SIGOPT: Using Schema to Optimize XML Query Processing

Stelios Paparizos*
Microsoft Search Labs
stelios.paparizos@microsoft.com

Jignesh M. Patel†
University of Michigan
jignesh@eecs.umich.edu

H. V. Jagadish‡
University of Michigan
jag@eecs.umich.edu

Abstract

There has been a great deal of work in recent years on processing and optimizing queries against XML data. Typically in these previous works, schema information is not considered, so that evaluation techniques can continue to be used even in the absence of one. However, schema information is often available and, in this paper, we show that when available it can be exploited to great advantage in ways that complement "traditional" XML query optimization. To be usable in practice, we require that aspects of schema, essential for our purposes, be captured in a Schema Information Graph *(SIG). We exploit such meta-data knowledge with a preprocessing enumeration phase that detects potentially interchangeable evaluation units – we call such units* Alternate Paths.

We show, within an algebraic framework, methods that can break down a pattern tree into elementary paths and substitute them by one or more less costly Alternate Paths. This approach allows us to present various rewritten forms of the XML query to the query optimizer, and allows the DBMS to explore a larger space of query evaluation plans. We assessed the benefits of the proposed techniques experimentally with the XMark data set and show that the SIG-based optimizations can result in significant performance improvements.

1 Introduction

XML is the ubiquitous approach for representing semi-structured data. Yet, most approaches treat XML as schema-less and do not consider schema information at all during the query evaluation process. This is done so that evaluation techniques can work correctly independently of the presence of a schema. However, in practice, most industrial XML datasets often have predefined schemas or one can employ schema inference methods on them. Ignoring such metadata knowledge leads to many missed potential optimization opportunities.

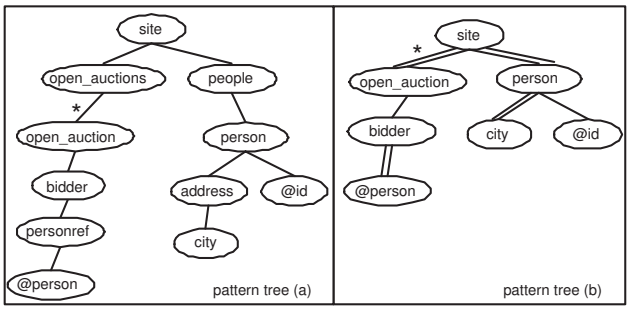


Figure 1. Interchangeable Pattern Trees.

In this paper, we explore opportunities for schema information to affect the query evaluation performance. Our goal is to develop a *practical* solution that can perform well within the constraints of an optimizer. For this purpose we create a simple structure, called *Schema Information Graph* (SIG), to store metadata knowledge. We process this graph to generate a list of potentially interchangeable pieces called *Alternate Paths*. We use these paths during runtime to replace parts of the query with more cost-efficient alternatives. The main benefit of this approach is to pay a preprocessing penalty up front but provide the optimizer with an inexpensive method to increase the search space when generating physical execution plans, thus enhancing overall performance.

To simplify presentation of examples we adopt a pattern tree based native algebraic XML approach (as in [7]). The proposed techniques use schema knowledge to determine that two pattern trees are interchangeable because they return the same results when applied on the database. Yet, depending on the underlying implementation engine, access methods and available indexes, the evaluation cost of one could clearly dominate the other. An example is shown in Figure 1. Here, tree (a) should be preferred when using navigation steps (due to the '//' relationship), whereas tree (b) is more effective when using straight forward binary structural joins [1]. The rewrites can produce both tree (a) from (b) and (b) from (a), along with multiple other variations – then the optimizer can choose the most effective.

*Work done while author was at University of Michigan.

†Supported in part by NSF under grant IIS-0093059.
‡Supported in part by NSF under grant IIS-0438909.

