Practical Indexing XML Document for Twig Query*

Hongzhi Wang[1], Wei Wang[2,3], Jianzhong Li[1], Xuemin Lin[2,3], and Reymond Wong[2]

[1] Harbin Institute of Technology, Harbin, China
{wangzh, lijzh}@hit.edu.cn
[2] University of New South Wales, Australia
{weiw, lxue, wong}@cse.unsw.edu.au
[3] National ICT of Australia, Australia

Abstract. Answering structural queries of XML with index is an important approach of efficient XML query processing. Among existing structural indexes for XML data, F&B index is the smallest index that can answer all branching queries. However, an F&B index for less regular XML data often contains a large number of index nodes, and hence a large amount of main memory. If the F&B index cannot be accommodated in the available memory, its performance will degrade significantly. This issue has practically limited wider application of the F&B index.

In this paper, we propose a disk organization method for the F&B index which shift part of the leave nodes in the F&B index to the disk and organize them judiciously on the disk. Our method is based on the observation that the majority of the nodes in a F&B index is often the leaf nodes, yet their access frequencies are not high.

We select some leaves to output to disk. With the support of reasonable storage structure in main memory and in disk, we design efficient query processing method). We further optimize the design of the F&B index based on the query workload . Experimental results verified the effectiveness of our proposed approach.

1 Introduction

XML has been widely used as a format of information exchange and representation over the Internet. Due to its hierarchical structures, XML data can be naturally modeled as a labeled rooted tree. As a result, most XML query languages, such as XPath and XQuery, allow user to retrieve part of nodes of the tree that satisfy certain structural constraints. For example, query $a/b[d]/c$ on XML tree in Figure 1 only retrieve nodes with tag c whose parent has tag b and contains at least one child with tag d.

Efficient query processing technologies of XML data are in great demand. One major challenge of XML query processing is the queries involving path

* This work was partially supported by ARC Discovery Grant – DP0346004 and the Defence Pre-Research Project of the 'Tenth Five-Year-Plan'of China no.41315.2.3.

constraints. Using index to directly retrieve qualified nodes is a nature way to accelerate such path queries. Consequently, a number of such indexes have been proposed, including DataGuides [2], 1-index [8], F&B index [5]. Among them, F&B index is the smallest index that can answers all branching queries. However, the F&B index tends to have a large number of nodes, and hence occupying a large amount of memory. For example, the complete F&B index for 100M XMark benchmark datas [11] has 1.35 million nodes. It is obvious that if the available memory size is limited such that the F&B index cannot be accommodated in the memory, its performance will degrade significantly.

Such shortcoming limits the application of F&B index. One possible method to solve this problem is to use virtual memory managed by operation system. It is obviously not a good way. Because the main memory exchange strategy is not designed for query processing. Another solution is build approximate index. Current approximate F&B indexes mainly focus on approximate of 1-index that only supports single path query. Therefore this problem is still not resolved.

An observation of F&B index for tree-structured XML document is that about half of its nodes are leaf node(And during query processing with F&B index, leaf nodes are often accessed at last step. Based on this observation, we design our method of storing parts of leaves to disk. The storage structure of leaf nodes in disk should support the cooperation with index in main memory. And the choice of which nodes to output to disk is another problem to solve in this paper.

Our contributions in this paper includes:

- A strategy of cooperation of main memory and disk F&B index is presented. So that main memory size problem of F&B index is solved.
- We design a leave storage strategy and compacted structure of index in main memory. Such structure saves main memory further more without affecting the efficiency of query.
- Optimized strategies of choosing nodes to output to disk is designed. The strategies consider two instances, with workload and without workload.
  - In the instance without workload, based on cost model formulation and experiment, it can be seen that the optimization interval is limited so that random selection method is practical.
  - In the instance with workload, we design a heuristic method to select optimized node set to output to disk.

This paper is organized as follows. Section 2 presents some preliminaries. The storage structure and query processing strategy are discussed in section 3, The node selection strategies of instances without and with workload are presented in section 4. Section 5 gives results of experiments and the discussion. In section 6, related work is presented and conclusions are drawn in section 7.

