Data Delivery in a Service-Oriented World: The BEA AquaLogic Data Services Platform

Michael Carey and the
AquaLogic Data Services Platform Team
BEA Systems, Inc.
San Jose, CA, USA
mcarey@bea.com

ABSTRACT

"Wow. I fell asleep listening to SOA music, and when I woke up, I couldn't remember where I'd put my data. Now what?" Has this happened to you? With the new push towards service-oriented architectures (SOA) and process orientation, data seems to have been lost in the shuffle. At the end of the day, however, applications are still about data, and SOA applications are no different. In this paper, we present BEA's approach to serving up data to SOA applications. 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. ALDSP provides a new, declarative foundation for building SOA applications and services that need to access and compose information from a range of enterprise data sources. The paper covers both the foundation and the key features of ALDSP, including its underlying technologies, its overall system architecture, and its most interesting capabilities.

1. INTRODUCTION

The introduction of relational database management systems in the 1970's created a productive new world that enabled developers of data-centric applications to work much more efficiently than ever before. Application development in the pre-relational era meant hand-writing, tuning, and maintaining large procedural programs to access and manipulate the application's data. Relational database systems made it possible for developers to write much simpler, declarative queries to accomplish the same tasks. Physical system details such as indexes and clustering were hidden by the relational model, enabling developers to focus first on the logical tasks at hand. Performance could be tuned later on since changes in the physical schema no longer required their application code to change as well. Higher-level views could be created if desired, making it possible for application-tailored schemas to be created on top of the main relational schema. The relational revolution was a huge success by all measures – virtually all major enterprise applications employ relational databases for their persistence, and large enterprises have handed many of their operations over to packaged applications such as SAP, PeopleSoft, Siebel, and SalesForce.com.

Developers of data-centric enterprise applications face a new and different barrier today. Relational databases have been so successful that there are many of them available (e.g., Oracle, DB2, SQL Server, and MySQL, to name a few of the more prominent ones). A typical enterprise is likely to have a number of relational databases within its corporate walls, and information about key business entities such as customers or employees is likely to reside in at least several of these systems. In addition, much of the information, even if it is stored relationally, will be relationally inaccesssible because it is under tight application control. Access to application-controlled information must come through the application APIs, as they enforce the rules and logic of the "business objects" of the application. As a result, enterprise application developers currently face a huge integration challenge: A given business entity of interest is now likely to reside in a mix of relational databases, packaged applications, and perhaps even in files or in legacy mainframe systems and/or applications. When a new, "composite" application needs to be created from these parts, it's back to procedural programming. This time, however, it's procedural programming against a wide variety of different subsystems, APIs, and data formats – essentially hand-coding what amounts to a distributed query or update plan.

The enterprise IT world is abuzz today about a new trend – service-oriented architecture (SOA) [1]. The goal of SOA, anticipated in [2] where it was referred to as megaprogramming, is composite application development. To respond to rapidly changing business requirements, IT organizations have a need to develop new applications by reusing their existing application and data assets as buildng blocks. XML-based Web services [3] are a step in the right direction, offering physical-level normalization for intra- and inter-enterprise function invocation and information exchange. Web service orchestration and coordination languages [4] are another step forward, on the process side, but they are still procedural languages by nature. Thus, when it comes to developing composite data access or update logic, developers are still essentially left to design, tune, and maintain brittle distributed query or update plans by hand. To truly support the data requirements of composite application development, we need more – we need a *declarative* way to create *data services* [5] for composite applications.

At BEA, we are exploiting the opportunity created by the recent rapid emergence of Web services and XML standards for enterprise application integration. In particular, we are leveraging the W3C XML, XML Schema, and XQuery recommendations to deliver a standards-based foundation for declarative data services development [6]. The BEA AquaLogic Data Services Platform (ALDSP) [7], first introduced in mid-2005 as ALDSP 2.0, targets developers of composite applications that need to access and compose information from a range of enterprise data sources. Sup-

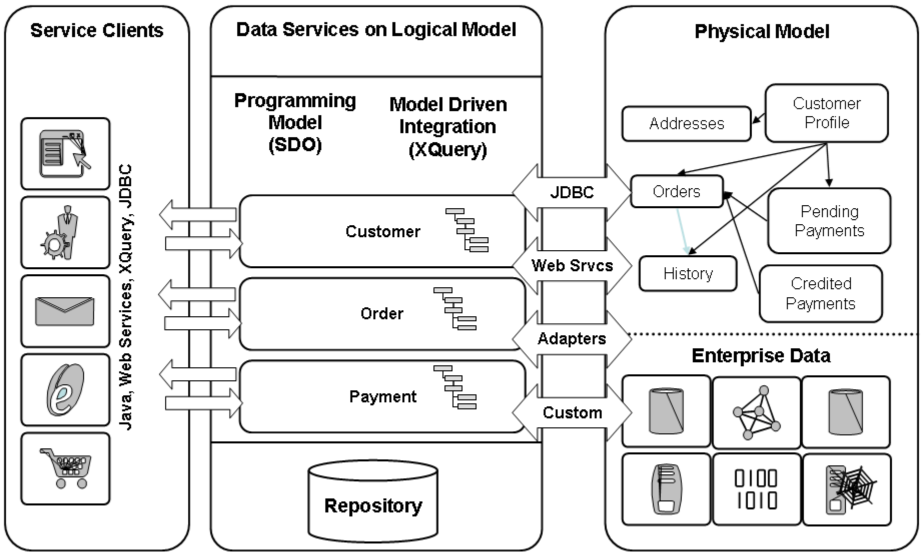


Figure 1: Data services *a la* ALDSP.

ported sources include Web services, packaged applications, stored procedures, relational tables and views, XML and delimited files, and Java-fronted custom data sources. In this paper, which is aimed at the database R&D community, we provide a technical overview of ALDSP. We discuss both the foundations and the key features of ALDSP, including its underlying technologies, its architecture, and its salient capabilities.

The remainder of this paper is organized as follows: Section 2 reviews the concept of data services and explains how ALDSP supports the design, development, and maintenance of declarative, reusable data services. Section 3 provides an overview of ALDSP's architecture and its approach to XML query processing, showing how the underlying technology enables ALDSP to truly deliver on the promises of declarative data services and reuse. Section 4 provides an overview of ALDSP's approach to handling updates to data services, showing how these promises are also extended to update processing when possible and how developers can extend or replace ALDSP's default update processing when desired. Section 5 briefly discusses some key related work that influenced ALDSP as well as comparing ALDSP to a few of the most closely related commercial systems. Finally, Section 6 concludes the paper and lists some of the current extensions under development for incorporation into future releases of ALDSP. In addition, for in-person attendees of SIGMOD 2006, ALDSP will be demonstrated live as well [8].

