Towards practical meta-querying ☆

Jan Van den Bussche[a], Stijn Vansummeren[a,1,*], Gottfried Vossen[b]

[a] *Limburgs Universitair Centrum, Universitaire Campus, B-3590 Diepenbeek, Belgium*
[b] *University of Muenster, D-48159 Muenster, Germany*

Abstract

We describe a meta-querying system for databases containing queries in addition to ordinary data. In the context of such databases, a meta-query is a query about queries. Representing stored queries in XML, and using the standard XML manipulation language XSLT as a sublanguage, we show that just a few features need to be added to SQL to turn it into a fully-fledged meta-query language. The good news is that these features can be directly supported by extensible database technology.
© 2004 Elsevier Ltd. All rights reserved.

Keywords: Databases; Meta-querying; Stored queries; XML; XSLT; Extensible database technology

1. Introduction

Enterprise databases often contain not only ordinary data, but also queries. Examples are view definitions in the system catalog; usage logs or workloads; and stored procedures as in SQL/PSM or Postgres [1]. Unfortunately, these queries are typically stored as long strings, which makes it hard to use standard SQL to express *meta-queries:* queries about queries. Meta-querying is an important activity in situations such as advanced database administration, database usage monitoring, and workload analysis. Examples of meta-queries to a usage log are:

(1) Which queries in the log do the most joins?
(2) Which queries in the log return an empty answer on the current instance of the database?
(3) View expansion: replace, in each query in the log, each view name by its definition as given in the system catalog.
(4) Given a list of new view definitions (under the old names), which queries in the log give a different answer on the current instance under the new view definitions?

Query 1 is *syntactical:* it only queries the stored queries on the basis of their expressions. Query 2 is *semantical:* its answer depends on the results of dynamically executing the stored queries. Query 3

is again syntactical, but more so than query 1 in that it also performs syntactical transformations. Query 4 is syntactical and semantical together.

To express meta-queries, database administrators and other advanced users typically resort to a programming language like Perl, in combination with Dynamic SQL. It would be much nicer if they would not have to "leave" the database system and could express their meta-queries directly in Interactive SQL. Indeed, already in 1993, in his SIGMOD Innovations Award speech, Jim Gray urged the database community to lower the wall between data and programs. In the same vein, the Asilomar Report puts the unification between programs and data high on the database research agenda [2]. As queries are an important kind of program in the context of databases, support for meta-querying thus seems to be a step in the right direction towards understanding how we can unify programs and data in database systems.

In this paper, we present a practical meta-querying system based on the relational model. Our main design goal was to use current DBMS technology and only extend standard SQL with specific meta-querying features where necessary. Stored queries are represented as syntax trees in XML format. This representation provides a convenient basis for syntactical meta-querying. Indeed, rather than reinventing the wheel and designing a new sublanguage for syntactical manipulation of stored queries, it allows us to use the standard XML transformation language XSLT for this purpose. Many syntactical meta-queries can then directly be expressed simply by allowing XSLT function calls within SQL expressions.[2]

This combination of SQL and XSLT gives us a basic level of expressive power, but for more complex syntactical meta-queries we need a bit more. To this end, we enrich the SQL language with *XML variables* which come in addition to

SQL's standard range variables. XML variables range not over the rows of a table, but rather over the subelements of an XML tree. The range can be narrowed by an XPath expression. (XPath is the sublanguage of XSLT used to locate subelements of XML documents.) XML variables thus allow us to go from an XML document to a set of XML documents. Conversely, we also add *XML aggregation* [4], which allows us to go from a set of XML documents to a single one.

SQL combined with XSLT and enriched with XML variables and aggregation offers all the expressive power one needs for ad-hoc syntactical meta-querying. To allow for semantical meta-querying as well, it now suffices to add an *evaluation function*, taking the syntax tree of some query as input, and producing the table resulting from executing the query as output. We note that a similar evaluation feature was already present in the Postgres system.

What we obtain is *Meta-SQL*: a practical meta-query language. Meta-SQL has as advantage that it is *not* "yet another query language": it is entirely compatible with modern SQL implementations offered by contemporary extensible database systems. Indeed, these systems already support calls to external functions from within SQL expressions, which allows us to implement the XSLT calls. Furthermore, XML variables and the evaluation function can be implemented using *set-valued* external functions. As we will show, the powerful feature of "lateral derived tables", part of the SQL:1999 standard, turns out to be crucial to make this work. XML aggregation, finally, can be implemented as a user-defined aggregate function.

We emphasize again that we are not proposing yet another database language. Instead, our main design goal was to stick as closely as possible to standard SQL. Of course, a drastic alternative is to abandon the relational model altogether and move to, e.g., an XML-XQuery environment, where meta-querying does not pose any problem. However, given the widespread use of relational databases, a conservative approach such as ours remains important.

This paper is further organized as follows. In Section 2, we combine SQL with XSLT. In

---

[2] We embrace XSLT because it is the most popular and stable standard general-purpose XML manipulation language to date. When other languages, notably XQuery [3], will take over this role, it will be an easy matter to substitute XSLT by XQuery in our overall approach.

Section 3, we add XML variables. In Section 4, we move on to semantical meta-querying. In Section 5, we describe how Meta-SQL can be implemented using extensible database technology. We give some experimental performance results of our prototype in Section 6. In Section 7, we conclude with a discussion of our approach.

2. SQL + XSLT

Consider a standard relational database, except that in a table some columns can be marked to be of type "XML". In any row of that table, the component corresponding to a column of type XML holds an XML document. At the present conceptual level, we do not yet care about how this is implemented.

XSLT [5] is a widely used manipulation language for XML documents. An XSLT program takes an XML document as input, and produces as output another XML document (which could be in degenerate form, holding just a scalar value like a number or a string). Using the XSLT top-level parameter binding mechanism [5], programs can also take additional parameters as input.[3]

Hence, to query databases containing XML, it seems natural to extend SQL by allowing calls to XSLT functions, in the same way as extensible database systems extend SQL with calls to external functions. However, in these systems, external functions have to be precompiled and registered before they can be used. In Meta-SQL, the programmer merely includes the source of the needed XSLT functions and can then call them directly.

