Pathfinder: Compiling XQuery for Execution on the Monet Database Engine

Jens Teubner

University of Konstanz
Dept. of Computer & Information Science
Box D 188, 78457 Konstanz, Germany
teubner@inf.uni-konstanz.de

Abstract

The W3 Consortium is currently developing the XQuery specification to query XML data. It is still unclear, however, how these data can be stored and retrieved efficiently. We propose an XML storage and query execution system that is based on the main memory database system Monet. This paper describes a technique to compile an entire XQuery expression into a program that can then be executed on the Monet system.

1 Introduction

XQuery is arising as the new "intergalactic query language" to query XML data. The strongly typed language operates on XML's data model, the *tree*. But although a lot of theoretical work has already been done in the area, it is still largely unclear how huge amounts of data can be stored and retrieved efficiently.

Main memory-oriented database systems like Monet [1] are a promising platform for XML storage. Monet comes with a powerful interface language (MIL) that does not only provide a series of algebra-like operators, but also control structures and variables. Monet can be extended by user-defined types and functions.

In this article, we propose a technique that compiles any XQuery expression into a MIL program that can be executed efficiently on a Monet database system.

The starting point for our mapping scheme will be an XQuery expression that is already converted to a normalized form like the "Core" language proposed by the W3C [5] and statically type-checked. We will briefly recapitulate the W3C proposal and point out its relevant properties in Section 2, as well as its consequences for a mapping to an execution system. As we chose the Monet database system as our execution system, we will give a short Monet overview in Section 3 before we describe our mapping approach in Section 4 and clarify it with some examples in Section 5. The results of our mapping will be reviewed and summarized in Section 6.

2 The XQuery Core Language

The World Wide Web Consortium specifies the formal semantics of XQuery using a subset of XQuery called "XQuery Core". This variant is free of syntactic sugar and — most importantly for us — it is fully statically typed.

The XQuery Formal Semantics proposal specifies the type `xs:anyItem` as the main building block of the XQuery type system. An *item* can either be a value of an XML Schema atomic type like `xs:integer` or `xs:string`, or a node (element, attribute, etc.). XQuery types are then described using regular expressions of these basic blocks, specifying structural constraints on nodes and the shape of their subtrees.

XQuery expressions evaluate to ordered sequences of items. The items within one sequence do not have to be of the same type, but can be arbitrarily combined into heterogeneous, although flat sequences. A sequence like `(42, "abc", <a/>)` is a valid XQuery expression that evaluates to a sequence containing an integer, a string and an element node.

As *any* expression evaluates to a sequence of items in the XQuery data model, a single item is defined to be identical to a sequence containing exactly this one item. The sequence is then called a *singleton sequence*.

In particular the last two features demand considerable flexibility from an XQuery execution system. It must be capable of dealing with a wide range of possible types, although in most situations the actual types will be simple.

2.1 Item Type Decidability

The XQuery specifications allow different item types to be randomly mixed in XQuery expressions. In most cases, however, sub-expressions will evaluate to values or sequences of exactly one item type. XPath expressions, as an example, will always return nodes

only, while string operations will always return strings. If not explicitly requested by the user (which rarely makes sense), we will not deal with heterogeneous sequences.

2.2 Sequence or Single Item?

Following the XQuery semantics, a single item is identical to a sequence containing exactly this item. Although this suggests an implementation that stores everything as a sequence, such an approach will have significant performance drawbacks.

With a closer look at the static type of XQuery expressions, we can decide very precisely, which expressions are likely to return a result sequence, and which ones will definitely return only single values. Examples are XPath expressions that usually return node sequences, while `for`-bound variables or the results of arithmetic operations will always be single items.

An expression's item type and an estimation of the result size are determined during static typing. In the W3C Formal Semantics draft the respective information is gathered by the functions `prime()` and `quantifier()`. Our compiler uses an equivalent subtype check to estimate the result size. Common optimization techniques will further restrict an expression's possible type.

It seems indeed feasible to use a monomorphic solution for query execution that uses the actual primitive data types wherever possible instead of a fully polymorphic solution that uses a "boxed" representation of all data. The latter solution would pack all data items in a uniform data structure ("box") and use pointers to these boxes instead of actual values for parameter passing [10]. Although this approach reflected XQuery semantics and made a mapping very uniform, its performance drawbacks due to the necessary boxing/unboxing operations are obvious.

