AC-Index: An Efficient Adaptive Index for Branching XML Queries

Bo Zhang[1] Wei Wang[2] Xiaoling Wang[1] Aoying Zhou[1]

[1] Fudan University, China
{zhangbo, wxling, ayzhou}@fudan.edu.cn

[2] University of New South Wales, Australia
weiw@cse.unsw.edu.au

Technical Report
UNSW-CSE-TR-0701
January 2007

THE UNIVERSITY OF
NEW SOUTH WALES

School of Computer Science and Engineering
The University of New South Wales
Sydney 2052, Australia

Abstract

Query-adaptive XML indexing has been proposed and shown to be an efficient way to accelerate XML query processing, because it dynamically adapts to the workload. However, existing adaptive index suffers from a number of issues, such as lack of support for general types of XML queries, and unsatisfactory query and update performances. In this paper, we propose a new query-adaptive index named AC-Index. It is designed to supports XML path queries with branching predicates. We propose efficient index construction, query processing, and index adaptation algorithms for the AC-Index, together with a number of optimizations to further boost the performance of the index. Our experimental results demonstrate that the AC-Index significantly outperformed previous approaches in terms of query processing and adaptation efficiencies.

1 Introduction

XML has become the standard for data representation and exchange on the Internet. The rapid popularity of XML repositories requires systems that can store and query XML data efficiently. Among other XML query languages, XPath [2] is an expressive query language and has been extensively adopted. The semantics of XPath queries is based on a flexible tree traversal model. For example, an XPath query $//museum[phone]$ will find all the museums with phone number information at any level in the XML document tree. Indexing the structure of XML data is thus an effective way to accelerate query processing, because it can greatly reduce the search space and also be an integral part of other XML operations. Researchers have proposed various kinds of XML indexes [7, 13, 10, 9, 14, 8, 4, 16, 6] to facilitate the query processing. Among them, adaptive indexes [4, 14, 8, 6] are well-known for their high performances as they can adapt their structures to suit the query workload.

However, most of the proposed adaptive indexes can only accommodate a rather limited class of XPath queries efficiently. Little attention has been given to the problem of building an adaptive index for the more general and complicated queries. For example, APEX [4] were designed to only support *suffix path queries* of the form $//l_1/l_2/\ldots/l_k$ (where l_i is a tag name) efficiently, and cannot tackle branching queries. Rewriting is needed to handle path queries with more than one occurrence of the //-axis. It does not support queries with branching predicates either.

Another issue of the existing adaptive indexes is the query processing and adaptation efficiency. For example, query rewriting and multiway structural join have to be used for queries containing a descendant axis in the middle of the query expression. The adaptation method for APEX does not make full use of the current index, and has to traverse the *entire* XML data tree for each update. The adaptation method essentially process each node in the XML data graph individually, thus failing to take advantage of the facts that a group of "similar" nodes[1] can be processed in batches. Some other adaptive indexes, e.g., A(k)-index and D(k)-index, are intrinsically biased towards more specific queries; they may also incur frequent global updates if the query workload changes significantly.

In this paper, we propose the AC-index designed for general types of XPath queries. The basic idea is to index and manipulate a group of "similar" nodes together, which, in our case, is designed to be F&B index nodes. We propose efficient algorithms to constructing the index, update the index to accommodate frequent queries, and using the index to answer query. Our experiment results shows that the proposed index significantly outperforms previous approaches in both query and adaptation efficiencies.

Our contributions of the paper can be summarized as follows:

- We propose an adaptive index that supports arbitrary path queries with branching predicates. We show that all the issues regarding monitoring the frequent queries, checking query containment, etc, are more challenging than, for example, simple path queries without descendant axis or branching predicates.

[1] Intuitively, these are nodes that are indistinguishable for the class of queries the index supports.

- We propose an efficient query processing and index update algorithms for the index. Our methods make full use of the existing index for efficient index update. In addition, the granularity of our update (or validation) is a group of nodes instead of individual nodes. These design choices significantly improves the query and update performance of the proposed index.

- We have conducted extensive experiments to evaluate the performance of the proposed AC-Index with previous approaches. previous indexes.

The remaining part of this paper is organized as follows. Section 2 introduces the background knowledge and some related work. In Section 3 we present the data model and related concepts. Section 4 introduces the data structures of the AC-Index as well as the construction, update and query processing algorithm for the index. Several optimizations are discussed in Section 5. Experimental results are reported and analyzed in Section 6 and Section 7 concludes the paper.

2 Related Work

In the interest of space, we briefly overviews most relevant work in the literature.

