Query Processing in the AquaLogic Data Services Platform

Vinayak Borkar, Michael Carey, Dmitry Lychagin,
Till Westmann, Daniel Engovatov, and Nicola Onose*
BEA Systems, Inc.
San Jose, CA, USA

ABSTRACT

BEA recently introduced a new middleware product called the AquaLogic Data Services Platform (ALDSP). The purpose of ALDSP is to make it easy to design, develop, deploy, and maintain a *data services* layer in the world of service-oriented architecture (SOA). ALDSP provides a declarative foundation for building SOA applications and services that need to access and compose information from a range of enterprise data sources; this foundation is based on XML, XML Schema, and XQuery. This paper focuses on query processing in ALDSP, describing its overall query processing architecture, its query compiler and runtime system, its distributed query processing techniques, the translation of XQuery plan fragments into SQL when relational data sources are involved, and the production of lineage information to support updates. Several XQuery extensions that were added in support of requirements related to data services are also covered.

1. INTRODUCTION

When relational database management systems were introduced in the 1970's, database researchers set out to create a productive new world in which developers of data-centric applications could work much more efficiently than ever before. Instead of writing and then maintaining lengthy procedural programs to access and manipulate application data, developers would now be able to write simple declarative queries to accomplish the same tasks. Physical schemas were hidden by the logical model (tables), so developers could spend much less time worrying about performance issues and changes in the physical schema would no longer require corresponding changes in application programs. Higher-level views could be created to further simplify the lives of developers who did not need to know about all the details of the stored data, and views could be used with confidence because view rewriting techniques were developed to insure that queries over views were every bit as performant as queries over base data. This data management revolution was a roaring success, resulting in relational database offerings from a number of software companies. Almost every enterprise application developed in the past 15-20 years has used a relational database for its persistence, and large enterprises today run major aspects of their operations using relationally-based packaged applications like SAP, Oracle Financials, PeopleSoft, Siebel, Clarify, and SalesForce.com.

Due to the success of the relational revolution, combined with the widespread adoption of packaged applications, developers of data-centric enterprise applications face a new 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). Any given enterprise is likely to find 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 under the control of packaged or homegrown applications that add meaning to the stored data by enforcing the business rules and controlling the logic of the "business objects" of the application. Meaningful access must come through the "front door", by calling the functions of the application APIs. As a result, enterprise application developers face a huge 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. New, "composite" applications need to be created from these parts – which seems to imply a return to procedural programming.

Composite application development, once called megaprogramming [1], is the aim of the enterprise IT trend known as service-oriented architecture (SOA) [2]. XML-based Web services [3] are a piece of the puzzle, providing a degree of physical normalization for intra- and inter-enterprise function invocation and information exchange. Web service orchestration or coordination languages [4] are another piece of the puzzle on the process side, but are still procedural by nature. In order to provide proper support for the data side of composite application development, we need more – we need a declarative way to create *data services* [5] for composite applications.

The approach that we are taking at BEA is to ride the XML wave created by Web services and associated XML standards for enterprise application development. We are exploiting the W3C XML, XML Schema, and XQuery Recommendations to provide a standards-based foundation for declarative data services development [6]. The BEA AquaLogic Data Services Platform (ALDSP), newly introduced in mid-2005, supports a declarative approach to designing and developing data services [7]. ALDSP is targeting developers of composite applications that need to access and com-

*Work done while visiting BEA from the Computer Science Department at the University of California, San Diego.

VLDB '06, September 12-15, 2006, Seoul, Korea.

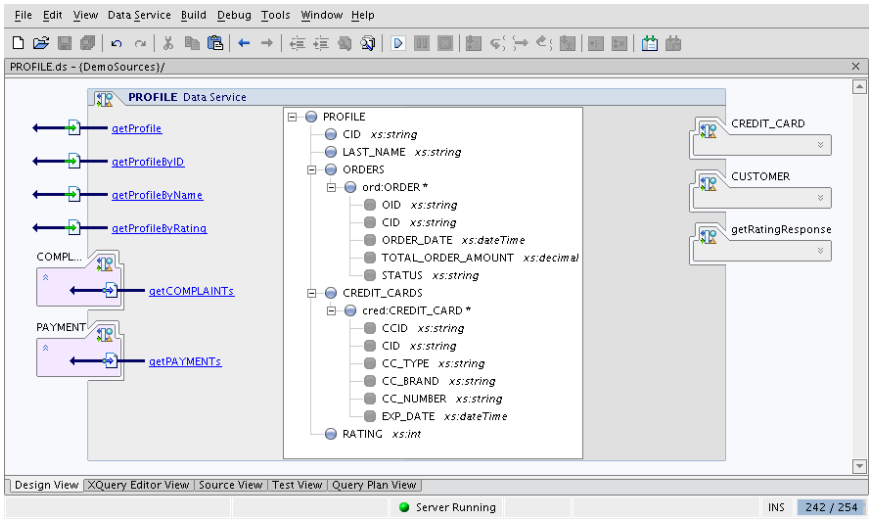

Figure 1: ALDSP Data Service – Design View

pose information from a range of enterprise data sources, including packaged applications, relational databases, Web services, and files, as well as other sources. In this paper, we examine the query processing capabilities of ALDSP in detail, explaining the XML query processing problems that arise in the context of data services and the approach taken to address these problems in ALDSP. We also briefly touch on several related aspects of ALDSP, including caching, security, and update handling.

The remainder of this paper is organized as follows: Section 2 provides an overview of ALDSP, covering its user model, APIs, and system architecture. Section 3 begins the exploration of query processing in ALDSP, describing ALDSP's use and extensions of XQuery, the nature and role of metadata in ALDSP, and the steps involved in query processing; an example is presented that runs throughout the rest of the paper. Section 4 drills down further into query compilation, covering error handling, view optimization, SQL generation for accessing relational data sources, and support for inverse functions. Section 5 focuses on query execution in ALDSP, covering its XML data representation, query plans and operators, and data source adaptors. Also covered in Section 5 is the ALDSP runtime support for asynchronous execution, caching, and handling of slow or unavailable data sources. Section 6 explains how the ALDSP query processing framework supports update automation. Section 7 explores data security in ALDSP, focusing on its interplay with the system's query processing framework. Section 8 provides a brief discussion of related work that influenced ALDSP as well as other related commercial systems. Section 9 concludes the paper.

2. ALDSP OVERVIEW

To set the stage for our treatment of query processing in ALDSP, it is important to first understand the ALDSP world model and overall system architecture. We cover each of these in turn in this section.

2.1 Modeling Data and Services

Since it targets the SOA world, ALDSP takes 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* [6]. 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 is a description of 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, which are 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, which are 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, which are service calls for traversing relationships from one 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 becomes 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. 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. In this example, which will be discussed further later on in the paper, the data service shown was created by declaratively composing calls to several lower-level (in this case physical) data services.

