Data storage practices and query processing in XML databases: A survey
Su-Cheng Haw, Chien-Sing Lee
Faculty of Information Technology, Multimedia University, 63100 Cyberjaya, Malaysia

ABSTRACT
With the rapid emergence of XML as a data exchange standard over the Web, storing and querying XML data have become critical issues. The two main approaches to storing XML data are (1) to employ traditional storage such as relational database, object-oriented database and so on, and (2) to create an XML-specific native storage. The storage representation affects the efficiency of query processing. In this paper, firstly, we review the two approaches for storing XML data. Secondly, we review various query optimization techniques such as indexing, labeling and join algorithms to enhance query processing in both approaches. Next, we suggest an indexing classification scheme and discuss some of the current trends in indexing methods, which indicate a clear shift towards hybrid indexing.

1. Introduction
As the amount of data grows exponentially via the Internet, web applications such as search engines, electronic publishing, web services, e-business and e-learning portals are starting to use XML as the de facto standard for data exchange and data transfer. Besides being flexible, neutral, self-describing and extensible in nature, the structure of XML may vary from a flat regular data-centric structure to a deep irregular document-centric structure. The wide range of possible structures further popularizes XML as the most prominent representation for all kinds of data. This creates several challenges to the document and database communities as the need for effective storage and query processing becomes critical.

Being semi-structured data, there are two main approaches to storing XML documents: (1) using XML-enabled database (XED) such as relational database, object-oriented database and so on, and (2) using a native XML database (NXD). Although, recently, another storage known as hybrid storage has emerged, this approach merely combines the XED, in particular, the relational database and NXD solutions; allowing some parts of the structured XML to be mapped into relational data while the other parts can be stored in XML data type itself.

Using the XED approach, querying the XML document is subject to the query engine in the underlying storage. Using the native approach, there are two possible ways for structural query processing, namely: (1) traversing the XML database sequentially to find the matching pattern and (2) query processing using the decomposition-matching–merging approach.

To efficiently query XML documents, an efficient indexing structure with some query processing algorithms are necessary. In addition, an efficient labeling scheme provides quick identification of relationships within nodes in the tree structure. Hence, these motivate us to focus on reviewing major techniques based on indexing, labeling and some query processing technologies. This paper aims to provide an overview of XML databases and the related technologies to new researchers venturing into this area.

Our paper is structured as follows, Section 2 presents the preliminaries on the XML data model, XML queries and the XML storage model. Section 3 first elaborates on the mapping approaches to shred XML into relational database, followed by query performance for each mapping approach and subsequently suggests some combination of indexes to enhance query evaluation. Section 4 classifies the indexing techniques and discusses the utilization of indexes to enhance query performance in an NXD. Lastly, Section 5 concludes the paper.

2. Preliminaries
Fig. 1 shows an XML document, PublicationList, which will be used as an example throughout this paper.

2.1. XML data model
An XML document can be modeled as a rooted, nested, ordered node-labeled or edge-labeled data tree. These two models are in
fact, equivalent except for the placement of the labels; labels are on
the nodes in the node-labeled model and on the edges in the edge-
labeled model. Nevertheless, most research papers use the node-
labeled model. The edge-labeled model is used in some scenarios,
for example, as the representation of most structural indexes and
when shredding XML data into relational storage using the Edge
approach (to be introduced in Section 3.2). Fig. 2 shows examples
of the edge-labeled data tree of the PublicationList XML.

2.2. XML query

Much research has been done on XML query languages. Exam-
pies of prior works on XML query languages are Quilt [1], XPath
[2], XQuery [3] and XML query algebras such as XAL [4], YATL [5]
and Lore [6]. XML algebra typically provides a solid ground to de-
finite the semantics of a query language by its power of expression.
On the other hand, XML query languages provide the means to ex-
tact and manipulate data from XML documents. Although most of
the query languages differ in detailed grammars and representa-
tion, they share a common feature, that is, queries usually make
use of path expression for query evaluation [7].

Since XML is semi-structured data, there are typically two types
of user queries, namely full-text queries (keyword-based search)
and structural queries (complex queries specified in tree-like
structure) as depicted in Fig. 3. A keyword search is somehow sim-
ilar to content retrieval in information retrieval technology. Con-
versely, a structural search is to retrieve matches on the tree
where it has the tags and structure (relationship) specified in
the query criteria. Structural queries can be classified further into
path query and twig query. For each type of query, it may consist
of only single type parent–child (P–C) relationship, single type ancestor–
descendant (A–D) relationship or mixed types of both relation
ships.

Path query defines query on one single element (consists of only
one leaf node) at a time while twig query defines query on two or
more elements (consists of two or more leaf nodes). Thus, they are
also known as Simple Path Expression and Branching Path Expres-
sion respectively. In both cases, query nodes may be elements,
attributes or text. However, query edges for path query are either
P–C or A–D relationships, whereas query edges for twig query
may be P–C, A–D or sibling (preceeding and following), which also
determine the ordering of the relationships. In XPath [2] notation,
P–C relationship is denoted by ‘/’ while A–D relationship is de-
noted by ‘//’. A complex query, however, is a twig query that con-
ists of at least a branching edge and may contain many edges
formed by the basic P–C or A–D relationships.

2.3. XML storage

The first approach to storing XML documents is to employ tra-
ditional databases such as relational database or object-oriented
database as the underlying storage. The second is to develop a spe-
cialized system, which is known as native storage. Recently, there
is another approach, namely hybrid storage, which allows data to
be stored using mapping to relations and also allows storage of
XML sub-trees in its native format [8]. The underlying storage rep-
resentation has a significant impact on the efficiency of query pro-
cessing. Basically, a storage strategy can be defined as efficient if
the system manages to retrieve data accurately, use storage re-
sources competently and update data and schema correctly.

2.3.1. XML-enabled databases (XED)

XEDs are the databases that store the XML data in other formats
such as tabular data, spreadsheet, object and so on, other than the
XML format itself. Yet, these technologies usually provide a means
to shred XML into their underlying format. Using the relational
database as the storage, it provides maturity, scalability, portability
and stability [9]. On the other hand, object-oriented databases use class inheritance to govern the data integrity and to support complex relationships. There are also some approaches, which use object-relational database to combine both relational and object-oriented technologies. Although XEDs are advantageous in terms of providing the database features (such as scalability, concurrency control, recovery service and so on), these approaches are vulnerable especially when evaluating query of a large dataset.

Relational databases. Most enterprise today have long secured the use of relational databases for high-end transaction processing system. They have spent large investments amounting to trillions of dollars. As such, simply replacing relational databases with a pure XML database is not a good choice. Examples of some XML storage models based on relational databases are STORED [10], XRel [11] and XISS/R [12]. Since there are mismatches between the XML-structured data and relational data, mapping plays an important role in providing seamless integration between these database infrastructures.

Object-oriented databases. There are only minor discussions and publications based on object-oriented databases. This may be due to several reasons as follows [13]. First, applying object-oriented database techniques is less difficult compared to relational technology. Second, any publication on this subject would have to disclose the proprietary technical details. Finally, the absence of a standard object-oriented query language has hindered the development of tests to compare these implementations. ORA-SS [14], BDViedo3 [15] and Sem-ODB [16] are some examples of XML databases using object-oriented database as the underlying storage.

Since relational databases drive most businesses of any size today, the focus of this paper is using relational databases as an example of XED (see Section 3).

2.3.2. Native XML databases

A native storage basically means building a specialized data manager that contains XML as its fundamental unit of its logical model. These data are stored and retrieved in their original structure, with no mapping process required. Nevertheless, the NXD requires a particular underlying physical storage model, which can be a custom database or any typical database model [17]. Using this approach may work best, especially on scalability, data retrieval and handling of huge amounts of data. Nevertheless, it is not suitable when integration between various heterogeneous XML documents is needed.

Basically, an NXD falls into two main categories: (1) document-based storage and (2) node-based storage. A document-based storage (also known as text-based) will store the entire XML document in text form and provide some database transaction support such as indexing, materialized view and so on in accessing the document. A simple strategy for this might store the document as a Binary Large Object (BLOB) or Character Large Object (CLOB) in a relational table or as a file in a file system and provide XML-aware indexes over the document. In contrast, a node-based (also known as model-based) storage models the XML document as the internal model such as Document Object Model (DOM) or Simple API for XML (SAX) from the document and stores this model. How the model is stored depends on the physical underlying database. For instance, storing the DOM to relational database might result in tables such as Entities, Values, Attributes, Levels, and so on. TIMBER [18], XBase [19] and Natix [20] are some examples of native storage.

2.3.3. Hybrid storage

Recently, commercial RDBMS vendors such as IBM DB2 9 (formerly code-named “Viper”) [21], Microsoft SQL Server 2005 [22], Oracle 9i and 10g [23] and so on have shown significant interest in providing support for XML data management. These hybrid systems have the added advantage of being capable of storing both relational (structured) and XML (semi-structured) data, thus, allowing applications to access a single data repository. The main objective of such systems is to preserve the benefits associated with commercial RDBMS offerings including high levels of reliability, availability, security, concurrency and customer support while making it easy to manage and integrate existing corporate data with data modeled in hierarchical XML structures.

Fig. 4 depicts the general hybrid storage architecture. By using hybrid storage, the user is not limited to either native or relational storage alone. The user may choose to store different parts of the XML based on the required level of granularity [23]. For example, if we want to query the top-level elements of PublicationList such as book, title, section, journal and so on, these elements can be stored in relational tables, while the section details may be stored as XML datatype. With this, it gives the flexibility of storing useful and queryable information in relational tables while not decomposing the entire XML document. Henceforth, it saves the time and speed needed in reconstructing the document as well as the speed to load data [24].

Being hybrid storage, users can search both XML data and relational data types with structured query language (SQL) or XQuery [3] and also combine SQL with XQuery (see Fig. 4). In addition, to efficiently process queries of XML data, most of these commercial RDBMSs leverage cost-based query optimization technology to evaluate different data access strategies and select a low-cost option. Zhang et al. [25] have proposed a new operation XRjoin, which joins the XML data with relational data.

Prototype systems such as MARS [27], ReXSA [28] and work done by Mlynkova [29] have been proposed to support the hybrid storage. In MARS [27], some information is redundantly stored to improve the performance of query and data retrieval. They also propose some query rewriting using views. In ReXSA [28], Moro et al. address the problem to decide which portion of the data to persist as XML and which portion as relational data by considering the qualitative properties such as reuse, evolution and performance profiles. Mlynkova [29] propose an adaptive mapping strategy which is able to determine which mapping strategies suited most based on the sample set of XML and query workload.

