SQL to XQuery Translation in the AquaLogic Data Services Platform

Sunil Jigyasu
Persistent Systems Pvt Ltd
sunil_jigyasu@persistent.co.in

Sujeet Banerjee
Persistent Systems Pvt Ltd
sujeet_banerjee@persistent.co.in

Vinayak Borkar
BEA Systems, Inc
vborkar@bea.com

Michael Carey
BEA Systems, Inc
mcarey@bea.com

Kanad Dixit
Persistent Systems Pvt Ltd
kanad_dixit@persistent.co.in

Anil Malkani
BEA Systems, Inc
amalkani@bea.com

Sachin Thatte
BEA Systems, Inc
sachin@bea.com

Abstract

SQL has long been the standard language for retrieving and manipulating data in relational database systems. XML has become the standard format for data exchange, and XQuery is on its way to becoming the standard language for querying XML data. The BEA AquaLogic Data Services Platform provides a service-oriented, XML-based view of heterogeneous enterprise data sources and allows this view to be queried using XQuery. The AquaLogic DSP includes a JDBC driver that connects the old (SQL) world with the new (XML) world via a SQL-to-XQuery translator. This paper outlines the issues related to creating such a driver and details the approach used to translate SQL queries into XQuery expressions. The paper also touches on performance considerations related to handling XML query results in a context where JDBC result sets are the desired output format.

1. Introduction

For the past few decades, enterprises have been using relational databases and SQL to store and manage their data assets. As a result, many existing enterprise applications and client tools are SQL-based. Today, a new breed of applications are being created, mostly in the integration space, applications that are transporting, accessing, and even storing data in XML form due to its efficacy as a platform-independent data format. In addition, XML-based Web services are becoming a common way for enterprise applications to make their functionality available in today’s integration scenarios. The BEA AquaLogic Data Services Platform [5] provides enterprises with a unified, service-oriented, XML-based view of data from heterogeneous information sources and enables them to be integrated and queried using XQuery [1]. This is a natural and effective approach to dealing with data in the context of the Web services and SOA (Service-Oriented Architecture) IT movements [4].

To bridge the gap between its XML-based approach and existing SQL-based applications and tools, AquaLogic DSP includes a JDBC driver that provides SQL access to its otherwise XML-based universe of data. This driver is centered on a SQL-to-XQuery translation component that converts SQL statements into XQuery expressions that can be executed by AquaLogic DSP. The driver also converts the results of executing such XQuery expressions into JDBC result sets. This approach permits legacy SQL-based applications and tools to utilize the fruits of a data service architect’s AquaLogic DSP integration labor “as is”, i.e. without their having to be rewritten to issue XQuery instead of SQL queries. This is especially useful when it comes to reporting, as there are a number of good SQL-based reporting tools available.

In this paper, we describe the issues and technical approach taken to converting SQL statements to XQuery expressions and XQuery results to JDBC result sets. While the discussion is in the context of the AquaLogic DSP product, the principles should be generally applicable elsewhere. Should XQuery be as successful as its advocates would like it to be, we hope that others may benefit from this work as well.
The BEA AquaLogic Data Services Platform (DSP) is middleware to enable service-oriented applications and Web applications to access data from multiple heterogeneous sources in a uniform way. It does so by facilitating declarative development and maintenance of a data services layer in the overall IT architecture of the enterprise. The data services layer is organized as a collection of data services [4], each of which provides a set of access, navigation, and/or update functions related to some meaningful business object in the enterprise (e.g., a Customer, Service Case, or Order).

AquaLogic DSP allows data service developers and consumers to build their data services and applications without regard for the different kinds of data sources and their different APIs and data representations. As shown in Fig 1, all Physical sources of data (relational tables, stored procedures, Web services, files, custom Java functions, etc.) are exposed to developers as callable functions that return a sequence of complex XML elements. XQuery can be used to query over these functions and/or to define new functions for higher-level views (logical data services) that transform and integrate data from one or more of the physical data services. Since XQuery is a declarative language, the resulting higher-level data services and queries are amenable to the use of query optimization techniques.

In this context, the problem at hand is to bridge the gap between the AquaLogic DSP world and the SQL world. Bridging this gap involves mapping between DSP artifacts and SQL artifacts, translating queries from SQL to XQuery, and mapping XQuery results (efficiently) into SQL results. Figure 1 provides a high-level architectural summary of the problem.

2.1. XQuery in a Nutshell

While we assume that our readers are thoroughly familiar with SQL, we provide a brief overview of XQuery since it is still an emerging standard [1,3]. XQuery is based on a model of XML called the XQuery data model. Since XML data is naturally ordered, the XML data model is based on ordered trees. Central to the XML data model is the notion of a sequence. Queries in XQuery consume and produce sequences that consist of atomic values (based on the primitive types of XML Schema) and/or XML nodes (element, attribute, text, document, and so on).

