Distrib Parallel Databases (2009) 25: 125–164
DOI 10.1007/s10619-008-7032-5

Adaptive workload allocation in query processing in autonomous heterogeneous environments

Anastasios Gounaris · Jim Smith · Norman W. Paton · Rizos Sakellariou · Alvaro A.A. Fernandes · Paul Watson

Published online: 28 October 2008

Abstract The increasing prevalence of networked storage and computational resources, along with middleware for managing resource access and sharing, raises the prospect that queries can be run over resources obtained on demand, rather than on dedicated infrastructures. However, the movement of query processing into non-dedicated environments means that it is necessary to take account of the partial information and unstable conditions that characterise autonomous, shared, distributed settings. Thus, query processing on grid platforms needs to be adaptive, revising evaluation strategies at query runtime in response to the evolving environment, such as changes to machine load and availability. To address this challenge, adap-

Communicated by Ahmed K. Elmagarmid.

A. Gounaris (✉)
Department of Informatics, Aristotle University of Thessaloniki, Thessaloniki 541 24, Greece
e-mail: gounaria@csd.auth.gr

N.W. Paton · R. Sakellariou · A.A.A. Fernandes
School of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, UK

N.W. Paton
e-mail: norm@cs.man.ac.uk

R. Sakellariou
e-mail: rizos@cs.man.ac.uk

A.A.A. Fernandes
e-mail: alvaro@cs.man.ac.uk

J. Smith · P. Watson
School of Computing Science, University of Newcastle upon Tyne, Newcastle upon Tyne NE1 7RU, UK

J. Smith
e-mail: jim.smith@ncl.ac.uk

P. Watson
e-mail: paul.watson@ncl.ac.uk

tive techniques are described that: (i) balance load across plan partitions supporting intra-operator parallelism; (ii) remove bottlenecks in pipelined plans supporting inter-operator parallelism; and (iii) combine the two aforementioned techniques. The approach has been empirically evaluated in a grid-enabled adaptive query processor.

Keywords Adaptive query processing · Dynamic resource allocation · Load balancing · Distributed query processing · Query optimization · Grid computing

1 Introduction

The increasing prevalence of networked storage and computational resources, along with modern trends for accessing and sharing resources in wide area environments, such as grid platforms [18], cloud computing (e.g., Amazon EC2) and peer-to-peer networks [36], raises the prospect that queries can be run over machines and data stores obtained on demand. However, the movement of query processing into non-dedicated environments means that it is necessary to take account of the partial information and unstable conditions that characterise autonomous, shared, heterogeneous settings. Thus, query processing on such environments needs to be adaptive, revising evaluation strategies at query runtime in response to the evolving environment, such as changes to machine load and availability.

Adaptive query processing (AQP) has attracted significant interest in recent years, and the results yielded thus far can offer speedups of an order of magnitude in many cases [6, 16]. Nevertheless, most of this research has concentrated on the cases in which the computational resources available are pre-determined and their characteristics are stable and known before execution, both when these resources are co-located (e.g. [5, 29]), and when they are geographically dispersed (e.g., [26, 48]). Unfortunately, these assumptions are not valid for query processing in autonomous, heterogeneous environments, where dynamic outsourcing of processing tasks to non-dedicated, heterogeneous computers can take place and the effectiveness of parallelism depends more on the exploitation of the actual machine capabilities and efficient workload distribution, than on the way data is partitioned in the storage software [40]. Consequently, adaptivity in such environments places a greater emphasis on monitoring and learning the behavior of participating machines, the actual communication bandwidth between them, and the impact of this behavior on the progress of query execution on the fly. Moreover, it manifests itself mostly as changes in the way available resources contribute to the query execution rather than as runtime modifications of the query tree shape, and the order in which data or operators are processed. To date, although parallelism is the most commonly adopted approach to speeding up evaluation, adaptive techniques dealing with issues such as which machines should be employed for parallel query execution in heterogeneous settings, and how to distribute workload across heterogeneous machines, are still in their infancy.

1.1 Problem description

This work deals with workload allocation in the context of query processing in autonomous, heterogeneous environments (e.g., computational Grids, PlanetLab [12]).

An amount of workload is allocated to a computational resource, if at least one query plan fragment is executed on that resource, either partially (e.g., as in the case of partitioned parallelism), or completely. Resources are assumed to be autonomous, i.e., owned by third parties, in the sense that their actual capabilities are unknown and can only be inferred by monitoring the progress of the execution of parts of the query plan on them. Moreover, they are non-dedicated, and as such, there is no guarantee that they can execute a query plan fragment exclusively, which may cause fluctuations in their performance at runtime. Heterogeneity denotes the difference in the processing capabilities of the resources available.

A basic difficulty in efficiently executing a query on an autonomous and heterogeneous platform is that the unavailability of accurate statistics at compile time and evolving runtime conditions may lead to sub-optimal execution of a query plan. Statistics may refer to the data being processed (e.g., actual selectivities of predicates) or to the machines available (e.g., average processing time of a foreign function). Typically, the former mostly impact on the construction of the query plan, whereas the latter relate to scheduling and workload allocation decisions.

Common problems in workload allocation in wide-area settings stem from the different characteristics of the machines contributing to the evaluation of a partitioned operator, the loads on machines (which are autonomous and may run many other jobs), the bandwidth of the connections between machines providing raw data, the cost of processing foreign functions, and so on. As so many important factors differ between machines and change with time, a challenge for the query processor is to assign query fragments to resources in a way that takes into account the differences and the runtime changes.

Adaptive workload allocation is particularly relevant to query plan execution where partitioned parallelism is employed, since it is the main approach to attaining balanced execution (i.e., the proportion of workload allocated to a resource is proportional to its, possibly changing, performance). Adaptive load balancing becomes more complicated if the parallelised operations store intermediate state, like the hash join and group-by relational operations; we call such operators stateful. Let us assume, for example, that a query optimizer constructs a plan in which there is a hash join parallelised across multiple sites. The smaller input is used to build the hash table, which is probed by the other input. A hash function applied to the join attribute defines the site for each tuple. In this case, any data repartitioning concerning the tuples not processed yet needs to be accompanied by repartitioning of the state that has already been created within the instances of the hash joins in the form of hash tables.

Dynamic data (and possibly state) repartitioning is just one aspect of the problem, addressing what might be considered as *horizontal* imbalance, in which the completion of an operator within a query plan is possible only on the completion of the slowest of the sibling partitions evaluating the operator. In a pipelined plan, however, there can also be what might be considered as *vertical* imbalance, in which an operator can only process data at the rate it is delivered by its children, and can only deliver data at the rate it can be consumed by its parents (assuming that the size of buffers between operators is small compared to the data set to be processed). In essence, addressing vertical imbalance involves the identification of bottlenecks within pipelines, and their reduction or removal by allocating additional resources to the associated operator(s), thus further increasing the degree of parallelism. Bottlenecks may occur within

a pipeline either as a result of inappropriate scheduling decisions, or because appropriate decisions are overtaken by events in an intrinsically unstable environment.

1.2 Outline of contributions

Adaptive query processing techniques are inadequate for parallel execution over arbitrary heterogeneous networks [40], especially when several adaptivity strategies need to be combined; we defer a detailed description of related work to Sect. 5. This paper presents a comprehensive, effective and efficient solution to the problems of imbalanced partitioned parallelism and existence of bottlenecks mentioned above. The solution includes a technique that dynamically balances intra-operator load across computational nodes both for stateful and stateless operations, and is capable of changing the degree of parallelism and moving load to new resources on-the-fly to overcome bottlenecks. In particular, the paper makes the following key contributions:[1]

- It describes an architecture for *adaptive query processing* (AQP) that (i) covers both data and state repartitioning, and (ii) is capable of allocating new resources dynamically. The architecture allows the development and application of different adaptivity functionalities within the same context, which is a contribution in its own right. Key features of the architecture are that it is non-centralised, and its components communicate with each other asynchronously according to the publish/subscribe model [17]. Thus it can be applied to loosely-coupled, autonomous environments such as the grid and cloud computing services.
- It presents an implementation of the architecture as extensions to the OGSA-DQP[2] service-based distributed query processor for the grid [1], which demonstrates the practicality of the approach.
- It describes adaptive query processing strategies for balancing load and removing bottlenecks within parallel query plans, and demonstrates their effectiveness in decreasing the query response time in practice through a collection of experiments. The experiments fall into three categories: large-scale ones run either (i) in the wild or (ii) on a cluster that aim to show the effectiveness of the approach in practice, and (iii) small scale ones in a controlled environment with few varying parameters that aim to provide insights into the details of the strategies, such as the overheads incurred. The results of the empirical evaluation of the prototype presented show that it can yield significant performance improvements in representative scenarios. In addition, the overhead remains reasonably low, which is a significant property when adaptivity is not required.

The rest of the paper is structured as follows. Section 2 describes an architecture for static query processing that is extended with components that support the implementation of adaptive behaviour. Sections 3 and 4 demonstrate adaptations to maintain load balance and to remove bottlenecks, presenting respectively the techniques and their evaluation. Related work is in Sect. 5, and Sect. 6 concludes the paper.

[1] An early version of parts of this work has appeared in [20].

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

2 Overview of architecture

2.1 The adaptivity framework and its components

This section discusses an architectural framework for AQP that is capable of accommodating various kinds of adaptations (including adaptive data repartitioning and resource allocation). The core idea is to separate the phases of collecting feedback from the execution and environment, analysing this feedback, and responding to changes based on the feedback analysis. Thus, AQP is decomposed into *monitoring*, *assessment* and *response* phases. In contrast, these are inherently present and often conflated in existing AQP systems. We identify three different kinds of adaptivity components, i.e., one component for each phase of the adaptivity cycle. Any AQP technique requires at least one component of each different kind to cover all the adaptivity phases. Consequently, any adaptation is based on collaboration of decoupled entities. A promising way to achieve this in a service-based environment is through message exchange, and to this end, the components support a publish/subscribe interface [17]. The functionality of the framework components is as follows.

Monitoring: A monitoring component acts as a source of notifications on the dynamic behaviour of distributed resources and of query execution. It may perform basic integration and filtering of raw events both to avoid flooding the system with low-level notifications, and to provide support for higher-level notification specification (e.g., by sending a notification only if the load of a machine and the amount of available memory have changed by more than 10%).

Assessment: The role of the assessment component is to establish whether there exist opportunities for improvement of plan performance (or any other QoS criteria), and whether there is a problem with the current execution that needs to be addressed, in order to activate the response mechanisms. It performs its task by correlating and analysing notifications from multiple monitoring components.

Response: The response component is responsible for: (i) identifying valid responses to the issues notified by the assessment component; (ii) evaluating the expected benefit and cost for each valid response; (iii) selecting the most efficient one; and (iv) interacting with the evaluation engine to enforce its decisions.

2.2 Extending exchanges

As indicated in Sect. 1.1, the adaptations described in this paper are applicable to stateful (as well as stateless) operators. This subsection describes how exchange operators handle the issues arising by making use of underlying infrastructure for fault tolerance. The description of the algorithms for fault tolerance is out of the scope of this paper; details can be found in [45]. Here, we provide an overview of the parts that are used for query plan adaptation.

Exchanges [22] comprise two parts that can run independently: exchange producers and exchange consumers (Ex-Prod and Ex-Cons in Fig. 1). 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 their processing and they are not needed any more by

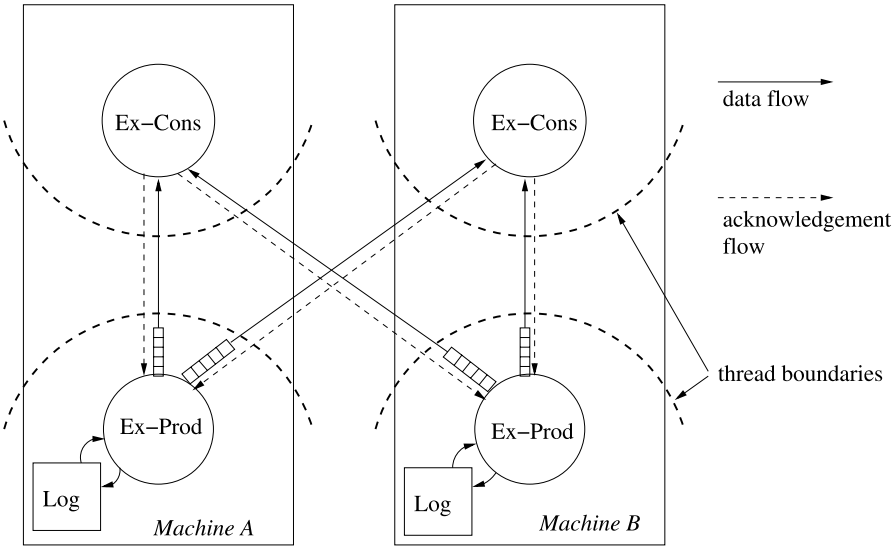

Fig. 1 The enhanced exchanges

the operators higher in the query plan, e.g. through having arrived at the next but one machine downstream, the checkpoints are returned in the form of acknowledgment tuples. Figure 1 shows an example of the data and acknowledgement flows when data is partitioned between two machines (that also hold the data initially). On receipt of the acknowledgement tuples, the recovery logs are pruned accordingly.

In practice, the recovery logs contain, at any point, the tuples that have not finished their processing by the evaluators to which they were sent, and thus include all the in-transit tuples and the tuples that form operator state. 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.

