Journal of Computer Science and Cybernetics, V.39, N.4 (2023), 323–342
DOI no 10.15625/1813-9663/19018

AN IMPROVED INDEXING METHOD FOR QUERYING BIG XML FILES

DINH DUC LUONG[1], VUONG QUANG PHUONG[2], HOANG DO THANH TUNG[2,*]

[1]*Food Industrial College, 426 Nguyen Tat Thanh Street, Tan Dan Ward, Viet Tri City, Phu Tho Province, Viet Nam*

[2]*Institute of Information Technology, Vietnam Academy of Science and Technology, 18 Hoang Quoc Viet Street, Cau Giay District, Ha Noi, Viet Nam*

Abstract. The exponential growth of bioinformatics in the healthcare domain has revolutionized our understanding of DNA, proteins, and other biomolecular entities. This remarkable progress has generated an overwhelming volume of data, necessitating big data technologies for efficient storage and indexing. While big data technologies like Hadoop offer substantial support for big XML file storage, the challenges of indexing data sizes and XPath query performance persist. To enhance the efficiency of XPath queries and address the data size problem, a novel approach that is derived from the spatial indexing method of the R-tre family. The proposed method is to modify the structure of leaf nodes in the indexing tree to preserve XML-sibling connections. Then, new algorithms for constructing the new tree structure and processing sibling queries better are introduced. Experimental results demonstrate the superior efficiency of sibling XPath queries with reduced data sizes for indexing, while other XPath queries exhibit notable performance improvements. This research contributes to the development of more effective indexing methods for managing and querying large XML datasets in bioinformatics applications, ultimately advancing biomedical research and healthcare initiatives.

Keywords. Big data, indexing, analysis of XML, bio-XML files, XML query processing.

1. INTRODUCTION

XML documents store structured text data, also known as semi-structured data [1, 2]. They have been popular for decades because of their flexible data structure and easy sharing over the Internet. Usually, the XML documents used on the internet are not very large. However, the rapid growth of the internet and biotechnology recently has produced very large bioinformatics XML documents that can reach gigabytes, and terabytes [3]. Reputable sources such as SRA (Sequence Read Archive), NCBI Genome, and Ensembl provide access to such data, including decoded sequences. For example, on NCBI's website, the data stored in August 2019 was 6.2 terabases, statistics since 1982 show the amount of data doubled every 18 months. Therefore, despite a lot of areas where XML documents are used to store data, we focused our research and experimentation on one of the most prominent areas recently in the creation of giant XML documents, Bioinformatics [4, 5]. Bioinformatics XML documents contain definition data in the form of text tags, which create a flexible XML structure that is

*Corresponding author.
E-mail addresses: luongdd@fic.edu.vn (D.D. Luong); vqphuong@ioit.ac.vn (V.Q. Phuong); tunghdt@ioit.ac.vn (H.D.T. Tung).

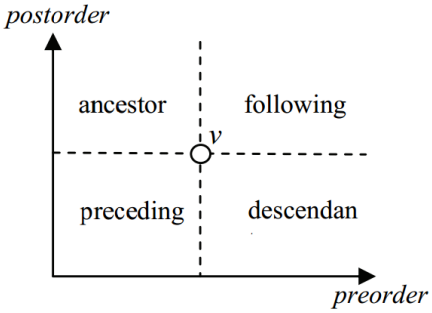

Figure 1: XPath axis in 2-dimensional space

rich and varied due to contributions from numerous independent biological individuals and organizations [6–8]. Current research on bioinformatics XML documents typically separates the data into two sources:

- The first source is data of tags and XPath queries can be used to get information on the relationship among entries of the data.
- The second source is biological data, which can be sequences of DNA, protein, and RNA. There are also numerous ad-hoc queries available, such as similarity search queries.

In this paper, we will focus on the first data source and XPath queries [9, 10]. To locate specific elements in an XML document, we use the positional path, which is an expression that specifies how to navigate in an XML tree from one node to another. A positional path consists of positional steps, each step consisting of the "axis", "node", and "predicate". To locate a particular node in an XML document [11, 12], we incorporate multiple positional steps, each of which refines the search process. An axis indicates which node is related to the current node, which should be included in the search [13–15]. The XPath [16] specification lists a family of 13 axes, an axes-based query like this: $descendant :: b/followsibling :: [location() != Last()]$. XPath is designed to query XML documents and retrieve data through elements (tags) based on a family relationship path (father, brother, and child) or relative (ancestor, descendants, preceding, and following) among the elements.

In the 13-axis XPath queries, we found out the first concern that 46% (6/13 axes) are siblings or derived from sibling axes. In the remaining 7 axes, there are some axes that retrieve most of the XML data, such as the preceding or the following of an element, which means almost half of the data size. According to Figure 1, we can see that the preceding or following are impractical for bioinformatics large data because the data returned will be too large to analyze. Therefore, instead of studying to improve the efficiency of all XPath query axes as in recent studies, it would be better if we only focus on improving the efficiency of commonly used axes such as the siblings, children, and ancestors, in which special attention is paid to the sibling.

Next our concern is how to index such a large amount of data [17–20]. Those documents can only be stored on multiple hard disks, or in a distributed storage system. To minimize the number of times data needs to be fetched from the hard drive, indexing methods need to be supported by querying methods to maximize the use of main memory, such as Cache and Buffer. In addition, the indexing methods require that the resulting data match the

query request as much as possible to reduce post-processing costs. For example, the XPath queries extract all data with the same origin/sibling relationship of a type of White Mouse or extract all data that are descendants of African pigs. With exceptionally large data sources, traditional indexing methods can have the following problems, the first is the amount of data that is indexed will be very large, even larger than the data source, and the second is the amount of data generated by queries can also be too large to analyze.

Therefore, a new indexing method for bioinformatics XML data needs to solve two problems simultaneously: (1) To reduce data source size as well as not to generate too much of the indexed data [21, 22]. (2) To optimize common XPath query performance on indexed data [23, 24]. The paper uses a method to reduce data source size by converting text data to numbers and then proposes a new indexing tree structure by modifying the R-tree [5] structure optimized for sibling queries and some new query algorithms to enhance sibling-related query performance. The proposed method has been implemented by modifying an original R-tree representing the R-tree family. In the future, this approach can be extended to other variants such as R*, R+ tree,.. similarly.

The paper covers the following content, Section 2 will present through recent studies related to our research. Section 3 will present the proposed method and algorithms. Section 4 is the experimental results of the proposed method. Section 5 is the conclusion of the paper.

2. RELATED WORKS

The XML semi-structured data model, like any structured data model, also includes widely used proprietary query languages, namely XPath and XQuery [26–28]. XPath serves as the fundamental form of XQuery and can work in two modes: sequential and indexed. In sequential mode, XML data is queried without preprocessing and requires a full data read [29, 30]. In indexed mode, XML data is preprocessed to build another data structure for efficient querying, which also generates smaller output data like a compression method and allows direct querying on output data without decompressing. This compression method usually focuses on solving only the part of the descriptive data that makes up the XML document structure.

