Bapat Road, Pune 411016 (IN). DIXIT, Kanad, Pravin [IN/IN]; Flat No. 5, Aviraj Society, Survey No. 163-5 Dp Road, Pune, Maharashtra 411007 (IN). THATTE, Sachin, Sadashiv [US/US]; 2669 Thrus Court, San Jose, CA 95125 (US). BORKAR, Vinayak, Ravindra [IN/US]; 4730 Cielo Vista Way, San Jose, CA 95129 (US). MALKANI, AnU [US/US]; 52 Mission Cielo Avenue, Fremont, CA 94539 (US).


[Continued on next page]

Abstract: In accordance with embodiments, there are provided mechanisms and methods for translating Structured Query Language (SQL) queries into extended Markup Language (XML) query language to allow Service Oriented Architecture (SOA) compliant applications to interact with SQL based information systems. As shown in Figure 2A, determining from Data Services Platform artifacts, a correspondence with SQL database artifacts (block 202). Then, at least one SQL query is translated into at least one equivalent expression in XQuery based upon the determined correspondence (block 204). The Data Services Platform can then compile and execute the XQuery.
Designated States (unless otherwise indicated, for every kind of regional protection available): ARIPO (BW, GH, GM, KE, LS, MW, MZ, NA, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IS, IT, LT, LU, LV, MC, NL, PL, PT, RO, SE, SI, SK, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Published:

— with international search report

For two-letter codes and other abbreviations, refer to the “Guidance Notes on Codes and Abbreviations” appearing at the beginning of each regular issue of the PCT Gazette.
SYSTEM AND METHOD FOR TRANSLATING SQL QUERIES INTO XML QUERY LANGUAGE

CLAIM OF PRIORITY


FIELD OF THE INVENTION

The current invention relates generally to interfaces for Service Oriented Architecture (SOA), and more particularly to a mechanism for translating Structured Query Language (SQL) queries into extended Markup Language (XML) query language for use with Service Oriented Architectures.

BACKGROUND

Increasingly, enterprises are looking for ways to simplify access and organization of Information Technology (IT) services. One mechanism for providing such IT simplification is Service Oriented Architecture (SOA). Service Oriented Architectures include treating services as building blocks in an enterprise's IT installation, and define interfaces for moving information among these building blocks. Application of SOA principles promises faster development cycles, increased reusability and better change tolerance for software components.

Unfortunately, enterprises that implement SOA often find that the start-up complexities of SOA delays, if not derails, the expected return on investment. While SOA simplifies the complexity of an IT environment, organizations lack sufficient experience with SOA technology required for a quick, trouble-free implementation.

One area of SOA that receives significant developer time and attention is the perceived need to make applications and services of the SOA world work with so-named legacy systems. For example, many enterprises have significant investments in information
systems based upon tools for which structured query language (SQL) is the requisite interface. Accordingly, there is an ongoing need for improved techniques for enabling greater numbers of SOA applications to interact with SQL based information systems.

BRIEF DESCRIPTION OF THE DRAWINGS

FIGURES 1A-1B are high-level overviews illustrating service oriented architecture enabled computing embodiments in which techniques for providing an SQL interface for XQuery based service architecture may be implemented.

FIGURES 2A - 2C are operational flow diagrams illustrating a high level overview of a technique for translating Structured Query Language (SQL) queries into extended Markup Language (XML) query language in an embodiment.

FIGURES 3A - 3F are high-level overviews illustrating an example of an SQL to XML query language translator embodiment.

FIGURE 4 is a hardware block diagram of an example computer system, which may be used to embody one or more components in an embodiment.

DETAILED DESCRIPTION

The invention is illustrated by way of example and not by way of limitation in the figures of the accompanying drawings in which like references indicate similar elements. References to embodiments in this disclosure are not necessarily to the same embodiment, and such references mean at least one. While specific implementations are discussed, it is understood that this is done for illustrative purposes only. A person skilled in the relevant art will recognize that other components and configurations may be used without departing from the scope and spirit of the invention.

In the following description, numerous specific details are set forth to provide a thorough description of the invention. However, it will be apparent to those skilled in the art that the invention may be practiced without these specific details. In other instances, well-known features have not been described in detail so as not to obscure the invention.

Although a diagram may depict components as logically separate, such depiction is merely for illustrative purposes. It can be apparent to those skilled in the art that the components portrayed can be combined or divided into separate software, firmware and/or
hardware components. For example, one or more of the embodiments described herein can be implemented in a network accessible device/appliance such as a router. Furthermore, it can also be apparent to those skilled in the art that such components, regardless of how they are combined or divided, can execute on the same computing device or can be distributed among different computing devices connected by one or more networks or other suitable communication means.

In accordance with embodiments, there are provided mechanisms and methods for translating Structured Query Language (SQL) queries into extended Markup Language (XML) query language. These mechanisms and methods can enable embodiments to provide SQL based applications the capability to interact with SOA compliant information systems. The ability of embodiments to provide interaction between SQL applications and SOA based information systems can enable easier expansion of SOA based architectures. Some embodiments can provide improved correctness, i.e., that an SQL statement that has been translated into XQuery language for use with SOA information systems be correct in terms of the semantics of both languages. Embodiments can insure that the XQuery does what the SQL query would have done, and that correct SQL queries are translated. Some embodiments can provide improved efficiency, i.e., efficient translation methods are employed in order to function in intensive, ad hoc query environments. Some embodiments can provide improved extensibility and maintainability, i.e., the translator allows incremental additions of new SQL and/or new XQuery support over time.

