Efficient Processing of XML Path Queries Using the Disk-based F&B Index *

Wei Wang [1] Hongzhi Wang [1,3] Hongjun Lu [2]
Haifeng Jiang [4] Xuemin Lin [1] Jianzhong Li [3]

[1] University of New South Wales, Australia, {weiw,lxue}@cse.unsw.edu.au
[2] Hong Kong University of Science and Technology, China, luhj@cs.ust.hk
[3] Harbin Institute of Technology, China, {wangzh,lijzh}@hit.edu.cn
[4] IBM Almaden Research Center, USA, jianghf@us.ibm.com

Abstract

With the proliferation of XML data and applications on the Internet, efficient XML query processing techniques are in great demand. Answering queries using XML indexes is a natural approach. A number of XML indexes have been proposed in the literature; among them, F&B Index is one powerful index as it is the smallest index that answers all twig queries. However, an F&B Index suffers from the following two problems: (1) it was originally proposed as a memory-based index while its size is usually large in practice and (2) answering queries using an F&B Index is not fully optimized. These problems limit the benefits and even applications of F&B Indexes in practice.

In this paper, we propose a highly optimized disk organization method for an F&B Index; the result is a disk-based F&B Index with good clustering properties. In addition, novel query processing algorithms exploiting the physical organization of the disk-based F&B Indexes are proposed. Experimental results verify that our disk-based F&B Index can scale up for large data size with good query performance compared with state-of-the-art XML query processing algorithms.

* This work was partially supported by UNSW FPG Grant (PS06863), UNSW Goldstar Grant (PS07248), and ARC Discovery Grant (DP0346004).

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

1 Introduction

XML has become the *de facto* standard for information representation and exchange over the Internet. An XML document contains hierarchically nested elements. Therefore, it can be naturally modeled as a tree, where elements are modeled as nodes in the tree and direct nesting relationships between elements are modeled as edges between nodes [22]. Standard XML query languages, e.g., XPath and XQuery, can retrieve a subset of the XML data nodes satisfying certain path constraints. For example, the XPath query $//book[appendix]//figure$ will retrieve all figure nodes that appear under books that have appendix sections.

With the proliferation of XML data and applications on the Internet, efficient XML query processing techniques are in great demand. Processing path expression queries efficiently still remains a great challenge. Current approaches can be roughly classified into two categories. One is to process the query on-the-fly, using the newly proposed "join" operators, such as structural join [25] and twig join [4]. These approaches have received great attentions and have been implemented in several systems [7]. The other is to index the data and answer queries by probing the index only, e.g., DataGuide [5], 1-index [18] and F&B Index [12]. Among them, F&B Index is one of the most powerful structural indexes as it was shown to be the smallest index that answers all twig queries [12]. However, an F&B Index suffers from the following two problems: (1) *lack of scalability*: F&B Index was proposed as a memory-based index only; however, its size is usually large in practice. To the best of our knowledge, there is no proposed solution for the case when an F&B Index cannot be accommodated in memory, and (2) *lack of efficiency*: answering queries using an F&B Index is complicated and not fully optimized. Even for the simple query that only contains /, searching in an F&B Index is *non-deterministic* in nature, meaning that multiple branches need to be searched. For

harder queries that involve //, many subtrees need to be thoroughly traversed. This can be costly even if the whole index is in memory, let alone the case when the index is on the disk.

These problems severely limit the benefits and even applications of F&B Indexes in practice. As a consequence, many recent proposals try to build *approximate* indexes [15, 20, 8], which usually have a smaller size than their accurate counterparts; however, this only partially *alleviates* the index size and query processing efficiency problems mentioned above.

In this paper, we first address the size problem of the F&B Index by proposing a disk-based F&B Index. We identify that a crucial factor that affects the query performance is the clustering scheme in the storage layer. Therefore, we propose a clustering method that has several salient properties which enable efficient and novel query processing algorithms. In addition, we further optimize our disk-based F&B Index by integrating the state-of-the-art coding schemes widely used in the join-based XML query processing approaches. We then address the efficiency problem by devising a set of novel query processing algorithms. They are based on tree traversals, disk scan and *segment-based join*, all of which are efficiently supported by our disk-based F&B Index. In particular, range-based query processing and segment-based join algorithms are proposed which greatly enhance the efficiency of processing queries with // by avoiding unnecessary tree traversals.

Our contributions in the paper can be summarized as follows:

- We solve the lack-of-scalability problem of the F&B Index by proposing a highly optimized disk-based F&B Index with good clustering properties.
- A set of novel query processing algorithms based on the disk-based F&B Index are proposed. These algorithms solve the lack-of-efficiency problem.
- Our extensive experimental results demonstrate that our proposed disk-based F&B Index can scale up to large data sizes with excellent query performance.

The rest of the paper is organized as follows. Section 2 introduces some background knowledge. Section 3 presents the basic version of our disk-based F&B Index and Section 4 discusses tree traversal-based and range-based algorithms to process various kinds of path queries efficiently. An advanced version of our disk-based F&B Index is introduced in Section 5, together with a set of efficient query processing algorithms. We present our experimental results and analyses in Section 6. Related work is described in Section 7 and Section 8 concludes the paper.

2 Preliminaries

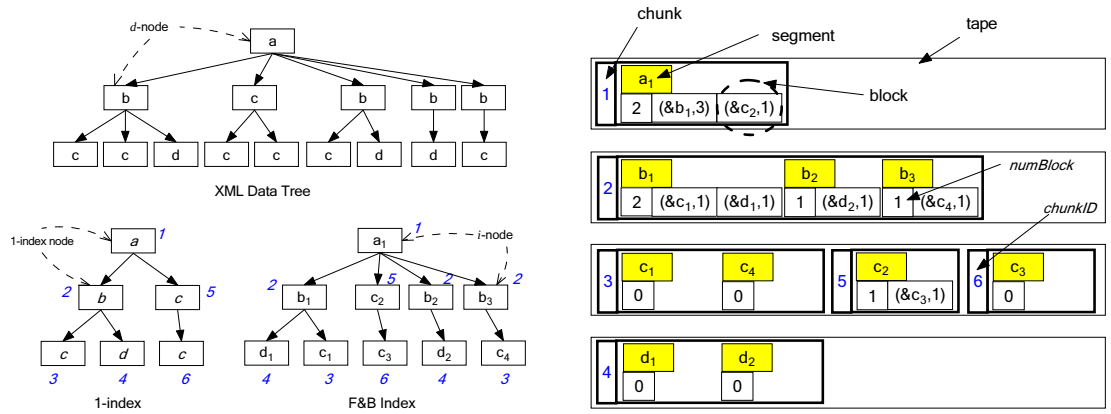
XML data is usually modeled as labeled trees: elements and attributes are mapped to nodes in the trees and direct nesting relationships are mapped to edges in the trees. In this paper, we only focus on element nodes; it is easy to generalize our methods to the other types of nodes defined in [22].

