A Peformant XQuery to SQL Translator

Christopher Ré, James Brinkley and Dan Suciu
University of Washington, Seattle
Technical Report #2006-06-02

June 6, 2006

Abstract

We describe a largely complete and efficient XQuery to SQL translation for XML publishing. Our translation supports the entire XQuery language, except for functions, if statements and upwards navigation axes. The system has three important properties. First, it preserves the correct XQuery semantics. This is accomplished by first translating XQuery into core-XQuery, using a complete XQuery implementation, Galax. Second, we optimize the resulting SQL queries. We develop a comprehensive framework for optimizing the XQuery to SQL translation, which is effective for a wide range of XQuery workloads. Third, our translation is platform independent. Our system achieves high degree of efficiency on a wide range of relational systems. This paper reports an extensive experimental validation on several XQuery workloads, using MySQL, PostgreSQL, and SQL Server, and compares this approach with five native XQuery engines: Galax (the newer, optimized version), Saxon, QizOpen, IMDB and Quexo.

1 Introduction

XQuery can be implemented in at least three ways: as a native XML engine; on top of an existing relational engine's infrastructure; or by translating it into general SQL.

Galax [7], Natix [9], and Timber [16] are examples of native XML engines. Such engines need to reimplement most of the functionality of a relational database system, e.g. the storage manager, the query processor, and the optimizer. This engines have more flexibility to support all features of XQuery and can be optimized specifically for XQuery. It is a matter of debate, however, how long this approach will take to catch up with relational technology.

Xperanto [3, 19, 18, 20], and the XQuery implementation in Yukon [14, 15] are examples of the second approach. These systems evaluate an XQuery expression by using their existing relational operators (joins, index lookups, group-by's etc), and optimize the operator expression using their existing optimizers. Major database vendors favor this approach because it allows them to leverage

the huge infrastructure that they already have for processing relational data. Vendors support only a fragment of XQuery, namely the fragment that maps naturally to their engine's relational operators. As a result, different vendors support different fragments. Moreover, not all relational engines have XQuery extensions: most notably the popular open source systems MySQL and PostgreSQL do not support XQuery.

This paper is about the third approach: translate XQuery expressions into SQL statements in order to execute them on *any* relational database system. This middleware approach is useful in applications that already use a relational database and would like to offer an XML/XQuery interface, without having to migrate to a new database server that supports XQuery. This task is known as *XML publishing*. The middleware approach is attractive because it does not depend on a specific database platform. However, this approach is technically quite difficult, because of the huge mismatch between XQuery and SQL. In addition, the translation often results in very complex SQL expressions, which need to be *optimized* by the middleware before being sent to the relational engine. A translation algorithm was described in SilkRoute [5], but it assumed an unordered data model for XQuery. Some specific optimization techniques have been described in [19, 6] (optimizing the XML construction), and [13] (eliminating foreign key joins and unions), but no complete framework for optimizing the translated queries exists.

We describe in this paper a complete, and efficient XQuery to SQL translation. Our system, called SilkRoute II, translates arbitrary XQuery expressions to SQL, and faithfully preserves the XQuery semantics. The system supports the XQuery language, except for functions, if statements [1] and upwards navigation axes. SilkRoute II has three important properties.

The first property is that it preserves the correct semantics, while capturing the entire language. For that, we use a reference XQuery implementation, Galax, to first translate an XQuery expression into core-XQuery. Our translation algorithm applies to core-XQuery expressions that are automatically generated by Galax. This ensures that we capture the right semantics during translation. The downside is that the translation becomes difficult, since it is no longer restricted to some toy XQuery fragment. One contribution of this paper is to develop an intermediate representation (IR) for mappings from relational databases to XML, which enables the translation of the entire language.

The second property is that it generates highly efficient SQL queries, for a wide range of XQuery workloads. We describe here a wide range of optimization techniques, in addition to those previously mentioned in the literature, which improve the performance of the resulting SQL queries. While some of the optimization on this list have been discussed before in various settings, to the best of our knowledge this paper is the first to present a comprehensive list of optimization techniques that apply to a variety of XQuery workloads. One contribution of this paper is the optimization framework for the XQuery to SQL

[1] If statements can be compiled as the union of the true and false portions of the branch combined with the condition and negation, respectively.

translation.

The third property is that our system is platform independent. Moreover, it achieves high efficiency on a wide range of relational systems. We do this by experimenting with several platforms and making choices in our optimization techniques that apply across multiple platforms. We do allow some minimal platform specific hooks, for example to estimate cost over a particular engine.

We validate our system on several workloads, and over several relational engines (MySQL, PostgreSQL, and SQL Server). We also compare this with several existing native XQuery engines: Galax, Saxon, QizOpen, IMDB and Quexo. The extensive experimental results, including the comparison of two different approaches to XQuery evaluation, are one contribution of this paper.

Our framework is that of the XML publishing problem, when a relational database is given and the XQueries are over an XML view over the database. The view may be hierarchical, arbitrarily deep, but not recursive. We do not address the XML storage problem, where the relational schema is designed specifically for XML storage and processing: see [12] for a detailed explanation of the differences between the two.

While our results are of direct interest for XML publishing, they also shed light on trade-offs of relational and XML engines. We found that SilkRoute II in conjunction with a high performance database system (SQL Server) outperformed *all* five native XQuery engines that we tried, on *all* query workloads, and on *all* data sizes (both small and large). Even with the least efficient relational database system of the three we tried (MySQL), SilkRoute II still outperformed the native XQuery engines on most (but not all) queries. Moreover, none of the native XQuery engines scaled to large data sets, while SilkRoute II did over all three relational engines. These findings point both to the maturity of relational technology and to the high quality of the SQL queries generated by SilkRoute II. We found that naively generated SQL queries run much slower on the relational engines, and are easily outperformed by the native XQuery engines on small datasets. Naive SQL queries sometimes did not terminate or could not run at all. Thus, the optimizations performed by SilkRoute II had a dramatic impact, on many queries in the workloads we tried. But perhaps the most surprising finding was that most of the XQuery native engines supported a fragment of XQuery that is less complete than that supported by SilkRoute II. This was unexpected. Given the difficulties in translating XQuery to SQL, we expected native XQuery engines to support a larger fragment than SilkRoute II.

The paper is organized as follows. Section 2 shows some of the challenges encountered by a complete and performant XQuery to SQL translation. Sec. 3 describes the system's architecture. Sec 4 describes our new intermediate representation. Sec. 5 describes the optimization framework. Sec. 6 contains the experimental validation. Related work is in Sec. 7. We conclude in Sec. 8.

2 Challenges

XQuery and SQL are based on different paradigms (imperative v.s. declarative) and manipulate different data types (ordered sequences v.s. sets of tuples). We illustrate here concretely some of the challenges faced by a general XQuery to SQL translator. Our examples are drawn from the XBrain project [2]. SilkRoute II is an XML publishing system, not an XML storage system (see the taxonomy in [12]), hence it assumes a given relational database and does not design one from scratch in order to execute XQueries. It is based on the GAV publishing approach. In the following examples we omit describing the relational schema, but mention that most parent/child relationships in the XML view correspond to key/foreign key relationships in the relational data. In each example the relevant parts of the relational schema will be obvious.

