Indexing and Searching XML Documents based on Content and Structure Synopses

Weimin He, Leonidas Fegaras, and David Levine

University of Texas at Arlington, CSE
Arlington, TX 76019-0015
{weiminhe,fegaras,levine}@cse.uta.edu

Abstract. We present a novel framework for indexing and searching schema-less XML documents based on concise summaries of their structural and textual content. Our search query language is XPath extended with full-text search. We introduce two novel data synopsis structures that correlate textual with positional information in an XML document and improves query precision. In addition, we present a two-phase containment filtering algorithm based on these synopses that improves the searching process. Our experimental evaluation shows that our data synopses indexing scheme outperforms the standard XML indexing scheme based on inverted lists; the query evaluation based on our data synopses is more accurate than related approximate approaches that do not consider positional information; our two-phase containment filtering algorithm is more efficient than a single-phase brute force algorithm.

1 Introduction

As XML has become the *de facto* form for representing and exchanging data, there is an increasing interest in indexing and searching text-centric XML documents. Recently, XML query languages, such as XPath and XQuery, have been extended with full-text search capabilities. These queries are potentially more precise than simple IR-style keyword-based queries, not only because each search keyword can be associated with a structural context, which is typically the path to reach the keyword in a document, but structural constraints can also be used to specify the structural relationship between multiple search keywords.

Consider, for example, the running query Q used throughout the paper:

```
//auction//item[location ~ "Dallas"]
   [description ~ "mountain" and "bicycle"]/price
```

against a pool of indexed XML documents. It searches for the prices of all auction items located in Dallas that contain the words "mountain" and "bicycle" in their description. When searching for documents that satisfy this query, we do not want to waste any time by considering those that do not match the structural constraints of the query or those that do not contain the search keywords at relative positions as specified by the structural relationships in the query. For

example, we do not want to consider a document that, although has items located in Dallas, none of these items has both "mountain" and "bicycle" in their descriptions, even though there may be other items in this document, which are not located in Dallas but have both "mountain" and "bicycle" in their titles.

Current XML indexing techniques, such as [6], combine structure indexes and inverted lists extracted from XML documents to fully evaluate a full-text query against these indexes and return the actual XML fragments that answer the query. This is accomplished by performing containment joins over the sorted inverted lists derived from the element and keyword indexes. Since all elements and keywords have to be indexed, such indexing schemes may consume a considerable amount of disk space and may be time-consuming to build. More importantly, the query evaluation based on these indexes may involve many joins against very long inverted lists that may consider many irrelevant documents at the early stages. Although many sophisticated techniques have been proposed to improve these joins by skipping the irrelevant parts of these lists, it is still an open research problem to make them effective for a large document pool.

In this paper, we present a new framework for indexing and searching schemaless XML documents based on condensed summaries extracted from the structural and textual content of the documents. Instead of indexing each single element or term in a document, we extract a structural summary and a small number of data synopses from the document, which are indexed in a way suitable for query evaluation. The result of a query evaluation is a list of document locations that best match the query. A document location includes meta information about the document, such as the document URL, structural summary, and description. Based on the retrieved meta information, the client can choose some of the returned document locations and request a full evaluation of the query over the chosen documents using any existing XML query engine and return the XML fragments as query answers. To find all indexed documents that match the structural relationships in a query, the *query footprint* is extracted from the query and is converted into a pipelined plan to be evaluated against the indexed structural summaries. The resulting document locations that match the query footprint are further filtered out using the data synopses associated with the search predicates in the query and returned to the client.

