(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
26 January 2006 (26.01.2006)

PCT

(10) International Publication Number
WO 2006/009768 A1

(51) International Patent Classification[7]: G06F 17/30

(21) International Application Number:
PCT/US2005/021259

(22) International Filing Date: 9 June 2005 (09.06.2005)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/582,706 23 June 2004 (23.06.2004) US
10/948,523 22 September 2004 (22.09.2004) US

(71) Applicant *(for all designated States except US)*: **ORACLE INTERNATIONAL CORPORATION** [US/US]; 500 Oracle Parkway, Redwood Shores, CA 94065 (US).

(72) Inventors; and

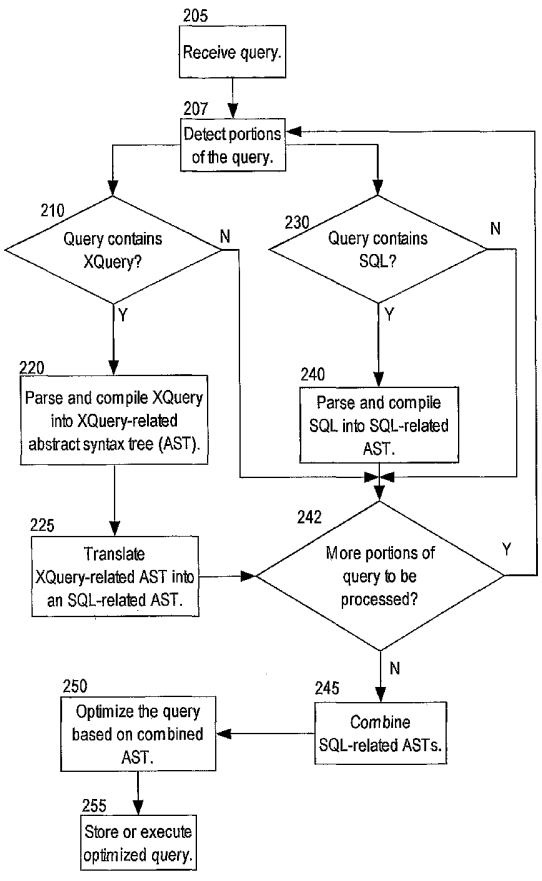
(75) Inventors/Applicants *(for US only)*: **LIU, Zhen, Hua** [US/US]; 1017 Wayne Way, San Mateo, CA 94403 (US). **KRISHNAPRASAD, Muralidhar** [IN/US]; 34136 Summerwind Terrace, Fremont, CA 94555 (US). **MANIKUTTY, Anand** [IN/US]; 764 Marlin Avenue, Apt. 4, Foster City, CA 94404 (US). **WARNER, James** [US/US]; 280 Easy Street, #309, Mountain View, CA 94043 (US). **ZHANG, Hui, X.** [US/US]; 34290 Kenwood Drive, Fremont, CA 94555 (US). **ARORA, Vikas** [US/US]; 8 Locksley Avenue, #4F, San Francisco, CA 94122 (US). **KOTSOVOLOS, Susan, M.** [US/US]; 1319 Eaton Avenue, San Carlos, CA 94070 (US).

(74) Agents: **DRAGANOFF, Stoycho** et al.; Hickman Palermo Truong & Becker LLP, 2055 Gateway Place, Suite 550, San Jose, CA 95110-1089 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BW, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KM, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NA, NG, NI, NO, NZ,

[Continued on next page]

(54) Title: EFFICIENT EVALUATION OF QUERIES USING TRANSLATION



(57) Abstract: Techniques are provided for processing a query including receiving the query, where the query specifies certain operations; determining that the query includes a first portion in a first query language and a second portion in a second query language; generating a first in-memory representation for the first portion; generating a second in-memory representation for the second portion; generating a third in-memory representation of the query based on the first in-memory representation and the second in-memory representation; and performing the certain operations based on the third in-memory representation.

WO 2006/009768 A1

OM, PG, PH, PL, PT, RO, RU, SC, SD, SE, SG, SK, SL, SM, SY, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, YU, ZA, ZM, ZW.

(84) **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, 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*

— *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments*

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.

EFFICIENT EVALUATION OF QUERIES USING TRANSLATION

FIELD OF THE INVENTION

[0001] The present invention relates to query processing. The invention relates more specifically to efficient evaluation of queries using translation.

BACKGROUND OF THE INVENTION

[0002] The approaches described in this section could be pursued, but are not necessarily approaches that have been previously conceived or pursued. Therefore, unless otherwise indicated herein, the approaches described in this section are not prior art to the claims in this application and are not admitted to be prior art by inclusion in this section.

[0003] Relational database management systems (RDBMSs) store information in tables, where each piece of data is stored at a particular row and column. Information in a given row generally is associated with a particular object, and information in a given column generally relates to a particular category of information. For example, each row of a table may correspond to a particular employee, and the various columns of the table may correspond to employee names, employee social security numbers, and employee salaries.

[0004] A user retrieves information from and makes updates to a database by interacting with a database application. The user's actions are converted into a query by the database application. The database application submits the query to a database server. The database server responds to the query by accessing the tables specified in the query to determine which information stored in the tables satisfies the query. The information that satisfies the query is retrieved by the database server and transmitted to the database application. Alternatively, a user may request information directly from the database server by

constructing and submitting a query directly to the database server using a command line or graphical interface.

[0005] Queries submitted to the database server must conform to the syntactical rules of a particular query language. One popular query language, known as the Structured Query Language (SQL), provides users a variety of ways to specify information to be retrieved. Another query language based on the Extensible Markup Language (XML) is XML Query Language (XQuery). XML Query language may have multiple syntactic representations. For instance, one of them is a human-readable version and another is an XML representation (XQueryX). XQuery is described in "XQuery 1.0: An XML Query Language." W3C Working Draft July 23, 2004 at www.w3.org/TR/xquery. XQueryX is described in "XML Syntax for XQuery 1.0 (XQueryX)." W3C Working Draft 19 December 2003 at www.w3.org/TR/xqueryx. Another related technology, XPath, is described in "XML Path Language (XPath) 2.0." W3C Working Draft 12 November 2003 at www.w3.org/TR/xpath20. XQuery and XQueryX may use XPath for path traversal.

[0006] To implement XQuery support in RDBMSs, one approach, referred as coprocessor approach, is to embed a general purpose XQuery processor inside an RDBMS engine and have the XQuery processor execute XQuery on behalf of the RDBMS SQL processor. The coprocessor approach has the SQL processor treat the XQuery coprocessor as a black box. During the execution of the SQL statement, the SQL processor handles the XQuery portion of the query by passing the text of the XQuery portion of the query, and the necessary XML values, as input to the XQuery processor. The XQuery processor then returns the results of processing the XQuery portion of the query to the SQL processor and the SQL processor performs any other appropriate operations specified by the query.

[0007] The coprocessor approach has numerous problems. First, the XQuery processor is not aware of any of the underlying techniques for storing XML data. Therefore, the XQuery processor needs fully materialized XML as input. Consequently, the XML input needed by the XQuery processor must be constructed or materialized by the RDBMS. Often the XML input needed for the XQuery is stored in the database and may be "shredded" into one or component XML elements, and those XML elements may be stored in one or more relational or object relational tables. Under these conditions, the process of materializing the XML data is time and resource consuming, and therefore makes the coprocessor approach inefficient.

[0008] A second problem with the coprocessor approach is that the XQuery portion of an incoming query cannot be optimized with the SQL portion of the incoming query (and vice-versa). Specifically, the XQuery processor is not able to optimize the SQL portion of the query; and the SQL processor is not able to optimize the XQuery portion of the query. Therefore, the SQL and XQuery parts of the query are separately optimized (if at all), which is suboptimal. In addition, the underlying storage of the data needed in the XQuery portion of the query will be stored in a form other than XML (such as being shredded into multiple XMLType columns). Since the XQuery processor is not aware of the form in which the underlying data is stored, the XQuery processor is not able to optimize execution of the XQuery operations based on storage information.

[0009] A third problem with the coprocessor approach occurs when an XQuery processor is invoked multiple times, where the output of a first XQuery becomes the input to a second XQuery in the original query. For example, in the case where the output of a first XQuery must be passed as input to a second XQuery, the output of the first XQuery must be generated as XML. This dictates that the XQuery processor, after determining the result of

the first XQuery, must materialize the result as XML in an XML document and send the XML document to the SQL processor. The SQL processor then passes the XML document back to the XQuery processor along with the second XQuery. The XQuery processor will then retrieve and process the second XQuery with the XML document. This constitutes numerous wasted communication and computational steps and wasted bandwidth.

[0010] Therefore, there is clearly a need for techniques that overcome the shortfalls of the co-processor approach described above.

BRIEF DESCRIPTION OF THE DRAWINGS

[0011] The present invention is illustrated by way of example, and not by way of limitation, in the figures of the accompanying drawings and in which like reference numerals refer to similar elements and in which:

[0012] FIG. 1 is a block diagram that depicts a system for efficient evaluation of queries using translation.

[0013] FIG. 2 is a flow diagram that depicts a process for efficient evaluation of queries using translation.

[0014] FIG. 3 is a block diagram that illustrates a computer system upon which an embodiment of the invention may be implemented.

DETAILED DESCRIPTION

[0015] Techniques for efficient evaluation of queries using translation are described. In the following description, for the purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be apparent, however, that the present invention may be practiced without these specific details.

In other instances, well-known structures and devices are shown in block diagram form in order to avoid unnecessarily obscuring the present invention.

1.0 INTRODUCTION

[0016] The techniques described herein are in no way limited to any particular embodiment or aspect of the embodiment. One example embodiment of the techniques described herein is a database server that accepts queries in SQL, XQuery, and XQueryX. This embodiment is described for illustrative purposes only.

[0017] When the database server receives a query, it determines whether any portion of the query is in a query language other than SQL (e.g. XQuery or XQueryX). For each such portion, the database server parses the portion and compiles the portion into an abstract syntax tree (AST) in an abstract syntax related to the non-SQL query language. Such ASTs are referred to herein as "non-SQL ASTs" or as AST related to particular query languages, such as XQuery ASTs. The non-SQL AST is then converted into an AST in an abstract syntax related to SQL. Such ASTs are referred to herein as "SQL ASTs." This is repeated for each portion of the query that is in a non-SQL query language (e.g. XQuery or XQueryX). Each portion of the query in the SQL is also compiled into an SQL AST. The database server then combines all of the ASTs corresponding to each portion of the query. This combined AST can then be optimized and executed or stored for later execution.

[0018] The techniques described herein apply, at least, to queries that have one or more portions of the query in one or more declarative query languages. Declarative query languages allow one to specify information to be retrieved from a data source without needing to specify how the information is to be retrieved.

2.0 XML DATABASE OPERATIONS

[0019] Some RDBMSs and object-relational database systems (ORDBMS) support "XML" or "XMLType" as a native datatype. Using XMLType, users can store XML documents in databases via the use of XML tables or XMLType columns of tables. Furthermore, users can convert their relational data into XMLType views via the use of SQL/XML publishing functions, such as XMLElement, XMLConcat, etc. XQuery can be used in SQL through a function such as XMLQuery, which enables queries on XMLType values. The XMLTable function enables one to convert XML values (possibly from one or more XMLType columns, or values returned from an XQuery) into a virtual relational table. Consider an example where a table called "purchaseOrder" is an XMLType table with each row storing a purchaseOrder XML document instance. Each XML document instance has contents similar to the following:

```
<PurchaseOrder>
  <ShippingAddress>345, 35 Ave, Berkeley, CA 94613<ShippingAddress>
  <items>
    <lineitem><name>XQuery Book</name><price>46</price></lineitem>
    <lineitem><name>SQL/XML Guide</name><price>78</price><lineitem>
  </items>
</PurchaseOrder>
```

[0020] The following SQL statement, with XQuery embedded in the XMLQuery function, finds the ShippingAddress of all the purchaseOrder XML document instances which have a purchase item whose price is greater than forty-five:

```
select xmlquery('for $i in /PurchaseOrder where $i/items/lineitem/price > 45 return
$i/ShippingAddress' passing value(p) returning content)
from purchaserOrder p;
```

[0021] Here is an example of converting the XML document instance into relational tables via XMLTable construct:

```
select  xt.name, xt.price
```

```
from purchaseOrder p, xmltable('/PurchaseOrder/items/lineitem' passing value(p)
                                columns
                                    name varchar2(20) path 'name',
                                    price number path 'price') xt;
```

3.0 SYSTEM OVERVIEW

[0022] FIG. 1 is a block diagram that depicts a system for efficient evaluation of queries using translation.

[0023] The system illustrated in FIG. 1 includes a database server 150. The database server 150 is a logical machine. The database server 150 includes a non-SQL parser unit 110a, an SQL parser unit 110b, a compiler unit 120, a translator unit 130, and a further query processing unit 140. Each of the units 110a, 110b, 120, 130, and 140 may be a logical machine. Each logical machine may run on separate physical computing machines or may be running on the same physical computing machine as one or more of the other logical machines. Various embodiments of computers and other physical and logical machines are described in detail below in the section entitled Hardware Overview. In one embodiment, each of the units 110-140 are software units running on one or more processors on one or more computers, and those one or more processors on one or more computers make up the database server 150. The database server 150 may include other software units not described herein. The units 110-140 may all be part of the same software program or may be part of separate software programs. That is, a single software program may perform the functionality of two or more of the units 110-140. Alternatively, a first software program may perform some of the functions for a particular unit 110-140 and a second software program may perform other functions for the particular unit 110-140.

