

Databases in Grid Applications: Locality and Distribution

Paul Watson

School of Computing Science, University of Newcastle, Newcastle-upon-Tyne, UK
Paul.Watson@newcastle.ac.uk

Abstract. This paper focuses on two areas that experience in building database-oriented e-science applications has shown to be important. Firstly, methods of promoting data locality are vital due to the high cost of moving data in service-based distributed systems. Databases provide an excellent basis for achieving this due to their potential for moving computation to data. The paper also describes a new infrastructure that further promotes locality by enabling service-based computations to migrate to data. Secondly, the ability to combine information from a set of distributed databases has proved invaluable in many applications. The paper describes the design of an adaptive distributed query processing system that is able to exploit facilities offered by an underlying grid infrastructure. In addressing these two areas, the paper gives an overview of some of the generic components that have been designed to simplify the integration of databases into e-science applications.

1. Introduction

The 2001 paper “Databases and the Grid” ([1]) argued that databases should play an important role in e-science applications. This was because they offered a way to manage and publish the deluge of structured data that the large numbers of new grid computing [2] projects were planning to generate. Four years on, it is clear that e-science has encouraged the publication of information on an unprecedented scale. The cost of data collection, and the potential value of data to large groups of scientists is so great that funding bodies are now insisting that projects make their data available for others to use. The days in which much scientific data was stored on the “C:” drive of a researcher’s computer, unavailable to others and likely to disappear at the end of the project, are now thankfully coming to an end.

The 2001 paper argued that database servers provide a wide range of facilities that both publishers and consumers can exploit in order to meet their requirements, including the ability to manage very large quantities of information, security, querying, update, concurrency control, high availability, manageability, resource control, versioning, schema evolution and change notification. However, it went further and argued that simply encouraging the use of databases was not enough, and that it was necessary to provide technology to reduce the complexity of making databases available through a service-based interface (now that Web Services is becoming the most commonly used distributed systems technology for building grid applications), and to

remove the need to write bespoke consumers for every database to be accessed by an application. Consequently, the paper argued that there were advantages in defining and making available implementations of a generic wrapper for exposing a database as a service.

It may be argued that databases should not be directly exposed at all to external clients. It is certainly the case that when databases are made available over the web, the database itself is almost always hidden behind an application that presents only a restricted set of functionalities. There are often good business reasons for this: organisations present those functionalities that are most likely to benefit them in their aims. So an on-line bookstore will present ways of keyword searching and viewing a customer's shopping basket. But, it will not allow arbitrary queries against its databases because they might negatively affect its business. e.g. by making it easy for another store to download information about the prices of all the books so they can then set their own prices accordingly.

This approach of offering only restricted functionality could be taken by e-science systems: the owners of data could decide on a particular set of functions and make only those available, rather than provide a general query interface. This approach is adopted by some e-science systems (especially those that provide a web interface), but it is clear that this can be restrictive as it requires the data owners to guess in advance how scientists may wish to use the data; e.g. what patterns they will search for, and how they will choose to combine data from multiple databases. For file-based scientific data, the predominant solution to the equivalent problem has been to make all the data available for copying (e.g. by FTP) to a client's local disk from where it can be processed in whatever way is required. However, this approach is generally not feasible for databases due to the costs of transferring large amounts of data (some scientific databases hold Terabytes of data), of installing and configuring a local database to hold the data, and, of keeping the local database synchronised with the remote database. Therefore, if data publishers do not wish to restrict the potential uses of their data, they will need to make it available by storing it in a database, and then exposing generic functionalities - in particular query processing - to external clients.

In this paper, we give an overview of some work that has had the aim of easing the construction of efficient, database-oriented, e-science applications. It focuses on two main areas that experience has shown to be important. Section 2 describes some approaches to promoting data locality given the high cost of moving data around in service-based distributed systems. It explains how the use of databases provides an excellent basis for achieving this, and introduces a new infrastructure that further promotes locality by enabling service-based computation to migrate to data. Section 3 then considers the integration of information from a set of distributed databases. This includes the design of an adaptive distributed query processing system that is able to exploit facilities offered by an underlying grid infrastructure. In addressing these areas, both sections give an overview some of the generic components that have been designed to make it easier to integrate databases into e-science applications. Finally, Section 4 draws conclusions and attempts to look-ahead to research and industry trends that are likely to have implications for future work on databases and e-science.

2. Promoting Locality

One of the main lessons that have been learnt from the experience to date of designing and deploying grid applications based around databases is that data movement is very expensive, and should be avoided where possible. While this is a general rule to be followed in the design of distributed systems in which there is the potential for large amounts of information to be moved over networks [3], the problem is accentuated in grids due to the fact that exchanging messages between Web Services is very expensive. Currently, the main expense is due to the fact that data held within a service (e.g. in a database) is converted to and from a character-based representation of XML so that it can be packaged in a SOAP message and sent between services. This imposes a CPU cost for converting to and from the XML. It also results in a large increase in size between an efficient representation within a service (e.g. within a database) and the tagged, character-based XML representation, incurring increased transfer costs as more bytes are shipped. A further overhead is imposed if messages are encrypted by the commonly used WS-Security protocol due to the cost of converting the XML into canonical form before encryption.

In fact, it is not actually necessary to represent data in character-encoded XML in order to send it between services. SOAP is defined as an Infoset [4], so allowing more efficient representations between services [5], though today it is very rare to find services that support them.

