Inverse Functions in the AquaLogic Data Services Platform

Nicola Onose*
University of California
San Diego, USA
nicola@cs.ucsd.edu

Vinayak Borkar Michael J. Carey
BEA Systems, Inc
San Jose, USA
{vborkar,mcarey}@bea.com

ABSTRACT

When integrating data from heterogeneous sources, it is often necessary to transform both the schemas and the data from the underlying sources in order to present the integrated data in the form desired by its consuming applications. Unfortunately, these transformations—particularly if implemented by custom code—can block query optimization and updates, leading to potentially severe performance and functionality limitations. To circumvent these problems, the BEA AquaLogic Data Services Platform provides support for user-defined inverse functions. This paper describes the motivation, design, user experience, and implementation associated with inverse functions in ALDSP. This functionality debuted in version 2.1 of ALDSP in March 2006.

1. INTRODUCTION

Developers of data-centric enterprise applications are facing a crisis today. Relational databases have been so successful that there are many different systems available (Oracle, DB2, SQL Server, and MySQL, to name a few), and any given enterprise is likely to have a number of different relational database systems and databases within its corporate walls. Moreover, information about key business entities such as customers or employees is likely to reside in several such systems. In addition, while most "corporate jewel" data is stored relationally, much of it is not relationally accessible – it is owned by packaged applications such as SAP, Oracle Financials, PeopleSoft, Siebel, Clarify, or SalesForce.com, or custom homegrown applications. These applications add meaning to the stored data by enforcing the business rules and controlling the logic of the *business objects* of the application. Meaningful access to their data requires calling the functions of the applications' APIs. As a result, enterprise application developers face a major integration challenge today: bits and pieces of any given business entity reside in a mix of relational databases, packaged applications, and perhaps even in files or in legacy mainframe systems and/or applications.

*Work carried out while author was an intern at BEA Systems.

To create new, *composite* applications from disparate parts, XML-based Web services [1, 2] are a piece of the puzzle. Web services provide a degree of physical normalization for intra- and inter-enterprise function invocation and information exchange. In order to provide proper support for the data side of composite application development, however, we need more – we need a declarative way to create *data services* [3] for use in composite applications. The composite application approach that we are taking at BEA is to ride the wave created by Web services and associated XML standards. We are using the W3C XML, XML Schema, and XQuery Recommendations as a standards-based foundation for declarative data services development [4]. The BEA AquaLogic Data Services Platform (ALDSP), first introduced in 2005, supports a declarative approach to designing and developing data services [5]. ALDSP is aimed at developers of composite applications that need to access and compose information from a range of enterprise data sources, including packaged applications, relational databases, Web services, and files, as well as other sources.

When designing and implementing data services, one of the main goals is to provide a set of abstractions so that applications can see and manipulate integrated enterprise data in a clean, unified, meaningful, canonical form. Doing so invariably requires transforming data – restructuring and unifying the schemas and the instance-level data formats of the disparate data sources. Names are reformatted, addresses normalized, differences in units reconciled, and so on, all in order to provide application developers (the consumers of data services) with a natural and easily manipulable view of the underlying data. Such transformations, while highly useful to the end consumers of the data, can lead to performance challenges. When the resulting data is queried, it is crucial for performance that much of the query processing (especially for selections and joins) be pushable to the underlying sources, particularly the relational sources. It is also important that it be possible for updates to the transformed view of the data be translatable into appropriate source updates. Unfortunately, if data transformations are written in a general-purpose programming language such as Java, this can be difficult due to the fact that the transformations are opaque to the query processor.

To permit general user-defined data transformations without sacrificing query pushdown and updatability, ALDSP provides a means for a data service developer to register *inverse functions* with the system so that it can un-transform

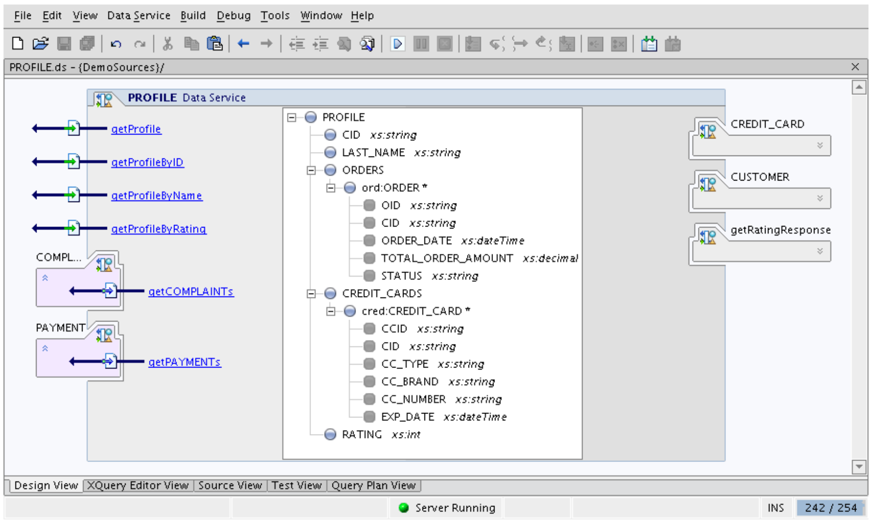


Figure 1: ALDSP Data Service – Design View

data when pushing query predicates or decomposing updates into the required underlying data source updates. The remainder of this paper describes inverse functions in ALDSP, including typical use cases, the user experience for developers, the use of inverse functions in query and update processing, and the performance benefits that they provide. Section 2 sets the stage with an overview of ALDSP. Section 3 describes the data service developer's view of inverse functions, showing in detail how they can be used to tackle several typical data integration use cases. Section 4 describes how inverse functions are actually utilized during query processing, explaining how they were added to ALDSP's rule-based query optimizer and update decomposition component. Section 5 presents a set of experimental ALDSP performance measurements that illustrate the importance of inverse functions. Section 6 concludes the paper.

2. ALDSP: A WHIRLWIND TOUR

To provide the context for our treatment of inverse functions and their use in query and update processing in ALDSP, it is important to first understand the ALDSP world model and system architecture.

2.1 Modeling Data and Services

ALSDSP targets the SOA world, so it is based on a service-oriented view of data. ALDSP models the enterprise (or a portion of interest of the enterprise) as a set of interrelated *data services* [4]. Each data service is a set of service calls that an application can use to access and modify instances of a particular coarse-grained business object type (e.g., customer, order, employee, or service case). A data service has a "shape", which characterizes the information content of its business object type; ALDSP uses XML Schema to describe each data service's shape. A data service also has a set of read methods, the service calls that provide various ways to request access to one or more instances of the data service's business objects. In addition, a data service has a set of write methods, the service calls that support updating (e.g., modifying, inserting, or deleting) one or more instances of the data service's business objects. Last but not least, a data service has a set of navigation methods, the service calls that traverse relationships from a business object returned by the data service (e.g., customer) to one or more business object instances from a second data service (e.g., order). Each of the methods associated with a data service is realized as an XQuery function that can be called in queries and/or used in the creation of other, higher-level logical data services.

Figure 1 shows a screen capture of the design view of a simple data service. In the center of the design view is the shape of the data service. The left-hand side of the figure shows the service calls that are provided for users of the data service, including the read methods (upper left) and navigation methods (lower left). The objective of an ALDSP data service architect/developer is to design and implement a set of data services, like the one shown, that together provide a clean, reusable, and service-oriented "single view" of some portion of an enterprise. (As we will see, that is where user-defined transformation functions and inverses become important.) The right-hand side of the design view shows the dependencies that this data service has on other data services that were used to create it.

