RESOURCE AWARE QUERY PROCESSING ON THE GRID

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2005

By
Anastasios Gounaris
School of Computer Science

Contents

List of Tables

List of Figures

Abstract

The Grid provides facilities that support the coordinated use of diverse, autonomous resources, and consequently, provides new opportunities for wide-area computations. Such computations can be effectively expressed and evaluated as database queries, when they involve integration of data from remote databases and calls to analysis tools. In this way, they can benefit from implicit, transparent to the user parallelisation across multiple computational resources that the Grid makes available, and from well-established optimisation techniques.

However, Grid resources, as well as being heterogeneous, may also exhibit unpredictable, volatile behaviour. Thus, query processing on the Grid needs (i) to be capable of taking account of the resource heterogeneity; and (ii) to be adaptive, in order to cope with evolving resource characteristics, such as machine load and availability. The work presented in this thesis proposes techniques that address these two challenges, and significantly improve the performance of query processing on the Grid, going beyond the current state-of-the-art in wide-area distributed query processing. This is achieved by efficiently selecting resources before query execution according to their characteristics known at this point, and by adapting the query execution according to the runtime behaviour and availability of resources.

Adaptive query processing (AQP) has been investigated in a more generic way in this thesis, and this has led to the development of an architectural framework that is capable of accommodating many AQP systems, both for Grids and for more traditional environments. Moreover, a generic mechanism for extracting monitoring information from the query execution to support multiple AQP techniques is part of the thesis contributions. Thus, the work presented should be of value to developers and researchers of both Grid query processors and more generic AQP systems, who will be able to incorporate part, or all, of the proposals in their work.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

Acknowledgements

Looking back a few years, it is amazing to realise how and to which extent others have helped, supported and influenced my work. I have been blessed to work with friendly and competitive groups, and collaborate and associate with academics and researchers of high quality.

I am grateful to my supervisors, Rizos Sakellariou and Norman W. Paton, and to my advisor, Alvaro A. A. Fernandes, for their continuous guidance, encouragement and thoughtful advice. Special thanks are also due to Nedim Alpdemir and to our partners in the University of Newcastle, Paul Watson, Jim Smith and Arijit Mukherjee. Without their assistance, the work embodied in this thesis would not have been possible. Sandra Sampaio also offered invaluable help at the first stages of this work.

Thanks also to all the members of the Information Management Group, in particular Bonn, Cornelia, Daniele, Desmond, Henan, Marcelo, Mike, Nedim, Paul, Phil, Serafeim, Veruska and Yeliz with whom I have shared office and many moments, both difficult and happy, during the years of my PhD.

Finally, this work was supported by the Distributed Information Management Programme of the Engineering and Physical Science Research Council, through Grant GR/R51797/01.

Chapter 1

Introduction

Database systems typically provide an interface for submission of queries written in a declarative language (e.g., SQL, OQL, XQuery etc.) to allow users to define what result is to be computed, without having to specify how this is to be evaluated. The responsibility of finding an execution plan, and executing it, rests with the database system. This task is called *query processing* and has been the topic of rather intensive research since the 1970s (e.g., [SWKH76, SAC$^+$79]). However, most of this research has concentrated on the cases in which the computational resources available are predetermined, both when these are co-located (*centralised query processing*) and when they are geographically dispersed (*distributed query processing - or DQP*).

Unfortunately, query processing on the *Grid* does not fall into this category, as the Grid is a volatile, unpredictable environment in itself [FK03], and thus the resources it provides are subject to frequent, unforeseeable changes. The Grid opens new directions for query processing in wide-area environments, as it provides solutions for problems such as security, authorisation, authentication, resource discovery, and so on. However, the volatility, multiple ownership and heterogeneity of the environment necessitate the development of novel query processing methods.

To this end, the aim of the thesis is to propose and evaluate techniques that significantly improve the performance and robustness of query processing on the Grid. The core concept is to take into consideration the potentially evolving characteristics and behaviour of the (arbitrary) computational resources that are available.

The remainder of this introduction is structured as follows. Section 1.1 discusses in brief the Grid, which is the setting of the research of the thesis. Section 1.2 introduces the main issues related to grid query processing, discussing its importance and highlighting the aspects that have motivated the research described in this thesis. Section

1.3 summarises the technological advances that provide the context in which the investigation of query processing on the Grid is carried out. Next, Section 1.4 provides the hypothesis that this thesis investigates, and an overview of the contributions it makes. The outline of the dissertation is in Section 1.5.

1.1 Grid: A new setting for distributed computations

The Grid aims to create a new era in the way computations that involve the combination and co-operation of diverse, autonomous resources are carried out [FK03], just as the emergence of the Internet radically altered the manner in which computers at distinct locations are connected to each other. It enables the dynamic collaboration and the coordinated sharing of resources between different administrative domains, referred to as *virtual organisations* [FK03]. Through the development of standards [GGF], protocols [FKT01], and toolkits (e.g., [GLO]), the Grid can support resource discovery, allocation, secure authorisation and authentication, monitoring, etc. Thus, many of the challenges that are inherent in wide-area computations are now addressed by application-independent Grid middleware. Grid applications can use the Grid to select and schedule the computational and data resources that are necessary for them to execute. Such resources may include CPU cycles, memory storage devices, commodity machines, networks, database systems, data, programs, and so on. As such, the Grid constitutes a new environment for inter-organisation distributed applications, naturally oriented towards non-trivial, computation- and data-intensive applications.

Computations on the Grid can be deemed as an advanced form of computations over distributed, heterogeneous and autonomous resources. The main difference is that the Grid provides improved support for resource discovery, access, authorisation, authentication and monitoring through standard, open, general-purpose protocols and interfaces, which are not supported by current Internet and Web infrastructures. Thus, it can serve as an efficient middleware that provides the above functionalities, and on top of which applications can be developed. Also, Grid computing, in contrast with traditional distributed computing, focuses on large-scale resource sharing (i.e. not primarily file exchange as on the web, but rather direct access to computers, software, data and other resources) for innovative applications.

The main topic of this thesis is the exploitation and integration of two types of Grid resources, which are both required for query processing: (i) database management systems (DBMSs) that store data in a structured format and can be accessed through the

Grid (called *Grid databases* or *Grid-enabled databases* in the remainder of the thesis); and (ii) machines that can be accessed through the Grid and can host computations, which manipulate the data retrieved from the Grid-enabled DBMSs.

1.2 On Querying Grid-Enabled Databases

Even within the boundaries of an enterprise, the current trend in data management is towards decentralised systems and architectures, based on peers that (i) are administered autonomously, and (ii) can both consume and provide data and computation [Ive02]. This paradigm is more flexible, scalable and dynamic than traditional centralised approaches. Inevitably, these advantages come at the expense of the lack of a single point of access to all these data. The Grid offers standard mechanisms for accessing such nodes, and query processing technologies can be used to combine their data.

One of the most relevant cases to query processing on the Grid is when multiple organisations agree to share their data to perform a common task. For example, a Grid database application may involve autonomous bioinformatics labs across the world sharing their experimental results stored in organisational databases, and their computational resources in order to identify molecules which might inhibit the growth of various types of cancer cells. As discussed in more detail in Section 2.3, query processing on the Grid is a promising solution for combining and analysing data from different stores, and can be used in any such scenario. This is achieved by using (i) mature and well-established techniques (e.g., parallel query processing), along with novel ones (e.g., adaptive query processing), from the area of database research, to process data, and (ii) the Grid infrastructure to discover and access remote resources in a way that meets the application requirements.

In general, query processing on the Grid is similar to query processing for data integration (i.e., answering data integration queries [Ive02]) in the sense that in both domains (i) the queries are submitted over a global schema, and (ii) the data is fetched from the sources through wrappers. Note that the methodology for the construction of the global schema is out of the scope of the current thesis, as efficient solutions for this problem have been developed (e.g., [MP03]). The main difference between the two domains is that the resource scheduling decisions in data integration query processing are typically limited to the data sources for data retrieval, and a single, centralised machine for the rest of processing, whereas, in grid query processing, an arbitrarily large number of machines may be employed.

Submitting a query to retrieve data from a database connected to the Grid may not differ, from the user perspective, from querying a local database; for example, in SQL-like query languages the query may have the simple form of *Select-From-Where*. Combination of data in distributed databases naturally occurs in the case in which there are many database tables referred to in the *From* clause that belong to different databases, whereas, the user need not be aware of the exact location of each source. In addition, analysis tools, can be called on the data in the form of user-defined functions (UDFs), and, in this way, can be encapsulated in the query.

The standard approach to query processing involves the transformation of declarative queries into query plans, and execution of such plans in the context of a query engine. The operators that are expressed within such plans conform to algebras, and, typically, the phase of plan construction involves subsequent mappings of plans from one algebra (e.g., logical algebra) to another (e.g., physical or parallel algebra). A logical algebra comprises operators, such as *join, select,* and *project*. A physical algebra comprises operators, such as *hash-join, sort-merge, sequential scan, indexed scan*, and the parallel algebra extends the physical algebra by communication-related operators, such as *exchanges* [Gra93]. Before execution, the plan needs to be scheduled across physical resources. The decisions on plan construction and resource scheduling rely heavily on metadata about machine characteristics and data properties.

Since the 1970s and the development of influential systems such as the System-R [SAC+79] and its distributed counterpart System-R* [ML86], static compilation of query plans, and subsequent execution has been the main choice for database system developers. However, when the metadata required are not available or accurate at compile time, or change during execution, the query processor needs to revise the current execution plan. In this case, query processing is called *adaptive (or dynamic) query processing (AQP)*[1]. In AQP, the system monitors its execution and its execution environment, analyses this feedback, and possibly reacts to any changes identified, to ensure that either the current execution plan is the most beneficial, or no change in the current plan can be found that will result in better performance.

AQP is particularly relevant to settings in which query planning must take place

[1]Note that a narrower definition for AQP is adopted in this thesis compared to works such as [HFC+00], in the sense that here, the attention is restricted to cases where AQP produces effects during the execution of the same rather than of a subsequent query.

in the presence of limited or potentially inaccurate statistics for use by the query optimiser, and where queries are evaluated in environments with rapidly changing computational properties, such as loads or available memory [HFC+00]. As such, the relevance of AQP is growing with the prevalence of computing environments that are characterised by a lack of centralised control, such as the Web and the Grid [FK99]. Such environments are not only inherently more complex to model, but it is often the case that runtime conditions are sufficiently volatile to compromise the validity of optimisations made statically. In addition, robust cost models are harder to come by, thereby reducing the likelihood that the optimiser will select a sufficiently efficient execution plan [Kos00, ROH99]. The problem is further aggravated by the fact that useful statistics, like resource properties, selectivities and histograms (e.g., [BC02, Cha98, Ioa96]), may be inaccurate, incomplete or unavailable.

1.2.1 Main issues

Efficient query execution over the Grid poses challenges beyond those already addressed. The main issues in high-performance query processing on the Grid are summarised as follows:

- **The selection and scheduling of the resources** that will participate in query evaluation from an unlimited and heterogeneous pool. When the data sources are pre-specified, and the rest of query processing occurs in a single node, this is not an issue. However, query processing on the Grid, as discussed previously, does not fall into this category.

- **The volatility of the environment.** The Grid, as an execution environment, is subject to frequent changes. Machines are autonomous and owned by multiple administrative domains; thus they may join and be withdrawn from the Grid in an unpredictable way. Also, machine properties such as machine load, and amount of memory available are expected to change frequently. Consequently, queries need to be expressed and evaluated in a way that allows the runtime execution engine to map them onto the resources, and to adapt the query plan on the basis of information that is available at runtime.

- **The unavailability of statistics.** Traditionally, the query optimiser uses a potentially extensive set of statistics about the data processed and machines used. Because of the heterogeneity, volatility and autonomous status of data resources,

these statistics may not be available, or may be inaccurate and incomplete at compile time, or may change at runtime.

Also, research on query processing on the Grid is hindered by the unavailability of existing generic grid query processors. To date, there is little evidence in the literature of the existence of Grid-enabled query processors, apart from those the development of which is, to a certain extent, work described in this thesis, namely Polar* [SGW+03] and OGSA-DQP [AMP+03].

1.2.2 Outline of the approach

In this thesis, two directions for improving query processing on the Grid are explored: adaptivity and parallelism. It becomes apparent that solutions in which the initial decisions on the query plan and how this is scheduled over the available nodes cannot be changed after the evaluation of the query has started, may be inefficient, and cannot guarantee high-performance for a wide range of data- or computation-intensive applications over the Grid. To meet these challenges, concepts from the emerging technologies of AQP need to be employed [HFC+00], to adapt to the evolving resource characteristics.

Also, orthogonally to the issue of adaptivity, the performance of query processing can be enhanced in two ways [Has96]: by speeding up the individual modules of the system, and by using many modules for processing the query in parallel. The first approach, apart from being costly, cannot scale, as it is limited by the capabilities of the modules such as memory size, disk I/O speed and CPU cycles. However, the second approach, has been successfully deployed in databases [Sto86], due to the fact that query algebras naturally allow for parallelism [DG92]. To this end, novel scheduling techniques that select and schedule resources, based on their characteristics, in a way that query processing can benefit from parallelism need to be developed.

1.3 Research Context

There are several factors that motivate the investigation of query processing on the Grid, the combination of which provides an opportune and timely context for research on this area:

- Distributed Query Processing (DQP), which is a broader domain than query processing on the Grid, has become a mature technology [Kos00]. Thus, query

processing on the Grid can build upon established DQP techniques, along with other techniques for traditional query processing [Gra93], and concentrate more on the issues of adaptivity and parallelism that are not adequately addressed in traditional DQP.

- The emergence of Adaptive Query Processing (AQP) as an effective approach to correcting bad initial decisions on the plan construction [HFC+00, BB05].
- The research of this thesis follows research conducted on the performance of parallel object databases [SSWP04]. Consequently, there is a significant amount of experience in parallel query processing and the benefits of parallelism.
- The emergence of the Globus Toolkit [GLO] as the *de facto* standard middleware for the development of Grid applications.
- The development of generic wrappers for a broad range of commercial DBMSs, extending them with a Grid interface in the context of the OGSA-DAI project [OD]. This, combined with the existence of authentication and authorisation mechanisms within Globus, makes access to remote Grid databases practical.
- Finally, the research conducted in this thesis has both benefitted from and has contributed to the development of the Polar* and OGSA-DQP query processors for the Grid. These systems, as well as being a contribution in their own right, constitute the platform upon which the research outcomes have been tested.

Query processing on the Grid can have many flavours and it can be examined under different and complementary perspectives. For instance, it can be deemed as equally significant to develop techniques that try to optimise the performance of a single query when a set of resources has already been allocated to it, and to try to optimise the overall usage of resources that are shared between multiple queries. Also, the complexity and type of queries can range from simple ones that comprise a small number of operators of a limited operator set, to quite complex ones that comprise a large number of operators of arbitrary diversity. Consequently, it is important to state the exact context of the reaserch described hereby, and any underlying assumptions made:

- This thesis deals with the exploitation and integration of data and computational resources without being concerned how exactly these have joined a *virtual organisation*, how they can leave it, and how any authorisation and authentication checks have been conducted.

- The focus is on improving the performance of a single query, and the optimisation criterion is to reduce the query response time. Moreover, the queries that have been examined during evaluation are of the *Select-From-Where* type, and may involve calls to external user-defined functions. When the proposals of this thesis can be applied to a broader range of queries and operators, this is explicitly stated.

- The problem of how a query can continue its execution after a resource has left the *virtual organisation* has been examined in the context of the same project that this thesis is part of [SW04], but the research outcome of this work does not belong to the contributions of the present thesis, and thus will not be presented in detail.

- The emergence of the Grid has motivated the work described in this thesis, and any software development has been made according to Grid standards and on top of Grid middleware, such as the Globus Toolkit; nevertheless, the evaluation has been based either on simulation of a generic wide-area heterogeneous environment, or on a set of machines that belong to the same administrative domain. Although both these approaches may differ significantly from a realistic Grid environment, they have been regarded as adequate to support the claims of the present thesis.

1.4 Thesis Aims and Contributions

The aim of this work is to investigate, propose and evaluate techniques that are missing from classical query processing and improve significantly the performance of query processing on the Grid. The thesis is that, due to the inherent volatility and uncertainty of the Grid, AQP can provide greatly improved performance, and due to the fact that the Grid is oriented towards intensive applications, the performance of query processing can benefit significantly from parallelism.

In particular, the research described here makes the following contributions:

- A comprehensive scheduling algorithm, that allocates query operators to nodes, without restricting parallelism, and taking into account the properties of the (heterogeneous) machines that are available. Parallel query processing has been extensively investigated over homogeneous pools of resources [DG92], but it

has not been exploited to its full potential in query processing over heterogeneous machines [Kos00]. The result is that traditional DQP may be effective in integrating data sources, but lacks the infrastructure to cope with computation intensive queries in an efficient way. The proposal in this work addresses this limitation in a practical way, in the sense that it incurs low overhead and can scale well with respect to the number of machines available and the size of the query plan being scheduled.

- A framework for describing and constructing AQP systems in a systematic way, allowing for component reuse. AQP thus far suffers from two major shortcomings:

  – current techniques are designed in an isolated way, which does not permit them to be combined [IHW04, BB05]; and,

  – most AQP proposals have focused either on completely centralised query processing (e.g., [AH00, KD98]), or on centralised processing of data retrieved from remote sources (e.g., [AFTU96, Ive02]) and data streams (e.g., [MSHR02, CF03]). By following such an approach, the resources used for query execution are predefined, and thus the focus is mostly on adapting to changing properties of the data processed (such as cardinalities of intermediate results and operator selectivities). This is of paramount importance for query processing on the Grid, as crucial information about the data may be missing at compile time. However, of equal significance are adaptations to changing properties of the arbitrary set of resources that query processing on the Grid uses both for data retrieval and for other types of data manipulation, such as joins. Currently, AQP with respect to changing resources is not addressed as satisfactorily as with respect to changing data properties, such as operator selectivities and sizes of intermediate results.

  The framework proposed tackles the first of the two limitations above. It is based on the decomposition into and the separate investigation of three distinct phases that are inherently present in any AQP system:

  – *monitoring* of query execution and environment to collect the necessary feedback;

  – *assessment* of the feedback collected to establish issues with the current query execution or opportunities for improvement; and

  - *responding* to the identified monitoring events based upon the results of the assessment process.

- A generic mechanism for monitoring the execution of query plans on the fly. As adaptivity is likely to prove crucial, precise, up-to-date, efficiently obtainable data about runtime behaviour is essential, along with robust mechanisms that analyse such data. In general, the quality of adaptations and query optimisation decisions is largely dependent on the quality and accuracy of the information that is available. The Grid, and the Globus toolkit, provide support for monitoring resources, such as machine availability and network bandwidth. Complementary to this, information on the execution of the current query is also required for making adaptations dynamically. The present work investigates a generic monitoring approach based upon self-monitoring operators to meet this requirement, with the following key characteristics: (i) it can support a large spectrum of existing AQP techniques, (ii) it provides the foundation for building new ones, and (iii) it adheres to the notions of the adaptivity framework introduced earlier.

- A complete instantiation of the adaptivity framework that seeks to tackle the second limitation of existing AQP techniques identified previously, i.e., adapting to changing resources. In particular, two important cases are investigated:

  - adaptive workload balancing of parallel query processing; and
  - adaptive resource scheduling.

It is felt, and it has been found in this thesis, that all the above proposals are highly beneficial for query processing on the Grid.

1.5 Thesis structure

The remainder of this dissertation is structured as follows.

Chapter 2 presents background material on DQP and computational models for Grid programming and task execution. Then, it investigates the benefits of combining these two domains, and how this combination on the one hand facilitates DQP, and on the other, provides an alternative, promising model for describing and executing potentially complex tasks. The chapter also discusses in more depth the main research issues of the thesis. Finally, it introduces the two (static) Grid

query processors that are very closely related to the work of this thesis: Polar* and OGSA-DQP. The part of these systems that is the contribution of this thesis corresponds to the query compiler of the system, which is described in more detail than other components.

Chapter 3 addresses the problem of resource heterogeneity-aware scheduling of the query plan, in a manner that can exploit the benefits stemming from parallelism. The novel scheduling algorithm extends the Polar* system, and is implemented and evaluated in its context.

Chapter 4 discusses the *Monitoring-Assessment-Response* generic adaptivity framework, presenting its characteristics and benefits. Also, based on the notions of the framework components, it presents related work on AQP.

Chapter 5 describes and evaluates the self-monitoring operators approach to monitoring of execution plans. It corresponds to the monitoring component of the framework.

Chapter 6 provides an overview of the extensions built upon OGSA-DQP, to instantiate the adaptivity framework, using the technique described in Chapter 5 for monitoring. It also presents and evaluates two cases of AQP that are particularly relevant to query processing on the Grid: dynamic partitioning of workload, across machines that evaluate the same part of the query plan in parallel, according to the machines' actual (i.e., monitored) and continuously changing performance, and dynamic allocation of resources for computation-intensive parts of the query plan when such resources become available.

Chapter 7 reviews the thesis, and concludes with some suggestions for future investigation.

Related work is presented separately in every chapter where this is appropriate, because of the diverse nature of the issues examined. In particular, it is discussed in Chapter 2 for work on Grid databases, Chapter 3 for work on scheduling of distributed queries, Chapter 4 for AQP, and Chapter 5 for query monitoring.

Chapter 2

Query Processing and the Grid

Query processing and grid technologies have been evolving mostly separately [Wat03], being regarded as non-overlapping technological areas. This chapter serves a three-fold purpose: firstly, to present background material on these two areas, introducing the terminology, concepts and state-of-art; secondly, to discuss the issues involved in their combination, stating the benefits envisaged and the difficulties anticipated and thus having a closer look at the research issues of the current thesis; thirdly, to present two Grid-enabled query processors, namely Polar* and OGSA-DQP. The research aims described in this thesis have been pursued in close association with the development of these two systems.

The chapter is organised as follows: Section 2.1 gives an overview of Distributed Query Processing (DQP). Section 2.2 discusses how computations are described in Grids. Next, the novel opportunities stemming from the combination of Grid and database technologies, along with the main challenges, are discussed in Section 2.3. The final part of the chapter deals with grid-enabled query processors. The presentation of Polar* and OGSA-DQP is given in Sections 2.4 and 2.5, respectively. Other systems are described in Section 2.6. Finally, Section 2.7 summarises the chapter.

2.1 Distributed Query Processing

Query processing on the Grid involves submission of queries across multiple, logically interrelated, geographically dispersed databases that are connected via a network, and thus inherently falls under the scope of Distributed Query Processing (DQP). Distributed database systems are often characterised by their architectures and the capabilities of the participating nodes [Kos00]. Dominant architectural paradigms include

client-server, *peer-to-peer* [Ora01], and *wrapper-mediator* [GMPQ+97]. The machines comprising the distributed database systems may differ in their capabilities and characteristics, and may be autonomous. In the former case, the distributed database systems are referred to as *heterogeneous* (as opposed to *homogeneous*). If the distributed database systems at various sites are administered autonomously and possibly exhibit some form of heterogeneity, they are referred to as *federated* or *multidatabase systems* [OV99, Hsi92, Kos00]. This section examines the query processing aspects of such systems, and more specifically the typical phases during query processing (Section 2.1.1), the main optimisation approaches specialised for DQP (Section 2.1.2), execution techniques tailored to DQP (Section 2.1.3), and finally the relationship with parallel query processing and parallel databases (Section 2.1.4).

2.1.1 Phases of Query Processing

The success of databases lies largely in their ability to allow users to specify their tasks declaratively, relieving them from the burden of specifying how these tasks should be processed; the responsibility of devising an efficient evaluation manner rests with the query processor. Consequently, modern query processors have evolved into complex artifacts, encapsulating all the logic, prediction mechanisms, and techniques required to ensure near-optimal execution of the submitted queries. Typically, query processing, both for distributed and centralised settings, involves the following phases [GMUW01, Kos00]:

- Query Translation
- Query Optimisation
- Query Execution

Queries are expressed using declarative query languages, with the most common ones being SQL, OQL, XPath or XQuery. The first step is to parse these queries, and translate them into an internal representation, which is usually a form of well-established query algebras or calculus (*Query Translation* phase). At this stage, type-checking also takes place to ensure that the query is syntactically correct.

For each query, many different ways for evaluation may exist (unless it is very simple). The role of *Query Optimisation* is to examine the possible alternatives, or a reasonable subset of them, and establish which one is predicted to be the most efficient. The output query plan can be represented, in most of the cases, as a *directed acyclic*

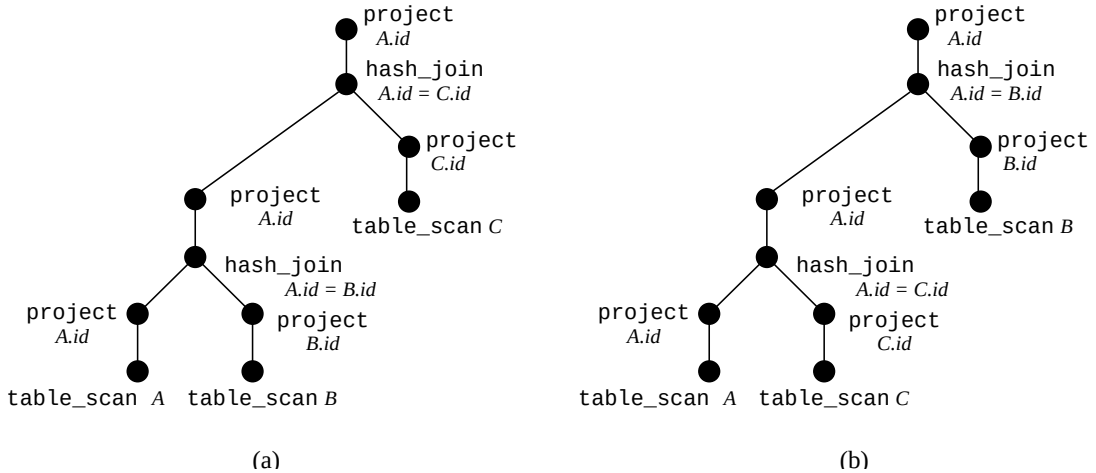


Figure 2.1: Candidate query plans for a three-table join query.

Name	**Signature**
sequential scan	*seq_scan(table name, predicate)*
hash join	*hash_join(left input, right input, predicate)*
project	*project(input, list of fields)*
unnest	*unnest(input, collection attribute, new field)*
operation call	*operation_call(input, external_program, parameters, predicate)*

Table 2.1: The signatures of the physical operators examined in this thesis. Each operator except scan has either one or two child operators as input.

graph (DAG), and more specifically as a binary tree, in which the nodes are query operators, such as *joins, scans, projects* and so on. The optimiser is responsible for deciding on the tree shape, the order of operators, and their implementation algorithm (e.g., a *join* can be realized as *hash join, nested loop, sort-merge* and so on). In a distributed setting, the optimiser is also responsible for the scheduling (or site selection) of the different operators in the query.

Suppose that there are three tables *A, B* and *C*, which all have a field named *id*, and we want to retrieve the *id* field of table *A* after joining the tables on the following conditions: *A.id = B.id* and *A.id = C.id*. In OQL, this can be expressed in the following way: *select a.id from a in A, b in B, c in C where a.id=b.id and a.id = c.id*. In simple words, in the *select* clause, the attributes that will be returned to the user are defined. The *from* clause contains the tables that provide the data. The *where* clause enumerates all the conditions on the values of attributes that have to be satisfied. The formal specification is part of the ODMG standard [CB00].

This query is transformed into a *select-project-join* query plan. Figure 2.1 shows two of the candidate query plans that can be considered in the scheduling phase. In general, the scheduling of multiple plans may be examined. The plans in the figure differ in the order of the joins. In this case, the optimiser is responsible for deciding which pair of tables should be joined first. Table 2.1 presents the implementation algorithms of operators that are taken into consideration in this thesis. The scan is responsible for extracting the data from a table, and filtering them if there is any predicate. The *hash-join* operator joins two data input. Firstly, it builds a hash table on the left input, and secondly, it probes this hash table by using the right input. *Project* is used to prune the list of tuple fields. The *unnest* operator is useful when there are collection attributes within a tuple: in this case, it flattens a tuple into a set of single-valued tuples. Finally, the *operation_call* encapsulates calls to (external) UDFs.

The final phase of query processing involves the actual *Query Execution*. The plan is sent to the execution engines at the sites decided by the optimiser. The engines provide implementations of query operators, and evaluate the plan received according to the *iterator* model of execution in almost any advanced system [Gra93, Kos00]. The benefits of this model include pipelining of data, avoidance of flooding the system with intermediate results in case of a bottleneck, and explicit definition of the execution order of query operators.

DQP builds upon two important assumptions regarding optimisation and execution, respectively:

- there exists a mechanism that specifies the set of resources (along with their characteristics) to be taken into consideration for query scheduling, and
- there exists a mechanism that addresses the security, authentication and authorisation issues that arise when remote resources are accessed.

The lack of such generic mechanisms limits the applicability of DQP over heterogeneous and autonomous resources.

In conventional static query processors, the three phases of query processing occur sequentially in the order they were presented. Adaptive systems differ in that they interleave query optimisation with query execution.

2.1.2 Query Optimisation

Although optimisation for single-node queries is well understood, at least for structured data [Ioa96, Cha98], optimisation of distributed queries still poses many challenges, as different distributed environments may shape it towards different directions [OB04]. For example, in some environments accuracy is not as important as returning the first results as early as possible, or the economic cost may be an issue.

Two main aspects of any optimisation approach, apart from whether it can affect the query execution after it has started as in adaptive systems, are:

- the search strategy that defines the way to generate, and consequently the number of, alternative plans, and
- the basis on which it is established which plan is the most efficient.

The bulk of work proposing *plan enumeration* algorithms to implement search strategies for DQP can be divided in two main categories: *dynamic programming* and *two-step (or two-phase) optimisation* [Kos00]. Dynamic programming builds trees in a bottom-up manner: more complex (sub)plans are derived from simpler ones, starting from the leaf nodes of the final query plan [KS00]. The algorithm, in its classical form, examines all the potentially optimal combinations of subplans, discarding the inferior solutions as early as possible. In this way, it ensures that the best plan (with respect to a cost metric) is actually found. However, this advantage comes at the expense of a prohibitively high computational complexity [Kos00], even when simplifying assumptions, such as that each part of the query plan is evaluated only at a single site, are made. Variants of the dynamic programming approach trade the guaranteed optimality of decisions with lower complexity, to make the techniques suitable for practical use.

The main alternative to dynamic programming is two-step optimisation. This approach, in the first step, optimises all the aspects of the query plan apart from the site selection; the resulting plan from this phase is a non-distributed one. The scheduling of this query plan occurs in the second step. Thus the complexity of optimisation drops significantly.

Orthogonally to the search strategy followed, the query optimiser needs to be equipped with mechanisms to compare different (sub)plans. The objectives may differ according to different settings. Classic objectives include query response time, and resource consumption that relates to query throughput. For their evaluation, *cost models* are incorporated in the optimiser. To a large extent, the quality of the optimisers that use such models depends on the accuracy of the models. In the absence of cost

models, *heuristics* may be employed, and in this case the optimisation is termed as *heuristic-based*, as opposed to *cost-based*. An example of a heuristic is to prefer plans that minimise the size of intermediate data produced during evaluation, or to minimise the volume of data needed to be transmitted over the network.

For the operation of the cost models and the appliance of heuristics, a potentially large variety of metadata is required [GMUW01, Ioa96, Cha98]. Such metadata contain information about (i) the data processed, such as sizes, cardinalities and histograms, (ii) the operation selectivities, and (iii) resource information, such as amounts of memory available, characteristics of machines, and time costs to read pages from disk.

Cost models for distributed settings extend the models developed for centralised systems in that, as well as the CPU and disk input/output related costs [GMUW01], they take into consideration communication costs (e.g., [GHK92]). The biggest challenge though is that, especially for distributed systems comprising autonomous resources, the statistical information needed by query optimisers is often not readily available and may be inaccurate, and information about the characteristics of the available machines may be incomplete [LOG92, EDNO97]. Several techniques have been proposed to tackle this problem. In wrapper-mediator approaches, wrappers are extended to expose statistics (e.g., [ROH99]). Other techniques rely on dummy queries running before execution to collect statistics (e.g., [ZSM00]), on cost model calibration (e.g., [RZL04, RZL02, GST96]), or on probabilistic models (e.g., [ZMS03]).

Many modern distributed query applications involve the combination of different data sources, which may belong to different administrative domains and possess data of various quality levels. Furthermore, the resources may not be provided for free. Consequently, users may be interested, as well as in the execution time, in aspects such as monetary cost, data freshness, volume of data accessed, and so on. In such scenarios, quality of service criteria become more important than query costs, and the optimiser is constructed to reflect this situation (e.g., [BKK+01, SAL+96]). In addition, data stores may have different capabilities for providing data relevant to the query. Thus, the decisions of an optimiser operating in such environments are influenced, if not dictated, by such capabilities (e.g., [GMPQ+97]).

Finally, in adaptive query processing, the optimiser policies, or a subset of them, are reevaluated in light of new, updated information that becomes available during query execution.

2.1.3 Query Execution

Query execution in DQP considers more issues than in traditional centralised systems [Gra93]. In particular, the issue of selecting the location of data stores arises when multiple options exist [Kos00]. *Dynamic data placement* techniques, given the appropriate authorisation, redistribute data copies in such a way that queries can be executed more efficiently, as the data location has significant impact on load balance and communication cost. This can be achieved by two main categories of techniques, namely *replication* of data sources and *caching* of (partial) results of queries for potential reuse. Other execution techniques focus on increasing the pipelining of evaluation (e.g., [IFF+99]) to return to the client initial results as early as possible. This is important when a query is investigatory (e.g., a web query for flights) and the user wants to terminate it as soon as a sample of results has been received. Decreasing communication costs has also attracted a lot of interest. Approaches to this end include *multithreading* to overlap communication with the rest of processing, *row blocking* to reduce the number of times communication occurs, and *semijoins* [BGW+81] to reduce the volume of data transmitted [Kos00]. Adaptive systems are also equipped with the capability to suspend and restart execution.

2.1.4 Parallel databases

Parallel databases may be deemed as a particular type of distributed databases, especially when they adopt the *shared-nothing* architectural model [Sto86]. A main difference is the nature of participating resources: distributed databases comprise independent, and often heterogeneous, geographically dispersed machines, whereas parallel databases are individual systems with multiple processors residing in the same location [Ioa96]. Operating in a more controlled environment, parallel databases are free of the complexities imposed by unknown information, heterogeneity and autonomy, and can concentrate more on performance issues, such as achieving linear *speed-up* and *scale-up* [DG92].

Evaluation can be speeded up by processing the query plan in parallel, transparently to the user. The three classical forms of parallelism in DQP are *independent*, *pipelined* and *partitioned (or intra-operator)*. Independent parallelism can occur if there are pairs of query subplans, in which one does not use data produced by the other. For instance, in the left plan in Figure 2.1, tables *A* and *C* can be scanned in parallel. Pipelined parallelism covers the case where the output of an operator is consumed

by another operator as it is produced, with the two operators being, thus, executed concurrently. This might be the case of table *B* being scanned, and the output to be directly processed by the *project* operator in order to be used to probe the hash table that contains the tuples from table *A* in Figure 2.1(a). In partitioned parallelism, an operator of the query plan has many clones, each of them processing a subset of the whole data. This is the most profitable form of parallelism, especially for data and computation intensive queries. For example, multiple instances of the scan operator for table *A* can be spawned, each retrieving a different set of rows.

In theory, all these forms are applicable to DQP, and can be used to speed up distributed applications as well. In this case, query processing is termed as being both distributed and parallel.

2.2 Computations on the Grid

Programming Grid applications directly upon the low-level protocols and infrastructure provided is a laborious, error-prone and inflexible task [FK03]. Most commonly, Grid computations are described at a higher level, and may be executed in the context of more generic computing environments that act as middleware [Ken03]. In query processing on the Grid, computations are expressed, as already discussed, as declarative queries. This section deals with other typical forms of tasks on the Grid, so that the merits of the query processing approach can be better demonstrated (in the next section).

There exist three main approaches to describing computations on the Grid [Ken03]:

- A significant portion of Grid computations can be described as a set of loosely coupled, master-slave, subtasks that may be executed concurrently, with trivial inter-dependencies and synchronisation requirements. Usually, these subtasks evaluate the same algorithm over a different dataset. Such applications are particularly demanding when the complete dataset is large. A well-known representative of this class is the SETI@home project [ACK+02], which uses Internet-connected computers in the Search for Extraterrestrial Intelligence (SETI) on a volunteer basis. Each contributing machine participates by analysing a small portion of a large amount of radio telescope data at each run. Analogous examples can be drawn from various scientific fields including fluid dynamics, particle physics, astronomy and biology. Several generic execution environments

have been developed for this sort of computation. They focus either on achieving high throughput by exploiting idle CPU time of commodity resources (e.g., [ACK+02]), or on meeting more complex user constraints on performance, economic cost, and so on (e.g., Nimrod-G [BAGS02]).

- Nowadays, an increasing number of tasks, both from the business and the academic sector, are described as and automated by workflows. Compared to the previous approach, workflows are suitable for applications whose subtasks have stronger inter-dependencies. An example of such an application is the ^{my}Grid project [GPSG03], which has developed a loosely-coupled suite of components to support data-intensive *in-silico* experiments in biology. As for the previous approach, there are systems that provide a workflow execution environment on the Grid. For example, Condor [FTL+01, TTL03] provides a tool to describe workflows that have the form of a directed acyclic graph (DAG). Other systems target applications of specific requirements, e.g., the DataCutter project [BFK+00] is well suited for data-intensive workflows.

- In a simpler case, Grid computations can be expressed as, and encapsulated in calls to Grid portals, so that the user effort to describe tasks is minimised. Portals can be particularly designed for a scientific domain (e.g., computational chemistry, astrophysics and so on), or to provide generic services such as basic access to Grid resources [Ken03]. However, there is a trade-off between the simplicity of the description of the computation and its potential complexity.

2.3 Combining Database and Grid technologies: opportunities and challenges

This section discusses (i) why query processing is an appealing paradigm for describing and executing potentially complicated tasks on the Grid, compared to other approaches that were discussed in Section 2.2; and (ii) which aspects of DQP that were introduced in Section 2.1 pose the greatest challenges when applied to query processing on the Grid.

2.3.1 Novel Opportunities

The combination of DQP and Grid technologies is beneficial from both perspectives. DQP provides a computing paradigm in addition to those described in Section 2.2, thus enabling more applications to benefit from the Grid. On the other hand, the Grid can significantly facilitate DQP, thus forming a relationship of mutual interest and presenting new opportunities for advanced applications. This can be better illustrated with the help of the following example that combines data integration and analysis to perform a non-trivial task.

EXAMPLE 2.3.1 *Suppose that two independent bioinformatics labs work jointly on a project, which involves finding proteins that (i) belong to a specific category catA and (ii) are similar to those proteins that have successfully interacted with more than 1% of a set of testing proteins. One lab has expertise in ontologies classifying the proteins, and exposes to the other lab a relation Classification(contologyid, cproteinid) representing the classification of the proteins. The other lab conducts the interaction experiments and publishes a relation Interactions(iproteinid, iprotein, iproportion) holding information about the proportion of successful interactions of each protein. Protein similarity is established on the grounds of the sequence alignment scores provided by widely adopted tools like BLAST [AGM+90].*

Figure 2.2 shows a sequence of steps that could be implemented as a workflow and performs the task of Example 2.3.1. Initially, the relevant data are retrieved from their stores. Following this, the data referring to the same protein are matched, before BLAST is called on them. Each of these subtasks is implemented separately and is specific to this experiment. The performance of the system depends on the implementation of the subtasks. For example, parallelisation issues and selection of appropriated resources is coded within the subtasks. Also, as the complexity of a request increases, it becomes increasingly difficult for a developer to make decisions as to the most efficient way to program the sequence of steps.

In grid querying, this computation can be written in a query language as shown in Figure 2.3. The query retrieves the relevant data from the two databases, joins them according to the join condition, and calls the BLAST external function, producing the same results as previously. A big advantage is that the developers are only required to express their requests. In addition, performance issues are handled by the query optimiser in a way transparent to the user. For example, as sequence alignment is a notably expensive operation, the optimiser may decide that partitioned parallelism

```
Task 1: Retrieve data from Classification with
        contologyid = catA.
        Store results temporarily.
Task 2: Retrieve data from Interactions with
        iproportion > 0.01.
        Store results temporarily.
Task 3: Match data from the two result sets if the have
        the same protein identifier.
Task 4: Call BLAST on results of Task 3.
```

Figure 2.2: The sequence of steps corresponding to the example 2.3.1

```
select BLAST(iprotein)
from Interactions I, Classification C
where I.iproteinid = C.cproteinid
and I.iproportion > 0.01
and C.contologyid = catA
```

Figure 2.3: The query corresponding to the example 2.3.1

should be applied to the BLAST calls. To this end, additional BLAST repositories belonging to third parties are selected and employed, without requiring any particular configuration by the persons conducting the experiment.

The above example demonstrates that DQP is an appealing solution for a broad range of Grid applications due to its:

- Declarative, as opposed to imperative, manner of expressing potentially complex computations that integrate independent data resources and analysis tools, which are currently either not feasible, or must be carried out using non-database technologies, such as workflows. As such, DQP can
  - increase the variety of people who can form requests over the Grid since these requests do not require an in-depth knowledge of the Grid technologies; and
  - reduce development times for the Grid programming tasks that can be expressed as database queries.
- Implicit provision of optimisation and parallelism that makes efficient task execution more likely.

The Grid is also an appealing platform on which DQP can be deployed. Query processing on the Grid is characterised by the following key differences from traditional DQP over heterogeneous and potentially autonomous databases:

- The Grid provides systematic access to remote data and computational resources, addressing the security, authentication and authorisation problems involved, and, as such, the Grid enables remote sources to be used not only for data retrieval tasks, but also for computational ones [FK03].
- The Grid provides mechanisms for dynamic resource discovery, allocation and monitoring [CFFK01].
- The Grid provides mechanisms for monitoring network connections [WSH99], which is an essential feature for a query engine to efficiently execute queries in wide-area environments.
- Grid middleware conforms to (currently evolving) standards [DAI] and there exist publicly available reference implementations for uniform Grid-enabled access to commercial Object-Relational and XML databases [OD].

Consequently, Grid environments meet the basic requirements for efficient deployment of DQP, as identified in Section 2.1.1.

2.3.2 Novel Challenges

2.3.2.1 Extensive Adaptivity

A significant similarity between Grid and traditional DQP is the need for adaptivity during query execution [HFC+00]: the success and endurance of database technology is partially due to the optimisers' ability to choose efficient ways to evaluate the plan that corresponds to the declarative query provided by the user. The optimiser's decisions are based on data properties, such as cardinalities and predicate selectivities, and on environmental conditions, such as network speed and machine load, as explained in Section 2.1.2. In both Grid-enabled and non-Grid-enabled DQP over heterogeneous and autonomous sources, information about data properties is likely to be unavailable, inaccurate or incomplete, since the environment is highly volatile and unpredictable. In fact, in the Grid, the virtual organisation and the set of participating resources is expected to be constructed either per query or per session [AMP+03].

These characteristics of the Grid environment have an impact on an extensive set of the decisions of the optimiser, in contrast with the cases addressed by existing adaptive techniques [HFC+00, GPFS02]. The quality of the decisions of the optimiser components is basically controlled by the quality of their input information rather than by the policies they implement. This is due to the fact that such policies are typically well established and validated. Examples of issues arising in Grid environments include:

- *Decisions on operator order and shape of the query plan:* Such decisions are affected mostly by the sizes of the input and intermediate data. For the former, accurate statistics about the data stores needs to be available, and for the latter, information about the predicate and the filter selectivities are required. Due to the expected lack of such accurate information, it is unlikely that the initial decisions by this component would be near optimal.

- *Decisions on physical implementations of operators:* Mapping a logical algebraic query plan to a physical one involves the mapping of each logical operator to one of its potentially many physical implementations. For example, a logical join can appear in a query execution plan as a (blocked) nested loop, a hash join, a sort-merge join, a pipelined hash join, and so on. Knowing the specific, individual physical characteristics of computational resources, such as the amount of the available memory, and properties such as ordered attributes, is crucial for ensuring good performance.

- *Decisions on plan scheduling:* The optimiser's scheduler assigns a subplan to at least one physical machine. If the execution mechanism does not provide implicit mechanisms for defining the order of operator execution within a subplan (e.g., it does not follow the iterator execution model [Gra93]), the scheduler needs to make these additional decisions as well. Its policies are based mostly on the properties of the physical resources.

Existing solutions for adaptive query processing (AQP) can address only partially the above issues. They can compensate for inaccurate or unavailable data properties (e.g., [AH00, KD98]), bursty data retrieval rates from remote sources (e.g., [Ive02]), and provision of prioritised results as early as possible (e.g., [RRH99]), but they also suffer from the following major limitations, which prohibit their usage in query processing on the Grid:

- They are too specific in terms of the problem they address and are designed in isolation [IHW04]. As such, they cannot easily be combined to meet the broader adaptivity demands of query processing on the Grid.
- They mostly focus on centralised query processing.
- They do not yet provide robust mechanisms for responding to changes in the pool of available resources, even when the data are initially stored remotely.

Advances in AQP are presented in more detail in Chapter 4. However, the above limitations make more obvious the need for

- definition of abstractions and architecture paradigms that unify the currently isolated adaptive techniques in a generic framework, and
- novel extensions to AQP techniques to cover the requirements of Grid query processing.

