source data without re-computing the views on the updated data from scratch?

[2]A query Q is contained in Q' if and only if the result of Q is contained in the result of Q' for any XML data.

These questions pose unique challenges for maintaining materialized views of XML keyword search compared with structured queries. Due to the ambiguity of keyword search, a search engine may not always be able to accurately identify user intentions. Many approaches therefore follow the best-effort approach to determine the semantics based on the current XML document and user input keywords [144, 56, 43, 86, 82]. When new knowledge is added to the data, they may refine their semantics and the corresponding query results. Therefore even if we only consider the AND semantics of keyword searches, inserting new data nodes can result in deletions in the materialized views, as shown in Example 12.2. This is different from structured queries, where unless the view definition contains negation, insertion to the data will not cause deletion in the materialized views. Similarly, deletion in XML data can cause insertions to the materialized views of keyword searches.

Example 12.2 *Consider a materialized view V for keyword search query Q (*Accord, award*) on the XML tree $\mathcal{D}$ in Figure 12.1. It is reasonable that the keyword matches in the subtree rooted at node* dealer *(2) constitute query result V. Now we insert a subtree rooted at* award *(10) into $\mathcal{D}$ and obtain the new XML tree $\mathcal{D}'$. Leveraging the information of the newly inserted data, a better semantics of Q would be: searching for the award of Accord cars. To be consistent with this semantics,* award *(4) in the result V should be replaced with the newly inserted* award *(10) [144, 91, 86, 56].*

To address these challenges, we present in Section 12.3 a general framework for exploiting materialized views for XML keyword search. We first identify the relevant materialized views that potentially can be used to answer a user query. We prove that given a set of relevant materialized views $\mathcal{V}$ and a user query Q, the decision problem of finding the minimal answer set in $\mathcal{V}$ is NP-complete.

We design and implement an XML keyword search engine that can answer queries using materialized views. A polynomial time approximation algorithm is proposed that finds the optimal answer set of materialized views to evaluate the input query. In order to keep the materialized views fresh, they are incrementally maintained upon XML data insertion or

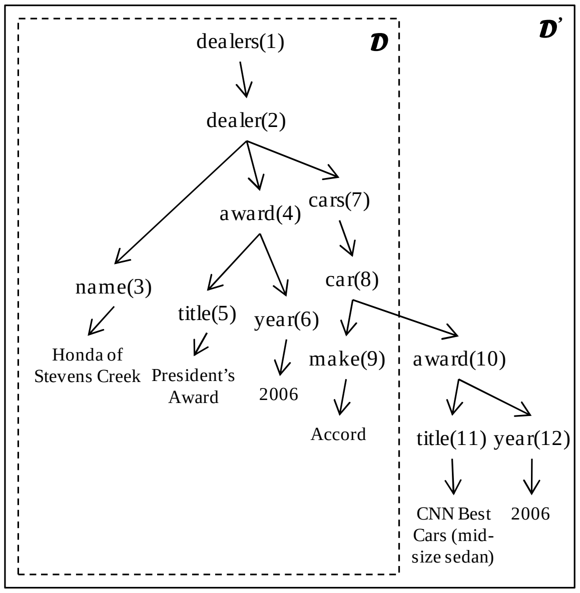


Figure 12.1: XML Trees $\mathcal{D}$ and $\mathcal{D}'$

deletion. The realization of these functionalities depends on how query results are defined. Our keyword search engine adopts a commonly used semantics to define query results for XML keyword search, the SLCA semantics, which will be reviewed in Section 12.4. SLCA semantics is used in XML keyword search systems,[3] including [86], XSeek [91], XKSearch [144], [126] and [61]. The techniques that we propose for answering queries using views and maintaining materialized views of XML keyword searches can be directly incorporated into these systems.

Our study of exploiting materialized views for evaluating XML keyword searches appeared as a poster paper in the 24th International Conference on Data Engineering (ICDE), 2008 [92].

12.2 Definitions

We consider an XML tree model, in which internal nodes denote elements and attributes, and leaf nodes denote text values. Two update operations on XML data are supported. It is easy to see that any updates to the data can be accomplished by a sequence of these primitive update operations.

[3]Some systems perform additional node filtering after SLCA computation is done.

1. $Insert(n, n')$, denoting an insertion of a tree rooted at node n as a child of node n'.

2. $Delete(n)$, denoting a deletion of node n along with its subtree.

In the following, we use *delta tree* to refer to the inserted or deleted subtree.

In this thesis we study incremental view maintenance upon an insertion and deletion of a single subtree. Multiple updates can be coped with one by one using our techniques. Note that there is a trade-off between the cost of update and the freshness of views. Maintaining the views after a number of data updates can potentially be less expensive than doing so after each single update; however, the latter guarantees that the views keep fresh and the former one does not. Different applications may be better served by different strategies.

The query issued by user is a set of keywords. We take the AND semantics of XML keyword search and define the query result of XML keyword search as follows.

Definition 12.1 *The* query result *of evaluating a keyword search Q on XML data $\mathcal{D}$, denoted as $Q(\mathcal{D})$, consists of* selected *lowest common ancestors (LCA) of keyword matches of Q. Specially, each $s \in Q(\mathcal{D})$ satisfies: the set of keyword matches in s's subtree contains at least one match to each keyword in Q; and this set is not the same as the set of keyword matches in the subtree rooted at any child of s.*

A materialized view *is a query whose result is materialized. Queries and views are used interchangeably.*

The size of a query Q, denoted as $|Q|$, is the number of keywords in Q. The size of a query result (or materialized view) on data $\mathcal{D}$, denoted as $|Q(\mathcal{D})|$, is the number of data nodes it has.

Definition 12.1 gives a general definition of XML keyword search result for AND semantics. It specifies a necessary condition that each XML keyword search result should satisfy. There are many variations in the literature of defining query results for XML keyword search involving AND condition [56, 86, 43, 144, 61, 82, 91], all of which are special cases of Definition 12.1.

We also define the relationship between two keyword queries with respect to the set of keywords they contain.

Definition 12.2 *Let Q_1 and Q_2 be two keyword queries. We say Q_2 is a* subquery *of Q_1, or Q_1 is a* superquery *of Q_2, denoted as $Q_2 \trianglelefteq Q_1$, if every keyword in Q_2 is in Q_1. Q_2 is a* proper subquery *of Q_1, denoted as $Q_2 \triangleleft Q_1$, if $Q_2 \trianglelefteq Q_1$, and there exists a keyword k, $k \in Q_1, k \notin Q_2$.*

Q is a union (intersection) of Q_1 and Q_2, denoted as $Q = Q_1 \cup Q_2$ ($Q = Q_1 \cap Q_2$), if Q consists of the keywords that are in either Q_1 or Q_2 (in both Q_1 and Q_2).

The difference of Q_1 and Q_2, denoted as $Q_1 - Q_2$ is a query that consists of the keywords in Q_1 except those also in Q_2.

12.3 Analysis

Given the general definition of XML keyword search result, we now discuss how to identify relevant views, answer sets, and the minimal answer set of a query, in order to leverage materialized views for query answering.

Definition 12.3 *A set of view $\mathcal{V} = \{V_1, V_2, \ldots, V_m\}$* can answer query *$Q$, if there exists an algorithm $\mathcal{A}$ such that for any data $\mathcal{D}$, $Q(\mathcal{D}) = \mathcal{A}(\mathcal{V}(\mathcal{D}))$, where $\mathcal{V}(\mathcal{D})$ is the set of materialized views, $\mathcal{V}(\mathcal{D}) = \bigcup\{V(\mathcal{D}) | V \in \mathcal{V}\}$. $\mathcal{V}$ is named as an* answer set *of Q.*

Next we discuss how to exploit materialized views in the context of XML keyword search. The following proposition shows that only the views that are subqueries of a query Q are relevant for evaluating Q.

Proposition 12.1 *Consider a universe of keywords $K = \{k_1, k_2, \ldots, k_n\}$, a set of materialized views $\mathcal{V} = \{V_1, V_2, \ldots, V_m\}$, $V_i \trianglelefteq K$, $1 \leq i \leq m$, and a query $Q \trianglelefteq K$. Let $\mathcal{V}'$ be the*

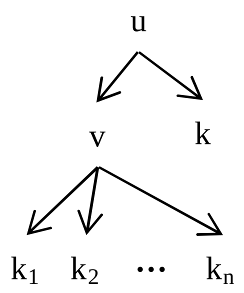


Figure 12.2: Proposition 7.2

set of subqueries of Q in $\mathcal{V}$: $\mathcal{V}' = \{V | V \in \mathcal{V} \wedge V \trianglelefteq Q\}$. We claim that Q can be answered by $\mathcal{V}$ if and only if Q can be answered by $\mathcal{V}'$. A view in $\mathcal{V}'$ is called a relevant view *to Q.*

Proof. $\Leftarrow$*: If Q can be answered by $\mathcal{V}'$, since $\mathcal{V}' \subseteq \mathcal{V}$, Q can be answered by $\mathcal{V}$.*

$\Rightarrow$*: Suppose (i) Q can be answered by $\mathcal{V}$, (ii) but can not be answered by $\mathcal{V}'$. Assumption (ii) indicates that there exists a document $\mathcal{D}$, for which there does not exist an algorithm $\mathcal{A}$, such that $\mathcal{A}$ can derive the results of $Q(\mathcal{D})$ from $\mathcal{V}'(\mathcal{D})$ (i.e. $\mathcal{A}(\mathcal{V}'(\mathcal{D})) = Q(\mathcal{D})$). Now for every distinct word in tag names or text values t in $\mathcal{D}$, $t \notin Q$, we replace all the occurrences of t with t', where t' is a word that does not appear in $\mathcal{D}$, and denote the new XML tree to be $\mathcal{D}'$. Since $\mathcal{D}'$ and $\mathcal{D}$ only differ in non-keyword words, $Q(\mathcal{D}) = Q(\mathcal{D}')$, $\mathcal{V}'(\mathcal{D}) = \mathcal{V}'(\mathcal{D}')$. According to the assumption (ii), there does not exist algorithm $\mathcal{A}$ such that $\mathcal{A}(\mathcal{V}'(\mathcal{D}')) = Q(\mathcal{D}')$. From the construction of $\mathcal{D}'$, it is easy to see that for any view V_i that contains keywords that are not in Q (i.e. $V_i \in \mathcal{V}$, $V_i \notin \mathcal{V}'$), we have $V_i(\mathcal{D}') = \emptyset$. Therefore $\mathcal{V}(\mathcal{D}') = \mathcal{V}'(\mathcal{D}')$. Together with the assumption (ii), we have: there does not exist an algorithm $\mathcal{A}$ such that $\mathcal{A}(\mathcal{V}(\mathcal{D}')) = Q(\mathcal{D}')$. However, this indicates that Q can not be answered by $\mathcal{V}$ (on XML document $\mathcal{D}'$), which conflicts with assumption (i).*

Proposition 12.2 shows that an answer set of an XML keyword query Q must have the union of keywords to be the same as the set of keywords in Q.

Proposition 12.2 *Given a query Q and a set of relevant materialized views $\mathcal{V} = \{V_1, V_2, \ldots, V_m\}$, $V_i \trianglelefteq Q$, $1 \leq i \leq m$, if Q can be answered by $\mathcal{V}$, i.e $\mathcal{V}$ is an answer set of Q, then $\bigcup(V \mid V \in \mathcal{V}) = Q$.*

Proof. Assume that Q can be answered by $\mathcal{V}$, but $\bigcup(V \mid V \in \mathcal{V}) \neq Q$. Since $V_i \trianglelefteq Q$, there exists a keyword k such that $k \in Q$, and $k \notin \bigcup(V \mid V \in \mathcal{V})$. Let the set of keywords $\bigcup(V \mid V \in \mathcal{V})$ be $\{k_1, k_2, ..., k_n\}$. We can construct an XML tree $\mathcal{D}$ as illustrated in Figure 12.2. Note that we must have the knowledge of where the matches to k locate, in order to find the result of Q. However, none of the views in $\mathcal{V}$ provides such information according to Definition 12.1. Therefore, Q can not be answered by $\mathcal{V}$.

To answer a keyword query Q from a set of views that satisfies the condition in Proposition 12.2, we must find an algorithm that can compute the query result of Q from the results of Q's (proper) subqueries. However, whether this is achievable depends on the definition of the keyword query result. Note that if, for a query result definition, the result of a query can not be computed from the results of its proper subqueries, then only the materialized view that is exactly the same as a query can be used to answer the query, which significantly limits the practical benefits of materialized views. Therefore in the rest of the papers, we do not consider such a trivial case, and focus the discussion on the general case where the answer set of a query can be any set of materialized views satisfying Proposition 12.2.

Fortunately, a commonly used query result definition of XML keyword search, SLCA [144, 91, 61, 126], allows the result of a query to be computed from the results of its subqueries, as will be analyzed in Section 12.4. It is an open question whether other XML keyword search result definitions in the literature allow query answering using subqueries.

If for a given query, there are multiple answer sets in a set of materialized views, then we need to find the best one. Intuitively, the fewer number of views in an answer set, the more efficiently it can be used to answer the query due to less computational cost and smaller intermediate query results.[4] However, finding such an answer set is hard.

Definition 12.4 *Consider a universal keyword set K, a query Q, $Q \subseteq K$, and a set of materialized views $\mathcal{V}$, $\forall V \in \mathcal{V}$, $V \subseteq K$. The* minimal answer set problem *is the following:*

[4]To be more accurate, we should select the answer set that can evaluate the query with the minimal cost. However, the cost measurements depend on the complexity of the algorithms that answer a query using views, which in turn depends on the definition of query result. Therefore we defer the discussion of finding the optimal answer set with minimal cost in Section 12.4 when the query definition is set.

given an integer i, can we find an answer set $\mathcal{V}'$ of Q, i.e. $\mathcal{V}' \subseteq \mathcal{V}$, $\forall V \in \mathcal{V}'$, $V \trianglelefteq Q$, $\bigcup(V \mid V \in \mathcal{V}') = Q$, such that $|\mathcal{V}'| \leq i$?

Theorem 12.1 *The minimal answer set problem is NP-complete.*

Proof. It is easy to see that this problem is in NP. We prove that this problem is NP-complete by reducing the set cover problem to it in polynomial time, that is, Set cover $\leq_P$ minimal answer set. Consider an instance of Set Cover: a universe $\mathcal{U}$, a collection $\mathcal{S}$ of subsets of $\mathcal{U}$, $\forall S \in \mathcal{S}$, $S \subseteq \mathcal{U}$, and an integer j, can we select a collection $\mathcal{S}'$ of at most j subsets from $\mathcal{S}$, $\mathcal{S}' \subseteq \mathcal{S}$, whose union is the universe $\mathcal{U}$, i.e. $\bigcup(S \mid S \in \mathcal{S}') = \mathcal{U}$? We construct a transformation in polynomial time: let $K = \mathcal{U}$, $\mathcal{V}' = \mathcal{S}'$, $\mathcal{V} = \mathcal{S}$, $Q = \mathcal{U}$, and $i = j$. It is easy to see that $\mathcal{U}$ has a set cover $\mathcal{S}'$ with at most j subsets if and only if Q has an answer set $\mathcal{V}'$ with at most i views.

12.4 Accelerating XML Keyword Search Using Materialized Views

Section 12.3 discusses the problem of identifying relevant materialized views and minimal answer sets for XML keyword search queries in general. Several problems remain to be addressed in order to feasibly and fully exploit materialized views. First, how to evaluate a query from an answer set? Second, how to find the optimal answer set that requires the minimal cost of query processing? Third, if we can not find an answer set for a query, can we maximally leverage the relevant views with minimal accesses to source data/index? Finally, how can we efficiently keep materialized views up-to-date? The answers to these questions depend on the specific definition of keyword query result.

In this Section, we present an XML keyword search system that exploits materialized views by answering the above questions. It adopts a commonly used query result definition based on the semantics of SLCA, which (or its variation) is used in many XML key search engines.

Search Semantics

To define query results of an XML keyword search, we follow the widely adopted semantics, Smallest Lowest Common Ancestor (SLCA).

For the convenience of notation, we overload the concept of SLCA with respect to XML node sets instead of keywords.

Definition 12.5 *Let $N_1, N_2, ..., N_n$ be n sets of nodes in XML data $\mathcal{D}$. $SLCA(\mathcal{D}, N_1, N_2, ..., N_n)$ consists of all such nodes s:*

1. *s contains at least one node in each N_i ($1 \leq i \leq n$) in its subtree.*
2. *There does not exist a descendant of s that satisfies condition 1.*

As we can see, suppose query Q contains keywords k_1, ..., k_n, and let M_i be the set of matches to k_i ($1 \leq i \leq n$), then $SLCA(\mathcal{D}, Q)$ = $SLCA(\mathcal{D}, M_1, M_2, ..., M_n)$.

As an example, for query Q = (*A, B*) on the XML data $\mathcal{D}_1$ in Figure 12.5, we have $SLCA(\mathcal{D}_1, \{A, B\})$ = $SLCA(\mathcal{D}_1$, matches to *A*, matches to *B*)= $\{$*E* (0.1.2)$\}$.

To efficiently retrieve query result of XML keyword search, we need to be able to quickly find out the lowest common ancestor of nodes. For this purpose, we assign each XML node n a *Dewey* label [133] as a unique ID, referred as $Dewey(n)$. We record the relative position of a node among its siblings, and then concatenate these positions using dot '.' starting from the root to compose the Dewey ID for the node. Dewey ID can be used to detect the relationship between nodes. A node n_1 is an ancestor of node n_2 if and only if $Dewey(n_1)$ is a prefix of $Dewey(n_2)$. This indicates that the $lowest\ common\ ancestor$ (LCA) of nodes n_1 and n_2 has the Dewey ID which is the longest common prefix of $Dewey(n_1)$ and $Dewey(n_2)$. Besides, we say $Dewey(n_1)$ is smaller than $Dewey(n_2)$ if n_1 appears before n_2 in document order.

Furthermore, to retrieve keyword matches efficiently, we build the commonly used *data inverted index* which maps a word in the data to a sorted list of the Dewey IDs of the data nodes whose tags or text values contain this word.

In this section, we present the algorithm of answering queries using relevant views for query result definition given in Definition 4.7. We first show that a keyword query can be answered by a set of subqueries whose union is equal to the query.

Proposition 12.3 *For two queries Q_1 and Q_2 on data $\mathcal{D}$, $SLCA(\mathcal{D}, Q1 \cup Q2)$ = $SLCA(\mathcal{D}$, $SLCA(\mathcal{D}, Q1), SLCA(\mathcal{D}, Q2))$.*

Proof. According to Definition 4.7, for each $s \in SLCA(\mathcal{D}$, $Q_1 \cup Q_2)$, s contains all the keywords in both Q_1 and Q_2. Therefore s must contain at least one node in $SLCA(\mathcal{D}, Q_1)$ and one node in $SLCA(\mathcal{D}, Q_2)$ in its subtree. Furthermore, none of s's descendants can contain a node in $SLCA(\mathcal{D}, Q_1)$ and a node in $SLCA(\mathcal{D}, Q_2)$ in its subtree (otherwise, such a descendant disqualifies s to be in $SLCA(\mathcal{D}, Q_1 \cup Q_2)$). According to Definition 12.5, all such s nodes compose $SLCA(\mathcal{D}$, $SLCA(\mathcal{D}, Q_1), SLCA(\mathcal{D}, Q_2))$.