As used herein, the term service is intended to be broadly construed to include any application, program or process resident on one or more computing devices capable of providing services to a requestor or other recipient, including without limitation network based applications, web based server resident applications, web portals, search engines, photographic, audio or video information storage applications, e-Commerce applications, backup or other storage applications, sales/revenue planning, marketing, forecasting, accounting, inventory management applications and other business applications and other contemplated computer implemented services. As used herein, the term application is intended to be broadly construed to include any data entry, update, query or program that processes data on behalf of a user. Users may be human or computational entities. As
used herein, the term artifact is intended to be broadly construed to include any concept that can be used to describe functions or properties of a given computational object.

FIGURES 1A - 1B are high-level overviews illustrating service oriented architecture enabled computing embodiments in which techniques for providing an SQL interface for XQuery based service architecture may be implemented. As shown in FIGURE 1A, a Data Services Platform (DSP) 104 is used to provide a mechanism by which a set of applications, or application portals 94, 96, 98, 100 and 102, can integrate with, or otherwise access in a tightly couple manner, a plurality of services. Such services may include, without limitation, a Materials Requirements and Planning (MRP) system 112, a purchasing system 114, a third-party relational database system 116, a sales forecast system 118 and a variety of other data-related services (not shown). In an embodiment, an SQL based system 120 interfaces with Data Services Platform 104 via a Java Data Base Connectivity (JDBC) driver 111. (The Java Database Connectivity (JDBC) API is a standard SQL database access interface.) Such SQL based services may include, without limitation, Crystal Reports and Business Objects, both products of Business Objects S.A. of Paris France, Microsoft Access, a product of Microsoft Inc. of Redmond Washington, DB Visualizer, a product of Minq Software of Stockholm Sweden and others. In an embodiment, JDBC driver 111 implements a technique for presenting Data Services Platform artifacts (applications, projects, data services, and data service functions, etc.) as SQL database artifacts (catalogs, schemas, tables, columns, etc.) and translating JDBC queries written in SQL received from SQL based system 120 into equivalent expressions in XQuery that can be compiled and executed by the Data Services Platform 104. Such techniques will be described in further detail herein with reference to FIGURES 2A - 3F. Although not shown in FIGURE 1A for clarity, in one embodiment, one or more of the services may interact with one or more other services through the Data Services Platform 104 as well.

Internally, the Data Services Platform 104 employs an integration engine 110 to process requests from the set of portals 94 - 102 (and JDBC driver 111) to the services 112 - 118. The integration engine 110 allows access to a wide variety of services, including data storage services, server-based or peer-based applications, Web services, services 112 - 118 and any other services capable of being delivered by one or more computational
devices are contemplated in various embodiments. A services model 108 provides a structured view of the available services to the application portals 94 - 102. As used herein, the term view is intended to be broadly construed to include any mechanism that provides a presentation of data and/or services in a format suited for a particular application, service, client or process. The presentation may be virtualized, filtered, molded, or shaped. For example, data returned by services to a particular application (or other service acting as a requestor or client) can be mapped to a view associated with that application (or service). Embodiments can provide multiple views of available services to enable organizations to compartmentalize or streamline access to services, increasing the security of the organization's IT infrastructure.

In one embodiment, the services model 108 provides a plurality of views 106 that may be filtered, molded, or shaped views of data and/or services into a format specifically suited for each portal application 94 - 102. In one embodiment, data returned by services to a particular application (or other service acting as a requestor or client) is mapped to the view 106 associated with that application (or service) by Data Services Platform 104. Embodiments providing multiple views of available services can enable organizations to compartmentalize or streamline access to services, thereby increasing the security of the organization's IT infrastructure. In one embodiment, services model 108 may be stored in a repository 122 of service models. Embodiments providing multiple services models can enable organizations to increase the flexibility in changing or adapting the organization's IT infrastructure by lessening dependence on service implementations.

FIGURE 1B is a high level schematic of an integration engine 110 illustrated in FIGURE 1A with reference to one example embodiment. As shown in FIGURE 1B, the integration engine 110 includes an interface processing layer 140, a query compilation layer 150 and a query execution layer 160. The interface layer 140 includes a request processor 142, which takes the request 10, typically received from the portal applications 94 - 102 or from SQL based system 120 via JDBC driver 111, and processes this request into an XML query 50. In the event that the request is from an SQL based service 120, the request is forwarded via the JDBC driver 111 to an SQL/XQuery translator 144, which exposes an interface for accepting SQL queries as input and returns translated XQueries corresponding to the SQL input. In an embodiment, JDBC driver 111 processes Data
Services Platform schemas to create JDBC compatible metadata for use by the SQL/XQuery translator. SQL/XQuery translator uses this metadata to translate SQL query into one or more XQuery queries. Techniques for translating SQL into XQuery will be described below in greater detail with reference to FIGURES 2A - 3F. Ultimately, the interface layer sends the XML query to the query compilation layer.

Within the query compilation layer, a query parsing and analysis mechanism receives the query from the client applications, parses the query and sends the results of the parsing to a query rewrite optimizer. The query rewrite optimizer determines whether the query can be rewritten in order to improve performance of servicing the query based upon one or more of execution time, resource use, efficiency or other performance criteria. The query rewrite optimizer may rewrite or reformat the query based upon input from one or more of a source description and a function description if it is determined that performance may be enhanced by doing so. In one embodiment, a goal of 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 query rewrite optimizer. However, in another embodiment, generating "patterned" XQuery queries that readily facilitate later optimizations by the query rewrite optimizer is implemented as a second goal of the JDBC driver. A runtime query plan generator generates a query plan for the query provided by the query rewrite optimizer based upon input from one or more of the source description and the function description.