2.3.2.2 Harnessing the available power

As mentioned in Chapter 1, the most efficient and scalable way to enhance the performance of query processing is to increase the number of its modules operating in parallel. The forms of parallelism in query processing were presented in Section 2.1.4. The execution engines typically provide for pipelined parallelism through the iterator execution model (Section 2.1.1). To employ partitioned parallelism, the scheduler part of the optimiser needs to have the capability to select and allocate multiple resources to the same plan fragment. The Grid offers an arbitrarily large, and inevitably heterogeneous pool of resources. However, the selection and scheduling of the resources that will participate in query evaluation from an unlimited and heterogeneous pool is an open issue, even in its static form. Generic Grid schedulers, like Condor [TTL03], support DAGs that can represent query plans but they do not provide for pipelined or partitioned parallelism. Existing scheduling algorithms for distributed databases either support limited partitioned parallelism if all the participating machines have the same capabilities (e.g., [ESW78]), or no partitioned parallelism at all (e.g., [ML86]). Thus, they are inappropriate for intensive query applications and unable to harvest the benefits of the typically heterogeneous resources that a Grid makes available to its users.

2.4 Polar*: An early Grid query processor

The purpose of this and the following section is to present two real query processors for the Grid that can realise the concept of evaluating data combination and analysis tasks using query processing, as discussed in Section 2.3. The sections discuss also the optimisation and evaluation techniques that the two systems employ, and the extent to which (i) they adopt traditional DQP techniques, as presented in Section 2.1, and (ii) they address the challenges to DQP arising from the Grid environment, as discussed in Section 2.3.

Polar* is the first recorded Grid-enabled query processor [SGW+03, SGW+02]. It accesses remote data stores and performs remote operation calls using the Globus 2 middleware.

Polar* evolved from an object-oriented (and more specifically, ODMG [CB00] compliant) parallel database system, called Polar [SSWP04]. As an object-oriented DQP system, Polar* is capable of handling complex data structures that are common to many real scientific applications. As an evolution of a parallel system, it is equipped with mechanisms enabling intensive use of parallelism. Results presented in [SGW+03] show that parallelism in DQP over heterogeneous resources can have beneficial effects, similarly to what happens in conventional parallel computing over homogeneous machines. By being capable of evaluating queries such as the one in Figure 2.3, Polar* has illustrated that DQP can serve as a suitable platform for tasks that

- integrate data from multiple sources, and
- call analysis tools on the data.

As such, the development of Polar* has served as a proof of concept, validating the claims about the potential benefits of Grid query processing made in Section 2.3.

The rest of the section provides a brief overview of the data model employed in Polar*, the architecture, the compiler and the execution engine.

2.4.1 The ODMG data model

Polar* adopts the model and query language of the ODMG object database standard [CB00]. As such, all resource wrappers must return data using structures that are consistent with the ODMG model. Queries are written using the ODMG standard query language, OQL.

The reason for choosing the object data model rather than the relational one, is that it provides a richer model for source wrapping and for representing intermediate data. The model used for DQP is thus rich enough to support straightforward wrapping of relational and object databases. In addition, the object model, as opposed to the relational one, fits better with applications and computations that use advanced data structures and require relatively more powerful operations. Features of object databases that must be dealt within (distributed) query processing are the allocation of unique identifiers to all the objects thereby supporting navigation and path expressions in queries, the support of collection-valued attributes, and the fact that an object database can be accessed both through a query language and also by mapping database objects into applications.

The ODMG specification includes the Object Definition Language (ODL), which is programming language independent, and is used to specify the database schema. Figure 2.4 shows the database schema of Example 2.3.1 in ODL. In addition, it defines the Object Interchange Format (OIF) for retrieving and storing data from and to (flat) files, respectively.

2.4.2 Architecture

Figure 2.5 illustrates the high level architecture of the system. The metadata repository, which is needed to compile a query, is populated by both data from the database schema (e.g. types of attributes) in the ODL format, and information about the Grid nodes. Each Grid node has specific computational capacities and data stores. The coordinator node is a usual Grid node with the additional characteristic that it stores the metadata and has the Polar* query compiler installed, as well as the evaluation engine.

Polar*, like its predecessor, Polar, is designed in a modular way, comprising a set of distinct components (Figure 2.6). The components communicate with each other through well defined interfaces, without revealing their implementation details. The components are:

- The Query Compiler, which includes:
  - The Query Parser.
  - The Query Optimiser, which comprises: (i) the Logical Optimiser; (ii) the Physical Optimiser; (iii) the Partitioner; and (iv) the Scheduler.
- The Query Evaluator.

```
forward class Classification;
forward class Interaction;

struct blastResult {
 string protein;
 long score;
};

static set<blastResult> BLAST(in string protein);

class Classification
 (extent Classifications
  key contologyid )
 {
  attribute string contologyid;
  attribute string cproteinid;
 };

class Interaction
 (extent Interactions
  key iproteinid )
 {
  attribute string iproteinid;
  attribute string iprotein;
  attribute string iproportion;
 };
```

Figure 2.4: The ODL schema corresponding to Example 2.3.1

- The Metadata Repository.

The parser is responsible for the translation phase. Optimisation occurs according to the two-phase approach. As such, it is divided into single-node and multi-node optimisation. The logical and physical optimisers perform the former, whereas the partitioner and the scheduler carry out the latter. The instances of query evaluators execute the plan produced by the optimiser. There is one instance per machine selected to participate in the execution. The components may access the metadata repository to perform their tasks. The metadata store is built upon Shore [CDF+94], in compliance with the ODMG standard.

Polar* is a static query processor. As such, the query translation, optimisation and execution occur sequentially for each incoming query. The rest of the section examines in more detail the way in which query plans are constructed and executed in Polar*.

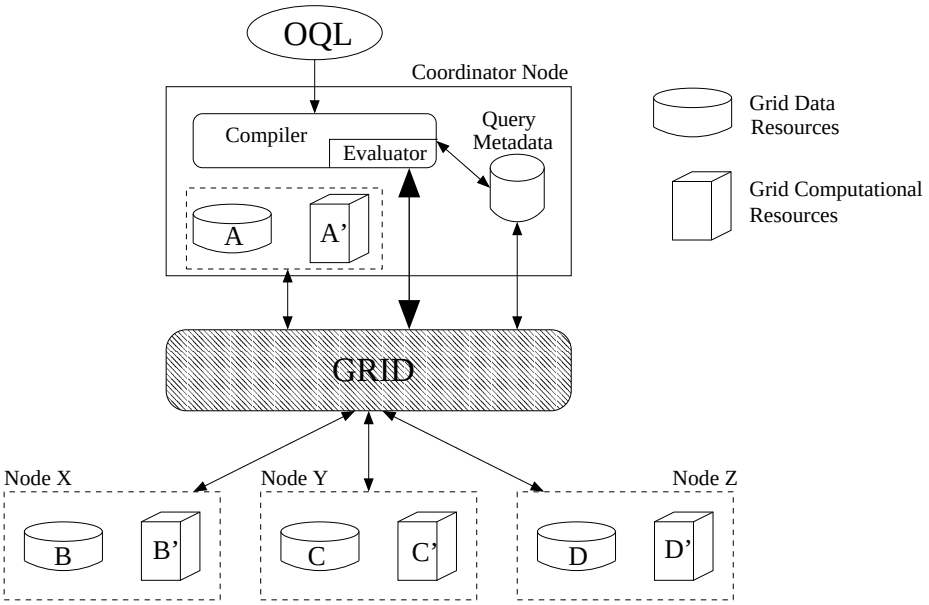


Figure 2.5: The Polar* architecture.

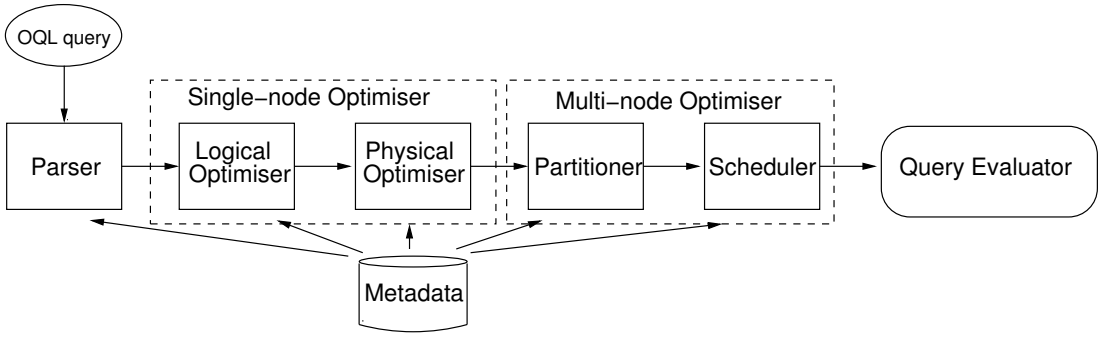


Figure 2.6: The components of Polar*.

2.4.3 Query Planning

As mentioned previously, Polar* follows the two-step optimisation paradigm, which is popular for both parallel and distributed database systems [Kos00]. For the single-node optimisation, large parts of the optimiser of lambda-DB [FSRM00] have been incorporated. The internal representation of plans in lambda-DB, and consequently in Polar*, is based upon the *monoid calculus* and the associated algebra [FM00] of Fegaras and Maier. As such, the OQL queries, are parsed and transformed into a monoid calculus expression. Subsequently, the plan in this intermediate format is typechecked and then is ready for the actual plan construction.

2.4.3.1 Construction of the single-node plan

The single-node plan is constructed by applying logical and physical optimisation to the output of the parser.

The logical optimiser operates as follows.

- It *normalises* the plan in monoid calculus, which involves (i) query unnesting, (ii) fusion of multiple selection operators into a single one, and (iii) application of DeMorgan's laws to the predicates.

- It maps the monoid calculus into the logical algebra of [FM00]. Figure 2.7(a) depicts a plan for the example query in Figure 2.3 expressed in the logical algebra. The logical algebra contains object-relational operators such as *scan, unnest,* and *project* without specifying their actual implementation, when more than one exists (e.g., *hash join* and *nested loop* are both possible implementations of the join logical operator).

- It creates multiple equivalent logical plans and chooses the one that results in the production of less intermediate data [Feg98]. Changing the order of the operators and the shape of the query results in plans that produce different numbers of intermediate results.

- It pushes, or inserts, projections as close to the scans as possible.

The overall aim of the optimisation is to create a plan with as low a query response time as possible. The main heuristic employed (i.e., to reduce the volume of temporary data) performs well to this end [Feg98], incurring minimal overhead, as it is based on a greedy bottom-up algorithm.

The physical optimiser transforms the optimised logical expressions into physical plans by selecting algorithms that implement each of the operations in the logical plan (Figure 2.7(b)). For example, in the presence of data ordering, the optimiser prefers *sort-merge joins* to *hash joins*. The mapping policies are defined in rules, and transformed into code by the OPTGEN optimiser generator [Feg97].

The number of physical plans that are produced is configurable; the higher this number is, the more options for the multi-node optimiser exist, but also, the higher the query compilation cost is.

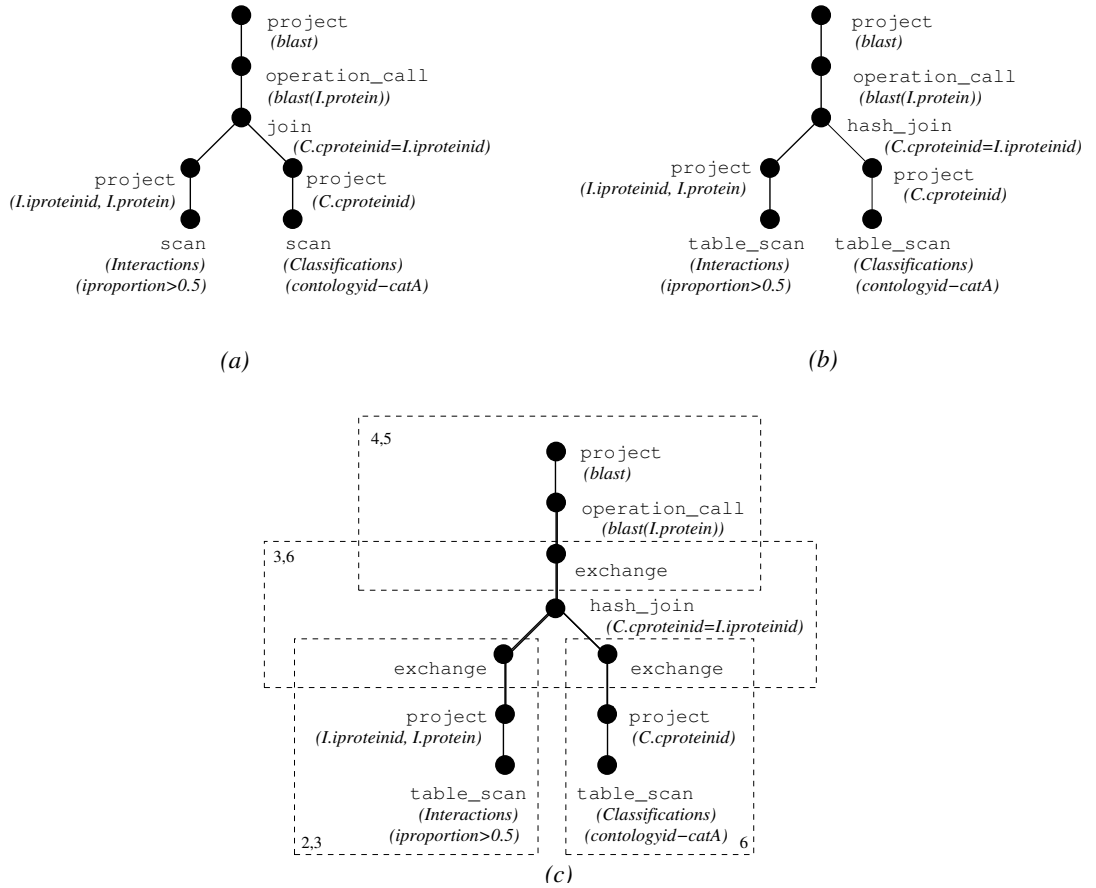


Figure 2.7: Example query: (a) single-node logical plan, (b) single-node physical plan, (c) multi-node physical plan.

2.4.3.2 Construction of the multi-node plan

The multi-node optimiser comprises two components: the *partitioner*, which splits the plan into fragments, thereby defining the points where communication over the network takes place, and the *scheduler*, which allocates a set of machines to each such fragment.

Plan partitioning The first step to transform a single-node plan into a multi-node one is by inserting parallelisation operators into the query plan, i.e., Polar* follows the operator model of parallelisation [Gra90], by using the *exchange* operator. Exchanges comprise of two parts that can run independently: exchange producers and exchange consumers (Ex-Prod and Ex-Cons in Figure 2.8, respectively). The producers have an outgoing buffer for each consumer, in which they add the tuples they collect from the operators lower in the query plan. For each *exchange* operator, a

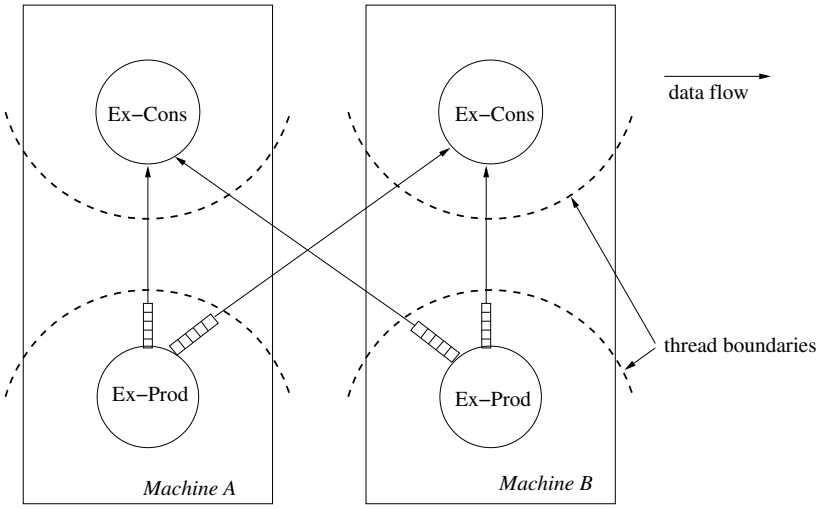


Figure 2.8: The exchange operators

data distribution policy needs to defined, in order to identify the consumer for each tuple. Currently, the policies Polar* supports include *round_robin*, *hash_distribution* and *range_partitioning*. The last two policies provide support for non-uniform data distribution among instances of the same physical operator, which is desirable for heterogeneous environments.

The partitioner firstly identifies whether an operator requires its input data to be partitioned by a specific attribute when executed on multiple processors (for example, so that the potentially matching tuples from the operands of a join can be compared). These operators are called *attribute sensitive* operators [Has96]. [SPWS99] presents the classification of the parallel operators as attribute sensitive or attribute insensitive. Secondly, the partitioner checks whether data repartitioning is required, i.e., whether data needs to be exchanged among the processors, for example for joining or for submitting to an *operation_call* on a specific machine. There are two such cases: (i) when the children of an attribute sensitive operator in a query plan are partitioned by an attribute other than its partitioning attribute, or there is no data partitioning defined for them, then data repartitioning needs to take place; and (ii), when not all the candidate nodes for evaluating parts of the query can evaluate a physical operator, which may be case for *operation_calls*. The physical algebra extended with *exchange* constitutes the parallel algebra used by Polar*. The *exchange*s are placed immediately below the operators that require the data to be repartitioned. A multi-node query plan is shown in Figure 2.7(c), where the *exchange*s partition the initial plan into many subplans (delimited by dashed lines in the figure).

In the example query, there is one attribute sensitive operator, the *hash_join*, which needs to receive tuples partitioned by the *proteinID* attribute. Since the children of the *hash_join* are not partitioned in this way, two *exchanges* are inserted at the top of each subplan rooted by the join. *Operation_call* is attribute insensitive, but since it can be called only from specific nodes, an *exchange* is inserted as its child. The optimiser checks if the *exchanges* transmit redundant data. A *project* operator ensures that only data that are required by operators higher in the query plan are transferred through the network. If there are no such *projects* placed by the single-node optimiser, the parallel optimiser inserts them.

Plan scheduling The final phase of query optimisation is to allocate machine resources to each of the subplans derived by the partitioner, a task carried out by the scheduler in Figure 2.6 using an algorithm based on that of Rahm and Marek [RM95]. This algorithm is principally for parallel databases, which consist of homogeneous nodes. As such, it can operate well in a limited range of cases in grid querying.

To run the example query, suppose that six machines are available, and that three of the machines host databases (numbers *2* and *3* for the *Classification* database table, and *6* for the *Interactions*). The *table_scan* operators are placed on these machines in order to save communication cost. For the *hash_join*, the scheduler tries to ensure that the relation used to construct the hash table can fit into main memory, for example, by allocating more nodes to the join until predicted memory requirements are satisfied or all available memory is exhausted [RM95]. In the example, nodes *3* and *6* are allocated to run the *hash_join*. As some of the data is already on these nodes, this helps to reduce the total network traffic. The data dictionary records which nodes support BLAST, and thus the scheduler is able to place the *operation_call* for BLAST on suitable nodes (*4* and *5* in Figure 2.7(c)). The cost of a call to BLAST is much higher than the cost to send a tuple to another node over the network. Additionally, in any case, the whole set of the results of the *hash_join* needs to be moved from the nodes *3* and *6* at some point, to be returned to the user, which means that some communication cost is inevitable. For these two reasons, the optimiser has chosen the maximal degree of parallelism for the BLAST operation. This choice incurs lower computation cost, without increasing the communication.

2.4.4 Query Evaluation

The query engine implements each operator according to the *iterator* model. As such, operators implement three main methods: *open*, *next*, and *close*. The root operator of the query tree calls *open* on its children, the children call it on their children, and so on until the *open* call is propagated to the leaf operators. *Next* is called in the same manner for each tuple, until there are no other tuples to be processed. At that point, the propagation of *close* occurs to finish the execution in a tidied-up way. In centralised query processing, applying the *iterator model* results in a pure pull-based mode of execution. The presence of *exchanges* alters this characteristic as it enforces its producers and consumers to run independently in different threads. Thus, in a multi-node setting, parts of the query plan are executed simultaneously.

Apart from multithreading, Polar* employs another technique common to DQP systems (Section 2.1.3), which is row blocking. The tuples, which may represent complex collection structures, are serialised and placed into buffers of configurable size. Communication between nodes occurs only if the buffers are filled or the buffers are partially filled but there are no more data to be processed. The underlying mechanism is MPICH-G [KTF02], which is a grid-enabled implementation of MPI (Message Passing Interface) [SOHL$^+$98].

Polar* supports two kinds of data repositories, which affect the implementation of the *table_scan* operator. Firstly, there is built-in support for data retrieval from the Shore [CDF$^+$94] server, which supports the ODMG model. Secondly, Polar* enables access to commonly used database systems, such as MySQL, through ODBC and client libraries. In this case, the results are transformed into the ODMG model, through OIF mappings, to ensure that there is a common representation of the tuples evaluated.

Another operator that is of specific interest is the *operation_call*. This operator is capable of loading code dynamically that conforms to statically generated stub code based on signatures of external functions defined in the ODL schema. The stub code is responsible for transforming the input tuples into the format required by the function, and the output tuples into the ODMG format.

In summary, query execution in Polar* is performed as follows. The evaluators receive a plan fragment from the query compiler, which is installed on them via the MPICH-G interface. Next, the operators in the plan fragment are executed as iterators. This operator execution may in turn lead to the transmission of data between nodes, using the MPICH-G interface. MPICH-G is layered above the low-level Globus 2 infrastructure, hiding many of its implementation details and providing a higher-level

Application Programming Interface (API) for the development. As such, Polar*, like Polar, realizes a parallel query as a MPI program. Query processing in Polar* involves concurrent login to multiple machines, transfer of executables and other required files, and start-up of processes on the separate resources. Concurrent login to separate accounts is managed through GSI (the Grid Security Infrastructure for enabling secure authentication and communication [FKT01]), executable staging across wide area connections through GASS (Global Access to Secondary Storage [FKT01]), and start-up through DUROC (Dynamically-Updated Request Online Coallocator [FKT01]). However, MPICH-G provides an interface to all these features.

2.4.5 Polar*'s approach regarding the novel challenges

Section 2.3 identified two main directions for efficient DQP on the Grid: adaptivity and parallel scheduling. For the former, the Polar* provides very limited features, as it is designed as a static query processor. The only dynamic feature is that during data communication handled by *exchanges*, a lagging consumer does not constrain the flow to a quicker consumer, thus hiding limited fluctuations in resource performance and availability [SHCF03].

Regarding scheduling, the Polar* prototype supports partitioned parallelism, but it bases its approach on techniques from parallel homogeneous databases [RM95]. Thus, it essentially does not take into account the heterogeneity of the Grid environment.

2.5 OGSA-DQP: Services meet Grid query processing

Web Services (WSs) [GGKS02] have emerged as a broadly adopted and appealing computing paradigm for loosely-coupled distributed applications, as they provide language and platform-independent techniques for describing, discovering, invoking and orchestrating collections of networked computational services. The significance of their benefits has forced the Grid community to recast its middleware functionalities as *Grid Services (GSs)* and thus to develop the Open Grid Services Architecture (OGSA), and a reference middleware implementation released as Globus 3 [FKNT02, TCF+02]. Web and Grid service technologies are converging rapidly and they complement each other. Indeed, WSs mostly focus on platform-neutral description, discovery and invocation, while GSs are more interested in managing service state and lifetime, dynamic discovery and efficient use of distributed computational resources.

OGSA-DQP [AMP+03] is a service-based distributed query processor, which reuses large parts of Polar*, and is built on top of Globus 3. OGSA-DQP is service-based in two orthogonal dimensions:

- it manifests itself as a collection of services, and
- it accesses remote data repositories and analysis tools that are in the form of services.

OGSA-DQP services are built on top of *Grid Data Services (GDSs)* developed in the context of the OGSA-DAI initiative [OD], which aims to provide a common service interface to Grid-connected DBMSs. GDSs extend the basic GSs of Globus 3. In the remainder of this section, the main characteristics of each of the above classes of service are discussed.

2.5.1 Grid Services and Grid Data Services

GSs in Globus 3 implement basic, application-independent functionalities that are useful to any service. They implement a standard interface for registering their instances with registry services. This interface enables other services to query the registries and retrieve information about them. In this way, they provide for registration and discovery. Instances are created dynamically from *Service Factories* and are identified by their unique *handle*. Their lifetime is manageable; thus failed instances can be disposed of in a tidy manner. GSs provide also mechanisms for notification subscription, production, delivery and receipt.

GDSs extend GSs by providing a registry facility for the publication of other GDSs. They are also capable of creating instances tailored to specific, application-dependent requirements with regard to database location and query language. Also, GDSs can be Grid-enabled wrappers of relational, object-relational and XML databases.

2.5.2 OGSA-DQP Services

OGSA-DQP [AMP+03] re-engineered Polar* to conform to service-based principles. OGSA-DQP defines two services, both extending GDSs:

- Grid Distributed Query Services (GDQSs) that encapsulate the Polar* compiler, and

- Grid Query Evaluation Services (GQESs) that implement the functionality of the Polar* evaluators.

The main differences between the two systems that are relevant to this thesis, are the following:

- OGSA-DQP is not tightly coupled with any persistent storage system, such as SHORE. The result is that OGSA-DQP services are significantly more light-weight than the corresponding Polar* components.
- In OGSA-DQP, metadata are kept in main memory and constructed on a per-session basis, whereas in Polar* metadata are persistent, and thus can be arbitrarily outdated.
- OGSA-DQP employs the GDS generic interface to access remote databases, whereas Polar* relies on manually constructed wrappers.
- In OGSA-DQP, communication between service instances occurs in the form of XML documents transmitted over SOAP/HTTP. Thus, there is no need for MPI messaging, as in Polar*.
- The *operation_calls* in Polar* load local code dynamically, whereas in OGSA-DQP, although they are still conceived of by the compiler as typed *user-defined functions (UDFs)*, they constitute calls to WSs. A consequence of this difference is that they can be executed in any GQESs, contrary to what happens in Polar*.

2.5.3 Query Planning and Evaluation in OGSA-DQP

The single-node optimisation policy in OGSA-DQP is the same as in Polar*, i.e., employing heuristics that minimise the volume of intermediate results and the size of data transmitted over the network, to reduce the query response time. Multi-node optimisation differs in the scheduling policy, to reflect the fact that *operation_calls* can be deployed on any machine. Also, the compiler is modified to output the query plan in XML (see Figure 2.9 for a simple example). Finally, the optimiser has been modified to read metadata from main memory and not to access any repository.

The steps for setting up query sessions and submitting queries in a GS environment are shown in Figure 2.10:

```
<Partition>
 <evaluatorURI>6</evaluatorURI>
 <Operator operatorID="0" operatorType="TABLE_SCAN">
  <tupleType>
   <type>Classification</type> <name>Classification.OID</name>
   <type>string</type> <name>Classification.contologyid</name>
   <type>string</type> <name>Classification.cproteinid</name>
  </tupleType>
  <TABLE_SCAN>
   <dataResourceName> Classifications </dataResourceName>
   <GDSHandle> http://rpc52.cs.man.ac.uk:9090/ogsa/services/ogsadai
               /GridDataServiceFactory </GDSHandle>
   <tableName> Classifications </tableName>
  </TABLE_SCAN>
 </Operator>
 <Operator operatorID="1" operatorType="APPLY">
  <tupleType>
   <type>string</type> <name>Classification.cproteinid</name>
  </tupleType>
  <APPLY>
   <inputOperator> <OperatorID>0</OperatorID></inputOperator>
   <applyOperationType>PROJECT</applyOperationType>
   <parameters>
    <attributeName>Classification.cproteinid</attributeName>
   </parameters>
  </APPLY>
 </Operator>
 <Operator operatorID="2" operatorType="EXCHANGE">
  <tupleType>
   <type>string</type> <name>Classification.cproteinid</name>
  </tupleType>
  <EXCHANGE>
   <inputOperator> <OperatorID>1</OperatorID></inputOperator>
   <consumers>
    <operatorReference>
     <EvaluatorURI>7</EvaluatorURI>
     <OperatorID>0</OperatorID>
    </operatorReference>
    <operatorReference>
     <EvaluatorURI>1</EvaluatorURI>
     <OperatorID>0</OperatorID>
    </operatorReference>
   </consumers>
   <producersNumber>0</producersNumber>
   <producers>
   </producers>
   <arbitratorPolicy>
    <ROUND_ROBIN> 1
    </ROUND_ROBIN>
   </arbitratorPolicy>
  </EXCHANGE>
 </Operator>
</Partition>
```

Figure 2.9: An example plan fragment that retrieves the *Classification* relation from evaluator *6*, projects the *cproteinid* attribute, and sends the resulted tuples to evaluators *1* and *7*. The fragment corresponds to the lower right partition in Figure 2.7(c).

1. Grid Data Services (GDSs) are registered to a Grid Registry. WSs that may play the role of UDFs are registered too.
2. The client starts a session and chooses which registry to contact, and which Grid and Web services registered to this registry to use.
3. The client informs the static query coordinator about the selected services. The coordinator creates a global database schema by collating the local schemata (only naming conflicts are resolved, as data integration issues are out of the scope of this thesis). The WSs are interpreted as typed UDFs, based on their WSDLs. This step concludes the creation of a query session.
4. The client submits a query to the query coordinator. The latter is responsible for parsing the query statement, creating a query plan, optimizing and parallelising it.
5. The static coordinator dynamically creates as many GQESs as the different sites chosen for evaluation.
6. The results are returned to the client.

2.5.4 OGSA-DQP's approach regarding the novel challenges

OGSA-DQP does not differ from Polar* in how it tackles the issues of resource heterogeneity and adaptivity. However, as data and resource metadata are retrieved for each session in OGSA-DQP, there is a higher probability that the metadata is more accurate than in Polar*.

2.6 Other Grid-enabled database query systems

Although the initial focus of Grid data management was on file-based storage and movement [MBM+99], and until the writing of this thesis, Polar* and OGSA-DQP have been the only known fully fledged generic Grid-enabled query processors, there is an increasingly growing interest in adopting database query technologies in a Grid context.

For example, SkyQuery [MSBT03, NSGS+05] applies DQP over Grid-connected databases containing astronomic data. The execution has similarities with OGSA-DQP/Polar*, e.g., calls to WSs are regarded as typed UDFs. The main differences

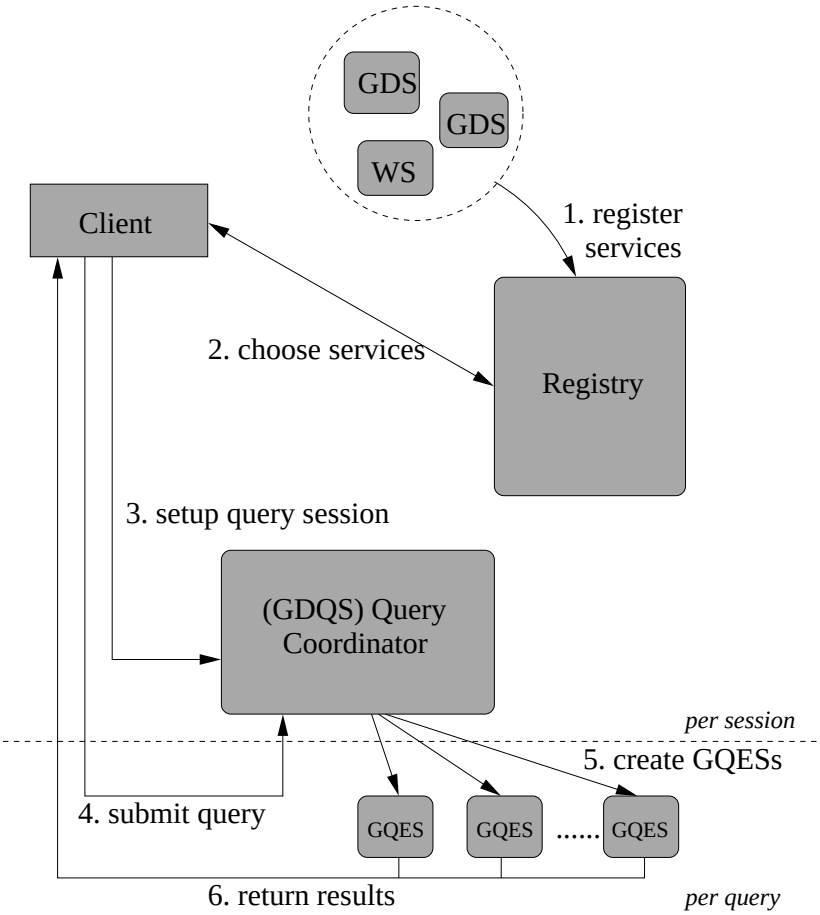


Figure 2.10: The typical steps for setting up a query session (1-3) and evaluating queries (4-6).

is that OGSA-DQP i) accesses Grid rather than Web services, ii) supports partitioned parallelism, iii) can employ Grid machines that may not hold data in order to evaluate parts of the query plan, and iv) is generic with respect to the underlying databases supported and not tailored to a specific scientific scenario.

GridDB-lite [NCK+03] is a project motivated by data-intensive scientific applications on the Grid, built upon DataCutter [BFK+00]. It identifies, like Polar* and OGSA-DQP, the importance of combining data integration with analysis, but employs query technologies for facilitating only the former. The data is stored in flat files, viewed as database tables with the help of specific, manually constructed wrappers. Then, the users express their retrieval tasks as SQL-like queries. However, the query is not evaluated using database technologies. Functionalities that correspond to database scan, predicate evaluation in an operator, and operation call, are implemented not as database operators but as stand-alone independent services. Thus, the benefits

of well-established query optimisations with respect to operator order and implementation algorithm selection cannot be applied. Parallelism is supported but needs to be specified explicitly by the user. Overall, GridDB-lite benefits from the declarative manner of expressing potentially complex tasks in query processing, but develops its own execution mechanisms, thus not exploiting the full potential of a DQP system.

Another project that supports database interfaces for data processed in a workflow is GridDB [LFP03]. Its key feature is that the inputs and outputs of each workflow process are presented to the user as database tables, over which queries can be submitted. As in GridDB-Lite, this enables declarative task expression. However, GridDB takes one step further, and employs a technique devised for adaptive query processors [RRH00] to prioritise partial results.

Generic interfaces to Grid databases have been developed in two projects, OGSA-DAI [OD] and European Datagrid's Spitfire [BBH+02]. In OGSA-DQP, the former has been adopted, as it exposes itself as a GS and provides a uniform access (i.e., wrapping) for a wide range of commercial DBMSs. In addition, it is developed in close association with the emerging standard for database access [DAI].

2.7 Summary

This chapter provided background material about Distributed Query Processing (DQP) and Grid Computations, and discussed their integration. From the perspective of Grid computing, DQP offers an alternative manner to describe and run computations, characterised by ease of expression and system-performed optimisation. From the DQP perspective, the Grid offers a platform that addresses infrastructure concerns regarding security and coordinated discovery and allocation of remote and autonomous resources.

In addition, this chapter identified the issues of generic adaptivity mechanisms covering adaptations to volatile wide-area environments, and resource scheduling, which constitute the research problems addressed in this thesis (Chapter 1), as an important challenge for efficient and effective DQP on the Grid.

The last sections dealt with Grid-enabled database systems, presenting in more detail two existing DQP systems for the Grid, Polar* and OGSA-DQP. The development of these systems is of considerable importance for the remainder of the thesis. As there are no relevant validated simulators and models describing the Grid environment and the behaviour of Grid-enabled query processors yet, the research results are

evaluated in the context of these two real systems. Thus, they form the basis for a testing platform. As stated in Chapter 1 explicitly, the contribution of this thesis to the development of Polar* and OGSA-DQP corresponds to the compiler of the systems.

The integration of Grid and database technologies (Section 2.3) has been discussed also in [GPSF04]. Descriptions of Polar* have appeared in [SGW+03, SGW+02], and of OGSA-DQP in [AMP+03, AMG+04].

Chapter 3

Scheduling Queries in Grids

This chapter presents a novel scheduling algorithm for queries running in distributed heterogeneous environments like the Grid. In wide-area query processing, the resource scheduling problem is defined as the problem of (i) selecting resources and (ii) matching subplans with resources. In the non-database literature, sometimes, scheduling involves the problems of defining the execution order of jobs (which may correspond to subplans in query processing) and exploiting pipelined parallelism in addition to the resource selection and the job matching [DSB+03]. However, in databases, the issues of execution order and pipelined parallelism have been effectively and efficiently addressed by adopting well-established execution models, such as iterators [Gra93], and thus need not be the responsibility of query schedulers.

To date, scheduling algorithms for heterogeneous distributed databases compromise partitioned parallelism (e.g., [ESW78, Kos00]) and consequently are not suitable for a significantly wide range of intensive queries[1], for which parallelism is particularly beneficial. The algorithm proposed addresses this limitation in a practical way with low-complexity cost. It allows queries over Grid databases to employ both partitioned and pipelined parallelism using a diverse set of resources that may or may not store data. This contribution allows one to move a step towards attaining high performance in grid querying as it may improve the performance of query processing significantly.

Resource selection is an integral phase of query optimisation in distributed query processing. Regardless of the plan enumeration approach, e.g., dynamic programming, two-step optimisation, and so on, the query optimiser needs to decide which resource

[1]Intensive queries are the queries that either process large volumes of data, or apply expensive computations onto the data processed, and consequently place a heavy demand on any of the CPU, I/O bandwidth and network bandwidth types of resources (or their combinations).

will evaluate each part of the query. In architectures similar to the ones of Polar* and OGSA-DQP, there is a dedicated component in the query optimiser for this purpose, namely the scheduler, as described in Chapter 2. The new algorithm is implemented by re-engineering this component.

The chapter is organised as follows. The discussion of related work appears in Section 3.1. The proposed solution is described in Section 3.2. This solution is evaluated in Section 3.3. Section 3.4 summarises the chapter.

3.1 Related Work

Existing scheduling algorithms and techniques, either from the database or the Grid or the parallel computing research communities, are inadequate for parallel query processing on the Grid basically because the way they select machines and allocate tasks compromises partitioned parallelism in a heterogeneous environment. For example, generic DAG schedulers (e.g., [KA99, TWML02, SZ04]), and their Grid variants (e.g., [TTL03]) tend to allocate a graph vertex to a single machine, which corresponds to no partitioned parallelism (if the DAG represents a query plan, a vertex corresponds to a query operator). More comprehensive proposals (e.g., GrADS [DSB+03]) still rely on application-dependent *"mappers"* to map data and tasks to resources, and thus come short of constituting complete scheduling algorithms on their own right. In addition, no mapper has been proposed that covers query processing on the Grid. Efficient proposals for mixed-parallelism scheduling (e.g., [RvG01]) and parallel database scheduling (e.g. [ESW78, DGG+86]), are restricted to homogeneous settings. Contrary to the above approaches, the proposal of this thesis effectively addresses the resource scheduling problem for Grid databases in its entirety, allowing for arbitrarily high degrees of partitioned parallelism across heterogeneous machines.

This section presents additional scheduling proposals found in the literature and discusses their differences with the algorithm proposed. It is divided into two parts, one examining contributions coming from the database community and thus tailored to database queries, and the other referring to schedulers for more generic computations in Grids.

3.1.1 Resource Allocation in Database Queries

Distributed query processing has mostly been influenced by some pioneering systems. The most influential, R* [ML86], simplified the problem of resource scheduling by neglecting the benefits of partitioned parallelism. The data is retrieved from a single site only, and are joined on a single site, which is either the site of one of the inputs or the site that asked for the data. SDD-1 [BGW+81] focused on *semijoins* and also did not employ partitioned parallelism. Distributed Ingres [ESW78] took a step forward, and provided for partitioned parallelism, but only for machines that store data. Their scheduling algorithm assumed that all the participating machines have the same computational capabilities (a property that, in the general case, cannot be expected to hold on the Grid). It also forced a choice to use either all nodes available or just one. [RM95] discusses an approach for load balancing employing partitioned parallelism, and although it refers to completely homogeneous environments, it does not force the system to employ all the available nodes when these are not needed, extending the work of [WFA92]. Other existing techniques for parallel and distributed databases do not consider partitioned parallelism or completely skip the resource selection phase by assuming a fixed set of resources and then trying to schedule tasks over these resources (e.g., [GI97, MBGS03]). Moreover, they usually assume homogeneous and stable environments (e.g., [WFA92, WCwHK04]). For example, Gamma [DGG+86] assumes that all the available machines are similar in terms of capabilities, connection speed and ownership.

In early commercial distributed database systems, the reason for not developing schedulers supporting partitioned parallelism was that the communication cost was the predominant cost. The biggest effort was put into minimising this cost [TTC+90], and each operator was executed at a single site. In modern applications, and due to advances in network technologies, the computation cost is often the dominating cost (e.g., [SA02]). However, advanced distributed query processing systems may employ only pipelined and independent parallelism (e.g., [ROH99, SAL+96]). A question may arise as to whether existing techniques that do not employ partitioned parallelism (like, for example, the *query/data/hybrid shipping* in client-server architectures [Kos00]) are efficient in most cases. For a large range of queries, the answer is negative. The parallelisation saturation point (i.e. the point after which further parallelisation does not yield any benefit) is lower when the start-up cost increases. The start-up cost depends strongly on the machine and the evaluation method, but in most cases it occurs once for all the operators processed on a single machine. Therefore, the contribution

of the startup cost becomes less significant for expensive queries. When the ratio between workload and machines used increases, the saturation point increases as well. These properties are quite appealing for tasks in computational grids insofar as they are envisaged to be of significant size and complexity.

3.1.2 Generic Resource Scheduling on the Grid

In the area of DAG scheduling to support mixed parallelism, there have been many interesting proposals. The algorithm proposed may be regarded as an adaptation of the task allocation in [RNvGJ01, RvG01] for heterogeneous environments, tailored to the needs of query processing. With respect to heterogeneity, [BDS03] takes one step further, by considering heterogeneous clusters of homogeneous machines, but has significantly higher complexity and is not compatible with the iterator model of query execution [Gra93]. Ordinary DAG schedulers for Grid environments, such as [SCS+04, SZ04, TTL03], do not consider partitioned parallelism, nor the type of inter-dependencies between operators in a query execution plan (e.g., the order of operators in a query plan may be fixed and some operators need to be computed earlier than others, whereas other operator sets may be able to execute in a pipeline). Not considering partitioned parallelism is the usual case even for homogeneous systems [KA99].

The proposal in Section 3.2 is, to a certain extent, compatible with the GrADS project [DSB+03], as it can play the role of the GrADS mapper for database query applications. The main differences lie in the fact that, in query processing, the initial pruning of the resource sets cannot be done in a completely application-independent way, as in GrADS, because database locality is very important for efficient query plan construction. Also, in GrADS, the scheduling algorithm runs many times, one for each candidate machine group, whereas, in the proposal of this thesis, the scheduler is executed only once, resulting in lower algorithm execution times. In the context of [DSB+03], [YD02] has presented a heuristic that allocates nodes in an incremental way similar to the scheduler proposed. One difference between the two techniques, apart from the fact that the graphs in [YD02] are simpler than typical query plans, is that in [YD02] the machines are first chosen and then it is decided how to use them, whereas in the approach followed in this thesis, a bottleneck is first identified and then an attempt is made to increase the parallelism. Other schedulers developed in GrADS (e.g., [PBD+01]) do not provide for different degrees of parallelism in different parts of the query plan, and may require the user to prune first, explicitly, an extended set of resources. In the current proposal, this is a responsibility of the scheduler.

Other powerful schedulers, such as Condor [TWML02, TTL03], suffer mainly from the limitations in the dependencies in the graphs supported, and the restriction of allocating each node in the graph to a single site [KA99].

3.2 An algorithm for scheduling queries in heterogeneous environments

3.2.1 Problem Definition

It has been demonstrated that parallelising a subplan over the complete set of available resources may cause significant performance degradations, and harms the efficiency of resource utilisation [WFA92]. In homogeneous systems, for which each machine has the same processing capacity, the scheduling problem is simplified to the problem of finding the optimal degree of parallelism for each subplan. However, the problem of resource scheduling on the Grid is actually more complicated than choosing the correct degree of parallelism. Grid schedulers should decide not only how many machines should be used in total, but *exactly which these machines are*, and which parts of the query plan each machine is allocated[2].

The three dimensions (i.e., how many, which and for which part of the query) cannot be separated from each other to simplify the algorithm in a divide-and-conquer fashion. For example, it is meaningless to determine the number of selected nodes from a heterogeneous pool without specifying these machines; this is in contrast to what can be done in homogeneous systems since in a heterogeneous setting each machine may have different capabilities. Another important aspect is the efficiency of parallelisation, which indicates how efficiently the resources have been used (a more specific definition is given in Section 3.3.4). The parallelisation efficiency is of significant importance especially when the available machines belong to multiple administrative domains and/or are not provided for free. Thus, the aim is, on one hand to provide a scheduler that enables partitioned parallelism in heterogeneous environments with potentially unlimited resources, and on the other hand to keep a balance between performance and efficient resource utilisation. As the problem is theoretically intractable [KA99], effective and efficient heuristics need to be employed.

[2]The related and common problem of devising optimal workload distribution among the selected machines is orthogonal to the scheduling problem and will not be discussed here.

3.2.2 Solution Approach

Existing schedulers (mostly from the parallel database community) that allow for intra-operator parallelism do not deal with resource selection and try to tackle the problem of how to use the complete set of preselected resources (e.g., [GI96, GI97, WFA95, CYW96, LCRY93, HCY94, HCY97, MWK98, HM95]). In addition, some of these schedulers are able to prune such sets of resources rather than using all the available ones (e.g., [RM95]), although they typically make the assumption that all the resources available are similar (e.g., [WFA92, HS91, Hon92, Has96, Liu97a, Liu97b]). As, in a Grid setting, a scheduler also needs to decide which particular resources to use, because of the machine heterogeneity, such solutions are not practical. On the other hand, an exhaustive search for all the possible combinations of machines and query subplans is an obviously inefficient solution. Thus, employing a heuristic, the requirement is to provide a solution that can scale well with the number of machines that are available; the algorithm proposed here meets this requirement.