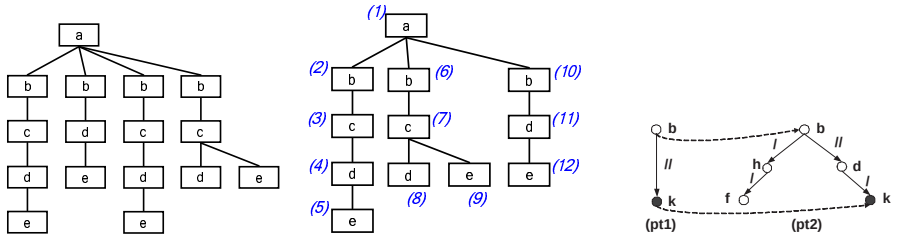
Many indexes for XML data have been proposed to support efficient query processing. They can be categorized into workload-insensitive indexes [7, 13, 10, 9, 14, 16] and workload-adaptive indexes [5, 4, 8, 6]. The focus of [8, 6] is mainly to avoid over-refinement and thus reduces the size of graph bisimulation-based indexes [9]. Index fabric [5] mentioned the idea to customize the index for frequent queries, but without elaborating on it. APEX [4] is a complete proposal to a fully adaptive index optimized for frequent queries in the workload. However, APEX is only designed for single-path queries; in addition, its query adaptation procedure is extremely costly.

Another closely related area is materialized views and semantic caching for XML queries [3, 1, 11]. The idea is to find a rewriting of the queries using one of the materialized views for efficient query processing. However, they rely on the backend XML database to further process the rewritten queries to extract results from the materialized views.

2.1 XML Caching and Views

Our AC-Index can be seen as a well-organized cache of frequent materialized views. Much work has been done to provide XML cache/view for XML databases [1, 11]. Among them, [1] proposes a frame using materialized XPath views to answer queries. They come up with a set of rules for finding containment matching between query and views, and matching algorithm is provided. Because each matching process still includes navigation process, this work is inefficient with a large number of views to compare with. [11] forcusing on the problem of efficient view selection. They use string-based checking method which enable filtering out much unconcerned views, by which effcient cache lookup is achieved. Our work is similiar with [11] in that we also use some heuristic method to filter out unconcerned queries as much as possible, and our filtering method make full use of the containment characteristic compared with [11].

3 Preliminary



(a) An Example XML Data Tree

(b) Its F&B Index ((*x*)) is the pre-order traversal number of a node

(c) A Homomorphism Example

Figure 3.1: A Running Example

An XML document is usually modeled as an unordered rooted node-labeled tree, named *XML Data Tree*. An example of an XML data tree is showed in Figure 3.1(a). An F&B index is the smallest index that can answer all the branching queries [9]. The F&B index for the example XML data tree is shown in 3.1(b). We assign an ID to each F&B index node based on a pre-order traversal.

A path query containing a combination of the fragments $\{/, //, []\}$ can be expressed as a labeled pattern tree. For example, we show the pattern tree for $//b[h/f]/d//k$ as `pt2` in Figure 3.1(c). In both the query processing and adaptation phases of the AC-Index, we need to perform containment test between XML path queries. Consider two path queries, p_1 and p_2, p_1 *contains* p_2, if there exists an *homomorphism* from the pattern tree of p_1 to that of p_2 [12]. For instance, consider the pattern tree `pt1` and `pt2` shown in Figure 3.1(c) for path queries $//b//k$ and $//b[h/f]//d/k$. Figure 3.1(c) shows a homomorphism from `pt1` to `pt2`, in which all query nodes in `pt1` (b and k) are mapped to the corresponding nodes with the same labels in `pt2`, and all the paths in `pt1` subsume those in `pt2` (e.g., $b//k$ subsumes $b//d/k$). For the class of XPath queries we consider in the paper, polynomial algorithm exists to test the containment relationship between two XPath queries [17].

4 The AC-Index

In this section, we give an overview of the system architecture and introduce the data structures, initialization, query processing, and adaptation methods for the proposed AC-Index.

4.1 System Architecture

The architecture of the AC-Index system is shown in Figure 4.1. There are four main components: (a) the *initialization module* constructs the initial AC-Index. (b) The *query processing module* accepts the query pattern trees and execute the queries on top of the AC-Index. (c) The *query statistics module* maintains important system statistics, e.g., (approximate) frequencies of queries. (d) The

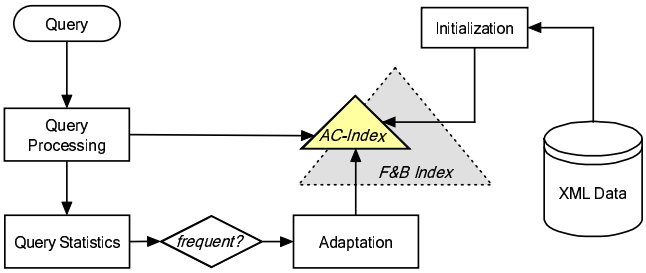


Figure 4.1: System Architecture

adaptation module adapts the AC-Index to the current workload to improve query performance.