[0024] The non-SQL parser unit 110a takes a non-SQL query, or portion of a query, as input and converts it to a second representation (such as SQL). For example, the non-SQL parser unit 110a may be an XQuery parser unit 110a that takes as input an XQuery query and

converts it into an XQueryX representation. The compiler unit 120 takes a query as input and produces an in-memory representation of the query. For example, the compiler unit 120 may take as input an XQueryX query and compile that into an XQuery AST. In one embodiment, the compiler unit may take as input queries in more than one query language, and queries of each query language are compiled into different formats of in-memory representation. For example, an SQL query may be compiled into an SQL AST, whereas an XQueryX query may be compiled into an XQuery AST. Alternatively, queries in one or more different query languages may be compiled into similar or the same format of in-memory representation. In alternative embodiments, there are separate parser units 110a and 110b and compiler unit 120 for each query language. For example, there may be an XQuery parser unit 110a and an SQL parser unit 110b.

[0025] The translator unit 130 converts among the various formats of in-memory representations. For example, the translator unit 130 may convert an XQuery AST into an equivalent SQL AST, or vice-versa.

[0026] The further query processing unit 140 takes an in-memory representation as input and provides query optimization, storage, and / or, execution of the query based on the in-memory representation of the query. The further query processing unit 140 may also perform the step of combining one or more in-memory representations of queries or parts of a query and performing query optimization, storage, and / or execution of the query or queries based on the combined in-memory representations.

[0027] The database server 150 is communicatively coupled to a database 160. The database 160 may be a relational database, an object-oriented database, a file, a repository, or any form of structured data stored on a machine-readable medium. The database server 150 may perform (e.g. using the further query processing unit 140) certain operations required by

the query against the database 160 based on the in-memory representations produced by the compiler unit 120, translator unit 130, or further query processing unit 140. In various embodiments, coupling is accomplished by optical, infrared, or radio signal transmission, direct cabling, wireless networking, local area networks (LANs), wide area networks (WANs), wireless local area networks (WLANs), the Internet, or any appropriate communication mechanism.

4.0 FUNCTIONAL OVERVIEW

[0028] FIG. 2 is a flow diagram that depicts a process for efficient evaluation of queries using translation.

[0029] In step 205, a query is received. The query may be in any appropriate format. For example, the query may be in SQL, XQuery, or XQueryX. The query may also utilize a language for addressing parts of a markup language document, such as XPath. The query may contain one or more "portions". Each of the portions may be in the different formats than each of the other portions. For example, in the context of FIG. 1, the database server 150 may receive a query that contains both SQL and XQuery portions:

```
select xmlquery('for $i in /PurchaseOrder where $i/items/lineitem/price > 45 return
$i/ShippingAddress' passing value(p) returning content)
from purchaserOrder p;
```

where the outer portion of the query is in SQL and the portion of the query inside the xmlquery(...) is in XQuery. The query may also be in a single format.

[0030] In step 207, the query is processed in order to detect whether there are portions of the query in one or more query languages. Once the portions of the query are detected in step 207, then checks are performed to determine whether the query contains XQuery (step 210) or SQL (step 230). In other embodiments, other checks would be performed to determine whether the query contained statements in other particular query languages (e.g.

XQueryX) and steps similar to those for XQuery (steps 210-225) or SQL (steps 230-245) would be performed for queries in each of those other query languages.

[0031] In step 210, a check is performed to determine whether the query contains XQuery. Detecting that a query contains operations to be performed in XQuery may include searching for and finding an XQuery indicator or function call. For example, the non-SQL parser unit 110a may parse the query and detect an XMLQuery function and thereby determine that the query contained within the parentheses is in XQuery format. In various embodiments, step 210 also includes determining whether the query contains XQueryX or XPath and the subsequent steps 220-225 are performed on any XQueryX or XPath queries or subqueries that are found.

[0032] If the query contains no XQuery, then step 242 is performed. Step 242 is described below. Alternatively, if the query does not contain XQuery or SQL statements and, moreover, contains only elements that are not recognizable by the database server 150, then a message may be sent to the query submitter or a system administrator indicating that the format of the query is not recognizable.

[0033] If the query does contain XQuery, then the XQuery portion of the query is parsed and compiled into an XQuery AST in step 220. The XQuery portion of the query may be parsed using any appropriate parser. The parsed XQuery is then compiled into an in-memory representation of the XQuery. The in-memory representation of the XQuery portion of the query is formatted in a way that is compatible with the later steps. The techniques described herein are not limited to any particular in-memory representation. The examples herein will use an abstract syntax tree. ASTs capture the semantic meanings of queries while removing syntactic details.

[0034] The AST for the portion of the query in XQuery will be in a particular abstract syntax related to XQuery. In step 225, the XQuery AST for the XQuery portion of the query is converted into an equivalent SQL AST in a particular abstract syntax related to SQL. Each term in the AST is converted in turn. In one embodiment, the elements at the "leaves" or deepest level of the AST are converted from the XQuery-related abstract syntax to the SQL-related abstract syntax. Then the nodes on the next lowest level are converted. The "higher" levels of the AST are processed one level at a time and from the bottom up. Alternatively, one or more of the leaves of the AST are converted and the parent nodes of these leaves are converted once all of their child nodes in the AST are converted. Details of what XQuery statements are converted to which SQL statements are given in the section entitled XQuery Translation and in '706. Once the XQuery AST has been converted into an equivalent SQL AST, then the equivalent SQL AST may later be combined with any other SQL ASTs in step 245 (described below).

[0035] After step 225 is performed, then, in step 242, a check is performed to determine whether any other portions of the query need to be processed. If there are more portions of the query to process, then step 207 is performed. Alternatively, if there are more portions of the query to process, steps 210 or 230 may be performed. If there are no more portions of the query to process, then step 245 is performed. In general, any portion of the original query that is in a language supported by the database server 150 may be processed. For example, if a query has a first XQuery portion, a second XQuery portion, and an SQL portion, then steps 210-225 are be performed for each of the first XQuery portion and the second XQuery portions and steps 230-240 are performed for the SQL portion of the query. The compiled portions of the query are then combined (described below with respect to steps 245-255).

[0036] In step 230, a check is performed to determine whether the query contains SQL. For example, the SQL parser unit 110b may parse the query (in step 207) and detect an SQL portion of the query and thereby determine that the query contains SQL (in step 230). If the query does not contain SQL, then step 242 is performed. Step 242 is described above. If the query does contain SQL, then in step 240, the SQL portions of the query are parsed and compiled into an SQL AST. Various embodiments of parsing and compiling queries in XQuery are given above in relation to step 220. Techniques for parsing and compiling queries in SQL (or any query language) are similar to those described for XQuery but may use an SQL parser and SQL syntax rules for the parsing. The resulting in-memory representation, such as an SQL AST, contains the semantics of the SQL portion of the query in an abstract syntax related to SQL.

[0037] After step 240 is performed, then, in step 242, a check is performed to determine whether any other portions of the query need to be processed. Once any XQuery portions of the query have been parsed, compiled, and converted to an SQL AST and any SQL portions of the query have been parsed and compiled into an SQL AST, then the ASTs representing the different portions of the query may be combined in step 245. Combining the ASTs may comprise forming a new AST for the query and pointing to or copying the ASTs representing the different portions of the query. Alternatively, one or more of the ASTs representing the different portions of the query may point to or incorporate one or more of the other ASTs representing the different portions of the query. The combined AST is in an SQL-related abstract syntax and represents the entire query. For example, in the context of FIG. 1, the further query processing unit 140 combines the ASTs produced in steps 225 and 240.

[0038] In step 250, the combined AST is used as a basis for optimization of the query. Since the entire query is represented in a single abstract syntax, any appropriate single-

abstract-syntax optimization technique may be used to optimize the query. In step 255 the optimized query is executed or stored for later execution.

[0039] Various embodiments of the techniques described herein enable a query that contains subqueries in multiple query languages to be stored or executed based on an AST in a single abstract syntax. One of the benefits of embodiments of these techniques is that, since the AST that represents the query is in a single abstract syntax, the entire query may be optimized as if it were originally written in a single query language.

[0040] Various embodiments of the techniques described herein enable a query to arrive in a first query language (e.g. XQuery) and for the query to be processed and translated into an equivalent form of a second query language (e.g. SQL). This may be beneficial when the processing or optimization techniques available for the second query language are in some way preferable to those of the first query language. For example, consider a system that does not have XQuery optimizers, but does have SQL query optimizers. Using the techniques described herein, if a query arrives in the XQuery format, the query may be processed and an SQL AST may be generated. The SQL AST may then be optimized using SQL query optimizers. The optimized, equivalent query (as represented by the optimized, SQL AST) may then be executed in place of the original XQuery, thereby saving query processing time.

[0041] In the examples discussed herein, the database server 150 receives the non-SQL query or portions of a query and converts them to SQL. The techniques described herein, however, are not limited to such embodiments. For example, in other embodiments, a middle-tier server that acts as middleware between a database application and a database server 150 may perform the conversions as described herein. The converted SQL query would then be sent to and executed on the database server 150.

[0042] The techniques described herein are presented in terms of a conversion from one abstract syntax to another. In other embodiments of the techniques described herein, the portion of a query in a first syntax (e.g. XQuery) may be converted to a second syntax (e.g. SQL), before it is compiled into an abstract syntax.

5.0 XQUERY TRANSLATION

[0043] As noted above, the techniques described herein provide for converting an AST in one abstract syntax into an AST of another abstract syntax. Below is a description of the conversion between XQuery ASTs and SQL ASTs.

5.1. TRANSLATION OF EXPRESSIONS

[0044] XQuery expressions are rewritten to their equivalent SQL expressions. For instance a literal in XQuery gets mapped to a string or numeric literal (OPNTSTR) in SQL. The following table lists the mapping of general expressions in to their SQL equivalents. Section 5.2 describes the mapping of individual XQuery operators and functions to SQL operators.

5.1.1. EFFECTIVE BOOLEAN VALUE

[0045] The effective Boolean value (EFB) of a sequence is computed implicitly during processing of the following types of expressions:

- Logical expressions (and, or)
- The fn:not function
- The WHERE clause of a FLWOR expression
- Certain types of predicates, such as a[b]
- Conditional expressions (if)
- Quantified expressions (some, every)

[0046] The effective Boolean value returns "false" in the following cases. Otherwise it returns "true".

- An empty sequence
- The Boolean value false
- A zero-length value of type xs:string or xdt:untypedAtomic
- A numeric value that is equal to zero
- The xs:double or xs:float value NaN

[0047] Example rule: To map EFB(*expr*) to SQL, the following rules are applied:

i) Translate expr to its SQL equivalent.

ii) If the static type of expr indicates that the quantifier is 1 (i.e. singleton expr) then

i. If the type is Boolean and the SQL type is also Boolean (i.e. it is mapped to one of the logical operators), then nothing to do

ii. If the type is Boolean and SQL type is number, then add IS NOT NULL (case *<expr>* when 1 then 1 else null)

iii. If the type is numeric then add IS NOT NULL (case <expr> when 0 then 0 when NaN then 0 else 1)

iv. If the type is any other scalar, then add *IS NOT NULL(expr)*

iii) If the static type of expr indicates that the quantifier is * or + then

i. If the type is number or Boolean - convert the collection to a subquery and add the following subquery expression on top - *EXISTS(select * from (select count(*) cnt, sum(value(p))sm from table(xmlsequence(<expr>)) x where (x.cnt = 1 and x.sm = 1) or (x.cnt > 1))*

ii. For all other types map it to *IS NOT NULL* (<expr>) in case the <expr> is a non-subquery operand or to *EXISTS(<expr>)* if expr is an SQL subquery.

5.1.2. ATOMIZATION OF VALUES

[0048] Atomization and conversion to scalar values are required in a number of places. Atomization is determined by the static type analysis. In XQuery this is represented using the fn:data() function.

[0049] The result of fn:data() is the sequence of atomic values produced by applying the following rules to each item in the input sequence:

- If the item is an atomic value, it is returned.
- If the item is a node, its typed value is returned.

[0050] Atomization is used in processing the following types of expressions:

- Arithmetic expressions

- Comparison expressions
- Function calls and returns
- Cast expressions
- Computed element and attribute constructors.

[0051] When rewriting atomization, if the underlying SQL object is an XMLType (or node) an OPTXT2SQLT operator is used to convert the node value to the equivalent SQL type.

[0052] Example rule: Whenever atomization is required and the underlying SQL object's type is not scalar, add the OPTXT2SQLT operator with the desired type. OPTXT2SQLT takes the input XML and the SQL type to convert the result to and atomizes the value to the result.

5.1.3. LITERAL EXPRESSIONS

[0053] Literal Expressions in XQuery are translated to SQL literals. Boolean are mapped as numbers 0 & 1. For example, the expression "1" is mapped to STRTCONS with value "1". Numeric literals are mapped to SQL literals of type NUMBER and string literals are mapped to SQL literals with type VARCHAR2.

[0054] Example rule: Map XQuery literals to SQL literals with the appropriate type information. In case of a string literal, if it is > 4K, then map to a set of concat operations with an empty_clob in the beginning.

