An Adaptive Path Index for XML Data using the Query Workload *

Jun-Ki Min\textsuperscript{a,1} Chin-Wan Chung\textsuperscript{a,2} Kyuseok Shim\textsuperscript{b,3}

\textsuperscript{a} Division of Computer Science
Department of Electrical Engineering \& Computer Science
Korea Advanced Institute of Science and Technology
373-1, Kusong-dong, Yusong-gu, Taejon, 305-701, Korea

\textsuperscript{b} School of Electrical Engineering and Computer Science
Seoul National University
San 56-1, Shilim-dong, Kwanak-gu, Seoul, 151-742, Korea

Abstract

Due to its flexibility, XML is becoming the \textit{de facto} standard for exchanging and querying documents over the Web. Many XML query languages such as XQuery and XPath use label paths to traverse the irregularly structured XML data. Without a structural summary and efficient indexes, query processing can be quite inefficient due to an exhaustive traversal on XML data. To overcome the inefficiency, several path indexes have been proposed in the research community. Traditional indexes generally record all label paths from the root element in XML data and are constructed with the use of data only. Such path indexes may result in performance degradation due to large sizes and exhaustive navigations for partial matching path queries which start with the self-or-descendent axis(“/”). To improve the query performance, we propose an adaptive path index for XML data (termed APEX). APEX does not keep all paths starting from the root and utilizes frequently used paths on query workloads. APEX also has a nice property that it can be updated incrementally according to the changes of query workloads. Experimental results with synthetic and real-life data sets clearly confirm that APEX improves the query processing cost typically 2 to 69 times compared with the traditional indexes, with the performance gap increasing with the irregularity of XML data.

\textit{Key words:} XML, Semistructured Data, Path Index, Query Processing

* A preliminary version of this paper is to appear in the proceedings of ACM SIGMOD 2002.
\textsuperscript{1} Email: jkmin@islab.kaist.ac.kr
\textsuperscript{2} Corresponding author. Email: chungcw@islab.kaist.ac.kr
\textsuperscript{3} Email: shim@ee.smu.ac.kr

Preprint submitted to Elsevier Science 20 May 2002
1 Introduction

The emergence of the World Wide Web has dramatically increased the amount of data of all kinds available electronically. And, there has been an increasing interest in XML (eXtensible Markup Language) [5]. XML can describe a wide range of data, from regular to irregular, from flat to deeply nested, and from tree shaped to graph shaped. Due to its flexibility, XML is rapidly emerging as the de facto standard for exchanging and querying documents on the Web required for the next generation Web applications including electronic commerce and intelligent Web searching. XML data is an instance of semistructured data [1]. XML documents comprise hierarchically nested collections of elements, where each element can be either atomic (i.e., raw character data) or composite (i.e., a sequence of nested subelements). Tags stored with elements in an XML document describe the semantics of the data. Thus, XML data, like semistructured data, is hierarchically structured and self-describing.

Several XML query languages [2,4,7,8,11] have been also proposed recently. XML Query languages such as XPath [8] and XQuery [4] use path expressions to traverse irregularly structured XML data. Thus, the navigation of irregularly structured graph is one of essential components for processing XML queries. Since the objects may be scattered at different locations in the disk, processing XML queries may result in significant performance degradation. Furthermore, query processing with a label path for partial matching is very inefficient due to the navigation of an entire XML data graph. However, structural summaries or path indexes can speed up query evaluation on XML data by restricting the search to only relevant portions of the XML data. Thus, the extraction of the structural summary and index structures for the semistructured data in order to improve the performance of query processing have received a lot of attention recently. Examples of such index structures include DataGuides [14], 1-indexes [19], the Index Fabric [9], and extensions of inverted indexes [17,23]. The details on these index structures are described in Section 2.

DataGuides and 1-indexes are in the category of generalized path indexes that represent all paths starting from the root in XML data. They are generally useful for processing queries with path expressions starting from the root. However, these indexes are very inefficient for processing queries with partial matching due to the exhaustive navigation of the indexes. Furthermore, these path indexes are constructed with the use of data only. Therefore, they do not take advantages of query workloads to process frequently used path expressions effectively.

Our Contributions. In this paper, we propose APEX which is an Adaptive
Path indEx for XML data. APEX does not keep all paths starting from the root and utilizes frequently used paths to improve the query performance. In contrast to the traditional indexes such as DataGuides, 1-indexes and the Index Fabric, it is constructed by utilizing a data mining technique to summarize paths that appear frequently in query workloads. APEX also guarantees to maintain all paths of length two so that any label path query can be evaluated by joins of extents in APEX without scanning original data. APEX has the following novel combination of characteristics to improve the performance of processing queries.

- **Efficient Processing of Partial Matching Queries**: Since traditional path indexes keep all label paths from the root element, they are efficient to handle queries with a simple path expression which is a sequence of labels starting from the root of the XML data. However, partial matching queries with the self-or-descendent axis (“/”) should be rewritten to queries with simple path expressions. The problem here is that, the exhaustive navigation for the query rewriting degrades the performance. In contrast to traditional path indexes, APEX is designed to support these path expressions efficiently. APEX consists of two structures: a hash tree and a graph structure. Since the hash tree maps incoming label paths to nodes of graph structure, the nodes needed to compute the results of partial matching path queries are obtained easily.

- **Workload-Aware Path Indexes**: Traditional path indexes for semistructured data are constructed with the use of data only. Therefore, it is very difficult to tune the indexes toward efficient processing of frequently used queries. In APEX, frequently used path expressions in query workloads are taken into account using the sequential pattern mining technique [3,13] so that the cost of query processing can be improved significantly. Although any sequential pattern mining algorithm with the anti-monotonicity property can be used to extract frequently used paths, the problem space of the sequential pattern mining is subtly different from ours. Thus, we devise a simple algorithm that counts all subpaths in query workloads by one scan.

- **Incremental Update**: When we decide to rebuild APEX due to the change of query workloads, we do not build APEX from the scratch. Instead, APEX is incrementally updated in order to reduce the overhead of construction.

We implemented our APEX and conducted an extensive experimental study with both real-life and synthetic data sets. Experimental results show that APEX improves query processing cost typically 2 to 69 times better than existing indexes, with the performance gap increasing with the irregularity of XML data.

The remainder of the paper is organized as follows. In Section 2, we discuss related work. In Section 3, we present the data model and basic notations for APEX. We present an overview of APEX in Section 4 and describe the
construction algorithms for APEX in Section 5. Section 6 contains the results of our experiments, showing the effectiveness and comparing the performance of APEX to existing path indexes. Finally, Section 7 summarizes our work.

2 Related Work

Many database researchers developed various path indexes to support label path expressions. Goldman and Widom [14] provided a path index, called the strong DataGuide. The strong DataGuide is restricted to a simple label path and is not useful in complex path queries with several regular expressions [19]. The building algorithm of the strong DataGuide emulates the conversion algorithm from the non-deterministic finite automaton (NFA) to the deterministic finite automaton (DFA) [15]. This conversion takes linear time for tree structured data and exponential time in the worst case for graph structured data. Furthermore, on very irregularly structured data, the strong DataGuide may be much larger than the original data.

Milo and Suciu [19] provided another family of indexes (1/2/T-index). Their approach is based on the backward simulation and the backward bisimulation which are originated from the graph verification area. The 1-Index coincides with the strong DataGuide on tree structured data. The 1-Index can be considered as a non-deterministic version of the strong DataGuide.

In object-oriented databases, access support relations [16] are used to support frequently used reference chains between two object instances. Therefore, it materializes access paths of arbitrary lengths and thus it can be used for indexing XML documents. Note that access support relations and the T-index support only predefined subsets of paths.

Cooper et al. [9] presented the Index Fabric which is conceptually similar to the strong DataGuide in that it keeps all label paths starting from the root element. The Index Fabric encodes each label path to each XML element having a data value as a string and inserts the composition of the encoded label path and the data value into an efficient index such as the Patricia trie. The index block and XML data are both stored in relational database systems. The evaluation of queries encodes the desired path traversal as a search key string and performs a lookup. The Index Fabric does not keep all parent-child relationships of elements since it does not keep the information of XML elements which do not have data values. Thus, the Index Fabric is not effective to process partial matching queries.