The query compilation layer passes the query plan output from the runtime query plan generator to a runtime query engine in the query execution layer. The runtime query engine is coupled with one or more functions that may be used in conjunction with formulating queries and fetch requests to sources, which are passed on to the appropriate service(s). The service responds to the queries and fetch requests with results from sources. The runtime query engine of the query execution layer translates the results into a format usable by the client or portal application, such as without limitation XML, in order to form the XML query results.

Before responses or results are passed back to the client or portal application making the request, a query result filter in the interface layer determines what
portion of the results will be passed back to the client or portal application and in what form or format, forming a response 58. In the event that the results must be formatted into an SQL compatible table, query result filter 146 prepares the result into SQL compatible format based upon the SQL schema metadata 60. When properly formed, the response is returned to the calling client or portal application.

While the present invention is described with reference to an embodiment called AquaLogic® by BEA systems of San Jose, California, which implements executable programs written in the Java™ programming language and usually (though not always) executed in response to an invocation from an HTML page, the present invention is not limited the AquaLogic Product, nor to the Java™ programming language and may be practiced using other programming languages, i.e., JSP and the like without departing from the scope of the embodiments claimed. (Java™ is a trademark of Sun Microsystems, Inc.)

FIGURE 2A is an operational flow diagram illustrating a high level overview of a technique for translating Structured Query Language (SQL) queries into extended Markup Language (XML) query language in an embodiment. The technique for translating Structured Query Language (SQL) queries into extended Markup Language (XML) query language shown in FIGURE 2A is operable within a JDBC driver, such as JDBC driver 111 of FIGURE 1A, for example. As shown in FIGURE 2A, determining from Data Services Platform artifacts, a correspondence with SQL database artifacts (block 202).

Then, at least one SQL query is translated into at least one equivalent expression in XQuery based upon the determined correspondence (block 204). The Data Services Platform can then compile and execute the XQuery.

FIGURE 2B is an operational flow diagram illustrating a high level overview of a technique for determining from Data Services Platform artifacts, a correspondence with SQL database artifacts in an embodiment. As shown in FIGURE 2B, determining from Data Services Platform artifacts, a correspondence with SQL database artifacts includes mapping queryable SQL artifacts to corresponding data services platform artifacts (block 212). Queryable SQL artifacts shall be defined to include, without limitation, names of database catalogs, schemas, tables, columns and any other types of artifacts that can be queried using for example an SQL SELECT statement. Data Services Platform artifacts shall be defined to include, without limitation, applications, projects, data services, data
service functions and any other types of artifacts. Artifacts include anything that can be used to describe functions or properties "about" a given computational object.

FIGURE 2C is an operational flow diagram illustrating a high level overview of a technique for translating SQL query into at least one equivalent expression in XQuery based upon the determined correspondence in an embodiment. As shown in FIGURE 2C, information related to SQL semantics is captured (block 222). At least a portion of the information related to SQL semantics is moved from SQL-relevant locations within the query to XQuery-relevant locations (block 224). An XQuery expression is generated (block 226).

A technique for mapping SQL requests into XQuery requests against data services will now be described in greater detail with reference to Data Services Platform 104 and JDBC driver 111 in an example embodiment. This technique involves determining from Data Services Platform artifacts, a correspondence with SQL database artifacts and translating JDBC queries written in SQL into equivalent expressions in XQuery that the Data Services Platform can then compile and execute based upon the determined correspondence.

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, hi an embodiment, the JDBC driver 111 considers analogies of these artifacts in the Data Services Platform 104. These analogies affect the way SQL statements are validated and XQuery expressions are produced. Artifacts in the Data Services Platform data world include, without limitation, applications, projects, data services, and data service functions.

Data Services Platform 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 foldering mechanism to organize the artifacts of the application. Data services, as mentioned earlier, are a central concept in Data Services Platform. A data service is a collection of functions
"about" a given business object. Last but not least, data source functions are the actual targets (i.e., data sources) for queries.

For physical data services such as those that provide access to relational data or Web services, the data service functions are derived automatically via a data source metadata import process. In one embodiment, physical data service functions may be externally defined (i.e., they are opaque), in contrast, logical data services maybe explicitly authored by data service developers at application development time. In an embodiment, a data service is captured as a .ds file, which is an XQuery file that contains definitions of all of its data service 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.

Illustrated below as Example 1, a .ds file for a simple physical data service (with certain annotation details removed for simplicity) in an embodiment is of the kind produced automatically in Data Services Platform responsive to a request to import metadata from a relational data source such as an Oracle™ database containing a table called CUSTOMERS. A typical XQuery against the data service function CUSTOMERS() of Example 1 is shown in Example 2.

```
declare namespace fl = "lchTestDataServices/CUSTOMERS" ;
import schema namespace tl = "Id:TestDataServices/CUSTOMERS " at "IdiTestDataServices/schemas/CUSTOMERS .xsd";

declare function fl:CUSTOMERS ()
   as schema-element (tl:CUSTOMERS) *
   external ;
```

Example 1: 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 1 is clearly such a function. Upon execution, its results will look like those depicted in Example 2. 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 nsO =

"Id : TestDataServices/CUSTOMERS" at "Id : testDataServices/schemas/CUSTOMERS.xsd";

for $c in nsO : 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 2: XQuery against the data service function CUSTOMERS() of Example 1

In an embodiment, Data Services Platform artifacts are mapped by the JDBC driver into analogous relational artifacts for use in SQL queries. An Data Services Platform application embodiment may be organized as follows: (i) An application name; and (ii) One or more projects. Each project can contain folder hierarchies and data service (.ds) and schema (.xsd) files. FIGURES 3A - 3B are high-level overviews illustrating an example of an SQL to XML query language translator embodiment.