The paper will seek to resolve the problem encountered on exceptionally large XML documents, here are some methods that have been implemented. XGrind [31] and Xpress [32] are two compression methods for XML documents. XGrind replaces attribute and element names with unique characters, while Xpress encodes XML document label paths into separate segments using inverse math. Both approaches enable direct querying on compressed documents, but they only handle simple path queries and exhibit a linear size ratio between compressed and original data, which limits their ability to efficiently support multiple queries.

Hence, the approach XQzip was introduced by Cheng and NG [33]. XQzip employs the Structure Index Tree (SIT) for the original XML document. On the other hand, XQueC by Arion et al. [34] employs the structure summary tree for storing XML documents. These techniques yield improved compression ratios and quicker query performance. To enhance the efficiency of XPath queries, Arroyuelo et al. [35] introduced the concept of constructing a label tree from the XML tree structure and subsequently employing the array bit index method for

compressing XML documents. The data segment is compressed through conventional means. In the method proposed by Qian et al. [36], the XML document's structure is separated from its content and compressed independently to optimize data transfer bandwidth and decrease latency. However, queries are limited due to the poor number of indexing methods applicable to text data. To overcome this limitation, the structural description by tags in the XML document is converted into a digital order/coordinate form, enabling the application of stronger indexing methods.

Dietz [37] introduced the initial approach that incorporates structured encoding techniques alongside algorithms for tree traversal based on pre-order and post-order values. One limitation of this approach is the need to repeatedly compute the pre-order and post-order pairs whenever new data is introduced. To address this concern, Li and Moon introduced XISS [38], which employs the B+-tree method for indexing both elements and attributes. However, the query performance of this technique is not well due to numerous intermediate results, and each time a new node is added, a complete reorganization becomes necessary. XR-tree [39] is a new method that focuses on solving the problem with two lists of nodes containing descendants and ancestors, and the task is to find parent-child pairs. XR-stack algorithm improves B+-tree by using a list $SL(N)$ containing elements Ei that are pointed to by at least one entry m but not by the ancestors of N, and nodes are linked from left to right. The size of $SL(N)$ is difficult to estimate accurately due to the variety of XML documents, but is expected to be larger than the original data file. XR-tree uses a structured join to efficiently find elements with relationships between two data sets. Another approach is to move tag names into 2D space using pre order and post-order for X and Y axes, respectively. This approach divides the four main XPath query paths equally into four spaces, simplifying node selection based on their relationships in space. Some studies have used R-tree methods to index this data.

The R-tree is a widely used indexing method for multidimensional data, especially for non-uniformly distributed data. It divides the data space into rectangles and allows for overlapping, with each rectangle representing a Minimum Bounding Rectangle (MBR) containing nodes of the tree. Algorithms designed for the R-tree are optimized for minimal overlapping MBRs. The AR*-tree [40] is an improved algorithm from T.Grust [41] that converts each entry in the XML document into *n-dimensional* space by splitting it into n entries of *1-dimensional*, creating a node group. Each entry contains index information in only one dimension, and the algorithm chooses which dimensions are necessary for querying while ignoring unnecessary ones. This method generates a large index file size due to the use of many pointers. Therefore, with large XML documents, this method will create too large an index size.

From the above methods, it shows that a general characteristic is that they have not solved queries optimally by indexing and not optimized the size of the generated index file. A balanced approach is necessary to optimize data size and ensure XPath query performance. However, the existing researches mainly focus on small, randomly generated XML documents. We see that research needs to focus on a prominent field where there is large and exceptionally large XML data so that the evaluation results are practical and oriented. In this paper, our research focuses on bioinformatics XML documents because they have a large size and are very diversified.

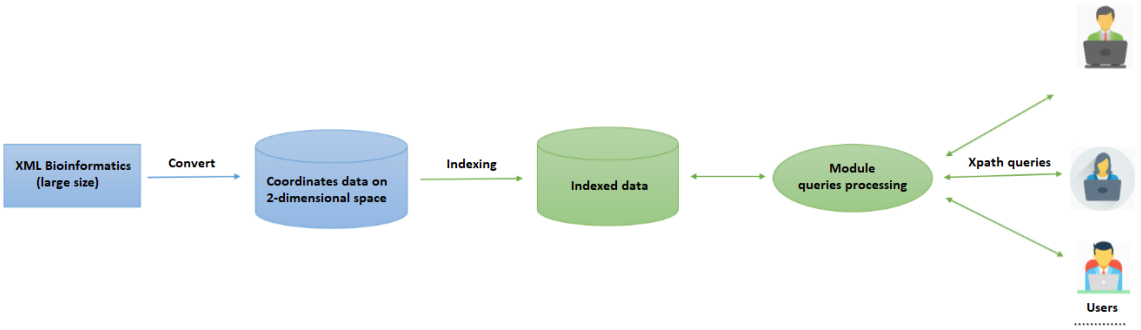

Figure 2: Generalized process model

3. PROPOSED METHOD BIOX-TREE

To make it easy for the reader to visualize, Figure 2 describes the general process of how the indexing of a large bioinformatics XML document according to the proposed method, called BioX-tree, is performed. The large bioinformatics XML document is converted into 2-dimensional space in Dietz [37] way. Then, these data is indexed to create an indexed data file. From here based on this, it is possible to execute XPath queries.

3.1. Converts XML document to 2-dimensional spatial coordinates

Dietz's method transforms the XML documents by extracting XML-tag locations and converting them into spatial representations using Cartesian coordinates. An XML document is described as a tree with a parent-child hierarchy, and then nodes are indexed with a value pair based on first and second-order tree traversal algorithm, this value pair forms the NodeID for each tag name [38]. However, using spatial indexing methods based on R-tree to index the transformed XML data is more likely to experience problems:

First, it is the problem of overlap proportional to the scope of the query space. That means, when a query has a large spatial scope, it is more likely to overlap with the node spaces of the index tree (they also overlap). As a result, it takes more time to search the tree, retrieve the data, and post-correct the returned data. Meanwhile, the XPath queries after spatially being converted will be huge windows for the search, which will greatly affect the performance of the queries. Figure 3 presents an XML document instance with a small number of points representing XML data in a plane. Assuming we want to retrieve all descendants of the current node E, we would need to use the query window {pre(E), ∞; 0, post(E)} [42]. However, the necessary search area is the white rectangular region, with its top-left corner at node E. Thus, the very large gray-shaded area in that query window is redundant. Meanwhile, we easily see that the points distributed in the space are converged in the middle and stretched infinitely.

Second, the XML data structure shows explicit and orderly relationships between elements such as father, child, and brother,... However after transformation, those relationships are only shown in the distribution of the four regions as shown in Figure 3.