Lim et al. [30] propose a space-efficient and query-friendly hybrid model by eliminating some storage redundancy while showing that query performance will not be affected. In fact, query performance can be improved based on their proposed query transformation algorithm. On the other hand, Abdel Kader et al. [31] perform an experimental analysis on the hybrid storage on two based models, i.e., the (i) structural mapping, and (ii) the XML data type, and varying the ratio of semi-structured to structured data with respect to vertical (tabular) and horizontal (data instances) dimensions.
3. Relational database as instance storage and its query processing

The major challenge in using relational databases as instance storage is that one needs to resolve the conflict between the hierarchical nature of the XML data model and the two-level nature (row and column) of the relational data model. There are three alternatives for doing so: (1) the relational schema-based approach, (2) the Edge approach and (3) the Attribute approach. The first strategy produces a relational schema from a Data Type Definition (DTD) or XML schema (XSD). Using the second strategy, instead of producing relational tables for each XML document, several XML elements and attributes are combined into a single table named Edge. Applying the third strategy is to perform a horizontal partitioning of the Edge approach. Thus, tuples with distinct tags are stored in separate tables.

Generally, if relational databases are used as instance storage, query optimization is quite restricted. Firstly, there is semantic mismatch between the XML query language (such as XQuery) and SQL. Secondly, such SQL statements are often complex and difficult to develop and tune; ultimately leading to performance problems.

3.1. The relational schema-based approach

The Relational schema-based approach [32,33] involves a shared-inlining method [34]. In this approach, a DTD or XSD is necessary to show the relationship between elements in the hierarchical structure. Nevertheless, in general, DTDs can be complex. Shanmugasundaram et al. [9] propose three transformation strategies to simplify the details of a DTD and yet generate a relational schema out of it. These methods are (a) flattering transformation, which convert recursive definition into a flat representation, (b) simplification transformation, which simplifies the unary operators into a single one and (c) grouping transformation, which groups all the sub-elements with the same tag.

The shared-inlining method ensures that any single element node is represented in exactly one relation. Fig. 5(a) shows the DTD graph based on the PublicationList XML data. From the DTD graph, we need to transform it into relational schema, by deciding what kind of relations to create. Relations are created for all elements in the DTD graph whose nodes have an in-degree greater than one. Nodes with an in-degree of one are kept inlined. In contrast, all sub-elements with one-to-many relationships cannot be inlined and have to be stored in separate tables. An ID is assigned to each tuple in a table and a ParentID column is included to identify its parent. If an element can appear at most once in its parent, it is inlined as one of the columns in its parent’s table. Fig. 5(b) shows the transformed relational schema from the DTD. After the shredding process of DTD into relational tables, queries over XML data can be easily transformed into SQL queries. For example, consider the following sample query in XPath notation [2], Q1 : /book/title //text = ‘data model’. This query wants to retrieve all book title, which contains the text ‘data model’. In relational database, the following SQL statement can evaluate this query:

```sql
SELECT b.title
FROM book b, section s, chapter c
WHERE s.text = ‘data model’ AND
   b.ID = c.ParentID AND
   c.ID = s.ParentID
```
To enhance query processing, index can be built on the (ID, ParentID) columns. Depending on the complexity of the query, by building index on these columns, query retrieval can be two to many times faster than without the index. Nevertheless, for simple query, performance is similar with or without indexes.

### 3.2. The Edge approach

In the Edge approach [35], the XML document is viewed as a tree and each edge of the tree is represented as a tuple in a single table. An ID is assigned to every element in the XML document. Every tuple in the Edge table consists of the source and destination IDs of the nodes, the Label that describes target element tag, Ordinal that displays the order of the sibling nodes, Flag which indicates whether the node is referencing to another node or text value and the data information (if the destination is a leaf node).

A node, X, can be joined to another node, Y, if and only if X.TargetID = Y.SourceID. Based on this property, it is easy to transform XML twig queries (support twig query with P–C relationship only) into SQL queries. For instance, consider the query, Q2: '/publications/book[/id]/title = 'XML Overview'. This query wants to retrieve the book id where the title is 'XML Overview'. The corresponding SQL statement can be written as below:

```
Select ID.Data
From Publications, Book, ID, Edge
Where Publications.SourceID = 0 and
     Book.SourceID = Publications.TargetID and
     ID.SourceID = Book.TargetID and
     Pub.Label = 'publications' and
     Book.Label = 'book' and
     ID.Label = 'id' and
     Publications.Title.Label = 'title' and
     ID.TargetID = Title.SourceID and
     Book.TargetID = Title.SourceID and
     ID.TargetID = ID.SourceID and
     Title.Data = 'XML Overview'
```

The main limitation of this approach is it requires multiple self-joins of the Edge table. Join is the most expensive evaluation in relational database. This will certainly cause performance degradation during query processing. In addition, the Edge table may grow very large and hence, it is not scalable to support large-scale XML datasets. Furthermore, this approach is unable to support twig queries with A–D edges, as it does not have enough information to trace from the source node to the target node.

To improve query performance, as suggested in [35], an index can be built on (Label, Data) to reduce the execution time of common queries, such as select * from Edge where title = 'XML Overview'. In Tian et al. [33], they identified that it is also very important to build indexes on (SourceID, Ordinal) and (TargetID) which is useful for traversing from a child node to its parent.

### 3.3. The Attribute approach

The Attribute approach [35] is defined as horizontal partitioning on the Label field of the Edge table. Clustering on the Label as index for the Edge table leads to the same effect. Thus, tuples with different Labels are grouped in different tables. This approach breaks the original Edge table into multiple relations based on the Label attribute.

Using this approach, the SQL statement to retrieve Q2 is as shown below:

```
Select ID.Data
From Publications, Book, ID, Title
Where Publications.SourceID = 0 and
     Publications.TargetID = Book.SourceID and
     Book.TargetID = ID.SourceID and
     Book.TargetID = Title.SourceID and
     Title.Data = 'XML Overview'
```

Similarly, to enhance query processing, index can be built on the (SourceID, TargetID) columns. Generally, this approach is good for XML documents with less distinct labels. However, if an XML document consists of large distinct labels, there will be multiple relations, which waste space and cause overhead in maintaining the database catalog. Similar to the Edge approach, this approach is unable to support twig queries with A–D relationships.

### 3.4. Summary and discussion

Newer mapping techniques revolve around the main objective, i.e., improving query response time. Khan and Rao [36] have proposed a technique for automatic mapping from an XML document to relational databases. They have demonstrated that their novel approach preserves the nested structure of the XML documents. Bohannon et al. [37] have proposed LegoDB, a physical database design automated wizard tool for designing relational storage structures for XML data, based on different workloads and data characteristics. Besides, LegoDB helps a user to evaluate some of the many mapping strategies of [9,19,33,38]. However, they consider only a fixed large set of physical schemas, each with a built-in mapping to the given XML schema. Recently, Atay et al. [39] have proposed (i) a lossless schema mapping algorithm to generate a database schema from a DTD, which makes several improvements over existing algorithms, and (ii) two linear data mapping algorithms to map ordered XML data into relational data. However, all prior works have adhered to decomposing into normalized relational schemas. Normalized schemas convert an XML document into a typically large number of tuples of different relations. Performance is definitely an issue if we query on some parts of the XML document resulting in a SQL query with large number of joins to retrieve and reconstruct all the necessary. Motivated by this problem, Balmin and Papatkonstantinou [40] have proposed denormalization on some schemas (involving inspection on a set of relational design anomalies; such as non-atomic values, functional dependencies and multi-valued dependencies) to reduce the number of joins. Although many research initiatives have examined how to map and store XML data in a relational database system, none of the papers have properly indicated which approach is best suited for which business environment. Since XML is so flexible and extensible, there is no solution, which can fit all requirements in different scenarios.

In our recent study [41], we evaluated the performance on various mapping approaches in terms of (i) database file size, (ii) database creation time and (iii) query retrieval time on various real and synthetic datasets to see which method outperform the rest under various types of dataset environments. Findings indicate that the Edge approach has the most modest storage size. However, the time taken to create such a database is tremendously slow for a large dataset. Secondly, in terms of query retrieval time, the Schema-based approach is the fastest. The Schema-based approach requires lesser number of joins as compared to the Edge and Attribute approaches. Besides, it has an added advantage as it could replace joins with selections. This will enhance the query processing cost. Third, with index utilization, both Schema-based and Attribute approaches improve query retrieval performance by about two to many folds of the response time. This is typically critical on a large dataset.
we conclude that the Schema-based approach is the best for most types of datasets, whether structured or semi-structured, simple query or complex query and medium to large file sizes.

4. Native database as instance storage and its query processing

Since data are stored and retrieved in their original hierarchical structure, the most general way to answer a query is to traverse in a depth-first search. Traditional path traversing are inefficient to answer several types of complex query (query with many joins and branches) and query with A–D relationships such as book//section. For example, to retrieve matches of such query, all downward paths starting from any book element is traversed down to its child one by one to check whether there exists any section element. This is done until every child (reaching until the leaf node) is evaluated. For the next set of matches, it needs to backtrack to its previous visited book element node and start the search again from the next child. Thus, this is certainly very exhaustive and inefficient. To overcome performance degradation due to excessive traversal, index structures (to be discussed in Section 4.1) have been introduced. These techniques certainly reduce the portion of the XML tree to be scanned during query processing.

Another query processing approach (see Section 4.2) involves joining indexes with some algorithms based on decomposition-matching–merging processes. Firstly, the twig query pattern (1) decomposes a set of binary relationships between each pair of nodes or into a set of path queries; (2) matches each binary component or path component of the query pattern against the XML database; and (3) merges–joins matched pairs or paths to obtain the final result.

4.1. Indexing

Generally, indexing is a well-known technology to improve the efficiency and scalability of query processing by reducing the search space. In XML technology, there are three main types of structural indexes, i.e., Path indexing, Node indexing and Sequence-based indexing. Fig. 6 depicts the structural classification for indexes. In this section, we will discuss the three types of indexing. We analyze how each approach works, as well as its advantages and disadvantages.

The outline of this section is as follows. Section 4.1 discusses the indexing techniques. Section 4.2 discusses query processing based on a decomposition-matching–merging approach.

4.1.1. Path indexing

The Path indexes can be further classified into Traditional indexing, Similarity-based indexing, Forward and Backward indexing and Advanced indexing.

**Traditional indexing.** This group of indexing generally creates a path summary from the XML database starting from the root to the respective node. As a result, the structural index is in general quite large as every database node is referenced within the index. Some examples of the traditional indexes are DataGuide [42], Index Family [43] and Index Fabric [44].

