Efficient Dynamic Indexing and Retrieval of XML Documents using Three-Dimensional Quasi-BitCube

Biren Shah Abhilash Gummadi Jong P. Yoon Vijay Raghavan

University of Louisiana at Lafayette, P. O. Box 44330, CACS, Lafayette, LA 70504, USA
{bshah, axg1814, jyoon, raghavan}@cacs.louisiana.edu

ABSTRACT

XML is a new standard for exchanging and representing data on the Internet. Techniques for indexing and retrieval of XML data is drawing increasing attention since they enable one to access certain parts of retrieved documents easily. However, they provide little or no support for adding new documents to an existing document collection, requiring instead that the entire collection be re-indexed. Modern applications, based on XML indexing and retrieval, operate in dynamic environments that require frequent additions to document collections. An indexing structure known as the BitCube has been proposed to perform fast query processing on XML documents. One of the major disadvantages in using a BitCube is its inefficient memory management. In this paper, we propose an extended BitCube, also known as a Quasi-BitCube, which manages memory much more effectively while maintaining the same query processing efficiency of a BitCube. Our work also aims at enabling dynamic (or incremental) indexing of new documents to an existing Quasi-BitCube, without requiring the entire collection to be re-indexed. We have performed an extensive set of experiments to test the effectiveness of both the Quasi-BitCube index structure and the proposed dynamic algorithm to create that indexing structure. The results show that, Quasi-BitCube manages memory much more efficiently than the BitCube, without compromising on the query processing time. Our results from the experiments to test the performance of our dynamic indexing algorithm show that it provides better update and search costs than earlier schemes like the one used in XQEngine, with acceptable space overheads.

1. INTRODUCTION

A majority of traditional business applications, transactional systems and enterprise applications rely on relational databases to maintain their data. As portals, knowledge management systems and even e-mail have joined the mainstream and have become indispensable daily tools, a typical organization's enterprise information is no longer maintained as structured data alone. Typically, structured data are the data with a repeated structure that can be easily stored in the data tables of a relational database. Semi-structured databases [1], unlike traditional databases, do not have a fixed schema known in advance. Broadly speaking, semi-structured data is self-describing and can model heterogeneity more naturally than either relational or object-oriented database systems. The eXtensible Markup Language (XML) [3] is a commonly used data modeling technique for such data, and the application of common XML tools blurs the distinction in handling structured and unstructured data.

XML is a simplified subset of the Standard Generalized Markup Language (SGML). It provides a file format for representing data, a schema for describing data structure, and a mechanism for extending and annotating Hyper-Text Markup Language (HTML) with semantic information. The XML data model carries both data and schema information, being naturally suitable to represent semi-structured data. It is a standard for representing and exchanging information on the Internet.

As XML is an evolving data representation format, the awareness and acquaintance of XML among the database developers and users is not adequate. As more and more data are being represented in XML format, more tools for maintaining the XML data are developed. As XML has become a part of critical databases, the performance of such tools has become a matter of concern. Research on indexing XML databases is being actively pursued and is delivering efficient and effective algorithms.

The representation of documents in XML paved way for the possibility of content-based retrieval. The widespread use of XML in digital libraries, product catalogues, scientific data repositories and across the web prompted the development of appropriate searching and browsing methods for XML documents. As enterprise applications (or, web services) continue to build upon XML, it is critical that they include a search functionality that is fully compatible with XML. In order to optimize query processing, the data need to be organized (indexed) in a way that facilitates efficient retrieval. Without indexes, the database may be forced to conduct a full data scan to locate the desired data record, which can be a lengthy and an inefficient process. Additionally, modern applications operate in dynamic environments that require frequent additions to document collections. There is an urgent need for an XML indexing and retrieval technique that not only supports dynamic indexing of new documents but also aids in efficient query processing.

The rest of the paper is organized as follows. Section 2 describes some of the related works in this area. Section 3 introduces some of the preliminary operations used elsewhere in the paper. The proposed indexing approach is described in Section 4. Section 5 describes the dynamic indexing algorithm for our index structure. In section 6, we provide experimental results to access the properties of indexing and dynamic indexing based on our approach and we compare it with previous approaches. In section 7, we summarize the results of our study, draw conclusions and identify future work.

2. RELATED WORK

Among the types of indexes supported or under exploration by commercial database vendors are B+ trees [15], hash indexes [15], signature files [6], inverted files [21], latent semantic indexing [14] and R-trees [8]. These indexing techniques can be evaluated based on access/insertion/deletion time and disk-space needed. Each indexing technique differs in its implementation and target use and at the same time offers the potential to improve query performance for different applications.