XQuery is a functional, side-effect-free language. An XQuery query consists of a prologue and a body, where the body is an expression. The result of a given query is the result of evaluating its body expression in the environment defined by its prologue. XQuery expressions can be simple, such as constants (“John Doe” or 1.3), variables, arithmetic expressions, function calls, or path expressions. They can also be combined to form more complex expressions via operators, functions, FLWOR expressions (the heart of XQuery for users familiar with SQL), typeswitch expressions, and node constructors.

XQuery is rich enough to support navigation within an XML input document, combining of data from multiple XML inputs, and generation of new XML structures from one or more XML inputs. To generate new XML structures, XQuery adopts a JSP-like approach. XQuery embeds a subset of XML’s syntax and enriches it with expressions that are executed dynamically and replaced inside the XML structures with their results. Query writers can switch between literal XML and query expressions via curly braces.

In the context of SQL’s query capabilities, the most important expression in XQuery is the FLWOR (pronounced “flower”) expression, which is analogous to the SELECT-FROM-WHERE-ORDER BY query block in SQL. The components of a FLWOR are:

• A for clause that generates one or more value sequences, binding the values to one or more query variables, roughly similar to the FROM clause in SQL.

Figure 1: JDBC Driver and the AquaLogic Data Services Platform
• A let clause that binds a temporary variable to the result of a query expression, similar to temporary views (the WITH clause) in advanced SQL dialects.
• A where clause that contains Boolean predicates that restrict the FOR clause’s variable bindings, similar to the WHERE clause in SQL.
• An order by clause that contains a list of expressions that dictate the order of the XML output items, similar to SQL’s ORDER BY clause.
• A return clause that specifies the desired XML output. The return clause is analogous to the SELECT clause in SQL, but the structures that it can specify are much richer. (This clause is where the JSP-like XML node construction syntax of XQuery is heavily used.)

In general, an XQuery FLWOR expression can consist of any number of for and/or let clauses followed by an optional where clause, an optional order by clause, and a mandatory return clause.

2.2. SQL vs. XQuery

For data handling, XQuery has much of the richness of SQL and more – it includes support for subqueries, union, intersection, difference, aggregate functions, sorting, existential and universal quantification, conditional expressions, user-defined functions, and static and dynamic typing, in addition to various constructs to support document manipulation (e.g., query primitives for order-related operations). Being an expression language, XQuery is more composable (or “orthogonal”) in its grammatical design. We will see later that this compositability will be very useful in the SQL to XQuery translation process.

At the data artifact level, the XQuery data model is much richer than SQL’s data model (relations or tables, which are multisets of n-ary tuples or rows). Tables are flat, while XML data can be hierarchically structured, often nesting several levels deep. Tables are highly regular, while XML tends to be more varied; structural variations, typing variations, and missing data are more the norm than the exception with XML. Relational data is unordered, while order often has an important meaning in XML. Relational data is unordered, while order often has an important meaning in XML data (particularly for documents!). Tables have fairly static schemas, while XML schemas tend to be more extensible and the “self-describing” nature of XML blurs the data/meta-data distinction. Moreover, XML data may or may not even have an associated schema, while relational data always does.

2.2. Problem Scope

As explained in the Introduction, the goal of the AquaLogic DSP JDBC driver is to enable legacy SQL-based applications and tools, most notably reporting tools such as Crystal Reports or Business Objects, to be run against AquaLogic DSP data services. The JDBC driver provides a table-oriented view of (a subset of) the DSP world, providing SQL access and returning JDBC result sets rather than XQuery result sequences. This makes it possible for the data integration work done for SOA application purposes to be leveraged for reporting as well. It also allows the problem scope to be limited in several important ways:

1. SQL views are tabular, so the XML data to be queried via the JDBC driver must also be regular and “flat”. AquaLogic DSP only provides SQL access to data service functions that yield “flat” XML results as shown in Example 1, i.e., results where the XML looks like the obvious row-by-row serialization of a table.

   <ns0:CUSTOMERS>  
   <CUSTOMERID>55</CUSTOMERID>  
   <CUSTOMERNAME>Joe</CUSTOMERNAME>  
   </ns0:CUSTOMERS>  
   <ns0:CUSTOMERS>  
   <CUSTOMERID>23</CUSTOMERID>  
   <CUSTOMERNAME>Sue</CUSTOMERNAME>  
   </ns0:CUSTOMERS>  
   ...

   Example 1: Flat XML function result

2. XQuery 1.0 supports only reads, not updates. Reporting tools are read-only as well. Thus, only SQL SELECT statements need to be supported.

3. Different database products offer different dialects of SQL, but reporting tools and cross-database-platform applications usually take a least common denominator approach to their use of SQL. Thus, only SQL92 [2] needs to be supported

   It is important to note that the first simplification does not fundamentally limit the SQL-accessibility of information in AquaLogic DSP. Since it is possible to define new data services on top of other data services, one can always define additional, “flat” data service functions that normalize and expose the desired information for the purpose of JDBC access.

