A Logic-based Approach to Cache Answerability for XPath Queries

M. Franceschet[1,2] and E. Zimuel[1,2]

[1]Informatics Institute, University of Amsterdam,
Kruislaan 403 – 1098 SJ Amsterdam, The Netherlands
[2]Dipartimento di Scienze, Università "Gabriele D'Annunzio",
Viale Pindaro, 42 – 65127 Pescara, Italy

Abstract. We extend a recently proposed model checking-based algorithm for the evaluation of XPath queries with a cache strategy to store the results of the (most frequently) asked queries and to re-use them at occurrence. We experimentally show that, as soon as the cache is warm, the proposed optimization is quite effective. We complement our proposal with a broad experimental comparison of different strategies for XPath query processing.

1 Introduction

The XML Path Language version 1.0 (XPath, in the following), is a query language for XML documents proposed in 1999 by the World Wide Web Consortium(W3C) [1]. Compared to some later proposals of the W3C, like XPath 2.0 [2] and XQuery [3], the XPath language, and in particular its navigational fragment, or Core XPath [4], is simple, clean, and intuitive. As a result, XPath has become very popular among XML users and many software houses have extended their products with XPath tools. Moreover, researchers in both the computational logic and the database communities devised quite a large number of solutions for the evaluation of XPath queries, including tree traversal methods [4–6], model checking-based methods [7–9], automata-based methods [10–12], join-based methods [13–15], and sequence matching-based methods [16, 17]. However, we are aware of few papers that aim to compare the relative performance of such algorithms ([18] compares join-based and sequence matching-based methods, while [19] evaluates XML indexes for structural joins).

This paper gives two contributions. We extend the above list of evaluation methods with a logic-based approach to answer XPath queries with the aid of cache mechanisms. Moreover, we make a thorough experimental comparison of the following four evaluation techniques for XPath:

TopXPath (Section 8.1 of [4]). The idea that lies behind this algorithm is to rewrite the original query into a Boolean combination of filter-free paths (sequences of steps without filters). The evaluation of the filter-free path is performed by reading the path string from left to right and sending the output of

the current step to the input of the next step, if any. For instance, consider the query:

$$\pi[\phi] = /\text{child} :: \text{site}/\text{child} :: \text{regions}[\text{descendant} :: \text{item}/\text{following} :: \text{payment}]$$

The method works in two phases. First, the query filter ϕ is rewritten by reading it from right to left and inverting each axis. The inverted filter becomes:

$$\varphi = \text{self} :: \text{payment}/\text{preceding} :: \text{item}/\text{ancestor} :: *$$

Then, the above query is evaluated as $\pi \cap \varphi$, that is, by taking the intersection of the result of π (with the singleton containing the tree root as initial context set) and the result of φ (with the set of all tree nodes as initial context set).

BottomXPath ([9]). The idea here is to rewrite the original query into a modal formula and then evaluate the formula *bottom-up*, that is, each formula is evaluated after the evaluation of its subformulas. As an example, consider again the above query $\pi[\phi]$. The corresponding modal formula is:

$$\text{regions} \wedge \langle\text{parent}\rangle(\text{site} \wedge \langle\text{parent}\rangle\text{root}) \wedge \langle\text{descendant}\rangle(\text{item} \wedge \langle\text{following}\rangle\text{payment})$$

where tags are interpreted as atomic propositions (root is a propostion that is true exactly at the tree root) and each axis is simulated by a corresponding modality. The modal formula is evaluated bottom-up exploiting the fact that the truth value of any subformula can be computed from the truth values of its direct subformulas.[1]

CacheBottomXPath. This is a cache optimization of BottomXPath that we propose and evaluate in this paper (see Section 2 for the details). The query is first converted into a modal formula and then chopped into a set of subformulas. Then, each subformula, in bottom-up order, is searched in the cache. If the subformula is found, no evaluation is performed, since the result has been already computed. Otherwise, the subformula is evaluated and its result is possibly stored in the cache.

Arb ([12]). This is an automata-based method. The XML document is first converted into a binary tree representation. Then, two deterministic binary tree automata, one working bottom-up and the other one working top-down, are generated from the query. The actual evaluation is performed in two steps: (i) first, the bottom-up query automaton runs on the XML binary tree; (ii) then, the top-down query automaton runs on the XML binary tree enriched with information computed during the bottom-up run. Finally, the entire XML document is returned with selected nodes marked up in XML fashion.

An analysis of the worst-case computational complexity of the above four methods does not help much to determine the most efficient evaluation strategy. Let us focus on the navigational part of XPath known as Core XPath [4], which

[1] This bottom-up principle holds for many modal and temporal logics. A notable example is Computation Tree Logic (CTL), a popular specification language in the context of formal verification of software and hardware systems [20].

is supported by all the above methods. Let k be the query complexity and n be the data complexity. On Core XPath, the worst-case complexity of TopXPath, BottomXPath, and CacheBottomXPath is $O(k \cdot n)$, while Arb terminates in $O(K + n)$, where K is an exponential function of k. A closer look inside the four algorithms reveals the following. In order to solve a query of length k on a tree of size n it happens that: (i) TopXPath visits each tree node a number of times between 0 and k (each node might be visited a different number of times) (ii) BottomXPath visits each tree node exactly k times; (iii) CacheBottomXPath visits each node the same number of times between 0 and k and it spends extra time proportional to the cache loading factor in order to search into the cache; and (iv) Arb visits each tree node twice (independently on the query complexity) and it spends extra time that might be exponential in k in order to build the tree automata. All the algorithms spend a constant amount of time at each node but BottomXPath (and its cache-based version) is particularly efficient since it operates mostly on Boolean values.

