Distributed Query Processing on the Grid

Jim Smith[1], Anastasios Gounaris[2], Paul Watson[1], Norman W. Paton[2], Alvaro A.A. Fernandes[2], and Rizos Sakellariou[2]

[1] Department of Computing Science, University of Newcastle upon Tyne, Newcastle, NE1 7RU, UK
{Jim.Smith, Paul.Watson}@newcastle.ac.uk
[2] Department of Computer Science, University of Manchester, Manchester, M13 9PL, UK
{gounaris, norm, alvaro, rizos}@cs.man.ac.uk

Abstract. Distributed query processing (DQP) has been widely used in data intensive applications where data of relevance to users is stored in multiple locations. This paper argues: (i) that DQP can be important in the Grid, as a means of providing high-level, declarative languages for integrating data access and analysis; and (ii) that the Grid provides resource management facilities that are useful to developers of DQP systems. As well as discussing and illustrating how DQP technologies can be deployed within the Grid, the paper describes a prototype implementation of a DQP system running over Globus.

1 Introduction

To date, most work on data storage, access and transfer on the Grid has focused on files. We do not take issue with this – files are clearly central to many applications, and it is reasonable for Grid middleware developers to seek to put in place effective facilities for file management and archiving. However, database management systems provide many facilities that are recognised as being important to Grid environments, both for managing Grid metadata (e.g., [3]) and for supporting the storage and analysis of application data (e.g., [18]).

In any distributed environment there are inevitably multiple related data resources, which, for example, provide complementary or alternative capabilities. Where there is more than one database supported within a distributed environment, it is straightforward to envisage higher-level services that assist users in making use of several databases within a single application. For example, in bioinformatics, it is commonly the case that different kinds of data (e.g., DNA sequence, protein sequence, protein structure, transcriptome) are stored in different, specialist repositories, even though they are often inter-related in analyses.

There are perhaps two principal functionalities associated with distributed database access and use – distributed transaction management and distributed query processing (DQP) [12]. This paper is concerned with DQP on the Grid,

M. Parashar (Ed.): GRID 2002, LNCS 2536, pp. 279–290, 2002.

and both: (i) discusses the role that DQP might play within the Grid; and (ii) describes a prototype infrastructure for supporting distributed query optimisation and evaluation within a Grid setting.

There is no universally accepted classification of DQP systems. However, with a view to categorising previous work, we note that DQP is found in several contexts: in distributed database systems, where an infrastructure supports the deliberate distribution of a database with some measure of central control [13]; in federated database systems, which allow multiple autonomous databases to be integrated for use within an application [11]; and in query-based middlewares, where a query language is used as the programming mechanism for expressing requests over multiple wrapped data sources (e.g., [9]). This paper is most closely related to the third category, in that we consider the user of DQP for integrating various Grid resources, including (but not exclusively) database systems.

Another important class of system in which queries run over data that is distributed over a number of physical resources is parallel databases. In a parallel database, the most common pattern is that data from a centrally controlled database is distributed over the nodes of a parallel machine. Parallel databases are now a mature technology, and experience shows that parallel query processing techniques are able to provide cost-effective scaleability for data-intensive applications (e.g., [15]). This paper, as well as advocating DQP as a data integration mechanism for the Grid, also shows that techniques from parallel database systems can be applied in support of data access and analysis for Grid applications.

The claims of this paper with respect to DQP and the Grid are as follows:

1. In providing integrated access to multiple data resources, DQP in and of itself is an important functionality for data intensive Grid applications.
2. The fact that certain database languages can integrate operation calls with data access and combination operations means that DQP can provide a mechanism for integrating data and computational Grid services.
3. Given (1) and (2), DQP can be seen to provide a generic, declarative, high-level language interface for the Grid.
4. By extending technologies from parallel databases, implicit parallelism can be provided within DQP environments on the Grid.

The paper makes concrete how these claims can be supported in practice by describing a prototype DQP system, Polar*, which runs over the Globus toolkit, and illustrates the prototype using an application from bioinformatics.

