

Modular Adaptive Query Processing for Service-Based Grids^a

^aA short version of this report has appeared in the Proceedings of the 3rd IEEE International Conference on Autonomic Computing (ICAC) 2006.

Anastasios Gounaris, Norman W. Paton, Rizos Sakellariou, Alvaro A.A. Fernandes
{gounaris, norm, rizo, alvaro}@cs.man.ac.uk

UoM
School of Computer Science - University of Manchester
Oxford Road, Manchester M13 9PL, UK

Jim Smith, Paul Watson
{Jim.Smith, Paul.Watson}@ncl.ac.uk

NCL
School of Computer Science, University of Newcastle-upon-Tyne, UK



CoreGRID Technical Report
Number TR-0076
March 7, 2007

Institute on Knowledge and Data Management

CoreGRID - Network of Excellence
URL: <http://www.coregrid.net>

Modular Adaptive Query Processing for Service-Based Grids

Anastasios Gounaris, Norman W. Paton, Rizos Sakellariou, Alvaro A.A. Fernandes
{gounaris, norm, rizo, alvaro}@cs.man.ac.uk
UoM

School of Computer Science - University of Manchester
Oxford Road, Manchester M13 9PL, UK

Jim Smith, Paul Watson
{Jim.Smith, Paul.Watson}@ncl.ac.uk
NCL

School of Computer Science, University of Newcastle-upon-Tyne, UK

CoreGRID TR-0076

March 7, 2007

Abstract

Distributed and heterogeneous environments present significant challenges to complex software systems, which must operate in the context of continuously changing loads, with partial or out-of-date information on resource capabilities. A distributed query processor (DQP) can be used to access and integrate data from distributed sources, as well as for combining data access with data analysis. However, in heterogeneous environments, statically constructed query plans may commit a query evaluator to following significantly suboptimal strategies. As such, there is considerable interest in using adaptive query processors (AQPs) in such settings to provide self-optimizing behaviour. However, with many possible adaptive strategies available, it is important that AQPs can be constructed in a systematic and efficient manner. This paper presents an approach to the development of AQPs in which adaptive behaviour is implemented using cooperating monitoring, assessment and response components. It is shown how this decomposition has been applied in the development of an adaptive DQP system for service-based grids, which reallocates load at query runtime, thereby supporting self-optimization. Experimental results for query processing in wide-area grids show that the approach can be effective in practice, with queries making local decisions to change their behaviour in the light of changing resource throughput.

1 Introduction

Service-based grids [7] have emerged as an attractive paradigm for developing loosely-coupled distributed applications. They provide language and platform-independent techniques for describing, discovering, invoking and orchestrating collections of distributed computational services, thus facilitating the development of complex wide-area applications. Such applications have caused novel software engineering requirements to surface. One of the most prominent is that, due to the increasing complexity of modern systems, on the one hand, and the unpredictable and volatile nature of platforms for distributed computations on the other, the systems need to be able to adapt their execution autonomously to runtime conditions.

In such a setting, autonomic computational techniques that yield all of *self-configuration*, *self-optimization*, *self-healing* and *self-protection* are potentially relevant [16]. As a result, effective autonomic solutions may have to be wide ranging in their scope, and thus ad-hoc point solutions are unlikely to yield manageable development or maintenance costs. The autonomic computing community has proposed several decompositions (e.g. [4, 20]) and notations (e.g.

This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

[5, 19]) for describing autonomic behaviours, but experience with such approaches is still being gained, and to date there is little evidence of such approaches being applied in the area of this paper, namely adaptive query processing (AQP) [8, 3].

AQP has become an active research area in recent years, characterised by the proposal of effective but narrowly specialised techniques [8, 14]. Proposals exist for addressing the problems of fluctuations in memory (e.g., [26]), bursty arrival rates of remote data (e.g., [13]), inaccurate data statistics (e.g., [15]), changing machine load (e.g., [23]), changing network load (e.g., [17]), and variable machine availability (e.g., [21]). These proposals impact on the running query plan at different granularities as some techniques rely on self-adaptive operators (e.g., [25]), whereas others may create a different plan for the remainder of the execution (e.g., [15]). However, although adaptive techniques cover such a broad range, their designs are often tailored to specific needs, narrowly specialized, and cannot be easily combined [14]. Thus AQP systems are characterised by the lack of shared abstractions and architectures, and do not conform to any common framework that provides built-in support for interactivity and interconnectivity.

This paper makes a two-fold contribution. Firstly, it proposes an architectural framework for AQP that is capable of accommodating various kinds of self-optimizing behaviour. Secondly, it shows how the framework has been implemented as an extension to the OGSA-DQP service-based distributed query processor [1].

The remainder of the paper is structured as follows. The generic framework, along with its benefits and its components, are discussed in Section 2. Section 3 reviews the OGSA-DQP system, and Section 4 describes how the framework has been deployed to develop adaptive extensions. An experimental evaluation of the resulting infrastructure using the PlanetLab [22] global grid testbed is described in Section 5. Section 6 concludes the paper.

2 The Monitoring-Assessment-Response Framework

2.1 Rationale and Description of the Framework

This section presents a generic framework for adaptive techniques in database query processing. The framework is based on the decoupling of semantically distinct phases in adaptive query execution. By definition, a query processing system is adaptive if it receives information from its environment, analyses this information and revises its behaviour dynamically in an iterative manner, i.e., if there is a feedback loop between the environment and the behaviour of the query processing system [11]. A finer-grained analysis of the feedback loop leads to the identification of three semantically distinct phases, namely:

- *monitoring*, which is concerned with the collection of feedback, both on the query execution itself and on the resources available;
- *assessment*, which deals with the evaluation of the collected feedback and the establishment of potential opportunities for improvement, or problems with the ongoing execution, and
- *response*, which plans and enacts the decisions made.

