Adapting to Changing Resource Performance in Grid Query Processing

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

[1] University of Manchester
{gounaris,norm,rizos,alvaro}@cs.man.ac.uk
[2] University of Newcastle upon Tyne
{jim.smith, paul.watson}@ncl.ac.uk

Abstract. The Grid provides facilities that support the coordinated use of diverse resources, and consequently, provides new opportunities for wide-area query processing. However, Grid resources, as well as being heterogeneous, may also exhibit unpredictable, volatile behaviour. Thus, query processing on the Grid needs to be adaptive, in order to cope with evolving resource characteristics, such as machine load. To address this challenge, an architecture is proposed that has been empirically evaluated over a prototype Grid-enabled adaptive query processor instantiating it.

1 Introduction

Grid query processing is particularly relevant where there is a need to integrate information and analysis from different sources for specific periods of time, and to *e-Science* applications, the owners of which, contrary to the typical *e-business* scenario, lack the computational capacity to run some of their tasks and conduct *in-silico* experiments. Especially for the latter case, Grid query processing, like many Grid computations, is likely to place a significant emphasis on high-performance and scalability. Traditionally, query processors often attain scalability and improved performance by relying on the benefits of parallelism. *Pipelined* parallelism has been examined and adopted to different extents in wide-area query processing [15]. Complementarily, query processing can benefit significantly from partitioning the operators within a query execution plan across multiple nodes, a form of parallelism commonly referred to as *intra-operator* or *partitioned* [13], in which all the clones of an operator evaluate a different portion of the same dataset in parallel. GridDB [16] and OGSA-DQP [1] are examples of Grid-enabled database systems that support access to Grid computations and databases, and exploit parallel heterogeneous infrastructures to meet demanding application requirements.

A basic difficulty in efficiently executing a query on the Grid is that the unavailability of accurate statistics at compile time and evolving runtime conditions (such as CPU loads and network bandwidth) may cause load imbalance that detrimentally affects the performance of static techniques for partitioned parallelism. Hence, a challenge for the query processor is to define intra-operator data-partitioning that takes into account these changes. Failing to do so in an efficient way may annul the benefits of parallelism. Just as in homogeneous, controlled environments (e.g., clusters of similar nodes), a slowdown in even a single machine that is not followed by the correct rebalancing, causes the whole system to underperform at the level of the slow machine [2]. To tackle this, the system needs not only to be able to capture these changes as they occur in a

wide-area environment, but also to respond to them in a comprehensive, timely and inexpensive manner by devising and deploying appropriate repartitioning policies.

Adaptive load balancing becomes more complicated if the parallelised operations store intermediate state or have incoming queues, like the *hash join* and *exchange* query operators (we call such operators stateful). Assume, for example, that a query optimizer constructs a plan in which there is a hash join parallelised across multiple sites. A hash function applied to the join attribute defines the site for each tuple. In this case, any data repartitioning of unprocessed tuples needs to be accompanied by repartitioning of the hash tables that had already been created within the hash joins.

This paper presents a comprehensive, effective and efficient solution to the problem above. It dynamically rebalances intra-operator parallelism across Grid nodes for both stateful and stateless operations and, in particular, it makes the following contributions:

- It proposes an architecture for *adaptive query processing* (AQP) that is characterised by the following features: it is non-centralised, it is service-oriented, and its components communicate with each other asynchronously according to the publish/subscribe model. Thus it can be applied to loosely-coupled, autonomous environments such as the Grid.
- It presents an implementation of the architecture through extensions to the OGSA-DQP[3] distributed query processor for the Grid [1], demonstrating the practicality of the approach. The resulting prototype has been empirically evaluated and the results show that it can yield significant performance improvements, in some cases by an order of magnitude, in representative examples. In addition, the overhead remains reasonably low, which is important when adaptivity is not required.

The remainder of the paper is structured as follows. The extensions to the static OGSA-DQP system in order to transform it into an adaptive one are presented in Section 2. Section 3 demonstrates adaptations to workload imbalance. Related work is in Section 4, and Section 5 concludes the paper.