2. DATA SERVICES IN ALDSP

2.1 Declarative Data Services

ALDSP was designed from the ground up to provide support for the concept of data services as first described in [5] and detailed further in [6]; the latter makes the technical case for declarative, XML-based data services. Since it targets the world of SOA, ALDSP takes a service-oriented view of data. ALDSP models an enterprise (or a portion of interest) as a set of interrelated data services, each of which is "about" a particular coarse-grained business object type (such as customer, order, employee, or service case). Each data service has a "shape", which is a description of the information content of its associated business object type; ALDSP uses XML Schema to describe each data service's shape. Each data service provides a set of service calls that an ALDSP client (or another service) can use to access and modify instances of its business object type. Typically the majority of these service calls are read methods, each of which provides one way to fetch one or more instances of the data service's business objects. Others are navigation methods, calls that traverse relationships from a business object instance returned by the service (e.g., customer) to one or more instances of a business object type fronted by a second data service (e.g., order). Last but not least, others are write methods, calls that support updating (e.g., modifying, inserting, or deleting) one or more instances of the data service's business objects. Each of the service calls for a data service is modeled as an XQuery function (either system-provided or user-defined) that can be called in queries and/or leveraged in the creation of other, higher-level, user-defined data services.

Figure 1 provides a high-level summary of the ALDSP approach to data services. On the bottom right is reality – a mix of databases, Web services, packaged applications, legacy mainframe data, and so on. When pointed at an enterprise data source by a developer, ALDSP introspects the source's metadata (e.g., the SQL metadata for a relational data source or WSDL file for a Web service). This introspection guides the automatic creation of one or more *physical data services* that make the data source available for use in the XML world of ALDSP. Applying this process to a relational data source yields one data service (with one read method and one update method) per table or view; the return shape in this case is the natural XML-ification of a row of the table or view. In the presence of foreign key constraints, introspection also produces navigation functions that exploit the constraints. Introspection of a given Web service yields one data service per distinct Web service operation return type, with functions corresponding to each of the Web ser-

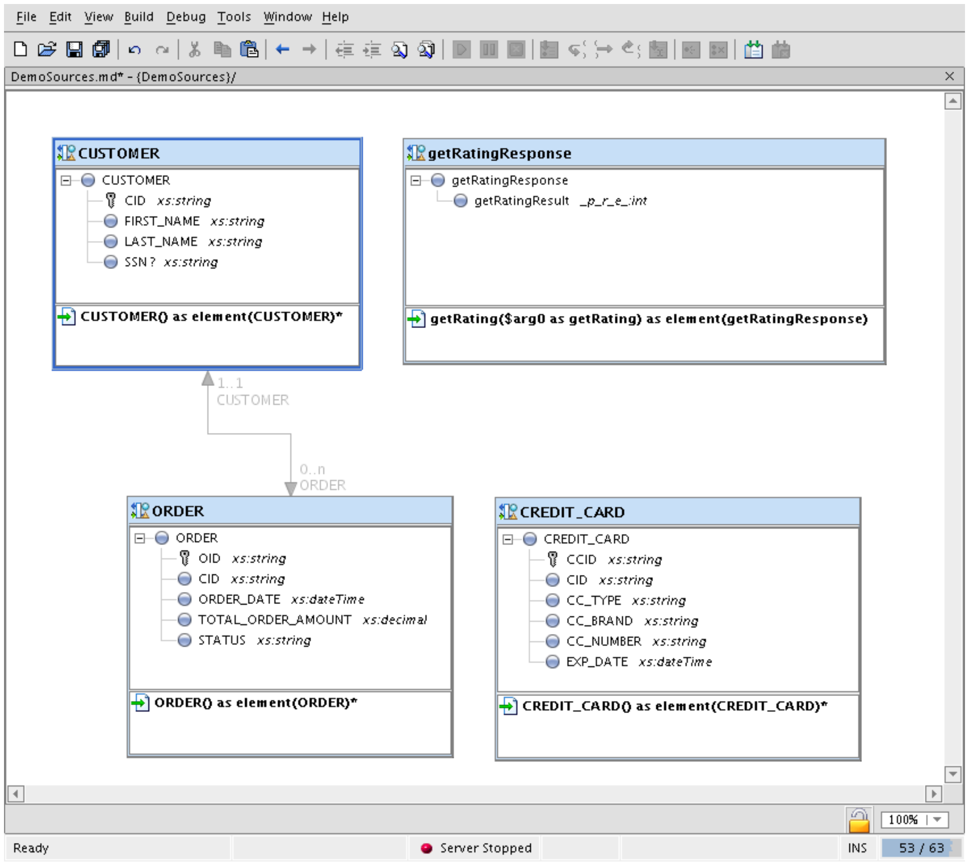


Figure 2: Physical data services.

vice operations and each functions' input and output types corresponding to the information in the WSDL. Other functional data sources are similarly modeled. The result, referred to as the physical model in Figure 1, is a uniform, "everything is a data service" view of the enterprise's data sources.

The center of Figure 1 shows the view of enterprise information seen by client applications. Rather than interacting directly with the underlying data sources, or even with the more uniform physical model, an ALDSP client application interacts with a meaningful set of interrelated logical data services. As discussed above, each such data service is "about" some coarse-grained business entity that was chosen to be meaningful to applications and to serve as a reasonable unit of interaction for client-server calls. Each logical data service is internally composed from one or more data services of the physical model (or lower-level logical data services) using XQuery as the service composition language. The lower-level data services appear to the composer as functions that consume and produce simple- and complex-typed XML data. XQuery is a declarative, functional language which is ideally suited to orchestrating such functions and also to transforming XML data [9] in order to produce the desired business object shapes out of the information accessible from the underlying services.

2.2 Building Data Services in ALDSP

To really convey what ALDSP and declarative data services are about, we will use a simple example. Suppose that we wish to build a logical data service that enables applications (or other services) to access customer profile information. Further, suppose that the useful information about customers resides in several different enterprise information sources. Suppose that one relational database (say an Oracle database) contains CUSTOMER and ORDER tables, while a second relational database (perhaps DB2) contains customer CREDIT_CARD information. In addition to these tabular data sources, suppose that a document-style Web service, hosted somewhere within the enterprise, provides a credit rating lookup capability that accepts a customer's name and social security number (as a getRating input document) and returns a credit rating for the customer. Figure 2 shows the physical-level data services model that would result from asking ALDSP to introspect these data source and make them available for use in creating new, composite data services. The physical model diagram summarizes the shapes and operations associated with each service, and it also shows a relationship between the customer and order services that was automatically created due to the co-location of these tables in the same database and the foreign key constraint that existed between them there.

