November 2000

Kweelt, the Making-of: Mistakes Made and Lessons Learned

Arnaud Sahuguet
University of Pennsylvania

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-00-23.

Kweelt, the Making-of: Mistakes Made and Lessons Learned

Abstract

In this paper we report our experience in building Kweelt, an open source Java framework for querying XML based on the recent Quilt proposal.

Kweelt is intended to provide a reference implementation for the Quilt language but also to offer a framework for all kinds of experiments related to XML including storage, optimization, query language features, etc. And we report in this paper on the differences entailed by the use of two different storage managers, based respectively on character files and relational databases.

An important design decision was to do a "direct" implementation of Quilt. Instead of relying on preconceptions (and misconceptions!) inherited from our database query processing background, we wanted this reference implementation to expose exactly what is easy and what is hard both in terms of expressiveness and of efficiency. The process has lead naturally to what may in hindsight be called mistakes, and to formulate lessons that will hopefully be used in future implementations to mix-and-match pieces of existing technology in databases and programming languages for optimal results.

Comments

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-00-23.

KWEELT, the Making-of: Mistakes Made and Lessons Learned

Arnaud Sahuguet
Department of Computer and Information Science
University of Pennsylvania
sahuguet@saul.cis.upenn.edu

November 2000

Abstract

In this paper we report our experience in building Kweelt, an open source Java framework for querying XML based on the recent Quilt proposal.

Kweelt is intended to provide a reference implementation for the Quilt language but also to offer a framework for all kinds of experiments related to XML including storage, optimization, query language features, etc. And we report in this paper on the differences entailed by the use of two different storage managers, based respectively on character files and relational databases.

An important design decision was to do a "direct" implementation of Quilt. Instead of relying on preconceptions (and misconceptions!) inherited from our database query processing background, we wanted this reference implementation to expose exactly what is easy and what is hard both in terms of expressiveness and of efficiency. The process has lead naturally to what may in hindsight be called mistakes, and to formulate lessons that will hopefully be used in future implementations to mix-and-match pieces of existing technology in databases and programming languages for optimal results.

Technical Report

History:

- revision (14-Mar-01): fixed error in XPath predicate examples (page 3).

1 Introduction

In this paper we report our experience in building Kweelt, an open source Java framework for querying XML based on the recent Quilt proposal [5].

Our initial motivation was born out of frustration while searching for a query language to do advanced – and not so advanced – XML data integration. Past proposals [22, 29, 7, 6] lack features like join between multiple documents, order preservation, free navigation. Most importantly, available implementations are provided as monolithic architectures that cannot be extended. In particular, when the query engine and the storage manager form one indivisible module, it is difficult to experiment with various backends, for example to support very large documents.

When the Quilt language was proposed, this looked like a good opportunity to resolve this frustration. Instead of just *yet another implementation of a query language for XML*, we chose to build a modular and extensible framework called Kweelt to query XML, where users could replace and extend components as they see fit. The intended use of Kweelt was to offer a reference implementation for the Quilt language but also to offer a framework for all kinds of experiments related to XML including storage, optimization, query language features, etc.

We have conducted one of these experiments already. We report in this paper on the differences entailed by the use of two different storage managers, based respectively on character files and relational databases.

An important decision was to do a "direct" implementation, one based straightforwardly on the semantics of Quilt as described in the original proposal. The alternative would have been to rely on the preconceptions (and misconceptions!) inherited from our database query processing background. Instead, we wanted this reference implementation to expose exactly what is easy and what is hard both in terms of expressiveness and of efficiency. We intended this process to lead naturally to what may in hindsight be called mistakes, and to formulate lessons that can be used in future implementations to mix and match pieces of existing technology in databases and programming languages for optimal results.

As a side-effect, the Kweelt experience turned out also to be instructive in terms of comparing a paper proposal (Quilt) against a running "flesh and bones" implementation (Kweelt). As of this writing and to the best of our knowledge, Kweelt is still the only full implementation based on Quilt (the authors of Quilt have implemented a parser).

The rest of the paper is organized as follows. In Section 2, we give an overview of both XPath and the Quilt query language, which relies heavily on the former. Section 3 exposes the overall design of the Kweelt framework including its philosophy, its architecture and evaluation strategy. We then describe in details two XML storage modules (backends), one based on DOM and one based on a relational encoding of XML. Section 6 presents some preliminary benchmarks that compare the

backends with respect to our evaluation strategy. In Section 7, we detail the lessons we learned from the Kweelt experience. Finally we present some related and future work before we offer our conclusions.

2 Hands on XML!

Since Quilt relies on XPath, we first describe XPath before we present the language itself.

2.1 Finding your Way in a Document using XPath

"*The primary purpose of XPath is to address parts of an XML document. In support of this primary purpose, it also provides basic facilities for manipulation of strings, numbers and booleans.*" [28]. In this section, we will focus on the navigational aspects of the language.

An XPath navigation expression (aka location path) is a sequence of navigation steps (aka location steps) separated by the `/` symbol. A navigation step is defined by an axis specifier, a node-test and some optional predicates (see Figure 1).

LocationPath	::=	*LocationPath* ["/" *LocationStep*]
LocationStep	::=	*AxisSpecifier* "::" *NodeTest* ("[" *Predicate* "]")*
NodeTest	::=	*name* | "*" | "node()" | "text()" | "comment()" | "processing-instruction()"

Figure 1: XPath syntax for navigation expressions

The axis defines how to navigate the document tree. The nodetest is a filter to keep or prune nodes based on their nature (text, element, attribute) or label (tag or attribute name). Predicates are boolean XPath expressions that can be further applied to nodes. A navigation expression returns a list of nodes. The order is defined by the document order for forward axis or the reverse document order for backward axis. The possible values for *AxisSpecifier* are presented in Figure 3–left. Axis form a partition of the document, from the point of view of the current node (see Figure 3–right).

The XPath syntax is verbose and various syntactic abbreviations are available (see [28]).

Abbreviation	Expansion	Abbreviation	Expansion
(nothing)	`child::`	`.`	`self::node()`
`@`	`attribute::`	`..`	`parent::node`
`//`	`/descendant-or-self::node()/`	`/`	the node tree root

Figure 2: Abbreviated Syntax

Examples of XPath expressions and their abbreviated syntax are given below:

```
(1 ) /child::book[attribute::year=2001]/child::title
```

```
(1') /book[@year=2001]/title
(2 ) /descendant::author/parent::node()[attribute::year>2000]
(2') /descendant::author/..[@year>2000]
```

Forward Axis
CHILD
DESCENDANT
DESCENDANT-OR-SELF
FOLLOWING
FOLLOWING-SIBLING
Backward Axis
PARENT
ANCESTOR
ANCESTOR-OR-SELF
PRECEDING
PRECEDING-SIBLING
Other axis
SELF
ATTRIBUTE
NAMESPACE