Finally, the R-tree will distribute the elements regardless of their relationship scattered across the nodes of the tree. As a result, XPath queries executed on the R-tree will require a lot of execution overhead and will result in redundant results. This will incur post-processing

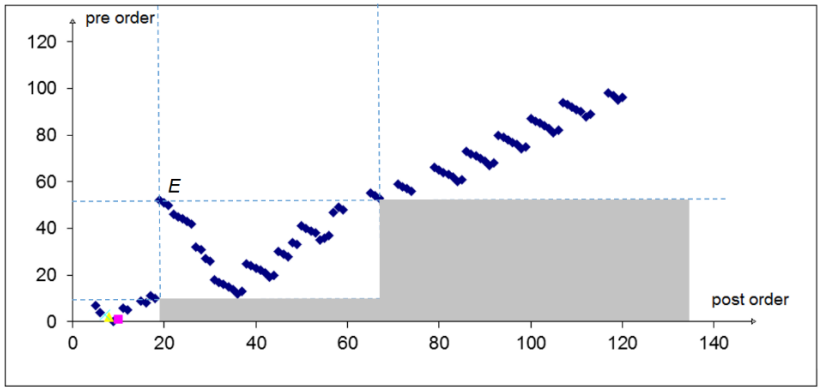

Figure 3: The spatial scope of the descendant query of E includes gray and white areas

costs to shorten the results.

According to our observations, the data is distributed on a linear axis like an airplane wing. We tested various XML documents with a few hundred nodes to a few hundred thousand nodes showing the same result. The reason may be: (1) The characterization of the data after spatial transformation; (2) The number of XML element sibling relationships is much more than the parent-child relationship. All the previous methods do not consider these characteristics, they only focus on improving the R-tree structure to suit XPath queries.

3.2. BioX-tree index structure

Based on the above analysis, we construct an improved R-tree method that takes care of data characterization and overcomes the disadvantages of XPath queries, named BioX-tree. The BioX-tree applies the strategy of changing the structure of the leaf nodes and improves the Insert/Split algorithms to maintain the connection between the elements in the tree for sibling relationships in XML data. Meanwhile, the method minimizes the spatial distribution effect on the queries and only increases the index tree size insignificantly compared to the original R-tree.

We designed the BioX-tree index tree structure, the R-tree family. The tree must strictly maintain the XPath sibling relationship between the elements so that sibling elements will not be arbitrarily scattered across the index tree. Specifically, the leaf nodes of the BioX-tree will contain only siblings of the same parent element in the original XML data. The sibling elements in a leaf node will be arranged in the correct order from front to back, consecutively. If there are multiple leaf nodes containing siblings of the same parent, each node will have two pointers pointing to the other leaf nodes in the order that contains the previous and the following siblings. As a result, the children of a parent element in the original XML data are centrally distributed over a set of leaf nodes that are connected by 2 pointers on each node. Thus, we can see that: (1) Adding 2 pointers per leaf node will not increase the node size significantly over the overall tree nodes; (2) BioX-tree will easily find all sibling elements only need any sibling element found previously on the BioX-tree. In the BioX-tree, each leaf element/entry equates to a tag name position in the XML data; each entry consists of 5 attributes {pre(E), post(E), par(E), att(E), tag(E)}; each node contains many entries and will have a size corresponding to 1 block on the hard drive. Similar to the R-tree, a non-leaf node has the form (pointer, MBR), where the pointer refers to the child node. Unlike the

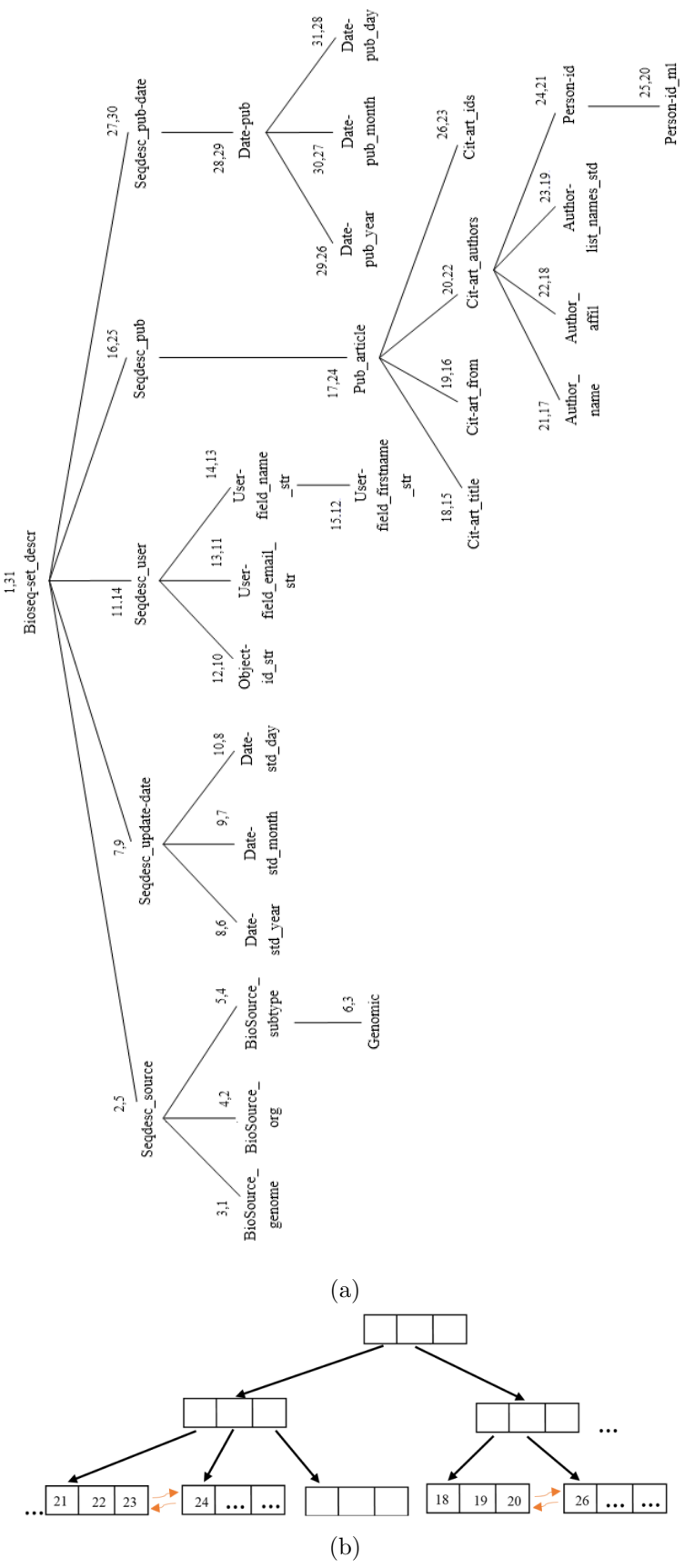

Figure 4: (a) Tree hierarchy of elements in rice DNA XML document. (b) leaf nodes are linked on the BioX-tree

R-tree, each leaf node has two pointers named the previous pointer and the next pointer to connect with other leaf nodes that contain sibling elements in the previous or following order.

For illustration, Figure 4(a) depicts the tree structure of an XML document containing rice DNA data (referred to as X document). They are numbered in pairs of pre- and post-tree traversal order. Figure 4(a) notes these two-value pairs on each tag name, respectively pre-post. For example, the root tag will be noted as 1–31, where 31 is the total number of X's tag names. After indexing the transformation data (for identification, we use 1 traversal value, pre-order, for each tag element), those elements will be stored in the leaf nodes of the BioX tree, as shown in Figure 4(b).