Proposition 12.1 *For query Q = $Q_1 \cup Q_2 \cup ... \cup Q_k$, we have $SLCA(\mathcal{D}, Q)$ = $SLCA(\mathcal{D}$, $SLCA(\mathcal{D}, Q_1)$, ..., $SLCA(\mathcal{D}, Q_k))$.*

Given a query Q and a set of materialized views $\mathcal{V}$, if a keyword in Q is not in any views in $\mathcal{V}$, then $\mathcal{V}$ itself is not sufficient to answer the query. To maximally leverage materialized views, we relax the definition of answer set as a subset of views $\mathcal{V}'$ that contain all keywords in Q found in $\mathcal{V}$. When $\mathcal{V}' \neq Q$, we additionally access the data inverted index to retrieve the matches to keywords that are in Q but not in $\mathcal{V}'$, which together with $\mathcal{V}'$ can answer Q.

Definition 12.6 *Given a query Q and a set of materialized views $\mathcal{V}$, we generalize the* answer set *of Q as $\mathcal{V}'$, such that $\mathcal{V}' \subset \mathcal{V}$, $\bigcup(V \mid V \in \mathcal{V}') = \bigcup(V \mid V \in \mathcal{V}) \cap Q$.*

The algorithm of answering a query using views is presented in Figure 12.3. It first invokes procedure $findRelViews$ (Figure 35) to find $answerSet$, an answer set of Q from

```
1: $answerSet$ = findAnswerSet($Q$, $\mathcal{V}$)
2: if $answerSet \neq \emptyset$ then
3:     $SLCA(\mathcal{D}, Q)$ = $answerSet[1]$
4: for $j$ = 2 to $answerSet.size$ do
5:     $SLCA(\mathcal{D}, Q)$ = $computeSLCA(SLCA(\mathcal{D}, Q),$
       $SLCA(\mathcal{D}, answerSet[j]))$ {apply the algorithm
       in [144]}
6: for each keyword $k$ in $Q - answerSet$ do
7:     $SLCA(\mathcal{D}, Q)$ = $computeSLCA(\mathcal{D}, SLCA(\mathcal{D}, Q),$
       matches of $k$ in $\mathcal{D}$) {apply the algorithm in [144]}
```

Figure 12.3: Answering Query Using Views

$\mathcal{V}$ (line 1). Then it computes the SLCAs of the views in the answer set (line 2-5). If Q has a keyword k that is not in the answer set, we access the data inverted index to retrieve the matches to k, which is used to answer Q according to Corollary 12.1 (line 6-7).

Example 12.3 *Consider evaluating $Q = \{A, B, C, D, E\}$ given a set of materialized views $\mathcal{V}$: $Q_1 = \{A, B\}$, $Q_2 = \{A, B, C\}$, $Q_3 = \{D\}$, $Q_4 = \{B, D\}$, $Q_5 = \{E, F\}$. $Q_1, ..., Q_4$ are subqueries of Q and therefore are relevant to Q. Since keyword E is not in any view that is a subquery of Q, we need to access the matches to E in the data inverted index.*

For the purpose of computing SLCA, we adopt the algorithm proposed in XK-Search [144].[5] The cost of answering Q on data $\mathcal{D}$ with a set of materialized view $\mathcal{V}$ using the Algorithm in Figure 12.3 is

$$cost(\texttt{SLCA}) = O(|V_{min}| \cdot \sum_{V_i \in \mathcal{V}} \texttt{log}|V_i|) \tag{12.1}$$

which is bounded by $O(|V_{min}(\mathcal{D})||\mathcal{V}|\text{log}|V_{max}(\mathcal{D})|)$, where V_{min} and V_{max} are the smallest set and biggest set, respectively, among all materialized views in the answer set together with the sets of keyword matches retrieved from the data inverted index.

Finding Optimal Answer Set

In Section 12.3 we discussed the problem of selecting the minimum answer set with respect to a general query result definition for XML keyword search, which is NP-hard. Now we

[5]Specifically, our implementation adopts the Indexed Lookup Eager algorithm, as it generally outperforms Scan Eager and stack algorithms.

discuss for SLCA semantics, whose time complexity of answering queries using views is given in Eq. 12.1, how to select the *optimal* answer set of a query which has the minimal cost of processing the query among all answer sets. It is easy to see that this problem is also NP-complete, as in the special cases when each materialized view has the same size, the goal becomes finding the smallest number of views, which is the same as the minimal answer set problem shown to be NP-complete in Theorem 12.1.

We propose Algorithm 12.4 in search of the optimal answer set from a set of materialized view $\mathcal{V}$ for query Q on data $\mathcal{D}$.[6] Let $\mathcal{V}_Q$ be the set of relevant materialized views of Q in $\mathcal{V}$, $answerSet$ be the set of views selected from $\mathcal{V}_Q$ to evaluate Q, $currQ$ record the remaining keywords in Q that are not in $answerSet$. We set $Q.cost = \texttt{log}|Q(\mathcal{D})|$, where $|Q(\mathcal{D})|$ denotes the size of the materialized view of Q. According to the time complexity of answering a query using views that we have discussed, it is crucial to first pick the relevant materialized view Q (i.e. $Q \in \mathcal{V}_Q$) whose size $|Q(\mathcal{D})|$ is the smallest, i.e. $Q.cost$ is minimal among all relevant views to minimize the processing cost (line 5). Let this view be Q_s. Now the problem becomes: selecting a subset of $\mathcal{V}_Q$ whose union is equal to $\bigcup(Q \mid Q \in \mathcal{V}) \cap Q - Q_s$, such that the cost $\sum\limits_{Q_i \in \mathcal{V}} \texttt{log}|Q_i|$ is minimized. We use a greedy algorithm to select the rest of the views. For each $Q' \in \mathcal{V}_Q$, we use $Q'.cover$ to record the number of uncovered keywords in Q that can be covered by Q', which is initialized to be $|Q'|$, and $Q.benefit$ = $Q.cover/Q.cost$. At each step, we select the view that has the maximal $benefit$ value among all the views in $\mathcal{V}_Q$, with ties broken arbitrarily (line 13). When a query Q' is chosen and added to $answerSet$, for each previously uncovered keyword k that is now covered by Q', we find each view Q'' in $\mathcal{V}_Q$ that contains k, and decrease $Q''.cover$ by one, and update $Q''.benefit$ (line 6-9, 16-19). The procedure continues till all the keywords in Q are covered, or none of the views in $\mathcal{V}_Q$ can provide additional cover (line 12, 14-15).

After selecting a set of relevant views, we invoke the Algorithm in Figure 12.3 to compute the results of Q using views and data inverted index if necessary.

[6]Although the processing cost in Eq. 12.1 involves both materialized views and keyword match lists, we only need to discuss how to select views to compose an answer set since keyword match lists are determined directly by $\bigcup(V \mid V \in \mathcal{V}) \cap Q$.

```
1: $currQ$ = the set of keywords in $Q$ {$currQ$ records the set of uncovered keywords in $Q$}
2: $answerSet = \emptyset$ {$answerSet$ records the set of selected materialized views}
3: $\mathcal{V}_Q$ is a subset of $\mathcal{V}$ consisting of subqueries of $Q$
4: for $Q' \in \mathcal{V}_Q$, set $Q'.cover = |Q'|$, $Q'.cost = log|Q'(\mathcal{D})|$, $Q'.benefit = Q'.cover/Q'.cost$
5: select $Q'$ such that $Q'.cost$ is minimal over all views in $\mathcal{V}_Q$
6: for each keyword $k \in Q'$ do
7:    for each view $Q'' \in \mathcal{V}_Q$ that contains $k$ do
8:       $Q''.cover$ = $Q''.cover$-1
9:       $Q''.benefit$ = $Q''.cover/Q''.cost$
10: $answerSet = answerSet \cup Q'$
11: $currQ$ = $currQ$ - $Q'$
12: while $currQ \neq \emptyset$ do
13:    select $Q'$ such that $Q'.benefit$ is maximal over all views in $\mathcal{V}_Q$
14:    if $Q'.benefit$=0 then
15:       break
16:    for each keyword $k \in currQ \cap Q'$ do
17:       for each view $Q'' \in \mathcal{V}_Q$ that contains $k$ do
18:          $Q''.cover$ = $Q''.cover$-1
19:          $Q''.benefit$ = $Q''.cover/Q''.cost$
20:    $answerSet = answerSet \cup Q'$
21:    $currQ$ = $currQ$ - $currQ \cap Q'$
22: return $answerSet$
```

Figure 12.4: Find Answer Set for a Query

Example 12.4 *Continuing Example 12.3, we have $\mathcal{V}_Q$ is $\{Q_1, Q_2, Q_3, Q_4\}$, and initialize $currQ = \{A, B, C, D, E\}$, and $answerSet = \emptyset$. Suppose their costs (the size of the materialized views) are 100, 60, 80 and 20, respectively. Since $Q_4.benefit$ = $Q_4.cover/Q_4.cost$ = 0.1 has the largest $benefit$ in $\mathcal{V}_Q$, we update $answerSet = \{Q_4\}$, $currQ = \{A, C, E\}$. Now $Q_3.cover = Q_4.cover = 0$, $Q_1.cover = 1$ and $Q_2.cover = 2$. Next step, we choose the view with the largest $benefit$: Q_2, and update $answerSet = \{Q_2, Q_4\}$, $currQ = \{E\}$. Now, no query in $\mathcal{V}_Q$ covers any keyword in $currQ$, so the algorithm in Figure 35 terminates.*

To compute $SLCA(\mathcal{D}, Q)$, we invoke the algorithm in Figure 12.3. We first compute $SLCA(\mathcal{D}, Q) = SLCA(\mathcal{D},\ SLCA(\mathcal{D}, Q_2),$ $SLCA(\mathcal{D}, Q_4))$. Then we access the data for the matches to keyword E in $currQ$, and update $SLCA(\mathcal{D},\ Q)$ = $SLCA(\mathcal{D},\ SLCA(\mathcal{D},\ Q)$, matches of $E)$.

Note that although the algorithm looks similar as the greedy algorithm for weighted set cover problem, the cost of answering queries using views is *not* the summation of the cost of all selected views, as the case for weighted set cover. We know that the greedy algorithm of weighted set cover has an approximation ratio of $\mathtt{ln}|Q|+1$. Now an immediate question is whether the algorithm in Figure 35 has an approximation bound. We prove that the cost of answering a query Q using the answer set returned by Figure 35 is no more than $2(ln|Q|+1)$ times of the cost of answering Q using the optimal answer set.

Theorem 12.2 *The algorithm in Figure 35 can find the optimal answer set with an approximation ratio of $2(\mathit{ln}|Q|+1)$, where $|Q|$ is the number of keywords in Q.*

Proof. We use OPT to denote the cost of answering the query using the optimal answer set S, and APP to denote that using the answer set S' found by the algorithm in Figure 35. Suppose S consists of p views: $S = \{V_1...V_p\}$, where V_1 is the smallest materialized views in S; and S' has q views: $S' = \{V_1'...V_q'\}$, where V_1' is the smallest materialized view in S'. So we have $OPT = |V_1(\mathcal{D})| \cdot \sum_{i=2}^{p} \mathit{log}|V_i(\mathcal{D})|$, and $APP = |V_1'(\mathcal{D})| \cdot \sum_{i=2}^{q} \mathit{log}|V_i'(\mathcal{D})|$.

Consider an instance Ins of the Weighted Set Cover (WSC) problem: a universe $\mathcal{U}$ consists of all keywords in Q, and a collection $\mathcal{S}$ consists of subsets of $\mathcal{U}$, $\forall S \in \mathcal{S}$, $S \subseteq \mathcal{U}$. Each $S \in \mathcal{S}$ is associated with a cost c_S which is a positive real number. In Ins, each S is a view $V \in \mathcal{V}_Q$, and $c_V = V(\mathcal{D})$. WSC is the problem of finding a collection $\mathcal{S}' \subseteq \mathcal{S}$, such that $\bigcup(S \mid S \in \mathcal{S}') = \mathcal{U}$ and $\sum_{S\in\mathcal{S}'} c_S \leq l$. Suppose for Ins, the optimal cost is OPT_{Ins}.

Let Ins' be another instance of WSC, in which each keyword $k \in V_1'$ is removed from the universal set $\mathcal{U}$, and all other settings are the same as Ins. Let the optimal cost of Ins' be $OPT_{Ins'}$.

Since V_1 is the smallest materialized view in S, we have

$$\sum_{i=2}^{p} \mathit{log}|V_i(\mathcal{D})| \geq \frac{p-1}{p} \cdot \sum_{i=1}^{p} \mathit{log}|V_i(\mathcal{D})| \geq \frac{p-1}{p} OPT_{Ins}$$

which means

$$OPT \geq |V_1(\mathcal{D})| \cdot \frac{p-1}{p} \cdot OPT_{Ins} \tag{12.2}$$

On the other hand, after we choose the smallest materialized view V_1' from $\mathcal{V}_Q$ in the algorithm in Figure 35, the problem has become exactly WSC' described above. The procedure of choosing $V_2'...V_q'$ in Figure 35 has an approximation ratio of ln$|Q|$+1 [44]. Therefore,

$$\sum_{i=2}^{q} log|V_i'(\mathcal{D})| \leq OPT_{Ins'} \cdot (ln|Q|+1)$$
$$\leq OPT_{Ins} \cdot (ln|Q|+1)$$

which means

$$APP \leq |V_1'(\mathcal{D})| \cdot OPT_{Ins} \cdot (ln|Q|+1) \tag{12.3}$$

According to Figure 35, V_1' is the smallest materialized view in $\mathcal{V}_Q$, we have $|V_1(\mathcal{D})| \geq |V_1'(\mathcal{D})|$. When $p=1$, there is only one relevant view and our algorithm finds it as the optimal solution. When $p>2$, from Eq. 12.2 and Eq. 12.3 we have

$$\frac{APP}{OPT} \leq 2(ln|Q|+1)$$

As we can see, there are several operations that are often performed in Figure 35, including finding whether a view is materialized, and finding all views that cover a given a keyword. To speed up the processing, we build a *view inverted index* maps a word to a sorted list of IDs of the queries that contain this word. A *view existence index* maps a query to a boolean value, denoting whether the result of the query is materialized or not.

Now we analyze the complexity. In line 3, we find all subqueries of Q that are materialized, which takes $O(|\mathcal{V}|)$, where $|\mathcal{V}|$ is the number of materialized views. The while loop in line 12-22 is similar to the greedy algorithm for weighted set cover problem [44], which takes $O(|\mathcal{V}||Q|^2)$.The total time complexity of the algorithm in Figure 35 is therefore $O(|\mathcal{V}||Q|^2)$.

12.5 Incrementally Maintaining Views

For materialized views to be useful, they need to keep fresh upon data updates. In this Section, we discuss how to incrementally maintain materialized views of XML keyword search

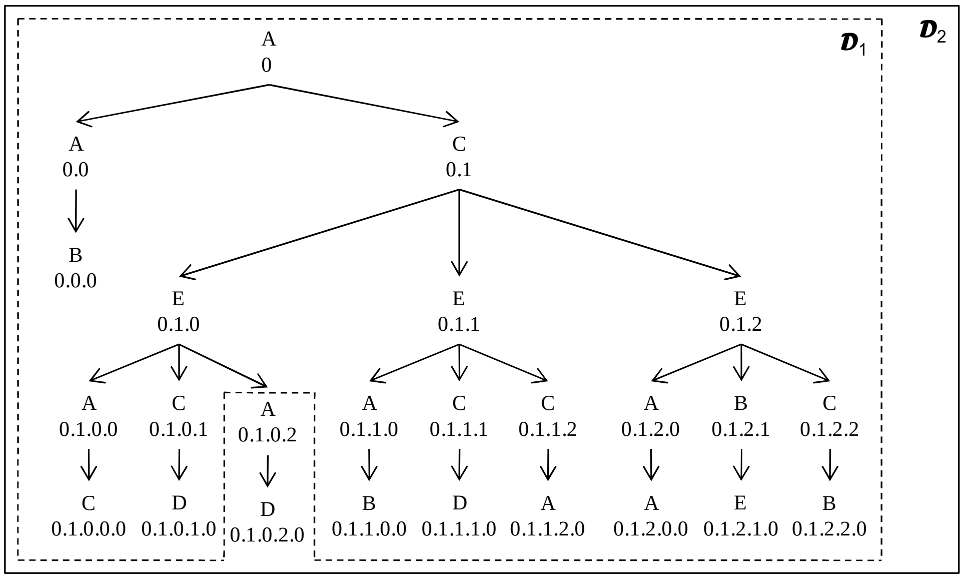


Figure 12.5: XML Trees $\mathcal{D}_1$ and $\mathcal{D}_2$

upon insertion and deletion of an XML subtree.

Identifying Affected Views Not all the views are affected by an XML data update. The first task is to efficiently identify affected views to be checked for maintenance. Note that SLCA is defined based on the nodes that match keywords, thus a view may be affected by an update if the delta tree contain at least one match to a keyword in the view. For a given update, we traverse the delta tree (inserted subtree or deleted subtree) and find the keywords contained in the update. For each keyword, we use the view inverted index (introduced in Section 12.4) to find out the list of views that contain this keyword. Then by traversing on these lists in a similar fashion as merge-sort, we can find out all the views that need to be checked for maintenance, and the keywords in the update tree that each view contains. Next we focus the discussion on maintaining each view for data insertion and data deletion, respectively.

Incremental View Maintenance upon Insertion

For materialized views to be useful, they need to be maintained upon insertions or deletions of subtrees.

First let us look at how to identify affected views to be checked for maintenance upon an update. Recall that SLCA is defined based on the nodes that match keywords, thus a view may be affected by an update if the delta tree contains at least one match to a keyword in the view. For a given update, we traverse the delta tree (inserted subtree or deleted subtree) and find the keywords contained in the update. For each keyword, we use the view inverted index (introduced in Section 12.4) to find out the list of views that contain this keyword. Then by traversing on these lists in a similar fashion as merge-sort, we can find out all the views that need to be checked for maintenance, and the keywords in the update tree that each view contains.

In this Section we discuss view maintenance for data insertion. Consider the insertion of a subtree rooted at node n to a node n' in an XML tree, $Insert(n, n')$. Suppose the original data is $\mathcal{D}_1$, the updated data is $\mathcal{D}_2$,[7] and the inserted subtree is $\mathcal{T}$. A data insertion can result in an insertion and deletion in the view.

The algorithm of maintaining the materialized views of keyword search upon a subtree insertion is presented in Figure 37. We consider two different cases.

