XQuery Layers

Daniele Braga Alessandro Campi Stefano Ceri Paola Spoletini

Politecnico di Milano - Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci 32, 20133 Milano, Italy

braga|campi|ceri|spoleti@elet.polimi.it

ABSTRACT

XML is becoming widespread as data interoperability standard in many application domains. An increasing number of researchers and professionals, who are not computer scientists (although they may have a strong technical background), needs to query and transform XML data during their working activities. Such tasks typically require simple queries and partial awareness of the XML data model, in the context of a given, domain-specific XML-based protocol.

The W3C community has proposed XQuery as the standard query language for XML [9]. XQuery has a huge expressive power - as it encompasses features belonging both to query and functional languages, but it may be considered as too complex for the above user profiles; well-designed subsets of XQuery are sufficient to satisfy their needs.

In this paper, we propose six layered subsets of XQuery, targeted to cover user communities with increasing needs. The initial layers are based upon XQBE (XQuery By Example), a visual XML query language, strictly less expressive than XQuery. We argue that the first layers are easier to learn and to master than the full language, also thanks to the availability of simple visual interfaces, and that these layers cover most of the needs of many user communities.

1. INTRODUCTION

XML-based data descriptions are used in a large variety of fields such as chemistry (CML), education (SCORM), e-commerce (FinXML), medicine (XChart), genetics (Genentech BLAST) and mathematics (MathML). Sometimes, the experts of these fields have excellent programming skill; more often, however, domain specialists don't have such a skill and they expect that XML (and XML-related technologies) will simplify their interaction with data without requiring a long training activity.

Moreover, many schemas in the aforementioned application domains correspond to flat data structures, for which the full expressive power of XQuery is oversized. When more complex data structures are in use, the typical data recombination and transformation tasks do not require advanced features of a query or transformation language.

At present, users have two alternatives: XPath, which is often not expressive enough, and full XQuery, which is typically too complex. In particular, the classical XQuery patterns (as shown, for instance, in the W3C use cases [15]) combine some basic intuitive data access primitives, such as selections and projections, with iterations and multiple variable bindings. While the former remind of a declarative approach to the task of data extraction, the latter are most often entangled to procedural programming. The combination of these two paradigms makes writing XQuery programs a difficult task. We will show that most query patterns within several real-world application domains do not require such a complex combination of heterogeneous primitives.

This paper defines six progressive language *layers*, designed in order to incrementally add expressive power (but also complexity). We do not suggest that XQuery engines should be diminished in their expressive power, but rather that their use should be enabled through simple user interfaces, acting as filters and presenting the constructs through some visual aids. Example-driven user interfaces, such as Zloof's QBE [16] supported by MS Access, have been successful in enhancing database accessibility by naive users.

Inspired by QBE, we designed a visual query language for XML, called XQBE [7], whose expressive power is strictly included into the expressive power of XQuery. We also defined a subset of XQBE, called CoreXQBE, carrying the most significant core concepts of the language, and therefore useful for formalising the language's semantics. We next extended XQBE to support views [6].

Starting from this background (and from our experience in teaching XQuery and XQBE), in this paper we propose these four levels, next denoted as SimpleXQBE, CoreXQBE, FullXQBE, and ViewXQBE, as "entry levels" to XQuery; they progressively introduce "visual concepts" into XQBE. Our fifth level is simply obtained from XQuery by subtracting functions and types, and the sixth level corresponds to the entire XQuery.

The paper is organised as follows. Section 2 overviews XQBE, then Section 3 presents the XQuery layers, Section 4 describes how layers are used in classical XML standards, and Section 5 overviews other approaches to XQuery subsetting.

2. XQBE IN A NUTSHELL

The syntax and semantics of XQBE are introduced here informally, with an example; a full description is in [7].