2 Preliminary

XML data are usually modeled as labeled rooted trees: elements and attributes are mapped to nodes in the trees and direct nesting relationships between two

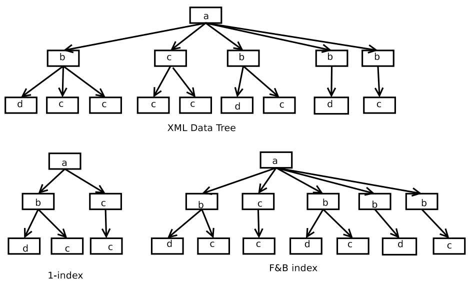


Fig. 1. The Example of Index

elements are mapped to edges between the corresponding nodes in the tree. In this paper, we only focus on element nodes; it is easy to generalize our methods to other types of nodes defined in [12].

All structural indexes for XML data take a path query as input and report *exactly* all those matching nodes as output, via searching within the indexes. Equivalently, those indexes are said to *cover* those queries. Existing XML indexes differ in terms of the classes of queries they can cover. DataGuide [2] and 1-index [8] can cover all *simple path queries*, that is, path queries without branches. [5] showed that F&BIndex is the minimum index that covers all branching path queries. We note that if the XML data is modeled as a tree, its 1-index and F&BIndex will also be a tree. Each index node n in an F&B index is associated with its *extent*, which is the set of data nodes in the data tree that the index node n represents for.

We show an running example in Figure 1; specifically, we show an example XML data tree, its 1-index, and its F&B Index. To distinguish nodes with the same tag, we append an sequence number after its label. For instance, both $b1$ and $b2$ in the 1-index have the label b. In the 1-index, all the second level b element in the data tree are classified as the same index node with tag b; this is because all those node cannot be distinguished by their incoming path, which is a/b. However, those b are classified into three groups in the F&BIndex; this is because branching path expressions, e.g., $a[c]/b$ and $a[d]/b$, can distinguish them. Compared to the 1-index which has only 6 nodes, the F&BIndex has much more nodes (10 in our example). It can be shown that in the worst case, an F&BIndex has the same number of nodes as the document does.

3 Disk Storage of F&B Index

In this section, we propose and discuss both in-memory and disk organizations when we shift parts of leaf nodes of an F&B index to disk in order to fit the rest of the index to the limited amount of main memory. Query processing methods based on this new organization are also discussed.

3.1 Store Leaves of F&B Index to Disk

One of the problems that limits the wide application of the F&B index is its large size for many irregular real datasets. When available main memory is not enough to accommodate an F&B index, one naive solution is to reply on the virtual memory managed by the operating system. It is obviously not a good solution, as the virtual memory replacement policy is not designed for database query processing, let alone specific access pattern for the XML index.

In order to solve this problem, a natural approach is to judiciously select part of the F&B index to the disk under the available memory constraint. We propose to shift the leaf nodes of the F&B index to the disk before considering the internal nodes. This strategy is based on the following two observations on the statistical nature and access patterns on the F&B index:

– We observe that leaf index nodes constitute the major part for many tree-modeled XML datasets. For example, we show the ratio of number of leaf index node versus the number of total index nodes for three typical XML datasets (DBLP, XMark, and Treebank) in Table 1. We can observe that the ratio is above 50% for all the three datasets. Therefore, shifting the leaf nodes of an F&B index to disk can effectively reduce the memory requirement.
– Traversal-based query processing techniques are commonly used on the F&B index. Our second observation is that, during tree traversal, query processing can stop in the internal nodes; even if the query results reside on extents of the leaf nodes, they are the last part to be accessed. Furthermore, for many complex branching queries, when the branching predicates do not contain value comparison, they can be evaluated without accessing the extents of the leaf nodes, because the semantics of the branching predicates are to test the existence of a particular type of node satisfying the branching path constraint.

Even though shifting some internal nodes to disk can further save main memory, processing queries on the structure obtained by such shifting will brings more disk I/O than on the structure obtained only by shifting leaves to disk in average. It is because when processing query by traversal index, interval nodes have more chances to be accessed than leave nodes.