Case 1: The inserted subtree $\mathcal{T}$ contains all keywords in Q (line 5-11 of Figure 37). Since there must exist nodes in $\mathcal{T}$ that are the SLCA nodes of Q, such an insertion qualifies new SLCA nodes. We apply the algorithm proposed in [144] on $\mathcal{T}$ to compute $SLCA(\mathcal{T}, Q)$, and have $SLCA(\mathcal{T}, Q) \subseteq SLCA(\mathcal{D}_2, Q)$.

Example 12.5 *Consider $\mathcal{D}_2$ in Figure 12.5, obtained after an insertion of a subtree $\mathcal{T}$ rooted at node $A(0.1.0.2)$ to the original tree $\mathcal{D}_1$. Let a query Q be $\{A, D\}$. We have node $SLCA(\mathcal{D}_1, Q) = \{0.1.0, 0.1.1\}$. Since $\mathcal{T}$ contains both keywords in Q, by applying algorithm [144], we have node $A(0.1.0.2) \in SLCA(\mathcal{D}_1, Q)$ as a new SLCA.*

On the other hand, $SLCA(\mathcal{T}, Q)$ may disqualify some existing SLCA nodes. Specifically, if a node in $SLCA(\mathcal{T}, Q)$ is a descendant of an SLCA node $s \in SLCA(\mathcal{D}_1, Q)$, then s is no longer an SLCA on the updated data $\mathcal{D}_2$. To find such s node(if exist), instead of checking every node in $SLCA(\mathcal{T}, Q)$ with respect to $SLCA(\mathcal{D}_1, Q)$, it is equivalent

[7]Note that upon a data update, the Dewey labels of some nodes may be affected. Efficient maintenance of Dewey labeling is an orthogonal issue, and has been investigated in literature [80, 41, 109].

```
1: input: query $Q$, $SLCA(\mathcal{D}_1, Q)$, original data $\mathcal{D}_1$, $Insert(n, n')$, new data $\mathcal{D}_2$,
   delta tree $\mathcal{T}$
2: output: $SLCA(Q, \mathcal{D}_2)$
3: let $k_1, \ldots, k_p$ be keywords in $Q$ that are not contained in $\mathcal{T}$
4: $SLCA(\mathcal{D}_2, Q) \leftarrow SLCA(\mathcal{D}_1, Q)$
5: **if** $\mathcal{T}$ contains all keywords **then**
6:    $SLCAT \leftarrow computeSLCA(\mathcal{T}, Q)$ {apply the algorithm in [144]}
7:    **if** if $n$ is a descendant of $s \in SLCA(\mathcal{D}_1, Q)$ **then**
8:       $SLCA(\mathcal{D}_2, Q) \leftarrow SLCA(\mathcal{D}_2, Q) - \{s\}$
9:    $SLCA(\mathcal{D}_2, Q) \leftarrow SLCA(\mathcal{D}_2, Q) \cup SLCAT$
10:   return $SLCA(\mathcal{D}_2, Q)$
11: $currLCA \leftarrow n$
12: **for** $j \leftarrow 1$ to $p$ **do**
13:    $currLCA \leftarrow$ the ancestor of $currLCA$ and $lowest(k_j, currLCA, \mathcal{D}_1)$ [144]
14: **if** $currLCA$ is not an ancestor of any $s \in SLCA(\mathcal{D}_1, Q)$ **then**
15:    $SLCA(\mathcal{D}_2, Q) \leftarrow SLCA(\mathcal{D}_2, Q) \cup currLCA$
16:    **if** $currLCA$ is a descendant of $s$ **then**
17:       $SLCA(\mathcal{D}_2, Q) \leftarrow SLCA(\mathcal{D}_2, Q) - \{s\}$
18: return $SLCA(\mathcal{D}_2, Q)$
lowest $(k_j, r, \mathcal{D}_1)$
1: {$lowest(k_j, r, \mathcal{D}_1)$ is lowest ancestor-or-self of node $r$ in $\mathcal{D}_1$ that contains a
   match to keyword $k_j$.}
2: $lm \leftarrow leftMatch(k_j, r, \mathcal{D}_1)$
3: $rm \leftarrow rightMatch(k_j, r, \mathcal{D}_1)$
4: return the lower one of $LCA(lm, r)$ and $LCA(rm, r)$
```

Figure 12.6: View Maintenance upon Data Insertion

to check n. Intuitively, if n is a descendant of $s \in SLCA(\mathcal{D}_1, Q)$, then for every node $s_{\mathcal{T}} \in SLCA(\mathcal{T}, Q)$, $s_{\mathcal{T}}$ is a descendant of s. On the other hand, if node $s_{\mathcal{T}}$ is a descendant of s, then n must be a descendant of s, as s is not a node in the tree rooted at n. Since SLCA nodes do not have ancestor-descendant relationship, there is at most one ancestor of n in $SLCA(\mathcal{D}_1, Q)$, which can be found efficiently according to the following proposition.

Proposition 12.4 *If there is a node $s \in SLCA(\mathcal{D}, Q)$ that is an ancestor of n, then s has the largest Dewey ID that is smaller than $Dewey(n)$ among all the nodes in $SLCA(\mathcal{D}, Q)$.*

Proof. Suppose s does not have the largest Dewey ID that is smaller than $Dewey(n)$ among all nodes in $SLCA(\mathcal{D}, Q)$, i.e. there is a node $s' \in SLCA(\mathcal{D}, Q)$, such that $Dewey(s) < Dewey(s') < Dewey(n)$. Since $Dewey(s)$ is a prefix of $Dewey(n)$, it is easy to see that $Dewey(s)$ must also be a prefix of $Dewey(s')$, and therefore s is an ancestor of s'. However, since SLCA nodes do not have ancestor-descendant relationship, this is impossible, thus s must have the largest Dewey ID that is smaller than $Dewey(n)$.

Symmetrically, we have the following proposition with omitted proof due to limited space, which will be used later in this Section.

Proposition 12.5 *If there are nodes in $SLCA(\mathcal{D}, Q)$ that are descendants of n, then node s with the smallest Dewey ID that is larger than n among all the nodes in $SLCA(\mathcal{D}, Q)$ is a descendant of n.*

The propositions show that to find the ancestor (descendant) of a given node n in $SLCA(\mathcal{D}, Q)$, we only need to search node s with the largest Dewey ID that is smaller than $Dewey(n)$ (with the smallest Dewey ID that is larger than $Dewey(n)$), then check if s is an ancestor (descendant) of n. Since $SLCA(\mathcal{D}, Q)$ is sorted by dewey ID, this can be done by a binary search.

Example 12.6 *Continuing Example 12.5, for the newly found node in $SLCA(\mathcal{T}, Q)$: $0.1.0.2$, we find the largest Dewey ID in $SLCA(\mathcal{D}_1, Q)$ that is smaller than $0.1.0.2$: $0.1.0$. Since node $0.1.0$ is an ancestor of node $0.1.0.2$, it no longer qualifies to be an SLCA. We remove it and have $SLCA(\mathcal{D}_2, Q) = \{0.1.0.2, 0.1.1\}$.*

Case 2: Now let us consider an inserted subtree $\mathcal{T}$ that does not contain all keywords in Q. According to the discussion in Section 12.5, we can find the keywords in Q that are not contained in $\mathcal{T}$, denoted as $k_1, ..., k_p$. Three steps need to be performed to compute $SLCA(\mathcal{D}_2, Q)$, as shown in Figure 37.

First, we find the the lowest ancestor of n that contains matches to keywords $k_1, ..., k_p$ in its subtree, denoted as $currLCA$, which is a potential new SLCA (line 12-14). We define $lowest(k_j, n, \mathcal{D})$ as the lowest ancestor of n in XML data $\mathcal{D}$ that contains a match to keyword k_j. As shown in [144], $lowest(k_j, n, \mathcal{D})$ must be either the LCA node of $leftMatch(k_j, n, \mathcal{D})$ and n, or the LCA node of $rightMatch$ $(k_j, n, \mathcal{D})$ and n, where $leftMatch(k_j, n, \mathcal{D})$ is the match to k_j in $\mathcal{D}$ with the largest Dewey ID that is smaller than $Dewey(n)$, and $rightMatch(k_j, n, \mathcal{D})$ is the match to k_j with the smallest Dewey ID that is larger than $Dewey(n)$. Since the list of nodes matching a keyword are sorted by their Dewey ID in the data inverted index, $leftMatch(k_j, n, \mathcal{D})$

and $rightMatch(k_j, n, \mathcal{D})$ can be found efficiently using binary search. Initially, we set $currLCA$ = n. For each keyword k_j from k_1 to k_p, if $currLCA$ is a descendant of $lowest(k_j, n, \mathcal{D})$, then we set $currLCA$ = $lowest(k_j, n, \mathcal{D})$. Finally, $currLCA$ is the lowest ancestor of n that contains matches to all the keywords.

Example 12.7 *Consider $\mathcal{D}_1$ and $\mathcal{D}_2$ in Figure 12.5. Let query $Q = \{B, D, E\}$. $SLCA(\mathcal{D}_1, Q)$ = $\{0.1.1\}$. Since $\mathcal{T}$ does not contain keywords B and E, we need to find the lowest ancestor of node $0.1.0.2$ that contains keywords B and E in its subtree. The list of nodes in $\mathcal{D}_1$ that match keyword B is: {0.0.0, 0.1.1.0.0, 0.1.2.1}. We find $leftMatch(B, 0.1.0.2, \mathcal{D}_1)$ to be $0.0.0$, and $rightMatch(B, 0.1.0.2, \mathcal{D}_1)$ to be $0.1.1.0.0$. Therefore $lowest(B, 0.1.0.2, \mathcal{D}_1)$= $LCA(0.1.1.0.0, 0.1.0.2)$=0.1. Similarly, we find the lowest ancestor of $0.1.0.2$ that contains E, which is $0.1.0$. The lowest ancestor of $0.1.0.2$ that contains all the keywords in Q is therefore node 0.1.*

Next, given $currLCA$, the lowest ancestor of n that contains matches to all keywords in Q, we need to check whether $currLCA$ qualifies to be a node in $SLCA(\mathcal{D}_2, Q)$ or not (line 14). Specifically, if there exists a node $s \in SLCA(\mathcal{D}_1, Q)$, such that $currLCA$ is an ancestor of s, then $currLCA$ is disqualified. This can be checked according to Proposition 12.5.

Example 12.8 *Continuing Example 12.7, we check whether $currLCA = 0.1$ is an ancestor of a node in $SLCA(\mathcal{D}_1, Q)$ = $\{0.1.1\}$, which is indeed the case. Therefore $currLCA$ is not a new SLCA, and $SLCA(\mathcal{D}_2, Q)$ = $\{0.1.1\}$.*

Finally, if $currLCA$ is identified as a node in $SLCA(\mathcal{D}_2, Q)$, then we need to further check whether any existing SLCA node should be removed (line 16-17). This is done by checking whether any existing SLCA node is an ancestor of $currLCA$ using Proposition 12.4.

Now we analyze the complexity of Figure 37. If $\mathcal{T}$ contains all keywords, the time complexity is $M_{\mathcal{T}min}|Q|d\log M_{\mathcal{T}max}$, where $M_{\mathcal{T}min}$ and $M_{\mathcal{T}max}$ are the minimum and maximum number of matches to a keyword in delta tree $\mathcal{T}$, respectively, and d is the depth of

```
1: input: query $Q$, $SLCA(\mathcal{D}_1, Q)$, original data $\mathcal{D}_1$, $Delete(n)$, new data $\mathcal{D}_2$,
   delta tree $\mathcal{T}$
2: output: $SLCA(\mathcal{D}_2, Q)$
3: let $\mathcal{T}$ be the subtree rooted at $n$
4: $SLCA(\mathcal{D}_2, Q)$ = $SLCA(\mathcal{D}_1, Q)$
5: **if** $n$ is an ancestor of $s \in SLCA(\mathcal{D}_1, Q)$ **then**
6:     $SLCA(\mathcal{D}_2, Q) = SLCA(\mathcal{D}_2, Q) - \{s\}$
7:     return $SLCA(\mathcal{D}_2, Q)$
8: **else if** $n$ is a descendant of $s \in SLCA(\mathcal{D}_1, Q)$ **then**
9:     Let $k_1, ..., k_p$ be the keywords that are contained in $\mathcal{T}$
10:    $currLCA$ = $s$
11:    **for** $j$ = 1 to $p$ **do**
12:        $currLCA$ = the ancestor of $currLCA$ and $lowest(k_j, s, \mathcal{D}_2)$)
13:    $SLCA(\mathcal{D}_2, Q) = SLCA(\mathcal{D}_2, Q) - \{s\}$
14:    **if** $currLCA$ is not an ancestor of any $s \in SLCA(\mathcal{D}_2, Q)$ **then**
15:        $SLCA(\mathcal{D}_2, Q) = SLCA(\mathcal{D}_2, Q) \cup currLCA$
16: return $SLCA(\mathcal{D}_2, Q)$
lowest ($k_j, r, \mathcal{D}_2$)
1: {$lowest(r, k_j)$ is the match node of $k_j$ in $\mathcal{D}_2$ that has the lowest LCA with $r$.}
2: $lm$ = $leftMatch(k_j, r, \mathcal{D}_2)$
3: $rm$ =$rightMatch(k_j, r, \mathcal{D}_2)$
4: return the lower one of $LCA(lm, r)$ and $LCA(rm, r)$
```

Figure 12.7: View Maintenance upon Data Deletion

$\mathcal{T}$. If $\mathcal{T}$ does not contain all keywords, then the running time of line 12-13 of Figure 37 is $O(|Q|d\text{log}M_{max})$, where M_{max} is the largest number of matches to a keyword of Q in $\mathcal{D}_1$. The running time of line 14 and 16 is O(logM_{min}), where M_{min} is the minimum number of matches in the whole XML tree to a keyword in $\mathcal{D}_1$, which is the upper bound of $|SLCA(\mathcal{D}_1, Q)|$. So the time complexity when $\mathcal{T}$ does not contain all keywords in Q is $O(|Q|d\text{log}M_{max})$.

Therefore, the overall complexity of Figure 37 is O(max$\{M_{Tmin}|Q|d\text{log}M_{Tmax}$, $|Q|d\text{log}M_{max}\}$).

Incremental View Maintenance upon Deletion

Now let us consider a deletion of a subtree $\mathcal{T}$ rooted at node n from the XML data, $Delete(n)$. Let $\mathcal{D}_1$ be the original data and $\mathcal{D}_2$ be the updated data. According to the discussion in Section 12.5, we can efficiently find the keyword k_1,..., k_p that are contained in the deleted tree $\mathcal{T}$.

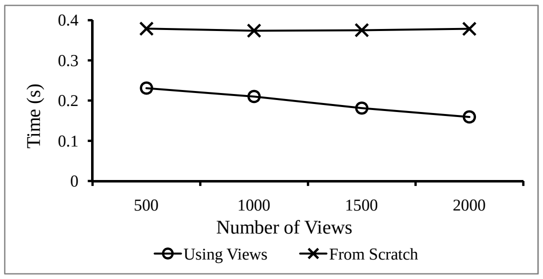
The algorithm for maintaining a materialized view Q upon a data deletion is presented in Figure 38. There are three possible cases considering the relationship of n and an existing SLCA node $s \in SLCA(\mathcal{D}_1, Q)$.

Case 1: If n is neither an ancestor nor a descendant-or-self of any $s \in SLCA(\mathcal{D}_1, Q)$, then the deletion will not affect the query result, i.e. $SLCA(\mathcal{D}_2, Q)$ = $SLCA(\mathcal{D}_1, Q)$.

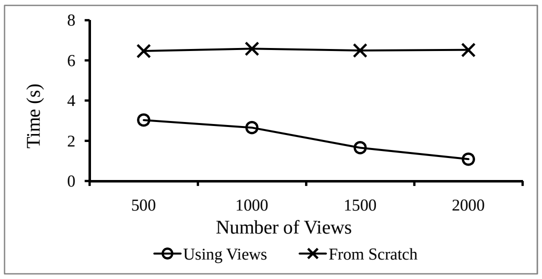
Case 2: If n is an ancestor of a node $s \in SLCA(\mathcal{D}_1, Q)$, then none of such s is an SLCA node for the updated data, as it has been removed. $SLCA(\mathcal{D}_2, Q) = SLCA(\mathcal{D}_1, Q) - s$, for all s (line 5-7 of Figure 38).

Case 3: If n is a descendant of a node $s \in SLCA(\mathcal{D}_1, Q)$, then s may no longer contain all the keywords of Q after the deletion. If so, we search for the lowest ancestor of s that contains all keywords in Q, denoted as $currLCA$, and replaces s with $currLCA$ for $SLCA(\mathcal{D}_2, Q)$. The way to find $currLCA$ is similar as Figure 37 (line 11-12), except that the initial value of $currLCA$ is set to be s rather than n. The reason is that n does not contain any keyword in $\mathcal{D}_2$ (as the subtree rooted at n has been removed), so if we set $currLCA$ as n initially, we need to check all keywords in Q to find the potential new SLCA $currLCA$. On the other hand, since $s \in SLCA(\mathcal{D}_1, Q)$, s may no longer contain keywords $k_1, ..., k_p$ in D_2 because of the removal of subtree $\mathcal{T}$, but it must contain all other keywords in its subtree, so we only need to check these p keywords. Therefore it saves time by initializing $currLCA$ to be s. After we find $currLCA$ which is the node that contains all the keywords in Q, it is put into the SLCA list $SLCA(\mathcal{D}_2, Q)$ if it does not have any descendant in $SLCA(\mathcal{D}_2, Q)$ (line 14-15), checked according to Proposition 12.5. This check is necessary, as exemplified in the following example.

Example 12.9 *Consider Q = $\{A, C, D\}$ on Figure 12.5, where $\mathcal{D}_2$ is the original data, $\mathcal{D}_1$ is the updated data with a deletion of the subtree $\mathcal{T}$ rooted at node $n = A(0.1.0.2)$ from $\mathcal{D}_2$. To show this example, suppose that node 0.1.0.1.0 is B rather than D. Notice that n is a descendant of node $s = 0.1.0$ in $SLCA(\mathcal{D}_2, Q) = \{0.1.0, 0.1.1\}$. Since $\mathcal{T}$ contains keywords A, D, we know s contains keyword $Q - \{A, D\} = \{C\}$ in its subtree. $lowest(A, s, \mathcal{D}_1) = 0.1.0$, and $lowest(D, s, \mathcal{D}_1) = 0.1$. So s is no longer an SLCA of Q on $\mathcal{D}_1$, instead, $C(0.1)$ becomes a potential new SLCA. However, since $C(0.1)$ is an ances-*



(a) Auction



(b) DBLP

Figure 12.8: Average Query Processing Time, Varying Number of Views

tor of $E(0.1.1) \in SLCA(\mathcal{D}_1, Q)$*,* $C(0.1)$ *is disqualified to be an SLCA in* $SLCA(\mathcal{D}_1, Q)$*. Therefore,* $SLCA(\mathcal{D}_1, Q)$ = $\{0.1.1\}$.

The complexity of Figure 38 is similar as Figure 37. Line 5, 8 and 14 takes $O(\log M_{min})$. Line 11-12 takes
$O(|Q| d \log M_{max})$. Therefore, the total time complexity of Figure 38 is $O(|Q| d \log M_{max})$.