DataGuides [42] are general path indexes that summarize all paths in the source that start from the root. Each path in a DataGuide is unique. A single source may result in multiple DataGuides. Thus, it is important to decide what kind of DataGuide should be built and maintained. Intuitively, a minimal DataGuide might seem desirable, as it is most compact in size, with less traversal paths. Yet, a minimal DataGuide (as shown in Fig. 7(a)) is difficult to maintain especially during insertion of a new node. On the other hand, strong DataGuide (shown in Fig. 7(b)) is created based on the intention that the label path that reaches the same set of objects in the source is the same as the label path that reaches the same object in DataGuide. Since strong DataGuide is more common, henceforth, the term DataGuide refers to strong DataGuide.

Each node in a DataGuide has an extent for the corresponding nodes in the source. For example, the extent in the node p6 in Fig. 7(b) is the set of {7, 10}, where each element can be reached by the label path book.chapter. However, DataGuide is only applicable to queries with simple path expressions. That is, it is not feasible for multiple and branching path expressions. For example, consider the following query, Q3: book//section.

This query wants to retrieve all section of a book. If we traverse the DataGuide, the returned results are nodes p2 and p9 and the corresponding extents are [2, 3] and [13, 14] respectively. DataGuide does not preserve hierarchical relationships among individual nodes. Thus, there is no path information between nodes p2 and p9. As a result, traversing the DataGuide alone cannot return the result for query Q2. We need to traverse the original data source and DataGuide simultaneously.

DataGuide construction takes linear time for tree-structured data. However, it requires exponential time in the worst case for a graph-structured data. Furthermore, on a very irregularly structured data, DataGuide may be much larger than the original data source.

Index Fabric’s data structures are based on tries [44]. A trie is a tree that uses part of the key to navigate the search. Each key is a sequence of characters, and a trie is organized around these characters rather than the entire keys. Tries are searched from the root of the tree, and follow the edges that correspond to the characters of the search key. In Index Fabric, each data path is encoded using unique designators (special characters). In order to interpret these designators, designator dictionaries are designed to map between the element tags and designators. Fig. 8(a) depicts the designator dictionary, (b) root-to-leaf path encoding and (c) the Index Fabric.

---

**Fig. 6.** Structural indexes classification.
(a) Minimal DataGuide (b) Strong DataGuide.

Fig. 7. (a) Minimal DataGuide (b) Strong DataGuide.

<table>
<thead>
<tr>
<th>Tagname</th>
<th>Designator</th>
</tr>
</thead>
<tbody>
<tr>
<td>publications</td>
<td>P</td>
</tr>
<tr>
<td>book</td>
<td>B</td>
</tr>
<tr>
<td>journal</td>
<td>J</td>
</tr>
<tr>
<td>id</td>
<td>I</td>
</tr>
<tr>
<td>title</td>
<td>T</td>
</tr>
<tr>
<td>chapter</td>
<td>C</td>
</tr>
<tr>
<td>section</td>
<td>S</td>
</tr>
<tr>
<td>text</td>
<td>E</td>
</tr>
<tr>
<td>figure</td>
<td>F</td>
</tr>
<tr>
<td>table</td>
<td>A</td>
</tr>
<tr>
<td>caption</td>
<td>N</td>
</tr>
</tbody>
</table>

Fig. 8. (a) Designator dictionary (b) root-to-leaf path query and (c) Index fabric.
For example, based on the PublicationList XML database, publications is represented by P, B for book, C for chapter and so on. Thus, the path publications/book/chapter is encoded as PBC. However, a trie may grow very large. Thus, to enable incremental scaling to a larger numbers of keys, Index Fabric is usually based on Patricia Trie (PT). Insensitive to the length of inserted strings, PT is a simple form of compressed trie, which merges single child nodes with their parents. Nonetheless, PT can become large and unbalanced structures. This results in significant performance degradation. To balance the tries and support large XML, Cooper et al. [44] propose building multiple layers of tries, and conducting searches by proceeding from layer to layer. Index Fabric does not keep the information of XML elements which do not have data values. Thus, it is inefficient for processing partial matching queries, that is, queries that does not begin from the root of the document [45].

**Similarity-based indexing.** This group of indexing basically groups nodes in data trees based on local similarity, in order to reduce the size of the index tree. Besides, similarity-based indexing are adaptive indexes designed to support frequently used path query by adjusting its structure according to the query workload. Some examples of indexes in this category are adaptive path index (APEX) [45], A\((k)\)-index [46], D\((k)\)-index [47], M\((k)\)-index [48] and XMine [49].

Adaptive Path indEX (APEX) is a method to manage path indexes for XML data adaptively [45]. While the traditional indexes such as DataGuide, T-index, A\((k)\)-index and Index Fabric maintains all paths from the root, APEX does not keep all paths starting from the root. However, it utilizes only frequently used paths by applying sequential pattern mining technique. Furthermore, APEX is workload aware, i.e., it can be dynamically updated according to the changes in query workloads. APEX consists of two structures, a graph structure \((G_{APEX})\) to represent the structural summary of XML data and a hash tree structure \((H_{APEX})\) that consists of incoming label path to the respective nodes as shown in Fig. 9. Each label in the data graph is kept as a node in the \(H_{APEX}\) known as \(hnode\). Each \(hnode\) points to either a node of \(G_{APEX}\) or another hash table. Each node in \(G_{APEX}\) (known as \(xnode\)) maps to an entry of a \(hnode\) of \(H_{APEX}\). An extent, i.e., a set of edges in the data graph is assigned to each \(xnode\). The hash tree is used to find nodes of the structure graph for a given label path.

For example, assume that the following query Q4://book/title is the frequently used path. The query processor looks up the hash tree with the label path book.title in reverse order and retrieve \(xnode\&2\), which returns extents of \((2, 6)\) and \((3, 9)\). Hence, nodes 6 and 9 are the answers to the label path in query Q3. Let us look into another example. Let us suppose that we want to retrieve the answer for the query //section/text. Since the query path does not exist in the \(H_{APEX}\), it is cut into the sub-queries of //section and //text. Query processor looks up \(H_{APEX}\) for section and follows the entry to get \(xnode\&7\) in \(G_{APEX}\). The extents in \(xnode\&7\) are \((7, 13)\) and \((10, 14)\). Similarly, \(xnode\&8\) with the extent \((13, 15)\) is returned for text. The extents from both paths are then joined to produce the edge set \((7, 13, 15)\). Thus, only node 15 is returned as the answer.

Chen et al. [47] propose \(D(k)\)-index, a generalization of \(A(k)\)-index [46] based on the concept of bisimilarity. \(D(k)\)-index is an adaptive structural summary for general graph-structured data. However, it can be tuned to support frequently used path expressions by adjusting its structure according to the query workload. Each index node \(v\) in \(D(k)\)-index has two attributes: (1) \(k_v\) which denotes the local similarity requirement on \(v\), and (2) \(v\)-extent which is the set of data nodes associated with \(v\). The value of \(k_v\) can be obtained by mining the current query load. The choice of \(k_v\) should guarantee that the majority of queries accessing \(v\) are less than or equal to \(k_v\) in length. Thus, most queries on \(v\) can be directly performed on the index graph without the validation process as opposed to the \(A(k)\)-index. Furthermore, Chen et al. [47] also propose the PROMOTE algorithm to refine \(v\) by recursively promoting all of its parents and then partitioning extent to meet the new local similarity requirement. This certainly improves system performance.

\(D(k)\)-index construction begins with the label-split index graph, in which all index nodes are 0-bisimulation equivalence classes. Then, it proceeds to construct the 1-bisimulation equivalence classes. It repeats this process until the local similarity requirements of all index nodes are satisfied. It takes \(O(km)\) time in the worst case, in which \(k\) is the maximal local similarity requirement and \(m\) is the number of edges in the data graph. By varying \(k\) according to the query workload changes, \(D(k)\)-index can be flexible and of a smaller size as compared to \(A(k)\)-index. Nevertheless, \(D(k)\)-index has a major drawback; i.e., over-refinement of irrelevant index, data nodes and over-qualified parents as illustrated in Fig. 10. Suppose we need to increase the local similarity of the index node \(E\) from 0 to 1. The results after PROMOTE is node \(E\) has been split into two. However, their corresponding data nodes 6 and 7 are actually

---

**Fig. 9.** APEX index.
1-bisimilar, and therefore, should have stayed together in one index node.

**Forward and backward indexing.** This group of indexing "covers" the forward and backward path to support all queries for branching path expressions effectively. Nonetheless, their index sizes tend to be very huge. Among these are F&B [50], BCPI [51] and Multiple Depths Forward and Backward (MDFB) [52].

The F&B index [50] is a "covering index" that is generated using the notion of inverse edges to get structural summaries that capture information about in-coming and out-going paths. Thus, unlike all the other approaches mentioned earlier, it supports branching path expression queries. The F&B index is constructed based on the following principles (1) For every (edge) label \( l \), add a new label \( l' \); (2) For every edge \( e \) labeled \( l \) from node \( u \) to node \( v \), add an inverse edge \( e' \) with label \( l' \) from \( v \) to \( u \); (3) Compute the 1-Index or DataGuide on this modified graph.

Based on the PublicationList XML document, the F&B index constructed is somehow similar to strong DataGuide as in Fig. 7(b). Since the F&B index is an index that covers all branching path expression, all queries can be processed on the index itself. Let us take a look at Q5://book/chapter[/title]/section.

This query wants to retrieve all title and section elements under the path book/chapter. To process such query, we can think of these branching path expressions as a basic forward label path (also known as primary-path) with Boolean path conditions on intermediate labels. The primary-path is the path that remains when all parts between bracket '[' and ']' are removed. In this case, the primary-path is book/chapter/section. The corresponding node path (refer to Fig. 7(b)) is \( p_2/p_6/p_9 \). The other parts of the branching path expression act as path constraints on the primary-path. In this case, the nodes labeled chapter asserting to the condition they have the path /title as the out-going edges are retrieved. Only one node fulfills the criteria, i.e., \( p_8 \). Thus, the final answer to this query is the node path \( p_2/p_6/p_8/p_9 \).

The size of the F&B index is in practice quite huge and may approach the size of the base XML database itself. There is certainly very little performance gain as evaluating a query on this large index is just like evaluating the query on the base data. Kaushik et al. [51] propose BPCI, an index scheme, which restricts the class of queries that can be answered by eliminating branching path expressions that are deemed less important (such as those tags are of lesser interest, exploit the local similarity to cut down the index size and restricting the "tree-depth") to reduce the index size.

The attractive features about the BPCI is that one can choose to define the level of aggregation that one may desire by specifying the attributes and tags to be indexed, the tree-depth and the local similarity values. By relaxing the rules, one may get a more general index. Conversely, by adding in tighter constraints, the index sizes reduce but so does the class of queries that it can answer. Hence, there is a tradeoff between picking larger values for the parameters of the BPCI and performance of the index in answering queries.