To have a better understanding of the relative performance of the methods under testing, we conducted a probing *experimental* evaluation on synthetic and simulated real data. The main goals of our investigation are: (i) to understand the effectiveness of the cache optimization introduced in CacheBottomXPath; (ii) to compare the performance of the top-down and bottom-up approaches implemented in TopXPath and BottomXPath, respectively, on randomly generated data, and (iii) to test the scalability of the automata-based method encoded in Arb when the query length grows. In particular, is the automata construction step a bottleneck for query processing in Arb?

The rest of the paper is as follows. In Section 2 we review BottomXPath and describe CacheBottomXPath. The results of our experimental evaluation are discussed in Section 3. We conclude in Section 4.

2 XPath Evaluation Methods

Even if the algorithms mentioned in Section 1 work on, or can be easily extended to, full XPath, we will evaluate them on the navigational fragment of XPath, or Core XPath, that was defined in [4]. With respect to full XPath, this fragment disallows the axes attribute and namespace, node tests different from a tag or *, comparison operators and functions. What remains can be used to navigate the XML tree only. The algorithms that we test essentially differ only on this fragment.

As noticed in [21], Core XPath can be viewed as a Modal Logic, interpreted over tree structures, whose modalities behave like the XPath axes. Modal Logic [22] extends Propositional Logic with modalities that, similarly to XPath axes, are used to browse the underlying relational structure. Let Σ be a set of proposition symbols. A *formula* in the multi-modal language is defined as follows:

$$\alpha = p \mid \alpha \wedge \alpha \mid \alpha \vee \alpha \mid \neg\alpha \mid \langle R_i \rangle \alpha$$

where $p \in \Sigma$ and $1 \leq i \leq c$ for some integer $c \geq 1$. A multi-modal logic for XPath contains a propositional symbol for each XML tag and a modality $\langle \mathrm{X} \rangle$ for each XPath axis X. Modal formulas are interpreted at a given state of a given model in the usual way [22]. E.g., $\langle \mathrm{X} \rangle \alpha$ is true at state s iff there exists a state t reachable from s through the relation X such that α is true at t. The *truth set* of a formula α w.r.t. a model M is the set of states of M at which α is true.[2]

We refer to [4] (Section 8.1) and [12], respectively, for a complete description of TopXPath and Arb. In the rest of this section, we review BottomXPath [9] and we introduce CacheBottomXPath. BottomXPath inputs an XML tree T and a Core XPath query q and returns the answer set for q with respect to T in the following two steps:

BottomXPath(T, q)

1: translate q into a modal formula α_q;
2: retrieve the truth set of α_q w.r.t. T;

The translation of step 1 works as follows. Each tag is mapped to a corresponding proposition symbol and * is mapped to the truth value true. Moreover, a new proposition root is introduced to identify the tree root. The query path is read from right to left and each axis (not belonging to a filter) is mapped to the modality corresponding to the inverse of the axis. Finally, each query filter is translated by reading it from left to right and by mapping each axis to the corresponding modality and each Boolean operator to the corresponding Boolean connective. For instance, the query:

/child::a[parent::b/following::c]/descendant::d[preceding::e or not(following::*)]

is mapped to the formula:

$\mathrm{d} \wedge \langle \mathrm{ancestor} \rangle (\mathrm{a} \wedge \langle \mathrm{parent} \rangle \mathrm{root} \wedge \langle \mathrm{parent} \rangle (\mathrm{b} \wedge \langle \mathrm{following} \rangle \mathrm{c})) \wedge$
$(\langle \mathrm{preceding} \rangle \mathrm{e} \vee \neg \langle \mathrm{following} \rangle \mathrm{true})$

The truth set of the resulting modal formula (step 2 of BottomXPath) is computed by the procedure XPathCheck as follows. XPathCheck inputs an XML tree T and an XPath modal formula α and returns the truth set for each subformula of α (including α itself) in document order. The algorithm is similar to the model checking procedure for the temporal logic CTL, a popular specification language in the context of finite-state program verification [20]. We first describe the data structures used by the algorithm. XPathCheck takes advantage of a Boolean matrix A, where the rows represent formulas and the columns represent nodes, in order to label nodes with formulas that are true at them. Initially, each entry of A is set to 0. For each subformula of α numbered with i and each node of T numbered with j, the procedure sets $A[i, j]$ to 1 if and only if the formula i is true at the node j. Moreover, XPathCheck stores the tree T as a set of linked objects each of them representing a tree node. Each object contains a field with the preorder rank of the node, a field containing the XML