Our AC-Index can also be viewed as a "secondary index" that sits on top of an F&B index (the shadowed triangle in the figure). The *goal* of the AC-Index is exactly to create a *small* and *adaptive* index that answers a sizeable portion of the query workload efficiently.

4.2 Data Structures

An F&B index for the XML data is a fine partition of nodes according to their incoming and outgoing paths, such that it can answer all the branching queries. However, an F&B index is query-independent and could be *over-refined* and thus *sub-optimal* for a given query workload. For example, answering the query $//e$ using the example F&B index in Figure 3.1(b) requires a traversal of the *complete* F&B index index.

The basic idea of our AC-Index is to group F&B index nodes according to the frequent queries in a given workload, such that the query results can be efficiently retrieved across a few nodes in the AC-Index. A straight-forward solution is to keep a list of frequent queries, together with their corresponding lists of F&B index nodes in their results. However, this solution suffers from the following problems:

- *Low space utilization*: it is not uncommon that two different queries will have overlapping query results. If the same elements are stored multiple times in the index, significant amount of space would be wasted.
- *Slow response time*: sequential search has to be used when answering queries, and this solution obviously does not scale well with the number of frequent queries in the workload.
- *Unable to answer non-identical queries*: only identical queries can be answered in this simple solution.

Our AC-Index is designed to address all the above issues by (a) organizing the frequent queries in a non-redundant way leveraging the containment hierarchy among queries; (b) designing efficient lookup methods to locate relevant queries; and (c) being able to answer a larger class of queries efficiently by novel query processing techniques.

More specifically, the AC-Index consists of three parts: an F&B index for the XML data, an array of groups of F&B index nodes (named *IGroups*) and a

QTable

id	Query	subQs	groups
1	//a	⊥	{1}
2	//b	⊥	{2}
3	//c	⊥	{3}
4	//d	{6}	{4}
5	//e	⊥	{5}
6	//c//d	⊥	{6}

IGroups

id	IGroup
1	(1)
2	(2) (6) (10)
3	(3) (5) (7)
4	(11)
5	(9) (12)
6	(4) (8)

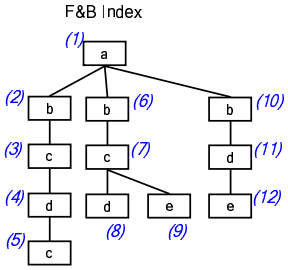


Figure 4.2: An Example AC-Index ('⊥' stands for a NULL pointer. *subQs* is a list of pointers pointing to *QTable.id*; *groups* is a list of pointers pointing to *IGroup.id*; *IGroup* is a list of pointers pointing to nodes in the F&B index)

query table (named *QTable*). Figure 4.2 shows the example AC-Index adapted for the frequent query $//c//d$.[1]

The F&B index is built in the initialization step of the AC-Index for the XML document. The *IGroups* is an array of groups of F&B index nodes. Intuitively, each group belongs to at least one query as part of the query result(s). One property that will become obvious later is that all the F&B index nodes within the same IGroup will either be accessed together for a query or none of them will be accessed. The QTable records a list of frequent queries. Each entry of the QTable consists of three fields: *Query*, *subQs* and *groups*. The Query field keeps the queries as plain strings. The subQs field is a list of child queries that are immediately contained by the current query. This design eliminates the data redundancy problem and also facilitates containment checking which is frequently used in query processing and adaptation processes. The *groups* field is a list of pointers, each of which points to one group in the *IGroup* table. The F&B index nodes associated with groups fields are those that belong to the current query but not belonging to any of its descendant queries. We remark that the reason for having the IGroup table is that it is possible for a group of F&B index nodes to be shared among multiple queries.

Let $Result(Q)$ be the result of a query in the form of a list of its corresponding F&B index nodes. Our design of the index has the following property for every query in the QTable:

$$Result(Q) = (\cup_{Q_i \in Q.subQs} Result(Q_i)) \cup Q.groups \tag{4.1}$$

Example 1 *Figure 4.2 shows the ACIndex for the example XML data tree in Figure 3.1(a). For the 4th query, $//d$, its subQs field is nonempty and points to the 6th query, $//c//d$. In addition, its IGroups field points to the 4th group in IGroup, which is essentially the F&B index node numbered (11) in the F&B index. Similarly, query $//c//d$ does not have any child query, and has only the 6th group in IGroup as its result. According to Equation* (4.1)*, the result of query $//d$ is F&B index nodes 11, 4, and 8.*

[1]The rest queries (i.e., $//tag$) are added in the initialization phase (See Section 4.3).

