The VLDB Journal (2006) 15(1): 53–83
DOI 10.1007/s00778-004-0150-4

REGULAR PAPER

Jan-Marco Bremer · Michael Gertz

Integrating document and data retrieval based on XML

Received: 24 June 2003 / Accepted: 24 December 2004 / Published online: 12 August 2005

Abstract For querying structured and semistructured data, data retrieval and document retrieval are two valuable and complementary techniques that have not yet been fully integrated. In this paper, we introduce integrated information retrieval (IIR), an XML-based retrieval approach that closes this gap. We introduce the syntax and semantics of an extension of the XQuery language called XQuery/IR. The extended language realizes IIR and thereby allows users to formulate new kinds of queries by nesting ranked document retrieval and precise data retrieval queries. Furthermore, we detail index structures and efficient query processing approaches for implementing XQuery/IR. Based on a new identification scheme for nodes in node-labeled tree structures, the extended index structures require only a fraction of the space of comparable index structures that only support data retrieval.

Keywords Integrated information retrieval · Data retrieval · Document retrieval · XML · Index structures · Structural join

1 Introduction

Data retrieval and document retrieval are two complementary techniques to query data sources. Data retrieval comprises exact queries that allow a user to specify a precisely defined subset of a data source. Document retrieval arranges elements of a given document collection according to their relevance to a set of query terms. Relevance is defined based on statistical variances in term distributions throughout documents.

For a long time, users and application developers have been looking for way to integrate both retrieval approaches [27] because it is awkward and expensive to maintain two types of systems, each supporting only one of the retrieval approaches while potentially duplicating content. Furthermore, both retrieval approaches are largely applied to text-rich data sources. Therefore, it seems natural to integrate both of them into a single framework for arbitrary textual data. However, the integration of data and document retrieval for arbitrary data sources is not straightforward.

Most of today's document data have at least a partial structure (semistructure). Therefore, data retrieval principles can be easily applied to document data that are, for instance, represented through the Extensible Markup Language (XML [12]). Query languages such as XPath [9] and XQuery [10] prove this. Document retrieval, however, is not as easy to apply to arbitrary semistructured data for the following reasons. Relevance-based ranking employed by document retrieval requires a collection of documents as input. But in an arbitrary data source, no such single collection can be identified. Moreover, in the case of structured data, the data are often decomposed and mapped to, for instance, different relations in a relational database to avoid redundancies and update anomalies. Applying such a decomposition to marked-up text causes logical units of text that are essential for document retrieval to be split up. Each resulting piece of text might contribute to multiple higher-level concepts, which can only be put together dynamically. Therefore, existing document retrieval extensions that rely on a single static document view over, e.g., relational data [5], have only limited use.

Another open question for an integration of data and document retrieval is the composition of their respective operations. Document retrieval as a strictly Boolean keyword search [33] is easy to integrate into data retrieval, yet lags the important relevance-based ranking. Fuzzy text similarity operations [22] employ some principles of document retrieval, but no ranking.

Models for semistructured data like XML already employ a principle that is important for ranked document retrieval: order. Query languages like XPath [9] or XQuery [10] that recognize this order can be extended by a document retrieval operator to rearrange the order of data fragments based on their relevance to a term subquery [34, 35, 50].

J.-M. Bremer · M. Gertz (✉)
Department of Computer Science, University of California at Davis, One Shields Avenue, Davis, CA 95616, USA
E-mail: bremer@computer.org, gertz@cs.ucdavis.edu

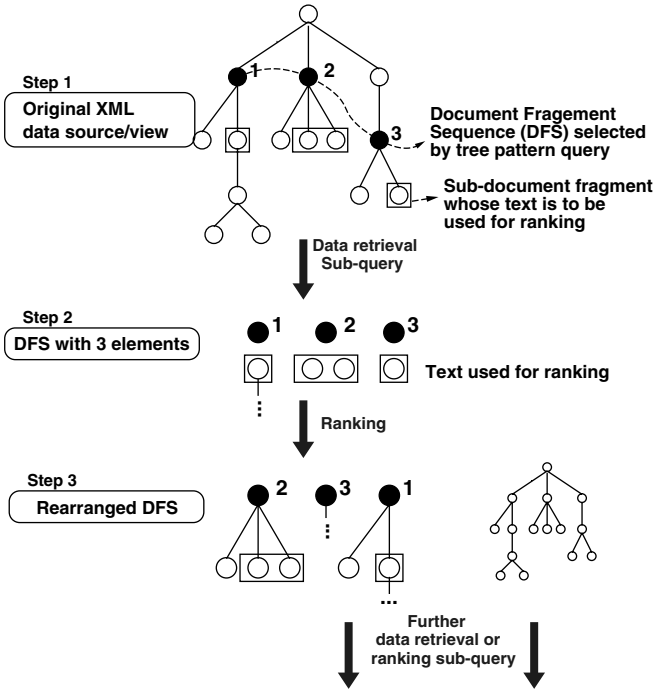

Fig. 1 Integrated information retrieval: document retrieval, based on document fragment sequences (DFSs) instead of whole documents, and thereby embedded into data retrieval

However, all existing extensions in this direction are operators that are only applicable to the final sequence of data fragments selected by a data retrieval query. The role of an operator for ranked document retrieval *within* arbitrary subqueries remains unclear.

In this paper, we introduce an operator that closes this gap and leads to a general *integrated information retrieval* (IIR) approach with more expressive power than data and document retrieval alone. The approach relies on a single, tree-structured data source as input. In IIR, document retrieval is *embedded* into data retrieval and provides for an arbitrary nesting of data and document retrieval operations, as illustrated in Fig. 1.

From a tree-structured source, a sequence of document fragments that are subtrees of the source is selected by means of a data retrieval (sub-)query in step 1. In step 2, based on a document retrieval query in the form of a set of terms, a relevance ranking weight is determined for each fragment in the sequence or only parts of each fragment (boxes in Fig. 1). In step 3, the intermediate sequence is then rearranged according to these weights. Subsequent data retrieval or document retrieval queries can be applied to the rearranged sequence or only its top k elements.

Embedded document retrieval supports standard document retrieval techniques on sources that do not provide for a natural view as document collection. Furthermore, it provides for any number of dynamically created document collection views. As a key factor for enhanced query capabilities in IIR, we introduce *dynamic ranking*. Dynamic ranking demands that only term statistics local to the current, intermediate document fragment sequence be used to derive relevance weights in an embedded document retrieval subquery. The dynamic ranking principle thus extends the statistical techniques underlying document retrieval to arbitrary local contexts.

The principles of IIR can be applied to arbitrary semistructured data. As one example, we give the syntax and semantics of XQuery/IR, which we introduced earlier [14]. XQuery/IR extends XQuery by a single operator called *rank*. We provide syntax and semantics of the *rank* operator and illustrate the functionality of the extended query language by means of a number of examples.

Compared to existing query approaches, there are some obvious costs associated with embedded document retrieval. As we will show in this paper, the term distribution statistics required for a dynamic ranking of an intermediate fragment sequence can be incorporated into existing set-based (XML) query processing approaches [3, 46, 71]. However, the additional word statistics significantly enlarge known index structures, which commonly already far exceed the size of the indexed source. To address this concern, we present a new node identification scheme and improved index structures based on these new node identifiers. Beyond IIR, the node identification and indexing scheme has widespread applications in comparable approaches to query processing for semistructured data.

We show that our index structures can efficiently support embedded document retrieval and tree-pattern queries, which build the foundation for querying semistructured data. The indexing overhead usually remains at less than half the size of a text-rich, real-world XML source, as our experiments clearly demonstrate. This is about 50% below the indexing overhead of the simplest, most storage-efficient comparable approach [71], and even more efficient compared to approaches that use a database system to store the index structures [65, 71]. This is despite the fact that these implementations do not support document retrieval. Furthermore, our node identification scheme can be used as an alternative to the interval node identification scheme [2], which most current approaches to XML indexing rely on.

For a number of different XML sources, we provide detailed structure and term statistics. These statistics can guide future research on index structures for data and document retrieval on XML.

In summary, this paper presents the following main contributions:

1. A **conceptually new approach to integrated data and document retrieval** in general, and relevance-based document retrieval in particular. This approach is illustrated by XQuery/IR, which realizes the new approach based on XML and XQuery.
2. A **new node identification scheme** that has applications in many existing indexing approaches.
3. **Storage-efficient index structures** that support tree-pattern queries and incorporate a mapping from logical node identifiers to their physical counterparts.

The rest of the paper is organized as follows. In Sect. 2, we discuss the data model underlying our approach and the

principles of data and document retrieval adopted in the rest of this paper. The discussion includes examples of the XQuery language. The extension of XQuery by a *rank* operator is presented in Sect. 3. In Sect. 4, we introduce a complete indexing approach to support integrated information retrieval. In particular, we introduce a new node identification scheme underlying the index structures. Query processing for integrated information retrieval is discussed in Sect. 5. In Sect. 6, we analyze various properties of the index structures as well as their effectiveness in supporting query processing. We present related work in Sect. 7 and conclude the paper in Sect. 8.

We assume that the reader is familiar with the very basic concepts and notions underlying XML, XPath, and XQuery.

2 Foundations

In this section, we establish the foundations of our integrated document and data retrieval approach based on XML. We introduce a data model, a basic schema structure, and core elements of data and document retrieval models and approaches that we assume in the rest of the paper.

2.1 Data model

As is customary, we model semistructured data, such as XML data, as ordered, node-labeled trees. We assume just a single tree structure. This is a more general model than relying on a collection of documents. In common XML terminology, the tree structure represents a single document, which consists of a hierarchy of subtrees representing document fragments.

In our model, we leave out some details about XML, such as comments, processing instructions, references, and the distinction between elements and attributes. The model follows the spirit of an XML information set [26] and an XQuery data model [31]. It can also be thought of as an order-preserving variant of earlier models such as the object exchange model (OEM) [52][1, p. 19]. In what follows, we first formally introduce the data model. Then we introduce additional terminology used in the rest of the paper.

Assume a set L of node labels and a set T of text string values including the empty string ϵ. Given a set X, let $seq(X)$ denote the set of all sequences that can be built over elements from X. Let $\emptyset$ denote the empty sequence.

Definition 1 *(Document fragment and document)*
A document fragment (DF) is a node-labeled tree $(V, root, children, parent, label, text)$ *with nodes* V *and root node* $root \in V$ *such that*

- $Children : V \rightarrow seq(V)$ *is a function that represents parent–child relationships*;
- $Parent : V \rightarrow V$ *is the inverse function of children, i.e., if* $v \in children(v')$, *then* $parent(v) = v'$;
- $label : V \rightarrow L$ *is a function that assigns a name (label) to each node; and*
- $text : V \rightarrow T$ *is a function that assigns text to nodes.*

A document is a document fragment. $\mathcal{F}$ *denotes the set of all document fragments (DFs) over* L *and* T.

To emphasize the fact that in this paper a single document represents a complete (semi)structured database, we also refer to a document as *data source* or just *source*.

For a DF $F \in \mathcal{F}$, we adopt the common notions of rooted label path as a sequence of labels starting with the root label $label(root)$ and, analogously, rooted data path as a sequence of nodes with pairwise parent–child relationships in F.

Based on the definition of DF we can now formally specify document fragment sequences:

Definition 2 *(Document fragment sequence)*
A document fragment sequence S *is a sequence of elements of* $\mathcal{F}$, *i.e.,* $S \in seq(\mathcal{F})$.

Let $|S|$ *denote the number of elements in* S. $first_k(S)$, $k \in \mathbf{N} \cup \{\infty\}$, *denotes the subsequence of* S *consisting of the first* k *elements of* S *or all elements if* $k > |S|$.

2.2 Schema

As schema, we assume a slightly extended (tree-structured) DataGuide [36], which we call *Extended DataGuide* (XDG). Schema information stronger than a DataGuide, e.g., a DTD or XML Schema for XML, may be available, but we do not require such information.

Like a DataGuide, an XDG enumerates all rooted label paths in a data source. In addition, we assume that for each node in the XDG the maximum number of its instances under a single parent node in the source is known, for example, the maximum number of /library/book/author nodes under any single parent node of type /library/book. In Sect. 4, we discuss the implications of this assumption and how this information can be obtained. The following specifies an XDG more formally.

Definition 3 *(Extended DataGuide)*
Let $D = (V, root, children, parent, label, text)$ *be a data source. The Extended DataGuide (XDG) for* D *is a triple* $(F, maxpos, nodeno)$, *where*

- F *is a document representing the tree-structured DataGuide for* D,
- $nodeno$ *is a function,* $nodeno : V_F \rightarrow M$, *where* V_F *are the nodes in* F *and* $M \subseteq \mathbf{N}$ *is the set of node numbers* $1, \ldots, |V_F|$. $nodeno$ *assigns a unique node number to each node in* F *in a preleft order traversal, and*
- $maxpos : V_F \rightarrow \mathbf{N}$ *is the function that returns the maximum number of siblings of a certain node type (node in* F*) under any related single parent node in* D.

Rooted label paths in a document D and nodes in the related XDG G are equivalent. Furthermore, every node in a source has a unique rooted label path. Therefore, we also

informally apply the functions *maxpos*() and *nodeno*() to nodes in D and use node numbers from G in the place of rooted label paths and vice versa.

2.3 Data retrieval

In what follows, we illustrate some basic concepts of XQuery [10] as a representative of typical XML data retrieval languages and as the language we will extend in Sect. 3. Then, we outline structural joins as the main technique to process tree-pattern queries, which build the core of XQuery queries. The structural join approach is especially suitable for extending standard query processing by the document retrieval functionality proposed in this paper.

2.3.1 XQuery

We illustrate some basic concepts of XQuery as they apply to this paper with some examples. For a more detailed discussion of XQuery, we refer the reader to the XQuery working drafts [10, 31]. We will extend some of the following examples in Sect. 3.4 to illustrate a new document retrieval operator for XQuery.

Example 1 Select paragraphs of articles published on Feb. 15, 2002 from a news document database.

```
document("news.xml")
    //article[./date="2002-02-15"]//paragraph
```

The example consists of a single XPath [9] expression, which alone is a valid XQuery query. The query extracts article DFs at arbitrary depths ("//") from a source named news.xml, selecting only those article nodes that have a date child node with value 2002-02-15. From the remaining article DFs, all paragraph nodes at arbitrary depth are determined and appended to a single sequence of DFs in document order [9].

Example 2 List all articles that appeared before 1996, including and ordered by their first author and title.

```
FOR $a IN document("bib.xml")//article
WHERE $a/year < 1996
ORDER BY ($a/authors/author[1], $a/title DESCENDING)
RETURN
  <result>
    <fstAuth>{$a/authors/author[1]/text()}</fstAuth>
    {$a/title}
  </result>
```

Example 2 defines a single loop that iterates over article DFs selected by an XPath expression. For all article nodes with a year child node that has a value less than 1996, the article's first author and title nodes are extracted. Complete title DFs build another part of the result fragments. New tags named fstAuth and result are introduced for the selected author information (using a path expression on $a nodes) and result sequence of DFs, respectively.The returned fragments are ordered by ascending author name and descending title. Braces ("{}") help to distinguish query parts that are to be evaluated from verbatim text.

The return clause in XQuery allows one to *construct* DFs that do not have a direct match in the queried source. Constructed fragments can be put together from fragments that are found in the source and from new nodes like fstAuth or result as in the above example.

Example 3 Convert a list of news articles classified under a certain category to a list of categories with their related articles.

```
<news_by_category>
{FOR $c IN document("news.xml")//category
 RETURN
   <category>
     <name>{$c/name/text()}</name>
     {FOR $a IN document("news.xml")
                 //article[@cid = $c/@id]
      RETURN
        <title aid={$a/@id}> {$a/title/text()} </title>
   </category> }
</news_by_category>
```

Example 3 represents a join between category and article DFs based on a news category identifier (cid). In XQuery, joins have to be specified explicitly as nested loops. The node type news_by_category introduces a new root node for the elements in the result.

The final Example 4 illustrates string search capabilities as proposed for XQuery ([20], Sect. 1.6). The example is very similar to Example 1, extending the earlier example by a Boolean search for the string "New York."

Example 4 Select articles with a paragraph that contains the string "new york."

```
document("news.xml")
    //article//paragraph[contains(./text(), "new york")]
```

All four examples above are *exact* queries. They select exactly those document fragments from a source that exhibit a specified path or tree pattern, or satisfy a predicate. The last example demonstrates XQuery's capabilities for exact keyword search. XQuery does not support weighted keyword search as employed by document retrieval. We will introduce a respective extension of XQuery in Sect. 3.

2.3.2 Query processing through structural joins

At the core of XML query languages like XQuery are path- and tree-pattern queries. For instance, the queries Location/Books//Title and Location[./Id/text() = "Zurich"]/Books//Title are examples of a path and a tree pattern, as Fig. 2 demonstrates. Edges in such patterns represent parent–child and ancestor–descendant relationships. Node labels or text values under certain nodes further constrain the patterns. A distinguished node called the *selection node* determines the root of fragments to be returned as the query result.

One of the most common approaches to processing path- and tree-pattern queries are *structural joins* [3, 46, 71].

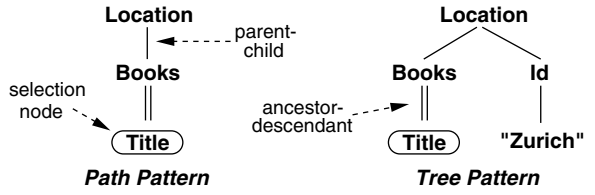

Fig. 2 A path-pattern and a tree-pattern query with matches in the form of book titles in the data source shown in the center of Fig. 4

Structural joins (SJs) are based on index structures that consist of lists of logical identifiers for nodes in a source. Each list is related to a certain node predicate, i.e., a common node label or label path, or a word occurrence. Node identifiers used in SJ approaches allow a query engine to determine parent–child and ancestor–descendant relationships between nodes. This allows for reassembling instances of the full pattern as they occur in a source by joining node ID lists [3].

For example, to assemble the Id–"Zurich" subpattern from Fig. 2, the node ID lists corresponding to label Id and term "Zurich" are accessed. Node IDs from these lists are matched to find pairs that have a parent–child relationship and thus resemble the pattern. Keeping both lists in document order allows for executing this kind of join operation in at most linear time with respect to the number of matching pairs of node IDs [3].

The most significant query capabilities other than tree-pattern queries in a language such as XQuery are construction clauses. Structural joins do not support such clauses directly. However, the same logical node identifiers as described above can serve as the basis for an extended SJ approach that allows for constructing new DFs and keeping track of the origin of parts of DFs constructed in an intermediate query step [28].

Therefore, for discussing the implementation of the approach introduced in this paper, we assume path- and tree-pattern queries as the underlying query model for data retrieval. We rely on a SJ approach because index structures for SJs are very similar to inverted file index structures in document retrieval, which is discussed next. SJ index structures are especially suitable for an extension of XML data retrieval by embedded document retrieval, as we will show in Sect. 4.