When pointed at a data source by a data service developer, ALDSP introspects the data source's metadata (e.g., SQL metadata for a relational data source or WSDL files for a Web service). This introspection guides the automatic creation of one or more physical data services that make the source available for use in ALDSP. Introspecting a relational data source yields one data service (with one read method and one update method) per table or view. The shape in this case corresponds to the natural, typed XML-ification of a row of the table. In the presence of foreign key constraints, introspection also produces navigation functions that encapsulate the join paths provided by the constraints. Introspecting a Web service (WSDL) yields one data service per distinct Web service operation return type. The data service functions correspond to the Web service's operations, and the functions' input and output types correspond to the schema information in the WSDL. Other functional data sources are modeled similarly. The result is a uni-

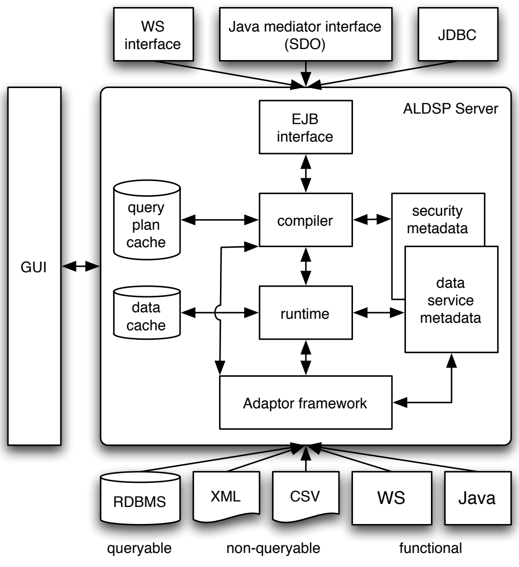


Figure 2: Overview of ALDSP Architecture

form, "everything is a data service" view of an enterprise's data sources that is well-suited for further use in composing higher-level data services using XQuery.

2.2 ALDSP Architecture

Figure 2 depicts the architecture of ALDSP. At the bottom of the picture are the various kinds of data sources that ALDSP supports. The data source types are grouped into three categories – queryable, non-queryable, and functional. Queryable sources are sources to which ALDSP can delegate query processing; relational databases fall into this category. Non-queryable sources are sources from which ALDSP can access the full content of the source but which do not support queries; XML and delimited files belong in this category. Functional sources are sources which ALDSP can only interact with by calling specific functions with parameters; Web services, Java functions, and stored procedures all fall into this category. In the world of SOA, this last source category is especially important, as most packaged and home-grown applications fit in here. Also, it is common for this category of source to return complex, structured results (e.g., a purchase order document obtained from a call to an order management system). To facilitate declarative integration and data service creation, all data sources are presented to ALDSP developers uniformly as external XQuery functions that have (virtual) XML inputs and outputs.

Sitting above the data source level in Figure 2 is the ALDSP adaptor framework, which connects ALDSP to the available data sources. Adaptors have a design-time component that introspects data source metadata to extract the information needed to create the typed XQuery function models for sources. They also have a runtime component that controls and manages source access at runtime. Above the adaptor layer is a middleware query processing subsystem that consists of a query compiler and a runtime system. The query compiler is responsible for translating XQuery queries and function calls into efficient executable query plans. To do its job, it must refer to metadata about the various data services in the enterprise as well as to security metadata that controls who has access to which ALDSP data. Also, ALDSP maintains a query plan cache in order to avoid repeatedly compiling popular queries from the same or different users. The runtime system is made up of a collection of XQuery functions and query operators that can be combined to form query plans; the runtime also controls the execution of such plans and the resources that they consume. In addition, the ALDSP runtime is responsible for accepting updates and propagating the changes back to the affected underlying data sources. More details about the ALDSP query processing architecture are available in [6].

The central box in Figure 2 is the ALDSP server. The server is made up of the components described in the previous paragraph, and it has a remote (EJB) client-server API that is shared by all of ALDSP's client interfaces. These include a Web service interface, a Java mediator interface (based on Service Data Objects [7], a.k.a. SDO), and a JDBC/SQL interface. The SDO-based Java mediator interface allows Java client programs to call data service methods as well as to submit ad hoc queries. In the method call case, a degree of query flexibility remains, as the mediator API permits clients to include result filtering and sorting criteria along with their request. The ALDSP server also has two graphical interfaces, a design-time data service designer that resides in the BEA WebLogic Workshop IDE and a runtime administration console for configuring logical and physical ALDSP server resources.

3. INVERSE FUNCTIONS IN ALDSP

We now examine the inverse function feature of ALDSP from a data architect's perspective. We will describe several typical use cases and see how inverse functions address them, looking at the steps required to define them in ALDSP as well as the XQuery and metadata artifacts that result from doing so. As mentioned in the introduction, the motivation for adding inverse functions to ALDSP was to allow data service developers to define and utilize custom, user-defined data transformations without sacrificing query performance or updatability. It is important to note that this ALDSP feature was customer-driven—it was developed and added to the product in response to several ALDSP 2.0 customers who were defining their own data transformations in Java and then struggling with the query pushdown and update limitations that resulted from their opacity.

3.1 An Example Scenario

To provide concrete examples of typical inverse function use cases, let us assume that ALDSP is being asked to integrate customer-related data from a number of sources, one of which is the following table from a relational data source:

```
CREATE TABLE WESTCUSTOMER (
   CUSTID VARCHAR(10) NOT NULL,
   FNAME VARCHAR(20),
   LNAME VARCHAR(20),
   MONTHLYSAL INTEGER,
   HIRED INTEGER );
```

Our examples will be based on scenarios involving the integration of data about individuals who are customers of a hypothetical Internet job placement firm, GotJobs.com. The table shown above is one of GotJobs' data sources, and it contains information about the customers in the Western region of the firm's business geography. Each customer

```
xquery version "1.0" encoding "WINDOWS-1252";

(::pragma ... ::)
declare namespace ns4=
 ``lib:EastWestInverseDataServices/DateLibrary";
declare namespace ns3=
 ``lib:EastWestInverseDataServices/NameLibrary";
declare namespace ns2=
 ``ld:EastWestInverseDataServices/WESTCUSTOMER";

import schema namespace ns0=... at ...;
declare namespace ns1=...;

(::pragma function <f:function kind=``read" ... ::)
declare function ns1:getWestCustomers()
     as element(ns0:WestCustomerView)*
{
 for $WESTCUSTOMER in ns2:WESTCUSTOMER()
 return
 <ns0:WestCustomerView>
   <customerId>
     {fn:data($WESTCUSTOMER/CUSTID)}
   </customerId>
   <fullName?>{ns3:fullname
     ($WESTCUSTOMER/LNAME, $WESTCUSTOMER/FNAME)}
   </fullName>
   <monthlySalary?>
     {fn:data($WESTCUSTOMER/MONTHLYSAL)}
   </monthlySalary>
   <dateHired?>
     {ns4:y2kdate($WESTCUSTOMER/HIRED)}
   </dateHired>
 </ns0:WestCustomerView>
};

(::pragma function <f:function kind=``read" ... ::)
declare function ns1:getWestCustomersByName
  ($fullname as xs:string)
     as element(ns0:WestCustomerView)* {
  for $WestCustomerView in ns1:getWestCustomers()
  where $fullname = $WestCustomerView/fullName
  return $WestCustomerView
};

(::pragma function <f:function kind=``read" ... ::)
declare function ns1:getOldWestCustomers
  ($beforedate as xs:dateTime)
     as element(ns0:WestCustomerView)* {
  for $WestCustomerView0 in ns1:getWestCustomers()
  where $WestCustomerView/dateHired lt $beforedate
  return $WestCustomerView0
};
```

Figure 3: WestCustomers Dataservice

record has a customer id and a separately stored first and last name. In addition, being a job placement firm, the data that GotJobs keeps about each customer includes their starting salary and date of hire. The salary data is stored as a monthly amount, in dollars. Being a post-Internet-bubble firm, the date of hire data is encoded as a simple integer and expressed in terms of the number of days since Y2K (i.e., days since January 1, 2000).

