Exploration of data integration scenarios using OGSA-DAI WSRF 3.0

Xiaochen Yu
Aug. 20, 2007

MSc in High Performance Computing
The University of Edinburgh
Year of Presentation: 2007
Acknowledgement

I would like to express my thankfulness to Dr. Mario Antonioletti and Mr. Bartosz Dobrzelecki, for their patience and care, and for giving me guidance during the process of my project, an invaluable start to my academic career.
CONTENT

1 INTRODUCTION TO THE PROJECT ................................................................. 5
  1.1 BACKGROUND AND MOTIVATION ....................................................... 5
  1.2 TERMINOLOGY .................................................................................... 6
  1.3 A BRIEF INTRODUCTION TO OGSA-DAI ......................................... 9
  1.4 OTHER RELATED SOFTWARE ............................................................. 11
    1.4.1 Data access and integration ......................................................... 11
    1.4.2 Data integration job management ................................................ 14
  1.5 PROBLEM SPECIFICATION FOR THIS PROJECT ............................... 14
    1.5.1 Abstracting a use case from real world examples ....................... 14
    1.5.2 Abstracting the use case into a DAG ............................................ 16
    1.5.3 Generating an operation tree from the DAG ............................... 17
    1.5.4 Mapping the smallest subtrees to OGSA-DAI workflows .............. 19
    1.5.5 Summary of the problem and work plan ..................................... 20

2 COMPARISON BETWEEN OGSA-DAI WSRF 2.2 AND OGSA-DAI WSRF 3.0 20
  2.1 THE OGSA-DAI WSRF 2.2 TESTING .................................................. 21
  2.2 THE OGSA-DAI WSRF 3.0 TESTING .................................................. 25
  2.3 SUMMARY ............................................................................................ 29

3 EXPLORATIONS WITH THE BASIC DATA INTEGRATION SCENARIOS ...... 29
  3.1 MOTIVATION AND OVERVIEW ......................................................... 29
  3.2 EXPLORATION OF DATA INTEGRATION SCENARIOS ....................... 30
    3.2.1 Introduction to the use case ....................................................... 30
    3.2.2 Experimental setup ................................................................. 32
    3.2.3 Scenario 1 ................................................................................ 33
    3.2.4 Scenario 2 ................................................................................ 35
    3.2.5 Scenario 3 ................................................................................ 38
    3.2.6 Scenario 4 ................................................................................ 39
  3.3 A RECAP OF DATA INTEGRATION SCENARIOS AND DATA INTEGRATION USE CASES 42
  3.4 LOCATION TAGGING ALGORITHM FOR THE SMALLEST SUBTREES .... 42

4 A DATA TRANSMISSION COUNT OPTIMISED ALGORITHM FOR OGSA-DAI .... 48

5 FURTHER WORK ....................................................................................... 50

6 CONCLUSIONS ....................................................................................... 50

7 REFERENCES .......................................................................................... 50

FIGURE 1-1: A PIPELINE WORKFLOW IN OGSA-DAI ........................................ 9
FIGURE 1-2: A SHOWCASE OF OGSA-DAI DATA SINK .................................... 10
FIGURE 1-3: A SHOWCASE OF OGSA-DAI DATA SOURCE .............................. 11
FIGURE 1-4: WS-JDBC SERVLET ENGINE ..................................................... 12
FIGURE 1-5: NEW GRID MIDDLEWARE MODEL FROM G-DSE ..................... 13
FIGURE 1-7: A USE CASE OF COMPLEX DATA INTEGRATION ..................... 15
FIGURE 1-8: THE OPERATION TREE OF THE USE CASE ............................. 18
FIGURE 2-1: COMPARISON BETWEEN 2.2 AND 3.0 .................................... 21
1 Introduction to the project

1.1 Background and motivation

There are many types of public data available on the internet. For example, there are “hundreds of databases for different levels of biology research,” [1] and their number is growing rapidly. Often the data stored in a database belongs to a specific domain, but the scientific researchers using these databases need to collect data from a broader range, therefore, it would be advantageous if data from these different sources could be integrated.

An example where public data is being integrated lies in the research being carried out on metabolic pathways, a series of chemical reactions catalyzed by enzymes in a living system [2]. Traditionally, in this area biologists have to input the name of a pathway of interest with various searching constraints, into a web form that acts as an interface to a KEGG pathway database, one of the major pathway databases in the world located in Kyoto, Japan, to find the enzymes and compounds that are involved in this pathway. They then have to go to protein-protein reaction databases, such as BIND and MINT, to find out what reactions took place between these proteins and compounds. Finally, they have to go to PubMed, a research paper repository, to search for publications for these reactions before they have accumulated enough comprehensive material for their studies. Different kind of metabolic pathway studies have different uses for the data retrieved. A medical researcher would concentrate on data for a specific pathway, which shows evidence of an undiscovered disease or for a potential new drug target. A computational systems biology researcher would gather massive amounts of data that relating to a certain set of pathways, trying to reconstruct a metabolic pathway network, which demonstrates a more distinctive numerical relationship between genes and proteins. However, in the context of data integration, the query activities they use for these databases are similar. Each of these activities includes launching different queries on different databases, gathering the resulting data together and possibly using this data to generate constraints for other queries, saving both the intermediate and final resulting data, which are essential to their research.

Another common factor in this data integration example is that all the databases are relational. Although users normally query for data by interacting with web forms, their inputs and settings for the search constraints are nevertheless translated into query statements in a form usable by relational databases. What is also usually significant for this kind of data integration is that the databases involved are often geographically distributed – the reasons for this can be either practical or historical. On the one hand, biological information shows self-completeness at the same microscopic levels – quite often people query for data at only one level (e.g. genes’ information, without any related protein or DNA information) for their research. In these cases, storing gene data and protein data in one database would both be meaningless and break the clearness in the structuring of data. We can deem the biological information preservation in these cases as “horizontal structuring biological data”. Meanwhile, in some other databases, biological data are structured and stored by the classifications of the organ or the species they are from – a chemical reaction may have a distinct effect in a reaction network on different organs (or species), so in these cases, the biological information from the different microscopic levels can be collected solely from one type of organ (or species) and stored. We call this kind of data preservation “vertical structuring biological data”. Much of this data has been collected by different organisations, and stored in databases they have. Some of these databases have become the most popular databases used today. Therefore, neither removing overlapping data in some of these databases nor merging the data into one database is feasible or worthwhile. As a consequence of this, in many biological database searches, users have to input the same query constraints to different databases to get hold of a complete set of the data of interest.

What has been described in the previous example is a basic workflow acting on distributed databases. Taverna [3], a biological workflow editor, provides a user interface which enables users to integrate data by modifying a query result from one database and using it as the query keyword for another database. In this manner, the data flows from one place to another in a workflow, according to a route defined by the user.
Another area where technology has been used to facilitate the data integration process lies in the study of astrophysics. SkyQuery [4] is “a distributed query execution system for the astronomy community of the world, which will enable them to run combined queries on the existing heterogeneous astronomy archives” [4]. Today, there are astronomic databases around the world, which may contain data for the same celestial bodies but under different wavelength, derived from different types of astronomical telescopes. To get to know more about a feature (e.g. constituent, age, etc), one has to study the spectrum composition of it. To do this, SkyQuery users do not have to query the different databases separately; rather, they only put some SkyQL statements, a query language derived from SQL, in a web form before a certain application, called “SkyQuery web service”, parses the SkyQL and returns the resulting data and images back to them. An important issue in this case is the time users have to wait between a SkyQL statement being submitted and the result being returned, because astronomical data is often large and the data integration algorithms are time consuming. SkyQuery tackles this challenge by planning the route the data flows prior to launching the workflow. This scheduling effort alleviates the load on the network and increases the potential currency in the execution of the data integration operations. SkyQuery is at a prototype stage. Currently, there are only three databases that can be queried.

From the examples above, we see how data integration tooling helps scientists perform their science. In the biological domain a lot of these things are currently done manually, although tools such as Taverna help to automate some of this work. At the other extreme, the SkyQuery application is a good example where an application helps scientists to retrieve data from different databases. Therefore, it would be meaningful to develop data integration applications that have a generic scope. This is the area that this project explores, with the aid of Grid technology.

The Grid is a future form of the internet where users share resources. In the Internet, resource sharing is limited. For example, the most common uses of the Internet are downloading / uploading files from / to an FTP server; sending / receiving emails; and the browsing of web pages. In a Grid perspective, the resources shared by users have wider scope. In the Grid, a piece of equipment or application has the potential to be a shared resource if it can be controlled by, or the information it provides can be accessed by, a distributed user community. In reality, it is the data produced by such equipments or applications is what users require. OGSA-DAI is a grid middleware that facilitates the sharing of data. The working mechanisms for this are discussed in the following chapters.

1.2 Terminology

In this section several terms used across this thesis are defined.

**Data integration**

In this thesis, the term “Data integration” refers to the activity a person, or a computer program, performs to combine two or more data sets together. The aim of the combination is to create a single data set that another computer program can analyse and generate information that is useful some scientific research.