2.4 Document retrieval

Document retrieval is concerned with ordering documents from a usually flat collection of documents by *relevance* to a simple query of typically only a few terms [8, 55, 59]. Document retrieval is also known as information retrieval, although the result of a document retrieval query are *pointers* to relevant documents rather than information, i.e., meaningful, user-interpreted data [41]. *In this paper, we use the term (integrated)* information retrieval *exclusively to denote our retrieval approach that integrates document and data retrieval.*

In document retrieval, relevance is based on term distribution statistics. A term is a word, found within a document's text, or just the stem of a word. Stemming words, and thus mapping multiple different words to a single stem, has proven useful to compensate for semantically equivalent but syntactically different word forms.

To determine relevance, a numerical weight is assigned to each term-document pair in a document collection. The weight represents the term's significance within the document content. The relevance-based ordering of documents with respect to a query, also called *ranking*, is obtained by summing up query-term weights for every document and determining the highest sum. Because documents are represented by terms and their multiple occurrences within a single document, the standard model underlying most document retrieval approaches is called the bag-of-words model.

A standard approach to weighting terms is the term frequency-inverse document frequency (tf-idf) approach [59]. Tf-idf assigns higher weights to terms with high in-document and low overall document frequencies, i.e., few documents that contain the term. Other schemes that are not based on tf-idf still rely on the same term distribution statistics, and thus show similar relative weights. Retrieval *effectiveness* in document retrieval commonly refers to the quality of the ranking obtained through different weighting formulas. Recent flavors of tf-idf and other document retrieval approaches are discussed, for examples, by Baeza-Yates and Ribeiro-Neto [8].

Term distributions and, in particular, values such as the inverse document frequency may vary widely throughout a document collection. Hence, the significance of a term within a document and thus its relevance depend largely on the context. In contrast to our approach, in most existing retrieval approaches, the context is always the complete document collection. For example, the idf is a static value determined once when a document collection is indexed.

Indexing is the process of parsing a document collection, assigning an identifier to each document, and keeping track of the number of times each term is observed within a document (tf or "term counter"). Other parameters of statistical value like the document length or number of unique words per document are extracted in this step as well.

The standard index format for storing term frequencies is the *inverted file* format [68]. An inverted file is an indexed sequential file that maps term identifiers to lists of pairs of document identifiers and counters. These lists are also called *postings* lists. Term-independent information like document length is much less extensive and can easily be stored in a sequential file of fixed-size records.

In document retrieval, the relatively simple inverted file access structure has proven superior to more complex structures like B-trees for indexing, index storage, and query processing, in particular for large amounts of data [68, p. 109] or even data of Web scale [16].

3 XQuery/IR and embedded document retrieval

In this section, we introduce the basic elements of XQuery/IR, which extends XQuery [10] by a document retrieval operator with well-defined syntax and semantics. XQuery/IR provides an interface and framework for integrated document and data retrieval queries. The rest of this paper will be concerned with query processing strategies and supporting index structures that efficiently implement the core components of this framework.

3.1 Concepts

In traditional document retrieval, a query result is a relevance-based total order of documents. Documents are the sole units within a collection. The result of a query is directly presented to the user. These facts raise two major questions for integrating document retrieval into an XML query language:

1. What is a suitable equivalent for document and document collection within an arbitrary XML source?
2. What is the role of ranked document retrieval *within* a query as opposed to its usual application as final query operation?

XML queries extract document fragments (DFs) from a source. DFs are the only available data units suitable to replace the notion of a document from standard document retrieval. In XQuery in particular, a query result is a *sequence* of DFs [31]. Hence, it is desirable to have an embedded document retrieval approach that *modifies the order of DFs within a sequence of DFs*. This can be implemented through a single operator in the same way as the order operator works within XQuery (see Example 2 in Sect. 2.3.1).

In the resulting embedded document retrieval approach, document fragment sequences (DFSs) replace the notion of document collection. The order established by the new operator can be transparent to subsequent queries that do not rely on any order. Yet the order can be exploited by other subqueries or shown as the end result to the user. Furthermore, an embedded document retrieval provides for query capabilities that cannot be realized by either document or data retrieval alone: retrieval based on a local context of interest.

Data retrieval applied to an intermediate, ranked result allows one to derive information from a ranked DFS, for example, aggregated information. Document retrieval applied to an intermediate DFS provides for localized rankings *if term statistics that are used for the ranking are specific to the input DFS*. Relying on the same term statistics that standard document retrieval uses, but in an *arbitrary* local context, is an important new concept, which we call *dynamic ranking principle*. The dynamic ranking principle generalizes earlier approaches to localized ranking such as passage retrieval [45, 58] and other approaches whose local context is still limited to a per-document basis [34, 40, 50, 64, 66].

On the one hand, the dynamic ranking principle is just a requirement that follows from our assumption to have a single source (view). In such a source, there is no notion of document collection from which a single, static set of term statistics can be derived. On the other hand, dynamic ranking provides for a new kind of document retrieval query that enables the detection of *locally relevant* information.

As a simple example, the query term "shotgun" in the context of biological data will likely lead to information pieces about the so-called shotgun gene sequencing approach. In a more general context, e.g., the Web, "shotgun" would likely be considered more related to firearms instead of biology. In general, term statistics vary widely throughout a source. The notion of relevance is closely tied to these statistics. Therefore, local context for document retrieval is a valuable tool to gain new insights into text-rich information sources. Moreover, document retrieval based on local context subsumes standard document retrieval.

Notice that the benefit of the *integrated information retrieval* (IIR) approach is not an improvement of local retrieval effectiveness (Sect. 2.4) – any existing weighting function can be plugged into our approach. It is rather the enabling of a weighted ranking in arbitrary contexts that provides the true power of dynamically ranked DF sequences.

Furthermore, localized ranking eliminates the need to predefine a static document collection, potentially having to materialize different document collection views for the same source. Therefore, IIR can also eliminate the need for special document management systems added to a general database system.

3.2 Requirements

The above considerations lead us to the following requirements for integrating a document retrieval operator into an XML query language, in particular XQuery.

Total order. The operator should order an arbitrary sequence of DFs based on relevance.

Local context. The operator should allow for the elimination of less relevant elements in a ranked DFS.

Closure. The operator should be closed within XQuery and therefore be applicable to arbitrary intermediate query results.

Transparency. The operator should not affect queries in a way other than changing the order within or eliminating elements from a DFS.

Exchangeability. The relevance-weighting algorithm underlying the operator should be exchangeable. A user should have means to influence the weighting algorithm for a query.

Furthermore, we impose the following additional requirement as it increases the potential utility of an embedded ranking:

Visibility. The operator should make ranking weights explicit by showing them as part of a ranked result, but without causing side effects to the embedding query.

In what follows, we introduce the syntax and semantics of an operator that meets the above requirements.

3.3 Syntax

XQuery/IR is built on a single new operator called *rank*. The operator is used within a RankBy clause, which is an extension of the XQuery FLWOR expression [10, Sect. 3.8] in the following way:

```
FLWORExpr ::= (ForClause | LetClause)+ WhereClause?
              (OrderByClause | RankByClause)?
              "return" ExprSingle
```

The syntax and semantics of RankByClause is very similar to OrderByClause. The latter is defined in the XQuery draft [10, Section 3.8] as:

```
OrderByClause ::= ("order" "by" | "stable" "order" "by")
                  OrderSpecList
OrderSpecList  ::= OrderSpec ("," OrderSpec)*
OrderSpec      ::= ExprSingle OrderModifier
OrderModifier  ::= ("ascending" | "descending")? ...
```

A list of order specifications (OrderSpec) within the OrderBy clause determines the DFs to be sorted in ascending or descending order. The sorting algorithm can be required to be stable. Example 2 in Sect. 2.3.1 showed an example of an OrderBy clause.

We define a RankBy clause that extends FLWOR expressions as follows (we use the same syntactical EBNF representation as the XQuery specification [10]).

Definition 4 *(RankBy clause)*

```
RankByClause ::= ("rank" "by" | "stable" "rank" "by")
                 QuerySpecList
                 ("based" "on" TargetSpecList)?
                 ("limit" n ("%")? )?
                 ("ascending" | "descending")?
                 ("using" MethodFunctCall)?
QuerySpecList ::= ExprSingle ("," QuerySpecList)?
TargetSpecList ::= PathExpr ("," TargetSpecList)?
```

QuerySpecList is a list of strings (constant DFs) or expressions that return DFs interpreted as strings. TargetSpecList is a list of path expressions that determine sub-DFs of the input DFS. The text of these sub-DFs is then used exclusively for ranking the elements of the DFS. MethodFunctCall refers to an XQuery function.

Unlike OrderBy, ascending and descending are applied to the whole QuerySpecList. QuerySpecList represents a single document retrieval-style query rather than a list of order criteria. In addition, there are <based on>, <limit>, and <using> clauses. The optional MethodFunctCall allows for choosing a certain internal weighting algorithm. The call can have the style of a FunctionCall in the XQuery working draft [10, Sect. 3.1.5] but would refer to an internal function whose implementation is hidden. Ideally, an API would allow one to plug different weighting algorithms into a system while letting the user choose one by name.

The basic semantics of XQuery/IR is as follows. A ranking query given in the QuerySpecList is used to determine a ranking weight for every DF in the input DFS. The weight is computed by an algorithm that may be chosen at runtime by means of the <using> clause. A set of path expressions within the <based on> clause allows for limiting the text on which the weight is based to only parts of input DFs (boxes in Fig. 1). The <limit> clause eliminates trailing elements from the ranked result.

In what follows, we illustrate the semantics of XQuery/IR by means of some examples. In Sect. 3.5, we give a signature and formal semantics of the weighting algorithm underlying XQuery/IR and then formally define the *rank* operator.

3.4 Examples

As indicated above, XQuery/IR queries are very similar to XQuery queries that use an OrderBy clause. The following example query, which is similar to Example 4, illustrates this aspect. At the same time, the query shows main differences to Boolean text searches as presented by Example 4:

Example 5 Retrieve a maximum of 100 paragraphs with relevant information about New York from a news data source:

```
FOR $d IN document("news.xml")//article//paragraph
RANK BY "new," "york" LIMIT 100
RETURN $d
```

The result of Example 5 might look as follows:

```
<paragraph weight="0.96">
  New York fire fighters...
</paragraph>
<paragraph weight="0.81">
  Wall Street today closed at a record...
</paragraph>
<paragraph weight="0.79">
  ...
```

The XML attribute "weight" is an explicit representation of the computed relevance weight.

Example 6 Select articles that appeared before 1935, and sort them by their abstracts' relevance to Albert Einstein:

```
FOR $a IN document("bib.xml")//article
WHERE $a/year < 1935
RANK BY "Albert," "Einstein" BASED ON $a/abstract
RETURN
 <early_paper>
  <firstAuth>{$a/authors/author[1]/text()}</firstAuth>
  {$a/title}
  {$a/abstract}
 </early_paper>
```

The relevance weight in Example 6 is based only on parts of the articles, namely the contents of abstracts. The simple subquery in the RANK BY clause determines these parts. In a more advanced query, the terms "Albert" and "Einstein" could be replaced by a subquery as well. This subquery could select the text to be used as query from another document as shown in the next example.

Example 7 Use selected text, which describes Albert Einstein's accomplishments, from another source (authors.xml), to rank the articles from Example 6:

```
FOR $a IN document("bib.xml")//article
WHERE $a/year < 1935
RANK BY document("authors.xml")
  //author[./name="Einstein"]/accomplishments
RETURN
  <early_paper>
   ...
```

In the next example, we assume a set of news articles that belong to a certain news category.

Example 8 Rank a given set of articles by their relevance to the category assigned to each article, utilizing the keywords given for each category. List only the ten most relevant articles for each category:

```
<news_by_category>
{FOR $c IN
document("news.xml")//category
 RETURN
  <category>
   {$c/name}
   {FOR $a IN document("news.xml")
                  //article[@cid = $c/@id]
    RANK BY $c/keywords LIMIT 10
    RETURN
      <title aid={$a/@id}> {$a/title/text()} </title>
  </category> }
</news_by_category>
```

In this example, the ranking is executed in an inner loop for each category. As in Example 7, the input to RANK BY is dynamically computed through a path query relative to the context node $c.

The final example assumes a source named biology.xml containing a set of publications in the general area of biology. A scientist is searching for the top experts in the field of shotgun sequencing, which is a technique used in genome sequence assembly. While some publications might be classified under related keywords, in general, these keywords are insufficient to precisely identify respective experts directly. Because the search area is quite specific, a manual, time-consuming search is hard to avoid with existing query approaches.

An automated approach to tackle this problem is to select all publications and rank them by relevance to a suitable set of biological terms. Then, by summing up the weights on a per-author basis, the authors with the highest number of most relevant publications can be determined. These authors are good candidates for being experts in the field of genome sequencing. This is what the following query specifies.

Example 9 Determine experts in the field of genome sequencing as discussed above:

```
LET $input :=
  FOR $p IN document("biology.xml")//publication
   RANK BY "shotgun," "genome," "gene," "sequencing"
  RETURN $p
RETURN
  <shotgunseq_experts>
    FOR $author IN distinct-values(
          document("biology.xml")//publication//author)
    LET $pub := $input//publication[.//author = $author]
    WHERE sum($pub/@weight) >= 4.0
    ORDER BY $author
    RETURN
      <expert>
        {$author/name}
        <expdegree>{sum($pub/@weight)}</expdegree>
      </expert>
  </shotgunseq_experts>
```

The query also serves as an example of the usefulness of an explicitly represented weight attribute. In the example, the weight attribute is used to eliminate authors with low overall relevance to the search field. Furthermore, the sum of the weight values serves as indicator for the overall degree of expertise (expdegree). To keep the query simple, experts in the query result are not ordered by their overall degree of expertise, a step that could be added easily.

3.5 Semantics

Assume a set $seq(\mathcal{F})$ of document fragment sequences (DFSs) related to a given source as introduced in Sect. 2.1. Let *weight* be a special name in the label set L underlying $\mathcal{F}$. Queries in the XQuery/IR extension can be simple terms or, as Example 7 shows, DFSs that are selected from the set $seq(\mathcal{F})$ by means of a subquery. Therefore, we model queries as DFSs whose text can be interpreted as a set of simple query terms. In particular, typical term queries like "XML, retrieval" can be considered as a sequence of document fragments (DFs), each having only a single text string node.

Definition 5 *(Weighting algorithm)*
Assume a set of queries $\mathcal{Q} = seq(\mathcal{F})$. Let $\mathcal{Q}$ include the DFs consisting of only a string from T and in particular single terms. Let $\mathcal{W}$ be the set of operators

$$W : seq(\mathcal{F}) \times \mathcal{Q} \to seq(\mathcal{F}),$$
$$W(S, q) = S'$$

such that S' is equal to S except that each DF in S' has an additional node labeled weight at the root. The weight node has a single number value representing the relative relevance weight of the respective DF within S. Then an operator $W \in \mathcal{W}$ is called a weighting algorithm.

While the DFS that is used as query can be interpreted as a set of query terms, Def. 5 does not impose this limitation. The interpretation of queries and limitations to allowed query types depend only on the weighting algorithm W.

Within a DFS to which a weighting algorithm is applied, a standard bag-of-words or any other document retrieval model can be assumed. Any ranking scheme, e.g., one of the numerous flavors of tf-idf, can be applied as long as the respective parameters can be provided. In the weighting algorithm, the *weight* node added to DFs is an explicit representation of the computed ranking weight. In XML, the most suitable implementation of the additional weight node is as an attribute with a reserved name to avoid side effects.

In what follows, we define the ranking operator for XQuery/IR in terms of a weighting algorithm W and the

order operator underlying the OrderBy clause in XQuery as discussed earlier.

Definition 6 *(Rank operator)*
The rank operator is an operator:

$$rank : seq(\mathcal{F}) \times \mathcal{P} \times \mathbf{N} \times \mathcal{Q} \times \mathcal{W} \longrightarrow seq(\mathcal{F})$$
$$rank(S, \pi, k, q, W) = first_k(order_{weight}(W(\pi(S), q))),$$

where $\pi \in \mathcal{P}$ is a path expression and $\mathbf{N}$ is the set of natural numbers. $order_{weight}$ refers to the sorting of DFs within S based on the weight attribute. $First_k$ was described in Def. 2; it eliminates all but the first k elements from S.

We have already illustrated the rank operator in Fig. 1. *Rank* takes a DFS S as input, applies the path expression π to each of its elements, and collects the resulting DFs *for each DFS element*. When no <based on> expression is used in a query, π is empty. In that case, all text is supplied to the weighting algorithm W and used for rearranging the DFs.

After the sequence is sorted using the computed weight, trailing elements of the resulting DFS are eliminated through $first_k$, specified in a <limit> clause. These kinds of eliminating elements based on position can be improved when the weighting algorithm supplies meaningful relative weights. In this case, keeping only the first DFS elements that amount to a certain percentage of the total weight of all elements would provide a better means of selecting only the most relevant DFs. Such an advanced cutoff requiring a special algorithm to implement is another reason why we do not rely on XQuery's existing order by construct for rearranging the ranked fragment sequence as proposed in the recent XQuery and XPath full-text requirements [18].

In this section, we have defined a small extension of the XQuery language that enables an approach to integrated data and document retrieval. The rest of this paper will be concerned with an efficient implementation of the above framework, in particular the processing of XQuery queries into which the ranking is embedded.

4 Supporting index structures

In this section, we discuss how an integrated information retrieval (IIR) language like XQuery/IR can be efficiently supported by suitable index structures. We focus on the most common kinds of path- and tree-pattern queries and the ranking of their intermediate results through embedded document retrieval.

4.1 Overview

We rely on a modified structural join approach (Sect. 2.3.2) to process the path- and tree-pattern queries discussed in Sect. 4.2. We modify and extend the basic structural join approach in the following way.

1. We use a new logical node identification scheme that encodes complete rooted data paths. The encoding is context-independent, space efficient, and fast to decode. No decoding is required to determine parent–child and ancestor–descendant relationships between node IDs (Sect. 4.3).
2. In all index structures, we group node IDs by common, rooted label path. In combination with an XDG, this provides for efficient index accesses and utilization of storage space (Sect. 4.4).
3. The main path index structure employs a sparse storage of node IDs that has no negative impact on reconstructing index lists but further reduces the index size.
4. We extend the standard set of index structures for structural joins by a physical address index. This index maps logical node IDs to their physical addresses at no overhead beyond storing just the addresses.

The index modifications and extensions, and the new node ID scheme in particular are necessary to avoid excessive storage overhead. Existing node identification and indexing approaches mainly ignore storage efficiency [3, 46, 65, 70, 71]. This is not desirable in our approach because the additional term statistics that are needed to support embedded document retrieval would make the index structures unreasonably large, as discussed in Sect. 4.5.