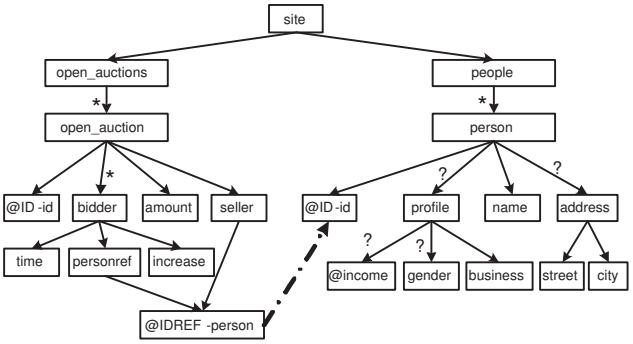


Figure 2. A Schema Information Graph (SIG).

Of course, we cannot address all possible optimizations using schema information. In principal, we suggest a practical technique to deal with '/' and '//' axes in pattern tree branches. We allow for branches of a pattern tree to be replaced by 'quicker' ones. It is important to note that this optimization is not possible without schema information, since determining the "equivalence" of paths a/b/c and a//c requires examining the schema.

2 Schema and Alternate Paths

XML schema can be extremely complex, and can carry a great deal of information. Processing this directly at query optimization time is very hard. We observe that not all schema information is of use for query optimization. In this section, we define a *Schema Information Graph* (SIG) as a simple structure that captures precisely the schema knowledge that is useful for the query optimizations we are interested in. An additional benefit of this approach is that one can be neutral with respect to the format in which schema is represented, whether as DTD or XML Schema or something else. SIG also captures any partial schema information that may have been inferred from the data if no schema was given. SIG is a directed graph. We start by describing its basic units: namely, nodes and edges.

Definition 2.1 *A node $v \in V$ is an XML element, attribute, ID or IDREF. A directed edge $e \in E$ between two nodes u and v specifies a relation between node u and v.*

Nodes in the graph represent XML elements, attributes, IDs and IDREFs. Note that IDs and IDREFs are treated as separate nodes rather than regular attributes. They require special consideration in XML databases and treating them as such in metadata representation simplifies things. The relation between nodes in a graph is either a structural inclusion (parent-child) or a node reference (join).

Definition 2.2 *Given a graph $G = (V, E)$ and a directed edge e from node u to node v, $e = (u, v) \in E$, the Clustering ($Cluster_e$) annotation specifies how many child instances of v can be obtained for each u. The value of $Cluster_e$ can be one of the following:*
"-" *: exactly one node v exists for each u.*
"?" *: zero or one node v exists for each u.*
"+" *: one or more nodes v exist for each u.*
"*" *: zero or more nodes v exist for each u.*

Edges can be annotated with "-", "?", "+" and "*" corresponding to how many children each node can have. For example an edge annotated with "*" between nodes A and B (i.e. A - * - B) states that in the actual data an instance of node A can have zero or one or multiple child instances of node B. The default annotation is "-" (exactly one); it is usually omitted to reduce clutter in figures and discussion. The edge annotation is very important because it captures the heterogeneity in XML data.

Definition 2.3 *A Schema Information Graph (SIG) is a directed graph $G = (V, E)$ where:*

- *Associated with each node $v \in V$ is T_v, specifying if the node is an element, attribute, ID or IDREF.*
- *Associated with each edge $e \in E$ ($e = (u, v)$) is Rel_e, specifying the relation between u and v (parent-child or join) and a $Cluster_e$, specifying the Clustering information between u and v.*

An illustrative example of a SIG with the proper annotations is shown in Figure 2. The figure shows only a fraction of the XMark [9] schema; the entire schema can be described with a SIG but the generated figure would be too complex to show here. Nodes representing elements have just the name of the node, attributes start with '@', IDs start with '*@ID*' and IDREFs '*@IDREF*'. Edges with solid arrows represent parent-child relations and dotted arrows ID-IDREF joins. Clustering annotations are shown next to each edge. To reduce clutter, the default "-" is omitted.

Aside from a stage to store immediate paths along with their relations, the SIG structure enables detection of alternative evaluation procedures. There are inferred subsumed and equivalent paths that can be produced by converting a parent-child relationship to ancestor-descendant or the opposite. Note that a path is an acyclic chain in the graph.