2 Grid Services for Adaptive Query Processing

OGSA-DQP has been implemented over the Globus Toolkit 3 Grid middleware. It provides two types of Grid Services to perform static query processing on the Grid, GDQS (Grid Distributed Query Service) and GQES (Grid Query Evaluation Service). A GDQS contacts resource registries that contain the addresses of the computational and data resources available and updates the metadata catalog of the system. It accepts queries from the users, which are subsequently parsed, optimised, and scheduled employing intra-operator parallelism (e.g., [11]). The query plan consists of a set of subplans that are evaluated by GQESs. A GQES is dynamically created on each machine that has been selected by the GDQS's optimiser to contribute to the execution. GQESs contain the query execution engine, which adopts the *iterator* pipelining model of execution [13]. Data communication is encapsulated within an enhanced *exchange* operator [12], as described later. Inter-service transmission of data blocks is handled by SOAP/HTTP. Remote databases are accessible from the scan operators as GDSs (Grid Data Services) exposed by the generic wrappers developed in

[3] OGSA-DQP is publicly available in open-source form from www.ogsadai.org.uk/dqp.

the OGSA-DAI project (www.ogsadai.org.uk). Also, arbitrary Web Services can play the role of typed foreign functions and be invoked from queries (with the *operation call* operator being responsible for the execution).

Adaptive GQESs (AGQESs) instantiate a novel architecture for AQP that distinguishes between the *monitoring* (i.e. feedback collection), feedback *assessment*, and *response* stages of adaptations. Each AGQES comprises four components (Fig.1): one for implementing the query operators, thus forming the query engine (which is the only component in static GQESs), and three for adaptivity. The extraction of monitoring information is based on self-monitoring operators, as reported in [10]. As such, the query engine is capable of producing raw, low-level monitoring information (such as the number of tuples each operator has produced so far, and the actual time cost of an operator). The *MonitoringEventDetector* component collects such information and acts as a source of notifications on the dynamic behaviour of distributed resources and of query execution. The *Diagnoser* performs the assessment phase, i.e., it establishes whether there is an issue with the current execution (e.g., workload imbalance). The *Responder* decides whether and how to react. Its decisions may affect not only the local query engine, but any query engine participating in the evaluation. The adaptivity components can subscribe to each other and communicate asynchronously via notifications. Note that the above approach implies that the GDQS optimiser need not play any role during adaptations, and the distributed AGQESs encapsulate all the mechanisms required to adjust their execution in a decentralised way.

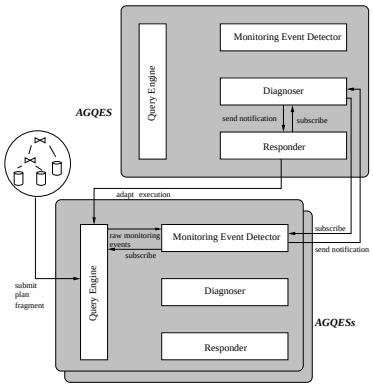


Fig. 1. An adaptive architecture for dynamic workload balancing.

3 Adapting to Workload Imbalance

3.1 Approach

The execution of a plan fragment over a fixed set of resources is considered to be *balanced* when all the participating machines finish at the same (or about the same) time. Workload imbalance may be the result of uneven load distribution in the case of homogeneous machines, but in the case of heterogeneous machines and the Grid, it might be the result of a distribution that is not proportional to the capabilities of the machines employed (both because the machines are different and because their capabilities are subject to dynamic changes). To achieve workload balance during execution we configure the AGQESs in the following way. The *MonitoringEventDetector* is active in each site evaluating a query fragment, and receives raw monitoring events from the local query engine. There also needs to be one activated *Diagnoser* and one *Responder* that subscribe to the *MonitoringEventDetectors* (Fig.1).

Monitoring The query engine generates notifications of the following two types:

- **M1**, which contains information about the processing cost of a tuple. Such notifications are generated by the exchange operators that form the local root of subplans (i.e, exchange producers) and include (i) the cost of processing an incoming tuple in milliseconds; (ii) the average waiting time of the subplan leaf operator for this tuple, which corresponds to the idle time that the relevant thread has spent; and (iii) the current selectivity.
- **M2**, which contains information about the communication cost of an outgoing buffer of tuples. Such notifications are generated by exchanges that form the local root of subplans, and include: (i) the cost of sending a buffer in milliseconds; (ii) the recipient of the buffer; and (iii) the number of tuples that the buffer contains.