Algorithm 1 FindMatch (Q_n)

1: $mincost \leftarrow \infty$
2: **for all** query Q_i in the QTable **do**
3: $pos \leftarrow$ LastMatch (Q_n, $outputNode(Q_i)$)
4: $(prefix, suffix) \leftarrow$ SplitQuery (Q_n, pos) {We find the last occurrence of $outputNode(Q_i)$ in Q_n and thus split Q_n into two parts. }
5: **if** Q_i contains $prefix$ **then**
6: $Q_c^{up} \leftarrow$ GetComposingQuery (Q_i, $prefix$)
7: $Q_c^{down} \leftarrow suffix$
8: **if** EstimateCost (Q_i, $prefix$, $suffix$) $< mincost$ **then**
9: $mincost \leftarrow$ EstimateCost (Q_i, $prefix$, $suffix$)
10: **return** Q_i that achieves the $mincost$

Algorithm 2 ExecQuery (Q_n, Q_m, Q_c^{up}, Q_c^{down})

1: **for all** F&B index node $n \in Result(Q_m)$ **do**
2: $valid \leftarrow$ Eval (n, Q_c^{up})
3: **if** $valid =$ `TRUE` **then**
4: Output Eval (n, Q_c^{down})

4.3 Initializing the AC-Index

The initialization phase is similar to that of the APEX [4]: we add a list of hypothetical queries in the form of $//tag$ to the index. The difference is that the unit of manipulation in our AC-index is F&B index nodes instead of each individual elements in the XML document. The purpose of inserting such queries is to ensure that there *always* exists a query in the AC-index that contains any new queries.

The algorithm to build the initial AC-Index is rather straight-forward: (a) first we builds the full F&B index, (b) then we combine all the F&B index nodes with the same tag name into a group and put them into the IGroup table. (c) finally, we insert all the query $//tag$ into the QTable, and link them to the corresponding IGroup entries.

4.4 Query Processing

Given a new query Q_n, if it is identical to one of the existing queries in the AC-Index, it can be easily answered by fetching its result in a recursive manner, according to Equation (4.1). A feature of the AC-Index is that it may efficiently answer queries that does not exactly match the materialized queries, thus significantly boosts the performance of the system. We introduce two such strategies in the following.

Utilize Matching Queries The *first* approach is based on the idea that a materialized query, Q_m might offer some clues to finding the result of a new query Q_n, provided that Q_m *matches* Q_n. We use the following example to illustrate the intuition behind the "match" concept and the query processing method.

Example 2 *Consider* $Q_n = //a[./b]/c/d/e$ *and* $Q_m = //a/c/d$. *The prefixes of depth 3 for* Q_n *and* Q_m *are* $//a[./b]/c/d$ *and* $//a/c/d$, *respectively. Although they are not identical, it is obvious that the former is contained in the latter. The suffix of the query* Q_n *queries is* $./e$. *The observation is that we can answer* Q_n *by* Q_m *by refining* Q_m*'s result by an* upward composing query[2] $Q_c^{up} = .\backslash.\backslash[./b]$ *(where* $\backslash$ *denotes the parent axis in XPath), and then evaluation a* downward composing query $Q_c^{down} = ./e$ *for the qualified results after the previous step.*

We summarize the query processing algorithm using matching queries in the general case in Algorithms 1 and 2. We first invoke FindMatch to find the an existing frequent query, Q_m that has the minimum estimated cost to answer the new query Q_n, we then invoke ExecQuery using Q_m on top of the F&B index. Note that answering path queries using an F&B index, i.e., the Eval, is well studied in [9, 16]. In FindMatch, for each query Q_i in QTable, we compare it against the new query Q_n (Lines 3–4) and find the upward composing query and downward composing query (Lines 6–7) if Q_i matches Q_n (Line 5). The algorithm to obtain the composing query is due to [17]. Last, we select the Q_i that "best" matches Q_n in terms of the expected query processing cost for the two composing queries. Currently a naïve cost estimator is used that only takes into consideration the complexities of the composing queries. It is part of our future work to investigate and incorporate more accurate cost estimation methods into this work.

Utilizing Query Containment Relationships The *second* approach is based on the idea that a materialized query, Q_p might contains a superset of query result of a new query Q_n.

Example 3 *Consider* $Q_n = //a[./b]/c/d/e$ *and* $Q_p = //a//e$. *It is obvious that* $Q_p \supseteq Q_n$. *Thus we can obtain the result of* Q_n *by* validating Q_p*'s result against* Q_n. *The validation is essentially evaluating an upward composing query* $Q_c^{up} = .\backslash d\backslash c\backslash a[./b]$, *and has been implemented in function* Eval *(See Algorithm 2). On the other hand, if we have another query* $Q'_p = //a[./b]/c//e$, *it is obvious that since* $Q_p \supseteq Q'_p \supseteq Q_n$, *we should answer* Q_n *using the* minimal *query that contains* Q_n, *i.e.,* Q'_p.