Algorithm 1 High level description of the scheduling algorithm.

```
Phase1 : perform initial allocation
Phase2 :
repeat
  get costliest parallelisable operator
  define the criteria for machine selection
  repeat
    get the set of available machines
    check if more parallelism is beneficial
  until no changes in the degree of parallelism of the costliest operator
until no changes in which operator is the costliest
```

The algorithm receives a query plan which is partitioned into subplans and each of the operators of the query plan is scheduled on one machine. After this initial resource allocation, which is at the lowest possible degree of partitioned parallelism, it enters a loop (see Algorithm 1). In that loop, the algorithm takes the most costly operator that can be parallelised, and defines which criteria should be used for selecting machines for this operator. Following this step, the algorithm increases the operator's degree of parallelism if that increase improves the performance of the query plan above a certain threshold, in line with [RNvGJ01, RvG01]. When no more changes can be made for that operator, the algorithm re-estimates the cost of the plan and the operators in order to do the same for the new most costly operator. The loop exits when no changes in the parallelism of the most costly operator can be made. The threshold is of considerable

importance. In principle, the smaller the threshold, the closer the final point is expected to be to the optimal point[3]. However, this comes at the expense of higher compilation time. A bigger threshold may force the algorithm to terminate faster, but also to stop and return a number of nodes, which yields a final response time that can be improved on, although it can still be much smaller than the initial.

From a higher level point of view the algorithm transforms an existing plan to a more efficient one at each step, by modifying the set of resources allocated to a part of the query plan. Transformational approaches to query optimisation have already been employed for constructing query plans (e.g., simulated annealing and iterative improvement techniques [Ioa96]) but not for scheduling resources [CHS99]. The algorithm assumes that a two-step optimisation approach is followed, as the single-node query plan needs to have already been constructed.

To estimate the costs, the scheduler requires a cost model which

- assigns a cost to a parallel query plan, and
- assigns a cost to every physical operator of that query plan.

Any such cost model is suitable, as the scheduling algorithm is not based on any particular one, following the approach of [DBC03, DSB+03]. By decoupling the cost model and the scheduling algorithm, enhancements in both these parts can be developed and deployed independently. The cost model is also responsible for defining the cost metric, with query completion time being a typical choice. The importance of cost models is significant, given that heuristic-based optimisations can err substantially in distributed heterogeneous environments [ROH99]. In this work, a variant of the cost model developed in [SSWP04] is used.

3.2.3 The Input of the Algorithm

The inputs to the algorithm are:

- A partitioned single-node optimised plan, with *exchanges* placed before *attribute sensitive* operators (e.g., joins) and *operation_calls*. *Attribute sensitive operators* are those that, when partitioned parallelism is applied, the data partitioning among the operator clones depends on values of specific attributes of the data.

[3]The convergence to the optimal point depends additionally (i) on the accuracy of cost estimates, and (ii) on the absence of local minima in the performance for different degrees of parallelism.

- A set of candidate machines. For each node, certain characteristics need to be available. The complete set of these characteristics depends on the cost model and its ability to handle them. However, a minimum set that is required by the algorithm consists of the available CPU power, the available memory, the I/O speed, the connection speed, and locality information with regard to the data and external functions employed in the query (i.e., which database tables and external programs are local to this resource). Such metadata can be provided by MDS [CFFK01] and NWS [WSH99], which is the typical approach for many Grid tools (e.g., [DSB+03]).

- A threshold a, referring to the improvement in performance. This improvement is caused by transformations to the query plan. The improvement ratio is given by $\frac{t_{old} - t_{new}}{t_{old}}$, where t_{old} and t_{new} are the time costs before and after the transformation, respectively. The cost model is responsible for computing these costs. The partitioned parallelism is increased only when the improvement ratio is equal to or greater than the threshold.

3.2.4 Detailed Description of the Algorithm's Steps

3.2.4.1 Notation

A query plan $QueryPlan$ is represented as a directed acyclic graph $G = (V, E)$, where V is a set of n operators, and E is the set of edges. M is the set of the m available machines. Each machine M_i is described by the vector:

$$\{CPU_i, Mem_i, I/O_i, ConSpeed_i, tables_i, programs_i\},$$

where CPU_i, Mem_i, I/O_i and $ConSpeed_i$ are the available CPU power, available memory, disk bandwidth and average connection speed of the ith machine respectively (to keep the presentation simple, it is assumed here that all connections from the ith machine are of the same speed). $tables_i$ and $programs_i$ are the lists of database tables participating in the query and programs called externally by the query that reside on that machine, respectively. M_{table} is the set of machines that can evaluate a *scan(table,...)* operator, whereas $M_{program}$ is the set of machines that can evaluate an *operation_call(...,program,...)*. Without loss of generality, an assumption is made that the cost metric is time units. $TimeCost(G)$ (or $TimeCost(QueryPlan)$) is the time cost of the plan, while $TimeCost(V_i)$ is the time cost of operator V_i.

Algorithm 2 The first step of the scheduling algorithm after the initial resource allocation in detail.

repeat
STEP 1: Estimate the cost of query and operators
$n = NumberOfOperators$
$m = NumberOfMachines$
$a = threshold$
for $i = 1$ *to* n **do**
$estimate\ TimeCost(V_i)$
end for
$estimate\ TimeCost(QueryPlan)$
$list\ V_{sorted} = SORT(V)\ on\ TimeCost(V_i)$
//if true, the costliest operator has been allocated more machines
$bool\ changes\_made = false$
RUN STEP 2: Examine the costliest operator - see Algorithm 3
until $!changes\_made$

3.2.4.2 Algorithm description

The algorithm consists of two phases. In the first phase, a query plan without partitioned parallelism is constructed. The resource allocation in this phase is mostly driven by data locality. For example, the *scans* are placed where the relevant data reside and the *joins* are placed on the node of the larger input, unless more memory is required [RM95]. As there already exists a significant number of proposals for resource scheduling without partitioned parallelism (e.g., [ML86]), this phase is not covered in detail.

In the second phase, which is the main contribution of this work and was summarised in Algorithm 1, there are two basic steps (Algorithms 2 and 3), and a third one for exiting (Algorithm 4). In the first step, the cost model evaluates the cost of the query plan and its individual operators. The operators are sorted by their cost. The second step is the step where the partitioned parallelism can be increased. The most costly operator that has a non-empty set of candidate machines M' is selected. The set of candidate machines consists of the nodes that (i) are capable of evaluating the relevant operator, (ii) have not yet been assigned to that operator, and (iii) have either been started up, or have a start-up cost (SUC) that permits performance improvement larger than the relevant thresholds (see the first *repeat* loop in Algorithm 3). For this operator the *check_more_parallelism* function (Algorithm 5) is repeatedly called until the query plan cannot be modified any more. Each call can increase the partitioned

Algorithm 3 The second step of the scheduling algorithm after the initial resource allocation in detail.

STEP 2: Examine the costliest operator
$i = 0$
repeat
  $i++$
  if $V_{sorted}[i] = SCAN(table, ...)$ **then**
    //define the candidate nodes for each kind of operator
    $M^{'} = M_{table}$
  else if $V_{sorted}[i] = OPERATION\_CALL(..., program, ...)$ **then**
    $M^{'} = M_{program}$
  else
    $M^{'} = M$
  end if
  //remove already allocated nodes
  $M^{'} = M^{'} - (V_{sorted}[i] \rightarrow machines)$
  for $j = 1$ *to* $LengthOf(M^{'})$ **do**
    if $(SUC_j \geq (1 - a) \times TimeCost(V_{sorted}[i]))$
    $OR\ (SUC_j \geq (1 - a) \times TimeCost(QueryPlan))$ **then**
      //do not consider machines with relatively high startup cost
      $M^{'} = M^{'} - M_j^{'}$
    end if
  end for
until $(M^{'}\ not\ empty)\ \ OR\ (i = n)$
//Parallelise the most expensive parallelisable operator, which is $V_{sorted}[i]$
if $V_{sorted}[i] = Communication - Bounded$ **then**
  $L[1] = ConSpeed,\ L[2] = ConSpeed \times CPU,\ L[3] = CPU$
else if $V_{sorted}[i] = I/O - Bounded$ **then**
  $L[1] = I/O \times ConSpeed,\ L[2] = CPU,\ L[3] = Mem$
else if $V_{sorted}[i] = Memory - Bounded$ **then**
  $L[1] = Mem,\ L[2] = CPU$
else if $V_{sorted}[i] = CPU - Bounded$ **then**
  $L[1] = CPU,\ L[2] = ConSpeed \times CPU,\ L[3] = ConSpeed$
end if
repeat
  $local\_changes\_made = check\_more\_parallelism(L, M^{'}, V_{sorted}[i])$
  $changes\_made = changes\_made\ OR\ local\_changes\_made$
  if $local\_changes\_made$ **then**
    //update the set of candidate machines
    $M^{'} = M^{'} - (V_{sorted}[i] \rightarrow machines)$
  end if
until $!local\_changes\_made$

Algorithm 4 The exiting step of the scheduling algorithm.

STEP 3: Exit
for $i = 1$ *to* n **do**
  if $V_i = EXCHANGE$ && $LengthOf(V_i \rightarrow machines) = 1$ **then**
    $parent(child(V_i)) = parent(V_i),\ child(parent(V_i)) = child(V_i)$
  end if
end for
EXIT

Algorithm 5 The basic auxiliary function, which is responsible for checking greater degrees of parallelism.

This function returns $true$ **if allocating a new machine**
to operator v **is beneficial.**
bool $check\_more\_parallelism(criteria\_list\ L,$
$machine\_set\ M^{'},\ operator\ v)\{$
$bool\ result = false$
// the loop runs until all the criteria are checked
// or an addition of a machine improves the query plan adequately
while L **do**
  $machine\ i = choose\_according\_to(L, M^{'})$
  if $(ImprovementRatio(v) \geq a)$ **then**
    if $(ImprovementRatio(QueryPlan) \geq a)$ **then**
      // $v \rightarrow machines$ holds the set of machines allocated to v
      $v \rightarrow machines = v \rightarrow machines \rightarrow add(i)$
      update scheduling of other operators affected
      $return\ true$
    else
      // the first choice criterion is removed
      $L = L - L[1]$
    end if
  end if
end while
$return\ result$
$\}$

Algorithm 6 The auxiliary functions, which are responsible for machine selection and conflict resolution

This function is called by choose_according_to. It checks whether machine M is better than the machine $temp$ in terms of the choice criteria for machines of list L.

$machine\ check\_properties(list\ L,\ machine\ M_i^{'},\ machine\ temp)\{$
$machine\ result = temp$
$int\ k = 1$
while $L[k]$ **do**
  if $F(M, L[k]) > F(result, L[k])$ **then**
    $return\ M$
  else if $(F(M, L[k]) = F(result, L[k]))\ \ AND\ \ (L[k+1])$ **then**
    $k = k + 1$ //go to the next criterion in the choice list
  else
    $return\ result$
  end if
end while
$return\ result$
$\}$

This function returns the machine from the set of machines $M^{'}$ that satisfies the choice criteria that are in the list L

$machine\ choose\_according\_to(criteria\_list\ L,\ set\ M^{'})\{$
$machine\ result = M_1^{'}$
for $i = 2\ to\ LengthOf(M^{'})$ **do**
  $result = check\_properties(M_i^{'}, L, result)$
end for
$return\ result$
$\}$

parallelism by one degree at most, as only one machine can be added to an operator at a time.

check_more_parallelism (Algorithm 5) is the main auxiliary function that checks whether the addition of one machine for a specific operator in the query plan is profitable. Such an addition affects all the operators between the closest *exchange* higher and the closest *exchange* lower in the plan. The list of choice criteria for machines, which is one of the function's parameters, defines which machine is checked first, with the machine selection and conflict resolution being done in the *choose_according_to* function (Algorithm 6). L symbolises the list of choice criteria for machines. The criteria can either be in the form of the standard machine properties (e.g., available CPU) or combinations of them. The function F (Algorithm 6) is used to quantify properties of machines, and returns the absolute value of a criterion when applied on a specific node. For example, if $L = [mem, CPU \times ConSpeed]$, this corresponds to two criteria. The first is the available memory, and the second is the product of the CPU speed with the available connection speed. As such, in this example, $F(M_i, L[1])$ returns the available memory of the ith machine as an absolute value, and $F(M_i, L[2])$ returns the product of the values of the CPU and connection speeds of the ith machine. The function F is defined as follows:

$$
\begin{array}{ll}
a, b \,\epsilon\, (CPU,\ Mem,\ I/O,\ ConSpeed),\ op \,\epsilon\, (+, -, *, /) & \\
L[x]\ :\ a & \{F(M_y, L[x]) = M_y[a]\} \\
\quad \mid a\ op\ b & \{F(M_y, L[x]) = M_y[a]\ op\ M_y[b]\}
\end{array}
$$

The next command in *check_more_parallelism*, i.e., after the machine to be checked has been selected, involves the evaluation of the achieved improvement ratios with the help of the cost model. Two improvement ratios are computed: one that refers to a specific operator and another that refers to the whole plan. The decisions are based essentially only on the evaluation of the second one, as the aim is to make the query plan run faster, and not simply to speed up specific operators. The reason the first one is computed is that it is less complex than the second, and if a modification does not satisfy the first condition, there is no reason to evaluate the cost of the complete plan for the second improvement ratio. If the improvement ratios are above the threshold, then the query plan is modified accordingly. Otherwise, the function iterates after removing the first element of the list of choice criteria for machines.

A prerequisite for the evaluation of the improvement ratios, and of the cost of operators in general, is to define a data distribution policy, i.e., to define the amount of data each operator instance will evaluate. Finding an optimal data distribution policy is

NP-hard [WC96]; however, a simple and efficient approach for heterogeneous settings is to allocate to each machine an amount of tuples that is proportional to its available CPU power, if is assumed that the CPU cost is the cost that dominates according to the cost model, and proportional to its connection speed, if the communication cost is the higher cost.

Step 3 of the scheduling algorithm evaluates the exit function. If Step 2 (Algorithm 3) has not resulted in any changes in the query plan, then the algorithm checks the scheduling of *exchanges*, deletes them if necessary, and exits (Algorithm 4). Otherwise, the algorithm goes back to step 1, to evaluate again the cost of the plan and their operators, and subsequently to try to parallelise the operator that has become the most costly after the changes. The scheduling of *exchanges* is checked in order to delete the ones that receive data from and send data to the same single node.

The algorithm is independent of the physical implementation of the operators that can differ between database systems. As shown in Step 2 (Algorithm 3), the machines with high disk I/O rates are preferred for retrieving data, the machines with high connection speeds are preferred when the query cost is network-bound, the machines with large available memory are chosen for non-CPU intensive tasks, like *unnests*, and the machines with high CPU capacity are selected for the rest, CPU-intensive operations.

3.2.5 Algorithm's Complexity

The algorithm comprises two loops. Steps 1 and 2 (Algorithms 2 and 3) are repeated until the exit condition is met. Also, step 2 runs a second loop until a local exit condition is satisfied. The outer loop can be repeated up to n times, where n is the number of physical operators in the query plan. The inner loop can be repeated up to m times, where m is the number of available machines. So, the worst-case complexity of the algorithm is of $O(n \times m)$, which makes it suitable for complex queries and when the set of available machines is large.

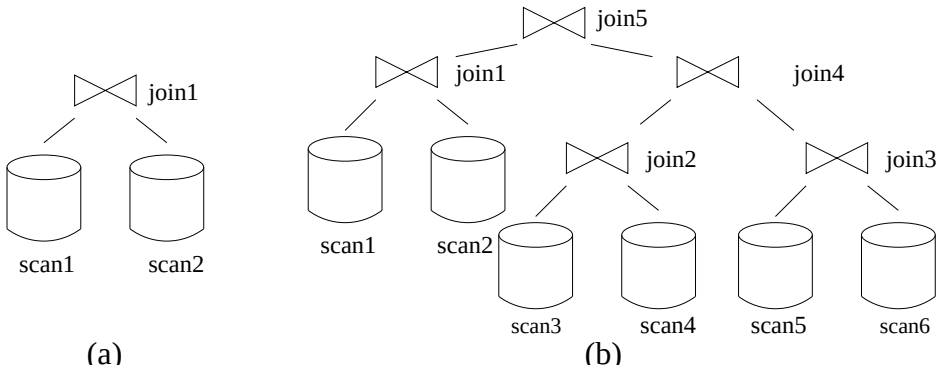


Figure 3.1: The two queries used in the evaluation.

3.3 Evaluation

This section presents the evaluation of the scheduler proposed against existing and other common-sense techniques from distributed databases that do not employ, or employ only limited partitioned parallelism, and against techniques from parallel databases that use all the available nodes. The aim is to compare the efficiency of the proposal for resource selection and allocation to subplans by taking into account both the performance in time units and the parallelisation efficiency. According to the results of the evaluation, the scheduling proposal can significantly improve the performance of distributed queries evaluated in heterogeneous environments.

3.3.1 Evaluation Approach and Settings

The evaluation of the proposed scheduler has been based on simulation; the simulator was built by extending the Grid-enabled query compiler in Polar* [SGW+03, AMP+03] with a simplified version of the cost model in [SPSW02] (the simplified version is presented in Appendix A). The cost model is used by the query compiler to estimate query response time for different schedules.

Two queries, with one and five joins, respectively, are used for the evaluation (Figure 3.1). These queries retrieve data from two and six remote tables, respectively. Each table contains 100,000 tuples. Two datasets are used. In the first one, *setA*, the average size of a tuple is 100 bytes, whereas in the second, *setB*, it is 1Kbyte. All the joins are on a key-foreign key condition, and produce 100,000 tuples. The joins are implemented by single-pass *hash joins*, which are CPU-bound (i.e., their cost depends on the CPU power of the evaluating machine). Also, there are no replicas, so the *scans* cannot be parallelised.

The initial machine characteristics are set as follows: those machines that hold data (2 in the first query and 6 in the second) are able to retrieve data from their store at a rate of 1MB/sec. The average time, over all machines participating in the experiment, to join two tuples is 30 microseconds. On average, data is transmitted over the network at a connection speed of 600KB/sec. It is assumed (i) that the available memory on each machine is enough to hold the hash table if this machine is assigned a *hash join*, and (ii) that the connection speed is a property of the data sender only and not of the pair sender-receiver. The machine start-up cost, which includes the initialisation of a pre-existing query evaluation engine and the submission of the query subplan, is 1 second. The parameters above form the input to the cost model adopted by the system, and are realistic, according to the experience gained from the OGSA-DQP Grid-enabled query processor [AMP+03].

For such configurations and datasets, the two queries are CPU intensive. That is, when they are applied to the first dataset, and due to the size of the joins, the computation cost is greater than the communication cost and the disk I/O cost by an order of magnitude without partitioned parallelism. If the second dataset is used, the I/O cost is still smaller, but the communication cost, although smaller before the increase of the partitioned parallelism, contributes significantly to the final cost.

3.3.1.1 Workload Distribution

The parallel execution of operators is always *workload-balanced*, in the sense that each machine chosen to evaluate a portion of a *hash join* is allocated a number of tuples that is inversely proportional to its *hash join* evaluation speed. For example, if 1000 tuples are to be distributed across two machines for a *hash-join*, and these machines evaluate the *hash-join* in 15 and 60 microseconds, respectively, then the machines will be allocated 800 and 200 tuples, respectively. In a few cases in the following sections where a different workload distribution is assumed, this is explicitly stated. As the computation cost is the dominant cost, this workload distribution policy ensures high performance, without adding much complexity.

3.3.2 Performance Improvements

In this experiment, the two example queries are evaluated when the number of extra nodes (i.e., the machines that do not store any base data) varies between 0 and 20. From the extra machines, 25% have double the CPU power and connection speed

of the average (i.e., they evaluate a join between two tuples in 15 microseconds and transmit data at 1.2MB/sec), 25% have double CPU power and half connection speed (i.e., they evaluate a join between two tuples in 15 microseconds and transmit data at 300KB/sec), 25% have half CPU power and double connection speed (i.e., they evaluate a join between two tuples in 60 microseconds and transmit data at 1.2MB/sec), and 25% have half CPU power and connection speed (i.e., they evaluate a join between two tuples in 60 microseconds and transmit data at 300KB/sec). Two configurations of the proposal are compared, one with lower improvement ratio threshold and one with higher, against six other simpler approaches:

1. using all the available nodes in the way that parallel systems tend to do, taking into consideration their heterogeneous capabilities for workload balancing though (in the way described in Section 3.3.1.1), and not considering them as being the same (i.e., using all the available nodes without applying workload balancing);

2. not employing partitioned parallelism and placing the *hash joins* on the site where the larger input resides in order to save communication cost;

3. employing limited partitioned parallelism and placing the *hash joins* on the nodes that already hold data participating in the join, i.e., not employing nodes that have not been used for scanning data from store;

4. using only the two most powerful from the extra machines to parallelise all the query operators;

5. using all machines evenly, which is the only case where the number of tuples assigned to each machine is equal (i.e., the workload is not inversely proportional to the *hash join* evaluation speed as in the first approach); and

6. applying the first approach only to the most expensive operator and not to the complete query plan (this applies only to the multi-join query).

Figures 3.2-3.5 show the results when the two queries are applied to the two datasets (the thresholds are different in the two queries). The dashed lines in the charts depict the non parallelisable cost of the queries. For these queries, the non parallelisable cost is the cost to retrieve data from the non-replicated stores, which is not affected by the different degrees of the join parallelism. The numbers above the bars representing the performance of the algorithm proposed in this chapter, show how many machines are

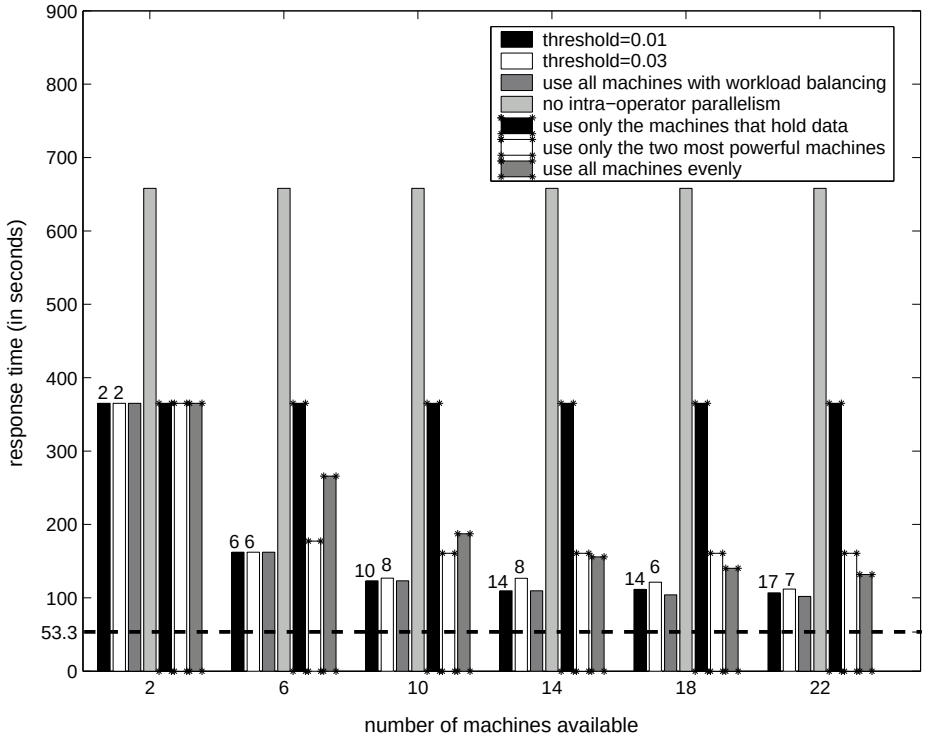


Figure 3.2: Comparison of different scheduling policies for the 1-join query for setA

chosen. In all the other cases, the number of machines used is a result of the approach applied.

The scheduling approach described in the thesis is tunable, and better execution times can be achieved by using a smaller threshold. However, this benefit comes at the expense of employing more machines. From the figures, the following conclusions can be drawn:

1. the proposed scheduler manages to reduce the parallelisable cost; compare, for example, the difference of the algorithm proposed (first two bars in each bar set) from the dashed line, with the difference of the performance of the system with no intra-operator parallelism (fourth bar in each bar set) from this line;

2. techniques with no, or limited, partitioned parallelism (e.g., employing only the machines that store the databases, parallelising only the most costly operator) yield significantly worse performance than the algorithm described;

3. policies that use only a small set of powerful nodes or do not try to perform workload balancing (i.e., the workload is not distributed according to the machine

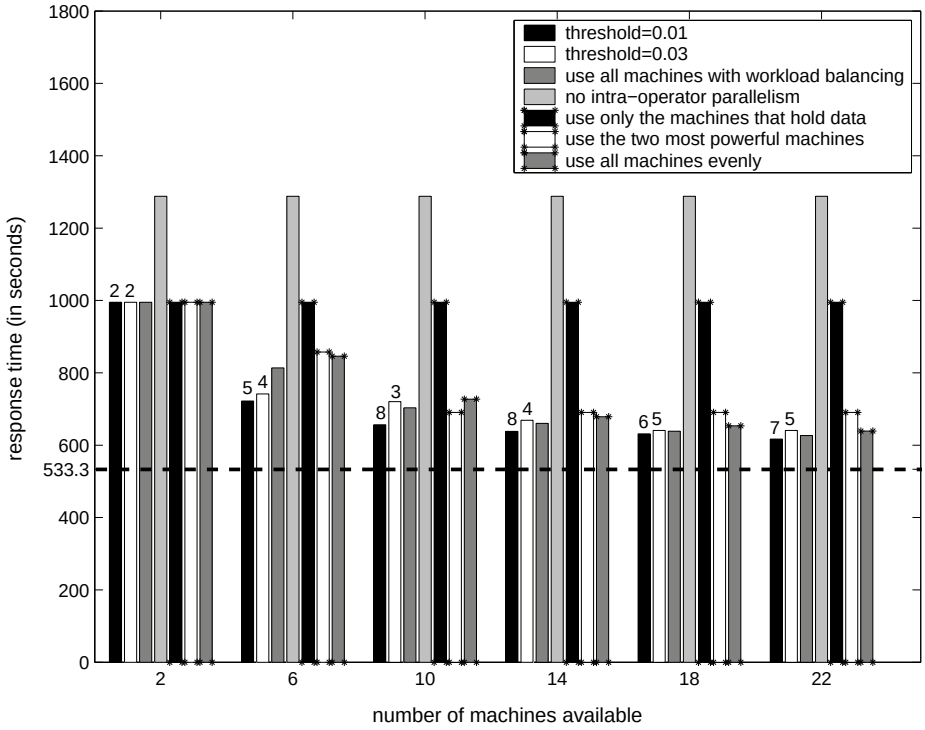


Figure 3.3: Comparison of different scheduling policies for the 1-join query for setB

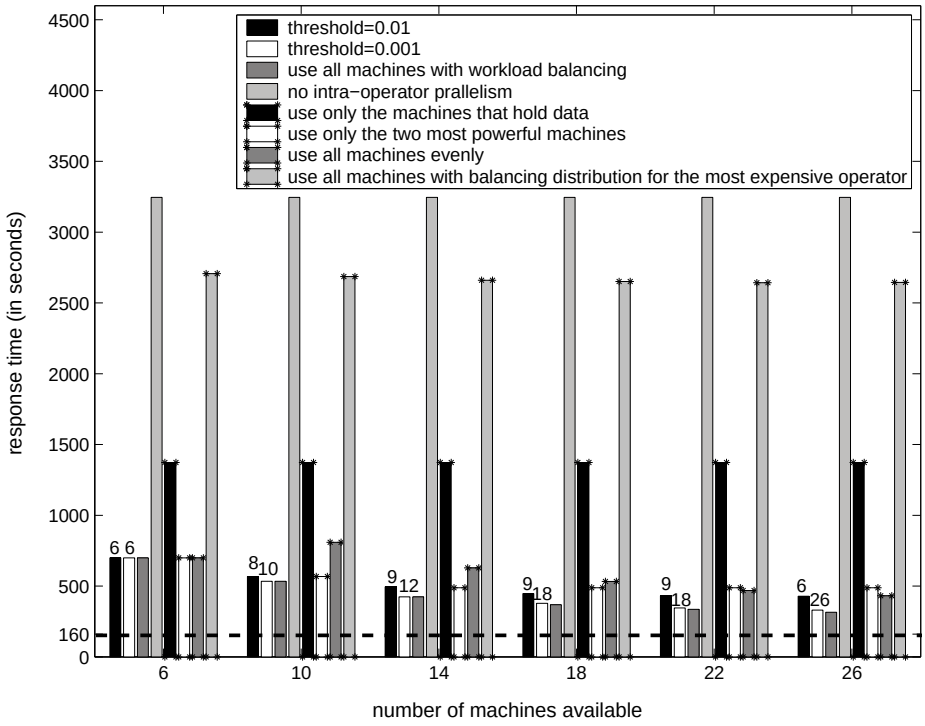


Figure 3.4: Comparison of different scheduling policies for the 5-join query for setA

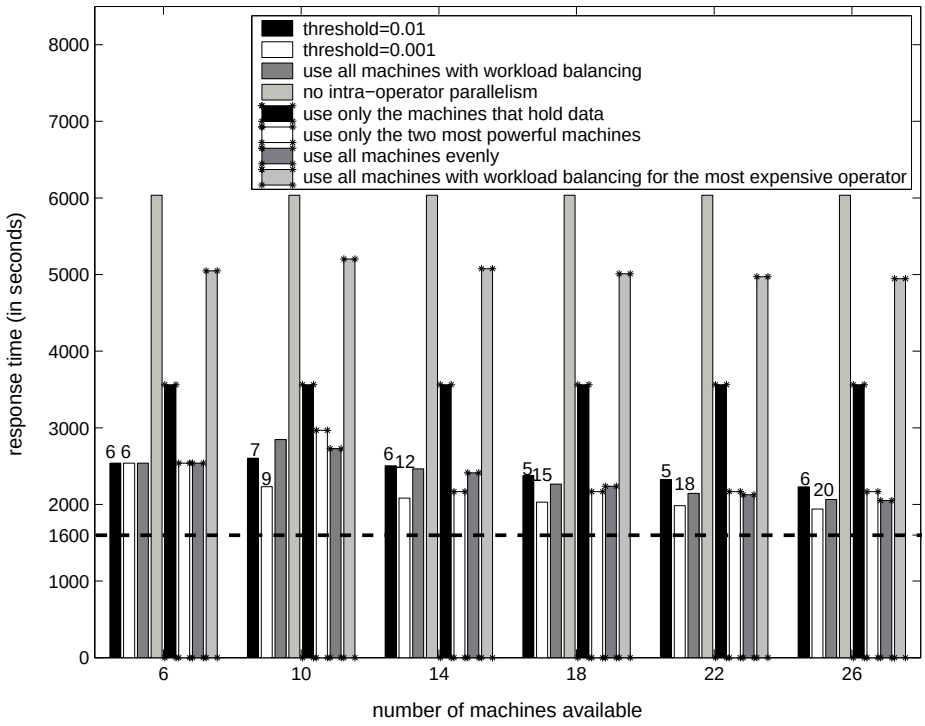


Figure 3.5: Comparison of different scheduling policies for the 5-join query for setB

capabilities) are also clearly outperformed by the scheduler proposed, provided that the threshold is not relatively high for complex queries;

4. relatively high thresholds can operate well when the cost of the query is concentrated on a single point in the query plan (Fig. 3.2 and 3.3);

5. as expected, using all nodes and applying workload balancing can operate very well for specific machine configurations, provided that it is easy to calculate the appropriate workload distribution as in the queries over *setA*. However, such a policy has severe drawbacks which are presented later.

Table 3.1 shows that the time to execute the proposed scheduling algorithm for each of the two queries and thresholds is reasonable, and negligible compared to the query completion time cost.

These queries are essentially CPU-bound. Nevertheless, the behaviour of the algorithm is similar for network and disk I/O-bound queries in presence of replicas. In network-bound queries, the emphasis is placed on parallelising the *exchange* operators, which are responsible for data communication, whereas, in I/O-bound queries, the performance is improved mostly when the operators that retrieve data from store

Number of joins	Threshold	Scheduler Cost
1	0.01	0.0106 secs
1	0.03	0.0055 secs
5	0.001	0.3138 secs
5	0.01	0.2083 secs

Table 3.1: The time cost of the scheduling algorithm for 20 extra nodes.

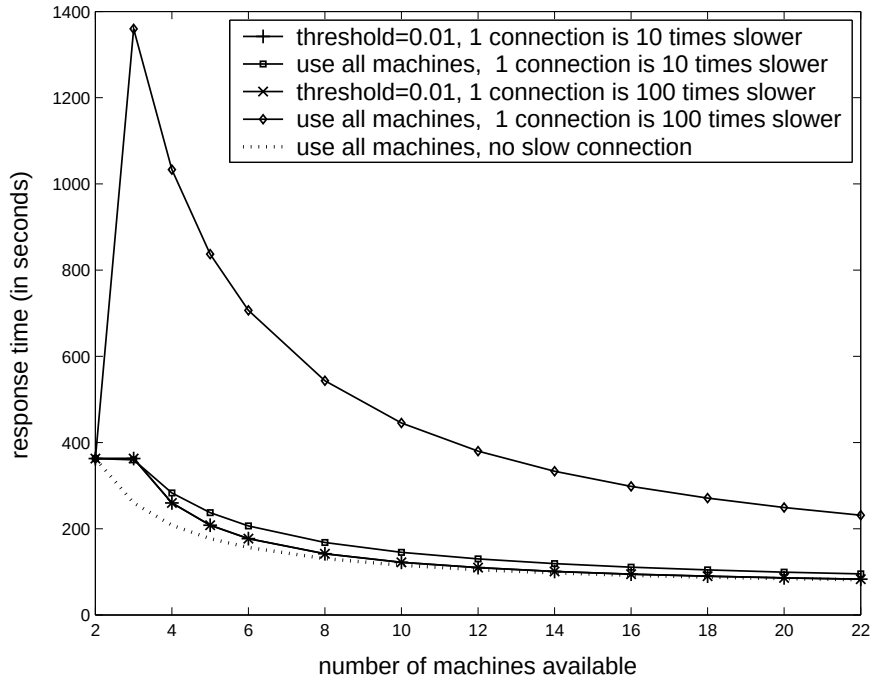


Figure 3.6: Comparison of different scheduling policies in the presence of a slow connection for the single-join query (note that the 1st and 3rd line types essentially overlap).

are partitioned across different machines. In both cases, the scheduling proposal is capable of identifying the performance bottleneck, as it did for the CPU-bound queries in the experiments above.

3.3.3 Performance Degradation in Presence of Slow Connections

In Figures 3.6 and 3.7 the approach proposed is compared with the approach of employing all the nodes when the two queries, which have one and five joins, respectively, run over *setA*, and there is just one machine with a slow connection. Two cases are considered: in the first, the slow connection is ten times slower than the average (i.e., 60

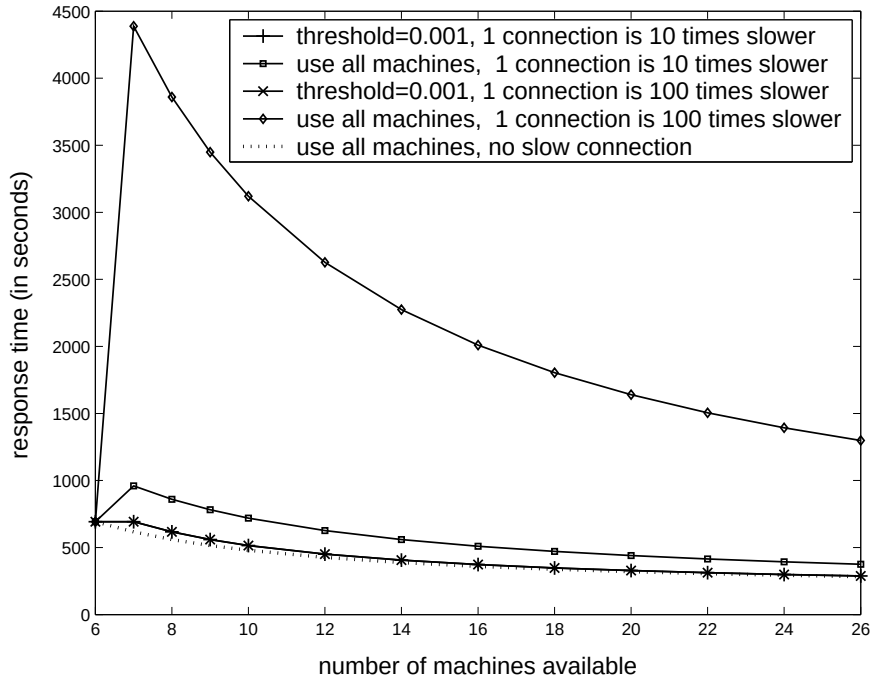


Figure 3.7: Comparison of different scheduling policies in the presence of a slow connection for the 5-join query (note that the 1st and 3rd line types essentially overlap).

KB/sec); and in the second, it is 100 times slower. All the other resources are homogeneous, i.e., the *hash join* evaluation speed is 30 microseconds for each machine, and the connection speed is 600 KB/sec for each machine apart from the one with the slow connection. In a homogeneous setting, using all the machines and applying workload balancing yields the optimal performance, provided that the workload granularity is large enough so that start-up costs are outweighed. The approach in Section 3.2 is not affected by the presence of a slow connection. From the figure, it can be seen that the new algorithm behaves exactly the same in both cases and does not employ the machine with the slow connection. Moreover, its performance is very close to the optimal behaviour (i.e., the performance if all machines are used and there is no machine with a slow connection - see dotted line in the figures). To the contrary, the performance degrades significantly when all nodes are used. These results show the merit of approaches that avoid utilisation of all the available nodes, as they allow the performance to degrade gracefully when the available machines are slow or they have slow connections. Even if the number of slow connections is large compared to the total number of machines available, the new algorithm may show no performance degradation at all; conversely a single slow connection is enough to slow down the whole query if all

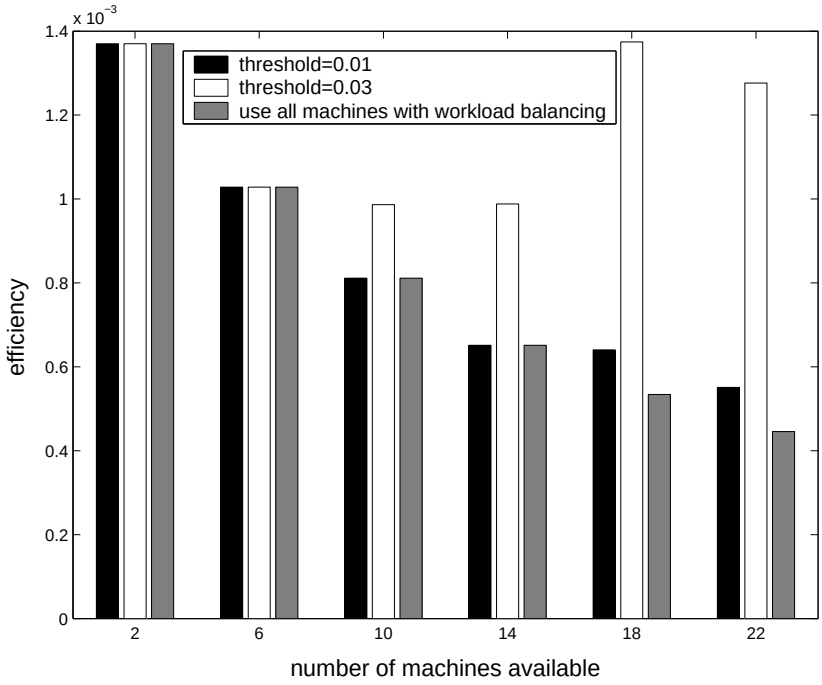


Figure 3.8: Comparison of the efficiency of different scheduling policies for the single-join query.

machines are used. The scenario of this experiment is one of the many where naive extensions from parallel computing are inappropriate for Grid settings.

3.3.4 Parallelisation Efficiency

Using machines efficiently is an important issue, especially in a Grid setting where the resources usually belong to different organisations and there may be a charge for using them. Borrowing notions from parallel systems [Mol93], efficiency is defined as the inverse of the product of the response time and the number of machines used. According to this definition, sensible comparisons of the efficiency of different schedulers can be made only for the same query. Higher values of the efficiency indicate, in practice, better utilisation of the resources. Consider, for example, three cases: (i) a query executes in 2 time units and 2 machines are used; (ii) a query executes in 1 time unit and 4 machines are used; and (iii) a query executes in 0.9 time units and 10 machines are used. In the first two cases, the parallelisation efficiency is the same, and the machines are exploited to the same extent. However, in the third case, the efficiency is significantly lower, although the response time is decreased. This indicates that usage of more machines has not been traded for performance improvements as efficiently as

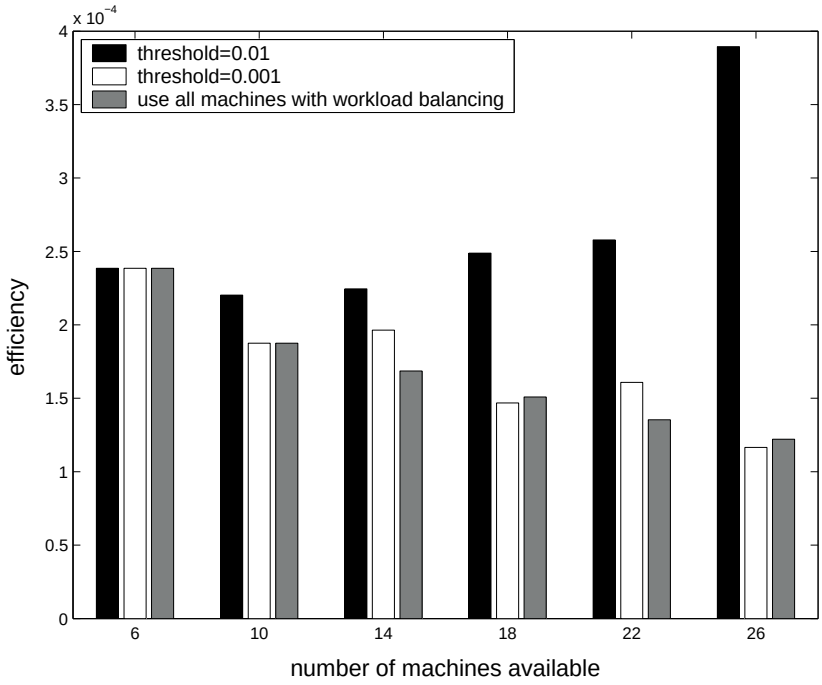


Figure 3.9: Comparison of the efficiency of different scheduling policies for the 5-join query.

in the first two cases.

Figures 3.8 and 3.9 show the efficiency of the scheduler proposed as compared against the policy of employing all the machines available, for the two queries applied to the first dataset, respectively (the performance of these queries is shown in Figures 3.2 and 3.4). For lower degrees of parallelism, the efficiency is much higher without the performance being considerably lower. By tuning the threshold we can trade the efficiency for better response times. This makes the new proposal easily adaptable to cases where the nodes are not provided for free and the economic cost may be considered as significant as the time cost.

3.4 Summary

Current distributed database applications operating in heterogeneous settings, like computational Grids, tend to run queries with a minimal degree of partitioned parallelism, with negative consequences for performance when the queries are computation and data intensive. Also, naive adaptations of existing techniques in the parallel systems literature may not be suitable for heterogeneous environments for the same reasons. The

main contribution of this chapter is the proposal of a low complexity resource scheduler that allows for partitioned parallelism to be combined with the other well-established forms of parallelism (i.e., pipelined and independent) for use in a distributed query processor over the Grid. Given that high performance and scalable querying cannot be achieved without partitioned parallelism, the scheduling algorithm contributes significantly to the aims of this thesis.

Two main attributes of the scheduler proposed are its resource-awareness and practicality. The proposal is novel as there is no previous work that deals with the scheduling of resources in heterogeneous environments to support arbitrary degrees of partitioned parallelism, in a way that considers the resource heterogeneity. The practicality of the approach lies in the fact that it is not time-consuming, it is effective in environments where the number of available resources is very large, it is dependable, and minimises the impact of slow machines or connections.

The evaluation showed that the approach yields performance improvements when no, or limited, partitioned parallelism is employed, and can outperform extensions from parallel databases that use all the resources available. It can also mitigate the effects of slow machines and connections. By being able to choose only the nodes that contribute significantly to the performance, it uses the machines more efficiently, and thus can be easily adapted to cases where the resources are not provided for free.

In summary, this chapter has contributed: (i) an analysis of the limitations of existing parallel database techniques to solve the resource scheduling problem in Grid settings; (ii) an algorithm that aims to address the limitations characterised in (i); and (iii) empirical evidence that the algorithm in (ii) meets the requirements that led to its conception in an appropriate manner and is thus of practical interest.

A less detailed description of the scheduling algorithm has appeared in [GSPF04].

Chapter 4

A Framework for Adaptive Query Processing

Having dealt with the problem of (static) resource scheduling for Grid queries in the previous chapter, this chapter proposes an architectural framework for adaptive query processing (AQP) that is capable of accommodating various kinds of adaptations, and facilitates the development of adaptive query evaluators. The basis of the framework is the decomposition of AQP into three phases:

- *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 identification of problems with the ongoing execution or the establishment of potential opportunities for improvement; and
- *response*, which enacts the decisions made.

The benefits of such a decomposition include component reuse, more systematic development, and easy deployment in a service-oriented environment as discussed later. The architectural framework has been instantiated by enhancements to the service-based and Grid-enabled OGSA-DQP system, which was described in Chapter 2. However, this chapter examines different cases of monitoring, assessment and response, and reviews existing work on adaptive query processing on the basis of the framework concepts and components; the enhancements to OGSA-DQP are discussed in a following chapter.

