Pattern Tree Matching for XML Queries

Vincent Aguiléra Frédéric Boiscuvier Sophie Cluet Bruno Koechlin
vincent.aguilera@inria.fr *firstname.lastname*@xyleme.com

1 Introduction

The main paradigm underlying a query language for XML such as X-Query [8] is that of selecting data by matching tree patterns made of path expressions. The path expressions describe traversals along sub-elements with variables getting bound to some nodes along these paths. Hence, a key operation in query processing over XML is, given a tree pattern, to produce a set of bindings for the pattern variables. In this paper, we present an implementation of this operation whose complexity, when the documents do not feature recursive elements, is linear in the number of selected data items.

We are interested by large scale applications with many users where efficiency is a key factor. Although XML is still relatively a newcomer, these applications are growing in business areas such as content providers or big corporations. Also, as more and more XML documents will be available online, there will be a strong need for a Web scale XML search engine. An interesting aspect of the applications we are targeting is that they also require that first answers be returned fast.

The evaluation of structured queries over loosely structured data such as XML has been extensively studied during the last ten years [5]. The techniques differ slightly according to the system used to store the data: object-oriented (e.g., [4]), dedicated (e.g., [13]) or relational (e.g., [11]). Still, in all these works, pattern trees are matched by navigating within the documents, eventually with the help of some dedicated indexes. When nodes within the pattern tree are linked by an ascendant (in opposition to parent) relationship, this simply does not scale. Another approach is used in [15]. The authors create one relation per path in the documents structures and match patterns by performing joins between these relations. This approach is well adapted to applications with a fixed schema. However, it does not scale well (too many relations, difficult updates) when there are many different DTDs to consider or when the structure of the documents tends to change, even slightly (note that a large part of the popularity of XML is its flexibility). The implementation that we present in this paper relies on an indexing mechanism similar to that of [12]. A full text index associates to each tag or word of the database the documents in which it occurs, as well as its relative positions within these documents. The authors use this index to evaluate linear path expressions by performing joins between posting lists. We propose an extension of this technique to full pattern trees, with a special focus on the join algorithms and ordering problem that it raises.

As stated before, we are interested by large scale XML applications with many users and the need to get first answers fast. Together, these three characteristics reduce the solution scope. Indeed, efficiency and large scale somehow imply in-memory indexes, many users forbid the use of large intermediate structures, and fast first answers means pipeline evaluation. We propose an implementation of the pattern tree matching operator relying on merge joins and an adaptation of the full text index technology. Our original contribution is threefold.

- We implemented a full text index mechanism that stores the relative position of words within their documents. As in [12], the position is calculated using Dietz's numbering scheme. We present a way to encode these numbers that seriously reduces the memory consumption of the index structure. As a matter of fact, while allowing to support more complex structured queries, our index has a size similar or smaller to that commonly attributed to full inverted indexes [16].

- Using this index, a pattern tree matching operation is equivalent to a sequence of joins among posting lists. We propose an adaptation of the merge join algorithm to deal with ordered posting lists. Also, we study the join ordering problem in this particular context with a special attention to duplicate elimination.

- Finally, measures show the effectiveness of our approach that generally improves previous implementations by a factor larger than 10.

The paper is organized as follows. In Section 2, we define the pattern tree matching operator and show how it is used to implement X-Queries. Section 3 presents the indexing mechanism and Section 4 the pattern tree matching implementation. Finally, in Section 5, we measure our implementation and compare it with previous ones.

2 Preliminaries

X-Query [8] is becoming widely accepted as the standard for querying XML documents. As most of its predecessors for semistructured or XML data (e.g., [4, 6, 7, 9, 10]), it provides the ability to query documents by pattern tree matching. For instance, consider Query **Q4** from the XMark [3] benchmark.

FOR	$b IN document("auction.xml")/site/open_auctions/open_auction
WHERE	$b/bidder/personref[id="person18829"]
	BEFORE
	$b/bidder/personref[id="person10487"]
RETURN	<history> $b/reserve/text() </history>

The query lists the reserves of those open auctions where a certain person issued a bid before another person. Figure 1 shows the pattern tree that one has to match against the document named *auction.xml* to find out the elements that constitute this list.

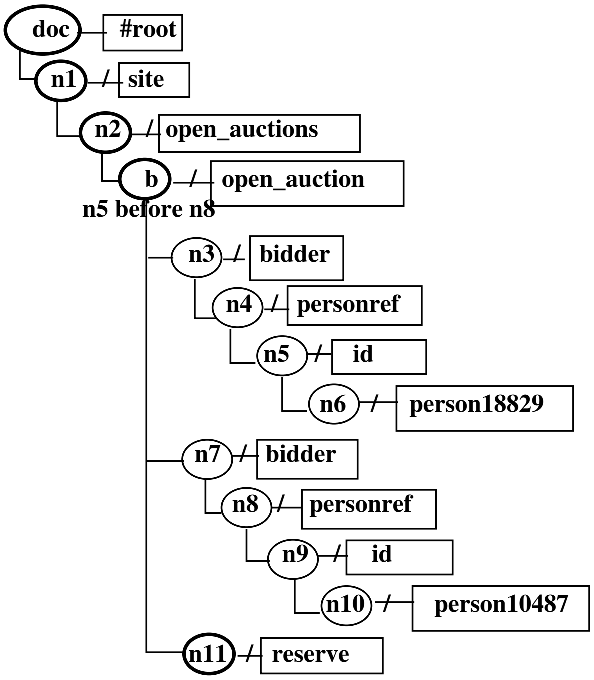


Figure 1: The Pattern Tree Corresponding to Query Q4

Each node of the tree is labelled with (i) a variable (e.g., **b**) and (ii) either a word (on leaves only, e.g., *person10487*) or the name of an element/attribute (e.g., *bidder* and *id*). Additionally, a bold circle around a node variable indicates that it corresponds to a join rather than a semi-join. This distinction is needed to capture the two semantics of path expressions.

- When they occur in **RETURN** or **FOR** clauses, path expressions have a bag semantics. I.e., we must consider all possible combinations of elements that match them. This corresponds to join operations. Similarly, when several variables are defined in the **FOR** clause, joins are used to connect them.

- In a **WHERE** clause, path expressions have a set semantics. I.e., we are interested in finding only the first combination that satisfies it. This corresponds to semi-joins.