Big_String_Literal -> empty_clob() || 4kliteral1 || 4kliteral2 ... || literaln
OPTTCA(OPTTCA(OPTTCA(OPTECLOB, literal1), literal2), ... literaln)

5.1.4. BUILT-IN TYPE CONSTRUCTOR, CAST EXPRESSIONS

[0055] The XQuery CAST and type constructors are mapped to SQL TO_CHAR, TO_NUMBER and XMLCast. XMLCast is used for casting explicitly to user-defined simple types (e.g. hatsize) and for converting simple scalar types to XML values (for passing into functions etc..).

[0056] The following table explains the mapping of XML datatypes to their SQL equivalents. The constructor column is used to check the validity of the value (e.g. byte may be < 127 and greater than -128). The constructor may not be needed if the static type indicates that the expression is of the right type (or a subtype). Constant folding may be performed to eliminate the constructor.

[0057] Example rule: Check datatype to which to cast. If the input is a constant, then check the bounds and raise an error if appropriate. Else if it is a numeric datatype add the TO_NUMBER and the bounds check. If it is a date type, convert it to the TIMESTAMP_TZ with the appropriate format.

XML Datatype	SQL Data Type	Example SQL conversion
xs:integer	NUMBER	TO_NUMBER(<expr>)
xs:positiveInteger	NUMBER	OPTXMLCNV(TO_NUMBER(<expr>),QMTXT_POSITIVEINTEGER)
xs:negativeInteger	NUMBER	OPTXMLCNV(TO_NUMBER(<expr>),QMTXT_NEGATIVEINTEGER)
xs:nonPositiveInteger	NUMBER	OPTXMLCNV(TO_NUMBER(<expr>),QMTXT_NONPOSITIVEINTEGER)
xs:nonNegativeInteger	NUMBER	OPTXMLCNV(TO_NUMBER(<expr>),QMTXT_NONNEGATIVEINTEGER)
xs:int	NUMBER	OPTXMLCNV(TO_NUMBER(<expr>),QMTXT_INT)
xs:short	NUMBER	OPTXMLCNV(TO_NUMBER(<expr>),QMTXT_SHORT)
xs:double	BINARY_DOUBLE	TO_BINARY_DOUBLE(<expr>)
xs:float	BINARY_FLOAT	TO_BINARY_FLOAR(<expr>)
xs:byte	NUMBER	OPTXMLCNV(TO_NUMBER(<expr>),QMTXT_BYTE)
xs:string	VARCHAR2/ CLOB	TO_CHAR(<expr>)
xs:unsignedByte	NUMBER	OPTXMLCNV(TO_NUMBER(<expr>),QMTXT_UNSIGNEDBYTE)
xs:unsignedShort	NUMBER	OPTXMLCNV(TO_NUMBER(<expr>),QMTXT_UNSIGNEDSH

		ORT)
xs:unsignedInt	NUMBER	OPTXMLCNV(TO_NUMBER(<expr>),QMTXT_UNSIGNEDINT)
xs:long	NUMBER	OPTXMLCNV(TO_NUMBER(<expr>),QMTXT_LONG)
xs:unsignedLong	NUMBER	OPTXMLCNV(TO_NUMBER(<expr>),QMTXT_UNSIGNEDLONG)
xs:decimal	NUMBER	TO_NUMBER(<expr>)
xs:Boolean	NUMBER	Case <expr> when null then 0 when 0 then 0 when NaN then 0 else 1
xs:base64Binary	RAW/BLOB	OPTXMLCNV(<expr>, QMTXT_BASE64BINARY)
xs:hexBinary	RAW/BLOB	OPTXMLCNV(<expr>, QMTXT_HEXBINARY)
xs:dateTime	TIMESTAMP_TZ	OPTXMLCNV(<expr>, QMTXT_DATETIMETZ)
xs:time	TIMESTAMP_TZ	OPTXMLCNV(<expr>, QMTXT_TIMETZ)
xs:date	TIMESTAMP_TZ	OPTXMLCNV(<expr>, QMTXT_DATETZ)
xs:gday	TIMESTAMP_TZ	OPTXMLCNV(<expr>, QMTXT_GDAYTZ)
xs:gMonth	TIMESTAMP_TZ	OPTXMLCNV(<expr>, QMTXT_GMONTHTZ)
xs:GYearMonth	TIMESTAMP_TZ	OPTXMLCNV(<expr>, QMTXT_GYEARMONTHTZ)
xs:GMonthDay	TIMESTAMP_TZ	OPTXMLCNV(<expr>, QMTXT_GMONTHDAYTZ)
xs:gYear	TIMESTAMP_TZ	OPTXMLCNV(<expr>, QMTXT_GYEARTZ)

5.1.5. SEQUENCE CONSTRUCTORS

[0058] XMLConcat() is used for concatenating sequences. However, XML constructors are needed for converting scalar values to XMLType. For example, the sequence constructor (1, 2, 3) is mapped to XMLConcat(XMLCast(1), XMLCast(2), XMLCast(3)).

[0059] Example rule: Iterate over all the input of the sequence constructor. For each expression, convert it into its SQL equivalent. If the result type is a simple scalar, add an

XMLCast operand on top of it. Create an XMLConcat() to concatenate the result into a single XMLType.

5.1.6. RANGE EXPRESSION

[0060] Range expressions may be handled by using an operator OPTXNRNG(). See the range operator in operator listing. This range operator returns an XMLType containing a list of integers.

[0061] Example rule: Map to the OPTXNRNG operator.

5.1.7. SET EXPRESSIONS (UNION, INTERSECT, MINUS, EXCEPT)

[0062] Set operations are transformed to value operation in case of unions on values. If XMLType(Seq) may be mappable to SQL UNION, INTERSECT, MINUS, and / or EXCEPT constructs, and doing so may eliminate duplicates among nodes.

[0063] Example rule: Map the set expressions to the SQL UNION, INTERSECT, MINUS, and EXCEPT constructs. The order/map method is used on the XMLType to perform node level operations.

5.1.8. ARITHMETIC EXPRESSIONS

[0064] Static typing ensures that input may be numerical values or atomization and type casts are added. The translation simply converts it to the SQL arithmetic expression.

[0065] Example rule: Convert the XQuery arithmetic expression to its SQL equivalent. See operators table for detailed mapping of the various operators.

5.1.9. VALUE COMPARISON EXPRESSIONS

[0066] Static typing ensures that input may be scalar values or atomization and type casts are added. The translation simply converts it to the SQL comparison expression.

[0067] Example rule: Convert the XQuery comparison expression to its SQL equivalent. See operators table for detailed mapping of the various operators.

5.1.10. GENERAL COMPARISON EXPRESSIONS

[0068] Static typechecking may convert any general comparison expression to a value comparison if possible. If both sides are non collection values and the types are compatible they are converted to value comparison. For example, the expression, $po/PoNo = 21 may be converted to $po/PoNo eq 21 if the type quantifier of $po/PoNo is not a collection (*, + etc.).

[0069] If the static type information for both the sides are known compatible scalar types (e.g. integer *) they are mapped to EXISTS subqueries. For example, $po//LineItems = 21 may get mapped to EXISTS(select * from TABLE(XMLSEQUENCE(<xpath-conv-for $po//LineItems>)) x where value(x) = 21).

[0070] If the static type is unknown (untypedAtomic *) then the equivalent general comparison operator is used.

[0071] Example rule: Given *expr1 GCOMP expr2,* check the compatibility of the static types of the two expressions.

- If the type of both sides is untypedAtomic, they are both converted to a VARCHAR2 type.
- If one side is untypedAtomic and the other is a numeric value, then the untypedAtomic value is converted to the BINARY_DOUBLE.

[0072] Now check the quantifier for the type (e.g. quantifier (integer *) is *). For example:

- If the quantifier for both sides is a singleton (empty or ?) then map the GCOMP to the SQL value comparison operator.
- If expr1 quantifier type is a collection (* or +) and expr2 quantifier is a singleton then map to

  EXISTS(select null from TABLE(XMLSEQUENCE(*expr1*) x
  Where value(x) *VCOMP expr2*) (VCOMP is the value comparison

equivalent)

e.g. $po//LineItemNo < 20 becomes (assuming the static type of $po//LineItemNo is integer*)

EXISTS(select null from TABLE(XMLSEQUENCE(*$po//LineItemNo*) x

Where value(x) < 20)

- If expr2 quantifier type is a collection (* or +) and expr1 quantifier is a singleton then map to

  EXISTS(select null from TABLE(XMLSEQUENCE(*expr2*) x

  Where expr1 *VCOMP* value(x)) (VCOMP is the value comparison equivalent)

  e.g. 20 < $po//LineItemNo becomes (assuming the static type of $po//LineItemNo is integer*)

  EXISTS(select null from TABLE(XMLSEQUENCE(*$po//LineItemNo*) x

  Where 20 < value(x))

- If both expressions are collections then map the expression to

  EXISTS(select null from TABLE(XMLSEQUENCE(*expr1*) x

  Where EXISTS (select null from TABLE(XMLSEQUENCE(*expr2*) y

  Where value(x) *VCOMP* value(y)))

  e.g. $po1//LineItemNo < $po2//LineItemNo becomes

  EXISTS(select null from TABLE(XMLSEQUENCE(*$po1//LineItemNo*) x

  Where EXISTS (select null from TABLE(XMLSEQUENCE(*$po2//LineItemNo*) y

  Where value(x) < value(y)))

5.1.11. NODE COMPARISON EXPRESSIONS

[0073] Node comparisons are handled by using the order method on XMLType. They are mapped to the SQL value comparison operators.

[0074] Example rule: Map to the SQL value comparison operators as described herein.

5.1.12. ORDER COMPARISON EXPRESSIONS

[0075] Order comparison expressions are used in the FLWOR order by clause. These are mapped to the SQL order by clause.

[0076] Example rule: Map Order comparison expressions to SQL order by clause expressions.

5.1.13. LOGICAL EXPRESSIONS (AND, OR, NOT)

[0077] XML logical expressions are mapped to SQL logical expressions. SQL has 3-valued logic, but empty sequences are mapped to NULL and this works for non-constraint operations. Constraints may be an important issue, since a NULL value from a constraint is treated as matching the constraint.

[0078] Example rule: Map logical expressions to SQL logical expressions (AND, OR). In case when the logical expression appears as a top-level expression (outside of the WHERE clause or IF clause) then add a CASE Expression to the result. E.g. if the query is the expressions "a < 20 and b > 30", map it to CASE WHEN (a < 20 and b > 30) then 1 else 0.

5.1.14. FLWOR EXPRESSION

[0079] FLWOR expressions are mapped to SQL select expressions. The LET clauses are mapped as common sub expressions in the SQL query. The RHS of the *for*-clause is mapped to the *from*-clause, the where-clause is mapped to the SQL where-clause and the return-clause is mapped to the SQL select-clause. If node identities are to be preserved in the query, then the query block is marked as NO_MERGE.

```
for <var>  in <rhs-expr1>,
    <var2> in <rhs-expr2>
where <cond-expression>
order by <o1>, <o2>.. <on>
return <ret-expr>
```

is mapped to

```
select /*+ NO_MERGE */ XMLAGG( <sql-ret-expr> )
from TABLE(XMLSEQUENCE( <sql-rhs-expr1> ) as "var1"
     TABLE(XMLSEQUENCE( <sql-rhs-expr2> ) as "var2"
where <sql-cond>
order by <sql-o1>, <sql-o2>, .. <sql-on>
```

[0080] Example 1: Simple FLWOR clause

```
for $i in (1,2,3)
where $i > 1
return $i+ 2
```

is mapped to

```
select xmlagg(XMLCast(XMLCast(value("$i") as number) + 1 as xml))
from table(xmlsequence( xmlconcat ( cast (1 as xmltype(sequence)),
                                    cast (2 as xmltype(sequence)),
                                    cast (3 as
xmltype(sequence))))
                                returning sequence) as "$i"
where XMLCast(value("$i") as number) > 1;
```

[0081] Example 2. FLWOR clause with XPath expressions:

```
for $i in doc("foo.xml")/PurchaseOrder
where $i/PoNo = 21
return <A>{$i}</A>
```

becomes

```
select xmlagg(XMLElement("A", value("$i")))
from table(xmlsequence( extract ( select
extract(Res,'/Contents/*') from resource_view
where equals_path(res,'/foo.xml') = 1),
                                    '/PurchaseOrder'))) "$i"
where XMLCast( OPTXATG(value("$i", '/PoNo') as number) = 21
```

5.1.14.1. LET CLAUSE HANDLING

[0082] A LET clause expression is inlined into the query expression (and marked as common subexpression) if node identities need not be preserved. Otherwise a subquery is created with the LET clause expressions as it's select list. The subquery is marked as non-mergeable to prevent view merging.