Specifically, XML elements of the same origin (parent) will be stored in the same leaf node. In the case of a leaf node whose number of entries exceeds the limit, that node will be split into 2 and there will be 2 pointers connecting them to each other. This is to ensure a connection between sibling elements is maintained. The arrows in brown indicate the pointers that link a leaf node to its preceding and succeeding sibling nodes.

In this example, the siblings with pre-order tree traversal values of 21, 22, 23 are inserted into the same leaf node in the XML document. When this leaf node becomes full, element 24 will be stored in a new leaf node. Two pointers are connected back and forth between the two leaf nodes to maintain the connection of those sibling elements.

3.3. Spatial analysis of the BioX-tree

With the structural design of the BioX-tree as shown above, Figure 5 depicts the MBRs of the leaf nodes of a BioX-tree in space. We recognize that MBRs of ancestral or parent XML elements are always larger and also cover MBRs of descendants or descendants.

Figure 5 shows the leaf nodes of the BioX-tree in 2-dimensional space through the minimum bounding rectangles (MBR). Where, $R1$ is the MBR of the sub-tags of the tags (2;5) of Figure 4(a), likewise $R2$, $R3$ in Figure 5 are the MBRs of the sub-tags of the tags (7;9), (11;14) in Figure 4(a), and similarly for smaller MBRs. From the above observations, we propose theorems for the correlation between the MBRs (of the leaf nodes) of the BioX-tree with the subtree of the XML tag when represented in the BioX-tree space.

Theorem 1. *Suppose in an XML document, the T tag is the parent of the* $t_1, t_2, \ldots, t_n$ *(sibling) tags. Then, in the BioX-tree space, the MBRs that bound the sub-trees of* $t_1, t_2, \ldots, t_n$ *will always be completely separate (do not intersect).*

For instance, $R1, R2$, and $R3$ represent bounding rectangles of sub-trees that are consistently distinct and arranged from left to right within the space depicted in Figure 2. To establish this theorem, we can readily observe that the pre-order values of elements in the Minimum Bounding Rectangle (MBR) for the left sibling's sub-tree always have smaller values than the pre-order values of elements in the MBR for the right sibling's sub-tree. Conversely, with post-order values, the values on the left are consistently greater than those on the right. Consequently, the MBRs for sub-trees of sibling tags in the BioX-tree space of XML documents are always separate. Hence, Theorem 1 demonstrates that grouping sibling XML tags with the same parent into the same leaf node of the BioX-tree may not significantly optimize the R-tree structure for queries. This observation has been supported by experimental results presented in the paper on the BioX-tree. Our proof is based on

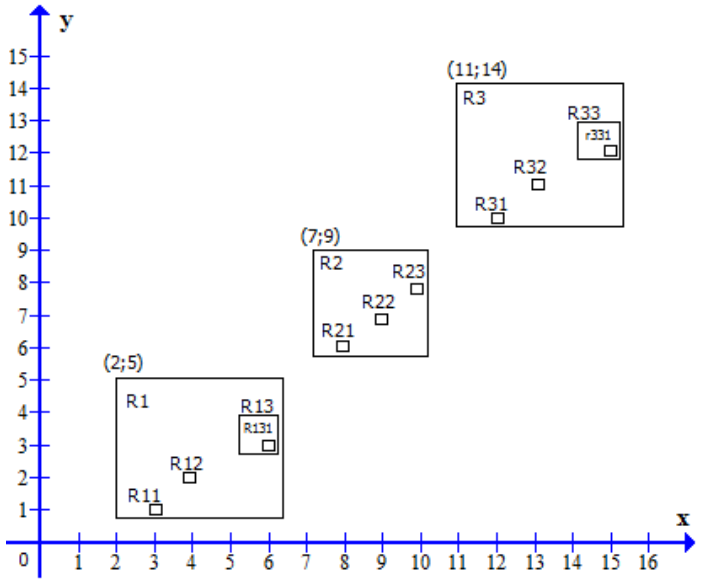

Figure 5: MBRs of the leaf nodes in the BioX-tree

the observation of the geometrical distribution of the rectangles and the numbering rule for traversing the tree. It can be explained more clearly by mathematical formulas, but we do not present it in this paper.

Theorem 2. *Assuming an XML document where the T tag serves as the parent of sibling tags $t_1, t_2, \ldots, t_n$. Within the BioX-tree space, excluding the MBRs of the subtree of the first sibling (t_1) and the last sibling (t_n), the MBR encompassing $t_1, t_2, \ldots, t_n$ will invariably enclose all the MBRs of the subtrees of $t_1, t_2, \ldots, t_{n-1}$.*

As an illustration, let's consider the case where R encompasses $R2$ as well as $R21, R22$, and $R23$. To establish this theorem, it becomes apparent that the pre-post values of t_1 and t_n are positioned before and after the pre-post value of T, respectively. Hence, the pre-post values within the subtrees of $t_1, t_2, \ldots, t_{n-1}$ evidently fall within the MBR range of both t_1 and t_n. Building upon Theorem 2, we can derive Consequence 3, which pertains to the query algorithm for searching an XML tag within the BioX-tree structure. Our proof is based on the observation of the geometrical distribution of the rectangles and the numbering rule for traversing the tree. It can be explained more clearly by mathematical formulas, but we do not present it in this paper.

Consequence 3. *Assuming the process of locating a $t-tag$ within the BioX-tree involves the query algorithm identifying the pre-post value pair of t in both the MBRs of nodes $R1$ and $R2$. If $R1$ is contained within $R2$, the algorithm will no longer need to traverse another subtree whose MBR is also encompassed by $R2$ in order to find t. This is because it is guaranteed that t belongs to the subtree of $R1$.*

Based on the theorems and consequences presented, we have undertaken a redesign of the BioX-tree algorithms in order to enhance query optimization and address inherent structural limitations.

3.4. Proposed algorithms

The above analysis helps us propose new insert and query algorithms. The goal of new algorithms is to build BioX-tree up, increase XPath sibling query efficiency, and reduce redundant tree-traversing steps due to the disadvantages of BioX-tree structure. Our new algorithms are:

- Algorithm 1: INSERT(N, E) is used to insert a new element E into an existing index tree or during the creation of a new tree. Unlike the original algorithm, this algorithm will find (using Algorithm 2: FIND_SIBLING_NODE(N, E)) and insert into the leaf nodes that contain the siblings of that element. Moreover, this algorithm will consider whether the element is the first or the last sibling. If so, the nodes that contain that element will only have a maximum capacity of m (minimum) elements instead of M (maximum) elements. As a result, the size of the MBR bounding that node will be smaller and can reduce the possibility of intersecting with other MBRs of the tree. These changes are due to Theorem 1.
- Algorithm 3: FIND_NODE(N, E) is improved from the original algorithm to find elements in the tree. Unlike the original algorithm, this algorithm avoids redundant searches in subtrees thanks to Theorem 2 and Consequence 3. Specifically, while a query looks for element E if a tree traversing step continues with nodes R and R inside $R1$, that query will skip the other subtrees of $R1$ and store the value R. At any internal node:
  - If there is R containing $R1$, subtree will be ignored.
  - If there is R in $R1$, save R to compare with the next step, continue to browse R.
- New XPath query algorithms including Algorithm 4: SIBLING_QUERY$(N, E,$ RESULT) and Algorithm 5: CHILDREN_QUERY$(N, Q,$ RESULT) are completely newly built because they will use pointers to search. As a result, the algorithm avoids most of the overlapping between the MBRs of the index tree. Other XPath query algorithms such as FOLLOWING_QUERY, PRECEDING_QUERY, ANCESTOR_QUERY are changed thanks to the improved FIND_NODE(N, E) algorithm. However, since these changes are not significant, they will not be covered in this paper.

