Type-Based XML Projection

Véronique Benzaken[1] Giuseppe Castagna[2] Dario Colazzo[1] Kim Nguyễn[1]

[1]LRI, Université Paris-Sud 11, Orsay - France [2] École Normale Supérieure de Paris - France

ABSTRACT

XML data projection (or pruning) is one of the main optimization techniques recently adopted in the context of main-memory XML query-engines. The underlying idea is quite simple: given a query Q over a document D, the subtrees of D not necessary to evaluate Q are pruned, thus obtaining a smaller document D'. Then Q is executed over D', hence avoiding to allocate and process nodes that will never be reached by navigational specifications in Q.

In this article, we propose a new approach, based on types, that greatly improves current solutions. Besides providing comparable or greater precision and far lesser pruning overhead our solution, unlike current approaches, takes into account backward axes, predicates, and can be applied to multiple queries rather than just to single ones. A side contribution is a new type system for XPath able to handle backward axes, which we devise in order to apply our solution.

The soundness of our approach is formally proved. Furthermore, we prove that the approach is also complete (i.e., yields the best possible type-driven pruning) for a relevant class of queries and DTDs, which include nearly all the queries used in the XMark and XPathMark benchmarks. These benchmarks are also used to test our implementation and show and gauge the practical benefits of our solution.

1. MOTIVATIONS AND CONTRIBUTION

As explained by Marian and Siméon [14], main-memory XML query engines are often the primary choice for applications that do not wish or cannot afford to build secondary storage indexes or load a database before query processing. One of the main optimisation techniques recently adopted in this context is XML data projection (or pruning) [14, 9].

The basic idea behind document projection is very simple and powerful at the same time. Given a query Q over a document D, sub-trees of D that are not necessary to evaluate Q are pruned, thus yielding a smaller document D'. Then Q is executed over D', hence avoiding to allocate and process nodes that will never be reached by navigational specifications in Q. This ensures that evaluation over D' is equivalent to and more efficient than the evaluation over D.

As shown in [14, 9], XML navigation specifications expressed in queries tend to be very selective, especially in terms of document structure. Therefore, pruning may yield significant improvements both in terms of execution time and in terms of memory usage (for main-memory XML query engines, very large documents can not be queried without pruning).

1.1 State of the art

Marian and Siméon[14] propose that the actual data-needs of a query Q (that is, the part of data that is necessary to the execution of the query) is determined by statically extracting all paths in Q. These paths are then applied to D at load time, in a SAX-event based fashion, in order to prune unneeded parts of data. The technique is powerful since: (i) it applies to most of XQuery *core*, (ii) it can be applied to a set of queries over the same document, and (iii) it does not require any *a priori* knowledge of the structure of D. However, this technique suffers some limitations. First, the document loader-pruner is not able to manage *backward axes* nor path expressions with predicates (sometimes called "qualifiers") which, especially the latter, can contain precious information to optimise pruning. Also, as a consequence of (iii), the technique does not behave efficiently in terms of loading time and pruning precision (hence, memory allocation) when `//` occurs in paths. Indeed, when `//` is present in a projection path, the pruning process requires to visit all descendants of a node in order to decide whether the node contains a useful descendant. What is worst is that pruning time tends to be quite high and it drastically increases (together with memory consumption) when the number of `//` augments in the pruning path-set. As a matter of facts, in this technique pruning corresponds to computing a further query, whose time and memory occupation may be comparable to those required to compute the original query. In particular, in this technique every occurrence of `//` may yield a full exploration of the tree (e.g. see in [14] the test for the XMark [17] query Q7 which only contains three `//` steps and for which just computing the pruning takes longer than executing the query on the original document). Therefore, pruning execution overhead and its high memory footprint may jeopardise the gains obtained by using the pruned document. Finally, as we explain in Section 5, the precision of pruning drastically degrades (even nullified) for queries containing the XPath expressions `descendant::node`$[cond]$, which are very useful and used in practice.

Bressan *et al.* [9] introduce a different and quite precise XML pruning technique for a subset of XQuery FLWR expressions. The technique is based on the *a priori* knowledge of a data-guide for D. The document D is first matched against an abstract representation of Q. Pruning is then performed at run time, it is very precise, and, thanks to the use of some indexes over the data-guide, it ensures good improvements in terms of query execution time. However, the technique is one-query oriented, in the sense that it cannot be applied to multiple queries, it does not handle XPath predicates, and cannot handle backward axes (recall that the encodings of [15] are defined for XPath, and no extension to XQuery-like languages is known). Also, the approach requires the construction and management of the data-guide and of adequate indexes.

1.2 Our contribution

In this article, we present a new pruning approach which is applicable in the presence of typed XML data. This is often the case, as most applications require that data are valid with respect to some external schema (e.g. DTD or XML Schema).

Our technique combines the advantages of the previously mentioned works while relaxing their limitations. Unlike [14, 9], our approach accounts for backward axes, performs a fine-grained analysis of predicates, allows (unlike [9]) for dealing with bunches of queries, and (unlike [14]) cannot be jeopardised by pruning overhead. Our solution provides comparable or greater precision than the other approaches, while it requires always negligible or no pruning overhead. Moreover, contrary to [14, 9], our approach is formally proved to be *sound* (pruning does not alter the result of queries) and, furthermore, we can also prove it to be *complete* (it produces the best possible type-driven pruning) for a substantial class of queries and DTDs.

For the sake of presentation we introduce our framework in three steps. In the first step, we consider a simplified version of XPath, we dub XPath$^\ell$, which includes only upward/downward axes and unnested disjunctive predicates. We define for XPath$^\ell$ a static analysis that determines a set of type names, a *type projector*, that is then used to prune the document(s). One of the particular features of this approach is that our pruning algorithm is characterised by a constant (and low) memory consumption and by an execution time linear in the size of the document to prune. More precisely, a pruning based on type projectors is equivalent to a single bufferless one-pass traversal of the parsed document (it simply discards elements not generated by any of the names in the projector). So if embedded in query processors, pruning can be executed during parsing and/or validation and brings no overhead, while if used as an external tool it requires a time always smaller than or equal to the time used to parse the queried document. Soundness and (partial) completeness results for the static analysis are stated.

The second step consists of extending the analysis to the whole XPath (more precisely, to XPath 1.0), that is, we need to show how to deal with missing axes and with general predicates as defined in the XPath specification. This is done by associating to each XPath query Q a XPath$^\ell$ query P which soundly approximates Q, in the sense that the projector inferred for P is also a sound projector for Q.

The final step is to extend the approach to XQuery (hence, to XPath 2.0). This is obtained by defining a path extraction algorithm as done in [14]. Our path extraction algorithm improves in several aspects (in particular, in terms of extracted paths' selectivity) the one of [14]. It also computes the XPath$^\ell$ approximation of the extracted paths so that the static analysis of the first step can be directly applied to them.

We gauged and validated our approach by testing it both on the XPathMark [12] and on the XMark [17] benchmarks. This validation confirmed expected results: thanks to the handling of backward axes and of predicates the precision of our pruning is in general noticeably higher than for current approaches; the pruning time is linear in the size of the queried document and has a very low memory footprint; the time of the static analysis is always negligible (lower than half a second) even for complex queries and DTDs. But benchmarks also brought unexpected (and pleasant) results. In particular, they showed that type-based pruning brings benefits that go beyond those of the reduced size of the pruned document: by excluding a whole set of data structures (those whose type names are not included in the type projector), the pruning may drastically reduce the resources that must be allocated at run-time by the query processor. For instance, our benchmarks show that for several XMark and XPathMark queries our pruning yields a document whose size is two thirds of the size of the original document, but the query can then be processed using three times less memory than when processed on the original document. This is a very important gain, especially for DOM-based processors, or memory sensitive processors as Galax [1]. As an aside we want to stress that our technique relies on the definition of a new type system for XPath able to handle backward axes, which constitutes a contribution on its own.

The article is organised as follows. Section 2 introduces basic definitions and notations: data model, DTD, validation, projection, type projector. In Section 3 we define XPath$^\ell$ and its semantics, and formally describe how general XPath predicates can be soundly approximated in it. In Section 4 we present our type projectors inference algorithm for XPath$^\ell$, state its formal properties, and deal with the missing XPath axes. In Section 5 we extend our approach to XQuery. Section 6 discusses our implementation and reports the results of our benchmarks. We finally conclude in Section 7 by presenting the perspectives of this work.

For space reasons all proofs of properties are omitted from this presentation. They can be found in the extended version of this work.

2. NOTATIONS

2.1 Data Model

For the sake of concision we present our solution for a simplified version of the XQuery data model where we do not consider node attributes. The extension of our approach to attributes is straightforward (and included in our implementation, see Section 6). An instance of the XQuery data model can then be generated by the following grammar:

$$\begin{array}{llcl} \textbf{Trees} & t & ::= & s_{\mathbf{i}} \;\mid\; l_{\mathbf{i}}[f] \\ \textbf{Forest} & f & ::= & () \;\mid\; f,f \;\mid\; t \end{array}$$

Essentially, it is an ordered sequence of labelled ordered *trees* (ranged over by t), that is an ordered *forest* (ranged over by f), where each node has a unique *identifier* (ranged over by $\mathbf{i}$) and where $()$ denotes the empty forest. Tree nodes are labelled by *element tags* (ranged over by l) while, without loss of generality, we consider only leaves that are text nodes (that is, strings, ranged over by s) or empty trees (that is, elements that label the empty forest).

We define a complete partial order $\preceq$ on forests (and thus on trees) by relating a forest with the forests obtained either by adding or by deleting subforests:

DEFINITION 2.1 (PROJECTION ($\preceq$)). *Given two forests f and f' we say that f' is a* projection *of f, noted as $f' \preceq f$, if f' is obtained by replacing some subforests of f by the empty forest.*

DEFINITION 2.2 (GOOD FORMATION). *A forest is* well formed *if every identifier $\mathbf{i}$ occurs in it at most once. Given a well-formed forest f and an identifier $\mathbf{i}$ occurring in it, we denote by $f@\mathbf{i}$ the unique subtree t of f such that $t = s_{\mathbf{i}}$ or $t = l_{\mathbf{i}}[f']$. The set of identifiers of a forest f is then defined as $Ids(f) = \{\mathbf{i} \mid \exists\, t.\ f@\mathbf{i} = t\}$*

Henceforth we will consider only well-formed forests and confound the notions of a node with that of the identifier of the node.