These low-level notifications are sent to a *MonitoringEventDetector* component, which:

- groups the notifications of type M1 by the identifier of the operator that generated the notification, and the notifications of the type M2 by the concatenated identifiers of the producer and recipient of the relevant buffer;
- computes the running average of the cost over a window of a certain length, discarding the minimum and maximum values; and
- generates a notification to be sent to subscribed *Diagnoser*, if this average value change by a specified threshold *thresM*.

The default configuration is characterised by the following parameters. The monitoring frequency for the query engine is one notification for each 10 tuples produced (for M1) and one notification for each buffer sent (for M2); the low level notifications from the query engine are sent to the local *MonitoringEventDetector*; the window over which the average is calculated (in the *MonitoringEventDetector*) contains the last 25 events; and the threshold *thresM* to generate notifications for *Diagnosers* is set to 20%. This means that the average processing cost of a tuple needs to change by at least 20%, before the *Diagnoser* is notified. All these values and thresholds are configurable for any component, but determining an optimal setting has left for future work.

Assessment The assessment is carried out by the *Diagnoser*. The *Diagnoser* gathers information produced by *MonitoringEventDetectors* to establish whether there is workload imbalance. Assume that a subplan p is partitioned across n machines, and that $p_i,\ i = 1 \ldots n$, is the subplan fragment sent to the ith AGQES. The *MonitoringEventDetectors* notify the cost per processed tuple $c(p_i)$ for each such subplan, as explained earlier. Also the *Diagnoser* is aware of the current tuple distribution policy, which is represented as a vector $W = (w_1, w_2, \ldots, w_n)$, where w_i represents the proportion of tuples that is sent to p_i. To balance execution, the objective is to allocate a workload $w_i^{'}$ to each AGQES that is inversely proportional to $c(p_i)$. The *Diagnoser* computes the balanced vector $W^{'} = (w_1^{'}, w_2^{'}, \ldots, w_n^{'})$. However, it only notifies the *Responder* with the proposed $W^{'}$ if there exists a pair of w_i and $w_i^{'}$ for which $\frac{|w_i - w_i^{'}|}{w_i}$ exceeds a threshold *thresA*. This is to avoid triggering adaptations with low expected benefit.

The cost per tuple for a subplan $c(p_i)$ can be computed in two ways:

- **A1**, which takes into account only the notifications of type M1 that are produced by the relevant subplan instance; or

– **A2**, which additionally takes into account the notifications of type M2 that are produced by the subplans that deliver data to the relevant subplan instance, and contain the communication costs for this delivery.

The default configuration is characterised by the following parameters. The threshold *thresA* to generate notifications for *Responders* is set to 20%; and the communication cost between subplans in the same machine (i.e., when the exchange producer and consumer reside on the same machine) is considered zero.

Response For operator state management, the system relies on an infrastructure that has been developed mainly to attain fault tolerance. The description of the fault-tolerance features is out of the scope of this paper; details can be found in [18]. Here, we briefly discuss those features that are used for state repartitioning. Exchanges comprise two parts that can run independently: exchange producers and exchange consumers. The producers insert checkpoint tuples into the set of data tuples they send to their consumers. They also keep a copy of the outgoing data in their local recovery log. When the tuples between two checkpoints have finished processing and are not needed any more by the operators higher up in the query plan, the checkpoints are returned in the form of acknowledgment tuples. In practice, the recovery logs contain, at any point, the tuples that have not finished being processed by the evaluators to which they were sent, and thus include all the in-transit tuples, and the tuples that make up operator states. This provides an opportunity to repartition state across consumer nodes by extracting the tuples stored in the recovery logs, and applying the data repartitioning policy to these tuples as well.

The *Responder* receives notifications about imbalance from the *Diagnoser* in the form of proposed enhanced workload distribution vectors $W^{'}$. To decide whether to accept this proposal, it contacts all the evaluators that produce data to estimate the progress of execution in line with [7]. If the execution is not close to completion, it notifies the evaluators that need to change their distribution policy, and the *Diagnosers* that need to update the information about the current tuple distribution (i.e, $W \leftarrow W^{'}$). The data distribution can change in two ways:

– **R1**, where the tuples in the recovery logs (i.e., the tuples already buffered to be sent, and the tuples already sent to their consumers but not processed) are redistributed in accordance with the new data distribution policy. We call this redistribution *retrospective*.
– **R2**, where the buffered tuples and the recovery logs are not affected. We call this redistribution *prospective*.