The edges of the tree are labeled with the relationship that exists between the nodes: either parent-child (denoted by a simple '/') or ascendant-descendant ('//'). The example features only the former.

Finally, simple selections can be added to be evaluated within the matching process (e.g., *n5 before n8*).

In this paper, we present the implementation of an operator for matching pattern trees as shown in Figure 1 against a list of documents. This operator, called **patternScan**, is now defined.

2.1 Definition of the PatternScan Operator

The **PatternScan** operator filters a forest of XML trees according to a given pattern and returns a set with one tuple per matching tree structure found somewhere in the forest.

XML Trees

To simplify the presentation, we model XML documents as *rooted ordered node-labeled trees* where nodes represent indifferently elements, attributes or text. In the actual implementation, we do the distinction. We call an *XML tree* a tree that models an XML document.

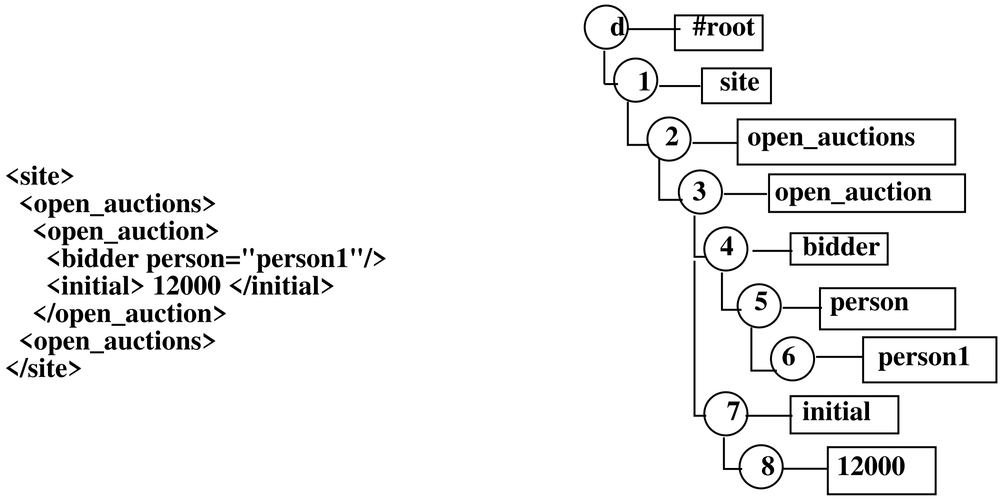


Figure 2: An XML document d and the corresponding XML tree

Figure 2 shows such a tree. Given a forest F of XML trees, we label the root node of each XML tree in F with a special label denoted as *#root*, distinct from any other label in F.

Pattern Trees

Let $T = (N, E, R_T)$ be a XML tree rooted at R_T, where N is the set of nodes in T and E its set of edges. We derive a *pattern tree* P from T by adding two functions :

- $join$ from N to $\{true, false\}$ to distinguish join from semi-join nodes.

- $pred$ from E to the set $\{parentOf, ascendantOf\}$ to capture the relationship between nodes.

Pattern Tree Matching

Let $P = (N_p, E_p, R_P, pred, join)$ be a pattern tree and $T = (N_t, E_t, R_T)$ be an XML tree.

An injective function m from N_p to N_t is a *match* between P and T iff it preserves labelling and *pred*-relationship. That is, for all nodes n_1 and n_2 of N_p we require that :

- $label(n_1) = label(m(n_1))$,
- $pred(n_1, n_2) = parentOf$ if and only if $m(n_1)$ is a parent of $m(n_2)$ in T.
- $pred(n_1, n_2) = descendantOf$ if and only if $m(n_1)$ is an ascendant of $m(n_2)$ in T.

Let m be a match between P and T. We say that we *apply* m to T by defining the following set of tuples :

$$apply(m, P, T) = \{[n : m(n)], n \in N_p, join(n) = true\}$$

PatternScan

Let $Match$ be the set of matches between P and T. The result of the $PatternScan$ operator applied to P and T is defined by :

$$PatternScan(P, T) = \{apply(m, P, T), m \in Match\}$$

Finally, let F be a finite forest of XML trees. Then :

$$PatternScan(P, F) = \bigcup_{t \in F} PatternScan(P, t)$$

3 Indexing Mechanism

Our implementation of the **PatternScan** operator relies on an indexing mechanism that we present in this section.

Words Encoding

Some full text indexes (called full inverted indexes) store the offset associated to each word occurrence (or posting) in a document in order to answer queries about words proximity. For instance, queries such as: “bidder” **near** “person”.

We improve this mechanism so as to answer structured queries such as “person” is an attribute whose value contains “person1” or “bidder” is an ancestor element of “person1”. For this, as in [12], we encode words using the Dietz’s numbering scheme. More precisely, we associate to each word, its position in the document relatively to its ancestor and descendant nodes. This position is encoded by three numbers that we call pre-order, post-order and level. This is illustrated in Figure 3.

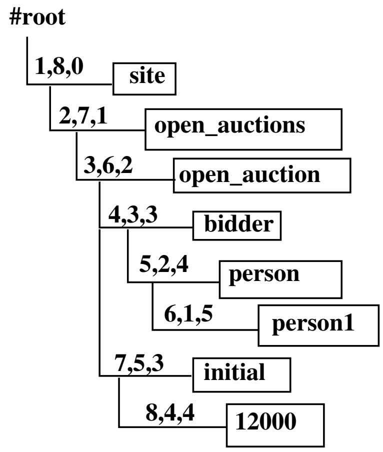


Figure 3: Pre-order, Post-order and Level Numbering

Given an XML tree T, the pre (resp. post) order number of nodes in T are assigned according to a left-deep traversal of T. Every time we enter (resp. exit) a node, we increment the pre (resp. post) counter and assign its value to the current node. The level number is the number of ancestors of the current node. Given two nodes n and m in T, the two following properties stand :

- n is an ancestor of m if and only if $pre(n) < pre(m)$ and $post(m) > post(n)$
- n is the parent of m if and only if n is an ancestor of m and $level(n) = level(m) - 1$

Note that an equivalent alternative would be to record, for each element/attribute, the offsets of its first and last leaf words.

