Effective Pruning for XML Structural Match Queries

Yefei Xin Zhen He Jinli Cao
Department of Computer Science and Computer Engineering
La Trobe University
VIC 3086
Australia
$shawn.xin@oracle.com, \{z.he,\ j.cao\}@latrobe.edu.au$

Abstract

Extensible Markup Language (XML) is becoming the *de facto* standard for exchanging information over the Internet, which results in the proliferation of XML documents. This has led to increased interest in this area by the research community. One of the main challenges is processing large collections of XML documents efficiently. Most current methods suffer from two drawbacks: an inability to complement each other to further enhance query processing performance without modifying the existing query processing engine; and an incapability of being customized for different structural and usage characteristics. This paper presents a new approach for structural query processing called Property-Driven Pruning Algorithm (PDPA), which offers the twin features of *structural query processing independence* and *plug-and-play properties* to overcome both drawbacks. PDPA consists of two phases: the offline and the online phase. During the offline phase, a list of pruning properties is added into the original XML documents. During the online phase, the input queries are modified with a list of carefully selected properties which are used during query processing to quickly prune non-matching candidate documents. We have proposed an *exhaustive* and a *greedy heuristic* algorithm. The experimental results based on both algorithms demonstrate that PDPA can improve XML query processing performance in a variety of situations by up to two fold.

1 Introduction

Extensible Markup Language (XML) has become the *de facto* standard for data exchange over the Internet because of its inborn features, such as machine-independence, vendor-independence, flexibility, scalability, etc. The structure of XML data can be defined by DTD (Document Type Definition) or XML Schema. Due to the proliferation of XML documents, many query languages, such as XPath and XQuery are designed to retrieve XML fragments.

One of the challenges is efficiently processing queries over large XML document collections. A large amount of work has been reported on XML query optimization. It can be partitioned into two main areas: efficient structural query processing in relational databases [1, 6, 10, 19, 20, 22, 23, 24, 41, 47, 49]; and XML structure indexing [7, 11, 25, 26, 29, 30, 34]. Effective pruning of candidate documents for structural query processing is still a very active research topic. Structural query processing refers to finding all occurrences of a given set of structural relationships (such as parent-child and ancestor-descendant relationships) in an XML database. A core component of XPath, XQuery and tree pattern queries is the structural relationships specified in them. The following is an example query that retrieves all sections of papers that contain at least one figure and at least one table:

```
//paper//section[figure AND table]
```

Most existing structural query processing algorithms suffer from the following drawbacks:

1. Inability to complement each other without modifying the query processing engine itself. That is, one algorithm can not be laid on top of another to further speed up query processing without modifying the existing query processing engine.

2. Incapability of being customized to take advantage of different structural and usage characteristics. For example, if the height of the documents in the collection varies significantly, then the system can be customized to use the height of the document as a property for pruning candidate documents. However, if all documents in the collection have the same height, then height should not be used as a pruning property and other properties may be more appropriate.

In this paper, we propose a new approach, the Property-Driven Pruning Algorithm (PDPA), which overcomes the two drawbacks mentioned above. PDPA overcomes the first drawback by introducing the feature of *structural query processing independence*, which allows PDPA to be used on top of any existing structural query processing engine without modifying the query engine at all, as long as it processes queries in a top-down manner. Most existing engines process queries in this way. Our solution involves adding extra information to the XML documents and the input queries. The query processing automatically speeds up when the underlying query execution engine processes the modified query on the modified documents. To demonstrate this, we have built PDPA on top of the Oracle Berkley DB XML [12] with no modification of the database engine itself. PDPA overcomes the second drawback by introducing the *plug-and-play properties* feature. This allows users to add customized properties into PDPA, which incorporates both structural and usage characteristics. Once a set of properties is plugged into PDPA, then PDPA uses them to prune candidate documents. The embedded properties are stripped from the query results before they are returned to the user.

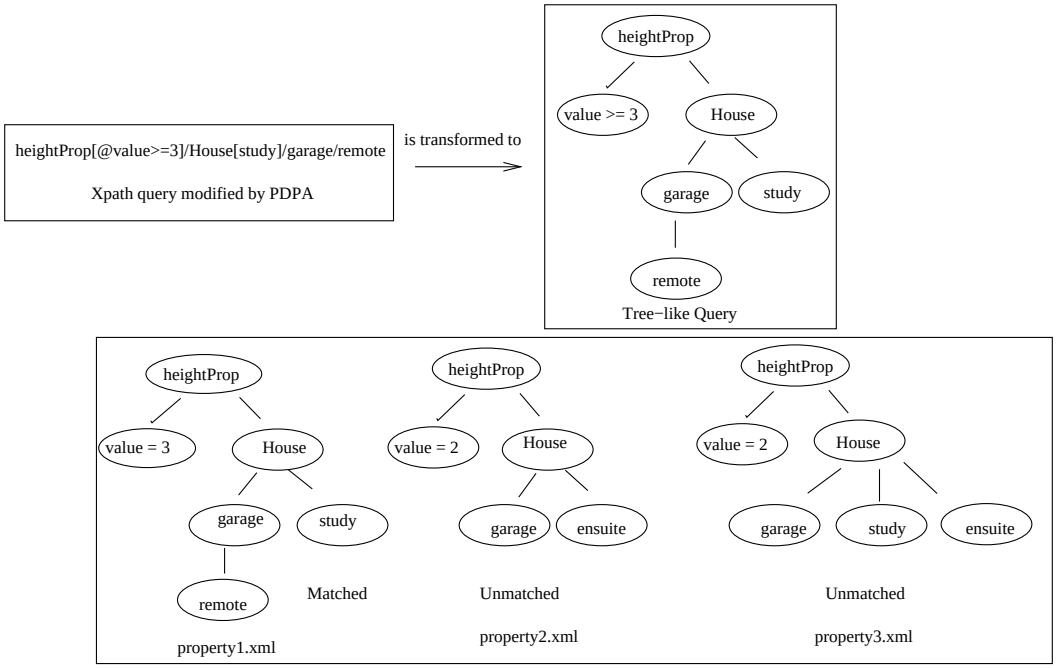


Figure 1: How the PDPA property works

Figure 1 shows an example query searching over a collection of three XML documents. The query looks for houses that have a study room and a garage with a remote control. Note only property1.xml matches the query. The question is how do we find this fact quickly? We do this by adding a height property to the query and to the three documents. Using the height property we can prune out the two documents property2.xml and property3.xml when processing the XPath query. The reason is property2.xml and property3.xml both have height lower than that of the query and therefore can not match the query. The pruning takes advantage of the top-down processing approach to reduce the number of candidate documents in the search space. Using properties for pruning is challenging, because there is a need to estimate the impact of both including and excluding the property. The cost estimation is difficult, since the query can be complex and the number of documents that match a set of properties is difficult to estimate.

Apart from the height properties mentioned above we propose a number of other properties as well. One such

property is the element summary property, which just counts the number of occurrences of a certain tag in the query tree versus the document tree. If the document does not contain enough occurrences of the sought tag then it can be pruned away. This property can be effective in a number of practical situations. For example a query on an XML database for restaurants may contain a wifi tag to indicate the user wishes to find restaurants that offer a certain type of wifi. If the document collection contains only 5% of restaurants that offer any type of wifi, then the fact that the query asks for wifi can be used to prune away 95% of the documents. Other data sets such as the DBLP[40] and swissprot[40] contain varying number of tags. These data sets can be queried in ways that require different number of occurrences of the tags, which in turn can be pruned by element summary properties.

PDPA has two phases, the offline and the online phase. The offline phase adds pruning properties into the original XML documents based on the pruning power of the added properties. Properties that have higher pruning power are placed on top of properties that have less pruning power. The pruning power of a property refers to how many documents can be filtered by using the property during query processing. Pruning power is determined by query workload information which includes all queries that have been executed in the past in addition to how often they are executed. The online phase selects a subset of the properties to be incorporated in the query. Only a subset of properties is selected because some properties have low pruning power for the particular query being executed. We have proposed two algorithms for selecting properties, an *exhaustive* and a *greedy heuristic* based algorithm. We have conducted comprehensive experiments to compare the performance of PDPA against the original query engine without running PDPA.

The rest of this paper is organized as follows: Section 2 presents the related work in the area of structural query processing for XML documents. The problem is defined in Section 3, while Section 4 discusses the proposed PDPA in great detail. Sections 5 and 6 present a detailed experimental setup and evaluation of the experimental results respectively, and Section 7 concludes this paper.

2 Related Work

We discuss related work under the following three categories: structural indexes, relational database-based structural query processing and mixed mode processing.

The structural index-based approaches is the most similar to our work. In this approach, various types of indexed summaries of XML document structures are built. These summaries are used to efficiently answer structural queries. Three early indexes include DataGuides[30], 1-index[34] and F&B-index[25]. These indexes effectively index the entire structure of the XML documents by indexing all paths from the root. This leads to the following drawbacks: a large storage space is needed; it does not take advantage of highly selective sub-structures to allow quick initial pruning of the search space; it does not consider query workload characteristics. The A(k)-index[26] improves over the 1-index by reducing the large storage size requirement. It achieves smaller storage size by storing only approximate information for paths longer than k. However, it still does not isolate highly selective sub-structures to take full advantage of them nor does it consider query workload characteristics. The APEX[11] and the D(K)-index[7] index structures improve on the previously mentioned index structures by adapting themselves to changing query workload characteristics. However, they do not distinguish between highly and lowly selective sub-structures and therefore do not take advantage of this important feature of XML document structures to build the most effective data structure. Elghandour et al. [15] propose a system that helps users decide which indexes to build for an XML document collection. The system is tightly coupled to the query optimizer using database statistics to perform cost estimation. Chen et. al. [9] propose an XML indexing approach which allow every fully specified XML query to be answerable using a single lookup. In contrast to the above work, PDPA does not attempt to index all or part of the structure of the XML document but rather allows users to plug in different types of pruning properties. These properties are then only used if they are found to be highly selective for a particular query and document collection.