[0083] Example with node identities preserved:

```
for $i in doc("foo.xml")/PurchaseOrder//LineItems
 let $j := doc("baditems.xml")//BadItems
  where $i/ItemNo eq $j/ItemNo
  return ($i, $j/BadItem)
```

becomes

```
select xmlagg(xmlconcat("$i", OPTXATG("$j",'/BadItem')))
```

```
from
 (select /*+ NO_MERGE */ value("$I") as "$I",
          (select XMLAgg(OPTXATG(value(x))
           from  table(xmlsequence(
              extract ( select extract(Res,'/Contents/*')
                        from resource_view
                       where equals_path(res,'/baditems.xml') = 1),
                                '//BadItems'))) "x"
        ) as "$j"
   from table(xmlsequence( OPTXATG(
OPTXATG ( select extract(Res,'/Contents/*')
                        from resource_view
                        where equals_path(res,'/foo.xml') = 1),
                                '/PurchaseOrder'),
'//LineItems))) "$i"
    )
where exists( select null from table(xmlsequence(
OPTXATG("$j",'/ItemNo'))) x
                where XMLCast(OPTXATG("$I",'/ItemNo')as number) =
XMLCast(x as number));
```

[0084] Example without preservation of node identities: If node identity preservation is not critical, then the LET clause may be inlined into the expression itself directly. This optimization may be done either by requiring the user to have a pragma specifying that node identities are not essential. This may be also be done implicitly be examining the globally to determine whether any node related operations are used in the query.

```
for $i in doc("foo.xml")/PurchaseOrder//LineItems
 let $j := doc("baditems.xml")//BadItems
  where $i/ItemNo eq $j/ItemNo
  return $i
```

becomes

```
select xmlagg(value("$i")
from table(xmlsequence(OPTXATG (
                        OPTXATG (
                          select extract(Res,'/Contents/*')
                          from resource_view
                          where equals_path(res,'/foo.xml') = 1),
                         '/PurchaseOrder'),
                        '//LineItems))) "$i"
where exists( select null
              from table(xmlsequence(
                          OPTXATG( (select XMLAgg(OPTXATG(value(x))
                                    from  table(xmlsequence(
              extract (select extract(Res,'/Contents/*')
                       from resource_view
                       where equals_path(res,'/baditems.xml') = 1),
                             '//BadItems'))) "$j"
```

```
                where XMLCast(OPTXATG("$i",'/ItemNo') as number) =
                      XMLCast(OPTXATG("$j",'/ItemNo') as number));
```

[0085] Example technique: Since preventing view merging may adversely affect query performance, the WHERE clause for the FLWOR expression is first searched to see if it includes any of the LET variable. If not, then the LET clause may be evaluated as a result of the FLWOR clause (along with the return).

[0086] For example in the following query,

```
for $i in doc("foo.xml")/PurchaseOrder//LineItems
 let $j := count(doc("baditems.xml")//BadItems[ItemNo =
$i/ItemNo])
  where $i/ItemNo > 200
  return $j
```

$j is often used in the return clause and not in the WHERE clause – so that the WHERE clause may be evaluated before the LET clause. This query is equivalent to

```
for $j in
  for $i in doc("foo.xml")/PurchaseOrder//LineItems
  where $i/ItemNo > 200
  return
    count(doc("baditems.xml")//BadItems[ItemNo = $i/ItemNo]
      return $j
```

[0087] Example rules: Normalize Type declarations: If the FOR or LET clause involves any type declaration, check the static type of the expression corresponding to the clause. If it is the same or a subtype of the declared type then ignore the type declaration. If it is a supertype of the declared type, then add a TREAT expression on the expression and map it to SQL. Otherwise raise an error. *For <var> <type> := <expr>* is normalized to *for <var> := TREAT<expr> as <type>* and then mapped to SQL.

[0088] Convert all expressions in the FOR, WHERE, LET and RETURN clauses to their SQL equivalent. Map the FOR clause expressions to SQL FROM clauses (joins). If node identity need not be preserved, then inline the LET clause expression wherever it is referenced. For example:

```
For <var1> in <expr1>,  <var2> in <expr2>
```

```
let <var3> in <expr3>
  where <cond-referencing-var3>
return <expr4>
```

is mapped to

```
select xmlagg(<expr4>) /* inline var3 references with expr3 */
from table(xmlsequence( <expr1> ) as "var1" ,
     table(xmlsequence( <expr2>) as "var2",...
where <cond-referencing-var3>   /* inline var3 references with
expr3 */
```

[0089] Otherwise, if node identity is to be preserved, examine the LET clauses in the FLWOR expression to determine if they may be evaluated before the WHERE clause, by checking whether the variables defined in the LET clauses are used in the WHERE clause. Add a NO_MERGE hint on the inner query block to indicate that view merging should not happen.

[0090] If the LET clause needs to be evaluated before the WHERE clause, map the LET clause expression as a select list subquery and map the WHERE clause to the SQL WHERE clause of the outer query block. For example:

```
For <var1> in <expr1>,  <var2> in <expr2>
let <var3> in <expr3>
  where <cond-referencing-var3>
return <expr4>
```

is mapped to

```
select xmlagg( <expr4>  )
from   (select /*+ NO_MERGE */
              value("var1") as "var1",
              value("var2") as "var2", ..
              <expr3> as "var3"
         from table(xmlsequence( <expr1> ) as "var1" ,
              table(xmlsequence( <expr2>) as "var2",...
        )
where <cond-referencing-var3>
)
```

[0091] If the LET clause need NOT be evaluated before the WHERE clause, map the LET clause expression as a select list subquery, but map the WHERE clause to the SQL WHERE clause of the inner query block. For example:

```
For <var1> in <expr1>,  <var2> in <expr2>
```

```
let <var3> in <expr3>
  where <cond-not-referencing-var3>
return <expr4-refecencing-var3>
```

is mapped to

```
select xmlagg(<expr4-referencing-var3> )
from
  (select /*+ NO_MERGE */
          value("var1") as "var1",
          value("var2") as "var2", ..
          <expr3> as "var3"
    from table(xmlsequence( <expr1> ) as "var1" ,
                  table(xmlsequence( <expr2>) as "var2",...
    where <cond-referencing-var3>
       )
```

5.1.15. PATH EXPRESSIONS

[0092] Path expressions are mapped to SQL expressions. An operator OPTXATG is used to extract out individual nodes in the path expression. It represents a single step traversal. Static typechecking is used to optimize some of the path expression conversion.

5.1.15.1. PATH STEPS WITH NAME TEST

[0093] This represents the standard XPath 1.0 path expressions. Simple path traversals with name tests are rewritten to the OPTXATG operator. Static type checking is used to figure out the type and cardinality of the various steps. This is later used for translation. Predicates are mapped to relational WHERE clauses after normalization. General comparisons involving collection elements are mapped to subqueries involving value comparisons. If there is no static type checking information available, then each step is assumed to produce an untypedAny.

[0094] OPTXATGs are further optimized (or collapsed) based on the input arguments. For example:

```
$i/PurchaseOrder/PoNo
```

is mapped to

```
OPTXATG(OPTXATG($i, 'PurchaseOrder'), 'PoNo').
```

[0095] OPTXATGs are further optimized (or collapsed) based on the input arguments. For example the expression,

(<A><B>33</B></A>)/A/B

is mapped to

OPTXATG(OPTXATG(XMLElement("A", XMLElement("B",33)), 'A'), 'B')

[0096] The XATG that extracts A and the XMLElement() creating A are collapsed and the result is XMLElement("B", 333) which corresponds to the result <B>33</B>.

[0097] In a second example, path predicates are mapped to relational predicates:

```
$i/PurchaseOrder/PoNo eq 21
```

gets mapped to

```
XMLCast( OPTXATG( OPTXATG ( $i , 'PurchaseOrder'), 'PoNo') as
number) = 21
```

[0098] The previous mapping is only valid if during static type checking the type of PoNo is an atomic value that may be cast to a number. If there is no schema information available, then the static type information may only yield the fact that PoNo is of xs:anyType. The XMLCast in this case may perform atomization of the values and raise error if the input (PoNo) is not a single atomic value or element castable to a number.

[0099] If the general comparison operator (=) was used and the type information is not known, then it has to be treated as a collection comparison. In this case, the path predicate is rewritten to a TABLE subquery using the value comparison. For example:

```
$i/PurchaseOrder/PoNo = 21
```

gets mapped to

```
EXISTS( select null
       from table(xmlsequence(OPTXATG( OPTXATG ( $i ,
'PurchaseOrder'), 'PoNo')))) x
      where XMLCast( value(x) as number) = 21 )
```

[0100] A path expression that involves predicates in the path step itself is also handled in a similar fashion. For example:

```
$i/PurchaseOrder[PoNo eq 21]
```

gets mapped to

```
select OPTXATG( $i, 'PurchaseOrder')
from dual
where XMLCast( OPTXATG( OPTXATG ( $i , 'PurchaseOrder'), 'PoNo')
as number) = 21
```

and in the case of general comparison with no schema inputs,

```
$i/PurchaseOrder[PoNo = 21]
```

gets mapped to

```
select XMLAGG(value(v))
from table(xmlsequence(OPTXATG($I, 'PurchaseOrder')) v
where exists(
        select null from
        table(xmlsequence(OPTXATG( value($v), 'PoNo'))) x
        where XMLCast(value(x) as number) = 21);
```

5.1.15.2. PATH STEPS WITH KIND TEST

[0101] Kind test involve checking the type of the node (e.g. text(), processing-instruction() etc.). XQuery adds more sets of type check such as the name and schema type of the node. For example, $i/element(foo, bar) indicates that the child element named foo of type bar needs to be extracted. The OPTXATG operator is enhanced to take in a node type in addition to the node name for extraction.

5.1.15.3. PATH STEPS WITH FILTER EXPRESSIONS

[0102] Filter expressions are handled by normalizing the path expression and pushing the path expression into the context node. For example, $i/PurchaseOrder/(for $j in LineItems return count($j/Orders) may be normalized into (for $j in $i/PurchaseOrder/LineItems return count($j/Orders)).

[0103] Example rule: For each step of the path expression map it to an SQL operator as follows:

a) If the step is a name test, then map it to the OPTXATG operator. *<expr> <step> <QName-or-wildcard> maps to OPTXATG(<expr>, <step>, <localname>, <namespace>)*

b) If the step is a kind test, then map it to the OPTXATG operator with type information *<expr> <step> <type> is mapped to OPTXATG(<expr> , <step>, <type>)*

c) If the step is a filter step, then normalize the expression as follows - *<expr> <step> <filterexpr> is normalized to (for $m in <expr> return <filterexpr> with the context node in the filter expr changed to $m.* This is then rewritten to SQL.

[0104] For example, $i/PurchaseOrder/(for $j in LineItems return count($j/Orders)) is normalized into for $m in $i/PurchaseOrder return (for $j in $m/LineItems return count($j/Orders)) and then mapped to SQL.

[0105] For predicates in the path expression, the static type of the expression containing the predicate may be checked as followed:

a) If the static type indicates that the expression results in a collection (quantifier = * or +), then create a subquery with the expression and map the predicate to the WHERE clause.

b) Else if the static type indicates that the expression results in a singleton node, map to a

5.1.16. CONDITIONAL EXPRESSIONS

[0106] If-then-else expressions are mapped to the SQL CASE WHEN Expressions.

[0107] Example rule: Given *if <expr1> then <expr2> else <expr3>*. Add the effective Boolean value operator to expr1 if necessary (as determined by the static type checking), and map the expression to *CASE WHEN <expr1 > then <expr2> else <expr3>*.

5.1.17. QUANTIFIED EXPRESSIONS

[0108] Quantified expressions may be mapped into SQL EXISTS clauses. For example to find all purchaseorders where at least one of the lineitem number is present in the bad items list,

```
for $I in ora:view("po_TAB")//PurchaseOrder
where some $j in $i//LineItem satisfies
          for $k in ora:view("bad_items") where $k//ItemNo =
$j/ItemNo return $k,
```

where "ora:view()" is an XQuery function that returns the data from a relation table in XML form, may be mapped to

```
select value("$I")
from  "po_TAB" "$I"
where exists(
     select( select "$k"
             from (select value(p) "$k" from "bad_items" p)
             where OPTXATG("$k",'//ItemNo') =
OPTXATG("$j",'/ItemNo')
    )
    from (
     select value("$j") as "$j"
     from table(xmlsequence(OPTXATG(value("$I"), '//LineItem')))
"$j"
    )
  )
```

5.1.18. DIRECT ELEMENT CONSTRUCTOR EXPRESSION

[0109] Element constructors are mapped to XMLElement() operator. Attributes inside the element are mapped to the XMLAttributes() clause in the XMLElement() operator.

[0110] Example,

```
<A> { "21"  } </A> is mapped to XMLelement(NAME "A", '21') and
<A b="21">22</A> is mapped to XMLElement(NAME "A",
XMLAttributes(21 as "b"), '22')
```

[0111] Example rule: Map any element constructor to XMLElement() using XMLAttributes() for attribute construction.

5.1.19. COMPUTED ELEMENT CONSTRUCTOR EXPRESSION

[0112] Computed element constructor is also mapped to XMLElement(). Any computed attribute constructor that is a child of the element constructor is optimized and mapped to the XMLAttributes() clause. The XMLElement() operator is relaxed to allow dynamic element names. The operator may also be modified to make free standing attribute children to become the element's attributes.

```
element {"a" } { "21" }
```

is mapped to

```
XMLElement(NAME EXPR 'a', '21')
```

and

```
element {"a" } {
  Attribute b { "21" }
  {22}
}
```

is mapped to

```
XMLElement(NAME EXPR 'a', XMLAttributes('21' as "a"), '22')
```

[0113] Example rule: Map any computed element constructor to XMLElement() and map child attribute constructors to XMLAttribute().

5.1.20. COMPUTED ATTRIBUTE CONSTRUCTOR EXPRESSION

[0114] Attribute constructors are handled by allowing the XMLAttribute() as a top level SQL function.

```
Attribute "a" { "21"  } </A>
```

is mapped to

```
XMLAttribute(21 as "a")
```

[0115] Example rule: Map Attribute constructors to XMLAttribute.

