**RCdex**: An Index Structure for Tree Matching in the Presence of Commutations

K. Selçuk Candan  
Jaikannan Ramamoorthy  
Mehmet E. Dönderler  
Computer Science and Engineering Dept.  
Arizona State University  
Tempe, AZ, 85287, USA.  
{candan,mehmet.donderler, jaikannan.ramamoorthy}@asu.edu

**ABSTRACT**

Tree-structured data models and formats are increasingly being used in many applications to represent and store data objects. Since the structural information inherent in such data cannot be captured in conventional index structures, quick retrieval of tree-structured data necessitates special indexing techniques. A variety of algorithms, such as ViST and Data Guides, have been proposed to answer tree queries efficiently. Nonetheless, a class of tree-structured data, which are insensitive to certain commutations between the elements, cannot be indexed using the existing techniques. In this paper, we propose an index structure (RCdex) to efficiently answer wildcard (“*”) queries in the presence of commutations. We experimentally show that this approach provides large savings in query execution time when commutations between the nodes of the tree are allowed.

1. **INTRODUCTION**

Tree-structured representation of data objects has become a vital component of many applications. Hierarchical relationships within a given object constitute significant information that has to be preserved and used in retrieval, but conventional index structures are not suitable for this.

Extensible Markup Language (XML) [1] enables the description of such tree-like hierarchical structures. Object structure may be defined through a Document Type Definition (DTD) or through an XML schema. DOM [2] and LORE [3] are two well-known tree-based data models for XML documents and objects. When an XML object is represented in its tree form, each node corresponds to an element (or an attribute of an element) in the XML object. A child node corresponds to a sub-element or an attribute of the parent node. For each node, there is a tag that indicates its name.

XML objects can be queried using XML query languages, such as XQuery [4], XML-QL [5], and XPath [6]. These languages retrieve data using path expressions, which are essentially paths followed from the root of the XML tree to the leaves while satisfying the retrieval conditions of the query. This way not only the content descriptors, but also the structure of the XML object is used in retrieval.

Since XML data is hierarchical in nature, we cannot use conventional indexing techniques for making retrieval efficient. A variety of indexing techniques, such as ViST [7] and Data Guides [8], have been proposed to answer XQuery queries efficiently. Nevertheless, a class of tree-structured data, which are insensitive to certain commutations between elements, cannot be indexed using these techniques:

- **Application I: Schedule/Plan Databases**: The need for storing and retrieving schedules/plans in an efficient manner is apparent, especially for large companies with automated manufacturing processes, which may want to take advantage of previously generated work plans. Many plans (for instance, query execution plans in databases [9]) have hierarchical/tree-like structures. In many cases, the order of various actions in plans, however, may change (i.e., actions may commute) without affecting the result; i.e., plan data may be commutation insensitive. For example, in relational algebra, two selection operations may commute yielding different query processing plans with the same results. Similarly, other algebras, such as the ontology-composition algebra in [10], allows for commutativity.

- **Application II: Multimedia Databases**: Many multimedia standards define objects as a structured collection of media objects. For example, X3D [11] is an XML based language for describing 3D worlds. X3D nodes are expressed as XML elements, and each node has its own attributes. In general, the same 3D object can be described using different X3D structures. This is because some X3D elements, such as spatial transformations, are commutation insensitive. An example 3D world and its description with two different (but commutation-equivalent) X3D structures are presented in Figure 1.

In this paper, we propose path and tree matching and indexing techniques to efficiently answer queries in the presence of such commutations.

![Figure 1: (a) An example 3D world and (b, c) alternative descriptions](image-url)
1.1 Contributions of this Paper

We see that in general there can be two types of commutativity rules:

- the elements of the tree can commute; or
- the order of the children of a given node can change.

In this paper, we restrict our discussion to the first type of commutativity rules. This problem can be formally stated as follows: Given a given pattern tree (query tree) in a given target tree (tree structured data) collection. We introduce a novel two-level indexing mechanism to enable processing of tree-structured data in the presence of commutations. We show that using properly chosen canonical paths, it is possible to cluster and retrieve tree-structured data efficiently in the presence of commutations.

The organization of this paper is as follows: in Section 2, we define the problem of tree matching and retrieval in the presence of commutations; in Section 3, we provide a short overview of the existing tree indexing techniques and their shortcomings; in Section 4, we discuss a special case of the commutative matching problem, where paths are matched in the presence of commutations. We show that using properly chosen canonical paths, it is possible to cluster and retrieve paths efficiently. In Section 5, we introduce canonical path segments and show how to use these to enable retrieval of tree-structured data in the presence of commutations of node labels. In Section 6, we present a novel two-level indexing mechanism to enable processing of wildcard queries, where some elements in the path query are replaced with wildcard ("*"), labels. The result is the RCdexpath structure for indexing paths, which forms the basis for the RCdex index structure. In Section 7, we experimentally show that the proposed indexed structure works efficiently. We give the related work on tree matching and draw our conclusions in Sections 8 and 9, respectively.

2. PROBLEM STATEMENT

The problem we address in this paper is to efficiently retrieve tree-structured data in the presence of commutativity rules. This problem can be formally stated as follows: Given

- an alphabet, \( \mathcal{A} \), of labels,
- a set \( \mathcal{T} \) of tree structured objects, where each \( t \in \mathcal{T} \) is an ordered, directed, labeled tree \( t = (V_t, E_t, l_t) \), where
  - \( V_t \) are the nodes,
  - \( E_t \) are the edges,
- a set \( \mathcal{C} \) of commutativity rules of the form
  - \( c_i \equiv a \leftrightarrow b \), where \( a, b \in \mathcal{A} \), which denotes that nodes labeled “a” and “b” may be swapped.
- a query tree, \( Q \), and a query tree, \( q \) = \((V_q, E_q, l_q)\), where
  - \( l_q : V_q \rightarrow \mathcal{A} \cup \{\ast\} \) is a mapping from the nodes of the query tree to an alphabet, \( \mathcal{A} \cup \{\ast\} \) (here “\( \ast \)” is a wildcard character that matches any character in \( \mathcal{A} \)),
- \( A \) = \{A, B, C, D\}
- \( c = \{A \leftrightarrow C, \ D \leftrightarrow C\} \)

Figure 2: An example: (a) the alphabet and commutativity rules, (b) query tree, and (c,d) two matching documents, \( t_1 \) and \( t_2 \).

find the set \( \mathcal{R} \subseteq \mathcal{T} \) of trees that match query \( q \).