DEFINITION 2.3 (ROOT ID). *Given a tree t, if $t = s_{\mathbf{i}}$ or $t = l_{\mathbf{i}}[f]$ then we define $RootId(t) = \mathbf{i}$.*

2.2 DTDs and validation

In this work we present the approach for DTDs, but the treatment for XML Schema is similar.[1] Following [13] we define a DTD as a *local tree grammar*, namely a pair (X,E) where X is a distinguished *name* (actually, a non-terminal meta-variable) and E is a set of productions (or *edges*) of the form $\{X_1 \to R_1,\ldots,X_n \to R_n\}$ such that

1. the X_i's are pairwise distinct;
2. each R_i is of the form $a_i[r_i]$ or *String*, where a_i is an element tag, and each r_i is a regular expression over *names* $\{X_1,\ldots,X_n\}$;
3. for each pair $X_i \to a_i[r_i]$ and $X_j \to a_j[r_j]$, $i = j$ if and only if $a_i = a_j$;
4. X is in $\{X_1,\ldots,X_n\}$ (it denotes the root element type).

In the following we write $Names(r)$ for the set of all names used in r and $DN(E)$ for the set of names defined in E (that is, $\{X_1 \ldots X_n\}$). We also say that r is a regular expression over (X,E), if r is a regular expression over names in $DN(E)$. We will use W,X,Y,Z to range over *names*. We use Greek letters to range over sets of names (in particular we use π to stress that the set of names is a *type projector* [cf. Def 2.6] and κ and τ to stress that the set is used as a context or as a type, respectively [cf. Section 4.1]) and S to range over sets of (node) identifiers. When speaking of DTDs we will often identify them with their set of edges E, leaving the root X as implicit.

DEFINITION 2.4 (VALID TREES). *A tree t is* valid *with respect to a* DTD *(X,E), if there exists a mapping (interpretation) $\Im$ from $Ids(t)$ to $DN(E)$ such that:*

1. *$\Im(RootId(t)) = X$*
2. *for each $\mathbf{i}$ in $Ids(t)$, if $t@\mathbf{i} = s_{\mathbf{i}}$ then $\Im(\mathbf{i}) = Y$ and $(Y \to String) \in E$*
3. *for each $\mathbf{i}$ in $Ids(t)$, if $t@\mathbf{i} = l_{\mathbf{i}}[t_1,...,t_n]$, then $\Im(\mathbf{i}) \to l[r] \in E$ and $\Im(RootId(t_1)),\ldots,\Im(RootId(t_n))$ is generated by r.*

In this case we say that t is $\Im$-valid with respect to (X,E) and write $t \in_{\Im} (X,\ E)$ to indicate it.

Algorithms to validate XML trees are well known (see [13]). Every validation algorithm produces, as a side effect, an interpretation for the validated tree. Note that if t is valid with respect to a DTD, then there is a unique interpretation $\Im$ from t to the DTD. This is a direct consequence of the fact that, in DTDs, element tags determine their content (as stated by the third condition on local tree grammars).

2.3 Type projectors

Given a tree t valid with respect to a DTD (X,E), we can use subsets of $DN(E)$ to project that tree. Essentially, only nodes that are associated with names in the projecting subset of $DN(E)$ are kept in the projection. Of course not every subset of names can be used to project a tree, since we want to delete whole subtrees (not nodes in the middle of a tree), thus if we discard some name, we must also discard all the names it generates. In order to define formally this notion we need to define the reachability relation $\Rightarrow_E$, that we introduce below together with several other definitions that we use later in the paper.

[1]The extension of our approach to XML Schema simply needs some special treatment of local elements. More difficult instead is to modify it so as to obtain efficient pruning also for the new XPath 2.0 tests that check the schema of nodes. See the discussion in our conclusion.

DEFINITION 2.5 (FORWARD REACHABILITY). *Given a* DTD *(X,E) and $Z \in DN(E)$, we write $Z \Rightarrow_E Y$ if and only if $Z \to a[r] \in E$ and $Y \in Names(r)$. We use $\Rightarrow_E^+$ and $\Rightarrow_E^*$ to denote respectively the transitive closure and the transitive and reflexive closure of $\Rightarrow_E$.*

Strings of names are called *chains* and ranged over by c, c_i, c',... In particular we use $Chains_{(X,E)}(Y)$ to denote the set of all chains rooted at Y, defined as $\{Y X_1 \ldots X_n \mid Y \Rightarrow_E X_1 \Rightarrow_E \ldots \Rightarrow_E X_n, n \geq 0\}$. We use $Names(c)$ to denote the set of all names occurring in a chain c.

DEFINITION 2.6 (TYPE-PROJECTORS). *Given a* DTD *(X,E), a (possibly empty) set of names $\pi \subseteq DN(E)$ is a* type projector *for (X,E) if and only if there exists $C \subseteq Chains_{(X,\,E)}(X)$ such that*

$$\pi = \bigcup_{c \in C} Names(c)$$

A type projector is thus a set of names generated (i.e. reached) by a suite of productions starting from the root of the DTD. A type projector can be used to prune a valid tree as follows:

DEFINITION 2.7 (TYPE DRIVEN PROJECTIONS). *Let π be a type projector for (X,E) and t a forest or tree such that $t \in_{\Im} (X,E)$. The π-projection of t, noted as $t\backslash_{\Im}\pi$, is defined as follows:*

$$\begin{array}{lcll} l_{\mathbf{i}}[f]\backslash_{\Im}\pi & = & l_{\mathbf{i}}[f\backslash_{\Im}\pi] & \Im(\mathbf{i}) \in \pi \\ l_{\mathbf{i}}[f]\backslash_{\Im}\pi & = & () & \Im(\mathbf{i}) \notin \pi \\ s_{\mathbf{i}}\backslash_{\Im}\pi & = & s_{\mathbf{i}} & \Im(\mathbf{i}) \in \pi \\ s_{\mathbf{i}}\backslash_{\Im}\pi & = & () & \Im(\mathbf{i}) \notin \pi \\ (f,f')\backslash_{\Im}\pi & = & (f\backslash_{\Im}\pi),(f'\backslash_{\Im}\pi) & \end{array}$$

In words, pruning erases (by replacing it by an empty forest) every node that corresponds to a name not in π.

LEMMA 2.8. *Let π be a type projector for (X,E). Then for every tree $t \in_{\Im} (X,E)$ it holds $(t\backslash_{\Im}\pi) \preceq t$.*

3. XPATH AND XPATH$^\ell$

In XPath, queries are expressed by defining a path of steps separated by `/`. For instance,

$$\begin{array}{rcl} Q & = & /\mathtt{descendant}::author \\ & & /\mathtt{child}::\mathtt{text}[\mathtt{self}::\mathtt{node}=\text{"}Dante\text{"}] \\ & & /\mathtt{parent}::book/\mathtt{child}::title \end{array}$$

is the query that returns all titles of books whose author is "Dante". First, the navigational part instructs to descend to all text nodes whose parent is an author ($/\mathtt{descendant}::author/\mathtt{child}::\mathtt{text}$), then the predicate selects those nodes that are the string "Dante" (`[self::node="`*Dante*`"]`), and finally the navigation ascends to the book element and descends to the title.

The inference rules we define in Section 4 do not work directly on queries such as Q. The rules are defined for XPath$^\ell$ a subset of XPath that we introduce in this section. XPath$^\ell$ includes downward and upward axes and a special kind of predicates. In order to statically analyse Q (or any other XPath query that is not in XPath$^\ell$), we will find a XPath$^\ell$ query that approximates Q soundly with respect to the pruning inferred by the rules (Section 3.3), and use it to deduce the pruning for Q.[2] Of course, these approximations, as well as those we introduce later on, will only be used to determine the pruning: the pruned document will be queried by the original query.

For the sake of presentation, we first deal with "simple paths", that is, path expressions with upward and downward axes in which no predicate occurs. Then, in Section 3.2 we add XPath$^\ell$ predicates,

[2]For instance, the approximation of our sample query Q is obtained by replacing in Q the predicate `[self::node]` for the current one.

i.e. disjunctions of simple predicates, and finally in Section 3.3 we show how to approximate generic XPath conditions into XPath$^\ell$. The missing axes are dealt with in Section 4.3.

3.1 Simple paths

Simple paths are defined by the following grammar:

$$
\begin{array}{rcl}
\mathit{SPath} & ::= & \mathit{Step} \mid \mathit{SPath}/\mathit{SPath} \mid /\mathit{SPath} \\
\mathit{Step} & ::= & \mathit{Axis}\texttt{::}\mathit{Test} \\
\mathit{Axis} & ::= & \texttt{self} \mid \texttt{child} \mid \texttt{descendant} \\
 & \mid & \texttt{parent} \mid \texttt{ancestor} \mid \texttt{ancestor-or-self} \\
 & \mid & \texttt{descendant-or-self} \\
\mathit{Test} & ::= & \mathit{tag} \mid \texttt{node} \mid \texttt{text}
\end{array}
$$

where *tag* is a meta-variable ranging over element tags. Henceforward, we omit the treatment of leading / (i.e., absolute paths) and of `descendant-or-self` and `ancestor-or-self` axes: their handling would blur definitions and can be easily deduced from the rest.

The formal semantics of paths is given in three definitions. First, we formalise *Test* filtering, then *Axis* selections, and finally we combine the two notions to define the semantics of a single step *Axis* :: *Test*. The definitions comply with the W3C XPath semantics [2].