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 [5] and 1-index [18] can cover all simple path queries, that is, path queries without branches. [12] showed that F&B Index is the minimum index that covers all branching path queries. Note that if the XML data is modeled as a tree, its 1-index and F&B Index will also be a tree. Each index node n can be associated with its extent, which is the set of data nodes in the data tree that belong to the index node n.

We show an example XML data tree, its 1-index, and its F&B Index in Figure 1(a). In the 1-index, all the second level b elements in the data tree are classified as the same tag b index node; this is because all those nodes cannot be distinguished by their incoming path, which is a/b. However, those bs are classified into three groups in the F&B Index; 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&B Index has many more nodes (10 in our example). It can be shown that in the worst case, an F&B Index has the same number of nodes as the data tree does.

Notations

We describe some notations used in the rest of the paper. We distinguish the nodes in an original data tree and the nodes in its 1-index (or F&B Index); the former are termed *d-nodes* and the latter are termed *1-index nodes* (or *i*-nodes, respectively). We assume that there exists a total order among all tags. Then we can label each 1-index node using its `min-pre-order` traversal number, denoted as its *1-index node number*. A `min-pre-order` traversal is a `pre-order` traversal that always chooses to descend into the subtree of the unvisited child node that has the smallest tag. For example, assuming a dictionary order for the tags, a `min-pre-order` traversal on the example 1-index in Figure 1(a) will first visit the b node (and its subtree) under the root a node, before visiting the c node. Since the F&B Index is a refinement of the 1-index, each i-node n in the F&B Index corresponds to a unique 1-index node and thus is assigned the 1-index node number (denoted as $n.1indexID$). We term the `child` axis as **PC** axis and term `descendant-or-self` axis as **AD** axis. Sometimes, we refer to a path without branching predicates or AD axis as a *simple path*.



(a) An example XML data tree, its 1-index and its F&B Index. 1$indexID$ numbers are marked as italic numbers near each i-node.

(b) The disk-based F&B Index, consists of tapes, chunks and segments.

Figure 1: Organization of the Disk-based F&B Index

3 The Basic Index Structure

In this section, we describe the basic structure of our disk-based F&B Index, as well as its construction algorithm based on a novel clustering scheme. Its advanced version will be introduced in Section 5.

It is easy to store a tree naïvely onto the disk: one can simply replace each in-memory pointers with disk pointers. However, such a scheme is not likely to work well because disk access is page-based and its cost is much higher than the cost of random memory access. Clustering is thus crucial to the performance of our data structure. Our basic idea is to analyze index access patterns and store data that is frequently accessed together close on the disk too. To that end, we employ the following three clustering criteria:

C1 We cluster all i-nodes with the same tag name together on a logical unit called *tape*. The intuition is that the answer to most path queries consists of i-nodes with the same tag name.

C2 The child i-nodes of an i-node are clustered as follows: we group the child i-nodes from the same parent by their tag names. This is based on the observation that tree traversal-based query processing on the F&B Index is based on the *designated child access* operator, i.e., to access all the child i-nodes having the designated tag name together. For example, to evaluate query $./a/b$, we need to access *all* the i-nodes of tag a for each of the context node, even if some tag a i-node might not have a tag b i-node as its child.

C3 We further cluster nodes with the same tag according to their 1$indexID$. This is because the answers to simple path queries are exactly those i-nodes having the same 1$indexID$. It also enables an advanced query processing method to be discussed in Section 4.2.

As a result, the F&B Index shown in Figure 1(a) can be stored and clustered on the disk as shown in Figure 1(b). We describe several important organization concepts in the resulting disk-based F&B Index as follows:

Tape A tape holds all i-nodes with the same tag name, according to clustering criterion **C1**. For example, as there are four different tags in the example F&B Index, four tapes are created in its disk-based version.

Segment and block Each i-node in the F&B Index corresponds to a unique segment. As a result of clustering criterion **C2**, we are able to record in each segment a *compressed* representation of its child segment pointers: we keep a *block* for each group of child segments with the same tag, and a field *numBlock* as the total number of such blocks. A block consists of a disk pointer to the first child segment in the group, followed by the size of the group. For example, in Figure 1(b), i-node a_1 in Figure 1(a) corresponds to the segment a_1 in Figure 1(b); i-node a_1 has four child i-nodes: b_1, c_1, b_2, b_3. They form two blocks: the block for the three tag b child i-nodes, represented as $(\&b_1, 3)$ and the block for the single tag c child i-node, represented as $(\&c_2, 1)$. $a_1.numBlock$ is therefore set to 2. As a special case, since leaf i-nodes do not have any child i-nodes, their *numBlock* fields are set to 0 (e.g., see segment c_1).

Chunk All segments with the same 1$indexID$ are stored in a *chunk*, according to our clustering criterion **C3**. Chunks on the same tape are sorted by their *chunkID*s in an ascending order. For example, although segments c_1 and c_4 are from different parent i-nodes, they are put in the same chunk (with $chunkID = 3$ on the tape for tag c segments), because they have the same 1$indexID$.

It can be easily shown that our disk-based F&B Index correctly preserves all the structural information

of the F&B Index. Its space is linear to the number of i-nodes in the F&B Index.

The three clustering criteria help create a disk-based F&B Index with minimum page accesses for many types of queries. For example, it is easy to verify that the disk-based F&B Index in our example has the minimum number of disk accesses for *all* simple path queries. On the other hand, if, for example, we store c segments in `pre-order` on the same tape, the access cost for query $a/b/c$ is likely to be higher because the result, segments c_1 and c_4, is likely to be on different disk pages.

Building the Index

Algorithm 1 BuildDiskF&BIndex($\mathcal{T}$)

1: Build 1-index of $\mathcal{T}$; assign `min-pre-order` traversal number to each 1-index node.
2: Build the F&B Index according to [12]. {It is easy to keep track of the $1indexID$ for each F&B Index node as it was split from some 1-index node.}
3: Calculate the disk address of the corresponding segment of each i-node by sorting on $(tag, 1indexID, parentID)$.
4: Write all the i-nodes as segments on the tapes according to their appropriate disk address; also fix the disk pointers in blocks.

The algorithm to build our disk-based F&B Index for a given XML data tree $\mathcal{T}$ is given in Algorithm 1. It can be shown that we can simply sort all segments according to $(tag, 1indexID, parentID)$ to obtain a clustering scheme that satisfies all of our three clustering criteria. Here, $parentID$ is just a unique identifier assigned to each i-node in the F&B Index. The building cost is the total cost of building the F&B Index, sorting all index nodes and writing all the nodes to the disk. Assuming the F&B Index can be constructed in memory, the building process will incur $O(n \log n + m \log m)$ CPU cost and $O(m)$ I/O cost, where n is the number of d-nodes in the XML document and m is the number of i-nodes in the resulting F&B Index, assuming the height of the XML data tree is a constant.