Figure 3 shows the end goal of our simple example. It shows the ALDSP design view for the integrated customer profile data service that we are aiming to create. Central to the service is its shape; note that an instance of the shape will contain customer data, nested order data, nested credit card data, and a credit rating for a given customer. The upper left-hand side of the design view lists the read methods, which in our example include getProfile( ) to get the integrated profile for all customers, getProfileById( ) to get the profile for one customer by customer id, getProfileByName( ) to get the integrated profiles for all customers with a specified name, and getProfileByRating( ) to get profiles for all customers with a specified credit rating. The lower left-hand side lists the navigation

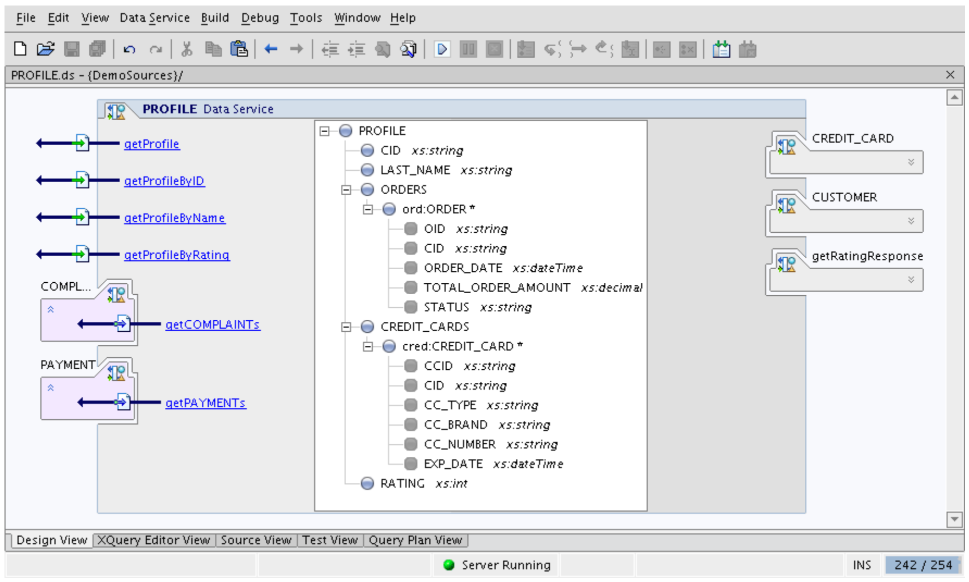


Figure 3: Customer profile data service (design view).

```
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 4: Customer profile data service (source view).

methods. In the figure we assume two related data services, one for customer complaint records and one for payment records. Finally, while not relevant to the consumers of a data service, the right-hand side of the design view shows lower-level services that the service in question depends on (i.e., uses in its implementation).

Figure 4 shows the XQuery source code for several of the data service functions for our customer profile example. The first read method, getProfile( ), takes zero arguments and is responsible for computing and returning complete profiles for all customers with entries in the CUSTOMER table of the first relational database. Note that table contents are accessed via XQuery functions 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 the navigation function that ALDSP automatically created for this purpose based on key/foreign key metadata. Also for each customer, the CREDIT_CARD table in the second relational database is accessed, and finally the credit rating Web service is called. The result of this function is a series of `<PROFILE>` elements, one per CUSTOMER, that integrates all this information from the different data sources. It is important to notice that this "get all instances" function encapsulates all of the integration and correlation details of the customer profile data service. Once the integration and correlation have been done in the main function for the data service, each remaining read method is trivial to specify, as shown for getProfileById( ) in Figure 4. This "integrate once and reuse" design pattern for data services is an extremely important design pattern, as it is the key to achieving data independence in ALDSP. Finally, it is also worth noting that ALDSP developers can choose to avoid working at the XQuery source level when building composite data services, as ALDSP's design-time tool set includes both a visual XQuery editor as well as an XQuery source-level editor.

2.3 Client APIs and Security

ALDSP supports several flavors of API, each tailored to different data service usage scenerios. These include two Java APIs based on returning and re-submitting service data objects (SDO) [10] for data transfer purposes. The SDO standard is a work in progress by IBM, BEA, Oracle, SAP, and XCalia that specifies a programming model and an XML wire format for transmitting a graph of complex objects to a client, accessing and updating the graph there (disconnected from the data provider), and then returning the object graph together with any changes back to the original provider for