12.6 Experiments

For performance evaluation, we compared the proposed techniques that exploits materialized views as much as possible for query evaluation, referred as *Using Views*, with the approach that only uses indexes without views, referred as *From Scratch*. We also compared the performance of incremental view maintenance, referred as *Incremental Maintenance*, with re-computing materialized views from indexes, referred as *From scratch*. The *From Scratch* approach for query evaluation/view materialization is implemented based on XKSearch [144].

An inverted full-text index for XML data was built using Oracle Berkeley DB, and is used by all comparison approaches whenever necessary, including the case when a query can not be (totally) answered by materialized views, as well as for view maintenance.

The experiments were conducted on a machine with 3.0GHz AMD Athlon(TM) dual-core CPU, 4.0GB memory, running Microsoft Windows Server 2008 Enterprise operating system. The algorithms were implemented in Microsoft Visual C++ 8.0.

Two data sets are used in the experiments. A synthetic auction data set generated by XMark with default schema has a size of up to 1.5GB; and a real-world DBLP data set has a size of 436MB.[8]

Exploiting Materialized Views for Evaluating Queries

To evaluate the effectiveness of materialized views in query evaluation, we synthesized a workload of queries following the existing approaches where the distribution of keyword occurrences in the query workload satisfies Zipf-Law and the exponent z is 1 [79, 118, 54, 103]. We extract all distinct words in tag names and text values in each XML data set. Each word is given a random rank in the Zipfian distribution. Each test query is generated containing a random number of keywords varying from 2 to 6, where each keyword is randomly selected based on the Zipfian distribution. Since it is an open problem of selecting which views to be materialized for XML keyword search, we take a baseline approach. According to Section 12.3, for a given query, only its subqueries can be used as relevant views, thus it is more reasonable to materialize small queries than larger ones. We therefore generate the views randomly in the same way as queries, except that each view contains either 2 or 3 keywords.

For 2000 test queries with 1000 materialized views on Auction data of 216MB, the average time per query of answering queries using views and that of answering queries from scratch are 0.31 second and 0.38 second, respectively. Whenever a query has a relevant view, we refer this as a hit to materialized views. The overall hit rate is 55.4%. Among this only 3.8% of the queries can find exactly same views, and 6.4% of the queries can be completed answered by views. For DBLP data, the average query processing times for using views and from scratch are 4.05 second and 5.23 second, respectively. It has a hit rate of 30.8%, 1.8% of the queries can find exactly the same views, and 2.5% of the queries can be completely answered by views. The average look-up time on view inverted indexes is less than 0.001 second, which is negligible.

Note that the strategy of selecting which views to materialize is outside the scope of

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

this chapter. We currently randomly generates the views based on the Zipfian distribution. Since the materialized views only contain 2 or 3 keywords, a good hit rate is achieved. However, most of the queries have to resort to data indexes to be fully answered. A more carefully designed view selection algorithm would decrease the number of index accesses and achieve a larger performance speedup for the whole query workload. To avoid the bias of varying hit rates, in the following we only report and analyze the average processing time for queries that can be partially or fully evaluated by views.

We test the efficiency of exploiting materialized views for query evaluation from two aspects: varying the number of materialized views and varying data size, respectively.

1) Scalability over Number of Views. In this test, we increase the number of materialized views from 500 to 2000 on both the Auction data set of size 216MB and the DBLP data set. 2000 queries are tested.

Figure 12.8 (a) and (b) shows the average query processing time, varying the number of views. As we can see, if materialized views are exploited, the query processing time significantly decreases when the number of views increases. When more materialized views are available, it is more likely that a query will have relevant views, and therefore fewer accesses to data index is needed and less query evaluation computation is required. Although an increasing number of materialized views entails a larger cost for identifying relevant views and finding optimal answer set, such performance overhead is imperceptible ($<$ 0.001 second for view look-up), and the overall performance speedup becomes larger when the number of views increases. On the other hand, the time of query evaluation without leveraging materialized views remains unchanged.

We also observe that the saving on DBLP data set is relatively larger than that of Auction. By analyzing its data characteristics, we find that the average number of matches to each word in DBLP is larger than that of Auction, and therefore its query evaluation is much more expensive. On the other hand, the result of a query, even for a 2-keyword query, is very small. For instance, the number of matches to “database" is huge, while the number of results of evaluating a query “database, Levine" is much smaller. Therefore, as long as a query can find a relevant view, its processing time is dramatically reduced.

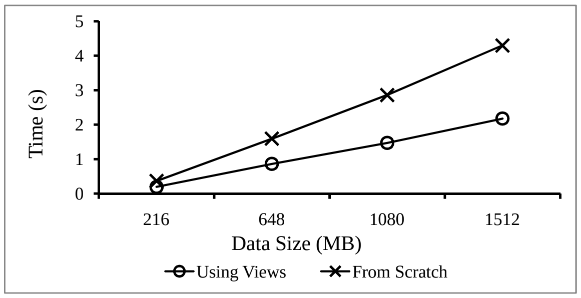


Figure 12.9: Average Query Processing Time, Varying Data Size

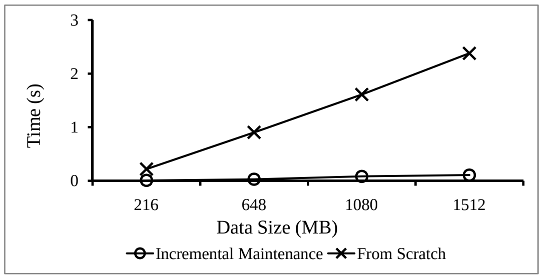
2) Scalability over Data Size. To test the efficiency of our approach with respect to different data sizes, we test the Auction data with sizes varying from 216MB to 1512MB, by replicating the file multiple times. 1000 materialized views and 2000 queries are randomly generated. The average query processing is shown in Figure 12.9. As we can see, the processing time of using views and that of from scratch both increase linearly when data size increases. The performance speedup of using views becomes larger when the data size increases.

Materialized View Maintenance

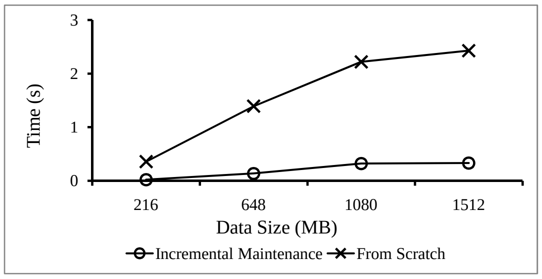
To test the efficiency of incremental view maintenance upon data update, we test its processing time with respect to different data size and delta tree size.

We randomly generated 1000 materialized views and 10 delta trees, and report the average time required to maintain a single view upon a single update in Figure 12.10 and 12.11. To generate a delta tree, we first find for each tag name, the average size of its subtree. For example, the average subtree size of *item* is 63. To generate a delta tree of size about 63, we randomly pick a node of tag *item* in the original data and use a copy of its subtree as a delta insertion tree. The insertion or deletion position is randomly selected. The processing times shown in the figures do not include the time for maintaining the inverted index and the Dewey labels, as these are the same for both approaches and are orthogonal to this study.

1) Scalability of Data Size. We vary the Auction data size from 216MB to 1512MB to test the efficiency of our approach. Delta trees are randomly selected with an average

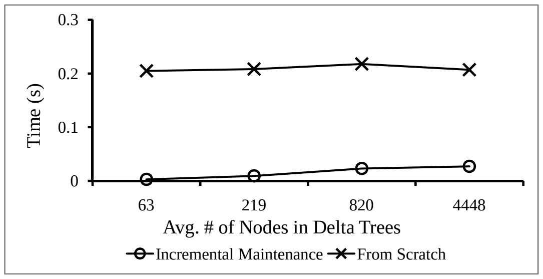


(a) Insertion

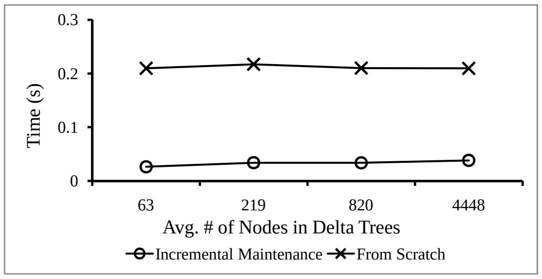


(b) Deletion

Figure 12.10: Average View Maintenance Time, Varying Data Size



(a) Insertion



(b) Deletion

Figure 12.11: Average View Maintenance Time, Varying Delta Tree Size

size of 63. As shown in Figure 12.10 (a) and (b), the processing times of incremental view maintenance and view maintenance from scratch both increase linearly with the increase of data size. Incremental maintenance is far more efficient than computing views from scratch, and performance benefits become larger when the data size increases.

2) Scalability of Delta Tree Size. The performance of incremental view maintenance and computing views from scratch while varying the average delta tree size from 63 to 4448 nodes on the Auction data is shown in Figure 12.11 (a) and (b). The time of computing from scratch is almost not affected, as the size of the delta tree is small compared to the original data. On the other hand, the time required for incremental view maintenance increases when the size of delta trees increases. This is because a larger delta tree contains more words, therefore more materialized views are likely to be affected, and a higher maintenance cost would be expected. Furthermore, for views that have all keywords appear in the delta tree, the number of query results within the delta tree likely increases when the delta tree size increases, and therefore also counts for a longer maintenance time.

In summary, experiment evaluation shows that our approach is much more efficient in answering XML keyword queries using views and incrementally maintaining views, compared with answering queries and maintaining views from scratch.

12.7 Summary

In this chapter we address an open problem of exploiting and maintaining materialized views for XML keyword search. We analyze the problem of identifying the best answer set of materialized views to evaluate a given query, which is NP-hard. We present the first XML keyword search engine that can answer queries using materialized views for SLCA semantics. We propose a polynomial time approximation algorithm for finding a good answer set of a given query from a set of materialized views, and develop the algorithm of answering query using its answer set. For materialized views to be useful for dynamic XML data, we design incremental view maintenance algorithms upon data updates. Our techniques can be incorporated into the XML keyword search systems that adopt SLCA semantics [91, 144, 126, 61]. Experimental evaluation shows significant performance improvements of our approach over computing query results or views from scratch.

Chapter 13

SEARCHING WORKFLOW HIERARCHIES

13.1 Motivation and Goal

As discussed in Chapter 2, besides tree and graph data, we also investigate in another data model, i.e., nested graph, which contains a graph and recursive composite node definitions. As to be discussed later, due to the uniqueness of this data model, existing result definition for keyword search on trees and graphs fail to produce meaningful results on nested graphs.

Workflow hierarchy is one type of data that conforms to this data model. Workflow hierarchies are widely used in scientific [17, 59, 123, 134, 22, 127] and business [20, 134] domains, which ease the analysis, maintenance and reusability of workflows. As an example, a workflow hierarchy describing the recipe of curry chicken is shown in Fig. 1.7. A node represents a task, which can be a step in a recipe, a web service invocation, a database query, a program run, or an experiment step, etc. A directed solid edge between nodes represents their dependency, dataflows, or control flows (AND/OR/XOR), referred as *dataflow edge*. For instance, in the bottom box in Figure 1.7, after tasks *add tenderizer* (0.1.0.0), we need to *wait 10 min* (0.1.0.3), and then have the data fed into the next task *put into skillet* (0.1.1.0).

In order to reduce the analysis complexity, enable modularity and re-use [13, 108], simplify provenance analysis [17, 23, 42], and achieve security [32], *composite task* is often defined to abstract a group of tasks into a single task, as supported in many workflow management systems, such as Kepler [2] and myExperiment [4]. For example, composite task *tenderize chicken breast* (0.1.0) is an abstraction of a group of tasks consisting of *add tenderizer*, *sprinkle curry powder*, *add garlic* and *wait 10 min*. Dotted lines connecting group of tasks to its abstraction composite task are referred to as *abstraction edges*. Composite tasks can be recursively defined to form a workflow hierarchy, such as the one in Figure 1.7. On the other hand, the most detailed tasks (those shown in italic in bottom box Figure 1.7) are called *atomic tasks*.

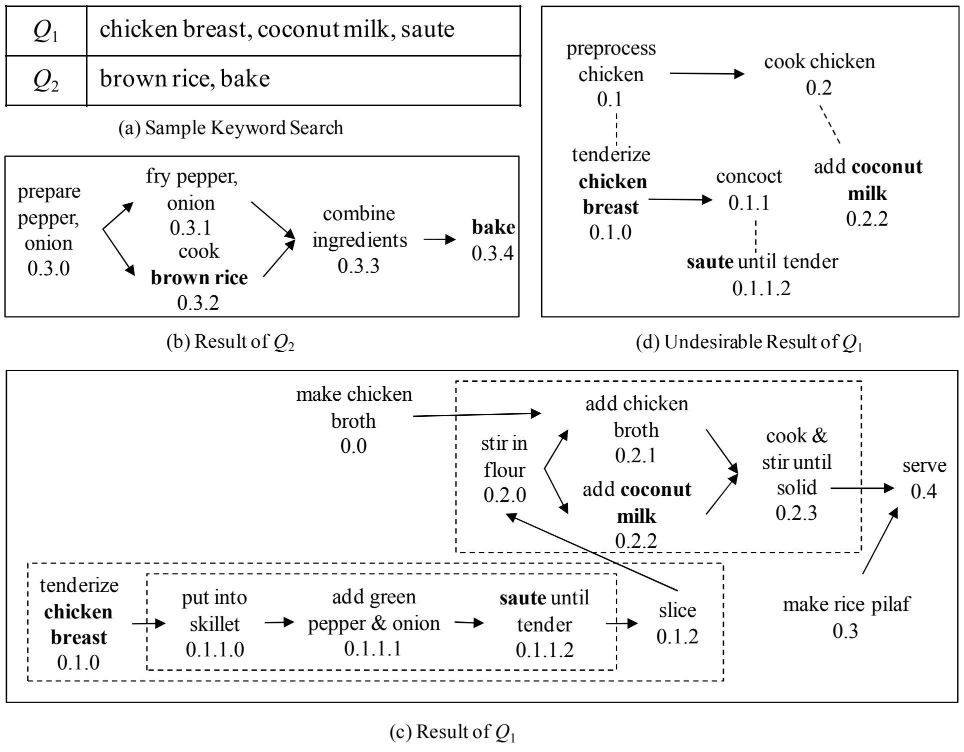
It is highly desirable if a user can search relevant workflow hierarchies in a repository using keywords, and then re-use or revise them as needed when designing new workflows, so that the design phase will be easier and be shortened compared with designing new ones from scratch.Suppose that a user would like to make a dish using chicken breast and coconut milk by sauting, but doesn't have a recipe in mind. She would issue a keyword query "*chicken breast, coconut milk, saute*" (Q_1 in Figure 13.1(a)) on a repository of recipes to find useful ones.

We can easily find the workflow hierarchies in the repository that contain matches to query keywords. Suppose Fig. 1.7 is one of such workflows in the repository, where keyword matches are in bold font. Obviously, returning the whole workflow hierarchy as a query result is not *concise*, as an overwhelming volume of information is delivered to the user (e.g., many workflow hierarchies in the repository [2, 9, 7, 4] contain hundreds of nodes).

The immediate challenge is how to define query results for keyword search on workflows. Given much research done on keyword search on graph-structured data (e.g., relational) and tree-structured data (e.g.,XML), a natural question is whether we can adopt their approaches: defining a query result on workflow hierarchies as a smallest tree in the data that contains the query keywords. A result for query Q_1 "*chicken breast, coconut milk, saute*" using these approaches is shown in Fig. 13.1(d).

However, such query results are not desirable for two reasons. First, the results do not necessarily capture the dataflows among keyword matches, and thus fail to be *informative* on node relationships. For example, the relationship of tasks containing *chicken* (0.1.0) and *saute* (0.1.1.2) is expressed as a path of both dataflow edges and cross-layer expansion edges, while their dataflow is not captured, which should be: *tenderize chicken breast* (0.1.0) $\rightarrow$ *put into skillet* (0.1.1.0) $\rightarrow$ *add green pepper & union* (0.1.1.1) $\rightarrow$ *saute until tender* (0.1.1.2).

Besides, returning smallest subtrees does not necessarily produce *self-contained* query results. Consider another query Q_2 "*brown rice, bake*" in Figure 13.1(a). The smallest subtree is the path from *cook brown rice* (0.3.2) to *bake* (0.3.4). However, such a path itself does not have a semantic meaning. A clear meaning can only be obtained if we consider



Q_1	chicken breast, coconut milk, saute
Q_2	brown rice, bake

(a) Sample Keyword Search

prepare pepper, onion 0.3.0
fry pepper, onion 0.3.1
cook **brown rice** 0.3.2
combine ingredients 0.3.3
bake 0.3.4
(b) Result of Q_2
preprocess chicken 0.1
cook chicken 0.2
tenderize **chicken breast** 0.1.0
concoct 0.1.1
add **coconut milk** 0.2.2
saute until tender 0.1.1.2
(d) Undesirable Result of Q_1
make chicken broth 0.0
stir in flour 0.2.0
add chicken broth 0.2.1
add **coconut milk** 0.2.2
cook & stir until solid 0.2.3
serve 0.4
tenderize **chicken breast** 0.1.0
put into skillet 0.1.1.0
add green pepper & onion 0.1.1.1
saute until tender 0.1.1.2
slice 0.1.2
make rice pilaf 0.3
(c) Result of Q_1

Figure 13.1: Sample Queries and Results on Workflow Hierarchies

the nodes in this path together with nodes (0.3.0, 0.3.1) (shown Figure 13.1(b)), which correspond to a composite task *make rice pilaf* (0.3), meaning that these nodes as a whole is a workflow about *making rice pilaf*.

According to our user studies (Section 5.2), when a user issues a keyword query on a repository of workflow hierarchies, each of which has abstractions with different granularities of atomic tasks, it is likely that s/he is interested in retrieving *workflow views* that contain these keywords and show their relationships. In our example, it is desirable to present Figure 1.5(c) as a query result of Q_1. This result explicitly captures the dataflow among keyword matches (shown in bold). For example, after *saute (the chicken) until tender* (0.1.1.2), we *slice (it)* (0.1.2), *stir (it) in flour* (0.2.0) and then *add coconut milk (0.2.2)*. Note that the dataflows among nodes in different composite tasks (i.e., different dashed line boxes) are not explicitly shown in the workflow hierarchy, but are *dynamically synthesized* from the workflow hierarchy. Such a query result is also self-contained, corresponding to a composite task *curry chicken*.

In this chapter, we present WISE, a Workflow Information keyword Search Engine, which, to the best of our knowledge, is the first work that returns query results capturing query keywords and their dataflows. WISE has been published in PVLDB 10 [98] and demonstrated at ICDE '09 [125]. The contributions of WISE include:

First, we address an open problem of defining query results for keyword search on hierarchical workflows. As we have discussed, a good query result should be informative (i.e., capturing the keyword matches and their dataflows), self-contained (i.e., having a name/goal), and concise (i.e., the minimal graph that is informative and self-contained). To achieve this, we start with formally defining the concept of a *view* of workflows, which is a graph defined in the same spirit as defining tree using Tree-Adjoining Grammar (TAG) [69] and defining strings using context-free grammar. Then we define a workflow search result as a *minimal view* of a workflow that contains query keywords.

