Integrating distributed data sources with OGSA–DAI DQP and VIEWS

BY BARTOSZ DOBRZELECKI1,*, AMREY KRAUSE1, ALASTAIR C. HUME1, ALISTAIR GRANT1, MARIO ANTONIOLETTI1, TILAYE Y. ALEMU1, MALCOLM ATKINSON2, MIKE JACKSON1 AND ELIAS THEOCHAROPOULOS2

1EPCC, University of Edinburgh, James Clerk Maxwell Building, Mayfield Road, Edinburgh EH9 3JZ, UK
2National e-Science Centre, University of Edinburgh, Edinburgh EH8 9AA, UK

OGSA-DAI (Open Grid Services Architecture Data Access and Integration) is a framework for building distributed data access and integration systems. Until recently, it lacked the built-in functionality that would allow easy creation of federations of distributed data sources. The latest release of the OGSA-DAI framework introduced the OGSA-DAI DQP (Distributed Query Processing) resource. The new resource encapsulates a distributed query processor, that is able to orchestrate distributed data sources when answering declarative user queries. The query processor has many extensibility points, making it easy to customize. We have also introduced a new OGSA-DAI VIEWS resource that provides a flexible method for defining views over relational data. The interoperability of the two new resources, together with the flexibility of the OGSA-DAI framework, allows the building of highly customized data integration solutions.

Keywords: query processing; data integration; distributed systems

1. Introduction

The mediator–wrapper architecture (Wiederhold & Genesereth 1997) is a practical way of building data integration systems. This design introduces wrappers that provide a standard interface to heterogeneous resources and mediators to present a global view of wrapped data sources. The OGSA-DAI (Open Grid Services Architecture Data Access and Integration) middleware (Jackson et al. 2007) provides a convenient way of exposing data sources using Web services and is often used to implement wrappers. Data integration solutions, built on top of OGSA-DAI, often make use of the wrapping functionality, but employ their own bespoke query evaluation services. OGSA-DQP (Distributed Query Processing) (Lynden et al. 2009) is an example of such a system. This paper describes two extensions that significantly enhance the mediation and wrapping

*Author for correspondence (b.dobrzelecki@ed.ac.uk).

One contribution of 15 to a Theme Issue ‘e-Science: past, present and future II’.
capabilities of the OGSA-DAI middleware. One of them is a distributed query processor that allows federated databases to be created. The other provides a flexible method for defining views over relational data.

The structure of this paper is as follows. Section 2 introduces an example use case and walks through a solution using the default OGSA-DAI relational wrappers. The OGSA-DAI DQP component is introduced in §3. Section 4 discusses implementation details of the query processor. Section 5 describes the OGSA-DAI VIEWS resource. A performance analysis is presented in §6. This is followed by some current applications. Section 8 gives an overview of related distributed query systems. Section 9 concludes the paper and highlights plans for the future.

2. Data integration with OGSA-DAI

OGSA-DAI is a powerful workflow engine geared towards integration of heterogeneous data resources. The most commonly exploited feature of the framework is its ability to provide Web-service wrappers to data sources. However, this feature alone is only a small subset of the OGSA-DAI feature set. The OGSA-DAI workflow engine allows data processing to take place close to the data. Users can integrate multiple data resources exposed by a single OGSA-DAI server, but more importantly it is possible to build complex data integration systems by coordinating workflow execution and data transfer between several distributed OGSA-DAI Web services.

To demonstrate OGSA-DAI functionality, we will introduce a simple data integration use case. For simplicity, we choose a use case from the business domain. The following sections will gradually add complexity to this example. Our use case involves two geographically distributed branches (B1 and B2) of a company. Each branch stores data about its orders in a horizontally partitioned orders table. The problem is to bring about the union of these two distributed datasets.

The first step in solving this use case is to deploy an OGSA-DAI service in a Web application container at each branch. The default OGSA-DAI deployment exposes only one resource called the Data Request Execution Resource (DRER). This resource is responsible for executing OGSA-DAI workflows. To expose the actual database, we need to deploy an additional relational data source wrapper (B1Resource). It is now possible to query the data resource by constructing a query–transform–deliver workflow (QTD).