Verbose Formal Semantics XQuery has both a formal and informal semantic. SilkRoute II internally deals with the formal semantic representation of XQuery. For example, consider the following XQuery:

```
for $c in /patient/surgery/csmstudy,
 $e in /patient/surgery/anatomymap/anatomymapelement
where $c/trial/stimsite/text() = $e/stimsite/text()
return <result> { $c/oid/text(), $e/site_label }
       </result>
```

This seems like a simple join, however each of the two join expressions can potentially be a sequence. As a result the normalized core expression looks something like:

```
for $c in /patient/surgery/csmstudy,
 $e in /patient/surgery/anatomymap/anatomymapelement
where (Some $s1 in $c/trial/stimsite/text()
        (Some $s2 in  $e/stimsite/text()
             return op:equal($s1,$s2)))
return <result> { $c/oid/text(), $e/site_label }
       </result>
```

In order to faithfully implement the semantics of XQuery, SilkRoute II relies on the normal query form generated by a reference implementation of XQuery [7]. This normal form is incredibly complex, even for simple queries, which makes the translation task much harder. However, we argue that any complete XQuery to SQL translator should use this normal form as a starting point, in order to correctly adhere to the semantics.

Query Flattening Continuing the previous example, a naive translation into SQL could look like the following.

```
SELECT C.oid, E.site_label
FROM Anatomymapelement E, Csmstudy C
WHERE EXISTS
  (SELECT * FROM Trial T
   WHERE T.cid = C.cid and E.stimsite = T.stimsite)
```

In general two `EXISTS` quantifiers should be generated, but we used here the fact that each `anatomymapelement` has a unique `stimsite` subelement.

Such nested SQL queries frequently arise when translating from XQuery. There is a wide range of behaviors in the way relational engines handle nested subqueries. Most commercial systems optimize them seamlessly. However, earlier versions of MySQL do not even support nested queries. The latest version

allows them but with with very poor performance. Query flattening will allow us to rewrite the following equivalent query:

```
SELECT DISTINCT C.oid, E.eid, E.site_label
FROM Anatomymapelement E, Csmstudy C, Trial T
WHERE T.cid = C.cid and E.stimsite = T.stimsite
```

The `DISTINCT` keyword eliminates duplicates corresponding to multiple `Trial`s. Notice that we had to add the key attribute `E.eid` to preserve the same semantics. The middleware will receive a tuple stream that includes this attribute, but will simply ignore it, and construct the XML answer using the other two attributes. All relational engines, even the non-commercial engines, we tried handle such SQL queries seamlessly.

Trimming Queries and Minimization Consider the following query:

```
for $p in /patient[surgery/trial/trialcode/text()=2]
return
  <patient>
    { $p/name,
      for $s in $p/surgery[trial/trialcode/text()=2]
      return <surgery>
              { $s/date,
                $s/trial[trialcode/text()=2]
              }
             </surgery>
    }
  </patient>
```

This is a typical trimming query, adapted from a real (and more complex[2]) query on the Website [2]: find all patients who had a surgery with a trial with a trial code 2; for these patients list only their surgeries that had a trial code 2; etc. There is some redundancy present in this XQuery: if a `trial` element satisfies the condition:

`trial[trialcode/text()=2]`

then its parent `surgery` element satisfies the condition

`surgery[trial/trialcode/text()=2]` ;

similarly the grand-parent `patient` satisfies

`patient[surgery/trial/trialcode/text()=2]`

This redundant form is, nevertheless, the only way to express such trimming queries in XQuery. The problem is that this redundancy propagates into the resulting SQL. For example, the SQL query corresponding to the innermost `trial` element is the following horrendous expression:

```
SELECT t.trial_number, t.slide
FROM trial AS t
WHERE EXISTS(SELECT * FROM trialcode e
             WHERE e.tid = t.tid and e.code=2)
 and EXISTS(
        SELECT * FROM surgery AS s
        WHERE EXISTS(
                SELECT * FROM trial AS t2, trialcode AS e2
                WHERE t.sid = s.sid and s.sid=t2.sid
                  and t2.tid = e2.tid and e2.code=2)
           and EXISTS(
                SELECT * FROM patient AS p
```

[2]We dropped the `csmstudy` element between `surgery` and `trial`.

```
WHERE s.pid = p.pid and EXISTS(
   SELECT *
   FROM surgery AS s3 trial AS t3, trialcode AS e3
   WHERE p.pid = s3.pid and s3.sid = t3.sid
     and t3.tid = e3.tid and e3.code=2)))
```

After eliminating the redundancies, however, the query becomes equivalent to:

```
SELECT t.trial_number, t.slide
FROM trial AS t
WHERE EXISTS(SELECT * FROM trialcode e
             WHERE e.tid = t.tid and e.code=2)
```

The process that simplifies the first query into the second one is called *query minimization* and has been intensively studied in the theoretical database community [1]. However, to the best of our knowledge no database systems today implements query minimization. We found that by implementing minimization in the middleware one can considerably increase the performance of trimming XQuery expressions, which are quite common in some real life workloads.

Order XQuery manipulates ordered sequences, while SQL manipulates sets of tuples. Order is strictly controlled in XQuery, in two ways: by the document order (the default), and through the `order by` clause. Consider:

```
for $p in /patient
order by $p/dateOfBirth/data()
return <patient>
        { $p/name,
          for $s in $p/surgery
          order by $s/physician/name/text()
          return <surgery>
                    ...
                    $s/trial[trialcode > 2]
                 </surgery>
        }
      </patient>
```

There are three orders here: the `patient` elements are ordered by their date of birth; the `surgery` elements are ordered by the physician's name, and the `trial` elements are in document order. Each of the corresponding SQL statements must be carefully constructed in order to preserve the intended order, while allowing the middleware to merge-join the tuples corresponding to the three nested elements. The challenge here is for the middleware system to have a good representation of the intermediate order of each subcollection, to be able to trace the intended order through various constructs, including intermediate expressions introduced by `let` bindings.

Position predicates Consider the following simple XPath expression:

```
patient[3]/surgery[trial[1]/trialcode/text()='2']
```

This returns all surgeries of the third patient, where the first trial had some trial code 2. Using bare-bones SQL, the following query corresponds to `patient[3]`:

```
SELECT X.*
FROM Patient X, Patient Y
WHERE X.pid >= Y.pid
GROUP BY X.pid
HAVING count(Y.pid) = 3
```

The SQL query returning `patient[3]/surgery/trial[1]` is horrendous. Some database systems support the `SELECT TOP 3 ...` construct, which can be used to implement position predicates and results in somewhat more efficient queries. By contrast, position predicates are supported very efficiently by native XML engines, since they just have to lookup the third, or the first element in an ordered list. We show here, however, that by adding simple position information to the relational database it is possible to optimize most position queries significantly.

