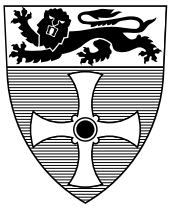
UNIVERSITY OF
NEWCASTLE



University of Newcastle upon Tyne

COMPUTING SCIENCE

Failure Recovery Alternatives In Grid Based Distributed Query Processing: A Case Study

J. Smith and P.Watson.

TECHNICAL REPORT SERIES

No. CS-TR-957 April, 2006

TECHNICAL REPORT SERIES

No. CS-TR-957 April, 2006

Failure Recovery Alternatives In Grid Based Distributed Query Processing: A Case Study

Jim Smith and Paul Watson.

Abstract

Fault-tolerance has long been a feature of database systems, with transactions supporting the structuring of applications so as to ensure continuation of updating applications in spite of machine failures. For read-only queries the perceived wisdom has been that support for fault-tolerance is too expensive to be worthwhile. Distributed query processing (DQP) is coming to be seen as a promising way of implementing applications that combine structured data and analysis operations in dynamic distributed settings such as computational grids. Accordingly, a number of protocols have been described that support tolerance to failure of intermediate machines, so as to permit continuation from surviving intermediate state. However, a distributed query can have a non-trivial mapping onto hardware resources. Because of this it is often possible to choose between a number of possible recovery strategies in the event of a failure. The work described here makes an initial investigation in this area in the context of an example query expressed over distributed resources in a Grid and shows that it can be worthwhile to make this choice between recovery alternatives dynamically, at the point a failure is detected rather than statically beforehand.

Printed and published by the University of Newcastle upon Tyne,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

SMITH, J., WATSON, P..

Failure Recovery Alternatives In Grid Based Distributed Query Processing: A Case Study
[By] J. Smith, P. Watson.

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Science, 2006.

(University of Newcastle upon Tyne, Computing Science, Technical Report Series, No. CS-TR-957)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE
Computing Science. Technical Report Series. CS-TR-957

Abstract

Fault-tolerance has long been a feature of database systems, with transactions supporting the structuring of applications so as to ensure continuation of updating applications in spite of machine failures. For read-only queries the perceived wisdom has been that support for fault-tolerance is too expensive to be worthwhile.
Distributed query processing (DQP) is coming to be seen as a promising way of implementing applications that combine structured data and analysis operations in dynamic distributed settings such as computational grids. Accordingly, a number of protocols have been described that support tolerance to failure of intermediate machines, so as to permit continuation from surviving intermediate state. However, a distributed query can have a non-trivial mapping onto hardware resources. Because of this it is often possible to choose between a number of possible recovery strategies in the event of a failure. The work described here makes an initial investigation in this area in the context of an example query expressed over distributed resources in a Grid and shows that it can be worthwhile to make this choice between recovery alternatives dynamically, at the point a failure is detected rather than statically beforehand.

About the author

Jim Smith worked as a computer programmer for some years with the UK electricity supply industry, then moved to the University of Newcastle upon Tyne where, following studies for MSC and PHD, he is working as an RA on the Polar* project.

Paul Watson is Professor of Computer Science and Director of the North East Regional e-Science Centre. In total, he has over thirty refereed publications, and three patents. Professor Watson is a Chartered Engineer, a Fellow of the British Computer Society, and a member of the UK Computing Research Committee.

Suggested keywords

COMPUTATIONAL GRIDS,
DISTRIBUTED QUERY PROCESSING,
FAULT-TOLERANCE,
PARALLEL QUERY PROCESSING,
ROLLBACK-RECOVERY

Failure Recovery Alternatives In Grid Based Distributed Query Processing: A Case Study

