A Full Pattern-based Paradigm for XML Query Processing

Véronique Benzaken[1], Giuseppe Castagna[2], and Cédric Miachon[2]

[1] LRI, UMR 8623, C.N.R.S., Université Paris-Sud, Orsay, France
[2] C.N.R.S., Département d'Informatique, École Normale Supérieure, Paris, France

Abstract. In this article we investigate a novel execution paradigm—ML-like pattern-matching—for XML query processing. We show that such a paradigm is well adapted for a common and frequent set of queries and advocate that it constitutes a candidate for efficient execution of XML queries far better than the current XPath-based query mechanisms. We support our claim by comparing performances of XPath-based queries with pattern based ones, and by comparing the latter with the two efficiency-best XQuery processor we are aware of.

1 Introduction, Motivations, and Goals

In this article we investigate a novel execution paradigm—namely ML-like pattern-matching—for XML query processing. We show that such a paradigm is well adapted for a common and frequent set of queries and thus could be used as a compilation target for XQuery. More precisely, to do so, we endow the pattern-matching based language ℂDuce with an SQL-like query language that we introduce in this article and dub ℂQL. ℂDuce [4, 20] (pronounce "seduce") is a strongly and statically typed *pattern-based* higher-order functional programming language for XML. It is standard compliant (XML, Namespaces, Unicode, XML Schema validation, DTD, etc.) and fully operative and implemented (the distribution of ℂDuce/ℂQL is available at www.cduce.org). One of the distinguishing characteristics of ℂDuce is its pattern algebra. ℂDuce inherits and extends XDuce [21] pattern algebra and implements it by a very efficient "just in time" compilation [19]. ℂQL is a query language in which queries are written using patterns (in the sense of ℂDuce patterns) and where the execution mechanism is based on pattern-matching (in the sense of ML-like languages). ℂQL/ℂDuce patterns are more similar to ML patterns than to XPath expressions. With respect to XPath expressions, ℂDuce patterns are far more declarative inasmuch as while the former strictly indicate navigation paths, the latter reflect the whole structure of matched values and they can be composed by applying boolean combinators.

To demonstrate that pattern-matching is relevant for query compilation and evaluation in the XQuery context, we also introduce, for free, some syntactic sugar to yield an XQuery-like programming style. We call this extension $\mathbb{C}\mathrm{QL}_X$. We chose to experiment with $\mathbb{C}\mathrm{QL}_X$ as we wanted to fully exploit the already implemented ℂDuce's pattern compilation schema and runtime support rather than re-implementing it in the context of XQuery.

Several proposals for defining query languages for XML have been made [2, 6, 14, 15] and a comparative study can be found in [1]. Among them we choose to briefly recall the main features of XQuery[15] as (one of) our purpose is to show that the experiments performed with ℂQL apply obviously to it.

XQuery [15, 17] is becoming the W3C standard in terms of query languages. An earlier proposal was Quilt [11], which borrowed many functionality from XPath [13],

XQL [23], XML-QL [16], SQL, and OQL. XQuery is a strongly and statically typed functional language whose type system was largely inspired by XDuce [21].

```
<books-with-prices>
 { for  $b in $biblio//book,
        $a in $amazon//entry
   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>
```

A query is expressed by a FLWR expression: for (iteration on a set of nodes), let (variables binding), where (filter the elements according to a condition), and return (construct the result for each node satisfying the where clause) The query on the side is an example of FLWR expression (it is the Q_5 query of the XML Query Use Cases). Pattern expressions in XQuery, such as $amazon//entry or $a/price/text(), are based on XPath [13]. Many works among them [12] attempts to optimise XQuery evaluation.

The immanent purpose of this article is to investigate whether pattern-matching *à la* ℂDuce is well adapted for main memory XML query processing. The answer is positive and ℂDuce's patterns and pattern-matching mechanism can serve as an execution mechanism for XQuery. Indeed, the need for efficient main memory query processing is still of crucial interest. As pointed out by many works a bunch of application scenarios such as message passing, online processing do not manipulate huge documents. We advocate that ℂDuce patterns are a better candidate for XML patterns (again in the sense of [10]) than path expressions. We base our plea on the following observations:

1. ℂDuce patterns are more declarative: different patterns can be combined by boolean combinators, thus, in a declarative way. Furthermore, they can represent the whole structure of matched values. This allows the capture of larger quantities of information in a single match.
2. ℂDuce patterns are more efficient: our measurements show that a query written in $\mathbb{C}\mathrm{QL}_X$ using the navigational style is always slower (sometimes even after some optimisation) than the same query written in ℂQL (even when the latter is obtained from the former by an automatic translation). Of course this claim must be counterbalanced by the fact that our comparison takes place in ℂDuce, a language whose implementation was specifically designed for efficient pattern matching resolution. Nevertheless we believe that this holds true also in other settings: the fact that ℂDuce patterns can capture the whole structure of a matched value (compared with paths that can capture only a subpart of it) makes it possible to collect physically distant data in a single match, avoiding in this way further search iterations. To put it simply, while a path expression pinpoints in a tree only subtrees that all share a common property (such as having the same tag) a ℂDuce pattern does more as it can also simultaneously capture several unrelated subtrees.

Our claim is supported by benchmark results. We performed our experiments in ℂDuce rather than XQuery since this is of immediate set up: XPath/XQuery patterns are implemented in ℂDuce as simple syntactic sugar, while an efficient integration of ℂDuce patterns in XQuery would have demanded a complete rewriting of the runtime of a XQuery processor (but, again, we think that this is the way to go). So instead of comparing results between standard XQuery and a version of XQuery enriched with ℂDuce patterns, we rather compare the results between ℂQL (the standard ℂDuce query lan-

guage) and $\mathbb{C}$QL$_X$ (that is $\mathbb{C}$QL in which we only use XQuery patterns and no $\mathbb{C}$Duce pattern).

Furthermore, in order not to bias the results with implementation issues, in all our experiments with $\mathbb{C}$QL$_X$ we avoided, when this was possible[1], the use of "//" (even if the "//"-operator is available in $\mathbb{C}$Duce): whenever in our tests we met a (XQuery) query that used "//" (e.g. the query earlier in this section) we always implemented it by translating (by hand) every occurrence of "//" into a minimal number of "/". Such a solution clearly is much more effi cient (we program by hand a minimal number of "/" searches instead of using "//" that performs a complete search on the XML tree) and does not depend on how "//" is implemented in $\mathbb{C}$Duce (in $\mathbb{C}$Duce "//" is implemented by a recursive function whose execution is much less optimised than that of "/" which, instead, enjoys all the optimisations of the $\mathbb{C}$Duce runtime). Therefore it is important to stress that in this article we are comparing hand-optimised XQuery patterns in $\mathbb{C}$QL$_X$ with automatically generated (therefore not very optimised) $\mathbb{C}$Duce patterns in $\mathbb{C}$QL: the results of our tests, which always give the advantage to the second, are thus very strong and robust.

The existence of an automatic translation from (a subset of) XPath patterns to $\mathbb{C}$Duce ones, is a central result of our work. This work demonstrates that XPath-like projections are redundant and in a certain sense, with respect to patterns, problematic as they induce a level of query nesting which penalises the overall execution performance. We thus defi ned a formal translation of $\mathbb{C}$QL$_X$ to $\mathbb{C}$QL and showed that it preserves typing. This translation maps every $\mathbb{C}$QL$_X$ query into a (flat) $\mathbb{C}$QL one (i.e., with all nesting levels induced by projections removed), and is automatically implemented by the $\mathbb{C}$Duce/$\mathbb{C}$QL compiler. Not only such a translation is useful from a theoretical point of view, but (i) it accounts for optimising queries and (ii) shows that the approach can be applied both to standard XQuery (in which case the translation would be used to compile XQuery into a pattern aware runtime) and to a conservative extension of XQuery enriched with $\mathbb{C}$Duce patterns (in which case the translation would optimise the code by a source to source translation, as we do for $\mathbb{C}$QL$_{(X)}$). Whatever in $\mathbb{C}$Duce or in XQuery this transformation allows the programmer to use the preferred style since the more effi cient pattern-based code will be always executed. We also adapt logical optimisation techniques which are classical in the database fi eld to the context of pattern based queries and show through performance measurement that they are relevant also in this setting.