The second design choice is the data organization of the in-memory F&B index as well the disk portion of the F&B index. A naive solution would be to replace all the in-memory pointers with the disk-based pointers. However, such simple solution is not likely to work well in practice, because following random disk-based pointers usually incur random I/Os, which is extremely costly. Therefore,

Table 1. Leaf Nodes Ratio for Three Typical XML Datasets

Dataset	#nodes	#index_nodes	#leaf_nodes	Ratio
DBLP	4620206	3477	4122	84.4%
XMark	2048193	294928	516935	57.1%
Treebank	2437667	1253163	2280279	55.0%

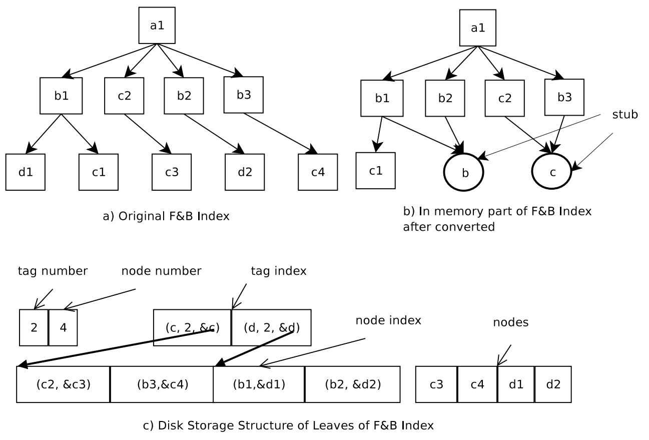


Fig. 2. In-memory and Disk Organizations of the F&B Index

by taking into consideration of the query processing efficiency, we summarize the desiderata of a good organization as:

1. The in-memory part of the F&B index should cooperate seamlessly with the portion of index on the disk;
2. The I/O cost spent to answer queries on the disk portion of the F&B index should be minimized.

Our proposed solution consists of two parts accordingly:

1. In order to satisfy the first requirement, we introduce special **stubs** which serves as the connections between main memory part of the index and the disk part. A stub is extremely memeory-efficient in that it consists of a label name only; it is yet powerful enough for all the query processing.
2. We employ clustering and indexing techniques for the disk portion of the F&B index. Index nodes stored on the disk are clustered according to their tag names as well as incoming paths. Indexes are also built on it to support fast retrieval during query processing.

As an example, the F&B index with partial nodes shifted to the disk is depicted in Figure 2, where $a1$ is an index node with tag a, $b1$, $b2$ and $b3$ are index nodes with tag b, $c1$, $c2$, $c3$ and $c4$ are index nodes with tag c. In c) of Figure 2, $\&c$ and $\&d$ points to the address of the first node with tag c and tag d, respectively. We will discuss the details of the above design on this example in the following.

Stub. A special type of nodes, *stubs*, is introduced, which serves as the connection between main memory index and the disk part. For each stub, we only keep the tag of this node. We use an optimization that merges stubs with the same name into one to further save the memory. We will show shortly that this does not affect our query processing ability or efficiency. Furthermore, many branching predicates only needs to check the existence of index node with a particular tag rather than retrieving a specific index node or its extents, this optimization will not slow down the query processing of such branching queries. In our example index shown in Figure 2, suppose we choose nodes d1, d2, c3, c4 to be shifted to the disk. As shown in Figure 2(b), in the in-memory part of the index, d1 and d2 are replaced with a stub with tag name "d"; c3 and c4 are replaced by a stub with the tag name "c" (both in circle). To see why such organization does not hinder branching predicate checking, let's assume the query is $a/b[c]/d$. When node b3 is accessed, we need to check whether it satisfies the branching predicate, $[c]$; we only need to access the stub labeled "c" and can return `true` without the need to access the actual index node c4 which are now stored on the disk.

Clustering and Indexing. Index nodes on the disk are clustered based on their tag names and incoming paths. This is based on the observation that the leaf index nodes relevant to an XML query often have the same tag name and incoming path; as a result, such cluster strategy has the advantage of reducing physical I/O during the query processing. As shown in Figure 2(c), the four nodes that have been shifted to the disk, i.e., c3, c4, d1, and d2, are laid out contiguously according to the clustering criteria above.

Based on the clustering criteria, additional indexes are built to accelerate the access of F&B index nodes on the disk. Specifically, we build a *tag index* and a *node index*. The tag index is based on the property that all index nodes on the disk with the same tag are clustered together. Each entry in tag index points to the first *node index entry* of the same tag in the *node index*. For example, there are two entries in the tag index (shown in Figure 2(c)) for the running example. For the entry (c, 2), 'c' is the tag name; 2 is the number of F&B index nodes with the corresponding tag (i.e., 'c') on the disk; the associated pointer links to the first entry in the node index with the corresponding tag. The node index records the incoming path information for each corresponding F&B index nodes.