When pointed at an enterprise data source by a 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. Applying introspection to 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

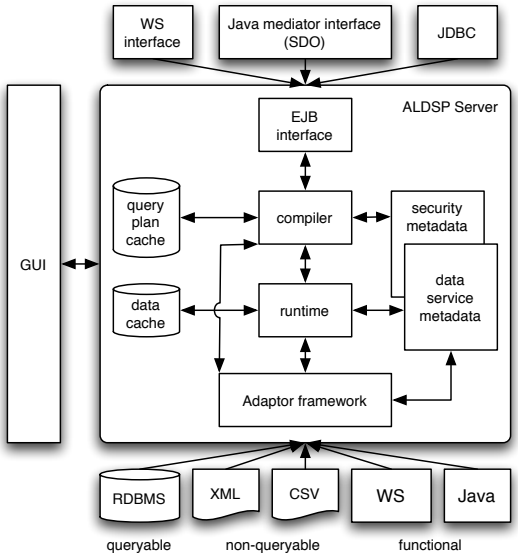

Figure 2: Overview of ALDSP Architecture

XML-ification of a row of the table. In the presence of foreign key constraints, introspection also produces navigation functions (i.e., relationships) 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 function input and output types correspond to the schema information in the WSDL. Other functional data sources are modeled similarly. The result is a uniform, "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 System Architecture

Figure 2 depicts the overall architecture of ALDSP. At the bottom of the picture lie the various kinds of enterprise data sources that ALDSP supports. Data source types are grouped into three categories – queryable sources, non-queryable sources, and functional sources. 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 (a.k.a. comma-separated value) 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 here. Also, it is not uncommon 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). For this reason, ALDSP focuses on integrating information from functional data sources as well as from queryable sources and non-functional sources. 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 is responsible for connecting 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 fairly traditional (as viewed from 35,000 feet) query processing architecture 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. Finally, in addition to the aforementioned query plan cache, ALDSP maintains an optional data cache that can be used to cache data service call results for data services where a performance/currency tradeoff can be made and is desirable.

The main box in Figure 2 is the ALDSP server. The server consists 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 [8], 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. Currently, the ALDSP server's client APIs are all stateless in order to promote server stability; thus, service call and/or query results are completely materialized in the ALDSP server's memory before being returned (despite the fact that the runtime system is capable of pipelining the processing of queries and their results). ALDSP also has server-side APIs to allow applications in the same JVM to consume the results of a data service call or query incrementally, as a stream, or to redirect them to a file, without materializing them first. Finally, 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. QUERY PROCESSING IN ALDSP

We are now ready to begin our examination of ALDSP's query processing architecture. We will start by looking at certain salient aspects of our XQuery language support in ALDSP. We will then discuss the role of metadata in query processing and the phases involved in query processing in ALDSP. This section will close by highlighting ALDSP's use of XQuery for defining data services through a small but illustrative example.

3.1 XQuery Support

The language used to define data services in ALDSP is XQuery. The current release of the ALDSP product, version 2.1, supports the version of the XQuery 1.0 Working Draft published in July 2004 [9]. An ALDSP data service is defined using a data service file that contains XQuery definitions for all of the functions associated with

the data service and that refers to one or more XML Schema files that define the data service's shape and other relevant XML data types. Because of its data-oriented nature, ALDSP relies heavily on the typed side of XQuery. A great deal of type information is statically obtainable from data source metadata, and most ALDSP users think in terms of typed data and enjoy the early error detection/correction benefits of static typing. Most ALDSP users also appreciate features in the ALDSP XQuery editor, like path completion, that depend on the presence of static type information. The approach taken to static typing in ALDSP varies somewhat from the pessimistic approach taken in the current XQuery specification. Finally, ALDSP has added several extensions to XQuery that users find extremely convenient for common use cases. Let us now delve further into each of these aspects of XQuery in ALDSP.

Static typing as described in the XQuery specification is *name-based*. Types are always specified using XML Schema and identified by name. The notation element(E) denotes the type of an element named E with content type ANYTYPE, while the notation schema-element(E) denotes the type of an element named E that must exist in the current XQuery context (and it is an error if the type is not found there). In the XQuery specification, when a query uses an element constructor to create a new named XML element, the static content type of the new element is ANYTYPE (until it is explicitly validated).

In applying XQuery to our data-centric use cases, we have found that a black-and-white, must-explicitly-validate approach to data typing is not what our users want in terms of the resulting XQuery experience. To address users' wants, the ALDSP XQuery compiler treats element(E) differently: it applies *structural typing* when analyzing XQuery queries. Unlike the specification, the use of an element constructor in ALDSP does not result in the element's contained data being effectively reverted to an unvalidated state. Instead, during static analysis, ALDSP computes the type of the element named E to be the element type with name E and with a content type that is the structural type of the contained content. (The type annotation on the element itself at runtime will still be element(E, ANYTYPE), per the XQuery specification, but the runtime type annotations on its content will survive construction.) This eliminates the need for validation in cases where all of the data flowing into the system is typed, which is the norm in ALDSP. Moreover, a key feature of ALDSP is its support for views (layers of XQuery functions). View unfolding is needed (as in relational database systems) to enable views to be efficient. Structural typing plays an important role in enabling view unfolding, as structural typing means that the type of an expression does not change when an element is constructed around the expression and then subsequently eliminated by child navigation.

One syntactic extension that ALDSP has made to XQuery is the addition of a grouping syntax. For data-centric use cases, grouping (often combined with aggregation) is a common operation. Our experience has been that most users find the standard XQuery approach to grouping unfriendly. It is also hard for a query processor to discern the user's intentions in all cases in the absence of explicit group-by support. For these reasons, ALDSP extends the XQuery FLWOR expression syntax with a group-by clause (yielding a FLWGOR syntax, where the "G" is silent):

group (var1 **as** var2)? **by** expr (**as** var3)? (**,** expr (**as** var4)?)*

The result of grouping after the group-by clause's point in a FLWGOR is the creation of a binding tuple containing var2, var3, var4, and so on, with the input tuple stream being grouped by the grouping expression(s). In the resulting stream, each var2 binding corresponds to the sequence of var1 values that were associated with identical grouping expression values in the input. For example, the following ALDSP grouping query computes and returns elements that associate each existing customer last name with the set of customer ids of customers with that name:

```
for $c in CUSTOMER()
let $cid := $c/CID
group $cid as $ids by $c/LAST_NAME as $name
return <CUSTOMER_IDS name="{$name}">{
  $ids
}</CUSTOMER_IDS>
```

Another ALDSP extension to XQuery, which is quite small but *extremely* useful for most ALDSP use cases, is the addition of a syntax for the optional construction of elements and attributes. By its very nature, XML is all about handling "ragged" data – data where XML fragments can have various missing elements and/or attributes. XQuery handles such data well on the input side, but has no convenient query syntax for renaming and transmitting such data to the output after element or attribute construction. ALDSP offers a conditional construction syntax ("?"), applicable to both elements and attributes, that solves this problem. For example, the "?"-expression `<FIRST_NAME?>{$fname}</FIRST_NAME>` is equivalent to:

```
if (exists($fname))
   <FIRST_NAME>{$fname}</FIRST_NAME>
   else ()
```

The result element `<FIRST_NAME>` is constructed iff the first name value bound to the variable fname is non-empty, accommodating this (common) possibility.