Jim Smith, Paul Watson

5th April 2006

Abstract

Fault-tolerance has long been a feature of database systems, with transactions supporting the structuring of applications so as to ensure continuation of updating applications in spite of machine failures. For read-only queries the perceived wisdom has been that support for fault-tolerance is too expensive to be worthwhile. Distributed query processing (DQP) is coming to be seen as a promising way of implementing applications that combine structured data and analysis operations in dynamic distributed settings such as computational grids. Accordingly, a number of protocols have been described that support tolerance to failure of intermediate machines, so as to permit continuation from surviving intermediate state. However, a distributed query can have a non-trivial mapping onto hardware resources. Because of this it is often possible to choose between a number of possible recovery strategies in the event of a failure. The work described here makes an initial investigation in this area in the context of an example query expressed over distributed resources in a Grid and shows that it can be worthwhile to make this choice between recovery alternatives dynamically, at the point a failure is detected rather than statically beforehand.

keywords computational grids, distributed query processing, fault-tolerance, parallel query processing, rollback-recovery,

1 Introduction

Much work [13] has been done to support access to multiple distributed, autonomous databases, particularly addressing issues relating to heterogeneity, consistency, and availability. However, systems have tended to gather data to a central site for inter-site joins. As described in [19], the emergence of computational grids [5] provides support and motivation for the evolution of the more open query processing espoused in [4] where participants contribute not just data but also function and cycle providers. In such an environment, many widely distributed and autonomous resources may be utilized in the execution of a particular query. Furthermore, it seems likely that the applications will often be demanding, so that resource failures may be not only likely but also costly. It is then better to tolerate the fault rather than throwing away the work done already unless the resources required for completion are not available.

Previous work work [21] describes a basic implementation of support for fault-tolerance in a publicly-available distributed query processing system for the Grid, OGSA-DQP [1]. In that work, the enhanced system is evaluated through measurements of overhead and recovery cost to show that significant gains can be made through recovering and continuing after a failure. However, that earlier work considered only a single recovery scenario, where a failed machine is replaced by an equivalent. In continuation, the work reported here demonstrates for an example

scenario suited to the Grid based nature of the system that there is in general a range of alternative recovery strategies and that it can be desirable to make the choice between these alternatives dynamically on the occurrence of an actual failure.

The rest of this paper is structured as follows. Section 2 discusses related work. Section 3 describes a mapping of an example query onto distributed computational resources and identifies a number of alternative recovery strategies which can be employed following machine failure during query execution. Section 4 reviews the support for fault-tolerance provided in an enhanced version of OGSA-DQP, emphasizing features not described in earlier work. Section 5 presents initial experimental demonstration of the use of alternate recovery strategies in practice. Section 6 concludes.

2 Related Work

Transactions [9] are widely used to structure applications which need to ensure consistent access to persistent data, especially when updates to the data are required. Typically, operations which update persistent state are recorded in a site log so that they can be undone and/redone during recovery from a failure to get back to a consistent state. A commit protocol, typically two phase commit is employed to ensure updates to distributed databases are either all committed or all aborted. Checkpointing database state in such settings reduces the cost of recovery since log entries prior to the checkpoint do not need to be redone. Such recovery techniques aim to ensure the persistent databases can be brought to a consistent state. The application issuing updates can be coded to retry any aborted transaction. Otherwise, or if it's own internal state is lost, the application must restart. This is undesirable if the application is expensive.

Workflows [10] for instance can be structured using internal transactions and maintaining intermediate state in a database to ensure that work already committed need not be redone during recovery. This state can then be replicated to achieve high availability [12]. An individual stateful application which might be called by a workflow can be recovered by logging interactions with the application to support re-creation of the internal application's state after a failure [3].

Like workflow, distributed queries are evaluated through a directed graph structure, but while workflow execution is likely to be event driven, queries typically follow a pipelined data flow pattern. This pipelined nature, and the typically wide area distribution, together with the high level expression of queries, has motivated the exploitation of recovery protocols built into the query algebra rather than at a lower, system, level. Example approaches include: [18, 11] implemented in stream processing [2]; [14] targetted at data warehouse loading; and [21] implemented in the Grid based distributed query processing system OGSA-DQP. While it is important first to implement a protocol that can support some degree of fault-tolerance in such pipelined computations, it is also important to examine the use of that protocol in practice. The contribution of this paper is to consider practical recovery strategies in an example scenario. It transpires that even in this simple case, there are typically multiple possible strategies and that it can be beneficial to choose between the alternatives dynamically at run-time.