**Advanced indexing.** There is a wide variety of XML indexing techniques done in the recent years. Nevertheless, we will not discuss these in detail, as most of them are still new, unclear or still evolving with no proper implementation yet.

Lian et al. [53] propose MIS index, an index structure, which index infrequent structures in the database. MIS indexes not only paths, but also arbitrary useful structures of high selectivity. Besides, it is space-efficient since its index size is small. Furthermore, since the MIS are infrequent, the index prunes large amounts of data faster. More recently, Lian et al. [54] explore further the incorporation of data mining into the SG-tree (proposed in [53]) algorithm.

Yan and Liang [55] propose MXI, an indexing method that supports efficient path query based on embedded single or multiple DTDs. Indexes are constructed for both DTDs and XML documents respectively based on certain coding scheme. Making use of the information in DTD, the query is matched against the DTD tree. MXI adopts the matching method of the XISS system [56], which decomposes the complex path expression to multiple simple path expressions.

Mohammad and Martin [57] propose LITIX, an index structure which combines the DataGuide path index and some level information of the XML nodes. The knowledge of the level is crucial to determine the nodes whose level indicates a potential answer for a given query, and at the same time, to eliminate the nodes that violate the element level order given by the query.

Jiang et al. [58] introduce GString, a novel sequencing method to capture the semantics of the underlying data graph. Meaningful components of the graph structure are located and used as the basic units in sequencing. This technique reduces the size of resulting sequences and also enables semantic-based searching.

Jin and Bao [59] propose an XML index structure BTP-Index, composed of XML structure index mechanism. The backbone is a Suffix tree, and XML content index mechanism is based on Tries & PT (similar to Index Fabric [44]), for the evaluation of keyword search query.

**Summary and discussion on path indexing.** In Section 4.1.1, various path indexing techniques have been presented. This section answers the following questions:-

1. What are the existing path indexing techniques?
2. How do these path indexing techniques work?
3. What are the advantages and disadvantages of these techniques?

**Table 1** summarizes the advantages and disadvantages of the path indexing techniques mentioned above.
Table 2 depicts each indexing technique support for both path query and twig query with (i) only P–C edges, (ii) only A–D edges and (iii) mixed relationships. This table provides an overall summary on the queries supported. The ‘+’ indicates that the features are being supported while the ‘−’ indicates that it is not supported. Nevertheless, this table does not provide enough information, as to how efficient the query processing is for each technique. For example, DataGuide and Index Family are efficient in supporting path query with single P–C edges, but unable to support query with A–D edges and twig query. The A(k)-index, D(k)-index and M(k)-index are only effective in supporting query of length less than k while APEX is only effective for short queries. On the other hand, F&B, BPCI and MDFB support all types of queries, but their index sizes are rather huge.

4.1.2. Node indexing

Node indexing approaches create indexes on each node by its positional information (based on labeling schemes) within the XML tree. Thus, this group of indexing is also known as labeling scheme, encoding scheme or numbering scheme. Unlike path indexes, node index schemes can determine the hierarchical relationships between a pair of nodes efficiently. Moreover, using nodes as the basic unit of query provides greater flexibility. Matching each node in the query tree and structurally joining these matches can process any tree structure query efficiently. Generally, they can be broadly categorized into four main categories; Subtree labeling, Prefix-based labeling, Multiplicative labeling and Hybrid labeling as shown in Fig. 6 earlier. Some of the methods in these categories are discussed in this section.

Subtree labeling. This category is the simplest; where the label of a given document node v in D encodes the position and the extent of the subtree Dv of D that is rooted in v, by means of offsets in the sequence of nodes resulting from traversing (at least part of) the document tree in a specific order. While the exact representation of the subtrees varies accordingly, for the given nodes v, w in D, their A–D and P–C relationships are always determined by testing whether Dv contains Dw. The label of a node is usually concise in this group of labeling scheme. Nevertheless, their performance degrades in an update intensive environment, as the labels usually need to be regenerated. The subtree labeling can be further broken-down into two subclasses: interval encoding and region encoding.

Interval encoding is the earliest labeling scheme proposed. Dietz [60] introduce the first numbering scheme based on tree traversal order on a tree data structure. Each node is labeled with a pair of unique integer consisting of preorder and postorder traversal sequences. His proposition is: for two given nodes v and w of a tree D, v is an ancestor of w if and only if v occurs before w in the preorder traversal of D and after w in the postorder traversal. By using this approach, we can determine the A–D relationship easily. Nevertheless, the P–C relationship could not be determined directly. Furthermore, this method is inefficient for a dynamic XML document. Whenever a new node is inserted or deleted, the preorder and postorder values need to be recomputed. To overcome the limitation of Dietz’s numbering scheme, Li and Moon [56] propose a numbering scheme based on (order, size). To enable future insertions gracefully, size (v) can be an arbitrary integer larger than the total number of current descendant of v. However, a global reordering is necessary when all the reserved spaces have been consumed. Moreover, it is not clear how one assigns a large enough value for “size”, based on the three propositions.

Other references such as Chien et al. [61] and Yu et al. [62] refer to this scheme as durable node numbering (DNN). Chien et al. [61] apply the scheme to multiversion document management. Their

Table 1

Summary of advantages and disadvantages of path indexing.

<table>
<thead>
<tr>
<th>Index Type</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Strong DataGuide (NXD)</td>
<td>Simplify traversal path. No insertion/update problem (set of extent can be extended).</td>
<td>Index graph size grows exponentially. The source and DataGuide may need to be traversed simultaneously in some cases. Does not support multiple and branching path expressions.</td>
</tr>
<tr>
<td>2. Index Family (NXD)</td>
<td>Simplify traversal path. No insertion/update problem (set of extent can be extended or created to support new node).</td>
<td>Index graph size may grow quadratically (2-index). Many templates must be built in advance (T-index). Does not support multiple and branching path expressions.</td>
</tr>
<tr>
<td>3. Index Fabric (NXD/XED)</td>
<td>Index size controllable as it can be partitioned horizontally and vertically. Support all types of queries due to its Patricia Trie structure.</td>
<td>Encoding keys may need to be regenerated. The designator dictionary to map data graph to index may grow very large.</td>
</tr>
<tr>
<td>4. APEX index (NXD)</td>
<td>No insertion/update problem. Supports partial queries path matching. Good for short queries.</td>
<td>Workload info needs to be collected using data mining approach to determine the frequently used queries. APEX cannot directly answer other path expressions of length more than one. Does not support multiple and branching path expressions.</td>
</tr>
<tr>
<td>5. A(k)-index (NXD)</td>
<td>Good for short queries. Size never grows larger than the source. No insertion/update problem (supports subgraph addition and incremental update).</td>
<td>Not accurate as only approximation is used to answer longer queries. Does not support multiple and branching path expressions.</td>
</tr>
<tr>
<td>6. D(k)-index (NXD)</td>
<td>Adaptable to query workload. Index size smaller compared to A(k)-index. No insertion/update problem (supporting subgraph and new edge adding).</td>
<td>Workload info needs to be collected to determine the frequently used queries. Over-refinement of irrelevant index, data nodes and over-qualified parent.</td>
</tr>
<tr>
<td>7. M(k)-index (NXD)</td>
<td>Adaptable to query workload. Index size much smaller compared to D(k)-index due to multiple resolution across different path of the data graph. Reduces over-refinement of irrelevant index and data nodes by grouping them together. No insertion/update problem (supporting subgraph and new edge adding).</td>
<td>Workload info needs to be collected to determine the frequently used queries. Does not support multiple and branching path expressions.</td>
</tr>
<tr>
<td>8. F&amp;B Index (NXD)</td>
<td>Supports all types of queries especially branching path efficiently.</td>
<td>Index size often tends to be too large. Does not support updating – index needs to be regenerated.</td>
</tr>
<tr>
<td>9. BPCI Index (NXD)</td>
<td>Supports all types of queries especially branching path efficiently. Index size much smaller – eliminating branching path expression that are less important based on the level of aggregation defined.</td>
<td>Does not support dynamic update – index needs to be regenerated.</td>
</tr>
<tr>
<td>10. MDFB (NXD)</td>
<td>Supports all types of queries especially branching path efficiently.</td>
<td>Does not support dynamic update – index needs to be regenerated.</td>
</tr>
</tbody>
</table>
Sparse preorder and Range (SPaR) labels are pairs \((\text{DNN}(v), \text{range}(v))\) consisting of a durable node number and a range, which serves the same purpose as the "size" described above. Let \(\text{DNN}(E)\) and \(\text{range}(E)\) denote the DNN and the range of a given element \(E\); then a node \(B\) is descendant of node \(A\) iff \(\text{DNN}(A) < \text{DNN}(B) < \text{DNN}(A) + \text{range}(A)\). When the elements in the document are updated, their SPaR numbers remain unchanged. When new elements are inserted, they are assigned a DNN and a range that do not interfere with the SPaR of their neighbors to maintain sparsity by keeping the intervals of nearby nodes as far apart as possible.

The region encoding (a.k.a. range encoding) schemes are originally designed for structured text databases. A node in the document tree corresponds to a substring of the entire string of the XML document. Such substrings can be naturally identified by region coordinate, which is interpreted as a pair of integers \((\text{docno}, \text{wordno}, \text{level})\) for \(E\)-index and \((\text{docno}, \text{wordno}, \text{level})\) for \(T\)-index. The position of a node in the document, which is denoted as \((\text{docno}, \text{begin: end}, \text{level})\) gives the starting and ending positions of the substring. Zhang et al. \[63\] propose using inverted lists to process query based on containment. Fig. 11 illustrates the structure of the text index \((T\text{-index})\) and element index \((E\text{-index})\) based on the sample PublicationList XML. Each inverted list records the occurrences of a word or an element known as term. Each occurrence is indexed by its document number, position and depth within the document, which is denoted as \((\text{docno}, \text{begin: end}, \text{level})\) for \(E\)-index and \((\text{docno}, \text{wordno}, \text{level})\) for \(T\)-index. The position \((\text{begin}, \text{end}, \text{wordno})\) of a term is given by counting word numbers in the XML document based on depth-first traversal.

To process the expression of "\(a/b\)" in the inverted list of "\(a\)" and "\(b\)" are retrieved. Occurrences of the two lists are merged if they satisfy the Containment Property (An occurrence of a term \(T_1\) encoded as \((D_1, P_1, L_1)\), contains an occurrence of a term \(T_2\), encoded as \((D_2, P_2, L_2)\), if and only if: (1) \(D_1 = D_2\), and (2) \(P_1\) nests \(P_2\)). Using the Direct Containment Property (An occurrence of a term \(T_1\), \((D_1, P_1, L_1)\), direct contains \(T_2\), \((D_2, P_2, L_2)\), if and only if: (1) \(D_1 = D_2\), (2) \(P_1\) nests \(P_2\) and (3) \(L_1 + 1 = L_2\), the expression "\(a/b\)" can be processed by merging the inverted list. Although this labeling scheme is very effective in determining the relationship among nodes, nevertheless, it assumes that once a position is assigned, it is never changed. This renders it unsuitable in handling XML with frequent updates.