3.2 Data Source Metadata

As described earlier, ALDSP introspects data source metadata in order to generate an XQuery-based model of the enterprise in the form of physical data services. The pertinent metadata is captured and the pragma facility in XQuery is used to annotate system-provided, externally-defined XQuery functions with the information that the compiler and runtime need to implement these functions. As described, backend data source accesses are modeled as XQuery functions with typed signatures. For relational databases, each table or view is surfaced as a function in the corresponding ALDSP physical data service; primary and foreign key information is captured in the pragma annotations, as is the RDBMS vendor, version, and connection name. In the case of Web services, the location of the WSDL is captured in the corresponding function annotations. More detail on ALDSP's metadata capture approach can be found in [10]. Once captured, the source metadata is used by the ALDSP compiler, graphical UI, query optimizer, and runtime. The compiler, for example, uses the captured native type and key information to optimize query plans. The runtime relational database adaptors use the connection names to connect to the associated backend database systems.

3.3 Query Processing Phases

Query processing in ALDSP involves the usual series of query processing stages:

1. Parsing: recognize and validate query syntax.
2. Expression tree construction: translate query into internal form for further analysis.
3. Normalization: make all query operations explicit.
4. Type checking: perform type checking and inference.

5. Optimization: optimize query based on source info.

6. Code generation: create query plan that can be cached and executed.

7. Query execution: execute plan and return results.

The first four stages comprise the analysis phase of compilation, at the end of which the compiler has determined that the query is both syntactically and semantically valid and has filled in the details of any implicit operations (such as atomization) implied by the query. The "secret sauce" occurs in the next stage, optimization, where the plan for executing the query is determined based on the query, the data sources accessed, and the metadata known about those sources. The code generation stage turns the plan into a data structure that can be interpreted efficiently at runtime. Finally, once compiled, the generated query plan can be executed (repeatedly, possibly with different parameter bindings each time). In the remainder of the paper we provide more details on each of these phases as well as the overall query processing process. Before doing so, however, let us wrap up this overview with an example that can be used to tie together the discussions later in the paper.

3.4 Running Example

An example of a data service was given in Figure 1. Figure 3 shows the XQuery source for this logical data service, showing how the first read function of the data service might look when ALDSP is being used to integrate information from two relational databases and a Web service.

It is assumed in the figure that one relational database contains the CUSTOMER and ORDER tables, while a different database contains the CREDIT_CARD table. In addition to these three tabular data sources, a document-style Web service provides a credit rating lookup capability that takes a customer's name and social security number and returns his or her credit rating. The read method, getProfile(), takes no arguments and returns customer profiles for all customers listed in the CUSTOMER table in the first relational database. Note that the contents of the tables are accessed via XQuery functions that are called in the body of the query. For each customer, the ORDER table is accessed to create the nested <ORDERS> information; here the access is via a navigation function that ALDSP has automatically created for this purpose based on key/foreign key metadata. Also for each customer, the CREDIT_CARD table in the other database is accessed, and then the credit rating Web service is called. The result of this function is a series of XML elements, one per CUSTOMER, that integrates all of this information from the different data sources. Finally, once the integration and correlation of data has been specified in this function, which is the main (or "get all instances") function for this data service, each of the remaining read methods of the data service becomes trivial to specify, as is also demonstrated in Figure 3.

```
xquery version "1.0" encoding "UTF8";

(::pragma ... ::)
declare namespace tns=...
import schema namespace ns0=...
declare namespace...

(::pragma function ... kind="read" ...::)
declare function
  tns:getProfile() as element(ns0:PROFILE)* {
for $CUSTOMER in ns3:CUSTOMER()
return
<tns:PROFILE>
  <CID>{fn:data($CUSTOMER/CID)}</CID>
  <LAST_NAME>{
    fn:data($CUSTOMER/LAST_NAME)
  }</LAST_NAME>
  <ORDERS>{
    ns3:getORDER($CUSTOMER)
  } </ORDERS>
  <CREDIT_CARDS>{
    ns2:CREDIT_CARD()[CID eq $CUSTOMER/CID]
  }</CREDIT_CARDS>
  <RATING>{
    fn:data(ns4:getRating(
      <ns5:getRating>
        <ns5:lName>{
          data($CUSTOMER/LAST_NAME)
        }</ns5:lName>
        <ns5:ssn>{
          data($CUSTOMER/SSN)
        }</ns5:ssn>
    </ns5:getRating>)/ns5:getRatingResult)
  }</RATING>
</tns:PROFILE>
};

(::pragma function ... kind="read" ...::)
declare function
  tns:getProfileByID($id as xs:string)
  as element(ns0:PROFILE)* {
  tns:getProfile()[CID eq $id]
};

...

(::pragma function ... kind="navigate" ...::)
declare function
  tns:getCOMPLAINTs($arg as element(ns0:PROFILE))
  as element(ns8:COMPLAINT)* {
  ns8:COMPLAINT()[CID eq $arg/CID]
};

...
```

Figure 3: ALDSP Logical Data Service – XQuery Source

4. QUERY COMPILATION

We now examine the salient aspects of the ALDSP XQuery compilation process. In addition, we discuss how recently added support for inverse functions in ALDSP 2.1 enables optimizations and updates that would otherwise be blocked by commonly occurring integration-oriented data value transformations.

4.1 Analysis

The first major phase in query compilation is the analysis phase. Most of what happens in this phase simply implements behaviors prescribed by the XQuery specification, so we will cover this phase briefly, focusing only on its novel aspects – which have to do with error recovery and type-checking.

The obvious use of the XQuery compiler occurs at runtime, when each new query is submitted and needs to be compiled and executed. However, the ALDSP product includes a rich graphical XQuery editor, and the XQuery compiler helps to support this graphical tool. The ALDSP graphical XQuery editor is interesting in that its model – i.e., the query representation that it operates upon as the editing actions occur – is not just any supporting data structure, but *the XQuery itself*. A major benefit of this approach (and the reason for doing it) is that it ensures a robust two-way editing experience together with ALDSP's XQuery source editor. However, this architecture also poses challenges, in that the graphical XQuery editor relies heavily on the query compiler. This induces requirements in the areas of error handling and type-checking during the analysis phase of compilation. This phase actually has two modes – its policy is to fail on first error when invoked for query compilation on

the server at runtime, but to recover as gracefully as possible when being used by the XQuery editor at data service design time. A data service is a (potentially large) collection of XQuery functions, so the required design time behavior is to locate as many errors as possible by attempting to analyze as many of the data service's functions as possible in the course of a single compilation.

The ALDSP XQuery compiler can recover and continue after a number of possible errors, including syntax errors in XQuery prolog declarations (including function declarations). On encountering a parsing error, it attempts to skip to the end of the declaration (the first ";" token) and continue from there. If a function body is in error, but the signature is error-free, the signature is retained and still available for use in checking other uses of the function. During the expression tree construction phase, the query compiler can encounter a variety of errors - e.g., references to non-existent items, duplicate items, or schema import errors. To recover, the analyzer substitutes a special error expression that has the same input expressions for the offending expression. Errors found during normalization are handled similarly. Finally, errors that are detected during the type-checking stage lead type inference to assign an error type to the offending expression. In each stage of analysis, errors are reported, and only error-free functions are retained as candidates for further stages of analysis.

