The Query Language TQL

Giovanni Conforti Giorgio Ghelli
Antonio Albano Dario Colazzo Paolo Manghi Carlo Sartiani

Dipartimento di Informatica, Università di Pisa, Pisa, Italy

May 1, 2002

Abstract

This work presents the query language TQL, a query language for semistructured data, that can be used to query XML files. TQL substitutes the standard path-based pattern-matching mechanism with a logic-based mechanism, where the programmer specifies the properties of the pieces of data she is trying to extract. As a result, TQL queries are more 'declarative', or less 'operational', than queries in comparable languages. This feature makes some queries easier to express, and should allow the adoption of better optimization techniques. Through a set of examples, we show that the range of queries that can be declaratively expressed in TQL is quite wide. The implementation of TQL binding mechanism requires the adoption of non-standard techniques, and some of its aspects are still open. In this paper we implicitly report about the current status of the implementation by writing all queries using the version of TQL that has been implemented, and that can be freely downloaded from `//tql.di.unipi.it/tql`.

1 Introduction

The *Tree Query Language* [1] (TQL) is a query language for tree-shaped semistructured data. The language is based on the set comprehension (match-filter-construct) paradigm, in the tradition of SQL, StruQL, Lorel, Quilt, XQuery (among many others). However, the match-filter operation is expressed in TQL using a variant of the ambient logic [2], a logic defined to describe process structure and behaviour. TQL adopts a subset of that logic, for its ability to describe trees.

The TQL logics is used to express the binding (match-filter) part of a query. The same logic can be exploited to describe those properties of the data that are usually expressed through types and constraints. This implies that:

- TQL queries can be exploited in order to check whether a data source has a type, or satisfies a constraint;
- whenever a type or a constraint C is known to hold for a data source, the binder B of TQL queries is equivalent to its refinement $B \wedge C$, which is a legitimate TQL binder, as we will see. This refinement opens the way for new optimizations, or even for the static declaration of an empty result, if the unsatisfiability of $B \wedge C$ can be detected.

We will exemplify later these two properties.

The promise of combining the expression of types, constraints, and queries in just one language, and to use this synergy for optimization and error-checking purposes, is the kernel of the TQL project. But the language is also worth studying for its ability to express complex queries by declaring the properties of what one is looking for, instead of describing a path to arrive there. While most interesting properties, as we will show through examples, are heavily path-based, others involve negation, implication, universal quantification, and are expressed in other languages, such as XQuery, by resorting to external functions or by operational means, which makes optimization and formal reasoning on the queries quite harder. While many programmers are perfectly comfortable with operational-oriented programming and reasoning, others find declarative expression easier, and there is at least a pattern, exemplified in Section 5.3, where the TQL style clearly pays off. In TQL, whenever you are able to describe a property of a specific tag (e.g., *title* is a key for each *article*), by substituting the constant with a variable you obtain the query that finds all tags with the same property (e.g., find all pairs x, y such that tag x is key for y).

This feature is reminiscent of prolog-like languages. However, TQL does not share datalog problems with negation, partly because TQL is born with negation, and mostly because we restrict ourselves to a monotone form of recursion.

In the rest of the paper we present the expressive power of TQL and some of the properties we discussed here, through a succession of examples, all tested on the current TQL implementation. In some cases we also present an XQuery equivalent query, for the sake of comparison, and also to clarify our usage of the terms 'declarative' and 'operational' expression of queries.

The current version of TQL data model is unordered. This makes TQL unusable in document-oriented applications, but this lack of order is very important, in terms of allowed optimization, in database-like applications. Dealing with order is left as a future extension.

The contact author is Giovanni Conforti `confor@di.unipi.it`

2 Related work

Many query languages for semistructured data and XML have been designed in the past years: StruQL, Lorel, XQL, XML-QL, YATL, etc. Building on this research, W3C is designing XQuery [3], a standard query language for XML data, which subsumes many concepts coming from these languages. XQuery (still a work in progress) is a typed, Turing-complete query language that can be used in both XML-enabled database systems and native XML systems.