The remainder of this paper is structured as follows. Section 2 presents the principal components of a DQP system for the Grid, in particular indicating how this relates to other Grid services. Sections 3 and 4 describe, respectively, how queries are planned and evaluated within the architecture. Section 5 illustrates the proposal using an example involving bioinformatics databases and analysis tools. Finally, Section 6 presents some conclusions and pointers to future work.

2 Architecture

The two key functions of any DQP system are query compilation and query execution. In the Polar* system described in this paper, both these components are based on those designed for the Polar project [16]. Polar is a parallel object database server that runs on a shared-nothing parallel machine. Polar* exploits Polar software components where possible, but as Polar* must provide DQP over data repositories distributed across a Grid, there are a number of key differences between query processing in the two systems. These include:

1. The data dictionary must describe remote data storage and analysis resources available on the Grid – queries act over a diverse collection of application stores and analysis programs.
2. The scheduler must take account of the computational facilities available on the Grid, along with their variable capabilities – queries are evaluated over a diverse collection of computational resources.
3. The data stores and analysis tools over which queries are expressed must be wrapped so that they look consistent to the query evaluator.
4. The query evaluator must use Grid authentication, resource allocation and communication protocols – Polar* runs over Globus, using MPICH-G [5].

A representative query over bioinformatics resources is used as a running example throughout the paper. The query accesses two databases: the Gene Ontology Database *GO* (*www.geneontology.org)* stored in a MySQL (*www.mysql.com*) RDBMS; and *GIMS* [2], a genome database running on a Polar parallel object database server. The query also calls a local installation of the BLAST sequence similarity program (*www.ncbi.nlm.nih.gov/BLAST/*) which, given a protein sequence, returns a set of structs containing protein IDs and similarity scores. The query identifies proteins that are similar to human proteins with the GO term *8372*:

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

In the query, *protein* is a class extent in *GIMS*, while *proteinTerm* is a table in *GO*. Therefore, as illustrated in Figure 1, the infrastructure initiates two subqueries: one on GIMS, and the other on GO. The results of these sub-queries are then joined in a computation running on the Grid. Finally, each protein in the result is used as a parameter to the call to BLAST.

One key opportunity created by the Grid is in the flexibility it offers on resource allocation decisions. In the example in Figure 1, machines need to be found to run both the join operator, and the operation call. If there is a danger that the join will be the bottleneck in the query, then it could be allocated to a system with large amounts of main memory so as to reduce IO costs associated with the management of intermediate results. Further, a parallel algorithm could be used to implement the join, and so a set of machines acquired on the Grid

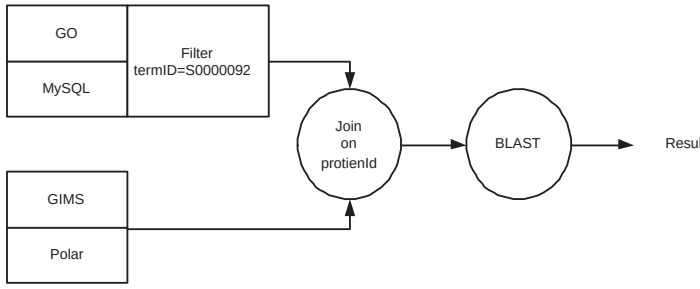

Fig. 1. Evaluating the example query.

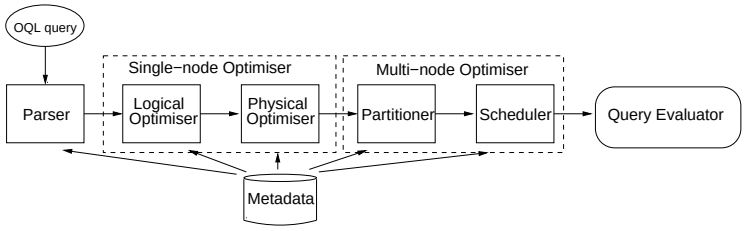

Fig. 2. The components of Polar* query compiler.

could each contribute to its execution. Similarly, the BLAST calls could be speeded-up by allocating a set of machines, each of which can run BLAST on a subset of the proteins.