Among the index-based approaches, the work that is most similar to ours is the Minimal Infrequent Structures (MIS) index [29]. This approach finds MIS structures in the XML document collection and then indexes the document collection using these structures. The MIS structures are those structures that have high selectivity and are therefore the most effective ones for pruning. Their work does not take query workload information into consideration when finding MIS structures which means they may index structures that are very selective but never used since no query has that structure. They have a rigid definition of MIS which precludes simple yet powerful pruning properties like

height and maximum number of children across all nodes of a document which can be used in PDPA.

Kim [27] proposed an approach that uses element trees and distribution encoded bitmaps to efficiently skip elements when processing structural joins. This technique exploits the distribution of elements as well as the context information to efficiently process structural join queries. In contrast to this work, PDPA does not provide a faster method of processing each of the structural joins inside the query but rather allows users to plug in different types of properties at the top of the documents and queries in order to prune the number of documents that need to be considered during query processing. Therefore, PDPA can be used in conjunction with the method proposed by Kim.

There has been much investigation into how best to use relational database systems to process XML structural queries, mostly based on some variant of storing the following information per element: document ID; presort order; postsort order; and level number. This information is stored in a table in the database. Structural query processing involves performing various types of efficient multi-predicate joins on this data. Zhang et al. [49] proposed the MPMGJN algorithm to efficiently perform the multi-predicate join. The results showed MPMGJN can outperform standard RDBMS join algorithms by an order of magnitude. Al-Khalifa et al. [1] improved over MPMGJN by creating an I/O and CPU optimal join algorithm that uses stacks for matching binary structural relationships against an XML database. Chien et al. [10] and Jiang et al. [23] proposed various types of indexes to speed up join processing. Bruno et al. [6] developed a novel holistic twig join algorithm which matched entire twigs at once instead of binary structural relationships. This allowed their algorithm to dramatically reduce the size of intermediate results. Jiang et al. [24] improved on the twig processing approach of Bruno et al. [6] by extending it to take advantage of all or partly indexed XML documents. Jiang et al. [22] then extended twig processing to efficiently handle OR predicates. Grust et al. [19] proposed a database index structure that supports *all* XPath axes which can live completely inside a relational database system. The implementation can benefit from R-trees which are now part of most relational databases. Grust et al. [20] next proposed a novel algorithm called the staircase join which uses tree properties such as subtree size, intersection of paths, inclusion and disjointness of subtrees to improve XPath performance. The above algorithms, unlike our algorithm, do not take advantage of query workload information to speed up structural query processing. Chen et al. [8] proposed an XML structure indexing technique which is effective at reducing IO and avoids generating redundant intermediate results.

Wang et al.[44] proposed a novel index structure called ViST which represented both XML documents and queries by structure-encoded sequences. Then query processing is equivalent to finding subsequence matches. ViST supports dynamic index updates and relies solely on B+-trees. Rao and Moon [37, 38] proposed PRIX which is a way of indexing XML documents and processing twig patterns in an XML database using Prüfer sequences. PRIX allows holistic processing of a twig pattern without breaking the twig into root-to-leaf paths and processing these paths individually. Prasad and Kumar [36] improved on PRIX by proposing a variation of the Prüfer sequences. They establish interesting properties to the sequencing method which leads to a new efficient algorithm. Results show their method outperforms PRIX in a variety of situations. Wang and Meng [43] address the problem of query equivalence with respect to subsequence matching of XML documents and also introduce a performance-oriented principle for sequencing tree structures. The results showed their approach outperform earlier methods. Tatikonda et al. [42] have proposed a method for processing structural joins by representing the XML tree as a sequence and then processing the query by searching for the longest common subsequences. Their method uses a sequence of inter-linked early pruning steps coupled with a simple index structure that enabled a reduction in the search space. This method was found to be up to 2 or 3 fold faster than current state-of-the-art techniques. Again PDPA can be used in conjunction with these techniques by placing the pruning properties at the top of the documents which corresponds to changing the beginning of the sequences.

Koch et al.[28] propose a string matching approach for efficient searching and navigation of XML documents and streams. Their technique can be used for pre-filtering XML documents. It differs from existing techniques by usually requiring only a fraction of the input at a time for processing and thereby uses both main memory and the CPU very efficiently. Experiments show their algorithms when used on an in-memory query engine can experience speed-ups of two orders of magnitude.

In the mixed mode processing approach proposed by Halverson et al. [21], the XML query processing engine uses a mixture of native and relational database technologies. The system uses a cost model to find the optimal combination of techniques to use for a given query. The system is very complex and does not take advantage of query workload information to enhance query performance. Gou and Chirkova [18] survey high performance techniques for querying large XML data repositories. In particular they are interested in techniques for matching twig patterns. They study

two major classes of XML query processing techniques: the relational approach and the native approach.

Boncz et al. [4] developed the MonetDB Relational XQuery system which uses relational data management infrastructure to create a fast and scalable XML database. The system implements all essential XML database functionalities. The system extended state-of-the-art techniques by developing techniques such as loop-lifted staircase join and efficient relational query evaluation strategies for XQuery theta-joins with existential semantics.

The minimization of tree pattern queries has been studied by Amer-Yahia et al.[2]. They propose algorithms that identify and eliminate redundant nodes in the pattern early. This produces a tree pattern that can be matched against the document collection more efficiently. They consider tree pattern minimization in the presence and absence of integrity constraints. There has also been work on the containment and equivalence of fragments of XPath expressions. Deutesh and Tannen [13, 14] investigates containment of XPath expressions under various integrity constraints and provides decidability, undecidability and hardness results. Miklau and Suciu [33] study the containment and equivalence problems for XPath queries involving wildcards, branching and descendant edges. They prove for this type of queries the problem is co-NP complete and develop an efficient algorithm for special cases of these queries. Michiels et al. [32] identified one of the shortcomings of most existing XQuery compilers is that they do not support tree pattern algorithms. They state this is due to the fact these systems do not support tree patterns in their XML algebra. Accordingly they propose an XML algebra that does support tree patterns.

Query optimization for XML queries have been studied in the existing literuature[3, 31, 48]. McHugh and Widom [31] propose a cost-based query optimizer for the XML database *Lore*. They define logical and physical query plans, database statistics, a cost model and describe plan enumeration for reducing the large search space. Their optimizer is fully implemented in *Lore* and they report some initial results. Wu et al. [48] present five algorithms for structural join order optimization for XML tree pattern matching. They performed an extensive performance study of the relative merits of the different algorithms and reported their relative tradeoffs. Balmin et al. [3] propose heuristic-based rewrite transformations to group XPath expressions to optimize concurrent evaluation of multiple XPath expressions. They also propose cost-based optimization to order the groups within the execution plan.

An important component of query optimization is selectivity estimation. There has been much existing work on selectivity estimation for XML documents[16, 45, 46]. One of the most recent work is by Fisher and Maneth [16]. They propose a selectivity estimation technique for XML documents which estimates selectivity for all XPath axes, gives a guaranteed range which the actual selectivity lies in and allows incremental updates to the XML database.

XML documents can be modeled as graphs. Gou and Chirkova[17] propose an algorithm which finds the top-ranked twig-pattern matches from large graphs. They propose the DP-B algorithm which retrieves the exact top-ranked match from potentially exponentially many matches in linear space and time. A second proposed algorithm called DP-P can run in far less than linear time and space in practice.

Bressan et al.[5] propose strategies to prune XML documents to smaller sizes with respect to a given query. The parts of the documents pruned are guaranteed to not affect the result of the query. Since the documents are smaller the queries can be processed faster. Our work differs from theirs in that we prune away entire documents instead of parts of documents. We therefore do not need to store the pruned documents whereas they need to store the smaller pruned documents in order to feed it to the query execution engine. Therefore our work has lower storage overhead compared to theirs. In addition our work can be in conjunction with theirs by first using our algorithms to prune away the documents that can not be in the result set before using their algorithm to reduce the size of the documents.

Moro et al.[35] propose algorithms to efficiently determine which subscribers to push incoming XML documents to in a publish and subscribe scenario. They propose the BoXFilter, which is based on a new treelike indexing structure that organizes the queries based on their similarity, which is then used to prune away queries that are not related to the incoming documents. They use string matching techniques to determine the similarity between queries. Their work differs from ours in that we prune documents instead of queries.

Sanz et. al. [39] propose a method of finding XML documents that are similar in terms of both content and structure to a given query. They argue this is useful for internet applications since there XML document collections often have heterogenous structure. In contrast to this work we find exact matches to queries instead of approximate matches.

3 Problem Definition

This section defines the pruning problem in terms of two phases: the offline and online phase. The goal of the offline phase is to find an optimal ordering of properties to be added into documents, which leads to minimum overall query processing cost for a given query workload. The goal of the online phase is to find the optimal ordering of properties to be included and rewrite the query with these properties, which leads to minimum processing cost of executing a specific query.

3.1 Query definition

Our system is designed to handle the tree structure component of XPath, XQueries and tree pattern queries. We handle both parent-child predicates and ancestor-descendent predicates and a mixture of them. Currently we do not consider wildcard predicates but it is a candidate for future work. We allow filtering constraints (predicates) that involve structure only. For example: //paper//section[figure AND table].

When the properties of an XML document matches the properties of a query that does not mean the XML document itself matches the query. It just means we are not able to prune out the XML document from the result set using the properties.

3.2 Offline Phase Definition

The offline phase is executed when the query engine is offline. This phase generates and adds pruning properties into the documents, which are then used in the online phase.

To make the pruning more effective, a set of properties that have high pruning power for the particular set of documents and historically collected queries is selected. Next, the order by which the properties are added to the document needs to be determined. Properties with high pruning power should be added toward the top of the document since they more dramatically reduce the work for lower placed properties. However, a property which is very effective for one query may not be effective for a different query, hence query workload information is used to determine the overall pruning power of each property. Therefore, the main problem of the offline phase is to determine the pruning power of the properties for a given workload and use that to order the inclusion of the properties into documents.

The problem is formally defined as follows:

1. Given a set of XML documents *D*, and a set of **n** XML queries with associated workload statistics, let Qs be the historical query workload statistics, which consists of a set of query frequency pairs that can be extracted from a query log Q. Formally, Qs is defined as follows:

$$Qs = \{ \langle q_1 , f_1 \rangle , \langle q_2 , f_2 \rangle , \ldots , \langle q_i , f_i \rangle , \ldots , \langle q_n , f_n \rangle \} \tag{1}$$

   where q_i is the i^{th} XML query and f_i is the frequency of q_i.