The structure of the chapter is as follows. Section 4.1 discusses related work on surveys and taxonomy of AQP approaches. Section 4.2 deals with the presentation of the framework along with its components. The next section, Section 4.3, provides an overview of the main alternatives in monitoring, assessment and response for AQP. Sections 4.4 and 4.5 discuss existing centralised and distributed AQP proposals, respectively. Section 4.6 summarises the chapter.

4.1 Related Work

To date, there has been no effort to develop generic frameworks for constructing AQP systems, like the one introduced in Section 4.3. Thus, this section discusses related work on surveys and taxonomy of AQP approaches, rather than on adaptivity frameworks. However, none of them is as comprehensive in terms of the techniques examined as the discussion in this thesis.

An attempt for a unifying comparison of AQP systems has appeared in [BB05]. The systems in this work are divided in three categories: (i) those that reoptimise query plans, (ii) those that are based on adaptive tuple routing like Eddies [AH00], and (iii) those for continuous queries over streams. However, this taxonomy is not consistent, since adaptive stream query processing can be based on plan modifications, or tuple rerouting. The aspects examined include monitoring techniques, and techniques to avoid duplicate results, to reuse work done before adaptations, and to reduce response overhead. These are complementary to the aspects examined in this thesis, which focus on the functionality of monitoring, assessment, and response rather than on the specific implementation approaches for each component. Another drawback of [BB05] is that it ignores (i) the assessment phase, (ii) techniques for monitoring changing resources apart from evolving data statistics, and (iii) adaptive operators such as Ripple joins [HH99]. An earlier survey, classifies the systems based on the monitoring frequency and the response form (called effect) [HFC$^+$00].

[MRS$^+$04] proposes two dimensions along which one can evaluate AQP techniques: risk, which refers to the degree to which the adaptation may cause performance degradation, and opportunity, which refers to the aggressiveness of the system. However, it is felt that these aspects are covered by the capability of the responder to choose a profitable response, and by the monitoring frequency, respectively. Finally, in [Ive02], a number of AQP techniques are compared along adaptivity frequency, response form (called power), scope of response impact, and response overhead.

4.2 The Monitoring-Assessment-Response Framework

A consensus is emerging that, so far, efforts in AQP have resulted in a collection of effective, but also rather narrowly specialized and isolated techniques [IHW04], rather than in a generic framework as comprehensive as in other areas of database research (e.g., generic algebraic transformations for query optimisation, generic interface of query operators to support the iterator execution model, etc.). This section presents a generic framework for adaptive techniques in database query processing.

4.2.1 Description and Benefits of the Framework

As mentioned earlier, the framework is based on the decoupling of 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 during execution, i.e., if there is a feedback loop between the environment and the behaviour of the query processing system during query execution. A finer-grained analysis of the feedback loop leads to the identification of three conceptually distinct phases, namely *monitoring*, *assessment* and *response*.

The construction of general frameworks for identifying or composing generic and reusable techniques for monitoring, assessment or response has the following key advantages.

- 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. Dynamically gathered information about the actual selectivity of an operator can be used either to re-route tuples through joins (like in eddies [AH00]), to reconstruct a query execution plan for the remainder of the query (in line with [KD98]), to build more accurate predictions of the query completion time, etc. By decoupling the generation of selectivity information and its usage, it becomes possible to use a single monitoring mechanism for all the above adaptive techniques. In another example, mechanisms for identifying memory shortage can be used to allocate more memory, or use more machines, or change the algorithm being used, or a combination of all

these. The same problem caused by changes in the environment can be tackled by many responses to these changes. These responses may be fine grained (e.g., directing the next tuple to a particular node) or coarse grained (e.g., rerunning the optimizer over some or all of the query). Substitutability and reusability are desirable for assessment and response as well. A form of response can be decoupled from the problem it tries to tackle. For example, reoptimisation of the query plan may be useful in different cases, including the unavailability of accurate statistics at compile time, lack of response from remote data sources, or unexpected memory shortage.

- The adoption of a generic framework by the developers of AQP systems makes the design activity more systematic. Developers can focus on the internal logic of the components of the framework, without worrying about how the others are implemented internally, but the developer of the internal logic does have to think about the local and remote interfaces. As such, the development process is expected to be less costly and less prone to error.

- By focusing on the interfaces of cohesive, decoupled modules, the design of the framework conforms to the emerging Web and Grid Services paradigms, which are suitable for advanced, wide-area applications.

- The framework components support the use of high-level *policies* that modify and shape their behaviour. An example policy is: respond only if the query completion time is expected to improve by at least 10%. Such policies are of paramount importance for autonomic systems [WHW+04].

- Finally, 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, and their specific interconnections.

Figure 4.1 depicts the components of the framework and shows how these cooperate to cause an adaptation. The query evaluator generates information about the state and progress of the evaluation of a query execution plan. The resource repository provides up-to-date information about the available resources. Based on such

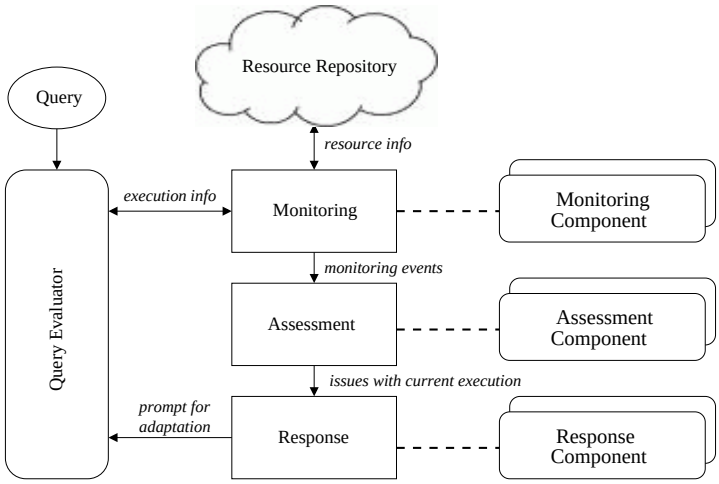


Figure 4.1: The monitoring, assessment and response phases of AQP, and the associated components.

raw monitoring information, events[1] are generated, and passed on to the assessment component. The latter evaluates these events in order to verify (i) whether they imply changes in the values of relevant properties, and (ii) whether such changes are an issue to be addressed in the current execution (i.e., whether the optimality of execution has been compromised). Once an issue has been identified, the responder component is notified. In the response phase, the system tries to identify potential ways to respond. If such ways are found, the execution engine is notified accordingly and its behaviour changes as a result. As such, deploying adaptive strategies involves the interconnection of monitoring approaches, means of assessment and forms of response.

As shown in Figure 4.1, multiple adaptivity components may be deployed to realise a single adaptivity phase. In Chapter 6, adaptations based on multiple monitoring components, and on single assessment and response components will be presented in detail.

4.2.2 The components of the framework

Any AQP technique requires at least one of each different kind of framework components to cover all the adaptivity phases. Thus, even the simplest adaptation is based on collaboration of decoupled entities. To this end, the components support a publish/subscribe interface [EFGK03] to provide and ask for services to and from other

[1] In the context of the adaptivity framework, the terms *event* and *notification* will be used interchangeably.

components, respectively. 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 and of the ongoing query execution. Other adaptivity components (including monitoring ones) interact with the MC in order to subscribe to it. The subscription procedure, as well as enabling the MC to compile a list of the modules that are interested in its notifications, specifies the mode of transmission (either push or pull, i.e., on request), and the kind of notifications it requires from the set of all the possible notifications that the MC is able to produce.

The MC interacts with other adaptivity components to deliver the notifications requested. Such notifications are in a standardised or commonly agreed form that hides how monitoring is carried out. Finally, 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 and the amount of available memory have 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 criteria), and whether there is a problem with the current execution that needs to be addressed in order to activate the self-adaptive mechanisms. 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 multiple monitoring components. The notification analysis may involve the evaluation of event-condition-action rules, the rerun of (parts of) the query optimiser, the computation of rolling averages, the update of prediction models, the comparison against static estimates and more. 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 components interact with ACs to subscribe, and monitoring components interact with ACs to deliver notifications.

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 benefit and cost for each valid response (e.g., by using cost functions); (iii) selecting the most efficient one, and (iv) interacting with the evaluation engine in order to enforce its decisions. The RC is able to subscribe to other components (e.g., ACs)

and to receive notifications.

4.3 Analysis of Adaptive Query Processing

To date, there is no mechanism or benchmark to compare AQP techniques systematically. In fact, there are too many non-comparable features amongst them [GPFS02]. The framework introduced in the previous section provides a background for describing each technique in terms of the way in which it ranges over the three adaptivity phases of monitoring, assessment and response. As these phases form a pipeline, the input and the output of each phase are of particular interest. This section provides a taxonomy of the different types of such output in existing AQP proposals, along with other, complementary aspects of the behaviour of the components.

4.3.1 Monitoring

4.3.1.1 Monitoring Events and Focus of Adaptivity

The monitoring component analyses raw monitoring information (or feedback, as these terms will be used interchangeably in the context of adaptivity monitoring) from the query engine and the resources available. It also produces notifications (*monitoring events* in Figure 4.1) processed by other adaptivity components, such as the assessment ones. According to the different type of focus of AQP systems, such notifications cover various interesting, "suspicious" observations that denote updated property values, but not necessarily problems. The updated values of these properties are the conditions the AQP attempts to adapt to. Thus there is a very close correlation between the focus of adaptivity and the type of monitoring events produced. The focus of AQP can fall into the following categories:

1. *Dataset Volume*, which includes
   (a) *Dataset Cardinality* (i.e., the number of tuples in a dataset), and
   (b) *Dataset Size* (i.e., the size of tuples in a dataset).
2. *Data Characteristics*, which has to do with the properties of the initial, intermediate and final datasets. It also includes
   (a) *Value Distribution* of tuple attributes,

(b) *Indices*, and

(c) *Data Order*.

3. *Operator Cost*, either in time units or in any other QoS metric.

4. *Resources*, which can be divided into

   (a) *Resource Pool* for changes in the number of resources that are candidate for participating in the query evaluation,

   (b) *Resource Memory* for up-to-date information on the amount of memory available, and

   (c) *Resource Performance and capabilities* for statistics on

      i. *Resource Processing Power*, and/or

      ii. *Resources Connection Bandwidth*.

5. *User Input*, such as priority ratings for different parts of the result, or for the rate of updates of partial results.

It is worth noting that these updates may be filtered in such a potentially configurable way that non-interesting changes are not passed on to other components. For example, changes in the expected selectivity may be considered interesting if they differ more than 10% from the previous expected value, or, modifications in the resource pool may be examined only if the new machines have connection speed and memory that exceeds a corresponding threshold. Additionally, they can be combined to provide more meaningful information. For example, the selectivity of an operator can be calculated from the ratio of the output and input dataset cardinalities of this operator.

4.3.1.2 Raw Monitoring Information

The focus of adaptivity, whose categories were presented previously, defines, to a large extent, the nature of the feedback collected from the query execution and/or the execution environment. Indeed, the relationship between the raw monitoring information and the type of monitoring events produced by the monitoring component is quite straightforward and intuitive in most of the cases. For example, for changes in dataset cardinalities and dataset sizes, monitoring information about the number of tuples consumed and produced by operators, and their sizes, respectively, is needed. Monitoring the number of available resources is the most basic information required for detecting

modifications in the resource pool. Changes in the resources' memory can be identified either by monitoring the memory available in that resource or the memory consumed by the operators running in it. To adapt to resource processing power, several approaches can be followed: e.g., to monitor either the CPU load of a machine explicitly, or to infer the CPU load by monitoring the time cost of CPU-bound operators. Similarly, for the connection bandwidth changes, the core monitoring information can be either the bandwidth, or, in some scenarios, the time operators wait for data from remote sources.

4.3.1.3 Monitoring frequency

The monitoring frequency is a significant characteristic of any AQP technique, as the frequency of collecting feedback determines the maximum frequency of potential adaptations. This happens because it is the feedback collection that drives the complete adaptivity cycle, which consists of the three phases of monitoring, assessment and response. The different levels of frequency are as follows:

1. *Inter-operator*, which refers to the cases in which the adaptivity cycle can be triggered through acquisition of raw monitoring information only between the execution of the operators in the query plan; and

2. *Intra-operator*, which refers to the cases in which the adaptivity cycle can be triggered many times during the execution of the same operator.

4.3.2 Assessment

After identifying an event described in Section 4.3.1, the next step is to diagnose whether this constitutes an issue with the current execution, which means that the query plan is being evaluated in a suboptimal way without yet having decided if the execution can be improved, let alone how. That is, the set of *Issues* (see Figure 4.1), is the result of the assessment phase and contains the set of potential states of the system that can prompt for a modification to the query evaluation, if there exists a beneficial one. These states explicitly denote the problems that need to be addressed.

There are five main categories of possible diagnosed issues:

1. *Suboptimal Execution*, which includes:

(a) *Suboptimal Physical Plan*: the system knows that there is a better query plan than the current one, although it does not know whether it is beneficial to change the plan on the fly.

(b) *Suboptimal Operator Scheduling*: the system knows that there is a better execution model and/or order for operator evaluation, although it does not know whether it is beneficial to change the execution model on the fly.

(c) *Suboptimal Resource Selection*: the system knows that there is a more suitable set of resources that can be selected, although it does not know whether it is beneficial to change the resource selection and scheduling on the fly.

(d) *Workload Imbalance*: the system knows that there is a better workload distribution, although it does not know whether it is beneficial to change this distribution on the fly.

2. *Resource Shortage*, which includes:

   (a) *Insufficient Memory*: the system requires more memory.

   (b) *Insufficient Processing Power*: the system requires more processing power.

   (c) *Insufficient Bandwidth*: the system requires more connection bandwidth.

3. *Resource Idleness*, which includes:

   (a) *Idle Memory*: the system under-utilises its allocated memory.

   (b) *Idle CPU*: the system under-utilises its allocated CPU.

   (c) *Idle Bandwidth*: the system under-utilises its allocated network bandwidth.

4. *Unmet Performance Expectations*: the query plan is annotated with performance expectations that the current query execution cannot meet.

5. *Unmet User Requirements*: the query plan cannot meet the user requirements that are either submitted along with the query submission, or during query execution.

These categories examine the quality of the current execution from different, complementary perspectives. The first category views the current execution from the optimiser's perspective: given the existence of updated information, the optimisation policies applied at compile time would lead to a different query plan if they could be applied at runtime. The second and the third class examine the quality of execution

from the perspective of the resource requirements and utilisation, respectively. The fourth category is concerned with the validity of the assumptions made at compile time about the execution, and uses these assumptions to establish whether an issue with the current execution exists or not. Finally, the fifth category considers the quality of the current execution on the grounds of explicit requirements posed by the users.

4.3.3 Response

4.3.3.1 Response Forms

Each query adaptation is manifested through a specific form, which is called a *Response Form*. We distinguish between five categories of response. In order to achieve adaptivity, the developer can choose among the following options and their combinations:

1. *Operator Reconfiguration*, where the operator configuration changes while the physical implementation and the connections with other operators remain the same. This class also includes the cases for which the operator implementation provides for adaptivity itself. These operators are adaptive variants of well-established physical operators, where the adaptivity phases, including response, are included within the operator implementation. Operator reconfiguration corresponds to the physical reoptimisation of the query plan.

2. *Operator Replacement*, where the physical implementation of an operator changes while the other aspects of the query plan remain the same. This response form corresponds to the physical reoptimisation of the query plan as well.

3. *Operator Rescheduling*, where the execution order of operators and/or the model of execution changes (e.g., from iterator to sequential), while the mappings of logical operators to physical ones in the query plan remains the same.

4. *Machine Rescheduling*, where the set of machines allocated to a part of the query plan changes, resulting in a new parallel plan for the same physical one. It corresponds to the rescheduling of a query.

5. *Plan Reoptimisation*, where a new equivalent algebraic expression is chosen for a part of the query plan, resulting in a new, arbitrarily different query execution plan. It corresponds to the complete reoptimisation of a query.

From the above, it can be inferred that each response form corresponds to a different kind of impact on the execution plan representing the query evaluation.

4.3.3.2 Response Scope

Depending on the granularity and the locality of the part of the query execution plan affected by the adaptation, the different response can be divided into the following categories, which correspond to the scope of the response impact on the query plan:

1. *Operator instance:* only an operator instance on a single machine is affected. In general, in parallel query processing, there may be many instances of the same operator running on different machines, which may not be affected. In essence, this category covers the case of a specific operator instance on a particular machine adapting autonomously.

2. *Operator:* all the instances of an operator in the query plan are affected, while the degree of intra-operator parallelism may be greater than one. In other words, the adaptations that refer to a specific operator in the query plan regardless its machine allocation, belong to this group.

3. *Partition instance:* only a specific instance of a plan partition on a single machine is affected. Typically, a partition instance is a pipeline plan fragment.

4. *Partition:* all the instances of a partition are affected.

5. *Query plan:* the query plan is affected in a more generic way that cannot be captured by the previous categories.

The basic benefit of distinguishing between the different scope in response is that it may be more profitable first to try to address a problem at a more local level, like an operator instance, than to proceed directly to more radical modifications affecting larger parts of the executing plan.

4.3.4 Architecture and Environment

Orthogonally to the monitoring, assessment and response phases, an important characteristic of AQP proposals is the execution environment they can operate in, and the architectural paradigms they assume. For these, the main alternatives are presented next.

4.3.4.1 Data Locality

AQP techniques differ in the data sources they can access. These data sources can be:

1. *Local.*
2. *Remote.*
3. *Streams*, which is a specific case of remote sources, but due to its peculiarity, is regarded as a separate class.

4.3.4.2 Query Processing Locality

The data locality may not always define the locality of the parts of query processing that do not deal with data retrieval. The alternatives are:

1. *Centralised* query processing, where the query execution occurs in a single geographical place.
2. *Non-centralised* query processing, where the execution occurs in multiple geographical places.

Additionally, the distinction between *single-node* and *multi-node* processing can be drawn. Parallel query processing is multi-node and centralised, whereas distributed query processing is by its nature multi-node and non-centralised.

4.3.4.3 Adaptivity Locality

The adaptivity mechanism can be engineered as the following artifacts:

1. *Operator*, in which the adaptivity mechanisms are encapsulated in the operator implementation.
2. *Central component*, in which there is a central component that drives and coordinates the adaptivity.
3. *Distributed components*, in which there are multiple distributed components that cooperate for adaptivity.

Technique	Monitoring		Assessment	Response		Architecture		
	Focus	Frequency	Issue	Response Form	Impact	Data Local.	QP Local.	Adapt. Local.
sorting [PCL93a]	resource memory	intra-operator	insufficient/idle memory	operator reconfiguration	operator instance	N/A	central	operator
PPHJ [PCL93b]	resource memory	intra-operator	insufficient/idle memory	operator reconfiguration	operator instance	N/A	any	operator
ripple [HH99]	user input	intra-operator	user requirements	operator reconfiguration	operator instance	N/A	central	operator
XJoin [UF00]	resource connections	intra-operator	idle CPU	operator rescheduling	operator instance	remote	central	operator
juggle [RRH00]	user input	intra-operator	user requirements	operator reconfiguration.	operator instance	N/A	central	operator
mergesort [ZL97]	resource memory	intra-operator	insufficient/idle memory	operator reconfiguration	operator instance	N/A	central	central
pipeline scheduler [UF01]	user input, data volume, operator cost	intra-operator	suboptimal operator scheduling, user reqs	operator rescheduling	partition instance	N/A	central	central

Table 4.1: Summarising table of operator-based AQP proposals according to the classifications of Section 4.3.

4.4 Centralised Adaptive Query Processing Techniques

This section presents the AQP techniques that refer to centralised query processing. Regarding the adaptivity locality, such techniques employ adaptivity mechanisms engineered either as operators (Section 4.4.1), or as a central component. In the latter case, AQP can be applied to both single-node and multi-node query processing. As the bulk of the work on AQP is for single-node systems, this class is further partitioned according to the data locality, i.e., the type of data sources that can be accessed: local (Section 4.4.2), remote (Section 4.4.3), and streams (Section 4.4.4). Multi-node centralised AQP systems are discussed in Section 4.4.5.

4.4.1 Operator-based Adaptivity

This part describes query operators that have the capability to adjust their behaviour according to changing conditions and the information available at runtime. A summary of their characteristics is given in Table 4.1. A common property is that, in adaptive operators, the adaptivity cycle is triggered many times during their execution, which corresponds to intra-operator monitoring frequency. Adaptive operators also act autonomously in order to adapt to changes but have limited effects on the executing query plan, as they cannot perform radical changes to it. Typically, the scope of their responses is limited to a single operator instance. Additionally, they have all been proposed for centralised query processing, although, in some cases, they can be applied to

a non-centralised environment as well.

4.4.1.1 Memory Adaptive Sorting and Hash Join

Memory adaptive operators monitor the amount of memory available, and assess this information to identify if there is idle memory or a memory shortage. In these cases they respond by self-reconfiguration. [PCL93a] introduces techniques that enable external sorting to adapt to fluctuations in monitored memory availability. External sorting requires many buffers to run efficiently, and memory management significantly affects the overall performance. The memory-change adaptation strategy introduced is *dynamic splitting*, which adjusts the buffer usage of external sorts to reduce the performance penalty resulting from memory shortages and to take advantage of excess memory. It achieves that by splitting the merge step of external sorts into a number of sub-steps in case of memory shortage, and by combining sub-steps into larger steps when sufficient buffers are available. Thus the operator is self-reconfigured. Alternative OS-level strategies, like *paging* and *suspension*, are non-adaptive and yield poor results [PCL93a]. Adopting memory-adaptive physical operators for sort operations avoids potential under-utilisation of memory resources (e.g., in the case when memory reserved prior to execution based on estimates is in fact more than the memory required) or thrashing (e.g., in the case when memory reserved prior to execution is in fact less than the memory required), and thus reduces the time needed for query evaluation.

For join execution, when the amount of memory changes during its lifetime, a family of memory adaptive hash joins, called *partially preemptible hash joins (PPHJs)*, is proposed in [PCL93b]. As in [PCL93a], they aim at avoiding both memory idleness and shortage. Initially, source relations are split and held in memory. When memory is insufficient, one partition held in memory flushes its hash table to disk and deallocates all but one of its buffer pages. The most efficient variant of PPHJs for utilising additional memory is when partitions of the inner relation are fetched in memory while the outer relation is being scanned and partitioned. This method reduces I/O and, consequently, the total response time of the query. Although such reconfigurations cannot adapt very well to rapid memory fluctuations, PPHJ outperforms non-memory-adaptive hybrid hash joins, which employ strategies such as *paging* and *suspension*.

4.4.1.2 Operators for Producing Partial Results Quickly

In many applications, such as online aggregation, it is useful for the user to have the most important results produced early. To this end, pipelining algorithms are used for the implementation of join and sort operations. The other kind of operators are block ones, which produce output after they have received and processed the whole of their inputs.

Ripple joins are initially proposed in [HH99] and enhanced later in [LEHN02]. They constitute a family of physical pipelining join operators that maximise the flow of information during processing of statistical queries that involve aggregations. They generalise block nested loops (in the sense that the roles of the inner and outer relations are continually interchanged during processing) and hash joins. Ripple joins adapt their behaviour during processing according to user preferences about the accuracy of the partial result and the time between updates of the running aggregate. These preferences can be submitted and modified during execution. Given the user preferences, the ripple joins adaptively set the rate at which they retrieve tuples from each input of the ripple join. The ratio of the two rates is reevaluated after a block of tuples has been processed, to ensure that the user requirements are met.

XJoin [UF00] is a variant of Ripple joins. Apart from producing initial results quickly, XJoin is also optimised to hide intermittent delays in data arrival from slow and bursty remote sources by reactively revising the execution order of different parts in the query plan. Thus idle CPU times are avoided. The connections to remote data sources are monitored and the objective of the assessment is to identify states that result in CPU idleness. The execution occurs in three stages. Initially, XJoin builds two hash tables, one for each source. In the first stage, a tuple, which may reside on disk or in memory, is inserted into the hash table for that input upon arrival, and then is immediately used to probe the hash table of the other input. A result tuple will be produced as soon as a match is found. The second stage is activated when the first stage blocks, and it is used for producing tuples during delays. Tuples from the disk are then used to produce some part of the result, if the expected number of tuples generated is above a certain activation threshold. The last stage is a clean-up stage as the first two stages may only partially produce the final result. In order to prevent the creation of duplicates, special lists storing specific time-stamps are used.

To obtain and process the most important data earlier, a sort-like re-ordering operator, called *juggle*, is proposed in [RRH00]. It is a pipelining dynamic user-controllable

reorder operator that takes an unordered set of data and produces a nearly sorted result according to user preferences (which can change during runtime) in an attempt to ensure that interesting items are processed first. The feedback it collects from the environment is user-defined priorities, which are assessed to ensure that interesting tuples are more likely to be processed early. It is a best-effort operator, in the sense that it does not provide any performance guarantees. The mechanism, on receipt of updated user preferences, tries to allocate as many interesting items as possible in main memory buffers, i.e., juggle is capable of reconfiguring itself dynamically. When consumers request an item, it decides which item to process. The operator uses the two-phase *Prefetch & Spool* technique: in the first phase tuples are scanned, and uninteresting data are spoooled to an auxiliary space until input is fully consumed. In the second phase the data from the auxiliary space are read.

4.4.1.3 Extensions to Adaptive Operators

A work on memory-adaptive sorting, which is complementary to [PCL93a] as discussed earlier, is presented in [ZL97]. This method focuses on improving throughput by allowing many sorts to run concurrently, while the former focuses on improving the query response time. Because it depends on the sort size distribution, in general, limiting the number of sorts running concurrently can improve both the throughput and the response time. The method proposed enables sorts to adapt to the actual input size and changes in available memory space. For the purpose of memory adjustment, all the possible stages of a sort operation regarding memory usage are identified and prioritised. The operations may be placed in a wait queue because of insufficient memory in the system. Priority ratings are assigned to the wait queues as well. When memory becomes available, the sorts in the queue with the highest priority are processed first. A detailed memory adjustment policy is responsible for deciding whether a sort should wait or proceed with its current workspace. If it is decided to wait, further decisions on which queue to enter, and whether to stay in the current queue or to move to another, are made. Like [PCL93a], [ZL97] has impact only on the configuration of specific operator instances and adapts with intra-operator frequency. However, it relies on a central view of all sorts running on the machine, and thus, contrary to [PCL93a], does not operate at the operator level.

The method in [UF01] for quicker delivery of initial results in pipelined query plans extends the notions of Ripple join [HH99] and XJoin [UF00] with the capability to reorder join operators in the query plan. Instead of scheduling operators, the *Dynamic*

Technique	Monitoring		Assessment	Response		Architecture		
	Focus	Frequency	Issue	Response Form	Impact	Data Local.	QP Local.	Adapt. Local.
mid-query reoptimisation [KD98]	data volume, operator cost	inter-operator	subopt. physical plan, perf. expectations	operator reconf., plan reoptimisation	query plan	local	central	central
progressive optimisation [MRS+04]	data volume, operator cost	inter-operator	subopt. physical plan, perf. expectations	operator reconf., plan reoptimisation	query plan	local	central	central
ingres [SWKH76]	data cardinality	intra-operator	subopt. oper. scheduling	operator rescheduling	query plan	local	central	central
eddies [AH00]	data cardinality, oper. cost	intra-operator	subopt. oper. scheduling	operator rescheduling	query plan	any	central	central
stems [RDH03]	data cardinality, oper. cost	intra-operator	subopt. physical plan, operator scheduling	operator rescheduling, replacement	query plan	any	central	central
juggle-eddy [RH02]	data cardinality, operator cost, user input	intra-operator	subopt. oper. scheduling, user reqs	operator rescheduling	query plan	any	central	central

Table 4.2: Summarising table of AQP proposals that access local stores, primarily, according to the classifications of Section 4.3.

Pipeline Scheduler groups them into independent units of execution, and then schedules these units. This reduces the initial response time of the query but may result in increases in the total execution time. The algorithm monitors the current selectivity and the cost of each of the execution units, along with dynamically changed user input about importance ratings of different datasets. The operator order is modified on the fly in the light of the monitored system behaviour and updated user requirements. The aim is to produce as many interesting result items quickly as possible, in a way that meets the user requirements if there are any stated. Ideally, the order should be recomputed every time a tuple is processed by a stream, but, in fact, the scheduler is invoked less often in order to avoid large overheads.

4.4.2 Accessing Local Data Stores

In this part, single-node AQP techniques that rely on central adaptivity-related components, rather than on adaptive operators, and, at least in their initial proposal, the query engine is co-located with the data stores are presented (a summary is given in Table 4.2). However, some of them, as explained below, have been successfully deployed in query processing over remote databases as well.

Kabra and deWitt [KD98] introduced an algorithm that detects sub-optimality in

query plans at runtime, through on-the-fly collection of query statistics, and improves performance by either reallocating resources such as memory or by reoptimising the remainder of the query plan, i.e., the impact of adaptations is larger compared to the techniques examined thus far. This method aims to combat the problem of constructing plans based on inaccurate initial statistical estimates about the data sources. The plan is annotated with expected sizes of intermediate results and time costs of operators. This information is monitored at runtime, and it is assessed to check if there is a difference between real values and annotations, which implies that the optimiser might have constructed a suboptimal plan. For monitoring, a new, dedicated operator is used. The re-optimisation algorithm relies heavily on intermediate data materialisation, and takes into account monitored information such as value distribution that is not explicitly used in the assessment phase. However, it is activated only between the completion of execution of separate pipelined fragments of the plan, which restricts the opportunities for adaptation. The proposal includes specific techniques to reduce the monitoring overhead and to prohibit the system from adapting if the expected benefit is not significant. More recent work, [MRS+04], has enhanced the above technique by enabling adaptations during the execution of pipelines.

A more dynamic and highly influential technique to tackle the same problem of inaccurate data statistics is *Eddies* [AH99, AH00, Des04]. Eddies constitute a query processing mechanism that continuously (i.e., even for each tuple) reorders operators on the fly in order to adapt dynamically to changes in operator costs and selectivities, provided that these operators are pipelined [AH99, AH00]. They essentially dynamically reroute the tuples through operators in order to achieve better response times. The approach is shown to provide significant improvements even in scenarios where remote data sources are accessed, and in stream query processing [TB02]. In order to insert eddies in a query plan, joins should allow frequent and efficient reoptimisation. Joins that are appropriate for eddies include Ripple joins, pipelined hash join, XJoin, and index joins, with the Ripple join family being the most efficient. Compared to [KD98], eddies do not alter the nature of operators but only their order, and adapt with intra-operator frequency. Also eddies do not rely on plan annotations: the assessment uses only the monitored information to check whether there is a better operator order. Adaptive ordering of nested-loop joins, based on monitoring information about the cardinalities of intermediate results, has been employed in the early Ingres database system [SWKH76], as well.

Technique	Monitoring		Assessment	Response		Architecture		
	Focus	Frequency	Issue	Response Form	Impact	Data Local.	QP Local.	Adapt. Local.
query scrambling [UFA98]	resource connections	inter-operator	idle CPU	operator resched., plan reoptimisation	query plan	remote	central	central
Bouganim [BFMV00b]	resource connections, memory	inter-operator	idle CPU, insufficient memory	operator resched., plan reoptimisation	query plan	remote	central	central
Tukwila [IFF+99, ILW+00, Ive02, IHW04]	resource connections, memory, pool, data volume, operator cost	inter-operator	idle CPU, insufficient memory, sub-opt. physical plan, perf. expectations	operator rescheduling, reconfiguration, replacement, plan reoptim.	query plan	remote	central	central
bindjoins [Man01]	operator cost	inter-operator	suboptimal resource selection	operator reconfiguration	query plan	remote	central	central

Table 4.3: Summarising table of AQP proposals that access remote stores, according to the classifications of Section 4.3.

Two non-trivial extensions to Eddies have appeared in [DH04] and [RDH03], respectively. The former provides a solution, called *Stairs*, to the problem of management of state (e.g., hash tables) within operators, to enable better routing policies. The latter introduces *SteMs (State Modules)*, which allow eddies not only to reorder operators for each tuple (or set of tuples) but also to change operator implementation. The Stems replace series of pipelined hash joins, and they also enable different queries to share states, as a single stem exists for each data source independently of the number of queries that this source participates in. They actually encapsulate indices in a unary operator and are proved to be efficient for computing multiple joins in a pipelined fashion.

Finally, eddies have been successfully combined with the juggle operator described previously, to form the *juggle-eddy* operator [RH02], which reorders both tuples and operators, in a way that combines the characteristics of the two constituent techniques.

4.4.3 Accessing Remote Sources

In this section, AQP systems that target problems specific to centralised query processing over remote sources are discussed (Table 4.3 provides a summary).

In the light of data arrival delays, a common approach is to minimise idle time by performing other useful operations, thus attenuating the effect of such delays. *Query Scrambling* [AFT98, AFTU96, UFA98] and its variants [BFMV00a, BFMV00b] are

two representative examples in this area.

Query scrambling focuses on connections to remote sources and more specifically tries to address problems incurred by delays in receiving the first tuples from a remote data source that result in idle CPU. The default response form is to reorder operators so that the system performs other useful work in the hope that the problem will eventually be resolved and the requested data will arrive at or near the expected rate from then on. If changes in the execution order to avoid idling (which is always beneficial) do not solve the problem, new operations are inserted in the query plan, which is risky. In query scrambling, there is a trade-off between the potential benefit of modifying the query plan on the fly and the risk of increasing the total response time instead of reducing it.

[BFMV00a, BFMV00b] also deal with the problem of unpredictable data arrival rates, and, additionally, with the problem of memory limitation in the context of data integration systems. A general hierarchical dynamic query processing architecture is proposed. Planning and execution phases are interleaved in order to respond in a timely manner to delays. The query plan is adjusted to the remote source connections and memory consumption. Materialisation points are dynamically inserted into pipelines to reduce the memory requirements of the query plan, in case of memory shortage [BKV98]. In a sub-optimal query plan, the operator ordering can be modified or the whole plan can be re-optimised, by introducing new operators and/or altering the tree shape. In general, operators that may incur large CPU idle times are pushed down the tree. Scheduling such operators as soon as possible increases the possibility that some other work is available to schedule concurrently if they experience delays. During the construction of query plans, bushy trees are preferred because they offer the best opportunities to minimise the size of intermediate results. Thus, in case of partial materialisation, the overhead remains low. The query response time is reduced by running several query fragments concurrently (with selection and ordering being based on heuristics) and partial materialisation is used, as mentioned above. Because materialisation may increase the total response time, a *benefit materialisation indicator (bmi)* and a *benefit materialisation threshold (bmt)* are used. A *bmi* gives an approximate indication of the profitability of materialisation and the *bmt* is its minimum acceptable value.

The *Tukwila* [IFF+99, ILW+00] project has developed a data integration system in which queries are posed across multiple, autonomous and heterogeneous sources. Tukwila attempts to address the challenges of generating and executing plans efficiently

with little knowledge and variable network conditions. The adaptivity of the system lies in the fact that it interleaves planning and execution and uses adaptive operators. The operators in the query plan are annotated with their expected output cardinality. The system monitors the availability of remote data sources, the time an operator spends waiting for input tuples to infer the resource connection speed, the number of tuples each operator produces and the amount of memory it requires, along with the operator cost. The assessment and response are achieved by event-condition-action rules. An example of such a rule is: if at the end of the execution of a pipeline, the output cardinality is half of the expected one, then reoptimise the remainder of the query plan. Possible actions include operator reconfiguration to change the amount of memory allocated or the remote data source, operator reordering, operator replacement, and re-optimisation of the remainder of a query plan. The Tukwila system integrates adaptive techniques proposed in [KD98, UFA98, BFMV00a]. Re-optimisation is based on pipelined units of execution. At the boundaries of such units, the pipeline is broken and partial results are materialised. The main difference from [KD98] is that materialisation points can be dynamically chosen by the optimiser (e.g., when the system runs out of memory). Tukwila's adaptive operators adjust their behaviour to data transfer rates and memory requirements. A *collector operator* is also used, which is activated when a data source fails so as to switch to an alternative data source.

The Tukwila adaptive query engine has also been used for XML query processing [IHW02, Ive02]. As in the case in which data is structured, the monitoring information includes aspects such as data cardinalities and rates of tuple production. In [Ive02, IHW04] an extension to [IFF$^+$99] is presented, which incorporates the notion of eddies to route tuples through operators in order to allow tuples to be routed in different query plans running in parallel. As many different plans can be used during evaluation, a final clean-up phase is required to ensure result correctness.

BindJoins have also the capability to change the data sources accessed on the fly, like the Tukwila's collector operator, in a self-reconfiguration manner [Man01]. The difference is that (i) BindJoins focus on operator cost rather than on resource connections, and (ii) they adapt when the operator cost can be reduced by a better machine allocation, rather than when performance expectations are not met.

Technique	Monitoring		Assessment	Response		Architecture		
	Focus	Frequency	Issue	Response Form	Impact	Data Local.	QP Local.	Adapt. Local.
CACQ [MSHR02], psoup [CF03]	data cardinality, operator cost	intra-operator	suboptimal operator scheduling	operator rescheduling	query plan	stream	central	central
dQUOB [PS01]	data characteristics	intra-operator	subopt. oper. scheduling	operator rescheduling	query plan	stream	central	central
chain [BBDM03]	data cardinality	intra-operator	subopt. oper. scheduling	operator rescheduling	query plan	stream	central	central
stream [MWA+03, BMM+04]	data cardinality, resource memory	intra-operator	subopt. oper. scheduling, insuf. memory	operator rescheduling, reconfig.	query plan	stream	central	central

Table 4.4: Summarising table of AQP proposals over streams, according to the classifications of Section 4.3.

4.4.4 Stream Query Processing

In stream query processing many assumptions that are the basis of traditional database systems do not hold [CcC+02, GÖ03]. Not only are the queries inherently long-running, but data are pushed to the query evaluation engine asynchronously, rather than being pulled from a permanent store on demand. Thus, instead of simply answering streams of queries over static data, data can be streamed over queries as well. As the data characteristics and arrival rates of input streams fluctuate over time, and the resources available are subject to changes, AQP techniques have been employed to gain dependability and reasonable performance. The techniques that will be discussed next operate over streams produced by sensors and other autonomous remote sources, but, as in all techniques presented to this point, the query processing takes place in a centralised manner (Table 4.4).

An initial blending of Eddies [AH00] and Stems [RDH03] yielded the *CACQ (Continuously Adaptive Continuous Queries)* technique [MSHR02], which performs adaptations over many queries running in parallel. Thus the focus is slightly shifted towards inter-query rather than intra-query adaptivity, which is out of the scope of this thesis. CACQ shares the characteristics of Eddies and Stems: it adapts with intra-operator frequency, it impacts on the complete query plan, it monitors the operator cost and intermediate data cardinality (to infer the operator selectivity), and it revises the execution order, when, in the light of updated information, this becomes suboptimal. Stems are used only to share state, and their capability to replace operator implementations is not exploited in the context of CACQ. Nevertheless, CACQ provides a promising solution for adapting to changing workload, which affects operator cost, and skewed value distribution, which affects operator selectivity within streams.

[CF03] presents *PSoup*, which is an extension to CACQ. PSoup's query engine permits the queries to refer to data arrived before the submission. The basic concept is to treat data streams and queries in the same way, using and extending the SteM and Eddies technology. Stems in Psoup store queries as well, and eddies route queries and not only data tuples, although both stems and eddies were not initially designed for this. It is important to note that the software platform for all the above has been *TelegraphCQ* [CCD+03, KCC+03].

dQUOB conceptualises streaming data with a relational data model, thus allowing the stream to be manipulated through SQL queries [PS00, PS01]. For query execution, runtime components embedded into data streams are employed. Such components are called *quoblets*, which correspond to query operators. Detection of changes in data stream behaviour is accomplished by a statistical sampling algorithm that runs periodically and gathers statistical information about the selectivities of the operators into an equi-depth histogram. Based on this information, the quality of quoblets order is re-assessed, and the system may choose to reorder operators on the fly.

Two other, related adaptive proposals for stream systems are the *Chain* [BBDM03] and *STREAM (Stanford Stream Data Manager)* systems [MWA+03, BMM+04]. The Chain focuses on adapting to data arrival rates, like, for example, XJoin [UF00]. However, the aim is to keep memory usage at a minimum level rather than avoiding idle CPU times. Thus Chain tries to control the size of intermediate results stored in input queues of operators, and monitors their sizes, instead of monitoring the resource connections. Nevertheless, it employs the response form of operator rescheduling to achieve this.

[MWA+03] uses the same technique as Chain in the context of the STREAM system. It also allows query operators to reconfigure the memory they are allocated explicitly. When there is not enough memory, the system starts evaluating queries over streams in an approximate manner. The memory allocation policy is re-evaluated, in such a way that the precision of final results is maximised. In general, approximation is not acceptable in traditional query processing, but in many wide-area scenarios accessing remote data there is no requirement for 100% accuracy. STREAM, like Chain, also monitors the operator selectivities to obtain their actual value, and thus, to enable the determination of the optimal execution order [BMM+04]. However, heuristics are employed as an exhaustive search of possible ordering is not considered due to its complexity. AQP in STREAM attempts to balance three conflicting objectives: low runtime overhead, high speed of adaptivity and good convergence to the solution of a

Technique	Monitoring		Assessment	Response		Architecture		
	Focus	Frequency	Issue	Response Form	Impact	Data Local.	QP Local.	Adapt. Local.
river [ADAT+99]	resource performance	intra-operator	workload imbalance	operator reconfiguration	operator	local	central	central
flux [SHCF03]	resource performance	intra-operator	workload imbalance	operator reconfiguration	operator	local	central	central
parad [HM02]	data cardinality, resource pool, memory	inter-operator	insufficient memory	machine rescheduling	query plan	local	central	central

Table 4.5: Summarising table of parallel AQP proposals according to the classifications of Section 4.3.

static system if parameters stabilize. Different variants of operator ordering algorithms are presented, which trade, to a various extent, one of these objectives in favour of the others. If the runtime overhead decreases, the convergence and the adaptivity speed degrade. Algorithms with worse convergence properties adapt more quickly and incur lower overhead. Quick adaptations may not converge satisfactorily and may be quite costly as well.

4.4.5 Parallel Query Processing

Most of the work in AQP refers to traditional non-parallel query processing. However, there exist three proposals that consider adaptations when the execution is partitioned across many nodes (Table 4.5). The first two operate at the operator level, monitor the performance of participating nodes, and adapt the workload distribution to avoid imbalances. The third technique is capable of modifying the machine scheduling on the fly, to ensure that there is enough aggregate memory to evaluate partitioned joins.

A *River* [ADAT+99, AD03] is a dataflow programming environment and I/O substrate for clusters of computers to provide maximum performance even in the face of performance heterogeneity. In intra-operator parallelism, data is partitioned to be processed among system nodes. The two main innovations in a River are *distributed queues (DQ)*, which can tolerate faulty consumers, and *graduated declustering (GD)*, which can tolerate faulty producers. These techniques are complementary and enable the system to provide data partitioning that adapts to changes of the performance in production and consumption nodes. DQs are responsible for naturally balancing load across consumers running at different rates. The aim is to make the rate of consumers equal to the rate of delivery from the producers. GD is used for full-bandwidth balanced production, with the effect that all available bandwidth is utilised at all times,

and all producers of a given data set complete near-simultaneously. It requires the existence of replicas of initial data though, and favours the faster producers. In both cases, the adaptation is manifested through the reconfiguration of either the producers or the consumers. Thus the impact of adaptation is constrained to these operators.

One limitation of Rivers is that they cannot operate when data cannot be delivered to arbitrary nodes. E.g., for parallel joins, tuples for which the value of the join attributes is equal need to be routed to the same node (i.e., the data partitioning is content-sensitive). The *Flux* operator extends the concepts of rivers for repartitioning of data according to the resources' performance to avoid workload imbalance, even when the data distribution is content-sensitive [SHCF03]. Essentially, the flux is an enhanced version of the exchange operator that encapsulates parallelism and communication in parallel query processing [Gra90].

Finally, the authors in [HM02] have proposed a technique, called *ParAd* that adjusts the degree of intra-operator parallelism for joins, when all the available nodes are homogeneous. At the end of the execution of each pipeline plan fragment, the system monitors the number of processors available, the amount of memory available and data statistics such as volume of intermediate results. The machine scheduling algorithm is called when changes in the monitoring information are detected, to ensure that all joins in the query plan are executed in main memory. I.e., the number of machines executing joins can be modified dynamically.

4.5 Distributed query processing

Thus far, the AQP techniques referred to centralised query processing. However, query processing on the Grid falls, by its nature, in the scope of DQP. The AQP systems for this kind of query processing can be divided into two categories, according the purpose of monitoring: (i) to keep track of the data volumes processed, and (ii) to keep track of the properties of the resources that contribute to the query evaluation. These two categories are discussed separately, and Table 4.6 provides a summary.

4.5.1 Focusing on data properties

To make better decisions regarding query evaluation, algorithms may wait until they have collected statistics derived from the generation of intermediate results thus far. Such an approach has been used in centralised query processors, such as the mid-query

Technique	Monitoring		Assessment	Response		Architecture		
	Focus	Frequency	Issue	Response Form	Impact	Data Local.	QP Local.	Adapt. Local.
mind [ONK+97]	data cardinality	inter-operator	subopt. resource selection	machine rescheduling	partition	remote	non-central	central
adaptive SDD-1 [YS97]	data cardinality	inter-operator	subopt. operator schedul., resource selection	operator resched., machine resched.	partition	remote	non-central	central
aurora* [CBB+03]	data cardinality	intra-operator	resource shortage	machine rescheduling	operator	stream	non-central	distributed
conquest [NWM98, NWMN99]	resources, data volume	inter-operator	suboptimal physical plan, resource selection, imbalance	plan reoptimisation, machine rescheduling	query plan	remote	non-central	central
SwAP [ZOTT05]	data cardinality	intra-operator	subopt. oper. scheduling	operator rescheduling	query plan	any	non-central	distributed

Table 4.6: Summarising table of distributed AQP proposals according to the classifications of Section 4.3.

reoptimisation technique [KD98]. The *MIND* system [ONK+97, ONK+96] uses a similar approach in a distributed environment. It address the problem of where to perform combination operations, like joins, on partial results of a global query, which are returned by local DBMSs in a multidatabase system. Due to the fact that useful statistics about these results are difficult for a static optimiser to estimate, decisions on where to execute operations on data that reside at remote sources are deferred until the arrival of the actual partial results. The system monitors the cardinality of partial results produced by each database, and assesses this information in order to identify better machine scheduling policies. The decisions affect the plan partition that is scheduled dynamically. Since the system waits for databases to produce partial results, it can adapt only at predefined points of execution. It also requires a central adaptivity component to monitor, assess and respond. [YS97] provides an adaptive variant of the SDD-1 algorithm that introduced *semi-joins* [BGW+81], and exhibits the same adaptive behaviour as MIND, with the additional capability to decide at runtime the join order, and not only the machine allocation. However, this response form is the result of the assessment of the monitoring information about the data cardinalities as well.

