EFFICIENT SEARCH OF GENERAL AND-OR KEYWORD QUERIES IN XML DATA

Wang Xianjun

NATIONAL UNIVERSITY OF SINGAPORE

2007

EFFICIENT SEARCH OF GENERAL AND-OR KEYWORD QUERIES IN XML DATA

Wang Xianjun

(B. Sci. Fudan University, P. R. China)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2007

Acknowledgement

I would like to express my gratitude to all those who have shared the graduate life with me and helped me in all kinds of ways. Without their encouragement and support I would not be able to write this section.

Firstly, I would like to thank my supervisor, Professor Chan Chee Yong for his guidance. He helped me to build a comprehensive understanding of my research topics, and provided me with a source of stimulating suggestions. His extraordinary patience and all kinds of supports are important for me.

I would like to particularly thank Sun Chong, Ni Yuan and Goenka Amit Kumar for our discussions on my research work which helped me to acquire a deeper and broader view.

My other collagues of the database group of the computer science department, Chen Su, Chen Ding, Cheng Weiwei, Cao Yu, Li Yingguang, Xu Linhao, Yang Xiaoyan, Zhang Zhenjie, Xiang Shili and Ni Wei, have been of great help.

I also feel the need to thank Chen Su, Zhuo Shaojie and Guo Dong for their encouragement and support in life for years especially during the period of thesis writing. They are such good and dedicated friends.

Finally, I would like to thank my parents, who are always trusting in me and back up all of my decisions. They taught me to be thankful to life and made me understand that the process is much more important than the end-result.

Contents

Summary

This thesis examines general form keyword search queries in XML data. The keyword search for XML documents are important as XML has become the standard for representing web data. Existing approaches have focused on integrating keyword search with XML query language which require knowledge of query or algebra syntax. Recent work got rid of this limitation and developed web-like keyword search approaches. They attempted to address the conjunctive keyword searching problem based on the notion of smallest lowest common ancestor (SLCA) semantics. However, they rarely consider keyword search with operators other than AND.

In this thesis, we have presented a novel approach to process general form AND-OR keyword search queries. To the best of our knowledge, this is the first work to handle keyword queries with any combination of AND and OR operators.

We utilize the tree structure to represent the keyword search query. The query can be easily parsed into a query tree, with keywords in leaf nodes and operators in root as well as intermediate nodes, and operands attached as children of the operator nodes. Using the query tree, not only the query is naturally divided into several subqueries in the form of subtrees in the query tree, but also the

processing can be broken up and specialized according to the type of the query nodes. Consequently, no matter how many types of general form queries there are, the processing methods we need to consider are now limited to three: how to process the keyword node in the query tree, and how about the AND operator nodes and the OR nodes.

We adopted the AND processing from SLCA computing algorithms and proposed a comparison mechanism for OR processing which prunes intermediate results that cover other intermediate results. By delivering to the parent node the intermediate results immediately when a new one is produced, a pipeline is built in the query tree. We do not need to wait for all the matches of the child nodes coming out. The first searching result can be quickly output while the search is still running for following results. Quick response is critical to keyword search end users. An important benefit due to the tree structure and the pipelined approach is that the effect of increase in number of keywords is reduced by logarithm.

The efficiency of our approach is verified via comprehensive experiments. Although the evaluation time is increasing with an increase in keyword frequency, our approach has exhibited satisfying processing response and outperforms previous approaches in most cases especially when the query is a complex one. We also find by experimental studies that our approach responds similarly to equivalent queries with different depths and structures. That avoids query rewriting due to the complexity and is surely to benefit both end users and search engine designers.

List of Figures

Chapter 1

Introduction

Keyword search is a proven user-friendly way of querying in document systems and World Wide Web.

For traditional query on relational databases, the processing approach is constrained by the structured query imposed by the SQL language. Users are supposed to have a knowledge of the structure of the data or document that is to be queried. They can only write a query by describing the data structure as well as their constraints. In addition to the structure constraint, the complexity of query language is another cause that these methods are not so friendly and keyword search is proposed as an alternative means.

As XML becomes the standard for representing web data, effective and efficient methods to query XML data have become an increasingly important problem.

An XML query typically involves one or more sets of structurally related XML elements that are the processing context used by the query. The structure informa-

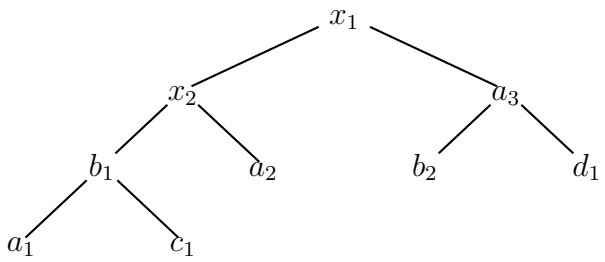


Figure 1.1: Example XML Trees T_1

tion is used either to evaluate conditions or to return results. If a user knows the document structure, he can write a meaningful query in XQuery [5] (or XPath [4]) specifying exactly how the nodes involved in the query are structurally connected to each other. If the user does not have any knowledge of the structural relationships, a keyword search query will be more helpful as long as the user can tell the element tag names.

However, unlike a structured query where the connection among the data nodes matching the query is specified precisely in the "where" clause (in XQuery or SQL) or as variable bindings (in XQuery), we need to automatically connect the match nodes in a meaningful way. Recent work attempted to address the above problem based on the notion of *smallest lowest common ancestor (SLCA)* semantics.

The following example illustrates the concept of SLCA-based keyword search.

Example 1.1

Consider the XML tree T_1 shown in Figure 1.1, where the keyword nodes are annotated with subscripts for ease of reference. Consider a keyword search using the keywords $\{a, b\}$ on T_1. The lowest common ancestor(LCA) found will be $\{x_2, b_1, a_3\}$ as x_2 is the LCA of $\{a_2, b_1\}$, b_1 is the LCA of $\{a_1, b_1\}$, a_3 is the LCA of $\{a_3, b_2\}$.

But x_2 is not a SLCA because it has a descendant node b_1 that is a SLCA. As a result, the SLCA-based keyword search will return a set of $\{a_2, b_1\}$. □

Not only the SLCA notion provides a meaningful connection, but also indicates the granularity as well as the content of the returned information. However, all those work focus on keyword conjunction but rarely consider keyword search with operators other than AND. Therefore, in this thesis we introduce a novel approach for processing general form keyword search queries that are any combination of AND and OR operators.

1.1 Contributions

In this thesis, we are first to present an efficient approach for general form AND-OR keyword search queries. Our contributions are summarized as follows:

- We propose a tree structure to represent the general form queries, no matter how complex the query is. Utilizing the tree structure, we gain opportunities for optimizing.
- We design a pipelined processing approach. The AND processing part is adopted from SLCA algorithms. The OR processing part is designed based on a comparing mechanism.
- Effectiveness and efficiency of our approach as well as some good properties for keyword search are verified by extensive experimental study.

1.2 Organization

This thesis is organized as follows. We introduce the related work in Chapter 2. In Chapter 3 we present some basic definitions and notations as well as data models. Our novel approach for general form keyword query processing is presented in Chapter 4 and Chapter 5, introducing query transformation and processing respectively. We exhibit our experimental study in Chapter 6 and conclude in Chapter 7.

Chapter 2

Related Work

Extensive research has been done on keyword search. Besides those in the areas of information retrieval and full-text search, [10, 7, 8] are systems supporting keyword search over relational databases. [9] is the extension work on top of relational databases supporting keyword search in XML documents.

Keyword search over XML databases has also attracted interest. Several approaches attempt to support information retrieval style search by expanding XQuery or other structured query languages [13, 14, 17, 12, 9, 16]. Among these, [13, 12] consider ranking schemes as well, which is one of the typical IR issues. Proximity search is studied in [17, 13].

The idea of computing the most specific elements for conjunctive queries has been actively explored using LCA (Lowest Common Ancestor), which is the closest research area relevant to this work. As extensions of LCA, MLCA, SLCA and GDMCT have been proposed in [18], [20] and [19] respectively.

2.1 Keyword Search over Relational Databases

In the studies of BANKS [10], DBXplorer [7], and DISCOVER [8], a database is viewed as a graph with tuples (or objects) as nodes and relationships as edges. It is required that all query keywords appear in the tree of nodes or tuples that are returned as the answer to a query.

BANKS answers keyword queries by searching for steiner trees [11] containing all keywords, using heuristics during the search. The identification of connected trees is an NP-hard problem. As a result, the implementation of BANKS is tuned for a graph that fits in main memory. Since it requires that all the data edges fit in memory, it is not feasible for large data sets.

The structural constraints expressed in RDBMS schema is exploited in DBXplorer and DISCOVER to facilitate query processing. They share similar architectures and first get the tuples containing keywords from the master index. After that, a set of SQL queries corresponding to all different ways to connect the keywords based on the schema graph are generated. The selection of the optimal execution plan is proven to be NP-complete. Trees of tuples containing all the keywords are connected through primary-foreign key relationships and are output as query results.

Since RDBMS schema is needed in processing, the approaches can not be applied if the XML documents can not be mapped to a rigid relational schema. Besides, they encounter similar problem as BANKS that they may need to read a

huge number of connecting tuples from the disk since it is impractical to store all the connections between all pairs of nodes in the inverted index.

XKeyword [9] extends the work of DISCOVER by materializing path indices. It reduces the number of joins in the generated SQL queries and provides fast response times.

2.2 Integrating Keyword Search with XML Query Language

Recently, there has been interests in integrating keyword search with structured XML querying, among which [17] and [13] are two relatively early works. In [17] XML-QL is extended with keyword search on subtrees of certain tags. It helps novice users formulate queries even when they have no idea of the document structure. Besides, inverted file indices for XML documents are established in a relational database system. So full-text search as well as distributed query processing are supported in a relational environment in [17].

XIRQL [13] is an extension of XQL for information retrieval. Several IR-related features are supported in this system like weighting and ranking, relevance-oriented search, data types with vague predicates, and semantic relativism.

XXL search engine is presented in [14], which has an SQL-like syntax. Both exact-match and semantic-similarity search conditions can be expressed in XXL because it exploits the structural information as well as the rich semantic annota-

tions. IR-style relevance ranking is supported in XXL. Ontological information and suitable index structures are used to improve the search efficiency and effectiveness.

Xyleme [22] creates its own query language for XML query processing. It is an extension of OQL [23] and provides a mix of database and information retrieval characteristics.

Various XML full-text query languages have also been proposed. A recent work [27] presents XFT algebra that accounts for element nesting in XML document structure to evaluate queries with complex full-text predicates.

Although the above languages support flexible querying of XML, they still require knowledge of query or algebra syntax and are not suitable for naive users.

XRANK system [12] extends web-like keyword search to XML and requires no knowledge of query syntax any more. The focus is its ranking mechanism. Given a tree T containing all the keywords, XRANK assigns a score to T using an adaption of PageRank algorithm of Google [26]. The score is obtained by combining the ranking of all the ranked elements with keyword proximity considering document order. The keyword search algorithm in XRANK utilizes inverted lists and returns subtrees as answers. However, XRANK does not return connected trees to explain how the keywords are connected to each other. Only the most specific result is output although maybe it has parts that are semantically unrelated.

XSearch [15] is closely related to XRANK but employs more information-retrieval techniques. Proximity is included in the ranking formula in terms of the size of the relationship tree and it won't be affected by the order of children, which is

different from XRANK. The main focus of XSearch is in laying the foundations for a semantic search engine over XML documents. It attempts to return meaningful results based on query as well as document structure. Two nodes are considered to be semantically related if and only if there are no two distinct nodes with the same tag name on the path between these two nodes (excluding themselves). A heuristic called *interconnection* relationship is used to determine whether two nodes are meaningfully related. However, interconnection does not work when two unrelated nodes are under same entities. During execution, it uses an all-pairs interconnection index to check the connectivity between nodes, which is not efficient for large XML documents and thus is impracticable in practice.