Distributed query processing is being increasingly seen as an important tool for expressing complex distributed Grid based computations in a conveniently high level way. For instance, SkyQuery [15] supports DQP over Grid resources with WS being represented as typed used defined functions. GridDB-lite [16] supports access of large scale scientific data from large parallel repositories via SQL queries. In the context of Grid oriented query processing systems, Polar* [19] and OGSA-DQP [1] are distinguished in supporting placement of parts of the query plan on

machines which don't hold data, rather like the compute servers of ObjectGlobe [4], and then using established parallel query processing techniques to seek a benefit through data parallelism. In Figure 1 for instance, a simple query which accesses some expensive operation "F" is evaluated through exploitation of three copies of the WS hosting that operation, in order to reduce the response time. Work in

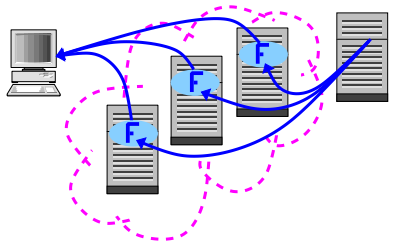


Figure 1: .

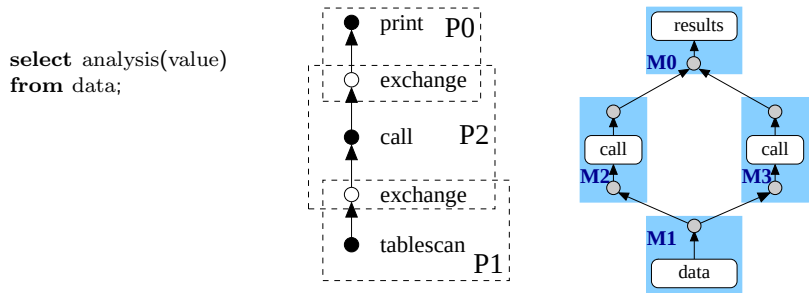
Polar* demonstrated that speedup of an example query in the field of bioinformatics accessing an expensive analysis function could be beneficial even in a heterogeneous environment [20]. The work described here focuses on the requirements for fault-tolerance arising in such query evaluations. Equally however, a query requiring a large join might profit through parallelization over dynamically acquired resources by being able to use a memory based algorithm. Such an approach is demonstrated for instance in [22] in the context of a PC cluster.

3 Recovery Options

Figure 2 shows how an example query might be mapped onto distributed resources by the DQP compiler. The query, shown in Figure 2(a) applies an expensive function call which is hosted by a publicly available WS to data accessed from a remote source. The compiler has generated from the query text a parallel plan shown in Figure 2(b) which implements the query using three partitions, P0, P1, P2. It happens that at the time the query is executed, there are two copies of the WS instantiated on machines which are available to the DQP instance. The compiler has chosen to employ both these instances in its execution plan. Thus, query execution shown in Figure 2(c), is distributed between the user's machine M0, the machine hosting the data source M1, and the two machines M2, M3 hosting the WS which exports the analysis call. During query execution tuples are retrieved from the data source on M1 and divided between M2 and M3. The result tuples on M2,M3 containing the outputs of calls to analysis are forwarded to M0 where the whole result is returned to the user.

The component of the plan allocated to a specific machine is an instance of a partition defined in the parallel plan. The single partition containing the operation call has been replicated on two different machines. In general most partitions in a parallel plan can be replicated in this way; the root partition is an exception. In the following discussion a horizontal slice of a query plan formed by such replication is referred to as a replica set. Every partition of the parallel plan can be represented as a replica set, even if the cardinality of that replica set is restricted to 1. Thus, the example plan has three replica sets, of which two have cardinality 1 and one has cardinality 2.