2. Given a set of **m** function pairs

$$\delta = \{\langle \eta_1 (d) , \varphi_1 (d, q, p)\rangle , \ldots , \langle \eta_i (d) , \varphi_i (d, q, p)\rangle , \ldots , \langle \eta_m (d) , \varphi_m (d, q, p)\rangle\} \tag{2}$$

   where $\eta (d)$ is a function that takes a document $d \in D$ and outputs a property *p*; $\varphi (d, q, p)$ is a function that takes a document $d \in D$, a property *p*, an XML query *q* and outputs whether *d* matches *q* in terms of *p*.

3. We seek to find ζ, a subset of δ, and an ordering τ of ζ, such that when the properties in ζ are added to D in τ order, the cost *C* is minimized, where *C* is defined as follows:

$$C (Q , \tau , \zeta) = \sum_{i=1}^{n} (cost (q_i , \tau , \zeta) \times f_i) \tag{3}$$

   where $cost (q_i , \tau)$ is the time needed to process query q_i with ordering τ.

3.3 Online Phase Definition

The online phase is the period when the query engine is ready to answer queries. When a query is given, only the properties that apply to the query are kept. For instance, an element A is kept in the offline phase, but may not be useful for the given query. Then the problem becomes which subset of the remaining properties should be used for processing the query. The properties used should be the ones that are most effective in pruning away candidate documents for the query.

The offline phase added the set of properties according to descending overall pruning power for the given workload. However, during the online phase we are optimizing for the particular query currently being run, hence we should adjust the use of the properties based on their pruning power for the particular query. We achieve this by excluding properties that have weak pruning power for the particular incoming query. Using inappropriate properties that have weak pruning power reduces query processing performance by increasing the amount of matching without pruning away many documents. Therefore, the online phase is focused on finding the particular subset of properties to use for the current query.

The problem is formally defined as follows:

Given an XML query q, a document collection D with embedded properties D', and an optimal ordered set ζ (a subset of δ, where δ is defined by Equation 2), we seek to find a subset of ζ, which preserves ζ's order such that the cost of executing query q is minimized.

4 Property-Driven Pruning Algorithm

The previous section defined the problem that needs to be solved for the offline and online phases. The proposed solution to the problem is called the Property-Driven Pruning Algorithm (PDPA). PDPA has the twin features of *structural query processing independence* and *plug-and-play properties*. This section provides a detailed description of PDPA.

4.1 Overview of the approach

The Property-Driven Pruning Algorithm (PDPA) adds extra pre-pruning properties to the documents and queries in order to accelerate query processing, which will work in any query processing engine that processes queries in a top-down manner. The feature is termed *structural query processing independence*. PDPA also describes a general algorithm, which provides the flexibility of taking any set of properties as input and modifying the XML doucments and queries for the most effective pruning. The feature is termed *plug-and-play properties*.

PDPA is separated into two phases, the offline and online phases. During the offline phase, documents and historical queries are analyzed and pruning-related information is combined to determine the order of the properties added.

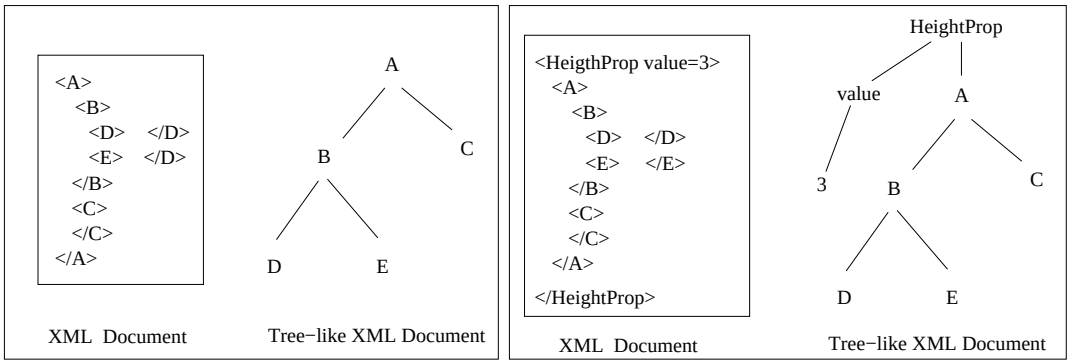
Table 1 gives some example properties that can be used in PDPA. It is important to note that these are example properties only and that other properties can also be used.

Property	**Description**
Maximum Height(d)	The number of nodes from the root node to the furthest leaf node in document d.
Maximum Children(d)	The maximum number of children of any node within document d.
Element summary(d, x)	The number of occurrences of nodes with tag x in document d.
Sibling summary(d, x, y)	The number of occurrences of nodes with tags x and y as siblings in document d.
Parent child summary(d, x, y)	The number of occurrences of nodes with tag x having a child with tag y in document d.

Table 1: Example properties that can be used in PDPA.

For example, consider the XML document tree shown in Figure 2 (a). As the height of the tree is 3, then the property named 'HeightProp' with attribute value 3 is added into the document (shown in Figure 2 (b)). Later when a

query comes with a height greater than 3, the document can be pruned using the height property since the document is shorter than the query and therefore can not match it.



(a) Original XML Document (b) Modified XML Document

Figure 2: Property Example: Adding Height Property to an XML document

The elementary summary property is the summarized information of the element extracted from historical queries, which has a higher pruning power according to Equation 4 in Section 4.2. The number of times that the selected element appears in a query or a document is then considered as the value of the property. For example, if A is the selected element and it appears 3 times in an XML document, then the property named 'AProp' with attribute value 3 will be added into the document.

When using an XML DBMS which processes queries in a top-down manner, the order of the properties is extremely important, because more selective properties will prune more candidate documents for a query, thus leading to fewer scans required during online processing. Therefore, the ranking Equation 5 defined in Section 4.2, is used to determine the order of the selected properties.

During the online phase, a subset of the properties is selected, based on the estimated cost. The selected properties are added to queries (the order remains the same). Statistics encapsulating the pruning power are used to decide which subset of properties should be included. The query is then rewritten by adding selected properties.

The high-level overview of the system is shown in Figure 3. First of all, the XML documents are stored in an XML DB. Secondly, the original documents and historical query log are passed into the offline system. The offline system analyzes the input data and outputs a list of properties in descending order of pruning power. Where pruning power incorporates the selectivity of the pruning property for the queries in the workload and the frequency of the queries using the property. Thirdly, the XML documents are modified with properties extracted from the previous step. In addition, a histogram is generated, which will be used for the online phase. The histogram contains information regarding the number of documents that match a particular range of property values. Fourthly, queries and histograms generated in the offline phase are passed into the online system. The online system analyzes queries in terms of the histograms and then chooses a subset of the properties. In the last step, the queries are rewritten with the subset of properties selected in the previous step, then the modified queries are executed in the XML DB.

The high-level algorithm for the entire system is described in Figure 4. The high level offline algorithm has 3 lines. In line 1 the original document and the historical query workload are analyzed together to arrive at a list of pruning properties and the histogram needed in the online phase. Next in line 2 the properties are ranked according to their pruning power and finally in line 3 they are embed at the top of the documents in ranked order. The high level online algorithm has 4 lines. In line 1 we extract the properties from the query. Next in line 2 we use the histograms created by the offline algorithm to select a subset of the properties with the highest pruning power for the current query to embed into the query. Finally in line 3 and 4 we embed the selected properties in the query and execute the query respectively. The embedded properties are stripped from the query results before they are returned to the user.

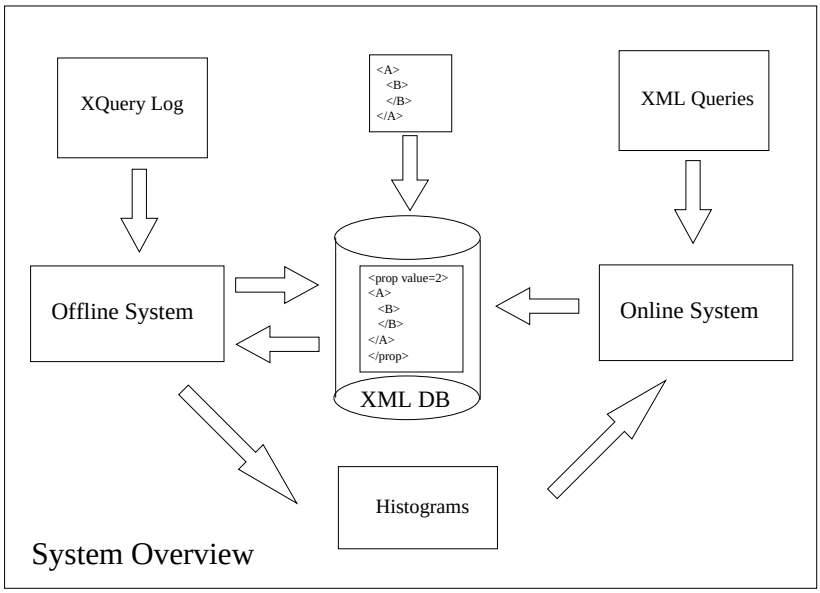


Figure 3: Overview of the System

OfflineSystem(D:Original Documents, Q:Historical Queries)
//extract a list of pruning property p_i and its histogram $H(p_i, D)$ from both D and Q
1. $\{(p_i, H(p_i, D))|i \in N\}$ = xmlStructuralAnalyzer(xmlQueryAnalyzer(Q), D)
//compute ranked property list P', given $P = \{p_i|i \in N\}$
2. P' = xmlStructuralRanker(P)
//generate modified documents D', given D and P'
3. D' = xmlStructuralModifier(P', D)

//Hs is a set of stored histograms, $\{H(p_i, D)|i \in N\}$
OnlineSystem(q:A query, Hs:Histograms)
//extract properties P'' that q contains
1. P'' = xmlQueryScanner(q)
//select the properties P^* based on estimated cost
2. P^* = propertySelector(xmlQueryEstimator(P'', Hs, D'))
//generate modified query q', given q and P^*
3. q' = queryRewriter(q, P^*)
//execute modified query q
4. executeQuery(q')

Figure 4: High-Level System Algorithm

4.2 Offline System

The offline phase firstly analyzes the document set and the query workload to find the properties using Equation 4, and then computes the optimal ordering of these properties using Equation 5. Finally, the ordered properties are embedded into the original documents. We store the properties as elements inside the documents themselves because this allows us to directly use existing XML query processing engines without modification. The properties are stored at the top of the document because most processing engines process documents top down and thereby automatically use the tags to prune the search space. If we stored the index in a separate place then we will need to modify existing XML query processing engines to incorporate PDPA. In contrast, most existing indexing techniques are designed to be placed inside the query processing engine itself and hence the index is stored separate from the documents.