4 Query Processing Algorithms

In this section, we describe the query processing algorithms on the basic version of the proposed disk-based F&B Index. We discuss fundamental algorithms that are based on tree traversals as well as a novel algorithm based on disk scan. We note that both approaches are well supported by the physical clustering scheme employed in the index.

4.1 Query Processing Using Tree Traversals

In this subsection, we consider processing all types of path queries using only the *designated child access*, i.e., the physical tree traversal-based operator that accesses all the child segments (of the current segment) with a designated tag.

There are two different logical operators we can use based on designated child accesses: we can either (1) sequentially retrieve only *one* child segment within the block or (2) retrieve *all* the child segments within the block. We term the former operator FetchNextChild(s, t) and the latter FetchAllChildren(s, t), where s is the parent segment and t is a tag name.

Using different logical operators results in different traversal strategies in the query processing. To illustrate, let us consider the simplest case where the query is a path query with PC axes only. For example, consider the query $/x/y$. In general, during the traversal, there will be multiple x_i segments satisfying $/x$ and multiple y_i^j segments satisfying $/x/y$ within each x_i's subtree. It is well-known that we have two classical traversal strategies: *DFS* (depth-first search) and *BFS* (breadth-first search). DFS will access one matching x_i and immediately descend into its subtrees and try to match y_i^js. BFS will access all matching x_is together first, then for each matching x_i, continue matching all their child y_i^js together. It is obvious that DFS and BFS can be easily implemented using FetchNextChild() and FetchAllChildren() operators, respectively.

BFS (or DFS) is always applicable for other types of queries as well. If a query has branching predicates with PC axes only, e.g., $/x[z]/y$, we only need to recursively evaluate all the branching predicates, e.g., $./z$, before descending into the subtrees. A subtle difference is that we only need to know if the result of the branching path query is empty or not, instead of requiring all the results. Therefore, we introduce an additional parameter, *mode*, to the query processing algorithm. When the incoming parameter *mode* is set to `EXISTING`, the query processing algorithm will immediately return upon finding the *first* result.

If the query contains AD axes, e.g., $x//y$, we *have to* traverse the *whole* subtrees under each of the tag x segment to retrieve all the segments with tag y, again using either BFS or DFS traversal strategy. Similarly, if the AD axis appears within a branching predicate, as $.//y$ does in $x[.//y]/z$, we can immediately return as soon as the *first* answer is found. This is indicated by the *mode* parameter too.

The complete pseudo-code for the BFS and DFS traversal-based query processing algorithms are not hard but long and complicated. In the interest of space, we omit them and only give an illustrating example. In the rest of the paper, we assume BFS and DFS query processing has been encapsulated in the QueryPath(n, p, *method*, *mode*) function, where n is root segment to be queried, p is a query, *method* is either `BFS` or `DFS` and *mode* is either `ALL` or `EXISTING`.

Example 1 (DFS, PC axis, Branching Predicate)

Let us consider the example F&B Index in Figure 1(a) and the query /a[b/d]//c. We extract the first axis step /a[/b/d] from the query. If DFS is used, we will invoke FetchNextChild *to get one match first, i.e., segment a_1; then we test all the branching predicates against the current match. In this case, we need to recursively evaluate another path query ./b/d for the current segment a_1.* FetchNextChild *will be repeatedly invoked to find the first matching ./b segment, which is b_1, and then the first matching ./d segment from b_1, which is d_1. Note that since the evaluation of this query is for branching test only, we can immediately return* `TRUE` *to the calling routine now that one answer, i.e., d_1, is found. Since a_1 passes all the branching predicate tests, we recursively evaluate the rest of the query, i.e., .//c, immediately for a_1 and output its result. After all the matching /a nodes are processed, the algorithm will stop.*

4.2 Range-based Query Processing

Although the performance of the tree traversal-based query processing has been greatly improved over previous systems, it still suffers from the problem of traversing a lot of unnecessary intermediate segments when processing queries involving AD axes. In this subsection, we introduce a novel range-based query processing method that can directly return all results without unnecessary traversals for certain types of queries.

Let us consider queries in the form of $p//x$, where p is a simple path without AD axes and x is a tag. Our basic idea is that: *if all the descendant x segments under p are clustered within a range and if we can find the starting and ending positions of the range*, we can *avoid* a lot of tree traversals by just fetching all segments within that range.

The following lemma not only assures us of the feasibility of this range fetching idea, but also gives constructive ways to find such ranges.

Lemma 1 [Range Property] *Given a query in the form of $p//x$, x segments in the answer will be laid out contiguously within a range on the tape for tag x in the disk-based F&B Index. More specifically, let all segments satisfying p belongs to chunk r. Let there be k 1-index nodes with tag x in the subtree rooted at r in the corresponding 1-index, denoted as x_i ($1 \le i \le k$) and having 1-index number u_i. All the segments belonging to chunk $i \in [min, max]$ on the tape for x is exactly the answer to the query, where $min = \min_i\{u_i\}$ and $max = \max_i\{u_i\}$.*

Example 2 *Consider the example F&B Index in Figure 1(a) and the query a//c. a matches with segment a_1, whose chunkID is 1. In the 1-index shown in Figure 1(a), there are 3 1-index nodes tagged c within the subtree of 1-index node with number 1. They correspond to chunks 3, 5 and 6, respectively. Therefore, we know that $min = 3$ and $max = 6$. It is easy to verify that, say, any tagged c segment that is outside the subtree rooted at a_1* cannot *reside in a chunk between [3, 6] on the c tape.*

According to Lemma 1, we need to find out the range $[min, max]$ for a given query. A naïve way to do this is to (1) keep a copy of the 1-index and (2) pinpoint the 1-index node r that satisfies path p and traverse its subtree to find such min and max. We can do better by observing that (1) the 1-index is actually *embedded* in our data structure (represented as chunks) and (2) such min and max can be precomputed for every valid combination of (n, x), in order to save traversals among chunks in the runtime. This observation results in a lookup table H in the form of $(chunkID, tag, minChunkID, maxChunkID)$, where $chunkID$ corresponds to the 1-index node number, tag is a tag that appears in the subtree rooted at the $chunkID$ 1-index node, $minChunkID$ ($maxChunkID$) is equal to min (max), respectively. The size of the lookup table H is $O(n_{1index} \cdot T)$, where n_{1index} is the number of nodes in a 1-index and T is the number of tags. Algorithm 2 describes the resulting RangeFetch algorithm, which is based on tree traversal, table lookup and disk scan.