2.3 Lowest Common Ancestor Computation

The algorithms for computing the LCA of nodes in a tree are well known already [24, 25]. From the study in [16] on, LCA computation applied to XML keyword search queries has been extensively studied.

MEET [16] also creates a query language to enable keyword search in XML documents. The *meet* operator is introduced to help users query XML databases with whose content they are familiar with, but without requiring knowledge of tags and hierarchies. The semantics of the meet operator is the nearest concept (i.e. lowest ancestor) of objects. It operates on multiple sets where all nodes in the same set are required to have the same schema. The meet operator of two nodes v_1 and

v_2 is implemented efficiently using joins on relations, where the number of joins is the number of edges of the shorter one of the paths from v_1 and v_2 to their LCA.

In contrast to [16], some other works do not require schema information, thus present a more user-friendly interface.

The concept of Smallest LCAs (SLCAs) was first proposed in [20]. SLCAs are defined to be the LCAs that do not contain other LCAs. According to the SLCA semantics, the result of a keyword query is the set of nodes that (i) contain the keywords either in their tags or in the tags of their descendant nodes and (ii) they have no descendant node that also contains all the keywords either in its own tag or in the tags of its descendant nodes. Meaningful LCAs (MLCAs) is a similar concept with SLCAs. Two nodes matching to different keywords are considered to be meaningfully related if their LCA is an SLCA; a set of nodes consisting of one match to each keyword is meaningfully related if every pair is meaningfully related, and a MLCA is defined as the LCA of these nodes.

Y. Li et al [18] incorporates MLCA search in XQuery and proposes a simple, novel XML document search technique, namely Schema-Free Query. By marking structurally ambiguous elements with *mlcas* keyword and ambiguous tag names with *expand* function, it enables users to query an XML document without full knowledge of the document schema. At the same time, any partial knowledge available to the user can be exploited to advantage. The predicates in an XQuery are specified through MLCA. A stack-based algorithm is deviced for the MLCA computation using structural joins.

Although both of the concept of MLCAs and that of interconnection in XSearch are designed to capture the meaningful fragments of the XML document based on tag names as well as keywords provided in a query, they are quite different when XML data has more than one logical hierarchy, for example, when a entity have different tag names. We have mentioned above that XSearch fail to recognize meaningful structure when entities have different tag names. In contrast, search based on MLCAs can recognize this fact and avoid returning incorrect result.

XKSearch also makes an effort to improve the efficiency and effectiveness of keyword search against LCAs. For each keyword the system maintains a sorted list of nodes that contain the keyword. The key property of SLCA search is that, given two keywords k_1 and k_2 and a node v that contains keyword k_1, one only needs to find the left and right matches of v in the list of k_2 in order to discover potential solutions. If the number of keywords is more than two, the SLCA computation is generalized based on the property: $slca(S_1, \ldots, S_k) = slca(slca(S_1, \ldots, S_{k-1}, S_k)$ where S_1 to S_k are keyword lists and $k > 2$. The Indexed Lookup Eager algorithm is thus derived and completes the computation accessing the k keyword lists in just one round. Delivery of SLCAs is pipelined while intermediate LCAs are removed if they are not SLCAs. The Scan Eager algorithm is exactly the same as the Indexed Lookup Eager algorithm except that it maintains a cursor for each keyword list. Experiments show that the Indexed Lookup Eager algorithm outperforms stack-based algorithms [12, 18] by orders of magnitude when the keywords have different frequencies. Meanwhile, the Scan Eager algorithm has been proven to be the best

variant for the case where the keywords have similar frequencies.

It can be observed that the SLCA computation in XKSearch goes a binary way in that for a query with k keywords, the computation is transformed into a sequence of $k-1$ intermediate SLCA computations, each taking a pair of keyword lists as inputs and outputs another list. An important observation is that the result size is bounded by $min|S_1|, \ldots, |S_k|$. However, XKSearch incurs many unnecessary SLCA intermediate computations even when the result size is small. C. Sun et al. [21] optimizes the SLCA computation by exploiting this observation. Their multiway-SLCA approach takes one data node from each keyword list in a single step. An "anchor" node is chosen to drive the multiway SLCA computation and the match anchored by this node is computed. The selections of the anchor node as well as the next match are optimized based on the properties of the anchor node and the algorithm thus can minimize redundant computations.

Recently, V. Hristidis et al. proposes the concept of Grouped Distance Minimum Connecting Trees (GDMCTs), which is another variant of LCAs in [19]. It provides an optimized version of the LCA-finding stack algorithm. When the result consists of more than one path return subtrees, the stack-based algorithm first reduced each path to an edge labeled with the path length, and then groups the isomorphic reduced subtrees into a generalized tree. Thus the set of LCAs are returned along with efficiently summarized explanations on why each node is an LCA, which is the most important contribution of the work.

All the above research works utilizing LCA computation aim to and can only

be applied to process conjunctive queries, i.e. AND queries. They provide no efficient solution for queries that contain an OR operation as LCA computation is naturally incapable of dealing with disjunction of nodes. Observe this, C. Sun et al. in [21] attempt to extend their approach to process more general keyword search queries supporting combination of AND and OR boolean operators. However, they only produce efficient algorithm that restricts the input keyword search query to be expressed in conjunctive normal form (CNF). If the query is expressed in disjunctive normal form (DNF) or any other forms, it has to be either transformed into CNF first or be processed in a naive way.

This is the original motivation of our work that we intend to develop an efficient approach of processing AND-OR keyword search queries in general form, i.e. any combination of AND and OR operators without any additional conditions. Besides, we provide a web-like style of keyword search that users are not required to have any knowledge of the data being queried. They do not have to know any query language either. We adopt the SLCA computation for conjunctive processing and devise a comparison mechanism uniquely for disjunctive processing. Combining these two and employing the hiding tree structure of the general form query, we develop a pipelined multiway approach for general AND-OR keyword search.

Chapter 3

Preliminaries

Our approach for general keyword search is to be applied to an XML document, which is conventionally represented by a tree structure. Part or whole of the document will be returned as the search result. Before we introduce the details of our approach, some preliminary information will be clarified regarding the data model of the document being queried as well as the search result. We also introduce a notion of *anchor nodes* in the core of SLCA computation approach.

3.1 Data Model

The eXtensible Markup Language (XML) is a hierarchical format. An XML document consists of nested XML elements starting with the root element. Each element can have attributes and values, in addition to nested subelements. XML also supports intra-document references represented using IDREFs, and inter-document

references represented using XLink. An XML document can optionally have a schema. Besides XML Schema, Document Type Description (DTD) is a commonly used method to describe the structure of an XML document and acts like a schema. Since in our approach no schema information is needed, we will not discuss the schema related issues. Figure 3.1 shows an example XML document representing the proceedings of a conference. The ⟨*conf*⟩ element is the root element.

```
⟨conf⟩
  ⟨name⟩VLDB⟨/name⟩
  ⟨year⟩2006⟨/year⟩
  ⟨paper⟩
    ⟨title⟩Efficient Discovery of XML Data Redundancies⟨/title⟩
    ⟨authors⟩
      ⟨author⟩Cong Yu⟨/author⟩
      ⟨author⟩H.V. Jag⟨/author⟩
    ⟨/authors⟩
  ⟨/paper⟩
  ⟨paper⟩
    ⟨title⟩Answering Tree Pattern Queries Using Views⟨/title⟩
    ⟨authors⟩
      ⟨author⟩Laks V.S. Lakshmanan⟨/author⟩
      ⟨author⟩Hui(Wendy) Wang⟨/author⟩
      ⟨author⟩Zheng(Jessica) Zhao⟨/author⟩
    ⟨/authors⟩
  ⟨/paper⟩
  ...
  ⟨paper⟩
    ...
  ⟨/paper⟩
⟨/conf⟩
```

Figure 3.1: Example XML Document

We use tree structure to model XML documents. An XML document is a rooted, ordered, labeled tree. Each node corresponds to an element or a value,

the root node of the tree corresponding to the root element. The edges connecting nodes represent element-subelement or element-value relationships. Node labels are either tags or values of the nodes. The ordering of sibling nodes implicitly defines a total order on the nodes in a tree, obtained by a preorder traversal of the tree nodes.

There are several labeling schemes for assigning a numerical id to each node in XML tree structure. Here we use Dewey numbers [1] as our choice based on the work in [6]. With Dewey labeling, each node is assigned a vector that represents the path from the document's root to the node. Each component of the path represents the absolute order of an ancestor node and each path uniquely identifies the absolute position of the node within the document.

The example XML document in Figure 3.1 with Dewey labeling is shown in Figure 3.2. Using Dewey labeling, it is convenient to represent orders and relationships between nodes in XML tree structure. The LCA of nodes can be easily derived by common prefix computing as well.

We use $<$ to represent the preceding relationship of two Dewey numbers. For example, $0.2.1.0 < 0.2.1.1$. The node with Dewey number 0.2.1.0 precedes the node with Dewey number 0.2.1.1 in preorder traverse. We use $\prec$ to represent the prefix relationship. For example, $0.2.1 \prec 0.2.1.1$. Then the node with Dewey number 0.2.1 is on the path from the root node to the node with Dewey number, i.e. the ancestor of the latter one. The former node is also the parent of the latter one because the difference of the path length from root is only 1. Then it can be easily

derived that 0.2.1.0 and 0.2.1.1 are the Dewey numbers of two sibling nodes as they have the same parent.

The above rules are displayed as follows. For two XML tree nodes n_1, n_2, and their Dewey numbers d_1, d_2,

- `Document order`:

  if $d_1 < d_2$, then n_1 comes before n_2 in document sequence.

- `Siblings relationship`:

  n_1 and n_2 are siblings if and only if d_1 and d_2 only defer in the last component.

- `Ancestor-Descendant relationship`:

  n_1 is the ancestor of n_2 if and only if $d_1 \prec d_2$.

- `Parent-Child relationship`:

  n_2 is the child of n_1 if and only if $d_1 \prec d_2$ and length of d_1 equals that of d_2 minus 1.

- `LCA`:

  the LCA of n_1 and n_2 is the node with Dewey number which is the longest common prefix of d_1 and d_2.

