Data Services

Michael J. Carey
University of California, Irvine
2091 Donald Bren Hall (DBH)
Irvine, CA 92697-3435
mjcarey@ics.uci.edu

Nicola Onose
University of California, Irvine
2091 Donald Bren Hall (DBH)
Irvine, CA 92697-3435
onose@ics.uci.edu

Michalis Petropoulos
University of California, San Diego
9500 Gilman Drive
San Diego, CA 92093-0114
mpetropo@gmail.com

ABSTRACT

Data services provide access to data drawn from one or more underlying information sources. Compared to traditional services, data services are model-based, providing richer and more data-centric views of the data they serve; they often provide both query-based and function-based access to data. In the enterprise world, data services play an important role in SOA architectures. They are also important when an enterprise wishes to controllably share data with its business partners via the Internet. In the emerging world of software-as-a-service (SaaS), data services enable enterprises to access their own externally stored information. Last but not least, with the emergence of "cloud data management", a new class of data services is appearing. This paper aims to help readers make sense of these technology trends by reviewing data service concepts and examining approaches to service-enabling individual data sources, creating integrated data services from multiple sources, and providing access to data in the cloud.

1. INTRODUCTION

Today's business practices require access to enterprise data by both external as well as internal applications. Suppliers expose part of their inventory to retailers, and health providers allow patients to view their health records. But how do enterprise data owners make sure that access to their data is appropriately restricted and has predictable impact on their infrastructure? In turn, application developers sitting on the "wrong side" of the Web from their data need a mechanism to find out which data they can access, what their semantics are, and how they can integrate data from multiple enterprises. Data services are software components that address these issues by providing rich metadata, expressive languages, and APIs for service consumers to use to send queries and receive data from service providers.

Data services are a specialization of Web services that can be deployed on top of data stores, other services, and/or applications to encapsulate a wide range of data-centric operations. In contrast to traditional Web services, services that provide access to data need to be *model-driven*, offering a semantically richer view of their underlying data and enabling advanced querying functionality. Data service consumers need access to enhanced metadata, which are both machine-readable and human-readable, such as schema information. This metadata is needed to use or integrate entities returned by different data services. For example, a getCustomers data service might retrieve customer entities, while getOrdersByCID retrieves orders given a customer id. In the absence of any schema information, the consumer is not sure if he/she can compose these two services to retrieve the orders of customers. Moreover, consumers of data services can utilize a query language and pass a query expression as a parameter to a data service; they can use metadata-guided knowledge of data services and their relationships to formulate queries and navigate between sets of entities.

Modern data services are descendants of the stored procedure facilities provided by relational database systems, which allow a set of SQL statements and control logic to be parameterized, named, access-controlled, and then called from applications that wish to have the procedure perform its task(s) without their having to know the internal details of the procedure [42]. The use of data services has many of the same access control and encapsulation motivations. Data owners (database administrators) publish data services because they are able to address security concerns and protect sensitive data, predict the impact of the published data services on their resources, and safeguard and optimize the performance of their servers. In traditional IT settings, it is not uncommon for stored procedures to be the only permitted data access method for these reasons, as letting applications submit arbitrary queries has long been dismissed as too permissive and unpredictable [12]. This is even more of an issue in the world of data services, where providers of data services typically have less knowledge of (and much less control over) their client applications.

We are now moving towards a hosted services world and new ways of managing and exchanging data. The growing importance of data services in this movement is evidenced by the number of contexts within which they have been utilized in recent years: data publishing, data exchange and integration, service-oriented architectures (SOA), data as a service (DaaS), and most recently, cloud computing. Lack of standards, though, has allowed different middleware vendors to develop different models for representing and querying their data services [14]. Consequently, federating databases and integrating data in general may become an even bigger headache. This picture becomes still more complicated if we consider databases living in the cloud [3]. Integrating data across the cloud will be all the more daunting because the models and interfaces there are even more diverse, and also because common assumptions about data locality, consistency, and availability may no longer hold.

The purpose of this article is threefold: i) to explain basic data services architecture(s) and terminology (Section 2), ii) to present three popular contexts in which data services are being deployed today, along with one examplary commercial

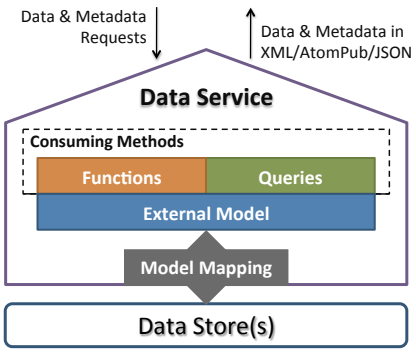

Figure 1: Data Services Architecture

framework targeting each context (Sections 3, 4, and 5), and iii) to discuss certain advanced issues, such as transactions and consistency, pertaining to data services (Section 6). We use a very simple "enterprise-y" e-commerce scenario (involving customer and order data) to illustrate the different data service styles, which will include basic, integrated, and cloud data services. We chose a simple e-commerce scenario because it is intuitive and some variant of it makes sense for each of the styles. We conclude by briefly examining emerging trends and exploring possible directions for future data services research and development (Section 7). Although we do include product-based examples, it is not our intention to recommend products or to survey the product space; the purpose of our examples is concreteness (and reality).

2. DATA SERVICES ARCHITECTURE

A data service can be employed on top of data stores providing different interfaces, such as database servers, CRM applications, or cloud-based storage systems, and using diverse underlying data models, such as relational, XML, or simple key/value pairs. In each case, however, the data and metadata of data services are exposed to their consumers in terms of a common *external model*. Figure 1 presents the architecture of a data service, where the mapping between the model of the data store and the external model of the data service is shown. Users building data services can define the external model and express the mapping, however declarative or procedural it may be, either manually or by using utilities provided by the data service framework to automatically generate the external model and mapping for certain classes of data stores. In the declarative case, these mappings are often similar in many respects to view definitions in relational databases [42]. Note that data services can encapsulate read access and/or update access to the underlying data; we focus largely on read access here for ease of exposition.