Loop Invariants XQuery can be used for document transformation. As a result we have encountered queries like the following:

```
for $p in /patient,
    $s in $p/surgery[.//trialcode=2]
return <surgery>
        { $s/photo,
          $p/imagingstudy/mrseries[mrslice/text()>5]
        }
       </surgery>
```

The problem here is that the same collection of `mrseries` is copied in all surgeries of the same patient. A naive execution of this query will compute the expressions
`$p/imagingstudy/mrseries[mrslice/text()>5]` redundantly, once for each surgery of the patient `$p`. Of course, this is a loop invariant, and can be factored out in a `let` statement. Standard compiler technology can detect loop invariants automatically and move them outside the loop. Some modern XQuery processors already implement this. However, this is much harder to implement in an XQuery to SQL translator, since its actions are limited to submitting SQL queries and merging their resulting tuple streams. The issue is that the middleware needs now to store the XML fragment that is loop invariant. This is simple for the query above, since only one value of the loop invariant needs to be stored by the middleware. It can be much more difficult in general, as illustrated by the following:

```
for $p in /patient,
    $s in $p/surgery[.//trialcode=2]
order by $s/date/text()
return <surgery>
        { $s/photo/text(),
          $p/imagingstudy/mrseries[mrslice/text()>5]
        }
       </surgery>
```

Now surgeries from different patients are interleaved, and the middleware needs to store the value of the loop invariant for each patient separately.

Heterogeneous Sequences In XQuery it is possible to iterate over sequences of different types. For example it is easy to construct, in XQuery,

an heterogeneous collection of `patient`s and `physician`s. The structure of a `patient` element is different from that of a `physician` element, and hence different SQL queries are needed to construct them. If all `patient` elements precede the `physician` elements, then the middleware could use `UNION ALL`. But if the elements are shuffled, e.g. if the XQuery expression sorts them by their `name`, then an alternative is to return the two tuples streams separately, and let the middleware merge them.

Varying Capabilities of SQL Engines Efficient translation from XQuery to SQL relies on different and sometimes ill-supported features. For example, our queries are highly nested. Such highly nested queries pose problems for optimizers and even for execution against some older engines.

3 Architecture

SilkRoute II is a middleware system for exporting an existing relational database to XML. We describe here its architecture.

Canonical XML View Initially, the data owner sees a *canonical view* of the underlying relational database: each table is an element, having multiple `row` subelements, with one subelement for each attribute of that table. This is a standard conceptual XML view of a relational database, roughly corresponding to the `XML RAW` mode in SQL Server.

Public XML View The data owner starts by defining a *Public XML View* over the relational data. This is expressed as a large XQuery program over the canonical view. This program can have, say, 1000-2000 lines, or can be even larger, in order to construct a complex, hierarchical XML view of the entire relational database. New XML views can be easily defined in terms of existing once, since XQuery is fully compositional, and SilkRoute II fully supports the compositionality. The public view (or views) is kept virtual.

User XQueries End users see only the public XML view of the relational database, not the actual relational database. They submit XQuery expressions over that view, as if it were a materialized XML document. They can also refer to multiple XML views of the same relational database, but to keep our discussion simple we will assume throughout the paper that they refer to a single view, called the public XML view. SilkRoute II composes the user's query with the public view definition, which results in an XQuery expressed directly over the canonical view. This is the query that is translated into one or several SQL queries and executed on the relational engine. Notice that it is easy for users to materialize the entire public view if the want that: they just write an XQuery that returns the root of the public view.

Intermediate Representation Internally, SilkRoute II uses an *intermediate representation*, IR, to represent XQueries over the canonical view. We have designed the IR specifically to enable the large variety of the optimizations needed in the XQuery to SQL translation: the IR is described in some detail in Section 4. The IR is compositional in the following sense: given an intermediate representation I that maps the relational database to some XML view, and an

XQuery expression Q over that XML view, then it is possible to construct a new intermediate representation J that is equivalent to $Q \circ I$: the meaning of J is that it maps the relational database directly to the result of Q. In the first application of compositionality, I is the canonical view and Q is the public view definition: then $Q \circ I$ results in an internal representation of the public view. In the second application, I is the internal representation of the public view, and Q is the user's XQuery: then $Q \circ I$ results in an internal representation that needs to be translated to XQuery, then optimized and executed.

Formal Semantics Each XQuery is first translated into the the XQuery Core language, which is the standard XQuery semantics. We used the translation module in the reference XQuery implementation Galax [7] for this task. This enables SilkRoute II to faithfully implement the standard XQuery semantics. This point should not be underestimated: previous XQuery to SQL translations use a direct approach, thus feeling free to twist the XQuery semantics, or to restrict the language, whenever a translation to SQL is too difficult. We feel that, for a middleware approach to XQuery to be credible, one needs to support the full language.

SQL Queries and Tagger When an IR needs to be executed, SilkRoute II generates one or more SQL queries. The number of SQL queries depends only on the IR expression, and not on the data, and is typically small (1 or 2 for all XMark queries, between 1 and 6 for the more complex HBP queries; see Sec.6). These SQL queries are then rewritten and optimized: this phase is described in Section 5.

Finally, SilkRoute II issues all queries against the relational engine, using an ODBC connection. Then, it reads the resulting tuple streams, merges them, and adds the XML tags. Each SQL query has an `ORDER BY` clause, ensuring that the tuples return in the right order, hence SilkRoute II needs to hold in main memory only one tuple from each tuple stream. This is called a *constant space XML tagger* in [19].

4 Intermediate Representation

Designing a good intermediate representation, IR, for representing mappings from relational data to XML is crucial for a complete and efficient SQL to XQuery translation. Such a representation needs to meet three conflicting goals

- It must be complete, i.e. capable for representing any mapping that can be expressed in XQuery, while faithfully preserving its semantics.
- It must SQL-like: in other words, it should be easy to read from it the SQL queries that would implement that mapping.
- It must be compositional, i.e. it must be easy to apply an XQuery to a given IR and obtain a new IR.
- It must be simple, to allow optimizations.

```
[IR]  ::= [IRS]*
[IRS] ::= FROM [SQL-from-tables] [IR]
        | WHERE [SQL-where-condition] [IR]
        | ORDER BY [SQL-order-by-attributes] [IR]
        | <xmltag > [IR] . . . [IR] </xmltag>
        | LEAF [SQL-select-expression]
        | LEAF [XQuery-function](IR)
```

Figure 1: Definition of the IR Language. Attributes are omitted to avoid clutter.

We define our IR in Fig. 1. An IR expression is best visualized as a tree (more accurately: a forest), where each node is labeled by one of the following: `FROM`, `WHERE`, `ORDER BY`, an XML tag, or `LEAF`. The figure omits attributes to reduce clutter, but they are easy to add. We will illustrate next the IR language through several examples, explaining how it addresses the three conflicting goals.

Consider the following XQuery expression:

```
XQ1:
for $x in /patient
return
  <patient> <name>{ $x/name }</name>
            <surgeries> {
               for $y in $x/surgery
               return <id> { $y/sid } </id>
            </surgeries>
  </patient>
```

This is translated to the following in the intermediate representation.

```
IR1:
ORDER BY p.pid
FROM Patient p
 <patient>  <name> LEAF( p.Name ) </name>
            <surgeries>
                ORDER BY s.sid
                FROM Surgery s
                WHERE s.patient_id = p.pid
                <id> LEAF( s.sid ) </id>
            </surgeries>
 <patient>
```

The meaning of the `FROM` clause is that it generates a set of patient tuples `p`, and for each constructs a `<patient>` element. The `ORDER BY` clause specifies that this set has to be ordered by `p.pid` (assumed to be the document order in our simple example). Underneath a `<patient>` we construct a `name` and `surgeries` element. The latter contains an ordered list of `<id>` elements, obtained by executing the inner `FROM` statement. Notice that in an IR tree an `ORDER BY` clause appears right above the `FROM` statement to which it applies.