As we will see in the following sections, this way of indexing XML documents is very efficient for pattern matching. Indeed, in most cases, the evaluation of a *PatternScan* operation requires a single scan over the posting lists corresponding to the pattern nodes. But it has other advantages.

- Since it requires no previous knowledge of the documents structure, it is particularly suited to the XML applications we are considering where DTDs, when they exist, tend to change with time or authors and where documents are fairly heterogeneous.
- By ordering each posting list of the index by first document identifier then pre-order number, we maintain the natural order of the database. This is good for several reasons:
  - we preserve the original order of the elements when computing a query,
  - we support positional predicates such as **before** or **after** (a simple comparison between two pre-order numbers), and

– it is easy to have a fast access to a tree node given its pre-order number. Indeed, if the documents are stored in a native XML database with a DOM-like interface, one just has to store the number of descendants of each node to support retrieval with an average complexity of log(n).

Space Consumption

We are targeting large scale, many users applications where efficiency is a key factor. Typical customers for these applications are content providers or corporations with large intranets. Web scale XML search engines also belong to this kind of applications. To provide suitable response time, it is important that the indexes remain in main memory. Thus, space consumption is an important issue.

It is generally admitted that the best implementations of full text indexes that have posting lists consisting only of documents identifiers require 10 to 20% of the indexed database size. By simply adding offsets, full inverted indexes generally use up to 60% of the indexed data [16]. This ratio can be improved by removing commonly used words (called stopwords). These numbers show what a naive implementation of the (pre-order, post-order, level) numbering can lead to in terms of space consumption.

Our implementation of the index mechanism stems from an observation that we made during experiments: the difference between pre-order and post-order numbers is usually very small. By further studying this characteristic, we found out the following property (see proof in Appendix).

Given any XML tree, the following equation is true:

post-order(n) - pre-order(n) + level(n) + 1 = card(n)

where card(n) is the number of nodes in the subtree rooted at n.

We exploit this property in the following way. Since card(n) equals 1 for leaf nodes, two numbers allow to compute the third. Because pre-order is used as a sort key for the index and level is small, they are the two numbers we store. For internal nodes, we store pre-order, level and the difference between pre and post-order. Note that the lower is the node in the tree, the smaller is the difference and thus the size of the encoding. Also, leaf nodes constitute the bulk part of an XML tree. This is why, in most cases, two (resp. three) bytes are enough to encode words relative position in a small (resp. large document). An extra byte is used to store information about the size of the encoding (if the three bytes are not enough), and the type of the word (whether attribute, element or simple word).

With this encoding, the main memory representation of the index is about 50% of the database size. In Section 5, we present measures that we made using the XMark benchmark [3] on four databases whose sizes range from 110MB to 1.1GB. The XMark documents are large. The size of the index RAM representation for the smaller (resp. larger) database is 57MB (resp. 536MB) (we did not stem words).

We are currently working on further decreasing the size of our indexes. One idea that we are considering is to put the three numbers in different arrays so as to exploit their similarities to better compress them. Consider for example the level information associated to the tag *open_auction*. The chances are that this information is the same for all occurrences of the tag. This can easily be exploited by a compression mechanism to shrink the associated posting list.

4 Pattern Scan Implementation

In Section 2, we gave a definition of the **PatternScan** operator. In order to implement it efficiently, we rely on an equivalent algebraic definition that we now give.

4.1 PatternScan Algebraic Definition

We first need to define the *extent* of a label in a forest of XML trees.

Label extent: Let F be a forest of XML trees, and l be a node label in F. We call the *extent* of l in F, and denote $ext(l, F)$, the set of nodes in F labelled l. If a is a pattern tree node, we denote $ext(a, F)$ the set of tuples defined as follows :

$$ext(a, F) = \{[a : n], n \in ext(label(a), F)\}$$

To link this definition with the index presented in the previous section, note that $ext(l, F)$ corresponds to the posting list of l in the index over F.

PatternScan: Let $P = (N, E, R_P, pred, join)$ be a pattern tree and $k = Card(N)$. We say that a one-to-one mapping m between N and $[1; k]$ is a *numbering* of P if m is such that for every nodes n and n' in P, $m(n) < m(n')$ iff n is an ancestor of n'. We denote n_i the node in N such that $m(n) = i$, for i in $[1; k]$.

We are now ready to define the semantics of the *PatternScan*.

$$PatternScan(P, F)$$

$$=$$

$$\prod_{n \in N, join(n)} (ext(_1, F) \bowtie_{pred(n_{<2}, n_2)} ext(n_2, F) \ldots \bowtie_{pred(n_{<k}, n_k)} ext(n_k, F))$$

where $n_{<i}$ is the parent of n_i in the pattern tree.

Figure 4 shows a pattern tree and its algebraic translation.

The algebra is based on sets, in opposition to bags (in fact, ordered sets as we will see in the sequel). Since, all nodes have distinct identifiers, the joins do not generate duplicates. This is