Let us see an example of all this, at the same time applying it to meta-querying. Consider a simplified system catalog table, called `Views`, containing view definitions. There is a column `name` of type string, holding the view name, and a column `def` of type XML, holding the syntax tree of the SQL query defining the view, in XML format. For example, over a movies database,

suppose we have a view `DirRatings` defined as follows:

```
select director, avg(rating) as avgrat
from Movies group by director
```
Then table `Views` would have a row with value for `name` equal to 'DirRatings', and value for `def` equal to the following XML document:

```
<query>
 <select>
  <sel-item> <column>director</column>
  </sel-item>
  <sel-item>
   <aggregate>
       <avg/>
       <column-ref> <column>rating
       </column> </column-ref>
   </aggregate>
       <alias>avgrat</alias>
  </sel-item>
 </select>
 <from>
  <table-ref> <table>Movies</table>
  </table-ref>
 </from>
 <group-by>
  <column-ref> <column>director</column>
  </column-ref>
 </group-by>
```

Fig. 1 shows the same document as a DOM tree [8], which is perhaps clearer.

To achieve uniformity in the specific XML format for representing SQL syntax trees, we must agree on some fixed BNF syntax for SQL. A BNF grammar can be easily transformed into an XML DTD [9], which then specifies the XML format to
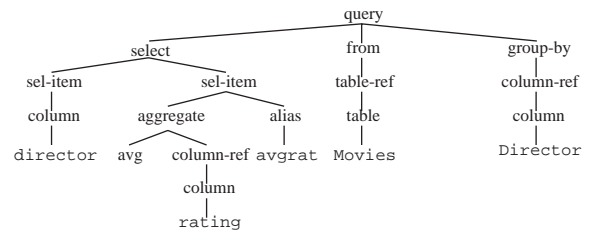

Fig. 1. Syntax tree of SQL query `select director, avg(rat-ing) as avgrat from Movies group by director`.

---

[3] In this paper, we cannot include a tutorial on XSLT, for which we refer to the Web or to the literature [6,7].

use. In this paper, we use the BNF grammar given by Date [10]. The derived DTD is given in Appendix A.

Now recall the first example meta-query from the Introduction, but applied to our `Views` table: "which queries do the most joins?" For simplicity of exposition, let us identify the number of joins an SQL query does with the number of table names occurring in it. To express this meta-query in Meta-SQL, we write an auxiliary XSLT function `count_tables`, followed by an obvious SQL query calling this function:

```
function count_tables returns number
begin
  <xsl:template match="/">
    <xsl:value-of select="count(//table)"/>
  </xsl:template>
end

select name from Views
where
  count_tables(def) = (select max
  (count_tables(def)) from Views)
```

The first line in the Meta-SQL program above declares the XSLT function in Meta-SQL; between `begin` and `end` one writes arbitrary XSLT code. Of course, in general, Meta-SQL allows multiple XSLT functions to be declared and called in the SQL query that follows the function declarations.

XSLT programs execute by applying *templates*, starting at the root of the input document. Every template is associated with an XPath expression, which specifies at what nodes in the input document the template should be executed. The XPath expression/associated with the only template in the `count_tables` function above specifies that the template should be executed when we are at the root of the input document. At that point, we compute the set of all `table` subtrees by the `//table` XPath expression, and compute the cardinality of this set by the `count` function.

As a second example, suppose we are given a list `Removed` of names of tables that are going to be removed, and we want to know which views will become invalid after this removal because they mention one of these table names. To express this meta-query in Meta-SQL, we write:

```
function mentions_table param tname string
returns string
begin
  <xsl:param name="tname"/>

  <xsl:template match=" /" >
  <xsl:if test="//table[string(.)=$tname]" >
          true
          </xsl:if>
  </xsl:template>
end

select name from Views, Removed
where mentions_table(def, Removed.name) =
'true'
```

Here function `mentions_table` takes, in addition to the input tree, an extra parameter `tname` of type string. The line `<xsl:param name="tname"/>` brings this parameter inside the execution environment of the XSTL program. Again, our only template is executed when we are at the root of the input document. At that point we compute the set of all `table` subtrees whose string value equals the `tname` parameter by the XPath expression `//table[string(.)=$tname]`.[4] If this set is non-empty then we output the string `true`. Otherwise we do not generate any output, implicitly returning the empty string.

As a final example, suppose we are given a second view definitions table `Views2`, and for every view name that is listed in both views tables, we want a new definition that equals the union of the first definition and the second definition. To express this meta-query in Meta-SQL, we write:

```
function unite param v2 xml returns xml
begin
  <xsl:param name="v2"/>

  <xsl:template match=" /" >
    <union>
```

---

[4] Parameters in XSLT are accessed by prepending them with a $.

```
  <xsl:apply-templates/>
  <xsl:apply-templates select="$v2"/>
 </union>
</xsl:template>

<xsl:template match=" *" >
 <xsl:copy> <xsl:apply-templates/>
 </xsl:copy>
</xsl:template>
end

select name, unite(v.def,v2.def)
from Views v, Views2 v2
where v.name=v2.name
```

Here, the function `unite` returns XML and takes, in addition to the input tree, an extra parameter `v2` of type XML. We have two templates. The first template is executed at the root of the input document, where we construct a new document whose root is labeled with `union`. The children of this root are computed by first processing the rest of the input document, and then processing the document found in `v2`. The second template is executed at every node, where we copy that node, and recursively process all its children, effectively copying the entire tree of the node under inspection to the output.

3. XML variables and XML aggregation

The simple combination SQL + XSLT is already quite useful, but its full potential is only realized when we add a language construct that allows us to extract the subelements of an XML document. For example, the simple meta-query

give all pairs (v, t), where v is a view name and t is a table name mentioned in the definition of view v

is otherwise not expressible.

We therefore add *XML variables* to SQL: while the standard SQL range variables range over the rows of a table, XML variables range over the subelements of an XML document. Like range variables, XML variables are bound in the from-clause, in a similar way variables are bound in OQL [11] and in XQuery [3]. More specifically, an XML variable x is bound in a from-clause using a construct of the following form:

x in $y[e]$.

Here,

- y is a previously bound XML variable or a column reference, or an XSLT function call, of type XML; and
- e is an XPath expression [12] specifying which subelements of y we want x to range over.

A bound XML variable can appear in an SQL expression anywhere a column reference can.