To further validate the feasibility of pattern-matching as an execution model we also compared $\mathbb{C}$QL performances with those of XQuery processors. Since our language is evaluated in main memory (we do not have any persistent store, yet) as a fi rst choice we compared $\mathbb{C}$QL performance with Galax [3] that besides being a reference implementation of XQuery, uses the same technologies as $\mathbb{C}$Duce (noticeably, it is implemented in OCaml). However, the primary goal of Galax is compliance with standards rather than effi ciency and for the time being the (web) available version does not implement any real optimisation and has poor memory management (even if [22] proposes some improvements), which explains that $\mathbb{C}$QL outperformed Galax (of an order of magnitude

[1] Of course there exist types (such as t = <a>[t | <b>[]]) and queries (//<a>) for which such a translation is not possible,

up to tens of thousands of time faster). Therefore we decided to run a second series of tests against Qizx [18] and Qexo [7], the two efficiency best XQuery implementations we are aware of. The tests were performed on the first five XML Query Use Cases [9] and on queries Q1, Q8, Q12, and Q16 of the XMark benchmark [24]. This set of tests gave a first positive answer to practical feasibility of CQL-pattern matching. We were pleased to notice that CQL was on the average noticeably faster than Qizx and Qexo especially when computing intensive queries such as joins[2] (cf. Q4 and Q5 use cases in Section 4 and query Q8 and Q12 of XMark). These results are even astounding if we consider that while Qizx and Qexo are compiled into bytecode and run on mature and highly optimised Java Virtual Machines (that of course we parametrised to obtain the best possible results), CDuce essentially is an interpreted language (it produces some intermediate code just to solve accesses to the heap) with just-in-time compilation of pattern matching. In the "todo" list of CDuce a high priority place is taken by the compilation of CDuce into OCaml bytecode. We are eager to redo our tests in such a setting, which would constitute a more fair comparison with the Java bytecode and should further increase the advantage of the CQL execution model.

Outline The article is organised as follows. In Section 2 we briefly recall CDuce features which are useful for understanding the definition of CQL: types, expressions and patterns. In Section 3 we present CQL's syntax and semantics. We give the typing rules for the defined language. We then present $\mathbb{C}\mathrm{QL}_X$ showing how to define projections. We formally define the translation from $\mathbb{C}\mathrm{QL}_X$ to CQL and show that such a translation yields an unnested CQL query and preserves typing. In Section 4 we propose several optimisations and in Section 5 we report on performance measurements we did. We draw our conclusions and present our current and future research directions in Section 6.

2 Types, expressions and patterns

A CQL query is written as

$$\mathsf{select}\ e_0\ \mathsf{from}\ p_1\ \mathsf{in}\ e_1,\ p_2\ \mathsf{in}\ e_2,\ldots,p_n\ \mathsf{in}\ e_n\ \mathsf{where}\ c$$

where the p_i's and e_i's respectively denote CDuce patterns and expressions. To define CQL then we have to define CDuce patterns and (a minimal subset of) expressions. A complete presentation of CDuce is beyond the scope of this paper (see instead the documentation—tutorial and user manual—and do try the prototype available at `www.cduce.org`), therefore we present here only (a subset of) CDuce values and just one complex expression, transform, used to define the semantics of CQL queries.

Since in CDuce/CQL patterns are types with capture variables let us start our presentation with them.

2.1 Types

CDuce type algebra includes three family of scalar types: (i) Integers, that are classified either by the type identifier Int, or by interval types i--j (where i and j are integer literals),

[2] We would like the reader to notice that we did not perform any further optimisation relying on specific data structure such as hash tables. Our very purpose was to assess CDuce pattern matching as an execution primitive for XML query processing in which XQuery could be compiled.

or by single integer literals like 42 that denotes the singleton type containing the integer 42. (*ii*) Characters, classified by the type identifiers Char (the Unicode character set) and Byte (the Latin1 character set), by intervals c--d (where c and d are Character literals that is single quoted characters like 'a', 'b', ..., or backslash-escaped directives for special characters, Unicode characters, ...), or by single character literals denoting the corresponding singleton types. (*iii*) Atoms that are user defined constants; they are ℂDuce identifiers escaped by a back-quote such as `nil, `true, ... and are ranged over by the type identifier Atom or by singleton types.

The other types of ℂDuce's type algebra are (possibly recursively) defined from the previous scalar types and the types Any and Empty (denoting respectively the universal and empty type) by the application of type *constructors* and type *combinators*.

Type combinators ℂDuce has a complete set of Boolean combinators. Thus if t_1 and t_2 are types, then $t_1\&t_2$ is their intersection type, $t_1 \mid t_2$ their union, and $t_1 \setminus t_2$ their difference. For instance the type Bool is defined in ℂDuce as the union of the two singleton types containing the atoms true and false, that is `true | `false.

Type constructors ℂDuce has type constructors for record types $\{ a_1=t_1;\ldots;a_n=t_n \}$, product types (t_1,t_2), and functional types $(t_1\text{->}t_2)$. For this paper the most interesting constructors are those for sequences and XML.

Sequence types are denoted by square brackets enclosing a regular expression on types. For instance, in ℂDuce strings are possibly empty sequences of characters of arbitrary length, and thus String is defined and implemented as [Char$*$] (i.e. it is just a handy shortcut). The previous type shows that the content of a sequence type can be conveniently expressed by regular expressions on types, which use standard syntax[3]:

$$R ::= t \mid R\,R \mid R|R \mid R* \mid R+ \mid R?$$

The general form of an XML type is $< t_1\ t_2 > t_3$ with t_i's arbitrary types. In practise t_1 is always a singleton type containing the atom of the tag, t_2 is a record type (of attributes), and t_3 a sequence type (of elements). As a syntactic facility it is possible to omit the back-quote in the atom of t_1 and the curly braces and semicolons in t_2, so that XML types are usually written in the following form: $<tag\ \ a_1=t_1\ \ a_2=t_2\ \ldots\ a_n=t_n>[\,R\,]$.

In the first row Figure 1 we report a DTD for bibliographies followed by the corresponding ℂDuce types: note the use of regular expression types to define the sequence types of elements (PCDATA is yet another ℂDuce convention to denote the regular expression Char*).

2.2 Expressions and patterns

Expression constructors mostly follow the same syntax as their corresponding type constructors, so a record expression has the form $\{ a_1=e_1;\ldots;a_n=e_n \}$, while a pair expression is (e_1,e_2). The same conventions on XML types apply to XML expressions: instead of writing <`book {year="1990"}>[...] we rather write <book year="1990">[...]. In the second row of Figure 1 we report on the left a document validating the DTD of the first row and on the right the corresponding (well-typed) value in ℂDuce: note that

[3] These are just a very convenient syntactic sugar (very XML-oriented) for particular recursive types.