Example 3.1

In Figure 3.2, node *name* has Dewey number 0.0, and node *year* has Dewey number 0.1. Since $0.0 < 0.1$, node *name* precedes node *year*. They are siblings at the same time. The Dewey number 0.1.0 has a prefix 0.1, which is the Dewey number of node *year*. According to the rules listed above, node 2006 is a descendant (as well as a child in this case) of node *year*. □

Sometimes during the processing of keyword search a part of the XML document is used to represent intermediate or final result. This part is denoted *document*

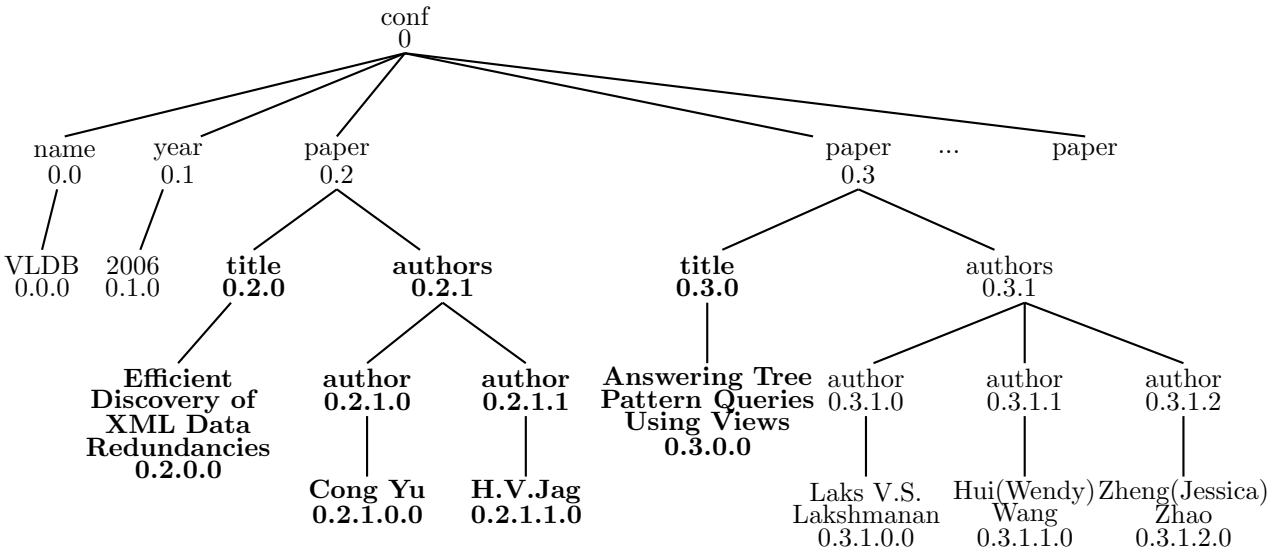


Figure 3.2: Example XML Document With Dewey Labeling

fragment. The document fragment is a consecutive part of an XML document that contains some or all of the elements in the original document. The document fragment is not necessarily well formed. There can be several separate trees without a common root node. However, all the parent-child, ancestor-descendant and the sibling relationships between two nodes in the document fragment are completely preserved as they are in the original document.

We use a tuple `(begin, end)` to denote the document fragment. The label `begin` denotes the beginning node of the fragment, and `end` is the last node of the fragment. Since there may be several nodes sharing the same tag, we will use the Dewey numbers instead of the node tags in practice.

Example 3.2

In Figure 3.1, the fragment in the inner box is a valid document fragment, which

is not well-formed. It begins at the element *title* and ends at the value of the next *title* element and can be expressed in a tuple (0.2.0, 0.3.0.0). Its counterpart in Figure 3.2 are the three subtrees rooted at node *title*(0.2.0), *authors*(0.2.1) and *title*(0.3.0) respectively in the bold font. □

3.2 Search Result

When the keyword search query is applied to the XML document, a set of smallest document fragments containing all the keywords may be returned as result. By smallest we mean that the document fragment does not contain a smaller document fragment that also contains all the keywords. For each document fragment, the lowest common ancestor node of the subtrees corresponding to it is called the LCA of the document fragment, which can be easily inferred from the tuple.

Definition 3.2.1 *For a document fragment D with tuple $(begin, end)$, its LCA is the lowest common ancestor of its beginning and ending node, i.e. $lca(D) = lca(begin, end)$.*

The example below is a simple conjunctive keyword search query with only two keywords input and one result returned.

Example 3.3

Suppose a keyword query containing two keywords *XML* and *view* is applied to the XML document in Figure 3.1. The data node with value *Efficient Discovery*

of XML Data Redundancies (0.2.0.0) under the element node *title* will be found to contain one of the keywords *XML*. After that, in the data node with value *Answering Tree Pattern Queries Using Views* (0.3.0.0) under the element node *title* the other keyword *'view'* is found. An intuitive perception is conceived that the part containing these two data nodes, which is the content in the box in Figure 3.2, should be returned. However, since the query result should be subtrees, the LCA of the document fragment is finally returned in place of the subtree rooted at *conf* node. □

In the following chapter we will clarify the syntax and transformation of the keyword search query before we present the query processing in our work.

3.3 Anchor Nodes

We adopt the multiway approach in [21] for SLCA computation. As a result, we have to make the notion of *anchor node* as well as some of its properties clear since it is the central idea of the approach.

Let $K = \{w_1, \cdots, w_k\}$ denote an input set of k keywords,where each keyword w_i is associated with a set S_i of nodes in an XML document T (sorted in document order).A set of nodes $S = \{v_1, \cdots, v_k\}$ is defined to be a *match* for K if $|S| = |K|$ and each $v_i \in S_i$ for $i \in [1, k]$. We use S_i to denote the data node list (sorted in document order) associated with the keyword w_i.

Given two nodes v and w in a document tree T, $v \prec_p w$ denotes that v precedes

w (or w succeeds v) in document order in T; and $v \preceq_p w$ denotes that $v \prec_p w$ or $v = w$.

We use $v \prec_a w$ to denote that v is a proper ancestor of w in T, and $v \preceq_a w$ to denote that $v = w$ or $v \prec_a w$.

Consider a node v and a set of nodes S. The function $next(v, S)$ returns the first node in S that succeeds v if it exists; otherwise, it returns null. The function $pred(v, S)$ returns the *predecessor* of v in S, that is, the last node in S that precedes v if it exists; otherwise, it returns null.

The function $closest(v, S)$ computes the *closest node* in S to v as follows:

$$closest(v, S) = \begin{cases} pred(v, S) & \text{if } lca(v, next(v, S)) \prec_a \\ & lca(v, pred(v, S)), \\ next(v, S) & \text{otherwise.} \end{cases}$$

The function $closest(v, S)$ returns null if both $pred(v, S)$ and $next(v, S)$ are null; and it returns the non-null value if exactly one of $pred(v, S)$ and $next(v, S)$ is null. The function $lca(v, w)$ computes the *lowest common ancestor* (or LCA) of the two nodes v, w and returns null if any of its arguments is null.

Now we come to the notion of *anchor nodes*.

Definition 3.3.1 *A match $S = \{v_1, \cdots, v_k\}$ is said to be* anchored *by a node $v_a \in S$ if for each $v_i \in S - \{v_a\}$, $v_i = closest(v_a, S_i)$. We refer to v_a as the* anchor node *of S.*

The properties of the anchor node shown below guarantee that the matches are restrict to those that are anchored by some node. We omit the proofs and direct

interested readers to [21].

Lemma 3.3.2 *If $lca(S)$ is an SLCA and $v \in S$, then $lca(S) = lca(S')$, where S' is the set of nodes anchored by v.*

Lemma 3.3.3 *If $lca(S)$ and $lca(S')$ are distinct SLCAs, then $S \cap S' = \emptyset$.*

Lemma 3.3.4 *Let V and W be two matches such that $V \prec_p W$. If $lcaW$ is not a descendant of $lcaV$, then for any match X where $W \prec_p X$, $lcaX$ is also not a descendant of $lcaV$.*

Lemma 3.3.5 *Consider two matches S and S'. They are almost the same except for two nodes $u \in S$ and $v \in S'$, where $u \preceq_a v$, then $lca(S) \preceq_a lca(S')$.*

Lemma 3.3.5 can be easily deduced from Lemma 3.3.4

Along with the anchor node, now we need a triple `(begin, end, anchor)` to represent the anchored match. The label `anchor` stands for the anchor node of the match in SLCA computation. The other two labels remain the same meanings in the tuple `(begin, end)` representing a document fragment.

Chapter 4

Keyword Search Queries

The general form keyword search query we discussed is the combination of AND and OR boolean operators. Although the keyword queries can be expressed in either one of CNF and DNF, we seek a more general form that has no restrictions.

4.1 Query Syntax

The AND-OR keyword search queries are of the form:

$$Q \ = \ (Q) \mid (Q) \ AND \ (Q) \mid (Q) \ OR \ (Q) \mid k,$$

where k denotes some keyword.

The query syntax supports any combination of `AND` and `OR`. Conventionally AND operation will be applied prior to OR operation. An example query is as follows:

Example 4.1

VLDB AND ((XML AND views) OR (Jag AND Lakshmanan))

The query asks for any information containing *'VLDB'* as well as *'XML'* and *'views'*, or *'Jag'* and *'Lakshmanan'*. □

4.2 Query Transformation

To process the keyword search query, we should first parse the query and get the information of keywords and operators. The query will be transformed into a multiple-branched query tree, where the keywords and operators information are stored in the tree nodes.

There are two types of nodes in the query tree. The operator nodes represent the boolean operators in the query, and the keyword nodes represent the keywords in the query. Keyword nodes reside in leaves of the tree while the root and intermediate nodes are operator nodes. The child nodes of those operator nodes are the corresponding operands. Levels of the operator nodes are determined by the operation order as well as the association indicated by the parentheses. Accordingly, inner terms are lower in the query tree.

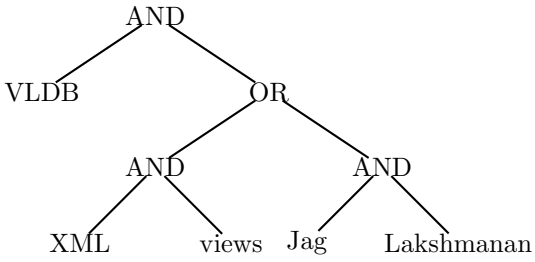


Figure 4.1: Eample Query Tree

For the query in Example 4.1, the corresponding query tree is illustrated in Figure 4.1. The two innermost terms *'XML AND views'*, and *'Jag AND Lakshmanan'* are at the bottom of the tree. They are connected by a parent operator node *OR*, which is the right child of the root node. The left child is another keyword *'VLDB'*. The root node `AND` denotes that the outermost operation is a conjunction.

For a node in the query tree, the type information (whether it is an AND operator, an OR operator or a keyword) is stored in the node. For each operator node we also maintain its child node list. If the query node is a keyword, its characters will be stored as well, which are used to get records from database. Besides, a database cursor is maintained for every keyword node marking the current position in the keyword data list in the database. If one keyword appears more than once in the query, multiple cursors will be maintained and accessed separately with regard to every appearance of the keyword. Consequently, no confusion will be caused.

We choose the tree structure not only because it is a good form that can represent any general form keyword search query with any combination of AND and OR operations, but also because tree structure can be easily decomposed and recomposed during processing. Every subtree of the query tree is a general form keyword search query itself. Thus the original query can be easily broken down to smaller and simpler subqueries. Those subqueries can be AND queries, OR queries, or queries only containing one keyword. Different processing approaches can be applied according to the types of these subqueries.

During the processing, the intermediate matching document fragment at each

query node is recorded in the form of the `(begin, end, anchor)` triple. Details of the algorithms will be discussed in the next chapter.

Chapter 5

AND-OR Query Processing

In this chapter, we present our approach for processing general form keyword search queries in XML data.

After the keyword search query has been parsed into the query tree, the processing begins from the root node and spreads downward to every tree node. It asks for one at a time appropriate matching document fragment from each child of the current node to be processed. The child nodes ask their children in the same way recursively, and matching document fragments are passed upward and processed according to the type of the parent node. If the parent node is an AND node, a conjunction of all the document fragments from child nodes is performed and a smallest document fragment covering all those document fragments is produced as a new match. If the parent node is an OR node, the most preceding one among all the document fragments from child nodes is chosen as the new match. All the intermediate matches at each query tree node are produced in the document sequence,

Algorithm 1 `General Keyword Search Algorithm`

```
1: result = ∅
2: while (getNext(root) ≠ null) do
3:     result = result ∪ lca(root.triple)
4: end while
5: return result
getNext (node)
1: if (node is a keyword node) then
2:     return getNextKey(node)
3: else if (node.operator = AND) then
4:     return getNextAnd(node)
5: else if (node.operator = OR) then
6:     return getNextOr(node)
7: end if
```

which is convenient for further computations above.