Figure 1 also highlights the two prevailing methods for consuming data services: either through *functions* or through *queries*. Functions encapsulate the data and make it accessible only through a set of carefully defined, and possibly application-specific, function signatures. In contrast, queries can be formulated based on the external model using a (possibly restricted) query language. Functions capture the current practice of exporting parameterized stored procedures as data services. When clients utilize queries as consuming methods, various query languages can be used, among them SQL, XPath/XQuery, or proprietary languages

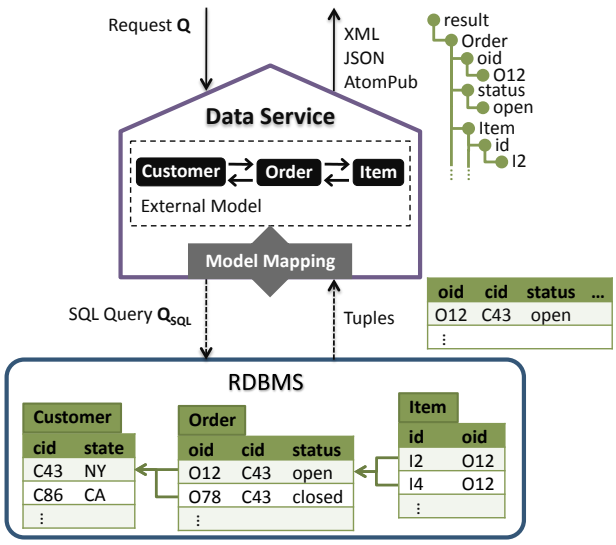

Figure 2: Service-Enabling a Relational Data Store

such as SOQL from Salesforce.com [43] or Microsoft's OData queries [37]. The mapping between the query language of the data store (if any) and the query language utilized by service clients is performed during each service request based on consulting the model mapping of the data service. The information used to guide this mapping is provided at design time by the data service implementer. The syntax in which data service output is returned to clients varies from JSON [31] to XML to AtomPub [28], as shown in Figure 1.

A last point that needs clarification is how the external model is made available to the clients of data services. Apart from their regular data-returning functions and queries, data service frameworks generally provide a set of special functions and queries that return a description of the external model in order to guide clients in consuming a service's interconnected functions and/or queries in a meaningful way.

In contrast, traditional Web services have a much simpler architecture: they are purely functional and their implementations are opaque. Some traditional web services rely on WSDL [19] to describe the provided operations, where the input messages accepted and the output messages expected are defined, but these operations are semantically disconnected from the client's point of view, with each one providing independent functionality. Therefore, the presence of a data model is distinctive to data services.

3. SERVICE-ENABLING DATA STORES

In the basic data service usage scenario, the owners of a data store enable Web clients and other applications to access their otherwise externally inaccessible data by publishing a set of data services, thus *service-enabling* their data store. Microsoft's WCF Data Services framework [35] and Oracle's ODSI [38] are two of a number of commercial frameworks that can be used to achieve this goal. (A survey of such frameworks is beyond the scope of this article.)

To illustrate the key concepts involved in service-enabling a data store, we will use the WCF Data Services framework as an example. We show how the framework can be used to service-enable a relational data store (although in principle any data store can be used.) Figure 2 shows a re-

lational database that stores information about customers, the orders they have placed, and the items comprising these orders. The owners map this internal relational schema to an external model that includes entity types Customer, Order and Item. Foreign key constraints are mapped to navigation properties of these entity types, as shown by the arrows in the external model of Figure 2. Hence, given an Order entity, its Items and its Customer can be accessed.

WCF Data Services implements the Open Data Protocol (OData) [37], which is an effort to standardize the creation of data services. OData uses the Entity Data Model (EDM) as the external model, which is an object-extended E/R model with strong typing, identity, inheritance and support for scalar and complex properties. EDM is part of Microsoft's Entity Framework, an object-relational mapping (ORM) framework [2], which provides a declarative mapping language and utilities to generate EDM models and mappings given a relational database schema.

Clients consuming OData services, such as those produced by the WCF Data Services framework, can retrieve a Service Metadata Document [37] describing the underlying EDM model by issuing the following request that will return the entity types and their relationships:

```
http://<service_uri>/SalesDS.svc/$metadata
```

OData services provide a uniform, URI-based querying interface and map CRUD (Create, Retrieve, Update, and Delete) operations to standard HTTP verbs [22]. Specifically, to create, read, update or delete an entity, the client needs to submit an HTTP POST, GET, PUT or DELETE request, respectively. For example, the following HTTP GET request will retrieve the *open* Orders and Items for the Customer with cid equal to C43:

```
http://<service_uri>/SalesDS.svc/
    Customers('C43')/Orders?
    $filter=status eq 'open'&
    $expand=Items
```

The second line of the above request selects the Customer entity with key C43 and navigates to its Order entities. The $filter construct on the third line selects only the 'open' Order entities, and the $expand on the last line navigates and inlines the related Item entities within each Order. Other query constructs, such as $select, $orderby, $top and $skip, are also supported as part of a request.

To provide interested readers with a bit more detail, a graphical representation of the external model (EDM) together with examples of the OData data service metadata and an AtomPub response from an OData service call are available online in Appendix A.

Driven by the mapping between the EDM model and the underlying data store's relational model, the WCF Data Services framework automatically translates the above request into the query language of the store. In our example then, under the covers, the above request would be translated into the following SQL query involving a left outer join:

```
SELECT Order.*, Item.*
FROM Customer JOIN Order LEFT OUTER JOIN Item
WHERE Customer.cid='C43' AND Order.status='open'
```

The WCF Data Services framework can also publish stored procedures as part of an OData service, called *Function Imports*, which can be used for any of the CRUD operations and are integrated in the same URI-based query interface.