Tree matching is concerned with finding the instances of a given pattern tree (query tree) in a given target tree (tree structured data) collection. Formally, the problem of tree matching can be defined as follows: given two ordered, directed, labeled trees \( Q \) and \( T \), the query tree \( Q \) matches the target tree \( T \) at a node \( x \) iff there exists a one-to-one mapping \( \mu \) from the nodes of \( Q \) to the nodes of \( T \) (with respect to the node \( x \)) which preserves the labels and the ancestor/descendant relationships of the nodes. Figure 2 provides an example query, applicable commutativity rules, and two example objects matching the query based on the commutativity rules.

3. BACKGROUND: STRUCTURE ENCODING FOR TREE MATCHING

Providing efficient support for tree-based queries with wildcards poses a challenging task. The related work section (Section 8) of this paper provides an overview of the existing approaches and solutions.

Most of the approaches proposed in the literature attempt to answer tree queries by relying on expensive structural join operations: queries are decomposed into branch subqueries and the results from multiple path queries are joined to obtain the answer to the original query. A recent index structure, ViST [7], on the other hand, relies on the observation that querying XML data by structure is equivalent to finding subsequence matches on a structure-encoded representation of the tree data. ViST represents both XML documents and XML queries (including those with branches, or wildcards, \\("*/") in structure-encoded sequences, provides a unified index on both content and structure, and avoids expensive explicit join operations.

In ViST, both XML documents and queries are represented using the preorder sequence of their tree structures. A structure-encoded sequence, derived from a preorder traversal of an XML document or a query, is a sequence of (symbol, prefix) pairs:

\[
D = (a_1;p_1)(a_2;p_2)\ldots(a_n;p_n)
\]

where \( a_i \) represents a node in the XML document tree and \( p_i \) is the path from the root node to node \( a_i \). For example, the tree in Figure 2(c) would be encoded as

\[
t_1 = (C, e)(A; C)(B; C A)(C; C A B)(D; C A B C)(C; C A B)
\]

whereas the query (Figure 2(b)) would be encoded as

\[
q = (A, e)(\ast; A)(B; A \ast)(D; A \ast B)(C; A \ast B D)(C; A \ast B)
\]
For further efficiency, ViST uses a B+ tree to index nodes in the suffix tree by their (Symbol, Prefix) pairs. This first index (D-Index) enables retrieval of all nodes in the suffix tree by their (Symbol, Prefix) pairs. This third B+ tree stores XML document IDs with respect to their corresponding suffix tree nodes, where the suffix tree node id is used as the key for indexing. Queries with wildcard characters, ‘*’ and ‘/’, are handled using range searches on the B+ trees.

3.1 Limitations

However, existing structure encoding techniques have a number of limitations that prevent their use when there is commutative flexibility in the data. As commutations of the elements of the tree would destroy the encoding of the strings that are indexed, ViST is not suitable for indexing when commutations between elements are allowed. For example, although we would like the document

\[ t_1 = (C, c)(A; C)(B; CA)(C; CAB)(D; CABC)(C; C) \]

in Figure 2(c) to be returned when the query

\[ q = (A, c)(A; A)(B; A)(D; A B)(C; A B D)(A; A B) \]

in Figure 2(b) is posed subject to the commutativity rules

\[ C = \{ A \rightarrow C, D \rightarrow C, B \rightarrow C \} \]

listed in Figure 2(a), it is easy to see that a straightforward matching does not hold between the sequences \( t_1 \) and \( q \).

3.2 Observations

Despite these limitations, the structure-encoding approach highlights the importance of two issues for efficient tree matching:

- The first issue is indexing of the substrings of the structure-encoded sequences using a suffix tree (or after suitable numbering of the nodes, using a B+ tree (S-Index in Figure 3) which captures the ranges of suffix tree nodes that satisfy the ancestor/descendant relationships based on structure-encoded sequences).
- The second issue is efficient addressing of path segments. In non-commutative data, this can be achieved using various index structures, such as compact tries, hash tables, or B+ trees (D-Index in Figure 3). Therefore, in this paper, we develop a novel index structure (\( \text{RCdex} \)) for indexing paths in the presence of commutations and wildcard labels. We show how to use this index structure to index trees for efficient retrieval of trees.

ViST uses a dynamic labeling method for labeling suffix tree nodes, and hence, the suffix tree itself is not materialized.

4. PATH MATCHING IN THE PRESENCE OF COMMUTATIONS

In this section, we discuss how to match paths in the presence of commutations. We highlight key observations that enable matching of two paths that are equivalent under a given set of commutativity rules. In Section 5, based on the matching techniques introduced here, we will develop a tree matching technique to efficiently index trees.

4.1 Canonical Paths

The first construct we introduce to deal with the existence of commutations is the canonical path. We define the canonical path as follows:

**Definition 4.1 (Canonical Path).** Let

- \( A \) be an alphabet;
- \( p = v_0 \rightarrow v_1 \rightarrow \ldots \rightarrow v_n \) be a path;
- \( l : \{v_0, \ldots, v_n\} \rightarrow A \) be its node labeling;
- \( \text{rank} : A \rightarrow Z^+ \) be a ranking function corresponding to a total (such as lexicographic) order of the elements of the alphabet \( A \);
- \( \mathcal{P}(p, C) \) is the set of all permutations of path \( p \) allowed by the set \( C \) of commutativity rules;
- \( \text{order} : \mathcal{P}(p, C) \times \{v_0, \ldots, v_n\} \rightarrow Z^+ \) be a function which returns the position of a given vertex \( v_i \) in a given permutation of \( p \).

Then, the canonical form \( p^c \) of the path \( p \) is a permutation of \( p \) in \( \mathcal{P}(p, C) \) (i.e., \( p^c \in \mathcal{P}(p, C) \)), such that

\[ \forall p^c \in \mathcal{P}(p, C) \text{prop}(p^c, p) = \text{true} \]

The proposition \( \text{prop}(p^c, p) \) is defined as...
The legal permutations of $p$ is $O(n^2)$ in-memory operations. Canonical form transformation can be achieved step by step by finding the first, the second, ..., and the last node in $p'$ applying the commutativity rules $C$ on $p$. This procedure is demonstrated as follows for the path $p_2$ of Example 4.2 above:

- Initialize $p'_\text{temp}$ to $p_2$.
  $$p'_\text{temp} = p_2 = C \rightarrow B \rightarrow A$$
- Iteration 1: Identify the first leftmost node (position 1) of $p'_\text{temp}$.
  - The potential leftmost commutation position of node $A$ is 2.
    $$C \rightarrow B \rightarrow A \Rightarrow A \rightarrow B \quad C \rightarrow A \rightarrow B$$
  - The potential position of $B$ is 1.
    $$C \rightarrow B \rightarrow A \Rightarrow g \rightarrow C \quad B \rightarrow C \rightarrow A$$
  - The potential position of $C$ is 1 (there is no need to apply any rule, as $C$ is already in the farthest left position possible).