[2] In computational logic, the problem of finding the truth set of a formula is well-known as the (global) *model checking problem* [20].

tag of the element that the node represents, and pointers to the parent, first child, right and left siblings nodes. Finally, XPathCheck represents the formula α as its parse tree PT_α. Each node of PT_α represents a subformula β of α and it is stored as an object containing a field with the main operator of β, a field containing the index of the corresponding row in A, and pointers (at most 2) to the argument nodes of the main operator of β. It is convenient to insert in A the subformulas of α in postorder with respect to a visit of PT_α (so that the subformulas of α can be scanned bottom-up) and the nodes of T in preorder with respect to a visit of T (so that each truth set is sorted in document order).

XPathCheck works as follows. Given a tree T and a formula α, it processes each subformula β of α by visiting the parse tree PT_α in postorder. In this way, each subformula of β is checked before β itself is verified. The verification of β depends on the its main operator:

1. if β is root, then XPathCheck sets $A(\beta, 1)$ to 1 (the first column of A is associated to the tree root);
2. if β is *, then XPathCheck sets $A(\beta, j)$ to 1 for each node j;
3. if β is a tag a, then XPathCheck sets $A(\beta, j)$ to 1 for each node j tagged with a;
4. if β is $\beta_1 \wedge \beta_2$, then, for each node j, XPathCheck sets $A(\beta, j)$ to 1 if $A(\beta_1, j) = 1$ and $A(\beta_2, j) = 1$ (and similarly for the disjunction and negation cases)[3];
5. if β is $\langle \mathrm{X} \rangle \beta_1$, then, for each node j, XPathCheck sets $A(\beta, j)$ to 1 if there exists a node k reachable from j trough the relation induced by X such that $A(\beta_1, k) = 1$.

The check of subformulas of the form $\langle \mathrm{X} \rangle \beta_1$ depends on the axis X. In general, it is a tree searching algorithm that possibly labels nodes with $\langle \mathrm{X} \rangle \beta_1$. For instance, if the axis is descendant, then the procedure first retrieves the nodes labelled with β_1 and then it labels each ancestor of such nodes with $\langle \text{descendant} \rangle \beta_1$ if the ancestor is not already labelled with it. Notice that, for each axis X, the formula $\langle \mathrm{X} \rangle \beta_1$ can be checked by visiting each tree node only a constant number of times, hence in linear time with respect to the number of nodes of the tree. Moreover, most of the operations are performed on Boolean values. It follows that XPathCheck runs in time proportional to the product of the formula length and the XML tree size. Since the mapping from queries to formulas (step 1 of BottomXPath) takes linear time and the resulting formulas have linear lengths with respect to the lengths of the input queries, we can conclude that BottomXPath runs in $O(k \cdot n)$, where k is the query length and n is the XML tree size.

2.1 Cache Answerability for XPath Queries

BottomXPath repeats the computation of the truth set for each instance of the same subformula. This can be avoided as follows. Both a *formula cache* M,

[3] Notice that the matrix entries for β_1 and β_2 are known when β is processed, since β_1 and β_2 are subformulas of β and hence their postorder ranks in the parse tree are smaller than the postorder rank of β.

storing the past formulas, and a *truth set cache* A, storing the truth sets for past formulas, are maintained. When a new formula is checked, first the formula is searched in M. If the formula is found, then no further processing is necessary. Otherwise, the sub-formulas of the original formula that are not present in M are added to M and their truth sets are computed and added to A.

We now describe the optimization in more detail. Consider a formula α. We represent α with its parse tree PT_α and the truth set cache with a Boolean matrix A as described above. The new entry is the formula cache that is implemented using a *hash table* M where the keys are the formula strings and the collision resolution method is by chaining. Each object of the linked lists associated to the hash table contains the formula string, the index of the corresponding row in A, and the usual pointer to the next object in the list. The processing of α is as follows: (a) the parse tree PT_α is generated, (b) PT_α is visited in postorder and, for each node (subformula) x, the following steps are done:

1. the formula string s associated to x is built by visiting the tree rooted at x;
2. the string s is searched in the hash table M;
3. if an object y with key s is found in M, then x is updated with the row index of the matrix A corresponding to the formula s, which is read from y;
4. otherwise, a new row from A, say l, is assigned to the formula s, a new object for s is inserted in the hash table with the row index l, the object x is updated with the row index l, and finally the truth set for s is computed possibly updating the l-th row of A.

The described optimization is particularly effective in a client/server scenario. Consider the case of a static XML document on a server and a number of users ready to repeatedly query the document from remote clients. The server stores the query answer cache for all the posed queries, while each client stores the cache for the queries posed locally. It is possible for the same user to pose similar queries (containing common sub-queries) at different stages. Moreover, it is likely that different users ask for similar or even for the same query. When a query is posed on a client, first an answer for the query is searched in the local cache stored on the client. If the answer is found, then it is returned to the user. Otherwise, the sub-queries of the original query that do not have a cached answer are shipped to the server and the answers for them are searched in the global cache stored on the server. The found answers, if any, are shipped to the client user and the client cache is updated with them. The missing answers are computed on the server, the global cache on the server is updated with them, the answers are shipped to the client user and finally the client cache is also updated. When the querying is done, the query cache can be stored in secondary memory and loaded again if the querying restarts.