“**parallel operation**” and “**following operation**”

In a workflow, from a temporal perspective, there are two ways in which two or more operations can take place. On the one hand, there are operations that occur concurrently, the resulting data from these operations is put together. In this thesis, a group of operations having this property are called “parallel operation”s. On the other hand, there are data operations that occur in sequence, the resulting data of an operation is then operated on by another operation. In this thesis, if an operation depends on the completion of one or more operations, this operation is called a “following operation” while all the operations which should provide their output as the input for this operation are called “preceding operations”.

**Web service**

In this thesis, the term “Web service” refers to the activity a person, or a computer program, performs to combine two or more data sets together. The aim of the combination is to create a single data set that another computer program can analyse and generate information that is useful some scientific research.
W3C has a very precise and clear definition of a web service, therefore this definition is quoted directly here, with some explanations to help understanding.

“A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description…” [25]

The phrase “machine-to-machine interaction” in the definition means that from a user’s perspective, a web service is networked application, which can only be used through a program to invoke functionalities in the web service. The functionalities available and its inputs are specified in a WSDL file. The WSDL file uses XML, as XML can easily be parsed by a computer application. A coder using a web service only needs to download its WSDL file into their Integrated Developing Environment (IDE), the IDE will parse the WSDL file to generate code template for the programmer. The coder then completes the code with their problem-specific parameters.

**Data service**

In this thesis, a data service refers to a web service which exposes databases.

**Web Service Resource Framework (WSRF)**

According to the Globus alliance definition:

“The WS-Resource Framework (WSRF) is a set of six Web services specifications that define what is termed the WS-Resource approach to modelling and managing state in a Web services context.” [24]

**DAG**

A DAG is a directed graph in which there is no directed cycle. Simply put, if one starts from any of the vertexes for this kind of graph, and goes along the edges with the direction indicated in the edges, they can never reach the vertex from which they started.

**Source node and destination node**

In this thesis, in a DAG, for an edge E, if E starts from node A and ends at node B, then A is the source node of E and B is the destination node of E.

**Dataflow**

In this thesis, the term dataflow refers to the entire movement of datasets in one data integration.

**Smallest subtree**

In this thesis, the *smallest subtree* in a tree refers to a subtree whose descendant nodes are all leaves.

**Root**

In this thesis, the *root* in a tree or subtree refers to a node which does not have any parent nodes.

**Leaf**

In this thesis, a *leaf* in a tree or subtree refers to a node which does not have any descendant nodes.

**Operation tree**

Since the nodes in the tree used in the algorithms stand for one operation to data which takes place at one service, the tree is named as “operation tree” for clearer indication.
Original table

In this thesis, the original tables refer to the tables stored in the databases.

Intermediate result / intermediate table

In this thesis, the Intermediate result / intermediate table refers to the output of data integration operations, which is not the final result.

Location

In this thesis, the location of an operation refers to the URL of a data service which is allocated to perform the operation defined; the location of an original table refers to the data service exposing the database this table resides at.

Tag

In this thesis, the tag of a node refers to the URL of the data service which is allocated to perform the operation defined in this node.

Location tagging

In this thesis, Location tagging refers to an activity which adds the data service URL information to a node in an operation tree. Through tagging a location to a node, the operation represented in the node is determined to be allocated to the data service on the URL.

Operation pattern

In this thesis, the operation pattern refers to the entire tags’ and operations’ information contained in a smallest subtree whose entire nodes are tagged.

JVM’s Garbage collection and heap size

In chapter 2, the concept of the JVM’s Garbage collection and heap size are used. As background knowledge, a brief introduction of java heap size and garbage collection¹ is given below:

Garbage collection is the JVM's process of freeing up unused Java objects in the Java heap. The Java heap is where the objects of a Java program live. It is a repository for live objects, dead objects, and free memory. When an object can no longer be reached from any pointer in the running program, it is considered "garbage" and ready for collection.

The JVM heap size determines how often and how long the VM spends collecting garbage. An acceptable rate for garbage collection is application-specific and should be adjusted after analyzing the actual time and frequency of garbage collections. If you set a large heap size, full garbage collection is slower, but it occurs less frequently. If you set your heap size in accordance with your memory needs, full garbage collection is faster, but occurs more frequently.

The goal of tuning the heap size is to minimize the time that the JVM spends doing garbage collection ... To ensure maximum performance during benchmarking, high heap size values might be needed to ensure that garbage collection does not occur during the entire run of the benchmark.

¹ This comes from the online tutorial available at: http://edocs.bea.com/wls/docs70/perform/JVMTuning.html#1106798.
1.3 A brief introduction to OGSA-DAI

The OGSA-DAI WSRF version [8] is a WSRF compliant Grid middleware product allowing users to construct workflows that commonly consists of database query, dataset transformations and dataset delivery.

In the WSRF version of OGSA-DAI, a user invokes a data service, Data Request Execution Service (DRES), and retrieves the result of their request through a Data Request Execution Resource (DRER). Apart from the query result, in OGSA-DAI the databases exposed to the user are deemed to be another kind of resource, Data Resource (DR), as well.

Furthermore, Data Sink Resource (DSiR) and data source resource (DSoR) can be used to transfer data between OGSA-DAI data services. These two types of resource can be used when multiple OGSA-DAI data services to transfer data between each other. The functionality of these two resources will be illustrated after another important feature of OGSA-DAI is introduced, the workflows.

Commonly, a web services only performs a single “task” on behalf of its user, returning any results to the user. In OGSA-DAI, a “task” can not only send the result back to the user, but also pass its result to a following “task” to be processed further; the “task” is referred to as an “activity” and a collection of activities, which does not deliver its result to the user until all the activities finish, is called a workflow. According to the different compositions of activities, workflows fall into three classes. In a pipeline workflow, the activities are ordered one after the other, execute in sequence and take input from the output of the previous activities, normally the first activity retrieving data directly from a database. Figure 1-1 depicts an example of a pipeline workflow. A sequence workflow consists of pipeline workflows, and a parallel workflow consists of pipeline workflows operating in parallel. The use of workflows is advantageous over normal web services as it gives users the freedom to combine the functionalities provided and determine which are applied to the data retrieved from the database. The component executing a workflow in OGSA-DAI is called the OGSA-DAI engine.

A workflow can retrieve data not only from data resources, but also from data sinks and data sources. The data sinks and data sources are to be found within the memory of the OGSA-DAI engine, in the form of the instances of various classes, depending on the output or inputs of the activity delivering data to data sink / data source.

![Figure 1-1: A pipeline workflow in OGSA-DAI](image-url)

A typical usage of an OGSA-DAI data sink is shown in Figure 1-2. Two OGSA-DAI services, namely OD Service1 and OD Service 2, have three pipeline workflows. A data sink is established at OD Service2 first, the pipeline workflow that writes to this sink and that obtaining data from this sink can be launched then. To provide the data to be process by the pipeline, the first activity in the pipeline, at OD Service 1, needs to wait for data to become available at the data sink at OD Service2. When the data...
becomes available, the data (in this case a web row set) is obtained by the OD Service1 and the other activities can then process the data. As can been seen, the data is “pulled” by OD service. Once the data is read out, the sink is automatically destroyed at OD Service2 by the OGSA-DAI engine.

Figure 1-2: A showcase of OGSA-DAI data sink

A typical use case of an OGSA-DAI data source is shown in Figure 1-3. The jobs of data processing of OD Service1 and OD Service 2 are the same as for the previous example, but this time the data delivery approach is changed. The resulting data set is sent by the data service that generates it to the OGSA-DAI data source, a buffer for data, which resides in the memory of another data service. The data source is accessed and its content is retrieved when those contents are needed. As can be seen, the data transmission using data source is asynchronous.

In the case shown in Figure 1-3, the data source is used to fulfill an asynchronous data transmission from OD Service1 to OD Service2. At first, a data source is established at OD Service1. Then the pipeline workflow is created at OD Service2 to retrieve a data set from a database and deliver the results to the data source residing at OD Service1. At last, another pipeline workflow created at OD Service1 read the data source and perform activities upon the data. Once the data is read out, the source is automatically destroyed at OD Service1.
OGSA-DAI has good function scalability, as the basic job executing element, an activity, can be written by a user, without modifying or referring to the implementation of the OGSA-DAI engine.

This project aims to investigate tooling for data integration workflows that act on geographically distributed databases. OGSA-DAI is an ideal choice of Grid data middleware to do this for three reasons: first, OGSA-DAI hides the heterogeneities of DBMS-provided APIs and data types amongst databases and provides users with a unified programming interface, which only exposes the data processing details to the user. This reduces the amount of time required for coding work, as well as leading to a concise structure of the code. Secondly, the data processing functionalities needed for the workflow can easily be written and deployed at every OGSA-DAI service. This is an important advantage that OGSA-DAI offers as no service dedicated to specific computation is needed, therefore data can be processed as it is retrieved from the database, without requiring any further transmission. Thirdly, the workflow and data sink / source mechanics in OGSA-DAI facilitate the realisation of cross service workflows, in which the result of one workflow can be sent to another service as an input to the workflow at that service.

1.4 Other related Software

An investigation has been carried out to find related software that could achieve the same goals set for this project. In this section OGSA-DAI is compared to other data access and integration middlewares, and the implementation of data integration job management is reviewed and summarised for a selection of Grid softwares.