Hence, both $B$ and $C$ have the potential to commute to position 1 on the path. Since $\text{rank}(B) < \text{rank}(C)$, $B$ will occupy position 1 in $p'_\text{temp}$.

$$p'_\text{temp} = B \rightarrow C \rightarrow A$$

- Iteration 2: Identify the second leftmost node (position 2) of $p'_\text{temp}$.
  - The potential position of $C$ is 2 (there is no need to apply any rule, as $C$ is already in the farthest left position possible after $B$).
  - The position of $A$ cannot change, as $A$ and $C$ do not commute.

Thus, $C$ and $A$ occupy the second and third positions in $p'_\text{temp}$, respectively.

$$p' = p'_\text{temp} = B \rightarrow C \rightarrow A$$

Note that this is a fast in-memory operation which does not cause any disc accesses. Therefore, the complexity has no practical impact on the retrieval performance.

4.2 Wildcard Labels

Most path-based query languages, contain two wildcard symbols, "*" and "//":

- each "*" matches exactly one node label in the data trees;
- each "//" can match zero or more consecutive node labels at the same time.

Due to the commutative semantics, in this paper, we mostly ignore the "//" wildcard symbol. "//" can match an arbitrary number of node labels, and replacing "//" with all possible numbers of "*" symbols yields a large number of alternative queries to process. In Section 5, we will present a special class of "//" tree queries that can be answered efficiently.

Given an alphabet, $A$, in order to enable wildcard queries, we extend the query alphabet to $A_\ast = A \cup \{\ast\}$, such that

$$\forall a \in A \ast \quad \ast \equiv \ast$$

where $\equiv$ denotes label equivalence. Furthermore,

$$\forall a \in A \ast \quad \text{rank}(\ast) < \text{rank}(a)$$
Figure 4: Splitting a tree into its path segments at the immutable points

4.3 Summary

In this section, we presented the concept of canonical paths. Next, we describe how to leverage canonical paths for matching trees in the presence of commutations. In Section 6, we will show how to index canonical paths efficiently to enable the creation of a structure-encoded sequence based index structure for commutative trees.

5. TREE MATCHING AND INDEXING IN THE PRESENCE OF COMMUTATIONS

In this section, we discuss how to match trees in the presence of commutations. We use path matching techniques discussed in the previous section to develop tree matching algorithms.

5.1 Immutable Nodes and Labels

Given a tree $t$ and a path $p$ on this tree, not all permutations, $p' \in P(p,C)$, are legal. For example, consider the tree in Figure 2(c) and the corresponding rules in Figure 2(a). In this case, if we consider the left-most path, we can see that although the labels $B$ and $C$ are commutative based on the rules, the node labeled $B$ and the node labeled $C$ are not commutative\(^1\). This is simply because the node labeled $B$ has another child, and therefore, is immutable; i.e., the commutativity rule $B \leftrightarrow C$ is not applicable. In order to capture the existence of such immutable nodes, we extend the alphabet $\mathcal{A}$, with immutable elements into $\mathcal{A}_{ext}$, such that

$$\forall a \in \mathcal{A}, \ (a \in \mathcal{A}_{ext}) \land (a \notin \mathcal{A}_{ext}),$$

where $a_{\perp}$ is the immutable version of $a$. The immutability of $a_{\perp}$ is ensured simply by the fact that the set, $C$, of commutativity rules can never contain any rules regarding any immutable label. Figure 4 shows an example where the immutable nodes split the tree into path segments connected in an hierarchical fashion; commutations cannot move across path segment boundaries at the immutable nodes.

5.2 Canonical Trees

We first define canonical trees which cluster trees that are equivalent under the commutativity rules.

**Definition 5.1 (Canonical trees).** Let

- $\mathcal{A}$ be an alphabet (possibly containing "$\ast\$") and $\mathcal{A}_{ext}$ be its immutably extended version;
- $t(V_i, E_i, l_i)$ be an ordered, directed, labeled tree;
- $t_{ext}(V_i, E_i, l_{i,ext})$ be a label-transformed version of $t$ such that all immutable nodes in $V_i$ have immutable labels;
- $\mathcal{P}(t,C)$ is the set of all order-preserving permutations of the tree $t$ allowed by the set $C$ of commutativity rules.

Then, the canonical form $t^c$ of the tree $t$ is a permutation of $t_{ext}$ in $\mathcal{P}(t_{ext}, C)$ (i.e., $t^c \in \mathcal{P}(t_{ext}, C)$), such that

$$\forall p_i \text{ is a path in } t^c : p_i \equiv_{t^c} p_i^c$$

where $\equiv_{t^c}$ means equivalence under label matching. \(\diamondsuit\)

Given the above definition, the following lemma holds.

**Lemma 5.1.** The following is true:

$$\forall t_{ext} \in \mathcal{P}(t_{ext}, C), \ t^c \equiv (t_{ext})^c$$

In other words, all legal permutations of a given tree $t$ (with respect to its label-transformed version $t_{ext}$) have the same canonical tree. \(\diamondsuit\)

Furthermore, the following lemma also holds.

**Lemma 5.2.** The following is true: Given $t_{ext}' \in \mathcal{P}(t_{ext}, C)$ and a preorder ranking of the paths in both trees ($t_{ext}$ and $t_{ext}'$),

$$\forall p_i \in t_{ext} \text{ and } r_i \in t_{ext}', \ p_i^c \equiv_{t^c} r_i^c$$

In other words, corresponding paths in all legal permutations of a given tree $t$ (with respect to its label-transformed version $t_{ext}$) have the same canonical path. \(\diamondsuit\)

From Definition 5.1, it is easy to see that the structure of a tree $t$ is preserved in its canonical form $t^c$ at the immutable points (with respect to its label-transformed version $t_{ext}$). Since $t^c$ preserves the order of path segments in $t$ ($t^c \in \mathcal{P}(t_{ext}, C)$), the correctness of Lemmas 5.1 and 5.2 follows when Lemma 4.1 is used for path segments of the tree $t$.

Although simple, these two lemmas enable rewriting of tree-structured data such that all legal permutations will be accessible through a common index.