An important issue involved in the described optimization concerns the cache maintenance strategy [23]. Such a strategy specifies how to warm-up the cache, that is, how to populate the cache in advance with queries that are likely to be frequently asked. Moreover, it specifies when to insert new queries and to delete old ones from the cache. We did not implement any particular cache maintenance

strategy in CacheBottomXPath. Indeed, our current goal is to compare high-level evaluation strategies for XPath. Such an evaluation might hint how to program an optimized full-fledged evaluator for XPath.

3 An Experimental Evaluation

This section contains the results of our experiments on both synthetic and simulated real data. We implemented TopXPath, BottomXPath, and CacheBottomXPath in C language, taking advantage of Expat XML document parser (`http://expat.sourceforge.net`). We used the Arb implementation that is available at Christoph Koch's website `http://www.infosys.uni-sb.de/~koch/projects/arb`. We ran all the programs in main memory. We performed our experiments with XCheck [24], a benchmarking platform for XML query engines. We ran XCheck on an Intel(R) Xeon(TM) CPU 3.40GHz, with 2 GB of RAM, running Debian Gnu/Linux version 2.6.16. All times are in seconds (or fraction). Processing a query involves several steps, including parsing the document, compiling and processing the query, and serializing the results. The *response time* is the time to perform all these steps. We mostly measured the *query processing time* (the time spent for the pure execution of the query), which is the most significant for our purposes. Because of space limitations, this section contains only a fraction of the experiments and of the data analysis that we performed. The complete experimental evaluation is available at the website associated to this paper: `http://www.sci.unich.it/~francesc/pubs/xsym06`. The website includes also the source codes of the programs that we implemented for this paper. We stress that all the experiments that we performed are based on data (XML documents and queries) and software (query engines and data generators) that are publicly available and hence they are completely reproducible. We tried to devise experiments in the spirit of *scientific testing* as opposed to competitive testing [25], that is, experiments that allow to draw general conclusions instead of comparing absolute time values. In the rest of this section we will abbreviate TopXPath as TXP, BottomXPath as BXP, and CacheBottomXPath as CBXP.

3.1 Experiments on Synthetic Data

This section contains the results of our experiments on synthetic (i.e., artificial) data. Due to their flexibility, synthetic data are useful to uniformly test specific capabilities of an engine by using specific benchmarks (also known as micro-benchmarks [26]). We evaluated the performance of the XPath engines under consideration while changing the following parameters: data size, data shape, query length, and query type.

We performed different experiments with different goals. An *experiment* consists of an input, and output and a goal. The experiment's *goal* is what we want to measure. The experiment's input is a set of XML documents (*data set*) and a set of XPath queries (*query set*). Finally, the experiment's output is a set

of results that need to be interpreted with respect to the goal of the experiment. We generated the data set with MemBeR data generator [26]. It allows controlling different parameters for an XML document, including tree size, tree height, and maximum node fanout. As for the queries, they were generated with XPathGen, a random Core XPath query generator that we implemented for this paper.[4] XPathGen can generate queries with an arbitrary length and with an arbitrary nesting degree of filters. It allows controlling the following parameters for a query: length, axes probabilities, and filter probability. The query length is the number of atomic steps of the form axis::test that the query contains. Each axis has a corresponding probability of being selected during the query generation. This allows the generation of queries that are biased towards some of the axes. Finally, the filter probability controls the filter density in the query (that is, the number of query filters divided by the query length). This allows the generation of path-oriented queries (when the filter probability is low) and filter-oriented queries (when the filter probability is high). It is worth noticing that, in each generated query, each node test is * and the first step of the query is always descendant::*. As a consequence, it is very unlikely to generate queries with an empty result. Moreover, the intermediate and final results of the generated queries are quite large.

We used the documents described in the table below, where the meaning of the columns is as follows: n is the tree size (the number of tree nodes), avgd is the average node depth (the depth of a node is the length of the unique path for the node to the root), maxd is the maximum node depth (the height of the tree), avgf is the average node fanout (the fanout of a node is the number of children of the node), and maxf is the maximum node fanout.

doc	**n**	**avgd**	**maxd**	**avgf**	**maxf**	**doc**	**n**	**avgd**	**maxd**	**avgf**	**maxf**
D1	200,000	7.4	8	2.8	5	D5	500,000	12	13	2	61
D2	500,000	4	4	26	35	D6	50,000	6.7	7	4	9
D3	500,000	6.8	7	6	12	D7	5,000,000	6.9	7	8	16
D4	500,000	9.5	10	3	16	D8	100,000	6.8	8	4.6	5

We performed the following experiments:

Experiment E1. With this experiment we tested the engines' performance while increasing the filter probability. We set the query length $k = 10$ and varied the filter probability $p \in \{0, 0.25, 0.5, 0.75, 1\}$. Each axis is equi-probable. For each value of p, we generated 25 queries, ran them against document D1, and measured the overall query processing time. The results are below:

E1	**p = 0**	**p = 0.25**	**p = 0.5**	**p = 0.75**	**p = 1**
TXP	19.03	22.38	23.8	25.79	27.89
BXP	16.13	16.19	16.18	16.19	16.52
CBXP	10.28	8.97	8.51	8.68	10.03
Arb	15.41	14.47	13.96	14.19	14.31

[4] The source code is available at the paper website.