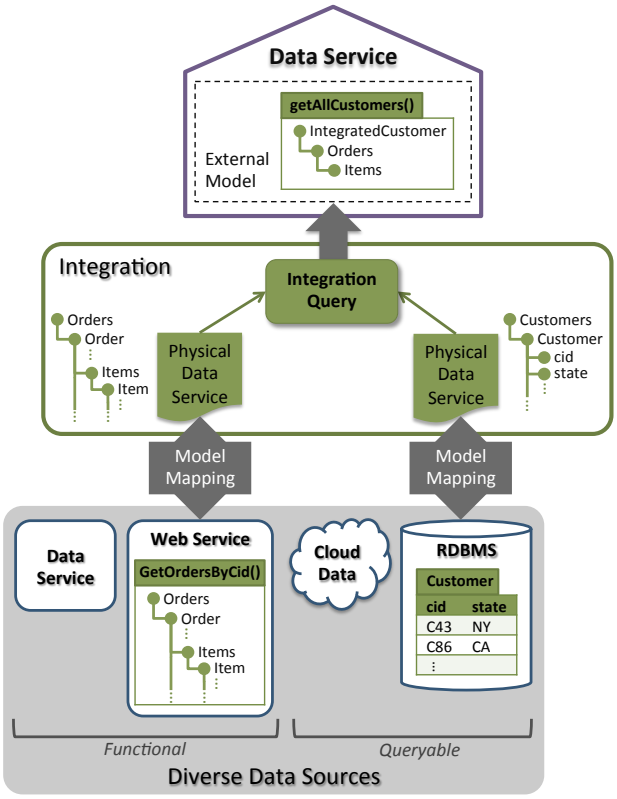

Figure 3: Integrating and Service-Enabling Enterprise Data

4. INTEGRATED DATA SERVICES

In the second data service usage scenario, data services are created to integrate as well as to service-enable a *collection of data sources*. The underlying sources are often heterogeneous in nature. The result of integration is that consumers of a data service see what appears to be one coherent, service-enabled data source rather than being faced with a hodge-podge of disparate schemas and data access APIs.

Figure 3 shows an example usage scenario. We will illustrate this use case using a commercial data services middleware platform, the AquaLogic Data Services Platform, ALDSP [13], developed at BEA Systems and later rebranded as ODSI, the Oracle Data Services Integrator [38]. ODSI is based on a functional model [44] and the functional data service consuming method of Figure 1. (Again, we use ODSI as just one example of the products in this space.)

At the bottom of Figure 3 we see a variety of data sources. These can include queryable data sources, such as relational databases and perhaps data stored and managed in the cloud (see Section 5), as well as functional data sources like traditional Web services and other pre-existing data services. For each type of data source, the data services platform provides a default mapping that gives the data service architect a view of that data source in terms of a common data and programming model. In ODSI, what the data service architect sees initially is a set of *physical data services* that are modeled as groups of functions that each take zero or more XML arguments and return XML results. For instance, a relational table is modeled as a data service that has read, cre-

ate, update, and delete functions; the read function returns a stream of XML elements, one per row from the underlying table, if used in a query without additional filter predicates. A Web service-based data source is modeled as a function whose inputs and outputs correspond to the schema information found in the service's WSDL description.

For a given set of physical data services, the task of the data service architect is then to construct an integrated model for consumption by client application developers, SOA applications, and/or end users. In the example shown in Figure 3, data from two different sources, a Web service and a relational database, are being combined to yield a *single view of customer* data service that hides its integration details from its consumers. In ODSI, the required merging and mapping of lower-level data services to create higher-level services is achieved by function composition using the XQuery language [9]. (This is very much like defining and layering views in a relational DBMS setting.) Because data service architects often prefer graphical tools to hand writing queries, ODSI provides a graphical query editor to support this activity; Appendix B describes its usage for this use case in a bit more detail for the interested reader.

Stepping back, Figure 3 depicts a *declarative* approach to building data services that integrate and service-enable disparate data sources. The external model is a functional model based on XQuery functions. The approach is declarative because the integration logic is specified in a high-level language – the integration query is written in XQuery in the case of ODSI. Because of this approach, suppose the resulting function is subsequently called from a query such as the following, which could either come from an application or from another data service defined on top of this one:

```
for $cust in ics:getAllCustomers( )
where $cust/State = 'Rhode Island'
return $cust/Name
```

In this case, the data services platform can *see through* the function definition and optimize the query's execution by fetching only Rhode Island customers from the relational data source and retrieving only the orders for those customers from the order management service to compute the answer. This would not be possible if the integration logic was instead encoded in a procedural language like Java or a business process language like BPEL [32]. Moreover, notice that the query doesn't request all data for customers; instead, it only asks for their names. Because of this, another optimization is possible: The engine can answer the query by fetching only the names of the Rhode Island customers from the relational source and altogether avoid any order management system calls. Again, this optimization is possible because the data integration logic has been declaratively specified. Finally, it is important to note that such function definitions and query optimizations can be composed and later decomposed (respectively) through arbitrary layers of specifications. This is attractive because it makes an incremental (piecewise) data integration methodology possible, as well as allowing for the creation of specialized data services for different consumers, without implying runtime penalties due to such layering when actually answering queries.

In addition to exemplifying the integration side of data services, ODSI includes some interesting advanced data service modeling features. ODSI supports the creation and publishing of collections of interrelated data services (*a.k.a.* dataspaces) [13]. Metadata about collections of published data services is made available in ODSI through catalog data services. Also, in addition to supporting method calls from clients, ODSI provides an optional generic query interface that authorized clients can use to submit ad hoc queries that range over the results of calls to one or more data service methods. Thus, ODSI offers a hybrid of the two data service consuming methods discussed in Section 2.

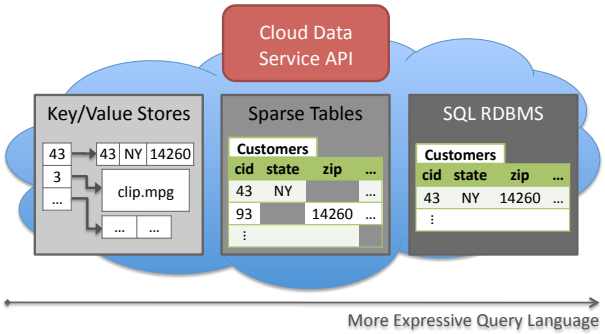

Figure 4: Data Services and Cloud Storage

To aid application developers, methods in ODSI are classified by purpose. ODSI supports read, create, update, and delete methods for operating on data instances. It also supports relationship methods to navigate from one object instance (e.g., a customer) to related object instances (e.g., complaints). ODSI methods are characterized as being either functions (side-effect free) or procedures (potentially side-effecting). Finally, a given method's visibility can be designated as being one of: accessible by outside applications, useable only from within other data services, or private to one particular data service.

5. CLOUD DATA SERVICES

Previous sections described how an enterprise data source or an integrated set of data sources can be made available as services. In this section we focus on a new class of data services designed for providing data management in the cloud.