To set the stage for our data integration and data transformation examples, let us suppose that GotJobs is a large firm, with customers located in all regions of the country. GotJobs has grown incrementally, originating in the West but then adding other regional job placement firms to its portfolio. As is often the case with mergers and acquisitions, the other divisions of GotJobs each have their own customer data with similar information content but with differences in some of the underlying data representation details. GotJobs is now in the process of cleaning up its enterprise IT situation, and one of the tasks involved is creating a (virtual)

```
package JavaFuncs;

public class Dates {

  private static final Calendar Y2K =
    new GregorianCalendar(2000,
            Calendar.JANUARY, 1);

  public static Calendar y2kdate
        (Integer elapseddays) {
   try {
      Calendar date = (Calendar) Y2K.clone();
        date.add(Calendar.DATE,
                elapseddays.intValue());
        return date;
    } catch (Exception e) {
        return null;
      }
  }

  public static Integer y2kdays(Calendar date) {
    try {
      long y2kMillis = Y2K.getTime().getTime();
      long dateMillis = date.getTime().getTime();
      long delta = (dateMillis - y2kMillis) /
                   (1000L*60L*60L*24L);
      return new Integer((int) delta);
    } catch (Exception e) {
        return null;
      }
   }
}
```

Figure 4: Implementation of Time Transformations

single view of customer that will allow its customer data to still reside in its divisional IT systems while also being accessible and updatable uniformly by the new, corporate-level applications that GotJobs is developing. GotJobs is "going SOA" for its new applications, so the single view of customer is being developed using ALDSP and its data services methodology.

The chief data architect for GotJobs has decided to create, as part of a layered data architecture, a set of uniform data services, one per region, that present each region's customers in an enterprise-wide canonical form. After analyzing the various regions' customer schemas, and considering the needs of the new applications, it has been decided that the date of hire for customers should be represented more traditionally, as a date value, and that customers' names should be surfaced as a single string value in lastname, firstname format. Each region will start by constructing a customer data service that normalizes its regional information and provides access to it in this canonical form. Access to the resulting normalized data is to be parameterized on various search criteria, e.g., by name, by date of hire, by customer id, and so on.

Figure 3 shows a concrete example of what the canonical data service looks like for the West Coast region of the company, showing ALDSP source code for the WestCustomers data service (based on the WESTCUSTOMER table). To keep the example short, only three data service operations are shown—-one that provides access to the normalized form of all West Coast customers, one that provides searched access by full name, and another that provides searched access based on date of hire. The West Coast customer data is accessed using the function WESTCUSTOMER() that resulted from asking ALDSP to introspect the WESTCUSTOMER table to make it available as a physical data service for this data services project. A closer look at the body of the

```
package JavaFuncs;

public class LastNameFirstName
{
  public static String fullname(String ln,
                    String fn) {
     return (ln == null || fn == null) ?
          null : (ln + ", " + fn);
  }

  public static String firstname(String name) {
    try { return name.substring
      ( name.indexOf(',') + 2);
    } catch (Exception e) { return null; }
  }

  public static String lastname(String name) {
    try {
      int k = name.indexOf(',');
      return name.substring( 0, k );
    }  catch (Exception e) { return null; }
  }
}
```

Figure 5: Implementation of Name Transformations

getWestCustomers() operation shows how the required normalization is handled—-GotJobs already had Java functions that normalize hire dates (y2kdate) and names (fullname), and these were introspected by ALDSP and then utilized in the definition of getWestCustomers().[1] Figure 4 shows GotJobs' Java class for handling dates, and Figure 5 shows their Java class for handling name transformations. Once introspected by ALDSP, these functions become available for use in XQuery, as illustrated in the getWestCustomers() method of Figure 3. Notice that the other two WestCustomers search functions are then expressed very succinctly in terms of this main integration function, both for simplicity of expression and to avoid replicating its integration/transformation logic (so that subsequent changes to the logic will be isolated if the GotJobs IT architecture evolves).

Given the work done so far by the GotJobs IT department, we now have a working WestCustomers data service that normalizes the West Coast customer data and provides searched access. However, a significant problem remains—performance. With only the data service definition and Java functions of Figures 3 through 5, ALDSP can call the Java functions but cannot fully optimize queries that involve them. For example, if the data service operation getWestCustomersByName() is invoked, since the query predicate is on the result of calling the Java function fullname(), ALDSP will end up streaming all of the WESTCUSTOMER rows into the middle tier in order to compute each customer's full name and compare it to the function parameter. Likewise for requests to run the data service operation getOldWestCustomers()—lacking any additional information, its invocation will require mid-tier evaluation of the Java function y2kdate to convert the WESTCUSTOMER integer hire date into a regular date for comparison to the function's input parameter. There is another significant problem as well—updates. In the absence of additional information, neither the fullName nor dateHired elements in the single view of customer will be updatable since they are the result of opaque Java function calls.

[1]The Java functions used for the examples in this section are intentionally simple; in practice, such functions can be much more complex internally.

This is where ALDSP's inverse function support comes in. In the remainder of this section, we will examine how inverse functions and related semantic rules can be provided to ALDSP to help with such use cases. We will then look at how inverse function definitions are surfaced to data architects working in the context of ALDSP's graphical data service design environment.

```
declare namespace f1 = ...

(::pragma function
<f:function nativeName="y2kdate" ...
 xmlns:dat= ...>
 <inverseFunctions>
  <inverseFunction name="dat:y2kdays"
   parameterIndex="1"/>
 </inverseFunctions>
 <equivalentTransforms>
   <pair source="xpat:dateTime-greater-than"
    target="day:dateGT" arity="2"/>
   <pair source="xpat:dateTime-less-than"
    target="day:dateLT" arity="2"/>
   <pair source="xpat:dateTime-equal"
    target="day:dateEQ" arity="2"/>
 </equivalentTransforms>
 <params>
   <param nativeType="java.lang.Integer"/>
 </params>
</f:function>
::)
declare function f1:y2kdate($x1 as xsd:int?)
 as xsd:dateTime? external;

(::pragma function
<f:function nativeName="y2kdays" ...
 xmlns:dat= ...
 <params>
  <param nativeType="java.util.Calendar"/>
 </params>
</f:function>
::)
declare function f1:y2kdays($x1 as xsd:dateTime?)
 as xsd:int? external;

(::pragma function ... ::)
declare function f1:dateEQ($date1 as xsd:dateTime?,
     $date2 as xsd:dateTime?) as xsd:boolean? {
 f1:y2kdays($date1) eq f1:y2kdays($date2)
};

(::pragma function ... ::)
declare function f1:dateLT($date1 as xsd:dateTime?,
     $date2 as xsd:dateTime?) as xsd:boolean? {
 f1:y2kdays($date1) lt f1:y2kdays($date2)
};

(::pragma function ... ::)
declare function f1:dateGT($date1 as xsd:dateTime?,
     $date2 as xsd:dateTime?) as xsd:boolean? {
 f1:y2kdays($date1) gt f1:y2kdays($date2)
};
```

Figure 6: XQuery Library for Time Transformations

3.2 1:1 Data Transformations

Let us first examine the hiring date use case in our example. This use case involves what we call a 1:1 transformation: The Java function y2kdate transforms one value (an integral number of days since Y2K) into a different, but equivalent, value (a calendar date). In order for ALDSP to optimize predicates or perform updates involving getWestCustomers()/dateHired, the system needs to be able to invert the y2kdate function—it needs to know that the Java function y2kdays is its inverse, and can thus be used to turn a

```
declare namespace f1 = ...

(::pragma function
<f:function nativeName=''fullname'' ...
  <inverseFunctions>
    <inverseFunction name=''nam:lastname''
      parameterIndex=''1''/>
    <inverseFunction name=''nam:firstname''
      parameterIndex=''2''/>
  </inverseFunctions>
  <equivalentTransforms>
    <pair source=''xqu:string-greater-than''
      target=''nam:fullnameGT'' arity=''2''/>
    <pair source=''xqu:string-less-than''
      target=''nam:fullnameLT'' arity=''2''/>
    <pair source=''xqu:string-equal''
      target=''nam:fullnameEQ'' arity=''2''/>
  </equivalentTransforms>
  <params>
    <param nativeType=''java.lang.String''/>
    <param nativeType=''java.lang.String''/>
  </params>
</f:function>
::)
declare function f1:fullname($last as xsd:string?,
                    $first as xsd:string?)
  as xsd:string? external;

(::pragma function ... ::)
declare function f1:firstname($full as xsd:string?)
  as xsd:string? external;

(::pragma function ... ::)
declare function f1:lastname($full as xsd:string?)
  as xsd:string? external;

(::pragma function ... ::)
declare function f1:fullnameEQ($fullname1 as xsd:string?,
      $fullname2 as xsd:string?) as xsd:boolean? {
   (f1:lastname($fullname1) eq f1:lastname($fullname2)) and
   (f1:firstname($fullname1) eq f1:firstname($fullname2))
};

(::pragma function ... ::)
declare function f1:fullnameLT($fullname1 as xsd:string?,
      $fullname2 as xsd:string?) as xsd:boolean? {
   (f1:lastname($fullname1) lt f1:lastname($fullname2)) or
   ((f1:lastname($fullname1) eq f1:lastname($fullname2)) and
    (f1:firstname($fullname1) lt f1:firstname($fullname2)))
};

(::pragma function ... ::)
declare function f1:fullnameGT($fullname1 as xsd:string?,
      $fullname2 as xsd:string?) as xsd:boolean? {
   (f1:lastname($fullname1) gt f1:lastname($fullname2)) or
   ((f1:lastname($fullname1) eq f1:lastname($fullname2)) and
    (f1:firstname($fullname1) gt f1:firstname($fullname2)))
};
```