2 Related Work

There is an increasing interest in recent years for full-text search over XML documents. Khalifa *et al* [1] propose a bulk algebra called TIX, which integrates simple IR scoring schemes into a traditional pipelined query evaluator for an XML database. TeXQuery [2] supports a powerful set of fully composable full-text search primitives, which can be seamlessly integrated into the XQuery language. In [3], the authors present a framework that relaxes a full-text XPath query by dropping some predicates from its closure and scoring the approximate answers using predicate penalties. XRank [5] extends Google-like keyword search to XML. The authors propose an algorithm for scoring XML elements that

takes into account both hyperlink and containment edges. A recent work, XKSearch [7], introduces the concept of *smallest lowest common ancestors* (SLCAs) and proposes two efficient algorithms, Indexed Lookup Eager and Scan Eager, for keyword search in XML documents according to the SLCA semantics. However, all these proposals consider fully indexing and querying XML documents, which may involve costly containment joins among long inverted lists to evaluate a full-text XML query.

3 Query Language and Meta-Data Indexing

Our query language is XPath extended with a full-text search predicate $e \sim S$, where e is an XPath expression. This predicate returns true if at least one element from the sequence returned by e matches the *search specification*, S. A search specification is a simple IR-style boolean keyword search that takes the form

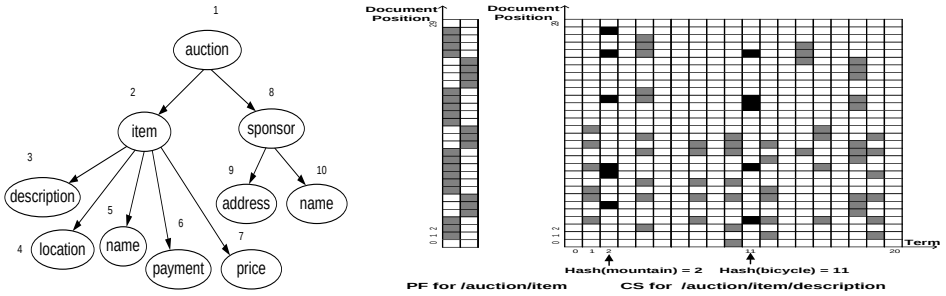
$$\text{“term”} \mid S_1 \text{ and } S_2 \mid S_1 \text{ or } S_2 \mid (S)$$

where S, S_1, and S_2 are search specifications. A term is an indexed term that must be present in the text of an element returned by the expression e.

As an XML document is indexed, all essential meta-data are extracted from the document. In particular, three kinds of meta-data are indexed: *Structural Summary* (SS), *Content Synopses* (CS), and *Positional Filters* (PF).

3.1 Structural Summary

A structural summary is a tree that describes the structural make-up of the XML data in a document. It concisely captures all unique paths in an XML document. For example, the structural summary of an XML document related to auctions is shown in Figure 1(a). Each node in an SS has a tagname and Id and one SS node may be associated with many elements in the actual document.