The cloud is quickly becoming a new universal platform for data storage and management. By storing data in the cloud, application developers can enjoy a pay-as-you-go charging model and delegate most administrative tasks to the cloud infrastructure, which in turn guarantees availability and near-infinite scalability. As depicted in Figure 4, cloud data services today offer various external data models and consuming methods, ranging from *key-value stores* to *sparse tables* and all the way to *RDBMSs in the cloud*. In terms of consuming methods (see Figure 1), key-value stores offer a simple function-based interface, while sparse tables are accessed via queries. RDBMSs allow both a functional interface, if data access is provided via stored procedures only, and a query-based interface, if the database may be queried directly. A recent and detailed survey on cloud storage technologies [16] proposes a similar classification of cloud data services, but further differentiates sparse tables into *document stores* and *extensible record stores*. With respect to the architecture in Figure 1, some of these services forgo the model mapping layer, choosing instead to directly expose their underlying model to consuming applications.

To illustrate the various types of cloud data services, we will briefly examine the Amazon Web Services (AWS) [1]

platform, as it has arguably pioneered the cloud data management effort. Other IT companies are also switching to building cloud data management frameworks, either for internal applications (e.g., Yahoo!'s PNUTS [20]) or to offer as publicly available data services (e.g., Microsoft's WCF data services, as made available in Windows Azure [34]).

Key-Value Stores: The simplest kind of data storage services are *key-value stores* that offer atomic CRUD operations for manipulating serialized data structures (objects, files, etc.) that are identifiable by a key.

An example of a key-value store is Amazon S3 [1]. S3 provides storage support for variable size data blocks, called *objects*, uniquely identified by (developer assigned) keys. Data blocks reside in *buckets*, which can list their content and are also the unit of access control. Buckets are treated as subdomains of s3.amazonaws.com. (For instance, the object customer01.dat in the bucket custorder can be accessed as http://custorder.s3.amazonaws.com/customer01.dat.)

The most common operations in S3 are:

- create (and name) a bucket,
- write an object, by specifying its key, and optionally an access control list for that object,
- read an object,
- delete an object,
- list the keys contained in one of the buckets.

Data blocks in S3 were designed to hold large objects (e.g. multimedia files), but they can potentially be used as database pages storing (several) records. Brantner et al [11] started from this observation and analyzed the trade-offs in building a database system over S3. However, recent industrial trends are favoring the incorporation of more DBMS functionality directly into the cloud infrastructure, thus offering a higher level interface to the application.

Dynamo [21] is another well-known Amazon example of a key-value store. It differs in granularity from S3 since it stores only objects of a relatively small size (< 1 MB), whereas data blocks in S3 may go up to 5 GB.

Sparse Tables: Sparse Tables are a new paradigm of storage management for structured and semi-structured data that has emerged in recent years, especially after the interest generated by Google's Bigtable [18]. (Bigtable is the storage system behind many of Google's applications and is exposed, via APIs, to Google App Engine [25] developers.) A sparse table is a collection of data records, each one having a row and a set of column identifiers, so that at the logical level records behave like the rows of a table. There may be little or no overlap between the columns used in different rows, hence the "sparsity". The ability to address data based on (potentially many) columns differentiates sparse tables from key/value stores and makes it possible to index and query data more meaningfully. Compared to traditional RDBMSs that require static (fixed) schemas, sparse tables have a more flexible data model, since the set of columns identifiers may change based on data updates. Bigtable has inspired the creation of similar open source systems such as HBase [29] and Cassandra [15].

SimpleDB [1] is Amazon's version of a sparse table and exposes a Web service interface for basic indexing and querying in the cloud. A column value in a SimpleDB table may be atomic, as in the relational model, or a list of atomic values (limited in size to 1KB). SimpleDB's tables are called *domains*. SimpleDB queries have a SQL-like syntax and can perform selections, projections and sorting over domains. There is no support for joins or nested subqueries.

A SimpleDB application stores its customer information in a domain called Customers and its order information in an Orders domain. Using SimpleDB's REST[1] interface, the application can insert records (id='C043', state='NY') into Customers and (id='O012', cid='C043', status='open') into Orders. Further inserts do not necessarily need to conform to these schemas, but for the sake of our example we will assume that they do.

Since SimpleDB does not implement joins, joins must be coded at the client application level. For example, to retrieve the orders for all NY clients, an application would first fetch the client info via the query:

```
select id from Customers where state='NY'
```

the result of which would include C043 and would then retrieve the corresponding orders as follows:

```
select * from Orders where cid='C043'
```

A major limitation for SimpleDB is that the size of a table instance is bounded. An application that manipulates a large volume of data needs to manually partition ("shard") it and issue separate queries against each of the partitions.

RDBMSs: In cloud computing systems that provide a virtual machine interface, such as EC2 [1], users can install an entire database system in the cloud. However, there is also a push towards providing a database management system itself as a service. In that case, administrative tasks such as installing and updating DBMS software and performing backups are delegated to the cloud service provider.

Amazon RDS [1] is a cloud data service that provides access to the full capabilities of a MySQL [39] database installed on a machine in the cloud, with the possibility of setting several "read replicas" for read-intensive workloads. Users can create new databases from scratch or migrate their existent MySQL data into the Amazon cloud. Microsoft has a similar offering with SQL Azure [7], but chooses a different strategy that supports scaling by physically partitioning and replicating logical database instances on several machines. A SQL Azure source can be service-enabled by publishing an OData service on top of it, as in Section 3. Google's Megastore [5] is also designed to provide scalable and reliable storage for cloud applications, while allowing users to model their data in a SQL-like schema language. Data types can be string, numeric types, or Protocol Buffers [26] and they can be required, optional or repeated.

Amazon RDS users manage and interact with their databases either via shell scripts or a SOAP[2]-based Web services API. In both cases, in order to connect to a MySQL instance, users need to know its DNS name, which is a subdomain of rds.amazonaws.com. They can then either open a MySQL console using an Amazon-provided shell script, or they can access the database like any MySQL instance identified by a DNS name and port.

6. ADVANCED TECHNICAL ISSUES

[1]Representational State Transfer [22]
[2]Simple Object Access Protocol [36]

So far we have mostly covered the basics of data services, touching on a range of use cases (single source, integrated source, and cloud sources) along with their associated data service technologies. In this section we will briefly highlight a few more advanced topics and issues, including updates and transactions, data consistency for scalable services, and issues related to security for data services.