Figure 1 depicts the components of the framework and shows how these cooperate to cause an adaptation. The framework is generic in the sense of being both adaptivity environment independent, and technique independent, whilst being capable of capturing many existing AQP techniques. It makes no assumptions as to the number of adaptivity components cooperating to achieve an adaptation, or their specific interconnections. The construction of general frameworks for identifying or composing generic and reusable techniques for monitoring, assessment or response has the following key advantages.

1. It allows component reuse and enables the assembling of different combinations, thus covering a wider spectrum of capabilities. For example, a particular approach to monitoring can be used with different forms of assessment and response, or different categories of response can be made in the light of a single approach to monitoring and assessment.
2. The adoption of a generic framework makes the design activity more systematic. Developers can focus on the internal logic of the components without worrying about how the others are implemented internally, although the developer of the internal logic does have to think about the local and remote interfaces.
3. By focusing on the interfaces of cohesive, decoupled modules, the design of the framework conforms to service-oriented architectures, which are suitable for advanced, wide-area applications.

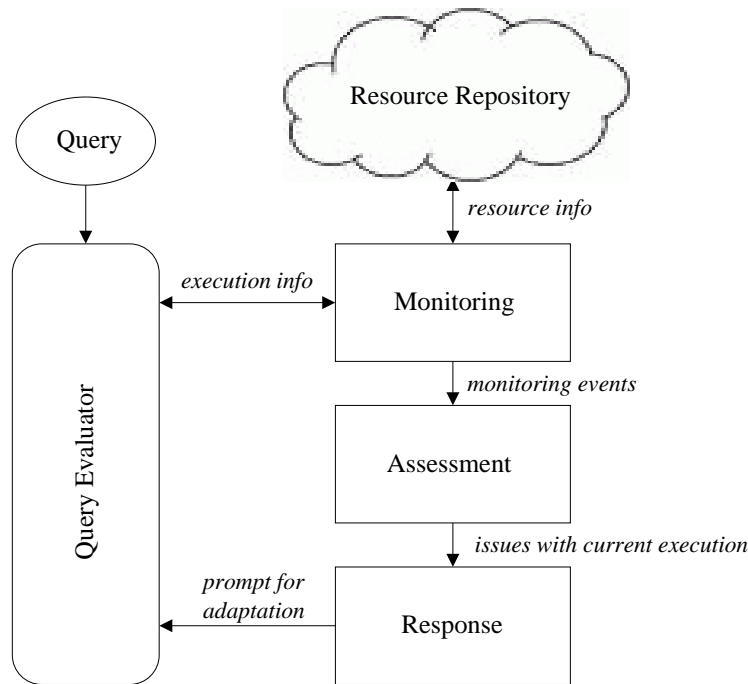


Figure 1: The monitoring, assessment and response phases of AQP.

In addition to the concrete instantiation of this framework as an extension to OGSA-DQP described in Section 4, significant progress has been made towards the development of generic monitoring components, as described in [9]. This work introduces the notion of *self-monitoring* operators as an assessment and response-independent approach to collecting feedback about query execution on the fly. It is important to note that the generality of the approach has not compromised performance, as the overheads associated are reasonable, or even negligible in many cases [9].

2.2 The Components of the Framework

An instantiation of the framework for an AQP technique requires at least one of each different kind of framework component to cover all the adaptivity phases. To this end, the components support a publish/subscribe interface [6], to provide and ask for services. The behaviour of the framework components, i.e., their functionality and interactions, is as follows:

Monitoring: a monitoring component (MC) acts as a source of notifications on the dynamic behaviour of distributed resources. Other adaptivity components (including monitoring ones) can subscribe to it. Thus the MC interface supports subscription requests. The MC interacts with other adaptivity components to deliver the notifications requested. Such notifications are in a commonly agreed form that hides how monitoring is carried out. Also, the MC performs basic integration and filtering of 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 has changed by more than 10%).

Assessment: The role of the assessment component (AC) is to establish whether there exist opportunities for improvement of plan performance (or any other QoS criterion), and whether there is a problem with the current execution that needs to be addressed. In either case, the AC sends a relevant notification to the appropriate response component. The AC performs its task by correlating and analysing notifications from one or more monitoring components. An AC interacts with other services for two purposes: to subscribe to monitoring components, and to send notifications about problems and opportunities to response components.

Response: the response component (RC) is responsible for: (i) identifying valid responses to the issues identified by the assessment component (e.g., by exploring a search space, or by using predefined lists for each identified

issue); (ii) evaluating the expected benefits and costs for each valid response (e.g., by using cost functions); (iii) selecting the most efficient one; and (iv) interacting with the evaluation engine to implement its decisions. The RC is able to subscribe to other components (e.g., ACs) and to receive notifications.

Another key motivation behind the framework is that there is a finite set of kinds of things to monitor (e.g., operator throughput, operator selectivity, memory available), forms of assessment to conduct (e.g. load imbalance, insufficient memory), and forms of response (e.g. change query plan, change allocation of operators to nodes, change plan parameters) with which to realise adaptations. The existence of such sets enables the creation of small “vocabularies” for communication between the response adaptivity phase and the query engine (Figure 1). It is not the aim of this work to analyze the phases in depth and classify prior work on adaptive query processing. The purpose of this discussion is rather to demonstrate that being able to define small “vocabularies” for the inter-phase communication, the framework becomes of practical significance because well defined collections of messages can be defined that reflect the outcomes, and characterise the interactions, of each of the components.