XML	CDuce
<!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)	ℂDuce Types: type Bib = <bib>[Book*] type Book = <book year=String>[Title (Author+ | Editor+) Publisher Price] type Author = <author>[Last First] type Editor = <editor>[Last First Affiliation] type Title = <title>[PCDATA] type Last = <last>[PCDATA] type First = <first>[PCDATA] type Affiliation = <affiliation>[PCDATA] type Publisher = <publisher>[PCDATA] type Price = <price>[PCDATA]
<?xml version="1.0"?> <bib> <book year="1994"> <title>TCP/IP Illustrated</title> <author> <last>Stevens</last> <first>Richard</first> </author> <publisher>Addison-Wesley</publisher> <price> 65.95</price> </book> <book year="1984"> <title>The Lambda Calculus</title> <author> <last>Barendegt</last> <first>Henk</first> </author> <publisher>North-Holland</publisher> <price>92.00</price> </book> </bib>	<bib>[<book year="1994">[<title>"TCP/IP Illustrated" <author>[<last>"Stevens" <first>"Richard"] <publisher>"Addison-Wesley" <price>"65.95"] <book year="1984">[<title>"The Lambda Calculus" <author>[<last>"Barendegt" <first>"Henk"] <publisher>"North-Holland" <price>"92.00"]]

Fig. 1. DTD/ℂDuce-types and document/values for bibliographies

strings are not enclosed in brackets since they already denote sequences (of characters). Besides expression constructors there are also function definitions and operators (expression destructors). For the purpose of this article we are particularly interested in operators that work on sequences. Besides some standard operators, the most important operator for processing XML data (and the only ℂDuce iterator we present here) is transform, whose syntax is:

$$\text{transform } e \text{ with } p_1 \text{ -> } e_1 \mid p_2 \text{ -> } e_2 \mid \ldots \mid p_n \text{ -> } e_n$$

with $n \geq 1$ and where e, e_1, e_2, …, e_n are (expressions that return) sequences and p_1, p_2, …, p_n are *patterns* whose semantics is explained below.

The expression above scans the sequence e and matches each element of e against the patterns, following *a first match policy* (that is, first against p_1 then, only if it fails, against p_2, and so on). If some p_i matches, then the corresponding e_i is evaluated in an environment where variables are substituted by the values captured by the pattern. If no pattern matches, then the empty sequence is returned. When all the elements of e have been scanned, transform returns the concatenation of all results.[4]

[4] In short, transform differs from the classic map (also present in ℂDuce) since it uses pattens to filter elements of a sequence and therefore, contrary to map it does not preserve the length of the sequence.

Clearly, in order to fully understand the semantics of transform we need to explain the semantics of patterns. The simplest patterns are variables and types: a variable pattern, say, x always succeeds and captures in x the value it is matched against. If *e* is a sequence of integers then transform *e* with x -> (if x>=0 then [x] else [ ]) returns the subsequence of *e* containing all the positive integers. A type pattern *t* instead succeeds only when matched against a value of type *t*. More complex patterns can be constructed by using type constructors and/or the combinators "&" and "|". So $p_1\&p_2$ succeeds only if both p_1 and p_2 succeed (p_1 and p_2 must have pairwise distinct capture variables), while $p_1|p_2$ succeeds if p_1 succeeds or p_1 fails and p_2 succeeds (p_1 and p_2 must have the same set of capture variables). For instance the pattern x&Int succeeds only if matched against an integer, and in that case the integer at issue is bound to x. Since the type of positive integers can be expressed by the interval 0--$*$ (in integer intervals $*$ stands for infinity) then the previous transformation can be also written as transform *e* with x&(0--$*$) -> [x] . We can use more alternatives to transform the negative elements into positive ones instead of discarding them: transform *e* with x&(0--$*$) -> [x] | x&($*$--0) -> [-x].

If we know that *e* is a sequence formed only of integers, then in the expression above we can omit "&($*$--0)" from the second pattern as it constitutes redundant information (actually ℂDuce automatically gets rid at compile time of all redundant information).

Patterns built by type constructors are equally simple. For instance, the pattern <book year=y>[<title>t <author>[_ f] ;_] matches any bibliographic entry binding to y the value of the attribute year, to t the string of the title, and to f the <first> element of the first author name. The wildcard _ is often used in patterns as a shorthand for the type Any (it matches any value, in the case above it matches the <last> element in the name) while ";_" matches tails of sequences.

Assuming that books denotes a variable of type [Book$*$] the code below:

```
transform books with
  | <book year=("1999"|"2000")>[ _  <author>[ _ <first> f ] ;_ ] -> [ f ]
  | <_>[ _  <author>[<last>s ;_] ;_ ] -> [s]
```

scans the sequence of elements of books and for each book it returns the string of the first name if the book was published in 1999 or 2000, or the string of the last name otherwise.

Besides variables and types there are two (last) more building blocks for patterns: default patterns and sequence capture variables.

Default patterns are of the form (x:=*v*); the pattern always succeeds and binds x to the value *v*. Default patterns are useful in the last position of an alternative pattern in order to assign a default value to capture variables that did not match in the preceding alternatives. This allows the programmer to assign default values to optional elements or attributes. For instance imagine that we want to change the previous transform so that if the publication year is not 1999 or 2000 it returns the last name of the *second* author element if it exists, or the string "none" otherwise. It will be changed to:

```
transform books with
  | <book year=("1999"|"2000")>[ _  <author>[ _ <first> f ] ;_ ] -> [ f ]
  | <_>[ _  Author  <author>[<last>s _]  ;_ ]  |  (s:="none") -> [ s ]
```

We guarded the second branch by an alternative pattern assigning "none" to s when the first pattern fails (that is, when there is no second author). Note that the string "none" is

$$\frac{(\ var(p_i)\wedge var(p_j)=\varnothing,\ \text{for } i\neq j\ )\qquad \Gamma,(t_1/p_1),\ldots,(t_{i-1}/p_{i-1})\vdash e_i : [t_i+] \quad \Gamma,(t_1/p_1),\ldots,(t_n/p_n)\vdash e : t,\ c : Bool}{\Gamma\vdash \text{select } e \text{ from } p_1 \text{ in } e_1,\ \ldots, p_n \text{ in } e_n \text{ where } c\ :\ [\,t*\,]}\ (select)$$

Fig. 2. Typing rule for queries

returned also when the book has editors instead of authors (see the definition of Book type in Figure 1). To filter out books with only editors, the pattern of the second branch should be <_>[_ (Author (<author>[<last>s _] | (s:="none")) ;_]. The pattern succeeds if and only if the title is followed by an author, in which case either it is followed by a second author (whose lastname is then captured by s), or by a publisher (and s is then bound to "none").

Sequence capture variables patterns are of the form x::*R* where *R* is a type regular expression; the pattern binds x to a *sequence* of elements. More precisely it binds *x* to the sequence of all elements matching *R* (regular expressions match sequences of elements rather than single elements). Such patterns are useful to capture whole subsequences of a sequence. For instance, to return for each book in the bibliography the list of *all* authors and to enclose it in a <authors> tag can be done compactly as follows:

```
transform books with <book>[ _  (a::Author+) ;_ ] -> [ <authors>a ]
```

Note the difference between [x::Int] and [(x & Int)]. Both accept sequences formed of a single integer, but the first one binds x to a sequence (of a single integer), whereas the second one binds it to the integer itself.

Finally we want to stress that the type inference algorithm of ℂDuce/ℂQL is better than that of XQuery since it always infer a type more precise than the one inferred by XQuery. An example can be found in the extended version of this paper [5].