An OGSA-DAI workflow consists of a number of units called activities. Activities perform a well-defined data-related task and can be associated with a specific resource and therefore define its interface. Users interact with OGSA-DAI services by submitting workflows to a Data Request Execution Service (DRES). User requests are encapsulated in SOAP (Simple Object Access Protocol; Gudgin et al. 2007) messages and sent over HTTP (Fielding et al. 1999). The QTD workflow is a recurring interaction pattern. Figure 1b presents a graphical representation of a QTD workflow used to obtain data from a data source. The first activity (SQLQuery) is targeted at the B1Resource resource and takes a Structured Query Language (SQL) query as its input. It produces a list of tuples, OGSA-DAI’s internal representation of relational data. Tuples are then
Integrating distributed data sources

Figure 1. A client communicates with the OGSA-DAI server by invoking Web service operations provided by the Data Request Execution Service (DRES). The most commonly invoked operation is a request for a workflow to be evaluated by the Data Request Execution Resource (DRER). (a) OGSA-DAI deployment at branch 1; (b) OGSA-DAI workflow.

consume the TuplesToByteArrays activity, which serializes data from its internal representation to a form suitable for inclusion in the SOAP response. The resulting document is then sent back to a client. The QTD workflow could be all that we need to provide the solution to our use case. The solution would involve submitting the QTD workflow to both of the branches and calculating the union of the results on the client side. This solution, however, would not fit the OGSA-DAI philosophy, which advocates encapsulation of data processing in activities and moving computation closer to the data.

To be able to calculate the union on the server side, we need to coordinate data transfer between OGSA-DAI services. The interaction is shown in figure 2 and involves three workflows:

- The first workflow submitted to B2 creates a DataSink resource at B2. A data sink is an OGSA-DAI resource that allows clients to push data to an OGSA-DAI server. This enables asynchronous data transfer. The request response includes the resource identifier for data sink.
- The second workflow submitted to B1 is similar to a QTD workflow. It is used to query the B1Resource but, instead of delivering the results to the request response, it pushes data to the DataSink resource.
- The third workflow submitted to B2 queries the B2Resource and calculates the union of the tuples delivered to the DataSink resource. Finally, the results are delivered to the client in a SOAP response.

The interaction required to calculate the union on the server side is quite complicated. The OGSA-DAI client toolkit application programming interface (API) simplifies programming such interactions. Hand-coding data integration
scenarios involving two data sources is not unreasonable. However, once the number of data resources increases, the process quickly becomes tedious and error-prone. A resource that allows users to orchestrate such interactions in a declarative fashion is described in §3.

3. Creating federations using OGSA-DAI DQP

We start by adding complexity to our use case. In addition to horizontally partitioned orders table, we now introduce another location—the headquarters (HQ) where the central database of all the clients is stored (the clients table). We are interested in querying, in a declarative way, the integrated view of the orders data from all branches with the data about customers. When we formulate queries, we are not interested in the exact location of the data. All we want to do is to get answers to questions like: ‘Give me names and phone numbers of all customers who have at least one order for which the price exceeds 100 000’ (Q1). In other words, we are interested in a single database view of all the data sources.

The OGSA-DAI DQP data resource allows the creation of virtual databases by federating over a set of distributed data resources. It exposes integrated database schema and provides a read-only client interface that is identical to any other relational data resource.
Integrating distributed data sources

A new DQP resource can be created in two ways. It can be a static federation, configured by an OGSA-DAI server administrator, and created automatically each time a container is restarted, or it can be created dynamically by a client request via a factory activity. Both methods use a configuration document that defines a federation. A configuration document lists data sources identified by the URL of the OGSA-DAI service and the resource name. Once created, a DQP resource is ready to accept queries. Figure 3 shows the federation framework in our scenario.