For example, the meta-query quoted above can now be expressed as follows:

```
function string_value returns string
begin
end

select v.name, string_value(x)
from Views v, x in v.def[//table]
```

Note that the body of function `string_value` is empty; indeed, the empty XSLT program does exactly what we want here, namely, to return the string value of an XML document (in this case, a `table` subelement).

As another example, suppose we are given a log table `Log` with stored queries (in a column `Q`), and we want to identify "hot spots": subqueries that occur as a subquery in at least ten different queries. To express this meta-query, we write:

```
select s
from Log l, s in l.Q[//query]
group by s
having count(l.Q) >= 10
```

XML aggregation: XML variables allow us to go from an XML document to a set of XML documents. Conversely, we want to be able to go from a set of XML documents to a single one. Thereto, we add a natural aggregate function on XML columns, called CMB (for "combine"), also used by Shanmugasundaram et al. [4]. Just like the

result of the standard aggregate function SUM on a list of numbers $x_1, \ldots, x_n$ is the sum $x_1 + \cdots + x_n$, the result of CMB on a list of XML documents $d_1, \ldots, d_n$ is the combined XML document
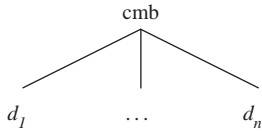
So, the top-level element of $CMB(d_1, \ldots, d_n)$ is always labeled 'cmb', and has the documents $d_1, \ldots, d_n$ as its subelements.[5]

As an example, suppose we are given a view definitions table `Views3` similar to `Views`, except that the same view name may appear with multiple definitions. Suppose we want to ask the following meta-query: for each view name, give the Cartesian product of its definitions. Thereto, we first write an easy XSLT function `cartprod` (see Appendix B) that transforms an input of the form $CMB(d_1, \ldots, d_n)$ into the document
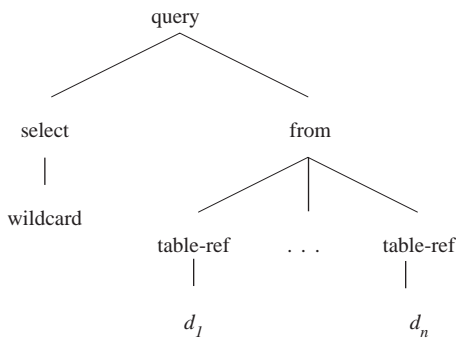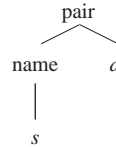

We then write:

```
select name, cartprod(CMB(def))
from Views3 group by name
```

As another example, recall the third example meta-query from the Introduction: view expansion in the log. To express this meta-query in Meta-SQL, we first write two auxiliary XSLT functions (see Appendix B).

---

[5] CMB is not commutative, so in the outcome of an XML aggregation, the particular order of the grouped documents is undetermined and will be implementation-dependent. Shanmugasundaram et al. [4] consider an ordered version of CMB.

- `pair`, taking a string parameter s, and transforming an input document d into the document

- `rewrite`, taking an XML parameter p of the form
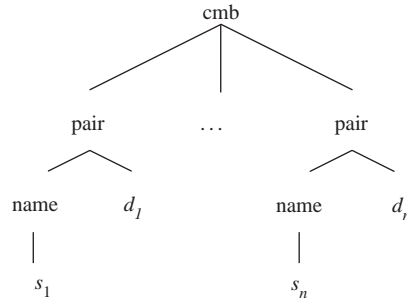
where $s_1, \ldots, s_n$ are different strings and $d_1, \ldots, d_n$ are arbitrary XML documents, and transforming an input query q by replacing every occurrence of an element of the form

by a copy of d_i.

We then write:

```
select rewrite(l.Q, (select CMB(pair(def,
name)) from Views)) from Log l
```

4. Semantic meta-querying

The language we have so far: SQL combined with XSLT, and enriched with XML variables and XML aggregation, gives us all the power we need for ad-hoc *syntactical* meta-querying. We now complete Meta-SQL so as to allow *semantical* meta-querying as well.

To this end, we add a function EVAL for dynamic evaluation of SQL queries. EVAL takes an SQL query (more correctly, its syntax tree in XML format) as input, and returns the table

resulting from executing the query. A call to EVAL can appear in an SQL expression anywhere a table reference can; the resulting table can thus be ranged over by a standard range variable.

As an example, suppose we are given a table `Customer` with two attributes: `custid` of type string, and `query` of type XML. The table holds queries asked by customers to the catalogue of a store. Every query returns a table with attributes `item`, `price`, and possibly others. The following meta-query shows for every customer the maximum price of items he requested:

```
select custid, max(t.price)
from Customer c, EVAL(c.query) t
group by custid
```

EVAL is all we need, provided we have enough information about the output scheme of the stored queries we are evaluating. For example, in the previous meta-query, we are only interested in the `price` attribute, and we know that every stored query evaluates to a table that indeed has a `price` column. But what if we are given an arbitrary collection of stored queries without information about their output schemes? They could even all have different output schemes!

Such a situation neatly fits the genre known as "semistructured data" [13]: data that has a structure (scheme), but we do not know it, and it can be irregular. Since XML is the standard format for semistructured data, and since we already have XML variables in Meta-SQL, we can easily solve the problem with an untyped version of EVAL. This UEVAL function works just like EVAL, except that the table resulting from the dynamic evaluation of the query is presented as a set of XML documents.

More concretely, suppose a particular stored query evaluates to a table with attributes (A, B, C). Then every output row (a, b, c) is represented as the XML document
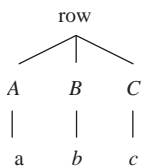
The resulting set of XML documents is ranged over by an XML variable rather than a standard range variable; this is a new use we make of XML variables in the Meta-SQL language.

As a simple example, recall the second example meta-query from the Introduction: "which queries in the log return an empty answer?" To express this meta-query, we write:

```
select Q from Log l
where not exists
  (select x from x in UEVAL(l.Q))
```

Note that x is an XML variable, whereas t in the previous example is a standard SQL range variable.

The fourth example meta-query from the Introduction (query comparison after view expansion) is equally easy to express, given that we already saw how to express view expansion in the previous section. In particular, note that we can apply EVAL and UEVAL not only to queries directly stored in the database, but also to queries coming from a syntactical meta-subquery (such as view expansion).