**Prefix-based labeling.** This is the most diverse class of labeling schemes. In the prefix labeling scheme (a.k.a path-based labeling scheme), the label of a given node \(v\) encodes the nodes on the path from the document root down to \(v\) as a sequence to uniquely denote an ancestor of \(v\) on that path. Thus, given a node \(v\) and its ancestor, \(u\), their relationship could be determined precisely, i.e., \(u\) is an ancestor of \(v\) iff \(\text{label}(u)\) is a prefix of \(\text{label}(v)\). However, this labeling scheme has some limitations in terms of space consumption and efficiency. The size of the label grows with the length of the encoded path. In the worst case, its size is \(O(n)\). Hence, path encoding takes up more space as compared to subtree encodings, whose label size is \(O(\log n)\). This further affects the efficiency during the query evaluation process, as more time is needed to process the longer encoded path. Among some of the labeling scheme under this category are tree location address \[64\], simple prefix \[65\], Dewey ID \[66\], extended Dewey \[67\] and ORDPATH \[68\].

O’Neil et al. \[68\] propose a prefix-based labeling scheme called ORDPATH. ORDPATH is similar conceptually to the Dewey ID \[66\]. ORDPATH encodes the P–C relationship by extending the parent’s ORDPATH label with a component for the child. For instance, 1.5.3 is the parent, 1.5.3.9 the child. The main difference between ORDPATH and Dewey ID is that, even numbers are reserved for further node insertions in ORDPATH. An example of tree labeling using ORDPATH is depicted in Fig. 12. During the initial labeling, ORDPATH only assigns positive and odd integers. Even and negative integer values are reserved for later insertions. For instance, if the newly inserted node is to be added to the right of all the existing children, its label is generated by adding +2 to the last ordinal of the last child. However, if the newly inserted node is to be added to the left of all the existing children, its label is generated by adding -2 to the last ordinal of the first child. To insert a new node between any two siblings of a parent node \(u\), an even ordinal falling between the odd ordinals of the two siblings added with a new odd component is generated. For example, by providing the new siblings with the even caret 4, the sequence to fall between sibling nodes 1.3.3 and 1.3.5, are: 1.3.4, 1.3.4.1, 1.3.4.3, 1.3.4.5 and so on. The value 4 in component 1 (or any even value in any non-terminal component) represents a caret only, that is, it simply does not count for ancestry: 1.3.4.1 is a child of 1.3 and a grand child of 1. Nonetheless, the caret does have an effect on the ORDPATH order, since comparison 1.3.3 < 1.3.4.1, 1.3.4.3, 1.3.4.5 < 1.3.5 is true. New insertions can always be caretted in between any two existing sibling nodes. For example, we can insert a node between 1.3.3 and 1.3.4.1 by using 1.3.4.1. However, this approach is not suitable for deep trees. As can be seen in Fig. 12, the size of ORDPATH label may become quite huge, especially in

<table>
<thead>
<tr>
<th>Table 2</th>
<th>Query support for each path indexing technique.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path index</td>
<td>Path query</td>
</tr>
<tr>
<td>1. Strong DataGuide</td>
<td>✓</td>
</tr>
<tr>
<td>2. Index Family</td>
<td>✓</td>
</tr>
<tr>
<td>3. Index Fabric</td>
<td>✓</td>
</tr>
<tr>
<td>4. APLEX index</td>
<td>✓</td>
</tr>
<tr>
<td>5. A(k)-index</td>
<td>✓</td>
</tr>
<tr>
<td>6. D(k)-index</td>
<td>✓</td>
</tr>
<tr>
<td>7. M(k)-index</td>
<td>✓</td>
</tr>
<tr>
<td>8. R&amp;F index</td>
<td>✓</td>
</tr>
<tr>
<td>9. BPCI index</td>
<td>✓</td>
</tr>
<tr>
<td>10. MDFB</td>
<td>✓</td>
</tr>
</tbody>
</table>
trees with large depth and fan-out values. To cope with such trees, ORDPATH uses labels that do not reflect ancestry and thereby looses some of its expressivity.

Some variations of ORDPATH are Dynamic Level Numbering Scheme (DLN) [69] and work done by Harder et al. [70]. DLN reduces the renumbering effort after insert operations by the introduction of subvalues. The idea is that between two consecutive level values a and b, further values are possible by adding a suffix to a. Yet the resulting IDs need to be larger than all IDs of child nodes of a. This is accomplished by inserting a special character between a and the suffix, which is greater than the dot separating the level values. While DLN supports right side insertions after existing siblings, left-hand insertions may quickly lead to inflated labels. Nevertheless, the DLN has an added advantage in terms of having shorter reorganization intervals in case of unfavorable insertion orders. On the other hand, Harder et al. [70] introduce a parameter called distance, which is used to determine the gap needed to initially leave free space in the labeling between neighboring nodes at a given level for future node insertions. Similar to ORDPATH, only the odd division values are used for initial labeling while the even division values are reserved to indicate if overflow occurs.

**Multiplicative labeling.** This category of labeling scheme uses atomic numbers to identify nodes. Relationships between nodes can be computed based on some arithmetic properties of the node labels. The idea is to find a mapping from a given irregularly structured document tree $D$ to a regular tree $D'$ such that some of the arithmetic properties in $D'$ carry over to $D$. The main limitation is the computation of multiplicative labeling is very expensive. Hence, it is unsuitable to label a large-scale XML document. Among some of these labeling schemes are UID [71], recursive UID [72], prime numbering [73] and BIRD [74].

Weigel et al. [74] propose BIRD, which is an integer-based node identification scheme to answer structural queries. The BIRD scheme works as follows. Firstly, a structural summary (which is actually the dataGuide) called Ind (DB) is constructed from the tree database (DB). N is the set of nodes of DB while M is the set of nodes of Ind (DB). For each node $n \in N$, a certain interval size, or so-called as weight, is used to number all nodes. Together with an index mapping $\phi: N \rightarrow M$, they unify all weights needed for $n$ with the same index node $m$, where $m \in M$, selecting the maximal interval size among all members of the equivalence class $\phi^{-1}(m)$. This unified weight is attached to the associated node $m$ of the Ind (DB). When enumerating the nodes of the database, they reserve this interval size for all subtrees rooted at any of the nodes in $\phi^{-1}(m)$. Since not all these subtrees are of the same size, some numbers remain unused in the enumeration so that one could further find room to insert other nodes.

Labeling a document tree DB with BIRD ID is done in three phases. First, DB is traversed once to determine for each document node $n$ the number of children of $n$, which is later used to determine the aforementioned subtree interval of $n$. In the second phase, the Ind (DB) is traversed bottom-up to compute and store the weight. Finally, DB is traversed again in pre-order traversal order to assign BIRD weight to all document nodes in DB, based on the weights in Ind (DB). Consider the database DB with DataGuide, Ind (DB) shown in Fig. 13(a) and (b) respectively. Each node $m$ of the Ind (DB) is annotated with its child-balanced weight (in this case let $s=1$, which means 1-equivalent), $w_i(m)$ and the pre-weight $w'_i(m)$ (in brackets). The authors define some rules to compute pre-weight as shown below.

$$w'_i(m) := \begin{cases} w_i(m) & \text{iff } m \text{ has any child } m_1, \\ 1 & \text{otherwise}, \end{cases}$$

$$w_i(m) := \max\{w'_i(m)|m_1,m'\},$$

where $\text{childCount}(n)$ denotes the number of children of node $n$.

Nevertheless, only the weights $w_i(m)$ are stored physically in DataGuide. To understand how the depicted pre-weights and weights are computed, consider the leaf node that resides in the deepest level in Fig. 13 (b) (in this case, it is caption). The procedure runs bottom-up and begins with the leaf where the pre-weight is
fixed to $w_1(caption) = 1$. Next, we consider the index node figure, which is associated via $\phi^{-1}$ with database node 142 only. The child-Count$(142) = 1$ and hence, the child weight is multiplied by a factor $1 + 1 = 2$ according to the definition above. The resulting pre-weight is therefore $w_1(figure) = 2$. Both index nodes text and table are leaves, and hence their pre-weight is fixed to 1. As a result, the weight of each of the three siblings text, figure and table is the maximum of their pre-weight, i.e., 2. On the higher levels, pre-weights and weights are computed in exactly the same way until we reach the root with weight $w_1(publications) = 288$.

With BIRD labeling scheme, relationship between any two nodes can be determined easily. For any descendant $n'$ of a node $n$, we have ID$(n) <$ ID$(n') <$ ID$(n) + w(n)$. For instance, let $n'$ be figure (node 142) and $n$ be chapter (node 126), we have 126 < 142 < 126 + 270. On the other hand, for any child $n'$, the parent can be determined by ID$(n') =$(ID$(n)$ mod b) if we know the weight $b$ of the unknown parent. Assume $n'$ be figure (node 142) and the weight of figure's parent is 6 (obtained from Fig. 13 (b)), therefore, 142 – (142 mod 6) = 142 – 4 = 138. Thus, BIRD allows for a limited number of node insertions, until an overflow occurs when a node has more children than its ID range allows for. If this happens, then a global reallocation of IDs becomes necessary.

**Hybrid Labeling.** Most labeling techniques towards the later years are based on hybrid approaches. This group of labeling schemes uses some mixed combination of the strengths of existing approaches with the intention to support faster query processing.

BLAS [75] is a Bi-Labeling-based system for efficiently processing complex queries over XML data. Each node in BLAS is labeled with two labels, i.e., P-labeling and D-labeling. BLAS uses P-labeling (P represents path) to process queries involving consecutive child axis, and D-labeling to process queries involving descendant axis traversal. Firstly, the BLAS approach decomposes a twig pattern query into several suffix path queries. Suffix path queries are queries that start with an optional descendant axis step followed by zero or more child axis steps. Next, for every child axis in a suffix path, qualified nodes can be retrieved directly by checking on their P-labels. Hence, this certainly reduces the number of joins required. However, for path queries with A–D axis after decomposition, each axis needs to be evaluated one after another based on their D-labels. All qualified nodes (intermediate results) are then subsequently matched and merged using some structural join algorithms.