3 ℂQL: a Pattern-based query language for XML processing

The formal syntax of ℂQL is given by the following grammar:

Queries

q::=select e from f where c | select e from f

Bindings

f::=p in e , f | p in e

Conditions

c::=‘true | ‘false | not(c) | c or c | c and c | member(e , e) | e *bop* e

Expressions

e::=x | v | [$e\ldots e$] | flatten(e) | q | ⟨ $e\ \ell=e\ldots\ell=e$ ⟩ e | *op*(e)

Patterns

p::=x | t | p&p | p | p | (p,p) | ⟨ $p\ \ell=p\ldots\ell=p$ ⟩ p | [r] | (x:=v)

Pattern regular expressions

r::=p | (x :: r) | r | r | $r\,r$ | r+ | $r*$ | r?

Types

t::=B | t | t | t&t | $t\setminus t$ | (t,t) | ⟨ $t\ \ell=t\ldots\ell=t$ ⟩ t | [R] | Empty | Any

where *op* ranges over sequence operators (*op*∈{distinct_values, count, avg, max, min, sum}), *bop* over boolean relations (*bop*∈{=, >>, >=, <<, <=}), x over variables, and v over *val-*

ues (*viz.* closed expressions in normal formal and constants for integers and characters); flatten takes a sequence of sequences and returns their concatenation (thus, for instance, the infix operator @ that denotes the concatenation of two sequences is encoded as $e_1@e_2 = \mathsf{flatten}\,[e_1\ e_2]$).

The non-terminal R used in the definition of types is the one defined in Section 2.1. Patterns, ranged over by p, and types, ranged over by t, are simplified versions of those present in ℂDuce and have already been described; note that types include full boolean combinations: intersection ($t\,\&\,t$), union ($t \mid t$), and difference ($t \setminus t$). The reader can refer to [4] for a more detailed and complete presentation.

```
<books-with-prices>
select <book-with-price>[t1
                <price-amazon>p2
                <price-bn>p1 ]
from <bib>[b::Book*] in [biblio],
     <book>[t1&Title _* <price>p1] in b,
     <reviews>[e::Entry*] in [amazon],
     <entry>[t2&Title <price>p2 ;_] in e
where t1=t2
```

As an example, the query Q_5 of XQuery described in the introduction would be written in ℂQL as shown on the left-hand side.

The typing rule for the select-from-where construction is given in Figure 2. It states that the condition c must be of type Bool and that the e_i's must be non-empty homogeneous sequences. In this rule (t/p) is the type environment that assigns to the variables of p the best type deduced when matching p against a value of type t and $var(p)$ is the set of variables occurring in p [5] (see [4] for formal definitions and the optimality of the deduced types).

```
transform e1 with p1 –>
  transform e2 with p2 –>
                ⋱
          transform en with pn –>
            if c then [e0] else [ ]
```

The semantics of a select-from-where expression (in the form as it is at the beginning of Section 3) is defined in terms of the translation into ℂDuce given on the left-hand side. In our context transform plays exactly the same role as the "for" construct of XQuery core does [17]. However, the peculiar difference is that our pattern matching compilation schema is based on non-uniform tree automata which fully exploit types to optimise the code [19] as well as its execution. This translation is given only to define in an unambiguous way the semantics of the new term. It is not intended to impose any execution order, since such a choice is left to the query optimiser. In fact the optimiser can implement this more efficiently; for instance, if c does not involve the capture variables of some p_i, the query optimiser can, as customary in databases, push selections (and/or projections) on some components as shown in Section 4.

3.1 ℂQL$_X$

In order to investigate and compare pattern-matching with XQuery, we have extended ℂQL with projection operators *à la* XPath. Let e, be an expression denoting a sequence of XML elements, and t be a ℂDuce type, then e/t denotes the set of children of the elements in e whose type is t. The formal semantics is defined by encoding: $e/\!/t$ is encoded as transform e with <_>[(x::t | _)*] –> x. It is convenient to introduce the syntax $e/@a$ to extract the sequence of all values bound to the attribute a of elements in e, which is encoded in ℂDuce as transform e with <_ a=x>_ –> [x]

[5] The condition $var(p_i) \wedge var(p_j) = \varnothing$ for $i \neq j$ is not strictly necessary but may be useful in practise and simplifies both the proofs and the definition of the optimisations.

XQuery:

```
<bib>
 {
  for $b in $biblio/bib/book
  where $b/publisher = "Addison-Wesley"
            and $b/@year > 1990
  return
    <book year="{ $b/@year }">
     { $b/title }
    </book>
 }
</bib>
```

$\mathbb{C}$QL$_X$:

```
<bib> select <book year=y>[t]
from  b in [biblio]/<book>_ ,
        p in [b]/<publisher>_ ,
        t  in [b]/<title>_ ,
        y in [b]/@year
where (p = <publisher>"Addison-Wesley")
          and (y>>"1990"));;
```

$\mathbb{C}$QL:

```
<bib> select <book year=y>[t]
from  <bib>[b::Book*] in [biblio],
        <book year=y>[t&Title _+
                    <publisher>"Addison-Wesley";_] in b;;
where y>>"1990"
```

Fig. 3. XQuery and the two $\mathbb{C}$QL programming styles.

Figure 3 illustrates how to code the same query in XQuery, $\mathbb{C}$QL$_X$, and $\mathbb{C}$QL. The query at issue is the query Q1 from the XQuery Use Cases. While XQuery and $\mathbb{C}$QL$_X$ code make use of simple variables that are bound to the results of projections, the $\mathbb{C}$QL one fully exploits the pattern algebra of $\mathbb{C}$Duce (we leave as an exercise to the reader how to use regular expressions on types to modify the pattern in the second from clause of the $\mathbb{C}$QL query so as to completely get rid of the where clause).

Finally, since we use $\mathbb{C}$QL$_X$ to mimic XQuery in a full pattern setting, it is important to stress that the semantics of where clauses in $\mathbb{C}$QL (hence in $\mathbb{C}$QL$_X$) is not exactly the same as in XQuery. The latter uses a set semantics according to which a value satisfying the where close will occur at most once in the result (e.g. as for `SELECT DISTINCT` of SQL), while $\mathbb{C}$QL/$\mathbb{C}$QL$_X$ follow the SQL convention and use a multi-set semantics. As usual, the multi-set semantics is more general since the existential semantics can easily obtained by using the distinct_values operator (which takes a sequence and elides multiple occurrences of the same element). The presence of such an operator has no impact on the translation from $\mathbb{C}$QL$_X$ to $\mathbb{C}$QL.

3.2 Translation from $\mathbb{C}$QL$_X$ to $\mathbb{C}$QL

It is quite easy to see that projections can be encoded in $\mathbb{C}$QL, since the two transform expressions used to encode projections correspond, respectively, to: flatten(select x from <_>[(x::t | _)$*$] in e), and to select x from <_ a=x>_ in e. Nonetheless it is interesting to define a more complex translation from $\mathbb{C}$QL$_X$ to $\mathbb{C}$QL that fully exploits the power of $\mathbb{C}$Duce patterns to optimise the code. Therefore in this section we formally define a translation that aims at (i) eliminating projections and pushing them into patterns (ii) transforming as many as possible selection conditions into patterns. As a result of this translation the number of iterations is reduced and several where clauses are pushed into patterns. In order to formally define the translation we first need to introduce the expression and evaluation contexts $E\{\ \}$ and $C\{\ \}$:

$\bar{q}$ ::= select $\bar{e}$ from $\bar{f}$ where $\bar{c}$ | select $\bar{e}$ from $\bar{f}$

$\bar{e}$ ::= x | v | [$\bar{e}\ldots\bar{e}$] | flatten($\bar{e}$) | $\{\ \ell = \bar{e}, \ldots \ell = \bar{e}\ \}$ | $\bar{e}.\ell$ | ($\bar{e}$, $\bar{e}$) | $op(\bar{e})$ | $\bar{q}$