Second, we develop efficient algorithms for query result generation. Unlike generating search results on graphs/trees, where only extractions of source data are needed, WISE dynamically constructs query results by *synthesizing* the dataflows among keyword matches. Given the workflow hierarchies containing all keywords in the query, the algorithm for generating results has optimal time complexity.

Experiments show the effectiveness and efficiency of WISE, compared with existing workflow search engines [2, 9, 7, 4] and a search engine on graph data [58].

Although the running example for WISE is a simple recipe workflow hierarchy, the techniques that we propose is applicable for all workflow hierarchies. Such a multi-resolution data structure is a generalization of graphs and trees, and is widely used in many domains, such as scientific experiments, web services, spatial and temporal data, hierarchical plans, etc. For instance, in spatial data, there are edges among the data points in a graph, and a graph can be abstracted to a data point in a recursive way.

13.2 Workflow Hierarchy

In this section, we formally define workflow and workflow hierarchy.

Definition 13.1 (workflow) *A* workflow $W = (V, E)$ *is a directed graph where each node represents a task and each edge indicates the dataflow, dependency or control flow between two tasks.*

The bottom box in Figure 1.7 shows a workflow of curry chicken recipe.

Definition 13.2 (composite task) *A* composite task c *is an abstraction of a group of tasks* S*, denoted as* $c = abs(S)$*.*

If $c = abs(S)$, c is called the *parent* of the nodes in S, and nodes in S are called the *children* of c. Ancestors and descendants are recursively defined.

In Figure 1.7, each group of tasks within a dotted border is abstracted into a composite task, pointed to from the group of tasks by a dotted edge. Composite tasks can be recursively defined.

Definition 13.3 (workflow hierarchy) *A* workflow hierarchy $H = (W, root)$ *consists of a workflow* $W(V, E)$ *and a set of composite task specifications. Nodes in* V *are called* atomic tasks *or* leaf tasks. $root \in V$ *is the root of the hierarchy, whose name represents the name/goal of the workflow hierarchy. The edge set* $E = \{E_a, E_d\}$ *consists of both* abstraction edges *(*E_a*) and* dataflow edges *(*E_d*). An abstraction edge (*S*,* $abs(S)$*)* $\in E_a$ *connects a set of nodes to their corresponding composite task. A dataflow edge (*u*,* v*,* d*) represents that an output of task* u *is an input of task* v*, where* $u, v \in V$*, and* d *denotes the data item sent from* u *to* v*.*[1] *A* subworkflow hierarchy *of* H *is a workflow hierarchy whose root is a node in* V*.*

Note that two composite tasks do not overlap, i.e., $\forall c = abs(S)$ and $c' = abs(S')$, $c \neq c'$, $S \cap S' = \emptyset$. If $c = abs(S)$ and $u \in S$, we say S is the *cluster* of u.

Each node in a workflow hierarchy can have annotations, which record its name, conditions of the task execution, or possibly a deadline, indicating that the task must be

[1]For edges that are from or to an external node of the workflow hierarchy, its corresponding u or v nodes are captured by dummy nodes added to the workflow.

finished no later than the deadline during the execution of the workflow (referred to as "Event-based workflow"), etc. The data items transferred between tasks can be of different types, such as materials in the recipe, data files in the experiment, gene sequences, etc. Note that there can be multiple dataflow edges between two nodes if multiple data items are transferred. There can also be annotations on edges, which specify the control flow (AND/OR/XOR) between two tasks.

Figure 1.7 shows a workflow hierarchy. The bottom box is a workflow, and each dotted edge represents a composite task specification. Note that the edges that involve composite tasks (e.g., the edge from 0.1.2 to 0.2) are not part of the workflow hierarchy, but their relationship can be derived from the edges between atomic tasks, which is illustrated in Definition 13.5.

Although a workflow hierarchy bears some similarity with a tree model, they have some key differences. First, the relationships among "sibling nodes" are different. Siblings in a tree structure are modeled as either a linearly ordered list or a set; whereas the siblings in a workflow hierarchy represent a (possibly cyclic) graph, where the dataflow edges explicitly capture their relationships. Second, the semantics of parent-child relationship are different. A parent-child relationship in a tree generally specifies the relationship between two distinct objects. In a workflow hierarchy, the task represented by a child is part of the detailed procedure of performing the task represented by the parent. These differences invalidate techniques for keyword search on trees/graphs, pose unique challenges to query processing and demand novel approaches.

13.3 Search Results of WISE

Now we discuss how to define query results for keyword search on workflow hierarchies. Each result should satisfy three properties: (1) informative: the result should contain all dataflows between any two tasks matching keywords, so that the user gets the relationships of the query keywords; (2) self-contained: the result should contain all the tasks for achieving a goal; (3) concise: removing any edges from the result will make it violate informativeness or self-containedness.

To achieve these goals, we first identify the minimal workflow hierarchies that contain all query keywords, then define its minimal views as query results.

Identifying Minimal Workflow Hierarchies

To be *informative* and *concise*, we first identify the smallest workflow hierarchies in the repository that contain at least one match to each query keyword.

Definition 13.4 (Minimal Workflow Hierarchy) *A* minimal workflow hierarchy $H = (V, E, root)$ *of a keyword search Q on a repository of workflows R is a workflow hierarchy, such that*
1. Every keyword in Q has at least one match in H;
2. There does not exist a subworkflow hierarchy of H that satisfies condition 1.

Note that there are typically multiple minimal workflow hierarchies when processing a keyword query on a repository of workflows. Each minimal workflow hierarchy will derive a query result, as to be discussed later, all of which compose the set of results for the query.

For example, consider Q_2 "*brown rice, bake*" in Fig. 1.5(a). There are two workflow hierarchies in Figure 1.7 containing keyword matches *brown rice* and *bake*: the one rooted at *curry chicken* (0) and the one rooted at *make rice pilaf* (0.3). The one rooted at *curry chicken* (0) is not considered as a minimal workflow hierarchy as it does not satisfy condition 2 in Definition 13.4: it has a subworkflow rooted at *make rice pilaf* (0.3) containing all query keywords.

Note that we do not "compose" a new workflow hierarchy from several unconnected sub-workflow hierarchies in the repository, in order to guarantee that each query result has a clear semantic meaning.

Defining Query Results as Minimal Views

Minimal Views

Unfortunately, returning the whole minimal workflow hierarchy itself to users is neither *informative* nor *concise*. Consider Q_1 as an example, where a minimal workflow hierarchy is Fig. 1.7. Returning the entire *curry chicken* hierarchy makes it difficult for users to find the dataflows among keyword matches, as they have to go up and down the layers and manually construct the dataflows. These manual operations are tedious and time-consuming for the users especially when the workflow hierarchy is large and complex.

Under this observation, we define the notion of *view* of a workflow hierarchy. Views can be considered as a projection of the 3D workflow hierarchy on to a 2D plane which hides less important information and simplifies analysis.

Definition 13.5 (View) *A view $View = (V, E)$ of a workflow hierarchy $H = (V', E', root)$ is a directed graph with labels on edges. The node set V, $V \subseteq V'$ satisfies:*
1. There does not exist $u, v \in V$, u is an ancestor of v in H.
2. $\forall u \in V'$ and u is a leaf node, $\exists v \in V$ such that v is an ancestor-or-self of u.
3. For any $u, v \in V$, $(u, v, d) \in E_d$ if and only if $\exists u', v' \in V'$, u' and v' are descendant-or-self of u and v, respectively, and $(u', v', d) \in E'_d$.

Condition 1 indicates that a view is a two dimensional projection of the three dimensional workflow hierarchy, flattening out the nested hierarchy for the ease of user comprehension. Nodes in a view can have dataflow relationships, but not abstraction relationships. Condition 2 ensures that the node set V of a view covers all leaf nodes (atomic tasks) in the workflow hierarchy H: every leaf node in V' must have one corresponding zoomed-out node in V. Since views may contain composite nodes whose edges are not explicitly present in the workflow hierarchy, their edges need to be induced, as specified in Condition 3. For example, since there is a dataflow between *wait 10 min* (0.1.0.3) and *put into skillet* (0.1.1.0), there should also be a dataflow between their parents, *tenderize chicken breast* (0.1.0) and *concoct* (0.1.1). Condition 3 guarantees that the dataflow edge set E_d in a view is faithful with respect to edge set E'_d according to H: the view preserves the dataflow among nodes in the view. Note that a view may hide the dataflows of two nodes in the workflow that are abstracted into a single node in the view, e.g., the edge between nodes 0.1.0.0 and 0.1.0.3 in Figure 1.5(a). Conditions 2 and 3 together ensure that a view is "semantically complete" with respect to the name/goal of H, and hence a self-contained information unit whose name/goal is the same as H.

Note that Definition 13.5 bears some similarity with the TAG [69] and context-free grammars. Context-free grammars have rules for rewriting symbols as strings of other symbols, tree-adjoining grammars have rules for rewriting the nodes of trees as other trees, and the proposed workflow views allow rewriting the nodes of a workflow as other graphs. That is, a view is a graph defined on a nested graph hierarchy, analogous to the frontier defined on a tree in TAG, and to a string defined in a context-free grammar.

As we can see, a view is a projection of a three dimensional workflow hierarchy to a two dimensional plane that preserves the dataflows among the tasks in the view. Obviously a workflow hierarchy can have many views, as there are many projections of a three dimensional object, depending on the viewing plane. For example, the tasks in each solid rectangle in Figure 1.7 compose a view of the *curry chicken* workflow hierarchy. Fig. 1.5(c) is another view of *curry chicken*.

We now define a keyword search result on workflow hierarchies as a minimal view of a minimal workflow hierarchy that preserves all keyword matches in the workflow hierarchy (which is also a philosophy of keyword search on relational databases or XML, where a result is a minimal tree that contains at least one match to each keyword).

Definition 13.6 (Minimal View) *For a keyword search Q on a repository of workflows R, the* minimal view $View(H, Q) = (V, E)$ *of a minimal workflow hierarchy H (Definition 13.4) is the view with the smallest number of tasks over all the views of H that contain all the keyword matches of Q in H.*[2]

Definition 13.7 (Query Result) *For a keyword search Q on a repository of workflows R, the set of query results consists of the minimal view of each minimal workflow hierarchies in the repository.*

Note that such a result definition is general for all types of workflow hierarchies where the nodes and edges may have annotations as discussed in Section 13.2. Given a

[2]Note that since the nodes in a view can not have ancestor-descendant relationships, only keyword matches that do not have descendant keyword matches are selected as nodes in a view, which are annotated with their ancestor keyword matches (if any).

minimal workflow hierarchy and a query, the minimal view is unique, which can be considered as a projection of a three dimensional workflow hierarchy on a two dimensional viewing plane defined by the query. The result is *informative* since it captures all keyword matches and their dataflows. The result is *self-contained* since the view serves an integrated goal and has a unique name, which is the same as the corresponding minimal workflow hierarchy. Furthermore, the result is *concise*, as we opt to use the *minimal view* among all views of each minimal workflow hierarchy.

13.4 Algorithms

After defining query results for keyword search on workflow hierarchies, we present the algorithms of the WISE system that achieve the semantics efficiently.

Data Processing

We design labeling schemes and indexes for workflow hierarchies to efficiently find minimal workflow hierarchies and their minimal views.

Labeling of nodes and edges. Each node n in the workflow hierarchy is assigned a unique label $NID(n)$. Since we need to explore the ancestor-descendant relationships of nodes to generate results, we use the Dewey labeling scheme, as shown underneath each node in Fig. 1.7. The label of the root is 0, and the label of a node n is composed by the concatenation of the label of its parent and a unique integer ID within the cluster that n is in. The unique in-cluster ID of a node can be arbitrarily set, i.e., the nodes in a cluster can have an arbitrary order, independent of the dataflows (thus nodes can be ordered even in a cyclic graph). NIDs of nodes are ordered alphabetically. The node labels don't record dataflow information, but parent-child information. They enable efficient retrieval of lowest common ancestor (LCA) of two nodes u and v, whose node label is the longest common prefix of $NID(u)$ and $NID(v)$. Each edge is also assigned a unique integer ID.

Leaf adjacency lists of nodes. To efficiently find dataflows among keyword matches that may not be explicitly present in the data, we build a *leaf adjacency list* for each node n in the workflow hierarchy, denoted as $LAL(n)$. $LAL(n)$ consists of IDs of the edges between leaf nodes u and v, such that u is a descendant of n and v is not a descendant of n. For each edge in $LAL(n)$, we also record its direction, as well as the data items transferred. For example, suppose the ID of the edge from node 0.1.0.3 to node 0.1.1.0 in Figure 1.7 is e_1 and it transfers data item *chicken breast*, then e_1 (outgoing, chicken breast) $\in LAL(0.1.0.3)$ and $LAL(0.1.0)$, where "outgoing" means that it is an outgoing edge from 0.1.0.3 to 0.1.0. Similarly, e_1 (incoming, chicken breast) $\in LAL(0.1.1.0)$ and $LAL(0.1.1)$. Leaf adjacency lists are used to efficiently derive dataflow edges in a query result, as will be discussed later. Intuitively, according to Definition 13.5 there is a dataflow edge between two composite nodes u and v if and only if there is an edge e between their leaf descendants, and such an edge e is recorded in $LAL(u)$ and $LAL(v)$. By leveraging leaf adjacency lists and a hash table, an edge can be derived in O(1) time.

Indexes. To speed up query processing, an inverted index is built which maps a keyword to the list of nodes in the workflow repository whose names/descriptions contain the keyword, sorted by their NID. We also build a B+ tree index on NIDs that retrieves the subworkflow rooted at node NID, referred to as Dewey index.

The leaf adjacency list and indexes are built offline. They both take an affordable amount of space: in the worst case, each dataflow edge is recorded in every ancestor of each endpoint of the edge. Thus the leaf adjacency list takes $O(|E_d|h)$ space where $|E_d|$ is the number of dataflow edges in the workflow hierarchy, and h is the height of the workflow hierarchy. If each node contains at most p keywords, then the inverted index takes $O(|V|p)$ space where $|V|$ is the number of nodes in the workflow hierarchy. The Dewey index takes $O(|V|)$ space.

Query Processing

WISE uses Algorithm 16 to retrieve relevant query results for keyword searches on workflow hierarchies. It consists of two steps: identifying minimal workflow hierarchies, and constructing the minimal view for each minimal workflow hierarchy. We use Q_1: "*chicken breast, coconut milk, saute*" as a running example.

Retrieving minimal workflow hierarchies. We begin by obtaining the list of match nodes for each keyword using the inverted index. In our running example (Q_1), we obtain the matches to *chicken breast*: 0.1.0, *coconut milk*: 0.2.2 and *saute*: 0.1.1.2.

Note that sometimes a user may issue a query whose keywords do not exactly match the words in the data, but are semantically related. This can be addressed by looking each keyword up in a dictionary of synonyms. For instance, if the user query contains keyword “saute”, we look it up in the dictionary and find its synonyms, e.g., “fry” and “panfry”. Then we search the inverted index for the matches to “saute”, “fry” and “panfry”, and take the union of their matches as the matches to keyword “saute”. In the experiments we have tested the efficiency of WISE when synonyms are considered in query processing. Alternatively, we can also use an ontology, which not only records the similarity among keywords but also the containment relationships (e.g. “saute” is a special type of “cook”). Furthermore, the similarity measurement can be used as part of the ranking scheme.

Then procedure $findMWHs$ identifies the minimal workflow hierarchies using the Indexed Lookup Eager Algorithm [144], considering only expansion edges without dataflow edges. In our running example, there is only one minimal workflow hierarchy, which is the entire *curry chicken* hierarchy, as none of the descendants of *curry chicken* (0) contains all three query keywords. As discussed, returning the entire minimal workflow hierarchy is not informative or concise, thus we propose novel algorithms for computing minimal views of each minimal workflow hierarchy.

Identifying minimal views of minimal workflow hierarchies. After identifying minimal workflow hierarchies, procedure $grouping$ groups the keyword matches according to the minimal workflow hierarchies that they belong to. This is done by first merging the lists of keyword matches into a single list $mergedList$, and then grouping the matches using a single traversal of $mergedList$ and the list of the roots of minimal workflow hierarchies.

Finally, we need to identify minimal views of minimal workflow hierarchies. For each task in the minimal workflow hierarchy, we need to determine whether to include it in the view or not, and extract or synthesize the dataflow among the nodes in the view. $genMV$ performs a single depth-first traversal of each minimal workflow hierarchy in the order of NID, and a traversal on the list of keyword matches sorted by NID. Let cn be the node

Algorithm 16 Keyword Search on Workflow Hierarchies

```
keywordSearch ($keyword[n]$, indexes)
 1: for $i$ = 1 to $n$ do
 2:     $matches[i] = word2NID(keyword[i])$
 3: $roots[r] = findMWHs(matches)$ [144]
 4: $matchGroup[r] = grouping(matches, roots)$
 5: for $i$ = 1 to $r$ do
 6:     $result[i] = genMV(roots[i], matchgroup[i], indexes)$
grouping ($matches[n][p], roots[r]$)
 1: $mergedList[m]$ = merge-sort $matches[n][p]$ into a sorted list
 2: $i = j = 1$
 3: $matchgroup[i] = \emptyset$ for all $1 \leq i \leq r$
 4: while $i \leq r$ and $j \leq m$ do
 5:     if $ancestor-or-self(roots[i], mergedList[j])$ then
 6:         $matchgroup[i] = matchgroup[i] \cup mergedlist[j]$
 7:         $j++$
 8:     else if $matchgroup[i]! = \emptyset$ then
 9:         $i++$
10:     else
11:         $j++$
genMV ($root, matchgroup[g]$, indexes)
 1: $cn$ = root; $cm = 1$; $mv = \emptyset$
 2: $EdgeHash$ = a hash table initialized as empty
 3: while $cn \neq$ null and $matchgroup[cm] \neq$ null do
 4:     if $cn = matchgroup[cm]$ then
 5:         if $ancestor(cn, matchgroup[cm+1])$ then
 6:             {$cn$ is a match node and has descendant matches}
 7:             $cm++$
 8:             continue
 9:         else
10:             {$cn$ is a match node and has no descendant match}
11:             $outputnode(mv, cn, EdgeHash)$
12:             $cn = cn$'s next node in $NID$ order, which is not a descendant of $cn$
13:     else if $ancestor(cn, matchgroup[cm])$ then
14:         {$cn$ is not a keyword match and has descendant matches}
15:         $cn = cn$'s first child
16:     else
17:         {$cn$ is not a keyword match and has no descendant match}
18:         $outputnode(mv, cn, EdgeHash)$
19:         $cn = cn$'s next node in $NID$ order, which is not a descendant of $cn$
20: return $mv$
outputnode ($mv, cn, EdgeHash$)
 1: add $cn$ into $mv$
 2: for each edge entry $eid(incoming/outgoing, d) \in LAL(cn)$ do
 3:     if there is an entry $(eid, u)$ in the $EdgeHash$ then
 4:         {$u$ is the other endpoint of $eid$ that has been output}
 5:         add an edge from $u$ to $cn$ (or from $cn$ to $u'$) into $mv$ with data item $d$
 6:     else
 7:         insert an entry $(eid, cn)$ into $EdgeHash$
```

in the workflow hierarchy currently being visited and $currMatch$ be the current keyword match being visited in $mergedList$. During the traversal:

(1) If cn has descendant matches (which is true if cn is an ancestor of $currMatch$, or $cn = currMatch$ and is an ancestor node of the next node in $mergedList$), we do not output cn, but update cn to be the first child of the current cn, i.e., continue to traverse its subworkflow hierarchy. If cn matches a keyword, $currMatch$ is updated to be the next keyword match.