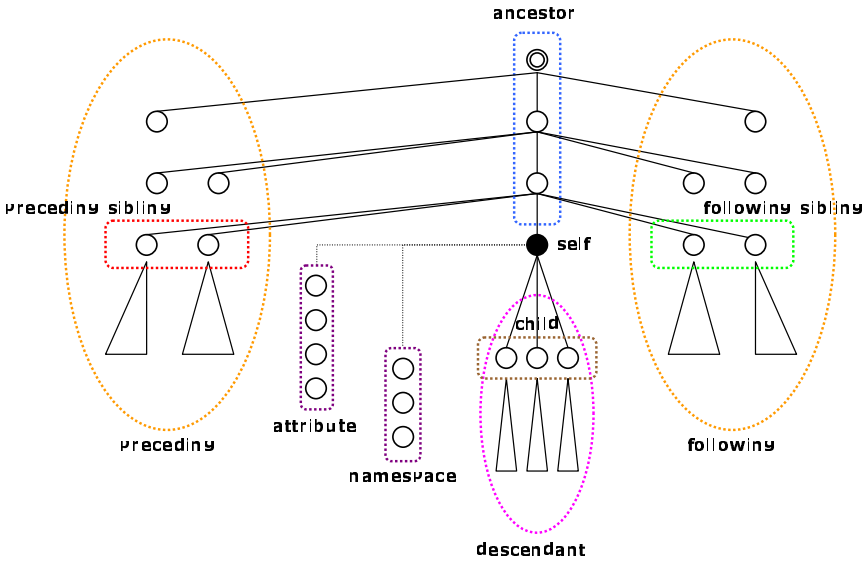


Figure 3: XPath navigation axis

An XPath navigation is always evaluated within a *context* that carries some information about the previous navigation. More concretely a context consists of the current node, its position in the top-level nodelist and the size of the nodelist. The nodelist corresponds to the result of the evaluation of a previous XPath expression. In the example above, `attribute::year="2001"` is evaluated in the context of the current node, i.e. the one returned by `/child::book`.

In XPath, the equality between two nodes is defined by the notion of *string value*, i.e. the concatenation of all the text nodes in the sub-tree.

Predicates if not handled carefully can lead to unexpected results. For instance we have `/a[b][c]` $\equiv$ `/a[c][b]` $\equiv$ `/a[b and c]` but `/a[b][2]` $\neq$ `/a[2][b]`. In the second one, the left hand side expression means the "from the current node, the second `a` child with a `b` child", while the right hand side means "from the current node, the second `a` child if it has a `b` child". Predicates are not commutative.

The abbreviated syntax can also lead to mistakes. `//book[5]` does not return the fifth book in the document. It gets expanded into `/descendant-or-self::node/child::book[5]` which returns the fifth `book` of every descendant from the root. The correct expression is `(//book)[5]`.

XPath is a very permissive language where operations are defined on almost any type with some implicit coercion rules: nodesets have a boolean value (empty equals false); equality and inequality

between nodesets are defined in an existential way, etc. A formal semantics for XPath can be found in [32]. It is important to keep in mind that XPath is a navigation language and its purpose is to identify a subset of the nodes inside a given document.

2.2 Querying Documents using Quilt

Quilt is a new proposal that – as its name implies – borrows the best features from several languages "*in a way in which Quilt queries can combine information from diverse data sources into a query result with a new structure of its own*"[5]. The proposal addresses all the query requirements published by W3C [30]. From a database point of view, the core of Quilt really is comprehensions [4] (variable bindings, iterators) on top of XPath (object model, navigation), within a modernized SQL-ish `select-from-where`-like construct . For a full description of the proposal, see [5]. In the rest of the section we will address the salient and most interesting features of the language, from our implementation point of view.

Inside a Quilt query, the entry point to an XML tree is `document`. `document("items.xml")` will return the root node of the corresponding document. From then on, the document can be navigated using XPath. As an example, we present below a solution to *Query 2* from the W3C use-cases [30] (section R) with the simplified corresponding DTDs (Figure 4). The use-case illustrates the use of Quilt to query relational data and the example is based on a fictional auction scenario with users, items and bids.

items.dtd

```
<!ELEMENT items              (item_tuple*)>
<!ELEMENT item_tuple (itemno,description)>
<!ELEMENT itemno                (#PCDATA)>
<!ELEMENT description           (#PCDATA)>
```

bids.dtd

```
<!ELEMENT bids               (bid_tuple*)>
<!ELEMENT bid_tuple (userid,itemno, bid)>
<!ELEMENT userid                (#PCDATA)>
<!ELEMENT itemno                (#PCDATA)>
<!ELEMENT bid                   (#PCDATA)>
```

Figure 4: Simplified DTDs for the W3C use case

The purpose of the query (Figure 5) is to combine information from both `items.xml` and `bids.xml` files. The join is performed on `itemno` (line 6). The query should be read as follows. Variable `$i` binds to `item_tuple` nodes from the `item.xml` document (line 5). For each instance, variable `$b` binds to a list of `bid_tuple` from `bids.xml` such that the `itemno` value is the same as the one of the current instance (line 6). For each instance of `$i` and for each instance of `$b`, a tuple is produced. Tuples are then pruned by the `WHERE` clause (line 7). For each tuple, a `<item_tuple>` element is created (line 9) with 3 children: the `itemno` (line 10) and `description` (line 11) children of the node bound to `$i` and a new element `<high_bid>`. The content of the latter is either the empty string if the value of `$b` for this instance is the empty node list or the value of the maximum (line 12). The produced `<item_tuple>` elements are sorted by `itemno` (line 13) and surrounded by an

outer `<result>` (lines 3 and 15).

Query 2, from W3C use cases (R)

```
1  -- For all bicycles, list the item number, description and
2  highest bid (if any), ordered by item number. --
3  <result>
4     (
5     FOR $i IN document("items.xml")//item_tuple
6     LET $b := document("bids.xml")//bid_tuple[itemno = $i/itemno]
7     WHERE contains($i/description, "Bicycle")
8     RETURN
9        <item_tuple>
10          $i/itemno ,
11          $i/description ,
12          IF ($b) THEN <high_bid> max($b/bid) </high_bid> ELSE ""
13       </item_tuple> SORTBY (itemno)
14    )
15 </result>
```

Figure 5: Query 2

The semantics of the language is similar to XML-QL [7]. `FOR/LET` clauses create an ordered list of tuples (one field for each variable); the `WHERE` clause prunes it; the `RETURN` clause transforms it into XML (the clause can be seen as a template).

We want to make the following remarks. Because Quilt uses XPath and because XPath is a powerful navigation language, queries can be (re)written in many different but equivalent ways. For instance in many cases[1], XPath predicates can be replaced by Quilt `WHERE` clauses and vice-versa. If we look at Quilt in a procedural (rather than declarative) way, this can make a big difference in terms of performance.

Quilt queries preserve order. It is important to bear in mind that order is not always well defined: how should we order nodes coming from two different documents?

Quilt offers a `DISTINCT` keyword to remove duplicates. This raises two issues. First, the notion of node equality has to be defined. Apparently the one from XPath has been chosen where nodes are compared against their *string value*. Second, duplicate elimination does not preserve order.

As a convenient shorthand, Quilt offers the arrow operator (`->`) to dereference `IDREF` (this is like the `id()` operator in XPath). In both cases, the operator can only be evaluated if some structural information (DTD, XML-Schemas, etc.) about the XML document is available. For instance, in the previous example, if we assume that `userid` is not an sub-element of `bid_tuple` but an `IDREF` attribute, the expression `bid_tuple/@userid->` has no meaning because the query evaluator has

[1] Keep in mind that XPath predicates are subtle.

no clue where to look for. With some structural information, the evaluator will look for *the element* that has an attribute of type `ID` with a value equals to the value of attribute `userid`.

3 Kweelt

3.1 Design Philosophy

The goal of Kweelt is to provide a modular and extensible framework to query XML, where users can replace and extend components as they see fit.

Separation between query engine and storage module
The first critical decision was to isolate completely the query engine from the storage module. The former is responsible for parsing, massaging and evaluating queries; the latter is in charge of manipulating XML. The bridging between both is accomplished via various APIs.

This design permits to use multiple storage modules, even within the same query. We think this is a realistic assumption because XML documents have all sort of shapes, sizes, durabilities and there is no one-size-fits-all way to manipulate XML. Moreover people who already have their favorite tools are more likely to adopt a new technology if it is non intrusive and compatible with their *legacy* tools.