DEFINITION 3.1 (FILTERING). *Given a tree t and a set of nodes $S \subseteq Ids(t)$ we define*

$$
\begin{array}{rcl}
S ::_t l & = & \{\mathbf{i} \in S \mid t@\mathbf{i} = l_{\mathbf{i}}[f]\} \\
S ::_t \texttt{node} & = & S \\
S ::_t \texttt{text} & = & \{\mathbf{i} \in S \mid \exists\, s\ .\ t@\mathbf{i} = s_{\mathbf{i}}\}
\end{array}
$$

DEFINITION 3.2 (AXES SELECTION). *Given a tree t and a set of nodes $S \subseteq Ids(t)$ (called context nodes), we define $[\![Step]\!]_t(S)$ as the set of nodes resulting by applying Step to each node in S*

$$
\begin{array}{rcl}
[\![\texttt{self}]\!]_t(S) & = & S \\
[\![\texttt{child}]\!]_t(S) & = & \bigcup_{\mathbf{i}\in S}\{\mathbf{i}' \mid (\mathbf{i},\mathbf{i}') \in \mathscr{E}(t)\} \\
[\![\texttt{parent}]\!]_t(S) & = & \bigcup_{\mathbf{i}\in S}\{\mathbf{i}' \mid (\mathbf{i}',\mathbf{i}) \in \mathscr{E}(t)\} \\
[\![\texttt{descendant}]\!]_t(S) & = & \bigcup_{\mathbf{i}\in S}\{\mathbf{i}' \mid (\mathbf{i},\mathbf{i}') \in \mathscr{E}(t)^+\} \\
[\![\texttt{ancestor}]\!]_t(S) & = & \bigcup_{\mathbf{i}\in S}\{\mathbf{i}' \mid (\mathbf{i}',\mathbf{i}) \in \mathscr{E}(t)^+\}
\end{array}
$$

where $\mathscr{E}(t)$ is the edge relation *of t, that is, $\mathscr{E}(t) = \{(\mathbf{i},\mathbf{i}') \mid t@\mathbf{i} = l_{\mathbf{i}}[f,t',f'] \wedge RootId(t') = \mathbf{i}'\}$, and $\mathscr{E}(t)^+$ is its transitive closure.*

DEFINITION 3.3 (SIMPLE PATH SEMANTICS). *Given t, a set $S \subseteq Ids(t)$ and a path SPath, we define the evaluation of path SPath over S nodes as follows:*

$$
\begin{array}{rcl}
[\![\mathit{Axis} :: \mathit{Test}]\!]_t(S) & = & ([\![\mathit{Axis}]\!]_t(S)) ::_t \mathit{Test} \\
[\![\mathit{SPath}_1/\mathit{SPath}_2]\!]_t(S) & = & [\![\mathit{SPath}_2]\!]_t([\![\mathit{SPath}_1]\!]_t(S))
\end{array}
$$

3.2 Predicates

XPath queries use predicates to express some filtering conditions that cannot be expressed by simple paths. Predicates mix *structural conditions* (directly expressed by means of paths) with *non-structural conditions* (expressed by functions, operators, values, etc...).

We have seen an example of a non-structural condition in the query Q extracting all book titles of books written by Dante, defined at the beginning of the section. The best pruning for the Q query is the one that deletes all books whose authors do not include Dante. To implement such a pruning, one should extract from the query value-based conditions (e.g. being equal to "Dante"). This would drastically complicate the treatment without bringing a significant gain: previous experiments have shown that navigational specifications are already sufficient to obtain important improvements in memory reduction and query execution time [14]. Hence we'd rather abstract out non-structural conditions and only retain structural ones. More precisely, our analysis will have to work only on conditions defined as follows:

$$\mathit{Cond} \quad ::= \quad \mathit{SPath} \mid \mathit{Cond}\ \texttt{or}\ \mathit{Cond}$$

XPath$^\ell$ is then defined by the following grammar:

$$\mathit{Path} ::= \mathit{Step} \mid \mathit{Step}[\mathit{Cond}] \mid \mathit{Path}/\mathit{Path}$$

We will use meta-variables *Path* and *P* to range over these paths, and reserve *SPath* for simple paths and *Q* for general XPath queries. Note that the definition of *Cond* uses simple paths, therefore in XPath$^\ell$ conditions are not nested.

Semantics of XPath$^\ell$'s paths is defined by substituting in Definition 3.3 *Path* for *SPath* and by adding the following cases

$$
\begin{array}{l}
[\![\texttt{self} :: \texttt{node}[C]]\!]_t(S) = \{\mathbf{i} \in S \mid Check_t[C](\mathbf{i})\} \\
[\![\mathit{Axis} :: \mathit{Test}[C]]\!]_t(S) \quad = [\![\mathit{Axis} :: \mathit{Test}/\texttt{self} :: \texttt{node}[C]]\!]_t(S)
\end{array}
$$

where $Check_t[Cond](\mathbf{i})$ is the following boolean function:

$$
\begin{array}{rcl}
Check_t[Path](\mathbf{i}) & = & [\![Path]\!]_t(\{\mathbf{i}\}) \neq \varnothing \\
Check_t[C_1\ \texttt{or}\ C_2](\mathbf{i}) & = & Check_t[C_1](\mathbf{i}) \vee Check_t[C_2](\mathbf{i})
\end{array}
$$

3.3 Handling XPath predicates

The predicates of the previous section cover only a small part of XPath. If we want to apply our analysis to XPath and XQuery we must be able to deal with the more general expressions used in conditions.

In this section we show how to rewrite every predicate *Exp* expressible in XPath to a simple condition *Cond* such that *Cond* is a sound approximation of *Exp* with respect to data needs: the pruning determined for *Cond* preserves the semantics for *Exp*. In other words, if we take a generic XPath query *Q* and approximate all its predicates to infer a projector π, then the execution of (the original) *Q* on a given document or on the document pruned by π yield the same result. This rewriting, together with the treatment of missing axes of Section 4.3, allows us to deal with a large subset of XQuery and XPath queries, covering those in XPathMark [12] and XMark [17] benchmarks.

More formally, we show how to rewrite an expression *Exp* into a condition *Cond*, where *Exp* is defined as

$$\mathit{Exp} ::= Q \mid \mathit{Exp}\ \mathit{op}\ \mathit{Exp} \mid f(\mathit{Exp}_1, \ldots, \mathit{Exp}_n) \mid \mathit{AExp}$$

where *op*∈{`eq`, `ne`, `lt`, `le`, `gt`, `ge`, `=`, `!=`, `<`, `<=`, `>`, `>=`, `is`, `<<`, `>>`, `or`, `and`} is an operator, *AExp* ranges over arithmetic expressions (see [2]) and base values (PCDATA), *f* ranges over XPath and XQuery functions and operators [5] such as `count`, `contains`, `is-zero`, `not`, `empty`, etc., and *Q* is a generic XPath query, that is:

$$Q\ ::=\ \mathit{Step} \mid \mathit{Step}[\mathit{Exp}] \mid \mathit{Step}/Q \mid \mathit{Step}[\mathit{Exp}]/Q$$

The rewriting is obtained by a path-extracting function ***P*** that applied to an expression *Exp* returns a set of *simple* paths whose "`or`" constitutes the approximation of *Exp*.[3]

Let us outline the rewriting by an example. Consider the predicate `[position()>1 and parent::node/`*book*`/`*author*`="`*Dante*`"`

[3]For lack of space we cannot present the full treatment of predicates that we have implemented in our prototype. In particular, we do not consider absolute paths (although they need special treatment they do not introduce any significant problem) nor we formally define the approximation for each XPath and XQuery function.

`and` *year*>1313]. In our system this predicate is approximated by `[ self::node or parent::node/`*book*/*author* or *year* `]`. Essentially, given a predicate *Exp* we obtain a condition *Cond* that soundly approximates it by retaining the disjunction of all structural conditions (like `parent::node/`*book*/*author* and *year* in the previous example), plus either `descendant-or-self::node` or `self::node` if some non-structural condition is present (for instance, `position()>1`). The choice between `self::node` and `descendant-or-self::node` depends on the functions and operators used in the condition: for instance functions like `position` or `count` require `self::node` since their execution requires only the root nodes; instead a function such as `string` needs the whole tree. Therefore we suppose to have a predefined function $\boldsymbol{F}$ that for each f returns either `descendant-or-self::node` or `self::node`. For the sake of generality we suppose that this function depends on the position of the argument in n-ary function. Thus, for, say, `count(`*SPath*`)` and `string(`*SPath*`)`, we have $\boldsymbol{P}(\texttt{count}(\mathit{SPath})) = \mathit{SPath}/\boldsymbol{F}(\texttt{count},1) = \mathit{SPath}/\texttt{self::node}$, and $\boldsymbol{P}(\texttt{string}(\mathit{SPath})) = \mathit{SPath}/\boldsymbol{F}(\texttt{string},1) = \mathit{SPath}/\texttt{descendant-or-self::node}$. Formally, we have:

$$\begin{array}{lcl}
\boldsymbol{P}(\mathit{Step}) & = & \{\mathit{Step}\} \\
\boldsymbol{P}(\mathit{Step}[\mathit{Exp}]) & = & \mathit{Step}/\boldsymbol{P}(\mathit{Exp}) \\
\boldsymbol{P}(\mathit{Step}/Q) & = & \mathit{Step}/\boldsymbol{P}(Q) \\
\boldsymbol{P}(\mathit{Step}[\mathit{Exp}]/Q) & = & \mathit{Step}/(\boldsymbol{P}(Q)\cup\boldsymbol{P}(\mathit{Exp})) \\
\boldsymbol{P}(\mathit{Exp}\ op\ \mathit{Exp}') & = & \boldsymbol{P}(\mathit{Exp})\cup\boldsymbol{P}(\mathit{Exp}') \\
\boldsymbol{P}(f(\mathit{Exp}_1,\ldots,\mathit{Exp}_n)) & = & \bigcup_{i=1,n}(\boldsymbol{P}(\mathit{Exp}_i)/\boldsymbol{F}(f,i))\cup \\
 & & \cup\{\texttt{self::node}\}
\end{array}$$

where we used the notation Step/A as a shorthand to denote the set $\{\mathit{Step}/\mathit{SPath} \mid \mathit{SPath}\in A\}$ when A is a set of simple paths (similarly for A/Step).

The presence of $\{\texttt{self::node}\}$ in the last line is motivated by the fact that when we have a non structural condition, paths must not be used to restrict the inferred projectors, since this would not yield a sound approximation. More precisely, when *Exp* is purely structural, that is it only involves paths in (possibly nested) conditions, then these paths are extracted to refine the projection. For instance, in `descendant::node[child::`*a*`]` we can use the condition `[child::`*a*`]` to refine projection inference : we select only element types having an *a* child. On the other hand, when *Exp* is not purely structural, as in `descendant::node[not(child::`*a*`)]` or `descendant::node[count(child::`*a*`)<5]`, we can not use the same projector as for `descendant::node[child::`*a*`]`: if we use `[child::`*a*`]` to restrict the projection, we would alter the result of the last two queries, so the projector would be unsound. To guarantee soundness, we extract paths from the arguments `not` and `count` and add the condition $\{\texttt{self::node}\}$ to ensure that we do not prune nodes necessary to the evaluation of the functions. So, for the two queries, after condition rewriting, we have the approximating query `descendant::node[child::`*a* `or self::node]`, yielding a sound projector.

To resume, to indicate the fact that, in the presence of not purely structural conditions, paths must not be used to restrict inferred projectors, we add the always true condition $\{\texttt{self::node}\}$. Of course, we could have adopted more precise (and complex) techniques, but we preferred this solution as we consider it a good compromise between precision and simplicity.

We want also to stress that here we reach the limits of XQuery and XPath type systems. If we had worked on more advanced XML languages such as ℂDuce [6] or ℂQL [7] their richer type system (it includes union, intersection, negation, and singleton types) would allow us to precisely capture more predicates and use them for a much finer pruning (as it is done in ℂQL query optimisation).