Given these overheads, it is very important to minimise both the number of messages exchanged between services, and the amount of data contained within them. In general, when designing service-based applications, to minimise the number of messages exchanged it is very important to design services that operate at as large a granularity as possible. Currently, there are two approaches to designing a Web Service, one of which has a tendency to lead to poor service design (ironically, this is the one best supported by tools). The approaches are:

1. Take an existing object-oriented program, select the methods that you wish to expose remotely and make those the operations of the Web Service. This is supported and encouraged by many Web Service tools, which will automatically generate Web Service interfaces (WSDL) from methods selected by the user. Similarly, client-side tools allow consumers to bind to operations offered by a remote service as if they were object methods, and perform “remote method calls” on them. This has the advantage that the programmer of the client and server do not need to know anything about web service interfaces (WSDL), or SOAP messages as those are automatically generated. However, the grave danger is that this approach leads to fine-grained operations that are suitable for efficient object-oriented programming within a service or possibly over a LAN, but are hopelessly inefficient for building internet-scale distributed systems, given the overheads of SOAP and the cost of transferring data between remote services.

2. First, design the messages that will be exchanged by the service. A key aim should be to design messages that encourage large-granularity, inter-service interactions. For example, it is better not to design messages that encourage a consumer to get results from a database one at a time if it is possible to batch data so as to reduce the total number of inter-service interactions. Once the messages have been designed then services can be written to generate and consume them. This approach requires

more thought and implementation work for programmer than does (1) but it leads to more efficient services and distributed applications.

In data-oriented applications, the main way to minimise both the number of message exchanges and the amount of data exchanged is to try to avoid moving data as much as possible through moving the computation to the data, rather than *vica versa*. Data services built around databases have tremendous potential to apply this principle as they allow clients to send queries the database for processing. This is in contrast with file-based services in which, except for some exceptions [6], it is usually necessary for the client to copy the data locally (e.g. by FTP) for processing.

We now consider three components that exploit database technology, and have been designed on the principle of reducing data movement where possible. These are the OGSA-DAI database service, the SkyServer MyDB, and the Active Information Repository.

2.1 OGSA-DAI

The OGSA Data Access and Integration project (OGSA-DAI) is a collaboration between groups within EPCC, the UK National e-Science Centre, IBM, Oracle and the Universities of Newcastle and Manchester. Since 2002, it has provided a Web Service database wrapper that supports common operations such as querying [7].

Consumers interact with the service through *Perform* documents that encapsulate a set of activities. Supported activities include queries (e.g. SQL queries for relational databases and XPath queries for XML datastores) and a variety of methods for delivering the results (e.g. store locally for later access or return the result to a 3rd party). By allowing consumers to pack a set of activities into a single interaction with the service, OGSA-DAI encourages the large-grained interactions that are necessary for efficient, service-based distributed computing. The activity mechanism is extensible, so service providers may define their own new activities for particular tasks (e.g. a text database could offer specialist text-searching activities). Interactions between consumers and the database service can be secured, e.g. by GSI [8]) or WS-Security [9].

OGSA-DAI therefore simplifies the process of publishing data. It also assists consumers by providing a common interface to a database that is independent of the specific DBMS product that has been wrapped (the main exception to this is in the different forms of SQL supported by different vendors; however our experience is that the problems can be largely mitigated by sticking to SQL standards widely supported across the database vendors). As will be seen in Section 3, this also simplifies the task of integrating data from multiple databases. The DAIS [10] working group of the Global Grid Forum is attempting to set a standard for exposing data through a Web Service interface. Ideally, the database vendors would themselves agree on, and support, such a standard. This would remove the need for third parties to provide wrappers for each different DBMS, and would also open the way for internal performance optimisations, e.g. in generating XML result sets directly from the internal representation.

Building on the foundation offered by OGSA-DAI for sending a set of activities to a database server in a single message, there are other steps that can be taken to further minimise the amount of data that is transferred. Stored Functions and methods (in ob-

ject and object-relational database servers) allow users to define computations that are executed in the database server when called from within queries. These can often be used to process or filter results in the database engine so as to reduce the amount of data returned to the client [11]. This is almost always more efficient than to have the database service return the result of a query to the consumer and then have it process the result. There are two exceptions: a) where the method/function call results in a result set that is significantly greater than the result set before the method is applied and b) if the database server is overloaded then the overall performance of an application may be slowed by the cost of executing the computation in the DBMS rather than on an external compute server.

Stored procedures can also be used to encapsulate a set of operations or queries at the database server so that they can be repeatedly called by a consumer specifying only the identifier and any parameters.

2.2 Storing Query results in the database

In many cases, the result of one query will later be the subject to further query processing (for example to further filter information), possibly by another client. Allowing query results to be stored back in the database service, rather than returned to the client, has the potential to reduce the number of times when data is transferred between the database service and the consumer. It can also reduce the load on the database server by preventing the need to re-execute queries to re-generate results. Finally, it also removes the need for the consumer to devote resources to storing the (potentially large) results of queries. A successful pioneer of this approach is the MyDB facility in the SkyServer [11] which gives users their own logical space in the database to store the results of queries.

2.3 Providing computational resources close to the database

Despite the opportunities to exploit stored procedures and methods in queries, it is not always possible to perform all computation within the database service itself. For example, it may be necessary for a complex piece of software to analyse the data produced by a query. In general, this usually means that the consumer sends the query to the database service and receives in return the results that are then sent to an analysis service. This incurs the expense of transferring potentially large amounts of data from the database service to a remote machine for analysis (in practice, it may well travel indirectly, via the consumer that sent the query to the database server, so incurring extra data transfers).