Definition 2.4 *Given a database D and a path P, the Root-Leaf rl matching of P on database D is the set of paths produced after the pattern tree matching $h_P(D)$ has been produced and only the root and leaf nodes are maintained from each path instance. In other words $rl_P(D) = Project_{maintain[root(P),leaf(P)]}[h_P(D)]$.*

The well known procedure of pattern matching[1] h of a tree P on a database D produces trees that have similar structure. A matching of a path P on a database will produce a set of paths similar in structure with P. The difference between regular pattern tree matching and the Root-Leaf one is that in the resulting set of paths, only the root and leaf nodes of every path are maintained and not the entire structure; thus projecting out potential intermediate nodes.

[1] Formal detailed definitions of a pattern tree and the matching procedure can be found in [7]. They are omitted due to space limitations.

Definition 2.5 *Given a database D, a path P and a path Q, P is a subset of Q, if the Root-Leaf matching rl of P is a subset of the Root-Leaf matching rl of Q in the database D. In other words ($P \subseteq Q$) if $[rl_P(D) \subseteq rl_Q(D)]$.*

Note, since XML is a tree structure and given the definition of Root-Leaf matching, it is always true that a/b/c $\subseteq$ a//c. In a database conforming to the schema of Figure 2, an example subset relation is *open_auction/seller/@person* $\subseteq$ *open_auction//@person*.

Definition 2.6 *A path P and a path Q are equivalent when P is a subset of Q and Q is a subset of P. In other words, $P \equiv Q$ if $P \subseteq Q$ and $Q \subseteq P$.*

Equivalent paths produce the same set of final results. Assuming a database conforming to the schema in Figure 2, equivalent paths are *open_auction/bidder/increase* $\equiv$ *open_auction//increase*.

Without schema knowledge, equivalence and containment can only be determined by performing the full matching procedure and comparing the actual set of resulting paths. This is costly, so instead, the available schema information can be used to determine if paths are *Schema Subsumed* or *Schema Equivalent*.

Definition 2.7 *A path Q Schema Subsumes a path P (represented as $P \sqsubseteq Q$), when:*

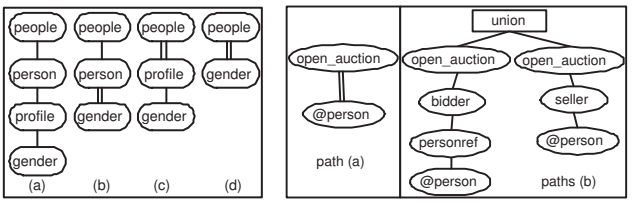
- *P and Q have the same start and end nodes*
- *P has at least one more node than Q*
- *for two or more edges $e_1, \ldots, e_n$: $[e_1, \ldots, e_n \in P$, but $e_1, \ldots, e_n \notin Q]$, there exists an edge g : $[g \notin P$, but $g \in Q]$ that follows these precedence substitution rules: (a)'-'→'?'→'*' or (b)'-'→'+'→'*' or (c) the combination of '?' and '+' is substituted by '*'.*

Definition 2.8 *A path P and a path Q are Schema Equivalent, (represented as $P \doteq Q$) when:*

- *Q Schema Subsumes P ($P \sqsubseteq Q$)*
- *there does not exist another path R in the SIG G that is Schema Subsumed by Q ($\forall R \in G : R \neq P, R \not\sqsubseteq Q$)*

If one considers the schema in Figure 2, an example of Schema Subsumed paths are *open_auction/seller/@person* $\sqsubseteq$ *open_auction//@person*. An example of Schema Equivalent paths are *open_auction/bidder/personref/@person* $\doteq$ *open_auction//personref/@person*.

Schema Subsumed and Equivalent paths provide a good foundation to argue about correctness of rewrites. Yet, during runtime, determining equivalence or containment can be an expensive procedure, especially for large amounts of metadata information. Our goal is to generate a practical method that induces only a small overhead during query plan generation. Thus, we introduce *Alternate Paths*.