Now we must turn this representation into a sequence of SQL queries, to issue the engine. In this case, we notice that there is an element constructor `<surgeries>` between the two `FROM` nodes. This means that we need to issue a separate SQL query corresponding to each `FROM` node:

```
Q1: SELECT p.pid, p.Name
```

```
    FROM Patient p
    ORDER BY p.pid
Q2: SELECT p.pid, s.sid
    FROM Patient p, Surgery s
    WHERE s.patient_id = p.pid
    ORDER BY p.pid, s.sid
```

Notice that in the second query we order by `p.pid` first, in order to allow the tagger to sort-merge the two tuple streams, and second by `s.sid`, to keep the surgeries in document order (which we assume here is given by `s.sid`).

Alternatively, the two queries can be combined into one left outer join:

```
SELECT DISTINCT p.pid, p.Name, s.sid
FROM Person p LEFT OUTER JOIN Surgery s
 ON p.pid = surgery.patient_id
ORDER BY p.pid, s.sid
```

Compositionality Consider now what happens if we apply another XQuery to IR1, the intermediate representation of the previous query. This corresponds to the XQuery expression below:

```
XQ2:
let $z := (for $x in /patient
           return
            <patient> <name>{ $x/name }</name>
                      <surgeries> {
                        for $y in $x/surgery
                        return <id> { $y/sid } </id>}
                      </surgeries>
            </patient>
return $z/patient/surgeries/sid
```

The inner query is computed first, then the XPath expression `surgeries/sid` is applied to the result. Applying the outer query is simulated by SilkRoute II on the IR of the inner query, which is `IR1` above. In this simple example we just need to evaluate the Axis expression directly on the intermediate representation. This "erases" the extra tags from `IR1`, resulting in:

```
IR2:
ORDER BY p.pid
FROM Patient p
   ORDER BY s.sid
   FROM Surgery s
   WHERE s.patient_id = p.pid
   <id> LEAF( s.sid ) </id>
```

Clearly a single SQL query can be issued here, by combining the two `FROM` nodes and the two `ORDER BY` nodes.

Order The IR allows fine-grained control of the output order through the `ORDER BY` nodes. These nodes should occur explicitly in the IR, in order to define both the document order and the to implement the `order by` clause in XQuery. For the document order we make the convention in SilkRoute II that the primary key of each table defines the document order for the canonical representation of that table. The `order by` clause in an XQuery expression overrides this, which translates into a new `ORDER BY` node in IR, above that corresponding to the document order.

An important point is that the `ORDER BY` clauses allow us to reason about the order, and use this during query composition. To illustrate this point consider the following modification to `XQ2`:

```
XQ3:
let $z := (for $x in /patient
           order by $x/age
           return
               <sugeries> {
                   for $y in $x/surgery
                   order by $y/date/text()
                   return $y
               </surgeries>}
for $u in $z/surgery
order by $u/room/text()
return $u/sid
```

The inner query results in the following IR:

```
IR3':
ORDER BY p.age, p.pid
FROM Patient p
      <surgeries>
          ORDER BY s.date, s.sid
          FROM Surgery s
          WHERE s.patient_id = p.pid
          <surgery> ...complex IR... </surgery>
      </surgeries>
```

Next, when the outer query is applied, SilkRoute II will remove the extra tags, resulting in a query similar to `IR2` and first consolidate the two `FROM` and `ORDER BY` clauses. This results in the following IR, equivalent to `IR2`:

```
ORDER BY p.age, p.pid, s.date, s.sid
FROM Surgery s, Patient p
WHERE s.patient_id = p.pid
LEAF s.sid
```

Now it can apply the `order by $u/room/text()` clause, which results in the final intermediate representation:

```
IR3:
ORDER BY s.room
ORDER BY p.age, p.pid, s.date, s.sid
FROM Surgery s, Patient p
WHERE s.patient_id = p.pid
LEAF s.sid
```

This is immediately translated to SQL:

```
SELECT s.sid
FROM Surgery s, Patient p
WHERE s.patient_id = p.pid
ORDER BY s.room, p.age, p.pid, s.date, s.sid
```

Heterogeneous Sequences A very simple illustration of a heterogeneous sequence is:

```
XQ4:
let $z := (for $x in //physician, //nurse)
for $x in $z
order by $x/name/text()
return $x
```

This returns a heterogeneous sequence of `physician` and `nurse` elements. The elements are interleaved, since the resulting sequence is sorted by `name`. This can be easily expressed in IR (to reduce clutter we assume redundant foreign key joins have been eliminated; see Sec. 5)

```
IR4:
ORDER BY x.name
   FROM physician
       ....IR for a physician...
   FROM nurse
       ....IR for a nurse...
```

5 Middleware Optimizations

5.1 Overview

Our middle-ware optimizer consists of three phases: IR rewriting, SQL generation, and SQL optimization. We describe each phase next. The optimizer uses a configuration file specifying some characteristics of the SQL engine, e.g. whether it supports nested queries, outer joins, etc, and uses these characteristics to guide in some of the optimization choices. If no information is available in the configuration file, then the optimizer assumes the worst (e.g. that the engine does not support nested queries).

5.2 Phase 1: IR Rewriting

In the first phase the optimizer does some simple rewritings on the IR: it consolidates `FROM` and `ORDER BY` clauses and pushes aggregates down whenever possible. The purpose is mostly to clean up the IR generated by the translator. Adjacent `FROM`, `WHERE`, and `ORDER BY` clauses are consolidated, making it easier for the SQL generator to navigate the IR tree.

5.3 Phase 2: SQL Generation

Here the optimizer chooses how to partition the IR tree in order to generate SQL queries, decides what computations to do in the middleware, and optimizes loop invariants. We illustrate the details next.

Partitioning The purpose of this step is to partition the IR tree into connected components, by removing some of the `xmltag` nodes (see Fig. 1). We illustrate this on the following query:

```
for $x in X
return
  <r> <aa> { for $y in Y
             where $y/@id = $x@id
             return $y/a } </aa>
      <bb> { for $z in z
             where $z/@id = $x/@id
             return $z/b } </bb>
  </r>
```

which results in the following IR:

```
ORDER BY x.xid
FROM X x
<r> <aa> ORDER BY y.yid
         FROM Y y
         WHERE y.xid = x.xid
         <a> LEAF y.a </a> </aa>
    <bb> ORDER BY Z.zid
         FROM Z
         WHERE z.xid = x.xid
         <b> LEAF z.b </b> </bb>
</r>
```

The intermediate representation can be partitioned in several ways. One is into three connected components, corresponding to the three `FROM` clauses. This results in three separate SQL queries:

```
Q1: SELECT x.xid
    FROM X x
    ORDER BY x.xid
Q2: SELECT x.xid, y.a
    FROM X x, Y y
    WHERE x.xid = y.xid
    ORDER BY x.xid, y.yid
Q3: SELECT x.xid, z.b
    FROM X x, Z z
    WHERE x.xid = z.xid
    ORDER BY x.xid, z.yid
```

A second is to partition it into two connected components: one with the first two `FROM` clauses, the other with the third `FROM` clause. This results in the following two SQL queries:

```
Q1': SELECT X.xid, Y.a
     FROM X LEFT OUTER JOIN Y ON Y.xid = X.xid
     ORDER BY X.xid, Y.yid
Q2': SELECT x.xid, z.b
     FROM X x, Z z
     WHERE x.xid = z.xid
     ORDER BY x.xid, z.yid
```

Another choice is to have a single partition, hence combine all three queries into one single outer join. This, however, generates a costly cross product between `Y` and `Z`.