The work in [CBB+03] is distinguished from other adaptive stream query processing techniques in that it does not restrict the execution to occur in a single, central location. It builds upon the *Aurora* centralised query processor [CcC+02] to produce distributed stream processors, both for single-ownership domains (*Aurora**) and multiple-ownership ones (*Medusa*). The system monitors the load of each node participating in the evaluation, in terms of the tuples it processes. As the input rates of

data streams vary over time, and are unpredictable, the load changes frequently, and resource shortage problems may arise. To tackle these problems, a response, in the form of machine rescheduling, is made. This can be done in two ways. Firstly, by changing the machine to which an operator is allocated (*box sliding*), and secondly, by partitioning an operator across multiple machines on the fly (*box splitting*), provided that the operators do not hold state at the moment of splitting. An important aspect of the system is that the adaptivity mechanism is decentralised, and the nodes need to cooperate to adapt.

4.5.2 Focusing on changing resources

The *Conquest* query processing system enables dynamic query optimisation in a parallel environment for long-running queries following a triggering approach in response to runtime changes [NWM98, NWMN99]. The triggering approach is similar to the event-condition-action rules used in Tukwila [IFF+99]. Monitored changes relate to resources (e.g., new processors become available, while others may be withdrawn), and data characteristics relevant to query statistics (e.g., a buffer queue becomes empty, or the volume of partial results is higher than initially estimated). Such changes make the re-computation of cost estimates and plan re-optimisation necessary. In the assessment phase, suboptimality in the physical plan and in the scheduling, and imbalance in the workload distribution are identified. The capabilities of Conquest for modifying a query plan to address these problems are very strong, especially in terms of partition scheduling, but are limited to unary operators, such as scans (if there exist replicas) and single-input user-defined functions. Moreover, the user needs to provide a significant set of metadata for each user-defined function registered in order to enable adaptations, and to define the triggers at compile time, which may not be practical. In addition, the adaptivity architecture is centralised at all phases (monitoring, assessment, and response), which makes the approach non-scalable.

An investigation of Eddies and Stems in a distributed setting has been made in [ZOTT05, Zho03]. The resulting system is called *SwAP (Scalable and Adaptable query Processor)*. The basic idea of this work is to place an Eddy on each machine used in the query evaluation. By monitoring input and output dataset cardinalities on each site, information about the operator selectivities, and the workload and transmission speed of different machines can be inferred. For example, if one machine consumes

tuples faster than another, this may indicate that it is less loaded. The monitoring information is necessary for applying the *routing* policies in Eddies. However, the proposal suffers from serious limitations in the way it handles intra-operator parallelism. In particular, it replicates data on different sites to avoid the creation of wrong results when adaptively routing tuples across multiple instances of a join. Nevertheless, a key feature is that, by deploying an eddy at each site, no central adaptivity control is required, and thus the architecture is scalable. Orthogonally to SwAP, routing policies specifically for distributed Eddies are examined in [TD03].

4.6 Summary

The contribution of this chapter is twofold. Firstly, a generic framework for AQP is proposed, to facilitate the development of AQP systems and their combination. It is based on the decomposition into and the separate investigation of three distinct phases that are inherently present in any AQP system: (i) *monitoring* of query execution and environment to collect the necessary feedback; (ii) *assessment* of the feedback collected; and (iii) *responding* according to the results of the assessment process. Secondly, a taxonomy of the existing AQP proposals is presented on the grounds of (i) the focus of monitoring and the frequency of feedback collection, (ii) the issues examined in the assessment phase, (iii) the form and impact of response, and (iv) the type of data sources, and the locality of the query processing and the adaptivity mechanisms.

Table B.1 in Appendix B summarises the characteristics of the AQP proposals examined, providing a unified view of the Tables 4.1-4.6. From these tables, two significant observations can be made:

- The same issue can be tackled by multiple response forms. For example, operator reconfiguration in PPHJ [PCL93b] and machine rescheduling in ParAd [HM02] both tackle memory shortage. Also, a response form may be employed for several problems with the current execution. In Eddies [AH00] operators are reordered to avoid plan suboptimality, whereas in Tukwila [IFF+99] the same response form addresses CPU idleness. Moreover, the same monitoring information can be assessed with a view to establishing different problems (e.g., [KD98] and [ONK+97] both monitor data cardinality for different reasons). Finally, the assessment of different monitoring information can lead to the same result (e.g., bindjoins [Man01] and Conquest [NWM98, NWMN99] assess different information to identify the same problem). These remarks prove that the framework

is capable of revealing the commonalities between techniques, and thus of serving its purpose as a basis for combining and reusing adaptivity components.

- The following areas of AQP have not attracted enough attention to date: (i) monitoring focused on changing resource properties, (ii) AQP for distributed query processing, and (iii) distributed adaptivity mechanisms. All of them are particularly relevant to query processing on the Grid, and are examined later in this thesis.

The description of the AQP framework has been discussed in [GPSF04] as well. Also, parts of the presentation of the AQP techniques have appeared in an early survey that is part of the work of this thesis [GPFS02].

Chapter 5

Monitoring a Query Plan

Monitoring constitutes the first phase in any AQP technique, as discussed in Chapter 4. To adapt the query execution to changing resources, the Grid monitoring middleware may be useful. More specifically, mechanisms such as the MDS [CFFK01] and NWS [WSH99] are provided, which monitor Grid resources in an application-independent way. Complementary to this, this chapter introduces a new generic monitoring approach, namely *self-monitoring* operators, which can monitor the query execution itself and its performance with respect to the environment and the resources employed. This second aspect of monitoring is necessary, as AQP on the Grid relies on runtime information both from the resources in the execution environment and from the execution itself.

In line with the vision of the monitoring-assessment-response framework of the previous chapter to develop AQP techniques according to a single generic framework in order to facilitate component reuse and sharing, this chapter presents the self-monitoring operators in an assessment-independent manner. It discusses the monitoring of query execution as a topic in its own right, and it shows how the proposal can support existing AQP techniques; applications of self-monitoring operators to Grid-specific adaptations are demonstrated in the next chapter.

The core concept in the approach followed is to make the query execution engine capable of monitoring itself, by transforming the operators comprising the engine into self-monitoring ones. Monitoring the execution of a query can provide evolving estimates for properties of the query, such as its completion time and the number of values in its result. Such information can be useful for providing feedback to users and refining cost models for use in subsequent queries, as well as for AQP. However, this thesis

examines monitoring only from the perspective of AQP.

Self-monitoring operators, as a generic mechanism for monitoring, (as well as being independent of any particular assessment approach) should be characterised by the following desired properties:

- capability to drive adaptations;
- comprehensiveness, to cover as many monitoring cases as possible;
- ease of incorporation into existing systems, to attain applicability;
- scalability in terms of query size and number of machines participating in query processing; and
- low overhead, to be of practical interest.

This chapter discusses the extent to which the above properties are achieved by self-monitoring operators. Its structure is as follows. Section 5.1 discusses the related work. Section 5.2 introduces the self-monitoring operators, discussing the information they can capture. Section 5.3 describes how self-monitoring operators are relevant to the assessment process of adaptivity. The approach is evaluated in Section 5.4. Section 5.5 concludes the chapter.

5.1 Related Work

In AQP, monitoring is not normally addressed as a stand-alone topic. Rather, individual proposals either tend to group together an approach to monitoring, a means of assessment, and a form of response (e.g., [UFA98]); or just take the existence of monitoring for granted (e.g., [TD03]). Simply assuming that the monitoring information that drives adaptation is in place justifies both the necessity and the pertinence of dealing with monitoring separately. To the best of the author's knowledge, the work of this thesis is the first that does so.

Three different approaches to monitoring for AQP can be identified in the literature: use of an independent and centralised component within the query processor for monitoring (e.g., [BFMV00b, NWMN99]); construction of new physical query operators dedicated to statistics collection (e.g., [KD98]); and transformation of traditional operators to self-monitoring ones (e.g., [HH99]). Other existing proposals employing self-monitoring operators are tailored to specific AQP techniques rather than providing

generic information [UF00, AH00, HH99, UF01, RRH99, PCL93a, ZL97, PCL93b]. Thus they differ significantly from the proposal of this thesis.

Centralised monitoring components and operators dedicated to monitoring suffer from the following limitations. A centralised monitoring component, apart from requiring significant changes in the architecture of query engines, does not scale well in parallel or distributed settings, due to the communication overhead incurred. Dedicated operators require modifications in the query optimisers, which are responsible for deciding which monitoring operators are employed for each query and where. Both centralised components and dedicated operators suffer from limitations in the scope of the monitoring information that can be gathered. For example, a dedicated monitoring operator can collect useful information about the value distribution of intermediate results, but cannot provide any information about the time cost of other operators in the query plan, as it can only monitor the data it processes. On the other hand, a centralised component can observe the behaviour of algebraic operators and their cost, but cannot monitor data properties like value distribution. An important detail is that operator-specific monitoring cannot be performed using the two alternative approaches to monitoring, i.e., dedicated monitoring operators or centralised components, because operator implementation details are transparent to them.

Kabra and DeWitt [KD98] provide an example of using a separate monitoring operator for collecting statistics about data on the fly to drive AQP, provided that this is possible in one pass of the input. The statistics collected include the cardinality of intermediate results, their average size, and certain histograms. However, their approach requires the monitoring points to be defined at compile time, it cannot operate in parts of the plan that are executed in a pipelined fashion, and, it cannot capture timings referring to other operators (e.g., the time taken for a lower operator to process a tuple). These limitations, which are essentially limitations of the approach to monitoring in which dedicated operators are employed, do not arise in the approach presented in this thesis.

In essence, a wide range of adaptive techniques can implement their assessment and response strategy on top of self-monitoring operators (specific examples are mentioned in Section 5.3.4). This cannot be achieved by operators dedicated to statistics collection or new components in the architecture of the query engine, as both these techniques can capture a significantly smaller amount of monitoring information.

The overhead of monitoring has not been explicitly considered in the literature of AQP. Information about the overhead is included in LEO [SLMK01], which monitors

the query plan but only collects information about operator and predicate selectivities, and about the cardinalities of the intermediate results. This additional information is stored so as to enable the adjustment of the query optimizer for the subsequent queries. The overhead is about 5% and has been regarded by the authors as small.

Self-monitoring operators have been employed in two recent techniques that estimate the progress of query execution dynamically. This information can be useful in AQP, as the system may choose not to proceed to any adaptations if the query is close to completion. In [LNEW04], the output cardinality and the average tuple size of each pipelining segment of the query plan is measured. The overhead imposed is around 1%. The prediction formulas employed measure the execution cost in abstract units *U*, that can cover metrics such as CPU cycles, I/Os and bytes processed. The total cost of the query is the total number of *U*s required by all pipelining segments. The formulas to build estimates are heuristic and based on proportional logic (i.e., the average cost per tuple for the remainder of the query execution is equal to the average per tuple cost for the completed part of the execution), as in the approach of this thesis.

[CNR04] is a similar work that is capable of reporting back the percentage remaining. Metrics such as the number of returned tuples do not reflect the actual execution status because of blocking operators. For example, there might be a very expensive pipelining join followed by a (blocking) inexpensive sort. In this case, the first results are returned after most of the execution has been completed. The total work is described by the number of *next()* calls needed, as the iterator model is assumed. The problem is to define accurately how many *next()* calls are needed, which involves the estimation of how many tuples each operator produces. However, this is typically known only after the completion of the query for operators with variable selectivity. The solution involves the monitoring of the output cardinalities of the operators that produce data in a pipeline (i.e., table scans, index scans, group-by, etc.) The overhead is not considered as it is deemed to be negligible by the authors.

5.2 Self-monitoring operators

The approach to monitoring query execution proposed in this thesis is based on self-monitoring operators that capture metrics in the form of

- counters;
- timings (i.e., placing two timestamps and computing their difference); and

Symbol	Description
n	number of tuples produced so far
n_{inp}^{j}	number of tuples received from the jth input
t	time elapsed since the operator was created
t_{real}	time the operator is active
t_{tuple}	time to process a tuple, i.e., time to evaluate the next() function in the iterator model [Gra93]
s	size of an output tuple
mem	memory used
t_{wait}^{j}	time waiting since last tuple from the jth input

Table 5.1: General measurements on a physical query operator.

- computations of tuple sizes.

This section identifies measurements that can be taken from physical operators. Although the measurements should be able to be expressed as counters, timings, or sizes, this is not a very restricting limitation as they can cover, as shown below, a broad range of operator properties. There are two kinds of measurements, corresponding to two different levels of monitoring: generic measurements that can be applied to any physical operator, e.g., index-scan, hash join and so on; and operator-specific measurements that decompose the operator's functionality into simpler parts, and that are essential for monitoring at a finer granularity.

5.2.1 Operator-independent monitoring

Table 5.1 presents quantifiable properties that are common to all physical operators. Such properties are not related to specific implementations or functionalities of the operators, and cover the following distinct aspects of their behaviour:

1. *Operator workload and selectivity:* for this, the measurements that are useful are the number of tuples consumed n_{inp}^{j}, which is equal to the number of tuples processed, and the number of tuples produced n.

2. *Operator cost:* to monitor the cost of the operator, various timings can be captured. The time elapsed since the operator's instantiation t reflects the evaluation time of that operator in systems where all tuples are first processed by one operator before being sent to another. In systems that follow a different approach

(e.g., the iterator model of query execution [Gra93]), in the absence of blocking operators, this time may converge for all the operators in a query plan, and approximate the query execution time. In such systems the time the operator is active, t_{real}, is not the same as t. At a finer granularity, t_{tuple} gives the time cost for each data item processed.

3. *Resource requirements:* when an operator needs to maintain certain state, it is important to monitor its memory requirements mem, along with the size s of intermediate results produced, especially in the case when these have to be kept in main memory or on secondary storage, as such resources are not always abundant.

4. *Connections with other operators and data stores:* as well as obtaining basic measurements, characteristics of the execution of the part of the query plan below the relevant operator can be inferred, such as the delivery rate of data sources, and in a distributed setting, potential points of network failure, by monitoring the time the operator waits for its inputs to deliver data (t^j_{wait}).

However, more useful and easily exploited monitoring information is obtained by aggregate statistics, e.g., averages, sums, counts, minimums and maximums. Aggregates can be taken in two ways. In one approach, a window is assumed and only the measurements that belong to that window are used to compute the aggregate. Windows can be either overlapping or disjoint, and their widths can be defined in either time units or the number of most recent tuples. In the second approach, the aggregate is computed over all the values seen. For each of the metrics in Table 5.1, additional information can be derived by performing aggregate functions on them. For example, the average number of result tuples $avg(n)$ over a period of time gives the output rate for that period; the sum of the sizes of each output tuple $sum(s)$ equals the size of that intermediate result; and the minimum time waiting for new tuples from a remote data source $min(t^j_{wait})$ can provide an upper bound on the data delivery rate.

Orthogonal to the nature of the measurements, there are numerous potential policies with regard to the frequency of monitoring. Some metrics need to be computed only once during the lifetime of a particular instance of a physical operator (e.g., the time elapsed since the operator's instantiation). Other information is inferred from observing each of the tuples that comprise the operator's input separately, or just some of them (e.g., by sampling).

Symbol	Description
n_{cond}	number of conditions evaluated per predicate
t_{pred}	time to evaluate a predicate

Table 5.2: Measurements for operators that evaluate predicates.

5.2.2 Operator-specific monitoring

In Section 5.2.1, the information collected was generic to all operators and independent of their role in the query plan. However, monitoring at a finer level of granularity may require specific data from distinct operator instances, according to their functionality. By drawing such distinctions, the set of measurements in Table 5.1 can be further extended. As the functionality of different operators is well-established, monitoring the inner basic functions of each operator is still generic and implementation-independent. Such monitoring can be crucial to understand in-depth implementation-specific properties like execution time. For instance, a system may experience significant variances in the performance of a hash join that is evaluated completely in main memory, due to the existence of skew in the sizes of the buckets in the hash table. If no operator-specific monitoring is considered, such a cause of performance degradation is harder to identify.

Operator-specific monitoring can be applied to any kind of operator. The operators in Table 2.1 along with *exchanges*, which are a representative set of physical operators, have been chosen to demonstrate this approach These operators are sufficient for evaluating SQL and OQL queries of the *Select-From-Where* form in a parallel or distributed environment.

An example of operator functionality that is not present in all operators is the predicate evaluation (see Table 2.1). A predicate consists of one or more conditions. Table 5.2 summarises monitoring information with regard to the evaluation of predicates.

5.2.2.1 Operators that retrieve tuples from store

Operators that touch the store include scans and some joins in object-oriented environments. Because the store format is usually different from the tuple format required by the query processor, a mapping between the two formats needs to take place. Monitoring information that is relevant to this kind of operators is shown in Table 5.3.

Symbol	Description
t_{conn}	time to connect to source
n_{pages}	number of pages read
t_{page}	time to read a page
t_{map}	time to map store format into evaluation format

Table 5.3: Measurements for operators that touch the store.

Symbol	Description
s_i	size of a tuple in the ith input in bytes
B_j	size of the jth bucket in bytes
N_j	cardinality of the jth bucket
M_j	number of tuples in the right input that correspond to the jth bucket
t_{hash}	time to hash a tuple
t_{conc}	time to concatenate two tuples

Table 5.4: Hash-Join-specific measurements.

5.2.2.2 Hash-join

A hash join is executed in two phases. In the first phase, the left input is consumed and partitioned into buckets by hashing on the join attribute of each tuple in it. In the second phase, the same hash function is used to hash the tuples in the right input. The tuples of the right input are concatenated with the corresponding tuples of the left input by probing the hash table. Subsequently, the predicate is applied over the resulting tuple. The optimiser needs to ensure that the left input is the smallest input. Table 5.4 presents metrics that are particular to hash joins.

5.2.2.3 Unnest

The unnest operator takes as input a tuple with a n-valued attribute (or relationship), and produces n single-valued tuples. The cardinality of the collection attribute or relationship $Card_{col}$ can be monitored (Table 5.5).

Symbol	Description
$Card_{col}$	cardinality of a multi-valued attribute

Table 5.5: Unnest-specific measurements.

Symbol	Description
s_i	size of input tuple
$n_{buffers\_sent\_i}$	number of buffers sent to the ith consumer
$n_{buffers\_received\_i}$	number of buffers received from the ith producer
t_{pack}	time to pack a tuple
t_{unpack}	time to unpack a tuple

Table 5.6: Exchange-specific measurements.

5.2.2.4 Exchange

The exchange operator encapsulates parallelism in multi-node environments. It performs two functions concurrently. It packs tuples into buffers and sends these buffers to consumer processors, while receiving packed tuples from buffers sent by producers and unpacking them. The monitoring information for exchanges is given in Table 5.6.

From the above, it is evident that operator-specific measurements for a specific operator are defined solely on the basis of the distinctive functions that this operator performs, which are common in any of its implementations. This ensures that the measurements can apply to multiple systems, and provides the criterion for defining the measurements of operators not included in Table 2.1.

5.3 Enabling query plan adaptations

Traditionally, database systems use optimisers that rank candidate query plans on their predicted cost and, typically, select a plan on the basis of its low predicted cost. If the actual cost of the selected plan turns out to be substantially different from that predicted by the cost model, this may indicate that the chosen plan is not in fact the most suitable. Thus there needs to be an association between the information collected during monitoring and the cost model for the algebra, which models the behaviour of individual algebra operators. The cost metrics can be indirect (e.g., size of intermediate results), or direct (e.g., execution time). Not only the complete query plan, but also the operators that comprise it can be annotated with performance predictions. Monitoring the cost of the operators can thus inform the calibration of the cost model used in estimation based on a post-mortem analysis. However, identifying erroneous estimates that refer to the final state of the operator at runtime, which is a monitoring task directly related to dynamic query execution, may not be trivial. To this end, the monitoring

mechanism should be enhanced (i) with the capability to predict the final cost of a query plan, or subplan, based on monitoring information that has become available up to that point; and (ii) with the capability to identify operation states that will prevent the system from reaching the expected performance.

In this section, the monitoring approach is applied to the operators in Table 2.1. More specifically, it is verified that a deviation from initial expectations can be not simply detected (Section 5.3.1), but also predicted on the fly. It is first examined if this can be achieved through *local* monitoring (Section 5.3.2), i.e., without passing monitoring information between operators; then, in Section 5.3.3, this constraint is relaxed. The reasons why one would choose to perform local monitoring are threefold: firstly, one might not want any extra communication overhead regardless of the potential benefits; secondly, the query plan could be executed using blocking operators or materialisation points, which means that, effectively, only one operator is active at any time; and thirdly, initial estimates may only be available for particular operators or particular properties of operators, as is commonly the case for heuristic-based optimisation.

Regarding the predictions, this work does not seek to propose accurate formulas for all the possible cases, implementations, system configurations, value distributions, etc, but rather to demonstrate that such a generic monitoring approach is suitable as a basis for prediction mechanisms. For this reason, the signatures of the prediction formulas are more important than the formulas themselves, as they depict more explicitly the monitoring information required to predict whether there will finally be a deviation from the expected performance or not.

The final part of the section (Section 5.3.4) complements the above by demonstrating how the self-monitoring operators can be applied to other adaptive query processing techniques that support different approaches to assessment and response, some of which may not use prediction mechanisms at all.

The cost of operators is estimated according to the detailed cost model described in [SPSW02][1], in which the cost metric is time units. Here, the focus will be on three aspects of operator execution: the selectivity σ, as it determines the workload for the remainder of the query plan and is hard to predict accurately at compile time when no statistics are available; the size of the result S; and the completion time T, which defines the operator's cost.

In the remainder of the section the following additional notation is used: for each

[1] This cost model is partially discussed in Appendix A, along with the development of a variant of it.

Symbol	Description
σ	monitored selectivity
S	monitored size of result set
S_{inp}^{j}	monitored size of the jth input
T	monitored completion time
$\widehat{\sigma}$	selectivity as known at compile time
$\widehat{S}$	size of result set as known at compile time
$\widehat{S_{inp}^{j}}$	size of the jth input as known at compile time
$\widehat{T}$	completion time as known at compile time
$\widehat{n_{inp}^{j}}$	number of tuples received from the jth input as known at compile time

Table 5.7: Symbols denoting additional operator properties.

property x being monitored at runtime, $\widehat{x}$ is its static value, either known or estimated at compile time. Each operator is annotated at compile time with expected selectivity $\widehat{\sigma}$, result size $\widehat{S}$, input cardinality $\widehat{n_{inp}^{j}}$, input size $\widehat{S_{inp}^{j}}$, and time cost $\widehat{T}$. Table 5.7 summarises the additional notation.

5.3.1 Detecting Deviations

Spotting deviations from the expected selectivity $\widehat{\sigma}$, result size $\widehat{S}$ and completion time $\widehat{T}$ is supported by the framework in a straightforward manner. From Table 5.1 we have:

$$\sigma = \begin{cases} \frac{n}{n_{inp}^{1} \cdot n_{inp}^{2}}, & \text{for binary operators} \\ \frac{n}{n_{inp}^{1}}, & \text{otherwise} \end{cases} \tag{5.1}$$

$$S = sum(s) \tag{5.2}$$

$$T = \begin{cases} sum(t_{tuple}), & \text{or} \\ t_{real} \end{cases} \tag{5.3}$$

After the operator has finished its execution, these values can then be compared against the initials estimates, i.e., $\widehat{\sigma}$, $\widehat{S}$ and $\widehat{T}$, respectively, in order to assess their accuracy.

Operator	**Selectivity** σ	**Result Size** S	**Completion Time** T
seq. scan	$\sigma(n, n^1_{inp}) = \frac{n}{n^1_{inp}}$	$S(\sigma, \widehat{S^1_{inp}}) = \sigma \cdot \widehat{S^1_{inp}}$ [S1], or $S(\sigma, \widehat{n^1_{inp}}, s) = \sigma \cdot \widehat{n^1_{inp}} \cdot avg(s)$ [S2]	$T(t_{map}, t_{pred}, t_{page}, \widehat{n^1_{inp}}, \widehat{n_{pages}}) = (avg(t_{map}) + avg(t_{pred})) \cdot \widehat{n^1_{inp}} + avg(t_{page}) \cdot \widehat{n_{pages}}$ [T1], or $T(t_{tuple}, \widehat{n^1_{inp}}) = avg(t_{tuple}) \cdot \widehat{n^1_{inp}}$ [T2], or $T(t_{real}, t_{lasttuple}, n^1_{inp}, \widehat{n^1_{inp}}) = t_{real} + t_{lasttuple} \cdot (\widehat{n^1_{inp}} - n^1_{inp})$ [T3]
hash join	$\sigma(n, n^1_{inp}, n^2_{inp}) = \frac{n}{n^1_{inp} \cdot n^2_{inp}}$	$S(\sigma, \widehat{n^1_{inp}}, \widehat{n^2_{inp}}, \widehat{S^1_{inp}}, \widehat{S^2_{inp}}) = \sigma \cdot \widehat{n^1_{inp}} \cdot \widehat{n^2_{inp}} \cdot (\widehat{S^1_{inp}} + \widehat{S^2_{inp}})$ [S1], or $S(\sigma, s, \widehat{n^1_{inp}}, \widehat{n^2_{inp}}) = \sigma \cdot avg(s) \cdot \widehat{n^1_{inp}} \cdot \widehat{n^2_{inp}}$ [S2]	$T(t_{hash}, t_{conc}, t_{pred}, n_{pairs}, \widehat{n^1_{inp}}, \widehat{n^2_{inp}}) = avg(t_{hash}) \cdot (\widehat{n^1_{inp}} + \widehat{n^2_{inp}}) + (avg(t_{pred}) + avg(t_{conc})) \cdot n_{pairs}$ [T1], or $T(t_{real}, t_{lasttuple}, n^2_{inp}, \widehat{n^2_{inp}}) = t_{real} + t_{lasttuple} \cdot (\widehat{n^2_{inp}} - n^2_{inp})$ [T3]
project / op. call	$\sigma() = 1$	$S(\widehat{n^1_{inp}}, s) = \widehat{n^1_{inp}} \cdot avg(s)$ [S2]	$T(t_{tuple}, \widehat{n^1_{inp}}) = avg(t_{tuple}) \cdot \widehat{n^1_i}$ [T2], or $T(t_{real}, t_{lasttuple}, n^1_{inp}, \widehat{n^1_{inp}}) = t_{real} + t_{lasttuple} \cdot (\widehat{n^1_{inp}} - n^1_{inp})$ [T3]
unnest	$\sigma(n, n^1_{inp}) = \frac{n}{n^1_{inp}}$	$S(\widehat{n^1_{inp}}, s) = \widehat{n^1_{inp}} \cdot avg(s)$ [S2]	$T(t_{tuple}, \widehat{n^1_{inp}}) = avg(t_{tuple}) \cdot \widehat{n^1_{inp}}$ [T2], or $T(t_{real}, t_{lasttuple}, n^1_i, \widehat{n^1_{inp}}) = t_{real} + t_{lasttuple} \cdot (\widehat{n^1_{inp}} - n^1_{inp})$ [T3]

Table 5.8: Prediction formulas exemplifying how the monitoring information can support predictions in AQP.

5.3.2 Predicting Deviations

If the overall goal is to predict, rather than simply detect deviations, the monitoring framework should provide the necessary input to the prediction mechanism. Table 5.8 gives examples of prediction formulas that use the monitoring information and can be applied for that purpose.

The prediction formulas about the final output size belong to two categories: firstly, when the operator does not change the size of the tuple (i.e., the average size of the input tuples is equal to the average output size) and the initial estimate of the input size is correct (S1); and, secondly, when the size does change or the initial estimate is inaccurate (S2). For the final time cost, three approaches have been considered: firstly, to decompose the operator function into subfunctions, such as those in the cost model used (if this is possible), and to use cost information about these subfunctions obtained up to that point, which implies the most detailed measurements (T1); secondly, to build the prediction on the cost of the operator up to that point assuming that the elapsed time is proportional to the number of input tuples, which, intuitively, cannot perform well when system parameters change (T2); and thirdly, to base the prediction on the cost of the last tuple (or of the n last tuples) processed, which, again intuitively, can adapt better to load fluctuations, but may be unduly affected by temporary load changes (T3).

5.3.2.1 Monitoring Sequential Scans

Based on the measured cardinalities of the input and output at a given point in the execution, the final selectivity can be estimated, e.g., as in Table 5.8. A new estimate for the final cardinality of the result can be obtained by multiplying the monitored selectivity by the known cardinality of the stored extent $\widehat{n^1_{inp}}$. The total size of result can be predicted in both ways mentioned in the previous paragraph. For the estimation of the total execution time, all three ways considered in the previous paragraph can be applied. In the first one, which requires the identification of simpler operator subfunctions, one can follow the approach of [SPSW02], where the cost can be divided into the cost for transforming the format of the tuples (if necessary), evaluating the predicates, and reading the pages:

$$T = t_{page} \cdot n_{pages} + (t_{map} + t_{pred}) \cdot n^1_{inp}$$

All these parameters can be monitored (Tables 5.2 and 5.3). Different implementations are expected to vary significantly as to their cost, and the contribution of each of the three common subcosts to the total execution time. Thus, monitoring at a finer level may be important in order to identify and quantify such differences.

5.3.2.2 Monitoring Hash Joins

The approach for predicting the final values of the selectivity and the size of the output of a hash join is similar to the one used for scans. The main difference between scans and joins is that the cardinalities of the inputs are more likely to be estimated rather than measured and building estimates on previous estimates may result in compound errors [IC91]. The total execution time of a hash join consists of the time to hash the tuples for both inputs, the time to concatenate all the relevant pairs of tuples and the time to evaluate the join predicate:

$$T = t_{hash} \cdot (n^1_{inp} + n^2_{inp}) + (t_{pred} + t_{conc}) \cdot n_{pairs}$$

The initial estimate of the cost of a hash join at compile time uses a constant value for the time required to hash a tuple. This constant can be monitored (Table 5.4) and thus can be corrected in case it is not accurate. Another constant is used for the time to concatenate two tuples, which is also error prone. The number of pairs of tuples concatenated is more difficult to estimate. The optimiser can make a simple assumption that there is a uniform distribution of tuples across the hash table buckets. Another option, if the left input has already been consumed, is to use the time elapsed along with the time taken to evaluate the last tuple, as shown in the second formula in the relevant field of Table 5.8.

5.3.2.3 Monitoring Projects, Unnests and Operation Calls

For projections and operation calls, the cardinality of the output is the cardinality of the input, as their selectivity is always equal to 1. The size and time prediction formulas resemble those for scan (Table 5.8). Unnests differ in that they may have a selectivity greater than 1.

5.3.2.4 Monitoring Exchanges

The time cost of an instance of exchange is the sum of the costs to receive packed tuples from remote nodes, unpack them, pack tuples into a buffer, and send the packed tuples to other nodes. The communication cost is the dominant cost.

The actual values of cardinality, size of the output, and time cost of the operator can be monitored. However, no more accurate estimates for the number of tuples to be produced can be made at runtime without communication of monitoring information across a network, other than the estimates made by the optimiser at query compile time. This is because this metric depends on the number of tuples that other, remote instances of exchange send to that node, and in order to get this information, data transmission over network is required. For this reason, exchanges are not considered in Table 5.8.

On-the-fly updating of the predictions for the output size and the time cost can occur in a limited range of situations. A better estimate of the size can be made if the observed average tuple size is different than the estimated one, but again the information about the total number of result tuples is missing.

More accurate estimates of the time cost are produced by adding together more accurate estimates of the component costs. The time cost to transmit data depends on the input cardinality, the average size of a tuple, and the network speed between the two nodes involved. It also depends on system parameters that are not expected to vary for a given system, such as the size of a buffer and the space overhead for each buffer. From the above three variables, more accurate values can be obtained for the average tuple size. If this size remains the same, the system cannot detect deviations from expected cost caused by fluctuations in the network bandwidth, or from the expected number of incoming tuples. Changes in the expected cardinality of the input and the output are tracked, but cannot be predicted on the fly.

5.3.2.5 Generalisation for other operators

The prediction formulas for scans and unnests that only use notation from Tables 5.1 and 5.7 can be generalised for any operator with selectivity different from 1. The formulas for projections can be generalised for any operator with selectivity equal to 1. The formulas for hash joins that do not use notation from Table 5.4 can be applied to any binary operator. In general, the formulas used here are simple and may not be appropriate in all usage scenarios. It is not the aim of this thesis to explore their validity

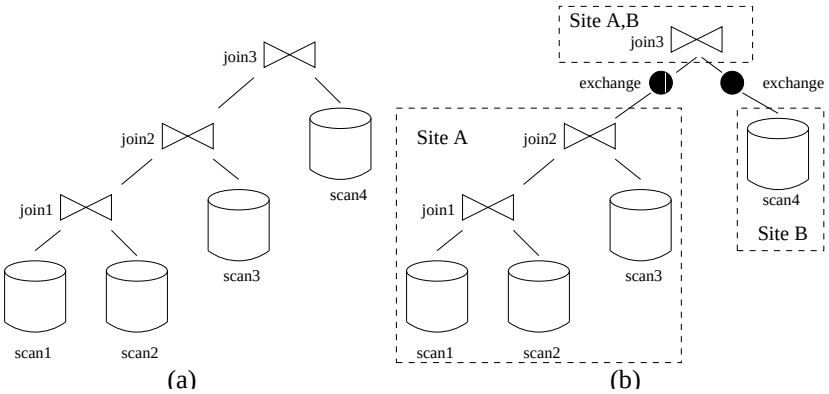


Figure 5.1: Example query plans executed over (a) a single machine, and (b) two machines

over more diverse usage scenarios. The role of such formulas in the monitoring task is to provide feedback for adaptive query processors. Although the performance criteria were defined to be the selectivity of operators, the size of (intermediate) results, and the time cost, there is no fundamental reason why this set cannot be extended and tailored to different system characteristics.

5.3.3 Propagating monitoring information

As the monitoring information is created within the scope of operators, communication of monitoring information between operators even in different machines can occur in the same way as the data items manipulated by operators are exchanged (e.g., through the exchange operator [Gra90] in the operator model of parallel execution), without requiring the development of specific mechanisms for this purpose. Section 5.3.2 examined the case of monitoring without communication overhead. This section shows how relaxing this constraint can enhance monitoring precision. Firstly, the case in which data is not transmitted to remote nodes is discussed, then the case in which monitoring data is shared among different nodes is discussed. In the first case, the communication overhead can remain low, as the information does not have to be conveyed through the network. Actually, it may not need be passed between operators physically at all, but simply recorded for access by later operators.

5.3.3.1 Sharing information among different operations in a node

When lower operators in the query plan propagate more accurate estimates to operators that lie above them, estimates for the latter become more accurate. The formulas in Table 5.8 allow on-the-fly predictions of the final number of tuples, the final size of the result and the final time cost. These formulas depend on initial estimates of the input cardinality ($\widehat{n^1_{inp}}$ and/or $\widehat{n^2_{inp}}$) and size ($\widehat{S^1_{inp}}$ and/or $\widehat{S^2_{inp}}$). Monitoring allows more accurate estimates of these properties. The input size and the input cardinality of an operator are the output size and the output cardinality of its children, respectively. All physical operators, except exchange, are able to produce more accurate predictions for these two metrics. This function operates in a recursive way that results in the propagation of better estimates from the lowermost to the topmost operator provided that an exchange operator does not break that chain.

Consider the query plan in Fig. 5.1(a). For each join, the expected cardinalities of the inputs are computed at compile time. Even if the selectivities of the three joins are estimated with the same accuracy, the estimate for the output of the third join can be much worse than the estimate for the second join and even worse than for the first one. [IC91] explains how propagation of errors affects the quality of these estimates. All operators can continuously update their expected output cardinalities and selectivities if monitoring is in place. The propagation of these measurements results in the third join having an up-to-date estimate for its inputs. These inputs also have up-to-date estimates for their inputs and so on. In that way, the effect of potentially inaccurate initial estimates can be ameliorated.

The formulas of Table 5.8 remain the same. However, for each operator the values $\widehat{n^j_{inp}}$ and $\widehat{S^j_{inp}}$ are replaced with the relevant predictions of its child.

5.3.3.2 Sharing information among different nodes

In the previous example, assume now that the fourth scan is placed on another node, and that the third join is evaluated through partitioned parallelism on both sites. In the operator model of parallelism, tuples are exchanged between nodes through the exchange operator (Fig. 5.1(b)). If no communication across sites is permitted for monitoring, exchanges cannot give up-to-date estimates. In this case, the third join can only use the initial estimates computed at query compile time. However, if there is no zero-communication constraint, the monitored information can be transmitted to and across exchanges. In this way, each instance of exchange can predict on the fly the total

number of buffers and the number of tuples that will be sent to each consumer. New estimates of the output cardinalities can be produced by gathering this information from all the exchanges.

Allowing monitored information to be transmitted over the network has additional benefits. The relative workload of the nodes can be monitored by tracking and comparing the number of tuples each instance of an operator receives. Moreover, the connection speed between two nodes can be monitored by recording the time when a buffer is sent from a node and the time it arrives at its destination. Finally, the relative load between nodes can be monitored by tracking and comparing the average times to process a tuple on different sites. Hence, communication overhead can be traded for such benefits.

This approach to propagating the monitoring information through the query plan allows for adaptive schemes where operators adapt autonomously (e.g., [ZL97]) as well as approaches that co-ordinate the query re-optimisation centrally (e.g., [BFMV00b]).

5.3.4 Supporting Existing Adaptive Approaches

According to the feedback they collect from the query execution, AQP systems can be classified in three broad categories.

The adaptive systems that monitor the rate at which they receive their input belong to the first category. A typical example is the XJoin [UF00], a variant of pipelined hash joins that hides delays in the arrival of the input tuples by performing other operations when the inputs are blocked. In self-monitoring operators, the input tuple rate and the time waiting since the last tuple was processed can be monitored for each operator. Consequently, it can be inferred whether an input is blocked by using a threshold. Ginga [PLP02], Query Scrambling [UFA98], and Bouganim *et al* [BFMV00b] also deal with the problem of experiencing delays in the delivery of the first tuples from a remote source. [BFMV00b] proposed an approach that generalised Query Scrambling to adapt not only to blocked connections, but to any changes in the data delivery rates as well. To monitor the delivery rates, they employ a new component, whereas in the approach proposed in this thesis, this could be easily achieved within the operator. Information about the data delivery rates can trigger adaptation also in the context of Rivers [ADAT$^+$99], a proposal for parallel I/O intensive applications, which monitors the bandwidth between data producers (e.g., disks) and consumers (e.g. scan operators).

Another group comprises systems that focus on the workload and the productivity of operators measured in tuples. For example, Eddies [AH00], a very dynamic technique, encapsulates a multi-join, and dynamically chooses the order of the individual joins for each incoming tuple. The basic routing policy observes the number of tuples received by each join so far, and the number of tuples produced. Both these metrics are covered by the proposed approach, not only for the joins, but for all the operators (Table 5.1). Also, Flux [SHCF03] extends the traditional exchange operator to adapt to fluctuations in resource availability (like resource and memory loads) while executing a query in a pipelining mode. It relies on the on-the-fly selection of simple statistics like the number of tuples processed and the time the operator is active. In [UF01], the Dynamic Pipeline Scheduler tries to reduce the initial response time of the query, basing its adaptive behaviour on the number of the tuples consumed so far by the operators and on their selectivities.

More generic systems, in terms of the information they collect from a query plan, fall in the third category. Kabra and DeWitt [KD98] use a separate monitoring operator for collecting statistics about data on the fly, provided that this is possible in one pass of the input. Such statistics include the cardinality of intermediate results, their average size, and certain histograms. The Tukwila system [IFF+99] integrates adaptive techniques proposed in [KD98, UFA98]. A special operator is also used to switch to an alternative data source, when the initial source fails. The execution information that the system monitors for active operators is the number of tuples produced so far (to check whether the optimisers estimates were adequately accurate) and the time waiting since the last tuple was received (to identify slow or blocked connections). The Conquest query processing system resembles Tukwila in adopting a triggering approach to respond to runtime changes [NWMN99]. Characteristics related to query execution that can trigger actions include updates to operator selectivities and sizes of intermediate results. Also, the system monitors the load of the resources. Self-monitoring operators can infer relative levels of load by comparing the time to process a tuple at different points of the execution. Operator selectivities and sizes are monitored explicitly.

Table 5.9 summarises the aspects of self-monitoring in Table 5.1 that are used by the AQP systems examined (referring to specific operators). It demonstrates that the proposal is generic enough to support many adaptive systems with different functionalities and requirements. The approach presented integrates and extends existing monitoring approaches with regard to data characteristics and execution cost. In essence,

Systems	**Monitored Operators**	n	n^j_{inp}	t_{real}	t_{tuple}	s	t^j_{wait}
Bouganim *et al* [BFMV00b]	scan & scan's parent		√	√			√
Conquest [NWMN99]	any	√	√		√	√	
Dyn. Pip. Scheduler [UF01]	join	√	√				
Eddies [AH00]	join	√	√				
Flux [SHCF03]	exchange		√	√			
Ginga [PLP02]	scan & scan's parent		√	√			√
Kabra & DeWitt [KD98]	any	√				√	
Q. Scrambling [UFA98]	join & scan		√	√			√
River [ADAT+99]	scan		√	√			√
Tukwila [IFF+99]	any	√	√	√		√	√
XJoin [UF00]	join		√	√			√

Table 5.9: Monitored information that can provide input to existing AQP systems.

any of the above adaptive techniques can implement its assessment and response strategy on top of the monitoring framework presented.

5.4 Evaluation

The presentation of the experimental results in this section serves two purposes. Firstly, to provide insights into how large the overhead of monitoring and predicting can be, and, secondly, to assess the accuracy of the predictions based on monitoring. The data used in the experiments are from the OO7 benchmark [CDN93]. The measurements are taken on a dedicated PC with a 1.13GHz AMD Athlon CPU and 512 MB memory (of which 330 - 370 MB were available for query processing at the time of the measurements), running Redhat Linux 7.1. The query engine used is part of the Polar* Grid-enabled distributed query processor [SGW+02]. The operators are implemented in C++ according to the iterator model [Gra93] and all are single-pass, i.e., all intermediate data sets are stored in main memory, although the data starts off on disk. The granularity of the system's timer is one microsecond.

Operator	Characteristics
Scan A	average size of tuples is 155bytes
Scan B	average size of tuples is 727bytes
Scan C	average size of tuples is 2Kbytes
Scan D	average size of tuples is 20Kbytes
Hash-Join A	1 tuple per hash table bucket
Hash-Join B	10 tuples per hash table bucket
Hash-Join C	20 tuples per hash table bucket
Hash-Join D	200 tuples per hash table bucket
Project	project one tuple field out of 10
Unnest	fan-out is set to 3, average size of initial tuples is 155bytes

Table 5.10: The operators used in the experiments for monitoring overheads.

5.4.1 Overhead of Monitoring

The measurements fall in three categories: firstly, those that involve counters (e.g., cardinalities of input, output and hash table buckets); secondly, those that require timings, i.e., two timestamps are taken and their difference is computed (e.g., time to evaluate a tuple, time to change the tuple format from the storage format to the evaluator format), and thirdly, those that compute the size of a tuple. The size of the tuple is not statically known in two cases. Firstly, when the tuple has one or more tuple fields with string type of undefined length; and, secondly, when there is a collection attribute of undefined collection size. Measuring the size of a collection requires a counter. Measuring the size of a string of characters involves identifying the tuple fields in the tuple that are string-valued and computing the length of each.

Inserting a counter in an operator has a very small overhead, measured at 0.03 μsecs. The overhead of measuring timings is of the order of microseconds (1.11 μsecs). The time cost of measuring the size of a string of characters depends on the size of the string. For small strings, the overhead is small but for larger strings it can become several milliseconds (e.g., for a 1MByte string this takes 0.0044 secs).

The operators that were used in the experiments for monitoring overheads are shown in Table 5.10. All the joins are on a key/foreign key condition. Table 5.11 depicts the magnitude and relative importance of the overheads, as it shows the percentage increase in the cost of evaluating a tuple due to monitoring. For each of the operators in Table 5.10, the time cost is given (2nd column in Table 5.11). The last three columns show the increase in the cost when a counter, a timing and a character counter

Operator	**time (in $\mu secs$)**	**counter (%)**	**timing (%)**	**100-byte string (%)**
Scan A	16.82	0.18	6.60	70.63
Scan B	25.50	0.12	4.35	46.60
Scan C	48.81	0.06	2.27	24.34
Scan D	350.57	0.01	0.32	3.39
Hash-Join A	8.22	0.36	13.50	144.52
Hash-Join B	13.02	0.23	8.52	91.22
Hash-Join C	16.25	0.18	6.83	73.11
Hash-Join D	62.86	0.05	1.77	18.90
Project	0.89	3.39	125.27	1340.71
Unnest	10.16	0.30	10.92	116.88

Table 5.11: The overhead of taking measurements compared to the cost of the operators for each tuple processed (for the hash-joins, the cost is for each tuple of the input that probes the hash table).

for a 100-byte string are applied to each tuple processed, respectively. As expected, the relative overheads are higher for computationally-inexpensive operators, like project, and significantly lower for the computationally-expensive ones, like hash join. The overhead of a counter is negligible for all the operators. Placing two timestamps is more costly than projecting an attribute, but the percentage overhead is relatively low for other operators (between 0.32% and 13.5%). Measuring the size of a string has essentially no cost if the string is a few bytes long. If the length is 100 characters or more, the performance may degrade significantly. For instance, it may increase the cost of a hash join by up to 144%, when the size is monitored for each tuple processed by the operator. If the size is computed for one tuple in ten, the increase is only 14.4%, and if the frequency is 5% (one in twenty tuples is monitored), the increase falls to 7.2%. However, it is not usually necessary to compute this in a database setting, as often, the length of a string is stored explicitly. The values in Table 5.11 can inform the choice of monitoring frequency by indicating broadly the overhead that can be anticipated.