During the course of the query execution, any of the machines participating could fail. In a failure, a machine might in practice disappear for good as far as the query execution is concerned; i.e. if the query completes before the machine



(a) Query text. (b) Parallel plan. (c) Mapped execution plan.

Figure 2: Mapping an example query.

becomes available. Alternatively, the machine might return to service swiftly, e.g. after a reboot. For the purpose of this work, responses addressing just a single machine failure at a time are addressed. A set of basic operations that can be used to respond to such single machine failures is described below .

restart(query, from) The simplest recovery strategy is to restart the query; this option can clearly be taken in response to failure of any machine. The restart could be from one of various stages. Thus, a second parameter is included to represent (in some way) the choice of where to restart from. For instance, starting from the compiled execution plan, avoids repeating the compilation stage and might be appropriate if for instance a required data source has failed transiently, i.e. has failed but quickly been restored. However, starting from scratch with the original query text offers greatest flexibility and might allow a query to be rerun correctly if a required resource has failed persistently. If the query plan has a point at which intermediate results are fully materialized, it is also possible to restart from that point, thereby saving the cost of repeating all work leading up to that point.

reduce(replica-set) *Reduce* is applied to a replica set to reduce its degree of parallelization by one. Specifically, where the failed machine is one of a set over which a partition of the parallel plan has been parallelized, *reduce* can effect recovery by re-parallelizing the partition over the same set of machines minus the one which had failed. *Reduce* assumes the presence of an underlying recovery protocol offering support for transient state which existed on the failed machine to be restored on the surviving machines after failure. A recovery protocol typically supports this by replicating such transient state in some way, e.g. though checkpointing.

replace(partition) *Replace*, which is applied to a single instance of a partition, is least intrusive to other parts of the query plan. If a single machine fails, the lost partition instance is recreated on a spare machine and the only impact on surviving machines is the need for reconnection of communications with neighbours. Like *reduce*, *replace* inevitably relies on an underlying recovery protocol to provide support for restoration of lost transient state during recovery.

Clearly, there are restrictions on applicability. For instance, only restart from query text is always applicable while *reduce* and *replace* both place constraints

on the machines used in recovery. While constraints of this form appear to be inevitable in such recovery operations, the above selection of recovery operations is not intended to be definitive in this initial work so a formal definition of the relevant constraints is omitted.

4 Implementation

4.1 OGSA-DQP

OGSA-DQP [1] is a publicly available infrastructure which supports user submission of distributed queries over data and analysis resources, the former exposed as GDSs (Grid Data Services) via the OGSA-DAI infrastructure [17] and the latter as WSs (Web Services). The infrastructure implements two Grid Services [6], as follows.

- A GDQS (Grid Distributed Query Service) maintains the metadata catalogue describing the available computational resources and databases. A GDQS accepts user queries expressed in OQL over its global schema. It initiates compilation and optimization of queries to yield execution plans.
- A GQES (Grid Query Evaluation Service) is an evaluation engine that is capable of running a subplan of a distributed query plan generated by a GDQS. An instance of this service is created on each machine the optimizer decides should participate in the distributed query execution. Distributed query execution is therefore performed by a set of GQESs that communicate by exchanging tuples. The use of multiple GQESs allows exploitation of parallelism (e.g. parallelizing joins over a set of GQESs) and also fault-tolerance, as described in this work. The service comprises an execution engine which realizes the physical algebra, in the *iterator* style [8] and includes support for two key operations.
  - *perform* accepts a query subplan, specified as an XML document, and instantiates that plan within the query engine.
  - *putData* accepts a buffer full of tuples from another GQES which are intended for further processing in this GQES. This interface is employed within the exchange operator, after [7], to support the movement of tuples between GQESs.