$\bar{f}$ ::= p in $\bar{e}$,$\bar{f}$ | p in $\bar{e}$

$\bar{c}$ ::= 'true | 'false | not($\bar{c}$) | $\bar{c}$ or $\bar{c}$ | $\bar{c}$ and $\bar{c}$ | $\bar{e}$ bop $\bar{e}$ | member($\bar{e}$, $\bar{e}$)

$E\{\,\} ::= \{\,\} \mid [e_1 \ldots e_n\, E\{\,\}\, \bar{e}_q \ldots \bar{e}_m] \mid \mathsf{flatten}(E\{\,\}) \mid \langle \bar{e}\ \ell = e \ldots \ell = e \rangle E\{\,\}$
$\mid \langle \bar{e}\ \ell = e \ldots \ell = E\{\,\}\ \ell = \bar{e} \ldots \rangle.\bar{e} \mid (E\{\,\}, \bar{e}) \mid E\{\,\}/t \mid E\{\,\}/@a \mid op(E\{\,\})$

$C\{\,\} ::= \{\,\} \mid \mathsf{not}(C\{\,\}) \mid c \text{ or } C\{\,\} \mid C\{\,\} \text{ or } \bar{c} \mid c \text{ and } C\{\,\} \mid C\{\,\} \text{ and } \bar{c}$
$\mid e\ bop\ E\{\,\} \mid E\{\,\}\ bop\ \bar{e} \mid \mathsf{member}(e, E\{\,\}) \mid \mathsf{member}(E\{\,\}, \bar{e})$

where $m,n \geq 0$.

Definition 1. *The translation $\mathcal{P}[\![\ ]\!]$ is defined by the following rewriting rules:*

- $\mathcal{P}[\![\text{ select } E\{q\} \text{ from } f \text{ where } c]\!] = \mathcal{P}[\![\text{ select } E\{\mathcal{P}[\![q]\!]\} \text{ from } f \text{ where } c]\!]$, *$q$ is not a $\bar{q}$ expression*
- $\mathcal{P}[\![\text{ select } E\{\bar{e}/t\} \text{ from } f \text{ where } c]\!] = \mathcal{P}[\![\text{ select } E\{x\} \text{ from } f, x \text{ in } [\bar{e}/t] \text{ where } c]\!], x \notin bv(f)$
- $\mathcal{P}[\![\text{ select } E\{\bar{e}/@a\} \text{ from } f \text{ where } c]\!] = \mathcal{P}[\![\text{ select } E\{x\} \text{ from } f, x \text{ in } [\bar{e}/@a] \text{ where } c]\!], x \notin bv(f)$
- $\mathcal{P}[\![\text{ select } \bar{e} \text{ from } f \text{ where } C\{q\}]\!] = \mathcal{P}[\![\text{ select } \bar{e} \text{ from } f \text{ where } C\{\mathcal{P}[\![q]\!]\}]\!]$, *$q$ is not a $\bar{q}$ expression*
- $\mathcal{P}[\![\text{ select } \bar{e} \text{ from } f \text{ where } C\{\bar{e}/t\}]\!] = \mathcal{P}[\![\text{ select } \bar{e} \text{ from } f, x \text{ in } [\bar{e}/t] \text{ where } C\{x\}]\!], x \notin bv(f)$
- $\mathcal{P}[\![\text{ select } \bar{e} \text{ from } f \text{ where } C\{\bar{e}/@a\}]\!] = \mathcal{P}[\![\text{ select } \bar{e} \text{ from } f, x \text{ in } [\bar{e}/@a] \text{ where } C\{[x]\}]\!], x \notin bv(f)$
- $\mathcal{P}[\![\text{ select } \bar{e} \text{ from } f \text{ where } \bar{c}]\!] = \mathcal{P}[\![\text{ select } \bar{e} \text{ from } \mathcal{F}[\![f, \ldots]\!]_{bv(f)} \text{ where } \bar{c}]\!]$

where $\mathcal{F}[\![\ ]\!]$ is defined as:

- $\mathcal{F}[\![p \text{ in } e, f]\!]_\Gamma = \mathcal{F}[\![p \text{ in } e]\!]_\Gamma, \mathcal{F}[\![f]\!]_{\Gamma \cup bv(\mathcal{F}[\![p \text{ in } e]\!]_\Gamma)}$
- $\mathcal{F}[\![p \text{ in } E\{q\}]\!]_\Gamma = \mathcal{F}[\![p \text{ in } E\{\mathcal{P}[\![q]\!]\}]\!]_\Gamma$
- $\mathcal{F}[\![p \text{ in } E\{\bar{e}/t\}]\!]_\Gamma = $ *if* $\bar{e}$ *has type* [Any] *then* <_>[(x::t|_)*] in $\bar{e}$, $\mathcal{F}[\![p \text{ in } E\{x\}]\!]_{\Gamma \cup \{x\}}$, $x \notin \Gamma$
  else [(<_>[(x::t|_)*] | x::'nil)*] in [$\bar{e}$], $\mathcal{F}[\![p \text{ in } E\{\mathsf{flatten}(x)\}]\!]_{\Gamma \cup \{x\}}$, $x \notin \Gamma$
- $\mathcal{F}[\![p \text{ in } E\{\bar{e}/@a\}]\!]_\Gamma = $ *if* $\bar{e}$ *has type* [Any] *then* <_ a=x>_ in $\bar{e}$, $\mathcal{F}[\![p \text{ in } E\{[x]\}]\!]_{\Gamma \cup \{x\}}$, $x \notin \Gamma$
  else [(<_ a=x>_ | x::'nil)*] in [$\bar{e}$], $\mathcal{F}[\![p \text{ in } E\{x\}]\!]_{\Gamma \cup \{x\}}$, $x \notin \Gamma$
- $\mathcal{F}[\![p \text{ in } \bar{e}]\!]_\Gamma = p \text{ in } \bar{e}$

Apart from technical details, the translation is rather simple: contexts are defined so that projections are replaced according to an innermost-rightmost strategy. For instance if we have to translate x/t1/t2, (x/t1) will be considered first thus removing the projection on t1 prior to performing the same translation on the remainder. The first three rules replace projections in the select part, 2nd and 3rd rules incidentally perform a slight optimisation (they get rid of one level of sequence nesting) in the case the projection is applied to a sequence with just one element (this case is very common and dealing with it allows for further optimisation later on); the 4th, 5th, and 6th rules replace projections in the "where" part by defining new patterns in the "from" part, while the 7th rule handles projections in the "from" part. The latter resorts to an auxiliary function $\mathcal{F}$ that needs to store in a set Γ the capture variables freshly introduced.

So far, we have proved just a partial correctness property of the translation, namely that it preserves the types (the proof is omitted for space reasons):

Theorem 1 (Type preservation). $\Gamma \vdash q : t \Rightarrow \Gamma \vdash \mathcal{P}[\![q]\!] : t$

The result of $\mathcal{P}$ is a query in $\mathbb{C}$QL since all projections have been removed. From a practical viewpoint, the use of a projection in a query is equivalent to that of a nested query.

```
<bib>
select <book year=y>[t]
from b in (select v from <_>[(yb::Book|_)*] in [biblio], v in yb)
     p in (select v from <_>[(yp::Publisher|_)*] in [b], v in yp)
     t  in (select v from <_>[(yt::Title|_)*] in [b], v in yt)
     y in (select yy from <_year=yy>_ in [b])
 where (p = <publisher>"Addison-Wesley") and (y>>"1990")
```

Let Q be the query obtained from the $\mathbb{C}$QL$_X$ query in Figure 3 by replacing <book>_, <publisher>_, and <title>_ respectively by

Book, Publisher, and Title (the resulting query is semantically equivalent but more readable and compact). If we expand the projections of Q into its corresponding sequence of select's we obtain the query at the end of the previous page (we used boldface to outline modifications).