5.4.2 Overhead of Predictions

This part examines the overhead of predicting deviations, as discussed in Section 5.3.2. Here, only the scan operator is analysed, but a similar approach can be followed for the remaining operators. The results for all operators are shown in Table 5.12.

It is assumed that the system holds information about the size and cardinality of

the stored collections. The selectivity of an operator is given by $\sigma = \frac{n}{n^1_{inp}}$, where n and n^1_{inp} are the monitored cardinality of the output and the input up to that point of execution, respectively (Section 5.3.2.1). The output cardinality is predicted by multiplying the monitored selectivity with the known input cardinality. It requires two counters that are updated for each tuple and the evaluation of one formula. The formula may be evaluated at various frequencies, but it is processed in time significantly less than a microsecond. The cost of the two counters is of the order of nanoseconds ($0.03\ \mu secs$ each). So, the overhead of predicting the final number of tuples produced is some fraction of a microsecond. If the output tuples do not contain strings with variable length, the final output size is predicted by multiplying the monitored selectivity with the known size of the stored collection, and the overhead of this prediction is the overhead incurred by two counters as well. If the tuple produced does contain strings of undefined length, the total size is given by

$$S = \sigma \cdot \widehat{n^1_{inp}} \cdot avg(s)$$

where $avg(s) = \frac{sum(s) \cdot freq}{n^1_{inp}}$ and $freq$ specifies every how many tuples the tuple size s is monitored. The cost of making these predictions is essentially dominated by the cost of measuring the length of the strings.

Predicting the total time for completion of the operator involves one timing t_{tuple} being captured for each monitored tuple as follows:

$$T_{total} = avg(t_{tuple}) \cdot \widehat{n^1_{inp}}$$

The average overhead is $1.11\ \mu$secs for each monitored tuple, which is the cost of a single timing, plus the cost of updating a counter.

Table 5.12, which derives from the 3rd and the 4th column of Table 5.11, shows the relative overhead of making predictions. As the prediction of the output size depends on the size of the variable length strings (if any) and is not generic, it is not shown in the table. If there are no collection attributes or variable-length strings, then the cost is the same as the cost to compute the output cardinality.

Operator	**Overhead computing output cardinality (%)**	**Overhead computing operator time (%)**
Scan A	0.36	(6.6/*freq* + 0.18)
Scan B	0.24	(4.35/*freq* + 0.12)
Scan C	0.12	(2.27/*freq* + 0.06)
Scan D	0.02	(0.32/*freq* + 0.01)
Hash-Join A	0.72	(13.5/*freq* + 0.36)
Hash-Join B	0.46	(8.52/*freq* + 0.23)
Hash-Join C	0.36	(6.83/*freq* + 0.18)
Hash-Join D	0.1	(1.77/*freq* + 0.05)
Project	6.78	(125.27/*freq* + 3.39)
Unnest	0.60	(10.92/*freq* + 0.30)

Table 5.12: The percentage increase in the operator cost when predictions are made.

5.4.3 Accuracy of predictions

The formulas introduced in Section 5.3 are rather straightforward and may be expected to give better results when the system is uniform in terms of load, attribute value distribution, operator workload, etc. However, it is interesting to examine how large the deviations are when the formulas are applied to skewed data. Since all operators require initial estimates for their input cardinality, an error in that cardinality compromises the accuracy at exactly the same magnitude. Consequently, it is important that an operator not only is able to make accurate predictions about the cardinality of its result set, but also that it is able to pass that information on to its parent operator in the query tree, as described in Section 5.3.3.

Consider three scans. The first, scan1, has selectivity 10% and the tuples that satisfy the scan condition are spread in a uniform manner across its extent. The second, scan2, also has a uniform distribution, but the selectivity is 50%. The third scan, scan3, has a selectivity of 50%, and is satisfied by all but the first 25% and the last 25% of the tuples. Figure 5.2 shows how accurate the predictions for the output cardinality are at each stage in the process of query execution. The formulas used imply that the final prediction of the selectivity of the predicate is the same as the monitored selectivity at that point (Table 5.8). This means that the baseline (i.e., 0% deviation from an accurate prediction) is given either by the prediction when 100% of the query execution has completed, or by the monitored selectivity after the finish of execution. If the tuples that satisfy the predicate are distributed across the dataset in a uniform manner (e.g., scan1 and scan2) the accuracy is very high and not dependent on the

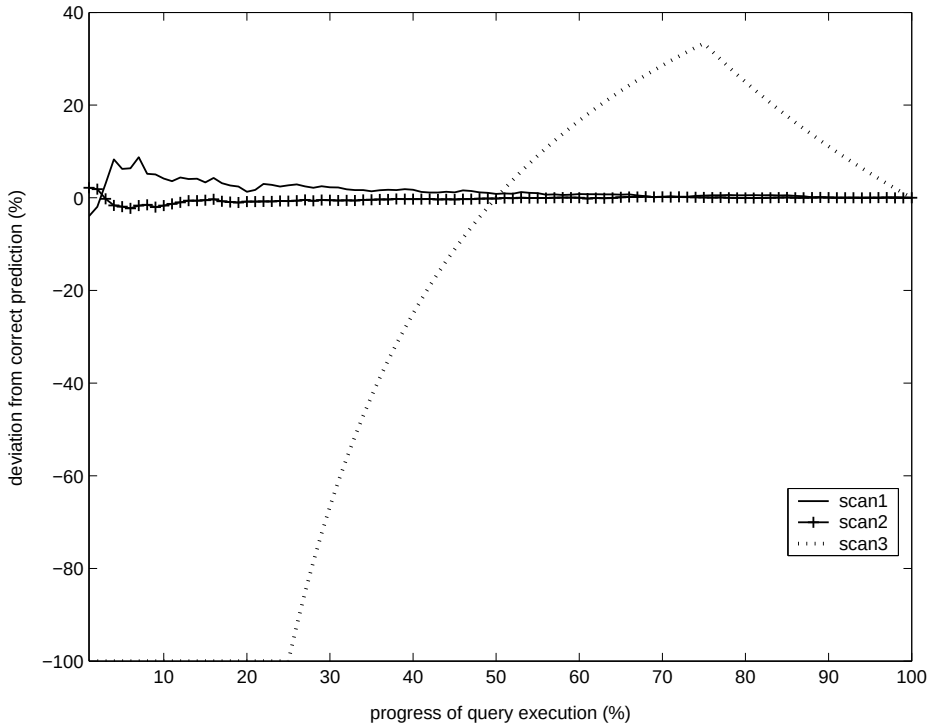


Figure 5.2: The accuracy of the predictions for the output cardinality of the three scans at different stages of the operator execution.

selectivity. However, for skewed distributions with unfavourable shapes (e.g., scan3), the predictions can be erratic over the course of execution. Figure 5.3 shows that the operator response time can be accurately predicted from the very early stages for all three scans, as well (every tuple is monitored and the machine load does not vary during execution in this experiment[2]).

5.4.4 General remarks on the evaluation

There are several lessons to be learned from the evaluation of the overheads related to monitoring:

1. In the approach proposed, there are three types of monitored information: counters, timings, and sizes of variable-length strings. The overhead of these three types is not dependent on the type of the query operator. The costs of counting and of computing a time interval are constant for a given system, whereas the cost of measuring the size of a string depends on its size.

[2] In [GPFS03], experiments are presented for more monitoring frequencies and load variations.

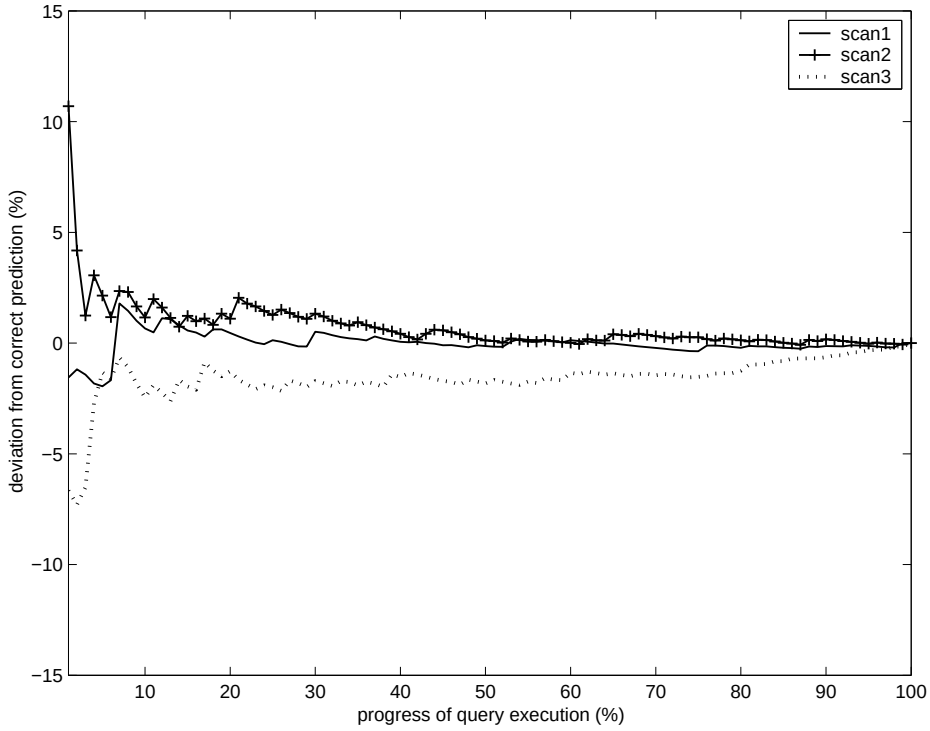


Figure 5.3: The accuracy of the predictions for the response time of the three scans.

2. The cost of a counter is negligible for all the operators examined. However, this is not true for timings and string computations.

3. The cost of computing the output cardinality is lower than 1% for all operators examined except project, for which it is 6.78%. So, it can be regarded as low. Additionally, the relative cost of predicting the final response time is lower than 13.5% for all operators except project, even if the time cost of each tuple is measured separately. If the time cost is measured at a frequency lower than 10% (i.e., one in ten tuples is timed), the cost becomes lower than 1.5% for these operators.

4. The results presented can be transferred to any *Select-Project-Join* query. This is because (i) such queries can be evaluated using the operators examined here; and (ii) the number of operators in the query plan does not have any impact on the monitoring cost as every operator is being monitored independently.

In general, the relative overhead incurred by monitoring remains low if the monitoring frequency of inexpensive operators (and the monitoring frequency of all operators

when the environment is stable) remains low. Notice that in multi-pass implementations of operator algorithms, where the data cannot fit entirely in main memory, the average cost of the operator is expected to be significantly higher, whereas the cost of monitoring is expected to remain the same. Consequently, in such systems the contribution of the monitoring cost to the total execution time is envisaged to be even smaller. Thus, the results presented here with respect to the proportional overhead of monitoring approximate the worst-case scenario, as other query processors are expected to behave either similarly to or worse than the query processor used, in terms of the monitoring overhead. For accurate predictions, a good knowledge of the input sizes and cardinalities is always required, which means that the children also need to be able to make good predictions and pass on relevant information to their parent. If the load of the system does not vary, the total response time can be predicted accurately from the early stages of execution. Skewed distributions impose significant errors but, even in such cases, predictions based on the very simple formulas presented can be better than direct usage of estimates produced at compile time.

5.5 Summary

So far, adaptive query processors have tended to use potentially efficient, but *ad hoc*, ways of collecting feedback from the environment and the query plan itself, analysing that feedback and choosing a reaction, all grouped together. It has been argued in this thesis that these three functions can be studied separately, in order to exploit the benefits of *divide-and-conquer* techniques and to gain generality, substitutability, and reusability. The main contribution of this section is the construction of a generic technique for monitoring the execution of query plans, based on self-monitoring query operators, with a view to employing this technique for adaptive query processing on the Grid.

The main features of the approach presented are:

1. The approach is generic in the sense that it does not depend on any particular adaptive system or form of adaptation.

2. The approach is capable of driving adaptations. It is able to collect information that is directly relevant to the assessment process of adaptivity by establishing where a plan is deviating from its anticipated behaviour, as it can identify and

predict erroneous initial estimates on the fly. In other words, it can provide the necessary background for on-the-fly adaptation, as discussed in Section 5.3.

3. The approach is comprehensive. It covers a broad range of query execution aspects, although it is based only on counters, timings, and size computations, as discussed in Section 5.2. Moreover, it provides monitoring information that is sufficient to support most AQP proposals to date, as demonstrated in Section 5.3.4.

4. The approach can be easily implemented and incorporated in existing query processors. It can be easily implemented as it employs only counters, timestamps and tuple size computations. It can be easily integrated into existing query engines, as it does not require changes in the architecture or in the internal logic of the query optimiser. Thus modifications are required neither in the engine architecture nor in the optimiser, but only in the implementation of the operators themselves, which is deemed to be less disruptive to the query compilation and evaluation architecture.

5. The approach can scale well with the number of machines used in query execution because there exists no central monitoring point. For the same reason, it is not affected by the size of a query plan in terms of the number of operators comprising the latter. Consequently, it fits better into distributed environments with potentially large numbers of nodes.

6. It accommodates different levels of detail in the monitoring information (i.e., operator-independent vs. operator-specific monitoring), monitoring frequency and data movement. In particular, this section discusses instantiations of the approach in which (i) no monitoring data is passed between the operators of the algebra, (ii) monitoring data is passed between operators of the algebra only within a single computational node, and (iii) information is passed between computational nodes in a distributed plan. Thus the approach is able to trade monitoring quality for monitoring overhead, as discussed in Section 5.3.3.

7. Finally, the monitoring approach was experimentally evaluated. As the *cost* of monitoring and the *quality* of the results obtained by monitoring are important, experiments have been conducted on both these features, which are presented in Section 5.4. This cost cannot be described as low or high as there is no general

consensus on these terms, but it is felt that the overheads incurred are reasonable and our results encouraging. In addition, the experimental results provide insights into how the frequency and the intensity of monitoring impact on its cost.

The material of this section has appeared in [GPFS04], and is discussed in more detail in [GPFS03].

Chapter 6

Adapting to Changing Resources

In Chapter 4, with a view to making the development of AQP systems more systematic, an implementation strategy was suggested, in which monitoring, assessment and response can be clearly distinguished. In the same chapter, it was identified that adapting to changing resources is a field that has not been sufficiently explored in AQP to date, especially in wide-area environments. Also, in Chapter 5, it was discussed how self-monitoring operators can provide up-to-date information about the query execution to drive adaptations.

The aim of this chapter is to bring all these together and, in particular, to present a complete instantiation of the adaptivity framework presented in Chapter 4 in the context of the OGSA-DQP query processor that was introduced in Section 2.5. The resulting adaptive prototype employs self-monitoring operators, and is capable of

- adapting to workload imbalance by repartitioning data (and operator state when required); and
- reacting to changes in the resource pool, by employing new machines that become available.

The remainder of the chapter is structured as follows. Related work is in Section 6.1. The extensions to the static OGSA-DQP system in order to transform it into an adaptive one are presented in Section 6.2. Sections 6.3 and 6.4 demonstrate adaptations to workload imbalance and resource availability, respectively. Section 6.5 concludes.

6.1 Related Work

Query processing on the Grid is a special form of distributed query processing over wide-area autonomous environments. Work in this area has resulted in many interesting proposals such as ObjectGlobe [BKK+01], Garlic [JSHL02] and Mariposa [SAL+96], but has largely ignored the issues of intra-query adaptivity.

Adaptive query processing is an active research area [BB05]; solutions have been developed to compensate for inaccurate or unavailable data properties (e.g., [AH00, KD98]), manage bursty data retrieval rates from remote sources (e.g., [Ive02]), and provide prioritized results as early as possible (e.g., [RRH99]). However, as already identified in Chapter 4, they usually focus on centralised, mostly single-node query processing, and do not yet provide robust mechanisms for responding to changes in the available resources, especially when an arbitrarily large number of autonomous resources can participate in the query execution, as it is the case in Grid query processing.

As an example that does consider distributed settings, [IHW04] deals with adaptations to changing statistics of data from remote sources, whereas the proposal of the current thesis, complementarily, focuses on changing resources. Moreover, sources in [IHW04] only provide data, and do not otherwise contribute to the query evaluation, which takes place centrally. Eddies [AH00] are also used in centralised processing of data streams to adapt to changing data characteristics (e.g., [CF03]) and operator consumption speeds. When Eddies are distributed, as in [TD03, ZOTT05], such consumption speeds may indicate changing resources. Distributed Eddies in [ZOTT05] can perform inter-operator balancing, whereas the focus here is on intra-operator balancing; in this dimension, the two proposals complement each other. Nevertheless, the approach proposed in this thesis is more generic as (i) it is not clear how distributed Eddies in [TD03] can extract the statistics they need in a wide-area environment, and how they can keep the messaging overhead low; (ii) Eddies cannot handle all kinds of physical operators (e.g., traditional hash joins); (iii) redistribution of operator state is not supported; and (iv) Eddies are difficult to encapsulate mechanisms for types of adaptation other than tuple routing, such as dynamic resource allocation. Adapting to changing data properties has also been considered in distributed query processing over streams [CBB+03]. Some forms of reoptimisation of parallel plans, including using new machines on the fly, are presented in [NWMN99]. However, this approach is not generic since it can be applied only to a limited range of unary operators. In [Ive02], substitution of data sources on the fly is supported to tackle data source failure; the

approach presented in this thesis is more generic as it can cover resources that provide both data and computations, and can adapt not only to failures.

For data and state repartitioning, the most relevant work is the Flux operator for continuous queries [SHCF03], which extends exchanges and uses partitioned parallelism. However, the Flux approach is more suitable for more tightly coupled architectures (such as shared-nothing parallel machines) in which synchronous communication is more realistic, as it assumes low latency of messaging, and feasible direct migration of operator state across processors. Thus it cannot be applied easily to a Grid setting. Further, it is not extensible to support other kinds of adaptations, such as increasing the degree of parallelism dynamically. Rivers [ADAT+99] follow a simpler approach, and are capable of performing only data (but not state) repartitioning. State management has also been considered in [DH04], but only with a view to allowing more efficient, adaptive tuple rerouting within a single-node query plan. Finally, [ZRH04] has examined possible operator state management techniques to be used in any single-node adaptation.

6.2 Grid Services for Adaptive Query Processing

The adaptivity framework described in Chapter 4 has been implemented as an extension to OGSA-DQP [AMG+04]. This section presents a summary of the extensions made, whereas the following sections demonstrate how the extensions are used to achieve adaptations to changing resources. The overall architecture is presented in Figure 6.1. The *MonitoringEventDetector* component instantiates the monitoring component of the framework (see also Figure 4.1) and integrates the raw events produced by the query engine. The *Diagnoser* performs the assessment phase, i.e., establishes whether there is an issue with the current execution. The *Responder* is notified of any such issues and chooses how to react. In the adaptations examined in this chapter, a single *Diagnoser* and a single *Responder* are sufficient, whereas multiple *MonitoringEventDetectors* are required, one at each evaluation site.

In summary, the execution engine produces notifications that include the per tuple cost of individual operators (or sets of operators) in the query plan, or the per tuple cost of sending data from one evaluator to another, or information about the amount of memory available and the current machine load. These notifications are processed by the *MonitoringEventDetector*, which notifies the *Diagnoser*, when substantial value

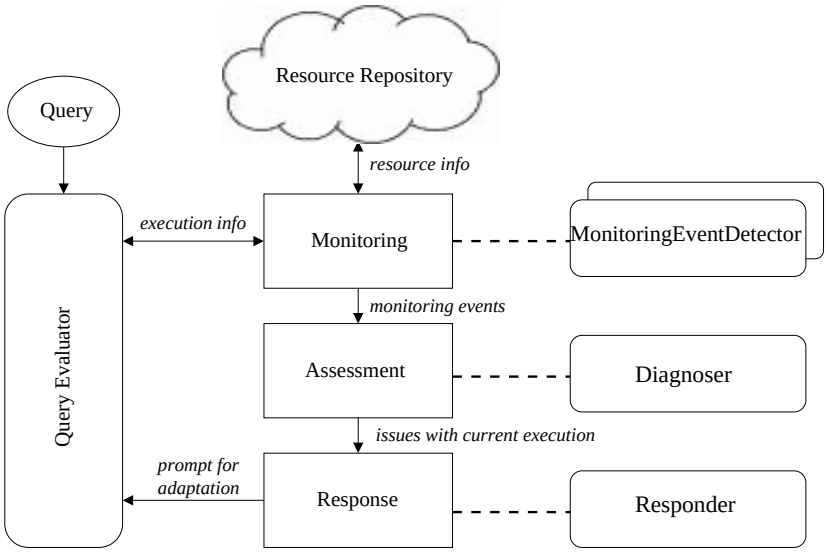


Figure 6.1: The architecture of the instantiation of the adaptivity framework.

changes have been detected. The *Diagnoser* is capable of diagnosing two issues, imbalanced workload distribution across evaluators or insufficient number of evaluators, and of notifying the *Responder* accordingly. The *Responder*, when it decides to react, it does so in two ways: either by modifying the workload distribution, or by changing the machine allocation. For both cases, appropriate messages are sent back to the evaluation engine. Also, the components can exchange subscription messages. The notifications mentioned can be combined in various ways, to implement several adaptation strategies and to address different issues. For instance, changing machine allocation may be the performed to boost performance, or to replace a failed machine. The exact protocols will be presented later for each of the two types of adaptations examined in this thesis separately.

Certain measures have been taken in order to avoid unstable conditions and flooding the system with messages unnecessarily. Firstly, small value changes are not propagated to the *Diagnoser*, and small changes in the workload distribution are not sent for further consideration to the *Responder*. Moreover, the *Responder* may not react if the execution is close to completion or the interval between adaptations is short.

The extensions made to the static version of OGSA-DQP fall into three categories that will be examined separately:

1. extensions that are concerned with the role of GQESs, and the implementation of the framework components;

2. extensions to the evaluation engine, within GQESs; and

3. extensions to GDQS.

6.2.1 Adaptive GQESs

As discussed in Section 2.5, OGSA-DQP has been implemented over the Globus Toolkit 3 Grid middleware [GLO] in Java, apart from the compiler, which has been inherited from Polar* and has been implemented in C/C++. It provides two types of Grid Services to perform static query processing on the Grid, GDQS (Grid Distributed Query Service) and GQES (Grid Query Evaluation Service). A GDQS contacts resource registries that contain the addresses of the computational and data resources available, to update the metadata catalog of the system. It accepts queries from the users in OQL, which are subsequently parsed, optimised, and scheduled employing intra-operator parallelism (Section 2.4). The query plan consists of a set of subplans that are evaluated by GQESs. A GQES is dynamically created on each machine that has been selected by the GDQS's optimiser to contribute to the execution. GQESs contain the query execution engine. Data communication is encapsulated within the exchange operator [Gra90], which has been enhanced as described later. Inter-service tuple transmission is handled by SOAP/HTTP as in many Web/Grid Service applications, and, for performance reasons, it does not happen for each tuple separately, but the tuples to be transmitted are aggregated in buffers.

Adaptive GQESs (AGQESs) encapsulate the framework components for monitoring, assessment and response (Chapter 4). Each AGQES comprises four components: 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 Chapter 5 and in [GPFS04]. As such, the query engine is capable of monitoring its own behaviour, and of producing raw, 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 integrates the raw events produced by the query engine. It detects value changes (e.g., change in the average cost of an operator and the number of machines available). The *Diagnoser* establishes whether there is an issue with the current execution, such as workload imbalance. The *Responder* is responsible for the response phase. For each type of diagnosed issue that it can be notified of, it stores a corresponding response base containing all the possible responses to this issue. Its decisions may affect not only the local query engine, but any

```
<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 6.2: Notification schema definition.

query engine participating in the evaluation. The query engine is extended to be capable of handling such response messages. The adaptivity notifications and subscription requests are transmitted across AGQESs as XML documents over SOAP/HTTP.

It is important to note that the above approach implies that the GDQS optimiser need not play any role during adaptations, and the distributed AGQESs encapsulate all the mechanisms required to adjust their execution in a decentralised way.

6.2.1.1 Adaptivity Notifications

The set of types of notifications exchanged between components are described in the AGQES interface in XSD (*XML Schema Definition language*). The notification schema is shown in Figure 6.2, and consists of two parts:

- generic fields, which are common to all notifications;
- 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

```
AdaptivityComponent {
    public:
       Queue inputQueue;

    private:
       AdaptivityComponent[] subscribers;

       analyseNotification(Notification) {
       }

       sendNotification(Notification,subscribers) {
       }

       subscribe() {
       }

    while (true) {
       Notification not = inputQueue.getNotification();
       analyseNotification(not);
    }
}
```

Figure 6.3: Sketch of the interface of adaptivity components.

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.

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 (see Chapter 5), to denote that the performance of a physical machine has changed, to denote that a physical operator is partitioned across machines in an imbalanced way, and so on (Figure 6.2).

6.2.1.2 Implementation of Adaptivity Components

Figure 6.3 provides a sketch of the implementation of each adaptivity component, regardless its kind (i.e., *MonitoringEventDetector, Diagnoser, Responder*). Each adaptivity component exposes to other parts of the system only its queue for incoming messages. At runtime, in each component, a thread is running continuously in the background 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.

Each component conforms to a generic interface in which there are the following functions:

- *analyseNotification(Notification)*, for the analysis of input messages;
- *sendNotification(Notification)*, for publishing events; and
- *subscribe()*, for registering with other adaptivity 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 *analyseNotification(Notification)* function is implemented. This is in analogy to the iterator model of execution [Gra93], according to which the operators conform to the *open()-next()-close()* interface, and differ in the exact implementation of the above functions.

6.2.2 Extensions to the Evaluation Engine

The extensions to the evaluation engine are at two levels: (i) the execution engine has become capable of receiving adaptivity notifications that prompt for modifications of 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 behaviour dynamically.

The first type of extensions has been realised through the development of a function *analyseNotification()* within the query engine, similarly to the *analyseNotification()* functions in the adaptivity components. This entails that the evaluation engine receives adaptivity-related messages in a dedicated message queue, and provides the mechanisms for their correct interpretation.

Extensions to exchanges are more intrusive and are described in more detail below.

6.2.2.1 Extending Exchanges

Fault-tolerance capabilities [1] To achieve fault tolerance the system relies on a roll-back checkpoint-based protocol, which extends the approach in [CBB+03] (details

[1] The development of fault tolerance capabilities in the context of the adaptive OGSA-DQP is not part of the work of the thesis. However, this functionality is presented here, as it will be leveraged to perform some of the adaptations that are described later.

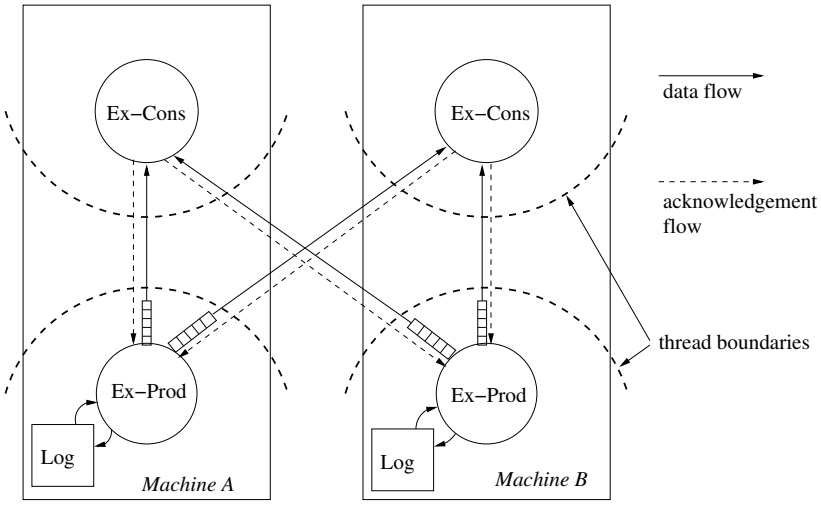


Figure 6.4: The enhanced exchanges

can be found in [SW04]). As mentioned earlier, exchanges comprise of exchange producers and exchange consumers (Ex-Prod and Ex-Cons in Figure 6.4, respectively). 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 the operators higher in the query plan, the checkpoints are returned in the form of acknowledgment tuples. Figure 6.4 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.

Monitoring Capabilities In the adaptive OGSA-DQP system, exchanges implement the self-monitoring approach introduced in Chapter 5. Moreover, they are capable of propagating such monitoring information as adaptivity notifications, which are subsequently integrated in the *MonitoringEventDetector* component.

Response Capabilities The interface of exchanges has been enhanced with the following functions:

- *setDataReDistribution()*, which applies to the producer thread of exchanges, and modifies the proportion of workload that each of the consumers receives.

```
<Partition>
 <ADAPTIVITY_CONFIGURATION>
   <!-- ADAPTIVITY_COMPONENTS ->
   ...
   <!-- QUERY_PLAN_INFO ->
   ...
 </ADAPTIVITY_CONFIGURATION>
 <Operator operatorID="2" operatorType="EXCHANGE">
  <tupleType> ... </tupleType>
  <EXCHANGE>
   <inputOperator> <OperatorID> ... </OperatorID></inputOperator>
   <consumers> ... </consumers>
   <producersNumber> ... </producersNumber>
   <producers> ... </producers>
   <arbitratorPolicy> ... </arbitratorPolicy>
   <EXPECTED_CARDINALITY> ... </EXPECTED_CARDINALITY>
  </EXCHANGE>
 </Operator>
</Partition>
```

Figure 6.5: An example of a plan fragment in adaptive OGSA-DQP.

- *addConsumer()*, which applies to the producer thread of exchanges, and adds a new exchange consumer.
- *addProducer()*, which applies to the consumer thread of exchanges, and adds a new exchange producer.

To implement a response, when a response notification received by the query engine, and analysed, one of the functions above may be called.

6.2.3 Extensions to the GDQS

In static 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 the 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.

Figure 6.5 shows an example excerpt of such documents, which can be compared against the excerpt in Figure 2.9 in Section 2.5, and in which the new fields are in capital letters. 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 make AGQESs capable of adapting 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. Thus, 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 (`adaptivity_components` field in Figure 6.5); and
- metadata about the global query plan (`query_plan_info` field in Figure 6.5).

6.3 Adapting to Workload Imbalance

6.3.1 Motivation and Problem Statement

Grid query processing, like many Grid computations, is likely to place a significant emphasis on high-performance and scalability. Traditionally, query processors often attain scalability by partitioning the operators within a query execution plan across multiple nodes, a form of parallelism commonly referred to as *intra-operator* or *partitioned* [Gra93]. In this way, all the clones of an operator evaluate a different portion of the same dataset in parallel. In Chapter 3, a novel algorithm has been introduced to schedule resources and to define the degree of partitioned parallelism for each part of the query, when there is a potentially huge pool of available resources. Orthogonally to the issue of resource scheduling, a basic difficulty in efficiently executing a query on the Grid is that the unavailability of accurate statistics at compile time and evolving runtime conditions may cause *load imbalance* that detrimentally affects the performance of the query execution. The execution of a plan fragment over a statically scheduled set of resources is considered as *balanced* when all the participating machines finish at the same (or about the same) time. Workload imbalance may be the result of uneven load distribution in the case of homogeneous machines. But in the

case of heterogeneous machines and the Grid, it might be the result of a distribution that is not proportional to the capabilities of the machines employed.

In the Grid, common problems stem from the different CPU characteristics of each machine contributing to the evaluation of a partitioned operator, the loads on machines (which are autonomous and may run many other jobs), the amount of memory available per node, the bandwidth of the connections between machines providing raw data, the cost of processing foreign functions, and so on. In a heterogeneous setting, loads, network bandwidth and available memory differ between machines, so a challenge for the query optimiser is to define intra-operator load-partitioning that takes into account these differences. If the necessary information is unavailable at compile time, the system needs to be able to extract it on the fly and adapt the execution accordingly. Failing to do so in an efficient way may, to a significant extent, if not totally, annul the benefits of parallelism. Just as in homogeneous, controlled environments (e.g., clusters of similar nodes), a slowdown in even a single machine that is not followed by the correct rebalancing, causes the whole system to underperform at the level of the slow machine [ADAT+99]. For long-running queries an additional challenge lies in the fact that properties like machine load, connection bandwidth and cost of operators are modified during execution. The result is that the per-tuple processing cost of each evaluator is constantly changing in an unpredictable way. To tackle this, the query processor needs not only to be able to capture these changes occurring in a wide-area environment, but also to respond to them in a comprehensive, timely and inexpensive manner by devising and deploying appropriate repartitioning policies.

Adaptive load balancing becomes more complicated if the parallelised operations store intermediate state, like the hash join and group-by relational operations (such operators are called 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.

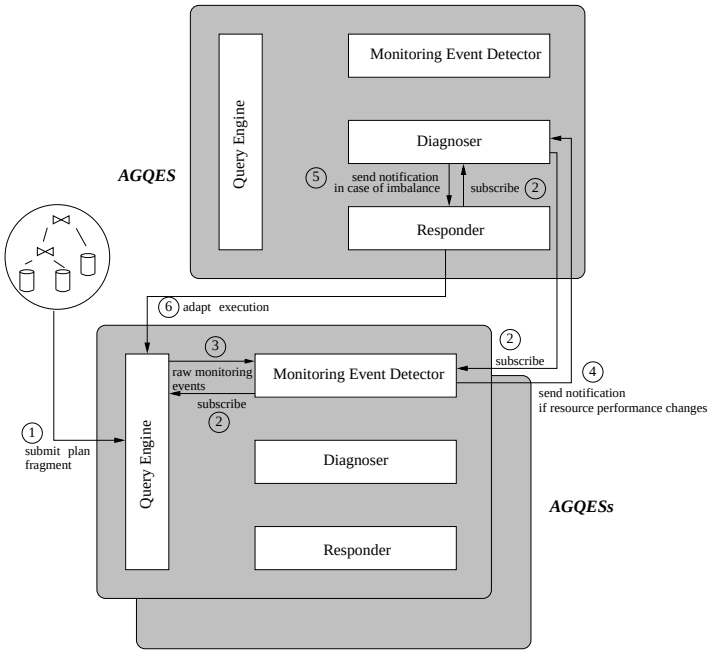


Figure 6.6: Instantiating the adaptive architecture for dynamic workload balancing.

6.3.2 Approach

To achieve workload balance during execution, the AGQESs are configured 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, globally accessible *Diagnoser* and *Responder*, that subscribes to the *MonitoringEventDetectors*. Further, it is assumed that the locations of the *Diagnoser* and *Responder* do not change during execution. Figure 6.6 presents this configuration, along with a summary of the messages exchanged and their order. Below, it is described in more detail how the monitoring, assessment, and response phases take place.

The configuration information sent by the GDQS to the AGQESs includes:

- The handles of all participating AGQESs. This information is considered only by the AGQES that holds the globally accessible *Diagnoser* and *Responder*, and is necessary to conduct the subscription.

- The sets of clone instances of exchange producers and their corresponding consumers, along with the identifiers of the operators that form the boundaries of subplan fragments, i.e., the operators that form the local root and the local leaf operators. This information is sufficient for obtaining complete knowledge on

```
<complexType name="adaptivityConfigurationType">
  <sequence>
    <!-- ADAPTIVITY_COMPONENTS ->
    <element name="AGQESHandle" maxOccurs="unbounded"/>
    <!-- QUERY_PLAN_INFO ->
    <element name="ExchangeSet" maxOccurs="unbounded">
      <complexType>
        <sequence>
          <element name="ProducerExchange" maxOccurs="unbounded"/>
          <element name="ConsumerExchange" maxOccurs="unbounded"/>
        </sequence>
      </complexType>
    </element>
    <element name="LocalConnection" maxOccurs="unbounded">
      <complexType>
        <sequence>
          <element name="SubplanRoot"/>
          <element name="SubplanRightMostLeaf"/>
          <element name="SubplanLeftMostLeaf"  minOccurs="0"/>
        </sequence>
      </complexType>
    </element>
  </sequence>
</complexType>
```

Figure 6.7: XSD for the adaptivity configuration metadata for dynamic workload balancing.

how data flows across subplans.

The corresponding XSD is shown in Figure 6.7.

6.3.3 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 their specific fields include (Figure 6.2):
  - the cost of processing an incoming tuple in milliseconds;
  - the average waiting time of the subplan leaf operator for this tuple in milliseconds, which corresponds to the idle time that the relevant thread has spent; and

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

Figure 6.8: Schema definition of notifications sent by the *MonitoringEventDetector* component.

  - 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:

  - the cost of sending a buffer in milliseconds;
  - the recipient of the buffer; and
  - the number of tuples that the buffer contains.

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

- groups the notifications of type M1 by the identifier of the operator that generated the notification, and the notifications of the type M2 by the concatenated identifiers of the producer and recipient of the relevant buffer;
- computes the running average of the cost over a window of a certain length, discarding the minimum and maximum values as outliers; and
- generates a notification to be sent to subscribed *Diagnosers*, if these average values change by a specified threshold *thresM*. The XSD of such a notification is shown in Figure 6.8, and covers both M1 and M2 analysis.