The algorithm for processing AND-OR general form keyword search queries is illustrated in Algorithm 1. At each query node, we use the function *getNext* to fetch the next suitable document fragment and record it in the triple attached to the node. The *getNext* function will direct the processing to different routines according to the type of query nodes. The smallest subtree containing the document fragment at the root node is computed via the *lca* function in step 3 and output to the final result set.

In the following, we will introduce the processing approach for each type of query node. We begin with the simplest one *getNextKey*.

5.1 Keyword Processing

Before we start to introduce the procedure of *getNextKey*, we first recall some properties of anchor node and LCA computation.

Algorithm 2 `Processing Keyword Nodes`

getNextKey (node)
Input: keyword node
Output: triple (begin, end, anchor)

1: $node.cursor = node.cursor + 1$
2: **if** $(nodeList[node.cursor] = null)$ **then**
3: return null
4: **else if** $(nodeList[node.cursor + 1] \neq null)$ **then**
5: **while** $nodeList[node.cursor] \preceq_a nodeList[node.cursor + 1]$ **do**
6: **if** $(nodeList[node.cursor + 1] \neq null)$ **then**
7: $node.cursor++$
8: **else**
9: break
10: **end if**
11: **end while**
12: **end if**
13: $node.triple.begin = node.triple.end = node.triple.anchor = nodeList[node.cursor]$
14: return *node.triple*

According to Lemma 3.3.5, if two matches S and S' only differ in two nodes v and w where $v \prec_a w$, then $lca(S) \prec_a lca(S')$. Given that we only accept as result the smallest document fragments that do not contain others, the match S will be pruned.

Based on this fact, we optimize the procedure of finding the next fitting keyword data node by skipping those that are ancestors of other nodes in the keyword node lists.

For every keyword node in the query tree we maintain a cursor, which is initiated to the first data node, marking the next data node in the corresponding keyword data lists to be processed. To get the next keyword data node, search begins from the cursor until a data node is found which is not the ancestor of the following (step 5 to 11). The triple is produced from the Dewey number of the data node

directly (step 13). The cursor is advanced every time the function *getNextKey* is called until it reaches the end and reports a null result (step 3).

Example 5.1

Consider a keyword *author* in Figure 3.2. The triples produced at this keyword node are (0.2.1.0, 0.2.1.0, 0.2.1.0), (0.2.1.1, 0.2.1.1, 0.2.1.1), (0.3.1.0, 0.3.1.0, 0.3.1.0), (0.3.1.1, 0.3.1.1, 0.3.1.1), and (0.3.1.2, 0.3.1.2, 0.3.1.2), in sequence. The cursor of the keyword has been advanced five times. □

5.2 And Processing

When we encounter an `AND` node, a conjunction should be performed among its child nodes, which is the function *getNextAnd* called in step 4 in the function *getNext* of Algorithm 1. The semantics of `AND` operation is to find a smallest document fragment which covers all matching document fragments from the child nodes.

The And processing approach is adopted from the *multiway-SLCA* algorithm in [21]. The detail of the function *getNextAnd* is demonstrated in Algorithm 3. A child list is maintained for each AND node. The child nodes are denoted as $child[i]$ in the algorithm, where i is from 1 to the amount of the child nodes denoted as $childCount$.

By Lemma 3.3.3, to compute the new match, document fragments from child nodes already used before should be skipped. As a result, new document fragments from child nodes are fetched, which is performed in steps 1 to 7. If any one of the

Algorithm 3 `Processing And Nodes`

getNextAnd (node)
Input: And node
Output: triple (begin, end, anchor)

1: **for** *each child*[*i*] **do**
2: {prepare each child node for candidate fragments}
3: $child[i].triple = getNext(child[i])$
4: **if** $(child[i].triple = null)$ **then**
5: return null
6: **end if**
7: **end for**
8: {choose the anchor node}
9: $node.triple.anchor = last(child[i].triple.anchor)\ for\ each\ i \in [1, childCount]$
10: **for** *each child*[*i*] **do**
11: {compute the anchored match}
12: $child[i].triple = getClosestTriple(child[i], node.triple.anchor)$
13: **end for**
14: $node.triple.begin = first(child[i].triple.begin)\ for\ each\ i \in [1, childCount]$
15: $node.triple.end = last(child[i].triple.end)\ for\ each\ i \in [1, childCount]$
16: return *node.triple*

child runs out of data nodes, no new matches can be found and the algorithm returns null (step 5).

Once the child nodes are ready, the anchor node is computed. In step 9, the last one among all the anchors of the document fragments from child nodes is selected as the anchor of the match. The corresponding anchored match is computed in steps 10 to 13, by choosing from each child node appropriate document fragments closest to the current anchor (based on Lemma 3.3.2). The selection is performed by comparing LCAs of the anchor and neighboring document fragments of the child node in the function *getClosestTriple* displayed in Algorithm 4. The function keeps on fetching the next document fragment of the current child node until it finds the lowest LCA. The match is found and represented as a triple of which *begin* is

Algorithm 4 getClosestTriple

```
getClosestTriple (node, anchor)
Input: query node, anchor
Output: triple (begin, end, anchor)
 1: olderTriple = node.triple
 2: while (getNext(node) ≠ null) do
 3:    if (lca(anchor, triple.anchor) ≺_a lca(anchor, olderTriple.anchor)) then
 4:       {older triple is closer}
 5:       node.triple = olderTriple
 6:       return node.triple
 7:    end if
 8: end while
 9: return node.triple
```

the first of all the *begins* and *end* is the last of all the *ends* of the child document fragments.

We use two examples to illustrate the detailed procedure of AND processing.

Example 5.2

Consider *'XML AND views'* in Example 3.3 and the document fragment in Figure 3.1. If the query is applied to the document fragment, the processing begins from the root node of the query tree, and function *getNextAnd(And)* is called. There are two children of the *AND* node: XML, and *views*, both are keyword nodes. Subsequently, *getNextKey(XML)* and *getNextKey(views)* are called. The first returns a triple (0.2.0.0, 0.2.0.0, 0.2.0.0), and the second returns (0.3.0.0, 0.3.0.0, 0.3.0.0). The latter one thus is selected as the anchor of current `AND` operation. The anchored match is computed and the triple (0.2.0.0, 0.3.0.0, 0.3.0.0) is returned as the matching fragment (exactly the content in the box in Figure 3.1).

□

Example 5.3

Consider another one *'author AND Jag'*. Functions *getNextKey(author)* and *getNextkey(Jag)* are called. The first returns a triple (0.2.1.0, 0.2.1.0, 0.2.1.0). The second returns a triple (0.2.1.1.0, 0.2.1.1.0, 0.2.1.1.0) and is chosen as the anchor. Based on it, the closest triple is computed. The LCA of current triple of *author* and the anchor is 0.2.1. The LCA of the next triple of *author* (0.2.1.1, 0.2.1.1, 0.2.1.1) and the author is 0.2.1.1 and is recognized as closer. The following triple is (0.3.1.0, 0.3.1.0, 0.3.1.0) and the LCA is 0, which is the ancestor of the previous LCA 0.2.1.1. As a result, the triple (0.2.1.1, 0.2.1.1, 0.2.1.1) is the closest to the anchor. The new match is computed according to step 14 and 15 in Algorithm 3 and a triple (0.2.1.1, 0.2.1.1.0, 0.2.1.1.0) representing the subtree rooted at the node *author* (with Dewey number 0.2.1.1) is returned. □

In steps 1 to 7, the child nodes are prepared in the sequence they appear in the query. Different from the SLCA computing algorithms in [21], the child nodes are not sorted according to the frequencies of their document fragments. That is because in the tree structure, it is quite costly to get all the document fragments sorted at each query node. Furthermore, the sorted lists in [21] can be reused because they are keyword data lists. In contrast, the sorted document fragments cannot be reused because they are computed according to given query terms. As a result, the sorting procedure is a waste in a sense.

On the other hand, due to the tree structure, the processing at the AND node stops once any one of its child nodes runs out of new matches. It ensures that

the total number of intermediate results produced is no more than the smallest among the numbers of document fragments from child nodes. At the same time, redundant computing of new document fragments from other child nodes is avoided and processing time as well as database accesses are saved.

Example 5.4

We continue with the processing of *'author AND Jag'* at the AND node in Example 5.3. Functions *getNextKey(author)* and *getNextkey(Jag)* are called again. The first returns a triple (0.3.1.0,0.3.1.0,0.3.1.0) and the second returns null. The checking at step 4 in Algorithm 3 reports a null result of the AND processing. Further calling of *getNextKey(author)* is skipped although there are two more document fragments at the *author* keyword node according to the result in Example 5.1. □

5.3 Or Processing

The semantics of OR operation is to combine the intermediate results from its child nodes by eliminating those document fragments that cover others. Then *getNextOr* finds one document fragment at a time which is a smallest independent one. By smallest, we mean that the fragment does not cover other fragments. By independent, we mean that the fragment does not intersect with others. Thus we need to compare the document fragments pairwise between every two child nodes, pruning those that do not suit until we output a fit one.

Thus the core of OR processing is the comparison. A straightforward method

can be as follows:

1. Compute all the document fragments from child nodes and put them in a set.
2. Compare every two document fragments by computing their LCAs.
3. Discard the document fragments whose LCAs are ancestors of others' and output the left to the result set.

Unfortunately, most of the time the naive method is unsatisfactory. First of all, the comparison between every two document fragments is quite time-consuming, even if the comparison within the same node can be skipped (given that the matches at the query node are output in document order). Secondly, quite a number of LCA computations are brought in on demand of the comparison. Unless the LCA of a document fragment is recorded, every time it is involved in a comparison, its LCA computation is performed again. Last and the most importantly, the processing pipeline in the query tree breaks down because we have to wait for the processing at the OR node finish producing all its matches and even worse, we need to sort the matches for further processing.

We thus attempt to find an optimized method avoiding the shortcomings listed above. The observation that the document fragments from one child nodes are naturally in document order assists the optimization against the large number of comparisons. Consider two document fragment D_1 and D_2 which are two matches at query node q_1, and another document fragment D_3 from query node q_2. If $D_3 \prec_p D_1 \prec_p D_2$ and D_3 is disjoint with D_1, then D_3 is also disjoint with D_2. This

is a generalization of Lemma 3.3.4. Based on this, if a preceding document fragment is disjoint with an early document fragment at some node, it won't get related with the document fragment produced at the same node in the following. That is to say, comparisons are not needed for obviously faraway document fragment pairs.

Furthermore, LCA computations are not always needed to decide whether a document fragment covers others. Recall that we use a triple `(begin, end, anchor)` to represent the document fragment. By comparing the labels in the triple, we can perceive the relationships between two document fragments at a smaller expense (It is apparent that the cost of comparing two Dewey numbers is cheaper than that of computing and comparing the LCAs of two pairs of Dewey numbers).

There can be three possible relationships between two document fragments, represented in the form of triples as follows:

For two matching document fragments A and B, the triple of A is *a(begin, end, anchor)*; the triple of B is *b(begin, end, anchor)*. Suppose $a.begin \preceq_p b.begin$:

1. $a.begin \preceq_p b.begin \preceq_p b.end \preceq_p a.end$

   A covers B.

2. $a.begin \preceq_p b.begin \preceq_p a.end \preceq_p b.end$ or $a.end \preceq_a b.begin$

   A intersects with B. Further LCA computing is needed to decide whether A covers B or B covers A.

3. $a.end \preceq_p b.begin$ and $a.end \npreceq_a b.begin$

   A and B are disjoint.

Thus we can infer the relationships between two document fragments by comparing their `begin` and `end` labels instead of comparing LCAs. Consequently, LCA computing is performed only when necessary. Unqualified intermediate matches are eliminated according to the result of comparison.