In terms of type-checking, the ALDSP XQuery compiler does structural typing, as mentioned earlier. In addition, it takes a more optimistic approach to typing than that prescribed in the XQuery specification. XQuery says that f($x) is valid iff the static type of $x is a subtype of the static type of the parameter of f(). This is very restrictive, and would force the use of type coercion expressions to eliminate compile-time type errors. ALDSP modifies the static type-checking rule to say that f($x) is valid iff the static type of $x has a non-empty intersection with the static type of the parameter of f(). Additionally, we introduce a typematch operator to enforce the correct XQuery semantics at runtime. If it can be shown at compile-time that $x is indeed a subtype of the static type of f's parameter, the typematch operator is not introduced.

4.2 General Optimizations

One of the most important features of ALDSP is that one can create data services that service-enable information composed from multiple data sources – data services that can then be *efficiently reused* in queries or in other, more application-specific data services. As an example, the running example in Figure 3 shows how a data service designer might author one "get all" function to encapsulate the details of the required data composition/integration logic and then reuse that function elsewhere, even within the same data service. If the reuse adds predicates, or doesn't use a portion of what the underlying function creates, efficiency demands that the predicate be pushed into the underlying function and/or that any unused information not be fetched at all when running the resulting query. To this end, ALDSP performs XQuery view optimizations, involving function inlining and unnesting, that are analogous to view unfolding optimizations in relational query processing. As an example of source access elimination, consider the following XQuery fragment that could easily result from function inlining:

```
let $x := <CUSTOMER>
             <LAST_NAME>{$name}</LAST_NAME>
             <ORDERS>...</ORDERS>
          </CUSTOMER>
return fn:data($x/LAST_NAME)
```

The result produced can be replaced with `$name` without the need to ever construct the ORDERS, making it unnecessary to even fetch the data contributing to ORDERS in order to execute this query.

Because views (layers of data services) are common in ALDSP, it is essential that queries involving views be efficiently compiled and optimized. To ensure that this is the case, views are actually optimized using a special sub-optimizer that generates a partially optimized query plan; this plan can be further optimized in the context of a particular query. This architecture factors the query usage-dependent part of view optimization out, making it possible for the query-independent part to be performed once and then reused when compiling each query that uses the view. Caching and cache eviction is used to bound the memory footprint of cached view plans.

Another important area of query optimization for ALDSP is the handling of group-by operations. Because of the nested nature of XML, and the naturalness (and therefore attractiveness) of hierarchical schemas as information models for data services, queries that compose nested results are extremely common in ALDSP. Figure 3 is a typical example, with order and credit card data nested within each customer. These query patterns mean that ALDSP frequently needs to perform outer-join and group-by operations on keys in order to bring the data to be nested together. To do this efficiently, ALDSP aims to use pre-sorted or pre-clustered group-by implementations when it can, as this enables grouping to be done in a streaming manner with minimal memory utilization. If a join implementation maintains clustering of the branch whose key is being used for grouping, no extra sorting is required. In most ALDSP use cases, a constant-memory group-by can be chosen, without any explicit clustering operation being needed. In the worst case, ALDSP falls back on sorting for grouping, which then can possibly be pushed to the backend (if it is relational).

A third key area of ALDSP query optimization relates to joins and join processing. The ALDSP query compiler is a descendent of the original BEA compiler from a "pure, XQuery-for-XML's-sake" XQuery processing engine [11]. The original compiler was aimed at in-memory XML processing, with a focus on stream processing of large messages, and had no notion of joins or data sources (or, as a result, of having an opportunity to offload processing to relational sources). In contrast, due to its data-centric target use cases, the ALDSP XQuery compiler rewrites expressions into joins where possible in order to prepare for possible subsequent SQL generation. Also, even when joins cannot be pushed to a backend, they can be executed using different implementations that have been studied for 30+ years in the relational database world.

As a multi-source data composition engine, ALDSP must perform distributed joins. Its central distributed join method is a method referred to internally as PP-k (parameter-passing in blocks of k). In order to compute A join B if B comes from a relational source, ALDSP will usually opt to perform a pipelined PP-k join: k tuples are fetched from source A, a request is issued to fetch from B all those tuples that would join with any of the k tuples from A, and then a middleware join is performed between the k tuples from A and the tuples fetched from B. This is repeated in blocks of k tuples from A until source A is exhausted. The request for B tuples takes the form of a parameterized disjunctive SQL query with k parameters (or sets of parameters, in the case of a composite join predicate, plus any other pushable predicates) that are bound using A values and executed once per block. For ALDSP use cases, this method provides an excellent tradeoff between the required middleware join memory footprint (which is small) and the latency imposed by roundtrips to the source B (since it does only 1/k as many roundtrips as it would if it processed A tuples one-by-one). A small value of k means many roundtrips, while large k approximates a full middleware index join; by default, ALDSP uses a medium-sized k value (20) that has been empirically shown to work well.

4.3 SQL Plan Preparation

A major objective of the ALDSP query compilation process is ensuring that good SQL pushdown is possible when one or more of the physical data services in a query are drawn from queryable relational databases. The SQL pushdown problem is handled in two steps. The first step is implemented by optimizer rules that prepare the XQuery expression tree for SQL pushdown analysis. In this first step, as mentioned earlier, join expressions are introduced. In particular, join expressions are introduced for each "for" clause in the query. Next, any where conditions and let clauses are pushed into joins where possible (based on variable dependencies). At this point, the query expression tree has joins and has its other operations pushed down; further SQL preparation then occurs as follows. First, joins that occur inside lets are rewritten as left outer joins and brought out into the outer FLWR expression containing the let clause. Next, ordering clauses are optimized based on presorted prefixes. Finally, clauses are locally reordered based on their acceptability for pushdown to the backends that produce the data required for their execution; this also results in some reordering of adjacent joins to enable more effective SQL pushdown.

4.4 SQL Generation

Once the expression tree has been prepared in this way, the process of actual SQL generation begins for portions of the tree that operate on data that originates from relational sources. In terms of input data, as described earlier, a relational table or view is modeled as an external XQuery function that returns a sequence of elements corresponding to its rows, and the content of each row element is a sequence of column elements that contain typed data according to a well-defined set of SQL to XML data type mappings. NULLs are modeled as missing column elements, so the rows can be "ragged", as is most natural for XML.

Pushdown analysis looks at regions of the XQuery expression tree that involve data that all comes from the same relational database (which it can determine from the metadata associated with the physical data service functions). Things considered to be *pushable* to SQL include clauses of the query compiler's extended FLWOR expressions (e.g., 'for', 'let', 'where', 'order by', 'join', 'group-by'), constant expressions, certain XQuery functions and operators (including logical operators, numeric and date-time arithmetic, various string functions, comparison operations, aggregate functions, and the sequence functions subsequence, empty, and exists), typecast and type constructor functions, if-then-else expressions, a subset of quantified expressions that can be translated into semi- or anti-semi-join operations, and filter expressions with numeric predicates. Other expressions can first be evaluated in the XQuery runtime engine and then pushed as SQL parameters (variables and external variables). Some notable *non-pushable* expressions include XQuery node constructors, complex path expressions, expressions on sequence types (instance of, typeswitch, castable), and explicit validation. (Fortunately, explicit validation is rarely of interest, as relational data is already typed upon entry into the ALDSP world.)