How does this proposal relate to the IBM autonomic computing toolkit, as discussed in [12, 20]? Although conducted independently the approaches have several similarities. In [12], the *Autonomic Manager (AM)* plays a key role, and consists of four parts: monitoring, analysis, planing and execution. In our framework we have three parts: monitoring, assessment, and response, which encapsulates both planning and execution. We do not have a separate execution component, as changes to query plans are implemented using potentially diverse collections of operations on query evaluation services. A further difference is that the AM plays a centralising, coordinating role, whereas there is no such need in our proposal, where remote adaptivity components can subscribe to each other and make local decisions. Also, in AM, the components share knowledge, whereas, in this paper, they exchange and acquire the information needed on demand. Our feeling is that our framework is more dynamic, in the sense that, through a subscription scheme, it enables the creation of federations of adaptivity components, which can produce effects in various scopes (i.e., local adaptations, adaptations affecting the nodes of a particular LAN, global adaptations, etc.). In addition, through the notification publishing scheme, we can leverage existing technology, and benefit from space, time and synchronisation decoupling during component interactions [6].

The above functional decompositions seem largely orthogonal to that of the Accord framework [18], in which components expose functional, control and operational ports, and inter-component behaviour is expressed using rules. The diversity of such proposals suggests that further work is required to identify the most common recurring themes and effective abstractions.

3 Service-based Distributed Query Processing

The framework from Section 2 has been instantiated to provide an adaptive extension to the OGSA-DQP service-based distributed query processor¹. This section describes OGSA-DQP, and Section 4 describes how it has been extended to exhibit self-optimizing behaviour. 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 services, users and developers can integrate data from multiple databases.

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(contologyid,cproteinid)* and *Sequence(sproteinid,ssequence)*, 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 the Grid, and call *BLAST* on the results produced. In OGSA-DQP, this task is conveniently described by the query in Figure 2. The figure also shows the query execution plan (produced by the query optimizer within the GDQS without any human intervention) that performs this task. The nodes of the plan are query operators: scans, projects, joins, exchanges (for data communication) and a call to a Web Service, in this example. The operators may be executed on different machines, in parallel. The

¹Freely available for download in open source form at www.ogsadai.org/dqp.

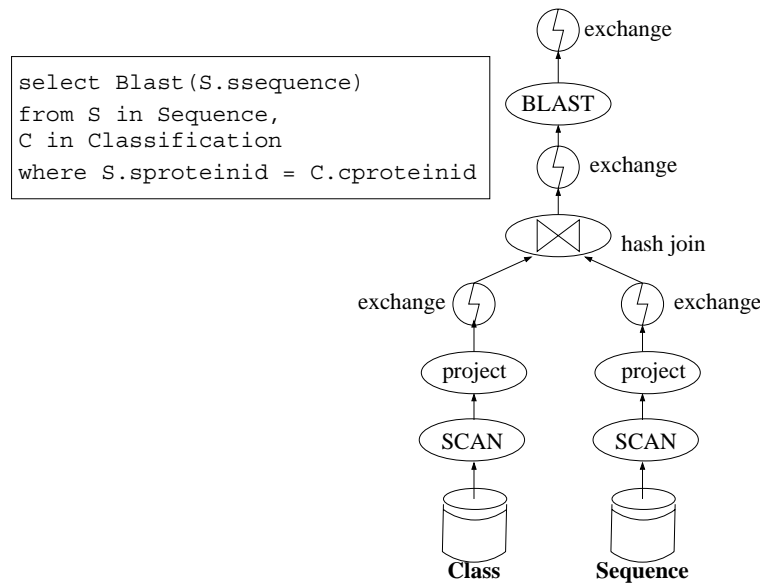


Figure 2: Example query plan.

scan operators are specifically designed to access Grid-enabled databases, e.g., in the case of OGSA-DQP, they rely on OGSA-DAI Grid Data Services (GDSs).

Suppose that the optimizer decides 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 Figure 3. These fragments are executed in the context of GQESs.

As such, the evaluation of the query plan is achieved through orchestration of multiple GQESs, GDSs and WSs, 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 (static service orchestration). In many areas, composing and orchestrating services on a per-task basis is adequately dynamic, but this is not always the case for database query processing. Decisions such as the order of operators, and the selection of machines to evaluate each query operator, are taken by the optimizer and are based on data properties (such as number of tuples in databases and predicted sizes of intermediate results) and on machine properties (such as the memory size of each machine available to participate in the query execution). When the databases accessed and the services contacted are autonomous, distributed Grid resources, such critical information may be unavailable, incomplete, inaccurate, out-of-date, or subject to changes during evaluation, in which case the execution plan is likely to be suboptimal. To tackle this, query processing needs to be adaptive, as described in the following section.

4 Service-based Adaptive Query Processing

4.1 Adaptive, Self-optimizing Query Services

As presented in Section 3, OGSA-DQP provides two types of Grid Services to perform static query processing, GDQS and GQES. To endow the existing OGSA-DQP software with self-optimizing behaviour, extensions have been made that fall into three categories: (i) extensions to GQESs, so that they implement monitoring, assessment and response components; (ii) extensions to the evaluation engine, within GQESs to provide low-level monitoring information and to support changes to query plans at runtime; and (iii) extensions to the metadata sent to evaluators by GDQSs. These extensions are described below.