Although XML can support both structure and content-based information retrieval, efficient indexing is an important

problem in improving the performance of XML query processing. Typical indexing techniques [6, 8, 14, 15, 21], from the database and information retrieval communities, however, are still not satisfactory. It is partly because they cannot scale much beyond their current point to larger collections, and partly because semantic and structural equivalencies are not efficiently checked and maintained in the indexes.

Index structures for semi-structured data have been developed in recent years. Examples of such indexes for semi-structured data are XQEngine [11], Dataguides [7], XMill [13], Toxin [16] and ViST[18]. A new data structure, called X-tree [2], has been introduced for storing very high dimensional data.

To overcome the problem of efficiently managing large collections of XML data, we have proposed a new 3-dimensional bitmap indexing technique (BitCube) in [19]. A BitCube is conceptually defined to store information in the form of bits pertaining to the existence of relationships between documents, paths and words. It supports bit wise operations to handle various types of queries and this is what makes it highly efficient in terms of query processing. In spite of this advantage, it consumes large volumes of memory. The BitCube structure is a sparse structure and, when maintained in main memory, creates a memory bottleneck.

Full-text information retrieval systems have traditionally been designed for archival environments. Almost all of the above indexing approaches were developed for large static document collections and they provide little or no support for adding new documents to an existing document collection, requiring instead that the entire collection be re-indexed.

[4] proposed a technique for fast incremental indexing of full-text information retrieval using inverted files. Their method, however, achieves best performance by limiting the number of times an inverted list will be relocated due to its growth, thereby requiring large batches of new documents to be added to the index to reduce the overall number of updates.

[10] proposed an effective mechanism for incremental update of indices in structured documents. It uses an implementation technique of Bottom Up Scheme (BUS) [17] in a relational database management system to facilitate the incremental update of indices. To conserve space, BUS saves indexing information in the leaf nodes, whereas in intermediate nodes, it is computed at run time. As a result, if a user wants to get information at an intermediate level, all the term frequencies at leaf nodes need to be accumulated to the corresponding ones in the intermediate level, which may take a certain amount of time and thereby affecting the overall retrieval performance. To facilitate the accumulation of term frequencies into the corresponding internal nodes, each element is assigned a unique element identifier according to the order of the level-order tree traversal. This eventually leads to costly updates for certain changes in the element structures.

In this paper, we contribute to two different indexing enhancement schemes. First, to overcome the memory problem, we propose an extension of our BitCube indexing structure, known as Quasi-BitCube. Quasi-BitCube is fine-tuned to manage memory much more optimally and at the same time retains the same query processing efficiency of a BitCube. We compare our new indexing time and size with our earlier work that uses BitCube. Second, we propose an efficient algorithm for dynamic indexing of new XML documents to an existing index structure (i.e. Quasi-BitCube), without requiring the entire collection to be re-indexed. We compare our dynamic indexing time with XQEngine [11] and the traditional approach of indexing, where the entire document collection is re-indexed. From the retrieval perspective, we compare the query processing time of our new index enhancement schemes with *BitCube* [19] and *XQEngine* [11] for different types of queries.

3. PRIMITIVE OPERATIONS

3.1 Density of an XML Document

An XML document can be described by its density. Let D denote any document collection. Let $e_1, e_2, \ldots, e_m$ be the m paths and $w_1, w_2, \ldots, w_n$ denote the n words in a given document d. Let N_w and N_e be the total number of unique words and paths, respectively, in the entire document collection D. $d(i,j) = 1$ if d contains word w_j in path e_i, and 0 otherwise. Density of d is defined as:

$$density(d) = \frac{C}{N_e \times N_w}, \quad (1)$$

where C is the number of unique (e,w) pairs contained in d and is defined as:

$$C = |(e_i,w_j)|_{mxn}, \quad (2)$$

such that $d(i,j) = 1$, for $1 \le i \le N_e$, $1 \le j \le N_w$. A document d is t-popular if $density(d) \ge t$, $(0 \le t \le 1)$ for a given t, which is the density threshold. A document d is t-unpopular if $density(d) < t$, $(0 \le t \le 1)$. For example, Table 1 shows a document bitmap for a sample document d. e_1, e_2, e_3 represent the paths and w_1, w_2, w_3, w_4, w_5 represent the words contained in d. Let the document collection D to which document d belongs have 6 unique paths and 20 unique words. Using equation (1) and (2), density of d is given by $density(d) = 6/(6x20) = 0.05$.

3.2 Index Value of a Document Bitmap

In the index structure that we are using, each document has an associated index value, which represents a positional value in the index structure, useful in efficiently accessing the document bitmap. For example, from Table 2.a, $indexValue(d_1) = 1$, $indexValue(d_2) =$, and so on.