In order to find the minimal query[3], Q_p, that contains the new query, Q_n, we just start with the query $//tag$, where $tag = \text{LastNode}(Q_n)$, and recursively descend to its first child node that contains Q_n.

4.5 Adapting the AC-Index to the Query Workload

The AC-Index is a workload-aware index and will adapt to new frequent queries and remove infrequent queries from its data structure, based on the information collected by the query statistics module. This is supported by two basic operations: *inserting* a query, and *deleting* a query.

[2]Given two queries X and Y, a composing query of X to Y is a query C such that $C \circ X = Y$ [11].

[3]That is, it is a query that contains Q_n and none of its descendant queries (in the QTable) contains Q_n.

Algorithm 3 UPDATEINDEX (Q_n, Q_p)

```
1: todo ← {Q_p} ∪ sibling(Q_p)
2: for all query Q_i ∈ todo do
3:    UPDATEQUERY (Q_i, Q_n)
4: Q_p.subQs ← Q_p.subQs ∪ Q_n
```

Algorithm 4 UPDATEQUERY (Q, Q_n)

```
1: if Q_n contains Q then
2:    for all queries Q' that is a parent of Q do
3:       Q'.subQs ← Q'.subQs ∪ Q_n
4:    Q_n.subQs ← Q_n.subQs ∪ Q {No need to descent into Q''s child queries}
5: else
6:    for all IGroup g in Q.groups do
7:       if g ⊆ Result(Q_n) then
8:          for all queries Q' that contains a reference to g AND Q' ≠ Q do
9:             Q'.subQs ← Q'.subQs ∪ Q_n
10:         Q_n.groups ← Q_n.groups ∪ g
11:      else if g partly overlaps with Result(Q_n) then
12:         Append a new group g' = g ∩ Result(Q_n) to IGroup table
13:         for all queries Q' that contains a reference to g AND Q' ≠ Q do
14:            Q'.groups ← (Q'.groups − g) ∪ g'
15:         Q_n.groups ← Q_n.groups ∪ g'
16:         g ← g ∩ g'
17:   for all sub-query Q_j ∈ Q.subQs do
18:      UPDATEQUERY (Q_j, Q_n) {Recursive call is needed as Q's child queries
         might overlap with Q_n}
```

Insert New Frequent Queries The main task here is to adjust the containment hierarchy for queries in QTable to accommodate a new query Q_n. We first find a minimal query, Q_p, that contains Q_n; this can be done using the same procedure as described in Section 4.4. Next, we need to insert Q_n as a child query of Q_p, and adjust the contents of existing queries as we need to ensure there is no duplicates. Lemma 1 helps us to limit the scope of the update. Algorithm 3 thus updates the AC-Index in a top-down, recursive manner for all the affected queries.

Lemma 1 *Consider inserting a new query Q_n under another query Q_p. Denote the sibling queries of Q_p as sibling(Q_p) (i.e., the queries correspond to the sibling nodes of Q_n in the AC-Index), then it is sufficient to adjust queries that are descendants of Q_p or any query in sibling(Q_p).*

The main work horse of the insertion algorithm is UPDATEQUERY (Algorithm 4). It needs to handle several cases. The first case is when Q_n contains Q according to the containment test, we only need to insert Q_n between all the parent query of Q and Q (Lines 2–4). The second case is when $Result(Q_n)$ and $Result(Q)$ partially overlaps. Note that $Result(Q_n)$ has already been computed as a list of F&B index nodes either by our AC-Index or by the F&B index. We need to iterate through all the IGroups in Q (Lines 6–16), perform split if necessary (Line 12), and link the IGroups to the appropriate queries (Lines 8–10,

and 13–16, for two different cases). Finally, we recursively update all the child queries of Q (Lines 17–18). We use the following example to further illustrate the algorithm.