3. SQL to XQuery Translation

   We are now ready to describe how the AquaLogic DSP JDBC driver maps SQL requests into XQuery requests against data services. This involves presenting AquaLogic DSP artifacts as SQL database artifacts and translating queries written in SQL into equivalent expressions in XQuery that AquaLogic DSP can then compile and execute. The translator supports almost
all of the SELECT functionality of SQL-92; full details of its SQL-92 construct coverage can be found in [5].

3.1. Mapping of Queryable Artifacts

SQL SELECT statements refer to names of database catalogs, schemas, tables, and columns. These are the queryable artifacts in the relational database world. Before we can understand the translation of SQL to XQuery in the context of AquaLogic DSP, it is important to understand what the JDBC driver considers to be the analogues of these artifacts in the AquaLogic DSP world. We shall see that these analogies affect the way SQL statements are validated and XQuery expressions are produced.

The key artifacts in the AquaLogic DSP data world are applications, projects, data services, and data service functions. AquaLogic DSP applications are similar to databases in the SQL world, providing an accessible universe of information artifacts. An application can contain multiple projects, which can also contain additional folders, thereby providing a folding mechanism to organize the artifacts of the application. Data services [4], as mentioned earlier, are the most central concept in AquaLogic DSP. A data service is a collection of functions “about” a given business object. Lastly, data source functions are the actual targets (i.e., data sources) for queries.

For physical data services that provide access to relational data or Web services, the data service functions are derived automatically through a metadata import process. Physical data service functions are externally defined (i.e., they are opaque). In contrast, logical data services are authored by data service developers at application development time. A data service is captured as a .ds file, an XQuery file that contains definitions for each of a given data service’s functions. The body of each data service function for a logical data service is an XQuery written in terms of one or more lower-level data service function calls.

Example 2 shows the .ds file for a simple physical data service (with certain minor details omitted for simplicity). This .ds file is of the kind produced automatically when AquaLogic DSP is asked to import metadata from a relational data source like an Oracle database table called CUSTOMERS. A typical XQuery against the CUSTOMERS() data service function of Example 2 is shown in Example 3.

```
declare function f1:CUSTOMERS() 
    as schema-element(t1:CUSTOMERS)*
    external;
```

Example 2: Physical data service .ds file

Any data service function that returns a sequence of “flat” XML elements is a candidate for presentation through the JDBC driver. Example 2 is clearly such a function. Upon execution, its results will look like those depicted in Example 1. The function result is a sequence of XML elements, each one being called a CUSTOMERS element, that look like the rows of a table. Each CUSTOMERS element has sub-elements that look like the columns of the table.

```
import schema namespace ns0 =
    "ld:TestDataServices/CUSTOMERS" at
    "ld:TestDataServices/schemas/CUSTOMERS.xsd";
for $c in ns0:CUSTOMERS()
    where $c/CUSTOMERNAME eq "Sue"
    return
<RECORD>
    <CUSTOMERS.CUSTOMERID>
        {fn:data($c/CUSTOMERID)}
    </CUSTOMERS.CUSTOMERID>
    <CUSTOMERS.CUSTOMERNAME>
        {fn:data($c/CUSTOMERNAME)}
    </CUSTOMERS.CUSTOMERNAME>
</RECORD>
```

Example 3: Using a data service function

We are now ready to understand the way that AquaLogic DSP artifacts are mapped by the JDBC driver into analogous relational artifacts for use in SQL queries. As we have explained, an AquaLogic DSP application is organized as follows:

(i) An application name.
(ii) One or more projects.
(iii) Each project can contain folder hierarchies and data service (.ds) and schema (.xsd) files.

Figure 2 summarizes the analogies of AquaLogic artifacts with those of the SQL world presented by the JDBC driver. As shown in the figure:

(i) The AquaLogic DSP application name becomes the SQL catalog name.
(ii) The path to a .ds file plus the .ds file name becomes the SQL schema name.
(iii) The data service function name itself becomes a SQL table alias name if the function does not have any input parameters. (If a function has parameters, it becomes a callable SQL stored procedure.)
(iv) The simple-type child elements of the table element in the data service’s associated schema file become the SQL table’s column names. (More about this below.)
The most important AquaLogic DSP artifact is the data service function. A function declared within a ds file returns a typed XML element sequence upon execution. The JDBC driver treats such a function as a table and translates SQL statements over the table into XQueries over the function. Every data service function will have a return type which has been defined in an XML Schema definition (.xsd) file by the AquaLogic data service developer at DSP application development time.

In short, the JDBC driver treats AquaLogic DSP data service functions as relational tables. The return type of such functions must be flat XML, as previously discussed. Notice that the CUSTOMERS() function returns a CUSTOMERS* sequence type, which will actually lead to the returning of an array of CUSTOMERS elements at the AquaLogic client/server data passing boundary, as indicated in Figure 2. The CUSTOMERS element has two child elements, CUSTOMERID and CUSTOMERNAME, which map to columns in the CUSTOMERS table.