Table 1: Example (Density)

(e_i,w_j)	w_1	w_2	w_3	w_4	w_5
e_1	1	0	1	0	0
e_2	1	0	0	1	0
e_3	0	1	0	0	1

Table 2.a: Example (Density and Index Values before Incremental Indexing)

Document	d_1	d_2	d_3
Density	0.4	0.3	0.2
Index value	1	2	3

Table 2.b: Example (Density and Index Values after Incremental Indexing)

Document	d_1	d_2	d_3	d_4	d_5
Density	0.4	0.3	0.2	0.25	0.35
Index value	1	3	5	4	2

3.3 Shift Operator

When new documents are added incrementally into the existing index structure, the index value of each document d_i is shifted by a value s_i in order to allow insertions of new documents at their appropriate positions. Let ip and $ip_{(new)}$ be the

index values of a document d before and after dynamic indexing, respectively. Shift of d is defined as:

$$shift(d) = ip_{(new)} - ip \quad (3)$$

For example, Table 2.a shows the density and index values of documents d_1, d_2, d_3 in the original index structure. Documents d_4 and d_5 are incrementally added into the index structure corresponding to Table 2.a. Table 2.b shows the density and index values of documents d_1, d_2, d_3, d_4, d_5 in the dynamically updated index structure. Using equation (3), $shift(d_1)$ = 1-1 = 0, $shift(d_2)$ = 3-2 = 1, and so on.

3.4 Document Ordering

Let $p_1, p_2, \ldots, p_N$ be the density values of N documents $d_1, d_2, \ldots, d_N$. The N documents are said to be in order determined by its density if and only if the following 2 conditions hold true:

i. $indexValue(d_1)=1$, $indexValue(d_2)=2$, …, $indexValue(d_N)=N$

ii. $p_1 \geq p_2 \geq \ldots \geq p_N$

For example, if density is used to determine document ordering, from Table 2.b, we see that a proper ordering of documents is d_1, d_5, d_2, d_4, d_3.

4. QUASI-BITCUBE

An XML document is defined as a set of (e,w) pairs, where e denotes an element path and w denotes a word in a path. A BitCube [19] is a 3-dimensional bitmap index. A BitCube represents a set of documents together with a set of paths and a set of words for each path. A BitCube for XML documents is defined as BitCube = (d,e,w,b), where d denotes an XML document, e denotes a path, w denotes a word, and b is either 0 or 1, the value for a bit in BitCube (if e contains w, the bit is set to 1, and 0 otherwise). Figure 1.a shows an example BitCube.

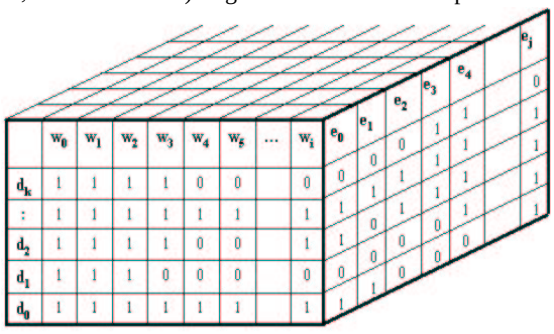

Figure 1.a: BitCube

A Quasi-BitCube is an extension of BitCube that retains many of its powerful features and at the same time has a lot more structural advantage. The main advantage of a BitCube (or a Quasi-BitCube) lies in its high-speed query processing ability. The primary enhancement of a Quasi-BitCube is the rectification of memory constraints in the so-called BitCube. The structure is significantly reorganized to contain as small number of bits as deemed pragmatic while retaining the same query processing and indexing times. Unlike other index structures, where I/O and caching are important (since they do not fit entirely into the main memory) for efficient access, our index structure is dense and stores information in the form of bits and hence can entirely fit into the main memory. I/O operations are, however, required and are dominant during the index creation or modification.

The BitCube structure is sparse. There is a *documents bit vector (dbv)* for each (e,w) pair. The bit is SET if the (e,w) pair exists in the corresponding document and RESET otherwise. In a real-time environment, it is very unlikely that a document will contain all the paths and the words contained in all the other documents. This means that numerous bits towards the top of each documents bit vector are consuming space and are superfluous (see Figure 1.b).

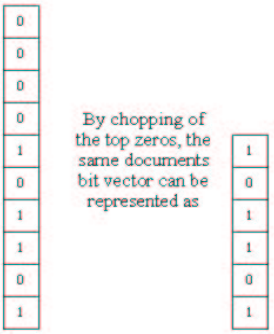

Figure 1.b: Documents Bits Vector Example

To circumvent the possibility that a document contains all the paths and words contained in all the other documents, we use document ordering, based on the density of each document. Our heuristic here is to have the documents with high density placed towards the bottom of the index structure. A document with a greater number of unique (e,w) pairs is more popular than a document with fewer number of unique (e,w) pairs. So, if there exists a document that contains all of the paths and words contained in all the other documents, then it will have the highest density, and hence will be placed towards the bottom. Additionally, documents can have similar properties and popularities. In such cases, documents can be clustered [19] to further resolve the issue of ordering equally popular documents. Thus, the resulting index structure will be dense towards the bottom and sparse towards the top. The redundant (zero bits) bits from the sparse end of the structure can now just be discarded.