QTable

id	Query	subQs	groups
1	//a	⊥	{1}
2	//b	⊥	{2}
3	//c	⊥	{3}
4	//d	{6, 7}	⊥
5	//e	⊥	{5}
6	//c//d	⊥	{6, 7}
7	//b[//e]//d	⊥	{4, 7}

IGroups

id	IGroup
1	(1)
2	(2) (6) (10)
3	(3) (5) (7)
4	(11)
5	(9) (12)
6	(4)
7	(8)

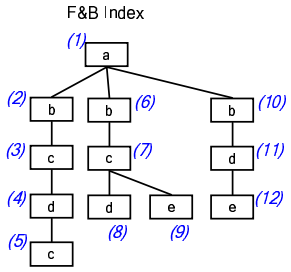


Figure 4.3: The AC-Index After Inserting Query $//b[//e]//d$

Example 4 *Consider the AC-Index shown in Figure 4.2. If we want to insert a new frequent query $Q_n = //b[//e]//d$, whose result is F&B index nodes $\{8, 11\}$, we first locate Q_p as the 4th query, $//d$. Since $//d$'s sibling queries are non-overlapping with Q_n, we only need to update $//d$ and its own descendant queries (i.e., only $//c//d$ in this example). The only IGroup for $//d$ is 11, and it is a subset of Q_n, and we only needs to add 11 to Q_n.groups (Lines 8–10). Next, we descend to the child query $//c//d$, whose result is F&B index nodes $\{4, 8\}$. We need to create a new group $g' = \{8, 11\} \cap \{4, 8\} = \{8\}$ (Line 12), and then update the groups information of both $//c//d$ and Q_n (Lines 13 – 16). The final AC-Index after update is shown in Figure 4.3.*

Delete Old Infrequent Queries When a query cease to be frequent among the recent N queries, we have to delete it from the AC-Index. The main task is to adjust the queries and groups associated with the query to be deleted (Q_d). In the interest of space, we briefly outline the procedure: we first find all the parent queries that contains Q_d in their subQs fields; we then copy $Q_d.subQs$ and $Q_d.groups$ to the result of Q_d's parent queries; after that, we can safely remove Q_d.

5 Further Optimizations

A number of optimizations have been identified and integrated to the AC-Index. In the interest of space, we only briefly introduce two optimizations here.

Filtering Checking containment relationship between two path queries is a relatively expensive operation. In addition, this operation is frequently called in query processing and adaptation algorithms (See Algorithms 1 and 4). We employ several simple tests to *filter out* cases where two queries definitely do not have containment relationships, and thus boost the performance. To test if Q contains q, we first perform the following tests:

- *output node test*: the output nodes of Q and q must be identical.

- *query size test*: let the total number of nodes in the pattern tree of a query Q be $size(Q)$, then we test if $size(Q) \leq size(q)$.
- *main query path test*: in the pattern tree for query Q, we denote the length of the path from the root of the tree to the output node as main query path length, $len(Q)$, then we test if $len(Q) \leq len(q)$.

We *only* invoke the query containment test [17] after Q and q pass all the above test. The correctness of the filtering tests can be derived directly from the pattern tree homomorphism.

Fast Index Adaptation Another time-consuming operation in updating the index is the intersection of a new query Q_n with the IGroups of existing queries in the index. While the UpdateQuery (Algorithm 4) is easy to understand and implement, we can devise a more efficient algorithm by performing multiple F&B index nodes intersections simultaneously. The basic idea is to first collect all the IGroups that needs to be intersected with $Result(Q_n)$, then perform all the intersections in the same time in a sort-merge fashion. More specifically, we maintain all the F&B index nodes in the IGroups sorted and thus only need to build a minheap out of all the participating IGroups. A single pass over $Result(Q_n)$ can compute all the intersection results with the help of the minheap.