Adding the properties to the documents and query is simply done by adding a sequence of elements at the top of the XML documents and queries. We can assign each element a name such as e1 and have an associated attribute called value (as shown in Figure 2 (b)). For example for the height property the value attribute states the minimum height of a document that can potentially match the query. We also have a small mapping from each element name to the actual definition of the property. For example we can say element e1 correspond to "parent child summary (d, x, y)".

The overall offline system is shown in Figure 5, which consists of three sub-phases, the analyzing phase, ranking phase and modifying phase. During the analyzing phase, historical queries are analyzed to find properties with high pruning power. There are many types of properties that can be used including those described in Table 1. The developer can customize PDPA to use any set of properties. In this section, we demonstrate how element summary properties (first mentioned in Table 1) can be used to achieve high pruning power.

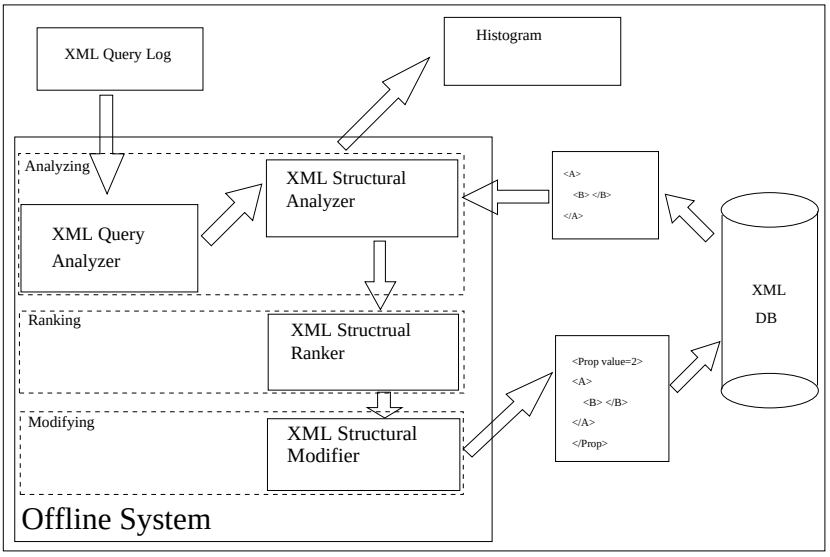


Figure 5: Offline System

An element summary property $e = (e_t, e_x)$ consists of a tag name, e_t and e_x being the number of times it appears in a document. By using this information, we can prune a document if e_t appears less times in the document than in the query. The element summary properties are extracted on a basis of a score which represents the pruning power of the property. The score uses statistics of the query workload to assign a score indicative of the pruning power of the property for the given workload. The following equation defines $score(e)$ for an element summary property e:

$$score(e) \; = \; \sum_{i=1}^{n} (\; freq(q_i) \; \times \; avg\_height(q_i, e_t) \;) \tag{4}$$

where n is the number of queries in the query log Q and $freq(q_i)$ is the frequency of a query $q_i \in Q$ and $avg\_height(q_i, e_t)$

is the average height of element e_t in query q_i, where height is defined as the number of node traversals away from the root node. Note in the case of ancestor-descendent (AD) relationships (e.g. A//B//C) the height is still computed as the number of traversals (in this case down ancestor-descendent links) from the root node. We could put some ad hoc scaling factor to elements that contain the AD relationships but it would be difficult and complex to find the best scaling factor to set (especially if it contains a mixture of AD and parent child(PC) relationships, eg. A/B//C/D). Therefore to keep the algorithm simple we treat both PC and AD relationships the same. Experimental results show our algorithm works very well for queries with AD relationships.

According to Equation 4, elements occurring in queries with higher frequency have higher scores. Furthermore, because the XML DBMS mostly processes queries in a top-down manner, the element further down from the root should be pruned more aggressively (hence is given a higher score), since pruning them late may cause more processing before finding the document does not match. For example a node that is at the bottom of the query tree will not normally be matched against the documents until other nodes closer to the top of the tree have been matched. Using such a node as a pruning node effectively moves it up the query tree so it can be used to prune away documents quicker. On the other hand there is little benefit in using a node that is already at the top of the query tree as a pruning node, since it is already matched first against the document collection.

It is important to note that Equation 4 does not take the document collection into consideration when determining the score assigned to a property. This is because it is too computationally expensive to assess the pruning power of every possible element summary property against the entire document collection. Hence we focus on assessing the pruning power of the properties on the queries first as a way of filtering out properties that are not very likely to be useful. For example properties that are not used in many queries or are high up in the query tree. We use Equation 5 to further refine our choice of properties and rank them in terms of pruning power. It is there that we incorporate the document collection.

Equation 4 is related to Equation 3 of the problem definition since Equation 4 uses the height of the element within the query as an indicator for the execution cost savings from using that element. In addition Equation 5 also incorporates the execution frequency of the query in the cost.

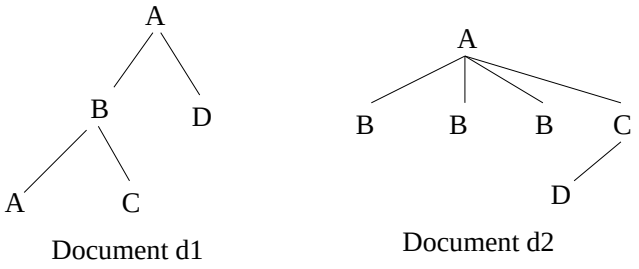


Figure 6: Example documents

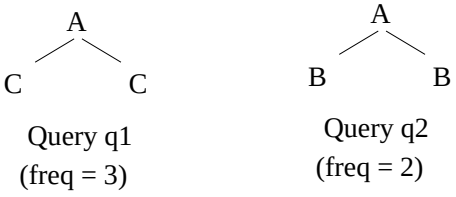


Figure 7: Example queries

We show an example of how Equation 4 can be used to compute the score of the element summary properties for documents d1 and d2 in Figure 6, when given a query workload consisting of the queries q1 and q2 in Figure 7. In the example the element summary properties are denoted by their corresponding tag name. For example the element summary property that prunes using tag A is denoted as A. The scores are computed as follows:

$$
\begin{aligned}
score(A) &= freq(q1) \times avg\_height(q1, A) + freq(q2) \times avg\_height(q2, A) \\
&= 3 \times 0 + 2 \times 0 \\
&= 0 \\
score(B) &= freq(q1) \times avg\_height(q1, B) + freq(q2) \times avg\_height(q2, B) \\
&= 3 \times 0 + 2 \times 1 \\
&= 2 \\
score(C) &= freq(q1) \times avg\_height(q1, C) + freq(q2) \times avg\_height(q2, C) \\
&= 3 \times 1 + 2 \times 0 \\
&= 3 \\
score(D) &= freq(q1) \times avg\_height(q1, D) + freq(q2) \times avg\_height(q2, D) \\
&= 3 \times 0 + 2 \times 0 \\
&= 0
\end{aligned}
$$

Too many properties increase the size and complexity of documents and can slow down query processing. Hence, Equation 4 can be used to find a threshold number of properties with the highest pruning power to include in the documents. However, this is only one possible way of selecting properties.

During the ranking phase, a list of selected properties is ranked using a ranking equation that actually measures how many documents are pruned away using the property for the given query workload. This is not too time consuming, because the analysis step described above has already narrowed down the number of properties to be considered. The equation to calculate the ranking score for each selected property p' is as follows:

$$
ranking\_score(\ p'\ D)\ =\ \sum_{i=1}^{n} (\ prop(q_i,\ p', D)\ \times\ freq(q_i)\ ) \tag{5}
$$

where n is the number of queries in the query log Q, $prop(q_i,\ p', D)$ is the number of documents in collection D pruned using property p' and $freq(q_i)$ is the frequency of query q_i.

Equation 5 gives a higher score to properties that have higher pruning power throughout the whole document collection by incorporating historical query frequencies. To compute $prop(q_i, p', D)$ we run the query q_i on the document collection D with the pruning property p' and record how many documents were pruned by the pruning property. Equation 5 is related to Equation 3 of the problem definition since the $prop(q_i, p', D)$ term is the number of documents pruned by property p' for query q_i, this maps to the execution cost of q_i term of Equation 3, since the more documents are pruned by the property the lower the cost of executing the query. Equation 5 also incorporates the execution frequency of the query.

The following shows how Equation 5 can be used to compute the ranking scores of the element summary properties of the document collection shown in Figure 6 and the queries in Figure 7. In this example the document collection is denoted using λ. The scores are computed as follows:

$$
\begin{aligned}
ranking\_score(A, \lambda) &= prop(q1, A, \lambda) \times freq(q1) + prop(q2, A, \lambda) \times freq(q2) \\
&= 0 \times 3 + 0 \times 2 \\
&= 0 \\
ranking\_score(B, \lambda) &= prop(q1, B, \lambda) \times freq(q1) + prop(q2, B, \lambda) \times freq(q2) \\
&= 0 \times 3 + 1 \times 2 \\
&= 2 \\
ranking\_score(C, \lambda) &= prop(q1, C, \lambda) \times freq(q1) + prop(q2, C, \lambda) \times freq(q2) \\
&= 2 \times 3 + 0 \times 2 \\
&= 6 \\
ranking\_score(D, \lambda) &= prop(q1, D, \lambda) \times freq(q1) + prop(q2, D, \lambda) \times freq(q2) \\
&= 0 \times 3 + 0 \times 2 \\
&= 0
\end{aligned}
$$

During the modifying phase, the ranked properties are added into the original documents in descending order according to the ranking score. In addition, an index for each property is created so that during the online phase, some of the inserted properties can be bypassed if they are deemed to be not very selective for the current query.