Algorithm 2 RangeFetch($p//t$)

```
1: ret = ∅
2: s = QueryPath(root, p, DFS, EXISTING)
3: if s ≠ NULL then
4:    if (s.chunkID, t) is in the lookup table H then
5:       (min, max) = H[(s.chunkID, t)]
6:       ret = fetchChunk(min, max)
7: return ret
```

Example 3 *Consider running RangeFetch to process the same query a//c against the example F&B Index. Evaluating path a using* `DFS` *and* `EXISTING` *options will quickly return the first matching segment a_1. We then lookup the table H for $chunkID = 1$ and $tag = c$, which will return $min = 3$ and $max = 6$. After obtaining the min and max values, we can fetch all the c segments within that range using efficient (sequential) disk scans.*

We note that queries in the form of p/t are special cases of $p//t$ and can be easily handled by generalizing the RangeFetch() algorithm. On the other hand, if p is not a simple path, e.g., it contains branching predicates, RangeFetch algorithm is not applicable. This kind of complex queries (involving branching predicates and/or AD axes) can still be efficiently processed using an advanced query processing algorithm, SegSJ, to be discussed in Section 5.

5 Advanced Disk-based F&B Index

In this section, we discuss some further optimizations on the data structure and algorithms. They enable new query processing techniques for several types of queries. More specifically, our basic idea is: given the huge success of join-based query processing techniques based on coding schemes [25, 6, 23, 4, 23, 10], *how can we integrate the similar coding idea into our data structures and take advantage of it*? Our answer is affirmative; our solutions can be summarized as (1) attach codes to each d-node in the extent of each i-node in the F&B Index and (2) devise efficient query processing algorithms given the availability of the codes and other new data structures.

5.1 Introducing New Data Structures

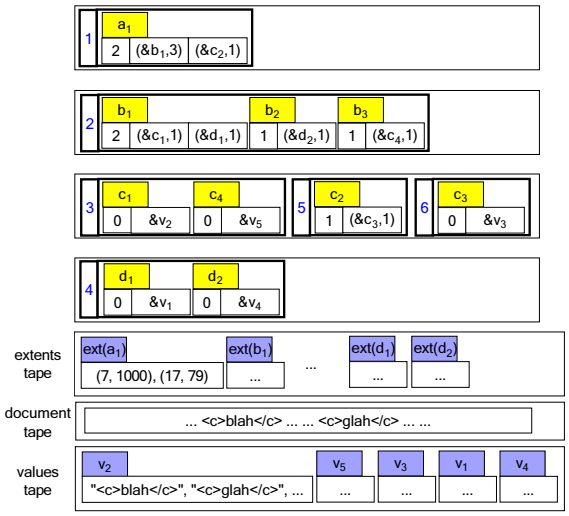


Figure 2: Adding Region Codes, Source XML Document and Values

Figure 2 shows the resulting data structure after adding *region codes*, *source XML document* and *values* into our basic data structures. They are stored on three new tapes shown at the bottom of the figure. The first new tape is the *extents tape*, for storing the extents of all segments in the F&B Index; the rest of the tapes will be introduced soon in Section 5.3. Recall that an extent is a set of d-nodes that belong to a segment. Each d-node can be uniquely identified by its region code (we use the popular $(start, end)$ codes [25]). For example, in Figure 2, we can see segment a_1's extent has the two d-nodes. Their region codes are (7, 1000) and (17, 79), respectively.

We enforce the following important constraints on the region codes:

1. The region code of an extent is exactly its physical offsets of the start and end positions. In other words, given the region code of a d-node, we can retrieve its textual representation, which also includes all its descendant nodes, if any. This property is used in Section 5.3.
2. The d-nodes in the extent of a segment are sorted according to their $(start, end)$ values. This gives us a nice property to efficiently check the ancestor-descendant relationship for any two *segments*, due to Lemma 2.

Lemma 2 [Structural Relationships Between Segments] *Given any two segments R and S with their extents sorted (in the same order), let the first d-nodes in their extents be p and q, respectively. Then segments R and S are of ancestor-descendant relationship in the F&B Index if and only if p and q have the ancestor-descendant relationship in the XML data tree, which can be checked by testing if $(p.start < q.start < p.end)$ holds.*

In terms of implementation, two changes are made to the original data structures[1]:

- A pointer is created in each segment that points to its extent on the extents tape.
- We embed the region code of the first d-node in each segment's extent into the segment itself. This is an optimization for the SegSJ algorithm (to be discussed shortly), which only needs to access such nodes. For the example F&B Index shown in Figure 2, the first region code, $(7, 1000)$, will be physically stored together with segment a_1.

5.2 Processing Queries Using Joins

One of the most important benefits of having extents encoded in region coding scheme is that all queries with //-axis can be processed in a very efficient way, similar to that of structural join. Let us look at an example. In our basic data structures, the query $/a/b[d]//c$ can only be processed using tree traversal-based method (RangeFetch algorithm is not applicable, because of the branching predicate $[d]$), which might be inefficient. With our advanced data structures, we can process the query as follows:

1. Choose a two-phase plan as follows: (1) Compute $R = /a/b[d]$ and (2) The original query can be computed as $R//c$.
2. Use tree traversal-based method to evaluate the first phase: $/a/b[d]$. Our disk-based F&B Index can evaluate such twig queries efficiently.
3. Use Algorithm SegSJ(R, $//c$) to compute the final result. The algorithm is shown in Algorithm 3. The algorithm constructs the region code lists from ancestor segments and descendant segments on-the-fly. After that, a state-of-the-art structural join algorithm is applied. The correctness of the algorithm follows from Lemma 2.

[1] They are not shown in Figure 2, in order not to clutter the figure.

Algorithm 3 SegSJ(A, D)

1: Construct $R(start, end)$ by iterating through the extents of all segments in A, choose the first d-node and append its $(start, end)$ value pair to R.
2: Construct $S(start)$ by iterating through the extents of all segments in D, choose the first d-node and append its $start$ value to S.
3: Apply a state-of-the-art structural join algorithm over R and S.

Astute reader might want to find out the difference between using this SegSJ algorithm and using a structural algorithm on the original data. We note that:

- The original structural join algorithm cannot take advantage of the structural index and hence require multiple joins. Even the most recent improvement in [14] can only take advantage of 1-index; for our example twig query, it requires another join (and union) to resolve the branching predicate $[d]$, because 1-index cannot *cover* branching queries.
- For each segment, only *one* d-node in the original document needs to participate in the join. This greatly reduces the number of elements that need to be joined in both sets.
- We note that R and S are not necessarily sorted according to $(start, end)$. There are two categories of structural join algorithms to use: the partition-based algorithms [23] and the sort-merge-based algorithms [3]. This is an optimization problem and the algorithm with the lowest estimated cost should be chosen.