4. STATIC ANALYSIS

In this section we define deduction rules to statically infer from a XPath$^\ell$ path P and a DTD E a type-projector for an input document validating E. We show that the analysis is sound, and that it enjoys completeness for a large class of queries when E is a $*$-guarded and non-recursive DTD (see Definition 4.3 below). Soundness means that executing the query on the original document and on the document pruned by the inferred projector yields the same result. Completeness means that if we take a type projector smaller (i.e., more selective) than the inferred one, then there exists a document validating E for which the result of the two executions is not the same. When the conditions on DTDs or on queries are relaxed the analysis is still sound but it may be not complete. Nevertheless, as we will illustrate, it still is very precise.

In order to define our static type inference we proceed in two steps.

1. Given a path P and a DTD E we type P by the set of all elements that may appear in the result of applying P to a document validating E. This is done in Section 4.1 (actually, we will be more precise and type P by the set of all names of E that *generate* the elements in the result).
2. We use the type inference at the previous point to define the inference of type projectors. In particular we will use the cases in which the previous type inference returns the empty set to determine the points in which pruning must be performed. This is done in Section 4.2.

4.1 Type inference

Given a path *Path* and a DTD E we want to find a set of names of E that generates elements that can be found in the result of P. Formally, we want to infer a set $\tau\subseteq DN(E)$ such that

$$\forall t\in_{\mathfrak{I}} E.\ \mathfrak{I}([\![\mathit{Path}]\!]_t(\mathit{RootId(t)}))\subseteq\tau \qquad (1)$$

which states the soundness of the analysis.

Moreover, we aim at an analysis which is precise enough to guarantee, on a large class of types and for a large class of queries, that whenever the path semantics is empty over all possible instances of the input DTD, then the inferred type τ is empty, as well:

$$\forall t\in_{\mathfrak{I}} E.\ \mathfrak{I}([\![\mathit{Path}]\!]_t(\mathit{RootId(t)}))=\varnothing\ \Rightarrow\ \tau=\varnothing \qquad (2)$$

(the converse is a consequence of (1)). The precision described by (2) will then be used during the inference of type-projectors to discard elements that are useless in the evaluation of *Path*.

We start by inferring types for single-step paths.

DEFINITION 4.1 (SINGLE STEP TYPING). *Let E be a* DTD *and $\tau\subseteq DN(E)$, then:*

$$\begin{array}{rcl}
\boldsymbol{A}_E(\tau,\texttt{ancestor}) & = & \bigcup_{Y\in\tau}\{Z \mid Z\Rightarrow_E^+ Y\} \\
\boldsymbol{A}_E(\tau,\texttt{child}) & = & \bigcup_{Y\in\tau}\{Z \mid Y\Rightarrow_E Z\} \\
\boldsymbol{A}_E(\tau,\texttt{parent}) & = & \bigcup_{Y\in\tau}\{Z \mid Z\Rightarrow_E Y\} \\
\boldsymbol{A}_E(\tau,\texttt{descendant}) & = & \bigcup_{Y\in\tau}\{Z \mid Y\Rightarrow_E^+ Z\} \\
\boldsymbol{A}_E(\tau,\texttt{self}) & = & \tau \\
\boldsymbol{T}_E(\tau,a) & = & \{Y \mid Y\in\tau,\ E(Y)=a[r]\} \\
\boldsymbol{T}_E(\tau,\texttt{node}) & = & \tau \\
\boldsymbol{T}_E(\tau,\texttt{text}) & = & \{Y \mid Y\in\tau,\ E(Y)=\mathit{String}\}
\end{array}$$

The type of a single step query *Axis* :: *Test* for the DTD (X,E) is then given by $\mathbf{T}_E(\mathbf{A}_E(\{X\},\mathit{Axis}),\mathit{Test})$. Soundness of this definition, i.e. property (1), is given by the following lemma.

LEMMA 4.2. *Let t be a tree $\mathfrak{I}$-valid with respect to the* DTD *E. For every $S\subseteq \mathit{Ids}(t)$ and type τ, if $\mathfrak{I}(S)\subseteq\tau$ then*
1. $\mathfrak{I}([\![\mathit{Axis}]\!]_t(S))\subseteq\boldsymbol{A}_E(\tau,\mathit{Axis})$
2. $\mathfrak{I}(S ::_t \mathit{Test})\subseteq\boldsymbol{T}_E(\tau,\mathit{Test})$

Primitive Single Step

$$\frac{}{\Sigma \vdash_E \mathit{Axis} :: \texttt{node} :\ (\mathbf{A}_E(\Sigma_\tau, \mathit{Axis})\ ,\ \Sigma_\kappa \cup \mathbf{A}_E(\Sigma_\tau, \mathit{Axis}))}\ \mathit{Axis} \in \{\texttt{self},\ \texttt{child},\ \texttt{descendant}\}$$

$$\frac{}{\Sigma \vdash_E \mathit{Axis} :: \texttt{node} :\ (\mathbf{A}_E(\Sigma_\tau, \mathit{Axis})) \cap \Sigma_\kappa\ ,\ \mathbf{A}_E(\Sigma_\kappa, \mathit{Axis}) \cap \Sigma_\kappa)}\ \mathit{Axis} \in \{\texttt{parent},\ \texttt{ancestor}\}$$

$$\frac{}{\Sigma \vdash_E \texttt{self} :: \mathit{Test} :\ (\mathbf{T}_E(\Sigma_\tau,\ \mathit{Test})\ ,\ (\Sigma_\kappa \cap \mathbf{A}_E(\mathbf{T}_E(\Sigma_\tau,\ \mathit{Test}), \texttt{ancestor})) \cup \mathbf{T}_E(\Sigma_\tau,\ \mathit{Test}))}\ \mathit{Test} \neq \texttt{node}$$

$$\frac{\forall X_i \in \Sigma_\tau, P_j \in \mathit{Cond}\ , \quad (\{X_i\}, \Sigma_\kappa) \vdash_E P_j :\ \Sigma^{ij}}{\Sigma \vdash_E \texttt{self} :: \texttt{node}[\mathit{Cond}] :\ (\tau\ ,\ (\Sigma_\kappa \cap \mathbf{A}_E(\tau, \texttt{ancestor})) \cup \tau)}\ \tau = \{X_i \mid \exists j. \Sigma^{ij}_\tau \neq \varnothing\}$$

Encoded Single Step

$$\frac{\Sigma \vdash_E \mathit{Axis} :: \texttt{node}/\texttt{self} :: \mathit{Test} :\ \Sigma'}{\Sigma \vdash_E \mathit{Axis} :: \mathit{Test} :\ \Sigma'}\ \begin{array}{c}\mathit{Test} \neq \texttt{node} \\ \wedge \\ \mathit{Axis} \neq \texttt{self}\end{array} \qquad \frac{\Sigma \vdash_E \mathit{Axis} :: \mathit{Test}/\texttt{self} :: \texttt{node}[\mathit{Cond}] :\ \Sigma'}{\Sigma \vdash_E \mathit{Axis} :: \mathit{Test}[\mathit{Cond}] :\ \Sigma'}\ \begin{array}{c}\mathit{Test} \neq \texttt{node} \\ \vee \\ \mathit{Axis} \neq \texttt{self}\end{array}$$

Composed paths

$$\frac{\Sigma \vdash_E \mathit{Step} :\ \Sigma'' \quad \Sigma'' \vdash_E \mathit{Path} :\ \Sigma'}{\Sigma \vdash_E \mathit{Step}/\mathit{Path} :\ \Sigma'}$$

Figure 1: Inference rules for single step queries

The presence of upward axes makes the typing of composed paths much more difficult. To ensure precision, i.e. property (2), we have to be careful in dealing with DTDs in which an element may occur in the content of different elements. The naive solution consisting of inferring a type for composed paths by composing the functions we just defined for single steps, works only in the absence of upward axes. This can be illustrated by an example. Consider the following DTD rooted at X:

$$\{X \to c[Y,\ Z],\ Y \to a[W, \mathit{String}],\ Z \to b[\mathit{String}],\ W \to d[Y?]\}$$

and observe that Y occurs in two different element content definitions. If we consider the path $\texttt{self} :: c/\texttt{child} :: a/\texttt{parent} :: \texttt{node}$ over documents of the above DTD, then the precise type that this path should have is $\{X\}$. However, by using Definition 4.1 we end up with $\{X, W\}$. This is because the first step selects $\{Y\}$ and then, according to Definition 4.1, the second step selects $\{X, W\}$, as Y is in the content definition of these two names.

To solve this problem we introduce particular types, called *contexts*, to be updated at each step and containing names already encountered in previous steps. We then use them to refine type inference for upward axes. In the previous example, when typing the first step we build a *context* $\{X, Y\}$ indicating that for the moment the two names are the only ones visited by the traversal. Then, we use Definition 4.1 to type `parent` thus obtaining $\{X, W\}$, as before, but this time we intersect it with the context thus obtaining the precise answer $\{X\}$.

This idea is formalised by the (deterministic) type system of Figure 1. We use the meta-variables τ to range over types and κ over contexts, both denoting sets of names defined by the input DTD E. An environment, ranged over by Σ, is a pair (τ, κ); we use Σ_τ and Σ_κ to denote the first and second projection of Σ, respectively.

Environments	Σ	$::=$	(τ, κ)
Judgements	J	$::=$	$\Sigma \vdash_E \mathit{Path} :\ \Sigma$

The judgement $(\tau_c, \kappa_c) \vdash_E \mathit{Path} :\ (\tau_r, \kappa_r)$ means that given a DTD E, starting from the names in τ_c and the current context κ_c, the path *Path* generates the names τ_r in an updated context κ_r.

An environment (τ, κ) is well-formed with respect to E, if $\tau \subseteq DN(E)$, and $\kappa \subseteq \tau \cup \mathbf{A}_E(\tau, \texttt{ancestor})$, that is, if the context contains only names that occur in chains ending with names in τ. A judgement $\Sigma \vdash_E \mathit{Path} :\ \Sigma'$ is well formed if both Σ and Σ' are well formed with respect to E. It is easy to see that the type inference rules of Figure 1 preserve well-formedness.