Consider the first query in the W3C use cases [15], which reads "*List books published by Addison-Wesley after 1991, including their year and title*". This query is shown in Figure 1. The basic interpretation of an XQBE query is that the XML fragments that match the *source* graph (the one on the left) are transformed into the fragments described by the *construction* graph (the one on the right). In XQBE all XML *elements* in the target document are represented as rectangles labeled with the tagname, their *attributes* are represented as black circles (with the attribute *name* on the

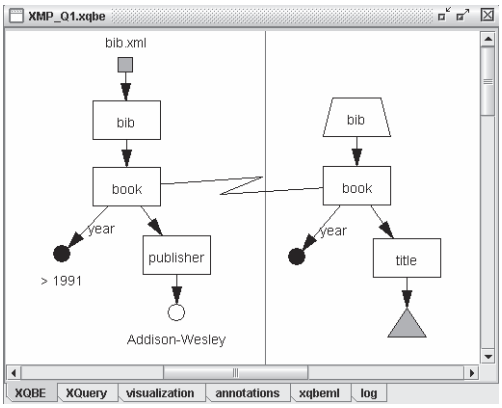

Figure 1: Q1 in XQBE

arc between the rectangle and the circle), and their PCDATA content is represented as an empty circle. Containment between XML items is represented by directed arcs, while black and empty circles may be labeled with conditions on the values they represent. A query always has a vertical line in the middle, that separates the source and construction graphs, so that the transformation has a "natural" reading order from left to right. The correspondence between the components of the two parts is expressed by explicit binding edges that cross the line and connect the nodes of the source part to the nodes that will take their place in the query result.

The core "conceptual" primitives of the language are *selection*, *iteration*, and *projection*, which are expressed by the source graph, the binding edges, and the construction graph respectively. Selection takes place by querying for the conjunction of all conditions upon values that label the graph, together with the structural constraints expressed by the graph topology. Iterations take place over those nodes in the source graph which are bound to the construction graph; according to the visual paradigm, the construction is iterated so as to build *as many* elements in the result *as* the qualified elements of the target document. Projection takes place in two "directions": in *depth*, as with the inclusion of descendants of the extracted elements, and in *breadth*, as with explicitly choosing which siblings are to be included in the result and which ones are to be pruned.

The XQBE query in Figure 1 extracts data from the document `bib.xml`: the source graph matches all the `book` elements with a `year` attribute whose value is greater than 1991 and a `publisher` subelement whose PCDATA content equals "Addison-Wesley". The XQuery equivalent is:

```
<bib> { for $b in doc("bib.xml")/bib/book
        where $b/publisher="Addison-Wesley" and $b/@year > 1991
        return <book year="{ $b/@year }">
                  { $b/title }
               </book>  }  </bib>
```

3. LAYERED XQUERY

The first four layers refer to XQBE, and therefore share its visual query paradigm. Figure 2 compares the first three layers, called SimpleXQBE, CoreXQBE, and FullXQBE. Intuitively, SimpleXQBE has one simple left-to-right binding, CoreXQBE has arbitrary bindings but uses few core concepts, FullXQBE adds more concepts enabling, e.g., aggregation, negation, and ordering.