1.4.1 Data access and integration

GReIC [9] is a Grid data middleware that has a similar underlying motivation as OGSA-DAI: to enable access through web services to both relational and textual databases on a Grid. The use of this software is similar to that of OGSA-DAI: by programming using some libraries [10] a user can invoke a GReIC data access service. There is not an apparent method to keep the result data from a GReIC data access service for subsequent services to process, as is enabled by workflows in OGSA-DAI.
SRB [11] provides a data middleware with a graphic user interface for sharing data across distributed, heterogeneous data resources, such as databases and file systems, separated by different administrative and security domains. This middleware has a Client / Server structure and it facilitates collection building, managing, querying and preserving data [11]. Efforts have been made to increase the reliability and availability of data storage, and data can be accessed either by its logical or physical name [11]. However, SRB does not provide functionalities for users to perform programmed computation to data at the data source, i.e. if a user wants to apply any computation on data accessed though SRB, they have to download the data to local storage and run the appropriate application on the data. In OGSA-DAI, the computation can be performed at the service side, provided the user has access to operate on the service machine.

WS-JDBC [12] is a web service enabled JDBC driver. Like the OGSA-DAI, every database using WS-JDBC is exposed as a web service, by a Servlet. Client and the Servlet communicate using SOAP messages. An architecture using WS-JDBC is shown as Figure 1-4. The Web services JDBC uses a Servlet engine, which receives the client’s queries in the form of SOAP messages while it uses the appropriate JDBC database drivers to communicate with the databases. The significance of WS-JDBC is that it provides a unified API for accessing different relational database; therefore when querying different relational databases the user only needs to use a single set of APIs, furthermore, client codes are portable to all the relational databases exposed by WS-JDBC. However, in terms of data manipulation, Web services JDBC is not as powerful as OGSA-DAI. There is no counterpart in WS-JDBC to activity and workflow mechanisms used in OGSA-DAI. To perform a data integration, the user nevertheless has to retrieve the data to their local machine and process it with their own applications.

“In the current Grid architecture, the Resource Framework Layer (RFL), the Information System Framework (IS) and the Information Data Model are able to retrieve run time data and information from the defined Grid node elements: Computing Elements (CE) and Storage Elements (SE)” [13]. G-DSE is a project aimed at architecture analysis of Grid middleware, proposing the concept of Query Element (QE), in order to extend Grid semantics [14] and enhance the Grid middleware model to manage data sources [14]. The new model is shown as Figure 1-5. The QE concept will benefit data integration workflow monitoring and brokering, because the data sources are arranged in a standard manner, as shown in Figure 1-6. However, there is not a wide realisation of this concept currently.
Figure 1-5: New Grid Middleware model from G-DSE.

Image courtesy of [14]

Figure 1-6: A parallel view of the job manager and of the dbmanager of Globus developed from G-DSE.

Image courtesy of [14]
1.4.2 Data integration job management

The ability to facilitate data access is a necessary condition for any middleware to be chosen for this project. OGSA-DAI has advantages over all the other candidate softwares and has been chosen to be used in this project. As mentioned in the introduction of this thesis, the SkyQuery project implements certain job management algorithms for brokering jobs amongst Grid nodes to accelerate the speed of query execution. Currently there are no job management implementations in OGSA-DAI. In this subsection several Grid middleware implemented job management softwares are reviewed and the features of each of them are abstracted. The outcome of the review will be reflected in the design of this algorithm, which is detailed in chapter 5.

The speed of the execution of the data integration is optimised in SkyQuery, by always sending the smaller chunk of data to a service to the larger chunk of data in a data is located. When a user inputs a SkyQL query into the web form on skyquery.net, the SkyQuery first parse this data-integrating query into separate subqueries, each targeted at one of the databases participating in this query. After this the subqueries are sent out to every SkyNode, not for execution, but to get an estimate of the size of data that is to be retrieved; this size is called the cardinality in SkyQuery. Having all the cardinalities of the sub queries, SkyQuery sorts the subqueries by increasing order of cardinality. The array of queries is then sent out to the SkyNode which has the largest cardinality. While it takes and starts to process its subquery, the SkyNode forwards the array of subqueries to the node with the second largest cardinality. In this way the subquery passing chain allocates jobs to every node involved in the data integration. Once the SkyNode with the smallest cardinality finishes its job, it sends the data to the service with second smallest cardinality. Having received the data, the second data service applies certain spatial matching algorithm upon the data received and the local data. In the end, the integrated data is sent back to the SkyQuery from the SkyNode with the largest cardinality. The data transmission uses a chain topology to communicate with each other and then a star-join topology with SkyQuery. This strategy has two advantages. First, the use of the bandwidth in this chain topology is better than that of a star-join topology, because input and output connections of one node are less in this topology. Secondly, the computational load of a machine, either a SkyNode or SkyQuery, is alleviated, thanks to the distributed data integration operation. Both of the above produce a better performance in SkyQuery. There are two noticeable design ideas in SkyQuery. The first is that the allocation of jobs is calculated and planed before the workflow is launched. The second is that the data transmission is kept distributed across all the participating data services. These two features are discussed further in this thesis, when job allocation algorithms are introduced.

1.5 Problem specification for this project

1.5.1 Abstracting a use case from real world examples

In this section, the two examples at the beginning of the thesis are re-examined. The data integration requirements are abstracted and a use case is derived for this project. Some exploration is done on the use case and finally two project stages are set up, specifying the essential processes need to be carried out.

Operations that take place either in parallel or in sequence as can be seen in the workflow of the biology example. The parallel operations are seen when the retrieved data from databases MINT and BIND are combined together; the queries on these two databases are therefore parallel operations. The following operations are seen when the resulting data from KEGG becomes the query constraint for both MINT and BIND; the following operation of the query to the KEGG is the transformation of the data which makes valid inputs for the following operations.

In the SkyQuery example, the data integration is more complicated. The database queries can be carried out simultaneously, while the operation that act on the result data can only take place when two sets of result data are available. Therefore, in this case, the first group of parallel operations is the database queries that act on the two databases with the smallest cardinality. The following operation of these two operations is the application of the spatial matching algorithm. This application, in turn, is the parallel
operation of the database query on the database with the third smallest cardinality. In this sense, if there are N databases which a user wants to exert the spatial matching algorithm on, there will be N-1 pairs of parallel operations, whose results are put together. In turn, each pair of parallel operation has a following operation, which exerts the spatial match algorithm on the results.

The two examples above are typical instances of data integration that take place every day in the world amongst scientists. In the biology example, the databases involved in the metabolic pathways are amongst the largest biological databases. In the astrophysics (SkyQuery) example, the workflow consists of recursive spatial match algorithm are performed by astronomers every day. Therefore, an abstraction can be made from these two examples as a use case, the analyses of which will determine the problem space of data integration in this project. A use case containing parallel operation pairs and multilevel following operation pairs is designed and show in Figure 1-7. A description of this graph is given below.

Figure 1-7: a use case of complex data integration
In the data integration use case shown in Figure 1-7, the cylinders represent database, the rectangle with a corner folded over represent the original database tables or intermediate results, and the arrows represent operations. The names of databases, intermediate results and operations are labelled in the diagram. Tables whose names start with a “t” are intermediate results; tables whose names start with a “T” indicate an original table. Original tables and intermediate results are in cases collectively referred to as “data” in this section.

In this use case, the databases are exposed by different OGSA-DAI data services. Data service 1 exposes DB1, data service 2 exposes DB2, data service 3 exposes D3 and data service 4 exposes DB4.

1.5.2 Abstracting the use case into a DAG

The diagram presented is a DAG – there is no directed cycle in this diagram. This is because running a set of operations on a piece of data to generate exactly the same data is not useful in data integration. Therefore, in the diagram shown in Figure 1-7 the data are interpreted as the DAG vertexes, and the operations are interpreted as the DAG edges. Please note that the node representing a user is not included in the DAG. The DAG vertex linking to the user represents the final result, and the arrow coming out from it represents a delivery. There could be multiple deliveries, in which case, there will be multiple nodes representing users.

It can be seen from the figure that in this use case, each database has two tables. An arrow coming out from a piece of data to another represents the fact that an operation has been applied on that data, resulting in another piece of data. There are cases where more than one operation is run on one data. More interestingly, there are cases where multiple arrows with the same name go into a DAG vertex – this represents an operation on multiple sets of data, such as an SQL Join operation or a spatial match in SkyQuery. Eventually the integration result is sent to the end user.

Please note that there will be no cases where more than one arrows links to the user, as this would mean that a data operation has taken place at the client side. Neither will there be cases where arrows with different names link to a table. This would indicate a situation where different operations on different data generate the same result.

The expression of operations represented in this DAG is shown below; from the numbers of brackets in it one can see the advantage of representing the data integration with a DAG instead:
1.5.3 Generating an operation tree from the DAG

In terms of perception, specifying data integration in the form of images is more amenable to humans, while computer programs handle are representations as the one shown in Statement 1-1: operational representation of the DAG. However, one can see from this expression that there is a repeated operation O4. If operated as it stands, repetitions like O4 introduce performance issues. In that case, the same data would be transmitted across to a data service to be processed, disregarding the fact that there may already be a copy at the destination service; it is likely that the operation would be preformed disregarding this information. Therefore, a data structure is needed which is able to find and remove this type of repetition – a tree is an ideal choice for a computer program to fulfill this task. Algorithm 1-1 depicts the conversion from a DAG to an operation tree.