As a last example, the meta-query: "for each query in the log whose answer includes a `price` column, give the maximum of that column" is expressed as follows:

```
function get_price returns number
begin
  <xsl:template match=" /" >
   <xsl:value-of select="/row/price"/ >
  </xsl:template>
end

function has_price returns string
begin
  <xsl:template match=" /" >
   <xsl:if test="/row/price" >
   true
   </xsl:if>
  </xsl:template>
end

select l.Q, max(get_price(x))
from Log l, x in UEVAL(l.Q)
```

```
where has_price(x) = 'true'
group by l.Q
```

Run-time errors: Both EVAL and UEVAL expect their input to be (the XML syntax tree of) a correct SQL query. If that is not the case, we consider this to be a programming error, and a run-time exception will be raised. Moreover an application of EVAL can still fail even if its argument is a correct SQL query, namely, when the query result does not have the expected columns.

Designing a meta-query language where dynamic evaluation of stored queries is statically typed so as to be safe from such run-time errors is possible [14], but leads to an overly restricted formalism. In the design of Meta-SQL, we have opted to prefer flexibility and expressive power above static typing.

5. Implementation

Meta-SQL is entirely compatible with modern SQL implementations offered by contemporary extensible database systems. Extensible (also called object-relational, or universal) database systems [15] support user-defined data types for the columns of tables, and allow user-defined functions on these types to be called within SQL.

Extensibility is now part of the new SQL:1999 standard, and the major commercial vendors are aggressively moving to support it.

In Fig. 2 we illustrate the architecture of an implementation of Meta-SQL, explained in more detail in this section.

Implementing XML columns: To support XML columns, it suffices to define a data type 'XML'. We could derive this type from the built-in type CLOB (Character Large Object) and store XML documents as texts, but we could also derive from BLOB (Binary Large Object) and store XML documents as binary encodings of their DOM tree structures.

Implementing XSLT calls: Starting from Meta-SQL source code, consisting of a number of XSLT functions, followed by an SQL query using these functions, the Meta-SQL compiler does the following automatically:

(1) For each XSLT function, it generates a wrapper function (in an external programming language such as C or Java) that invokes an XSLT processor, thus performing the required XSLT transformation on the arguments, and returns the result. If necessary, this result is converted to an SQL datatype like number or string.
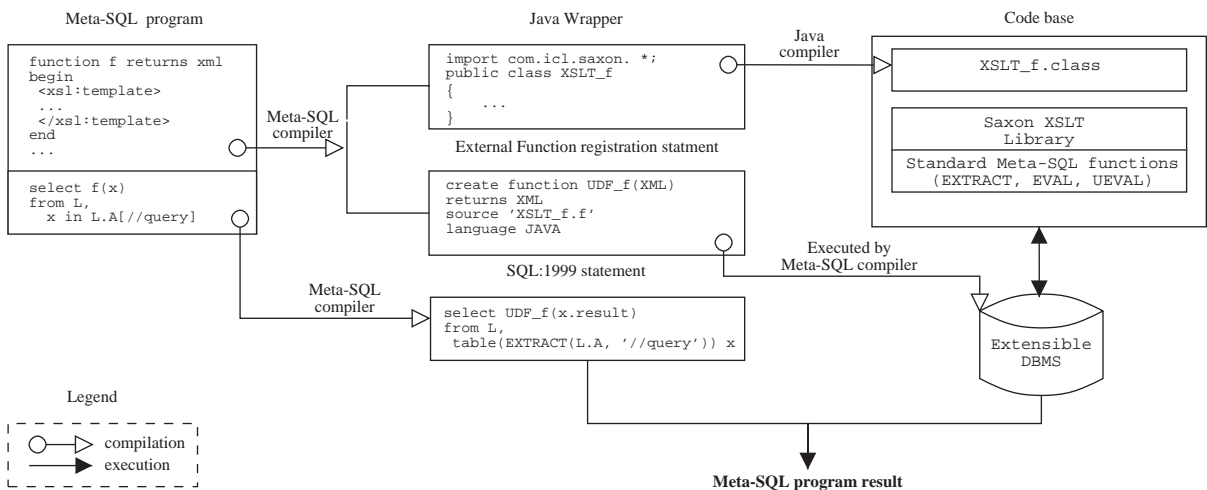(2) All wrapper functions are compiled and put together in an object library.

Fig. 2. Implementing Meta-SQL.

(3) Every wrapper function is registered in the database system (using SQL:1999 CREATE FUNCTION statements).

(4) With the functions in place, the final Meta-SQL query can now be executed literally by the database system.

Almost all available XSLT processors provide an interface to various programming languages, and can be loaded together with the application that invokes them. Hence, the compiled wrapper functions can be loaded into the database server together with the XSLT processor functionality and executed as efficiently as possible.

Implementing XML variables: Now the first and the last step in the above plan become more involved: in the first step, we must generate an additional function EXTRACT; in the last step, we must perform some rewriting on the SQL query. We next explain this in a bit more detail.

To support the XSLT calls, we needed only *single-valued* external functions: they return as output only a single value, be it a string, a number, or an XML document. To support XML variables, however, we need a *set-valued* external function. Specifically, the Meta-SQL system provides a function EXTRACT, which takes an XML document y and an XPath expression e as input, and which returns the set of all subelements of y that satisfy e. This set is returned as a one-column table with attribute name result.

Every XML variable binding, say, *x in y*[*e*], is now rewritten by the Meta-SQL compiler into a call EXTRACT(*y*,'*e*'). This call returns a table to which x is again bound, but now x has become a standard SQL range variable over the single attribute result.

For example, recall the first example of Section 3:

```
select v.name, string_value(x)
from Views v, x in v.def[//table]
```

This query will be rewritten as follows:

```
select v.name, string_value(x.result)
from Views v, table(EXTRACT(v.def,'//
table')) x
```

The above from-clause contains two table expressions. Note that the variable ranging over the first table expression, namely v, is directly used in the second table expression. This is an instance of what SQL:1999 calls a "lateral derived table" [16]. Such lateral joins were not allowed in SQL-92; we see that they are crucial here. We point out that they were present in OQL from the outset [11].

Implementing EVAL and UEVAL: The function UEVAL can, like EXTRACT, be realized as a set-valued external function. This evaluation function takes an SQL syntax tree in XML as input; unparses it; sends the SQL query to the database; receives the answer rows; transforms them into XML as explained in the previous section; and returns the results. The XML variable ranging over the UEVAL result is handled in the same way as above.