Let N_d be the total number of documents in any given collection D. Let N_w and N_e be the total number of unique words and paths, respectively, in D. Let $dbv(e,w)$ denote the documents bit vector at any given (path, word) pair, (e,w). Let $size(e,w)$ denote the length (in number of bits), of $dbv(e,w)$, which is the number of bits actually stored, after the 0's towards the top of $dbv(e,w)$ are discarded. The total size (in number of bytes) of the BitCube or a Quasi-BitCube index structure is given by:

$$total\ size(BitCube\ or\ Quasi\text{-}BitCube) = \frac{\sum_{i=1}^{N_w}\sum_{j=1}^{N_e} size(i,j)}{8}\ bytes \quad (4)$$

Since, for a BitCube, $size(i,j) = N_d$, $\forall\ i, j, 1 \leq i \leq N_w, 1 \leq j \leq N_e$, equation (4) reduces to

$$total\ size(BitCube) = \frac{\sum_{i=1}^{N_w}\sum_{j=1}^{N_e} N_d}{8} = N_d \times N_e \times N_w\ bytes \quad (5)$$

The main characteristics of Quasi-BitCube index structure can be summarized as follows:

i. Simple, yet dynamic structure
ii. Memory efficient structure
iii. Efficient query processing due to bit-wise operations
iv. Documents are ordered based on their density

5. DYNAMIC INDEXING

Creating an index (static) for semi-structured data (like XML) that is physically stored in flat files requires two parses of the original data as explained below:

i. Before building any index structure, the entire data space needs to be parsed once to extract important metadata information like its basic structure, relationships, etc. This information could be used to optimize the initial size of the index, create mapping tables for efficient access, determine the ordering, etc.

ii. Once the metadata information is made available, during the second parse, the index structure is then built in a way that minimizes not only its size but also the overall access cost.

For any indexing structure, dynamically updating the existing one when new data arrives plays a very crucial role. We propose an algorithm to efficiently perform this task of dynamically indexing the Quasi-BitCube index structure. Our algorithm differs from the traditional indexing, in the sense that the dynamic index can be obtained by a simple permutation of our original index. The resultant index structure, after dynamic indexing, is the same as the index structure that would be created if we had used the traditional approach of re-indexing all the documents from scratch.

5.1 Notation

Let d_1, d_2, …, d_N be the N documents in the original document collection. These N documents are indexed into a Quasi-BitCube Q. Let e be any path, w be any word and d be any document. Let d_{N+1}, d_{N+2}, …, d_{N+M} be the M documents to be incrementally added into the Quasi-BitCube index structure Q. Let O_e and O_w be the total number of unique paths and words, respectively, in the N documents in the original document collection. Let C_e and C_w be the new number of unique paths and words, respectively, in the M documents to be incrementally added. Let ip and $ip_{(new)}$ be the index values of a document before and after dynamic indexing, respectively. Let dbv_{old} and dbv_{new} be the documents bit vectors of the original and the dynamically updated index structure, respectively. Let $dbv(e,w)$ be the documents bit vector at (e,w) at any given point of time during program execution.

$$dbv(e,w)[i] = \begin{cases} 1 \text{ iff the document corresponding to the index value } i \text{ contains word } w \text{ in path } e. \\ 0 \text{ otherwise} \end{cases}$$

5.2 Dynamic Indexing Algorithm

Our algorithm efficiently supports dynamic indexing of new documents into the already created index structure. The process of computing the index data for the new set of documents to be added incrementally is efficiently merged with the ordered documents of the existing index structure, to create a new structure that reflects the effective and unified indexing organization of the entire document collection (old and new) as a whole. Figure 2 depicts the sequence of operations performed for dynamic indexing. The actual algorithm is shown in Figure 3. To improve the efficiency, our algorithm totally eliminates the second parse of the new document collection while using a little more main memory. The amount of indexing time saved by doing this is significantly more than the amount of extra memory used. Our algorithm uses dynamic hashing as a 2-D index structure that maps all the unique (e,w) pairs as keys, in the new set of documents to be incrementally added, to the corresponding documents bit vector as their values. The density and other metadata information for dynamic indexing are computed in parallel with the creation of the 2-D index. Creating such a 2-D index structure totally eliminates the second parse of the new document collection, thereby further improving on the indexing time, with acceptable space overheads.