(a) Structural Summary (b) Data Synopses

Fig. 1. Structural Summary & Data Synopses Examples

3.2 Content Synopses

A node in a structural summary is called a text node if the node contains text in the document. To capture the textual content of a document, for each text node k in the structural summary S of the document D, we construct a *content synopsis* (CS) H_p^D to summarize the textual data associated with k, where the path p is the unique simple path from the root of S to the node k in S. H_p^D is a bit matrix of size $L \times W$, where W is the number of term buckets and L is the document positional ranges of the elements that directly contain terms associated with node k. The positional information is represented by the document order of the begin/end tags of the elements. More specifically, for each term t contained directly in an element associated with k, whose begin/end position is b/e, we set all matrix values $H_p^D[i, \text{hash}(t) \bmod W]$ to one, for all $\lfloor b \times L/|D| \rfloor \leq i \leq \lfloor e \times L/|D| \rfloor$, where 'hash' is a string hashing function and $|D|$ is the document size. That is, the $[0, |D|]$ range of tag positions in the document is compressed into the range $[0, L]$. H_p^D is implemented as a B-tree index with index key p because during the query processing, we need to retrieve the content synopses of all documents for a given path p. For example, the content synopsis for SS node **description** is illustrated on the right in Figure 1(b). Each dark cell represents a bit set to one. As we can see, after the term "bicycle" is hashed to the term bucket 11, we obtain a bit vector that has 4 one-bit ranges (displayed with black color). Each one-bit range represents a **description** element that directly contains "bicycle" in the document. The start/end of a range corresponds to the document order of begin/end tag of a **description** element. Since a node in a structural summary may correspond to many elements in a document, the positional dimension is very useful information when evaluating search predicates in a query. In our running query example, both "mountain" and "bicycle" have to be in the same description element in a document to satisfy the query Q. If we had used one-dimensional Bloom filters [4], to check whether the bits for both terms are on, we may have gotten a prohibitive number of false positives. For instance, Q may have returned an unqualified document that has an item whose description contains "mountain", and another item whose description contains "bicycle". As such, term positional information is crucial in increasing the search precision. With our content synopses, we can evaluate the search predicate description $\sim$ "mountain" and "bicycle" by bitwise *anding* the vectors H_3["mountain"] and H_3["bicycle"], which are the two black-bit vectors extracted from the content synopsis in Figure 1(b). If all bits in the resulting bit vector are zeros, the corresponding document does not have both terms in the same **description** element and thus does not satisfy the search predicate.

3.3 Positional Filters

Although the positional information in CS enforces the constraint that the terms in a single search predicate must be in the same element associated with the predicate, it can not ensure that different elements associated with different search predicates are contained in the same element in a document. For example,

given the relevant bit vectors H_3["mountain"], H_3["bicycle"], and H_4["Dallas"] only, we can not enforce the containment constraint in Q that the item whose location contains "Dallas" must be the same item whose description contains "mountain" and "bicycle". To address this problem, for each non-text node n in the structural summary of a document, we construct another type of data synopsis, called *Positional Filter* (PF), denoted by F_p^D. As we did for H_p^D, F_p^D is also implemented as a B-tree with index key p. F_p^D is a bit matrix of size $L \times M$, where L is the document positional ranges of the elements associated with node n that is reachable by the label path p, and M is the number of bit vectors in F_p^D. Here, the value of M should be no less than 2 because we want to map consecutive elements in a document to different bit vectors, thus reducing the bit overlaps of consecutive elements when their mapped begin/end ranges intersect. The positional filter for SS node **item** is demonstrated on the left in Figure 1(b). The 7 one-bit ranges indicate there are 7 **item** elements in the document.

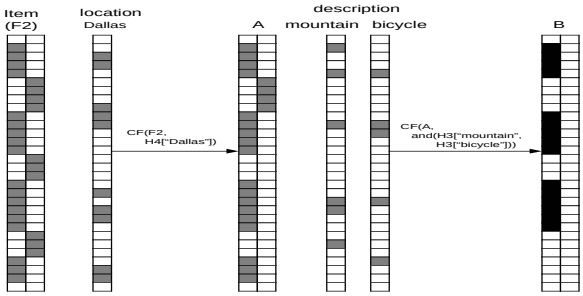


Fig. 2. Testing query Q Using Data Synopses

3.4 Containment Filtering

With positional filters, we can enforce the element containment constraints in the query using an operation called *Containment Filtering*. Let F be a positional filter of size $L \times M$ and V be a bit vector extracted from a content synopsis whose size is $L \times W$. The *Containment Filtering* $CF(F, V)$ returns a new positional filter F'. The bit $F'[i, m]$ is on iff:

$$\exists k \in [0, L] : \ V[k] = 1 \ \wedge \ \forall j \in [i, k] : \ F[j, m] = 1$$

Basically, the *Containment Filtering* copies a continuous range of one-bits from F to F' if there is at least one position within this range in which the corresponding bit in V is one. Figure 2 shows how the data synopses are used to determine whether a document is likely to satisfy the query Q (here $M = 2$). First, we do a containment filtering between the initial positional filter F_2 and the bit vector