Interestingly, TXP shows worse performance as the query filter density increases. This can be explained as follows. During query evaluation, TXP separates the query into paths and filters. Paths are evaluated form the tree root, while filters are processed with respect to the set of all tree nodes, which is more expensive. Hence, filter-oriented queries are more difficult for TXP. Independently on the filter probability, TXP is always the slowest while CBXP is always the fastest. Arb and BXP are competing.

Experiment E2. With this experiment we tested the engines' performance while increasing the query length. We set the filter probability $p = 0.25$ and varied the query length $k \in \{5, 10, 15, 20, 25\}$. All the axes are equi-probable. For each value of k, we generated 25 queries, ran them against document D1, and measured the overall query processing time. We also computed the query scalability factors.[5] The results are below (where the columns named qs contain the query scalability factors with respect to the adjacent query lengths):

E2	**k = 5**	**qs**	**k = 10**	**qs**	**k = 15**	**qs**	**k = 20**	**qs**	**k = 25**
TXP	13.65	0.82	22.31	0.94	31.57	0.85	35.95	1.09	48.87
BXP	9.42	0.86	16.25	0.88	21.5	1	28.07	1.03	36.09
CBXP	4.98	0.89	8.89	1.19	12.47	1.01	16.83	1.05	22.07
Arb	14.36	0.55	15.79	1.44	34.2	25.95	1183.41	5.52	8167.5

TXP, BXP and CBXP scale up linearly when the query length is increased. On the contrary, the performance of Arb is discontinuous, as witnessed by the query scalability factors. It is almost irrelevant to the query length up to length 10. However, for longer queries, the performance of Arb grows exponentially in the query length. This can be explained as follows. For long query strings, the time spent by Arb during the automata construction, which exponentially depends on the query length, dominates the pure query evaluation time (the time spent to run the automata), which is independent on the query length. CBXP shows the best global performance. BXP comes as second, while TXP and Arb competes up to length 15, where the performance of Arb explodes exponentially.

Experiment E3. With this experiment we tested the engines' performance while changing the document tree shape. We set the query length $k = 5$ and the filter probability $p = 0.25$. All the axes are equi-probable. We generated 25 queries according to these paremeters. As for the data set, we used documents in the sequence (D2, D3, D4, D5). All the document trees in the sequence have the same size and vary their shape. In particular, the trees in the sequence move from wide-and-short to narrow-and-long trees. For each document, we measured the overall query processing time. The results are below:

[5] Given a document D and two queries q_1 and q_2 of length l_1 and l_2 respectively, with $l_1 < l_2$, let t_1 be the processing time for q_1 on D and t_2 be the processing time for q_2 on D. The *query scalability factor*, as defined in [27], is the ratio $(l_1 \cdot t_2)/(l_2 \cdot t_1)$. If this factor is smaller than 1 (respectively, equal to 1, bigger than 1), then the engine scales up sub-linearly (respectively, linearly, super-linearly) when the query length increases.

E3	**h = 4**	**h = 7**	**h = 10**	**h = 13**
TXP	28.45	31.83	34.89	36.41
BXP	19.88	22.45	24.79	25.26
CBXP	10.3	12.11	13.65	13.39
Arb	38.42	38.37	37.74	30.4

Interestingly, TXP, BXP and CBXP perform better on wide-and-short trees, while Arb gives its best on narrow-and-long trees. Recall that the natural data model for Arb is a binary tree (arbitrary trees are preprocessed and converted to binary trees). This might explain why Arb is fastest on structures that are close to binary trees. Notice that document D5, on which Arb shows the best performance, has an average fanout of 2. As for global performance, CBXP is still the fastest. BXP comes as second, while TXP and Arb competes, with Arb outperforming TXP on narrow document trees.

Experiment E4. With this experiment we tested the engines' performance while increasing the document tree size. We set the query length $k = 5$ and the filter probability $p = 0.25$. All the axes are equi-probable. We generated 25 queries according to these paremeters. As for the data set, we used the sequence (D6, D3, D7) of documents of increasing size and width. Each document in the sequence has the same maximum depth and roughly the same average node depth. For each document, we measured the overall query processing time. We also computed the data scalability factors.[6] The results are below (where the columns named ds contain the data scalability factors with respect to the adjacent document sizes):

E4	**n = 50,000**	**ds**	**n = 500,000**	**ds**	**n = 5,000,000**
TXP	3.1	0.93	28.71	0.97	278.8
BXP	2.16	0.96	20.73	0.97	202.06
CBXP	1.22	0.95	11.62	0.98	113.58
Arb	4.28	0.65	27.69	1.06	293.01

Mostly, the scalability of all engines is linear. The sub-linear behavior of Arb in the first track depends on the time taken to build the tree automata, which is independent on the tree size. This time is dominated by the pure query processing time (the time to run the tree automata) in the second track. As for global performance, CBXP is still the fastest. BXP comes as second, while TXP and Arb are close.

