Meta-SQL: Towards Practical Meta-Querying

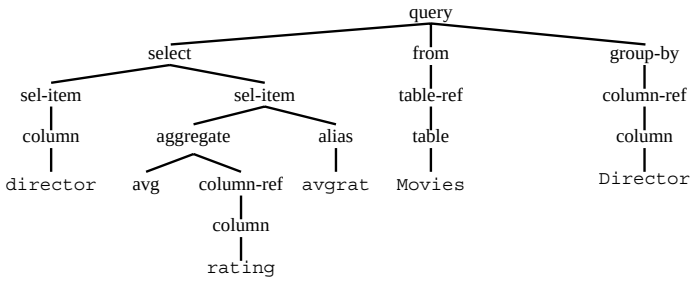
Jan Van den Bussche[1], Stijn Vansummeren[1], and Gottfried Vossen[2]

[1] Limburgs Universitair Centrum
[2] University of Münster

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. 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. Here are some examples of meta-queries to a usage log. *(i)* "Which queries in the log do the most joins?" *(ii)* "Which queries in the log return an empty answer on the current instance of the database?" *(iii)* View expansion: "Replace, in each query in the log, each view name by its definition as given in the system catalog." *(iv)* 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?"

We present *Meta-SQL*, a system that allows the expression of meta-queries directly in SQL. Our presentation is extremely condensed; a full paper on the language and our prototype implementation is available.[3]

Storing SQL expressions as syntax trees. Consider a simplified system catalog table, called `Views`, containing view definitions. Traditionally, there is a column `name` of type string, holding the view name, and a column `def`, also of type string, holding the SQL expression defining the view. Because such a flat string representation of an SQL expression makes structured syntactical manipulations difficult, we instead declare `def` to be of type XML, which is an allowed column datatype in Meta-SQL (and in many modern SQL implementations as well). We then store SQL expressions as syntax trees in a convenient XML format. Here is an example:



[3] J. Van den Bussche, S. Vansummeren, and G. Vossen. Towards practical meta-querying. http://arxiv.org/abs/cs.DB/0202037.

Calling XSLT from within SQL. Suppose, in addition to our `Views` table, 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 a simple auxiliary XSLT program `mentions_table`, which we then call from within SQL:

```
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'
```

Indeed, XSLT is a widely used manipulation language for XML data. An XSLT program takes an XML tree as input, and produces as output another XML tree (which could be in degenerate form, holding just a scalar value like a number or a string). We also use the XSLT top-level parameter binding mechanism, by which programs can take additional parameters as input. Our system embraces XSLT because it is the most popular and stable standard general-purpose XML manipulation language to date. When other languages, notably XQuery, will take over this role, it will be an easy matter to substitute XSLT by XQuery in our system.

As a second 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:

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

Here `unite` is an easy XSLT program (omitted) that transforms two trees t_1 and t_2 into the tree

<union> t_1 t_2 </union>

XML variables. Consider the 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." We express this query in Meta-SQL using an *XML variable*:

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

Here, `x` is an XML variable ranging over the `<table>` subelements of `v.def`. XML variables are bound in the from-clause, in a similar way variables are bound in OQL. We can use any XPath expression within the square brackets to delimit the range of nodes.

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
```

EVAL. So far we have only seen examples of meta-queries that rely on the syntax of the stored SQL queries only. Meta-SQL also allows the expression of meta-queries whose answer depends on the results of dynamically executing stored queries. This is done via the new built-in function EVAL, which takes an SQL query (more correctly, its syntax tree in XML format) as input, and returns the table resulting from executing the query.

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
```

There is also a function UEVAL in case we have no information about the output scheme of the stored queries we are evaluating. UEVAL presents the rows resulting from the dynamic evaluation of the query in XML format.

The System. We have developed a prototype Meta-SQL implementation, usable on top of DB2 UDB, and freely available at http://www.luc.ac.be/theocomp. Our implementation takes a cross-compilation approach, and heavily relies on the facility of user-defined functions.