```
<complexType name="Notification">
    <sequence>
       <element name="description" value="DIAGNOSED_ISSUE"/>
       <element name="destinationId value="RESPONDER"/>
       <!-- rest 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>
```

Figure 6.9: Schema definition of notifications sent by the *Diagnoser* component.

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.

6.3.4 Assessment

The assessment is carried out within a *Diagnoser*. A *Diagnoser* gathers information produced by *MonitoringEventDetectors* to establish whether there is workload 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 processed tuple $c(p_i)$ for each such subplan, as explained earlier. Firstly, the *Diagnoser* identifies the different partition instances based on the configuration information it has received during initialisation (Figure 6.7). Also, the *Diagnoser* is aware of the current tuple distribution policy, which is represented as a vector $W = (w_1, w_2, \ldots, w_n)$, where w_i represents the proportion of tuples that is sent to p_i. To balance execution, the objective is to allocate a workload $w_i^{'}$ to each AGQES that is inversely proportional to $c(p_i)$. The *Diagnoser* computes the balanced vector $W^{'} = (w_1^{'}, w_2^{'}, \ldots, w_n^{'})$. However, it only notifies the *Responder* with the proposed $W^{'}$ if there exists a pair of w_i and $w_i^{'}$ for which $\frac{|w_i - w_i^{'}|}{w_i}$ exceeds a threshold *thresA*. This is to avoid triggering adaptations with low expected benefit.

Figure 6.9 shows the schema of the notifications published. The p_i fragment is defined by the `subplanRootOpId` and `subplanRootOpEvaluator` elements. `performanceIndicator` corresponds to the cost per tuple $c(p_i)$. w_i and $w_i^{'}$ are specified in `currentProportion` and `proposedProportion`, respectively.

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

- **A1**, which takes into account only the notifications of type M1 that are produced by the relevant subplan instance; or
- **A2**, which additionally takes into account the notifications of type M2 that are produced by the subplans that deliver data to the relevant subplan instance, and contain the communication costs for this delivery.

The default configuration is characterised by the following parameters:

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

6.3.5 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 this proposal, it contacts all the evaluators that retrieve store data to estimate the

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

Figure 6.10: Schema definition of notifications sent by the *Responder* component.

progress of execution in line with [CNR04]. According to this approach, the ratio of the tuples already produced by the evaluators that contact the store, and the expected final cardinality (which exists as annotation in the query plan), provides a good approximation of the progress of the query execution. The information about the number of tuples already produced is available from the *MonitoringEventDetectors*, and is provided as a response to a request of the *Responder* (`numProducedTuples` field in Figure 6.8).

If the execution is not close to completion, i.e., the progress has not exceeded a threshold *thresR*, it notifies the evaluators that need to change their distribution policy, which are the evaluators that send data to the imbalanced subplan, and the *Diagnosers* that need to update the information about the current tuple distribution (i.e, $W \leftarrow W'$). The former are identified based on the configuration information received during initialisation (Figure 6.7).

The schema of the notifications that are delivered to all the AGQES, which send data to the imbalanced subplan, is shown in Figure 6.10. This message is put in the relevant queue of the evaluation engine and not in the queues of the adaptivity components (the value of the `destinationId` is "AGQES_RESPONDER"). The analysis of the message leads to a call to the *setDataReDistribution()* function of the exchanges sending data to the problematic subplan (`modifiedOpId` in Figure 6.10). For each consumer of such exchanges, there is an entry of type `consumerReference` in the notification, and a corresponding entry of type `newProportion`, which specifies the new proportion that the relevant consumer is allocated to.

The recovery logs, deployed mainly to attain failure recovery, provide an opportunity to repartition operator state across consumer nodes by extracting the tuples stored in the recovery logs, and applying the data repartitioning policy to these tuples as well. Thus, 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. This redistribution is called *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. This redistribution is called *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 *hash-joins* that build hash tables, retrospective redistribution is the only valid option.

By allocating a number of tuples to each subplan instance that is inversely proportional to its cost per tuple, all the instances p_i of a subplan p are expected to finish at the same time (i.e., balanced execution), which results in better response times.

The default configuration of the *Responder* is not to react if more than 95% of the execution has completed, i.e., $thresR = 0.95$.

Figure 6.11 summarises the different combinations of assessment and response for the two types of monitoring information produced by the evaluation engine.

6.3.6 Evaluation

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

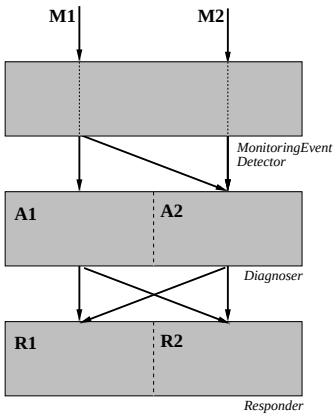


Figure 6.11: The flow of notifications across adaptivity components for dynamic workload balancing.

- in the presence of perturbed machines, the performance (i.e., response time) improves by several factors and the magnitude of degradation, in some cases, by an order of magnitude;
- the overhead remains low and no flooding of messages occurs; and
- the system can adapt efficiently even to very rapid changes.

Two example queries are used:

Q1:
```
select EntropyAnalyser(p.sequence)
from protein_sequences p
```

Q2:
```
select i.ORF2 from protein_sequences p,
protein_interactions i where i.ORF1=p.ORF;
```

The tables *protein_sequences* and *protein_interactions*, along with the *EntropyAnalyser* Web Service (WS) operation, are from the OGSA-DQP demo database[2] and they contain data on proteins and results of a bioinformatics experiment, respectively (*ORF, ORF1, ORF2* are all protein identifiers). The *protein_sequences* used in the experiments is slightly modified to make all the tuples the same length to facilitate result analysis. Q1 retrieves and produces 3000 tuples. It is computation-intensive

[2]The demo database is free for download from www.ogsadai.org.uk/dqp, as the rest of the OGSA-DQP system.

rather than data- or communication-intensive. The reason why a small dataset is chosen is that the current OGSA-DQP infrastructure is a service-based one, and so, data and message communication between services is XML-document-centric, which prevents high performance for big datasets. Thus, the adaptive prototype, which relies on SOAP/HTTP messaging, is more suitable for computationally expensive queries (e.g., queries that contain calls to complex data analyses). However, this is a temporary limitation as there are already proposals to develop high-performance message sending in SOAP/HTTP (e.g., [SSW04]), and has not inhibited the evaluation of the adaptivity mechanisms. Moreover, as shown in the experiments, Q1 is chosen in such a way that data communication and retrieval do contribute to the total response time. This contribution is even more significant in Q2 which joins *protein_sequences* with *protein_interactions*, which contains 4700 tuples (using a *hash join*, the hash table of which is copied across all instances to facilitate adaptations). So, Q1 and Q2 are complementary to each other: in the former, the most expensive operator is the call to the WS, and in the latter, a traditional operator such as join.

The adaptations described can be applied to an arbitrarily large number of machines. However, 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 for the evaluation of *EntropyAnalyser* in Q1, and the join in Q2, unless otherwise stated. The data are retrieved from a third machine. All machines run RH Linux 9, are connected by a 100Mbps network, and are autonomously exposed as Grid resources. It was ensured that they were unloaded at the time of the experiments in order to avoid result distortion. The third machine retrieves and sends data to the first two as fast as it can. The iterator model is followed, but the incoming queues within exchanges can fit the complete dataset. Thus, and due to the pipelined parallelism, the data retrieval is completed independently of the progress of the WS calls and joins. For each result, the query was run three times, and the average is presented here. Finally, two methods to create artificial load for machine perturbation have been: (i) by programming a computation to iterate over the same function multiple times, and (ii) by inserting *sleep()* calls.

6.3.6.1 Performance Improvements

This set of experiments demonstrates the capability of AGQESs to degrade their performance gracefully when some machines experience perturbations. Thus they exhibit significantly improved performance compared to static GQESs. In the first experiment,

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

Table 6.1: Performance of queries in normalised units.

the cost of the WS call in Q1 in one machine is set to be exactly 10 times more than in the other, and the responses are prospective (response type R2). The first row of Table 6.1 shows how the system behaves under different configurations. More specifically, the columns in the table correspond to the following cases:

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

The results are normalised, so that the response time corresponding to *no ad / no imb* is set to 1 unit for each query. The percentage of degradation due to imbalance is given by the difference of the normalised performance from 1. The adaptivity overhead is defined as the overhead incurred when adaptivity seems not to be needed (i.e., there is no imbalance)[3], which can be computed by the difference of the second and the third column of Table 6.1 (1st row). This difference is 5.9%. When one WS is perturbed and there are no adaptivity mechanisms, the response time of the query increases 3.53 times (4th column in Table 6.1). 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,

[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.

performing significantly better than without adaptivity (45% increase when adaptivity is enabled as opposed to 253% when it is disabled). This increase is just 17.8% of the increase without adaptivity.

The 2nd row in Table 6.1 shows the results when the experiment is repeated, and the adaptation is retrospective (type R1 of response). The increase in response time when the adaptivity is not enabled (*no ad / imb*) remains stable as expected (3.53 units). However, the average overhead (*ad / no imb*) is nearly three times more (15.3% of the execution). This is because it is now more costly to perform log management, as the tuples already sent to remote evaluators need to be discarded and redistributed in a tidy manner. Because of the larger overhead, the degradation of the performance in the imbalanced case (*ad / imb*) is larger than for prospective response (1.57 times from 1.45).

The same general pattern is observed for Q2 as well, using the second method to create imbalance artificially. In this case, the perturbation is caused in one machine by the insertion of a *sleep(10msecs)* call before the processing of each tuple by the join. The 3rd row of Table 6.1 shows the performance when the adaptations are retrospective. The overhead is 11%, and adaptivity, in the case of imbalance, makes the system run 1.31 times slower instead of 1.71.

Varying the Size of Perturbation Q1 is rerun for the cases in which the perturbed WS is 10, 20 and 30 times costlier, and adaptations are prospective. Figure 6.12 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.

Effects of Different Policies Thus far, the assessment has been carried out according to the type A1, in which communication cost is not taken into account. The next experiment takes a closer look at the effects of different adaptivity policies. Three cases are examined:

- 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);

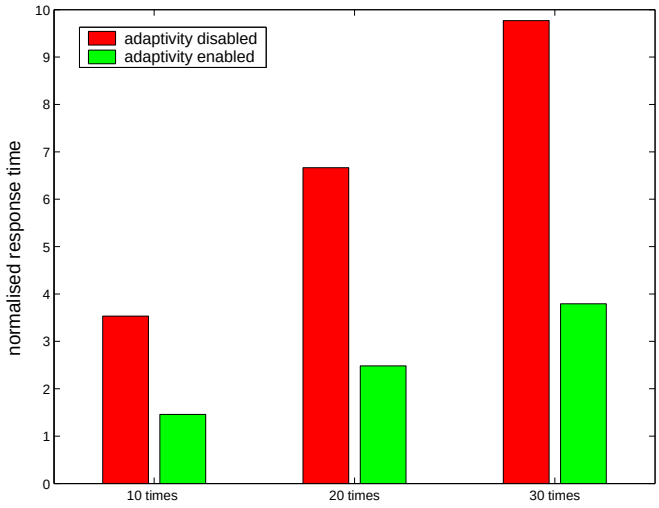


Figure 6.12: Performance of Q1 for prospective adaptations.

- 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
- when the *Diagnoser* does take into account the communication cost to send data to the subplan examined for imbalance, and no state is recreated (type A2 of assessment combined with type R2 of response).

In essence, when the communication cost is not considered (assessment A1), an assumption is made that the cost for sending data overlaps with the cost of processing data due to pipelined parallelism (i.e., assessment A1 takes into account the pipelining). It is believed that such an assumption is valid for the specific example queries, and indeed, this is verified by the experimental results discussed next.

The performance of the three configurations for Q1 is shown in Figure 6.13. Although all of them result in significant gains compared to the static system, some perform better than others. From this figure it can be observed: (i) that taking into consideration the pipelining by performing the assessment of type A1 has an impact on the quality of the decisions and results in better repartitioning (see the difference between the leftmost and the rightmost bar in each group); and (ii) that retrospective adaptations (R1 response) behave better than the prospective ones for bigger perturbations (see the difference between the leftmost and the middle bar in each group). The latter is also expected, as the overhead for recreating state remains stable independently of the size of perturbations, whereas the benefits of removing tuples already sent to the slower

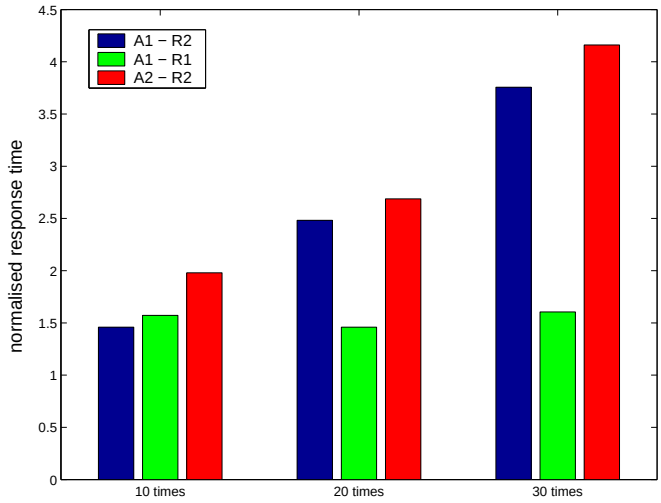


Figure 6.13: Performance of Q1 for different adaptivity policies.

consumers, and re-sending them to the faster ones increases for bigger perturbations.

Also, from Figure 6.13, it can been seen that the bars referring to retrospective adaptations remain similar with different sizes of perturbation, which means that the size of performance improvements increases with the size of perturbations. This happens for two complementary reasons: (i) the higher the perturbation, the more tuples are evaluated by the faster machine, in a way that outweighs the increased overhead for redistributing tuples already sent or buffered to be sent; and (ii) for any of these perturbations, only a very small portion of the tuples is evaluated by the slower machine, which makes the performance of the system less sensitive to the size of perturbation of this machine.

Experiments with Q2 lead to the same conclusions. Figure 6.14 shows the behaviour of the join query when the *sleep()* process sleeps for 10, 50 and 100 msecs, respectively, and adaptations are of type A1 of assessment and R1 of response. As already identified in Figure 6.13, retrospective adaptations are characterised by better scalability, and their performance is less dependent on the perturbation.

Table 6.2 corresponds to Figure 6.13 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

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

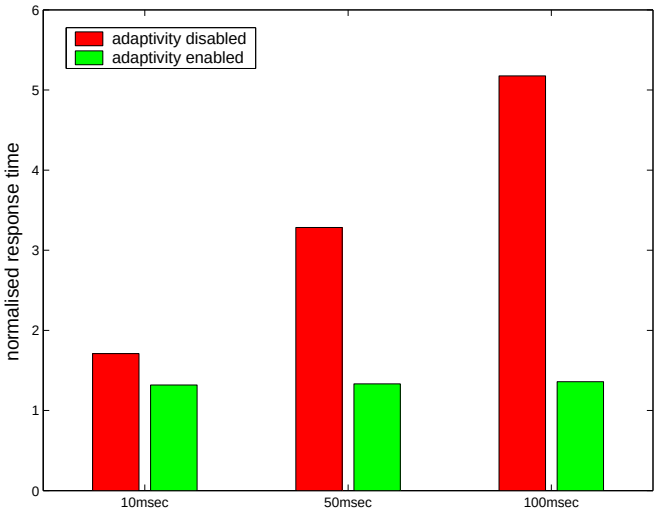


Figure 6.14: Performance of Q2 for retrospective adaptations.

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

Table 6.2: Ratio of tuples sent to the two evaluators.

effective. This is because the dataset is relatively small, and by the time workload 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 increases.

Varying the dataset size From the figures presented up to this point, retrospective adaptations outperform the prospective ones, but suffer from higher overhead. The reason why prospective adaptations exhibit worse performance is that a significant proportion of the tuples have been distributed before the adaptations can take place. Intuitively, this can be mitigated in larger queries. Indeed, this is verified by increasing the dataset size of Q1 from 3000 tuples to 6000, and making one WS call 10, 20 and 30 times costlier than the other, while the adaptations are prospective. Figure 6.15 shows the results, whose trends are more similar to those when adaptations are retrospective (i.e., Figure 6.15 is closer to Figures 6.13 for Q1 and 6.14 for Q2, compared to Figure 6.12), and lead to better performance improvements.

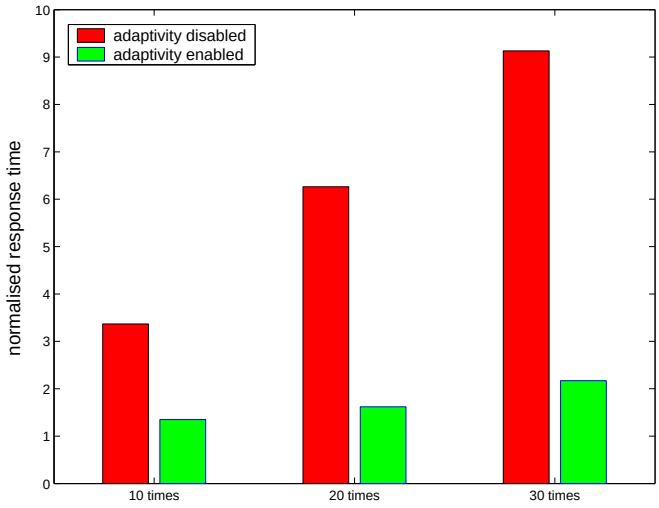


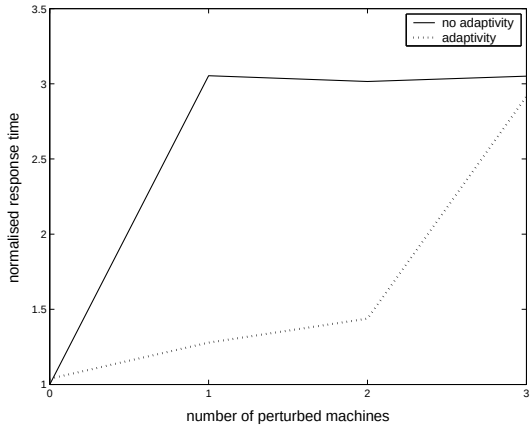
Figure 6.15: Performance of Q1 for prospective adaptations and double data size.

Case	A1-R2	Increase from Table 6.2
10 times	9.28	166%
20 times	9.01	182%
30 times	10.96	240%

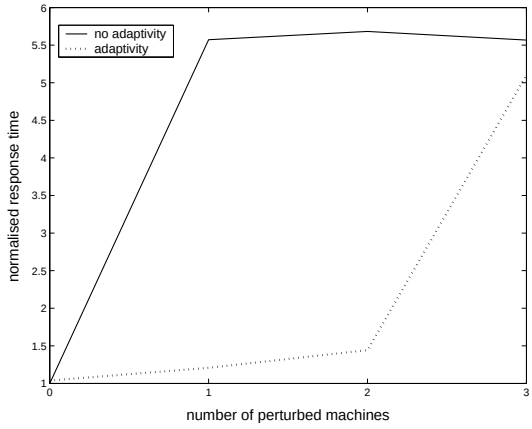
Table 6.3: Ratio of tuples sent to the two evaluators.

Table 6.3 shows the ratio of tuple distribution for this dataset, and the comparison with the relevant values in Table 6.2. Again, the difference between prospective and retrospective adaptations is much smaller than previously.

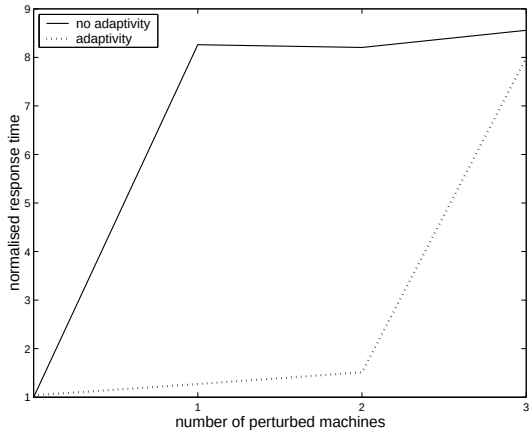
Varying the number of perturbed machines Figure 6.16 complements the above remarks by showing the performance of Q1 for different numbers of perturbed machines when adaptations are retrospective (three machines have been used for WS evaluation in this experiment). Again, perturbations are inserted by making one WS call 10, 20 and 30 times costlier than the other (Figure 6.16(a), (b) and (c), respectively). Due to the dynamic balancing property, the performance degrades very gracefully in the presence of perturbed machines. As explained in detail earlier, the performance when adaptivity is enabled, is very similar for different magnitudes of perturbation, when there is at least one unperturbed machine. Thus the plots corresponding to the case of enabled adaptivity are similar for up to two out of three perturbed machines. Note that the percentage of degradation (i.e., difference from value 1 in the figures) can be improved by an order of magnitude.



(a) 10 times



(b) 20 times



(c) 30 times

Figure 6.16: Performance of Q1 for retrospective adaptations.

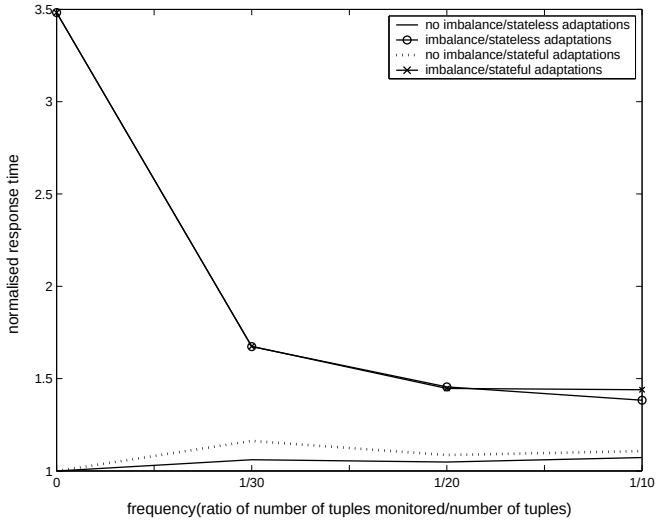


Figure 6.17: Effects of different monitoring frequencies in Q1.

6.3.6.2 Overheads

This set of experiments investigates overheads. Q1 is run when there is no WS perturbation. As shown from Table 6.1, the overhead of prospective adaptations is 5.9% (the *ad / no imb* performance value is 1.059 units). This value is the average of two cases. When the adaptivity mechanism is enabled but no actual redistribution takes place, the overhead is 6.2%. However, due to slight fluctuations in performance that are inevitable in a real wide-area environment, if the query is relatively long-running, the system may adapt even though the WSs are the same. For prospective adaptations, a poor initial redistribution may have detrimental effects, since by the time the system realises that there was no need for adaptation, the stored tuples may already have been sent to their destination. Nevertheless, on average, the system behaves reasonably with respect to small changes in performance and incurs a 5.6% overhead. The ratio of the number of tuples sent to the two machines is slightly imbalanced: 1.21. The overhead is slightly smaller than when no actual redistribution occurs as there are benefits from the redistribution.

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

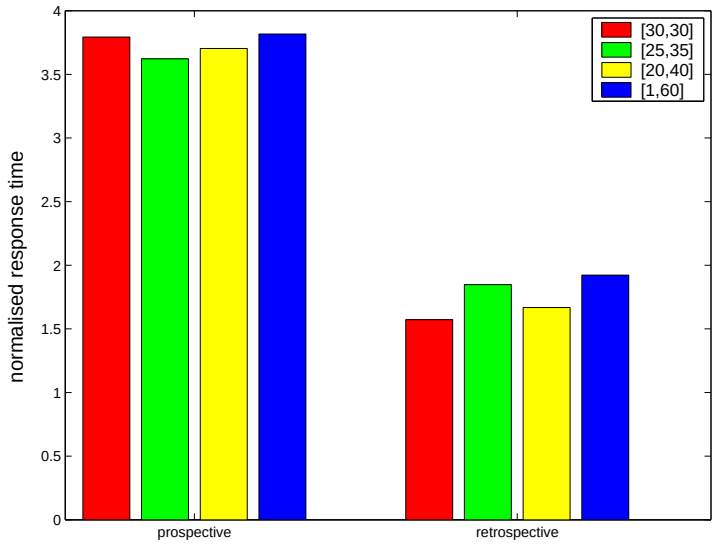


Figure 6.18: Performance of Q1 under changing perturbations.

are reasonable and are worth-while, given the scale of expected gains during perturbations.

Varying the monitoring frequency The behaviour of the system for Q1 is also examined 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 in Figure 6.17) and the overhead incurred (1st and 3rd plots in Figure 6.17) 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 (Section 5.4), 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.

6.3.6.3 Rapid Changes

The next set of experiments aims to show the dynamic nature of the system. Thus far, the perturbations have been stable throughout execution. A question arises as to whether the system can exhibit similar performance gains when perturbations vary in magnitude over the lifetime of the run. In these experiments the perturbation varies

Runtime perturbation	Difference from near-optimal static
[30,30]	10.2%
[25,35]	29.0%
[20,40]	16.4%
[1,60]	34.7%

Table 6.4: Ratio of tuples sent to the two evaluators.

for each incoming tuple in a normally distributed way, so that the mean value remains stable. Figure 6.18 shows the results when the differences in the two WS costs in Q1 vary between 25 and 35 times, between 20 and 40 times, and between 1 and 60 times The leftmost bar in each group in the figure corresponds to a stable cost, which is 30 times higher (this is the same as the bar A1-R2, 30 times in Figure 6.13 for prospective adaptations), and is presented again for comparison purposes. It can be seen that the performance of the adaptivity is modified only slightly, which enables us to claim that the approach to dynamic balancing proposed in this thesis can adapt efficiently to rapid changes of resource performance.

Comparison with near-optimal static configuration Continuing the previous test, let us assume that the system has a perfect prediction mechanism, which enables it to estimate near-optimal distribution policies statically (thus avoiding all the overheads). Table 6.4 shows the difference in performance when the system adapts to machine perturbations at runtime, and the redistributions are retrospective. From the table, and taking into consideration also the fact that the adaptivity mechanisms incur a non-negligible overhead, it can be seen that the adaptations are reasonably close to the optimal behaviour, even when the size of perturbation changes dynamically.

6.4 Adapting to Resource Availability

6.4.1 Motivation and Problem Statement

Dynamic data (and possibly state) repartitioning is just one aspect of the problem of adapting to changing resources, as they both deal with the changing behaviour of a stable set of computational resources. In a more generic case, this simplifying assumption must be relaxed. Long-running queries should be able to acquire new resources on the fly, both to resolve performance problems and to respond to new opportunities.

There are many reasons why this cannot happen statically at compile time. The resources may be initially unavailable; or, the load of a machine up to a certain point of execution may be too high for the system to decide to use it; or, due to unpredictability, unavailability and incorrectness of data statistics, the query cost may have been underestimated. For all these undesirable situations, the query processing needs to be re-adjusted on the fly, with an eye on the remote resources that can be allocated.

Such adaptations are complicated by the fact that a dynamic resource allocation must be accompanied by dynamic data (and possibly state) repartitioning so that the new machines are allocated a proportion of the query workload. Consequently, the adaptations that employ resources on the fly subsume the adaptations that repartition data at runtime and were presented in Section 6.3. However, this section deals only with the aspects that install a new AGQES and allocate an initial workload to it during execution, as the other aspects were examined previously.

6.4.2 Approach

The *monitoring-assessment-response* architectural framework can accommodate adaptations of this kind by activating a single *MonitoringEventDetector*, a single *Diagnoser*, and a single *Responder*. These components need not reside at the same machine, although this is desirable in order to reduce communication costs.

The configuration information sent by the GDQS to the AGQESs includes:

- All the information sent for dynamic balancing as discussed in Section 6.3.2.
- The handles of all AGQES Factories that are registered. This information is considered only by the AGQES that holds the globally accessible *MonitoringEventDetector*, and is necessary to conduct the monitoring of available resources.
- For the AGQES holding the *Responder*, a copy of the subplans comprising the query plan, so that the *Responder* can clone them in new AGQESs.

6.4.3 Monitoring

The *MonitoringEventDetector* component subscribes to all registered AGQES factories (AGQESFs). The factories have the capability to spawn AGQESs, but they do so only if the static optimiser, at compile time, or a *Responder*, at runtime, decide accordingly. The AGQESFs continuously update a specific element of their interface with

```
<complexType name="Notification">
    <sequence>
       <element name="description" value="DETECTED_EVENT"/>
       <element name="destinationId" value="DIAGNOSER"/>
       <!-- rest generic fields -->
       ....
       <!-- specific fields -->
       <element name="isAvailable" />
       <element name="memoryAvailableMB" />
       <element name="cpuLoad" />
    </sequence>
</complexType>
```

Figure 6.19: Schema definition of notifications sent by the *MonitoringEventDetector* component.

the current amount of memory available (retrieved through the Java VM), and, if they are Unix-based, with their recent load. The difference between monitoring a query engine and an AGQESF from the viewpoint of a *MonitoringEventDetector* is that, for the former, the notifications are sent in push mode, whereas the latter send the monitoring notifications upon request by the *MonitoringEventDetector*.

A notification is published when the availability changes, or when either the load of the amount of memory available changes more than a threshold *thresM*, which is set to 20%. The schema in XSD of such notifications is presented in Figure 6.19.

6.4.4 Assessment

In this prototype development and evaluation, the objective is to handle the common case where the external cost is significant to overall performance. To this end, the following simple rule has been used: use any machine that becomes available and assign to it a subplan that contains a WS invocation (if there is any), provided that the new machine can evaluate such a WS locally. Thus the *Diagnoser* propagates a notification sent by the *MonitoringEventDetector* to the *Responder*, if this notification denotes change of the status of a machine from non-available to available. The development of more robust and comprehensive mechanisms to decide whether a new machine should be employed on the fly is left for future work.

```
<complexType name="Notification">
  <sequence>
    <element name="description" value="RESPONSE_NEWPRODUCER"/>
    <element name="destinationId value="AGQES_RESPONDER"/>
    <!-- rest generic fields -->
    ....
    <!-- specific fields -->
    <element name="ExchangeReconfiguration">
    <complexType>
       <element name="modifiedOpId" />
       <element name="producerReference" />
    </complexType>
    </element>
  </sequence>
</complexType>
```

Figure 6.20: Schema definition of notifications sent by the *Responder* component to notify of a new producer.

6.4.5 Response

As in the case of workload imbalance, the *Responder* reacts to new machine availabilities, only if the execution is not close to completion, i.e. the progress has not exceeded a threshold. This threshold is a tunable parameter set to 95% in the default configuration. The actual response consists of the following three steps:

1. An AGQES is created remotely by the *Responder*. Subsequently, the partitioned subplan that contains a WS invocation is sent to the new AGQES. For both actions, the same mechanism as the one employed by the GDQS to initiate execution is used. At this point, the new evaluator remains temporarily idle because the other AGQESs have not been notified of its existence yet.

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 (example schema is presented in Figure 6.20). The analysis of this notification within the evaluation engine, results in a call to the *addProducer()* function of the exchange consumers affected.

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 by calling their *addConsumer()* function (Figure 6.21); and (ii) to modify the data partitioning policy of the AGQESs that send data to the new AGQES to take into account the new consumer. The new consumer is initially assigned a proportion of the complete workload which is equal to the average proportion