4.3 Online System

As mentioned in the problem statement, the online phase first removes the properties which do not apply to a given query. Next, it finds the best combination of properties from the remaining properties to embed into the query. There is a need to estimate the impact of including or not including the property. Moreover, cost estimation is difficult, since the query can be complex and the number of documents that match each property is unknown.

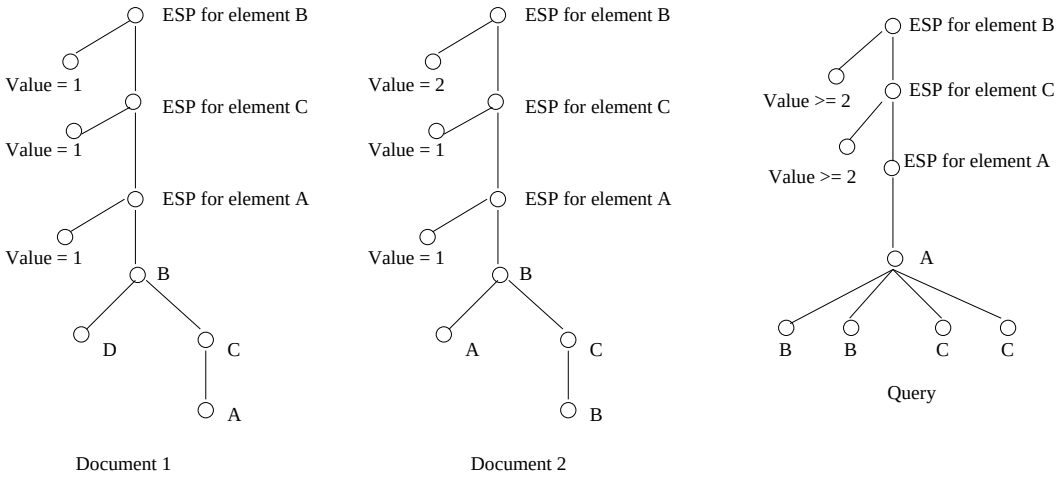


Figure 8: Example document collection and query with embedded properties.

Figure 8 shows an example document collection with three embedded properties and a query using only two out of the three properties. The documents and query have element summary properties (ESP) embedded in them. For example the top property for document 1 is an ESP property that has an attribute value equal to 1. This means there is only one tag B in document 1. If a query has more than 1 tag B then it can not match the query. Among the three

properties embedded in the documents we only select the first two to be used in the query. The third property is not assigned any predicate on its value attribute and therefore is unselected. The reason is the first property allows us to prune document 1 and the second property allows us to prune document 2 but the third property does not help us prune any properties. In the online system we use various statistics to compute whether each properties should be selected or not.

The overall online system is described in Figure 9. Firstly, the input query is scanned to find the properties that satisfy the built-in properties. Secondly, if we are using the exhaustive algorithm (described in Section 4.3.1), we must then estimate the cost per document of processing the input query (B of Equation 6). Thirdly, the histograms are used to calculate the costs of different combinations of properties. Fourthly, the properties with minimum estimated cost are selected. Note: the properties which are selected depend on whether the exhaustive or greedy (described in Section 4.3.2) algorithm is used. Lastly, the query is rewritten to include the selected properties and executed.

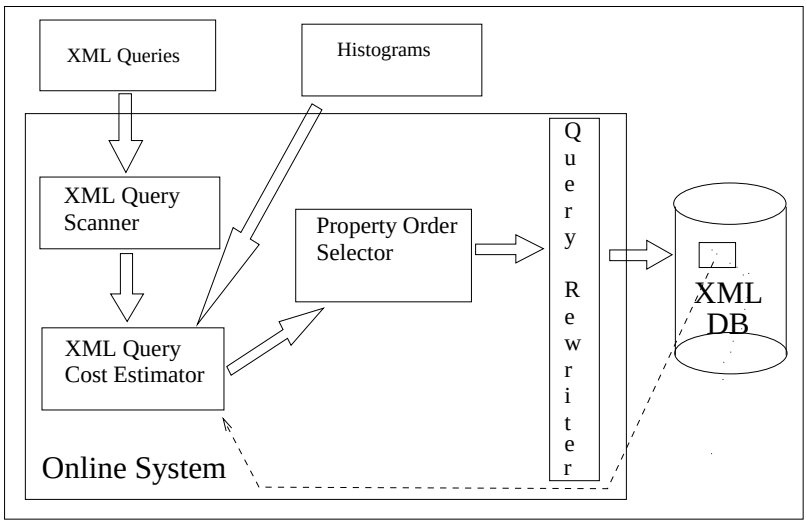


Figure 9: Online System

4.3.1 Exhaustive Algorithm

We first present an exhaustive algorithm to find the optimal subset of applicable properties (properties that apply to the query) to be included into a query. This algorithm calculates the estimated query processing cost of embedding all possible combinations of properties into the query. Then, the combination of properties with the lowest estimated cost is embedded into the query. The equation to calculate the estimated cost of processing one document for query q when using a set of selected properties ($p_1,\ p_2,\ \ldots,\ p_m, D$) is shown as follows:

$$cost(p_1,\ p_2,\ \ldots,\ p_m, q, D)\ =\ (1 + \sum_{i=1}^{m-1} \prod_{j=1}^{i} sel(p_i, q, D)) \times F\ +\ \prod_{i=1}^{m} sel(p_i, q, D) \times B \tag{6}$$

where p_i is the i^{th} selected property, F is the estimated cost per document to process added properties, $sel(p_i, q)$ is the selectivity of p_i for query q, m is the number of properties selected for one combination and B is the estimated cost per document for processing the original query, and D is the document collection. B is estimated online by taking a small partition of the document set and executing the query on it and then dividing the time by the number of documents.

The selectivity of a property p_i for query q is defined as the fraction of documents remaining after property p_i is used to prune the document set for query q. This is determined by using a histogram for p_i. The histogram stores the number of documents for each value of e_x of p_i.

The left hand side of Equation 6 is the additional processing cost of adding the set of properties $p_1 ... p_m$ and the right hand side is the reduced cost of executing the query as a result of adding the properties. The costs of adding

a property p_i equals the product of the selectivities of all properties before p_i multiplied by the constant cost of processing any property. The reason for this is lower selectivity (higher pruning power) before p_i leads to less work during processing of p_i since most documents will already be pruned out before getting to p_i. This idea is also used to compute the reduced cost of executing the query as a result of adding the properties.

We now show an example of cost of a set of properties for a document collection. Figure 10 shows an example document with three properties embedded in the top of it and a query $q1$. For example, we are interested in computing the cost of including the first two of the embedded properties ($p1$ and $p2$). According to Equation 6, the cost will be as follows:

$$\begin{aligned} cost(p1,\ p2, q1) &= (1 + sel(p1, q1) + sel(p1, q1)sel(p2, q1))F + sel(p1, q1)sel(p2, q1)B \\ &= (1 + \frac{4}{10} + \frac{4}{10} \times \frac{5}{10})F + \frac{4}{10} \times \frac{5}{10}B \\ &= \frac{8}{5}F + \frac{1}{5}B \end{aligned}$$

The selectivities $sel(p1, q1)$ and $sel(p2, q1)$ are computed using the histograms shown in Figure 11. In this example, $sel(p1, q1)$ equals $\frac{4}{10}$ since the query $q1$ has two instances of B and according to Figure 11 (a) there are only 4 documents that have 2 or more instances of tag B.

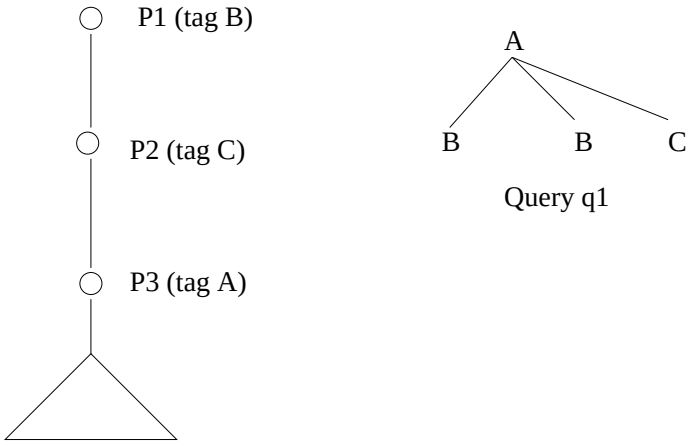
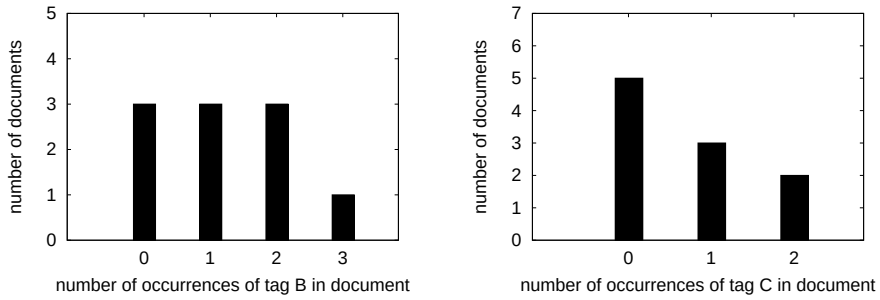


Figure 10: Example document with properties embedded and query.

The run-time complexity of the exhaustive algorithm is $O(2^a)$ where a is the total number of properties that apply to a given query, since it needs to compute Equation 6 for every combination of m properties for $1 \leq m \leq a$.

