OceanQuery: report editor query language

Marcin Stefaniak[1], **Łukasz Kamiński**[2]

Abstract: We present design principles of OceanQuery, the query language behind the Ocean Genrap report editor. Ocean Genrap reporting system is able to query data from a variety of data sources, including existing SQL relational databases. However, the usability principles of the editor application lead beyond SQL, to a stack-based and tree-result query language. We present OceanQuery by analogy and comparison to the XQuery core of FLWOR expressions. Particular attention is paid to aggregations and grouping, which is explicit in OceanQuery. We include results of efficiency experiments in comparision against other major XQuery engines.

Keywords: reporting tools, query languages, XQuery, grouping

1. Introduction

We study software tools for authoring reports. A report is a document schema which is supposed to be filled with data retrieved from a data source, often a database. Documents are hierarchical, while relational database (SQL) query result is flat. Indeed, document can contain several nested iterations of data (tables, listings, enumerations, etc), while SQL result is only a single sequence of uniform rows. When designing Ocean Genrap report editor application principles of end-user usability were among the top priorities. For instance, it is not required to understand SQL in order to create or edit reports. Furthermore, a selected report fragment can be cut/copied and pasted elsewhere. Because of that, we needed a query language that would be compositional, in such a way that a query fragment could be naturally pasted into another place. An elegant query language emerges from this requirement, based on the concepts of environment/variable stack known from programming language theory. It also follows that the query result is freely tree-shaped. We call our query language OceanQuery throughout this paper.

Among several execution engines of OceanQuery the most interesting one is SQL engine. It translates OceanQuery queries into SQL statements, including aggregations and group-by statements (for which there is an explicit language construct in OceanQuery), so that database server calculates aggregations whenever possible. The details of translation into SQL are left outside the scope of this paper.

This work is organized as follows: in next section related work is surveyed, especially XQuery to SQL translators. In the 3rd section, OceanQuery language is described using a syntax similar to XQuery and its FLWR expressions. Next section contains results of stress test experiments. Finally, conclusions are drawn.

[1] Institute of Informatics, University of Warsaw, Banacha 2, 02-097 Warsaw, Poland
e-mail: `stefaniak@mimuw.edu.pl`
[2] ComArch SA, Leśna 2, 02-844 Warsaw, Poland e-mail: `Lukasz.Kaminski@comarch.pl`

2. Related work

There is a huge variety of query languages – let us classify them roughly according to the structure and organization of input data and result data. The data can be organized according to one of the following models (the list is not exhaustive)

- relational – data is structured as flat tables (example: SQL)
- tree-shaped, hierarchical – data forms a (usually finite) tree (example: XML)
- object, network – data in objects that may reference other objects (example: OQL)

A few major query languages compared to OceanQuery are listed in table 1.

Table 1. Comparison of data models of query languages

language	input	result
SQL	relational	flat table
OQL	object	object
XQuery	tree-shaped	tree-shaped
SBQL	object	tree-shaped
OceanQuery	network	tree-shaped

Note: SBQL is the Stack-Based Approach query language developed by researchers under lead of K.Subieta (Subieta, 2006). It seems conceptually similar to OceanQuery; both of them employ the notion of an environment stack. The stack, along with variable scope and binding, are fundamental in the programming languages theory. Yet, within the field of query languages (dominated by the relational SQL) they were scarcely supported.

However, the difference between network and tree-shaped input data model is negligible, and XQuery (W3C XQuery, 2006) also emerges as a similar language to OceanQuery. XQuery belong to the vast family of query languages for tree-shaped, semistructured data. The family includes such languages as TQL (Conforti et al, 2002), Lorel (Abiteboul et al, 1997) and Quilt (Chamberlin et al, 2001) – the predecessor of XQuery.

There has been a considerable research on translating (DeHaan et al, 2003) and compiling (Grust et al, 2004) XQuery to SQL. Those systems encode XML documents into relational tables using additional data, like dynamic intervals technique, in order to allow all XPath expressions. Another works towards this goal are SilkRoute systems (Re et al, 2006) and (Fernandez et al, 2002), where the XML query addresses existing relational database, but only a subset of XQuery is supported. The presence of those research makes XQuery especially relevant to our work.

Although OceanQuery looks similar, its design was not influenced heavily by those other query languages, simply because they were either immature or unknown to the authors then. What contributed most to the design principles of our query language was the usability of the report editor application. We are unaware of other SQL-compliant report editor products which would widely allow copy-and-paste manipulation on report fragments.