For example,

```
select Q from Log l
where not exists
  (select x from x in UEVAL(l.Q))
```

is compiled into

```
select Q from Log l
where not exists
  (select x.result from table(UEVAL(l.Q)) x)
```

The implementation of EVAL is a bit more complicated, because EVAL returns not a set of XML documents, but a normal SQL table ranged over by a standard range variable. In this case, the Meta-SQL compiler first determines the specific attributes that are mentioned in connection with this range variable. A specific table-valued external function having this set of attributes as output scheme is then generated and registered in the database system. This evaluation function will actually not send its exact argument SQL query to the database for evaluation, but rather its projection on the output scheme in question.

For example, recall the first example of Section 4:

```
select custid, max(t.price)
from Customer c, EVAL(c.query) t
group by custid
```

To implement this query, the Meta-SQL system will generate an evaluation function, say, EVAL_1, with output scheme (`price`), and will rewrite the query to

```
select custid, max(t.price)
from Customer c, table(EVAL_1(c.query)) t
group by custid
```

Here, $EVAL\_1(q)$, when called on any stored query q, evaluates and returns the projection $\pi_{\texttt{price}}(q)$.

If the select-clause would have additionally included `sum(t.qty)`, then the system would have generated a different evaluation function EVAL_2 with output scheme (`price,qty`) and according behavior.

Implementing XML aggregation: The XML aggregate function CMB introduced in Section 3 can be directly provided as a user-defined aggregate function.

A Working Prototype: We have developed a prototype implementation [17] on top of DB2 UDB [18], which we chose because it is freely available to university research and teaching.

We implement the external XSLT wrapper functions in Java, in conjunction with the popular and free Java-based XSLT processor SAXON [19]. SAXON also provides a convenient Java-XML interface making the EXTRACT function very easy to write.[6]

Currently, DB2 only supports the extension of existing aggregation operators, and not the definition of new ones. Therefore, we were unable to implement the CMB aggregation operator.

6. Experimental performance evaluation

In this section we describe some performance experiments on our prototype implementation. Unless stated otherwise, the results shown are

---

[6] IBM provides an 'XML Extender' to DB2 UDB which already provides an XML data type (derived from CLOB as we do), but with insufficient functionality for our needs. For example, there is also an Extract function, but it is much weaker than the EXTRACT function we need to implement our mechanism of XML variables.

averages of multiple executions of the test under discussion.

Java Overhead: Our first test measures the overhead implied by calling external functions written in Java. To that cause we created three simple single-valued external functions, CST, MUL and CAP, which respectively return a constant number, multiply a number with a constant, and transform the input string to capital letters. The following four queries were executed:

```
select * from T
select CST(A), B from T
select MUL(A), B from T
select A, CAP(B) from T
```

Here, T consists of an integer column A and a character column B. The first query is executed to measure the time needed to select tuples from T without calling an external function.

Fig. 3 shows the results for varying table sizes. As was to be expected, the running times of all functions grow linearly in the number of input tuples. Since the time needed to execute the last three queries closely resembles that of the first, we may conclude that there is a minimal overhead involved with external functions programmed in Java.

XSLT Processor overhead: Our prototype implements XSLT functions by external wrapper functions calling the SAXON XSLT processor. To
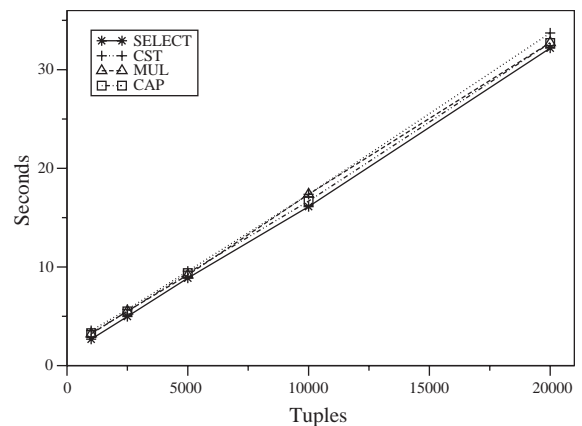

Fig. 3. Performance of external functions in Java.

measure the startup cost of the XSLT processor, we compared the Meta-SQL query

```
function XSLT_CST returns number
begin
  <xsl:template match=" /" >
  40513
  </xsl:template>
end
```

```
select XSLT_CST(A) from T
```

whose execution time is dominated by starting up the XSLT processor, with the query

```
select CST(A) from T
```

Here, CST is single-valued external function which also returns the same constant on all inputs. Table T consists of a single XML column, whose values are single-node trees.

Fig. 4 shows the results for varying table sizes. The XSLT function running time is significantly larger than that of CST, and grows linearly in the number of input tuples, which indicates that there is a constant startup cost imposed by the XSLT processor. As such, a Meta-SQL implementation would greatly benefit from more efficient XSLT/XML processors.

A valid question to ask next is how the size of the input XML document affects the running time
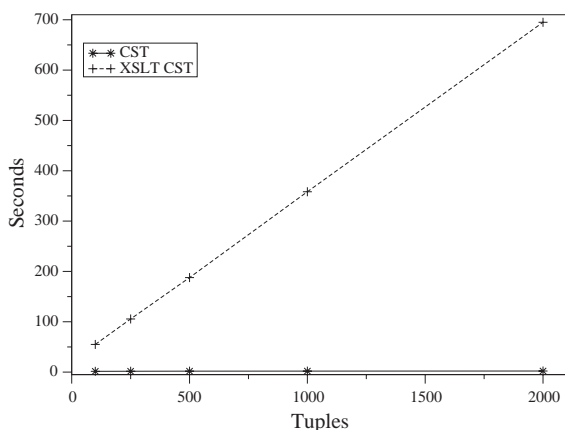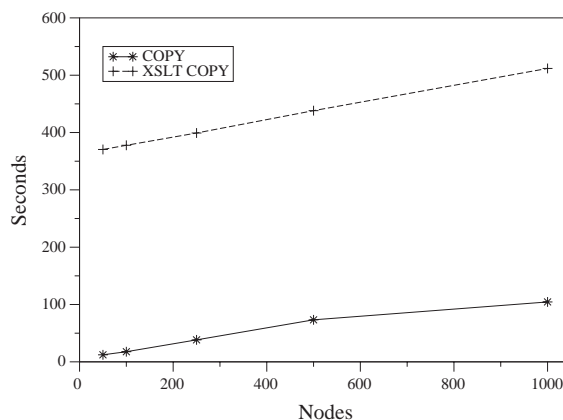

Fig. 5. Performance of XSLT calls.

of an XSLT function. To this cause we compared the following two queries:

```
select XSLT_COPY(A) from T1
select COPY(A) from T1
```

Here, both XSLT_COPY and COPY copy their input to the output. The only difference is that XSLT function XSLT_COPY does so by XSLT template matching whereas external function COPY performs a true copy. Table T1 consists of a single XML column A.