The translation $\mathcal{P}[\![\ ]\!]$ unnests these select's yielding the query of Figure 4. In the next section we show that this has a very positive impact on performances.

```
<bib>
 select <book year=y>[t]
 from <_>[(yb::Book|_)*] in [biblio],
      b in yb,
      <_>[(yp::Publisher|_)*] in [b],
      p in yp,
      <_>[(yt::Title|_)*] in [b],
      t in yt,
      <_ year=y>_ in [b]
 where (p = <publisher>"Addison-Wesley")
         and (y>>"1990")
```

Fig. 4. $\mathcal{P}[\![\ Q\ ]\!]$

4 Pattern Query Optimisation

In this section we adapt classical database optimisation techniques to our patterns. Such optimisations evaluate, as customary in the database world, conditions just when needed thus reducing the size of intermediate data contributing in the result construction and also avoiding to visit useless parts of the document. More precisely, we proceed in four steps.

Conjunctive Normal Form. The first step consists in putting the where condition in conjunctive normal form and then moving into the from clause the parts of the condition that can be expressed by a pattern. This is expressed by the following rewriting rule:

select e from f where c $\quad\leadsto\quad$ select e from f,$\Theta^1(CNF(c))$ where $\Theta^2(CNF(c))$

where CNF(c) is a conjunctive normal form of c, $\Theta^1(c)$ represents the part of c that can be expressed by a pattern and thus remounted in the "from" part, and $\Theta^2(c)$ is the part of c that remains in the "where" condition. Formally, Θ^1 and Θ^2 are the first and second projections of the function Θ defined as:

Definition 2. *Let i denote a scalar (i.e. an integer or a character) and v a value*

$\Theta(v\text{=}e) = (v \text{ in } [e], \text{'true})$

$\Theta(e \text{ >= } i) = (\ i\text{--}* \text{ in } [e], \text{'true})$

$\Theta(e \gg\ i) = (\ [\![i+1]\!]\text{--}* \text{ in } [e], \text{'true})$

$\Theta(e \text{ <= } i) = (\ *\text{--}i \text{ in } [e], \text{'true})$

$\Theta(e \ll i) = (\ *\text{--}[\![i-1]\!] \text{ in } [e], \text{'true})$

$\Theta(\text{count}(e) \text{ = } i) = (\ [\ \_^i] \text{ in } [e], \text{'true})$

$\Theta(\text{count}(e) \gg i) = (\ [\ \_^i \ \_+] \text{ in } [e], \text{'true})$

$\Theta(\text{count}(e) \text{ >= } i) = (\ [\ \_^i \ \_^*] \text{ in } [e]\ , \text{'true})$

$\Theta(\text{count}(e) \ll i) = (\ [(\_?)^{i-1}] \text{ in } [e]\ , \text{'true})$

$\Theta(\text{count}(e) \text{ <= } i) = (\ [(\_?)^i] \text{ in } [e], \text{'true})$

$\Theta(\text{member}(v,e)) = (\ [\ \_^*\ v\ \_^*] \text{ in } [e], \text{'true})$

$\Theta(c_1 \text{ and } \ldots \text{ and } c_n) = (\ (\Theta^1(c_1), \ldots, \Theta^1(c_n))\ , \Theta^2(c_1) \text{ and } \ldots \text{ and } \Theta^2(c_n)\ ).$

$\Theta(c) = (\varepsilon, c)$ *(if none of the above applies)*

where we use the notation $\_^i$ to denote the juxtaposition of i occurrences of "_". So for instance the third rule indicates that the constraint, say, count(e) = 3 can be equivalently checked by matching e against the pattern [_ _ _] (i.e. [$\_^3$]). We also used the notation $[\![f(i)]\!]$ for the constant that is the result of $f(i)$.

If we apply the rewriting to the query in Figure 4, then we can use the first case of Definition 2 to express the condition p = <publisher>"Addison-Wesley" by a pattern,

```
<bib>
 select <book year=y>[t]
 from <_>[(yb::Book_)*] in [biblio],
      b in yb,
      <_>[(yp::Publisher|_)*] in [b],
      p in yp,
      <_>[(yt::Title|_)*] in [b],
      t in yt,
      <_ year=y>_ in [b],
      <publisher>"Addison-Wesley" in [p]
 where y>>"1990"
```

Fig. 5. Θ on $\mathcal{P}[\![\ Q\ ]\!]$

```
<bib>
select <book year=y>[t]
from <_>[(yb::Book|_)*] in [biblio],
      <_ year=y>_&
      <_>[(yp::Publisher|_)*]&
      <_>[(yt::Title|_)*] in yb,
      <publisher>"Addison-Wesley" in yp,
      t in yt
where y>>"1990"
```

Fig. 6. $\mathcal{P}[\![\ Q\ ]\!]$ after the first 3 optimisation steps

yielding the query of Figure 5 (the rewriting does not apply to y ≫ "1990" since "1990" is not a scalar but a string).

Useless Declarations Elimination. The second step consists in getting rid of useless intermediate in-declarations that are likely to be generated by the translation of $\mathbb{C}$QL$_X$ into $\mathbb{C}$QL:

select $e_\circ$ from f_1, x in e, f_2, p in $[x], f_3$ where $c \rightsquigarrow$ select $e_\circ$ from $f_1, x\&p$ in e, f_2, f_3 where c

Pattern Consolidation. The third step of optimisation consists in gathering together patterns that are matched against the same sequences.

select $e_\circ$ from f_1, p_1 in e, f_2, p_2 in e, f_3 where $c \rightsquigarrow$ select $e_\circ$ from $f_1, p_1\&p_2$ in e, f_2, f_3 where c

Note that both rewriting systems are confluent and noetherian (when applied in the order). The result of applying these rewriting rules to the query in Figure 5 is shown in Figure 6.

Pushing Selections The fourth and last step is the classical technique that pushes selections as close as possible to the declarations of the variables they use. So for instance the clause y ≫ "1990" in Figure 6 is checked right after the capture of y. This is obtained by anticipating an if_then_else in the implementation[6]. The algorithm, which is standard, is omitted.

This kind of optimisation is definitely not new. It corresponds to a classical logical optimisation for the relational model. The benefit is to reduce the size of the intermediate data that is used to the obtain the result.

5 Experimental Validation

Performance measurements were executed on a Pentium IV 3.2GHz with 1GB of RAM running under FreeBSD 5.2.1. We compared performance results of the same query programmed in the different flavors of $\mathbb{C}$QL. So we tested a same query in: (i) $\mathbb{C}$QL$_X$, that is $\mathbb{C}$QL in which we use the same path expressions as the XQuery query and no $\mathbb{C}$Duce pattern, (ii) $\mathbb{C}$QL$_P$, the $\mathbb{C}$QL program automatically generated by applying to

[6] More precisely, the query in Figure 6 is implemented by

```
transform [biblio] with <_>[(yb::Book|_)*] ->
   transform yb with <_ year=y>[(yp::Publisher|yt::Title|_)*] ->
     if (y>>"1990") then transform yp with <publisher>"Addison-Wesley" -> [<book year=y>[t]]
        else [ ]
```