Space precludes an in-depth treatment of what happens during SQL pushdown analysis and SQL generation. However, Tables 1 and 2 show an illustrative subset of the types of pushdowns that come out of this query compilation phase. In each case, we show a small XQuery snippet that accesses one or several tables from a single relational database and we show the SQL that would be generated for the snippet. Actual SQL syntax generation during pushdown is done in a vendor/version-dependent manner, so in cases where the syntax depends on that, we show Oracle SQL in the table. ALDSP supports SQL generation for the common versions of Oracle, DB2, SQL Server, and Sybase. For each one, the SQL pushdown framework knows what functions are pushable (and with what syntax), how outer joins are supported, where subqueries are permitted, and so on. There is also a notion of a "base SQL92 platform" – basically any other relational database – for which ALDSP generates more conservative SQL92 queries. This enables ALDSP to handle most all relational databases, varying in terms of the aggressiveness of SQL pushdown based on its knowledge about the specific relational database source in question.

4.5 Inverse Functions

Data services provide an abstraction so that applications can see data in a clean, canonical, convenient form – quite possibly as opposed to what might be most direct and efficient form for the ALDSP layer to query and update. For example, suppose that we wished to extend the customer information captured in the integrated customer profile example of Figure 3 to include how long each customer has been one of our customers. Further suppose that the "customer since" date is stored in the physical customer table as an integer column `SINCE` (holding the number of seconds since January 1, 1970) whereas the desired type of the customer profile's `SINCE` element is xs:dateTime to provide consistency with other parts of the target XML applications. ALDSP allows externally provided Java functions to be registered for use in queries, so the data service developer could import an external function `int2date` that performs the required data conversion and call it from within the `tns:getProfile()` data service function body:

```
...
<tns:PROFILE>
  <CID>{fn:data($CUSTOMER/CID)}</CID>
  <LAST_NAME>{
    fn:data($CUSTOMER/LAST_NAME)
  }</LAST_NAME>
  <SINCE>{ns1:int2date($CUSTOMER/SINCE)}</SINCE>
  ...
<tns:PROFILE>
...
```

This provides the desired result, but it does so at the expense of interposing a function call that will be seen as a black box by the compiler. To see why this is a problem, consider what would happen to a query that selects customer profiles for those people who became customers only after a given date $start:

```
for $c in tns:getProfile()
where $c/SINCE gt $start
return $c
```

After function inlining, this query would start with

```
for $c1 in ns3:CUSTOMER()
where ns1:int2date($c1/SINCE) gt $start
return <tns:PROFILE> ... </tns:PROFILE>
```

and the resulting where clause would not be SQL-pushable due to the presence of the Java function.

ALDSP provides *inverse function* support to enable a developer to declare another function `date2int` as the *inverse* of `int2date` and register a *transformation* rule

$$(\texttt{gt}, \texttt{int2date}) \longrightarrow \texttt{gt-intfromdate}$$

where the right-hand side is an XQuery function that is implemented as:

```
declare function f1:gt-intfromdate($x1 as xs:dateTime,
                                   $x2 as xs:dateTime)
  as xs:boolean?{
  f1:date2int($x1) gt f1:date2int($x2)
};
```

```
for $c in CUSTOMER()
where $c/CID eq "CUST001"
return $c/FIRST_NAME
```

↓

```
SELECT t1."FIRST_NAME" AS c1
FROM "CUSTOMER" t1
WHERE t1."CID" = "CUST001"
```

(a) simple select-project

```
for $c in CUSTOMER(),
    $o in ORDER()
where $c/CID eq $o/CID
return
<CUSTOMER_ORDER>{
  $c/CID, $o/OID
}</CUSTOMER_ORDER>
```

↓

```
SELECT t1."CID" AS c1, t2."OID" AS c2
FROM "CUSTOMER" t1
JOIN "ORDER" t2
ON t1."CID" = t2."CID"
```

(b) inner join

```
for $c in CUSTOMER()
return
<CUSTOMER>{
  $c/CID,
  for $o in ORDER()
  where $c/CID eq $o/CID
  return $o/OID
}</CUSTOMER>
```

↓

```
SELECT t1."CID" AS c1, t2."OID" AS c2
FROM "CUSTOMER" t1
LEFT OUTER JOIN "ORDER" t2
ON t1."CID" = t2."CID"
```

(c) outer join

```
for $c in CUSTOMER()
return
<CUSTOMER>{
  if $c/CID eq "CUST001"
  then $c/FIRST_NAME
  else $c/LAST_NAME
}</CUSTOMER>
```

↓

```
SELECT CASE
  WHEN t1."CID" = "CUST001"
  THEN t1."FIRST_NAME"
  ELSE t1."LAST_NAME"
  END AS c1
FROM "CUSTOMER" t1
```

(d) if-then-else

```
for $c in CUSTOMER()
group $c as $p by $c/LAST_NAME as $l
return
<CUSTOMER>{
  $l,
  count($p)
}</CUSTOMER>
```

↓

```
SELECT t1."LAST_NAME" AS c1,
       COUNT(*) AS c2
FROM "CUSTOMER" t1
GROUP BY t1."LAST_NAME"
```

(e) group-by with aggregation

```
for $c in CUSTOMER()
group by $c/LAST_NAME as $l
return $l
```

↓

```
SELECT DISTINCT t1."LAST_NAME" AS c1
FROM "CUSTOMER" t1
```

(f) group-by equivalent of SQL distinct

Table 1: Pushed Patterns (1)

```
for $c in CUSTOMER()
return
<CUSTOMER>
  $c/CID,
  <ORDERS>{
    count(
      for $o in ORDER()
      where $o/CID eq $c/CID
      return $o
    )
  }</ORDERS>
</CUSTOMER
```

↓

```
SELECT t1."CID" AS c1,
       COUNT(t2."CID") AS c2
FROM "CUSTOMER" t1
LEFT OUTER JOIN "ORDER" t2
ON t1."CID" = t2."CID"
GROUP BY t1."CID"
```

(g) outer join with aggregation

```
for $c in CUSTOMER()
where
  some $o in ORDERS()
  satisifes $c/CID eq $o/CID
return $c/CID
```

↓

```
SELECT t1."CID" AS c1
FROM "CUSTOMER" t1
WHERE EXISTS(
  SELECT 1
  FROM "ORDERS" t2
  WHERE t1."CID" = t2."CID")
```

(h) semi join with quantified expression

```
let $cs:=
  for $c in CUSTOMER()
  let $oc := count(
    for $o in ORDER()
    where $c/CID eq $o/CID
    return $o
  )
  order by $oc descending
  return
  <CUSTOMER>{
    data($c/CID), $oc
  }</CUSTOMER>
return
  subsequence($cs, 10, 20)
```