Note that we will still need a boxed data representation in some cases, as XQuery is inherently polymorphic. But in real-world applications these cases are rare, considering the above observations.

3 The Monet Database System

Main memory-oriented database systems have recently shown to outperform traditional databases due to their cache and CPU optimized execution. Besides that, the Monet database kernel [1] has several aspects that make it a promising choice for XQuery execution.

3.1 Binary Table Data Model

Monet is a relational system that operates with binary tables only ("BAT"s, Binary Association Tables). This fits well with the XML indexing scheme that we use to encode our XML data. This indexing scheme, the XPath accelerator [6], uses two integer values to encode the XML tree structure.

In a nutshell we enumerate the nodes in the XML tree in a pre- and a postorder tree-walk. The two integer values (*pre-* and *postorder rank*) that we store for each node contain the full structural information of the XML tree.

3.2 Algebra-like Interface Language.

The Monet database system can be accessed via the Monet Interface Language MIL. This language provides algebra-like operations as well as variables or control structures which allows to compile an entire XQuery expression into a single MIL program that will then evaluate the whole query on the database system.

3.3 Extensibility

Monet can easily be extended with user-defined data types and operators. We have developed the "staircase join" operator that has been shown to speedup XQuery evaluation [7].

Staircase join uses knowledge about properties of our relational tables that originate from the tree-structure of the underlying data. Due to this origin, the values in the *pre/post* table are not randomly distributed, but exhibit characteristics that the staircase join algorithm takes advantage of. Off-the-shelf relational databases are not aware of these characteristics and have to rely on their own statistics to optimize queries. Staircase join can be implemented in existing relational databases, bringing awareness of tree-structures to the database system.

Staircase join has proven to be particularly efficient to evaluate the XPath axes `ancestor` and `descendant`, even for huge amounts of data. Because of their recursive definition, these axes have traditionally been hard to implement efficiently.

The Pathfinder compiler will translate an XQuery expression into a MIL program that can then be run in the Monet database system and evaluate the entire expression. The generated program will only require a small extension module to the Monet database kernel, providing a minimum of additional data types and the staircase join operator.

4 The Pathfinder Mapping Scheme

4.1 Item Representation in MIL

Following the considerations in Section 2 we use primitive data types wherever possible.

For the simple data types, the mapping from XQuery to Monet types is straightforward. The types are either readily available in Monet or provided by an extension module as user-defined types. The important `node` type uses integer values as its implementation type. This integer is the preorder rank in

the XPath accelerator mapping scheme that we use to store our XML documents.

Monet's function overloading mechanisms ensure that correct semantics are used in function calls or type-casts. With the help of MIL's `type` operator, we may also determine the actual type of any data object at runtime.

Additionally, we need a boxed representation for items where we cannot infer the static type precisely enough. We define another Monet type, `item`, that we use whenever the static type is a choice of at least two of the types in Table 1. Type `item` is implemented by an integer and is basically a foreign key to five global BATs that store the actual values. The five BATs are listed in Figure 1. Each `item` value is unique over all these tables. Our implementation will use a bit encoding scheme of the integers to determine the actual type of any `item` value easily.

4.2 Sequence Representation

We store sequences rather straightforward as BATs. Note that in contrast to SQL database systems BATs are *ordered* sequences of value pairs ("BUNs", Binary UNits), which matches the XQuery semantics of sequences. As we do not need a second column, we use BATs with `nil` heads and store our data in the tail. (The two BAT columns are referenced as *head* and *tail* in Monet.) By declaring the head type as `void` (*virtual oid*), these `nil` values don't have to be stored in the database, reducing memory consumption to a minimum.

As we want to keep our data representation as simple as possible for performance reasons, we use this sequence representation only for expressions that could possibly result in a sequence of length longer than one. Static typing gives us an estimation of the result size that can be described as one of five quantifiers: `0`, `1`, `?`, `+`, or `*` for 'empty', 'exactly one item', 'zero or one items', 'one or more items', and 'zero or more items'.

Our compiler chooses a BAT representation only for the latter two cases, `+` and `*`. If static typing reveals a sequence length of at most 1 (quantifiers `1` and `?`), we use primitive Monet values. The decision can be implemented as a single subtype check. If the expression's static type is a subtype of `xs:anyItem?`, we use the primitive representation, otherwise we use a BAT.