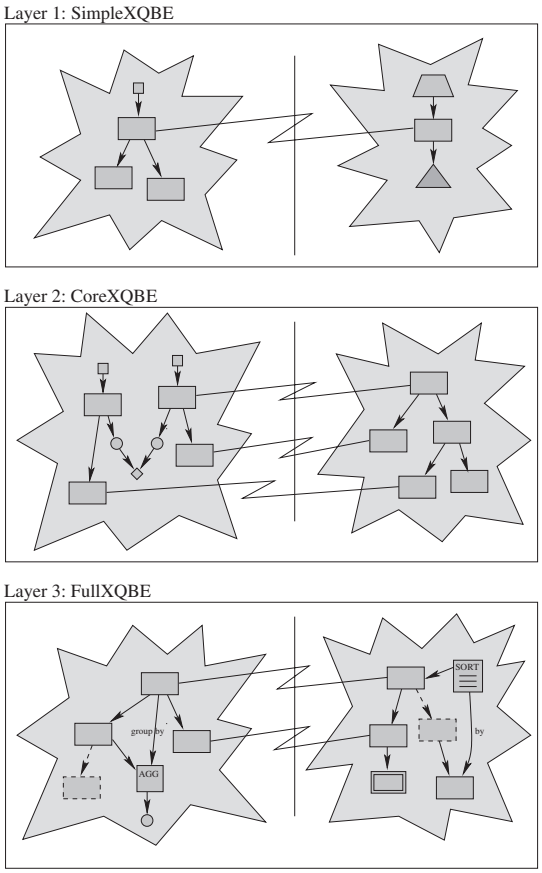

Figure 2: Informal comparison of Layers 1-3

The four XQBE layers are defined in terms of their equivalent XQuery syntax of Figure 4, whose productions are layered. In the grammar, `<nonterminals>` are represented within angle brackets and `'terminals'` are enclosed within single quotation marks.

3.1 Layer 1: SimpleXQBE

In terms of the visual syntax of XQBE, SimpleXQBE is limited to conjunctive queries with only one binding edge; the source graph mentions just one document, and the construction graph uses a replica of the extracted node "as is", with as many additional tags (trapezoidal nodes) as needed. Our rationale for choosing this simplification is that it uses just one iterator and it supports the equivalent of a conjunctive selection and a projection over an entire element; therefore, it is appropriate as the first step for learning XQuery.

In terms of "conceptual querying primitives", one can match whatever tree pattern in the source documents, describing the pattern with a conjunction of structure and value constraints. Then, one of the nodes in the pattern is chosen for an iteration: for each XML item that matches the chosen node, a trivial projection of the item is generated in output, possibly accompanied by some newly generated tags. The "bulk" projection is expressed by a grey triangle without bifurcations.

Syntactically, XQuery is obtained by adding the productions 1-11 to some basic productions common to all the XQBE-based layers. We define *fwr expressions* (to be pronounced 'fewer') as the restriction of XQuery expressions generated by the EBNF grammar of Layer 1. The reason

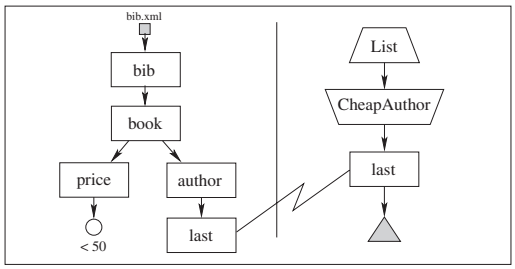

Figure 3: A query in SimpleXQBE

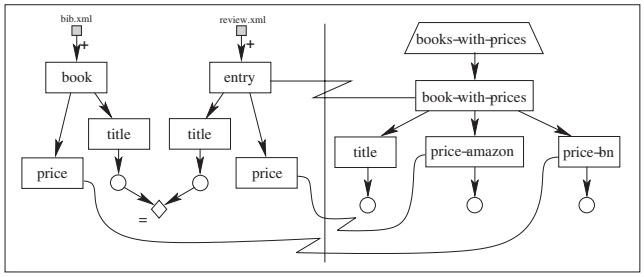

Figure 5: A multi-document join in CoreXQBE

for the *fwr* acronym is that the basic query structure is restricted to a rigid combination of *f*or-*w*here-*r*eturn expressions[1]. In addition, variables which appear in a FOR clause are restricted to be placed along a chain connecting the root of the document to the variable which carries the binding. In other words, such variables are only useful as starting points of path expressions in the Where clause. This restricted grammar enables the writing of conjunctive queries with simple left-to-right binding passing.

As an example, consider: "*Extract the surnames of all authors of all books cheaper than 50 dollars*". Its SimpleXQBE representation in Figure 3 demonstrates the use of trapezoidal nodes: those with the shorter edge above denote *single tag pairs* "enveloping" the whole result of the nodes below, while those with the larger edge above denote *multiple tag pairs*, each one individually wrapping one of the nodes below. The extracted `last` elements are replicated as is in the result. The XQuery statement is:

```
<List> { for $b in doc("bib.xml")/bib/book,
           $a in $b/author
         where $b/price < 50
         return <CheapAuthor> { $a } </CheapAuthor> } </List>
```

3.2 Layer 2: CoreXQBE

In terms of the visual syntax of XQBE, the second layer adds the capability of drawing multiple binding edges and to arbitrarily place projection constructs (including bifurcations) below the bound nodes of the construction graph, as well as to include within such projections nodes that are bound to other nodes in the source part, so as to embed the result of other expressions within the current iteration. Also, the limitation to one source document is removed, and a (rhomboidal) node type is added for expressing join comparisons.