FIGURE 3A illustrates a high-level overview of analogies of artifacts with those of the SQL world presented by the JDBC driver in an embodiment. As shown in the FIGURE 3A, the Data Services Platform application name becomes the SQL catalog name 302. The path to a .ds file plus the .ds file name becomes the SQL schema name 304. The data service function name itself becomes a SQL table alias name if the function does not have any input parameters 306. (If a function has parameters, it becomes a callable SQL stored procedure.) The simple-type child elements of the table element in the data service's associated schema file become the SQL table's column names 308.

One Data Services Platform 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. Each data service function will have a return type that has been defined in an XML Schema definition (.xsd) file by the data service developer at application development time.
In short, the JDBC driver treats Data Services Platform data service functions as relational tables. The return type of such functions must be flat XML, in an embodiment. For example, the CUSTOMERS( ) function returns a CUSTOMERS* sequence type, which will actually lead to the returning of an array of CUSTOMERS elements at the Data Services Platform - client/server boundary, as indicated in FIGURE 3A. The CUSTOMERS element has two child elements, CUSTOMERID and CUSTOMERNAME, which map to columns in the CUSTOMERS table.

2. PROGRESSIVE, STEP-WISE TRANSLATION

SQL and XQuery have different semantics, so semantic information is captured, stored and processed as needed to correctly translate each part of a SQL query into an equivalent XQuery expression. In an embodiment, 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-relevant locations to XQuery-relevant locations; and (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 (AST). 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.

In an embodiment, 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. As used herein, the term "syntactic validation" is intended to mean checking whether the SQL query meets rules of syntax for SQL. The result of the first stage of translation is an abstract syntax tree representing the input SQL query. At this stage, the context information useful for further processing has been 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; 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. 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, a 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 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. This typed view node may be referred to as a result set-node (RSN). The 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

```sql
SELECT * FROM CUSTOMERS INNER JOIN ORDERS ON CUSTOMERS.CUSTOMERID = ORDERS.CUSTID
```

contains four (4) views with an RSN for each view: the CUSTOMERS table view, the ORDERS table view, the join view representing the result of inner join operation between the CUSTOMERS and ORDERS tables, and the query view representing the selected rows and columns from the result of the join 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.

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 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 3B illustrates the mapping of RSNs to SQL views for a SQL query involving three tables, an inner join, two subqueries, and a union in an embodiment.

4. QUERY CONTEXTS

Many semantic validations that cannot be performed during the first stage are reserved for later stages. A SQL query such as SELECT CUSTOMERJD FROM CUSTOMERS is valid only if CUSTOMERDD 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 are 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 3C has three (3) contexts, one for the innermost query on CUSTOMERS returning CUSTOMER© as ID, one associated with the intermediate
query on this view, and one associated with the outermost query returning all columns of this subquery.

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.

5. QUERY TRANSLATION PROCESS

Mechanisms for performing the actual translation will be discussed with reference to an example embodiment. For the generation of XQuery, the following information is used: (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, datatypes and nullability. (iii) XQuery functions. Many SQL functions can be directly mapped to functions in the XQuery Functions and Operators library. A translator embodiment uses a preconfigured map of SQL and XQuery functions. The information for (i) and (ii) above may be obtained by querying the Data Services Platform application, hi one embodiment, a remote metadata APIs used to provide the information for (i) and (ii). In addition, the following information is computed during translation: (iv) Variable names and binding information and (v) Expression datatypes.

References to columns in a table become XPaths to corresponding elements in XML data returned by the function. As a result, the 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 3.
for $varlFR0 in nsO:getCustomerIds()
return
  <RECORD>
  <ID>{fn:data($varlFR0/CUSTOMERID) }
  </ID>
  </RECORD>

Example 3: Function returning flat XML

Here, $varlFR0 is generated and is bound to the RSN of nsO:getCustomerIds(). During stage-three, references to CUSTOMERDD in the SQL are resolved to XPaths to the CUSTOMER 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 $varlFR0/CUSTOMERID.

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 comprises of column names, arithmetic operators, functions with arguments, and numerical values. Expressions may contain unbound variable names, if it is in the WHERE clause, and the variable values may be provided later using JDBC prepared statements. However, discussion of the present embodiment will be limited to constants for brevity. 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 4: A very simple SQL query

In an example, the SQL query of Example 4 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 one implementation of XQuery provided by BEA Systems, Inc. of San Jose, California includes a group-by extension. For translating the SQL group-by clause, the group-by extension made to XQuery is used.