4.2 Recoverable OGSA-DQP

In order to evaluate support for fault-tolerance in distributed query processing, an enhanced version of OGSA-DQP is being developed. Many details of the enhancement are described in the earlier work [21]. This section gives only a brief summary and highlights differences. Figure 3 illustrates the structure of OGSA-DQP-REC.

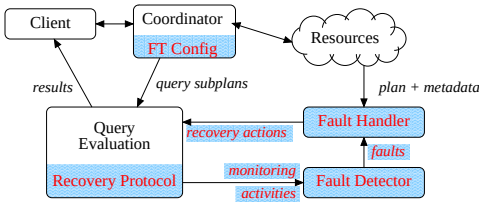


Figure 3: Components of OGSA-DQP-REC.

The client provides the means by which the user can specify a Quality Of Service (QOS) which can be translated into a requirement for fault-tolerance provision. Such a specification might for instance require tolerance to one machine failure during a query execution.

The coordinator takes a user query, generates a plan, and instantiates the required query evaluation environment. Based on the user's specification of the quality of service, the coordinator has to acquire the resources necessary to support the required provision of fault-tolerance. The optimizer has then to take account of the fault-tolerance requirements when generating a query plan, for instance when choosing the number of data source replicas and/or the scheduling of operators.

An enhanced algebra implements an example recovery protocol based on backing up tuples upstream till they are acknowledged as 'used' from downstream.

The Fault Detector (FD) monitors the running system so that it can notify the Fault Handler (FH) of failures. For the purpose of this work a simple FD has been implemented. This comprises a single thread *probe* per machine that regularly calls a null operation exported by the Evaluator Factory on that machine, decrementing a counter with each call. A separate thread *probe-set* increments all the per *probe* counters at a regular interval, in order to diagnose a failure when any one of them reaches some configurable value f. Thus, assuming both *probe* and *probe-set* employ the same interval I, a failure should be detected in approximately $f \times I$ seconds.

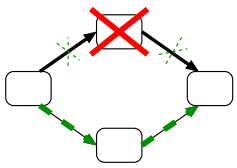
The FH acts upon notifications from the FD, deciding upon and effecting appropriate changes to the running system, for instance substituting a suitable spare machine for one which has failed, or perhaps aborting the query and causing a suitable error indication to be returned to the user if there is no available resource. To perform this task, the FH uses a description of the plan allocated to the evaluators and metadata describing both the fault-tolerance provision and resources which are or may become available in order to support that fault-tolerance provision. The FH is divided into two parts. A Global Fault Handler (GFH) is responsible for deciding on the overall strategy to pursue for a distributed computation in the event of a failure notification, and instructing relevant Local Fault Handlers (LFH). LFHs are responsible for performing reconfiguration operations locally.

In this work, two of the operations, *reduce* and *replace*, described in Section 3 are implemented in FH. The operations are distributed between GFH and LFH; the central GFH allowing coordination of what are inevitably distributed operations. The implementations are illustrated in figure 4. The high level operation to redistribute retrospectively, employed in *reduce*, is responsible not only for reconciling neighbours up and downstream before changing the distribution of tuples specified in the upstream neighbours, but also for ensuring that transient state distributed across the replica set is subsequently correctly distributed across the reduced replica set. A simple way of achieving this goal is to kill the surviving replicas and replay all tuples backed up in the upstream neighbours using the revised distribution policy. An alternative is to disable these surviving replicas while tuples which had been sent to the failed machine are retrieved for replay and then set the surviving replicas to restart, but preserving already accumulated state. While the latter is likely to be cheaper, the former, and simpler, alternative is currently implemented in OGSA-DQP-REC.

5 Experimental Results

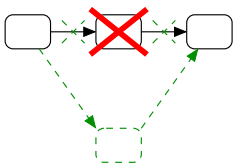
The experiments are performed using OGSA-DQP-REC over a local area network comprising a cluster of 860MHz machines having 512MB main memory each, interconnected via a 100Mbps fast ethernet switch and a separate client 3GHz machine on a different subnet having 1GB main memory. An example dataset contains a

1 disable neighbours
2 redistribute retrospectively
3 reconnect survivors
4 enable neigbours



(a) *reduce*.