```
<complexType name="Notification">
  <sequence>
    <element name="description" value="RESPONSE_NEWCONSUMER"/>
    <element name="destinationId value="AGQES_RESPONDER"/>
    <!-- rest generic fields -->
    ....
    <!-- specific fields -->
    <element name="ExchangeReconfiguration">
    <complexType>
       <element name="modifiedOpId" />
       <element name="consumerReference" />
    </complexType>
    </element>
  </sequence>
</complexType>
```

Figure 6.21: Schema definition of notifications sent by the *Responder* component to notify of a new consumer.

of the pre-existing AGQESs. If this default policy is not to be followed, an explicit notification like the one in Figure 6.10 is sent. The dynamic balancing mechanism presented in Section 6.3 can correct bad initial decisions at runtime. Both retrospective and prospective partitioning can be applied; however, only the latter is used during evaluation.

6.4.6 Evaluation

The main aim of the evaluation is to establish if the overheads for monitoring resources dynamically, and for allocating additional machines at runtime, can be outweighed by the performance gains. Q1 of Section 6.3 is used as the example query.

6.4.6.1 Performance Gains

Q1 is evaluated for different costs of WS calls, specifically when the cost is 40, 80 and 120 msecs per tuple. We assume that at compile time only a single machine is available to evaluate the *EntropyAnalyser* WS, but that another machine becomes available as soon as the execution starts. Table 6.5 presents the gain in query response time when the system allocates the new machine. As the number of machines evaluating the costlier part of the query has doubled, the query response time drops almost by half.

WS cost	no new AGQES	new AGQES
0.040	123.02	68.51
0.080	242.47	131.82
0.120	360.82	195.98

Table 6.5: Performance of Q1 with and without dynamic resource allocation (in secs).

6.4.6.2 Overheads

Two types of overhead are considered: the monitoring overhead to contact a remote machine to retrieve up-to-date information, and the overhead of response, which comprises the three steps described in Section 5.1. Both involve non-trivial inter-service communication, and thus are non-negligible. The monitoring overhead is approximately 0.3 secs every time a remote machine is contacted. For the previous query in which one AGQES is created on the fly and two existing AGQESs are notified of the existence of the new AGQES, the response overhead is approximately 1 second (the cost to create a new AGQES and install a plan fragment is the dominant cost). From Table 6.5, it can be seen that the improvements are well worth such overheads.

6.5 Summary

The volatility of the environment in parallel query processing over heterogeneous and autonomous wide-area resources makes it imperative to adapt to changing resource properties, in order not to suffer from serious performance degradation. This chapter proposes two solutions, one for dynamic workload balancing through data and operator state repartitioning, and another for dynamic resource allocation. Both solutions have been implemented through extensions to the Grid-enabled open-source OGSA-DQP system. The implementation is particularly appealing for environments such as the Grid, as it is based on loosely-coupled components, engineered as Grid Services. The results of the empirical evaluation are promising: performance is significantly improved (by an order of magnitude in some cases), while the overhead remains low enough to allow the benefits of adaptation to outweigh its cost for a wide range of scenarios.

In particular, the chapter makes the following key contributions:

- It proposes an instantiation of the *monitoring-assessment-response* architectural

framework for AQP, which was introduced in Chapter 4. The framework instantiation (i) covers both data and operator state repartitioning, and (ii) is capable of allocating new resources dynamically. The development of such adaptivity functionalities in the same architectural context is a contribution in its own right. Key features of the architecture include that it is non-centralised, service-oriented, and its components communicate with each other asynchronously according to the publish/subscribe model. Thus it can be applied to loosely-coupled, autonomous environments such as the Grid.

- It presents the implementation, which has been made through extensions to the OGSA-DQP distributed query processor for the Grid (Section 2.5). This demonstrates the practicality of the approach. The resulting prototype has been empirically evaluated and the results show that it can yield performance improvements by several factors, if not order of magnitude, in representative examples. In addition, the overhead remains reasonably low, which is significant when adaptivity is not required.

- It examines two case studies that have not been sufficiently explored in AQP to date, namely adaptive workload balancing and resource allocation in wide-area distributed query processing.

- The approach followed in this thesis leverages mechanisms deployed for fault-tolerance to achieve management and movement of operator state, aiming at component and software reuse.

Chapter 7

Conclusions

This chapter concludes the thesis with a review of the work presented and of the way in which the aim of this thesis has been accomplished (Section 7.1). In addition, this chapter summarises the main results, stating their significance (Section 7.2), and discusses outstanding issues along with directions for future work (Section 7.3).

7.1 Overview

The aim of the thesis was to propose and evaluate techniques that significantly improve the performance and robustness of query processing on the Grid. This aim has been attained through the development and performance investigation of:

- a resource selection and scheduling algorithm, which is suitable also for intensive queries, through the provision of partitioned parallelism; and
- an adaptive query processing (AQP) approach, which addresses the inherent volatility and unpredictability of the Grid environment and conforms to a generic architectural framework.

Both proposals revolve on a common goal: to take account of the potentially evolving characteristics and behaviour of the computational resources that are available. The proposed scheduler allocates resources as long as these are expected to improve the performance of the query evaluation, whereas the adaptivity techniques adapt to runtime resource behaviour and state.

The evaluation has been conducted in the context of two Grid-enabled query processing systems, namely Polar* and OGSA-DQP, the development of which has been, to

a certain degree, part of the work of the current thesis. The evaluation results support the claim that the proposals presented are both efficient and practical. They incur low overhead, whereas they can improve performance by an order of magnitude in some cases.

For the development of the resource scheduling algorithm, considerable attention has been paid to its suitability for practical use. To achieve this, the algorithm employed a computationally inexpensive heuristic rather than an exhaustive search for all possible solutions.

The design of the AQP techniques has been largely influenced by the need to avoid this to occur in an *ad hoc*, isolated, non-extensible way. The fact that many different AQP approaches may be desirable and yield performance improvements in a Grid setting, has increased the need for a more systematic development process that enables their combination. To this end, a generic framework for AQP has been introduced, which has provided the context for the specific adaptations that have been examined. The basis of the framework is the decomposition of adaptivity into *monitoring*, *assessment* and *response*, an approach which is well-established in many autonomous and adaptive systems (e.g., [KC03]), but rather unknown in the AQP community [GPSF04][1]. Pursuing the idea of a generic adaptivity framework even further, a generic approach to extracting information from the execution of a query plan has been developed to drive adaptations both in a Grid setting and in a more traditional environment.

7.2 Significance of Major Results

The major results of this thesis are:

- An evaluated resource selection and scheduling algorithm that selects and allocates resources to query fragments, which together comprise the query execution plan (Chapter 3).
- A generic adaptivity framework for AQP, which distinguishes between the monitoring, assessment and response phases that are inherently present in adaptivity (Chapter 4).

[1] A reason for this might be the fact that most of the AQP approaches have been proposed after the mid-90s, and as described in [HFC+00], AQP is a technology in evolution. Thus the emphasis has mostly been on functionality, rather than on broadly accepted abstractions, systematic development and component reuse.

- An evaluated generic mechanism to extract information from the execution of query plans, which is based on self-monitoring operators (Chapter 5).
- An instantiation of the adaptivity framework as an extension to the OGSA-DQP system, which provides (static) Grid Services for query processing (Section 6.2). The resulting system has been used to implement and evaluate techniques for adapting to changing resources, and, in particular:
  - for adapting to changing resource behaviour that results in imbalanced execution (Section 6.3), and
  - for adapting to changing resource availability (Section 6.4).

The Polar* and OGSA-DQP systems discussed in Sections 2.4 and 2.5 respectively, apart from providing the basis for the development and the evaluation of the contributions above, are also of significant interest in their own right. These systems have been implemented by the Polar*/OGSA-DQP project's team [POL], being a joint effort between the Universities of Manchester and Newcastle upon Tyne. The contribution of the work of this thesis to the development of Polar* and of the non-adaptive OGSA-DQP is related to the compiler/optimiser component.

7.2.1 Resource Scheduling for Grid Query Processing

A resource scheduling algorithm has been designed to support scalable and computationally intensive query processing on the Grid. Both these requirements are in line with the broader vision of Grid computing and its orientation towards large-scale, non-trivial, and possibly high-performance applications. The proposal goes beyond the current state-of-the-art in distributed query processing (DQP), as it lays emphasis on and does not overlook partitioned parallelism, which is an effective approach to obtaining scalability and improved performance. Moreover, it can take into account the potentially vast pool of available resources, and the specific capabilities of the latter. Thus, it avoids oversimplifying assumptions that are typically made in parallel databases, such as that all machines are characterised by the same properties, and that the physical location of an operator does not have an impact on the query execution. The evaluation results show significant improvements compared to other approaches, developed for use with distributed or parallel databases. For the evaluation, the Polar* query engine has been used, employing a variant of the cost model developed in Polar [SSWP04].

The scheduling algorithm is of reasonably low computational complexity, and is presented in an application-independent way. It is extensible in terms of the query operators considered, and orthogonal in terms of the cost model, which describes the behaviour of each operator if run on a specific machine. As such, it can be applied to a wide range of query processors operating in a heterogeneous environment.

7.2.2 The Monitoring-Assessment-Response Framework for Adaptive Query Processing

It has become a consensus that AQP techniques are being implemented in an *ad hoc* way (e.g., [IHW04, BB05, GPSF04]), which compromises their capability to be combined. To make the development of AQP techniques more systematic, a generic framework has been developed. The core idea is to investigate separately the phases of collecting feedback from the execution and environment, analysing this feedback, and responding to changes based on the feedback analysis, steps that are inherently present but often conflated in AQP systems. Thus, AQP is decomposed into monitoring, assessment and response phases. Three different kinds of adaptivity components are identified, 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, which is a suitable paradigm for the Grid.

The construction of frameworks for identifying or composing generic and reusable techniques for monitoring, assessment or response has the following key advantages. Firstly, it allows component reuse and enables the assembling of different combinations, thus covering a wider spectrum of capabilities. Secondly, the adoption of a generic framework by the developers of AQP systems makes the design activity more systematic. Thirdly, by focusing on the interfaces of cohesive, decoupled modules, the design of the framework conforms to the emerging Web and Grid Services paradigms, which are suitable for advanced, wide-area applications. Finally, a framework like the one proposed is generic in the sense of being both adaptivity environment independent, and technique independent. It makes no assumptions as to the number of adaptivity components cooperating to achieve an adaptation, or their specific interconnections.

7.2.3 Self-monitoring operators

Self-monitoring operators have been proposed for extracting monitoring information from the query plan in a generic manner, independently of the techniques for assessment and response. However, they are capable of collecting information that is directly relevant to the assessment process by establishing where a plan is deviating from its anticipated behaviour, as they can identify and predict erroneous initial estimates on the fly, and are sufficient to support most AQP proposals to date. Thus, self-monitoring operators can provide the necessary background for on-the-fly adaptations, and consequently, they can be of interest for many AQP proposals.

A key feature of the approach is that it can be easily implemented and incorporated in existing query processors, as it employs only counters, timestamps and tuple size computations and it affects only the implementation of the query operators. In addition, it can scale well with the number of machines used in query execution because there exists no central monitoring point. The evaluation has been conducted in the context of Polar* and the results have been encouraging, both in terms of the cost of monitoring, and in terms of the quality of the monitoring information.

7.2.4 Adaptive Grid Services for Query Processing

To instantiate the monitoring-assessment-response adaptivity framework, and employ self-monitoring operators in a Grid environment, an *Adaptive Grid Query Evaluation Service (AGQES)* has been developed, as an extension to the static GQES provided by the OGSA-DQP system.

This engineering approach transforms the query evaluators of OGSA-DQP into autonomous entities, capable of self-configuring and self-optimising. It allows many such entities to collaborate for adaptation, through the support of a publish/subscribe interface. Overall, it provides a context for the development of arbitrarily complex AQP techniques, and demonstrates the practicality and the benefits of a generic architectural framework.

7.2.5 Adapting to Changing Resources

Adaptivity techniques have been presented that are capable of (i) adapting to workload imbalance; and (ii) reacting to changes in the resource pool during query execution

over Grid resources. Both were implemented by using the AGQESs. From the perspective of AQP technology, they extend the scope of cases for which AQP is demonstrated to yield benefits, as they explore (i) monitoring of changing resources; (ii) adaptations in distributed query processing; and (iii) distributed adaptivity mechanisms. Previously, each of these three fields was insufficiently examined; also, their combination is a novelty. From the perspective of Grid Service engineering, the adaptations provide an example of dynamic service orchestration.

The empirical evaluation shows that the performance improvements that can be achieved in a reasonably wide range of scenarios are significant, and they can outweigh the overhead incurred. In particular, the experiments demonstrate the capability of the system to gracefully degrade its performance in the presence of perturbed machines, and to boost query evaluation when new computational resources become available.

7.2.6 The Polar* and OGSA-DQP systems

The Polar* and OGSA-DQP query processing systems for the Grid have been developed as proofs of concept, to demonstrate the role that DQP can play in Grid computations by combining data access and analysis. Indeed, they have been used to show that a significant set of Grid tasks can be described as database queries, and be executed as query plans using both well-established and novel query processing techniques. Such techniques include the capability of query processors to apply optimisation policies to the graph that represents the query execution plan, and to employ parallelism to speed-up the execution in a way transparent to the user.

7.2.7 Lessons Learned

Investigating techniques and strategies to improve the performance of query processing on the Grid has been an exciting exercise. Apart from the results mentioned above, the following, more generic lessons have been learned:

- Taking into account the heterogeneous resource characteristics, and their dynamic evolution, has been a correct choice toward performance improvements. It is not clear if other approaches can yield improvements by up to an order of magnitude.

- Empirical evaluation of wide-area, autonomous systems is a difficult and time-consuming activity. In the absence of sufficient theoretical models, comprehensive and validated simulations may be a better or easier evaluation mechanism.

- Evaluation of frameworks is also difficult and not well defined. In this thesis, the adaptivity framework has been examined from three complementary points of view. Firstly, it has been used as an explanatory platform to compare and present the similarities of different AQP techniques. Secondly, insight has been provided as to whether generic adaptivity components, such as monitoring ones, are feasible. Thirdly, complete adaptations have been developed according to the framework. Although all these implicitly constitute a validation of the framework proposal, it is still not clear whether and how a generic framework can be formally or more explicitly validated.

- AQP lacks the theoretical background of other relevant domains, such as control theory. Control theory deals with the problem of assessing collected feedback and responding to monitored changes from another perspective. It builds upon two strong assumptions regarding the adaptive system: the performance of the system is a function of controlled and tunable (set of) parameters, and there exist mathematical models that describe with sufficient accuracy the behaviour of the system over time. Unfortunately, query processing cannot satisfy the first criterion, as it always executes in a best-effort manner. For the latter, modeling the behaviour of distributed querying is still an open research issue, and involves several, both independent and correlated, variables that need to be adjusted at runtime. Thus, it would result in a multivariable non-linear adaptive control system that can be handled only to a very limited extent theoretically, and not truly supported by commercial systems in practice [ÅW95]. For these reasons, results from control theory cannot be transferred into adaptive querying, and there is very limited theoretical understanding of AQP.

7.3 Open Issues and Directions for Future Work

7.3.1 Outstanding Issues

This thesis does not, of course, represent a complete answer to the problems associated with efficient query processing on the Grid or with exploitation of (dynamic) resource

metadata to improve performance. There remain a number of outstanding issues, several of which are described below:

- The resource scheduler has been evaluated in an environment in which resource characteristics do not change during execution and thus there is no need to reschedule resources dynamically; it would be interesting to test the scheduling algorithm for rescheduling the query plan on the fly. Also, although the algorithm performs well, there is space for further improvements, the performance and the associated overhead of which may be worth investigating. For example, the current version of the algorithm first tries to parallelise the most costly operator, and if it fails, it stops. Thus, it may not exploit the potential benefits of further parallelising the other operators. An improvement could consider the difference in the cost of the two most expensive operators and, if it is below a certain threshold, parallelise both of them. Another improvement could re-evaluate the capabilities of the machines affected by a scheduling of a new machine. For example, the load of a machine should be increased after a subplan has been allocated to it. Finally, an open issue is to develop techniques for efficient threshold parameter setting in different circumstances.

- The adaptivity components for dynamic workload redistribution and resource allocation on the fly rely heavily on threshold parameters. As for the scheduling algorithm, little insight has been provided as to how to set these parameters efficiently. Clearly, there is a trade-off between the speed of adaptations and the overhead associated, along with the danger of over-reacting. Relatively high thresholds are used for more conservative approaches, whereas low thresholds result in a more aggressive adaptivity behaviour. However, this trade-off has not been examined in this work in depth.

- For the adaptations examined, the query engine (through self-monitoring operators) and the interface of the AGQESs provide all the monitoring information required. In a more generic adaptivity case, information from higher level Grid monitoring services, such as MDS [CFFK01], for computational resources, and NWS [WSH99], for network bandwidth, may need to be acquired. This does not affect the design of the AGQESs; it just broadens the scope of adaptations that can be supported in the future.

- According to the evaluation results, dynamic resource allocation may lead to significant gains in response time. Nevertheless, the development of robust, cost-based assessment policies to decide whether to employ a new resource has not been examined. As mentioned earlier, the scheduling algorithm is a strong candidate, provided that all the metadata it requires is either known or feasible to obtain at runtime, and a cost model exists.

7.3.2 Future Work

The work of this thesis, along with the development of the Polar* and OGSA-DQP systems, has suggested several promising avenues for further research:

- Developing a cost model to describe the query engine of OGSA-DQP, like the one that has been developed for the Polar/Polar* evaluators [SSWP04]. This will allow the design of a simulator of the OGSA-DQP system, which is significant (i) for deploying cost-based optimisation approaches; (ii) for faster evaluation of extensions and techniques built upon OGSA-DQP; and (iii) for better understanding of the behaviour of the system.

- Making the monitoring of query plans adaptive itself. For example, it may be important to be able to modify the intensity, the granularity and the frequency of monitoring at runtime.

- Evaluating the AQP techniques developed in a real Grid environment, such as the UK National Grid Service or the PlanetLab [Cul03].

- Investigating and incorporating in the adaptive OGSA-DQP prototype new adaptations, such as adapting to fluctuating network bandwidth and substituting resources on the fly. In addition, existing AQP techniques that tackle the problem of unknown data statistics at compile time are of particular interest for query processing on the Grid.

- Developing generic components and techniques for assessment and response. This thesis has demonstrated how adaptations to changing resources in Grid query processing can be developed according to the monitoring-assessment-response framework. Moreover, it has discussed how the self-monitoring approach to extracting information from query execution can be deployed in many AQP systems. However, as the thesis aim was to extend the state-of-the-art in

Grid query processing, solutions for the problem of generic adaptivity components have not been pursued any further[2]. Such components should also be able to handle resolution of conflicts between their results, especially when there are multiple instances of the same type of component.

- Investigating in depth the relationships of AQP with *autonomic computing* [KC03], and how results from one area can be transferred to the other. The emergence of autonomic computing has been motivated by the increasing complexity of modern systems, on the one hand, and the unpredictable and volatile nature of platforms for distributed computations, such as the Grid, on the other. As such, it shares, to a large extent, the same motivations as the work in the current thesis. The emphasis of autonomic computing is on the development of self-managing systems that are capable of monitoring themselves with a view to continuously maintaining a certain level of QoS, by achieving the properties of *self-configuration, self-optimisation, self-healing* and *self-protection.* AQP is aimed at achieving several of the goals of autonomic computing, especially with respect to self-optimisation and self-healing. For example, Section 6.4 has demonstrated how AQP can by used to achieve self-optimisation by employing new computational resources during execution to boost performance, when these resources become available.

- Investigating techniques to improve performance of query processing on the Grid when the optimisation criteria are other than the query response time (e.g., overall resource utilisation and monetary cost). Also, this thesis has focused on single query optimisation; it will be interesting to propose techniques for multi-query optimisation as well.

[2]To this end, a follow-on project has already been funded (EPSRC Grant EP/C537137).

Bibliography

[ACK+02] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[AD03] Remzi H. Arpaci-Dusseau. Run-time adaptation in river. *ACM Trans. Comput. Syst.*, 21(1):36–86, 2003.

[ADAT+99] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, David A. Patterson, and Katherine A. Yelick. Cluster i/o with river: Making the fast case common. In *IOPADS*, pages 10–22, 1999.

[AFT98] Laurent Amsaleg, Michael J. Franklin, and Anthony Tomasic. Dynamic query operator scheduling for wide-area remote access. *Distributed and Parallel Databases*, 6(3):217–246, 1998.

[AFTU96] Laurent Amsaleg, Michael J. Franklin, Anthony Tomasic, and Tolga Urhan. Scrambling query plans to cope with unexpected delays. In *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems, December 18-20, 1996, Miami Beach, Florida, USA*, pages 208–219. IEEE Computer Society, 1996.

[AGM+90] S. F. Altschul, W. Gish, W. Miller, E.W Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[AH99] Ron Avnur and Joseph M. Hellerstein. Continuous query optimization. Technical Report CSD-99-1078, University of California, Berkeley, 1999.

[AH00] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA.*, pages 261–272. ACM, 2000.

[AMG+04] M. Nedim Alpdemir, Arijit Mukherjee, Anastasios Gounaris, Norman W. Paton, Paul Watson, Alvaro A. A. Fernandes, and Desmond J. Fitzgerald. OGSA-DQP: A service for distributed querying on the grid. In Elisa Bertino, Stavros Christodoulakis, Dimitris Plexousakis, Vassilis Christophides, Manolis Koubarakis, Klemens Böhm, and Elena Ferrari, editors, *Advances in Database Technology - EDBT 2004, 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18, 2004, Proceedings*, pages 858–861. Springer, 2004.

[AMP+03] M. Nedim Alpdemir, A. Mukherjee, Norman W. Paton, Paul Watson, Alvaro A. A. Fernandes, Anastasios Gounaris, and Jim Smith. Service-based distributed querying on the grid. In Maria E. Orlowska, Sanjiva Weerawarana, Mike P. Papazoglou, and Jian Yang, editors, *Service-Oriented Computing - ICSOC 2003, First International Conference, Trento, Italy, December 15-18, 2003, Proceedings*, pages 467–482. Springer, 2003.

[ÅW95] Karl Johan Åström and Björn Wittenmark. *Adaptive Control*. Addison-Wesley, Reading, MA, USA, 1995.

[BAGS02] Rajkumar Buyya, David Abramson, Jonathan Giddy, and Heinz Stockinger. Economic models for resource management and scheduling in grid computing. *Concurrency and Computation: Practice and Experience*, 14(13-15):1507–1542, 2002.

[BB05] Shivnath Babu and Pedro Bizarro. Adaptive query processing in the looking glass. In *CIDR*, pages 238–249, 2005.

[BBDM03] Brian Babcock, Shivnath Babu, Mayur Datar, and Rajeev Motwani. Chain : Operator scheduling for memory minimization in data stream

systems. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 253–264. ACM, 2003.

[BBH+02] W. H. Bell, D. Bosio, W. Hoschek, P. Kunszt, G. McCance, and M. Silander. Project Spitfire - Towards Grid Web Service Databases. In *Global Grid Forum 5*, 2002.

[BC02] Nicolas Bruno and Surajit Chaudhuri. Exploiting statistics on query expressions for optimization. In Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, pages 263–274. ACM, 2002.

[BDS03] Vincent Boudet, Frederic Desprez, and Frédéric Suter. One-step algorithm for mixed data and task parallel scheduling without data replication. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France, CD-ROM/Abstracts Proceedings*, page 41. IEEE Computer Society, 2003.

[BFK+00] Michael D. Beynon, Renato Ferreira, Tahsin M. Kurç, Alan Sussman, and Joel H. Saltz. Datacutter: Middleware for filtering very large scientific datasets on archival storage systems. In *IEEE Symposium on Mass Storage Systems*, pages 119–134, 2000.

[BFMV00a] Luc Bouganim, Françoise Fabret, C. Mohan, and Patrick Valduriez. A dynamic query processing architecture for data integration systems. *IEEE Data Eng. Bull.*, 23(2):42–48, 2000.

[BFMV00b] Luc Bouganim, Françoise Fabret, C. Mohan, and Patrick Valduriez. Dynamic query scheduling in data integration systems. In *ICDE*, pages 425–434, 2000.

[BGW+81] Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve, and James B. Rothnie Jr. Query processing in a system for distributed databases (sdd-1). *ACM Trans. Database Syst.*, 6(4):602–625, 1981.

[BKK+01] Reinhard Braumandl, Markus Keidl, Alfons Kemper, Donald Kossmann, Alexander Kreutz, Stefan Seltzsam, and Konrad Stocker. Objectglobe: Ubiquitous query processing on the internet. *VLDB J.*, 10(1):48–71, 2001.

[BKV98] Luc Bouganim, Olga Kapitskaia, and Patrick Valduriez. Memory-adaptive scheduling for large query execution. In Georges Gardarin, James C. French, Niki Pissinou, Kia Makki, and Luc Bouganim, editors, *Proceedings of the 1998 ACM CIKM International Conference on Information and Knowledge Management, Bethesda, Maryland, USA, November 3-7, 1998*, pages 105–115. ACM, 1998.

[BMM+04] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. Adaptive ordering of pipelined stream filters. In Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 407–418. ACM, 2004.

[CB00] R. G. G. Cattell and D. K. Barry. *The Object Database Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[CBB+03] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Çetintemel, Ying Xing, and Stanley B. Zdonik. Scalable distributed stream processing. In *CIDR*, 2003.

[CcC+02] Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Monitoring streams - a new class of data management applications. In *VLDB*, pages 215–226, 2002.

[CCD+03] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Fred Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[CDF+94] M. Carey, D.J. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White,

and M. Zwilling. Shoring up persistent applications. In R. Snodgrass and M. Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994*, pages 383–394. ACM Press, 1994.

[CDN93] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 benchmark. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 22(2):12–21, 1993.

[CF03] Sirish Chandrasekaran and Michael J. Franklin. Psoup: a system for streaming queries over streaming data. *VLDB J.*, 12(2):140–156, 2003.

[CFFK01] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *10th IEEE Symp. On High Performance Distributed Computing*, 2001.

[Cha98] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 1-3, 1998, Seattle, Washington*, pages 34–43. ACM Press, 1998.

[CHS99] Francis Chu, Joseph Y. Halpern, and Praveen Seshadri. Least expected cost query optimization: an exercise in utility. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 138–147. ACM Press, 1999.

[CNR04] Surajit Chaudhuri, Vivek R. Narasayya, and Ravishankar Ramamurthy. Estimating progress of long running SQL queries. In Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 803–814. ACM, 2004.

[Cul03] David E. Culler. Planetlab: An open, community-driven infrastructure for experimental planetary-scale services. In *USENIX Symposium on Internet Technologies and Systems*, 2003.

[CYW96] Ming-Syan Chen, Philip S. Yu, and Kun-Lung Wu. Optimization of parallel execution for multi-join queries. *IEEE Trans. Knowl. Data Eng.*, 8(3):416–428, 1996.

[DAI] The database access and integration services (DAIS) standard, www.gridforum.org/6_data/dais.htm.

[DBC03] Holly Dail, Francine Berman, and Henri Casanova. A decoupled scheduling approach for grid application development environments. *J. Parallel Distrib. Comput.*, 63(5):505–524, 2003.

[Des04] Amol Deshpande. An initial study of overheads of eddies. *SIGMOD Record*, 33(1):44–49, 2004.

[DG92] David J. DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[DGG+86] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna. Gamma - a high performance dataflow database machine. In Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi, editors, *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings.*, pages 228–237. Morgan Kaufmann, 1986.

[DH04] Amol Deshpande and Joseph M. Hellerstein. Lifting the burden of history from adaptive query processing. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *Proceedings of the Thirtieth International Conference on Very Large Data Bases, Toronto, Canada, August 31 - September 3 2004*, pages 948–959. Morgan Kaufmann, 2004.

[DSB+03] H. Dail, O. Sievert, F. Berman, H. Casanova, A. YarKhan, S. Vadhiyar, J. Dongarra, C. Liu, L. Yang, D. Angulo, and I. Foster. Scheduling in the grid application development software project. In J. Nabrzyski, J. Schopf, and J. Weglarz, editors, *Grid resource management: state of the art and future trends*. Kluwer Academic Publishers Group, 2003.

[EDNO97] Cem Evrendilek, Asuman Dogac, Sena Nural, and Fatma Ozcan. Multidatabase query optimization. *Distributed and Parallel Databases*, 5(1):77–113, 1997.

[EFGK03] Patrick Th. Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[ESW78] Robert S. Epstein, Michael Stonebraker, and Eugene Wong. Distributed query processing in a relational data base system. In Eugene I. Lowenthal and Nell B. Dale, editors, *Proceedings of the 1978 ACM SIGMOD International Conference on Management of Data, Austin, Texas, May 31 - June 2, 1978*, pages 169–180. ACM, 1978.

[Feg97] Leonidas Fegaras. An experimental optimizer for OQL. Technical Report TR-CSE-97-007, University of Texas at Arlington, 1997.

[Feg98] Leonidas Fegaras. A new heuristic for optimizing large queries. In Gerald Quirchmayr, Erich Schweighofer, and Trevor J. M. Bench-Capon, editors, *Database and Expert Systems Applications, 9th International Conference, DEXA '98, Vienna, Austria, August 24-28, 1998, Proceedings*, pages 726–735. Springer, 1998.

[FK99] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., 1999.

[FK03] Ian Foster and Carl Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, San Francisco, CA, USA, second edition, 2003.

[FKNT02] Ian Foster, Carl Kesselman, J. M. Nick, and Steve Tuecke. Grid Services for Distributed System Integration. *IEEE Computer*, 35(6):37–46, 2002.

[FKT01] Ian Foster, Carl Kesselman, and Steve Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int. J. Supercomputer Applications*, 15(3), 2001.

[FM00] Leonidas Fegaras and David Maier. Optimizing Object Queries Using an Effective Calculus. *ACM TODS*, 24(4):457–516, 2000.

[FSRM00] Leonidas Fegaras, Chandrasekhar Srinivasan, Arvind Rajendran, and David Maier. lambda-db: An ODMG-based object-oriented DBMS. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors,

Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA., page 583. ACM, 2000.

[FTL+01] James Frey, Todd Tannenbaum, Miron Livny, Ian T. Foster, and Steven Tuecke. Condor-g: A computation management agent for multi-institutional grids. In *10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10 2001), 7-9 August 2001, San Francisco, CA, USA*, pages 55–66. IEEE Computer Society, 2001.

[GGF] Global Grid Forum (GGF), http://www.gridforum.org.

[GGKS02] K. Gottschalk, S. Graham, H. Kreger, and J. Snell. Introduction to Web Services Architecture. *IBM Sys. Journal*, 41(2):170–177, 2002.

[GHK92] Sumit Ganguly, Waqar Hasan, and Ravi Krishnamurthy. Query optimization for parallel execution. In Michael Stonebraker, editor, *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, June 2-5, 1992.*, pages 9–18. ACM Press, 1992.

[GI96] Minos N. Garofalakis and Yannis E. Ioannidis. Multi-dimensional resource scheduling for parallel queries. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996.*, pages 365–376. ACM Press, 1996.

[GI97] Minos N. Garofalakis and Yannis E. Ioannidis. Parallel query scheduling and optimization with time- and space-shared resources. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*. Morgan Kaufmann, 1997.

[GLO] The Globus toolkit, http://www.globus.org.

[GMPQ+97] Hector Garcia-Molina, Yannis Papakonstantinou, Dallan Quass, Anand Rajaraman, Yehoshua Sagiv, Jeffrey D. Ullman, Vasilis Vassalos, and Jennifer Widom. The tsimmis approach to mediation: Data models and languages. *J. Intell. Inf. Syst.*, 8(2), 1997.

[GMUW01] Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer D. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2001.

[GÖ03] Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, 2003.

[GPFS02] Anastasios Gounaris, Norman W. Paton, Alvaro A. A. Fernandes, and Rizos Sakellariou. Adaptive query processing: A survey. In Barry Eaglestone, Siobhán North, and Alexandra Poulovassilis, editors, *Advances in Databases, 19th British National Conference on Databases, BNCOD 19, Sheffield, UK, July 17-19, 2002, Proceedings*, pages 11–25. Springer, 2002.

[GPFS03] Anastasios Gounaris, Norman W. Paton, Alvaro A. A. Fernandes, and Rizos Sakellariou. Monitoring query execution plans. Technical report, Dept. of Comp. Science, Univ. of Manchester, 2003.

[GPFS04] Anastasios Gounaris, Norman W. Paton, Alvaro A. A. Fernandes, and Rizos Sakellariou. Self-monitoring query execution for adaptive query processing. *Data Knowl. Eng.*, 51(3):325–348, 2004.

[GPSF04] Anastasios Gounaris, Norman W. Paton, Rizos Sakellariou, and Alvaro A. A. Fernandes. Adaptive query processing and the grid: Opportunities and challenges. In *15th International Workshop on Database and Expert Systems Applications (DEXA 2004)*, pages 506–510. IEEE Computer Society, 2004.

[GPSG03] C.A. Goble, S. Pettifer, R. Stevens, and C. Greenhalgh. Knowledge integration: In silico experiments in bioinformatics. In Ian Foster and Carl Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure"*. Morgan Kaufmann Publishers, 2003.

[Gra90] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990.*, pages 102–111. ACM Press, 1990.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[GSPF04] Anastasios Gounaris, Rizos Sakellariou, Norman W. Paton, and Alvaro A. A. Fernandes. Resource scheduling for parallel query processing on computational grids. In *5th International Workshop on Grid Computing (GRID 2004), 8 November 2004, Pittsburgh, PA, USA, Proceedings*, pages 396–401. IEEE Computer Society, 2004.

[GST96] Georges Gardarin, Fei Sha, and Zhao-Hui Tang. Calibrating the query optimizer cost model of iro-db, an object-oriented federated database system. In T. M. Vijayaraman, Alejandro P. Buchmann, C. Mohan, and Nandlal L. Sarda, editors, *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, September 3-6, 1996, Mumbai (Bombay), India*, pages 378–389. Morgan Kaufmann, 1996.

[Has96] W. Hasan. *Optimization of SQL Queries for Parallel Machines*. PhD thesis, Stanford Univeristy, 1996.

[HCY94] Hui-I Hsiao, Ming-Syan Chen, and Philip S. Yu. On parallel execution of multiple pipelined hash joins. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994.*, pages 185–196. ACM Press, 1994.

[HCY97] Hui-I Hsiao, Ming-Syan Chen, and Philip S. Yu. Parallel execution of hash joins in parallel databases. *IEEE Trans. Parallel Distrib. Syst.*, 8(8):872–883, 1997.

[HFC+00] Joseph M. Hellerstein, Michael J. Franklin, Sirish Chandrasekaran, Amol Deshpande, Kris Hildrum, Samuel Madden, Vijayshankar Raman, and Mehul A. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Eng. Bull.*, 23(2):7–18, 2000.

[HH99] Peter J. Haas and Joseph M. Hellerstein. Ripple joins for online aggregation. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA.*, pages 287–298. ACM Press, 1999.

[HM95] Abdelkader Hameurlain and Franck Morvan. Scheduling and mapping

for parallel execution of extended SQL queries. In *CIKM '95, Proceedings of the 1995 International Conference on Information and Knowledge Management, November 28 - December 2, 1995, Baltimore, Maryland, USA*, pages 197–204. ACM, 1995.

[HM02] Abdelkader Hameurlain and Franck Morvan. CPU and incremental memory allocation in dynamic parallelization of SQL queries. *Parallel Computing*, 28(4):525–556, 2002.

[Hon92] Wei Hong. Exploiting inter-operation parallelism in XPRS. In Michael Stonebraker, editor, *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, June 2-5, 1992.*, pages 19–28. ACM Press, 1992.

[HS91] Wei Hong and Michael Stonebraker. Optimization of parallel query execution plans in XPRS. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems (PDIS 1991), Fontainebleu Hilton Resort, Miami Beach, Florida, December 4-6, 1991*, pages 218–225. IEEE Computer Society, 1991.

[Hsi92] David K. Hsiao. Federated databases and systems: Part I - a tutorial on their data sharing. *VLDB J.*, 1(1):127–179, 1992.

[IC91] Yannis E. Ioannidis and Stavros Christodoulakis. On the propagation of errors in the size of join results. In James Clifford and Roger King, editors, *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, Denver, Colorado, May 29-31, 1991.*, pages 268–277. ACM Press, 1991.

[IFF+99] Zachary G. Ives, Daniela Florescu, Marc Friedman, Alon Y. Levy, and Daniel S. Weld. An adaptive query execution system for data integration. In Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA.*, pages 299–310. ACM Press, 1999.

[IHW02] Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. An XML query engine for network-bound data. *VLDB J.*, 11(4):380–402, 2002.

[IHW04] Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. Adapting to source properties in processing data integration queries. In Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 395–406. ACM, 2004.

[ILW+00] Zachary G. Ives, Alon Y. Levy, Daniel S. Weld, Daniela Florescu, and Marc Friedman. Adaptive query processing for internet applications. *IEEE Data Eng. Bull.*, 23(2):19–26, 2000.

[Ioa96] Yannis E. Ioannidis. Query optimization. *ACM Comput. Surv.*, 28(1):121–123, 1996.

[Ive02] Z. Ives. *Efficient Query Processing for Data Integration*. PhD thesis, University of Washington, 2002.

[JSHL02] Vanja Josifovski, Peter M. Schwarz, Laura M. Haas, and Eileen Lin. Garlic: a new flavor of federated query processing for DB2. In Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, pages 524–532. ACM, 2002.

[KA99] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1):41–50, 2003.

[KCC+03] Sailesh Krishnamurthy, Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Samuel Madden, Fred Reiss, and Mehul A. Shah. Telegraphcq: An architectural status report. *IEEE Data Eng. Bull.*, 26(1):11–18, 2003.

[KD98] Navin Kabra and David J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 106–117. ACM Press, 1998.

[Ken03] K. Kennedy. Languages, compilers and run-time systems. In Ian Foster and Carl Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure"*. Morgan Kaufmann Publishers, 2003.

[Kos00] Donald Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 32(4):422–469, 2000.

[KS00] Donald Kossmann and Konrad Stocker. Iterative dynamic programming: a new class of query optimization algorithms. *ACM Trans. Database Syst.*, 25(1):43–82, 2000.

[KTF02] Nicholas T. Karonis, Brian R. Toonen, and Ian T. Foster. Mpich-g2: A grid-enabled implementation of the message passing interface. *CoRR*, 2002.

[LCRY93] Ming-Ling Lo, Ming-Syan Chen, Chinya V. Ravishankar, and Philip S. Yu. On optimal processor allocation to support pipelined hash joins. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993.*, pages 69–78. ACM Press, 1993.

[LEHN02] Gang Luo, Curt Ellmann, Peter J. Haas, and Jeffrey F. Naughton. A scalable hash ripple join algorithm. In Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, pages 252–262. ACM, 2002.

[LFP03] David T. Liu, Michael J. Franklin, and Devesh Parekh. Griddb: A database interface to the grid. In Alon Y. Halevy, Zachary G. Ives, and AnHai Doan, editors, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, page 660. ACM, 2003.

[Liu97a] K. Liu. Performance evaluation of processor allocation algorithms for parallel query execution. In *Proceedings of the 1997 ACM symposium on Applied computing*, pages 393–402. ACM Press, 1997.

[Liu97b] Kevin Hao Liu. Performance study on optimal processor assignment in parallel relational databases. In *International Conference on Supercomputing*, pages 84–91, 1997.

[LNEW04] Gang Luo, Jeffrey F. Naughton, Curt Ellmann, and Michael Watzke. Toward a progress indicator for database queries. In Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 791–802. ACM, 2004.

[LOG92] Hongjun Lu, Beng Chin Ooi, and Cheng Hian Goh. On global multi-database query optimization. *SIGMOD Record*, 21(4):6–11, 1992.

[Man01] Ioana Manolescu. *Optimization Techniques for querying heterogeneous distributed data sources*. PhD thesis, Universite de Versailles, France, 2001.

[MBGS03] Tobias Mayr, Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Leveraging non-uniform resources for parallel query processing. In *3rd IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2003), 12-15 May 2003, Tokyo, Japan*, pages 120–129. IEEE Computer Society, 2003.

[MBM+99] R. W. Moore, C. Baru, R. Marciano, A. Rajasekar, and M. Wan. Data-Intensive Computing. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastrcuture*, chapter 5, pages 105–129. Morgan Kaufmann, 1999.

[ML86] Lothar F. Mackert and Guy M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi, editors, *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings.*, pages 149–159. Morgan Kaufmann, 1986.

[Mol93] D. Moldovan. *Parallel Computing: From Applications to Systems*. Morgan Kaufmann Publishers Inc., 1993.

[MP03] Peter McBrien and Alexandra Poulovassilis. Data integration by bi-directional schema transformation rules. In Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors, *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 227–238. IEEE Computer Society, 2003.

[MRS+04] Volker Markl, Vijayshankar Raman, David E. Simmen, Guy M. Lohman, and Hamid Pirahesh. Robust query processing through progressive optimization. In Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 659–670. ACM, 2004.

[MSBT03] Tanu Malik, Alexander S. Szalay, Tamas Budavari, and Ani Thakar. Skyquery: A web service approach to federate databases. In *CIDR*, 2003.

[MSHR02] Samuel Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, pages 49–60. ACM, 2002.

[MWA+03] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Singh Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[MWK98] Stefan Manegold, Florian Waas, and Martin L. Kersten. On optimal pipeline processing in parallel query execution. Technical Report INS-R9805, Centrum voor Wiskunde en Informatica (CWI), 1998.

[NCK+03] Sivaramakrishnan Narayanan, Umit V. Catalyrek, Tahsin M. Kurc, Xi Zhang, and Joel H. Saltz. Applying database support for large scale data driven science in distributed environemnts. In *Proc. of the 4th Workshop on Grid Computing, GRID'03*, 2003.

[NSGS+05] María A. Nieto-Santisteban, Jim Gray, Alexander S. Szalay, James Annis, Aniruddha R. Thakar, and William O'Mullane. When database systems meet the grid. In *CIDR*, pages 154–161, 2005.

[NWM98] K. Ng, Z. Wang, and R. Muntz. Dynamic reconfiguration of sub-optimal

parallel query execution plans. Technical Report CSD-980033, UCLA, 1998.

[NWMN99] Kenneth W. Ng, Zhenghao Wang, Richard R. Muntz, and Silvia Nittel. Dynamic query re-optimization. In *SSDBM*, pages 264–273, 1999.

[OB04] Mourad Ouzzani and Athman Bouguettaya. Query processing and optimization on the web. *Distributed and Parallel Databases*, 15(3):187–218, 2004.

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

[ONK+96] Fatma Ozcan, Sena Nural, Pinar Koksal, Cem Evrendilek, and Asuman Dogac. Dynamic query optimization on a distributed object management platform. In *CIKM '96, Proceedings of the Fifth International Conference on Information and Knowledge Management, November 12 - 16, 1996, Rockville, Maryland, USA*, pages 117–124. ACM, 1996.

[ONK+97] Fatma Ozcan, Sena Nural, Pinar Koksal, Cem Evrendilek, and Asuman Dogac. Dynamic query optimization in multidatabases. *IEEE Data Eng. Bull.*, 20(3):38–45, 1997.

[Ora01] A. Oram. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly, 2001.

[OV99] M.T. Ozsu and P. Valduriez, editors. *Principles of Distributed Database Systems (Second Edition)*. Prentice-Hall, 1999.

[PBD+01] Antoine Petitet, Susan Blackford, Jack Dongarra, Brett Ellis, Graham Fagg, Kenneth Roche, and Sathish Vadhiyar. Numerical libraries and the grid. *International Journal of High Performance Applications and Supercomputing*, 15(4):359–374, 2001.

[PCL93a] HweeHwa Pang, Michael J. Carey, and Miron Livny. Memory-adaptive external sorting. In Rakesh Agrawal, Seán Baker, and David A. Bell, editors, *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings.*, pages 618–629. Morgan Kaufmann, 1993.

[PCL93b] HweeHwa Pang, Michael J. Carey, and Miron Livny. Partially preemptive hash joins. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993.*, pages 59–68. ACM Press, 1993.

[PLP02] Henrique Paques, Ling Liu, and Calton Pu. Ginga: a self-adaptive query processing system. In *Proceedings of the 2002 ACM CIKM International Conference on Information and Knowledge Management, McLean, VA, USA, November 4-9, 2002*, pages 655–658. ACM, 2002.

[POL] The polar* project, www.ncl.ac.uk/polarstar.

[PS00] Beth Plale and Karsten Schwan. dquob: Managing large data flows using dynamic embedded queries. In *HPDC*, pages 263–270, 2000.

[PS01] Beth Plale and Karsten Schwan. Optimizations enabled by relational data model view to querying data streams. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01), San Francisco, CA, April 23-27, 2001*, page 20. IEEE Computer Society, 2001.

[RDH03] Vijayshankar Raman, Amol Deshpande, and Joseph M. Hellerstein. Using state modules for adaptive query processing. In Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors, *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 353–364. IEEE Computer Society, 2003.

[RH02] Vijayshankar Raman and Joseph M. Hellerstein. Partial results for online query processing. In Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors, *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*, pages 275–286. ACM, 2002.

[RM95] Erhard Rahm and Robert Marek. Dynamic multi-resource load balancing in parallel database systems. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland.*, pages 395–406. Morgan Kaufmann, 1995.

[RNvGJ01] Andrei Radulescu, Cristina Nicolescu, Arjan J. C. van Gemund, and Pieter Jonker. Cpr: Mixed task and data parallel scheduling for distributed systems. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium (IPDPS-01), San Francisco, CA, April 23-27, 2001*, page 39. IEEE Computer Society, 2001.

[ROH99] Mary Tork Roth, Fatma Ozcan, and Laura M. Haas. Cost models do matter: Providing cost information for diverse data sources in a federated system. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 599–610. Morgan Kaufmann, 1999.

[RRH99] Vijayshankar Raman, Bhaskaran Raman, and Joseph M. Hellerstein. Online dynamic reordering for interactive data processing. In Malcolm P. Atkinson, Maria E. Orlowska, Patrick Valduriez, Stanley B. Zdonik, and Michael L. Brodie, editors, *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 709–720. Morgan Kaufmann, 1999.

[RRH00] Vijayshankar Raman, Bhaskaran Raman, and Joseph M. Hellerstein. Online dynamic reordering. *VLDB J.*, 9(3):247–260, 2000.

[RvG01] Andrei Radulescu and Arjan J. C. van Gemund. A low-cost approach towards mixed task and data parallel scheduling. In Lionel M. Ni and Mateo Valero, editors, *Proceedings of the 2001 International Conference on Parallel Processing, ICPP 2002, 3-7 September 2001, Valencia, Spain*, pages 69–76. IEEE Computer Society, 2001.

[RZL02] Amira Rahal, Qiang Zhu, and Per-Åke Larson. Developing evolutionary cost models for query optimization in a dynamic multidatabase environment. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems, 2002 - DOA/CoopIS/ODBASE 2002 Confederated International Conferences DOA, CoopIS and ODBASE 2002 Irvine, California, USA, October 30 - November 1, 2002, Proceedings*, pages 1–18. Springer, 2002.

[RZL04] Amira Rahal, Qiang Zhu, and Per-Åke Larson. Evolutionary techniques for updating query cost models in a dynamic multidatabase environment. *VLDB J.*, 13(2):162–176, 2004.

[SA02] Etzard Stolte and Gustavo Alonso. Optimizing scientific databases for client side data processing. In Christian S. Jensen, Keith G. Jeffery, Jaroslav Pokorný, Simonas Saltenis, Elisa Bertino, Klemens Böhm, and Matthias Jarke, editors, *Advances in Database Technology - EDBT 2002, 8th International Conference on Extending Database Technology, Prague, Czech Republic, March 25-27, Proceedings*, pages 390–408. Springer, 2002.

[SAC+79] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In Philip A. Bernstein, editor, *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 30 - June 1*, pages 23–34. ACM, 1979.

[SAL+96] Michael Stonebraker, Paul M. Aoki, Witold Litwin, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: A wide-area distributed database system. *VLDB J.*, 5(1):48–63, 1996.

[SCS+04] Sameer Shivle, Ralph H. Castain, Howard Jay Siegel, Anthony A. Maciejewski, Tarun Banka, Kiran Chindam, Steve Dussinger, Prakash Pichumani, Praveen Satyasekaran, William Saylor, David Sendek, J. Sousa, Jayashree Sridharan, Prasanna Sugavanam, and Jose Velazco. Static mapping of subtasks in a heterogeneous ad hoc grid environment. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004), CD-ROM / Abstracts Proceedings, 26-30 April 2004, Santa Fe, New Mexico, USA*. IEEE Computer Society, 2004.

[SGW+02] Jim Smith, Anastasios Gounaris, Paul Watson, Norman W. Paton, Alvaro A. A. Fernandes, and Rizos Sakellariou. Distributed query processing on the grid. In Manish Parashar, editor, *Grid Computing - GRID 2002, Third International Workshop, Baltimore, MD, USA, November 18, 2002, Proceedings*, pages 279–290. Springer, 2002.

[SGW+03] Jim Smith, Anastasios Gounaris, Paul Watson, Norman. W. Paton, Alvaro. A. A. Fernandes, and Rizos Sakellariou. Distributed query processing on the grid. *International Journal of High Performance Computing Applications*, 17(4):353–367, 2003.

[SHCF03] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In Umeshwar Dayal, Krithi Ramamritham, and T. M. Vijayaraman, editors, *Proceedings of the 19th International Conference on Data Engineering, March 5-8, 2003, Bangalore, India*, pages 25–36. IEEE Computer Society, 2003.

[SLMK01] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. LEO - DB2's learning optimizer. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 19–28. Morgan Kaufmann, 2001.

[SOHL+98] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI - The Complete Reference*. The MIT Press, Cambridge, Massachusetts, 1998. ISBN: 0-262-69215-5.

[SPSW02] Sandra Sampaio, Norman W. Paton, Jim Smith, and Paul Watson. Validated cost models for parallel oql query processing. In Zohra Bellahsene, Dilip Patel, and Colette Rolland, editors, *Object-Oriented Information Systems, 8th International Conference, OOIS 2002, Montpellier, France, September 2-5, 2002, Proceedings*, pages 60–75. Springer, 2002.

[SPWS99] Sandra Sampaio, Norman W. Paton, Paul Watson, and Jim Smith. A parallel algebra for object databases. In *DEXA Workshop*, pages 56–60, 1999.

[SSW04] Balasubramanian Seshasayee, Karsten Schwan, and Patrick Widener. Soap-binq: High-performance soap with continuous quality management. In *ICDCS*, pages 158–165, 2004.

[SSWP04] Jim Smith, Sandra Sampaio, Paul Watson, and Norman W. Paton. The design, implementation and evaluation of an odmg compliant, parallel object database server. *Distributed and Parallel Databases*, 16(3):275–319, 2004.

[Sto86] M. Stonebraker. The case for shared nothing. *IEEE Data Engineering Bulletin*, 9(1):4–9, 1986.

[SW04] Jim Smith and Paul Watson. Applying low-overhead rollback-recovery to wide area distributed query processing. Technical Report CS-TR-861, The University of Newcastle upon Tyne, School of Computing, 2004.

[SWKH76] Michael Stonebraker, Eugene Wong, Peter Kreps, and Gerald Held. The design and implementation of ingres. *ACM Trans. Database Syst.*, 1(3):189–222, 1976.

[SZ04] Rizos Sakellariou and Henan Zhao. A hybrid heuristic for DAG scheduling on heterogeneous systems. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004), CD-ROM / Abstracts Proceedings, 26-30 April 2004, Santa Fe, New Mexico, USA*. IEEE Computer Society, 2004.

[TB02] Wee Hyong Tok and Stéphane Bressan. Efficient and adaptive processing of multiple continuous queries. In Christian S. Jensen, Keith G. Jeffery, Jaroslav Pokorný, Simonas Saltenis, Elisa Bertino, Klemens Böhm, and Matthias Jarke, editors, *Advances in Database Technology - EDBT 2002, 8th International Conference on Extending Database Technology, Prague, Czech Republic, March 25-27, Proceedings*, pages 215–232. Springer, 2002.

[TCF+02] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, and C. Kesselman. Grid Service Specification. Technical report, Open Grid Service Infrastructure WG, Global Grid Forum, 2002. Draft 5, November 5, 2002.

[TD03] Feng Tian and David J. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB*, pages 333–344, 2003.

[TTC+90] Gomer Thomas, Glenn R. Thompson, Chin-Wan Chung, Edward Barkmeyer, Fred Carter, Marjorie Templeton, Stephen Fox, and Berl Hartman. Heterogeneous distributed database systems for production use. *ACM Comput. Surv.*, 22(3):237–266, 1990.

[TTL03] Douglas Thain, Todd Tannenbaum, and Miron Livny. Condor and the grid. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Inc., 2003.

[TWML02] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. Condor – a distributed job scheduler. In Thomas Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, 2002.

[UF00] Tolga Urhan and Michael J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Eng. Bull.*, 23(2):27–33, 2000.

[UF01] Tolga Urhan and Michael J. Franklin. Dynamic pipeline scheduling for improving interactive query performance. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 501–510. Morgan Kaufmann, 2001.

[UFA98] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost based query scrambling for initial delays. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA.*, pages 130–141. ACM Press, 1998.

[Wat03] Paul Watson. Databases and the grid. In Fran Berman, Geoffrey Fox, and Tony Hey, editors, *Grid Computing: Making The Global Infrastructure a Reality*. John Wiley & Sons Inc., 2003.

[WC96] Chihping Wang and Ming-Syan Chen. On the complexity of distributed query optimization. *IEEE Trans. Knowl. Data Eng.*, 8(4):650–662, 1996.

[WCwHK04] Jun Wu, Jian-Jia Chen, Chih wen Hsueh, and Tei-Wei Kuo. Scheduling of query execution plans in symmetric multiprocessor database systems. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004), CD-ROM / Abstracts Proceedings, 26-30 April 2004, Santa Fe, New Mexico, USA*. IEEE Computer Society, 2004.

[WFA92] Annita N. Wilschut, Jan Flokstra, and Peter M. G. Apers. Parallelism in a main-memory dbms: The performance of prisma/db. In Li-Yan Yuan, editor, *18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver*, pages 521–532. Morgan Kaufmann, 1992.

[WFA95] Annita N. Wilschut, Jan Flokstra, and Peter M. G. Apers. Parallel evaluation of multi-join queries. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, pages 115–126. ACM Press, 1995.

[WHW+04] Steve R. White, James E. Hanson, Ian Whalley, David M. Chess, and Jeffrey O. Kephart. An architectural approach to autonomic computing. In *1st International Conference on Autonomic Computing (ICAC 2004), 17-19 May 2004, New York, NY, USA*, pages 2–9. IEEE Computer Society, 2004.

[WSH99] Rich Wolski, Neil T. Spring, and Jim Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Generation Computer Systems*, 15(5–6):757–768, 1999.

[YD02] Asim YarKhan and Jack Dongarra. Experiments with scheduling using simulated annealing in a grid environment. In Manish Parashar, editor, *Grid Computing - GRID 2002, Third International Workshop, Baltimore, MD, USA, November 18, 2002, Proceedings*, pages 232–242. Springer, 2002.

[YS97] Min J. Yu and Phillip C.-Y. Sheu. Adaptive join algorithms in dynamic distributed databases. *Distributed and Parallel Databases*, 5(1):5–30, 1997.

[Zho03] Yongluan Zhou. Adaptive distributed query processing. In Marc H. Scholl and Torsten Grust, editors, *Proceedings of the VLDB 2003 PhD Workshop. Co-located with the 29th International Conference on Very Large Data Bases (VLDB 2003). Berlin, September 12-13, 2003*. Technical University of Aachen (RWTH), 2003.

[ZL97] Weiye Zhang and Per-Åke Larson. Dynamic memory adjustment for external mergesort. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 376–385. Morgan Kaufmann, 1997.

[ZMS03] Qiang Zhu, Satyanarayana Motheramgari, and Yu Sun. Cost estimation for queries experiencing multiple contention states in dynamic multidatabase environments. *Knowl. Inf. Syst.*, 5(1):26–49, 2003.

[ZOTT05] Yongluan Zhou, Beng Chin Ooi, Kian-Lee Tan, and Wee Hyong Tok. An adaptable distributed query processing architecture. *Data Knowl. Eng.*, To appear, 2005.

[ZRH04] Yali Zhu, Elke A. Rundensteiner, and George T. Heineman. Dynamic plan migration for continuous queries over data streams. In Gerhard Weikum, Arnd Christian König, and Stefan Deßloch, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, pages 431–442. ACM, 2004.

[ZSM00] Qiang Zhu, Yu Sun, and Satyanarayana Motheramgari. Developing cost models with qualitative variables for dynamic multidatabase environments. In *ICDE*, pages 413–424, 2000.

Appendix A

A Simplified Cost Model

The cost model in [SPSW02], which is a detailed and validated one developed for parallel object database systems [SSWP04], has been simplified and adapted to operate in a distributed and autonomous environment and has been incorporated in the Polar* query engine. This model estimates the query completion time, by estimating the cost of each operator instance separately in time units. This cost is further decomposed into (i) computation cost, (ii) disk I/O cost, and (iii) communication cost. Here, only the operators employed in the evaluation queries in Section 3.3 will be examined, i.e., *scan, hash join* and *exchange,* as providing formulas for a more complete set of query operators is out of the scope of the work of this thesis.

A.1 Cost of Scan

The scan operator is used to retrieve the data from a store. Its time cost, T_{scan}, according to [SPSW02], is

$$T_{scan} = t_{seek} + t_{rot\_latency} + t_{transfer} \cdot n_{pages} + t_{map} \cdot N + (1 + \frac{n_{cond} - 1}{2}) \cdot t_{eval} \cdot N$$

The meaning of the variables is presented in Table A.1. If the scan is implemented as a *sequential scan* operator, it accesses disk pages sequentially, in order to retrieve all the objects of an extent, which is the equivalent of a database table in the ODMG model. The predicate (if it exists) is applied afterwards. If the scan is implemented as an *indexed scan* and there exists a predicate, the number of pages accessed can be reduced dramatically.

Variable	Description
n_{pages}	number of pages retrieved
S	size of tuples retrieved (in bytes)
N	number of tuples retrieved
n_{cond}	number of conditions in the predicate
t_{seek}	seek time of disk (in time units)
$t_{rot\_latency}$	rotational latency of disk (in time units)
$t_{transfer}$	transfer time of disk (in time units)
t_{map}	time to map an object to tuple format, which depends on the number of attributes in the object and their type (in time units)
t_{eval}	time to evaluate a condition of the predicate (in time units)
I/O_speed	rate of reading data from store (in time units per byte)

Table A.1: The parameters for estimating the time cost of a sequential scan.

The disk timings t_{seek}, $t_{rot\_latency}$, and $t_{transfer}$ are several orders of magnitude larger than the CPU - dependent timings t_{map} and t_{eval}. Polar experiments show that in relatively slow machines, the mapping and predicate evaluation time can be of the same order with disk read time. However, in the Grid, it is expected that the machines used powerful. Hence it can be assumed that the total cost of scans is dominated by the cost to read the data from the disk:

$$T_{scan} \approx t_{transfer} \cdot n_{pages} \approx I/O\_speed \cdot S$$

The size of tuples retrieved is the size of the whole database if a sequential scan is used to retrieve data from a remote database. If that database evaluates the operator using an index-based scan, the size of the tuples retrieved is approximately the size of the tuples that are produced by the scan operator after applying the predicate.

A.2 Cost of Hash Join

The *hash join* consumes the left input and hashes all its tuples on the join attribute, to populate the hash table. After consuming all the left input, for each tuple in the right one, it hashes that tuple on the join attribute, concatenates it with the tuples in the corresponding bucket in the hash table, and applies the predicate.

Variable	Description
N_{left}	number of tuples in the left input
N_{right}	number of tuples in the right input
$n_{buckets}$	number of buckets in the hash table
n_{cond}	number of conditions in the predicate
t_{hash}	time to hash a tuple (in time units)
t_{conc}	time to concatenate a tuple with another tuple (in time units)
t_{eval}	time to evaluate a condition in the predicate (in time units)

Table A.2: The parameters for estimating the time cost of a hash join.

The total cost is (according to [SPSW02]):

$$\begin{aligned} T_{hashjoin} &= t_{hash} \cdot (N_{left} + N_{right}) + t_{conc} \cdot \frac{N_{left}}{n_{buckets}} \cdot N_{right} + \\ &\quad (1 + \frac{n_{cond} - 1}{2}) \cdot t_{eval} \cdot \frac{N_{left}}{n_{buckets}} \cdot N_{right} \end{aligned}$$

The meaning of the variables is presented in Table A.2.

All the variables depend on the CPU power and memory access costs. As the inputs are expected not to be very small, and $t_{hash},\ t_{conc}$,and t_{eval} are expected to be of the same order, $N_{left} + N_{right}$ becomes significantly smaller than $\frac{N_{left}}{n_{buckets}} \cdot N_{right}$. As such, the cost can be approximated to:

$$\begin{aligned} T_{hashjoin} &= t_{conc} \cdot \frac{N_{left}}{n_{buckets}} \cdot N_{right} + (1 + \frac{n_{cond} - 1}{2}) \cdot t_{eval} \cdot \frac{N_{left}}{n_{buckets}} \cdot N_{right} \\ &= t_{hashjoin} \cdot \frac{N_{left}}{n_{buckets}} \cdot N_{right} \end{aligned}$$

where $t_{hashjoin}$ is a variable dependent on the CPU power:

$$t_{hashjoin} = t_{conc} + (1 + \frac{n_{cond} - 1}{2}) \cdot t_{eval}$$

The number of buckets is traditionally the number of distinct values of the join attribute in the left input.

A.3 Cost of Exchange

The cost of the exchange operator consists of the cost to pack tuples into buffers, to unpack them, and to send buffers to consumers. There are two cases: (i) the exchange sends data also to itself, or (ii) the exchange sends data exclusively to remote consumers.

In the first case, the cost of exchange is

$$T_{exchange} = ((t_{pack} + t_{unpack}) \cdot N + \frac{S + overhead \cdot \frac{S}{s_{packet} - overhead}}{ConSpeed}) \cdot (1 - \frac{1}{n_{consumers}})$$

and in the second

$$T_{exchange} = ((t_{pack} + t_{unpack}) \cdot N + \frac{S + overhead \cdot \frac{S}{s_{packet} - overhead}}{ConSpeed}$$

Table A.3 explains the parameters. The difference in the two formulas reflects the fact that if an exchange sends data also to itself, then a fragment of the data need not be transmitted over the network, as it remains on the same node.

Variable	Description
N	number of tuples in the input
S	size of the input (in bytes)
$ConSpeed$	connections speed (in bytes per time unit)
$overhead$	size of network overhead per packet (in bytes)
s_{packet}	size of network packet (in bytes)
$n_{consumers}$	number of consumer nodes
t_{pack}	time to pack a tuple, which depends on the number and types of its attributes (in time units)
t_{unpack}	time to unpack a tuple, depends on the number and types of its attributes (in time units)

Table A.3: The parameters for estimating the time cost of exchange.

Assuming that the communication cost prevails, the CPU cost is not significant, and the overhead is not large, the two formulas become

$$T_{exchange} = (\frac{S}{ConSpeed}) \cdot (1 - \frac{1}{n_{consumers}})$$

and

$$T_{exchange} = \frac{S}{ConSpeed}$$

A.4 Estimating the cost of plan partitions

The cost model accounts for independent, pipelined (or inter-operator), and partitioned (or intra-operator) parallelism in the same way as in [SPSW02]. In summary, the cost of a query plan is the sum of the costs of all plan partitions unless two or more plan partitions are executed independently (e.g., two subtrees rooted from the same node); in the latter case the larger cost among all such partitions suffices. The cost of a plan partition is the sum of the costs of the operators belonging to that partition. If a plan partition is evaluated on many nodes, the most costly subplan instance determines the cost of the plan partition.

Appendix B

Summary of AQP proposals

Technique	Monitoring		Assessment	Response		Architecture		
	Focus	Frequency	Issue	Response Form	Impact	Data Local.	QP Local.	Adapt. Local.
sorting [PCL93a]	resource memory	intra-operator	insufficient/idle memory	operator reconfiguration	operator instance	N/A	central	operator
PPHJ [PCL93b]	resource memory	intra-operator	insufficient/idle memory	operator reconfiguration	operator instance	N/A	any	operator
ripple [HH99]	user input	intra-operator	user requirements	operator reconfiguration	operator instance	N/A	central	operator
XJoin [UF00]	resource connections	intra-operator	idle CPU	operator rescheduling	operator instance	remote	central	operator
juggle [RRH00]	user input	intra-operator	user requirements	operator reconfiguration.	operator instance	N/A	central	operator
mergesort [ZL97]	resource memory	intra-operator	insufficient/idle memory	operator reconfguration	operator instance	N/A	central	central
pipeline scheduler [UF01]	user input, data volume, operator cost	intra-operator	suboptimal operator scheduling, user reqs	operator rescheduling	partition instance	N/A	central	central
mid-query reoptimisation [KD98]	data volume, operator cost	inter-operator	subopt. physical plan, perf. expectations	operator reconf., plan reoptimisation	query plan	local	central	central
progressive optimisation [MRS+04]	data volume, operator cost	inter-operator	subopt. physical plan, perf. expectations	operator reconf., plan reoptimisation	query plan	local	central	central
ingres [SWKH76]	data cardinality	intra-operator	subopt. oper. scheduling	operator rescheduling	query plan	local	central	central
eddies [AH00]	data cardinality, oper. cost	intra-operator	subopt. oper. scheduling	operator rescheduling	query plan	any	central	central

Technique	Monitoring		Assessment	Response		Architecture		
	Focus	Frequency	Issue	Response Form	Impact	Data Local.	QP Local.	Adapt. Local.
stems [RDH03]	data cardinality, oper. cost	intra-operator	subopt. physical plan, operator scheduling	operator rescheduling, replacement	query plan	any	central	central
juggle-eddy [RH02]	data cardinality, operator cost, user input	intra-operator	subopt. oper. scheduling, user reqs	operator rescheduling	query plan	any	central	central
query scrambling [UFA98]	resource connections	inter-operator	idle CPU	operator resched., plan reoptimisation	query plan	remote	central	central
Bouganim [BFMV00b]	resource connections, memory	inter-operator	idle CPU, insufficient memory	operator resched., plan reoptimisation	query plan	remote	central	central
Tukwila [IFF+99, ILW+00, Ive02, IHW04]	resource connections, memory, pool, data volume, operator cost	inter-operator	idle CPU, insufficient memory, subopt. physical plan, perf. expectations	operator rescheduling, reconfiguration, replacement, plan reoptim.	query plan	remote	central	central
bindjoins [Man01]	operator cost	inter-operator	suboptimal resource selection	operator reconfiguration	query plan	remote	central	central
CACQ [MSHR02], psoup [CF03]	data cardinality, operator cost	intra-operator	suboptimal operator scheduling	operator rescheduling	query plan	stream	central	central
dQUOB [PS01]	data characteristics	intra-operator	subopt. oper. scheduling	operator rescheduling	query plan	stream	central	central
chain [BBDM03]	data cardinality	intra-operator	subopt. oper. scheduling	operator rescheduling	query plan	stream	central	central
stream [MWA+03, BMM+04]	data cardinality, resource memory	intra-operator	subopt. oper. scheduling, insuf. memory	operator rescheduling, reconfig.	query plan	stream	central	central
river [ADAT+99]	resource performance	intra-operator	workload imbalance	operator reconfiguration	operator	local	central	central
flux [SHCF03]	resource performance	intra-operator	workload imbalance	operator reconfiguration	operator	local	central	central
parad [HM02]	data cardinality, resource pool, memory	inter-operator	insufficient memory	machine rescheduling	query plan	local	central	central
mind [ONK+97]	data cardinality	inter-operator	subopt. resource selection	machine rescheduling	partition	remote	non-central	central

Technique	Monitoring		Assessment	Response		Architecture		
	Focus	Frequency	Issue	Response Form	Impact	Data Local.	QP Local.	Adapt. Local.
adaptive SDD-1 [YS97]	data cardinality	inter-operator	subopt. operator schedul., resource selection	operator resched., machine resched.	partition	remote	non-central	central
aurora* [CBB+03]	data cardinality	intra-operator	resource shortage	machine rescheduling	operator	stream	non-central	distributed
conquest [NWM98, NWMN99]	resources, data volume	inter-operator	suboptimal physical plan, resource selection, imbalance	plan reoptimisation, machine rescheduling	query plan	remote	non-central	central
SwAP [ZOTT05]	data cardinality	intra-operator	subopt. oper. scheduling	operator rescheduling	query plan	any	non-central	distributed

Table B.1: Summarising table of AQP proposals according to the classifications of Section 4.3.