5.2.1 Complexity of Tree Canonization

Given a tree $t$ with $n$ nodes, the worst case complexity of identifying $t^c$ is $O(n^3)$ in-memory operations. It is easy to see that the cost of canonizing $t$ is the total cost of canonizing all path segments of $t$ due to immutable nodes. If all the path segments of $t$ were concatenated to form a single path with immutable nodes denoting the borders of the path segments, the cost of canonizing this single path with $n$ nodes would be $O(n^3)$. Note that canonizing trees is a fast in-memory operation similar to canonizing paths, and it does not cause any disc accesses. Therefore, the complexity has minimal practical impact on the retrieval performance. \(\diamondsuit\)

5.3 Query/Tree Matching

Using the above definition of the canonical trees, we can define ordered tree matching in the presence of commutations as appropriate one-to-one mappings between the nodes of the canonical forms of two given trees.
Definition 5.2 (Tree matching with commutations). Given a data tree \( t(V_t, E_t, l_t) \) and a query tree \( q(V_q, E_q, l_q) \), \( t \) is said to be ordered matching \( q \) subject to the commutativity rules in \( C \) iff there exists a transformation

\[
\mu_* : l_t \rightarrow l_q
\]

such that \( \forall v_i \in V_t \exists v_o \in A \ ( (l_q(v_i) = "*" \rightarrow (l'_q(v_o) = a)) \land (l_q(v_i) = "*", l'_q(v_o) = a)) \) and

\[
q^c \equiv_o t^e
\]

where \( \equiv_o \) denotes ordered tree matching based on the label equivalence under \( l'_q \) labeling of the query.

Intuitively, since all legal permutations of a given tree have the same canonical form, it is enough to check the matching between the canonical forms of the trees to see if two trees do match.

5.4 Indexing Trees in the Presence of Commutations

Given a tree \( t \) and its canonical form, \( t' \), under a given set of commutativity rules, \( C \), various index structures (including ViST [7]) can be used to retrieve \( t' \) exactly. Given a query tree, \( q \), without any wildcard symbols, such index structures can be used to find if \( q' \) matches \( t' \) (or therefore if \( q \) matches \( t \) under the commutativity rules).

This, however, is not enough when the query tree \( q \) contains nodes labeled with "*" wildcard symbol. Since the data label that will match the "*" is not know in advance, the exact set of commutativity rules that apply to "*" labeled query nodes are not known. Consequently, it is not possible to create a single canonical tree query \( g' \).

5.4.1 Naive Solution I

An obvious naive solution to the problem of matching trees, when the tree query \( q \) contains \( k \) nodes labeled with "*" wildcard symbol, is to replace each of the \( k \) wildcards with letters from the alphabet, \( A \), to obtain \(|A|^k \) possible queries. Let us call this set of queries as \( Q(q) \). Each of these no-wildcard queries, then, can be canonized separately and searched in the index to obtain matching trees. Let us set the number of canonized queries as \( Q'(q) \). The query complexity is \( O(|A|^k) \) queries to the index structure.

5.4.2 Naive Solution II

Another naive solution is to index all possible legal permutations of the trees with respect to a given set of commutativity rules. This approach is efficient only if there is no wildcard symbol in the query, in which case only one query to the index structure is needed. Otherwise, the query complexity is \( O(|A|^k) \) queries to the index structure similar to the previous naive approach. Moreover, the storage space required is very large, because all possible legal permutations (with respect to a given set of commutativity rules) of the trees should be indexed.

5.4.3 Suffix-Tree Based Rule-Clustered Index

We first present a suffix-tree based rule-clustered index structure, \( RC dex \), to process tree queries with wildcards. The main observation that leads into the index structure is that given a structure-encoded sequence of a given tree (with immutable nodes properly labeled) and a set of commutativity rules, we can split the sequence into path segments as shown in Figures 5(a) and (b). Each of these path segments can have commutations, but the commutations cannot pass across immutable sequence boundaries. Therefore, each of these path segments, and their various combinations, can be indexed using a rule-clustered index \( (RC dex_{path}) \), which will be explained in detail in Section 6. Based on this observation, we build a suffix tree to capture the overall structure of trees expressed through structure-encoded sequences that consist of these path segments.

Given the two trees in Figures 5(a) and (b), we can represent them in two structure-encoded sequences:

\[
t_1 = (P_1)(P_1P_3)(P_1P_3P_4)(P_1P_3P_5)(P_1P_3P_6)
\]

\[
t_2 = (P_1)(P_1P_3)(P_1P_3P_4)(P_1P_3P_5)(P_1P_3P_6)
\]

These strings then can be combined and represented in the form of a suffix tree as shown in Figure 5(c).

Given such a suffix tree and a query tree \( q \) (which does not contain any wildcard symbol), finding a match is equal to finding a path on the suffix tree from the root to the leaf that is equivalent (under commutativity rules) to the structure-encoded sequence that corresponds to \( q \). This can easily be implemented using the commutative path index structure \( (RC dex_{path}) \) presented in Section 6. Figure 6 provides an example that uses the suffix tree in Figure 5(c).

This index structure can be used to answer queries that contain wildcard "*" characters. A query tree that contains wildcards can be expressed in the form of a structure-encoded sequence. For example, the query in Figure 7 can be

Figure 5: (a,b) Two example trees; each \( P_i \) is a path segment and (c) the corresponding suffix tree

Figure 6: Commutative tree indexing using the path-index structure discussed in Section 6. The numbers in parentheses are (nodenumber, parentnumber).
represented in the form of the following structure-encoded sequence:
\[ q = (P_{1[*]})(P_{4[*]}P_7)(P_{3[*]}P_2)(P_{1[*]}P_3P_{4[*]})(P_{1[*]}P_3P_6), \]
where each \( P_{i[*]} \) is a subpath which contains one or more wildcards. The search will follow the following steps:

- Search for \( P_{1[*]} \) will return nodes 2 (\( P_1P_2 \)) and 6 (\( P_1P_7 \)) (among others) as potential children in the suffix tree.
- \( P_1P_2 \) is eliminated as it does not match the query.
- Before \( P_1P_2 \) is further queried, the query is modified to

\[ q = (P_1)(P_1P_3)(P_1P_3P_4)(P_1P_3P_5), \]

- The search on the suffix tree continues until \( P_1P_3P_4 \) is located.
- Search for \( P_1P_3P_{4[*]} \) returns node 9 (\( P_1P_3P_6 \)) as potential child in the suffix tree.
- Search for \( P_1P_3P_5 \) returns the corresponding document.