In terms of XQuery syntax, referring to the "Layer 2" section of Figure 4, CoreXQBE is simply obtained by (1) removing the SimpleXQBE Constraint that forces variable bindings in the for clause to form a chain and (2) substituting production 11 with productions 11bis and 12. But such simple syntactic differences introduce the possibility to arbitrarily nest fwr expressions one within the return clause of another, i.e. the possibility to arbitrarily project and prune the extracted XML items with full construction capabilities. In addition, removing the semantic constraint gives wide support to Cartesian products and flattening of hierarchical structures. Thus, the use of queries strictly belonging to CoreXQBE requires covering a "conceptual gap" w.r.t. the first layer, and dealing with abstractions such as Cartesian products (which imply iterating one tuple of bindings at a time instead of one node at a time), nested iterations (so that the inner loop occurs within a *context* that has fixed some parameters), etc. - this is not easy to grasp for nonprogrammers, and indeed each such concept should be separately presented.

The CoreXQBE example is given in Fig. 1; the query exhibits a construction part with explicit projections.

An example of CoreXQBE that adds a very small gap w.r.t. SimpleXQBE was given in Figure 1; that query exhibits a construction part with explicit projection. A major gap is covered by the next query, which presents both nested iteration and join over different documents. The query is "*Construct a joint catalogue for books sold in two stores: for each book found at both www.bn.com and www.amazon.com, list the title of the book and its price from each source*", again taken from the W3C Use Cases. In the visual version (Figure 5) the iteration occurs on `entry` elements, and each item is matched against `book`s with a join based on the title. The `price`s from both sources are transported to the construction graph by means of two further binding edges, and renamed according to their provenance. The equi-join condition is expressed by the rhomboidal comparison node. The XQuery statement reads as follows:

```
<books-with-prices> {
  for $a in doc("reviews.xml")//entry,
      $b in doc("bib.xml")//book
  where $b/title = $a/title
  return <book-with-prices>
           { $b/title }
           <price-amazon>{ $a/price/text() }</price-amazon>
           <price-bn>    { $b/price/text() }</price-bn>
         </book-with-prices> }
</books-with-prices>
```

3.3 Layer 3: FullXQBE

FullXQBE adds to CoreXQBE five orthogonal concepts which enrich the query language's expressive power but do not change the core mechanism for extracting and constructing XML contents. These concepts enable: (1) computing scalar values out of aggregation, (2) building arithmetic expressions (3) ordering the results, (4) using negation both in the source and construction graph, and (5) conditional projections in the construction graph. Visually, each of these orthogonal extensions is represented by adding suitable notations, e.g., new node types for expressing aggregate, arithmetic and sorting operations, dashed nodes to denote negation, and double-lined nodes to denote conditional projections. The details of (Full)XQBE are defined in [7].