Data Service Updates and Transactions: As with other applications, applications built over data services require transactional properties in order to operate correctly in the presence of concurrent operations, exceptions, and service failures. Data services based on single sources, for the most part, can inherit their answer to this requirement from the source that they serve their data from. Data services that integrate data from multiple sources, however, face additional challenges – especially since many interesting data sources, such as enterprise Web services and cloud data services, are either unable or "unwilling" to participate in traditional (two-phase commit-based) distributed transactions due to issues related to high latencies and/or temporary loss of autonomy. Data service update operations that involve non-transactional sources can potentially be supported using a compensation-based transaction model [8] based on Sagas [23]. The classic compensating transaction example is travel-related, where a booking transaction might need to perform updates against multiple autonomous ticketing services (to obtain airline, hotel, rental car, and concert reservations) and roll them all back via compensation in the event that reservations cannot be obtained from all of them. Unfortunately, such support is under-developed in current data service offerings, so this is an area where all current systems fall short and further refinement is required. The current state of the art leaves too much to the application developer in terms of hand-coding compensation logic as well as picking up the pieces after non-atomic failures.

Another challenge, faced both by single-source and multi-source data services, is the mapping of updates made to the external model to correspondingly required updates to the underlying data source(s). This challenge arises because data services that involve non-trivial mappings – as might be built using the tools provided by WCF or ODSI – present the service consumer with an interface that differs from those of the underlying sources. The data service may restructure the format of the data, it may restrict what parts of the data are visible to users, and/or it may integrate data coming from several back-end data sources. Propagating data service updates to the appropriate source(s) can be handled for some of the common cases by analyzing the lineage of the published data, i.e., computing the inverse mapping from the service view back to the underlying data sources based on the service view definition [2, 8]. In some cases this is not possible, either due to issues similar to non-updatibility of relational views [6, 33] or due to the presence of opaque functional data sources such as Web service calls, in which case hints or manual coding would be required for a data services platform to know how to back-map any relevant data changes.

Data Consistency in the Cloud: To provide scalability and availability guarantees when running over large clusters, cloud data services have generally adopted lower consistency models. This choice is defended in [30], which states that in large scale distributed applications the scope of an atomic update needs to be—and generally is—within an abstraction called *entity*. An entity is a collection of data with a unique key that lives on one machine, such as a data record in Dynamo. According to [30], developers of truly scalable applications have no real choice but to cope with the lack of transactional guarantees across machines and with repeated messages sent between entities. In practice, there are several consistency models that share this philosophy.

The simplest model is *eventual consistency* [46], first defined in [45] and used in Amazon Dynamo [21], which only guarantees that all updates will reach all replicas eventually. There are some challenges with this approach, mainly because Dynamo uses replication to increase availability. In the case of network or server failures, concurrent updates may lead to update conflicts. To allow for more flexibility, Dynamo pushes conflict resolution to the application. Other systems try to simplify developers' lives and resolve conflicts inside the data store, based on simple policies: e.g., in S3 the write with the latest timestamp wins.

PNUTS, Yahoo's sparse table store, goes one step beyond eventual consistency by also guaranteeing that all replicas of a given record apply all updates in the same order. Such stronger guarantees, called *timeline consistency*, may be necessary for applications that manipulate user data, e.g., if a user changes access control rights for a published data collection and then adds sensitive information. Amazon SimpleDB has recently added support for similar guarantees, which they call *consistent reads*, as well as for *conditional updates*, which are executed only if the current value of an attribute of a record has a specified expected value. Conditional update primitives make it easier to implement solutions for common use cases such as global counters or optimistic concurrency control (by maintaining a timestamp attribute).

Finally, RDBMSs in the cloud (Megastore, SQL Azure) provide ACID semantics under the restriction that a transaction may touch only one entity. This is ensured by requiring all tables involved in a transaction to share the same partitioning key. In addition, Megastore provides support for transactional messaging between entities via queues and for explicit two-phase commit.

Data Services Security: A key aspect of data services that is underdeveloped in current product and service offerings, yet extremely important, is data security. Web service security alone is not sufficient, as control over who can invoke which service calls is just one aspect of the problem for data services. Given a collection of data services, and the data over which they are built, a data service architect needs to be able to define access control policies that govern which users can do and/or see what and from which data services. As an example, an interesting aspect of ODSI is support for fine-grained control over the information disclosed by data service calls – where the same service call, depending on "who's asking", can return more or less information to the caller. Portions of the information returned by a data service call can be encrypted, substituted, or altogether elided (schema permitting) from the call's results [10]. More broadly, much work has been done in the areas of access control, security, and privacy for databases, and much of it applies to data services. These topics are simply too large [4] to cover in the scope of this article.

7. EMERGING TRENDS

In this article we have taken a broad look at work in the area of data services. We looked first at the enterprise, where we saw how data services can provide a data-oriented encapsulation of data as services in enterprise IT settings. We examined concepts, issues, and example products related to service-enabling single data sources as well as related to the creation of services that provide an integrated, service-oriented view of data drawn from multiple enterprise data sources. Given that clouds are rapidly forming on the IT horizon, both for Web companies and for traditional enterprises, we also looked at the emerging classes of data services that are being offered for data management in the cloud. As the latter mature, we expect to see a convergence of everything that we have looked at, as it seems likely that rich data services of the future will often be fronting data residing in one or more data sources in the cloud.

To wrap up this article, we briefly list a handful of emerging trends that can possibly direct future data services research and development. Some of the trends listed stem from already existing problems, while others are more predictive in nature. We chose this list, which is necessarily incomplete, based on the evolution of data services that we have witnessed while slowly authoring this report over the two last years. Again, while data services were initially conceived to solve problems in the enterprise world, the cloud is now making data services accessible to a much broader range of consumers; new issues will surely arise as a result.

Query Formulation Tools: Service-enabled data sources sometimes support (or permit) only a restricted set of queries against their schemas. Users trying to formulate a query over multiple such sources can have a hard time figuring out how to compose such data services. For a schema-based external model, recent work proposed tools to help users author only answerable queries, e.g., CLIDE [41]. These tools utilize the schemas and restrictions of the service-enabled data sources as a basis for query formulation, rather than just the externally visible data service metadata, to guide the users towards formulating answerable queries. More work is needed here to handle broader classes of queries.

Data Service Query Optimization: In the case of integrated data services with a functional external model, one could imagine defining a set of semantic equivalence rules that would allow a query processor to substitute a data service call used in a query for another service call in order to optimize the query execution time, thus enabling semantic data service optimization. For example, the following equivalence rule captures the semantic equivalence of the data services getOrderHistory and getOpenOrders when a [status = 'open'] condition is applied to the former:

getOrderHistory(cid) [status = 'open'] ≡ getOpenOrders(cid)

Work is needed here to help data service architects to specify such rules and their associated tradeoffs "easily" and to teach query optimizers to exploit them.