Figure 7: XQuery Library for Name Transformations

calendar date into a number of days since Y2K. In addition, for query optimization, the system needs to be told how to convert calendar date comparisons into comparisons on the numbers of days since Y2K—it needs equivalences to be defined for any boolean comparisons that might appear in query predicates for which the data service developer wants ALDSP to be able to perform SQL pushdown optimizations. The inverse function feature of ALDSP provides the means for a data service developer to declare these additional pieces of information to the system.

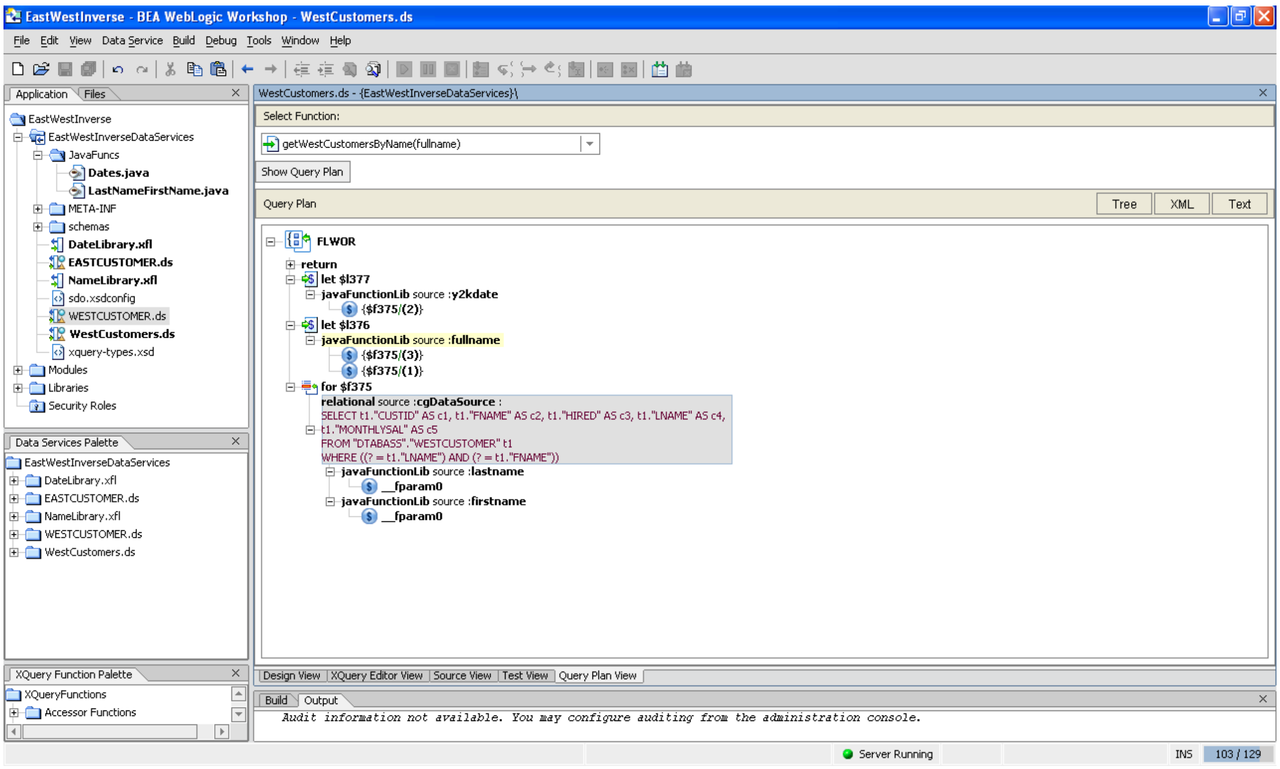
Figure 6 shows the ALDSP XQuery definitions (in an XQuery Function Library, or XFL) needed for the Java date transformation function use case. The initial version of this file was produced automatically when ALDSP was directed to introspect the Java class file for the Dates class of Figure 4. Such an XFL file initially contains one XQuery function per Java function, with those functions being flagged as external in their XQuery function signatures. This is how the functions y2kdate and y2kdays came into existence in the XFL file shown in Figure 6. The other functions shown there, as well as the <inverseFunction> and <equivalentTransforms> annotations in the <inverseFunctions> function annotation above the y2kdate function, were then added by the data service developer. The <inverseFunction> annotation declares y2kdays as being inversely related to the function y2kdate. (Ignore the parameterIndex attribute; we will explain its purpose in the next subsection.) The XQuery functions dateEQ, dateLT, and so on in Figure 6 each define ways to compare calendar dates by instead transforming them into integer days-since-Y2K values and comparing the resulting values. Last but not least, the <equivalentTransforms> annotation above the function y2kdate in the figure defines these XQuery functions as being semantically equivalent to performing the same comparisons using XQuery's built-in calendar date comparisons.

3.3 1:N Data Transformations

Let us now examine the name formatting use case in our example. This use case involves a 1:N transformation: The Java function fullname transforms a sequence of two values (the function's last and first name parameter strings) into a single, different, but equivalent, value (a full name string). In order for ALDSP to optimize query predicates or translate updates involving getWestCustomers()/fullName, the system needs to be able to invert the fullname function—it needs to know that the pair of Java functions lastname and firstname together form the inverse of fullname, and can therefore be used to turn a full name into a separated pair of values for last name and first name. As in the 1:1 case, for query optimization, the system also needs to know how to convert full name comparisons into a combination of comparisons on last and first names. The inverse function feature of ALDSP is designed to handle such 1:N transformations as well.

Figure 7 shows the ALDSP source code for the XQuery Function Library (XFL) for the Java name transformation functions. As in the previous case, this file was created initially by directing ALDSP to introspect a Java class file, in this case the class file for the LastNameFirstName class of Figure 5. The data service developer then added the XQuery-bodied comparison functions and the inverse declaration and equivalent transforms annotations on the fullname function in the XFL. For the most part, the XFL file in Figure 7 is similar to that of the 1:1 use case. However, notice that the <inverseFunctions> annotation block now contains two < inverseFunction > sub-annotations—indicating that to invert the function fullname(p1,p2), the functions lastname(p1) and firstname(p2) (in that order, parameter-index-wise) are both involved.