The information needed to make these resource allocation decisions comes from two sources. Firstly, the query optimiser estimates the cost of executing each part of a query and so identifies performance critical operations. Secondly, the Globus Grid infrastructure provides information on available resources. Once a mapping of operations to resources has been chosen, the single sign-on capabilities of the Grid Security Infrastructure simplify the task of gaining access to those resources.

3 Query Planning

Polar* adopts the model and query language of the ODMG object database standard [1]. As such, all resource wrappers must return data using structures that are consistent with the ODMG model. Queries are written using the ODMG standard query language, OQL.

The main components of the query compiler are shown in Figure 2. The Polar* optimiser has responsibility for generating an efficient execution plan for the declarative OQL query which may access data and operations stored on many nodes. To do this, it follows the two-step optimisation paradigm, which is popular for both parallel and distributed database systems [12]. In the first phase, the single node optimiser produces a query plan as if it was to run on one

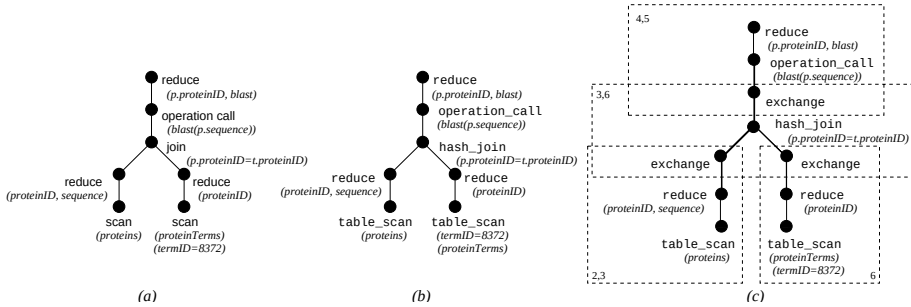

Fig. 3. Example query: (a) single-node logical plan, (b) single-node physical plan (c) multi-node physical plan.

processor. In the second phase, the sequential query plan is divided into several partitions or subplans which are allocated machine resources by the scheduler.

Figure 3(a) depicts a plan for the example query expressed in the logical algebra of Fegaras and Maier [4], which is the basis for query optimisation and evaluation in Polar*. The logical optimiser performs various transformations on the query, such as fusion of multiple selection operations and pushing *project*s (called *reduce* in [4] and in the figures) as close to *scan*s as possible.

The physical optimiser transforms the optimised logical expressions into physical plans by selecting algorithms that implement each of the operations in the logical plan (Figure 3(b)). For example, in the presence of indices, the optimiser prefers *index_scans* to *seq_scans*. Operation calls, like the call to BLAST, are encapsulated by the *operation_call* physical operator. For each OQL query, many physical plans are produced, and the physical optimiser ranks these according to a cost model.

A single-node plan is transformed into a multi-node one by inserting parallelisation operators into the query plan, i.e., Polar* follows the operator model of parallelisation [7]. The *exchange* operator encapsulates flow control, data distribution and inter-process communication. The partitioner firstly identifies whether an operator requires its input data to be partitioned by a specific attribute when executed on multiple processors (for example, so that the potentially matching tuples from the operands of a join can be compared [10]). Secondly, it checks whether data repartitioning is required, i.e., whether data needs to be exchanged among the processors, for example for joining or for submitting to an *operation_call* on a specific machine.

The *exchange*s are placed immediately below the operators that require the data to be repartitioned. For each *exchange* operator, a data distribution policy needs to be defined. Currently, the policies Polar* supports include *round_robin*, *hash_distribution* and *range_partitioning*. A multi-node query plan is shown in Figure 3(c), where the *exchange*s partition the initial plan into many subplans. The physical algebra extended with *exchange* constitutes the parallel algebra used by Polar*.