From the above example, it easy to see that if each subpath, \( P_i \), contains \( k \) wildcards, each subsequence in the corresponding structure-encoded sequence will contain at most \( k \) wildcards. Therefore, an index structure designed for matching paths with \( k \) wildcards can be used to answer queries that contain at most \( k \) wildcard (\( \ast \)) characters.

Note that due to the prefix-based construction of the structure-encoded sequence, this index structure can also be used to find paths that start at the root and end at the leaves or at the immutable nodes. For instance, the index structure in Figure 6 can be used to find the path \( P_1P_3 \).

5.5 Optimization of the Index Structure

In this section, we show that canonical paths introduced earlier can be used for creating a suffix-tree based index structure for indexing paths under commutations. Note that, in general, the suffix-tree implementation is inefficient as it requires explicit traversal of the index structure for all paths. This can be avoided by a ViST-like encoding and representation of the suffix tree, where the ancestor/descendant relationships in the suffix tree are efficiently maintained in the form of ranges in a \( B^+ \) tree (called S-index as discussed in the structure-encoded indexing background section, Section 3).

In the case of commutative trees, the ancestor/descendant relationships are discovered based on the canonical forms of the trees in the database. Such an encoding also enables us to extend the proposed index structure to match subtrees and subpaths, starting at the root or immutable nodes and ending at the leaves or immutable nodes, as long as the individual subpaths (\( P_i \)) do not contain any uncontrolled wildcard ("//") character; i.e., "/" is used only for matching entire subpaths. Thus, if a subpath is specified, the only wildcard symbol it can contain is the "\( \ast \)" wildcard character (Figure 8).

As discussed in Section 3.1, however, in ViST the structure-encoding index also requires a D-index which maintains subpaths of the trees (subsequences in the structure-encoded sequences of the trees). Since, for each subsequence of the structure-encoded sequence, this index is queried, the efficiency of the tree retrieval relies on the efficiency of this subsequence index structure.

In the next section, we present an index structure that indexes paths under commutativity rules. As shown in Figure 9(b), this index structure can be used, along with \( B^+ \) trees to maintain suffix tree ranges, to implement an efficient commutative tree index structure. Figure 9(a) shows the ViST index structure for comparison.

6. INDEXING COMMUTATIVE PATHS

In this section, we discuss how to index paths using their canonical forms and introduce a two-level indexing mechanism, \( \text{RC} \text{dex}_{\text{path}} \), for indexing paths with " \( \ast \)" wildcard labels. The \( \text{RC} \text{dex}_{\text{path}} \) forms the basis for the \( \text{RC} \text{dex} \) index structure as discussed in Section 5.5.

Given a path \( p \) and its canonical form, \( p^c \), under a given set of commutativity rules, \( C \), various index structures (including compact tries or hash tables) can be used to retrieve \( p^c \) exactly. Given a query path, \( q \), without any wildcard symbols, these index structures can be used to find if \( q \) matches \( p^c \) (or therefore if \( q \) matches \( p \) under the commutativity rules). Therefore, we will not discuss path queries without wildcard symbols any longer.

This, however, is not enough when the path query \( q \) contains nodes labeled with " \( \ast \)" wildcard symbol. Since the data label that will match the " \( \ast \)" is not know in advance, the exact set of commutativity rules that apply to " \( \ast \)" labeled query nodes can not be known. Consequently, it is not possible to create a single canonical query \( q^c \) that will match each possible data path \( p \). In the next subsections,
we discuss alternative solutions to this challenge.

6.1 Naive Solution I

An obvious naive solution to the problem of matching paths, when the path query \( q \) contains \( k \) nodes labeled with the “⋆” wildcard symbol, is to replace each of the \( k \) wildcard symbols with letters from the alphabet, \( A \), to obtain \( |A|^{k} \) possible queries. Let us call this set of queries as \( Q(q) \). Each of these no-wildcard queries, then, can be canonized separately and matched to the index to obtain matching paths. Let us denote the set of canonized queries as \( Q^c(q) \). The query complexity is \( O(|A|^{k}) \) queries to the index structure. The storage requirement, however, is \( O(m) \) where \( m \) is the number of paths indexed.

6.2 Naive Solution II

Another naive solution to path matching under commutativity is to index all possible legal permutations of the data paths with respect to a given set of commutativity rules. When a query is submitted, matching data paths can be found efficiently consulting the index structure without any canonization. When the query path contains \( k \) nodes labeled with the “⋆” wildcard symbol, the “⋆” wildcards in the query are replaced by the labels from the alphabet and search in the index. This approach is efficient only if there is no wildcard label in the query, in which case only one query to the index structure is needed. Otherwise, the query complexity is \( O(|A|^k) \) queries to the index structure similar to the previous naive approach. Nonetheless, the storage space is very large, because all legal permutations (with respect to a given set of commutativity rules) of the data paths must be indexed. Hence, in the worst case where all the labels commute, the storage complexity of this approach is \( O(n!) \) for each path indexed, where \( n \) is the path length.

6.3 Rule-Clustering

The alternative approach we rely on for indexing commutative paths is rule-clustering. In order to answer path queries possibly with wildcards (“⋆”s) efficiently, we propose a special index structure called \( RC_{dex, path} \). In this section, we explain how a path query containing “⋆”s can be answered using \( RC_{dex, path} \). For queries without wildcard labels, the task of path matching is much simpler, and it follows the explanation given here for “⋆” queries, without the discussions relating to “⋆” wildcard labels.

The \( RC_{dex, path} \) index contains two levels of index structures. In the first level, a rule clustering \((RC)\) index structure is used to index most general canonical forms of the input paths. This index structure achieves the first level of clustering. Then, a second level of index structures refine each general cluster. The two levels together form the \( RC_{dex, path} \).

6.3.1 Clustering “most general” canonical paths

The first level index structure, \( RC \), is created as follows:

![Figure 10: Rule clustering examples: (a) The two labels can be clustered; (b) none of the labels can be clustered](image)

Given a data path \( p \) to be inserted into \( RC \), we

1. create a new rule set \( C_{s} \) containing three rules
   \( c_{i} = (a \leftrightarrow b) \), \( (a \leftrightarrow \text{"⋆"}) \), and \( (b \leftrightarrow \text{"⋆"}) \)
   for each rule \( c_{i} \) in \( C \). We call the additional rules, the \( \text{"⋆"}-\text{commutativity} \) rules.
2. pick all possible combinations of at most \( k \) nodes in \( p \) and replace their labels with “⋆”s. Let \( p' \) be one of the resulting paths.
3. canonize each \( p' \) using \( C_{s} \) to get \( p'^{c_{s}} \).
4. store path \( p' \) under each index \( p'^{c_{s}} \) in \( RC \).