To illustrate the effect of the inverse functions and equivalent transforms for this more complex use case, Figure 8 shows the query execution plan produced by ALDSP for a call to the getWestCustomersByName() function from the WestCustomers data service. With the additional knowledge provided by these declarations, instead of having to examine all WESTCUSTOMER rows, ALDSP is able to push a SQL query with the predicate "WHERE ((? = t1.LNAME)



```
FLWOR
  return
  let $l377
    javaFunctionLib source :y2kdate
      {$f375/(2)}
  let $l376
    javaFunctionLib source :fullname
      {$f375/(3)}
      {$f375/(1)}
  for $f375
    relational source :cgDataSource :
    SELECT t1."CUSTID" AS c1, t1."FNAME" AS c2, t1."HIRED" AS c3, t1."LNAME" AS c4,
    t1."MONTHLYSAL" AS c5
    FROM "DTABASS"."WESTCUSTOMER" t1
    WHERE ((? = t1."LNAME") AND (? = t1."FNAME"))
      javaFunctionLib source :lastname
        __fparam0
      javaFunctionLib source :firstname
        __fparam0
```

```
Audit information not available. You may configure auditing from the administration console.
```

Figure 8: Optimized Plan After Pushing Selections

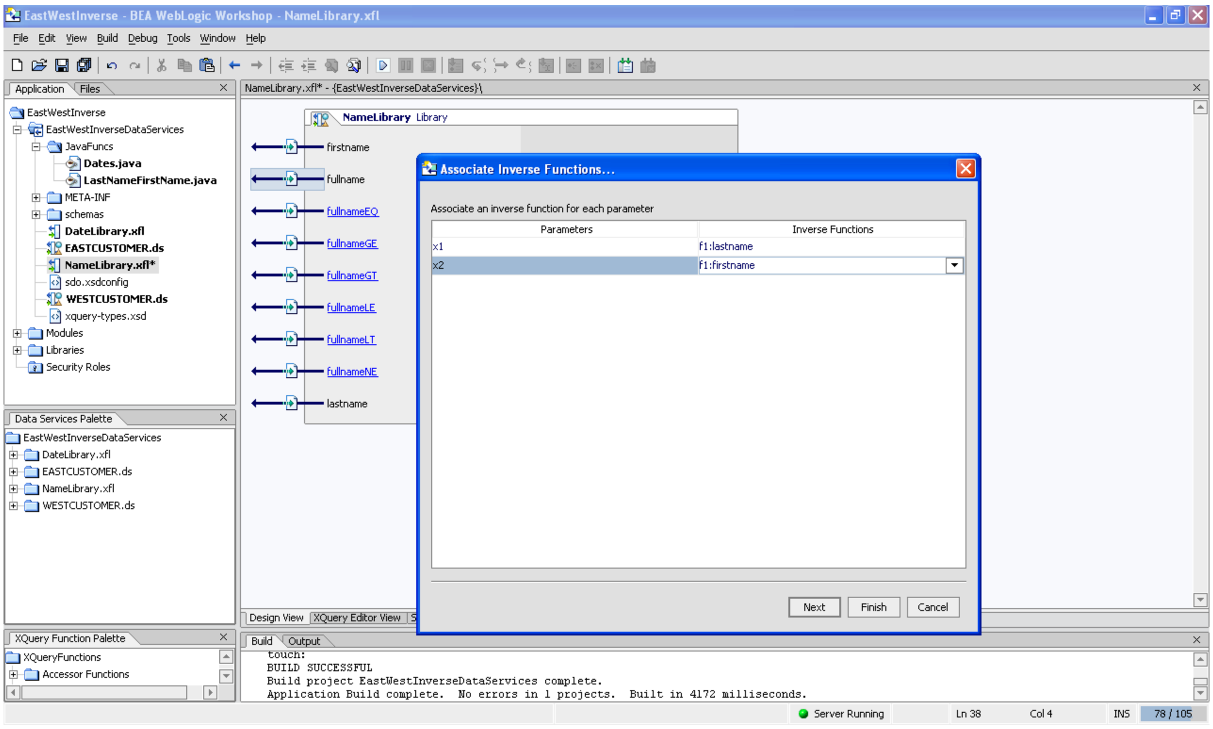
AND (? = t1.FNAME))", delegating most of the query processing work to the underlying relational data source and retrieving only relevant data into the ALDSP runtime. As indicated at the bottom of the query plan, the values bound to these parameters are the result of inverting the incoming function parameter (fullname) and then binding the resulting pair of values (lastname and firstname) as input to the SQL query. The query was arrived at through the use of the equivalent transform information; its predicate was produced by using this information to transform the full name equality predicate in getWestCustomersByName() into the equivalent conjunctive equality predicate on last and first names. We will examine this rewrite process in much more detail in Section 4.

3.4 User Experience for Inverse Functions

Having examined these use cases and their corresponding source files, readers will probably feel that having to hand-author the components of an <inverseFunctions> annotation would be tedious and error-prone. For this reason, ALDSP provides graphical UI assistance for defining inverse functions and equivalent transformations. Inverse functions can be declared graphically, as shown in Figure 9, by right-clicking on a function in the design view of an XFL and choosing its inverse(s) from a drop-down list of the other candidate functions in the XFL. In the figure, the fullname function has been selected, and lastname and firstname are being chosen as its first and second position inverses, respectively. Equivalent transforms can also be defined graphically, as shown in the screen shot of Figure 10, where the rules defining the equivalence between the built-in XQuery string comparisons on full name values and the XQuery-bodied last- and first-name based comparison functions are being specified.

3.5 Related Functionality

When we undertook the design and implementation of inverse functions, much to our surprise, we were unable to find directly related work in the database research literature to build upon. Our work on inverse functions is loosely related to work from the object-relational database era on supporting and optimizing queries involving user-defined data types. For example, the UC Berkeley ADT-Ingres project [8, 9] and its follow-on project Postgres [10] provided ways for developers to register functions to convert ADT instances to/from their external (printable) representations and to register comparison-related functions to enable hash- and B+ tree-based indexes to be used on ADT-valued columns. The ALDSP inverse function facility is different, however, in that there is no new data type involved or needed. Instead, the notion of inverse functions in ALDSP is lighter weight, specifying for one particular function of interest—typically a data transformation of some sort—how to invert the function when needed and how to transform the comparisons on its return type into equivalent XQuery functions involving comparisons on its argument types. Updates through a transformation can be enabled in some database systems by writing procedural code inside INSTEAD OF triggers, but ALDSP provides a declarative, higher-level way of achieving the same functionality by registering an inverse. A more in-depth (and recent) Google search for related work again yielded no research papers on inverses (except for a brief mention of their utility in [11]), but it did yield one related system called OpenLink Universal Server. The OpenLink



Associate an inverse function for each parameter

Parameters	Inverse Functions
x1	f1:lastname
x2	f1:firstname

```
touch:
BUILD SUCCESSFUL
Build project EastWestInverseDataServices complete.
Application Build complete.  No errors in 1 projects.  Built in 4172 milliseconds.
```

Figure 9: Declare Inverses Using the UI

system is a SQL-based data integration system that includes support for SQL inverse functions [12] that appear to be similarly motivated and to have capabilities similar to XQuery inverse functions in ALDSP. A difference in power is that OpenLink is restricted to transformations that preserve the original query's comparison operations.

4. HOW INVERSE FUNCTIONS WORK

We now explore how the inverse function facility of ALDSP works inside. We briefly review the structure of the ALDSP query optimizer. We then explain how inverse functions were added, including the nature of the rewrite rules and how they are controlled. We reflect briefly on the power of the resulting facility, and we close with a discussion of the role of inverse functions in update processing.