Many queries to XML data has the partial matching path expression because users of XML data may not be concerned with the structure of data and inten-
tionally make the partial matching path expression to get the intended result. Since the strong DataGuide, the 1-Index, and the Index Fabric record only paths starting from the root in the data graph, the query processor rewrites partial matching path queries into the queries with simple path expressions by the exhaustive navigation of index structures [12,18]. This results in performance degradation. In contrast, APEX is constructed from label paths which are frequently used in query workloads. Thus, APEX is very effective for queries with partial matching expressions.

3 Preliminary

In this section, we describe our representation of XML data and define some basic notations to explain our proposed index.

As shown in Figure 1, we represent the structure of XML data as the labeled directed graph which is similar to the OEM model [22]. Especially, two particular attributes, ID and IDREF, allow us to represent the structure of XML data as a graph.

**Definition 1** The structure of XML data is represented by the directed labeled edge graph $G_{XML}$. $G_{XML} = (V, E, root, A)$, $V = V_c \cup V_a$ where $V_c$ is the universe of non-leaf nodes and $V_a$ is the universe of leaf nodes, $E \subseteq V_c \times A \times V$ where $A$ is the universe of labels, $root \in V$ is the root of $G_{XML}$. Each node in $G_{XML}$ has a unique node identifier (nid).

Since XML elements are ordered and the results of XML queries must be in document order, each node keeps the document-order information. And nodes returned by the index are sorted using this information as a post-processing step. As shown in Figure 1, the reference relationship (i.e., ID-IDREF) is represented as an edge from a node for an IDREF typed attribute to a node for an element which has the corresponding ID typed attribute. In addition, the label of the edge from an element to IDREF typed child node starts with '@' and the edge for the reference relationship has the normal tag, which is the tag for the target node, as its label.

**Definition 2** A label path of a node $o$ in $G_{XML}$, is a sequence of one or more dot-separated labels $l_1 l_2 \ldots l_n$, such that we can traverse a path of $n$ edges $(e_1 \ldots e_n)$ from $o$, where the edge $e_i$ has the label $l_i$.

In Figure 1, movie.title and name are both valid label paths of node 7. In XML data, queries are based on label paths such as //movie/title.

**Definition 3** A data path of a node $o$ in $G_{XML}$ is a dot-separated alternating
Fig. 1. A sample XML data

sequence of labels and nids of the form \( l_1.o_1.l_2.o_2 \ldots l_n.o_n \), such that we can
trace from \( o \) a path of \( n \) edges \( (e_1 \ldots e_n) \) through \( n \) nodes \( (x_1 \ldots x_n) \), where
the edge \( e_i \) has the label \( l_i \) and the node \( x_i \) has the nid \( o_i \).

In Figure 1, \( movie.8.title.10 \) and \( name.11 \) are data paths of node 7.

**Definition 4** A data path \( d \) is an instance of a label path \( l \) if the sequence
labels made from \( d \) by eliminating nids is equal to \( l \).

Again in Figure 1, \( movie.8.title.10 \) is an instance of \( movie.title \) and \( name.11 \)
is an instance of \( name \).

**Definition 5** A label path \( A = a_1.a_2 \ldots a_n \) is contained in another label path
\( B = b_1.b_2 \ldots b_m \) if we have \( a_1 = b_i, a_2 = b_{i+1}, \ldots, a_n = b_{i+n-1} \) where \( 1 \leq i \) and
\( i + n - 1 \leq m \). When \( A \) is contained in \( B \), we also call that \( B \) contains \( A \) or \( A \)
is a subpath of \( B \). Furthermore, when \( A \) is a subpath of \( B \) and \( m = i + n - 1 \),
we call that $A$ is a suffix of $B$. ■

For example, the label path movie is a subpath of movie.title. And, the label path title is a suffix of movie.title.

4 Overview of APEX

In this section, we propose an example and the formal definition of APEX.

An example of APEX for Figure 1 is shown in Figure 2 when the required paths $= A \cup \{\text{director.movie, @movie.movie, actor.name}\}$ (see Definition 6). It is not necessary to know how to construct APEX at this point. The purpose of this example is to help understanding the definitions in this section.

As shown in Figure 2, APEX consists of two structures: a graph structure ($G_{APEX}$) and a hash tree ($H_{APEX}$). $G_{APEX}$ represents the structural summary of XML data. It is useful for query pruning and rewriting. $H_{APEX}$ represents incoming label paths to nodes of $G_{APEX}$. $H_{APEX}$ consists of nodes, called the hnode and each hnode contains a hash table. In an hnode, each entry of the hash table points to another hnode or a node of $G_{APEX}$ but not both. That is, each node of $G_{APEX}$ maps to an entry of an hnode of $H_{APEX}$. $H_{APEX}$ is a useful structure to find a node of $G_{APEX}$ for given label path. Furthermore, $H_{APEX}$ is useful in the incremental update phase (see details in Section 5.2). Also, each node of $G_{APEX}$ corresponds to an extent. The extent is similar to the materialized view in the sense that it keeps the set of edges whose ending nodes are the result of a label path expression of a query.

The strong DataGuide and the 1-Index of Figure 1 are shown in Figure 3. Note that, the strong DataGuide is larger than the original data and the 1-Index is
equal to the structure graph of XML data. The following XPath query q1 is an example query that retrieves all actors’ names.

q1: //actor/name

To compute q1 on the strong DataGuide in Figure 3(a), the edge lookup occurs 14 times on the index structure to prune and rewrite q1 at compile-time [18]. The query processor obtains the extent for MovieDB.actor.name. The behavior of query processor on the 1-Index is similar to that of the strong DataGuide.

However, APEX in Figure 2 is very efficient to compute q1 since the query processor just looks up the hash tree with actor.name in the reverse order. That is, the hash tree of APEX enables efficient finding of the nodes of $G_{APEX}$ for partial matching path queries.
Since making an effective index structure for all the queries is very hard, APEX changes its structure according to the frequently used paths. To extract frequently used paths, we assume that a database system keeps the set (= workload) of queries (= label paths). Furthermore, we adopt the support concept of the sequential pattern mining to identify frequently used paths [3,13].

Let the support of a label path \( p = l_i \ldots l_j \), denoted by \( \text{sup}(p) \), be the ratio of the number of queries having \( p \) as a subpath to the total number queries. Also, let \( \text{minSup} \) denote the user-specified minimum support.

**Definition 6** A label path \( p = l_i \ldots l_j \) in \( G_{XML} \) is a frequently used path if \( \text{sup}(p) \geq \text{minSup} \). Let \( p \) be a required path if it is either a frequently used path or the length of \( p \) is one. ■

**Definition 7** For a label path \( p \) of the form \( l_i, l_{i+1} \ldots l_j \) in \( G_{XML} \), an edge set, \( T(p) \), is \( \{ <o_{j-1}, o_j> \mid l_i.o_i \ldots l_{j-1}.o_{j-1}.l_j.o_j \) is a data path in \( G_{XML} \} \). That is, an edge set \( T(p) \) is a set of pairs of nids for the incoming edges to the last nodes that are reachable by traversing a given label path \( p \). ■

For example, the edge set \( T(\text{title}) \) in Figure 1 is \( \{<8,10>, <14,17>\} \). Label paths from \( \text{root} \) to 10 and 17 include \( \text{MovieDB.movie.title} \), \( \text{MovieDB.director.movie.title} \), \( \text{MovieDB.actor.movie.movie.title} \), \( \text{MovieDB.director.movie.movie}@\text{director.movie.movie.title} \). And, the common suffix of these label paths is “title”.

**Definition 8** Given a required path set \( R \), a label path \( p \in R \) is a maximal suffix in \( R \) if there is no path \( q \in R \) of which \( p \) is a suffix except \( p \). ■

For example, in Figure 2, we assume that the required paths = \( A \cup \{ \text{director.movie, @movie.movie, actor.name} \} \). In this case, \( \text{actor.name} \) is a maximal suffix in the required paths since a required path whose suffix is \( \text{actor.name} \) does not exist except \( \text{actor.name} \).