6 Experimental Evaluation

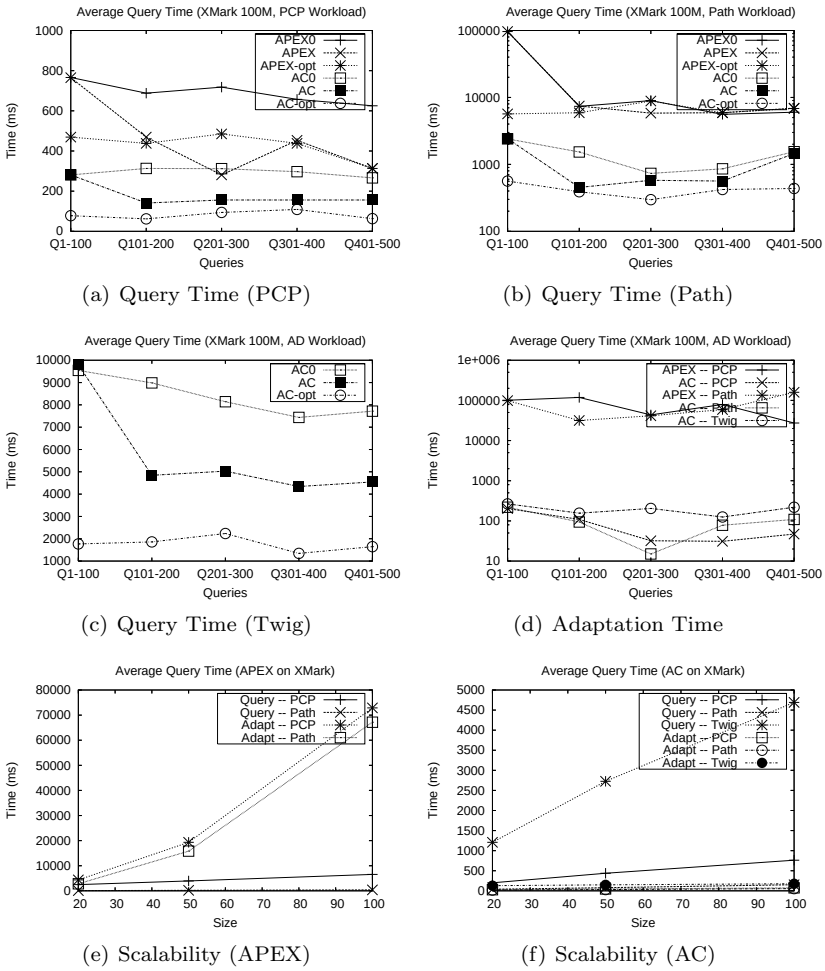
In this section, we report experimental results conducted on AC-Index (abbreviated as **AC**). The system we selected for comparison is APEX index [4] (abbreviated as **APEX**), as APEX is most closely related to our AC-Index and it is also one of the best query-adaptive indexes. We implemented all the algorithms in Java 1.4. All the experiments were conducted on a PC of 3.2GHz CPU, 2GB memory, and 80G hard disk. The operating system is Windows XP. We measure the elapsed time of all systems in terms of query processing time and update time. The dataset we used is XMark[15] — a widely used benchmark dataset modeling an auction site. We generate datasets of sizes 20M, 50M and 100M.

We generated three kinds of query workload named **PCP**, **Path**, and **Twig**. PCP workload includes queries that begins with a self-or-descendant axes (//) following by a path expression consisting of only parent-child axes (/). Path workload includes single path queries (i.e., //-edge can appear anywhere in the query). Twig workload is the most general workload allowing also branching queries. Note that APEX cannot support Twig workload.

In the interest of space, we omit the details of the query workload generation methods. Essentially we generated random, non-empty queries according to the workload characteristics. In each workload, we have a number of frequent queries[1] and they collectively accounts for 80% of the queries. All workloads have 500 queries and are divided into 5 batches[2]. In order to simulate the situation where frequent query distribution might vary over time, we fine tune

[1] i.e., whose frequency is above a *minsup* threshold.

[2] APEX cannot accommodate frequent updates. Therefore, we use a large batch size.



(a) Query Time (PCP)

(b) Query Time (Path)

(c) Query Time (Twig)

(d) Adaptation Time

(e) Scalability (APEX)

(f) Scalability (AC)

Figure 6.1: Experiment Results

the query workloads such that the common frequent queries in two adjacent batches are only 40%.

Query Performance We measure the query performances of APEX and AC against a 100M XMark dataset. For each indexes (I), we apply three variations: (a) I0: we construct the initial index, but does not adapt it to the query workload; (b) I: after each batch of queries, we adapt the index to the *past* workload; (c) I-opt: we adapt the index to the*next* batch of workload. We note that I0 is the baseline performance of a non-adaptive index, and I-opt is an offline optimal adaptive index. Measuring the performance of I0 and I gives us a better picture of how I performs.

The query performances for different workloads are shown in Figures 6.1(a), 6.1(b) and 6.1(c). We can draw the following observations:

- The adaptive versions of the indexes outperform their static versions (for

both APEX and AC). This is expected as adaptive index can reduce the query processing time of frequent queries.

- The performances of both indexes are close to their optimal versions. There are still some fluctuations which are mainly due to the change of the frequent query distributions – indexes adaptive to the current batch of queries are still sub-optimal for the next batch of queries.
- The optimal versions of the indexes can greatly outperform their static versions. Hence, if the distribution of frequent queries are stable, both index would approach the performance of their optimal versions.
- The average query time quickly drops after the first adaptation and remains fairly stable in the subsequent updates. This shows that both indexes can quickly adapts to the query workload and start to perform quite well even if there are small fluctuations in the query workload.
- Obviously, AC outperforms APEX, especially for the Path workload. This is because APEX cannot deal with queries in the Path workload directly, and have to decompose queries into several fragments and then perform multiple joins to obtain the query results.
- For the Twig queries, AC's performance is between AC0 and AC-opt. This is expected as we have only 40% of the queries that are common to adjacent batches of queries. Hence, AC adapts to the 80% of the queries, but only half of them are frequently issues in the next batch. This also shows that if the frequent query distribution is stable, AC can approach the performance of AC-opt, which greatly reduces the query processing time for frequent branching queries.