Information for query includes its the incoming path number and the pointer to the node with extent. For example, in c) of Fig 2, the entry (c2,&c3) of node index represents node c3 in F&B index in a). In this entry, c2 is the id of its father in F&B index and &c3 is the address of its real node. In c), part "nodes" is the storage of nodes with their extent. The pointer &c3 in entry(c2, &c3) points to c3 in this part. In order to accelerate the searching of index, in tag index, all entities are sorted by tag. In each cluster of node index, the nodes have the same tag name and sorted by their parent id.

In addition to the above three components, we also record the total number of tags and total number of nodes, for all the F&B index nodes shifted to the disk. The complete disk organization is thus shown in Figure 2.

The main motivation and the advantage of employing such clustering and indexing is to reduce the number of disk I/Os during the query processing. For example, in order to process query $a//d$ by traversing the F&B index shown in Figure 2(c), d1 and d2 will be accessed. Without clustering, d1 and d2 are likely to reside on different places on the disk. Without tag and node indexes, sequence scan of all the disk-resident F&B nodes will be required. As a contrast, with our clustering and indexing schemes, in order to retrieve d1, we first look up the tag index to find the starting position of node index entries with tag 'd'; we then search in the node index the address of the F&B node with tag 'b' and its parent node id equal to 'b1'; finally we fetch the node d1 from the disk by following the disk pointer. While searching node index for these two 'd' nodes, with the clustering, the disk blocks containing 'd' entries in node index need only be read into the main memory for one time because of the existence of buffers. Similarly the actual F&B index node d2 is likely to be in the buffer after d1 is accessed. Therefore, we can greatly reduce the physical I/O cost for many queries from two I/Os per F&B index node (one I/O to obtain the address the node in the node index and another to access the node) to a much smaller number. In fact, our experiments verified that with a reasonable amount of buffer pages, such saving is significant.

Building the F&B Index. We give the procedure to convert a totally in-memory F&B index to our F&B index structure once a subset of the leaf nodes, S, is given. We will discuss the criteria and algorithms to choose S in Section 4.

1. Sort the nodes in S by their tags and their parent IDs.
2. Gather the total number of tags and nodes in S.
3. Generate the node index from S and tag index from node index.
4. Compute the address of each node on the disk and fill it to corresponding entry in the node index. Compute the address of the first entry of each tag in the node index and fill it to corresponding entry in the tag index.
5. Store all these parts of storage in the structure discussed above.

We give an estimation of the memory that our method can save in Equation 1. The first item of Equation 1 is necessary memory size of when all nodes are in the main memory. The second item is the size of the stubs.

$$Size_{save} = \sum_{node \in S} size(node) - |T| \cdot size(tag) \tag{1}$$

where S is the set of nodes to output to disk and T is the set of tags of the nodes output to disk. $Size(tag)$ is a constant. The total storage size is estimated in Equation 2.

$$\begin{aligned} Size_{storage} &= log|T| + log|S| + |T| \cdot size_{entry\_of\_tag\_index} + \\ &+ |S| \cdot size_{entry\_of\_node\_index} + \sum_{node \in S} size(node) \end{aligned} \tag{2}$$

where $|S|$ and $|T|$ represent the size of S and T respectively. The first two items of Equation 2 are the storage cost of number of tags and number of nodes respectively. The second item is the size of tag index and the third item is the size of node index. The last item is the storage cost of nodes.

3.2 Query Processing Based on the Index

Our query processing algorithms for the F&B index with partial nodes on the disk is an extension of the common algorithms for in-memory F&B index. In the following, we briefly discuss the algorithms for the two types of queries: */-queries* and *//-queries.*

Algorithm 1. QueryPath(n, q, $type$, fid)

```
 1: for all twig in q.twigs do
 2:    for all child in n.children do
 3:       if twig.tag == child.tag then
 4:          if QueryPath(child,twig, EXISTING, n.id) == $\emptyset$ then
 5:             return $\emptyset$
 6: if q.nextLocStep != NULL then
 7:    for each child in n.children do
 8:       if child.tag == q.nextLocStep.tag then
 9:          result = result $\cup$ QueryPath(child,q.next,type,n.id)
10:          if type == EXISTING $\wedge$ result != $\emptyset$ then
11:             return result
12:    return result
13: else
14:    if n is a stub then
15:       return LoadExtent(n.tag,fid)
16:    else
17:       return n.extent
```