5.1.21. OTHER XML CONSTRUCTION EXPRESSIONS

[0116] Example rule: The XML constructors are mapped to the equivalent SQL/XML standard functions.

XMLComment	OPTXMLCOM
XMLProcessingInstruction	OPTXMLPI
CDataSection	OPTXMLCDATA
ComputedElemConstructor	OPTXMLELEM
ComputedAttributeConstructor	OPTXMLATTR
ComputedDocumentConstructor	OPTXMLROOT
ComputedTextConstructor	OPTXMLTXT

5.1.22. TYPESWITCH EXPRESSION

[0117] Typeswitch expressions are similar to if-then-else except that they switch on the type of the input. The typechecking may be performed using an SQL operator OPTXTYPCHK that checks the XQuery type of the input returning 1 if the type matches. If the static type information of the expression is known the typeswitch may be optimized away completely. The OPTXTYPCHK operator may be optimized away for most of the cases where the static type check information may optimize the type checking.

[0118] Example rule: Map Typeswitch to Case expression and use the OPTXTYPCHK to check the type of the input. Given

```
typeswitch <expr>
   case <var1> as <type1> return <expr1>
   case <var2> as <type2> return <expr2>
    ...
  default <exprn>
```

[0119] Check the static type of <expr>. Let this be *etype*. Now for each Case expression match the *etype* with the *type-i* in the Case expression. If the two types are the same or *etype* is a subtype of *type-i*, then optimize the typeswtich expression away and return the SQL equivalent of *expr-i*. If *type-i* is a subtype of *etype* then map the entire typeswitch expression to the SQL expression of the form

```
Case when OPTXTYPCHK(<expr>, <type1>) = 1 then <expr1>
```

```
    When OPTXTYPCHK(<expr>, <type2>) = 1 then <expr2>
    ..
else <exprn>
```

[0120] If no *type-i* is in the type hierarchy of *etype* then return the SQL equivalent of the default expression *exprn.*

5.1.23. INSTANCEOF EXPRESSION

[0121] InstanceOf expression may be evaluated using the OPTXTYPCHK operator and may be optimized using the static type of the input expression.

[0122] Example rule: Given *<expr1> instanceOf <type1>*. Check if the static type of <expr1> is the same or a subtype of <type1>. If so, then remove the expression. If the static type is a supertype of type1 then map to *OPTXTYPCHK(<expr1>, <type1>)*. Else it is an error.

5.1.24. CASTABLE EXPRESSION

[0123] Castable expressions are used to check if the input is castable to the given form. They may be mapped to SQL using an OPTCASTABLE operator that may be used to determine if the expression is castable to the other type. Note that this expression may be removed if the static type of the input is the same or a subtype of the input.

[0124] Example rule: Map <expr> *castable as <type>* is mapped to *OPTXTYPCHK(<expr>, <type>)*

5.1.25. TREAT EXPRESSION

[0125] Treat expressions are mapped to Case expressions.

[0126] Example rule: Map *treat <expr> as <type>* to *CASE WHEN OPTXTYPCHK(<expr>, <type>) = 1 then <expr> else error() end.*

5.1.26. VALIDATE EXPRESSION

[0127] Validate expressions are mapped to the XMLValidate() function. The XMLValidate() is an SQL operator that takes in a schema type (local or global) and returns the validated XML value back or an error.

[0128] Example rule: Map *validate <type> <expr>* to *XMLValidate(<expr>, <type>)*

[0129] Validate expressions may also be mapped to an XMLIsValid() function.

5.1.27. AGGREGATE EXPRESSION

[0130] XQuery allows aggregates to be present anywhere in the query. This is not directly supported by SQL. For example, the following XQuery returns all purchaseorders that have more than 21 lineitems in them.

```
for $i in doc("Po.xml")
where count($i/PurchaseOrder/LineItems) > 21
return $i
```

[0131] Aggregates are rewritten using a subquery to compute the aggregate.

```
select x.res
from (select res from resource_view where
equals_path(res,'Po.xml') = 1) x
where (
     select count(value(z))
     from table(xmlsequence(OPTIXATG(OPTIXATG(x.res
,'PurchaseOrder'), 'LineItems'))) z
          ) >21;
```

[0132] Example rule: When mapping Functions & Operators (F&O) to SQL expressions, if the F&O is an aggregate then map it to an SQL Subquery. Map *agg-func (<expr>)* to *(select sql-agg-func(value(p)) from table(xmlsequence(<expr>)) p)* .

5.1.28. POLYMORPHIC OPERATOR

[0133] Since XQuery allows overloading of arithmetic and comparison function to handle a variety of datatypes, the mapping to an SQL operator may vary depending on the

run-time input types of the operands. XQuery operators utilizing such overloading are called "polymorphic operators."

[0134] For example, consider, the following XQuery expression:

```
declare $b xs:boolean external;
(if ($b) then 3.3 else xs:date("2001-08-25") ) +
(if ($b) then 44 else xdt:yearMonthDuration("P5Y0M")
```

[0135] Depending on the value at run time for the external variable $b, the addition in XQuery can be translated to decimal addition (in this case, it adds decimal value 3.3 and 44) or can be translated to date addition with yearMonthDuration (in this case, it adds five years and zero months to the date '2001-08-25' which yields the date '2006-08-25').

[0136] Therefore, the determination as to whether this expression is mapped to the SQL decimal operator or SQL date addition operator may only be made at run time. To support this, the techniques described herein map arithmetic expressions, whose input data type is polymorphic as determined from static type check, into polymorphic SQL arithmetic operators. A polymorphic SQL arithmetic operator can dispatch to the appropriate SQL arithmetic operator at run time depending on the run time input types.

[0137] Similar translations are used for polymorphic XQuery comparison functions as well. Polymorphic XQuery comparison functions are mapped to polymorphic SQL value comparison operators.

[0138] As noted above, it may be beneficial to use polymorphic operator translation if the input types may vary during XQuery compile time. Furthermore, non-polymorphic XQuery expressions, such as 3.3 + 44, may still be directly translated it into non-polymorphic SQL expressions, e.g. using SQL decimal addition operators, instead of the polymorphic SQL operators.

5.1.29. XQUERY USER-DEFINED AND EXTERNAL FUNCTIONS

[0139] XQuery supports user-defined functions written in XQuery and external functions whose implementation is outside of the XQuery environment. For example, the body of a function may be written in a programming language such as the Java programming language.

[0140] User-defined XQuery functions may be translated into Oracle PL/SQL (Procedural Language/Structured Query Language) functions. This may be performed by translating the body of a user-defined XQuery function from an XQuery expression into a PL/SQL expression. Additionally, an invocation of an XQuery function may be translated into an invocation of a PL/SQL function in SQL.

[0141] The techniques described herein also support external user-defined functions in XQuery. For example, if the body of a function is written in the Java programming language, then the function may be mapped to an equivalent external user-defined function using an SQL external user-defined function written in the target language (for example, a Java user-defined SQL function). Therefore, an external user-defined function in XQuery, implemented in Java, C, PL/SQL, or any other appropriate language, may be translated into a user-defined PL/SQL function, written in Java, C, PL/SQL, or any other appropriate language supported by the SQL system.

5.1.30. XQUERY MODULE

[0142] XQuery supports modules. XQuery modules are fragments of XQuery code that can be independently created and imported or loaded into an XQuery processor. XQuery modules may be translated into Oracle PL/SQL packages that may be independently created and loaded into the database server.

5.2. MAPPING OF FUNCTIONS & OPERATORS

[0143] The following table illustrates the mapping of XQuery operators and standard functions (F&O) to existing or new SQL operators.

XQuery Operator	SQL mapping	Optimized	Notes
And	OPTAND		Empty sequence returns empty sequence. NULL on NULL is ok for these cases, since the WHERE clause may not be satisfied.
Or	OPTOR		-same
>	OPTXGT	OPTTGT	Optimization in case when General Comparison may be normalized to value comparison. May be translated to polymorphic SQL operator. May be translated to SQL exists subquery with value comparisons as illustrated in section 5.1.10 General Comparison Expression.
<	OPTXLT	OPTTLT	-same-
>=	OPTXGE	OPTTGE	-same-
<=	OPTXLE	OPTTLE	-same-
=	OPTXEQ	OPTTEQ	-same-
!=	OPTXNE	OPTTNE	-same-
	OPTTLT		Also add ERROR_ON_NULL(LHS) in case the left hand side (LHS) is NULLABLE (e.g. optional element/attribute) $i/b < 20 is mapped to i.b < 20 and error_on_null(i.b) if i.b is mapped to a nullable value.
gt	OPTTGT		Empty sequence returns empty sequence. NULL on NULL is ok for these cases, since the WHERE clause may not be satisfied. May be translated to polymorphic SQL operator.
eq	OPTTEQ		-same-
ne	OPTTNE		-same-
le	OPTTLE		-same-
ge	OPTTGE		-same-
node is	OPTTEQ		Node operation
>>	OPTTGT		-same-,
<<	OPTTLT		
range	OPTXNRNG		Range operator
union, |	OPTXUJ	OPTTUN	If adding map or order method on XMLType(Seq), then may reuse the regular UNION/INTERSECT etc.

intersect	OPTXINTR	OPTTIS	-same-
except	OPTXEXC	OPTTMI	-same-
+	OPTTAD		Add TO_NUMBER() on non-char inputs. May be translated to polymorphic SQL operator.
-	OPTTSU		-same-
mult	OPTTMU		-same-
div	OPTTDI		-same- -INF, +INF are handled by binary_float operators. May cast LHS or RHS to binary_float or binary_double if the XMLSchema datatype is float/double.
idiv	OPTTTR, OPTTDI		truncate(div) returns integer division
unary +	-		Ignored
unary -	OPTTNG		
mod	OPTTMO		or the divisor is positive or negative zero (0), or both, the result is NaN – Return 0 if the divisor is 0.
cast functions			See Datatype Mapping
Node Functions			
fn:nodename	OPTXNNAME		XPath operators
fn:string	OPTXSTRING		String conversion
fn:data	OPTXT2SQLT		This is an SQL operator which does atomization.
fn:base-uri	OPTXBURI		
fn:document-uri	OPTXDOCURI		Special Function to access document URI for docs. Either part of the XMLType or translate it to access the ANY_PATH of resource_view
Error Functions			
fn:error()	dbms_xquery.raiseError()		
fn:trace()	dbms_Xquery.trace()		
Math functions			
fn:abs	OPTTAB		
fn:ceiling	OPTTCE		
fn:floor	OPTTFL		
round	OPTTFL(a+0.5)		May add 0.5 and use floor: May normalize in XQuery to be xf:floor(a+0.5)
round-half-	OPTXFLHE		

to-even			
String functions			
fn:codepoints-to-string	-		NLS input needed
fn:string-to-codepoint	-		NLS input needed
fn:compare	-		May be equivalent to having in SQL as case lhs < rhs then -1 else case when lhs = rhs then 0 else 1.
fn:concat	OPTTCA		May map to multiple OPTTCA (SQL takes only 2 args)
fn:string-join	OPTXSJOIN	OPTTCO	May do with concat operators, but empty sequence needs to be taken into account.
fn:substring	OPTFL(x+0.5), OPTTSS		Add ROUND to all input args
fn:string-length	OPTTLN		
fn:normalize-space	OPTXSOPR		String operations (normalize space)
fn:normalize-unicode	OPTXSOPR		NLS support
fn:upper-case	OPTTUP		
fn:lower-case	OPTTLO		
fn:translate	OPTTRA		
fn:escape-uri	OPTXSOPR		String function (Escape URI)
Substring functions			
fn:contains	OPTTFN		Issue with NULL - XQuery says contains(() , "") is true ; Collation support (NLS) needed
fn:starts-with	OPTTSS, OPTFL(x+0.5)		Substring with position = 1; collation support needed
fn:ends-with	OPTTSS, OPTFL(x+0.5)		Substring with position = LENGTH(arg); collation support needed
fn:substring-before	OPTTSS, OPTTFN		OPTTSS(expr,1, OPTTFN(expr)); collation support needed
fn:substring-after	OPTTSS, OPTTFN		OPTTSS(expr,OPTTFN(expr)); collation support needed
String pattern match			

fn:matches	OPTRXLIKE		s flag matches n option; x option needs to be supported in OPTRXLIKE
fn:replace	OPTRXRPL		SQL replacement string uses \number whereas XQuery uses $number to refer to subexpressions.
fn:tokenize	OPTXSTKN		
Boolean Operations			
fn:true			
fn:false			
fn:NOT			
Date operations			
fn:get-years-from-yearMonthDuration	OPTXTRCT		
fn:get-months-from-yearMonthDuration	OPTXTRCT		
fn:get-days-from-dayTimeDuration	OPTXTRCT		
fn:get-hours-from-dayTimeDuration	OPTXTRCT		
fn:get-minutes-from-dayTimeDuration	OPTXTRCT		
fn:get-seconds-from-dayTimeDuration	OPTXTRCT		
fn:get-year-from-dateTime	OPTXTRCT		
fn:get-month-from-dateTime	OPTXTRCT		