(a) One-to-one (b) One-to-many

Figure 3. Example of Alternate Paths.

```
Algorithm Generate Alternate Paths
Input:  a Schema Information Graph (SIG)
Output:  Alternate Paths (APs) with cost annotations
procedure GenerateAPs(in SIG) {
Parse SIG, create a list for all nodes in the graph
step1 for every pair of nodes in the graph generate the
  direct acyclic path that connects them
step2 generate all the subsumption paths using the
  logic A/B/C ⊆ A//C, A/B/C/D ⊆ A//D etc.
step3 process the subsumption relationships
  check only the right side path of each subsumption
    if path exists only once in subsumption list
      then add path pair as one-to-one AP
    if path exists multiple times in subsumption list
      then create a one-to-many AP
step4 sort all paths on path size
step5 for all paths with size n>3 substitute their
  equivalent subpath to produce more Alternate Paths
  start n=4 and increase to n=SIG diameter
    if subpath sized n-1 has an equivalent path
    then substitute it and add as one-to-one AP
step6 for each AP pair, annotate with cost }
```

Figure 4. Generate Alternate Paths.

Definition 2.9 *Given a database D, a path P, a set of paths $Q_1, \ldots, Q_n$ and the cost $C_P, C_{Q1}, \ldots, C_{Qn}$ to produce the Root-Leaf matching rl of $P, Q_1, \ldots, Q_n$ from D. We call $Q_1, \ldots, Q_n$, $n \geq 1$ the Alternate Paths (represented as $\cong$) of P, when the Root-Leaf matching rl of P is equal to the union of the Root-Leaf matchings rl of $Q_i, \ldots, Q_n$ in the database D. In other words ($P \cong [Q_1, \ldots, Q_n]$) $\Leftrightarrow$ ($rl_P(D) \equiv [rl_{Q1}(D) \cup \ldots \cup rl_{Qn}(D)]$). Alternate Paths, for each path P, are stored as metadata with the SIG structure and annotated with the cost $C_P, C_{Q1}, \ldots, C_{Qn}$.*

The cost information C_P of the matching procedure represents the penalty the system will pay to construct the set of results produced when applying path P on the database. Data access methods, navigation steps, structural joins and indexes affect cost information. Cost models are system dependent and involve complex procedures, for example see [8, 12]. It is not within the scope of this paper to describe how to calculate the appropriate cost. We operate under the assumption that cost estimates and histograms exist and we use them in conjunction with Alternate Paths.

Essentially, Alternate Paths (APs) are a non-empty set of paths (can be singleton) that can substitute one path while still producing the same set of results. Alternate Path pairs can be one-to-one or one-to-many. Figure 3(a) shows a group of one-to-one pairs; note, (a)≅(b),(a)≅(c), (a)≅(d), (b)≅(d), (c)≅(d) but one cannot determine if (b)≅ (c). Figure 3(b) shows a one-to-many pair, one path can be substituted by two unioned together.

```
Algorithm Alternate Tree Branches
Input:  Alternate Paths list, Pattern Tree
Output:  Reconstructed Pattern Tree
procedure TreeBranches(in tree) {
  Call CreateSimplePaths(tree), get a list of paths
  For each path p in list call ReplaceWithAPs(p)
  Reconstruct tree }
procedure CreateSimplePaths(in tree) {
  For each branch do
    Traverse branch until leaf, subtree root,
      node with bound variable or filter condition
    if leaf node, create path from root to leaf
    if subtree root,
      create path from tree root to subtree root
      call CreateSimplePaths(pass subtree)
    if node with bound variable or filter condition,
      create path from root to bound/filter node
      continue traversal with rest of branch }
procedure ReplaceWithAPs(in path) {
  Check path entry on Alternate Paths list
  If one-to-one Alternate Path exists
    replace with path that has minimum cost
  If one-to-many Alternate Path exists
    replace with unioned paths if smaller combined cost
  adjust root and leaf node references to new paths }
```

Figure 5. Alternate Tree Branches.

ALTERNATE PATH GENERATION: Alternate Paths are generated from SIG using the procedure in Figure 4. The intuition behind the algorithm has three basic principles. First, the algorithm generates all the direct acyclic paths between every pair of nodes in the schema. Then it detects Schema Subsumed and Schema Equivalent paths. Using that information, it generates the list of Alternate Paths and associates the proper cost with each paired entry. For example the Alternate Paths in Figure 3 are generated when the algorithm is applied on the schema seen in Figure 2.

This brute force technique for calculating all such paths *a priori* can be thought as costly – it is $O(n \log^2 n)$. But since schema does not change that often, and the number of nodes even in complex schemas is not that big, this procedure only needs to be executed rarely. Once all Alternate Paths are identified and stored, data updates only need to update the cost information.