We have been investigating a way to address this problem by deploying database servers closely coupled to local compute resources. In practice, this usually means deploying the DBMS on one or more nodes of a cluster, while making the other nodes available to run services that can query the database and analyse the results. We call this combination of database server and local compute nodes an Active Information Repository (AIR) [12].

An example application is shown in Figure 1. A bioinformatics service analyses data from a database that is made available as a Web Service through an interface such as OGSA-DAI. The analysis service sends a query to the database and receives back from it a large quantity of data. It then processes this data and sends a small result set back to the consumer (the thickness of the arrows indicates the quantity of data transferred).

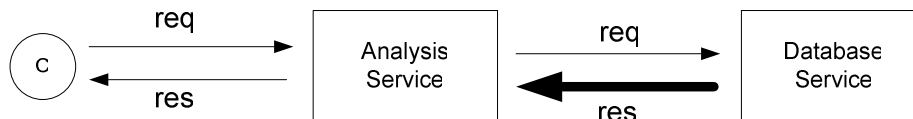


Fig. 1. Interactions between an Analysis Service and a Database Service

In order to avoid transferring large amounts of data over long distances, it is advantageous to deploy the analysis service close to the database on which it operates. Therefore, the database provider may deploy an AIR, with a cluster of compute servers close to the data. This raises the question of how the owner of the analysis service can deploy it on the AIR so that it is close to the database. If the service is created by the owner of the AIR then it may be straightforward for them to do so, but in e-science systems, remote scientists will invariably continually be devising new ways of analysing data, and it would be advantageous for there needs to be a way for general users of the database service to deploy services on it.

We have devised a general infrastructure (named Dynasoar) for the dynamic deployment of Web Services over a grid (or the Internet) [13]. We first describe the generic Dynasoar architecture before showing how it can be used to dynamically deploy services on an AIR, close to the database on which they operate.

Dynasoar provides a generic infrastructure for the dynamic deployment of Web Services in response to the arrival of messages for them to process. This is achieved by dividing the handling of the messages sent to a service between two components – a *Web Service Provider* and a *Host Provider* – with a well-defined interface through which they interact. The *Web Service Provider* receives the incoming SOAP message sent to the Web Service. It arranges for it to be processed by choosing a *Host Provider* and passing to it the SOAP message and any associated QoS information. The *Web Service Provider* holds a copy of the service code (in a “Service Store”) ready for when dynamic service deployment is required. The *Host Provider* controls computational resources (e.g. a cluster or a grid) on which services can be deployed, and messages processed. Therefore, it accepts the SOAP message from the *Web Service Provider* (along with any associated information) and is responsible for processing it and returning a response to the consumer (if one is required). When the message reaches the *Host Provider*, there are two main cases, depending on whether or not the service is already deployed on the node on which the message is to be processed.

If the Web Service is already deployed on one or more of the nodes that it controls, and those deployments can meet any QoS requirements, then the SOAP message is simply passed to one of the deployed services for processing. This case is shown in Figure 2. A request for a Web Service (s2) is sent by the Consumer to the endpoint at the *Web Service Provider* which passes it on to a *Host Provider*. In this case, the *Host Provider* already has the service s2 deployed (on nodes 1 and n in the diagram) and

so, based on current loading information it chooses one (node n) and routes the request to it for processing. A response is then routed back to the consumer. Note that the *Web Service Provider* is not aware of the internal structure of the *Host Provider*, e.g. the nodes on which the service is deployed nor the node to which the message is sent. It simply communicates with the *Host Provider*, which manages its own internal resources.

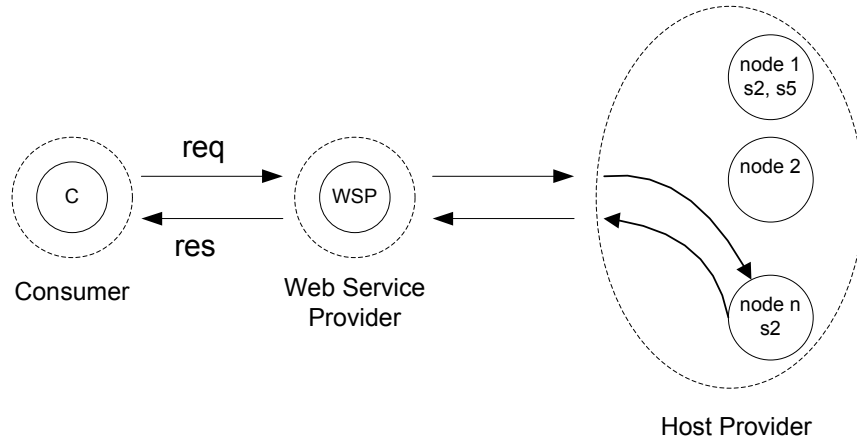


Fig. 2. A request is routed to an existing deployment of the service

The other case is where the Web Service is **not** already deployed. This may be because the *Host Provider* has no deployments of the service, or it may be because existing deployments cannot meet the QoS requirements (the GridSHED project [14],[15] has used mathematical models to generate heuristics that determine when it is desirable to install a service on another node, rather than use an existing deployment). Included in the information that the *Web Service Provider* sends to the *Host Provider* along with the SOAP message is an identifier for the service being called and the endpoint of the Service Store that contains the service code to be deployed. If the *Host Provider* must deploy a service in order to process the SOAP message then it sends a request to the Service Store to retrieve the service code. The *Host Provider* then installs the service on the selected node and routes the message to it. The message is then processed and the result (if any) returned to the consumer. An example of this case is shown in Figure 3. A request for a service (s4) is sent to by the consumer to the endpoint at the *Web Service Provider* which passes it on to a *Host Provider* (step 1 in the Figure). The *Host Provider* does not have the service s4 deployed on any of the nodes it controls and so, based on current loading information it chooses one (node 2), fetches the service code from the *Web Service Provider* and installs the service on that node (step 2). It then routes the request to it for processing (step 3). A response is then routed back to the consumer. Node 2 continues to host service s4, and so it is ready to process any other messages routed to it by the *Host Provider*.