The algorithm of conversion is straightforward, as is shown below. Please note that the operation tree generated here is not the spanning tree of the DAG.

Statement 1-1: operational representation of the DAG

O12(
  O11(
    O10(
      O6(T22,T41),
      O8(
        O3(
          O2(T11,t12), T22, O4(T11,T32,T42)
        ),
        O4(T11,T32,T42)
      ),
      O9(O5(T31,O6(T22,T41)),O7(O1(T12)))
    )
  )
)
Algorithm 1-1: transform DAG to operation tree

The generated operation tree of the DAG in the example therefore looks like:

1. From the DAG, find the edge that links to the user; make this the root of the tree;
2. From the DAG, find the other vertices to which this edge links, add these as descendents of the root node;
3. For every node in the lowest level of the tree, from the DAG, find the other vertices to which the representing query of this node links, add these as descendents of this node;
4. Repeat step 3 until all the leaves of the tree are the original table from the databases.

Figure 1-8: the operation tree of the use case

As is shown in Figure 1-8, the red spheres represent the original tables from the database, and the blue spheres represent queries. The root node represents the data the user wants.

Please note that if this thesis is printed in black and white, the red colour looks like a dark grey while the blue colour looks like light grey, therefore in the following description of the DAG, the term “red” should be interpreted as “dark grey” while the term “blue” should be interpreted as “light grey”.

The DAG is converted to tree in this application as it offers three advantages over the DAG representation:

1. In the operations tree, the intermediate tables are removed, therefore the set of elements (original tables and queries) in this data structure is the same as the set of elements involved in the query statement (see Statement 1-1).
2. The execution of the data integration is expected to avoid re-doing same operations on the same data set, which in the operations tree is represented as a branch node (blue sphere) with its unique sequential number in the label. Therefore the program can register all the operations which have finished by their sequential numbers, therefore whenever the same operation is encountered during the operation tree’s execution, rather than performing the same operation, the result of this operation’s previous homologue is retrieved.

3. In a tree data structure, the relationship of dependencies between queries and tables are significant. Let us make a comparison between the DAG (See Figure 1-7: a use case of complex data integration) and the tree (see Figure 1-8).

   In the DAG, the structure of the elements is table-centric. To make the tables in the graph unique, the information of the queries generating the tables is made redundant.

   If one wants to find the prerequisite operations and original tables for O3, they have to find all the edges labelled “O3”, and then find the set E of all the source nodes of O3, after this they have to find all the original tables in E and move them into set T, following, all the edges whose destination node is an element of E should be find and put into set P. Therefore the elements in T are the original tables for O3, and O3 is the following operation of all the elements in P.

   In the queries centric Using the Operations Tree, where the table information is compressed to minimal, the procedure for the same goal is much simpler: find the unique node O3, then take all its descendents, the blue ones are the dependant queries while the red ones are the dependant original tables.

1.5.4 Mapping the smallest subtrees to OGSA-DAI workflows

In this project, the data integration problem to be solved is proposed in the form of a DAG, and the DAG is transformed into operation trees, at last, the operation trees are analysed and used to construct workflows in OGSA-DAI. Note 1-1 shows these transformations of the data integration information.

### Note 1-1: the transformations of the data integration information

Having the operational tree in possess, an algorithm should be made to construct and launch OGSA-DAI workflows.

As stated previously, there are three kinds of workflows in OGSA-DAI. A pipeline workflow is the basic unit in a workflow parallelism.

The operation tree shown in Figure 1-8 can be executed in parallel. All the operations represented by the leaf nodes are queries to original tables, therefore they can be executed by OGSA-DAI as pipeline workflows. Within every subtree of the operation tree, all the leaf nodes in a subtree represent the preceding executions of the operation represented by the root, therefore, the pipeline workflows for the leaf nodes in this subtree can be put into an OGSA-DAI parallel workflow, and this parallel workflow can be in turn put into an OGSA-DAI sequence workflow, before the OGSA-DAI pipeline workflow of the root in put into this sequence workflow as well.

The problem lays within this idea is that which data service should be launched on. Take the example for \{O6, T22, T41\}. T22 is with DB2, which is exposed by data service 2, and T41 is with DB4, which is exposed by data service 4. It is certain that the query to the T22 will take place on data service 2 and that...
the query to the T41 will take place on data service 4, but all the data services in this use case are possible sites for the operation represented by O6 to take place. Therefore, the choices of data services drop into two classes: to choose one of the data service having the task to query the original table, or otherwise. To determine that which one choice will demonstrate better performance, experiment should be carried out.

1.5.5 Summary of the problem and work plan

After the completion of the executions of the operations in a subtree, the resulting data resides at the data service in which the sequence workflow was allocated. The execution of the sequence workflows will not complete at the same time, therefore it is a good strategy to make a data service which finishes its job before the other data service in its subtree to start transmitting data to the data service where a following operation will take place. By overlapping the transmission time and computation time, the overall execution time of a data integration will be decreased. Therefore the development of such an algorithm becomes one the tasks for this project as well.

However, before proceeding a suitable version of OGSA-DAI should be chosen for this project. At the time when this project was carried out, the latest version of OGSA-DAI (WSRF version) was version 2.2, but the OGSA-DAI (WSRF version) 3.0 pre beta was available. Therefore performance benchmarking of both versions of OGSA-DAI should be done, on the relating issues of the project.

Therefore, this project is divided into three stages. In the first stage, the OGSA-DAI WSRF 2.2 and OGSA-DAI WSRF 3.0 pre beta are benchmarked and compared on aspects relating aspects to this project. At the end of this stage, the version with more features facilitating the implementation of the program of this project and with better performance is chosen. In the second stage, some basic data integration scenarios are proposed and these are benchmarked. At the end of this stage, an optimised way of data transmission among a subtree can be found. In the third stage, an algorithm to overlap data transmission and computation is proposed. Due to the limited time of this project, only part of the proposed implementation was carried out. The completion of the implementation and the benchmarking of the algorithm is set as further work for this project.

The problem specification for this project can be summarised by the following three questions:

1. Which version of OGSA-DAI should this project use?
2. What is the fastest way that the smallest subtree can be executed in OGSA-DAI?
3. How to make the smallest data integration operation units that are represented as smallest subtrees be executed in parallel?

Investigations will be carried out for each of the above problems sequentially. The discussions for each of them are in sections 2, 3 and 4, respectively.

2 Comparison between OGSA-DAI WSRF 2.2 and OGSA-DAI WSRF 3.0

Using the scenario shown as Figure 2-1, a comparison between the performance of OGSA-DAI WSRF 2.2 and OGSA-DAI WSRF 3.0 is made in order to find the performance differences between these two versions of OGSA-DAI and eventually make the decision as which version of OGSA-DAI will be used for this project.
2.1 The OGSA-DAI WSRF 2.2 testing

Experimental setup:

Experiments were run using OGSA-DAI deployed on to WS-Core from the Globus Toolkit 4.0.3. Both the client side and the service side were run on a laptop: an HP Pavilion dv5128. The CPU for this laptop is a Core Duo T2400 1.83 GHz and the memory is 1GB. The experiment was run on Windows XP SP2 with J2SE 1.5.0.12. To control the heap size, ANT_OPTS=-XmsXXXm –XmxXXXm is used where “XXX” refers to an appropriate number, Xms refers to the minimum heap size, Xmx refers to maximum heap size, and “m” refer to the unit of the memory, MB.

The DBMS used for these experiments was MySQL 5.2 with connector version 3.1.7. The sample littleblackbook database table, distributed with OGSA-DAI, was used for all experiments. The average row length for this table is 66 bytes. The rows for this table have the following schema: int(11), varchar(64), varchar(128), varchar(20) [22].

The JVM was warmed up and measurements were taken. The mean time for 50 repeated runs for one query was calculated. Please note that beside the 50 repeating run, a warm up run had been done at first.

As can be seen in the Code 2-1, the process that is being timed involves the message containing the SQL query that is sent to the service; the service unpacks the message, runs the workflow that performs the SQL query and sends the resulting data back to the user in another message. The same SQL query is constructed each time in the loop.

There are two reasons for timing this specific section. Firstly, every relational database query using OGSA-DAI have this section in common. Secondly, besides this section, only some other smaller operations in terms of execution time are needed to complete a relational database query; these smaller operations includes connect to the relational database using JDBC, transforming the resulting data, etc. The OGSA-DAI version taking the smallest execution time would be chosen for this project.