Consider a subset of required paths \( Q = \{ q \mid q = l_i \ldots l_j \} \) whose common suffix is \( p = l_k \ldots l_j \). That is, \( p \) is not a maximal suffix in the required paths. In this case, making each edge set \( T(q) \) and \( T(p) \) increases the storage overhead since \( T(q) \subseteq T(p) \). Thus, we introduce the target edge set \( T^R(p) \) instead of the edge set \( T(p) \).

**Definition 9** Let \( Q_{XML} \) be a set of label paths of the root node in \( G_{XML} \). For each label path \( p \) in the required path set \( R \), let \( Q_G(p) = \{ l \mid l \in Q_{XML} \) and \( p \) is a suffix of \( l \} \), let \( Q_A(p) = \{ l \mid l \in Q_{XML} \) and every path \( q (\neq p) \) \( \in R \) having \( p \) as a suffix is a suffix of \( l \} \), and let \( Q(p) = Q_G(p) - Q_A(p) \). Finally, a target edge set, \( T^R(p) = \cup_{r \in Q(p)} T(r) \). ■

Note that, by Definition 9, if \( p \) is a maximal suffix in the required paths, \( T(p) = T^R(p) \) since \( Q_A(p) = \{ \} \). If \( p \) is not a maximal suffix in the required paths,
$T^R(p)$ is pointed to by a *remainder* entry of $H_{APEX}$. Again in Figure 2, we assume that *name* and *actor.name* are required paths. Thus, as shown in Figure 1, $T(\textit{actor.name}) = \{<2,3>, <4,5>\}$, and $T(\textit{name}) = \{<2,3>, <4,5>, <7,11>, <12,13>\}$. In this case, $T^R(\textit{actor.name}) = T(\textit{actor.name})$ since $Q_A(\textit{actor.name}) = \{\}$. However, $T(\textit{name}) \neq T^R(\textit{name})$. By definition 9, $Q_G(\textit{name}) = \{\textit{MovieDB.director.name}, \textit{MovieDB.actor.name}, \textit{MovieDB.movie.} @ \textit{actor.actor.name}, \ldots \}$, and $Q_A(\textit{name}) = \{\textit{MovieDB.actor.name}, \textit{MovieDB.movie.} @ \textit{actor.actor.name}\}$ since *actor.name* is a required path and has *name* as a suffix. Then $Q(\textit{name}) = \{\textit{MovieDB.director.name}, \textit{MovieDB.director.movie.} @ \textit{director.director.name}, \ldots \}$. Thus, $T^R(\textit{name}) = \{<7,11>, <12,13>\}$.

Now, we will define APEX.

**Definition 10** Given a $G_{XML}$ and a required path set $R$, APEX can be defined as follows. We introduce the root node of $G_{APEX}$, $xroot$ in APEX which corresponds to the root node of $G_{XML}$. By considering every required path $p \in R$, we introduce a node of $G_{APEX}$, with an incoming label path $p$, that keeps $T^R(p)$ as an extent only if $T^R(p)$ is not empty. For each edge $v.l.v'$ in $G_{XML}$, there is an edge $x.l.x'$ in APEX where the extent of $x'$ contains $<u, v'>$ and the extent of $x$ contains $<u, v>$ where $u$ is a parent node of $v$.

The following theorem proves that APEX is sufficient for the path index. By the definition of the simulation [6], if there is a simulation from $G_{XML}$ to $G_{APEX}$, all the label paths on $G_{XML}$ exist on $G_{APEX}$. Thus, all queries based on label paths can be evaluated on APEX.

**Theorem 1** There is a simulation from $G_{XML}$ to $G_{APEX}$.

**Proof:** Given $G_{APEX} = (V_x, E_x, xroot, A)$ and $G_{XML} = (V, E, r, A)$, there is a simulation from $r$ to $xroot$. Suppose, there is a simulation from $v \in V$ to $x \in V_x$, a full label path $q$ to $v$ is $l_1 \ldots l_m$, and $\exists v.l_{m+1}.v' \in E$. By Definition 10, there is a node $x'$ for $T^R(p' = l_i \ldots l_{m+1}$ where $1 \leq i \leq m)$ whose incoming path is $p'$, and $\exists x.l_{m+1}.x' \in E_x$. Therefore, there is a simulation from $G_{XML}$ to $G_{APEX}$. ■

Furthermore, $G_{APEX}$ satisfies the following theorem.

**Theorem 2** All the label paths whose lengths are 2 on $G_{APEX}$ are on $G_{XML}$.

**Proof:** Recall that APEX groups the edges with respect to the incoming label paths. By Definition 10, $\forall$ edge $x.l_j.x' \in E_x$, $\exists v.l_j.v' \in E$. By Definition 6, an incoming label path of $x$ is a label of incoming edge of $x$. Therefore, by letting the label of incoming edge of $x$ be $l_i$, the label path $l_i.l_j$ exists on $G_{XML}$. ■
APEX is a general path index since APEX at least keeps all the label paths whose length is 2 and at most keeps all the label paths on $G_{XML}$ corresponding to the frequently used paths. Thus, any label path query can be evaluated by look-up of $H_{APEX}$ and/or joins of extents.

5 Construction and Management of APEX

The architecture of the APEX management tool is illustrated in Figure 4. As shown in the figure, the system consists of three main components: the initialization module, the frequently used path extraction module and the update module.

![Architecture of APEX Management tool](image)

The initialization module is invoked without query workloads only when APEX is built first. This module generates $APEX^0$ that is the simplest form of APEX and is used as a seed to build a more sophisticated APEX. As the query workload is collected with the use of the current APEX, the frequently used paths are computed and used to update the current APEX dynamically into a more detailed version of APEX. The last two steps are repeated whenever query workloads change. Although the decision on how often the last two steps invoke is important, it is not the focus of our paper and we omit any further discussion on this issue since it depends on the preference of end user (e.g., by request or periodical).

5.1 $APEX^0$: Initial index structure

$APEX^0$ is the initial structure to build APEX. This step is executed only once at the beginning. Since there is no workload at the beginning, the required path set has paths of length one that is equivalent to the set of all labels in the XML data.
An example of the APEX⁰ for the XML data in Figure 1 is presented in Figure 5. The structure of APEX⁰ is similar to that of the 1-Representative Object (1-RO) proposed as a structural summary in [20]. As 1-RO contains all paths of length two in the XML data, APEX⁰ includes every label path of length two. However, in APEX, we have not only the structural summary in \( G_{APEX} \) but also the extents in the nodes of \( G_{APEX} \).

The algorithm of building APEX⁰ is shown in Figure 6. Each node in APEX⁰ represents a set of edges that have the same incoming label. Basically, we traverse every node in XML data (\( G_{XML} \)) in the depth first fashion. We first visit the root node of XML data (\( G_{XML} \)) and generate the root node for \( G_{APEX} \) first. We add an edge <NULL, root > to the extent of the root node in \( G_{APEX} \). Since each node in \( G_{APEX} \) represents a unique label and there is no incoming label for the root node of APEX, we represent the root node with a special incoming label ‘xroot’ for convenience. We then call the function exploreAPEX0 with the root node in \( G_{APEX} \) and the extent of the root node in \( G_{APEX} \).

In each invocation of exploreAPEX0, we have two input arguments: the newly visited node \( x \) in \( G_{APEX} \) and new edges just added to the extent of \( x \) in the previous step. We traverse all outgoing edges from the end point of the edges in \( \Delta ESet \) and group them by labels. Then we process edges in each group having the same label \( l \) one by one.

Let us assume that the node representing the label \( l \) in \( G_{APEX} \) is \( y \). Intuitively, we need to put the edges having the label \( l \) to the extent of \( y \) and connect \( x \) and \( y \) in \( G_{APEX} \). Then, we call exploreAPEX0 recursively with \( y \) and the newly added edges to the extent of \( y \). We give only newly added edges at this step as \( \Delta ESet \) for recursive invocation of exploreAPEX0 since the outgoing edges from the edges included previously to the extent have been all traversed already.