The final phase of query optimisation is to allocate machine resources to each of the subplans derived from the partitioner, a task carried out by the scheduler in Figure 2 using an algorithm based on that of Rahm and Marek [14]. For running the example query, six machines were available. Three of the machines host databases (numbers *2* and *3* for the GIMS database, and *6* for the GO database). For the *hash_join*, the scheduler tries to ensure that the relation used to construct the hash table can fit into main memory, for example, by allocating more nodes to the join until predicted memory requirements are satisfied. In the example, nodes *3* and *6* are allocated to run the *hash_join*. As some of the data is already on these nodes, this helps to reduce the total network traffic.

The data dictionary records which nodes support BLAST, and thus the scheduler is able to place the *operation_call* for BLAST on suitable nodes (*4* and *5* in Figure 3(c)). The scheduler uses a heuristic that may choose not to use an available evaluator if the reduction in computation time would be less than the increase in the time required to transfer data (e.g., it has decided not to use machine *1* in the example).

4 Query Evaluation

4.1 Evaluating the Parallel Algebra

The Polar* evaluator uses the *iterator* model of Graefe [8], which is widely seen as the model of choice for parallel query processing. In this model, each operator in the physical algebra implements an interface comprising three operations: *open()*, *next()* and *close()*. These operations form the glue between the nodes of a query plan. An individual node calls *open()* on each of its input nodes to prompt them to begin generating their result collections. Successive calls to *next()* retrieve every tuple from that result collection. A special *eof* tuple marks the end of a result collection. After receiving an *eof* from an input, a node calls *close()* on that input to prompt it to shut itself down.

We note that although the term *tuple* is used to describe the result of a call to *next()*, a tuple in this case is not a flat structure, but rather a recursive structure whose attributes can themselves be structured and/or collection valued.

To illustrate the iterator model, Figure 4 sketches an iterator-based implementation of a *hash_join* operator. *open()* retrieves the whole of the left hand operand of the join and builds a hash table, by hashing on the attributes for which equality is tested in the join condition. In *next()*, tuples are received from the right input collection and used to probe the hash table until a match is found and all predicates applying to the join result are satisfied, whereupon the tuple is returned as a result.

The iterator model can support a high degree of parallelism. Sequences of operators can support *pipeline parallelism*, i.e., when two operators in the same query plan are independent, they can execute concurrently. Furthermore, when invocations of an operation on separate tuples in a collection are independent, the operation can be partitioned over multiple machines.

```
class HashJoin: public Operator {
private:
    Tuple *t; HashTable h; Predicate predicate;
    Operator *left; set<Attribute> hash_atts_left;
    Operator *right; set<Attribute> hash_atts_right;
public:
    virtual void open() {
        left->open(); t = left->next();
        while (! t->is_eof()) {
            h.insert(t, hash_atts_left); t = left->next();
        }
        left->close(); right->open(); t = right->next();
    }
    virtual Tuple *next() {
        while (! t->is_eof()) {
            if (h.probe(t, hash_atts_right) && t->satisfies(predicate))
                return t;
            delete t; t = right->next();
        }
        return t;
    }
    virtual void close() {
        right->close(); h.clear();
    }
};
```

Fig. 4. Implementing hash-join as an *iterator*.

Whereas data manipulation operators tend to run in a request-response mode, *exchange* differs in that, once *open()* has been called on it, the producers can run independently of the consumers. Because tuples are complex structures, they are flattened for communication into buffers whose size can be configured. Underlying an instance of *exchange* is a collection of threads managing pools of such buffers so as to constrain flow to a lagging consumer, but to permit flow to a quicker consumer, within the constraints of the buffer pools. This policy is very conveniently implemented in MPI [17] where the tightly defined message completion semantics permit the actual feedback to be hidden within its layer. This use of MPI has enabled the Polar *exchange* to port easily to MPICH-G [5], for use with Globus.