Very Large Functional Models: For data services using a functional external model, if the number of functions is very large, it is hard or even impossible for the data owner to explicitly enumerate all functions and for the query developer to have a global picture of them. Consider the example of a data owner, who, for performance reasons, only wants to allow queries that use some non-empty subset of a set of n filter predicates. Enumerating all the $2^n - 1$ combinations as functions would be tedious and impractical for large n. Recent work has studied how models consisting of such large collections of functions, where the function bodies are defined by XPath queries, can be compactly specified using a grammar-like formalism [40] and how queries over the output schema of such a service can be answered using the model [17]. More work is needed here to extend the formalism and the query answering algorithms to larger classes of queries and to support functions that perform updates.

Cloud Data Service Integration: Since consumers and small businesses are starting to store their data in the cloud, it makes sense to think about data service integration in the cloud when there are many, many small data services available. For example, how can Clark Cloudlover integrate his Google calendar with his wife's Apple iCal calendar? At this point, there is no homogeneity in data representation and querying, and the number of cloud service providers is rapidly increasing. Google Fusion Tables [27] is one example of a system that follows this trend and allows its users to upload tabular data sets (spreadsheets), to store them in the cloud, and subsequently to integrate and query them. Users are able to integrate their data with other publicly available datasets by performing left outer joins on primary keys, called table merges. Fusion Tables also visualize users' data using maps, graphs and other techniques. Work is needed on many aspects of cloud data sharing and integration.

Data Summaries: As the number of data services increases to a "consumer scale", it will be hard even to find the data services of interest and to differentiate among data services whose output schemas are similar. One approach to easing this problem is to offer *data summaries* that can be searched and that can give data service consumers an idea of what lies behind a given data service. Data sampling and summarization techniques that have been traditionally employed for query optimization can serve as a basis for work on large-scale data service characterization and discovery.

Cloud Data Service Security: Storing proprietary or confidential data in the cloud obviously creates new security problems. Currently, there are two broad choices. Data owners can either encrypt their data, but this means that all but exact-match queries have to be processed on the client, moving large volumes of data across the cloud, or they have to trust cloud providers with their data, hoping that there are enough security mechanisms in the cloud to guard against malicious applications and services that might try to access data that does not belong to them. There is early ongoing work [24] that may help to bridge this gap by enabling queries and updates over encrypted data, but much more work is needed to see if practical (e.g., efficient) approaches and techniques can indeed be developed.

8. ACKNOWLEDGMENTS

We would to thank Divyakant Agrawal (UC Santa Barbara), Pablo Castro (Microsoft), Alon Halevy (Google), James Hamilton (Amazon) and Joshua Spiegel (Oracle) for their detailed comments and the time they devoted to carefully read an earlier version of this article. We would also like to thank the handling editor and the anonymous reviewers for

feedback that improved the quality of this article.

This work was supported in part by NSF IIS awards 0910989, 0713672 and 1018961.

9. REFERENCES

[1] Amazon Web Services, 2010. http://aws.amazon.com/.

[2] A. Adya, J. A. Blakeley, S. Melnik, and S. Muralidhar. Anatomy of the ADO.NET entity framework. In *SIGMOD Conference*, pages 877–888, 2007.

[3] D. Agrawal, A. E. Abbadi, S. Antony, and S. Das. Data Management Challenges in Cloud Computing Infrastructures. In *DNIS*, pages 1–10, 2010.

[4] Allen A. Friedman and Darrell M. West. Privacy and Security in Cloud Computing. *Issues in Technology Innovation*, 3, October 2010. Center for Technology Innovation at Brookings.

[5] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Léon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *CIDR Conference*, 2011.

[6] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6:557–575, December 1981.

[7] P. A. Bernstein, I. Cseri, N. Dani, N. Ellis, A. Kalhan, G. Kakivaya, D. B. Lomet, R. Manne, L. Novik, and T. Talius. Adapting Microsoft SQL Server for cloud computing. In *ICDE*, pages 1255–1263, 2011.

[8] M. Blow, V. Borkar, M. Carey, C. Hillery, A. Kotopoulis, D. Lychagin, R. Preotiuc-Pietro, P. Reveliotis, J. Spiegel, and T. Westmann. Updates in the AquaLogic Data Services Platform. In *IEEE International Conference on Data Engineering (ICDE)*, pages 1431–1442, 2009.

[9] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. W3C Recommendation 23 January 2007, 2007. http://www.w3.org/TR/xquery/.

[10] V. R. Borkar, M. J. Carey, D. Engovatov, D. Lychagin, P. Reveliotis, J. Spiegel, S. Thatte, and T. Westmann. Access Control in the AquaLogic Data Services Platform. In *SIGMOD Conference*, pages 939–946, 2009.

[11] M. Brantner, D. Florescu, D. A. Graf, D. Kossmann, and T. Kraska. Building a database on S3. In *SIGMOD Conference*, pages 251–264, 2008.

[12] Britton-Lee Inc. *IDM 500 Software Reference Manual Version 1.3*, 1981.

[13] M. J. Carey. Data Delivery in a Service-Oriented World: The BEA AquaLogic Data Services Platform. In *SIGMOD Conference*, pages 695–705, 2006.

[14] M. J. Carey. SOA What? *IEEE Computer*, 41(3):92–94, 2008.

[15] The Apache Cassandra Project, 2009. http://cassandra.apache.org/.

[16] R. Cattell. Scalable SQL and NoSQL data stores. *SIGMOD Record*, 39(4):12–27, 2010.

[17] B. Cautis, A. Deutsch, N. Onose, and V. Vassalos. Efficient rewriting of XPath queries using query set specifications. *PVLDB*, 2(1):301–312, 2009.

[18] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 15–15, 2006.

[19] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. W3C Note 15 March 2001, 2001. http://www.w3.org/TR/wsdl.

[20] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, 2008.

[21] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[22] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[23] H. Garcia-Molina and K. Salem. Sagas. *SIGMOD Rec.*, 16(3):249–259, 1987.

[24] C. Gentry. Computing arbitrary functions of encrypted data. *Commun. ACM*, 53(3):97–105, 2010.