When we consider the edges for each group with a distinct label, we have to
Procedure buildAPEX0(root)
begin
1. $xnode := hash('xroot')$
2. $xnode.extent := \{<\text{NULL}, root>\}$
3. exploreAPEX0($xnode, xnode.extent$)
end
Procedure exploreAPEX0($x, \Delta ESet$)
begin
1. $ESet := \emptyset$
2. for each $<u, v> \in \Delta ESet$ do
3. $ESet := ESet \cup \{o \mid o \text{ is an outgoing edge from } v\}$
4. for each unique label $l$ in $ESet$ do
5. $y := hash(l)$
6. if ($y = \text{NULL}$) {
7. $y := newXNode()$
8. insert $y$ into hash table
9. }
10. make_edge($x, y, l$)
11. $\Delta\text{newESet} := \text{a set of edges having } l \text{ in } ESet \setminus y.\text{extent}$
12. $y.\text{extent} := y.\text{extent} \cup \Delta\text{newESet}$
13. exploreAPEX0($y, \Delta\text{newESet}$)
14. }
end

Fig. 6. An algorithm to build APEX$^0$

check whether the node $y$ exists already. To find this, we can maintain a hash table and use it. In case, the node $y$ for the label $l$ does not exist, we generate a new node and put it to $G_{APEX}$. We also make sure that the new node can be located quickly by inserting the node into $H_{APEX}$. The $hash$ at Line (5) in Figure 6 is the hash function which returns a node for a given label. The procedure $make\_edge$ makes an edge from $x$ to $y$ with label $l$ if there is not a edge. For preventing the infinite traversal of cyclic data, we do not consider the edges which are already in the extent of the node $y$.

5.2 Frequently Used Path Extraction

Any sequential pattern mining algorithms such as the one in [3,13] may be used to extract frequently used paths from the path expressions appearing in query workloads. While we need to use the traditional algorithms with the anti-monotonicity property [21] for pruning, we have to modify them.

Consider a mail order company. Assume that many customers buy $A$ first, then $B$ and finally $C$. In the traditional sequential pattern problem, when a sequence of $(A, B, C)$ is frequent, all subsequences of length two (i.e., $(A, B)$, $(A, C)$, $(B, C)$) including $(A, C)$ are frequent. However, for the problem of
finding frequently used path expressions, it is not valid any more. In other words, even though the path expression of \(A.B.C\) is frequently used, the path expression of \(A.C\) may not be frequent. Therefore, in case we want to use traditional data mining algorithms which use the anti-monotonicity pruning technique, we need a minor modification to handle the difference.

Actually, we also found that the size of the query workload is not so large as that of data for sequential pattern mining applications. Thus, we used a naïve algorithm in our implementation that simply counts all subpaths that appear in the query workload by one scan.

![Diagram of path extraction process](image)

**Fig. 7.** The behavior of frequently used path extraction

The basic behavior of the frequently used path extraction module is described in Figure 7. Each entry of hash table in a node of \(H_{APEX}\) consists of five fields: \(label\), \(count\), \(new\), \(xnode\), and \(next\). The \(label\) field keeps the key value for the entry. The \(count\) field keeps the frequency of label path which is represented by the entry. The \(new\) field is used to check a newly create entry in a node of \(H_{APEX}\). The \(xnode\) field points to a node in \(G_{APEX}\) whose incoming label
path is represented by the entry. Finally, the next field points to another node in $H_{APEX}$. For simplicity, we omit new fields in Figure 7.

Suppose that the required path set was \{A, B, C, D, B.D\}. Then, the current state of $H_{APEX}$ is represented as Figure 7-(a). A label path B.D is represented as an entry in the subnode of D entry in the root node(HashHead) of $H_{APEX}$.

Let the workload $Q_{workload}$ become \{A.D, C, A.D\}. We first count the frequency of each label path which appeared in $Q_{workload}$ and store the counts in $H_{APEX}$. When we count, we do not use the remainder entry for counting. Figure 7-(b) shows the status of $H_{APEX}$ after the frequency count.

Finally, we prune out the label paths whose frequency is less than minSup. The status of $H_{APEX}$ after pruning is illustrated in Figure 7-(c). Assuming that minSup is 0.6, the label path whose frequency is less than 2 is removed. Thus, a label path B.D is pruned. However, label paths B and C still remain since a label path of length one is always in the required path set. Also, in the pruning step, the xnode fields, which are not valid any more by the change of frequently used paths, are set to NULL. The content for $T^R$(D) in Figure 7-(a) representing remainder.D in $H_{APEX}$ is the set of edges whose end nodes are reachable by traversing a label path D but not B.D. However, the content for $T^R$(D) in Figure 7-(c) should be changed to the set of edges whose end nodes are reachable by traversing a label path D but not A.D. Thus, we set the xnode field of remainder entry to NULL to update it later.

The algorithm of frequently used path extraction is presented in Figure 8. To extract frequently used paths from the given workload ($Q_{workload}$), the algorithm first sets count fields to 0 and new fields to FALSE of all entries in $H_{APEX}$.

The algorithm consists of two parts; the first part counts frequencies and the second part is the pruning phase. $H_{APEX}$ is used to keep the change of the workload. The algorithm invokes the procedure frequencyCount to count the frequency of each label path and its subpaths in $Q_{workload}$. In this procedure, the new field of a newly created entry in a node in $H_{APEX}$ is set to TRUE to identify a newly created entry in a node in $H_{APEX}$ during pruning phase.

The function pruning$H_{APEX}$ removes the hash entry whose frequency is less than the given threshold minSup(Line (4)-(5)). Even though the frequency of an entry in the root node (HashHead) of $H_{APEX}$ is less than minSup, it should not be removed since a label path of length one should always be in the required path set by Definition 6. If the frequency of an entry of the node in $H_{APEX}$ is less than minSup and the entry is not in the root node, the entry is removed from the hash node by the function hnode.delete (Line (6)-(7)). If all entries in a node of $H_{APEX}$ except the remainder entry are removed by the function hnode.delete, hnode.delete returns TRUE. And then we remove this
**Procedure** frequentlyUsedPathExtraction()

**begin**
1. reset all count fields to 0 and new fields to FALSE
2. frequencyCount()
3. pruning\(H_{APEX}(\text{HashHead})\)

**end**

**Function** pruning\(H_{APEX}(hnode)\)

**begin**
1. is_empty := FALSE
2. if \(hnode = \text{NULL}\) return is_empty
3. for each entry \(t \in hnode\) do {
4. if \((t.count < \text{minsup})\) {
5. \(t.next := \text{NULL}\)
6. if \((t \not\in \text{HashHead})\) {
7. is_empty := hnode.delete(t)
8. }
9. } else {
10. if (pruning\(H_{APEX}(t.next) = \text{TRUE}\))
11. \(t.next := \text{NULL}\)
12. if \((t.next \not= \text{NULL})\) and \((t.xnode \not= \text{NULL})\)
13. \(t.xnode := \text{NULL}\)
14. if \((t.new = \text{TRUE})\) and \((hnode.\text{remainder}.xnode \not= \text{NULL})\)
15. \(hnode.\text{remainder} := \text{NULL}\)
16. }
17. }
18. return is_empty

**end**

Fig. 8. Frequently Used Path Extraction Algorithm

node of \(H_{APEX}\) (Line (10)-(11)).

Finally, it sets the \(xnode\) field to NULL because it points to the wrong node in \(G_{APEX}\). As mentioned earlier, contents of some nodes of \(G_{APEX}\) may be affected by the change of frequently used paths. There are two cases. First, a label path \(q\) was a maximal suffix previously but it is not anymore. This is captured by the fact that an entry has not NULL value in both \(xnode\) and \(next\) fields (Line (12)-(13)). In this case, the algorithm sets the \(xnode\) field to NULL to update the \(xnode\) field appropriately later. Second, a new frequently used path influences the contents of the node of \(G_{APEX}\) for the \(remainder\) entry in the same node of \(H_{APEX}\) since the contents of \(remainder\) is affected by the change of frequently used paths. This is represented by the fact that a new entry appears in the node of \(H_{APEX}\) and the \(remainder\) entry in this node points to a node in \(G_{APEX}\) using the \(xnode\) field (Line (14)-(15)). In this case, the algorithm sets the content of the \(xnode\) field in the \(remainder\) entry to NULL to update it later.
5.3 The Update with Frequently Used Paths

After the entries in $H_{APEX}$ was updated with frequently used paths computed from the changes of query workloads, we have to update the graph $G_{APEX}$ and $xnode$ fields of entries in the nodes of $H_{APEX}$ that locates the corresponding node in $G_{APEX}$.