In case 1, A should be pruned because it contains a smaller match B. In case 2, the one that found to be the ancestor should be pruned. If the LCAs of two document fragments are by chance the same, any one of the document fragments can be pruned since they represent the same intermediate result. In case 3, neither of the two will be pruned. We can continue with the comparisons between other document fragments. If a document fragment is not pruned after it has been compared with its counterparts from all the other nodes, it will be output as a qualified match at the OR node.

In our approach, every child node of the OR node has a triple representing its current match except that those run out of new matches. If all the child nodes run out of new matches, the processing stops and returns null. If only one child node has new matches, its matching document fragment will be output directly as a match at the OR node without being compared. Otherwise, the comparisons will keep running and stops only when a match is output.

The detail of OR processing is shown in Algorithm 5. Before the central comparisons, some preparations are performed.

First of all, we prepare each child node for candidate document fragments by calling the function *checkChild* whose detail is demonstrated in Algorithm 6. If

Algorithm 5 `Processing Or Nodes`

getNextOr (node)
Input: Or node
Output: triple (begin, end, anchor)

```
1: if (checkChild(node) = false) then
2:     return null
3: end if
4: while (true) do
5:     prec = selectPrec(node)
6:     if ( prec = −1) then
7:         return null
8:     end if
9:     {the comparison begins}
10:    for (i = 0; i < childcount; i++) do
11:        while ((child[i].triple = null)or(i = prec)) do
12:            i++
13:        end while
14:        if (child[i].triple.begin ≺_p child[prec].triple.end) then
15:            if (child[i].triple.end ≺_p child[prec].triple.end) then
16:                {case 1}
17:                getNext(child[prec]); break
18:            else
19:                {case 2}
20:                if (ancestorLCA(child[prec], child[i]) = true) then
21:                    getNext(child[prec]); break
22:                else if (ancestorLCA(child[i], child[prec]) = true then
23:                    getNext(child[i])
24:                end if
25:            end if
26:        else if (child[prec].triple.end ≺_a child[i].triple.begin) then
27:            {case 2}
28:            if (ancestorLCA(child[prec], child[i]) = true) then
29:                getNext(child[prec]); break
30:            else if (ancestorLCA(child[i], child[prec]) = true then
31:                getNext(child[i])
32:            end if
33:        else
34:            {case 3}
35:        end if
36:        if (i = childCount) then
37:            {a round of comparison ends}
38:            node.triple = child[prec].triple
39:            child[prec].triple = getNext(child[prec])
40:            return node.triple
41:        end if
42:    end for
43: end while
```

Algorithm 6 `checkChild`

checkChild (node)
Input: Or node
Output: boolean

```
 1: count = 0
 2: for each child[i] do
 3:     if (child[i].triple = null) then
 4:         child[i].triple = getNext(child.[i])
 5:         count + +
 6:     end if
 7: end for
 8: if (count = node.childCount) then
 9:     return false
10: else
11:     return true
12: end if
```

Algorithm 7 `selectPrec`

selectPrec (node)
Input: Or node
Output: the *prec* index

```
 1: prec = 1
 2: while (child[prec].triple = null) do
 3:     prec + +
 4:     if (prec ≥ childCount) then
 5:         return -1
 6:     end if
 7: end while
 8: for (i = prec + 1; i < childCount; i + +) do
 9:     if (child[i].triple.begin ≺_p child[prec].triple.begin) then
10:         prec = i
11:     end if
12: end for
13: return prec
```

all the child nodes have no new matches any more, then no new matches can be computed at the OR node.

Since we want to output the matching document fragments in document order, it is straightforward that we start the comparison from the most preceding one among all the document fragments from child nodes. We select the first comparing triple by calling the function *selectPrec* which is displayed in Algorithm 7. If all

Algorithm 8 `ancestorLCA`

ancestorLCA (node, node)
Input: Query nodes n_1, n_2
Output: boolean

```
1: LCA_1 = lca(n_1.triple.begin, n_1.triple.end)
2: LCA_2 = lca(n_2.triple.begin, n_2.triple.end)
3: if (LCA_1 ≺_a LCA_2) then
4:     return true
5: else
6:     return false
7: end if
```

the child nodes have no new matches any more, *selectPrec* returns null indicating that no *prec* indexing the preceding document fragment exist. Otherwise, in steps 8 to 12 existing document fragments are compared by their `begin` label to decide the most preceding one to be returned.

After the preparation is done, a new round of comparison starts in step 10 in *getNextOr* in Algorithm 5. If the result of the comparison falls into case 2, the LCAs of the two document fragments have to be computed and compared by calling the function *ancestorLCA* in Algorithm 8 to decide whether any one of them should be pruned. If the *prec* triple is pruned in step 17 in case 1 or in steps 21 and 29 in case 2, the current round of comparison stops and a new round starts with an updated *prec* triple. Otherwise the comparison continues between the *prec* triple and the triples in the following, sometimes causing those triples updated. If the *prec* triple is not pruned after comparing with all the triples provided by other child nodes, then it is a suitable match and is returned (step 36 to 41).

It can be observed that the triple of child nodes are not necessarily updated every time *getNextOr* is called. They are only updated either when the triple has

not yet been produced or when the triple expires. Both the pruning in the cases above and the selection of the triple as matches can make the triple expire. Among those triples that are eligible as matches, we output them in document order. The order can be obtained at the same time the comparison runs.

Now we provide a whole view of our approach after the processing methods according to different types of query nodes have been introduced.

The search begins from the root node, and goes on in a top-down manner. Each child node of the root node is asked to provide a new one of theirs matches to compute the final match. Those intermediate query nodes then pass the requests to their children to compute their own matches. The request for matching document fragment is spread down until it reaches the leaf node i.e. keyword query node. Match at the leaf node is computed and a document fragment is returned to its parent node. The parent node gets all its child nodes ready for a match and then is able to compute one of its own match and returns the match to its parent node. When the root node finishes computing and finds a match to the query (or recognizes a null result to the query), the match is output and a new round of searching begins (or the searching stops).

We use an AND-OR query to demonstrate the processing detail of OR node as well as the flow of the whole query processing.

Example 5.5

Now consider the query *'(XML AND views) OR (author AND Jag)'*. The processing begins from the *OR* node in the query tree. It asks its two child nodes

for document fragments. Both of the *AND* nodes have not been processed yet and their triples are null. The processing then goes to *getNext(AND)* for both of them. The first returns a triple (0.2.0.0, 0.3.0.0, 0.3.0.0) (as in Example 5.2). The second one returns a triple (0.2.1.1,0.2.1.1.0,0.2.1.1.0) (as in Example 5.3). By checking the `begin` and `end` labels, the first document fragment are found to cover the second one and falls into case 1. The first one thus is pruned and the second triple (0.2.1.1,0.2.1.1.0,0.2.1.1.0) is returned.

The query processing continues and *getNextOr* is called again. At this time, the first *getNextAnd* returns null. If the second *getNextAnd* returns any match, the match will be output to result directly. However, as in Example 5.4, a null result is returned. Consequently, there is only one result found for the query *'(XML AND views) OR (author AND Jag)'*: the subtree rooted at element *author* with Dewey number 0.2.1.1. □

5.4 Analysis

We can observe that by delivering to the parent node the intermediate results immediately when a new one is produced, a pipeline is built in the query tree. We don't need to wait for all the matches of the child nodes coming out. The first searching result can be quickly output while the search is still running for following results. The quick response is a big satisfaction to keyword search end users.

Besides, since keywords are stored in database and fetched in document order,

and the processing at AND as well as OR node retain this property, matches are produced in document order naturally. The order in reverse assists in the processing at AND/OR node. Consequently, the cost of sorting search results is saved.

Different from the work in XKSearch [20] and in [21], our approach cannot utilize the frequency variation of the keywords appearing in the query for optimization. This is mainly because we cover the OR query in addition to absolute AND query. For an AND query, the result size is no larger than the size of the smallest intermediate result from its child nodes. However, for an OR query, the result size is no less than that of the largest intermediate match. It is possible that the size of result grows up to the sum of all the intermediate matches. Consequently, the OR query receives no benefit from the frequency bounding.

Furthermore, during processing we cannot pre-estimate the size of intermediate results especially when the query is a complex one whose query tree is deep and comprises of both AND and OR nodes. Even if we rearrange the keyword nodes according to their frequencies at the bottom of the query tree, we cannot control the processing flow to ensure that the intermediate nodes are still in frequency order. If we compute the frequencies of results for every intermediate nodes and get them rearranged at AND nodes to facilitate the processing, the cost is too expensive and not so rewarding. Worse still, the sorting requires all candidate matches to be ready, which spoils the pipeline.

Even though the frequency cannot be employed for optimization, the comparing of triples instead of LCA computing in our approach gains efficiency. Since

the keyword search query we study is in general form and no limit is set to its complexity, we cannot establish an upper bound of the time complexity for our algorithms. We will demonstrate our efficiency in the next chapter by extensive experiments instead.

Chapter 6

Performance Study

To verify the effectiveness as well as the efficiency of our approach, we conducted a comprehensive study to compare the performance against existing approaches for evaluating AND-OR keyword search queries.

6.1 Experimental Setup

We implemented our algorithms in Java using Apache Xerces XML parser and Berkeley DB [2]. The parser for keyword search query was also written in Java which builds a query tree before the query is processed.

Our experiments were conducted on the DBLP data [3]. All the data nodes are organized using a B+ tree where the keys are the keywords of the data nodes. The data associated with each key is a list of Dewey numbers of the data nodes directly containing the keyword.

We use AOG to refer to our general form AND-OR approach. The algorithm we mainly compared with is the AND-OR multiway-SLCA (AOMS) approach in [21]. Since the keyword search queries that can be processed in AOMS are limited to be in CNF, we rewrote the general form AND-OR queries into CNF for processing in AOMS. For example, the query *(algorithms AND 2005) OR (approach AND 1999)* will be rewritten into an equivalent query *(algorithms OR approach) AND (2005 OR approach) AND (2005 OR approach) AND (2005 OR 1999)*.

IAOMS is the indexed version of AOMS. The difference between AOMS and IAOMS is that IAOMS uses a lookup style method to find the next match while AOMS scans its keyword lists to get the next match. However, our approach can only apply the scanning method as we do not necessarily have a ready-for-use list to look up for the next match. That is due to the pipelined processing which produces only one new intermediate result for each query node when asked by their parent nodes. As a result, we do not have an indexed version of AOG and we compare AOG with both AOMS and IAOMS.

We also implemented two binary variants for comparing, AOSE for AND-OR Scan Eager and AOILE for AND-OR Indexed Lookup Eager. They are extensions of the binary approaches in [20] for AND-OR queries. Similar to AOMS and IAOMS, AOSE and AOILE can only be applied to CNF queries.

We generated general form AND-OR keyword search queries by varying the following parameters: the number of keywords in the query N, the height of the query tree H, and the frequency of each keyword. We also vary the query structure

to investigate performances of varied queries.

Our experiments were conducted on a 3.0GHz desktop with 1GB of RAM running Windows XP.