A /-query is a XPath query with only parent-child axis. Traversal-based method is commonly used for in-memory F&B index for such kind of queries. We present the sketch of the enhanced traversal-based method is in Algorithm 1, which is capable of working with F&B index with partial nodes on the disk. For each step of XPath query, at first, twig conditions are checked via recursive calls (Lines 1–5). The algorithm proceeds to process the next step in the query only after all the branching conditions have been satisfied. When encountering a leaf F&B index node, we shall take different steps depending on whether it is a in-memory index node or an in-memory stub. In the former case, the extent of the node will be appended to the result set; in the latter case, the extent should be loaded from disk However, we do no need to access the extent on the disk when checking twig conditions, because the tags of nodes on the disk are maintained in the corresponding stubs.

For //-queries, we need to traverse the whole subtrees of the current index node in order to evaluate the ancestor-descendant-or-self axis. We note that when the twig condition contains //, we can immediately return once a qualified node is met. In the interest of space, we skip the algorithm here.

A common procedure in the algorithms for both types of queries is the Load-Extent algorithm (shown in Algorithm 2), which loads the extents of index node on the disk. It has three steps and assumes that the tag index has been loaded into the main memory. Before query processing, tag index is loaded in main memory. At first, the entry of *tag* in node index is found. Then binary search is

Algorithm 2. LoadExtent(tag, fid)

```
1: $(begin\_index, end\_index) = searchIndex(tag)$;
2: $(begin\_pos, end\_pos) = searchPosition(fid, begin\_index, end\_index)$;
3: return LoadData(begin_pos, end_pos);
```

used to find the position of extent of requested node in node index. This binary search has two levels. The first level is to apply binary search on block level, that is to say, to find the block containing requested node by comparing with the first record and last record in disk. The second level is find the exact position in selected block. The usage of binary search is for the reason that it can locate to the disk containing required entry as soon as possible.

Cost Estimation

We now provide estimation of the disk I/O cost for processing a query on our data structure. With the assumption that buffer is "large enough" (its definition will be provided shortly), Equation 3 gives the upper bound of the physical I/O (PIO) and Equation 4 gives the upper bound of logical I/O (LIO) of processing a query Q.

$$PIO(Q) = \lceil \frac{\sum_{p \in Disk(tag(Q))} size(p)}{BLK} \rceil + \lceil \frac{num(tag_Q) * size_{node\_entry}}{BLK} \rceil \quad (3)$$

$$LIO(Q) = \sum_{p \in disk(Q)} size(p) + \lceil \log \lceil \frac{num(tag_Q) * size_{node\_entry}}{BLK} \rceil \rceil \quad (4)$$

where $Disk(tag(Q))$is the set of F&B index nodes on the disk with the request of Q; $disk(Q)$ is the set of F&B index nodes required by Q on the disk; $size(p)$ is the size of F&B index node p; BLK is block size; $num(tag_Q)$ is the number of index nodes on the disk with the same tag as the goal of query Q; $size_{node_entry}$ is the size of each entry in node index.

In Equation 3, accessing each nodes on the disk satisfying Q requires us to locate the position of the node on the disk ($\lceil \log \lceil \frac{num(tag_Q)}{BLK} \rceil \rceil$ times of disk read) and to access the node ($\lceil \frac{size(p)}{BLK} \rceil$ times of disk read). When estimating the number of physical disk I/Os, because of the existence of buffer, many of disk pages can be located in the buffer during the query processing. So with enough buffer pages, the number of disk I/Os is *smaller* than the total number of disk pages containing the entries in the node index with tag specified by Q, $\lceil \frac{num(tag_Q)}{BLK} \rceil$. Since the nodes satisfying query Q may not be continuous on the disk, accessing all the nodes as Q's query result needs at most the number of all disk pages containing any node as the result of Q.

The condition that the buffer is "large enough" is the number of buffer pages should be no less than the number in Equation 5. The intuition is that the buffer should be able to accommodate the largest of the node index for a specific tag plus the number of blocks used during access the extent (this number is 1 since extent of each node is accessed only once).

$$\max_{T \in S}\{\lceil \log(\lceil \frac{num(tag_Q) * size_{node\_entry}}{BLK} \rceil \rceil) + 1\} \quad (5)$$

where S is the family of sets. Each is a set of nodes in disk with the same tag.