If the SQL engine supports outer joins, then SilkRoute II greedily generates a partitioning that includes outer joins but avoids Cartesian products; if the SQL engine does not support outer joins then SilkRoute II generates the finest partition, with one SQL query for every possible connected component. The general partitioning problem has been studied in [6] and specific choices for outer-join and outer-union queries have been discussed in [19].

Computations in the middleware Consider the IR query `IR4`, from Sec. 4:

```
IR4:
ORDER BY x.name
   FROM physician
       ....IR for a physician...
   FROM nurse
       ....IR for a nurse...
```

This results in the following SQL query:

```
SELECT x.name
FROM (...SQL query for physician... UNION ALL
      ...SQL query for nurse...) as x
ORDER BY x.name
```

Here the SQL queries for physicians and nurses need to be padded with enough attributes (set to NULL) to make them union compatible. However, some SQL engines do not support nested subqueries in the FROM clause. An alternative is to issue the two SQL queries separately, each with an `ORDER BY name` clause; then merge the two tuple streams by `name` in the middleware. There are no more nested subqueries, and there is no need to make them union compatible, hence the two SQL queries are slightly simpler. Our heuristics for this rule is to apply it only when the underlying SQL engine does not support nested queries in the `FROM` statement (which, in our experiments, applied only to MySQL). In all other cases we generated the nested SQL query.

Loop Invariants Consider the following query, slightly modified from Sec. 2:

```
for $p in /patient,
    $s in $p/surgery[room=''ER45'']
return <surgery>
        { $s/photo,
          $p/imagingstudy[type=''MRI'']/name
        }
       </surgery>
```

The expression `$p/imagingstudy[type=''MRI'']/name` is independent of the surgery `$s` and will be executed unnecessarily once for every surgery of a given patient. If translated directly into SQL, the query generating the `name` subelements (which is one of the three SQL queries issued by the system, assuming no outer joins) would contain a cartesian product between the surgeries and the imagingstudies of the same patient.

The problem is that SQL does not have a `let` binding. Our solution is to add a `LET` clause to the IR, with the following syntax:

```
IR ::=  LET $var := IR1
        IN IR2
```

It defines an IR-level variable `$var` and binds it the result of `IR1`. The variable can be used anywhere in `IR2`, in a position of an IR subexpression. At runtime the tagger will first issue the SQL queries corresponding to `IR1` and construct the XML fragment, `XF`, then store it in memory. Next, it issues the SQL queries corresponding to `IR2` and inserts a copy of `XF` whenever possible. As we saw in Sec. 2, certain `order-by` clauses may force the optimizer to store multiple `XF` fragments, if they are requested in a different order from how they were generated: we avoid this by imposing restrictions on how `LET` statements commute with `ORDER BY` statements during the first phase of the optimization.

5.4 Phase 3: SQL Rewriting

This phase is the most extensive one consisting of several rewrite rules designed to optimize the often inefficient generated SQL queries. Our heuristic is to apply a rewriting if it decreases the cost of the query according to the optimizer [3]. If the necessary hook is not provided to access the optimizer, we attempt to make

[3]Sophisticated database engines allow us to calculate the estimated cost of a plan very quickly through SQL Queries

the query as flat as possible. Generally, only limited capability engines can not return costs and so this heuristic is usually an improvement.

Aggregate Flattening Optimizations Since XQuery has no explicit `group by` clause, queries computing aggregations and group-by are translated into inefficient SQL queries, which need to be optimized. Consider the following typical example, computing how many surgeries were done in each room:

```
let $r = distinct-values(/patient/surgery/room/text())
for $x in $r
let $v := /patient/surgery[room/text()=$x]
return <room> <name> { $x } </name>
              <cnt> { count($r) } </cnt>
       </room>
```

A direct translation to SQL is:

```
SELECT x.room, (SELECT count(*)
                FROM Surgery z
                WHERE x.room=z.room)
FROM (SELECT DISTINCT s.room FROM Surgery s) x
```

(We have eliminated here some redundant foreign-key joins.) The optimizer first flattens the nested `FROM` query then recognizes a group-by construct and rewrites to:

```
SELECT s.room, count(*)
FROM Surgery s
GROUP BY s.room
```

In more complex cases the SQL group-by requires an outer join, and determining when to use or when not to use an outer join requires a query containment check. We omit the details.

Flattening Existential Quantifiers We have shown in Sec. 2 how existential quantifier subqueries arise naturally in the translated SQL queries. Flattening such subqueries is similar to flattening queries with aggregates, and we omit further details (but refer the reader to the example in Sec. 2).

Combining both Rules Aggregate and quantifier flattening can be combined, and simplify quite complex queries. The following is a simple illustration.

```
SELECT COUNT(*)
FROM Patient P
WHERE EXISTS (SELECT * FROM Surgery S
              WHERE P.pid = S.pid)
```

becomes

```
SELECT COUNT(DISTINCT P.pid)
FROM Patient P, Surgery S
WHERE P.id = S.pid
```

Query Minimization The automatically generated SQL queries are sometimes redundant. Redundancy may already be present in the XQuery expression, such as in the trimming queries described in Sec. 2, or may be introduced by some optimization, e.g. as a result of flattening. Redundancy is eliminated through query minimization. We review here briefly the query minimization problem, and describe our algorithm. For more background we refer the reader to [1].

The query minimization problem is the following. Given a SQL query of the form:

```
SELECT DISTINCT v1, ..., vm
FROM R1 x1, R2 x2, ..., Rn xn
WHERE C1 and C2 and ... and Ck
```

where each condition `Ci` is either a selection condition (e.g. `x5.A = 'abc'`) or is a join condition (e.g. `x3.B = x9.C`). The query is called *minimal* if there exists no other equivalent query with fewer than `n` relations in the `FROM` clause. The query minimization problem is: given a SQL query as above, find another one that is equivalent and minimal.

For example the query:

```
Q1:
SELECT DISTINCT x.A
FROM R x, S y, R z
WHERE x.B = y.E and x.C = 9 and x.D = 'alpha'
  and z.B = y.E and z.D = 'alpha'
```

is minimized to

```
Q2:
SELECT DISTINCT u.A
FROM R u, S v
WHERE u.B = v.E and u.C = 9 and u.D = 'alpha'
```

In other words `Q1` and `Q2` are equivalent, and there is no other equivalent query with fewer than two tables in the `FROM` clause.

To minimize a query `Q1` one has to find a table in the `FROM` clause such that, denoting `Q2` is the query after eliminating that table, `Q1` is equivalent to `Q2`; then one proceeds by minimizing `Q2`. The crucial test is whether `Q2` is equivalent to `Q1`, and this is true iff there exists a homomorphism from the tuple variables of `Q1` to those of `Q2`. In our example, the homomorphism from `Q1` to `Q2` maps `x` to `u`, `y` to `v` and `z` to `u`.