XQuery type	MIL type
`xs:boolean`	`bit`
`xs:integer`	`int`
`xs:double`	`dbl`
`xs:string`	`str`
`node`	`node`$^{\Diamond}$

Table 1: Implementation types for simple XQuery types (and their subtypes). $^{\Diamond}$The `node` type is a user-defined extension to Monet's built-in types.

The quantifier `0` is the statically typed empty sequence. Such expressions will be eliminated by a core optimization phase in most situations. The few remaining cases (e. g. the return parts of `if-then-else`s) will be treated explicitly during compilation.

Note that we end up in having two representations for the empty sequence: If static typing lead to the BAT representation, we use the empty BAT. If we have at most one item and chose a representation with primitive types, we use Monet's special symbol `nil` that can be described as the equivalent to SQL's `NULL`.

4.3 XQuery Mapping

During query optimization, our compiler brings the input query into a normalized form with two characteristics that are important for the mapping to MIL:

1. The query is in SSA form (static single assignment form [4]). In a nutshell this means that all variables are assigned exactly once and never modified afterwards. This is typical for functional programming languages.

2. Operands of almost all operations must be atomic, that is, they must be either simple constants or variables. General expressions are only allowed in few places, like the right hand side of variable bindings. This feature is a prerequisite for optimization techniques like common subexpression elimination (CSE, [2]).

This form does not only help a preceding optimization phase, but also matches the Monet execution strategy very well. Monet materializes all intermediate results in main memory, which is made explicit by our normalization form. The mapping to MIL is thus assignment-based. For a uniform processing, we rewrite the query e to

$$result \leftarrow e;$$
$$\text{print}(result);$$

and describe mapping rules only for assignments $v \leftarrow e_i$. In some cases it might be necessary to create new variables during the mapping.

4.4 Dynamic Typing

XQuery is a strongly-typed language. But although a large amount of type information can be inferred during a static typing phase at compile time, the precise type information of some XQuery expressions is only known at query evaluation time. In the XQuery surface language, the operators `instance of` and `typeswitch` (the latter chooses one of several expressions based on the type of an input value) are available to query type information at evaluation time.

The W3C Formal Semantics specification describes the semantics of the above operators with *structural*

item	bit
#8	true
#16	false

bit_items

item	int
#9	42
#25	17

int_items

item	dbl
#18	2.74
#26	1.42

dbl_items

item	str
#11	"foo"
#27	"bar"

str_items

item	node
#20	%67
#28	%98

node_items

Figure 1: Five global BATs, one for each base type, store the actual values that are referenced by items of type `item`. Each `item` value must be unique over all these global BATs. We mark `item` values with a leading hash mark (`#`) and `node` values with a leading percent sign (`%`) to make them distinct from integers.

subtyping. Types in the XQuery type system are generally described by regular expressions. A subtype test is then an inclusion test of the state machines corresponding to the regular expressions [3]. Although other approaches have been published [9, 11], the test is in general rather expensive.

Fortunately, the number of possible types to check against is limited by XQuery's syntax constraints. The W3C specifications only allow the basic blocks of the XQuery type system: named atomic types and restrictions on a node's kind (element, attribute,...) or tag name, but no regular expressions thereof. Further type restrictions are only allowed in conjunction with *validation* (see below).

The type check thus boils down to a simple test on the Monet implementation type we use. If the data object in question has the `item` implementation type, the unboxed type is determined using the above mentioned bit encoding (see Section 4.1). Node kind and tag name can be tested with lookups in the corresponding BATs.

For further type tests, the expression to be tested must have been explicitly *validated* by the user. The `validate` operator invokes this process that annotates an expression with type information in form of a named type. The subtype condition can then be tested with the rather simple *named typing*: Only the type names have to be compared using a hierarchy of type names.

The expensive part of the subtyping tests is effectively left to the `validate` operator. Pathfinder will thus do this well-defined task within its runtime extension module, where it can be tuned to operate efficiently.

5 Example Mapping Rules

The following examples will show that compiling XQuery expressions to MIL is indeed feasible. The compilation strategy keeps data types as simple as possible which will reduce execution overhead compared to a fully boxed data representation. The mapping is described for three basic building blocks of the XQuery language: for the sequence constructor `(·,·)`, for `for`-iterations and for XPath steps.