As can be seen, the same code had been executed 51 times. The first time is for warming up the JVM and the last 50 times are used to calculate a mean time.
Two variations of the code shown in Code 2-1 were run with different SQL statements. In one variation, the SQL query retrieved 10 rows from the database, and in the other variation the SQL query retrieved 10k (10240) rows from the database. Both queries were executed 50 times after warming up the JVM and the time for each instance to run are plotted in Figure 2-2. A significant contrast of the performance between the two variations can be observed. Periodic peaks occur when querying 10 rows 50 times, while the time curve is quite constant throughout the 50 executions retrieving 10,000 rows.
In OGSA-DAI WSRF2.2 the plot for repeated executions of query for 10 rows shows periodic peaks. The reason for the periodic peaks observed when executing a query to retrieve 10 rows was thought to be due to the execution of the Java garbage collection. Further benchmarking was done regarding these periodic peaks.

In these benchmarkings, instances of query for different rows of record are executed repeatedly, and the average execution time and standard deviation of each instance are calculated. The test results are shown in Figure 2-3.

In Figure 2-3, the standard deviation is calculated for the 50 times of execution for each of the executions of one instance is plotted as a point; there are 13 points in this figure, each representing the instances querying 1, 100, 500, 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000 rows respectively.

Figure 2-3: OGSA-DAI WSRF2.2, standard deviation decreases as execution of the SQL query repeats.

It can be observed in Figure 2-4 that the more the rows are queried, the smaller the standard deviation. However, this cannot lead to the conclusion that the height of the peaks is larger if the number of rows retrieved is larger. This is because that for bigger queries the time will be greater so small variations in the time will proportionally be smaller and less significant and the vice versa.
More convincing evidence is a shrinking error bars of average execution times (base times) when the base time increases. This evidence is shown as Figure 2-5. In Figure 2-5, number of rows retrieved increases, hence the average execution time (base time) increases, however, the standard deviation, shown as the error bars, decreases. In other words, in OGSA-DAI WSRF2.2, the more rows that are queried repeatedly, the less the difference in each repeated execution of the query will be seen.

**Experiment 2: changing the heap size**

Further to the previous tests, efforts were made to verify that extra time taken, as is shown as the peak, by the execution of java garbage collection. To do this, the amount of memory available in runtime was increased and it was observed that this produced a slight delay in the execution of the garbage collection as shown in Figure 2-2 by the shift of the peaks to the right in the diagram.

In this test, the java heap size is set to 512 megabytes and 128 megabytes respectively, by running the java application that is partly shown in Code 2-1 with the –Xms and –Xmx parameters:

```java
java -Xms512m -Xmx512m <the_class_name>
```

**code 2-2: heap size set 512m**

and

```java
java -Xms128m -Xmx128m <the_class_name>
```

**code 2-3: heap size set 128m**

In this test, the SQL query is a data query for 1 row of record and this query is repeated for 50 times after the JVM is warmed up. The execution time for each of the executions with both java heap size settings are plotted in Figure 2-4.

![Figure 2-4 OGSA-DAI WSRF2.2, heap size increment decreases the peaks number](image)

From the testing result shown in Figure 2-4, it can be seen that the increment of the heap size shifts the peak by one phase. In this figure, when the heap size is 128 megabytes, there are 10 peaks in all the 50 repeated executed executions of the query for 1 row of record, and when the heap size is 512 megabytes, there are 9 peaks. This penomona shows the correctness of the assumption that the periodic extra time shown as the peaks is the used by the execution of the JVM’s garbage collection.

**Experiment 3: scaling with an increase of the number of rows retrieved**

The scaling of mean time when the number of rows of record is increased is investigated, as another phase of benchmarking of OGSA-DAI WSRF 2.2.
The test is done by querying different number of rows, determined by SQL ‘where’ clause. The number of rows used in this test were 1, 100, 1000, 5000, 10000. For each number of rows 50 queries were run, after the JVM was warmed up. The error bars indicates +/- half standard deviation.

![Graph showing execution time for different numbers of rows](image)

**Figure 2-5: OGSA-DAI WSRF2.2: mean time of 50 times of execution for queries for different numbers of rows of record**

The Figure 2-5 shows that generally the execution time increases with the number of rows of record that is queried.

To sum up, three experiments were performed for OGSA-DAI WSRF2.2. Two issues were investigated in these experiments. The first issue was to do with the periodic peaks observed in the execution of one query for small number of rows retrieved. It was discovered that the increment of the available memory for the JVM may decrease the number of the peaks, that said, the repetitive execution of the garbage collection takes a comparative large time when the number of rows of record retrieved is small. The second issue is to do with the scaling with the increase in the number of rows retrieved. It is discovered that generally, the execution time of one query increases with the increment of the number of rows of records that are queried.

In the next section, investigations are carried out using OGSA-DAI WSRF3.0.

### 2.2 The OGSA-DAI WSRF 3.0 testing

**Experimental setup:**

Experiments were run using WS-Core from the Globus Toolkit 4.0.4. Both the client side and the service side were run on the same laptop as previously and the experimental set-up was as before.

The timed section of the code is limited to a data request execution resource’s execution of a pipeline workflow. This workflow contains three activities: a SQLQuery activity a data service uses to retrieve data from a relational database; a TupleToWebRowSet activity that transforms a tuple that is output by the SQLQuery activity to a WebRowSet activity that is easy to handle by a java application; and a DeliverToRequestStatus activity that delivers the resulting data to the client side. The execution of the pipeline is executed 50 times and each execution time was recorded. The code used is show below.
Experiment 1: scaling with the increment of the rows of record that are queried

Using OGSA-DAI WSRF3.0 pre beta, the 50 times of repeated execution of a query could only be finished if the number of rows retrieved was less than 3000 for the machine that was used for these experiment. The test that produced the problem is Experiment 3 in this section.

In this section, a query is repeated 50 times as long as the number of rows used is less than 2000. The numbers of rows to retrieve was thus set to be 1, 50, 100, 200, 300, 400, 500, 750, 1000 and 2000. The mean time for each of the 50 times of repeated execution are calculated. The test result is shown in Figure 2-6.
From the test result it can be seen that in general, increasing the number of rows retrieved increases the execution time. This is similar to that in experiment 3 of the OGSA-DAI WSRF 2.2. It is still necessary to determine which version of OGSA-DAI is faster. This investigated in experiment 2.

The mean times for the different executions using a JVM heap size of 512 megabytes are plotted against the number of rows queried. It can be seen from the plot below that, unlike that is seen in Experiment 1 for OGSA-DAI WSRF 2.2, although a general increase can be seen, there is not apparent relationship between the number of rows to retrieve and the execution time. However, it can be observed that there are non-periodic peaks in this diagram.

![Graph showing execution time vs. number of rows for OGSA-DAI WSRF 3.0](image)

**Figure 2-7: OGSA-DAI WSRF 3.0, number of rows has trivial effect on execution time**

The comparison in the mean execution time for 50 repeated execution of one query is made. The numbers of rows retrieved by this query are 1, 20, 50, 100, 200, 300, 400, 500, 600, 700 and 1000 respectively.

![Graph comparing execution time for different numbers of rows](image)

**Figure 2-8: Comparison of execution time for queries that query different number of row of records**

From Figure 2-8, it can be seen that with each number of rows of retrieved in this experiment, the mean execution of execution of a query is shorter for OGSA-DAI WSRF 3.0 than OGSA-SAI 2.2.
This can be deemed to be an advantage that OGSA-DAI WSRF3.0 beta 1 has over OGSA-DAI WSRF2.2. However, as mentioned before, a disadvantage it has is that 50 repeated execution of a query can only be finished if the number of rows to be retrieved is less than 3000 for the machine used in these experiments. Experiment 3 shows the existence of this problem.

**Experiment 2: OGSA-DAI WSRF 3.0’s failure in retrieving large number of rows of records.**

For the laptop used in this experiment, querying 3000 rows 50 times could not be fulfilled, similarly for retrieving more than 3000 rows (4000, 5000, 6000), none of these reached the end either.

This issue was reported to the OGSA-DAI project team and this problem was resolved in the OGSA-DAI WSRF3.0 beta 2.

Querying 3000 rows, at execution run 43, produced the following error message:

```
A problem has occurred...
Error processing request; nested exception is:
    java.lang.OutOfMemoryError: Java heap space
```

Message 2-1: error message of OGSA-DAI WSRF 3.0 when executing repeated queries

The execution time of the queries that are done is plotted in the graph below.

![Graph showing execution time for 3000 rows](image)

**Figure 2-9: OGSA-DAI WSRF3.0 can not finish 50 rounds for 3000 rows**

We can observe a sharp increase before the termination of the run. After run 37, the execution time for each run increases all the way up, until run 42, which takes time about 7 times longer than the average time of all the previous runs, and the out of memory error occur in run 43. Memory leaking is thought to be the reason for this phenomenon. To reason the increase of the time, when the system is about to run out of memory, the paging increases, leading to the increased execution times, until the memory failure takes place.

**Experiment 3: changing the heap size**
If the available memory of JVM is increased the occurrence of the out of memory error would be delayed. Therefore, the java heap size is increased and the time for executing a query selecting 5001 rows was taken up to the point that the application terminates due to the memory error.

However, it can be seen from the plot that neither the individual execution time, the scaling, or the termination points are affected significantly by the different heap sizes, although minor differences (especially at the beginning and end) show that performance is relevant to the heap size. The different heap sizes set in this experiment are 64 megabytes, 128 megabytes, 256 megabytes and 512 megabytes.