2.3 Extensions to the OGSA-DQP distributed query processing system

This section describes static distributed query processing, as supported by the OGSA-DQP system [1], and the extensions implemented to incorporate the aforementioned framework. As well as supporting the evaluation of queries that access multiple service-wrapped databases (by way of OGSA-DAI [2]) and computational web services, OGSA-DQP has itself been implemented as a collection of interacting web services. The first service type is called the *Grid Distributed Query Service (GDQS)* and encapsulates a query optimizer, which receives user queries in a declarative language, and compiles and optimizes an execution plan for each query. The second type is the *Grid Query Evaluation Service (GQES)*, which resides at each site participating in the evaluation of a distributed query, and encapsulates a query engine that receives and processes fragments of the query plan, as constructed and scheduled by the GDQS. By combining these two types of service, users and developers can integrate data from multiple databases.

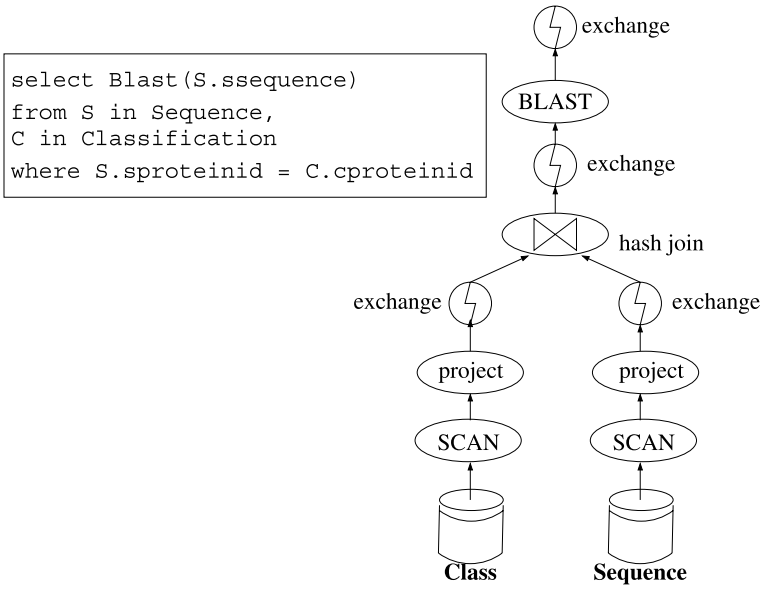

Fig. 2 Example query plan

For example, let us assume the existence of (i) two independent, grid-enabled bioinformatics databases, each containing one table of relevance to the example, namely *Classification* and *Sequence*, and (ii) implementations of the *BLAST* protein sequence similarity function, wrapped as Web Services. A non-trivial task is to integrate the data of the two remote data sources accessible over a grid, and call *BLAST* on the results produced. In OGSA-DQP, this task is declaratively specified by the query in Fig. 2. The figure also shows the query execution plan that performs this task. The nodes of the plan are query operators, in this example, scans, projects, joins, exchanges (for data communication) and a call to a Web Service, the latter being enabled by the *operation call* operator of OGSA-DQP. The operators may be executed on different machines, in parallel. The scan operators rely on OGSA-DAI Data Services; in this paper, we assume that scans are not parallelisable, i.e., data from any single table is accessed in an existing database, and not (for example) striped across multiple machines.

Suppose that the optimizer decides, using the static scheduling algorithm described in [21], that the join of the example query, implemented as a hash join algorithm, should be cloned (to benefit from partitioned parallelism) at both sites holding stored data, which are X and Y, respectively. Suppose further that it decides that the calls to BLAST are to be parallelised across these two sites, X and Y, as well as a third one, Z. The fragments that each site receives are depicted in Fig. 3.

The evaluation of the query plan is achieved through orchestration of multiple GQESs, OGSA-DAI data services and WSs wrapping foreign functions, coordinated by a GDQS. This type of query processing is static because the GQES services are configured for each query at the set-up phase and do not change during evaluation. For computations that are expected to last a relatively short period of time, or where

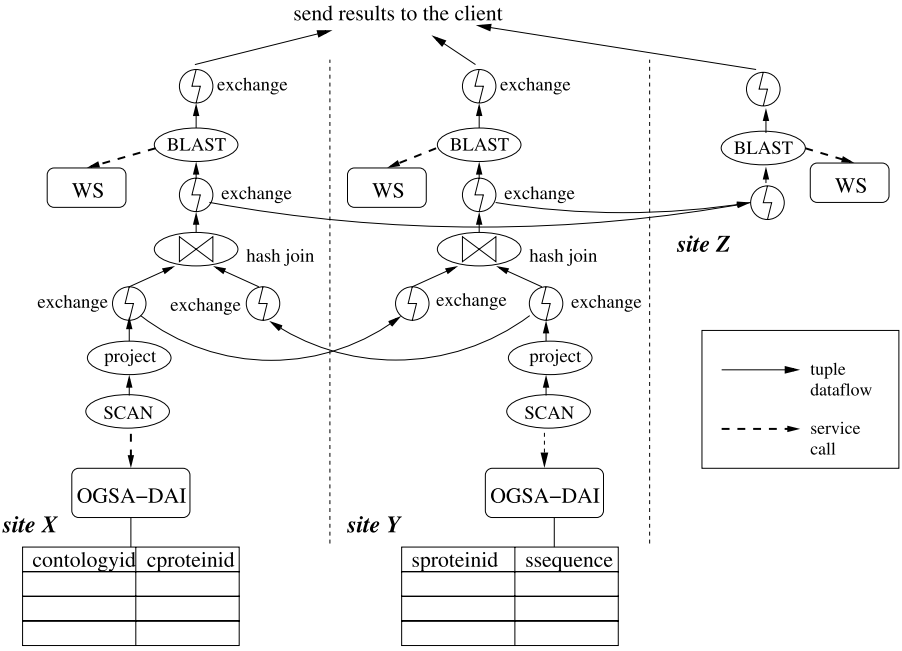

Fig. 3 Query plan distribution

the resources at their disposal are pre-determined and exclusively reserved, statically composing and orchestrating services is sufficient, but this is not always the case for database query processing. To implement the framework of Sect. 2.1, GQES have been transformed into adaptive services.

Adaptive GQESs (AGQESs) instantiate this framework. Each AGQES consists of four components: one query engine for implementing the query operators (the only component in static GQESs), and one for each of *monitoring*, *assessment* and *response*, as illustrated in Fig. 4. Monitoring is based on self-monitoring operators, as reported in [19]. As such, the query engine is capable of monitoring its own behaviour, and of producing low-level monitoring information (such as the number of tuples each operator has produced to this point, and the actual time cost of an operator). The *MonitoringEventDetector* component instantiates the monitoring component of the framework and integrates the events produced by the query engine. The *Diagnoser* performs the assessment phase, i.e., establishes whether there is an issue with the current execution (e.g., load imbalance). The *Responder* is notified of any such issues and chooses how to react. Its decisions may affect not only the local AGQES, but any AGQES participating in the evaluation. The adaptivity notifications and subscription requests are transmitted across AGQESs as XML documents over SOAP/HTTP.

The model above allows arbitrary connections of the adaptivity components. However, in the strategies investigated in this paper, the configuration selected is as follows: there is a separate *MonitoringEventDetector* on each participating site, and a single, globally accessible *Diagnoser* and *Responder* for each query to which *Moni-*

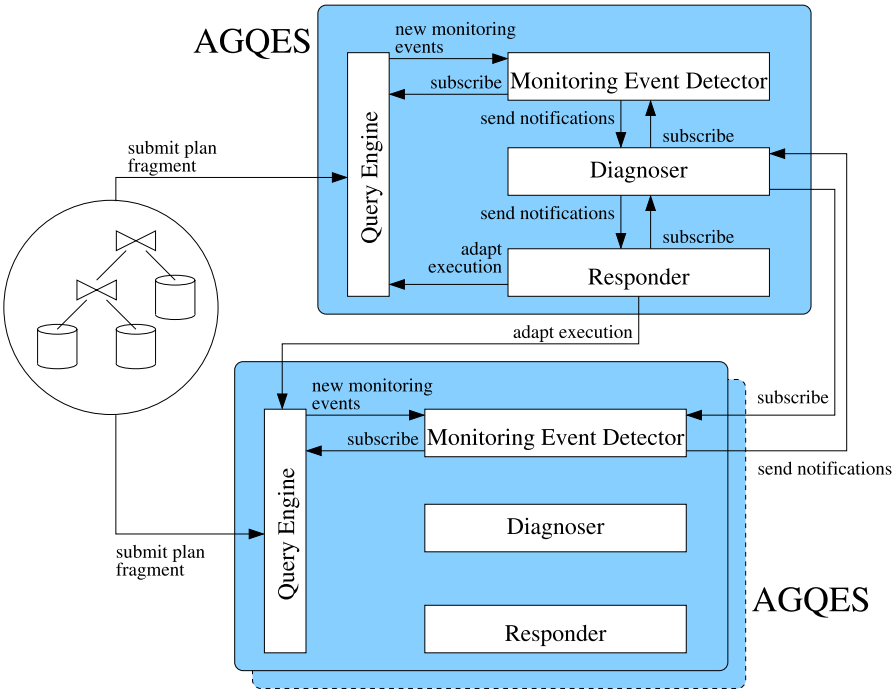

Fig. 4 Instantiating the adaptive architecture for dynamic load balancing

toringEventDetectors are subscribed. Also, since the operators of the query plan and their ordering are not modified during the adaptations examined, there is no need to establish a connection between a *Responder* and the optimiser within the GDQS, which is called to build the initial plan. Nevertheless, these are not limitations of the framework; there is no restriction on the number and the kind of subscription requests an adaptivity component can make. Additionally, in the generic case, a *Responder* can communicate with the compile-time optimiser of the system through the GDQS service interface, for instance to select an alternate realization of a part of the query plan.

3 Adapting to load imbalance and bottlenecks

3.1 Approach to load balancing

Load imbalance with respect to the execution of a plan fragment over a fixed set of resources may be the result of uneven load distribution in the case of homogeneous machines; however, in the case of heterogeneous machines, 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). In other words, load balancing is related to the suitability of

the allocation in terms of the rate with which the nodes can actually process the data supplied.

To achieve load balance during execution we configure the AGQESs in the following way. The *MonitoringEventDetector* is active in each site evaluating a query fragment, receiving raw monitoring events from the local query engine. There also needs to be a single activated *Diagnoser* and *Responder*; the components subscribe to each other, as illustrated in Fig. 4.

3.1.1 Monitoring

The query engine generates notifications of the following two types:

- **M1**, which includes notifications containing 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 includes notifications containing 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 notifications, which contain low-level information on plan progress, are sent to a *MonitoringEventDetector* component that:

- 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 as outliers; and
- generates a notification to be sent to subscribed *Diagnosers*, if these average values 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 the type M1) and one notification for each buffer sent (for the type 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.

3.1.2 Assessment

Assessment is carried out within a *Diagnoser*. A *Diagnoser* gathers information produced by *MonitoringEventDetectors* to establish whether there is load imbalance. Let us assume that a subplan p is partitioned across n machines, and p_i, $i = 1 \ldots n$, is the subplan fragment sent to the ith AGQES. The *MonitoringEventDetectors* notify the cost per tuple $c(p_i)$ for each such subplan, as explained earlier. The *Diagnoser* is also aware of the current tuple distribution policy, which is represented as a vector $W = (w_1, \ldots, w_n)$, where w_i represents the proportion of tuples that is sent to p_i. To balance execution, the objective is to allocate a load w_i' to each AGQES that is inversely proportional to $c(p_i)$. The *Diagnoser* computes the balanced vector $W' = (w_1', \ldots, w_n')$. 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 to be zero.

3.1.3 Response

The *Responder* receives notifications about imbalance from the *Diagnosers* in the form of proposed enhanced workload distribution vectors W'. To decide whether to accept the proposed change to the distribution vectors, it contacts all the evaluators that produce data to estimate the progress of execution in line with [9]. If the progress of execution is below a configurable threshold (which denotes the minimum interval before expected completion at which adaptations are allowed to take place), the *Responder* notifies the evaluators that need to change their distribution policy, and the *Diagnosers* that need to update their distribution vectors.

The data distribution can change in two ways:

- **R1**, in which 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*, as it applies both to new tuples being received for distribution, and also to tuples already forwarded through this redistribution point, as long as the tuples have not been finished with by the operators we are redistributing to; and

- **R2**, in which the buffered tuples and the recovery logs are not affected. We call this redistribution *prospective*, as it applies only from the present point onwards.

In the R1 case, operator state (in the form of buffers of exchange producers, incoming queues of exchange consumers, hash tables of hash-based operators, etc.) is effectively recreated in other machines. This may be useful when adaptations need to take effect as soon as possible, and it is imperative for redistributing tuples processed by stateful operators (to ensure result correctness). In other words, if the plan partition affected by the rebalancing contains operators such as *group by* and operators that build partitioned hash tables, retrospective redistribution is the only valid option; otherwise, it is optional.