[25] Google. Google App Engine. http://code.google.com/appengine.

[26] Google. Protocol Buffers: Google's data interchange format, 2008. http://code.google.com/p/protobuf.

[27] Google. Fusion Tables, 2009. http://tables.googlelabs.com.

[28] J. Gregorio and B. de hÓra. The Atom Publishing Protocol, 2005. http://datatracker.ietf.org/doc/rfc5023/.

[29] HBase, 2010. http://hbase.apache.org/.

[30] P. Helland. Life Beyond Distributed Transactions, an Apostate's Opinion. In *Conference on Innovative Data System Research (CIDR)*, 2007.

[31] JavaScript Object Notation. http://www.json.org/.

[32] D. Jordan and J. Evdemon. Web Services Business Process Execution Language Version 2.0. OASIS Standard, 11 April 2007, 2007. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html.

[33] Y. Kotidis, D. Srivastava, and Y. Velegrakis. Updates through views: A new hope. In *ICDE*, page 2, 2006.

[34] Microsoft, Inc. Windows Azure, 2009. http://www.microsoft.com/windowsazure/windowsazure/.

[35] Microsoft, Inc. WCF Data Services, 2011. http://msdn.microsoft.com/en-us/data/bb931106.

[36] N. Mitra and Y. Lafon. SOAP Version 1.2 Part 0: Primer (Second Edition). W3C Recommendation 27 April 2007, 2007. http://www.w3.org/TR/soap/.

[37] OData. Open Data Protocol, 2010. http://www.odata.org/.

[38] Oracle and/or its affiliates. Oracle Data Service Integrator 10gR3 (10.3), 2011. http://download.oracle.com/docs/cd/E13162_01/odsi/docs10gr3.

[39] Oracle Corporation and/or its affiliates. MySQL. http://www.mysql.com/.

[40] M. Petropoulos, A. Deutsch, and Y. Papakonstantinou. The Query Set Specification Language (QSSL). In *WebDB*, pages 99–104, 2003.

[41] M. Petropoulos, A. Deutsch, Y. Papakonstantinou, and Y. Katsis. Exporting and interactively querying Web service-accessed sources: The CLIDE System. *ACM Trans. Database Syst.*, 32(4), 2007.

[42] R. Ramakrishnan and J. Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[43] Salesforce.com, Inc. Salesforce.com Object Query Language (SOQL), 2010. http://www.salesforce.com/us/developer/docs/api/Content/sforce_api_calls_soql.htm.

[44] D. W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Trans. Database Syst.*, 6(1):140–173, 1981.

[45] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *SOSP*, pages 172–182, 1995.

[46] W. Vogels. Eventually consistent. *ACM Queue*, 6(6):14–19, 2008.

APPENDIX

A. SERVICE-ENABLING DATA SOURCES

In this Appendix we provide additional information related to the WCF Data Services example in order to make the example more concrete. Figure 5 shows a graphical representation of the EDM model for the relational database in Figure 2, as displayed to a WCF Data Service developer, along with the declarative mapping details for the Order entity at the bottom. The navigation properties are displayed at the bottom of each entity type, and the connecting lines denote the property associations between the entity types that facilitate navigation. Clients consuming WCF Data Services can retrieve metadata describing the underlying EDM model by issuing a request to return a service's entity types and their relationships, e.g.:

http://<service_uri>/SalesDS.svc/$metadata

Metadata is returned in the Conceptual Schema Definition Language (CSDL), an example of which is shown in Figure 6.

In terms of data service results in WCF, Figure 7 shows the results returned for the example query of Section 3, which requested a list of the open orders for Customer C43. The tuples retrieved by the underlying SQL query are subsequently transformed to the AtomPub result format used by OData services, as shown in the figure. Note that the query result contains link elements that allow the client to form navigation requests to related entities.

B. INTEGRATED DATA SERVICES

In this Appendix we dive a bit deeper into the ODSI example to more fully convey ODSI's approach to data service integration. In particular, Figure 8 presents a screen shot that shows how the ODSI graphical query editor would be used in the example scenario when the customer data source is a relational customer table and the orders and line items are coming from an order management Web service that takes a customer id as input and returns the orders for that customer as output.

Let us take a closer look at Figure 8. The upper left box in the query editor screen shot represents the Customer physical data service, which returns elements containing the CID, NAME, and STATE information for customers. The middle box represents the Web service call GetOrdersByCid( ), which takes an XML input containing the cid of a customer and returns an XML result containing the orders (including the nested line items) for this customer. The artifact being constructed is itself an XQuery query that will be the body of a callable data service function – a function called getAllCustomers( ) – and the box on the far right of the screen shot represents the XML result structure for the function. The rest of the screen shot is a graphical specification of how the data from the two physical data services is to be mapped and combined. The resulting getAllCustomers( ) data service function can then be made available as a callable Web service by simply instructing ODSI to do so through a few additional mouse clicks.

More information about ODSI's approach to data integration and its data service authoring tools may be found in [13, 38].