[1]Note that any path expression with (conjunctive) filtering conditions can be rewritten as a fwr expression with several additional variable bindings and a where clause containing the conjunction of the filtering predicates; therefore, the limitation introduced by production [PE], which disallows inlined filtering expression, is not a limitation of the expressive power of XQBE.

```
BASIC PRODUCTIONS: [ST] <startTag> ::= '<'  <name> <attrList>? '>'
                   [ET] <endTag>   ::= '</' <name> '>'
                   [EM] <emptyTag> ::= '<'  <name> <attrList>? '/>'
                   [NM] <name>     ::= any valid name for XML elements and attributes
                   [VA] <variable> ::= any variable that has been already bound in an outer expression
                   [CO] <const>    ::= any text constant to be interpreted as a string or a numeric value
                   [PE] <simp_p_e> ::= XPath expression (without filters) starting with 'doc()', '/', '//', or <variable>

LAYER 1:  [1] <query>          ::= <fwr_expr>  |  <startTag> '{' <query> '}' <endTag>  |  <emptyTag>  |  <const>
          [2] <fwr_expr>       ::= 'for' <var_list> <where_clause>? 'return' <return_content>
          [3] <var_list>       ::= <variable> 'in' <simp_p_e> ( ',' <variable> 'in' <simp_p_e> )*
          [4] <where_clause>   ::= 'where' <ex_quantifier>? '(' <conjunction> ')'
          [5] <ex_quantifier>  ::= 'some' <var_list> 'satisfies'
          [6] <conjunction>    ::= <atom> ( 'and' <atom> )*
          [7] <atom>           ::= <expression> <comparator> <expression>  |  'exists(' <simp_p_e> ')'
          [8] <expression>     ::= <const>  |  <variable>
          [9] <comparator>     ::= '='  |  '<'  |  '>'  |  '<='  |  '>='  |  '!='
         [10] <attrList>       ::= ( <name> '="' ( <const> | '{' <simp_p_e> '}' | '{' <fwr_expr> '}' ) '"') +
         [11] <return_content> ::= <emptyTag>  |  <variable>

     SimpleXQBEConstraint: If the <var_list> in [3] sets more than one variable bindings, these are associated with path expressions
                belonging to one single path connecting the root of the document to the target variable, i.e. the one used in [11].

LAYER 2: [11bis] <return_content> ::= <emptyTag>  |  <variable>  |  <startTag> <project_list>* <endTag>
         [12]    <project_list>   ::= '{' ( <return_content>  |  <simp_p_e>  |  <fwr_expr> ) '}'

LAYER 3: In order to add advanced features (full) several (orthogonal) extensions and rewritings are required:

 Aggr   [8bis]  <expression>     ::= <const> | <variable> | <computation> | <aggregate>
        [11ter] <return_content> ::= ( <emptyTag> | <variable> | <computation> | <aggregate> ) | <startTag> <project_list>* <endTag>

 Arith  [13]    <computation>    ::= an arbitrary arithmetic expression of <variable>s and <const>s
        [14]    <aggregate>      ::= the invocation of an aggregate function

 Sort   [15]    <order_clause>   ::= 'order by (' <name> ( ',' <name> )* ')' ('ascending' | 'descending')?
        [3bis]  <fwr_expr>       ::= 'for' <var_list> <where_clause>? <order_clause> 'return' <return_content>

 Neg    [6bis]  <conjunction>    ::= ( <atom> ( 'and' <atom> )* | <neg_clause> ) ( 'and' <neg_clause> )*
        [16]    <neg_clause>     ::= 'not(' <ex_quantifier>? '(' <atom> ( 'and' <atom> )* '))'

LAYER 4: In order to add views, only productions [1] and [W] have to be modified:

        [1bis] <query>        ::= <let_prologue>? <fwr_expr>  |  <startTag> '{' <query> '}' <endTag>  |  <emptyTag>  |  <const>
        [17]   <let_prologue> ::= 'let' <variable> := <query> ( ',' <variable> := <query> )*
```

Figure 4: EBNF specification of the proposed XQuery core for the first three layers with some extensions

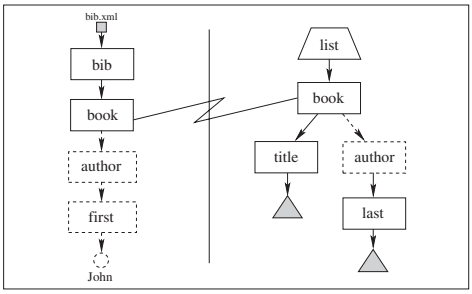

Figure 6: FullXQBE: negation and depth-projection

The extension from CoreXQBE to FullXQBE requires extending four productions and adding four more productions, as described in Figure 4.