To formulate a query, we need to know the names of the tables in our federation. Using the table names as defined in the physical databases may lead to name clashes. OGSA-DAI DQP resolves this problem by prefixing the table names with OGSA-DAI relational resource wrapper names. If the DQPResource is requested to provide the logical schema of its federation, the following table names will be constructed:

- HQResource_customers
- B1Resource_orders
- B2Resource_orders

Using resource names as prefixes may not be enough, as nothing prevents users from setting the same names for resources exposed by different services. In such cases it is up to the user configuring a DQP resource to use appropriate aliases. We can now formulate the query Q1 as:

```sql
SELECT DISTINCT c_name, c_phone
FROM HQResource_customers,
(SELECT * FROM B1Resource_orders
UNION ALL
SELECT * FROM B2Resource_orders
) AS orders
WHERE o_custkey = c_custkey
AND o_totalprice > 100000
```

The DQPResource will respond to this query, orchestrating evaluation and data transfers between Web services that expose the actual data.
4. DQP implementation details

A DQP resource encapsulates the OGSA-DAI distributed query processor. The client’s SQL query is parsed into an abstract syntax tree, which is used to generate a logical query plan (LQP). The LQP is then validated and normalized. The LQP is optimized by a chain of query plan transformers. The optimal plan is then split into partitions, translated into a set of OGSA-DAI workflows and submitted to OGSA-DAI services for execution.

The default set of optimizers performs both heuristic optimizations (like select push down) and cost-based rewriting (join ordering, for example). Optimizers take into account the locality of the data and always try to push as much processing as possible down through the OGSA-DAI relational data resource wrappers and into the underlying relational database management systems (RDBMSs). The optimization chain is fully configurable and it is possible to develop and deploy new and improved optimizers.

Query plans are represented as trees of relational operators. Figure 4 shows the LQP generated for the query-Q1. This clean internal representation simplifies reasoning about the run-time performance, while also making the query processor highly extensible. In addition to pluggable optimization, it is straightforward to introduce new relational operators and expose these to the query language surface using relation-valued user-defined functions (UDFs). It is also possible to extend the OGSA-DAI DQP functionality with custom scalar and aggregate UDFs.
OGSA-DAI is the distributed run-time environment that executes query plans. Each relational operator from an LQP is mapped to a partial workflow of OGSA-DAI activities. A complete workflow represents a single query plan partition. For each partition, a workflow is submitted to the appropriate OGSA-DAI service and executed asynchronously. The mapping between relational operators and OGSA-DAI workflows is fully configurable. Advanced users are able to experiment with improved or specialized operator implementations.

5. OGSA-DAI SQL VIEWS

A database view is a named, virtual table composed of the result set of a stored query. The OGSA-DAI VIEWS component allows for the definition of a view to be created on top of any resource able to execute an SQL query. Similar to a DQP resource, a view resource looks like a read-only relational resource providing table schema and executing queries. A view can be defined in this approach without requiring write access to the underlying RDBMS, meaning that views can be defined over remotely located read-only databases. A VIEWS resource is configured by providing a map between table names and queries. Implementation is based on query rewriting on the abstract syntax tree level.

There are many uses of VIEWS. Figure 5 shows how VIEWS could be used in the scenario described in this paper. A view can be used to join and simplify multiple tables into a single virtual table. This is often applied to simplify the writing of queries from the client’s perspective. Take, for example, the SQL statement for query Q1. The users at the HQ are interested in data from all branches joined with the client database. They are rarely interested in the orders data only. Instead of writing complex queries with derived tables, they could simply ask:

```
SELECT DISTINCT c_name, c_phone
FROM clientorders
WHERE o_totalprice > 100000
```
To make this possible, the HQView resource is deployed, which defines a view using the following mapping:

```
clientorders ->
  SELECT *
  FROM HQResource_customers,
  (SELECT * FROM B1Resource_orders
   UNION ALL
   SELECT * FROM B2Resource_orders
  ) AS orders
  WHERE o_cuskey = c_custkey
```

A use of VIEWS is to limit the exposure of tables to the world outside the organization. Assume that branch B1 does not want HQ to know about pending orders but instead wishes to expose information about the completed orders only. Instead of exposing the entire `orders` table, the B1View resource is deployed. The following view is defined on top of the B1Resource:

```
orders -> SELECT * FROM orders WHERE o_orderstatus = 'F'
```

OGSA-DAI can be configured to hide B1Resource from clients; so HQ will only be aware of resource B1View. VIEWS can be defined to provide security down to the level of individual rows by factoring in information from clients about their identity and privileges.