	Size	Size2	ft $_{CQL}$	$\mathbb{C}$QL$_X$	$\mathbb{C}$QL$_X^{opt}$	$\mathbb{C}$QL$_P$	$\mathbb{C}$QL$_P^{opt}$	$\mathbb{C}$QL	Qizx	Qexo
Q1	36 Kb		0.01	0.01	0.01	0.02	0.01	0.01	0.45	0.60
Q1	1.8 Mb		0.23	0.26	0.25	0.26	0.26	0.24	0.76	1.01
Q1	14 Mb		1.90	2.00	1.99	1.98	2.07	1.93	2.18	2.89
Q1	35 Mb		4.79	5.13	5.04	5.03	5.24	4.90	4.44	5.80
Q2	36 Kb		0.01	0.01	0.01	0.01	0.01	0.01	0.46	0.61
Q2	1.8 Mb		0.24	0.26	0.26	0.25	0.25	0.25	1.00	1.04
Q2	14 Mb		1.87	2.06	2.06	2.01	2.01	1.99	3.77	3.55
Q2	35 Mb		4.74	5.27	5.27	5.14	5.14	5.08	8.16	7.79
Q3	36 Kb		0.01	0.01	0.01	0.01	0.01	0.01	0.47	0.60
Q3	1.8 Mb		0.24	0.25	0.26	0.25	0.25	0.25	0.99	1.03
Q3	14 Mb		1.90	2.03	2.02	2.01	2.02	2.01	3.66	3.27
Q3	35 Mb		4.81	5.18	5.18	5.14	5.14	5.13	7.90	6.86
Q4	36 Kb		0.01	0.05	0.05	0.05	0.05	0.05	0.53	
Q4	70 Kb		0.02	0.17	0.17	0.14	0.14	0.14	0.68	
Q4	144 Kb		0.02	0.61	0.61	0.52	0.52	0.49	1.17	
Q4	575 Kb		0.09	10.73	10.73	9.94	9.94	8.63	10.97	
Q4	1.8 Mb		0.24	113.01	113.01	89.31	89.31	88.70	104.12	
Q5	36 Kb	535 Kb	0.08	1.69	0.79	1.17	0.71	0.54	4.44	27.88
Q5	144 Kb	43 Kb	0.03	0.52	0.24	0.38	0.24	0.17	1.79	9.31
Q5	575 Kb	171 Kb	0.11	7.87	3.49	5.92	3.34	2.46	20.74	127.39
Q5	1.8 Mb	535 Kb	0.31	78.27	36.54	53.25	31.04	22.93	197	>1h
Q5	3.5 Mb	535 Kb	0.55	157.70	72.28	105.38	62.24	45.02	392	

(ft = file load time, Size2 column reports the sizes of the second document in joins)

Fig. 7. Summary of all test results on the XQuery Use Cases

the $\mathbb{C}$QL$_X$ query the transformation $\Theta(\mathcal{P}[\![\ ]\!])$ of Sections 3.2 and 4 and the two rewritings for pattern consolidation and useless declaration elimination that clean up the "garbage" introduced by the translation, (iii, iv) $\mathbb{C}$QL$_X{}^{opt}$ $\mathbb{C}$QL$_P{}^{opt}$, which are obtained by optimising the two previous queries by the classical optimisation algorithm of pushing selections, presented at the end of Section 4, and finally (v) $\mathbb{C}$QL that is a handwritten (and hand optimised) query in $\mathbb{C}$QL. As we explained in the introduction, in order to strengthen our results we chose not to use "//" in $\mathbb{C}$QL $_X$ (since this is less heavily optimised by the $\mathbb{C}$Duce runtime than "/") and instead always use the static types to translate it in terms of "/" (as all the queries we considered allowed us to do so). This gives a clear further advantage to $\mathbb{C}$QL$_X$.

To perform our tests we chose, somewhat arbitrarily, queries Q1, Q2, Q3, Q4, and Q5 of the XML Query Use Cases. We then performed a second set of tests based on the XMark benchmarks. Here our choice of queries was less random as we explain below.

Finally to test $\mathbb{C}$QL runtime we compared our results with three different implementations of XQuery: Galax [3], Qizx [18], and Qexo [7, 8]. Galax is a reference implementation of XQuery and, as $\mathbb{C}$Duce, it is implemented in OCaml. Qizx/open is an open-source Java implementation of XQuery specifications developed for commercial distribution and whose target is the efficient execution of queries over large databases. Qexo is a partial implementation of the XQuery language that achieves high performance by compiling queries down to Java bytecode using the Kawa framework. The sources of the queries are omitted for space reasons but they can all be found in the extended version [5].

5.1 Use Cases

Briefly, query Q1 performs a simple selection. Queries Q2 and Q3 are "reconstructing" queries, they both scan the whole bibliography, while the first one returns a flat list of title author pairs

(each pair being enclosed in a <result> element), the second returns the title and all authors grouped in a <result> element. For each author in the bibliography, query Q4 lists the author's name and the titles of all books by that author, grouped inside a "result" element. Last, query Q5 performs a join between two documents: for each book found both in the document bound to biblio and in that bound to amazon Q5 lists the title of the book and its price from each source.

The results of our tests are presented in Table 7, from which we omitted the times for Galax: as a matter of fact we did not completed all the tests of Galax since it was soon clear that the performances of Galax are several orders of magnitude worse than those of Qizx and Qexo.

Measurements were performed for each query on randomly generated documents of different sizes (expressed in KBytes). We also followed the spirit of the benchmark and we generated documents with a selectivity rate (that we judged) typical of the bibliographic application (that is quite low). Each test was repeated several times and the table reports the average evaluation times (in seconds). We have reported the loading time (in the column headed by "ft", file load time) of the XML document from the global execution time in order to separate the weight of the query engine and that of the parser in the overall performances (of course we are interested in the former and not in the latter). The execution times always include the time for performing optimisation, when this applies, for type checking (just for ℂQL variants) and the file load time.

	Size	ft $_{CQL}$	ℂQL$_X$	ℂQL$_P$	ℂQL	Qizx	Qexo
Q1	1.5 Mb	0.15	0.15	0.15	0.15	0.57	0.74
Q1	29 Mb	2.57	2.58	2.58	2.58	2.16	2.58
Q1	72 Mb	6.61	6.65	6.64	6.62	4.42	5.08
Q1	145 Mb	14.10	14.18	14.15	14.13	8.16	9.31
Q8	1.5 Mb	0.15	0.21	0.21	0.17	1.00	34.51
Q8	29 Mb	2.57	26.03	22.96	13.09	75.90	>1h
Q8	72 Mb	6.61	156	133	72.81	476	
Q8	145 Mb	14.19	630	542	285	1838	
Q12	1.5 Mb	0.16	0.21	0.21	0.20	0.87	
Q12	29 Mb	2.59	21.22	20.57	14.70	38.30	
Q12	72 Mb	6.68	127	122	86.35	216	
Q12	145 Mb	14.36	481	457	319	824	
Q16	1.5 Mb	0.15	0.16	0.16	0.16	0.62	0.78
Q16	29 Mb	2.57	2.65	2.64	2.63	2.15	2.63
Q16	72 Mb	6.63	6.87	6.85	6.82	4.42	5.08
Q16	145 Mb	14.24	14.60	14.54	14.50	8.16	9.31

(ft = file load time)

Fig. 8. Summary of all test results on XMark

By comparing the load time with the overall execution time (we recall that the latter includes the former) it is clear that the only computationally meaningful queries are the Q4 and Q5 ones (Q4 was not executed in Qexo since it does not implement the distinct_values operator). In these two cases the best performances are obtained by ℂQL.[7]

5.2 XMark

Following the suggestion of Ioana Manolescu (one of the XMark authors) we chose four queries that should give a good overview of the main memory behaviour of the query engines.