Fig. 5 shows the running times of these queries on tables with 1000 documents, for varying document sizes. Both running times grow linearly when the number of nodes per tuple increases, as was to be expected. Combined with the previous results, this indicates that calling an XSLT function has a rather large start-up cost, but a relatively small execution cost.

Extract: As described in the Section 5, XML variables are implemented by a set-valued external function called EXTRACT. To see how set-valued external functions compare with single valued functions, we created a single-valued function named SCALAR_EXTRACT, which evaluates the XPath expression/on its input, and executed the following three queries:

```
select x from T, x in B[//*]
select x from T1, x in B[/]
select SCALAR_EXTRACT(B) from T1
```


Fig. 4. Performance of XSLT calls.

Here, `T` consists of an XML column `B`, which is populated by 1000 documents of m nodes. Table `T1` contains the result of the first query. Hence, the second and third query return the same output as the first; here, the second query still uses an XML variable, whereas the third query makes a direct function call. The difference is that `EXTRACT` is called 1000 times in the first query, each time returning m tuples, whereas in the second and third query `EXTRACT` and `SCALAR_EXTRACT` are called $m \times 1000$ times, each call returning a single tuple. Thus, this test compares the overheads involved with set-valued functions returning multiple tuples, set-valued functions returning a single tuple, and single-valued functions returning a single tuple.

We timed the execution for varying m, as shown in Fig. 6. Due to the large running times, the test was only timed once.

We can safely conclude that returning multiple tuples from a set-valued function is not a problem. Indeed, the running time of the first query is quite reasonable and increases linearly with m (although it appears constant in Fig. 6 due to the large running times of the other queries). In particular, it is multiple times faster than returning m times a single tuple. As such, the implementation of XML variables in our approach performs very reasonably.

Since both `EXTRACT` and `SCALAR_EXTRACT` use the XSLT processor to evaluate their XPATH

expressions and because the running time of the second query resembles that of the third, we can also conclude that the overhead of a set-valued external function is about the same as that of a single-valued one.

With the following two queries, we measure the overhead of an `EXTRACT` call versus the actual amount of work that needs to be done:

```
select x from T, x in B[//*]
select C from T, T2
```

Here, `T` is as before and `T2` is the table with XML column `C`, containing the result of the first query on a single document in `T`. Since all documents in `T` are the same, both queries return the same result. By adding the time the XSLT processor needs to evaluate the XPath expression `//*` 1000 times to the timing of the second query, we get an estimate of the time needed to calculate the result of the first query, without actually calling `EXTRACT`.

As can be seen from Fig. 7, the running time of both queries grow linearly in their input. However, the first query outperforms the second one when m grows larger. Hence, although `EXTRACT` has some startup overhead, it is efficient when applied to large documents. Indeed, it even outperforms a setting in which the same amount of work needs to be done, but no call to a set-valued external function is made. Combined with our previous
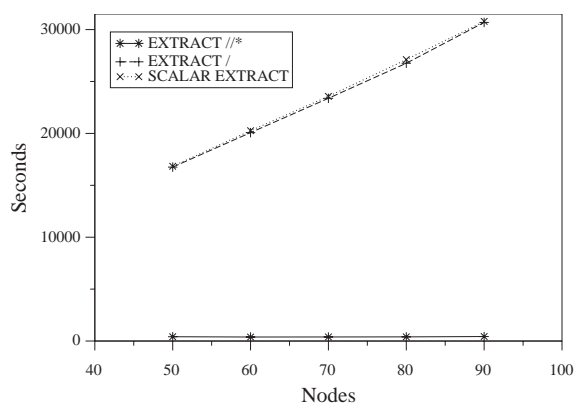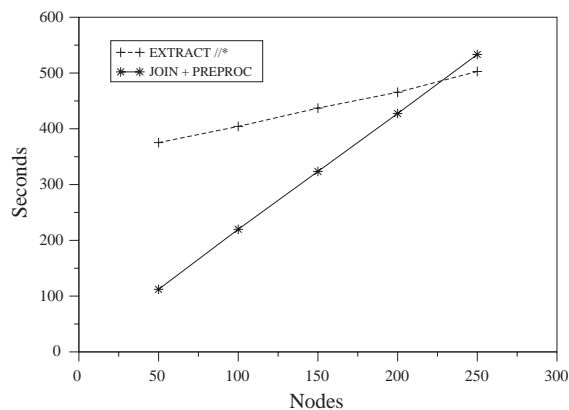

Fig. 6. Performance of Extract—Experiment 1.

Fig. 7. Performance of Extract—Experiment 2.

results, this shows that XML variables are efficiently implementable.

Conclusion: The experiments above indicate that our proposal for implementing meta-querying features *on top of* the database engine induces constant, predictable overheads, which is certainly good news.

Still, the running times are sometimes too high. Therefore, more efficient XSLT processors would certainly help. The ideal solution would be to incorporate XSLT processing directly into the query processor, as has recently been suggested by Moerkotte [20]. It might also help to implement the EVAL and UEVAL functions entirely in a programmed SQL language such as Oracle's PL/SQL, or SQL/PSM, in which case the overhead of calling a table function could be avoided.

On the other hand, our current implementation method has the advantage that it is generally applicable without a need to change the internal query processor (which is often not feasible).