Finding a homomorphism is, in theory, an NP-complete problem, and a naive algorithm would try to map every tuple variable `x` in `Q1` to every tuple variable `y` in `Q2`, resulting in an exponential running. However, the following simple heuristic, developed independently though similar to Gottlob [10], dramatically reduces the search space. Compute a set of pairs `C(x,y)` where `x` and `y` are tuple variables in `Q1` and `Q2` respectively: the meaning of `C(x,y)` is that `x` is

"compatible" with x. (1) initially all pairs of variables are in C (2) if x and y refer to two different table names, then remove (x,y) from C. (3) Repeat the following, until no more change: consider a pair (x,y) in C, and let x.A = x'.B be a join condition in Q1. Consider all tuple variables y' s.t. y.A = y'.B is a join condition in Q2: if none of the pairs (x',y') is in C then remove (x,y) from C. Once the table C has been computed it suffices to restrict the search to those homomorphisms that map every tuple variable x only to some y s.t. (x,y) is in C. While theoretically still exponential in the worst case, this improved algorithm runs much faster in practice. (For example it runs in linear time when all table names are distinct.) In all our experiments it always ran in under one second.

Position Predicates One feature of XQuery are position predicates. These are very easy to support by native XML engines, but very hard to support by XML publishing middleware. An example of such a query is:

```
for $p in /patient
where $x/surgery[1]/hospital_id
    = $x/surgery[last()]/hospital_id
return $p/name
```

This query finds all patients whose first and last recorded surgeries occurred at the same hospital. The translated SQL is very complex, as we have already suggested in Sec. 2.

Our solution is to add position information into the relational database. In our example we would modify the table Surgery and add two attributes, say p1 and p2 representing the ascending and descending position of that surgery element. The positioning is relative to the siblings, and is not global: for example all surgeries that are the first surgeries of their patient will have p1 = 1, all second surgeries will have p1 = 2 etc. Similarly last surgeries will have p2 = 1, before-last surgeries have p2 = 2 etc. The values of these attributes must be precomputed and stored in the relational database.

The question now, is how to inform SilkRoute II of the special meaning of the two attributes p1 and p2. For this we use the SilkRoute:Order function. This function allows the user to specify how to compute the ascending and descending orders for this particular portion of the view or query. The function is used in the public view definition, and in any derived view definition, as in the following example:

```
for $p in CanonicalView()/Patient/row
return {
  for $s in CanonicalView()/Surgery/row
  where $s/pid = $p/pid
  return SilkRoute:Order(
   $s/p1,
   $s/p2,
   $s
 }
```

Here `SilkRoute:Order(a,b,c)` returns the value `c`, but instructs the system that the attributes `a` and `b` represent the ascending position and descending position respectively.

Using this information during query rewriting is similar to rewriting queries using views, and here we deploy a simple heuristics based on pattern matching. In our example, the original query can be translated into the following SQL query:

```
SELECT DISTINCT P.pid, P.name
FROM Patient P, Surgery S1, Surgery S2
WHERE P.pid = S1.pid and P.pid = S2.pid
 AND S1.p1 = 1 AND S2.p2 = 1 AND S2.hid = S.hid
ORDER BY P.pid
```

Order of Optimizations The order optimization and position predicate optimization are performed in the unoptimized SQL. These optimizations are not cost based, they are applied whenever possible.

When the unoptimized SQL query enters the optimizer, it can appear highly nested. This nesting renders minimization useless, since minimization requires set oriented, flat queries. As a result, we apply our flattening rewrite techniques. These rewritings are applied with a least fixed point semantic. In order to support older engines, such as older version of MySQL, we prefer to flatten where clauses before select clauses. This is because we can easily partition nested selections into multiple queries but do not have support in the middleware for selections.

We only partition further at this phase if we are required because of older engines, we do not consider denesting the partition. After flattening, we apply minimization and execute the query.

6 Experiments

Our experimental validation has three goals. The first is to test the quality of the SQL queries generated by SilkRoute II, by measuring their efficiency and scalability. The second is to compare the XQuery to SQL translation approach to native XQuery processors. And the third is to test the effectiveness of the various optimizations described in the paper. In all three goals we sought a validation for a variety of XQuery workloads and a variety of relational and XQuery platforms.

XQuery workloads We experimented with three workloads(XMark, Human Brain Project (HBP), and TPC), and report here only the first two. XMark [17] has 20 XQueries that exercise features such as position predicates, joins, groupings and user-defined functions, and a synthetic XML generator: we generated data sets of 10MB, 100MB, and 500MB. To run SilkRoute II, we designed a relational schema for the XMark XML data, consisting of 15 tables, and shredded the XML data into these tables. Then, we wrote a public view to reconstruct the original XML data faithfully, including order. Since

Driver	UnixODBC	2.2.10
Relational database systems	MySQL	4.0.18
	PostgreSQL	7.4
	SQLServer	8.0
Native XQuery systems	Galax	4.0 (cvs)
	Saxon	8.1
	QizOpen	0.3
	Qexo	1.7
	QuiLogic IMDB	3.1

Figure 2: Software Versions Used

SilkRoute II applies to XML publishing scenarios with non-recursive schemata. We ignored the recursive part of the XML data and, hence, did not handle the XMark Queries 15 and 16.

The Human Brain Project [2] (HBP) is an interesting XML publishing application. The original data is in a relational database with 36 tables, and the Website offers an XQuery interface. The XML view is a complex, hierarchically structured data. There are 11 canned queries on the Website, and these are the queries reported in our paper. This workload has some interesting characteristics not present in XMark. It is characterized by a large number and varied mixture of structural and relational joins. Most queries are much more complex than the XMark queries. In order to run the native XML engines on this data set we materialized the XML public view, and obtained an XML document of 30MB.

Platforms We ran the SQL queries generated by SilkRoute II by connecting through UnixODBC to one of three relational databases: MySQL, PostgreSQL, and SQL Server. We ran XQueries natively on five native XML platforms: Galax (version 4.0 which adds a powerful optimizer), Saxon, QizOpen, Quexo, and QuiLogic IMDB. All five systems are main memory XQuery processors. Figure 2 contains all the version numbers.

System All experiments are on a Linux Fedora Core 2 machine with an SMP build. It has 512MB of RAM and a Pentium 4 2.8Ghz CPU with hyper-threading. All databases except SQL Server reside on the same machine. We connect to all databases are connected via UnixODBC. All tests are performed with only the required database and executable active. All experiments are with a cold cache.

Experimental Methodology We report end-to-end running time. This includes disk I/O for relational systems (since we ran with a cold cache) and parsing time for XML engines. MySQL query evaluation times are very sensitive to memory load: our numbers represent average runs of the system. All times are in seconds, and represent averages over several runs.

6.1 Quality of SQL Queries Generated by SilkRoute II

Figures 3,4,5, and 6 show the running times on the 10MB and 100MB XMark data. We grouped conceptually the XMark queries into three groups: the first group (queries 1,2,3) are XPath-like queries, the second group (queries 8,9,

11, 12) represent relational-style queries with joins, while the others have less crisp features and are not reported here. (Recall that SilkRoute II cannot run queries 15, 16). The SQL queries we generated performed extremely well, with occasional hiccups from MySQL, which performed poorly on query 3. On the 100MB data set MySQL also handled query 9 poorly, by choosing an inefficient plan for a join query. Still, MySQL could execute all queries, which was not the case for the native XQuery engines.