In the R1 case, operator state is effectively recreated in other machines. This may be useful when adaptations need to take effect as soon as possible, and is imperative for redistributing tuples processed by stateful operators (to ensure result correctness).

3.2 Evaluation

The experiments presented in this section show the benefits of redistributing the tuple workload on the fly to keep the evaluation balanced across evaluators, which results in better performance. The main results can be summarised as follows:

– in the presence of perturbed machines, performance (i.e., response time) improves by several factors and the magnitude of degradation, in some cases by an order of magnitude;

– the overhead remains low and no flooding of messages occurs; and
– the system can adapt efficiently even to very rapid changes.

Two example queries are used:

Q1:
```
select EntropyAnalyser(p.sequence)
from protein_sequences p
```
Q2:
```
select i.ORF2 from protein_sequences p,
protein_interactions i where i.ORF1=p.ORF;
```

The tables *protein_sequences* and *protein_interactions*, along with the *EntropyAnalyser* Web Service operation, are from the OGSA-DQP demo database and they contain data on proteins and results of a bioinformatics experiment, respectively (the *protein_sequences* used in the experiments is slightly modified to make all the tuples the same length to facilitate result analysis). Q1 retrieves and produces 3000 tuples. It is computation-intensive rather than data- or communication-intensive. However, as shown in the experiments, Q1 is chosen in such a way that data communication and retrieval do contribute to the total response time. This contribution is even more significant in Q2, which joins *protein_sequences* with *protein_interactions*, which contains 4700 tuples. So, Q1 and Q2 are complementary to each other: in the former, the most expensive operator is the call to the WS, and in the latter, a traditional operator such as join.

The adaptations described can be applied to an arbitrarily large number of machines. However, as the purpose of the current evaluation is to provide useful insights into the behaviour and effectiveness of the adaptivity policies rather than into how the complete system functions, a carefully controlled experimentation environment is required. Thus two machines are used for the evaluation of *EntropyAnalyser* in Q1, and the join in Q2, unless otherwise stated. The data are retrieved from a third machine. All machines run RedHat Linux 9, are connected by a 100Mbps network, and are autonomously exposed as Grid resources. The third machine retrieves and sends data to the first two as fast as it can. The iterator model is followed, but the incoming queues within exchanges can fit the complete dataset. Due to the pipelined parallelism, the data retrieval is completed independently of the progress of the WS calls and joins. For each result, the query was run three times, and the average is presented here. Finally, we have used two methods to create artificial load leading to machine perturbation: (i) programming a computation to iterate over the same function multiple times, and (ii) inserting *sleep()* calls.

Performance Improvements This set of experiments demonstrates the capability of AGQESs to degrade their performance gracefully when machines experience perturbations. Thus, they exhibit significantly improved performance compared to static GQESs. In the first experiment, we set the cost of the WS call in Q1 in one machine to be exactly 10 times more than in the other, and the responses are prospective (response type R2). The first row of Table 1 shows how the system behaves under different configurations. More specifically, the columns in the table correspond to the following cases:

– *no ad / no imb:* there is no imbalance between the performance of the two services, and adaptivity is not enabled;
– *ad / no imb:* there is no imbalance between the performance of the two services, and adaptivity is enabled;
– *no ad / imb:* one WS call is ten times costlier than the other, thus there is imbalance between the two services, and adaptivity is not enabled; and
– *ad / imb:* there is imbalance, and adaptivity is enabled.

The results are normalised, so that the response time corresponding to *no ad / no imb* is set to 1 unit for each query. The percentage of degradation due to imbalance is given by the difference of the normalised performance from 1. The "unnecessary" adaptivity overhead is the overhead incurred when adaptivity is not needed (i.e., there is no imbalance)[4], which can be computed by the difference between the second and the third columns of Table 1 (1st row). This difference is 5.9%. When one WS is perturbed and there are no adaptivity mechanisms, the response time of the query increases 3.53 times (4th column in Table 1). For this type of query, the cost to evaluate the WS calls is the highest cost. However, it is not dominant, as there is significant I/O and communication costs. Thus, a 10-fold increase in the WS cost results in a 3.53-fold increase in the query response time. The adaptive system manages to drop this increase to 1.45 times, performing significantly better than without adaptivity (45% increase when adaptivity is enabled as opposed to 253% when it is disabled).