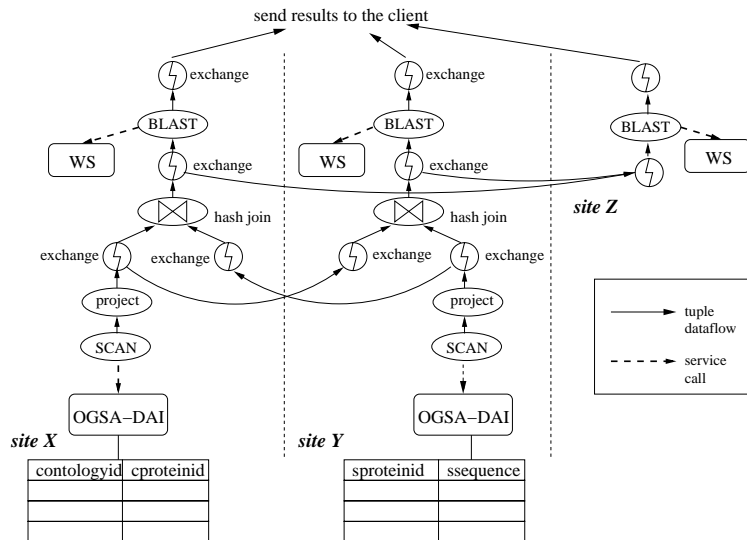


Figure 3: Query plan distribution.

4.1.1 Extensions to GQESs

Adaptive GQESs (AGQESs) encapsulate the framework components for monitoring, assessment and response. This means that, as there is an AGQES at each site participating in query evaluation, there can be multiple instances of adaptivity components (e.g., multiple monitoring components) that interact at a service level to perform an adaptation. In particular, each AGQES comprises four components, as illustrated in Figure 5: one for implementing the query operators and thus forming the query engine (which is the only component in static GQESs), and three for adaptivity, in line with the adaptivity framework.

Monitoring is based on self-monitoring operators, as reported in [9]. As such, the query engine is capable of monitoring its own behaviour, and of producing raw, low-level monitoring information (such as running aggregates for the number of tuples produced and the time spent by an operator). The *MonitoringEventDetector* component instantiates the monitoring component of the framework, i.e., it integrates the raw events produced by each query engine and detects significant changes (e.g., in the average cost of an operator or in the number of machines available). The *Diagnoser* is notified of such detected changes and establishes whether there is an issue with the current execution (e.g., workload imbalance), thereby completing the assessment phase. The *Responder* is notified of any such issues and chooses how to react. Its decisions may affect not only the local query engine, but any query engine participating in the evaluation. The query engine is extended to be capable of handling such response messages.

Although the internal architecture of an AGQES has become more complex, the interface remains simple, with components exchanging messages using a straightforward publish/subscribe interface. The adaptivity notifications and subscription requests are transmitted across AGQESs as XML documents over SOAP/HTTP. There is as yet no widely agreed standard for notifications in Web Services, but our approach essentially implements a subset of direct notification in WS-notification².

Adaptivity Notifications The set of types of notifications exchanged between components are described in the AGQES interface, as happens with any kind of inter-service communication. The notification schema is shown in Figure 4, and consists of two parts: generic fields, which are common to all notifications; and specific fields, which may differ for different types of notifications.

The generic fields include: (i) *description*, which specifies the type of the notification (e.g., whether it is a subscription); (ii) *destinationId*, which specifies the recipient component in a possibly remote AGQES; (iii) *originatorService*, which specifies the handle of the AGQES that sends the notification; (iv) *messageId*, which allows unique message identification; and (v) *correlatedMessageId*, which specifies any other notifications that may need to be taken into account for the correct processing of the current message.

²<http://www-106.ibm.com/developerworks/library/specification/ws-notification/>

```

<complexType name="Notification">
  <sequence>
    <!-- generic fields -->
    <element name="description"/>
    <element name="destinationId"/>
    <element name="originatorService"/>
    <element name="messageId"/>
    <element name="correlatedMessageId"/>
    <!-- specific fields -->
    <choice>
      <element name="MonitoringInformation">
        <complexType>...</complexType>
      </element>
      <element name="PerformanceChange">
        <complexType>...</complexType>
      </element>
      <element name="ImbalancedNode">
        <complexType>...</complexType>
      </element>
      ...
    </choice>
  </sequence>
</complexType>

```

Figure 4: Notification schema definition.

The specific fields are defined separately for each type of notification, and thus are tailored to the particular kinds of adaptations that the system supports. Examples include fields to propagate monitoring information from the query engine, to denote that the performance of a physical machine has changed, and to denote that a physical operator is partitioned across machines in an imbalanced way.

Implementation of Adaptivity Components Each adaptivity component exposes to other parts of the system the following operations:

- *getNotification(Notification)*, for the receipt and analysis of input messages;
- *putNotification(Notification)*, for sending messages conforming to the schema in Figure 4.

At runtime, in each component, a thread is running continuously to retrieve the notifications that have arrived in the queue. The analysis of a notification is specific to the notification type (e.g., notifications for denoting change in the cost of an operator, workload imbalance, request for plan modification, etc.) and may involve sending a new notification to (some of the) subscribed components. As such, the adaptivity components differ in the types of notification they can analyse, and how the analysis is conducted, i.e., how the *getNotification(Notification)* function is implemented.