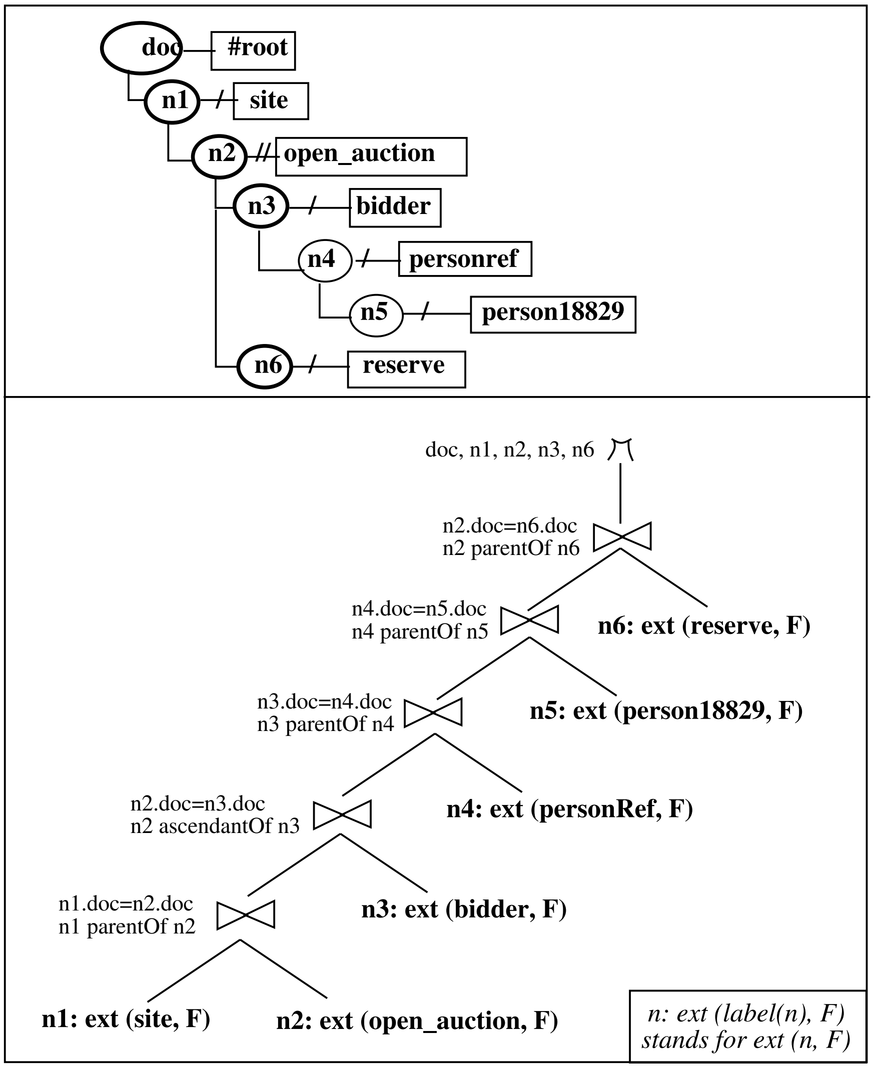


Figure 4: A PatternScan Operation and its Algebraic Translation

not the case of the projection operation that has to perform duplicate elimination. In fact, this operation is mainly here to take into account the set semantics of the path expressions occurring in the **WHERE** clause. Only the variables corresponding to these paths are projected out.

Let us now concentrate on two problems: (i) which join algorithms should we use and (iii) what is the best joins sequence.

4.2 Join Algorithm

A large number of join algorithms have been proposed in the literature. However, many of them do not fit our needs. Remember that we are targeting applications with many concurrent queries and requiring that first answers be returned fast. Furthermore, also consider that the result of a pattern tree query can be exponential in the size of the pattern. For these reasons we cannot consider join algorithms involving intermediate data structures, especially if the size of those structures depends on the size of the result. These constraints leave mainly two algorithms : nested-loop-join and merge-join. Since the index posting lists are sorted, we obviously favour a merge like algorithm.

Let us now illustrate how the following algebraic expression can be evaluated:

Expression 1: $ext(F, a) \bowtie_{\text{ascendantOf}} ext(F, b)$

The first step is to retrieve the posting lists l_a and l_b associated to labels a and b. Since they are sorted by tree identifier then by pre-order number, the results of the join can be obtained by a two-levels sort merge operation without additional cost of sorting. In the first level, l_a and l_b are merged by tree identifiers. Then, in the second one, given a tree identifier, we merge the postings by examining their ancestor-descendant relationship.

A merge-like algorithm is also used in [12] to evaluate linear path expressions. We do more than simply extending this technique to tree queries. We have just seen that a merge-join is adapted at the forest level (i.e., the first stage of the above informal algorithm). As we will see in the sequel, the merge operation at the tree level raises interesting problems that have to be seriously considered. Notably, it may entail re-readings of the same posting list (a nested loop within the main merge loop) that (i) are very expensive when the documents are large and (ii) destroy an order necessary to the next join and projection operations. In [12], the authors state :

Despite the fact that an element set may have to be scanned multiple times by the inner **foreach** *loop, ... the join algorithm is still highly effective.*

The measures in Section 5 show that we do 10 times better.

We now study the tree level join-merge algorithm.

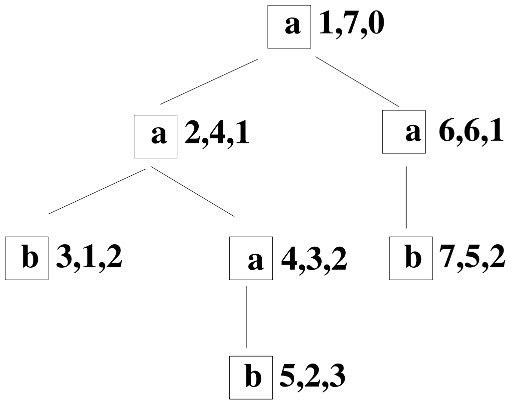
The ADJoin algorithm

Let us first illustrate how the merge-join algorithm can be adapted to evaluate Expression 1. This is the purpose of the *ADJoin* (*AD* stands for Ancestor-Descendant) given in Figure 6 at the end of this paper. It takes as arguments an index I, a tree identifier T and two node labels a and b. Its output is a list of couples of nodes (n_1, n_2) in T such that $label(n_1) = a$, $label(n_2) = b$ and n_1 is an ancestor of n_2.

For instance, Figure 5 shows a simple tree on the left-hand side and the output corresponding to the ADJoin between the posting lists of a and b.

The algorithm starts with an initialization phase where variables *outer* (resp. *inner*), points to the head of the posting list $ext_I(a, T)$ (resp. $ext_I(b, T)$). One can note that given an element o in the outer list (we will use (4,3,2) as an example), the *inner* list can be decomposed into three consecutive sublists :

- The *preMatch* list is composed of elements i such that $pre(i) \leq pre(o)$ (e.g., (3,1,2)).
- The *match* list is composed of elements i such that $pre(o) < pre(i)$ and $post(i) < post(o)$, i.e. o is an ancestor of i (e.g., (5,2,3)).



[a: (1,7,0), b: (3,1,2)]

[a: (1,7,0), b: (5,2,3)]

[a: (1,7,0), b: (7,5,2)]

[a: (2,4,1), b: (3,1,2)]

[a: (2,4,1), b: (5,2,3)]

[a: (6,6,1), b: (7,5,2)]

Figure 5: An XML Tree (with recursion) and the Result of an ADJoin

- The $postMatch$ list is composed of elements that are not in $preMatch$ or $match$ (e.g, (7,5,2)).