3.2 Approach to bottleneck reduction

The previous form of adaptivity dealt with the balanced execution across the n instances of a subplan (or partition) p to which intra-operator parallelism has been applied. In general, a query execution plan *QEP* consists of a set of m such partitions, $P = \{p^1, p^2, \ldots, p^m\}$. If there are no blocking operators, all the partitions are executed concurrently; in an efficient plan, they are expected to take approximately the same length of time to evaluate when fully utilizing local resources. If this is not the case, then one or more of the partitions forms a bottleneck [46], and the execution of the whole plan slows down. This may be for several reasons, such as suboptimal initial resource scheduling decisions. For example, if an expensive operator has been allocated the same resources for the same number of tuples as an inexpensive one, then the throughput of these two operators will vary significantly and the query execution time will largely be determined by the completion time of the costliest operator. Several approaches could be taken to removing such bottlenecks (e.g. changing which analyses are allocated to which machines, changing the order in which the operators are applied with a view to reducing the number of calls to costly operators [24]). The approach examined in this work increases the degree of parallelism of the costliest partition, so that the difference between its predicted time cost and that of the other partitions decreases, if the communication cost is not the dominant cost in the query plan.

As such, the adaptivity policy to tackle bottlenecks focuses on the efficiency of vertical (pipelined) parallelism within the query plan, whereas the policy to tackle imbalanced execution deals with the efficiency of horizontal (partitioned) parallelism. The requirements on the adaptivity components remain the same: each site participating in query execution holds a *MonitoringEventDetector*, and there is a single, globally accessible *Diagnoser* and *Responder*. Additionally, both policies rely on the same monitoring information, which has been described in Sect. 3.1.1, thus providing an example of component reuse across different AQP techniques. Monitoring information of both types *M1* and *M2* are used when adapting to bottlenecks. The descriptions of the distinct assessment and response phases are presented below.

3.2.1 Assessment

The role of assessment in this adaptivity strategy is to identify the costliest parallelisable partition, and to establish whether the response component should be asked to

consider increasing its degree of intra-operator parallelism. To this end, a sequence of steps is followed:

1. The average cost of processing an incoming tuple by the ith instance of the jth partition is denoted c_i^j. This is included in the monitoring information collected and its inverse is equal to the throughput of the partition instance. For each updated value of c_i^j received from a *MonitoringEventDetector*, the estimated response time for the relevant partition instance p_i^j, C_i^j, is computed. To do this, we need the number of tuples that this partition is expected to process, N_{exp}^j, and a workload distribution vector $W^j = \{w_1^j, w_2^j, \ldots, w_n^j\}$, which contains the proportion of the tuples that each partition instance receives. N_{exp}^j depends on the selectivities of its children. The following formula holds: $C_i^j = c_i^j \cdot N_{exp}^j \cdot w_i^j$.
2. The estimated cost C^j for a partition p^j is computed using the formula: $C^j = max(C_i^j), i = 1, \ldots, n$, where n is the number of instances of the partition (i.e., its current degree of intra-operator parallelism).
3. The partitions are sorted by their cost. If the costliest one does not contain an operator that must be run on a specific machine, such as *scan*, it is considered to be parallelisable. Then a heuristic is applied that guarantees that the intra-operator parallelism is increased only if the communication cost does not dominate (if it does, increasing the intra-operator parallelism is not expected to yield any benefits):

$$C_{max}^j > (1 + thresA) \cdot \text{avg}(communication\_cost).$$

The average communication costs can be extracted from monitoring notifications of type M2 that have already been sent to the *Diagnoser*. *thresA* is set to *0.05* in the experiments.

A further control heuristic is used in some of the experiments. In a pipeline that contains multiple parallelisable partitions, assessment may be conducted either as soon as a bottleneck is detected involving *any* of the pipelined partitions, or it can be *deferred* until throughput information is available for *all* partitions in the pipeline. This heuristic is intended to avoid premature commitment of additional resources to resolve a bottleneck that may not be the bottleneck for the whole query.

Given that the cost estimates rely on the load distribution vector, as shown in the first step, there is a danger, if the vector is not balanced in the way discussed in the previous section, that a specific partition may erroneously seem to be a bottleneck point. To avoid that, it is preferable to combine the adaptivity policies and proceed to bottleneck removal only when load balancing has been achieved.

3.2.2 Response

The set of actions that the *Responder* takes to respond to bottleneck diagnosis can be deemed to be a superset of the actions required in the case of load imbalance. During its creation by the GDQS coordinator service, the *Responder* is also notified of all the machines available. The *Responder* reacts to bottlenecks only if the execution is not very close to completion (this is a tunable parameter set to 95% in the experiments),

as in the load balancing strategy. The actual response is activated only if there are machines available, and consists of the following three steps:

1. An AGQES is created remotely by the *Responder*. Subsequently, the partitioned subplan is sent to the new AGQES. However, this remains temporarily idle because the other AGQESs have not yet been notified of its existence.
2. A notification is sent to the AGQESs that consume data from the new evaluator, for them to update their catalogs and wait for data.
3. A notification is sent to the AGQESs that send data to the new evaluator (i) to inform their relevant exchange producers that data can be sent to the new AGQES; and (ii) to modify the data partitioning policy of the AGQESs that send data to take into account the new consumer. This is similar to the response form for imbalanced execution. The new consumer is initially assigned a proportion of the complete workload which is equal to the average proportion of the pre-existing AGQESs. The dynamic balancing mechanism presented in Sect. 3.1 can correct bad initial decisions. All responses carried out in the experiments are prospective.

Essentially, in this approach, bottleneck is implicitly defined as the costliest plan partition, which is actually the case for parallel (pipelined) execution in heterogeneous settings [46]; in such settings, the costliest partition defines the query response time. The approach presented increases the partitioned parallelism until the communication cost, or the cost to retrieve data from non-parallel physical storage infrastructure, starts to dominate.

4 Evaluation

This section presents experimental results of the prototype developed. Using a real prototype, rather than simulating the behavior of the system and the environment, is, in our view, a strong advantage of this work, since end users are mostly interested in the actual impact of the adaptivity techniques on the system performance. Several experimental settings have been chosen that complement each other and examine the system from different perspectives. In experiments "in the wild" (i.e., in real, wide area environments), the system is deemed as a black box. Such experiments can reveal its actual high level behaviour, but cannot be easily repeated and interpreted in detail, due to the large number of varying factors. On the other hand, small scale experiments provide useful insights into the low-level details of the adaptivity techniques. A middle solution is to run experiments in a cluster setting. In this section, all three approaches have been followed. The configurable parameters are those mentioned in Sect. 3, unless explicitly redefined.

As discussed previously, the responses of both adaptations manifest themselves at the operator level; however, although in the case of adaptations for load balancing, the effectiveness of the adaptations can be evaluated in experiments that involve a single (parallelised) operator (e.g., as in [42]), in the case of bottlenecks, the effectiveness of the adaptations can be measured in experiments that involve pairs of (parallelised) operators. As such, in terms of query plans, the experiments have deliberately been designed to enable fine-grained analyses of the adaptations by using simple queries.

4.1 Evaluation in the wild

This section describes some performance results obtained for adaptive load balancing implemented using the infrastructure described in Sect. 3.1. The results have been obtained using PlanetLab [12], a resource currently comprising hundreds of machines world-wide that serves as a shared evaluation environment for wide area distributed experiments. As PlanetLab resources are used concurrently by multiple users, it provides an excellent way to test adaptivity techniques "in the wild", though the corollary is that it does not provide a controlled setting for repeatable experiments.

A user at a participating institution who wishes to perform an experiment in the PlanetLab environment is allocated a *slice*. Such a slice gives the user an account on each machine in a subset of the machines in PlanetLab. Upon logging-in to such an account, a user's processes are isolated from those of other users in a separate virtual environment, or *sliver*. While monitoring utilities, such as *ps* and *top* do not show individual processes belonging to other users, the overall load on the machine is visible, and is typically both significant and variable. Broadly speaking, multiple concurrent users of a single machine each have the impression of exclusive access to a machine of varying performance, which is somewhat less powerful than the actual machine. Thus, PlanetLab provides the right kind of unpredictability to experimentally evaluate the adaptivity measures described in this paper.

The experiments used a slice containing 55 machines: 21 machines in the UK, 18 machines elsewhere in Europe, 8 machines in Asia, and 8 machines in the US. However, for the duration of these particular experiments the number of these machines which were accessible varied between about 20 and 30. The script driving the queries selects for each run those machines that are running and have been most responsive to an hourly probe. The result is that different subsets of the machines are used for different runs. The specifications of machines in the slice vary, for instance: in cpu speed between 1 and 3 GHz; in memory between 512 MB and 2 GB; and in cache size between 256 and 2048 KB. However, apart from the fact that many of the machines have some trade off in their parameters, e.g., lower speed but more memory, the overall load appears to be both high and variable. This makes it generally difficult to select a favoured set of machines purely from their static specification.

4.1.1 Experiment 1: Runs on the PlanetLab

The experiments explore load balancing for a query that invokes external operations using data from a table in which one attribute, *sequence*, represents a gene sequence; in the experiments, the table contains 100,000 tuples. In addition, operation calls *analysis1* and *analysis2*, which have identical cost and are stateless, are implemented as web services, which are available at multiple sites, thereby supporting partitioned parallelism. Each operation performs a complex analysis as might occur in a practical bioinformatics scenario.

In this environment, the performance of the following pipeline example query is measured:

Q1:
```
select analysis1(s.ssequence),
       analysis2(s.ssequence)
from Sequence s;
```

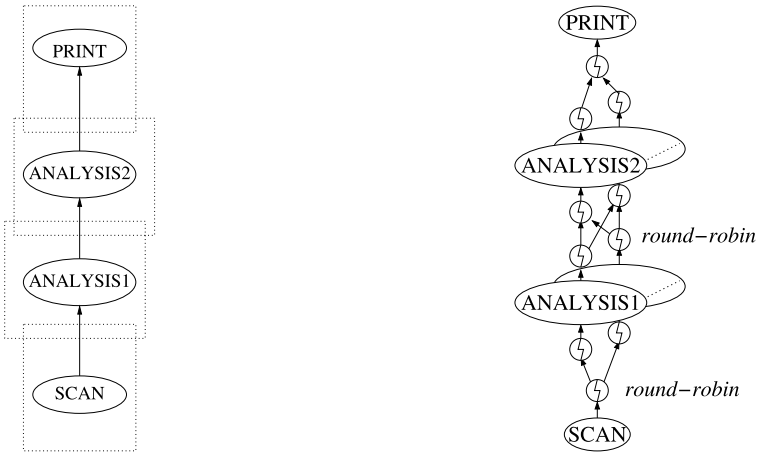

Fig. 5 Planning a query containing two operation calls

The AGQESs and associated software were installed on each available machine in the PlanetLab slice. One of these machines also hosts the benchmark database. Each of the remaining available machines in the PlanetLab slice hosts a service exporting operation *analysis1* or *analysis2*. Each query is initiated from a user workstation at Newcastle and the query results returned to the same machine.

The query compiler generates first a logical plan of the form shown in Fig. 5(a). The plan defines the organization of the key data processing operators required to perform the query. The data is accessed by a *scan* operator and forwarded via two *operation call* operators to a *print* which outputs the result to the user. From the logical plan, the compiler generates a physical plan, by dividing the logical plan into partitions which can be parallelized. In this case, as shown by the dotted boxes, there are four partitions. The compiler recognizes that there must be a single instance of *print*, which just returns the results to the client, and that while the number of instances of *scan* is fixed by the number of copies of the data source, there can independently be multiple instances of each of the *operation call* partitions, depending on the number of instances of the relevant service that are available. Additionally, the physical plan contains instances of an *exchange* operator, which implements data redistribution as required. The scheduler then decides how many copies of each partition should be included in the plan to be submitted for execution. In the executable plan shown in Fig. 5(b), while a single copy of the scan partition is allocated, corresponding to the single data source, each of the operation call partitions is replicated for each machine hosting the relevant operation. The exchange operators are parameterized so as to implement an even *round-robin* redistribution of tuples. This redistribution can be adjusted at runtime by the adaptivity support to achieve a balance when the multiple destinations are found to have unequal throughput.

Figure 6 shows the impact of runtime adaptivity where increasing numbers of machines are used to parallelise the analysis operations (for example, if there are 4

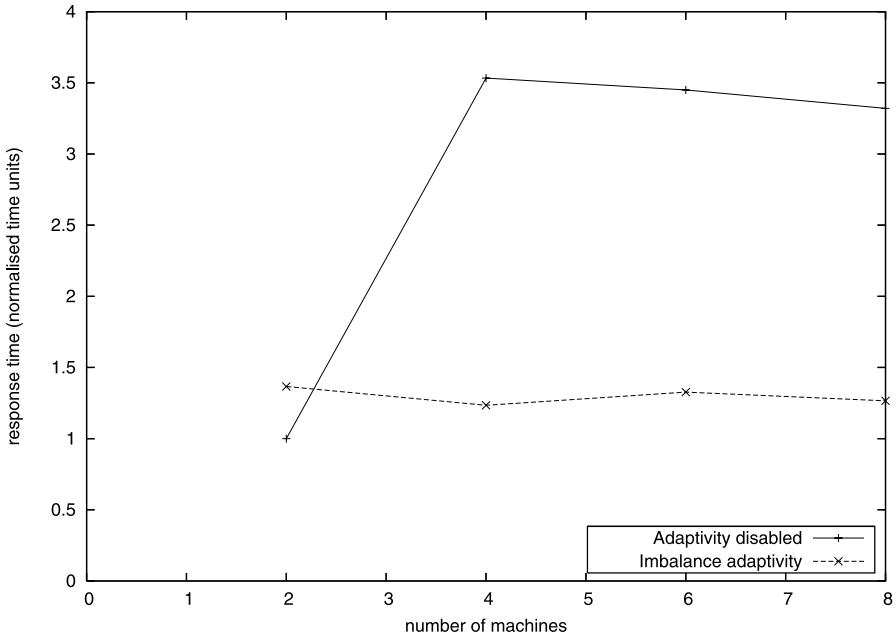

Fig. 6 Results for low-cost operation calls

compute machines, there are 2 instances of each of *analysis1* and *analysis2*). The query uses low cost versions of *analysis1* and *analysis2*, in which there is little scope for parallelism providing speedup because the evaluation of the operation calls is not a substantial portion of the overall cost of the query. Indeed, little or no speedup is exhibited. However, the inclusion of a slow machine in the schedule has a much more detrimental effect on the statically scheduled plan, when compared to the adaptive system, which gives three times better performance. This demonstrates that adaptive load balancing can be used to provide more predictable performance in an unpredictable environment.