6.2 Experimental Results

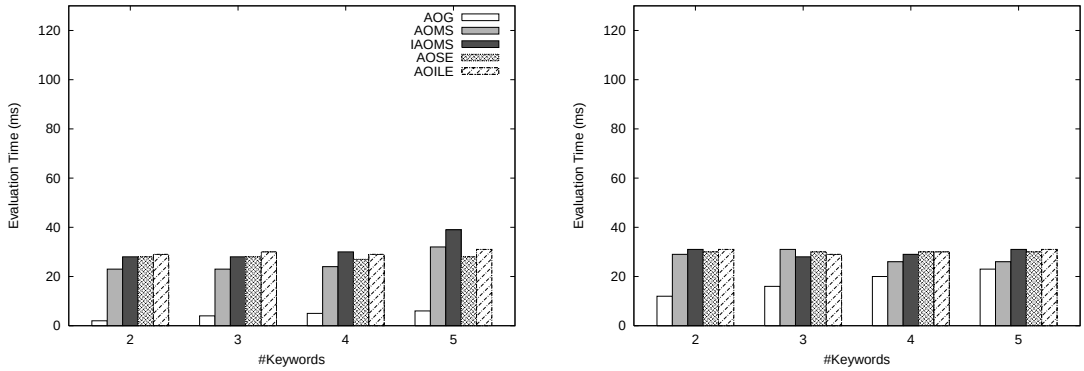
As mentioned above, AOMS, IAOMS, AOSE and AOILE can only process keyword queries in CNF. Consequently, they cannot be applied to pure OR queries which our approach can easily deal with. We omit the performance study of pure OR queries here as a result.

First of all, we compare our approach with the *multiway-SLCA* approach in pure AND queries.

Experiment 1. Pure AND Queries

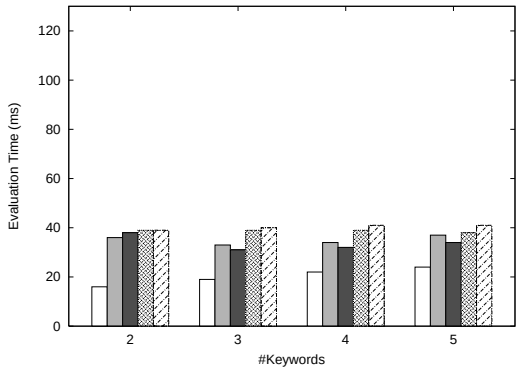
Pure AND queries refer to keyword search queries that consist of AND nodes and keywords only, for example, *focus AND peer AND ieee*. In this experiment, we vary the number of keywords from 2 to 5 and compare the performances of the 5 approaches. The results are displayed in Figure 6.1 under different keyword frequencies. In Figure 6.1(a), 6.1(c) and 6.1(e), all the keywords have the same frequencies of 10, 100 and 1000 respectively. In Figure 6.1(b), 6.1(d), and 6.1(f), frequencies of keywords varies from 10 to 100, 10 to 1000, and 100 to 1000 respectively.

In the binary and *multiway-SLCA* approach, all the keyword lists are sorted.

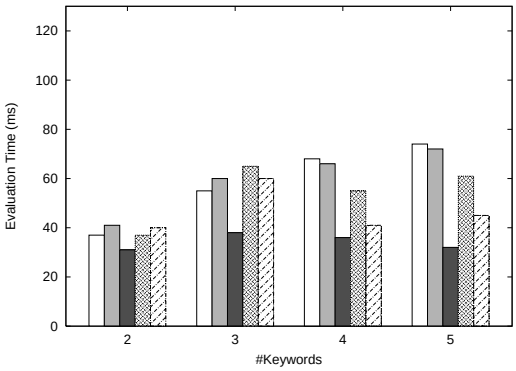


(a) small frequency = 10

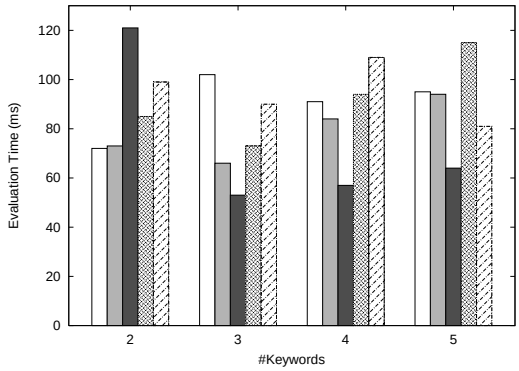
(b) frequency (10, 100)



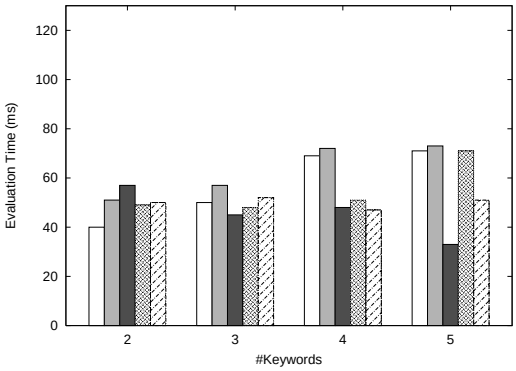
(c) medium frequency = 100



(d) frequency (10, 1000)



(e) large frequency = 1000



(f) frequency (100, 1000)

Figure 6.1: Pure AND Queries

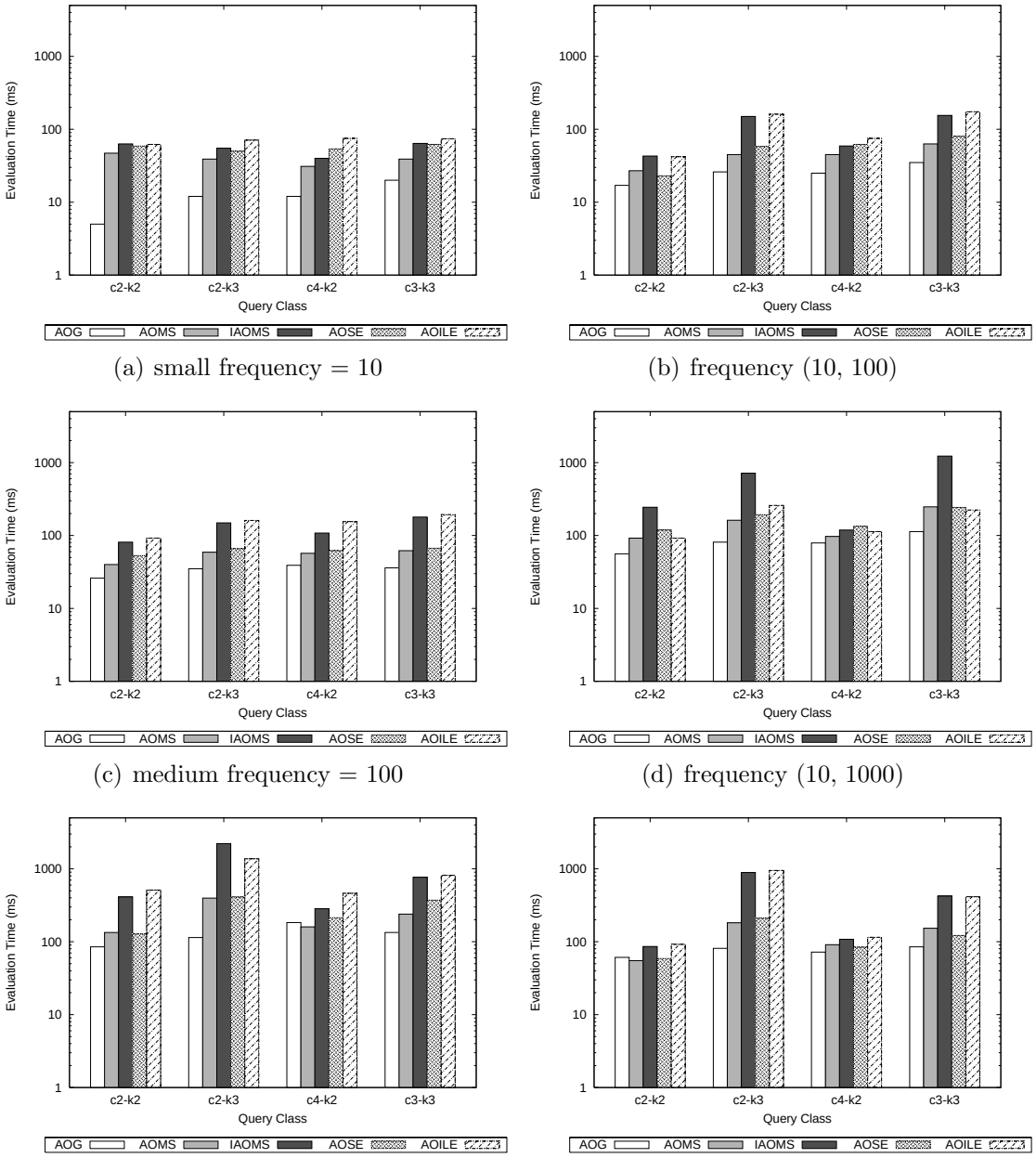
Their database cursors also get ready before matches are computed. As a result, no matter the keyword frequency is large or small, the evaluation time always includes a startup cost which is only related to the number of keywords. As AOG does not

perform a pre-sorting and only accesses the keyword data nodes during the query processing, its performance is more related to the keyword frequency. When the keyword frequency is small, AOG takes advantage of zero startup cost and ends quickly (as in Figure 6.1(a), 6.1(b), 6.1(c)). When the keyword frequency is large, the influence of startup costs in the binary and *multiway-SLCA* approaches decrease and their optimizations utilizing the sorted keyword lists to get the next match take effect. Consequently, they reveal better performances (as in Figure 6.1(e)). Besides, when the frequencies vary significantly (as in Figure 6.1(d) and 6.1(f)), the indexed lookup method IAOMS and AOILE are more efficient. Genrally AOSE and AOILE reveal worse performences than AOMS and IAOMS. But still they outperform AOG in pure AND queries.

Experiment 2. CNF Queries

CNF queries can be directly be processed by the binary and *multiway-SLCA* approaches. We adopted the AND processing method from their SLCA computing approach but did not introduce their optimization making use of frequency knowledge because this optimization can only be applied in conjunctive computation and can not be generalized into AND-OR processing. Nevertheless, with OR processing introduced, in each conjunction the sorting cost increases compared to pure AND queries in the binary and *multiway-SLCA* approaches. In contrast, the label comparing method instead of LCA computation for AND and OR processing in AOG saves up the time cost and redeems the weakness mentioned above.

The results of CNF query is demonstrated in Figure 6.2. The evaluation time



(a) small frequency = 10

(b) frequency (10, 100)

(c) medium frequency = 100

(d) frequency (10, 1000)

(e) large frequency = 1000

(f) frequency (100, 1000)

Figure 6.2: CNF Queries

on the y-axis is in logscale. Each class of queries is denoted by cM-kN, where M denotes number of conjunctions in the query and N denotes number of keywords in each conjunction. Then the number of keywords is N multiplied by M.

It is noticed that for CNF queries, the number of keywords in each conjunction M has a larger impact than the number of conjunctions in the binary and *multiway-SLCA* approaches, especially for the indexed version IAOMS and AOILE. However, our approach is less sensitive to the query structure and exhibits a steady trend that the evaluation time is linear to the number of keywords in the query. This is due to the spread-down processing style in the query tree.

In average, the evaluation time is reduced by 50 percent using our approach compared with the evaluation time of AOMS. We also outperform IAOMS greatly especially when the number of keywords in each conjunction exceeds 3. The performances of AOSE and AOILE are even worse when the keywords have similar frequencies. But when the frequency varies, AOILE has a relatively better performance than the *multiway-SLCA* approach although AOG is still the winner.