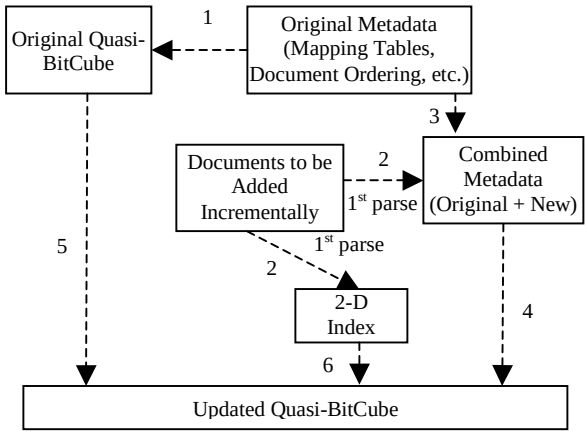

Figure 2: Incremental Indexing

i. **For each** document d_i (i = 1 … N) in the original document collection
   $p_i = density(d_i)$
ii. **Set** $count = 0$
iii. **For each** document d_j (j = N+1 … N+M) to be incrementally added **{**
   $p_j = density(d_j)$
   For each (e,w) pair in document d_j {
      If there does not exist $dbv(e,w)$ in 2D-index then
         Create new $dbv(e,w)$
      $dbv(e,w)[count] = 1$
      Update 2D-index structure.
   }
   $count = count + 1$
}
iv. Determine the new document ordering using the density values from (i) and (iii) above.
v. **For each** document d_i (i = 1 … N) in the original document collection
   $s_i = shift(d_i)$
vi. **For each** document d_j (j = N+1 … N+M) to be incrementally added
   $ip_{j(new)} = indexValue(d_j)$
vii. Create a new (e,w) index plane of size (O_e+C_e)x(O_w+C_w).
viii. **For each** document d_i (i = 1 … N) in the original document collection, determine its new index value and update the dbv_{new} as **{**
   $ip_{i(new)} = ip_i + s_i$
   For each (e,w) pair of document d_i in dbv_{old}
      $dbv_{new}(e,w)[ip_{i(new)}] = dbv_{old}(e,w)[ip_i]$
}
ix. **For each** dbv corresponding to an (e,w) pair in 2D-index **{**
   $count = 0$
   For each document d_j (j=N+1 … N+M) to be incrementally added, set $ip_{j(new)}{}^{th}$ bit of the dbv_{new} as {
      $dbv_{new}(e,w)[ip_{j(new)}] = dbv(e,w)[count]$
      $count = count + 1$
   }
}

Figure 3: Incremental Indexing Algorithm

6. EXPERIMENTS

6.1 Goals

A detailed set of experiments were carried in order to achieve the following goals:

i. To measure the memory savings attributable to the Quasi-BitCube index structure by comparing its index size and time with that of BitCube for different data distributions.
ii. To measure the efficiency of our dynamic indexing algorithm by comparing it with traditional indexing and XQEngine.
iii. To determine the optimal incremental batch size.
iv. To measure the retrieval efficiency of the Quasi-BitCube index structure by comparing it with BitCube and XQEngine for different types of query operations.

6.2 Data Sets

To achieve the above goals, four sets of experiments were conducted using synthetic and real data sets (document collections). The specification of the computer used in our experiments is Intel Pentium 4 with 2.2 GHz processor and 1GB RAM running Red-Hat Linux version 8.0.

The synthetic data set was generated using IBM's XML Generator [9]. Unlike real data sets, the distribution of data within the document collection can be controlled in the case of synthetic data sets. We varied certain parameters in the IBM's XML Generator for generating document collections having four different data distributions: Sparse and Skewed, Sparse and Uniform, Dense and Skewed, Dense and Uniform.

We used the public XML database DBLP [12] as the real data set. The public XML database DBLP contains over 400,000 XML documents that can be grouped into one of the following seven categories: Thesis (79), Article (147758), WWW (39), Book (1029), Proceedings (4092), In-proceedings (258423) and In-collections (1106). The numbers in the bracket indicate the corresponding number of DBLP documents in that category. Each document of DBLP corresponds to a publication in any one of the seven categories listed above. The version 0.56 of XQEngine was used for experimental comparisons.

6.3 Performance Evaluation

Instead of the absolute value, we sometimes report the Gain in the time taken to dynamically index a given set of new documents. Gain is defined as:

$$Gain = \frac{dynamic\ indexing\ time\ of\ a\ given\ indexing\ scheme}{dynamic\ indexing\ time\ of\ the\ traditional\ scheme} \quad (6)$$

For example, a gain value of 0.1 implies that the given dynamic indexing scheme is faster than the traditional scheme of re-indexing all the documents by a factor of 10. Thus, the notion here of maximizing the overall gain means that its value in equation (6) should be minimized.