It should be noted that the Consumer need not in any way be aware of the fact that a Web Service is dynamically deployed – its interaction with the service is through the standard mechanism of sending a message to the service endpoint offered by the *Web Service Provider*.

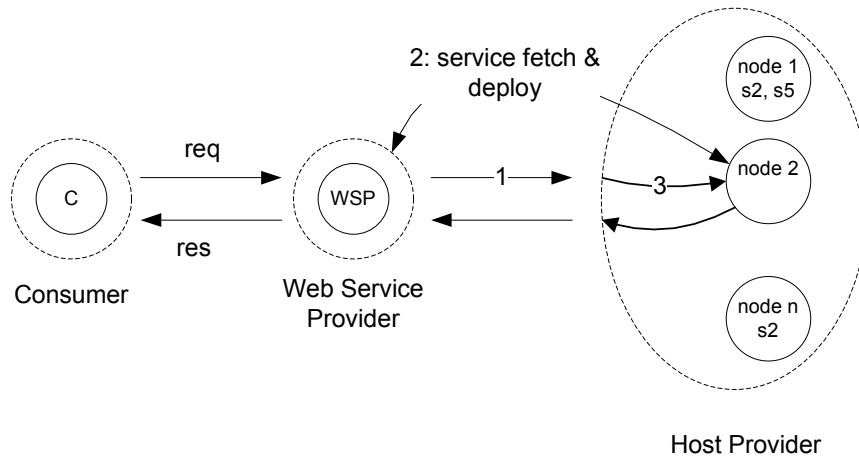


Fig. 3. A service is dynamically deployed to process a request

Once a service is installed on a node it remains in place ready to process future messages until it needs to be reclaimed. This has the potential to generate large efficiency gains when compared to job-based scheduling systems in which each job execution requires its own transport and installation of the code.

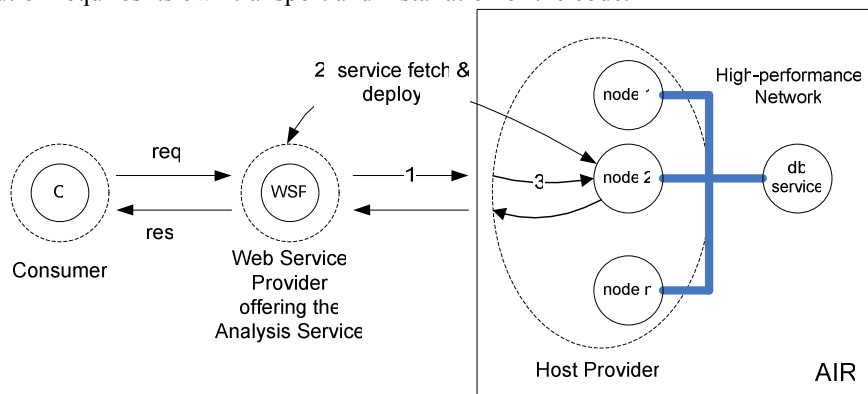


Fig. 4. The Analysis Service is dynamically deployed near the Database Service

The Dynasoar infrastructure can support the dynamic deployment of services on the AIR (Figure 4). The AIR cluster runs the Host Provider component. The provider of the Analysis Service (which may be a lone researcher or a commercial company that specialises in writing analysis services) runs the Web Service Provider component, and makes the service available through this. Either the consumer (through providing the information in the request header) or the Analysis service provider can specify that the service should run on the AIR cluster close to the database service. Therefore when the Web Service Provider receives a message for the Analysis Service, it passes it to the cluster's Host Provider to ensure that it is processed close to the data (Figure 4 step 1). In the Figure, the Host Provider fetches and deploys the

Analysis service from the Web Service Provider (Figure 4 step2) before the request is processed (on node 2).

By utilising the AIR and Dynasoar, services can be dynamically deployed close to the database service so as to avoid moving large amounts of data over long distances. To further reduce data movement, the SkyServer MyDB approach can be used to store results back in the database for further, later analysis and sharing. Therefore, the overall aim is to restrict analysis and movement of the data to within the AIR where possible. Only when absolutely necessary should data be moved outside the AIR. This may occur, for example, if data is to be combined with data from another database. In this case, a method is needed to efficiently query over data held in a set of distributed databases. This is the subject of the next section.