(i) Correctness: A SQL statement that has been translated into XQuery must be correct in terms of the semantics of both languages; the XQuery must do what the SQL query would have done, and all correct SQL queries must be translated.
(ii) Efficiency: In order to cater to intensive, ad hoc query environments, efficient translation methods must be employed.
(iii) Extensibility and Maintainability: The translator must be built to allow incremental additions of new SQL and/or new XQuery support over time.

Also worthy of mention at this point is one non-goal for this work: Generating optimized XQuery queries was not a goal. AquaLogic DSP has a fairly extensive query compilation and optimization component, so we decided early on that any/all optimizations should be left to the XQuery processor. The goal for the JDBC driver is simply to generate a proper XQuery that is equivalent to each SQL query. Further minimization and optimization steps are then performed at the XQuery level by the AquaLogic DSP query optimizer. However, generating “patterned” XQuery queries that readily facilitate later optimizations by the XQuery processor was kept in mind as a secondary goal.

3.2. Query Translation Goals

The design of the SQL to XQuery translator was driven by the following goals:

3.3. Query Translation Approach

The AquaLogic JDBC driver’s SQL to XQuery translator takes a component-based approach to translating SQL into XQuery. This allows a clean separation of translation tasks among the components.

3.4.1. Progressive, Step-wise Translation

SQL and XQuery have different semantics, so semantic information must be captured, stored and processed as needed to correctly translate each part of a SQL query into an equivalent XQuery expression. The translation is performed progressively in three stages:
(i) Validate the given SQL query and capture information related to SQL semantics.
(ii) Move pieces of semantic information from SQL-specific locations to XQuery-relevant locations.
(iii) Perform XQuery expression generation.

The first stage performs the SQL recognition and builds an abstract syntax tree of nodes representing the SQL query. The second and third stages use a tree-walker to traverse this abstract syntax tree. The second stage modifies the AST produced in stage-one, moving AST nodes to appropriate locations in the tree where the tree-walker of stage-three can use them in generating XQuery. Pieces of semantic information pertaining to SQL, originally stored by position in
stage-one, are moved to locations in the AST for use in generating a semantically correct XQuery.

The first two stages are preparatory. The actual XQuery generation is done only in the third stage. The syntax rules for SQL are applied in the first stage. The input SQL query is verified for syntactical correctness, and syntactically invalid SQL is rejected immediately. The result of the first stage of translation is an abstract syntax tree representing the input SQL query. At this stage, all of the context information useful for further processing is captured.

The abstract syntax tree created at the end of stage-one is a data structure with typed, heterogeneous nodes. Since these nodes are created to capture SQL semantics, their locations in the tree are SQL-inspired. In stage-two, nodes are moved to locations that are more relevant for consumption by stage-three during generation of XQuery. In stage-three, this transformed AST is traversed and, based on the context information in the nodes (see Section 3.5), the XQuery is generated piece by piece. Translated query snippets are stored in intermediate buffers and assembled as the translation proceeds, applying XQuery rules of syntax, position, casts, and scope on these snippets as needed.

3.4.2. Typed Components for Generation

When the translator parses the input SQL in stage-one, it generates an AST where each node is a typed node (i.e., a Java class instance) whose type is designed to correspond to some SQL abstraction. For example, the most fundamental abstraction in SQL is that of a relational view. Queries on tables, join operations between two queries or tables, set operations involving two queries, and even the tables themselves are all treated as views – i.e., virtual tables with rows and columns. The translator uses these abstractions within the AST. A typed view node is created for each query (or subquery), each join operation on two views, each set operation on two queries, and each table. We will refer to this typed view node as a resultset-node (RSN) for convenience. All RSNs are of the same type and represent a tabular view of data.

The use of typed AST nodes enables translation to proceed in chunks. For example, a query such as

\[
\text{SELECT * FROM CUSTOMERS INNER JOIN ORDERS ON CUSTOMERS.CUSTOMERID = ORDERS.CUSTID }
\]

operation (all columns, in this case). During XQuery generation, each RSN can be translated into its own XQuery expression and these expressions can be assembled to produce the final XQuery.

![Figure 3: Mapping of Resultset Nodes (RSNs)](image)

Using typed components also allows distribution of intelligence among components. For example, a join RSN is responsible for holding information about the tables that it is a join of. The join RSN should possess the knowledge of how to utilize its information and generate an XQuery expression for the join.

As another example, consider the query

\[
\text{SELECT * FROM (A JOIN (B JOIN C ON B.C1 = C.C2) AS P ON A.C1 = P.C1) }
\]

This query contains RSNs for two join RSNs, one being a child of the other. Here the child RSN is responsible for generating its own join XQuery expression. The parent join RSN simply delegates this task to the child. Figure 3 illustrates the mapping of RSNs to SQL views for a SQL query involving three tables, an inner join, two subqueries, and a union.