Experiment E5. With this experiment we tested the engines' performance while changing the axes permitted in the queries. We set the query length $k = 5$, the filter probability $p = 0.25$, and varied the set of allowed axes as follows: (a) all the axes, (b) all the axes but following and preceding, (c) all the vertical axes (i.e., child, parent, descendant, ancestor), (d) all the forward vertical axes (i.e., child and descendant). In each case, we generated 50 queries, ran them against document D3, and measured the overall query processing time. The results are below:

[6] The *data scalability factor* is defined as for the query scalability factor except for the fact that it uses the size of the XML tree instead of the length of the query.

E5	(a)	(b)	(c)	(d)
TXP	56.33	38.8	37.1	39.86
BXP	40.48	28.35	28.77	29.12
CBXP	18.07	10.07	9.74	7.45
Arb	63.7	46.4	46.33	51.32

The message is clear: for all the engines under testing, following and preceding axes are the most expensive ones (compare columns (a) and (b)). If we prohibit these axes, the response time is almost the half. On the contrary, horizontal axes following-sibling and preceding-sibling are not problematic (compare columns (b) and (c)). The same for backward vertical axes parent and ancestor (compare columns (c) and (d)). In any case, CBXP is still the fastest, followed by BXP, TXP and Arb in this order.

Experiment E6. With this experiment we tested the performance of the cache optimization introduced in CBXP. We fixed the cache size to 64. We allowed all axes with the same probability and generated 500 queries by randomly choosing, for each query, a value for the query length $k \in \{1, 2, \ldots 10\}$ and for the filter probability $p \in [0, 1]$. We ran the queries against document D8. The results are illustrated in Figure 1. The left plot shows the processing times of the different engines for all the 500 queries. The right plot sums the processing times on adjacent intervals of 100 queries. In both plots, from top to bottom, Arb corresponds to the first (pink) line, TXP to the second (red) line, BXP to the third (green) line, and CBXP to the fourth (blue) line.

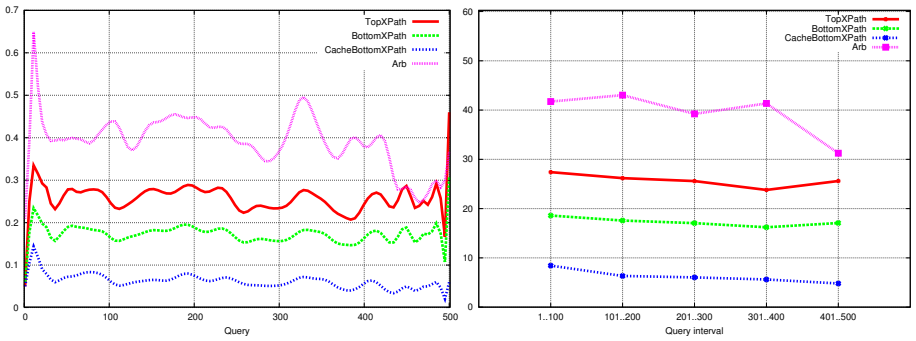

Fig. 1. The effectiveness of the cache

As for the effectiveness of the cache optimization, notice that CBXP is almost 3 times faster than BXP and, as expected, its relative performance increases as more queries are processed. Indeed, the ratios between the processing times of BXP and CBXP are 2.21, 2.78, 2.83, 2.89, and 3.56 on the 5 consecutive query intervals containing 100 queries, and the ratio is 2.77 on the whole query interval. As for global performance, CBXP is followed by BXP, TXP, and Arb in this

order. If we set to 1 the time spent by CBXP, then the time consumed by BXP is 2.77, that of TXP is 4.02, and that of Arb is 6.3.

3.2 Experiments on Simulated Real Data

This section contains the results of our experiments on simulated real data. We generated the documents using XMark data generator XMLGen [28]. It generates scalable XML documents simulating an Internet auction website. We generated three documents of increasing size, that we named SmallDoc, MedDoc, and BigDoc. The table below contains the documents' characteristics, where s is the document size in MB (notice that the maximum fanout of the documents is quite different while the average fanout and the depths are constant):

doc	n	s	avgd	maxd	avgf	maxf
SmallDoc	167,864	11.1	4.55	11	3.66	2,550
MedDoc	832,910	55.32	4.55	11	3.66	12,750
BigDoc	1,666,310	111.12	4.55	11	3.67	25,500

As for the query set, we used a fragment of the XPath benchmark XPathMark [27]. Our benchmark consists of 11 queries, each focusing on a different axis, with a natural interpretation with respect to XMark documents. For instance, query Q4 asks for the American items sold in the auction and corresponds the the XPath query /child::site/child::regions/child::*/child::item[parent::namerica or parent::samerica]. See the paper website for the full list of queries.