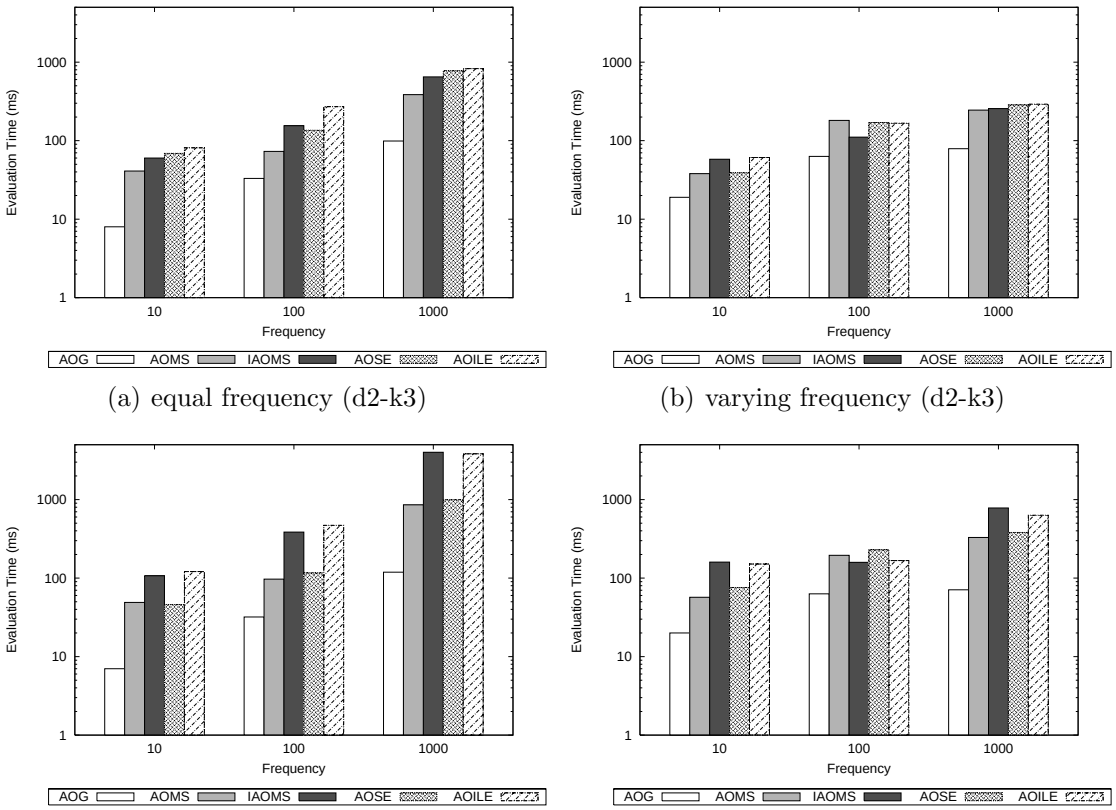
Experiment 3. DNF Queries

Since DNF queries cannot be directly processed by *multiway-SLCA* approach, query rewriting is needed. Generally, the transformed CNF query is more complex than the original DNF query with keywords duplicated. For example, the simplest CNF for the query

(editor AND 1999) OR (1997 AND ieee)OR (2001 AND c.) is

(editor OR 1997 OR 2001) AND (editor OR 1997 OR c.) AND (editor OR 2001 OR ieee) AND (1997 OR 2001 OR 1999) AND (1997 OR c. OR 1999) AND (editor OR c. OR ieee) AND (2001 OR 1999 OR ieee) AND (c. OR 1999 OR ieee)

The original DNF query will not be viewed as a very complex one if it is processed by AOG. However, its CNF counterpart may be quite a challenge for the *multiway-SLCA* approach.



(a) equal frequency (d2-k3)

(b) varying frequency (d2-k3)

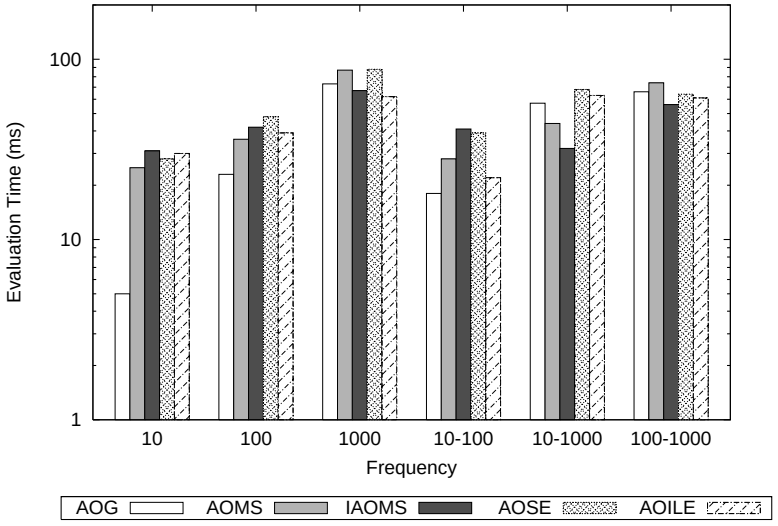
(c) equal frequency (d3-k2)

(d) varying frequency (d3-k2)

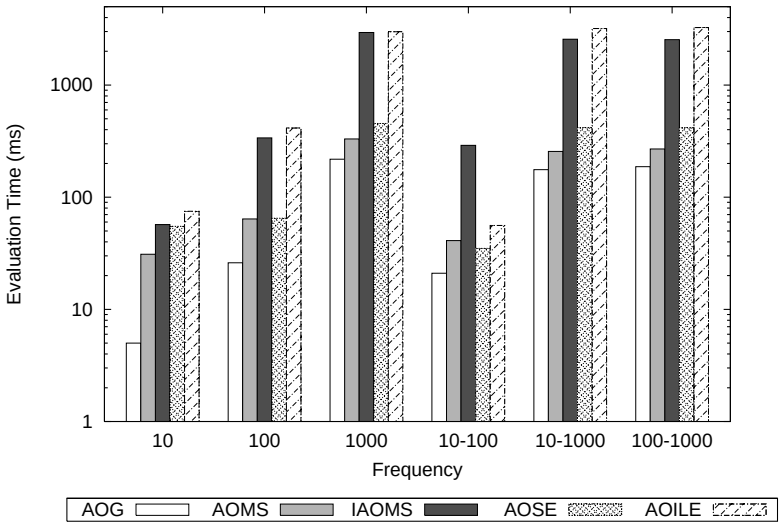
Figure 6.3: DNF Queries

In Figure 6.3, queries are classified in a similar way with CNF queries. The dM-kN in the caption of each figure denotes the number of disjunctions M in the query and the number of keywords N in each disjunction. Our approach obviously beats the other 4 by a significant magnitude. The average processing cost of AOG is 10 percent of the costs of AOMS and IAOMS, and 5 percent of the costs of AOSE and AOILE.

Experiment 4. Deep AND-OR Queries



(a) AND rooted

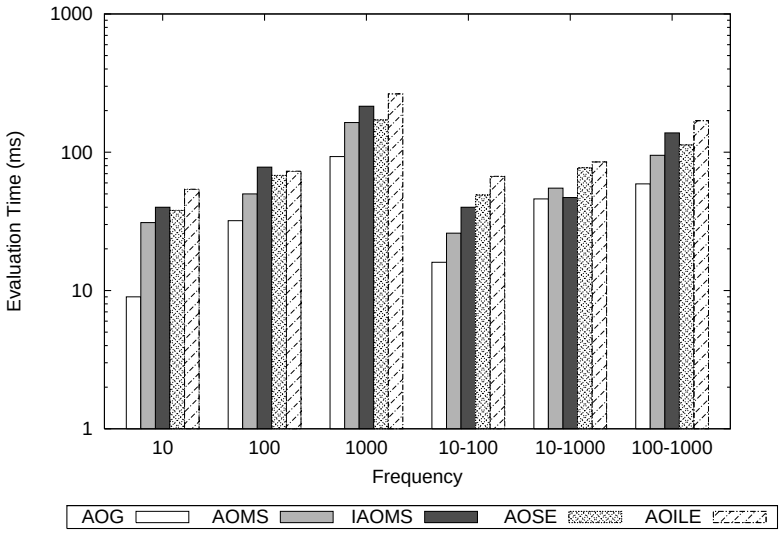


(b) OR rooted

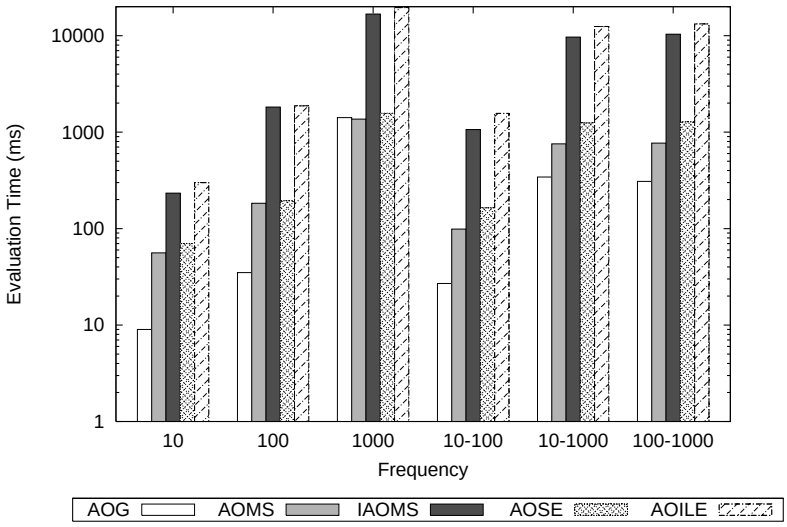
Figure 6.4: Queries With Depth of 4

We now examine the performance of deep AND-OR queries with a depth more than 3 in the query tree. Both CNF queries and DNF queries discussed before are shallow queries with a depth of 3. Since our approach is a pipelined one, the processing time is related to the length of the pipeline, i.e. the depth of the query

tree. Thus, deep AND-OR queries require longer processing time.



(a) AND rooted



(b) OR rooted

Figure 6.5: Queries With Depth of 5

In Figure 6.4 are the results of queries whose depth is 4. In Figure 6.5 are queries with depth 5. Compare the performances in Figure 6.4(a) and 6.4(b), we can find that the evaluation time of queries with an OR node as the root node of the query tree is far more than that with an AND node as the root node. Similar

trend can also be found in Figure 6.5(a) and 6.5(b).

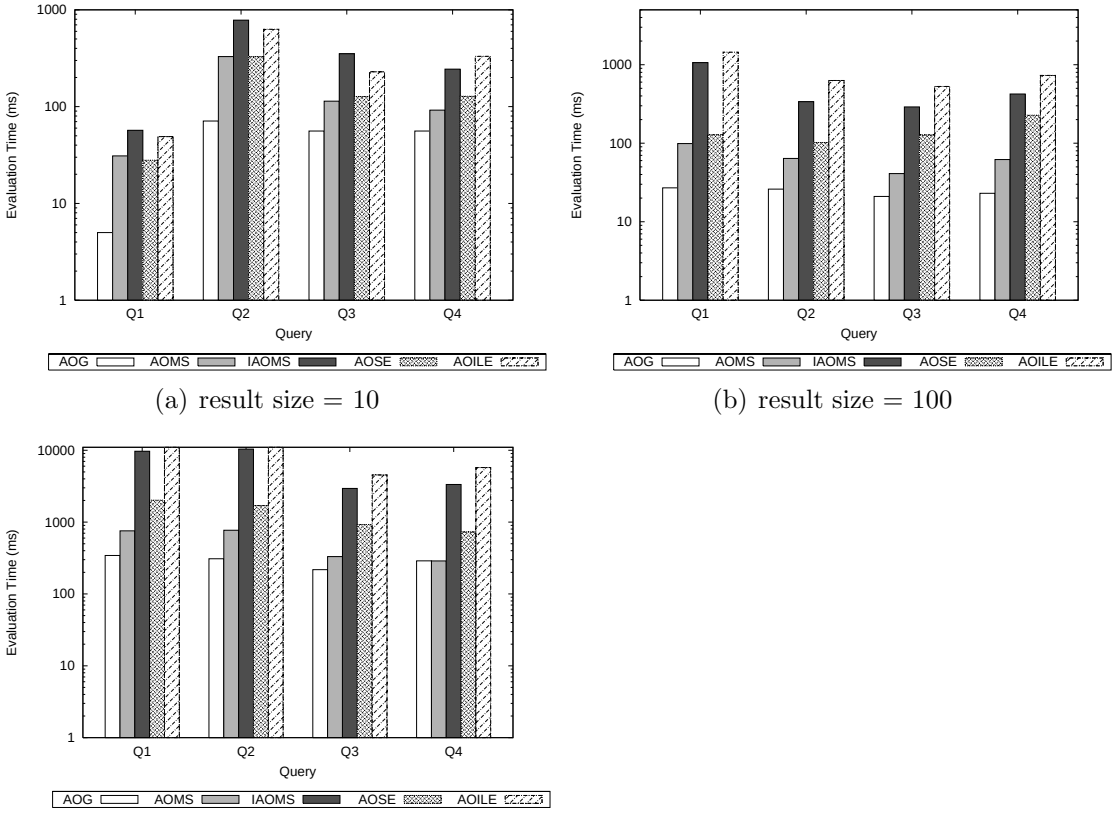
Furthermore, the increase in the evaluation time is not much when the root node is an AND node, comparing Figure 6.4(a) and 6.5(a). In contrast, there is a remarkable increase in the evaluation time when the root node is an OR node and the depth of the query changes from 4 to 5 (Figure 6.4(b) and 6.5(b)). Comparing the performances in both figures, it shows once again that AOG has a better capability of processing disjunctions while the binary and *multiway-SLCA* approaches are efficient only for conjunctive processing.