While TQL and XQuery are based on the same *bind-filter-reconstruct* paradigm, they differ in many aspects.

First of all, TQL, by design, is based on a logic that can express types, constraints, and queries, and is tailored for formal, and automated, manipulation. On the other side, XQuery is designed as an industrial-strength language, aimed at both database-oriented and document-oriented applications. As a consequence, TQL has a very sharp semantic definition, that can be completely defined in one page of formulae, while XQuery semantics is much more complex. On the other side, XQuery data model supports order and *oid-like* information, which are not dealt with in the current TQL version.

Second, even though XQuery expressive power is greater than TQL's (the former is Turing-complete), some queries can be more easily expressed in TQL, thanks to the greater expressive power of the tree-logic with respect to a pure matching mechanism.

Finally, XQuery features powerful vertical navigational facilities, while it lacks corresponding horizontal operators; TQL, instead, makes no difference between horizontal and vertical navigation, hence allowing the user to easily impose horizontal constraints on documents.

3 The Simplest Queries

3.1 The Input Data

We begin with some standard queries, borrowed from the W3C XMP Use Case [4]. These queries operate over the XML document available at `//tql.di.unipi.it/bib.xml`, which we assume to be bound to the variable `$Bib` in the global environment (the TQL system allows any document on the web to be bound to a variable). The document contains bibliography entries, whose structure is described by the following DTD:

```
<!ELEMENT bib  (book* )>
<!ELEMENT book  (title,  (author+ | editor+ ),
                publisher, price )>
<!ATTLIST book  year CDATA  #REQUIRED >
<!ELEMENT author  (last, first )>
<!ELEMENT editor  (last, first, affiliation )>
<!ELEMENT title  (#PCDATA )>
<!ELEMENT last   (#PCDATA )>
<!ELEMENT first  (#PCDATA )>
<!ELEMENT affiliation  (#PCDATA )>
<!ELEMENT publisher  (#PCDATA )>
<!ELEMENT price  (#PCDATA )>
```

The DTD specifies that a `book` element contains a `title`, one or more `author` elements or one or more `editor` elements, one `publisher` element and one `price` element; it also has a `year` attribute. An `author` contains a `last` and a `first` name elements. An `editor` element also contains an `affiliation`. Finally, `title`, `last`, `first`, `publisher`, and `price` elements contain string values.

In this paper we present the XML file using its more compact TQL-syntax representation, which looks as follows (the implemented system allows both XML and TQL visual presentations):

```
bib[
  book[year[1992]
    | title[FoundationsDatabases]
    | author[ first[Serge] | last[Abiteboul] ]
    | author[ first[Richard] | last[Hull] ]
    | author[ first[Victor] | last[Vianu] ]
    | publisher[Addison]
    | price[60]
  ]
  | book[year[1990]
    | title[SistemiOperativi]
    | author[ first[Piero] | last[Maestrini] ]
    | publisher[McGrawHill]
    | price[38]
  ]
...
```

In this format, `bib[C]` stands for an element tagged `bib` whose content is `C`, while `C1 | C2` is the concatenation of two elements, or, more generally, of two sets of elements. We use this non-XML notation because TQL is born as a language to query semistructured data in general, i.e. unordered forests with labeled nodes, and not just XML. XML is just one way to construct such forests, using tagged elements (and attributes) to build labeled nodes.

3.2 The Formal Presentation of TQL Data Model

More formally, TQL data model is defined by the following syntax and equations. The syntax specifies that a forest is either a leaf, or an empty forest, or a node labeled by *tag* and leading to a subforest, or the union of many forests. We choose to distinguish between a leaf '*tag* and a leaf *tag*[0] in order to model the XML distinction between PCDATA and empty elements.