Method SIBLING_QUERY complexity is calculated according to the number of nodes that must be retrieved. We have, the complexity for the query to find 1 element of the R-tree:

- In the best-case scenario, the time complexity for our algorithm is $O(\log_m N)$, where N represents the number of nodes in the tree, and m represents the number of entries in a node.
- The worst case is $O(N)$.
- The average case is $O(m \log_m N)$.

So the R-tree's SIBLING_QUERY is $x \times y$ window:

Algorithm 1 Insertion algorithm

```
1: procedure INSERT(N, E)
2:     Input: Node N containing entry E
3:     Output: Entry E inserted into the tree
4:     Begin
5:         SiblingNodes = FIND_SIBLING_NODE(N, E)
6:     if (SiblingNodes is found) then
7:         if (SiblingNodes is the first node or last node) then
8:             FullNodeCapacity = m
9:         else
10:            FullNodeCapacity = M
11:        end if
12:        if |SiblingNodes| < FullNodeCapacity then
13:            insert new context node E into SiblingNodes
14:        else
15:            CREATE_NEW_LEAF_NODE(E)
16:        end if
17:    else
18:        CREATE_NEW_LEAF_NODE(E)
19:    end if
20:    create pointers(pre, post, SiblingNodes)
21:    End
22: end procedure
```

- In the best-case scenario, the time complexity of the algorithm is $O(x \times y \times \log_m N)$, where N denotes the number of nodes in the tree, m represents the number of entries in a node, and x and y are additional factors contributing to the computation.

- The worst case is $O(x \times y \times N)$.

- The average case is $O(x \times y \times m \log_m N)$.

BioX-tree's sibling query has complexity:

- In the best-case scenario, the time complexity of the algorithm is $O(k+\log_m N)$, where N represents the number of nodes in the tree, m represents the number of entries in a node, and k denotes the number of siblings found in the query.

- The worst case is $O(k + N)$.

- The average case is $O(k + m \log_m N)$.

BioX-tree's query algorithm did not use the query window to find sibling elements.

Algorithm 2 Find sibling node algorithm

```
1: procedure FIND_SIBLING_NODE(N, E)
2:     Input: Current node N, entry E to find sibling.
3:     Output: Node N containing siblings of entry E
4:     Begin
5:     if (N is NOT a leaf) then
6:         for each entry E' in N whose MBR intersects with the MBR of entry E do
7:             FIND_SIBLING_NODE(N', E)
8:         end for
9:     else
10:        if (N contains an entry that is a sibling of E) then
11:            Return N
12:        end if
13:    end if
14:    End
15: end procedure
```

Algorithm 3 Find node algorithm

```
1: procedure FIND_NODE(N, E)
2:     Input: Current node N containing entry E
3:     Output: Node N containing entry E
4:     Begin
5:     min_N = N
6:     if (N is not a leaf) then
7:         for each entry E' of N whose MBR of N' intersects with the MBR of E do
8:             if (N' intersects or is inside min_N) then
9:                 Invoke FIND_NODE(N', E) where N' is the child node of N pointed to
   by E'
10:            end if
11:        end for
12:    else
13:        if (N contains an entry E) then
14:            Return N
15:        end if
16:    end if
17:    End
18: end procedure
```

Algorithm 4 Sibling query algorithm

```
1: procedure SIBLING_QUERY(N, E, RESULT)
2:     Input: Current node N (when starting, N will be the root node) and entry E to find siblings
3:     Output: List RESULT containing all entries that are siblings of entry E
4:     Begin
5:     Call FIND_NODE(N, E) to find node N' containing entry E
6:     if (node N' is not NULL) then
7:         Get entries E' in N'
8:         for each entry E'' in E' do
9:             insert E'' into RESULT
10:        end for
11:        if (following siblings pointer F is not NULL) then
12:            Call FOLLOWING_SIBLING_QUERY(N, F, RESULT)
13:        end if
14:        if (preceding siblings pointer P is not NULL) then
15:            Call PRECEDING_SIBLING_QUERY(N, P, RESULT)
16:        end if
17:    else
18:        Not found
19:    end if
20:    End
21: end procedure
```

Algorithm 5 Children query algorithm

```
1: procedure CHILDREN_QUERY(N, Q, RESULT)
2:     Input: Current node N (when starting, current node will be the root node), query window Q.
3:     Output: List RESULT containing all children of entry E
4:     Begin
5:     if (N is not a leaf) then
6:         Find entries E' in N with MBR intersects with MBR of Q
7:         for each entry E'' in E' do
8:             CHILDREN_QUERY(N', Q, RESULT) // where N' is a child node of N indicated by E''
9:         end for
10:    else
11:        Find entries E'' in N with MBR intersects with MBR of Q
12:        SIBLING_QUERY(N, E'', RESULT)
13:    end if
14:    End
15: end procedure
```

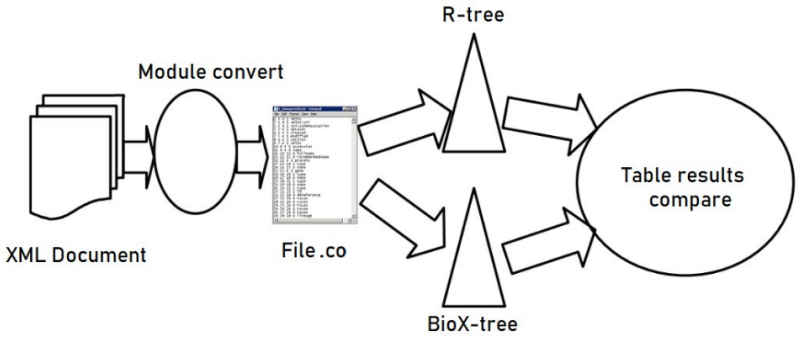

Figure 6: Model of experiment

4. EXPERIMENTS

4.1. Model and data of experiment

To test the proposed methods, we will choose large bioinformatics XML documents that contain typical data and convert them into multi-record tabular data. Each record will have five columns describing a tag with its spatial coordinates. We will evaluate the reduction in size due to the transformation and construct index trees using R-tree and BioX-tree methods. XPath query algorithms will be executed on these index trees to measure speed efficiency. The effectiveness achieved will be evaluated by comparing the results of multiple experiments, and the model of the experiment is shown in Figure 6.

