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.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, to post on servers or to redistribute to lists, requires a fee and/or special permission from the publisher, ACM.
VLDB '06, September 12-15, 2006, Seoul, Korea.
Copyright 2006 VLDB Endowment, ACM 1-59593-385-9/06/09.

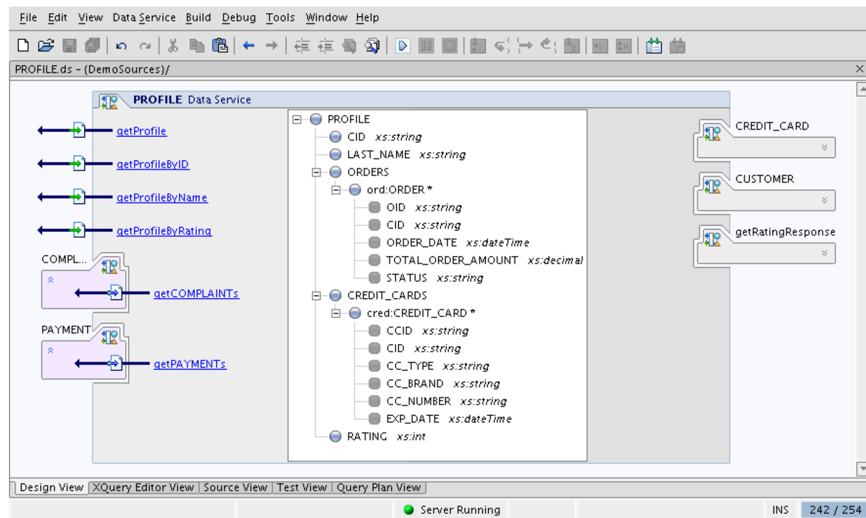

Figure 1: ALDSP Data Service – Design View

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

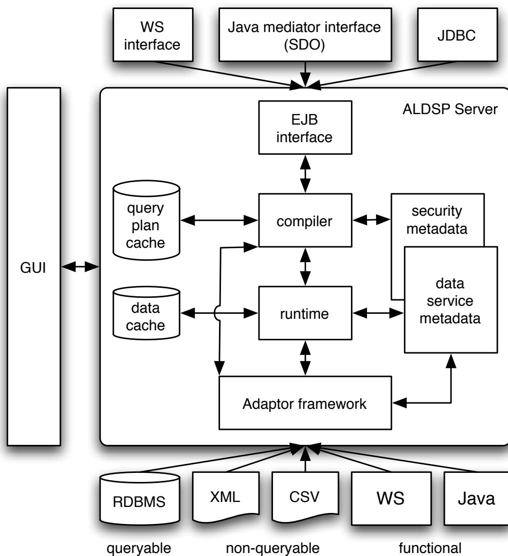

Figure 2: Overview of ALDSP Architecture

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.