This also permits to make some improvements on one side with no interference on the other. New storage modules can be implemented; the query engine can be improved with new language features or new optimizations; and everything will still work seamlessly, provided that components comply with the API.

Generic XML storage
In Kweelt we advocate a generic query engine with a generic storage module. By generic, we mean that the shape of the XML is not known in advanced and therefore every document has to be processed as a tree. This is to be contrasted with approaches such as STORED [8], SilkRoute [13] or DB2 XML Extender[2] that define a document specific mapping from XML to relational. With the growing number of lightweight free database managements systems already available for individuals and companies, we think it makes perfect sense to leverage on them for generic XML processing. See [25] and [27] for a discussion and comparison of XML storage strategies.

The query engine as a naive interpretor for Quilt
The second critical decision was to implement the query engine in a naive way, by looking at Quilt in a procedural rather than declarative way: the structure of the query will dictate its evaluation. The query is parsed into a an abstract syntax tree and interpreted as is rather than compiled.

[2] `http://www-4.ibm.com/software/data/db2/extenders/xmlext`

The main rationale was simplicity. Another one is that it was not – and it is still not – clear what else could be done. For SQL, a query would be mapped into relational algebra, but for XML queries, this is still on-going research. Both were reinforced by our decision to support the full Quilt proposal, including recursive functions and operators like `SHALLOW` and `FILTER` that make Quilt look like a general purpose programming language.

Extensible and modular design
The framework has been engineered with extensibility in mind, at the expense of performance. The abstract syntax makes it easy to add new constructs at the level of the parser or via user-defined functions. The language can also be enriched directly via Java code. The code heavily uses object-orientation and inheritance to make the development of extensions and new backends easy.

Optimization focused on XPath rather than Quilt
In the absence of an algebra for the query language, one should expect the optimization to be ad-hoc. Moreover our assumption to have multiple backends involved in the same query would require to handle cross-backend optimization where for instance a DOM implementation would work in concert with a relational database. The architecture of Kweelt does not forbid this situation but rather focuses on XPath optimization, i.e. trying to evaluate binding expressions (`LET` and `FOR`) as efficiently as possible). Our Node Factory API corresponds exactly to XPath navigation primitives.

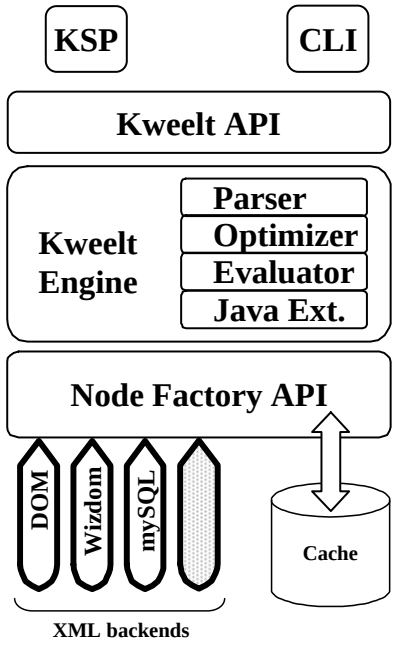
Delegation
Kweelt tries to delegate as much work as it can to the backend, but at the same time provides a callback mechanism in case the backend cannot perform a given task. This way, the evaluation engine can work seamlessly with very powerful or very limited backends. Ideally we would like Kweelt to run on embedded devices (like Palm computers) with limited resources (memory and CPU), all the work being done by remote backends.

Backends are in charge of the nodes from the XML documents, in terms of navigation, string value computation, nodelist operations and output. Kweelt is responsible for the nodes constructed by the query.

3.2 Architecture

The architecture of the Kweelt framework is presented in Figure 6–left. The core of the system is the Kweelt engine which parses and evaluates Quilt expressions. The optimizer is not implemented yet. The engine can also be enriched by Java extensions. The engine can interact with top-level applications via the Kweelt API and with low-level applications (XML backends) via the Node Factory API.

Top-level Applications The framework comes with two handy applications: a command-line interface (CLI) and the KSP publishing engine (Kweelt Server Pages) that permits to embed



NodeFactory API (simplified)

```
NodeFactory: (backend)
  Node getDocumentRoot(String docName)
Node: (backend)
  NodeList findAxis(int axis)
  NodeList findAxis(int axis, NodeTest nt)
  NodeList findAxis(int axis, NodeTest nt,
                    Predicate[] preds)
NodeList: (backend)
  methods to manipulate node lists
NodeTest: (query engine)
  boolean isTrue(Node n)
Predicate: (query engine)
  boolean isTrue(Node n,
                 int position,
                 int size)
```

Figure 6: Kweelt Architecture and API

queries inside XML documents that are evaluated and transformed by a Web server. KSP is a processor for the Cocoon engine[3].

XML backends When a Quilt expression refers to a document, a call is made to the *Node Factory* to return a *Node*. The engine can then call node methods to navigate the document and grab other nodes. The role of the XML backend is to provide implementations for node functions (XPath navigation mainly) and node list operations. The rest is the responsibility of the Kweelt engine.

3.3 API

The full description of the Kweelt API can be found on the project web site[4]. We give here in Figure 6–right only the details that explain what information is sent to the backends when evaluating pieces of a query.

Navigation inside the document is performed via the `findAxis` methods where `axis` is one the XPath axis (see 2.1). Depending on its capabilities, the XML backend supports one or more of the `findAxis` method natively. `findAxis(axis)` is always supported natively. The other two can be implemented non natively (see 3.4). The XML backend is only concerned with document management (parsing, caching if available), node navigation and node list operations.

[3] `http://xml.apache.org/cocoon`

[4] `http://db.cis.upenn.edu/Kweelt`

The XML backend is not expected to handle predicates[5], which can be arbitrarily complex XPath boolean conditions. In case it cannot, it simply makes a callback to Kweelt via the `isTrue(_,_,_)` method[6], for every node it is processing. Kweelt in turn will probably decompose the predicate and send back some pieces of it to the backend for evaluation (see 3.4). The amount of work the engine can delegate depends on the backend. Kweelt tries to push as much work to the backend, but might end up doing most of the work by itself via call-backs.

3.4 Evaluation strategy

The general process of evaluating a Quilt query has already been addressed in 2.2. We now describe how an XPath expression is evaluated using the API.

The evaluation algorithms for both Kweelt and the backend are presented in pseudo-code in Figure 7. We use `[..]` for lists, `+=` for list concatenation and indentation marks loop boundaries.

Path Evaluation (Kweelt)

```
1  NodeList eval(Node context-node, Step path[])
2    NodeList result = [ context-node ]
3    foreach step in path
4      tmp = []
5      foreach node in result
6        tmp += node.findAxis(step.axis,
7                             step.nodetest,
8                             step.predicates)
9      result = tmp
10   return result
11
12
13
14
```

`findAxis` (Backend)

```
NodeList findAxis(int axis, NodeTest nt,
                  Predicate preds[])
  NodeList tmp = findAxis(axis)
  NodeList result = []
  for node in tmp
    if nt(node).isTrue() result += node
  tmp = result
  result = []
  for p in preds
    for i=1 to size(tmp)
      if p(tmp[i], i, size(tmp)) == true
        then result += tmp[i]
    tmp = result
  return result
```

Figure 7: Evaluation Strategies: Kweelt (left), Backend (right)

Kweelt splits a location path into location steps, which are evaluated in turn from the context node. The granularity of the tasks delegated to the backend is at the level of location steps. Starting from the context node, Kweelt processes the output of the current navigation step (line 6) as the input of the next one (line 9). Nodes are concatenated in order in the final result.