4.1.2 Extensions to the Evaluation Engine

The extensions to the evaluation engine are at two levels: (i) the execution engine has become capable of receiving messages that influence the current execution; and (ii) the exchange operator has been extended both to produce monitoring information that drives the adaptivity cycle, and to modify its behavioural parameters dynamically. The first type of extension has been realised through the implementation of a *getNotification()* function within the query engine, similar to the *getNotification()* functions in the adaptivity components. This entails that the evaluation engine receives adaptivity-related messages and provides the mechanisms for their interpretation.

In the adaptive OGSA-DQP system, exchanges implement the self-monitoring approach introduced in [9]. Moreover, they are capable of propagating such monitoring information as notifications, which are subsequently integrated in the *MonitoringEventDetector* component.

4.1.3 Extensions Relating to GDQS Metadata

In non-adaptive OGSA-DQP, a GDQS contacts the GQESs it has created once, in order to send the query plan fragment that is to be evaluated by the evaluation engine within this GQES. This information is required to initialise the evaluation engine, and is sent in an XML document. In adaptive OGSA-DQP, as AGQESs contain not only an evaluation engine but also the adaptivity components, more information needs to be included in this XML document, to

initialise these components properly. Regarding the description of the query plan, the only change is that exchanges that form the local root of plan fragments are annotated with the estimated cardinality of the result set.

Also, to enable AGQESs to adapt autonomously, without the intervention of the GDQS, part of the metadata that is stored in the latter needs to be transferred to the former. An AGQES needs to be aware of the existence of other AGQESs, even if there is no direct data communication between them, in order to allow its adaptivity components to subscribe to remote counterparts. In addition, in order to take some adaptivity decisions, knowledge of the complete query plan is required, e.g., which subplans are clones of each other, and which subplans send data to other subplans. As such, the metadata sent by GDQSs to AGQESs can be divided in two main categories: metadata about the AGQESs participating in the execution, or available to participate; and metadata about the global query plan.

4.2 Adaptive Workload Balancing

A basic technique for optimising performance is to balance execution across AGQESs, in such a way that all the evaluators that process the same plan fragment finish at approximately the same time; this is essentially load balancing in the context of partitioned parallelism. A balanced load can be achieved if the amount of workload allocated to each AGQES is proportional to its processing speed.

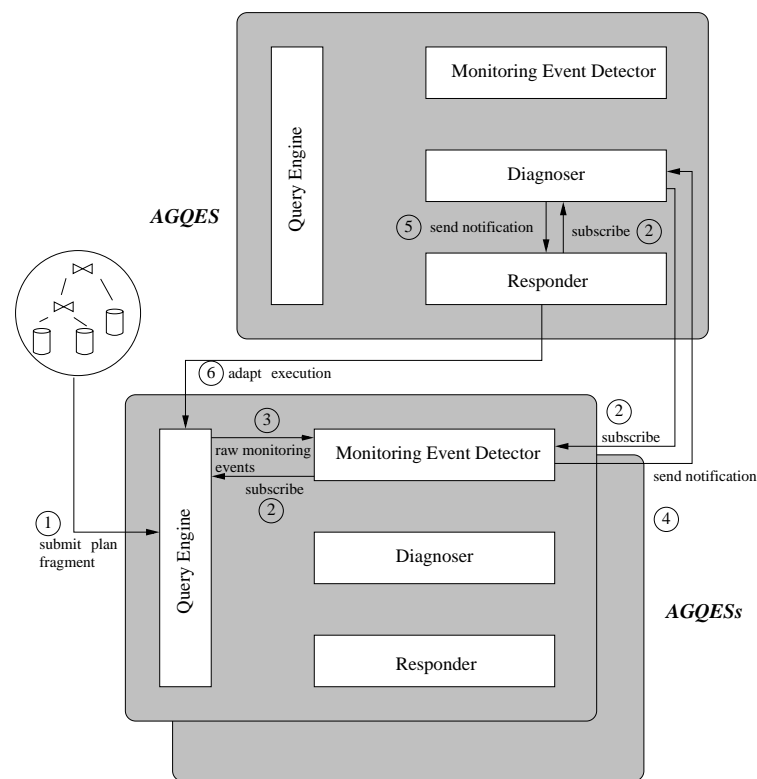


Figure 5: The architecture and the interactions for runtime workload balancing.

A service composition that is suitable for this purpose is as follows: a *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, globally accessible *Diagnoser* that subscribes to the *MonitoringEventDetectors*, and a single *Responder* that subscribes to the *Diagnoser*.

Figure 5 illustrates the interactions between the components of the framework in this scenario; the numbers on the message exchanges can be understood as follows:

1. *Submission of query plan fragment*: The GDQS allocates the fragments of the query plan to AGQESs. As an example, in Figure 3 the join is evaluated in parallel on two machines, and thus would be represented by two fragments submitted to AGQESs on the two machines.

```

<complexType name="Notification"> <sequence>
  <element name="description"
    value="DETECTED_EVENT"/>
  <element name="destinationId"
    value="DIAGNOSER"/>
  <!-- other generic fields --> ....
  <!-- specific fields -->
  <element name="rootOperatorID" />
  <element name="nonCommunicationCostMsec" />
  <element name="waitingCostMsec" />
  <element name="communicationCostMsec" />
  <element name="receiverEvalID" />
  <element name="numProducedTuples"/>
  ...
</sequence> </complexType>

<complexType name="Notification">
  <sequence>
    <element name="description"
      value="DIAGNOSED_ISSUE"/>
    <element name="destinationId"
      value="RESPONDER"/>
    <!-- other generic fields --> ...
    <!-- specific fields -->
    <element name="ImbalancedNode"
      maxOccurs="unbounded">
      <complexType>
        <element name="subplanRootOpId"/>
        <element name="subplanRootOpEvaluator"/>
        <element name="performanceIndicator" />
        <element name="currentProportion" />
        <element name="proposedProportion" />
      </complexType>
    </element>
  </sequence>
</complexType>

<complexType name="Notification">
  <sequence>
    <element name="description"
      value="RESPONSE_IMBALANCE"/>
    <element name="destinationId"
      value="AGQES_RESPONDER"/>
    <!-- other generic fields --> ...
    <!-- specific fields -->
    <element name="ExchangeReconfiguration">
      <complexType>
        <element name="modifiedOpId" />
        <element name="consumerReference" .../>
        <element name="newProportion" ... />
      </complexType>
    </element>
  </sequence>
</complexType>