TQL forests

$forest ::= \text{'}tag \mid \mathtt{0} \mid tag[forest] \mid forest \,|\, forest$

The formal definition of the data model is completed by the equations that specify that | is commutative and

TQL node-labeled forests can be equivalently described as edge-labeled trees, as done in [1].

associative, and 0 is its neutral element:

$$t \mid t' = t' \mid t \qquad t \mid (t' \mid t'') = (t \mid t') \mid t'' \qquad t \mid 0 = t$$

Hereafter we will elide the leaf constructor ', writing t[d] instead of t['d], unless ambiguity arises; the same abbreviation is supported in the implemented system.

3.3 Matching and Binding

The basic TQL query is `from Q |= A select Q'`, where Q is the *subject* (or *data source*) to be matched against the formula A, and Q' is the result expression. The matching of Q and A returns a set of bindings for the variables that are free in A. Q' is evaluated once for each of these bindings, and the concatenation of the results of all these evaluations is the query result.

For example, consider the following TQL query, that returns the titles of all books written in 1991, and is evaluated in an environment where $Bib is bound as specified above.

```
from $Bib |= .bib[.book[.year[1991]
                        And .title[$t]
                       ]
                ]
select title[$t]
```

The formula:

```
.bib[ .book[ .year[1991] And .title[$t] ] ]
```

is an ambient logic formula, which should be read as: "there is a path .bib[.book[ ]] that reaches a place that matches .year[1991] And .title[$t], i.e. a place where you find both a path .year[] leading to 1991 and a path .title[ ] leading to something, that you will call $t".

The formula .tag[A], read "there exists an element tag whose content satisfies A", is the most useful operator, but is actually defined in terms of three more basic operators: truth T, vertical splitting A' | A'', and element matching tag[A].

The element formula tag[A] only matches a one-element document: while .t[A] matches both forests t[D] and t[D] | t2[D2] | ... (provided that A matches D), the formula t[A] only matches the first one. The truth formula T matches every forest. Finally, the formula A_1 | A_2 matches D iff D is equal, modulo reordering, to D_1 | D_2, with A_i matching D_i. For example, the following pairs match, provided that $a is bound to Date:

```
title[IDB]|author[Date]|      author[$a]|title[IDB]|
year[1994]                   year[1994]

title[IDB]|year[1994]                  T

title[IDB]|author[Date]|          author[$a]|T
year[1994]

author[Date]                      author[$a]|T
```

The third formula can be read as: there is an author $a and something else, hence is equivalent to .author[$a]; the fourth pair matches as well, since the empty forest matches T. Hence, m[A] | T is equivalent to .m[A]; this is actually the official definition of the semantics of .m[A].

While in this example we matched $t with a leaf, a TQL variable can be matched against any forest, or against a tag.

For example, the following query returns any tag inside a book whose content matches .first[Serge]; .a.b[A] abbreviates .a[.b[A]].

```
from $Bib |= .bib.book.$tag.first[Serge]
select SergeTag[$tag]
```

Finally, the following query matches the formula year[1992] | $EveryThingElse against any book, hence it returns, for any book whose year is 1992, everything but the year:

```
from $Bib |= .bib.book[year[1992]
                       | $EveryThingElse
                      ]
select BookOf1992[$EveryThingElse]
```

Since we have two books of 1992, there are two possible bindings for $EveryThingElse, each corresponding to the whole content of a 1992 book without its year subtree; hence the result is:

```
BookOf1992[
    title[FoundationsDatabases]
    | author[ first[Serge] | last[Abiteboul] ]
    ...
]
| BookOf1992[
    | title[Interpreters]
    | author[ first[Vincent] | last[Aho] ]
    ...
]
```

Hereafter, as a convention, we use lowercase initials for variables that are bound to tags and uppercase initials for variables that are bound to forests.

3.4 Matching and Logic

TQL logic allows the programmer to combine matching and logical operators. For example, the condition in the following query combines the request for the existence of a title field, of a $x field containing Springer, and of either an author.last or an editor.last path leading to Buneman.

```
from $Bib |=
     .bib.book [.title[$t]
                And Exists $x. .$x[Springer]
                And (.author.last[Buneman] Or
                     .editor.last[Buneman])
               ]
select title[$t]
```

The pattern `Exists $x. .$x[A]` is common enough to deserve the abbreviation `.%[A]`, that we will use hereafter (see [1] for the exact definition of this abbreviation).