3.4.3. Query Contexts

Many semantic validations cannot be performed during the first stage. A SQL query such as

\[
\text{SELECT CUSTOMERID FROM CUSTOMERS }
\]

is valid only if
CUSTOMERID actually exists as a column in table CUSTOMERS. A query such as SELECT EMPNO FROM EMP GROUP BY EMPNAME is syntactically valid, but it is semantically incorrect because SQL mandates that a select-item, if it is a column, must be one of the group-by items if the query is a group-by query. Semantic validations require access to schema information and/or positional information in the AST, and during the first stage this is still under construction.

Syntax rules for validation can be applied during stage-one. The grammar for input SQL in stage-one drives syntactic validations on the input SQL. However, the semantic rules of the language are varied and many and cannot be applied during initial translation, as (i) the rules have to be applied based on the context, and (ii) the rules may be interdependent and may be complex. Semantic checks require context information which is captured during stage-one.

Query context information is organized according to the needs of SQL. For example, a query is a complete statement in SQL and it represents a view. A context is associated with every query, and a query containing a subquery will therefore have two contexts, one for the outer query and one for the inner query. For example, the example query depicted in Figure 4 has 3 contexts, one for the innermost query on CUSTOMERS returning CUSTOMERID as ID, one associated with the intermediate query on this view, and one associated with the outermost query returning all columns of this subquery.

![Query Contexts](image)

Examples of the information stored in contexts are (sub)query identification, the presence of aggregates, information about parent queries, and so on. The context provides a single point of access to all information about a particular query, such as the select-items in its SELECT clause, the conditions within its WHERE clause, and the ordering items in its ORDER BY clause.

3.5. Query Translation Process

We shall now discuss how the actual translation is performed. For the generation of XQuery, the following information is necessary:

(i) XQuery Function names and their locations

SQL table names map to data service functions. Generation of XQuery will require information about functions such as their names and locations for use in creating XQuery namespace imports and declarations in the query prolog.

(ii) Function return types and element metadata

SQL statement validation requires information about the columns of the table(s) being queried, including their names, data types and whether or not null values are permitted.

(iii) XQuery functions

Many SQL functions can be directly mapped to functions in the XQuery Functions and Operators library. The translator uses a preconfigured map of SQL and XQuery functions.

The information for (i) and (ii) above are obtained by querying the AquaLogie DSP application (using the remote metadata API, the details of which are outside the scope of this discussion). In addition, the following information is computed during translation:

(iv) Variable names and binding information

References to columns in a table become XPaths to corresponding elements in XML data returned by the function. As a result, all expressions in an XQuery over a data service function essentially contain XPaths to “column” elements in the XML. An XPath also contains the variable name of the “row” element in question. For example, suppose that a function getCustomerIds( ) returns a sequence of CUSTOMERS elements with CUSTOMERID as a child element. A translated XQuery snippet might look like Example 4.

```xquery
for $var1FR0 in ns0:getCustomerIds()
  return
  <RECORD>
    <ID>
      {fn:data($var1FR0/CUSTOMERID) }
    </ID>
  </RECORD>
```

Example 4: Function returning flat XML

Here, $var1FR0 is generated and is bound to the RSN of ns0:getCustomerIds( ). During stage-three, references to CUSTOMERID in the SQL are resolved to XPaths to the CUSTOMERID element using this variable name. XPath resolution also involves checking whether CUSTOMERID indeed exists as a child and whether any SQL alias names should affect the naming.
of the result element. The XPath for the SQL CUSTOMERID column is $var1FR0/CUSTOMERID.

(v) Expression datatypes

The translator computes the datatypes involved in the result of a SQL query. The columns of a SQL query projection are treated as expressions; the structure of expressions is captured in stage-one parsing, where expression trees are formed. An expression consists of column names, arithmetic operators, functions with arguments, and numerical values. (It may contain unbound variable names, if it is in the WHERE clause, and the variable values may be provided later using JDBC prepared statements. Here we limit our discussion to constants.) The datatypes of expressions are computed using a leaf-to-root, bottom-up approach on the abstract syntax tree of expressions. For numerical values, the datatype can be inferred from their representation, e.g., .1 is an integer and 5.6 is a double. For column names, the metadata fetched from the server provides the column datatype information. Given an expression combining these operands, the resulting datatype is inferred by applying the SQL rules of promotion and casting. The resulting expression datatype is mapped to a corresponding XQuery type and XQuery cast expressions are then generated.

SELECT * FROM CUSTOMERS

Example 5: A very simple SQL query

Suppose the very simple SQL query of Example 5 is to be translated into XQuery. The FROM items of SQL are translated into for expressions of XQuery. The filtering conditions in the WHERE clause of SQL are translated into where conditional expressions of XQuery. Similarly, SQL SELECT and ORDER BY map to XQuery return and order by expressions. XQuery 1.0 does not have group-by syntax, but BEA’s dialect of XQuery includes a group-by extension [5]. For translating the SQL group-by clause, we use the BEA group-by extension to XQuery.