4.1 Query Optimization in ALDSP

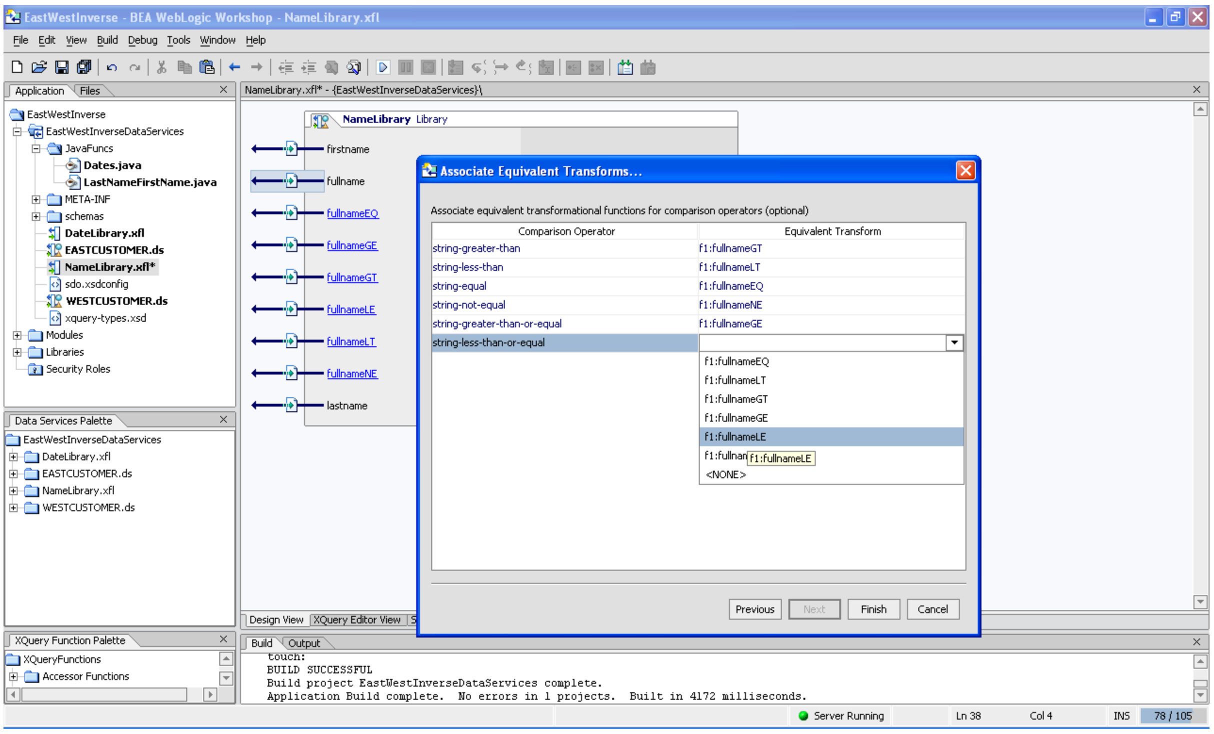
The ALDSP query processor compiles an XQuery into an evaluation plan that undergoes several optimization stages, each one implemented by rewriting rules. One important class of optimizations is related to function inlining and unnesting, and it is closely related to view unfolding in relational query processing. Other targeted key areas are detecting and processing relational-style joins and group-bys. Although it is always possible to perform mid-tier query plan evaluation, for performance reasons the ALDSP query processor tries to identify the largest fragments of the query that can be translated into SQL and then pushed down to relational data sources.

SQL pushdown optimization is implemented in two steps. The first step is implemented by optimizer rules that prepare the XQuery expression tree for SQL pushdown analysis. It is in this step that explicit joins are introduced and optimized and that other subexpressions that may be translatable into SQL are consolidated as well. The second step in SQL pushdown optimization generates SQL code for the portions of the expression tree that operate on data originating from relational sources. Several XQuery constructs and a number of the built-in functions and operators (logical operators, numeric and date-time arithmetic, various string functions, comparison operations, aggregate functions, and the sequence functions subsequence, empty, and exists) can be pushed down successfully as SQL. Certain other expressions can first be partially evaluated in XQuery and then pushed as bound SQL parameters. This latter category includes results from non-pushable function calls, such as Java calls in the absence of inverse functions, as well as operations on nodes or sequences of nodes. A more detailed treatment of ALDSP query optimization and SQL pushdown, as well as a more in-depth coverage of other aspects of the architecture of the ALDSP query processor, can be found in [6].

4.2 Adding Support for Inverse Definitions

Inverse functions were integrated into the ALDSP query processor by extending the optimizer with additional optimization rules that make SQL generation possible for use cases like those in Section 3 involving opaque data transformations in selection (or join) predicates.

What we needed to introduce are in fact equivalent transforms between functional expressions such as lt(y2kdate(x),y) $\equiv$ lt(x,y2kdays(y)). However, termination for an arbitrary set of such equivalences is undecidable, as can be shown by adapting the proof of undecidability for termination of term rewriting [13]. Thus, it is not possible to always explore all equivalent query plans given such rules. Controlling rule evaluation in such a way as to cover a useful range of use



Associate Equivalent Transforms...

Associate equivalent transformational functions for comparison operators (optional)

Comparison Operator	Equivalent Transform
string-greater-than	f1:fullnameGT
string-less-than	f1:fullnameLT
string-equal	f1:fullnameEQ
string-not-equal	f1:fullnameNE
string-greater-than-or-equal	f1:fullnameGE
string-less-than-or-equal	

f1:fullnameEQ
f1:fullnameLT
f1:fullnameGT
f1:fullnameGE
f1:fullnameLE
f1:fullnan f1:fullnameLE
<NONE>

```
touch:
BUILD SUCCESSFUL
Build project EastWestInverseDataServices complete.
Application Build complete.  No errors in 1 projects.  Built in 4172 milliseconds.
```

Figure 10: Declare Function Properties Using the UI

cases while also terminating quickly was therefore another design consideration.

Our implementation allows only directed rewriting *rules* of the form $C(f(x), y) \rightarrow B(x, y)$ or $C(x, f(y)) \rightarrow B(x, y)$. C are binary functions, typically comparison functions or other types of conditions, and f are *invertible* functions, i.e., those functions for which one or more *inverses* have been declared in the associated metadata. (For instance y2kdays from Figure 6 is the inverse of the unary function y2kdate.) B are XQuery expressions in which x and y may appear as free variables. For convenience, the B expressions are defined as XQuery functions, e.g., the XQuery-bodied comparison functions shown in the XQuery libraries of Figures 6-7. The choice of restricting our implementation to these types of rules was motivated by our focus on pushing boolean conditions to SQL that are constructed using comparison operators. The syntax for the rules (encoded in the XML annotations described in Section 3) can express all of these optimizations, and for many practical cases, we are able to bound the search space of rewritings that are of potential interest to us, as we will explain.

Algorithm 1, describes, in pseudo-code, how the query optimizer uses the rewriting rules to produce equivalent plans. In the following discussion, in order to simplify the presentation, we will not distinguish explicitly between an expression and its representation as a syntax tree.

The algorithm consists of a recursive rewriting procedure REWRITE-COND that, when given an expression tree E, tries all applicable rules from the input rule set S.

A rule r of the form $C(f(x), y) \rightarrow B$ (or $C(x, f(y)) \rightarrow B$) matches the node *nd* of E if *nd* is a call to the binary function C and its left (right) child is a call to f. Once a match is found, the rule is executed by calling the recursive

Algorithm 1 REWRITE-COND$(E; S)$

```
{$E$ is an XQuery expression}
{$S$ is a set of rules $\{p(x,y) \rightarrow B(x,y)\}$}
{where $p$ is of the form $C(f(x),y)$ or $C(x,f(y))$}
let $nd$ be the root of the expression tree of $E$
repeat
  rewrite the children of $nd$ recursively, bottom-up
  for all rules $r \in S$ that match $nd$ do
    if the rule has not already been applied to $nd$ then
      $max \leftarrow$ length of longest loop-free path
          starting from LHS pattern of $r$
      $i \leftarrow$ number of invertible functions in $E$
      $bound \leftarrow (max + 1) * i$
      APPLY-RULES$(E, S, r, bound)$
    end if
  end for
  reduce occurrences of $f^{-1}(f(x))$,
until $E$ does not change
```

```
subroutine APPLY-RULES:
  APPLY-RULES$(E; S; r; bound)$
  if $bound > 0$ then
    $F \leftarrow$ apply rule $r$ to E
    add $F$ as a choice to $E$
    for all rules $r'$ matching $E$ and not yet applied to $E$
        APPLY-RULES$(E, S, r', bound$-1$)$
  end if
```

Applying a rule $C(f(x), y) \rightarrow B(x, y)$ involves inlining the bindings for x and y obtained from the match against E inside the B expression and then outputting the instantiated B. The resulting F represents an alternative subplan that could be used instead of the original subplan of E.