application of the changes. ALDSP has a Java mediator API that offers a choice of statically- or dynamically-typed SDO programming. In this API, there is a proxy for each data service that supports data service calls (which each return SDOs to the caller), ad hoc queries, and submission of changed SDO graphs back to the service. ALDSP also has a Java control API that is very similar but tailored specifically for programmers who prefer working with BEA WebLogic Workshop Java controls. ALDSP also supports data access, updates, and ad hoc queries through generation of Web service APIs. For data service calls, each of these different APIs also provides a means for additional sorting/filtering requests to be made at data service function invocation time, thereby allowing a core set of data services to serve a broader range of data access needs. Finally, to support SQL-based reporting tools, ALDSP provides a JDBC driver that accepts SQL92 queries against "flat" data services (i.e., data services without nested shapes).

To provide access control, all of the functions for all of the available data services are securable resources managed by the BEA WebLogic Server security framework. Thus, security policies can be written to allow or deny access to any data service function by any outside user/group. ALDSP's hoc query capabilities and administrative features are also subject to access control via this framework. Security is "perimeter-based" in nature, in that access control is applied only at the client-server boundary of the system. Finger-grained access control is also supported. In particular, ALDSP provides the capability for a data service designer to register sensitive elements from the return shape of a data service as additionally securable resources – and then to control access to these elements via the same security framework. When access to some sub-shape is denied, the designer can choose to have the element in question omitted from the result (schema permitting, of course) or replaced with a specified default element value. Finally, to provide additional flexibility, ALDSP makes it possible for a data service designer to implement complex security policies using XQuery itself. It does this through the provision of a few special XQuery functions that provide access to the currently running security principal and that make it possible to ask "is access allowed?" given a security principal and a resource id.

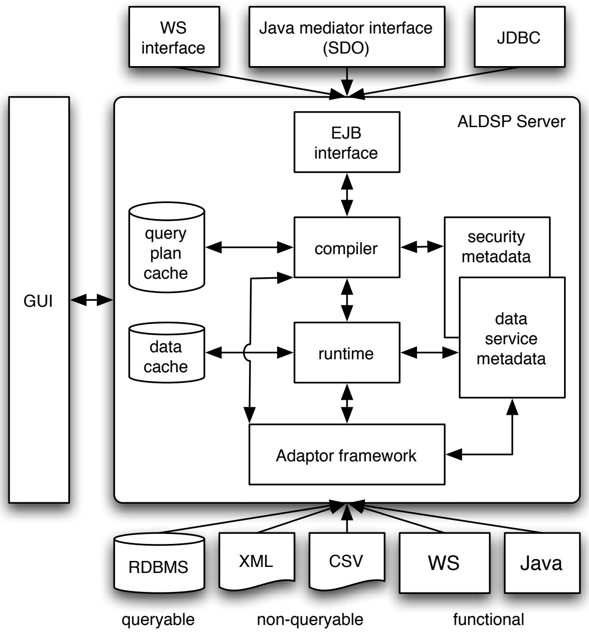
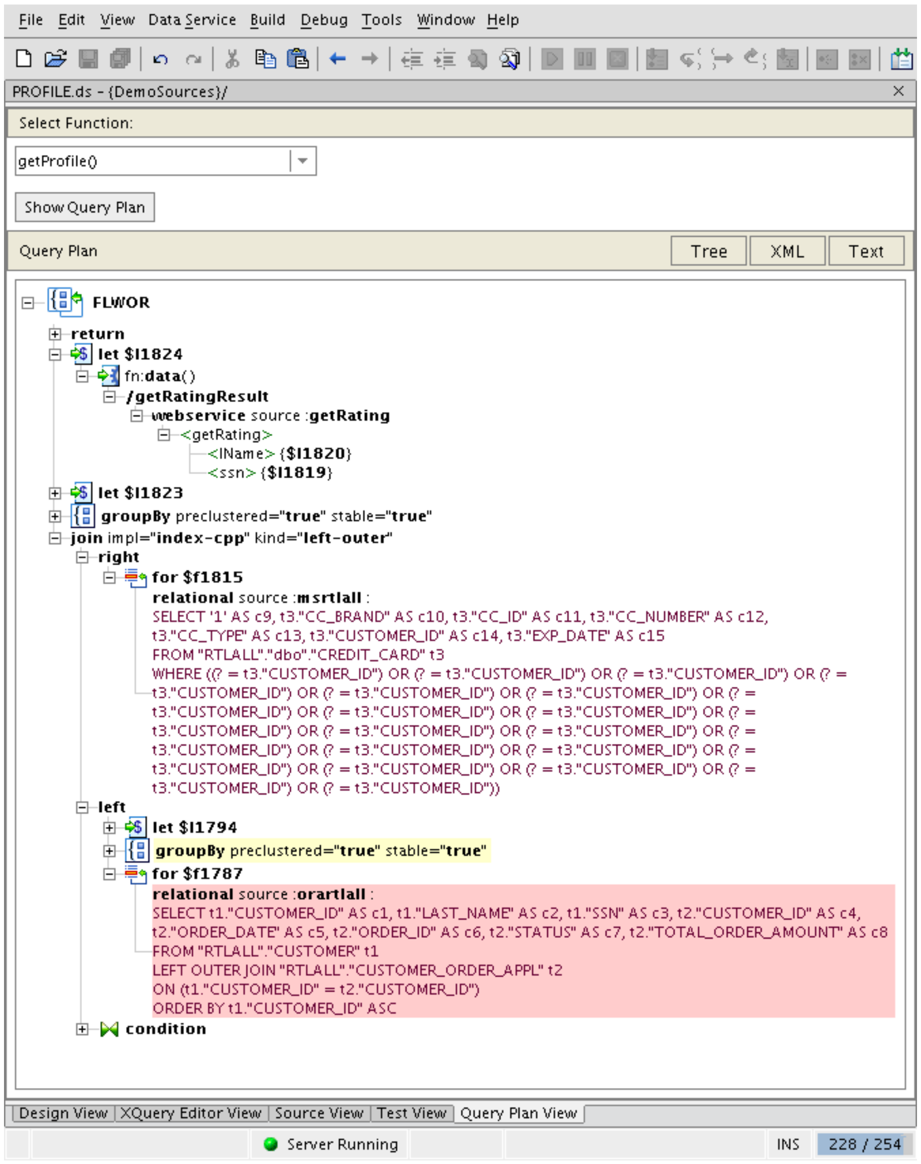


Figure 5: Overview of ALDSP architecture.

3. QUERY PROCESSING IN ALDSP

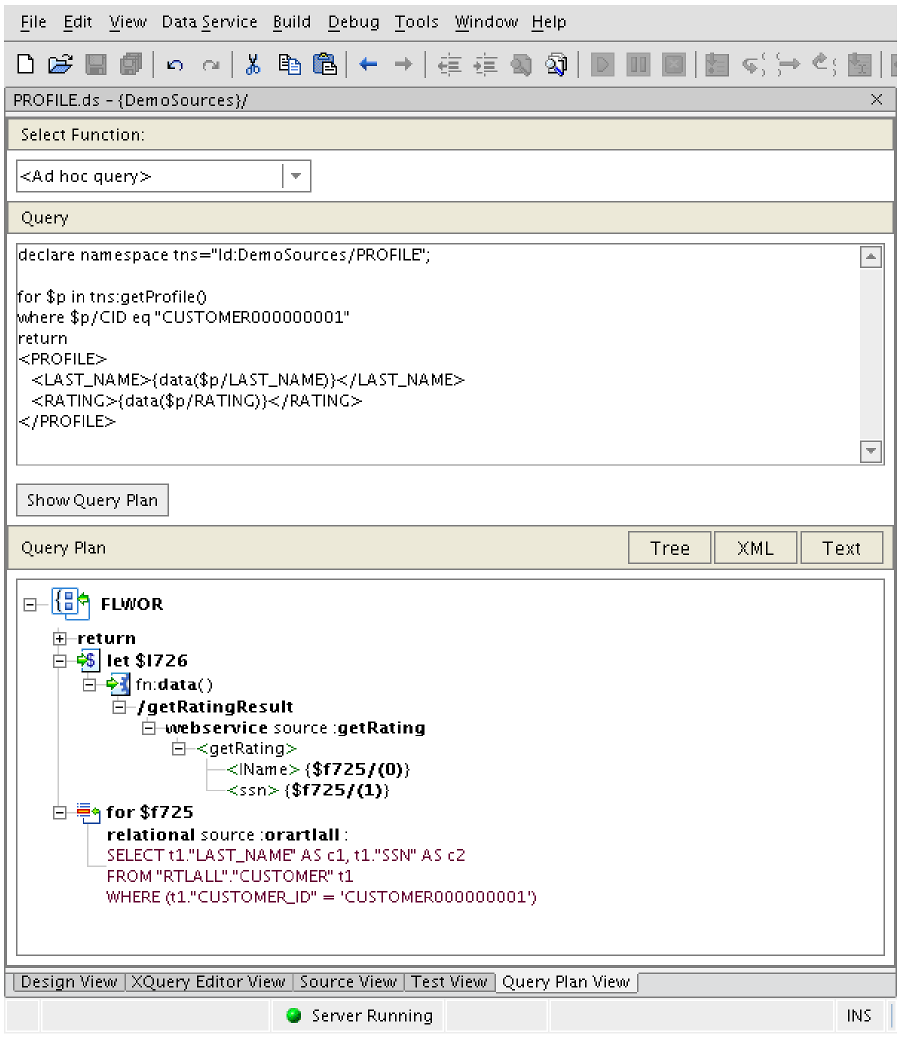
3.1 Query Processing Overview

Figure 5 provides an overview of ALDSP's internal architecture. At the bottom of the figure are various types of data sources that ALDSP supports. These are categorized into queryable sources, to which ALDSP can offload query processing (just RDBMS sources today), non-queryable sources (XML and non-XML files), from which ALDSP can simply fetch all data, and functional sources (including Web services and custom Java functions), from which ALDSP can obtain data only by invoking a function that requires a set of input parameter values. At the top of the figure are the various APIs that ALDSP provides for client access. Additionally, though not depicted in the figure, ALDSP provides server-side APIs for handling large XML results in a streaming fashion. At the left side of the figure is the ALDSP GUI, a set of graphical tools that live in the BEA WebLogic Workshop IDE and support the design and