Since MPICH-G is layered above Globus, a parallel query can be run as a parallel MPI program over a collection of wide area distributed machines, oblivious of the difficulties inherent in such meta-computing, which are handled by the underlying Globus services. A parallel program running over such a Grid environment has to find suitable computational resources, achieve concurrent login, transfer of executables and other required files, and startup of processes on the separate resources. In the Polar* prototype, concurrent login to separate accounts is achieved through GSI, and executable staging across wide

```
class PhysicalOperationCall: public PhysicalOperator {
private:
    string signature;                // operation to call
    list<expr*> expression;          // select args from tuple
    list<genform*> *predicate;       // predicate on output tuple
    int key;                         // from cross ref in oplib
    vector<concrete_object*> arg;    // sized appropriately
    class operation_library_stub;    // functions etc in shared library
    operation_library_stub *stub;
public:
    virtual void open() {
        input->open();       // input is pointer to operator
        const d_Operation *oper = search_metadata(signature);
        stub = load_operation_library(oper->operation_library_name());
        key = stub->xref(signature);
    }
    virtual tuple_object *next() {
        while (! t->eof()) {
            tuple_object *t = input->next();
            for (list<expr>::iterator it = expression.begin();
                 it != expression.end(); it++)
                arg[i+1] = t->evaluate(*it);  // leave arg[0] for result
            stub->call_operation(key, arg); t->insert(arg[0]);
            if (t->evaluate(predicate)) return t;
        }
        return t;
    }
    virtual void close() {
        unload_operation_library(stub); input->close();
    }
}
```

Fig. 5. The iterator-based implementation of external operation calls.

area connections through GASS, but all these features are accessed through the MPICH-G interface rather than in a lower level way.

4.2 Accessing Computational and Data Resources during Query Evaluation

From the point of view of the evaluator, both database and analysis operations referred to within a query are external resources, which must be wrapped to enable consistent passing of parameters and returning of results during query evaluation.

To support access to external tools such as BLAST, Polar* implements, in *iterator* style, the *operation_call* operator in Figure 5. While concerns regarding the integrity of the evaluator lead to an isolation of such an operation call within a separate user context in some databases, the view taken so far in Polar* is that

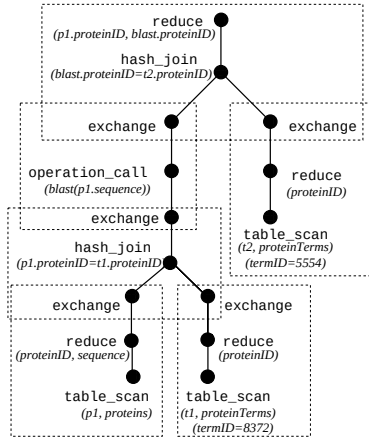

Fig. 6. Query plan for the example in Section 5.

higher performance is achievable when a user can in good faith code an operation to be executed in the server's own context. Thus, such operations are linked into dynamically loadable modules, with stub code generated by the Polar* system to perform the translation of types between tuple format and the format of the programming language used to implement the operation. At runtime, *operation_call* loads the appropriate module in its *open()* call and unloads it in *close()*. Within *next()*, the operator passes attributes from the current tuple to the stub code which assembles required parameters, makes the call and translates the result, which may be collection valued and/or of structured type, back to tuple format before passing it back as the result of the operator.

By making such an application available for use in a Polar* schema, the owner does not provide unrestricted, and thereby unmanageable, access on the internet. By contrast, since Polar* sits above Globus, a resource provider must have granted access rights to a user, in the form of a local login, which they can revoke. Subsequent accesses, while convenient to the user through the single sign-on support of Globus, are authenticated through the GSI.

A specific case that requires access to external operations is the provision of access to external, i.e. non Polar*, repositories. For example, the runtime interface to a repository includes an *external_scan* operator, which exports an *iterator* style interface in common with other operators. However, below the level of *external_scan* the interface to an arbitrary external repository requires special coding. When an external collection is defined in the schema, the system generates the definition of a class, which has the three operations of the *iterator* interface plus other operations to gather statistics and set attributes of the interface such as a query language and the syntax of results generated by the external repository.