On the backend side, the `findAxis(axis,nodetest,predicates)` calls are processed by default in the following way. The backend first calls the native `findAxis(axis)` method (line 3). It then applies in turn the nodetest (lines 5–7) and the predicates (lines 9–13) to every node from the result

[5] Note that predicates require a context-node (see 2.1).

[6] For `Predicate`.

set. Note that depending on the backend, the calls could also be evaluated natively.

4 DOM backend

This backend is the default backend for the Kweelt framework. It relies on any implementation of the DOM API which provides the following navigation capabilities:

DOM API (main methods)

```
NodeList getChildNodes()                          NamedNodeMap getAttributes()
Node getFirstChild()                              Node getLastChild()
Node getPreviousSibling()                         Node getNextSibling()
Node getParentNode()                              Document getOwnerDocument()
String getNodeName()                              short getNodeType()
NodeList getElementsByTagName(String name)  [...]
```

The backend implementation translates XPath navigation primitives into DOM calls. Some primitives can be directly mapped to the corresponding DOM methods (like the parent axis), but for others there is some extra work. The main difference between DOM and XPath is that one returns a single node while the other returns a list of nodes. Moreover, DOM does not support the notion of nodetest and predicate.
For `ANCESTOR`, `FOLLOWING-SIBLING` and `PRECEDING-SIBLING` we call repeatedly the corresponding DOM methods (`getParentNode`, `getNextSibling`, `getPreviousSibling`) until no more nodes can be reached. For `DESCENDANT` we call `getChildNodes` and apply the same axis on the intermediate result. For `FOLLOWING` and `PRECEDING` we have to remark that: *following nodes* correspond to the descendants of the following siblings of the ancestors of the current node while *preceding nodes* correspond to the descendants of the preceding siblings of the ancestors of the current node. See Figure 3 for the intuition.

A last crucial point concerns order. The issue here is that DOM does not provide a direct method to compare two nodes for their relative position in a document. To compare two nodes n_1 and n_2 using DOM, we need to compute for both the list of ancestor nodes up to their first common ancestor a, which maybe the root. If $ancestor(n_1) \subset ancestor(n_2)$ then n_1 comes first. Otherwise, the path from n_1 (resp. n_2) to a contains a child node of a. The order is defined by the relative index of this two child nodes. The complexity of the algorithm is linear in the depth of the tree and the maximum number of children of a node.
Since the method is called very often, this is expensive. But there is no other way because the DOM interface hides the implementation. This is all the more regrettable that node order is trivial to compute during parsing, by incrementing a counter every time your meet a new node. This is

exactly what is done in the Xerces implementation[7] with a method called `getNodeIndex()` that returns an `integer` value. Order can thus be computed at the cost of one numerical comparison. For any order-aware query, the improvement is far from negligible (see Section 6).

5 Relational backend

There are many ways to encode XML in relations but we already said we chose to go for generic ones. Even then, there are many options. See [14] for an overview of some of them.

For Kweelt, we have implemented the scheme proposed in [26], on top of the MySQL DBMS[8]. The reason for MySQL was that it is lightweight, free and widely used in XML-aware environments. This forced us to assume very little in terms of SQL capabilities since MySQL does not support SQL-92 features like nested queries or even union! The reason for the scheme is that it differs from the traditional edge-vertex encoding and it can answer path queries without the need for transitive closure, feature that is not supported by MySQL.

5.1 Overview of the XML encoding

We now present the main ideas that support the encoding scheme. See [26] for the full details. The encoding will be described on a chunk the *The Merchant of Venice* by William Shakespeare[9].

Every word of the XML document (i.e. words that belong to text elements, not attributes) is indexed according to its order of appearance in the document (starting from 1).

The play, with words indexed

```
<PLAY author=''Shakespeare''>
<TITLE>The[1] Merchant[2] of[3] Venice[4]</TITLE>
<PERSONAE>
<TITLE>Dramatis[5] Personae[6]</TITLE>
<PERSONA>The[7] DUKE[8] OF[9] VENICE[10]</PERSONA>
```

Every node in a document is uniquely identified by its navigational path from the root. The path is unique if navigation in only performed on the child axis. A *simple path* is such a canonical XPath expression where the index values have been removed. The advantage of *simple path*s is that even for large documents, the number of simple paths is usually quite small.

Every node can now be defined by: a reference to the document it belongs to, its *path* and its *position*. The position of a node is always defined by a pair: *(start, end)*. For text nodes, *start*

[7] `http://xml.apache.org/xerces-j`

[8] `http://www.mysql.com`

[9] Markup by Jon Bosak.

Element

pathID	wi	ei	lwi	lei
1	0	1	?	?
3	0	2	4	1
5	4	3	6	1
6	6	2	10	1

Text

pathID	value	wi	lwi
3	The Merchant of Venice	1	4
3	Dramatis Personae	5	6
6	The DUKE OF VENICE	7	10

Attribute

pathID	attvalue	wi	ei
2	Shakespeare	0	0

Path

pathID	pathexp
1	PLAY
2	PLAY/@AUTHOR
3	PLAY/TITLE
4	PLAY/PERSONAE
5	PLAY/PERSONAE/TITLE
6	PLAY/PERSONAE/PERSONA

Figure 8: The encoding of the Shakespeare play

(resp. *end*) corresponds to the index of the first (resp. last) word. For element nodes, *start* is defined as a pair: the index of the first word (or 0) that appears before the element opening tag, and the order of appearance of the opening tag, starting from this word. This second item (called *element index*) is necessary for cases such as `..Venice`^{4}`</TITLE><PERSONAE><TITLE>..`: *start* for the second `<TITLE>` element will be (4,3) since the opening tag comes in 3^{rd} position after the word with index 4. *end* is defined similarly. For an attribute node, *start* and *end* are equal to the *start* value of the parent element node.

The entire encoding scheme can therefore be described by four relations (see Figure 8), with the obvious notations: `wi` and `lwi` for word and last index; `ei` and `lei` for element and last element index; `pID` for `pathID`. We have omitted the `docID` column.

5.2 Systematic translation to SQL

The original paper [26] mentions a translation of XQL [22] queries into SQL but XQL does not support all the navigation axis of XPath. We present below the translation for two axis. The other axis can be translated similarly. See [23] for the full details.

For this encoding, queries are slightly different, depending on the nature of the current node (text or element). Also, some axis require to merge nodes coming from different relations. Queries are parameterized with a context node (the node from which the navigation starts). In Figure 9, information related to the context node is presented in a square box. In the examples below (`CHILD` `DESCENDANT`), we will – for simplicity – only consider element nodes.

In both cases the returned nodes have to belong to the subtree which implies some constraints on the `wi` and `lwi` value. For `CHILD` we must also check that the path is a one-step extension (one extra `/`). What is interesting with this encoding is that navigating the descendant axis is cheaper

than the child axis (in terms of the complexity of the query). This is to be contrasted with other encoding (see [14]) where `DESCENDANT` requires some transitive closure.

child::

```
SELECT * FROM ELEMENT e, PATH p
WHERE  e.docID = node.docID
AND    p.pathID = e.pathID
AND    p.pathexp LIKE node.pathexp + "/%"
AND    p.pathexp NOT LIKE node.pathexp + "/%/%"
AND    e.wi >= node.wi AND e.lwi <= node.lwi
ORDER BY e.wi, e.ei
```

descendant::

```
SELECT * FROM ELEMENT e, PATH p
WHERE  e.docID = node.docID
AND    p.pathID = e.pathID
AND    e.wi >= node.wi
AND    e.lwi <= node.lwi
ORDER BY e.wi, e.ei
```

Figure 9: Translation of XPath to SQL

5.3 Evaluation

Every navigation step is evaluated separately (see 3.4). For an expression like `//item_tuple/description`, Kweelt first expands it to `/descendant-or-self::node()/child::item_tuple/child::description`. It then sends a `DESCENDANT-OR-SELF` query to the database. For each node from the result set, it sends a `CHILD` query and filter out the results that are not `item_tuple`. Then for each result, it sends another `CHILD` query and filters out the results that are not `description`.