We show how FullXQBE deals with negation by discussing the query "*List the title and the authors' surnames for books which have no authors whose name is John*". The visual version is shown in Figure 6 Negation in the source graph express the fact that given nodes or path expressions should not exist; in the example, there should be no author whose first name is John (universal quantification). Negation in the construct graph expresses the fact that given nodes should be used to reach nodes but not be part of the result; in the example, we extract the last name element but do not enclose it within the author tag. The XQuery statement reads as follows:

```
<list> { for $b in doc("bib.xml")/bib/book
         where not( some $f in $b/author/first
                    satisfies $f/text() = "John" )
         return <result> { $b/titile }
                         { $b/author/last }  </result> } </list>
```

3.4 Layer 4: ViewXQBE

Visual languages become difficult to use as the size of the query grows. However, complex queries can be decomposed using the view mechanisms; ViewXQBE extends FullXQBE by adding views, as represented in Figure 7. The data extraction process associated to the view node is *totally independent* of the query in which it is embedded. This means that no parameter-passing mechanism is provided, and that the semantics is simply that of pre-computing all the embedded queries before computing the external query.

In terms of XQuery syntax, introducing views corresponds to including in the language a partial support for let clauses, thus removing the constraint over the kind of nodes on which iterations can range. In the EBNF of Figure 4 productions 1bis and 17 add to each query in FullXQBE a prologue that allows to embed other queries and then refer to their result by means of a variable. Note that, at this stage, the resulting

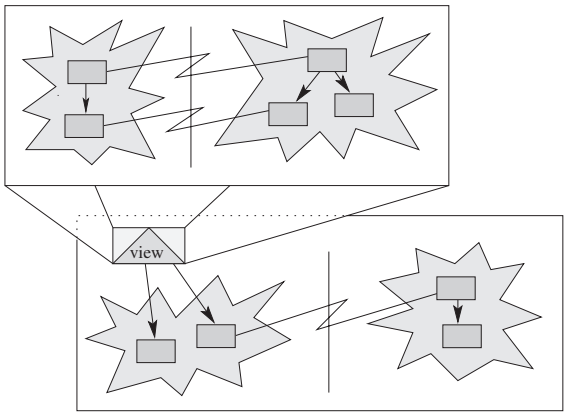

Figure 7: Views as nodes within other queries

XQuery grammar enables to interleave *for* and *let* clauses, which represent different kinds of bindings with different semantics, but with a bounded scope (let refers to a syntactic structure which is well parenthesized); thus, we still disallow an arbitrary interleaving of for and let clauses; this feature is one of the major sources of difficulty for beginners.

We exemplify ViewXQBE with a query that reads "*extract the authors of the first books about the Web published by Morgan Kaufmann*". The XQuery statement reads as follows:

```
<cheap-web-authors> {
 let $MKWeb := ( for $b in doc("bib.xml")/bib/book
                 where $b/publisher = "Morgan Kaufmann"
                       and contains($b/title, "Web")
                 return $b )
 for $b in $MKWeb
 where $b/@year = min($MKWeb/@year)
 return $b/author
} </cheap-web-authors>
```

The visual representation, omitted for brevity, reflects the pattern of Figure 7, with the assignment of variable `$MKWeb` representing the view definition. Of course *let* clauses per se are syntactic sugar, but also note that the XQuery statement conveniently defines the `$MKWeb` variable and uses it twice, once to restrict the scope of `$b` and once to compute the minimum year.

3.5 Layer 5: Arbitrarily nested XQuery

This layer basically gains the freedom to use all of the XQuery nesting capabilities and the freedom of interleaving arbitrarily for and let clauses, together with all the other language features, with the exclusion of user-defined functions and of querying data types.

For instance, consider the query asking to retrieve all the authors with different titles but exactly the same author set. Such query is expressed as:

```
<bib> { for $book1 in doc("bib.xml")//book,
            $book2 in doc("bib.xml")//book
            let $aut1 := for $a in $book1/author
                         order by $a/last, $a/first
                         return $a
            let $aut2 := for $a in $book2/author
                         order by $a/last, $a/first
                         return $a
        where $book1 << $book2 and
              not( $book1/title = $book2/title ) and
              deep-equal($aut1, $aut2)
        return <book-pair> {$book1/title}
                           {$book2/title} </book-pair> } </bib>
```

The query cannot be expressed in ViewXQBE (and therefore has no associated visualization), because it requires defining the variables with let semantics within the *context* of two specific books (i.e. to pass bindings to the sub-queries, which cannot be embedded into a view).