Figure 7 shows the impact of runtime adaptivity for more costly versions of *analysis1* and *analysis2*, in which a significant portion of the overall query cost can be reduced by partitioned parallelism. There is obvious performance degradation in the cases where adaptivity is disabled. In these cases, the scheduler can only make an initial placement, based on static parameters, in this case allocating the divisible portions of the computations evenly amongst the available machines. When this is done, the outcome is unpredictable, as the chance selection of a single heavily loaded machine may significantly increase response times. When adaptivity is enabled, by contrast, load is redistributed dynamically throughout the computation based on recently measured performance, allowing profitable use of the available machines and some speedup. In both the experiments, the variability of the capabilities of the participating machines can be seen to make static allocation a risky proposition, while autonomic rebalancing is able to mitigate this risk by continually readjusting the computation in order to exploit the changing set of machines that are performing well.

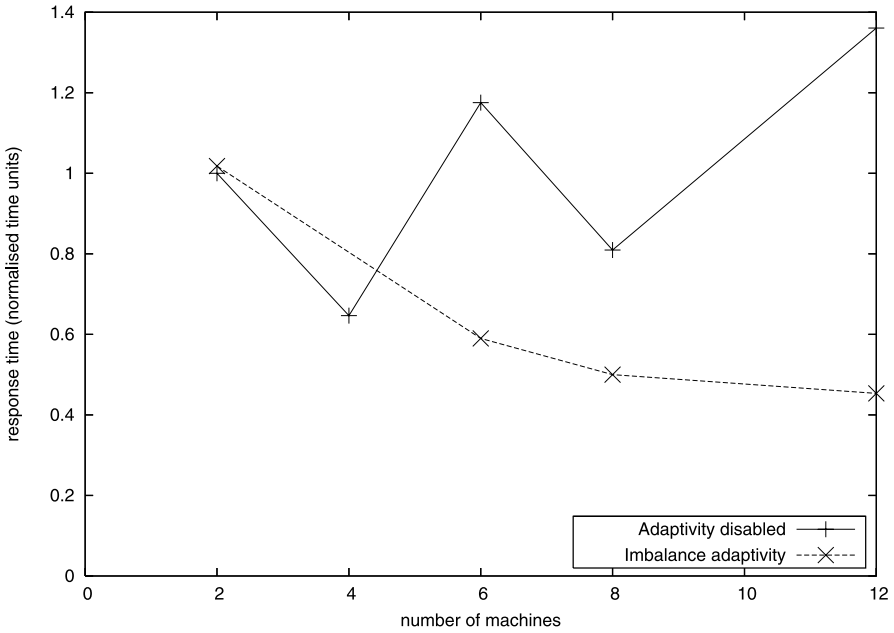

Fig. 7 Results for costly operation calls

Evaluation of the bottleneck removal strategy is deferred to the next subsection, which employs a cluster as the platform for experiments, to render the regeneration of a bottleneck across multiple runs feasible.

4.2 Evaluation in a cluster

4.2.1 Experiment 2: effectiveness of bottleneck removal

The experiments in this section have been conducted using an unloaded cluster consisting of 12 860 MHz PCs, each with 512 Mb of memory, connected using a 100 Mb/s Ethernet. One of the nodes is used as a coordinator (where queries are compiled and to which results are delivered). Queries are scheduled with a single node used for database access, with different nodes used for each operation call in a query.

Experiment 2.1: Resolving a single bottleneck The aim of this experiment is to establish how effective bottleneck removal is in the context of a single bottleneck. The following query is used:

Q2:
```
select analysis1(s.ssequence)
from Sequence s;
```

The difference with *Q1* is that there is just one call to an analysis service. The query is initially compiled in such a way that *Sequence* is accessed on a single node

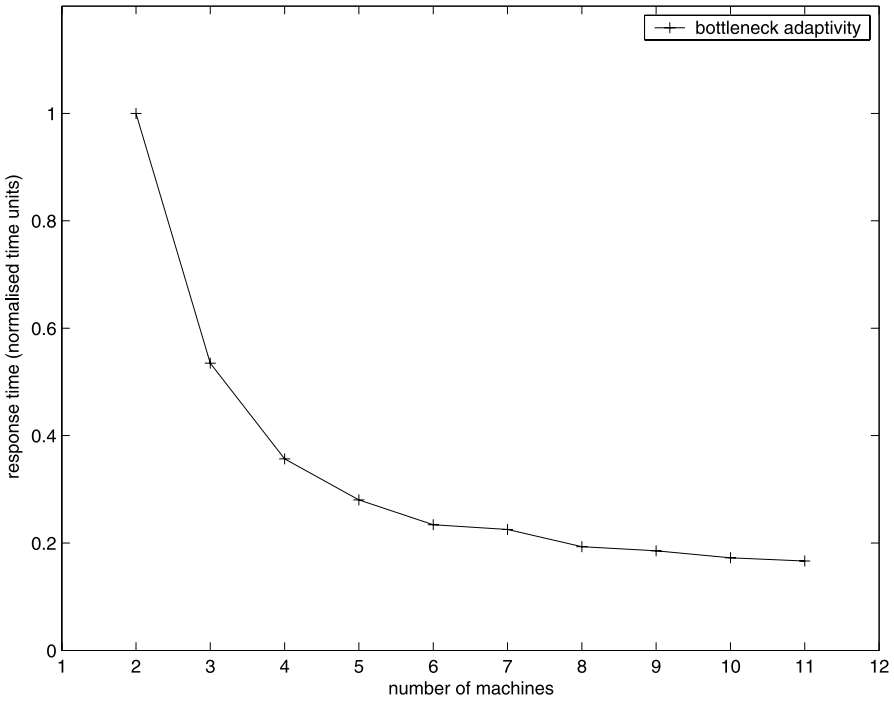

Fig. 8 Performance of Q2 with bottleneck adaptivity for 100,000 tuples

and a single *analysis1* is made available on a different node. Varying numbers of the remaining nodes are made available for absorbing the imbalance. The call to *analysis1* is the bottleneck in this plan, even when it is fully parallelised for any number of available machines considered in the experiment.

Figure 8 shows the impact of increasing the number of nodes with which to absorb the imbalance, running over a collection of 100,000 *Sequence*s. The non-parallel case is with 2 machines, as one machine is used for scanning the *Sequence*s. The figure is normalised such that *1* represents the cost of evaluating the query with adaptivity disabled. In essence, bottleneck removal has been successful at reducing the effects of the bottleneck, with significant speedups. This indicates that the bottleneck has been detected in a timely manner. Moreover, the graph in Fig. 8 is close to what someone would expect for the case where the degree of parallelism was set to its optimal at compile time rather than at runtime, which means that the adaptivity loop and the message exchange between components is efficient since it (i) does not prevent quick adaptations; and (ii) incurs a low overhead. The overall overhead will be examined more thoroughly in subsequent experiments.

Experiment 2.2: resolving multiple bottlenecks The aim of this experiment is to assess the effectiveness of the strategy given bottlenecks at different points in a pipeline.

Initially, the case is considered in which there are different levels of bottleneck at different points in the pipeline. *Q1* is used in this experiment.

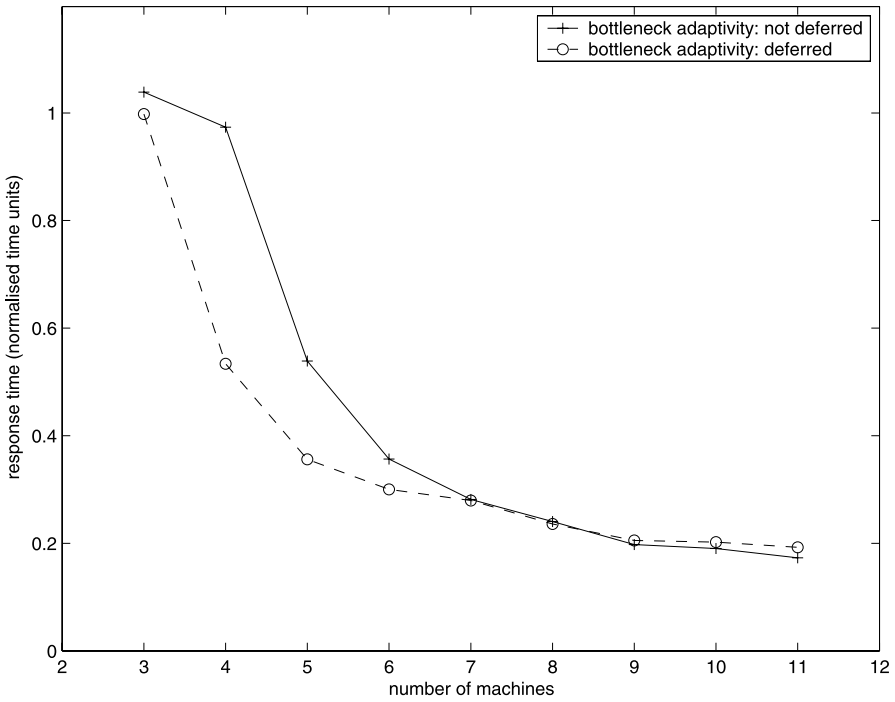

Fig. 9 Performance of Q1 with deferred and non-deferred bottleneck adaptivity. The call to one of the analysis is 5 times slower than to the other

In the first experiment, *analysis2* has been slowed down so that each call takes *5* times as long as *analysis1*. Figure 9 shows the impact of increasing the number of nodes with which to absorb the imbalance, running over a collection of 100,000 *Sequence*s, for both deferred and non-deferred bottleneck resolution. The first opportunity for adaptation is with a parallelism level of *4*, where a single machine is available for absorbing the bottleneck.

The curve for *non deferred* bottleneck resolution improves on the performance of the static case where there are significant numbers of machines available for absorbing the bottleneck. However, the plan has (by chance) been compiled in such a way that the partition containing *analysis2* is the parent of that containing *analysis1*, which in turn is the parent of the scan of *Sequence*. As a result, the (non-deferred) adaptive query processor obtains throughput figures for the evaluation of *analysis1* before it obtains the corresponding figures for *analysis2*, and thus may allocate nodes to the partition for *analysis1* that are then no longer available for resolving the bottleneck based on *analysis2* when it is subsequently shown to be the principal bottleneck. As a result, where only 4 processors are available for evaluating the query the single processor that is available for absorbing the bottleneck is assigned to *analysis1*, and there is little improvement in performance. With *deferred* bottleneck resolution, a more suitable allocation is made when a single machine is available for absorbing imbalance. Where larger numbers of machines are available to absorb the bottleneck, both adaptive strategies improve significantly on the static case.

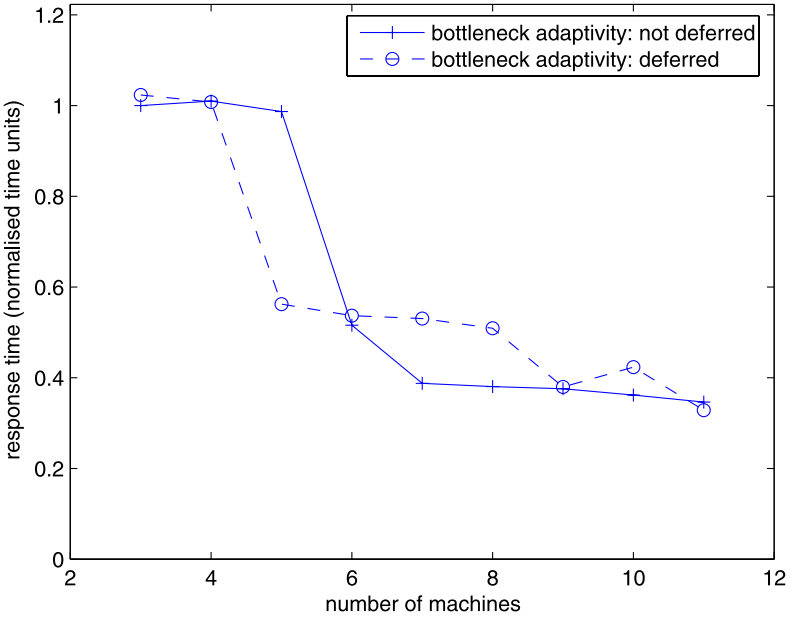

Fig. 10 Performance of Q1 with deferred and non-deferred bottleneck adaptivity. The response time of both the analysis tools per call is equal

In a second experiment, the case is considered in which the same level of bottleneck exists at different points in the pipeline. That is, *analysis2* has been configured so that each call takes as long as for *analysis1*. Figure 10 shows the impact of increasing the number of nodes with which to absorb the imbalance, running over a collection of 100,000 *Sequence*s for both deferred and non-deferred bottleneck resolution.