```
FLWOR
  return
  let $l1824
    fn:data()
      /getRatingResult
        webservice source :getRating
          <getRating>
            <lName> {$l1820}
            <ssn> {$l1819}
  let $l1823
  groupBy preclustered="true" stable="true"
  join impl="index-cpp" kind="left-outer"
    right
      for $f1815
        relational source :msrtlall :
        SELECT '1' AS c9, t3."CC_BRAND" AS c10, t3."CC_ID" AS c11, t3."CC_NUMBER" AS c12,
        t3."CC_TYPE" AS c13, t3."CUSTOMER_ID" AS c14, t3."EXP_DATE" AS c15
        FROM "RTLALL"."dbo"."CREDIT_CARD" t3
        WHERE ((? = t3."CUSTOMER_ID") OR (? = t3."CUSTOMER_ID") OR (? = t3."CUSTOMER_ID") OR (? =
        t3."CUSTOMER_ID") OR (? = t3."CUSTOMER_ID") OR (? = t3."CUSTOMER_ID") OR (? =
        t3."CUSTOMER_ID") OR (? = t3."CUSTOMER_ID") OR (? = t3."CUSTOMER_ID") OR (? =
        t3."CUSTOMER_ID") OR (? = t3."CUSTOMER_ID") OR (? = t3."CUSTOMER_ID") OR (? =
        t3."CUSTOMER_ID") OR (? = t3."CUSTOMER_ID") OR (? = t3."CUSTOMER_ID") OR (? =
        t3."CUSTOMER_ID") OR (? = t3."CUSTOMER_ID") OR (? = t3."CUSTOMER_ID") OR (? =
        t3."CUSTOMER_ID") OR (? = t3."CUSTOMER_ID"))
    left
      let $l1794
      groupBy preclustered="true" stable="true"
      for $f1787
        relational source :orartlall :
        SELECT t1."CUSTOMER_ID" AS c1, t1."LAST_NAME" AS c2, t1."SSN" AS c3, t2."CUSTOMER_ID" AS c4,
        t2."ORDER_DATE" AS c5, t2."ORDER_ID" AS c6, t2."STATUS" AS c7, t2."TOTAL_ORDER_AMOUNT" AS c8
        FROM "RTLALL"."CUSTOMER" t1
        LEFT OUTER JOIN "RTLALL"."CUSTOMER_ORDER_APPL" t2
        ON (t1."CUSTOMER_ID" = t2."CUSTOMER_ID")
        ORDER BY t1."CUSTOMER_ID" ASC
    condition
```

Figure 6: Query plan for getProfile( ).



```
declare namespace tns="ld:DemoSources/PROFILE";

for $p in tns:getProfile()
where $p/CID eq "CUSTOMER000000001"
return
<PROFILE>
  <LAST_NAME>{data($p/LAST_NAME)}</LAST_NAME>
  <RATING>{data($p/RATING)}</RATING>
</PROFILE>
```