As can be seen in the graph above, the performance plots are so similar that they are hard to distinguish from each other. Therefore, it can be seen that, unlike OGSA-DAI WSRF2.2, the heap size has hardly any effect upon the execution time for one query in OGSA-DAI WSRF3.0.

According to all the experiments that were carried out, a comparison is made between version 2.2 and version 3.0 pre beta.

2.3 Summary

As can be seen, OGDA-DAI WSRF 3.0 has a better performance than OGSA-DAI WSRF 2.2. Furthermore, unlike with OGSA-DAI WSRF 2.2, the java heap size has a trivial effect on the time of execution in OGSA-DAI WSRF3.0, and there are no periodic peaks in the experiments done using OGSA-DAI WSRF3.0, both of which mean that the OGSA-DAI WSRF3.0’s execution time is more predictable and independent with the setup of the machine used. Although there is some problem for the new version when a large number of rows are retrieved repeatedly, a situation that is not be expected in this project. Therefore, OGSA-DAI WSRF 3.0 is used as the middleware for this project.

3 Explorations with the basic data integration scenarios

3.1 Motivation and overview

OGSA-DAI establishes a platform on which users interact with databases by calling APIs. In this project, we explore several data integration scenarios in order to assess both the usability and performance of OGSA-DAI.
The rationale behind the design for these scenarios is to explore different data integration building-blocks and use the performance of these as a means to compare them, as well as specifying any difficulties / problems encountered in the process of implementing these.

The other topic in this experiment / exploration is the synchronization / delivery performance in OGSA-DAI. Different synchronization and delivery methods are benchmarked and the results from external projects are referred to, aiming at profiling and suggesting the best data transmission patterns for OGSA-DAI.

Towards the end, using the source code from the exploration scenarios and the conclusions from the profiling, a data integration application is made on top of the OGSA-DAI, in the hope of facilitating data integration to users.

Using a GUI application, a user constructs pipeline workflows operating across multiple databases with activities that either query a database or manipulate the resulting datasets, without having to worry about source / sink creation, intermediate delivery or synchronization.

3.2 Exploration of data integration scenarios

3.2.1 Introduction to the use case

The use case for these four scenarios is that a user wants to retrieve data from two databases, db1 and db2, that are geographically distributed, using independent queries. Then they want to exert some algorithm collectively on the datasets she retrieved.

Because the performance relating to the different transmission directions is what is being tested, a simple test SQL JOIN Query operation is chosen as the integration operation, as shown below:

```
1. USE DB1;
2. SELECT * FROM {
3.   (SELECT * AS t1 FROM DB1.T11 WHERE ID <= a_value)
4.   JOIN
5.   (SELECT * AS t2 FROM DB2.T21 WHERE ID > 10000 - a_value)
6. }
7. AS t0
8. WHERE ID <= 10000;
```

**Statement 3-1: fake query denoting data integration**

The line number at the beginning of each line and the AS keyword are there for reference and for reading convenience.

This query will not work normally because it is querying tables across geographically distributed databases; nevertheless, it is clear enough to specify the data integration operation in the tests.

Line 3 is the SQL Query on table T11 in database DB1, the resulting table is named t1. In line 5, database DB2’s table T21 is queried and the result is stored in table t2. To make the potential amount of transmission (i.e. the size of the resulting tables) close, the number of rows to be pulled from the database is set to be same. In fact, the two original tables are the same - the numbers of rows of both tables is 10000. In the different tests, the “set_a_value” will be set to different values, for the sake of doing a scaling benchmark.
The structure of a test Grid is shown as Figure 3-1. The three data services can communicate with each other.

There are two databases that a user is interested and three data services exposing both of these databases. The user can use data service 0, data service 1 and data service 2. Data service 1 exposes database db1; data service 0 exposes both database db1 and db2; and the data service 2 exposes database db2.

The data flow of the user’s data integration is depicted in Figure 3-2: data flow in the test case. The datasets are pulled out from database db1 and database db2, before the join operation takes place. The join operation can take place on either one of the data services or the client side.
3.2.2 Experimental setup

- crockett.epcc.ed.ac.uk [23] (“crockett” for short) is used. The hardware specification for crocket is shown in the following list: [23]

<table>
<thead>
<tr>
<th>computing node in crockett</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dell PowerEdge SC1435</td>
</tr>
<tr>
<td>Two dual core 1.8GHz Opteron processors.</td>
</tr>
<tr>
<td>4Gb of shared RAM.</td>
</tr>
</tbody>
</table>

OGSA-DAI WSRF 3.0 (pre beta) was installed on crockett and for the JVM J2SE 1.5.0_12 was used.

The Globus toolkit 4.0.4 was installed on crockett, deployed with OGSA-DAI 3.0. The Globus containers used ports 8080 and 8081 respectively, as two independent data services would be required. In the rest of this specification, we call the data services hosted by these two containers, DRES1 and DRES2 respectively.

As can be seen, rather than having two services located separately, we are hosting two data services on one machine. This has an impact on performance, such as diminished parallelism and network delays, so with this set up only the complexity difference between implementations can be evaluated.

A machine, “omiiuert”, hosts MySQL 4.1 which hosts an “ogsadai” database with two tables: these are the same as “littleblackbook”, but named as “db1”and “db2”.

As can be seen this situation differs from that originally presented where we should have two databases located separately, each having its own table which has now been simplified to one with two tables stored in the same database. The rationale for this simplification is that (1) the complexity of the
implementation for the use case will not be affected and that (2) the effect in the diminishment of the transmission concurrency is trivial, because the size of data to be transmitted is small.

At the client side, the OGSA-DAI 3.0 is installed as well. The purpose of this is to use its client toolkit to fulfill the data transformation, such as transforming a tuple coming from a database to a WebRowSet object. For JVM, J2SE 1.5.0_12 is used.

3.2.3 Scenario 1

Introduction

This scenario portrays the commonest usage of OGSA-DAI for retrieving data. In this scenario, two OGSA-DAI data services are hosted at different URLs exposing different databases. In order to retrieve the data, a user needs to construct two pipeline workflows to send to the data services. The data services will then retrieve the data for the user. The network topology of client, services and databases are shown as Figure 3-3: network topology in test scenario 1.

As shown, in this scenario, the client sends out two workflows each dictating a DRES to pull data from their associated tables with the two resulting tuples being sent to the client. Having two tuple in possession, the client can store the tuples in a local database and write, for example java code using JDBC, an application to perform the desired SQL UNITE operation. The sequence of execution of the client side and the two data services is shown in Figure 3-4.
**Figure 3-4: the execution sequence for the client and the data services in scenario 1**

**Implementation**

The pseudo code for client is shown as Code 3-1. As can be seen, the implementation algorithm is straightforward: while there is still data that needs to be retrieved, the data service that exposes this is sent a workflow and the resulting data is retrieved.

```java
If (there are more data services exposing databases to be connect to for data){
    Construct a pipeline workflow (service’s Data Request Execution Resource, query, database’s Date Resource file)
    Send the executed workflow
    Retrieve the data from the service
}
```

**Code 3-1: pseudo code for the timed section of code in scenario 1**

The code in Code 3-1 can be executed concurrently. To implement this, each loop in the code should be a Java Execution Service submitting an instance doing the exact same process as is shown in the loop in Code 3-1. The code for this concurrent execution is Code 3-2.

```java
While (there are more data services exposing databases to be connected to ){
    construct an Instance of a class (URL, Data Request Execution_Resource);
    future[index_of_an_instance] = exutionService.submit(this_Instance)
}
exutionService.shutdown();
private class (URL, Data Request Execution Resource){
    Construct a pipeline workflow (service’s Data Request Execution Resource, query, database’s Date Resource file)
    Send the executed workflow
    Retrieve the data from the service
}
```

**Code 3-2: concurrent execution for the timed code in scenario 1**

As a means to evaluate the performance of OGSA-DAI in scenario 1, the Code 3-2 is executed repeatedly for 10 times after the JVM has been warmed up. The reason for timing this specific section of the code is that it contains the basic operations are required to implement data integration in scenario 1. Variations of this code section are made, by putting in different SQL query statements in the SQLQuery activity in the pipeline. Tests have been done on the execution of these variations. The test result and analysis is shown below.

**Performance scaling with incremental rows to retrieve**
Figure 3-5 shows the performance scaling of OGSA-DAI in scenario 1, when increasing the numbers of rows retrieved from the two databases. The number of rows retrieved from each database is the same in this test.

![Figure 3-5: test result for scenario 1](image)

As can be seen in Figure 3-1, the execution time increases as the number of rows to retrieve in scenario 1 increase.

### 3.2.4 Scenario 2

**Introduction**

There will be cases where a data service exposes multiple databases. In this case if a user wants to get data from multiple databases and perform a SQL UNITE, they only need to construct one workflow which performs, not only the retrieving of data from both databases, but also does a SQL UNITE upon the resulting datasets to get the resulting data. The network topology is shown as Figure 3-6.