To determine the incremental batch size that maximizes the overall performance, we plot the Geometric Mean (GM) values of the Average Time per Document Insertion (ATDI) and the Gain for different incremental batch sizes. GM is computed as follows:

$$GM = \sqrt{ATDI \times Gain} \quad (7)$$

where,

$$ATDI = \frac{time\ required\ to\ incrementally\ add\ given\ documents}{number\ of\ documents\ incrementally\ added} \quad (8)$$

To maximize the overall performance, the GM values should be as small as possible.

6.4. Experiments and Results

6.4.1 Experiment I (Measure Memory Savings)

This experiment was performed to measure the effectiveness of the Quasi-BitCube index structure by comparing its index size and time with that of the BitCube for different data distributions.

The total number of unique words and paths contained in each synthetic document collection was fixed to 5000 and 4, respectively. Figure 4 compares the index size of Quasi-BitCube and BitCube for different data distributions with increasing number of documents. The size of the BitCube index is independent of the distribution of the data in the document collection. Since Quasi-BitCube orders documents based on their density, the index size highly depends on the distribution of the data in the document collection. The results show that for all four data distributions, the Quasi-BitCube index size is less than the BitCube index size. The Quasi-BitCube index structure saves most for sparse and skewed data than all the other distributions and least for dense and uniform data.

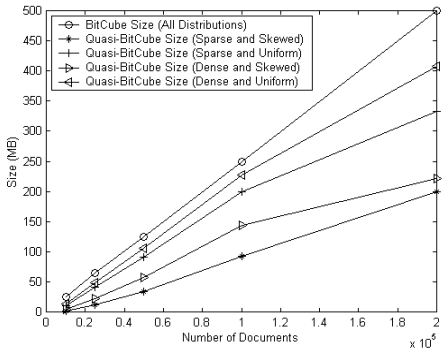

Figure 4: Experiment I: Index Size Comparison (Synthetic Data)

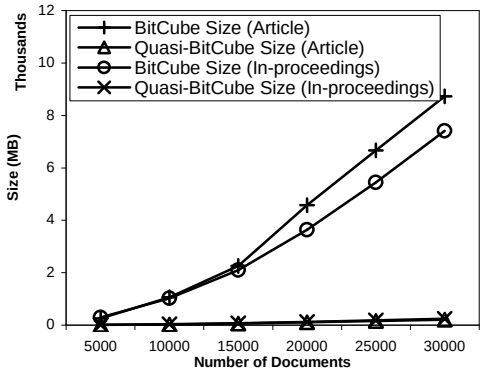

Figure 5: Experiment I: Index Size Comparison (Real DBLP Data)

Figure 5 compares the index size of Quasi-BitCube and BitCube with increasing number of documents selected from the real data (DBLP) collection. The documents were randomly selected from two of its largest collections, i.e. from article and in-proceedings category. Each document record contains information about a publication and the likelihood of two

distinct publications containing similar information is very less. Hence, the entire collection contains large number of unique words (i.e. low probability of word repetition). As a result, unlike the synthetic data set, the total number of unique words contained in each DBLP document collection increases almost linearly with the size of the collection. The degree of sparseness is thus very high, as a result of which the improvement is even higher than synthetic data sets. Quasi-BitCube index size clearly outperforms the BitCube index size.

The index times of both the structures were about the same. Due to space limitations, we do not show those results here. Thus, we see that, when compared to BitCube, the Quasi-BitCube structure saves significant amount of index memory without compromising on the indexing time.

6.4.2 Experiment II (Measure Efficiency)

This experiment was performed to measure the efficiency of our dynamic indexing algorithm by comparing it with traditional indexing and XQEngine.

The total number of unique words and paths contained in the synthetic document collection was fixed to 10000 and 12, respectively. The size of the synthetic document collection was fixed to 10000 documents. From the real data (DBLP) collection, we randomly selected 10000 documents from the article category. Efficiency is then measured by varying the incremental batch size. The corresponding gain values for the synthetic and the real data sets are shown in Figure 6 and Figure 7, respectively.

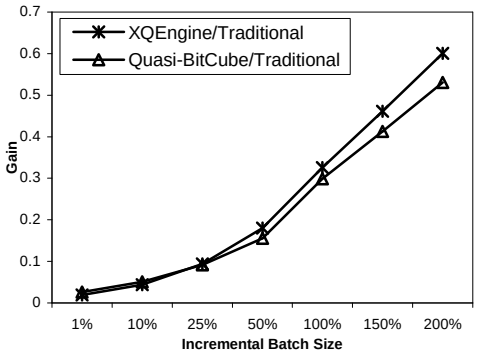

Figure 6: Experiment II: Dynamic Index Time Comparison by varying Incremental Batch Size (Synthetic Data)