5.3 Processing Subtree or Value Retrieval Queries

In this subsection, we explain the rationale and usage of the latter two tapes in Figure 2.

The second of the new three tapes is the *document tape*. It is used to store the original XML document. Because we enforce the region code of any d-node d to be the physical offsets of its start and end positions in the original XML document, we can retrieve the whole (data) subtree rooted at the extent (inclusive) by scanning the document tape from $d.start$ to $d.end$. This is a much more efficient way of reconstructing the whole subtree for non-leaf d-nodes in the data tree, because otherwise, the naïve approach of traversing all the descendant nodes, retrieving their contents and combining the results is extremely costly!

The last tape is the *values tape*. We store values for all extents of all leaf segments in the F&B Index on the values tape. This also entails adding pointers to the leaf segments to their values on the values tape. Our query processing algorithm will take advantage of the values tape by always fetching values directly from it. This strategy results in fewer I/Os when a query needs to access the contents of the leaf segments, as otherwise, we need to access extents and then the document tape. Results from our experiments show that the space overhead due to the values tape is acceptable: less than 30% of the size of the XMark document.

6 Experimental Evaluation

In this section, we present results and analyses of part of our extensive experiments on the disk-based F&B Index.

6.1 Experimental Setup

All of our experiments were performed on a PC with Pentium3 1 GHz CPU, 256 MB memory and 30 GB IDE hard disk. The OS is Windows 2000. We implemented our system using Microsoft Visual C++ 6.0. We implemented our **DFS**, **BFS**, **RangeFetch** and **SegSJ** algorithms. All algorithms return the complete query results instead of d-node identifiers unless otherwise stated. We used the LRU buffer replacement policy. In order to compare with other systems fairly, all the experiments were run in the warm buffer. We set the page size to 4096 bytes.

We selected **NoK**, **TwigStack** and **Kaushik *et al.*'s** query processing algorithms for comparison. We obtained the source code of NoK system [26] and TwigStack [4] from the original authors. NoK is one of the latest native XML storage and query processing systems while TwigStack is the best twig join algorithm to process all twig queries *without* index. We also implemented Kaushik *et al.*'s algorithm, which utilizes structural index to accelerate structural join processing; our SegSJ algorithm can be viewed as using structural join to accelerate index probing and thus makes the comparison with Kaushik *et al.*'s algorithm interesting.

The datasets we tested include XMark [21], DBLP [1], and TreeBank [2]. In the interest of space, we report results on the standard 113M XMark benchmark dataset in most of the following experiments. XMark features a moderately complicated and fairly irregular schema, with several deeply recursive tags. DBLP (highly regular) and Treebank (highly irregular) are included to test the two extremes of the spectrum in terms of the structural complexity. Some statistics of the datasets, their 1-index and their F&B Index are shown in Table 1. We used a relatively small buffer size (less than 1% of the data size) to better simulate the case when the XML data size grows up to Gigabytes. We used the following metrics: elapsed time (time), number of physical I/Os (PIO) and number of logical I/Os (LIO).

Table 1: Size of the Disk-based F&B Index

Dataset	Doc. Nodes	Tags	1-index Nodes	F&B Index Nodes
XMark	2048193	77	550	528076
DBLP	4620206	42	157	4112
TreeBank	131072	103	37906	122402

In order to better test and understand the characteristics of our disk-based F&B Index under different workloads, we designed a set of queries that has different characteristics. There are three major dimensions: query type, axis type and result size. Query type includes either *simple path* query (i.e., without branches) or *twig* query. Axis type includes *without AD axis* (i.e., PC axis only) or *with AD axis*. Result size includes *small* number of results or *large* number of results. The combination of these three dimensions generates eight categories. For example, twig query with AD axis and have a small result size is abbreviated as *AD-twig/small*. We select one representative query for each category for each dataset; we number those queries from 1 to 8, together with QX, QD, and QT prefixes (for **X**Mark, **D**BLP, and **T**reeBank, respectively). We show queries for the XMark dataset in Table 2. Applicable algorithms on different categories of queries are listed in Table 3.

Table 3: Applicable Algorithms on Queries

Queries	Type	Applicable Algorithms
Q?1, Q?2	PC-path	DFS, BFS, RangeFetch
Q?3, Q?4	PC-twig	DFS, BFS
Q?5, Q?6	AD-path	DFS, BFS, RangeFetch, SegSJ
Q?7, Q?8	AD-twig	DFS, BFS, SegSJ

6.2 Index Size

The total size of our disk-based F&B Index (including the optional values tape) is 177.15 MB, with 17.2 MB for all the tag tapes, 15.6 MB for the extents tape, 113 MB for the document tape, and 31.3 MB for the values tape. The space overhead of the pure index part (i.e., tag tapes, the extents tape and the value tapes) is about 57% of the document size (with about 28% due to values tape). The indexing overhead for DBLP and TreeBank are 86% and 158%, respectively. We note that such overhead is acceptable when compared to the results reported in some recent studies [14, 26, 19].

6.3 Varying System Parameters

In this subsection, we study the performance of our own system by varying various system parameters.

6.3.1 Varying Page Size

We tested the performance of our algorithms by varying the page size. The result is shown in Figures 4(a) to 4(d) (on Page 12). Note that both axes are in logarithmic scale. We can observe that the number of physical I/Os decreases exponentially with the increase of page size (with a reasonably large buffer pool). This is because (1) more data can be read in by one I/O and (2) a large page size increases the effectiveness of our "clustered" index. For example, it is more likely that all child segments with a same tag name will be able to be stored within a single page.

6.3.2 Varying Buffer Size

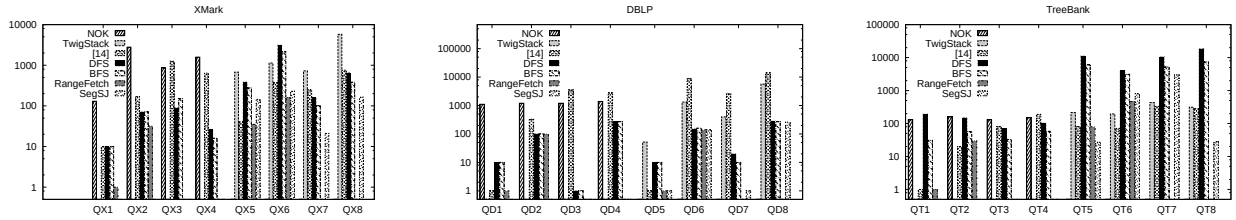
In this experiment, we varied the number of buffer pages and measure the performances of the algorithms. The results are shown in Figures 4(e) to 4(l). We varied the number of buffer pages from 4 to 356, thus it corresponds to buffer size from 16K to 1424K. Note that the Y-axis is in logarithmic scale.