Another related issue is the data grouping feature, known in SQL as a `GROUP BY` clause. The W3C recommendation XQuery language does not include such. This can be justified by the fact that in result-tree-shaped query languages grouping can be achieved using *distinct* operation and nested *for-where* expressions (Subieta, 2006). This could be a bit cumbersome for users creating and editing queries. Moreover, grouping patterns should be detected (Deutsch et al, 2004) before the query is executed as SQL in order to avoid bad efficiency (Re et al, 2006). Therefore, engineers (Borkar et al, 2004) and researchers (Gokhale et al, 2007) proposed enhancing XQuery with a kind of *group-by* clause. In OceanQuery we address this issue with explicit, nested and multi-criteria group-by construction. This is consistent with Genrap user interface (user can group by something on a report with a single manipulation) as well as with translation to SQL (efficient queries for the details of *group-by*).

3. Description of OceanQuery language

We shall describe OceanQuery using a syntax similar to XQuery description with respect to its FLWOR expressions. Ocean queries results with data structured in records and arrays rather than XML, so there are no „element” constructors in it. The form of the result data is more like JSON rather than XML, and our expressions will borrow a bit of syntax from JSON records.

OceanQuery is a statically typed language, but throughout this description we omit the typing information that is normally stored within OceanQuery expressions whenever it can be inferred from the context. Indeed, this type info is used when the report query is manipulated by user, or when the database schema is not available for some reason, so it's only auxiliary to processing and understanding OceanQuery queries.

One should have in mind that OceanQuery is not a language formalized and specified by a grammar, but rather, it is an in-memory data structure that is bound somehow to the report layout, and only certain expressions of query are visible at a time to the user. The best analogy is a spreadsheet application: a spreadsheet combines together a layout (rows and column width, font families and sizes, etc.) and a graph of calculations. This graph could be printed separately as a set of equations, but in practice only one equation at a time, from the selected cell is viewed or edited by a user. In this section, we barely discuss the layout issues of Ocean Genrap. Instead, we focus on the calculation part of the report, which altogether forms an OceanQuery query. Two document elements are relevant: the data is filled into *variable fields*, and there are *grids*, which are tabular iterations over sequence of data.

Each of iteration expressions introduce a variable into the scope of the grid. Occasionally, more than one simple type variable is introduced, for example when grouping by several criteria. Internally those variables are identified by a variant of de Bruijin indices, but for the purposes of this description we simply use variable names, for clarity prefixed with a dollar '$' sign.

3.1. Database schema model

The OceanQuery database schema model is inspired by the Entity-Relationship Model (Chen, 1976). The database schema is represented as a set of entities (types, domains), and relationships among them. Each entity may have one or more named attributes, each attribute is of a simple type, such as number, string, date-time, boolean. An instance of the entity instance may have attribute values, of corresponding names and types.

Unlike in the traditional ER model, the relationships are directed. Each relationship leads from source entity to target entity, and is a (partial) function from source entity instances to target entity instances. There are two major kind of relationships: plural (to-many) and singular (to-at-most-one). In the usual case of equational relationship (like primary key - foreign key) in relational database, two opposite relationships may be in the schema, and the fact that they are inverses remains as a mere hint for query optimizer.

Since in the domain of report querying only read-only data access is considered, we find „network" conceptual data model simple yet powerful. However, it does not support notions of inheritance nor variants – but neither do plain relational databases.

3.2. The core of query language

The main construct of OceanQuery is a loop over an entity set, similar to the `FOR` loop of XQuery's FLWOR expressions. Actually, since we don't have `LET` expressions, so it is more like FWOR (For-Where-OrderBy-Return) expression.