Figure 3 gives an overview of the index structures used in our approach. An XDG provides access to the SJ approach's two core index structures, which store groups of node IDs related to a certain term or path. In conjunction with these index structures, three further indexes provide (a) the term statistics required to rank arbitrary document fragment sequences (DFSs) and (b) physical addresses related to node IDs. These physical addresses point to document fragments within the indexed XML source. For processing queries, the index structures are accessed in the sequence indicated in the figure by arrows. Query processing follows the procedure described in Sect. 2.3.2.

Section 4.6 presents the complete index framework as introduced in this section, and it also provides a more detailed version of Fig. 3.

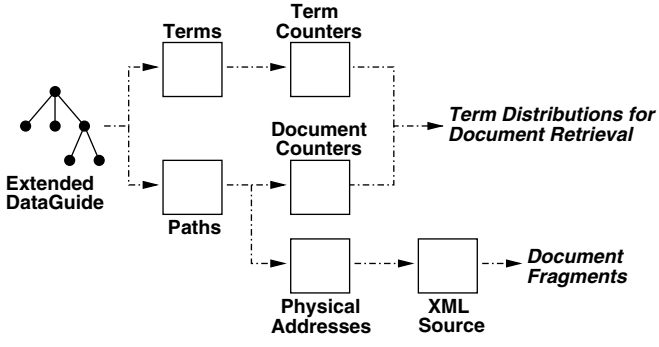

Fig. 3 Overview of index structures

4.2 Queries

We aim at supporting path-pattern and tree-pattern queries that contain conditions on parent–child, ancestor–descendant, and preceding and following sibling relationships. In XPath terminology [9], these relate to the *self*, *parent*, *child*, *ancestor*, *descendant*, *preceding-sibling*, and *following-sibling* axes.

Moreover, notably different from most related work in the structural join arena, we chose to support conditions with relative sibling positions rather than absolute node positions in document order [31]. We did so because typical queries like the XPath query //book/author[2] are only concerned with the sibling order among nodes of the same type. The example query selects the second author, if present, of all books from a source. Because the query is concerned only with the order among author siblings, the absolute position of an author node among all children of a book node within a source is neither relevant for this query nor sufficient to efficiently evaluate the query. However, absolute node positions are the only query options supported by existing structural join approaches [3, 21, 46, 71].

4.3 μPID node identifiers

At the heart of our approach is a new node identification scheme called the μPID scheme. μPIDs are instances of *minimal path identifiers*. In what follows, we first introduce minimal path identifiers and then discuss μPIDs.

4.3.1 Minimal path identifiers

Consider the example source at the center of Fig. 4, ignoring the rest of the figure for now. The black nodes in the figure mark the rooted data path p = /DigitalLibrary/Loc[5]/Books/Bk[4]/A[3], which is an instance of the label path l = /DigitalLibrary/Loc/Books/Bk/A. The information in p is sufficient to uniquely identify a particular book author.

Assume the label path l is given. What further *minimum* amount of information is necessary to fully represent p and thus uniquely identify the author node p specifies? – Starting from the root, the *sibling positions* 5, 4, and 3 for the Loc, Bk, and A nodes are sufficient to determine which particular A node is specified by p. However, these sibling positions are only required for nodes for which multiple siblings of the same label path exist anywhere in the source. No sibling position is ever required for the root node of a source (DigitalLibrary in the example) or for a Books node (assuming no Loc node has more than a single Books child). The steps within any rooted data path that require a sibling position can be determined from the XDG. This is because the XDG provides the maximum sibling positions for every rooted label path in a source (*maxpos*, see Def. 3). Hence the above rooted data path p can be minimally represented as a pair (/DigitalLibrary/Loc/Books/Bk/A, <5,4,3>).

Formally, a minimal path identifier is defined as follows:

Definition 7 *(Minimal path identifier)*
Let S be a data source and G the XDG for S. A minimal path identifier *(minPID) for S is a pair (p, s), where p is a rooted label path in G and s is a sequence of natural numbers. The numbers in s are sibling positions for all nodes p_i along p for which $maxpos(p_i) > 1$. The numbers are ordered by their distance from the root node.*

Relationships between two nodes a and b in a data source can easily be determined based on their minPIDs. If no label path of one of the minPIDs is a prefix of the other minPID's label path, then the nodes cannot have an ancestor–descendant or parent–child relationship. Otherwise, the minPIDs' sibling positions determine which, if any, of the two relationships the nodes have.

In what follows, we present an efficiently encoded instance of minPIDs called μPIDs ("micro-Path IDs").

4.3.2 μPID scheme

μPID node identifiers consist of a pair of integers. The first integer, called *node number*, is an identifier for a rooted label path as found in the XDG. According to Def. 3, we assume that all nodes in the XDG have such a sequential number assigned, following a preleft order traversal. The second integer of μPID node identifiers is called *position number* and contains the encoded sibling position sequence.

Figure 4 shows the construction of the position number for the data path p = /DigitalLibrary/Loc[5]/Books/Bk[4]/A[3] used above. The position number bit-accurately encodes sibling positions for the Loc[ation], Bk, and A[uthor] nodes.

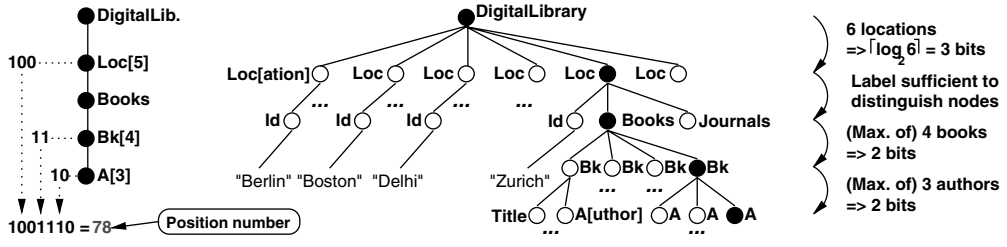

Fig. 4 μPID encoding of rooted data path /DigitalLibrary/Loc[5]/Books/Bk[4]/A[3] in a digital library example source: the numbers required to distinguish between sibling nodes (*right*) are bit-accurately encoded and appended to a k-bit integer (*left*). k is constant for each rooted label path. Position number and rooted label path uniquely identify a node in the source

The number of bits required at each step in a data path is determined by the maximum number of related sibling nodes anywhere in the source (*sibling fanout*). In Fig. 4, assuming every Books node in the source has at most four Bk children, $\lceil \log_2(4) \rceil = 2$ bits are sufficient to encode Bk sibling positions.

The sibling fanout information is a part of the XDG in the form of the *maxpos* function and can be obtained as follows. For a source that already exists at the time the XDG is constructed, the fanout can be derived from the source. Otherwise, the fanout can be derived from estimates or schema information that might exist at least for parts of a source. Some space in the sibling position range can be left to provide for future updates and insertions in a source or for constructing a data source from scratch. For updates and insertions, reserving such space in node identifiers is also state of the art in other node identification schemes [46].

Furthermore, our empirical studies on real-world data sources show that most node types vary only slightly in their number of siblings. Therefore, our encoding of node IDs is quite effective, as our experimental evaluation in Sect. 6 shows, although in the worst case, the length of μPIDs can grow linearly with the number of nodes in a source.

Worst-case μPID length. As a worst-case scenario, a source structure as shown on the left side of Fig. 5 results in the longest overall μPID length for a given total number of nodes n.

The source has a total depth of d with just three nodes at each level, and thus a total number of $n = 3d + 1$ nodes. With all nodes labeled with the same label A it takes only $\log(d+1)$ bits to encode the node numbers that identify the distinct label paths A, A.A, A.A.A, etc. Encoding the position numbers, however, requires two *position bits* at each depth level in order to distinguish between the three sibling nodes with identical label A. This leads to a total μPID length of $\log(d + 1) + 2d = O(n)$ bits. That this is really the worst μPID length that any source with n nodes can have follows mainly from the fact that the source in Fig. 5 is a deep source, where a maximum number of 0.5 out of 2 bits (25%) is wasted at each depth step. It is easy to see that with more or fewer than three nodes at each level the *relative* waste is smaller than 25% (e.g., 0.5 out of 3 bits for respective five A-nodes results in only 17% waste).

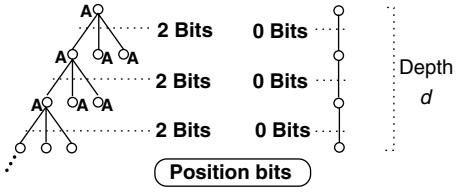

Fig. 5 Worst-case source for μPIDs (*left*): For a source with $n = 3d+1$ nodes, μPIDs can reach a maximum bit length of $2d + \log(d + 1) = O(n)$, the node number accounting for only $\log(d + 1)$ bits. A completely degenerate, linear tree (*right*) requires no bits for node positions, thus giving an optimal μPID length of $\log(n)$

However, a source structure that causes μPIDs to grow linearly is unlikely to occur in real data, as our experiments in Sect. 6 confirm. Moreover, in a partially degenerated source, μPID length is only affected locally, and thus only a few nodes exhibit this property. Therefore, although in the theoretical worst case the length of node IDs could grow linearly with the number of nodes in a source while growing only logarithmically in earlier approaches [44], in practice, the length of node IDs in our approach shows bounds that are similar to those in these earlier approaches. However, μPIDs directly encode more information in each node ID than in previous approaches. Overall, this makes μPIDs likely more suitable for efficient query processing than node ID schemes that have guaranteed length limits, but much weaker support for query processing, an aspect we will detail in Sect. 5.

Position numbers are constructed by appending bits related to single sibling numbers within a rooted data path. Therefore, all position numbers related to a certain node number, i.e., rooted label path, have the same bit length. Hence, they can be interpreted as *k-bit integers* as shown on the left side of Fig. 4. This fact is a main contribution to the efficiency of the μPID scheme. The bit length of position numbers related to a certain node number in an XDG can easily be kept with each XDG node. In what follows, we assume a function $numlen()$ that maps each XDG node to the length (possibly zero) of its related position numbers.

A node's μPID allows for directly deriving the node's parent node, ancestors, and preceding siblings. Other node relationships need to be derived as in related approaches through SJs. Compared to minPIDs, μPIDs make it even easier to check for node relationships. Parent–child and ancestor–descendant relationships can be determined through comparing node numbers and matching prefixes of position numbers. The time complexity of the comparison is reduced to *O(1)* from *O(k)* for minPIDs, where k is the number of nodes in the XDG.

The following propositions give a formal specification of core node relationships and the node order, which plays a crucial role in constructing index structures and evaluating queries with a position condition (Sect. 4.2). For this, let $a = (n_1, N_1)$ and $b = (n_2, N_2)$ be μPIDs, where the n_i are node numbers in an XDG G and the N_i are position numbers. Furthermore, let $prefix_k$, $prefix : \mathbf{N}^2 \rightarrow \mathbf{N}$, be the function that returns the k most significant bits of an integer.

Proposition 1 *(Node relationships)*
Node b is a child of node a if and only if

$$n_2 \in children(n_1) \text{ in } G \ \wedge N_1 = prefix_{numlen(n_1)}(N_2).$$

Replacing children *with* descendants *defines descendant relationships analogously.*

Proposition 2 *(Node order)*
Let n_0 be the least common ancestor node of n_1 and n_2, and $\ell_0 = numlen(n_0)$. Then, the node order $\leq$ on μPIDs can be

given as:

$$a \leq b \Leftrightarrow (\ell_0 = 0 \wedge n_1 \leq n_2) \vee (\ell_0 > 0 \wedge (prefix_{\ell_0}(N_1) < prefix_{\ell_0}(N_2) \vee prefix_{\ell_0}(N_1) = prefix_{\ell_0}(N_2) \wedge n_1 \leq n_2)).$$

Clearly, given two μPIDs, the relationship between respective nodes and their relative position within a source can be determined in constant time. For the node relationships, this requires an XDG that allows parent–child and ancestor-descendant relationships to be determined in *O(1)*, which can be accomplished using interval node IDs [2] (see also Sect. 6) for nodes in the XDG.

The above node order is not sufficient to establish a global document order and thus to reconstruct a source from fragments with just their μPIDs. However, complete reconstruction of a data source is not the purpose of an index structure. Furthermore, little additional information like that provided by the A-index introduced in the next section is sufficient to derive a global document order from μPIDs.

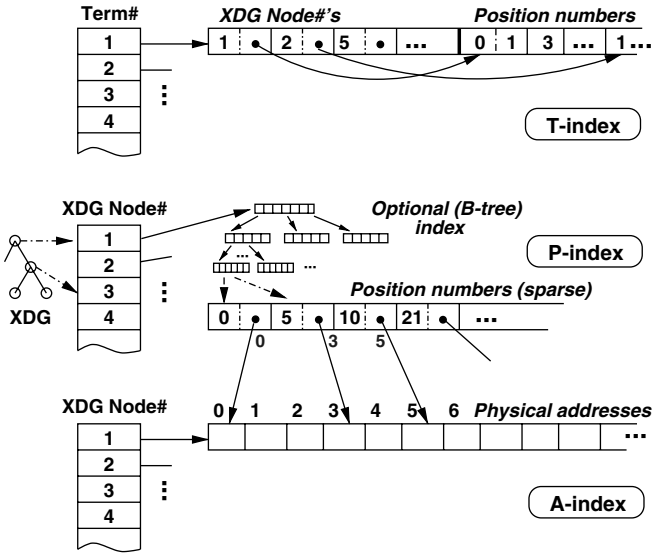

Fig. 6 Core index structures: P-, A-, and T-index

4.4 Core index structures

We use μPIDs within two standard index structures of the structural join (SJ) query processing approach: a *path index* (P-index) and a *term index* (T-index). The term index allows for processing term containment queries and the path index for the remaining path and tree-pattern queries. However, our indexes have two major differences with most existing approaches. Instead of using flat lists of μPIDs in document order, μPIDs are grouped by their XDG node numbers in both index structures. This is equivalent to grouping IDs by a common rooted label path.

The grouping allows us to avoid repeated storage of node numbers for each μPID in index lists. Only position numbers are stored repeatedly. Within each group the number of bits of position numbers is constant. Therefore, every element in the group can be directly addressed. Furthermore, the grouping provides for a more efficient processing of pattern queries. This is because a query pattern can first be matched against the relatively small XDG in main memory to identify all the qualifying node numbers. Then, only μPIDs of these node numbers have to be considered within the index structures. For path patterns, a similar procedure is employed by Yoshikawa et al. [70] but based on node IDs that are stored in database tables.

As a further improvement to the path index, we employ a sparse storage of position numbers. Position numbers related to the same XDG node number and in document order build a sequence of integers with gaps like **0**, 1, 2, **5**, 6, **10**, 11, 12, etc. Instead of storing all these elements, we store only the start element in each continuous subsequence together with the element's position in the full list, in the example (**0**,0), (**5**,3), (**10**,5) etc. (see also Fig. 6).

Example 10 In Fig. 4, consider the position numbers for author nodes as in the marked path. The marked author node has position number 78, as discussed previously. Its two preceding author siblings have position numbers $(1001101)_2 = 77$ and $(1001100)_2 = 76$. The construction is analogous to the one shown on the left side of the figure. Assuming the preceding book node Bk[3] has only two authors, these authors would have the position numbers $(1001000)_2 = 72$ and $(1001001)_2 = 73$. Hence, the subsequence of /Digital-Library/Loc/Books/Bk/A nodes for these two books is 72, 73, <gap>, 76, 77, 78.

An important observation, which we will confirm in our experiments in Sect. 6, is that in real-world data these gaps are relatively rare. Therefore, the path index size is significantly reduced by sparse storage.

Sparse storage has a further use. The index position associated with each position number can also be used as a direct pointer into another index structure, which stores μPID-related physical addresses. We call this index the *physical address index* (A-index). As the path index has to be accessed for query processing anyway, there is no additional cost associated with obtaining direct pointers into the A-index. Physical addresses then allow for the retrieval of source fragments. Physical addresses can be offsets into a file, tuple IDs in a relational database system, OIDs in an object database system, etc. It should be noted that in existing SJ approaches an efficient mapping of logical node identifiers to their physical counterparts is frequently not considered.

Besides sparse storage, μPIDs allow for another storage optimization within the P-index: when child nodes of a certain rooted label path always occur exactly once under their respective parent nodes, only information related to their parent nodes is stored in the P-index. This is possible because the childrens' node identifiers can be easily computed from their parents' identifiers.

Figure 6 shows the overall design of P-, A-, and T-indexes. The figure also shows an additional, small B-tree

index as an option to speed up nonlinear access to node identifiers in the path index. Depending on typical access patterns, such an index could also be employed on the term index. Nonlinear index access can be useful for joining μPID lists that are relatively short with respect to the length of the complete list [21], an aspect that will become obvious in Sect. 5.

4.5 Indexes for embedded document retrieval

The main parameters used in most weighting schemes for document retrieval such as the core tf-idf scheme discussed in Sect. 2.4 are term frequency (*tf*), the inverse of the document frequency (*df*), and document length (*dlen*). In embedded document retrieval, the analogous parameters are (1) the number of times a term occurs in a document fragment, (2) the number of fragments in an intermediate document fragment sequence that contain a term, and (3) the total number of nonunique terms found in a document fragment. Further term- or fragment-specific parameters can be incorporated in the same way as these three basic parameters. Therefore, in what follows, we focus on providing *tf*, *df*, and *dlen* for arbitrary fragment sequences (we use the abbreviations *tf*, *df*, *dlen* with respect to embedded document retrieval from now on).

The objective is to incorporate these parameters into the existing indexing framework laid out above while preserving the storage and access efficiency of the host data retrieval system. The major factor in achieving this goal is to exploit typical properties of semistructured sources.

4.5.1 Options for incorporating tf, df, *and* dlen

The *df* parameter is a byproduct of data retrieval or can be obtained through an additional structural join. The *df* of a term t in a DFS S is the number of elements in S that can also be found in the T-index list for t. If this list has to be accessed for the data retrieval part of a query anyway, no additional cost is associated with obtaining the *df*. This is the case, for instance, when a query first asks for document fragments that contain the term t at least once and then ranks them by their weight with respect to t. The additional join that is necessary otherwise is an unavoidable expense for the additional functionality embedded document retrieval provides.

Other than the *df* parameter, *tf* and *dlen* have to be explicitly stored. For the kind of storage and the storage location of *tf* and *dlen* counters, three basic options exist:

1. *Repeated storage vs. accumulation at runtime.*

Both *tf* and *dlen* can be stored repeatedly for each level of a source's tree structure, or just for the node they are *directly* related to. For example, for a term "Smith" occurring under a data path /book[4]/authors/author[1] (the fourth book's first author) in a source, one could just keep the information that (/book[4]/authors/author[1], "Smith") has a *tf* counter of one. In this case, a query that ranks the 12 /book[3-14] fragments according to their relevance to "Smith" would require the accumulation of all *tf* counters from authors, title, text, and whatever other descendant nodes a book node might have.