Han and Fu [76] propose a hybrid index structure, HiD, which consists of a combination of label path and data path information.
to efficiently support all kinds of queries. The label path is transformed into a binary tree and Haverman’s code is generated to serve as the label path identification. As for the data path, they adopted the DataGuide [42] as the index structure. Based on their observation, in a large XML document, there are usually few label path numbers and a lot of data path numbers. Furthermore, a label could have more than one label path and label path numbers.

Hence, the index framework to be constructed contains three layers, i.e., hash table to store the unique label path number for direct access, ordered list to group the rest of the same label path together and B + Tree index for efficient data path retrieval. With the hash table, a label is mapped into an address pointer. Each address pointer points to a corresponding entry in an ordered list, which in turn points to the root or the leaf node list in the B + Tree. In addition, they also propose some structural join algorithms to join the intermediate list results together.

He et al. [77] propose Hybrid Labeling of Sub-Structures (HLSS) by combining the power of interval encoding and the 2-hop approach [65]. HLSS identifies different types of substructures within a graph and encodes them using techniques suitable to the characteristic of each of them. Basically, HLSS assigns labels in two phases, namely, tree-reachability reduction and remainder graph-reachability encoding, where the first phase identifies and encodes strongly connected components as well as tree substructures, and the second phase encodes the remaining reachability relationships by compressing dense rectangular submatrices in the transitive closure matrix. During these two phases, four types of labels namely, strongly connected component label, interval label, portal label and remainder label are assigned to each node. Hence, given any two nodes u and v, testing whether u will reach v is straightforward. If either one has a strongly connected component label, its interval labels will be checked. If the answer is not affirmative, portal labels are retrieved to check the reachability between u’s out-portal and v’s in-portal. This involves testing whether their remainder labels intersect.

Summary and discussion for node indexing. In Section 4.1.2, various labeling schemes have been presented. Labeling schemes are the fundamental building blocks for many different structural join algorithms and a valuable complement to structural summaries. Thus, choosing a correct labeling scheme is crucial. There are several factors to consider when choosing a suitable labeling scheme as illustrated below:

- **storage**: How much storage space is available? Is there enough space to store each node label on disk or in memory?
- **nature of the data**: How large is the XML document? How frequent do they change?
- **query type**: Is structural query or full-text query (query with the tag and content) to be supported?
- **efficiency**: What is the manipulation cost of each labeling technique? How fast is the manipulation of node labels during query evaluation?

Table 3 summarizes the advantages and disadvantages of the labeling techniques mentioned above.

Table 4 depicts the expressiveness of each labeling scheme. Generally, a good labeling scheme should be able to determine the type of relationship present between any two nodes efficiently. As mentioned earlier, query edges for structural path query are either P–C or A–D relationships, whereas query edges for structural twig query pattern may be either P–C, A–D or sibling (preceding and following) relationships. Thus, a good labeling scheme should be able to support all these relationships. Table 4 provides an overall summary on the determination of relationships supported. The ‘✓’ indicates that the feature is supported while the ‘.’ indicates it is not supported.

In this section, we have seen the advantages and limitations of each labeling scheme. For example, subtree labeling schemes have fixed-sized labels and are suitable to cope with the characteristics of large XML trees. Yet, this scheme is weak in data updates, as a large number of nodes need to be relabeled for most update operation incurred. Although the prefix-based labeling schemes are very efficient in determining A–D and P–C relationships, they still suffer from large label size growth. Extra storage space is required to store these labels, which will affect query processing performance, as more disk access is needed. Hence, it is not suitable for large XML trees. The multiplicative labeling schemes, on the other hand, assume perfectly balanced and static trees to provide extra functionality, but waste huge number of labels for virtual nodes in a highly skewed tree. As such, label computations are rather expensive and rather impractical for real applications. Besides, this labeling scheme is not widely applied to any structural join processing. The hybrid labeling schemes merit the widest scope. As can be seen, most researchers toward the later years are increasingly proposing hybrid-labeling schemes. Hybrid labeling has an added advantage in terms of its potential to support larger class of queries and support dynamic updates efficiently. Nevertheless, label construction for these hybrid approaches is rather computationally expensive.

Recently, researchers [62,78–82] have diverted their focus to introducing new labeling schemes to support dynamic updates. XML databases are subject to change. Thus, the next step in making XML into a full-featured data exchange format is to support not only queries but also updates over XML content. We expect further experimentations with hybrid approaches in the near future especially those capitalizing on the semantics inherent in data structures, supporting complex queries and dynamic updates.

4.1.3. Sequence-based indexing

Sequence-based indexing such as VIST [83], PRIX [84] and xbw [85] transform both XML documents and queries into sequences, and evaluates queries based on sequence matching.

In Virtual Suffix Tree (VIST), XML data tree and queries are transformed into structure-encoded sequences, so that querying becomes equivalent to finding non-contiguous subsequence matches of the query in the data [83]. Each structure-encoded sequence is in the sequence of (symbol, prefix) pairs, (a1, p1), (a2, p2), …, (an, pn) where a represents a node in the XML data tree and represents the path from the root node to node a. For example, the XML document Doc1 and a query Q, shown in Fig. 14(a), are transformed into the sequences in Fig. 14(b). ViST is an indexing technique that unifies structure and content indexes into a single index. As can be seen in Fig. 14(b), the content is integrated into the sequence together with structure information. Most queries can be performed through direct subsequence matching. In Fig. 14(b), there is a single match since Q encoded sequence appears only once in the Doc1 encoded sequence. For query with wild-cards (‘*’ and ‘/’), the wild-cards will be discarded. However, the prefix paths of their sub nodes will contain a ‘*’ or ‘/’ symbol as a place holder. For instance, the query ‘//P/[[T=[XM(0)] Query’ is transformed into ‘(P,varepsilon)’, ‘(P,J)’, ‘(T,P,J)’, ‘(XML Query’, ‘P/J/T]’’. This query will be handled as a range query in sequence matching.

Although, VIST supports flexible queries without join operations, false alarms may exist in query results since a sequence match is not necessarily a tree match. For example, let us take a look at another example in Fig. 15. It is easy to see from the XML tree that Doc2 does not contain an answer for query Q2. However, the corresponding structure-encoded sequence of Q2 is a subsequence of Doc2. Thus, a false answer appears in the answer set. As a result, Rao and Moon [84] propose PRIX (PRüfer sequences for Indexing XML), another sequence-based indexing to overcome the limitation of VIST.
Then, a Prüfer sequence can be constructed as follows. Let post-order number to uniquely label each node in the data tree. A post-order sequence can be obtained efficiently. In the PRIX system, they used a series of refinement phases, namely (1) filtering by subsequence matching between trees and sequences. The twig pattern is also transformed into its Prüfer sequence. By performing subsequence matching against the indexed sequences in the database, followed by a series of filtering and refinement matches then undergoes a series of filtering and refinement phases, namely (1) filtering by subsequence matching, (2) refinement by connectedness, (3) refinement by structure and (4) refinement by leaf nodes. Consequently, the final answers are returned as shown in Fig. 16(e). Recently, Rao and Moon propose a new algorithm to holistically process a twig pattern without breaking the twig into root-to-leaf paths [87]. The evaluation studies show that PRIX yields good performance for processing queries with high selectivity.

Table 3
Summarization on advantages and disadvantages of various labeling schemes.

<table>
<thead>
<tr>
<th>Labeling scheme</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Extended preorder</td>
<td>Efficient in determining A–D and P–C relationships. Support insertion/update as long as the reserved spaces have yet been consumed.</td>
<td>Hard to determine the correct “size” for each node.</td>
</tr>
<tr>
<td>2. Inverted List</td>
<td>Support both structural and full-text queries. Fixed length label.</td>
<td>Not supporting insertion/update (label needs to be regenerated).</td>
</tr>
<tr>
<td>3. Dewey ID</td>
<td>Efficient in determining A–D and P–C relationships.</td>
<td>Label’s length scales up quadratically as number of fan-out and level increases. Not practical to support XML tree with huge fan-out.</td>
</tr>
<tr>
<td>4. ORDPATH</td>
<td>Support insertion/update efficiently without changing any existing label.</td>
<td>Not suitable for XML with deep trees. Label’s length scales up quadratically as number of fan-out and level increases. The label construction is rather computational expensive.</td>
</tr>
<tr>
<td>5. UID</td>
<td>Given a node label, its parent can be determined efficiently based on some calculation. Support insertion/update as long as the number of predefined fan-out is not over flowed.</td>
<td>Recursive look-up is necessary to determine query in A–D relationship.</td>
</tr>
<tr>
<td>6. Prime number</td>
<td>Supporting dynamic update; no re-labeling required Size unaffected by number of fan-out and levels.</td>
<td>Each prime number can be used only once. The label construction is rather computational expensive.</td>
</tr>
<tr>
<td>7. BIRD</td>
<td>Efficient in determining A–D and P–C relationships. Support insertion/update as long as the assigned weight for each node is not over flowed.</td>
<td>The label construction is rather computational expensive.</td>
</tr>
</tbody>
</table>

Table 4
Overview of the query support of each labeling technique.

<table>
<thead>
<tr>
<th>Labeling scheme</th>
<th>Path query</th>
<th>Twig query</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Extended preorder</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>2. Inverted List</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>3. Dewey ID</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>4. ORDPATH</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>5. UID</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>6. Prime number</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>7. BIRD</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

Fig. 14. (a) A XML document and an input query (b) Structure-encode.

Fig. 15. False alarm in ViST [84].