The following examples use a_i for atoms, i. e. for simple constants or variables. They are mapped to constants/variables in MIL. In our normalized query arbitrary expressions, denoted by e_i, can only occur in well defined situations.

5.1 Sequence Construction

$[\![\ v\ \leftarrow\ (\ a_1, a_2\ )\ ]\!]$

==

```
v := new (void, item);
v.insert ( a1 );
v.insert ( [item]( a2 ));
```

Depending on the type information, the correct return type must be created and the input operands need to be casted if necessary. In this example we assume operand a_1 to already be a sequence that is implemented as a BAT of `item`s. a_2 is, say, a sequence of integers that needs to be cast to `item`. The above MIL notation maps the cast operator `item` into the BAT a_2.

5.2 `for`-Iterations

$[\![\ v\ \leftarrow$ `for $x in` a_1 `return` $e_2\ ]\!]$

==

```
v := new (void, int);
a1@batloop {
  x := $t;
  [[ w ← e2; ]]
  v.insert (w);
}
```

The MIL `batloop` command iterates over all tuples in the BAT a_1 and executes the body for each tuple.[1] Within this body, the current head and tail values are available as the special variables `$h` and `$t` (our mapping uses the tail column to store sequence items). The example shows how our assignment-based mapping scheme nicely fits the mapping to MIL.

5.3 Path Expressions

With the staircase join implemented in a Monet extension module, XPath evaluation will be a simple function call for the corresponding step.

$[\![\ v\ \leftarrow\ a_1$`/descendant::node()` $]\!]$

==

```
v := staircasejoin_desc (doc, a1);
```

[1] A preceding optimization phase ensures that a_1 is a BAT variable.

6 Summary and Outlook

We propose a technique that will allow for efficient evaluation of XQuery expressions using existing relational database technology. Our mapping scheme compiles queries into single MIL programs, the interface language for the main memory database system Monet.

The generated program mostly operates on primitive data types, minimizing space and processing overhead during execution. Our mapping technique is assignment-based and will generate a series of rather simple MIL operations that are each assigned to temporary variables. This code fits very well with Monet's processing paradigms and will lead to efficient XQuery execution.

This MIL generation phase is part of the ongoing 'Pathfinder' project at the University of Konstanz. We are currently working on the Pathfinder compiler and aim at a full implementation of the W3C XQuery specifications. In collaboration with the Monet group at CWI we will provide a highly efficient XML database even for huge amounts of data.

Initial experiments that were done with handwritten MIL programs show that our approach is in fact promising. With documents generated by the XMark benchmark suite [12], even in the order of gigabytes, the Monet system has proven to be superior over an implementation on top of a commercial relational database system, but also over a "native" XML database system. [6, 8]

References

[1] Peter Alexander Boncz. *Monet — A Next-Generation DBMS Kernel For Query-Intensive Applications*. PhD thesis, University of Amsterdam, The Netherlands, May 2002.

[2] Olaf Chitil. Common Subexpression Elimination in a Lazy Functional Language. In Chris Clack, Tony Davie, and Kevin Hammond, editors, *Proceedings of the 9th International Workshop on Implementation of Functional Languages*, September 1997.

[3] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree Automata Techniques and Applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 1997. Release October 1st, 2002.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[5] Denise Draper, Peter Fankhauser, Mary F. Fernandez, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. Technical Report W3C Working Draft, World Wide Web Consortium, November 2002.

[6] Torsten Grust. Accelerating XPath Location Steps. In *Proc. of the 21st Int'l ACM SIGMOD Conference on Management of Data*, pages 109–120, Madison, Wisconsin, USA, June 2002. ACM Press.

[7] Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. In *Proc. of the 29th Int'l Conference on Very Large Data Bases (VLDB)*, September 2003.

[8] Torsten Grust, Maurice van Keulen, and Jens Teubner. On Accelerating XPath Evaluation in Any RDBMS. *ACM Transactions on Database Systems*, 2003, under revision.

[9] Haruo Hosoya. *Regular Expression Types for XML*. PhD thesis, The University of Tokyo, December 2000.

[10] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, New York, 1994.

[11] Martin Kempa and Volker Linnemann. Type Checking in XOBE. In *Datenbanksysteme für Business, Technologie und Web (BTW), 10. GI-Fachtagung*, 2003.

[12] A. Schmidt, F. Waas, M. Kersten, D. Florescu, I. Manolescu, M. Carey, and R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, April 2001.