4.3.2 Greedy Heuristic Algorithm

The exponential run-time complexity of the exhaustive algorithm may mean the overhead of running the algorithm outweighs any potential pruning benefits derived. Hence, we have developed a faster greedy heuristic algorithm that incurs less processing overhead and may find the approximately optimal combination. The heuristic is based on the intuition that properties that have lower selectivity (stronger pruning power) should be used before properties with higher selectivity. Using properties with lower selectivity earlier means a lot of documents are pruned quickly, resulting in less documents remaining for pruning by properties with weaker pruning power. This results in less overall query processing time.



(a) Histogram for property B

(b) Histogram for property C

Figure 11: Example Histograms for properties B and C

Figure 12 presents the greedy heuristic algorithm. The algorithm first narrows the properties down to only those that apply to the query and then it selects the property with the lowest selectivity to be the first property included. It then considers which property to include next from among those properties ranked lower than the selected property. Again, it is the property with lowest selectivity among these properties that is next included. This is continued until there are no more lower ranked properties to be selected.

GreedyAlgorithm(D:Original Documents, q: XML query,
$H(p,\ D)$: selectivity histogram, P': offline ordered list of properties)

1. let $r = 0$ be the ranked list of properties selected
2. $k = P'$ - properties in P' that do not apply to q
3. while $k \neq 0$
4. e = property in k with lowest selectivity computed using $H(p,\ D)$
5. insert e into the end of r
6. remove all properties from k that appear before and including e (according to offline order)
7. end while

Figure 12: Greedy heuristic algorithm

The run-time complexity of the greedy algorithm is $O(a^2)$ where a is the total number of properties that apply to a given query, since it selects the property with the lowest selectivity (highest pruning power) in an iterative way, shown in Figure 12.

5 Experimental Setup

This section describes the detailed setup of the experimental environment used to test the performance of PDPA.

5.1 Software and Hardware

All experiments for this paper are carried out using Oracle Berkeley DB XML[12] version 2.4 on an Intel Core Duo 1.83GHz laptop with 1G RAM running Windows XP professional. Oracle Berkeley DB XML version 2.4 offers the following features to optimize performance: a unique dynamic indexing system that enables optimized retrieval of XML content; targeted indexes and a statistical, cost-based query planning engine; flexible indexing of XML nodes, elements, attributes and meta-data; and node level indexes which improve query performance, especially for large XML documents.

5.2 Synthetic Data and Query Set

We generated synthetic data to conform to three different distribution functions: uniform, normal, and zipf distribution. The distribution functions are used to determine the height of the document and the number of children for each node. While generating each XML document set, a list of parameters is used, which are the maximum height of a document, the maximum number of children per node, the number of documents and the distribution function. The default values of the parameters used are shown in Table 2.

Parameter Description	**Default Value**
Maximum Height	8
Maximum Children of One Node	8
Number of Documents generated	10,000
Distribution Function	normal [1]

Table 2: Default Parameter Values of XML Data Generator

All experimental query sets are generated using the same approach as used to generate data sets (see Section 6.3). However, the query type varies for each query in order to test all possible relationship combinations. It is classified into the following three categories:

PC(/) Parent-child relationship only.
e.g. /P/M/M/N/B.

AD(//) Ancestor-descendant relationship only.
e.g. //P//M//M//N//B.

MIX(/|//) A mix of both parent-child and ancestor-descendant relationships.
e.g. //P/M//M//N/B.

To generate the queries, we again use the three different distribution functions: uniform, normal, and zipf distribution to determine the height of the query and the number of children for each query node. When generating each XML query set, a list of parameters are used, which includes all parameters needed to generate data sets and additional query type. The default values of the parameters used are shown in Table 3.

Parameter Description	**Default Value**
Maximum Height	8
Maximum Children of One Node	8
Number of Queries generated	10
Distribution Function	normal [2]
Query Type	MIX(/|//)

Table 3: Default Parameter Values of XML Query Generator

5.3 Real Data and Query Set

The real data set we used was the collection of all XML documents describing computer science journal and conference proceedings from the Digital Bibliography Library Project (DBLP)[40]. The collections contain meta data for all articles before October, 2002. The data collection contains 3332130 elements, 404276 attributes, a maximum depth of 6 and an average depth of 2.90228.

We generated queries which find all articles that contained the following elements:

[1]The default value is normal distribution, however in some experiments uniform or zipf distribution is used.

[2]The default value is normal distribution, however in some experiments uniform or zipf distribution is used.

cite[0 - 10] between 0 and 10 citations.

author[1 - 10] between 1 and 10 authors.

cdrom[0 - 1] denotes whether we search for articles with the cdrom tag or not.

ee[0 - 1] denotes whether we search for articles which have an associated electronic proceeding or not.

series[0 - 1] denotes whether we search for articles which are part of a series or not.

where the number between [0 - x] above is determined using a uniform random distribution for each query. In our experiments, we generated 10 real queries conforming to the specification above.

5.4 Algorithm Setup

The experimental results are based on comparing the following four algorithms:

Original This algorithm simply runs the original queries on the original data without any extra processing. It gives the bottom-line processing time of input queries.

PDPA-E This is PDPA using the exhaustive algorithm to select properties during the online phase.

PDPA-G This is PDPA using the greedy heuristic algorithm to select properties during the online phase.

PDPA-A This is PDPA with property selection disabled during the online phase. This excludes the overhead of calculating the estimated processing cost of using different combinations of properties. Comparing this algorithm with PDPA-E and PDPA-G will illustrate whether the benefits gained from cost estimation during the online phase outweighs the overhead incurred to perform the cost estimation.

When estimating the cost per document for processing the original query (term B of equation 6) we used 10% of the document collection. For the synthetic data set, the PDPA algorithms add ten properties into each document. However, in one experiment we varied the number of properties. Two of the properties are structure attribute properties: the maximum height of a document and the maximum children elements. The remaining are element summary properties, which are scored and ranked using Equation 4 and 5. The eight that we use are the ones that give the highest score according to Equation 4.

For the real data set, we used five element summary properties: cite, author, cdrom, ee and series. We did not use maximum height and the maximum number of children elements since for this particular data set, these properties are not effective for pruning.

6 Experimental Results

This section presents the results of five different experiments. In the first experiment, the query distribution is varied. In the second experiment, the data distribution is varied. In the third experiment, the number of documents is varied for the synthetic data. In the fourth experiment, the number of documents were varied for the real data. In the fifth experiment, the number of candidate pruning properties were varied for the synthetic data. In order to gain more confidence in the correctness of our implementation we have compared the output of the original XQuery engine against the various variants of PDPA. We found the outputs were the same.

6.1 Experiment 1: Varying Query Distribution

Figure 13 shows the processing time for uniformly, normally, and zipf distributed queries with different query types on 10,000 uniformly, normally, and zipf distributed documents respectively. As shown in Figure 13, PDPA-G outperforms Original in all cases by up to two fold, except PC query type based on uniformly distributed data and query. The reason for this is the pruning power of PDPA properties on normally or zipf distributed data and query is much stronger than that on uniformly distributed data and query. PDPA-G also outperforms PDPA-E because of its smaller processing overhead, and outperforms PDPA-A because of its better selection of pruning properties, which has stronger pruning power for the given query. PDPA-G and PDPA-A outperform the original query processing for all data and query distributions for the AD and MIX query types. The reason for this is that executing AD or MIX queries requires further look-ups to determine the finishing point. PDPA uses the embedded properties to pre-determine whether the candidate document requires further examination at the early stage, and significantly reduces the number of unnecessary look-ups when executing an AD or MIX query in a top-down manner.

PDPA-E sometimes performs worse than the original processing due to the additional overhead of exhaustively searching through all possible combinations of properties. It seems that sometimes this overhead outweighs any benefits gained from using properties.

PDPA-G and PDPA-A perform similarly to or sometimes worse than the original query processing for the PC query type because in this case, original query processing can finish fairly quickly when it is determined early that a particular parent-child relationship does not exist. PDPA-G and PDPA-A perform worse than the original query processing when both the query and data are uniformly distributed. This means PDPA-G and PDPA-A are not suitable for use in situations where data is uniformly distributed.

6.2 Experiment 2: Varying Data Distribution

Figure 14 shows the processing time for normal distributed queries on 10,000 uniformly, normally, and zipf distributed documents.

As shown in Figure 14, the processing time when using PDPA-G on normally and zipf distributed data is approximately 50% less than when using Original. However, the improvement when using PDPA-G on uniformly distributed data is relatively small. The reason for this is that many candidate elements are spread over the whole uniform distributed document collection and the selectivity of most elements is relatively low. Therefore, the pruning power of PDPA would be relatively weak. However, as mentioned in the previous experiment, uniformly distributed data are not common.

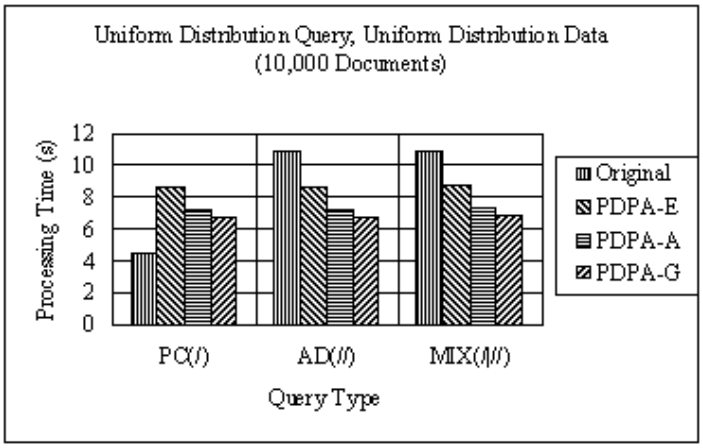
6.3 Experiment 3: Varying Number of Documents for Synthetic Data

Figure 15 shows the processing time when varying the number of documents for uniformly, normally, and zipf distributed queries on uniformly, normally, and zipf distributed documents respectively. This experiment is aimed at exploring the scalability of the various algorithms. When the number of documents is 5,000, which indicates the document collection is relatively small, using PDPA has very little advantage. The reason for this is the overhead of estimating the cost is not amortized over the small number of documents. However, when the number of the documents increases, PDPA shows much better improvement on performance for all distributions in contrast to Original. The processing time difference between PDPA-G and Original grows significantly for all distributions. This is because PDPA's superior pruning ability becomes a larger advantage as the number of documents that may be pruned increases.