The rules are relatively simple to understand. The first two rules implement our main idea: when we follow an axis *Axis*, we compute the type by $\mathbf{A}_E(\Sigma_\tau, \mathit{Axis})$; if the axis is a downward one, then we add this type to the current context, otherwise if the axis is an upward one, then we intersect it with the current context (both for the type part and for the context part). The rule for $\texttt{self} :: \mathit{Test}$ is slightly more difficult since it discards from the current set of nodes those that do not satisfy the test: the type is computed by $\mathbf{T}_E(\Sigma_\tau, \mathit{Test})$, while the context is obtained by erasing all the names that were in there just because they generated one of the discarded nodes; to do it it generates (the type of) all ancestors of the nodes satisfying the test, and intersects them with the current context. These first three rules are enough to type all the paths of the form $\mathit{Axis} :: \mathit{Test}$ since, as stated by the fifth typing rule, all remaining cases are encoded as $\mathit{Axis} :: \texttt{node}/\texttt{self} :: \mathit{Test}$. The fourth rule is the most difficult one: recall that *Cond* is a disjunction of *simple* paths; the type τ is obtained by discarding from Σ_τ all (names of) nodes for which *Cond* never holds; thus for each X_i in Σ_τ we compute the type of all the paths in *Cond*, and keep in τ only names for which at least one path may yield a non-empty result; the context then is computed as in the third rule, by discarding from the context all names that generated only names discarded from Σ_τ. Once more, all the remaining cases of conditional steps are encoded by this one, as stated by the sixth rule. Finally, step composition is dealt as a logical cut.

The type system is sound. It is also complete for DTDs that are $*$-guarded, non-recursive, and parent-unambiguous. Intuitively, a DTD is $*$-guarded when every union occurring in its productions is guarded by $*$ (or by +), it is non recursive if the depth of all documents validating it is bound, while it is parent-unambiguous if no name types both the parent and a strict ancestor of the parent of another name. Formally, we have the following definition

DEFINITION 4.3. *Let (X,E) be a* DTD.

1. *E is* $*$-guarded *if for each $Y \to l[r]$ in E, the regular expression is a product $r = r_1,\dots,r_n$ and whenever r_i contains a union, then $r_i = (r')*$;*
2. *E is* non-recursive *if it is never the case that $Y \Rightarrow^+_E Y$, for any name $Y \in DN(E)$;*
3. *E is* parent-unambiguous *if for all chains c and names Y,Z such that $cYZ \in Chains_{(X,E)}(X)$ the following implication*
$$cYc'Z \in Chains_{(X,E)}(X) \quad \Longrightarrow \quad c' = \varepsilon$$
holds (ε denotes the empty chain).

Non-recursivity and $*$-guardedness are properties enjoyed by a large number of commonly used DTDs. As an example, the reader can consider the DTDs of the XML Query Use Cases [3]: among the ten DTDs defined in the Use Cases, seven are both non-recursive and $*$-guarded, one is only $*$-guarded, one is only non-recursive, and just one does not satisfy either property. Furthermore our personal experience is that most of the DTDs available on the web are $*$-guarded. Concerning the parent-unambiguous property, although DTDs satisfying this property are less frequent (five on the ten DTDs in [3]), its absence is in practice not very problematic since, as we will see, only the presence of the `parent` axis may hinder completeness.

THEOREM 4.4 (SOUNDNESS AND COMPLETENESS). *Let (X,E) be a* DTD *and P a path. If $(\{X\},\{X\}) \vdash_E P : (\tau,\kappa)$ then (soundness):*
$$\tau \supseteq \bigcup_{t\in\Im E} \Im(\llbracket P \rrbracket_t(RootId(t)))$$
Furthermore, if (X,E) is $*$-guarded and non-recursive, and parent-unambiguous , then we also have (completeness):
$$\tau \subseteq \bigcup_{t\in\Im E} \Im(\llbracket P \rrbracket_t(RootId(t)))$$

To see why completeness does not hold in general consider the following DTD rooted at X and which is recursive and not $*$-guarded
$$\{X \to c[Y \mid Z],\ Y \to a[Y*,String],\ Z \to b[String]\}$$
and the following two queries $\texttt{self} :: c[\texttt{child} :: a]/\texttt{child} :: b$ and $\texttt{self} :: c/\texttt{child} :: a/\texttt{parent} :: node$. The type inferred for the first query contains both Y and Z. These are useless since the query is always empty. This is due to the non $*$-guarded union $Y \mid Z$: if we had $(Y \mid Z)*$ instead, then the query might yield a non-empty result, therefore Y and Z must correctly (and completely) be in the query type. The second query shows the reason why completeness does not hold in presence of recursion and backward axes (recursion with only forward axes does not pose any problem for completeness). The type of the second query should be $\{X\}$, but instead the type $\{X,Y\}$ is inferred. This is due to the recursion $Y \to a[Y*,\dots]$: since $Y \Rightarrow_E Y$, once Y is reached it is kept in the inferred type for every backward step.[4]

For queries over parent-ambiguous DTDs, completeness does not hold because the fourth rule in Figure 1—the one defined for $\texttt{self} :: \texttt{node}[Cond]$—is not precise for the parent axis. For instance, consider the following DTD rooted at X
$$\{X \to a[Y,Z],\ Y \to b[Z],\ Z \to c[\,]\}$$
and the query $\texttt{self} :: a/\texttt{child} :: b/\texttt{child} :: c/\texttt{parent} :: \texttt{node}$. The precise type of this query should be $\{Y\}$. However, the inferred type is $\{X,Y\}$. This is because the last step $\texttt{parent} :: \texttt{node}$ is typed with the context $\{X,Y,Z\}$ and this contains $\mathbf{A}_E(\{Z\},\texttt{parent}) = \{X,Y\}$. Here Z is the type for the c node selected by $\texttt{child} :: c$ and the $\mathbf{A}_E(,)$ operator assigns it $\{X,Y\}$ as parent type, even if the *real* parent type for Z in this case should be $\{Y\}$. Hence, the intersections operated by the type rule for `parent` are not powerful enough to guarantee precision for cases like this one. In a nutshell, this happens because in the presence of parent-ambiguous DTDs the type analysis may produce contexts containing false parent types (with respect the current type τ). This suggests that to be extremely precise, instead of sets of names, contexts should rather be sets of *chains* of names, computed and opportunely managed by the type analysis. However (i) managing sets of chains instead of simple sets of names dramatically complicates the treatment, due to recursive axes like `descendant`, (ii) the problem may arise only for queries that use parent axis and the concomitance of parent-ambiguity make the event rare in practice, and (iii) the loss of precision looks in most cases negligible. Therefore we considered that such a small gain (remember that completeness is just some icing on the cake since while it helps to gauge the precision of the approach its absence does not hinder its application) did not justify the dramatic increase in complexity needed to handle this case.

Note also that the type system, hence the completeness result, is stated for predicates of the form described in Section 3.2, therefore it does not account for the approximations introduced in Section 3.3. However very few non-structural conditions can be expressed at the level of types, so the impact of these approximations on completeness is very light.

4.2 Type-Projection inference

In this section we use the type inference of the previous section to infer type-projectors. Once more naive solutions do not work. For instance, for simple paths $Step_1/\dots/Step_n$, we may consider as type projector with respect to (X,E) the set $\bigcup_{i=1\dots n}\tau_i \cup \{X\}$, where for $i = 1\dots n$:
$$(\{X\},\{X\}) \vdash_E Step_1/\dots/Step_i : (\tau_i, -)$$
(we use "$-$" as a placeholder for uninteresting parameters). This definition is sound but not precise at all, as can be seen by considering $\texttt{descendant} :: \texttt{node}/Path$: the use of the above union yields a set containing τ_1 defined as
$$(\{X\},\{X\}) \vdash_E \texttt{descendant} :: \texttt{node} : (\tau_1, -)$$
that is, all descendants of the root X (no pruning is performed). Instead, we would like to discard, at least, all names that are descendants of X but that are not ancestors of a node matching *Path*. These are the names $Y \in \mathbf{T}_E(\mathbf{A}_E(\{X\},\texttt{descendant}),\texttt{node})$ such that
$$(\{Y\},\kappa) \vdash_E \texttt{descendant} :: \texttt{node}/Path : (\varnothing,\ -)$$
for some appropriate context κ. A similar reasoning applies to `ancestor`.

Such a selection is performed by the inference rules of Figure 2. For paths formed by a single step, if the step has no condition (first rule), then the type inference of the previous section is enough; otherwise (second rule) the step is transformed into a complex path (a simple trick to avoid the definition of several rules). Thanks to the third rule the type inference can work on just one node at a time, and thanks to the fourth and fifth rule, it just analyses paths whose

[4]The techniques developed in [11, 10] can be adapted to recover completeness for cases like the first query, while a more sophisticated type analysis could solve the problem with the second. In view of the precision of the current approach this is not a priority and we leave this investigation as future work.

Base and induction

$$\frac{\Sigma \vdash_E \mathit{Step} :\ (\tau,\kappa)}{\Sigma \Vdash_E \mathit{Step} : \tau\cup\kappa} \qquad \frac{\Sigma \Vdash_E \mathit{Step}[\mathit{Cond}]/\mathtt{self}::\mathtt{node} : \tau}{\Sigma \Vdash_E \mathit{Step}[\mathit{Cond}] : \tau} \qquad \frac{(\{X_1\},\kappa)\Vdash_E P:\tau_1 \quad \cdots \quad (\{X_n\},\kappa)\Vdash_E P:\tau_n}{(\{X_1,\ldots,X_n\},\kappa)\Vdash_E P : \bigcup_{i=1..n}\tau_i}\ \text{if no other rule applies}$$

Encoded Rules

$$\frac{\Sigma \Vdash_E \mathit{Axis}::\mathtt{node}/\mathtt{self}::\mathit{Test}/P : \tau}{\Sigma \Vdash_E \mathit{Axis}::\mathit{Test}/P : \tau}\ \begin{array}{c}\mathit{Test}\neq\mathtt{node}\\ \wedge\\ \mathit{Axis}\neq\mathtt{self}\end{array} \qquad \frac{\Sigma \Vdash_E \mathit{Axis}::\mathit{Test}/\mathtt{self}::\mathtt{node}[\mathit{Cond}]/P : \tau}{\Sigma \Vdash_E \mathit{Axis}::\mathit{Test}[\mathit{Cond}]/P : \tau}\ \begin{array}{c}\mathit{Test}\neq\mathtt{node}\\ \vee\\ \mathit{Axis}\neq\mathtt{self}\end{array}$$