Conjunction, disjunction, and universal quantification are operators that can be found in many match-based languages. TQL, however, has the full power of first-order logic, hence we can express universal quantification and negation of arbitrary formulas. This will be exemplified later.

4 Restructuring the Data Source

In TQL syntax, a subquery can appear wherever a forest expression is expected, as expressed by the following syntax:

TQL Queries

Q ::= `from` Q `|=`A `select` Q | *'tag* | `0` | *tag*`[`Q`]` | $Q|Q$

This freedom of nesting is a feature of most modern query languages, and is typically exploited to use the nesting structure of the query in order to describe the nesting structure of the result. For example, in our data source there is an entry for each book, containing the list of its author. We can restructure it to obtain an entry for each author, containing the list of its books. The structure of the result can be visualized as follows, where `(A)*` indicates an arbitrary repetition of the `A` structure:

```
(author[ authorname[...]  | (book[...])* ])*
```

Observe how this structure is reflected by the structure of the following query, with a from-select for each `*`.

```
from   $Bib |= .bib.book.author[$A]
select author[authorname[$A]
            | from $Bib |= .bib.book[author[$A]
                                     | $OtherFields
                                    ]
              select book[$OtherFields]
            ]
```

This query performs a nested loop. For each binding of `$A` to a different author, it returns a forest `result[author[$A]] | book[...]|...|book[...]]`, where `book[...]|...|book[...]` is the result of the inner query, i.e. it contains one book element for each book whose author is `$A`. As in a previous example, we extract, from the input book, all the fields but the author.

5 Schema-less XML data

As XML documents are not necessarily to come with a DTD, query languages should provide mechanisms for querying data regardless of the structure.

Alternatively, when schema information is fundamental for writing sensible queries, schema inference mechanisms are very useful. For example, one may be interested in finding the exact structure of the data, or in finding the mandatory elements in the data. Property checking tools may also be useful, so as to prove the validity of given assertions about the data. For instance, checking whether a certain set of tags is a primary key, or if a tag is mandatory in a specified path.

TQL provides all these mechanisms by simply combining *tag variables* (as in [6] and [7]) and ambient logic, as shown in the following sections.

5.1 Querying in absence of schema

We consider an XML document, bound to `$Bib2` in the global environment, which is similar to the `$Bib` file, but features some extra-elements with a title (i.e. article, phd, etc.), whose labels are not known a priori.

The following query selects the title of all elements, whatever the label, and wherever they are, that contain an element whose value is `Suciu`; the `*` operator iterates a path an arbitrary number of times (may be zero); `.%*` must be read as `(.%)*` and corresponds, roughly, to the XPath operator `//`.

```
bib[from $Bib2 |= .%*.$B[ $A[Suciu] | $Rest ]
    select $B[ Suciu[$A] | $Rest ]
   ]
```

This query constructs a Suciu's personal bibliography document, selecting all elements in `$Bib2` where he appears and inverting the tag with the content. The remaining information present in the elements involving Suciu are inserted in the result using the `$Rest` variable.

This query clearly reveals some of the differences between TQL and XQuery, in which it would be expressed as follows,

```
<bib>
   for $b in $Bib2//*,
     let $xx := $b/*,
       for $y in $xx
       where $y/data() = "Suciu"
   return <xf:name($b)>
                <Suciu>
                   xf:local-name($y)
                </Suciu>,
                { op:except($xx,$y)}
           </xf:name($b)>
<bib>
```

Observe how TQL's binding mechanism and horizontal navigation are more declarative than XQuery's, which adopts instead operational techniques:

- the definition of each binding to a variable requires a corresponding nested loop (`for` or `let`), while in TQL all free variables are bound in one single `from-select` clause;
- horizontal constraints are dealt with an external operator `op:except`, while in TQL these are expressed with the logic horizontal navigation operator |.