This evaluation has two problems. First the evaluation engine gets back some large intermediate results that are pruned inside the engine and not in the backend. Second and more critical, the number of SQL queries is exponential in the number of intermediate steps (see Section 6). This evaluation strategy is extremely naive but corresponds exactly to the one we use for DOM.

We now discuss two improvements. First, queries sent to the database do not use nodetests: the engine gets back a huge result set and has to prune itself. This is stupid because the XPath navigation with node-test can be easily translated. In the examples above, one simply needs to add one condition that tests the suffix of the path (last tag). This fix (fix1) does not reduce the number of queries but reduces the number of tuples returned from the database. Second, queries sent to the database do not take advantage of the encoding scheme. Instead of splitting XPath expressions into single steps, we should try to push as much as we can. Arbitrary XPath expressions can always be translated into SQL using nested queries, but the database we picked does not support nested queries. In this case it is not clear that all XPath queries can be answered. We are currently studying a restricted class of XPath queries that would permit such translation and an algorithm to generate queries with a minimal number of bindings.

6 Benchmarks (preliminary results)

In this section we present some preliminary performance results and lay out some benchmarking plans. For our experiments we picked an XML document that reports the statistics for the 1998 baseball season: the size is 655kb, with 45 unique paths (maximum depth is 6), around 27,000 element nodes and 25,000 text nodes. The file can be found at `http://metalab.unc.edu/xml/examples/baseball`.

Comparing the two backends

We first compare the performance of our two backends against the following query: *return the names of the cities where the team name starts with the letter A*. The query can be translated in Quilt as `//TEAM[startsWith(TEAM_NAME, "A")]/TEAM_CITY` (*Query*1), but can also be rewritten (optimized[10]) as `/descendant::TEAM[startsWith(TEAM_NAME, "A")]/TEAM_CITY` (*Query*2). The evaluation of the query includes navigation, computation of the string value of a node (to check the name of the team) and the final output of the XML tree. The size of the result is 3. We run both queries against DOM (Xerces DOM parser), SQL (relational backend) and SQL+fix1 (Figure 10) We indicate the time to evaluate the query and produce the result. For the SQL backends, we assume that the document is already in the database .

Query	DOM	SQL	SQL+fix1
*Query*1	19s	out of memory	27m ($\sim 55,000$ queries)
*Query*2	16.5s	out of memory	50s (125 queries)

Figure 10: Comparison of backend performance

The first thing to notice is that in both cases the optimized query runs faster. The other thing is that without fix1, the queries cannot be answered. Our current implementation uses JDBC to get the result out of the database and produce/materialize a `NodeList` that will be used for further pruning. Without fix1, the size of the intermediate nodelist corresponds to the number of nodes in the document (element and text). The second thing is that our SQL backend is slower than DOM. But to be fair, we should compare DOM against the best query one can write using the encoding. We have translated by hand *Query*2 (Figure 11) and it gets evaluated in less than 1s.

*Query*2

```
SELECT t2.value
FROM element n1, path n1_p, text t1, path t1_p, element n2, path n2_p, text t2, path t2_p
WHERE  n1.pathID=n1_p.pathID AND t1.pathID=t1_p.pathID AND n2.pathID=n2_p.pathID AND t2.pathID=t2_p.pathID
AND t1.wi>n1.wi AND t1.lwi<=n1.lwi AND t2.wi>n2.wi AND t2.lwi<=n2.lwi AND n2.wi>=n1.wi AND n2.lwi<=n1.lwi
AND n1_p.pathexp="/SEASON/LEAGUE/DIVISION/TEAM" AND t1_p.pathexp="/SEASON/LEAGUE/DIVISION/TEAM/TEAM_NAME"
AND n2_p.pathexp="/SEASON/LEAGUE/DIVISION/TEAM/TEAM_CITY" AND t1.value LIKE "A%"
```

Figure 11: *Query*2 translated by hand to SQL

[10] See Section 7.3.

Even these very limited results are insightful because they show that the benefits of using SQL will only appear if we can take advantage of the encoding. On the one hand every call to SQL is expensive, compared to DOM; on the other hand, a path can be translated into one unique query. This is a strong argument in favor of the automatic translation of location paths into SQL evoked at the end of Section 5.

Order

The second thing we measured is the impact of order on our DOM backends. We evaluate *Query*3 `count(/descendant::PLAYER/*)`, that returns the number of child nodes of the `<PLAYER>` nodes in the document. The query is order intensive because it needs to glue together the child nodes of all the PLAYER nodes. We run the query on 3 DOM backends. the first one uses Xerces via the DOM interface; the second one uses Xerces and the `getNodeIndex` method (see Section 4); the third one uses another implementation via the DOM interface. As expected, computing order via the DOM interface is not a good solution.

Query	Xerces (pure DOM)	Xerces with `getNodeIndex`	XXX (pure DOM)
*Query*3	3m47s	26s	3m5s

These are preliminary experiments that need to be reconducted in a more rigorous way. We would like to measure the quality of a backend in terms of raw performance, amortized performance (for backends like SQL that require pre-loading) and also resource consumption.

7 Mistakes Made, Lessons learned

7.1 XML is not just semi-structured data

The main lesson learned here is that XML is not just semi-structured data (see [1]). If it were, object-oriented databases or semi-structured repositories like Lore [16] would be the solution. The critical aspect of XML is that it is based on a document abstraction.

XML documents do not require any structure and when they carry one, it can be loose (DTD or Schema). In this sense, they are semi-structured.

XML documents are hierarchical and the nesting of elements is a way to represent information.

XML documents use a document-like navigation that is not constrained by the schema (unlike OO databases): from a node, you can navigate wherever you want, following XPath axis. This is radically different from semi-structured query languages that only allow *parent* $\rightarrow$ *child* navigation.

Finally, (some) XML documents require order to be preserved.

If everybody more or less agrees on the mathematical representation of XML (ordered tree or

graph[11]), the encoding and the navigation primitives vary a lot. Most approaches consider *parent* → *child* navigation only while free navigation is what is needed. We try to illustrate the differences in Figure 12 for the Shakespeare's play. The model presented in [31] captures almost all the features, even though order is handled in an ad-hoc way.

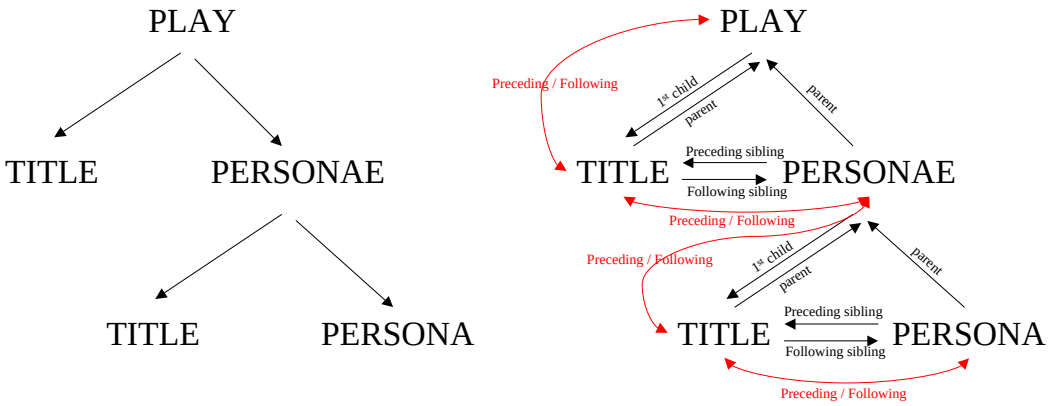


Figure 12: Semi-structured Data vs XML

7.2 Language Design

ASCII syntax A language not based on XML syntax is a good idea because it makes it more concise. However the presence of XML constructs creates some ambiguity with query language constructs (comparison operators and tags). The tradeoff is between the ease of writing queries for the user and the ease of writing the parser for the implementor, a good parser being one that parses efficiently and can return meaningful syntax-error messages. In Kweelt we have decided to reserve `<` and `>` for XML constructs and offer replacements for arithmetic operators[12].