Here's a brief grammar of OceanQuery expressions:

query ::= [*parameters*] `{` *body-expr* `}`
parameters ::= `GIVEN` (parameter-name `:` parameter-type)* `RETURN`
body-expr ::= (`"tagname" : (` *value-expr* | *fwor-expr*))*
fwor-expr ::= `FOR` variable `IN` *entity-expr*
  [`ORDER-BY` *value-expr+*]
  `RETURN {` *body-expr* `}`
entity-expr ::= (variable | `ALL` *entity-name*) *path-expr* [*filter-expr*]
filter-expr ::= `WHERE` *bool-expr*
attr-expr ::= variable *path-expr* . *attribute-name*
path-expr ::= (`/` relationship)*
value-expr ::= constant | *attr-expr* | ... arithmetic and logical expressions
bool-expr ::= *value-expr* ... // evaluating logical condition

Here's an example OceanQuery query reporting grades of students from a particular year.

```
GIVEN $year : int RETURN {
   "year" : $year
   "students" : FOR $s IN ALL Students WHERE $s.year = $year
   RETURN {
      "grades" : FOR $ac IN $s/attended_classes
                       ORDER-BY $ac/class.name RETURN {
          "class" : $ac/class.name
          "grade" : $ac.grade
      }
   }
}
```

The *value-expr* represents scalar expressions, i.e. they should evaluate to a simple value. They may refer to variables accessible in the current scope and may be built of various arithmetic and logical operators and functions that are supported by the database.

The FOR expression is a loop iterating over its variable values among a list of entity instances resolved from its *entity-expr*. Each variable stores an entity instance. Parameters are treated as top-level declared variables that store an entity instance or a simple value.

It should be also noted that the *path-expr* of an *attr-expr* should consist of relationships that are singular, so that the result of an *attr-expr* is a scalar value. On the other hand, the *entity-expr* is supposed to be plural, but no harm is done when it happens singular – it's just odd to have a single-row grid.

3.3. Named expressions

One might wonder why the LET expression is missing from OceanQuery query language. The LET expression introduces a variable in a similar fashion to FOR expression, but it does not iterate over a set of entity instances, but rather evaluate to a single entity instance or value. This actually might happen in OceanQuery when the *path-expr* of an *entity-expr* in a FOR expression does not contain a plural relationship, e.g. the data schema is inconsistent, or the static type correctness is not enforced.

However, there is no explicit LET expression because there is no corresponding report layout element. On the other hand, FOR expressions are naturally bound to grid (tabular iterations) layout elements, and FOR expression is accessible by selecting the particular grid. With LET expressions, there is no such element that would maintain the same scope, as the LET expression, and making artificial scope segments in the report seems very odd indeed.

Any query with LET expressions can be transformed into an equivalent query without LET expressions, simply by inlining the variables introduced by them. Yet the possibility to extract common expressions into LET variable is important for user-experience. We achieve this by using a technique "named expressions". Any element of a *body-expr* can be accessed by the user, and is identified by its *tag-name*. User can refer to them within another expressions in the same scope. During query evaluation, when *value-expr* is referred, it is inlined in the place of the referral. When *fwor-expr* is referred, its *entity-expr* (along with *filter-expr*, if present) is inlined. Finally, the variable field of the named expression may be hidden, so it's not displayed on the report layout.

The grammar is extended with the following:

entity-expr ::= ... | `REF` "tagname" // ... refers *fwor-expr*
value-expr ::= ... | `REF` "tagname" // ... refers *value-expr*

Within a single scope (for example, the top-level one) the structure of a query with named-variables is similar to a spreadsheet. Issues that apply to spreadsheets (e.g. cyclic references) may occur in OceanQuery, and the copy-paste behavior is similar to what modern spreadsheets offer. This way, named expressions are a generalization of spreadsheet computation, because there may be multiple scopes in the query. There are special rules for referring expressions across-the-scope. Obviously, we may refer to an expression that is defined in the same scope or in an ancestor scope. Moreover, there are some circumstances when it is meaningful to refer to an expression defined in a sub-scope. This commonly occurs when we make a sum over an existing variable field in a grid.

```
"b" : FOR $x IN entity RETURN {
  "c" : $x.val
}
"a" :  SUM(REF "b", REF "c")
```

It seems a bit complicated when considered together with aggregations and grouping. However, all the named expressions references can be inlined during query pre-processing phase, and the semantics and execution of OceanQuery query are defined without them. Thus, we shall not consider named expressions further.

Finally, there is a related feature called user-defined expressions. When a common sub-expression is dependent only on a single entity instance, which happens very frequently, it can be defined outside the OceanQuery query. The definition is stored in the so-called *user-defined schema*, which acts as an extension to the data source schema.

3.4. Aggregations

Besides the usual scalar-to-scalar expressions, we extend the core by aggregation expressions, which calculate a scalar value from a plurality of values. The same aggregation functions as in SQL are supported, that is, sum, average, maximum/minimum, as well as „count" and „exists" functions.