The accumulation of counters means a structural join for every level of the tree structure under a DF's root node. These joins do not come for free, but, being a core operation in the host system, their associated costs are at least minimized as far as possible. Path identifiers, as they are used in our approach in particular, allow for an effective selection of only small sublists containing all the potential matches.

To avoid accumulation at runtime, additional counters need to be stored. In the above example, a counter of one is stored again for (/book[4]/authors, "Smith"), assuming no other author is named "Smith," and another possibly greater counter for (/book[4], "Smith"). For *dlen*, the number of terms under each node, the same choice between dynamic and static counter accumulation has to be made.

2. *Direct storage of counters within existing core indexes, or in additional indexes, or in both.*

tf and *dlen* counters can be stored directly in the T-index or P-index, respectively, or in an additional index structure. Extending a core index structure is more storage efficient. Moreover, for a simultaneous counter accumulation during the processing of the data retrieval parts of a query, direct storage is required. Storing counters in a separate index always requires additional accesses besides accessing the T- and P-index.

3. *Counters of variable length or fixed length as approximations.*

Both *tf* and *dlen* counters may vary in their size and range. For this reason, counters in document retrieval are usually stored with variable-length encodings [68]. However, counters of varying length stored together with μPIDs that are of fixed length in the T- and P-index result in a variable length of all index entries. This eliminates optimization options for data retrieval based on a direct access to node ids. A variable length may also hinder a blockwise storage model that is required to support updates of the structure and text content of a source.

A hybrid solution between options 2 and 3 to support variable-length counters without negatively affecting the core indexes is to split counters. A shorter, fixed-length part is stored directly in the existing indexes. Only for counters that exceed this length, additional entries for the differences to the full counters are stored, for instance, in a *term counter index* (TC-index) added to the T-index. The additional index could then contain entries of varying length without affecting the execution of query portions that only relate to data retrieval.

The rationale behind this seemingly cumbersome approach is the following. Compared to complete documents in standard document retrieval, DFs will likely contain significantly less content, keeping counters small. In fact, we

specifically target XML sources or XML views over textual content with a potentially high degree of structured data. These sources can be expected to contain fewer terms or even only a single term per attribute and element. Thus, with only a minor impact on data retrieval, a counter of only a few bits may be sufficient to directly store most of the counters within the core indexes.

To summarize the options for placing *tf* and *dlen* counters into the core index framework, counters can be

1. Repeatedly stored or accumulated at runtime;
2. Stored directly within the existing core indexes, or in additional indexes, or in both;
3. Variable length or fixed length as approximations.

Variable-length storage within core indexes is not advisable, but it is an option when using an additional ("split") index.

4.5.2 Making the right choice

As our experiments in Sect. 6 indeed show, term counters are mostly very small. Only a few bits k are required within the T-index to represent most of the *tf* counters. Therefore, the preferred option to store term frequencies of a data source is a small extension of the related T-index and an additional TC-index. When approximate counters are sufficient, the TC-index can be eliminated. In what follows, we call a T-index that is extended by k-bit counters $T\text{-}index_{k\text{-}bit}$.

For the other choices in placing counters, it is clear that repeated storage of (accumulated) counters in the T-index (see the above "Smith" example) is hardly an option. Even with nonreplicated counters, the T-index is already larger than it would be in standard document retrieval because term occurrences are with respect to a document's substructures. Therefore, a single term counter as in standard document retrieval is potentially replaced by as many counters as there are nodes in the XDG. Repeated storage makes this even worse because every occurrence of a term at a source tree depth of n would result in up to $(n-1)$ additional counters within the T-index.

For the *dlen* parameter, the above does not hold to the same degree. There are always exactly as many *dlen* counters as there are nodes in a source. These nodes alone make up only a small fraction of the whole content. Therefore, storing the complete DF length related to every source node is acceptable. These counters, however, cannot be stored directly in the P-index because this would prevent the application of the efficient, sparse storage model used in the P-index as described in Sect. 4.4. Hence, there has to be an additional DF counter index (*DC-index*).

In the DC-index, the use of fixed-length counters is a requirement for addressing single counters. The latter is important as data retrieval queries usually address only a relatively small portion of all DFs in a source, and their related counters are likely not stored as a contiguous sequence. Fortunately, finding the right *dlen* counters in the additional DC-index comes for free because the P-index implicitly delivers their positions within the DC-index in the same way as addresses in the A-index.

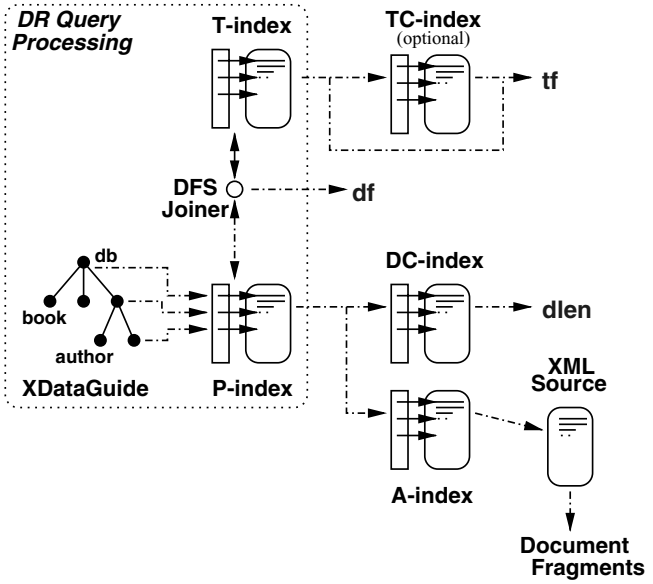

Fig. 7 Indexes and data flow paths: the *dotted box* surrounds the main structures for data retrieval. A-index and source together provide for the retrieval of source content. DC- and (optional) TC-index add support of embedded document retrieval

For this to work, the DC-index entries need to be grouped by XDG node numbers in the same way as the P-index and the A-index. The grouping then has a further positive side effect. Only counters related to a single type of node are stored together. Thus, only as many bits are required to store *dlen* counters as the range between minimum and maximum counter for the related DF implies. When fragments of a certain node type always have the same length, no counters need to be stored at all. As our experiments will show, this happens in more than just a few cases.

4.6 Resulting index framework

Figure 7 summarizes the data flow and access paths for the indexing framework resulting from the discussions in this section. Figure 7 is a more detailed version of Fig. 3 provided at the beginning of this section.

As shown in the figure, query processing for data retrieval relies on an Extended DataGuide (XDG), T-index, and P-index. Tree patterns are identified by joining node ID lists from T- and P-indexes (DFS Joiner). The content of each DF in the final sequence of DFs is obtained from an XML source through the additional A-index.

The document frequency of query terms (*df*) is a byproduct of a query's data retrieval portion. Term frequencies (*tf*) are stored directly in the T-index at a fixed bit length, possibly accompanied by an additional TC-index that complements large counters. TC- and T-indexes share the same structure. A DC-index contains precomputed DF counters (*dlen*) for every level of a source hierarchy. The positional information in the P-index allows for direct access to the length information of DFs.

5 Query processing

In this section, we discuss how the index structures introduced in Sect. 4 are used to process the parts of IIR queries related to data retrieval and embedded document retrieval.

5.1 Overview and notations

In this paper, we focus on the path- and tree-pattern queries discussed in Sect. 4.2, as these build the core of most XML query languages (Sect. 2.3.2). Path patterns do not contain branches and are most easily processed. Tree patterns consist of multiple path patterns. Therefore, the processing of tree-pattern queries can be reduced to the processing of path-pattern queries plus some extra operations.

Following this line of thought, we first discuss the processing of increasingly complex path-pattern queries in Sect. 5.2. In Sect. 5.3, we show how existing approaches to processing tree-pattern queries can easily be adapted to use our index structures. In addition, we outline how the newly introduced node IDs and indexes can provide for an execution of tree-pattern queries that is significantly more efficient than in existing approaches (Sect. 5.3.2). Finally, in Sect. 5.4, we discuss how embedded document retrieval can be implemented within the previously discussed framework.

We use the following functions as formal specifications of accesses to P-, A-, and T-indexes:

$$p\text{-}seq: \; M \rightarrow seq(N)$$
$$a\text{-}seq: \; M \rightarrow seq(N)$$
$$t\text{-}seq: \; T \times M \rightarrow seq(N)$$

p-seq maps a given node number to the list (sequence) of position numbers as found in the P-index. Analogously, *a-seq* maps a node number to a sequence of physical addresses that are represented by natural numbers. *t-seq* has the same functionality as *p-seq* but takes a term as additional argument.

5.2 Processing path-pattern queries

A path-pattern query consists of labeled nodes, possibly containing a term condition, but without branches in its graph representation. As an example, the left part of Fig. 2 shows the path-pattern query //Location/Books//Title. The processing of path-pattern queries depends on the kind of leaf node of the pattern and the location of the selection node.

We distinguish between the following three nonexclusive types of path-pattern queries:

Simple path patterns: patterns without term condition, where the selection node is the leaf node, e.g., the path expression //Location/Books//Title;

General path patterns: patterns with at least one labeled node below the selection node, e.g., //Location/Books[.//Title];

Path patterns with term condition: patterns with a term condition, e.g., //Location/Books//Title[contains(text(), "XML")].

5.2.1 Simple path patterns

Simple path patterns can be processed in the following three simple steps:

1. Find all matches of the pattern in the XDG. Matches are rooted label paths, identified by node numbers $n_1, \ldots, n_r$.
2. For each node number n_i, retrieve the index lists *a-seq*(n_i) *from the A-index*.
3. For each physical address in these lists, obtain the DF from the source.

As the XDG itself is a data source, the first step can be executed by any in-memory query processing algorithm. This can be done in polynomial time [37]. Because a simple path expression selects *all* nodes in a source that match a certain label path, there is no need to access the P-index. The physical addresses are directly retrieved from the address index.

5.2.2 General path patterns

If the selection node in the path pattern is not the pattern's leaf node, as in the example //Location/Books[.//Title], an access to the P-index is necessary. In this case, instead of accessing the A-index, the related lists in the P-index are accessed. Each list related to a node number n_i consists of position numbers of fixed bit length k_i. For the source shown in Fig. 4, k_i is 5 for the only match of the example path pattern because Location nodes require 3 bits and Bk nodes 2 bits.

The selection node in this example is Books, which is below the Location node but above the Bk node. Thus, to obtain node IDs for Books nodes from node IDs for Title nodes, the 3-bit prefix from each position number is extracted. To obtain the index position of the related physical addresses, these 3-bit numbers are joined with the Books-related P-index list. Only this list contains pointers to the physical addresses of Books nodes.

In short, the executed join operation is a semijoin with the common μPID prefix as join attribute A:

$$p\text{-}seq(\text{nodeno}(\text{LP}_1)) \;\ltimes_A\; p\text{-}seq(\text{nodeno}(\text{LP}_2)).$$

Here, LP_1 and LP_2 stand for the rooted label paths /DigitalLibrary/Location/Books and /DigitalLibrary/Location/Books/Bk /Title, respectively.

Notice that the join is necessary. It is not enough to just access Books nodes in the P-index directly because there may be nodes that do not have any Title descendants and thus do not satisfy the query pattern. Moreover, all the joins can be executed in an iterative fashion as described by Graefe [39].

If the example query //Location/Books[.//Title] has more than a single instance of a rooted label path in the queried source, then the above procedure is executed multiple times. For each instance of the pattern, a semijoin is executed based

on node number prefixes according to the position of the selection node.

The above procedure for processing path-pattern queries has a major advantage over related approaches that group node IDs by common, rooted label path [65, 70]. The earlier approaches cannot derive IDs of selection nodes directly. Rather than making the transition from Title to Books nodes directly, they need to execute a join to establish Title–Bk relationships and another join between Bk and Books nodes. This is because of the less expressive node IDs in these approaches. The extra P-index join in our case is only for deriving physical addresses, which is not discussed in the related approaches.

Approaches that do not group node IDs by common label path [3, 46, 71] have another disadvantage. They cannot avoid joining parts of lists for nodes that match only a single label in a pattern, but do not possibly match the whole pattern, e.g., Titles related to .../Location/Articles/.../Title nodes for a query pattern like //Location/Books[.//Title].

5.2.3 Path patterns with term condition

A term containment condition is necessarily a leaf node in a query pattern. If a query pattern contains such a term condition, the above query processing is modified and extended in the following way. First, after determining node numbers in the XDG that match the path pattern, the T-index is accessed instead of P-index or A-index. Then, for all terms $t_1, \ldots, t_m$ in the containment condition, node ID lists for all matching node numbers $n_1, \ldots, n_r$ are iteratively accessed.

If there are multiple terms involved, for each node number n_i, term lists are merged to obtain a single ordered list:

$$\bigcup_{j=1,\ldots,m} t\text{-}seq(t_j, n_i)\,.$$

Different types of term containment conditions can be supported through specialized merging strategies as discussed below. Independent of the merging strategy, the merging can be done in O(n) time where n is the sum of the lengths of all lists. The time estimation relies on the fact that the number of lists is always strictly limited by the number of nodes in the XDG.

With only a single term, the example query //Location/Books//Title[contains(text(), "XML")] changes the query execution plan shown earlier to

$$p\text{-}seq(\text{nodeno}(\text{LP}_1)) \;\rtimes_A\; t\text{-}seq(t, \text{nodeno}(\text{LP}_1))\;,$$

where t is the term number of "XML."

If the term condition is with respect to an inner node within a source, e.g., //Books[contains(text(), "XML")], the list merging is extended to all of the node's descendant nodes:

$$\bigcup_{j=1,\ldots,m} \bigcup_{d \in desc(n_i)} t\text{-}seq(t_j, d)\,.$$

Again, n_i represents node numbers matching //Books. $desc()$ denotes the set of node numbers of descendant nodes of n_i, including n_i. The merging of index lists required for term conditions over descendant nodes is very similar to the procedure used to accumulate term counters in the T-index that is extended by term occurrence statistics, as we will see in Sect. 5.4

For all path-pattern queries, if the final result is supposed to be in document order, all node-number-specific lists are merged into one list during the retrieval of physical addresses. This merging is in principle the same as above but uses document order instead of numerical order of node positions.

The algorithm in Fig. 8 summarizes the query processing algorithm for path patterns, not including a final document order. For presentation purposes, the algorithm is presented as processing whole node ID lists in each step. In the implementation, the processing is pipelined at the granularity of single node IDs.

Algorithm **PathPatternMatching**(Path pattern Q)
Input: Path pattern Q with leaf node ℓ and selection node s
Output: List R of physical addresses of nodes that match Q in s

1. If ℓ is a term condition C:
   (a) let $T := \{t_1, \ldots, t_m\}$ be the terms in C
   (b) let $\ell :=$ parent(ℓ)
2. Find the node numbers $N = \{n_1, \ldots, n_r\}$ in the XDG that match Q in ℓ (ignoring any term condition C)
3. **If** $\ell = s$ then for all n_i in N: append the A-index list for n_i to R; **done**.
   Otherwise continue with 4:
4. For every n in N:
   (a) obtain node id list for n from
      i. P-index, if Q does not have a term condition and thus C is empty; otherwise from
      ii. T-index by *merging* T-index (n, t)-lists for all t in T
   (b) determine the node id list for the selection node related to n:
      i. let n_s be the node (number) in the rooted label path for n that is related to the selection node s
      ii. join node id list for n with list for n_s based on position number prefixes:
      $L := p\text{-}seq(n) \ltimes p\text{-}seq(n_s)$
   (c) obtain physical addresses from A-index; append to result: $R := R \cup (a\text{-}seq(n_s) \ltimes L)$

Fig. 8 Algorithm for processing path pattern queries. Step 1 makes adjustments in case a term containment condition is present; step 2 determines XDG node numbers for pattern matches; step 3 handles the simplest pattern case that requires only A-index access; step 4 obtains node ID lists from P- or T-index including an additional access in case the selection node is some inner node; the merging of step 4(a)ii is detailed in Sect. 5.2.4

5.2.4 Supporting different term containment conditions

We distinguish between four kinds of term containment conditions, which determine the implementation of the merge operator "∪":

Strings	//X[contains(text(), “database system”)]
AND	//X[contains(text(), “database”) and contains(text(), “system”)]
OR	//X[contains(text(), “database”) or contains(text(), “system”)]
Complex	<condition involving AND and OR>

AND can be implemented as elimination of all node IDs from the term-related T-index lists that do not occur in both the “database” list and the “system” list. For OR, these lists have to be merged unconditionally while eliminating duplicates. *Complex* conditions can be implemented accordingly.

String conditions have to be simulated through an AND condition and a later postprocessing. In the above examples, all path patterns that contain both “database” and “system” within an X node have to be retrieved to determine potential pattern matches. Later on, the content of these node has to be searched to determine the actual matches.

Term conditions containing negations, e.g., //X[not contains(text(), “database”)] are more complex to process. They can be implemented through access of node ID lists related to //X in the P-index and then eliminating (//X, “database”)-specific T-index entries from these lists. The cost associated with this implementation depends on the selectivity of //X.

5.3 Processing tree-pattern queries

The standard approach to processing tree-pattern queries through SJs employs one join for each edge in the query pattern [3, 71]. The two input lists for the join consist of node IDs related to a certain node label, not a label path. Therefore, as a basic approach, if one can provide single input lists related to node labels, then the existing join algorithms can be directly applied.

5.3.1 Basic approach

Obtaining a single, ordered list for a node label ℓ in our approach involves two steps:

1. Identify all rooted label paths in the XDG that end in ℓ.
2. Merge the related lists from the P- or T-index, as discussed in Sect. 5.2.

However, there is an obvious optimization that can be applied in step 1. Each pair of labeled nodes that is connected by an edge in a given tree-pattern query trivially relates to a pair of full path patterns. The XDG provides all possible matches for these path patterns in the form of rooted label paths. The matches determine a subset of node numbers, whose related index lists provide all join results that are necessary and sufficient to obtain the full tree pattern. Considering complete path patterns instead of just node labels thus allows the query engine to eliminate some lists of node IDs that cannot possibly have matches with respect to the complete query.