To evaluate the practical effectiveness of the method, we selected bioinformatics XML documents that contain completely different structured and characteristic biological data. We utilized four bioinformatics datasets from reputable sources, covering different biological characteristics such as DNA, protein, and subspecies. The datasets are DNACorn (3.06 GB) and DNARice (15.8 GB) which provide information on corn and rice DNA, respectively, obtained from NCBI. Swissprot (3.9 GB) is a collection of functional information on proteins with precise, consistent, and comprehensive annotation from Uniprot, representing all protein descriptions. Allhomologies (1.26 GB) contain information on tree subspecies within the mouse family, obtained from Ensembl. We ran the test on a computer configuration CPU Intel Xeon E5520 - 2.7 GHz and RAM 20 GB, installed Windows Server 2016 Enterprise.

4.2. Experimental scenario

On each XML document, we randomly select 200 tag names and then execute XPath queries according to these tags, which also means finding tags with a familiar relationship with the selected tags. The queries are performed on a part of an XML document (XML small tree) of increasing size with the number of tag name/node of 20000, 40000, 60000 and 80000 tags, respectively. Due to the repetitive nature of the structure, we do not experiment on the entire XML document because unnecessary program runtime will be required. For each XPath query type, the average hard disk hits of 200 random queries per type will be used to evaluate the method's performance. In this experiment, we evaluate the algorithm speed based on the number of hard disk hits. Fewer hard disk hits mean higher query performance. This is a standard evaluation method for indexing data on a hard disk.

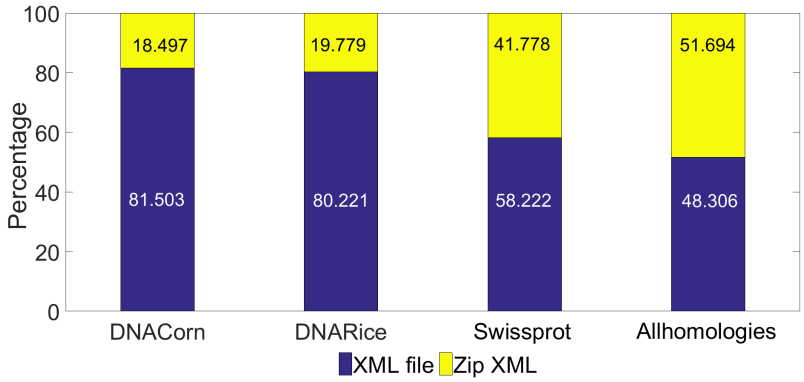

Figure 7: Compare XML data with data after transformation

4.3. Results of bioinformatics data compression

Figure 7 shows a quite good compression ratio, especially for documents describing DNA. However, the Allhomologies document describing subspecies information was quite surprising because of the low compression rate. The varying compression results of XML documents were analyzed, and it was determined that the cause was the Attribute tags. In particular, the Allhomologies document, which describes species information, has many Attribute tags. When a tag has numerous attributes, the entire content of the tag is represented as a long string, the conversion algorithm must separate each attribute into separate rows in the data table, thus increasing the size of the converted data. Hence, it becomes evident that the practical application of this transformation method does not consistently yield a favorable compression ratio due to its reliance on the XML document's structure. This paper primarily contributes to the exploration of the compression algorithm through the digital space conversion method. Further research endeavors will be dedicated to finding solutions to address this limitation.

4.4. Experimental results of queries

Hard disk indexing methods all evaluate the experimental efficiency of the algorithm based on the number of hard disk accesses because the hard disk access speed is much slower than the main memory. So, the speed efficiency for the indexing methods is mainly measured by the number of hard disk accesses. Time cost per clock will be considered when a method uses a lot of memory to improve computation efficiency. In this article, the proposed method completely does not use memory to improve efficiency. Therefore, the experiments still evaluate the speed efficiency according to the number of hard disk accesses, each access will take 1 block on the hard disk and put into memory equivalent to one node in the index tree.

Figures 8(a) and (b) show that the speed of the BioX-tree is much better than the R-tree. This result can be explained as follows, the R-tree method can only use range queries to find sibling or descendant elements. In contrast, the BioX-tree will use point queries to approach any sibling or child element and then pull out all the siblings of that element thanks to the connection pointers between leaf nodes of the BioX-tree.

The BioX-tree has made sibling queries significantly more efficient when compared to