3.6 Layer 6: XQuery with user defined functions and data types

The last step in layering XQuery is to add the possibility to define functions and to express predicates on data types. User defined functions provide a way for XQuery users to create re-usable expressions in their queries. The use of functions increases the expressive power of the language, giving to programmers the possibility of writing general reusable code. However, people having no programming background have hard time, especially with recursive functions. Below, we demonstrate the creation of user-defined functions with an example of a recursive function taken from the W3C Use Cases. This example expresses the query "*Prepare a (nested) table of contents for Books, listing all sections with titles and preserve the original attributes of each* `section` *element*".

```
declare function toc($book-or-sec as element()) as element()* {
  for $section in $book-or-section/section
  return <section> { $section/@* ,
                     $section/title ,
                     toc($section) }  </section>  };

<toc> { for $s in doc("book.xml")/book return toc($s) } </toc>
```

The addition of the possibilities to express predicates on data types (by means of the `instance of` construct) can be useful to identify an element ignoring its exact content. As an example, consider the following query, which counts elements describing information about people:

```
count(doc("doc.xml")//[. instance of element(*, people)])
```

4. LAYERED XQUERY IN DIFFERENT DOMAINS

In order to verify the real effectiveness of our process of layering XQuery, we analyzed a large set of applicative domains of XML and the typical queries made in those domains. Our analysis started from the W3C "XML Query Use Cases" [15]. We then classified the queries which are used in the most popular XML standards and databases, those ones we expected to be used by professionals who are not programmers. Results are shown in Figure 4.

The statistics were generated in this way: for each data extraction described either in the standards or in the usage examples, we built the corresponding XQuery, and then we classified it. The message conveyed by this work is not the specific, actual value of the table entries, because we introduce our own interpretation both in the selection and in the expression of the queries (in most cases out of natural language specifications), but rather the general considerations that can be drawn by an holistic analysis.

Of course, the W3C document includes a lot of use cases which fall outside of our first four classes, because the purpose of that document is to display the various issues and alternatives in all supported query expressions. Instead, in all other cases - except MathML - queries supported by the first four layers cover between 80% and 90% of the user's needs, with SimpleXQBE covering about 40%, and CoreXQBE about 60%.

↓ Scenarios \ Layers →	1	2	3	4	5	6
W3C Use Cases	1	4	3	5	80	8
Protein Sequence Database	30	24	22	18	5	1
ebXML	27	24	16	15	12	6
FinXML	26	43	18	6	7	0
FpML	51	18	15	10	5	1
MathML	13	14	18	13	12	30
SwissProt	37	20	22	12	9	0
XFRML	44	25	14	7	6	4

Figure 8: Queries classification

Every language has its own features; for example the Protein Sequence Database or the SwissProt dataset have a very flat structure and are typically queried with very simple data extractions, most of which (over 90%) contained in our first four layers. Other application domains, such as FinXML [2], ebXML [1], XFRML [5], FpML [3] and XBRL [4], are also based on flat representations; they sometime require complex queries. MathML represents mathematical information naturally based on trees; in this case, we noticed that many data extractions (about 30%) require recursive functions, and therefore are covered just by the sixth layer, but recursion is in many cases used with queries that otherwise would fall in the first layers.

5. RELATED WORK

Our approach to the stratification of XQuery is inspired to previous similar works dedicated to SQL, see [13, 14].

Several other researchers have worked on the definition of XQuery *cores*. Recently, [11] Hidders et al. have defined a subset of XQuery dedicated to researchers. The objective of the authors is to identify the minimum set of constructs which are equivalent in expressive power to XQuery (except fore some chosen and explained omissions). They focus on the core's formal semantics and address their efforts to the scientific community of researchers (rather then the user community). The premises of this work can be traced in [12, 10], which studies the partitioning XQuery into fragments and the equivalences of such fragments.