3 Optimization Opportunities

We have designed a rewrite that consumes a pattern tree and replaces parts of it with more cost-efficient alternatives using the Alternate Path list. The rewrite follows a few basic steps: a) First, it breaks the pattern tree into simple paths between (sub)tree root, leafs and nodes with bound variables or filtering conditions, b) Next, it consults the Alternate Path list and replaces these paths with 'faster' alternatives and c) Finally, it reconstructs the tree, adjusting any node references from succeeding operations from the old tree to the new one. Note that without schema knowledge the rewrite could not have replaced any branches and still produce the correct results. Pseudocode for the Alternate Tree Branches algorithm can be found in Figure 5. Here, we present a couple of interesting illustrative examples.

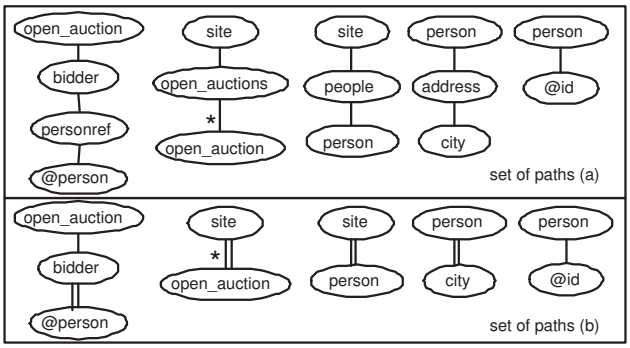


Figure 6. Alternate Paths for all branches in the patterns trees of Figure 1.

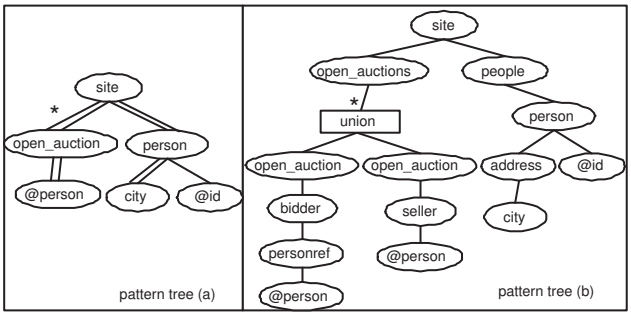


Figure 7. Rewrite with one-to-many APs.

The first example is based on the pattern trees in Figure 1. They both correspond to a query that asks for all *open_auction* items a *person* from a given *city* is bidding on. Important nodes for such a query are, *open_auction*, *person*, *city* and *@person*, *@id* for the ID-IDREF join. Consider going in the direction from tree (a) to tree (b). First, the simple paths are generated for tree (a), they are shown as paths (a) in Figure 6. Note, *open_auction* is considered a bound variable and has to be retained in the simple paths – else a longer path from *site* to *@person* would have been generated. Next, for each path of tree (a) the Alternate Paths are considered and the more cost-effective alternatives are selected. Assuming a schema as the one seen in Figure 2 and cost estimates based on simple structural joins (as in [1]), paths (b) of Figure 6 are selected. Note, for *open_auction/bidder/personref/@person* there exist two different one-to-one Alternate Paths, *open_auction//personref/@person* and *open_auction/bidder//@person*. Either one could have been chosen depending on the cost estimations. Finally, using paths (b) we reconstruct the 'faster' tree, shown as tree (b) in Figure 1. Note that the algorithm of Figure 5 could transform tree (b) into tree (a) in a direction opposite to the one just described – the underlying engine or available indexes could dictate such a direction and the algorithm handles it gracefully.

Another interesting scenario that further demonstrates the power of this technique is the transformation shown in Figure 7. Note how one path *open_auction//@person* has two paths as Alternate Paths: *open_auction/seller/@person*

Query	TLC	SIG	speed up
xmark13	0.5	0.42	1.19
xmark15	1.55	0.64	2.42
xmark16	1.74	0.71	2.45

Table 1. Response times (secs) for XMark factor1. TLC = Regular Timber algebra, SIG = After Schema Optimization.

and *open_auction/bidder/personref/@person*. Using the algorithm we go from tree (a) to tree (b) by combining the two paths with a union operation. For navigational query engine tree (b) could be orders of magnitude faster.