Primitive Rules

$$\frac{(\{Y\},\kappa)\vdash_E \mathtt{self}::\mathit{Test}:\ \Sigma \quad \Sigma\Vdash_E P:\tau}{(\{Y\},\kappa)\Vdash_E \mathtt{self}::\mathit{Test}/P : \{Y\}\cup\tau} \qquad \frac{(\{Y\},\kappa)\vdash_E \mathtt{self}::\mathtt{node}[P_1\mathtt{or}\ldots\mathtt{or}P_n]:\ \Sigma \quad \Sigma\Vdash_E P:\tau \quad \Sigma\Vdash_E P_i:\tau_i}{(\{Y\},\kappa)\Vdash_E \mathtt{self}::\mathtt{node}[P_1\mathtt{or}\ldots\mathtt{or}P_n]/P : \{Y\}\cup\tau\cup\tau_1\cup\cdots\cup\tau_n}\ n\geq 1$$

$$\frac{(\{Y\},\kappa)\vdash_E \mathit{Axis}::\mathtt{node}:\ (\{X_1,...,X_n\},\kappa') \quad (\{X_i\},\kappa')\vdash_E P:\ \Sigma^i \quad (\tau,\kappa')\Vdash_E P:\tau'}{(\{Y\},\kappa)\Vdash_E \mathit{Axis}::\mathtt{node}/P : \{Y\}\cup\tau\cup\tau'}\ \begin{array}{l}\mathit{Axis}\in\{\mathtt{parent},\mathtt{child}\}\\ \tau=\{X_i \mid \Sigma^i_\tau\neq\varnothing\}\end{array}$$

$$\frac{(\{Y\},\kappa)\vdash_E \mathtt{desc}::\mathtt{node}:\ (\{X_1,...,X_n\},\kappa') \quad (\{X_i\},\kappa')\vdash_E \mathtt{desc}::\mathtt{node}/P:\ \Sigma^i \quad (\tau,\kappa')\Vdash_E \mathtt{child}::\mathtt{node}/P:\tau'}{(\{Y\},\kappa)\Vdash_E \mathtt{desc}::\mathtt{node}/P : \tau\cup\tau'}\ \tau=\{X_i \mid \Sigma^i_\tau\neq\varnothing\}\cup\{Y\}$$

$$\frac{(\{Y\},\kappa)\vdash_E \mathtt{ancs}::\mathtt{node}:\ (\{X_1,...,X_n\},\kappa') \quad (\{X_i\},\kappa')\vdash_E \mathtt{ancs}::\mathtt{node}/P:\ \Sigma^i \quad (\tau,\kappa')\Vdash_E \mathtt{parent}::\mathtt{node}/P:\tau'}{(\{Y\},\kappa)\Vdash_E \mathtt{ancs}::\mathtt{node}/P : \tau\cup\tau'}\ \tau=\{X_i \mid \Sigma^i_\tau\neq\varnothing\}\cup\{Y\}$$

Figure 2: Projectors inference rules (where `ancs` and `desc` are shorthands for `ancestor` and `descendant`)

components have one of the following three forms: (i) `self`::*Test*, (ii) `self`::`node`[*Cond*], or (iii) *Axis*::`node`. These three cases are handled by the "Primitive Rules" of Figure 2: The first rule handles the case (i) simply by collecting the current context. The second rule handles the case (ii), by collecting besides the context also all the parts that are necessary to compute the condition (which in the rule is expanded in its more general form); the case (iii) is handled by the last three rules which are nothing but slight variations of the same rule according to the particular axis taken into account: each rule infers the type τ obtained by discarding from the type $\{X_1,...,X_n\}$ of the step, all names that are useless for the rest of the path, and then uses this τ to continue the inference of the projector.

THEOREM 4.5 (SOUNDNESS OF PROJECTOR INFERENCE). *Let (X,E) be a* DTD *and P a path. If $(\{X\},\{X\})\Vdash_E P:\tau$, then τ is a type projector for (X,E) and for every $t\in_{\mathfrak{Z}} E$*

$$[\![P]\!]_{t\backslash_{\mathfrak{Z}}\tau}(\mathit{RootId}(t)) = [\![P]\!]_t(\mathit{RootId}(t))$$

The above theorem states that executing the query P on a tree t returns the same set of nodes as executing it on $t\backslash_{\mathfrak{Z}}\tau$ the tree t pruned by the inferred projector. From a practical perspective it is important to notice that according to standard XPath semantics, the semantics of a query contains *only* the nodes of the result of the query not their sub-trees. The latter may thus be pruned by the inferred projector. Therefore, if we want to *materialise* the result of a query we must not cut these nodes, and rather use the projection $\tau=\tau'\cup\mathbf{A}_E(\tau'',\mathtt{descendant})$ where $(\{X\},\{X\})\Vdash_E P:\tau'$ and $(\{X\},\{X\})\vdash_E P:\ (\tau'';-)$.

Completeness requires not only completeness of the type system (thus, $*$-guarded, non-recursive, and parent-unambiguous DTDs), but also the following condition on queries:

DEFINITION 4.6. *An XPath query Q is* strongly-specified *if (i) its predicates do not use backward axes, (ii) along Q and along each path in the predicates of Q there are no two consecutive (possibly conditional) steps whose Test part is* `node`, *and (iii) each predicate in Q contains at most one path and this does not terminate by a step whose Test is* `node`.

For instance, among the following queries, only the first two are strongly-specified.

1. `descendant::node/self::a /ancestor::node`
2. `descendant::node[child::b]/self::a/parent::node`
3. `descendant::node/ancestor::node/self::a`
4. `descendant::node[child::b/child::node]/self::a`
4. `child:: a [descendant::node/parent:: b]/child::c`

Once more, we are in presence of a very common class of queries: for instance, almost all paths in the XMark and XPathMark benchmarks are strongly specified.

THEOREM 4.7 (COMPLETENESS OF PROJECTOR INFERENCE). *Let (X,E) be a $*$-guarded, non-recursive, and parent-unambiguous* DTD, *and P a strongly-specified path. If $(\{X\},\{X\})\Vdash_E P:\tau$, then there exists $t\in_{\mathfrak{Z}} E$ such that for each $Y\in\tau$, if $\pi=\tau\setminus(\{Y\}\cup\mathbf{A}_E(\{Y\},\mathtt{descendant}))$, then*

$$[\![P]\!]_{t\backslash_{\mathfrak{Z}}\pi}(\mathit{RootId}(t)) \neq [\![P]\!]_t(\mathit{RootId}(t))$$

The fact that completeness may not hold for not $*$-guarded, non-recursive, or parent-ambiguous DTDs, is a consequence of the analogous property of the type system. To see that also strong-specification is a necessary condition consider documents valid with respect to the following DTD rooted at X:

$$\{X\to a[Y,W],\ W\to c[\,],Y\to b[Z],\ Z\to d[\,]\}.$$

Query them by the following query which not strongly-specified since it does not satisfy condition (ii) of Definition 4.6

$$\texttt{self} :: a[\texttt{child} :: \texttt{node}].$$

$\{X,Y\}$ is an optimal projector for this query, but the presence of the condition `self :: node` makes the system to include also W in the inferred projector, thus breaking completeness. Concerning the presence of backward axes in predicates, consider the query $\texttt{self} :: a[\texttt{descendant} :: \texttt{node}/\texttt{ancestor} :: a]$ which does not satisfy condition (i). An optimal projector for this query on the same DTD is $\{X,Y\}$. However, since the `ancestor` condition is true for all descendants of a nodes, $\{W,Z\}$ is included in the projector as well. Finally, it is straightforward to check that the query $\texttt{self} :: a[\texttt{child} :: b \texttt{ or } \texttt{child} :: c]$, which does not satisfy condition (iii), is not complete for the same DTD.

Of course, it is possible to state completeness for other classes of queries but, once more, this seems an excellent compromise between simplicity and generality.

THEOREM 4.8 (DECIDABILITY). *Given a path P, a DTD E, and an environment Σ well-formed with respect to E, the inference of a context Σ' and a type τ such that $\Sigma \vdash_E P : \Sigma'$ and $\Sigma \Vdash_E P : \tau$ is decidable.*

4.3 Adding sibling, preceding and following axes.

We could deal with the missing XPath axes by adding specific inference rules. Instead we opt to use an approximation of these axes in term of the previous ones, since it appears as the best compromise between simplicity and efficiency.

The approximation is performed by two logical rewriting passes. In the first pass we rewrite preceding and following axes as specified in the W3C specifications [4]. Namely, we substitute each step *Axis* :: *Test* with $Axis \in \{\texttt{preceding}, \texttt{following}\}$ by the following equivalent path $\texttt{ancestor-or-self} :: \texttt{node}/(Axis\texttt{-sibling}) :: \texttt{node}/\texttt{descendant-or-self} :: Test$

The second pass is the one which introduces the approximation since it replaces all steps of the form *Axis*::*Test* with $Axis \in \{\texttt{preceding-sibling}, \texttt{following-sibling}\}$ by the path $\texttt{parent}::\texttt{node}/\texttt{child}::Test$.

Clearly, the static analysis of the approximation yields a less precise projection than the one we could obtain by working directly on the original query. However, we still achieve good precision of pruning in practice as we will show in Section 6. For instance, by applying the above rewriting to XPathMark queries Q9 and Q11, we were able to prune a document down to 7.5% of its original size.

5. EXTENSION TO XQUERY

In this section we extend the technique to XQuery. More precisely to the FLWR core of XQuery described by the following grammar:

$$\begin{array}{lcl} q & ::= & () \mid q,q \mid \langle tag\rangle q\langle /tag\rangle \mid Exp \\ & \mid & \texttt{for } x \texttt{ in } q \texttt{ return } q \mid \texttt{let } x = q \texttt{ return } q \\ & \mid & \texttt{if } q \texttt{ then } q \texttt{ else } q \end{array}$$

where the definition of *Exp* (given in Section 3.3) is extended with variables, and with generic XPath expressions Q of Section 3.3 that can be rooted at a variable or at `/` :

$$Exp ::= x \mid Q \mid x/Q \mid /Q \mid Exp\, op\, Exp \mid f(Exp,..,Exp) \mid AExp$$

Without loss of generality, we assume that FLWR expressions do not occur in `if`-conditions nor in predicates (every query can be put into this form by adding appropriate `let`-expressions). Also, we do not consider either queries which first construct new elements and then navigate on them (these are rarely used in practice), nor those containing XQuery clauses like `order_by`, `switch_case`, etc.: our approach can be easily extended to both cases.