H_4["Dallas"]. In the resulting positional filter A, only 5 one-bit ranges out of 7 in $F2$ are left. Counting from bottom to top, the $2nd$ and $4th$ one-bit ranges in $F2$ are discarded in A because there is no any one-bit range in H_4["Dallas"] that intersects with the $2nd$ or $4th$ range, which means that neither the $2nd$ nor $4th$ **item** element contains a **location** element that contains the term "Dallas". Similarly, we can do containment filtering between A and the resulting bit vector derived from the bitwise *anding* between H_3["mountain"] and H_3["bicycle"]. The 3 one-bit ranges in B indicate that 3 **items** out of 7 in F_2 satisfy all element containment constraints in the query. Thus, the document is considered to satisfy the query.

4 Query Processing

In this section, we briefly present the query processing in our framework. The first step in evaluating a full-text XPath query is deriving a query footprint from the query. A query footprint captures the essential structural components and all the *entry points* associated with the search predicates. In our running example, the query footprint of Q is:

```
//auction//item:1[location:2][description:3]/price
```

The numbers 1, 2, and 3 are the numbers of the entry points in the query footprint that indicate the places where data synopses are needed for query evaluation (one positional filter for the label path associated with entry point 1 and two content synopses for the label paths associated with the entry points 2 and 3). We have developed a general footprint derivation algorithm but, due to the space limitation, the algorithm is not presented in the paper.

Our numbering scheme, which is similar to that in [8], encodes each node k in a structural summary S by the triple (b, e, l), where b/e is the begin/end numbering of k and l is the level of k in S. Structural summaries are indexed as the mapping: $\mathcal{M}_{ss} : tag \rightarrow \{(S, k, b, e, l)\}$, where tag is the tagname of the node k in S, and b,e,l are as defined above. Thus, the key operation in the structural summary matching is a structural join between two tuple streams corresponding to two consecutive location steps in the query footprint. We leverage the iterator model in relational databases to form a pipeline of iterators derived from the query footprint to retrieve all matching structural summaries.

Let QF be the structural footprint of a query. The structural summary matching is accomplished by the function $\mathcal{SP}[\![QF]\!]$ that returns a set of tuples (ρ, S, k, b, e, l), where (S, k, b, e, l) is similar to that of $\mathcal{M}_{ss}$ and ρ is a vector of node numbers, such that $\rho[i]$ gives the node number in S corresponding to the ith entry point in QF. The function $\mathcal{SP}[\![QF]\!]$ is defined recursively based on the syntax of QF, generating structural joins for each XPath step. From the node numbers in ρ, we can derive the label paths from the structural summary that match the entry points in QF. In our running example, the label paths are `/auction/item`, `/auction/item/location`, and `/auction/item/description`.

Based on the retrieved label paths, documents that have data synopses associated with these label paths are retrieved and filtered using the containment filtering, and qualified documents are filtered out and returned to the client.

5 Hash-based Query Optimization

Using paths only as keys for data synopses indexing is not efficient because a popular path may be contained in a large number of documents and thus is associated with a large number of data synopses in the indexes. These retrieved long data synopsis lists may lead to an expensive join operation between two large lists at each step of the containment filtering. Based on the above observations, we refine our indexing scheme for data synopses and propose a hash-based two-phase containment filtering algorithm to improve query processing.

In order to reduce the number of content synopses and positional filters retrieved from the local indexes for a full label path during the containment filtering, as a document D is indexed, instead of using a full label path p as the key, we employ (p, hc) as the key to store a content synopsis H_p^D or a positional filter F_p^D, where p is the full label path associated with H_p^D or F_p^D, and hc is an integer value, which is the hash code of the document ID when mapped to a bit vector, called the *Document Mapping Vector (DMV)*. Basically, a DMV groups all the documents containing path p by the hash value of their document ID. Combined with another data structure called *Document Synopsis*, the above indexing scheme can reduce the number of data synopses retrieved for the path p during the containment filtering.