```
FLWOR
  return
  let $l726
    fn:data()
      /getRatingResult
        webservice source :getRating
          <getRating>
            <lName> {$f725/(0)}
            <ssn> {$f725/(1)}
  for $f725
    relational source :orartlall :
    SELECT t1."LAST_NAME" AS c1, t1."SSN" AS c2
    FROM "RTLALL"."CUSTOMER" t1
    WHERE (t1."CUSTOMER_ID" = 'CUSTOMER000000001')
```

Figure 7: Query plan for query over getProfile( ).

maintenance of data services.

In the center of Figure 5 is a block diagram of the main components of the ALDSP server from the point of view of query processing. Two of the main components are the ALDSP XQuery compiler and runtime system. These are descended from components of the core BEA streaming XQuery engine [11], but with significant extensions in the areas of query optimization, data source adaptors, and runtime handling of tuple-oriented data as well as nested XML data. The ALDSP server includes both a query plan cache and a data cache to speed query compilation and execution, respectively, when cache hits occur. (We will discuss these caches further later on.) Metadata is maintained about all data services [12], whether logical or physical services, and is used to influence the query compiler's optimization-related decision-making. Security metadata is maintained and drives the enforcement of security policies.

3.2 Optimization: The Key to Reuse

As described earlier, ALDSP aims to make it possible for developers to rapidly (i.e., declaratively) develop data services. Moreover, ALDSP aims to support an environment where these services can be developed once and then reused anywhere and everywhere they might be useful – i.e., "integrate once and reuse". This is easy to claim, but in order to deliver on this promise, ALDSP must provide a set of assurances that include:

1. A data service call, when executed, must be efficient. It must be at least as efficient as the code that a reasonable do-it-yourself data service developer would naturally hand-write using Java or a modern EAI tool.

2. A data service call whose definition is an XQuery that invokes other logical data services must be efficient. The use of layering and abstraction should not carry a runtime price tag.

3. A query or data service call that ends up using only a subset of the results from an underlying logical data service call must be efficient. Predicates must be evaluated as early and as close to the sources as possible, and sources whose data is not referenced by a query should not be involved in the execution of a query at all.

Only if these assurances are actually provided will developers feel safe in using ALDSP to build data services in the declarative, layered fashion that offers them the greatest benefits.

In order to deliver on its promises, ALDSP employs a number of query optimization and execution techniques borrowed from the worlds of database query optimization and distributed query processing. One of the most important aspects of ALDSP query compilation is compile-time function composition, which is similar to the view rewriting and unnesting optimizations found in relational query processors. During query compilation, function calls are inlined, queries are unnested when possible, XML element construction/deconstruction operations cancel one another out, and XML element construction operations whose results go unused are eliminated. These optimizations are critical to the latter two assurances listed above. ALDSP also works to push as much work as possible to the underlying sources. In particular, when relational sources are involved, ALDSP takes the viewpoint that relational DBMSs should be asked and expected to do what they have been becoming increasingly good at over the past few decades, so it strives to push as many of a query's selection, join, grouping, sorting, subquery, case expression, and other operations down in SQL form as it can rather than evaluating them in the middle tier. For XML queries that nest data, ALDSP pushes outerjoins and key-based sorts so that presorted (streaming) grouping will suffice in the middle tier, thereby minimizing the need to materialize intermediate results and providing preferential treatment for stream-based (i.e., pipelined) evaluation strategies due to their benefits. On the runtime side, query plans are constructed and evaluated at runtime based on an XML token iterator model [11]. For performing cross-source joins, ALDSP uses a pipelined, block-oriented, correlated parameter-passing join strategy called PP-k (parameter passing in groups of k join keys). Optimizations like presorting and PP-k joins are essential for ensuring that the first assurance above is provided by ALDSP.

Figure 6 shows the ALDSP query plan viewer displaying the execution plan for the getProfile( ) service call from our customer profile example data service. There are a number of things worth noting about the structure of this plan, which is best analyzed bottom up. This pipelined query execution plan begins by running an outerjoin query to retrieve orders with corresponding customer data, ordered by customer id; this enables the nesting of orders within customers to be done with a pipelineable presorted groupby operation in the ALDSP runtime. These results are then fed into a PP-k join that sends groups of k customer id values down to the second relational data source in order to retrieve the customers' credit card information. This does not perturb the overall way that the data stream is clustered by customer id, so the results are index-nested-loop-joined in memory with the waiting customer/order data and then used to form a call to the credit rating Web service in order to finish assembling the desired customer profile structure for each customer. Note that only k customers' with their orders need to be materialized in memory at a time in this query plan, thereby keeping its runtime memory demands modest. This plan clearly satisfies the first assurance.

Now let us examine Figure 7 in order to see how ALDSP does at providing the other two assurances. This figure shows the ALDSP plan viewer displaying the execution plan for an ad hoc XQuery. The query being examined was written against the getProfile( ) service call whose execution plan (Figure 6) we just finished examining. However, note that the query does not actually use data from the first database's ORDER table or the CREDIT_CARD table of the second database, and it contains a selection predicate on customer id. What we see by comparing this query plan with the previous one is that this query plan is much simpler – despite this query having been written in terms of the preceding query. This plan accesses only the LAST_NAME and SSN columns of CUSTOMER and the predicate on customer id has been pushed to the data source, so it issues only one SQL statement that will end up quickly retrieving two columns from one CUSTOMER row; it will then call the credit rating lookup service with the information required and be finished. This plan clearly satisfies all three assurances, as the layered nature of the scenerio has all been compiled away during query optimization. The resulting query is what a developer would likely have written to achieve the same goal in the absence of any layers of abstraction.

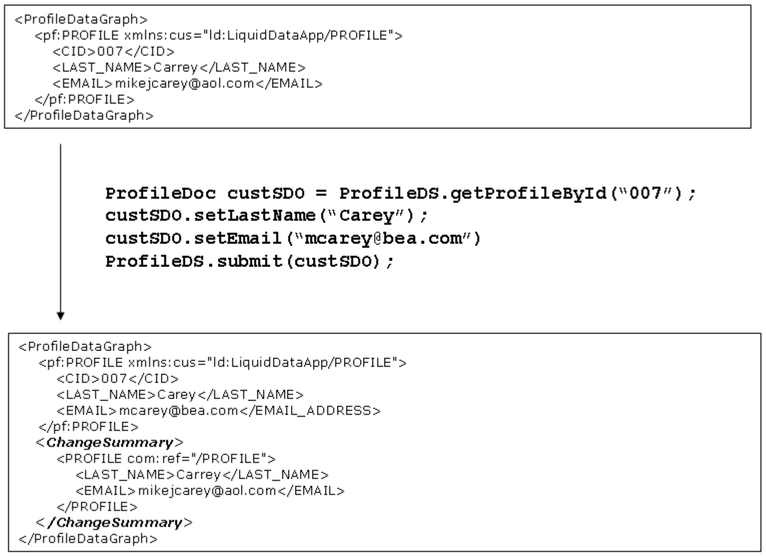
3.3 Advanced Topics

Before we close our coverage on the topic of query processing, we should briefly touch on a few other salient aspects of ALDSP. First of all, like many query processors, ALDSP does caching of both query plans and data. In addition to caching query plans for client-provided queries, however, ALDSP caches partially compiled plans for XQuery functions. Because of the importance of layering and abstraction to make "integrate once and reuse" a reality, our customers expect to be able to make heavy use of layering. Partial plan caching greatly speeds up the compilation of queries that use the functions (views) whose plans have been cached. Second, ALDSP also offers data caching, but it takes a functional approach. Developers and administrators can enable or disable caching at the level of individual data service functions and can configure their favorite RDBMS to serve as the cluster-wide ALDSP function cache. When caching is enabled for a data service function, ALDSP maintains a map of function calls and their parameter values to the results obtained and then uses this map as a lookaside cache to avoid recomputing the result again if it is sufficiently fresh. Consistency is based on the use of an administratively controlled TTL (time-to-live) per cached function. A typical good use of the cache would be to memoize the results of a credit check Web service call for some number of hours or days in order to avoid repeating such a high latency (and possibly expensive) operation repeatedly within a sufficiently short time interval.

ALDSP provides additional support for performance tuning and enhancement via the judicious use of parallelism and carefuly treatment of alternate data sources. ALDSP provides a handful of extra, built-in XQuery functions that provide access to these facilities. One of these, fn-bea:async( ), can be used to hint to the ALDSP query processor that its argument (any XQuery expression) is a good candidate for asynchronous evaluation. These functional hints provide a means for ALDSP queries to overlap potentially large wait-times for access to expensive data sources such as remotely located Web services. The other special functions, fn-bea:failover( ) and fn-bea:timeout( ), allow a data service developer to provide a backup expression to try either in the event of a failure during evaluation of a primary expression (for failover) or in the event of a timeout during evaluation of a primary expression (for timeout). These functions each take primary and backup expressions as arguments, and timeout takes the desired maximum wait-time as an additional parameter. A typical use case for either would be to wrap a call to a potentially slow or failure-prone Web service in order to encode a backup plan. The backup plan might be as simple as returning the empty sequence or as complex as trying a whole series of alternatives before giving up.

4. UPDATE PROCESSING IN ALDSP

ALDSP utilizes Service Data Objects [10], a.k.a. SDO, to support updates as well as reads for data from data services. Three of the four ALDSP APIs (all but JDBC/SQL) allow a client application to invoke a data service, then operate on the results, and finally submit the modified data back to the data service from whence it came in order to persist the changes. Figure 8 provides a simple illustration of how ALDSP's SDO-based Java mediator API looks