(2) If cn is not a keyword match and has no descendant match (which is true if cn is not an ancestor-or-self of $currMatch$), then we output cn, skip its subworkflow hierarchy and move to the next node in the workflow hierarchy which is not a descendant of cn.

(3) If cn is a keyword match and does not have descendant matches (which is true if $cn = currMatch$, and is not an ancestor of the next match node), then cn is directly interested by the user, and is output as part of the query result. The properties of match nodes can be displayed upon click. Since we do not output the expansion of cn, we move to the next node in the workflow hierarchy which is not a descendant of cn. We also move $currMatch$ to point to the next keyword match in $mergedList$.

In our running example, we have found the minimal workflow hierarchy, rooted at *curry chicken* (0). The keyword matches in the order of their NID are: *chicken breast* (0.1.0), *saute* (0.1.1.2), *coconut milk* (0.2.2). We traverse the minimal workflow hierarchy and the keyword match list in parallel. Initially, cn = *curry chicken* (0) and $currMatch$ = 0.1.0. Since cn is an ancestor of $currMatch$, we do not output *curry chicken*, but expand it and traverse its children. Later on when we come to cn = 0.1.0, since $cn = currMatch$, we output cn as a clickable node, then move cn to 0.1.1 and $currMatch$ to *saute* (0.1.1.2). The procedure continues until all nodes in the results are identified.

Next we discuss how to generate edges in the result. When outputting a node cn, we need to find the edges corresponding to cn. A naive approach would search each leaf descendant of every node u that has been output as well as each leaf descendant of cn, and check whether there is an edge between them. If so, it means an edge should exist between u and cn in the view (as discussed before, a view should preserve the edges between two nodes in the workflow hierarchy that are descendants of different nodes in the view). This approach is very inefficient, as u and cn may both have a large number of descendants, and they may be accessed multiple times.

We propose a much more efficient approach using LAL and a hash table, which will be shown to find each edge in $O(1)$. When outputting a node cn we traverse $LAL(cn)$; for each edge with ID e_i in $LAL(cn)$, we check it in a hash table, which maps an edge ID to an endpoint of the edge that has been output. The hash table is initially empty. If e_i is not in the hash table, it means the other endpoint of e_i has not been output, and we put an entry (e_i, cn) into the hash table. If e_i is in the hash table with entry (e_i, u), then u has been output, and there should be a dataflow edge between u and cn, whose direction and data item depends on the corresponding entry in $LAL(cn)$.

For example, when we output *tendrize chicken breast* (0.1.0), we check its LAL. Suppose there is an edge from 0.1.0.3 to 0.1.1.0 with ID e_1 and data item *chicken breast*, then $LAL(0.1.0) = \{e_1$ (outgoing, chicken breast)$\}$. Since node 0.1.1.0 has not been output yet, e_1 is not in the hash table, and we insert entry $(e_1, 0.1.0)$ into the hash table. When we output *put into skillet* (0.1.1.0), since $LAL(0.1.1.0) = \{e_1$ (incoming, chicken breast)$\}$, we check e_1 in the hash table, and get the entry $(e_1, 0.1.0)$. Therefore, we output an edge from 0.1.0 to 0.1.1.0 with data item “chicken breast”.

Theorem 13.1 *The results generated by Algorithm 16 for a keyword search Q on a repository of workflows R are the minimal views of all minimal workflow hierarchies $H(R, Q)$ in the repository (Definition 13.7).*

Proof. We adopt the Indexed Lookup Eager Algorithm [144] to compute minimal workflow hierarchies. In the following, we prove that a query result $QR = (V, E)$ generated by Algorithm 16 is the minimal view of a minimal workflow hierarchy that contains all keyword matches which do not have descendant matches.

First, we prove that QR is a view of $H(R, Q)$, i.e., it satisfies the three conditions of view in Section 13.2.

1. $\forall u \in H$, if $u \in QR$, then none of u's children is output in QR according to Algorithm 16 (recall that after we output a node cn, we move to the next node that is not a descendant-or-self of cn). Therefore, QR satisfies condition 1 in Definition 13.5.

2. $\forall u$, u is a leaf of H and $u \notin QR$, according to Algorithm 16, during the depth-first traversal of H, u must have an ancestor u' which is visited and output in QR (recall that we do not output a node if and only if it is expanded, or an ancestor node is output). In other word, u' is a node and has no descendant matching keywords. Therefore, QR satisfies condition 2 in Definition 13.5.

3. $\forall u, v \in V$, according to Algorithm 16, there is an edge from u to v, (u, v, d), if and only if the ID of the edge (e.g., e_i) is recorded in both $LAL(u)$ and $LAL(v)$. According to the design of leaf adjacency list, this happens if and only if there is an edge (u', v', d) between a descendant u' of u and a descendant v' of v labeled e_i. Therefore, an edge (u, v, d) between u and v is output in the view if and only if there exists an edge (u', v', d) between u' and v'. Thus QR satisfies condition 3 in Definition 13.5.

Next we prove that QR contains all keyword matches in the minimal workflow hierarchy that have no descendant keyword match. For any keyword match node m, each of m's ancestors will be visited and expanded during the traversal of the minimal workflow hierarchy, as it has a descendant match. Therefore, m will be visited and output, as all children of an expanded node will be visited.

At last we prove that QR generated by Algorithm 16 is minimal, i.e., it has the smallest number of nodes among all views of H that contain all keyword matches. Suppose there is another view of H, $QR' = (V', E')$ which has a smaller number of nodes than QR and contains all keywords. Since both views are from the same workflow hierarchy, there must be at least one composite node u' in H which is expanded in QR but not in QR', i.e., $\exists u \in V$ and $u' \in V'$, such that u is a descendant of u'. According to Algorithm 16, u' is expanded in QR only if it has descendant keyword matches. But since u' is output but not expanded in QR', none of its descendants can be output, thus QR' can not contain all the *keyword matches of H, which is a contradiction. Therefore, QR is the minimal view of H, and is a qualified query result.*

A minimal workflow hierarchy has exactly one minimal view for a query. Thus Algorithm 16 finds exactly the set of query results.

Theorem 13.2 *The time complexity of procedure $genMV$ is $O(N+E)$, where N, E are the number of nodes and edges in the output (minimal view), i.e., the optimal time complexity for finding minimal views. The overall time complexity of Algorithm 16 is $O(M_{min}kd\text{log}M_{max} + M + N + E)$, where M_{min} and M_{max} are the minimum and maximum number of matches to a keyword, respectively, M is the total number of keyword matches, d is the depth of the workflow hierarchy.*

Proof. In Algorithm 16, finding the matches using the inverted index takes $O(M)$ time, where M is the number of keyword matches. Procedure $findMWHs$ adopts the Indexed Lookup Eager Algorithm, which takes $O(M_{min}kd\text{log}M_{max})$ time [144]. The $grouping$ procedure traverses the roots of the minimal workflow hierarchies and the keyword match list, whose complexity is bounded by $O(M)$.

$genMV$ scans each minimal workflow hierarchy, but only visits the ancestor-or-self of the nodes in the minimal views. Since each non-leaf node in the minimal workflow hierarchy has at least 2 children, the total number of nodes visited is no more than $2N$. For each edge in the minimal view, $genMV$ has two operations: insert it into a hash table when the first endpoint is output, and retrieve it from the hash table when the second endpoint is output. Thus each edge takes $O(1)$ time to process. Therefore, the time complexity of $genMV$ is $O(N+E)$, which is the best possible complexity as it is equal to the output size. The overall time complexity of the algorithm is $O(M_{min}kd\text{log}M_{max} + M + N + E)$.

13.5 Experiments

To evaluate the effectiveness and efficiency of WISE, we compare its performance with three methods in the literature. *Node Return* outputs individual tasks that contain query keywords without dataflow information, as supported by the search modules in Kepler [2],

Biology	
QB_1	GenBank
QB_2	Get Sequence, Filter
QB_3	Get Promoters, Align
QB_4	Align, Blast, Get Promoters
QB_5	Filter, Synchronizer
QB_6	Record Updater, Filter
QB_7	Blast, Get Sequence, Merge
QB_8	Array Merge, Align
QB_9	Record Updater, Record Disassembler
QB_{10}	Align, Filter, Synchronizer
Geography	
QG_1	SVG Concatenate
QG_2	ExtractURL, ExtractShpURL
QG_3	WebService, RecoradDisassembler
QG_4	ClassifySample, extractAge
QG_5	ClassifySample, AssertPoint
QG_6	ClassifyBody, SVG To Polygon Converter
QG_7	Composite Actor, SVG Concatenate
QG_8	Add Point To SVG, QueryBodyAge
QG_9	Get 2D Point, Record Disassembler, Render Mapler
QG_{10}	Record Disassembler, SVG To Polygon Converter
Ecology	
QE_1	Add Grids
QE_2	GarpPrediction, GarpAlgorithm
QE_3	Future-Climate-Model, IJMacro
QE_4	LocationFile, I - DataPoints
QE_5	Create ASC Maps , Garp Prediction
QE_6	Calculate Best Rulesets , CV Hull to RasterMask
QE_7	IJMacro, II - EnvLayerSet
QE_8	Add Grids , GarpAlgorithm
QE_9	Future-Climate-Model,ConvexHull, GarpPrediction
QE_{10}	Create ASC Maps, Add Grids, I - DataPoints

Figure 13.2: Query Sets in Sample Scientific Domains for Testing WISE

Triana [9] and Taverna [7]. *Structure Return* outputs a whole workflow hierarchy if it contains all the query keywords, which is used in myExperiment [4]. The third approach, BLINKS [58], is a state-of-the-art keyword search engine on graphs, which output minimal subtrees in the data graph whose leaf nodes contain matches to query keywords. We implemented Structure Return and Node Return approaches with best-effort, each of which only returns information of the minimal workflow hierarchies for better precision. The implementation of BLINKS is obtained from the authors.

We have tested two aspects: the quality of search results measured by the amount of information returned, as well as user perceived precision, recall and F-measure; the

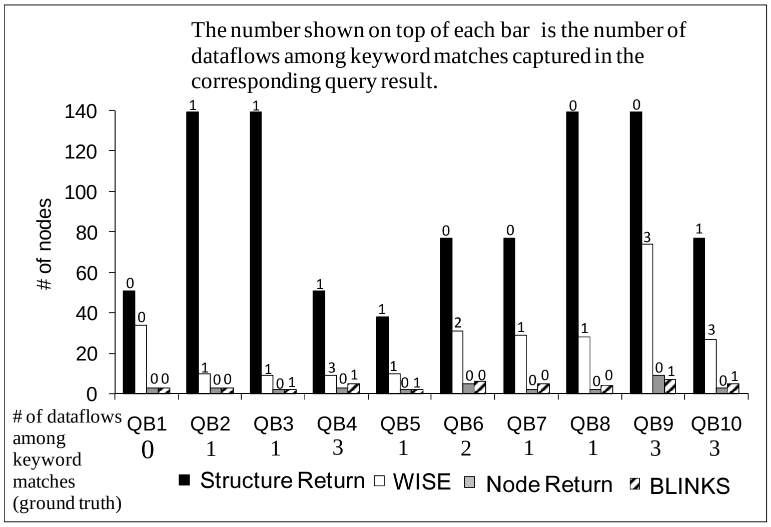


Figure 13.3: Number of Nodes vs. Number of Dataflows among Keyword Matches in the Query Results

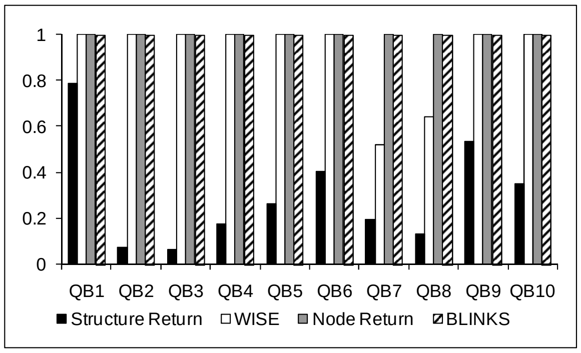


Figure 13.4: Precision of Workflow Search

efficiency of the search algorithms measured by processing time and scalability over data size.

In the efficiency test, we additionally test the efficiency of WISE in handling synonyms of keywords. Specifically, we use WordNet [10] to find a set of synonyms for each keyword, then take the union of the matches to all synonyms as the matches to this keyword.

Experimental Setup

The experiments were performed on a 3.6GHz Pentium 4 machine. The systems were implemented in Java using a commercial database as the backend. All experiments were

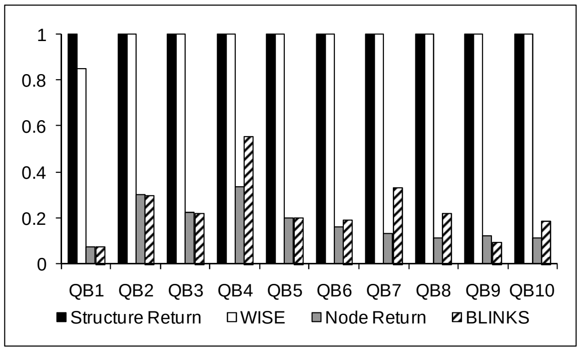


Figure 13.5: Recall of Workflow Search

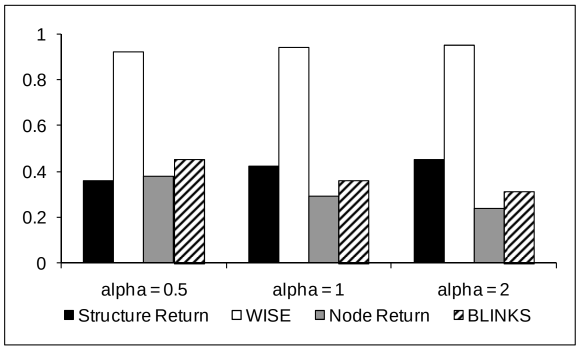


Figure 13.6: F-measure of Workflow Search

repeated 5 times independently with cold cache, and we report the average processing time discarding the maximum and minimum values.

Data Set. The data are obtained from the Kepler system[2] in MoML (Modeling Markup Language) format [3], which is the standard file format for specifying workflow hierarchies, widely used in many workflow systems such as Kepler [2], SEEK [6], GEON [1], etc.

Query Set. We have tested thirty queries for workflows in three sample application domains: Biology ($QB_1 - QB_{10}$), Geology ($QG_1 - QG_{10}$), and Ecology ($QE_1 - QE_{10}$). Queries QB_1 to QB_{10} are set by a biologist from the Biology department in Arizona State University. For example, QB_1 intends to find out the usage of *GenBank*, and QB_2 intends to find out how to filter sequences. The queries include single keyword queries, queries whose keywords appear in the same cluster, same layer, or different layers. All queries can be found in Figure 13.2.

Analysis of Information in Query Results. Fig. 13.3 shows the total number of nodes in the query result, the ground truth of the number of dataflows between keyword matches in the data, and the number of dataflow paths that are captured in the result, of each approach on queries QB_1 to QB_{10}. We can see that these approaches in the decreasing order of the amount of nodes output are: Structure Return, WISE, BLINKS, and Node Return. Each result of each approach contains all keywords of the corresponding query. However, the numbers of dataflow paths among keywords captured by these approaches are quite different, as shown above the bars in Fig. 13.3. The total numbers of dataflows between all pairs of distinct keywords in the data are considered as the *ground truth*, which are listed below the x-axis. Node Return has zero dataflows returned as no pair of the nodes in a query result are connected. BLINKS is unaware of the difference of dataflow edges and expansion edges and thus often fails to capture the dataflow paths between keywords. Structure Return outputs the entire minimal workflow hierarchies as query results, which only explicitly capture the dataflow paths among keyword matches that are leaf nodes. Even though it outputs much more data nodes than WISE, the amount of relevant information is limited. On the other hand, WISE can capture all the dataflows by explicitly displaying the dataflow paths connecting keywords in the query results, and thus return *informative* results.

We also evaluate the number of self-contained results generated from all four methods for QB_1 - QB_{10}. All results generated by WISE and Structure Return are self-contained, as they generated minimal views / minimal workflow hierarchies as results. Node Return does not generate any self-contained results. BLINKS generates self-contained results only when it happens to return exactly the nodes and edges in a cluster as a result. For QB_1 - QB_{10}, BLINKS does not generate self-contained results.

User Evaluation. To further verify the rationale of WISE's semantics and the acceptance of WISE's query results by users, we also performed a user study on QB_1 to QB_{10} to measure the *precision*, *recall*, and *F-measure* of these four approaches.

Ten students who are not aware of this project were invited for the survey. For each query, we provided the users with the search results generated by each of the four systems, as well as an option for them to specify their own query results if none of them are satisfactory, by circling all the nodes they wish to be returned (Recall that dataflow information is analyzed in Fig. 13.10). The ground truth is set based on the majority of agreements by the users. We then calculate the precisions and recalls of each approach on the ten queries based on the ground truth, which are shown in Fig. 13.4 and 13.5.

As we can see, Structure Return usually has a perfect recall as the entire minimal workflow hierarchies are returned for each query. However, it suffers a low precision as not all the nodes returned are relevant. Take QB_2 as an example, the users are only interested in the information about *Get Sequence* and *Filter*, which comprises only a small portion of the minimal workflow hierarchy. On the other hand, Node Return has a perfect precision on all queries, but suffers a very low recall. Consider QB_4, no information about how to use *Align*, *Blast* and *Get Promoters* together is returned. Similarly, BLINKS has a perfect precision, but low recall, since the results that are returned are generally not self-contained workflows.

WISE has both high precision and recall in general. There are a few queries on which WISE's search quality can be further improved. For query QB_1, WISE has a low recall because WISE outputs the cluster containing *GenBank*, and task *GenBank* is clickable but not expanded. However, the users prefer having the detailed information of *GenBank* directly. The reason WISE has a low precision for QB_8 is that the occurrences of the keywords *Array Merge* and *Align* in the data are far away from the start tasks of the minimal workflow hierarchy. In this case, the users prefer omitting some portion of the data from the start tasks to the keyword matches in the result for conciseness reason. QB_7 has the similar reason as QB_8.