Note that the index structure assumes that the maximum number of wildcard labels on each path is bounded by \( k \); i.e., the number of wildcards on the query can be no more than \( k \). Also, note that each path is indexed \( O(n^k) \) times in \( RC \). Therefore, the index structure is storage efficient only when \( k < < n \).

Given a query \( q \) with at most \( k \) wildcards (“⋆”s), \( RC \) is used as follows:

1. Using the new rule set, \( C_{s} \), we compute the canonical form, \( q^{c_{s}} \) of the query, \( q \). Intuitively, this is the most general canonical form of \( q \).
2. We, then, search \( q^{c_{s}} \) in \( RC \).

Given a query, \( q \), all paths having the same general canonical form as the query can be found using only one query to \( RC \). Obviously, all of the paths indexed under \( q^{c_{s}} \) do not match the query \( q \). It can be shown that a single general canonical path (of length \( n \)) in \( RC \), clusters

- only 1 path, if \( C_{s} \) is empty; and
- \( n! \) paths (assuming all these paths are in the database) if \( C_{s} \) contains all possible commutativity rules;

Therefore, we need to further refine our search.

6.3.2 Refining “most general” clusters

Let the cluster of paths indexed using the index \( s \) be denoted as \( RC(s) \). In order to search the canonical path \( q \) (subject to the commutativity rules in \( C \)) within the cluster, \( RC(q^{c_{s}}) \), we use a second level index structure. We create/maintain this second level index structure as follows: Let us assume that \( p \) is a path that is being indexed under the cluster \( RC(s) \) that corresponds to an index path \( s \).

1. The first step in the process is the creation of a rule set specific to the given index \( s \):
Paths = \{ACB, BBC, CBC\}
\[ A = \{A, B, C, D, E, F\} \]
\[ C = \{A \leftrightarrow B, \quad B \leftrightarrow C, \quad A \leftrightarrow C\} \]

(a)

Figure 11: An example: (a) paths, the alphabet and commutativity rules, and (b) indexing the paths in \( \mathcal{RC}_{\text{dexpath}} \) for 1-\* queries

- If \( p \) is the first path to be indexed under \( \mathcal{RC}(s) \), then we create a rule set, \( C_+ \), which contains

\[ (a \leftrightarrow b), (a \leftrightarrow “s_k”), (b \leftrightarrow “s_k”), \text{and} (“s_k” \leftrightarrow “s_b”) \]

for each rule \( c_i = (a \leftrightarrow b) \in C \) where \( a, b \in p \). In addition, \( \forall a \in p \), we include

\[ (a \leftrightarrow “s_k”) \]

into \( C_+ \). We call the rules that apply to the new term \( “s_k” \), the \( “s_k”\)-commutativity rules. Note that

\[ \forall a \in A \quad “s_k” \equiv \exists_1 a, \]

where \( \exists_1 \) denotes label equivalence. Furthermore,

\[ \forall a, c \in A \quad \text{rank}(“s_k”) < \text{rank}(c) \]

- If \( p \) is not the first path to be indexed under \( \mathcal{RC}(s) \), then we update \( C_+ \) by adding the rules (if they are not in \( C_+ \), already)

\[ (a \leftrightarrow b), (a \leftrightarrow “s_k”), (b \leftrightarrow “s_k”), \text{and} (“s_k” \leftrightarrow “s_b”) \]

for each rule \( c_i = (a \leftrightarrow b) \in C \) where \( a, b \in p \). In addition, \( \forall a \in p \), we include (if they are not in \( C_+ \), already)

\[ (a \leftrightarrow “s_k”) \]

into \( C_+ \).

2. We compress \( C_+ \), to achieve rule-clustering, by reducing the number of rules in \( C_+ \). We note that for some symbols \( a \) and \( b \) that appear in \( C_+ \), \( “s_a”- \) and \( “s_b”\)-commutativity rules may essentially be identical to each other, meaning that both symbols commute with the same set of symbols that appear in \( C_+ \). In such a case, we can merge the corresponding wildcard rules and rule sets:

- if for all rules of the form \( (”s_a” \rightarrow x) \) in \( C_+ \),

\[ (“s_b” \rightarrow x) \]

is also in \( C_+ \), then we create a combined symbol \( “s_a,b” \), remove \( “s_a” \) and \( “s_b” \), and replace the above rules with

\[ (“s_a,b” \rightarrow x) \]

Furthermore,

\[ \text{rank}(“s_a,b”) = \min\{\text{rank}(“s_a”), \text{rank}(“s_b”)\} \]

Figure 10(a) shows an example.

In general, if there are \( m \) symbols, \( a, b, \ldots, m \in C_+ \), such that all of them commute with the same set of symbols in \( C_+ \), then a combined symbol \( “s_{a,b,\ldots,m}” \) is created; all \( “s_a” \)’s that exist in \( C_+ \) (where \( \alpha \) may be any proper subset of symbols \( a, b, \ldots, m \)) are removed; and all \( “s_a \leftrightarrow x” \) rules that exist in \( C_+ \) are replaced with

\[ (“s_{a,b,c,\ldots,m}” \leftrightarrow x) \]

Similarly,

\[ \text{rank}(“s_{a,b,\ldots,m}”) = \min\{\text{rank}(“s_a”), \ldots, \text{rank}(“s_m”)\} \]

Note that each rule in \( C \) can either bring together two clusters (e.g., the rule between \( A \) and \( B \) in Figure 10(a)) or destroy an existing cluster (e.g., the rule between \( B \) and \( C \) in Figure 10(b) destroys the \( A, B \) cluster).

The total number of wildcard symbols is limited by a function of the original alphabet size and the rules, denoted as \( f(|A|, |R|) \), where \( A \) and \( R \) are the alphabet and the rule set, respectively. Note that \( f(|A|, |R|) \leq |A| \). Figure 11 depicts an example for three paths inserted and clustered into \( \mathcal{RC}_{\text{dexpath}} \) in order to answer 1-\* queries efficiently. The figure also shows that when the rules allow for effective clustering, the total number of wildcard symbols in \( C_+ \) for \( \mathcal{RC}(s) \) does decrease. Each \( \mathcal{RC}(s) \) may have one or more clusters, even though it is not apparent from the example in Figure 11. As discussed above, “\*” symbols in \( C_+ \) for \( \mathcal{RC}(s) \) may be combined to further cluster the paths. Figure 11 demonstrates an example of this by \( “s_{a,b,c} BC” \); all paths that correspond to this canonical form are indexed under \( “s_{a,b,c} BC” \) in \( \mathcal{RC}_{\text{dexpath}} \). The number of \( “s_{a,b,c}” \)’s in \( C_+ \) for \( \mathcal{RC}(s) \) can be at most \(|A| \), which may happen only if there exist no two or more symbols in the alphabet, \( A \), such that they have the same set of commutations applicable; i.e., they commute with the same set of symbols. Therefore, the total number of wildcard symbols in each \( C_+ \) is reduced by the commutativity rules if they allow for efficient clustering.