5.1 Document Synopses

In the first phase of the containment filtering, the goal is to quickly identify the documents that may contain all the path-term pairs derived from the structural summary matching and prune unqualified documents that contain only partial path-term pairs. This information derived from the first phase will guide the actual containment filtering in the second phase. To summarize, for all the documents that contain a path-term pair in the corpus, we construct a data structure called *Document Synopsis*, denoted by DS_p. Basically, for each text label path p, a document synopsis is a bit matrix of size $DL \times DW$, where DW is the number of term buckets and DL is the size of the hash table(DMV) for the document ID mapping. As a document containing the path-term pair (p, t) is indexed, p is first used as the key to find the corresponding DS_p in the indexes. Then, the term t is mapped to some bucket along the **Term** axis of DS_p to obtain the bit vector V along the **Document ID** axis, that summarizes all the documents containing (p, t). Finally, the ID of the document is mapped to some bit in V and the bit is set to one if it is zero.

The structure of a document synopsis is shown in Figure 3. The dark cells represent the one-bits. Suppose that the document synopsis is associated with the path `/biblio/book/paragraph`, since document 12 contains the path-term

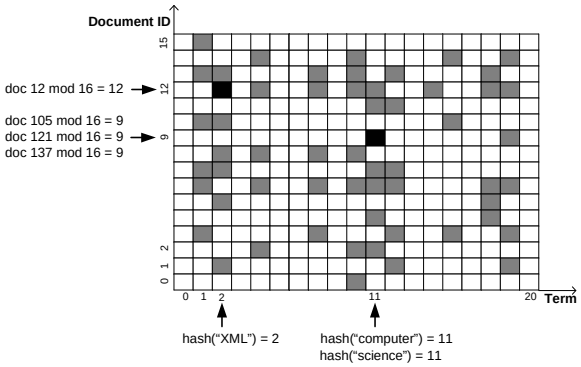


Fig. 3. A Document Synopsis Example

pair (`/biblio/book/paragraph`, "XML"), the corresponding bit is set to one, which is emphasized by a black cell in the figure. Note that different documents may be mapped to the same document ID slot and different terms may be hashed to the same term bucket.

5.2 Two-phase Containment Filtering

Based on the new indexing scheme and document synopses, we propose a two-phase containment filtering strategy to optimize our query processing, which is given in Algorithm 1. The first phase is a pre-processing stage that prunes unqualified documents that do not contain all the path-term pairs. The resulting bit vector V_f is a filter that carries information about all the documents that may contain all the path-term pairs $(p_1, t_1), (p_2, t_2), ..., (p_n, t_n)$, which is indicated by the one-bits in V_f. In the second phase, the real containment filtering is carried out with the guide of V_f. Basically, at each step of the containment filtering, (p_i, hc_i) is used as the key to retrieve all content synopsis hits or positional filter hits, where p_i is the corresponding path derived from SS matching, and hc_i is the index number of the one-bit in V_f. The goal is using V_f to avoid accessing unqualified data synopses and retrieve only the data synopses of the documents that contain all the path-term pairs, thus effectively reducing the number of data synopses retrieved from the indexes before the join operation.

6 Experimental Evaluation

We have implemented our framework using Java (J2SE 5.0) and Berkeley DB Java Edition 1.7.1 was employed as the storage manager. Our experiments were conducted on a WindowsXP machine with 2.8GHz CPU and 512M memory. The two datasets we used were synthetically generated from the XBench [10] and XMark benchmarks. The main characteristics of our datasets and data synopses size are summarized in Table 1. The query workload over each dataset is shown