The generated class is the template within which a user can implement an *iterator* style interface to the external repository. The mechanisms used are at

the discretion of the user, but the pattern in which the operations of this system-defined class are called is defined by the Polar* system. Fixed attributes, such as the syntax of result tuples returned, are set at the time the completed access class is registered with the Polar* system. Because the access class contains in its instance state a reference to the specification of the wrapped collection in the data dictionary, application data statistics gathered from the particular external store can be written into the data dictionary. The *open()*, *next()* and *close()* operations are simply called by *external_scan* in the usual iterator-based style. However, the results returned by the *next()* operation are translated from the selected syntax into Polar* tuples.

In the running example, the *GO* database is implemented using MySQL, so access to it is through such a system-specified user-defined class of operations. A delegated sub-query such as the access to *proteinTerm* tuples is expressed in SQL, and the results are formatted within the *MySQLAccess* class in Object Interchange Format (OIF)[1]. The *next()* operation of the *external_scan* operator parses each OIF instance to construct tuples which that returned as results to the evaluator.

5 Example from Bioinformatics

The example query introduced in Section 2 is straightforward, but has illustrated how DQP can be used to provide optimisation and evaluation of declarative requests over resources on the Grid. We note that the alternative of writing such a request using lower-level programming models, such as MPICH-G or a COG kit [19] could be quite time consuming. We note also that as the complexity of a request increases, it becomes increasingly difficult for a programmer to make decisions as to the most efficient way to express a request.

For example, a more complex request over the same resources as before could request information about the proteins with the GO term *5554* which are similar to the proteins with the GO term *8372*. In Polar*, such a request can be expressed as follows:

```
select p1.proteinId, p2.proteinId
from   p1 in protein, t1 in proteinTerm,
       p2 in Blast(p1.sequence), t2 in proteinTerm
where  p1.proteinId=t1.proteinId and t1.termID='8372' and
       p2.proteinId=t2.proteinId and t2.termID='5554'
```

This query is compiled into the parallel algebraic expression illustrated in Figure 6. The query plan is decomposed into 6 subplans, each of which is allocated to many nodes, in particular the data and CPU intensive operators like *hash-join* and *operation_call*, for which it is important to exploit parallelism. While retrieving the data from the GO database, the Polar* engine checks whether the data satisfies the condition on the *termID*. In addition, *reduce* operators are inserted before data is communicated over the network. These features are key to reducing the communication cost.

[1] OIF is a standard textual representation for ODMG objects.

6 Conclusions

One of the main hopes for the Grid is that it will encourage the publication of scientific and other data in a more open manner than is currently the case. If this occurs then it is likely that some of the greatest advances will be made by combining data from separate, distributed sources to produce new results. The data that applications wish to combine will have been created by different researchers or organisations that will often have made local, independent decisions about both the best database paradigm and design for their data. The role of DQP in such a setting is to provide high-level, declarative facilities for describing requests over multiple data stores and analysis facilities.

The ease with which DQP allows such requests to be phrased has been illustrated through example queries in Sections 2 and 5. Developing efficient execution plans for such tasks using existing Grid programming environments would take a skilled developer a significant time. We believe that DQP can serve an important role in Grid environments by: (i) increasing the variety of people who can form requests over multiple Grid resources; (ii) reducing development times for certain categories of Grid programming task; and (iii) enabling typical requests to be evaluated efficiently as a result of system-supported query optimisation and support for implicit parallelism.

This paper has described the Polar* prototype DQP system for the Grid. The prototype has been implemented over Globus middleware using MPICH-G, and experiments have been conducted over bioinformatics databases and analysis tools at the authors' geographically remote sites. Future work will: (i) extend the range of physical operators in the algebra; (ii) increase the amount of system information used by the scheduler in query planning; (iii) explore the development of more powerful scheduling algorithms; and (iv) conduct performance evaluations over more and larger databases. We also plan to evolve the Polar* system to be compliant with the emerging Open Grid Services Architecture [6], and to make use of standard service interfaces to databases [20] to reduce the cost of wrapper development.