Query-Response	**no ad / no imb**	**ad / no imb**	**no ad / imb**	**ad / imb**
Q1 - R2	1	1.059	3.53	1.45
Q1 - R1	1	1.15	3.53	1.57
Q2 - R1	1	1.11	1.71	1.31

Table 1. Performance of queries in normalised units.

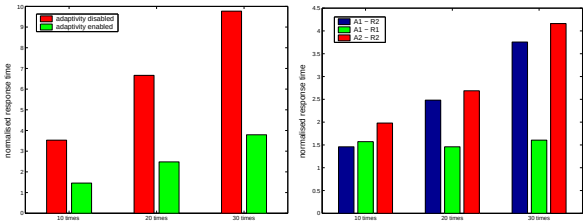
The 2nd row in Table 1 shows the results when the experiment is repeated, and the adaptation is retrospective (type R1 of response). The increase in response time when the adaptivity is not enabled (*no ad / imb*) remains stable as expected (3.53 units). However, the average overhead (*ad / no imb*) is nearly three times more (15.3% of the execution). This is because it is now more costly to perform log management, as the tuples already sent to remote evaluators need to be discarded and redistributed in a tidy manner. Because of the larger overhead, the degradation of the performance in the imbalanced case (*ad / imb*) is larger than for prospective response (1.57 times from 1.45).

The same general pattern is observed for Q2 as well, using the second method to create imbalance artificially. In this case, the perturbation is caused in one machine by the insertion of a *sleep(10msecs)* call before the processing of each tuple by the join. The 3rd row of Table 1 shows the performance when the adaptations are retrospective. The overhead is 11%, and adaptivity, in the case of imbalance, makes the system run 1.31 times slower instead of 1.71.

Varying the Size of Perturbation We reran Q1 for the cases in which the perturbed WS is 10, 20 and 30 times costlier, and adaptations are prospective. Fig. 2(a) shows that the improvements in performance are consistent over a reasonably wide range of perturbations. When the WS cost on one of the machines becomes 10, 20 and 30 times costlier, the response time becomes 3.53, 6.66 and 9.76 times higher, respectively, without dynamic balancing. With dynamic balancing, these drop to 1.45, 2.48 and 3.79 times higher, respectively, i.e., the performance improvement is significant consistently.

Effects of Different Policies Thus far, the assessment has been carried out according to the type A1, in which communication cost is not taken into account.

[4] Without adaptivity, the machines finish at the same time (the difference is in the order of fractions of seconds). This, in general, cannot be attained in a distributed setting. In more realistic scenarios, adaptivity is very rarely "unnecessary", even when distributed services are expected to behave similarly, but these experiments aim to show the actual overhead.



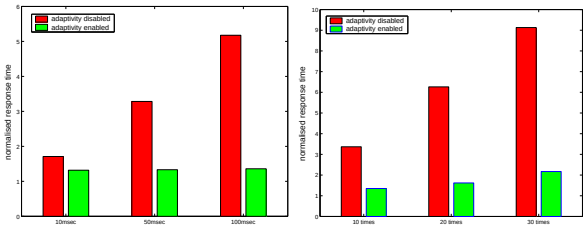
(a) (b)

Fig. 2. (a) Performance of Q1 for prospective adaptations; (b) Performance of Q1 for different adaptivity policies.

The next experiment takes a closer look at the effects of different adaptivity policies. Three cases are examined: (i) when the *Diagnoser* does not take into account the communication cost to send data to the subplan examined for imbalance, and no state is recreated (type A1 of assessment combined with type R2 of response); (ii) when the *Diagnoser* does not take into account the communication cost to send data to the subplan examined for imbalance, and state is recreated (type A1 of assessment combined with type R1 of response); and (iii) when the *Diagnoser* does take into account the communication cost to send data to the subplan examined for imbalance, and no state is recreated (type A2 of assessment combined with type R2 of response). In essence, when the communication cost is not considered (assessment A1), an assumption is made that the cost for sending data overlaps with the cost of processing data due to pipelined parallelism. We believe that such an assumption is valid for this specific experiment, and indeed, this is verified by the experimental results discussed next.