The query processing time spent by each engine to execute the entire benchmark on the three documents is shown in following table. The columns named ds contain the data scalability factor for the adjacent documents:

Engine	SmallDoc	ds	MedDoc	ds	BigDoc
TXP	0.73	1.01	3.66	0.99	7.28
BXP	1.29	1.02	6.50	0.99	12.93
CBXP	0.72	1.00	3.58	1.00	7.18
Arb	80.84	1.01	404.92	0.93	750.08

TXP and CBXP are the fastest, followed by BXP. Arb is far behind. Hence, in this case, TXP outperforms BXP. The situation was the opposite on synthetic data. This behavior is interesting. While the evaluation strategy encoded in TXP is query-driven, that is, it tries to access only those nodes that will be eventually selected by the query, BXP, CBXP, and Arb strategies are blind in this respect and might visit nodes that will not be part of the solution. XPathMark queries are very selective, that is, their partial and final results are small compared to the document tree size. On the contrary, synthetic queries have large intermediate and final results (almost all the tree nodes are always in these sets). Hence, TXP has a big advantage on selective queries with respect to BXP and Arb. Nevertheless, exploiting the cache optimization, CBXP still competes with TXP on selective queries. An additional cause for the bad performance of Arb on this benchmark is the shape of XMark documents, which tend to have a large

maximum fanout, while the natural data model for Arb is a binary tree. Finally, the data scalability of all engines is essentially linear.

The query processing and response times for each query in the benchmark with respect to MedDoc are depicted in Figure 2. For each query, from left to right, TXP corresponds to the first (red) bar, BXP to the second (green) bar, CBXP to the third (blue) bar, and Arb to the fourth (pink) bar. The relative performance on the other two documents is much similar.

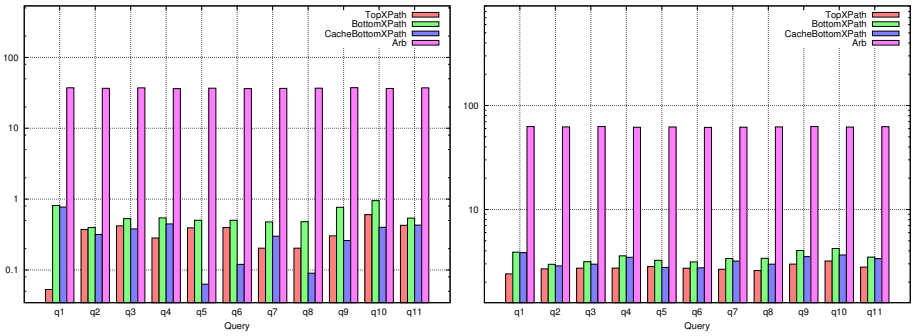

Fig. 2. Query processing (left) and response (right) times on MedDoc (log scale)

Moreover, the table below shows, for each engine, the minimum (min), maximum (max), average (mean), and standard deviation (deviation) of the processing times of the benchmark queries with respect to MedDoc. The ratio between the standard deviation and the mean is given in the last column (stability). This value is an indicator of the stability of the query response times for an engine.

engine	**min**	**max**	**mean**	**deviation**	**stability**
TXP	0.05	0.60	0.37	0.15	41%
BXP	0.40	0.95	0.53	0.17	32%
CBXP	0.06	0.77	0.32	0.20	62%
Arb	36.25	37.37	36.80	0.40	1%

The analysis per query confirms the above hypothesis about the performance of TXP and BXP. While TXP neatly outperforms BXP on highly selective queries like Q1, the performance of the two is comparable on queries with less selectivity like Q2. As expected, Arb is the most stable and TXP is less stable than BXP. CBXP is unstable due to the cold cache. We conjecture that CBXP is very stable if the cache is warm (well populated).

Finally, the effectiveness of the cache is well illustrated in Figure 3. The left plot refers to BXP and the right one is for CBXP. Notice how the cache optimization smooths the peaks of BXP. This action is more effective as the cache warms up.

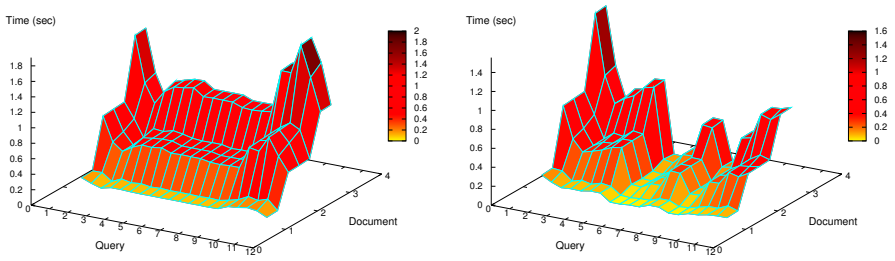

Fig. 3. The effectiveness of the cache in 3D

4 Conclusion

As mentioned in the introduction, many evaluation methods for XPath have been proposed. However, few attempts have been made to compare the performance of these methods. In particular, to the best of out knowledge, this is the first paper that empirically compares top-down and bottom-up methods for XPath. Our general conclusions are the following:

1. The cache optimization is effective and should be definitely integrated in an optimized full-fledged XPath/XQuery evaluator. Of course, a cache maintenance strategy should be adopted.
2. The top-down approach of TopXPath is more efficient than the bottom-up approach of BottomXPath on queries with high selectivity, while the opposite is true on poorly selective queries. Natural queries, like XPathMark ones, tend to be quite selective.
3. The tree automata-based approach implemented in Arb does not scale up with respect to the query length. When the query is relatively small, the approach is efficient and in fact the response times are independent on the query length, as claimed in [12]. However, this does not hold anymore when the size of the query grows.

References

1. World Wide Web Consortium: XML Path Language (XPath) Version 1.0. `http://www.w3.org/TR/xpath` (1999)
2. World Wide Web Consortium: XML Path Language (XPath) Version 2.0. `http://www.w3.org/TR/xpath20` (2005)
3. World Wide Web Consortium: XQuery 1.0: An XML Query Language. `http://www.w3.org/TR/xquery` (2005)
4. Gottlob, G., Koch, C., Pichler, R.: Efficient algorithms for processing XPath queries. In: VLDB. (2002) 95–106
5. Gottlob, G., Koch, C., Pichler, R.: XPath query evaluation: Improving time and space efficiency. In: ICDE. (2003) 379–390

6. Gottlob, G., Koch, C., Pichler, R.: Efficient algorithms for processing XPath queries. ACM Transactions on Database Systems **30** (2005) 444–491
7. Afanasiev, L., Franceschet, M., de Rijke, M., Marx, M.: CTL model checking for processing simple XPath queries. In: TIME. (2004) 117–124
8. Hartel, P.: A trace semantics for positive Core XPath. In: TIME. (2005) 103–112
9. Franceschet, M., Zimuel, E.: Modal logic and navigational XPath: an experimental comparison. In: M4M. (2005) 156–172
10. Neven, F.: Automata theory for XML researchers. SIGMOD Record **31** (2002) 39–46
11. Neven, F., Schwentick, T.: Query automata over finite trees. Theoretical Computer Science **275** (2002) 633–674
12. Koch, C.: Efficient processing of expressive node-selecting queries on XML data in secondary storage: A tree automata-based approach. In: VLDB. (2003) 249–260
13. Al-Khalifa, S., Jagadish, H.V., Koudas, N., Patel, J.M., Srivastava, D., Wu, Y.: Structural joins: A primitive for efficient XML query pattern matching. In: ICDE. (2002) 141–152
14. Bruno, N., Koudas, N., Srivastava, D.: Holistic twig joins: optimal XML pattern matching. In: SIGMOD Conference. (2002) 310–321
15. Grust, T.: Accelerating XPath location steps. In: SIGMOD Conference. (2002) 109–120
16. Fan, W., Park, S., Wang, H., Yu, P.S.: ViST: A dynamic index method for querying XML data by tree structures. In: SIGMOD Conference. (2003) 110–121
17. Moon, B., Rao, P.: PRIX: Indexing and querying XML using Prüfer sequences. In: ICDE. (2004) 288–300
18. Moro, M.M., Tsotras, V.J., Vagena, Z.: Tree-pattern queries on a lightweight XML processor. In: VLDB. (2005) 205–216
19. Chen, C., Hsu, W., Li, H., Lee, M.L.: An evaluation of XML indexes for structural join. SIGMOD Record **33** (2004) 28–33
20. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (1999)
21. Marx, M.: Conditional XPath, the first order complete XPath dialect. In: PODS. (2004) 13–22
22. Blackburn, P., de Rijke, M., Venema, Y.: Modal Logic. Cambridge University Press (2001)
23. Hsu, W., Lee, M.L., Yang, L.H.: Efficient mining of XML query patterns for caching. In: VLDB. (2003) 69–80
24. Afanasiev, L., Franceschet, M., Marx, M., Zimuel, E.: XCheck: A platform for benchmarking XQuery engines (demonstration). In: VLDB. (2006) `http://ilps.science.uva.nl/Resources/XCheck`.
25. Hooker, J.N.: Testing heuristics: We have it all wrong. Journal of Heuristics **1** (1996) 33–42
26. Afanasiev, L., Manolescu, I., Michiels, P.: MemBeR: a micro-benchmark repository for XQuery. In: XSym. Volume 3671 of LNCS. (2005) 144–161
27. Franceschet, M.: XPathMark: an XPath benchmark for XMark generated data. In: XSym. Volume 3671 of LNCS. (2005) 129–143 `http://www.science.uva.nl/~francesc/xpathmark`.
28. Schmidt, A., Waas, F., Kersten, M.L., Carey, M.J., Manolescu, I., Busse, R.: XMark: A benchmark for XML data management. In: VLDB. (2002) 974–985 `http://www.xml-benchmark.org`.