We also ran XMark on 500MB, but omit this graph since none of the native XML engines could scale up to 500MB.

Figure 7 shows the running time for HBP. These XQueries include some simple selection queries and a host of large and many times redundant joins. We present only the complex structural queries: 8,9,10, and 11. The others are either simple selections or structurally similar to the queries we present. These four complex queries ran under 3 seconds on SQL server, under 11 seconds on PostgreSQL and under 18 seconds on MySQL.

6.2 Comparing SilkRoute II with Native XQuery Engines

The same five figures also report the running times for the five native XQuery engines. Two remarks are in order. First, only one engine (QuiLogic's IMDB) could run on the 100MB XMark, and even then it could execute only some of the queries. Second, some of these engines to not support the full XQuery, and could not run some XMark queries or the HBP queries. This is somewhat ironic, since one expects the support for the full XQuery language to be one of the strengths of native XQuery engines, over a translated approach. Whenever a bar for an engine is missing, it means that it could not execute that query on that data instance, for one of the two reasons above.

When combined with SQL Server SilkRoute II outperformed *all* native XQuery engines, in *all* cases. On PostgreSQL, SilkRoute II also outperformed the native XQuery engines, with two exceptions, queries 10 and 11 in Fig. 7 where QizOpen edged ahead. On MySQL, SilkRoute II was generally better, but in a few cases MySQL ran significantly slower than the best XQuery engines: XMark queries 3 and 9 on 10MB, and XMark query 3 on 100MB, and HBP queries 9 and 10.

Overall, however, SilkRoute II outperformed significantly the native XQuery engines. We find this significant: we were expecting the XQuery engines to be more efficient on small data instances. On the 100MB XMark, some queries are executing in less than the time it takes for the native XQuery platforms to handle the 10MB document.

This, of course, is mostly due to the mature technology existing in relational database system. It also points, however, to the quality of the generated SQL queries.

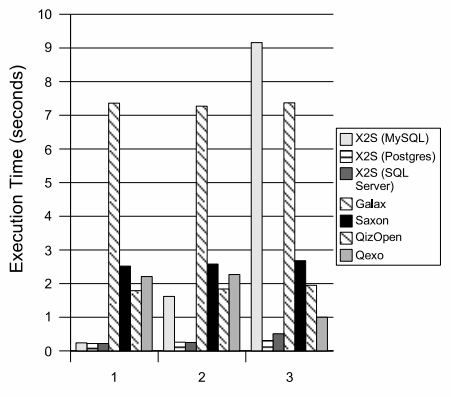

Figure 3: XPath Style (Xmark 10MB)

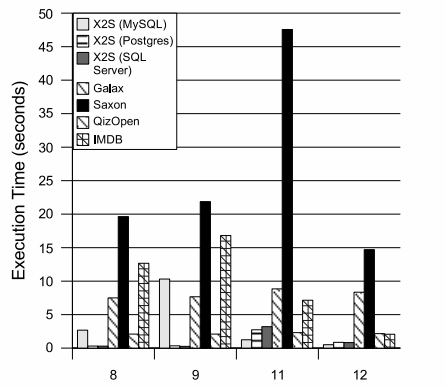

Figure 4: Relation-Style Joins (XMark 10MB)

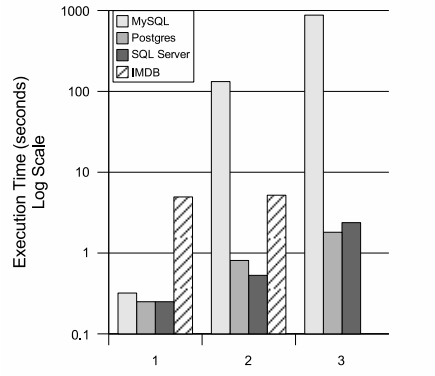

Figure 5: XPath Style (Xmark 100MB)

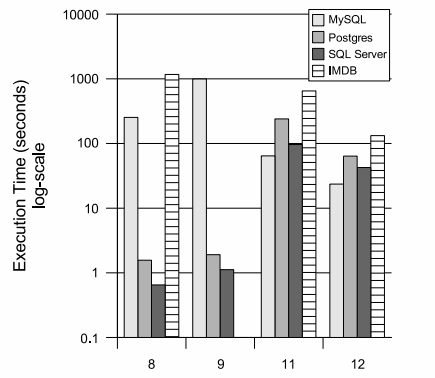

Figure 6: Relation-Style Joins (XMark 100MB)

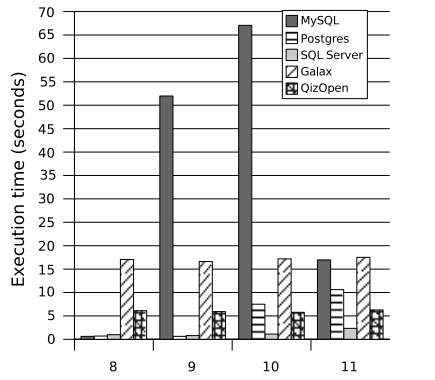

Figure 7: HBP Complex Queries

```
XMark Q11:
for $p in doc('Xmark.xml')/site/people/person
let $l := for $i in
  doc('Xmark.xml')/site/open_auctions/open_auction/initial
  where $p/profile/@income > (5000.0 * $i/text())
  return $i
return <items name="{ $p/name/text() }">
         { count($l) }
       </items>

Nested SQL Query:
SELECT person.id, (COUNT(*)
                   FROM open_auction
                   WHERE p.income > 5000*oa.initial)
FROM person
GROUP BY p.income, p.id
ORDER BY p.id

Flattened SQL Query:
SELECT person.id, COUNT(DISTINCT oa.id)
FROM person p LEFT OUTER JOIN open_auction oa
            ON p.income > 5000*oa.initial
GROUP BY p.income, p.id
ORDER BY p.id
```

Figure 8: XMark Query 11, the Nested and Flat SQL Translation

6.3 Effect of Individual Optimizations

Next we evaluate the effectiveness of some of the optimizations described in Sec. 5.

Aggregate Flattening Xmark queries 3, 8, and 11 were translated into highly nested SQL queries. The impact of flattening is as follows:

	XMark 3	XMark 8	XMark 11
PostgreSQL	1.80	1.62	240.75
without	1662.56	241.79	291.25
SQLServer	0.92	0.60	97.69 / 280.21 (*)
without	2.50	0.72	102.99

While PostgreSQL and SQL server support nested queries, they did benefit from flattening. MySQL could not handle nested SQL queries at all, so here flattening was the only way to make the queries run at all.

It turns out that the only case where applying flattening gave poor results was Query 11 on SQL Server. This is interesting, so we report the XQuery, the nested, and the flattened SQL queries in Fig. 8. Notice that there are no equality statements in the join condition. The SQL Server optimizer generated a better plan for the nested query. Because we can ask the SQL Server Optimizer, we are able to issue the better query plan[4]

Minimization Query minimization affected dramatically HBP query 10:

	HBP 10 With	HBP 10 With Out
MySQL	67.06	*
PostgreSQL	0.78	22.97
SQLServer	0.85	1.41

[4]This plan is sophisticated and not a direct analog of any SQL Query. This means that it would be difficult to coax this same performance out of the other engines

MySQL without minimization did not complete in an hour. Notice that all of the data sits in main memory. We see something interesting here, PostgreSQL also benefits from minimization, but less dramatically. SQL Server benefits the least. In this case it seems its ability to effectively handle memory has mollified the effect of poorly written queries.