4 Node Selection Strategies

In this section, we focus on the choice of the subset of leaf F&B index nodes to be shifted to the disk. We consider two scenarios: without workload information and with workload information.

4.1 The General Problem

It is unnecessary to shift all the leaf nodes to the disk if there are enough memory. Therefore, we consider optimal choices of selecting the subset of leaf index nodes to be shifted to the disk. We will start with the scenario where we have no workload information. In this scenario, the choice of index nodes depends on the size of the nodes and their potential cost of disk I/O if they are shifted to the disk. We discuss in the following the formulation of this optimization problem.

The cost model of physical I/O can be represented as

$$Cost_{physicalIO} = \sum_{T \in S} (f_{node} \cdot \lceil \frac{\sum_{a_i \in T} size(a_i)}{B} \rceil + \lceil \frac{num\_tag(T) \cdot size\_head}{B} \rceil) \quad (6)$$

where S it the set family, each element in which is a set of node with the same tag; a_i is node in T;f_{node} is the frequency of accessing to *node*, $size(node)$ is the size of *node* stored in disk and $num\_tag(node)$ is the number of nodes in disk with the same tag with *node*.

It is noted that Equation 6 represents the upper bound of physical I/O. The first item is the total storage size of nodes in the same tag. The second one is the max number of disk I/O necessary to find all the entries of extents of nodes in disk in the same query. Because binary search is used to find related node. The number of real accessed blocks during query is often smaller than $\lceil \frac{num\_tag(T) \cdot size\_head}{B} \rceil$.

The logical I/O cost is as following.

$$Cost_{logicalIO} = \sum_{node \in S} (f_{node} \cdot \lceil \frac{size(node)}{B} \rceil + \log \lceil \frac{num\_tag(node) \cdot size\_head}{B} \rceil + H)$$

where B is block size, S is the set of leaves selected to output to disk. Other symbols have the same meaning as Equation 6.

The first item of the formula is the number of I/O to read the extent of the index and the second item is the number of I/O to find the position of the position of a node with binary searching.

The problem can be represented as

$$\begin{aligned} &\text{minimize} && T = \begin{pmatrix} Cost_{physicalIO} \\ Cost_{logicalIO} \end{pmatrix} \\ &\text{Subject to } \textstyle\sum_{i \notin S} cost_{memory}(node_i) + size_{node} \cdot |T| \leq M \end{aligned} \tag{7}$$

This is a general definition of this problem. Without workload information, f_{node}'s are determined with the assumption that the frequency of all queries are same. So all f_{node}'s are set to equal value.

Obviously, this problem is hard to solve. But it is found in practise, the value of physical I/O, which determines the efficiency, in the cost model only changes in a small range. Therefore, without workload information, random selection is practical for the selection of nodes to output to disk. This point will be shown in the experiment.

4.2 The Solution Based on Workload

In this subsection, we consider the scenario where we have the query workload information. Intuitively, we can find better node selection strategies in this case.

Since the problem with workload information is still a hard one, we aim at finding good heuristic solutions. Our proposed heuristic has three levels.

The first level is based on weight in workload. This method is to add a weight property to each leaf node in the index. The queries are executed in the full F&B index. When a leaf node n is met, the weight of this query in workload is added to weight property of n. Then, all leaves are sorted based on weight. The nodes with the least weight number is outputted out disk. This method is reasonable. It is because the nodes accessed most should be contained in main memory.

If many nodes have same weight, in our heuristic method, the nodes with the largest extent should be outputted to disk. Intuitively such method can remain more nodes in main memory.

The third level is to output the nodes that have more nodes with same tag as them. From the Equation 6, because of the existence of ceiling function, adding a new tag to outside set brings more cost.

5 Experiments

In this section, we present part of the our extensive experiment results on the proposed improved F&B index.

5.1 Experimental Setup

All of our experiments were performed on a PC with Pentium 1GMHz, 256M memory and 30G IDE hard disk. The OS is Windows 2000 Professional. We implemented our system using Microsoft Visual C++ 6.0. We also implemented the F&B index building algorithm using random node selection or the heuristic

Table 2. Basic Information of XML Document for Experiment

Document Nodes	#Tags	#F&B Index Nodes	#Leaves in F&B Index	#leave ratio
413111	77	141084	83064	58.88 %

node selection algorithm based on workload, as well as the query processing algorithms. All the algorithms return the extents of the index nodes as the result. We used CLOCK algorithm as the buffer replacement policy.