After extracting of frequently used paths, each entry in a node of $H_{APEX}$ may have a pointer to another node of $H_{APEX}$ in the $next$ field or a pointer to the node of $G_{APEX}$ in the $xnode$ field, but the entry cannot have non-NULL value for both $next$ and $xnode$ fields. If the entry of a node $n$ in $H_{APEX}$ has a pointer to another node $m$ of $H_{APEX}$, there exists a longer frequently used path represented by the entry in $m$ whose suffix is represented by the entry in the node $n$. For example, consider the example of $H_{APEX}$ in Figure 7-(c). The $next$ field of the entry with the label $D$ in $H_{APEX}$ points to a node that has two entries; one is for the path $A.D$ and the other is for the rest of paths ending with $D$ except $A.D$. Recall that the paths are represented in $H_{APEX}$ in the reverse order. If the $xnode$ field in the entry of a node $n$ in $H_{APEX}$ points to a node $g$ in $G_{APEX}$, the extent of $g$ has edges with an incoming label path represented by the entry in $n$.

The basic idea of update is to traverse the nodes in $G_{APEX}$, check the validity of the nodes in $G_{APEX}$ with respect to $H_{APEX}$, and update not only the structure of $G_{APEX}$ with frequently used paths but also the $xnode$ field of entries in $H_{APEX}$.

Function lookup($path = l_1 \ldots l_n$)
begin
1. $hnode :=$ HashHead // the root of $H_{APEX}$
2. for $i = n$ to 1 do {
3. $t := hnode.hash(l_i)$
4. if ($t = NULL$) {
5. // $H_{APEX}$ keeps the $l_a.l_{i+1} \ldots l_n$ where $l_a \neq l_i$.
6. //Thus, $l_1 \ldots l_n$ is mapped to remainder.$l_{i+1} \ldots l_n$
7. return $hnode.remainder.xnode$
8. } else { // $H_{APEX}$ keeps the path for $l_1 \ldots l_n$
9. if ($t.next = NULL$)
10. return $t.xnode$ // $l_1 \ldots l_n$ is mapped to $l_i \ldots l_n$
11. else
12. $hnode := t.next$
13. }
14. }
end

Fig. 9. Lookup Function of APEX
First of all, we present the hash tree look-up function in Figure 9 which is a core function of the update module of APEX. To update $G_{APEX}$ according to the change of frequently used paths, we identify the nodes in $G_{APEX}$ with respect to $H_{APEX}$. Thus, before visiting a node in $G_{APEX}$, we look for $H_{APEX}$ using the $\text{lookup}$ function. The input of the $\text{lookup}$ function is the label path from the root to a node in $G_{APEX}$ and the output is the $\text{xnode}$ field of the entry which represents the longest suffix of the input, where the longest suffix is among required paths. Note that an entry for the longest suffix path always exists in $H_{APEX}$ since, by the definition of the required path (See Definition 6), the last label of the path to look for in $H_{APEX}$ always exists. Thus, while traversing the nodes in $G_{APEX}$, we can check the validity of the nodes in $G_{APEX}$ with respect to the change of frequently used paths.

Consider that the input of the $\text{lookup}$ function is $l_1 \ldots l_n$. The $\text{lookup}$ function looks for the nodes in $H_{APEX}$ iteratively in the reverse order of $l_1 \ldots l_n$ (Line (2)). After accessing a node in $H_{APEX}$ by $l_{i+1} \ldots l_n$, the $\text{lookup}$ function looks for the entry for $l_i \ldots l_n$ using the $\text{hash}$ function (Line (3)).

If there is no entry for $l_i \ldots l_n$, the $\text{lookup}$ function returns the $\text{xnode}$ field of the remainder entry of the node since $H_{APEX}$ keeps the entry for $l_i l_{i+1} \ldots l_n$ (where $l_n \neq l_i$) and the longest suffix of $l_1 \ldots l_n$ among required paths is $l_{i+1} \ldots l_n$ (Line (4)-(7)).

When the entry for $l_i \ldots l_n$ exists (Line (8)-(13)), the $\text{lookup}$ function checks the $\text{next}$ field whether the entry keeps the subnode in $H_{APEX}$ or not (Line (9)). If the $\text{next}$ field is NULL, the label path $l_i \ldots l_n$ is a maximal suffix of required paths. Thus, $\text{lookup}$ returns $\text{xnode}$ of the entry since $l_i \ldots l_n$ is the longest suffix of $l_1 \ldots l_n$ among required paths (Line (11)). Otherwise (i.e., $\text{next}$ is not NULL), the node pointed to by $\text{next}$ is retrieved since $l_i \ldots l_n$ may not be the longest suffix of $l_1 \ldots l_n$ among required paths (Line (12)).

Consider that the update module checks the validity of a node’s subnode in $G_{APEX}$ using $\text{lookup}$. If the $\text{xnode}$ field of the entry returned by $\text{lookup}$ actually points to the subnode, we do not need to do anything on $H_{APEX}$, since $H_{APEX}$ has the right information. However, if both $\text{next}$ and $\text{xnode}$ fields of the entry are NULLs, there are two cases. One is when the path represented by the entry in $H_{APEX}$ is a newly generated frequently used path due to the change of the query workload. The other is when the entry in the node $n$ pointed to a node $m$ in $H_{APEX}$ previously, but the node $m$ was deleted because the label paths represented by entries in the node $m$ are not frequent any more. In this case, both $\text{next}$ and $\text{xnode}$ fields of the entry in the node $n$ were set to NULL. The $\text{xnode}$ field of the entry returned by $\text{lookup}$ needs to be replaced to point to the corresponding node in $G_{APEX}$ after the modification of $G_{APEX}$. We generate a new node in $G_{APEX}$ that is reachable with the same label of the edge from the parent node of the subnode and set the entry’s $\text{xnode}$ field to point the node.
generated. The extent for the newly generated node in $G_{APEX}$ is computed by examining $G_{XML}$ and attached to the new node. We recursively visit the newly generated node with the extent. We still keep the subnode because it may be visited again through a different path later during the update phase.

For understandability, we explain the basic behavior of the dynamic update of APEX. Suppose we visit a node $x$ in $G_{APEX}$ with an label path $path_1$ and a edge set $\Delta ESet$ as illustrated in Figure 10. $\Delta ESet$ is a newly added edges to the extent of $x$ just before visiting $x$. $\Delta ESet$ is needed to update the subgraph of $x$.

![Fig. 10. Visiting a node by UpdateAPEX](image)

If $x$ was previously visited and $\Delta ESet$ is empty, then we do nothing since all edges and their subgraphs of $x$ were traversed before.

If $x$ is newly visited and $\Delta ESet$ is empty, we should traverse all outgoing edges of $x$ in $G_{APEX}$ to verify all subnodes of $x$ using $H_{APEX}$. For example, while $x$ has an outgoing edge to $y$ with label $l_1$, we verify the node $y$ using the $lookup$ function with the label path $path_1, l_1$

Whether $x$ is previously visited or not, if there is a change of the extent of it, we should update the subgraph rooted at $x$ according to the change of the extent (i.e., $\Delta Eset$).

We present the $updateAPEX$ in Figure 11 that does the modification of APEX with frequently used paths stored in $H_{APEX}$. Before calling the $updateAPEX$, we first initialize visited flags of all nodes in $G_{APEX}$ to FALSE. The $updateAPEX$ is executed with the root node in $G_{APEX}$ by calling $updateAPEX(xroot, \emptyset, NULL)$ where $xroot$ is the root node of $G_{APEX}$.