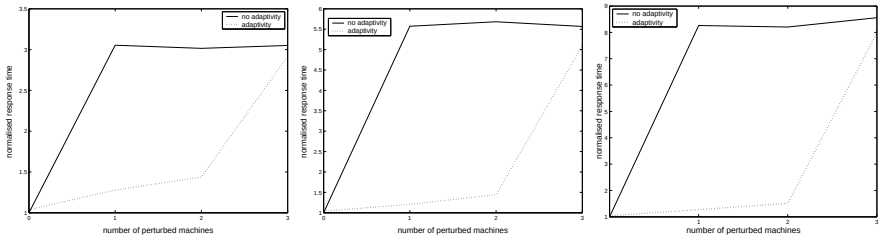
The performance of the three configurations for Q1 is shown in Fig. 2(b). Although all of them result in significant gains compared to the static system, some perform better than others. From this figure we can observe: (i) that taking pipelining into consideration (by performing the assessment of type A1) has an impact on the quality of the decisions and results in better repartitioning (see the difference between the leftmost and the rightmost bar in each group); and (ii) that retrospective adaptations (R1 response) behave better than the prospective ones for bigger perturbations (see the difference between the leftmost and the middle bar in each group). The latter is also expected, as the overhead for recreating state remains stable independently of the size of perturbations, whereas the benefits of removing tuples already sent to the slower consumers, and resending them to the faster ones increases for bigger perturbations. Also, from Fig. 2(b), it can been seen that the bars referring to retrospective adaptations remain similar with different sizes of perturbation, which means that the size of performance improvements increases with the size of perturbations. This happens for two complementary reasons: (i) the higher the perturbation, the more tuples are evaluated by the faster machine, in a way that outweighs the increased overhead for redistributing tuples already sent or buffered to be sent; and (ii) for any of these perturbations, only a very small portion of the tuples is evaluated by the slower machine, which makes the performance of the system less sensitive to the size of perturbation of this machine.

Experiments with Q2 lead to the same conclusions. Fig. 3(a) shows the behaviour of the join query when the *sleep()* process sleeps for 10, 50 and 100 msecs, respectively, and adaptations are of type A1 of assessment and R1 of response.



(a) (b)

Fig. 3. (a) Performance of Q2 for retrospective adaptations; (b) performance of Q1 for prospective adaptations and double data size.



(a) 10 times (b) 20 times (c) 30 times

Fig. 4. Performance of Q1 for retrospective adaptations.

As already identified in Fig. 2(b), retrospective adaptations are characterised by better scalability, and their performance is less dependent on the perturbation.

Varying the dataset size From the figures presented up to this point, retrospective adaptations outperform the prospective ones, but suffer from higher overhead. The reason why prospective adaptations exhibit worse performance is that a significant proportion of the tuples have been distributed before the adaptations can take place. Intuitively, this can be mitigated in larger queries. Indeed, this is verified by increasing the dataset size of Q1 from 3000 tuples to 6000, and making one WS call 10, 20 and 30 times costlier than the other, and the adaptations are prospective. Fig. 3(b) shows the results, which are very close to those when adaptations are retrospective (i.e., Fig. 2(b) for Q1 and Fig. 3(a) for Q2 compared to Fig. 2(a)), and lead to better performance improvements.

Varying the number of perturbed machines Fig. 4 complements the above remarks by showing the performance of Q1 for different numbers of perturbed machines when adaptations are retrospective (three machines have been used for WS evaluation in this experiment). Again, perturbations are inserted by making one WS call 10, 20 and 30 times costlier than the other (Fig. 4(a), (b) and (c), respectively). Due to the dynamic balancing property, the performance degrades very gracefully in the presence of perturbed machines. As explained in detail earlier, the performance when adaptivity is enabled, is very similar for different magnitudes of perturbation, when there is at least one unperturbed machine. Thus the plots corresponding to the case of enabled adaptivity are similar for up to two out of three perturbed machines. Note that the relative degradation (i.e., difference from value 1 in the figures) can be improved by an order of magnitude.

Overheads This set of experiments investigates overheads. We run Q1 when there is no WS perturbation. As shown from Table 1, the overhead of prospective adaptations is 5.9%. This value is the average of two cases. When the adaptivity mechanism is enabled but no actual redistribution takes place, the overhead is 6.2%. However, due to slight fluctuations in performance that are inevitable in a real wide-area environment, if the query is relatively long-running, the system may adapt even though the WSs are the same. For prospective adaptations, a poor initial redistribution may have detrimental effects, since by the time the system realises that there was no need for adaptation, the stored tuples may already have been sent to their destination. Nevertheless, on average, the system behaves reasonably with respect to small changes in performance and incurs a 5.6% overhead. The ratio of the number of tuples sent to the two machines is slightly imbalanced: 1.21. The overhead is slightly smaller than when no actual redistribution occurs as there are benefits from the redistribution.