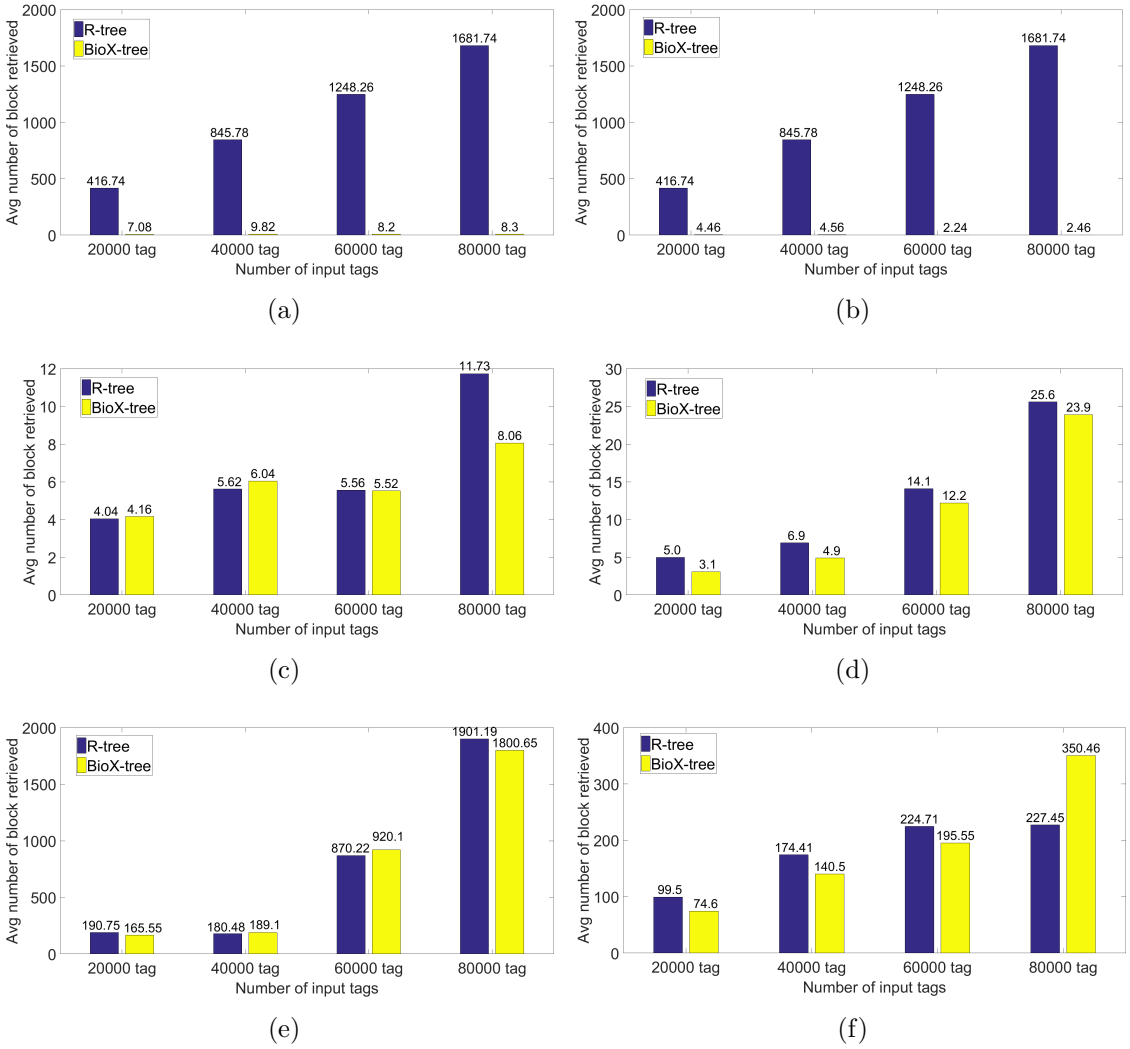

Figure 8: Compare (a) sibling query, (b) children query, (c) ancestor query, (d) descendant query, (e) following query, (f) preceding query between the BioX-tree and the R-tree

the R-tree thanks to its improved design. However, we also want to experiment with (conventional) range queries for the BioX-tree to test and evaluate how the quality effect on the query, in general, is with this new structure. Typically, these queries use rectangular areas to scan and search the space of the index tree. In particular, for XPath queries that must find all ancestors or all descendants, the proceeding/following elements of a given element, they use a quarter of the plane to search. The performance results of these queries will best reflect redundant multi-step problems resulting from the overlap and lead to a slowdown in search speed.

Figure 8(c) shows the performance of the BioX-tree slightly worse than the R-tree in some cases with distinct size data. The reason is that we have forced XML sibling elements into leaf nodes according to the BioX-tree rule. Possibly, that makes the index structure less optimal than the R-tree in some cases, leading to an unexpected overlap problem. However, due to the improvement of the query algorithm, the performance of the BioX-tree is not

much worse than the R-tree, sometimes even better.

Figure 8(d) shows the performance of the BioX-tree slightly better than the R-tree. The space containing descendant elements has a much greater density than the space containing ancestor elements, so if regular range queries are used, the BioX-tree will potentially have poor performance. Therefore, the BioX-tree finds only the children of each child element and then uses the pointers to obtain their sibling element.

Similarly, Figure 8(c), 8(e), 8(f) show the performance of the BioX-tree is slightly less expensive than the R-tree, and they both cost hard disk access. The reason is that a range query is forced to scan one of the four large regions of space, resulting in severe overlap. In the next study, we will have to find a solution to better solve this problem.

In conclusion, the method we showed was much better speed with sibling queries. With range queries, our method has kept almost the same as the original method. In short, the new method's sibling queries are much faster than that of the R-tree method. Other queries that take advantage of the sibling query's algorithm also provide better performance, like querying children for an element, because the children query is essentially sibling queries for the element's children. Similarly, descendant queries for an element are also sibling queries for the children and grandchildren of that element. However, because the number of descendant elements is so large that they are distributed randomly on disk space, the result is that the search by pointers is not very efficient. Finally, other queries that cannot take advantage of the sibling query algorithm, such as ancestor, following, preceding, must use the traditional R-tree query algorithm. Therefore, the query efficiency is highly dependent on the overlapping between the MBRs on the index tree. Naturally, the BioX trees have more overlapping than the R-trees. But fortunately, thanks to the improved FindNode algorithm, the BioX's query is approximately as efficient as the R-tree.

5. CONCLUSION

With the aim of improving the indexing method so that the information of large bioinformatics XML documents can be retrieved efficiently and at the same time can reduce the data size. We developed the BioX-tree indexing method to efficiently retrieve information from large bioinformatics XML documents while reducing data size. We converted the documents into tabular data with tag and coordinate information and improved the R-tree method for faster XPath queries, particularly sibling queries. We also analyzed the advantages and disadvantages of the proposed method with the original R-tree. Based on the correlation between the XML data tree structure and the BioXtree structure, we have given some theorems and consequences to improve several query algorithms, thereby overcoming the disadvantages of the BioX-tree. Experimental results show that the BioX-tree is more efficient for sibling XPath queries than the original R-tree method, while retaining effectiveness for other queries. Our focus on bioinformatics data is due to their large and diverse XML documents. The bioinformatics XML documents used in this paper have come from reputable sources, so we believe that the experimental results are objective and useful in real practice. The proposed method can be applied to upgrade other R-tree families with specific goals. In addition, our research is ignoring biological data in XML documents. The biological data is much larger than the structural data, and the queries for them are also very specialized. In the future, we will offer a more comprehensive solution for indexing entire XML documents.

REFERENCES

[1] S. Bog, et al., "XQuery 1.0: An XML query language (Second Edition)," *W3C Recommendation*, 2010.

[2] T. Bray, et al., "Extensible Markup Language (XML) 1.0 (Fifth Edition)," *W3C Recommendation*, 2008.

[3] J. Tatemura, "XML stream processing: Stack-based algorithms," *Advanced Applications and Structures in XML Processing: Label Streams, Semantics Utilization and Data Query Technologies (IGI Global)*, L. Changqing, L. Tok (Eds.), pp. 184–226, January 2010.

[4] S. Chen, H.G. Li, J. Tatemura, W.-P. Hsiung, D. Agrawal, and K.S. Candan, "Twig2stack: Bottom-up processing of generalized-tree-pattern queries over XML documents," *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB'06, VLDB Endowment, pp. 283-294, January 2006.

[5] J. Lu, T.W. Ling, Z. Bao, and C. Wang, "Extended XML tree pattern matching: theories and algorithms," *IEEE Trans. Knowl. Data Eng*, vol. 23, no. 3, pp. 402–416, August 2010.

[6] A.D. Baxevanis and B.F.F. Ouellette, *Bioinformatics: A Practical Guide to the Analysis of Genes and Proteins.* Third edition (Wiley), ISBN 0-471-478784, 2005.

[7] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg, "Ultrafast and memory-efficient alignment of short dna sequences to the human genome," *Genome Biology*, vol. 10, April 2009.

[8] H. Li and R. Durbin, "Fast and accurate short read alignment with burrows-wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754-1760, July 2009.

[9] N.S. Alghamdi, W. Rahayu, and E. Pardede, "Object-based semantic partitioning for XML twig query optimization," *Proceedings of the 2013 IEEE International Conference on Advanced Information Networking and Applications, AINA'13,* (IEEE Computer Society), Barcelona, Spain, pp. 61–70, June 2013.

[10] S. Haw and C. Lee, "Stack-based pattern matching algorithm for XML query processing," *J. Digit. Inf. Manage*, vol 5, no. 3, pp. 167-175, June 2007.