We can make several observations from the figures:

- We can observe that for all the queries, increasing the number of buffer pages reduces the number of physical I/Os; on the other hand, the number of logical I/Os remains the same. This is because the increase in the amount of buffer pages can effectively lower the page faults in the buffer.
- Let us consider the effect of the amount of buffer pages to each algorithm. It can be observed that DFS and BFS are very sensitive to buffer sizes, esp. the DFS algorithm. This is because they are traversal-based algorithms and possibly many intermediate i-nodes in the F&B Index need to be accessed. A large buffer size can effectively buffer the segments of those i-nodes and thus reduce the page faults. DFS is most "hungry" for buffers because our clustering method clusters sibling child segments in the same page; but DFS does not take advantage of this property but keeps descending into the subtrees of one child segment, which essentially brings new pages into the buffer. This also explains why BFS usually performs better than DFS in almost all query categories. On the other hand, RangeFetch and SegSJ are less sensitive to the buffer size, esp. the RangeFetch algorithm. This is because scanning segments, which only needs one buffer page, is the major cause of disk I/Os in both algorithms.
- Let us consider the effect of buffer sizes to each query category.
  - For the *PC-path* category (QX1, QX2), RangeFetch performs the best followed by BFS and then DFS. This is because RangeFetch can quickly locate the chunk which contains exactly all the query answers and output the whole chunk. Nevertheless, as the query only requires descending into the index, with a few buffer pages, the other traversal-based algorithm also performs well.
  - For the *PC-twig* category (QX3, QX4), DFS performs no worse than BFS. This is because, esp. in Q3 where the twig predicates are nested deeply, DFS can quickly determine if the twig predicate is satisfied or not. Similar to the last category, with a few more buffer pages, both algorithms achieve stable performance.
  - For the *AD-path* category (QX5, QX6), the best algorithm is RangeFetch, followed by SegSJ, BFS and DFS. RangeFetch is the fastest as it directly fetches the result without further compu-

Table 2: Statistics of Queries for XMark

QID	Type	Result	Query
QX1	PC-path/small	245	/site/regions/africa/item/description/parlist/listitem/text/keyword
QX2	PC-path/large	59486	/site/open_auctions/open_auction/bidder/date
QX3	PC-twig/small	267	/site/closed_auctions/closed_auction[annotation/description[parlist/listitem/text[keyword[bold]]]]/price
QX4	PC-twig/large	12679	/site/people[person[profile[education]/age]]/person/phone
QX5	AD-path/small	12206	/site/closed_auctions//emph
QX6	AD-path/large	138236	/site//person
QX7	AD-twig/small	3274	/site/people/person[//age]//education
QX8	AD-twig/large	29250	/site/closed_auctions/closed_auction[//description]//person



(a) XMark (b) DBLP (c) TreeBank

Figure 3: Comparing With Other Systems on XMark, DBLP, and TreeBank Datasets

tation or segment access. It is interesting to see that traversal-based approaches, BFS and DFS, can have a comparable performance with SegSJ, given enough buffers.

– For the *AD-twig* category (QX7, QX8), SegSJ works the best, followed by two traversal-based algorithms. The two traversal algorithms have only minor differences when the buffer is really small. Compared to the last category, BFS requires more buffers to arrive at a stable performance. This is because at the branching point, more than one tag needs to be searched in the subtree; this in general requires more segments (and hence pages) to be accessed. The most interesting finding might be in Figure 4(l): traversal-based approaches even outperform SegSJ when buffer is large. This is because the number of *//person* is very large but only a fraction of them appear in the QX8's result. Therefore, SegSJ accessed many unnecessary pages. On the other hand, with a large buffer size, traversal-based approach can buffer a larger part of the subtrees and gain some performance advantages.

It seems that the difference among different algorithms is more obvious when the result size of the query is small. This is because when the result size is large, many disk I/Os *have to* be made by any algorithm and this reduces the percentage of I/O differences made in the index probing phase.

6.3.3 Buffer Hit Ratio

We also collected buffer hit ratios for accessing the index under different buffer sizes, to partially verify the effectiveness of our clustering method. We only show the results for the 113M XMark dataset for QX6 (which is the most expensive query in our query set) in Figure 4(m). It can be observed that the hit ratio increases gradually for both traversal-based algorithms with the increase of buffer size, but BFS is much more buffer-friendly than DFS. RangeFetch almost always results in disk scans and thus its hit ratio is almost 0.0%. SegSJ only needs to access the index to construct its join inputs and do the join in the memory; therefore, its hit-ratio quickly reaches a high point beyond a small number of buffer pages.

6.3.4 Scalability

We performed experiments on different sized XMark datasets, with a fixed relative percentage of buffer size, in order to simulate and test the scalability of our disk-based F&B Index and its algorithms. We show the result for QX7 in Figure 4(n). From the figure, we can observe that all algorithms scale linearly with the increase of the size of the dataset. Since we are using a relatively very small buffer size, we expect that our system can scale well for even Gigabytes of XML data.

6.4 Comparing with Other Systems

In this subsection, we compare the performance of our system with the other three systems: NoK, TwigStack and Kaushik *et al.*'s algorithm. The first two systems do not return values, so we modified our algorithm not to fetch values either. Because NoK implicitly uses several megabytes of memory for buffering and TwigStack does not need buffering, we set buffer size to 1 MB for our system and Kaushik *et al.*'s algorithm. We included the final sorting time in our algorithms such that the results are in the document order. We measured the elapsed time of all the applicable algorithms and plotted them in Figure 3.

We will first focus on the representative XMark dataset (i.e., results shown in Figure 3(a)), and then discuss the impact of datasets with different structural

complexity to the relative performance of the algorithms.

NoK on XMark Dataset

Currently, the NoK implementation we received from the original authors can only support PC queries. So we only list its results for QX1 to QX4 in Figure 3(a). We can see that our system outperforms NoK by a large extent for all queries (10 to 100 times faster). We conjecture that clustering in our system plays an important role in outperforming NoK. For example, we found that NoK spends a large number of physical I/Os and thus time for QX2, whose result size is large.

TwigStack on XMark Dataset

We ran TwigStack for QX5 to QX8 and plotted the result in Figure 3(a). We find that even our traversal-based methods work better than TwigStack in all queries but QX6. As discussed above, for AD-twig queries (QX7, QX8), which are more complicated than other categories, traversal-based methods worked especially well. QX6 (`site//person`) is the best case for TwigStack and it worked very well; nonetheless, our SegSJ and RangeFetch still outperform TwigStack substantially. We note again that SegSJ is very efficient (up to 40 times faster than TwigStack) mainly because it only needs to join *less* data: only one d-node (or element) among all d-nodes belonging to the same segment needs to be accessed and joined.