3. Querying Distributed Data

While Section 2 has promoted the localisation of computations on data so as to avoid the cost of moving it, it is actually rare that all the information that could usefully be harnessed by a scientist exists in a single location. Therefore, the ability to combine information from multiple sources is a common requirement in e-science applications. The normal way to combine information in service-based distributed computing is through the use of workflow systems [16]. These allow expert users to choreograph the interactions between a set of services in order to combine their functionality in some way. For example, a workflow used in bioinformatics may take as input data from an experiment, use one service to do some preliminary analysis, compare the results with existing data extracted from a database service, and then use a literature service to find relevant papers that may shed light on the results [17]. Workflows provide a way to capture expert knowledge on how to carry out such a process. Once captured, the analysis of data can be automatically enacted in a performant, consistent and reproducible manner. As a result, we have seen examples where the cost of analysing the data generated by an experiment has been reduced by a factor of ten when compared to manually controlled analysis using web sites and scripts.

However, where it is necessary to access and combine information from more than one database, then the workflow would have to incorporate services to implement standard operations on data such as joins, unions, filters etc. The danger is therefore that the workflow designer must design, implement and deploy services to perform a specialised form of Distributed Query Processing (DQP) on data accesses from a set of specific database services. Further, there is a real risk of poor performance, or failure due to data overload, as workflow enactment engines do not offer the sophisticated data buffering and flow control found in DQP systems.

Consequently, the OGSA-DQP project was created to design and build a distributed query processing system [18] that would: (i) provide a generic way of compiling, optimising and executing distributed queries over a set of OGSA-DAI database systems, and (ii) explore how to exploit the underlying grid infrastructure. It should be stressed that the aim was to complement and enhance, rather than replace, workflow systems; whilst a distributed query can be run in a “stand-alone” manner within an application, it can also be called from within a workflow.

The OGSA-DQP system works as follows. A distributed query processing service is deployed. This has the same interface as the OGSA-DAI database service (e.g. consumers can send queries to it in *perform* documents). However, a client wishing to use the DQP service must also identify the set of OGSA-DAI database systems that it wishes to run queries over. The DQP service then sends *perform* documents to those services to obtain logical and physical metadata. Logical metadata includes information on the tables that these services offer, while the physical metadata includes information that will be of use to a query compiler and optimiser, such as table sizes and the availability of indices.

The consumer can then submit queries to the DQP service which compiles and optimises them. One of the aims of grid infrastructure is to provide a way by which applications can discover and exploit computational resources that are available to them (for example by locating a suitable server on which a job can then be run [19],[20]). Therefore, the DQP service takes this approach and utilises a set of dynamically acquired computing resources for executing the query. The DQP optimiser uses grid information services to determine the computational resources that are available and then works out how best to map the query to them.

The result is a query plan which is distributed over query evaluator services that are deployed on the available hosts. Execution can then take place, with each evaluator executing some portion of the query plan. This may be one or more operators (e.g. join, select) or, in order to exploit parallelism to improve performance, part of an operator (e.g. one element of a parallel hash join). Consider an example query over two distributed databases: the Gene Ontology Database GO, and the genome database GIMS:

```
select p.proteinId, Blast(p.sequence)
from   protein p, proteinTerm t
where  t.termId   = '8372'
and    p.proteinId = t.proteinId
```

This identifies proteins that are similar to human proteins with a GO term of 8372. It includes a call to a *Blast* sequence similarity service in order to return a set of protein ids and their associated similarity scores (OGSA-DQP allows computations encapsulated as Web Services to be incorporated into queries).

Figure 5a shows the query tree after logical and physical optimisation. The optimiser assesses the options for mapping the query plan onto the available nodes (as determined by the grid information service) and chooses a plan that attempts to reduce the response time by parallelising the hash join over a cluster of nodes such that the hash tables can fit entirely into main memory. This results in the execution plan of Figure 5b. The dashed boxes indicate how the plan is partitioned over a set of nodes. The exchange operators [21] manage the transfer of data between the partitions. In Figure 5b, these are shown at the intersection of partitions. In practise, an instance of the exchange operator is planted in each of the two overlapping partitions, one to send tuples and the other to receive. The iterator model [21] is used to control the flow of data around the execution plan (repeated requests for the next result tuple flow down from the top of the execution tree, and in response the tuples are generated and returned back up the tree), but rather than transfer data a tuple at a time between nodes

in response to requests (calls to “next” in the iterator model), data is buffered by the exchange operators, and sent in blocks for efficiency [18].

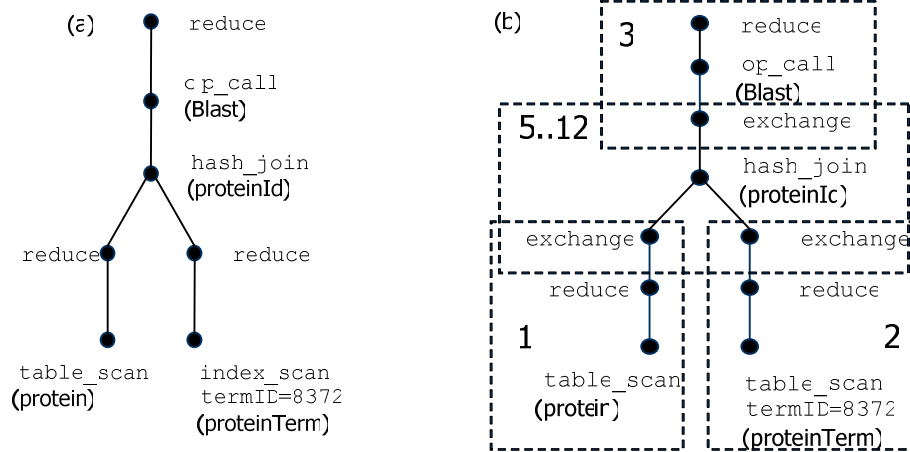


Fig. 5. (a) The Plan after Logical & Physical Optimisation; (b) the Execution Plan

The identifiers of the nodes over which the partition is to be distributed are shown within the partitions in Figure 5b. In this example, the hash join is to be distributed over nodes 5 to 12. The exchange operators on nodes 1 and 2 perform a hash on the join key of each outgoing tuple, and send the tuple to the appropriate node (5-12) where a local join can take place. In this way, parallelism can be utilised to speed-up queries by spreading operators over multiple nodes.