In Fig. 9, consider the tree-pattern query //Document[./Abstract//”XML”]//Author[./Address/Country/“USA”][.//Email],

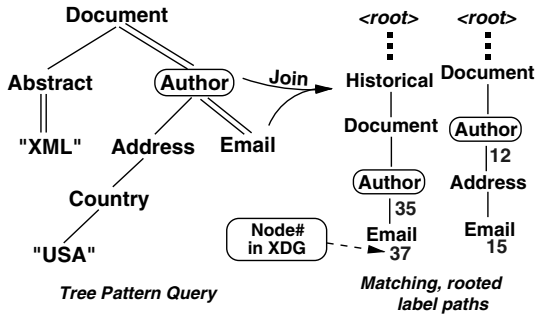

Fig. 9 Tree pattern query //Document[./Abstract//”XML”] //Author[./Address/Country/“USA”][.//Email] and instances of path patterns related to the join between the Author and Email node

which is based on a source similar to the one shown in Fig. 4. To implement the join between the Author and Email nodes in the pattern, the XDG might deliver the two rooted label paths shown on the right side of Fig. 9 as instances of the path pattern //Document//Author[//Email]. Assuming node numbers 15 and 37 for the matching Email nodes and 12 and 35 for Author, the join operation can be restricted to:

$$(p\text{-}seq(12) \cup p\text{-}seq(35)) \rtimes\!\!\ltimes_A (p\text{-}seq(15) \cup p\text{-}seq(37)) .$$

The restriction can result in substantial savings when there are many different label paths that contain nodes labeled Author or Email. Database-centered approaches like that of Yoshikawa et al. [70] keep track of the rooted label path related to every node ID. However, these approaches do not employ a custom-tailored storage scheme that keeps node ID lists sorted, and thus these approaches require an enormous storage overhead. Hence, extensive sorting is necessary to obtain sorted node ID lists [65].

The algorithms for actually executing the join are discussed in detail by Al-Khalifa et al. [3] and Zhang et al. [71]. The *stack-tree* family of algorithms [3] in particular guarantees linear execution time with respect to the number of matching node IDs.

In what follows, we outline a significant further improvement of the basic query processing algorithm presented above.

5.3.2 Further improving the basic approach

μPIDs encode complete, rooted data paths. Therefore, it is not necessary to execute a join for every single edge in a tree pattern. Consider the subpattern Author/Address/Country in Fig. 9. μPIDs that are instances of rooted label paths that match Country directly contain information about the Country nodes’ Author ancestor. Therefore, there is no need to execute any joins with Address nodes to establish relationships between Country and Author nodes. Based on the same argumentation, it becomes obvious that joins are required *only for every branch point* and not for every edge in a query pattern.

For instance, to find all patterns matching the right half of the example query in Fig. 9, it is sufficient to:

- Find matches for path pattern //Document//Author/Address /Country/"USA" in XDG and then T-index,
- Find matches for path pattern //Document//Author//Email in XDG and P-index, and
- Join node ID lists from both indexes at the prefix position related to the Author node (the branch point).

We refer to the logical join at the Author node based on a query's path patterns as a *macro join*. The actual joins are executed over node IDs related to rooted label paths that are instances of these path patterns. We call such joins *micro joins*. A macro join is implemented through one or more micro joins by either of the following two approaches:

1. Merge lists for the two branches in the query pattern into two single lists in document order, and apply a standard structural join algorithm on Author:

$$\bigcup_{i=1,\ldots,r} t\text{-}seq(\text{“USA”}, n_i) \ltimes_{Author} \bigcup_{j=1,\ldots,s} p\text{-}seq(n_j),$$

   where the n_i and n_j are matching node numbers for the two respective path patterns related to the "USA" and Email nodes. Here, the macro join is implemented by a single micro join.
2. Execute several micro joins, but only between lists related to rooted label paths that have a common branch point.

We discuss the second solution in more detail in what follows.

There are three leaf nodes in the example tree pattern, identified by the marked numbers 1, 2, and 3 in Fig. 10. The first step of the enhanced query processing for tree-pattern queries is analogous to the processing of path-pattern queries. Each leaf node has a related path pattern P_i starting at the root of the pattern tree. All instances of these patterns

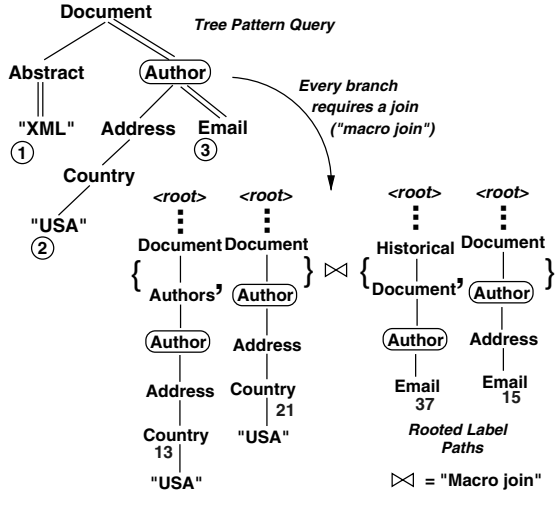

Fig. 10 Improved processing of a tree-pattern query: one *macro join* is executed for every branch point in the pattern. The macro join is implemented by potentially several *micro joins*. Micro joins are joins between lists of node IDs related to rooted label paths that are instances of the path patterns to the right and left of the branch point

can be found in the XDG. Assume the two instances for each of the patterns P_2 and P_3 as shown on the right side of the figure. Each instance is uniquely identified by a node number. Hence all instances related to a query path P_i can be considered a set $P_i = \{p_{i1}, p_{i2}, \ldots, p_{ik_i}\}$, where the p_{ij} are node numbers, overloading the symbol P_i with this second meaning. In the example, we have $P_2 = \{13, 21\}$ and $P_3 = \{37, 15\}$; assume $P_1 = \{5, 8, 9\}$ (the related instances are not shown in Fig. 10).

An instance of the full query pattern can be considered an ℓ-tuple of node numbers $T = (p_1, p_2, \ldots, p_\ell)$, $p_i \in P_i$, where ℓ is the number of leaf nodes in the query pattern. That is, the tuple consists of one rooted label path for each leaf-related path pattern in the query. There are $k_1 \times k_2 \times \cdots \times k_\ell$ potential matches for the query pattern. In the example, we have $3 \times 2 \times 2 = 12$ potential instances of the query pattern. However, notice that the paths related to node numbers 13 and 37 do not match, even though the paths below Author match and both have a Document ancestor above Author. Thus, there cannot be any matching $(x, 13, 37)$ tuple. Consequently, there are other combinations of node numbers that do not match the query tree pattern.

All the valid combinations of node numbers can be determined from the XDG *before* looking at any of their instances in an index list. Although this is an important aspect that can lead to a drastic decrease of join operations, it is not supported by any existing query processing approach. For the running example, a graph representing all valid triples of node numbers might look like the one shown in Fig. 11. A path from left to right through all the groups of node numbers represents a valid query pattern instance in the XDG. Only for these patterns do potential join results for node ID lists exist.

The outlined query processing approach rigorously exploits path summary information from the XDG to avoid joining node IDs that cannot possibly match. However, there is some additional complexity and potential cost attached to this. The number of joins between – now much shorter – node ID lists can increase. This can also mean that a node ID list has to be accessed repeatedly to join it with different matching lists.

However, having the full list of ℓ-tuples of matching node numbers easily available (as shown, e.g., in Fig. 11) together with detailed statistics about list lengths within

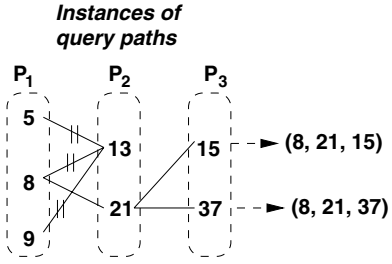

Fig. 11 Instances of path patterns in a query in the form of node numbers, and all possible combinations of node numbers to form a valid instance of the query's complete tree pattern

Algorithm **TreePatternMatching**(Tree pattern Q)
Input: Tree pattern query Q with ℓ leaf nodes and selection node s
Output: List R of μPIDs that match Q in the position of s

1. Determine all leaf nodes in Q (ignoring term conditions)
2. For each of the $1 \ldots \ell$ leaf nodes:
   (a) find all instances of the path pattern from the leaf to the root of Q in the XDG
   (b) let $P_i = \{n_{i1}, n_{i2}, \ldots, n_{ik_i}\}, i = 1, \ldots, \ell$ be the set of node numbers related to these instances
3. For all ℓ-tuples $T = (p_1, p_2, \ldots, p_\ell), p_i \in P_i$:
   (a) determine whether the rooted label paths related to node numbers $p_1, p_2, \ldots, p_\ell$ constitute a tree that matches Q
   (b) while checking this, for every branch point node, look up the total number of bits of its position number in the XDG
4. For every resulting tuple $T' = (p'_1, p'_2, \ldots, p'_\ell)$:
   (a) Fetch the list L for p'_1 from the P- or T-index
   (b) For i = 2 to ℓ:
      i. fetch L_i, the list related to p'_i, from P- or T-index
      ii. if p_i contains the selection node and p'_1 does not, swap L and L_i
      iii. semi-join L_i and L using L as the outer list: $L := L \ltimes L_i$
   (c) Append L to the result list R
5. If result needs to be in document order, merge ordered sublists within R into ordered list

Fig. 12 Basic algorithm for processing tree-pattern queries while considering only possible matches based on information available in the XDG

index structures provides for promising optimization opportunities. Furthermore, the sparse storage of path index entries (Sect. 4.4) reduces the number of list elements to be loaded throughout the join process, as demonstrated in Sect. 6. Sparse storage also provides for what we call *virtual joins*. Virtual joins are SJs that are executed over the range of node IDs represented by a single element in a sparse P-index list without expanding the element. By means of virtual joins, many more node IDs can be constantly cached in main memory, eliminating disk accesses.

The basic algorithm in Fig. 12 summarizes the improved query processing for tree-pattern queries as outlined in this subsection.

5.4 Processing document retrieval subqueries

The processing of document retrieval subqueries requires the accumulation of term counters, which, as we assume here, extend the T-index directly (Sect. 4.5). The actual work in doing so is not the accumulation, but getting to the right counters within the T-index. This requires the same joins and, thus, the same procedure as processing path-pattern queries with term conditions, which we discussed already in Sect. 5.2.3. Here, we build on this earlier discussion.

Algorithm **EmbeddedDocRetrieval**(DFS S, Terms T)
Input: Document fragment sequence S (that is grouped by node number and sorted); terms $T = \{t_1, \ldots, t_m\}$
Output: Lists $R_1, \ldots, R_m \in \mathbf{N}^{|S|}$ of *tf* accumulators for each term

For each term $t_i \in T$:

For each group $S_j \subseteq S$ of node ids with common node number:

1. let n be the node number of nodes in S_j
2. let R_{ij} be the sub-list of R_i related to S_j (the elements of R_{ij} are the counters for the nodes in S_j)
3. for all node numbers m that are descendants of n:
   if there is a T-index list for t_i and m, execute:
   $\begin{pmatrix} R_{ij} \\ S_j \end{pmatrix} \ltimes$ *t-seq*(t_i, m),
   accumulating counters in R_{ij} while joining their related nodes in S_j with T-index elements

Fig. 13 Accumulation of *tf* counters for a DFS and a set of terms. The accumulation is executed separately for each term and sub-DFS with common node number. The semijoin with the combined lists S_j and R_{ij} works as follows: S_j is joined with related T-index lists, and for each match at position p in S_j the term counter in the T-index list is added to the counter at position p in R_{ij}

As input to our algorithm, we assume a sequence S of node IDs that are grouped by node number, delivered by the preceding data or document retrieval subquery. For such a sequence, document fragment frequency (df) and fragment length ($dlen$) are computed as outlined in Sect. 4.6: for each node ID, $dlen$ can be found in the DC-index, and df is a byproduct of the data or document retrieval subquery preceding the current document retrieval subquery. The term frequency (*tf*) can be obtained for each query term t as described in the following and summarized in Fig. 13.

A counter with initial value zero is created for each node ID in S. Then, for each element in S, counters for all descendant nodes found in the T-index under term t have to be accumulated. This operation can be implemented as k joins between lists in the T-index and S. Here, k is the number of node types t occurs in. The ability to limit the joins to T-index lists specific to term t greatly reduces the counter accumulation process. In a further optimization of this procedure, S can be sorted by node and position number. Then the joins need only be executed between sublists of S with the same node number and lists in the T-index that contain possible descendant nodes with respect to this node number. The same procedure is repeated for each term in the document retrieval query.

In a more formal fashion, counter vectors R_i and fragment sequence S have the following form:

$$R_1 = \left(\overbrace{r_{11}, r_{12}, r_{13}, \ldots, r_{1q}}^{R_{11}}, \overbrace{r_{1(q+1)}, \ldots}^{R_{12}}, \ldots \overbrace{\ldots}^{R_{1k}} \right)$$

$$R_2 = (r_{21}, r_{22}, \ldots)$$

$\ldots$

$$S = \left(\overbrace{\underbrace{f_1, f_2, f_3, \ldots, f_q}_{S_1}}^{node\#\, n_1}, \overbrace{\underbrace{f_{q+1}, \ldots}_{S_2}}^{n_2}, \ldots, \overbrace{\underbrace{\ldots}_{S_k}}^{n_k} \right).$$

Let N be the total number of nodes in the underlying source and n the total number of nodes in the XDG. Then, a rough estimation of the complexity C of the counter accumulation for each term t_i looks as follows:

$$\begin{aligned} C = \; & |S_1| + \sum_{m \in desc(n_1)} |t\text{-}seq(t_i, m)| \\ & + |S_2| + \sum_{m \in desc(n_2)} |t\text{-}seq(t_i, m)| \\ & + \quad \ldots \\ & + |S_k| + \sum_{m \in desc(n_k)} |t\text{-}seq(t_i, m)| \\ = \; & |S| + \sum_{j=1}^{k} \sum_{m \in desc(n_j)} |t\text{-}seq(t_i, m)| \\ \leq \; & N + \sum_{j=1}^{k} \sum_{m=1}^{n} N = N + knN \\ \leq \; & N + n^2 N \sim n^2 N \,. \end{aligned}$$

For every term, every sublist S_j of S (with related node number n_j) is joined with all T-index lists of descendant nodes of n_j in the XDG. Each join requires that every element involved be visited only once because the lists are ordered. While the resulting effort of $O(n^2 N)$ seems expensive, the grouping of lists by node type provides for a reduced joining effort. This fact, however, is not covered by the general estimation that as an upper bound assumes a length of N for every node-type-specific index list and the maximum of n different node types in index lists and fragment sequence.

Some of the expenses involved are unavoidable and present a price that has to be paid for the additional power of ranking arbitrary fragment sequences. In practice, the expenses are bound by the limited number of fragments that contain a term. Furthermore, there are typically only a few node types that carry most of the text content in a source. This reduces the number of joins to be executed and leads to an acceptable performance, as the experiments in Sect. 6 suggest.

6 Evaluation

In this section, we experimentally analyze various properties of our node identification and indexing scheme. We discuss core index structures for data retrieval and extensions that are required for a full support of integrated information retrieval (IIR). In addition, we compare the index structures for data retrieval introduced in this paper with related index structures of earlier approaches.

6.1 Implementation

6.1.1 Index structures

We have implemented the path index (P-index) and term index (T-index) introduced in Sect. 4 as indexed sequential files. Both index structures utilize a bit-accurate storage of μPIDs. μPID position numbers are grouped by node numbers, which refer to nodes in the Extended DataGuide (XDG). μPIDs are arranged in document order. No overall compression is applied to these lists of node IDs. In what follows, the term μPID usually refers to position numbers that are part of a list of position numbers. Only the complete list is preceded by the related XDG node number.

For comparison with earlier structural join (SJ) approaches that exclusively rely on the interval (Ival) node identification scheme [2], we have also implemented P- and T-index based on Ival node IDs.

> **Ival node IDs** consist of a triple of integers (id, *maxid*, *depth*). id is a node's position in a preleft order walk through a data source, *maxid* is the largest node ID of any of the node's descendants, and *depth* is the depth of the node within the source. In the Ival scheme, ancestor–descendant relationships between nodes can be determined based on the containment of the nodes' [id, *maxid*] intervals. Parent–child relationship require an additional comparison between the depths of the nodes.

In order to make the index sizes for the Ival scheme comparable to those for the μPID scheme, we store Ival IDs using only the required number of bits. A source with n nodes and a maximum tree depth of d requires $2 \times \lceil \log_2(n) \rceil + \lceil \log_2(d) \rceil$ bits for storing interval bounds and tree depth of an Ival ID. As a result of this dense storage, our index implementation for the Ival scheme is significantly more storage efficient than the best implementation reported so far [71]. Improvements to the reported basic implementations only extend the indexes further [17, 21]. Furthermore, when stored as relations in a relational database, a multiple of the storage space of custom-tailored index structures is required [65, 71]. Therefore, the Ival-related index sizes that are reported in this paper can be considered lower bounds for the whole class of Ival-based SJ indexes.

While sparse storage as discussed in Sect. 4 is an important aspect of our index structures, it is not used for the Ival-scheme indexes. In theory, sparse storage can be used for any monotonic sequence of integers, e.g., interval bounds in the Ival scheme. However, because the numbers are assigned to nodes in one global tree traversal in the Ival scheme, hardly any contiguous sequences of numbers arise. Therefore, sparse storage would certainly increase the space required to store lists of Ival IDs related to, for instance, a particular rooted label path.

For the μPID approach, we also implemented the address index (A-index) and report its size. Like the P- and T-index, our A-index implementation uses no compression.

Therefore, the size of the A-index depends exclusively on how the source content is stored and, thus, the kind of physical identifiers used. In our XML-based implementation, physical identifiers are bit-accurate offsets into an efficiently compressed XML source. The source compression is similar to that reported by Tolani and Haritsa [67] and reduces sources to roughly a third of their uncompressed size. Offsets into a compressed source are about the smallest possible physical identifiers. Therefore, the reported A-index size represents a lower bound for such a mapping structure.

For the A-index, a simple storage optimization similar to the one discussed at the end of Sect. 4.4 for the P-index is used. The A-index does not contain address lists related to attribute nodes, whose parent nodes always contain this attribute. In this case, the physical address of the attribute node can be easily obtained from the parent's address in our implementation because attributes directly follow their parent nodes in the compressed source.

Because of its small size, we assume that the XDG can be kept in main memory. Even the largest XDG in our experiments required only a few kilobytes. Therefore, an XDG of even tens of thousands of nodes can be kept in main memory.

For the support of embedded document retrieval, we also implemented a DC-index and extended the core T-index by a term counter of just a single bit (Sect. 4.5). Counters larger than one have an additional entry in the TC-index. So far, the TC-index always uses a whole byte for every counter. Therefore, in practice, the TC-index size can be considered an upper bound for the size of any TC-index. The DC-index is equivalent to the A-index except that document fragment (DF) counters replace physical addresses. The length in bits of these DF counters is determined by the minimum and maximum number of terms found under a certain node type. Node types that always have exactly the same number of terms do not need any entries in the DC-index. This frequently applies to attribute nodes containing just a single term, but also to some element nodes and nodes with more than one term.

6.1.2 Query processing

Query processing is based on iterators [39], which directly operate on lists of node identifiers kept in the index structures described above. We have implemented the system described in this paper in Java, version 1.4, up to the operation of iterators over node ID lists. Caching is employed for the list elements at the current head of each iterator. The default size of this cache is 4 Kb. We observed no significant influence of the size of the cache on query processing as long as it stayed above a few kilobytes. No further application caching is used.