1 disable neighbours
2 install neweval
3 disconnect oldeval
4 connect neweval
5 enable neighbours



(b) *replace*.

Figure 4: Implementation of recovery operations.

single table with 10000 tuples, each containing a string attribute which serves as parameter to the analysis call. The latter is implemented in a web service which is hosted on two of the machines in the cluster. In these experiments, queries are submitted to the OGSA-DQP-REC system via a shell script. The OGSA-DQP-REC system compiles and and runs a query using machines in the cluster and writes query results to a file on the client machine. The compiler maps the query onto the data source machine and either one or two of the machines hosting the web service, depending on whether *reduce* or *replace* is being tested. The execution time, measured from submission to completion of a query is saved to a database by the controlling shell script, which also injects faults where required simply by making an *ssh* call to the chosen machine in the cluster and there calling *killall -9 java* which has the effect of aborting the tomcat web server there.

In figure 5, the elapsed time for completion of the example query is shown under the two alternative recovery strategies. In one case, the operation call is

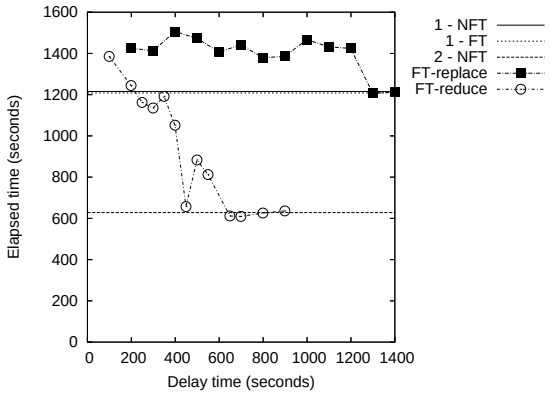


Figure 5: Measured results.

initially scheduled to just one of the machines hosting the web service, thereby

leaving one spare and only the *replace* option is enabled in FH. In the other case, the operation call is initially scheduled to both these machines and just the *reduce* option is enabled in FH. For comparison, the query performance was measured with the fault injection disabled, but using either one or both machines hosting copies of the web service and, in the former case, also with fault-tolerance support (i.e. recovery protocol and failure detector) enabled. The results show that the overhead of the fault-tolerance support is low and that for this expensive function, the benefit obtainable through parallelization is very good. The results also show the usefulness of the *reduce* operation. In this example, if failure occurs very near the start of the query execution, there is little to choose between the two approaches, but when the failure occurs later during query execution, the overall response is improved through both machines having been actively participating in the query execution up to the time of the failure.

The experiments might be taken to suggest the best approach is always to use all available machines and then apply *reduce* to recover from each failure that occurs, up to the point that the last of the replicas has failed and *replace* (using a dynamically acquired machine) or *restart* is enforced. However, it is not always beneficial to use all available machines, so there may be some spare, and in that case *replace* is preferable as it has a lower recovery cost. In general *reduce* may not always be applicable when there is more than one replica, for instance where the reduced replica set has insufficient memory to support a join which was parallelized over the set of machines. In response to failure of a machine participating in a complex parallel plan, it is possible to combine *reduce* of one replica set with *replace* of the original failed machine, for instance if the replica set containing the original failed machine doesn't support any reduction in parallelism. Even though a machine crash is in a sense a straight forward event, distributed queries can map onto distributed machines in complex ways so that responding to a machine crash is likely to entail some measure of choice between alternative options.