Algorithm 1: Two-phase Containment Filtering

Input: p_0 /* the path associated with positional filter */
$(p_1, t_1), (p_2, t_2), \ldots, (p_n, t_n)$ /* n path-term pairs associated with content synopses */
Output: L_{PF} /* the list of positional filter hits of qualified documents */
1: $L_{PF} := emptyList$;
2: /* Obtain the filtering vector V_f in phase one */
3: **for** i = 1 to n **do**
4: Use p_i as the key to retrieve DS_{p_i} in local indexes;
5: Map t_i along the Term axis in DS_{p_i} to obtain the bit vector $V_{p_i}^{t_i}$;
6: **end for;**
7: $V_f := \bigcap_{i=1}^{n} V_{p_i}^{t_i}$; /* bitwise anding all bit vectors */
8: /* Do actual containment filtering with the guide of V_f in phase two */
9: **for** each one-bit b_j in V_f **do**
10: k : = the index number of b_j in V_f;
11: $L_0^{b_j}$:= the positional filter list retrieved using (p_0, k) as the key;
12: **for** i = 1 to n **do**
13: $L_i^{b_j}$:= the positional filter list retrieved using (p_i, k) as the key;
14: $L_0^{b_j} := CF(L_0^{b_j}, L_i^{b_j})$;
15: **end for;**
16: $L_{PF} := L_{PF} \cup L_0^{b_j}$;
17: **end for;**
18: **return** L_{PF};

in Table 2. For the indexing scheme comparison experiment, we chose XBench as the dataset because the dataset size is the key factor for this experiment. To measure the query precision and our optimization algorithm, we chose XMark as the dataset because the number of documents is the key factor for these two experiments.

Table 1. Data Set Characteristics and Data Synopses Size

Data Set	Data Size (MB)	Files	Avg. File Size (KB)	Avg. SS Size (Byte)	Avg. CS Size (Byte)	Avg. PF Size (Byte)
XBench	1050	2666	394	432	20564	178
XMark	55.8	11500	5	417	306	16

6.1 Indexing Scheme Comparison

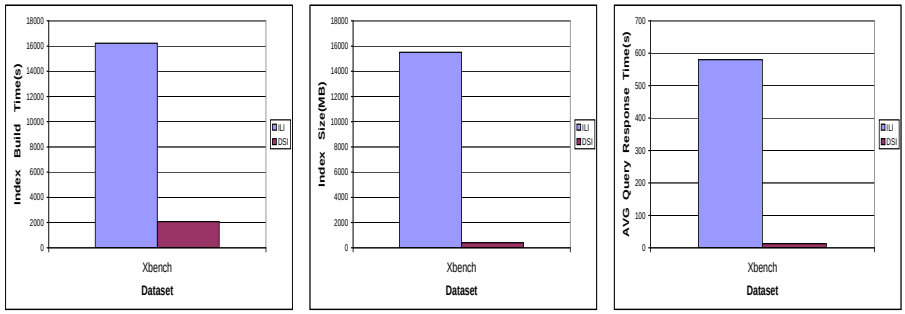
To demonstrate the efficiency of our Data Synopses Indexing (DSI) scheme, we implemented the standard inverted list-based XML indexing scheme(ILI) [8] using Berkeley DB and compared DSI with ILI in terms of space and time cost. Figure 4 shows that, since DSI avoids indexing each single element and keyword in the database, DSI consumes less than 8% index build time than ILI and the index size of DSI is only about 3% of that of ILI. The query response time of DSI is over 40 times faster than that of ILI because the query evaluation is over concise data synopses instead of over full inverted lists.