PRIX indexes and queries XML using Prüfer sequences [86]. PRIX first transforms each XML document into a sequence of labels by Prüfer’s method that constructs a one-to-one correspondence between trees and sequences. The twig pattern is also transformed into its Prüfer sequence. By performing subsequence matching against the indexed sequences in the database, followed by a series of refinement phases, all occurrences of a twig pattern in the database can be obtained efficiently. In the PRIX system, they used post-order number to uniquely label each node in the data tree. Then, a Prüfer sequence can be constructed as follows. Let $T_n$ be the data tree with $n$ nodes labeled from 1 to $n$. From $T_n$, delete a leaf node with the smallest label to form a smaller tree $T_{n-1}$. Let $a_i$ denote the label of the node that was the parent of the deleted node. This process repeats on $T_{n-1}$ to determine $a_{i-1}$ (the parent of the next node to be deleted) and continue until only two nodes joined by an edge are left. The sequence $(a_1,a_2,a_3,\ldots,a_{n-1})$ consists entirely of post-order numbers is known as Numbered Prüfer Sequence (NPS). If each number in an NPS is replaced by its corresponding tag, a new sequence that consists of XML tags known as Labeled Prüfer Sequence (LPS) is formed. Consider the following data tree, $T_n$ and query, $Q$ in Fig. 16 part (a) and (b). Fig. 16(c) shows the corresponding LPS and NPS for both $T_n$ and $Q$. Since LPS ($Q$) matches some subsequence occurrences in LPS($T_n$), there are few possible matches (shown in Fig. 16 (d)). Among them are nodes at positions $\{1,8,15,16\}$, $\{1,14,15,16\}$, $\{2,8,15,16\}$ and so on. The possible matches then undergoes a series of filtering and refinement phases, namely (1) filtering by subsequence matching, (2) refinement by connectedness, (3) refinement by structure and (4) refinement by leaf nodes. Consequently, the final answers are returned as shown in Fig. 16(e). Recently, Rao and Moon propose a new algorithm to holistically process a twig pattern without breaking the twig into root-to-leaf paths [87]. The evaluation studies show that PRIX yields good performance for processing queries with high selectivity.
Ferragina et al. [85] propose xbw transform of a labeled tree based on Burrows–Wheeler transform [88] for strings to compress, index and process XML data. The xbw transform uses path-sorting and grouping to linearize the labeled tree into two coordinated arrays, one capturing the structure and the other the labels. In their system, for a labeled tree, $T$, they build a sorted multiset $S$ consisting of triplets, one for each tree node. To build $S$, $T$ is visited in pre-order traversal. For each visited node $u$, the triplet $s[u] = (\text{last}[u], a[u], p[u])$ is inserted in $S$, where $\text{last}[u]$ is a binary flag set to 1 iff $u$ is the last child of its parent in $T$, $a[u] \in \Sigma$ denote the label of $u$, and $p[u]$ the string obtained by concatenating the symbols on the upward path from $u$’s parent to the root of $T$. Notice that $\pi[u]$ is formed by labels

![Diagram](image1)

**Possible matches**

In $LPS(Tn)$, node position at $<1 \ 14 \ 15 \ 16 >$, $<2 \ 8 \ 15 \ 16 >$, $...$, $<13 \ 14 \ 15 \ 16 >$

**After a series of refinements**

Final answers: $<9 \ 18 \ 17 \ 18 >$ and $<15,18,17,18 >$

![Diagram](image2)

**Sequence-Encoding**

$LPS(Tn) = B \ B \ C \ F \ S \ C \ B \ P \ B \ S \ C \ B \ P \ J \ P$

$LPS(Q) = 9 \ 8 \ 6 \ 7 \ 8 \ 9 \ 18 \ 15 \ 13 \ 14 \ 15 \ 18 \ 17 \ 18$

$LPS(Q) = B \ J \ P \ P$

$LPS(Q) = 2 \ 5 \ 4 \ 5$

**Listing 1. Algorithm GetChildren() [85]**

1. If $(\Sigma[l] \cup \Sigma) = \emptyset$ then return -1;
2. $c = \text{Sel}(i)$;
3. $y = \text{Sel}((A, c), f' \in F[c])$
4. $k = \text{rank}(\pi, i)$;
5. $z = \text{rank}(\pi_{\text{last}}, y - 1)$;
6. $\text{First} = \text{select}(\pi_{\text{last}}, z + k - 1) + 1$;
7. $\text{Last} = \text{select}(\pi_{\text{last}}, z + k)$;
8. return (First, Last).

**Fig. 16.** PRIX system.

**Fig. 17.** (a) A labeled tree $T$. Notice that $a[u] = a[v] = D$ and $p[u] = p[v] = BA$. (b) The multiset $S$ as a result of the pre-order visit of $T$. (c) $S$ after the stable lexicographic sort executed on the $\pi$’s component of its triplets [85].
of internal nodes only. Then, they stably sort $S$ lexicographically according to the $p$-component of its triplets. Fig. 17 illustrates an example. Hereafter, $S_{last}[i]$ refers to the last component of the $i$th triplet of $S$; $S_{a}$ contains all the labels of the nodes of $T$; $S_{p}$ contains all the upward labeled paths of $T$. The two important operations are rank and select over arbitrary sequences. Given a sequence $S[1..t]$ over an alphabet $R$, $\text{rank}c(S[1..q])$ is the number of times the symbol $c \in R$ appears in $S[1..q] = s_1, s_2, ..., s_q$, and $\text{select}c(S[1..q])$ is the position of the $q$th occurrence of the symbol $c \in \Sigma$ in $S$.

They propose an algorithm named GetChildren (depicted in Listing 1) to get all children of a particular node. Given an index $i$, $1 \leq i \leq t$, the algorithm GetChildren computes the range of rows $(\text{First,Last})$ such that $S[\text{First}, S[\text{First}]+1, ..., S[\text{Last}]$ are the children of node $S[i]$.

Consider the example in Fig. 17 and let the parent node be $p(v)$. To find all children of $p(v)$, parameters $c, y, k$ and $z$ need to be computed. Let $c$ denote the label of node $S[i]$. Firstly, $F[c]$, which is the first position of $S_{p}$ whose starting symbol is $c$, is computed. Then, followed by computation on $k$, the number of times the symbol $c$ appears in $S_{a}$ up to and including the position $S_{a}[i]$. Based on

<table>
<thead>
<tr>
<th>Sequence-based Index</th>
<th>Path query</th>
<th>Twig query</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ViST</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2. PRIX</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3. xbw-transform</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table 6
Query support for each indexing technique.

Listing 3.

```
1. //curA points to the head of AList
2. //curB points to the head of BList
3. While (curA != endOfList & & curB != endOfList) {...
4. if (BList[curB].start < AList[curA].start)
5. curB++; //advance cursor in BList
6. else {
7. begin the inner-loop join, then advance the cursor
8. if AList
9. temp_curB = curB;
10. //while B is in the region range with A
11. while (BList[temp_curB].start < AList[curA].end) {...
12. case 1: //query with child axes such as "\text{A}\text{B}".
13. if (AList[curA].level + 1 = BList[temp_curB].level)
14. addToSolution();
15. case 2: //query with descendant axes such as "\text{A}\text{B}".
16. addToSolution();
17. temp_curB++;
18. curA++;
19. }
20. curA++;
21. }
22. }
```

Listing 2.

```
1. if (i == 1) then return 1;
2. y = rank(A[0]);
3. k = rank1(S_{last}[i-1]) - rank1(S_{last}[y]), y = 1;
4. p = select(y, k = 1);
5. return p.
```

The core process of MPMGJN approach.

Fig. 18.
Fig. 17, we have $i = 4$ (the index position of $p(v)$), $c = B$ (label of $S[i]$ where $S[i]$ is a leaf), $y = 5$ since $S[5]$ is the first triplet having $i$-component that starts with $B$, $k = \text{rank}_B(Sa, 4) = 2$ since the number of times ‘B’ appears in $Sa$ until index position 4 is 2 and $z = \text{rank}_1(S_{\text{last}}, 4) = 1$ since the number of times ‘1’ appears in $S_{\text{last}}$ until index position 4 is 1. The First = 8 since $\text{select}_1(S_{\text{last}}, 2)$ pick-up the index position with the first ‘two’ occurrence with ‘1’ in $S_{\text{last}}$ is 7 and the Last = 8 since $\text{select}_1(S_{\text{last}}, 3)$ pick-up the index position with the first ‘three’ occurrence with ‘1’ in $S_{\text{last}}$ is 8. Thus, $p(v)$ has only one child, which is located at $S[8]$. They also propose another algorithm named GetParent (depicted in Listing 2) to find the parent of a particular node.

**Summary and discussion on sequence-based indexing.** Table 5 summarizes the advantages and disadvantages of sequence-based indexing while Table 6 shows the degree of support based on each type of query.

Although ViST and PRIX support all types of queries, false alarm may occur because sequence matching is not tree matching, and hence, wrong answers may be returned. The xbw-transform approach, however, does not support all types of queries and the determination of relationships among nodes is rather difficult.

---

**Listing 4. Algorithm StackTree**

1. // min_start is used to determine which entry in the stack 2. // is to be removed out from the stack 3. // all nodes in the stack with end attribute smaller than 4. //min_start are popped out of the stack 5. //Q is the input query 6. 7. Pre-order traversal (Q) { 8. min_start = Min (ALIST[cursorA].start, BLIST[cursorB].start) 9. 10. If (min_start = ALIST[cursorA].start) { 11. stackSol.push (node ALIST[cursorA]) 12. cursorA++ 13. } 14. Else if (query with child axes such as ‘A/B’) 15. { 16. //where top is the node on the top of the current stack 17. addToSol (top, BLIST[cursorB]) 18. } 19. Else if (query with descendant axes such as ‘A/B’) 20. { 21. //where nodelist is any ‘A’ node in the current stack 22. cursorB++ 23. }

---

**Listing 5. Algorithm PathStack**

1. // min_start is used to determine which entry in the stack 2. // is to be removed out from the stack 3. // all nodes in the stacks with end attribute smaller than min_start 4. //are popped out of the stack 5. //Q is the input query 6. 7. While (!endOfList in the leaf node list) { 8. min_start = Min (LIST[cursor].start) 9. //return the smallest start attribute 10. // of all nodes in Q and 11. //its associated list identifier, q 12. If (q is not the id of the leaf node in Q) 13. StackQ.push (node LIST[cursor] with pointer link to the top entry 14. //of its parent stack) 15. Else { 16. //q is a leaf node 17. addToSol (stackQ, 1) 18. //add Solution to SolutionList 19. pop(stackQ) 20. } 21. } 22. }

**4.2. Join indexes and decomposition-matching–merging processes**

In this section, we review some of the well-known join indexes and its associated algorithms such as MPMGJN [63], Stack-Tree [89], PathStack and TwigStack [90] and TSGeneric [91]. These algorithms serve as the fundamental blocks for structural join algorithms.

**The MPMGJN Approach.** Zhang et al. [63] propose the MPMGJN (Multi-Predicate MerGe Join) algorithm, which is the first native approach to implementing structural joins. MPMGJN is based on the region encoding labeling scheme, whereby each node in the XML document is annotated with $h_{\text{start}}, \text{end, level}}$. To answer a query such as ‘A/B’ or ‘A//B’, the two cursors, curA and curB, which initially point to the head of the AList (the list of the potential ancestors or parent) and BList (the list of the potential descendants or childs) are created. Both lists contain the occurrences of the nodes with tags A and B sorted in accessing order by the $\text{start, end, level}$ attribute. Then, they are compared with each other.

---

Fig. 19. The stack operation of Stack-Tree approach.
and advanced in line to implement the merge join operation. At each step of comparison, the cursors are advanced as shown in Listing 3.

Fig. 18 illustrates the matching process of MPMGJN. Basically, during the inner-loop join, each B-tagged node must be in the range of an A-tagged node, that is, \( A_{\text{start}} < B_{\text{start}} < B_{\text{end}} < A_{\text{end}} \). For query with ‘A/B’, the final answers are all performed using inner-joins. However, for query with ‘A/B’, the level attributes need to be accessed to confirm that the level difference between the nodes is one.