If $xnode$ was previously visited and $\Delta ESet$ is empty, then we do nothing (Line (1)). If $xnode$ is newly visited and $\Delta ESet$ is empty, we should traverse all outgoing edges of $xnode$ in $G_{APEX}$ to verify all subnodes of $xnode$ using $H_{APEX}$. For each ending vertex (i.e. $e.end$) of $xnode$ in $G_{APEX}$, we get the pointer $xchild$ by calling the $lookup$ function with the input the label path from the root to $e.end$ (Line (6)-(7)). If the value of $xchild$ is NULL, it means that the valid node in $G_{APEX}$ does not exist. Thus, we allocate a new node of $G_{APEX}$ and have $xchild$ point to the new node (Line (8)). If $xchild$ is not NULL, the $xnode$ field returned by $lookup$ points to a node in $G_{APEX}$ for the given label path.
Procedure updateAPEX(xnode, ΔESet, path)
begin
1. if (xnode.visited = TRUE) and (ΔESet = ∅) return
2. xnode.visited := TRUE
3. EdgeSet := ∅
4. if ΔESet = ∅ {
5. for each e that is an outgoing edge of xnode do {
6. newpath := concatenate(path, e.label)
7. xchild := lookup(newpath)
8. if (xchild = NULL) xchild := newXNode()
9. if xchild != e.end {
10. if EdgeSet = ∅ {
11. for each < u, v > ∈ xnode.extent do
12. EdgeSet := EdgeSet ∪ { o | o is an outgoing edge from v }
13. }
14. subEdgeSet := a set of edges with the label e.label in EdgeSet
15. ΔEdgeSet := subEdgeset - xchild.extent
16. xchild.extent := xchild.extent ∪ ΔEdgeSet
17. make_edge(xnode, xchild, e.label);
18. hash.append(newpath, xchild);
19. }
20. } else ΔEdgeSet := ∅
21. updateAPEX(xchild, ΔEdgeSet, newpath);
22. }
23. } else {
24. for each < u, v > ∈ ΔESet do
25. EdgeSet := EdgeSet ∪ { o | o is an outgoing edge from v }
26. for each unique label l in EdgeSet do {
27. newpath := concatenate(path, e.label)
28. xchild := lookup(newpath)
29. if (xchild = NULL) xchild := newXNode()
30. subEdgeSet := set of edges labeled l in EdgeSet
31. ΔEdgeSet := subEdgeset - xchild.extent
32. xchild.extent := xchild.extent ∪ ΔEdgeSet
33. make_edge(xnode, xchild, l)
34. hash.append(newpath, xchild)
35. updateAPEX(xchild, ΔEdgeSet, newpath);
36. }
37. }
end

Fig. 11. An algorithm to update APEX

If xchild and e.end are different, we compute the edge set which should be added to the extent of xchild (Line (10)-(14)). We select the edges, from computed edges, which are not already in the extent of xchild, and assign them to ΔEdgeSet. We then update the extent of xchild by adding edges in ΔEdgeSet (Line (15)-(16)). We next make an edge by invoking make_edge from
$xnode$ to $xchild$ with label e.label, if the edge does not exist (Line (17)-(18)). If $xnode$ has an outgoing edge to a node in $G_{APEX}$ which is different from $xchild$ with label e.label, make_edge removes this edge. If $xchild$ and e.end are equal, there is no change of the extent of the node $xchild$ (Line (20)). Thus, $\Delta$ESet is set to empty. Now, we call updateAPEX recursively for $xchild$ which is a child node of $xnode$.

Whether $xnode$ is previously visited or not, if there is a change of the extent of it, we should update the subgraph rooted at $xnode$ (Line (23)-(37)). In this case, we obtain outgoing edges from the end points of edges in $\Delta$ESet (Line (24)-(25)). In order to update the subgraph rooted at $xnode$, we partition the edges based on their labels and update $H_{APEX}$ and $G_{APEX}$ similarly as we processed for the case when $\Delta$ESet was empty. (Line (26)-(36)).

Let us consider the $H_{APEX}$, $G_{APEX}$ and $G_{XML}$ in Figure 12-(a) and Figure 12-(b). Figure 12 is a continuation of the example depicted in Figure 7. It shows the change of $G_{APEX}$ in Figure 12-(b) due to the changed state of $H_{APEX}$ as in Figure 12-(b), which is same as $H_{APEX}$ in Figure 7-(c). We invoke updateAPEX ($xroot$, $\emptyset$, NULL). Since the $\Delta$ESet is empty, the code in Line (4)-(23) in Figure 11 will be executed. Since there is only one outgoing edge with the label A and the ending node &1, we check the entry in $H_{APEX}$ for the path of A. The $xnode$ field of the entry returned by lookup points to the node &1. Thus, we do nothing and call updateAPEX(&1, $\emptyset$, A) recursively. This recursive call visits the node &1 with label path A. We have three outgoing edges from the node &1. Suppose we consider the node &2 first in the for-loop in line (5). We check the entry in $H_{APEX}$ with the path of A.B and find that the $xnode$ field of the entry points to the node &2. Thus, we do nothing again and invoke updateAPEX(&2, $\emptyset$, A.B) recursively. Inside this call, we check outgoing edges of &2. As illustrated in Figure 12-(b), the end node of an outgoing edge of &2 with label D is &4. However, for the input of A.B.D, lookup returns NULL that is the $xnode$ field of the entry for remainder.D, which should point to the node in $G_{APEX}$ representing all label paths ending with D except A.D. Thus, we make a new node &6 for remainder.D, compute extent of &6 and change the outgoing edge of &2 with label D to point to &6. Since there is no outgoing edge, we return back to &1 from the recursive call. We next consider the outgoing edge with the end node of &5 with the label path A.D from &1. The $H_{APEX}$ and $G_{APEX}$ including the extents after traversing every node in $G_{APEX}$ with updateAPEX are shown in Figure 12-(d).

6 Experiment Results

We empirically compared the performance of APEX with the strong DataGuide and the Index Fabric on real-life and synthetic data sets. In our experiments,
we found that APEX shows significantly better performance. In addition, in a number of cases, it is more than an order of magnitude faster than the other indexes.
6.1 Experimental Environment

The experiments were performed on Pentium III-866MHz platform with MS-Windows 2000 and 512 MBytes of main memory. The XML4J parser\(^4\) and the XML Generator\(^5\) from IBM were used to parse and to generate XML data. We implemented the strong DataGuide, the Index Fabric, and APEX in Java. The data sets were stored on a local disk. We begin by describing the XML data sets and query workloads used in the experiment.

**Data Sets.** To evaluate APEX over various structured XML data, we used real-life and synthetic data sets.

Real-life data sets come from the collection of the plays of Shakespeare [10]. To test the efficiency of APEX with regard to various data sizes, we combined some plays of Shakespeare into bigger ones: four tragedies.xml consists of four tragedies (Hamlet, Macbeth, Othello, and Lear), shakes\_11.xml consists of eleven plays, and shakes\_all.xml is an XML file containing all Shakespears's plays. Because the Shakespeare’s play data sets do not have ID and IDREF typed attributes, these are tree structured XML data.

Synthetic data sets are generated using the XML Generator from IBM with real-life DTDs: FlixML and GedML. The Flix Markup Language (FlixML) is a markup language for categorizing B-movie reviews for the XML-based B-movie guides\(^6\). The GedML is a markup language for the genealogical XML data [10]. These two kinds of synthetic data sets are graph structured. The Shakespeare’s play data sets show minor irregularities in the structure. The FlixML data sets (Flix01.xml, Flix02.xml, and Flix03.xml) have moderate irregularities and the structures of GedML data sets (Ged01.xml, Ged02.xml, and Ged03.xml) are highly irregular.

The other characteristics of the three kinds of data sets used in the experiment are summarized in Table 1. The two numbers in the last column of the table represent the number of distinct labels and IDREF typed labels (inside of parentheses), respectively.