Since *outer* and *inner* are ordered by pre-order numbers, we have the following property : given two successive outer elements o_1 and o_2, if o_1 is an ancestor of o_2 then $match(o_2)$ is strictly included in $match(o_1)$. Otherwise, $match(o_2)$ follows $match(o_1)$. This can easily be checked in the example.

The algorithm uses this property to advance in the two collections and avoid multiple scanning of the inner one (as would be the case if we used a nested-loop join). Once an outer is chosen, we advance in the inner list until we find its match zone (**while** loop starting on Line 8 in Figure 6). Then, while we stay in this zone, we produce results (**while** loop starting on Line 19). Finally, we check the next outer. If it is a descendant of the current one, we re-read the previous match zone (Instruction 28). Otherwise, we step forward in the inner collection. In the example, re-reads occurred for the second and third a postings.

The $PCJoin$ algorithm that evaluates joins with ParentOf predicate is not given here since it is similar to $ADJoin$. It simply adds a test to check levels.

So, whenever we want to join two posting lists, the $ADJoin$ and $PCJoin$ behave better in any case than the nested-loop-join. Furthermore, assuming that recursive elements are rare, in most cases we simply scan once inner and outer lists.

Bad Predicates

Both $ADJoin$ and $PCJoin$ work in a top-down way relatively to the pattern tree: i.e., the outer element is above the inner one. An alternative implementation swaps inner and outer. This apparently minor modification has in fact very bad consequences. For instance, consider again the XML tree in Figure 5 and assume that we perform a join according to b $descendantOf$ a (i.e., b is outer). Note that the match zone associated to the last b posting (i.e., (7,5,2)) should include at least (1,7,0) and (6,6,1). Since postings are sorted by pre-order, this means that the match zone of the last b is in fact the full a posting list. In other words, we have to re-read the inner collection from the start. Some measurements we made show that this can increase response time by a factor of 10^4, even with little recursion.

To summarize, the problem with $descendantOf$ and $childOf$ predicates is that their match zone is not "pure" but contains elements that do not match. Note that this is true only for recursive elements. Hence, we could use these predicates when we know there is no recursion. However, this is difficult to check beforehand on all documents. Indeed, although recursion is rare in documents, it is in fact fairly common in DTDs (consider standards such as DocBook [1], XBRL [2], ...).

Intermediate joins

The $ADJoin$ algorithm heavily relies on the assumption that its outer and inner collections are ordered. This is always true when joining two posting lists, but it may not be the case for intermediate results. For instance, in Figure 5, the result is not ordered on b.

In general, if no re-reads are performed, an $ADJoin$ returns a set ordered by both *inner* and *outer*. However, if re-reads are necessary, the result is ordered first by *outer* then by *inner*, i.e. *inner* by themselves are not ordered. To deal with unordered elements we modified the *ADJoin* algorithm as follows : whenever *outer* is iterated (lines 15 and 30 in Figure 6) if an *outer* has a smaller prefix than its predecessor then we rewind *inner* up to the beginning.

4.3 The Join Ordering Problem

We now consider the best way to order the sequence of joins corresponding to a *PatternScan* operation. Traditionally, this is achieved with the help of a good cost model and by applying dynamic programming. We do not have a good cost model and we believe it is very hard to come by when considering applications whose data is very heterogeneous and not strictly typed. Furthermore, the fact that we want to avoid intermediate data structures seriously reduces the solution scope.

Our join ordering is based on the following assumption : ideally, the *PatternScan* operation should be implemented by a sequence of merge joins (i) with no re-reads on computed collections, (ii) and such that the sequence output order can be exploited to evaluate the final projection operation.

Let us explain points (i) and (ii).

We have seen in 4.2 that any join in the sequence can be implemented using either $ADJoin$ or $PCJoin$. In these algorithms, re-reads can occur on the inner collection. This means that, if we allow the inner collection to be itself a sequence of joins, and since we do not store intermediate results, we may have to recompute the inner collection up to the re-read position each time a re-read is needed. This is obviously expensive and motivates (i). It also leads to considering only left-deep trees.

Again, (ii) stems from our rejection of intermediate data structures. Another reason is our need to return first answers fast. This forbids a sort operation. Hence, join results should be conveniently sorted if we are to eliminate the duplicates generated by the projection operation. Remember that this is important if we want to preserve the semantics of the query. Since (i) reduces the solution scope to left-deep trees, the goal is now to find a numbering of the pattern tree with respect to (ii).

Because of the predicates we support (see bad predicates above), the numbering of the pattern tree is performed top-down (i.e., according to the numbering definition given at the beginning of this section). It requires two traversals of the tree.

During the first pass, we enforce (ii) by numbering join nodes first. More precisely, we number all join nodes and join nodes only. Note that this is possible only because, given the way XQuery are translated into the algebra, all the ancestors of a join node are join nodes. When we have a choice between two nodes, we favour that with the smallest posting list. For instance, consider the pattern tree of Figure 4. **n1** and **n2** are numbered respectively 1 and 2. Then, we have a choice between **n3** and **n6**. If there are less *reserve*'s than *bidder*'s, then **n6** will be numbered 3.

During the second pass, we take care of the remaining nodes applying the same heuristics as before when a choice has to be made between several nodes.

Obviously, this plan generation algorithm does not always lead to the best solution. However, the experimental results in the next section show that it is very good in most cases and that it does not perform too bad in worst cases. Furthermore, the need to return first answers fast prevents us from using more sophisticated techniques.

5 Experimental Results

Our implementation of the *PatternScan* is written in C++, and is part of a large distributed system designed to store and query massive amount of XML data. The system runs on top of a cluster of standard PCs.

The goal of our experiments was to study the response time of a single index machine for varying queries and data size. Experiments where performed on a 866 Mhz Pentium III machine operating Linux, with 1.5 GBytes of memory. This section gives informations concerning the data sets and queries we used, and then provides some commented figures.

5.1 Data Sets

We used one data set (SP) from a real word application, *The plays of Shakespeare in XML* [1] and four synthetic data sets (F01, F02, F05, F10) generated using the XMark data generator, namely *xmlgen*. The characteristics of those data sets are given in table 1, among with the size of the associated index. One can notice that documents in synthetic data sets are quite large, around 100 MBytes each. As we will see, this can be particularly stressing for the *PatternScan* operator. Notably, when a pattern tree leaf node has a tiny posting list while the path that leads to it is not selective (see Query Q04).