Stage-one of translation performs lexical analysis on this SQL statement, parses the tokens generated by lexical analysis, and creates an AST, performing syntactic validations along the way. As used herein, the term "lexical analysis" is used to denote the process of taking an input string of characters (such as the source code of a computer program) and producing a sequence of symbols called "lexical tokens", or just "tokens", which may be handled more easily by a parser. FIGURE 3D shows the AST created at the end of stage-one. CTXO is a marker context indicating the outermost query scope. CTXI is the context associated with the query. CTXI 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:

```xml
<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 the 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 Data Services Platform application to which the JDBC
driver is connected using Data Services Platform's metadata API. In an embodiment,
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 3E 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 3F, a SQL FROM clause translates into an XQuery for
expression. In the present example, the translated XQuery snippet should be:

```xml
for $var1FRO in nsO:CUSTOMERS ( )
```

SvarlFRO 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 nsO
defined at the beginning of the query. When table metadata is fetched from the Data
Services Platform 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 nsO = "ld-.TestDataServices/CUSTOMERS"
at "Id: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 SvarlFRO, the return expressions are generated based on
this variable. For example:
return
<RECORD>
<ID>{fn:data($varlFR0/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( ). In one embodiment, the availability of the table metadata facilitates accomplishing this step. The simple SQL query is thus ultimately translated into the following XQuery:

```
import schema namespace nsO = "ldiTestDataServices/CUSTOMERS" at "Id:TestDataServices/schemas/CUSTOMERS.xsd";

<RECORDSET>
  { for $varlFR0 in nsO:CUSTOMERS ; return
    <RECORD>
      <CUSTOMERS.CUSTOMERID> (fn:data($varlFR0/CUSTOMERID) )
      <CUSTOMERS.CUSTOMERNAME>
      (fn:data($varlFR0/CUSTOMERNAME) )
    </CUSTOMERS
    </RECORD>
  }
</RECORDSET>
```

Example 5: Final result of translation

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

Example G: SQL with subquery

In another example, the subquery in Example 6 illustrates how SQL rules are applied during XQuery generation in an embodiment. This 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, an XQuery FLWOR involving the CUSTOMERS( ) function is bound to a let variable ($tempvarlFR2). For further querying on this view, the XQuery FLWOR involves the let variable $tempvarlFR2.
Example 7: Subquery translation

The outer SQL query projection in Example 6 comprises 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 the present example, INFO.ID and INFO.NAME are both qualified with the range variable name of subquery, INFO.

As previously described, 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 the present example, the RSN is that of a subquery named DSfFO. 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 the lets have bound variables generated, an XPath for an element inside a let can be written. In the
present case, it is $\text{varFR2}/\text{ID}$, where $\text{varFR2}$ is the generated variable name for the subquery in the let expression. Example 7 shows the final translation to XQuery of SQL in Example 6.

In another illustrative example, a left outer join of two tables is translated.

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

Example 8: Left outer join

The SQL statement in Example 8 returns all customers from the CUSTOMERS view together with any related payments from the PAYMENTS view. Example 9 gives the XQuery translation result for this query.