When the adaptations are retrospective, the overhead is significantly higher, as already discussed. However, the ratio of the tuples is close to the one indicating perfect balance: 1.01. From the above, it can be concluded that retrospective adaptations, even if they are not necessary for ensuring correctness, may be employed when perturbations are large. However, it is felt that the overheads imposed for both types of distribution are reasonable and are worthwhile given the scale of expected gains during perturbations.

We also examined the behaviour of the system for Q1, when the WS cost on one machine is 10 times greater than on the other, and the frequency of generating raw monitoring events from the query engine varies between 0 (i.e., no monitoring to drive adaptivity), and 1 notification per 10, 20 and 30 tuples produced. Both the adaptation quality and the overhead incurred were rather insensitive to these monitoring frequencies (figure omitted due to space limitations). This is because (i) the mechanism to produce low-level monitoring notifications has been shown to have very low overhead [10], and (ii) the adaptivity components filter the notifications effectively. On average, between 100 and 300 notifications are generated from the query engine, but the *MonitoringEventDetector* needs to notify the *Diagnoser* only around 10 times, 1-3 of which lead to actual rebalancing. Thus the system is not flooded by messages, which keeps the overhead low.

Rapid Changes The final set of experiments aims to show the dynamic nature of the system. Thus far, the perturbations have been stable throughout execution. A question arises as to whether the system can exhibit similar performance gains when perturbations vary in magnitude over the lifetime of the run. In these experiments the perturbation varies for each incoming tuple in a normally distributed way, so that the mean value remains stable. Fig. 5 shows the results when the differences in the two WS costs in Q1 vary between 25 and 35 times, between 20 and 40 times, and between 1 and 60 times The leftmost bar in each group in the figure corresponds to a stable cost, which is 30 times higher (e.g.,

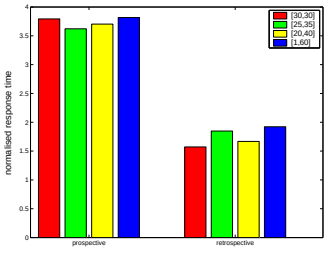


Fig. 5. Performance of Q1 under changing perturbations.

bar A1-R2, 30times in Fig. 2(b) for prospective adaptations), and is presented again for comparison purposes. We can see that the performance with adaptivity is modified only slightly, which enables us to claim that the approach to dynamic balancing proposed in this paper can adapt efficiently to rapid changes of resource performance.

4 Related Work

Query processing on the Grid is a special form of distributed query processing over wide-area autonomous environments. Work in this area has resulted in many interesting proposals such as ObjectGlobe [5], but has largely ignored the issues of intra-query adaptivity. Adaptive query processing is an active research area [4]. However, proposals usually focus on centralised, mostly single-node query processing, and do not yet provide robust mechanisms for responding to changes in the resource performance, which is important especially when an arbitrarily large number of autonomous resources can participate in the query execution, as it is the case in Grid query processing.

As an example that does consider distributed settings, [14] deals with adaptations to changing statistics of data from remote sources, whereas our proposal, complementarily, focuses on changing resource behaviour. Moreover, sources in [14] only provide data, and do not otherwise contribute to the query evaluation, which takes place centrally. Eddies [3] are also used in centralised processing of data streams to adapt to changing data characteristics (e.g., [6]) and operator consumption speeds. When Eddies are distributed, as in [19], such consumption speeds may indicate changing resources. Nevertheless, our approach is more generic as (i) it is not clear how distributed Eddies [19] can extract the statistics they need in a wide-area environment, and how they can keep the messaging overhead low; (ii) Eddies cannot handle all kinds of physical operators (e.g., traditional hash joins); and (iii) redistribution of operator state is not supported. Adapting to changing data properties has also been considered in distributed query processing over streams [8].