4 Experiments

Our solution was implemented as part of TIMBER [10]. We used the XMark [9] schema and data set for experiments. Factor 1 of XMark produces an XML document that occupies 722MB when stored in our system. Experiments were executed on an Intel Centrino 2G GHz machine. TIMBER was set up to use a 128MB buffer pool. All XML queries were translated into a pattern tree based algebra (TLC [7]). We compared the execution time before and after SIGOPT optimizations. Results are summarized in Table 1.

Schema-aware optimizations perform well in all cases. In many cases the query is executed 2 times faster than the regular TIMBER execution plan. The biggest benefit is seen when a query uses paths of the form many/many/many and is converted to many//many. In such cases, there is a big benefit from avoiding many structural joins. We present two such queries in xmark15 and xmark16. SIGOPT has minimal impact on queries where the matched data nodes are of the form 1/1/1/few for each result, for example xmark13.

5 Related Work

There has been some interest in optimizing XPath expressions possibly making use of any available schema knowledge in [4, 5, 11]. These papers focus on testing for containment of XPath expressions using various constructors (axes) like '/', '//', '*', '[]', '|' (or) and so on. Although interesting and theoretically sound, these papers largely focus on algorithms that test bounding and containment of different axes of an XPath expression. Instead, we use their principles on containment and suggest practical structures and optimization techniques that take advantage of schema knowledge and optimize an entire query within the constraints of a real system.

Besides general XPath expressions, the authors in [2] also consider pattern tree minimization that has similar principles with our approach. In general they try to minimize a tree pattern using some constraints, one being schema information. But, their schema awareness is not discussed in detail and they do not extensively produce 'quicker' plans for both navigation and structural join physical primitives.

There was an effort in TSL [6] to use rewrites expressed in terms of a set of views considering some DTD constraints. Also the PAT [3] algebra had derived constraints from DTDs in order to optimize expressions. In spirit, they are also similar to our approach in that they try to optimize an entire query. Yet, they are early efforts mostly considering structured or semi-structured documents (not exactly XML) and did not perform an in depth analysis of all potential optimizations using schema information in both the cost-based and rewrite phases.

6 Conclusion

In this paper, we introduced *Schema Information Graph* and *Alternate Paths*, practical structures that can capture important properties of XML metadata for use during query optimization. We also developed methods for generating these structures and optimization techniques that take advantage of them. We demonstrated experimentally that such optimizations can produce a significant performance increase. Although the optimization techniques we presented were within the framework of a tree algebra, we believe that they can be widely adopted in any XML engine.

References

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

[2] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. In *Proc. SIGMOD Conf.*, Jun. 2001.

[3] K. Bohm, K. Aberer, M. T. Ozsu, and K. Gayer. Query optimization for structured documents based on knowledge of the document type definition. In *Proc. IEEE Advances in Digital Libraries*, 1998.

[4] G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *Proc. PODS Conf.*, Jun. 2002.

[5] F. Neven and T. Schwentick. Xpath containment in the presence of disjunction, dtds, and variables. In *ICDT*, 2003.

[6] Y. Papakonstantinou and V. Vassalos. Query rewriting for semistructured data. In *Proc. SIGMOD Conf.*, 1999.

[7] S. Paparizos, Y. Wu, L. V. S. Lakshmanan, and H. V. Jagadish. Tree logical classes for efficient evaluation of XQuery. In *Proc. SIGMOD Conf.*, Jun. 2004.

[8] N. Polyzotis and M. Garofalakis. Statistical synopses for graph-structured XML databases. In *Proc. SIGMOD*, 2002.

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

[10] University of Michigan. The TIMBER native XML system. `http://www.eecs.umich.edu/db/timber`.

[11] P. T. Wood. Containment for XPath fragments under DTD constraints. In *Proc. ICDT Conf.*, 2003.

[12] Y. Wu, J. M. Patel, and H. V. Jagadish. Estimating answer sizes for XML queries. In *Proc. EDBT Conf.*, Mar. 2002.