Table 2. Query Workload over Each Dataset

Dataset	Query	Query Expression
XMark	Q1	/site//item[location ~ "United"][payment ~ "Creditcard" and "Check"]/description
XMark	Q2	//regions//item[location ~ "States"][payment ~ "Creditcard" or "Cash"]/name
XMark	Q3	/site//item[location ~ "United"][payment ~ "Creditcard"]/description
XMark	Q4	//regions//item[location ~ "States"][payment ~ "Check"]/quantity
XMark	Q5	/site//item[description//text ~ "gold"]/name
XMark	Q6	/regions//item[description//text ~ "character "]/payment
XMark	Q7	//closed_auction[type ~ "Regular"][annotation//text~ "heat"]/date
XMark	Q8	//closed_auction[annotation//text~ "heat" or "country"]/seller
XMark	Q9	//closed_auction[annotation//text~ "heat" and "country"]/buyer
XMark	Q10	//closed_auction[annotation//text~ "country"]/type
XBench	Q11	/article//body[abstract/p ~ "hockey"][section/p ~ "hockey" and "patterns"]/section
XBench	Q12	//article//body[section/p ~ "regular"][abstract/p ~ "hockey" or "patterns"]/abstract
XBench	Q13	/article//body[section/subsec/p ~ "hockey"][abstract/p ~ "hockey"]/abstract
XBench	Q14	/article//body[section/subsec/p ~ "regular"][abstract/p ~ "patterns"]/section
XBench	Q15	/article//body[section/p ~ "patterns"][abstract/p ~ "patterns"]/abstract
XBench	Q16	/article//body[section/p ~ "hockey"][abstract/p ~ "patterns"]/abstract
XBench	Q17	//prolog[keywords/keyword ~ "bold" or "regular"][title~ "regular"]/authors
XBench	Q18	//prolog[keywords/keyword ~ "bold"][title~ "bold"]/title
XBench	Q19	//prolog[genre ~ "Travel"] [keywords/keyword ~ "bold" or "stealth"]//author/name
XBench	Q20	//prolog[genre ~ "Travel"] [keywords/keyword ~ "bold"]/title

6.2 Query Precision Measurement

Since our data synopsis correlates content with positional information, we call it Two-Dimensional Bloom Filter (TDBF). We implemented the traditional One-Dimensional Bloom Filter (ODBF) [4] in our framework and compared the query precision of our TDBF with that of ODBF. The result is shown in Figure 5. The false positive rate of a query is defined as (1 - the relevant set size/the answer set size). We exploited XQuery engine Qizx/open [9] to evaluate each XMark query over the dataset to obtain the accurate relevant set for the query. From Figure 5, we can see that for queries Q_1, Q_2, Q_3, Q_4 and Q_7, the false posi-



(a) Index Build Time (b) Index Size (c) Query Response Time

Fig. 4. Comparison between ILI and DSI

tive rate of ODBF is over two times higher than that of TDBF because each of these queries contains multiple search predicates and the positional dimension in TDBF can effectively remove false positives during the containment filtering. For queries Q_5, Q_6, Q_8 and Q_{10}, two approaches produce the same false positive rate because each query contains only a single search predicate and the search predicate contains only one term or the boolean operator is "or". In that case, TDBF performs only a bitwise *oring* operation and the positional information is not helpful to reduce the false positives. For query Q_9, TDBF is better than ODBF because although it contains a singe search predicate, the boolean operator is "and", in which case the positional information can reduce the false positives during the containment filtering. The above result shows that TDBF is superior to ODBF when multiple predicates are presented in the query or a single predicate contains multiple disjunctive terms.