The dataset we tested is the standard XMark benchmark dataset. We used a scale factor of 0.2 and generated XML document of size 20M. We used the following metrics in our experiments: query processing time, number of physical I/Os (PIO) and number of logical I/Os (LIO). Some statistics of the dataset and its F&B index is shown in Table 2.

In order to test the performance of the experiment, we generated various query sets as workloads. First, we generate all the single query with only parent-child edge (the set is called *single query set* for brief) and random select some ratio of them as the workload. Second, we extracted path queries from the standard test query set of XMark to test the general efficiency of our method.

5.2 Performance Results

Performance Without Workload. We test the performance without workload with random selection of nodes to output to disk. The parameters we used is available memory 0.5M, page size 1024 bytes, buffer size 128 pages. We select 10% queries from uniform query workload randomly. We did five times of experiments with same setting and selected nodes randomly. The result is in Table 3. From the result, the standard deviation of runtime, PIO and LIO are 28.028, 0.10 and 5.58 respectively. It can be seen from these data that without workload information, there is only little difference in efficiency with various random nodes selections.

Performance of the Method with Workload. We fixed the buffer size 64 blocks, page size 1024 bytes and memory size of leaves 1M bytes. We used random selection from all single paths with only single slash and the path queries extracted from standard of XMark query. All the experiment runs in cold buffer. The efficiency is compared with the instance of random selection nodes to output to disk. The results of random selection and optimized by workload are shown

Table 3. Experiment Results of Random Node Selection

nodes in disk	#average run time	#average PIO	#average LIO
70492	356.14	21.03	962.34
69916	307.31	20.94	949.86
70045	323.06	21.03	960.97
70233	309.34	20.86	966.74
70336	378.74	20.77	961.03

Table 4. Comparison of Different Node Selection Strategies

method	#nodes in disk	#avg. run time	#avg. PIO	#avg. LIO
Heuristic Algorithm	65565	289.06	0	0
Random selection	69937	521.97	20.57	935.54

in Table 4, where avg. PIO and avg. LIO represent average PIO and LIO of running all queries in workload on the index stored in these two node selection strategies, respectively.

5.3 Changing System Parameters

In this subsection, we test our system by varying various system parameters.

Varying the Size of Workload. We test the performance by varying the size of workload. We select the workload randomly from uniform query workload and various the number of queries in workload if from the 10% to 100% of uniform query workload. We fixed main memory for leaves 0.5M, page size 1024 bytes and buffer size 128 pages. The performance results are shown in Figure 3(a) and Figure 3(b).

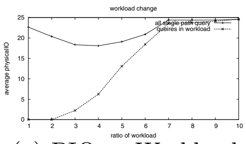
It can be observed that when the ratio of workload increases, PIO and LIO increases. This is because when the number of nodes related to queries in workload increase to more than the memory's capacity, the average efficiency of processing the queries in workload will decrease. While the average efficiencies of processing the queries in the whole singe query set is stable.

Varying Available Memory Size. We test the performance with various available memory size. We fix the number of queries in workload 10% of the uniform workload, page size 1024 bytes and buffer size 128 pages. The results are shown in Figure 3(c) and Figure 3(d).

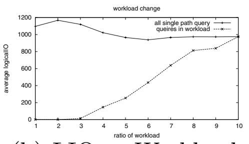
From Figure 3(c), we can observe that when memory size becomes large enough, the number of disk I/O changes to 0. It represents the instance that main memory is large enough to hold all the results nodes in the queries set. Since our storage is optimized for workload, so size of memory that makes the disk I/O of processing queries in workload 0 is smaller than that of processing queries in the whole singe-path query path.

Varying Cache Size. We test the performance with various buffer sizes. We fix the number of queries in workload 30% of the uniform query workload, page size 1024 bytes and available memory size 0.5M. The fixed parameters assure the number of I/O of querying workload queries non-zero. Since the number of logical does not chance with the varying of buffer size. We only test the change of PIO. The result is in Figure 3(e).

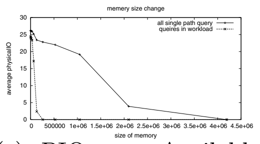
It can be observed from Figure 3(e), when the size of buffer reaches to a number, the number of PIO does not change. It is because all the entities of node index are cached, as we discussed in Section 3.1.