Kaushik *et al.*'s Algorithm on XMark Dataset

We ran Kaushik *et al.*'s algorithm [14] for QX1 to QX8 and showed the results in Figure 3(a). We find that it works pretty well in general, and outperforms TwigStack in all cases, which agrees with the experimental results in [14]. However, we found that it is still slower than the tree traversal-based algorithms (i.e., BFS and DFS) for all PC-path queries and all twig queries, while it works better than the tree traversal-based algorithms for AD-path queries. This is mainly due to the fact that when [14] processes a twig query, it decomposes a twig query into a set of "relaxed" components (by adding `//` before each component) before probing the structural index and doing the join. As a result, [14] has the worst performance for QX3, which has a deeply nested twig query structure. We note that our other two algorithms, RangeFetch and SegSJ, outperform [14] for almost all the queries.

Results on Other Datasets

The performances of all the algorithms on DBLP and TreeBank datasets are shown in Figures 3(b) and 3(c). It is obvious that all of our algorithms have significant performance advantages over other systems on the DBLP dataset. This is because the structure of the DBLP data is fairly simple and regular and our indexing and clustering technique works best under such situations. For the extremely complicated and irregular TreeBank data, the performance of our algorithms degrades. However, except for QT6 and QT7, some of our algorithms are still the fastest. The degradation in performance is due to the fact that the F&B Index for TreeBank data is almost useless: each data node is likely to be a distinct F&B Index node (See Table 1).

7 Related Work

There has been much previous work on indexing the values [17], structure and codes of XML data [9], and fragments of XML in a relational DBMS [19]. 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, such as DataGuide [5] and 1-index [18], and F&B Index [12]. Some work has also been devoted to find the *approximate* but smaller counterparts of the above indexes, including A(k)-index [15], D(k)-index [20], M(k)-index and M*(k)-index [8]. Several work focused on updating structural indexes [13, 20, 24].

XML queries can also be evaluated on-the-fly using the join-based approaches. Structural join and twig join are such operators and their efficient evaluation algorithms have been extensively studied [25, 6, 4, 23, 10]. Their basic tool is the coding schemes that enable efficient checking of structural relationship of any two nodes. TwigStack [4] is the best twig join algorithm to answer all twig queries without indexes; hence it is chosen for comparison in our experiment.

To the best our knowledge, there has not been much work reported in storing tree-shaped XML data on the disk in its native form. Lore's storage manager stores objects into variable length slots in a page using the first-fit algorithm; objects are clustered according to the depth-first traversal order [16]. [11] is a native XML storage manager that stores the XML data tree onto disk pages. Their basic clustering method is to cluster appropriate sized subtrees into one disk page. They also allow fine-tuning the clustering via a clustering matrix. NoK [26] is the latest native XML storage and query processing system. It features a succinct XML storage scheme that essentially stores a string representation of the XML data tree obtained from pre-order traversal. These schemes cannot be directly applied to our case, as the access pattern on the index is different from that on the XML data tree.

Most recently, the idea of integrating both tree traversal-based and join-based query processing has been proposed [7, 14]. The major difference of their systems with our proposal is that they either do not use a structural index [7] or they only use the index as a filter to accelerate structural joins [14]. On the contrary, our focus is more on using the structural in-

dex to its full power and integrating join-based query processing method into the system to accelerate index lookup. We feel the two approaches are both orthogonal and complementary to each other.

8 Conclusions

XML data is rich in structure; this calls for advanced indexing techniques in order to facilitate efficient query processing. In this paper, we address the lack-of-scalability and lack-of-efficiency issues associated with the in-memory F&B Index. We propose a disk-based F&B Index that features several good clustering properties. Those properties not only enable efficient tree traversal, but also give rise to several novel query processing methods based on disk scan and segment-based join. We also integrate coding schemes into our index to better support processing several types of queries. Extensive experiments have demonstrated that our proposed disk-based F&B Index has several salient features, including cache-friendliness behavior and good scalability. It also significantly outperforms alternative tree-based or join-based query processing systems.

We note that our experimental results showed that tree traversal-based approaches can sometimes even outperform optimized join-based approaches (including our SegSJ algorithm) for certain sophisticated queries. This result further extends a recent similar result on the effectiveness of tree traversal-based query processing [7] and suggests that tree traversal might be a complementary query processing technique to the join-based ones.

Acknowledgment

The authors would like to thank Ning Zhang, Nicolas Bruno and Alan Halverson for their help in carrying out the experiments. Hongzhi Wang was supported by an ARC research grant of Xuemin Lin and John Shepherd.

References

[1] DBLP XML records. Available at `http://dblp.uni-trier.de/xml/`.

[2] TreeBank. Available at `http://www.cs.washington.edu/research/xmldatasets/data/treebank`.

[3] S. Al-Khalifa, H. V. Jagadish, J. M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE 2002*.

[4] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *SIGMOD 2002*.

[5] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB 1997*.

[6] T. Grust. Accelerating XPath location steps. In *SIGMOD 2002*.

[7] A. Halverson, J. Burger, L. Galanis, A. Kini, R. Krishnamurthy, A. N. Rao, F. Tian, S. Viglas, Y. Wang, J. F. Naughton, and D. J. DeWitt. Mixed mode XML query processing. In *VLDB 2003*.

[8] H. He and J. Yang. Multiresolution indexing of XML for frequent queries. In *ICDE 2004*.

[9] H. Jiang, H. Lu, W. Wang, and B. C. Ooi. XR-Tree: Indexing XML data for efficient structural join. In *ICDE 2003*.

[10] H. Jiang, W. Wang, H. Lu, and J. X. Yu. Holistic twig joins on indexed XML documents. In *VLDB 2003*.

[11] C.-C. Kanne and G. Moerkotte. Efficient storage of XML data. In *ICDE 2000*.

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

[13] R. Kaushik, P. Bohannon, J. F. Naughton, and P. Shenoy. Updates for structure indexes. In *VLDB 2002*.

[14] R. Kaushik, R. Krishnamurthy, J. F. Naughton, and R. Ramakrishnan. On the integration of structure indexes and inverted lists. In *SIGMOD 2004*.

[15] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for efficient indexing of paths in graph structured data. In *ICDE 2002*.

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

[17] J. McHugh and J. Widom. Query optimization for XML. In *VLDB 1999*.

[18] T. Milo and D. Suciu. Index structures for path expressions. In *ICDE 1999*.

[19] S. Pal, I. Cseri, G. Schaller, O. Seeliger, L. Giakoumakis, and V. V. Zolotov. Indexing XML data stored in a relational database. In *VLDB 2004*.

[20] C. Qun, A. Lim, and K. W. Ong. D(k)-Index: An adaptive structural summary for graph-structured data. In *SIGMOD 2003*.