```
<ProfileDataGraph>
   <pf:PROFILE xmlns:cus="ld:LiquidDataApp/PROFILE">
      <CID>007</CID>
      <LAST_NAME>Carrey</LAST_NAME>
      <EMAIL>mikejcarey@aol.com</EMAIL>
   </pf:PROFILE>
</ProfileDataGraph>
```

```
ProfileDoc custSDO = ProfileDS.getProfileById("007");
custSDO.setLastName("Carey");
custSDO.setEmail("mcarey@bea.com")
ProfileDS.submit(custSDO);
```

```
<ProfileDataGraph>
   <pf:PROFILE xmlns:cus="ld:LiquidDataApp/PROFILE">
      <CID>007</CID>
      <LAST_NAME>Carey</LAST_NAME>
      <EMAIL>mcarey@bea.com</EMAIL_ADDRESS>
   </pf:PROFILE>
   <ChangeSummary>
      <PROFILE com:ref="/PROFILE">
         <LAST_NAME>Carrey</LAST_NAME>
         <EMAIL>mikejcarey@aol.com</EMAIL>
      </PROFILE>
   </ChangeSummary>
</ProfileDataGraph>
```

Figure 8: Disconnected updates via SDO.

in such a use case, showing the relevant client code snippet in the middle. In this small example, a typed SDO object is obtained from the customer profile data service through a call to ALDSP's Java mediator API. The object is operated on in the client application, leading to changes in several of the underlying element values. The changed SDO is then sent back to ALDSP via the submit call of the same data service. Figure 8 depicts the data service data as it flows each way. In the submit call, the new XML data is sent back along with a serialized change summary that identifies the portions of the data that have been changed and records their previous values. The ALDSP server examines the change summary in order to determine how to propagate the changes back to the underlying sources. Similar to reads, unaffected sources are not involved in an update and unchanged portions of source data are not updated.

To automatically propagate data changes to (just) the relevant backend data sources, ALDSP must be able to identify where the changed data came from in the first place. Basically, its lineage must be determined. ALDSP computes the required lineage by analyzing a designated data service read function (by default the first function, preferably the "get all" function if there is one). Primary key information, query predicates, and query result shapes are used together with the change list to determine which data from which sources are actually affected by a given update.

Figure 9 provides a rough sketch of the update decomposition process in ALDSP. An update enters as a submit call on a data service, such as our customer profile example, and is then decomposed into a set of updates for each affected source. In the figure, only the information that came from the relational sources has changed, so only those sources require updates to be propagated to them. A data service developer also has the option to associate an *update override* with a data service at any level; this allows the developer to specify a Java function to which control is handed when updates occur on its data. These update overrides can either extend or replace ALDSP's default update handling, providing a chance to enforce business rules, insert computed values, or even change the way that the update would have been handled altogether if need be.

When the submit method of a data service is called to submit a changed SDO or set of SDOs back to ALDSP, the atomic unit of update execution is the submit call. In the event that all data sources are relational and can participate in a two-phase commit (XA), the entire submit will be executed as an atomic transaction across the affected sources. Since data is read in one transaction, operated on disconnectedly, and then re-submitted later, ALDSP supports optimistic concurrency options to enable a data service designer to choose how to have the system decide if a given change or set of changes can safely be applied. Supported choices include requiring one of the following to be enforced:

1. All of the values that were *read* must still be the same (at update time) as their original (read time) values.

2. All of the values that were *updated* must still be the same (at update time) as their original (read time) values.

3. A *chosen subset* of the values that were read (such as a timestamp or a version id) must still be the same (at update time) as their original (read time) values.

ALDSP uses this choice in the relational case to condition the SQL updates that it generates (i.e., the "sameness" criteria is expressed as part of the where clause for the update statements that are sent to the underlying sources). If reliable update execution is required over non-XA sources, or updates are propagated to non-relational sources, the data service developer will need to use update overrides appropriately. To handle non-XA execution, an update override can optionally invoke a BEA WebLogic Integration workflow process.

5. RELATED WORK AND SYSTEMS

ALDSP's heritage can be traced back through years of research on distributed databases, heterogeneous distributed databases, federated databases, and multidatabases. However, ALDSP was primarily inspired by watching users of commercial EAI solutions –

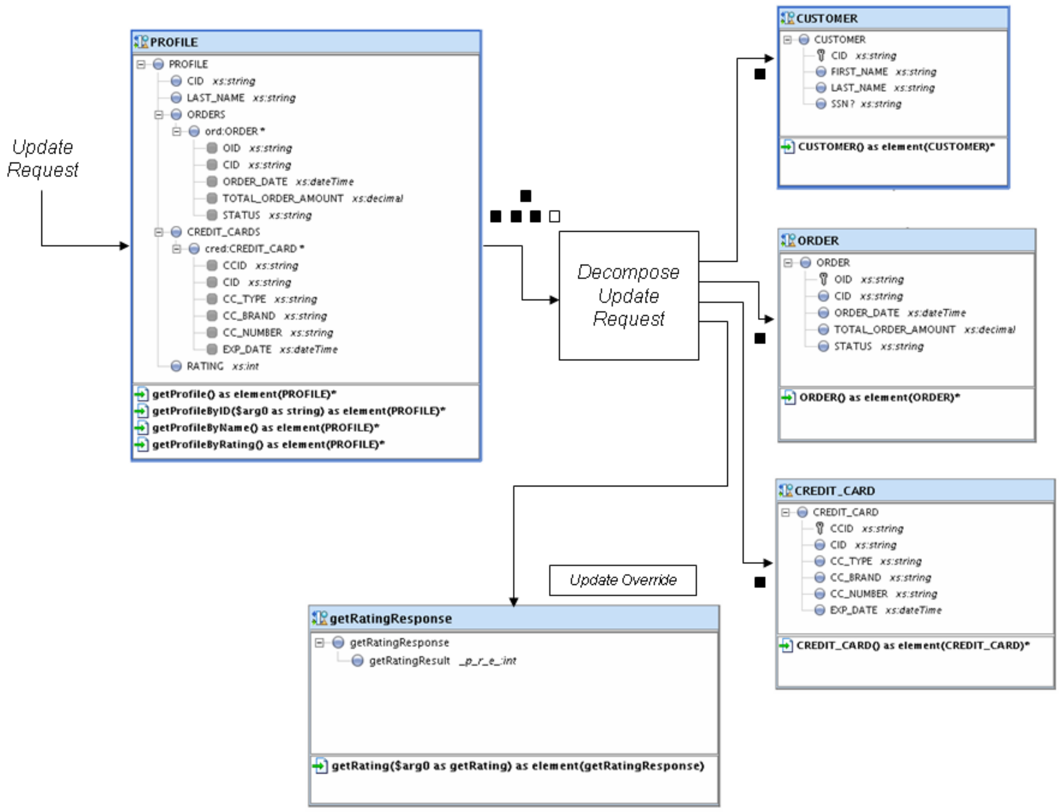


Figure 9: Update decomposition in ALDSP.

mostly of the workflow or process orchestration variety, including BEA's own WebLogic Integration 8.1 product – use those solutions to hand-author distributed query plans that access and combine data from disparate sources and deliver the results to applications that need it, such as portals, in real-time [13].