We compute the F-measure of each approach according to the average precision and recall across all the test queries, with parameter α= 0.5, 1 and 2, as presented in Fig. 13.6. WISE significantly outperforms Structure Return, Node Return and BLINKS.

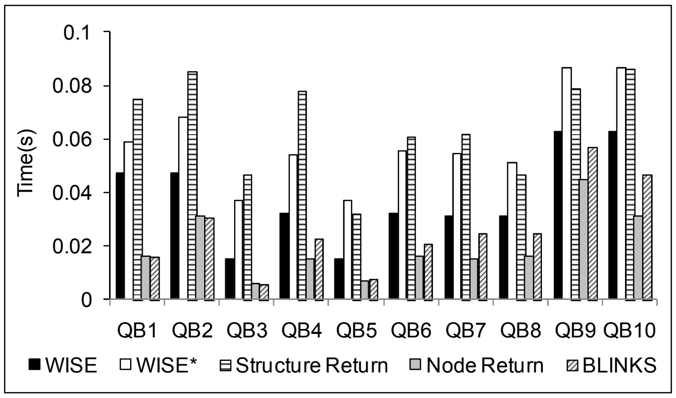


Figure 13.7: Query Processing Time of Workflow Search on Biology Data

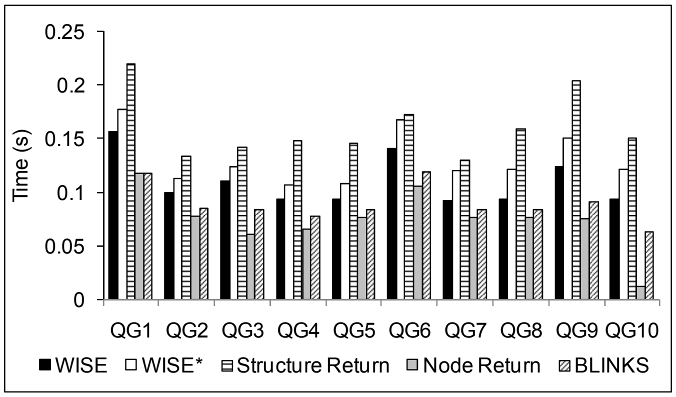


Figure 13.8: Query Processing Time of Workflow Search on Geography Data

Efficiency

Processing Time. The processing times of WISE, Structure Return, Node Return and BLINKS on the test data and query sets are shown in Fig. 13.7 - 13.9. WISE* denotes the approach that incorporates WordNet for finding synonyms of keywords. All approaches are efficient. Node Return is the fastest, followed by BLINKS and WISE, and Structure Return is the slowest. WISE* takes additional processing time for finding the synonyms of each keyword, as well as finding the matches to them. It can be seen that the additional time WISE* takes to handle synonyms is fractional.

Fig. 13.10(a) shows the breakdown of the average processing time of each approach over all 30 queries. Structure Return, Node Return and WISE start with finding the minimal workflow hierarchies; WISE further finds the minimal views; BLINKS has specific

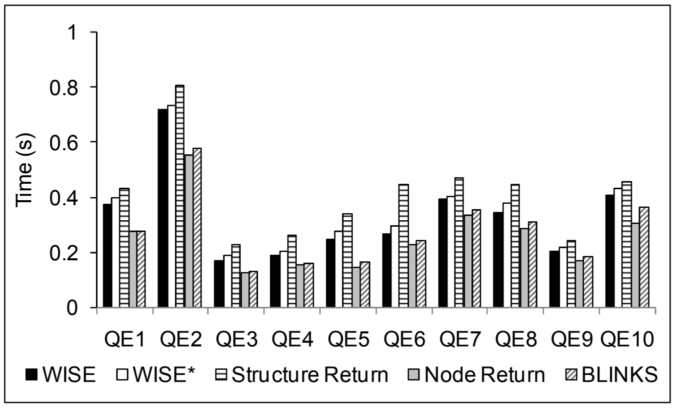
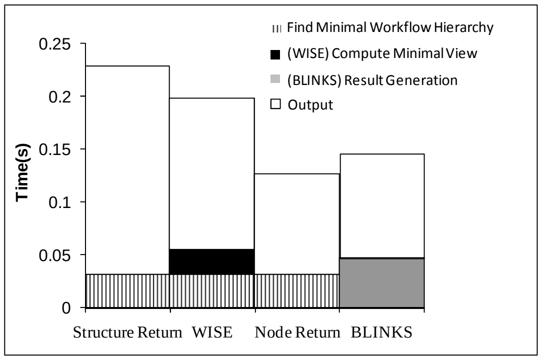
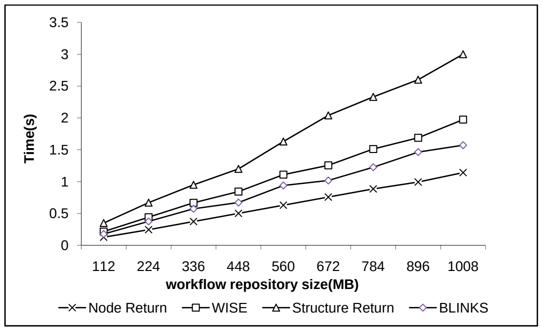


Figure 13.9: Query Processing Time of Workflow Search on Ecology Data



(a) Processing Time Breakdown



(b) Scalability

Figure 13.10: Workflow Search Processing Time Breakdown, Scalability and Number of Dataflows Captured

result generation algorithms; and all consume time for outputting results.

As we can see, the output time is dominant in the overall cost, which is determined by the amount of information output. Fig. 13.3 shows the number of nodes that are output by each approach for queries on Biology workflow repository. Structure Return, which outputs the whole minimal workflow hierarchies, has the largest output sizes and thus is the slowest. Its output size is followed by that of WISE, and then BLINKS. On the other hand, Node Return, which only outputs individual match nodes, has the smallest number of nodes returned and therefore is fastest.

We also observe that the algorithm of WISE for generating minimal view is very efficient, consuming a very small portion of its processing time. The result generation time of WISE is similar to those of BLINKS and Node Return. The processing overhead of WISE is mainly due to the additional information output, which captures more dataflows between keyword matches (shown in Fig. 13.3) and results in best search quality among all (shown in Fig. 13.4 and 13.5).

Scalability. We test the scalability of all four approaches over increasing data sizes by replicating the workflows in the repository multiple times. The processing times of QB_1 are shown in Fig. 13.10(b). All four approaches increase linearly when the data size increases. They scale similarly on other queries and the figures are omitted.

In summary, WISE achieves significantly better search quality compared with Structure Return, Node Return and BLINKS and returns self-contained, informative yet concise query results. It is efficient and scales well.

13.6 Summary

In this chapter we present WISE, a keyword search engine for workflow hierarchies, which are modeled as hierarchies of multiple layers. Due to the three dimensional data structure, existing techniques for searching trees or graphs are unhelpful for searching workflow hierarchies. To identify self-contained, informative and concise query results on workflow hierarchies, we identify the minimal views of minimal workflow hierarchies as query results, which captures the dataflows between keyword matches. We then designed an algorithm to find minimal views of a minimal workflow hierarchy with optimal time complexity. Experimental evaluations have shown the effectiveness and efficiency of WISE.

Chapter 14

Conclusions and Future Work

14.1 Conclusions

Keyword search is a very desirable way for casual users to access structured data. Compared with structured queries, keyword search is easy to use by casual users, and it enables the discovery of unexpected and interesting information. Compared with text search, searching structured data provides us with more opportunities to return meaningful results. By supporting keyword search on structured data, we can significantly enhance the usability of structured data and make it accessible by a large population of users. However, due to the ambiguity of keyword queries and the large search space, effectively and efficiently supporting keyword searches on structured data has many challenges.

In this dissertation we consider three structured data models: tree data, graph data and nested graph data. We identify various challenges in supporting keyword search on these data models and propose techniques to address each of them. The challenges and our solutions discussed in this dissertation include:

Structural Ambiguity: keyword queries have no structures, but the user is usually interested in one or a few specific structures. This is a unique challenge in processing keyword searches on structured data. We address this challenge from two perspectives: (1) Inferring structures for keyword queries to generate meaningful results, including identifying relevant keyword matches (Chapter 4), identifying return information (Chapter 5) , composing results based on relevant matches and return information (Chapter 6); (2) Since it is not always possible to generate perfect results, we also resolve structural ambiguity after results are generated by helping users analyze the results, the techniques of which include generating result snippets (Chapter 7) and clustering query results based on their structures (Chapter 9). A result snippet is a subtree of a query result, from which the user can learn the structure of the result. By structure-based result clustering, the user can quickly learn what types of structures exist in the query results.

Keyword Ambiguity: the user may not use the perfect keywords to query the structured data. The keywords may be misspelled, under-specified, over-specified, non-quantitative, etc. This challenge also exists in processing structured queries and in processing text search. However, it is generally an insignificant challenge in processing structured queries as the users of structured queries are usually database experts. Compared with text search, the techniques of resolving keyword ambiguity can be different due to different definitions of query results, and structured data provides more opportunities to better resolve keyword ambiguity.

To resolve keyword ambiguity, one way is to perform query cleaning and rewriting. In this dissertation, we focus on another approach: helping users analyze the query results, including generating result snippets (Chapter 7) and clustering query results based on their values and generating expanded queries from the clusters (Chapter 11). By reading the snippet of a query result, the user will likely understanding the precise meaning of the query keywords. For example, for query “Java”, the user will learn from the snippets that in some results, “Java” refers to a programming language while in some other results, “Java” refers to an island. By clustering and query expansion, the user can also distinguish different meanings of an ambiguous keyword.

User Preference Ambiguity: Many keyword queries are for information exploration purposes, where the user may not have a clear idea of what s/he wants. In this case, the user needs to check multiple results. For example, a user searching for an apparel store may need to check the information of multiple stores to decide which one to visit. Similar as keyword ambiguity, this challenge applies to both processing structured queries and processing text search. It is generally an insignificant challenge in processing structured queries, and compared with text search, structured data provides more opportunities to better resolve user preference ambiguity.

Both result snippets (Chapter 7) and clustering (Chapters 9 and 11) help users get insight from the query results. Besides, we also proposed techniques to generate a comparison table for a set of results (Chapter 8), which consists of features that highlight the differences of these results.

Efficiency: Efficiency is very important for a search engine, and generating keyword search results efficiently involves several unique challenges compared with structured query and text search. Compared to structured query, since the relevant nodes, their connections and the return information are not specified in keyword queries, the search space of keyword queries can be much larger. Compared to text search, the results of keyword search on structured data are not individual documents, but rather subtrees or subgraphs of the data that need to be dynamically identified. Given a structured data and keyword query, the number of ways to connect nodes matching query keywords can be far bigger compared with the size of the data.

To generate results efficiently, we proposed a framework in Chapter 12 that utilizes materialized views to answer keyword queries on XML. We discussed that given a set of materialized views and a keyword query, how to select the best set of views (a.k.a. answer set) to answer the query, and how to use the selected views to answer the query. We also discussed how to incrementally maintain the views when the data is updated.

Searching Nested Graphs: Due to the unique structure of nested graphs, existing methods for defining search results on tree or graph data fail to produce meaningful results on nested graphs. We propose in Chapter 13 a workflow search engine that generates self-contained, informative and concise query results on workflow hierarchies. To do so, we define a workflow search result as a *minimal view* of a workflow that contains query keywords, which can be considered as a projection of three dimensional data onto a two dimensional plane. In this way, the dataflows between keyword matches can be captured.

14.2 Future Work

In this section we discuss possible future works in several categories.

Processing Keyword Search on Structured Data with Improved and/or Controllable Efficiency. As we said before, it is much more costly to process keyword queries on structured data compared with processing keyword queries on text documents of comparable size. In order to achieve an efficiency comparable to that of Web search engines and satisfy the needs of impatient users, it is important to develop techniques for processing keyword searches on structured data with improved and/or guaranteed efficiency. One way to address this problem is to generate the results in the order of their ranking. However, this is possible only for a limited number of result semantics and ranking functions; besides, in many cases generating the first result is already very difficult (which corresponds to the group Steiner tree problem, an NP-hard problem). In Chapter 12, we initiate the study of using materialized views to answer keyword queries on XML and maintaining views, however, it is applicable to a specific search semantics (SLCA) on a specific type of data (XML).

Possible future directions for approaching the efficiency issue of keyword search engines include:
1) Developing query optimization techniques that estimate the cost of processing a keyword query using different strategies, and choose the strategies accordingly. For example, we may build a multi-resolution index of the data and use the right resolution/algorithm based on the estimated query cost. This is very much in line with the cloud computing framework, where the service provider needs to determine the optimal way of processing a user query, considering factors like query characteristics, user priority, etc.
2) Parallel computation. Parallel computation is necessary for processing large data. While parallel computing has been a popular research topic, leveraging such infrastructure for keyword search on structured data poses new challenges that have hardly been addressed. There is research on parallel computation of keyword queries on relational databases [114], while more generic approaches for handling general graphs and being capable of computing top-k results are needed.

Complementing Structured Search Engines with Web Search Engines to Generate Meaningful Results. In our prior works, query results or result analysis (snippet, comparison tables, cluster, etc.) are generated solely based on the query and the data. In fact, text search and structured search can be combined to provide more effective search results. For example, currently Web search engines have much more data and bigger query log than a relational database used by an enterprise or organization, thus the search results of a Web search engine carry more significant statistics. If a user-issued keyword query on a database is ambiguous and have multiple possible semantics, we may issue one or more Web queries, and use the set of results returned by the Web search engine to determine the likelihood of each semantics (e.g., based on the number of results returned). If a keyword in a user query does not appear in the data, we can also leverage Web search engines to find out its synonyms or related terms (e.g., the user may use keyword "DB" while the data contains word "database"). The research challenges herein include how to choose the right set of queries to submit to the Web search engine, and how to use the results returned by Web search engine to serve our purposes.

Self Explaining Structured Data. Unlike text documents, search results on structured data can provide results that integrate and aggregate information from multiple sources. Thus the ability to help users understand such analytic results is important. Techniques are demanded to allow users to ask certain types of questions to a structured data set, which is able to give concise and meaningful answers. Some useful types of questions include:
1) Provenance questions with respect to query result, e.g., why a tuple is/is not returned for a query? Where does a result come from?
2) Provenance questions with respect to ranking, e.g., why is one result ranked higher than another?
3) Relationship/analytical questions with respect to tuples. For example, what are the most significant relationships of these three tuples and what are other tuples that also possess such relationships?

While provenance analysis for structured queries has been studied [30, 35, 66], it is not yet available for keyword search.

To sum up, keyword search on structured data has been recognized as an important problem by database researchers, who have achieved promising results in this area. In order to develop a commercially successful keyword search engine on structured data that benefits a large population of users, there are still many open and challenging questions that await being studied in the future.

REFERENCES

[1] GEON. http://www.geongrid.org.

[2] Kepler. http://kepler-project.org/.

[3] MOML. http://ptolemy.eecs.berkeley.edu/papers/05/ptIIdesign1-intro/ptIIdesign1-intro.pdf.

[4] myExperiment. http://www.myexperiment.org/.

[5] Oracle berkeley db. http://www.oracle.com/technology/products/berkeley-db/index.html.

[6] Seek. http://seek.ecoinformatics.org.

[7] Taverna Project. http://taverna.sourceforge.net/.

[8] Timber project. http://www.eecs.umich.edu/db/timber/.

[9] Triana. http://www.trianacode.org/collaborations/index.html.

[10] Wordnet: a lexical database for the English language. http://wordnet.princeton.edu/.

[11] C. C. Aggarwal, N. Ta, J. Wang, J. Feng, and M. Zaki. Xproj: A Framework for Projected Structural Clustering of XML Documents. In *KDD*, 2007.

[12] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A System for Keyword-Based Search over Relational Databases. In *ICDE*, 2002.

[13] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher, and S. Mock. Kepler: An extensible system for design and execution of scientific workflows. In *SSDBM*, pages 423–424, 2004.

[14] A. Arion, V. Benzaken, I. Manolescu, and Y. Papakonstantinou. Structured Materialized Views for XML Queries. In *VLDB*, pages 87–98, 2007.

[15] K. Arrow. Social Choice and Individual Values. 1951.

[16] A. Balmin, F. Özcan, K. S. Beyer, R. Cochrane, and H. Pirahesh. A Framework for Using Materialized XPath Views in XML Query Processing. In *VLDB*, pages 60–71, 2004.

[17] Z. Bao, S. Cohen-Boulakia, S. Davidson, A. Eyal, and S. Khanna. Differencing provenance in scientific workflows. In *Data Engineering, 2009. ICDE 2009. IEEE 25th International Conference on*, 2009.

[18] Z. Bao, T. W. Ling, B. Chen, and J. Lu. Effective XML Keyword Search with Relevance Oriented Ranking. In *ICDE*, pages 517–528, 2009.

[19] Z. Bar-Yossef and M. Gurevich. Mining Search Engine Query Logs via Suggestion Sampling. *PVLDB*, 1(1):54–65, 2008.

[20] C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying Business Processes. In *VLDB*, pages 343–354, 2006.

[21] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword Searching and Browsing in Databases using BANKS. In *ICDE*, pages 431–440, 2002.

[22] O. Biton, S. C. Boulakia, and S. B. Davidson. Zoom*UserViews: Querying Relevant Provenance in Workflow Systems. In *VLDB*, pages 1366–1369, 2007.

[23] O. Biton, S. C. Boulakia, S. B. Davidson, and C. S. Hara. Querying and managing provenance through user views in scientific workflows. In *Proceedings of ICDE*, pages 1072–1081, 2008.

[24] C. Botev and J. Shanmugasundaram. Context-Sensitive Keyword Search and Ranking for XML. In *WebDB*, pages 115–120, 2005.

[25] A. Z. Broder. A Taxonomy of Web Search. *SIGIR Forum*, 36(2):3–10, 2002.

[26] G. Cao, J.-Y. Nie, J. Gao, and S. Robertson. Selecting Good Expansion Terms for Pseudo-Relevance Feedback. In *SIGIR*, pages 243–250, 2008.

[27] D. Carmel, H. Roitman, and N. Zwerdling. Enhancing Cluster Labeling Using Wikipedia. In *SIGIR*, pages 139–146, 2009.

[28] C. Carpineto, R. de Mori, G. Romano, and B. Bigi. An Information-Theoretic Approach to Automatic Query Expansion. *ACM Trans. Inf. Syst.*, 19(1):1–27, 2001.

[29] K. Chakrabarti, S. Chaudhuri, and S. won Hwang. Automatic Categorization of Query Results. In *SIGMOD Conference*, pages 755–766, 2004.

[30] A. Chapman and H. V. Jagadish. Why Not? In *SIGMOD Conference*, pages 523–534, 2009.

[31] S. Chaudhuri and R. Kaushik. Extending Autocompletion to Tolerate Errors. In *SIGMOD Conference*, pages 707–718, 2009.

[32] A. Chebotko, S. Chang, S. Lu, F. Fotouhi, and P. Yang. Scientific Workflow Provenance Querying with Security Views. In *WAIM*, pages 349–356, 2008.