Stage-one of the query translation process performs lexical analysis on the SQL statement, parses the tokens generated by the lexical analysis, and creates an AST, performing syntactic validations along the way. Figure 5 shows the AST resulting from stage-one. CTX0 is a marker context indicating the outermost query scope. CTX1 is the context associated with the query. CTX1 contains information about the query, such as access points to the list of columns in the projection, whether this query has aggregate functions in its projection, etc.

In stage-two, structural changes to this AST are made and the nodes are moved from their SQL positions into XQuery-relevant positions. The SQL column wildcard(*) indicates the return of all columns in the associated view. This could be translated into an XQuery $func/* returning all elements. However, this cannot always be done since SQL has the notion of range variables (alias names) on columns, which means that columns can be renamed. Instead, translation of the SQL statement SELECT CUSTOMERID ID, CUSTOMERNAME NAME FROM CUSTOMERS will result in an XQuery with the following expressions in its return:

  <ID> {fn:data($func/CUSTOMERID) } </ID>
  <NAME>{fn:data($func/CUSTOMERNAME)}</NAME>

Note that CUSTOMERID and CUSTOMERNAME are renamed to ID and NAME, the two SQL aliases.

Returning to our example, the column wildcard needs to be expanded, i.e., actual column information must be substituted for the column-wildcard. Metadata
for the involved tables is fetched from the AquaLogic DSP application to which the JDBC driver is connected using AquaLogic DSP’s metadata API. Fetched table metadata is cached locally for further use (more on that later). For each column of the involved tables, corresponding column nodes are added to the AST. The tree is thus prepared for consumption by stage-three, where serialization into XQuery will take place. Figure 5 shows the result of stage-two.

Stage-three uses a tree-walker to traverse the result of stage-two and serialize it into XQuery. Each RSN translates itself into an XQuery expression using information from the associated query contexts.

As shown in Figure 7, a SQL FROM clause translates into an XQuery for expression. In our example, the translated XQuery snippet should be:

```xml
for $var1FR0 in ns0:CUSTOMERS()

Svar1FR0 is a generated XQuery variable name associated with CUSTOMERS. For readability and debugging ease, the nomenclature of variable naming is based on the following: var – a common prefix, followed by the query context id (computed during stage-one), followed by the query zone and a unique number within that zone. The query zone is described as a window on the SQL query under consideration. In this case, since the SQL FROM clause is being evaluated, the query zone is FR.

Reference to the CUSTOMERS() function is made using the namespace ns0 defined at the beginning of the query. When table metadata is fetched from the AquaLogic DSP application running on the server, the information about function schemas and namespaces is also returned. This is used to create imports such as:

```xml
import schema namespace ns0 = "ld:TestDataServices/CUSTOMERS" at "ld:TestDataServices/schemas/CUSTOMERS.xsd";
```

Once the for expression is written, the return clause needs to be created. Since the function is bound to the variable $var1FR0, the return expressions are generated based on this variable. For example:

```xml
return
<RECORD>
<ID>{fn:data($var1FR0/CUSTOMERID)}</ID>
</RECORD>
```

Creating this return clause requires verifying that CUSTOMERID is indeed a column in the CUSTOMERS view, i.e., that CUSTOMERID is a child element in the return schema of CUSTOMERS(). This is easy since we have the table metadata available.

![Figure 7: Translation in Stage-three](image)

The simple SQL query that we started with is thus ultimately translated into the XQuery of Example 6.

```xml
import schema namespace ns0 = "ld:TestDataServices/CUSTOMERS" at "ld:TestDataServices/schemas/CUSTOMERS.xsd";

<RECORDSET>
{
  for $var1FR0 in ns0:CUSTOMERS() return
  <RECORD>
    <ID>{fn:data($var1FR0/CUSTOMERID)}
    </ID>
  </RECORD>
}
</RECORDSET>
```

**Example 6: Final result of translation**