3.1 Exploiting Dynamic Adaptivity

The DQP optimiser makes a decision on how best to distribute the query plan, based on the availability of resources just before the query is executed. However, there are obvious dangers in doing this in a dynamically changing environment. For example, what if during query execution a node becomes unable to deliver the expected performance as it is now also serving another, higher-priority service? What if one or more cheap, high-performance nodes become available just after a long query execution has begun? Or, what if the optimiser's predictions about the required computing resources prove highly inaccurate (for example, because a join produces more rows than expected) and so the consumer will not get the query result within the time they require? Finally, what if a node is withdrawn by its owner from the pool of available resources, or simply fails during query execution? To address such problems and opportunities without killing and re-starting the query, with all the delays and wastage of resources that entails, requires the DQP infrastructure to be adaptive. It must monitor the progress of the execution and the availability of resources, and take dynamic decisions about whether or not to make dynamic changes. Work in the Polar* project has been investigating the opportunities for adaptivity in distributed query execution [22],[23]. It has focussed on two areas that are now discussed: recovering from node failure, and adapting to improve performance.

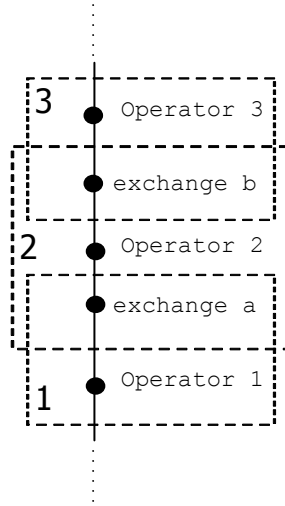


Fig. 6. A Linear Chain of Operators distributed over three nodes

In order to allow a query to recover from the failure of one of the nodes being used by DQP, uniquely numbered checkpoint markers are inserted by the exchange operators into the streams of tuples flowing between nodes. Further, copies of tuples are retained by the exchange operators in a local recovery log after they have been transmitted. The basic scheme will be illustrated by reference to a simple linear chain of operators distributed over three nodes as shown in Figure 6. The *exchange a* operator regularly inserts a checkpoint marker into the stream of tuples that it sends from node 1 to node 2. It also buffers the tuples that it sends to node 2 in a local recovery log. When a tuple arrives at node 2, it is processed by Operator 2 and the result is passed to *exchange b* for transmission to node 3. When a checkpoint marker arrives at *exchange b* on node 3, this indicates that it is safe for all the tuples that preceded the sending of that marker by *exchange a* on node 1 to be deleted from its recovery log. To cause this to happen, *exchange b* on node 3 sends an acknowledgement to *exchange a* on node 1 containing the number of the checkpoint marker.

This ensures that the failure of node 2 cannot cause the failure of the whole query. If node 2 fails, then a new node (2') is acquired and configured to act in its place. The *exchange a* operator on node 1 is informed that from now on it should send tuples to node 2', and requested to retransmit the tuples held in its recovery log to the new node.

The scheme can be used to recover from the failure of any node involved in the query processing. All the exchange operators buffer tuples in a local recovery log as they send them, until they receive an acknowledgement message from the node that is two places downstream in the query execution plan. In fact, by extending the span of the acknowledgements (e.g. retaining copies of tuples until the checkpoint marker has arrived n nodes downstream) an arbitrary number of node failures can be accommodated. The scheme also extends to more complex query graphs with arbitrary fan-in and fan-out by duplicating (fan-out) and aggregating (fan-in) checkpoint markers [22].

Work on DQP adaptivity has also encompassed making dynamic changes to the query plan to improve performance. For example in many DQP plans the work of an operator is shared across a set of nodes in order to exploit parallelism (e.g. the join operator in Figure 5). However, the data may not be evenly spread over the nodes, or some nodes may be more powerful than others. This can result in an imbalance of work that degrades the overall performance. Polar* has investigated the addition of a Monitoring-Assessment-Response framework [23] to detect and attempt to correct such imbalances. When the monitoring component detects an imbalance then the assessment component is called to decide whether a response is appropriate, and if so which of the possible range of responses is best. For example, if one node is underperforming relative to the others it may be better to redistribute work away from it, or, if a suitable new node is available, to configure and then utilise the new node to replace the underperformer.

In summary, the work on DQP has demonstrated its potential for combining information when compared to ad-hoc approaches such as bespoke workflow construction. Having the user express their intent through a declarative query language allows the compiler/optimiser to generate an execution plan that exploits parallelism, dynamic acquisition of grid resources, distribution and flow-control without any user intervention. There is also the added potential for exploiting adaptivity so as to allow dynamic re-configuration to improve performance and automatically recover from failure. The DQP design also shows that grid infrastructures have great potential as the basis for other forms of adaptive computation because they provide information on resource availability and offer ways to dynamically deploy computations on those resources.