While it is beyond the scope of this paper to implement a query system that fully realizes XQuery, advanced query processing (Sect. 5.3.2) is operational for the examples presented in Sect. 6.5. Performance of document and data retrieval portions solely depends on the efficient processing and joining of node ID lists. Consequently, this is the focus of the following evaluation. For the execution of arbitrarily nested data and document subqueries – the main focus of our work – no other benchmark exists so far. Furthermore, because the conceptual document retrieval extension we have introduced in this paper is independent of the weighting scheme used for ranking, it is not our intention to implement or promote any particular weighting scheme.

Table 1 *Big10* source composition: absolute and relative contributions and average number of words per element (attribute) node of included sources

Name	Size (Mb)	Size(%)	Average words
LA Times	475.3	38.2	14.7 (1.2)
Congr. Record	252.3	20.3	71.6
Federal register	238.7	19.2	82.8
DBLP	127.7	10.3	3.4 (3.7)
SwissProt	109.5	8.8	1.7 (1.7)
NASA	23.9	1.9	4.2 (2.7)
Shakespeare	7.3	0.6	5.1
Religion	6.7	0.5	25.6
Mondial	1.7	0.1	0.5 (1.9)
Total	*1243.1*	*100*	*21.0 (2.2)*

6.2 Data sources

6.2.1 Overview

In what follows, we present experimental results on a large number of publicly available XML sources of up to 1.3 GB. Some of these sources are constructed from standard TREC (Text Retrieval Conference[1]) disk 4 and 5 document collections. The document collections are converted into a single XML source by introducing an artificial root node. The *XMark 1Gb*, *XMark 500*, and *XMark 100* sources are generated by the XMark [62] project's *xmlgen* tool using size scaling factors 1, 4.2, and 10. *Reuters* refers to disk 1 of the Reuters Corpus [54]. Compiled into a single source, we include the very small *Shakespeare* play[2] source to allow for a comparison with the main experimental results reported in many related works.

Except for *Big10*, the rest of the sources originate from the University of Washington's XML Data Repository.[3] The *Big10* source is a composite source consisting of a number of diverse XML sources from the sites mentioned above. Details are given in Table 1. Through *Big10* we tried to simulate a large, diverse, text- and structure-rich XML source. Such a source is what we consider the main target of our approach – and indexing and query processing for semistructured data in general.

[1] `trec.nist.org`
[2] Formatted in XML by Jon Bosak
[3] `www.cs.washington.edu/research/xmldatasets/`

Table 2 General XML source statistics including total size (*Size*), total number of nodes (*Nodes*) and the percentage of attribute nodes (*attribs.*), total number of word tokens (*Terms*) in millions and number of unique terms (*unique*), average number of nonunique terms that directly occur in an element node (*Avg. terms*) and attribute node (in parentheses), number of distinct rooted label paths (*Paths*) and number of unique node labels, and, finally, the maximum depth of the source tree

Name	Size [Mb]	Nodes (attribs.)	Terms (unique)	Avg. terms	Paths (labels)	Depth
Big10	1243.1	16,235,910 (20%)	177.7 m (772,336)	13.2 (1.9)	946 (350)	9
Reuters	1354.0	37,864,292 (51%)	174.5 m (313,755)	6.9 (2.4)	27 (26)	7
XMark 1Gb	1118.0	20,532,978 (19%)	126.0 m (47,537)	7.0 (2.5)	548 (83)	12
XMark 500	469.4	8,631,135 (19%)	52.5 m (47,464)	7.0 (2.2)	548(83)	12
XMark 100	111.1	2,048,193 (19%)	12.4 m (46,235)	7.0 (2.2)	548 (83)	12
Financial Times	564.1	2,847,870 (0%)	92.0 m (396,571)	32.3	17 (17)	3
LA Times	475.3	5,472,913 (10%)	72.9 m (250,560)	14.7 (1.2)	71 (28)	7
DBLP	127.7	3,736,406 (12%)	12.9 m (412,267)	3.4 (3.7)	145 (40)	6
SwissProt	109.5	5,166,890 (42%)	8.6 m (136,950)	1.7 (1.7)	264 (99)	5
Shakespeare	7.3	179,609 (0%)	0.9 m (23,076)	5.1	58 (22)	5

Table 3 Node identifier lengths and index sizes for path, term, and address indexes using the μPID and the Ival node identification schemes, in bits and megabytes, respectively; for μPID-related indexes, the size relative to that of the Ival index is given in parentheses

	Path index [Mb]		Term index [Mb]			Node ID length [bits]			
Name	Ival	μPID (%Ival)	Ival	μPID (%Ival)	Addr. index [Mb]	Ival	μP_{max} (P#)	$Pos\#_{avg}$	(P/T)
Big10	106.5	23.1 (22%)	747.5	347.1 (46%)	51.9	52	51 (41)	22.5	24.4
Reuters	261.8	55.9 (21%)	1,113.8	519.1 (47%)	70.2	55	34 (29)	25.7	26.9
XMark 1Gb	139.5	24.2 (17%)	844.8	306.4 (36%)	63.8	54	39 (29)	21.1	19.8
XMark 500	56.6	9.5 (17%)	339.7	127.2 (37%)	25.1	52	38 (28)	20.0	18.8
XMark 100	12.0	2.0 (17%)	72.0	31.9 (44%)	5.6	46	36 (26)	17.9	16.6
Financial Times	16.6	0.9 (5%)	305.0	115.0 (38%)	10.5	46	26 (21)	20.1	18.0
LA Times	33.9	5.7 (17%)	386.2	216.5 (56%)	18.2	49	48 (41)	27.3	29.1
DBLP	22.3	5.1 (23%)	78.5	35.7(45%)	11.4	35	34 (26)	20.2	19.2
SwissProt	32.0	10.3 (32%)	54.0	25.8 (48%)	9.9	49	39 (30)	21.5	22.8
Shakespeare	0.9	0.3 (37%)	4.6	3.2 (70%)	0.5	39	35 (29)	26.2	28.2

6.2.2 Main statistics

Table 2 summarizes the main properties of the test sources. In the table, *Terms* refers to alphanumeric character sequences that are delimited in a standard way [8] by spaces, punctuation, etc. Terms are converted to lowercase, but no stemming [8] is applied. Number tokens are broken up beyond the standard tokenization into groups of four digits, a common technique in document retrieval [68]. Furthermore, apparently unique identifiers like "person7512" in the XMark sources are split up into a constant part, which still allows one to distinguish the identifier from a regular "person" term, and a number part.

The *Avg. terms* value, the number of unique paths (*Paths* column, equivalent to the number of nodes in the XDG), and the number of distinct node *labels* serve as an indicator of the structural complexity of a source.

6.2.3 Analysis

The *XMark* sources are in general structure rich and deep but contain relatively few terms. This is typical for structured data. At the other extreme, *Financial Times* and, to a lesser extent, *LA Times* as standard text document collections consist of long sections of text with little structure. The total number of XML element and attribute nodes is relatively small for these sources. Furthermore, despite its size, *Financial Times* is very shallow.

SwissProt stands out for its high percentage of structural components with little content. Terms are largely numbers and scientific notations. *Reuters* contains little structure but, with over half of all nodes, the largest percentage of attribute nodes. *Big10* combines properties of structured data and standard document collections, although it is dominated by its biggest contributors, in particular *LA Times*.

6.3 Core index structures

Table 3 shows the sizes of the core index structures for data retrieval for the μPID and Ival scheme. Furthermore, the bit lengths of their respective node identifiers are compared. The length of Ival identifiers (*Ival*) is constant for each source. For the μPID scheme, the total length (μP_{max}) and the length of the position number (*P#*) are listed. Sequences of only the position numbers are stored repeatedly in the indexes. Therefore, they are the main contributors to index sizes. However, their lengths vary with node types. Therefore, Table 3 also lists the average length of position numbers over all index elements ($Pos\#_{avg}$) for P-index (left) and T-index (right). In addition to Table 3, Table 4 presents relative sizes of the index structures with respect to the size of the uncompressed XML source.

Table 4 *Relative* (%) index sizes with respect to the XML source

	P-index		T-index		
Name	Ival	μPID	Ival	μPID	A-index
Big10	8.6	1.9	60.1	27.9	4.0
Reuters	19.3	4.1	82.3	38.3	5.2
XMark 1Gb	12.5	2.2	75.6	27.4	5.7
XMark 500	12.1	2.0	72.4	27.1	5.3
XMark 100	10.8	1.8	64.8	28.7	5.0
Fin'l Times	2.9	0.2	54.1	20.4	1.9
LA Times	7.1	1.2	81.3	45.6	3.8
DBLP	17.5	4.0	61.5	28.0	8.7
SwissProt	29.2	9.4	49.3	23.6	9.0
Shakespeare	12.3	4.1	63.1	43.8	6.8
Average	*13.2*	*3.1*	*66.5*	*31.1*	*5.5*

6.3.1 Node identifiers

Although μPIDs encode more information than Ival IDs, their maximum length is always below that of the Ival scheme. Still, only the μPIDs' position numbers, which are just one part of a μPID, are repeatedly stored in P- and T-indexes. Furthermore, the adaptive length of μPIDs significantly reduces their average length. This is because inner nodes have shorter related paths, and thus identifiers can be far shorter than the maximum length.

It is interesting to note that the effectiveness of the μPID scheme increases (a) with the size of the XML source and, in particular, (b) with an increasingly complex structure. This is because (a) increasingly larger sources are, overall, more regular as the *XMark* sources show. Sources just get larger, but their structure does not become more complex when their size increases beyond a certain point. (b) The XDG effectively encodes information about the structure outside the index, eliminating redundant storage of structure information with every index entry.

However, as *Shakespeare* shows, even a small source can have relatively long μPIDs. This holds when the source is relatively deep with a few node types but many nodes per document fragment. This is only typical for document data with very small units of text. We discuss further aspects of the μPID scheme throughout this section.

6.3.2 Path index

The size of the μPID P-index relative to the size of the related XML data source is about 2–4% for most sources, as shown on the left side of Table 4. This means that arbitrary tree patterns of labeled nodes can be discovered in, for instance, the 1.2-GB *Big10* source, using a path index structure of only 23 Mb. At that size, the P-index can serve as a pure main memory structure, even on an off-the-shelf desktop computer.

In comparison, for the Ival scheme, an average of four times as much storage space is required, although less information useful for query processing is encoded. Nevertheless, this size of Ival P-indexes in our implementation is only a fraction of the size reported in earlier work [71], especially for indexes stored in a relational database [65].

The size of the P-index depends mainly on the number of nodes per content unit and the regularity of a source, as is clearly visible in Tables 2–4. For instance, *Financial Times* and *LA Times* have the lowest number of nodes for their size and, therefore, the relatively smallest P-indexes. In addition, the sparse storage of P-index elements ensures a sublinear growth of the P-index with an increasing number of nodes. Figure 14 provides more insights into the effects of sparse storage.

First, the figure shows around 50% savings in storage space through sparse storage compared to full storage. These savings are realized even though no index fields need to be stored when all elements are stored explicitly. In the worst case, sparse storage can obviously be more wasteful than full storage because of these index fields. Hence sparse storage with its additional index field is effective at providing a mapping from logical to physical node identifiers and still reduces storage space.

The second value shown in Fig. 14 (*Stored elements*) represents the number of elements actually stored vs. the total number of elements represented. As for the relative storage space, *Financial Times* has by far the lowest value here because of its very regular structure. Only 7% of all nodes need to be explicitly represented in the P-index of *Financial Times*. For most sources, this value is about 30%.

A more detailed comparison between the stored and total number of elements for each list within the P-index is provided in Fig. 15 for *Big10*. *Big10* comprises 946 node numbers, i.e., distinct rooted label paths. For most of their related lists, only a small portion of elements needs to be stored (the figure does not show the full height of some of the spikes in the graph). In total, only about 30% of the represented elements are explicitly stored. Of the 946 stored lists, 163 consist of only a single element, even though there is more than one related node. The figure also shows that a small number of label paths comprise the majority of nodes (rooted data paths). These are likely leaf nodes bearing the main content. Another reason for some dominant spikes in Fig. 15, however, is the fact that a few sources contribute the majority of content to the composite source *Big10*.

The third and rightmost value in Fig. 14 (*single lists*) depicts the relative number of lists in the P-index that consist of only a single element, although the total number of elements represented is larger than one. As with the two other statistics, *SwissProt* profits least from sparse storage because it is both very structure rich and irregular.

6.3.3 Term index

T-indexes are in general much larger than P-indexes because information needs to be stored for multiple terms per node. Compared to the equivalent Ival index, still savings of roughly 50–60% can be observed in Table 3. Structural complexity combined with a large number of distinct terms per node determine the size of a T-index.

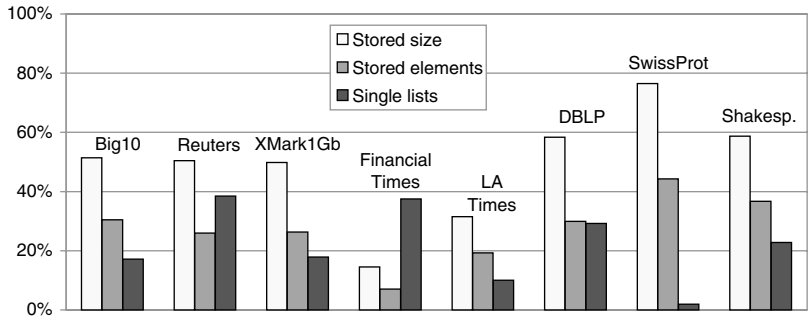

Fig. 14 Storage space for sparse storage compared to full storage (*stored size*), elements actually stored (*stored elements*), and number of P-index lists with only a single entry (*single lists*) for the P-index of eight of the test sources

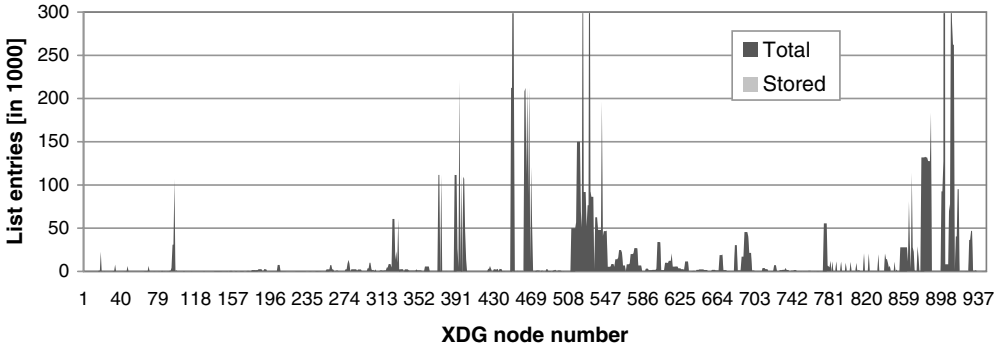

Fig. 15 Total vs. actually stored P-index entries for all lists of *Big10*

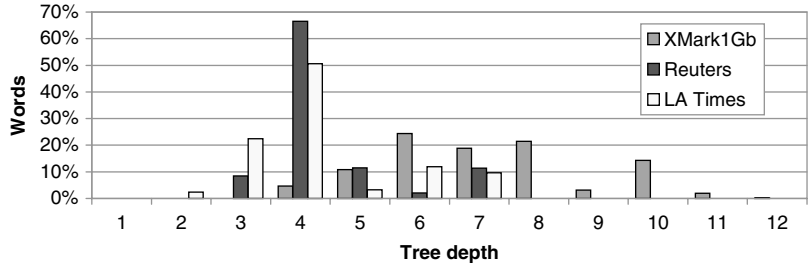

Fig. 16 Textual content at a certain tree depth for *XMark 1Gb*, *Reuters*, and *LA Times*. Content closer to the root relates to shorter μPIDs within the T-index

Furthermore, terms that occur closer to the root node require only relatively short μPIDs to be stored in the T-index. Figure16 shows that much of the textual content occurs significantly below the maximum source depths. Therefore, as for the P-index, the variable length of μPIDs makes storage more efficient.

Figure 17 presents details about the length of μPIDs as they are stored in the T-index. The average length of their position numbers is well below their maximum length (Table 3) because large parts of the textual content require only relative short position numbers. This is especially clear for *Financial Times*.

There are other aspects of a term index that can significantly decrease the size of the index. To name just one prominent example, it is well known in document retrieval that the size of any term index is mostly determined by a few high-occurrence terms like "the," "a," or "it" [8]. These so-called *stop words* are typically left out of the index as they have little value in queries, but greatly increase the index size [68]. In data retrieval, stop words have little value as well. Therefore, stop words can be eliminated from the T-index.

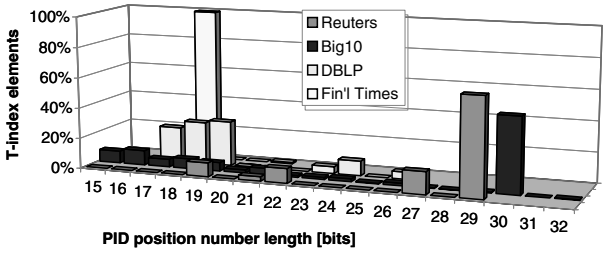

Fig. 17 T-index entries per μPID position number length for four sources. Significant portions of the content requires position numbers of a length well below their maximum length

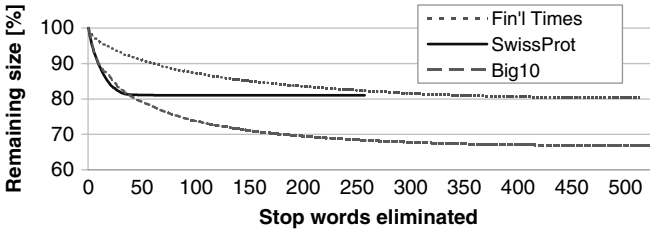

Fig. 18 T-index size on the elimination of the n most frequent stop words out of a list of nearly 600 common English stop words

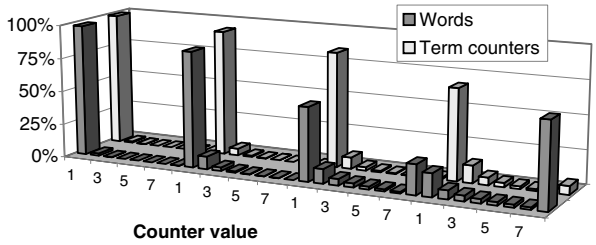

Fig. 19 Term counters and total words per counter value from value 1 to 8 for each of *SwissProt*, *Reuters*, *Big10*, *Congr. Record* (*left* to *right*)

Not used so far, this would lead to savings in storage space ranging from 20% for *SwissProt* with its little textual content to 34% for *Big10* (Fig. 18). The savings are achieved by eliminating about 300 of the most frequent stop words out of a list of nearly 600 standard English stop words. The 50 to 100 most frequent stop words show the greatest effects.