[11] E. Jiao, T. Ling, and C.Y. Chan, "Pathstack: A holistic path join algorithm for path query with not-predicates on XML data," *Database Systems for Advanced Applications,* Springer, vol. 3453, pp. 113–124, 2005.

[12] P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner, "MonetDB/XQuery: A fast XQuery processor powered by a relational engine," *SIGMOD*, pp. 479–490, June 2006.

[13] M. Kay, "Ten reasons why Saxon XQuery is fast," *IEEE Data Eng, Bull,* vol. 31, no. 4, pp. 65–74, January 2008.

[14] J. Siren, "Compressed suffix arrays for massive data," *SPIDE*, pp. 63–74, August 2009.

[15] H. Bjorklund, W. Gelade, M. Marquardt, and W. Martens, "Incremental XPath evaluation," *ICDT*, pp. 162–173, October 2009.

[16] N.S. Alghamdi, W. Rahayu, and E. Pardede, "Semantic-based Structural and Content indexing for the efficient retrieval of queries over large XML data repositories," *Journal of Future Generation Computer Systems*, vol. 37, pp. 212-231, July. 2014.

[17] B. Salzberg and V.J. Tsotras, "A comparison of access methods for time-evolving data," *ACM Computing Surveys*, vol. 31, no. 2, pp. 158-221, 1999.

[18] G. Li, J. Feng, J. Wang, and L. Zhou, "Effective keyword search for valuable lcas over XML documents," *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management (ACM)*, pp. 31–40, November 2007.

[19] Z. Liu and Y. Cher, "Reasoning and identifying relevant matches: For XML keyword search," *Proc. VLDB Endowment,* vol. 1, no. 1, pp. 921–932, August 2008.

[20] Z. Bao, T. Ling, B. Chen, and J. Lu, "Effective XML keyword search with relevance oriented ranking," *IEEE 25th International Conference on Data Engineering, (IEEE)*, pp. 517–528, April 2009.

[21] N.S. Alghamdi, W. Rahayu, and E. Pardede, "Semantic-based construction of content and structure XML index," *The 24th Australasian Database Conference (ADC), ADC'13, Adelaide*, vol. 137, pp. 61-70, January 2013.

[22] Z. Liu and Y. Chen, "Identifying meaningful return information for XML keyword search," *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data (ACM)*, pp. 329–340, June 2007.

[23] K. Sadakane and G. Navarro, "Fully-functional static and dynamic succinct trees," *ACM Transactions on Algorithms*, vol. 10, no. 16, pp. 1-39, May 2014.

[24] Hoang Do Thanh Tung and Dinh Duc Luong, "An improved indexing method for XPath queries," *Indian Journal of Science and Technology*, vol. 9, no. 31, pp. 1-7, 2016. DOI: 10.17485/ijst/2016/v9i31/92731

[25] Guttman, "R-Trees: A dynamic index structure for spatial searching," *Proceedings of SIGMOD (Boston, Massachusetts)*, vol. 14, no. 2, pp. 47–57, June. 1984.

[26] V. M¨akinen and G. Navarro, "Dynamic entropy-compressed sequences and full-text indexes," *ACM TALG*, vol.4, no. 3, pp. 306-317, July 2008.

[27] P. Ferragina, G. Manzini, V. M¨akinen, and G. Navarro, "Compressed representations of sequences and full-text indexes," *ACM Transactions on Algorithms*, vol. 3, no. 2, pp. 20-es, May 2007.

[28] T. W. Lam, W. K. Sung, S. L. Tam, C. K. Wong, and S. M. Yiu, "Compressed indexing and local alignment of DNA," *Bioinformatics*, vol. 24, no. 6, pp. 791–797, January 2008.

[29] P.M. Tolani and J.R. Haritsa, "XGRIND: A query-friendly XML compressor," *IEEE 18th International Conference on Data Engineering (IEEE)*, pp. 225-234, August. 2002.

[30] G. Navarro and V. M¨akinen, "Compressed full-text indexes," *ACM Comp. Surv*, vol. 39, no. 1 2-es, April 2007.

[31] H.L. Chan, W.K. Hon, T.W. Lam, and K. Sadakane, "Compressed indexes for dynamic text collections," *ACM TALG*, vol. 3, no. 2, pp. 21-es, May 2007.

[32] J.K. Min, M.J. Park, and C.W. Chung, "XPRESS: A queriable compression for XML data," *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data,* ACM, San Diego, California, pp. 122-133, June 2003.

[33] J. Cheng and W. Ng, "XQZip: Querying compressed XML using structural indexing," *International Conference on Extending Data Base Technology (EDBT)*, pp. 219-236, 2004.

[34] A. Arion, A. Bonifati, I. Manolescu, and A. Pugliese, "XQueC: A query-conscious compressed XML database," *ACM Trans. Internet Technol*, vol. 7, no. 2, pp. 1-35, May 2007.

[35] D. Arroyuelo, F. Claude, S. Maneth, V. M¨Akinen, G. Navarro, K. Nguyen, J. Sir En, and N. V Alim Aki, "Fast in-memory XPath search using compressed indexes," *Software: Practice and Experience (Wiley)*, vol45, no. 3, pp. 399-434, March 2015.

[36] B. Qian, H. Wang, J. Li, H. Gao, Z. Bao, Y. Gao, Y. Gu, L. Guo, Y. Li, J. Lu, Z. Ren, C. Wang, and X. Zhang, "Path-based XML stream compression with XPath query support web-age," *Information Management*, Springer Berlin, Heidelberg, pp. 329-339, 2012.

[37] P. Diet, "Maintaining order in a linked list," *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing (ACM)*, pp. 122–127, 1984.

[38] Q. Li, B. Moon, et al., "Indexing and querying XML data for regular path expressions," *Proceedings of the International Conference on Very Large Data Bases*, pp. 361–370, September 2001.

[39] H. Jiang, H. Lu, W. Wang, and B. C. Ooi, "XR-Tree: Indexing XML data for efficient structural joins," *Proc. the 19th International Conference on Data Engineering (ICDE)*, pp. 253-263, March 2003.

[40] Yaokai Feng and Akifumi Makinouchi, "A new structure for accelerating XPath location steps," *IAENG International Journal of Computer Science*, pp. 49-60, 2006.

[41] T. Grust, M. V. Keulen, and J. Teubner, "Accelerating XPath evaluation in any RDBMS," *ACM Transactions on Database Systems*, vol.29, no. 1, pp. 91-131, 2005.

[42] S. Haw and C. Lee, "Data storage practices and query processing in XML databases: A survey," *International Journal of Knowledge-Based Systems*, vol. 24, no. 8, pp. 1317-1340, 2011.

[43] X.L. Dong and D. Srivastava, "XML indexing," *Encyclopedia of Database Systems*. L. Liu, M.T. Özsu (eds), Springer, New York, NY. 2018, https://doi.org/10.1007/978-1-4614-8265-9-779

Received on September 28, 2023
Accepted on November 27, 2023