4. Conclusions and Future Challenges

The past few years have seen a large investment in building e-science applications. Many of these exploit databases, though many more could but don't. We believe that in order to allow application builders to exploit the potential of databases then two approaches are needed. Firstly, we must advertise the potential benefits of using database technology, as they are often not obvious to those who do not come from a database background (to many scientists, files are the natural way to store data). In the above paper, we have attempted to do this by describing their potential for achieving high-performance through moving computation to data, and for combining distributed information. The success stories of e-science applications that are already successfully utilising databases should also be influential. These come from a wide range of domains including Astronomy [11],[24], Meteorology [25] and Bioinformatics [17].

Secondly, it is important to try to provide "off-the-shelf" building-blocks that simplify the task of integrating databases into e-science applications. OGSA-DAI and OGSA-DQP are two existing examples of this, but in order to identify opportunities for new, useful services that could have generic usage it is necessary to try to identify likely trends in the use of databases in e-science.

In the short term the growth in the use of databases may be driven by the increasing importance of managing and exploiting structured metadata [26],[27]. By provid-

ing semantic information about data, this will in turn increase the potential and importance of distributed query processing for data federation.

In the medium term, the main data-related trend that we expect to see is a huge growth in real-time data being generated by sensors, e.g. for environmental and traffic monitoring. Dealing effectively with this new deluge of information will pose great challenges. One approach would be to simply store the sensor information into a database as it is produced and then allow clients to query it retrospectively. However, this misses the opportunity to act in near real time as new data arrives. For example, data from river water level sensors could, when levels rise sufficiently high, trigger flood plane modelling (possibly on a dynamically acquired server) to predict whether flooding will occur, and if so take necessary action. A key to acting on sensor data is the ability to detect patterns in the low-level events (e.g. a sensor reading) that represent important high-level events on which action can be taken, e.g. “water levels becoming dangerously high”. The results of existing research into continuous queries will clearly have a part to play here, and commercial database companies are making large investments in this area, largely due to the commercial importance of RFID tags [28]. Once a high-level event has been generated, then there is the need to take the appropriate action. This may include associating a workflow with an event (e.g. to run a flood plane model and plot the results on a map before sending an e-mail warning to those likely to be affected), or combining events (e.g. correlating a rise in pollution levels from a set of sensors in one area). Consequently, we expect that there will be an increase in the demand for middleware that can process events ([29],[30]). Section 3 described how the adoption of a common way of providing query access to databases allows the use of generic tools such as OGSA- DQP. In the same way, there would be an advantage in projects agreeing a common way to publish event-driven data. This would encourage the development of generic services to route, filter, log, replay and correlate events.

Over the coming years, we expect databases to become a basic component of all computer systems. Their footprint, in terms of disk, memory and CPU, is now relatively low compared even to a typical, cheap laptop PC. Consequently, there is no reason why their functionality and versatility should not be routinely exploited. This can be seen in moves by many operating systems designers to base their filesystems on a database (e.g. WinFS [31]). Where there exist compute resources that do not currently run a database, then dynamic service deployment systems such as Dynasoar, possibly exploiting Virtual Machine technologies, offer the opportunities to dynamically deploy a database on a node. This “databases everywhere” trend raises questions as to how these databases can be used to good effect. Clearly, there is the potential to utilise them for DQP query processing, but it would also be possible to use them to act as data caches, replicating all or some of the data from remote databases to reduce data transfer costs and query response times.

In conclusion, existing projects are now demonstrating the advantages of utilising databases in e-science projects. The functionality offered by database servers are proving highly useful for applications where sophisticated data querying and integration are required. However, they are also proving to be a good basis for building performant applications through their potential to minimise data movement. Consequently, as experience spreads through the community, we expect the use of databases in e-science to grow. In the short term this may be driven by the increasing impor-

tance of managing and exploiting structured metadata. In the medium term, the growth of event-driven processing for e-science will also increase the demand for database-oriented solutions, though much further work is needed to provide middleware to enable scientists to routinely write applications to extract full value from this dynamic data.

5. Acknowledgements

I would like to thank all of my collaborators at Newcastle and elsewhere for their contributions to the work described in this paper including: the longstanding Polar teams at Newcastle (Jim Smith) and Manchester (Norman Paton, Alvaro Fernandes, Rizos Sakellariou, Nedim Alpdemir, Sandra Sampaio, Tasos Gounaris); the Dynasoar and GridShed project teams (Chris Fowler, Charles Kubicek, John Colquhoun, Arijit Mukherjee, Mark Hewitt, Jennie Palmer, Isi Mitrani, Mike Fisher (BT), Paul McKee (BT)); the OGSA-DAI teams at Newcastle (Arijit Mukherjee), Edinburgh (Malcolm Atkinson), EPCC (Neil Chue Hong, Mario Antonioletti), IBM (Simon Laws, Brian Collins) and Oracle (Dave Pearson); the GAF team (Savas Parastatidis & Jim Webber); ^{my}Grid (Carole Goble and her team) and Rob Smith. The work described in this paper has undertaken in projects funded by the EPSRC, BBSRC, DTI and the UK e-Science core programme.