The first opportunity for adaptation is with a parallelism level of *4*, where a single machine is available for absorbing the bottleneck. Both adaptivity strategies allocate this node to the pipelined partition associated with one of the calls to the analysis tool. However, this has little effect on overall response time, as the bottleneck remains associated with the other call to analysis. Where there is a parallelism level of *5*, with two machines available for absorbing the bottleneck, the deferred approach appropriately allocates two machines for each of the partitions associated with the calls to the analysis operation, and obtains a significant speedup. However, the non-deferred approach allocates both the additional machines to the partitions nearest the scan, and thus leads to little overall performance improvement. With the higher levels of parallelism, both adaptive strategies significantly improve on the static case. Where the total number of machines is 7 and 8 (i.e., there are, respectively, 4 and 5 machines available to absorb the bottleneck), the *non-deferred* case has made better allocations of operation calls to machines than the *deferred* case; this does not result from any fundamental property of the methods—allocation decisions are made on the basis of dynamic feedback from monitoring of parallel plan fragments. Such dynamic feedback provides an evolving view of the behaviour of a complex software system, and at any point in time may give a somewhat misleading picture to the diagnoser and

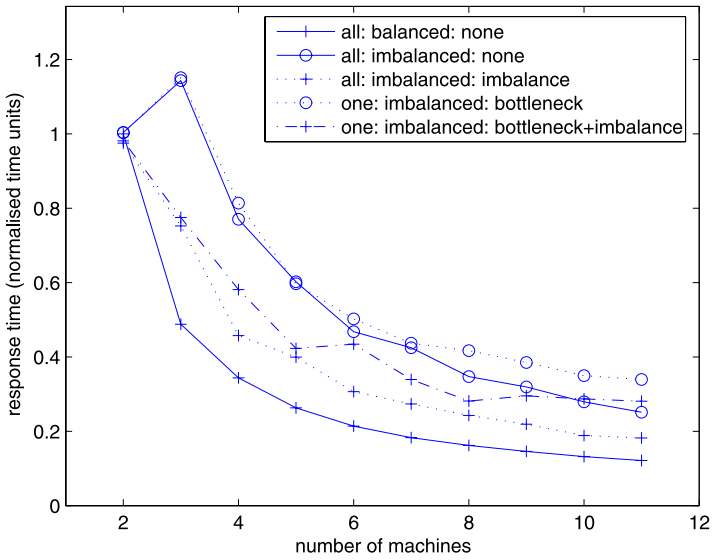

Fig. 11 Performance of Q2 for different adaptive strategies

responder. The lesson learnt from this experiment is that when there is a very limited number of additional machines available, *deferred* adaptivity should be preferred to *non-deferred*, as it may lead to more efficient use of those machines.

4.2.2 Experiment 3: Combining and comparing the adaptivity strategies

This section compares the use of the techniques for adapting to load imbalances and bottlenecks, both together and separately.

The experiment uses *Q2*, in an environment in which machines *3, 5, 7, 9* and *11* perform operation calls approximately 2.5 times more slowly than those allocated to machines *2, 4, 6, 8* and *10*, where machines are made available in the experiments in numerical order.

Figure 11 shows the impact of increasing the number of nodes with which to evaluate *Q2* running over a collection of 100,000 *Sequence*s. The plots on the graph have legends of the form *initial-allocation: environment-description: strategies-enabled*, where:

- *initial-allocation* is either *one* or *all*, to indicate if the plan is initially scheduled with the call to *analysis1* on a single machine or on every machine available
- *environment-description* is either *balanced* or *imbalanced*, indicating whether or not the operation calls allocated to odd numbered nodes have been slowed down; and
- *strategies-enabled* is either *none*, *bottleneck*, *imbalance* or *imbalance+bottleneck*, indicating which of the adaptivity strategies are enabled. When both are enabled, bottleneck removal is applied after the execution has been balanced.

The following observations can be made for the different traces in Fig. 11:

1. *all: balanced: none*. This configuration is essentially the best case performance on the available hardware, as the query is evaluated with maximum parallelism on an otherwise unloaded machine. No adaptivity strategies are enabled or required.
2. *all: imbalanced: none*. This configuration shows the effect of the introduction of imbalance in the absence of the adaptation that seeks to restore balanced load. Where three machines are available for running the query, performance is degraded by the even distribution of work across the two machines running *analysis1*, one of which is significantly slower than the other. Although increasing parallelism improves performance from the worst case by reducing the size of the dataset(s) allocated to the slowest machine(s), this only slightly improves on the performance of the static case with the level of parallelism available. Note that the reasonably strong performance with two processors is just chance—had response times of operations been slowed down on even numbered rather than odd numbered nodes, the poorest performance would have been for 2 machines rather than 3.
3. *all: imbalanced: imbalance*. This configuration shows the effect of the adaptation designed to remove imbalance on the configuration at (2). Overall, the adaptations have been effective, providing steadily improving performance with increasing levels of parallelism, even though odd numbered machines are significantly slower than their even numbered counterparts. By sending the appropriate amount of tuples to slower machines, performance improvements are observed for all machine additions. However, these improvements are less significant when adding slower machines, as shown by the changing angle of gradient in the graph.
4. *one: imbalanced: bottleneck*. This configuration shows how the bottleneck imbalance strategy adds parallelism to the call to *analysis1* from an initial allocation to a single node. This configuration is closest to (2), with the only difference being that in this case the parallelism is added by bottleneck adaptation rather than by the scheduler before query evaluation. The bottleneck adaptation is effective at exploiting the available parallelism, but as in the static case suffers significantly from the presence of slower machines.
5. *one: imbalanced: bottleneck+imbalance*. This configuration shows the effect of the introduction of imbalance adaptivity to configuration (4). For all parallel configurations, the combined adaptivity strategies improve significantly on the behaviour obtained in configuration (4), although the combination does not quite match the behaviour of configuration (3) where the available computational resources are the same. This is because the incremental introduction of parallelism inevitably leads to slower exploitation of the available machines, and because the interplay of the two adaptivity strategies requires greater numbers of adaptations to arrive at a steady state; the difference equals to the number of machines available minus one.

Overall, the improvement in the response time of the query due to the adaptivity techniques is shown by comparing (2) and (3); and (4) and (5). The former refers to the results of the online balancing technique, whereas the latter relates to the combination of bottleneck removal with balancing. Finally, the difference between (2) and (4), and (3) and (5) denotes the overhead of the bottleneck removal strategies.

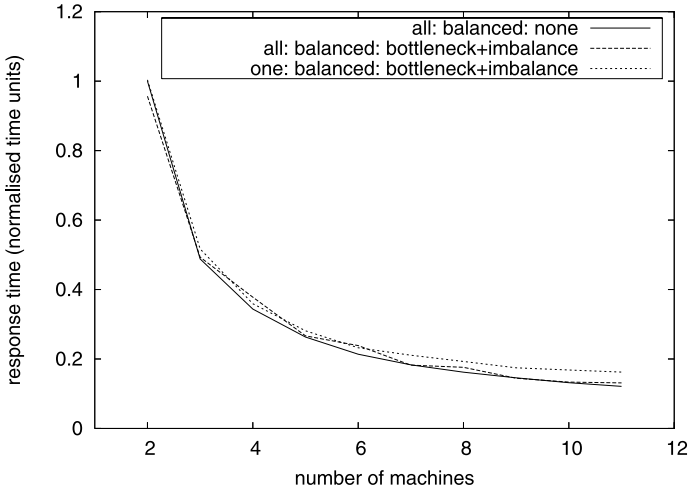

Fig. 12 Overheads for different adaptive strategies

The overheads are illustrated more explicitly in Fig. 12. To derive the overhead of the two adaptivity techniques running in combination, the first and the third plots need to be compared. Their difference corresponds to the cost of monitoring, assessing and responding in both ways examined. The first two plots give the overhead of the balancing strategy (i.e., overhead because of monitoring, assessing and modifying the workload distribution), whereas the overhead of the stand-alone bottleneck removal technique can be derived from the second and the third plots.

4.3 Evaluation in a small scale controlled environment

The experiments presented in this section explore the costs and benefits of redistributing the tuple workload on the fly to keep the evaluation balanced across evaluators. Throughout this section, experiments are carried out in a small-scale homogeneous environment into which imbalances are introduced, to allow detailed study of the behaviour of the adaptive techniques in a controlled setting.

Two example queries are used, *Q2* and *Q3*, which is as follows:

Q3:
```
select i.ORF2
from Sequence s, Interaction i
where i.proteinid=s.proteinid;
```

The tables *Sequence* and *Interaction* are from the OGSA-DQP demo database and they contain data on proteins and results of a bioinformatics experiment, respectively (the *Sequence* collection used in the experiments is slightly modified to make all the tuples the same length to facilitate result analysis). Unless otherwise stated, *Sequence* in this experiment contains 3000 tuples and *Interaction* contains 4700 tuples.

Both Q2 and Q3 are compiled in such a way that each table is accessed from a single node. Partitioned parallelism is used in Q2 for the calls to *analysis1* and in Q3 for the join. The join is implemented as a blocking hash join algorithm, with the hash table being cloned to all participating nodes. The pipeline fragment that is subject

to adaptivity consists of a *scan* operator, sending data to the *hash join* to probe the already built table, and subsequently to return the results through a *print*. As such, the adaptable query plan fragments of *Q2* and *Q3* are quite similar; in the former the middle operator is an *operation call* accessing *analysis1*, whilst in the latter it is a *hash join*.

In the previous experiments it was shown that the adaptations described can be applied to a large number of machines. However, to enable carefully controlled experiments to be conducted, and to ease interpretation of the behaviour of the system for specific kinds of adaptation, two machines are used in this experiment for the evaluation of *analysis1* in Q2, and the join in Q3, unless otherwise stated. The data are retrieved from a third machine. All machines are identical, run RH Linux 9, are connected by a 100 Mbps network, and are autonomously exposed as grid resources. It was ensured that they were unloaded at the time of the experiments to allow a detailed analysis of the behaviour of the adaptive techniques. The third machine retrieves and sends data to the first two as fast as it can.

For each result, the query was run three times after the system has been warmed up to ensure that in all cases the standard deviation remains below 5%, and the average is presented here. Finally, we have used two methods to create artificial load for machine perturbation: (i) by programming a computation to iterate over the same function multiple times, and (ii) by inserting *sleep()* calls. In the previous experiments, only the former method has been adopted.

4.3.1 Experiment 4: Behaviour of load balancing strategy

The objective of this experiment is to understand how effective the load balancing strategies described in Sect. 3.1 are, by measuring the performance of different configurations in a controlled environment.

Experiment 4.1: Alternative forms of response The objective of this experiment is to compare alternative strategies in the presence and absence of imbalance.

The following configurations are considered:

- *no ad/no imb:* there is no imbalance between the performance of the *analysis1* services in the two machines, and adaptivity is not enabled (bottomline case);
- *ad/no imb:* there is no imbalance between the two services, and adaptivity is enabled, so that the overhead can be revealed;
- *no ad/imb:* one WS call is costlier than the other, thus there is imbalance between the two services. Adaptivity is not enabled, so that the performance degradation due to imbalance can be revealed; and
- *ad/imb:* there is imbalance between the two services, and adaptivity is enabled, so that the improvements due to adaptivity can be revealed.

To create the imbalance in the first experiment, we set the cost of the WS call in Q2 in one machine to be exactly 10 times more than in the other for the whole duration of the query. As will be discussed later, this slows down the overall execution on one machine by a factor of 3.5 approximately. The responses are prospective (response type R2 in Sect. 3.1). The first row of Table 1 shows how the system behaves under different configurations.

Table 1 Performance of queries in normalised units

Query-Response	no ad/no imb	ad/no imb	no ad/imb	ad/imb
Q2-R2	1	1.059	3.53	1.45
Q2-R1	1	1.15	3.53	1.57
Q3-R1	1	1.11	1.71	1.31

The results are normalised, so that the response time corresponding to *no ad/no imb* is set to 1 unit for each query. The scale 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),[3] which can be computed by the difference of the second and the third column of Table 1 (1st row). This difference is 5.9%, and is due to the publishing of monitoring messages. 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 cost. Thus, a 10-fold increase in the WS cost, results in 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.

The 2nd row in Table 1 shows the results for the same experimental setup, except that the adaptation is retrospective (type R1 of response). When the adaptivity is not enabled (*no ad/imb*), the response time 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. On average, the size of the state recreated is 10% of the overall size (3000 tuples) approximately. 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 Q3 as well, using the second method to create imbalance artificially. The factor of performance degradation due to imbalance is less than 1 in these runs (0.71), to give an example of small imbalances. 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.

Experiment 4.2: Varying the size of perturbation The objective of this experiment is to compare the strategies for different levels of imbalance.

[3] 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.