Update Performance We show the adaptation efficiency of APEX and AC for different workloads in Figure 6.1(d). We can draw the following observations:

- AC significantly outperforms APEX in update performance. This is mainly because APEX need to traverse the *entire* XML data tree for each and every update. In contrast, our AC-Index uses the query hierarchy to narrow down the update scope and rearrange F&B index nodes instead of its extents.
- The average adaptation time for the three workload load is generally ordered as PCP $<$ Path $<$ Twig. This is expected as both the containment checking and cost of updating queries and their associated IGroups become more and more expensive in that order. Although not shown here, we note that the optimized update method (See Section 5) contributes significantly to the update efficiency under the Twig workload, where the query containment hierarchy is the most complicated.
- The adaptation time for AC is only a small fraction of the query time. So even if we include the cost of update into the AC's query time, it still outperforms both the static AC (AC0) and APEX.

Scalability We use the same set of query workloads on three sets of XMark datasets with different sizes (20M, 50M, and 100M). We show the scalability results of APEX and AC in Figures 6.1(e) and 6.1(f), respectively. We can draw the following observations:

- Both APEX and AC needs more time for query processing and adaptation with the increase of the data sizes.
- Since at least the sizes of the query results are increasing, we expect at least a linear increase in query times. The query time of APEX seems to be increasing more rapidly with the size than AC does. A similar trend can be observed regarding the adaptation time of both systems too.

7 Conclusions and Future Work

In this paper, we introduce the AC-Index, which is a workload-adaptive index for XML branching queries. The AC-Index organizes frequently occurring queries and their results in the query workload as in a hierarchical and non-redundant way. Efficient index construction, query processing and adaptation algorithms have been proposed. The effectiveness of the proposed index has been demonstrated in the experiment.

As one of the future work, we plan to incorporate a cost-based query processing module into the index. The basic idea is to take into consideration both the frequencies of the queries and their processing costs. Another future work is to generalize the index to support values-based predicates in the query as well as aggregate queries.

Bibliography

[1] Andrey Balmin, Fatma Özcan, Kevin S. Beyer, Roberta Cochrane, and Hamid Pirahesh. A framework for using materialized XPath views in XML query processing. In *VLDB*, pages 60–71, 2004.

[2] A. Berglund, S. Boag, D. Chamberlin, and M. F. Fernández. XML path language (XPath) 2.0 W3C candidate recommendation. World Wide Web Consortium, 2005.

[3] Li Chen and Elke A. Rundensteiner. ACE-XQ: A cache-aware XQuery answering system. In *WebDB*, pages 31–36, 2002.

[4] Chin-Wan Chung, Jun-Ki Min, and Kyuseok Shim. APEX: An adaptive path index for XML data. In *SIGMOD*, pages 121–132, 2002.

[5] Brian F. Cooper, Neal Sample, Michael J. Franklin, Gísli R. Hjaltason, and Moshe Shadmon. A fast index for semistructured data. In *VLDB*, pages 341–350, 2001.

[6] Damien K. Fisher and Raymond K. Wong. Adaptively indexing dynamic XML. In *DASFAA*, pages 233–248, 2006.

[7] Roy Goldman and Jennifer Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *VLDB*, pages 436–445, 1997.

[8] Hao He and Jun Yang. Multiresolution indexing of XML for frequent queries. In *ICDE*, pages 683–694, 2004.

[9] Raghav Kaushik, Philip Bohannon, Jeffrey F. Naughton, and Henry F. Korth. Covering indexes for branching path queries. In *SIGMOD*, pages 133–144, 2002.

[10] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon, and Ehud Gudes. Exploiting local similarity for indexing paths in graph-structured data. In *ICDE*, pages 129–140, 2002.

[11] Bhushan Mandhani and Dan Suciu. Query caching and view selection for XML databases. In *VLDB*, pages 469–480, 2005.

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

[13] Tova Milo and Dan Suciu. Index structures for path expressions. In *ICDT*, pages 277–295, 1999.

[14] Chen Qun, Andrew Lim, and Kian Win Ong. D(k)-index: An adaptive structural summary for graph-structured data. In *SIGMOD*, pages 134–144, 2003.

[15] A. Schmidt, F. Waas, M. Kersten, D. Florescu, I. Manolescu, M. Carey, and R. Busse. The XML benchmark project. Technical Report INS-R0103, CWI, April 2001.

[16] Wei Wang, Hongzhi Wang, Hongjun Lu, Haifeng Jiang, Xuemin Lin, and Jianzhong Li. Efficient processing of XML path queries using the disk-based F&B index. In *VLDB*, pages 145–156, 2005.

[17] Wanhong Xu and Z. Meral Özsoyoglu. Rewriting XPath queries using materialized views. In *VLDB*, pages 121–132, 2005.