In general, workload balancing has been thoroughly examined in parallel databases, but only assuming a context where participating machines either share resources such as disks and memory, or are inter-connected by fast dedicated networks in such a way that data communication is simple and not expensive. As OGSA-DQP is deployed in a different setting, the infrastructure for traditional workload balancing needs to be revisited. For data and state repartitioning, the most relevant work is the Flux operator for continuous queries [17]. The Flux approach has been implemented at the operator level, whereas, our approach is based on loosely coupled components, which can be more easily extended. Rivers [2] follow a simpler approach, and are capable of performing data (but not state) repartitioning. State management has also been considered in [9], but only with a view to allowing more efficient, adaptive tuple rerouting within a single-node query plan.

5 Conclusions

The volatility of the environment in parallel query processing over heterogeneous and autonomous wide-area resources makes it imperative to adapt to changing resource performance, in order not to suffer from serious performance degradation. This paper proposes a solution for dynamic workload balancing through data and operator state repartitioning. This solution is instantiated in the context of a more generic architectural framework implemented through extensions

to the Grid-enabled open-source OGSA-DQP system. The implementation is particularly appealing for environments such as the Grid, as it is based on loosely-coupled components, engineered as Grid Services, which communicate asynchronously and support the publish/subscribe model. The results of the empirical evaluation are promising: performance is significantly improved (by an order of magnitude in some cases), while the overhead remains low enough to allow the benefits of adaptation to outweigh the cost in a wide range of scenarios.
Acknowledgements: This work has been supported by the UK EPSRC grant GR/R51797/01, and by the UK e-Science Programme through the DAIT project.

References

1. N. Alpdemir, A. Mukherjee, A. Gounaris, N. W. Paton, P. Watson, and A. A. A. Fernandes. OGSA-DQP: A grid service for distributed querying on the grid. In *Proc. of 9th EDBT Conference*, pages 858–861, 2004.
2. R. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. Culler, J. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with River: Making the fast case common. In *Proc. of the Sixth IOPADS Workshop*, pages 10–22, 1999.
3. R. Avnur and J. Hellerstein. Eddies: continuously adaptive query processing. In *Proc. of ACM SIGMOD 2000*, pages 261–272, 2000.
4. S. Babu and P. Bizarro. Adaptive query processing in the looking glass. In *CIDR*, pages 238–249, 2005.
5. R. Braumandl, M. Keidl, A. Kemper, K. Kossmann, A. Kreutz, S. Seltzsam, and K. Stocker. ObjectGlobe: Ubiquitous query processing on the Internet. *VLDB Journal*, 10(1):48–71, Aug. 2001.
6. S. Chandrasekaran and M. Franklin. PSoup: a system for streaming queries over streaming data. *VLDB Journal*, 12:140–156, 2003.
7. S. Chaudhuri, V. Narasayya, and R. Ramamurthy. Estimating progress of execution for sql queries. In *Proc. of ACM SIGMOD*, pages 803–814, 2004.
8. M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *CIDR*, 2003.
9. A. Deshpande and J. M. Hellerstein. Lifting the burden of history from adaptive query processing. *Proc. of 30th VLDB Conf.*, pages 948–959, 2004.
10. A. Gounaris, N. W. Paton, A. A. A. Fernandes, and R. Sakellariou. Self monitoring query execution for adaptive query processing. *Data and Knowledge Engineering*, 51(3):325–348, 2004.
11. A. Gounaris, R. Sakellariou, N. W. Paton, and A. A. A.Fernandes. Resource scheduling for parallel query processing on computational grids. In *Proc. of 5th IEEE/ACM GRID Workshop*, pages 396–401, 2004.
12. G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *Proc. of ACM SIGMOD*, pages 102–111, 1990.
13. G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.
14. Z. Ives, A. Halevy, and D. Weld. Adapting to source properties in processing data integration queries. In *Proc. of ACM SIGMOD*, pages 395–406, 2004.
15. D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.
16. D. T. Liu, M. J. Franklin, and D. Parekh. GridDB: a relational interface for the grid. In *Proc. of ACM SIGMOD*, pages 660–660, 2003.
17. M. Shah, J. Hellerstein, S. Chandrasekaran, and M. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proc. of ICDE*, pages 25–36, 2003.
18. J. Smith and P. Watson. Fault-tolerance in distributed query processing. Technical Report CS-TR-893, School of Computing Science, The University of Newcastle upon Tyne, 2004.
19. F. Tian and D. DeWitt. Tuple routing strategies for distributed eddies. In *Proc. of 29th VLDB Conference*, pages 333–344, 2003.