```sql
SELECT INFO.ID, INFO.NAME
FROM (SELECT CUSTOMERID ID, CUSTOMERNAME NAME
      FROM CUSTOMERS) AS INFO
WHERE INFO.ID > 10
```

**Example 7: SQL with subquery**

Now let us consider Example 7, with a subquery, to understand how SQL rules are applied during XQuery generation. The example query contains two views: the
CUSTOMERS table view and the outer query on this view. The translator maps a query view to an XQuery let. Thus, a FLWOR involving the CUSTOMERS() function is bound to a let variable ($tempvar1FR2). For further querying on this view, the XQuery FLWOR involves the let variable $tempvar1FR2.

<RECORDSET>
  {  let $tempvar1FR2 :=
      <RECORDSET>
        {  for $var2FR2 in ns0:CUSTOMERS()
            return
            <RECORD>
              <ID>
                {fn:data($var2FR2/CUSTOMERID)}
              </ID>
              <NAME>
                {fn:data($var2FR2/CUSTOMERNAME)}
              </NAME>
            </RECORD>
        }
      </RECORDSET>
  for $var1FR2 in $tempvar1FR2/RECORD
  where ($var1FR2/ID>xs:integer(10) )
  return
  <RECORD>
    <INFO.ID>
      {fn:data($var1FR2/ID) }
    </INFO.ID>
    <INFO.NAME>
      {fn:data($var1FR2/NAME)}
    </INFO.NAME>
  </RECORD>
}  }  </RECORDSET>

Example 8: Subquery translation

The outer SQL query projection in Example 7 consists of two columns, INFO.ID and INFO.NAME, derived from the subquery on CUSTOMERS. SQL92 mandates that any references to columns in a subquery must either be unqualified – in which case they should be unambiguously resolved – or must be correctly qualified with a query range variable (alias). In our case, INFO.ID and INFO.NAME are both qualified with the range variable name of subquery, INFO.

As mentioned earlier, the query context is the central point of access to query information during translation. When the context receives an XPath resolution request for INFO.ID, it delegates the request to the RSNs of the query. In our case, the RSN is that of a subquery named INFO. Since this RSN is of subquery-type, it applies the SQL rules regarding name qualification on the request, and it checks if the requested column belongs to itself. Here INFO.ID is found to be returned from the subquery projection. Since all tuple-set abstractions in SQL are mapped to XQuery lets, and all lets have their bound variables generated, an XPath for an element inside a let can easily be written. In our case, it is $var1FR2/ID, where $var1FR2 is the generated variable name for the subquery in the let expression. Example 8 shows the final translation to XQuery of SQL in Example 7.

Let us now consider another important and illustrative example, one that involves a left outer join of two tables.

SELECT CUSTOMERS.CUSTOMERID,
    PAYMENTS.PAYMENT
FROM CUSTOMERS LEFT OUTER JOIN PAYMENTS
ON CUSTOMERS.CUSTOMERID=PAYMENTS.CUSTID

Example 9: Left outer join

The SQL statement in Example 9 returns all customers from the CUSTOMERS view together with any related payments from the PAYMENTS view. Example 10 gives the XQuery translation result for this query.

import schema namespace ns0 =
  "ld:TestDataServices/CUSTOMERS" at
  "ld:TestDataServices/schemas/CUSTOMERS.xsd";
import schema namespace ns1 =
  "ld:TestDataServices/PAYMENTS" at
  "ld:TestDataServices/schemas/PAYMENTS.xsd";
<RECORDSET>
  {  let $tempvar1FR4 :=
      <RECORDSET>
        {  for $var1FR2 in ns0:CUSTOMERS()
            let $tempvar1FR3 :=ns1:PAYMENTS()
              [($var1FR2/CUSTOMERID=CUSTID)]
            return
            if (fn:empty($tempvar1FR3) ) then
              (  <RECORD>
                  <CUSTOMERS.CUSTOMERID>
                    {fn:data($var1FR2/CUSTOMERID)}
                  </CUSTOMERS.CUSTOMERID>
                  <CUSTOMERS.CUSTOMERNAME>
                    {fn:data($var1FR2/CUSTOMERNAME)}
                  </CUSTOMERS.CUSTOMERNAME>
                </RECORD>
            )
            else
              (  for $var1FR3 in $tempvar1FR3
                  return
                  <RECORD>
                    <CUSTOMERS.CUSTOMERID>
                      {fn:data($var1FR2/CUSTOMERID) }
                    </CUSTOMERS.CUSTOMERID>
                    <CUSTOMERS.CUSTOMERNAME>
                      {fn:data($var1FR2/CUSTOMERNAME)}
                    </CUSTOMERS.CUSTOMERNAME>
                    <PAYMENTS.CUSTID>
                      {fn:data($var1FR3/CUSTID)}
                    </PAYMENTS.CUSTID>
                    <PAYMENTS.PAYMENT>
                      {fn:data($var1FR3/PAYMENT)}
                    </PAYMENTS.PAYMENT>
                  </RECORD>
              )
            )
          }
        </RECORDSET>
  for $var1FR4 in $tempvar1FR4/RECORD
  return
  <RECORD>
    <CUSTOMERS.CUSTOMERID>
      {fn:data($var1FR4/CUSTOMERS.CUSTOMERID)}
    </CUSTOMERS.CUSTOMERID>
  </RECORD>
</RECORDSET>
Example 10: Left outer join translation

In translating this query, the outer join condition CUSTOMERS.CUSTOMERID = PAYMENTS.CUSTID is translated into the XPath filter expression \[\!(\!$var1FR2/CUSTOMERID=CUSTID\!)\!\]. The join RSN uses this translated filter expression to produce an if-then-else-based left outer join expression (bound to let variable $tempvar1FR4) based on information about columns from its involved table RSNs. Finally, the query RSN composes this XQuery expression to create a return expression.

Finally, let us consider a more complex example that shows how grouping, ordering, scalar functions, and aggregate functions are all handled during translation. Consider the SQL query in Example 11. This query is translated into XQuery as shown in Example 12 (with prologue removed for simplicity).

Example 11: A complex query