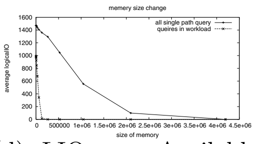
(a) PIO vs Workload



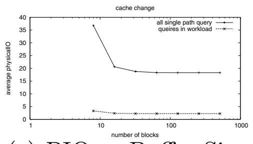
(b) LIO vs Workload



(c) PIO vs Available Memory



(d) LIO vs Available Memory



(e) PIO vs Buffer Size

Fig. 3. Experiment Results

6 Related Work

There have been many previous work on indexing the values [7], structure and codes of XML data [4]. We focus on structural indexes in this paper. Most of the structural indexes are based on the idea of considering XML document as a graph and partitioning the nodes into equivalent classes based on certain equivalence relationship. The resultant index can support (or cover) certain class of queries. A rigid discussion of the partitioning method and its supported query classes is presented in [10]. DataGuide [2] are based on *equivalent* relationship and 1-index [8] are based on the *backward bi-similarity* relationship. Both of them can answer all simple path queries. F&BIndex [5] uses both backward and forward bisimulation and has been proved as the minimum index that supports all branching queries (i.e., twig queries). These "exact" indexes usually have large amount of nodes and hence size, therefore, a number of work has been devoted to find their *approximate* but smaller counterparts. A(k)-index [6] is an approximation of 1-index by using only k-bisimilarity instead of bisimilarity. D(k)-index [9] generalizes A(k)-index by using different k according to the workload. M(k)-index and M*(k)-index [3] further optimize the D(k)-index by taking care not to over-refining index nodes under the given workload. Updating structural indexes has just receive attentions from researchers. Another different approach to index the XML data is to build indexes only for frequently used paths in the workload. APEX [1] is such an example and can be dynamically maintained.

7 Conclusion and Further Work

F&B index is the smallest structure index to answer all the branch queries of XML. But its size prevents its widely use. In this paper, a novel method of making F&B index practical is presented. The basic idea is to put some leaves of F&B index to disk. We design the storage structure and query processing strategy. For reduce disk I/O during query processing, we also present nodes to disk selection strategy. Without workload, we use random selection method. And with workload, we design heuristic method to select optimized nodes. Extensive experimental results proves the efficiency of our system. Our future include design an efficient method to output internal nodes to disk to support very large F&B index.

References

1. C.-W. Chung, J.-K. Min, and K. Shim. Apex: an adaptive path index for XML data. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD 2002)*, pages 121–132, 2002.
2. R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of 23rd International Conference on Very Large Data Bases (VLDB 1997)*, pages 436–445, 1997.
3. H. He and J. Yang. Multiresolution indexing of XML for frequent queries. In *Proceedings of the 20th International Conference on Data Engineering (ICDE 2004)*, pages 683–694, Boston, MA, USA, March 2004.
4. H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML data for efficient structural join. In *Proceedings of the 19th International Conference on Data Engineering (ICDE 2003)*, pages 253–263, 2003.
5. R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD 2002)*, pages 133–144, 2002.
6. R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for efficient indexing of paths in graph structured data. In *Proceedings of the 18th International Conference on Data Engineering (ICDE 2002)*, pages 129–140, San Jose, CA, USA, March 2002.
7. J. McHugh and J. Widom. Query optimization for xml. In *Proceedings of 25th International Conference on Very Large Data Bases (VLDB 1999)*, pages 315–326, 1999.
8. T. Milo and D. Suciu. Index structures for path expressions. In *Proceedings of the 7th International Conference on Database Theory (ICDE 1999)*, pages 277–295, 1999.
9. C. Qun, A. Lim, and K. W. Ong. D(k)-Index: An adaptive structural summary for graph-structured data. In *The 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD 2003)*, pages 134–144, San Diego, California, USA, June 2003.
10. P. Ramanan. Covering indexes for XML queries: Bisimulation - Simulation = Negation. In *Proceedings of 29th International Conference on Very Large Data Bases (VLDB 2003)*, pages 165–176, 2003.
11. A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB 2002)*, pages 974–985, 2002.
12. W3C. XML Query 1.0 and XPath 2.0 data model, 2003. Available from `http://www.w3.org/TR/xpath-datamodel/`.