Several variants of XQuery have been defined prior to its standardization (e.g. Quilt) or have been extending its expressive power (e.g. by adding updates, textual manipulation, reactive components, fuzzyness); the work of OptXQuery [8] is partially related to our effort, as it aims at simplifying the use of XQuery for expressing grouping. It does so, by adding an effective new clause, which however is not yet covered by most XQuery engines.

An important alternative to our approach is the use of XPath as the basic query language. XPath can be considered composed by two main components: one for building path expressions and one for defining functions. The main limitation of XPath is the lack of construction primitives, that from our point of view are fundamental, even for basic use of XQuery. For this reason, we support tag construction even in our first layer.

6. CONCLUSIONS

From our findings, we are confident that XQBE could be profitably used as the entry point of non programmers, with a progression from Simple to Core to Full to View versions. XQBE would always produce XQuery expressions on the side, so we expect users to progressively become expert of both formalisms, and eventually, when needed, turn to full XQuery (or, equivalently, to give up with a visual interface once programming becomes more natural and effective). The XQBE Client can be downloaded from http://dbgroup.elet.polimi.it. Regardless of XQBE, we believe that concepts identified in this paper are applicable to XQuery and could lead to several reduced subsets of the language that, for pedagogical reasons, could be taught and experienced in progression.

Acknowledgement (and disclaimer)

The inspiration of this paper comes from a conversation with Donald Kossmann at XIME-P, held in conjunction with ACM-SIGMOD 2005. At the workshop, Stefano Ceri advocated a layering of XQuery for the purpose of ease of optimization and implementation, but he was convinced that such an approach is either impossible or inconvenient, as XQuery lacks implementation orthogonality. Instead, we agreed that "pedagogical orthogonality" might be possible and useful. Of course, it is the authors' responsibility having tried that idea, to the best of their skills.

7. REFERENCES

[1] ebXML. http://www.ebxml.org.
[2] FinXML Home Page. http://www.finxml.org.
[3] FpML. http://xml.coverpages.org/fpml.html.
[4] XBRL Home Page. http://www.xbrl.org.
[5] XFRML Working Group Home Page. http://www.xfrml.org.
[6] Daniele Braga. XQBE: XQuery By Example, PhD Thesis. https://www.elet.polimi.it/upload/braga/phdthesis.pdf, 2005.
[7] Daniele Braga, Alessandro Campi, and Stefano Ceri. XQBE (XQuery By Example): a visual interface to the standard XML query language. *ACM Transactions on Database Systems (TODS)*, 30(2), June 2005.
[8] Alin Deutsch, Yannis Papakonstantinou, and Yu Xu. Minimization and group-by detection for nested xqueries. In *ICDE*, page 839, 2004.
[9] Andrew Eisenberg and Jim Melton. Xquery 1.0 is nearing completion. *SIGMOD Rec.*, 34(4):78–84, 2005.
[10] Jan Hidders, Stefania Marrara, Jan Paredaens, and Roel Vercammen. On the expressive power of XQuery fragments. volume 3774, pages 154–168, Trondheim, Norway, 2005.
[11] Jan Hidders, Philippe Michiels, Jan Paredaens, and Roel Vercammen. LiXQuery: a formal foundation for XQuery research. *SIGMOD Rec.*, 34(4):21–26, 2005.
[12] Wim Le Page, Jan Hidders, Jan Paredaens, Roel Vercammen, and Philippe Michiels. On the expressive power of node construction in XQuery. pages 85–90, Baltimore, Maryland, USA, 2005.
[13] Leonid Libkin. Expressive power of SQL. *Lecture Notes in Computer Science*, 1973, 2001.
[14] Peter Schauble and Beat Wuthrich. On the expressive power of query languages. *ACM Trans. Inf. Syst.*, 12(1):69–91, 1994.
[15] W3C. XML Query Use Cases. http://www.w3.org/TR/xmlquery-use-cases, Nov 2003.
[16] Moshé M. Zloof. Query-by-Example: A Data Base Language. *IBM Systems Journal*, 16(4):324–343, 1977.