```xml
<RECORDSET>
  
  <RECORD>
    
    <RECORDSET>
      
      for $var1FR2 in ns0:CUSTOMERS()
      for $var1FR3 in ns1:PO_CUSTOMERS()
      where ($var1FR2/CUSTOMERID = $var1FR3/CUSTOMERID)
      return
      
      <PO_CUSTOMERS.ORDERID> $var1FR3/ORDERID <RECORD>
      <PO_CUSTOMERS.CUSTOMERID> $var1FR3/CUSTOMERID <RECORD>
      <PO_CUSTOMERS.CUSTOMERID> $var1FR3/CUSTOMERID <RECORD>
      <PO_CUSTOMERS.CUSTOMERID> $var1FR3/CUSTOMERID <RECORD>
      <PO_CUSTOMERS.ORDERID> $var1FR3/CUSTOMERID <RECORD>
      <PO_CUSTOMERS.CUSTOMERID> $var1FR3/CUSTOMERID <RECORD>
      <PO_CUSTOMERS.CUSTOMERID> $var1FR3/CUSTOMERID <RECORD>
      </RECORDSET>
  
  for $varNewlet1 in $inter/RECORD
</RECORDSET>
```

Example 12: Translation of a complex query

In this example, the RSNs of the CUSTOMERS and PO_CUSTOMERS tables translate into a join (double for) with the condition CUSTOMERS.CUSTOMERID = PO_CUSTOMERS.CUSTOMERID, and their result is bound to the let variable $inter. Grouping is then performed using BEA's extension to the XQuery language. $inter is partitioned over CUSTOMERID and CUSTOMERNAME and the new groups are called var1GB4 and var1GB5 respectively. The group results are then ordered before return.

Notice that fn:concat (the XQuery equivalent of the SQL CONCAT function) takes the partition $var1Partition1 as an argument while fn:count uses var1GB4. This happens because of how the translation logic is distributed among the typed nodes in the AST. Based on the information stored in the context, different typed nodes in the AST translate themselves conditionally into XQuery expressions.

4. Result Handling

We now turn to the problem of mapping XQuery results into JDBC results. In a general setting, the use of XML as a format for data exchange has a number of advantages. However, if query results are transmitted as XML only to be converted into the required JDBC result set form, materializing and parsing XML on the client side imposes unnecessary overhead in terms of memory and processing resources in the JDBC driver.

Through initial prototyping, we found that performance could be measurably improved if we
replaced XML as the return type for translated XQuery expressions with a more compact format for conversion into JDBC result set form. The result data is actually returned as text interspersed with column and row separators, and it can then be parsed (using computed result schema information) to create a JDBC resultset. For example, an XQuery might return a text-encoded result such as the following:

```xml
>987654<Acme Widget Stores
>987655<Supermart
>987656<Ajax Distributors
>987657<Zenith Parts and Service
```

To generate such a result, the translator actually generates a translated XQuery such as this:

```xml
import schema namespace ns0 = "ld:TestDataServices/CUSTOMERS" at "ld:TestDataServices/schemas/CUSTOMERS.xsd";
fn:string-join
  let $actualQuery :=
    <RECORDSET>
      for $var1FR0 in ns0:CUSTOMERS()
        return
          <RECORD>
            <CUSTOMERS.CUSTOMERID>
              {fn:data($var1FR0/CUSTOMERID)}
            </CUSTOMERS.CUSTOMERID>
            <CUSTOMERS.CUSTOMERNAME>
              {fn:data($var1FR0/CUSTOMERNAME)}
            </CUSTOMERS.CUSTOMERNAME>
          </RECORD>
    </RECORDSET>
  for $tokenQuery in $actualQuery/RECORD
    return
      ("">",
      (fn-bea:if-empty(
        fn-bea:xml-escape(
          fn-bea:serialize-atomic(
            fn:data(
              $tokenQuery/CUSTOMERS.CUSTOMERID)
          ),""""))
      )
      (fn-bea:if-empty(
        fn-bea:xml-escape(
          fn-bea:serialize-atomic(
            fn:data(
              $tokenQuery/CUSTOMERS.CUSTOMERNAME)
          ),""""))
      ,""'),"

Notice here how the original query is wrapped with another query that returns string data interspersed with column and row delimiters ("""" and """" characters, respectively). Creating a wrapper query around the original query allows us to maintain a clean separation between JDBC result handling logic and the more complex SQL to XQuery translation logic.

5. Conclusion

In this paper we have motivated and described the design of the AquaLogic DSP JDBC driver. With its SQL92 to XQuery translator, this driver provides a way for legacy applications and tools to access integrated XML data services using SQL and JDBC. It thus connects the new XML world with the older relational world. Applications that use SQL for querying data, notably reporting tools such as Crystal Reports and Business Objects, can now also enjoy access to data from heterogeneous sources exposed as XML by the AquaLogic Data Services Platform.

6. Bibliography