XPath as a sub-dialect The choice of XPath as a sub-dialect of Quilt makes the language easier to learn but sometimes XPath constructs overlap/interfere/contradict with Quilt constructs. For instance XPath overloading and implicit coercions make some queries ambiguous.
The Kweelt implementation is based on an early Quilt proposal that relied heavily on XPath and considers that any XPath expression is a valid Quilt expression, which makes the language slightly more expressive but much more complex to use and implement. The latest Quilt proposal has somehow banished XPath from its syntax and offered some replacements probably inspired from XQL: only major axis are supported and specific functions (`AFTER`, `BEFORE`, etc.) are offered to support the missing features. This restriction should make the implementation and the optimization easier. However, it is important for the language to support – in one way or another – all the axis,

[11] To model references.
[12] `.<.`, `.<=.`, `.>.` and `.>=.`

because they are the specificity of XML as mentioned above (see 7.1).

Other details Some features from the proposal are arguable. For instance `DISTINCT` removes duplicates based on the string value of a node: this ignores attributes and tag names. Aggregate functions are not properly defined on empty lists.

Our implementation of the proposal has lead us to enrich the language itself in the following ways.

In-lined XML Kweelt supports the use of in-lined XML inside the query itself, using a syntax inspired from XML CDATA sections. This feature is extremely convenient and can always be simulated in Quilt by creating a temporary document with the corresponding content. Going even further, this suggests to use the XML data-model itself as a data-type for the query language. User-defined functions defined in Quilt implicitly use it.

User-defined Java functions Kweelt permits the user to enrich the language by importing new functions written in Java. This is very useful for domain specific queries and capabilities related to formatting and access to system information. Such functions are loaded at runtime and evaluated as regular built-ins. This is a good way to distinguish between core functions (that can be optimized) and others.

Dereferencing hints As mentioned in Section 2.2, in the absence of some structural information (DTD, Schemas) some queries are not well defined. The Kweelt syntax is enriched with dereferencing *hints*. Hints are simply pairs (element name, attribute ID name). `@spouse->{Male@name,Female@name}`, will tell the evaluation engine that to dereference attribute `@spouse`, it needs to go through `Male` and `Female` elements, looking for a match on attribute ID `name`. We think that this addition is useful for various reasons: (1) it is optional and can be added by hand by the user; (2) it gives more readability to queries by identifying the type of the element that gets pointed to; (3) it does not force the engine to look for the DTD or Schemas, parse and understand them; (4) it is compatible with any kind of structural description (DTD, Schemas, etc.); (5) one can easily imagine a preprocessing step that would retrieve the DTD/Schema of the documents used in the query and adorn the query with such hints. When there is no hint, Kweelt assumes that the name of the attribute ID is either `"ID"` or `"id"`, which turns out to the case in almost every DTDs we surveyed.

Need for types As a final word concerning the language, our experience testing the W3C use-cases and other more advanced examples showed that writing Quilt queries is prone to mistakes. Misspelling a tag or confusing a tag for an attribute leads to an empty result. A type-system is imperatively needed here. A practical approach would be to mimic what SQL-J[13] does for SQL. Even in the absence of structural information, we could imagine the a type-inference system that would detect some candidate errors (similar element and attribute names) and issue some warnings. This is common for modern compilers.

[13] `http://www.sqlj.org`

7.3 Optimization

In this section we address issues related to optimization of generic XML, which is different from schema specific optimizations presented in [8] or [13], or optimization related to the publication of XML from relational databases like in [24].

When we speak about optimization, it is insightful to look at what we have learned from relational databases. Even though a relational database is a complex system, the core of the optimizer remains relatively simple because it manipulates a small well-understood language (algebra) that contains the primitive components of the query language; and the storage manager deals with only a handful of access methods. When we move to XML, as illustrated by the table below, there is no well accepted/understood equivalent for the algebra and the access paths.

	Relational Databases	XML
Query Language	SQL	Quilt
Algebra	relational algebra	???
Access Paths	scan, index+condition, etc.	???

There are some on-going attempts [19] to map generically XML documents to a relational data-model and use standard relational optimization and rewriting using views. YATL [6] uses an algebra based on object-oriented operators. But they do not capture some key features of XML as enumerated in 7.1 and are backend dependent.

The most recent proposals for an XML algebra are [11] and [2]. These appear to be quite high-level, almost like a query language. The challenge in inventing such an algebra is to rely the right data-model, to identify the right set of primitives (that support the features of the query language), to find equivalence laws and algebraic rewritings and some mappings to access paths.
Another challenge is to come up with an accurate cost-model. This has already been addressed in the context of Lore [20].

Quilt optimization In this work, we did not build an optimizer but we tried to understand the issues faced when doing it. We report here on the experience we gathered in this direction, hoping to contribute to future work. We would like to make one important observation. The choice of the right primitives is not easy. We can imagine to have `CHILD` as a primitive in the algebra and `DESCENDANT` as a derived form. But we saw that our SQL backend executes calls to `DESCENDANT` more efficiently (since it does not require transitive closure). Also some very high costs are often hidden in the query: node comparison or the use of `DISTINCT` require to compute the *string value* of a node which might be very expensive (entire tree traversal in the worst case). Outputting the final document is also expensive.

XPath optimization The big mistake[14] we made for Kweelt was to split XPath navigation paths into navigation steps and have them evaluated one at a time. As we saw, this is killing us for the relational encoding because of the exponential blow-up. This mistake somehow forced us to identify a couple of ad-hoc rewritings that would improve performance and we present some of them.

A direct expansion of XPath abbreviations can lead to very slow queries. If we remark that `/descendant-or-self::node()/a/` and `/descendant::a/` are equivalent, we can get rid of one location step when we translate queries of the form `//child::node-test/`[15]. This can save a lot (remember *Query*1 and *Query*2 from Section 6).

XPath predicates are very subtle (see 2.1), but can be used to speed-up queries in Kweelt. First, given our evaluation strategy, it is always beneficial to move conditions from the `WHERE` clause to XPath. In the example of Section 2.2, an optimizer would have rewritten the query by pushing the predicate of the `WHERE` clause (line 7) into the XPath expression (line 5) that creates the `$i` binding. Second we should also order predicates by selectivity to decrease the size of intermediate results.

DOM-backend specific optimizations DOM implementations offer means to read and write XML. For Kweelt, having a read-only implementation would probably make the navigation faster and would consume less resources. When nodes are immutable, optimized data-structures can be used. Splitting DOM into a read-only API would be useful.