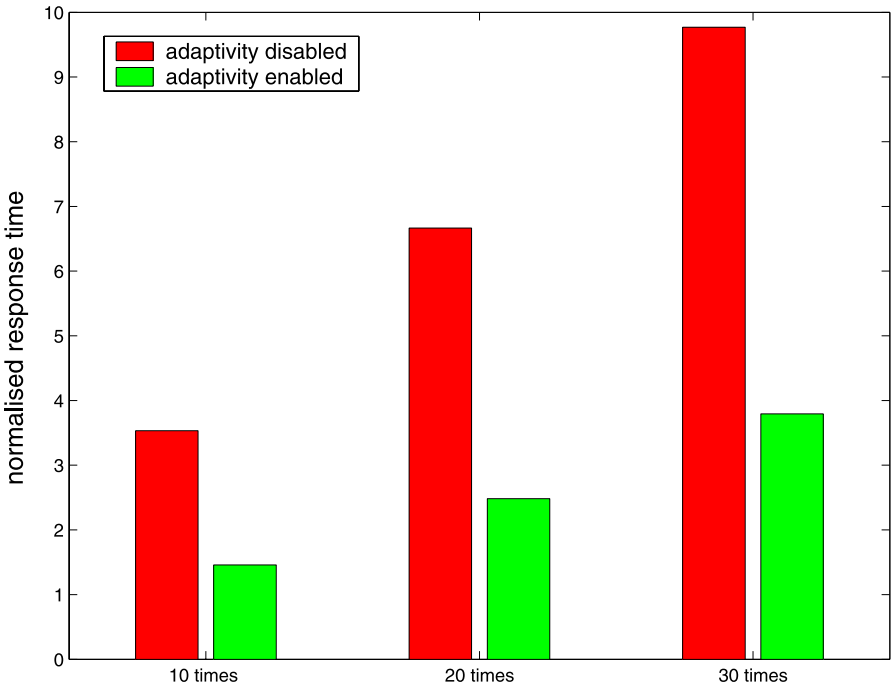

Fig. 13 Performance of Q2 for prospective adaptations

We reran Q2 for the cases in which the perturbed WS is 10, 20 and 30 times costlier, and adaptations are prospective. When the perturbed WS is 30 costlier, the increase in the overall query response time is close to an order of magnitude. Figure 13 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 of several factors, consistently.

Experiment 4.3: Effects of different policies The objective of this experiment is to compare assessment and response policies.

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

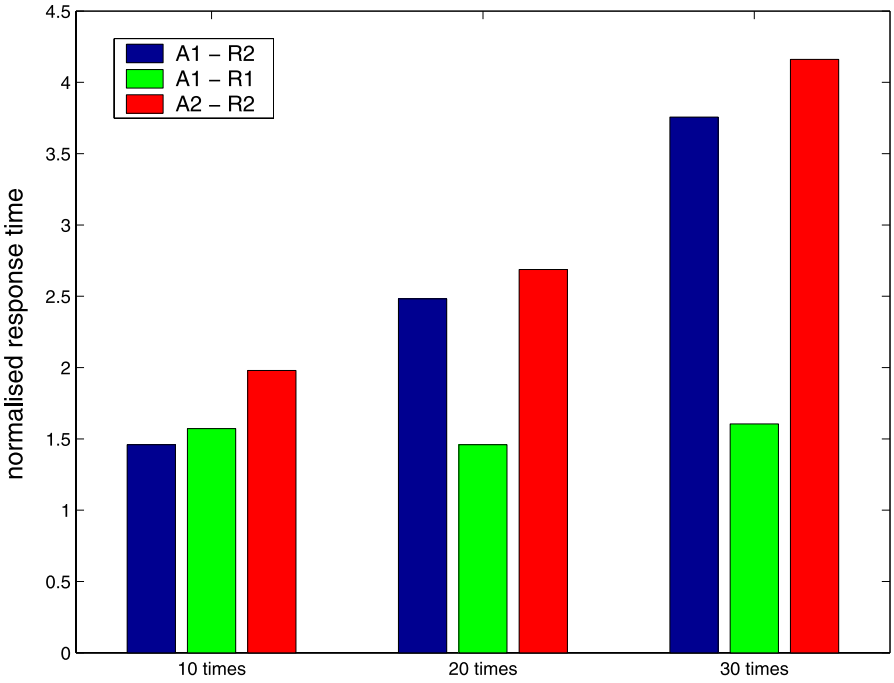

Fig. 14 Performance of Q2 for different adaptivity policies

data to the subplan examined for imbalance, and no state is recreated (type A2 of assessment combined with type R2 of response).

There are two important factors that impact on the implications of communication costs, namely the execution model and the network capacity with regard to the computational requirements of the query. When communications tasks overlap with expensive computational tasks, then the communication cost is hidden regardless of the plan topology, and since communication costs does not dominate, it can be omitted during the assessment/response phases. In other execution models (e.g., synchronous data delivery where the processing and transmission of data occurs sequentially) this might not be the case, even if the communication cost does not dominate. Our framework supports both cases during assessment; however its instantiation, which is an extension to the OGSA-DQP system, fits better to the first scenario. In essence, in our prototype, 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. Such an assumption is valid, and indeed, this is verified by the experimental results discussed next.

The performance of the three configurations for Q2 is shown in Fig. 14. 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 into consideration the pipelining, 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

Table 2 Ratio of tuples sent to the two evaluators

Case	A1-R2	A1-R1	A2-R2
10 times	5.58	11.21	3.16
20 times	4.95	11.42	4.33
30 times	4.55	16.45	3.89

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 re-sending them to the faster ones increases for bigger perturbations. Also, from Fig. 14, it can be 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 Q3 lead to the same conclusions. Figure 15 shows the behaviour of the join query when the *sleep()* process sleeps for 10, 50 and 100 ms, respectively, and adaptations are of type A1 of assessment and R1 of response. As already identified in Fig. 14, retrospective adaptations are characterised by better scalability, and their performance is less dependent on the perturbation.

Table 2 corresponds to Fig. 14 and shows the ratio[4] of the number of tuples sent to the two evaluators calling the WSs. The ratio is significantly higher for retrospective adaptations, which means that the system manages, in practice, to reroute data according to the performance of the evaluators. For prospective adaptations, for this demo query, although the monitoring information is the same, rerouting is not as effective. This is because the dataset is relatively small, and by the time load imbalance has been detected, a significant number of tuples has already been sent to its consumers. In retrospective adaptations, these tuples are redistributed, whereas such a redistribution cannot happen in the prospective ones. However, as will be demonstrated later, this is mitigated when the dataset size increases.

Experiment 4.4: Varying the dataset size The objective of this experiment is to explore the effect of dataset size on the effectiveness of the adaptive load balancing.

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 Q2

[4]ratio = number of tuples sent to the faster machine/number of tuples sent to the slower machine.

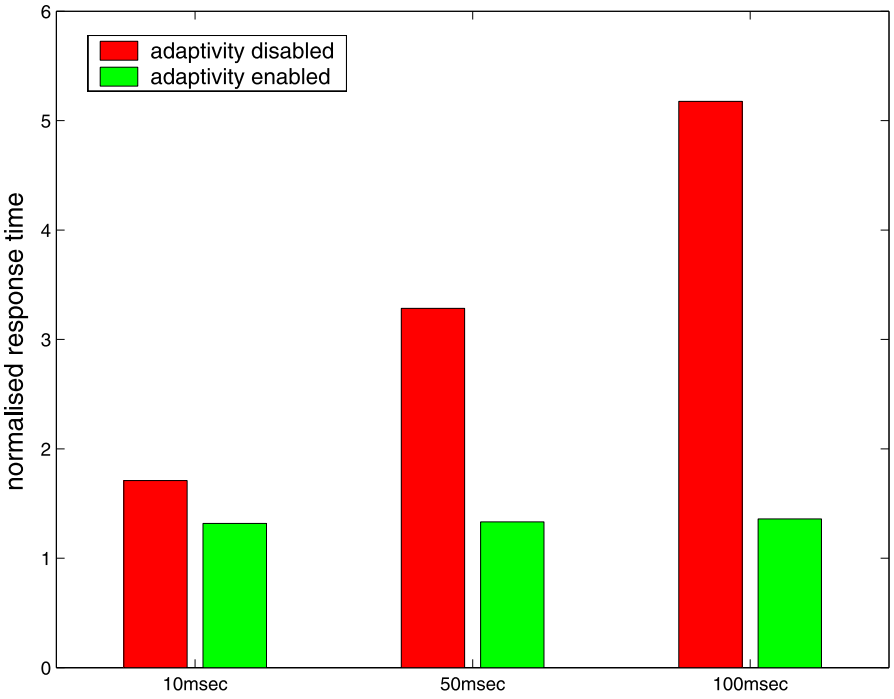

Fig. 15 Performance of Q2 for retrospective adaptations

from 3000 tuples to 6000, and making one WS call 10, 20 and 30 times costlier than the other, while the adaptations are prospective. Figure 16 shows the results, which are very close to those when adaptations are retrospective (i.e., Fig. 14 for Q2 and Fig. 15 for Q3 compared to Fig. 13), and lead to better performance improvements.

In summary, there is a trade-off between retrospective and prospective adaptations. The former are costlier but the latter suffer from poor performance when the time to process the tuples that have been distributed in a suboptimal manner is significant compared to the overall remaining execution time. An interesting extension to our work, left for the future, is to develop a cost model that quantifies these trade-offs with a view to deciding whether to employ retrospective or prospective adaptations on the fly.

Experiment 4.5: Varying the number of perturbed machines The objective of this experiment is to identify the effectiveness of the different approaches where varying numbers of machines exhibit reduced performance.

Figure 17 shows the performance of Q2 for different numbers of perturbed machines when adaptations are retrospective (three machines have been used for WS evaluation in this experiment). Perturbations are inserted by making selected WS calls 10 times costlier than the others. Without load balancing, the presence of an imbalance affecting any node substantially delays completion of the query. By contrast, when load balancing is enabled, performance degrades gracefully in the presence of

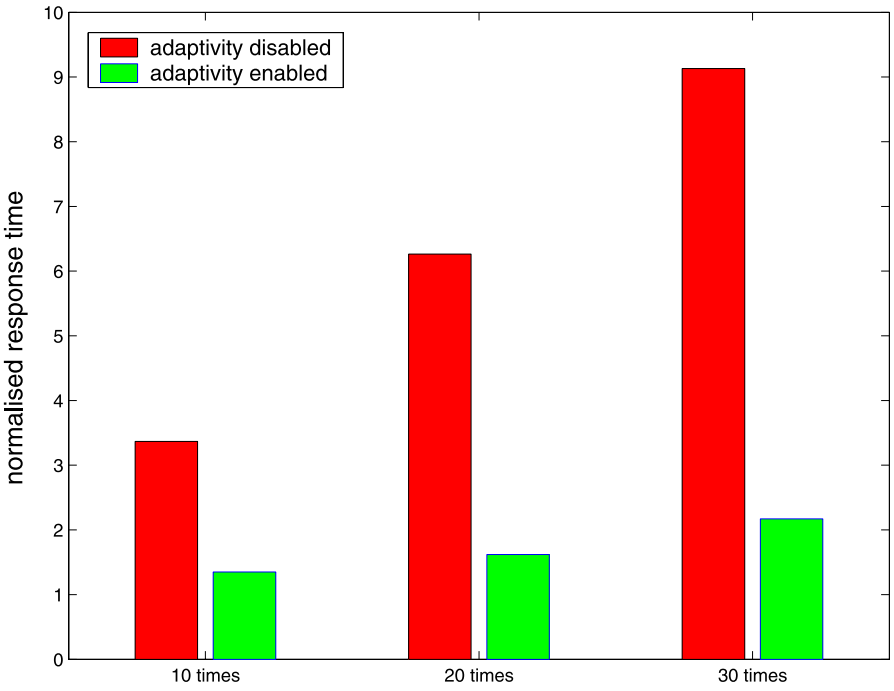

Fig. 16 Performance of Q1 for prospective adaptations and double data size

perturbed machines until such time as all machines are perturbed, and there is no way to avoid substantially reduced performance.

4.3.2 Experiment 5: Impact of monitoring frequency

The objective of this experiment is to complement the discussion of Experiment 4.1 on the overheads associated with adaptive load balancing and discuss the impact of monitoring frequency.

4.3.2.1 Experiment 5.1: Monitoring overhead for different frequencies of monitoring Figure 18 presents the behaviour of the system for Q2, 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 (2nd and 4th plots) and the overhead incurred (1st and 3rd plots) are rather insensitive to these monitoring frequencies. This is because (i) the mechanism to produce low-level monitoring notifications has been shown to have very low overhead [19], 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.

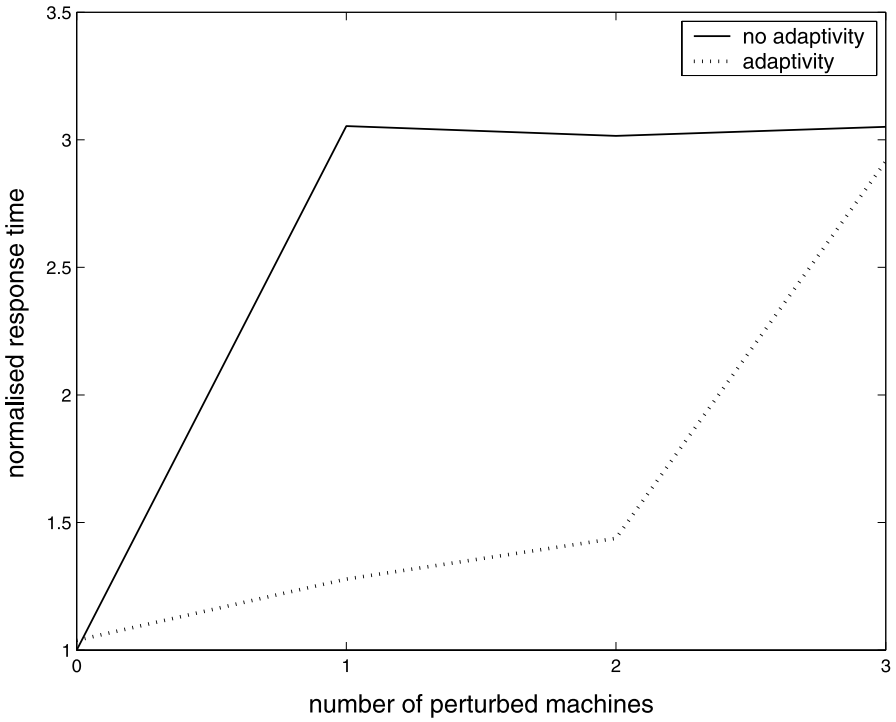

Fig. 17 Performance of Q2 for retrospective adaptations

4.3.3 Experiment 6: Changing load

The objective of this experiment is to understand how effective the load balancing strategies are in the context of rapidly changing loads instead of loads which are stable during the whole duration of the query execution.