The stack-tree approach. As seen earlier, in MPMGJN, a node may be accessed several times during the matching phase. Besides, in some cases such as when processing query with P–C edges, it visits some nodes and adds it to the solution unnecessarily, although the nodes do not participate in the final solutions. For instance, based on the example in Fig. 18, MPMGJN finds the child \( B_4 \) only after it has scanned through \( B_1 \) to \( B_4 \), where all the descendants, \( B_2 \) to \( B_3 \) have to be accessed unnecessarily. To overcome these limitations, Al-Khalifa et al. [89] propose Stack-Tree, which uses a nice stack property to cache the nodes nested on the same path in data tree. Listing 4 shows the core of the Stack-Tree algorithm. At each step, the data node with the smallest start attribute is taken out of its list. Let’s assume the same input query, ‘A/B’. Initially, the algorithm begins by checking if it is an A-tagged node, and pushes it into the stack. However, if it is a B-tagged node, Stack-Tree tries to use it to form solutions with existing A-tagged nodes in the stack.

Fig. 19 depicts the stack operations during the evaluation of the Stack-Tree algorithm. This algorithm guarantees that each node in the input list is scanned only once as compared to MPMGJN which require multiple scans. Chien et al. [21] improve and extend Stack-Tree by utilizing the B*-tree indexes to skip some unnecessary nodes. For example, when the current cursor, cursorA in AList is behind the current cursor, cursorB in BList, a probe on the B*-tree index of the descendant element node list can effectively forward cursorB to the first descendant element of cursorA and avoid accessing those in between.

The PathStack approach. The MPMGJN and Stack-Tree approaches both decompose queries into multiple binary relationships. As a result, these approaches generate large intermediate results. For instance, for an input query ‘A/B/C’, the query result of the first join, A join B, has to be written to disk first if its size
is too large to be contained in memory, and then be read back to memory to join with C, after A join B has been completed. This eventually will cause performance degradation due to high I/O cost disk access. Bruno et al. [90] propose a novel holistic path pattern matching called PathStack, which evaluates the query holistically without decomposing it into binary relationships. Thus, this method results in more manageable intermediate results. Listing 5 shows the core of the PathStack algorithm. Basically, the PathStack algorithm is structurally very similar to the Stack-Tree algorithm, except that it uses multiple stacks (one for each query node). Each node cached in the stacks has an associated pointer to the corresponding node in its parent stack to keep track of the path solutions. Thus, this guarantees that at all time, elements in all stacks are from the same path in the data tree. These matches can be output immediately as solutions through backtracking pointers associated with the elements in the stacks.

Fig. 20 illustrates the key idea of stack encoding in the PathStack algorithm. For example, for an input query A/B/C, there are three stacks, Stack C, Stack B and Stack A. Based on the XML SAX parser event, when an opening tag is encountered, the corresponding element is pushed into the stack named by its tag, which in turn, points to the topmost entry in its parent stack by a pointer. If a leaf node is reached, the path solutions are formed and added to the SolutionList (a list of possible solutions). Nevertheless, for input query with P–C ancestor, this method results in more manageable intermediate results. Listing 6 shows the core of the PathStack algorithm. Essentially, PathStack has two main phases as follows:

1. Output path solutions (lines 5–19): A list of root-to-leaf path solutions is output as partial solutions. This list is generated by repeatedly constructing the stack encodings to the query pattern and by iterating through the stream nodes in sorted order (as returned by the getNext function) of their start attributes.

2. Merge (line 20): All lists of path solutions in the first phase are merged to produce the final answers to the whole query twig pattern based on some common query nodes (the branching nodes).

In the getNext function, it firstly traverses down to the left-most leaf nodes (by recursively calling itself). Starting from the leaf node, it tries to find the highest possible query node with a descendant extension as in lines 29–36. A node q has a descendant extension if there is a solution for the sub query rooted at q composed entirely of the cursor elements of the query nodes in the sub trees. Given that all children have their descendant extensions, in order for node q to be returned, cursor Cq is advanced (line 32). If there is no common ancestor for all Cni in q, the child node with the smallest start attribute is returned (line 36).

Fig. 21 depicts an example of the TwigStack approach. Once node B1 is pushed into Stack B, a path solution is formed from Stack B?
do not have an ancestor B (the parent stack is empty). The cursor now advances to C5. The next call of getNext(Q) returns node B, whose descendant extension is (B1, C5). Similarly, the cursor of node A will be advancing to A5. After several calls of getNext(Q), eventually the cursor reaches solution (A7, B4, C9), which is the first match of the query. This process repeats until the cursors reach the end of the list of the data stream. The advantage in using the TSGeneric approach is the time taken to forward A’s cursor to A5 can be reduced significantly by invoking Cq → FwdToAncestorOf(Cnmax).

Summary and discussion. In this Section 4.2, the core operations on some structural join algorithms have been presented. Based on the decomposition-matching–merging processes, most researchers have proposed to decompose the twig query into a set of basic binary relationships using region encoding and subsequently perform matching of these binary relationships against the XML document before finally merge-joining these results. Jiang et al. [93] have extended the holistic approach and propose pipelining joining multiple inverted lists at one time based on B+–tree indexes so that no intermediate results are generated. As a result, many algorithms using indexes such as XR-tree [91], R-tree [94,95], B-tree [96] and CEI index [97] have emerged. On the other hand, Jiang et al. [98] extended the holistic approach to process twig queries with an OR predicate. Recently, Jiao et al. [99] and Yu et al. [100] have proposed a holistic path join algorithm for path and twig query respectively using NOT predicates.

Another similar approach is to decompose the twig query into a set of path queries, followed by a join operation to reconstruct the matched twig pattern. Polyzotis et al. [101] have proposed methods to reduce the number of intermediate results by introducing a filtration step based on some notion of synopses to facilitate query-approximate answers. They have proposed both TREE-SKETCH and TWIG-XSKETCH. Amer-Yahia et al. [102] have proposed to preprocess the query patterns before the matching phase is executed. Since the efficiency of tree pattern matching depends on the size of the pattern, it is essential to identify and eliminate redundant nodes in the pattern before the matching phase takes place. On the other hand, Zezula et al. [103] have proposed a novel technique, i.e., tree signature, to represent tree structures as ordered sequences of preorder and postorder ranks of the nodes.
storage. We have also seen that most commercial DBMS now seamlessly integrate XML data with structured data.

1. Using relational database as instance storage, mapping plays an important role to resolve the conflict between the hierarchical nature of XML data and the tabular format of relational data. There are many mapping techniques namely the Relational Schema-based, the Edge approach and the Attribute approach. Each mapping approach has different shedding techniques. As a result, the table structures of one method vary from one another. Generally, the Schema-based approach performs the best, followed by the Attribute and the Edge approaches respectively. Yet, overall, query optimization is quite restricted to the relational query processors. Nevertheless, with index utilization (using the existing B+-tree index in the relational query processors), both the Schema-based and Attribute approaches have improved query retrieval performance. This is critical for a large-scale dataset.

2. By creating native storage, there are several methods [96, 97, 107–109] to enhance query processing. One of the well-known techniques is to utilize indexes to reduce the search scope. Various index structures are surveyed in this paper, whereby we have suggested an indexing classification scheme of Path indexing; Node indexing and Sequence-based indexing. In addition, we have also surveyed several join indexes and the associated join algorithms to improve twig query (branching query) processing. These structural join approaches play an essential role in decomposing, matching and merging the final solutions.

3. Using hybrid storage enables users to seamlessly share, store, retrieve and update both existing structured data and XML data that had previously existed only in flat file or transient messages. Furthermore, it minimizes the amount of new skills required to incorporate XML data into their database environments, reducing labor costs and potentially speeding up project delivery cycles.

Toward later years, most researchers have shifted their focus towards augmenting various approaches together as hybrid indexing (elaborated earlier in Section 4). These technologies (combining structural index, labeling schemes and join algorithms) have high complementary potential in improving query processing [109–113]. For example, in our recent study [114] by combining the beautiful features of path summary in DataGuide and region encoding, along with some query processing, complex query processing has improved by "pre-matching" the node against the path instead of matching it individually. This has certainly reduced the number of joins required to evaluate a particular query.

We believe there are still many challenges that remain open. For example, how to process complex queries with many joins and branches, and how to infer appropriate schemas to store XML documents and capitalize on the semantics inherent in data structures should be addressed.

5. Conclusion

The optimization of queries on small documents seems to be of less importance in the early days. But, as the usage of XML shifts towards the data-oriented paradigm, datasets of a much larger scale ranging from few kilobytes to many gigabytes need to be supported. As such, efficient query retrieval becomes even more crucial.

In this paper, we have reviewed several major techniques for XML query processing based on two main storage classes, i.e., using relational database as instance storage and creating a native

References


Listing 7. Algorithm TSGeneric

1. If g is used to hold the tag of the next node to be process
2. /Cq is the cursor to point to some elements in the data stream of node g
3. /Q is the input query
4.
5. While (!endOLi in the leaf node list) {
6.   g = getNext(Q) //return the next node to be processed
7.   
8.   If (g is not the root of the query)
9.     cleanStack(Snumed, Cq) //clean the parent stack that out of the region range
10.   If (g is the root if $Snumed.size$ = empty) {
11.     cleanStack(g, Cq) //clean any node which is out of the region range
12.   If (g is not a leaf node)
13.     Stack.push(g, Cq, top($Snumed$))
14.   else
15.     outputSolution();
16.   }
17.   advance(Cq)
18.   }
19.   mergeAllPathSolution();
20. }
21.
22. Function getNextCursor(g) {
23.   If (g is leaf node) return g,
24.   For each child, which is children of g {
25.     n = getNextCursor(g); //recursive call to return the next node
26.     If (n / g) return n;
27.   }
28.   nmax = min acq[ { Cn → start }]
29.   nmax = max acq[ { Cn → start }]
30.   If (Cq -> fwdToAncestorOr(Cnmax) == TRUE)
31.     If (Cq is an ancestor of Cnmax)
32.       return g;
33.   }

Fig. 22. An example of XML document and the input query [91].

They used tree signatures as index structure and to find qualifying patterns through integration of structurally consistent path query.

Another group of researchers focus on answering the “prospective queries” by utilizing the XML to represent knowledge on the evolution of objects during their lifecycle [104]. Wang et al. [105] proposed a self-associated concept mapping approach for knowledge representation. On the other hand, Cheon et al. [106] proposed using knowledge representation and rule generation based on Bayesian network, as compared to traditional tree models.