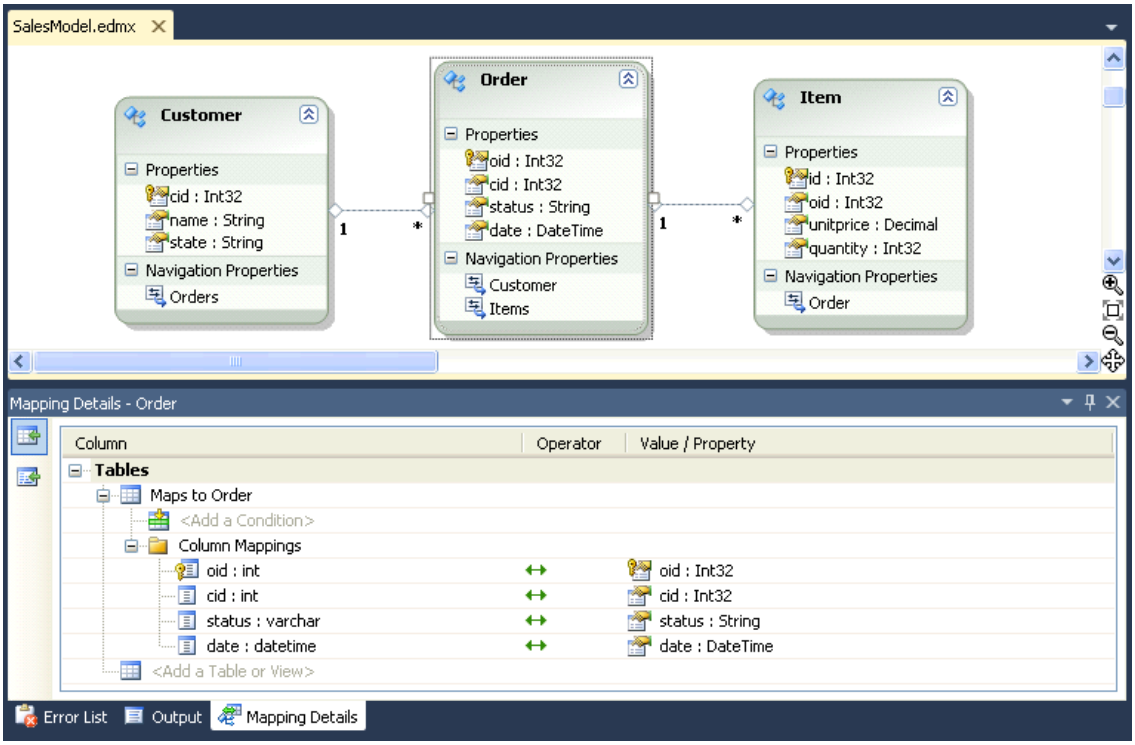

Figure 5: An Entity Data Model

```
- <EntityType Name="Customer">                                    Entity
  - <Key>                                                         Metadata
      <PropertyRef Name="cid" />
    </Key>
    <Property Name="cid" Type="Edm.Int32" Nullable="false" />
    <Property Name="name" Type="Edm.String" Nullable="false" MaxLength="50" Unicode="false" FixedLength="false" />
    <Property Name="state" Type="Edm.String" Nullable="false" MaxLength="2" Unicode="false" FixedLength="true" />
    <NavigationProperty Name="Orders" Relationship="SalesModel.FK_Order_Customer" FromRole="Customer"
      ToRole="Order" />
  </EntityType>
+ <EntityType Name="Item">
+ <EntityType Name="Order">
- <Association Name="FK_Order_Customer">
    <End Role="Customer" Type="SalesModel.Customer" Multiplicity="1" />
    <End Role="Order" Type="SalesModel.Order" Multiplicity="*" />
  - <ReferentialConstraint>
    - <Principal Role="Customer">
        <PropertyRef Name="cid" />
      </Principal>
    - <Dependent Role="Order">
        <PropertyRef Name="cid" />
      </Dependent>                                                Association
    </ReferentialConstraint>                                      Metadata
  </Association>
+ <Association Name="FK_Item_Order">
```

Figure 6: An OData Service Metadata Document

Figure 7: An OData Service Response Document

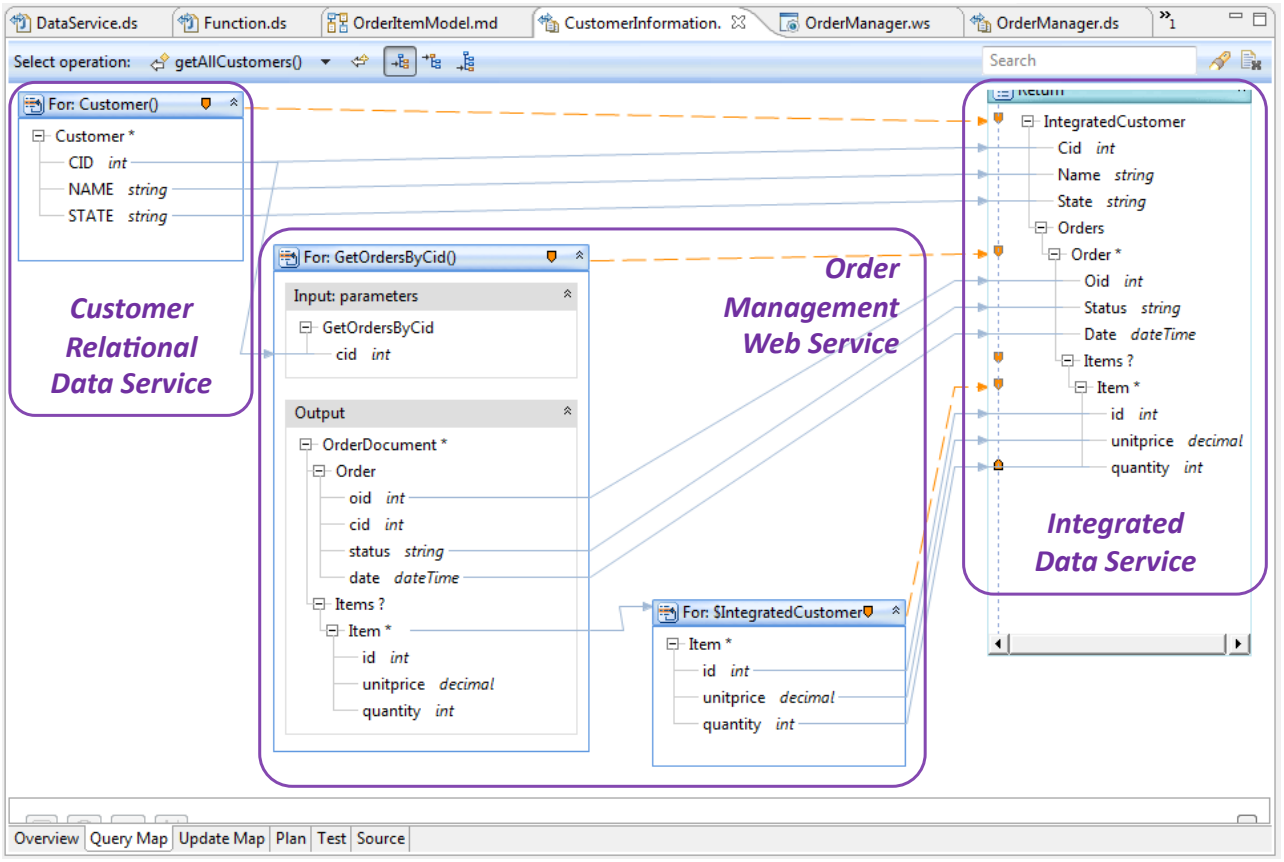

Figure 8: Integrating RDBMS and Web Service Data Sources