6.3.4 Address index

The size of an A-index mainly depends on two parameters of a source: total number of nodes and source size. Every node implies a physical address stored in the A-index. Bigger sources require longer physical IDs. The A-index sizes as shown in Table 3 reflect these facts. The A-indexsizes are mostly around 5% of the XML source. Only for the most structure-rich *DBLP* and *SwissProt* sources is a higher percentage required to store their A-index. The opposite is true for *Financial Times*, because this source contains relatively little structure. Furthermore, the selective elimination of A-index lists related to attributes as discussed above cuts the index size in half for *Reuters*. For other sources, savings are smaller but still significant.

6.4 Extended index structures

Table 5 provides insights into the storage space required for adding support of embedded document retrieval to the core index structures. In addition, the table repeats the sizes of the core index structures and presents the total size of all, now extended, index structures.

For all data sources, the total size of all indexes combined is about 50% or less of the size of the related source. This is remarkable as all earlier index structures have reported index sizes that exceed the size of the indexed source, although they support only data retrieval.

Extending the T-index by only a single bit as term counter requires a TC-index access for all counters greater than one. However, even a single bit is sufficient to represent most counters, as Fig. 19 clearly shows. For the *SwissProt*, *Reuters*, *Big10*, and the TREC *Congr. Record* sources, the background of the figure shows the distribution of counter values. Only counter values from one to eight are shown. Under counter value eight, all counters larger than seven are combined. The more text rich a source is, the more counters of larger value need to be stored. In the foreground of the figure, the total number of terms contributing to a counter of a certain size is shown for the four XML sources. Together, both graphs can serve as a signature of a source's content type.

Traditional document collections have a typical signature like *Congr. Record* on the right side of the figure. For *Congr. Record*, about 70% of the counters are one, although most terms go into counters of larger value. Thus, 70% of all counters can be accurately represented in the T-index$_{1\text{-bit}}$ that uses only 1 bit to represent term counters. On the other hand, data sets like *SwissProt* on the left side of Fig. 19 have a clear peak of the term *and* term counter distributions at a value of one. Ninety-nine percent of the counters and 98% of the terms can be found at value one. For *Big10*, the numbers are somewhere in the middle between the two extremes. Eighty-five percent of the counters and 54% of the terms can be found at counter size one.

Such term counter distributions suggest that the most effective support for embedded document retrieval is realized by extending the T-index by an n-bit counter for a small n and leaving out the TC-index all together. If n is small, the overhead for regular data retrieval queries is minimal. Every bit added to the T-index increases the index size by roughly 4%. That is, for example, about 14 Mb for *Big10* and 1.5 Mb for *DBLP*. This overhead is always below the overhead introduced by the TC-index, which duplicates μPIDs in addition to storing counters.

The size of the DC-index is generally very small. Due to the relatively low variance in DF lengths, DF counters can be stored in a few bits or even left out completely. This is very effective for keeping the DC-index size at a negligible minimum as Table 5 shows.

6.5 Query processing

We evaluate the query performance of our approach by means of the example tree patterns given in Table 6. The tree patterns are represented as XPath expressions on different data sources. Unlike earlier approaches, the processing of path pattern queries – no matter how complex – is a simple iteration through index lists in our approach. This is the fastest way to process such queries and more efficient than the execution of several joins, one for each edge in a pattern.

Table 5 Sizes of index structures for data retrieval (P-index, A-index, T-index) and extensions for document retrieval (DC-index, TC-index, single bit in T-index), in megabytes and percentage of source size; the total does not include the size of the TC-index

Name	P-index	A-index	DC-index	T-index_{1bit}	TC-index_{max}	Total
Big10	23.1 (2%)	51.9 (4%)	9.5 (1%)	360.6 (29%)	83.3 (7%)	533.7 (43%)
Reuters	55.9 (4%)	70.2 (5%)	16.2 (1%)	538.3 (40%)	46.9 (3%)	680.6 (50%)
XMark 1Gb	24.2 (2%)	63.8 (6%)	8.6 (1%)	325.4 (29%)	23.0 (2%)	422.0 (38%)
XMark 500	9.5 (2%)	25.1 (5%)	3.5 (1%)	137.0 (29%)	17.2 (4%)	175.1 (37%)
XMark 100	2.0 (2%)	5.6 (5%)	0.8 (1%)	35.7 (32%)	10.5 (9%)	44.1 (40%)
Fin'l Times	0.9 (0%)	10.5 (2%)	2.0 (0%)	121.1 (22%)	41.4 (7%)	134.5 (24%)
LA Times	5.7 (1%)	18.2 (4%)	4.6 (1%)	224.7 (47%)	34.5 (7%)	253.2 (53%)
DBLP	5.1 (4%)	11.4 (9%)	1.9 (1%)	37.2 (29%)	7.7 (6%)	55.6 (44%)
SwissProt	10.3 (9%)	9.9 (9%)	1.5 (1%)	26.8 (25%)	2.9 (3%)	48.5 (44%)
Shakespeare	0.3 (4%)	0.5 (7%)	0.1 (1%)	3.3 (45%)	0.4 (5%)	4.2 (58%)

Table 6 Example tree patterns in the form of path expressions for different data sources; *condition*, *selection*, and *result* denote the number of instances related to the condition and selection paths, and the result, respectively. <term> here stands for "macht," "market," and "stockmarket" in queries Q_5, Q_6, and Q_7, respectively. The terms occur in up to 197 different node types

Source	Query	Path expression	Condition	Selection	Result
Shakespeare	Q_1	//play[contains(./title, "Cleopatra")]/personae/persona	1	700	10
	Q_2	//play[contains(./title, "Cleopatra")]//persona	1	969	35
DBLP	Q_3	//article[contains(./author, "Abiteboul")]/title	49	111,609	49
	Q_4	//article[./author]/title	221,465	111,609	110,532
Reuters	Q_{5-7}	/reuters/newsitem/text[contains(., "<term>")]/p	4/270/276,899	5,673,107	4/270/237,115
SwissProt	Q_8	//entry[./features/metal]//author	18,817	566,308	41,519
Big10	Q_9	//entry[./features/metal]//author	18,817	566,308	41,519
XMark 1Gb	Q_{10}	/site/regions/asia/item[./mailbox/mail/text/keyword]	11,609	20,000	11,609
	Q_{11}	//*[contains(., "money")]	133,494	20,532,978	133,224

Table 7 Query evaluation times for the queries from Table 6. For our implementation, every time is given as the time of the first execution (*first*) and the average over 10 to 100 consecutive runs. *Time w/o* and *Time w/* denote the time excluding (w/o) or including the fetching of the related text content from the respective sources. Times for the other systems are comparable with *Time w/* and reported for all queries that did not run out of memory before completion

		Shakespeare		DBLP		Reuters			SwissProt	Big10	XMark 1Gb	
		Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{10}	Q_{11}
Time w/o	*first*	33 ms	40 ms	80 ms	330 ms	1.1 s	1.4 s	8.4 s	1.5 s	0.8 s	80 ms	4.5 s
	avg	1 ms	2 ms	22 ms	250 ms	1.0 s	1.4 s	4.5 s	0.5 s	0.5 s	15 ms	3.6 s
Time w/	*first*	100 ms	130 ms	100 ms	5 s	1.2 s	1.5 s	62 s	3.0 s	2.8 s	800 ms	54.7 s
	avg	**13 ms**	**18 ms**	**25 ms**	**5 s**	**1.1 s**	**1.5 s**	**61 s**	**1.7 s**	**1.7 s**	**490 ms**	**54.4 s**
Xalan DOM	*avg*	3.9 s	4.3 s									
Saxon	*avg*	1.8 s	1.5 s									
X-Hive	*avg*	1.8 s	1.9 s									

The path pattern query Q_{10} demonstrates the effectiveness of our approach even on a gigabyte-size source.

Consequently, all other queries contain a branch point to allow for a meaningful evaluation of query performance. Table 6 presents the cardinality of the input related to both of the resulting branches. Each of these branches may relate to multiple node numbers. In this case, multiple joins are executed and the results are merged afterwards.

For example, the selection branch in query Q_2 resolves to the two rooted label paths .../play/personae/persona and .../play/personae/group/persona with node numbers 8 and 10, respectively. The core execution plan for Q_2 is:

$$[\textit{t-seq}(\text{“Cleopatra''}, 3) \bowtie \textit{p-seq}(8)]$$
$$\cup[\textit{t-seq}(\text{“Cleopatra''}, 3) \bowtie \textit{p-seq}(10)]$$

In the plan, 3 is the node number for .../play/title. "$\bowtie$" denotes the sequential matching of node IDs from the two input lists. This query execution plan represents the most straightforward implementation of the algorithm presented in Fig. 12. For every matching combination of node numbers, the pattern instance is reconstructed. Options to speed up query execution, such as merging the lists for node numbers 8 and 10 first, additional index structures like the small index for each list as shown in Fig. 7, or an optimized ordering of joins, would improve execution times further. However, these options are not yet supported by our system.

Table 7 presents query evaluation times for all queries on an IBM Thinkpad T22 Notebook computer with an Intel Pentium 3 processor running at 900 MHz, 384 Mb DRAM, and a 20-GB (relatively slow Notebook) hard drive. Each query is run several times consecutively. Reported are the

times for the first (cold started) execution and the average over all executions. Furthermore, times are reported including the fetching of content into a Java StringBuffer (*Time w/*) and not including this content retrieval (*Time w/o*). In the latter case, the physical address is still obtained for each document fragment in a query result.

Table 7 also contains query execution times for the *Xalan*[4] DOM implementation, version 2.5, SAXON,[5] version 8, and the X-Hive 6.0 system.[6] Unfortunately, all these systems required too much main memory to run most queries to completion on the test machine. Other native XML database systems, for example the latest versions of Qexo,[7] QuiP,[8] and IPSI-XQ,[9] did not run any of the queries due to lack of memory. This demonstrates another advantage of our system that executes queries quickly with minimum memory usage.

It should be noted that the systems used for comparison employ a query execution model that is different from our structural join approach. Thus, their query execution times can only provide limited means to judge the performance of our implementation. However, unfortunately, there are no other systems based on structural joins freely available as of now.

Without fetching the related content, almost all queries can be evaluated within about 1 s. In general, fetching the content of result fragments adds a significant amount of time, which depends on how much text content each document fragment contains. The times clearly depend on the cardinality of the input data as, for instance, queries Q_3 and Q_4 demonstrate. The selectivity of the full tree pattern only plays a role in that it determines the number and kind of accesses to the A-index. As the only index, the A-index provides for both a direct access to each element and a sequential scan. Therefore, for selective results, single addresses can be accessed without using an iterator.

The input cardinalities shown in Table 6 are relatively small because they are based on full path patterns instead of just node labels. Thus the limited size of the input is not due to simplicity of the example query, but rather a main advantage of our indexing and query evaluation approach. In contrast to most existing approaches [3, 46, 56, 65, 71], our approach exploits the selectivity of full path patterns as they are part of the tree-pattern query to limit the input.

For example, to process a query such as Q_8, there is only one main join besides fetching physical addresses. The join resembles the query pattern from the path patterns for the selection and condition paths. This requires 18,817 + 566,308 = 585,125 element accesses. In most existing approaches, index lists for the pairs of node types *entry-features*, *features-metal*, and *entry-authors* need to be joined, adding joins for 50,000 *entry* and 40,925 *features* list elements.

The conceptually different processing of queries makes it difficult to directly compare query execution times of our and earlier approaches. However, in general, the queries we have used for experiments are more complex than reported in previous work. The query processing times presented here are still below those reported earlier. For instance, Al-Khalifa et al. [3] report execution times of a single join in the seconds range on a data set comparable to *XMark 1Gb*, a 500-MHz Pentium 3, and a larger cache than in our case. A query like Q_{10} on *XMark 1Gb*, however, would require several such joins. Furthermore, the experiments of Al-Khalifa et al. [3] do not include any content access.

Other advantages of our approach are demonstrated by queries Q_8 and Q_9. The construction of node IDs and their storage depend only on their local context within a source. Therefore, the queries take almost the same amount of time independent of whether they are executed on *SwissProt* or on the ten times as large *Big10* source that contains *SwissProt*. In fact, on *Big10* query execution was consistently slightly faster – likely because of a more favorable location of the indexes on disk.

Query Q_{11} demonstrates the system's ability to deal with expensive queries – the term "money" occurs 133,494 times in a total of 197 different node types and, thus, T-index sublists. Furthermore, without the additional A-index access, the execution of Q_{11} is very similar to the implementation of embedded document retrieval. Instead of joins with P-index lists, embedded document retrieval employs joins with an intermediate fragment sequence and its associated term counters (see the algorithm in Fig. 13). Conceptually, both kinds of joins are identical.

For *XMark 1Gb*, Fig. 20 shows the time it takes to accumulate counters for a hypothetical sequence of 10,000 fragments and for terms with increasing total number of occurrences. As the worst-case experiment, the full sequence was always joined with all term-related T-index lists. That is, each entry in each such term list was checked against each root node in the fragment sequence.

In other words, the fragment sequence was assumed to consist of 10,000 duplicates of the full *XMark 1Gb* source, the largest possible fragment that can be selected

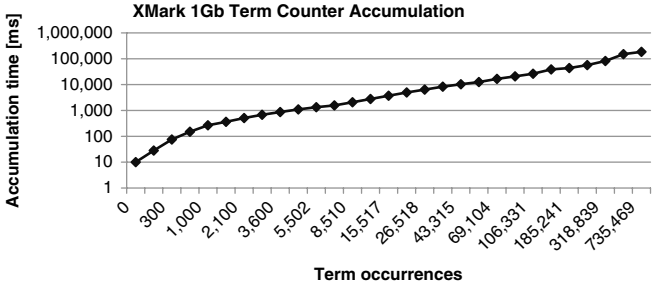

Fig. 20 Accumulating term counters with increasing number of occurrences from *XMark 1Gb* for a fragment sequence of 10,000 elements

[4] xml.apache.org/xalan-j/
[5] saxon.sourceforge.net
[6] www.x-hive.com
[7] www.qexo.org
[8] developer.softwareag.com/tamino/quip/
[9] www.ipsi.fraunhofer.de/oasys/projects/ipsi-xq/index_e.html

from the source. Yet, the system did not know about duplicates, and thus it did all the checks for descendant nodes and all the counter accumulation as described in the algorithm in Fig. 13. If the fragments had been selected from a much larger source (and not just duplicates of one complete source), then that source would have to be at least 10 terabytes large. Although T-index lists for such a source would be longer, too, this gives some impression of the magnitude of the experiment.

Figure 20 shows that, as expected, the execution time grows at most linearly with the number of occurrences of query terms. Even in this worst-case scenario, the accumulation takes only about 1 s for a term occurring 4,500 times in *XMark 1Gb*. At this occurrence frequency this term is likely a stop word. Thus, for any meaningful query term, the accumulation time can be expected to stay below 1 s. For a term occurring only a few hundred times – likely a term more significant for document retrieval – the time is reduced to under 100 ms. For other sources, the graph is very similar to that in Fig. 20.

In summary, the experiments in this section have shown that the node identification and indexing scheme proposed in this paper is extremely storage efficient. Furthermore, the experiments suggest that the index structures can efficiently support both data retrieval and embedded document retrieval. However, so far there are almost no systems available for comparison that are able to work on sources of any significant size, and obviously there is no established framework for testing the efficient execution of IIR queries. This naturally limits the expressiveness of the presented experiments for query execution.

7 Related work

We classify related work into (1) data retrieval approaches and their supporting index structures and underlying node identification schemes and (2) extensions of data retrieval schemes to support document retrieval.

7.1 Data retrieval

Our approach is based on structural joins on custom-tailored inverted files [68] as earlier approaches for XML [3, 46, 56, 71]. (In the general context of semistructured data, there is earlier work on structural joins, e.g., by Jagadish et al. [43] and work cited therein.) Unlike our approach, these schemes rely exclusively on interval node identifiers [2], do not group these identifiers by node type, and commonly neglect storage efficiency.

Through extensive materialization of common index access paths, Chien et al. [21] improve on the basic structural join approach of Al-Khalifa et al. [3] yet neglect index sizes. Bruno et al. [17] present the most effective structural join algorithms so far but also rely on a more extensive index structure. Still based on binary node relationships, their approach exploits the selectivity of the full query pattern to reduce intermediate query results. We achieve the same advantage by directly matching full path patterns, yet at index sizes of less than half that of the indexed source. Furthermore, as briefly discussed in Sect. 4.4 and shown in Fig. 6, our indexing scheme can be extended in a fashion comparable to earlier work [17, 21] with acceptable storage overhead. Moreover, so far the storage efficiency of our approach is achieved without applying index compression techniques as discussed, for instance, by Witten et al. [68]. Such compression can still be applied to the proposed index structures.

Work on node identification schemes for tree and graph structures in general originates from Peleg [53] and Santoro and Kathib [60] but does not include efficiently encoded path identifiers. The interval node identification scheme as discussed in detail by Abiteboul et al. [2] encodes less information than μPIDs but still results in larger indexes, as shown in Sect. 6 and our earlier work [15]. Sacks-Davis et al. [57] introduce path identifiers similar to μPIDs, but with an encoding that does not allow for quickly checking parent–child and ancestor–descendant relationships. Moreover, they do not discuss how to efficiently map their path identifiers to physical addresses. In our approach, node IDs directly include information about where to find related physical addresses. This direct mapping is the most efficient way to map logical to physical identifiers [30].

The μPID scheme is a prefix-based approach like the one discussed by Kaplan et al. [44]. Unlike μPIDs, their identifiers have a guaranteed length of $O(\log n)$ bits for a source of n nodes while, however, being less suitable for efficient query processing. Furthermore, storage efficiency is not only a matter of short, single Ids, but also of density of the index structures that use them, as we have shown here.

With respect to structural join approaches, only Li and Moon [46] address node identifiers that support data insertions. Our μPID scheme can be extended in an analogous fashion. Furthermore, more dynamic node identifiers are discussed by Cohen et al. [24], but, compared to μPIDs, the presented approach is weaker in its direct support of pattern queries. Different node IDs for SJs with the same drawbacks and, in addition, an extensive storage overhead are compared by Tatarinov et al. [65] on indexes (and content) stored in a relational database. Also based on SJs over relations, Yoshikawa et al. [70] extend a basic interval node ID scheme by information about related rooted label paths but without encoding the full rooted data path in every node ID as in our approach.

DataGuides as underlying our approach were introduced by Goldman and Widom [36] and first used within the Lore system [48]. However, query processing in Lore as well as in the more recent Natix XML data management system [32] builds on pointer chasing in secondary memory, which has been shown to be relatively inefficient compared to set-based query processing approaches [63]. Other indexing approaches for XML [23, 25, 49] support only limited kinds of queries and are unsuitable for extension to ranked document retrieval.