VIEWS has powerful data integration capabilities. VIEWS can be used to smooth out differences between table schemas. These mappings exploit the expressiveness of the SQL language and can range from simple column renaming to complex, value-replacing joins.

6. Performance

Understanding the performance of a complex distributed system like OGSA-DAI DQP is not a trivial task. Results presented in this section are meant to give the reader a feeling for what level of performance one may expect when running queries over federated databases.

The experimental set-up is presented in figure 6. The three machines used in the deployment match Amazon’s EC2 specification for the Small Instance. Instances are running Fedora LINUX 8, MySQL 5.0.45 and Sun JRE 6u18. Machines are geographically distributed between Ireland, the USA and Edinburgh. The client machine is co-located with the server in Edinburgh. The TPC-H benchmark (http://www.tpc.org/tpch/) was used to populate the databases. The standard 1Gb dataset is replicated on all exposed databases.

The query Q1 was used in all experiments. The size of the query was modified by adjusting the value of the `o_totalprice` filter. For queries sent directly to the database, the `orders` table is unioned with itself. In each experiment the client code iterated through the result set to make sure that de-serialization time is included in the timing results. All client queries are asynchronous. Results are
delivered to a data sink and fetched by a client. Reported times are averaged from five runs. Four experiments were performed.

— Experiment 1. Queries were sent from the client to the database at HQ using direct Java Database Connectivity (JDBC) connection.

— Experiment 2. Queries were sent from the client to the HQResource at HQ using OGSA-DAI Web service clients. This experiment demonstrates the overheads imposed by the Web service layer and serialization.

— Experiment 3. Queries were sent from the client to the DQPLocal resource, which is a federation of local resources exposed by the same OGSA-DAI instance. All of the local resources point to the same database. This fact, however, is hidden from the query compiler, which can push down only the o_totalprice filter down to the database. Most of the query evaluation happened in OGSA-DAI without access to indexes.

— Experiment 4. Queries were sent from the client to the DQPResource, which federates over distributed data resources. In this scenario, data are being transferred over the network.

The results are presented in table 1 and plotted in figure 7. The overhead introduced by the Web service layer and serialization increases with the size of the result set and contributes up to 75 per cent of the execution time on top of the direct JDBC connection for large datasets. This result is expected, as fetching data from the data sink resource involves multiple exchanges of SOAP messages. The bigger the result set, the more messages that need to be exchanged.
The cross-over point of the execution times obtained for experiments in which a query is run over federated resources is easy to explain. OGSA-DAI uses a streaming model in which each activity is executed concurrently in a separate thread. When a query is evaluated by a single OGSA-DAI instance that federates over local resources, all of the activities will compete for CPU cycles. When a query is evaluated by several distributed OGSA-DAI servers, the benefits of the parallelism implicit in activity workflows begin to make a difference. The cross-over is at the point where the benefits from parallel execution overshadow the data transfer penalties.

The cost of OGSA-DAI DQP over direct JDBC is offset by the benefit of presenting and querying distributed data sources as if they were a single data source. Whether this trade-off is acceptable or not is for specific users to determine. Alternative delivery formats and especially lightweight (non-SOAP-based) messaging protocols offer the potential to reduce the overheads and deliver more competitive performance.

Note that the default installation of MySQL does not perform selection push-down optimization. Our results for direct MySQL queries used a hand-optimized version of Q1. An unoptimized query took 51 s for the query with the filter value that selected the least number of rows.
7. Applications

OGSA-DAI DQP has been successfully applied in several e-Science projects. The MESSAGE (http://bioinf.ncl.ac.uk/message/) project, for example, delivered an infrastructure to handle data from a range of heterogeneous fixed and mobile sensor devices. The project used OGSA-DAI DQP to implement a virtual federated database integrating several databases storing data captured by distributed sensor networks.