```

Figure 6: Schema of notifications sent by the *MonitoringEventDetector* (top), *Diagnoser* (middle) and the *Responder* components (bottom).

2. *Subscription*: Each *Responder* subscribes to the *Diagnosers* that may identify issues to which it can respond – in this case it is looking for changes in the throughput of query fragments; each *Diagnoser* subscribes to the *Monitoring Event Detectors* that are monitoring the query fragments across which it seeks to identify load imbalance – the diagnoser determines if there is indeed load imbalance; and each *Monitoring Event Detector* subscribes to the *Query Engines* that are evaluating fragments from which it seeks to infer significant patterns in the evaluation. Steps 1 and 2 take place before the query starts to execute.
3. *Monitoring of query evaluation*: Raw monitoring events from the query engine are observed by the *Monitoring Event Detector*, which performs local filtering to avoid overwhelming the *Diagnoser* with inconsequential progress reports.
4. *Notification to Diagnoser*: The *Monitoring Event Detector* identifies information that it considers is worth passing on to the *Diagnoser*. In the example, it is considered significant if the throughput of a node has increased or reduced above a threshold amount (e.g. 20%). The schema of the notification message is provided in Figure

- 6.
5. *Notification to Responder*: The *Diagnoser* identifies an issue to which a response may be required. In the example, it is determined whether the current distribution of tuples to the join nodes is out of line with their throughput, and thus that load rebalancing may be required.
6. *Adapt execution*: The *Responder* determines that it is worthwhile to carry out an adaptation, and sends messages to AGQESs that revise the workload allocation (a *setDataReDistribution* message is implemented by each AGQES for this purpose). The details of the dynamic workload balancing technique are beyond the scope of this paper; the approach implemented in OGSA-DQP for load balancing is described in [10]. In essence, the state that needs relocated to enable the continued evaluation of the join algorithm (the hash table in the case of the hash join) is sent to suitable nodes from upstream caches, and the distribution policy adopted by upstream exchange nodes is revised to reflect the updated data distribution.

The above description involved a single instantiation of the framework. However, an additional instantiation has been implemented. Rather than balancing load across a collection of parallel partitions, bottlenecks are detected in the pipelined query evaluation, which are addressed through the addition of parallel nodes at the rate-limiting part of the pipeline. This approach uses the same *Monitoring Event Detector* as in load balancing, but a different *Diagnoser* is deployed, which monitors a larger number of plan fragments. In general, many adaptivity policies that correspond to different patterns of dynamic service orchestration are enabled by the architecture proposed. For each policy, the developer needs to define the schemas of the notifications that will be published, and to ensure that the adaptivity components can process these notifications.

5 Experimental Results

This section describes some performance results obtained for adaptive load balancing implemented using the infrastructure described in Section 4.2. The results have been obtained using PlanetLab [22], a resource currently comprising about 650 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. As such, the experiments described here complement earlier results on adaptive load balancing in controlled settings described in [10].

5.1 PlanetLab Configuration

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. On login 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* don’t 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 Section 4.2.

The experiments used a slice containing 55 machines. 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 3GHz; in memory between 512MB and 2GB; and in cache size between 256 and 2048KB. However, apart from the fact that many of the machines have some balancing of 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.

5.2 Experiment Setup

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, 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 example query is measured:

```
select analysis1(p.sequence) ,
       analysis2(p.sequence)
from   protein-sequence p;
```

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 Figure 7(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*, 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 Figure 7(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.

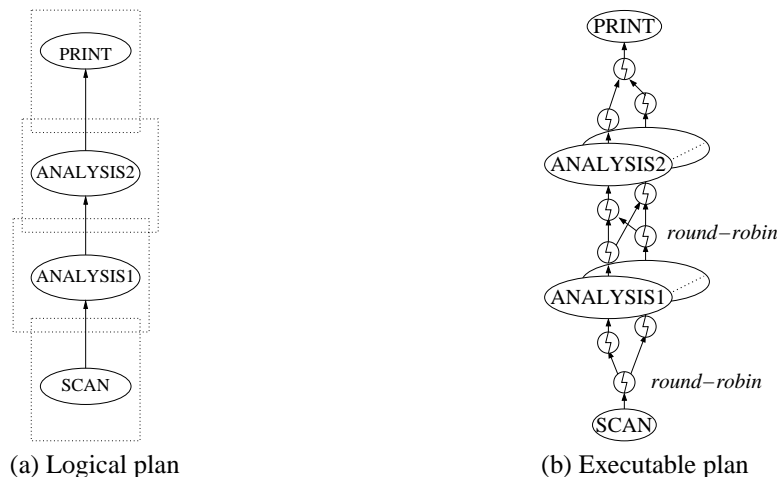


Figure 7: Planning a query containing two operation calls.