Below is the extension to core grammar.

value-expr ::= ... | *aggr-expr* | *count-expr* | *exists-expr*
aggr-expr ::= *aggr-kind* `(FOR` variable `IN` *entity-expr* `,` *value-expr* `)`
aggr-kind ::= `SUM` | `AVG` | `MAX` | `MIN`
count-expr ::= `COUNT (` *entity-expr* `)`
exists-expr ::= `EXISTS (` *entity-expr* `)`

The *entity-expr* is evaluated in the current environment to yield an entity-set, and another *value-expr* is evaluated with the environment extended with this entity-set.

Example:

```
{
   "students" : FOR $s IN ALL Students
       WHERE COUNT($s/attended_classes) > 5 RETURN {
      "avg_grade" : AVG(FOR $ac IN $s/attended_classes, $ac.grade)
   }
}
```

3.5. Grouping

OceanQuery supports nested grouping – a grid can be grouped and grouped again. It is possible to perform grouping by more than one expressions at once. The group expression is either a simple-type *value-expr*, or a singular *path-expr* evaluated in the context of the grouped *entity-expr*. The later is more or less equivalent in SQL to grouping by foreign keys, but it allows more usable typing of the group items.

The grammar is extended as follows:

entity-expr ::= (… | *entity-group-expr* | *entity-detail-expr*) [*filter-expr*]
entity-group-expr ::= `GROUP` variable `=` *entity-expr* `BY` *group-item*+
group-item ::= variable `=` (*value-expr* | *path-expr*)
entity-detail-expr ::= `DETAILS` group-variable

The meaning of *entity-group-expr* is to introduce one variable for every *group-item* and iterate over groups. Moreover, a special group-variable is introduced which represents the plurality of grouped entities. This group-variable can be used only in *entity-detail-expr*, which results in looping over details of the current group. Naturally, the type of details of the group is the same as the grouped *entity-expr*.

Below is an example:

```
FOR $g IN GROUP $s = ALL Students BY $y = $s.year, $p = $s/programme
RETURN {
   "programme" : $p.name
   "year" : $y
   "students" : FOR $s IN DETAILS $g RETURN { "student" : $s.name }
   "students_by_gpa" : FOR $g2 IN GROUP $s = DETAILS $g BY $s.gpa
   RETURN {
      "gpa" : $gpa
      "students" : FOR $s IN DETAILS $g2
      RETURN { "student" : $s.name }
   }
}
```

The *entity-expr* contained in *entity-group-expr* is forbidden to be *entity-group-expr*. This is not an obvious requirement, but we insisted on making our queries simple and concise. And because the reports are authored by user-interface gestures rather than by hand, the burden of making necessary transforms is placed on the software, not the user. However, not every query with nested groups can be naturally expressed in the restricted form, For example, if one makes a group and group it again by an aggregation of their details.

In (Borkar et al, 2004), an extension to XQuery was proposed in order to feature grouping, in such a way that an optional `group` clause is added to the FLWOR construct. This is naturally limited the number of groupings per loop iteration to one, and so their approach is equivalent to our restricted-form grouping.

3.6. Translation to SQL

Let us outline very briefly how OceanQuery queries are performed on SQL databases. The approach is quite similar to the SilkRoute (Fernandez et al, 2002)system. We assume that each entity corresponds to a database table, and that their primary keys are known. For each node in the tree of our query, an SQL query is issued and their results are merged into a data tree.

Let us call the places in the query with *body-expr* – that is, `FOR` expressions and the top-level – the *nodes* of that query. Each node can be translated to an independent SQL query. The SQL query for a node returns identity columns and value columns. The identity columns contain variable values – primary keys of entities or scalar values – corresponding to the variable environment of that node. The value columns contain scalar values corresponding to the *value-expr*s associated with that node. For each node except the root (top-level one), the result is grouped and inserted into the data queried for its parent node, possibly sorted according to the order-by clause. Within this framework, the problem of executing OceanQuery is reduced to constructing appropriate SQL queries for each OceanQuery node, which is beyond the scope of this paper.

4. Experimental Results

In this section, we present the results that demonstrate the time efficiency of the execution of Ocean queries. We measured the time of execution for three different database queries with varying data sizes using four different query engines:

- Saxon 8.9.0.3N [3] – XQuery engine
- MonetDB/XQuery v4.0 [4] – XQuery front-end to a high-performance database management system
- GenRap/XML – OceanQuery interpreter for in-memory XML-shaped data
- GenRap/SQL – OceanQuery engine for SQL databases.