7. Discussion

Other meta-query languages: Meta-SQL is the first practical language for meta-querying. At the same time, however, it is firmly based on our past experience in designing formalisms for meta-querying. More specifically, two of us (in collaboration with Neven and Van Gucht) have earlier introduced the *reflective* relational algebra [21] and the *meta* relational algebra [14], two formal meta-query languages based on the relational algebra rather than on SQL. The two formalisms differ in their approach: the reflective algebra is untyped, stores queries in so-called "program relations", and uses the basic relational algebra operators for the syntactical manipulation; in contrast, the meta algebra is statically typed and views stored queries as an abstract data type with specific operations for syntactical manipulation. Meta-SQL is the practical culmination of these two proposals, and stands in between the two approaches: like the meta algebra, it stores queries in an XML abstract data type, but like the reflective algebra, it is essentially untyped. Every meta-query expressed in the reflective relational algebra, or in the meta

relational algebra, can also be expressed in Meta-SQL.

Schema querying: Starting with the seminal paper on HiLog [22], the concept of *schema querying* has received considerable attention in the literature. Clearly, schema querying is a special kind of meta-querying. For instance, SchemaSQL [23] augments SQL with generic variables ranging over table names, rows, and column names. It is not difficult to simulate SchemaSQL in Meta-SQL, using XML variables and UEVAL calls. Of course, since SchemaSQL is more specialized, it allows more opportunities for optimized implementation [24].

Commercial XML extensions: All the major commercial ORDBMS vendors are providing XML extensions to their products, However, the emphasis there is mainly on publishing results of SQL queries in XML format (e.g., [4,25]), so that they can be further processed using the standard XML tools, including XSLT. This combination of SQL and XSLT is clearly quite different in scope compared to the combination we have proposed in this paper. The other direction, where large XML documents are decomposed and stored in tabular format, has also been researched (e.g., [26,27]) and is getting into the commercial products.

We also mention that several features of "SQL/XML" [28,29] such as the XML data type and the mapping of tables to XML values are similar to those found in this paper. Moreover, an Extract-like operator and functions operating on XML values (albeit through XQuery) are listed among the future work.

Text extensions: Given that most database systems now support a text data type with better functions for text searching and editing than standard SQL, one may wonder why we cannot support meta-querying simply by representing the stored queries as text. The answer is that for many meta-queries the *structure* of the stored queries is important. For example, searching for the use of a certain view name in a query is more than a simple pattern search. Using syntax trees and XSLT makes structural querying very easy.

XML query languages: Although the focus of this paper has been on meta-querying as opposed to general XML querying, we still would like

to conclude by pointing out that Meta-SQL, without EVAL, is not only a language for syntactical meta-querying, but can serve in general as a query language for databases containing XML documents in addition to ordinary relational data. Its closeness to standard SQL and object-relational processing is a major advantage. On the other hand, the treatment of XML documents as abstract data items seems less appropriate for "pure" XML databases, i.e., huge XML documents. However, as already indicated above, there seems to be a strong trend in the database processing world to decompose such XML databases into relational data anyway.

Acknowledgements

Appendix A. A DTD for SQL

The following is a reasonably complete DTD for syntax trees of SQL-92 select-statements, with the exception of the various join operators. The DTD is derived from the grammar given by Date [10].

```
<!ELEMENT query ((select, from, where?,
group_by?, having?, order-by?)
    | (union | except | intersect))>
<!ELEMENT select ((all | distinct)?,
(wildcard | sel-item+))>
<!ELEMENT all EMPTY>
<!ELEMENT distinct EMPTY>
<!ELEMENT wildcard EMPTY>
<!ELEMENT sel-item ((column |
(rangevar, (column | wildcard))
    | scalar | aggregate), alias?)>
<!ELEMENT rangevar (#PCDATA)>
<!ELEMENT column (#PCDATA)>
<!ELEMENT column-ref (rangevar?, column)>
<!ELEMENT scalar (alg-exp |
```

```
concat-exp | column-ref | aggregate
    | constant | query)>
<!ELEMENT aggregate (count-all
    | ((avg | count | max | min | sum),
    (all | distinct)?,
(alg-exp | concat-exp | column-ref |
constant | query)))>
<!ELEMENT count-all EMPTY>
<!ELEMENT avg EMPTY>
<!ELEMENT count EMPTY>
<!ELEMENT max EMPTY>
<!ELEMENT min EMPTY>
<!ELEMENT sum EMPTY>
<!ELEMENT alg-exp (scalar, (add | sub |
mul | div), scalar)>
<!ELEMENT add EMPTY>
<!ELEMENT sub EMPTY>
<!ELEMENT mul EMPTY>
<!ELEMENT div EMPTY>
<!ELEMENT concat-exp (scalar, scalar)>
<!ELEMENT constant (#PCDATA)>
<!ELEMENT from (table-ref+)>
<!ELEMENT table-ref ((table |
query), alias?)>
<!ELEMENT alias (#PCDATA)>
<!ELEMENT table (#PCDATA)>
<!ELEMENT where (cond-exp)>
<!ELEMENT cond-exp (not?, (cond-test |
and | or))>
<!ELEMENT not EMPTY>
<!ELEMENT cond-test (comparison | like |
in | match | all-or-any
    | exists | unique | overlaps |
    test-for-null)>
<!ELEMENT and (cond-exp, cond-exp+)>
<!ELEMENT or (cond-exp, cond-exp+)>
<!ELEMENT rowconstr (column-ref | scalar)
+>
<!ELEMENT comparison
(rowconstr, (eq | lt | let | gt | get |
neq), rowconstr)>
<!ELEMENT eq EMPTY>
<!ELEMENT lt EMPTY>
<!ELEMENT let EMPTY>
<!ELEMENT gt EMPTY>
<!ELEMENT get EMPTY>
<!ELEMENT neq EMPTY>
<!ELEMENT like((column-ref | scalar),
```

```
(column-ref | scalar)
    , (column-ref | scalar)?)>
<!ELEMENT in ((rowconstr, query) |
(scalar, scalar+))>
<!ELEMENT partial EMPTY>
<!ELEMENT full EMPTY>
<!ELEMENT match (rowconstr, unique?,
(partial|full)?, query)>
<!ELEMENT all-or-any (rowconstr,
(eq | lt | let | gt | get | neq)
    , (all | any)?, query)>
<!ELEMENT any EMPTY>
<!ELEMENT exists (query)>
<!ELEMENT unique (query)>
<!ELEMENT overlaps (scalar, scalar,
scalar, scalar)>
<!ELEMENT test-for-null (rowconstr)>
<!ELEMENT group-by (column-ref+)>
<!ELEMENT having (cond-exp)>
<!ELEMENT union (query, all?, query)>
<!ELEMENT except (query, all?, query)>
<!ELEMENT intersect
(query, all?, query)>
<!ELEMENT order-by ((column-ref,
(asc | desc)?)+)>
<!ELEMENT asc EMPTY>
<!ELEMENT desc EMPTY>
```