Some DOM implementations manage nodes in a lazy way: Xerces[16] create `DeferredNode`s for which the content is loaded as needed. Memory is also managed carefully via buffers and pools. Kweelt DOM backends cache document, to avoid multiple parsing. When resources are scarce, node caching (finer granularity) would be better. It is not clear what are the good caching strategies: this would require to trace some navigation patterns. Another promising optimization is to cache node string values instead of traversing the tree every time. If all the text nodes are stored adjacently, a string value can be represented as an offset.

SQL-backend specific optimizations The first thing is to push as much work to the backend using as few queries as possible. When some computation has to be performed outside of the engine, we should try to avoid to materialize intermediate results. Using the encoding presented, some navigation primitives require two queries and force the results to be merged outside of the database. By using a DBMS that supports `UNION`[17] and with minor modifications in the encoding we should be able to use only one query. But even as is, we can merge the two result sets using a stream based algorithm. When using JDBC, this would solve some of our out of memory problems.

[14] The mistake can be repaired by modifying the Node Factory API and the Kweelt evaluation strategy.

[15] We use a trailing `/` to make sure that there are no predicates that could interfere.

[16] `http://xml.apache.org/xerces-j`

[17] The MySQL DBMS does not!

Concerning the encoding, it clear that the right choice of indexes is crucial. Navigation relies of the implicit notion of regions which can be captured using R-Trees for instance. Also, since a lot of query operations are string comparisons (for paths), we should make sure that they are implemented efficiently by the database engine. The clustering of document nodes is also an issue. We currently load the document as we parse it. The placement of nodes on disk might not be the best one.

7.4 Need for better APIs

To query XML, DOM is not enough. We thought that an API based on XPath navigation axis would be, but we were wrong: it is not when we use backends with rich navigation capabilities. An interesting observation is that interfaces are very convenient because they abstract, but can be harmful when they abstract too much. In the case of DOM for instance, the computation of node order is very expensive when performed via the interface while we get it for free when we access the implementation directly.

DOM always return the full result set of a navigation. In the Shakespeare example, `(//PERSONA)[1]` will return all the `<PERSONA>` and Kweelt will then have to filter out all but the first one. Having all navigation primitives be iterator-based [17] would permit to avoid such situations. When dealing with range index, it is important not to materialize intermediate but also to know when to stop. Once the first `<PERSONA>` has been returned, there is no point in looking further. Another promising venue is the use of XML specific operators like XScan [18].

The challenge in designing the right APIs is to permit to pass enough information to the backend. A good API should make it possible to take advantage of some full-text or proximity-search capabilities in the backend.

7.5 Need for a better SQL

The implementation of the SQL backend revealed some serious limitations of SQL for our specific use. Current database implementations do not permit to define new aggregate functions (aka column functions). In our backend, we would like to grab all the nodes in a sub-tree, sort them and aggregate their content. This is needed when computing a node string value or when outputting the result. Another important need is to be able to order a set and return a sub-range like in `(//PERSONA)[5]`. Surprisingly we cannot do that in SQL[18] and Kweelt needs to go through each item of the result set to get the correct one(s).

[18] By digging in the IBM DB2 reference manuals, we found the `RANK` function from the OLAP extensions that could allow us to hack our way through.

7.6 Some Subtle Issues

Order Order is a luxury for which one has to pay (see results from Section 6). Our implementation default policy is to return ordered results. Whenever two nodesets have to be merged, the order must be preserved. This happens for query operators like `UNION` but also when we evaluate XPath expressions and split location paths. The bad news is that we need to check order quite often. The good news is that XPath navigation axis always return ordered nodesets, either is document or reverse document order. Two such nodesets do not need to be sorted, but simply merged in order, which is linear. Kweelt always tries to preserve order but there are cases where it cannot (e.g. nodes coming from various documents). For such cases, nodes are simply appended, with no order. Being able to detect early on when two nodesets cannot merged in order (by tagging them with a special flag) would lead to better performance. The issue of order is less relevant when we have access to efficient ways to compare node order, like the one presented in 7.3.

Node Copy vs Node Reference Based on our idea of delegation, Kweelt handles node references (pointers to nodes stored by the backend) rather than actual values. The cost of a node is much cheaper and the engine can manipulate more nodes at the same time.

This seemed to be a natural and fine decision but for two Quilt constructs: `SHALLOW` and `FILTER`. `SHALLOW` takes a node and strips it from its sub-elements. `FILTER` takes a forest and a set of nodes and returns only the forest nodes that appear in the set, while keeping the hierarchical and sequential relationships among forest nodes. These two constructs imply changes in the structure of the underlying document. The consequence of our decision is that forests created using these operators have inconsistencies: nodes with no parent, nodes with two parents, etc.

We illustrate the issue with the diagrams of Figure 13. The XML tree (this is a rooted forest) is represented in black solid lines. The result of the query (nodelist) is represented with gray arrowed boxes. Dotted lines correspond to pieces of the tree that have been removed.

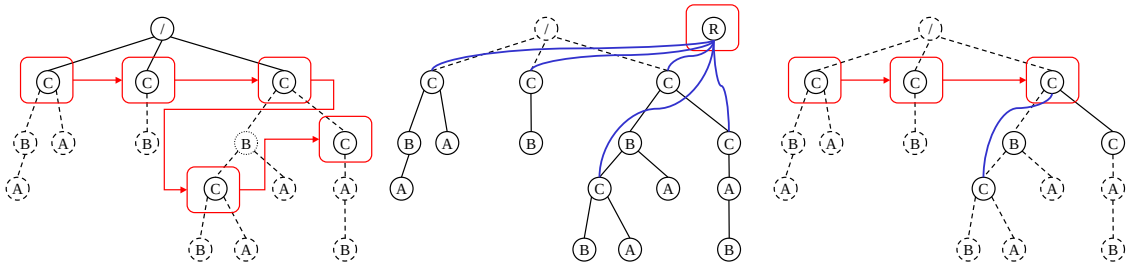


Figure 13: Quilt tree transformations

`SHALLOW` and `FILTER` again The two operators cannot be exported to the backend. In the case of the former, we actually use the following trick. We define a new type of nodes called *shallow nodes*

that encapsulates a regular node. The only difference is that its output method skips the subtree. For the latter, we have no other solution but to do the work in the engine.

7.7 Our Implementation

A final word about the lessons concerns the design of the implementation itself. Kweelt is written is Java 1.2. The parser has been generated using JavaCC which makes it very easy to enrich the language (XPath abbreviations are directly handled by the parser). The framework represents 200kb of byte-code. By using API and various design patterns [15] (Factory, Proxy, Iterator to name a few), the code is modular, readable and easy to maintain and modify. The code for KSP represents less than 150 lines.

The Kweelt source code is freely available from `http://db.cis.upenn.edu/Kweelt`.

8 Related Work

There is a luxuriant literature on XML, XML data-models and optimization. We have already referenced some of it across the paper. Unfortunately, there is very little that addresses issues related to ordered navigation, query evaluation for multiple storage backends and efficient implementation. The works that relate closely to ours are Lore [21], YAT [6] and XML-QL [7]/Strudel [9][19], all of which have been implemented.

XML-QL and YATL (YAT language) are based on a pattern-matching query paradigm. Lore uses path-expressions. None of them supports XPath navigation beyond `CHILD` and `DESCENDANT` . They supports though regular path expressions on the `CHILD` axis (inspired from [3]), which do not seem to be used in practice. See [12] for a comparison of these query languages.

The XML-QL implementation is built on top of the StruQL engine [10]: XML documents are transformed into StruQL graphs; the query is applied on the graph to produce another graph; the resulting graph is transformed into an XML document. Lore uses its own object repository. YAT is a middleware architecture with source capability based rewriting. It also supports ODBC.