5.3 Results

Figure 8 shows the impact of runtime adaptivity where increasing numbers of machines are used to parallelise the analysis operations (for example, if there are 4 compute machines, there are 2 instances of each of *analysis1* and

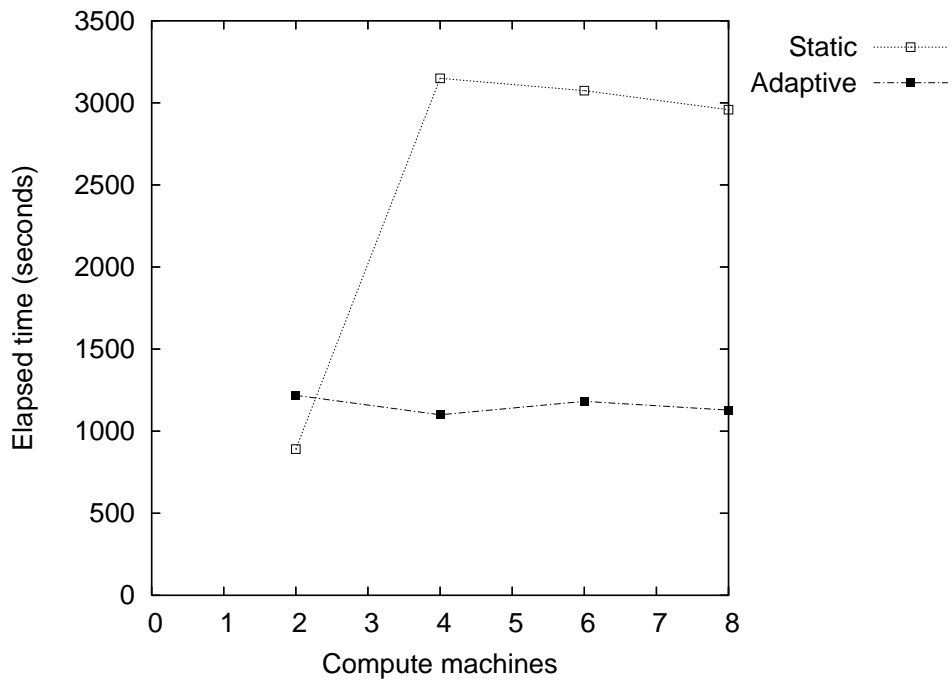


Figure 8: Results for low-cost operation calls.

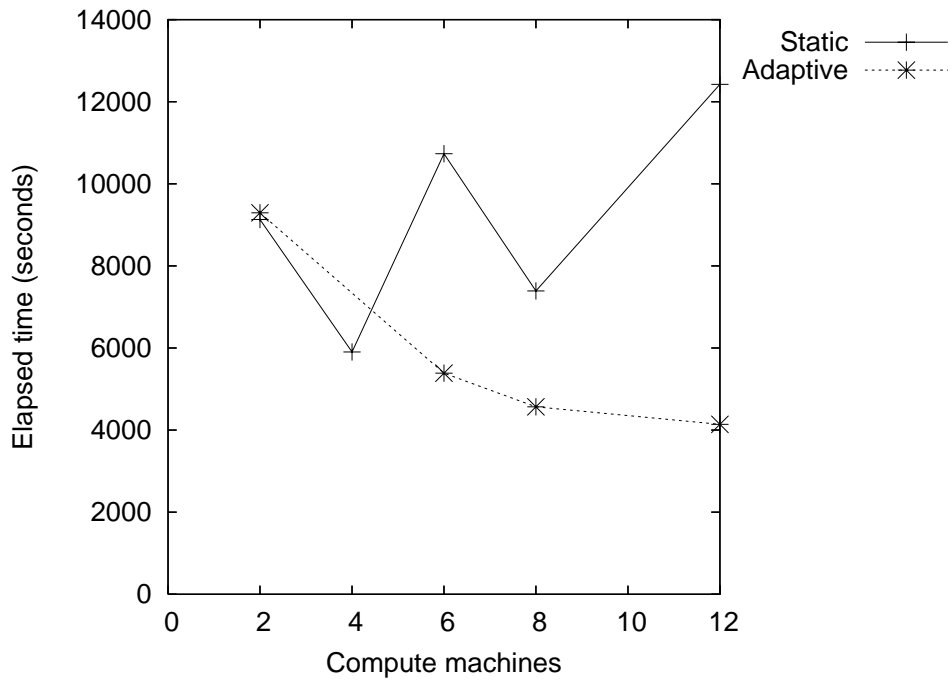


Figure 9: Results for costly operation calls.

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 9 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 parallelism³. 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.

6 Conclusions

Accessing, integrating and analysing data from multiple resources using query technologies in a service-oriented environment shows considerable promise. However, due to the volatility and the unpredictability of the setting, query processing stands to benefit from autonomic behaviour for many tasks, such as refining cost models [24] and changing query plans at runtime, as in AQP. AQP, however, suffers from a lack of generic models and behavioural abstractions. The potential of generic techniques for adaptive systems has been identified in various areas (e.g., for describing policies in mobile systems [5] and for network management [19]), but overall research into the systematic development of adaptive systems must be considered to be in its infancy. In AQP, several authors have investigated the provision of mechanisms for describing adaptive behaviour, such as the use of event condition action rules to describe adaptive policies (e.g. [21, 13]); such investigations are complementary to the work presented in this paper, which focuses more on how adaptive behaviour is decomposed than on how specific decisions are made.