↓ *Oracle*

```
SELECT t4.c1, t4.c2
FROM (
  SELECT ROWNUM AS c3, t3.c1, t3.c2
  FROM (
    SELECT t1."CID" AS c1,
           COUNT(t2."CID") AS c2,
    FROM "CUSTOMER" t1
    LEFT OUTER JOIN "ORDER" t2
    ON t1."CID" = t2."CID"
    GROUP BY t1."CID"
    ORDER BY COUNT(t2."CID") DESC
  ) t3
) t4
WHERE (t4.c3 >= 10) AND (t4.c3 < 20)
```

(i) subsequence() function

Table 2: Pushed Patterns (2)

Based on this additional information, the optimizer will infer that

```
int2date(x) gt y ≡ date2int(int2date(x)) gt date2int(y)
                 ≡ x gt date2int(y)
```

and the SQL pushdown framework will eventually be able to push the selection condition to the source, e.g. as

```
SELECT * FROM "CUSTOMER" t1 WHERE (t1."SINCE" > ?)
```

where the `?` parameter is the result of computing `date2int($start)` in the ALDSP XQuery engine. User-defined transformations and inverse functions constitute a powerful tool for such use cases, and can be used for single- or multi-argument transformations (e.g., full name versus first name and last name). Inverse functions are important both for SQL selection pushdown as well as for making updates possible in the presence of such transformations.

5. QUERY EXECUTION

The basic structure of the ALDSP runtime system is very similar to the one described in [11]. For incorporation in ALDSP, it has undergone changes due to evolution in the XQuery working draft between 2002 and 2004 (not discussed here) and the more data-centric, externally-connected world that ALDSP is designed to serve. Runtime system extensions driven by the new, data service oriented requirements of ALDSP include improvements in processing for larger volumes of data, more efficient handling of relational data (as relational data sources are common), and the addition of various features for handling slow or unavailable data sources. The latter extensions include support for asynchronous (concurrent) accesses to external data sources, middle-tier caching to offload backend data sources and/or to reduce latency for service calls to slowly changing sources, and failover support for slow or unavailable data sources.

5.1 Data Representation

Central to the ALDSP runtime system is the XML token stream originally described in [11]. The token stream provides an internal streaming API (like SAX) for query processing that materializes events (like StAX [12]). Unlike SAX or StAX, it represents the full XQuery Data Model (not just InfoSet), and in ALDSP we always use the typed version of the token stream – each data source adaptor feeds typed tokens into the ALDSP server runtime. More details about BEA's XML token stream representation and associated XML processing architecture can be found in [11].

To streamline the handling of relational data (as well as any other "flat XML" data), ALDSP has extended the previous runtime with several new ways to represent tuples. (Note that although XQuery never actually produces tuples, as they are not XML serializable or part of the data model, XQuery's FLWOR variable bindings imply support for tuples internally in the runtime.) The ALDSP runtime supports three different tuple representations that are optimized for specific use cases, and the optimizer decides which version to use based on the nature of the usage at the point where they are needed. Figure 4 shows the three different representations being used to represent sequence of two tuples. The stream representation consists of a (`BeginTuple`, `EndTuple`) pair that encloses the tuple content, and individual fields are separated by `FieldSeparator`. This representation has fairly low memory requirements but it implies expensive processing if some of the content of a tuple is needed rarely or needs to be skipped over. The single token representation represents the stream of tokens by a single Token; the stream representation has to be extracted from the Token for processing when needed. This representation has higher memory requirements and has expensive processing if accessed, but is cheap when content can be skipped. The array version represents the tuple as an array of Tokens and is usable when every field can be represented by a single Token, such as in the case of data entering ALDSP from relational sources. It has higher memory requirements but provides cheap access to all fields.

5.2 Plans and Operators

As described in [11], all runtime operators are realized as Token Iterators. Token Iterators consume and produce Token Streams in an iterative (and, if possible, streaming) way. ALDSP has added several operators in support of data-centric use cases, with the major ones being tuple-handling operators, a set of join operators, and a grouping operator. Several special-purpose operators were also added and will be mentioned in later subsections.

The tuple-handling operators that were newly added for ALDSP are construct-tuple, extract-field, concat-tuples (to join two tuples as a new, wider tuple), and extract-subtuple (which is essentially the converse of concat-tuples). Since tuples are not part of the XQuery Data Model and not visible at the user level, these operators are not user-accessible. However, they are used heavily internally.

In the original BEA XQuery runtime, joins were expressed as maps and were always performed via nested loops. The introduction of traditional join operators allowed for easy introduction of join processing knowledge into the XQuery engine. The current join repertoire of ALDSP includes nested loop, index nested loop, PP-k using nested loops, and PP-k using index nested loops. (A modular join implementation allows for the use of any existing join method in PP-k.) It is important to note that SQL pushdown is also a join method of sorts – where possible, as explained earlier, ALDSP aims to let underlying relational databases do as much of the join processing as possible for data residing in them, so the join operators in the runtime system are only for cross-source joins (with the most performant one being PP-k using index nested loops).

The ALDSP runtime has just one implementation of the grouping operator. The ALDSP grouping operator relies on input that is pre-clustered with respect to the grouping expression(s). Its job is thus to simply form groups while watching for the grouping expression(s) to change, indicating the start of the next group. If the input would not otherwise be clustered, a sort operator is used to provide the required clustering.

5.3 Adaptors

The ALDSP runtime supports data source adaptors to relational sources, Web services, custom Java functions, and XML and delimited files. Within the runtime, data from relational tables, views, and stored procedures are moved into and out of SQL types using the JDBC API. For Web services, both document/literal and rpc/encoded style Web services are supported. Data coming from Web services is validated according to the schema described in their WSDL in order to create typed token streams, and some special processing (e.g., mapping of arrays) is performed for rpc/encoded style services. For custom Java functions, data is translated to/from standard Java primitive types and classes, and array support is included. Finally, for files, XML schemas are required at file registration time, and are used to validate the data for typed processing in ALDSP.

Data source invocation at runtime involves the following steps for all source types:

1. Establish a connection to the physical datasource.

2. If invocation is parameterized, then translate parameters from

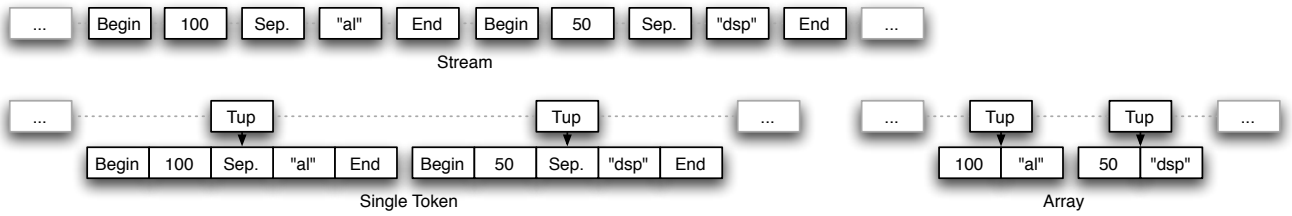

Figure 4: Tuple representations

the XML token stream to the data source data model.

3. Invoke the data source (execute SQL, call a stored procedure, a Web service, etc.)
4. Translate the result into XML token stream form.
5. Release/close the physical connection.