[1] available at `http://www.oasis-open.org/cover/bosakShakespeare200.html`

Data set	SP	F01	F02	F05	F10
Number of documents	37	1	2	7	17
Size in MByte	7.9	111	223	559	1100
Number of elements	115k	1.27M	2.5M	6.4M	12.3M
Number of attributes	0	380k	762k	1.9M	3.7M
Number of words	835k	11.4M	22.8M	56.7M	107M
Index size in MByte	5	57	116	276	536

Table 1: Characteristics of data sets used for experiments

5.2 Queries and Methodology

The SP data set is certainly the most frequently set of XML documents used for test purpose. In order to provide a performance metric that can ease comparison with other systems we propose five queries, detailed in 5.3.2. QS1 is used in [12]. QS2 to QS5 come from [14]. All of them can be evaluated using a PatternScan. They vary in terms of complexity and selectivity, i.e the number of results ranges from 35 to approximately 30k.

The XMark benchmark proposes 20 queries [2] that have been designed to test various aspects of a complete XML database system. As a consequence not all of them are meaningful for our tests. For instance the aim of Q18 is to test the ability of a system to handle user defined functions. This is clearly not a pattern matching problem. So we chose to evaluate those for which a single *PatternScan* operation can be used to retrieve all the information needed before (eventually) accessing the store to retrieve the resulting data. A PatternScan may return exactly the result of a query. This is the case for query Q6 and Q7.

Complex queries such as Q8, Q9, Q10 or Q11 involve joins between pattern scans, we ignore them.

For those familiar with the XMark benchmark [3], table 2 summarizes which query subset was used.

For each query, we measured (i) the time for the first result, (ii) the total evaluation time and (iii) the number of results.

5.3 Results

5.3.1 Shakespeare's Plays

Results for queries over the SP data set (see 5.3.2) are given in Figure 7 at the end of this paper. From top left to bottom, we find the graphs relative to number of results, evaluation time, time for the first answer and average time by result. The total evaluation time for QS1 compares favorably to the figures given in [12](query evaluated on a UltraSparc-II WS with 256MBytes for a 8MBytes database), 44 ms versus 500 ms.

[2]They can be found at `http://www.xml-benchmark.com`

Query	Used for experiments	Need additional predicate evaluation on the store	Needs accessing the store to build the results
Q01	yes	no	yes
Q02	yes	no	yes
Q03	yes, same pattern than Q2, i.e., same numbers.	first bidder increase multiplied by 2 smaller than the last	yes
Q04	yes	no	yes
Q05	yes	price > 40	yes
Q06	yes	no	no
Q07	yes. Involves 3 PatternScans (Q07.1, Q07.2 and Q07.3).	no	no
Q08	no		
Q09	no		
Q10	no		
Q11	no		
Q12	no		
Q13	yes	no	yes
Q14	yes	no	yes
Q15	yes	no	yes
Q16	no		
Q17	yes	no	yes
Q18	no		
Q19	no		
Q20	no		

Table 2: XMark queries subset used in our experiments

The most expensive query is QS2. It starts by joining posting lists *SPEECH* and *LINE*, and then joins the results with the posting list *hate*. The first join is quite expensive and unnecessary since (i) there are approximately 30k *SPEECH* elements and 107k *LINE* element in SP and (ii) every *LINE* happens to be child of a *SPEECH*. In this case we could benefit from schema information (either by using DTD or a dataguide) and remove unnecessary joins. By using this information, the evaluation time shrinks to 14ms.

5.3.2 XMark Queries

The corresponding results are given in Figure 8 at the end of this paper. The graph on the top left shows that the total evaluation time grows linearly with the size of the data set. Due to the characteristics of the generated documents, the number of results is also linear in the size of the database (see top right graph, we removed Q01 that has always one answer). The graph on the middle right shows that, given a query, the average time per result does not depend on the size of the data set.

We also compared our results with those given for the Monet XML system. The figures for Monet[3] were computed on a machine equipped with 1 GB of main memory and a Pentium III Katmai CPU clocked at 550 MHz over a data set of 100 MBytes.

On the middle left part of Figure 8 we show the Monet results and that of our system over the F01 and F10 data sets. Note that the data set used by Monet corresponds to F01. One can see that in most cases our results are better by a factor of ten to a hundred. For some queries, this is even true when we compare our F10 to their F01. Although Monet computes the full queries whereas we only retrieve the identifiers of the elements that constitute their results, these numbers are very good. Indeed, the operations that remain to be executed are performed on 0.01% to 1% of the data set. Given the RAM size and the fact that Monet uses a main memory database kernel, this should not be the most costly part.

An exception to these good performances concerns Query Q4. The problem here is similar to that of QS2 (see above): the leaf node of the pattern is very selective whereas the internal nodes are not. Since we do the joins going down, we perform a lot of unnecessary computation. However, note that this represents the most stressful case for our implementation and that we only triple their time. Furthermore, remember that we emphasize the need to return first results fast. As we can see in the lower part of Figure 8, the first result for Q4 still occurs 3 times faster than the Monet answer.

Appendix A: Property P

Given any XML tree, the following equation is true:

post-order(n) - pre-order(n) + level(n) + 1 = card(n)

where card(n) is the number of nodes in the subtree rooted at n.

The proof is by induction.

Let t_0 be a tree with one node n. Then,

post-order(n)=pre-order(n)=level(n)=0 and card(n)=1.

Thus t_0 satisfies P.

Let t be a tree with n nodes and that satisfies P. Let l be one of its leaves. We add a child f' to f. We call respectively pre-order$_f$, post-order$_f$, and level$_f$ the pre-order, post-order and level functions before the insertion of f'. We re-number the nodes in the following way:

The new leaf f':

- pre-order(f') = pre-order$_f$(f) + 1
- post-order(f') = post-order$_f$(f)
- level(f') = level$_f$(f) + 1
- card(f') = 1

Thus, post-order(f') - pre-order(f') + level(f') + 1 = post-order$_f$(f) - (pre-order$_f$(f) + 1) + (level$_f$(f) + 1) + 1 = 1 = card(f').