In summary, the contributions of the paper are as follows: (i) it presents a generic framework for developing and describing AQP systems; (ii) it illustrates a service-based adaptive query evaluation system as an instantiation of (i); (iii) it provides examples of dynamic orchestration enabled by (ii); and (iv) it demonstrates through an experimental evaluation in a global grid that the approach can improve on the performance of static query scheduling. Complementary evaluation results in more controlled settings are described in [10], and are promising, as they show that the approach described is light weight, incurs a low overhead, and can improve query response times by significant margins.

Acknowledgement: This work is supported by the EPSRC, through Grant GR/R51797/01, and through the UK e-Science Programme DAIT Project. We are pleased to acknowledge their support.

References

- [1] M. N. Alpdemir, A. Mukherjee, N. W. Paton, P. Watson, A. A. A. Fernandes, A. Gounaris, and J. Smith. Service-based distributed querying on the grid. In *Proc. of ICSOC*, pages 467–482, 2003.
- [2] M. Antonioletti, M. P. Atkinson, R. Baxter, A. Borley, N. P. Chue Hong, B. Collins, N. Hardman, A. C. Hume, A. Knox, M. Jackson, A. Krause, S. Laws, J. Magowan, N. W. Paton, D. Pearson, T. Sugden, P. Watson, and M. Westhead. The design and implementation of Grid database services in OGSA-DAI. *Concurrency - Practice and Experience*, 17(2-4):357–376, 2005.

³Where values are missing for one or other of the approaches at some level of parallelism, this reflects some form of node failure or timeout; the current PlanetLab infrastructure has proved quite unstable for long running tasks.

- [3] S. Babu and P. Bizarro. Adaptive query processing in the looking glass. In *CIDR*, pages 238–249, 2005.
- [4] J.P. Bigus, D.A. Schlosnagle, J.R. Pilgrim, W.N. Mills III, and Y. Diao. ABLE: A toolkit for building multiagent autonomous systems. *IBM Systems Journal*, 41(3):350–371, 2002.
- [5] C. Efstratiou, A. Friday, N. Davies, and K. Cheverst. Utilising the event calculus for policy driven adaptation on mobile systems. In *3rd Intl. Wshp on Policies for Distributed Systems and Networks (POLICY)*, pages 13–24. IEEE Press, 2002.
- [6] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.
- [7] Ian Foster, Carl Kesselman, J. M. Nick, and Steve Tuecke. Grid Services for Distributed System Integration. *IEEE Computer*, 35(6):37–46, 2002.
- [8] A. Gounaris, N. W. Paton, A. A. A. Fernandes, and R. Sakellariou. Adaptive query processing: A survey. In *19th British National Conference on Databases*, pages 11–25, 2002.
- [9] A. Gounaris, N. W. Paton, A. A. A. Fernandes, and Rizos Sakellariou. Self monitoring query execution for adaptive query processing. *Data and Knowledge Engineering*, 51(3):325–348, 2004.
- [10] A. Gounaris, N. W. Paton, R. Sakellariou, and A. A. A. Fernandes. Adapting to changing resource performance in grid query processing. In *1st Int. Workshop on Data Management in Grids*, pages 30–44. Springer-Verlag, 2005.
- [11] J. Hellerstein, M. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, 2000.
- [12] An Architectural Blueprint for Autonomic Computing. <http://www-03.ibm.com/autonomic/pdfs/ACJune 2005>.
- [13] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld. An adaptive query execution system for data integration. In *Proc. of ACM SIGMOD*, pages 299–310, 1999.
- [14] Z. Ives, A. Halevy, and D. Weld. Adapting to source properties in processing data integration queries. In *ACM SIGMOD*, pages 395–406, 2004.
- [15] N. Kabra and D. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proc. of ACM SIGMOD 1998*, pages 106–117, 1998.
- [16] J. Kephart and D. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.
- [17] V. Kumar, B.F. Cooper, and K. Schwan. Distributed Stream Management using Utility-Driven Self-Adaptive Middleware. In *Proc. 2nd Int. Conf. on Autonomic Computing*, pages 3–14. IEEE Press, 2005.
- [18] H. Liu and M. Parashar. A component based programming framework for autonomic applications. In *IEEE Internation Conference on Autonomic Computing*, pages 10–17.
- [19] L. Lymberopoulos, E. Lupu, and M. Sloman. An adaptive policy based management framework for differentiated services networks. In *3rd Intl. Wshp on Policies for Distributed Systems and Networks (POLICY 2002)*, pages 147–158. IEEE Computer Society, 2002.
- [20] B. Melcher and B. Mitchell. Towards and Autonomic Framework: Self-Configuring Network Services and Developing Autonomic Applications. *Intel Technology Journal*, 8(4):279–290, 2004.
- [21] K. Ng, Z. Wang, R. Muntz, and S. Nittel. Dynamic query re-optimization. In *Proc. of 11th SSDBM Conference*, pages 264–273. IEEE Computer Society, 1999.
- [22] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the internet. In *Workshop on Hot Topics in Networks (HotNets)*, pages 59–64. ACM Press, 2002.
- [23] M.I Shah, J. Hellerstein, S. Chandrasekaran, and M. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proceedings of the IEEE ICDE*, 2003.

- [24] M. Stillger, G. Lohman, V. Markl, and M. Kandil. LEO - DB2's learning optimizer. In *VLDB 2001*, pages 19–28. Morgan Kaufmann, 2001.
- [25] T. Urhan and M. Franklin. XJoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, 2000.
- [26] W. Zhang and P. Larson. Dynamic memory adjustment for external mergesort. *Proc. of 23rd VLDB Conference*, pages 376–385, 1997.