5.2 Checking Properties

In this section we show how tree logic formulae can be used to express properties of XML data. When a formula `A` expresses a property, we can check it by running the query `from Q |= A select success`: this query returns the leaf `success` if `A` holds over `Q`, and an empty forest otherwise.

As a first example we consider a query that verifies if the tag `title` is mandatory for `book` elements in the `$Bib` document.

```
from $Bib |= bib[Not .book[Not .title[T]]]
select title_is_mandatory
```

The formula `Not .book[Not .title[T]]` means: it is not the case that there exists a book whose content does not contain any title, i.e. each book contains a title. TQL actually features an operator `!a[A]` defined as `Not .a[Not A]` which we can directly use, as in the following query. Here `!book.title[T]` is an abbreviation for `!book[.title[T]]`, hence means: for every book there is a title.

```
from $Bib |= bib[ !book.title[T] ]
select title_is_mandatory
```

The formula `!a[A]` is dual to `.a[A]` in the same sense as $\forall x.A$ is dual to $\exists x.A$, or $\wedge$ is dual to $\vee$. In TQL, every primitive operator has a derived dual; this implies that negation can always be pushed inside any operator, hence you can write any query with no use of negation. Actually, when negation appears in a query, in most cases the TQL optimizer pushes it down to the query leaves (variables, expression of the content of a leaf, comparisons), since negation is quite expensive. This is the reason why, although we claim that unlimited negation is an important feature of TQL, you will see very little explicit use of negation in our examples.

The next query verifies that `title` never appears twice in a field.

```
from $Bib |= Not bib[.book[ .title[T] | .title[T] ] ]
select title_never_appears_twice
```

Another interesting property to verify is whether a given tag is a primary key. There are many possible generalizations of the relational notion of key to the semistructured case. The statement below, for example, says that `title` is a mandatory field, and that you cannot find two separate books with the same title (more precisely, with one title in common).

```
from $Bib |=
   bib[!book[.title[T]]
       And foreach $X. Not (.book.title[$X] |
                            .book.title[$X])
      ]
select each_title_is_key
```

Of course, if the system knows that `$Bib` satisfies `bib[!book[.title[T]]]`, this knowledge implies that

`bib[!book[.title[T]] And foreach $X. Not (.book.title[$X] | .book.title[$X]) ]`

is equivalent (over `$Bib`) to

`bib[foreach $X. Not (.book.title[$X] | .book.title[$X])]`.

We do not comment further on this point, since this kind of optimization is out of the reach of the current implementation of TQL.

Our last query checks that the `$Bib` element contains only elements labeled `book`, by asking that each tag inside the outer `bib` is equal to `book`.

```
from $Bib |= bib[foreach $x .$x[T] implies $x=book]
select only_book_inside_bib
```

This query can be rewritten using path operators as follows:

```
from $Bib |= bib[Not (.Not book[T])]
select only_book_inside_bib
```

Here `Not book` is a tag-expression that stands for any tag different from `book`. Hence, `.Not book[T]` means: there exists a subelement whose tag is different from `book`. Hence, `Not (.Not book[T])` means: there exists no subelement whose tag is different from `book`.

5.3 Extracting the Tags That Satisfy a Property

Every query Q in the previous subsection checks a property P of a tag `t`. In all such cases, if we substitute, in Q, `t` with a tag variable, we obtain a query that finds the set of *all* tags that satisfy P.

For example, we can extract all keys of books by taking the query that checks whether `title` is a key, and substituting `title` with `$k`, as follows:

```
from $Bib |=
     bib[!book[.$k[T]]
         And foreach $X. Not (.book.$k[$X] |
                              .book.$k[$X])
        ]
select key[$k]
```

It must be highlighted that this is possible because in TQL we can universally quantify even on a formula with other free variables (`$k`, in this case). The query evaluation algorithm we exploit to this aim is quite sophisticated, and is described in [5].

Generalisation by simple substitution is not possible in XQuery, where variables inserted to replace tags must at least be bound by an outer `for` clause, thus requiring the redesign of the original query.