Our system is also used to integrate medical data. The deployment diagram for the medical trial analysis application is presented in figure 8. This use case relies quite heavily on the extensibility provided by the OGSA-DAI framework and query processing functionality provided by the DQP resource. The project developed relational wrappers (OGSA-DAI resources) for the two Web services: an XNAT (http://www.xnat.org/) Web service that natively exposes a RESTful interface is an interactive image store, and a Freezerworks (http://www.freezerworks.com/) resource that understands SOAP messages and stores information about biological tissue samples. The DQP resource federates over these two Web services and over a relational database storing trial results.

The functionality of the DQP resource is further enhanced by a local-as-view mediator, which rewrites user queries and uses the query processor to optimize and execute. It is important to note that DQP can federate not only over relational databases, but over any data source that can be wrapped in a relational interface. The resource must be able to map queries formulated in some subset of SQL to queries suitable for the wrapped data source and describe results in terms of relational schema.

8. Related work

Our work builds on experiences of the OGSA-DQP project. We have developed our solution from scratch, with the goal of providing all the functionality delivered by the OGSA-DQP system. The most important difference is that we have moved the entire query evaluation into the OGSA-DAI run-time system. We have also redesigned the system to be more flexible and easier to extend. To distinguish the new distributed query processor from the OGSA-DQP project, we call the component described in this paper the OGSA-DAI DQP resource.

One of the early examples of a successful distributed query processor implemented with hierarchical Web services was developed by the SkyQuery project (Malik et al. 2003). The system provides a mediator–wrapper for integration of astronomy data archives. Similar data federation functionality is delivered by the MOBIUS project (Hastings et al. 2004)—a distributed query processor where individual data resources are exposed as XML (eXtensible Markup Language) services able to answer XPath queries. Mediation based on XML processing is also used by the XAware (http://www.xaware.org/) data integration system. Commercial products providing similar data federation functionality include among others the IBM WebSphere Information Integrator (http://www-01.ibm.com/software/data/integration) and the Virtuoso Virtual Database (http://virtuoso.openlinksw.com/).
Many of the solutions mentioned above adopt a centralized query processing approach. This means that what can be computed remotely is constrained by the capabilities of the data source. For example, for XML-based solutions, this is limited to what can be expressed in XPath or XQuery. In contrast, OGSA-DAI DQP not only sends as much processing as possible to the underlying data source, but can also perform arbitrary calculations on a remote OGSA-DAI server. This can be exploited in queries with user-defined functions or extended through relational operators. Fully distributed query evaluation provides means for better utilization of resources by exploiting parallelism implicit in many data-processing workflows. The flexibility of the OGSA-DAI DQP component allows users to create centralized federations with star topologies.

9. Conclusions and future work

This work has presented two OGSA-DAI extensions that provide an integrated solution for building federations of distributed data sources. These extensions are tightly integrated with OGSA-DAI. All stages of query processing and view rewriting happen within OGSA-DAI. We have demonstrated how both extensions interoperate. It is possible to define both a view on top of a DQP federation and a federation over a set of views. This interoperability allows for building highly customized data integration solutions.

Many of the techniques presented in this paper are already established within databases. The OGSA-DAI extensions described make them available across heterogeneous distributed databases. The system currently is only used to
Integrating distributed data sources

Future work will involve adding support for XML and RDF (Resource Description Framework) data sources. This will be achieved by developing new wrappers and extending the query language with XML- and RDF-specific functions for querying these types of data sources also. Our goal is to support all queries defined by the SQL-92 standard. We are constantly improving our optimizers by developing more accurate cost models. Enhancement of the query execution strategies is a key focus of the current work.

The OGSA-DAI project is funded by the EPSRC via the Open Middleware Infrastructure Institute, UK. We also gratefully acknowledge the input of our past and present partners and contributors to the OGSA-DAI project including: EPCC, IBM UK, IBM Corporation, NeSC, University of Manchester, University of Newcastle and Oracle UK. We also acknowledge the input of Craig Thomson and the EU FP6-funded BEInGRID project who worked on the OGSA-DAI SQL Views component.

References


Wiederhold, G. & Genesereth, M. 1997 The conceptual basis for mediation services. IEEE Expert 12, 38–47. (doi:10.1109/64.621227)