![Figure 3-6: network topology in scenario 2](image)
Implementation

The sequence of executions of the client and data service in scenario 2 is shown as Figure 3-7. The user needs to construct only one workflow that will be sent to the data service to be executed. This workflow consists of three activities. The CombineActivity, which collects data needed for the SQL UNITE, a TupleToWebRowSet activity and a DeliverToRequestStatus activity. This CombineActivity consists of two parts. The first part retrieves the data by specifying the databases that are to be queried and the SQL query statements to be performed on each database, the user defines what data should be stored in the data service; the second part is performing the SQL UNITE.

![Figure 3-7: the sequence of executions of the client and data service in scenario 2](image)

Performance scaling increasing the number of rows to retrieve

For the performance benchmarking, the timing starts when the client sends out the workflow and stops when the resulting data is returned. The timed section of code is as shown below.
Repeat this 11 times and calculate the average time of the last 10 times. The first time of execution is used for warming up the JVM.

The scaling is investigated by comparing the average execution time across variations in the timed code section. The variation is different the number of rows retrieved (e.g. 64, 128, 256, 512, 1k, 2k, 3k, 4k, 5k, 6k, 7k, 8k, 9k, and 10k). The number of rows retrieved by both data services is the same. The test result is shown as Figure 3-8.

Please note that in this experiment, the timed code section is repeated 11 times, rather than 51 times, in which case the scaling can only be taken up to about 3000 rows retrieved on crockett.

**Code 3-3: pseudo code for the timed section in scenario 2**

```java
PipelineWorkflow pipe = new PipelineWorkflow(); // construct a pipeline workflow
CombineActivity combine = new CombineActivity(); // construct the combine activity
// add the data resource id and SQL query statement for the first database
Combine.addDrId(first_drid);
Combine.addQuery(first_query);
// add the data resource id and SQL query statement for the second database
Combine.addDrId(second.drid);
Combine.addQuery(Second_query);
TupleToWebRowSetCharArrayActivity ttw = new TupleToWebRowSetCharArrayActivity();
DeliverToReuqestStatus dtrs = new DeliverToReuqestStatus();
...
// Add activities into the "pipe"
// construct the DataRequestExecutionResource - drer
For (int I = 0; I < 11; I ++){ // the first run is JVM warm-up
    // start timing
    Drer.execute (pipe...)
    // stop timing
}
```
As can be seen in Figure 3-8, the execution time of the timed section increases as the number of rows to retrieve in scenario 2 is increased.

3.2.5 Scenario 3

Introduction

In this scenario a user wants to use different data services to retrieve the data and then another data service to perform the SQL UNITE operation. The network topology is shown in Figure 3-9. The DRES1 exposes db1 and the DRES 2 exposes db2. The client sends a workflow to DRES0. DRES 0 retrieves the resulting data from DRES1 and DRES2 and returns the SQL UNITE data back to the user.

**Figure 3-9: network topology in scenario 3**

Implementation

An activity is deliberately written for DRES0 for this scenario. This is activity is call CollectiveSQLQueryActivity. The user input the data resource IDs of the databases and the SQL queries for each, the DRES0 will in turn send workflows contain activity querying db1 and sb2 with regarding SQL query statement to DRES 1 and DRES2. Then the DRES 1 and DRES 2 retrieve the data according to the SQL query statement and return the data back to the DRES0, and the DRES0 performs the SQL UNITE operation and send the resulting data back to the client.

Performance scaling with incremental rows to retrieve

**Figure 3-8: performance scaling with increment of data of scenario 2**
For the performance benchmarking, the timing starts when the client sends the perform document and stops when the web row set is received.

Repeat this test 11 times and calculate the average time of the last 10 runs. The first time of execution is used for warming up the JVM.

The scaling is tested by comparing the average execution time across various executions, querying 64, 128, 256, 512, 1k, 2k, 3k, 4k, 5k, 6k, 7k, 8k, 9k, and 10k rows.

![Figure 3-10: test result for scenario 3](image)

As can be seen in Figure 3-10, the execution time increases as the number of rows to be retrieved is increased in scenario 3.

### 3.2.6 Scenario 4

**Introduction**

The network topology in this scenario is similar to that of scenario 3, except that DRES 0 is obsolete and the DRES 1 acts as the broker DRES, and it is still associated with db1. Its data flow is shown in Figure 3-11.

![Figure 3-11: network topology in scenario 4](image)

**Implementation**
As can be seen from Figure 3-12 all the executions of DRES0 in the previous scenario migrate onto DRES1.

Figure 3-12: sequence diagram for scenario 4

Performance scaling with increasing numbers of rows to retrieve

For the performance benchmarking, the timing starts when the client sends the master perform document and stop when the web row set is returned.

The test is repeated 11 times and the average time of the last 10 runs is calculated. The first execution time is used for warming up the JVM.

The scaling is tested by comparing the average execution time when the numbers of rows retrieved are 64, 128, 256, 512, 1k, 2k, 3k, 4k, 5k, 6k, 7k, 8k, 9k, and 10k.
As can be seen in Figure 3-13, the execution time increases as the number of rows retrieved increases in scenario 4.

A comparison between the times is shown in Figure 3-14.

As can be seen, the performance of scenario 1 is the best of the four, and the performance of scenario 2 is the second best. The performance between scenario 3 and 4 are similar while the performance of OGSA-DAI in scenario 4 is slightly better than that in scenario 3 at some places in the diagram. The reason for this is thought to be that in scenario 3, the data service carrying out the integration operation needs to receive two data sets, therefore the overall execution time is the sum of the computation time and transmission time of the data service which is slower. This situation is expressed as Note 3-1.

\[
\text{Execution time} = \max (\text{computation time and transmission of the one of the data service, computation time and transmission time of the other data service})
\]

**Note 3-1: the execution time in scenario 3**

In scenario 4, when the computation that generating the a larger data set takes place on the machine that carries out the SQL UNITE operation, the longer transmission time is saved, therefore, the execution
time of the whole process equals to the smaller sum of computing time and transmission time of the two
data services in the scenario 3. This situation is expressed as Note 3-2.

\[
\text{Execution time} = \text{computation time and transmission time of the data service which carries out only computation and transmission but not SQL UNITE operation}
\]

Note 3-2: the execution time in scenario 4

3.3 A recap of data integration scenarios and data integration use cases

From all the test cases for the four data integration scenarios, three conclusions can be drawn:

1. Topology one is not an ideal approach because
   i. the load on the network is the heaviest, by transmitting the largest amount of data;
   ii. it produces the heaviest computational load at the client-side. All the data integration
       operations are carried out by the client side machine, therefore,
   iii. the client side needs a Relational database to carry out the execution of any SQL queries.

2. Topology two is the fastest approach for data integration, so whenever multiple databases are
   exposed by a data service, only this service should be used for the integration

3. Topology three is potentially slower and produces a heavier bandwidth load than topology four, as
   it has more data transmissions. The reason is discussed in the previous section.

These scenarios are the building blocks for more complicated networks of data services, which is
exemplified in the following section.

3.4 Location tagging algorithm for the smallest subtrees

As has been discussed, a tree data structure is ideal for representing an integration operation network and
and can be used to construct transmission optimised workflow. In the remaining part of this section, we will
discuss the various basic data service network Operation Patterns and their performances and will
eventually derive several algorithms detecting the Operation Patterns and determine the data
transmission direction before performing integrating operations.

In this section, we abstract the three basic data integration scenarios into tree structure and into different
Operation Patterns. The four scenarios represent the fundamental network settings with which one can
perform integrating operations upon data retrieved from two databases using OGSA-DAI. Scenario one,
where the integration operations have to be done at client side after original tables are retrieved from the
databases, is not included in discussion.

Let us recap the operation used to integrate the data. As the performance relating to different
transmission direction is what has been focused on and tested, a simple SQL JOIN Query operation is
chosen as the integration operation:
Statement 3-2: the pseudo query for the test scenarios

We are familiar with this “test query”, introduced in section 3.2.1. To visualise this operation, using the approach previously introduced a DAG can be draw:

```
1. USE DB1;
2. SELECT * FROM {
3. (SELECT * AS t1 FROM DB1.T11 WHERE ID <= a_value)
4. JOIN
5. (SELECT * AS t2 FROM DB2.T21 WHERE ID > 10000 - a_value)
6. }
7. AS t0
8. WHERE ID <= 10000;
```

![DAG for the test scenarios](image)

**Figure 3-15: the DAG for the test scenarios**

Also, this DAG can be converted to an Operation tree:
The Operation tree depends on the user’s specification of the data integration. In this thesis, we call all the original tables, the operations and their dependencies a “Query Flow”. One user specification for data integration results in only one Query Flow.

However, due to the geographic distribution of databases and data services, one Query Flow can be executed in different ways, i.e. the operations are possibly taken by different data service when executing the same Query Flow. At the time of execution, the operation pattern of this integration is the Operation tree of this integration whose nodes are tagged with service’s locations (URLs) which can be determined by the routing algorithm.

The routing algorithm is drawn from the three basic topologies in the performance test in the first half of the project. In the following section, we review the three topologies tested and deduce the algorithm.

First of all we know that an original table can only be accessed by the service that is exposing it.
Rule 1: In a Query Tree, the leaf nodes, which represent the original tables, are annotated with the location (URL) of the service exposing it.