Appendix B. XSLT programs

```
function cartprod returns xml
begin
  <xsl:template match=" /" >
   <query>
    <select> <wildcard/> </select>
    <from> <xsl:apply-templates
    select="cmb/*"/ > </from>
   </query>
  </xsl:template>

  <xsl:template match="/cmb/*" >
   <table-ref>
  <xsl:copy-of select=" ."/ >
   </table-ref>
  </xsl:template>
end
```

```
function pair param s string returns xml
begin
  <xsl:param name="s"/ >
  <xsl:template match="/" >
   <pair>
    <name> <xsl:value-of select="s"/ >
    </name>
    <xsl:copy-of select=" *"/ >
   </pair>
  </xsl:template>
end

function rewrite param p xml returns xml
begin
  <xsl:param name="p"/ >

  <xsl:template match=" *" >
   <xsl:copy>
    <xsl:apply-templates/>
   </xsl:copy>
  </xsl:template>

  <xsl:template match="table" >
   <xsl:apply-templates select="$p"
  mode="find" >
    <xsl:with-param
   name="search" select="string(.)"/ >
    <xsl:with-param name="caller" >
     <xsl:copy-of select=" ."/ >
    </xsl:with-param>
   </xsl:apply-templates>
  </xsl:template>

  <xsl:template match=" /" mode="find" >
   <xsl:param name="search"/ >
   <xsl:param name="caller"/ >
   <xsl:param name="found"
  select="cmb/pair[name=$search]"/ >
   <xsl:choose>
    <xsl:when test="$found" >
     <xsl:copy-of select="$found/query"/ >
    </xsl:when>
    <xsl:otherwise>
     <xsl:copy-of select="$caller"/ >
    </xsl:otherwise>
   </xsl:choose>
  </xsl:template>
end
```

References

[1] M. Stonebraker, et al., Extending a database system with procedures, ACM Trans. Database Syst. 12 (3) (1987) 350–376.

[2] P. Bernstein, et al., The Asilomar report on database research, SIGMOD Record 27 (4) (1998) 74–80.

[3] XQuery 1.0: an XML query language, W3C Working Draft, www.w3.org.

[4] J. Shanmugasundaram, E. Shekita, et al., Efficiently publishing relational data as XML documents, VLDB J. 10 (2–3) (2001) 133–154.

[5] XSL transformations (XSLT) version 1.0, W3C Recommendation 16 November 1999, www.w3.org.

[6] D. Tidwell, XSLT, O'Reilly & Associates, Sebastopol, CA, 2001.

[7] M.H. Kay, XSLT Programmer's Reference, Wrox Press, 2001.

[8] Document object model (DOM) level 2 core specification version 1.0., W3C Recommendation 13 November 2000. www.w3.org.

[9] Extensible markup language (XML) 1.0 (2nd edition), W3C Recommendation 6 October 2000, www.w3.org.

[10] C.J. Date, A Guide to the SQL Standard, 4th Edition, Addison-Wesley, Reading, MA, 1997.

[11] S. Cluet, Designing OQL: allowing objects to be queried, Inf. Syst. 23 (5) (1998) 279–305.

[12] XML path language (XPath) version 1.0, W3C Recommendation 16 November, 1999, www.w3.org.

[13] S. Abiteboul, P. Buneman, D. Suciu, Data on the Web: From Relations to Semistructured Data and XML, Morgan Kaufmann, Los Altos, CA, 2000.

[14] F. Neven, J. Van den Bussche, D. Van Gucht, G. Vossen, Typed query languages for databases containing queries, Inform. Syst. 24 (7) (1999) 569–595.

[15] M. Stonebraker, P. Brown, Object-relational DBMSs: Tracking the Next Great Wave, Morgan Kaufmann, Los Altos, CA, 1999.

[16] J. Melton, A. Simon, SQL:1999: Understanding Relational Language Components, Morgan Kaufmann, Los Altos, CA, 2001.

[17] Meta-SQL prototype, available at http://www.luc.ac.be/theocomp/.

[18] D. Chamberlin, A Complete Guide to DB2 Universal Database, Morgan Kaufmann, Los Altos, CA, 1998.

[19] M. Kay, SAXON: the XSLT processor, saxon.sourceforge.net.

[20] G. Moerkotte, Incorporating XSL processing into database engines, VLDB 2002, pp. 107–118.

[21] J. Van den Bussche, D. Van Gucht, G. Vossen, Reflective programming in the relational algebra, J. Comput. Syst. Sci. 52 (3) (1996) 537–549.

[22] W. Chen, M. Kifer, D.S. Warren, HiLog: a foundation for higher-order logic programming, J. Logic Programming 15 (3) (1993) 187–230.

[23] L.V.S. Lakshmanan, F. Sadri, I.N. Subramanian, SchemaSQL: a language for interoperability in relational multi-database systems, VLDB 1996, pp. 239–250.

[24] L.V.S. Lakshmanan, F. Sadri, I.N. Subramanian, On efficiently implementing SchemaSQL on an SQL database system, VLDB 1999, pp. 471–482.

[25] M.F. Fernandez, D. Suciu, W.C. Tan, SilkRoute: trading between relations and XML, Comput. Networks 33 (2000) 723–745 Proceedings WWW9.

[26] J. Shanmugasundaram, K. Tufte, C. Zhang, et al., Relational databases for querying XML documents: limitations and opportunities, VLDB 1999, pp. 302–314.

[27] A. Deutsch, M.F. Fernandez, D. Suciu, Storing semistructured data with STORED, SIGMOD 1999, pp. 431–442.

[28] A. Eisenberg, J. Melton, SQL/XML and the SQLX informal group of companies, ACM SIGMOD Record 30 (3) (2001) 105–108.

[29] A. Eisenberg, J. Melton, SQL/XML is making good progress, ACM SIGMOD Record 31 (2) (2002) 101–108.