Experiment 6.1: The effectiveness of load balancing for different levels of change in imbalance 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. Figure 19 shows the results when the differences in the two WS costs in Q2 vary between 25 and 35 times, between 20 and 40 times, and between 1 and 60 times, and the adaptations are stateless. The leftmost bar in each group in the figure corresponds to a stable cost, which is 30 times higher (e.g., bar A1-R2, 30 times in Fig. 14 for prospective adaptations), and is presented again for comparison purposes. For the ranges explored, the system is effective and efficient in handling different levels of imbalance and different frequencies in which imbalance occurs.

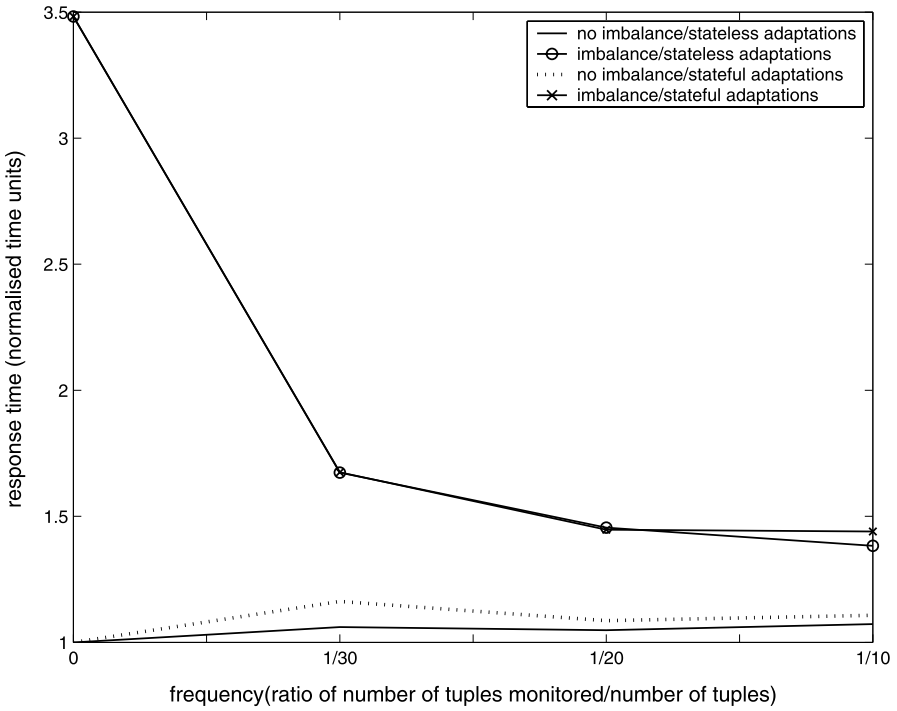

Fig. 18 Effects of different monitoring frequencies in Q2

4.4 Generalization to larger queries

The plans of the example queries comprise up to three operators, at most two of which can be parallelized. However, the results presented can be generalized to arbitrarily larger queries in terms of degree of operator parallelism, or number of operators in the query plan, or their combination. This is because the response phase of the adaptations impacts solely on individual operators; thus it is independent of the size of the query plan. Moreover, the assessment phase during operator load balancing involves the estimation of a vector with length equal to the degree of the operator parallelism; practically, this cost is negligible even for very large degrees. In bottleneck reduction, the assessment phase includes sorting the partition instances and then the partitions by their cost; again, typically, this cost is not significant. Aggregate monitoring costs may increase significantly for larger queries. However, our framework naturally supports the case in which each operator is subscribed to a different assessment/response component, thus achieving scalability. As such, the impact of monitoring on the total response time is expected to be similar to that reported in the previous sections.

5 Related work

Adaptive query processing is an active research area [6, 16]; solutions have been developed to compensate for inaccurate or unavailable data properties (e.g., [4, 5, 29,

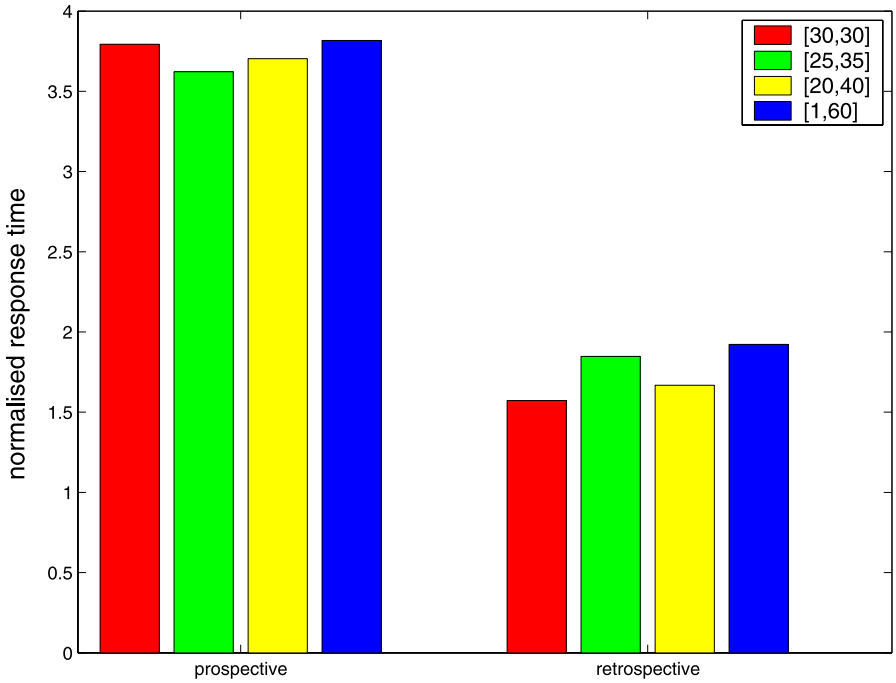

Fig. 19 Performance of Q2 under changing perturbations

32]), to manage bursty data retrieval rates from remote sources (e.g., [25]), and to provide prioritized results as early as possible (e.g., [41]). In works such as [5, 32], the focus is on changing the query plan at runtime in light of new statistical information becoming available. In summary, current adaptive query processing techniques assume centralized processing (even when data are retrieved from remote sources) and either change the query plan on the fly (e.g., [30]) or continuously reroute tuples through operators; Eddies [4] is a pioneer of the latter approach. The adaptations considered in this paper are finer grained than most approaches that involve reoptimization, and adaptations seek to respond to inaccurate or unstable resource properties for distributed queries rather than to inaccurate statistics about the data to be processed. As such, one can see this work as complementing, rather than superceding, or being superceded by, the work to date on adaptive reoptimization. The reality of work in adaptive query processing is that individual papers tend to consider only one kind of adaptation in response to one kind of challenge. Our paper describes two related kinds of adaptation (to address load imbalance and bottlenecks) and a specific kind of challenge (uncertain or changing resource properties or query resource requirements in distributed systems). Both the kinds of adaptation considered and the challenge dealt with have been overlooked by other interesting adaptive proposals [16]. To the best of our knowledge, our work is the first that considers adaptations to changing machine capabilities by deciding on the workload allocation and the number of the machines employed on the fly and is tailored to partitioned pipelined execution over heterogeneous resources. An interesting direction for future work is to investigate the

combination of adaptations to updated information about both data and resources and changes to both the query plan shape and the workload allocation.

In a distributed setting, [27] deals with adaptations to changing statistics of data from remote sources, whereas our proposal, complementarily, focuses on changing resources. Moreover, sources in [27] only provide data, and do not otherwise contribute to the query evaluation, which takes place centrally. Eddies [4] are also used in centralised processing of data streams to adapt to changing data characteristics (e.g., [8]) and operator consumption speeds. When Eddies are distributed, as in [48, 53], changes to consumption speeds may indicate changing resource performance. Nevertheless, our approach differs in several significant aspects: (i) the emphasis is on parallel query processing, and thus in the case of adaptive load balancing on changing the routing of data within rather than between operators; (ii) there is the potential to add to the resource pool during query evaluation, as in the case of adaptive bottleneck resolution; although proposals based on Eddies may dynamically reduce the number of tuples that ever pass through the operator that is responsible for a bottleneck, the option of allocating additional resource to the evaluation of that operator is not considered. The use of Eddies in parallel query processing is discussed in [39], where an Eddy operator splits output tuples over several query evaluators and merges incoming responses from those evaluators; in the case of hash joins, the dynamic redistribution of data is avoided by allocating the complete hash table to multiple parallel evaluators. Such an approach should be effective for join queries in which one operand is very much smaller than the other, although a central Eddy operator may be a bottleneck for higher levels of parallelism.

There are, however, two particularly relevant pieces of work on adaptive load balancing. For data and state repartitioning, the most relevant work is the Flux operator for continuous queries [42], which extends exchanges and uses partitioned parallelism. Flux operates in the same context as the work supported in our paper, i.e., heterogeneous machines, although with an emphasis on stream query processing, but it does not address bottlenecks and it does not consider employing additional machines on the fly. Flux differs from the adaptive load balancing approach described here by moving operator state between sibling operator fragments, rather than by resending data from upstream caches. We see the following benefits from the use of upstream caches: (i) the work required to extract the data for reallocation does not further load the machine(s) that have been detected as the source of the problem; (ii) extraction of state for reallocation from buffers is independent of the algorithm storing the state, reducing development costs because there is no need for different state-extraction functionality to be written for different stateful operators; (iii) the cache provides support for fault tolerance with quite modest overheads [45], whereas the fault tolerance scheme associated with Flux does substantial amounts of redundant work [43]. Also, our paper investigates the relative merits of different state movement strategies (prospective and retrospective) that can have a significant impact on performance. A more direct comparison of the performance of different load balancing strategies can be found in [38].

A further strategy for adapting to load imbalance is part of the Data In The Network (DITN) proposal [40]. In DITN, redundant plan fragments are executed when a fragment is late completing. In such an approach, however, adaptation may take

place after a prolonged period of imbalance; as a result, DITN is expected to be less responsive to changes in load balance than the approaches of Flux or OGSA-DQP, but there is no risk of the strategy thrashing (by adapting repeatedly to an unstable environment). In addition, there are circumstances in which DITN does significant additional work during query evaluation to reduce the level of coupling between plan fragments, which means that overheads can be significant.

Rivers [3] follow a simpler approach than Flux, but are capable of performing only data (and not state) repartitioning. State management has also been considered in [15], but only with a view to allowing more efficient, adaptive tuple rerouting within a single-node query plan. Finally, [54] has examined possible operator state management techniques to be used in any single-node adaptation.

There are also several pieces of work of relevance to bottleneck resolution. For example, adapting the allocation of plan fragments to resources has been considered in the Aurora distributed stream query processing system; the relevant paper discusses a wide range of options, but does not provide a detailed description of algorithms or evaluation results [10]. In the follow-on project, Borealis [51], operators may be reallocated between pairs or groups of similar rather than heterogeneous nodes, although there is no discussion of parallel query evaluation. The operator reallocations should help to resolve bottlenecks, although bottlenecks are not explicitly diagnosed, as they are in this paper.

Some forms of reoptimisation of parallel plans, including using new machines on the fly, are presented in [34], although this approach is less general than that presented here, since it can be applied only to a limited range of unary operators. In addition, some proposals defer the machine scheduling decisions until more accurate information about data statistics becomes available; however they suffer from significant limitations such as assuming that all machines available have the same characteristics (e.g., [23]) or do not consider intra-operator parallelism (e.g., [37, 52]). In [25], substitution of data sources on the fly is supported to tackle data source failure; by contrast, the work described in this paper applies to resources that provide both data and computations, and adapts principally to improve performance rather than to provide fault tolerance.

In general, work on distributed query processing over wide-area autonomous environments has resulted in many interesting proposals such as WSMS [46], ObjectGlobe [7], Garlic [28] and Mariposa [47], but has directed fairly little attention towards issues of intra-query adaptivity. In the Grid setting, Polar* [44], OGSA-DQP [1], GridDB [31] and GridDB-Lite [33] are examples of grid-enabled database systems that support access to distributed data resources, and exploit the parallelism available through heterogeneous infrastructures to meet demanding application requirements. However, none of the systems mentioned above tackles the adaptivity issues investigated in this work. Finally, the work in [49, 50] examine wide-area query processing from another perspective, focusing on the communication costs and the efficient network utilization. Our work can take communication costs into account but the adaptations are designed with a view to reducing the overall response time.