The old leaf f:

- pre-order(f) = pre-order$_f$(f)
- post-order(f) = post-order$_f$(f) + 1
- level(f) = level$_f$(f)
- card(f) = 2

Thus, post-order(f) - pre-order(f) + level(f) + 1 = (post-order$_f$(f) + 1) - (pre-order$_f$(f)) + (level$_f$(f)) + 1 = 2 = card(f).

The nodes of t whose numbers have changed:

The nodes of t whose numbers have changed are those that follow f in post-order. Let T be that set of nodes. We can partition it in two distinct sets: the ancestors of f (T_1) and the others (T_2).

- For all nodes n in T, post-order(n) = post-order$_f$(n) + 1;
- For all nodes n in T_1, card(n) = card$_f$(n) + 1, level and pre-order remain unchanged.
- For all node n in T_2, pre-order(n) = pre-order$_f$(n) + 1, level and card remain unchanged.

Thus, P remains true for all nodes.

Appendix B: Shakespeare Plays Queries

QS1 find all speech in acts

```
FOR    $s in SP//ACT//SPEECH
RETURN $s
```

QS2 Find all speech having a line containing the word 'hate'.

```
FOR    $s in SP//SPEECH
WHERE  CONTAINS($s//LINE,'hate')
RETURN $s
```

QS3 Find all the lines in The Tragedy of Antony and Cleopatra.

```
FOR    $p in SP/PLAY
WHERE  $p/TITLE/text()='The Tragedy of Antony and Cleopatra'
RETURN $p/ACT//SPEECH/LINE
```

QS4 Find all the lines in a speech whose speaker is Mark Antony.

```
FOR    $s in SP/PLAY/ACT//SPEECH
WHERE  $s/SPEAKER/text()='MARK ANTONY'
RETURN $s/LINE
```

QS5 Find all personae in The tragedy of Antony and Cleopatra.

```
FOR    $p in SP/PLAY
WHERE  $p/TITLE/text()='The Tragedy of Antony and Cleopatra'
RETURN $p//PERSONA
```

References

[1] The docbook dtd.
Available from `http://www.docbook.org/`.

[2] The xbrl dtd.
Available from `http://www.xbrl.org/`.

[3] The xml benchmark project.
Available from `http://www.xml-benchmark.org/`.

[4] S. Abiteboul, S. Cluet, V. Christophides, T.Milo, G. Moerkotte, and J. Simeon. Querying documents in object databases. *International Journal on Digital Libraries*, 1, 1997.

[5] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the web: from relations to semistructured data and XML.* Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, 1999.

[6] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.

[7] Vassilis Christophides, Sophie Cluet, and Jérôme Siméon. On wrapping query languages and efficient XML integration. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, Dallas, Texas, May 2000.

[8] Stephen DeRose. Xquery: A unified syntax for linking and querying general xml documents. Technical report, http://www.w3.org/TandS/QL/QL98/pp/xquery.html, 8 October 1998.

[9] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for XML. *Computer Networks (Amsterdam, Netherlands: 1999)*, 31(11–16):1155–1169, May 1999.

[10] Mary F. Fernandez, Daniela Florescu, Alon Y. Levy, and Dan Suciu. A query-language and processor for a web site management system. In *Workshop on Management of Semistructured Data*, Tucson, Arizona, May 1997. In conjunction with PODS/SIGMOD.

[11] Daniela Florescu, Ioana Manolescu, and Donald Kossmann. Integrating keyword search into xml query processing. In *Proceedings of the 9th World Wide Web Conference*, Amsterdam, May 2000.

[12] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. 2001.

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

[14] F. Rizzolo and A. Mendelzon. Indexing XML Data with ToXin. 2001.

[15] A. Schmidt, M. Kersten, M. Windhouwer, and F. Wass. Efficient Relational Storage and Retrieval of XML Documents. 2000.

[16] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes.* Morgan Kaufmann Publishers.

ADJoin algorithm	
Input:	a XyIndex I, a tree identifier T, two node labels a and b
Output:	a list of couples of nodes (n_1, n_2) in T such that $label(n_1) = a$, $label(n_2) = b$ and n_1 is an ancestor of n_2

```
   // initialization
 1 outer ← ext_I(T, a)
 2 inner ← ext_I(T, b)
 3 innerReread ← nil
 4 if(inner =nil) {
      // no inner
 5    STOP
 6 }

 7 while(outer ≠ nil){
 8   while(inner preMatch outer){
       // Iterate inner while inner preMatch outer.
       // After this loop, inner points to
       // a candidate descendant node of outer.
 9     inner ← next(inner)
10     if(inner = nil){
11       STOP
12     }
13   }
14   if(not(inner match outer)){
        // If outer is not an ancestor of inner, iterate outer.
15     outer ← next(outer)
16   }
17   else {
18     innerReread ← inner
19     while(inner match outer){
          // Iterate inner in the match zone and produce results.
20       output(outer, inner)
21       inner ← next(inner)
22       if(inner = nil){
23         GOTO 27
24       }
25     }
26   }
27    if(next(outer) ≠nil and next(outer) match outer){
      // Next outer is a descendant of current outer => rewind inner.
28     inner ← innerReread
29   }
30   outer ← next(outer)
31 }
```