```
import schema namespace nsO = 
    "Id:TestDataServices/CUSTOMERS" at 
    "Id:TestDataServices/schemas/CUSTOMERS.xsd";
import schema namespace nsl = "Id:TestDataServices/PAYMENTS" at 
    "Id:TestDataServices/schemas/PAYMENTS.xsd";
<RECORDSET>
    {  
        let $tempvarFR4 := <RECORDSET>
            {  
                for $varFR2 in nsO:CUSTOMERS ()
                    let $tempvarFR3 := nsl:PAYMENTS[
                            ($varFR2/CUSTOMERID = CUSTID)]  
                    return if (fn:empty($tempvarFR3)) then 
                        <RECORD>
                            <CUSTOMERS.CUSTOMERID>
                                {fn:data($varFR2/CUSTOMERID)}
                            </CUSTOMERS.CUSTOMERID>
                            <CUSTOMERS.CUSTOMERNAME>
                                {fn:data($varFR2/CUSTOMERNAME)}
                            </CUSTOMERS.CUSTOMERNAME>
                        </RECORD>
                    else (  
                        for $varFR3 in $tempvarFR3
                            return <RECORD>
                                <CUSTOMERS.CUSTOMERID>
                                    {fn:data($varFR3/CUSTOMERID)}
                                </CUSTOMERS.CUSTOMERID>
                                <CUSTOMERS.CUSTOMERNAME>
                                    {fn:data($varFR3/CUSTOMERNAME)}
                                </CUSTOMERS.CUSTOMERNAME>
                                <PAYMENTS.CUSTID>
                                    {fn:data($varFR3/CUSTID)}
                                </PAYMENTS.CUSTID>
                                <PAYMENTS.PAYMENT>
                                    {fn:data($varFR3/PAYMENT)}
                                </PAYMENTS.PAYMENT>
                        )
                    )
            )
    }
```

Example 9: Left outer join translation

In translating this query, the outer join condition CUSTOMERS.CUSTOMERID = PAYMENTS.CUSTID is translated into the XPath filter expression

\[ \text{fn:concat('Mr. ', CUSTOMERNAME}) \]\,

which is then used to construct the join expression. 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.

A more complex example shows how grouping, ordering, scalar functions, and aggregate functions are handled during translation. The SQL query in Example 10 is translated into XQuery as shown in Example 11 (with prolog removed for simplicity).
return
<RECORD>
  <PO_CUSTOMERS . ORDERID>
    {fn:data($var1FR3/ORDERID) }
  </PO_CUSTOMERS . ORDERID>
  <PO_CUSTOMERS . CUSTOMERID>
    {fn:data($var1FR3/CUSTOMERID) }
  </PO_CUSTOMERS . CUSTOMERID>
  <CUSTOMERS . CUSTOMERID>
    {fn:data($var1FR2/CUSTOMERID) }
  </CUSTOMERS . CUSTOMERID>
  <CUSTOMERS . CUSTOMERNAME>
    {fn: data($var1FR2/CUSTOMERNAME) }
  </CUSTOMERS . CUSTOMERNAME>
</RECORD>
</RECORDSET>

for $varNewlet1 in $inter/RECORD
  group $varNewlet1 as $varPartition1 by
    $varNewlet1/CUSTOMERS . CUSTOMERID
      as $varlGB4,
    $varNewlet1/CUSTOMERS . CUSTOMERNAME
      as $varlGB5
  order by (xs:int (fn: count {
      $varPartition1/PO_CUSTOMERS . ORDERID}))
descending
return
<RECORD>
  <concat85Mr .652CUSTOMERNAME7>
    {fn:data{
      (xs:string {
        fn:concat (xs:string ("Mr. "),
                    $varlGB5))})
    }}</concat85Mr .652CUSTOMERNAME7>
  <count80ORDERID7>
    {fn:data{
      (xs:int {
        fn:count (
          $varPartition1/PO_CUSTOMERS . ORDERID)
        )))
    }}</count80ORDERID7>
</RECORD>
</RECORDSET>

Example 11: 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 Sinter. Grouping is then performed using an extension to the XQuery language provided by BEA Systems, hie. of San Jose, California. Sinter is partitioned over CUSTOMERID and CUSTOMERNAME and the new groups are called varlGB4 and varlGB5 respectively. The group results are then ordered before return.
23

It is noteworthy that fnxsoncat (the XQuery equivalent of the SQL CONCAT function) takes the partition varlPartitionl as an argument while fhxount uses varlGB4. This happens because of how the translation logic is distributed among the typed nodes in the AST in the subject embodiment. Based on the information stored in the context, different typed nodes in the AST translate themselves conditionally into XQuery expressions.

Next, the problem of mapping XQuery results into JDBC results is addressed, hi 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.

In one embodiment, JDBC driver performance could be improved by replacing 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:

>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 = "ldcTestDataServices/CUSTOMERS" at "ldiTestDataServices/schemas/CUSTOMERS.xsd";

fn:string-join {
  let $actualQuery := <RECORDSET>
  { for $varlFR0 in ns0:CUSTOMERS ()
    return <RECORD>
      <CUSTOMERS .CUSTOMERID> (fn:data($varlFR0/CUSTOMERID) ) </CUSTOMERS .CUSTOMERID>
      <CUSTOMERS .CUSTOMERNAME>
        (fn:data($varlFR0/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) ))
    ","})
  "),'')

It is noteworthy that 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 enables maintaining a clean separation between JDBC result handling logic and the more complex SQL to XQuery translation logic.

The translator accesses table metadata (which is XML Schema data provided by the Data Services Platform metadata API) in order to correctly perform SQL to XQuery translation. It is also uses table metadata in order to decode the final query results. Since metadata access can be computationally expensive, one translator embodiment employs a "lazy" metadata access and loading strategy. Metadata fetched from the Data Services Platform server is stored in and made available from an in-memory cache, which allows recurring access to metadata without making additional server calls. This metadata is fetched from server only for tables actually involved in the query.

In an embodiment, the 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.

In other aspects, the invention encompasses in some embodiments, computer apparatus, computing systems and machine-readable media configured to carry out the foregoing methods. In addition to an embodiment consisting of specifically designed integrated circuits or other electronics, the present invention may be conveniently
implemented using a conventional general purpose or a specialized digital computer or microprocessor programmed according to the teachings of the present disclosure, as will be apparent to those skilled in the computer art.

Appropriate software coding can readily be prepared by skilled programmers based on the teachings of the present disclosure, as will be apparent to those skilled in the software art. The invention may also be implemented by the preparation of application specific integrated circuits or by interconnecting an appropriate network of conventional component circuits, as will be readily apparent to those skilled in the art.

The present invention includes a computer program product which is a storage medium (media) having instructions stored thereon/in which can be used to program a computer to perform any of the processes of the present invention. The storage medium can include, but is not limited to, any type of rotating media including floppy disks, optical discs, DVD, CD-ROMs, microdrive, and magneto-optical disks, and magnetic or optical cards, nanosystems (including molecular memory ICs), or any type of media or device suitable for storing instructions and/or data.

Stored on any one of the machine readable medium (media), the present invention includes software for controlling both the hardware of the general purpose/specialized computer or microprocessor, and for enabling the computer or microprocessor to interact with a human user or other mechanism utilizing the results of the present invention. Such software may include, but is not limited to, device drivers, operating systems, and user applications.