More precisely, our choice went on Q1, just because it is the simplest one, Q8 of the "chasing references" section since it performs horizontal traversals with increasing complexity, Q12 of the

[7] A word must be spent on the performances of Q5 for Qizx. Whenever Qizx syntactically detects a conjunctive join condition it dynamically generates a hash table to index it (for low selective benchmarks, as the one we performed in our tests, this brings far better performance, of course). Since we wanted to compare the performances of the query engines (rather than the OCaml and Java libraries for hash tables) and because we believe that index generation must be delegated to the query optimiser rather than implemented by the compiler, then in our test we disabled this index generation (this is done by just adding an "or false" to the join condition).

"join on values" section since it tests the ability to handle large intermediate results, and finally on Q16 of the "path traversals" section to test vertical traversals.

The results are summarised in Table 8. We did not perform the tests on the optimised versions of $\mathbb{C}\mathrm{QL}_X$ and $\mathbb{C}\mathrm{QL}_P$ since on the specific queries they are the identity function. Q12 times for Qexo are absent because execution always ended with an unhandled exception (probably because of a bug in the implementation of the arithmetic library of Qexo but we did not further investigate).

Once more if we compare the load time with the execution time we see that the only interesting queries to judge the quality of the engines are Q8 and Q12. In the other queries the execution time is very close to the load time, so the performance is completely determined by parsers. In these cases it is interesting to note that while $\mathbb{C}$QL uses an external parser, namely Expat (but the $\mathbb{C}$Duce interpreter has an option that the programmer can specify to use the less efficient but more debug-friendly Pxp parser), Qexo and Qizx have event driven parsers that interact with the query engines in order to avoid loading of useless parts of the document, whence the better performances. That said, when performances are determined by the query engine, as in Q8 and Q12, $\mathbb{C}$QL shows once more the best results[8].

6 Conclusion and Perspectives

In this article we presented $\mathbb{C}$QL, a full pattern-matching based query language for XML embedded in $\mathbb{C}$Duce. Our main purpose was to demonstrate that patterns and pattern matching (in $\mathbb{C}$Duce sense) are a good candidate as an evaluation mechanism for XML query processing. To do so, we first coupled $\mathbb{C}$Duce patterns with a syntax very common in the database area (select-from-where) and defined the new syntax in terms of a translation semantics. Then, we extended this syntax to allow for an XQuery-like style ($\mathbb{C}\mathrm{QL}_X$). We chose to experiment with $\mathbb{C}\mathrm{QL}_X$ rather than with XQuery because we wanted to rely on the very efficient pattern matching compilation and execution of $\mathbb{C}$Duce. Indeed, patterns are compiled into non-uniform tree automata [19] which fully exploit type information. In order to demonstrate the power of pure patterns, we provided an automatic translation from the former into the latter. Such a translation not only shows that projections are useless in the context of $\mathbb{C}$Duce patterns but also gives a formal way of unnesting queries. Then we investigated further optimisations: the well-known database logical optimisations. We therefore adapted such optimisations to the context of $\mathbb{C}$QL, providing a first step toward devising a general query optimiser in such a context.

In order to validate our approach we performed performance measurements. The purposes of such measurements were twofold: first, we wanted to demonstrate the efficiency of pattern-matching based query languages by comparing $\mathbb{C}$QL with efficiency-oriented implementations of XQuery and, second, we wanted to check the relevance of classical database logical optimisation techniques in a pattern-based framework. In both cases, the obtained results were encouraging.

We are currently working on the following topics: (i) develop further (classical) query optimisation techniques such as joins re-ordering, (ii) extend the pattern algebra in order to partially account for descendants axes (at the expense of loosing type precision but still reaching a typed construction) (iii) formally study the expressive power of $\mathbb{C}$QL and finally, in a longer-term, (iv) coupling $\mathbb{C}$QL with a persistent store and study physical optimisation techniques.

Acknowledgements. The authors want to thank Massimo Marchiori for the pointer to Qexo, Xavier Franc for the details on Qizx implementation, and Alain Frisch for the comments on this paper and his help with the implementation of $\mathbb{C}$Duce. Thanks to Dario Colazzo for his help on

[8] Once more the test for Qizx was performed with the dynamic generation of hash tables disabled.

XQuery type system. Very special thanks to Ioana Manolescu for her invaluable suggestions and careful reading of a preliminary version of this paper.

References

1. Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web : from Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers, 2000.
2. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. Wiener. The Lorel query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, 1997.
3. Bell-labs. *Galax*. `http://db.bell-labs.com/galax/`.
4. V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-friendly general purpose language. In *ICFP '03, 8th ACM Int. Conf. on Functional Programming*, pages 51–63, 2003.
5. V. Benzaken, G. Castagna, and C. Miachon. CQL: a pattern-based query language for XML. Complete version. Available at `http://www.cduce.org/papers`, 2005.
6. N. Bidoit and M. Ykhlef. Fixpoint calculus for querying semistructured data. In *Int. Workshop on World Wide Web and Databases (WebDB)*, 1998.
7. P. Bothner. Qexo - the GNU Kawa implementation of XQuery. Available at http://www.gnu.org/software/qexo/.
8. P. Bothner. Compiling XQuery to java bytecodes. In *Proceedings of the First Int. Workshop on XQuery Implementation, Experience and Perspectives <XIME-P/>*, pages 31–37, 2004.
9. Don Chamberlin, Peter Fankhauser, Daniela Florescu, Massimo Marchiori, and Jonathan Robie. XML Query Use Cases. T.-R. 20030822, World Wide Web Consortium, 2003.
10. Don Chamberlin, Peter Fankhauser, Massimo Marchiori, and Jonathan Robie. XML query (XQuery) requirements. Technical Report 20030627, World Wide Web Consortium, 2003.
11. D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML query language for heterogeneous data sources. In *WebDB 2000 (selected papers)*, volume 1997 of LNCS, pages 1–25, 2001.
12. Z. Chen, H. V. Jagadish, L. Lakshmanam, and S Paparizos. From tree patterns to generalised tree paterns: On efficient evaluation of xquery. In *VLDB'03*, pages 237–248, 2003.
13. J. Clark and S. DeRose. *XML Path Language (XPath)*. W3C Recommendation, `http://www.w3.org/TR/xpath/`, November 1999.
14. G. Conforti, G. Ghelli, A. Albano, D. Colazzo, P. Manghi, and C. Sartiani. "The Query Language TQL". In *In 5th Int. Workshop on the Web and Databases (WebDB)*, 2002.
15. World Wide Web Consortium. XQuery: the W3C query language for XML – W3C working draft, 2001.
16. A. Deutsch, M. F. Fernandez, D. Florescu, A. Y. Levy, and D. Suciu. "XML-QL: A Query Language for XML". In *WWW The Query Language Workshop (QL)*, 1998.
17. M. Fernández, J. Siméon, and P. Wadler. An algebra for XML query. In *Foundations of Software Technology and Theoretical Computer Science*, number 1974 in LNCS, 2000.
18. X. Franc. Qizx/open. http://www.xfra.net/qizxopen.
19. A. Frisch. Regular tree language recognition with static information. In *Proc. of the 3rd IFIP Conference on Theoretical Computer Science (TCS)*, Toulouse, Kluwer, 2004.
20. Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic Subtyping. In *Proceedings, Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE Computer Society Press, 2002.
21. H. Hosoya and B. Pierce. XDuce: A typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.
22. Amélie Marian and Jérôme Siméon. Projecting XML elements. In *Int. Conference on Very Large Databases VLDB'03*, pages 213–224, 2003.
23. A. J. Robie, J. Lapp, and D. Schach. "XML Query Language (XQL). In *WWW The Query Language Workshop (QL)*, Cambridge, MA, , 1998.

24. Albrecht Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. Xmark: A benchmark for xml data management. In *Proceedings of the Int'l. Conference on Very Large Database Management (VLDB)*, pages 974–985, 2002.