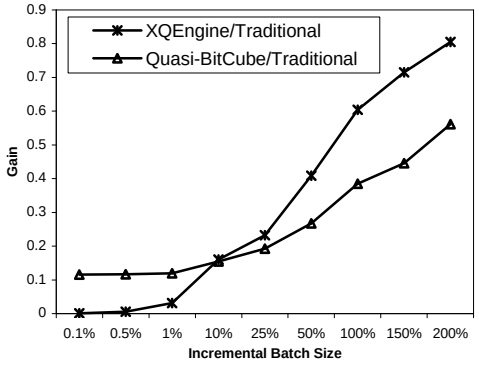

Figure 7: Experiment II: Dynamic Index Time Comparison by varying Incremental Batch Size (Real DBLP Data)

The results show that the incremental update time of our algorithm is significantly better than the traditional scheme of re-indexing the entire document collection. XQEngine performance is good for very small updates. The performance of our dynamic indexing algorithm, when compared with XQEngine, improves with the increase in the batch size and eventually outperforms it.

6.4.3 Experiment III (Determine Optimal Batch Size)

This experiment was performed to determine the optimal incremental batch size that not only minimizes the average time required per document insertion but also maximizes the overall gain.

For the synthetic data set used in Figure 6 and real data set used in Figure 7, the corresponding GM results are shown in Figure 8 and Figure 9, respectively.

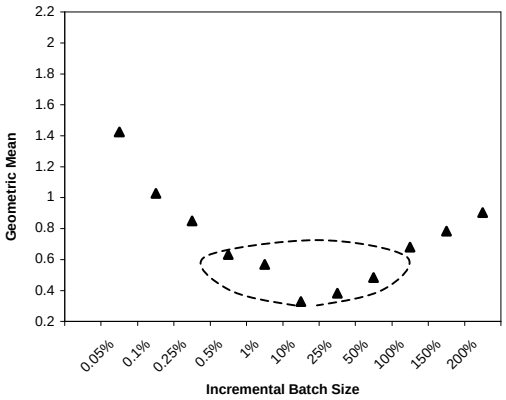

Figure 8: Experiment III: Geometric Mean to determine Optimal Batch Size (Synthetic Data)

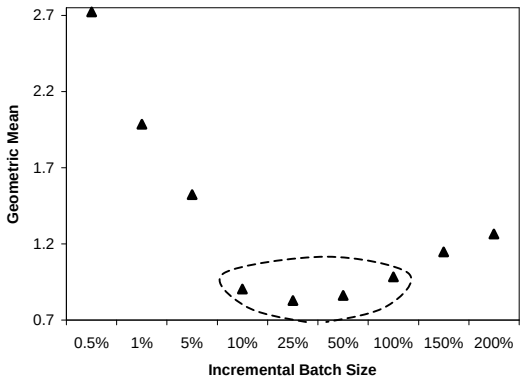

Figure 9: Experiment III: Geometric Mean to determine Optimal Batch Size (Real Data)

The results show that the GM values are high for very small and very large batch sizes. The GM initially decreases with the increase in the batch size. After it has reached its optimal value, it then follows a reverse trend and starts increasing with the increase in the batch size. The desirable optimal region is marked with dotted lines. We can say that the performance of our dynamic indexing approach is close to optimal when the GM values fall into the desired region.

6.4.4 Experiment IV (Measure Retrieval Performance)

This experiment was performed to measure the retrieval efficiency of the Quasi-BitCube index structure by comparing it with BitCube and XQEngine for different types of query operations.

We compared the query processing time for three different operations: word slice, path slice and dice. A path slice takes a path as input and returns a set of documents with words associated with the given path. A word slice takes a word as input and returns a set of documents with paths associated with the given word. A dice operation is a combination of multiple path and word slices.

We found that the query processing using Quasi-BitCube is at least as efficient as BitCube for all the three types of query operations. In [19], we have already shown that the query performance of BitCube is significantly better than that of XQEngine for word and path slice operations. The current version of XQEngine does not support complex queries (e.g. dice) involving logical AND-OR operations. Due to this limitation, we cannot compare our dice operation results with that of the XQEngine. Due to space limitations, we do not show the detailed results.

7. CONCLUSION AND FUTURE WORK

The main contributions of the paper are:

i. Quasi-BitCube, a memory efficient indexing scheme extended from BitCube is proposed. Since the information stored is in the form of bits, the entire index structure fits into the main memory and hence I/O operations are no longer a concern during information retrieval. I/O, however, plays a dominant role during index creation or modification. Our results show that Quasi-BitCube manages memory much more effectively and at the same time retains the same query processing efficiency of a BitCube. The execution time of Quasi-BitCube for different query operations is much more efficient than XQEngine.

ii. Efficient dynamic indexing algorithm that supports incremental addition of new XML documents to an existing index structure, without requiring the entire collection to be re-indexed. Experiments show that our dynamic indexing scheme provides better update and search costs than the traditional scheme, with acceptable space overheads. As the incremental batch size increases, our dynamic indexing algorithm outperforms not only the traditional scheme, but also XQEngine.