In order to apply the previous analysis to infer a projector for q, we first extract a set of XPath$^\ell$ expressions from q, denoting the data needs for q. This set of paths is extracted from the query by the extraction function $\boldsymbol{E}$, whose definition is given in Figure 3. The extraction function has the form $\boldsymbol{E}(q,\Gamma,m)$. The first parameter is the query at issue. The second parameter Γ is an environment that keeps track of bindings of the form $(x;\ \text{for } P)$ or $(x;\ \text{let } P)$, whose scope q is in (see the definition of Γ' in the last two lines of Figure 3, and observe, by a simple induction reasoning, that environments contain paths already in XPath$^\ell$). Finally, m is a flag indicating whether q is a query that serves to materialise a partial or final result ($m = 1$), or that just selects a set of nodes whose descendants are not needed ($m = 0$). Thus, the set of path expressions (possibly containing qualifiers) extracted from a top-level query q is $\boldsymbol{E}(q,\varnothing,1)$.

Once the set of paths are extracted from a query q, we use it to infer a projector for q according to rules in Section 4.2. Formally, for each P_i extracted from q we deduce a projector π_i, and use for the whole q the union of these projectors (projectors are closed by union). Also, note that the extracted path of a closed query will not contain free variables since possible free variables are persistent roots that must be solved before the analysis.

Most of the rules in Figure 3 are not difficult to understand, therefore only few of them deserve further commentary. The flag is needed since each path determining the result ($m = 1$) must be extended with `descendant-or-self`, in order to project on all nodes needed in the query result. This is done by the lines 6, 8, and 10 of the definition. Expressions are dealt in a way similar to the path extractor $\boldsymbol{P}$ of Section 3.3; the extractor $\boldsymbol{P}$ itself is used in line 12 to produce simple paths (where we used the notation $\texttt{or}(\{P_1,...,P_n\})$ for $P_1 \texttt{or} \ldots \texttt{or} P_n$, and omitted the—straightforward—rules for single step paths). Also note that when a result is computed (lines 2 and 5) paths in "for"-environments are added ("let" are added only if their binding variable is used).

These rules subsume and enhance the whole Marian and Siméon's technique [14]. In particular, (i) the technique we use to exclude useless intermediate paths is simpler and more compact, (ii) we do not need to distinguish between two kinds of extracted paths but, more simply, we always manage a unique set of path expressions, and (iii) last but not least, our path extractor can be used even if the user cannot access an XQuery to XQuery-Core compiler, which is necessary for [14].

Before applying the extraction function $\boldsymbol{E}$ to a query q we apply some heuristics that rewrite q so to improve the pruning capability of the inferred paths. Among these heuristics the most important is the one that rewrites

```
for y in Q/descendant-or-self::node
  return if C(y) then q else ()
```

into

```
for y in
  Q/descendant-or-self::node[C(self :: node)]
return q
```

whenever $C(y)$ is a condition referring only to y and does not use external functions ($C(\texttt{self} :: \texttt{node})$ is obtained by replacing `self :: node` for all occurrences of y free in C). If we apply $\boldsymbol{E}$ to the first query, then a path ending by `descendant-or-self::node` is extracted thus annulling further pruning: the entire forest selected

$$\begin{array}{rlcl}
1. & \boldsymbol{E}((),\Gamma,m) & = & \varnothing \\
2. & \boldsymbol{E}(AExp,\Gamma,1) & = & \{P \mid (x;\ \text{for } P) \in \Gamma\} \\
3. & \boldsymbol{E}(AExp,\Gamma,0) & = & \varnothing \\
4. & \boldsymbol{E}((q_1\texttt{,}q_2),\Gamma,m) & = & \boldsymbol{E}(q_1,\Gamma,m) \cup \boldsymbol{E}(q_2,\Gamma,m) \\
5. & \boldsymbol{E}(\text{<}tag\text{>}q\text{</}tag\text{>},\Gamma,m) & = & \{P \mid (x;\ \text{for } P) \in \Gamma\} \cup \boldsymbol{E}(q,\Gamma,1) \\
6. & \boldsymbol{E}(x,\Gamma,1) & = & \{P/\texttt{descendant-or-self}::\texttt{node} \mid (x;\ -P) \in \Gamma\} \\
7. & \boldsymbol{E}(x,\Gamma,0) & = & \{P \mid (x;\ -P) \in \Gamma\} \\
8. & \boldsymbol{E}(/P,\Gamma,1) & = & \{/P/\texttt{descendant-or-self}::\texttt{node}\} \\
9. & \boldsymbol{E}(/P,\Gamma,0) & = & \{/P\} \\
10. & \boldsymbol{E}(x/P,\Gamma,1) & = & \{P'/P/\texttt{descendant-or-self}::\texttt{node} \mid (x;\ -P') \in \Gamma\} \\
11. & \boldsymbol{E}(Step/q,\Gamma,m) & = & Step/\boldsymbol{E}(q,\Gamma,m) \\
12. & \boldsymbol{E}(Step[Exp]/q,\Gamma,m) & = & Step[\texttt{or}(\boldsymbol{P}(Exp))]/\boldsymbol{E}(q,\Gamma,m) \\
13. & \boldsymbol{E}(Exp_1\ op\ Exp_2,\Gamma,m) & = & \boldsymbol{E}(Exp_1,\Gamma,m) \cup \boldsymbol{E}(Exp_2,\Gamma,m) \\
14. & \boldsymbol{E}(f(Exp_1,\ldots,Exp_n),\Gamma,m) & = & \bigcup_{i=1,n}(\boldsymbol{E}(Exp_i,\Gamma,0)/\boldsymbol{F}(f,i)) \cup \{\texttt{self}::\texttt{node}\} \\
15. & \boldsymbol{E}(\texttt{if}\ q\ \texttt{then}\ q_1\ \texttt{else}\ q_2,\Gamma,m) & = & \boldsymbol{E}(q,\Gamma,0) \cup \boldsymbol{E}(q_1,\Gamma,1) \cup \boldsymbol{E}(q_2,\Gamma,1) \cup \{P \mid (x;\ -P) \in \Gamma\} \\
16. & \boldsymbol{E}(\texttt{for}\ x\ \texttt{in}\ q_1\ \texttt{return}\ q_2,\Gamma,m) & = & \boldsymbol{E}(q_1,\Gamma,0) \cup \boldsymbol{E}(q_2,\Gamma \cup \Gamma',m) \quad (\text{where } \Gamma' = \{(x;\ \text{for } P) \mid P \in \boldsymbol{E}(q_1,\Gamma,0)\}) \\
17. & \boldsymbol{E}(\texttt{let}\ x\ \texttt{=}\ q_1\ \texttt{return}\ q_2,\Gamma,m) & = & \boldsymbol{E}(q_1,\Gamma,0) \cup \boldsymbol{E}(q_2,\Gamma \cup \Gamma',m) \quad (\text{where } \Gamma' = \{(x;\ \text{let } P) \mid P \in \boldsymbol{E}(q_1,\Gamma,0)\})
\end{array}$$

Figure 3: XQuery path extraction

by Q is loaded in main memory. This also happens with the approaches of Bressan *et al.* [9] and of Marian and Siméon [14]. In our and Marian and Siméon's approach the query can be rewritten as above (this is not possible in [9] since their subset of XQuery does not include predicates). However, Marian and Siméon's path based pruning degenerates (no further pruning is performed) also for the second query, since the `descendant-or-self::node` ends up in the set of pruner paths, thus selecting all nodes. This is because their approach cannot manage predicates. In our approach instead predicates are taken into account and therefore only nodes satisfying $C(y)$ are kept by the projector, thus yielding a very precise pruning.

It is important to stress that despite their specific form the first kind of queries is very common in practice since they are generated from XQuery→XQuery-Core compilation of a non negligible class of queries (for instance Q13 of the XPathMark) or when rewriting upward axes into downward ones. This latter observation shows that the application of rewriting rules rules of [15] to extend Marian and Siméon's approach to upward axes is not feasible since the rewriting may completely compromise pruning.

6. EXPERIMENTS

We have implemented a complete version of the algorithm defined for full XPath. The code (available at `http://www.lri.fr/~kn`) is written in OCaml, uses the PXP library for parsing XML documents, and its correctness was verified for all tests. After the path extraction of Section 5, it performs the rewriting presented in Sections 3.3 and 4.3, and the static analysis defined in Section 4. The latter is extended to deal with attributes, with the wildcard test `element()`, with `{descendant,ancestor}-or-self` and `{preceding,following}-siblings` axes, and with absolute paths. It also uses a couple of heuristics. One heuristic rewrites the DTD E so that every name Y defined as $Y \to String$ occurs exactly once in the right hand side of an edge of E; this enhances the precision of pruning by reducing the number of conflicts on the leaves of the tree. The other heuristic keeps track of the depth of elements in the paths in order to improve pruning, especially in presence of recursive DTDs (this latter heuristics could be embedded in the formal treatment, but we preferred to keep it simpler). Pruning is then performed *in streaming* and merely consists of a one-pass traversal of the document. We also added an optional validation option, that makes it possible to prune the document while validating it. Programs that use an external validator can therefore prune their document without any overhead.

We performed our tests on a GNU/Linux desktop, with 3GHz processor, 512 MB of RAM and a single S-ATA hard-drive, using DTDs, document generator, and queries of XMark and XPathMark (the latter is interesting because its queries use all the available axes). Queries were processed by the latest version of Galax (that is, the 0.5.0). Swap was disabled to test memory limits.

For what concerns the overhead of the optimisation, tests confirmed that it is always negligible, both in memory and time consumption: the only noticeable overhead is pruning time, which is linear in the size of the pruned document, but can be embedded in document parsing and/or validation (e.g., for 60MB documents computing the projector took around 0.5s while pruning and saving the pruned document to disk was always below 10s). These results were confirmed by further experiments on large DTDs (e.g. XHTML) and long XPath expressions (twenty steps or so).

In Table 1 we report part of the results of our tests. For space reasons just a selection of XMark (QM) and XPathMark (QP) queries are presented.

Projector efficiency. The fourth line of Table 1 reports the effect of inferred projectors and it is an indicator of the selectivity of the query. For several XMark queries the size of the pruned document is around 70-80% of the size of the original document. This is due to the fact that XMark documents contain mixed-content `<description>` elements which account for about 70% of the total size. Thus, queries whose execution requires the whole content of `<description>` elements, preserve a large part of the file. On the contrary, for very selective queries like QM06, 99.7% of the document is discarded. Finally, for queries that are very little selective, like QP13, the whole document has to be kept. It should be noted in Table 1, fourth line, that for all XMark queries but QM14 we could prune more than 95% of the original document.