Included in the programming (software) of the general/specialized computer or microprocessor are software modules for implementing the teachings of the present invention, including, but not limited to providing mechanisms and methods for translating Structured Query Language (SQL) queries into extended Markup Language (XML) query language as discussed herein.

FIGURE 4 illustrates an exemplary processing system 400, which can comprise one or more of the elements of FIGURES 1A and 1B. Turning now to FIGURE 4, an exemplary computing system is illustrated that may comprise one or more of the components of FIGURES 1A and 1B. While other alternatives might be utilized, it will be presumed for clarity sake that components of the systems of FIGURES 1A and 1B are
implemented in hardware, software or some combination by one or more computing
systems consistent therewith, unless otherwise indicated.

Computing system 400 comprises components coupled via one or more
communication channels (e.g., bus 401) including one or more general or special purpose
processors 402, such as a Pentium®, Centrino®, Power PC®, digital signal processor
("DSP"), and so on. System 400 components also include one or more input devices 403
(such as a mouse, keyboard, microphone, pen, and so on), and one or more output devices
404, such as a suitable display, speakers, actuators, and so on, in accordance with a
particular application. (It will be appreciated that input or output devices can also similarly
include more specialized devices or hardware/software device enhancements suitable for
use by the mentally or physically challenged.)

System 400 also includes a machine readable storage media reader 405 coupled to a
machine readable storage medium 406, such as a storage/memory device or hard or
removable storage/memory media; such devices or media are further indicated separately
as storage 408 and memory 409, which may include hard disk variants, floppy/compact
disk variants, digital versatile disk ("DVD") variants, smart cards, read only memory,
random access memory, cache memory, and so on, in accordance with the requirements of
a particular application. One or more suitable communication interfaces 407 may also be
included, such as a modem, DSL, infrared, RF or other suitable transceiver, and so on for
providing inter-device communication directly or via one or more suitable private or public
networks or other components that may include but are not limited to those already
discussed.

Working memory 410 further includes operating system ("OS") 411 elements and
other programs 412, such as one or more of application programs, mobile code, data, and
so on for implementing system 400 components that might be stored or loaded therein
during use. The particular OS or OSs may vary in accordance with a particular device,
features or other aspects in accordance with a particular application (e.g. Windows®,
WindowsCE™, Mac™, Linux, Unix or Palm™ OS variants, a cell phone OS, a proprietary
OS, Symbian™, and so on). Various programming languages or other tools can also be
utilized, such as those compatible with C variants (e.g., C++, C#), the Java™ 2 Platform,
Enterprise Edition ("J2EE") or other programming languages in accordance with the
requirements of a particular application. Other programs 412 may further, for example, include one or more of activity systems, education managers, education integrators, or interface, security, other synchronization, other browser or groupware code, and so on, including but not limited to those discussed elsewhere herein.

When implemented in software (e.g. as an application program, object, agent, downloadable, servlet, and so on in whole or part), a learning integration system or other component may be communicated transitionally or more persistently from local or remote storage to memory (SRAM, cache memory, etc.) for execution, or another suitable mechanism can be utilized, and components may be implemented in compiled or interpretive form. Input, intermediate or resulting data or functional elements may further reside more transitionally or more persistently in a storage media, cache or other volatile or non-volatile memory, (e.g., storage device 408 or memory 409) in accordance with a particular application.

Other features, aspects and objects of the invention can be obtained from a review of the figures and the claims. It is to be understood that other embodiments of the invention can be developed and fall within the spirit and scope of the invention and claims. The foregoing description of preferred embodiments of the present invention has been provided for the purposes of illustration and description. It is not intended to be exhaustive or to limit the invention to the precise forms disclosed. Many modifications and variations will be apparent to the practitioner skilled in the art. The embodiments were chosen and described in order to best explain the principles of the invention and its practical application, thereby enabling others skilled in the art to understand the invention for various embodiments and with various modifications that are suited to the particular use contemplated. It is intended that the scope of the invention be defined by the following claims and their equivalence.
What is claimed is:

1. A method for translating Structured Query Language (SQL) queries into extended Markup Language (XML) query language (XQuery), the method comprising:
   
   determining from data services platform artifacts, a correspondence with SQL database artifacts; and
   
   translating at least one SQL query into at least one equivalent expression in XQuery based upon the determined correspondence.

2. The method of claim 1, wherein determining from data services platform artifacts, a correspondence with SQL database artifacts, comprises:
   
   determining from at least one concept that can be used to describe at least one of a function and a property of a computational object associated with a data services architecture compatible system, a correspondence with at least one concept that can be used to describe at least one of a function and a property of a computational object associated with an SQL database compatible application.

3. The method of claim 1, wherein determining from data services platform artifacts, a correspondence with SQL database artifacts, comprises:

   mapping queryable SQL artifacts to corresponding data services platform artifacts.

4. The method of claim 1, wherein translating at least one SQL query into at least one equivalent expression in XQuery based upon the determined correspondence comprises:
capturing information related to SQL semantics;

moving at least a portion of the information related to SQL semantics from SQL-relevant locations within the query to XQuery-relevant locations; and

generating an XQuery expression.

5. The method of claim 4, wherein capturing information related to SQL semantics comprises:

performing lexical analysis on the SQL query to produce tokens;

parsing tokens generated by lexical analysis;

creating an abstract syntax tree; and

performing syntactic validations.

6. The method of claim 5, wherein performing lexical analysis on the SQL query to produce tokens comprises:

taking the SQL query as an input string of characters and producing a sequence of symbols called "tokens".

7. The method of claim 5, wherein performing syntactic validations comprises:

checking whether the SQL query meets rules of syntax for SQL.

8. The method of claim 4, wherein moving at least a portion of the information related to SQL semantics from SQL-relevant locations within the query to XQuery-relevant locations comprises:
moving at least one node in the abstract syntax tree from an SQL-relevant position to an XQuery-relevant position.

9. The method of claim 4, wherein generating an XQuery expression comprises:
   traversing the abstract syntax tree; and
   serializing into an XQuery format expression.

10. The method of claim 1, wherein data services platform artifacts include at least one of:
    applications, projects, data services, and data service functions.

11. The method of claim 1, wherein SQL database artifacts include at least one of:
    catalogs, schemas, tables, and columns.

12. The method of claim 1, further comprising:
    generating a "patterned" XQuery query that readily facilitates later optimization by a query rewrite optimizer.

13. A machine-readable medium carrying one or more sequences of instructions for translating Structured Query Language (SQL) queries into extended Markup Language (XML) query language (XQuery), which instructions, when executed by one or more processors, cause the one or more processors to carry out the steps of:
    determining from data services platform artifacts, a correspondence with SQL database artifacts; and
    translating at least one SQL query into at least one equivalent expression in XQuery based upon the determined correspondence.
14. The machine-readable medium as recited in claim 13, wherein the instructions for carrying out the step of determining from data services platform artifacts, a correspondence with SQL database artifacts, include instructions for carrying out the step of:

determining from at least one concept that can be used to describe at least one of a function and a property of a computational object associated with a data services architecture compatible system, a correspondence with at least one concept that can be used to describe at least one of a function and a property of a computational object associated with an SQL database compatible application.

15. The machine-readable medium as recited in claim 13, wherein the instructions for carrying out the step of determining from data services platform artifacts, a correspondence with SQL database artifacts, include instructions for carrying out the step of:

mapping queryable SQL artifacts to corresponding data services platform artifacts.

16. The machine-readable medium as recited in claim 13, wherein the instructions for carrying out the step of translating at least one SQL query into at least one equivalent expression in XQuery based upon the determined correspondence include instructions for carrying out the steps of:

capturing information related to SQL semantics;

moving at least a portion of the information related to SQL semantics from SQL-relevant locations within the query to XQuery-relevant locations; and
generating an XQuery expression.

17. The machine-readable medium as recited in claim 16, wherein the instructions for carrying out the step of capturing information related to SQL semantics include instructions for carrying out the steps of:

- performing lexical analysis on the SQL statement to produce tokens;
- parsing tokens generated by lexical analysis;
- creating an abstract syntax tree; and
- performing syntactic validations.

18. The machine-readable medium as recited in claim 17, wherein the instructions for carrying out the step of performing lexical analysis on the SQL query to produce tokens include instructions for carrying out the step of:

- taking the SQL query as an input string of characters and producing a sequence of symbols called "tokens".

19. The machine-readable medium as recited in claim 17, wherein the instructions for carrying out the step of performing syntactic validations include instructions for carrying out the step of:

- checking whether the SQL query meets rules of syntax for SQL.

20. The machine-readable medium as recited in claim 16, wherein the instructions for carrying out the step of moving at least a portion of the information related to SQL
semantics from SQL-relevant locations within the query to XQuery-relevant locations include instructions for carrying out the step of:

moving at least one node in the abstract syntax tree from an SQL-relevant position to an XQuery-relevant position.

21. The machine-readable medium as recited in claim 16, wherein the instructions for carrying out the step of generating an XQuery expression include instructions for carrying out the steps of:

traversing the abstract syntax tree; and

serializing into an XQuery format expression.

22. The machine-readable medium as recited in claim 13, wherein data services platform artifacts include at least one of:

applications, projects, data services, and data service functions.

23. The machine-readable medium as recited in claim 13, wherein SQL database artifacts include at least one of:

catalogs, schemas, tables, and columns.

24. The machine-readable medium as recited in claim 13, further comprising instructions for carrying out the step of:

generating a "patterned" XQuery query that readily facilitates later optimization by a query rewrite optimizer.

25. An apparatus for translating Structured Query Language (SQL) queries into extended Markup Language (XML) query language, the apparatus comprising:
a processor; and

one or more stored sequences of instructions which, when executed by the
processor, cause the processor to carry out the steps of:
determining from data services platform artifacts, a correspondence with

SQL database artifacts; and

translating at least one SQL query into at least one equivalent expression in
XQuery based upon the determined correspondence.
START

Determine from Data Services Platform artifacts, a correspondence with SQL database artifacts

Translate at least one SQL query into at least one equivalent expression in XQuery based upon the determined correspondence

RETURN

FIG. 2A
FIG. 2B

4/11

START

Map queryable SQL artifacts to corresponding data services platform artifacts

RETURN

FIG. 2C

START

Capture information related to SQL semantics

Move at least a portion of the information related to SQL semantics from SQL-relevant locations within the query to XQuery-relevant locations

Generate an XQuery expression

RETURN
CUSTOMERS.ds
declare function
f1: CUSTOMERS ()
as schema - element
(t1:CUSTOMERS)*
external;

<ns0:ArrayOfCUSTOMERS xmlns:ns0=
"Id:TestDataServices/CUSTOMERS">
<ns0:CUSTOMERS>
  <CUSTOMERID>55</CUSTOMERID>
  <CUSTOMERNAME>Joe</CUSTOMERNAME>
</ns0:CUSTOMERS>
<ns0:CUSTOMERS>
  <CUSTOMERID>987655</CUSTOMERID>
  ...
</ns0:ArrayOfCUSTOMERS>

FIG. 3A
FIG. 3C
FIG. 3D
FIG. 3E
for $var1FR0 in ns0:CUSTOMERS()

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

FIG. 3F
SUBSTITUTE SHEET (RULE 26)
INTERNATIONAL SEARCH REPORT

International application No.
PCT/US06/03138

A. CLASSIFICATION OF SUBJECT MATTER
IPC: G06F 7/00( 2006.01),17/30( 2006.01)

USPC: 707/3,4,5,100,102,103R,104.1

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
U.S.: 707/3,4,5,100,102,103R,104.1

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

C. DOCUMENTS CONSIDERED TO BE RELEVANT

<table>
<thead>
<tr>
<th>Category</th>
<th>Citation of document, with indication, where appropriate, of the relevant passages</th>
<th>Relevant to claim No.</th>
</tr>
</thead>
</table>

Further documents are listed in the continuation of Box C.

See patent family annex.

Date of the actual completion of the international search
23 May 2006 (23.05.2006)

Date of mailing of the international search report
14 Jun 2006

Name and mailing address of the ISA/US
Mail Stop PCT, Attn: ISA/US
Commissioner for Patents
P.O. Box 1450
Alexandria, Virginia 223 13-1450

Facsimile No. (571) 273-3201

Authorized officer
Hosain Alam

Telephoje No. (703) 305-9600

Form PCT/BA/210 (second sheet) (April 2005)