**Query Workloads.** To estimate the efficiency of APEX for query processing, we generated 5000 XML queries randomly. A simple path expression is a sequence of labels starting from the root of the XML data. There can be a dereference operator (=>) with an attribute in the simple path expression due to the IDREF type attribute in XML data. In order to generate XML queries,

\(^4\) available at http://www.alphaworks.ibm.tech/xml4j

\(^5\) available at http://www.alphaworks.ibm.tech/xmlgenerator

\(^6\) available at http://www.xml.com
we stored all possible simple path expressions in XML data. To generate a query, we randomly selected a simple path expression, selected a subsequence of the simple expression randomly, and then put the self-or-descendent axis in front of the subsequence. We repeated this process until 5000 queries are generated. The queries generated can be thought of as XQuery queries [4] having formats of either //l_i//l_{i+1}/...//l_n or //l_i//...//l_j => l_{j+1}/...//l_n where \( l_j \) is a tag or an attribute (with the prefix '@'). We randomly selected 20% of the 5000 queries as the query workload. We found that the percentage of simple path expressions in the query workload generated by the above method was about 25%. We will represent the type of these queries as \( QTYPE1 \).

To evaluate more complicated partial matching queries, we also generated 500 XML queries having formats of //\( l_i//l_j \) on each data set. To make this kind of queries, we randomly selected a simple path expression and choose two distinct labels from the simple path. We will represent the type of these queries as \( QTYPE2 \). On the evaluation of the \( QTYPE2 \) queries, the query processor does not use the reference relationship (i.e., \( l_k => l_{k+1} \)). However, there may be only one label path, which contains the reference relationship, from \( l_i \) to \( l_j \). In this case, the result of such a query becomes empty. Thus, we did not make sure that the results of the queries are not empty.

As mentioned earlier, the Index Fabric does not keep the information of XML elements which do not have data values. Thus, the Index Fabric is not effective to process queries of \( QTYPE1 \) and \( QTYPE2 \). To compare the query performance of APEX with that of the Index Fabric, we generated 1000 XML queries represented as \( QTYPE3 \). This typed queries retrieve the XML elements which have the data value. The XQuery format for the \( QTYPE3 \) queries is //\( l_i//l_{i+1}/...//l_n[text()] = value \). Since, as shown in [9], the Index Fabric does

\[
\begin{array}{|c|c|c|c|}
\hline
\text{Data Set} & \text{nodes} & \text{edges} & \text{lables} \\
\hline
\text{four\_tragedy.xml} & 22791 & 22790 & 17(0) \\
\text{shakes\_l1.xml} & 48818 & 48817 & 21(0) \\
\text{shakes\_all.xml} & 179691 & 179690 & 22(0) \\
\text{Flix01.xml} & 14734 & 14763 & 62(3) \\
\text{Flix02.xml} & 41691 & 41723 & 64(3) \\
\text{Flix03.xml} & 335401 & 335432 & 70(3) \\
\text{Ged01.xml} & 8259 & 9699 & 65(14) \\
\text{Ged02.xml} & 30875 & 36228 & 77(14) \\
\text{Ged03.xml} & 381046 & 447524 & 84(14) \\
\hline
\end{array}
\]
not keep the dereference relationship information, the QTYPE3 queries do not contain the dereference operator. We also made sure that the results of the queries are not empty.

**Query Processor Implementation.** While evaluating a query //l_i/l_{i+1}/ \ldots /l_n of QTYPE1, if the longest suffix found from H_{APEX} is equal to the label path expression l_i.l_{i+1} \ldots l_n of the given query, the query processor gathers the ending nodes from the extents of nodes in G_{APEX} which can be located using H_{APEX}. These ending nodes are the result of the given query. Otherwise, the query processor repeatedly looks up H_{APEX} using l_i.l_{i+1} \ldots l_j by decreasing the label subscript j from n to i until the longest suffix found from H_{APEX} is equal to the label path l_i.l_{i+1} \ldots l_j. For each look-up of H_{APEX} (for each j), the query processor keeps the union of extents of nodes in G_{APEX} which can be located using H_{APEX}. Finally, to obtain the query result, the query processor performs a multi-way join operation on the edge sets in the extents acquired by look-ups.

To evaluate a query //l_i/l_j of QTYPE2, the query pruning and rewriting technique [12] is applied. This partial matching query is rewritten as the set of label paths with the format of l_i.l_{i+1} \ldots l_j in APEX. However, in the strong DataGuide, these partial matching queries should be rewritten as a set of simple path expressions in the form of l_1 \ldots l_i.l_{i+1} \ldots l_j.

Query processing of the QTYPE3 queries in the strong DataGuide and APEX consists two steps. In the first step, the nodes which satisfy the label path expression in the query are acquired using the index. The behavior of the query processor in this step is the same as that for QTYPE1. In the second step, the query processor tests the nodes by looking up the data table which keeps all node identifiers (nid) and corresponding data values. In the Index Fabric, the query is encoded as the search key string and the query processor performs the key search. Since the Index Fabric keeps all simple paths like the strong DataGuide, the whole index structure should be traversed to perform the partial matching path expression. The performance of the Index Fabric is affected by the size of the index block. In our experiment, we set the index block size of the Index Fabric to 8Kbytes.

6.2 Performance Result

In order to get the feeling about the structures generated by the strong DataGuide and APEX, we present the statistics regarding indexes in Table 2.

For APEX, we varied minSup between 0.002 and 0.05. Typically, the strong DataGuide produces more complex structures than APEX variants. It is not surprising since the strong DataGuide keeps all the possible paths from the
Table 2
Statistics of Index Structures.

<table>
<thead>
<tr>
<th>Structure</th>
<th>SDG</th>
<th>APEX(^0)</th>
<th>0.002</th>
<th>0.005</th>
<th>0.01</th>
<th>0.03</th>
<th>0.05</th>
</tr>
</thead>
<tbody>
<tr>
<td>four_tradegy.xml</td>
<td>Nodes</td>
<td>24</td>
<td>18</td>
<td>24</td>
<td>24</td>
<td>22</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>Edges</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>shakes_11.xml</td>
<td>Nodes</td>
<td>43</td>
<td>22</td>
<td>43</td>
<td>43</td>
<td>25</td>
<td>23</td>
</tr>
<tr>
<td></td>
<td>Edges</td>
<td>42</td>
<td>36</td>
<td>42</td>
<td>42</td>
<td>42</td>
<td>42</td>
</tr>
<tr>
<td>shakes_all.xml</td>
<td>Nodes</td>
<td>59</td>
<td>23</td>
<td>59</td>
<td>59</td>
<td>49</td>
<td>30</td>
</tr>
<tr>
<td></td>
<td>Edges</td>
<td>58</td>
<td>40</td>
<td>58</td>
<td>58</td>
<td>58</td>
<td>56</td>
</tr>
<tr>
<td>four_tradegy.xml</td>
<td>Nodes</td>
<td>163</td>
<td>65</td>
<td>159</td>
<td>126</td>
<td>79</td>
<td>66</td>
</tr>
<tr>
<td></td>
<td>Edges</td>
<td>162</td>
<td>127</td>
<td>162</td>
<td>161</td>
<td>153</td>
<td>142</td>
</tr>
<tr>
<td>shakes_11.xml</td>
<td>Nodes</td>
<td>138</td>
<td>65</td>
<td>137</td>
<td>119</td>
<td>84</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>Edges</td>
<td>138</td>
<td>117</td>
<td>138</td>
<td>137</td>
<td>132</td>
<td>131</td>
</tr>
<tr>
<td>four_tradegy.xml</td>
<td>Nodes</td>
<td>176</td>
<td>71</td>
<td>172</td>
<td>123</td>
<td>83</td>
<td>74</td>
</tr>
<tr>
<td></td>
<td>Edges</td>
<td>175</td>
<td>142</td>
<td>175</td>
<td>170</td>
<td>167</td>
<td>167</td>
</tr>
<tr>
<td>shakes_all.xml</td>
<td>Nodes</td>
<td>3976</td>
<td>66</td>
<td>880</td>
<td>428</td>
<td>243</td>
<td>121</td>
</tr>
<tr>
<td></td>
<td>Edges</td>
<td>4587</td>
<td>178</td>
<td>2956</td>
<td>1692</td>
<td>1032</td>
<td>518</td>
</tr>
<tr>
<td>four_tradegy.xml</td>
<td>Nodes</td>
<td>13092</td>
<td>78</td>
<td>1500</td>
<td>652</td>
<td>363</td>
<td>182</td>
</tr>
<tr>
<td></td>
<td>Edges</td>
<td>16105</td>
<td>221</td>
<td>6193</td>
<td>3307</td>
<td>1648</td>
<td>793</td>
</tr>
<tr>
<td>shakes_all.xml</td>
<td>Nodes</td>
<td>114679</td>
<td>85</td>
<td>2106</td>
<td>762</td>
<td>441</td>
<td>216</td>
</tr>
<tr>
<td></td>
<td>Edges</td>
<td>133071</td>
<td>255</td>
<td>10395</td>
<td>3961</td>
<td>2250</td>
<td>996</td>
</tr>
</tbody>
</table>