Note 3-3: rule 1 of subtree root tagging

To retrieve the best performance, all the nodes in the operation tree need to be tagged. The work can be done recursively. The next step is to tag the ancestors of every smallest connected tree in the operation tree. But first we need the rules for tagging – the rules are made from the conclusions of the three basic topologies.

In topology two, all the queries to the individual database are launched by DRES0 and the SQL JOIN on the retrieved table is done on this service. This service exposes the databases that the user wants data from.

To depict the Operation Pattern for this topology, simply add the location information into the Operation tree. The same colours are used to indicate same location for the operations. The operation tree is drawn in

Figure 3-18.

Figure 3-18: network topology and root tagging for test scenario 2

According to the tests for the scenarios, this is the fastest topology of all the three topologies in which the same data integration is deployed. To conclude this:

In the smallest connected subtree, if all the leaves have the same location, annotate their ancestors with this location.

Note 3-4: note for tagging for basic topology 2

Examples follow for the application of this rule. In Figure 1-8, O1 should be tagged with the location of T12 and O4 should be tagged with the location of T21, T32 or T42. Using same colour to indicate same location for the operations, the trees are drawn as in Figure 3-19.
For topology four, the operation tree nodes are drawn in green (light grey) if they correspond to DRES1, red (dark grey) if its location is the same as that of DRES2 in the tagged tree shown at the right of Figure 3-20.

The only difference between topology three and topology four is that the operation O0 is executed by a third party in topology three. Use yellow to denote this third data service.
Rule 2: In the smallest subtree, annotate the root with a URL where no leaf locates will not achieve optimised transmission time.

Note 3-5: rule 2 for root tagging

Examples for the application for Rule 2 are shown in Figure 3-21. In the example shown in Figure 1-8, O4 should be tagged with either the location of T21, T32 or T42. Therefore, there are four possibilities: O4 has the same location as T21, O4 has the same location as T32 or O4 has the same location as T42.

In black and white printing these diagrams will be indistinguishable. In the leftmost image, O4 and T21 are green, T32 is blue and T42 is yellow. In the image in the middle, O4 and T32 are blue, T21 is green and T42 is yellow. In the rightmost image, O4 and T42 are yellow, T21 is green and T32 is blue.

![Figure 3-21: instances of tagging in the example](image)

However, this rule does not give a precise regulation for the tagging of the root of a smallest subtree. To extend this rule, consider the topology chunked out from Figure 3-22.

According to Rule 2, to achieve the best transmission time, O2 should be tagged with the location of DB1, and O4 should be tagged either with the location of DB1, DB3 or DB4.

![Figure 3-22: an exception for rule 2](image)

Suppose the O2 is tagged with DS1’s URL and O4 is tagged with DS3’s URL. For O2 and O4’s executions, there will be two data transmission. Then a smallest subtree is left after degradation of this
operation tree. To tag O3, examine all the leaves in the tree. It can be seen that in this case O3 should be tagged with DS1, DS2 or DS4’s URL. For the execution of O3, two data transmission will be carried out. In all four data transmission are carried out in this case.

However, if T32 and T42 are transferred to DB1 when O4 is performed, there will be one data transmission less.

Therefore, another rule can be drawn:

**Rule 2’**: In the smallest connected subtree, annotate the root with a URL where the largest number of leaves locate. If there is more than one such URLs, put them in a set U. Get the set of all the determined root annotations of the neighboring smallest connected subtrees D. Align different elements in D by the occurrence of the element in decrease order in array A, and if two elements have the same times of occurrence, align them in random order. Go through A and find the first element F encountered that belongs to U. Annotate the undetermined root with F.

**Note 3-6: rule 2’ for tagging**

The problems with Rule 2’ are that:

- The time complexity is $O(n^2)$, so when the operation tree gets very large this algorithm is expensive.
- This rule does not guarantee the fastest computation.

Therefore, Rule 2 is still recommended and used by this project.

The size of data, which is also an important factor for the transmission time, is not taken into account in this algorithm; therefore, other than for an optimised transmission count, the tagging algorithm per se does not guarantee best transmission time.

However, this algorithm makes possible the parallel execution of the query flow and suggests a naïve scheme of data transmission among the parallel execution units.

## 4 A data transmission count optimised algorithm for OGSA-DAI

Previously in this project, the feasibility and performance of several data integration scenarios were explored. The network setups in the exploration scenarios consist of the client, the databases, the data services exposing the databases and the data services that perform the integration operations on the data retrieved. The topologies of the networks are set to be as simple as possible, so as to represent the building blocks of more complex data integration networks in the real world.

Several conclusions were made on the performance data after various experiments, and these were summarized into three rules for tagging the roots of smallest subtrees in an operation tree. These rules depict how to determine the data service which should perform the first integration operation after retrieving the data in order to realize a minimal data transmission count when the parallelism of operations is guaranteed.

To make the rules applicable to computer applications, the concepts of DAGs (Directed Acyclic Graphs) and tree are employed and simple algorithms are designed to transform a DAG to an operation tree. There is one algorithm describing how to construct a DAG for one’s data integration specification, one algorithm describing how to transform the DAG to a tree, one algorithm describing how to tag the nodes in the smallest subtree to reflect the job allocation, and one algorithm describes how to construct an OGSA-DAI pipeline workflow using the YoungestFatherQueryActivity and CollectiveQueryActivity from the tagged smallest subtree.
The data movement is represented in ascendant – descendent relationship, in which the ascendant is the destination of movement, while the descendent is the source of data.

The neighboring smallest subtrees are the largest unit within which a transmission optimised schedule can be made. In the design of the new databrowser, an application implementing the algorithms introduced in the previous section, the tagging is done at the client side at first, and then, according to these taggings, the activity workflows are constructed and launched to different OGSA-DAI services.

The parallel execution algorithm will be introduced with an example use case. Before this, several important preceding processes are recaptured.

First, the user draws a DAG depicting the requirements of the data integration needed. The DAG will be transformed into an operation tree at client side.

Second, according to rule 1 and rule 2, the operation tree is tagged. One of the possible tagging of the operation tree is shown in Figure 4-1.

![Figure 4-1: tagging the roots of the smallest subtree in the example](image)

Third, the user client sends the tagged operation tree to an OGSA-DAI data service, the broker.

According to the taggings, the broker launches workflows at each of the roots with the smallest subtrees, making them execute the lowest level data retrieve and computations. Meanwhile, the broker will launch workflows creating a data sink and a data source in the roots. In a data service represented as a root, the resulting data will be stored in the data sinks.

Hence is when the parallel execution of the data integration among roots of smallest subtrees takes place.
When a root node, N, in a subtree finishes its integration operation first, there are three possible node patterns at the same level where this node is located, if the data service represented by N is called S then:

1. All other nodes (1 or more) are original tables. In this case, the broker tells S to get data from the data services exposing the databases and integrate the resulting data sets along with what is in its data sink. Store the new resulting data into its data sink.

2. The other nodes are assigned to a data service, S0, generating integration result that are not yet finished with 0 or more original tables.

In this case the broker tells S to send the data from its sink to the source of S0; meanwhile the broker tells S0 to get data from data services exposing the databases and integrate the resulting data sets with what is in S0’s data sink and data source.

3. The other nodes consist of two or more data services generating integration result that are not yet finished and 0 or more original tables. In this case choose any one of these data services as S0 and execute the process stated in point 2.

The broker traverses the operation tree in an order in which, from left to right, the smallest subtrees are read and executed. The parallel execution terminates when the root of the operation tree finishes executing and the resulting data is retrieved from its data sink by the broker, before the broker sends the resulting data back to the client.

5 Further work

Due to the limited time available to this project, there is still some work to be carried out in the future.

The performance scaling against the increasing number of databases in the three basic data integration network topologies has not been benchmarked. As can be seen, Rule 2 assumes that topology 4 has the best execution time, however many leaves there are in the smallest subtree.

The algorithm has not been implemented in code for this project, therefore the performance of this algorithm is not known. In fact, to synchronise amongst OGSA-DAI services, using data source may not be the best approach. These issues need to be investigated by experiments.

As a first stage of data integration exploration using OGSA-DAI, the new data browser does not provide functionality to launch a DAG producing multiple final results, though its implementation is to simply attach a bulk send operation to the results, by the client side application.

6 Conclusions

Today there are many complex data integrations needed for biology and astronomy research. Meanwhile, in the Grid resources can be used regardless of their physical location.

This project aimed to make an application brokers complex data integration in a Grid context. The basic topology of data integration using OGSA-DAI is benchmarked and compared. Algorithms are developed according to the benchmarking results, to parse the users’ complex queries into workflows and broker the workflows executed across multiple OGSA-DAI data services. The OGSA-DAI data source is used to for overlapping the computation and data transmission.

7 References


14. G. Taffoni. When DBs met the GRID The Grid Data Source Engine. 2006


16. The DataMiningGrid project. The website of DataMiningGrid. 2007. Available from: http://www.dataminingGrid.org/


23. The specification of the machine used in the testing scenarios:
http://support.omii-europe.org/index.php/Edinburgh

http://www.globus.org/wsrf/faq.php#wsrf1

http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#defs