We used OCEAN GenRap 2006 M10 SDK, Java version [5] for OceanQuery evaluation. For testing GenRap/SQL, the MS SQL Server 2000 database was used.

[3] available at `http://saxon.sourceforge.net/`
[4] available at `http://monetdb.cwi.nl/projects/monetdb/XQuery/index.html`
[5] available at `http://genrap.comarch.com`

Table 2. Timings for test 1 (CPU sec)

N		Saxon	Monet	GR/XML	GR/SQL
1000		0.82	0.15	0.07	0.03
5000		15.25	0.23	0.32	0.11
10,000		61.61	0.35	0.69	0.25
50,000		-	1.36	3.8	1.5
100,000		-	2.78	8.0	2.5
200,000		-	7.77	20.9	6.9
500,000		-	19.1	-	25.0

Experiments were run on an Intel Pentium M1,73GHz system with 1GB RAM running Windows XP. All software configuration were default, except for Monet-DB/XQuery, which memory limit was increased to 640MB. Each test result is an average of ten executions and measuring user and system time spent. For Saxon, two runs were performed each time: one with the tested query, and one with empty query, so as to exclude the overhead of reading the data.

4.1. Test 1

This test features a single entity `invoice(id, name, date, value)` and calculating sums of `value` for `invoice`s grouped by `date`. The XQuery in Saxon form is

```
for $i in fn:distinct-values(./invoice/@date)
return <group><date> {$i} </date><values>{
      fn:sum(./invoice[@date = $i]/@value)
    }</values></group>
```

The corresponding OceanQuery contains explicit groupping.

Timing results for this test are shown in table 2. The number of invoices in the data is denoted by N. Saxon engine seems to be evaluating the query with $O(N^2)$ strategy, while the other engines use rather linear strategies. The GenRap/XML engine is slower (up to a constant), which is easily explained by the fact that it's unoptimized, low-performance query interpreter only.

4.2. Test 2

In the second test, the `invoice` schema is extended with entity `entry(id, count, price, name)`, with a one-to-many relationship between `invoice` and `entry`. The test query is supposed to calculate the average value of invoices, where the value of invoice is the sum of it's entries values, and the value of each entry is its price multiplied by its count attribute. Again, the XQuery is:

```
fn:avg((for $i in ./invoice
```

Table 3. Timings for test 2 (CPU sec)

N		Saxon	Monet	GR/XML	GR/SQL
5000		0.36	0.22	0.19	0.01
10,000		0.47	0.31	0.39	0.01
50,000		1.23	1.03	1.9	0.11
100,000		2.27	1.97	4.0	0.17
200,000		4.63	3.77	8.1	0.34
500,000		11.2	9.22	-	0.86
1000,000		25.4	18.4	-	1.7

```
return
  <vv value='{
    fn:sum( (for $j in $i/entry
      return <v value='{$j/@count * $j/@price}'/>)/@value )
    }'/>)/@value )
```

Timing results for this test are shown in table 3. The number of invoices in the data is about $\sqrt{N}$, and each invoice has its own $\sqrt{N}$ entries, which results in approximately N rows. It seems that all engines executed the query in a linear-time fashion, yet the GenRap/SQL engine was significantly high-performing, which could be attributed to the maturity of modern relational database engines.

4.3. Test 3

For test 3, the `invoice-entry` schema is further extended with `product(name, tax)` and `category(id, name)` entities. Each `entry` is related to exactly one `product`, and each `product` falls into exactly one `category`. A `product` is related to its `entry` instances by a relationship named `entries`, and a `category` is in relationship `products` with its `product`-s.

The query is supposed to determine for each category, the average value of total invoiced gross value of products from this category. And the gross value of a product is its value multiplied by the tax rate of the product. The XQuery is:

```
for $i in ./category return
  <category id='{$i/@id}' value='{
    fn:avg( (for $j in $i/product
      return <v value='{$j/@tax * fn:sum(
        (for $k in .//entry[@productId = $j/@id]
         return <v value='{$k/@count * $k/@price}'/>)/@value
      )}'/>)/@value )
    }'/>
```

The corresponding OceanQuery employs grouping as follows:

Table 4. Timings for test 3 (CPU sec)