In the main loop of REWRITE-COND, we compute a *bound* for the maximum number of rewriting steps (i.e., recursive calls to APPLY-RULES) that are allowed starting from a certain node of the original expression. The bound uses results from a static analysis stage performed during the compilation of the data service, when a *graph of dependencies* between rules is inferred. The nodes in this graph are the patterns $C(f(x), y)$ used when matching the rules, modulo renaming of free variables x and y. There is an edge from $C(f(x), y)$ to $C'(f'(t), v)$ if there is a rule $C(f(x), y) \rightarrow B$ in which a subexpression of B matches $C'(f'(t), v)$. The longest loop-free path in this graph starting from a certain pattern p tells us how many equivalent plans we can explore, starting from the matching expression, before rediscovering a match for p. To continue rewriting after reaching such a point is unlikely to be useful—if the call to f inside p could not be translated into SQL the first time, then it can never be translated. This is due in part to a monotonicity property of our SQL translation process: if a given subexpression cannot be pushed to SQL, then an expression that includes it will not be pushable either. We defer discussion of the coverage of our algorithm to Section 4.3.

After applying the rewriting rules derived from equivalent transforms, the optimizer also *reduces* an application of a function over its inverse, trying to maximize the chances for pushability. For unary functions, the reduction of f^{-1} with f eliminates the two function calls, leaving only the x argument of f. For functions of higher arity, we can only specify projections of their inverses. Suppose that f_k^{-1} is the projection of the inverse of f on the k^{th} component. Then $f_k^{-1}(f(x))$ will be reduced to the k^{th} projection of x. In the use case with a composite name (Figure 5), the inverse's projections of fullname are lastname and firstname, and we can reduce, for instance, lastname(fullname($n1,$n2)) to $n1.

Figure 11 applies this rewriting process to the WestCustomers data service example (Figure 3). The example starts from a call to getWestCustomersByName, which contains an equality condition on the fullName element of the variable $WestCustomerView. When inlined, the condition involves a call to the external function fullname, which precludes the direct translation of the query into SQL. However, fullname has inverse-related metadata associated with it (Figure 7), declaring its inverses to be lastname and firstname, together with several equivalent transforms, including one saying that an equality on fullnames can be replaced by the expression inside the fullnameEQ function body. Thus,

```
$fullname eq ns3:fullName($WESCUSTOMER/LNAME,
                          $WESTCUSTOMER/FNAME)
```

can be equivalently rewritten into:

```
f1:lastname($name) eq $WESTCUSTOMER/LNAME and
f1:firstname($name) eq $WESTCUSTOMER/FNAME
```

This can then be recognized (by the SQL generator) as equivalent to a selection over the WESTCUSTOMER table where the SQL '?' arguments are the results of the function calls f1:lastname($fullname) and f1:firstname($fullname), respectively. These function calls will be performed only once for the entire data service call, and the SQL query sent to the database engine will apply the condition efficiently.

At the end of the phase of optimization using inverse function rewritings, a query plan will actually contain sets of

```
declare function ns1:getWestCustomersByName
       ($fullname as xs:string)
     as element(ns0:WestCustomerView)*
{
  for $WestCustomerView in ns1:getWestCustomers()
    where $fullname = $WestCustomerView/fullName
  return
    $WestCustomerView
};
```

$\Downarrow$ *inline the call to getWestCustomers (from Figure 3)*

```
  for $WESTCUSTOMER in ns2:WESTCUSTOMER()
    where $fullname eq {ns3:fullname
    ($WESTCUSTOMER/LNAME, $WESTCUSTOMER/FNAME)}
  return
  <ns0:WestCustomerView>
    ...
  </ns0:WestCustomerView>
```

$\Downarrow$ *apply rule for fullnameEQ (from Figure 7)*

```
  for $WESTCUSTOMER in ns2:WESTCUSTOMER()
    where
    f1:lastname($fullname) eq $WESTCUSTOMER/LNAME and
    f1:firstname($fullname) eq $WESTCUSTOMER/FNAME
  return
    ...
```

$\Downarrow$ *generated SQL*

```
    SELECT t1.''CUSTID'' AS c1,
           t1.''FNAME'' AS c2,
           t1.''LNAME'' AS c4,
           t1.''HIRED'' AS c3,
           t1.''MONTHLYSAL'' AS c5
    FROM ''DTABASS''.''WESTCUSTOMER'' t1
    WHERE ((? = t1.''LNAME'') AND (? = t1.''FNAME''))
```

Figure 11: Pushing Selections on Names

equivalent subplans for each condition on which the inverse rewriting rules fired. In the current implementation, the optimizer picks from each set the first subplan that can be translated into SQL, if any. It would also be possible, in principle, to perform cost-based optimization and choose the alternative with the most promising cost, if the optimizer had function cost metadata available. (In general, selecting a query plan that uses inverse functions may be beneficial not only for data integration, but also for applications where materializing the result of a function is more expensive then querying the source data. This can occur, for example, in data compression or graphical processing [14].)

4.3 Coverage and Limitations

We now consider the limits of the ALDSP inverse rewriting approach. Suppose now that GotJobs is expanding further, and it opens a branch called EastEurope in a Slavic country that uses the cyrillic alphabet. Further, suppose that users from that branch should be able to see the names of the West Coast customers rewritten in cyrillic characters, possibly with additional formatting, as provided through a view called EastEuropeView. This view draws its information from the WestCustomers data service by applying an additional unary function cyrillicName($fullname), so after inlining all XQuery function calls, a selection on cyrillic names would be rewritten into

```
$name = ns5:cyrillicName(ns3:fullName
   ($CUSTOMER/LNAME, $CUSTOMER/FNAME))
```

To enable the rewrite optimizations needed for efficient querying in this use case, the Slavic data service developer will register a function latinName($cyrillicname) as the inverse of cyrillicName, as well as the equivalent transformation:

$$\mathsf{cyrillicName}(\$engName) = \$name \rightarrow \\ \$engName = \mathsf{latinName}(\$name)$$

In the graph of rule dependencies, the longest path starting from the pattern $x = \mathsf{cyrillicName}(y)$ has length 2, and the ALDSP rewriting algorithm will finally rewrite the selection condition above into

```
f1:lastname(ns5:latinName($name)) eq $CUSTOMER/LNAME and
f1:firstname(ns5:latinName($name)) eq $CUSTOMER/FNAME
```

which again can be pushed to the SQL engine by precomputing and binding the query parameters. Thus, the algorithm effectively handles multiple layers of inverse function usage.

In general, what we are looking for when controlling rule rewriting is to find all *useful* rewritings for our purposes, meaning all equivalent plans that can be pushed to SQL. While our approach handles the use cases we have seen from ALDSP customers, the general answer, if we relax our rule format restrictions, is negative—the current approach has its limitations. For example, consider a view created for display purposes in which customer names are processed by calls to the function displayedName($lname,$fname). Assume the existence of an invertible function norm that transforms last names into a standard format and a generalized equivalent transform rule stating that the combined effect of these functions is similar to applying fullName directly:

$$\mathsf{displayedName}(\mathsf{norm}(\$ln), \$fn) \rightarrow \mathsf{fullName}(\$ln, \$fn)$$

The ALDSP inverse rewriting algorithm, on the input

```
$name = displayedName(norm($CUSTOMER/LNAME),
                      $CUSTOMER/FNAME       )
```

would compute a bound of maximum 1 rewriting step to apply because there is no edge in the graph of dependencies starting from $\mathsf{displayedName}(\mathsf{norm}(x), y)$ and there is only one call to an invertible function, norm. Thus, it would stop after rewriting the expression into an equality involving a call to fullName, without performing a second (desirable) rewriting step to invert that call into a comparison on last and first names. Intuitively, the reason for not discovering all useful plans is that the rewriting rules provide only a limited context, and an analysis that only looks at their patterns may miss certain combinations of expressions. However, as mentioned in Section 4.2, allowing rules with arbitrary patterns would not be practical because most important problems become undecidable.