Loop Invariants The only query affected by this optimization was HBP 10:

	HBP 10	without
MySQL	67.06	97.06
PostgreSQL	.78	1.02
SQLServer	.85	.92

Although more limited in scope in our workloads, this optimization can make a difference. We view it as a necessary part of a large repertoire of optimization techniques: when applicable, it eliminates an unnecessary cross-product; when not applicable, it has no impact.

Position Predicates This optimization dramatically affected XMark queries 2 and 3:

	Xmark 2 10MB	Xmark 3 10MB	Xmark 2 100MB	XMark 3 100MB
MySQL	1.62	9.16	132.37	878.76
without	161.29	DNR	*	DNR
PostgreSQL	0.26	0.30	1.84	1.97
without	16.26	392.93	*	*
SQLServer	0.27	0.29	0.53	0.92
without	1.21	8.92	3.76	233.96

DNR indicates the query could not be run without the optimization. A star indicates that it did not complete within an hour. We notice here that our predicate rewrites clearly have an effect. Again, more sophisticated optimizers, like SQL Server, can mollify the effect of difficult queries. Position queries should be hard for traditional SQL engines since order is not a fundamental part of the data model. Here we show that there do exist techniques to remedy some portions of the impedance mismatch.

6.4 Discussion

The experiments demonstrated that it is possible to generate high quality SQL queries automatically from XQueries, which scale to large data sets, on a variety of relational database platforms. They also demonstrate that, with today's technology, XQueries are significantly faster, and scale to significantly larger data sets, when translated to SQL and run on relational database engines than when executed on native XQuery engines. This is always the case with high performance commercial database systems (e.g. SQL Server), and it is true in most cases with less efficient open-source database systems (MySQL). Surprisingly, the experiments also demonstrate that our XQuery to SQL translator supports a richer set of the XQuery language than some of the native XQuery engines. Finally, the experiments demonstrate the effectiveness of the optimization techniques: in some cases these optimizations made it possible for queries to run at all, while in other cases it made them dramatically faster.

7 Related Work

There exists a rich literature on XML publishing and the related XML storage problem. An excellent survey of both problems is [12].

Two early XML publishing systems are XPeranto and SilkRoute. Xperanto reuses the DB2 infrastructure to execute queries on XML data [3, 19, 18]. SilkRoute focused on the query translation problem, and on optimizing the result construction [8, 6, 5]. Pushing XML updates to a relational engine in the context of XML publishing has been discussed in [20]. MARS [4] describes advanced translation techniques in the presence of redundant storage.

Besides XPeranto and SilkRoute there has been little work on optimizing XML publishing. A recent paper [13] discusses two optimizations: eliminating redundant joins on foreign keys and eliminating complete unions when they can be replaced with a single query.

Much more work has been done on optimizing XML storage, where the relational representation may be specifically modified to best support XML queries. We mention here only two recent pieces of work in [11] and [15].

8 Conclusions

In this paper we have shown that it is feasible to have a highly performant and largely complete translation of XQuery to SQL. In order to accomplish this, we have described a novel intermediate representation for relational-to-XML transformations, and have described a framework for optimizing the SQL to XQuery translation. Our experimental evaluation has found that the optimizations we develop make the translation of XQuery to SQL more competitive than today's native XQuery engines.

Acknowledgments This work was funded in part by NIH Human Brain Project grant DC02310 and Suciu was partially supported by the NSF CAREER Grant IIS-0092955, NSF Grants IIS-0140493, IIS-0205635, and IIS-0428168, and a gift from Microsoft.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison Wesley Publishing Co, 1995.

[2] Jim Brinkley. The XBrain project. `http://quad.biostr.washington.edu:8080/xbrain/index.jsp`.

[3] M. Carey, D. Florescu, Z. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, and S. subramanian. XPERANTO: publishing object-relational data as XML. In *Proceedings of WebDB*, Dallas, TX, May 2000.

[4] A. Deutsch and V. Tannen. MARS: a system for publishing XML from mixed and redundant storage. In *VLDB*, Berlin, Germany, September 2003.

[5] M. Fernandez, Y. Kadiyska, A. Morishima, D. Suciu, and W. Tan. SilkRoute : a framework for publishing relational data in XML. *ACM Transactions on Database Technology*, 27(4), December 2002.

[6] M. Fernandez, A. Morishima, and D. Suciu. Efficient evaluation of XML middle-ware queries. In *Proceedings of ACM SIGMOD Conference on Management of Data*, Santa Barbara, 2001.

[7] M. Fernandez and J. Simeon. Galax: the XQuery implementation for discriminating hackers, 2002. available from `http://db.bell-labs.com/galax/`.

[8] M. Fernandez, D. Suciu, and W. Tan. SilkRoute: trading between relations and XML. In *Proceedings of the WWW9*, pages 723–746, Amsterdam, 2000.

[9] T. Fiebig and G. Moerkotte. Algebraic XML construction and its optimization in natix. *Journal of the WWW*, 4(3):167–187, 2001.

[10] G. Gottlob, C. Koch, and K. Schulz. Conjunctive queries over trees. In *PODS*, pages 189–200, 2004.

[11] T. Grust, S. Sakr, and J. Teubner. XQuery on SQL hosts. In *VLDB*, 2004.

[12] R. Krishnamurthy, R. Kaushik, and J.F. Naughton. XML-to-SQL query translation literature: The state of the art and open problems. In *XSYM*, pages 1–18, 2003.

[13] R. Krishnamurthy, R. Kaushik, and J.F. Naughton. Efficient XML-to-SQL query translation: Where to add the intelligence? In *VLDB*, pages 144–155, 2004.

[14] P.E. O'Neil, E.J. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: Insert-friendly XML node labels. In *SIGMOD*, pages 903–908, 2004.

[15] S. Pal, I. Cseri, G. Schaller, O. Seeliger, L. Giakoumakis, and V.V. Zolotov. Indexing XML data stored in a relational database. In *VLDB*, pages 1134–1145, Toronto, 2004.

[16] S. Paparizos, Y. Wu L.V.S. Lakshmanan, and H. V. Jagadish. Tree logical classes for efficient evaluation of XQuery. In *SIGMOD*, pages 71–82, Paris, 2004.

[17] A. Schmidt, F. Waas, M. Kersten, D. Florescu, M. Carey, I. Manolescu, and R. Busse. Why and how to benchmark XML databases. *Sigmod Record*, 30(5), 2001.

[18] J. Shanmugasundaram, , J. Kiernana, E. Shekita, C. Fan, and J. Funderburk. Querying XML views of relational data. In *Proceedings of VLDB*, pages 261–270, Rome, Italy, September 2001.

[19] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently publishing relational data as XML documents. In *Proceedings of VLDB*, pages 65–76, Cairo, Egypt, September 2000.

[20] I. Tatarinov, S. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and querying ordered XML using a relational database system. In *SIGMOD*, May 2002.