Experiment 5. Result Size

In the following two experiments, we try to find out other factors which have an impact on the evaluation time of our algorithm. We have demonstrated in the previous experiments that the frequency of keywords, as well as the query structure (for example, depth of the query, type of root node) are tightly connected with the performance.

Another factor related to the evaluation time of AND-OR queries is found to be the size of the final results, as indicated in Figure 6.6. Queries are generated randomly and grouped according to their result size. Evaluation time is noted down and compared.

When the result size is less than or equal to 10, the evaluation time is quite diverse, as in Figure 6.6(a). However, when the result size approaches 100 or more, the evaluation time for AOG, AOMS as well as IAOMS all fall into a relatively stable range respectively (in Figure 6.6(b) and 6.6(c)). In Figure 6.1(e), when the result



(a) result size = 10

(b) result size = 100

(c) result size = 1000

Figure 6.6: Queries With Varying Result Size

size is around 1000, AOSE and AOILE show very bad performances compared with the others. That is because of the large amount of intermediate results generated during the processing. When the result size is small, AOSE and AOILE sometimes can have better performances.

AOG still reveals better performance than AOMS and IAOMS.

Experiment 6. Vary Rewriting

We infer from Experiment 4 that the depth of the query may affect the evaluation time. It is also shown in Experiment 4 that the query structure have an impact as well. However, if the queries with different depths and structures but represent

the same semantics, will the structure difference affect the evaluation time? To investigate this, we choose Queries 13-15 and rewrite them into several equivalent queries with different depths and structures and compare their evaluation times.

Query 13:

1. (2005 AND views AND chapter)AND (information OR algorithms OR analysis)

2. (2005 AND (views AND chapter))AND (information OR (algorithms OR analysis))

3. 2005 AND views AND (chapter AND (information OR (algorithms OR analysis)))

4. 2005 AND (views AND (chapter AND (information OR (algorithms OR analysis))))

Query 14:

1. (2005 AND views) OR (chapter AND information) OR (algorithms OR analysis)

2. (2005 AND views) OR ((chapter AND information) OR (algorithms OR analysis))

3. (2005 AND views) OR (((chapter AND information) OR algorithms) OR analysis)

4. (((((2005 AND views) OR chapter)AND information) OR algorithms) OR analysis)

Query 15:

1. (2001 AND pages) OR (ieee AND database) OR (algorithms OR approach)

2. (2001 AND pages) OR ((ieee AND database) OR (algorithms OR approach))

3. (2001 AND pages) OR (((ieee AND database) OR algorithms) OR approach)

4. (((((ieee AND database) OR pages) AND 2001) OR algorithms) OR approach)

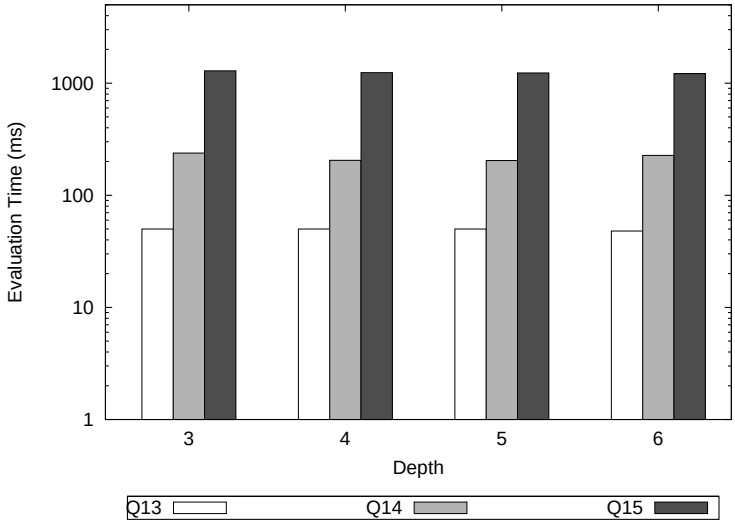


Figure 6.7: Varying Structure for Equal Queries

The evaluation times are shown in Figure 6.7. The x-axis denotes the depth of the query. We can notice that evaluation times hardly change with the transformation of the queries. That means for a given keyword search, no matter in which form it is expressed, our approach will return with similar response time. This is a useful property for keyword search processing because we do not need to rewrite the input queries for efficiency consideration. The search engine system is simplified while time cost is saved.

Chapter 7

Conclusion

In this thesis, we have presented a novel approach to process general form AND-OR keyword search queries. To the best of our knowledge, this is the first work to handle keyword queries with any combination of AND and OR operators.

We utilize the tree structure to represent the keyword search query. The query can be easily parsed into a query tree, with keywords in leaf node and operators in root as well as intermediate nodes, and operands attached as children of the operator nodes. Using the query tree, not only the query is naturally divided into several subqueries in the form of subtrees in the query tree, but also the processing can be broken up and specialized according to the type of the query nodes. Consequently, no matter how many types of general form queries there are, the processing methods we need to consider are now limited to three: how to process the keyword node in the query tree, and the AND operator node as well as the OR node.

We adopted the AND processing from SLCA computing algorithms ([16], [18], [20], [21]) and proposed a comparison mechanism for OR processing which prunes intermediate results that cover other intermediate results. By delivering to the parent node the intermediate results immediately when a new one is produced, a pipeline is built in the query tree. We do not need to wait for all the matches of the child nodes coming out. The first searching result can be quickly output while the search is still running for following results. Quick response time is critical to keyword search end users. An important benefit due to the tree structure and the pipelined-approach is that the impact of increase in keyword numbers in the query on query processing is reduced by logarithm.

The efficiency of our approach is verified via comprehensive experiments. Although the evaluation time is increasing with an increase in keyword frequency, our approach has exhibited satisfying processing response and outperforms *multiway-SLCA* approach in most cases especially when the query is a complex one. We also find by experimental studies that our approach responds steadily to equivalent queries in different structures. That avoids query rewriting due to the complexity and is surely to benefit both end users and search engine designers.

Our current work in this thesis still cannot handle queries with NOT operator, which is commonly used in full-texted keyword searches. As part of our future work, we intend to extend our approach to deal with complex keyword search queries with any combination of AND, OR, and NOT operators. Besides, our search returns the precise answers. Some other approximate answers that may interest the users

thus are completely rejected. Another direction consequently lies in integrating proximity search as well as ranking mechanism into our approach.

Bibliography

[1] V. Vesper. *Let's Do Dewey*. http://www.mtsu.edu/ vvesper/dewey2.htm.

[2] Berkeley DB. http://www.sleepycat.com.

[3] DBLP. http://www.informatik.uni-trier.de/ ley/db.

[4] W3C. *XML Path Language(XPath) 1.0*. http://www.w3.org/TR/xpath.

[5] Scott Boag, D. Chamberin, Mary Fernandez, Daniela Florescu, Jonathan Robie, Jerome Simeon. *XQuery 1.0: An XML query language*. http://www.w3.org/TR/xquery.

[6] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. *Storing and Querying Ordered XML Using a Relational Database System*. SIGMOD 2002.

[7] S. Agrawal, S. Chaudhuri, G. Das. *DBXplorer: A System for Keyword-Based Search over Relational Databases*. ICDE 2002.

[8] V. Hristidis, Y. Papakonstantinou. *DISCOVER: Keyword Search in Relational Databases*. VLDB 2002.

[9] V. Hristidis, Y. Papakonstantinou, A. Balmin. *Keyword Proximity Search on XML Graphs*. ICDE 2003.

[10] G. Bhalotia, C. Nakhe, A. Hulgeri, S. Chakrabarti, S. Sudarshan. *Keyword Searching and Browsing in Databases using BANKS*. ICDE 2002.

[11] J. Plesnik. *A Bound for the Steiner Tree Problem in Graphs*. *Math Slovaca*, 31:155-163, 1981.

[12] L. Guo, F. Shao, C. Botev, J. Shanmugasundaram. *XRank: Ranked Keyword Search over XML Documents*. SIGMOD 2003.

[13] N. Fuhr, K. Grobjohann. *XIRQL: A Query Language for Information Retrieval in XML documents*. SIGIR 2001.

[14] A. Theobald, G. Weikum. *The Index-Based XXL Search Engine for Querying XML Data with Relevance Ranking*. EDBT 2002.

[15] S. Cohen, J. Mamou. Y. Kanza, Y. Sagiv. *XSearch: A Semantic Search Engine for XML*. VLDB 2003.

[16] A. Schmidt, M. Kersten, M. Windhouwer. *Querying XML Documents Made Easy: Nearest Concept Queries*. ICDE 2001.

[17] D. Florescu, D. Kossmann, L. Manolescu. *Integrating Keyword Search into XML Query Processing*. WWW 2000.

[18] Y. Li, C. Yu, H. V. Jagadish. *Schema-free XQuey*. VLDB 2004.

[19] V. Hristidis, N. Koudas, Y. Papakonstantinou, D. Srivastava. *Keyword Proximity Search on XML Trees*. TKDE 2006.

[20] Y. Xu, Y. Papakonstantinou. *Efficient Keyword Search for Smallest LCAs in XML Databases*. SIGMOD 2005.

[21] C. Sun, C.-Y. Chan, A. K. Goenka. *Multiway SLCA-based Keyword Search in XML Data*. WWW 2007.

[22] V. Aguilera, S. Cluet, F. Wattez. *Xyleme Query Architecture*. WWW 2001.

[23] S. Cluet. *Designing OQL: Allowing Objects to be Queried. Information Systems*, 23(5): 279-305, 1998.

[24] D. Harel, R. E. Tarjan. *Fast Algorithms for Finding Nearest Common Ancestors. SIAM J. Comput.*, 13(2): 338-355, 1984.

[25] B. Schieber, U. Vishkin. *on Finding Lowest Common Ancestors: Simplification and Parallelization. SIAM J. Comput.*, 17(6): 1253-1262, 1988.

[26] S. Brin, L. Page *The Anatomy of a Large-scale Hypertextual Web Search Engine. Computer Networks*, 30(1-7): 107-117, 1998.

[27] S. Amer-Yahia, E. Curtmola, A. Deutsch. *Flexible and Efficient XML Search with Complex Full-text Predicates*. SIGMOD 2006.