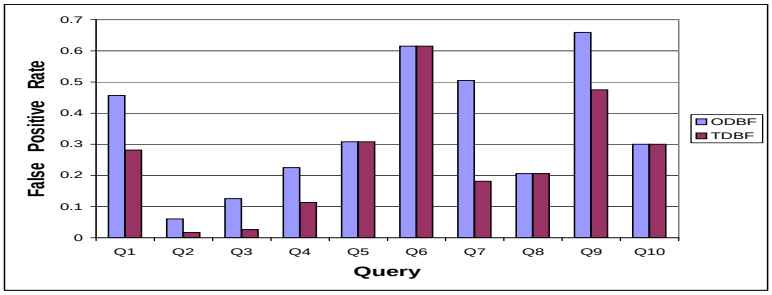


Fig. 5. Query Precision Comparison with One Dimensional Bloom Filter

6.3 Efficiency of Optimization Algorithm

To examine the efficiency of our Two-Phase Containment Filtering(TPCF) algorithm, we implemented One-Phase Containment Filtering(OPCF) algorithm in our framework. Then we evaluated all the XMark queries in Table 2 and compared the query response times between TPCF and OPCF. As we can see from Figure 6, for queries Q_1, Q_2, Q_3, Q_4 and Q_7, TPCF is one time faster than OPCF because these queries contain more search predicates, which may involve more containment filtering steps and more joins between long data synopsis lists during the query evaluation. In that case, TPCF can effectively prune the unqualified document locations in the first phase, thus reduce the overall query response time. For the remaining queries, since each query only contains one search predicate and only one containment filtering step is needed in the query evaluation, the query efficiency improvement from TPCF is not very significant. This result indicates that our two-phase containment filtering algorithm is more efficient when multiple search predicates are present in the user query.

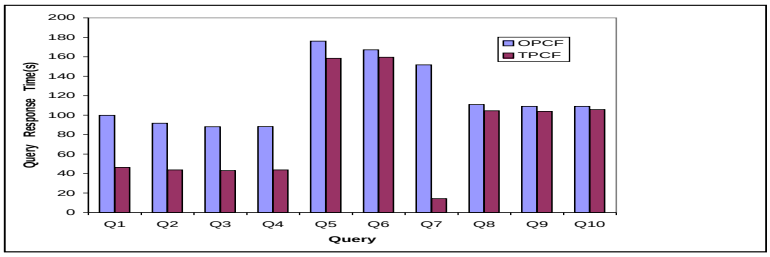


Fig. 6. Efficiency of Two Phase Containment Filtering Algorithm

7 Conclusion

We have presented a framework for indexing and searching XML documents based on condensed summaries extracted from the structural and textual content of the documents. Our indexing scheme is more efficient than traditional indexing schemes based on full inverted lists. Our data synopses correlate content with positional information and result in a more accurate evaluation of textual and containment constraints in a query. Our two-phase containment filtering algorithm can accelerate the searching process.

Acknowledgments: This work is supported in part by the National Science Foundation under the grant IIS-0307460.

References

1. Al-Khalifa S., Yu C., Jagadish H.V.: Querying Structured Text in an XML Database. Proc. of ACM SIGMOD, San Diego, USA (2003)4-15
2. Amer-Yahia S.,Botev C.,Shanmugasundaram S.: TeXQuery: A Full-Text Search Extension to XQuery. Proc. of the 13th Int. Conference on World Wide Web(WWW), New York, USA (2004)583-594
3. Amer-Yahia, S., Lakshmanan L.V.S., Pandit S.: FleXPath: Flexible Structure and Full-Text Querying for XML. Proc. of ACM SIGMOD, Paris, France (2004)83-94
4. Bloom B.: Space/Time Trade-offs in Hash Coding with Allowable Errors. Communications of the ACM, Vol. 13,(1970)422-426
5. Guo L., Shao F., Botev C., Shanmusundaram J.: XRANK: Ranked Keyword Search over XML Documents. Proc. of ACM SIGMOD, San Diego, USA (2003)16-27
6. Kaushik R., Krishnamurthy R., Naughton J.F., Ramakrishnan R.: On the Integration of Structure Indexes and Inverted Lists. Proc. of ACM SIGMOD, Paris, France (2004)779-790
7. Xu Y., Papakonstantinou Y.: Efficient Keyword Search for Smallest LCAs in XML Databases. Proc. of ACM SIGMOD, Maryland, USA (2005)537-538
8. Zhang C.: On Supporting Containment Queries in Relational Database Management Systems. Proc. of ACM SIGMOD, Santa Barbara, USA (2001)425-436
9. Qizx/open. http://www.axyana.com/qizxopen/.
10. XBench. http://se.uwaterloo.ca/∼ddbms/projects/xbench/.