6 Conclusions

Distributed query processing is coming to be seen as a way of combining computational and database resources through a high level level expression that is convenient to the user. However, such a trend suggests that while individual queries will become more highly distributed and more demanding, individual machine failures will be more likely. In this setting it becomes preferable to recover at least from an individual failure without having to start the interrupted query from scratch. This initial investigation of an example query suggests that there can be multiple recovery options available to a fault-tolerant DQP. While the set of basic recovery operations identified here is not definitive, it appears unlikely that a single recovery operation would prove universally optimal. Instead it seems that one or more of a generalized set of such operations might be applied dynamically to manipulate a running query plan so as to recover from a particular fault.

acknowledgments

The work reported here has been supported by a grant from the Engineering and Physical Sciences Research Council; number RES/0550/7020 and has profited from discussions with Alvaro A. A. Fernandes, Anastasios Gounaris, Norman W. Paton and Rizos Sakellariou of the Information Management Group at Manchester University who are colleagues in the same project.

References

[1] N. Alpdemir, A. Mukherjee, A. Gounaris, N. W. Paton, P. Watson, and Alvaro A. A. Fernandes. OGSA-DQP: A grid service for distributed querying on the grid. In *EDBT*, pages 858–861, 1979.

[2] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3):109–120, September 2001.

[3] R. S. Barga, D. B. Lomet, S. Paparizos, H. Yu, and S. Chandrasekaran. Persistent applications via automatic recovery. In *IDEAS*, pages 258–267, 2003.

[4] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, S. Pröls, S. Seltzsam, and K. Stocker. ObjectGlobe: Ubiquitous query processing. *The VLDB Journal*, 10(1):48–71, August 2001.

[5] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.

[6] I. T. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. Grid services for distributed system integration. *Computer*, 35(6):37–46, June 2002.

[7] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *SIGMOD*, pages 102–111, Atlantic City, NJ, USA, 1990. ACM Press.

[8] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.

[9] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[10] D. Georgakopoulos M. Hornick and A. Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, April 1995.

[11] J. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. Technical Report CS-04-05, Brown University, May 2004.

[12] M. Kamath, G. Alonso, R. Gunthor, and C. Mohan. Providing high availability in very large workflow management systems. In *EDBT*, pages 427–442, March 1996.

[13] D. Kossman. The state of the art in distributed query processing. *Computing Surveys*, 32(4):422–469, December 2000.

[14] W. Labio, J. Wiener, and H. Garcia-Molina. Efficient resumption of interrupted warehouse loads. In *SIGMOD*, pages 46–57. ACM Press, 2000.

[15] T. Malik, A. Szalay, T. Budavari, and A. Thakar. Skyquery: A web service approach to federate databases. In *CIDR*, 2003.

[16] Sivaramakrishnan Narayanan, Tahsin M. Kurç, Ümit V. Çatalyürek, and Joel H. Saltz. Database support for data-driven scientific applications in the grid. *Parallel Processing Letters*, 13(2):245–271, 2003.

[17] The OGSA-DAI project. http://www.ogsadai.org.uk, 2005.

[18] M. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, pages 25–36. IEEE, 2003.

[19] J. Smith, A. Gounaris, P. Watson, N. W. Paton, A. A.A. Fernandes, and R. Sakalleriou. Distributed query processing on the grid. In *International Workshop on Grid Computing*, pages 279–290, Baltimore, MD, USA, November 2002. Springer.

[20] J. Smith, A. Gounaris, P. Watson, N. W. Paton, A. A.A. Fernandes, and R. Sakalleriou. Distributed query processing on the grid. *International Journal of High Performance Computing Applications*, 17(4), November 2003.

[21] J. Smith and P. Watson. Fault-tolerance in distributed query processing. In *IDEAS*. IEEE Computer Society, 2005.

[22] Xi Zhang, Tahsin M. Kurç, Tony Pan, Ümit V. Çatalyürek, Sivaramakrishnan Narayanan, Pete Wyckoff, and Joel H. Saltz. Strategies for using additional resources in parallel hash-based join algorithms. In *HPDC*, pages 4–13. IEEE Computer Society, 2004.