5.4 Asynchronous Execution

For ALDSP queries, a large part of the overall query execution time is usually the time to access external data sources. This time can be spent on the network (e.g., when invoking remote Web services) or processing data inside an external data source (e.g., an RDBMS). To allow large latencies to be overlapped – e.g., to allow several independent Web service calls to occur in parallel – ALDSP extends the built-in XQuery function library with a function that provides XQuery-based control over asynchronous execution. With the extension, a query writer can indicate which parts of his or her query should potentially be executed asynchronously from the main query execution thread using a built-in function

```
fn-bea:async($slow-expr as item()*) as item()*
```

The argument to `fn-bea:async` is an XQuery expression that can be executed on another thread.

5.5 Function Cache

Another potential solution for coping with slow (or expensive) data sources is caching. ALDSP provides data service designers and administrators with a means to ask the system to cache the results of selected data service function invocations. A data service's designer has the ability to indicate statically, given their knowledge of the nature and the semantics of the data service, whether or not to permit caching to ever be used on its data service functions. When allowed, caching can then be enabled administratively with a specified TTL (time to live) on a selected data service function. The ALDSP mid-tier cache can be thought of as a persistent, distributed map that maps a function and a set of arguments values to the corresponding function result. If a data service function has caching enabled and the result of a given invocation is found in the cache and is not stale, it is used; if a cache miss occurs, the function call occurs and its result is both cached and returned to the caller.

The current cache implementation employes a relational database to achieve persistence and distribution in the context of a cluster of ALDSP servers. It is important to notice that ALDSP's cache is a function cache – rather like a Web service cache – so it is appropriate for use in turning high latency data service calls (e.g., involving a slow Web service, a very heavily loaded backend, or some other expensive computation) into single-row database lookups, which gives some indication of the use cases where caching may be appropriate. Also, the ALDSP cache is not a queryable, materialized view – it is a map. Additional caching capabilities are currently under consideration.

5.6 Failover

In a distributed world, especially in a world with Web services and potential downtimes or periods of high load, data sources can sometimes be too slow or unavailable. In such cases, for some applications, an incomplete but fast query result may be preferable to a complete but slow query result (or to no query result). Additionally, data sources can sometimes contain redundant information, so there can be an alternate data source worth trying for missing information in some applications. As a general means to address all of these considerations and potential desires, ALDSP provides XQuery-based control over how long to wait for an expression evaluation to finish. Users can thus handle unavailable or too slow data sources using built-in functions:

```
fn-bea:fail-over($prim-expr as item()*,
                 $alt-expr as item()*) as item()*
```

and

```
fn-bea:timeout($prim-expr as item()*,
               $millis as xs:integer,
               $alt-expr as item()*) as item()*
```

The first function, `fn-bea:fail-over`, takes two arguments, a primary expression and an alternate expression, both of which can be any legal XQuery expression. If the evaluation of `$prim-expr` succeeds, its result is returned, but if the evaluation fails at any point before the end of the query, the result of evaluating the alternate expression `$alt-expr` is returned instead. The second function, `fn-bea:timeout`, is similar, but it takes a time argument as well. If the evaluation of `$prim-expr` succeeds in less than `$millis` milliseconds, the result is returned, but if the evaluation fails or takes more than `$millis` milliseconds, then the result of evaluation `$alt-expr` is returned instead. The function `fn-bea:timeout` is well-suited for handling both slow and unavailable data sources uniformly, as in any event, when the time is up, the system fails over to the alternate expression. In both cases, if a partial result is desired, the empty sequence (( )) can be returned as the alternate.

6. UPDATES

ALDSP utilizes Service Data Objects [8] (a.k.a. SDO), to support updates as well as reads for data obtained from data services. The ALDSP APIs allow client applications to invoke a data service, operate on the results, and then put the data back in order to capture the changes made after the data was obtained. Figure 5 shows a small ALDSP Java mediator API code fragment to illustrate this SDO programming pattern. SDO offers both typed and untyped client-side programming models for application developers to use in navigating and updating XML data. When updates

```
PROFILEDoc sdo = ProfileDS.getProfileById("0815");
sdo.setLAST_NAME("Smith");
ProfileDS.submit(sdo);
```

Figure 5: Use of SDO in Java

affect an SDO object, such as changing the last name field in the small example here, the affected SDO object tracks the changes. When a changed SDO is sent back to ALDSP, what is sent back is the new XML data plus a serialized "change log" identifying the portions of the XML data that were changed and what their previous values were. ALDSP examines the change log and determines how to propagate the changes back to the underlying physical data sources. Unaffected data sources are not involved in the update, and unchanged portions of affected sources' data are not updated.

Change propagation requires ALDSP to identify where changed data originated – its lineage must be determined. ALDSP performs automatic computation of the lineage for a data service from the query body of the data service function designated by the data service designer as being its lineage provider. (By default this is the first function, and should be the "get all" function if there is one.) Space precludes provision of much detail here, but the computation is performed by a specialized rule set driven by the same rule engine used for the XQuery optimizer. Primary key information, query predicates, and query result shapes are used together to determine which data in which sources are affected by a given update. Also, as alluded to earlier, ALDSP includes inverse functions in its lineage analysis, enabling updates to transformed data when inverses are provided. As an example, for the customer profile update in Figure 5, the SDO object returned to ALDSP will indicate that only `LAST_NAME` has changed. Lineage analysis will determine that the underlying source for `LAST_NAME` is the `CUSTOMER` physical data service, and the update will be propagated only to that source. The other sources involved in the customer profile view are unaffected and will not participate in this update at all.

Each data service has a *submit* method that is called to submit a changed SDO or set of SDOs back to ALDSP, and the unit of update execution is a submit call. In the event that all data sources are relational and can participate in a two-phase commit (XA) protocol, the entire submit is executed as an atomic transaction across the affected sources. Since objects are read in one transaction, operated on, and then re-submitted later, ALDSP supports optimistic concurrency options that the data service designer can choose from in order to have the system decide if a given set of changes can safely be submitted. Choices include requiring all values read to still be the same (at update time) as their original (read time) values, requiring all values updated to still be the same, or requiring a designated subset of the data (e.g., a timestamp element or attribute) to still be the same. ALDSP uses this in the relational case to condition the SQL update queries that it generates (i.e., the sameness required is expressed as part of the where clause for the update statements sent to the underlying sources). ALDSP also provides an update override facility that allows user code to extend or replace ALDSP's default update handling [13].

7. SECURITY

ALDSP provides a flexible, fine-grained access control model for data services. Access control is available both on data service functions (specifying who is allowed to call what) and on the schemas of the return types of data service functions (permitting much finer control). Authentication and authorization services are provided by the BEA WebLogic Server security framework, which also permits third-party security plug-ins to be utilized. From a query processing standpoint, it is the fine-grained, element/attribute-level access control functionality of ALDSP that is the most interesting. For this, an individual subtree in a data shape returned by a data service may be represented as a labeled security resource. A security service administrator can then control the access policy for this resource. Unauthorized accessors will either see nothing (the data may be silently removed, if the presence of the subtree is optional in the schema) or they will see an administratively-specified replacement value.