root of XML data. Particularly, the size of the strong DataGuide for the highly irregularly structured data (GedML) becomes very large. As expected from the definition of APEX\(^0\), it has the most compact structure. When we increase the value of minSup, the number of frequently used paths decreases. In the query workload we generated, when the value of minSup is at least 0.05, the length of almost every required path becomes one. Thus, the structure of APEX in this case becomes very close to that of the APEX\(^0\).

We also plot the total query processing costs of the QTYPE1 queries for the three kinds of data sets over the strong DataGuide and APEX as minSup varied from 0.002 to 0.05 in Figure 13. We also show the cost of APEX\(^0\) because it represents the upper bound of the cost as we increase minSup. Note that the required path set becomes a set of paths with the length of one when we increase minSup to high values close to 0.05. Obviously, the query processing cost of APEX\(^0\) should be the most slow among APEX variants since we need to perform joins of extents for the path expression of the query with the length of at least two. Therefore, the query processing pays severe performance penalty and it was illustrated by the graphs for all experimental results.

Note that the query processing of the strong DataGuide is more inefficient than that of APEX\(^0\) for the moderately and highly irregularly structured data (FlixML and GedML). The query processing times of APEX\(^0\) for Ged02.xml
Fig. 13. Total execution times of $Q_{TYPE1}$
and Ged03.xml are about 1500 seconds and 25000 seconds, respectively. Whereas, those of the strong DataGuide are about 27000 seconds and 125000 seconds, respectively. The inefficiency of the strong DataGuide comes from the fact that its size generated for the XML document with a complex structure becomes very large. As Table 2 illustrates, the number of nodes and the number of edges of the strong DataGuide generated for Ged02.xml data are 13392 and 16105 respectively, while these of APEX$^0$ are 78 and 221. Furthermore, to process a partial matching path query, the query processor should traverse a lot of nodes in the strong DataGuide. This result confirms that the strong DataGuide is generally inefficient for complex XML data. In contrast to the strong DataGuide, in APEX$^0$, the results of some queries whose lengths are one can be directly obtained by $H_{APEX}$.

The performance of APEX is affected by the value of minSup. As minSup decreases, the number of frequently used paths increases. Thus, the results of more queries can be directly obtained by $H_{APEX}$. However, as minSup decreases, the query processing time does not always decrease. As shown in Figure 13-(b) and (c), when minSup is extremely small (i.e., 0.002), APEX does not show the best performance of APEX variants since APEX with small minSup keeps too many frequently used paths and there are overheads for the extent union operations to compute the query result.

As shown in Table 2, the sizes of the graph structures of the strong DataGuide and APEX with minSup of 0.002 and 0.005 for the moderately and less irregularly structured data (Play and FlixML) are similar. However, the query processing cost of APEX is cheaper than that of the strong DataGuide. This is because the query result can be directly obtained by $H_{APEX}$ without traversing $G_{APEX}$ of APEX. In addition, the query performance of APEX is significantly better for the highly irregularly structured XML data (GedML) than that of the strong DataGuide. From the result, we can conclude that APEX shows much better performance as the structure of the XML data gets more complex.

Figure 14 describes the total query execution time of the $QTYPE2$ queries over the strong DataGuide, APEX$^0$, and APEX with minSup of 0.005. Note that the Y-axis is in log-scale. We have chosen the value of 0.005 because it is one of values of minSup that generates an efficient APEX. We omit the evaluation results for the other data sets (e.g., four_tragedy.xml, Flix01.xml) since they show similar results.

To evaluate a query of $QTYPE2$, the query pruning and rewriting should be applied. To perform the query pruning and rewriting with the strong DataGuide, the query processor generally traverses the whole index structure from the root several times. However, the query processor for APEX traverses the index structure of $G_{APEX}$ starting from the nodes whose incoming edges’
labels are $l_i$. Therefore, the query pruning and rewriting overhead of APEX is less than that of the strong DataGuide. Even though there is the join overhead to obtain the query result, APEX shows the best performance over the various data sets. As shown in Table 2, the graph structure of APEX$^0$ is the most compact. Thus, the query pruning and rewriting on the APEX$^0$ takes less time than that of the others. However, the query processing cost of APEX$^0$ is high since there are massive join operations to obtain the query result. The $QTYPE2$ processing time of APEX$^0$ for Ged02.xml is up to 210000 seconds.

![Diagram](image_url)

**Fig. 14.** Total evaluation times of $QTYPE2$ [log scale]

Figure 15 describes the total execution time of the $QTYPE3$ queries over the Index Fabric, the strong DataGuide, and APEX with minSup of 0.005. We also omit the results of the other data sets. As mentioned in Section 2, the Index Fabric records each element which has a data value using the compositions of the encoded label path and the data value of this element. Thus, in the Index Fabric, the result of a query of $QTYPE3$ is obtained using only the index structure. On the contrary, in the strong DataGuide and APEX, the data nodes obtained by the index are tested whether the nodes have the given data value by looking up the data table. Thus, the Index Fabric shows the

![Diagram](image_url)

**Fig. 15.** Total evaluation times of $QTYPE3$ [log scale]
best performance for less irregularly structured data.

In contrast to the result for the minor irregular structure, the Index Fabric shows the worst performance for the moderately and highly irregularly structured data (FlixML and GedML). This result comes from the lossy compression inherent in the Patricia Trie. The Index Fabric keeps a composition of the encoded label path and the data value of an element as a string using the Patricia Trie. Thus, to evaluate a QTYPE3 query using the Index Fabric, the traversal of the whole index structure and the validation of each node with regard to the given label path expression are needed. This overhead degrades the performance of the Index Fabric. Whereas, due to the nice properties described above, APEX shows the best performance for complex XML data.

The effectiveness of APEX is affected by \textit{minSup}. In our experiment, APEX is efficient with the value of \textit{minSup} ranging between 0.002 and 0.01. When \textit{minSup} is 0.005, APEX shows the best performance with the query workload we generated over various XML data.

7 Conclusion

In this paper, we propose APEX which is an Adaptive Path indEx for XML data. APEX does not keep all paths starting from the root and utilizes frequently used paths to improve the query performance. In contrast to the traditional indexes such as DataGuides, 1-indexes and the Index Fabric, it is constructed by utilizing a data mining technique to summarize paths that appear frequently in query workloads. APEX can be incrementally updated in order to minimize the overhead of construction whenever the query workload changes. APEX also guarantees all paths of length two so that any label path expression can be evaluated by joins of extents in APEX without scanning original data.

To support efficient query processing, APEX consists of two structures: the graph structure \(G_{APEX}\) and the hash tree \(H_{APEX}\). \(G_{APEX}\) represents the structural summary of XML data with extents. \(H_{APEX}\) keeps the information for frequently used paths and points to the nodes in \(G_{APEX}\) which correspond to the ending nodes of the frequently used paths. Given a query, we use \(H_{APEX}\) to locate the nodes of \(G_{APEX}\) that have extents required to evaluate a query.

We implemented APEX and conducted an extensive experimental study with both real-life and synthetic data sets. Experimental results show that APEX improves query processing cost typically 2 to 69 times compared with the traditional indexes, with the performance gap increasing with the irregularity of XML data.
References