[21] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *VLDB 2002*.

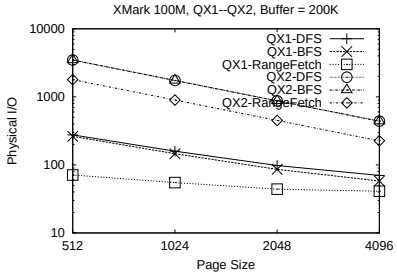
[22] W3C. XML Query 1.0 and XPath 2.0 data model, 2003. Available from `http://www.w3.org/TR/xpath-datamodel/`.

[23] W. Wang, H. Jiang, H. Lu, and J. X. Yu. PBiTree coding and efficient processing of containment joins. In *ICDE 2003*.

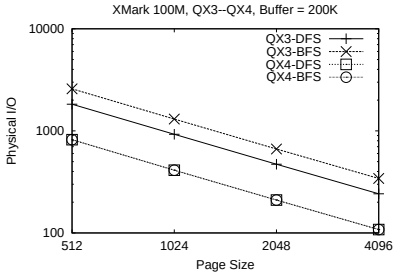
[24] K. Yi, H. He, I. Stanoi, and J. Yang. Incremental maintenance of {XML} structural indexes. In *SIGMOD 2004*.

[25] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD 2001*.

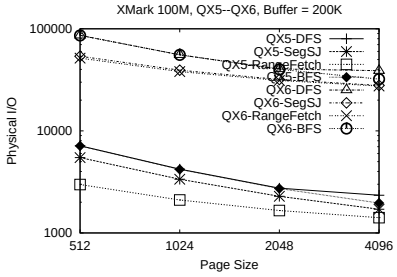
[26] N. Zhang, V. Kacholia, and M. T. Özsu. A succinct physical storage scheme for efficient evaluation of path queries in XML. In *ICDE 2004*.



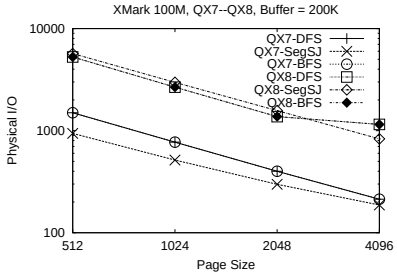
(a) PIO vs PageSize: QX1 and QX2



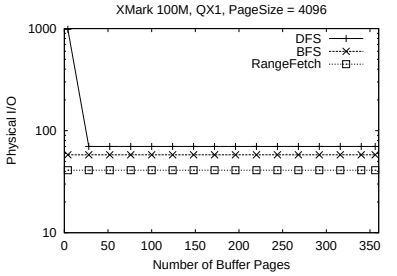
(b) PIO vs PageSize: QX3 and QX4



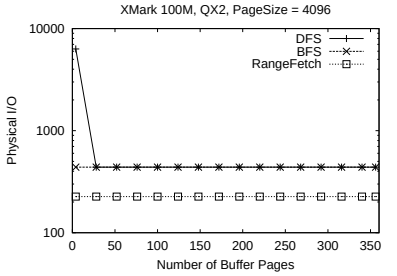
(c) PIO vs PageSize: QX5 and QX6



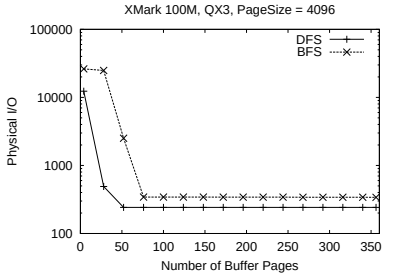
(d) PIO vs PageSize: QX7 and QX8



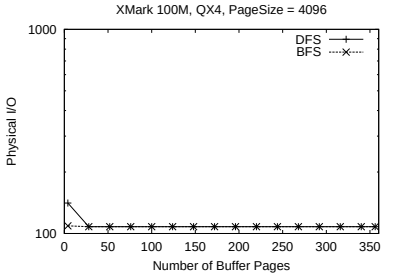
(e) PIO vs Buffer Size: QX1



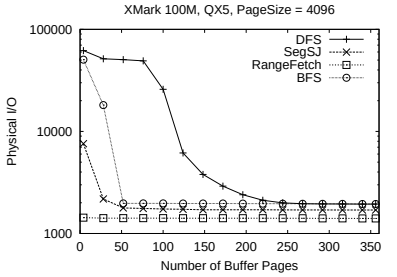
(f) PIO vs Buffer Size: QX2



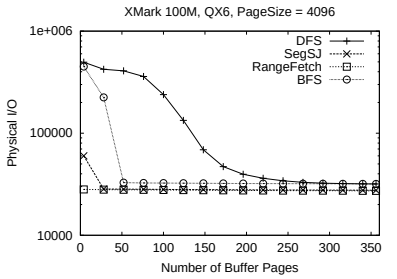
(g) PIO vs Buffer Size: QX3



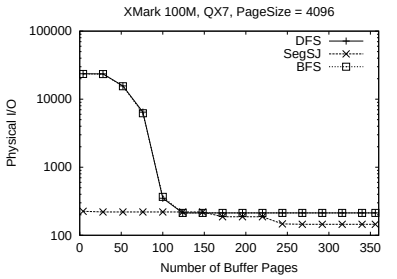
(h) PIO vs Buffer Size: QX4



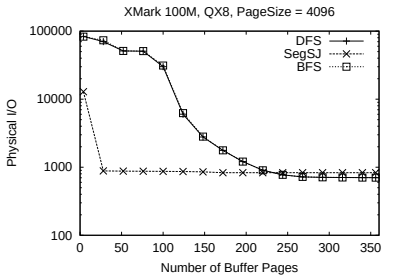
(i) PIO vs Buffer Size: QX5



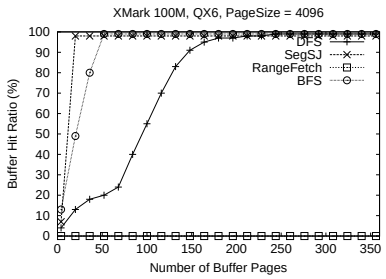
(j) PIO vs Buffer Size: QX6



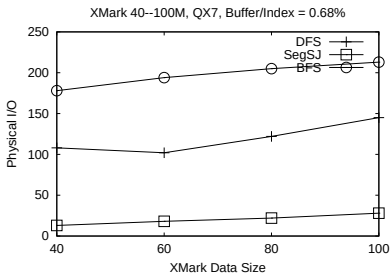
(k) PIO vs Buffer Size: QX7



(l) PIO vs Buffer Size: QX8



(m) Buffer Hit Ratio: QX6



(n) Scalability: QX7, PageSize=4096

Figure 4: Experimental Results