3. If the new rules added to \( C_+ \) (due to path \( p \)) invalidates some of the \*-rules (created in previous path insertions) in \( C_+ \), we then decompress those \*-rules. Decomposition of some of the \*-rules is needed only if new symbols (or rules) added to \( C_+ \) break the condition that the symbols in the \*-rules must commute with the same set of symbols in \( C_+ \). For instance, in Figure 10(b), the insertion of the rule \( B \leftrightarrow C \) (possibly due to the insertion of a path that contains the symbol \( C \)) destroys the symmetry between labels \( A \) and \( B \) present in Figure 10(a), and hence, necessitates a decomposition of the wildcard symbol \( “s_{a,b}” \).

Note that compression/decompression of the “\*” symbols can easily be avoided by creating only one global \( C_+ \) rule set and using it for all second level indexes, instead of creating a clustered rule set for each index. This, however, reduces the amount of clustering that can be achieved.

4. Then, for each selected node of \( p \), we replace its label (e.g., \( a \)) with a corresponding wildcard (e.g., \( “s_a” \) or \( “s_{a,b,\ldots,m}” \)). Let us denote the corresponding labeling \( \tilde{p} \). Note that there are multiple paths \( p \) that are indexed under replacement path \( \tilde{p} \).
5. We canonize $p'$ to get $p^{c+c}$ under the rule set $C_+$.
6. Finally, we store the original path $p$ under the index $p^{c+c}$.

Figure 12 depicts the two-phase $RC_{dex\text{path}}$ indexing process. Given a query $q$ and a corresponding first level cluster, $RC(q^{c\star})$, we use the second level index structure described above to find all the matching paths.

1. We replace the original wildcard labels with the replacement wildcards in $C_+$ (e.g., $*$, $*_a$, $*_a$ or $*_a*_b$...) to get $f^k(|A|, R)$ individual queries. Let us use $q'$ to denote one of these replacement queries.
2. Using $C_+$, we compute the canonical form, $q'^{c+c}$ of the replacement query $q'$.
3. We, then, search $q'^{c+c}$ in the second level index.

Figure 13 depicts the querying process.

6.3.3 Path Query Complexity

For a given query with $k^{\star\star}$ wildcard characters, the total number of queries executed is $f^k(|A|, R) \leq |A|^k$. This is because each $\star$ wildcard label in the original query is replaced by all replacement wildcards in $C_+$, and the number of the replacement wildcards is $f(|A|, R) \leq |A|^k$. Thus, for a $k^{\star}$ query, the number of queries executed is $f^k(|A|, R) \leq |A|^k$. If $f(|A|, R) < |A|$; rule clustering is more efficient than both the naive techniques. This can be seen clearly when the rule set $(R)$ allows for effective clustering by reducing the number of clusters in each $RC(s)$, as shown through our experiments in Section 7.

$RC_{dex\text{path}}$ assumes that the maximum number of wildcard labels on each path is bound by $k$. Since every path is indexed $O(n^k)$ times in $RC$ for $k^{\star\star}$ queries, the storage requirement is $O(n^k)$ for each path indexed. Therefore, the index structure is storage efficient only when $k << n$.

<table>
<thead>
<tr>
<th>Data Set Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of paths</td>
</tr>
<tr>
<td>Tree depth (path length)</td>
</tr>
<tr>
<td>Alphabet size</td>
</tr>
<tr>
<td>Commutativity rule sets</td>
</tr>
<tr>
<td>All applicable rules</td>
</tr>
<tr>
<td>Number of $\star$ wildcards in queries</td>
</tr>
</tbody>
</table>

Table 1: Experimented configurations

6.4 $RC_{dex\text{path}}$ and $RC_{dex}$

As discussed in Section 5.5, $RC_{dex\text{path}}$ index structure, for indexing paths under commutativity rules, presented here can be used for creating a structure-encoding based tree index structure, $RC_{dex}$. In general, for each subsequence in the structure-encoded sequence of a given tree query, the $RC_{dex\text{path}}$ index structure is consulted to find the matching subsequences and other index structures (such as $B^+$ trees) are used to ensure structural matches (Figure 9(b)). Therefore, the number of queries to the $RC_{dex\text{path}}$ index structure for a tree query depends on the length of the query sequence.

7. EXPERIMENTS

In this section, we present experiment results that validate the efficiency and effectiveness of the indexing algorithms presented in this paper.

Setup: For the experiments, we created various commutative data sets, each with different commutation properties and number of objects.

The experimented configurations (Table 1) include data which have different degrees of allowed commutations: no rules corresponds to the case where there are no commutations (obviously, in this case, $RC_{dex\text{path}}$ is not the most suitable index structure; we include this for the sake of completeness); 50 rules correspond to the situation where there are 50 randomly created rules in the system; 5 inter-commutative rules correspond to the case where the alphabet is split into 5 groups such that symbols within each group commute, but they do not commute with symbols in other groups; and all applicable rules correspond to the extreme case where all symbols are allowed to commute with each other. Also, in order to observe the impact of the number of $\star$ wildcards in queries, we also created 1- and 2-$\star$ index structures.

In order to observe and interpret the performance of the proposed optimization mechanisms, in addition to $RC_{dex\text{path}}$, we indexed all input paths after canonizing them, using a hash table, and for each 1- and 2-$\star$ commutative query, we generated multiple queries, each having different canonical form, to the hash table. This index structure corresponds to the Naive Solution I in Section 6.2. The implementation of the algorithms is done in Java and ran on Redhat 7.2 Linux workstations, with 1 GB RAM, 1.8 GHz Pentium IV processor, and 20 GB hard disk. For disk-based hash implementations, we use a public domain library called HashStore [13].