Acknowledgements

This work was funded by the EPSRC Distributed Information Management Initiative and the UK e-Science Core Programme, whose support we are pleased to acknowledge. We are also grateful for the contribution of Sandra Sampaio, whose work on Polar has been built upon extensively in Polar*.

References

1. R.G.G. Cattell and D.K. Barry. *The Object Database Standard: ODMG 3.0.* Morgan Kaufmann, 2000.
2. M. Cornell, N.W. Paton, S. Wu, C.A. Goble, C.J. Miller, P. Kirby, K. Eilbeck, A. Brass, A. Hayes, and S.G. Oliver. GIMS – A Data Warehouse for Storage and Analysis of Genome Sequence and Functional Data. In *Proc. 2nd IEEE Symposium on Bioinformatics and Bioengineering (BIBE)*, pages 15–22. IEEE Press, 2001.

3. P. Dinda and B. Plale. A unified relational approach to grid information services. Technical Report GWD-GIS-012-1, Global Grid Forum, 2001.
4. L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *ACM Transactions on Database Systems*, 24(4):457–516, December 2000.
5. I Foster and N. T. Karonis. A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems. In *Proc. Supercomputing (SC)*. IEEE Computer Society, 1998. Online at: http://www.supercomp.org/sc98/proceedings/.
6. I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid Services for Distributed System Integration. *IEEE Computer*, 35:37–46, 2002.
7. G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *ACM SIGMOD*, pages 102–111, 1990.
8. G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.
9. L. Haas, D. Kossmann, E.L. Wimmers, and J. Yang. Optimizing Queries Across Diverse Data Sources. In *Proc. VLDB*, pages 276–285. Morgan-Kaufmann, 1997.
10. W. Hasan and R. Motwani. Coloring away communication in parallel query optimization. In *Proceedings of the 21th VLDB Conference*, 1995.
11. D. Hsiao. Tutorial on Federated Databases and Systems. *The VLDB Journal*, 1(1):127–179, 1992.
12. D. Kossmann. The State of the Art in Distributed Query Processing. *ACM Computing Surveys*, 32(4):422–469, 2000.
13. M.T. Ozsu and P. Valduriez, editors. *Principles of Distributed Database Systems (Second Edition)*. Prentice-Hall, 1999.
14. E. Rahm and R. Marek. Dynamic multi-resource load balancing in parallel database systems. In *Proc. 21st VLDB Conf.*, pages 395–406, 1995.
15. S.F.M. Sampaio, J. Smith, N.W. Paton, and P. Watson. An Experimental Performance Evaluation of Join Algorithms for Parallel Object Databases. In R. Sakellariou et al., editors, *Proc. 7th Intl. Euro-Par Conference*, pages 280–291. Springer-Verlag, 2001.
16. J. Smith, S. F. M. Sampaio, P. Watson, and N. W. Paton. Polar: An architecture for a parallel ODMG compliant object database. In *Conference on Information and Knowledge Management (CIKM)*, pages 352–359. ACM press, 2000.
17. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI - The Complete Reference*. The MIT Press, Cambridge, Massachusetts, 1998. ISBN: 0-262-69215-5.
18. A. Szalay, P. Z. Kunszt, A. Thakar, J. Gray, and D. R. Slut. Designing and mining multi-terabyte astronomy archives: The sloan digital sky survey. In *Proc. ACM SIGMOD*, pages 451–462. ACM Press, 2000.
19. G. von Laszewski, I. Foster, J. Gawor, and P. Lane. A Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience*, 13(8-9):643–662, 2001.
20. P. Watson. Databases and the Grid. Technical Report CS-TR-755, University of Newcastle, 2001.