N		Saxon	Monet	GR/XML	GR/SQL
5000		0.87	3.32	0.36	0.03
10,000		2.04	9.7	0.73	0.04
50,000		16.0	-	4.0	0.14
100,000		46.1	-	8.6	0.26
200,000		145.5	-	18.3	0.36
500,000		625.5	-	-	0.87
1000,000		-	-	-	1.7

```
FOR $c IN ALL category RETURN {
  "id" : $c.id
  "value" : AVG(
     FOR $g IN GROUP $e = $c/products/entries BY $p = $e/product,
       $p.tax * SUM(FOR $k IN DETAILS $g, $k.count * $k.price)
}
```

Timing results for this test are shown in table 4. Again, the number of invoices in the data is $\sqrt{N}$, and each invoice has its own $\sqrt{N}$ entries. There is about $\sqrt[4]{N}$ categories and $\sqrt{N}$ products. Each entry refers to a product chosen randomly with uniform distribution, so for one product there are on average $\sqrt{N}$ entries referring to it. In this test, only GenRap/SQL prevailed. The reason behind the early fall of MonetDB/XQuery is its ineffective memory management - it overused the hard disk swap.

5. Conclusions

In this paper, we have presented the query language used in the report editor application Ocean GenRap. We have explained how it is related to a mainstream semi-structure query language XQuery, and the approach used to feature explicit grouping. The feasibility of such a language has been shown by experiments with its implementation in an industry-grade software product. It appears that high performance of query execution can be obtained by translating OceanQuery into SQL queries and executing these on a relational database engine. Further research should focus on efficient translation of OceanQuery into SQL queries and other database query languages.

5.1. Acknowledgment

This paper is based on results of the project “New generation of data access languages for intelligent reporting and business data analysis” realized by ComArch SA in cooperation with Poznań University of Technology within the Sectoral Operational Programme “Improvement of the Competitiveness of Enterprises, years

2004-2006", WKP_1/1.4.1/1/2005/41/41/279/2005/U. The project was funded in part by the European Regional Development Fund.

References

Abiteboul, S., Quass, D., McHugh, J., Widom, J., Wiener, J. L. (1997) The Lorel query language for semistructured data. *International Journal on Digital Libraries* **1**, 1, 68–88

Borkar, V. and Carey, M. (2004) Extending XQuery for grouping, duplicate elimination, and outer joins. *XML conference*

Chamberlin, D., Robie, J., and Florescu, D. (2001) Quilt: An XML query language for heterogeneous data sources. *Lecture Notes in Computer Science* vol.**1997**

Chen, P. P. (1976) The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems* **1**, 1, 9–36

Conforti, G., Ghelli, G., Albano, A., Colazzo, D., Manghi, P., and Sartiani. C. (2002) The query language TQL. *Proceedings of WebDB* 13-18 `http://www.db.ucsd.edu/webdb2002/papers/43.pdf`

DeHaan, D., Toman, D., Consens, M. P. and Özsu, M. T. (2003) A comprehensive XQuery to SQL translation using dynamic interval encoding. *Proceedings of SIGMOD Conference* 623–634

Deutsch, A., Papakonstantinou, Y., and Xu. Y. (2004) Minimization and group-by detection for nested XQueries. *Proceedings of ICDE* p. 839. IEEE Computer Society

Fernandez, M. F., Kadiyska, Y., Suciu, D., Morishima, A. and Tan, W. C. (2002)] Silkroute: A framework for publishing relational data in XML. *ACM Trans. Database Syst.* **27**,4,438–493

Gokhale, C., Gupta, N., Kumar, P., Lakshmanan, L., Ng, R. and Prakash. B. A. (2007)] Complex group-by queries for XML. *To appear in 2007 IEEE 23rd International Conference on Data Engineering (ICDE 2007)*

Grust, T. Sakr, S. and Teubner. J. (2004) XQuery on SQL hosts. *Proc. of the 30th Int'l Conference on Very Large Data Bases (VLDB)*

Re, C., Suciu, D., and Brinkley, J. (2006) A performant XQuery to SQL translator. *Technical Report University of Washington, Seattle* `http://silkroute.cs.washington.edu/SilkRouteII_TR.pdf`

Subieta. K. (2006) Stack-based approach and stack-based query language description. `http://www.sbql.pl`

W3C Consortium (2006) XQuery 1.0: an XML query language. *W3C Proposed Recommendation* `http://www.w3.org/TR/xquery/`