Correctness: The first observation is that the results returned by $RC_{dex\text{path}}$ and the naive indexing approach were identical, showing that the rule-clustering approach indeed produces correct document clusters, without introducing any misses or false positives.
Impact of the number of paths indexed to the querying time: Based on the amount of clustering opportunities, the number of paths in the database has varying impacts on the retrieval performance. For example, when there is only one “⋆” in the query, the number of clustering opportunities is small and the retrieval performance of the $RC_{dex}$path slightly deteriorates as the number of paths increases (Figure 14(a)). On the other hand, Figure 14(a) also shows that when there are two “⋆”s in the query (i.e., when the number of clustering opportunities is larger), the retrieval performance improves as the number of paths increases. In contrast, the retrieval performance of the naive approach stays constant, independent of the number of clustering opportunities. This shows that $RC_{dex}$path indeed achieves rule clustering effectively. The length of the indexed paths on the other hand impacts both the naive approach and $RC_{dex}$path. The impact of the alphabet size is negligible compared to the impact of the path size when the number of “⋆”s is equal to 1 (Figure 14(b)). Both figures show that $RC_{dex}$path is at least an order faster than the naive approach.

Impact of the possible commutations: Figure 14(c) shows the impact of clustering on the query performance for 1-⋆ and 2-⋆ queries, respectively. In the extreme case, when all possible commutations are allowed, $RC_{dex}$path performs an order faster than when a small number of commutations are available. The performance of the naive approach on the other hand somewhat suffers from the increase in the number of commutations because it does not benefit from clustering. $RC_{dex}$path works multiple orders faster as the amount of clustering increases.

Impact of the number of “⋆”wildcards in queries: Figure 14(c) also compares the impact of the number of wildcards in the query on the query performance. Although, there are less number of documents indexed in the case of 2-⋆ queries than that in the 1-⋆ case, the naive approach is multiple orders slower when there are more wildcards. The $RC_{dex}$path, on the other hand scales very well to the increase of the number of wildcards in the query as the performance degradation is minimal. This pattern is also visible in Figure 14(a).

Summary: The experiments, as depicted in Figures 14(a) through (c), showed that the $RC_{dex}$path indeed works efficiently and it scales better than the naive approach when the number of clusters in the data as well as the number of wildcards in the query increase. The performance gain of $RC_{dex}$path comes with storage and indexing burden due to the ⋆ query support. However, dynamic clustering of paths reduces the storage requirements.

8. RELATED WORK

Providing efficient support for structured tree-based queries with branches and wildcards poses one of the most challenging tasks in XML document indexing.

The most well-known index structures used for indexing path expressions are DataGuide [8], T-Index [14], IndexFabric [15], and APEX [16]. The DataGuide is a structural summary of a database, and makes it possible to query XML documents based on their structure when used for XML document indexing. In the DataGuide, each label path leads to exactly one DataGuide object, e.g. a component in an XML document. However, the strong DataGuide is restricted to path queries starting from the root, and hence, it is not useful for finding imperfect sub paths, i.e. partial matching queries. Moreover, storage overhead of the strong DataGuide is high for deeply nested structures. The T-Index is a non-deterministic structure for both tree and graph databases. It is tailored to fit queries matching a given path template, thus not being an aid in sub path matching. The IndexFabric indexes paths as well as the content of tree databases in a balanced hierarchy of Patricia tries [15]. The IndexFabric is effective for the retrieval of multimedia path expressions, but it becomes inefficient for the retrieval of range-based and sub-path queries, because a partial-match query is broken into several simple path-expression queries. The APEX[16] indexes label paths and maintains information about frequently used paths. The approach is very efficient when the same data is queried over and over again. XISS [17] indexes XML documents based on a numbering scheme for XML elements, and it uses elements as the basic unit of a query. A query is represented as a regular path expression, and complex path expressions are decomposed into a collection of simple path expressions. Each simple path expression is processed separately, and the results are joined together to obtain the final result of the query. Even though the approach is flexible in that all types of structural...
queries can be constructed using simple path expressions, it requires intermediate results to be joined together, which is an expensive operation.

A native XML database, called Timber, is built on a bulk algebra for manipulating trees [18]. As Timber stores XML documents natively, it has the advantage over mapping the XML data into the relational model in order to utilize existing relational database systems. As in other approaches proposed for tree structure matching, Timber processes pattern tree queries by independently locating candidate nodes, and determines for each pair of candidate nodes whether they satisfy the structural containment relationship or not. Thus, a query is processed one pattern-tree edge at a time joining the nodes satisfying the structural containment relationship based on the pattern tree query. As this involves structural join operations to obtain the final query result, it is an expensive operation. The structural join operations on nodes are expensive; therefore, most recent approaches try to avoid them in processing tree matching queries.

One such an approach called Extended DataGuide (XDG) is proposed in [19]. The XDG is based on trees, and provides a structural summary of the XML source. An XDG is a DataGuide where each node is assigned a unique node number that is obtained by the pre-left order traversal of the XDG. The node labels are indexed by a term index T-Index, which gives the sequence of all nodes with the same label in the XDG. Given two nodes in the XDG, it is also possible to determine if one is an ancestor of the other. A second index, called P-Index, which is a path index, is used to determine the instances of a certain rooted label path, and also to identify the addresses of the physical data locations in an efficient way. However, the procedure for processing pattern-tree queries is similar to other structural join approaches in that a pattern-tree query is decomposed into smaller units. The advantage of XDG stems from the fact that structural join operations are not performed on nodes but rather complete rooted paths, thereby reducing the number of joins needed to obtain the query answers.

Another recent indexing method ViST eliminates structural join operations altogether by using tree structures as the basic unit of a query[7]. ViST uses a document encoding scheme, called structure-encoding, which produces a sequence from the structure of an XML document, derived from a prefix traversal of the document tree. A structure-encoded sequence of a document is a pair of symbol and prefix values, where symbol and prefix represent a node in the XML document tree and the path from the root to the node, respectively. Thus, XML querying is transformed into structure-encoded noncontiguous sequence matching. ViST supports direct ("*") and arbitrary ("*/") ancestor-descendant pattern-tree queries. Both XDG and ViST support dynamic insertions.

In contrast to the above work, we focus on indexing of commutative paths and trees, where a set of commutativity rules describe which labels in the tree can be swapped without affecting the semantics of the data. As in ViST, we use structure-encoding to minimize the number of structural joins; however, in order to reduce the complexity due to commutations, we introduced rule clustering.

9. CONCLUSION

In this paper, we showed how to index tree data in the presence of commutations between elements. We showed that efficient indexing of commutative trees depends on efficient indexing of properly chosen canonical paths. We demonstrated that it is possible to cluster and retrieve paths with commutations efficiently. We introduced novel rule-clustering techniques to enable efficient processing of wildcard path queries using a novel index structure RCdex.path. The experiments we conducted showed that the proposed index structure worked very efficiently, especially when the commutativity rules lead into clustering in the data.

10. REFERENCES