fn:get-day-from-dateTime	OPTXTRCT		
fn:get-hours-from-dateTime	OPTXTRCT		
fn:get-minutes-from-dateTime	OPTXTRCT		
fn:get-seconds-from-dateTime	OPTXTRCT		
fn:get-timezone-from-dateTime	OPTXTRCT		Get only TZ Hour
fn:get-year-from-date	OPTXTRCT		
fn:get-months-from-date	OPTXTRCT		
fn:get-day-from-date	OPTXTRCT		
fn:get-timezone-from-date	OPTXTRCT		Get only TZ Hour
fn:get-hour-from-time	OPTXTRCT		
fn:get-minutes-from-time	OPTXTRCT		
fn:get-seconds-from-time	OPTXTRCT		
fn:get-timezone-from-time	OPTXTRCT		Get only TZ Hour
fn:adjust-dateTime-to-timezone	OPTADD		Need a wrapper. May be implemented with existing functions
fn:adjust-date-to-timezone	OPTADD		Oracle doesn't have date+timezone, only timestamp+timezone, date->timestamp, the time portion is midnight
fn:adjust-time-to-timezone	OPTADD		

fn:subtract-dateTimes-yielding-yearMonthDuration	OPTTSU		
fn:subtract-dateTimes-yielding-dayTimeDuration	OPTTSU		
QNames			
fn:resolve-qname	OPTXQNM		Qname functions
fn:expanded-qname	OPTXQNM		
fn:get-local-name-from-QName	OPTXQNM		
fn:get-namespace-uri-from-QName	OPTXQNM		
fn:get-namepace-uri-for-prefix	OPTXQNM		
fn:get-in-scope-prefixes	OPTXQNM		
fn:resolve-uri	OPTXURI		
functions on nodes			
fn:name	OPTXNODE		Node operators
fn:local-name	OPTXNODE		
fn:namespace-uri	OPTXNODE		
fn:number	OPTXT2SQLT		
fn:lang	OPTXNODE		
fn:root	OPTXNODE		
Sequence operations			
fn:zero-or-one	OPTXSOPR	ignored	Check sequence cardinality. If static typing may find that the occurance is zero or one, then this function is ignored.
fn:one-or-	OPTXSOPR	ignored	Check sequence cardinality. If static typing

more			may find that the occurance is one or one, then this function is ignored.
fn:exactly-one	OPTXSOPR	ignored	Check sequence cardinality. If static typing may find that the occurance is exactly once, then this function is ignored.
fn:boolean	OPTXGEB	ignored	Computes effective Boolean value
fn:concatenate	OPTXMLCONC		XMLConcat() may be reused
fn:index-of	OPTXSINDX		
fn:empty	IS NULL		Translated to a NOT NULL on the sequence
fn:exists	EXISTS, NOT NULL		This may be translated into the EXISTS subquery when operating on a query expression or translated to a IS NOT NULL on a variable.
fn:distinct-values	OPTXSDIST		This may be optimized into a select DISTINCT subquery in certain cases.
fn:insert-before	OPTXSOPR		Sequence operation (Insert before)
fn:remove	OPTXSOPR		Sequence operation (remove)
fn:reverse	OPTXSOPR		Sequence operation (reverse)
fn:subsequence	OPTXSOPR		Sequence operation (subsequence)
fn:unordered	ignored		Used by translation component
equals			
fn:deep-equal	OPTXDEEP		May be done using XMLType map method functions.
aggregate functions			
fn:count	OPTTCO		
fn:avg	OPTTAV		Need support for collations
fn:max	OPTTMX		-same-
fn:min	OPTTMN		-same-
fn:sum	OPTTSUM		-same-
sequence generators			
fn:id	OPTXNODE		
fn:idref	OPTXNODE		
fn:doc			Translated to (select xmlagg(res) from resource_view where equals_path(res,<arg>) = 1)
fn:collection			Translated to (select xmlagg(res) from resource_view where under_path(res,<arg>)=1)
Context positions			
fn:position			

fn:last			
fn:current-dateTime	STRTCTS		
fn:current-date	STRTCTS		
fn:current-time	STRTCTS		
fn:default-collation			
fn:implicit-timezone	OPTSESTZ		
Oracle provided functions			
ora:view			Translated to (select xmlagg(xmlelement("ROW", xmlforest(col1, col2...) from <table-name>) in case of relational tables and no xmlelement("ROW") for XMLType tables.
ora:contains	OPTXMLCONT		
ora:sqrt	OPTSQR		

[0144] The following SQL operators are also provided to perform XQuery related operations: OPTXTYPCHK performs type checking on the input so that it conforms to the given XQuery type (e.g. xs:integer). OPTXATG performs an XPath extraction operation. OPTXT2SQLT is used for casting XML type to SQL (XMLCast (xmltype expr as sqltype). OPTSQL2XMLT is used for casting SQL types to XML (XMLCast (sql-expr as xml-type)).

5.3. EXPRESSION MAPPING EXAMPLES

[0145] Some of the common expressions and their mapping are explained with examples in this section.

[0146] For example, Repository Queries (doc):

```
for $i in doc("/public/purchaseorder.xml")
where $i/PurchaseOrder/@Id eq 2001
return <PO pono={$i/PurchaseOrder/@Id}/>
```

which is rewritten to

```
select XMLAgg(XMLElement("PO", XMLAttributes(
    XMLCast (OPTXATG( OPTXATG("$i".res,'/PurchaseOrder'),'/@Id')
```

```
as number)
           as "pono")))
from (select res
      from resource_view
      where equals_path(res,'/public/purchaseorder.xml') = 1) "$i"
where XMLCast(OPTXATG("$i".res,'/PurchaseOrder/@Id') as number) =
2001;
```

gets rewritten to

```
select XMLAgg(XMLElement("PO", XMLAttributes(
            XMLCast(OPTXATG(OPTXATG(res,'/PurchaseOrder'), '@Id')
as number)
        as "pono")))
from  resource_view
   where equals_path(res,'/public/purchaseorder.xml') = 1
  and XMLCast(OPTXATG(res,'/PurchaseOrder/@Id') as number) = 2001;
```

[0147] For example, Repository (Collection):

```
for $i in collection("/public")
where $i/PurchaseOrder/@Id gt 2001
return <PO pono={$i/PurchaseOrder/@Id}/>
```

becomes

```
select XMLAgg(XMLElement("PO", XMLAttributes(
      XMLCast(OPTXATG("$i".xmlv,'/PurchaseOrder/@Id') as number)
as "pono")))
from  table(xmlsequence(select XMLAgg(res) as xmlv
      from resource_view
      where under_path(res,'/public') = 1) "$i"
   where XMLCast(OPTXATG("$i".xmlv,'/PurchaseOrder/@Id') as
number) > 2001));
```

[0148] For example, SQL Table Queries:

```
for $emp in ora:view("EMP"),
    $dept in ora:view("DEPT")
where $emp/ROW/DEPTNO = $dept/ROW/DEPTNO
return ($emp/ROW/ENAME, $dept/ROW/DNAME)
```

becomes

```
select XMLAgg(
          XMLConcat(XMLCast(OPTXATG("$emp".xmlv,'/ROW/ENAME') as
number),
                    XMLCast(OPTXATG("$dept".xmlv,'/ROW/DNAME') as
number)))
from (select XMLElement("ROW",XMLForest(empno, ename, sal,
deptno))
                as xmlv
      from emp ) "$emp",
     (select XMLElement("ROW",XMLForest(deptno, dname) as xmlv
      from dept) "$dept"
```

```
where XMLCast(OPTXATG("$emp".xmlv,'/ROW/DEPTNO') as number) =
      XMLCast(OPTXATG("$dept".xmlv,'/ROW/DEPTNO) as number);
```

which gets rewritten into

```
select XMLAgg(XMLConcat(e.ename, d.dname))
from emp e, dept d
where e.deptno =d.deptno;
```

6.0 EXAMPLE ALTERNATIVES

[0149] In the embodiments described herein, XQuery and XQueryX were presented as examples of query languages for querying XML language sources and SQL was presented as an example of a query language for querying relational databases. The techniques are in no way limited to those query languages. Any other query language may be used.

[0150] The techniques described herein present unique solutions for efficient evaluation of queries using translation. The techniques, however, are not limited to queries made on markup languages data sources. In other embodiments, any query language may be used. Queries in the query language may then be parsed and compiled into first form of in-memory representation. The first form of in-memory representation may then be converted into a second form of in-memory representation and processed further as described above.

[0151] The techniques described herein provide that the various formats of queries are first parsed and compiled into ASTs or other in-memory representations. These in-memory representations are then converted to a particular abstract syntax. In other embodiments, the elements of a query in a first syntax (e.g. XQuery) are parsed, compiled, and immediately converted to the particular format element-by-element. In the embodiment, there may not necessarily exist, at any particular time, an in-memory representation of the entire portion of the query in the first format.

7.0 HARDWARE OVERVIEW

[0152] FIG. 3 is a block diagram that illustrates a computer system 300 upon which an embodiment of the invention may be implemented. Computer system 300 includes a bus 302 or other communication mechanism for communicating information, and a processor 304 coupled with bus 302 for processing information. Computer system 300 also includes a main memory 306, such as a random access memory (RAM) or other dynamic storage device, coupled to bus 302 for storing information and instructions to be executed by processor 304. Main memory 306 also may be used for storing temporary variables or other intermediate information during execution of instructions to be executed by processor 304. Computer system 300 further includes a read only memory (ROM) 308 or other static storage device coupled to bus 302 for storing static information and instructions for processor 304. A storage device 310, such as a magnetic disk or optical disk, is provided and coupled to bus 302 for storing information and instructions.

[0153] Computer system 300 may be coupled via bus 302 to a display 312, such as a cathode ray tube (CRT), for displaying information to a computer user. An input device 314, including alphanumeric and other keys, is coupled to bus 302 for communicating information and command selections to processor 304. Another type of user input device is cursor control 316, such as a mouse, a trackball, or cursor direction keys for communicating direction information and command selections to processor 304 and for controlling cursor movement on display 312. This input device typically has two degrees of freedom in two axes, a first axis (e.g., x) and a second axis (e.g., y), that allows the device to specify positions in a plane.

[0154] The invention is related to the use of computer system 300 for implementing the techniques described herein. According to one embodiment of the invention, those

techniques are performed by computer system 300 in response to processor 304 executing one or more sequences of one or more instructions contained in main memory 306. Such instructions may be read into main memory 306 from another machine-readable medium, such as storage device 310. Execution of the sequences of instructions contained in main memory 306 causes processor 304 to perform the process steps described herein. In alternative embodiments, hard-wired circuitry may be used in place of or in combination with software instructions to implement the invention. Thus, embodiments of the invention are not limited to any specific combination of hardware circuitry and software.

[0155] The term "machine-readable medium" as used herein refers to any medium that participates in providing instructions to processor 304 for execution. Such a medium may take many forms, including but not limited to, non-volatile media, volatile media, and transmission media. Non-volatile media includes, for example, optical or magnetic disks, such as storage device 310. Volatile media includes dynamic memory, such as main memory 306. Transmission media includes coaxial cables, copper wire and fiber optics, including the wires that comprise bus 302. Transmission media can also take the form of acoustic or light waves, such as those generated during radio-wave and infrared data communications.

[0156] Common forms of machine-readable media include, for example, a floppy disk, a flexible disk, hard disk, magnetic tape, or any other magnetic medium, a CD-ROM, any other optical medium, punchcards, papertape, any other physical medium with patterns of holes, a RAM, a PROM, and EPROM, a FLASH-EPROM, any other memory chip or cartridge, a carrier wave as described hereinafter, or any other medium from which a computer can read.

[0157] Various forms of machine-readable media may be involved in carrying one or more sequences of one or more instructions to processor 304 for execution. For example, the instructions may initially be carried on a magnetic disk of a remote computer. The remote

computer can load the instructions into its dynamic memory and send the instructions over a telephone line using a modem. A modem local to computer system 300 can receive the data on the telephone line and use an infrared transmitter to convert the data to an infrared signal. An infrared detector can receive the data carried in the infrared signal and appropriate circuitry can place the data on bus 302. Bus 302 carries the data to main memory 306, from which processor 304 retrieves and executes the instructions. The instructions received by main memory 306 may optionally be stored on storage device 310 either before or after execution by processor 304.

[0158] Computer system 300 also includes a communication interface 318 coupled to bus 302. Communication interface 318 provides a two-way data communication coupling to a network link 320 that is connected to a local network 322. For example, communication interface 318 may be an integrated services digital network (ISDN) card or a modem to provide a data communication connection to a corresponding type of telephone line. As another example, communication interface 318 may be a local area network (LAN) card to provide a data communication connection to a compatible LAN. Wireless links may also be implemented. In any such implementation, communication interface 318 sends and receives electrical, electromagnetic or optical signals that carry digital data streams representing various types of information.

[0159] Network link 320 typically provides data communication through one or more networks to other data devices. For example, network link 320 may provide a connection through local network 322 to a host computer 324 or to data equipment operated by an Internet Service Provider (ISP) 326. ISP 326 in turn provides data communication services through the world wide packet data communication network now commonly referred to as the "Internet" 328. Local network 322 and Internet 328 both use electrical, electromagnetic

or optical signals that carry digital data streams. The signals through the various networks and the signals on network link 320 and through communication interface 318, which carry the digital data to and from computer system 300, are exemplary forms of carrier waves transporting the information.

[0160] Computer system 300 can send messages and receive data, including program code, through the network(s), network link 320 and communication interface 318. In the Internet example, a server 330 might transmit a requested code for an application program through Internet 328, ISP 326, local network 322 and communication interface 318.

[0161] The received code may be executed by processor 304 as it is received, and/or stored in storage device 310, or other non-volatile storage for later execution. In this manner, computer system 300 may obtain application code in the form of a carrier wave.