Execution time and memory occupation. The comparison of performances of the Galax query engine on an original document and its pruned version is given in Figures 4 and 5, which respectively report the processing times and main memory occupation for documents of 56MB. They show that time and memory gains are

	QM03	QM06	QM07	QM14	QM15	QM19	QP01	QP02	QP03	QP04	QP05	QP06	QP07	QP08	QP09	QP10	QP11	QP12	QP13	QP21	QP23
Original Document Size (MB)	930	2048*	1100	202	2048*	964	112	313	258	291	123	190	168	123	459	123	369	134	79	224	403
Pruned Document Size(MB)	25	5,3	42	139	24	24	89	50	46	50	98	133	123	99	35	98	28	107	78	152	42
Main Memory Usage (MB)	374	90	380	512	245	512	391	399	433	434	418	485	467	466	466	483	456	460	504	459	465
Gain in Size (% of original)	2.5	0.3	3.4	69.6	1.15	2.5	80.4	15.7	17.5	16.8	80.4	69.6	73.2	80.4	7.5	80.4	7.5	80.4	98.2	67.9	10.4
Gain in Speed ($\times$ faster)	17.8	110.1	28.2	3.9	62.6	7.5	1.5	3.6	3.7	4.3	1.5	2.9	2.6	1.1	4.9	1.6	4.2	1.6	1.0	3.6	3.6

*: biggest file the XMark generator was able to produce.

Table 1: Sizes (in MBytes) of the biggest document processed thanks to pruning, size of its pruned version, and memory used to process the latter. Percent of the pruned document and speedup of the execution time for a 56MB document.

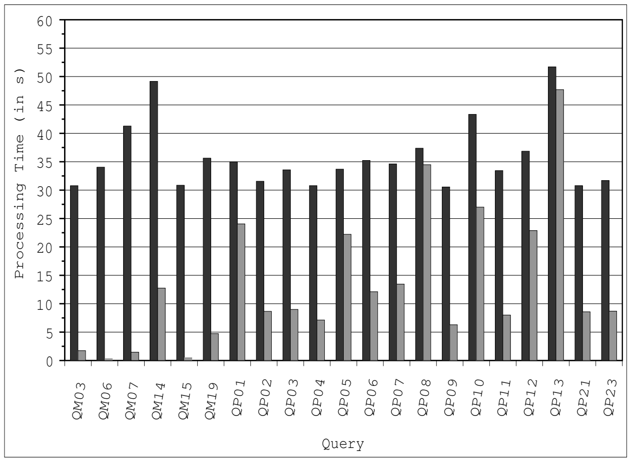

Figure 4: Processing time of a query on original (56MB) and pruned documents

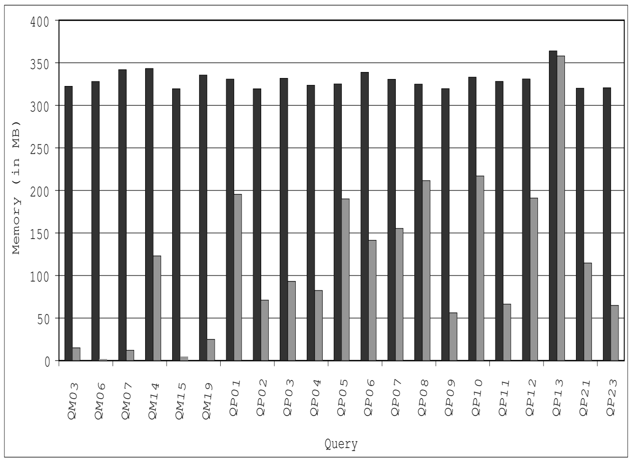

Figure 5: Memory used to process a query on original (56MB) and pruned documents

similar.

These gains translate in practice into much faster executions and the possibility to process much larger documents. The improvement can be measured by looking at the first and last lines of Table 1. The first line reports the size of the largest document it was possible to process thanks to pruning. This must be compared with the fact that, for all queries, the largest document that can be processed without pruning is 68MBytes large. The last line reports how many times the execution on a pruned document is faster than the execution on the original document. It is important to note that, depending on the nature of the query, the gain can be much higher than the proportion given by the percent of the size of the pruning. For instance, for queries such as QM14, QP6, and QP21 the size of the pruned document is two-thirds of the size of the original document, but they can then be processed from three to four times faster and, as Figure 5 shows, using three times less memory than when processed on the original. The latter is a huge gain when one knows that memory usage is one of the main bottlenecks for real life query processing (e.g., in DOM-based implementations of XPath or XSLT processors).

Quite informative, as well, is the data in the second line of Table 1 which reports, for each query, the size in MB of the maximum pruned document. It is interesting to see that, while the maximum size for an unpruned document is 68MB, we can process documents for which the projection has a size of 152MB (on disk). This is due to the fact that projecting a document not only reduces its size but also its *complexity* by reducing the number of types of nodes. This simplification of the document reduces the amount of extra-information the query engine has to keep for each node and, consequently, its memory usage. More precisely, the benefit of pruning out some (types of) nodes is twofold: first, the fan out of the document is reduced and this may impact memory usage for engines that chase sibling pointers and, second, the number of element names is reduced, which may reduce memory occupation when shredding.

These results are a clear-cut improvement over current technology. While we cannot directly compare processing performances since no implementation of the other pruning approaches is publicly available, we want to stress two points: (i) with one exception (QM14) the amount of pruning on common experiments is always equal or better with our approach than the others and (ii) performing pruning never is a bottleneck in our case thanks to fact that our solution consists of a single bufferless one pass traversal of the input document (on our 512MB machine we were able to efficiently prune arbitrary large documents, while in case of [14] pruning can end up using as much memory as the execution of the query).

7. CONCLUSION AND FUTURE WORK

The benchmarks show the clear advantages of applying our optimisation technique to query XML documents, and the characteristics of our solution make it profitable in all application scenarios. We discussed several aspects for which our approach improves the state of the art: for performances (better pruning, more speedup, less memory consumption), for the analysis techniques (linear pruning time, negligible memory and time consumption), for its generality (handling of all axes and of predicates), and, last but not least, for the formal foundation it provides (correctness formally proved, limits of the approach formally stated).

Future work will be pursued in three distinct areas: formal developments, database integration, and implementation issues.

For what concerns the formal treatment, we have to integrate in it the heuristics used in the implementation of the static analysis and to formally state the soundness and completeness of some approximations presented in the work. Also, it should be easy to adapt the approach to work in the absence of DTDs, by using dataguides/path-summaries instead. We intend also to adapt out technique to optimise queries written in ℂQL [7] the query language of ℂDuce [6]: as we said at the end of Section 3, their rich type system will allow us to assign more precise types to queries (for instance, it will be possible to capture by types many XPath predicates, since disjunction, conjunctions and negations can be handled by the corresponding type operators and the value of attributes and element contents can be expressed by singleton types) and thus to perform more selective pruning. Finally, we want to modify our approach so that it can yield efficient pruning also in the presence of XPath 2.0 predicates that test the XML Schema of nodes. Note indeed that such predicates are blockers for pruning: we have to leave the entire subtree intact so that the engine can verify that it has the specified schema. But since the projector inference algorithm already statically checks this property, the idea is to make the inference algorithm also rewrite predicates so as to push the schema tests down where they are strictly necessary, thus making further pruning possible.

From a database perspective we want to study the integration of our optimisation technique with classical database ones. Our technique must be viewed as a preliminary step that can be further combined with more traditional database optimisations. More precisely, as our technique is able to take into account the workload, in the line of [8], it could help the database administrator to deduce relevant clustering strategies of XML data on disk and to define well-adapted indexes and/or materialised views. Second, our pruning technique can also be used for pruning indexes. For example, if indexes over element tags are present before query processing (like in the TIMBER system), the index can be pruned as well. In TIMBER, for a 472 MB document, such an index can reach a 241MB size [16], thus it is worth being pruned, in order to improve buffer management and concurrent query evaluations.

Finally, implementation-wise, the natural extension of our work is to interface our pruning system with a query processing engine. This would bring several advantages: (i) the pruning overhead would be diluted in the parsing/validation phase and (ii) an interaction between the query engine and the loading module would provide a way not only to prune the document but to start answering the query in streaming, when possible.

Acknowledgements. We would like to thank Haiming Chen for pointing us an error in the two typing systems of a preliminary version of this work. This work benefitted from several discussions with and suggestions from Ioana Manolescu and Carlo Sartiani. Two of the three VLDB anonymous referees provided very useful feedback. This work was partially funded by the French ACI project "Transformation Langages for XML: Logics and Applications" (TraLaLA) and the French ACI young researcher project "WebStand".

8. REFERENCES

[1] Galax. `http://www.galaxquery.org`.

[2] XML Path Language (XPath) 2.0. `http://www.w3.org/TR/xpath20`.

[3] XML Query Use Cases. `http://www.w3.org/TR/xquery-use-cases/`.

[4] XQuery 1.0 and XPath 2.0 Formal Semantics. `http://www.w3.org/TR/xquery-semantics`.

[5] XQuery 1.0 and XPath 2.0 Functions and Operators. `http://www.w3.org/xquery-operators`.

[6] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *ICFP '03, 8th ACM Int. Conf. on Functional Programming*, pages 51–63, 2003.

[7] V. Benzaken, G. Castagna, and C. Miachon. A full pattern-based paradigm for XML query processing. In *PADL '05, the 7th Int. Symp. on Practical Aspects of Declarative Languages*, number 3350 in LNCS. Springer, 2005.

[8] V. Benzaken, C. Delobel, and G. Harrus. Clustering strategies in O_2: an overview. In *Building an Object-Oriented Database System: the Story of O_2*. Morgan Kaufman, 1992.

[9] S. Bressan, B. Catania, Z. Lacroix, Y-G Li, and A. Maddalena. Accelerating queries by pruning XML documents. *Data Knowl. Eng.*, 54(2):211–240, 2005.

[10] D. Colazzo. *Path Correctness for XML Queries: Characterization and Static Type Checking*. PhD thesis, Dip. di Informatica, Università di Pisa, 2004.

[11] D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. Types for Path Correctness for XML Queries. In *ICFP '04, 9th ACM Int. Conf. on Functional Programming*, 2004.

[12] M. Franceschet. XPathMark - An XPath benchmark for XMark generated data. In *XSym 2005, 3rd Int. XML Database Symposium*, LNCS n. 3671, 2005.

[13] D. Lee, M. Mani, and M. Murata. Reasoning about XML Schema Languages using Formal Language Theory. Technical report, IBM Almaden Research, 2000.

[14] A. Marian and J. Siméon. Projecting XML documents. In *VLDB '03*, pages 213–224, 2003.

[15] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *Proc. EDBT Workshop (XMLDM)*, volume 2490 of *LNCS*, pages 109–127. Springer, 2002.

[16] S. Paparizos and H.V. Jagadish. Pattern tree algebras: Sets or sequences? In *VLDB*, 2005.

[17] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *VLDB '02*, pages 974–985, 2002.