[33] L. J. Chen and Y. Papakonstantinou. Supporting Top-K Keyword Search in XML Databases. In *ICDE*, pages 689–700, 2010.

[34] Z. Chen and T. Li. Addressing Diverse User Preferences in SQL-Query-Result Navigation. In *SIGMOD Conference*, pages 641–652, 2007.

[35] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[36] T. Cheng and K. C.-C. Chang. Entity Search Engine: Towards Agile Best-Effort Information Integration over the Web. In *CIDR*, pages 108–113, 2007.

[37] T. Cheng, H. W. Lauw, and S. Paparizos. Fuzzy matching of Web queries to structured data. In *ICDE*, pages 713–716, 2010.

[38] T. Cheng, X. Yan, and K. C.-C. Chang. EntityRank: Searching Entities Directly and Holistically. In *VLDB*, pages 387–398, 2007.

[39] P.-A. Chirita, C. S. Firan, and W. Nejdl. Personalized Query Expansion for the Web. In *SIGIR*, pages 7–14, 2007.

[40] E. Chu, A. Baid, X. Chai, A. Doan, and J. F. Naughton. Combining Keyword Search and Forms for Ad Hoc Querying of Databases. In *SIGMOD Conference*, pages 349–360, 2009.

[41] E. Cohen, H. Kaplan, and T. Milo. Labeling Dynamic XML Trees. In *PODS*, 2002.

[42] S. Cohen, S. C. Boulakia, and S. B. Davidson. Towards a model of provenance and user views in scientific workflows. In *DILS*, pages 264–279, 2006.

[43] S. Cohen, J. Mamou, Y. Kanza, and Y. Sagiv. XSEarch: A Semantic Search Engine for XML. In *VLDB*, pages 45–56, 2003.

[44] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (Second Edition)*. The MIT Press, 2001.

[45] T. Dalamagas, T. Cheng, K.-J. Winkel, and T. Sellis. A Methodology for Clustering XML Documents by Structure. *Inf. Syst.*, 31(3):187–228, 2006.

[46] T. Dalamagas, T. Cheng, K.-J. Winkel, and T. K. Sellis. Clustering xml documents by structure. In *SETN*, pages 112–121, 2004.

[47] T. Dalamagas, T. Cheng, K.-J. Winkel, and T. K. Sellis. Clustering XML Documents Using Structural Summaries. In *EDBT Workshops*, 2004.

[48] B. B. Dalvi, M. Kshirsagar, and S. Sudarshan. Keyword Search on External Memory Data Graphs. *Proc. VLDB Endow.*, 1(1):1189–1204, 2008.

[49] A. Deutsch, M. F. Fernández, and D. Suciu. Storing Semistructured Data with STORED. In *SIGMOD Conference*, pages 431–442, 1999.

[50] B. Ding, J. X. Yu, S. Wang, L. Qin, X. Zhang, and X. Lin. Finding Top-k Min-Cost Connected Trees in Databases. In *ICDE*, pages 836–845, 2007.

[51] B. Ding, B. Zhao, C. X. Lin, J. Han, and C. Zhai. TopCells: Keyword-based search of top-k aggregated documents in text cube. In *ICDE*, pages 381–384, 2010.

[52] A. Doucet and H. Ahonen-Myka. Naïve Clustering of a large XML Document Collection. In *INEX Workshop*, 2002.

[53] L. Engebretsen and J. Holmerin. Clique Is Hard to Approximate within $n^{1-o(1)}$. In *ICALP*, pages 2–12, 2000.

[54] J. Feng, N. Ta, Y. Zhang, and G. Li. Exploit Sequencing Views in Semantic Cache to Accelerate XPath Query Evaluation. In *WWW*, 2007.

[55] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Keyword Proximity Search in Complex Data Graphs. In *SIGMOD Conference*, pages 927–940, 2008.

[56] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked Keyword Search over XML Documents. In *SIGMOD Conference*, pages 16–27, 2003.

[57] P. Hansen and F. S. Roberts. An impossibility result in axiomatic location theory. In *Mathematics of Operations Research*, 1996.

[58] H. He, H. Wang, J. Y. 0001, and P. S. Yu. BLINKS: Ranked Keyword Searches on Graphs. In *SIGMOD Conference*, pages 305–316, 2007.

[59] T. Heinis and G. Alonso. Efficient Lineage Tracking for Scientific Workflows. In *SIGMOD Conference*, pages 1007–1018, 2008.

[60] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-Style Keyword Search over Relational Databases. In *VLDB*, pages 850–861, 2003.

[61] V. Hristidis, N. Koudas, Y. Papakonstantinou, and D. Srivastava. Keyword Proximity Search in XML Trees. *IEEE Trans. Knowl. Data Eng.*, 18(4):525–539, 2006.

[62] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword Search in Relational Databases. In *VLDB*, pages 670–681, 2002.

[63] V. Hristidis, Y. Papakonstantinou, and A. Balmin. Keyword Proximity Search on XML Graphs. In *ICDE*, pages 367–378, 2003.

[64] Y. Huang, Z. Liu, and Y. Chen. eXtract: A Snippet Generation System for XML Search. *PVLDB*, 1(2):1392–1395, 2008.

[65] Y. Huang, Z. Liu, and Y. Chen. Query Biased Snippet Generation in XML Search. In *SIGMOD Conference*, pages 315–326, 2008.

[66] R. Ikeda and J. Widom. Panda: A System for Provenance and Data. *IEEE Data Eng. Bull.*, 33(3):42–49, 2010.

[67] M. Jayapandian and H. V. Jagadish. Automated Creation of a Forms-based Database Query Interface. *PVLDB*, 1(1):695–709, 2008.

[68] M. Jayapandian and H. V. Jagadish. Automating the Design and Construction of Query Forms. *IEEE Trans. Knowl. Data Eng.*, 21(10):1389–1402, 2009.

[69] A. K. Joshi and Y. Schabes. Tree-Adjoining Grammars and Lexicalized Grammars. In *Tree Automata and Languages*, pages 409–432. 1992.

[70] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional Expansion For Keyword Search on Graph Databases. In *VLDB*, pages 505–516, 2005.

[71] A. Kashyap, V. Hristidis, and M. Petropoulos. FACeTOR: Cost-Driven Exploration of Faceted Query Results. In *CIKM*, pages 719–728, 2010.

[72] B. Kimelfeld and Y. Sagiv. Finding and Approximating Top-k Answers in Keyword Proximity Search. In *PODS*, pages 173–182, 2006.

[73] J. M. Kleinberg. An Impossibility Theorem for Clustering. In *NIPS*, pages 446–453, 2002.

[74] L. Kong, R. Gilleron, and A. Lemay. Retrieving Meaningful Relaxed Tightest Fragments for XML Keyword Search. In *EDBT*, pages 815–826, 2009.

[75] G. Koutrika, A. Simitsis, and Y. E. Ioannidis. Précis: The Essence of a Query Answer. In *ICDE*, pages 69–78, 2006.

[76] G. Koutrika, Z. M. Zadeh, and H. Garcia-Molina. Data Clouds: Summarizing Keyword Search Results over Structured Data. In *EDBT*, pages 391–402, 2009.

[77] S. Kullback. The Kullback-Leibler Distance. In *The American Statistician*, 1987.

[78] M. L. Lee, L. H. Yang, W. Hsu, and X. Yang. XClust: Clustering XML Schemas for Effective Integration. In *CIKM*, 2002.

[79] R. Lempel and S. Moran. Predictive Caching and Prefetching of Query Results in Search Engines. In *WWW*, 2003.

[80] C. Li, T. W. Ling, and M. Hu. Efficient Processing of Updates in Dynamic XML Data. In *ICDE*, 2006.

[81] C. Li, N. Yan, S. B. Roy, L. Lisham, and G. Das. Facetedpedia: Dynamic generation of query-dependent faceted interfaces for wikipedia. In *WWW*, pages 651–660, 2010.

[82] G. Li, J. Feng, J. Wang, and L. Zhou. Effective Keyword Search for Valuable LCAs over XML Documents. In *CIKM*, pages 31–40, 2007.

[83] G. Li, S. Ji, C. Li, and J. Feng. Efficient Type-ahead Search on Relational Data: A TASTIER Approach. In *SIGMOD Conference*, pages 695–706, 2009.

[84] G. Li, B. C. Ooi, J. Feng, J. Wang, and L. Zhou. Ease: an effective 3-in-1 keyword search method for unstructured, semi-structured and structured data. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 903–914, New York, NY, USA, 2008. ACM.

[85] J. Li, C. Liu, R. Zhou, and W. Wang. Suggestion of Promising Result Types for XML Keyword Search. In *EDBT*, pages 561–572, 2010.

[86] Y. Li, C. Yu, and H. V. Jagadish. Schema-Free XQuery. In *VLDB*, pages 72–83, 2004.

[87] W. Lian, D. W. lok Cheung, N. Mamoulis, and S.-M. Yiu. An Efficient and Scalable Algorithm for Clustering XML Documents by Structure. *IEEE Trans. on Knowl. and Data Eng.*, 16(1):82–96, 2004.

[88] Y.-H. Liang, T.-J. Zhao, H. Yu, and J.-M. Yao. High Precision English base Noun Phrase Identification Based on “Waterfall” Model. In *Machine Learning and Cybernetics*, pages 4902–4907, 2005.

[89] F. Liu, C. T. Yu, W. Meng, and A. Chowdhury. Effective Keyword Search in Relational Databases. In *SIGMOD Conference*, pages 563–574, 2006.

[90] Z. Liu, Y. Cai, and Y. Chen. TargetSearch: A Ranking Friendly XML Keyword Search Engine. In *ICDE*, pages 1101–1104, 2010.

[91] Z. Liu and Y. Chen. Identifying Meaningful Return Information for XML Keyword Search. In *SIGMOD Conference*, pages 329–340, 2007.

[92] Z. Liu and Y. Chen. Answering Keyword Queries on XML Using Materialized Views. In *ICDE*, pages 1501–1503, 2008.

[93] Z. Liu and Y. Chen. Reasoning and Identifying Relevant Matches for XML Keyword Search. *PVLDB*, 1(1):921–932, 2008.

[94] Z. Liu and Y. Chen. Return Specification Inference and Result Clustering for Keyword Search on XML. *ACM Trans. Database Syst.*, 35(2), 2010.

[95] Z. Liu, Y. Huang, and Y. Chen. Improving XML Search by Generating and Utilizing Informative Result Snippets. *ACM Trans. Database Syst.*, 35(3), 2010.

[96] Z. Liu, S. Natarajan, and Y. Chen. Query Expansion Based on Clustered Results. *PVLDB*, 4(6), 2011.

[97] Z. Liu, S. Natarajan, P. Sun, S. Booher, T. Meehan, R. Winkler, and Y. Chen. XSACT: A Comparison Tool for Structured Search Results. *PVLDB*, 3(2):1581–1584, 2010.

[98] Z. Liu, Q. Shao, and Y. Chen. Searching Workflows with Hierarchical Views. *PVLDB*, 3(1):918–927, 2010.

[99] Z. Liu, P. Sun, and Y. Chen. Structured Search Result Differentiation. *PVLDB*, 2(1):313–324, 2009.

[100] Z. Liu, J. Walker, and Y. Chen. XSeek: A Semantic XML Search Engine Using Keywords. In *VLDB*, pages 1330–1333, 2007.

[101] Y. Lu, W. Wang, J. Li, and C. Liu. XClean: Providing Valid Spelling Suggestions for XML Keyword Queries. In *ICDE*, pages 661–672, 2011.

[102] Y. Luo, X. Lin, W. Wang, and X. Zhou. SPARK: Top-k Keyword Query in Relational Databases. In *SIGMOD Conference*, pages 115–126, 2007.

[103] B. Mandhani and D. Suciu. Query Caching and View Selection for XML Databases. In *VLDB*, pages 469–480, 2005.

[104] A. Markowetz, Y. Yang, and D. Papadias. Keyword search on relational data streams. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 605–616, New York, NY, USA, 2007. ACM.

[105] M. Muhr, R. Kern, and M. Granitzer. Analysis of Structural Relationships for Hierarchical Cluster Labeling. In *SIGIR*, pages 178–185, 2010.

[106] U. Nambiar and S. Kambhampati. Answering Imprecise Queries over Autonomous Web Databases. In *ICDE*, page 45, 2006.

[107] A. Nierman and H. V. Jagadish. Evaluating Structural Similarity in XML Documents. In *WebDB*, 2002.

[108] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. In *Bioinformatics*, pages 3045–3054, 2003.

[109] P. OąŕNeil, E. OąŕNeil, S. Pal, I. Cseri, and G. Schaller. ORDPATHs: Insert-Friendly XML Node Labels. In *SIGMOD*, 2004.

[110] N. Onose, A. Deutsch, Y. Papakonstantinou, and E. Curtmola. Rewriting Nested XML Queries Using Nested Views. In *SIGMOD Conference*, pages 443–454, 2006.

[111] M. J. Osborne and A. Rubinstein. A Course in Game Theory. In *MIT Press*, 1994.

[112] D. M. Pennock, E. Horvitz, and C. L. Giles. An Impossibility Theorem for Clustering. In *AAAI*, 2000.

[113] K. Q. Pu and X. Yu. Keyword Query Cleaning. In *VLDB*, 2008.

[114] L. Qin, J. Yu, and L. Chang. Ten Thousand SQLs: Parallel Keyword Queries Computing. *PVLDB*, 3(1):58–69, 2010.

[115] L. Qin, J. X. Yu, L. Chang, and Y. Tao. Querying Communities in Relational Databases. In *ICDE*, pages 724–735, 2009.

[116] M. Ramanath and K. S. Kumar. A Rank-Rewrite Framework for Summarizing XML Documents. In *ICDE Workshops*, pages 540–547, 2008.

[117] Y. Rubner, C. Tomasi, and L. J. Guibas. A Metric for Distributions with Applications to Image Databases. In *ICCV*, pages 59–66, 1998.

[118] P. C. Saraiva, E. S. de Moura, N. Ziviani, W. Meira, R. Fonseca, and B. RibeiroNeto. Rank-Preserving Two-Level Caching for Scalable Search Engines. In *SIGIR*, 2007.

[119] N. Sarkas, N. Bansal, G. Das, and N. Koudas. Measure-driven keyword-query expansion. *PVLDB*, 2(1):121–132, 2009.

[120] A. Sawires, J. Tatemura, O. Po, D. Agrawal, A. E. Abbadi, and K. S. Candan. Maintaining XPath Views In Loosely Coupled Systems. In *VLDB*, pages 583–594, 2006.

[121] A. Sawires, J. Tatemura, O. Po, D. Agrawal, and K. S. Candan. Incremental Maintenance of Path Expression Views. In *SIGMOD Conference*, pages 443–454, 2005.

[122] M. Sayyadian, H. LeKhac, A. Doan, and L. Gravano. Efficient Keyword Search Across Heterogeneous Relational Databases. In *ICDE*, pages 346–355, 2007.

[123] C. E. Scheidegger, H. T. Vo, D. Koop, J. Freire, and C. T. Silva. Querying and Re-Using Workflows with VisTrails. In *SIGMOD*, 2008.

[124] F. Shao, L. Guo, C. Botev, A. Bhaskar, M. Chettiar, F. Y. 0002, and J. Shanmugasundaram. Efficient Keyword Search over Virtual XML Views. In *VLDB*, pages 1057–1068, 2007.

[125] Q. Shao, P. Sun, and Y. Chen. WISE: A Workflow Information Search Engine. In *ICDE*, pages 1491–1494, 2009.

[126] C. Sun, C. Y. Chan, and A. K. Goenka. Multiway SLCA-based Keyword Search in XML Data. In *WWW*, pages 1043–1052, 2007.

[127] P. Sun, Z. Liu, S. B. Davidson, and Y. Chen. Detecting and Resolving Unsound Workflow Views for Correct Provenance Analysis. In *SIGMOD Conference*, pages 549–562, 2009.

[128] A. Tagarelli and S. Greco. Toward Semantic XML Clustering. In *SDM*, 2006.

[129] N. Tang, J. X. Yu, M. T. Özsu, B. Choi, and K.-F. Wong. Multiple Materialized View Selection for XPath Query Rewriting. In *ICDE*, pages 873–882, 2008.

[130] Y. Tao and J. X. Yu. Finding Frequent Co-occurring Terms in Relational Keyword Search. In *EDBT*, pages 839–850, 2009.

[131] A. Tombros, R. Villa, and C. J. V. Rijsberge. The Effectiveness of Query-specific Hierarchic Clustering in Information Retrieval. *Information Processing and Management*, 38(4):559–582, 2002.

[132] O. Vechtomova, S. E. Robertson, and S. Jones. Query Expansion with Long-Span Collocates. *Inf. Retr.*, 6(2):251–273, 2003.

[133] V. Vesper. Let's Do Dewey. http://www.mtsu.edu/ vvesper/dewey.html.

[134] M. Vrhovnik, H. Schwarz, S. Radeschütz, and B. Mitschang. An Overview of SQL Support in Workflow Products. In *ICDE*, pages 1287–1296, 2008.

[135] N. Wacholder, D. K. Evans, and J. Klavans. Automatic Identification and Organization of Index Terms for Interactive Browsing. In *JCDL*, pages 126–134, 2001.

[136] J. T. L. Wang, J. Liu, and J. Wang. XML Clustering And Retrieval Through Principal Component Analysis. *International Journal on Artificial Intelligence Tools*, 14(4):683, 2005.

[137] T. Wang, D. xin Liu, and X.-Z. Lin. XML Document Clustering by Independent Component Analysis. In *KDXD*, 2006.

[138] D. Xin, Y. He, and V. Ganti. Keyword++: A Framework to Improve Keyword Search Over Entity Databases. *PVLDB*, 3(1):711–722, 2010.

[139] G. Xing, J. Guo, and Z. Xia. Classifying XML Documents Based on Structure/Content Similarity. In *INEX*, 2006.

[140] G. Xing, Z. Xia, and J. Guo. Clustering XML Documents Based on Structural Similarity. In *DASFAA*, 2007.

[141] Extensible markup language (xml) 1.0, 2004. http://www.w3.org/TR/REC-xml/.

[142] J. Xu and W. B. Croft. Query Expansion Using Local and Global Document Analysis. In *SIGIR*, pages 4–11, 1996.

[143] W. Xu and Z. M. Özsoyoglu. Rewriting XPath Queries Using Materialized Views. In *VLDB*, pages 121–132, 2005.

[144] Y. Xu and Y. Papakonstantinou. Efficient Keyword Search for Smallest LCAs in XML Databases. In *SIGMOD Conference*, pages 537–538, 2005.

[145] Y. Xu and Y. Papakonstantinou. Efficient LCA Based Keyword Search in XML Data. In *EDBT*, pages 535–546, 2008.

[146] C. Yu and H. V. Jagadish. Schema Summarization. In *VLDB*, pages 319–330, 2006.

[147] B. Zhou and J. Pei. Answering Aggregate Keyword Queries on Relational Databases Using Minimal Group-bys. In *EDBT*, pages 108–119, 2009.