In terms of the database literature, ALDSP is best understood in relation to the pioneering work on the Functional Data Model [14] and the MultiBase federated database system [15] that first used that model to solve a similar problem for relational and network databases. ALDSP is different in that it applies modern XML technologies to the problem – specifically XML, XML Schema, and the functional query language XQuery – rather than having had to invent and "sell" either a new data model or a new query language. This allows ALDSP to leverage the Web service and SOA trends that are making all sorts of applications functionally accessible through open Web service APIs today. In addition, ALDSP takes a more coarse-grained functional approach, requesting and returning data shapes rather than the individual attribute values of an object, which is a more natural fit for SOA and better-suited both to efficient distributed communications and to exploitation of existing integration-inspired Web service APIs to applications. However, ALDSP was certainly influenced by this early work on the benefits of the "everything is a function" model for integration.

In terms of related commercial systems, there are a number of relationally-based data integration products available today. These include IBM WebSphere Information Integrator, Composite Information Server from Composite Software, and MetaMatrix Enterprise. ALDSP is related to these products because they too are targeted at composing data from multiple sources. However, ALDSP is only loosely related to these products because they each take a relational approach to the data integration problem and then glue a degree of service support on the bottom (to consume data from Web services) as well as on top (in order to publish tagged/nested XML data as a final processing step). The relational plus XML service glue approach is quite awkward for modeling services that return nested XML documents (e.g., the order lookup API of an order management system). Normalizing the results of a Web service into tabular form causes significant usability issues along several dimensions. Moreover, the approach of having XML glue on top of SQL does allow XML results to be returned from a service, but services built in this fashion are neither efficiently composable nor efficiently queryable, unlike ALDSP's data services. Lastly, several other software vendors also offer XQuery-based query engines that permit developers to access and combine XML data from multiple sources, e.g., Ipedo, DataDirect, and Software AG, but none offers a data service modeling approach or supports both reads and updates like ALDSP does.

6. CONCLUSIONS & CURRENT STATUS

In this paper we have provided a technically-oriented overview of the BEA AquaLogic Data Services Platform, or ALDSP for short. We explained the concept of data services and showed how data service designers and developers can use ALDSP to construct data services in a declarative manner. This provides developers of data-centric applications in the SOA world the same sorts of benefits in their world that SQL developers enjoy when building tightly-coupled, centralized database applications. ALDSP's declarative approach allows them to enjoy declarative programming, data independence, and system-provided optimization. We also explained the important requirement of being able to "integrate once and reuse" data services. We examined the query processing implications of this requirement – efficient execution strategies, no-cost layering, and compile-time elimination of unused data and sources from lower layers – and explored how ALDSP is able to meet the requirement through its approach to query processing. We also examined how ALDSP processes updates, including its use of service data objects (SDO) for change tracking and transport, automatic update propagation to relational sources, update overrides for other types of sources, and optimistic concurrency for safe disconnected-mode operation.

As of this writing, in March 2006, BEA has just shipped version 2.1 of ALDSP. Roadmap-wise, we have a number of activities now in process in the AquaLogic Data Services team. In the near-term, we are working on a new version of our JDBC/SQL driver. The current ALDSP 2.1 driver is layered on top of ALDSP's ad hoc XQuery and XML result handling facilities [16], while the next version takes a "bilingual" approach and provides native SQL and result set support. Longer-term, we are planning enhancements in the area of updates to non-relational data sources. This includes support for update processing based on sagas [17], with compensating transactions, and for meta-data driven (as opposed to Java update-override-driven) automation of updates to functional sources such as Web services.

7. ACKNOWLDGEMENTS

A product the size of ALDSP is obviously a major team effort. The current ALDSP team would like to acknowledge the individuals who have contributed as well but since moved on; you know who you are. We would particularly like to single out two unrelated Patels, one (Ajay) who provided air cover for the ALDSP project during difficult times, and the other (Rahul) who had a major influence on the overall vision for ALDSP.

8. REFERENCES

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

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

[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] Michael 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] BEA Systems, Inc. BEA AquaLogic data services platform 2.1. `http://edocs/aldsp/docs21/index.html`.

[8] V. Borkar, M. Carey, D. Lychagin, and T. Westmann. The BEA AquaLogic data services platform (demo). *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, 2006.

[9] XML Query. `http://www.w3.org/XML/Query`, July 2004.

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

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

[13] A. Halevy, N. Ashish, D. Bitton, M. Carey, D. Draper, J. Pollock, and A. Rosenthal. Enterprise information integration: Successes, challenges, and controversies. *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, 2005.

[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 a Multidatabase System*, pages 81–108. Springer-Verlag, New York, 1985.

[16] S. Jigyasu, S. Banerjee, M. Carey, V. Borkar, K. Dixit, A. Malkani, and S. Thatte. SQL to XQuery translation in the AquaLogic data services platform. *Proc. of the 22nd Int'l. Conf. on Data Engineering*, 2006.

[17] H. Garcia-Molina and K. Salem. Sagas. *Proc. of the ACM SIGMOD Int'l. Conf. on Management of Data*, May 1987.