In terms of query processing and caching, fine-grained security filtering occur at a fairly late stage of query processing. This is done so that compiled query plans and function results can still be effectively cached and reused across different users in ALDSP, even when fine-grained security is active. Function result caching is done before security filters have been applied, thereby making the cache effective across users; security filtering is applied to the results obtained in the event of a cache hit. In addition to access control, the ALDSP runtime has a fairly extensive set of auditing capabilities that utilize an auditing security service. Auditing can be administratively enabled in order to monitor security decisions, data values, and other operational behavior at varying levels of detail [13].

8. RELATED SYSTEMS

Too little can be said about related systems in the space remaining, but we point out several of the highlights here. ALDSP's heritage can be traced through many years of work in the database research community on distributed databases, heterogeneous distributed databases, federated databases, and multidatabases. It was actually inspired, however, by watching real users of real commercial integration solutions – mostly of the workflow or process orchestration variety – struggle to use those products to create what amount to distributed query plans by hand in order to serve bits of data from disparate sources up to applications, such as portals, that need it in real-time. In terms of the literature, ALDSP is most closely aligned with the pioneering work on the Functional Data Model [14] and the MultiBase federated database system [15] that first used it to solve a similar problem across relational and network databases approximately twenty-five years ago. We differ in applying modern XML technologies to the problem (XML, XML Schema, and the functional query language XQuery), which allows us to enjoy a "free ride" on the Web services and SOA waves that are causing all sorts of applications to be functionally accessible through open standard Web service APIs. Despite these differences, however, and the more coarse-grained nature of our approach, we were definitely influenced by the "everything is a function" model and its uncanny applicability to today's SOA universe.

In terms of other commercial systems, ALDSP is loosely related to virtual relational database based data integration products such as IBM's WebSphere Information Integrator [16], the Composite Information Server from Composite Software [17], and MetaMatrix Enterprise [18]. ALDSP is related to these products in that they are also aimed at composition of data across sources. However, it is only loosely related to them, as all three have taken a relational approach to the problem and then glued a degree of service support on the bottom (to consume data from services) as well as on the top (to create tagged/nested XML as data is on its way out of the system). This relational plus XML/service glue approach is awkward for modeling services (e.g., an order management system) that return nested XML documents as results; normalizing rich services into tabular views causes usability problems in several dimensions. Also, while the glue approach allows XML results to be returned,

the resulting services are neither efficiently composable nor efficiently queryable when this approach is used to create them. Several other vendors offer XQuery-based query engines that can access data from multiple sources, e.g., Software AG [19], DataDirect [20], and Ipedo [21], but none offers data service modeling or supports both reads and updates in the manner that ALDSP does.

9. CONCLUSION

In this paper, we have provided a technical tour of the BEA AquaLogic Data Services Platform from a query processing perspective. We have explained the world model that the product is based on, provided an overview of its architecture and features, and then discussed its query compiler, optimizer, and runtime system in more detail. The goal of the paper is to raise awareness and educate our colleagues in academic and industrial research about BEA's unique approach to data in the SOA world – i.e., data services – and about our extensive use of XQuery as a declarative foundation to support the creation, optimization, and efficient execution of data services. This is essentially a companion paper to [7]; that paper offers a much broader overview of ALDSP from a user's point of view, but contains little information about how it works inside. In contrast, we hope that the material presented in the present paper will be of interest and use to XML query processing enthusiasts in both academia and industry. Readers curious about performance aspects of ALDSP may be interested in [22].

Like any system, ALDSP is a work in progress. We have a number of extensions on the roadmap for the ALDSP product. In the query processing area, future plans include additional work on query optimization and on support for new data sources. With respect to query optimization, we are starting work on an observed cost-based approach to optimization and tuning; the idea is to skip past "old school" techniques that rely on static cost models and difficult-to-obtain statistics, instead instrumenting the system and basing its optimization decisions (such as evaluation ordering and parallelization) only on actually observed data characteristics and data source behavior. We are also planning support for "hints", but not for hints about physical operators; in a world with layers of abstraction, we need declarative hints that can survive correctly through layers of views and associated query rewrite optimizations. Finally, with respect to data source support, we have begun the creation of an extensible pushdown framework for use in teaching the ALDSP query processor to push work down to queryable data sources such as LDAP and/or to non-relational mainframe data sources; support for pushing work to queryable XML data sources is also on our long-term roadmap.

10. ACKNOWLEDGMENTS

Any large software product is necessarily a team effort. The authors would like to acknowledge the efforts of the entire ALDSP team at BEA, both present and past, for their contributions to creating, testing, and documenting the system described here.

11. REFERENCES

[1] G. Wiederhold, P. Wegner, and S. Ceri. Towards mega-programming. *Communications of the ACM*, 11(35):89–99, 1992.

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

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

[4] M. Singh and M. Huhns. *Service-Oriented Computing: Semantics, Processes, Agents*. Wiley, West Sussex, England, 2005.

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

[6] 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.

[7] M. Carey and the AquaLogic Data Services Platform Team. Data delivery in a service-oriented world: The BEA AquaLogic data services platform. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Chicago, IL, 2006.

[8] K. Williams and B. Daniel. An introduction to service data objects. *Java Developer's Journal*, October 2004.

[9] W3C. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/2004/WD-xquery-20040723/, July 2004.

[10] P. Reveliotis and M. Carey. Your enterprise on XQuery and XML schema: XML-based data and metadata integration. *Proc. of the 3rd Int'l. Workshop on XML Schema and Data Management (XSDM)*, April 2006.

[11] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. Carey, and A. Sundararajan. The BEA streaming XQuery processor. *The VLDB Journal*, 13(3):294–315, 2004.

[12] R. Benson. JSR 173: Streaming API for XML. http://jcp.org/en/jsr/detail?id=173.

[13] BEA Systems, Inc. BEA AquaLogic Data Services Platform™ 2.1. http://edocs.bea.com/aldsp/docs21/index.html.

[14] D. Shipman. The functional data model and the data language DAPLEX. *ACM Transactions on Database Systems*, 6(1):140–173, 1981.

[15] U. Dayal. *Query Processing in Database Systems*, chapter Query Processing in a Multidatabase System, pages 81–108. Springer-Verlag, New York, 1985.

[16] L. Haas, E. Tien Lin, and M. Tork Roth. Data integration through database federation. *IBM Systems Journal*, 41(4):578–596, 2002.

[17] Composite Information Server. http://www.compositesoftware.com/products/cis.shtml.

[18] R. Hauch, A. Miller, and R. Cardwell. Information intelligence: metadata for information discovery, access, and integration. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 793–798, Baltimore, Maryland, U.S.A., 2005. ACM Press.

[19] T. Fiebig and H. Schöning. Software AG's Tamino XQuery Processor. In *Proc. of the First Int'l. Workshop on XQuery Implementation, Experience and Perspectives <XIME-P/>*, pages 19–24, 2004.

[20] DataDirect XQuery. http://www.datadirect.com/products/xquery/.

[21] Ipedo XIP. http://www.ipedo.com/html/ipedo_xip.html.

[22] BEA Systems, Inc. BEA AquaLogic Data Services Platform Performance: A benchmark-based case study. http://www.bea.com/content/news_events/white_papers/BEA_ALDSP_Perf_Study_wp.pdf, December 2005.