Our algorithm will find all useful rewritings for sets of rules of the form $C_1(f(x), y) \rightarrow C_2(x, f^{-1}(y))$, where C_1,C_2 are comparison functions and f an invertible function, under the restriction that the sets of comparisons and invertible functions are disjoint and the y expressions do not contain calls to invertible functions. In the more general case, ALDSP takes a best effort approach without promising to exploit all rewriting opportunities (due to the way that the rewriting process is bounded).

4.4 Updates Through Inverse Functions

While we have focused heavily on condition rewriting and SQL predicate pushdown, inverse functions also play a crucial role in enabling ALDSP's update automation to work in the presence of opaque, user-defined transformations. As described more fully in [6], the ALDSP client APIs allow applications to invoke a data service, operate on the results, and then put the changed data back. If the data service being updated has components whose definitions involve custom Java transformations, the ALDSP update processor needs inverse function information in order to convert the incoming data into a form that can be written back to the underlying physical data sources. In this case, the equivalent transformations are not relevant; only the inverse definitions themselves are needed for successful update translation.

In order to perform an update against data obtained from a data service, its data source lineage must be computed. ALDSP statically computes the data lineage for a data service by analyzing the query body of the data service function that has been marked for use in update analysis by the designer of the data service. (One is chosen by default if no function has been specially marked.) The data lineage computation is performed by a specialized rule set that is driven by the same rule engine used for the ALDSP XQuery optimizer. Element/attribute constructors, primary key information, query predicates, and query result shapes are used together in a repeated function inlining process to determine how to propagate changes back to the underlying sources. Inverse function definitions allow this process to proceed successfully through Java functions where it would otherwise have been blocked and thus unable to determine data lineage.

5. EXPERIMENTAL EVALUATION

To quantify the benefits of having inverse function support in ALDSP, this section of the paper presents some simple but realistic experimental results based on the getWestCustomersByName(fullname) function introduced earlier. The underlying relational data for these experiments is the WESTCUSTOMER table, but with one extension–a CHAR(500) field named ETCETERA was added to the table in order to pad the tuples out to a more typical row size. The WestCustomers data service was correspondingly extended with one additional xs:string element, otherInfo, based on the data coming from this column. The WESTCUSTOMER table was populated by generating a collection of synthetic data with unique values in all columns for simplicity, with table sizes ranging from 1,000 to 100,000 rows. The experiments were run on a Dell XPS M1210 laptop machine with a 1.33 GHz Intel Core 2 Duo Processor T7600, 2GB of SDRAM, and a 160 GB SATA hard drive running Windows XP. The relational database server used for the experiments was Pointbase 4.4, a Java-based RDBMS that ships with the BEA WebLogic Platform 8.1 system.

Table 1 shows the structure of the experiments together with the experimental results. For each table size, the read function getWestCustomersByName was invoked with a full name string ("Smith1000, John1000") that ultimately resulted in the return of a single instance of the WestCustomers data service XML element type. This was done three ways for each table size–once without the use of inverse functions, once with inverse functions but with no useful indexes on the underlying WESTCUSTOMER table, and once with a composite non-unique index on (LNAME, FNAME) defined on WESTCUSTOMER. The first way, ALDSP is unable to push a WHERE clause to the RDBMS, so all of the WESTCUSTOMER data must be retrieved and the name-

Inverses	Indexed	100K customers	10K customers	1K customers	compile time
No	No	14400 msec	1500 msec	125 msec	125 msec
Yes	No	2600 msec	250 msec	15 msec	135 msec
Yes	Yes	8 msec	7 msec	5 msec	135 msec

Table 1: Execution times for getWestCustomersByName (yielding 1 match)

matching predicate must be evaluated in the ALDSP mid-tier engine. The second way, ALDSP is able to push the predicate "WHERE ((? = t1.LNAME) AND (? = t1.FNAME))", but a table scan is required to evaluate the predicate in PointBase since there is no supporting index. The third way, ALDSP is able to push the predicate, and PointBase is able to use a simple index scan to retrieve the requested data. The numbers shown in the table are *warm* times (based on repeated query executions).

The results in Table 1 indicate the trends that one would expect. First, the query compilation time is quite small, and adding inverse function rules adds only a very small overhead to the query compilation time. (Remember also that ALDSP uses a query plan cache to avoid reoptimizing queries, as discussed in Section 2.) Second, comparing the query execution times in rows 1 and 2 of the table, the impact of the predicate pushdown optimizations enabled by inverse functions is clear. Even when PointBase must perform a table scan, there is a large benefit (roughly 6x in this case) to not having to fetch unwanted data into ALDSP for mid-tier predicate evaluation. Finally, comparing the query execution times in rows 1 and 3 of Table 1, it is evident that inverse functions plus an appropriate underlying physical database design can enable many orders of magnitude of improvement. For example, with 100,000 rows in the WESTCUSTOMER table, a 14.4 second query becomes an 8 millisecond query. These results clearly emphasize the importance of having support for inverse functions and their associated optimization rules in a data integration system like ALDSP that supports opaque, user-defined data transformation functions.

6. CONCLUSIONS

In this paper, we have provided an in-depth description of the inverse function feature of BEA's AquaLogic Data Services Platform. Since ALDSP is aimed at integrating and service-enabling data from a variety of data sources, data transformations are a key part of the definition of most data services. To handle situations where XQuery is insufficient for performing the required transformations, ALDSP permits users to write and register their own custom transformations, implemented in Java. This then results in a set of transformation functions that are opaque to ALDSP's query optimization and update analysis components.

To break down the barriers imposed by user-defined Java transformation functions, ALDSP allows data transformation developers to specify inverse functions and equivalent transformations that enable ALDSP to push predicates and perform updates on data services that involve the use of such functions. We have described here the motivation for inverse functions, illustrated their use for addressing typical 1:1 and 1:N data transformation use cases, and discussed their implementation in some depth. We also presented a brief set of experimental results to demonstrate, in a practical way, how the presence of this feature in ALDSP can provide orders of magnitude improvements in query execution times when such transformations are involved.

7. REFERENCES

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures, and Applications*. Springer-Verlag, Berlin/Heidelberg, 2004.

[2] M. Huhns and M. Singh. Service-oriented computing: Key concepts and principles. *IEEE Internet Computing*, 1(9):75–81, 2005.

[3] M. Carey. Data services: This is your data on SOA. *Business Integration Journal*, Nov/Dec 2005.

[4] V. Borkar, M. Carey, N. Mangtani, D. McKinney, R. Patel, and S. Thatte. XML data services. *International Journal of Web Services Research*, 1(3):85–95, 2006.

[5] Michael J. Carey the AquaLogic Data Services Platform Team. Data delivery in a service-oriented world: the BEA AquaLogic Data Services Platform. In *SIGMOD Conference*, pages 695–705, 2006.

[6] Vinayak R. Borkar, Michael J. Carey, Dmitry Lychagin, Till Westmann, Daniel Engovatov, and Nicola Onose. Query processing in the AquaLogic Data Services Platform. In *VLDB*, pages 1037–1048, 2006.

[7] K. Williams and B. Daniel. An introduction to Service Data Objects. *Java Developer's Journal*, October 2004.

[8] James Ong, Dennis Fogg, and Michael Stonebraker. Implementation of data abstraction in the relational database system Ingres. *SIGMOD Record*, 14(1):1–14, 1984.

[9] Michael Stonebraker. Inclusion of new types in relational data base systems. In *ICDE*, pages 262–269, 1986.

[10] Michael Stonebraker and Lawrence A. Rowe. The design of Postgres. In *SIGMOD Conference*, pages 340–355, 1986.

[11] Kai-Uwe Sattler, Stefan Conrad, and Gunter Saake. Adding conflict resolution features to a query language for database federations. In *EFIS*, pages 41–52, 2000.

[12] SQL inverse functions. http://docs.openlinksw.com/virtuoso/sqlinverse.html. Part of the Openlink Virtuoso platform.

[13] Hans Zantema. Total termination of term rewriting is undecidable. *J. Symb. Comput.*, 20(1):43–60, 1995.

[14] Allison Woodruff and Michael Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *ICDE*, pages 91–102, 1997.