As a general observation, most of the existing work is evaluated based on very small test data sets, whose size often resembles only a tiny fraction of the total main memory of the computer the tests were run on. It is not untypical to find *Shakespeare* or even only parts of it being used for the evaluations. In our work, we avoid such pitfalls by using more realistic data, as Sect. 6 shows.

7.2 Document retrieval extensions

In the area of classical document retrieval, passage retrieval [45, 58] can be seen as a document retrieval extension of a rudimentary data retrieval approach. However, passage retrieval lacks a ranking of a dynamically established input as in our scheme. Similarly, other approaches [34, 40, 50, 64, 66] exploit structural information of XML to improve the weighted ranking of documents without abandoning the notion of a mostly static collection of (sub-)documents. Based on a simple SJ approach, both Myaeng et al. [50] and Shin et al. [64] also rely on an inefficient node identification and, thus, indexing scheme.

Fuhr and Grossjohann [35] extend a general XML query language by a document retrieval operator, which, however, does not provide for meaningful, arbitrarily nested subqueries. With the same limitation, Grabs and Schek [38] generalize the approach of dynamically accumulating biased term weights within a tree structure as underlying Fuhr and Grossjohann's work [35]. The biased term accumulation employed in both approaches can be applied in our approach as well. Recently, Al-Khalifa et al. [4] have applied the classical structural join algorithm [3] to accumulate term counters, which are used as the sole means to obtain relevant document fragments of arbitrary type. Our retrieval approach is conceptually different from theirs by ranking a more homogeneous set of document fragments previously selected by a precise subquery.

Unlike our approach, the focused text search in hierarchical structures proposed by Jacobsen et al. [42], the extension of an XML query language by Florescu et al. [33], the Proximal Nodes and related models [7, 51], and, finally, the algebra for structured text search proposed by Callan et al. [19] are all limited to Boolean keyword searches without support for a weighted ranking of dynamic sequences of document fragments. XQuery/IR [13, 14] is also more expressive than existing text retrieval extensions of relational and object-oriented databases [5, 29, 47, 69]. These extensions require a user to decide on a static document collection beforehand and not within a query, or they do not support a weighted ranking at all.

Recently, the XQuery and XPath full-text requirements [18] introduced a scoring mechanism for XQuery that largely follows our proposal for such an operator [14]. A notable difference is that we chose an implicit sorting of ranked results because this allows for a more flexible limiting of the result to the most relevant fragments, and the related statement provides the user with additional means to influence the ranking procedure. XQuery's full-text requirements rely on the existing *order by* clause to sort the result according to the assigned score. Botev et al. [6] proposed TeXQuery, which extends XQuery by full-text search capabilities as well. They emphasize the completeness of the extended language [11], while we focus especially on an efficient implementation of dynamic ranking. Therefore, our contributions are complementary to their work and can serve as the core of an efficient implementation of TeXQuery.

Schlieder and Meuss [61] discuss the matching of so-called structural terms, i.e., terms embedded into a labeled tree, against a single XML source. As in our approach, their goals are the integration of document and data retrieval, and dynamic ranking. However, while extending standard document retrieval, their approach does not provide for a full data retrieval language. Instead of using known ranking schemes on an arbitrary but given sequence of document fragments, they rely on an ad hoc ranking formula for all subtrees of a source that satisfy a certain text pattern. Their prototypical implementation can benefit from the index structures introduced in this paper.

8 Conclusions and future research

We introduced integrated information retrieval (IIR), a conceptually new approach to integrated data and document retrieval based on XML. In IIR, ranked document retrieval is embedded into an XML query language, working on arbitrary, intermediate sequences of document fragments (DFs). In particular, IIR provides for a meaningful nesting of data and document retrieval subqueries, which allows for answering new kinds of queries. Although based on XML at present, IIR can be employed in, e.g., relational databases, there eliminating the need to decide on a single, static view of a database as document collection.

Furthermore, we discussed the syntax and semantics of XQuery/IR, an extension of the XQuery language that allows a user to employ IIR. Finally, we detailed an efficient realization of IIR in terms of supporting index structures and efficient query processing.

For this realization, we have introduced a new node identification scheme called μPIDs ("micro path identifiers") that encodes rooted data paths. We have shown that μPIDs, when used in index structures for XML tree-pattern matching, provide for very small index structures. Discovery of tree patterns based on node labels can be supported with a (path) index size of only 2–4% of the indexed source. Adding support for term containment conditions, weighted ranking of arbitrary, intermediate sequences of document fragments, and the mapping of logical μPIDs to physical addresses still leaves the total index size at only about half of the size of the indexed source. The typical path index structure is about 80%, and the term index structure is 50% smaller than the most storage-efficient index implementation we provide for the widely used interval node identification scheme.

Our studies are based on a significant number of large, heterogenous XML sources of different structural

complexity. This variety of test sources also provides detailed insights into dependencies between source properties and index sizes and can guide future research on semistructured and XML data. In this paper, by giving background information and discussing realization alternatives, it has been our explicit goal to initiate future research on IIR and to present a fresh approach to data retrieval for XML.

Several areas are worthy of future research. To name just a few, we are presently seeking to integrate into a coherent framework the different query processing alternatives discussed. Furthermore, we are investigating how index compression techniques will allow for a further reduction of index sizes. Finally, we are interested in identifying those existing weighting algorithms that are particularly suitable for ranking dynamic document fragment sequences, or whether new algorithms are advisable.

Acknowledgements We thank the reviewers for their thoughtful and valuable suggestions on the draft of this paper. We also thank Reuters for providing us with the Reuters Corpus.

References

1. Abiteboul, S., Buneman, P., Suciu, D.: Data on the Web: From Relations to Semistructured Data and XML. Morgan Kaufmann, San Francisco (2000)
2. Abiteboul, S., Kaplan, H., Milo, T.: Compact labeling schemes for ancestor queries. In: Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 547–556 (2001)
3. Al-Khalifa, S., Jagadish, H., Koudas, N., Patel, J.M., Srivastava, D., Wu, Y.: Structural joins: a primitive for efficient XML query pattern matching. In: Proceedings of the IEEE International Conference on Data Engineering (ICDE), pp. 141–152 (2002)
4. Al-Khalifa, S., Yu, C., Jagadish, H.: Querying structured text in an XML database. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 4–15 (2003)
5. Alonso, O.: Oracle text white paper. Technical Report, Oracle Corporation, Redwood Shores, CA (2001)
6. Amer-Yahi, S., Botev, C., Shanmugasundaram, J.: TeXQuery: a full-text search extension to XQuery. In: Proceedings of the 13th World Wide Web Conference, pp. 583–594 (2004)
7. Baeza-Yates, R.A., Navarro, G.: Integrating contents and structure in text retrieval. SIGMOD Rec. **25**, 67–79 (1996)
8. Baeza-Yates, R., Ribeiro-Neto, B.: Modern Information Retrieval. Addison-Wesley, Reading, MA (1999)
9. Berglund, A., Boag, S., Chamberlin, D., Fernández, M.F., Kay, M., Robie, J., Siméon, J.: XML Path Language (XPath) 2.0. W3C working draft, W3C (November 2003). `www.w3.org/TR/2003/WD-xpath20-20031112`
10. Boag, S., Chamberlin, D., Fernández, M.F., Florescu, D., Robie, J., Siméon, J.: XQuery 1.0: An XML Query Language. W3C working draft, W3C (November 2003). `www.w3.org/TR/2003/WD-xquery-20031112/`
11. Botev, C., Amer-Yahia, S., Shanmugasundaram, J.: On the completeness of full-text search languages for XML. Technical Report, Cornell University (December 2003)
12. Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, E., Yergeau, F., Cowan, J.: Extensible Markup Language (XML) 1.1. W3C recommendation, W3C (February 2004). `www.w3.org/TR/2004/REC-xml11-20040204`
13. Bremer, J.-M., Gertz, M.: Query processing and index structures for integrated XML document and data retrieval. Technical Report CSE-2002-22, Department of Computer Science, University of California at Davis (2002)
14. Bremer, J.-M., Gertz, M.: XQuery/IR: integrating XML document and data retrieval. In: Proceedings of the 4th International Workshop on the Web and Databases (WebDB), pp. 1–6 (2002)
15. Bremer, J.-M., Gertz, M.: An efficient XML node identification and indexing scheme. Technical Report CSE-2003-04, Department of Computer Science, University of California at Davis (2003)
16. Brin, S., Page, L.: The anatomy of a large scale hypertextual Web search engine. In: Proceedings of the 7th World Wide Web Conference, Elsevier, Amsterdam, pp. 107–117 (1998)
17. Bruno, N., Koudas, N., Srivastava, D.: Holistic twig joins: optimal XML pattern matching. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 310–311 (2002)
18. Buxton, S., Rys, M.: XQuery and XPath full-text requirements. W3C working draft, W3C (May 2003). `www.w3.org/TR/2003/WD-xquery-full-text-requirements-20030502/`
19. Callan, J., Croft, W.B., Broglio, J.: TREC and Tipster experiments with InQuery. Inf. Process. Manage. **31**, 327–332, 343 (1995)
20. Chamberlin, D., Frankhauser, P., Florescu, D., Marchiori, M., Robie, J.: XML query use cases. W3C working draft, W3C (November 2003). `www.w3.org/TR/2003/WD-xmlquery-use-cases-20031112/`
21. Chien, S.Y., Vagena, Z., Zhang, D., Tsotras, V.J., Zaniolo, C.: Efficient structural joins on indexed XML documents. In: Proceedings of the 28th International Conference on Very Large Data Bases (VLDB), pp. 263–274 (2002)
22. Chinenyanga, T.T., Kushmerick, N.: An expressive and efficient language for XML information retrieval. In: Proceedings of the 24th International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 163–171 (2001)
23. Chung, C.W., Min, J.K., Shim, K.: Apex: an adaptive path index. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 121–132 (2002)
24. Cohen, E., Kaplan, H., Milo, T.: Labeling dynamic XML trees. In: Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), pp. 271–281 (2002)
25. Cooper, B.F., Samle, N., Franklin, M.J., Hjaltson, G.R., Shadmon, M.: A fast index for semistructured data. In: Proceedings of the 27th International Conference on Very Large Data Bases (VLDB), pp. 341–250 (2001)
26. Cowan, J., Tobin, R.: XML Information Set, 2nd edn. W3C recommendation, W3C (February 2004). `www.w3c.org/TR/2004/REC-xml-infoset-20040204`
27. Croft, W.B.: "What do people want from information retrieval?". D-Lib. Mag. (1995)
28. DeHaan, D., Toman, D., Consens, M.P., Özsu, M.T.: A comprehensive XQuery to SQL translation using dynamic interval encoding. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 623–634 (2003)
29. Dessloch, S., Mattos, N.M.: Integrating SQL databases with content-specific search engines. In: Proceedings of the 23rd International Conference on Very Large Databases (VLDB), pp. 528–537 (1997)
30. Eickler, A., Gerlhof, C.A., Kossmann, D.: A performance evaluation of OID mapping techniques. In: Proceedings of the 21st International Conference on Very Large Databases (VLDB), pp. 18–29 (1995)
31. Fernández, M., Marsh, J., Malhotra, A., Nagy, M., Walsh, N.: XQuery 1.0 and XPath 2.0 data model. W3C working draft, W3C (November 2003). `www.w3.org/TR/2003/WD-path-datamodel-20031112`
32. Fiebig, T., Helmer, S., Kanne, K.C., Moerkotte, G., Neumann, J., Schiele, R.: Anatomy of a native XML base management system. VLDB J. **11**, 292–314 (2002)

33. Florescu, D., Kossmann, D., Manolescu, I.: Integrating keyword search into XML query processing. In: Proceedings of the 9th International Word Wide Web Conference/Computer Networks. **33**(1–6), 119–135 (2000)
34. Fuhr, N., Gövert, N., Kazai, G., Lalmas, M.: INEX: initiative for the evaluation of XML retrieval. In: Proceedings of the ACM SIGIR 2002 Workshop on XML and Information Retrieval (2002)
35. Fuhr, N., Grossjohann, K.: XIRQL: a query language for information retrieval in XML documents. In: Proceedings of 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 172–180 (2001)
36. Goldman, R., Widom, J.: DataGuides: enabling query formulation and optimization in semistructured databases. In: Proceedings of the 23rd International Conference on Very Large Databases (VLDB), pp. 436–445 (1997)
37. Gottlob, G., Koch, C., Pichler, R.: Efficient algorithms for processing XPath queries. In: Proceedings of the 28th International Conference on Very Large Data Bases (VLDB), pp. 95–106 (2002)
38. Grabs, T., Schek, H.J.: Generating vector spaces on-the-fly for flexible XML retrieval. In: Proceedings of the ACM SIGIR 2002 Workshop on XML and Information Retrieval (2002)
39. Graefe, G.: Query evaluation techniques for large databases. ACM Comput. Surv. **25**, 73–169 (1993)
40. Guo, L., Shao, F., Botev, C., Shanmugasundaram, J.: XRank: ranked keyword search over XML documents. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 16–27 (2003)
41. Holmes, N.: The great term robbery. Computer **34**, 94–96 (2001)
42. Jacobsen, G., Krishnamurthy, B., Srivastava, D., Suciu, D.: Focusing search in hierarchical structure with directory sets. In: Proceedings of the 7th International Conference on Information and Knowledge Management (CIKM), pp. 1–9 (1998)
43. Jagadish, H., Lakshmanan, L.V., Milo, T., Srivastava, D., Vista, D.: Querying network directories. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 133–144 (1999)
44. Kaplan, H., Milo, T., Shabo, R.: A comparison of labeling schemes for ancestor queries. In: Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 954–963 (2002)
45. Kaszkiel, M., Zobel, J., Sacks-Davis, R.: Efficient passage ranking for document databases. ACM Trans. Inf. Syst. **17**, 406–439 (1999)
46. Li, Q., Moon, B.: Indexing and querying XML data for regular path expressions. In: Proceedings of the 27th International Conference on Very Large Data Bases (VLDB), pp. 361–370 (2001)
47. Maier, A., Novak, H.J.: DB2's full-text search products – white paper. Technical Report, International Business Machines Corporation (2001). `www-900.ibm.com/cn/software/db2/products/download/whitepaper/whitense.pdf`
48. McHugh, J., Widom, J., Abiteboul, S., Luo, Q., Rajaraman, A.: Indexing semistructured data. Technical Report, Stanford University, Stanford, CA (1998)
49. Milo, T., Suciu, D.: Index structures for path expressions. In: Proceedings of the 7th International Conference on Database Theory (ICDT99). Lecture Notes in Computer Science, vol. 1540, pp. 277–295. Springer, Berlin Heidelberg New York (1999)
50. Myaeng, S.H., Jang, D.H., Kim, M.S., Zhoo, Z.C.: A flexible model for retrieval of SGML documents. In: Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval. ACM Press, New York, pp. 138–145 (1998)
51. Navarro, G., Baeza-Yates, R.: Proximal nodes: a model to query document databases by content and structure. ACM Trans. Inf. Syst. **15**, 401–435 (1997)
52. Papakonstantinou, Y., Garcia-Molina, H., Widom, J.: Object exchange across heterogeneous information sources. In: Proceedings of the 11th International Conference on Data Engineering (ICDE), pp. 251–260 (1995)
53. Peleg, D.: Informative labeling schemes for graphs. In: Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science. Lecture Notes in Computer Science, vol. 1893, Springer, Berlin Heidelberg New York (2000)
54. Reuters Corpus, Volume 1, English language, 1996-08-20 to 1997-08-19, release data 2000-11-03 Format version 1, correction level 0 (2000). `http://about.reuters.com/researchandstandards/corpus/`
55. van Rijsbergen, C.J.: Information Retrieval, 2nd edn. Butterworths, London (1979)
56. Rizzolo, F., Mendelzon, A.: Indexing XML data with Toxin. In: Proceedings of the 3rd International Workshop on the Web and Databases (WebDB), pp. 49–54 (2001)
57. Sacks-Davis, R., Dao, T., Thom, J.A., Zobel, J.: Indexing documents for queries on structure, content and attributes. In: Proceedings of the International Symposium on Digital Media Information Base, pp. 236–245 (1997)
58. Salton, G., Allan, J., Buckley, C.: Approaches to passage retrieval in full text information systems. In: Proceedings of the 16th International ACM SIGIR Conference on Research and Development in Information Retrieval, ACM Press, pp. 49–58 (1993)
59. Salton, G., McGill, M.J.: Introduction to Modern Information Retrieval. McGraw-Hill, New York (1983)
60. Santoro, N., Khatib, R.: Labeling and implicit routing in networks. The Computer Journal **28**, 5–8 (1985)
61. Schlieder, T., Meuss, H.: Querying and ranking XML documents. J. Am. Soc. Inf. Sci. Technol. **53**(6), 489–503 (2002)
62. Schmidt, A.R., Waas, F., Kersten, M.L., Manolescu, I., Carey, M.J., Manolescu, I., Busse, R.: XMark: a benchmark for XML data management. In: Proceedings of the 28th International Conference on Very Large Data Bases (VLDB), pp. 974–985 (2002)
63. Shekita, E.J., Carey, M.J.: A performance evaluation of pointer-based joins. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 300–311 (1990)
64. Shin, D., Jang, H., Jin, H.: BUS: an effective indexing and retrieval scheme in structured documents. In: Proceedings of the 3rd ACM International Conference on Digital Libraries, pp. 235–243 (1998)
65. Tatarinov, I., Viglas, S.D., Beyer, K., Shanmugasundaram, J., Shekita, E., Zhang, C.: Storing and querying ordered XML using a relational database system. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 204–215 (2002)
66. Theobald, A., Weikum, G.: Adding relevance to XML. In: Proceedings of the 3rd International Workshop on the Web and Databases (WebDB). Lecture Notes in Computer Science, vol. 1997, pp. 105–124. Springer, Berlin Heidelberg New York (2001)
67. Tolani, P.M., Haritsa, J.R.: XGRIND: a query-friendly XML compressor. In: Proceedings of the 18th International Conference on Data Engineering (ICDE), pp. 225–234 (2002)
68. Witten, I.H., Moffat, A., Bell, T.C.: Managing Gigabytes—Compressing and Indexing Documents and Images, 2nd edn. Morgan Kaufmann, San Mateo, CA (1999)
69. Yan, T.W., Annevelink, J.: Integrating a structured-text retrieval system with an object-oriented database system. In: Proceedings of the 20th International Conference on Very Large Data Bases (VLDB), pp. 740–749 (1994)
70. Yoshikawa, M., Amagasa, T., Shimura, T., Shunsuke, U.: XRel: a path-based approach to storage and retrieval of XML documents using relational databases. ACM Trans. Internet Technol. **1**, 110–141 (2001)
71. Zhang, C., Naughton, J., DeWitt, D., Luo, Q., Lohman, G.: On supporting containment queries in relational database management systems. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 425–436 (2001)