6. Bibliography

- [1] P. Watson, "Databases and the Grid," in *Grid Computing: Making the Global Infrastructure a Reality*, F. Berman, G. Fox, and A. J. G. Hey, Eds.: Wiley, 2002, pp. 1060.
- [2] I. Foster, C. Kesselman, and S. Tuecke, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations," *International Journal of Supercomputer Applications*, vol. 15, 2001.
- [3] J. Gray, "Distributed Computing Economics," presented at "A Tribute to Roger Needham", Springer Monographs in Computing Science, 2004.
- [4] W3C, "SOAP 1.1." <http://www.w3.org/TR/2000/NOTE-SOAP-20000508/>.
- [5] P. Sandoz, A. Triglia, and S. Pericas-Geertsen, "Fast Infoset," Sun Microsystems, 2004, <http://java.sun.com/developer/technicalArticles/xml/fastinfoset>
- [6] T. M. Kuric, U. V. Catalyurek, C. Chang, and J. H. Saltz, "DataCutter and A Client Interface for the Storage Resource Broker with DataCuttter Services," University of Maryland, CS-TR-4133, 2000 2000.
- [7] OGSA-DAI, "OGSA-DAI." <http://www.ogsadai.org.uk/>.
- [8] V. Welch, I. Foster, C. Kesselman, O. Mulmo, L. Pearlman, S. Tuecke, J. Gawor, S. Meder, and F. Siebenlist, "Security for Grid Services," presented at High-Performance Distributed Computing, Seattle, 2003.
- [9] OASIS, "Web Services Security (WS-Security)." <http://www.oasis-open.org/committees/wss>.

- [10] Global Grid Forum, "GGF, Database Access and Integration Services (DAIS-WG)," 2005.
- [11] M. A. Nieto-Santisteban, A. S. Szalay, A. R. Thakar, W. J. O'Mullane, J. Gray, and J. Annis, "When Database Systems Meet the Grid," Microsoft Research, MSR-TR-2004-81, December 2004.
- [12] P. Watson and P. Lee, "The NU-Grid Persistent Object Computation Server.," presented at 1st European Grid Workshop, Poznan, Poland, 2000.
- [13] P. Watson and C. Fowler, "An Architecture for the Dynamic Deployment of Web Services on a Grid or the Internet," School of Computing Science, University of Newcastle CS-TR-890, February 2005.
- [14] J. Palmer and I. Mitrani, "Optimal Server Allocation in Reconfigurable Clusters with Multiple Job Types," presented at Computational Science and its Applications (ICCSA 2004), Assisi, Italy, 2004.
- [15] C. Kubicek, M. Fisher, P. McKee, and R. Smith, "Dynamic Allocation of Servers to Jobs in a Grid Hosting Environment," *BT Technology Journal*, vol. 22, pp. 251-260, 2004.
- [16] OASIS, "OASIS Web Services Business Process Execution Language." <http://www.oasis-open.org/committees/wsbpel>.
- [17] R. Stevens, R. McEntire, C. A. Goble, M. Greenwood, J. Zhao, A. Wipat, and P. Li, "myGrid and the Drug Discovery Process," *BIOSILICO*, vol. 2, pp. 140-148, 2004.
- [18] M. N. Alpdemir, A. Mukherjee, N. W. Paton, P. Watson, A. A. A. Fernandes, A. Gounaris, and J. Smith, "Service Based Distributed Querying on the Grid," presented at First International Conference on Service Oriented Computing, Trento, Italy, 2003.
- [19] T. Tannenbaum, D. Wright, K. Miller, and M. Livny, "Condor - A Distributed Job Scheduler," in *Beowulf Cluster Computing with Linux*, T. Sterling, Ed.: The MIT Press, 2002.
- [20] I. Foster, C. Kesselman, "Globus: A Metacomputing Infrastructure Toolkit," *International Journal of Supercomputing*, vol. 11, pp. 115-128, 1997.
- [21] G. Graefe, "Encapsulation of Parallelism in the Volcano Query Processing System," presented at SIGMOD Conference, Atlantic City, NJ, USA, 1990.
- [22] J. Smith and P. Watson, "Fault-Tolerance in Distributed Query Processing," School of Computing Science, University of Newcastle CS-TR-893, Feb 2005.
- [23] A. Gounaris, N. W. Paton, A. A. A. Fernandes, and R. Sakellariou, "Self-monitoring Query Execution for Adaptive Query Processing," *Data and Knowledge Engineering*, vol. 51, pp. 325-348, 2004.
- [24] N. A. Walton, A. Lawrence, and T. Linde, "AstroGrid: Initial Deployment of the UK's Virtual Observatory," presented at Astronomical Data Analysis Software and Systems XIII, Strasbourg, 2003.
- [25] B. Plale, J. Alameda, B. Wilhelmson, D. Gannon, S. Hampton, A. Rossi, and K. Droegemeier, "User-oriented Active Management of Scientific Data with myLEAD," *IEEE Internet Computing*, vol. 9, pp. 27-34, 2005.
- [26] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," *Scientific American*, 2001.

- [27] D. D. Roure, N. R. Jennings, and N. R. Shadbolt, "The Semantic Grid: Past, Present and Future," *Proceedings of the IEEE*, vol. 93, pp. 669-681, 2005.
- [28] ORACLE, "RFID and Sensor-Based Services," www.oracle.com/technologies/rfid, 2005.
- [29] G. Fox and S. Pallickara, "The Narada Event Brokering System: Overview and Extensions," presented at International Conference on Parallel and Distributed Processing Techniques and Applications, 2002.
- [30] J. Bacon, J. Bates, R. Hayton, and K. Moody, "Using Events to Build Distributed Applications," presented at 2nd International Workshop on Services in Distributed and Networked Environments, 1995.
- [31] Microsoft, "WinFS," msdn.microsoft.com/data/winfs/, 2005.