A similar generalisation can be performed for the queries that check whether a label is mandatory, or occurs only

once, inside another one. We present below a query that almost produces a DTD for any input XML file (modulo ordering) by extracting all the tags in the file and listing, for each of them, all the labels that must or may appear, and distinguishing among them the ones which may be repeated and the ones which only appear once. While it may look frightening, it has just been obtained by a trivial generalization of the simple queries we presented above.

```
from $parts |= .%*.$tag[.%[T]]
select $tag[   mandatory_subtags
                  [from $parts |=
                      Not (.%*.$tag[Not .$subtag[T]])
                   select $subtag[]
                  ]
             | optional_subtags
                  [from $parts |=
                      .%*.$tag[ .$subtag[T]]
                      And .%*.$tag[not .$subtag[T]]
                   select $subtag[]
                  ]
             | list_subtags
                  [from $parts |=
                      .%*.$tag[ .$subtag[T] |
                                .$subtag[T]
                              ]
                   select $subtag[]
                  ]
             | non_list_subtags
                  [from $parts |=
                      .%*.$tag[ .$subtag[T]]
                      And not .%*.$tag[ .$subtag[T] |
                                        .$subtag[T]
                                      ]
                   select $subtag[]
                  ]
          ]
```

6 Recursion

TQL logic also includes two monotonic recursion operators (`rec` and `maxrec`), very similar to the μ and ν operators (minimal and maximal fix point) of modal logic. These can be used to interpret the Kleene star operator `path*` and to express recursively definable forest properties. Consider for example the following formula:

```
rec $Binary. (%[$Binary | $Binary]) or %[0] or '%
```

The formula describes a binary tree, defined as either a node leading to two binary trees, or as a leaf; `%[0] Or '%` matches a leaf, which may be (in XML terminology) either an empty element, or a piece of PCDATA.

7 Conclusions

Although the language TQL originates from the study of a logic for mobile ambients, for the simplest queries it turns out to be quite similar, in practice, to other XML query languages.

However, the expression of queries which involve recursion, negation, or universal quantification, has in TQL a clear declarative nature, while other languages are forced to adopt a more operational approach.

All queries presented in this paper are executable in the prototype version of the TQL evaluator, and can be found in the file `demo.tql` in the standard distribution. The current version of the prototype works by loading all data in main memory, but is already based on a translation into an intermediate TQL Algebra [5], with logical optimizations carried on both at the source and at the algebraic level. The intermediate algebra works on infinite tables of forests, represented in a finite way, and supports such operations as complement, to deal with negation, coprojection, to deal with universal quantification, several kinds of iterators, to implement the | operator, and a recursion operator.

TQL is currently based on a unordered nested multisets data model. The extension of TQL's data model with ordering is an important open issue.

References

[1] L. Cardelli and G. Ghelli. A Query Language Based on the Ambient Logic. In *Proc. of European Symposium on Programming (ESOP), Genova, Italy*, 2001.

[2] L. Cardelli and A. D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *Proc. of Principles of Programming Languages (POPL)*. ACM Press, January 2000.

[3] Don Chamberlin, James Clark, Daniela Florescu, Jonathan Robie, Jérôme Siméon, and Mugur Stefanescu. XQuery 1.0: An XML Query Language. Technical report, World Wide Web Consortium, jun 2001. W3C Working Draft.

[4] Don Chamberlin, Peter Fankhauser, Massimo Marchiori, and Jonathan Robie. XML Query Use Cases. Technical report, World Wide Web Consortium, December 2001. W3C Working Draft.

[5] G. Conforti, O. Ferrara, and G. Ghelli. TQL Algebra and its Implementation. To appear in *Proc. of IFIP International Conference on Theoretical Computer Science (IFIP TCS), Montreal, Canada*, August 2002. Available at `http://tql.di.unipi.it/tql`.

[6] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. XML-QL: A Query Language for XML. Technical report, World Wide Web Consortium, August 1998. Submission to the World Wide Web Consortium.

[7] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language and processor for a web-site management system. In *Proc. of Workshop on Management of Semistructured Data, Tucson*, 1997.