Figure 6: The ADJoin algorithm

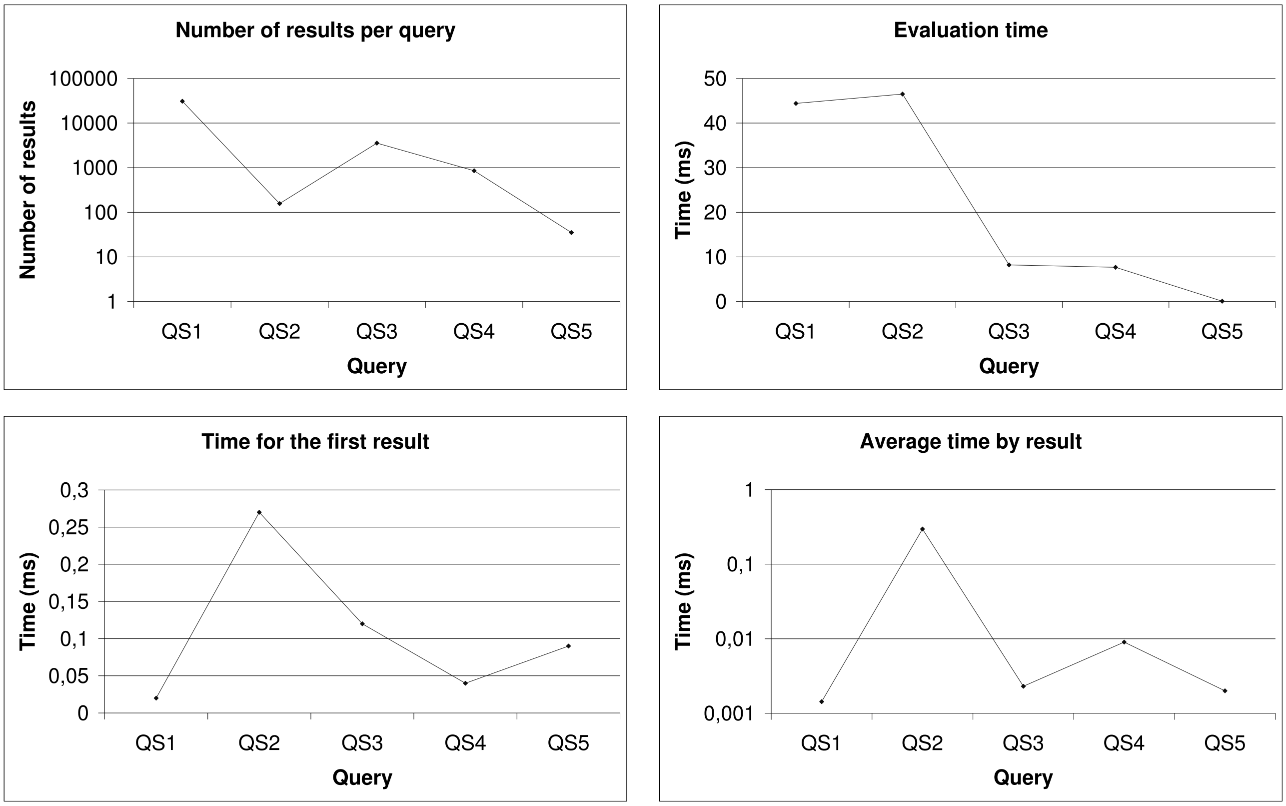


Figure 7: Experimental results for queries over the SP data set

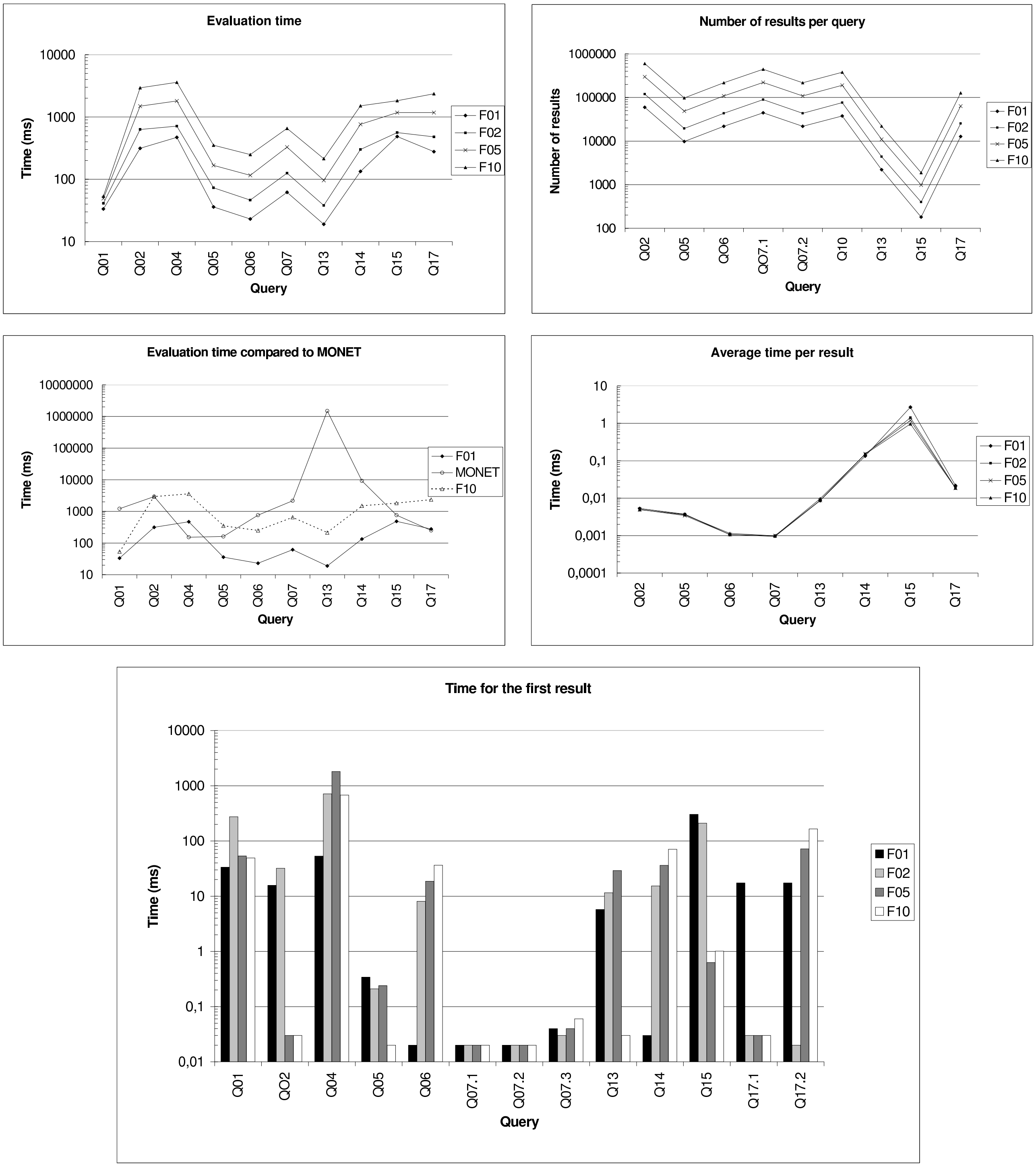


Figure 8: Experimental results for XMark Benchmark queries over data sets from 111 MBytes (F01) to 1.1 GBytes (F10)