XML-QL relies on some StruQL graph optimizations. Lore's optimization is based on path-expressions and makes use of a cost-model. Queries are mapped into the Lore algebra and compiled before being executed. The YAT algebra comes with the **Bind** and **Tree** operators to construct/decontruct arbitrary trees into relations along with traditional operators from object-oriented algebras. YATL also supports types.

We should not forget to mention that there are numerous commercial and non commercial imple-

[19] XML-QL is implemented on top of Strudel.

mentations of XQL, XPath, XSLT engines. Because of a lack of published documents, it is hard to compare with them. Most of them use DOM and text files as a backend. GMD-IPSI XQL query engine[20] uses a persistency extension on top of DOM to manage large documents. FatDog XQL[21] requires to index document first to *pre-build the internal data structures needed to enable subsequent fast retrieval.* Database and middleware vendors offer multiple solutions, where XML can be mapped to relational or stored as a BLOB.

9 Conclusion and Future Work

In this paper we have reported our experience in building Kweelt, a Java framework for querying XML, based on the Quilt query language proposal.

We can summarize our contribution in the following way. We have provided a reference implementation for the Quilt proposal and even added some improvements to the language. The implementation runs all the use-case examples from W3C [30]. Kweelt offers a modular and extensible platform to conduct experiments related to XML and we report some of them. We have developed two different storage managers that can be used by Kweelt as XML backends and presented some preliminary performance results. We have also pinpointed some optimization issues specific to each backend. More importantly, we have debunked a couple of myths concerning the cost of XML navigation: computing the descendant of a node does not necessarily require transitive closure and can actually be cheaper than computing its children. This means that regarding the descendant axis as the transitive closure a the child axis is not always a good idea. This is something to keep in mind when designing an algebra. We have also showed that order preservation – if not handled properly – is a luxury that can hardly be afforded.

In hindsight, using XPath navigation axis as the core of our Node Factory API was a terrible mistake as demonstrated by the relational backend: Kweelt issues more than $50,000$ queries to the database when a single one is enough. Our mistake was to choose the wrong granularity for the tasks sent to the backend (location step instead of location path).

Our immediate future work is to define and implement a systematic translation of navigation path into SQL, minimizing the number of subqueries and bindings. Another line of work is to investigate and evaluate a binary file backend. We also would like to better understand Quilt `SHALLOW` and `FILTER` operators and see how they can be supported by the backend. Our second more long term goal is to identify the right set of primitives for an XML algebra. We hope that the experience with Kweelt we have reported here will help us and others achieve this goal.

Acknowledgements: The author would like to thank Val Tannen for fruitful comments on a draft

[20] `http://xml.darmstadt.gmd.de/xql`

[21] `http://www.fatdog.com/`

of the paper; Laurent Dupont for the early prototype of Kweelt; Thien-Loc N'Guyen for a backend based on a binary encoding of XML; David White for the SQL backend; the Quilt authors.

References

[1] Serge Abiteboul, Peter Buneman, and Dan Suciu. *Data on the Web : From Relations to Semistructured Data and XML* . Morgan Kaufmann, October 1999.

[2] David Beech, Ashok Malhotra, and Michael Rys. A Formal Data Model and Algebra for XML, 1999. Available at `http://www-db.stanford.edu/dbseminar/Archive/FallY99/malhotra-slides/ma%lhotra.pdf`.

[3] Peter Buneman, Susan Davidson, Gerd Hillebrand, and Dan Suciu. A Query Language and Optimization Techniques for Unstructured Data. In *ACM-SIGMOD*, Montreal, Canada, June 1996.

[4] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, March 1994.

[5] Don Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *Invited paper, WebDB-2000*, May 2000.

[6] Sophie Cluet, Claude Delobel, Jérôme Siméon, and Katarzyna Smaga. Your mediators need data conversion! In *ACM-SIGMOD*. ACM Press, 1998.

[7] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A Query Language for XML. In *Proc. of 8th International WWWW Conference*, 1999.

[8] Alin Deutsch, Mary F. Fernandez, and Dan Suciu. Storing Semistructured Data with STORED. In *SIGMOD*. ACM Press, 1999.

[9] Mary Fernandez, Daniela Florescu, Jaewoo Kang, Alon Levy, and Dan Suciu. STRUDEL: A Web Site Management System. In *ACM-SIGMOD*, Tuscon, May 1997.

[10] Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A Query Language for web-site Management. *SIGMOD Record*, 26(3):4–11, September 1997.

[11] Mary Fernández, Jérôme Siméon, and Phil Wadler. A Data Model and Algebra for XML Query. FST TCS, Delhi, December 2000.

[12] Mary Fernandez, Jérôme Siméon, and Philip Wadler. XML Query Languages: Experiences and Exemplars. Communication to the XML Query W3C Working Group, September 1999.

[13] Mary Fernandez, WangChiew Tan, and Dan Suciu. SilkRoute: Trading between Relations and XML. In *Proc. of the 9th WWW Conference*, May 2000.

[14] D. Florescu and D. Kossmann. Storing and querying XML data using an RDBMS. *Data Engineering Bulletin*, 22(3), 1999.

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[16] Roy Goldman and Jennifer Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB'97*, 1997.

[17] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2), 1993.

[18] Zachary G. Ives, Alon Y. Levy, and Daniel S. Weld. Efficient Evaluation of Regular Path Expressions on Streaming XML Data. submitted for publication, 2000.

[19] Ioana Manolescu, Daniela Florescu, Donald Kossmann, Florian Xhumari, and Don Olteanu. XML and Relational: How to Live with Both. In *VLDB*, September 2000.

[20] Jason McHugh and Jennifer Widom. Optimizing branching path expressions. Technical report, Stanford University, 1999. Available from `http://dbpubs.stanford.edu`.

[21] Jason McHugh and Jennifer Widom. Query Optimization for XML. In *VLDB*, 1999.

[22] Jonathan Robie, Joe Lapp, and David Schach. XQL (XML Query Language). QL'98, September 1998. Available at `http://www.w3.org/TandS/QL/QL98/pp/xql.html`.

[23] Arnaud Sahuguet. Translation of XPath navigation primitives into SQL for a relational encoding of XML. Unpublished draft, October 2000. Available from `http://db.cis.upenn.edu/Publications`.

[24] Jayavel Shanmugasundaram, Eugene J. Shekita, Rimon Barr, Michael J. Carey, Bruce G. Lindsay, Hamid Pirahesh, and Berthold Reinwald. Efficiently Publishing Relational Data as XML Documents. In *VLDB*, 2000.

[25] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB*, 1999.

[26] Takeyuki Shimura, Masatoshi Yoshikawa, and Shunsuke Uemura. Storage and Retrieval of XML Documents using Object-Relational Databases. In *DEXA*, volume 1677. Springer-Verlag, 1999.

[27] Feng Tian, David J. DeWitt, Jianjun Chen, and Chun Zhang. The Design and Performance Evaluation of Various XML Storage Strategies.

[28] W3C. XML Path Language (XPath) 1.0. W3C Recommendation 16 November 1999. Available from `http://www.w3.org/TR/xpath`.

[29] W3C. XSL Transformations (XSL-T) 1.0. W3C Recommendation 16 November 1999. Available from `http://www.w3.org/TR/xslt`.

[30] W3C XML Query Working Group. XML Query Requirements, August 2000. `http://www.w3.org/TR/2000/WD-xmlquery-req-20000815`.

[31] Philip Wadler. A formal semantics of patterns in XSLT. In *Markup Technologies*, Philadelphia, December 1999.

[32] Philip Wadler. Two semantics of XPath. Note to the XSL W3C working group. Available at `http://www.cs.bell-labs.com/who/wadler/papers/xpath-semantics/xpath-sem%antics.pdf`, January 2000.