Recently, new paradigms for parallel data processing have emerged, such as MapReduce [13, 14] that perform load balancing at a higher level. Our techniques operate under assumptions that are different from those of MapReduce, and that bear

directly on how to address the issues of load rebalancing and bottleneck removal. Most notably, in MapReduce, good knowledge about the properties of the data to be processed and the computational requirements of the number of machines allocated to the map and reduce tasks is assumed. This knowledge drives the static decisions on the way source data is partitioned across nodes and the degree of partitioned parallelism. By contrast, our system is tailored to more autonomous environments, where this knowledge is incomplete, data cannot be removed from store before query execution to facilitate processing and these decisions are taken—and continuously refined—on the fly in response to the feedback collected at runtime. MapReduce relies on independent parallelism, whereas our work deals with the other two aspects of parallelism in query plans, namely partitioned and pipelined parallelism. In MapReduce, it is assumed that there is full control on the source data; by contrast, in our work we assume that the data sources are autonomous, and the place from where they can be retrieved is fixed. This has an impact on the way load balancing is enforced. MapReduce can effectively parallelise the process of retrieving data from source, and can achieve load balancing by allocating large chunks of data to separate machines in a way that faster machines process more chunks. In our approach, we perform load balancing at a finer level of granularity as tuples are routed to the machines according to their processing speed, which is monitored at runtime. In summary, MapReduce is geared towards the efficient parallelisation of independent tasks in a managed environment assuming full control of input data and adequate knowledge about the computational requirements of the task to be executed. Our solutions refer to a more wild setting, in which data is retrieved from predefined sources, there is no a-priori knowledge about the computational requirements of the task, and, as such, the only way to achieve good performance is to respond to empirical evidence collected on the fly by adapting sensibly. Finally, it is an open issue as to whether, and to which extent, query processing can benefit from the map-reduce framework. Unary operators such as selection and group by can be easily re-implemented as map-reduce functions, but this is may not be the case for other query operators. Such issues are examined in [11, 35].

6 Conclusions

The volatility of the environment provided for parallel query processing over heterogeneous and autonomous wide-area resources makes it imperative to adapt to changing resource properties, in order to avoid serious performance degradation. This paper proposes two solutions, one for dynamic load balancing through data and operator state repartitioning, and another for dynamic bottleneck resolution through resource allocation. Both solutions are instantiated in the context of the same generic architectural framework for constructing adaptive techniques. They are implemented through extensions to a distributed query processor for service-based grids. The implementation is particularly appealing for environments such as the grid and cloud computing, as it is based on loosely-coupled components, engineered as web services, which communicate asynchronously and support the publish/subscribe model. For state repartitioning, and with a view to software component reuse, the approach

adopted uses recovery logs that are kept for the purposes of fault tolerance guarantees. The results of the empirical evaluation are promising: performance is significantly improved in a variety of contexts (by an order of magnitude in some cases), while the overhead remains low enough to allow the benefits of adaptation to outweigh its cost for a wide range of scenarios. The results of this work pave the way towards further research in this area. Of particular interest are the topics of combining the adaptivity strategies of this paper with ones that modify the query plan on the fly, and of configuring the numerous tunable parameters of these strategies in an optimal, or near-optimal, way. These topics are left for future work.

Acknowledgements This work has been supported by the Engineering and Physical Sciences Research Council Distributed Information Management and e-Science programmes. This work was conducted while the first author was with the University of Manchester, UK.

References

1. Alpdemir, M.N., Mukherjee, A., Paton, N.W., Watson, P., Fernandes, A.A.A., Gounaris, A., Smith, J.: Service-based distributed querying on the grid. In: Proc. 1st ICSOC, pp. 467–482. Springer, Berlin (2003)
2. Antonioletti, M., Atkinson, M., Baxter, R., Borley, A., Chue Hong, N.P., Collins, B., Hardman, N., Hulme, A.C., Knox, A., Jackson, M., Krause, A., Laws, S., Magowan, J., Paton, N.W., Pearson, D., Sugden, T., Watson, P., Westhead, M.: The design and implementation of grid database services in OGSA-DAI. Concurr. Pract. Exper. **17**, 357–376 (2005)
3. Arpaci-Dusseau, R., Anderson, E., Treuhaft, N., Culler, D., Hellerstein, J., Patterson, D., Yelick, K.: Cluster I/O with river: making the fast case common. In: Proc. of the Sixth IOPADS Workshop, pp. 10–22 (1999)
4. Avnur, R., Hellerstein, J.: Eddies: continuously adaptive query processing. In: Proc. of ACM SIGMOD 2000, pp. 261–272 (2000)
5. Babu, S., Bizarro, P., DeWitt, D.: Proactive re-optimization. In: Proc. ACM SIGMOD, pp. 107–118 (2005)
6. Babu, S., Bizarro, P.: Adaptive query processing in the looking glass. In: CIDR, pp. 238–249 (2005)
7. Braumandl, R., Keidl, M., Kemper, A., Kossmann, K., Kreutz, A., Seltzsam, S., Stocker, K.: ObjectGlobe: ubiquitous query processing on the Internet. VLDB J. **10**(1), 48–71 (2001)
8. Chandrasekaran, S., Franklin, M.: PSoup: a system for streaming queries over streaming data. VLDB J. **12**, 140–156 (2003)
9. Chaudhuri, S., Narasayya, V., Ramamurthy, R.: Estimating progress of execution for sql queries. In: Proc. of ACM SIGMOD, pp. 803–814 (2004)
10. Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Cetintemel, U., Xing, Y., Zdonik, S.: Scalable distributed stream processing. In: CIDR (2003)
11. Yang, H.C., Dasdan, A., Hsiao, R.-L., Parker, D.S. Jr.: Map-reduce-merge: simplified relational data processing on large clusters. In: SIGMOD Conference, pp. 1029–1040 (2007)
12. Culler, D.E.: Planetlab: an open, community-driven infrastructure for experimental planetary-scale services. In: USENIX Symposium on Internet Technologies and Systems (2003)
13. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: OSDI, pp. 137–150 (2004)
14. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
15. Deshpande, A., Hellerstein, J.M.: Lifting the burden of history from adaptive query processing. In: Proc. of 30th VLDB Conf., pp. 948–959 (2004)
16. Deshpande, A., Ives, Z.G., Raman, V.: Adaptive query processing. Found. Trends Databases **1**(1), 1–140 (2007)
17. Eugster, P.Th., Felber, P.A., Guerraoui, R., Kermarrec, A.-M.: The many faces of publish/subscribe. ACM Comput. Surv. **35**(2), 114–131 (2003)

18. Foster, I., Kesselman, C.: The Grid: Blueprint for a New Computing Infrastructure, 2nd edn. Morgan Kaufmann, San Mateo (2003)
19. Gounaris, A., Paton, N.W., Fernandes, A.A.A., Sakellariou, R.: Self monitoring query execution for adaptive query processing. Data Knowl. Eng. **51**(3), 325–348 (2004)
20. Gounaris, A., Paton, N.W., Sakellariou, R., Fernandes, A.A.A.: Adapting to changing resource performance in grid query processing. In: 1st Int. Workshop on Data Management in Grids, pp. 30–44. Springer, Berlin (2005)
21. Gounaris, A., Sakellariou, R., Paton, N.W., Fernandes, A.A.A.: A novel approach to resource scheduling for parallel query processing on computational grids. Distrib. Parallel Databases **19**(2–3), 87–106 (2006)
22. Graefe, G.: Encapsulation of parallelism in the volcano query processing system. In: Proc. SIGMOD, pp. 102–111 (1990)
23. Hameurlain, A., Morvan, F.: CPU and incremental memory allocation in dynamic parallelization of SQL queries. Parallel Comput. **28**(4), 525–556 (2002)
24. Hellerstein, J.M., Stonebraker, M.: Predicate migration: optimizing queries with expensive predicates. In: SIGMOD Conference, pp. 267–276 (1993)
25. Ives, Z.: Efficient query processing for data integration. PhD thesis, University of Washington (2002)
26. Ives, Z., Florescu, D., Friedman, M., Levy, A., Weld, D.: An adaptive query execution system for data integration. In: Proc. of ACM SIGMOD 1999, pp. 299–310 (1999)
27. Ives, Z., Halevy, A., Weld, D.: Adapting to source properties in processing data integration queries. In: Proc. of ACM SIGMOD, pp. 395–406 (2004)
28. Josifovski, V., Schwarz, P., Haas, L., Lin, E.: Garlic: a new flavor of federated query processing for db2. In: Proc. of ACM SIGMOD, pp. 524–532 (2002)
29. Kabra, N., DeWitt, D.: Efficient mid-query re-optimization of sub-optimal query execution plans. In: Proc. of ACM SIGMOD, pp. 106–117 (1998)
30. Li, Q., Shao, M., Markl, V., Beyer, K.S., Colby, L.S., Lohman, G.M.: Adaptively reordering joins during query execution. In: ICDE, pp. 26–35 (2007)
31. Liu, D.T., Franklin, M.J.: GridDB: a data-centric overlay for scientific grids. In: Proc. VLDB, pp. 600–611. Morgan Kaufmann, San Mateo (2004)
32. Markl, V., Raman, V., Simmen, D.E., Lohman, G.M., Pirahesh, H.: Robust query processing through progressive optimization. In: Proc. ACM SIGMOD, pp. 659–670 (2004)
33. Narayanan, S., Kurc, T.M., Saltz, J.: Database support for data-driven scientific applications in the grid. Parallel Process. Lett. **13**(2), 245–271 (2003)
34. Ng, K., Wang, Z., Muntz, R., Nittel, S.: Dynamic query re-optimization. In: Proc. of 11th SSDBM, pp. 264–273 (1999)
35. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig Latin: a not-so-foreign language for data processing. In: SIGMOD Conference, pp. 1099–1110 (2008)
36. Oram, A.: Peer-to-Peer: Harnessing the Power of Disruptive Technologies. O'Reilly (2001)
37. Ozcan, F., Nural, S., Koksal, P., Evrendilek, C., Dogac, A.: Dynamic query optimization in multidatabases. IEEE Data Eng. Bull. **20**(3), 38–45 (1997)
38. Paton, N.W., Chávez, J.B., Chen, M., Raman, V., Swart, G., Narang, I., Yellin, D.M., Fernandes, A.A.A.: Autonomic query parallelization using non-dedicated computers: an evaluation of adaptivity options. VLDB J. (2008). doi:10.1007/s00778-007-0090-x
39. Porto, F., da Silva, V.F.V., Dutra, M.L., Schulze, B.: An adaptive distributed query processing grid service. In: Proc. 1st Data Management in Grids Workshop, pp. 45–57. Springer, Berlin (2005)
40. Raman, V., Han, W., Narang, I.: Parallel querying with non-dedicated computers. In: Proc. VLDB, pp. 61–72 (2005)
41. Raman, V., Raman, B., Hellerstein, J.: Online dynamic reordering for interactive data processing. In: Proc. of 25th VLDB Conference, pp. 709–720 (1999)
42. Shah, M., Hellerstein, J., Chandrasekaran, S., Franklin, M.: Flux: an adaptive partitioning operator for continuous query systems. In: Proc. of ICDE, pp. 25–36 (2003)
43. Shah, M.A., Hellerstein, J.M., Brewer, E.A.: Highly available fault-tolerant, parallel dataflows. In: Proc. SIGMOD, pp. 827–838 (2004)
44. Smith, J., Gounaris, A., Watson, P., Paton, N.W., Fernandes, A.A.A., Sakellariou, R.: Distributed query processing on the grid. Intl. J. High Perform. Comput. Appl. **17**(4), 353–368 (2003)
45. Smith, J., Watson, P.: Fault-tolerance in distributed query processing. In: Proc. 9th IDEAS, pp. 329–338 (2005)
46. Srivastava, U., Munagala, K., Widom, J., Motwani, R.: Query optimization over web services. In: VLDB, pp. 355–366 (2006)

47. Stonebraker, M., Aoki, P.M., Litwin, W., Pfeffer, A., Sah, A., Sidell, J., Staelin, C., Mariposa, A.Yu.: A wide-area distributed database system. VLDB J. **5**(1), 48–63 (1996)
48. Tian, F., DeWitt, D.: Tuple routing strategies for distributed eddies. In: Proc. of 29th VLDB Conference, pp. 333–344 (2003)
49. Wang, X., Burns, R., Terzis, A.: Throughput-optimized, global-scale join processing in scientific federations. In: NETB'07: Proceedings of the 3rd USENIX International Workshop on Networking Meets Databases, pp. 1–6. USENIX Association, Berkeley (2007)
50. Wang, X., Burns, R.C., Terzis, A., Deshpande, A.: Network-aware join processing in global-scale database federations. In: ICDE, pp. 586–595 (2008)
51. Xing, Y., Zdonik, S., Hwang, J.-H.: Dynamic load distribution in the Borealis stream processor. In: Proc ICDE, pp. 791–802 (2005)
52. Yu, M.J., Sheu, P.C.-Y.: Adaptive join algorithms in dynamic distributed databases. Distrib. Parallel Databases **5**(1), 5–30 (1997)
53. Zhou, Y., Ooi, B.C., Tan, K.-L., Tok, W.H.: An adaptable distributed query processing architecture. Data Knowl. Eng. **53**(3), 283–309 (2005)
54. Zhu, Y., Rundensteiner, E.A., Heineman, G.T.: Dynamic plan migration for continuous queries over data streams. In: Proc. ACM SIGMOD, pp. 431–442 (2004)