[0162] In the foregoing specification, embodiments of the invention have been described with reference to numerous specific details that may vary from implementation to implementation. Thus, the sole and exclusive indicator of what is the invention, and is intended by the applicants to be the invention, is the set of claims that issue from this application, in the specific form in which such claims issue, including any subsequent correction. Any definitions expressly set forth herein for terms contained in such claims shall govern the meaning of such terms as used in the claims. Hence, no limitation, element, property, feature, advantage or attribute that is not expressly recited in a claim should limit the scope of such claim in any way. The specification and drawings are, accordingly, to be regarded in an illustrative rather than a restrictive sense.

CLAIMS

What is claimed is:

1. A method of processing a query, comprising:

receiving the query, wherein the query specifies certain operations;

determining that the query comprises a first portion in a first query language and a second portion in a second query language;

generating a first in-memory representation for the first portion;

generating a second in-memory representation for the second portion;

generating a third in-memory representation of the query based on the first in-memory representation and the second in-memory representation; and

performing the certain operations based on the third in-memory representation.

2. The method of Claim 1, wherein the first in-memory representation and the third in-memory representation are formatted in a first abstract syntax and the second in-memory representation is formatted in a second abstract syntax, and wherein the step of generating the third in-memory representation comprises:

generating a fourth in-memory representation in the first abstract syntax based on the second in-memory representation; and

generating the third in-memory representation based on the first in-memory representation and the fourth in-memory representation.

3. The method of Claim 2, wherein the second in-memory representation comprises one or more in-memory representations of query elements in the second abstract syntax, and wherein generating the fourth in-memory representation comprises:

determining a second set of one or more equivalent in-memory representations of query elements in the first abstract syntax for the one or more in-memory representations of query elements in the second abstract syntax; and

generating the fourth in-memory representation in the first abstract syntax based on the second set of one or more equivalent in-memory representations of query elements in the first abstract syntax.

4. The method of Claim 3, wherein each in-memory representation of query elements in the one or more in-memory representations of query elements in the second abstract syntax corresponds to one or more in-memory representation of query elements in the second set of one or more equivalent in-memory representations of query elements in the first abstract syntax.

5. The method of Claim 1, wherein one or more of the first in-memory representation, the second in-memory representation, and the third in-memory representation are represented in memory as abstract syntax trees.

6. The method of Claim 1, wherein the first query language is Structured Query Language.

7. The method of Claim 1, where in the second query language is a markup query language.

8. The method of Claim 1, wherein the second query language is XQuery.

9. The method of Claim 1, wherein the second query language is XQueryX.

10. The method of Claim 2, wherein the first abstract syntax is an SQL-related abstract syntax and the second abstract syntax is an XQuery-related abstract syntax; wherein the second portion comprises an XQuery aggregation in the second abstract syntax; and wherein the step of generating the fourth in-memory representation comprises generating an SQL subquery in the first abstract syntax to compute the aggregation, said SQL subquery being generated based on the XQuery aggregation in the second abstract syntax.

11. The method of Claim 2, wherein the first abstract syntax is an SQL-related abstract syntax and the second abstract syntax is an XQuery-related abstract syntax; wherein the second portion comprises a literal expression in the second abstract syntax; and wherein the step of generating the fourth in-memory representation comprises generating an SQL literal in the first abstract syntax based on the literal expression in the second abstract syntax.

12. The method of Claim 2, wherein the first abstract syntax is an SQL-related abstract syntax and the second abstract syntax is an XQuery-related abstract syntax; wherein the second portion comprises an XQuery cast expression in the second abstract syntax; and wherein the step of generating the fourth in-memory representation comprises generating one of an SQL cast function and an SQL convert function in the first abstract syntax based on the XQuery cast expression in the second abstract syntax.

13. The method of Claim 2, wherein the first abstract syntax is an SQL-related abstract syntax and the second abstract syntax is an XQuery-related abstract syntax; wherein the second portion comprises a set expressions in the second abstract syntax; and wherein the step of generating the fourth in-memory representation comprises generating one of an SQL

UNION, an SQL MINUS, and an SQL INTERSECT in the first abstract syntax based on the set expressions in the second abstract syntax.

14. The method of Claim 2, wherein the first abstract syntax is an SQL-related abstract syntax and the second abstract syntax is an XQuery-related abstract syntax; wherein the second portion comprises an XQuery arithmetic expression in the second abstract syntax; and wherein the step of generating the fourth in-memory representation comprises generating an SQL arithmetic expression in the first abstract syntax based on the XQuery arithmetic expression in the second abstract syntax.

15. The method of Claim 2, wherein the first abstract syntax is an SQL-related abstract syntax and the second abstract syntax is an XQuery-related abstract syntax; wherein the second portion comprises an XQuery comparison in the second abstract syntax; and wherein the step of generating the fourth in-memory representation comprises generating an SQL comparison in the first abstract syntax based on the XQuery comparison in the second abstract syntax.

16. The method of Claim 2, wherein the first abstract syntax is an SQL-related abstract syntax and the second abstract syntax is an XQuery-related abstract syntax; wherein the second portion comprises an XQuery FLWOR order by clause in the second abstract syntax; and wherein the step of generating the fourth in-memory representation comprises generating an SQL order by clause in the first abstract syntax based on the XQuery FLWOR order by clause in the second abstract syntax.

17. The method of Claim 2, wherein the first abstract syntax is an SQL-related abstract syntax and the second abstract syntax is an XQuery-related abstract syntax; wherein the

second portion comprises an XML logical expressions in the second abstract syntax; and wherein the step of generating the fourth in-memory representation comprises generating an SQL logical expressions element in the first abstract syntax based on the XML logical expressions in the second abstract syntax.

18. The method of Claim 2, wherein the first abstract syntax is an SQL-related abstract syntax and the second abstract syntax is an XQuery-related abstract syntax; wherein the second portion comprises an XML FLWOR expression in the second abstract syntax; and wherein the step of generating the fourth in-memory representation comprises generating an SQL select expressions element in the first abstract syntax based on the XML FLWOR expression in the second abstract syntax.

19. The method of Claim 2, wherein the first abstract syntax is an SQL-related abstract syntax and the second abstract syntax is an XQuery-related abstract syntax; wherein the second portion comprises an XML Path expression in the second abstract syntax; and wherein the step of generating the fourth in-memory representation comprises generating an SQL path expression in the first abstract syntax based on the XML Path expression in the second abstract syntax.

20. The method of Claim 2, wherein the first abstract syntax is an SQL-related abstract syntax and the second abstract syntax is an XQuery-related abstract syntax; wherein the second portion comprises an XML if-then-else expression in the second abstract syntax; and wherein the step of generating the fourth in-memory representation comprises generating an SQL case-when expression in the first abstract syntax based on the XML if-then-else expression in the second abstract syntax.

21. The method of Claim 2, wherein the first abstract syntax is an SQL-related abstract syntax and the second abstract syntax is an XQuery-related abstract syntax; wherein the second portion comprises an XML quantified expression in the second abstract syntax; and wherein the step of generating the fourth in-memory representation comprises generating an SQL Exists expression in the first abstract syntax based on the XML quantified expression in the second abstract syntax.

22. The method of Claim 2, wherein the first abstract syntax is an SQL-related abstract syntax and the second abstract syntax is an XQuery-related abstract syntax; wherein the second portion comprises an SQL / XML construction expression in the second abstract syntax; and wherein the step of generating the fourth in-memory representation comprises generating an SQL construction expression in the first abstract syntax based on the SQL / XML construction expression in the second abstract syntax.

23. The method of Claim 2, wherein the first abstract syntax is an SQL-related abstract syntax and the second abstract syntax is an XQuery-related abstract syntax; wherein the second portion comprises an XML operator in the second abstract syntax; and wherein the step of generating the fourth in-memory representation comprises generating an SQL operator in the first abstract syntax based on the XML operator in the second abstract syntax.

24. The method of Claim 2, wherein the first abstract syntax is an SQL-related abstract syntax and the second abstract syntax is an XQuery-related abstract syntax; wherein the second portion comprises an XQuery sequence type operation in the second abstract syntax; and wherein the step of generating the fourth in-memory representation comprises generating an SQL type operation in the first abstract syntax based on the XQuery sequence type operation in the second abstract syntax.

25. The method of Claim 2, wherein the first abstract syntax is an SQL-related abstract syntax and the second abstract syntax is an XQuery-related abstract syntax; wherein the second portion comprises an XQuery type constructor in the second abstract syntax; and wherein the step of generating the fourth in-memory representation comprises generating an SQL scalar constructor in the first abstract syntax based on the XQuery type constructor in the second abstract syntax.

26. The method of Claim 2, wherein the first abstract syntax is an SQL-related abstract syntax and the second abstract syntax is an XQuery-related abstract syntax; wherein the second portion comprises an XQuery validate operation in the second abstract syntax; and wherein the step of generating the fourth in-memory representation comprises generating one of an SQL/XML IsValid operation and an SQL/XML Validate operation in the first abstract syntax based on the XQuery validate operation in the second abstract syntax.

27. The method of Claim 2, wherein the first abstract syntax is an SQL-related abstract syntax and the second abstract syntax is an XQuery-related abstract syntax; wherein the second portion comprises a polymorphic XQuery arithmetic operator in the

second abstract syntax; and wherein the step of generating the fourth in-memory representation comprises generating one of a polymorphic SQL arithmetic operator in the first abstract syntax based on the polymorphic XQuery arithmetic operator in the second abstract syntax.

28. The method of Claim 2, wherein the first abstract syntax is an SQL-related abstract syntax and the second abstract syntax is an XQuery-related abstract syntax; wherein the second portion comprises a polymorphic XQuery comparison operator in the second abstract syntax; and wherein the step of generating the fourth in-memory representation comprises generating one of a polymorphic SQL value comparison operator in the first abstract syntax based on the polymorphic XQuery comparison operator in the second abstract syntax.

29. The method of Claim 2, wherein the first abstract syntax is an SQL-related abstract syntax and the second abstract syntax is an XQuery-related abstract syntax; wherein the second portion comprises an XQuery function call in the second abstract syntax; and wherein the step of generating the fourth in-memory representation comprises generating an SQL function call in the first abstract syntax based on the XQuery function call in the second abstract syntax.

30. The method of Claim 2, wherein the first abstract syntax is an SQL-related abstract syntax and the second abstract syntax is an XQuery-related abstract syntax; wherein the second portion comprises a user-defined XQuery function in the second abstract syntax; and wherein the step of generating the fourth in-memory representation

comprises generating a user-defined PL/SQL function in the first abstract syntax based on the user-defined XQuery function in the second abstract syntax.

31. The method of Claim 2, wherein the first abstract syntax is an SQL-related abstract syntax and the second abstract syntax is an XQuery-related abstract syntax; wherein the second portion comprises an external XQuery function in the second abstract syntax; and wherein the step of generating the fourth in-memory representation comprises generating an external SQL function in the first abstract syntax based on the external XQuery function in the second abstract syntax.

32. The method of Claim 2, wherein the first abstract syntax is an SQL-related abstract syntax and the second abstract syntax is an XQuery-related abstract syntax; wherein the second portion comprises an XQuery module in the second abstract syntax; and wherein the step of generating the fourth in-memory representation comprises generating a PL/SQL package in the first abstract syntax based on the XQuery module in the second abstract syntax.

33. A method of processing a query comprising:

receiving the query, wherein the query specifies certain operations and is querying data stored in a markup language;

generating a first in-memory representation of the query;

generating a second in-memory representation based on the first in-memory representation, wherein the second in-memory representation is in the same form as would have been produced by processing an equivalent query that is querying a relational database; and

performing the certain operations based on the second in-memory representation.

34. The method of Claim 33, wherein the step of performing the certain operations comprises performing the certain operations by executing commands specified in the second in-memory representation against the relational database.

35. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 1.

36. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 2.

37. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 3.

38. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 4.

39. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 5.

40. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 6.

41. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 7.

42. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 8.

43. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 9.

44. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 10.

45. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 11.

46. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 12.

47. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 13.

48. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 14.

49. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 15.

50. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 16.

51. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 17.

52. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 18.

53. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 19.

54. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 20.

55. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 21.

56. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 22.

57. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 23.

58. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 24.

59. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 25.

60. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 26.

61. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 27.

62. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 28.

63. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 29.

64. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 30.

65. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 31.

66. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 32.

67. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 33.

68. A computer-readable medium carrying one or more sequences of instructions which, when executed by one or more processors, causes the one or more processors to perform the method recited in Claim 34.