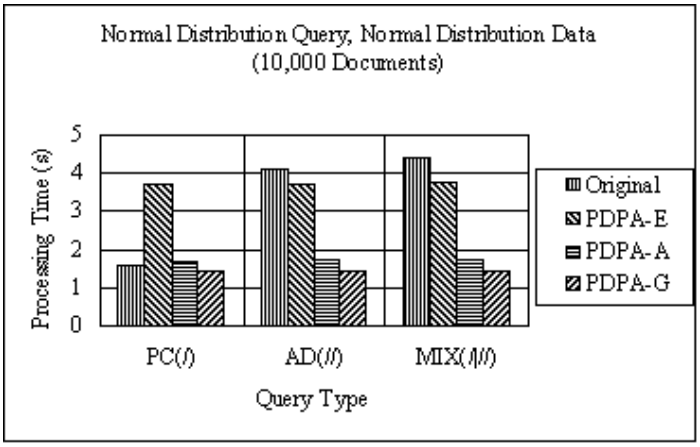
Furthermore, another observation from Figure 15(b) and (c) is that PDPA-E outperforms PDPA-A when the number of documents is relatively large. The overhead of estimating the execution cost done by PDPA-E becomes a smaller proportion of processing time and PDPA-E's ability to select better pruning properties becomes a larger advantage. PDPA-G outperforms PDPA-E in all conditions since it makes the best trade-off between computation overhead of cost estimation and performance gain from adding the properties that are effective at pruning.

6.4 Experiment 4: Varying the Number of Documents for the Real Data

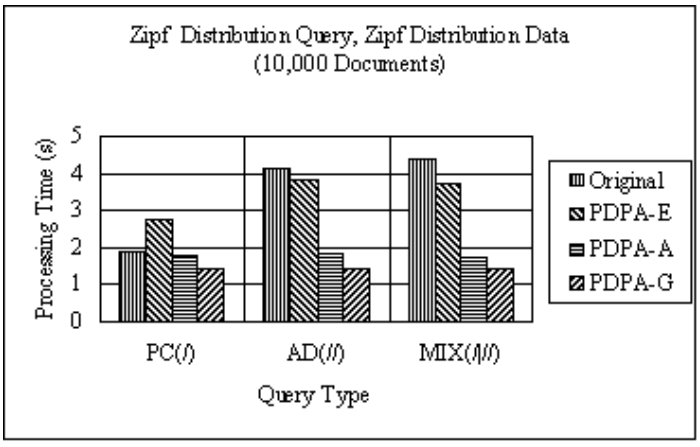
Figure 16 shows the processing time when varying the number of documents for the real DBLP data which was described in Section 5.3. In this experiment, we varied the number of documents from 10, 000 to 500, 000.



(a) Uniform Distribution



(b) Normal Distribution



(c) Zipf Distribution

Figure 13: Varying Query Distribution

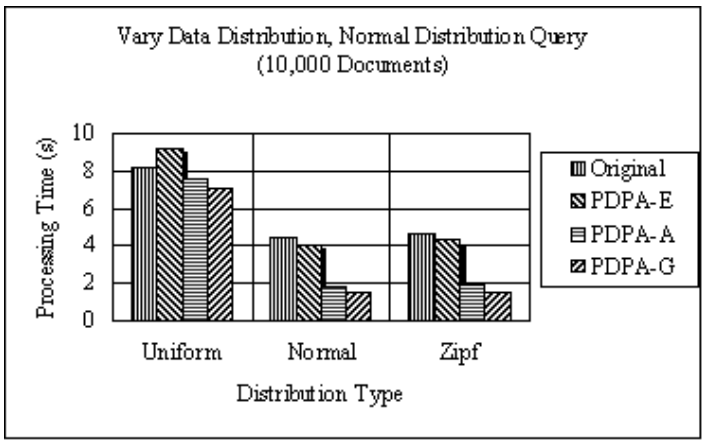


Figure 14: Varying Data Distribution

The results for this experiment show similar trends to Figure 15 (b). When we analyzed the data, we found that the data was normally distributed hence the similar results compared to the normally distributed synthetic data. These results are very encouraging since it shows that PDPA-G outperforms Original by up to two fold for the real data set.

6.5 Experiment 5: Varying the Number of Candidate Pruning Properties

Figure 17 shows the processing time when the number of candidate pruning properties used by the PDPA algorithms were varied from 2 to 16.

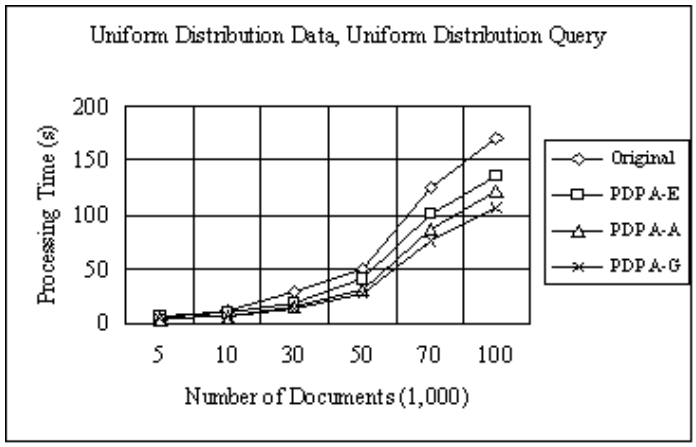
The results for this experiment show PDAP-G performed the best for all values tested. This shows the effectiveness of PDAP-G at efficiently picking the best pruning properties from among the candidate properties. PDAP-E spends too much time searching for the optimal set of pruning properties. The benefits of such an exhaustive search is outweighed by the high cost of the search process itself. PDAP-A shows poor performance when the number of candidate properties is high since it picks all the candidates irrespective of pruning effectiveness. Some of the properties are not effective at pruning and hence add to the processing cost of the query.

7 Conclusion

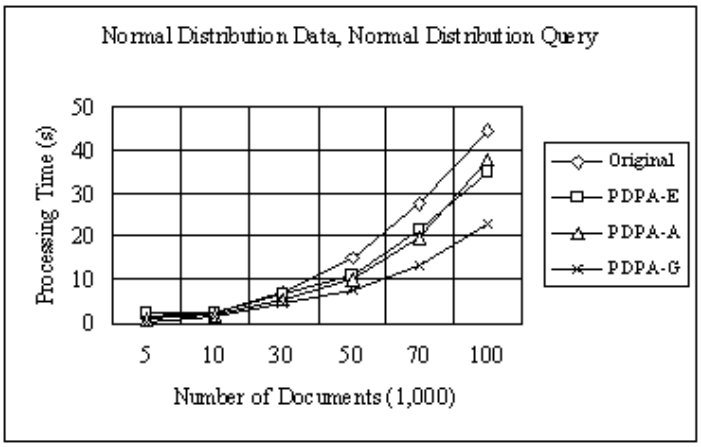
In this paper, we have proposed an effective pruning approach called the Property-Driven Pruning Algorithm (PDPA), which offers the highly desirable features of *structural query processing independence* and *plug-and-play properties*. The first feature means that PDPA can be laid on top of any existing XML query engine, which processes queries in a top-down manner. This is accomplished by pre-modifying the documents and queries before they are inserted into the query engine instead of modifying the query engine itself. The second feature means that PDPA can be customized to take advantage of different structural and usage characteristics. This is accomplished by designing the offline and online phases of PDPA so that it can work with any pruning property.

Experimental results show that PDPA consistently improves query execution performance for all the situations we tested for both real and synthetic data sets. From these experimental results, it can be concluded that PDPA's ability to pre-prune candidate documents leads to a significant speed-up of structural query processing.

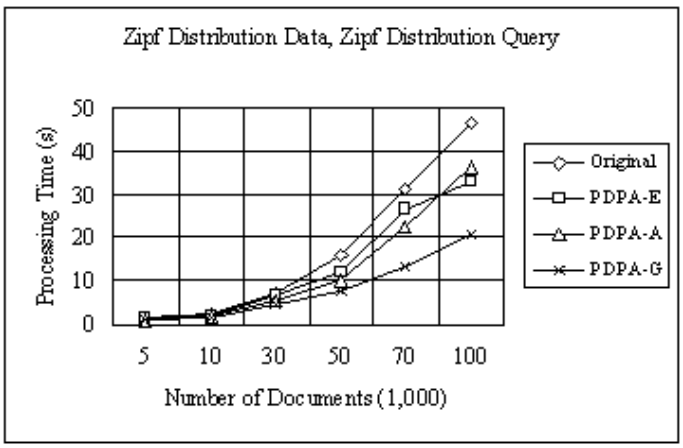
For future work, we would like to develop new pruning properties to insert into PDPA and assess the utility of the other properties described in Table 1 in detail. This will give users of PDPA a larger range of already designed properties to plug into PDPA. In addition, new PDPA selection algorithms would also be developed to suit different types of properties. Currently PDPA is designed to work for elements in XML documents. However extending it to work for attributes would not be hard. It just means parsing the XML documents for both attributes and elements when creating the properties and also embedding attribute properties inside the queries. An in depth study into how to take the most representative sample and the size of the sample for estimating cost B of Equation 6 is an important area of