Just recently, XQEngine has released its new version 0.60. In the future, we plan to evaluate our current work against this new release. Also, there is a growing demand of XML in the areas relating to XLinks, XPointers and Security. As part of our future work, we plan to extend our index structure to meet these growing demands.

References

[1] S. Abiteboul, P. Buneman, D. Suciu, "Data on the Web: from Relations to Semi-structured Data and XML", Morgan Kaufmann, (1999).

[2] S. Berchtold, D.A. Keim, H. P. Kriegel, "The X-Tree: An Index Structure for High-Dimensional Data", in *Proceedings of International Conference On Very Large Data Bases*, Bombay, India, (1996), pp. 28-39.

[3] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, "Extensible Markup Language (XML) 1.0 (Second Edition)", W3C Recommendation (http://www.w3.org/TR/REC-xml), (2000).

[4] E. Brown, J. Callan, W. B. Croft, "Fast Incremental Indexing for Full-Text Information Retrieval", In *Proceedings of 20th International Conference on Very Large Databases*, Santiago, Chile, (1994), pp. 192-202.

[5] C. Chan, Y. Ioannidis, "Bitmap Index Design and Evaluation", In *Proc. of International ACM SIGMOD Conference*, Seattle, WA, (1998), pp. 355-366.

[6] C. Faloutsos, "Signature Files: Design and Performance Comparison of Some Signature Extraction Methods", In *ACM SIGMOD*, Montreal, Canada, (1985), pp. 63-82.

[7] R. Goldman, J. Widom, "DataGuides: Enabling Query Formulation and Optimization in Semi-structured Databases", In *Proc. of the Intl. Conference on Very Large Databases*, Athens, Greece, (1997), pp. 436-445.

[8] A.. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," In *ACM SIGMOD* (B. Yormark, ed.), Boston, MA, (1984), pp. 47-57.

[9] IBM XML Generator, *http://www.alphaworks.ibm.com/tech/xmlgenerator*, Date: 1st May 2003, Time: 6:55 p.m.

[10] H. Jang, Y. Kim, D. Shin, "An Effective Mechanism for Index Update in Structured Documents", In *Proc. of the 8th Intl. Conference on Information and Knowledge Management*, Kansas City, MO, (1999), pp. 383-390.

[11] H. Katz, "XQEngine", *http://sourceforge.net/projects/xqengine/*, Date: 25th February 2004, Time: 9:45 a.m.

[12] M. Ley, DBLP database web site, http://www.informatik.uni-trier.de/ley/db, 2000.

[13] H. Liefke, D. Suciu, "XMill: An Efficient Compressor for XML Data", In *Proceedings of ACM SIGMOD Conference*, Dallas, TX, (2000), pp. 153-164.

[14] C. Papadimitriou, H. Tamaki, P. Raghavan, S. Vempala, "Latent Semantic Indexing: A Probabilistic Analysis," In *Proc. of 17th ACM Symp. on Principles of Database Systems*, Seattle, WA, (1998) pp. 159-168.

[15] R. Ramakrishnan, J. Gehrke, "Database Management Systems", McGraw-Hill, 2000.

[16] F. Rizzolo, A. Mendelzon, "Indexing XML Data with ToXin", In *Proceedings of 4th Intl. Workshop on the Web and Databases*, Santa Barbara, CA, (2001), pp. 49-54.

[17] D. Shin , H. Jang , H. Jin, "BUS: An Effective Indexing and Retrieval Scheme in Structured Documents", In *Proc. of the 3rd ACM Conference on Digital Libraries*, Pittsburgh, PA, (1998), pp.235-243.

[18] H. Wang, S. Park, W. Fan, P. Yu, "XML Indexing and Compression: ViST: A Dynamic Index Method for Querying XML Data by Tree Structures", In *Proceedings of ACM SIGMOD Conference*, San Diego, CA, (2003), pp. 110-121.

[19] J. Yoon, V. Raghavan, V. Chakilam, L. Kerschberg, "BitCube: A Three-Dimensional Bitmap Indexing for XML Document", In *Journal of Intelligent Information Systems*, *17* (2001), pp. 241-254.

[20] J. Yoon, A. Hafez, V. Raghavan, "Query Rewriting for Multimedia XML Data," In *Proc. of the 6th Intl. Workshop on MIS*, Chicago, IL, (2000), pp. 62-71.

[21] J. Zobel, A. Moffat, K. Ramamohanarao, "Inverted Files Versus Signature Files for Text Indexing", In *ACM Trans. on Database Sys.*, *23* (1998), pp. 453-490.