FIG. 1

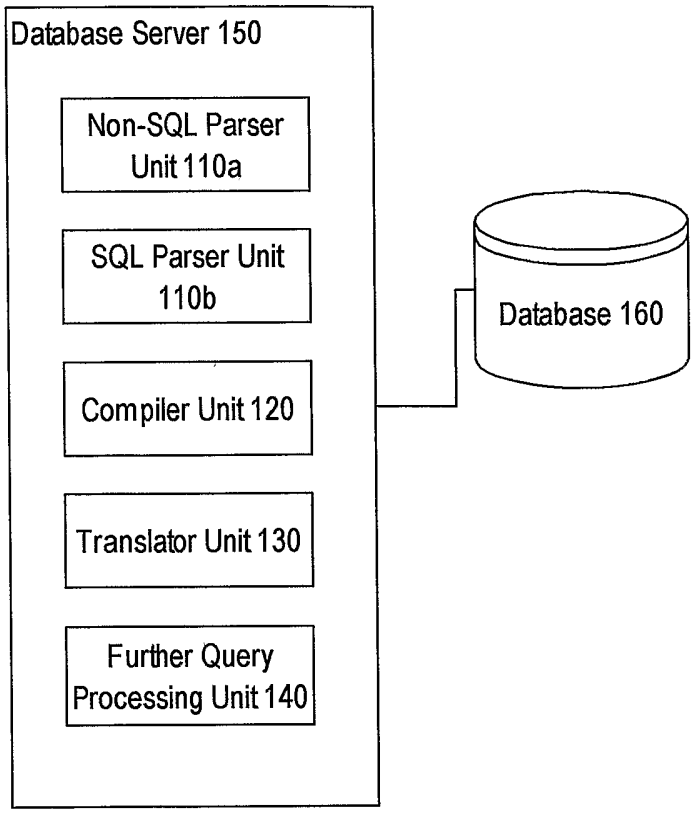

Database Server 150
Non-SQL Parser Unit 110a
SQL Parser Unit 110b
Compiler Unit 120
Translator Unit 130
Further Query Processing Unit 140
Database 160

FIG. 2

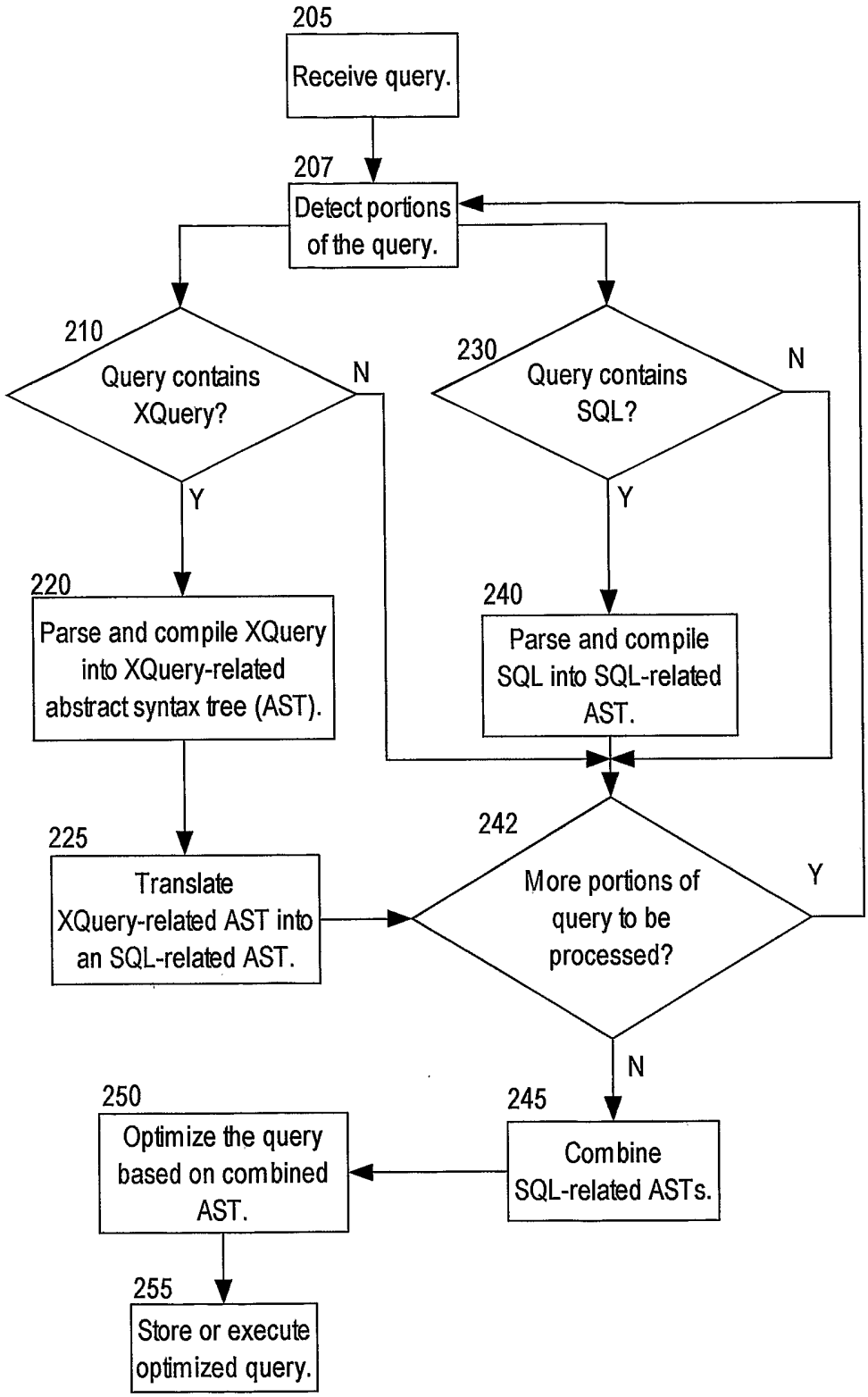
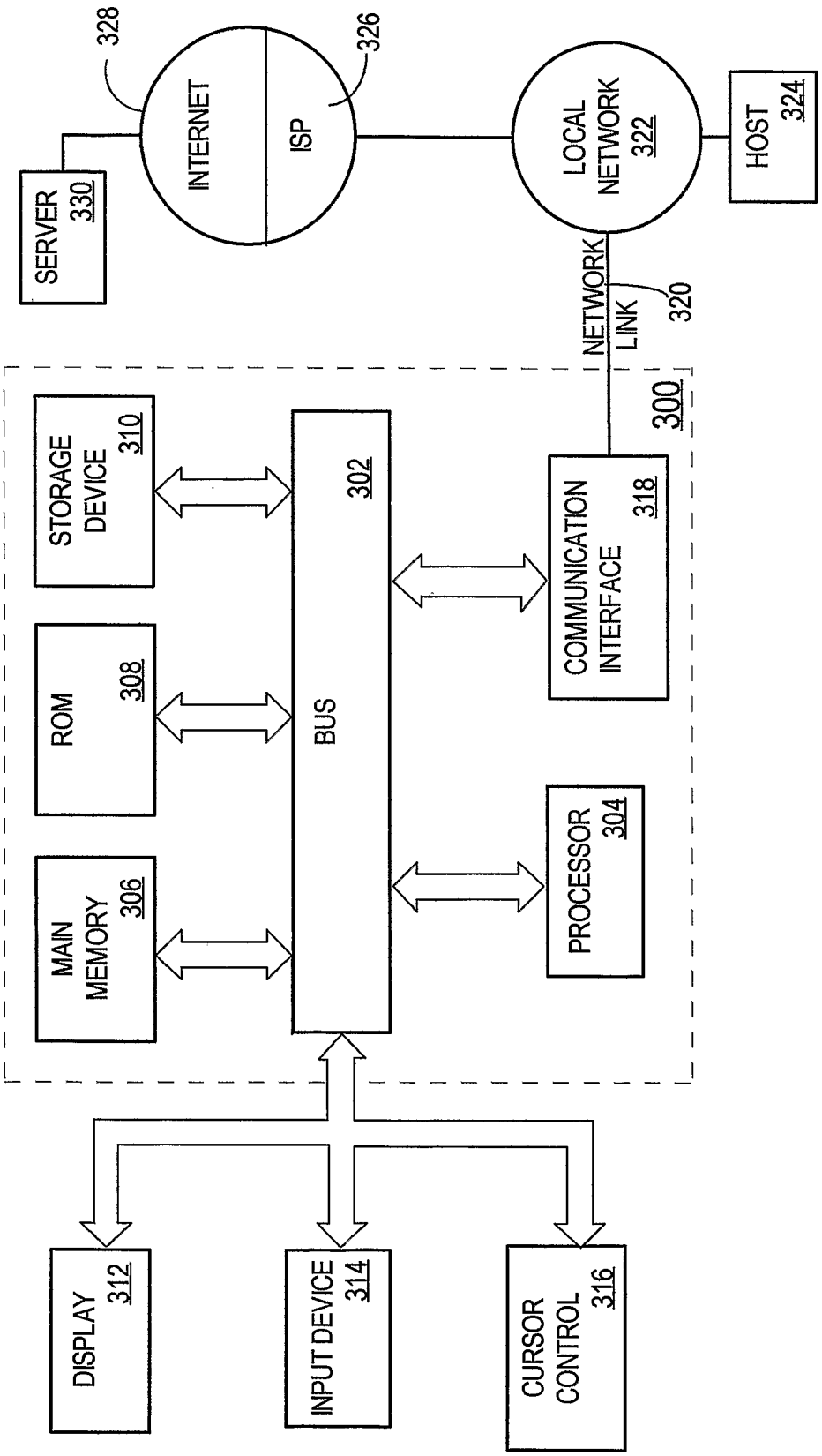

205
Receive query.
207
Detect portions of the query.
210
Query contains XQuery?
N
Y
220
Parse and compile XQuery into XQuery-related abstract syntax tree (AST).
225
Translate XQuery-related AST into an SQL-related AST.
230
Query contains SQL?
N
Y
240
Parse and compile SQL into SQL-related AST.
242
More portions of query to be processed?
Y
N
245
Combine SQL-related ASTs.
250
Optimize the query based on combined AST.
255
Store or execute optimized query.

FIG. 3


328
INTERNET
ISP
326
SERVER 330
LOCAL NETWORK 322
HOST 324
NETWORK LINK 320
300
STORAGE DEVICE 310
ROM 308
MAIN MEMORY 306
BUS 302
COMMUNICATION INTERFACE 318
PROCESSOR 304
DISPLAY 312
INPUT DEVICE 314
CURSOR CONTROL 316

INTERNATIONAL SEARCH REPORT

International Application No
PCT/US2005/021259

A. CLASSIFICATION OF SUBJECT MATTER
G06F17/30

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)
G06F

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

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)
EPO-Internal, INSPEC

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	FUNDERBURK, J. E:, MALAIKA, S., REINWALD, B.: "XML Programming with SQL/XML and XQuery" IBM SYSTEMS JOURNAL, vol. 41, no. 4, 2002, XP002353603 abstract page 646 page 662; figure 4	1-68
Y	MURTHY, RAVI, BANERJEE, SANDEEPAN: "XML Schemas in Oracle XML DB" PROC. OF THE 29TH VLDB CONFERENCE, 9 September 2003 (2003-09-09), pages 1009-1018, XP002353604 Berlin page 1010 page 1015	1-68

-/--

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

° Special categories of cited documents:

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
16 November 2005	24/11/2005

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL – 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	San-Bento Furtado, P

6

Form PCT/ISA/210 (second sheet) (January 2004)

INTERNATIONAL SEARCH REPORT

International Application No
PCT/US2005/021259

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
Y	ZHANG, HUI, TOMPA, FRANK: "XQuery rewriting at the relational algebra level" Computer Systems Science and Engineering CRL Publishing UK, vol. 18, no. 5, September 2003 (2003-09), pages 241-262, XP009056809 ISSN: 0267-6192 page 241 - page 244	1-68
Y	SHANMUGASUNDARAM J ET AL: "Querying XML Views of Relational Data" PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 2001, pages 1-10, XP002313815 abstract; figure 5 page 4, left-hand column page 10	1-68
A,Y	CHOI, BYRON, FERNANDEZ, MARY, SIMEON,JEROME: "The XQuery Formal Semantics: A Foundation for Implementation and Optimization"'Online! 31 May 2002 (2002-05-31), XP002353605 Retrieved from the Internet: URL:http://www-db.research.bell-labs.com/user/simeon/xquery-optim-planx.pdf> 'retrieved on 2005-11-11! abstract page 1, last paragraph page 7 - page 8; figure 3	1-68
Y	US 2001/037345 A1 (KIERNAN GERALD GEORGE ET AL) 1 November 2001 (2001-11-01) abstract paragraph '0026! paragraph '0048! paragraph '0140! - paragraph '0146!	1-68
Y	ZHANG XIN ET AL ASSOCIATION FOR COMPUTING MACHINERY: "Honey, I Shrunk the XQuery - An XML Algebra Optimization Approach" PROCEEDINGS OF THE 4TH. INTERNATIONAL WORKSHOP ON WEB INFORMATION AND DATA MANAGEMENT. (WIDM 2002). MCLEAN, VA, NOV. 8, 2002, PROCEEDINGS OF THE INTERNATIONAL WORKSHOP ON WEB INFORMATION AND DATA MANAGEMENT. (WIDM), NEW YORK, NY : ACM, US, 8 November 2002 (2002-11-08), pages 1-16, XP002316448 ISBN: 1-58113-593-9 abstract page 5, last paragraph - page 6	1-68

-/--

6

INTERNATIONAL SEARCH REPORT

International Application No
PCT/US2005/021259

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	JI-HOON KANG ET AL: "An xquery engine for digital library systems that support xml data" APPLICATIONS AND THE INTERNET WORKSHOPS, 2004. SAINT 2004 WORKSHOPS. 2004 INTERNATIONAL SYMPOSIUM ON 26-30 JAN. 2004, PISCATAWAY, NJ, USA,IEEE, 26 January 2004 (2004-01-26), pages 233-237, XP010684128 ISBN: 0-7695-2050-2 the whole document	1-68

6

Form PCT/ISA/210 (continuation of second sheet) (January 2004)

INTERNATIONAL SEARCH REPORT

International Application No
PCT/US2005/021259

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2001037345 A1	01-11-2001	US 6947945 B1	20-09-2005