(a) Uniform Distribution Query on Uniform Distribution Data



(b) Normal Distribution Query on Normal Distribution Data



(c) Zipf Distribution Query on Zipf Distribution Data

Figure 15: Varying the Number of Documents for the Synthetic Data

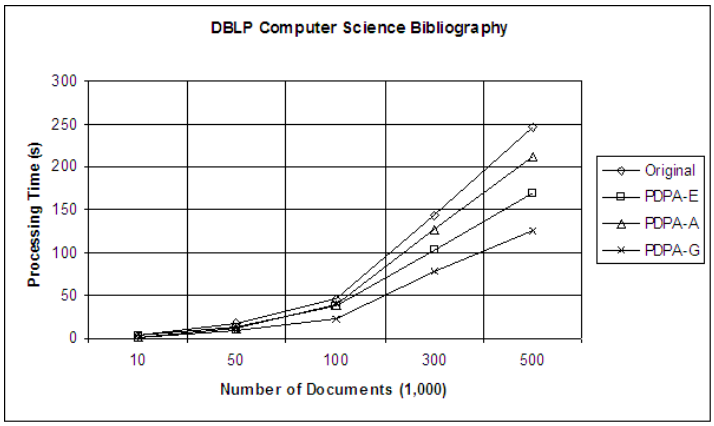


Figure 16: Varying the Number of Documents for the Real Data

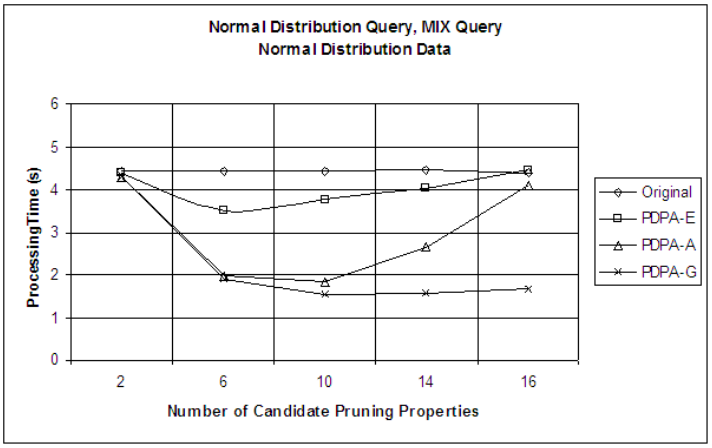


Figure 17: Varying the Number of Candidate Pruning Properties

future work. Finally, plugging PDPA into other XML query processing engines and comparing its performance is an important direction for future research. Finally, plugging PDPA into other XML query processing engines which use other state-of-the-art query processing algorithms and comparing its performance is an important direction for future research.

References

[1] S. Al-Khalifa, H. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proceedings of ICDE*, pages 141–154, 2002.

[2] Sihem Amer-Yahia, SungRan Cho, Laks V. S. Lakshmanan, and Divesh Srivastava. Tree pattern query minimization. *VLDB Journal*, 11(4):315–331, 2002.

[3] Andrey Balmin, Fatma Özcan, Ashutosh Singh, and Edison Ting. Grouping and optimization of XPath expressions in DB2®pureXML. In *Proceedings of ACM SIGMOD*, pages 1065–1074, 2008.

[4] Peter Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: A fast XQuery processor powered by a relational engine. In *Proceeedings of ACM SIGMOD*, pages 479–490, 2006.

[5] Stéphane Bressan, Barbara Catania, Zoé Lacroix, Ying Guang Li, and Anna Maddalena. Accelerating queries by pruning XML documents. *Data Knowledge Engineering*, 54(2):211–240, 2005.

[6] N. Bruno, N.Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proceedings of ACM SIGMOD*, pages 310 – 321, 2002.

[7] Q. Chen, A. Lim, and K. W. Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *Proceedings of ACM SIGMOD*, 2003.

[8] Ting Chen, Jiaheng Lu, and Tok Wang Ling. On boosting holism in XML twig pattern matching using structural indexing techniques. In *Proceeedings of ACM SIGMOD*, pages 455 – 466, 2005.

[9] Zhiyuan Chen, Johannes Gehrke, Flip Korn, Nick Koudas, Jayavel Shanmugasundaram, and Divesh Srivastava. Index structures for matching xml twigs using relational query processors. *Data Knowl. Eng.*, 60(2):283–302, 2007.

[10] S. Chien, Z. Vagena, D. Zhang, V. J. Tsotras, and C. Zaniolo. Efficient structural joins on indexed XML documents. In *Proceedings of VLDB*, pages 159 – 179, 2002.

[11] C. W. Chung, J. K. Min, and K. Shim. APEX: an adaptive path index for XML data. In *Proceedings of ACM SIGMOD*, 2002.

[12] Oracle Corporation. Oracle berkeley DB XML. viewed 28 March 2009 [ONLINE] http://www.oracle.com/database/berkeley-db/xml/index.html, 2009.

[13] Alin Deutsch and Val Tannen. Containment and integrity constraints for XPath fragments. In *Proceedings of the Internation Workshop on Knowledge Representation meets Databases*, 2001.

[14] Alin Deutsch and Val Tannen. Optimization properties for clasese of conjunctive regular path queries. In *DBPL*, pages 21–39, 2001.

[15] Iman Elghandour, Ashraf Aboulnaga, Daniel C. Zilio, Fei Chiang, Andrey Balmin, Kevin Beyer, and Calisto Zuzarte. An XML index advisor for DB2. In *Proceedings of ACM SIGMOD*, pages 1267–1270, 2008.

[16] Damien K. Fisher and Sebastian Maneth. Structural selectivity estimation for XML documents. In *Proceedings of the ICDE*, pages 626–635, 2007.

[17] Gang Gou and Rada Chirkova. Efficiently querying large XML data repositories: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(10):1381–1403, 2007.

[18] Gang Gou and Rada Chirkova. Efficient algorithms for exact ranked twig-pattern matching over graphs. In *Proceedings of ACM SIGMOD*, pages 581–594, 2008.

[19] T. Grust. Accelerating XPath location steps. In *Proceedings of ACM SIGMOD*, 2002.

[20] T. Grust, M. Keulen, and J. Teubner. Staircase join: Teach a relational DBMS to watch its (axis). In *Proceedings of VLDB*, pages 524–535, 2003.

[21] A. Halverson, J. Burger, L. Galanis, A. Kini, R. Krishnamurthy, A. Nagaraja Rao, F. Tian, S. D. Viglas, Y. Wang, J. F. Naughton, and D. J. DeWitt. Mixed model XML query processing. In *Proceedings of VLDB*, pages 225–236, 2003.

[22] H. Jiang, H. Lu, and W. Wang. Efficient processing of twig queries with OR-predicates. In *Proceedings of ACM SIGMOD*, pages 59–70, 2004.

[23] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML data for efficient structural joins. In *Proceedings of ICDE*, pages 253– 264, 2003.

[24] H. Jiang, W. Wang, H. Lu, and J. X. Yu. Holistic twig joins on indexed XML documents. In *Proceedings of VLDB*, pages 273–284, 2003.

[25] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *Proceedings of ACM SIGMOD*, 2002.

[26] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structure data. In *Proceedings of ICDE*, 2002.

[27] Jongik Kim. Advanced structural joins using element distribution. *Information Sciences*, 176(22):3300–3331, 2006.

[28] Christoph Koch, Stefanie Scherzinger, and Michael Schmidt. XML prefiltering as a string matching problem. In *Proceedings of ICDE*, 2008.

[29] W. Lian, N. Mamoulis, D. W. Cheung, and S. M. Yiu. Indexing useful structural patterns for XML query processing. *IEEE Transactions on Knowledge and Data Engineering*, 17(7), 2005.

[30] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom. Lore: A dabatase management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.

[31] Jason McHugh and Jennifer Widom. Query optimization for XML. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*, pages 315–326, 1999.

[32] Philippe Micheils, George A. Mihaila, and Jerome Simeon. Put a tree pattern in your algebra. In *Proceedings of ICDE*, pages 246–255, 2007.

[33] Gerome Miklau and Dan Suciu. Containment and equivalence for an XPath fragment. In *PODS*, pages 65–76, 2002.

[34] T. Milo and D. Suciu. Index structures for path expressions. In *Proceedings of Interational Conference on Database Theory*, 1999.

[35] Mirella M. Moro, Petko Bakalov, and Vassilis J. Tsotras. Early profile pruning on xml-aware publish-subscribe systems. In *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*, pages 866–877. VLDB Endowment, 2007.

[36] K. Hima Prasad and P. Sreenivasa Kumar. Efficient indexing and querying of XML data using modified prüfer sequences. In *Proceedings of CIKM*, pages 397–404, 2005.

[37] Praveen Rao and Bongki Moon. PRIX: Indexing and querying xml using prüfer sequences. In *Proceedings of the ICDE*, pages 288–300, 2004.

[38] Praveen Rao and Bongki Moon. Sequencing XML data and query twigs for fast pattern matching. *ACM Transactions Database Systems*, 31(1):299–345, 2006.

[39] I. Sanz, M. Mesiti, G. Guerrini, and R. Berlanga. Fragment-based approximate retrieval in highly heterogeneous xml collections. *Data Knowledge Engineering*, 64(1):266–293, 2008.

[40] Dan Suciu. Xml data repository. http://www.cs.washington.edu/research/xmldatasets/www/repository.html#dblp. Last Viewed: 11/8/2008.

[41] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *Proceedings of ACM SIGMOD*, pages 204 – 215, 2002.

[42] Shirish Tatikonda, Srinivasan Parthasarathy, and Matthew Goyder. LCSTRIM: dynamic programming meets XML indexing and querying. *Proceedings of VLDB*, 2007.

[43] Haixun Wang and Xiaofeng Meng. On the sequencing of tree structures for XML indexing. In *Proceedings of the ICDE*, pages 372–383, 2005.

[44] Haixun Wang, Sanghyun Park, Wei Fan, and Philip S. Yu. ViST: a dynamic index method for querying XML data by tree structures. In *Proceedings of the ACM SIGMOD*, pages 110–121, 2003.

[45] W. Wang, H. Jiang, H. Lu, and J. Xu Yu. Containment join size estimation: Models and methods. In *Proceedings of ACM SIGMOD*, pages 145–156, 2003.

[46] W. Wang, H. Jiang, H. Lu, and J. Xu Yu. Bloom histogram: Path selectivity estimation for XML data with updates. In *Proceedings of the VLDB*, pages 240 – 251, 2004.

[47] Y. Wu, J. M. Patel, and H. V. Jagadish. Structural join order selection for XML query optimization. In *Proceedings of ICDE*, 2003.

[48] Yuqing Wu, Jignesh M. Patel, and H. V. Jagadish. Structural join order selection for XML query optimization. In *Proceedings of ICDE*, pages 443–454, 2003.

[49] C. Zhang, J. Naughton, D. D. Witt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *Proceedings of ACM SIGMOD*, pages 425 – 436, 2001.